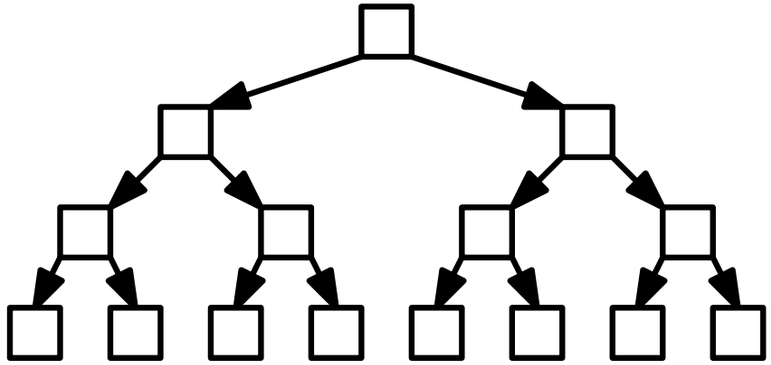
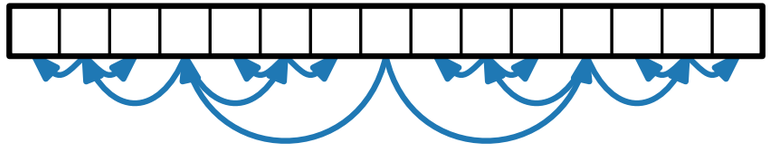
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen

Vorlesung 13:
Binäre Suchbäume





Alexander Wolff

Wintersemester 2025

Zwischentest II: 11.12.2025, 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

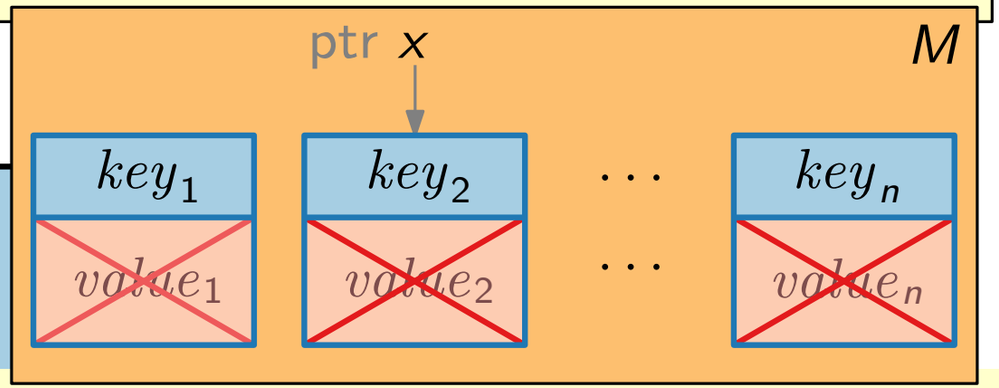
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k)	**Anfragen**
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Wörterbuch

Pointer



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	$\Theta(1)^\star$		–	–

*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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–

*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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum				

*unter bestimmten Annahmen.

Implementierung

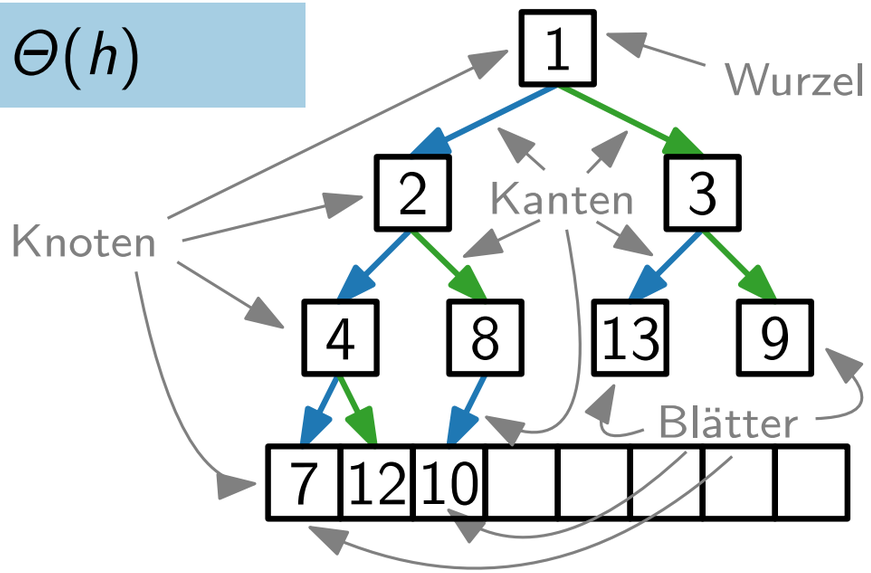
	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	**?**	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

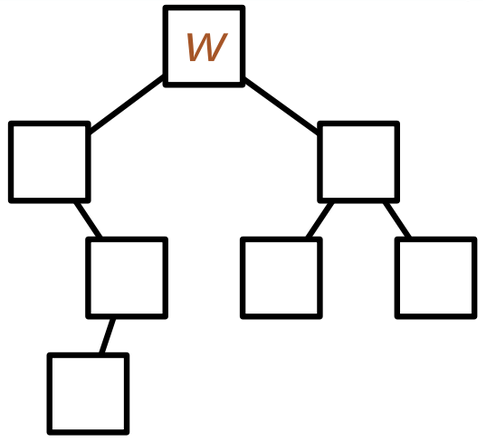
*unter bestimmten Annahmen.

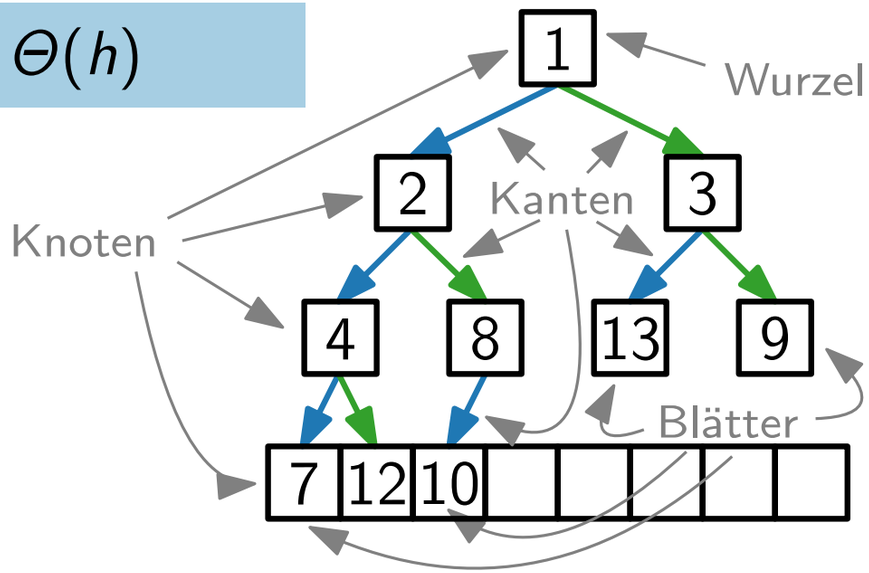


Implementierung

	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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.

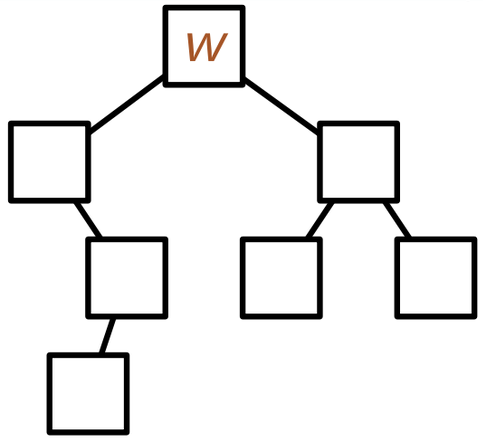




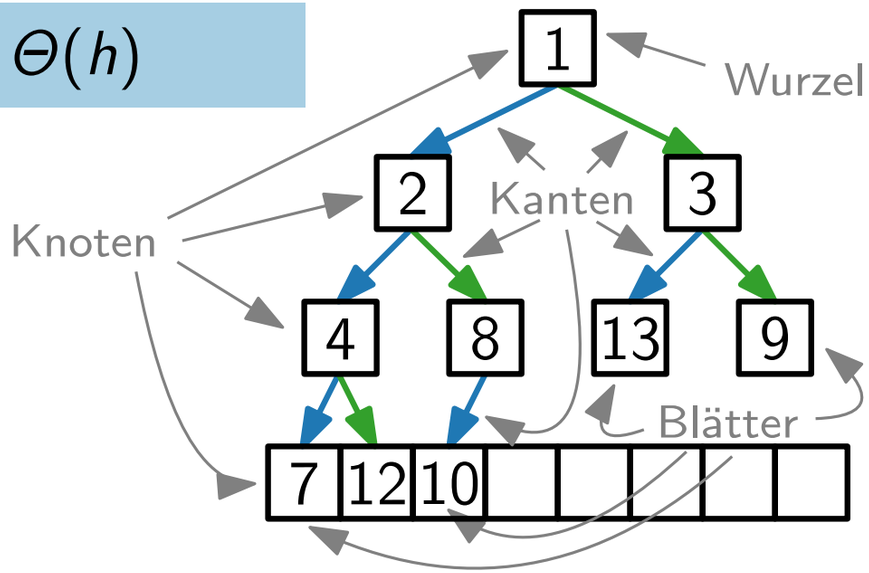
Implementierung

	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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



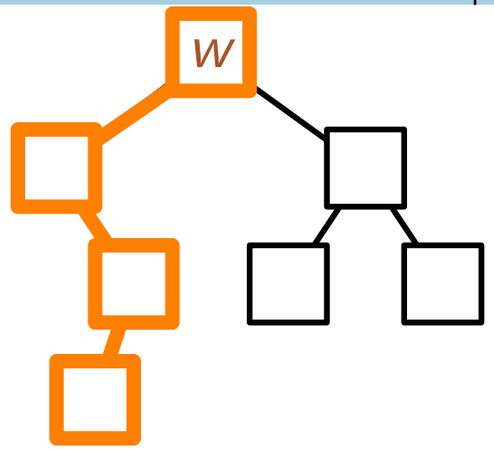
$h(T) =$ Höhe des Baums T



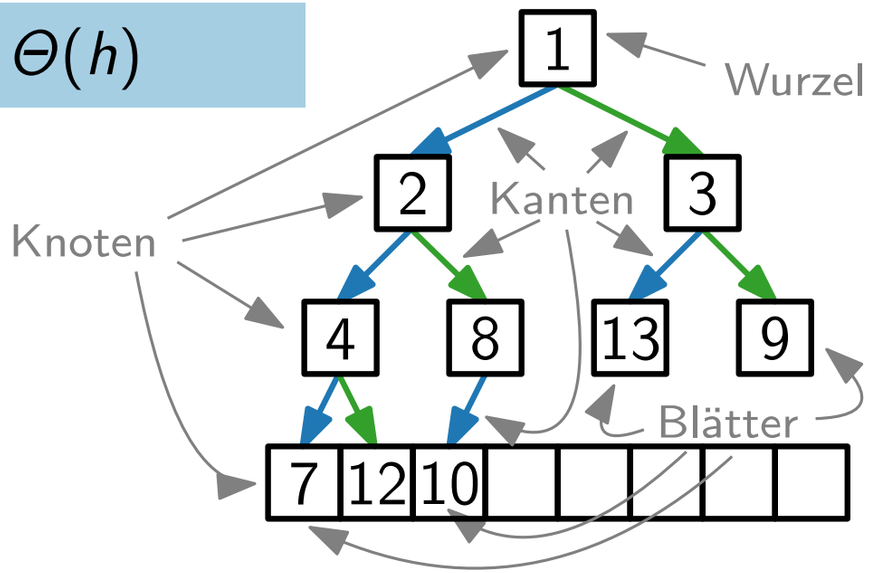
Implementierung

	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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



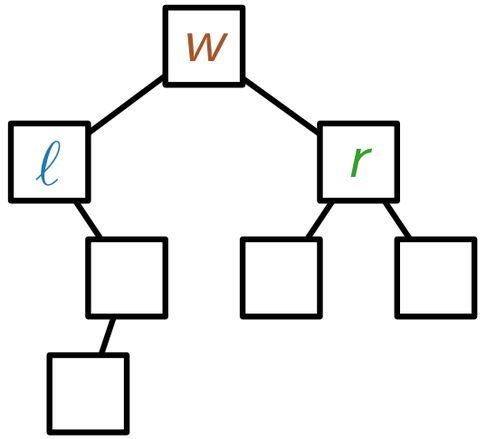
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad



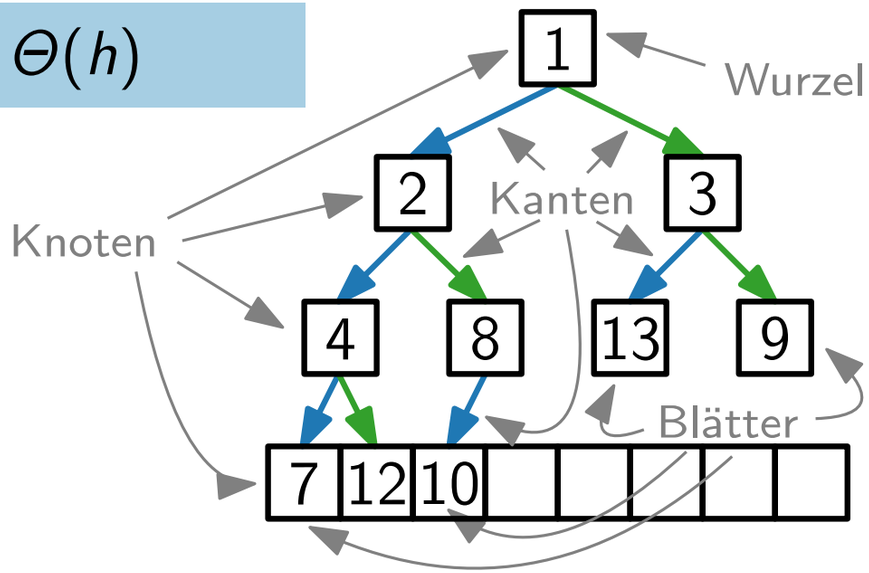
Implementierung

	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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



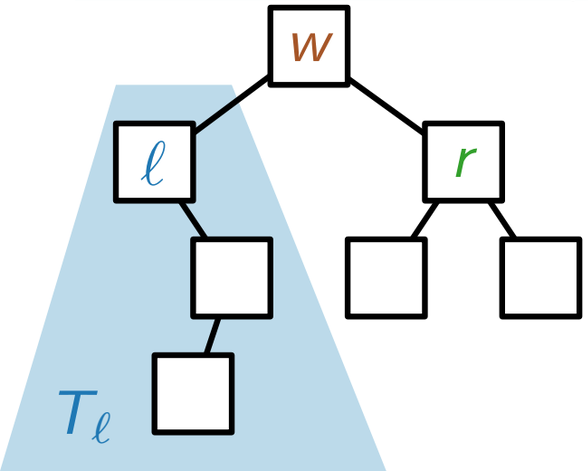
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad



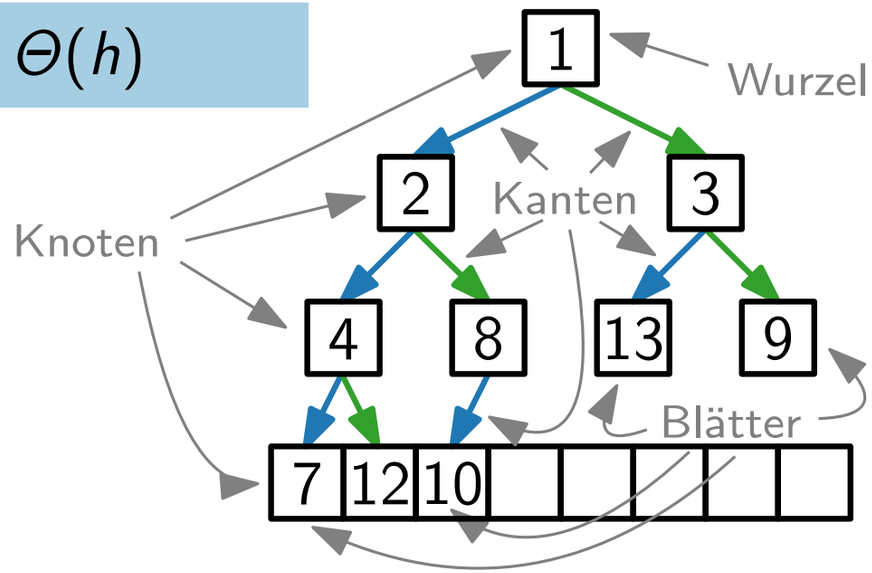
Implementierung

	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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



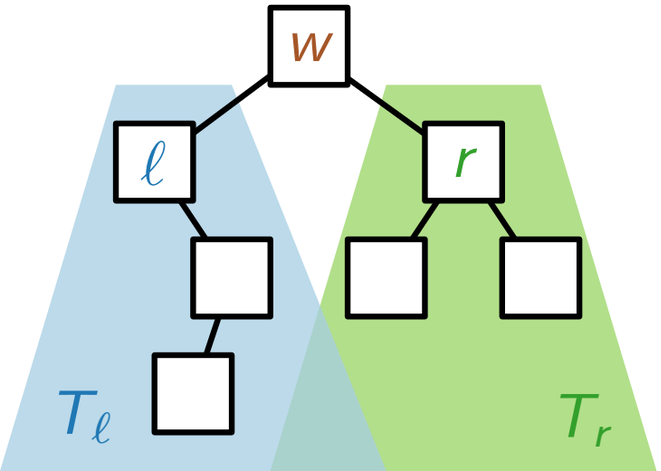
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad



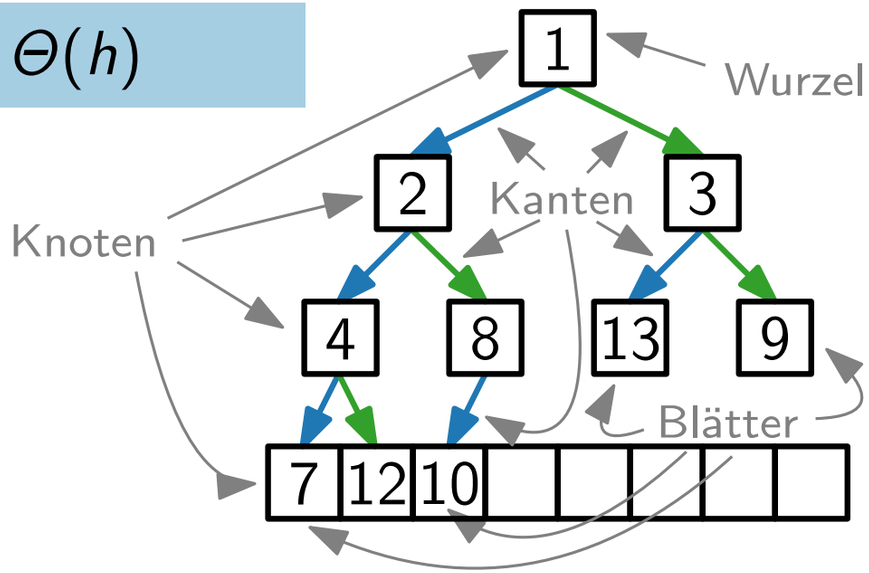
Implementierung

	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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



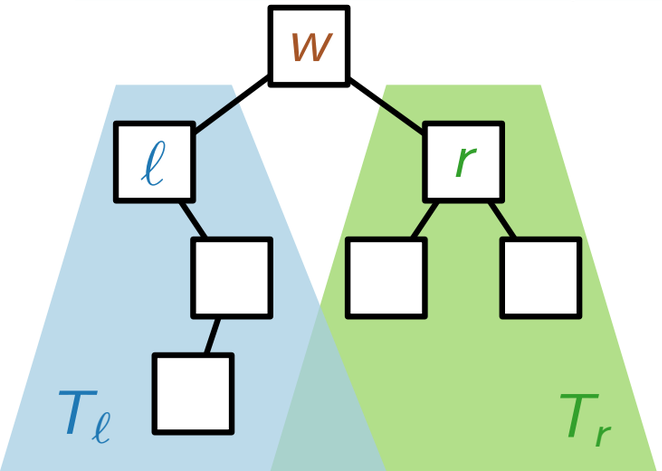
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad



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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MinHeap	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



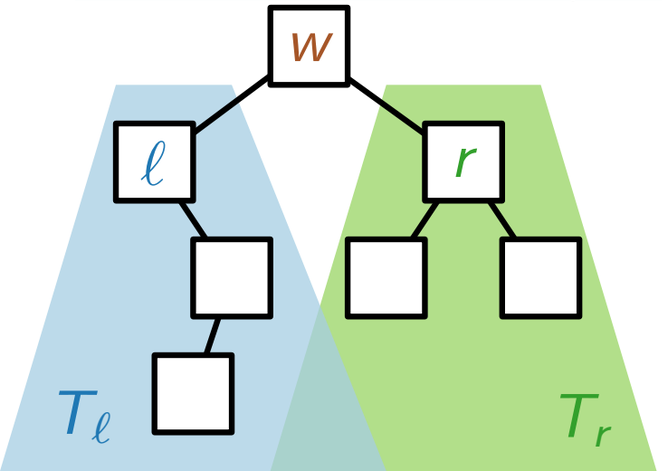
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad

$$= \begin{cases} \textbf{Aufgabe.} \\ \text{Finden Sie die} \\ \text{Rekursionsgleichung!} \end{cases} \quad \begin{array}{l} \text{falls Baum} = \text{Blatt} \\ \text{sonst.} \end{array}$$

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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MINHEAP	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



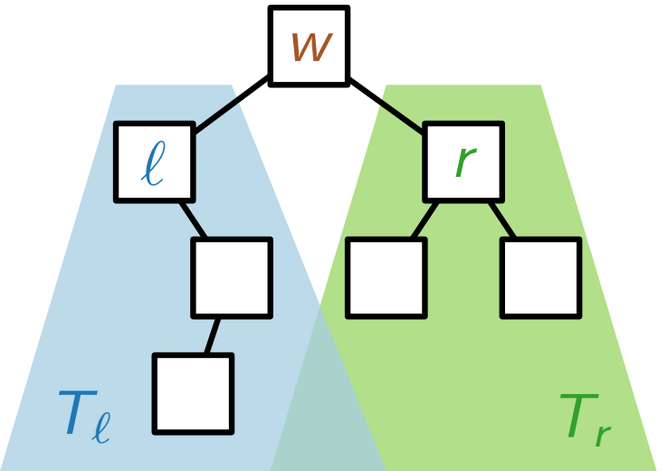
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad

$$= \begin{cases} 0 & \text{falls Baum} = \text{Blatt} \\ & \text{sonst.} \end{cases}$$

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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MINHEAP	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

*unter bestimmten Annahmen.



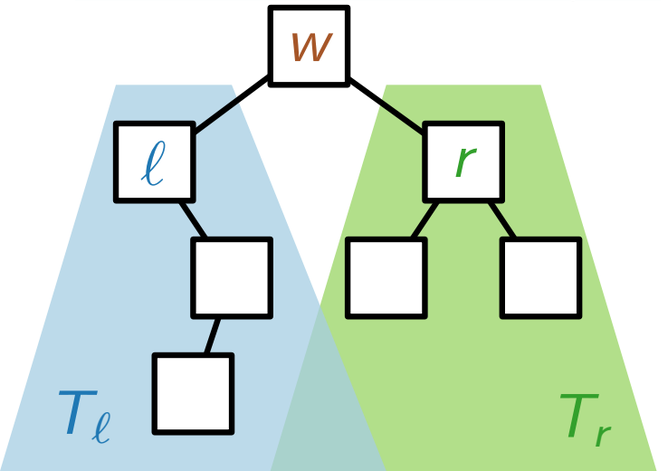
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad

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

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	$\Theta(1)^\star$	$\Theta(1)^\star$	–	–
MINHEAP	–	$\Theta(\log n)$	$\Theta(1)/-$	–
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

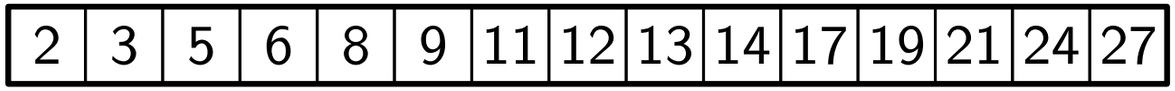
*unter bestimmten Annahmen.



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

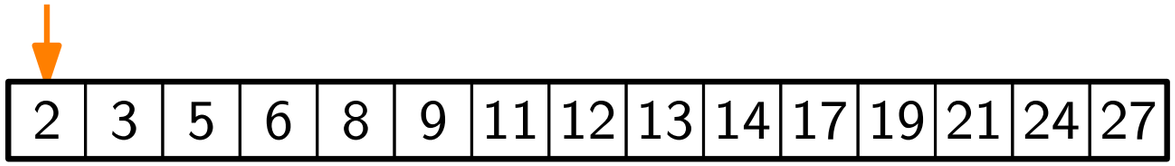
$$= \begin{cases} 0 & \text{falls Baum} = \text{Blatt} \\ 1 + \max\{h(T_\ell), h(T_r)\} & \text{sonst.} \end{cases}$$

Suche im sortierten Feld



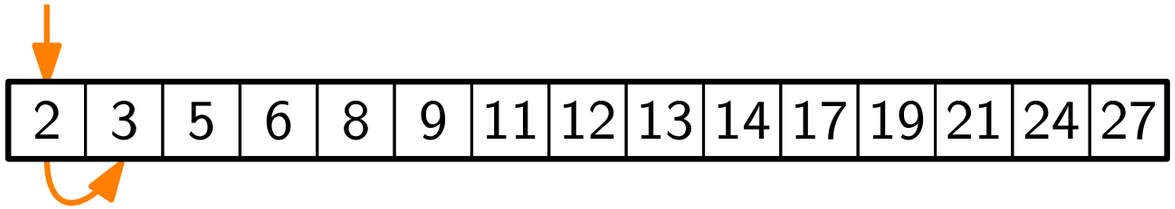
SEARCH(21)

Suche im sortierten Feld



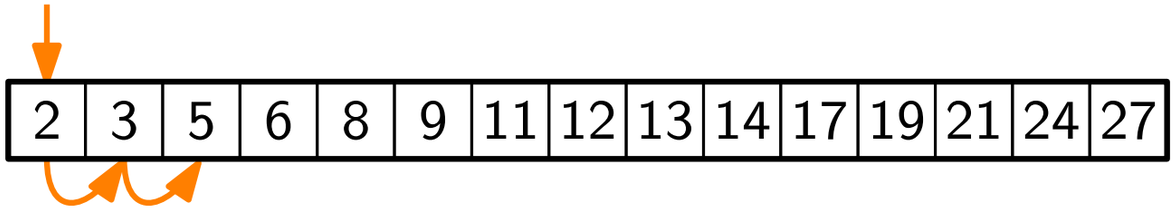
SEARCH(21)

Suche im sortierten Feld



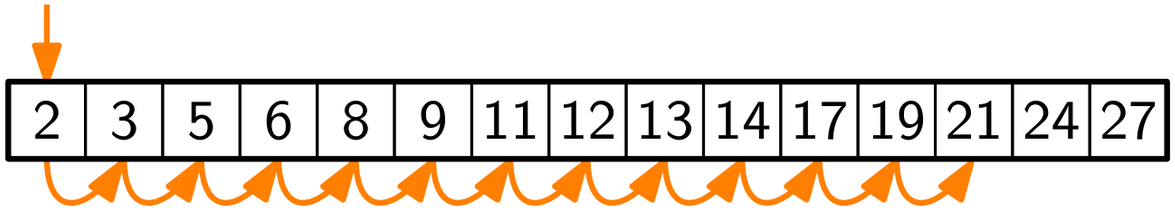
SEARCH(21)

Suche im sortierten Feld



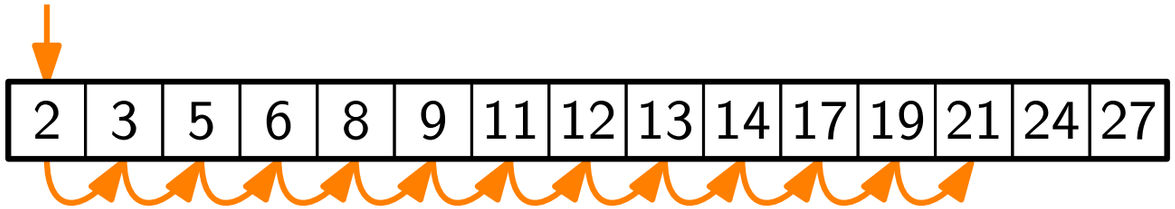
SEARCH(21)

Suche im sortierten Feld



SEARCH(21)

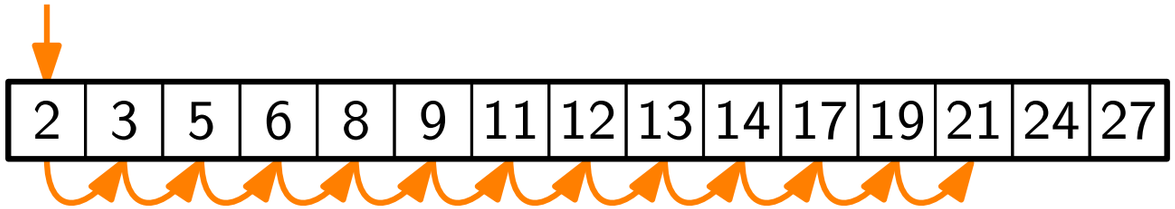
Suche im sortierten Feld



SEARCH(21)

Lineare Suche:

Suche im sortierten Feld

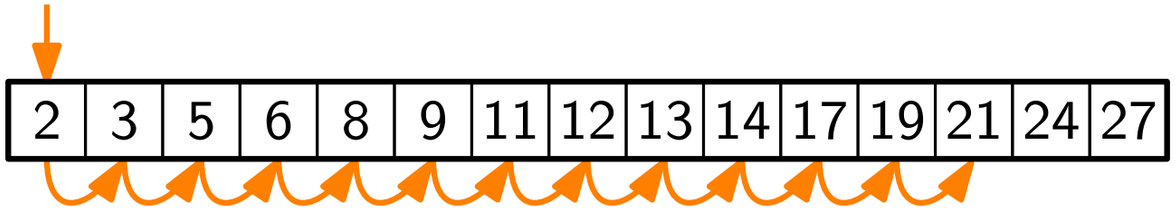


SEARCH(21)

hier

Lineare Suche: 13 Schritte

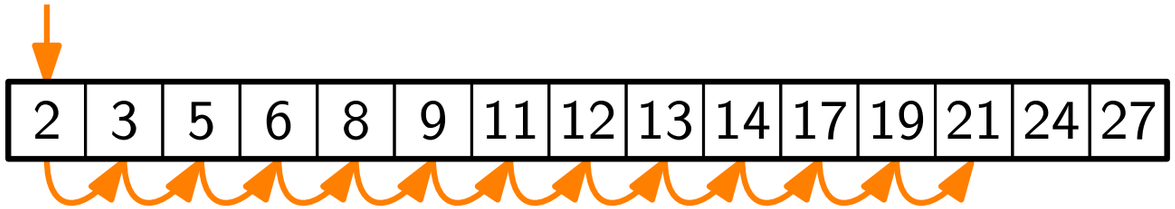
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13		Schritte

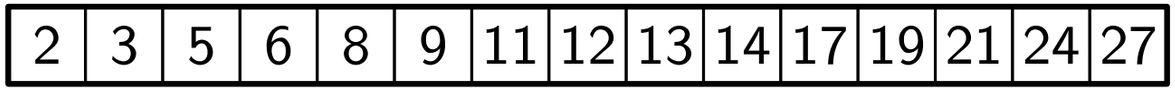
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte

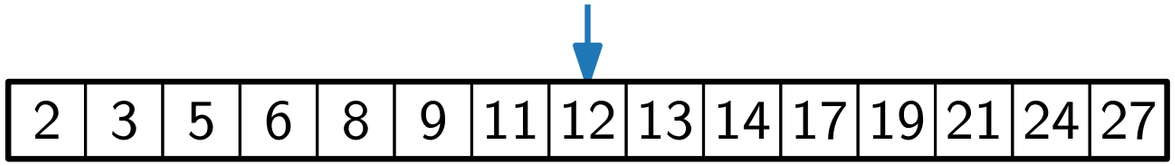
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte

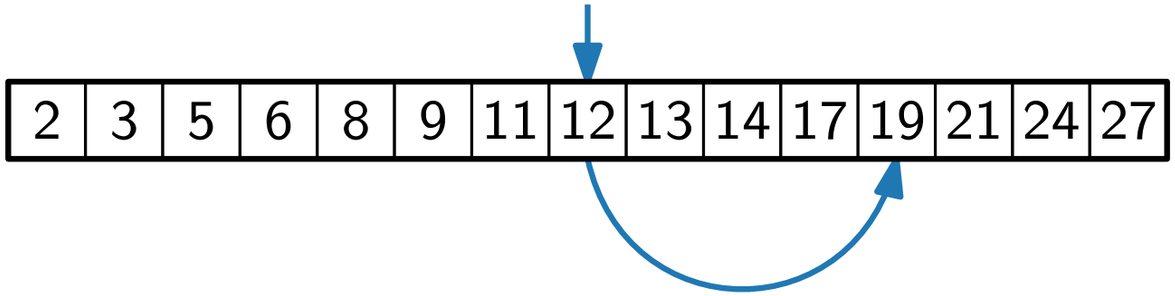
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte

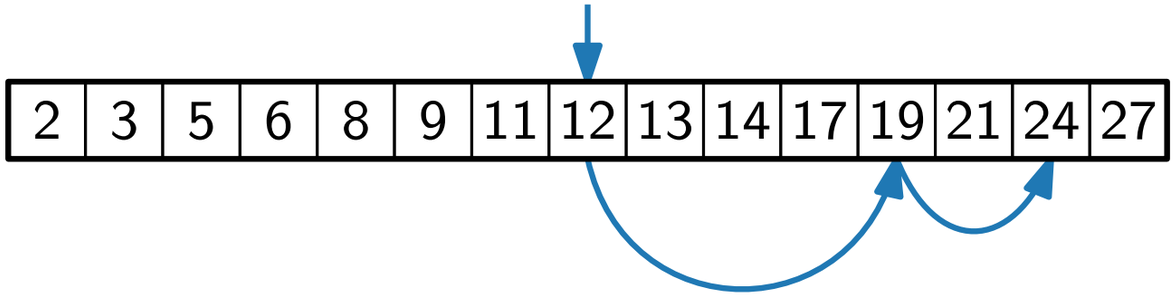
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte

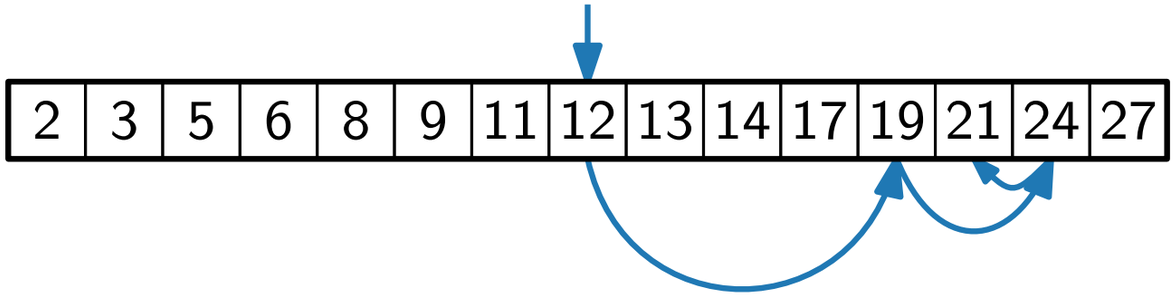
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte

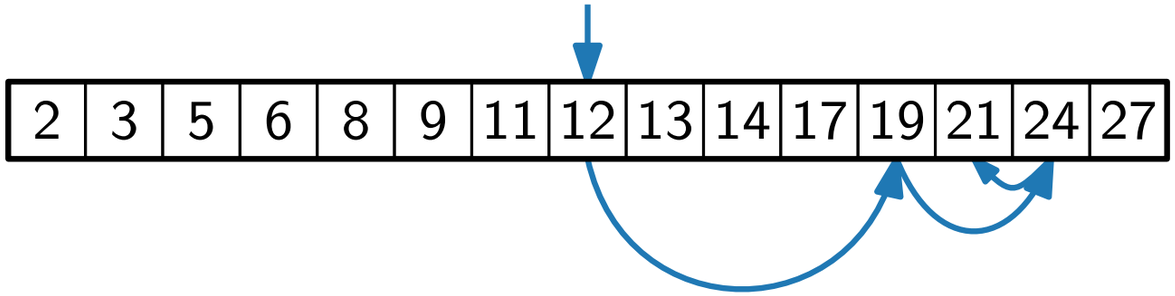
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte

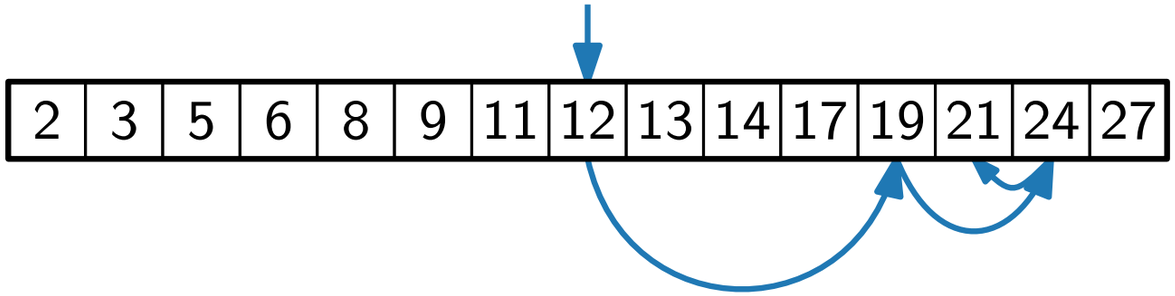
Suche im sortierten Feld



SEARCH(21)

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

Suche im sortierten Feld

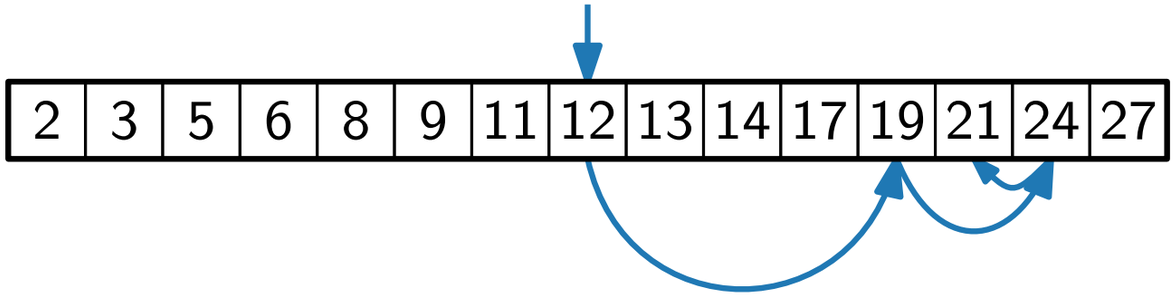


SEARCH(21)

	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

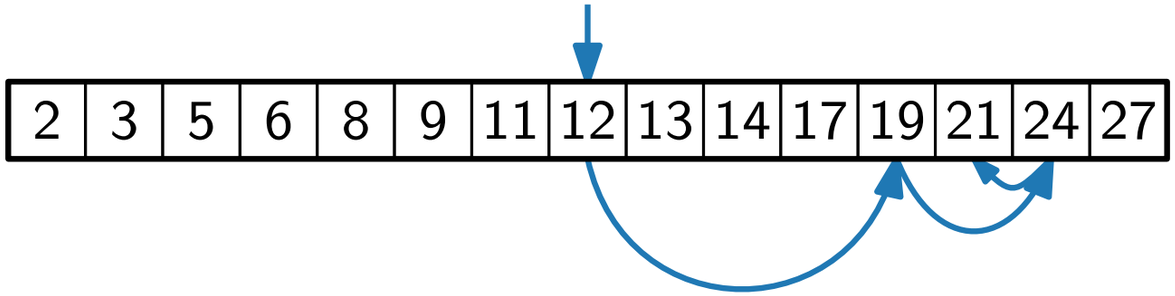


SEARCH(21)

	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



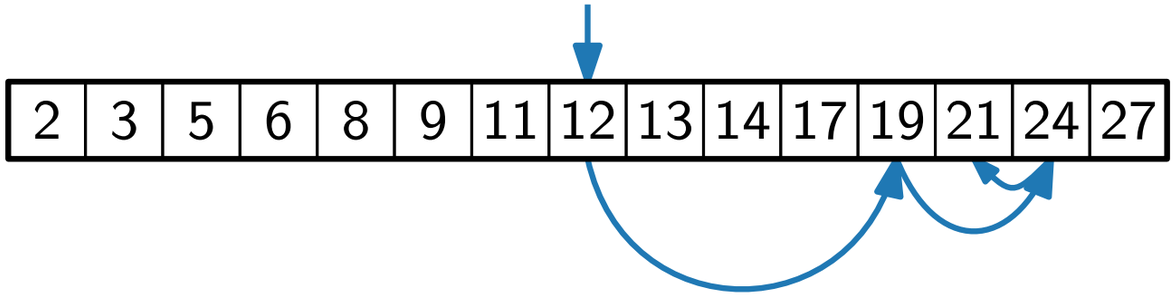
SEARCH(21)

