



Julius-Maximilians-

UNIVERSITÄT
WÜRZBURG

Lehrstuhl für

INFORMATIK I

Algorithmen & Komplexität



Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2020/21

13. Vorlesung

Binäre Suchbäume

Dynamische Menge

verwaltet Elemente einer
sich ändernden Menge M



Abstrakter Datentyp	Funktionalität	
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code>	$\left. \begin{array}{l} \text{ptr Insert(key } k, \text{ info } i) \\ \text{Delete(ptr } x) \\ \text{ptr Search(key } k) \end{array} \right\} \text{Änderungen}$	$\left. \begin{array}{l} \text{ptr Insert(key } k, \text{ info } i) \\ \text{Delete(ptr } x) \\ \text{ptr Search(key } k) \end{array} \right\} \text{Änderungen}$
<code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	$\left. \begin{array}{l} \text{ptr Minimum()} \\ \text{ptr Maximum()} \\ \text{ptr Predecessor(ptr } x) \\ \text{ptr Successor(ptr } x) \end{array} \right\} \text{Anfragen}$	$\left. \begin{array}{l} \text{ptr Minimum()} \\ \text{ptr Maximum()} \\ \text{ptr Predecessor(ptr } x) \\ \text{ptr Successor(ptr } x) \end{array} \right\} \text{Anfragen}$

Implementierung: je nachdem...

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	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)$	$\Theta(n)$

Implementierung

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^{\oplus}$	$\Theta(n)$

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^{\oplus}$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

^{*}) unter bestimmten Annahmen.

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^{\oplus}$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—

[⊕]) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

^{*}) unter bestimmten Annahmen.

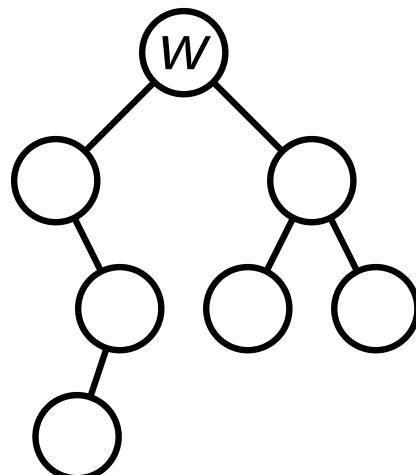
	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^{\oplus}$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

[⊕]) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

\star) unter bestimmten Annahmen.

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^\oplus$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^\star$	$\Theta(1)^\star$	—	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$



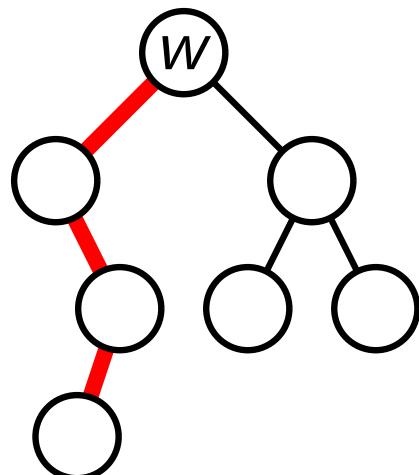
$h(T)$ = Höhe des Baums T

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

\star) unter bestimmten Annahmen.

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^\oplus$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^\star$	$\Theta(1)^\star$	—	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$



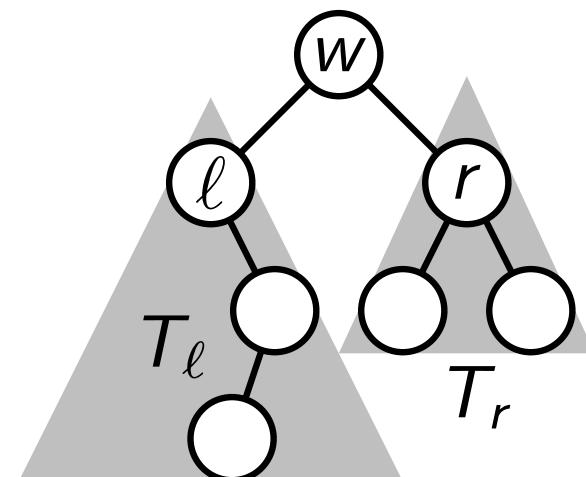
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

\star) unter bestimmten Annahmen.

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^\oplus$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^\star$	$\Theta(1)^\star$	—	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$



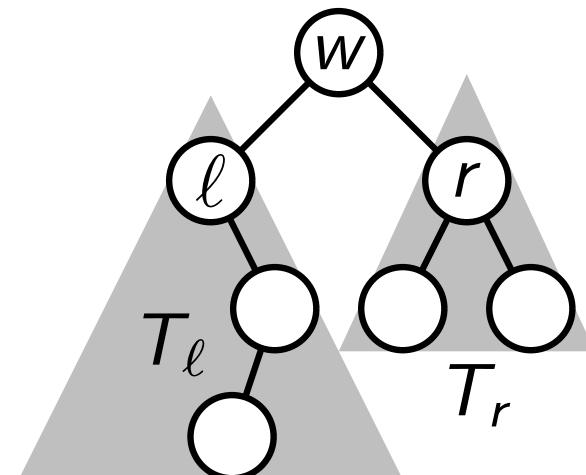
$h(T)$ = Höhe des Baums T
= Anz. Kanten auf längstem Wurzel-Blatt-Pfad
= $\left\{ \begin{array}{ll} \text{Finden Sie die} \\ \text{Rekursionsgleichung!} \end{array} \right.$ falls Baum = Blatt
sonst.

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Implementierung

\star) unter bestimmten Annahmen.

	Search	Ins/Del	Min/Max	Pred/Succ
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
unsortiertes Feld	$\Theta(n)$	$\Theta(1)/\Theta(n)$	$\Theta(1)^\oplus$	$\Theta(n)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$



$$\begin{aligned}
 h(T) &= \text{Höhe des Baums } T \\
 &= \text{Anz. Kanten auf längstem Wurzel-Blatt-Pfad} \\
 &= \begin{cases} 0 & \text{falls Baum = Blatt} \\ 1 + \max\{h(T_l), h(T_r)\} & \text{sonst.} \end{cases}
 \end{aligned}$$

\oplus) Weil wir nach dem Löschen (in linearer Zeit) einfach das neue Min/Max suchen können.

Suche im sortierten Feld

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

Suche 21!

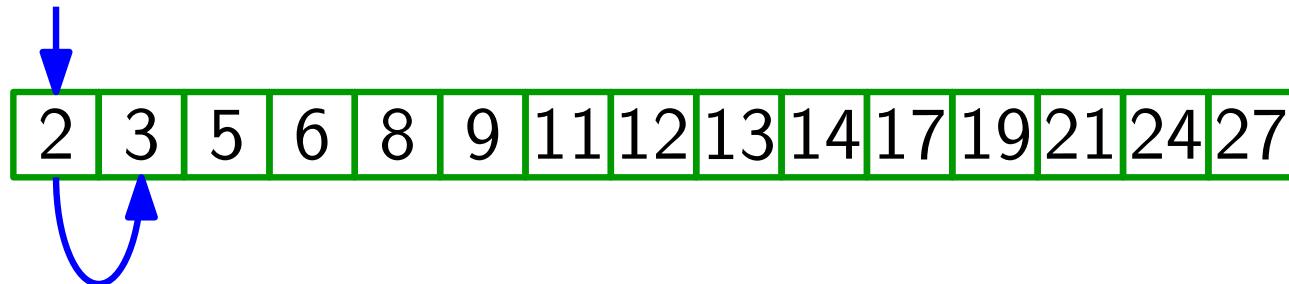
Suche im sortierten Feld



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

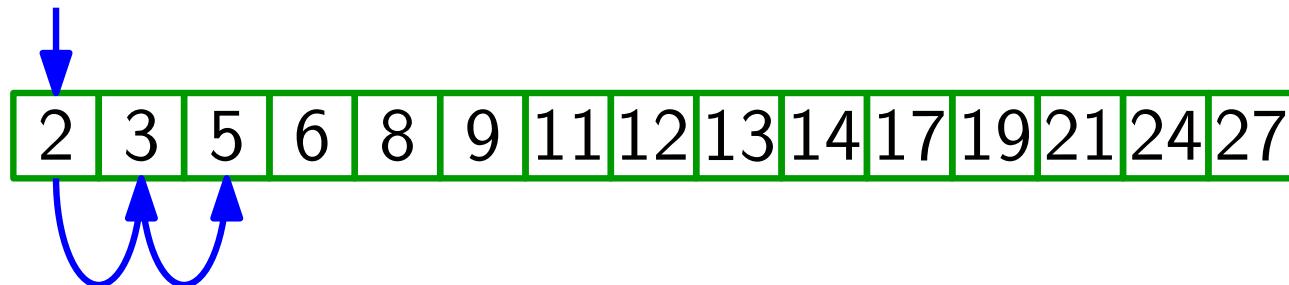
Suche 21!

Suche im sortierten Feld



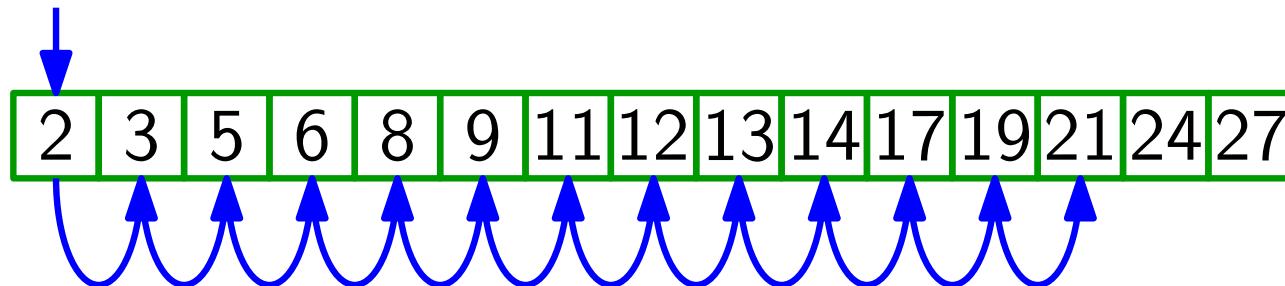
Suche 21!

Suche im sortierten Feld



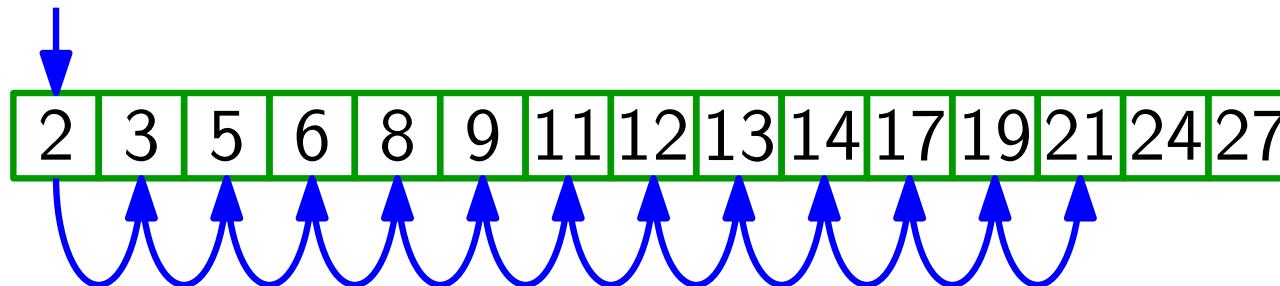
Suche 21!

Suche im sortierten Feld



Suche 21!

Suche im sortierten Feld



Suche 21!

hier im Worst Case
Lineare Suche: 13 n Schritte

Suche im sortierten Feld

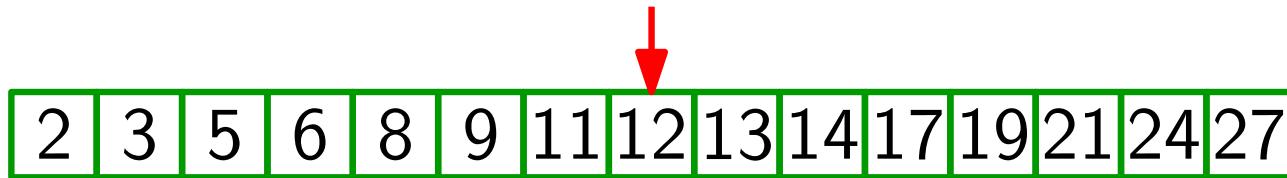
2	3	5	6	8	9	11	12	13	14	17	19	21	24	27
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

Suche 21!

hier im Worst Case

Lineare Suche: 13 n Schritte

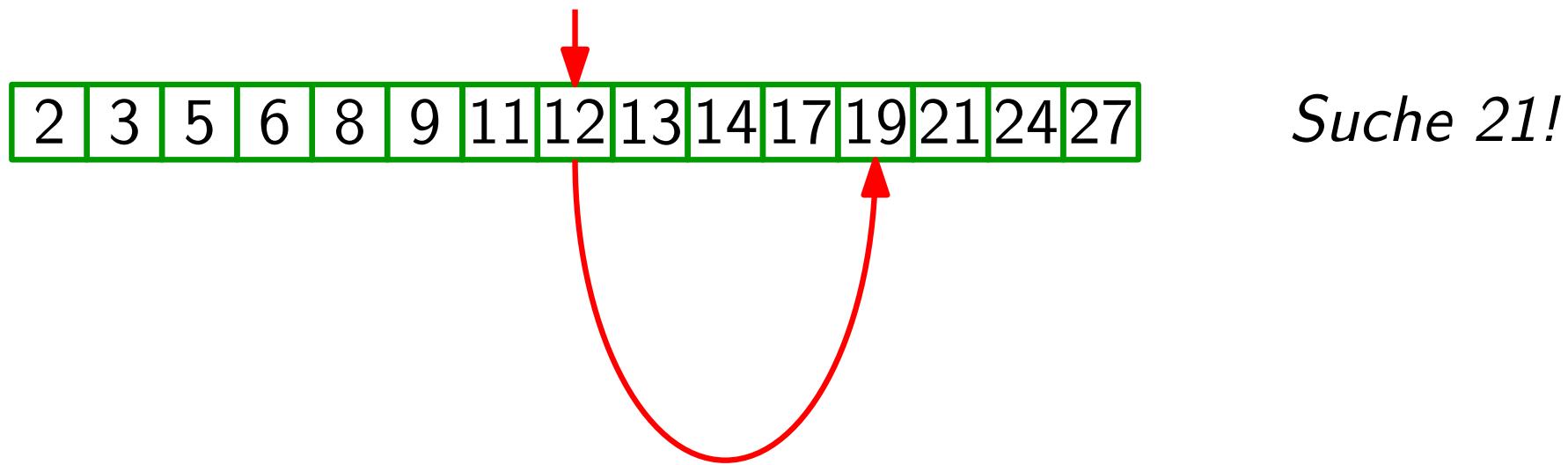
Suche im sortierten Feld



Suche 21!

hier im Worst Case

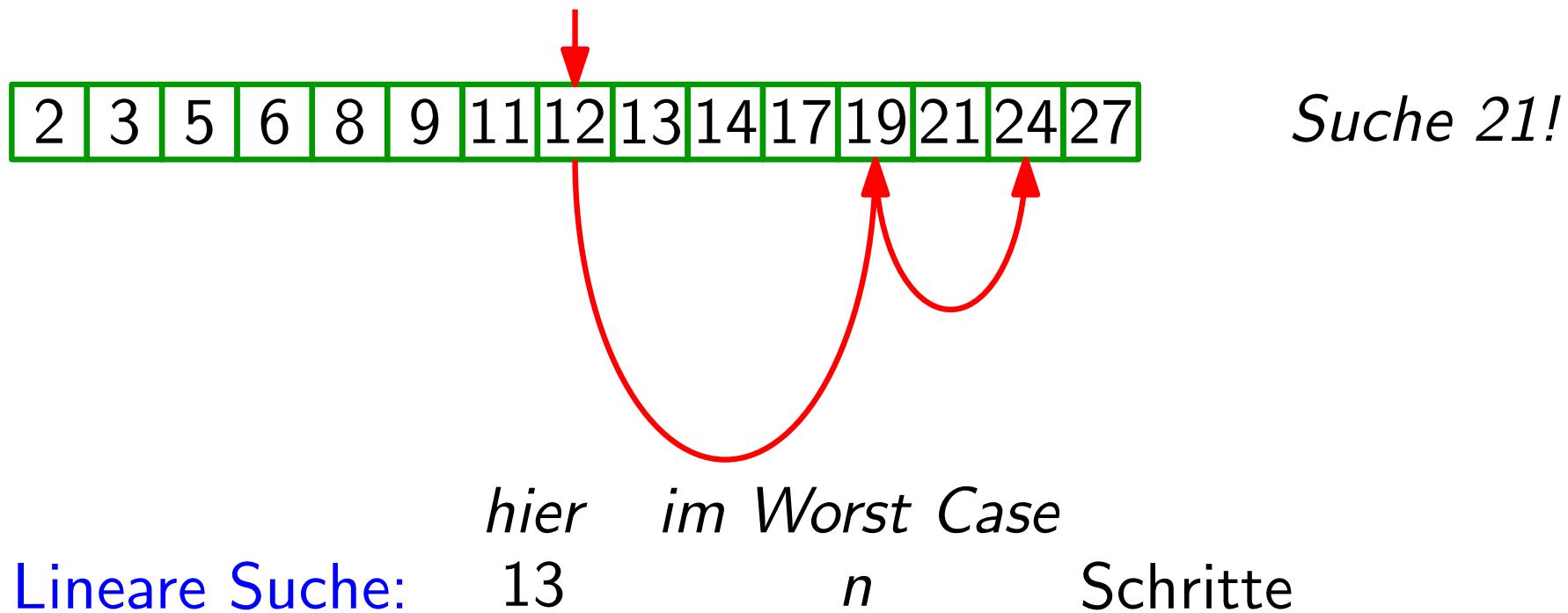
Suche im sortierten Feld



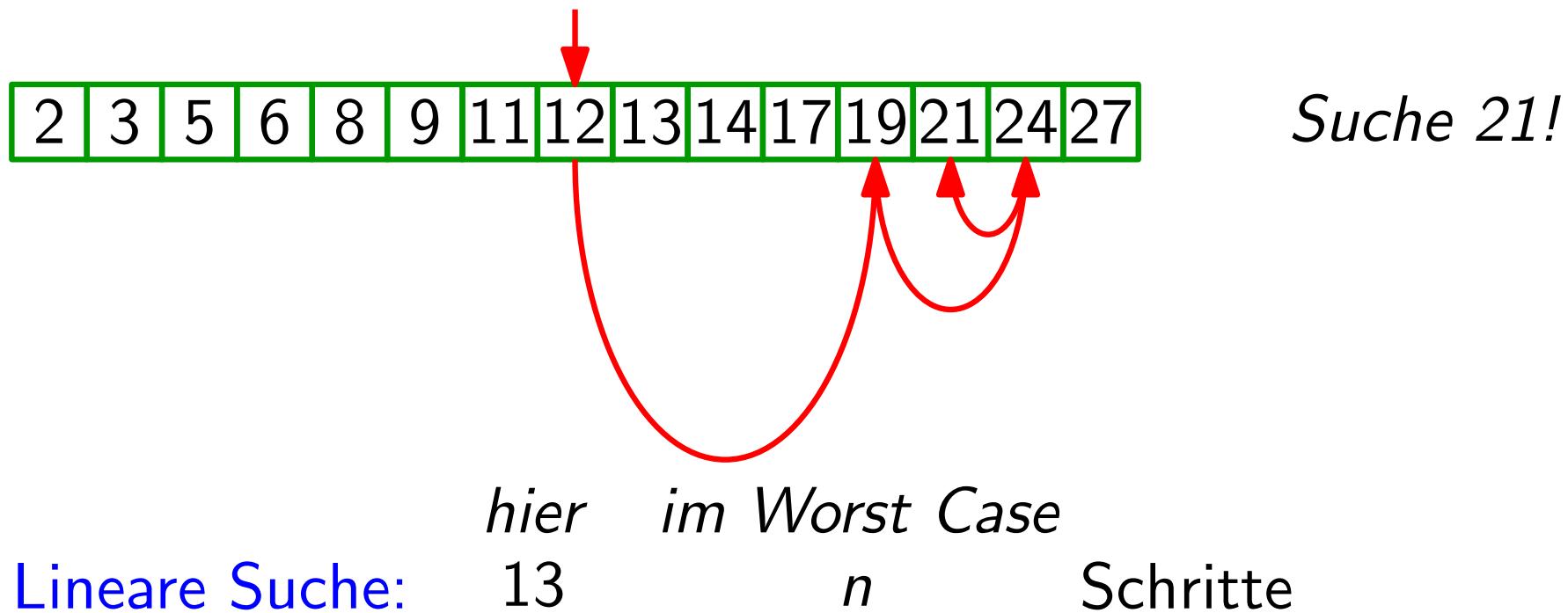
hier im Worst Case

Lineare Suche: 13 n Schritte

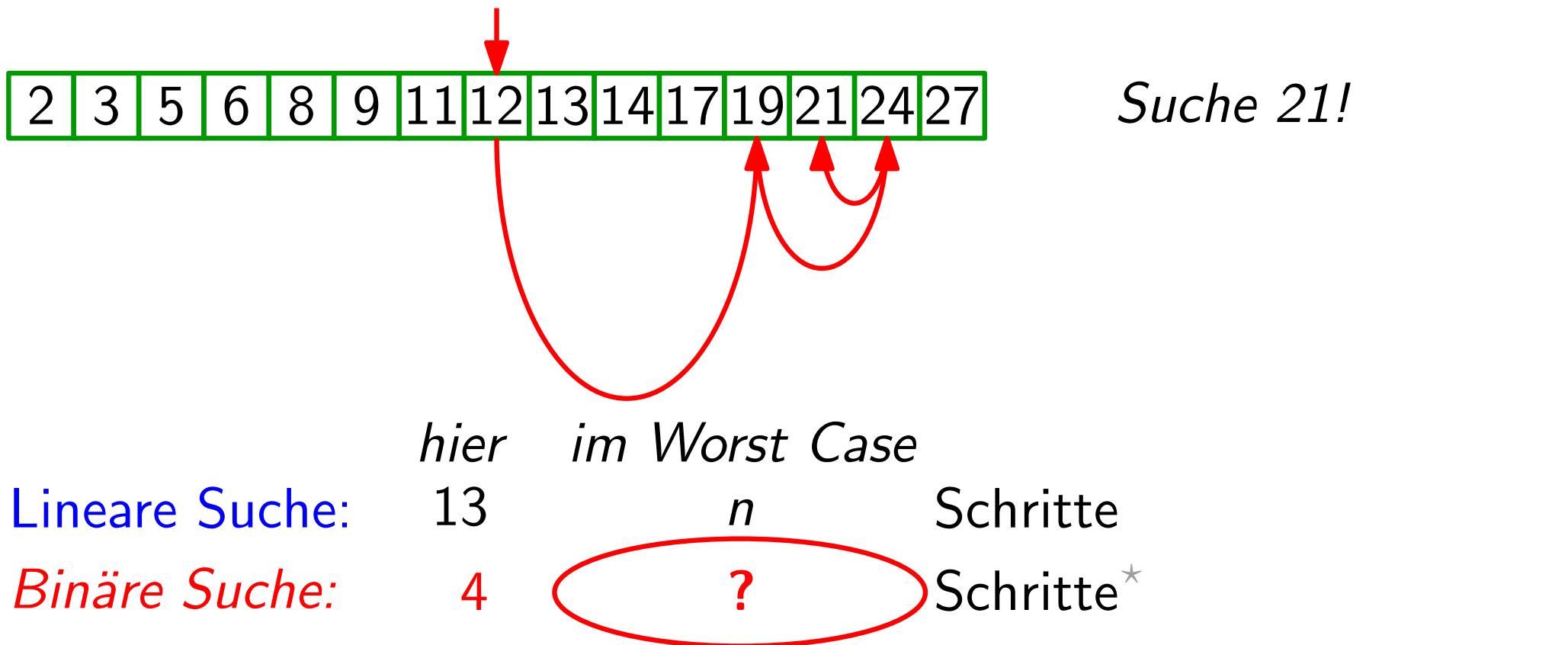
Suche im sortierten Feld



Suche im sortierten Feld

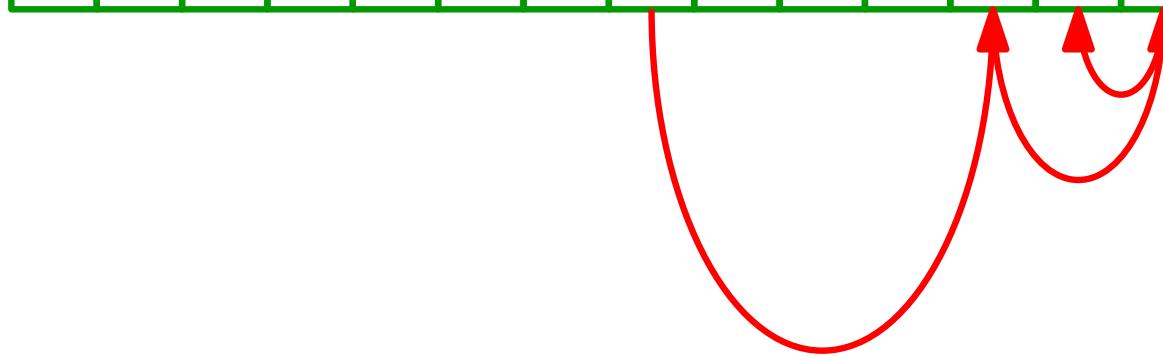
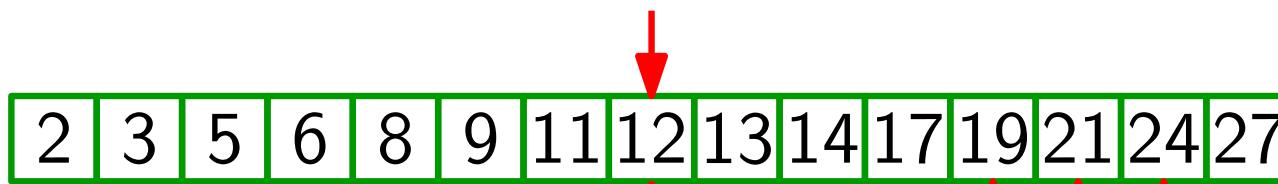


Suche im sortierten Feld



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

Suche im sortierten Feld



hier im Worst Case

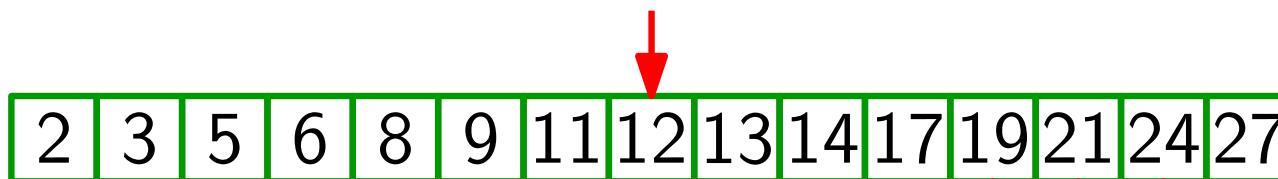
Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

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

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

Suche im sortierten Feld



Suche 21!

Lineare Suche: 13 Schritte

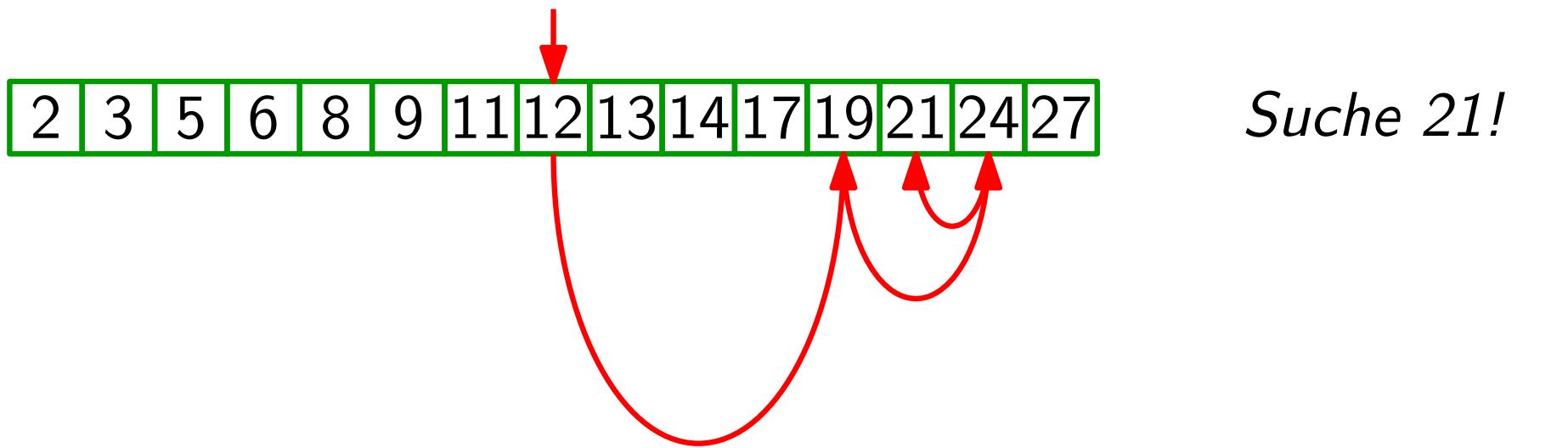
Binäre Suche: 4 Schritte*

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

genau: $T(n) \leq$ **Finden Sie die Rekursions(un)gleichung!!**

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

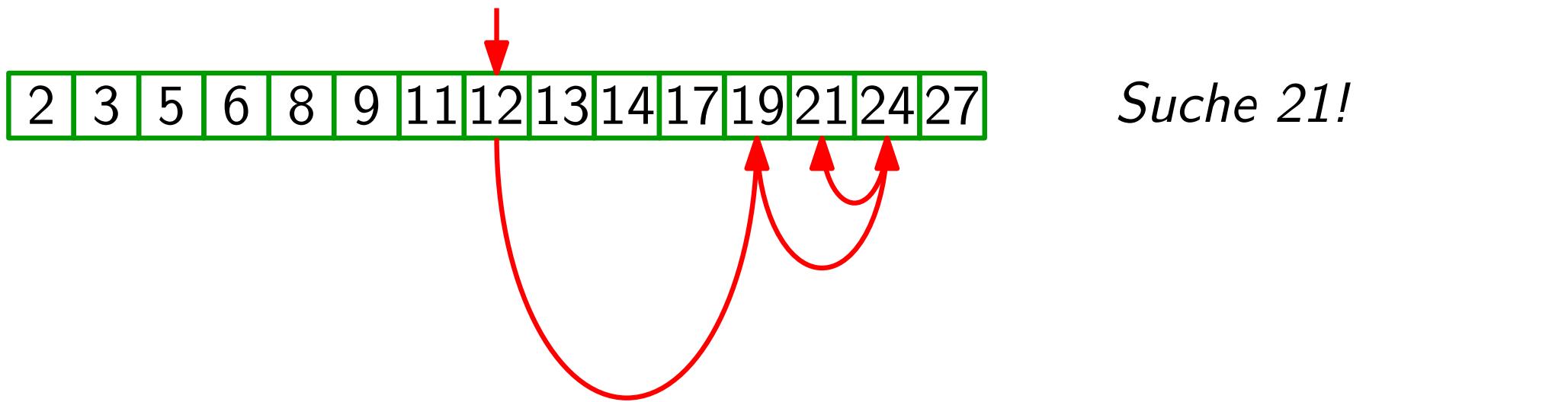
Binäre Suche: 4 Schritte*

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

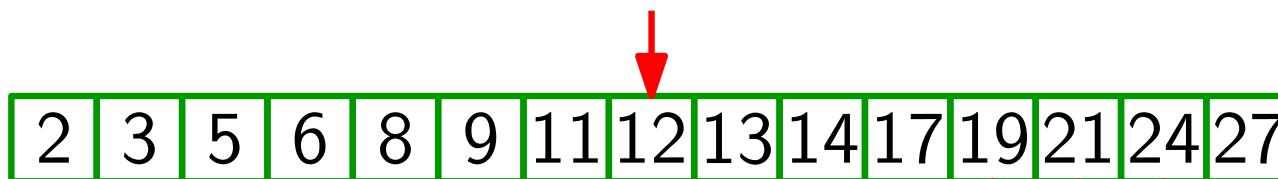
Binäre Suche: 4 Schritte*

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

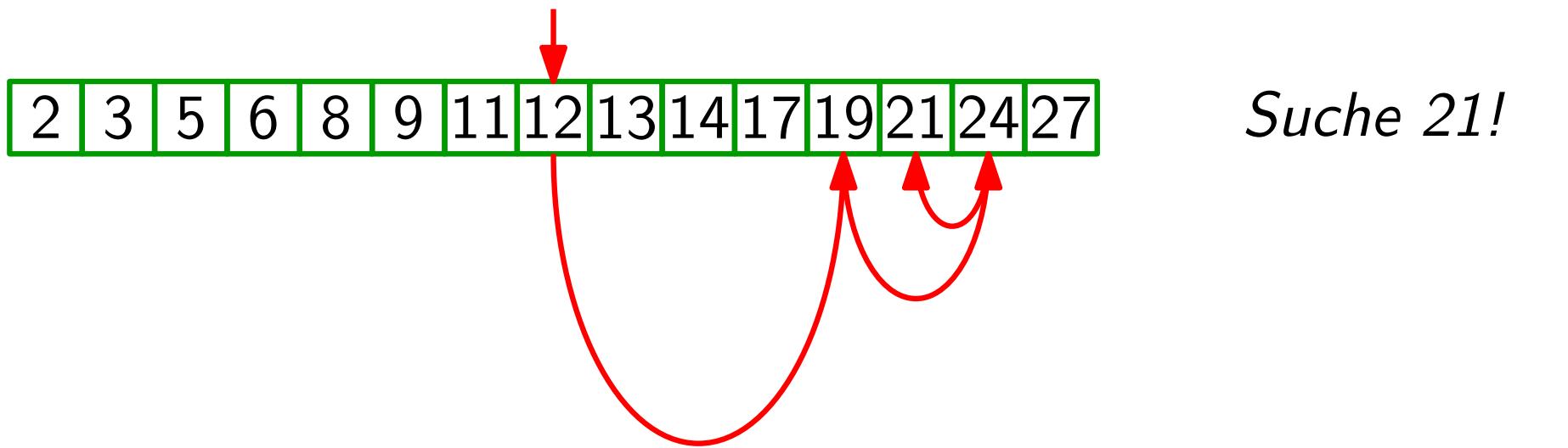
Binäre Suche: 4 Schritte*

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

$$\begin{aligned}
 \text{genau: } T(n) &\leq T(\lfloor n/2 \rfloor) + 1 \text{ und } T(1) = 1 \\
 &\leq \text{[redacted]} + 1
 \end{aligned}$$

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

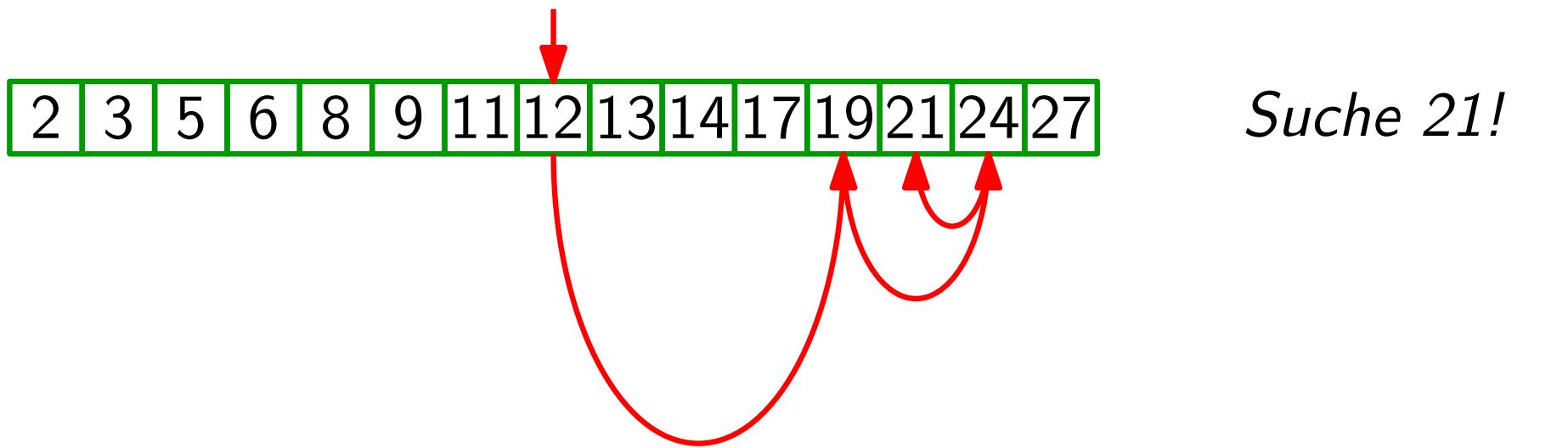
Binäre Suche: 4 Schritte*

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

$$\begin{aligned}
 \text{genau: } T(n) &\leq T(\lfloor n/2 \rfloor) + 1 \text{ und } T(1) = 1 \\
 &\leq T(\lfloor n/4 \rfloor) + 1 + 1
 \end{aligned}$$

