Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2023/24

6. Vorlesung

Prioritäten setzen

Prof. Dr. Alexander Wolff *Lehrstuhl für Informatik I*

Guten Morgen!

Tipps für unseren ersten Zwischentest am Do, 16. November:

- Lesen Sie die Definitionen der Klassen O, Ω und Θ gaaaanz genau – bis Sie sie *restlos* verstehen! Besonders Beweise der Art $f \notin O(g)$ machen erfahrungsgemäß Schwierigkeiten.
- Lesen Sie alle Vorlesungsfolien (Vorlesungen 1–6) und Kap. 1–4 & 6 im Buch [CLRS]!
- Machen Sie möglichst viele Übungsaufgaben in Kap. 3, 4, 6 [CLRS]!
- Programmieren Sie – z.B. Pseudocode aus der Vorlesung!
- Stellen Sie Fragen – Kommilitonen, Tutoren, Erklärhiwis, mir!
- Haben Sie schon ein Buch??

Guten Morgen!

Tipps für unseren ersten Zwischentest am Do, 16. November:

- Lesen Sie die Definitionen der Klassen O, Ω und Θ gaaaanz genau – bis Sie sie *restlos* verstehen! Besonders Beweise der Art $f \notin O(g)$ machen erfahrungsgemäß Schwierigkeiten.
- Lesen Sie alle Vorlesungsfolien (Vorlesungen 1–6) und Kap. 1–4 & 6 im Buch [CLRS]!
- Machen Sie möglichst viele Übungsaufgaben in Kap. 3, 4, 6 [CLRS]!
- Programmieren Sie – z.B. Pseudocode aus der Vorlesung!
- Stellen Sie Fragen – Kommilitonen, Tutoren, Erklärhiwis, mir!
- Haben Sie schon ein Buch??

Tipp: Das Buch „Algorithmen & Datenstrukturen: Die Grundwerkzeuge“ von Dietzfelbinger, Mehlhorn und Sanders (Springer, 2014) kann man im Uninetz kostenlos von der Unibib herunterladen.

Lesen!!

Guten Morgen!

Tipps für unseren ersten Zwischentest am Do, 16. November:

- Lesen Sie die Definitionen der Klassen O, Ω und Θ gaaaanz genau – bis Sie sie *restlos* verstehen! Besonders Beweise der Art $f \notin O(g)$ machen erfahrungsgemäß Schwierigkeiten.
- Lesen Sie alle Vorlesungsfolien (Vorlesungen 1–6) und Kap. 1–4 & 6 im Buch [CLRS]!
- Machen Sie möglichst viele Übungsaufgaben in Kap. 3, 4, 6 [CLRS]!
- Programmieren Sie – z.B. Pseudocode aus der Vorlesung!
- Stellen Sie Fragen – Kommilitonen, Tutoren, Erklärhiwis, mir!
- Haben Sie schon ein Buch??

Tipp: Das Buch „Algorithmen & Datenstrukturen: Die Grundwerkzeuge“ von Dietzfelbinger, Mehlhorn und Sanders (Springer, 2014) kann man im Uninetz kostenlos von der Unibib herunterladen. *Lesen!!*

- Erinnern Sie sich an die Linearzeitlösung für MaxSum?

Guten Morgen!

Tipps für unseren ersten Zwischentest am Do, 16. November:

- Lesen Sie die Definitionen der Klassen O, Ω und Θ gaaaanz genau – bis Sie sie *restlos* verstehen! Besonders Beweise der Art $f \notin O(g)$ machen erfahrungsgemäß Schwierigkeiten.
- Lesen Sie alle Vorlesungsfolien (Vorlesungen 1–6) und Kap. 1–4 & 6 im Buch [CLRS]!
- Machen Sie möglichst viele Übungsaufgaben in Kap. 3, 4, 6 [CLRS]!
- Programmieren Sie – z.B. Pseudocode aus der Vorlesung!
- Stellen Sie Fragen – Kommilitonen, Tutoren, Erklärhiwis, mir!
- Haben Sie schon ein Buch??

Tipp: Das Buch „Algorithmen & Datenstrukturen: Die Grundwerkzeuge“ von Dietzfelbinger, Mehlhorn und Sanders (Springer, 2014) kann man im Uninetz kostenlos von der Unibib herunterladen. *Lesen!!*

- Erinnern Sie sich an die Linearzeitlösung für MaxSum? Beweisen Sie ihre Korrektheit mit einer Schleifeninvarianten!

Lösen der Übungsaufgaben

- Geben Sie auf Ihren Lösungen immer die Namen aller (≤ 3) Autoren an – nur die bekommen Punkte!

Lösen der Übungsaufgaben

- Geben Sie auf Ihren Lösungen immer die Namen aller (≤ 3) Autoren an – nur die bekommen Punkte!

- Lösen Sie Aufgaben möglichst nur mit Mitgliedern *Ihrer* Übungsgruppe. So sehen Sie sich automatisch jede Woche.

Lösen der Übungsaufgaben

- Geben Sie auf Ihren Lösungen immer die Namen aller (≤ 3) Autoren an – nur die bekommen Punkte!
- Lösen Sie Aufgaben möglichst nur mit Mitgliedern *Ihrer* Übungsgruppe. So sehen Sie sich automatisch jede Woche.
- Wenn Sie nicht immer *alle* Aufgaben lösen können – nicht verzweifeln!

Lösen der Übungsaufgaben

- Geben Sie auf Ihren Lösungen immer die Namen aller (≤ 3) Autoren an – nur die bekommen Punkte!

- Lösen Sie Aufgaben möglichst nur mit Mitgliedern *Ihrer* Übungsgruppe. So sehen Sie sich automatisch jede Woche.

- Wenn Sie nicht immer *alle* Aufgaben lösen können – nicht verzweifeln! Wichtig ist, dass Sie's probiert haben!

Lösen der Übungsaufgaben

- Geben Sie auf Ihren Lösungen immer die Namen aller (≤ 3) Autoren an – nur die bekommen Punkte!
- Lösen Sie Aufgaben möglichst nur mit Mitgliedern *Ihrer* Übungsgruppe. So sehen Sie sich automatisch jede Woche.
- Wenn Sie nicht immer *alle* Aufgaben lösen können – nicht verzweifeln! Wichtig ist, dass Sie's probiert haben!
- Benützen Sie zum Fragen/Diskutieren das Diskussionsforum im ADS-Kursraum in WueCampus.

Heute: Wir „bauen“ eine Datenstruktur

Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Heute: Wir „bauen“ eine Datenstruktur

Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp

Implementierung

Heute: Wir „bauen“ eine Datenstruktur

Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung

Heute: Wir „bauen“ eine Datenstruktur

Datenstruktur:

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung

wie wird die gewünschte Funktionalität realisiert:
– wie sind die Daten gespeichert (Feld, Liste, . . .)?
– welche Algorithmen implementieren die Operationen?

Anwendung: Prozesssteuerung

Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anwendung: Prozesssteuerung

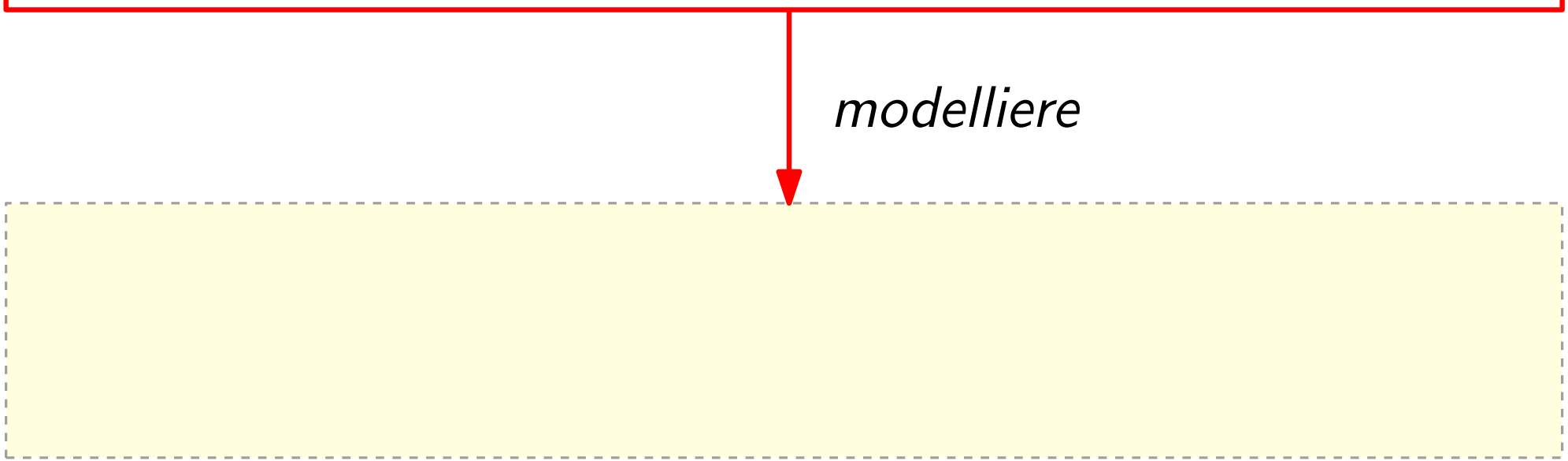
Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anforderung:
- Prozesse (mit ihrer Priorität) einfügen
- Prozess mit höchster Priorität finden/löschen
- Priorität von Prozessen erhöhen

Anwendung: Prozesssteuerung

Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anforderung:
- Prozesse (mit ihrer Priorität) einfügen
- Prozess mit höchster Priorität finden/löschen
- Priorität von Prozessen erhöhen



Anwendung: Prozesssteuerung

Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anforderung:
- Prozesse (mit ihrer Priorität) einfügen
- Prozess mit höchster Priorität finden/löschen
- Priorität von Prozessen erhöhen

modelliere

Abstrakter Datentyp: Prioritätsschlange

Anwendung: Prozesssteuerung

Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anforderung:
- Prozesse (mit ihrer Priorität) einfügen
- Prozess mit höchster Priorität finden/löschen
- Priorität von Prozessen erhöhen

modelliere

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	*Funktionalität*

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	*Funktionalität*
Insert(element x)	
element FindMax()	
element ExtractMax()	
IncreaseKey (element x, priorität p)	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	*Funktionalität*
Insert(element x)	$M = M \cup \{x\}$
element FindMax()	
element ExtractMax()	
IncreaseKey (element x, priorität p)	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	*Funktionalität*
Insert(element x)	$M = M \cup \{x\}$
element FindMax()	liefere $x \in M$ mit $x.key = \max\{y.key \mid y \in M\}$
element ExtractMax()	
IncreaseKey (element x, priorität p)	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	*Funktionalität*
Insert(element x)	$M = M \cup \{x\}$
element FindMax()	liefere $x \in M$ mit $x.key = \max\{y.key \mid y \in M\}$
element ExtractMax()	$x =$ FindMax(); $M = M \setminus \{x\}$; liefere x
IncreaseKey (element x, priorität p)	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	*Funktionalität*
Insert(element x)	$M = M \cup \{x\}$
element FindMax()	liefere $x \in M$ mit $x.key = \max\{y.key \mid y \in M\}$
element ExtractMax()	$x =$ FindMax(); $M = M \setminus \{x\}$; liefere x
IncreaseKey (element x, priorität p)	$x.key = p$

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

W-C-Laufzeiten meiner Implement.

Insert	FindMax	ExtractMax	IncreaseKey
$\Theta(\quad)$	$\Theta(\quad)$	$\Theta(\quad)$	$\Theta(\quad)$

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(\quad)$	$\Theta(\quad)$	$\Theta(\quad)$	$\Theta(\quad)$

$\star$
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(\quad)$	$\Theta(\quad)$	$\Theta(\quad)$

$\star$
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(\quad)$	$\Theta(\quad)$

$\star$
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

W-C-Laufzeiten meiner Implement.*

Insert	FindMax	ExtractMax	IncreaseKey
$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\quad)$

*
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

$\star$
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Achtung: Das Feld bekommt bei einer naiven Implementierung durch ExtractMax im Laufe der Zeit Lücken. Wie kann man das verhindern, ohne Elemente zu verschieben?

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
heute: Implementierung als Heap (Haufen)				

*
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Achtung: Das Feld bekommt bei einer naiven Implementierung durch ExtractMax im Laufe der Zeit Lücken. Wie kann man das verhindern, ohne Elemente zu verschieben?

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
heute: Implementierung als Heap (Haufen)	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

$\star$
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Achtung: Das Feld bekommt bei einer naiven Implementierung durch ExtractMax im Laufe der Zeit Lücken. Wie kann man das verhindern, ohne Elemente zu verschieben?

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

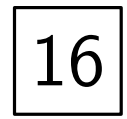
	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
heute: Implementierung als Heap (Haufen)	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Das ist exponentiell besser!

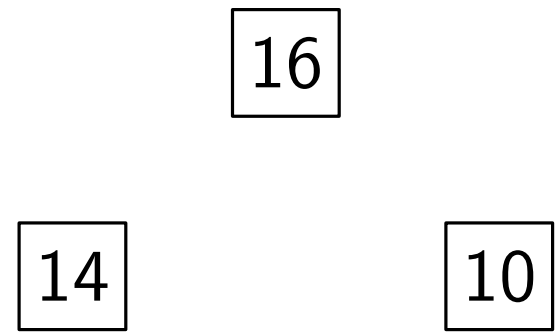
$\star$
- Daten werden in einem Feld gespeichert.
- Neue Elemente werden hinten angehängt (unsortiert).
- Maximum wird immer aufrechterhalten.
- Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Achtung: Das Feld bekommt bei einer naiven Implementierung durch ExtractMax im Laufe der Zeit Lücken. Wie kann man das verhindern, ohne Elemente zu verschieben?