	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

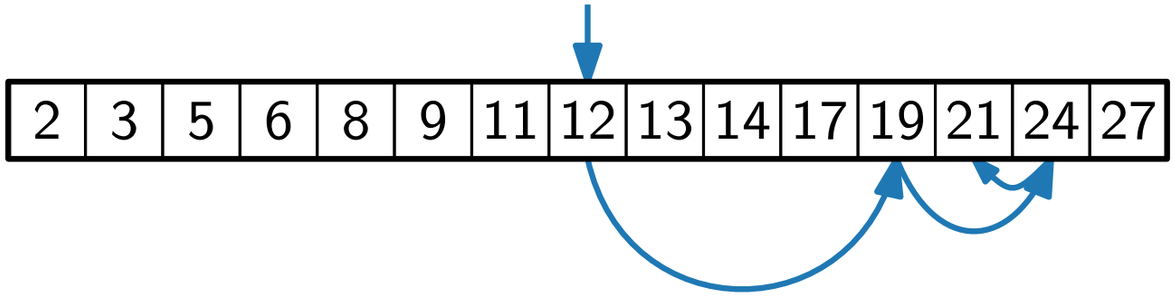
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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)

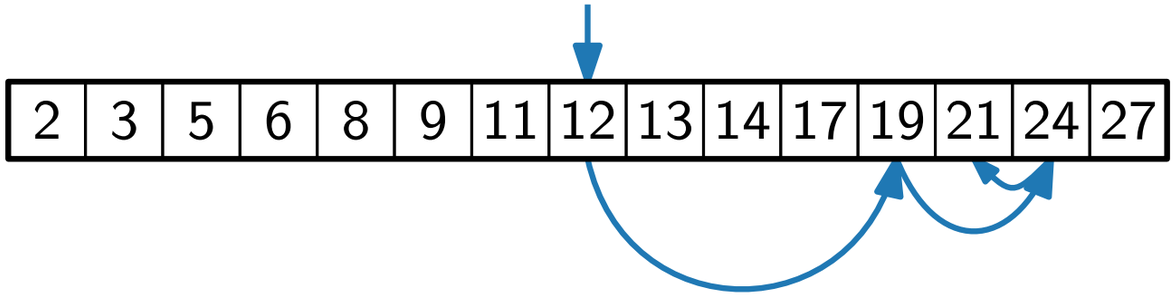
	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	**?**	Schritte

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)

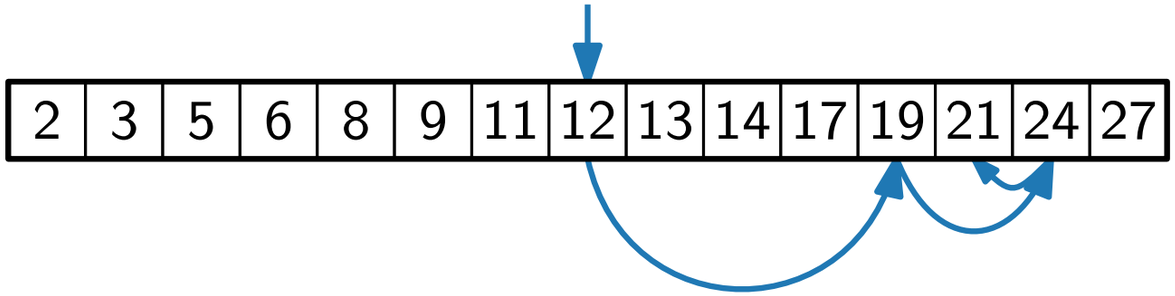
	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	**?**	Schritte

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

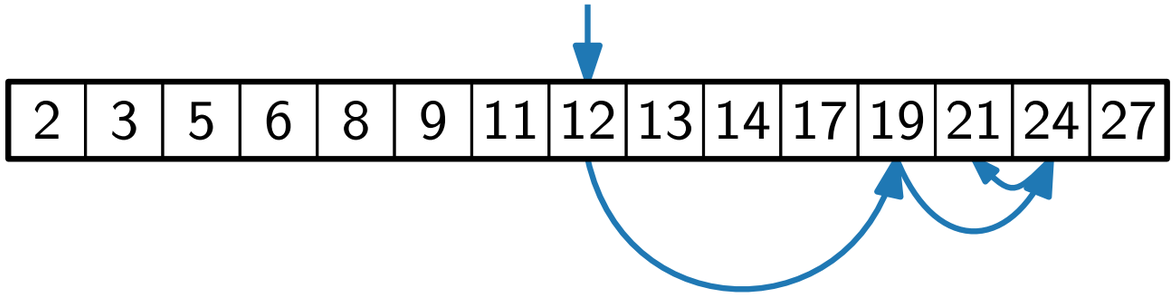
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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

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

$\leq$ $\qquad\qquad + 1$

Suche im sortierten Feld



SEARCH(21)

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

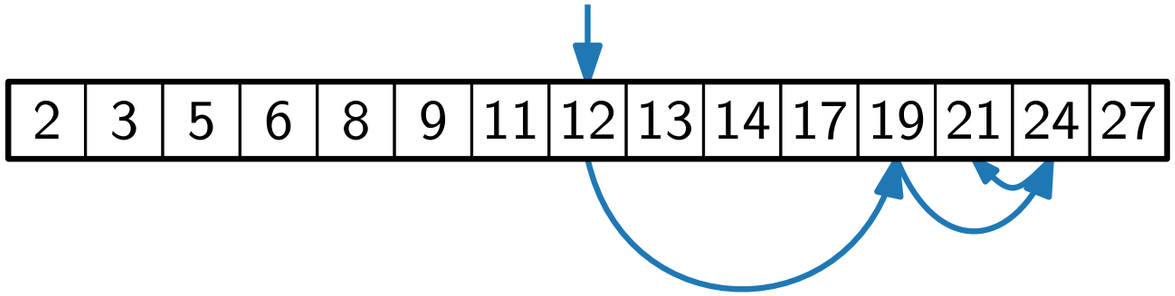
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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

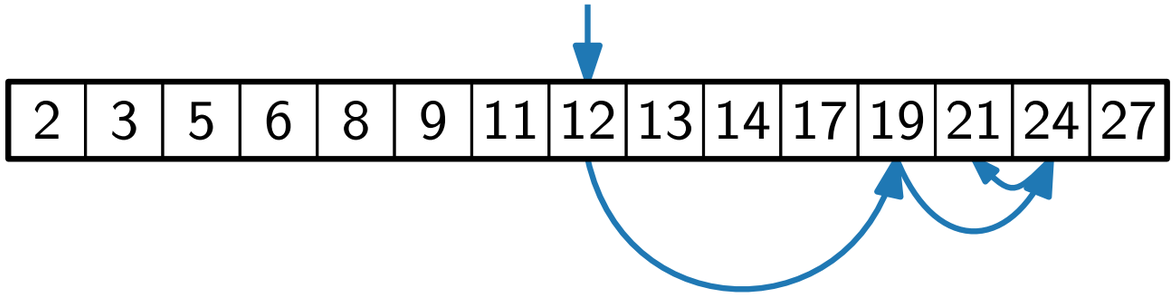
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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

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

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

Suche im sortierten Feld



SEARCH(21)

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

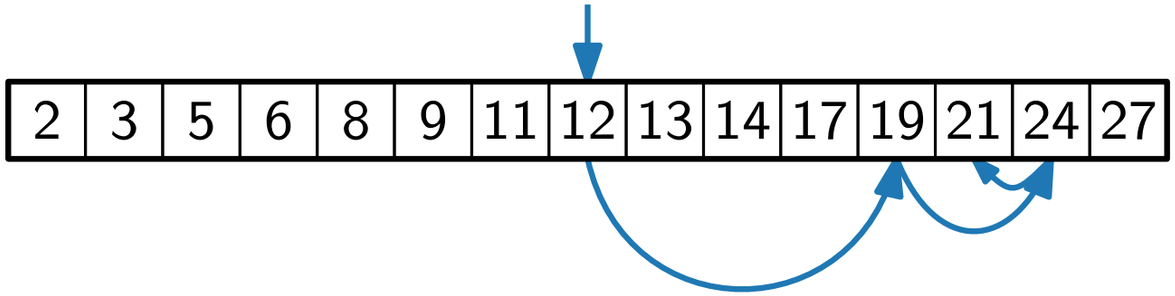
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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

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

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

Suche im sortierten Feld



SEARCH(21)

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

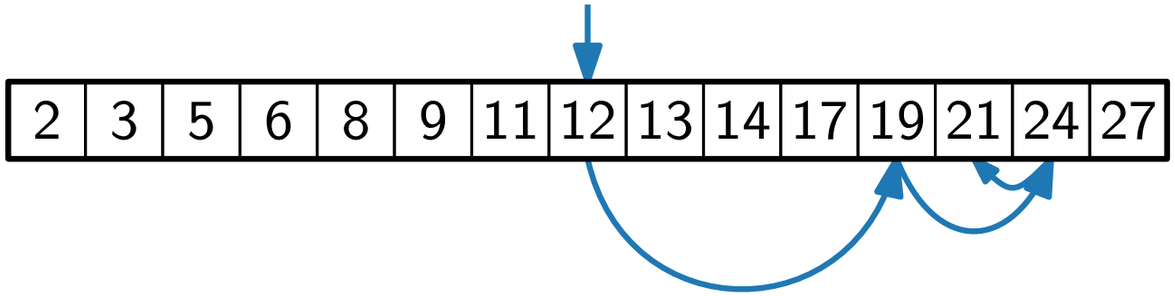
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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

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

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

Suche im sortierten Feld



SEARCH(21)

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

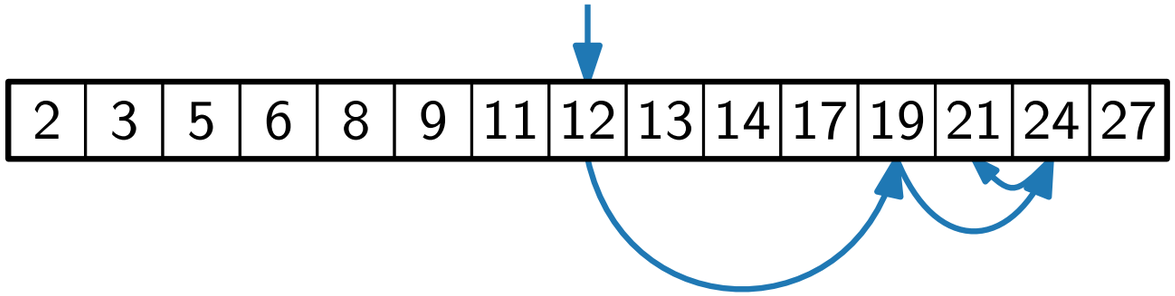
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

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

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

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

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 $<$).

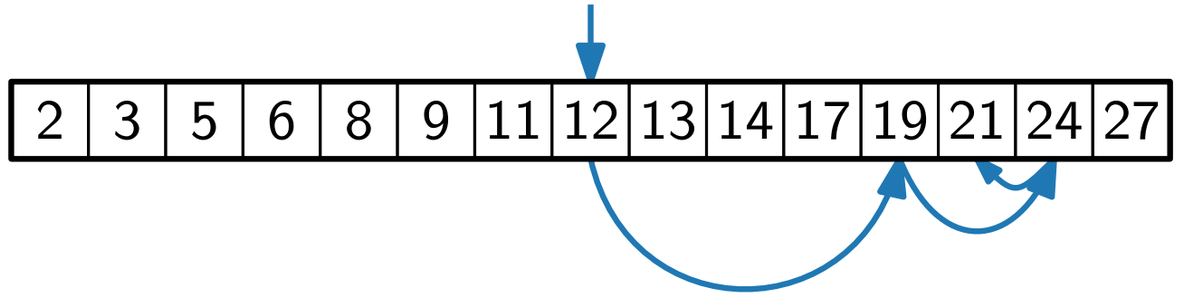
grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

$$= 1 + \lfloor \log_2 n \rfloor$$

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 $<$).

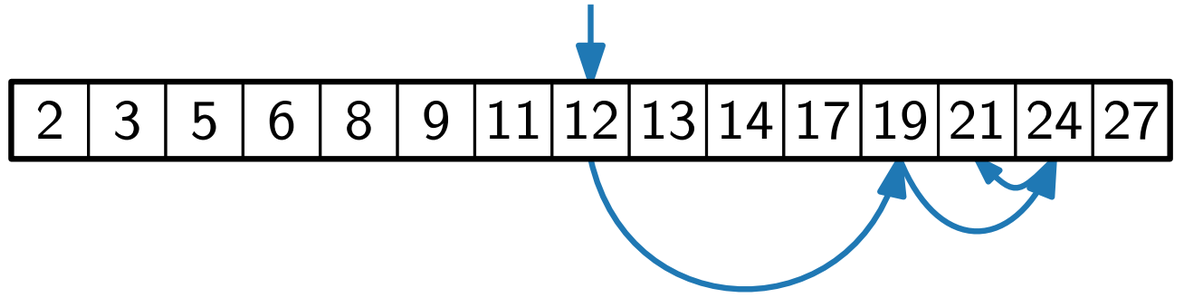
grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

$$= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2 (n+1) \rceil$$

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 $<$).

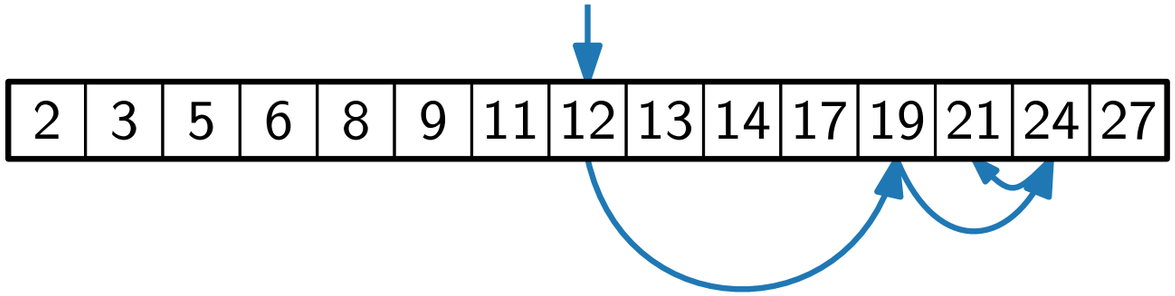
grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

$$= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil$$

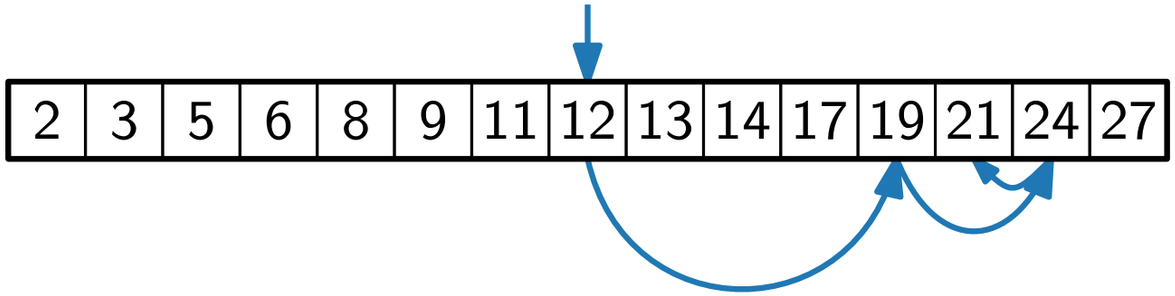
Suche im sortierten Feld



SEARCH(21)

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

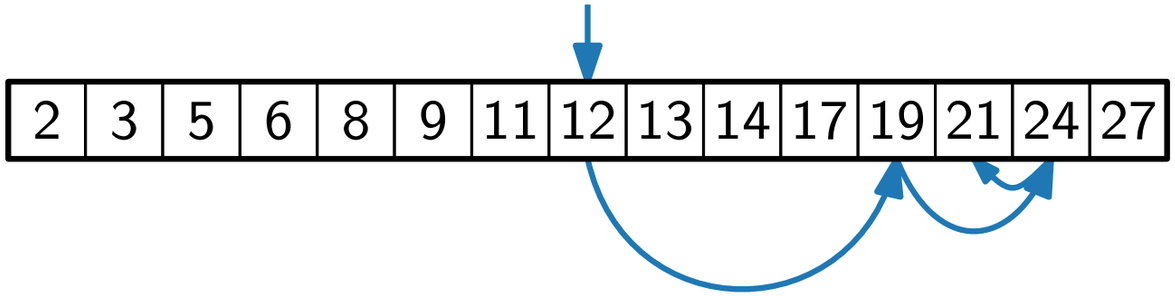
Suche im sortierten Feld



SEARCH(21)

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

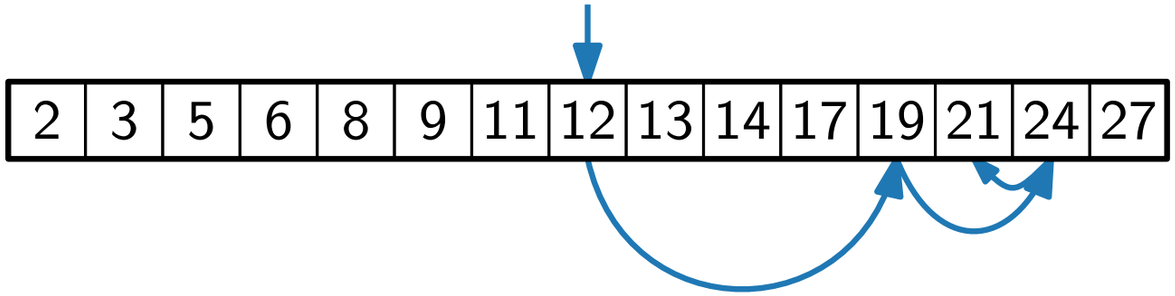
Suche im sortierten Feld



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

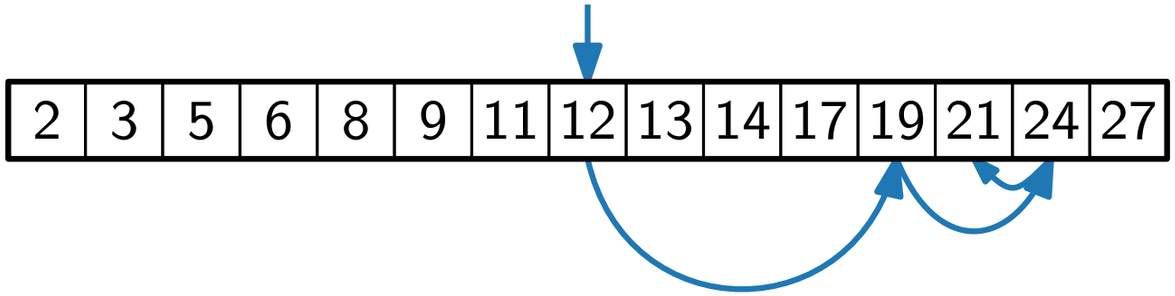
Suche im sortierten Feld



SEARCH(21)

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

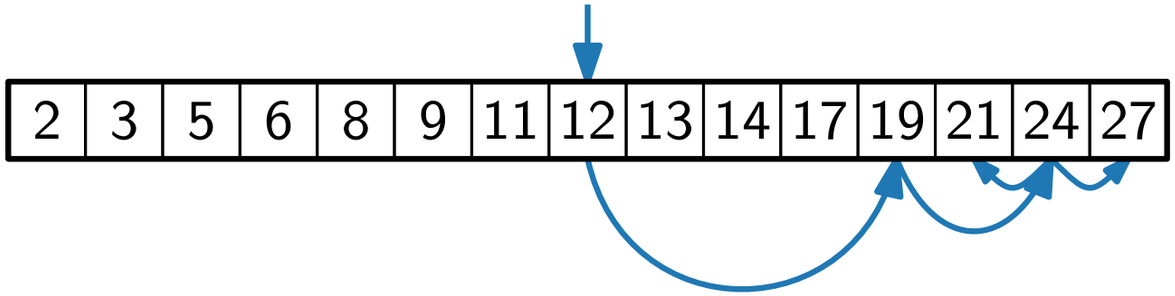
Suche im sortierten Feld



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

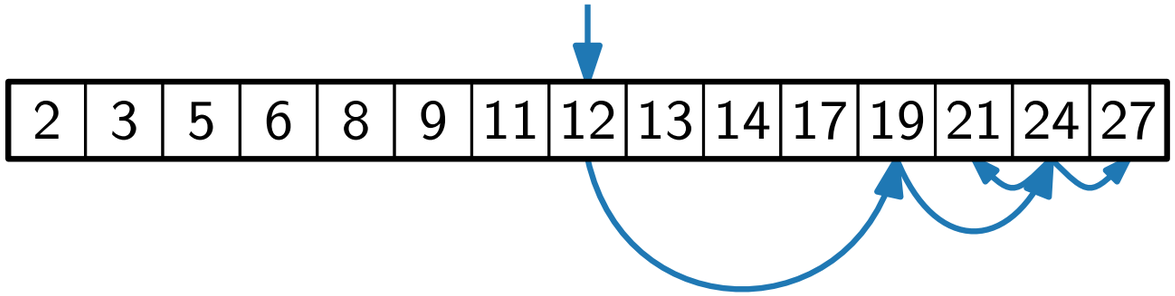
Suche im sortierten Feld



SEARCH(21)
SEARCH(27)

	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

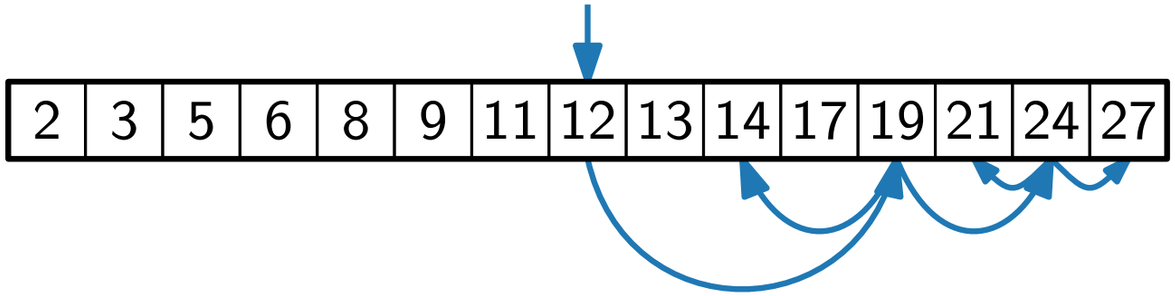
Suche im sortierten Feld



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

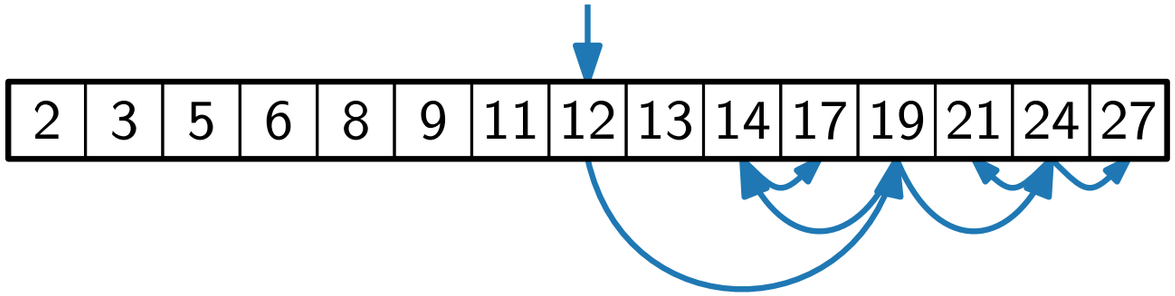
Suche im sortierten Feld



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

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

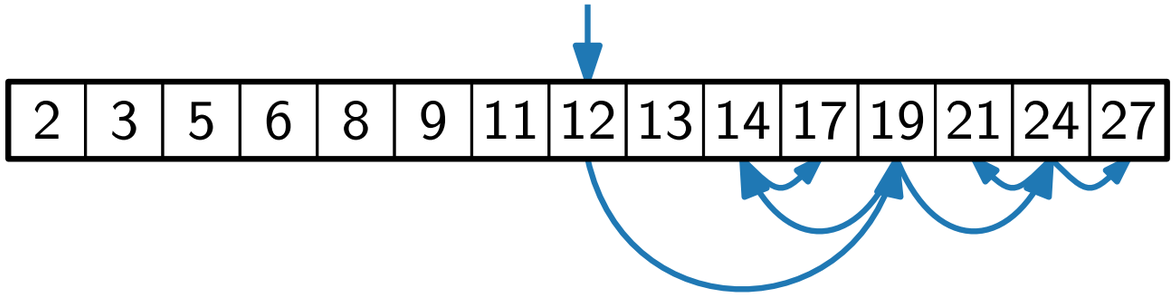
Suche im sortierten Feld



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

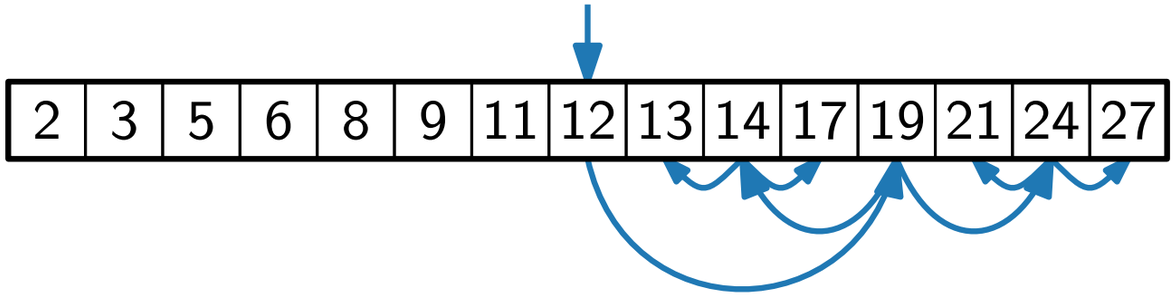
Suche im sortierten Feld



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

	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

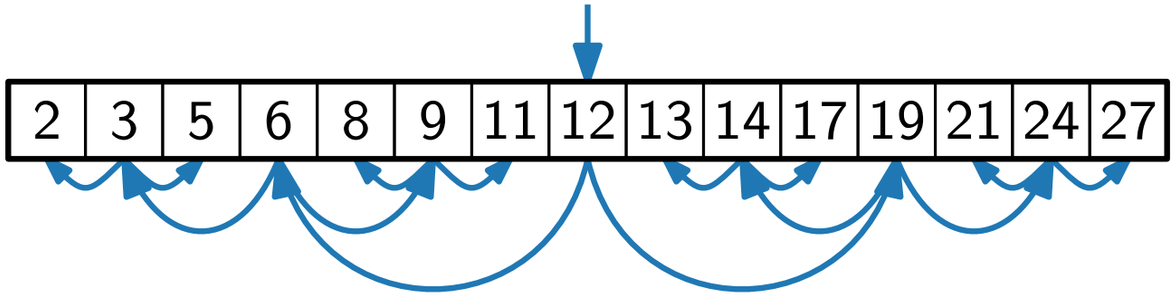
Suche im sortierten Feld



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

	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

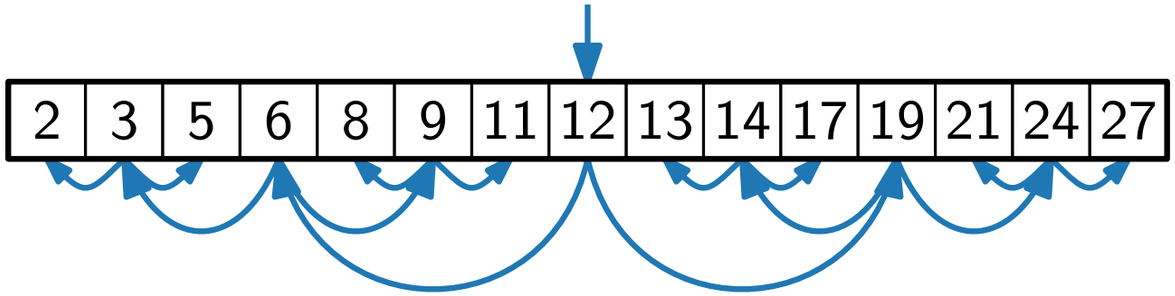
Suche im sortierten Feld



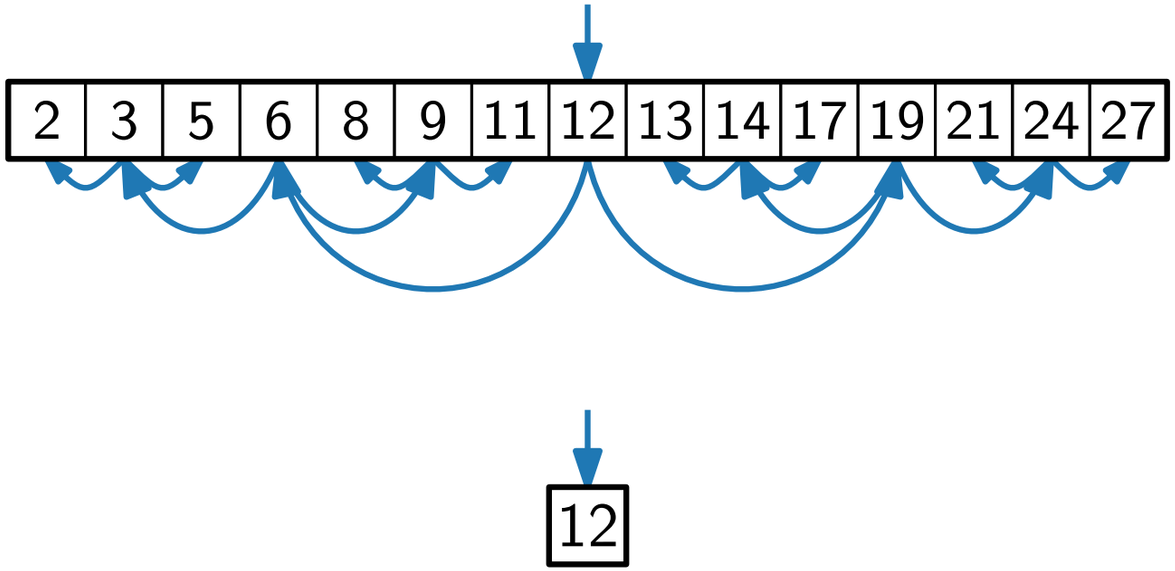
SEARCH(21)
SEARCH(27)
SEARCH(17)
SEARCH(13)