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

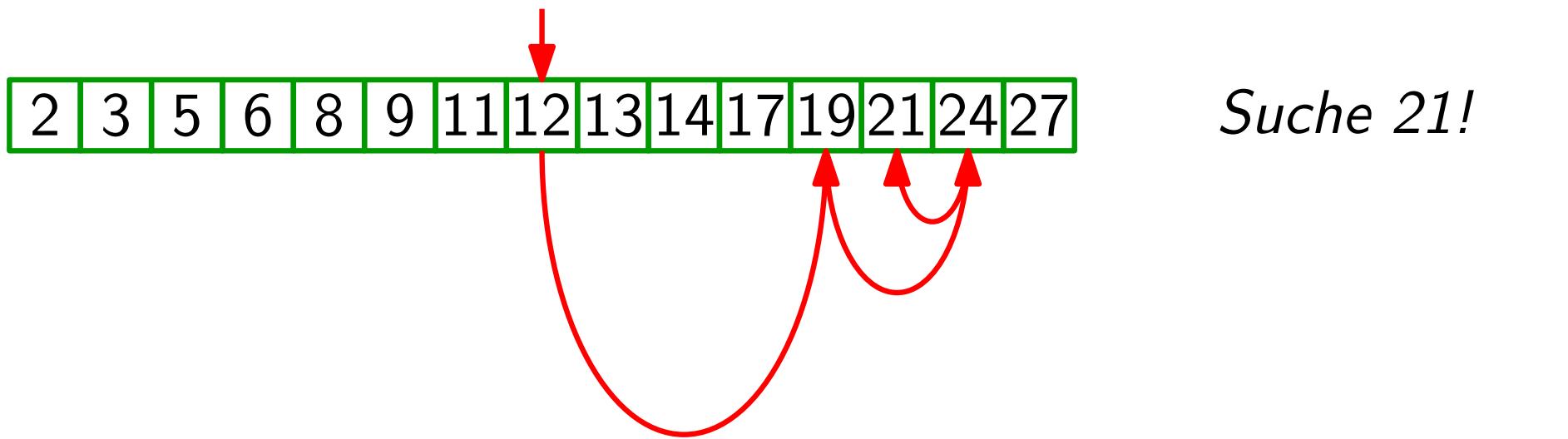
Binäre Suche: 4 Schritte*

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

$$\begin{aligned}
 \text{genau: } T(n) &\leq T(\lfloor n/2 \rfloor) + 1 \text{ und } T(1) = 1 \\
 &\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq
 \end{aligned}$$

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

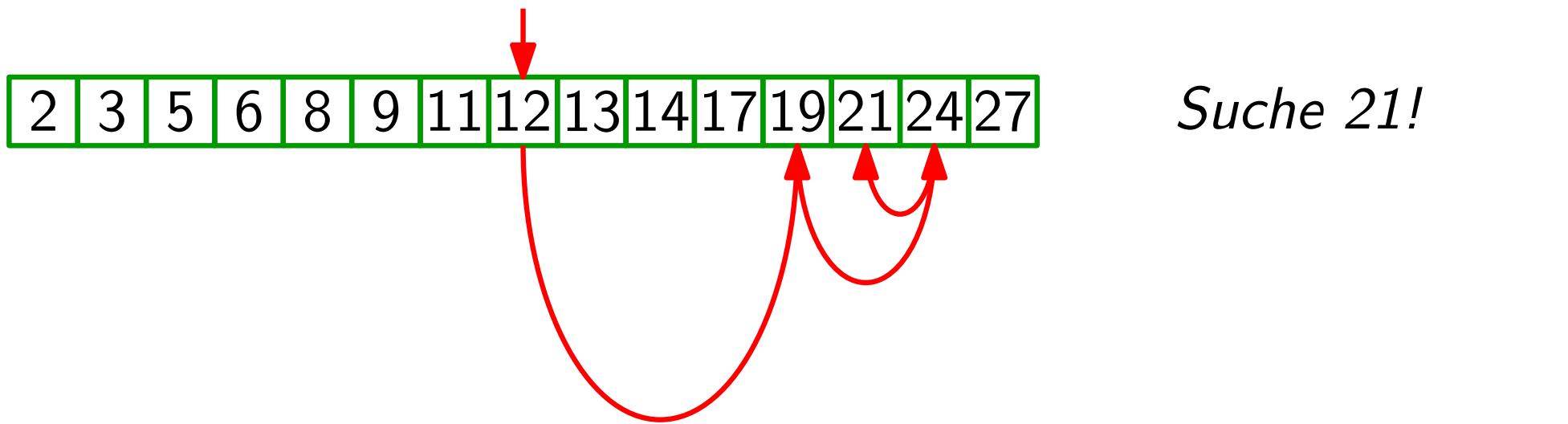
grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

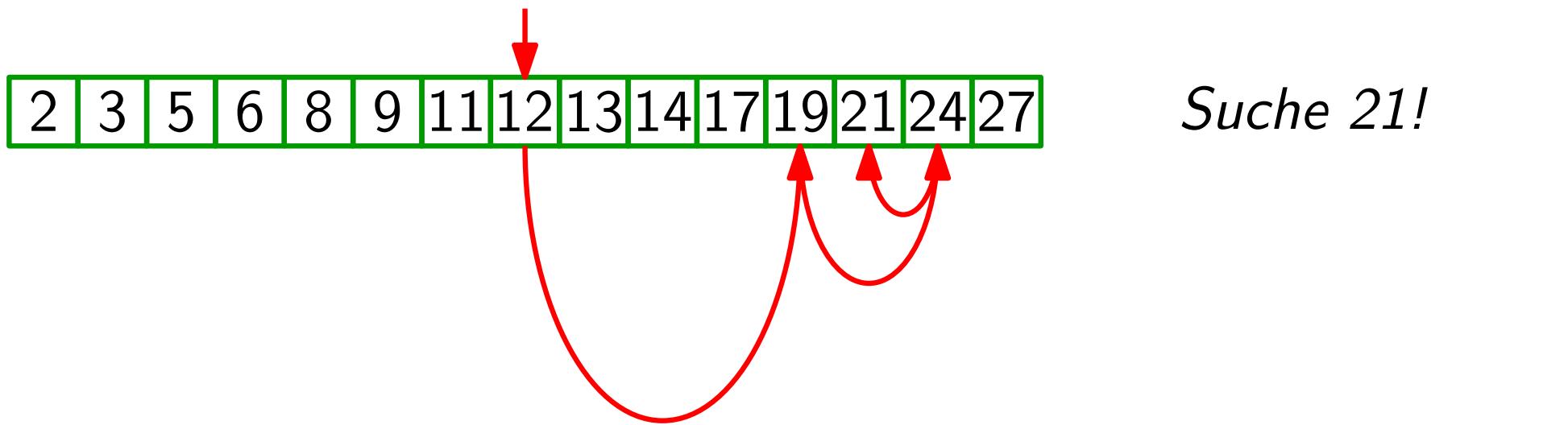
groß: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

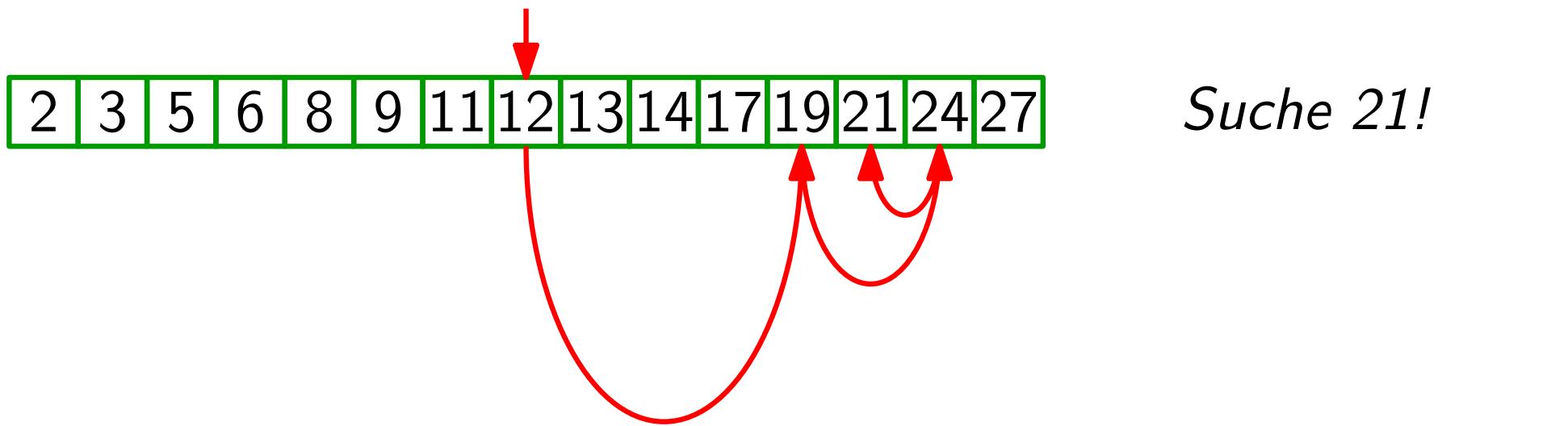
grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

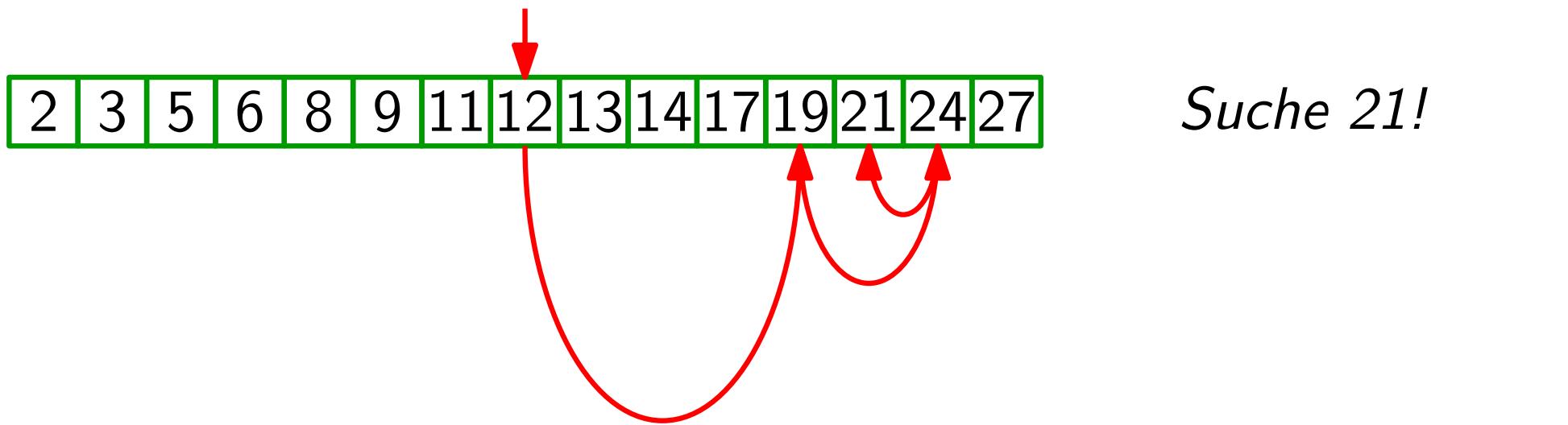
groß: Wie oft muss ich n halbieren, bis ich bei 1 bin?

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

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

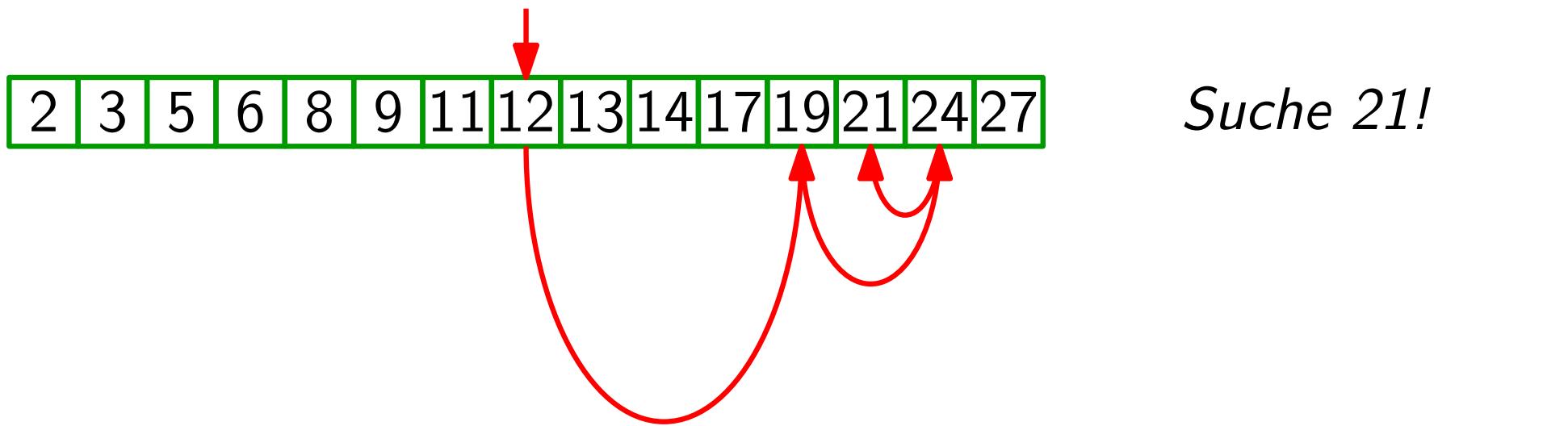
genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

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

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

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

Suche im sortierten Feld



Lineare Suche: 13 Schritte

Binäre Suche: 4 Schritte*

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

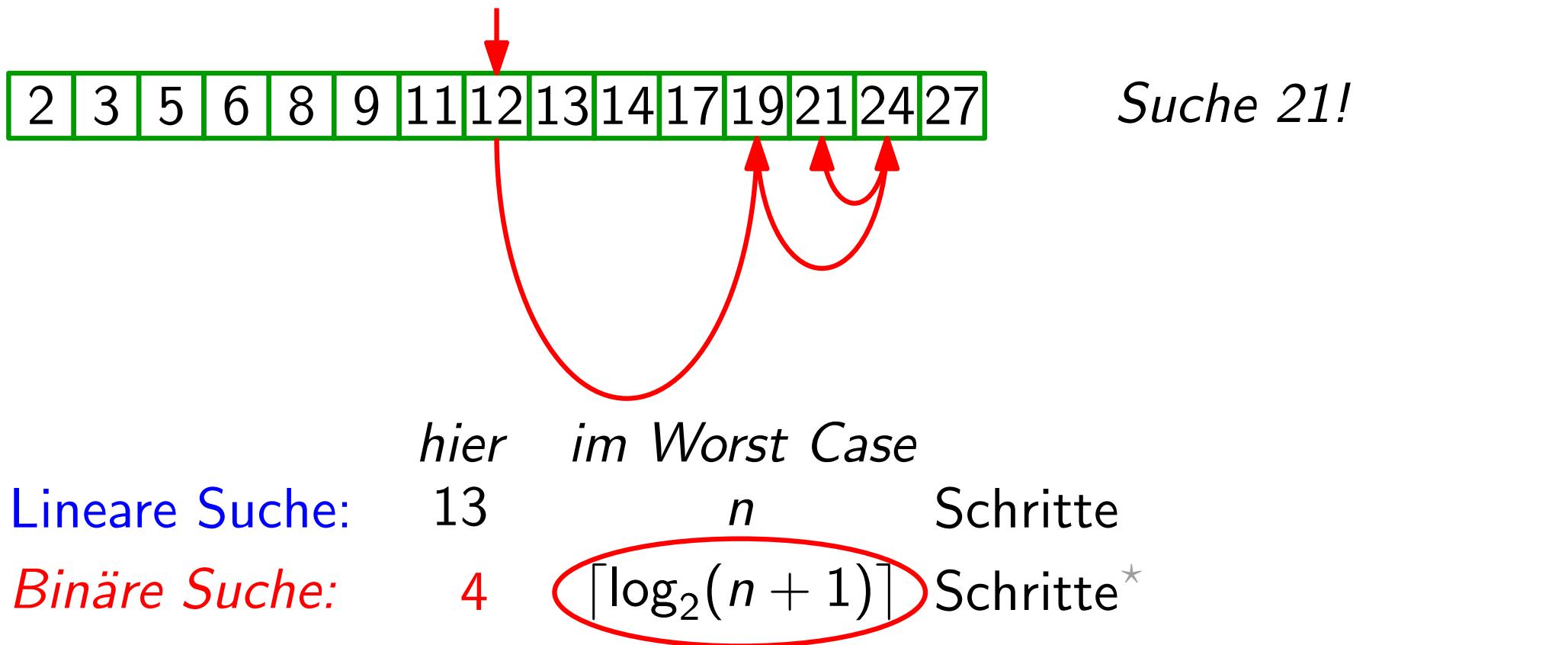
$$\text{genau: } T(n) \leq T(\lfloor n/2 \rfloor) + 1 \text{ und } T(1) = 1$$

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

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

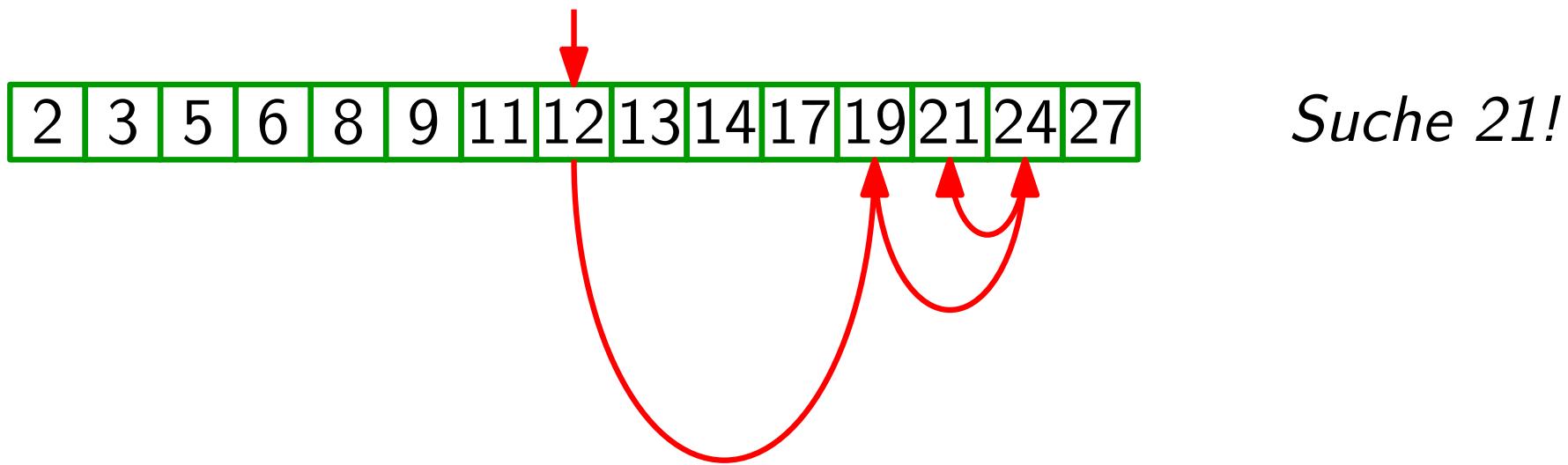
*) Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

Suche im sortierten Feld



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

Suche im sortierten Feld



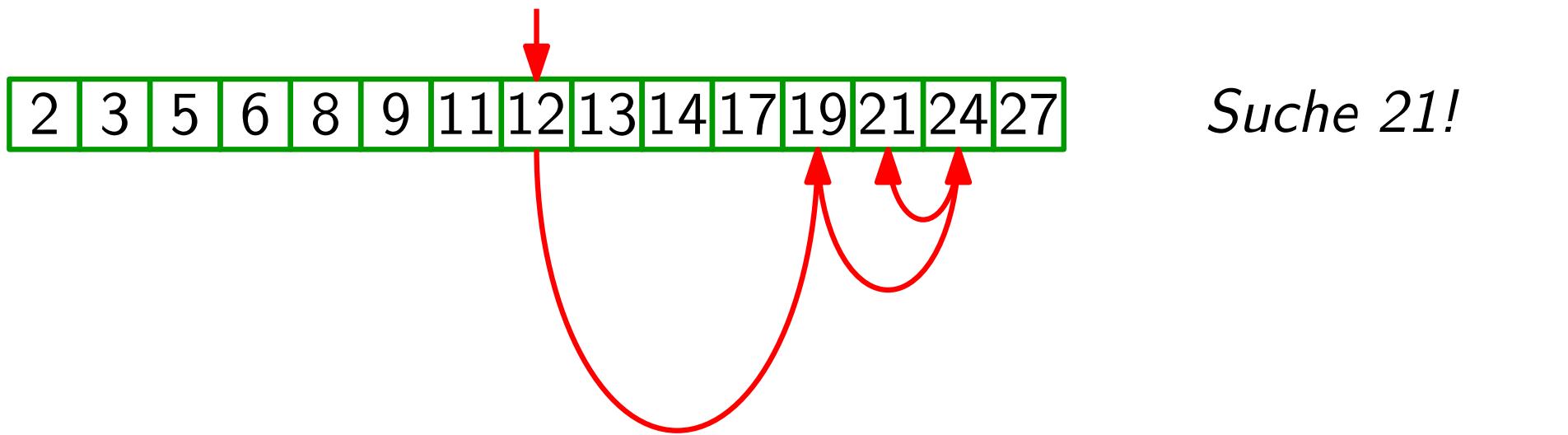
hier im Worst Case

Lineare Suche: 13 Schritte

Binäre Suche: 4 $\lceil \log_2(n + 1) \rceil$ Schritte^{*} 20

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

Suche im sortierten Feld



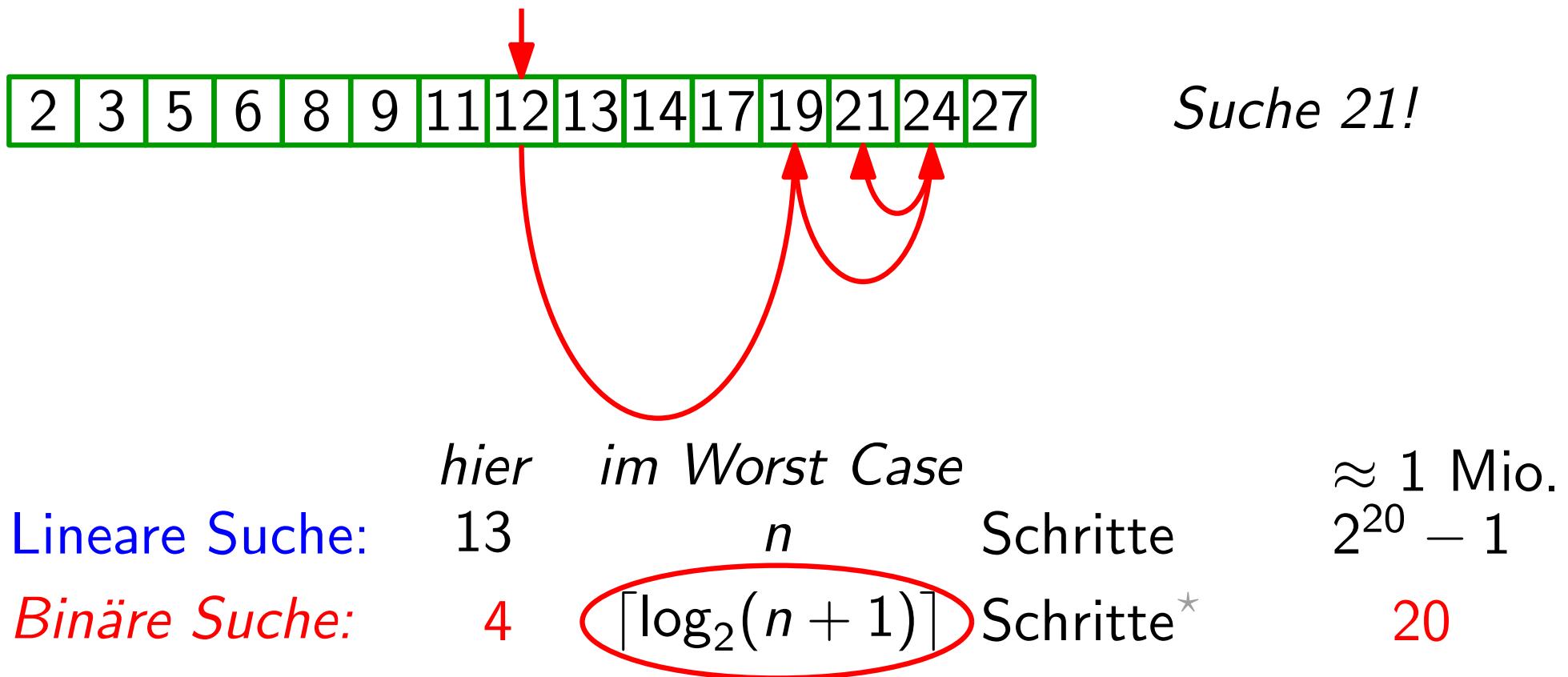
hier im Worst Case

Lineare Suche: 13 Schritte $2^{20} - 1$

Binäre Suche: 4 Schritte^{*} $\lceil \log_2(n + 1) \rceil$ 20

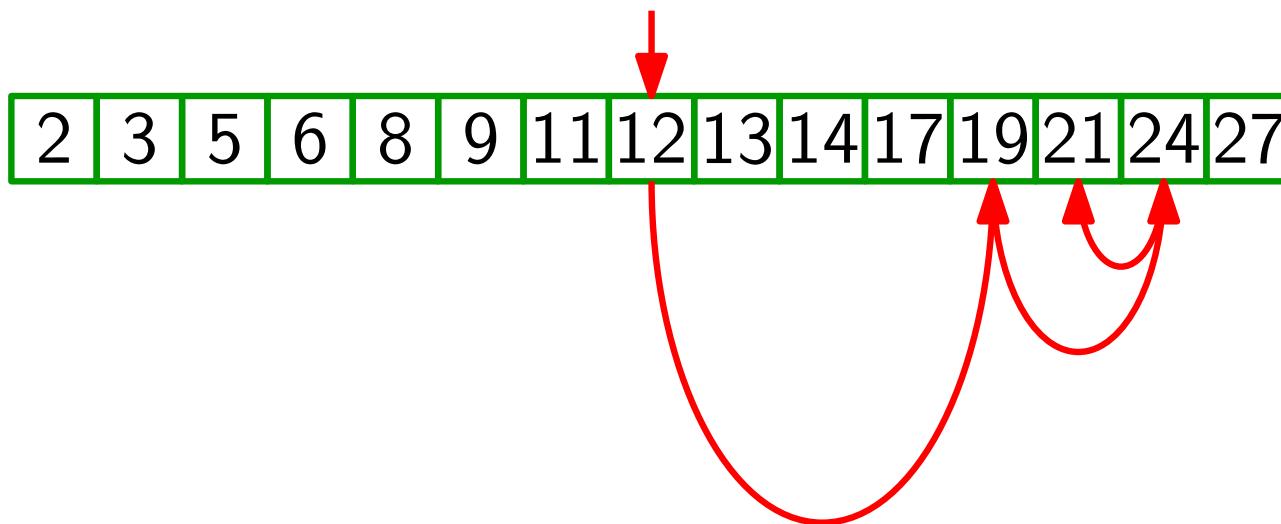
^{*}) Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. $=$ und $<$).

Suche im sortierten Feld



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

Suche im sortierten Feld

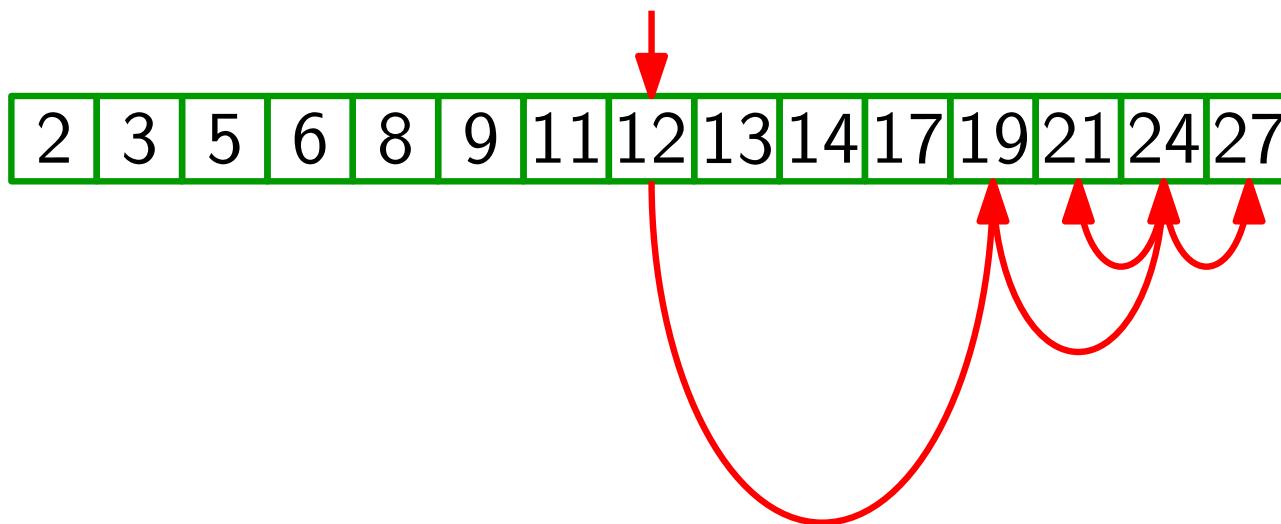


Suche 21!
Suche 27!

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

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

Suche im sortierten Feld



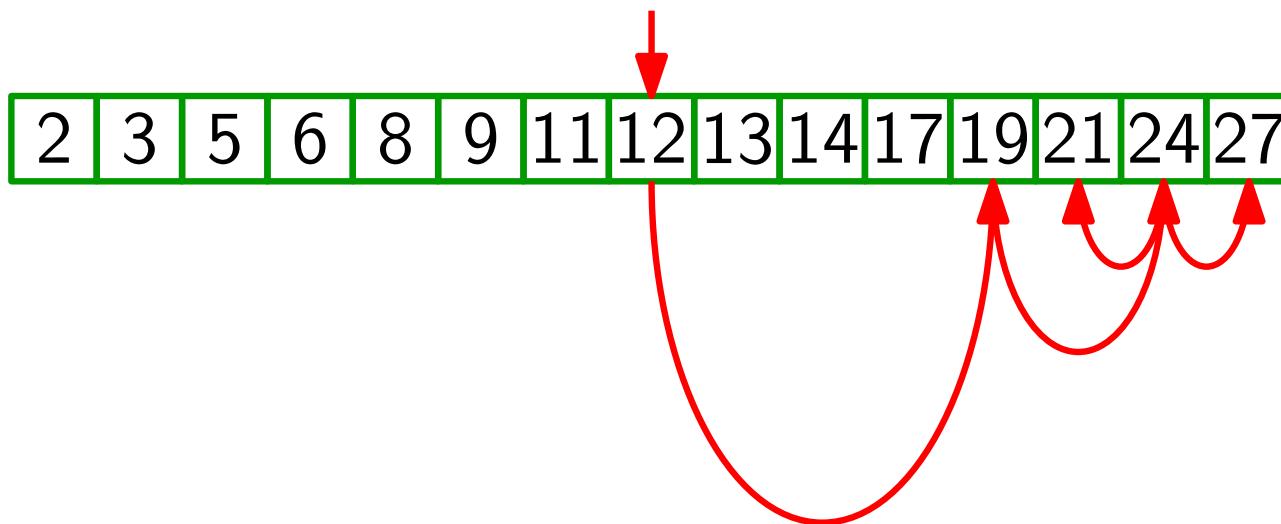
Suche 21!
Suche 27!