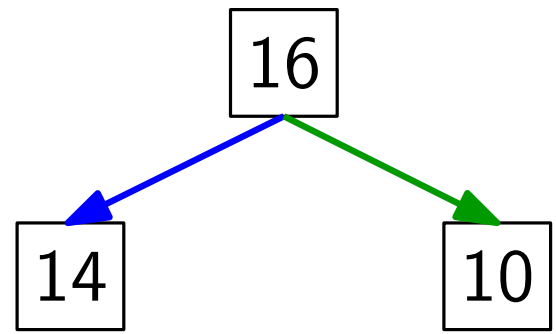
Bäume, gut gepackt



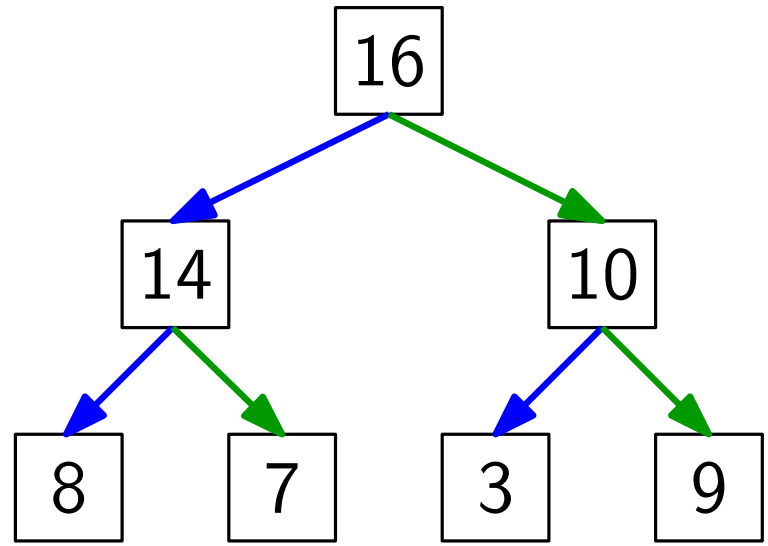
Bäume, gut gepackt



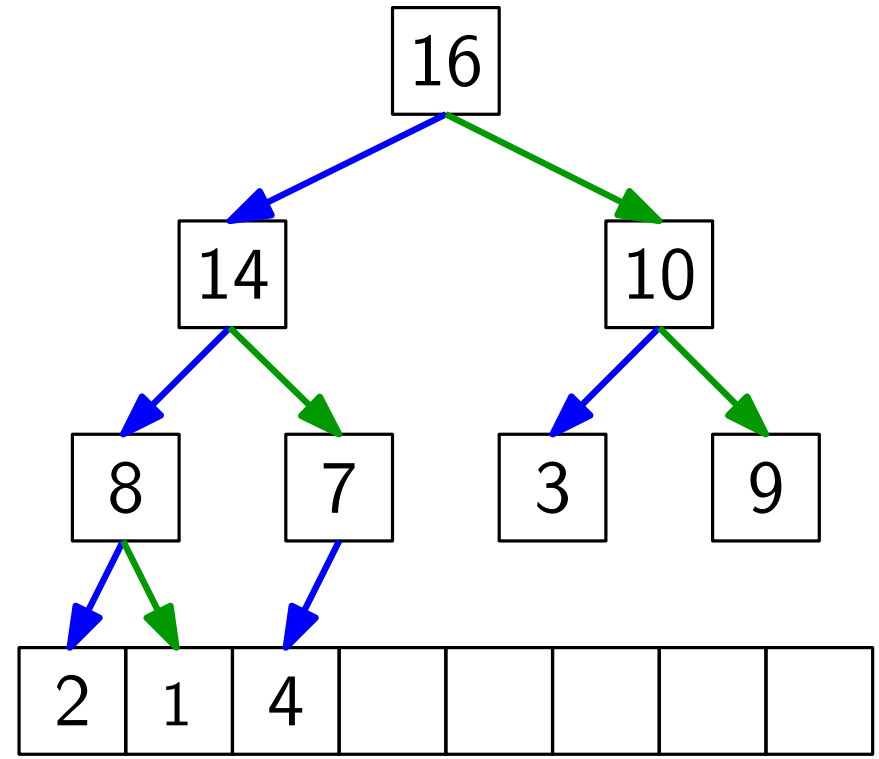
Bäume, gut gepackt



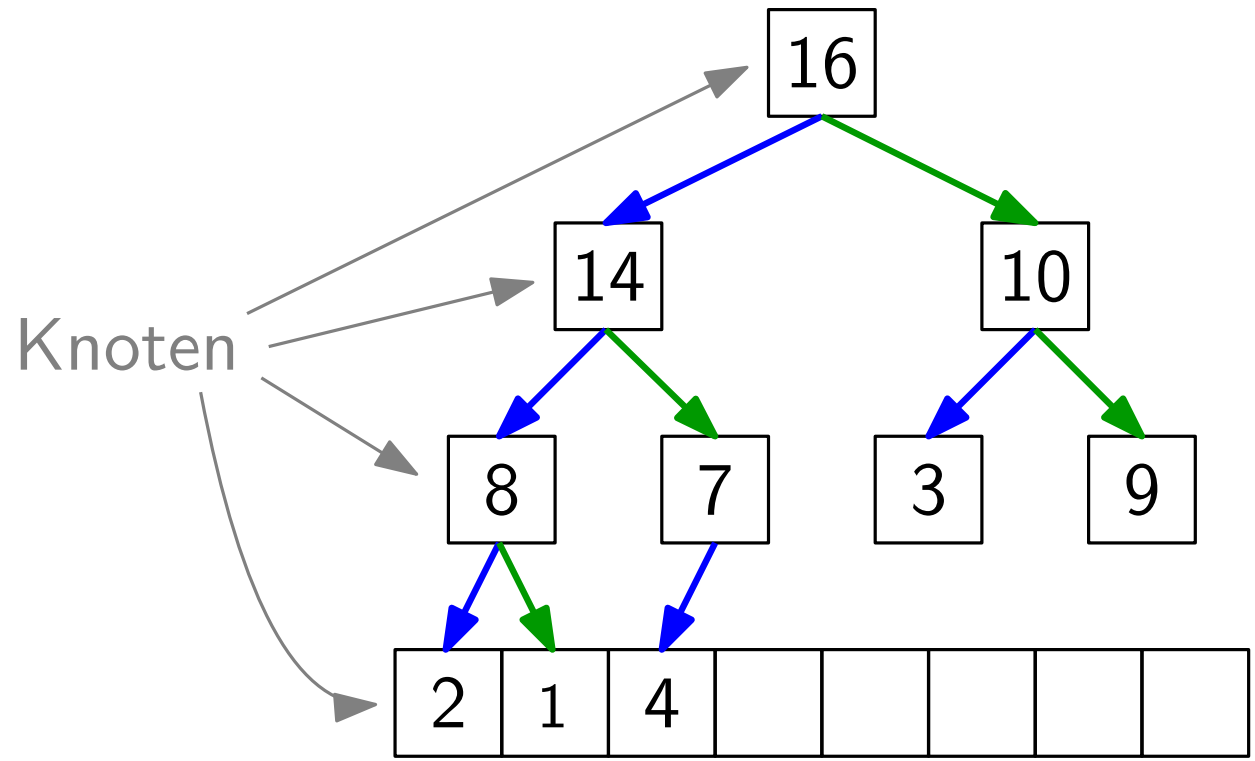
Bäume, gut gepackt



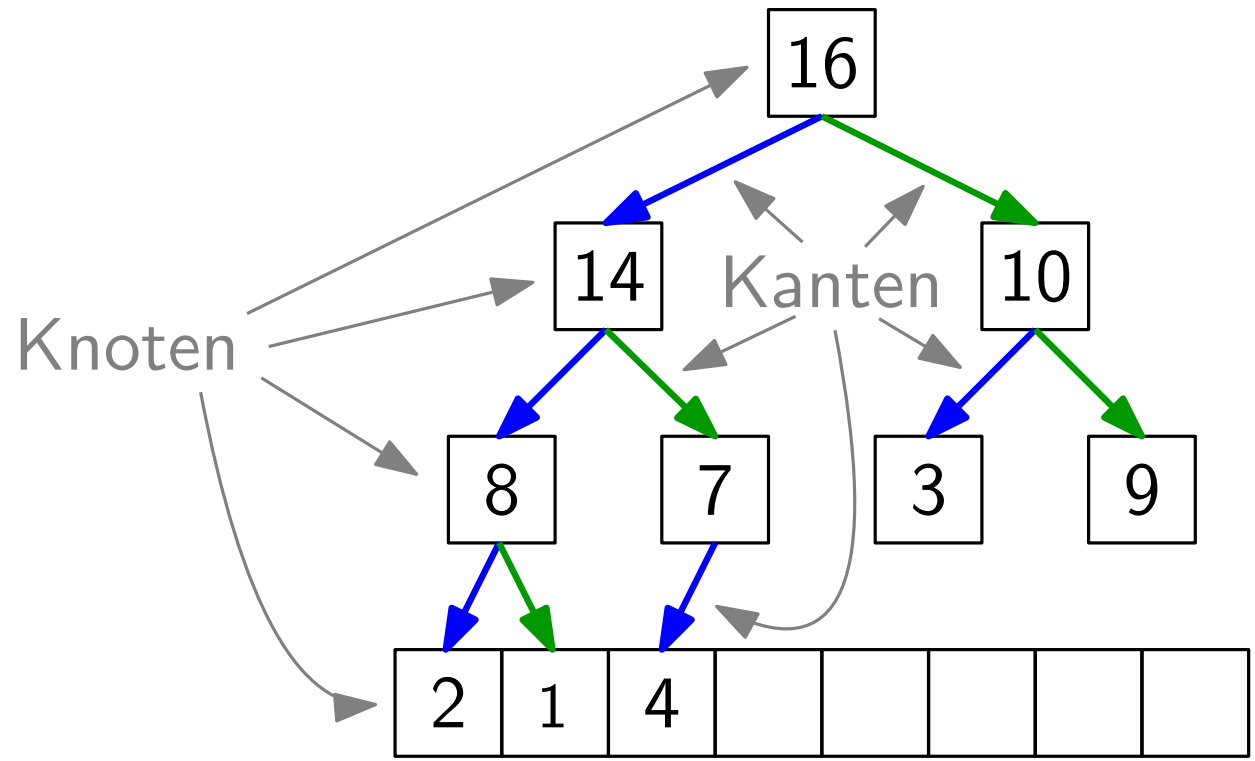
Bäume, gut gepackt



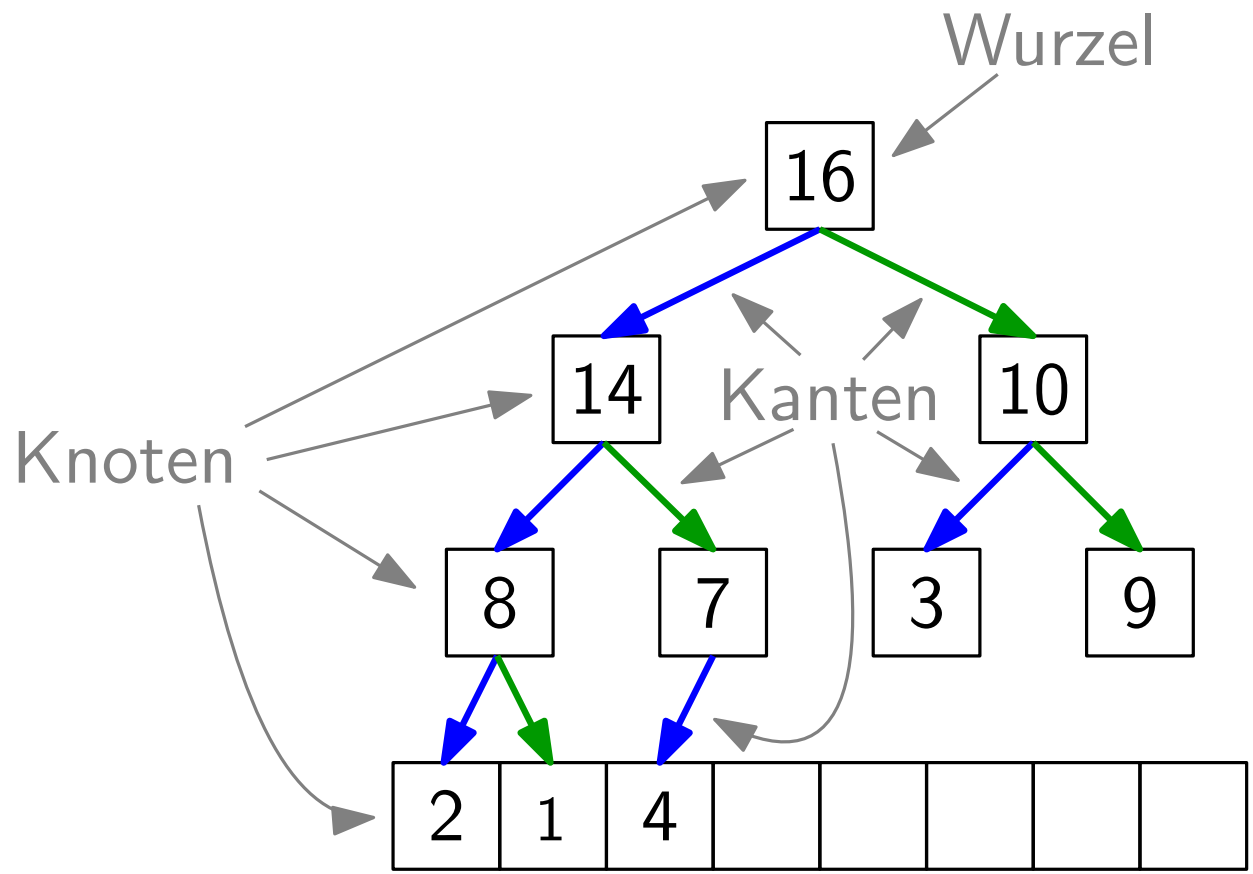
Bäume, gut gepackt



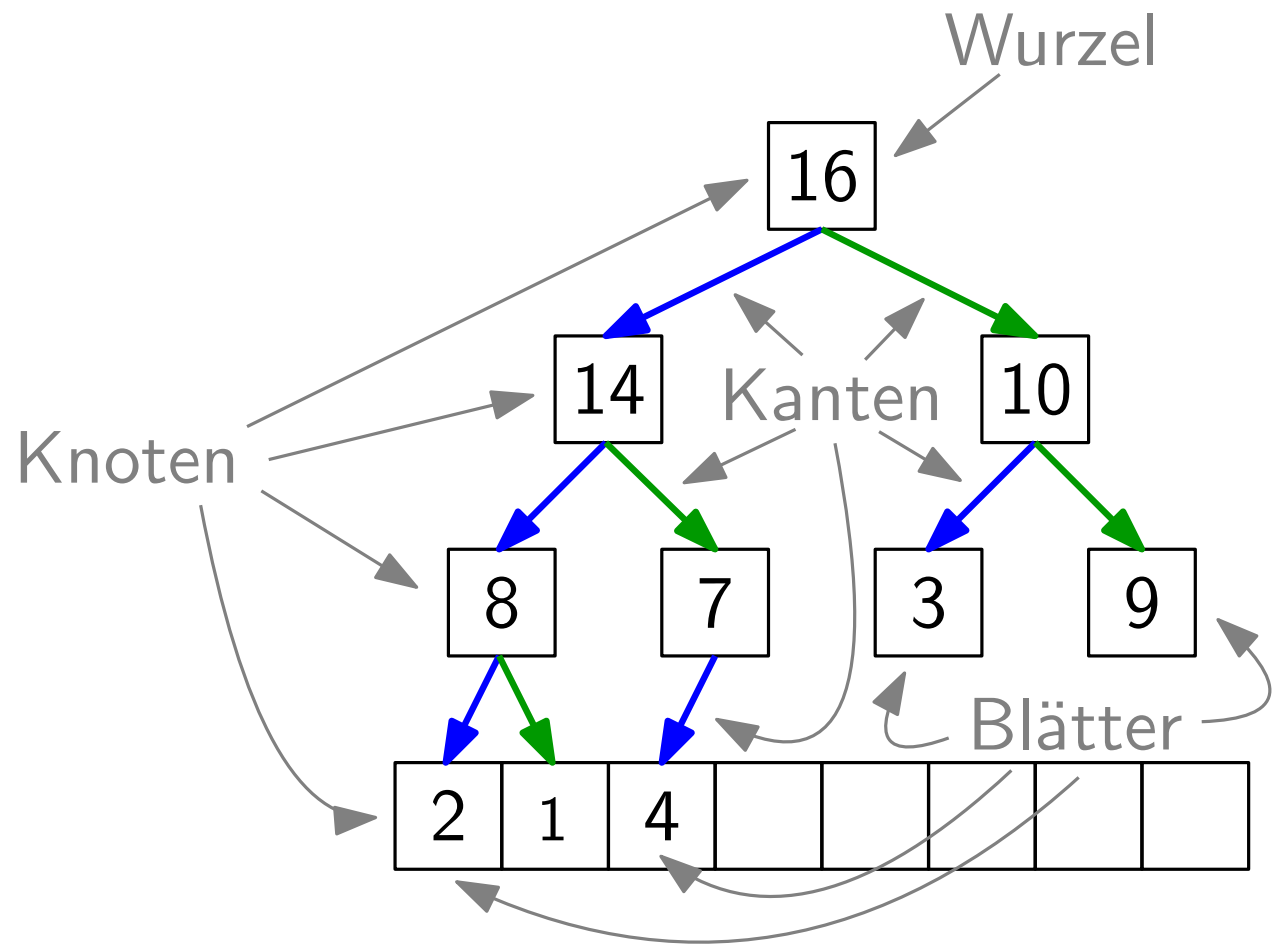
Bäume, gut gepackt



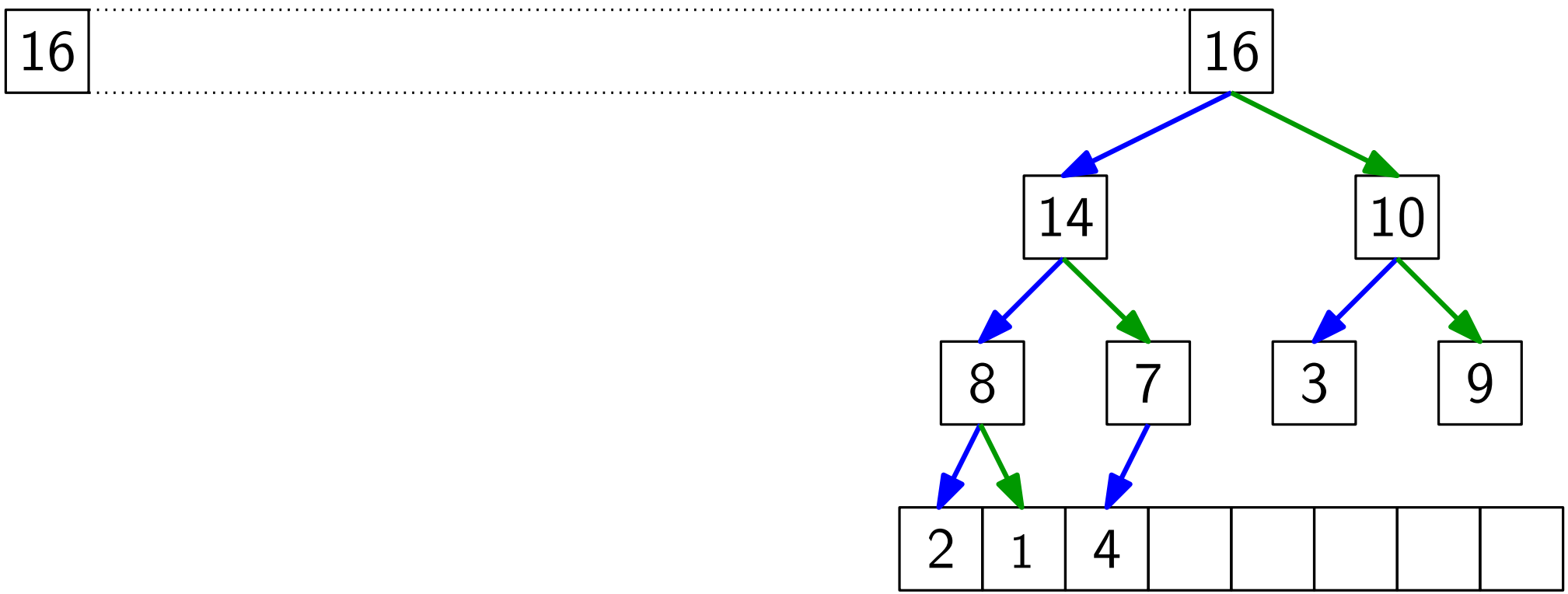
Bäume, gut gepackt



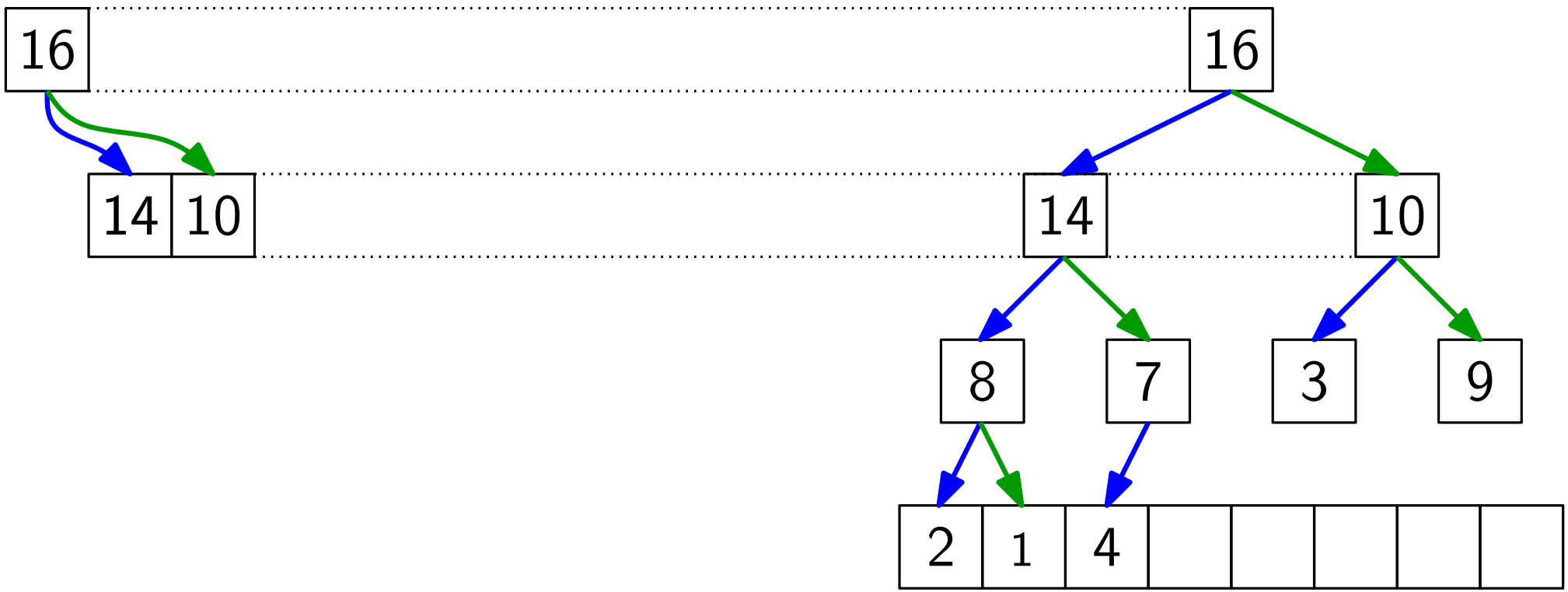
Bäume, gut gepackt



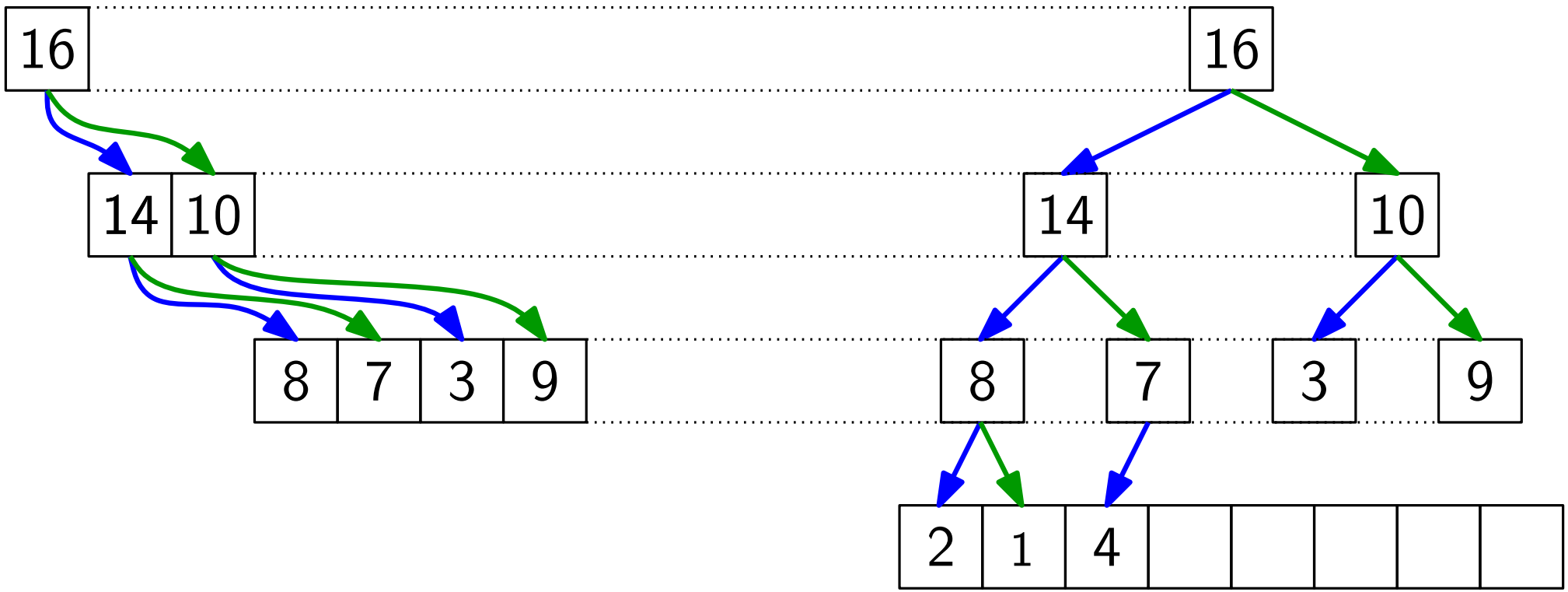