	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

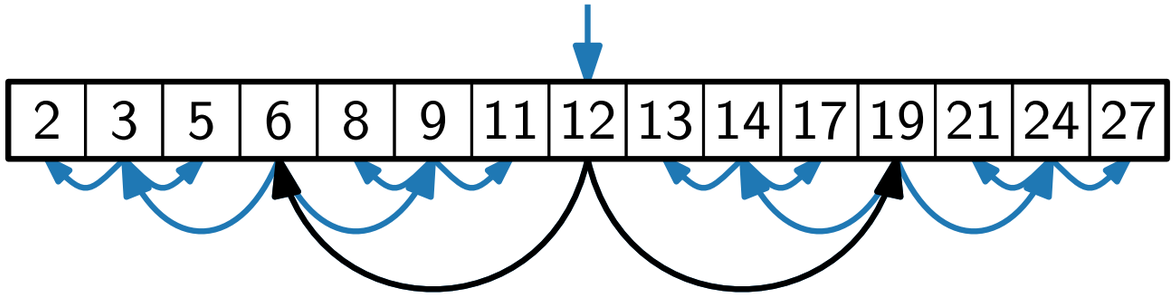
Suche im sortierten Feld

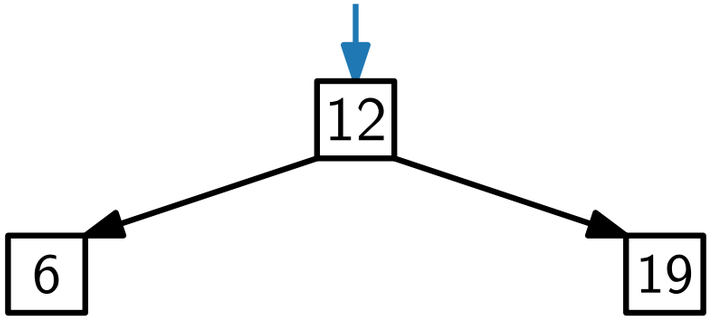


Suche im sortierten Feld

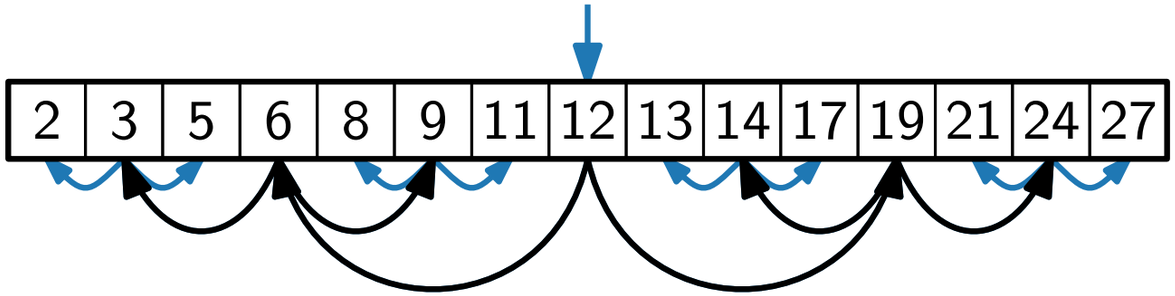


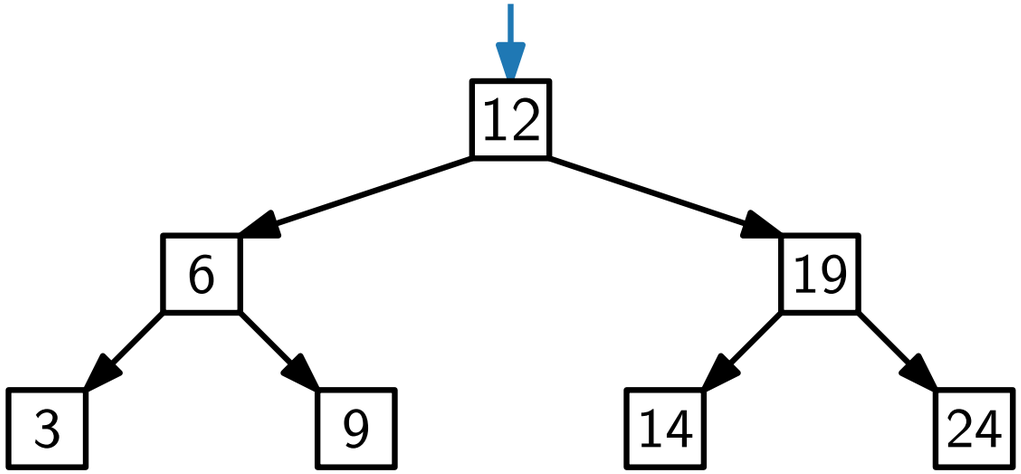
Suche im sortierten Feld



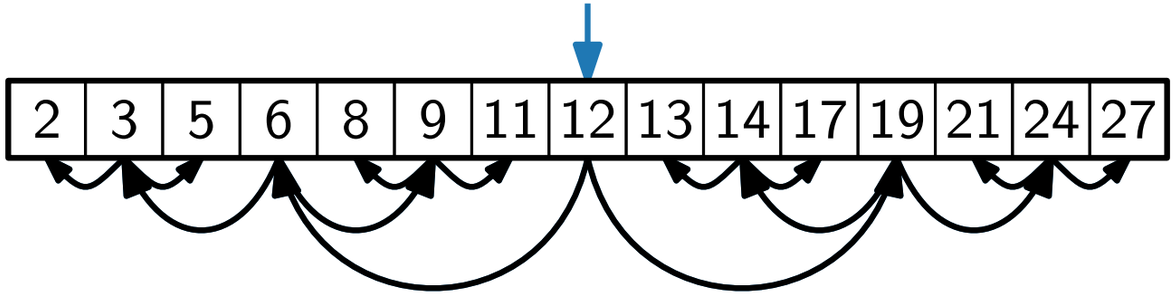


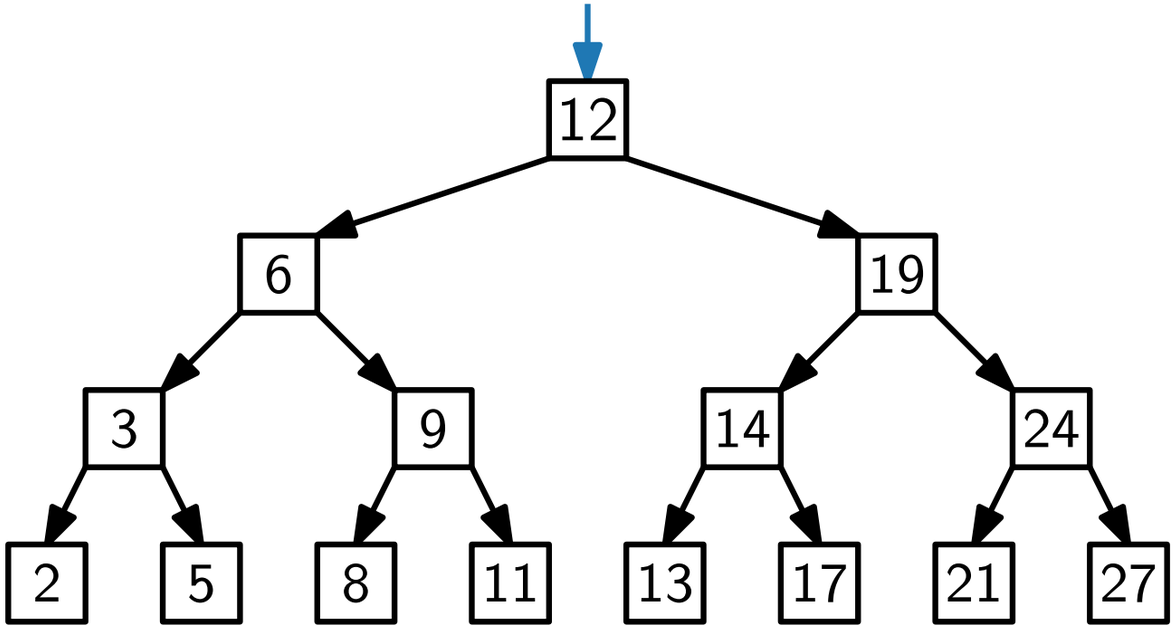
Suche im sortierten Feld



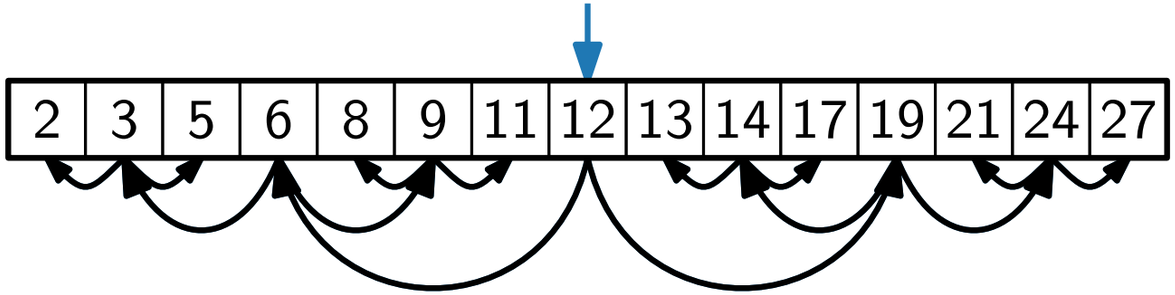


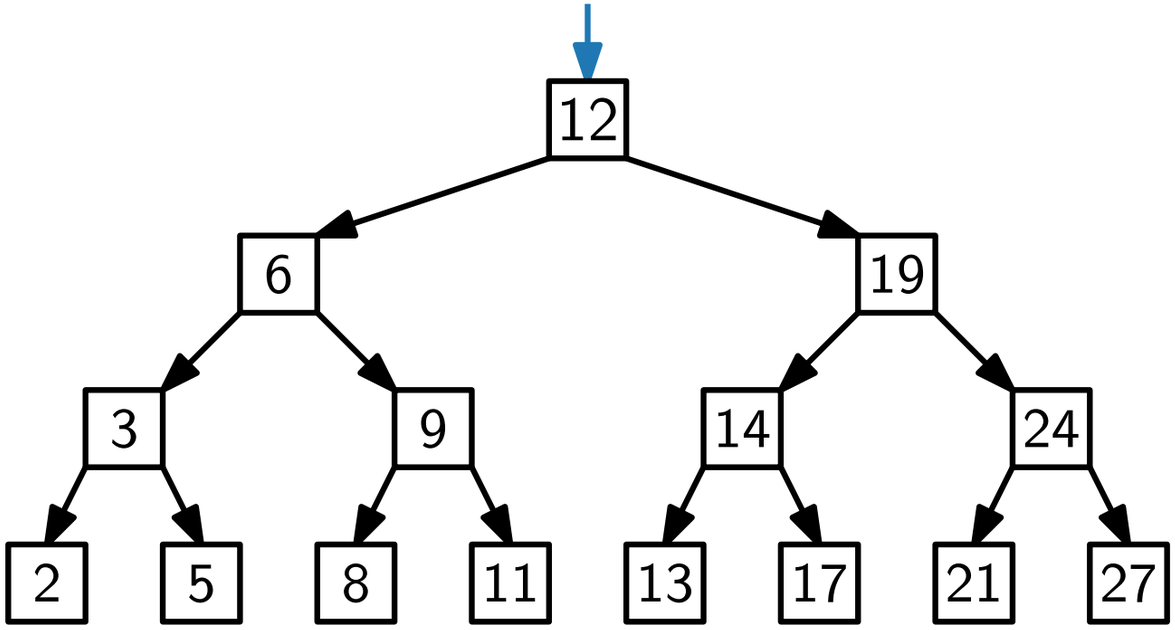
Suche im sortierten Feld





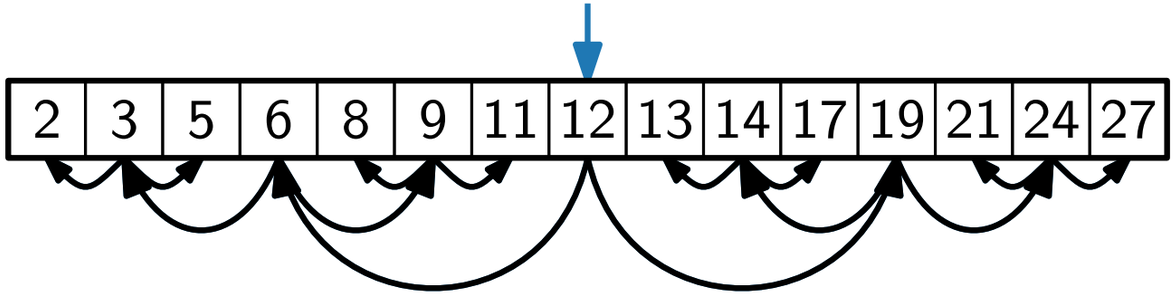
Suche im sortierten Feld

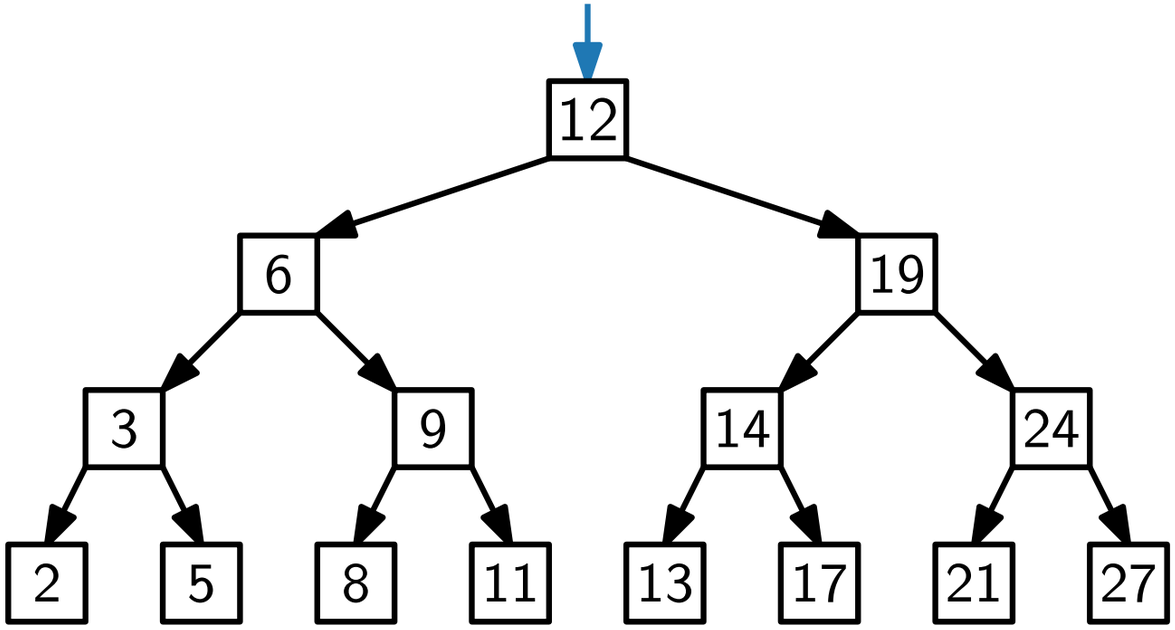




Binärer Suchbaum

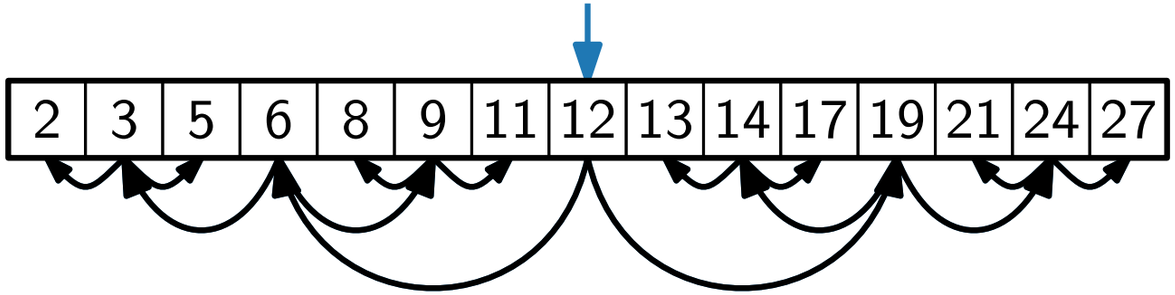
Suche im sortierten Feld

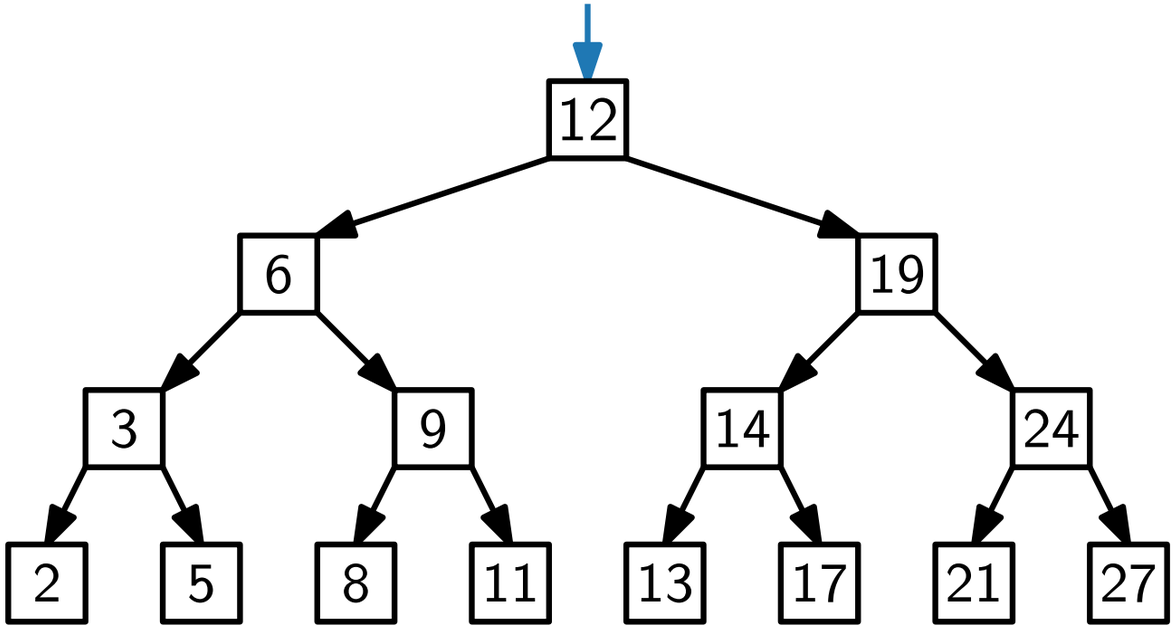




Binärer Suchbaum

Suche im sortierten Feld

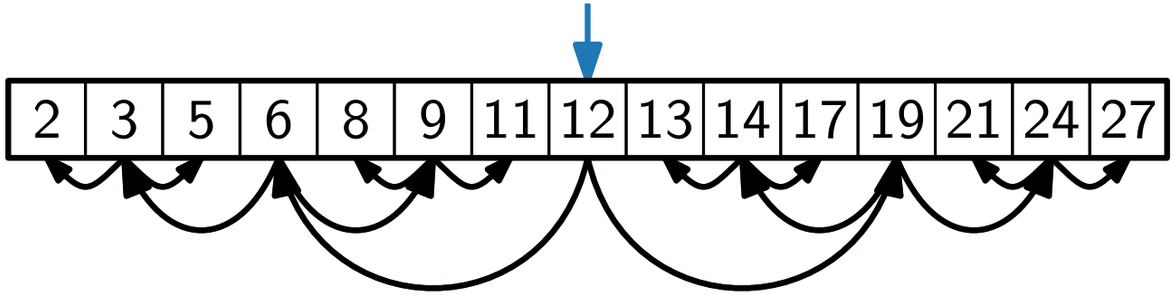


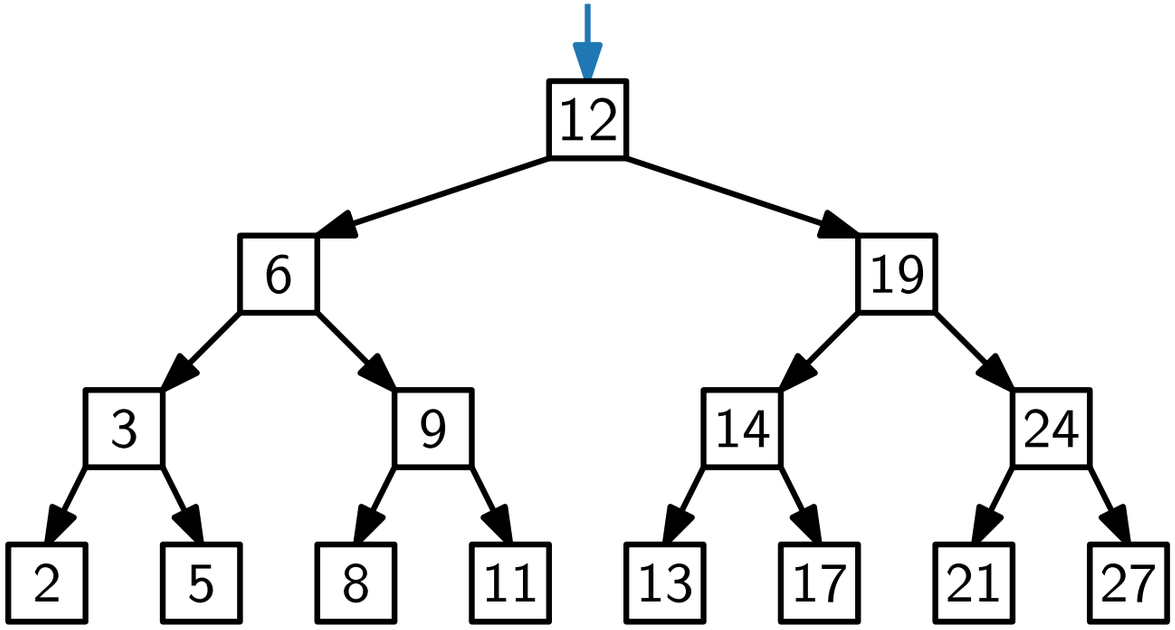


Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld

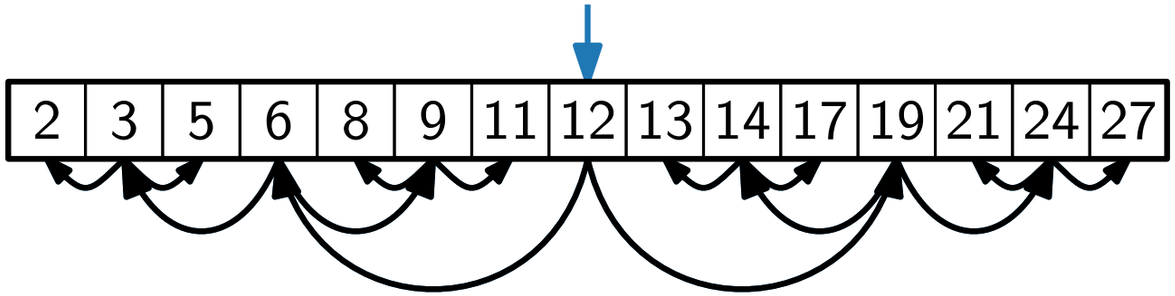


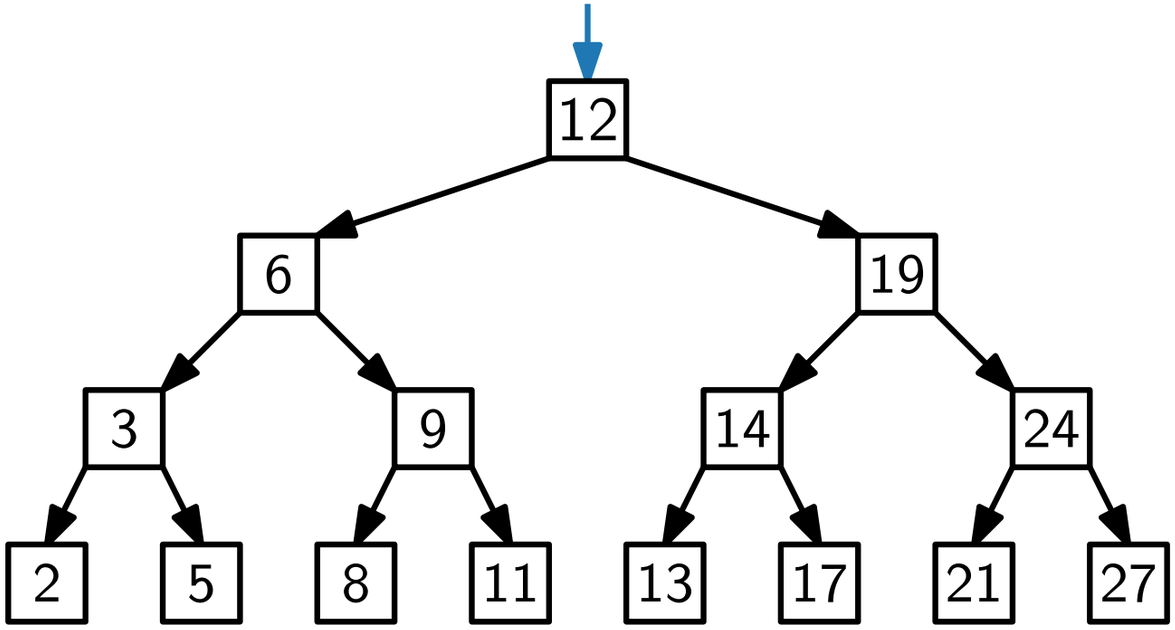


Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



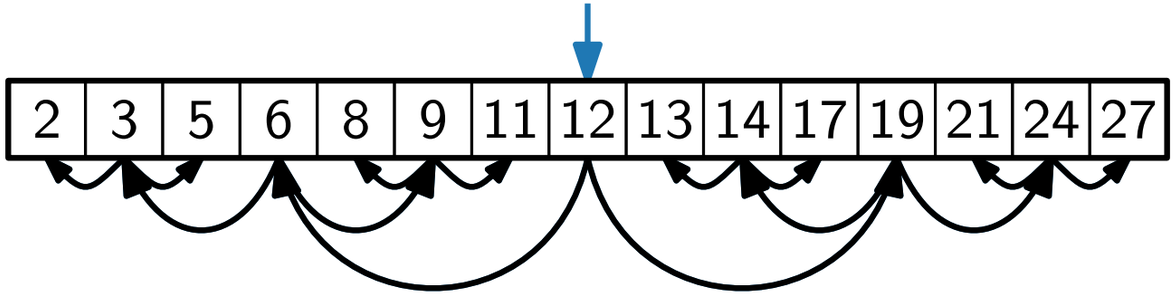


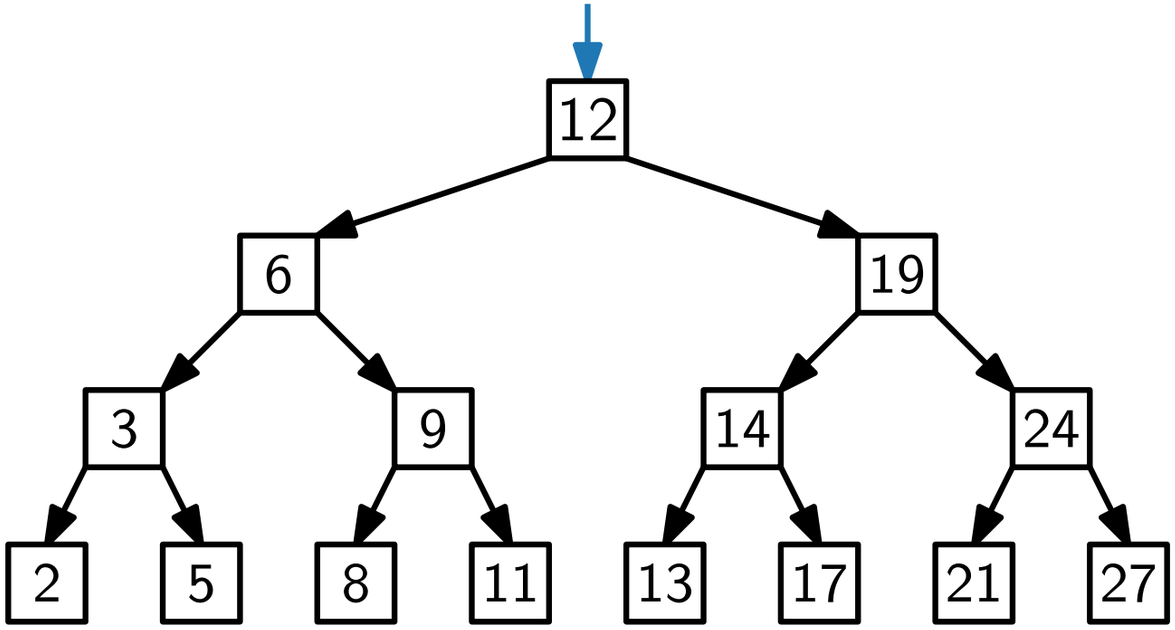
Jeder Knoten hat höchstens zwei Kinder.

Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



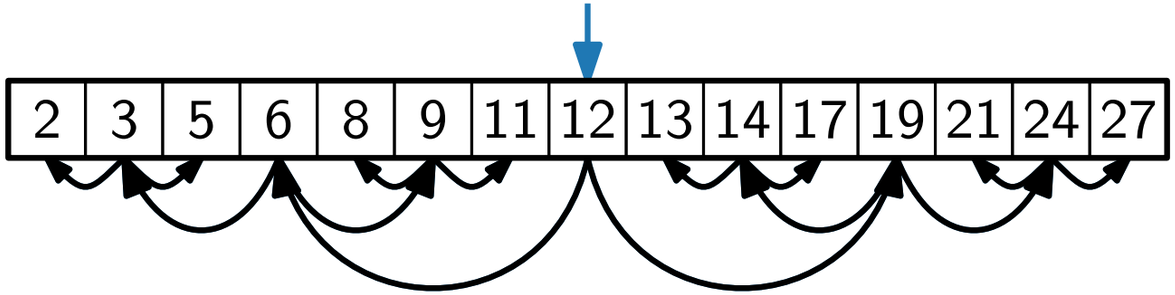


Jeder Knoten hat höchstens zwei Kinder.

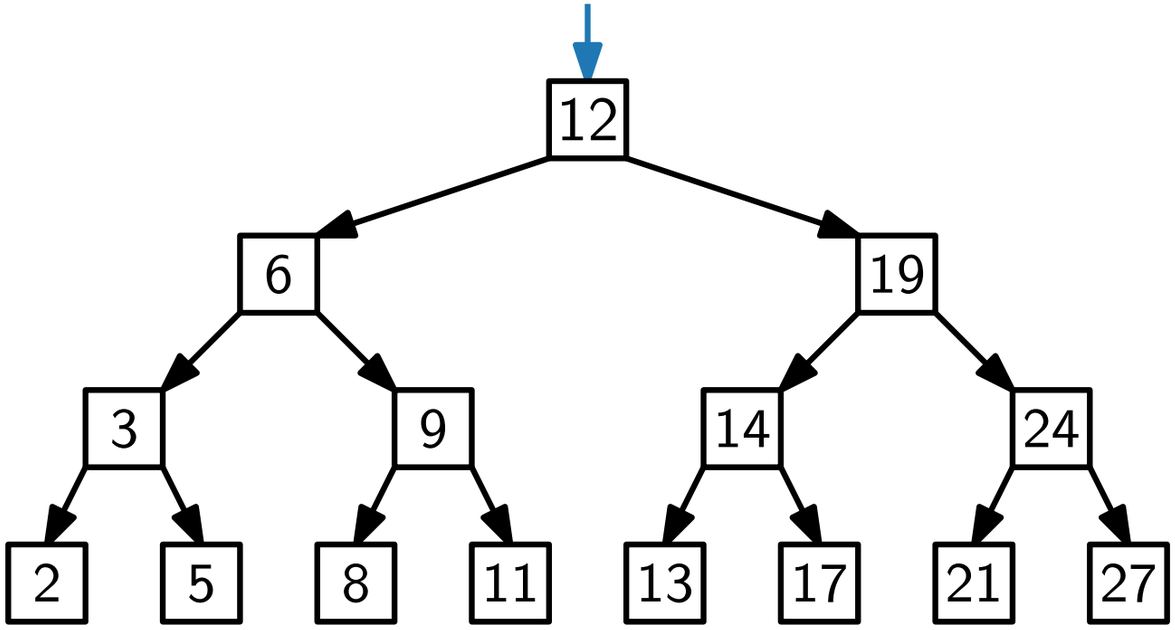
Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

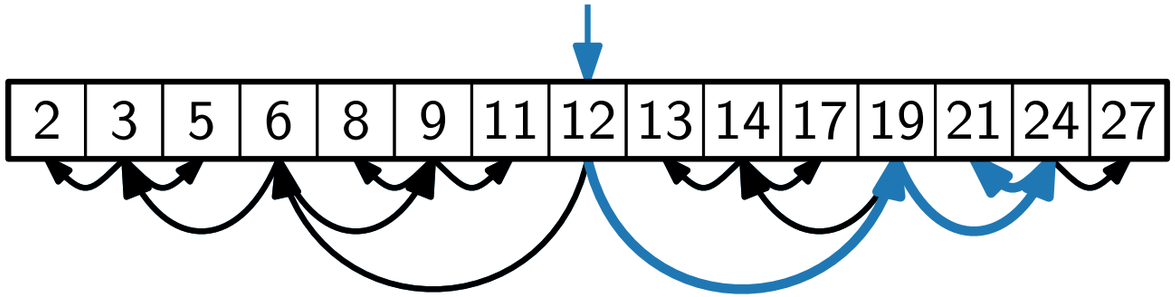


Jeder Knoten hat höchstens zwei Kinder.

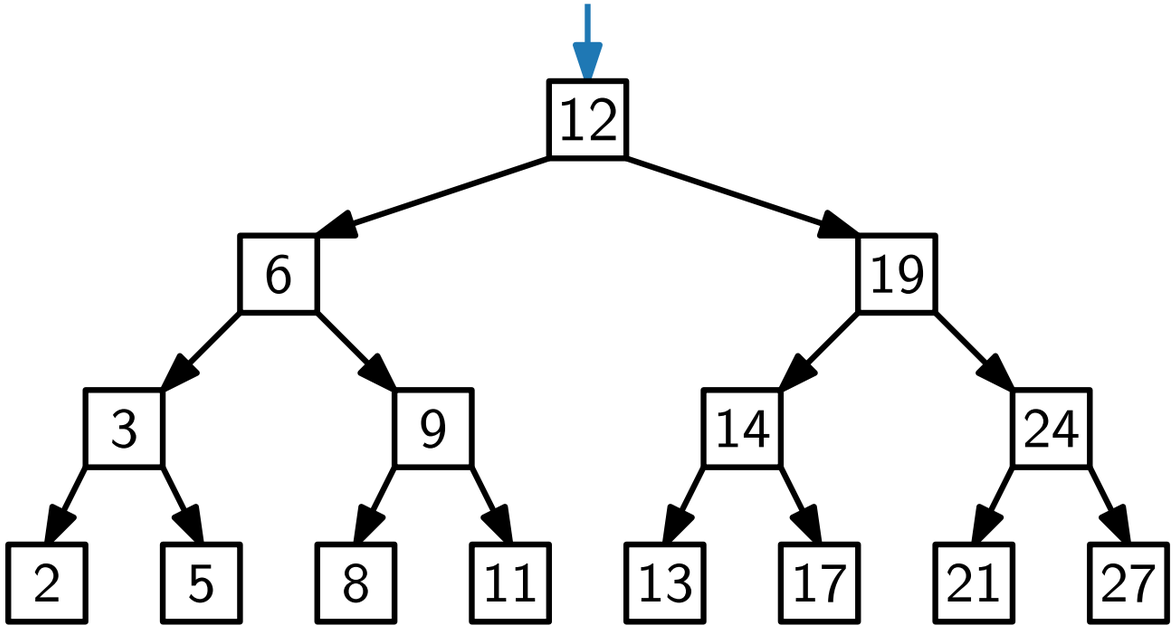
Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

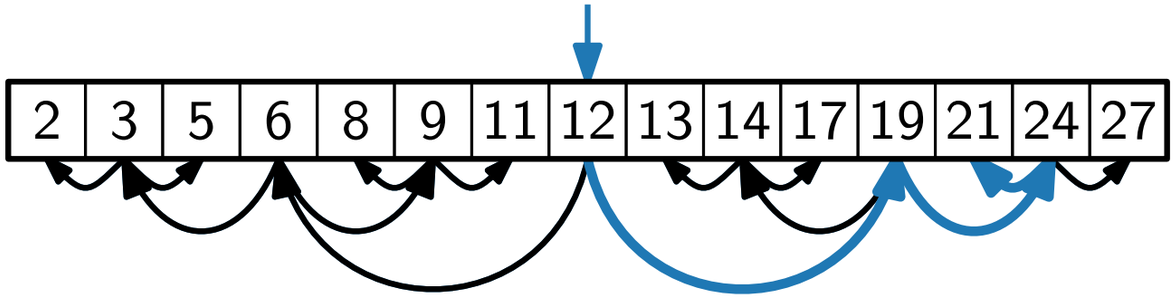


Jeder Knoten hat höchstens zwei Kinder.

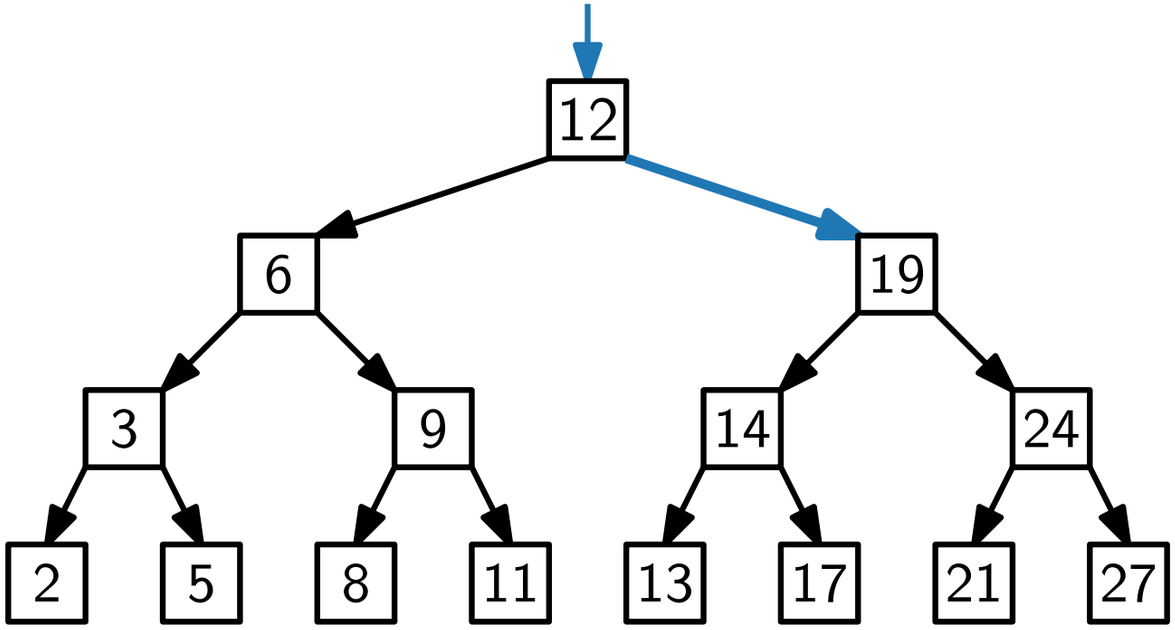
Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

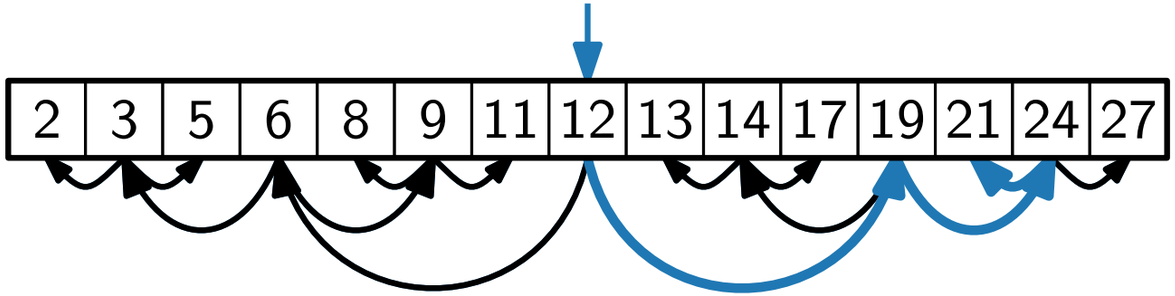


Jeder Knoten hat höchstens zwei Kinder.

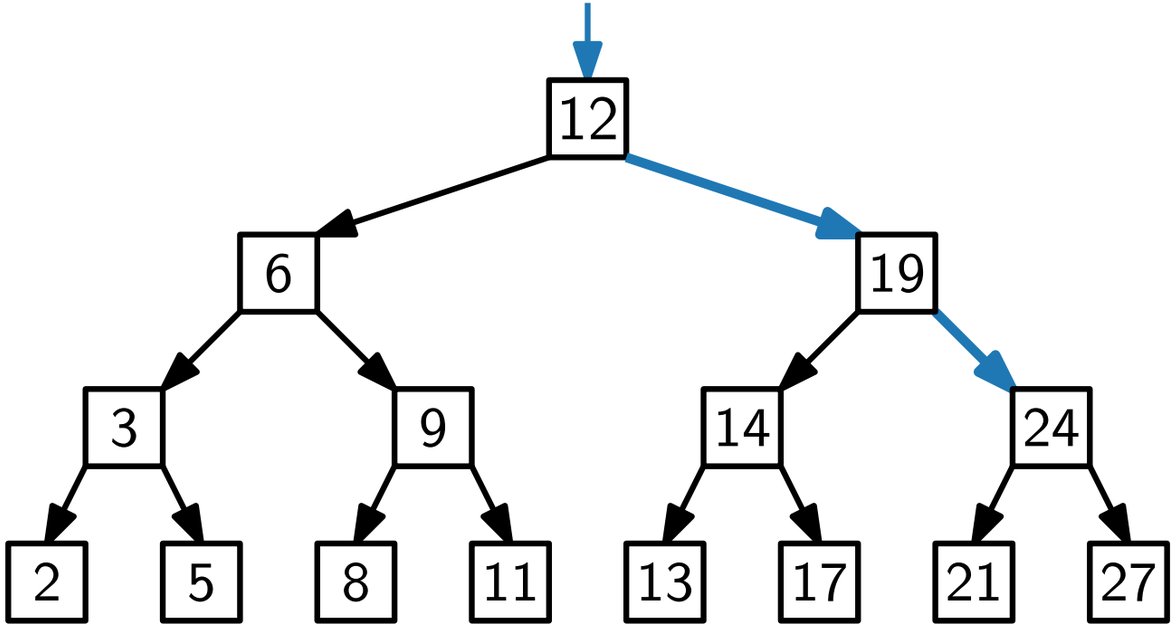
Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

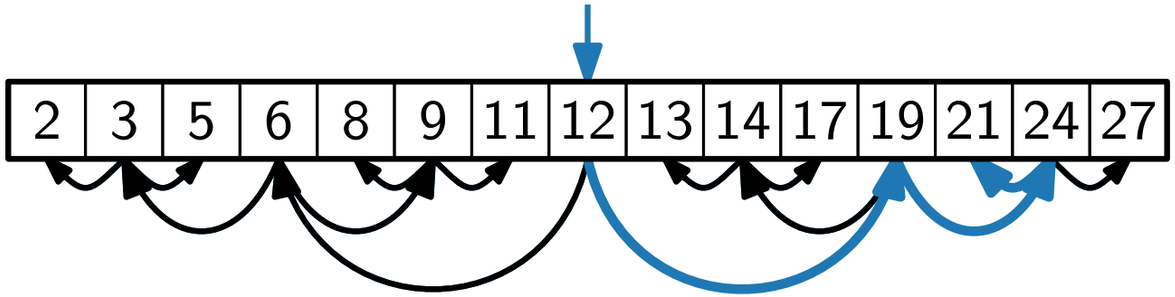


Jeder Knoten hat höchstens zwei Kinder.

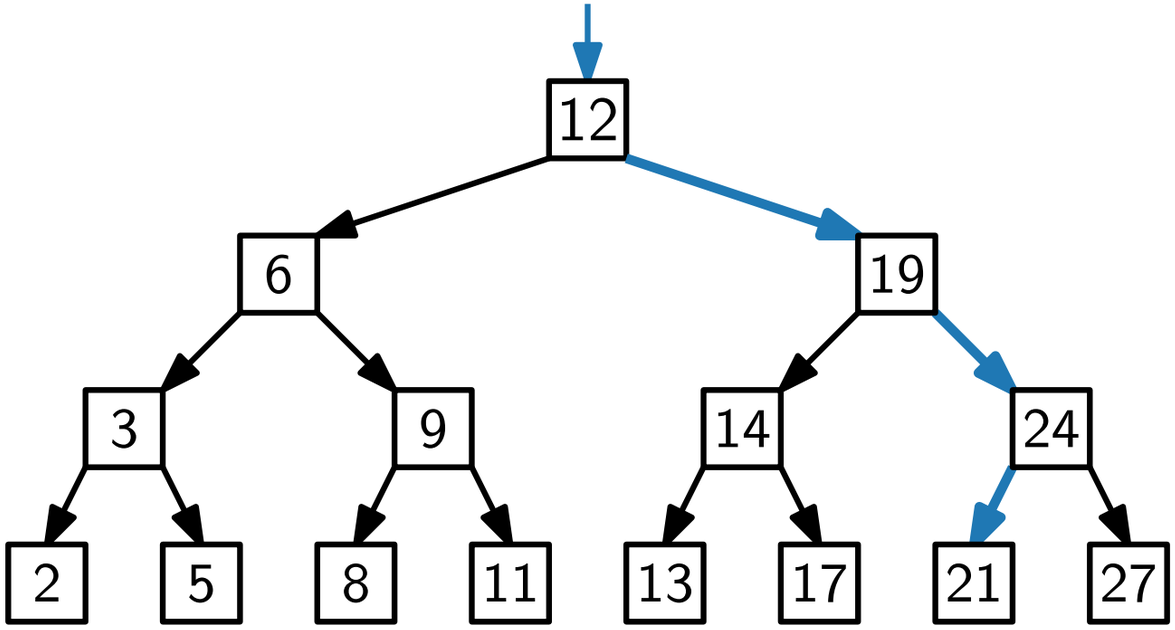
Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

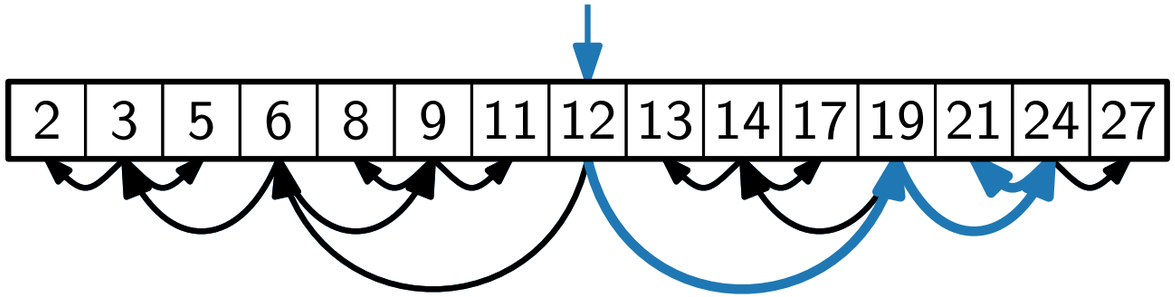


Jeder Knoten hat höchstens zwei Kinder.

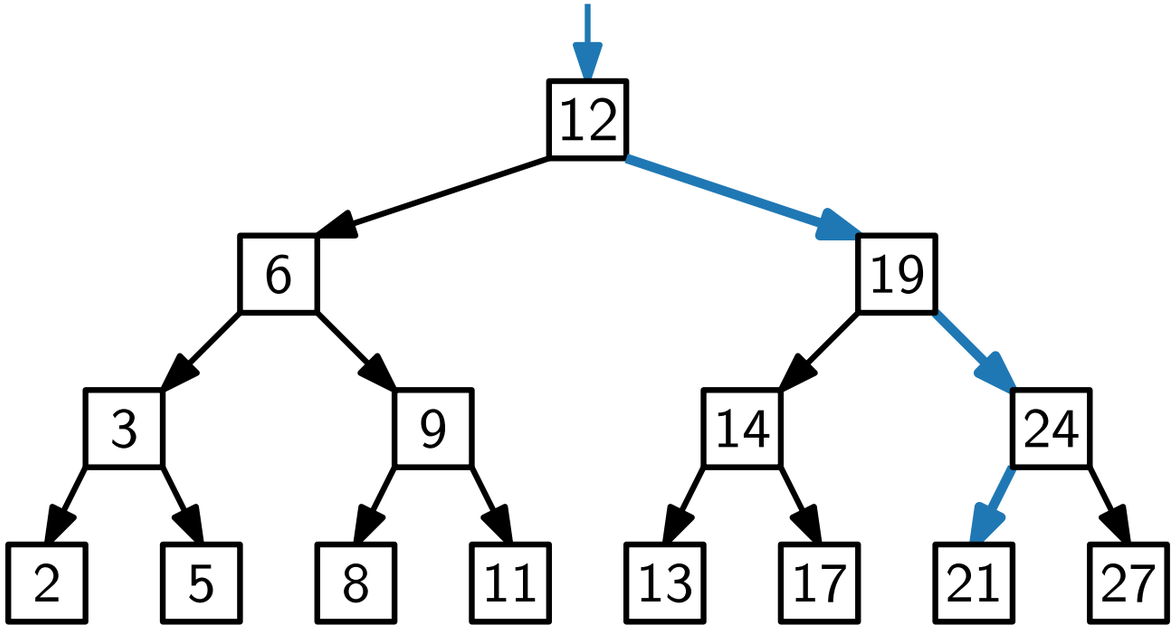
Binärer Suchbaum

zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)



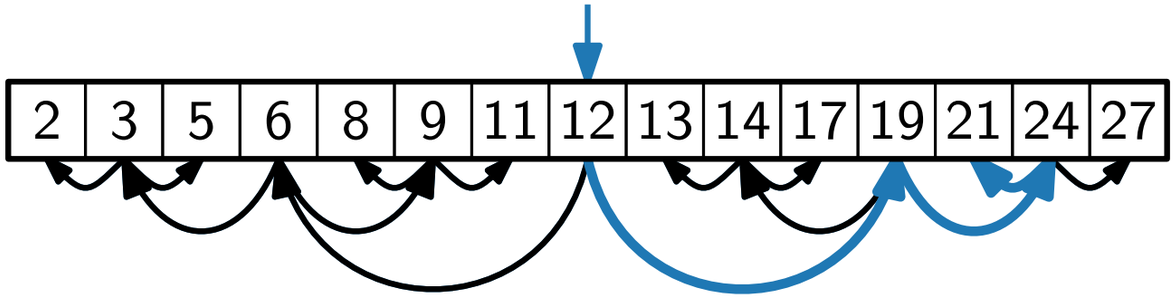
Jeder Knoten hat höchstens zwei Kinder.