Lineare Suche: 13 im Worst Case Schritte ≈ 1 Mio.
 $2^{20} - 1$

Binäre Suche: 4 $\lceil \log_2(n + 1) \rceil$ Schritte* 20

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

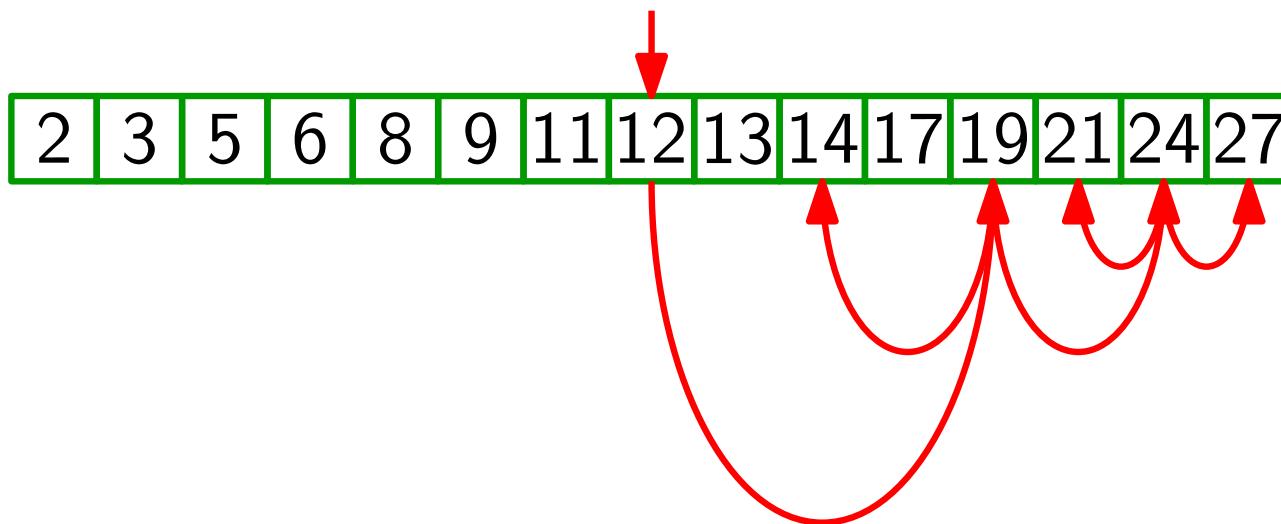
Suche im sortierten Feld



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

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

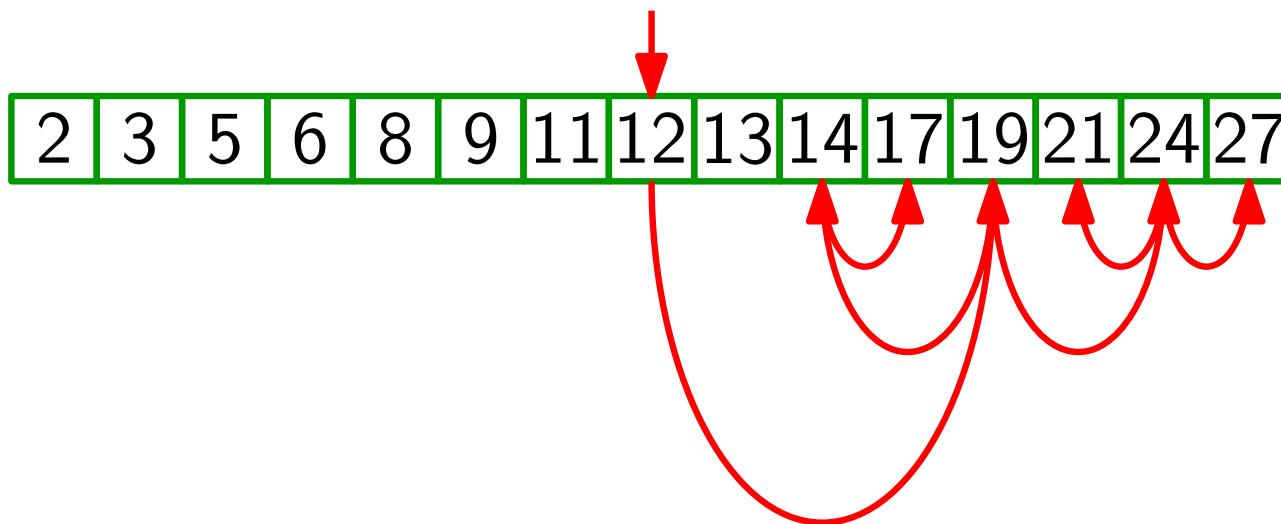
Suche im sortierten Feld



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

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

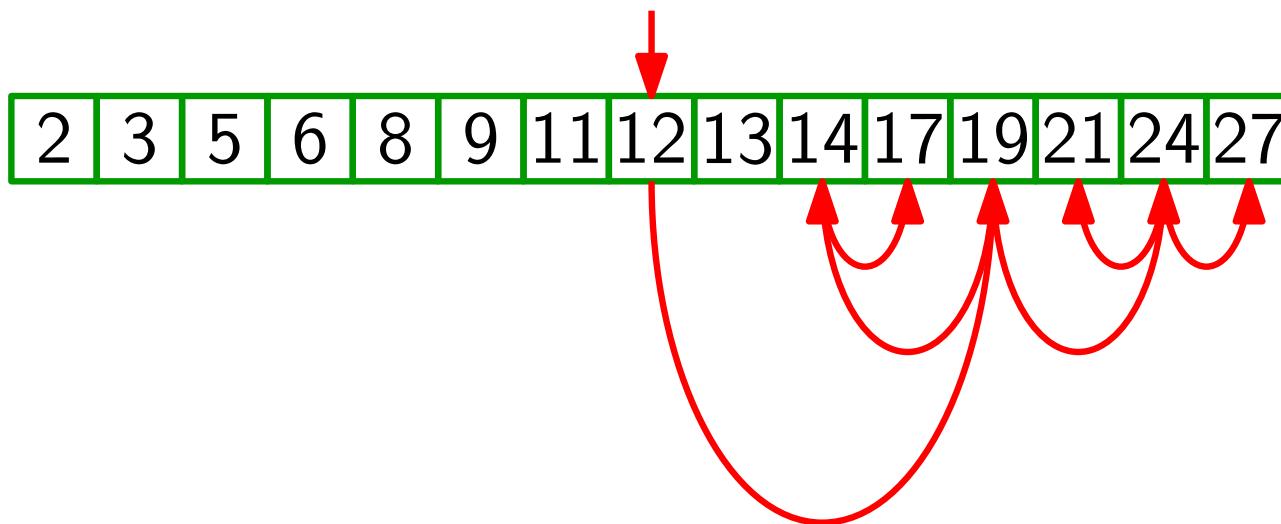
Suche im sortierten Feld



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

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

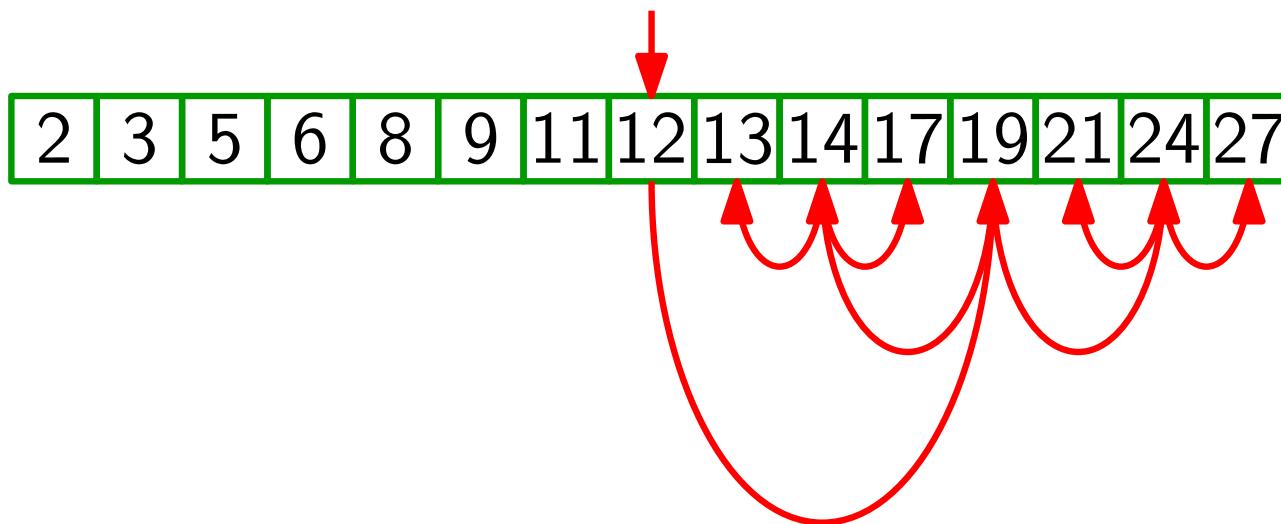
Suche im sortierten Feld



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

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

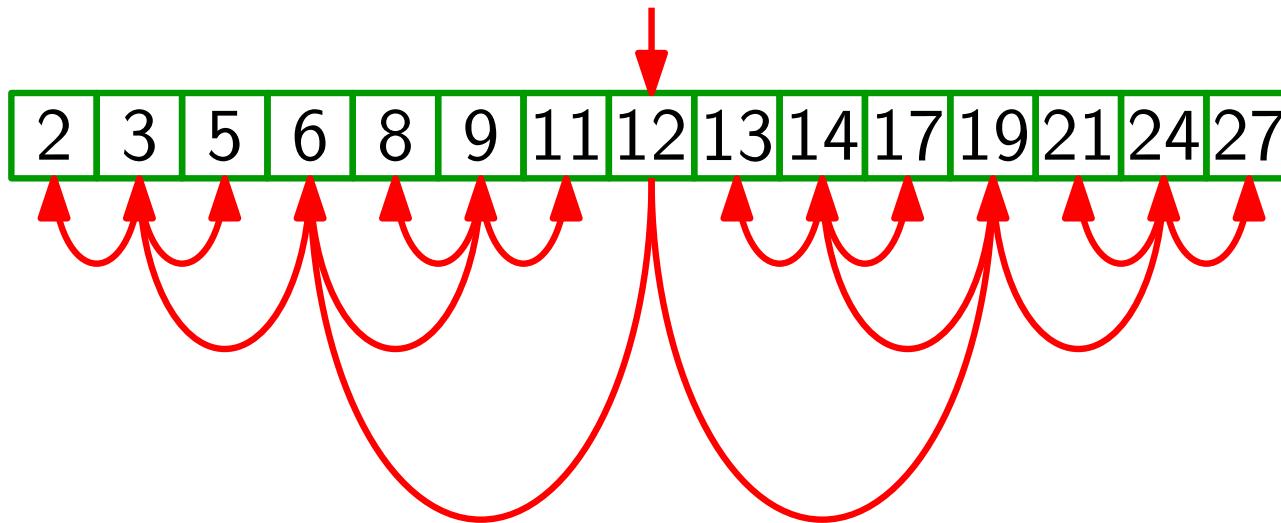
Suche im sortierten Feld



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

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

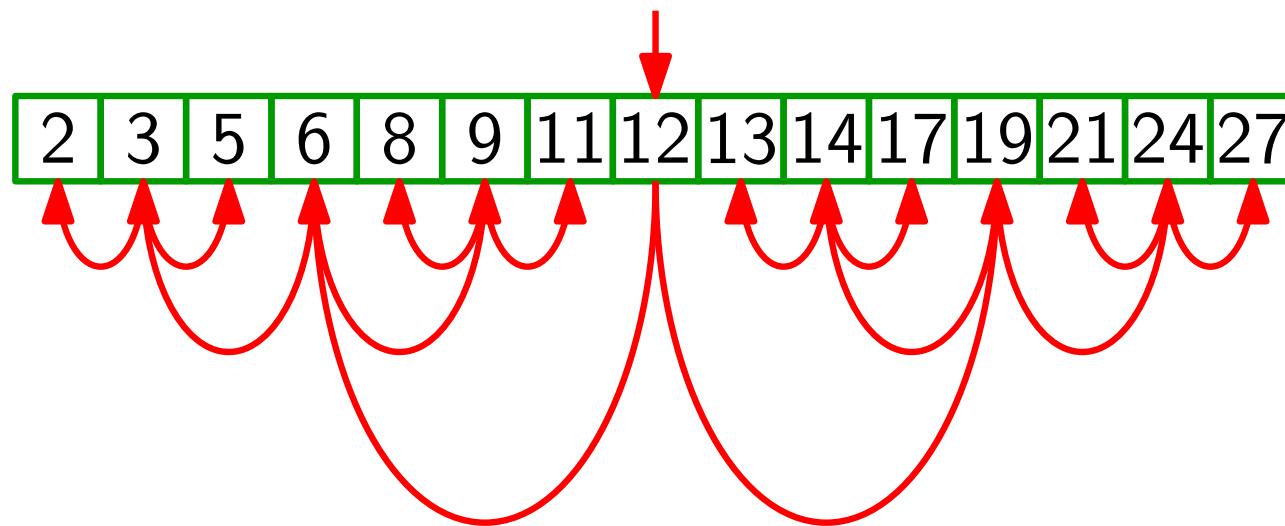
Suche im sortierten Feld



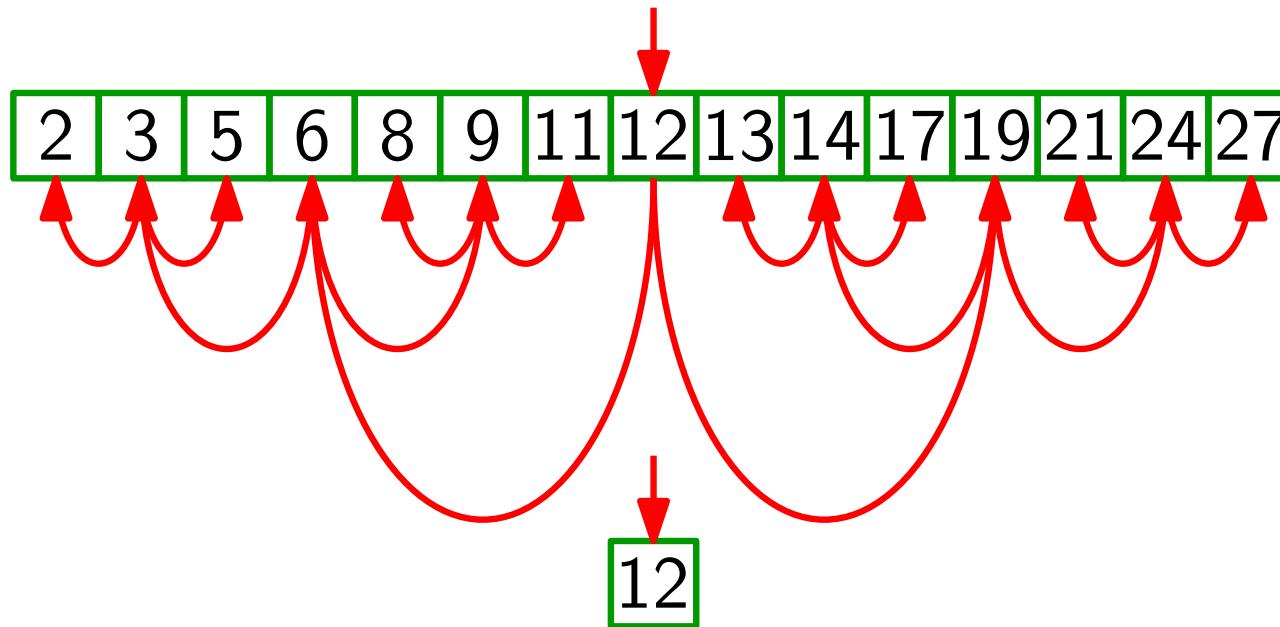
Lineare Suche:	13	<i>n</i>	Schritte	$\approx 1 \text{ Mio.}$ $2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte*	20