Bäume, gut gepackt



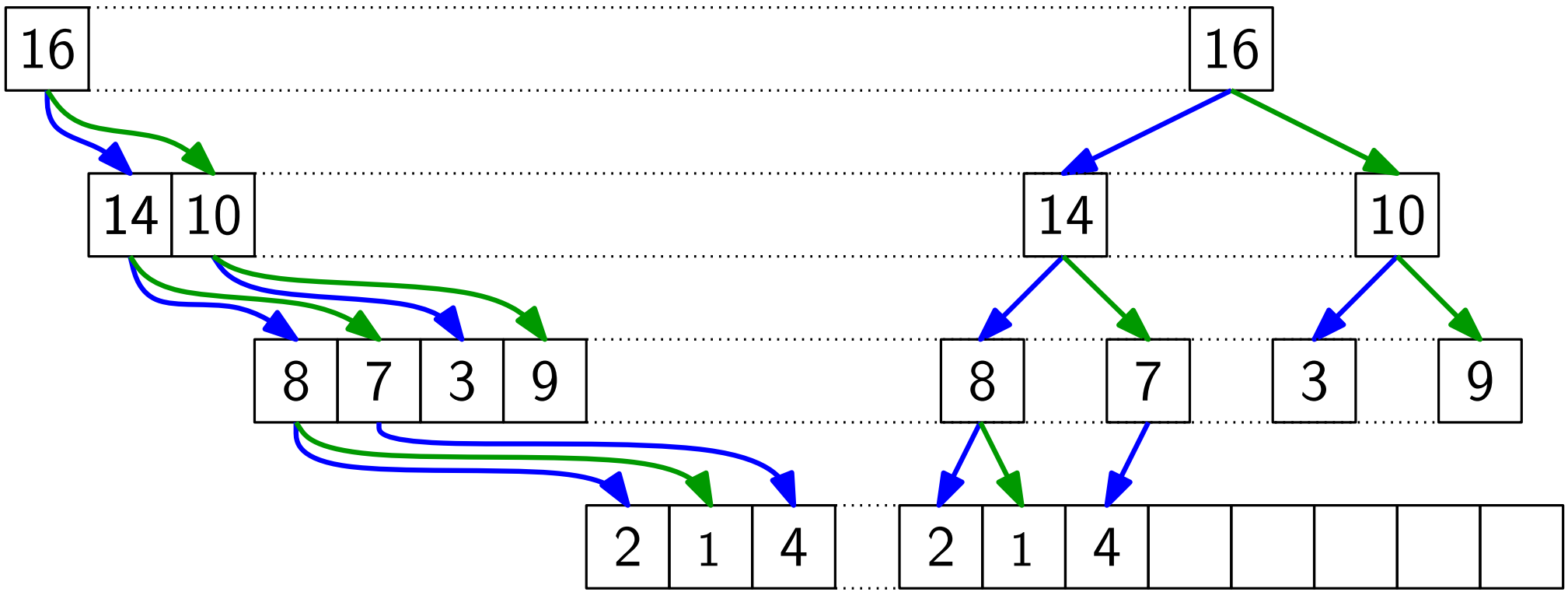
Bäume, gut gepackt



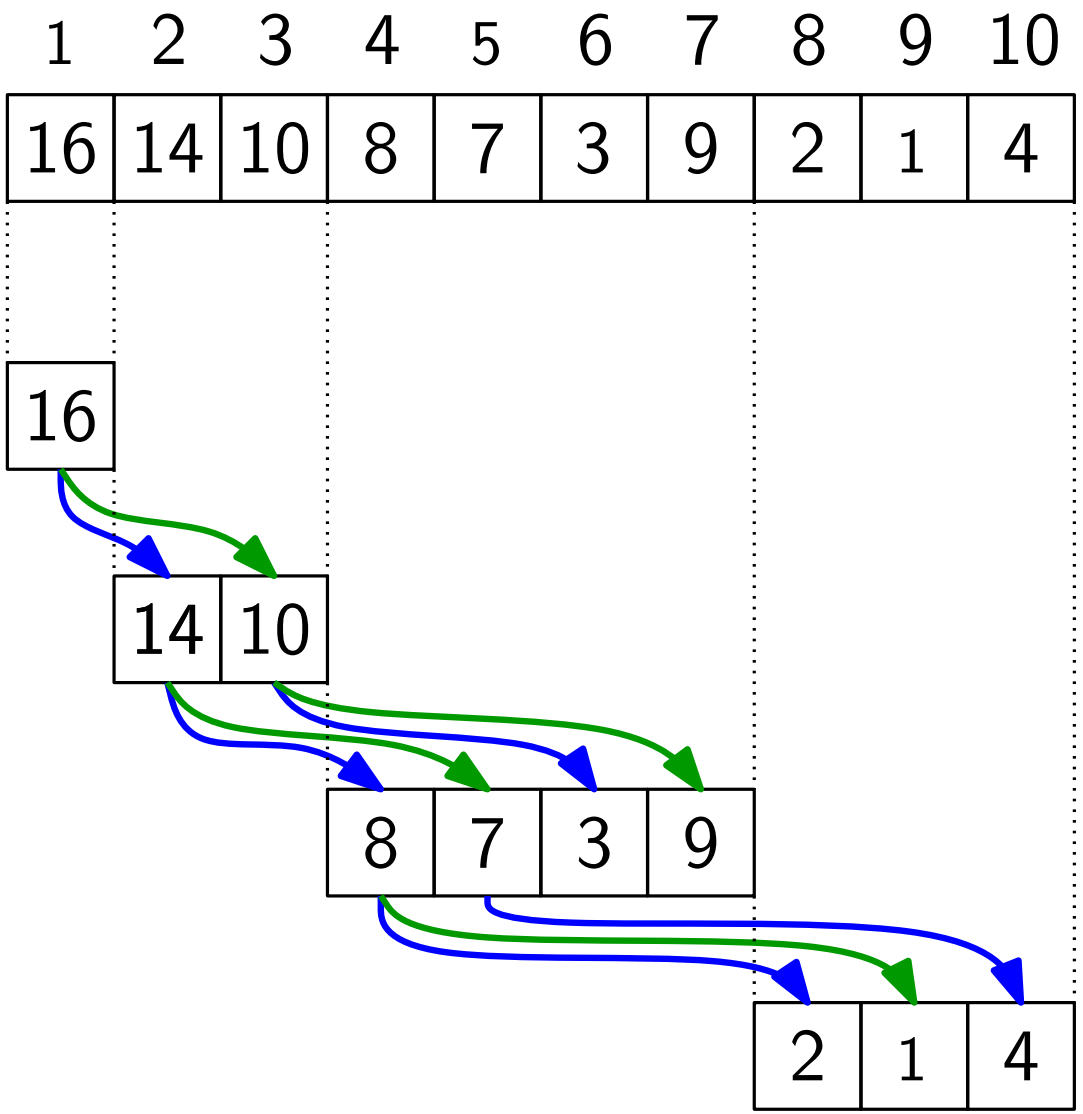
Bäume, gut gepackt

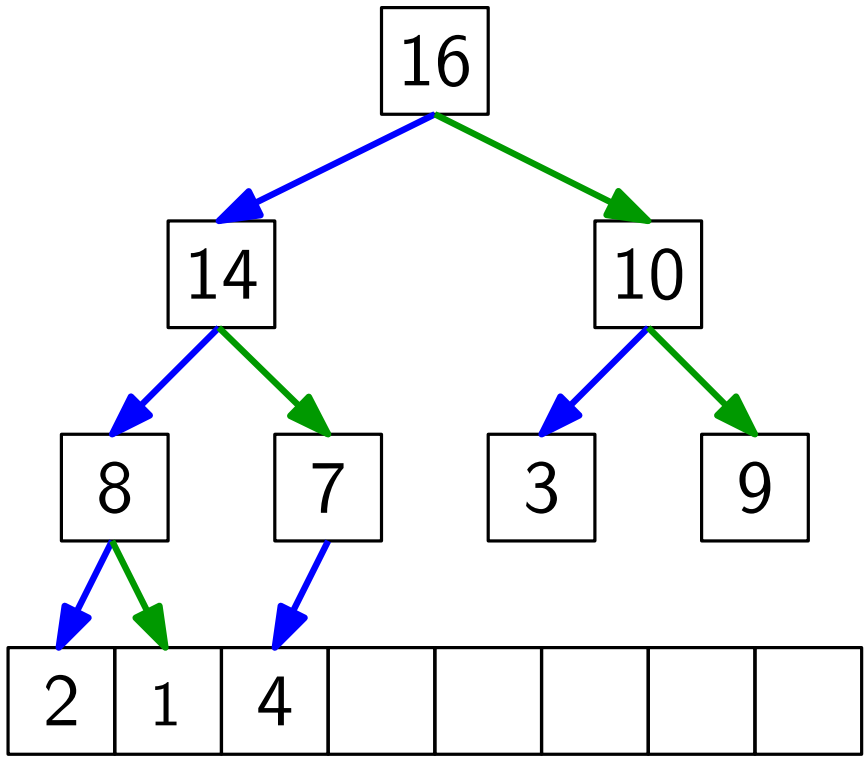


Bäume, gut gepackt

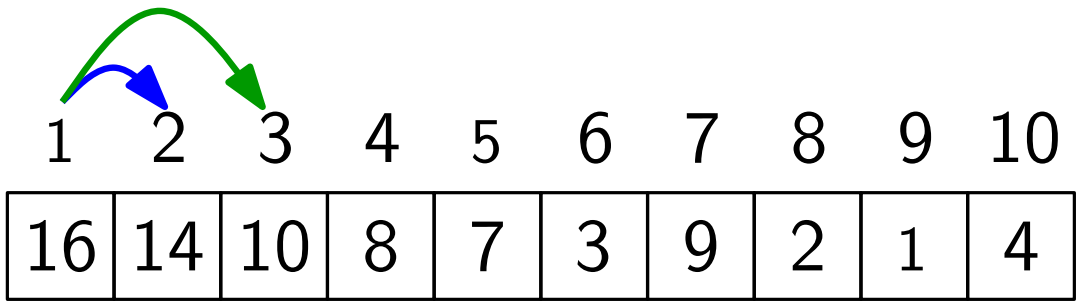


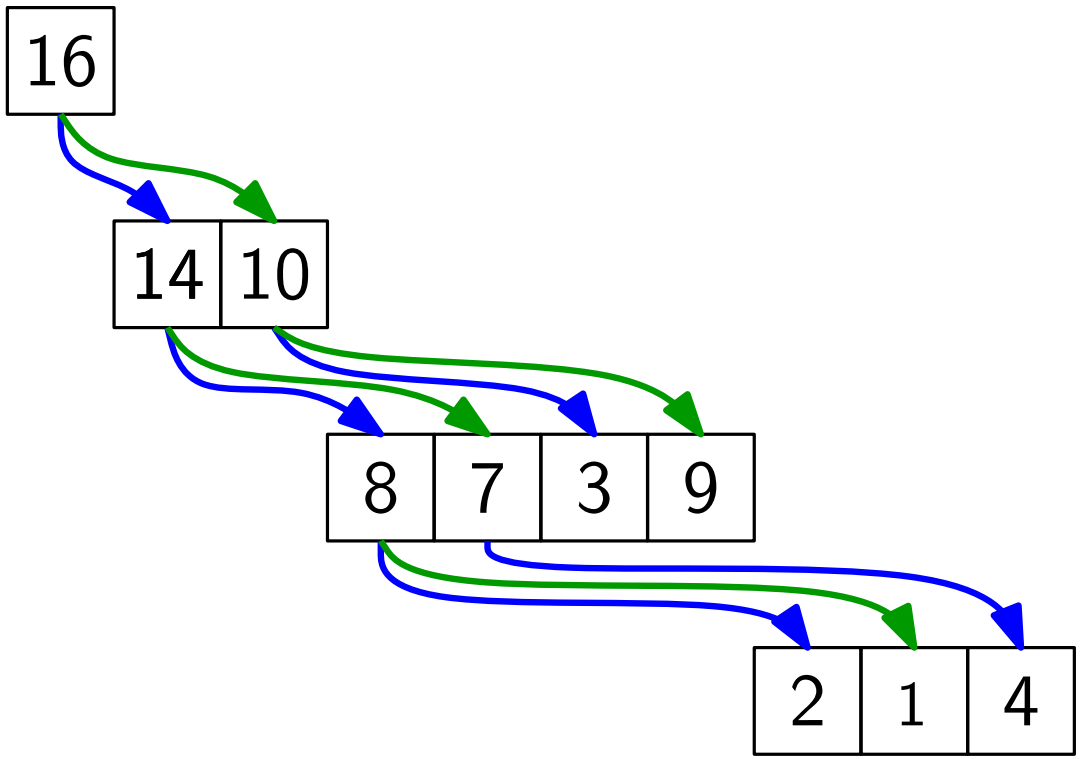
Bäume, gut gepackt

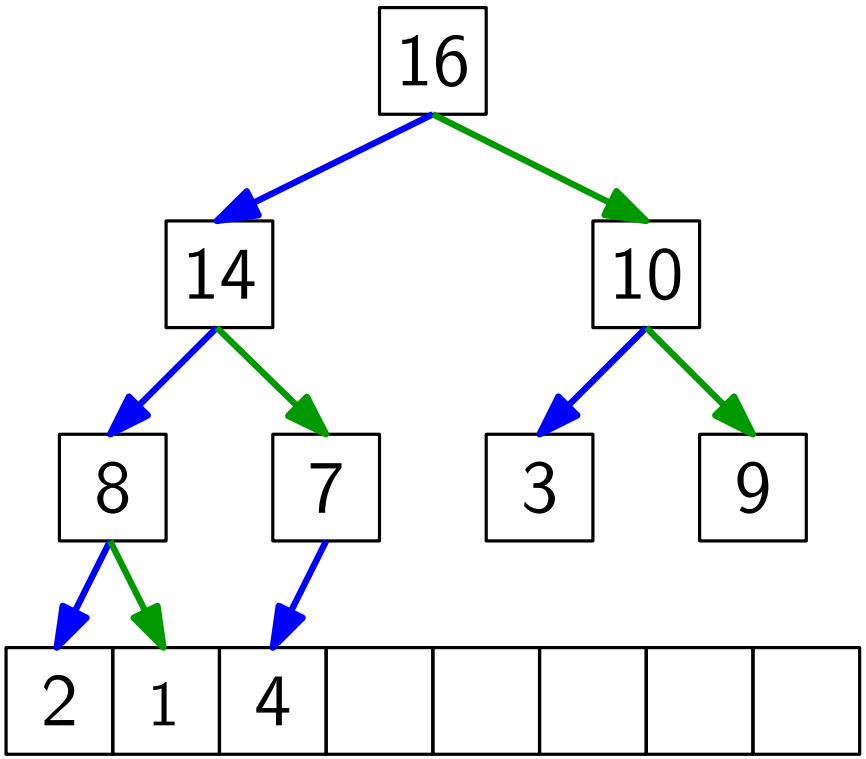




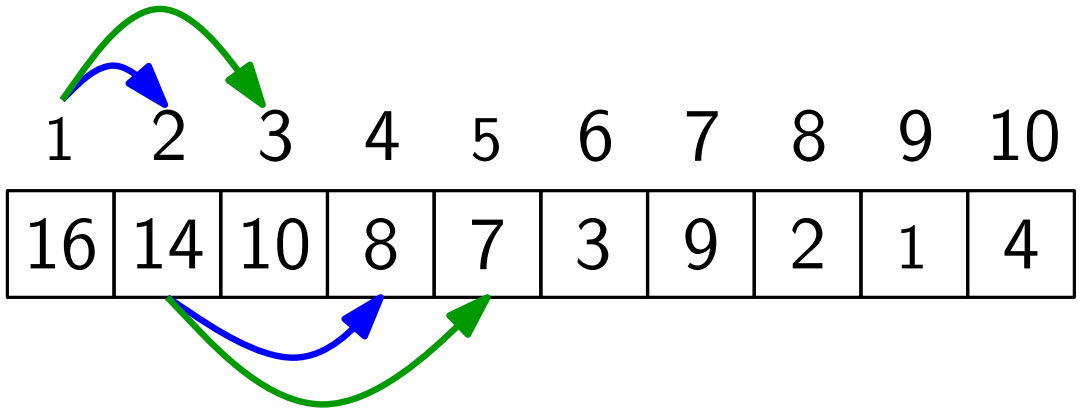
Bäume, gut gepackt

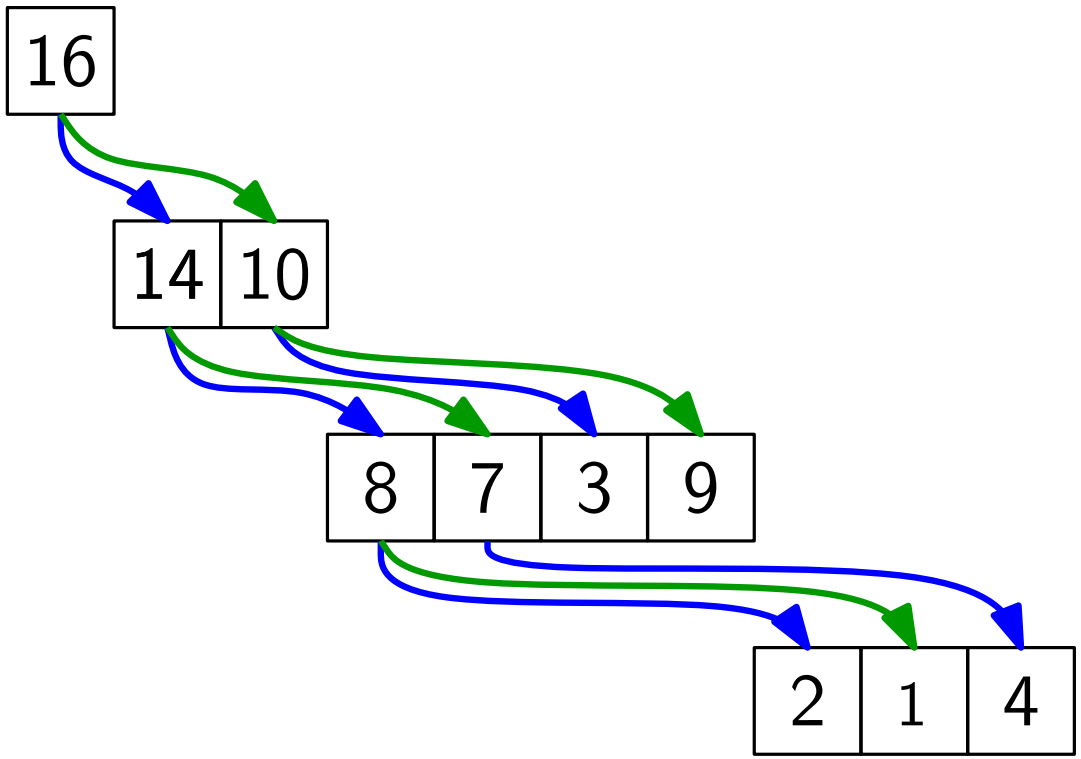


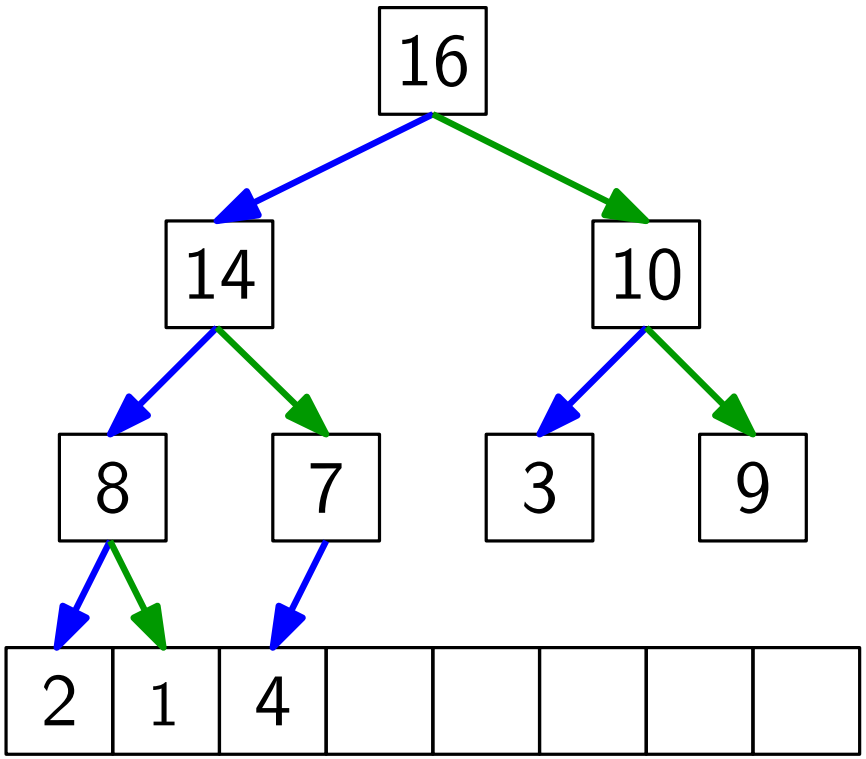




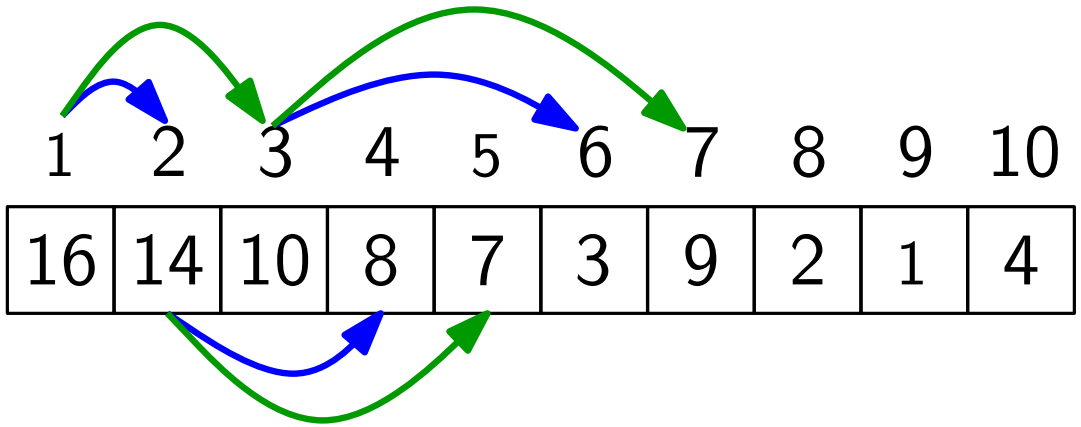
Bäume, gut gepackt

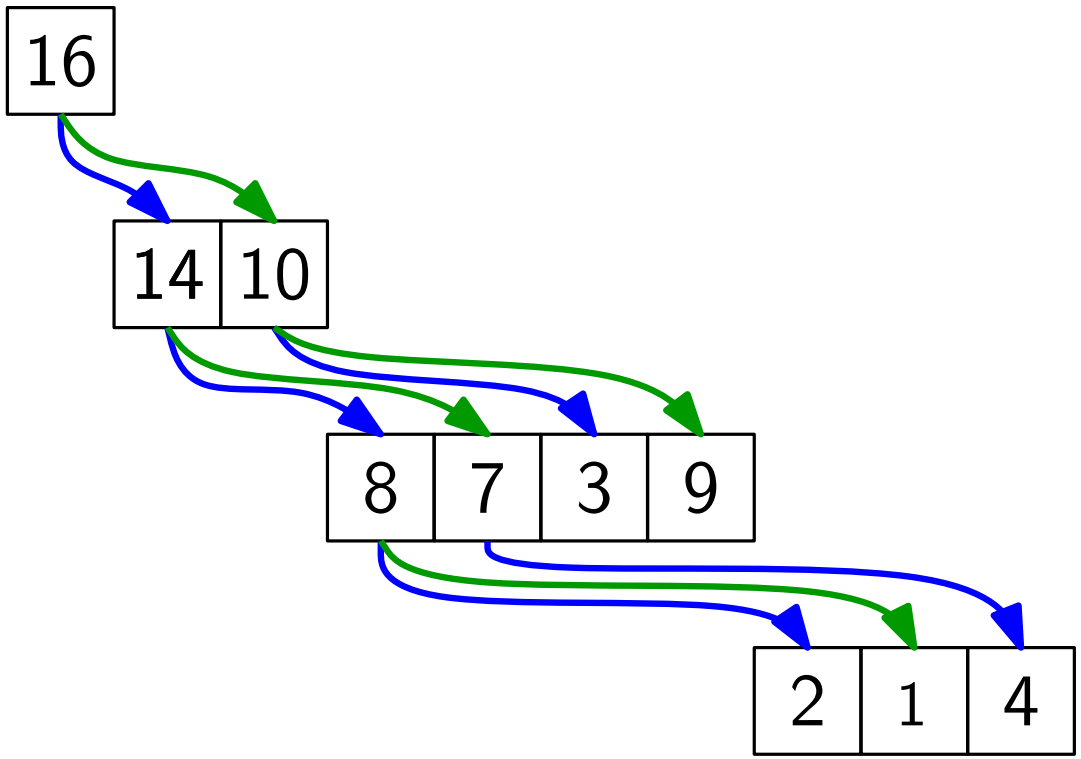