Binärer Suchbaum

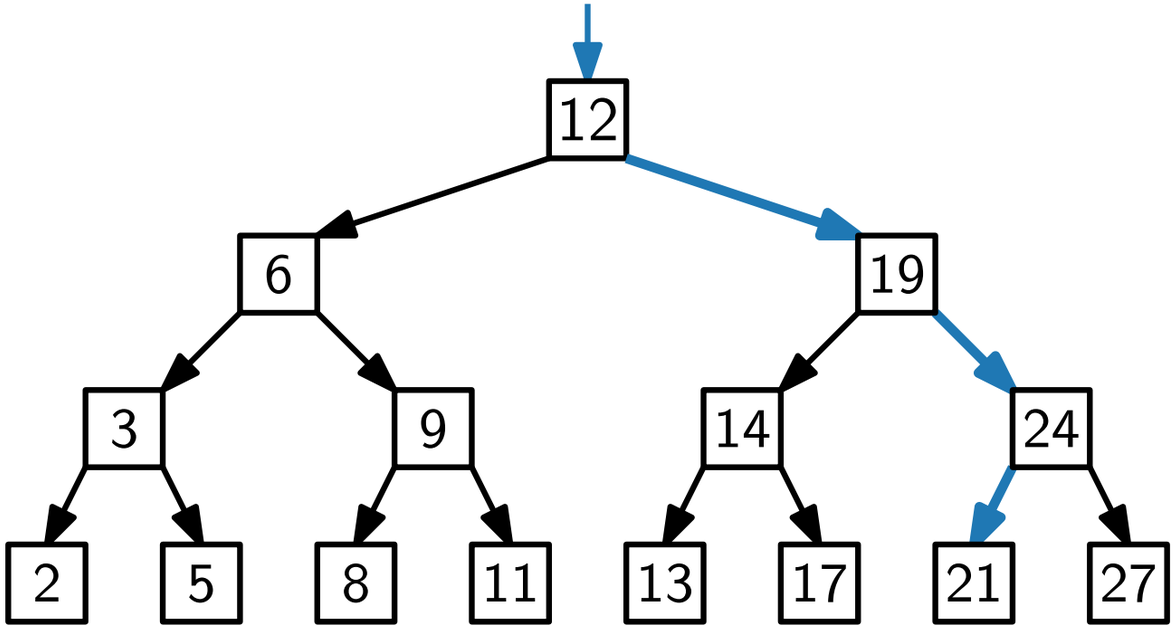
zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Binärer-Suchbaum-Eigenschaft.

Suche im sortierten Feld



SEARCH(21)



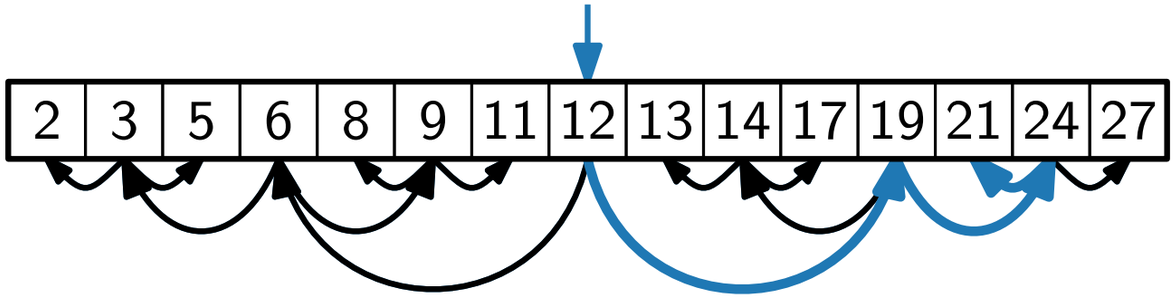
Jeder Knoten hat höchstens zwei Kinder.

Binärer Suchbaum

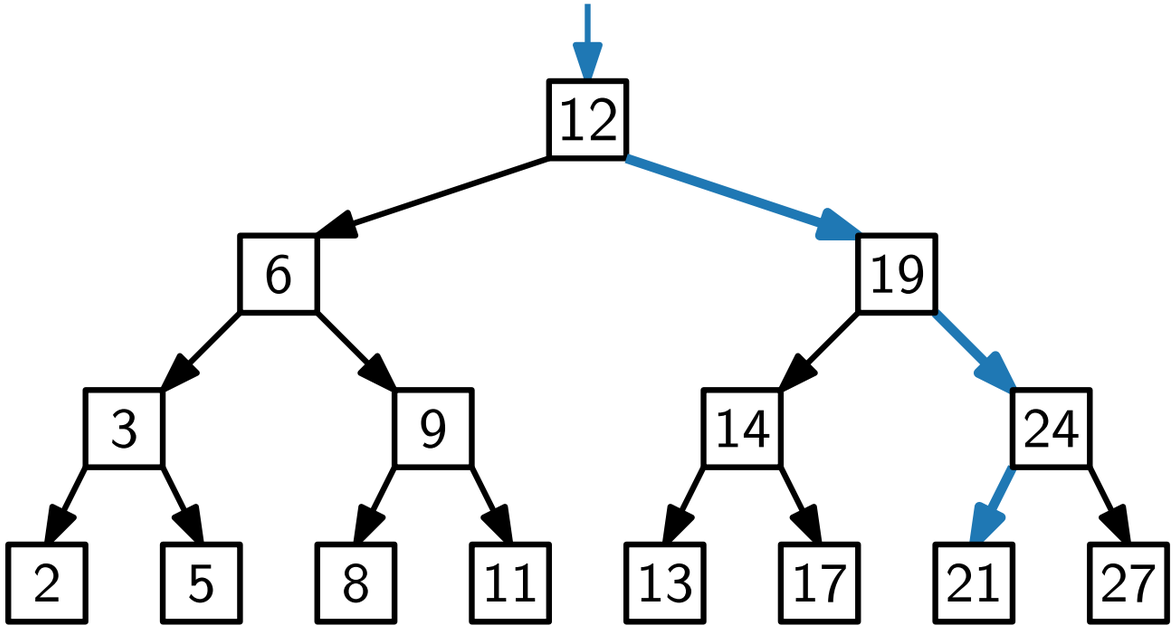
zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

Binärer-Suchbaum-Eigenschaft. Für jeden Knoten v gilt:

Suche im sortierten Feld



SEARCH(21)



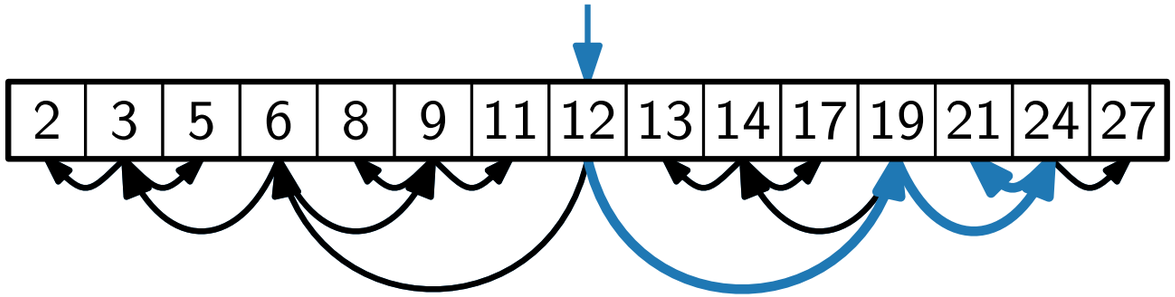
Jeder Knoten hat höchstens zwei Kinder.

Binärer Suchbaum

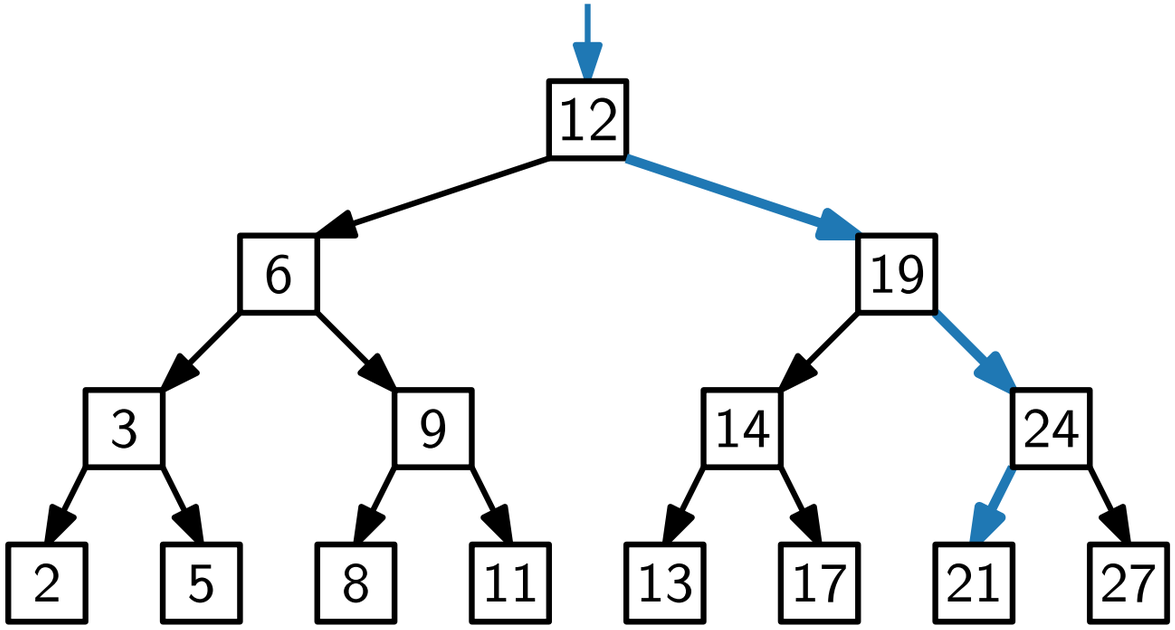
zusammenhängender kreisfreier Graph (mehr dazu in 2 Wochen)

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 $\geq$

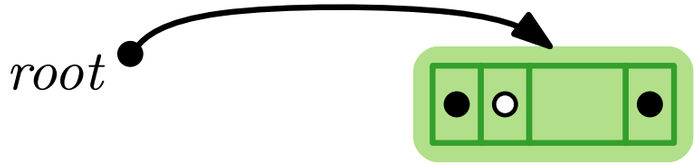
Binärer Suchbaum

Operation	Implementierung

Binärer Suchbaum

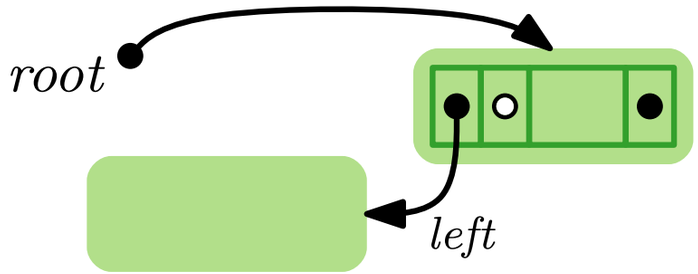
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum



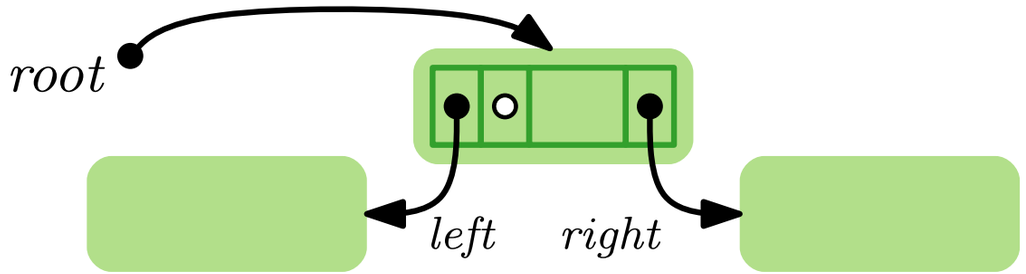
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum



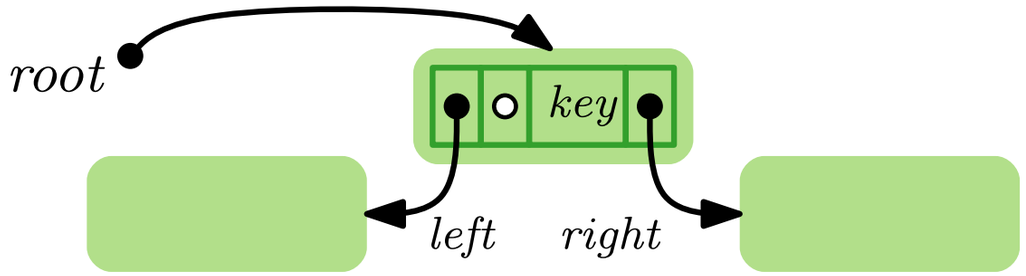
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum



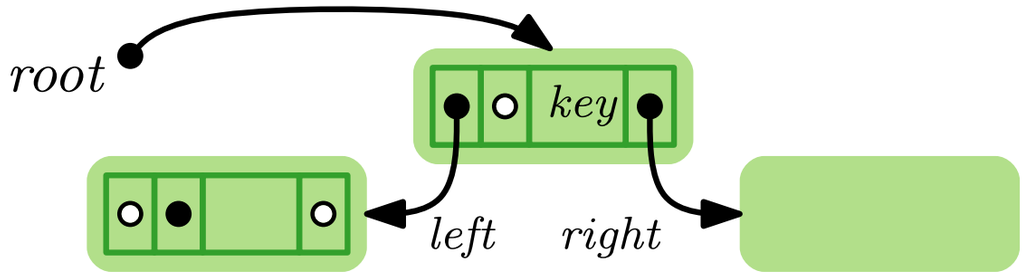
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum



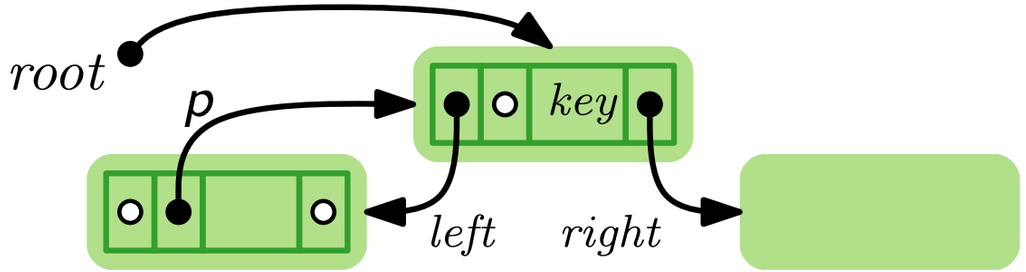
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum



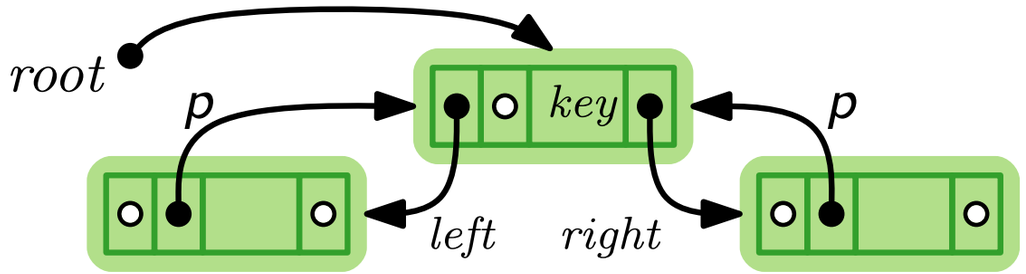
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum



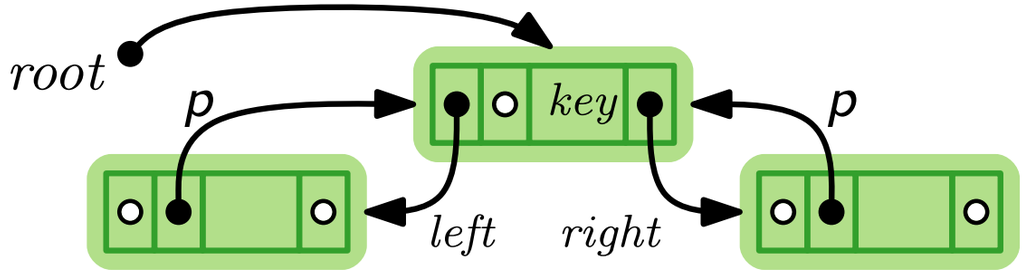
Operation	Implementierung
BinSearchTree()	

Binärer Suchbaum

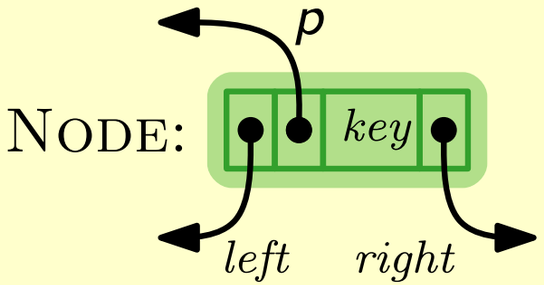


Operation	Implementierung
BinSearchTree()	

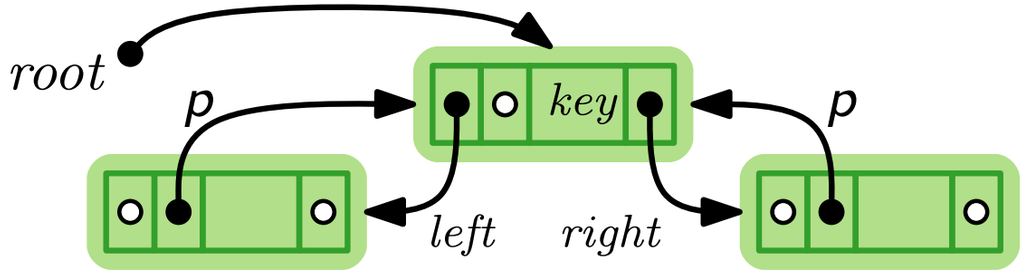
Binärer Suchbaum



Operation	Implementierung
BinSearchTree()	



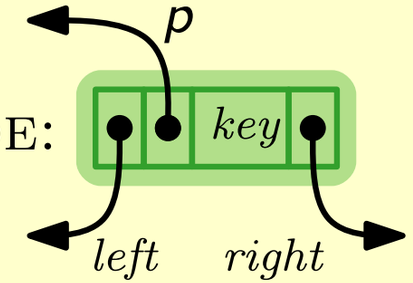
Binärer Suchbaum



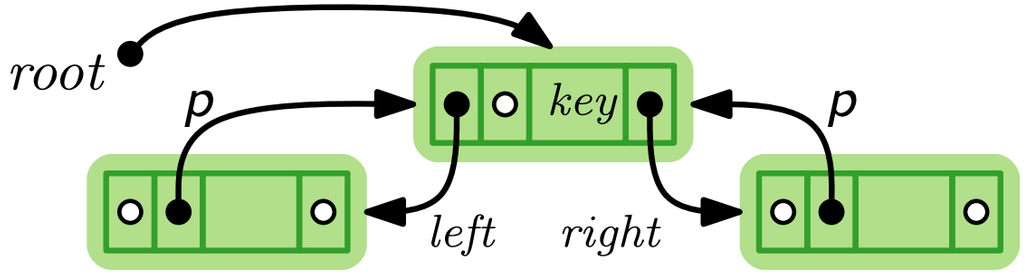
Operation	Implementierung
BinSearchTree()	

Node(key k, Node par)
$key = k$
$p = par$
$right = left = nil$

Node:



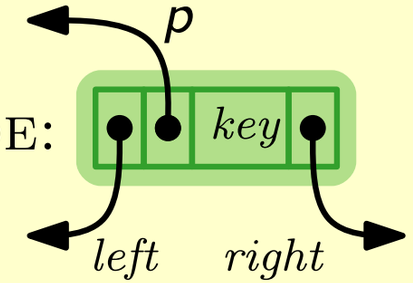
Binärer Suchbaum



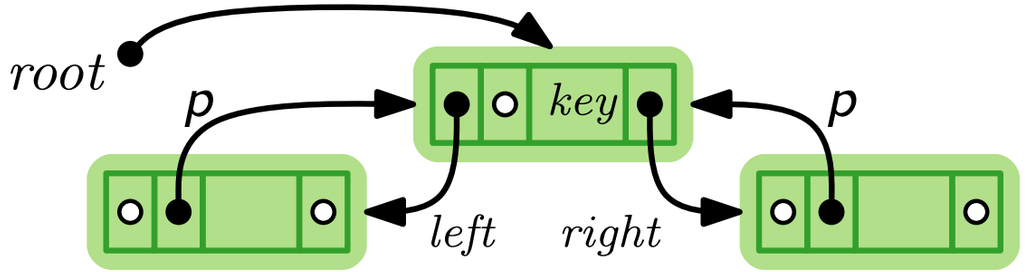
Operation	Implementierung
BinSearchTree()	$root = nil$

Node(key k, Node par)
$key = k$
$p = par$
$right = left = nil$

Node:

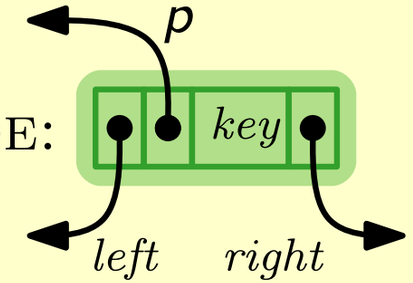


Binärer Suchbaum

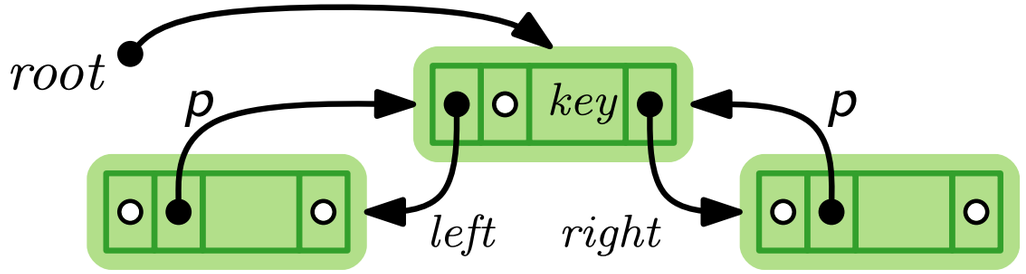


Operation	Implementierung
BinSearchTree()	$root = nil$
	Node(key k, Node par) $key = k$ $p = par$ $right = left = nil$
Node Search(key k)	
Node Insert(key k)	
Delete(Node x)	
Node Minimum()	
Node Maximum()	
Node Predecessor(Node x)	
Node Successor(Node x)	

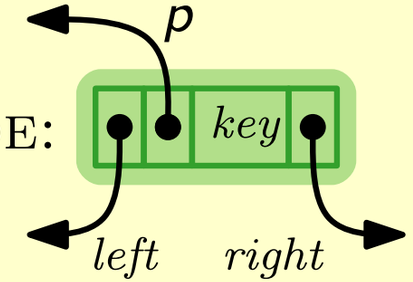
Node:



Binärer Suchbaum

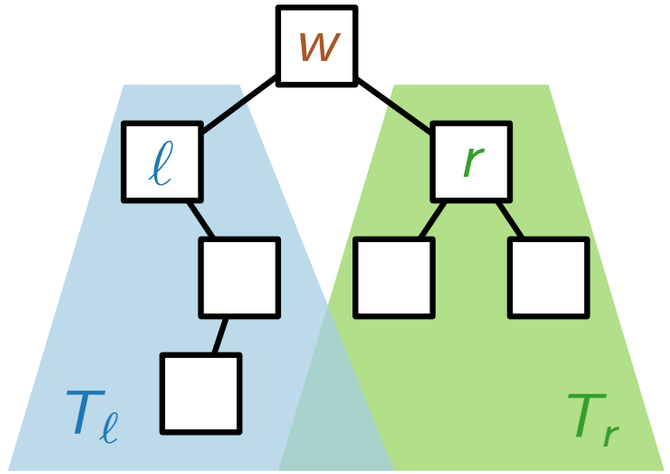


Operation	Implementierung
BinSearchTree()	$root = nil$
	Node(key k, Node par) $key = k$ $p = par$ $right = left = nil$
Node Search(key k)	TODO
Node Insert(key k)	
Delete(Node x)	
Node Minimum()	
Node Maximum()	
Node Predecessor(Node x)	
Node Successor(Node x)	



Inorder-Traversierung

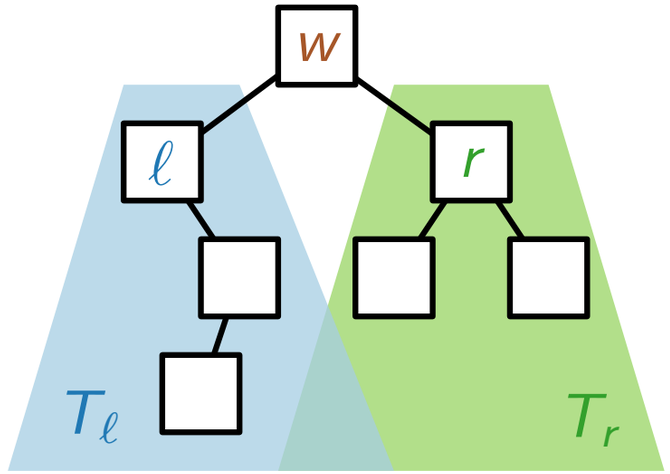
(Binäre) Bäume haben eine zur Rekursion einladende Struktur...



Inorder-Traversierung

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

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

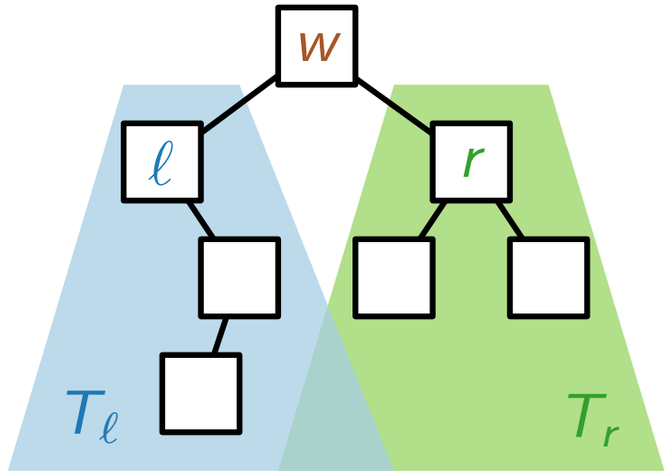


Inorder-Traversierung

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

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

Lösung:

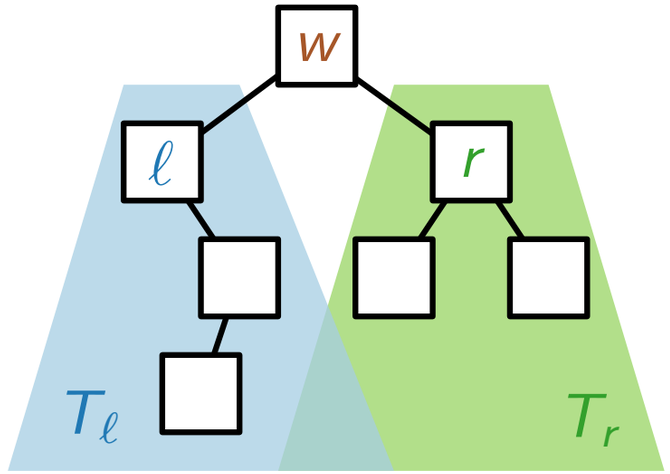


Inorder-Traversierung

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

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

Lösung: 1. Durchlaufe rekursiv linken Teilbaum der Wurzel.

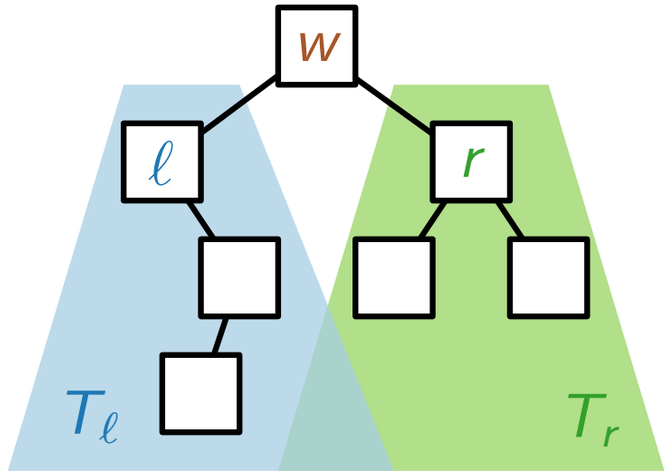


Inorder-Traversierung

(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.



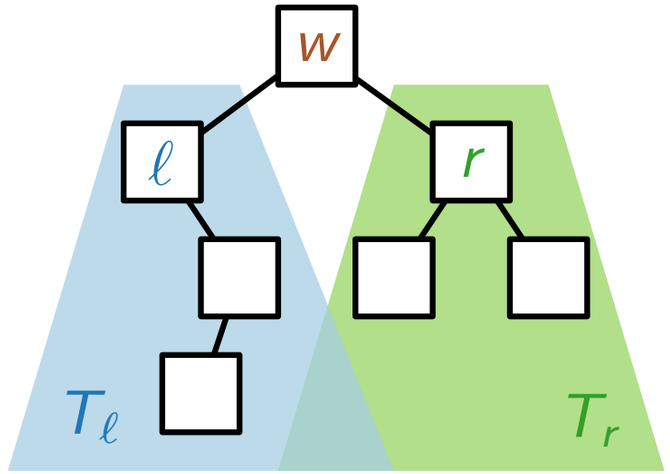
Inorder-Traversierung

(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

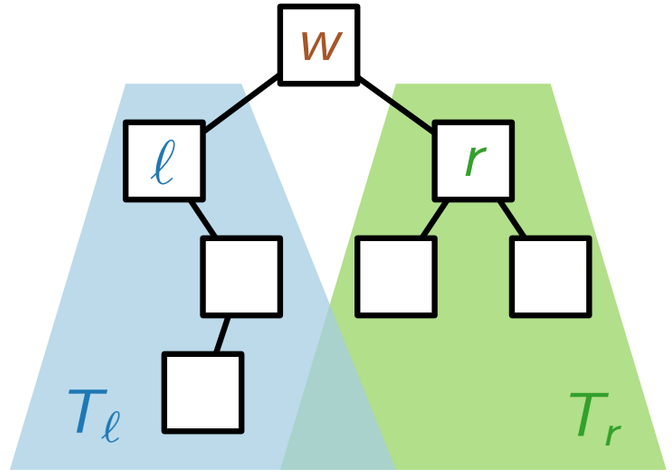
(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

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

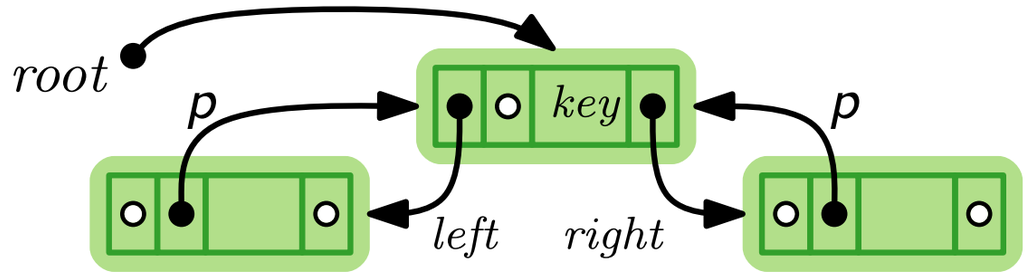
Lösung:
1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

```
InorderTreeWalk(Node x = root)
```



Inorder-Traversierung



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

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

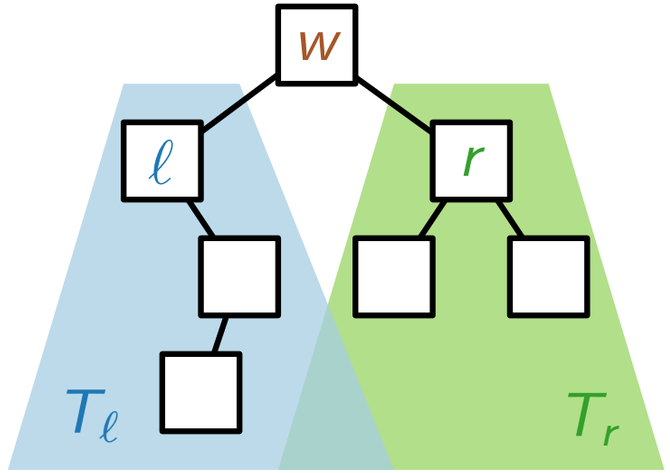
Lösung:

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

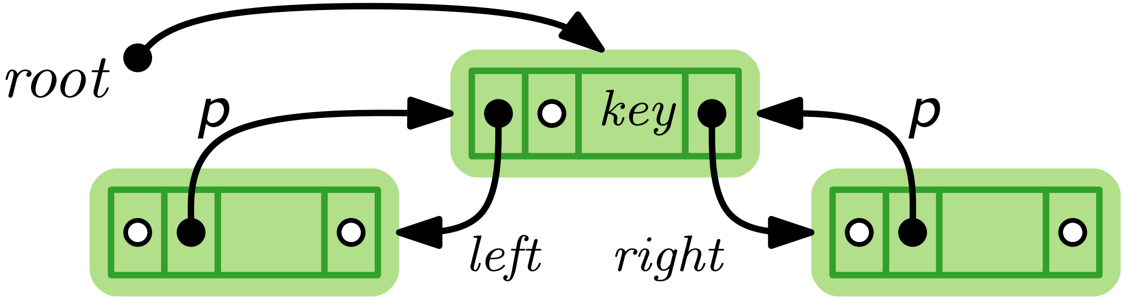
Code:

INORDERTREEWALK(Node $x = root$)

Aufgabe.
Geben Sie eine rekursive Implementierung an!



Inorder-Traversierung



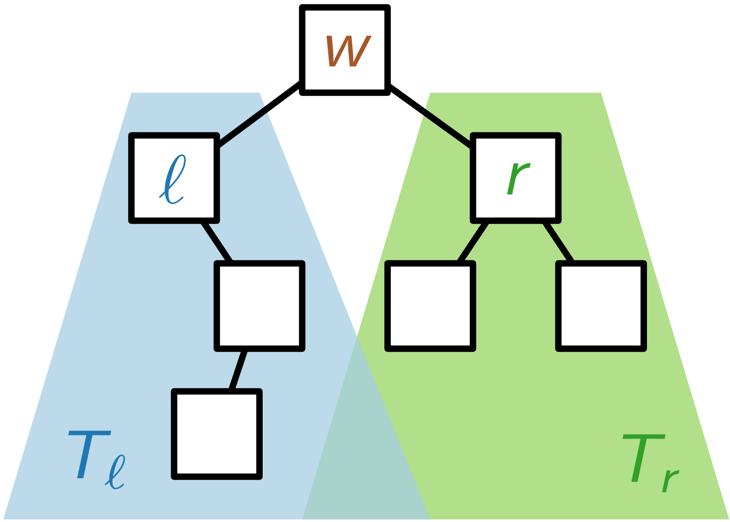
(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

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

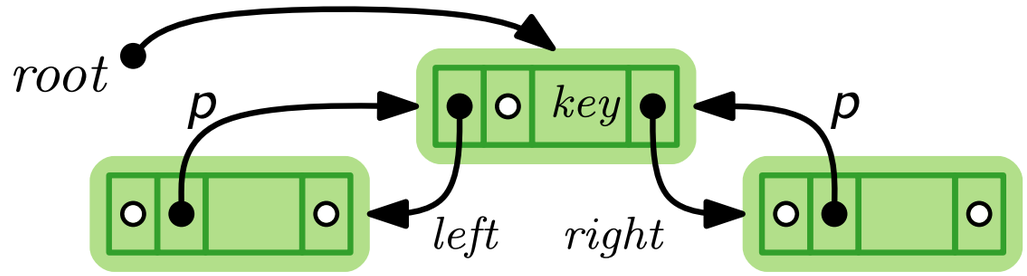
Lösung:
1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

InorderTreeWalk(Node $x = root$)
  if $x \neq nil$ **then**



Inorder-Traversierung



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

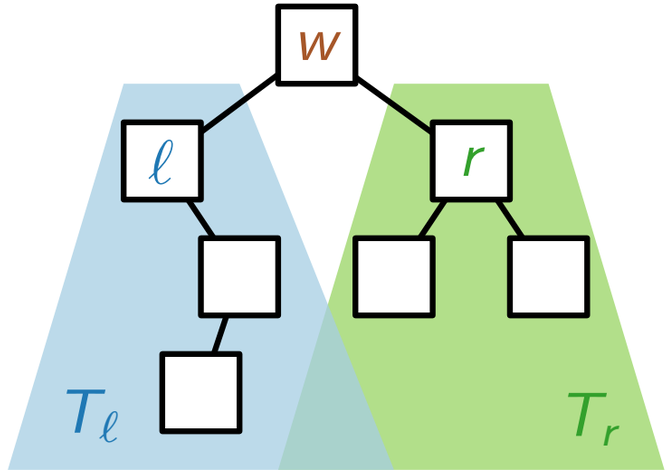
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

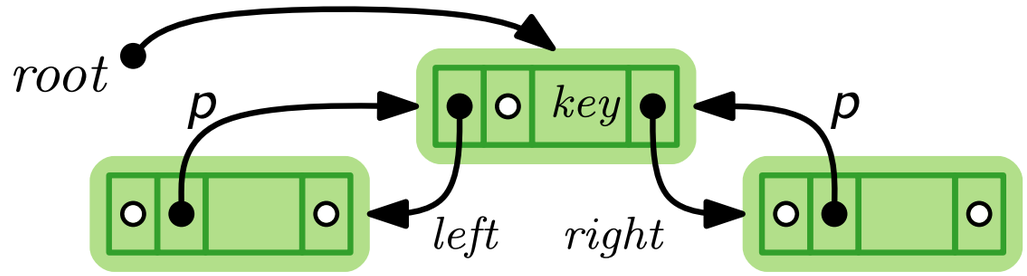
1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

```
InorderTreeWalk(Node x = root)
  if x ≠ nil then
    InorderTreeWalk(x.left)
```



Inorder-Traversierung



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

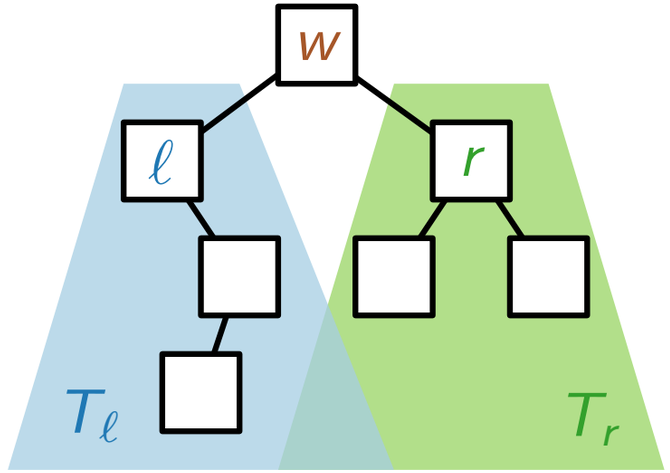
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

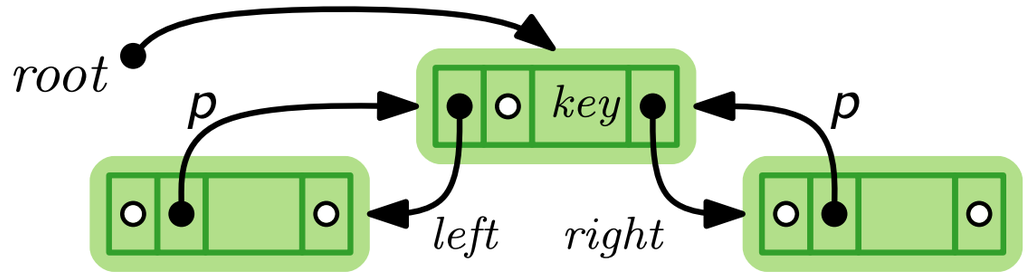
1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

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



Inorder-Traversierung



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

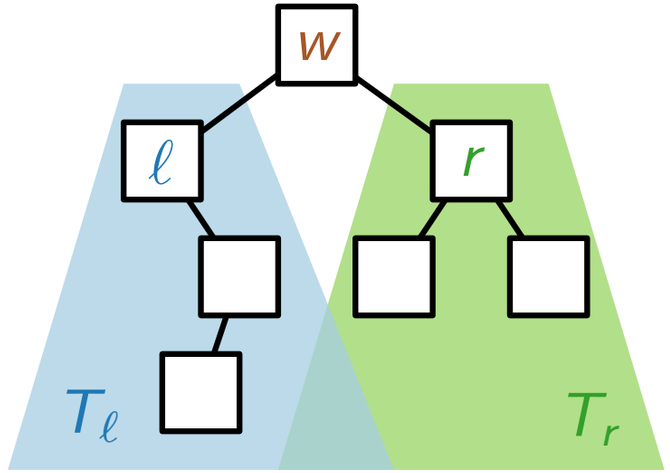
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

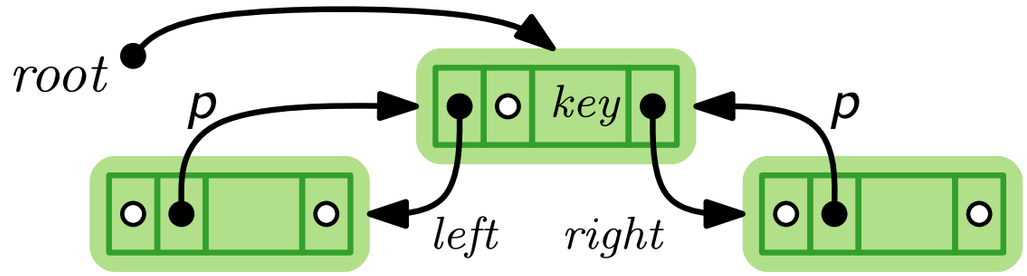
1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