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

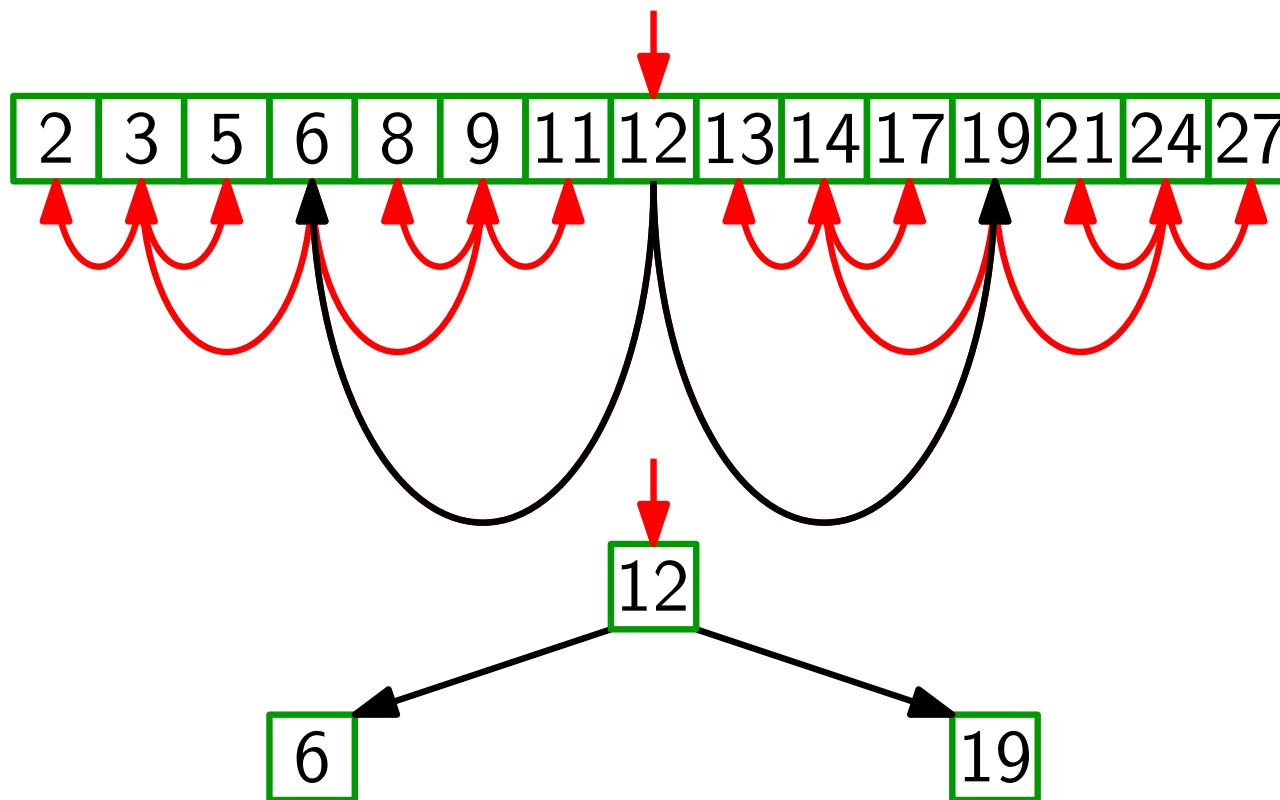
Suche im sortierten Feld



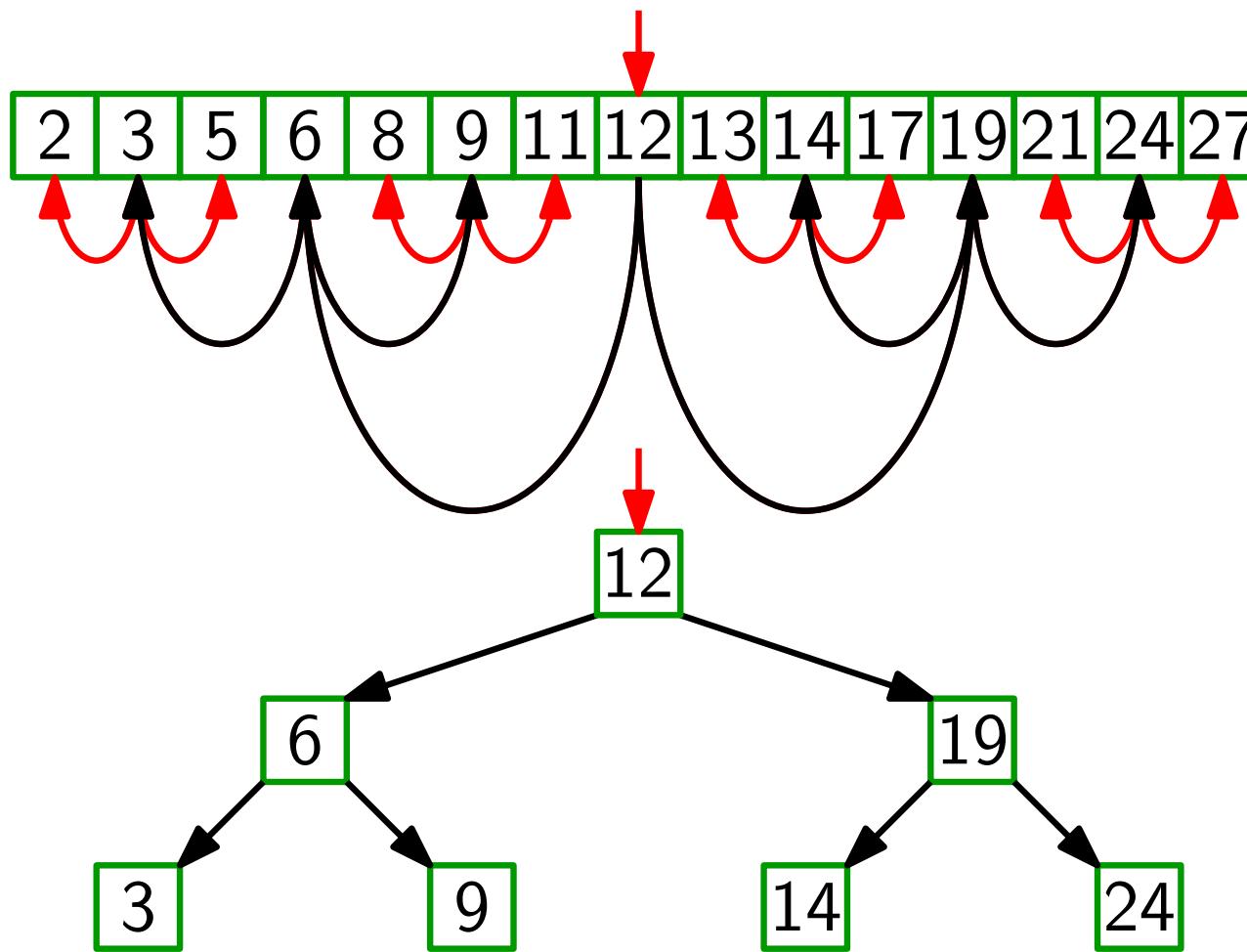
Suche im sortierten Feld



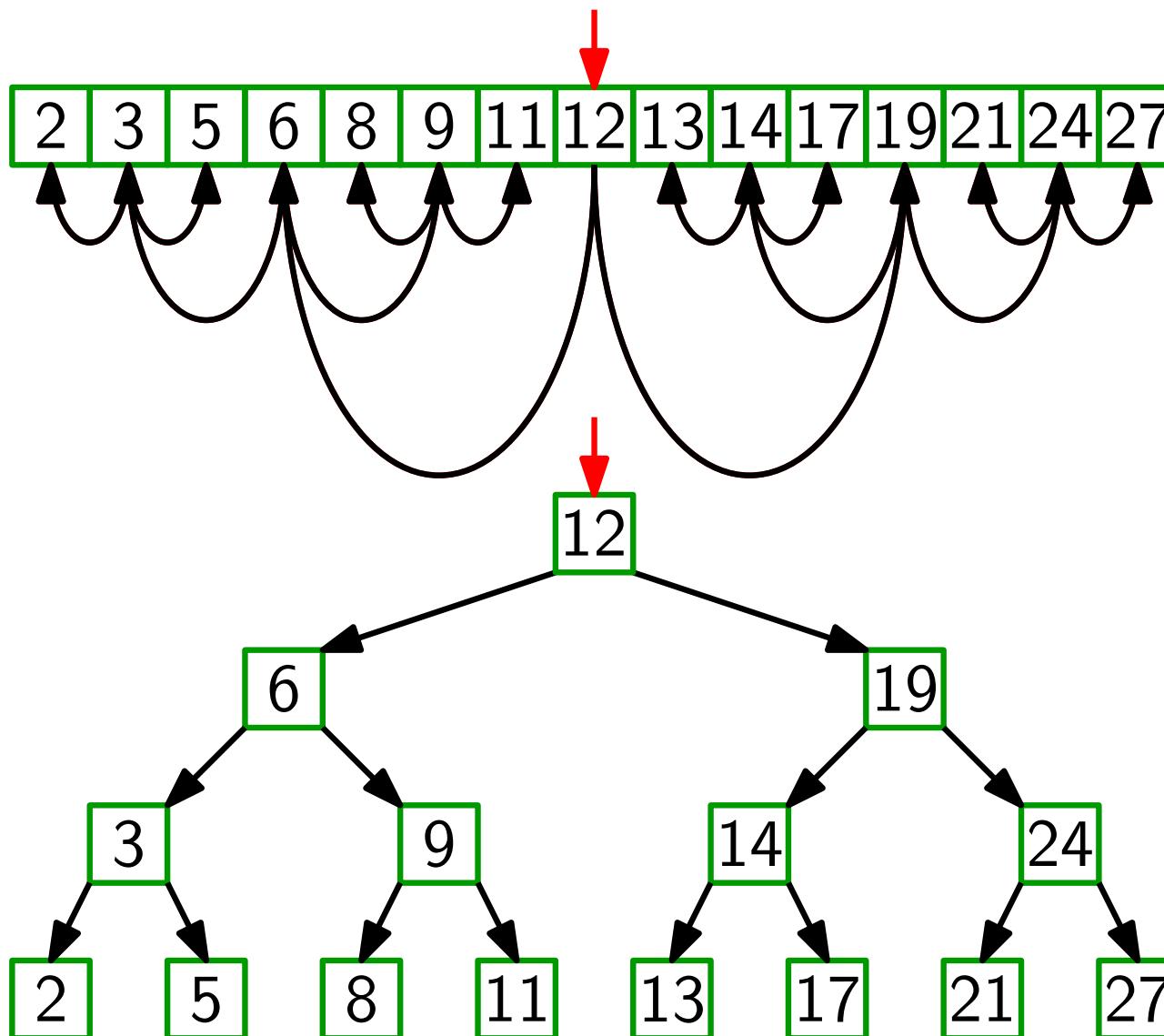
Suche im sortierten Feld



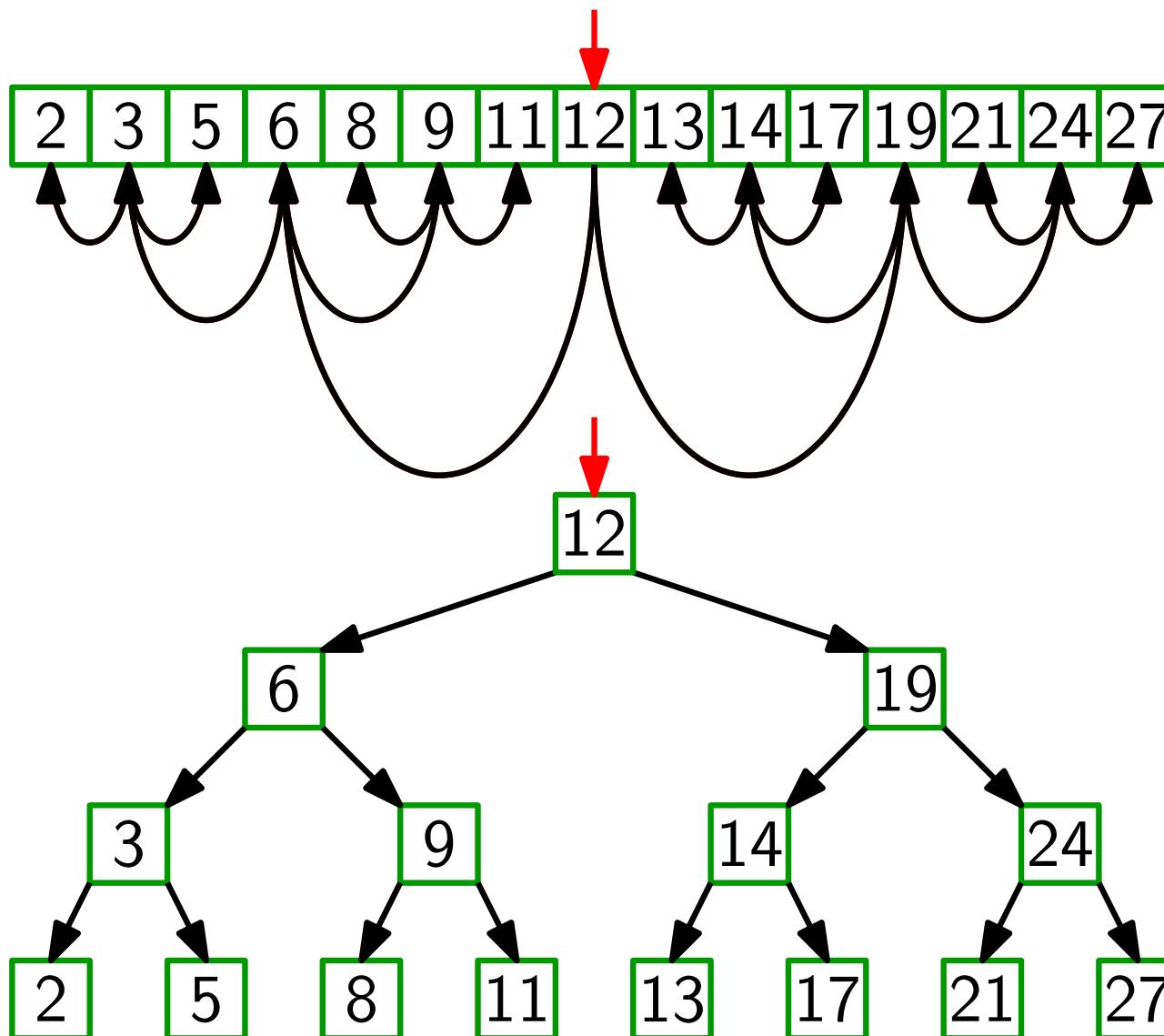
Suche im sortierten Feld



Suche im sortierten Feld

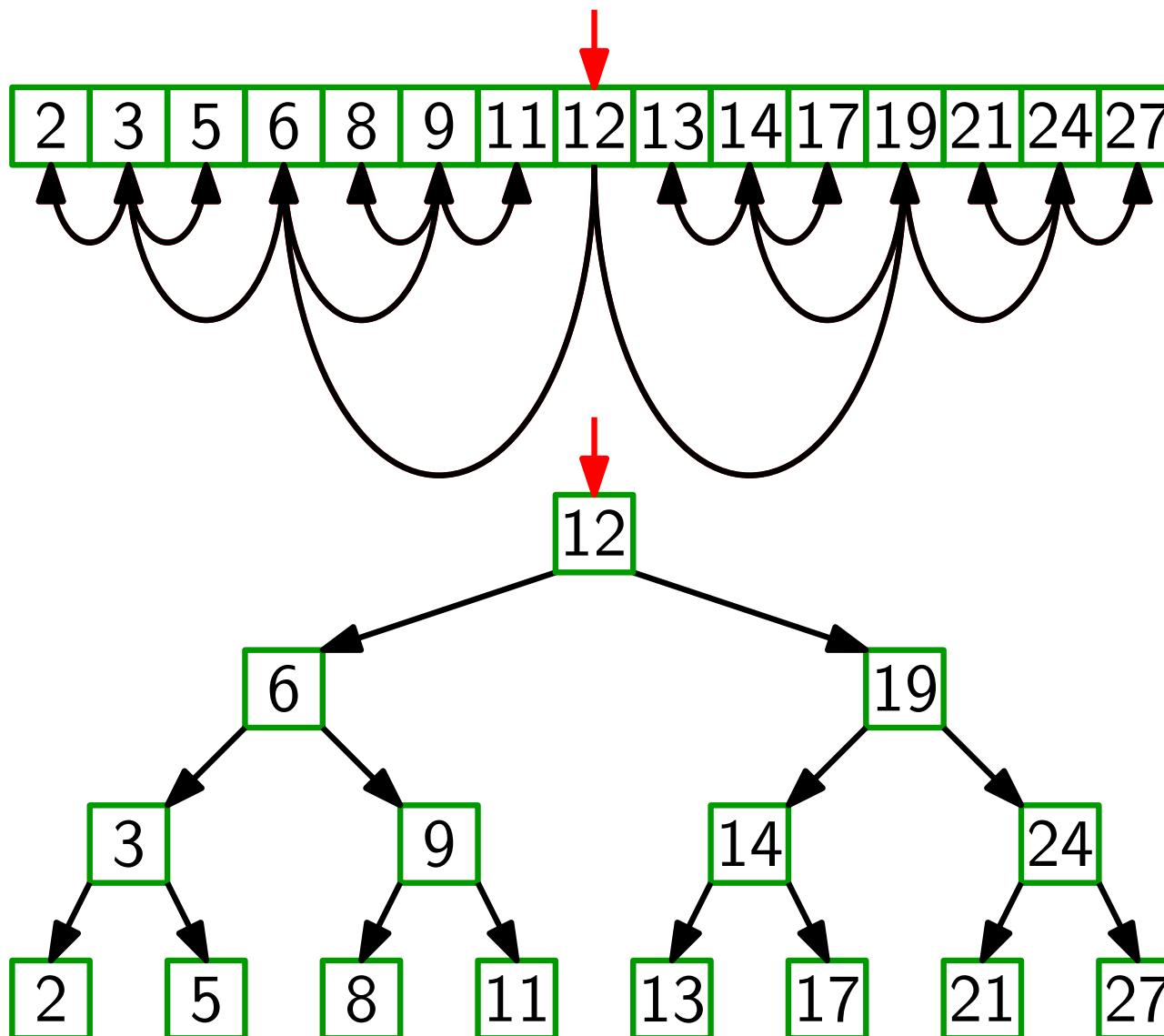


Suche im sortierten Feld



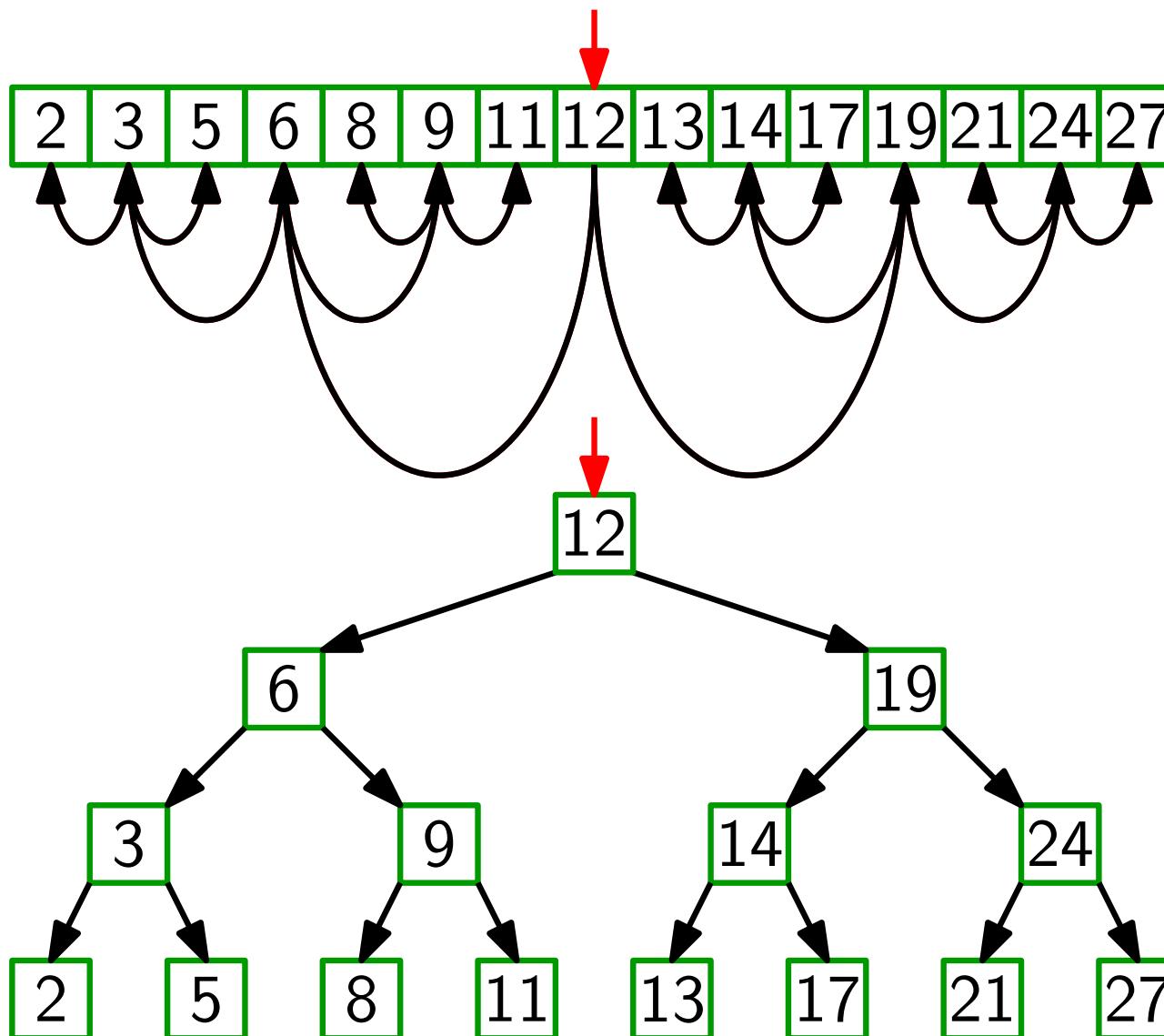
*Binärer
Suchbaum*

Suche im sortierten Feld



*Binärer
Suchbaum*

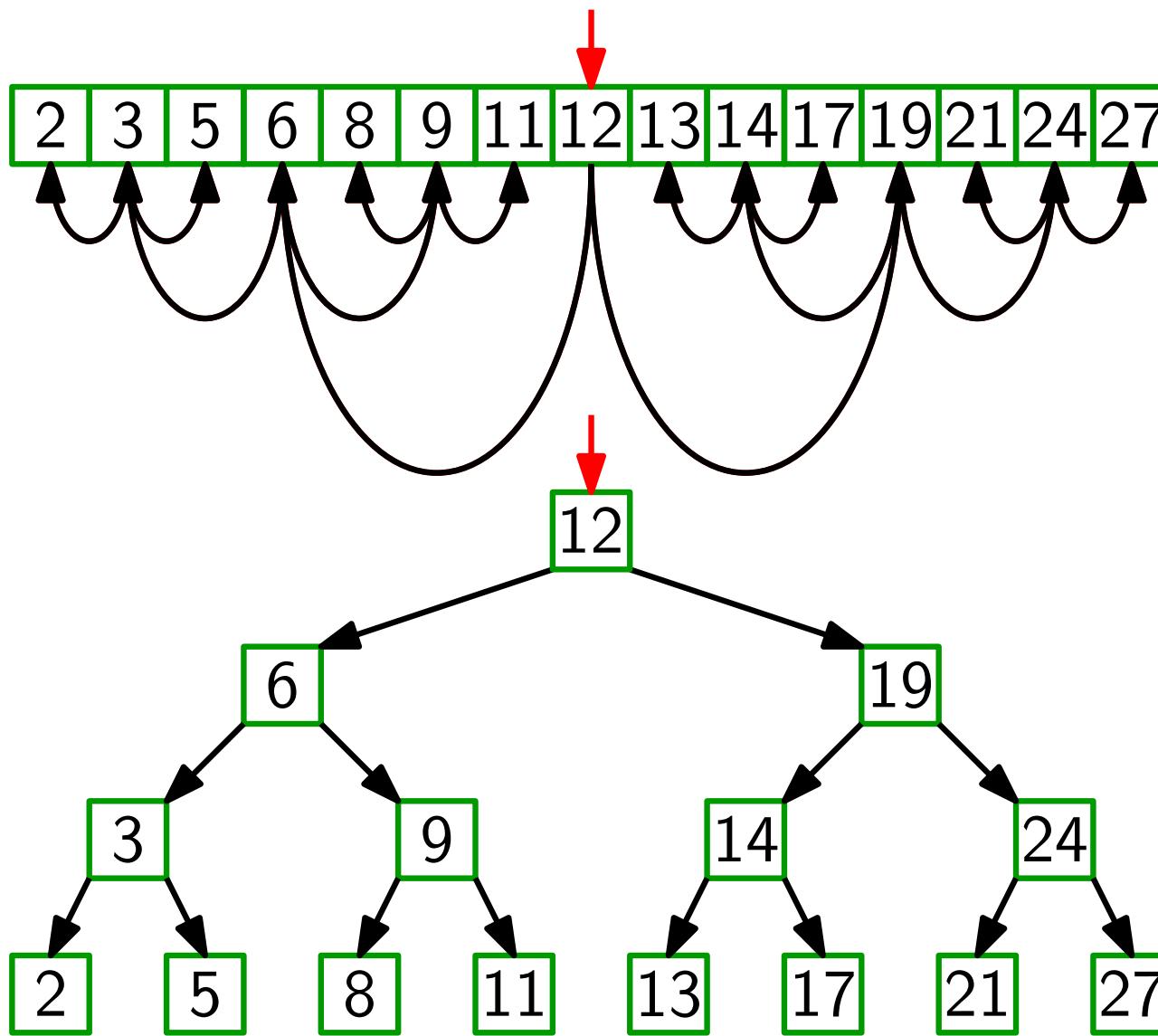
Suche im sortierten Feld



*Binärer
Suchbaum*

zusammen-
hängender
Wald

Suche im sortierten Feld

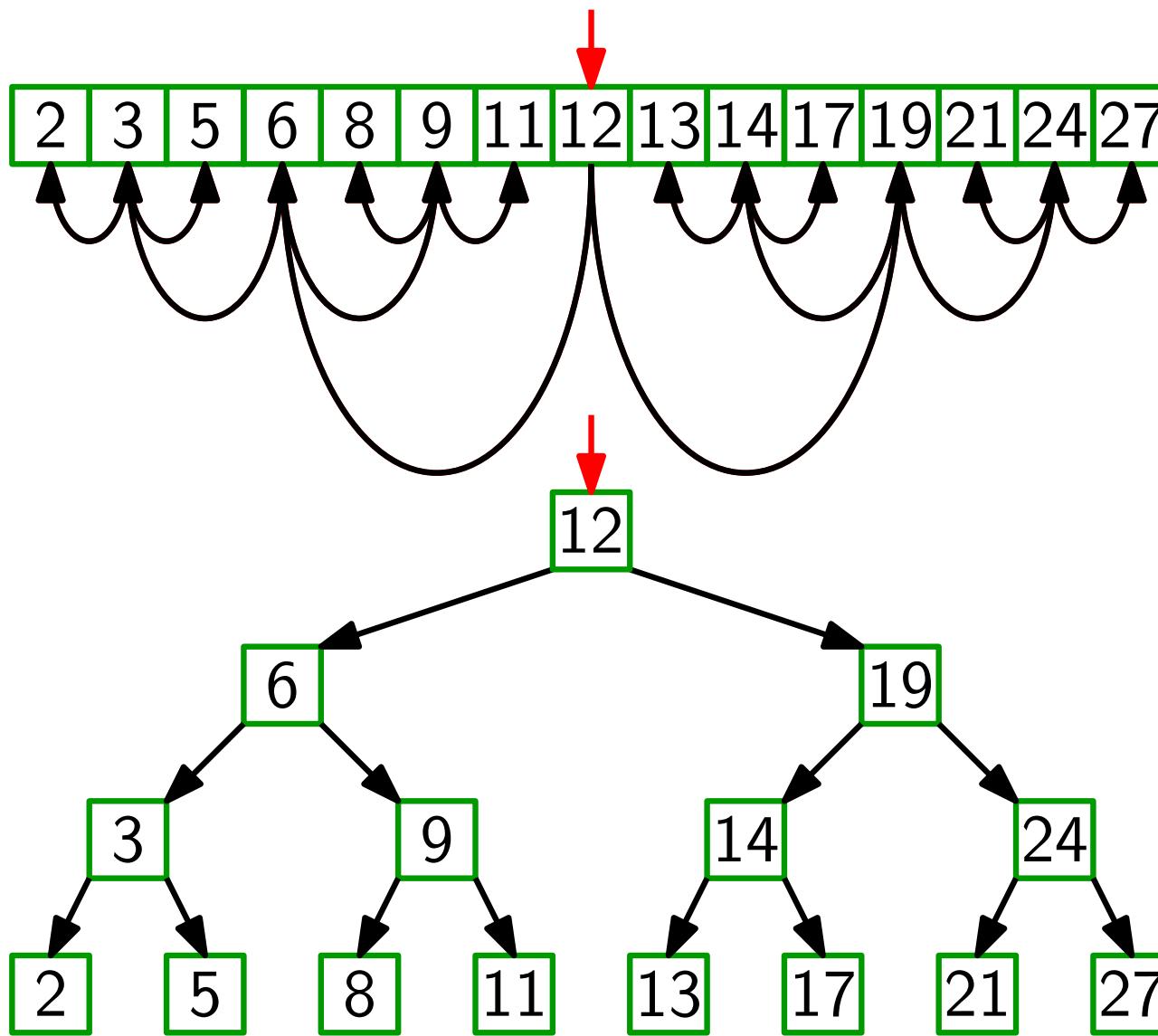


*Binärer
Suchbaum*

zusammen-
hängender
Wald

kreisfreier
Graph

Suche im sortierten Feld

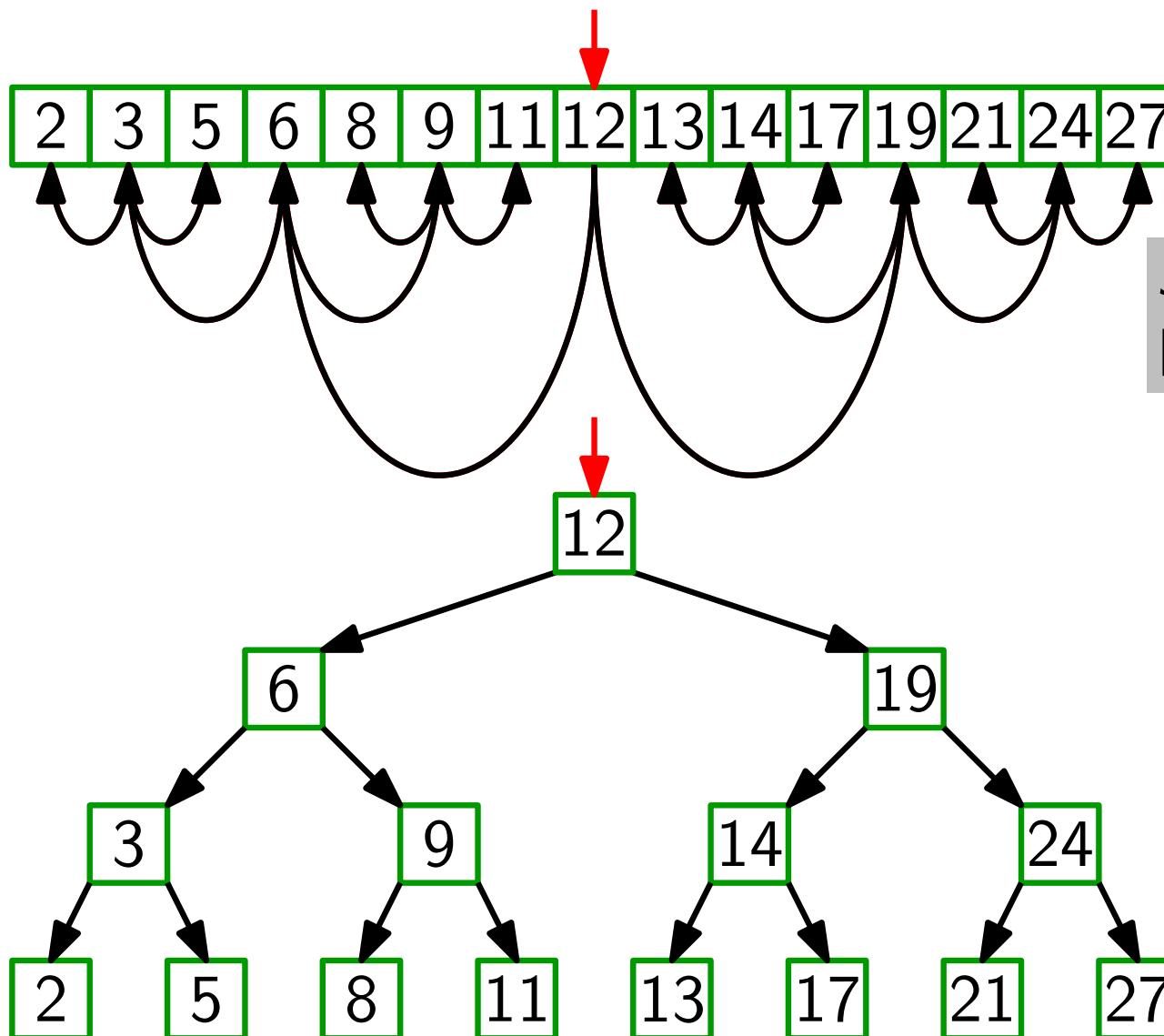


*Binärer
Suchbaum*

zusammen-
hängender
Wald

kreisfreier
Graph

Suche im sortierten Feld



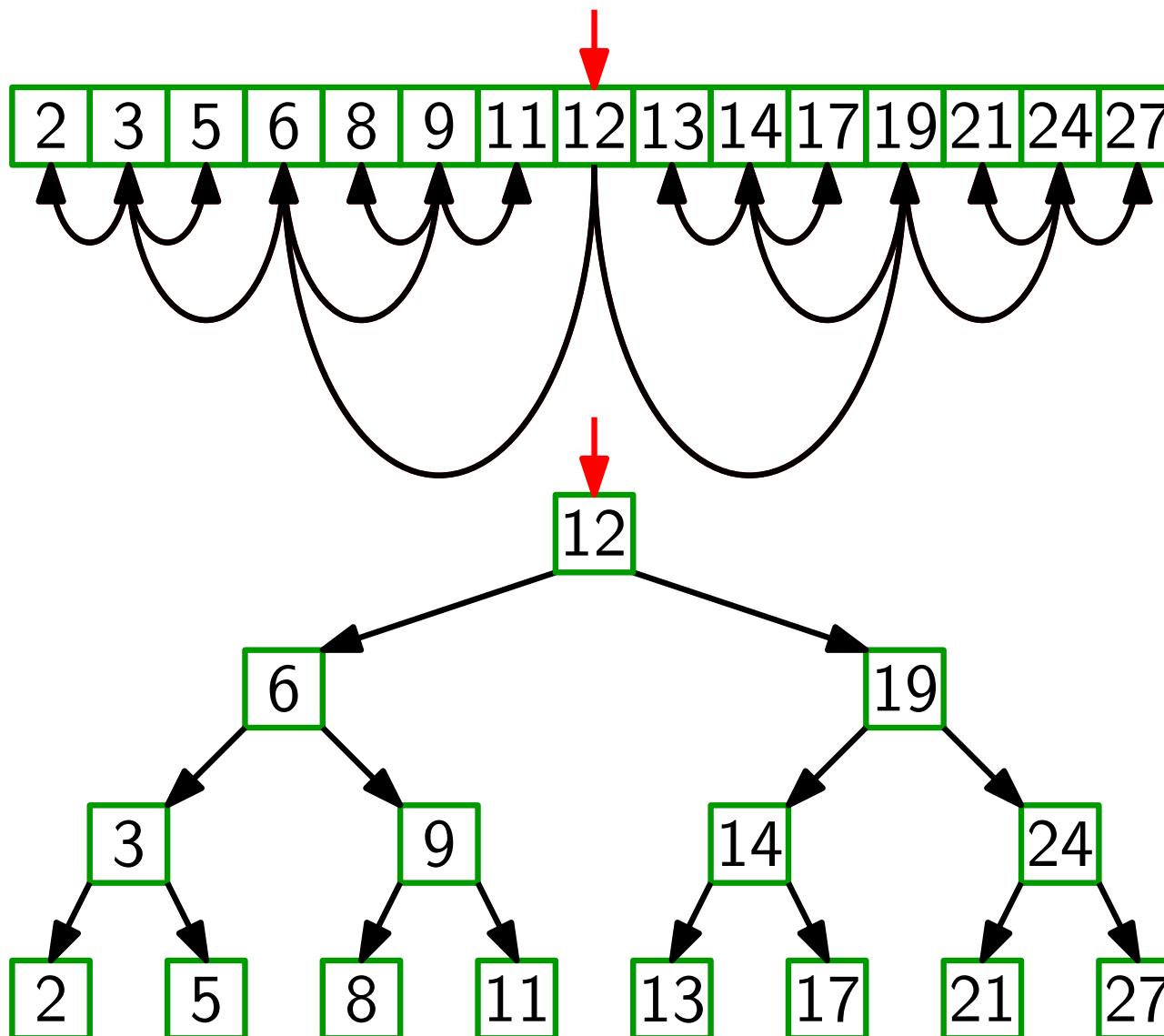
Jeder Knoten hat
höchstens zwei Kinder.

Binärer
Suchbaum

zusammen-
hängender
Wald

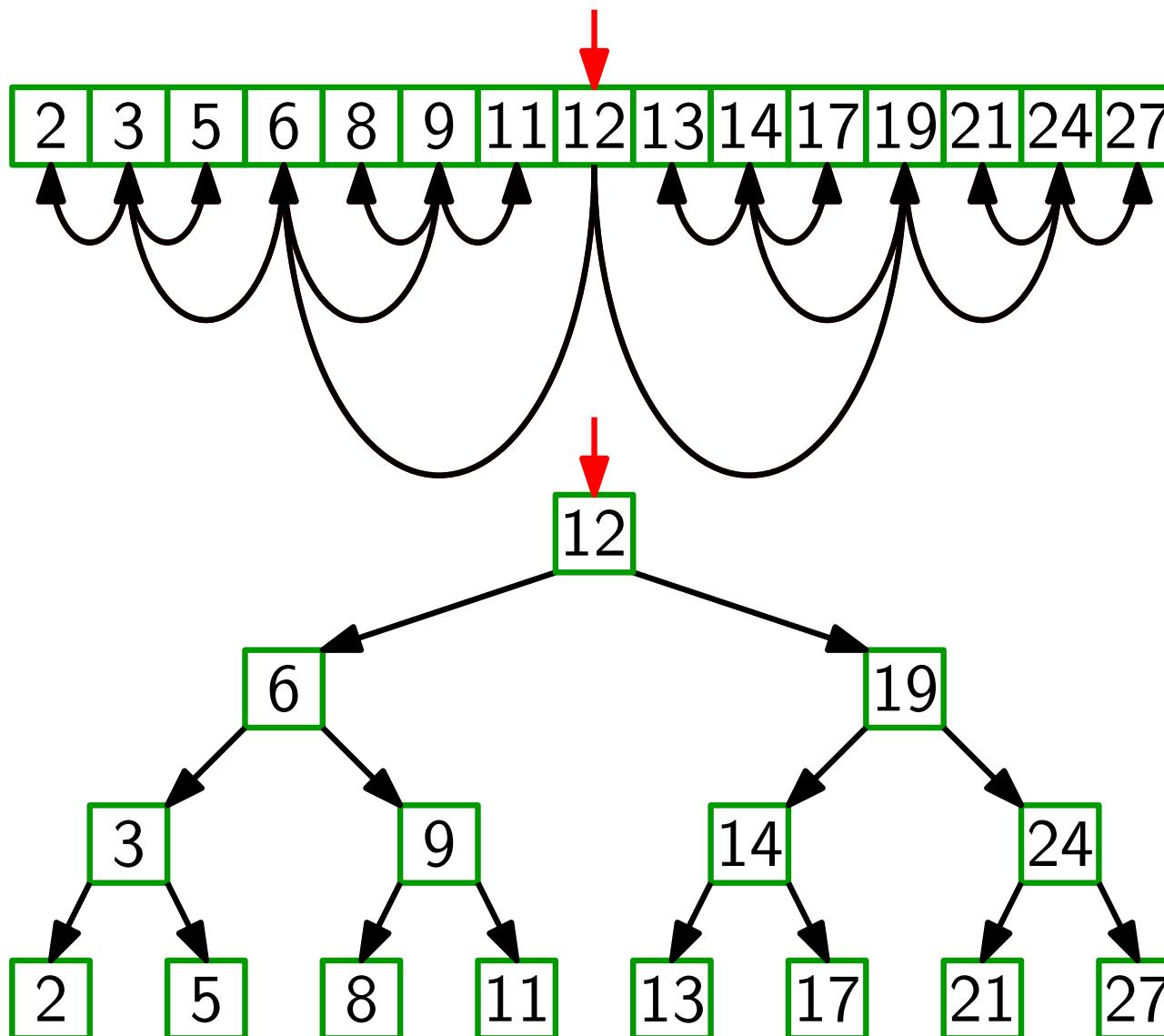
kreisfreier
Graph

Suche im sortierten Feld



*Binärer
Suchbaum*

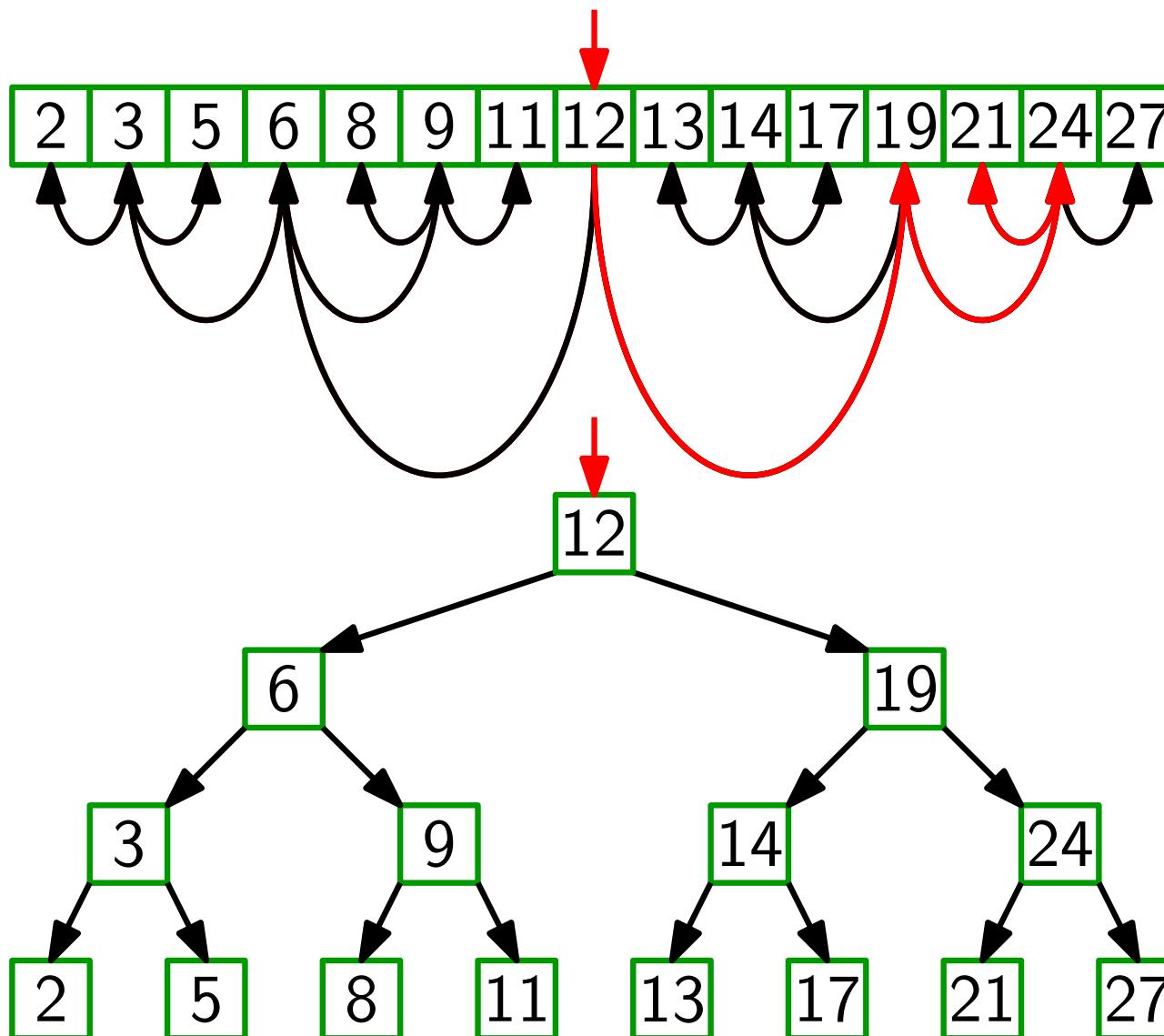
Suche im sortierten Feld



Suche 21!

Binärer
Suchbaum

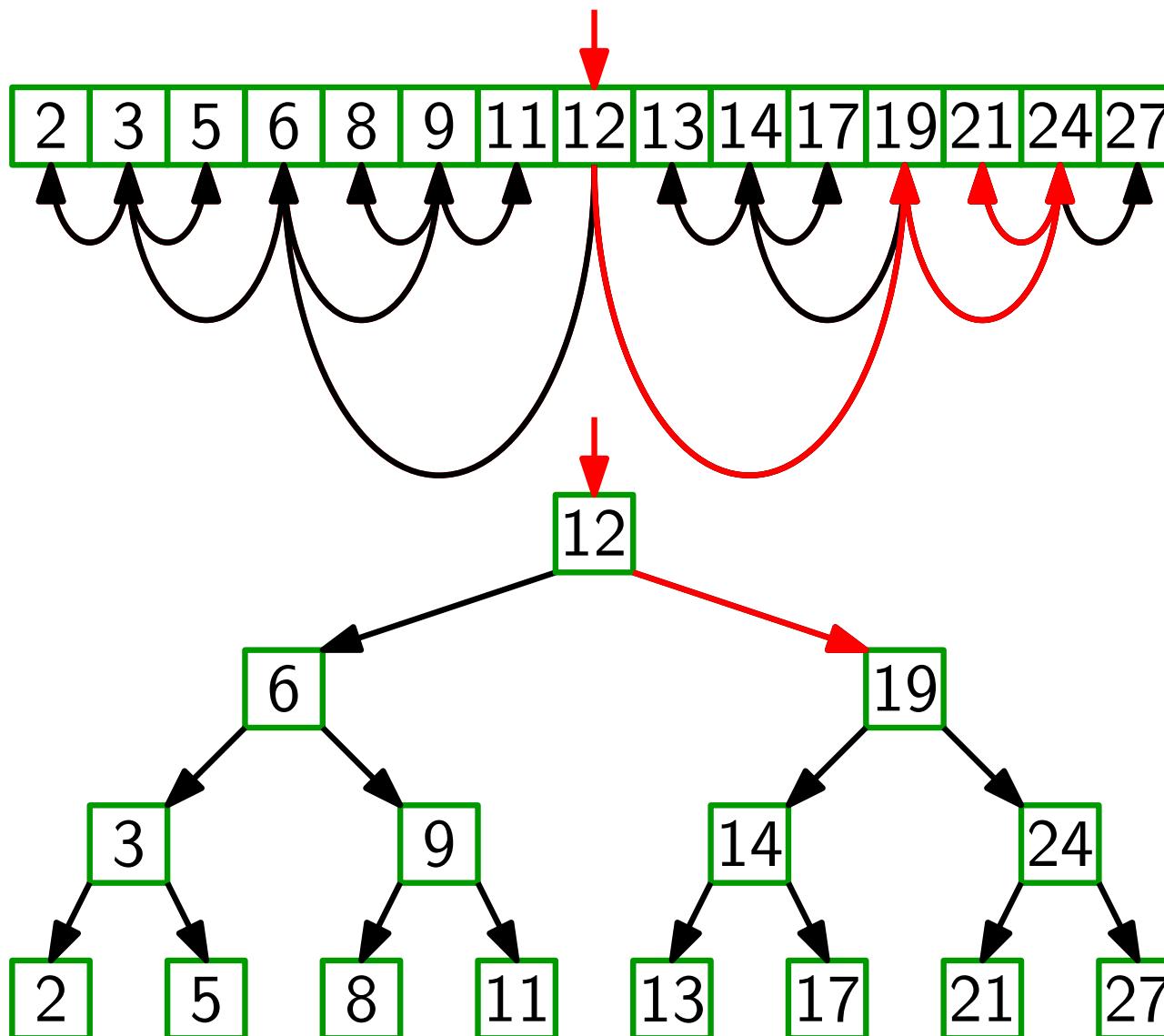
Suche im sortierten Feld



Suche 21!

Binärer
Suchbaum

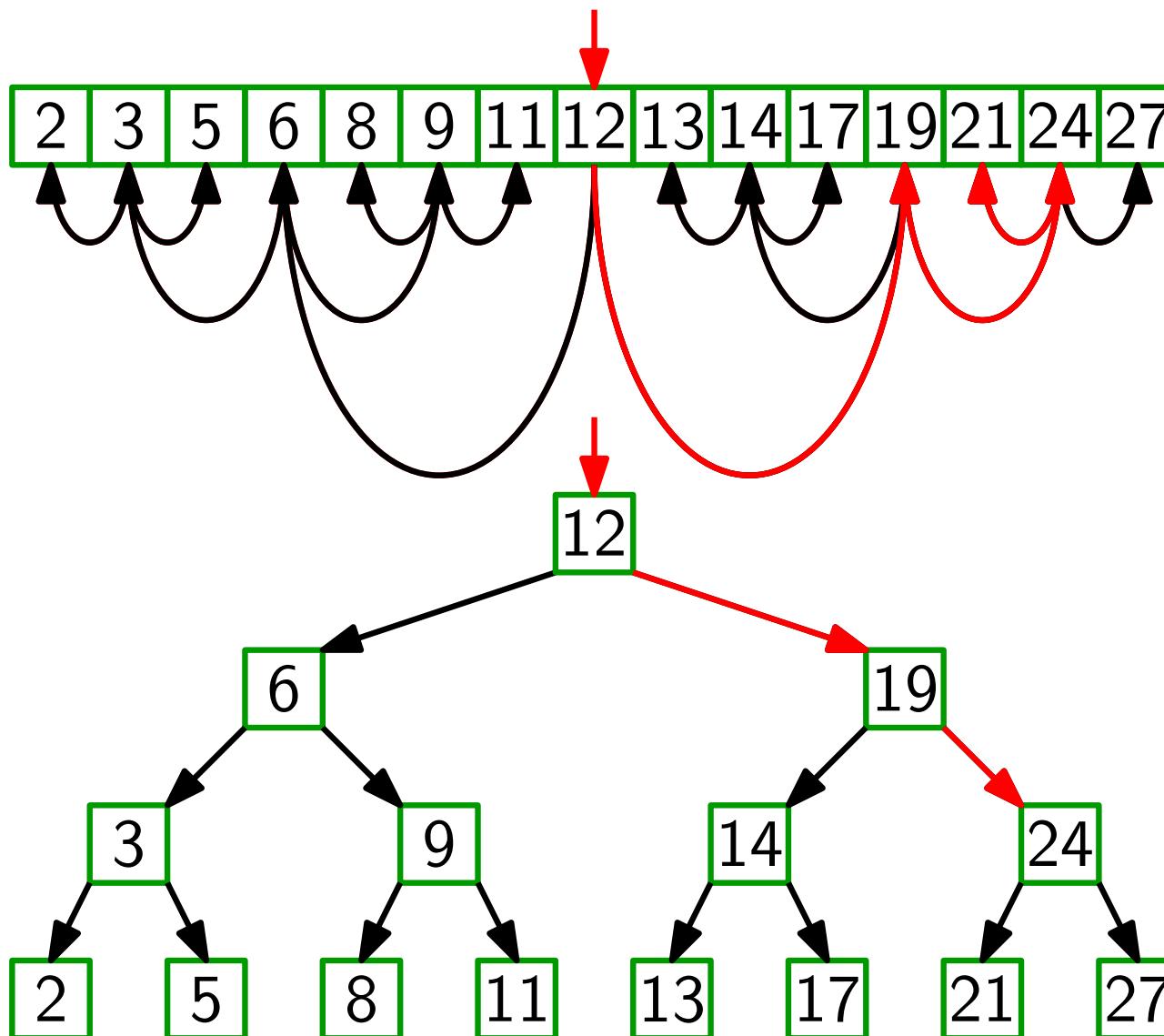
Suche im sortierten Feld



Suche 21!

Binärer
Suchbaum

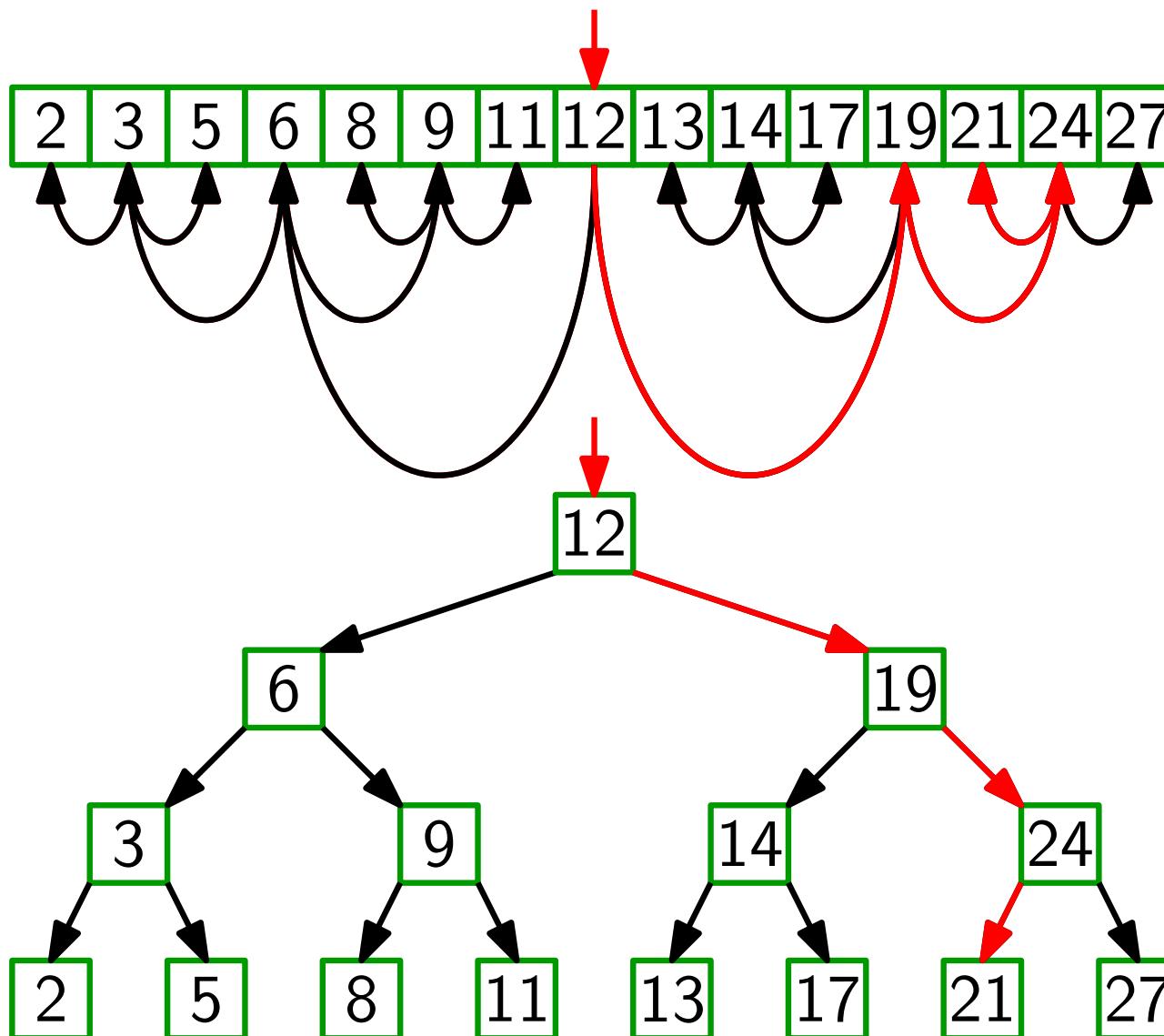
Suche im sortierten Feld



Suche 21!

Binärer
Suchbaum

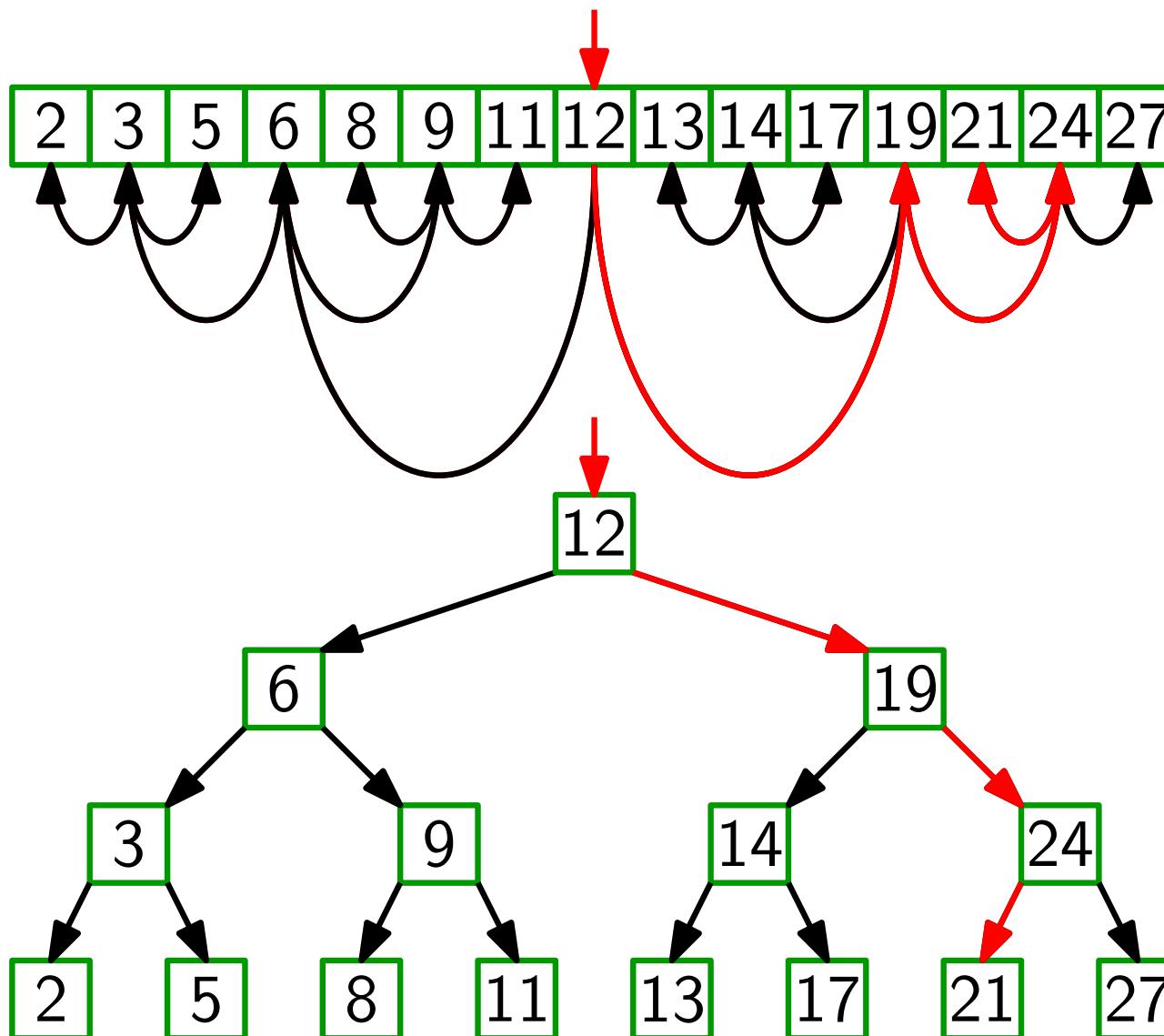
Suche im sortierten Feld



Suche 21!