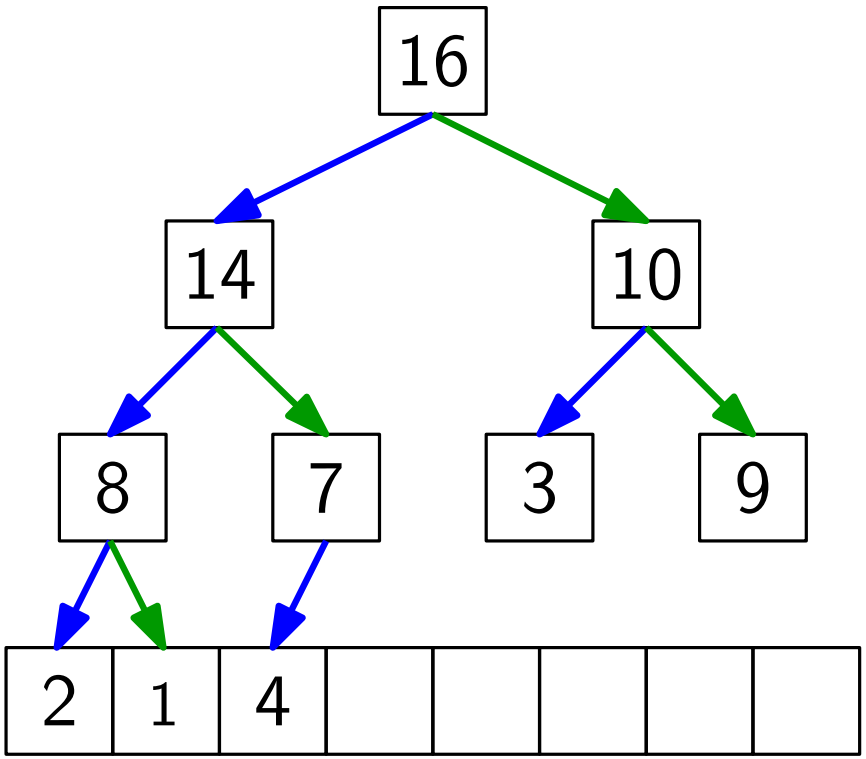




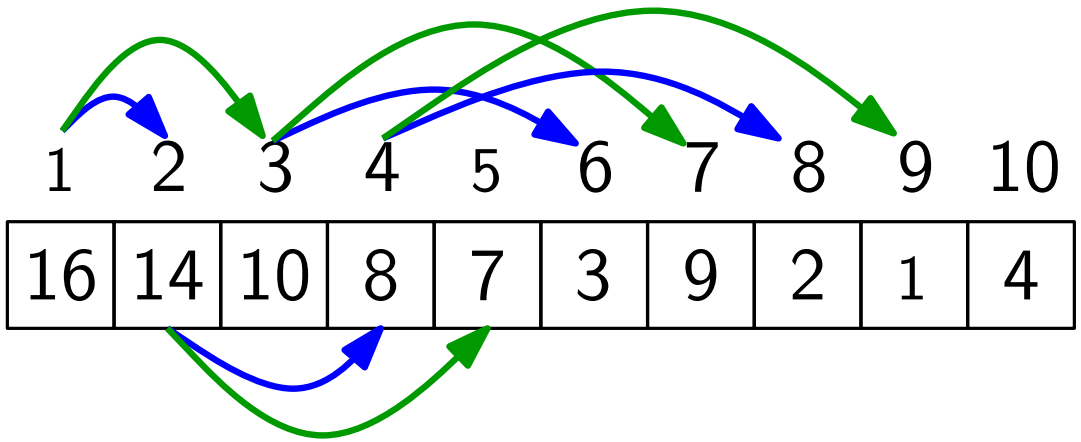
Bäume, gut gepackt

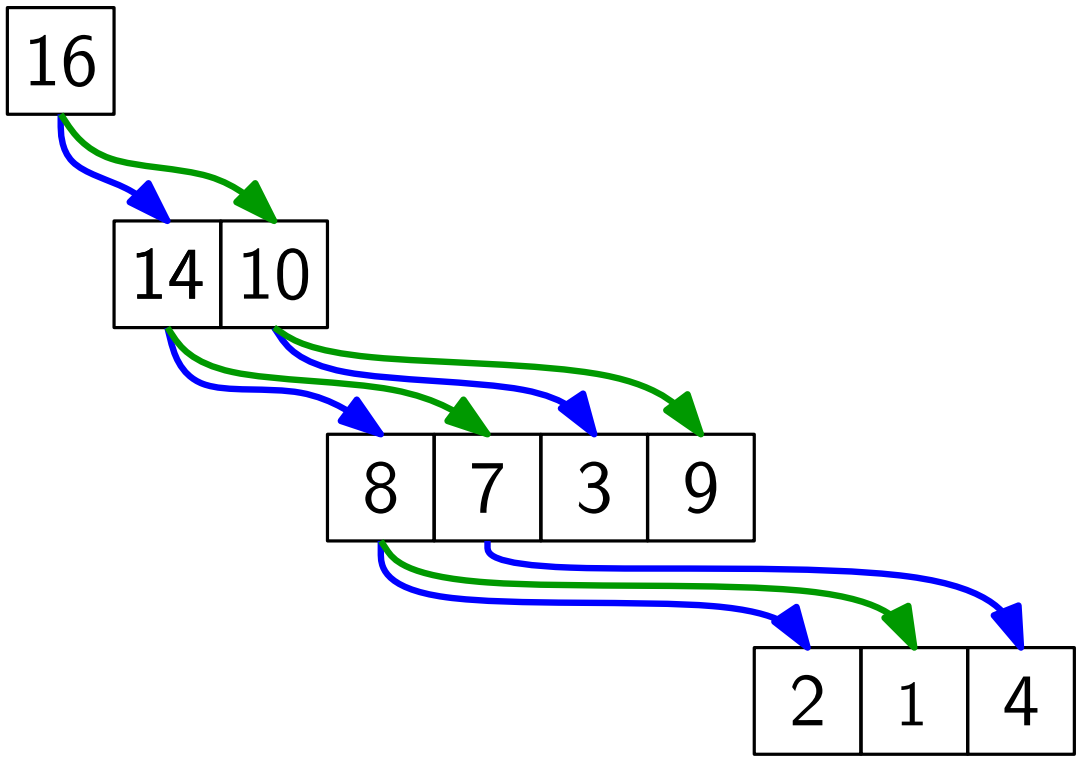


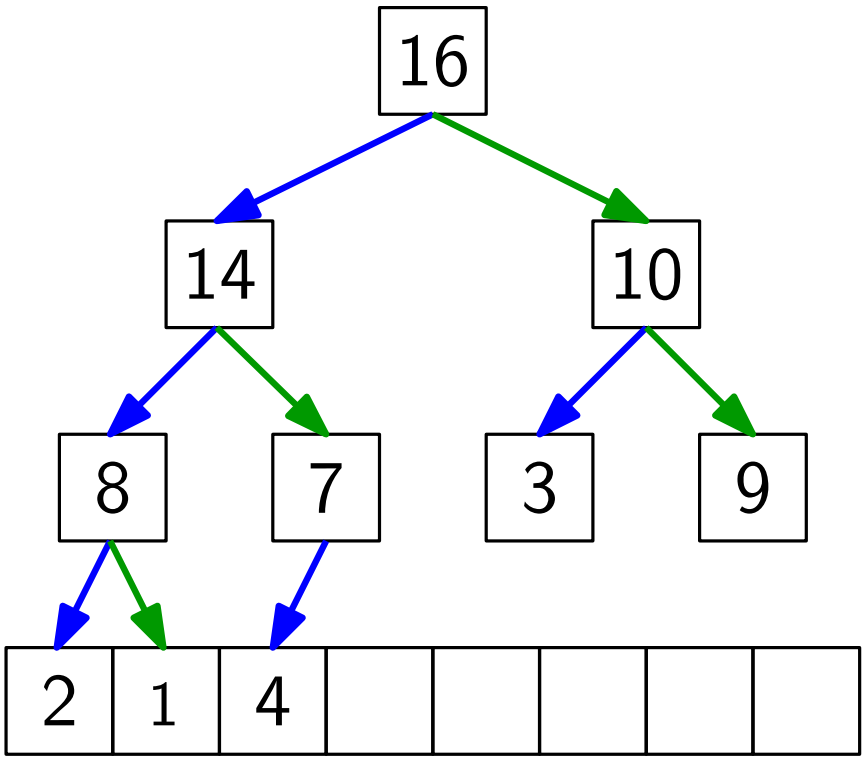




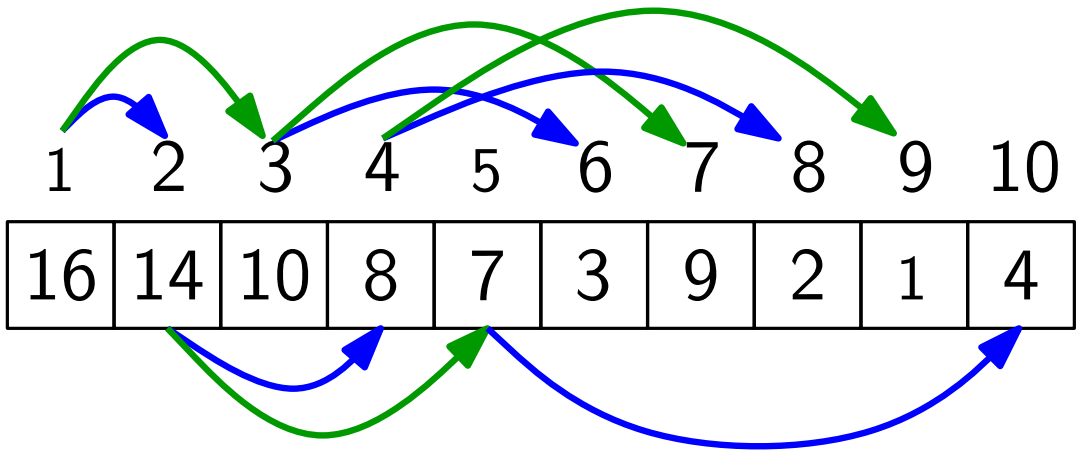
Bäume, gut gepackt

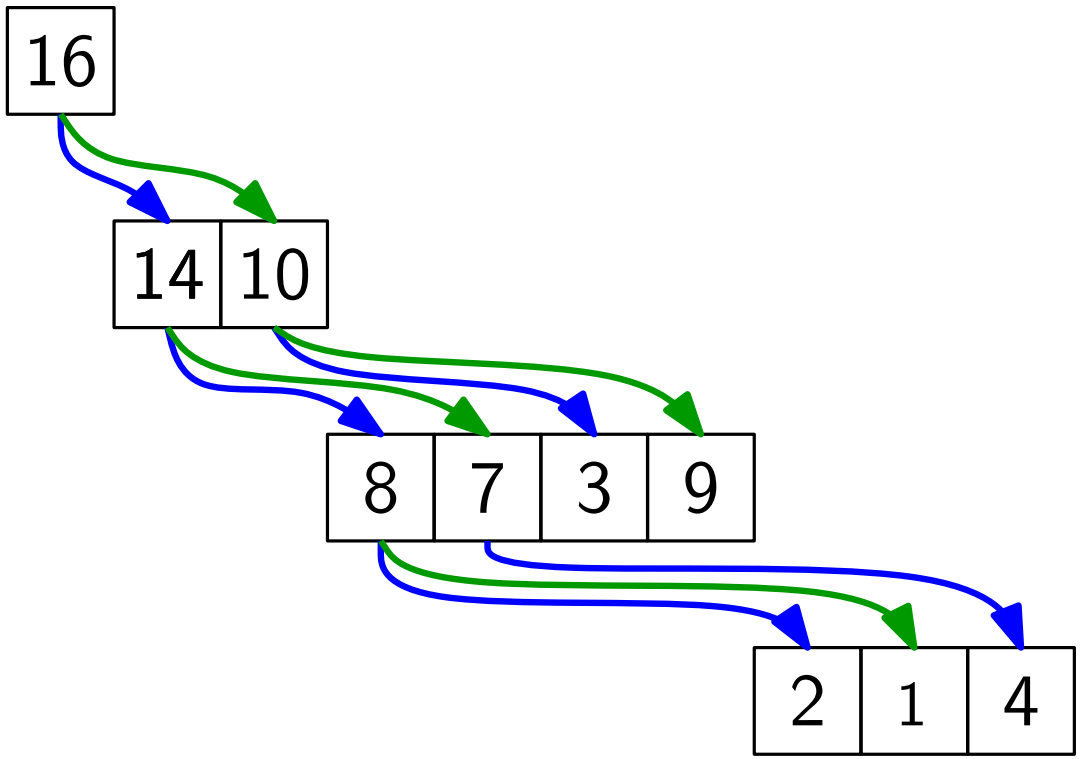


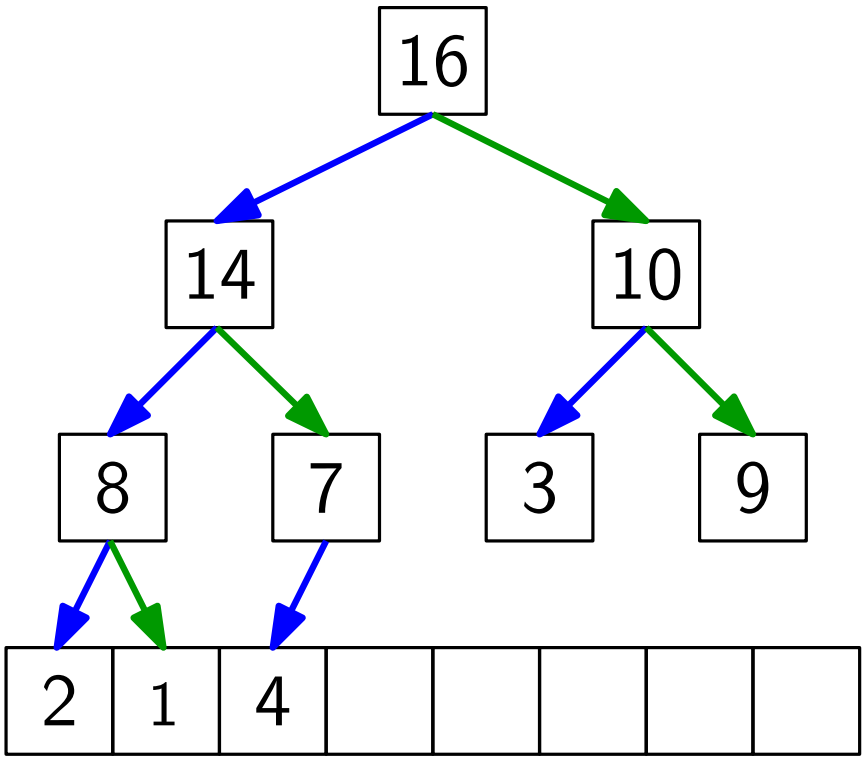




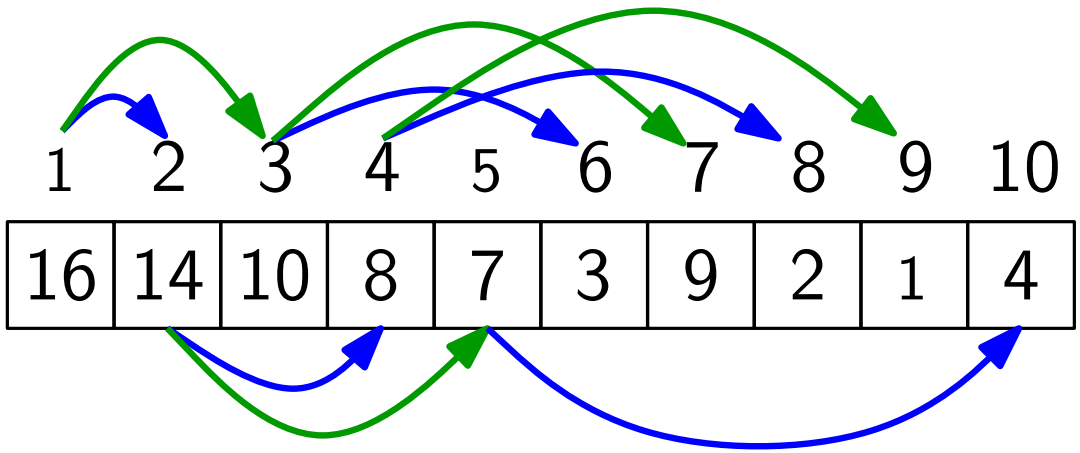
Bäume, gut gepackt



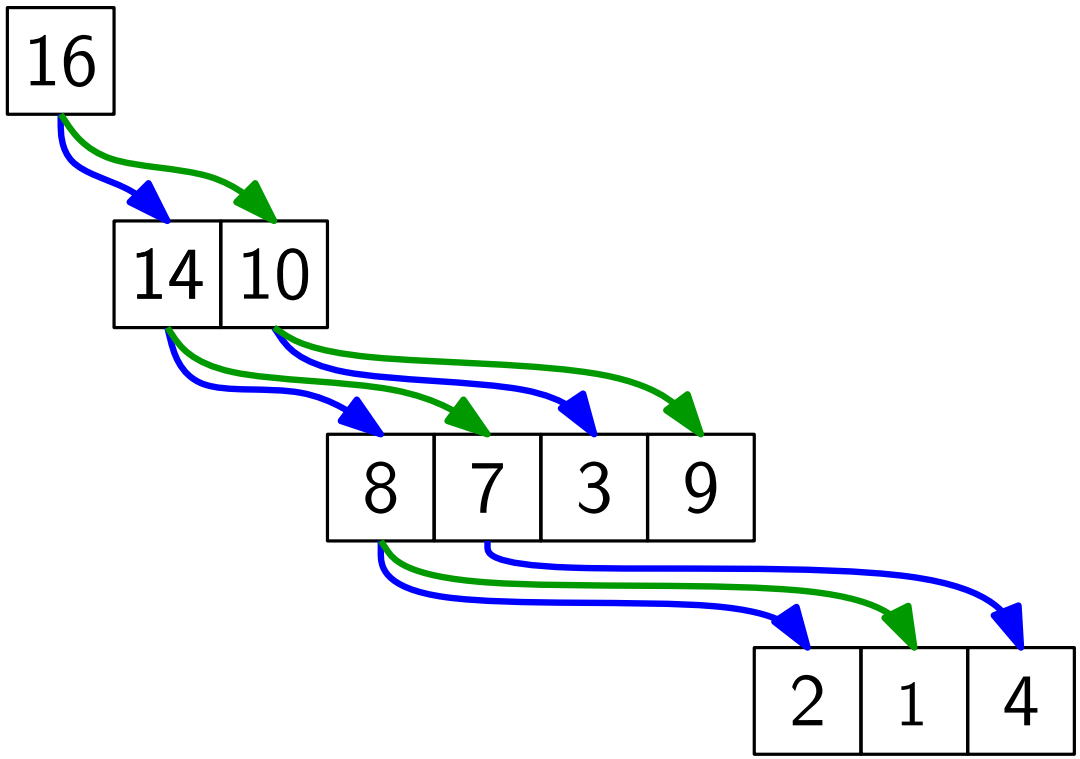


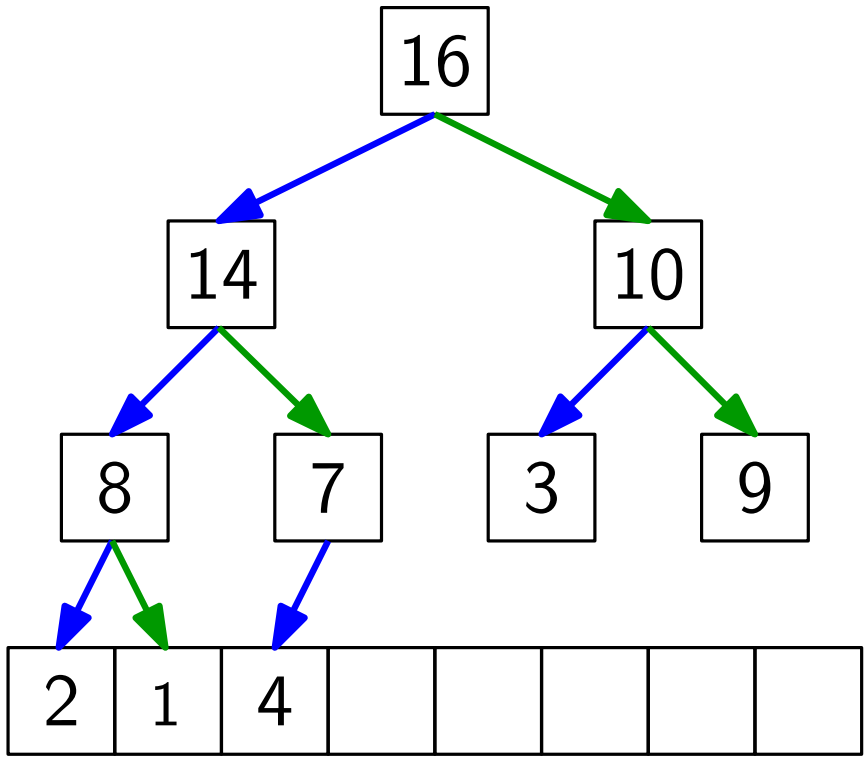


Bäume, gut gepackt

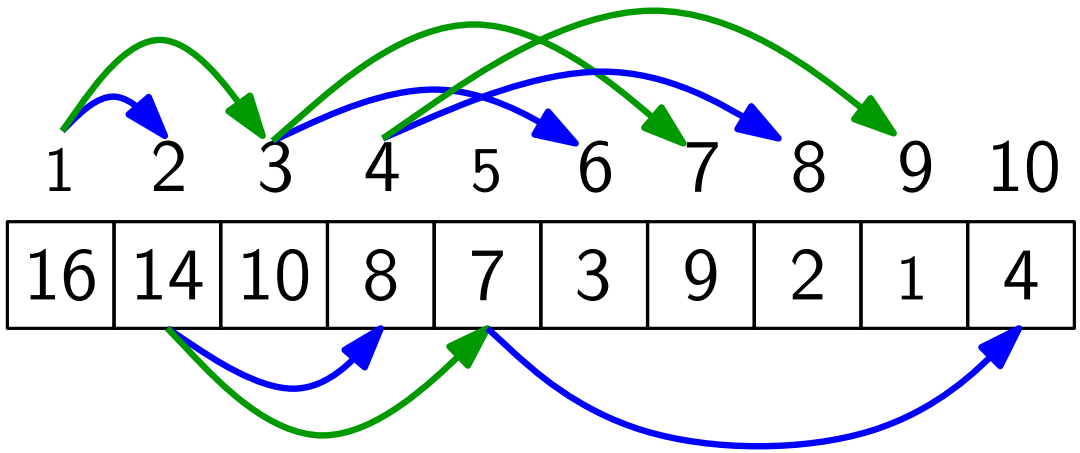


Pfeile implementieren:





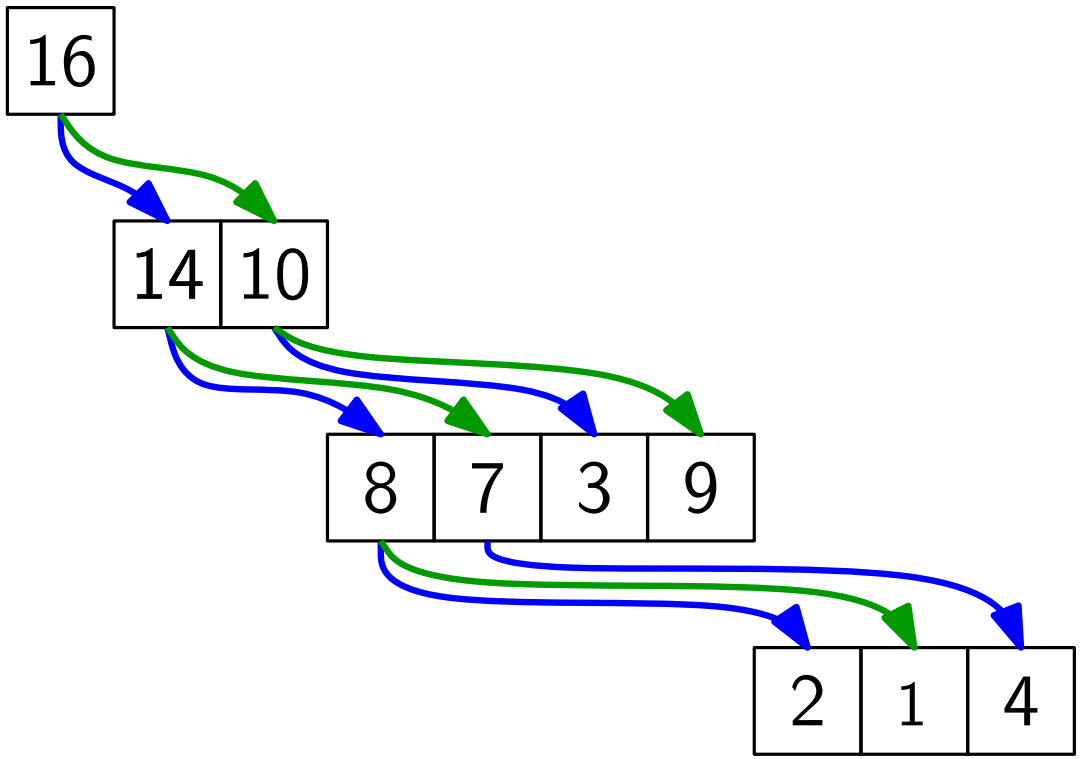
Bäume, gut gepackt

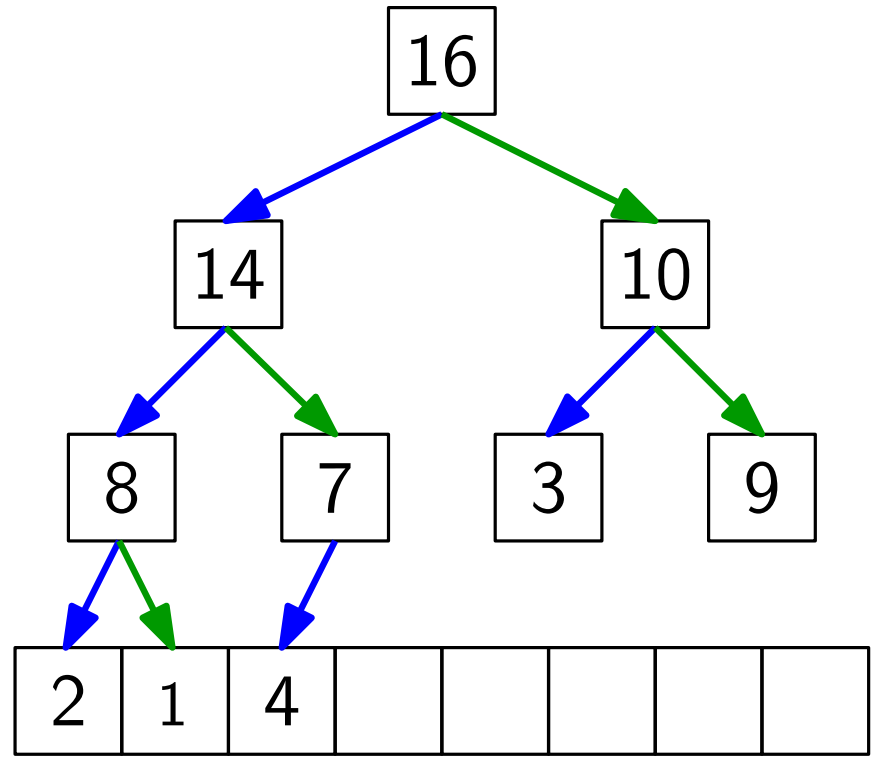


Pfeile implementieren:

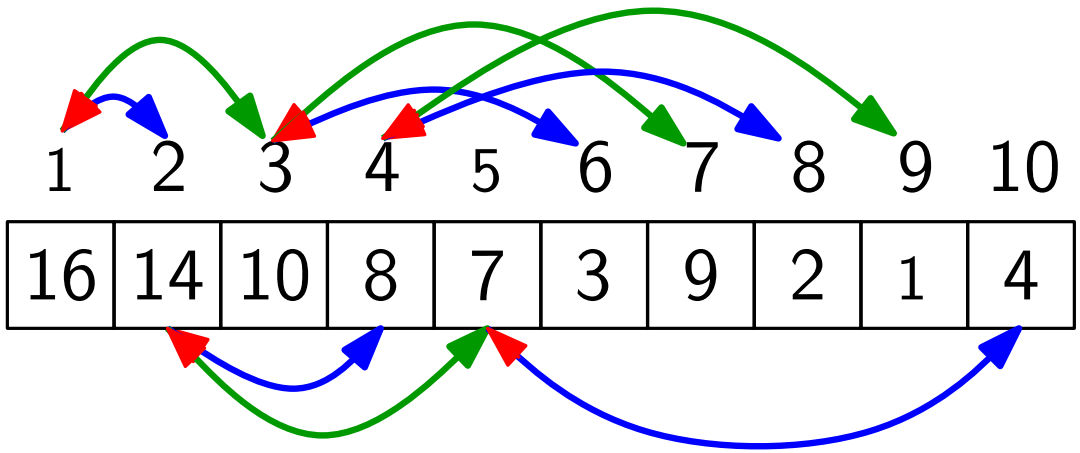
left(index i) **return** ...

right(index i) **return** ...





Bäume, gut gepackt

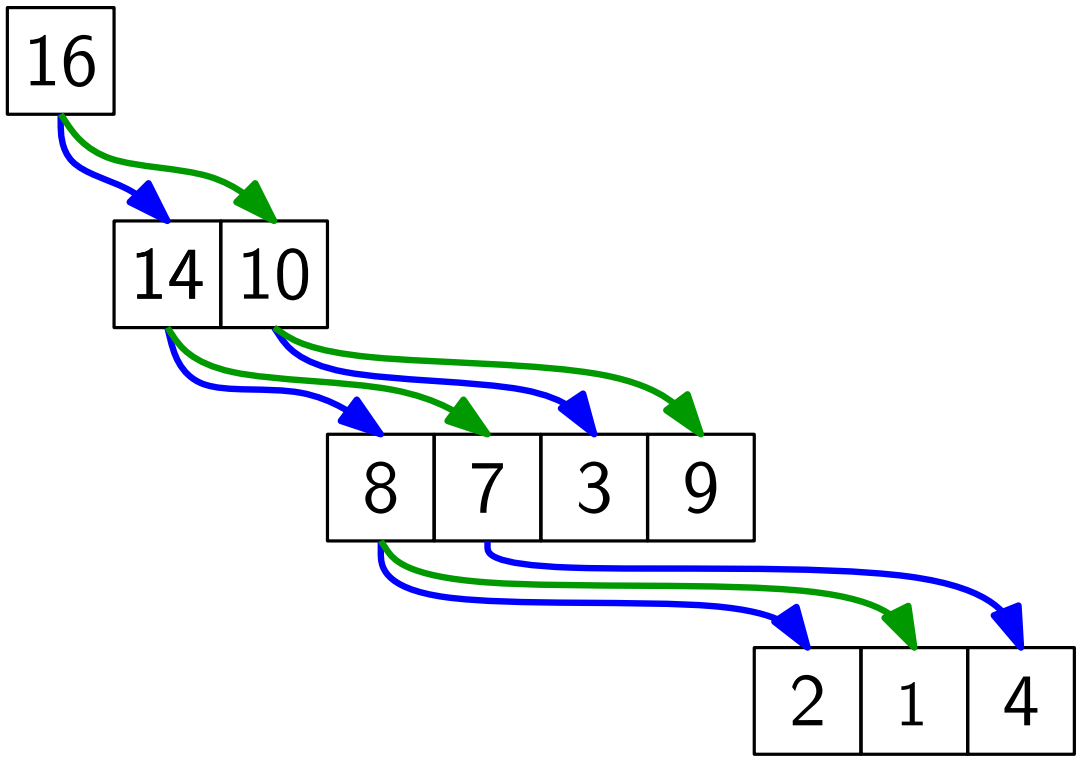


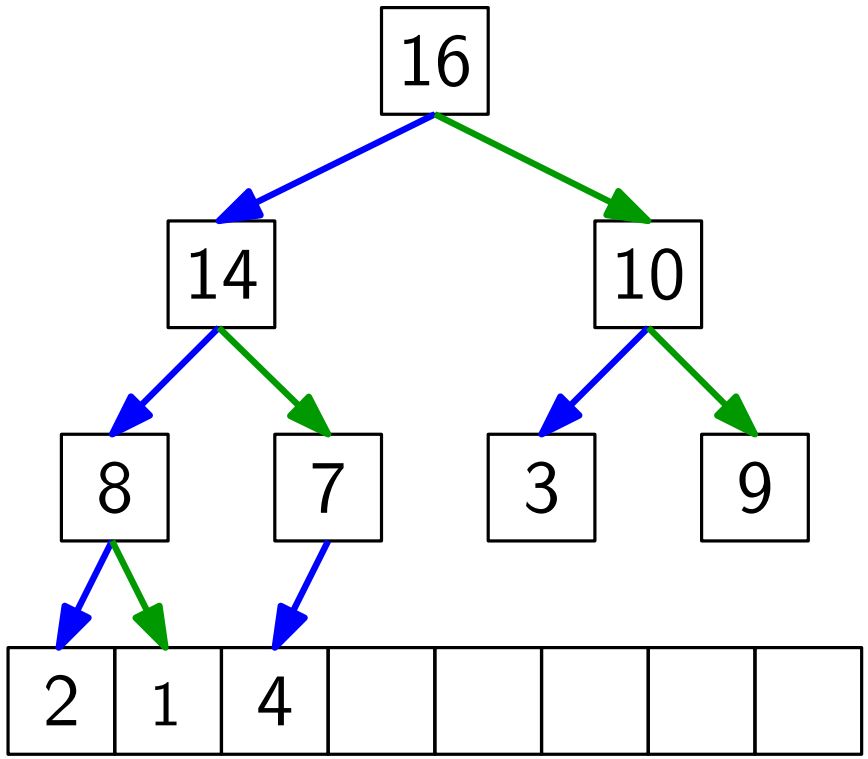
Pfeile implementieren:

left(index i) **return** ...

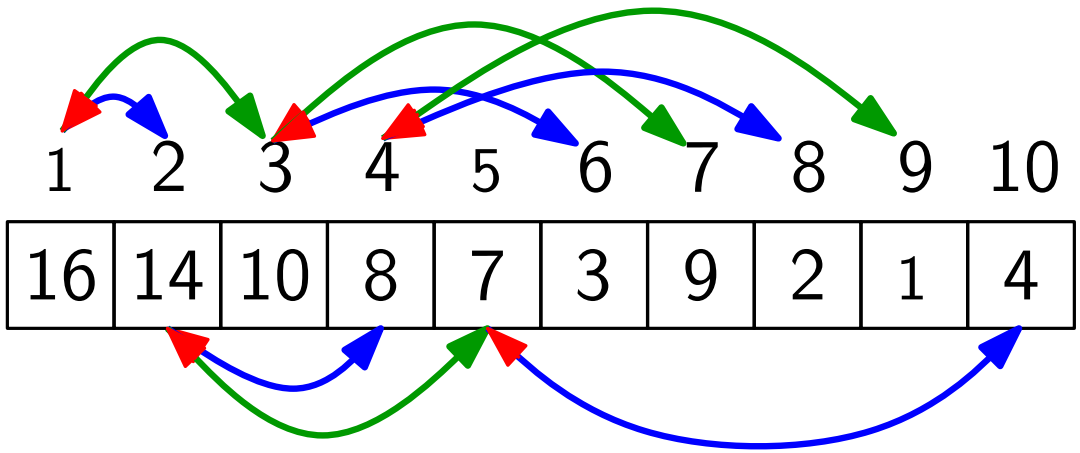
right(index i) **return** ...

parent(index i) **return** ...





Bäume, gut gepackt

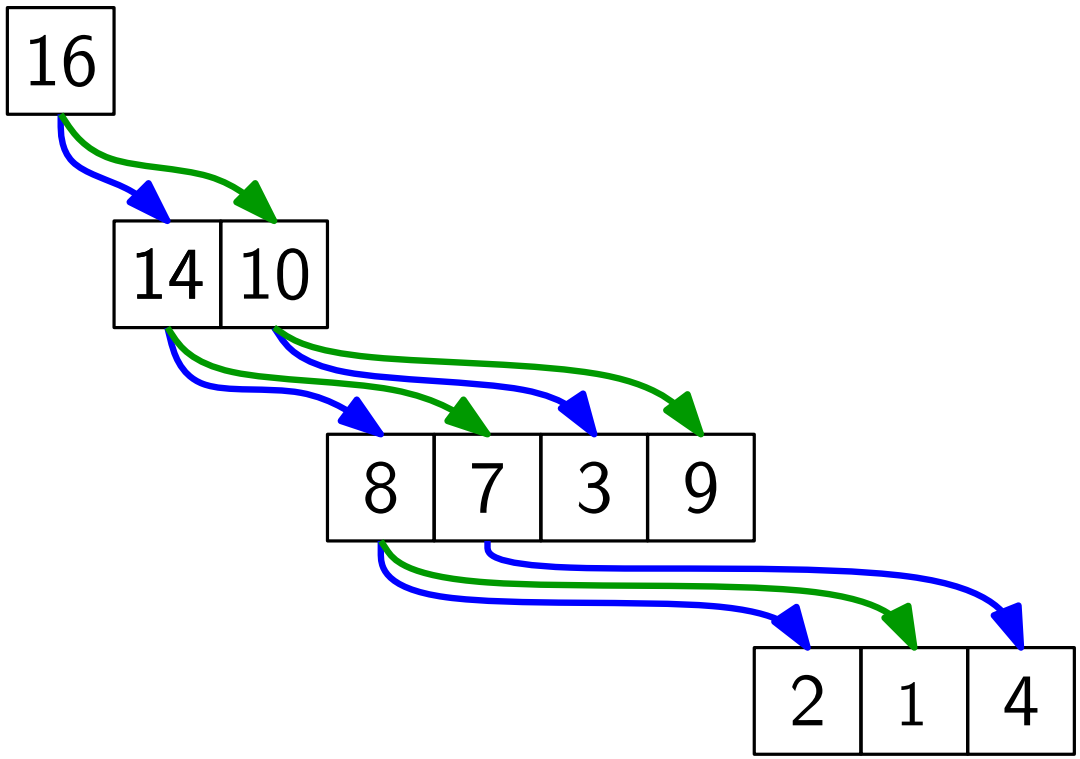


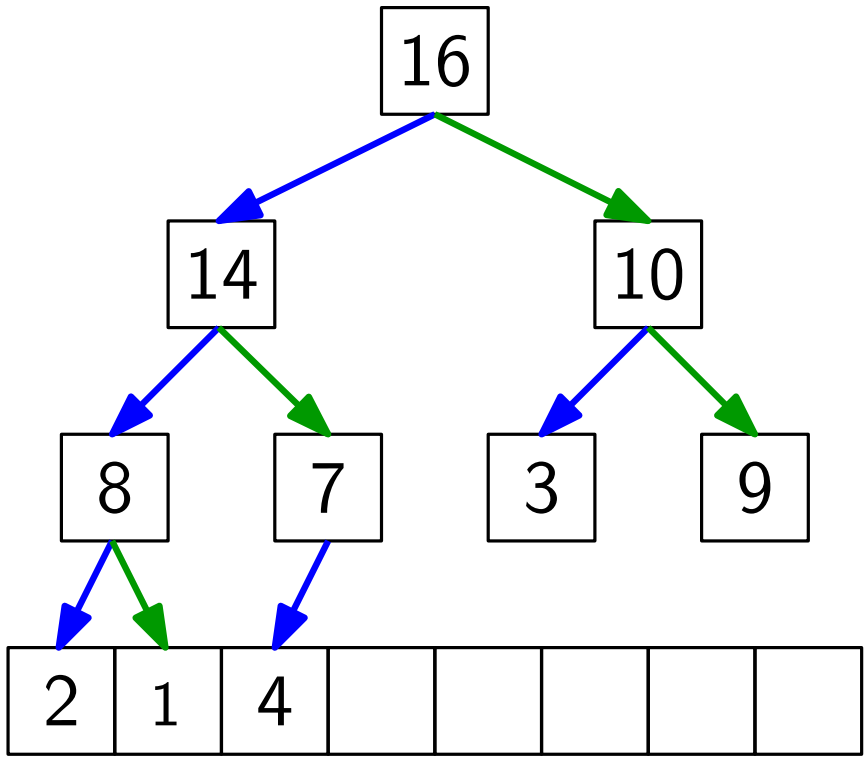
Pfeile implementieren:

left(index i) **return** $2i$

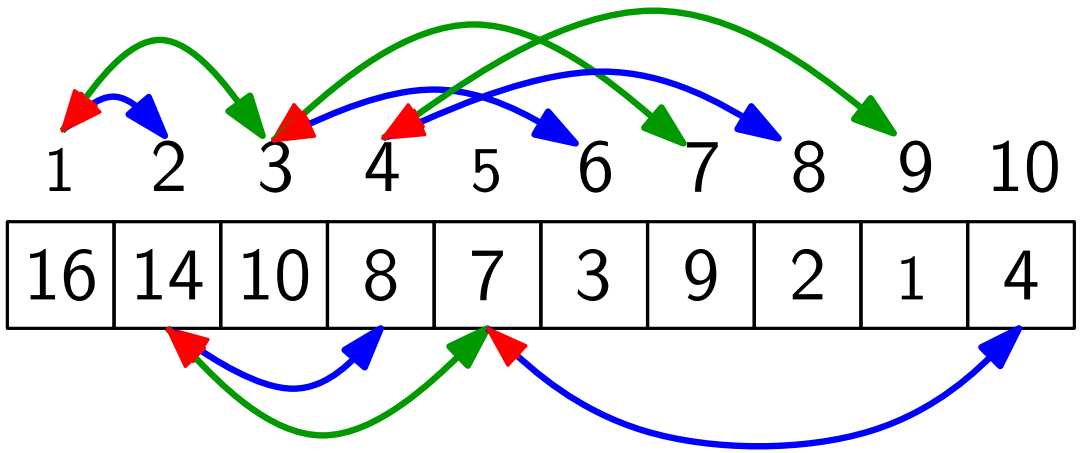
right(index i) **return** . . .

parent(index i) **return** . . .





Bäume, gut gepackt

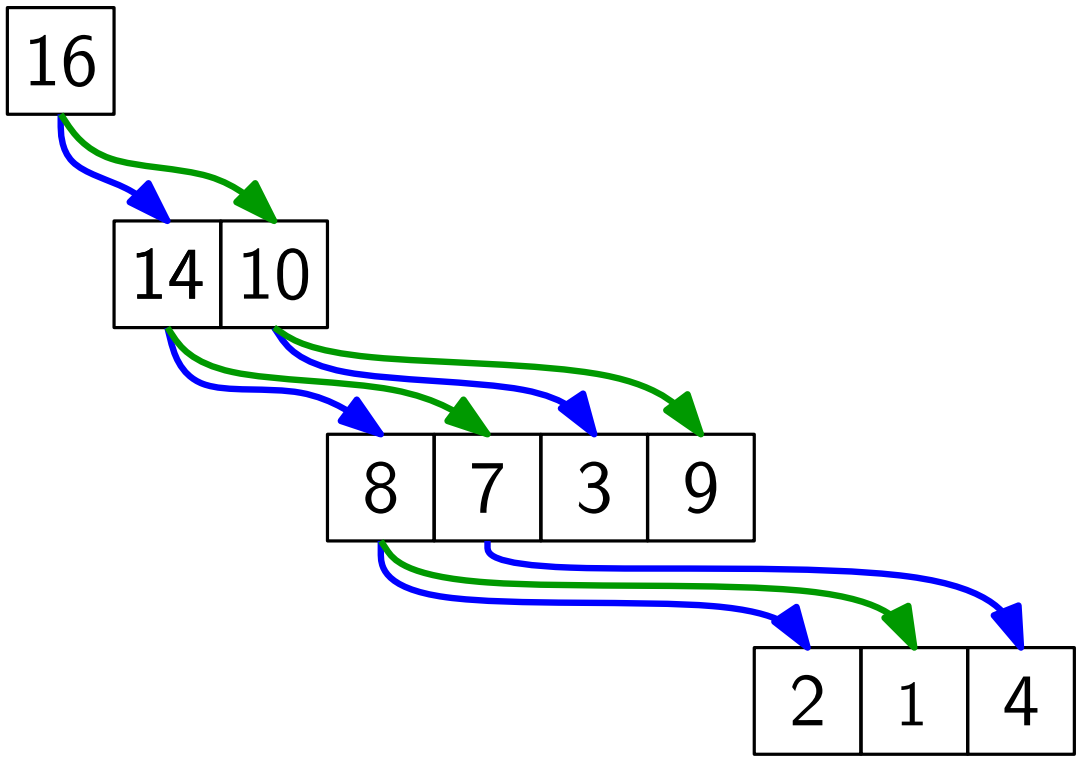


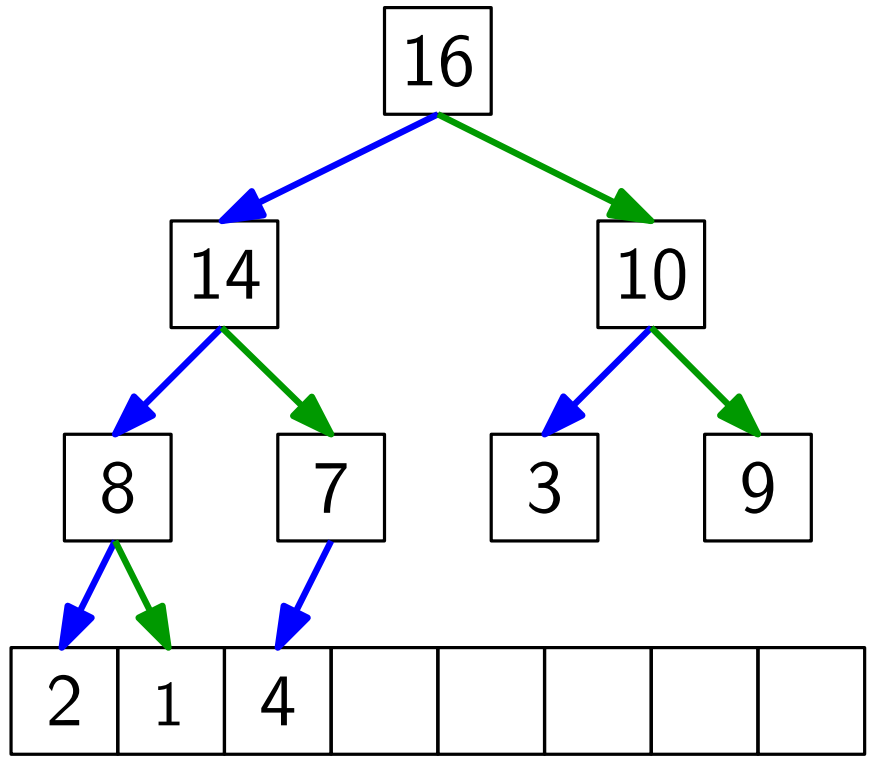
Pfeile implementieren:

left(index i) **return** $2i$

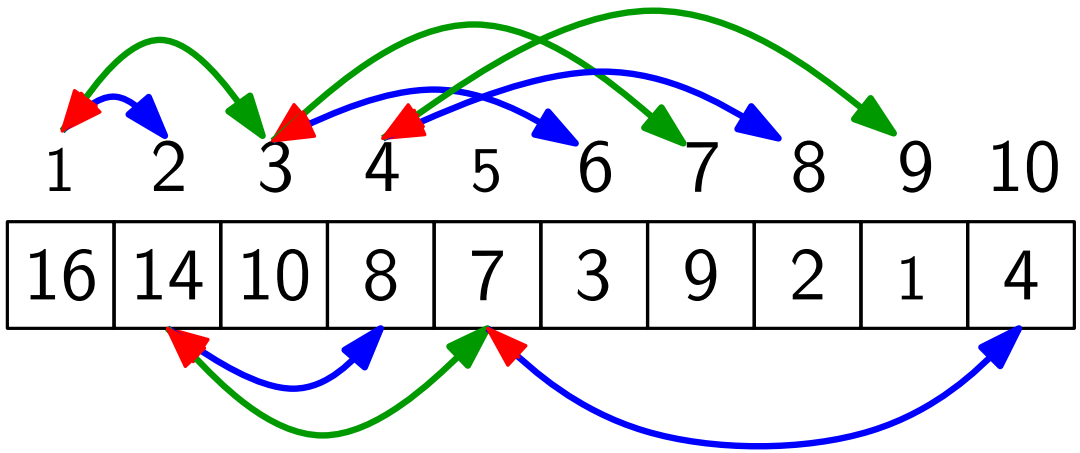
right(index i) **return** $2i + 1$

parent(index i) **return** . . .