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



Inorder-Traversierung



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

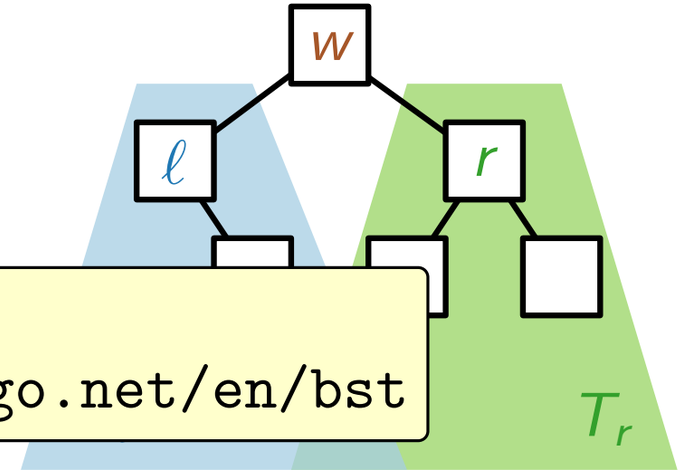
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

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)
```

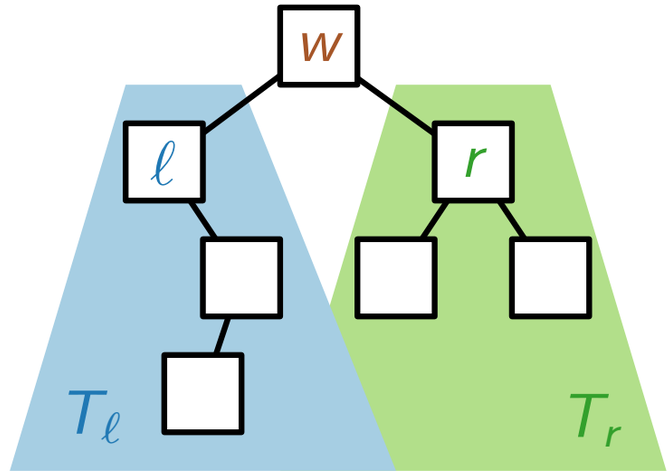


Demo.
https://visualgo.net/en/bst

Korrektheit

Code:

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

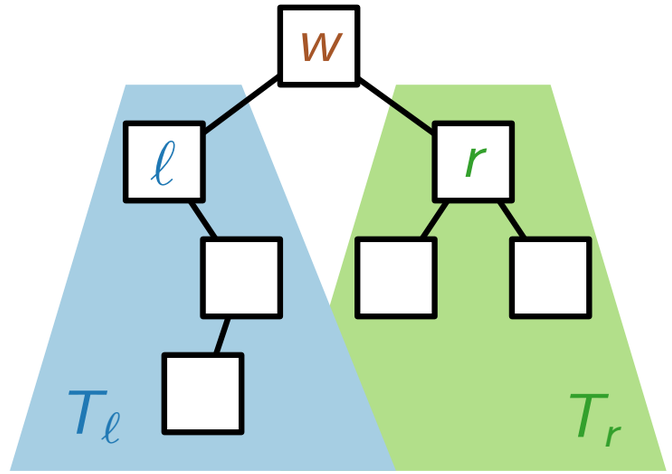


Korrektheit

Zu zeigen:

Code:

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

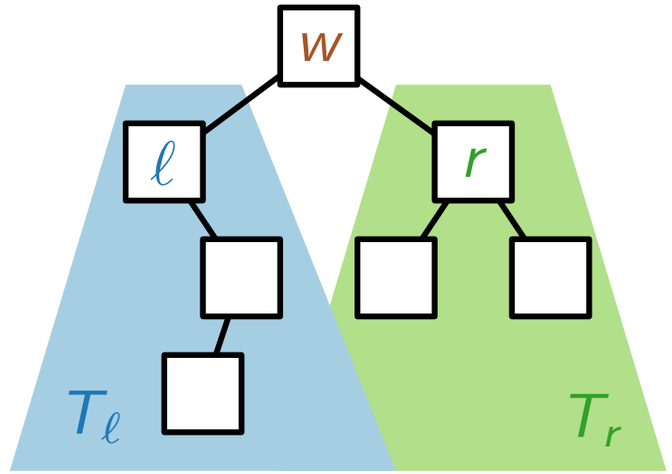


Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Code:

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

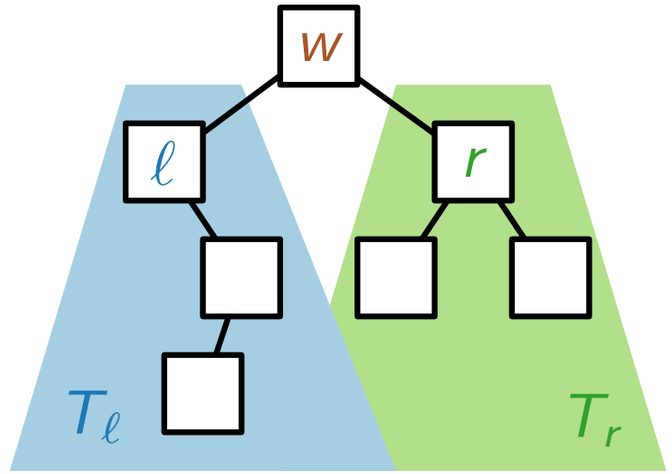


Korrektheit

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

Code:

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



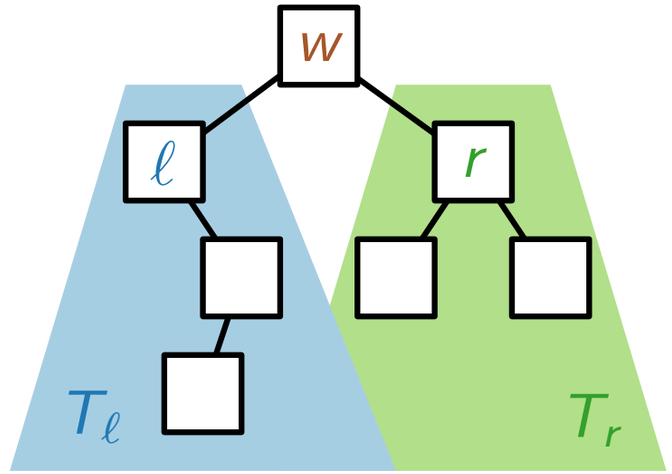
Korrektheit

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

$h = -1$:

Code:

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



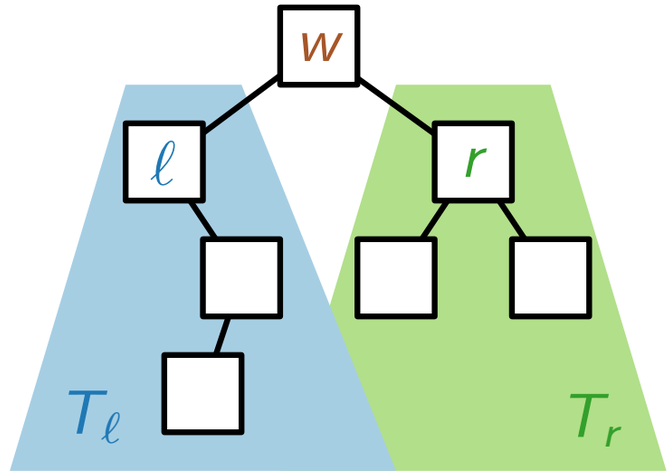
Korrektheit

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 ≠ nil then
    InorderTreeWalk(x.left)
    gib x.key aus
    InorderTreeWalk(x.right)
```



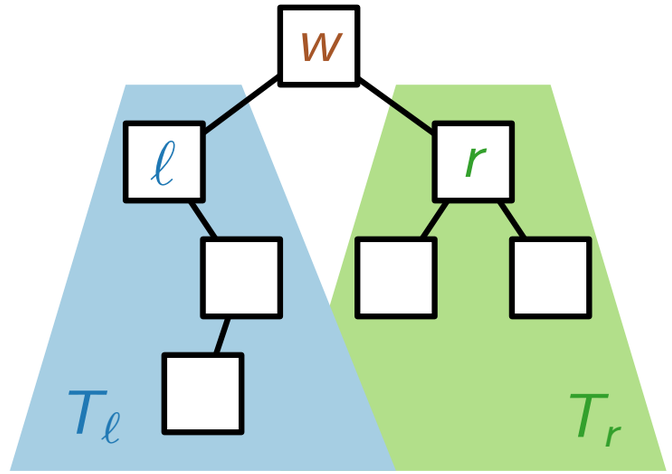
Korrektheit

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 ≠ nil then
    InorderTreeWalk(x.left)
    gib x.key aus
    InorderTreeWalk(x.right)
```



Korrektheit

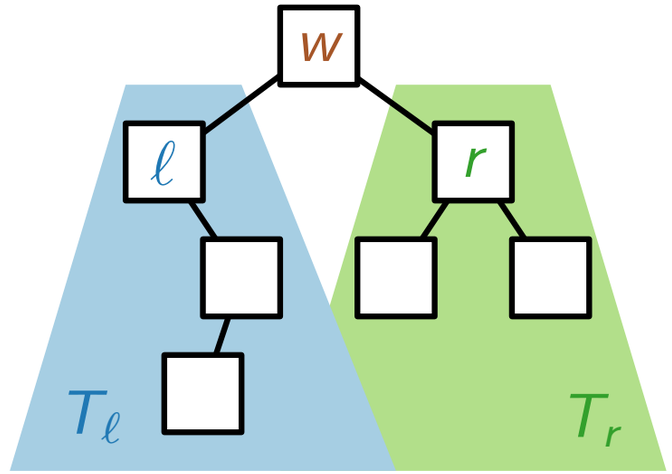
Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h.

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

$h \geq 0$:

Code:

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



Korrektheit

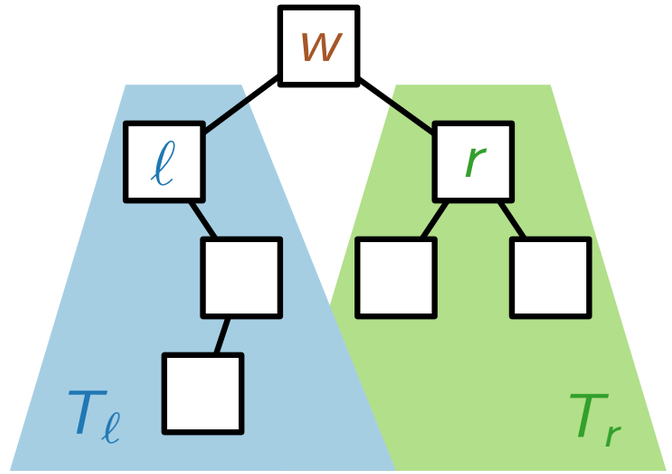
Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h.

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

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

Code:

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



Korrektheit

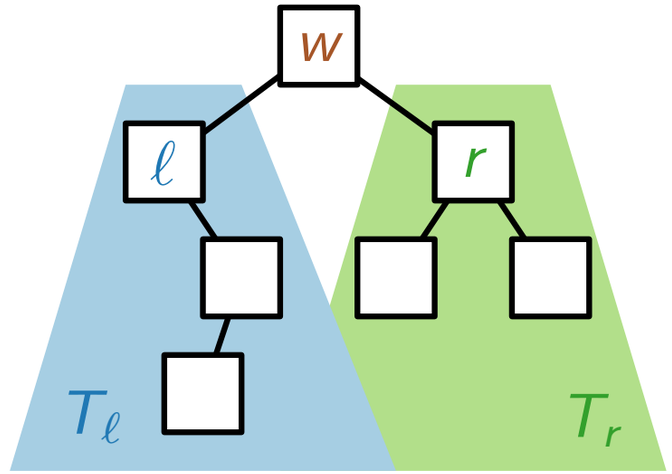
Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h.

$h = -1$: 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 ≠ 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. $root = nil$ ✓

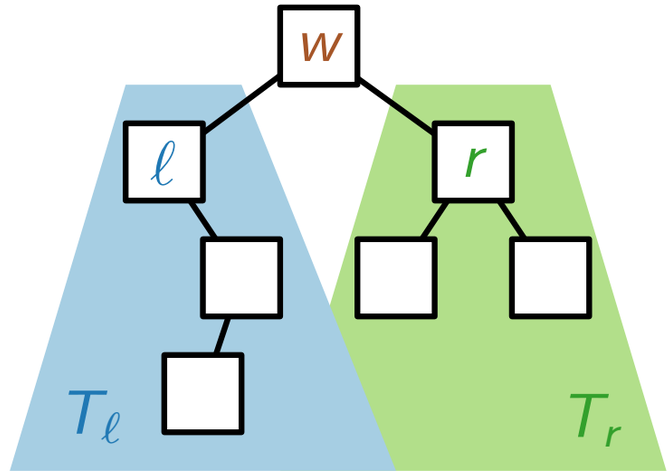
$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

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 ≠ 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. $root = nil$ ✓

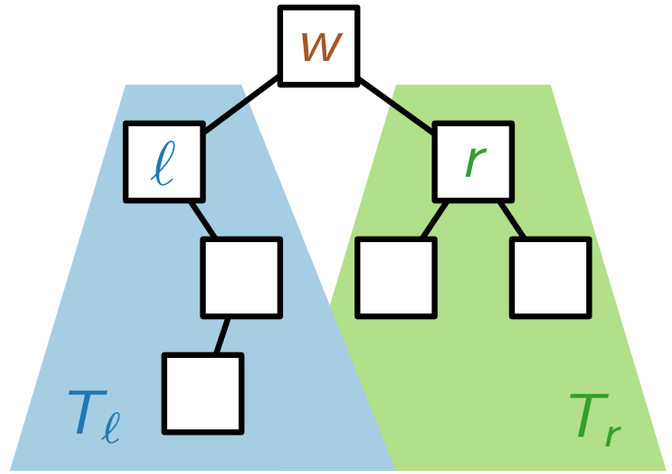
$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

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 ≠ 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 $< h$.

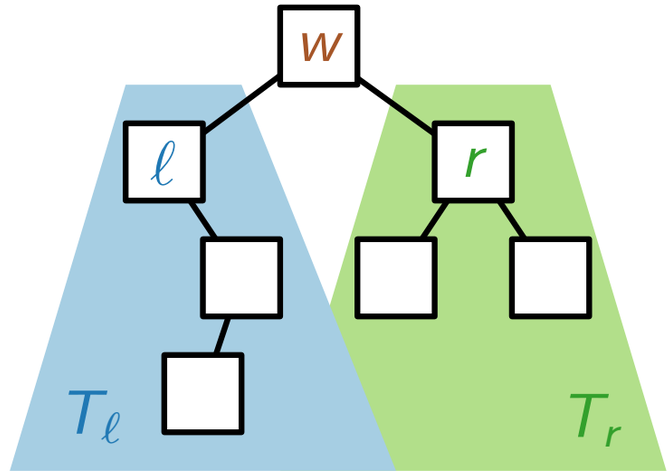
Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

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 ≠ 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

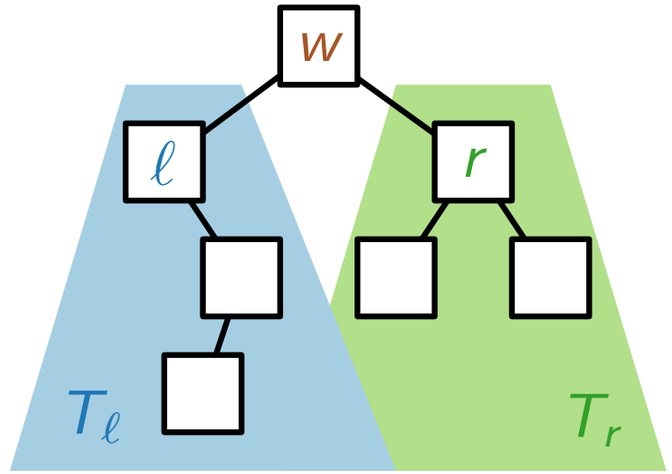
Also werden *ihre* Schlüssel sortiert ausgegeben.

$\Rightarrow$

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

Code:

```
InorderTreeWalk(Node x = root)
  if x ≠ 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

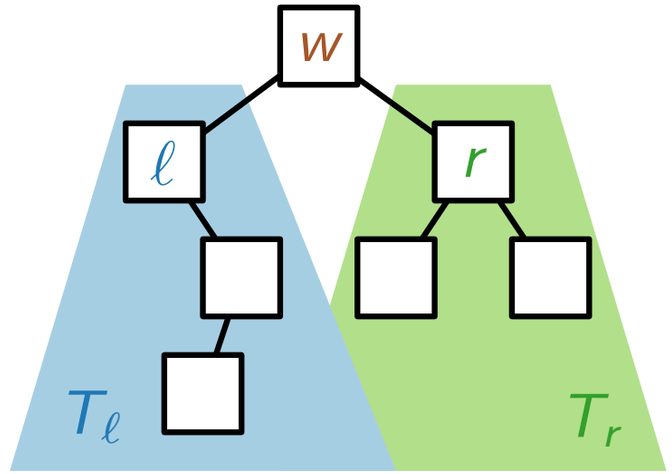
Also werden *ihre* Schlüssel sortiert ausgegeben.

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 ≠ 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

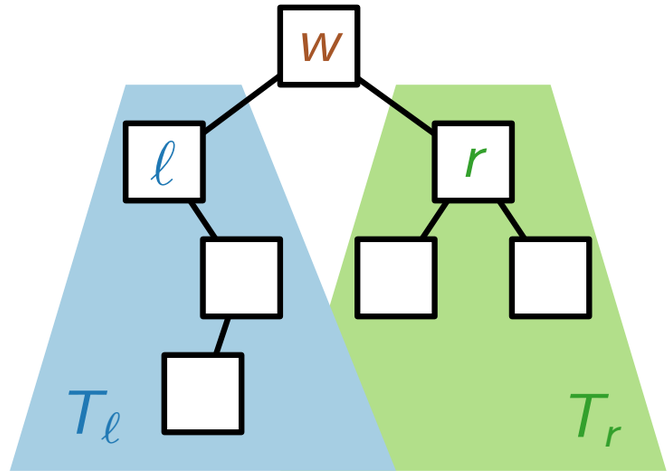
Also werden *ihre* Schlüssel sortiert ausgegeben.

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 ≠ nil then
    InorderTreeWalk(x.left)
    gib x.key aus
    InorderTreeWalk(x.right)
```

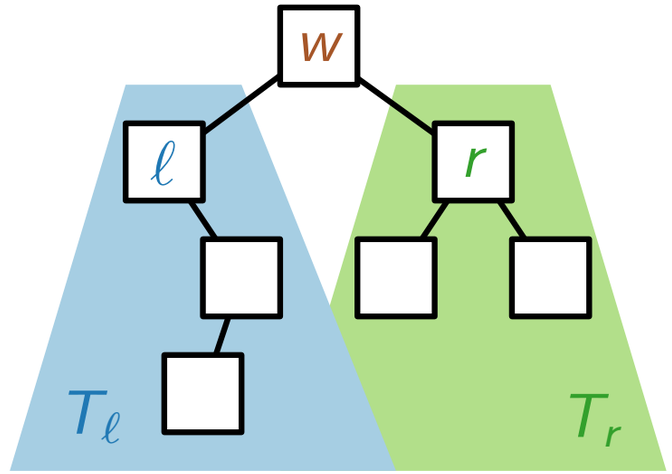


Laufzeit

$$T(n) =$$

Code:

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



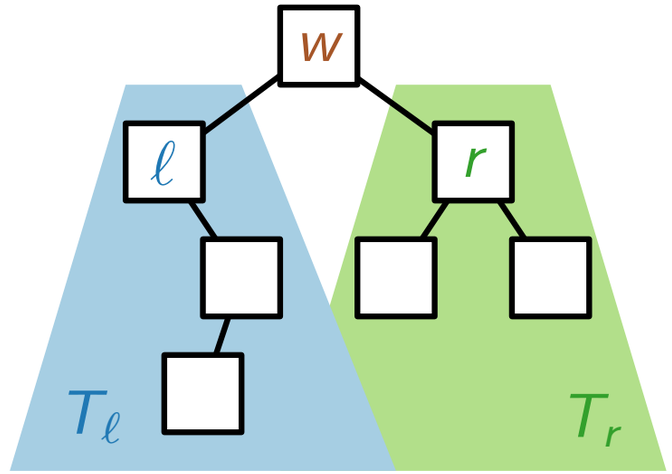
Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

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

Code:

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



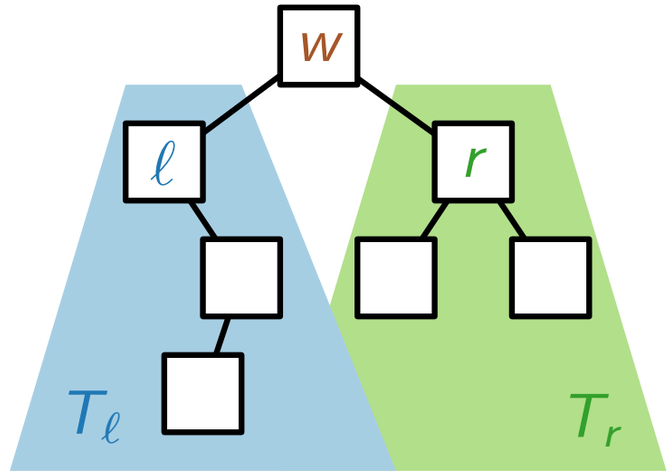
Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

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

Code:

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



Laufzeit

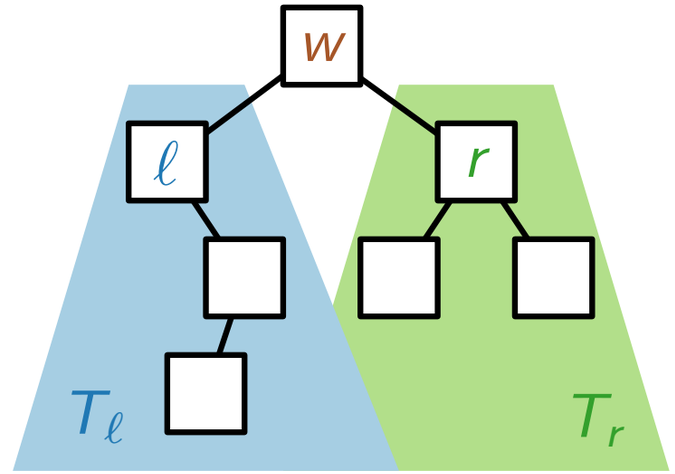
Anz. der Knoten im linken / rechten Teilbaum der Wurzel

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

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

Code:

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



Laufzeit

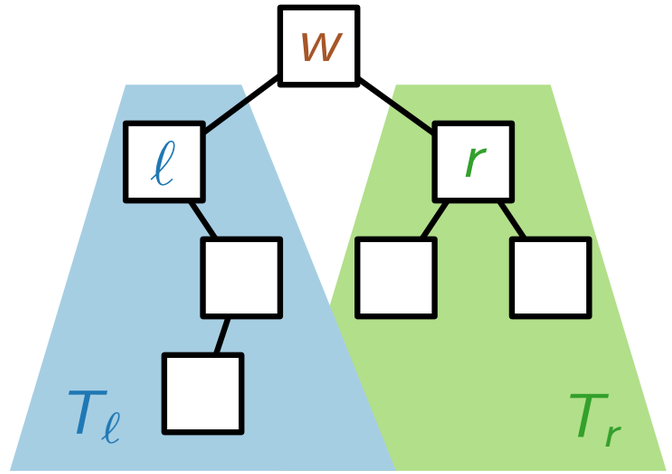
Anz. der Knoten im linken / rechten Teilbaum der Wurzel

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

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

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

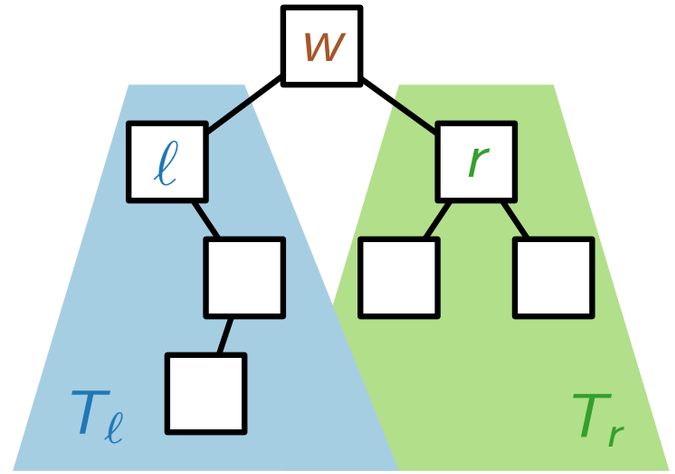
$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n-k-1) + 1 & \text{sonst.} \end{cases}$$

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

oder:

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

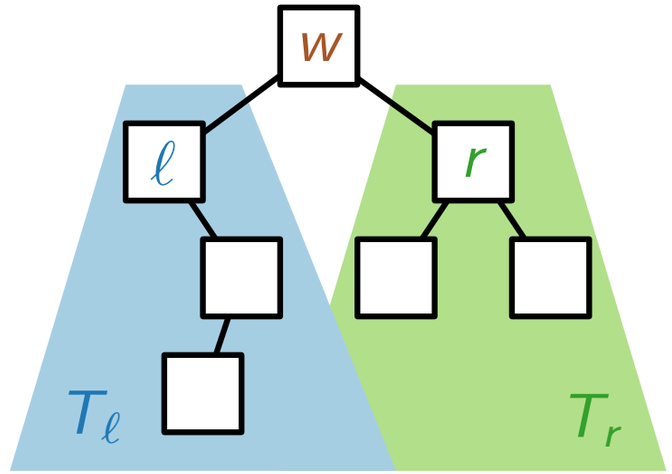
$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n-k-1) + 1 & \text{sonst.} \end{cases}$$

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.

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

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

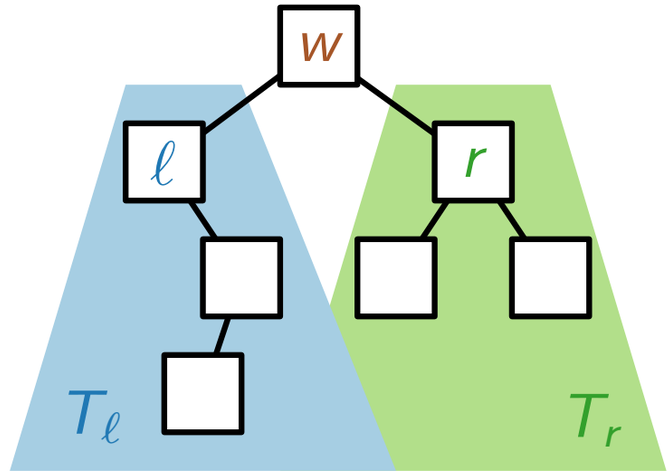
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n \; -1$

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

Für Bäume gilt: #Kanten $=$

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

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

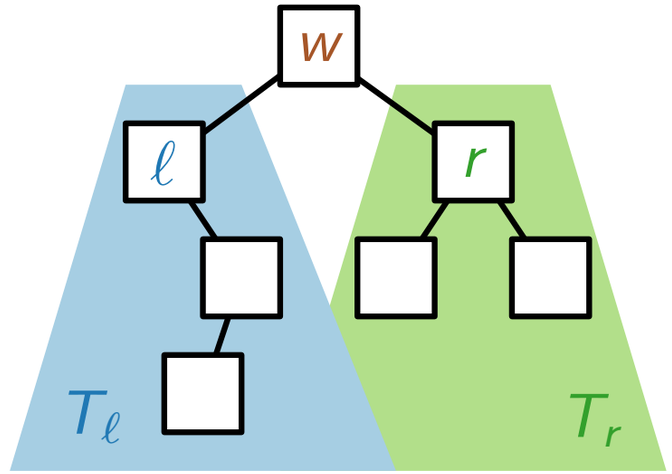
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n \; -1$

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

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

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) $T(n) \leq c \cdot n \; -1$

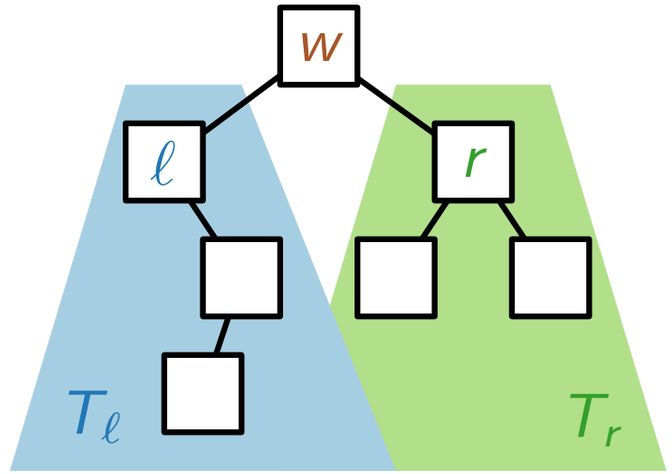
oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anzahl von Schritten aus.

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

Beweis durch Induktion

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

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

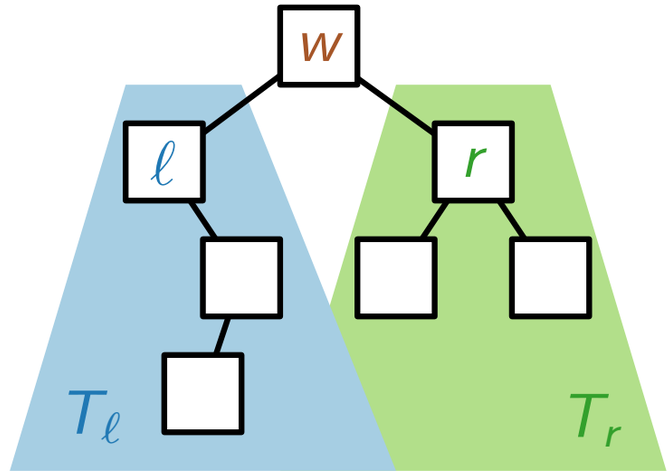
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n \; -1$

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

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

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

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

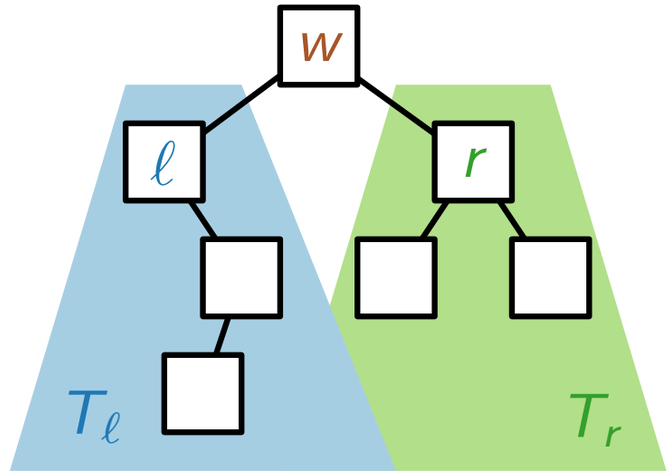
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n \; -1$

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

Für Bäume gilt: #Kanten $=$ #Knoten $-1 \; = \; n-1$

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) $T(n) \leq c \cdot n \; -1$

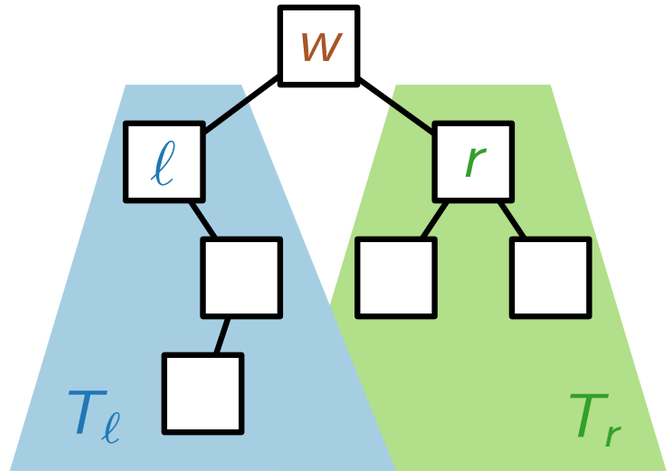
oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anzahl von Schritten aus.

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

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

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) $T(n) \leq c \cdot n \; -1$

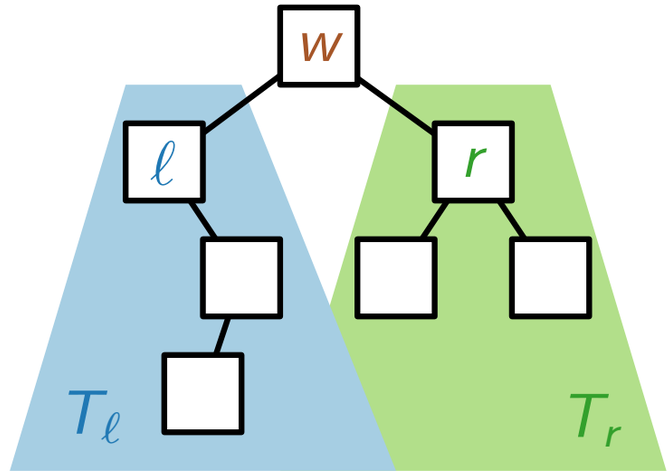
oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anzahl von Schritten aus.

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:

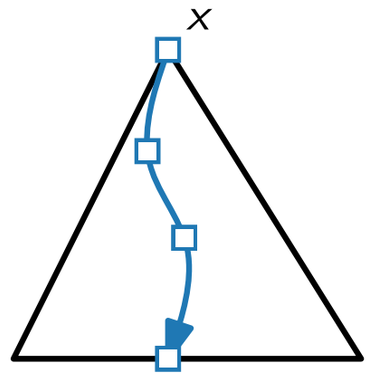
```
InorderTreeWalk(Node x = root)
  if x ≠ nil then
    InorderTreeWalk(x.left)
    gib x.key aus
    InorderTreeWalk(x.right)
```



Suche

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

Aufgabe.
Schreiben Sie Pseudocode für die rekursive Methode

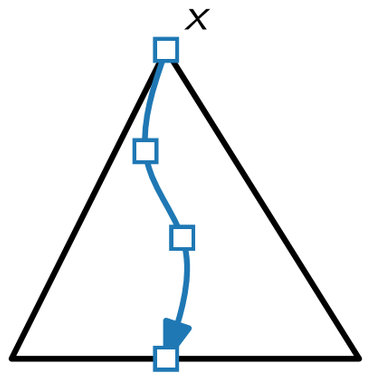


Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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
```

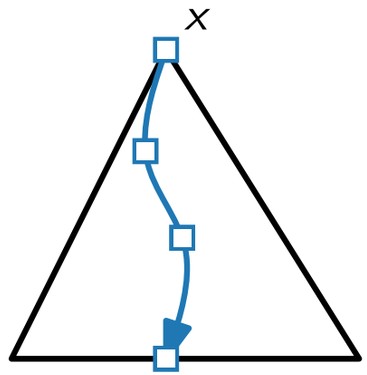


Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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
```

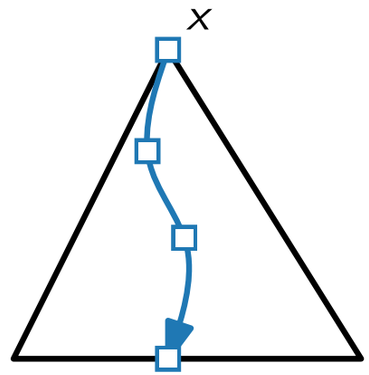


Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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
```

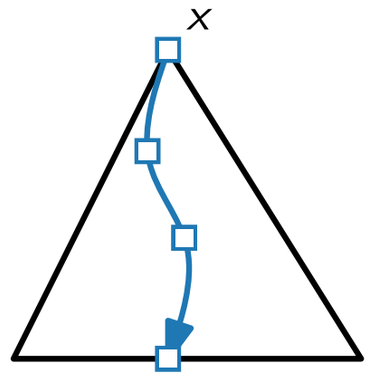


Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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)
```

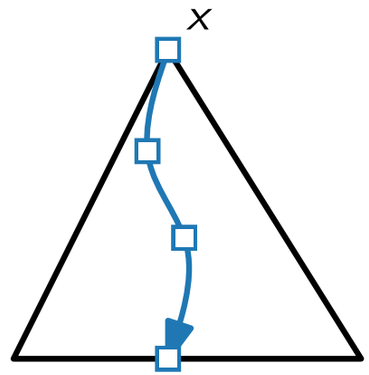


Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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)
```

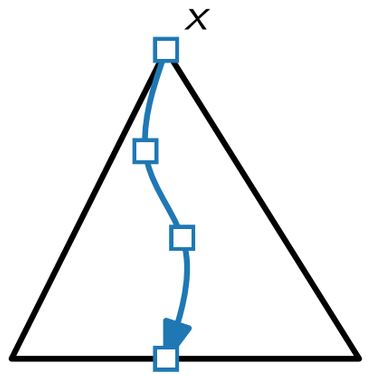


Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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)
```



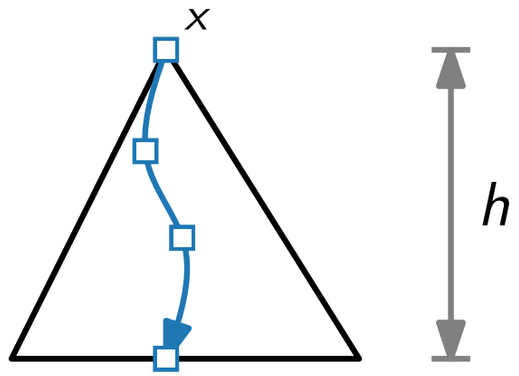
Laufzeit:

Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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)
```



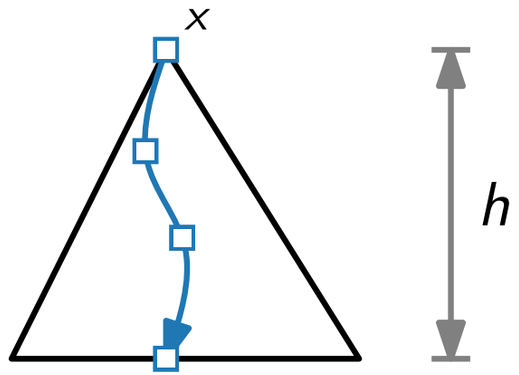
Laufzeit:

Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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: $\mathcal{O}(h)$

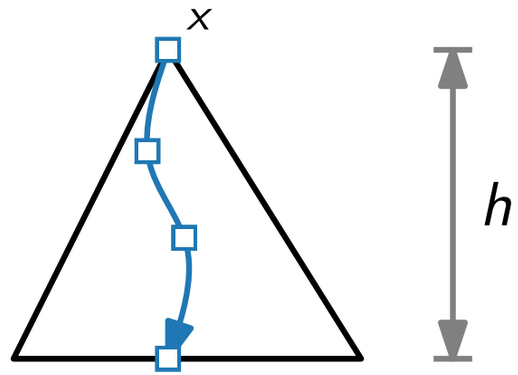
Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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: $\mathcal{O}(h)$

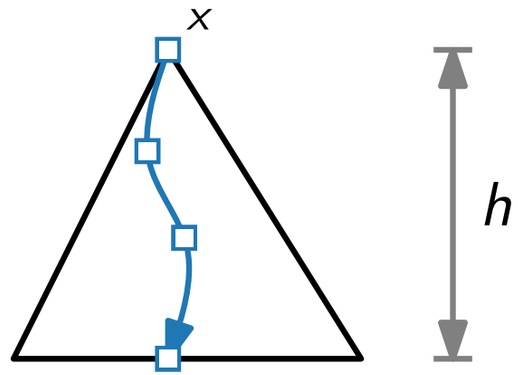
Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ 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)
```



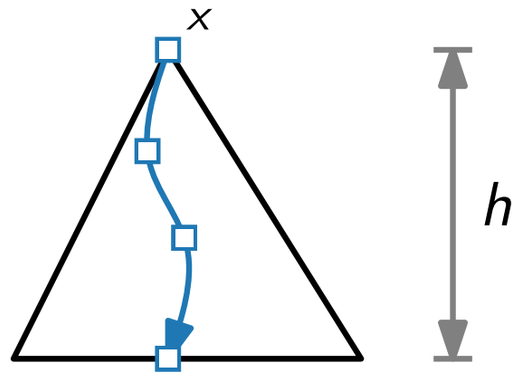
Laufzeit: $\mathcal{O}(h)$

iterativ

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: $\mathcal{O}(h)$

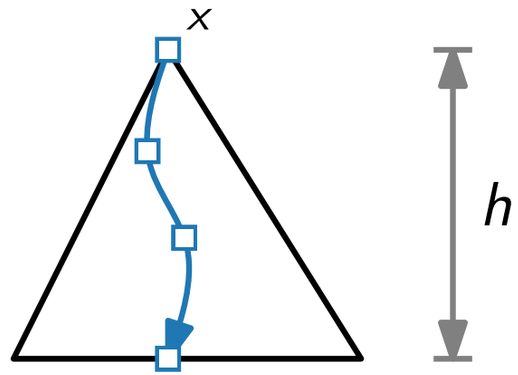
iterativ

```
while x ≠ nil and x.key ≠ k do
  
  
  
return x
```

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: $\mathcal{O}(h)$

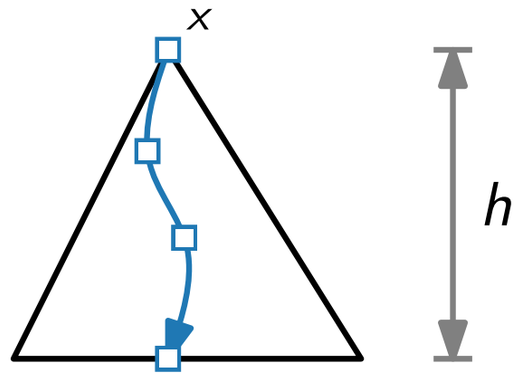
iterativ