Binärer
Suchbaum

Suche im sortierten Feld

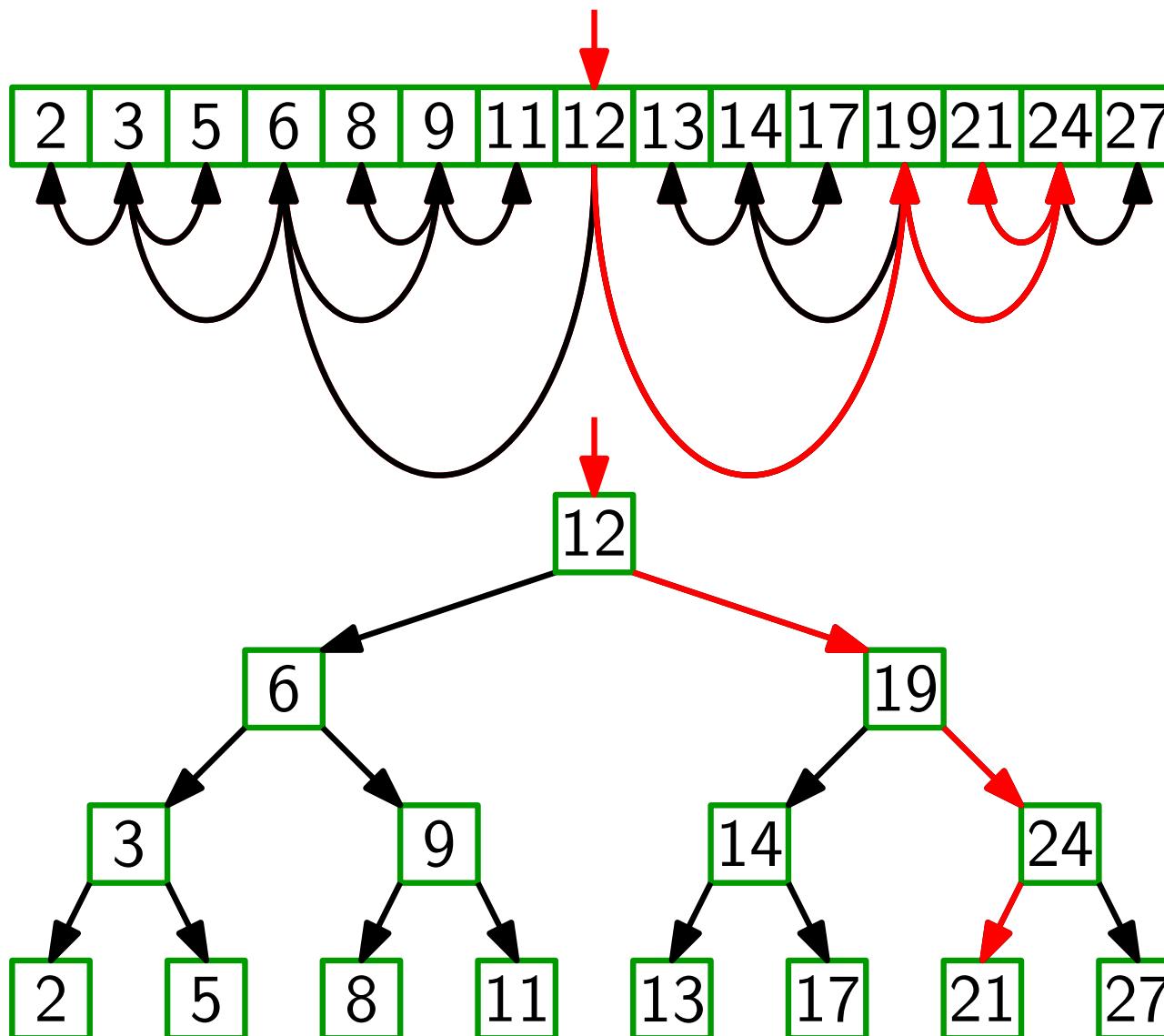


Suche 21!

Binärer
Suchbaum

Binärer-Suchbaum-Eigenschaft:

Suche im sortierten Feld



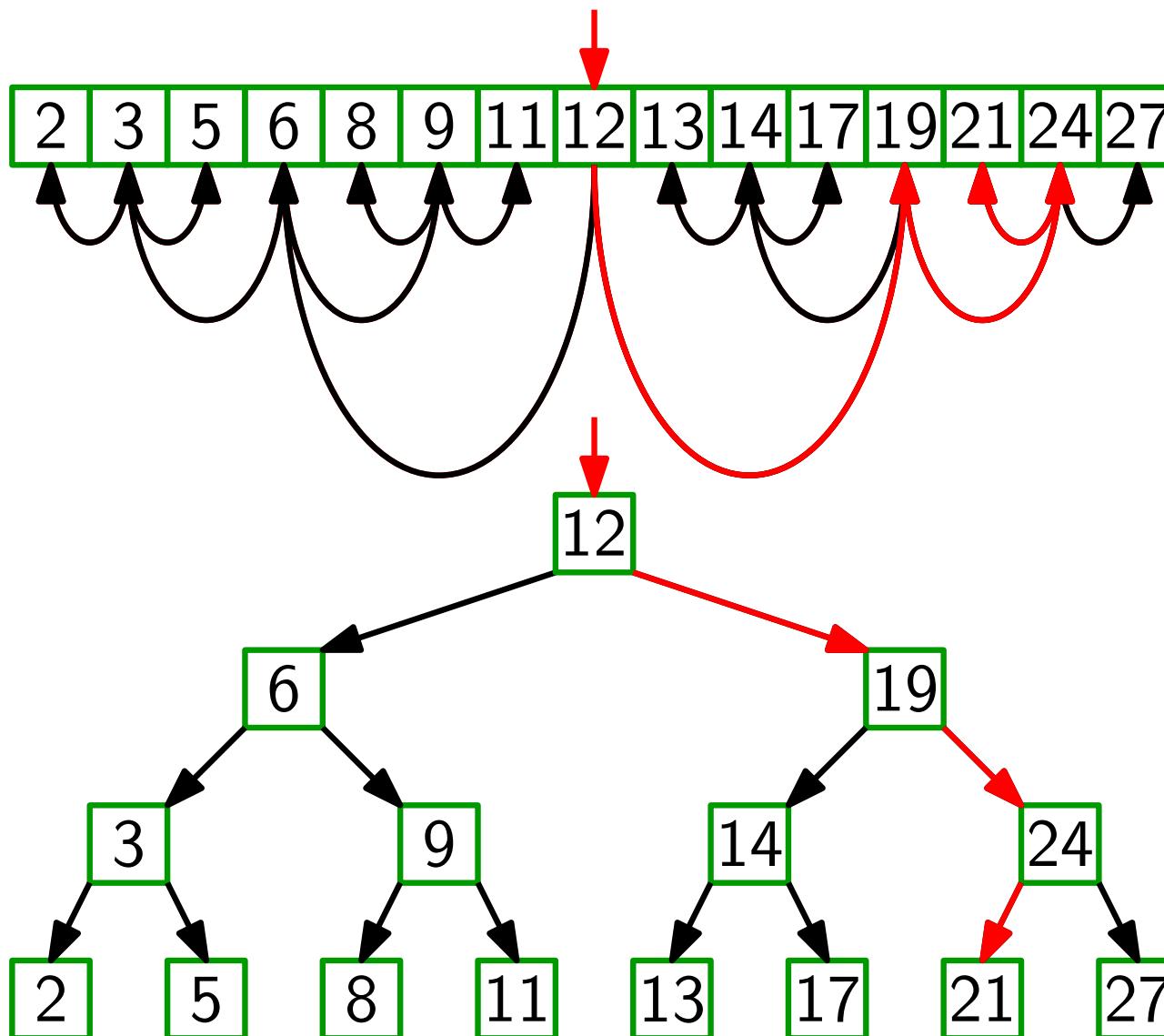
Suche 21!

Binärer
Suchbaum

Binärer-Suchbaum-Eigenschaft:

Für jeden Knoten v gilt:

Suche im sortierten Feld



Suche 21!

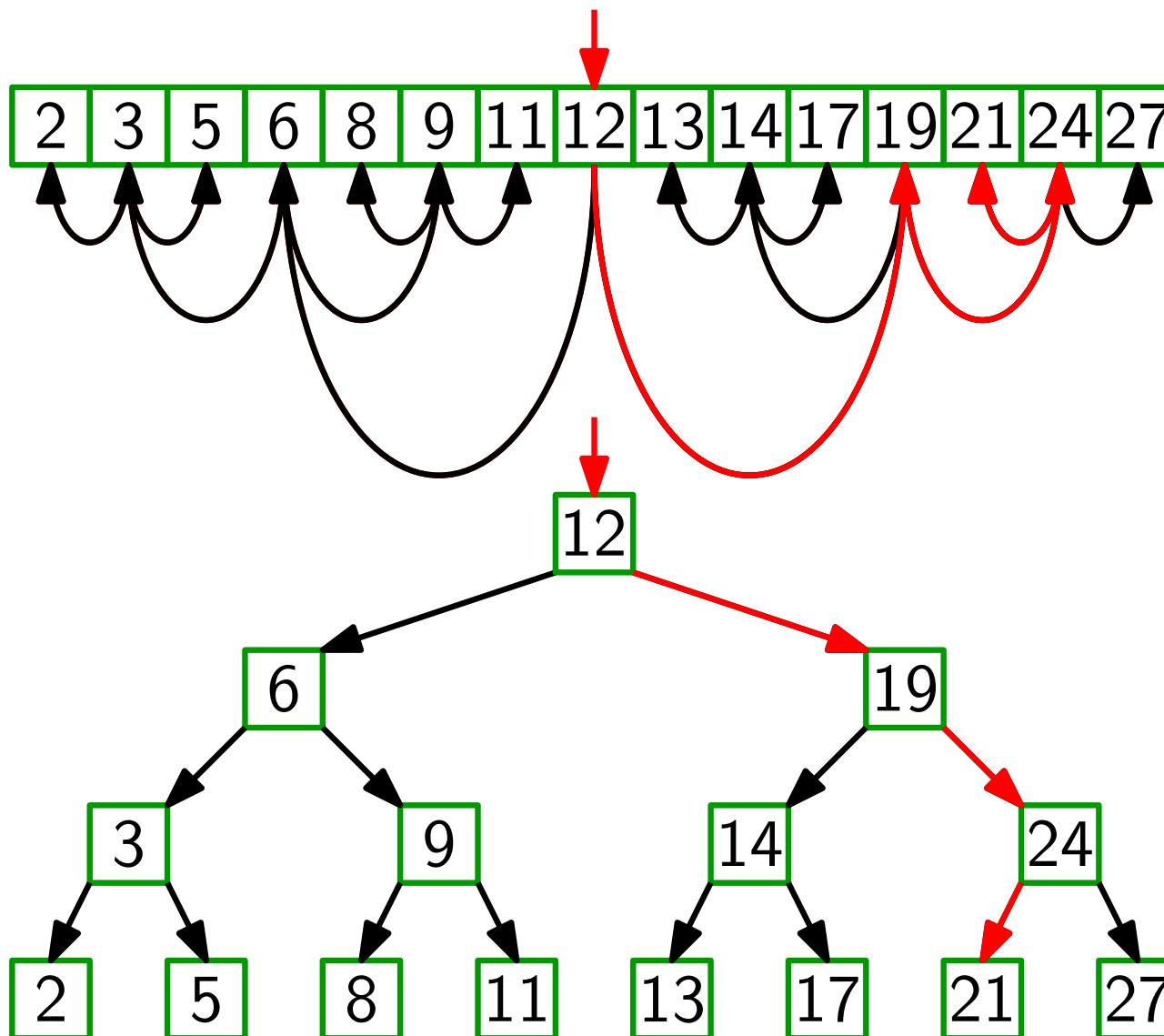
Binärer
Suchbaum

Binärer-Suchbaum-Eigenschaft:

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$

Für jeden Knoten v gilt:

Suche im sortierten Feld



Suche 21!

Binärer
Suchbaum

Binärer-Suchbaum-Eigenschaft:

alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
rechten

Für jeden Knoten v gilt:

\geq

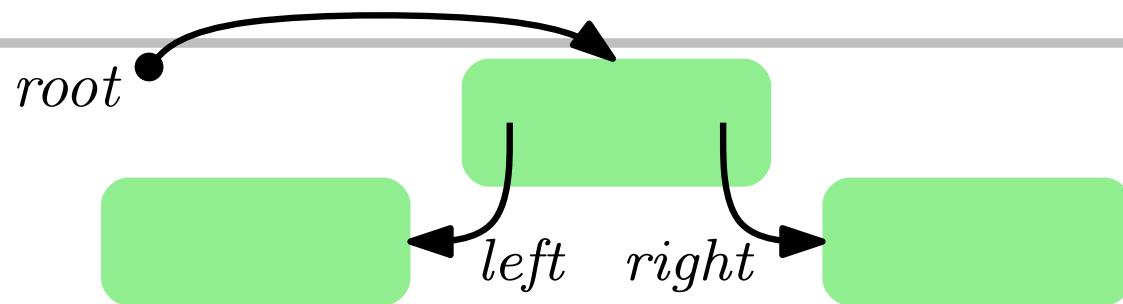
Bin. Suchbaum

Abs. Datentyp

BinSearchTree()

Implementierung

Bin. Suchbaum



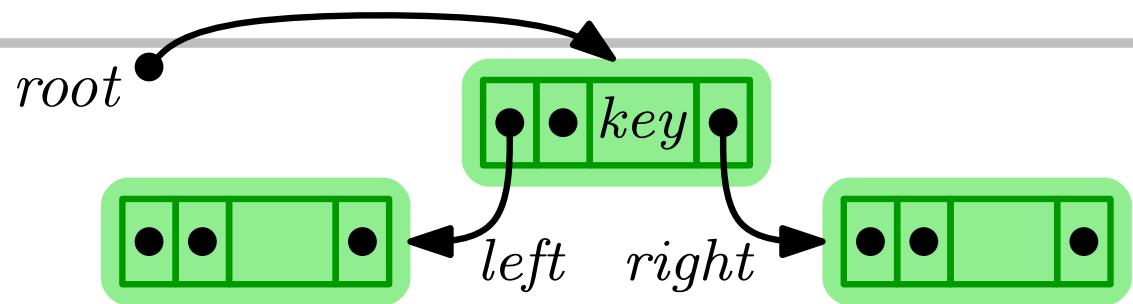
Abs. Datentyp

BinSearchTree()

Implementierung

Bin. Suchbaum

5



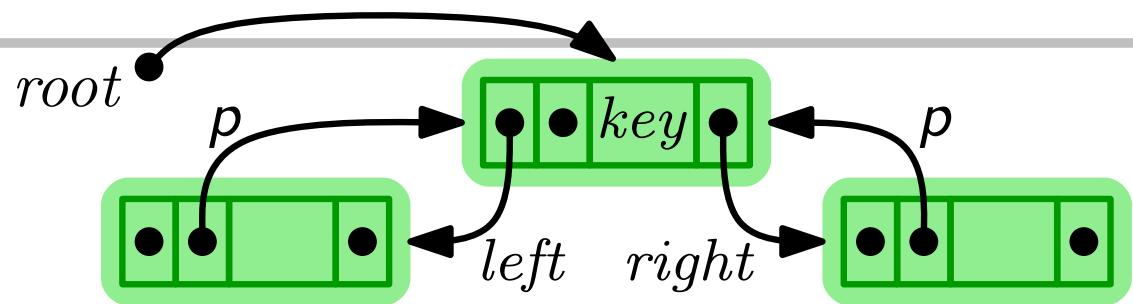
Abs. Datentyp

BinSearchTree()

Implementierung

Bin. Suchbaum

5



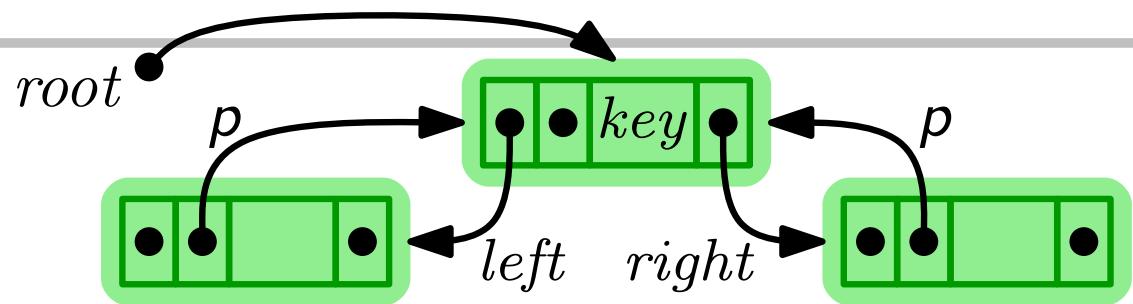
Abs. Datentyp

BinSearchTree()

Implementierung

Bin. Suchbaum

5



Abs. Datentyp

BinSearchTree()

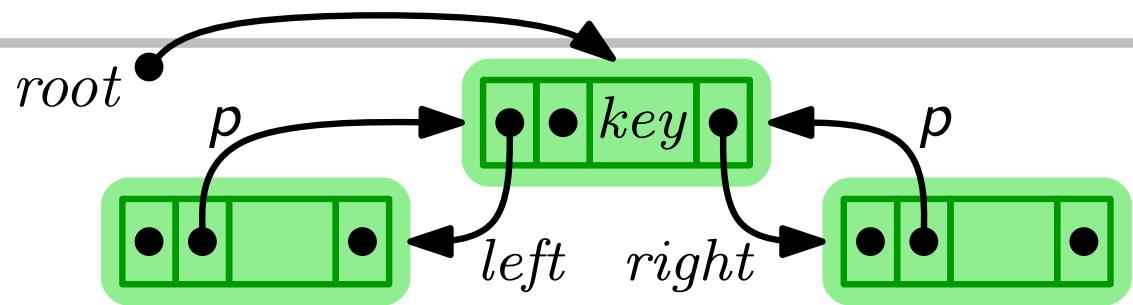
Implementierung

Node

key
key
Node left
Node right
Node p

Bin. Suchbaum

5



Abs. Datentyp

BinSearchTree()

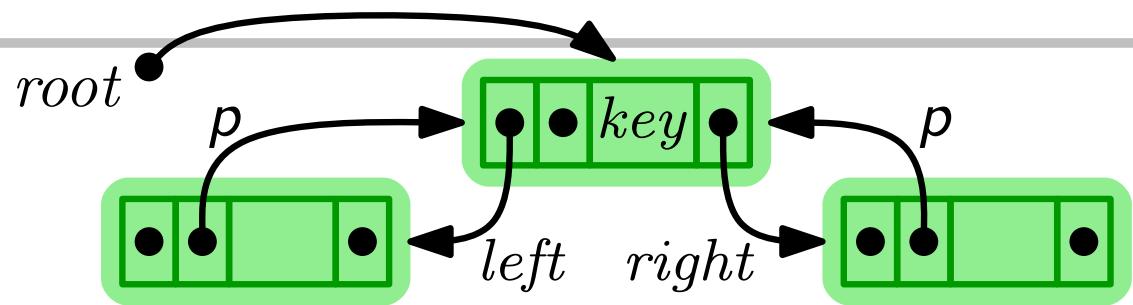
Implementierung

Node *root*

Node
key *key*
Node *left*
Node *right*
Node *p*

Bin. Suchbaum

5



Abs. Datentyp

BinSearchTree()

Implementierung

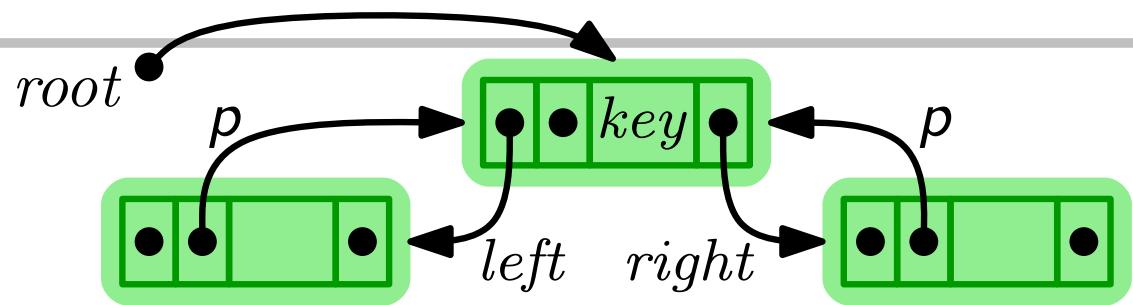
Node(key *k*, Node *par*)
key = *k*
p = *par*
right = *left* = *nil*

Node *root*

Node
key *key*
Node *left*
Node *right*
Node *p*

Bin. Suchbaum

5



Abs. Datentyp

BinSearchTree()

Implementierung

$root = nil$

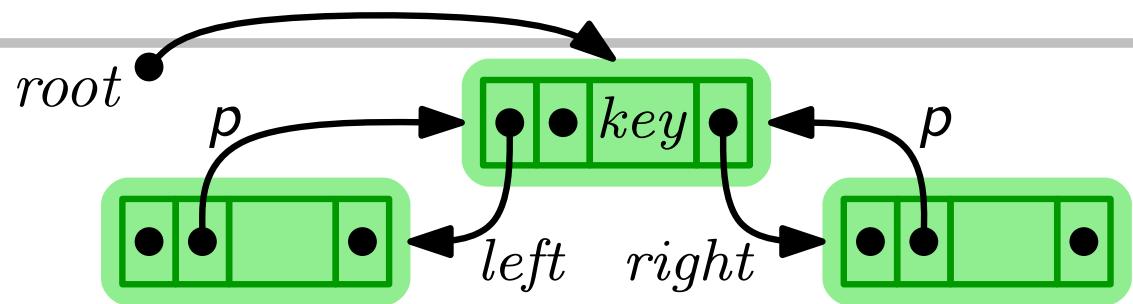
$Node(key\ k, Node\ par)$
 $key = k$
 $p = par$
 $right = left = nil$

$Node\ root$

$Node$
 $key\ key$
 $Node\ left$
 $Node\ right$
 $Node\ p$

Bin. Suchbaum

5



Abs. Datentyp

BinSearchTree()

Node Search(key k)

Node Insert(key k)

Delete(Node x)

Node Minimum()

Node Maximum()

Node Predecessor(Nd. x)

Node Successor(Node x)

Implementierung

root = nil

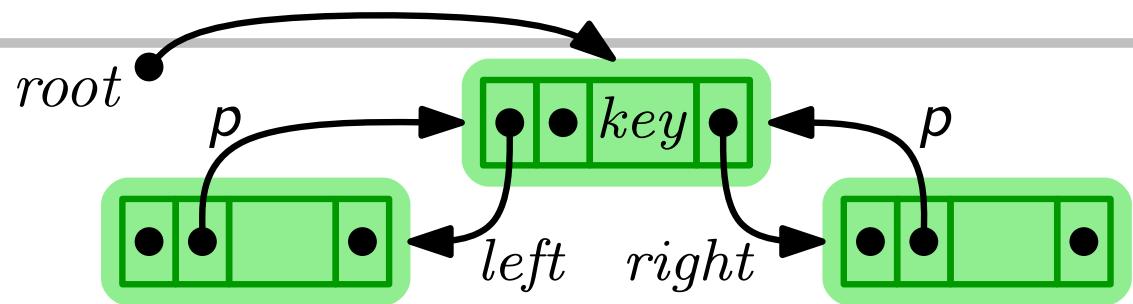
Node(key *k*, Node *par*)
key = *k*
p = *par*
right = *left* = *nil*

Node *root*

Node
key *key*
Node *left*
Node *right*
Node *p*

Bin. Suchbaum

5



Abs. Datentyp

BinSearchTree()

Node Search(key k)

Node Insert(key k)

Delete(Node x)

Node Minimum()

Node Maximum()

Node Predecessor(Nd. x)

Node Successor(Node x)

Implementierung

root = nil

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

Node *root*

Node
key *key*
Node *left*
Node *right*
Node *p*

TO *do*!

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!

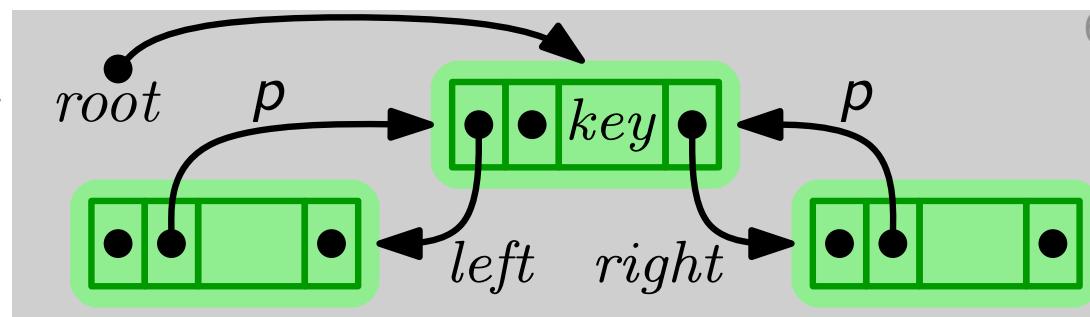
Lösung:

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:

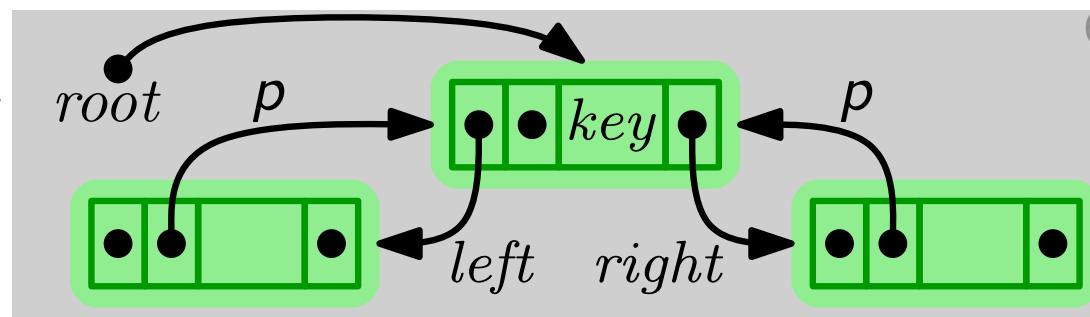
1. Durchlaufe rekursiv linken Teilbaum der Wurzel.
2. Gib den Schlüssel der Wurzel aus.
3. Durchlaufe rekursiv rechten Teilbaum der Wurzel.

Code:

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

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

Korrektheit

Code:

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

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

```
    InorderTreeWalk( $x.left$ )
```

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

```
    InorderTreeWalk( $x.right$ )
```

Korrektheit

zu zeigen:

Code:

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

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

```
    InorderTreeWalk( $x.left$ )
```

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

```
    InorderTreeWalk( $x.right$ )
```

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Code:

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

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

```
    InorderTreeWalk( $x.left$ )
```

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

```
    InorderTreeWalk( $x.right$ )
```

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. 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 Rf. ausgegeben.
Induktion über die Baumhöhe h .

$h = -1$:

Code:

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

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

```
    InorderTreeWalk( $x.left$ )
```

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

```
    InorderTreeWalk( $x.right$ )
```

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

Code:

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

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

```
    InorderTreeWalk( $x.left$ )
```

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

```
    InorderTreeWalk( $x.right$ )
```

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. 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 Rf. 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 Rf. ausgegeben.

Induktion über die Baumhöhe h .

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



$h \geq 0$: Ind.-Hyp. 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 Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. 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 Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.
 T_{links} und T_{rechts} haben Höhe $< h$.

Code:

InorderTreeWalk(Node $x = root$)

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

 InorderTreeWalk($x.left$)

 gib $x.key$ aus

 InorderTreeWalk($x.right$)

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. *[rekursive Def. der Höhe!]*

Code:

InorderTreeWalk(Node $x = root$)

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

InorderTreeWalk($x.left$)

gib $x.key$ aus

InorderTreeWalk($x.right$)

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. *[rekursive Def. der Höhe!]*

Also werden *ihre* Schlüssel sortiert ausgegeben.

Code:

InorderTreeWalk(Node $x = root$)

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

 InorderTreeWalk($x.left$)

 gib $x.key$ aus

 InorderTreeWalk($x.right$)

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. *[rekursive Def. der Höhe!]*

Also werden *ihre* Schlüssel sortiert ausgegeben.

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

Code:

InorderTreeWalk(Node $x = root$)

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

 InorderTreeWalk($x.left$)

 gib $x.key$ aus

 InorderTreeWalk($x.right$)

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. *[rekursive Def. der Höhe!]*

Also werden *ihre* Schlüssel sortiert ausgegeben.

Binärer-Suchbaum-Eigenschaft \Rightarrow

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

Code:

InorderTreeWalk(Node $x = root$)

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

 InorderTreeWalk($x.left$)

 gib $x.key$ aus

 InorderTreeWalk($x.right$)

Korrektheit

zu zeigen: Schlüssel werden in sortierter Rf. ausgegeben.

Induktion über die Baumhöhe h .

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

$h \geq 0$: Ind.-Hyp. sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} li. & re. Teilbaum der Wurzel.

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

Also werden *ihre* Schlüssel sortiert ausgegeben.

Binärer-Suchbaum-Eigenschaft ⇒

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

Code:

InorderTreeWalk(Node $x = root$)

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

InorderTreeWalk($x.left$)

gib $x.key$ aus

InorderTreeWalk($x.right$)

Laufzeit

$$T(n) =$$

Code:

```
InorderTreeWalk(Node x = 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

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


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  $x \neq nil$  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

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

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

oder:

Code:

InorderTreeWalk(Node $x = root$)

```

if  $x \neq nil$  then
    InorderTreeWalk( $x.left$ )
    gib  $x.key$  aus
    InorderTreeWalk( $x.right$ )
  
```

Laufzeit

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

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

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

Code:

InorderTreeWalk(Node $x = root$)

```

if  $x \neq nil$  then
    InorderTreeWalk( $x.left$ )
    gib  $x.key$  aus
    InorderTreeWalk( $x.right$ )
  
```

Laufzeit

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

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

oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. von Schritten aus.
 Für Bäume gilt: #Kanten = #Knoten - 1 = $n - 1$

Code:

InorderTreeWalk(Node $x = root$)

```

if  $x \neq nil$  then
  InorderTreeWalk( $x.left$ )
  gib  $x.key$  aus
  InorderTreeWalk( $x.right$ )
```

Laufzeit

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

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

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

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

Übung: zeig's mit Induktion!

Code:

InorderTreeWalk(Node $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    InorderTreeWalk( $x.\text{right}$ )
  
```

Laufzeit

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

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

oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. von Schritten aus.
 Für Bäume gilt: #Kanten = #Knoten - 1 = $n - 1$

Code:

InorderTreeWalk(Node $x = root$)

```

if  $x \neq nil$  then
  InorderTreeWalk( $x.left$ )
  gib  $x.key$  aus
  InorderTreeWalk( $x.right$ )
  
```

Laufzeit

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

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

oder: Für jeden Knoten und jede Kante des Baums führt InorderTreeWalk eine konstante Anz. 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 $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    InorderTreeWalk( $x.\text{right}$ )
```

Laufzeit

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

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

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

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

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

Code:

InorderTreeWalk(Node $x = \text{root}$)

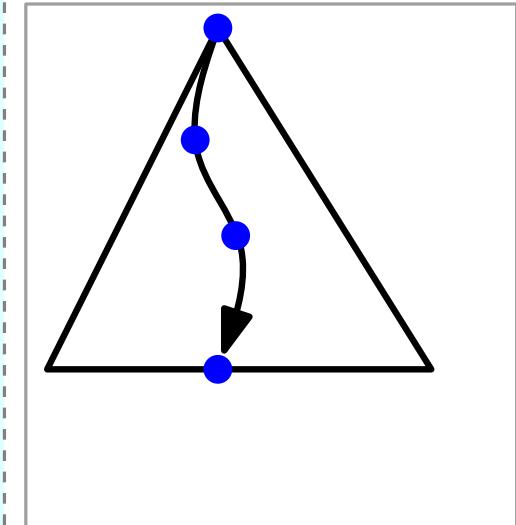
```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    InorderTreeWalk( $x.\text{right}$ )
```

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

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



Code:

InorderTreeWalk(Node $x = root$)

```
if  $x \neq nil$  then
    InorderTreeWalk( $x.left$ )
    gib  $x.key$  aus
    InorderTreeWalk( $x.right$ )
```

Suche

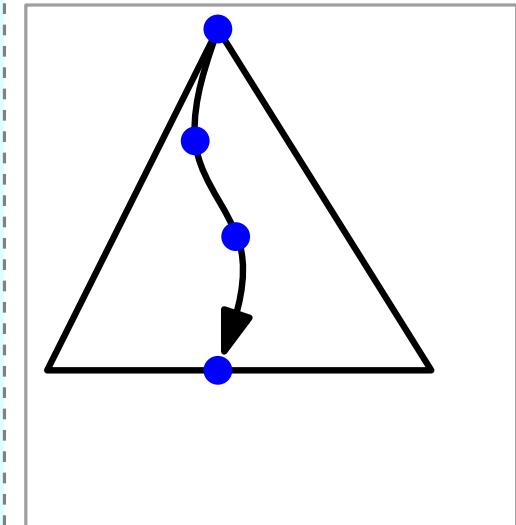
Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

Node Search(key k , Node $x = \text{root}$)

```

if  $x == \text{nil}$  or  $x.\text{key} == k$  then
    return  $x$ 
if  $k < x.\text{key}$  then
    return Search( $k, x.\text{left}$ )
else return Search( $k, x.\text{right}$ )

```



Code:

InorderTreeWalk(Node $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    InorderTreeWalk( $x.\text{right}$ )

```

Suche

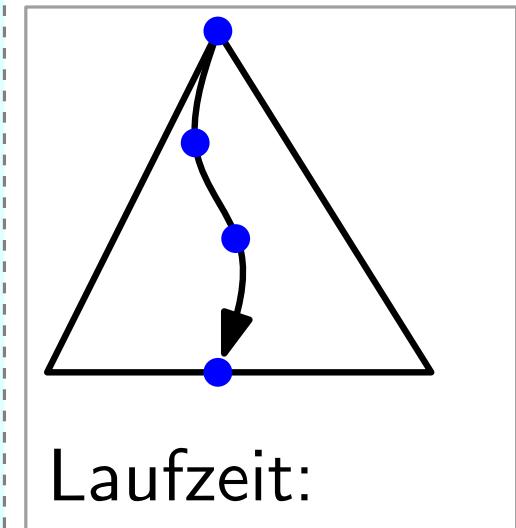
Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

Node Search(key k , Node $x = \text{root}$)

```

if  $x == \text{nil}$  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 $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.left$ )
    gib  $x.key$  aus
    InorderTreeWalk( $x.right$ )

```

Suche

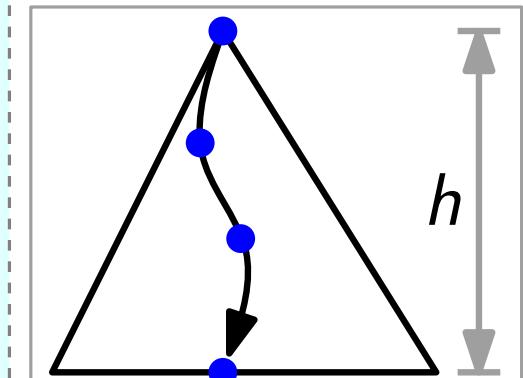
Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