Bäume, gut gepackt

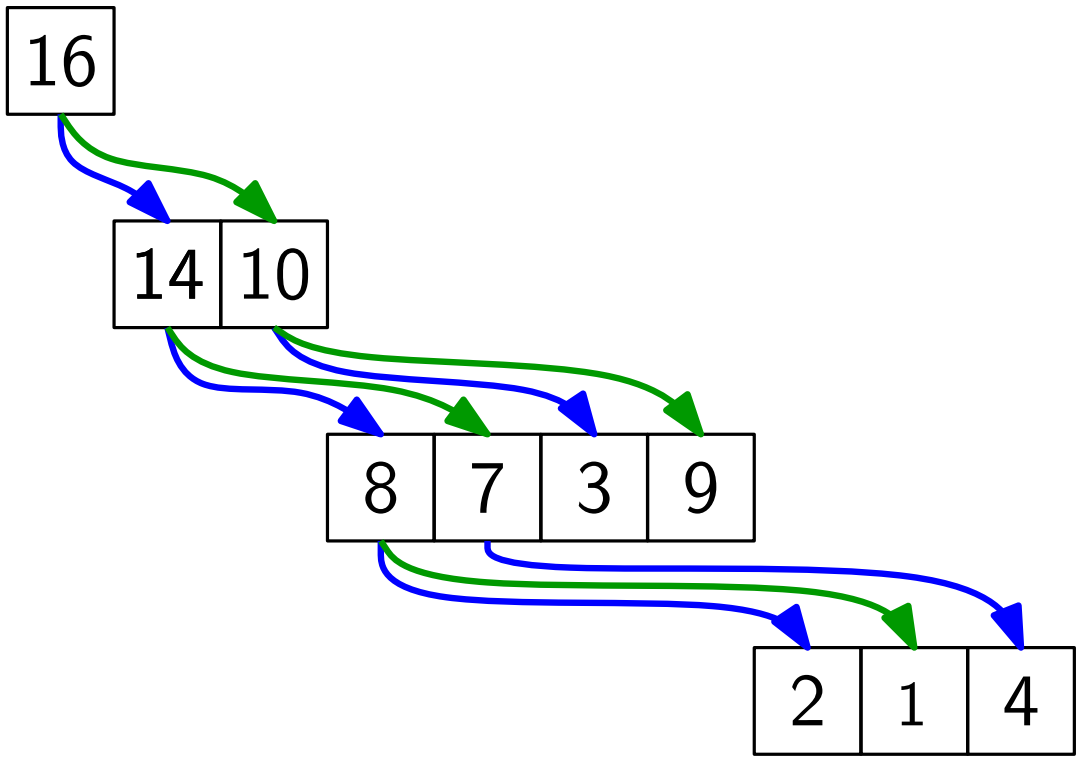


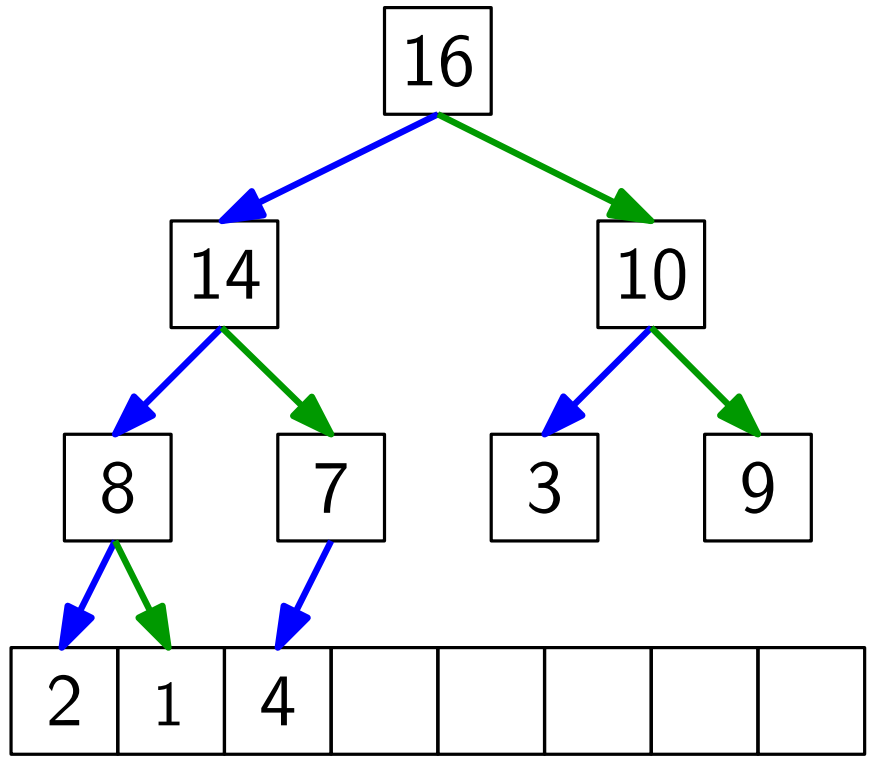
Pfeile implementieren:

left(index i) **return** $2i$

right(index i) **return** $2i + 1$

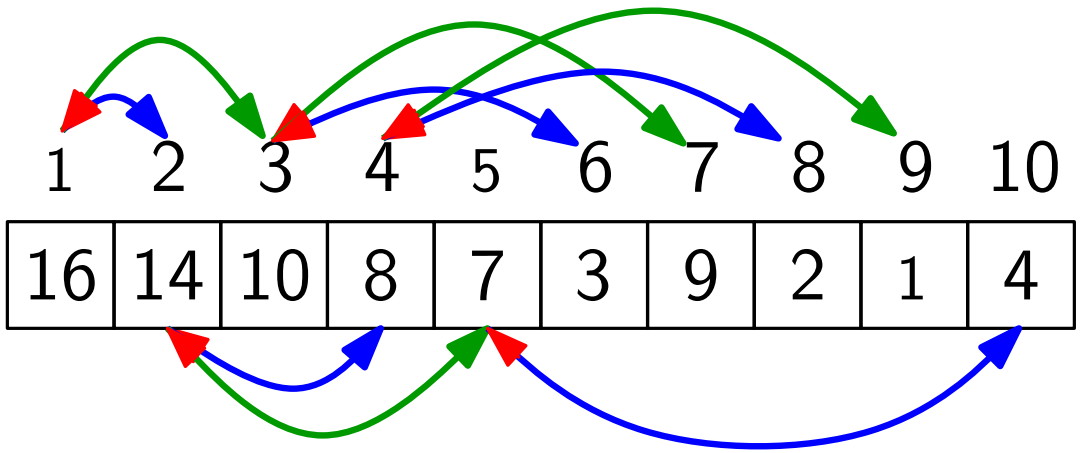
parent(index i) **return** $\lfloor i/2 \rfloor$





Bäume, gut gepackt

sehr schnelle Rechenoperationen!

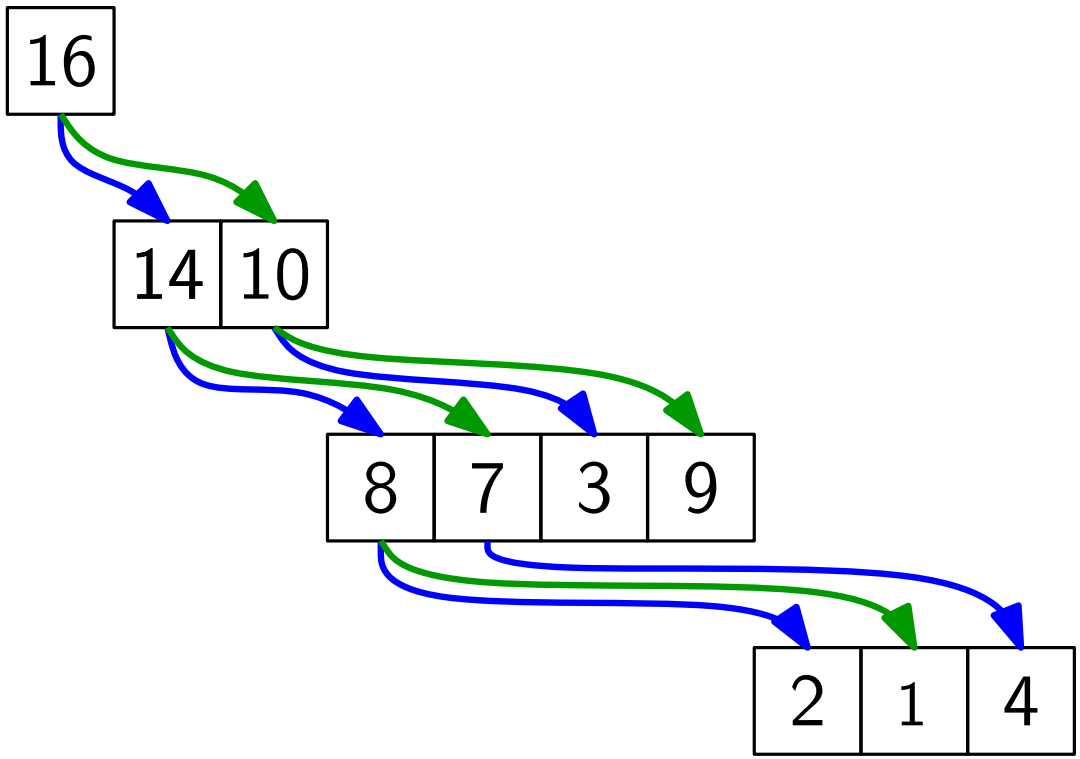


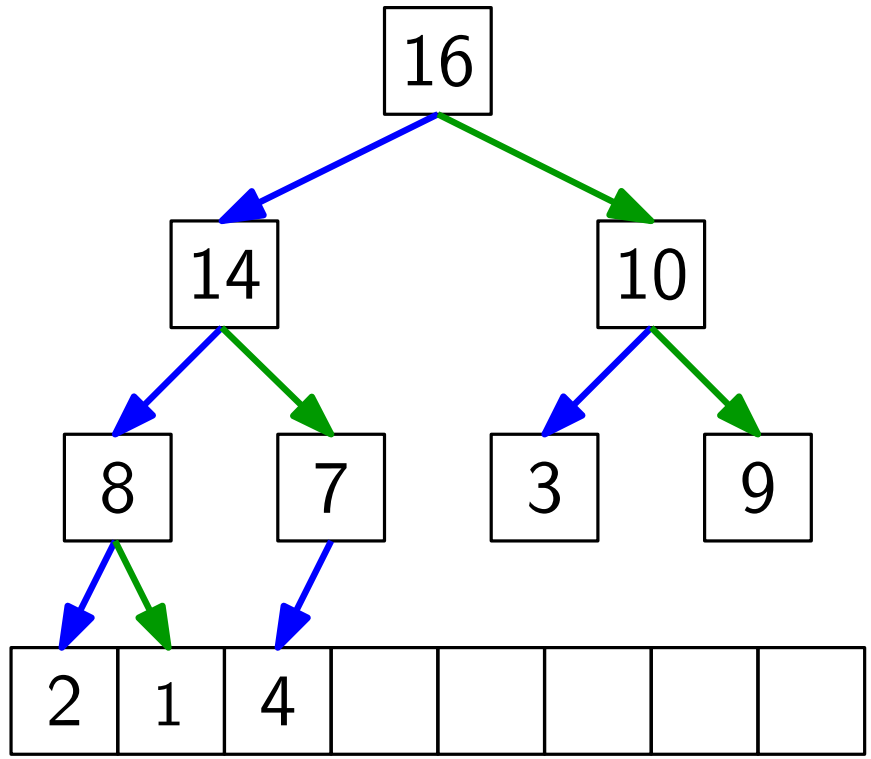
Pfeile implementieren:

left(index i) **return** $2i$

right(index i) **return** $2i + 1$

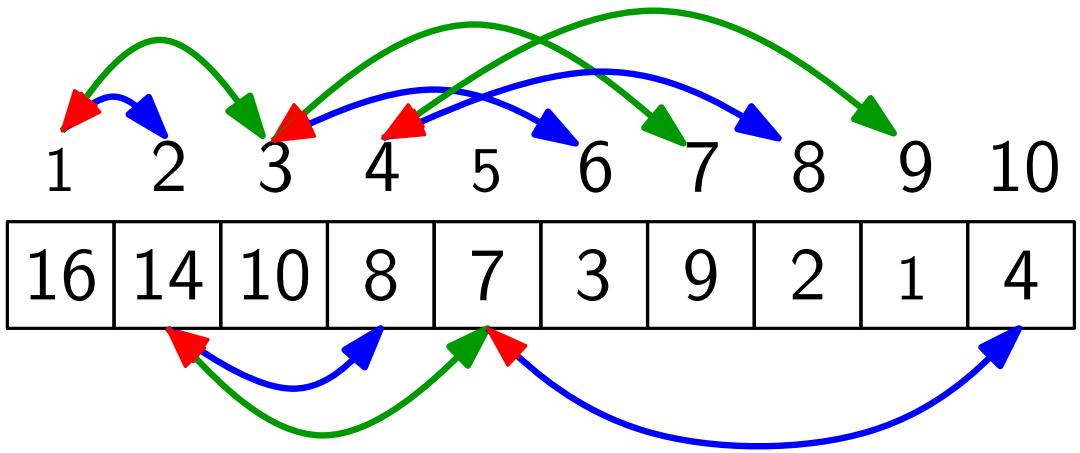
parent(index i) **return** $\lfloor i/2 \rfloor$





Bäume, gut gepackt

sehr schnelle Rechenoperationen!

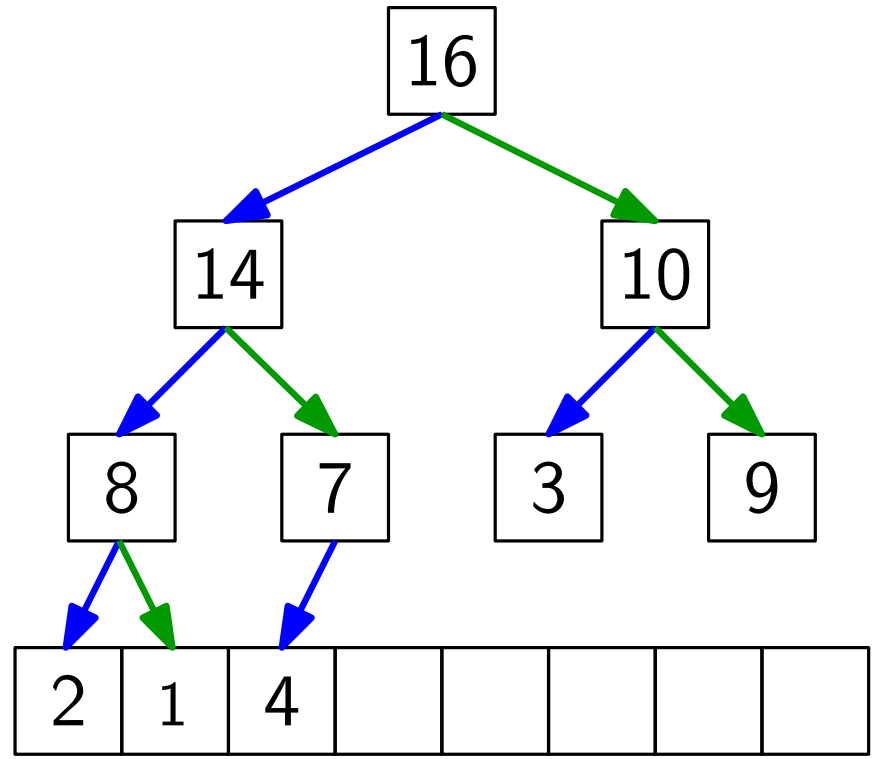


Pfeile implementieren:

left(index i)	**return**	$2i$
right(index i)	**return**	$2i + 1$
parent(index i)	**return**	$\lfloor i/2 \rfloor$

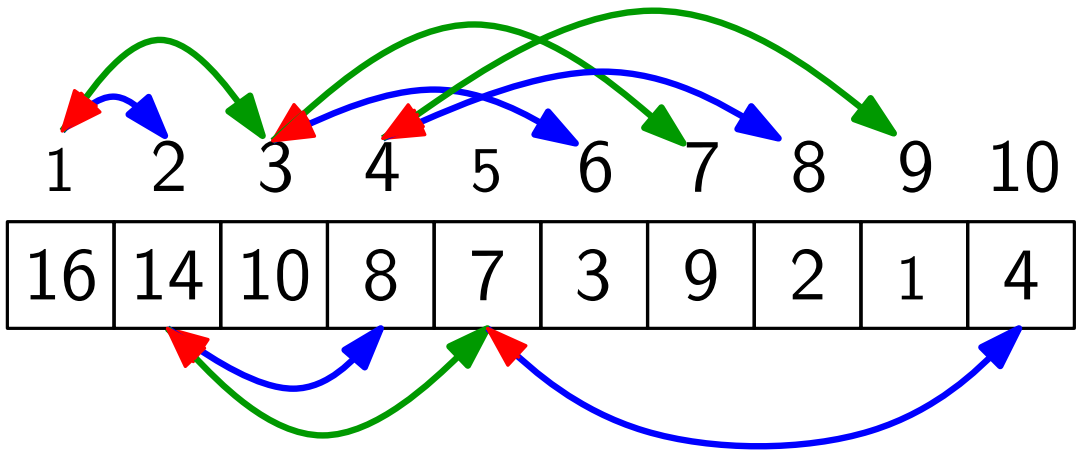
Definition:

Ein *Heap* ist ein Feld, das einem binären Baum entspricht, bei dem



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

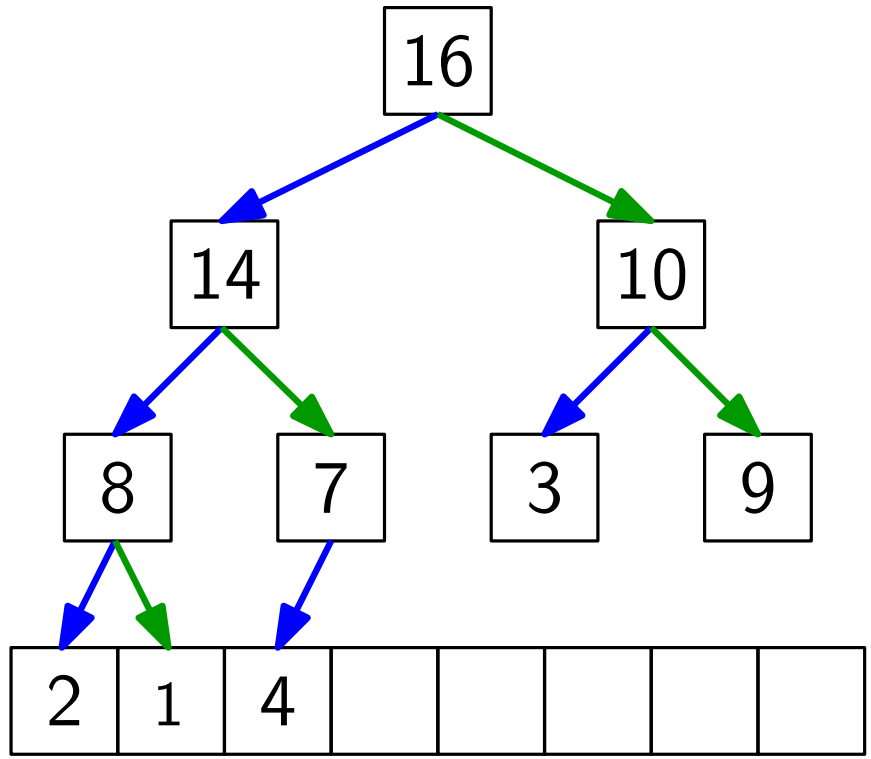
left(index i) **return** $2i$

right(index i) **return** $2i + 1$

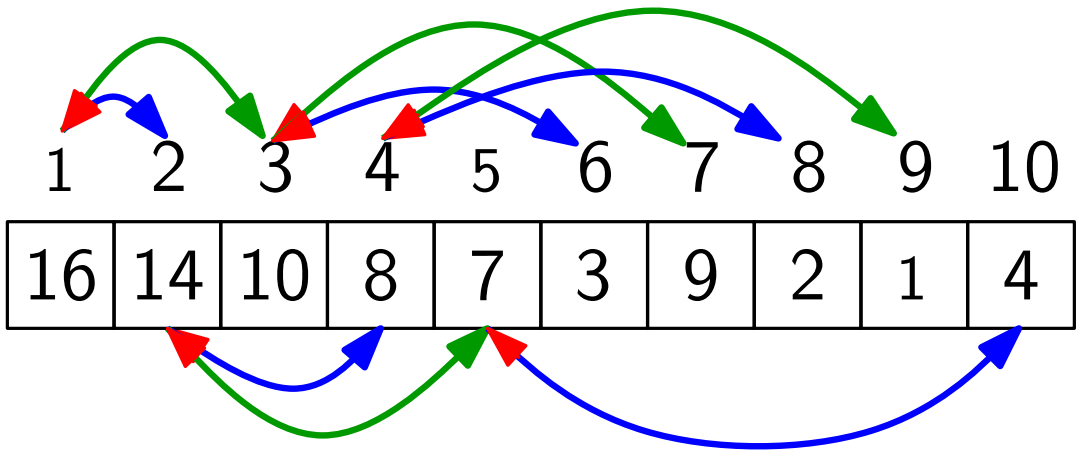
parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

Ein *Heap* ist ein Feld, das einem binären Baum entspricht, bei dem



Bäume, gut gepackt



sehr schnelle Rechenoperationen!