```
while x ≠ nil and x.key ≠ k do
    if k < x.key then
        
return x
```

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: $\mathcal{O}(h)$

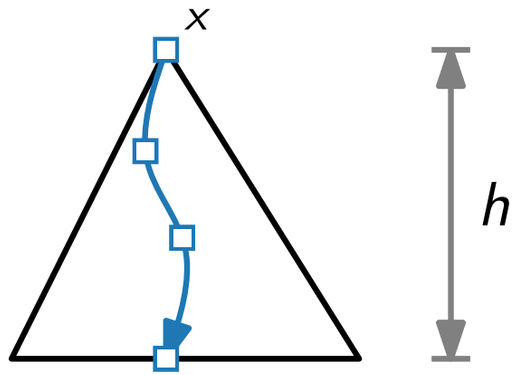
iterativ

```
while x ≠ nil and x.key ≠ k do
    if k < x.key then
        x = x.left
return x
```

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: $\mathcal{O}(h)$

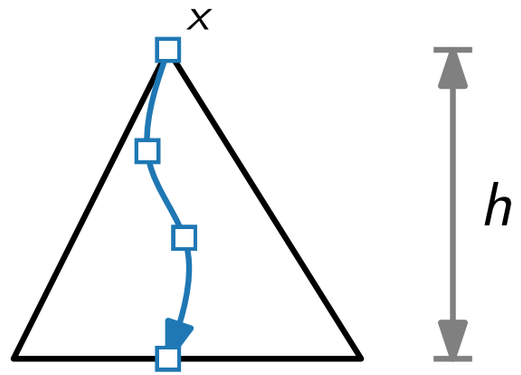
iterativ

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

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: $\mathcal{O}(h)$

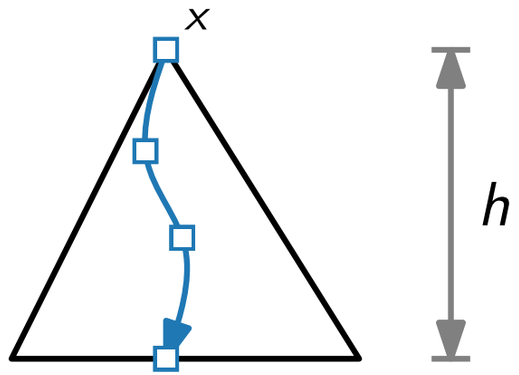
iterativ

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

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: $\mathcal{O}(h)$

iterativ

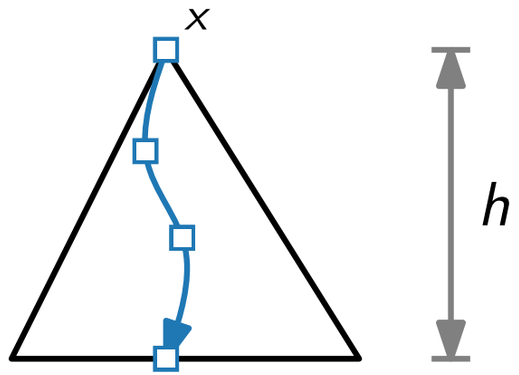
```
while x ≠ nil and x.key ≠ k do
    if k < x.key then
        x = x.left
    else x = x.right
return x
```

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)
```



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

iterativ

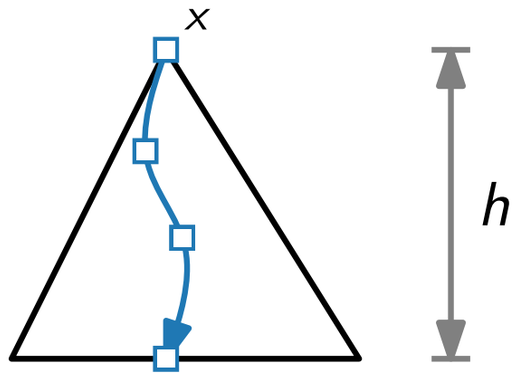
```
while x ≠ nil and x.key ≠ k do
    if k < x.key then
        x = x.left
    else x = x.right
return x
```

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)
```



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

iterativ

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

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

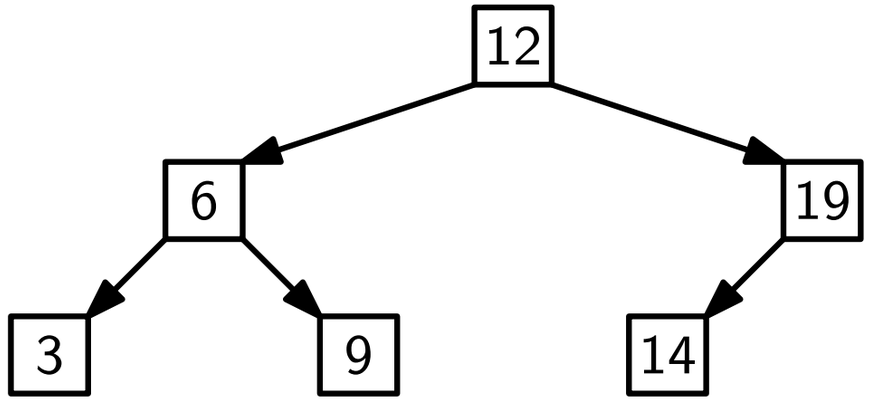
Trotzdem schneller, da keine Verwaltung der rekursiven Methodenaufrufe.

Minimum & Maximum

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

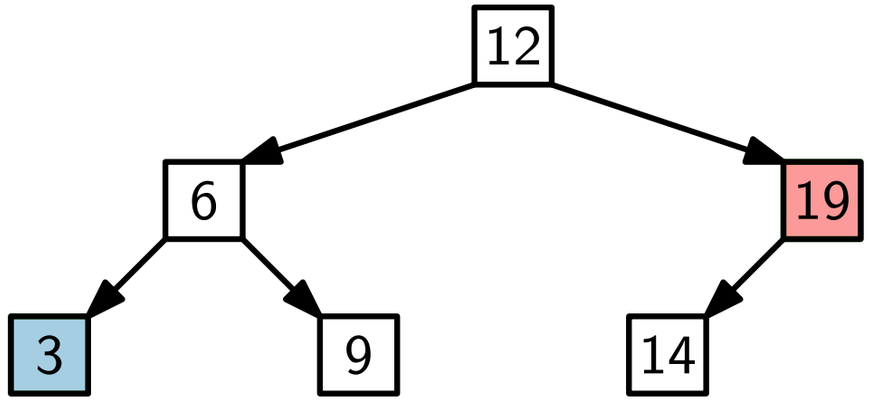
Minimum & Maximum

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



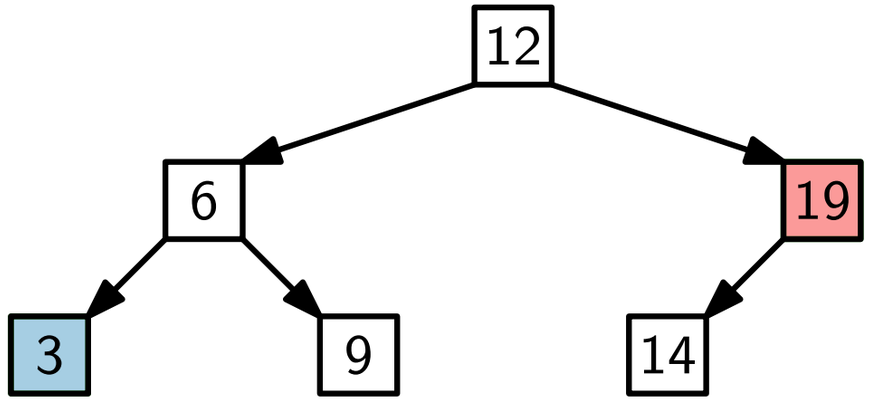
Minimum & Maximum

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



Minimum & Maximum

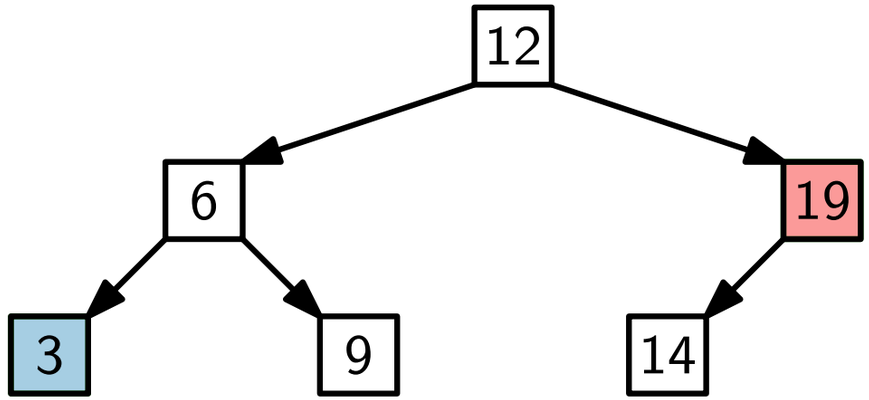
Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



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!

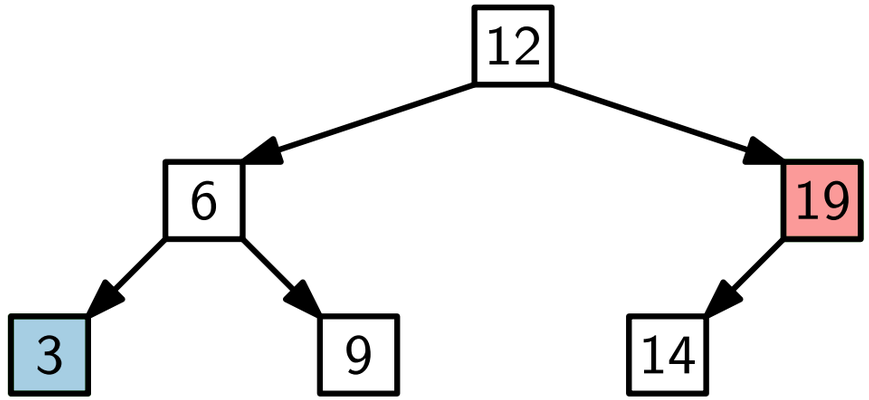
iterativ

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!

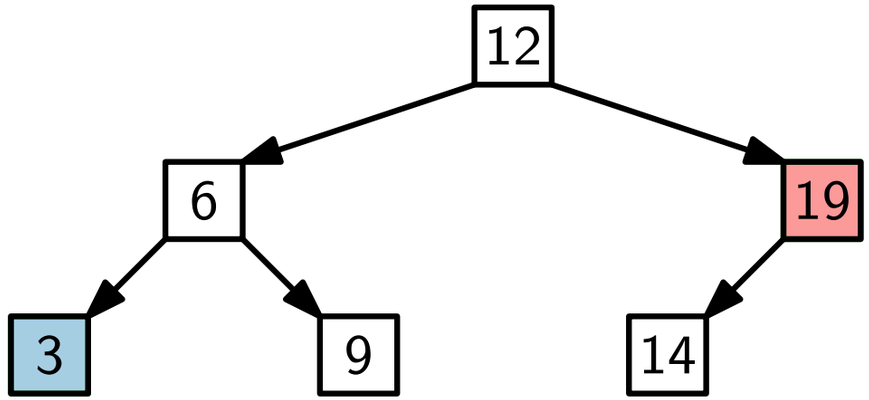
iterativ

```
Node MINIMUM(Node x = root)

  while x.left ≠ 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

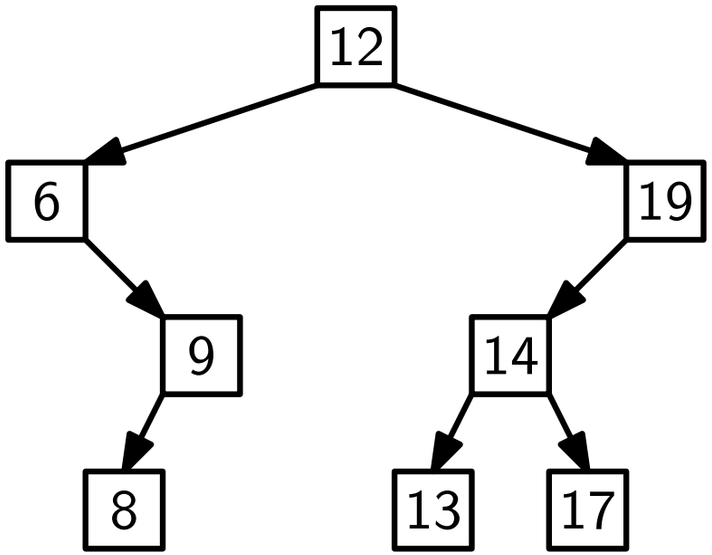
```
Node MINIMUM(Node x = root)
  if x == nil then return nil
  while x.left ≠ nil do
  ⌊    x = x.left
  return x
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Nachfolger (und Vorgänger)

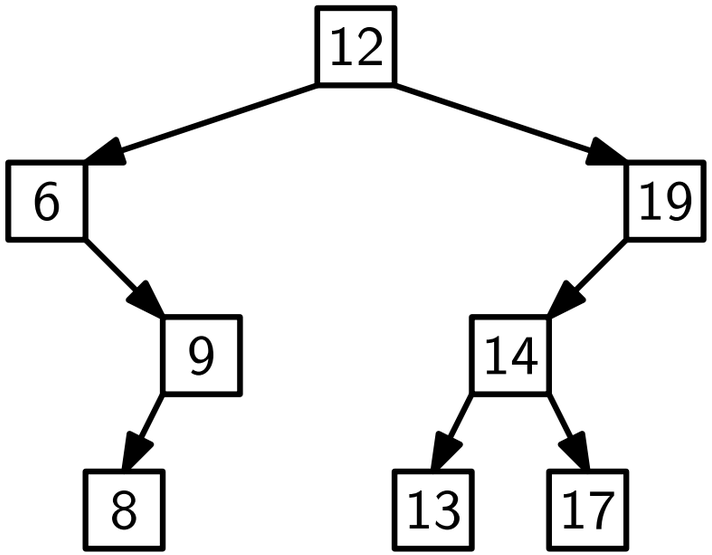
Vereinfachende Annahme: alle Schlüssel sind verschieden.



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

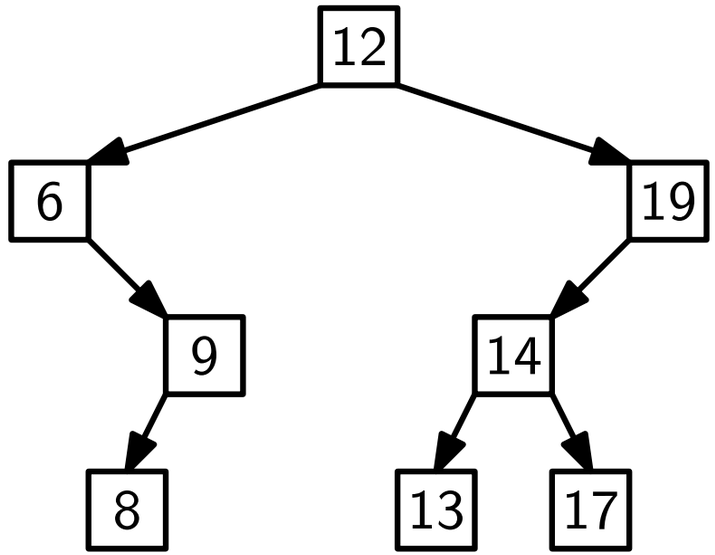
Erinnerung: $\textsc{Nachfolger}(x) =$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

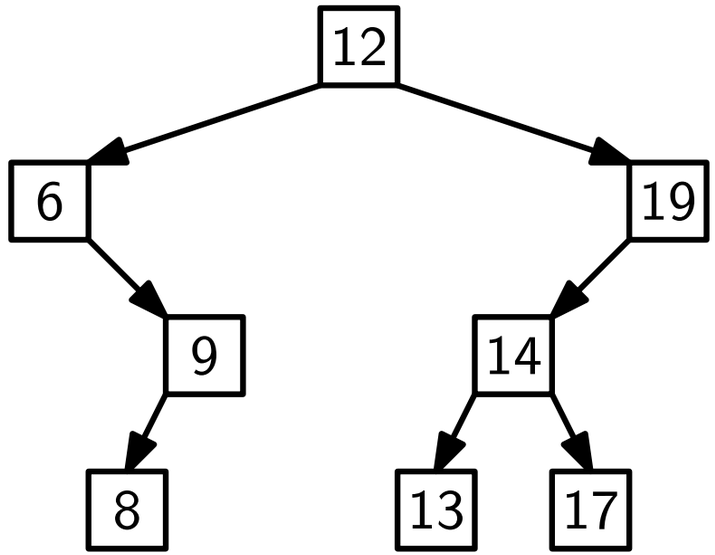


Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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

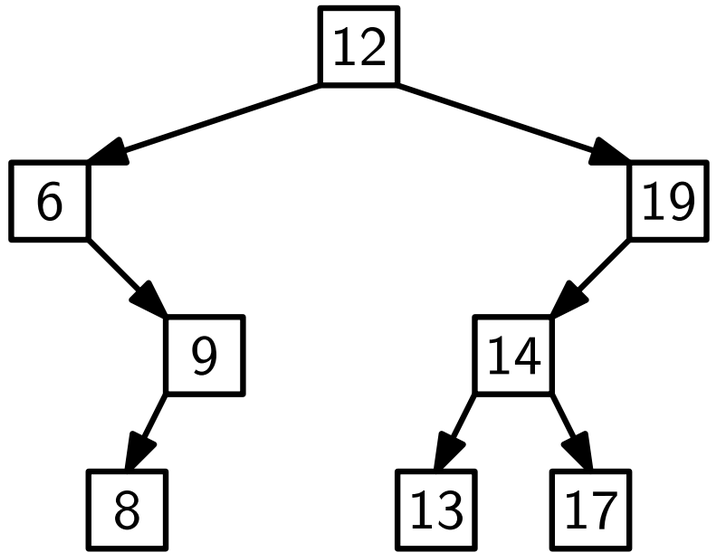


Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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

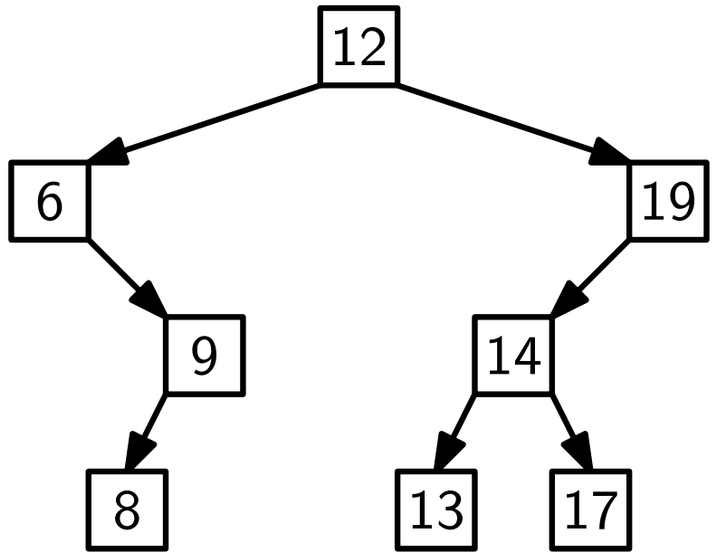


Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



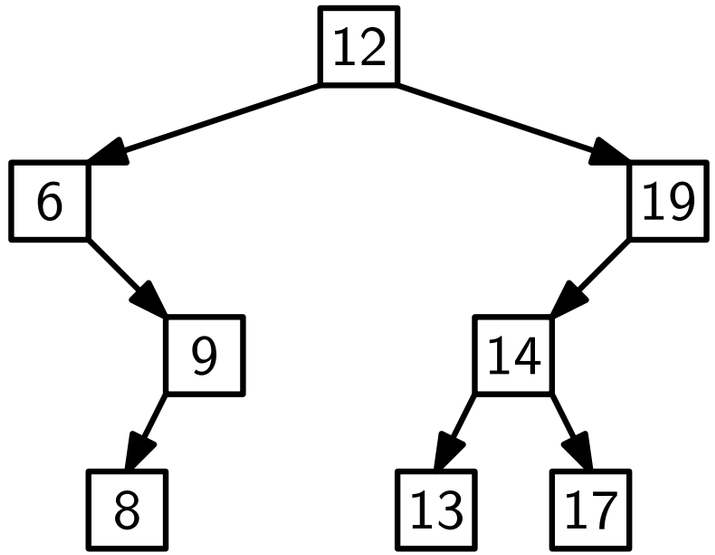
Nachfolger(19) $= nil$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



Nachfolger(19) $= nil$

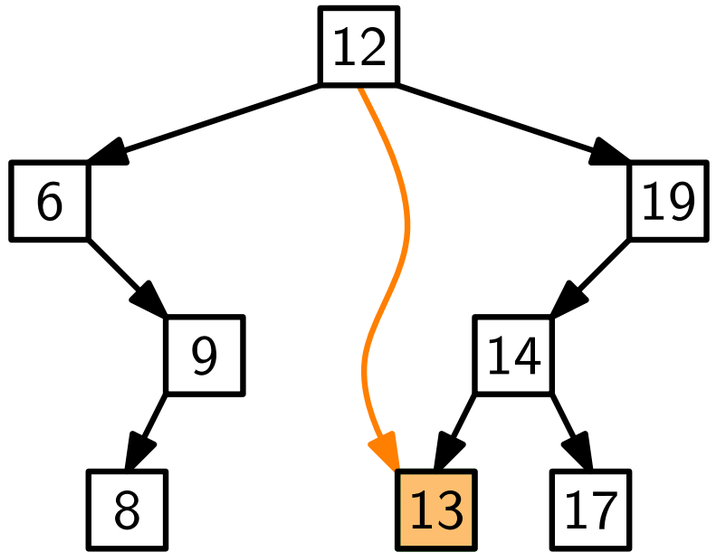
Nachfolger(12) $=$?

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



Nachfolger(19) $= nil$

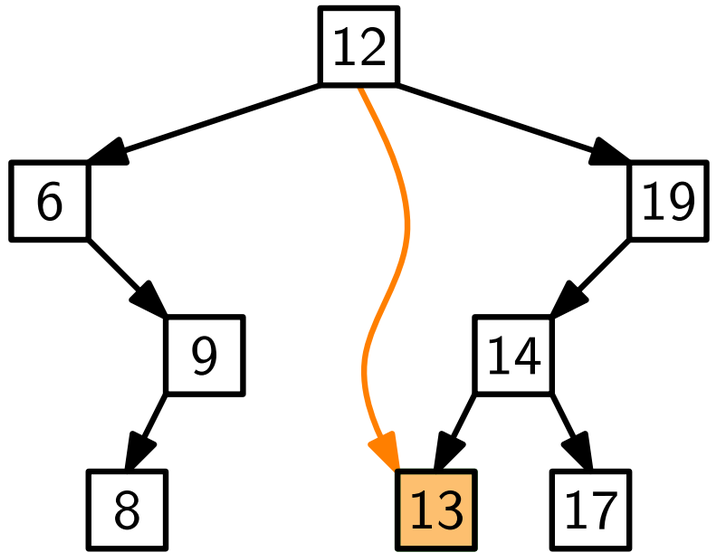
Nachfolger(12) $=$?

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



Nachfolger(19) $= nil$

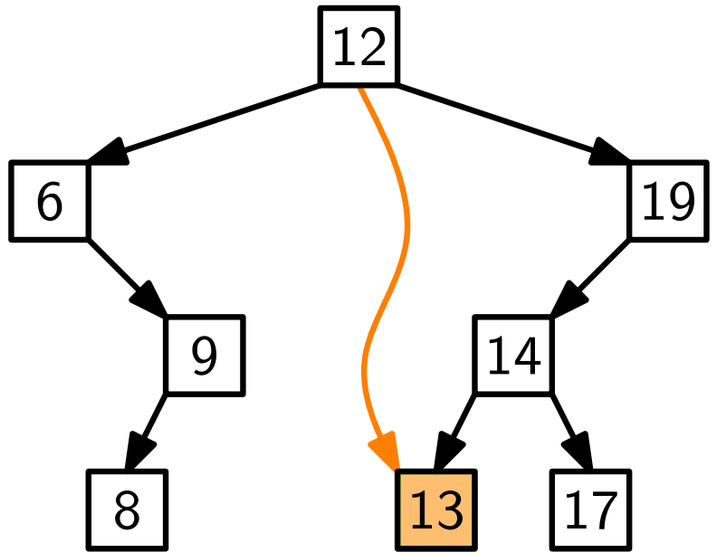
Nachfolger(12) $= 13 = \textsc{Minimum}(12.right)$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



Nachfolger(19) $= nil$

Nachfolger(12) $= 13 = \textsc{Minimum}(12.right)$

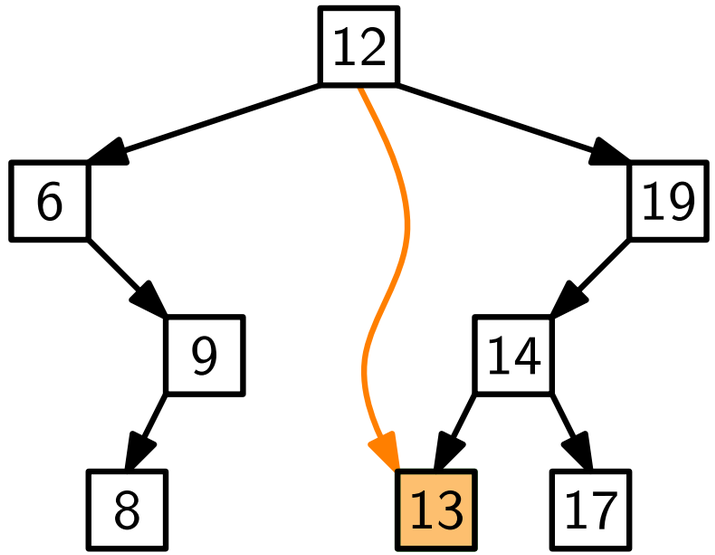
Nachfolger(9) $=$?

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



Nachfolger(19) $= nil$

Nachfolger(12) $= 13 = \textsc{Minimum}(12.right)$

Nachfolger(9) $=$?

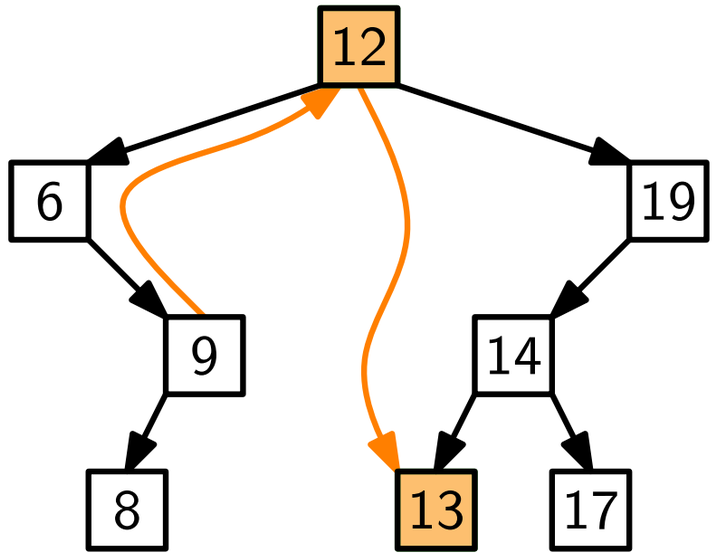
9 hat kein rechtes Kind; $9 = \textsc{Maximum}(12.left)$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



Nachfolger(19) $= nil$

Nachfolger(12) $= 13 = \textsc{Minimum}(12.right)$

Nachfolger(9) $=$?

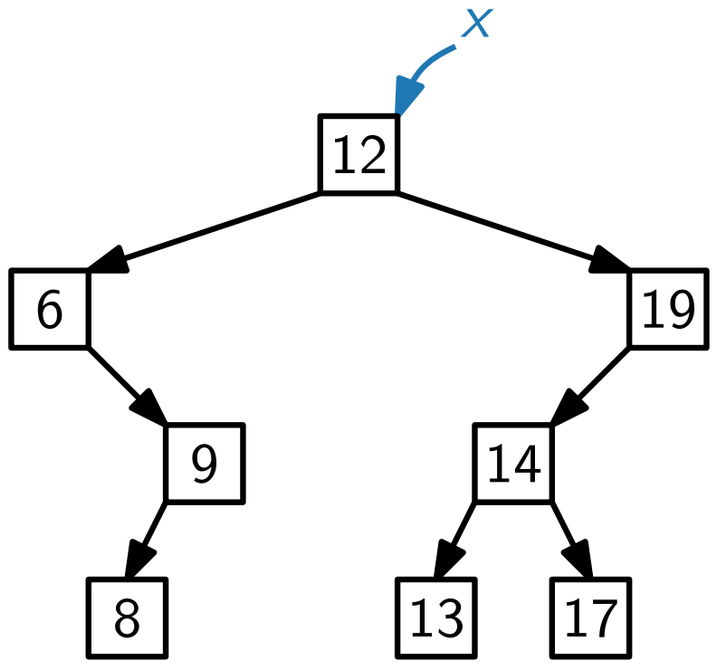
9 hat kein rechtes Kind; $9 = \textsc{Maximum}(12.left)$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



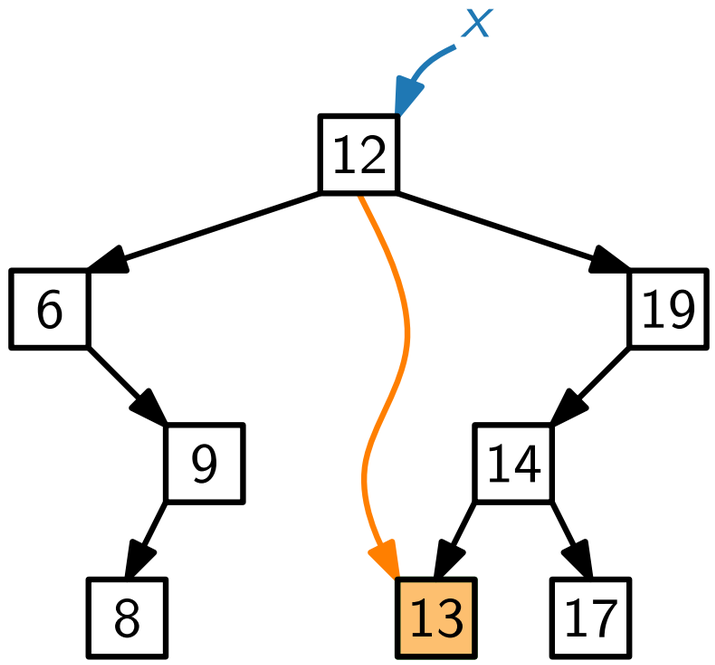

```
Node Successor(Node x)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



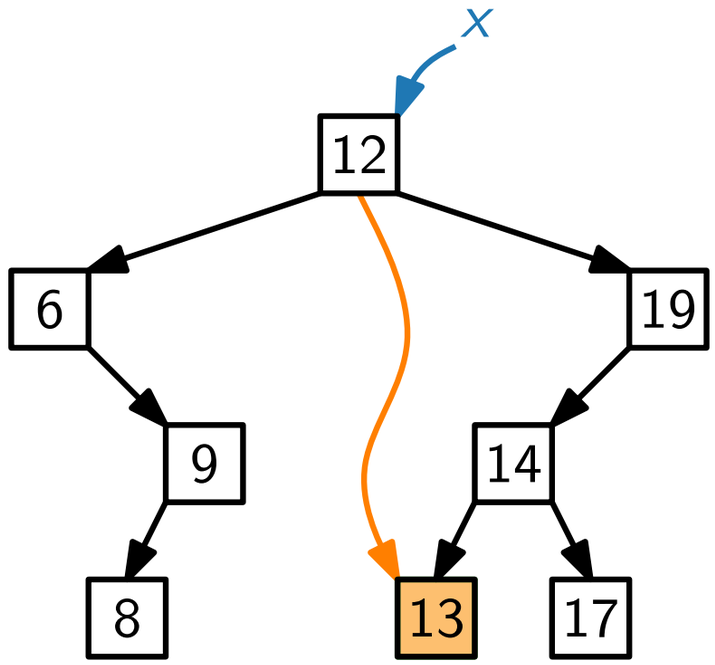

```
Node Successor(Node x)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



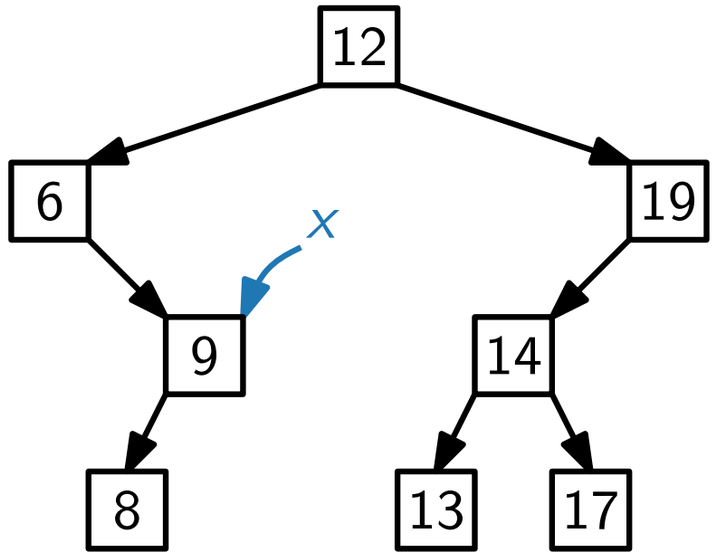

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



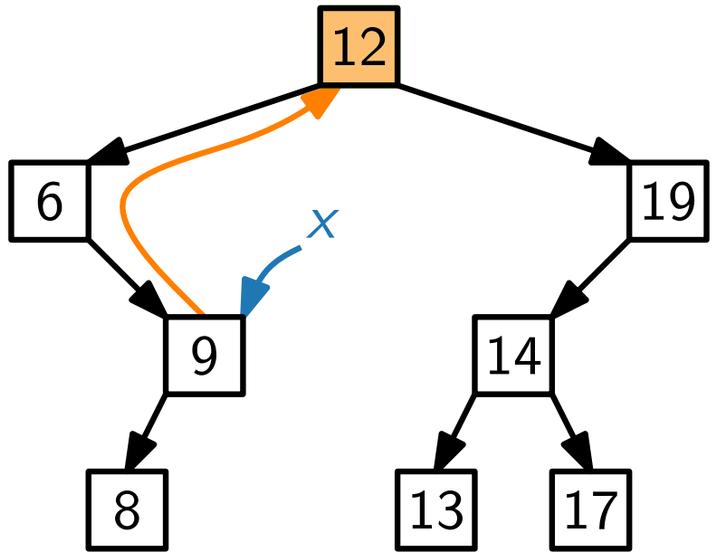

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



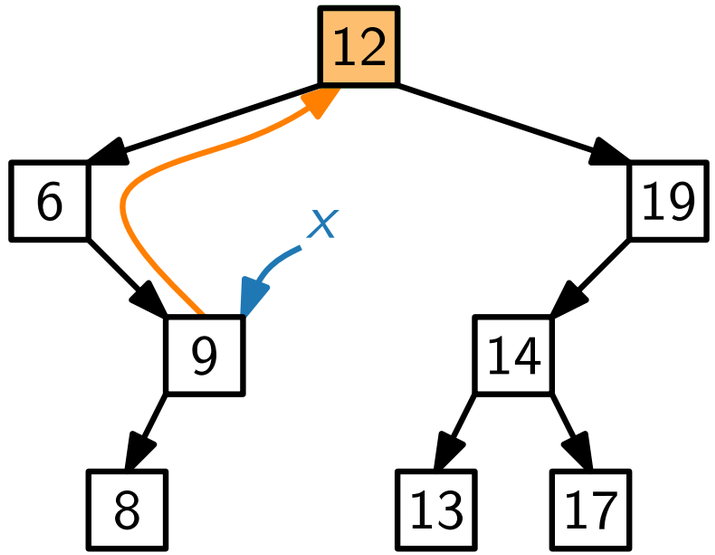

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



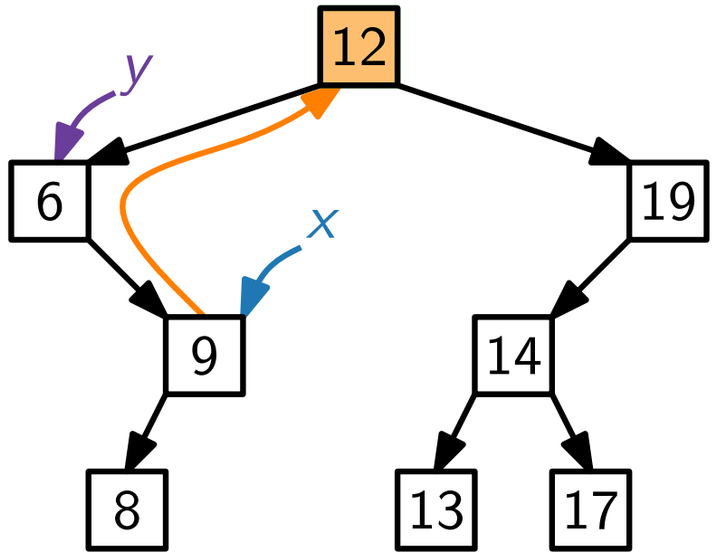

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



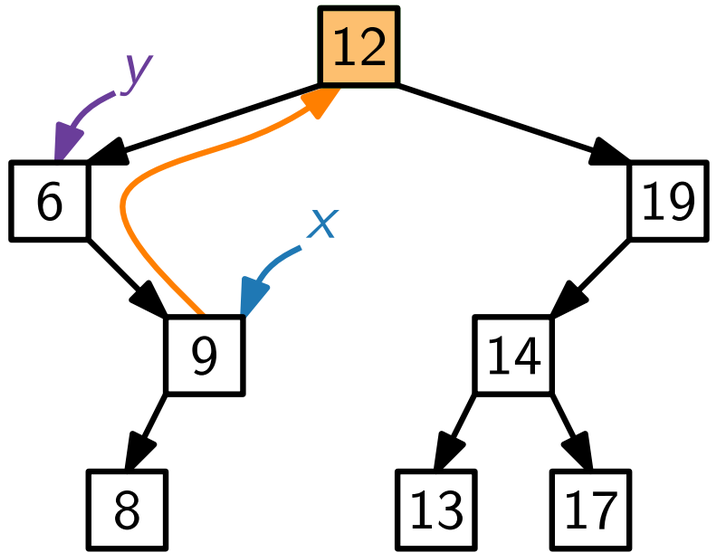

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



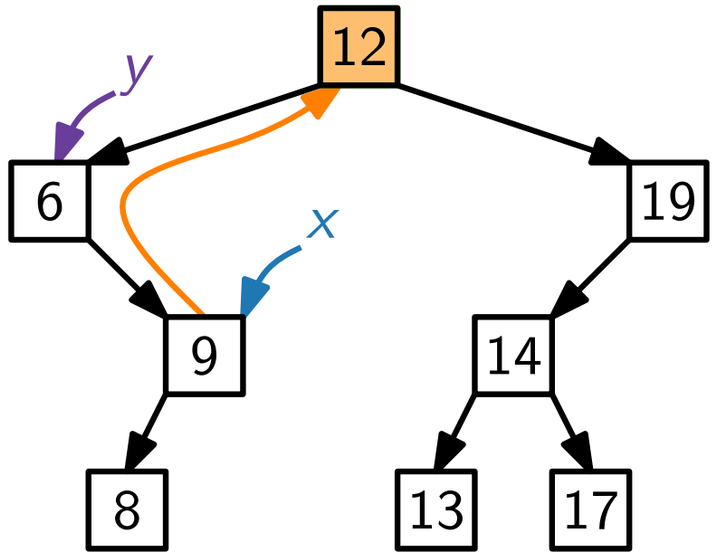

```
Node Successor(Node x)
    if x.right ≠ nil then
        return Minimum(x.right)
    y = x.p
    while y ≠ nil and x == y.right do
        