Node Search(key k , Node $x = \text{root}$)

```

if  $x == \text{nil}$  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 $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.left$ )
    gib  $x.key$  aus
    InorderTreeWalk( $x.right$ )

```

Suche

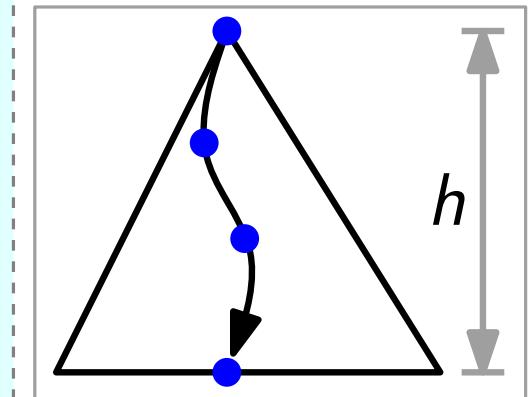
Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

Node Search(key k , Node $x = \text{root}$)

```

if  $x == \text{nil}$  or  $x.\text{key} == k$  then
    return  $x$ 
if  $k < x.\text{key}$  then
    return Search( $k, x.\text{left}$ )
else return Search( $k, x.\text{right}$ )

```



Laufzeit: $O(h)$

Code:

InorderTreeWalk(Node $x = \text{root}$)

```

if  $x \neq \text{nil}$  then
    InorderTreeWalk( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    InorderTreeWalk( $x.\text{right}$ )

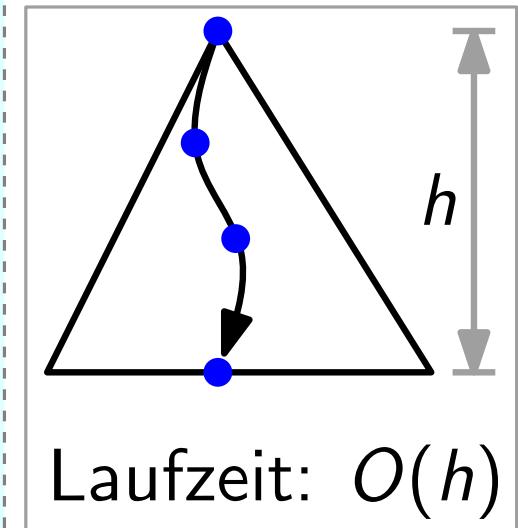
```

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

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)
```



Code:

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

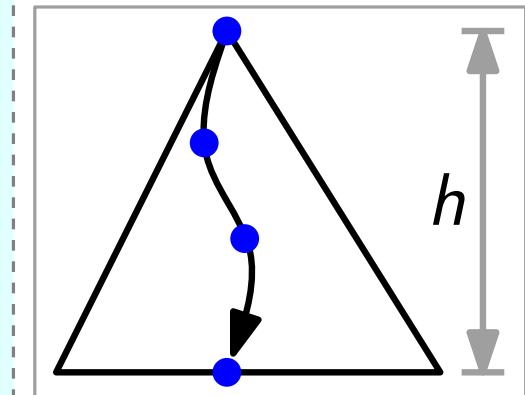
Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

rekursiv

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

iterativ



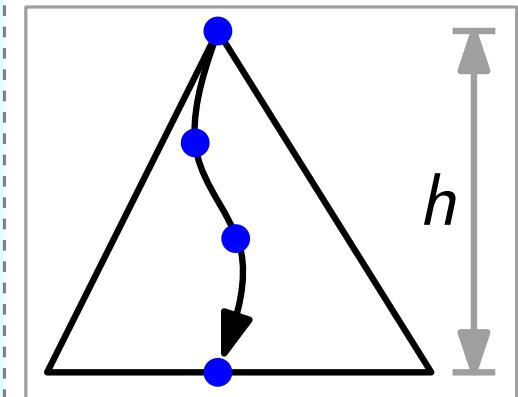
Laufzeit: $O(h)$

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

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: $O(h)$

iterativ

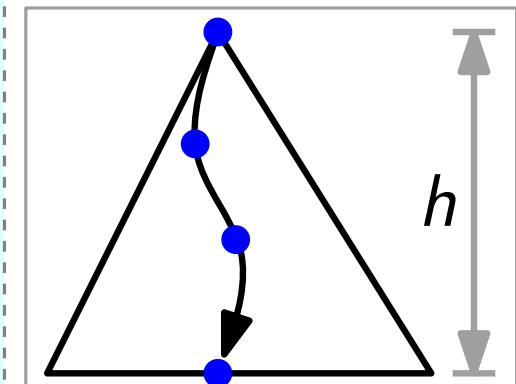
```
while x ≠ nil and x.key ≠ k do
  if k < x.key then
    x = x.left
  else
    x = x.right
return x
```

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

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: $O(h)$

iterativ

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

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

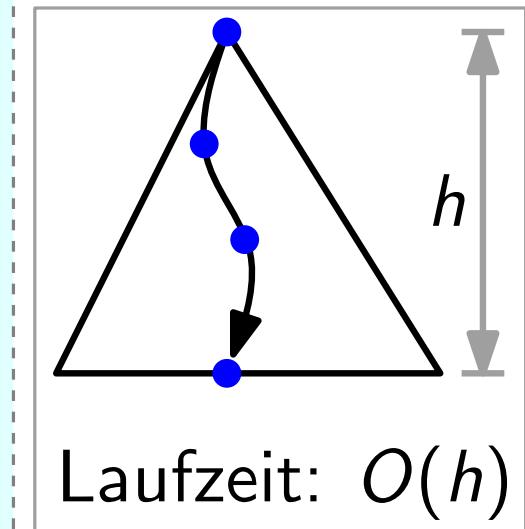
rekursiv

Node Search(key k , Node $x = \text{root}$)

```

if  $x == \text{nil}$  or  $x.key == k$  then
     $\quad \text{return } x$ 
if  $k < x.key$  then
     $\quad \text{return Search}(k, x.left)$ 
else  $\text{return Search}(k, x.right)$ 

```



iterativ

```

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

```

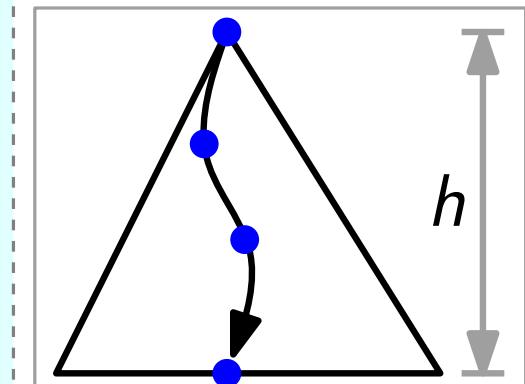
Laufzeit:

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

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: $O(h)$

iterativ

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

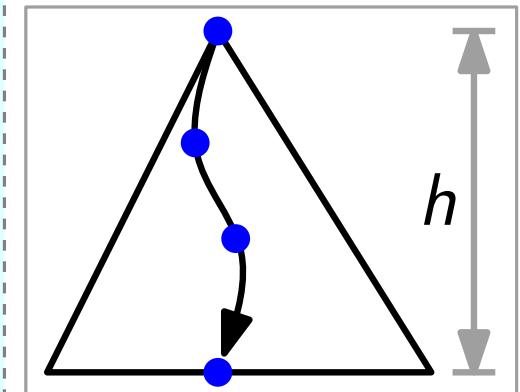
Laufzeit: $O(h)$

Suche

Aufgabe: Schreiben Sie Pseudocode für die rekursive Methode

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: $O(h)$

iterativ

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

Laufzeit: $O(h)$

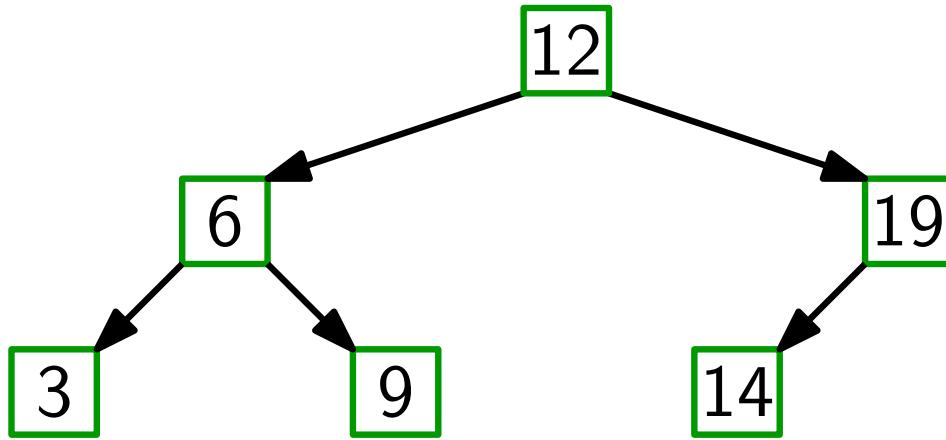
Trotzdem schneller,
da keine Verwaltung
der rekursiven
Methodenaufrufe.

Minimum & Maximum

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

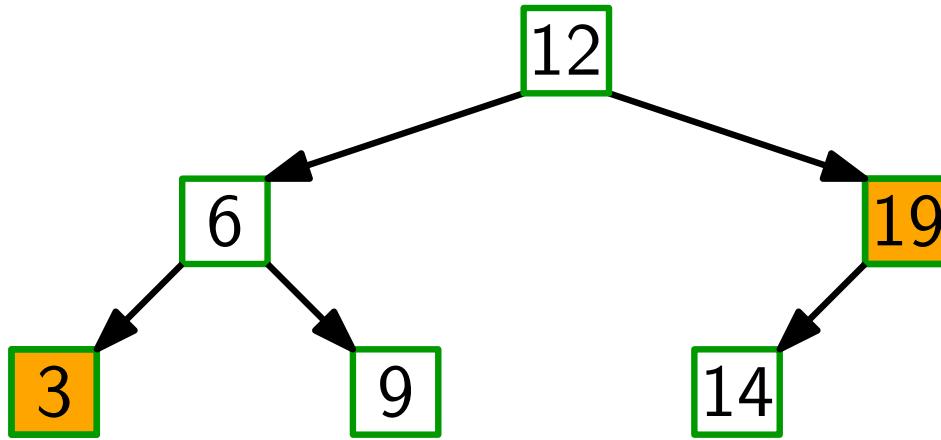
Minimum & Maximum

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



Minimum & Maximum

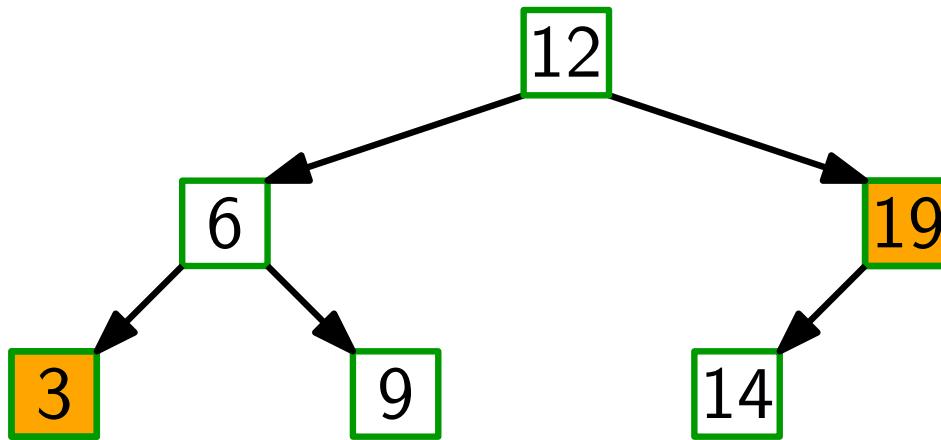
Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



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?



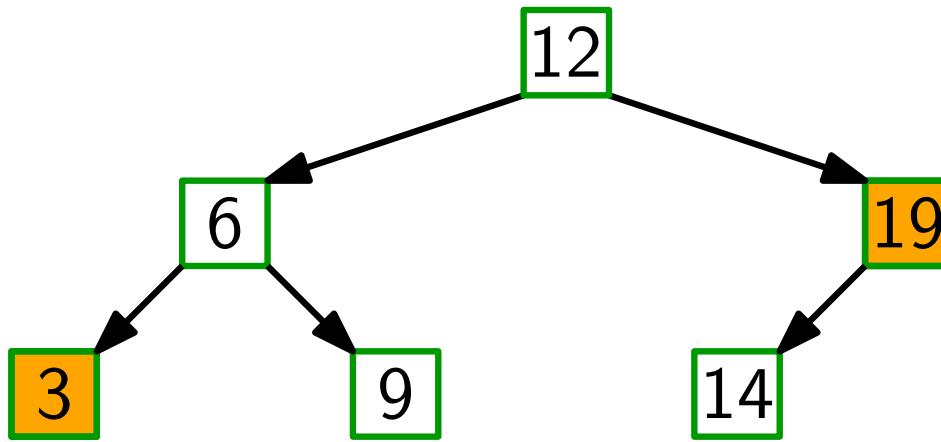
Antwort: Min steht ganz links, Max ganz rechts!

Aufgabe: Schreiben Sie für binäre Suchbäume die Methode

Node Minimum(Node $x = root$) — iterativ!

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!

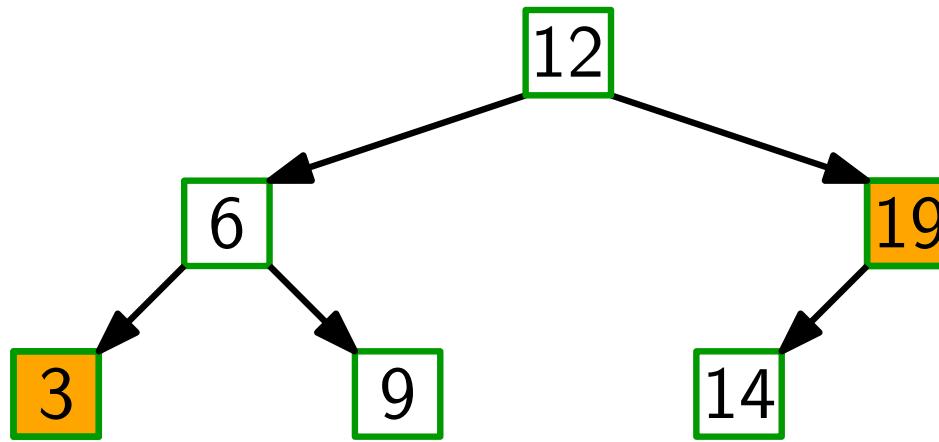
Aufgabe: Schreiben Sie für binäre Suchbäume die Methode

Node Minimum(Node $x = root$) — *iterativ!*

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

Minimum & Maximum

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



Antwort: Min steht ganz links, Max ganz rechts!

Aufgabe: Schreiben Sie für binäre Suchbäume die Methode

Node Minimum(Node $x = root$) — iterativ!

if $x == nil$ **then return** nil

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

 └ $x = x.left$

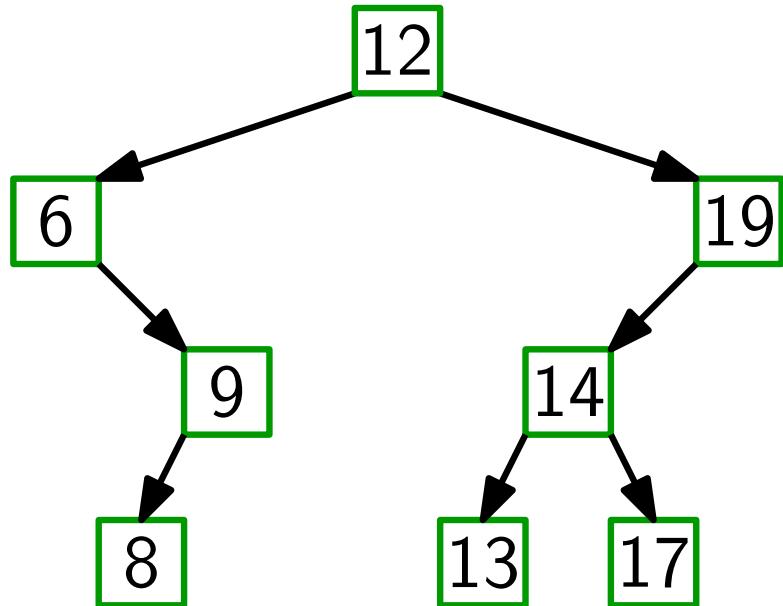
return x

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Nachfolger (und Vorgänger)

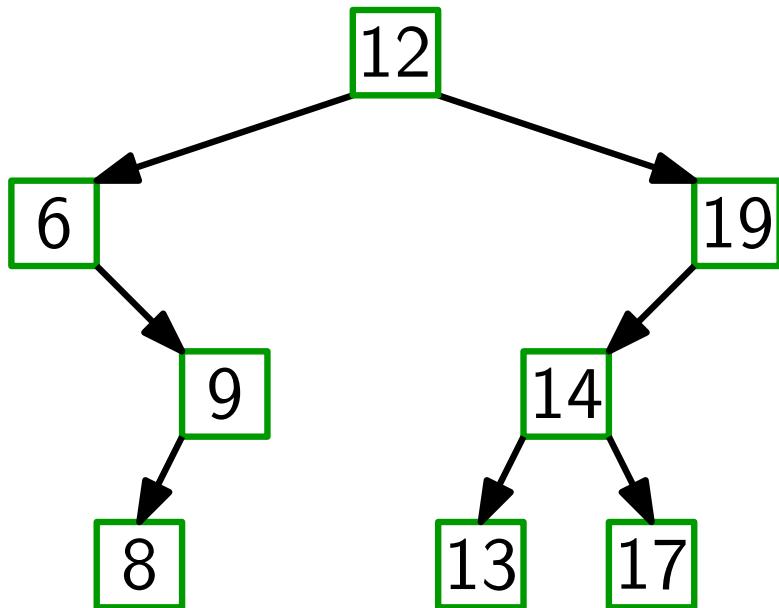
Vereinfachende Annahme: alle Schlüssel sind verschieden.



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

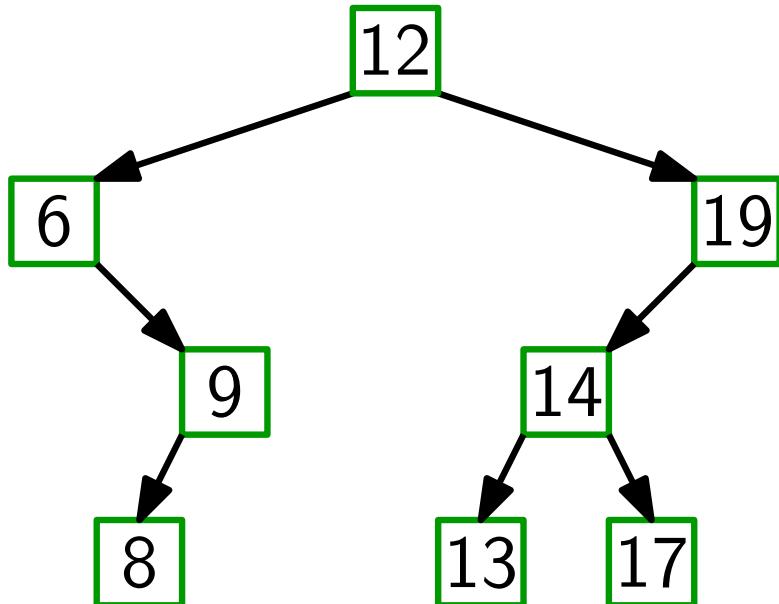
Erinnerung: Nachfolger(x) =



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

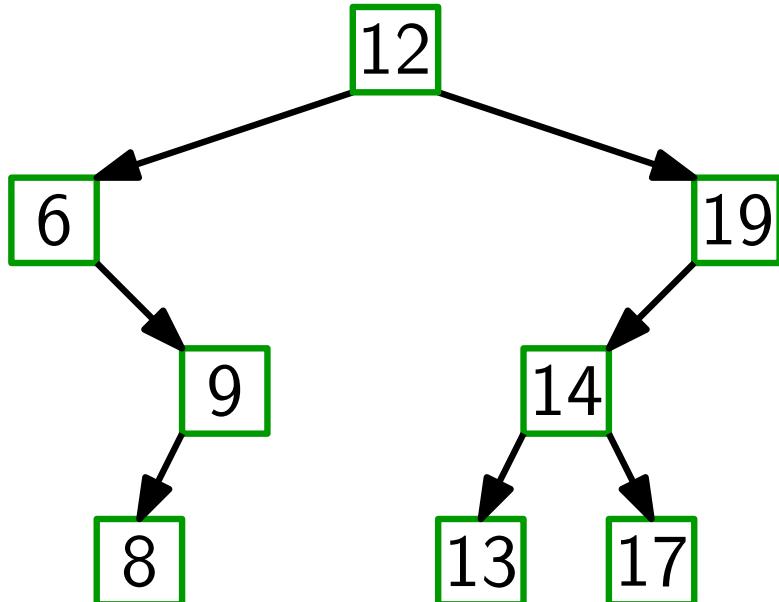
Erinnerung: $\text{Nachfolger}(x) = \text{Knoten mit kleinstem Schlüssel}$
 $\text{unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

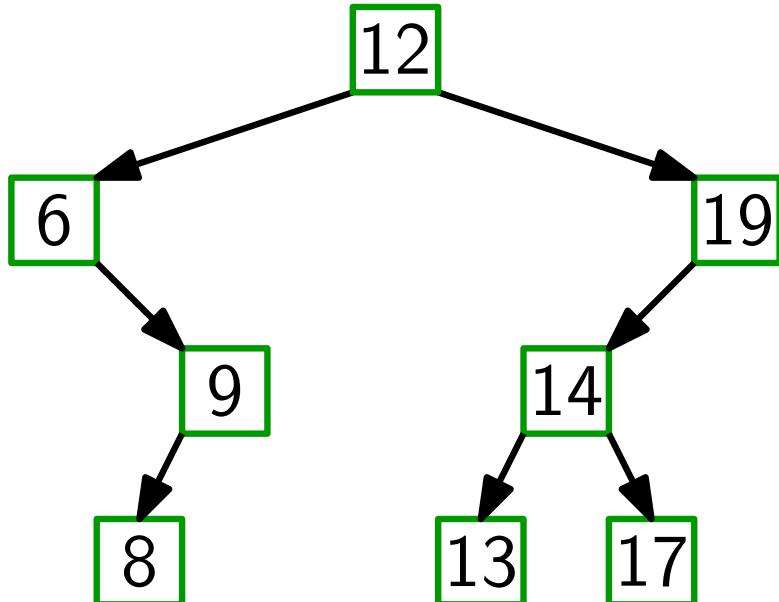
Erinnerung: $\text{Nachfolger}(x) = \text{Knoten mit kleinstem Schlüssel}$
 $\text{unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

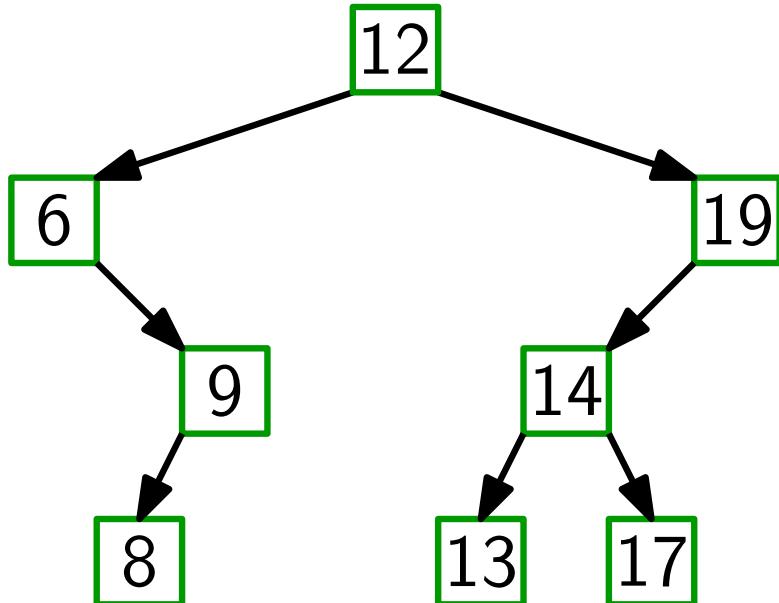
Erinnerung: $\text{Nachfolger}(x) = \text{Knoten mit kleinstem Schlüssel}$
 $\text{unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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

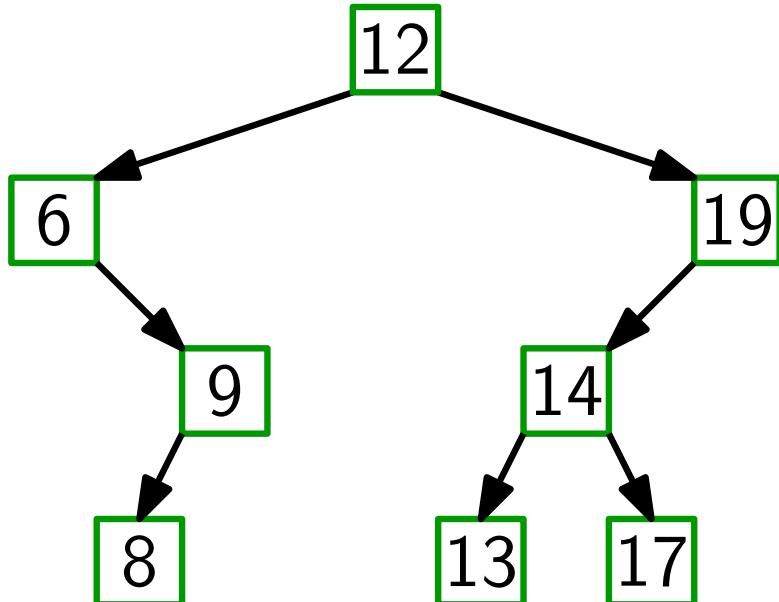


$\text{Nachfolger}(19) := \text{nil}$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



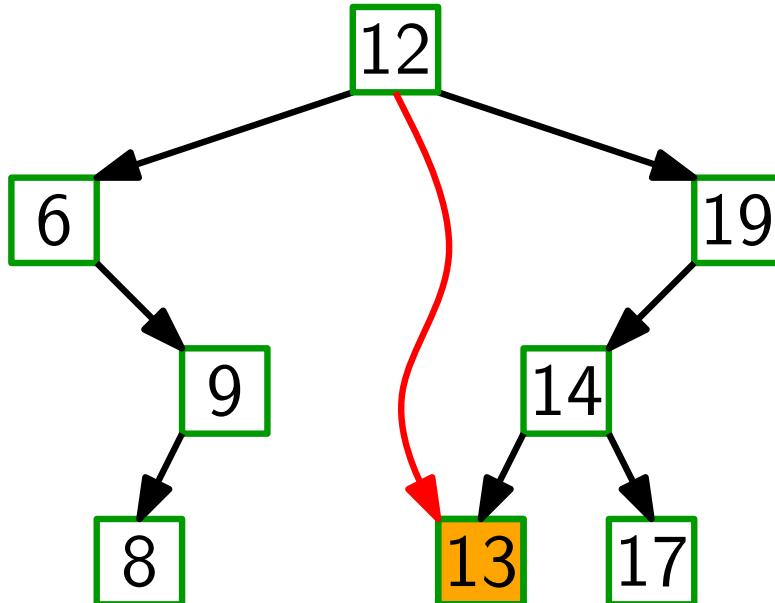
$\text{Nachfolger}(19) := \text{nil}$

$\text{Nachfolger}(12) = ?$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



$\text{Nachfolger}(19) := \text{nil}$

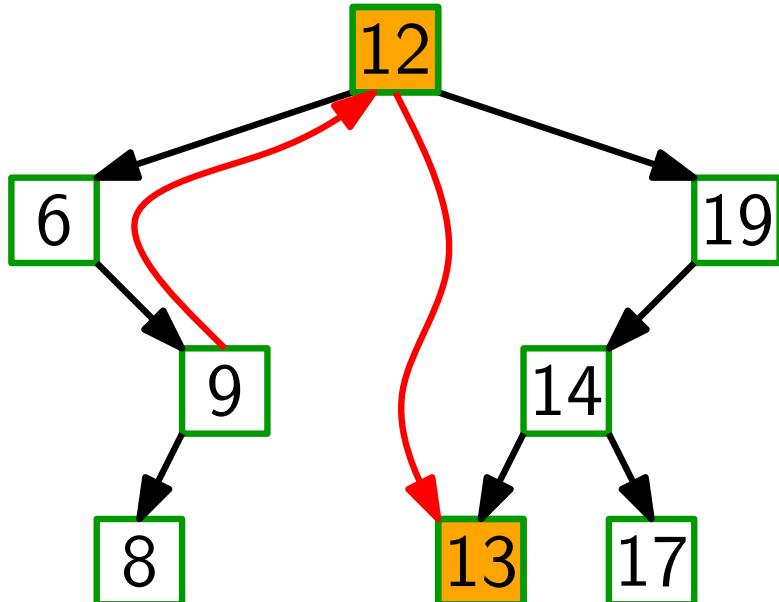
$\text{Nachfolger}(12) = ?$

$13 == \text{Minimum}(\text{,,}12.\text{right}\text{,,})$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



$\text{Nachfolger}(19) := \text{nil}$

$\text{Nachfolger}(12) = ?$

$\text{Nachfolger}(9) = ?$

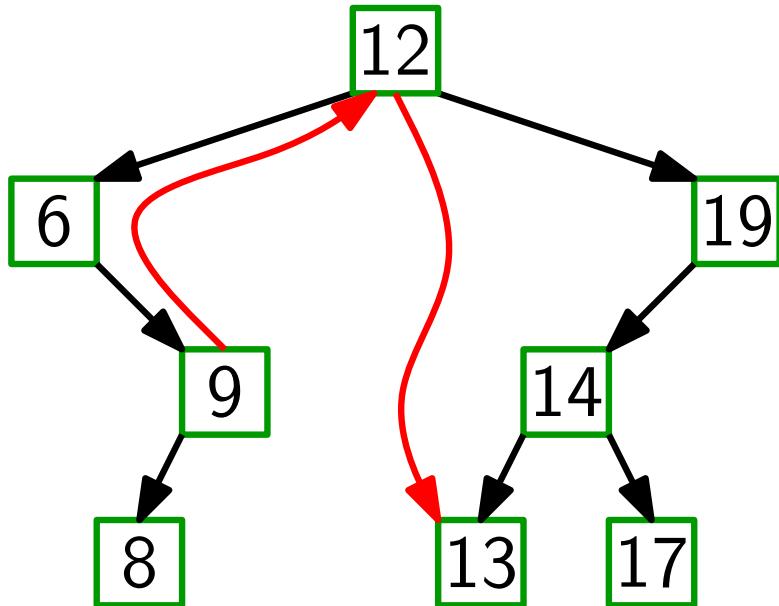
$13 == \text{Minimum}(\text{,,}12.\text{right}\text{,,})$

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

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

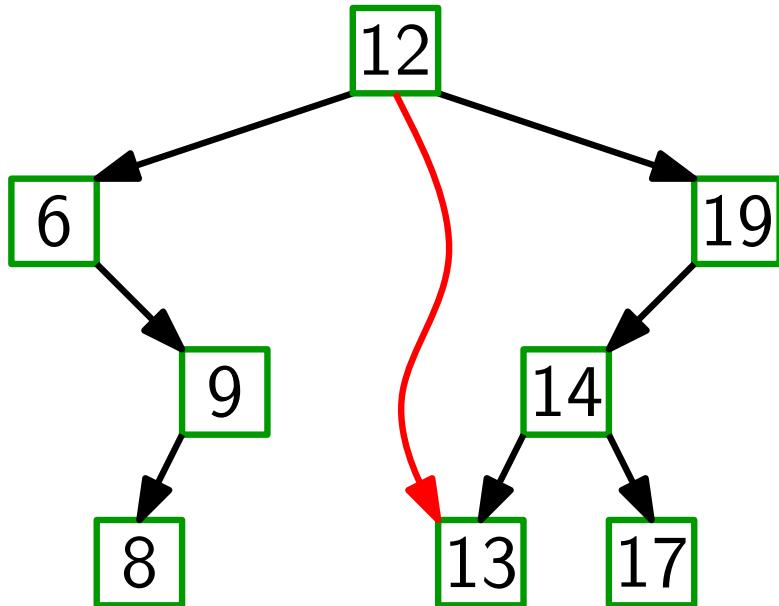
Erinnerung: $\text{Nachfolger}(x) = \text{Knoten mit kleinstem Schlüssel}$
 $\text{unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



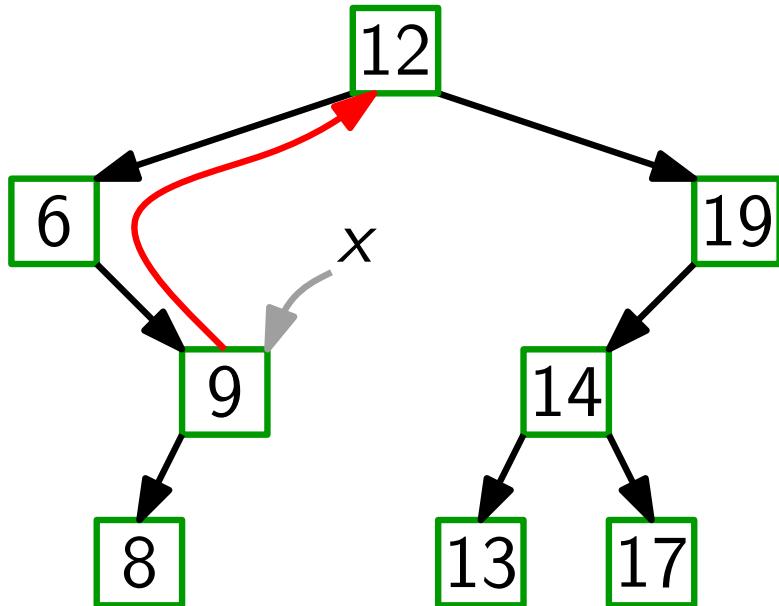
Node Successor(Node x)

```
if  $x.\text{right} \neq \text{nil}$  then
  return Minimum( $x.\text{right}$ )
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



Node Successor(Node x)

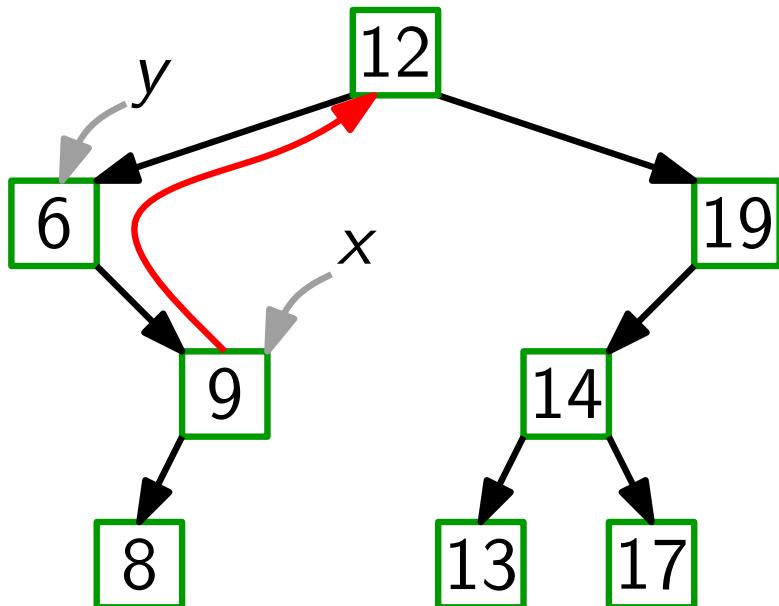
```

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

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: 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)