Pfeile implementieren:

left(index i) **return** $2i$

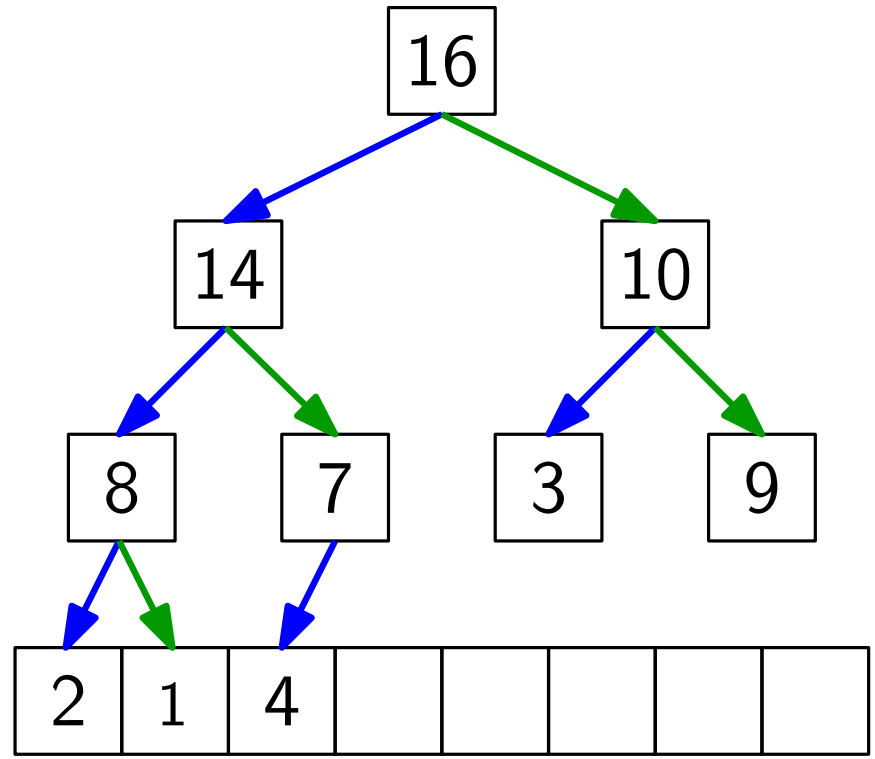
right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

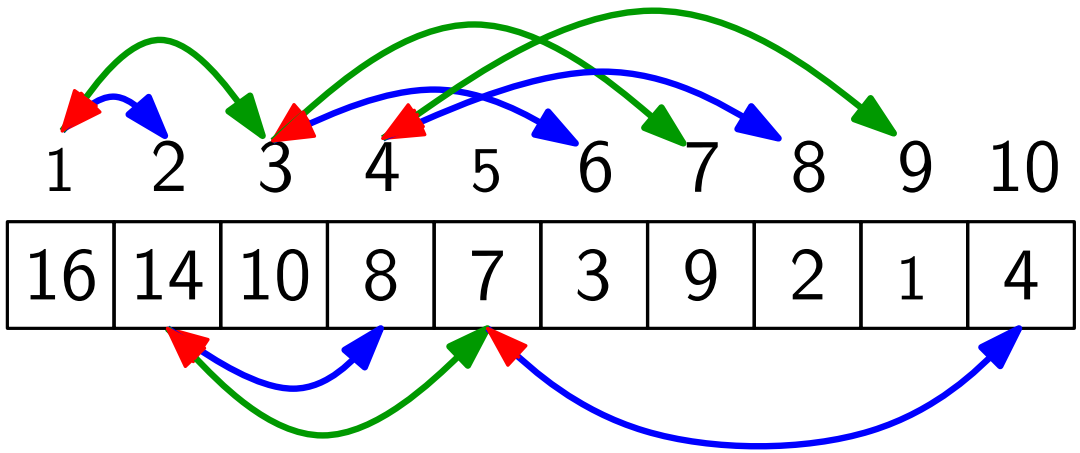
Ein *Heap* ist ein Feld, das einem binären Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

left(index i) **return** $2i$

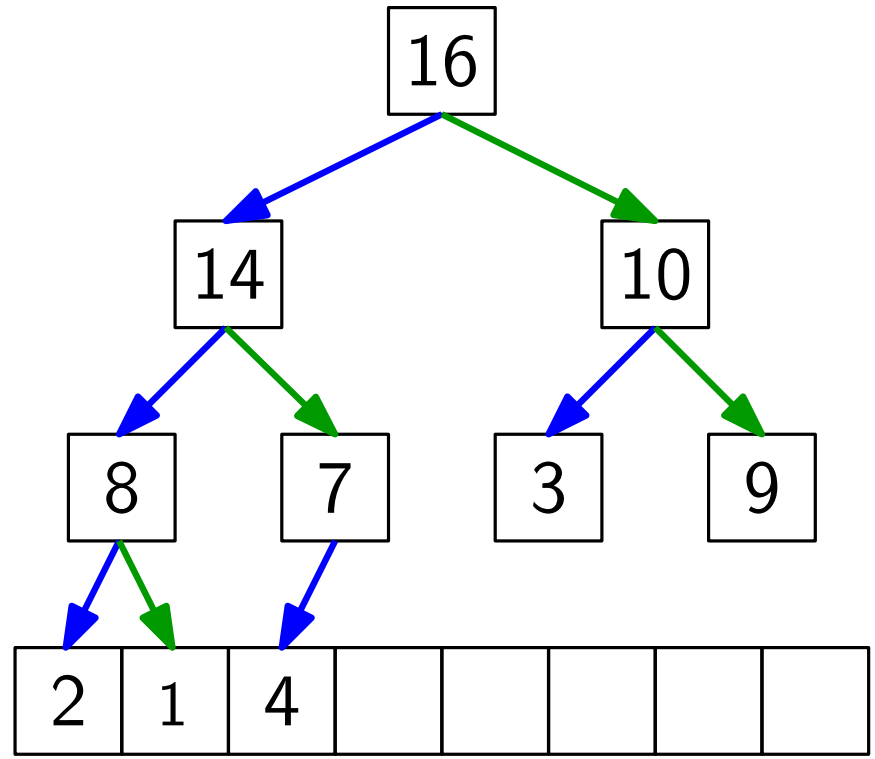
right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

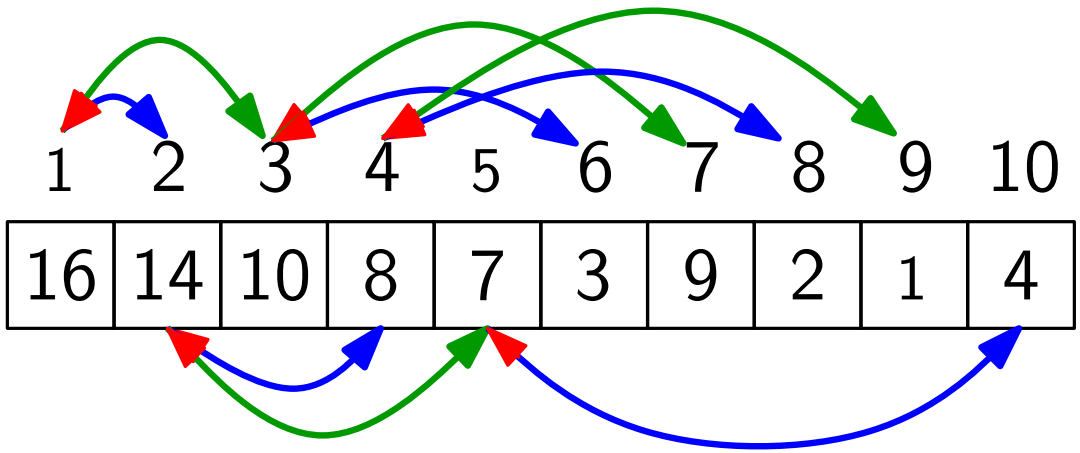
Ein *Heap* ist ein Feld, das einem binären Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

left(index i) **return** $2i$

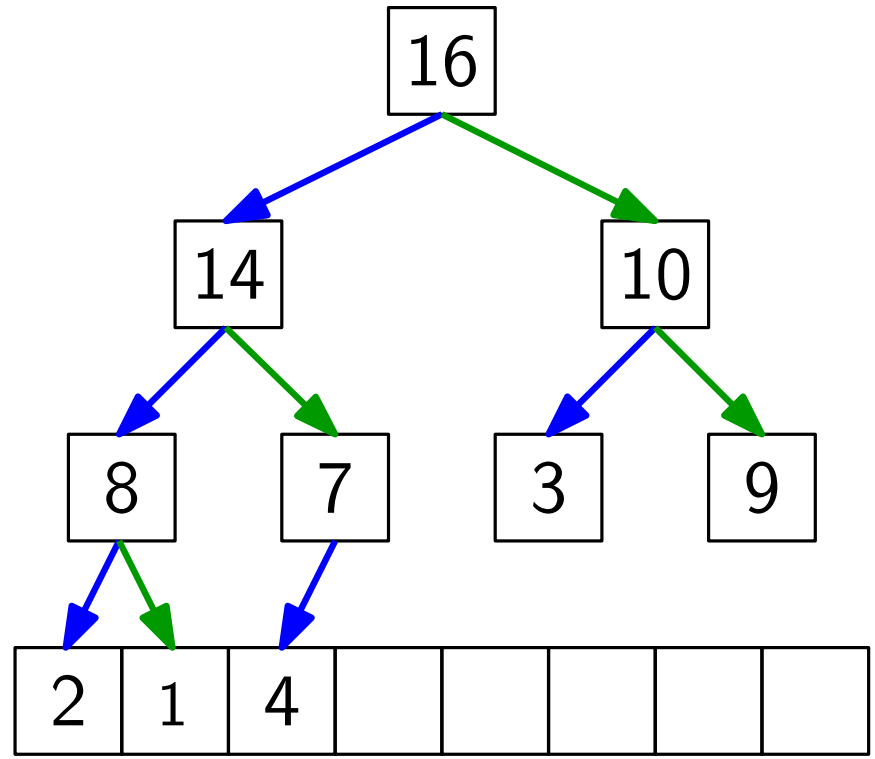
right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

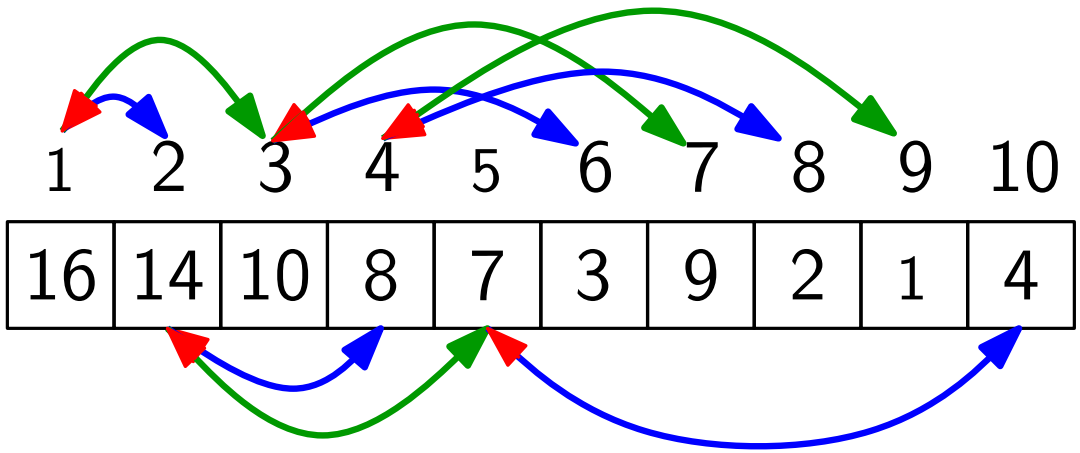
Ein *Heap* ist ein Feld, das einem binären Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und
- die *Heap-Eigenschaft* gilt.



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

left(index i) **return** $2i$

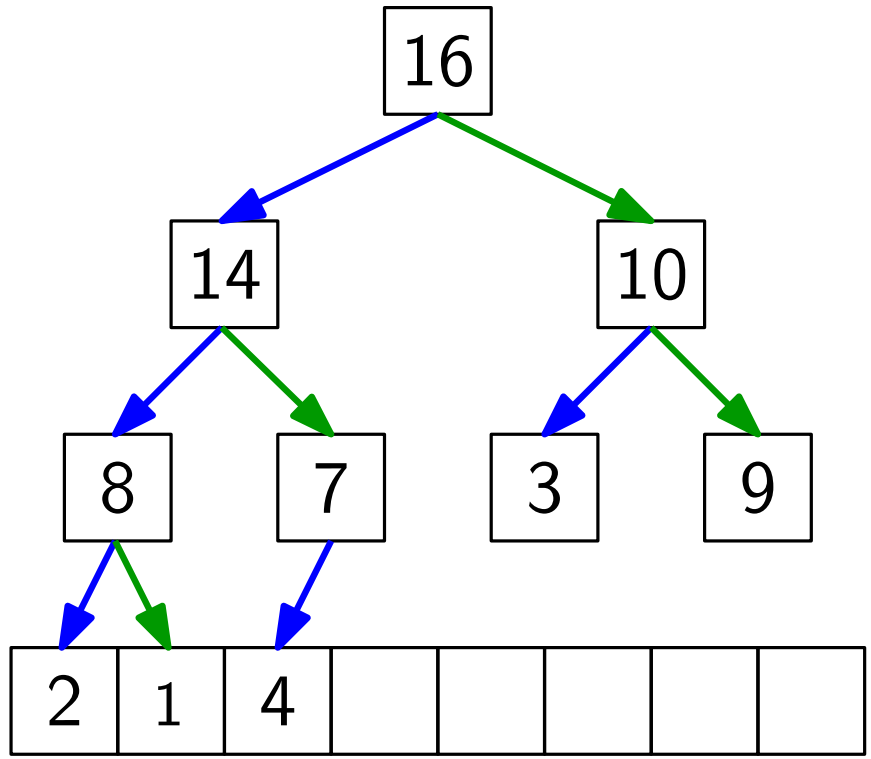
right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

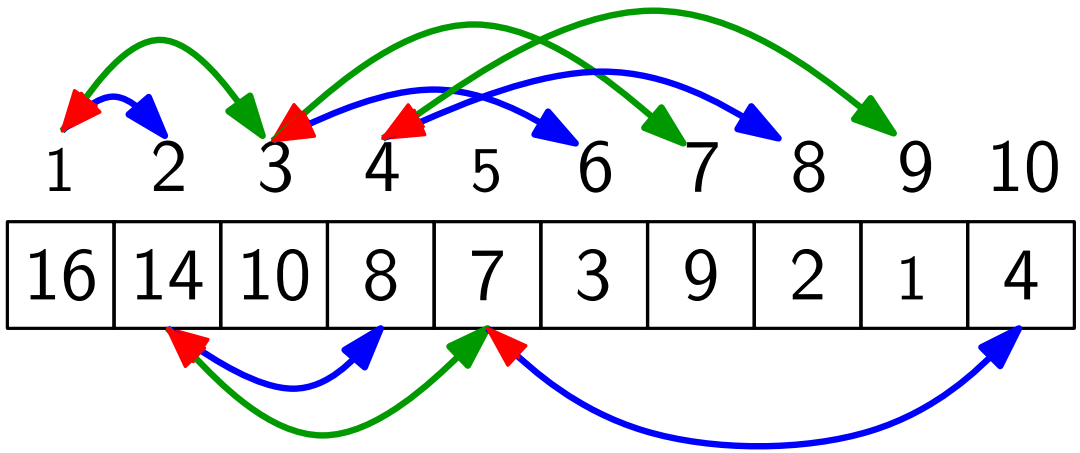
Ein Heap hat die

Max-Heap-Eigenschaft,



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

left(index i) **return** $2i$

right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

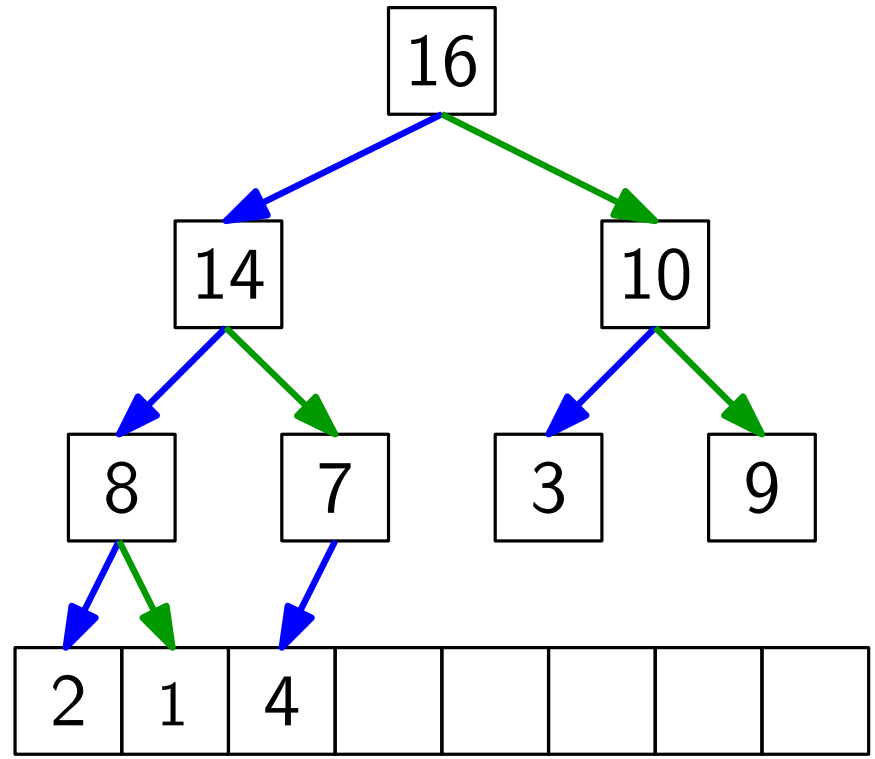
Definition:

Ein Heap hat die

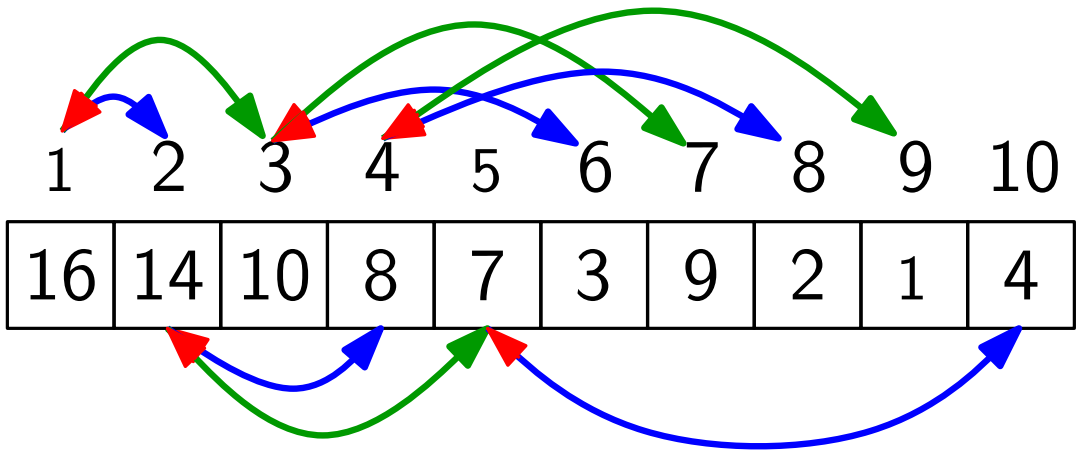
Max-Heap-Eigenschaft,

wenn für jeden Knoten $i > 1$ gilt:

$A[\text{parent}(i)] \geq A[i]$.



Bäume, gut gepackt



sehr schnelle Rechenoperationen!