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



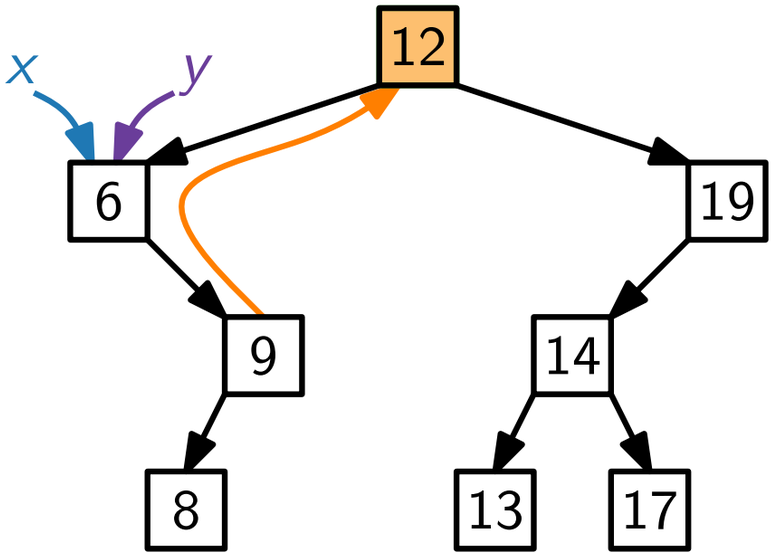

```
Node Successor(Node x)
    if x.right ≠ nil then
        return Minimum(x.right)
    y = x.p
    while y ≠ nil and x == y.right do
        x = y
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



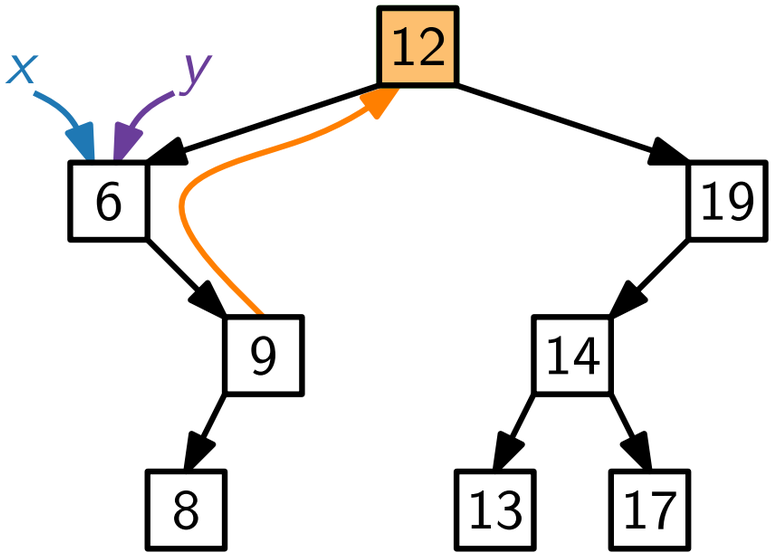

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



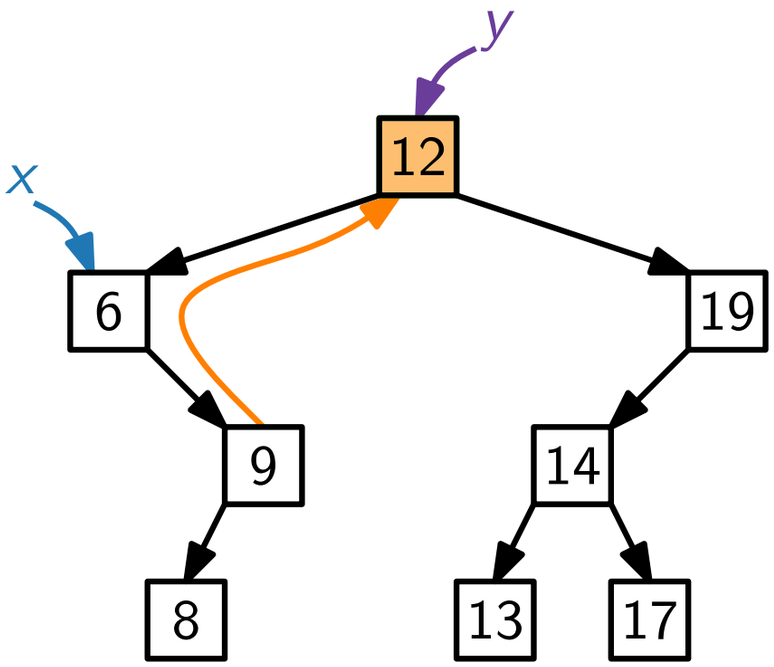

```
NODE SUCCESSOR(NODE x)
  if x.right ≠ nil then
    return MINIMUM(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



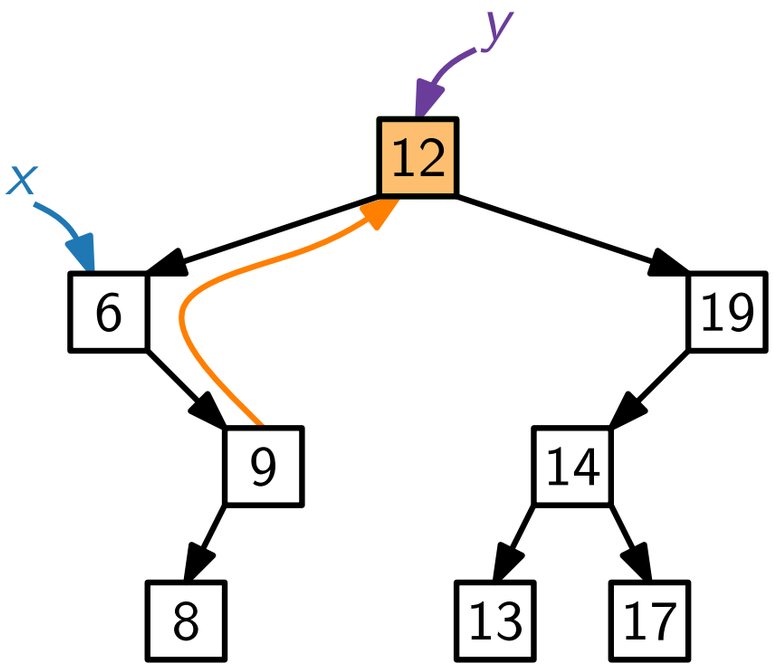

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



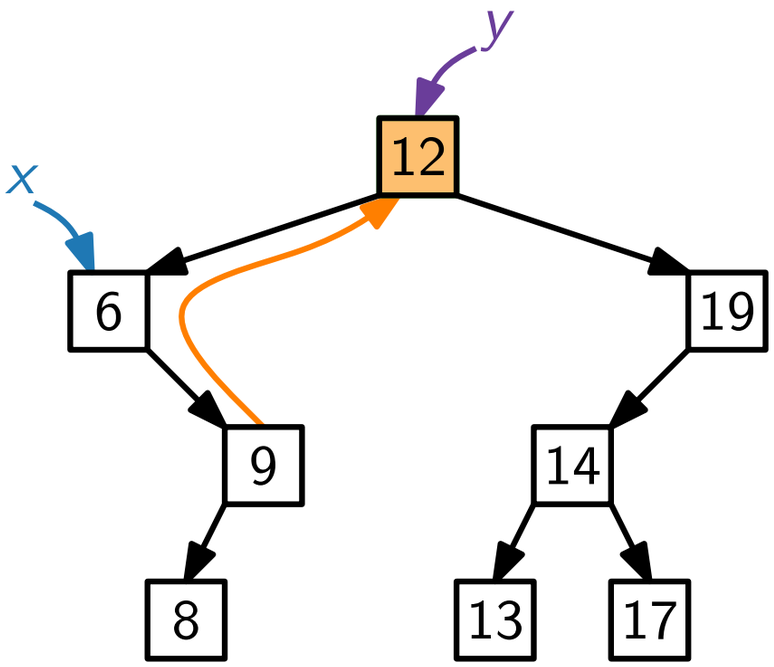

```
NODE SUCCESSOR(NODE x)
  if x.right ≠ nil then
    return MINIMUM(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



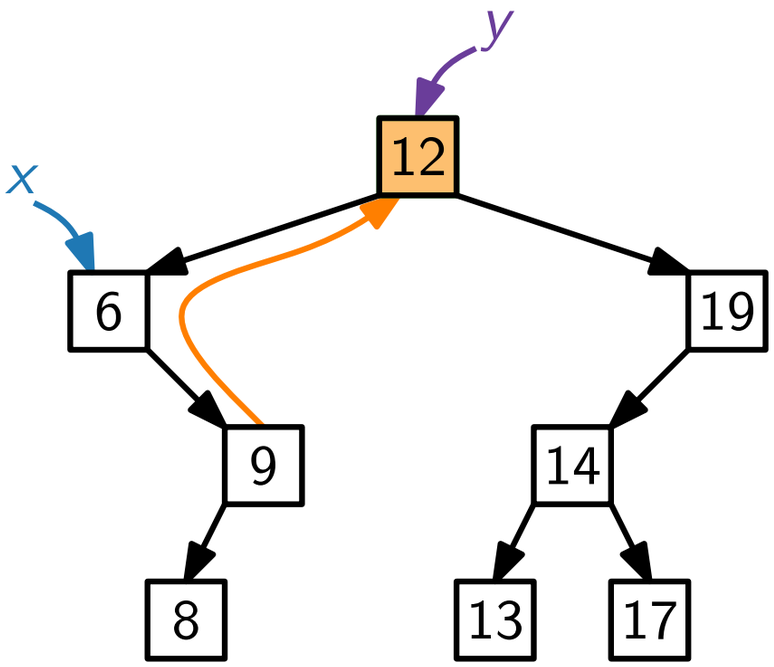

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
  return y
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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



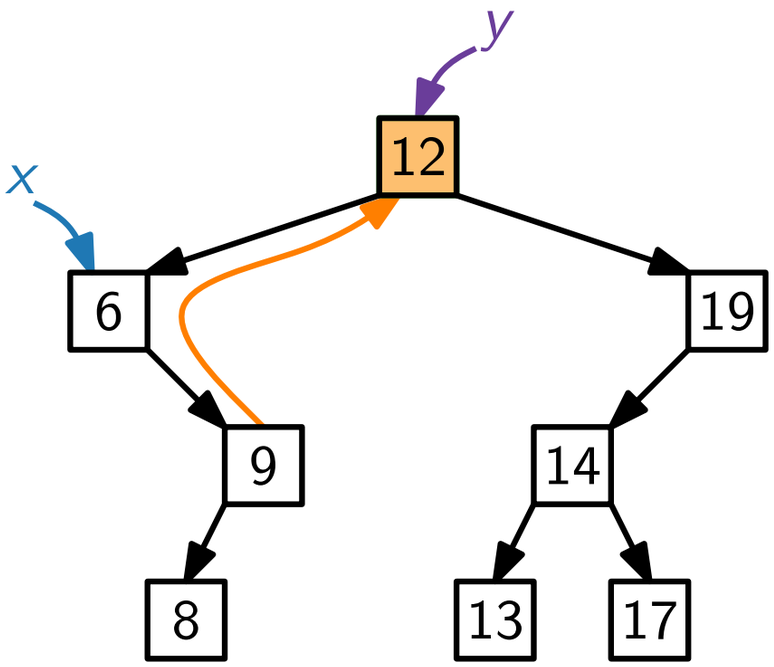

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
  return y
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

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

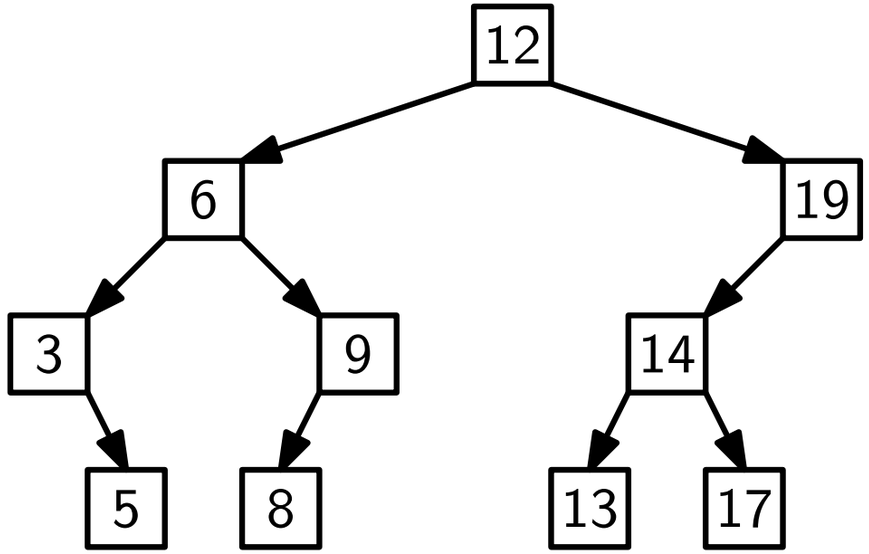


Tipp: Probieren Sie auch z.B. $\textsc{Successor}(19)$!

```
Node Successor(Node x)
  if x.right ≠ nil then
    return Minimum(x.right)
  y = x.p
  while y ≠ nil and x == y.right do
    x = y
    y = y.p
  return y
```

Einfügen

Node Insert(key k)



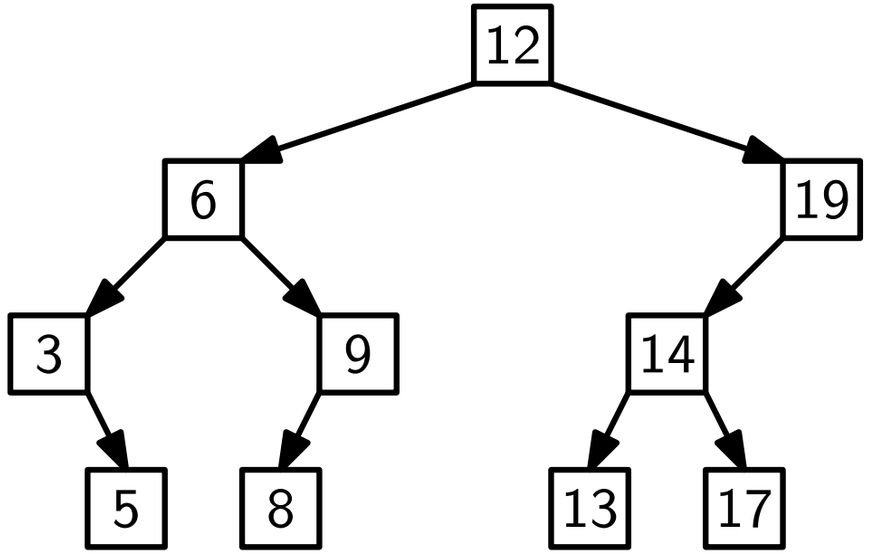
Insert(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$



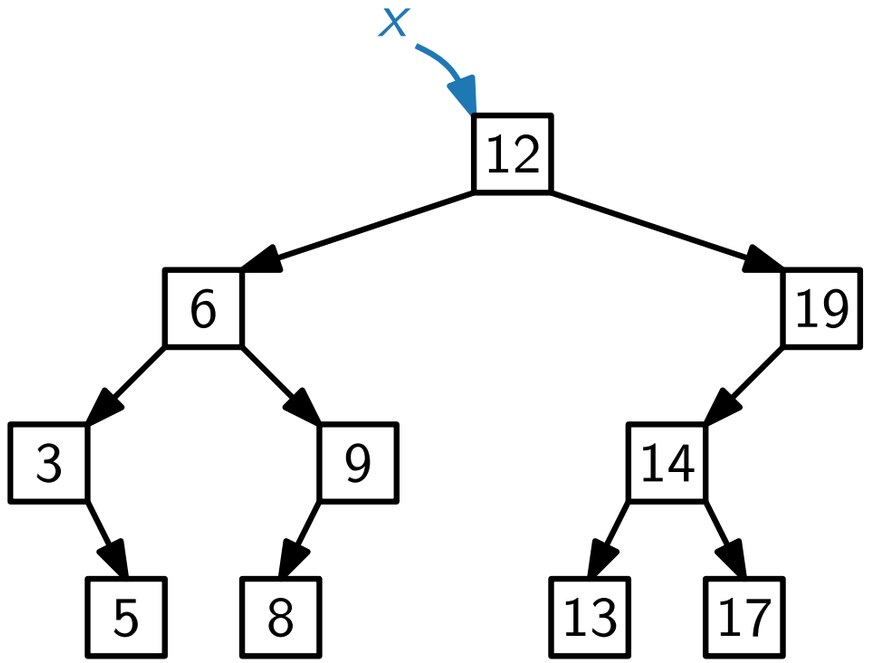
INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$



INSERT(11)

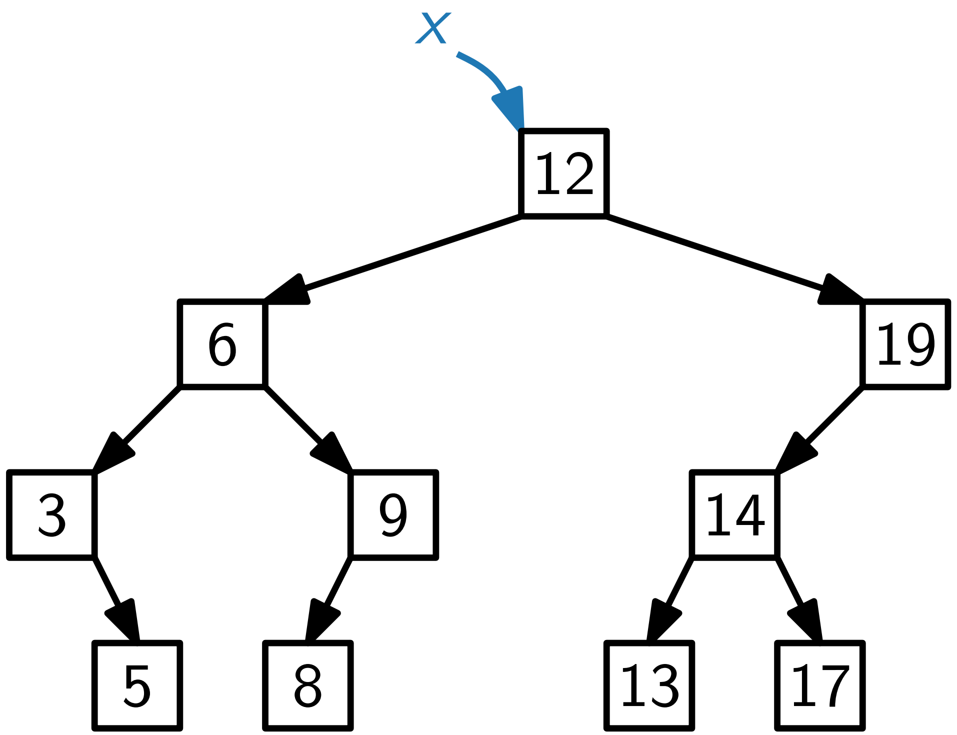
Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

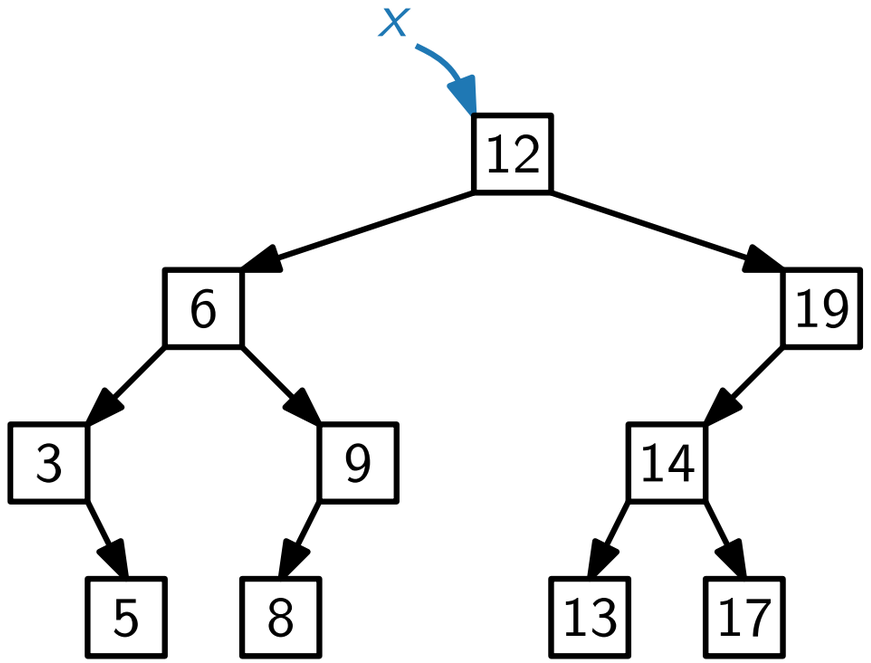
while $x \neq nil$ **do**



INSERT(11)

Einfügen

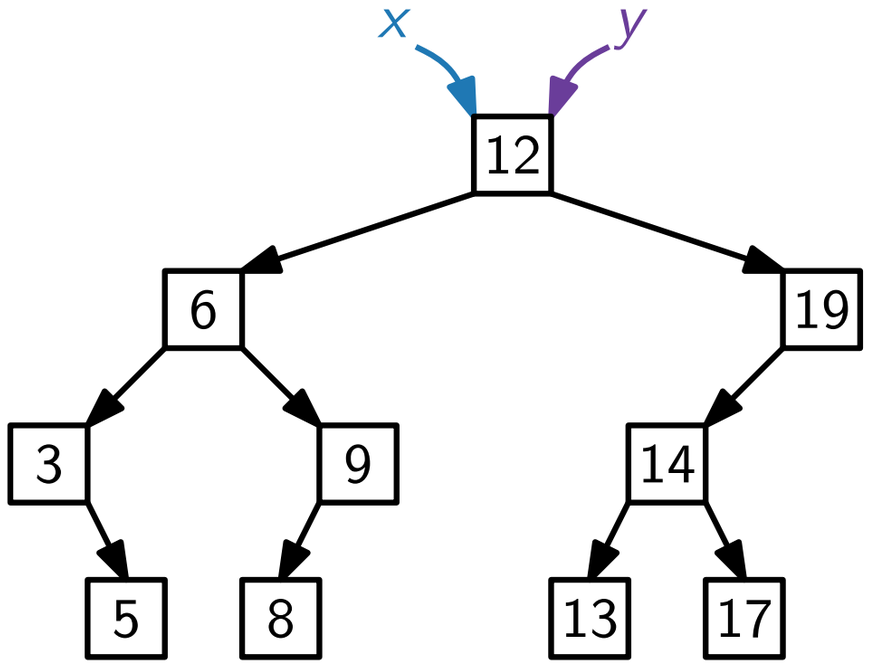
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
```



INSERT(11)

Einfügen

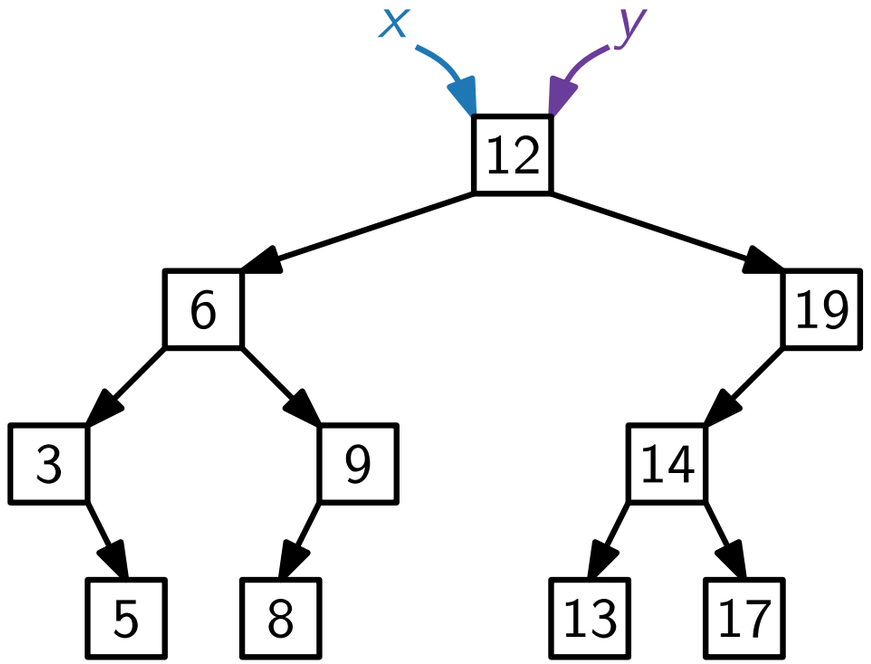
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
```



INSERT(11)

Einfügen

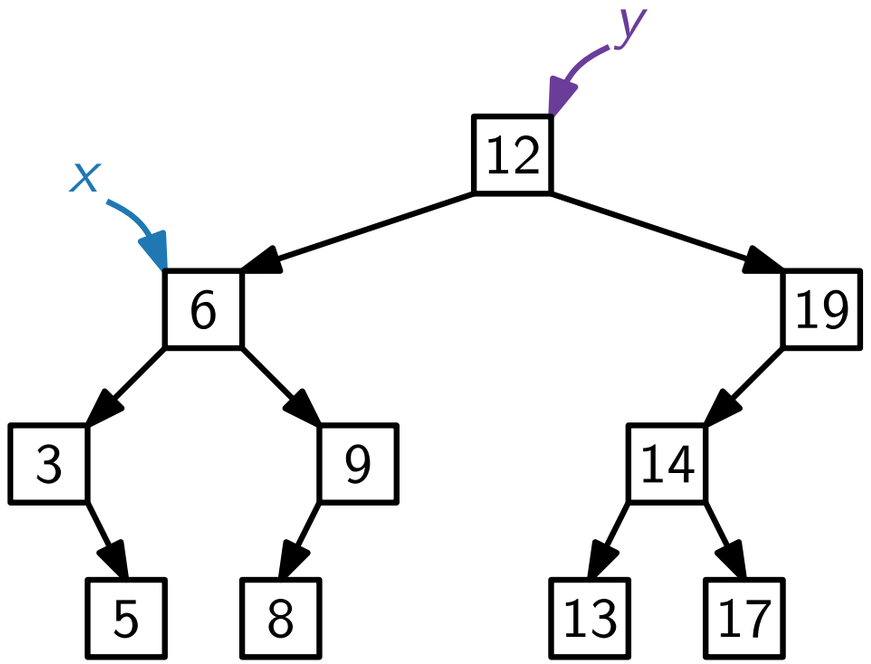
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
```



INSERT(11)

Einfügen

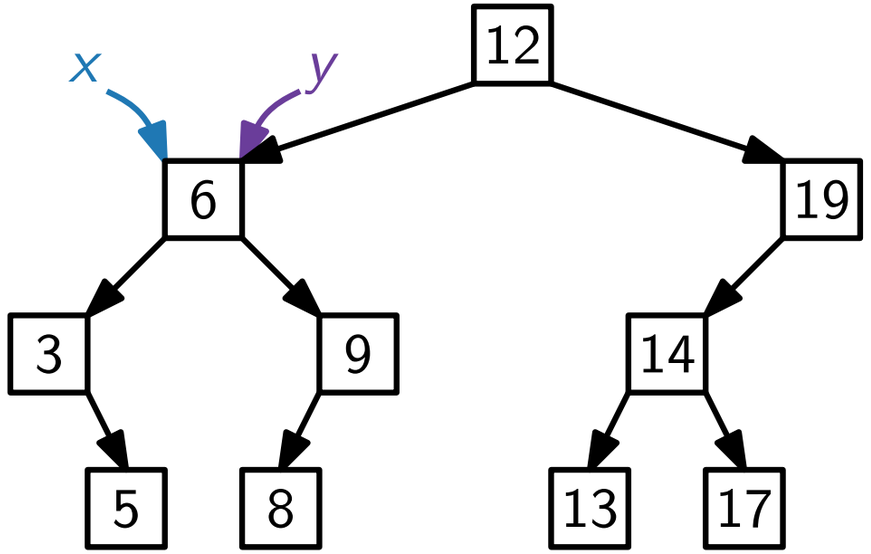
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
```



INSERT(11)

Einfügen

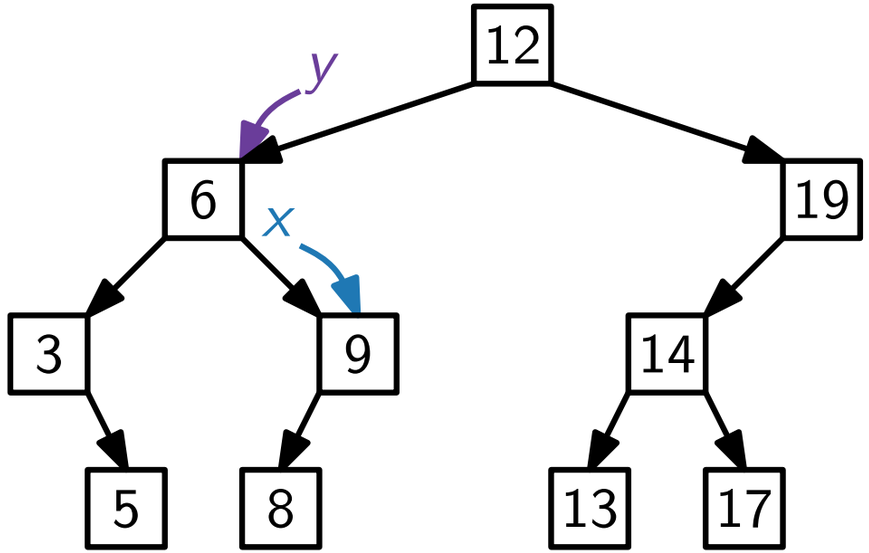
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
```



INSERT(11)

Einfügen

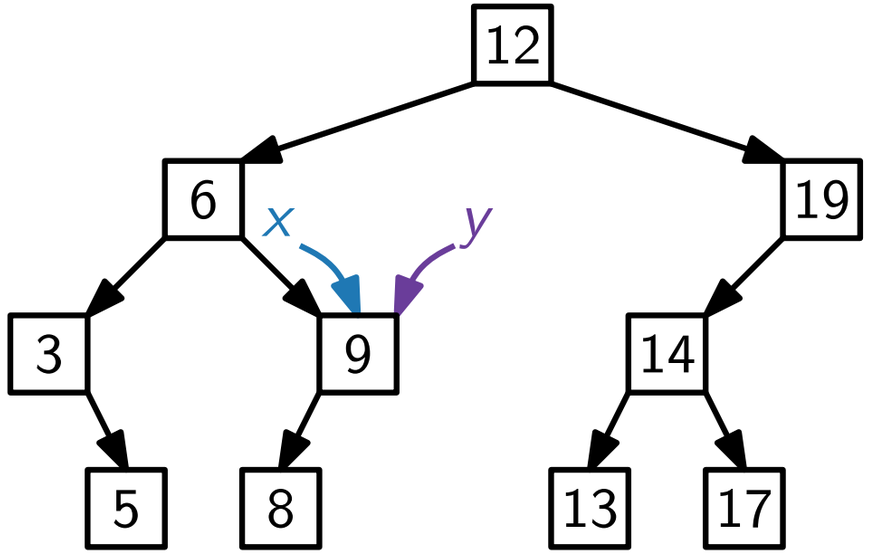
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
```



INSERT(11)

Einfügen

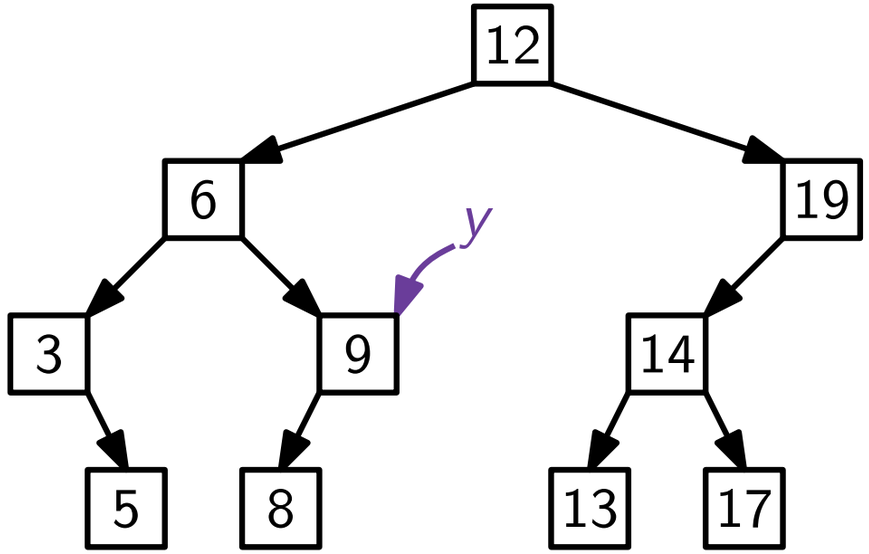
```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
```



INSERT(11)

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
```

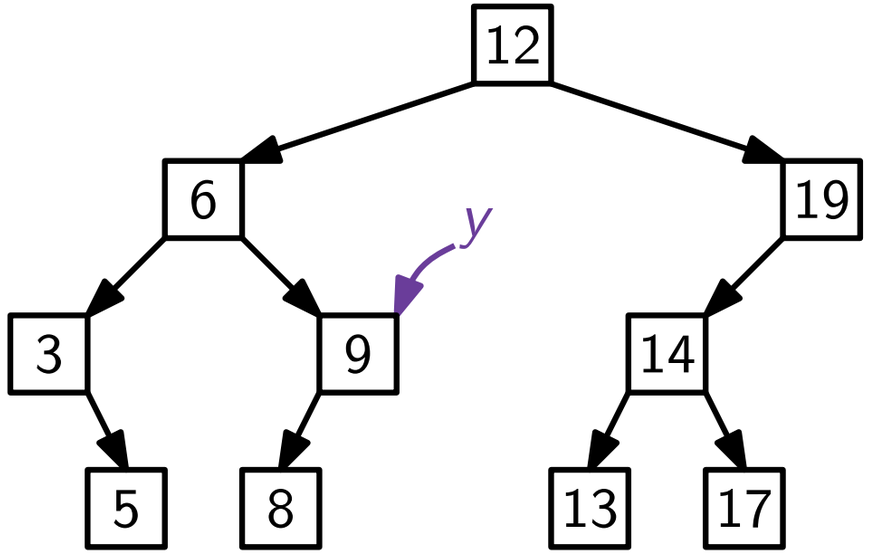


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
```

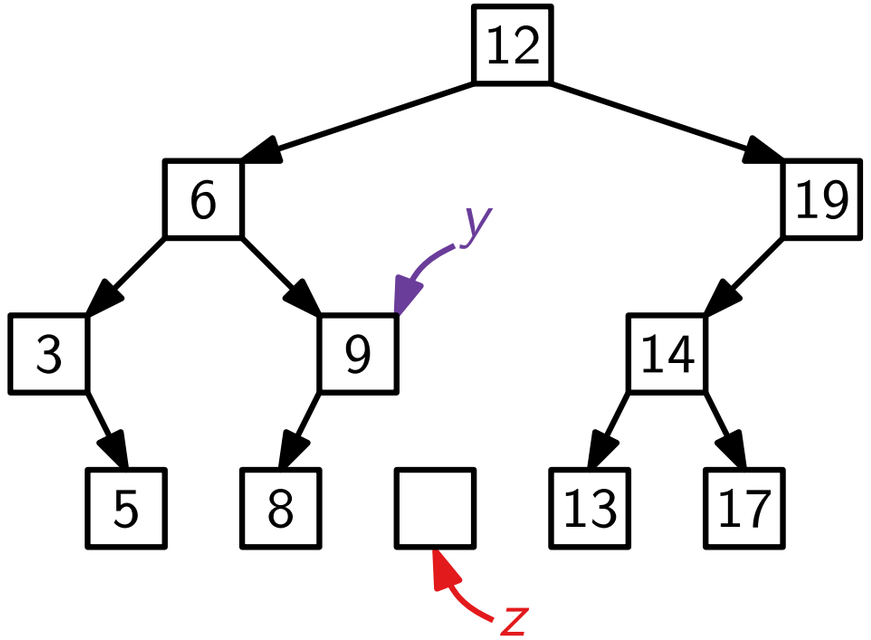


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
```

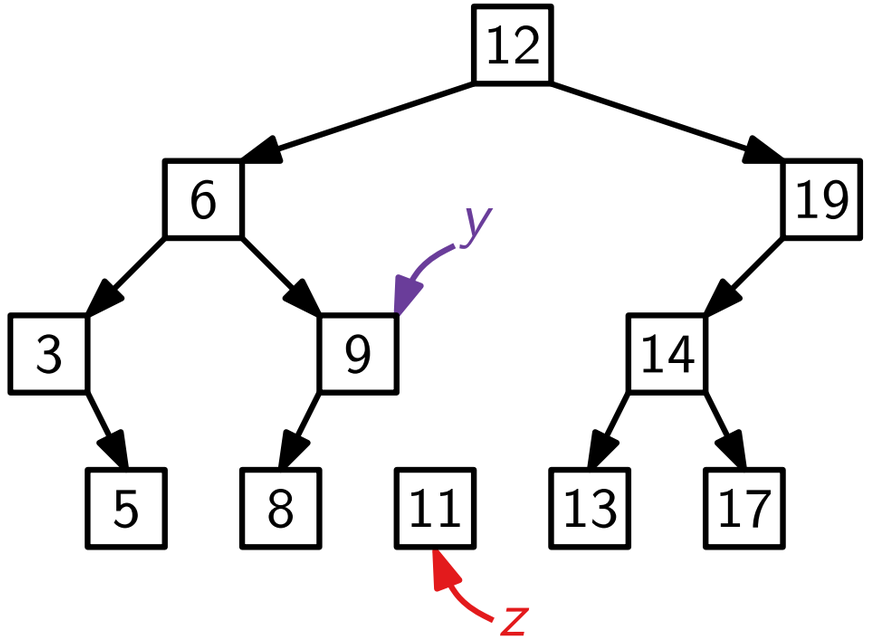


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
```

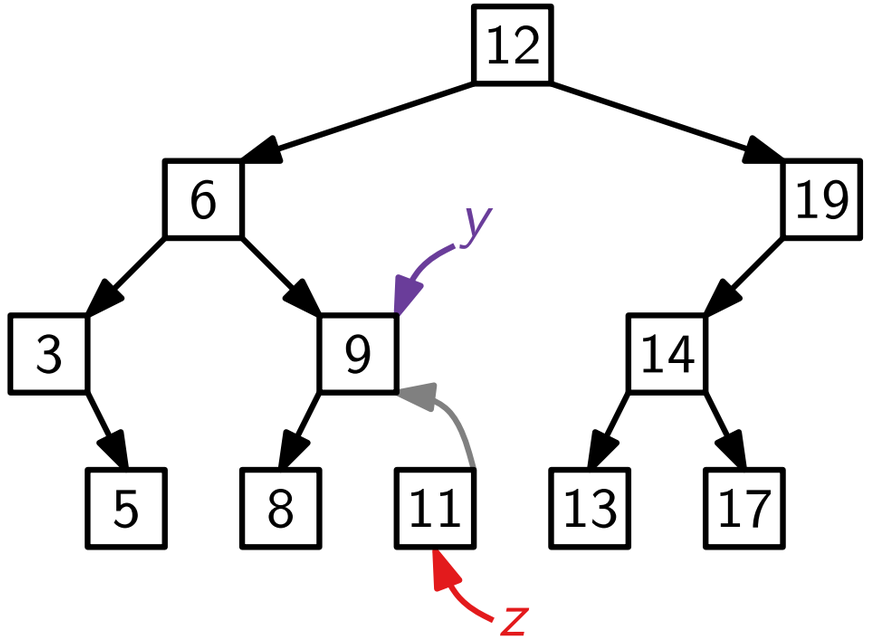


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
```

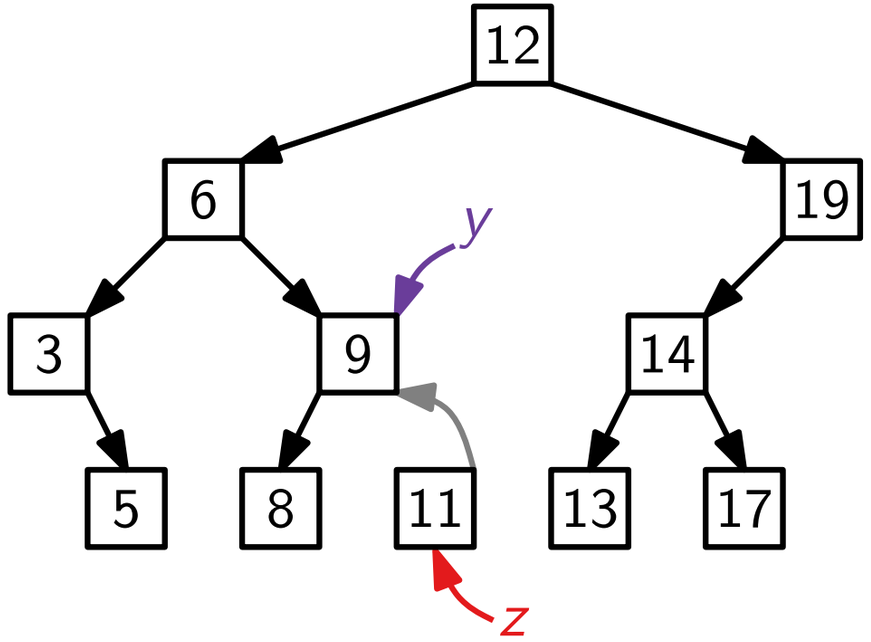


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
    if y == nil then root = z
```

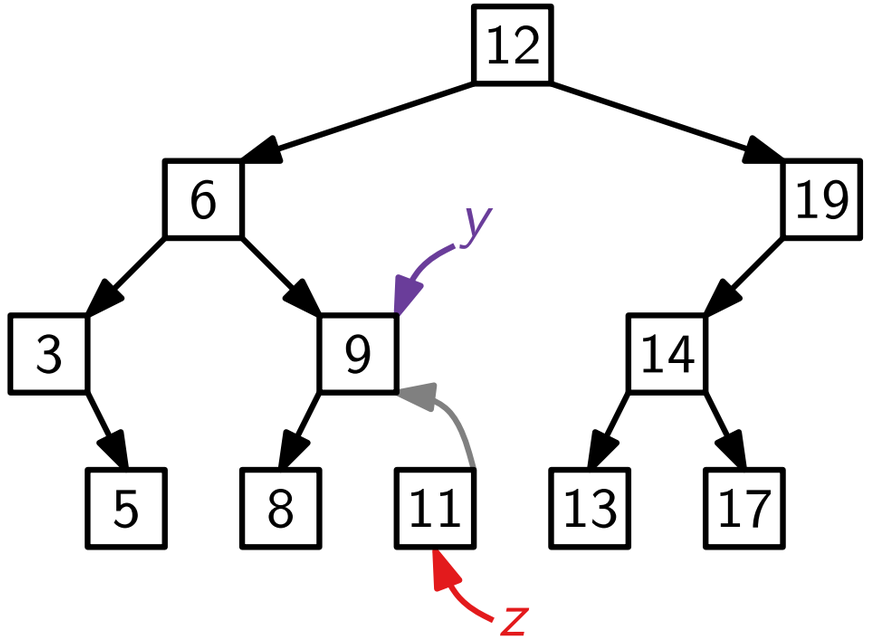


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
    if y == nil then root = z
    else
```