```

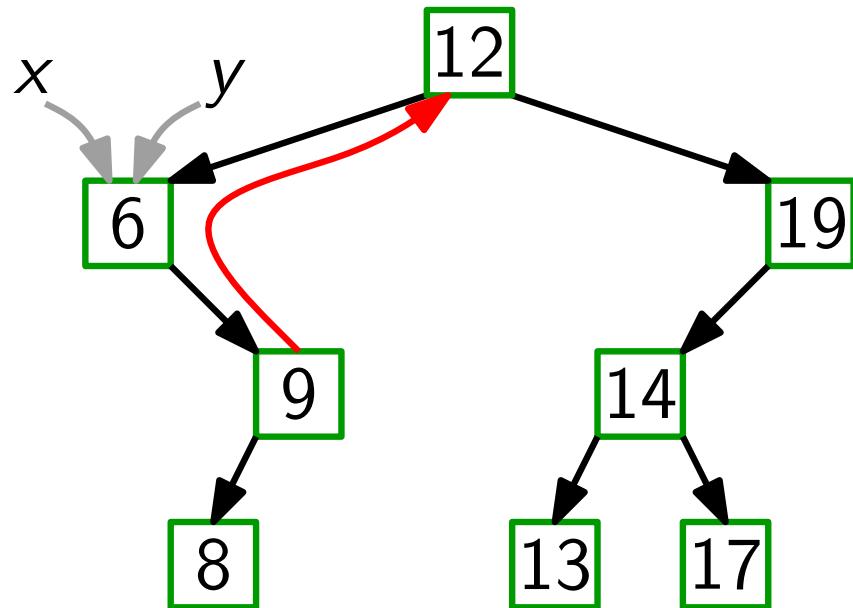
if  $x.right \neq nil$  then
   $\lfloor$  return Minimum( $x.right$ )
 $y = x.p$ 
while  $y \neq nil$  and  $x == y.right$  do
   $\lfloor$   $x = y$ 
   $\lfloor$   $y = y.p$ 
return  $y$ 

```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



Node Successor(Node x)

```

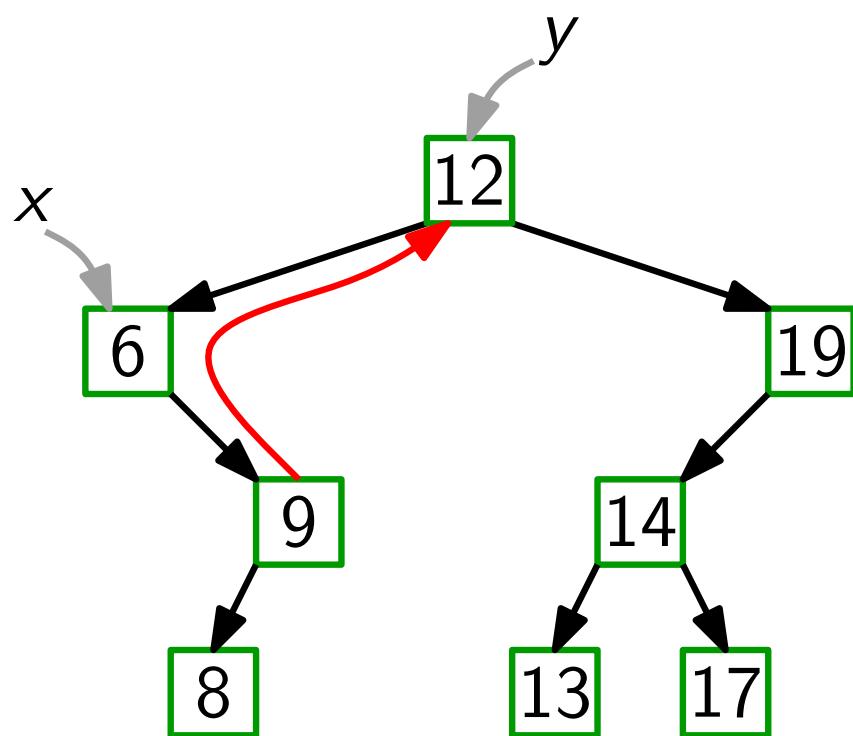
if  $x.\text{right} \neq \text{nil}$  then
    return Minimum( $x.\text{right}$ )
     $y = x.p$ 
while  $y \neq \text{nil}$  and  $x == y.\text{right}$  do
         $x = y$ 
         $y = y.p$ 
return  $y$ 

```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



Node Successor(Node x)

```

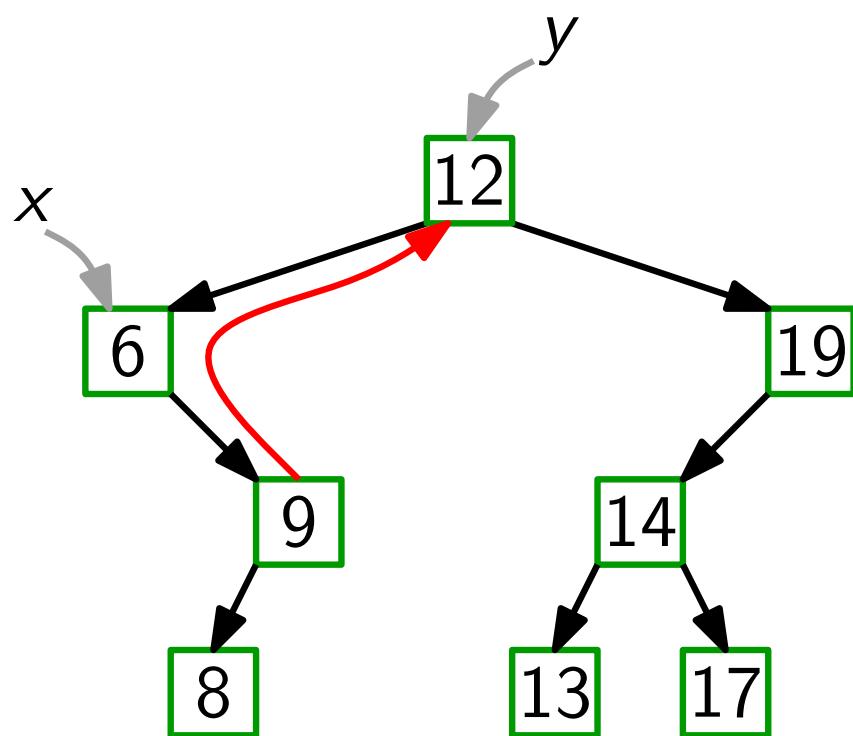
if  $x.\text{right} \neq \text{nil}$  then
  return Minimum( $x.\text{right}$ )
else
   $y = x.p$ 
  while  $y \neq \text{nil}$  and  $x == y.\text{right}$  do
     $x = y$ 
     $y = y.p$ 
  return  $y$ 

```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



Node Successor(Node x)

```

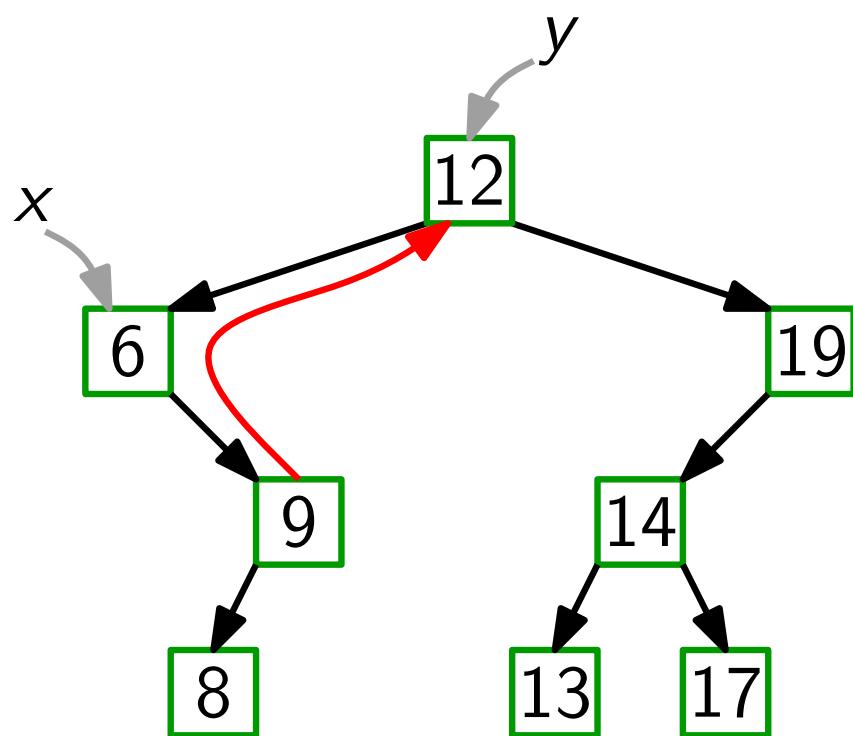
if  $x.\text{right} \neq \text{nil}$  then
  return Minimum( $x.\text{right}$ )
else
   $y = x.p$ 
  while  $y \neq \text{nil}$  and  $x == y.\text{right}$  do
     $x = y$ 
     $y = y.p$ 
  return  $y$ 

```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



Node Successor(Node x)

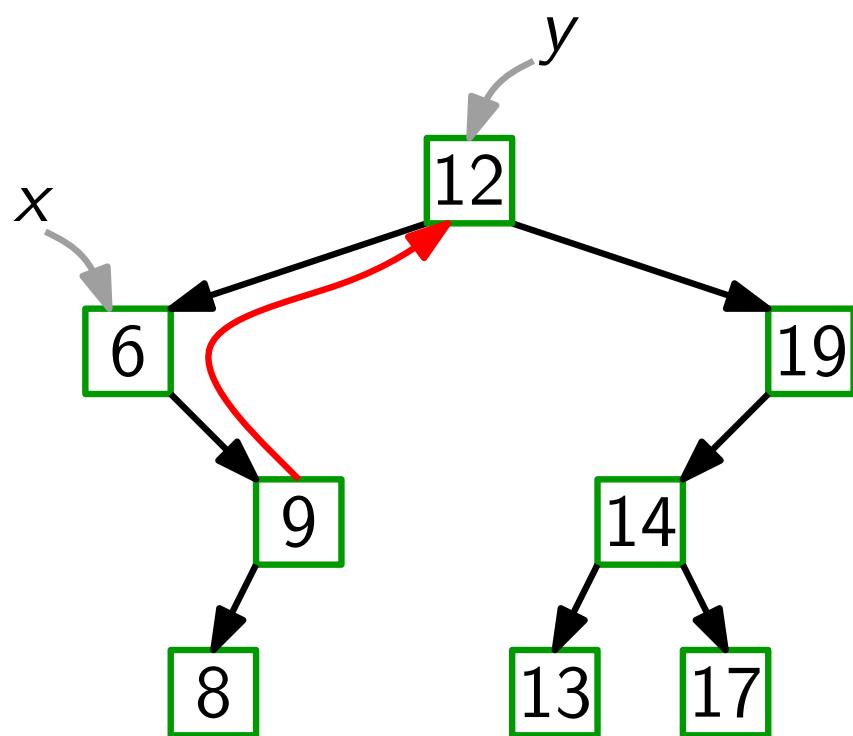
```

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

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

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



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

Node Successor(Node x)

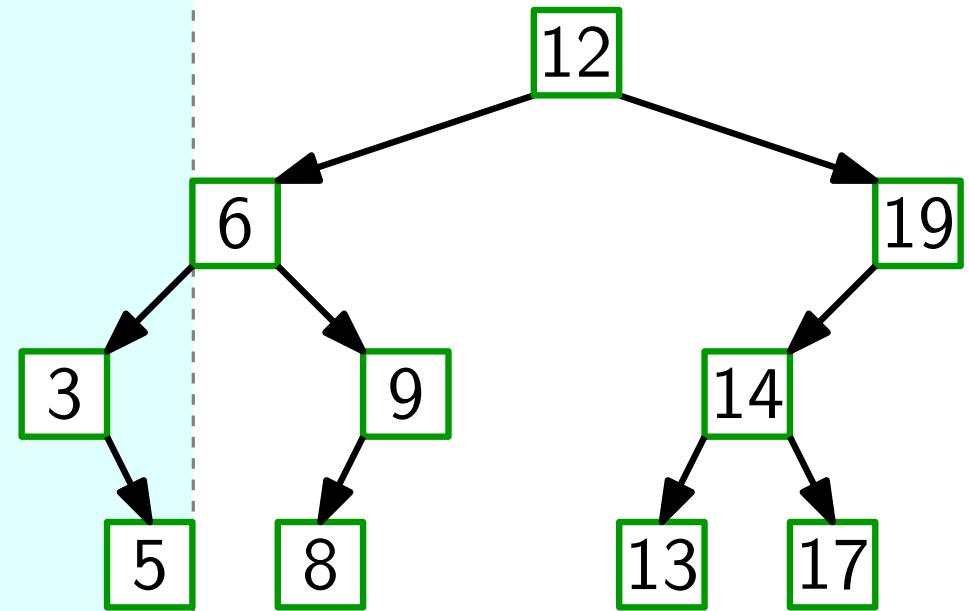
```

if x.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)



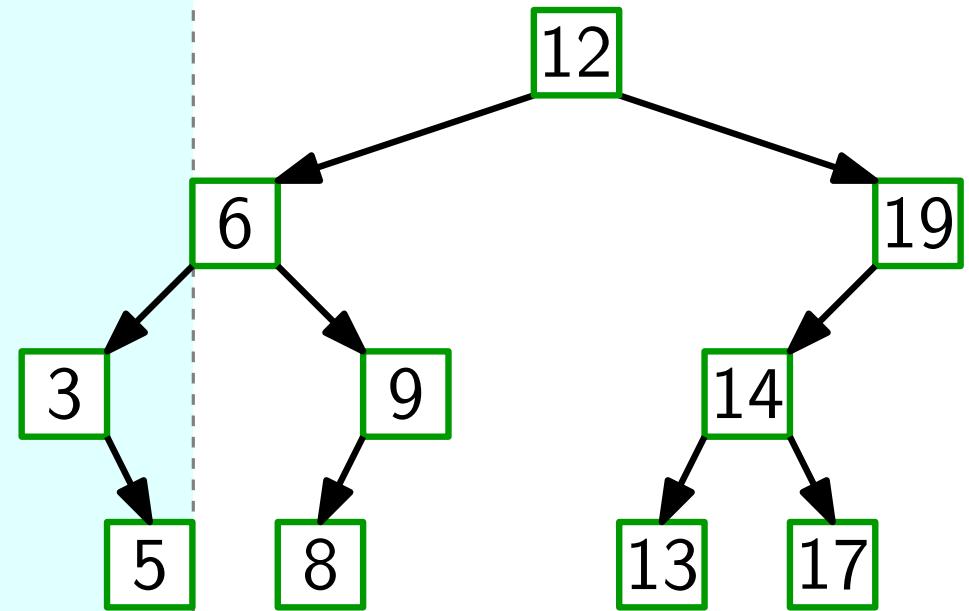
Insert(11)

Einfügen

Node Insert(key k)

$y = nil$

$x = root$



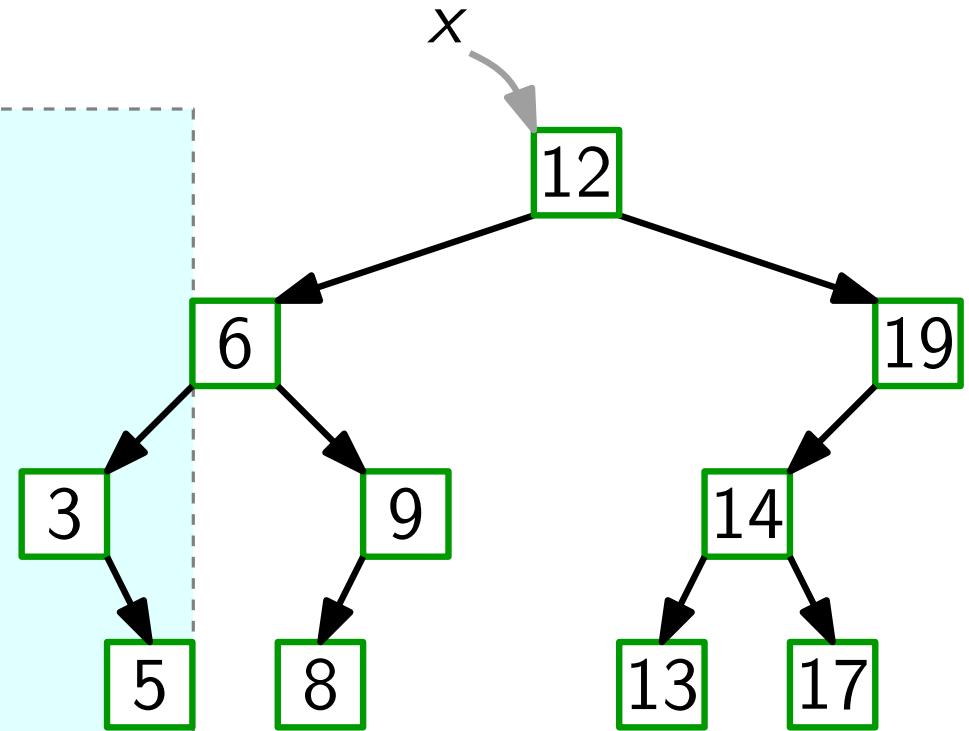
Insert(11)

Einfügen

Node Insert(key k)

$y = nil$

$x = root$



Insert(11)

Einfügen

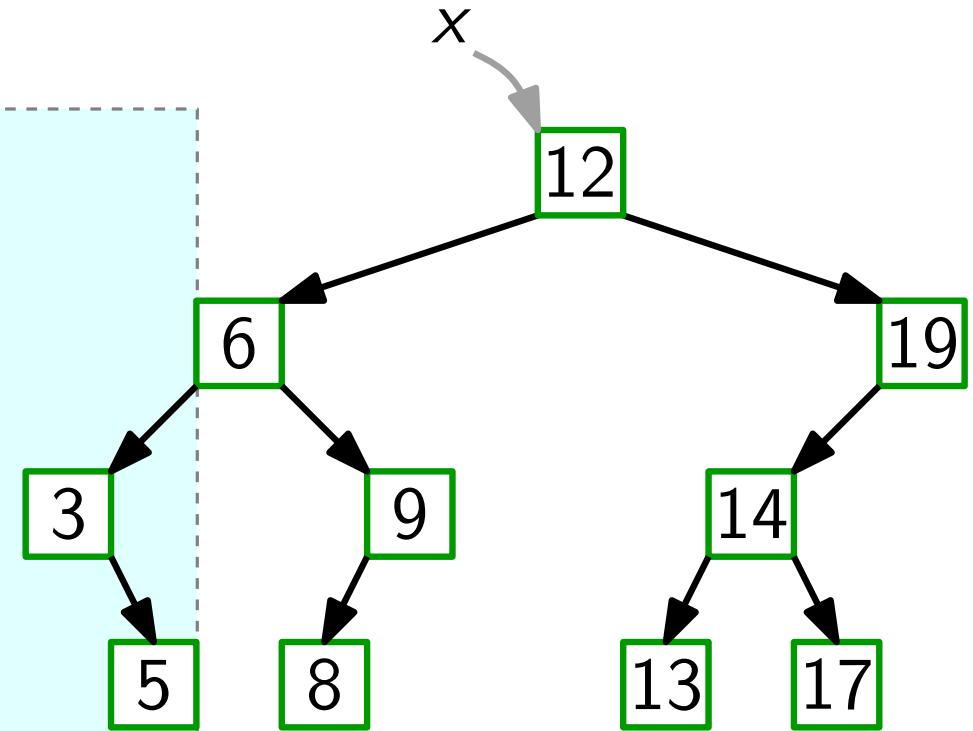
```
Node Insert(key k)
```

```
  y = nil
```

```
  x = root
```

```
  while x ≠ nil do
```

```
    y = x
```



Insert(11)

Einfügen

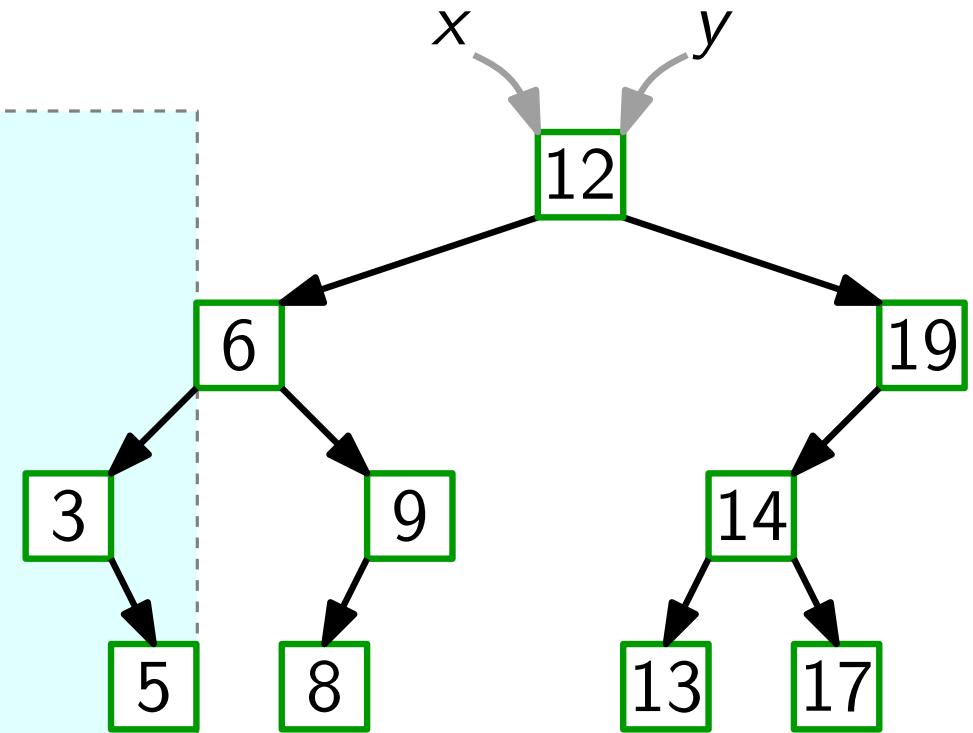
```
Node Insert(key k)
```

```
    y = nil
```

```
    x = root
```

```
    while x ≠ nil do
```

```
        y = x
```



Insert(11)

Einfügen

```
Node Insert(key k)
```

```
  y = nil
```

```
  x = root
```

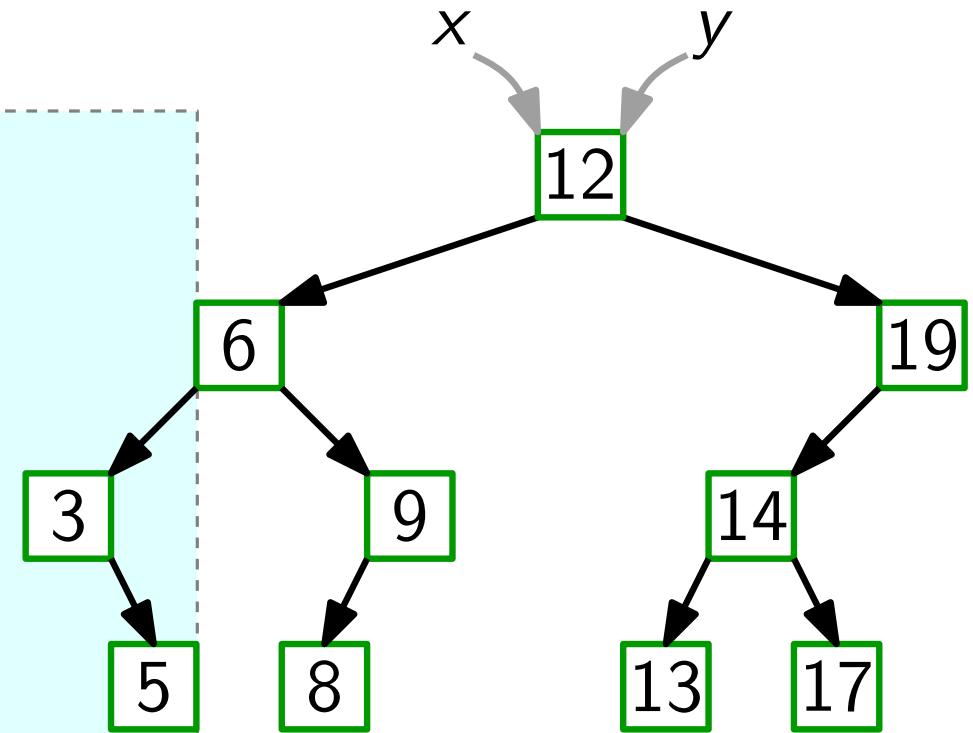
```
while x ≠ nil do
```

```
    y = x
```

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

```
        x = x.left
```

```
    else x = x.right
```

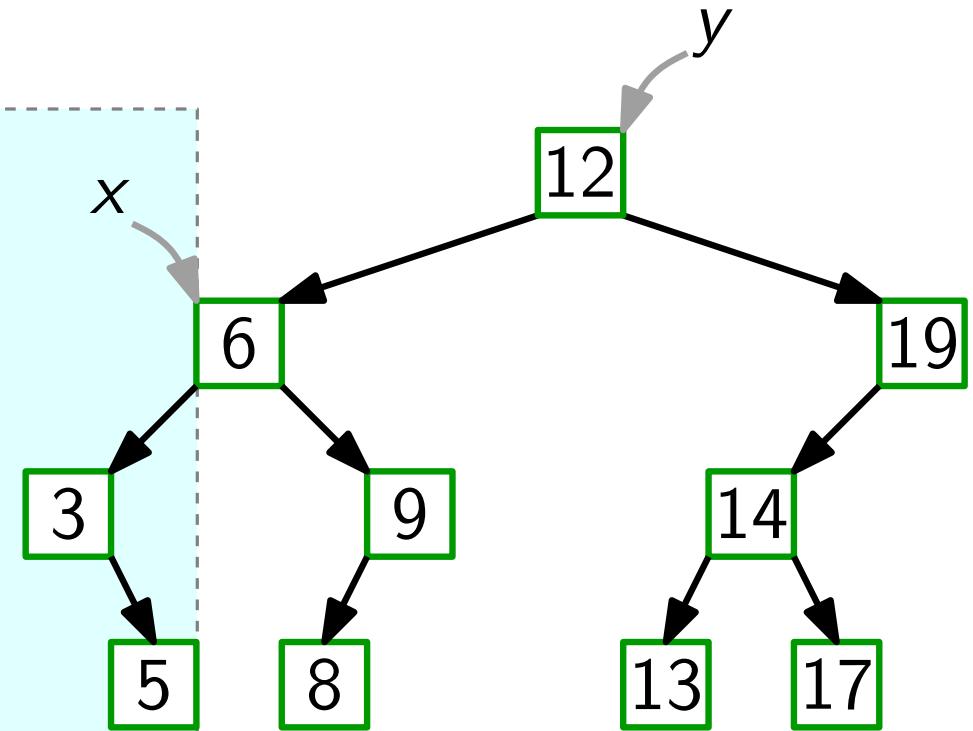


Insert(11)

Einfügen

```

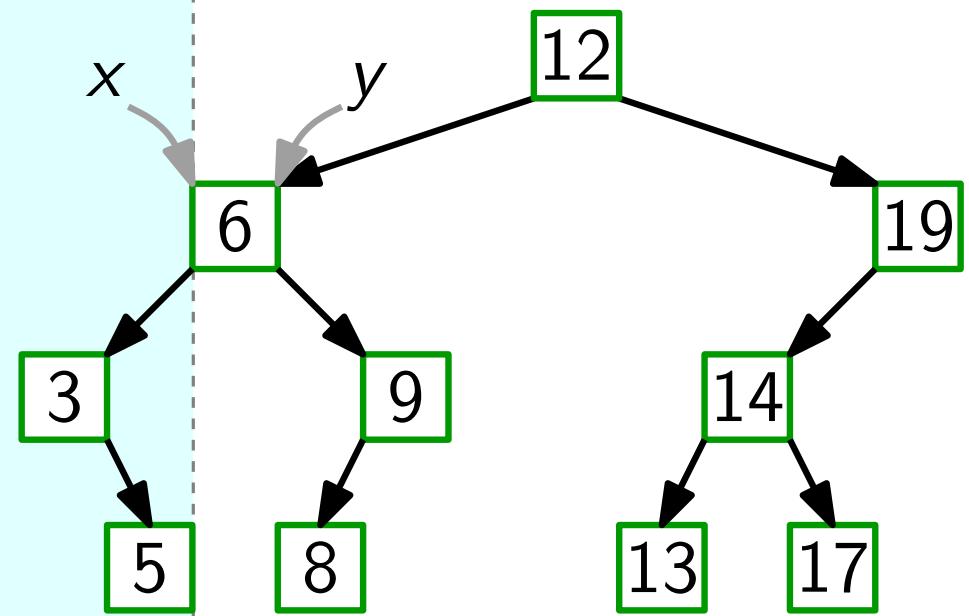
Node Insert(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  
```



Einfügen

```

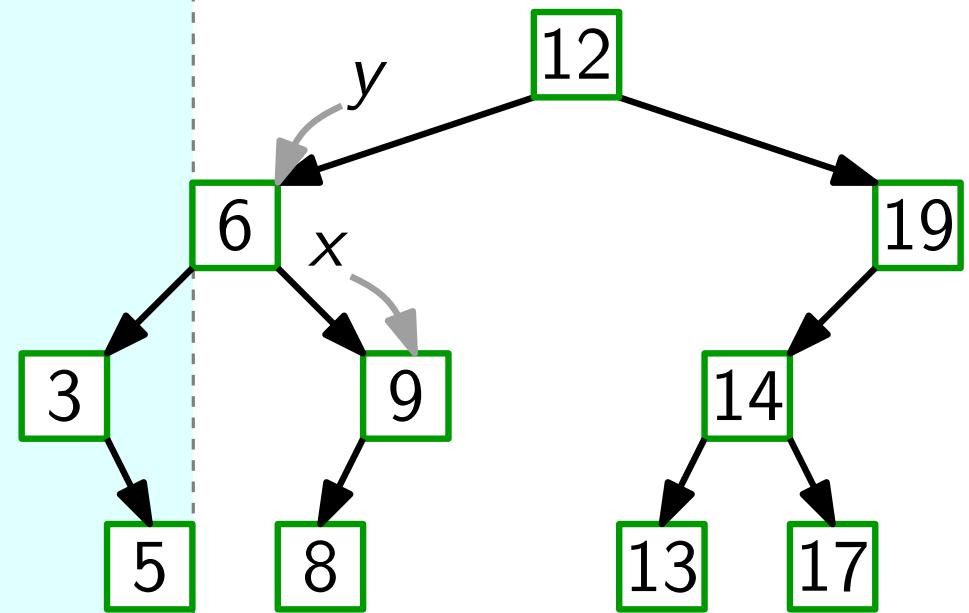
Node Insert(key k)
  y = nil
  x = root
  while x ≠ nil do
    y = x
    if k < x.key then
      x = x.left
    else x = x.right
  
```



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
```

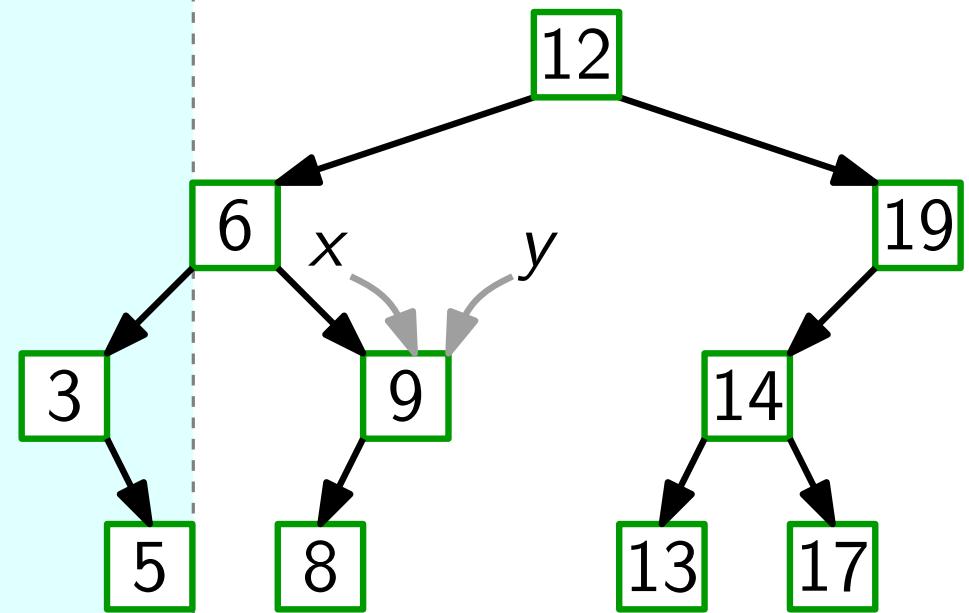


Insert(11)

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
  
```

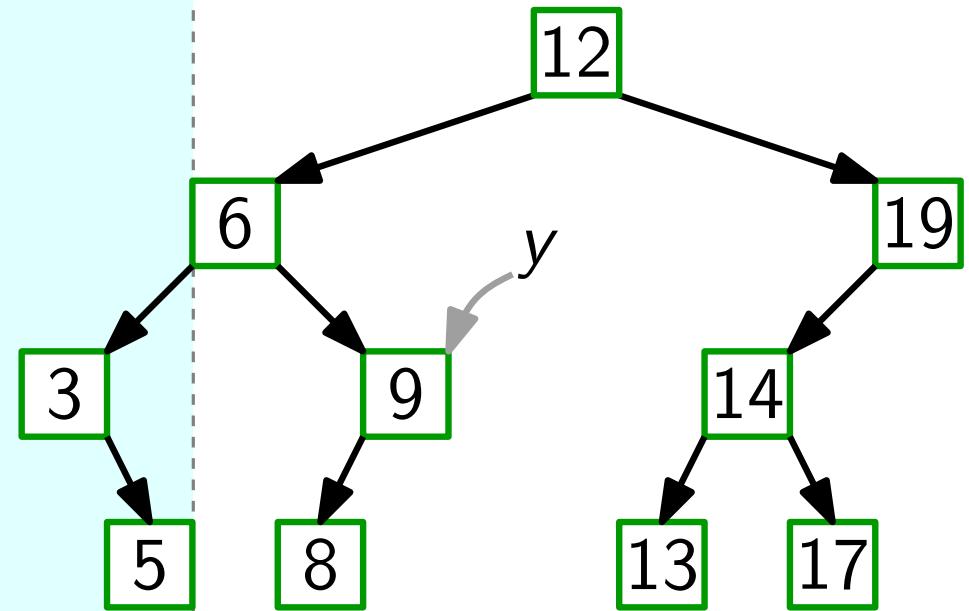


Insert(11)

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
  
```



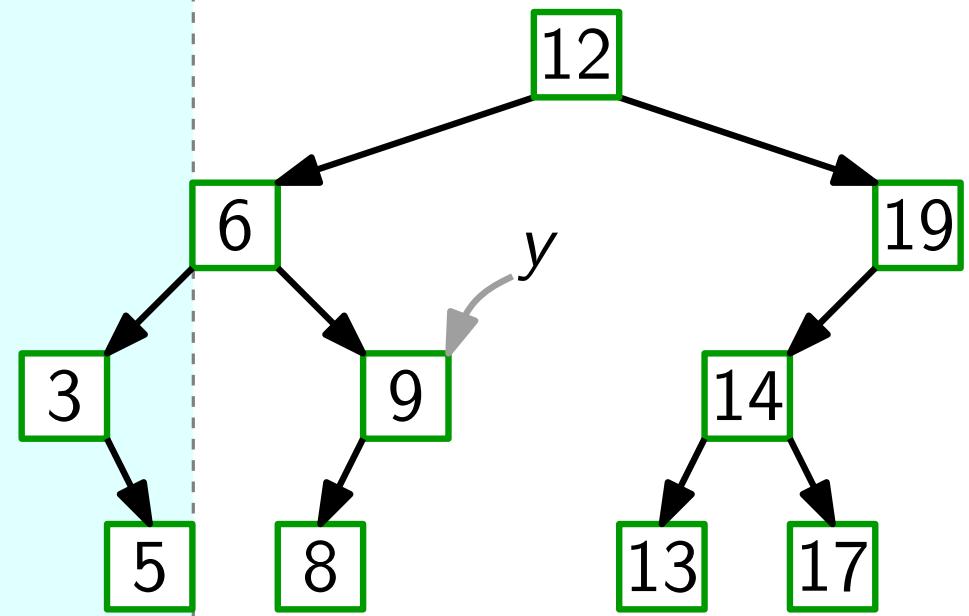
Insert(11)
 $x == \text{nil}$

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)

```

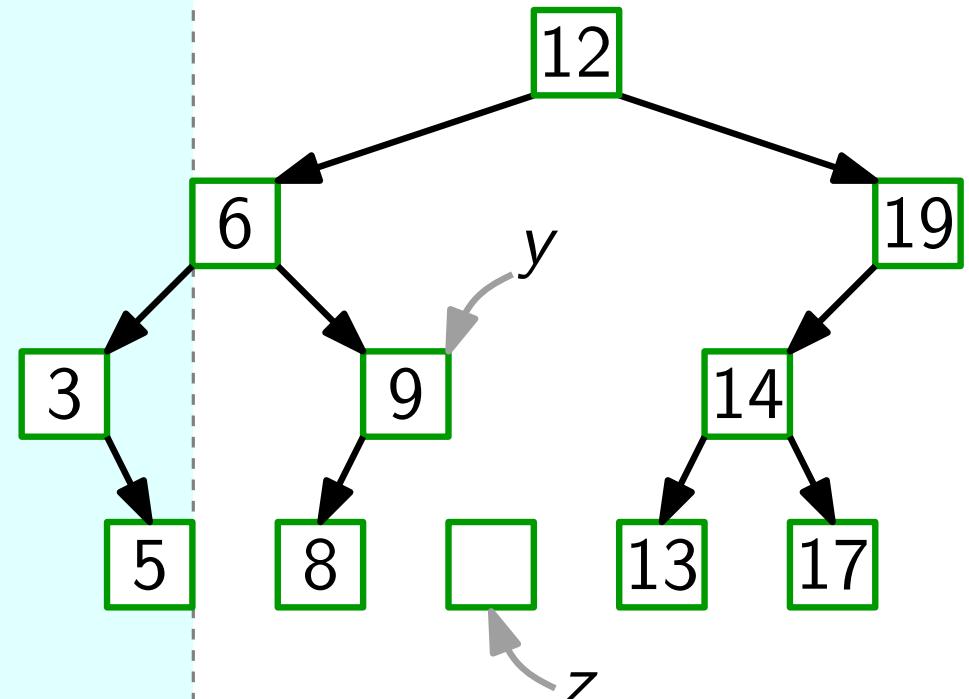


Einfügen

```

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)

```

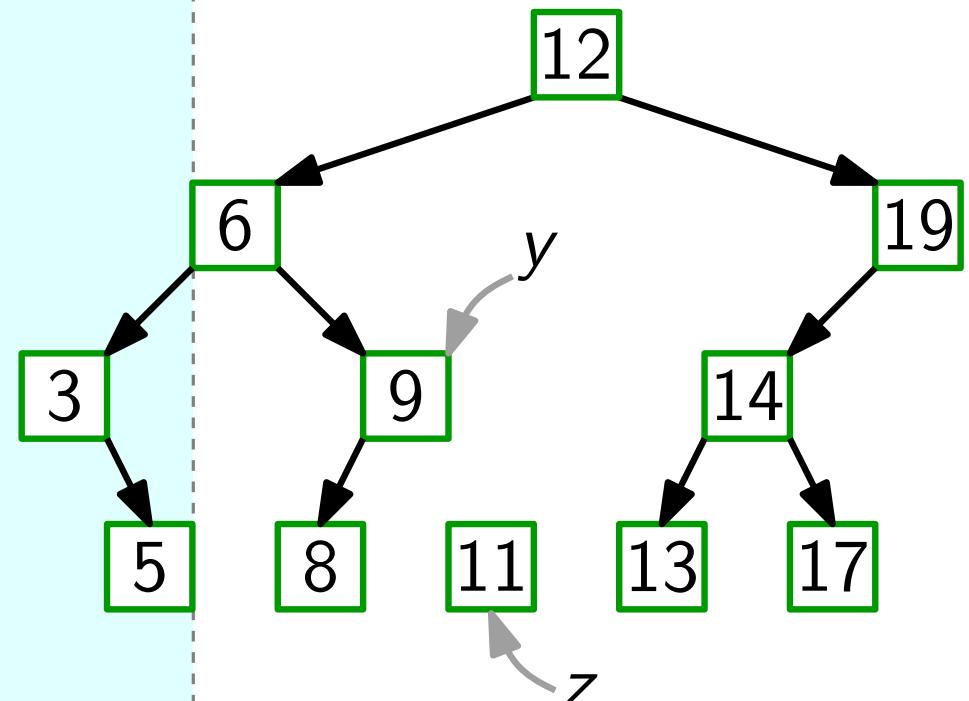


Einfügen

```

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)

```



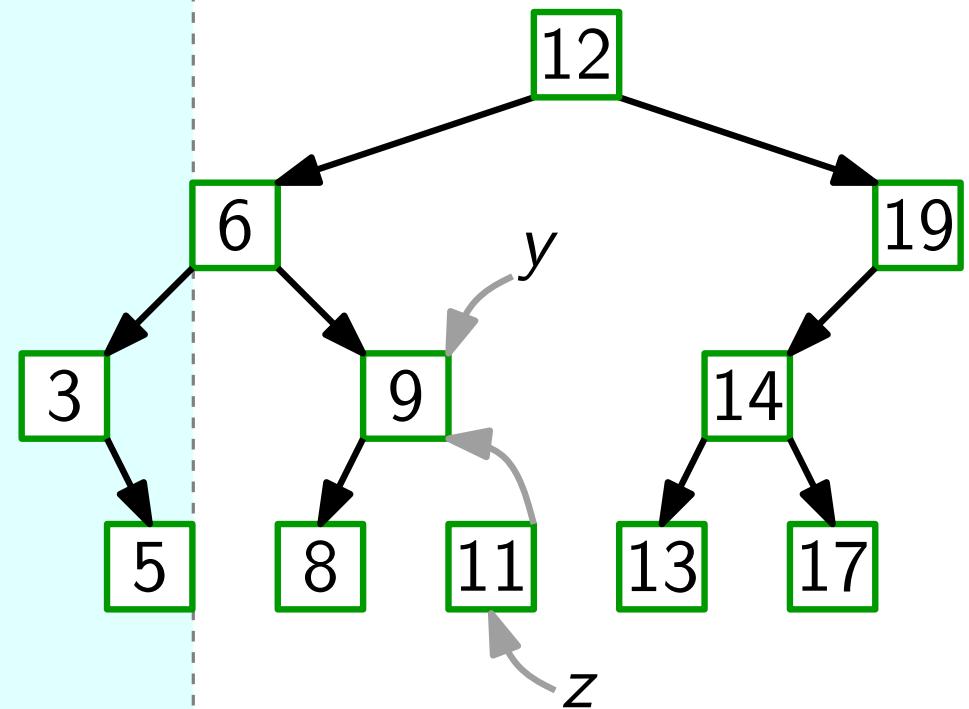
Insert(11)
 $x == nil$

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)

```



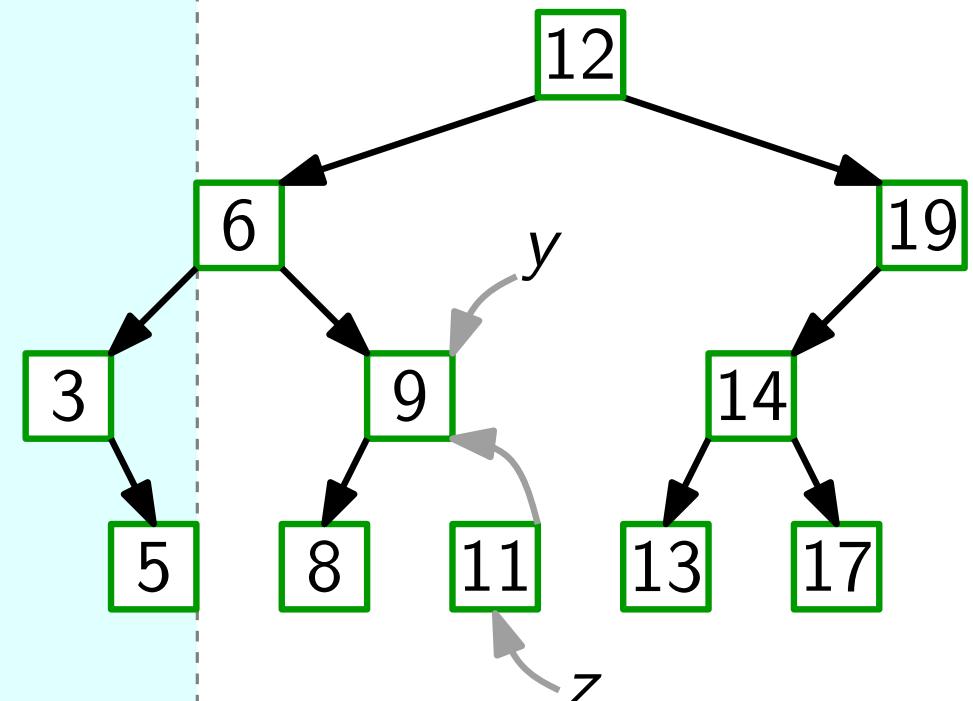
Insert(11)
 $x = nil$

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

```

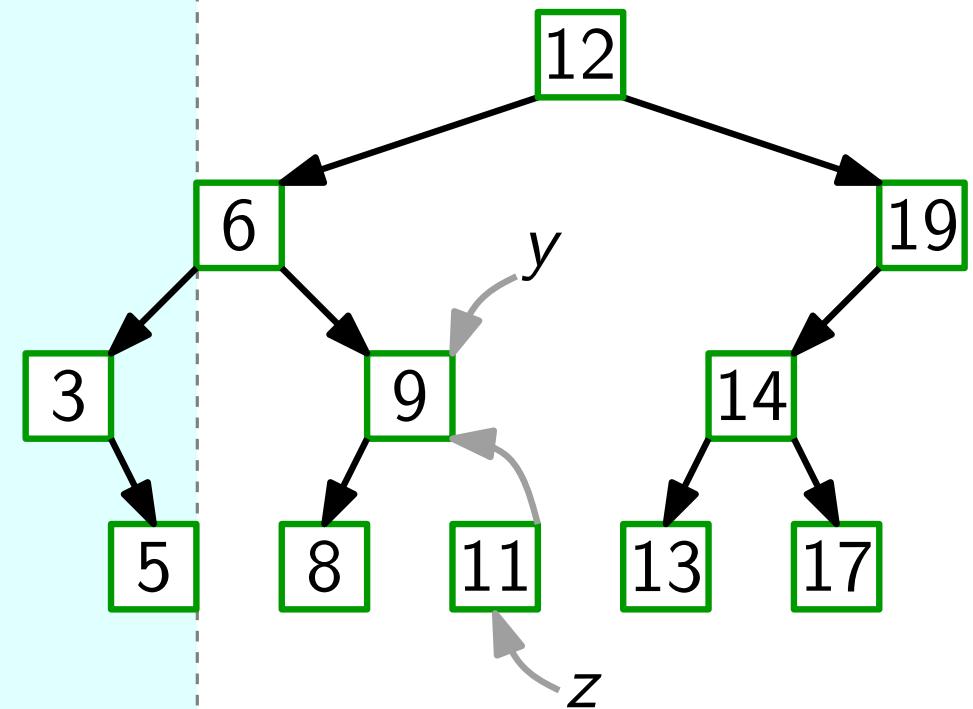


Insert(11)
 $x == nil$

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
    if k < y.key then y.left = z
    else y.right = z
  
```

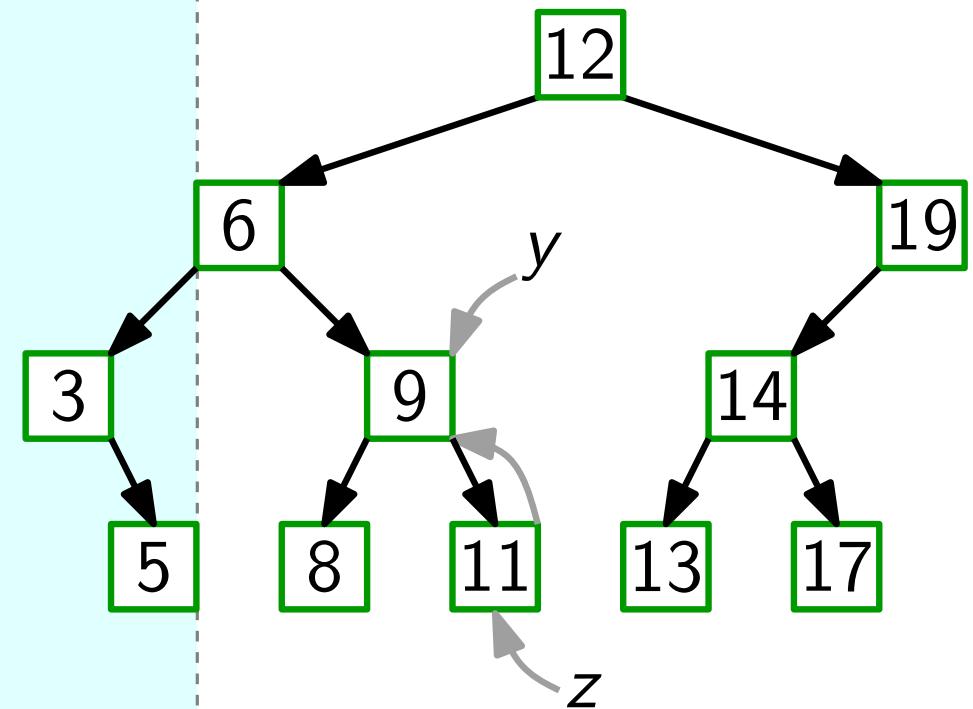


Insert(11)
 $x == nil$

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
  
```



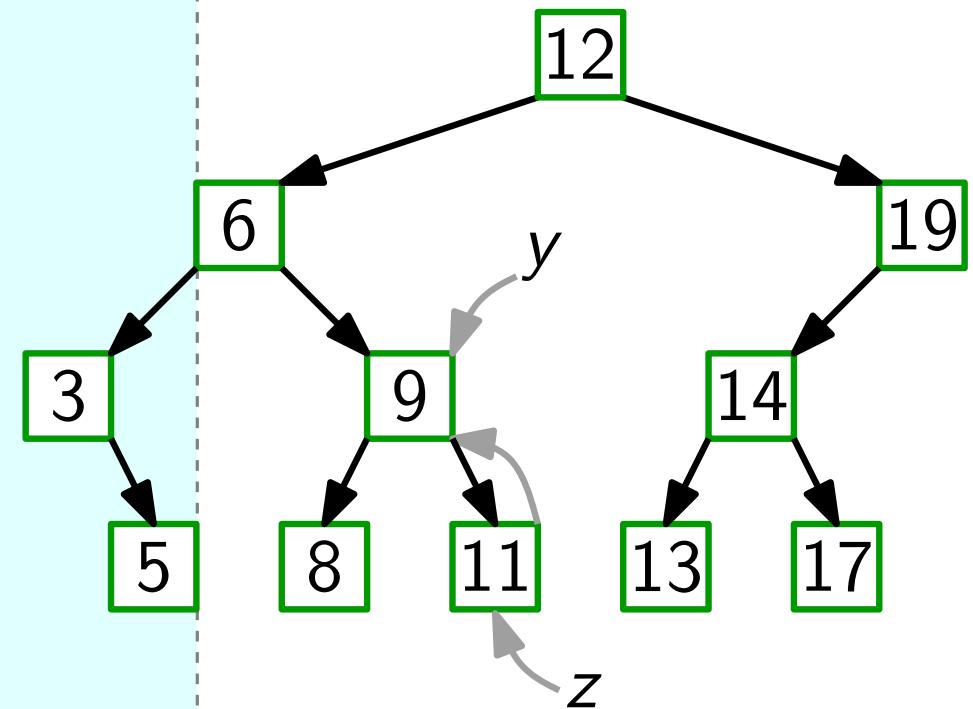
Insert(11)
 $x == nil$

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

```

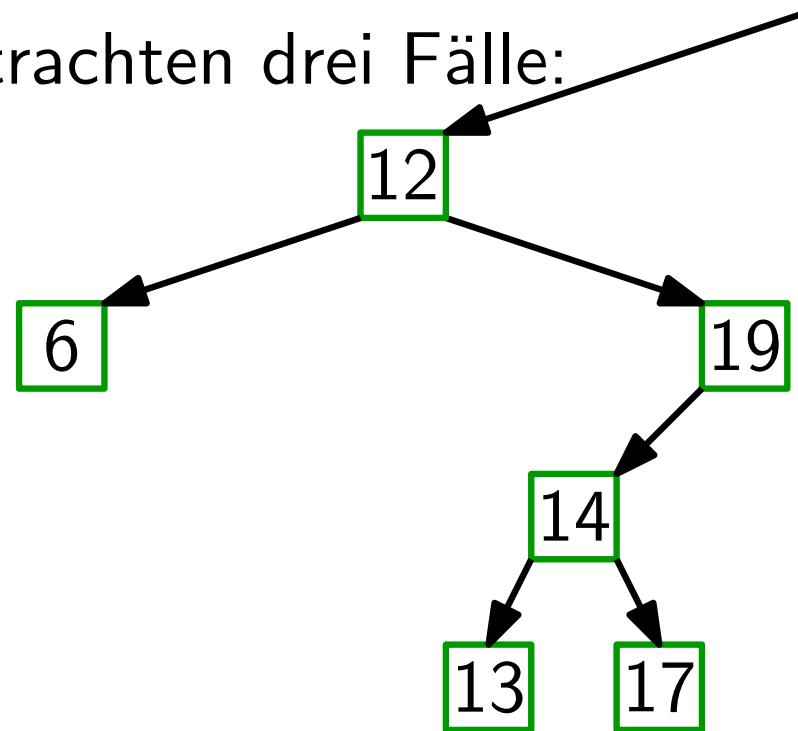


Insert(11)
 $x == nil$

Löschen

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

1. z hat keine Kinder.



2. z hat ein Kind x .

3. z hat zwei Kinder.

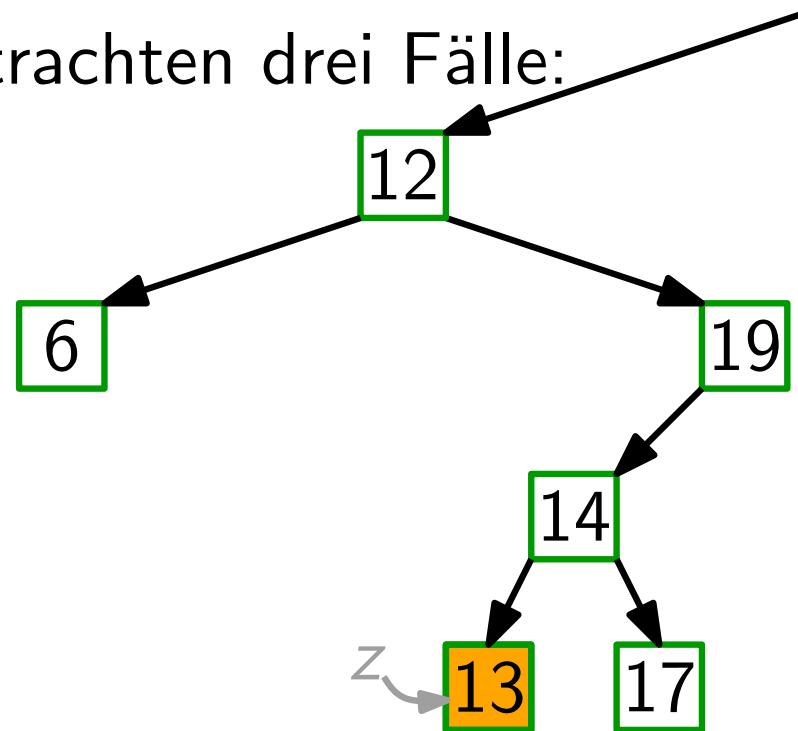
Löschen

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

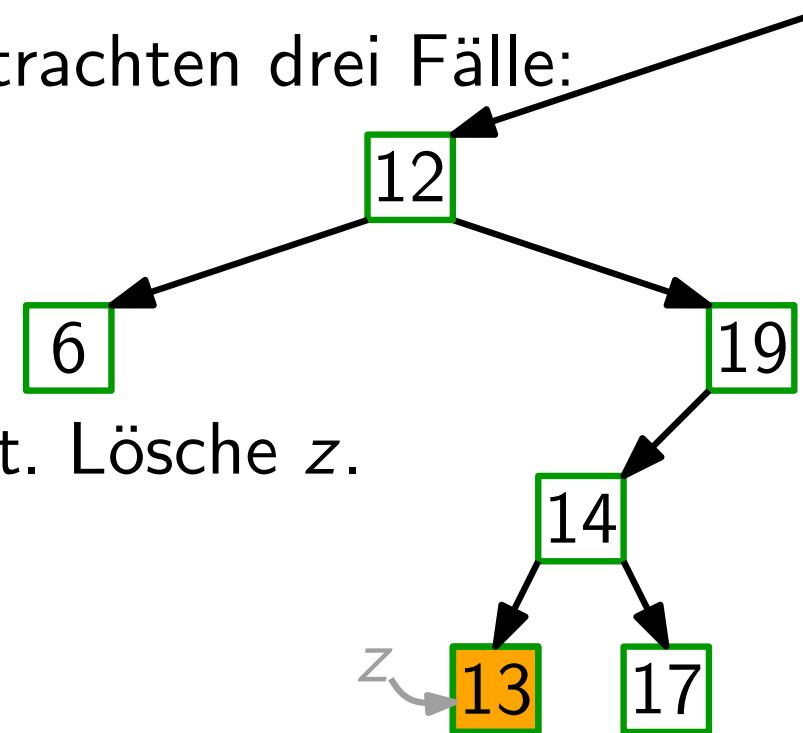
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

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

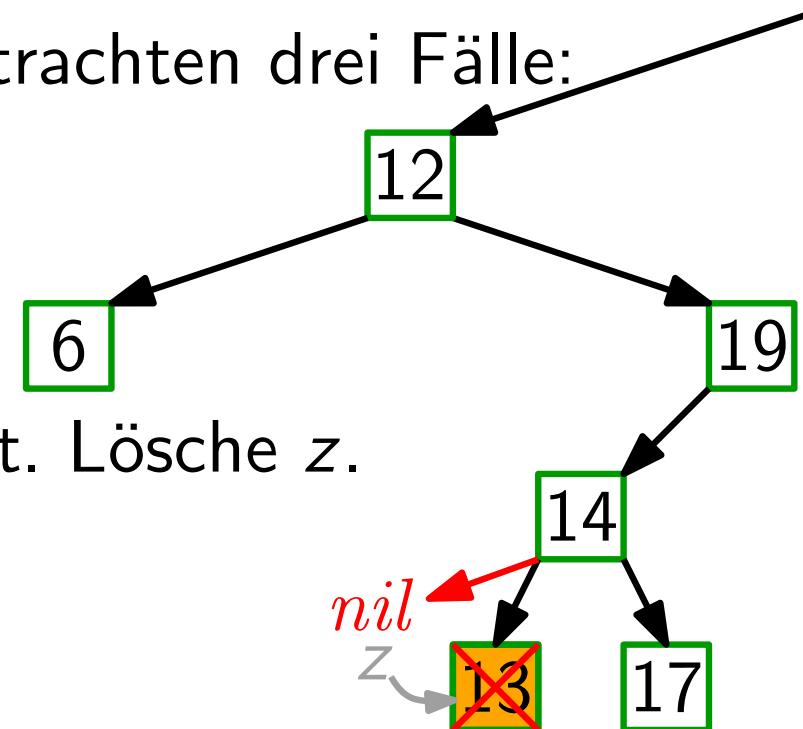
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

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

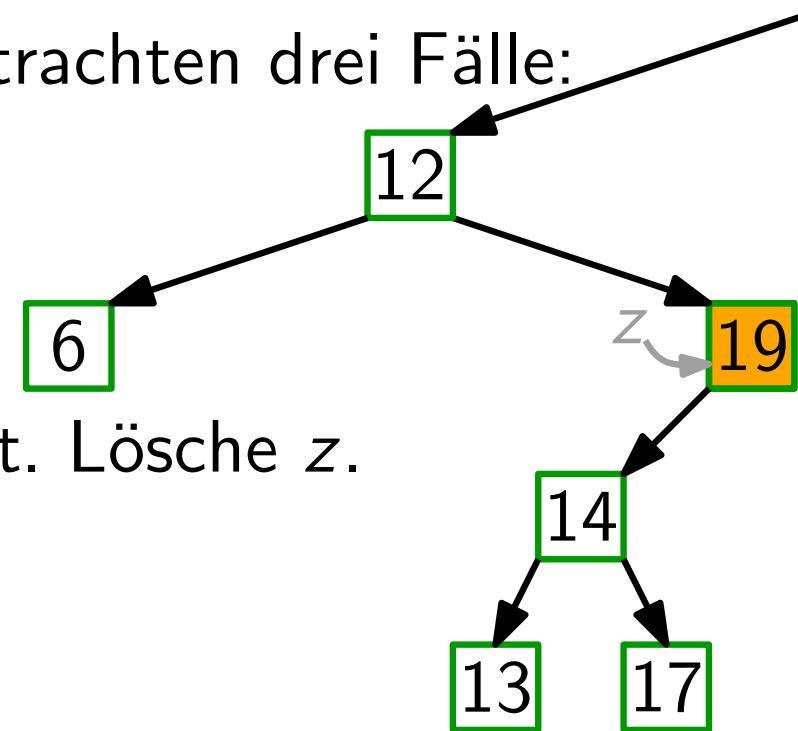
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

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 löschenende Knoten. Wir betrachten drei Fälle:

1. *z* hat keine Kinder.

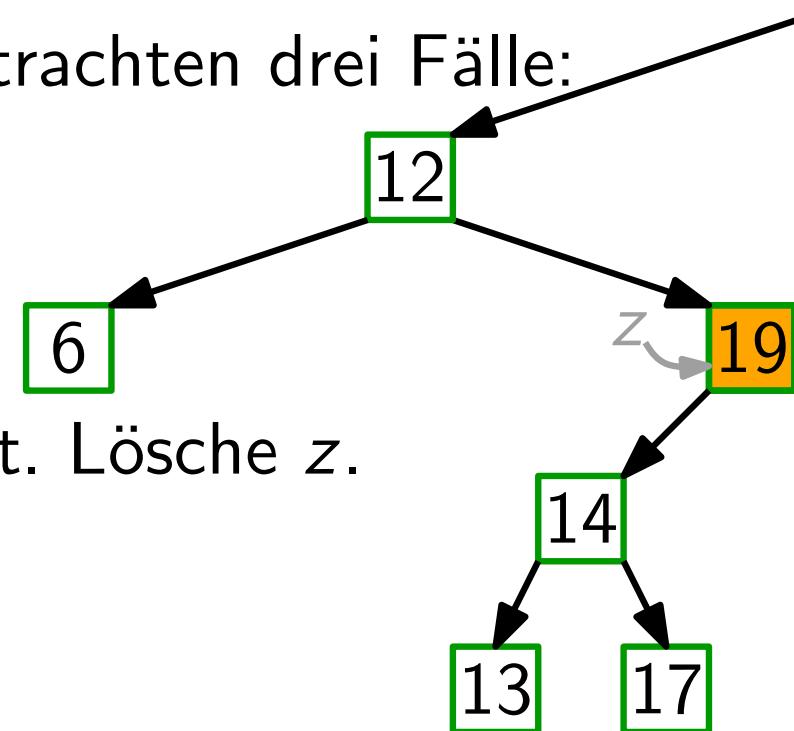
Falls z linkes Kind von $z.p$ ist, 6
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

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 löschenende Knoten. Wir betrachten drei Fälle:

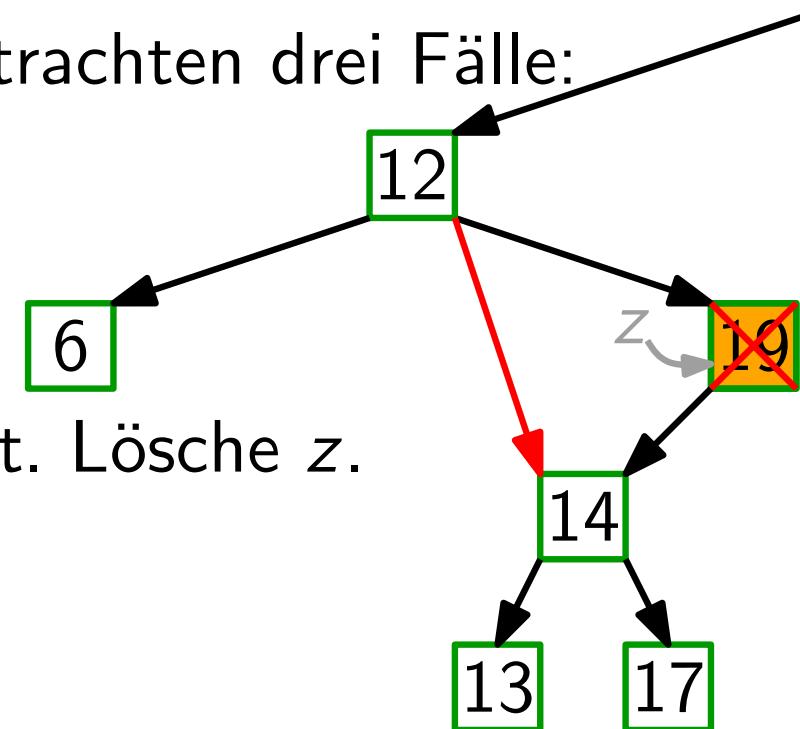
1. *z* hat keine Kinder.

Falls z linkes Kind von $z.p$ ist, 6
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

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

3. z hat zwei Kinder.



Löschen

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

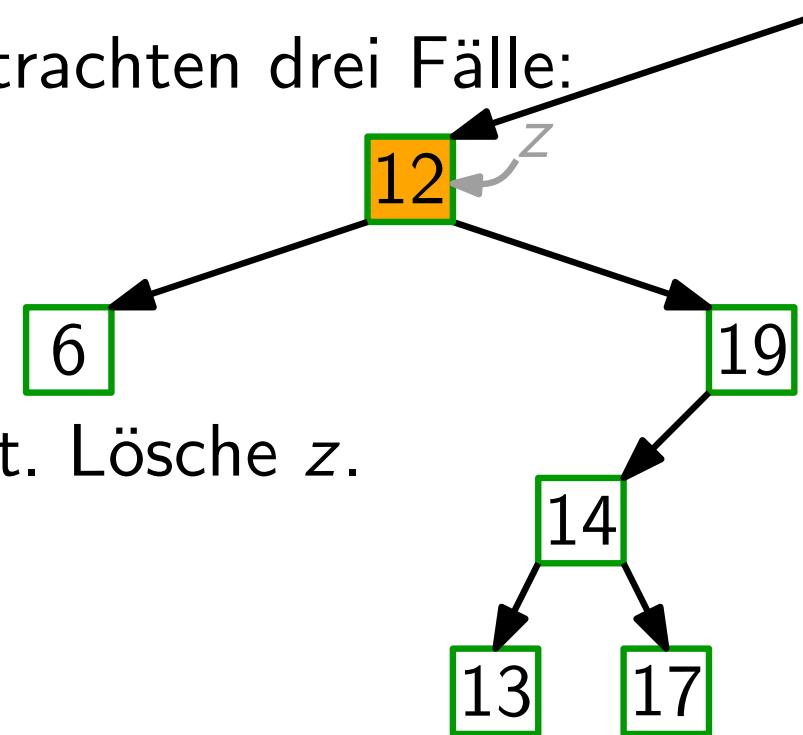
1. z hat keine Kinder.

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

2. z hat ein Kind x .

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

3. z hat zwei Kinder.



Löschen

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

1. z hat keine Kinder.

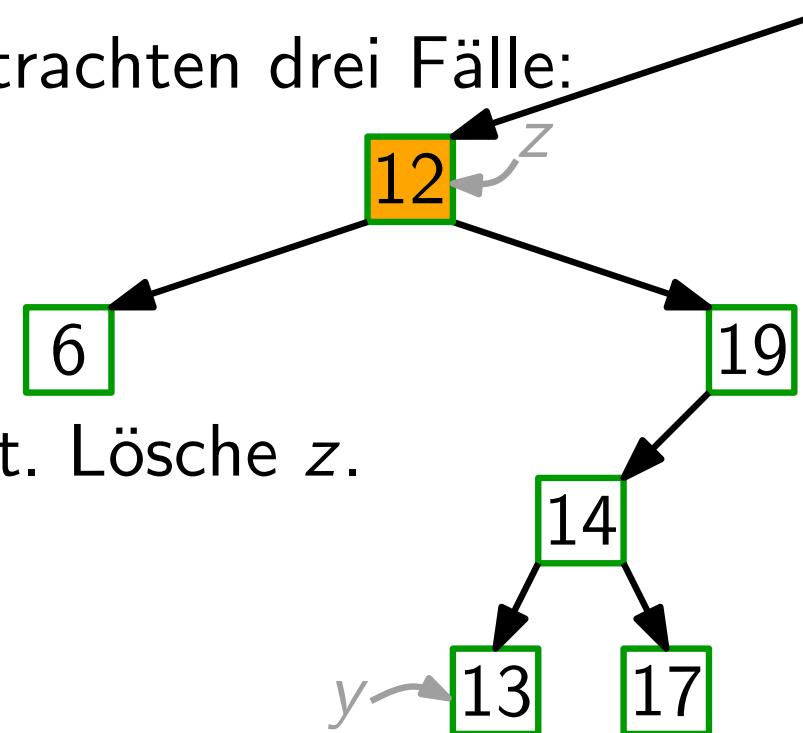
Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

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

3. z hat zwei Kinder.

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



Löschen

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

1. z hat keine Kinder.

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

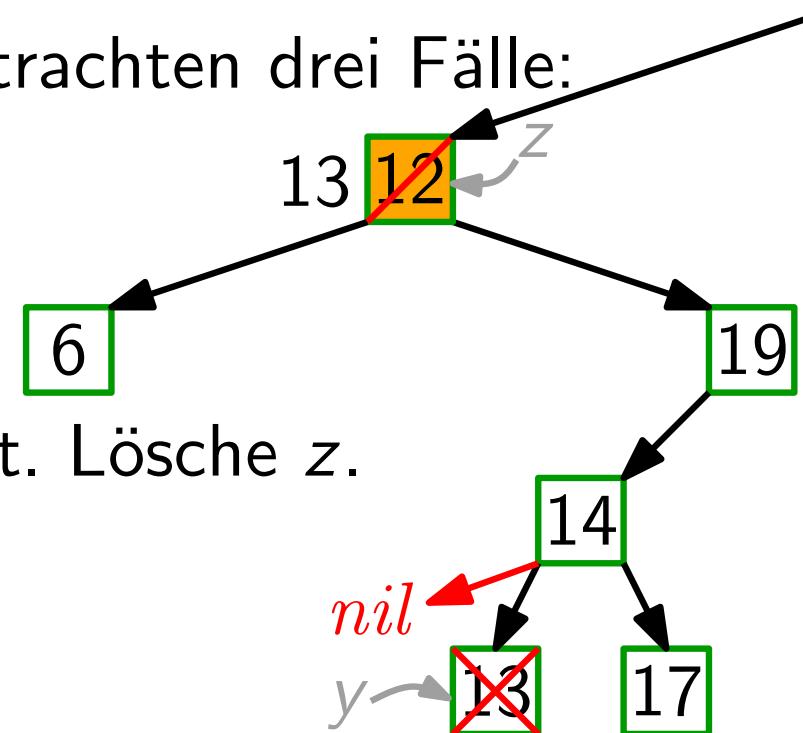
2. z hat ein Kind x .

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

Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.

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



Zusammenfassung

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

Zusammenfassung

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

Aber:

Zusammenfassung

- Satz.** Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $O(h)$ 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 $O(h)$ 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 $O(h)$ 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 $O(h)$ Zeit, wobei h die momentane Höhe des Baums ist.
- Aber:** Im schlechtesten Fall gilt $h \in \Theta(n)$.
- Ziel:** Suchbäume *balancieren* $\Rightarrow h \in O(\log n)$