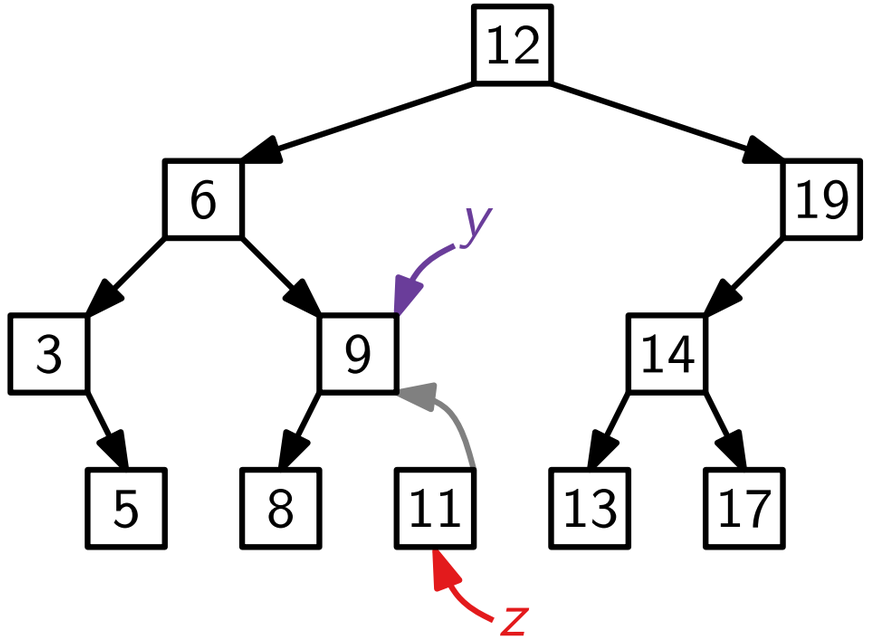


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
    if y == nil then root = z
    else
        if k < y.key then y.left = z
        else              y.right = z
```

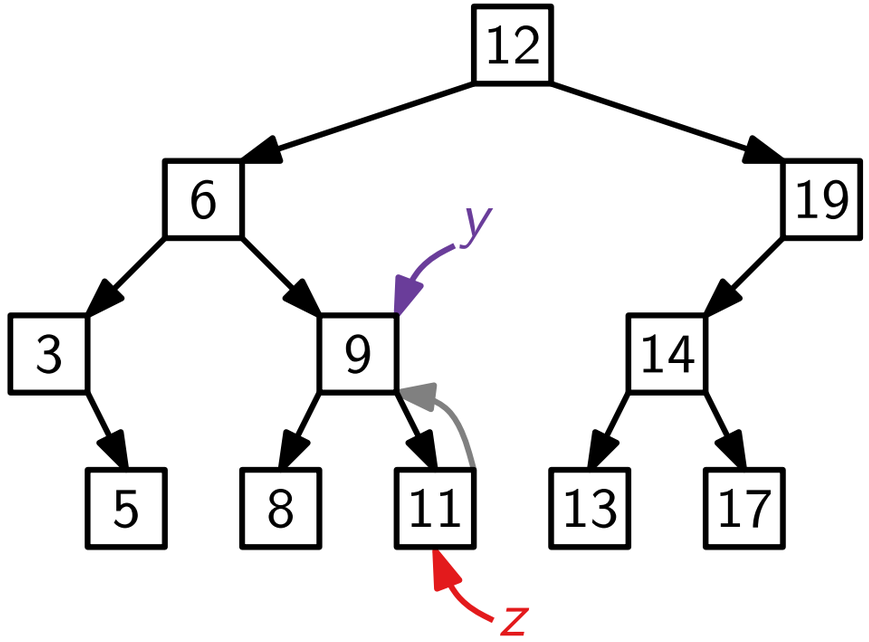


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = new Node(k, y)
    if y == nil then root = z
    else
        if k < y.key then y.left = z
        else              y.right = z
```

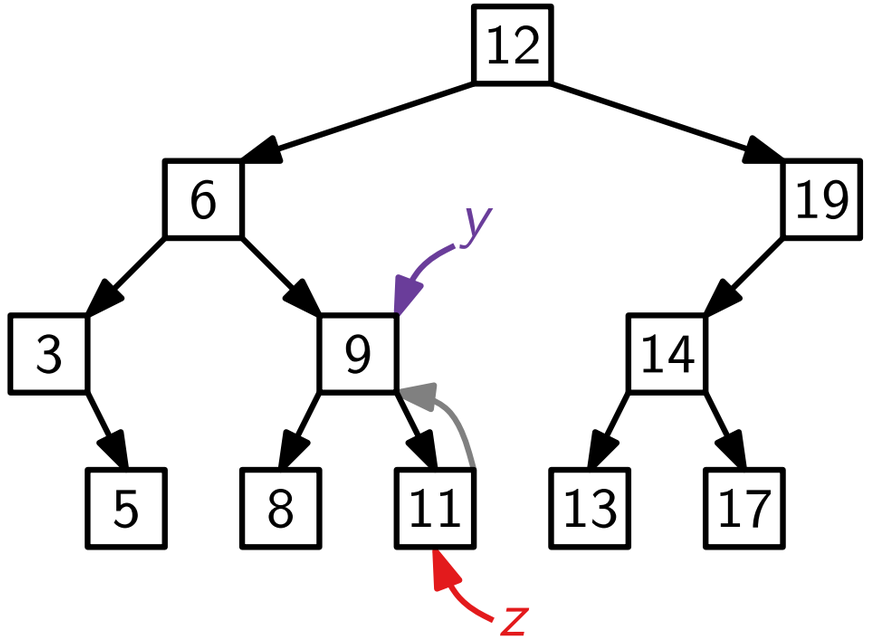


INSERT(11)

$x == nil$

Einfügen

```
Node INSERT(key k)
    y = nil
    x = root
    while x ≠ nil do
        y = x
        if k < x.key then
            x = x.left
        else x = x.right
    z = 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
```

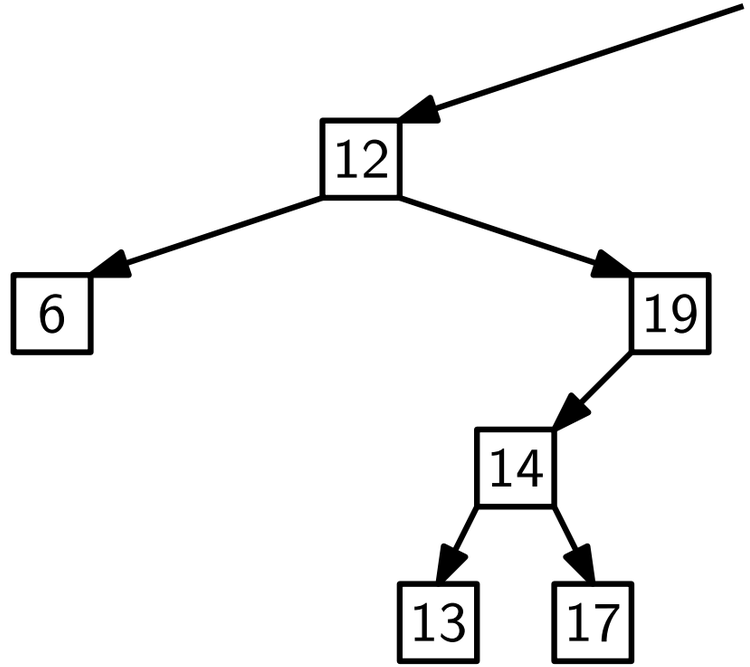


INSERT(11)

$x == nil$

Löschen

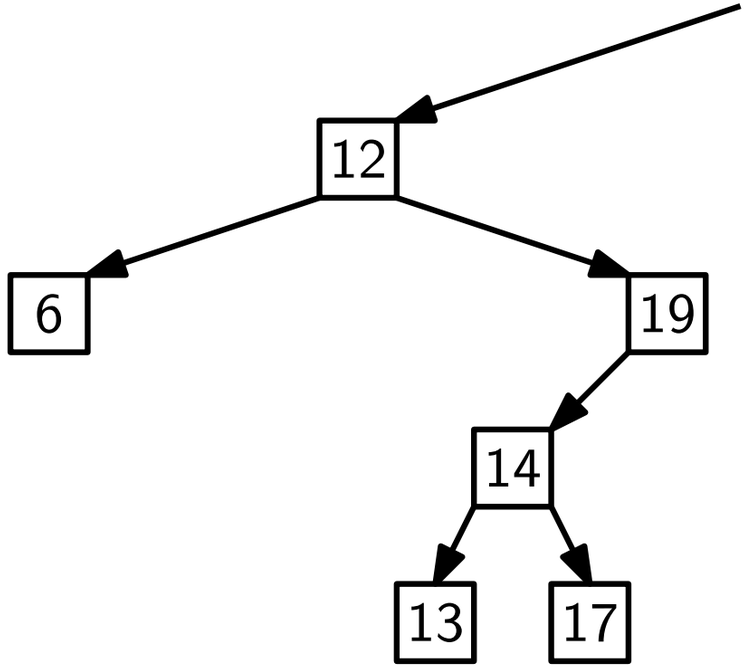
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:



Löschen

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

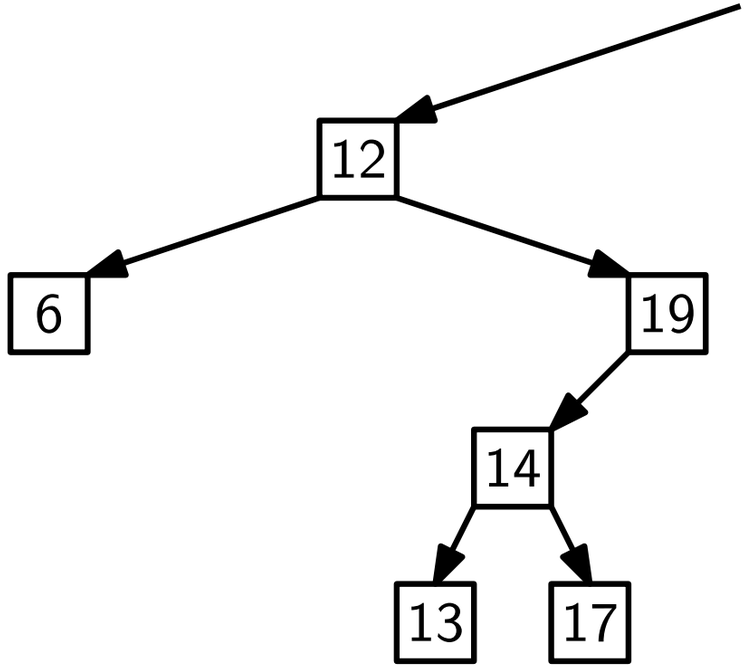
1. z hat keine Kinder.



Löschen

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

1. z hat keine Kinder.

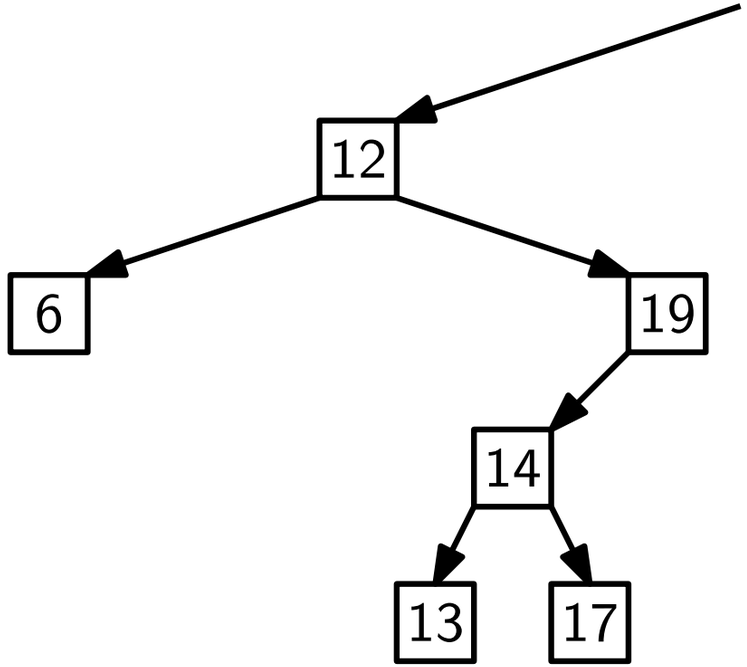


2. z hat ein Kind x.

Löschen

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

1. z hat keine Kinder.



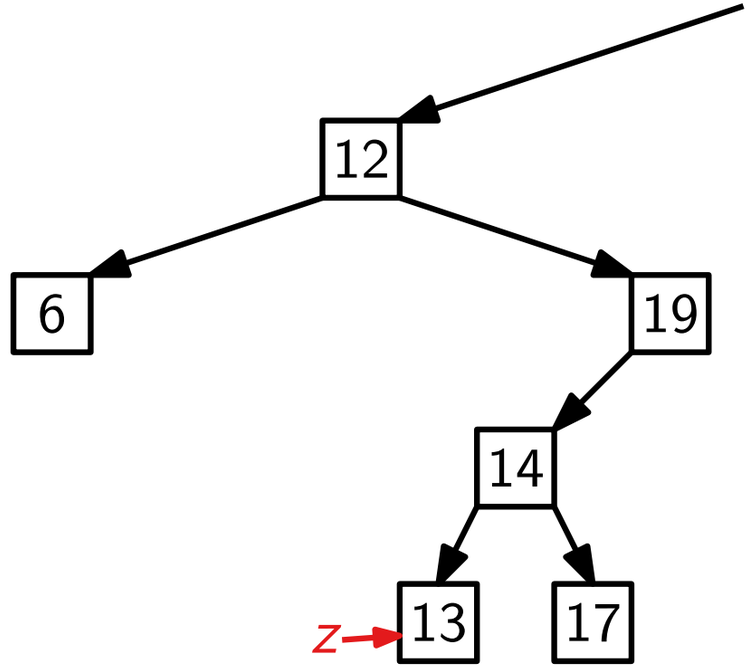
2. z hat ein Kind x.

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.

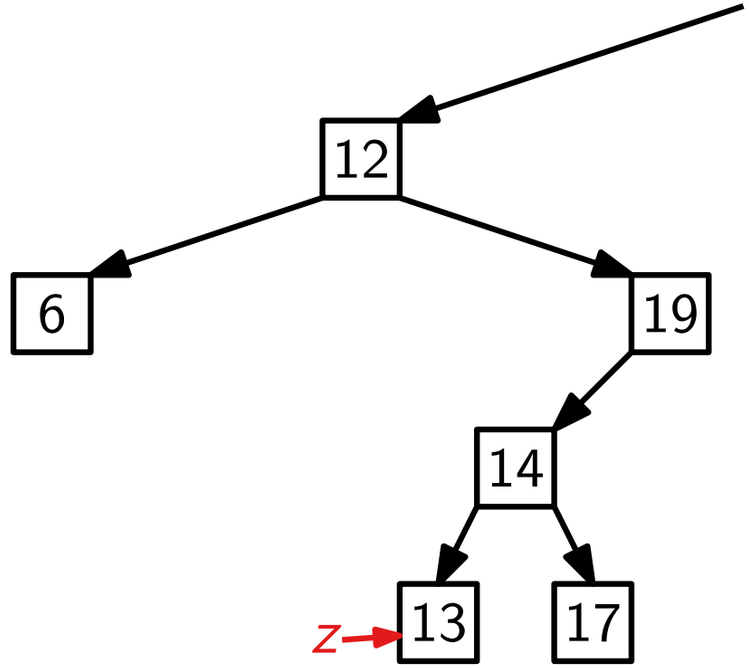
3. z hat zwei Kinder.

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.

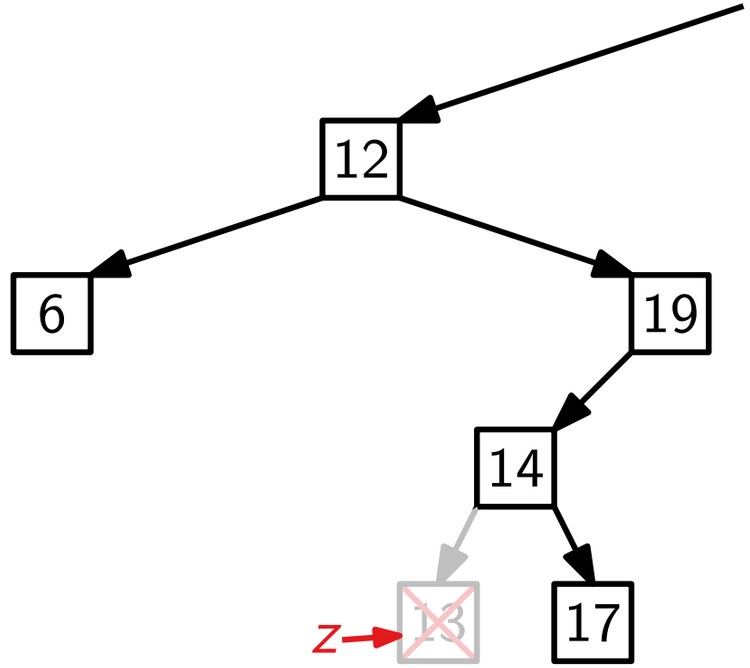
3. z hat zwei Kinder.

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.

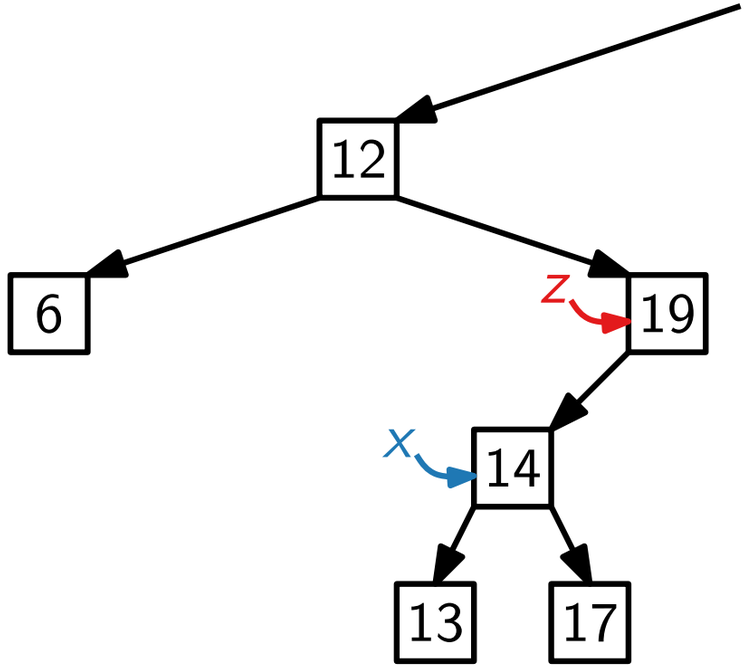
3. z hat zwei Kinder.

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:

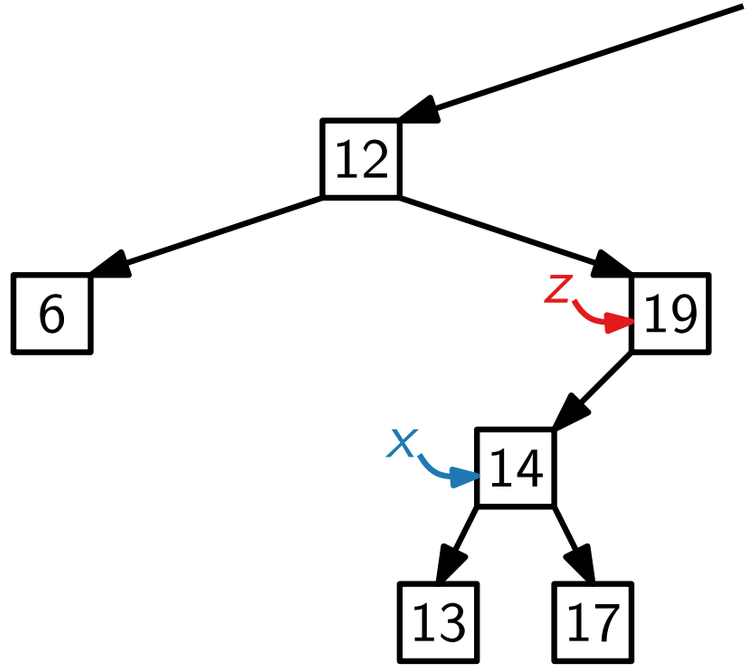
1. z hat keine Kinder.

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:

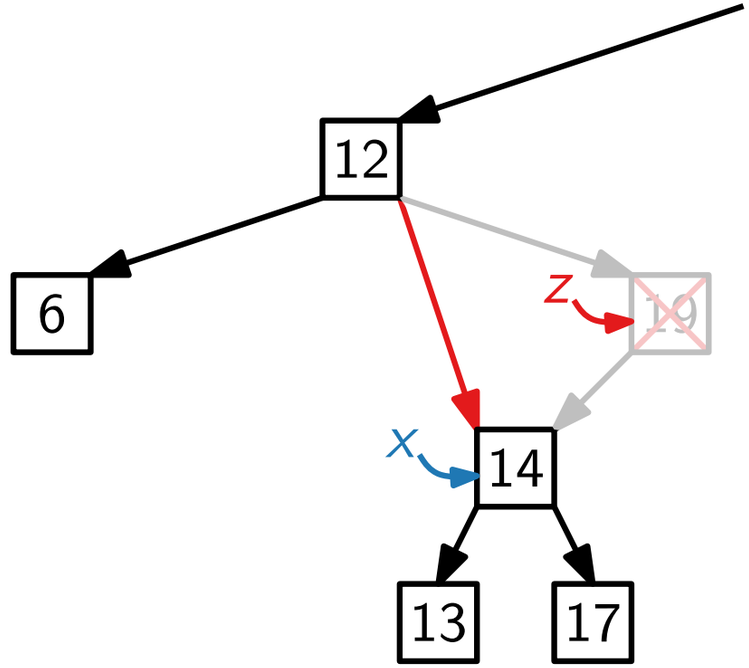
1. z hat keine Kinder.

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.

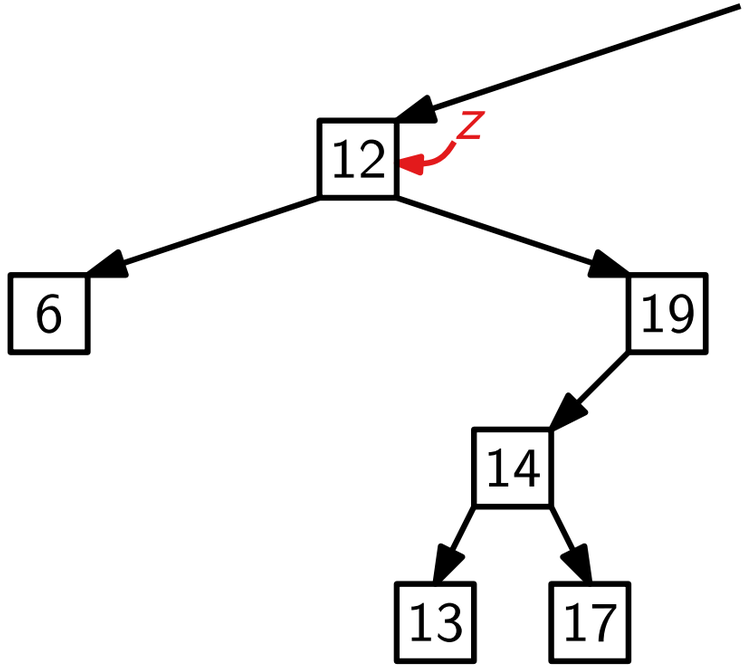


Löschen

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.

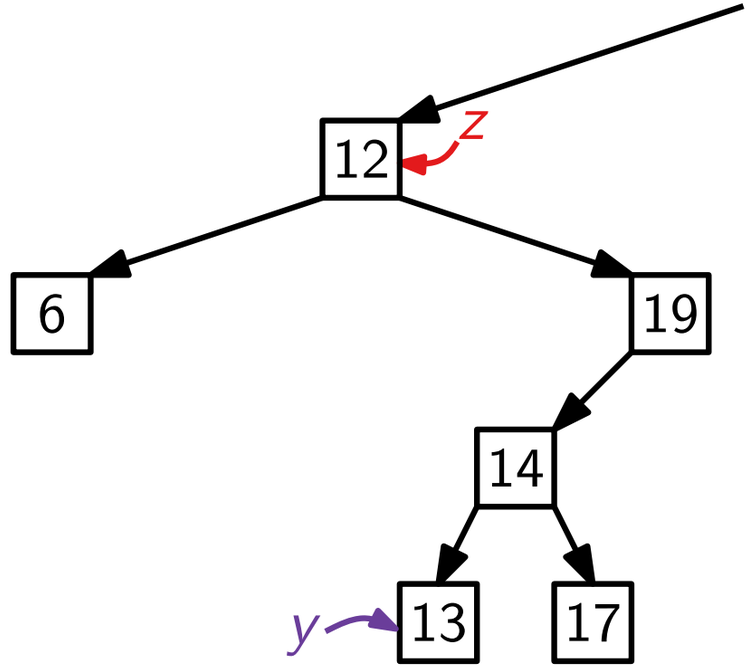
3. z hat zwei Kinder.

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.

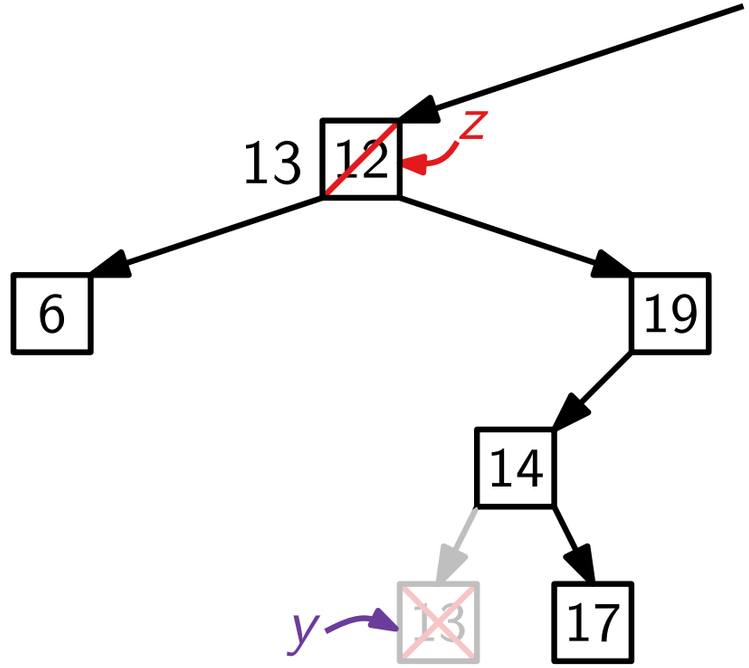
Setze $y = \textsc{Successor}(z)$ und $z.key = y.key$. Lösche y. (Fall 1 oder 2!)

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.

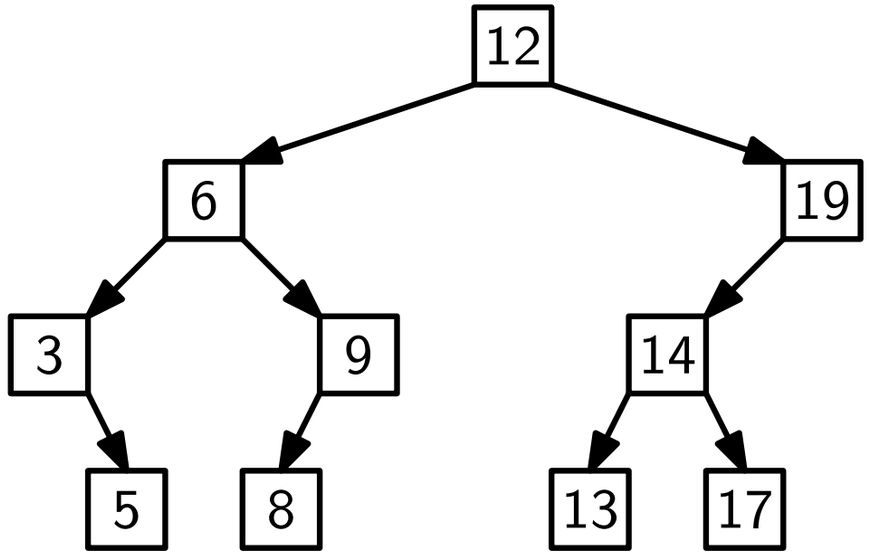
Setze den Zeiger von $z.p$, der auf z zeigt, auf x.
Setze $x.p = z.p$. Lösche z.

3. z hat zwei Kinder.

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

Zusammenfassung

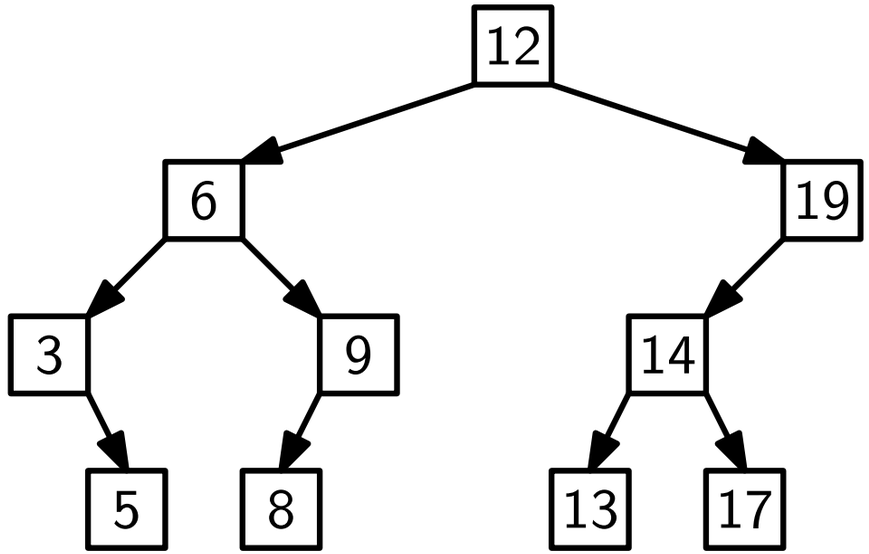
Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.



Zusammenfassung

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

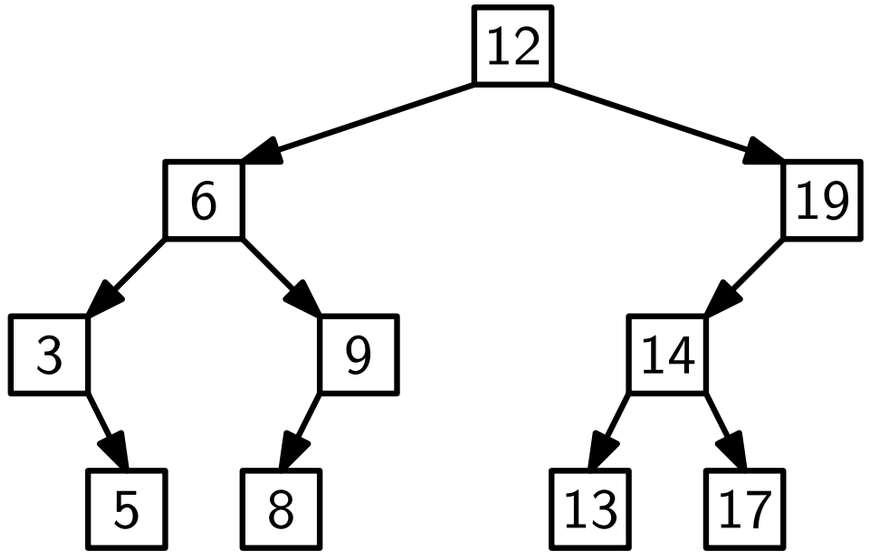
Aber: Im schlechtesten Fall gilt $h \in$



Zusammenfassung

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

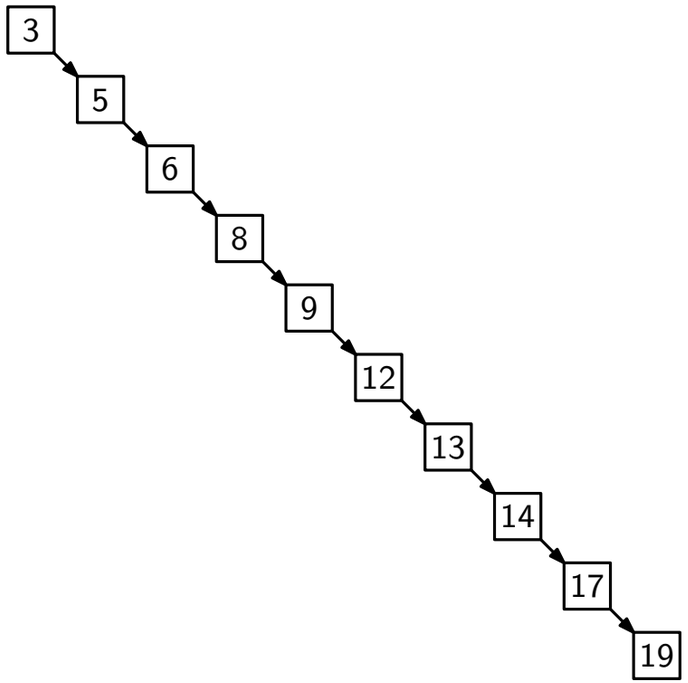
Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

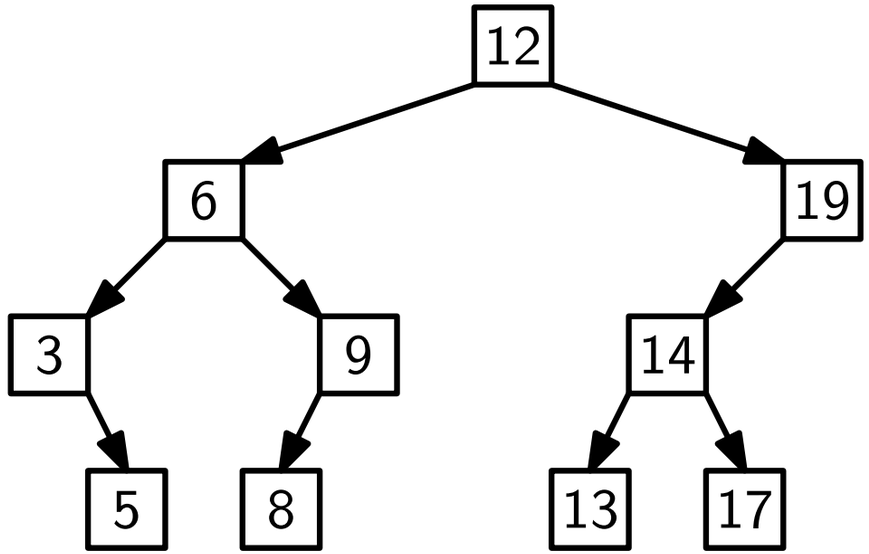


Zusammenfassung

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 $h \in \Theta(n)$.



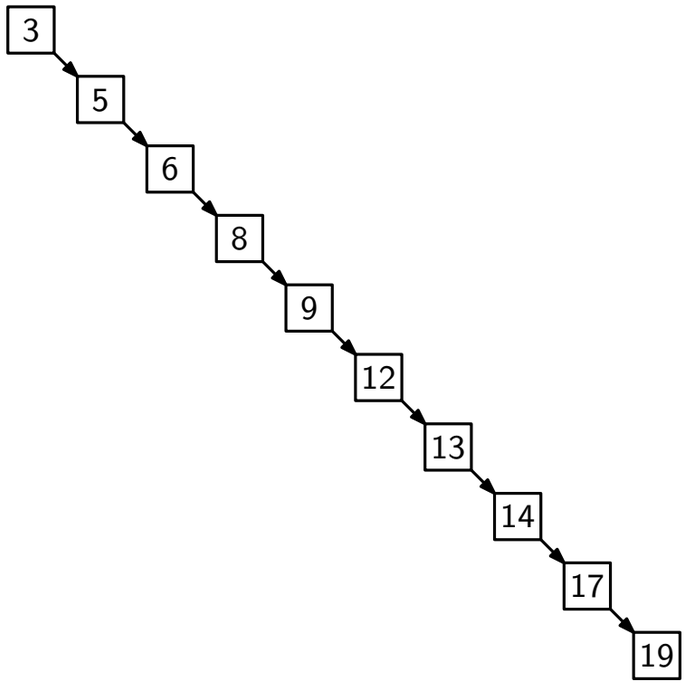


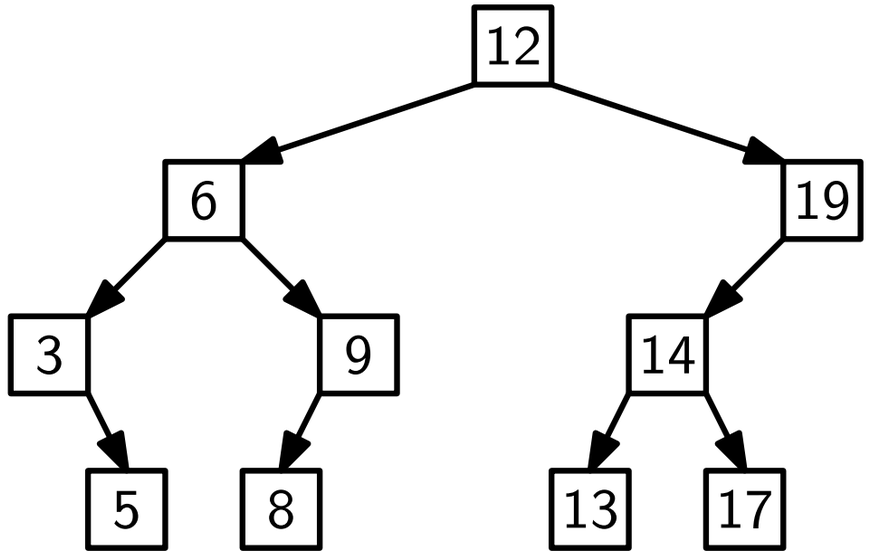
Zusammenfassung

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:



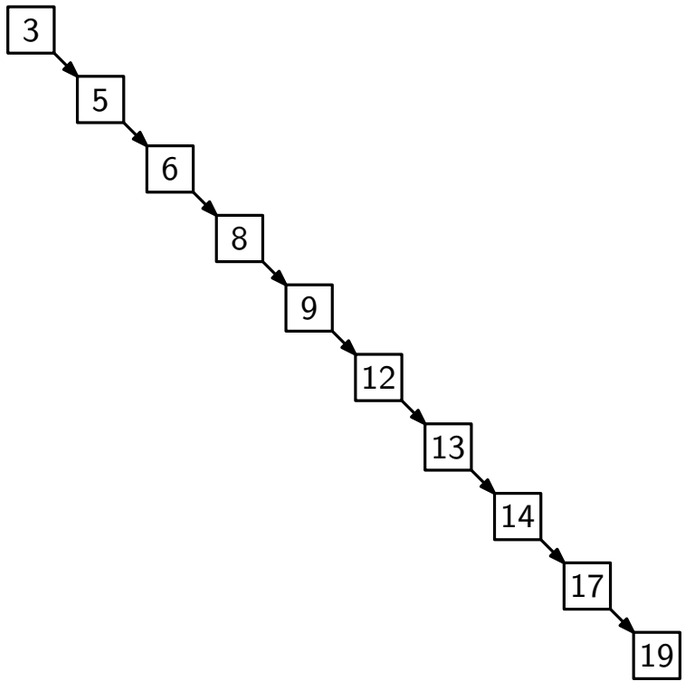


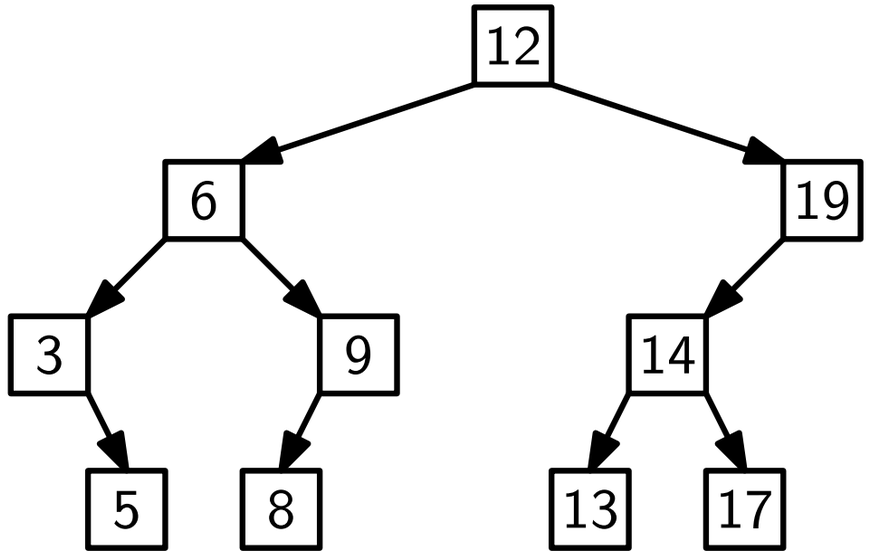
Zusammenfassung

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** $\Rightarrow$ $h \in \mathcal{O}(\log n)$