Pfeile implementieren:

left(index i) **return** $2i$

right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

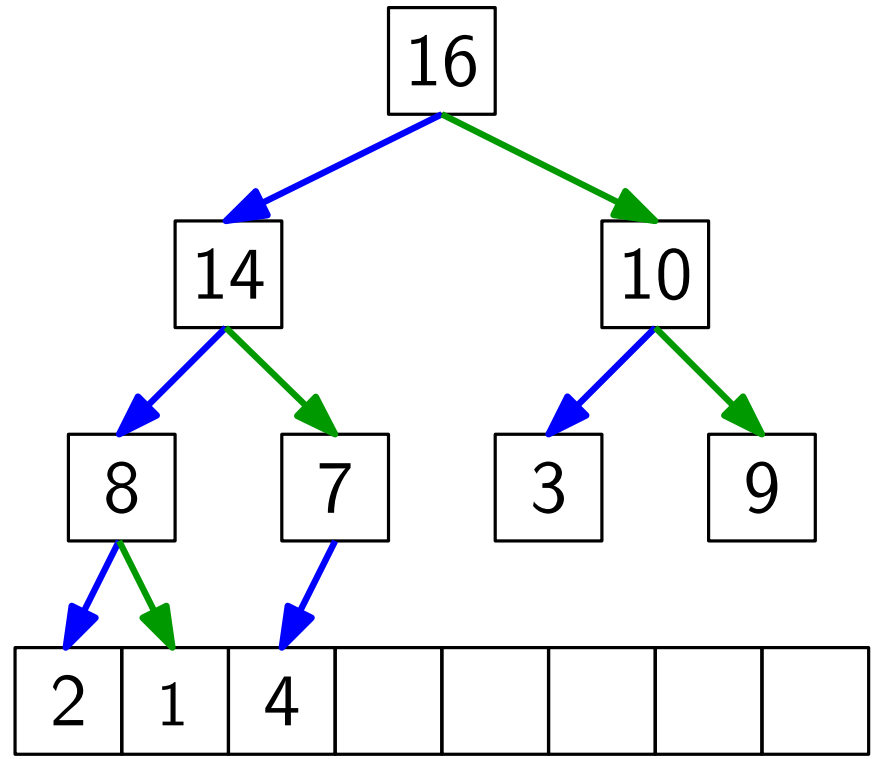
Definition:

Ein Heap hat die

Max-Heap-Eigenschaft,

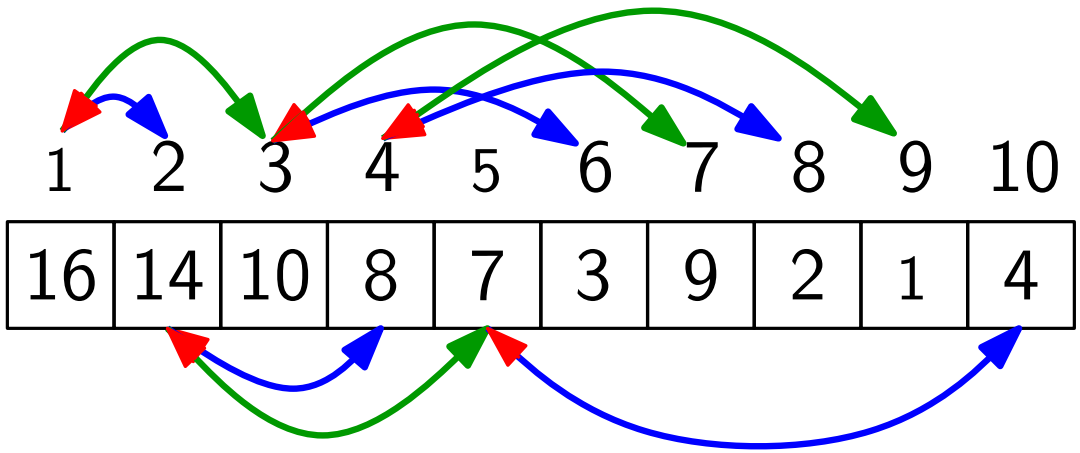
wenn für jeden Knoten $i > 1$ gilt:
$A[\text{parent}(i)] \geq A[i]$.

So ein Heap heißt *Max-Heap*.



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

left(index i) **return** $2i$

right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

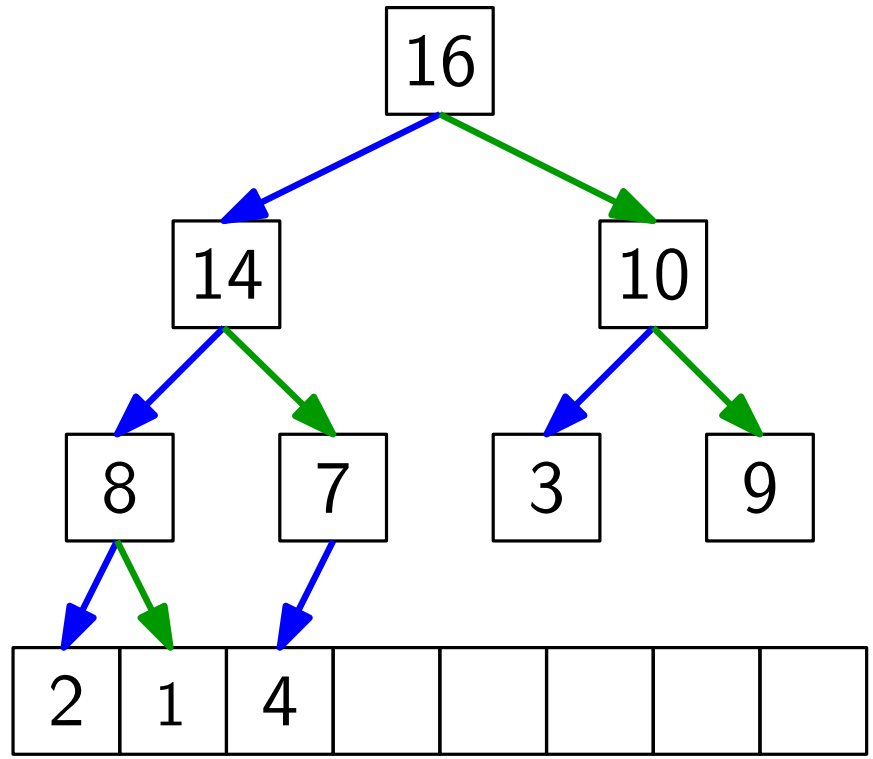
Ein Heap hat die

~~*Max-Heap-Eigenschaft*~~,

wenn für jeden Knoten $i > 1$ gilt:

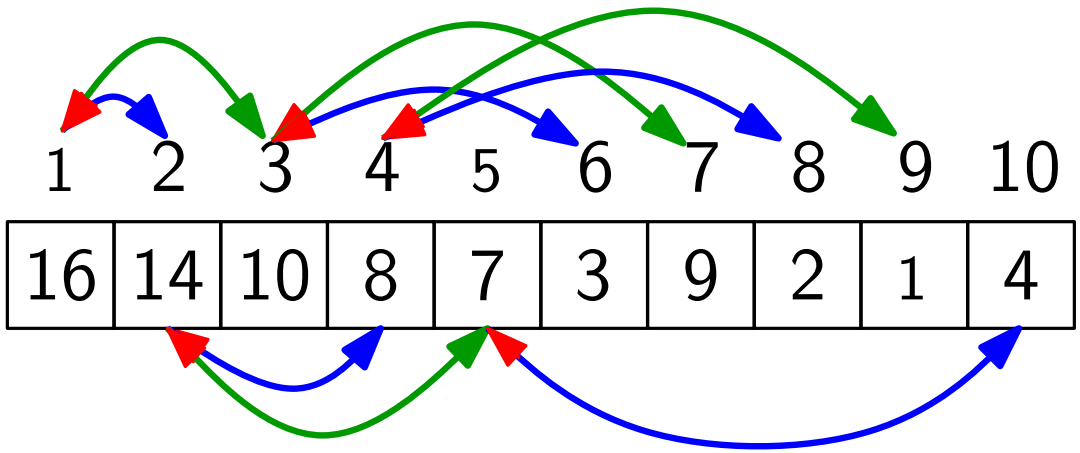
$A[\text{parent}(i)] \not\geq A[i]$.

So ein Heap heißt ~~*Max-Heap*~~.



Bäume, gut gepackt

sehr schnelle Rechenoperationen!



Pfeile implementieren:

left(index i) **return** $2i$

right(index i) **return** $2i + 1$

parent(index i) **return** $\lfloor i/2 \rfloor$

Definition:

Ein Heap hat die

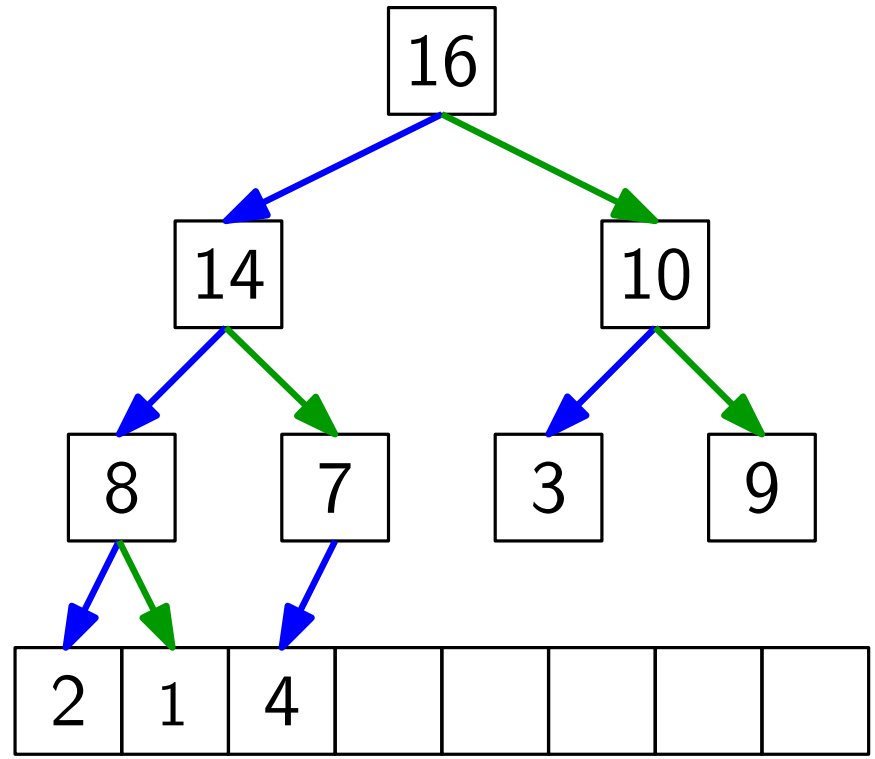
~~*Max*~~ *Min*-*Heap-Eigenschaft*,

wenn für jeden Knoten $i > 1$ gilt:

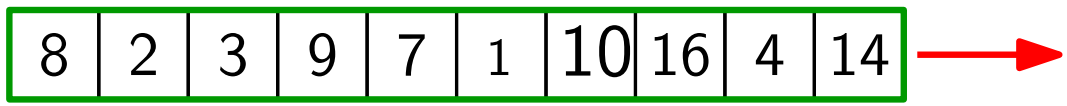
$A[\text{parent}(i)] \not\geq A[i]$.

$\leq$

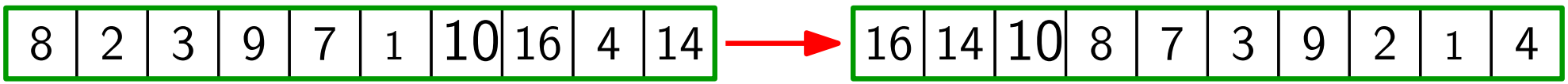
So ein Heap heißt ~~*Max*~~ *Min*-*Heap*.



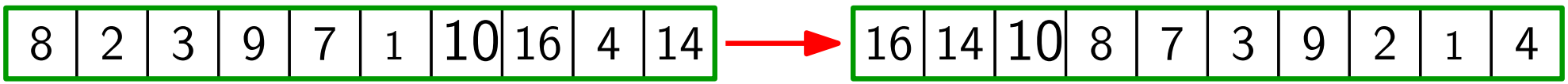
Baustelle



Baustelle



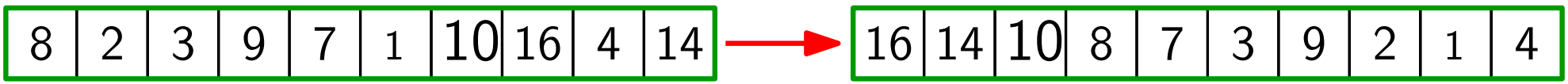
Baustelle



„totales Chaos“

Max-Heap-Eigenschaft

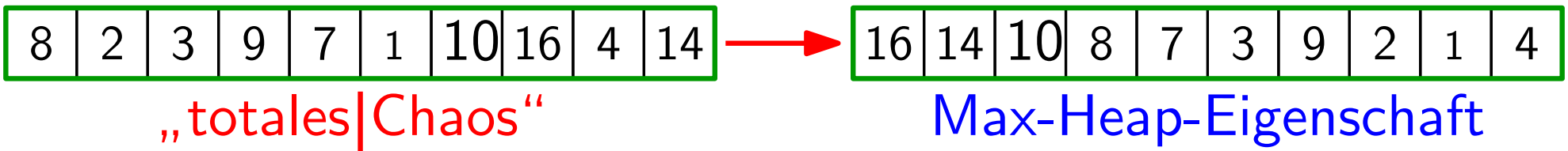
Baustelle



„totales Chaos“ Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

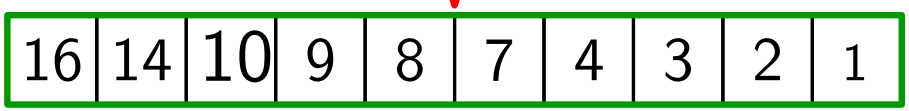
Baustelle



„totales Chaos“

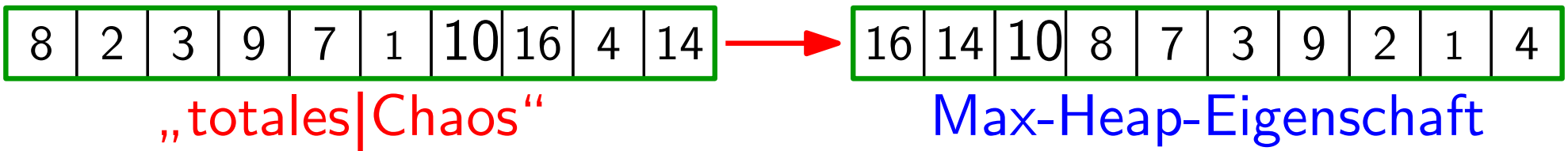
Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!



Absteigende Sortierung

Baustelle

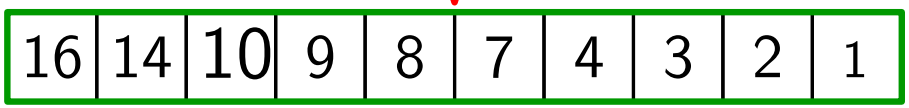


„totales Chaos“

Max-Heap-Eigenschaft

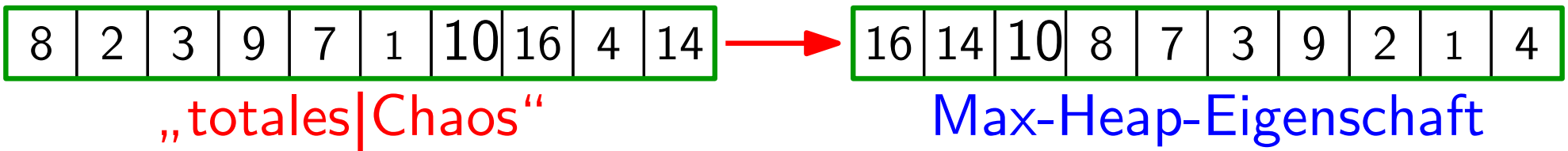
Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!



Absteigende Sortierung

Baustelle

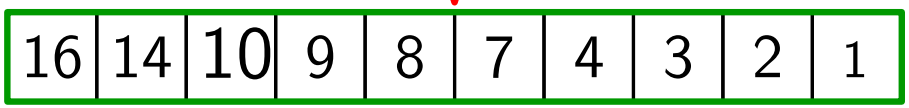


„totales Chaos“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

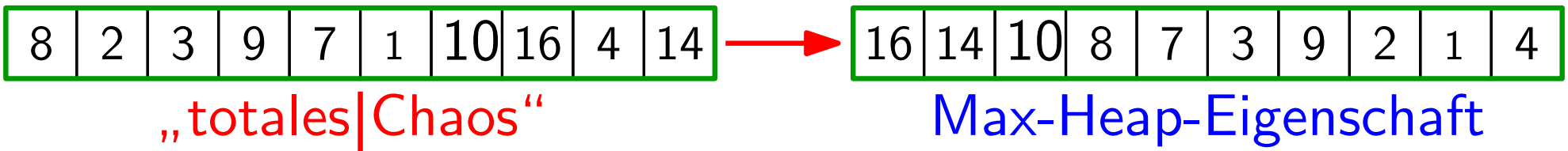
Nimm MergeSort!



Absteigende Sortierung

Fertig?

Baustelle

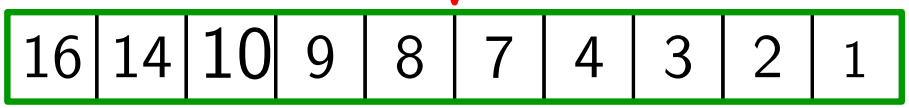


„totales Chaos“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

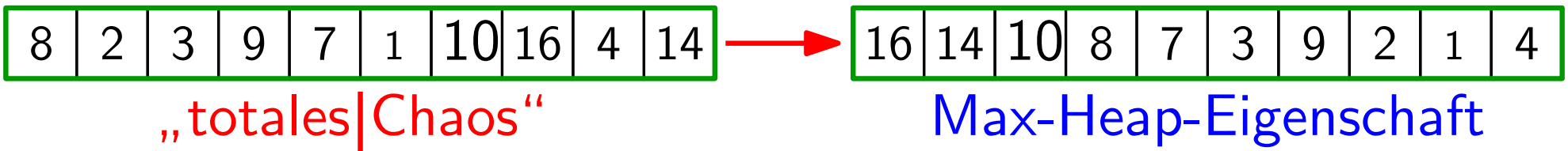
Nimm MergeSort!



Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Baustelle

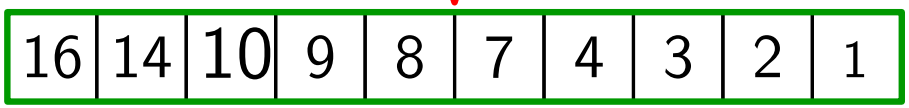


„totales Chaos“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

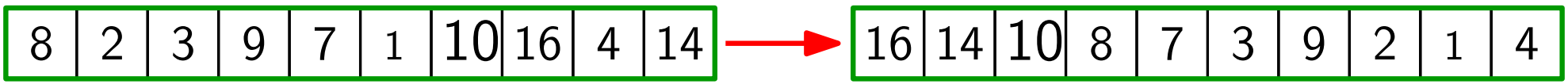


Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Baustelle

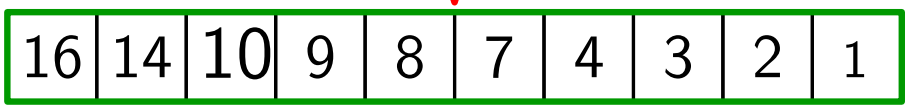


„totales Chaos“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!



Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!

Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

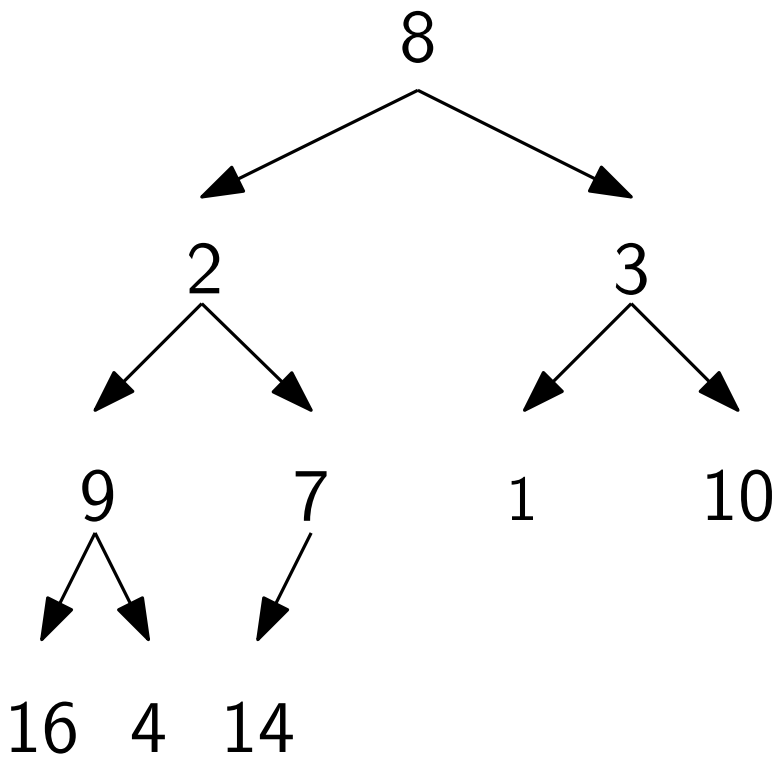
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

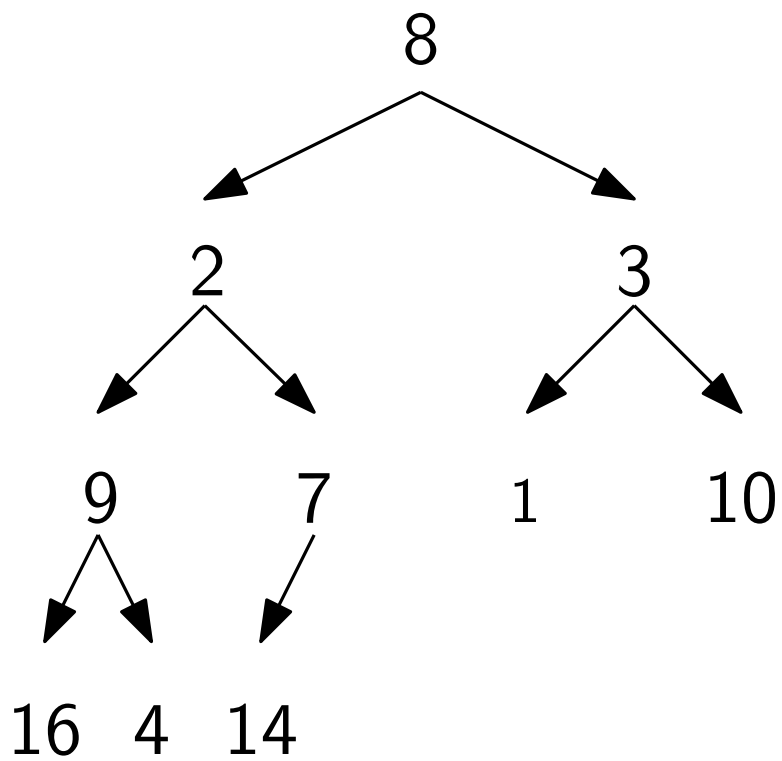
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

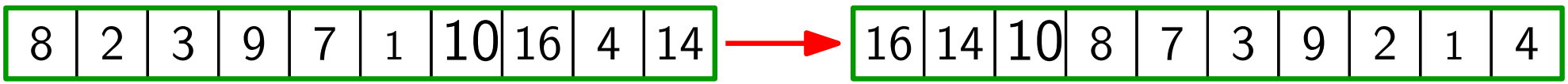
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

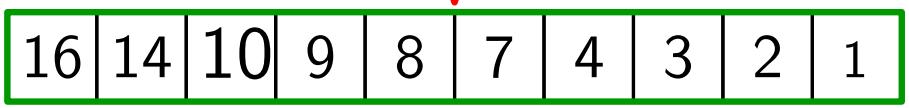


„totales Chaos“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

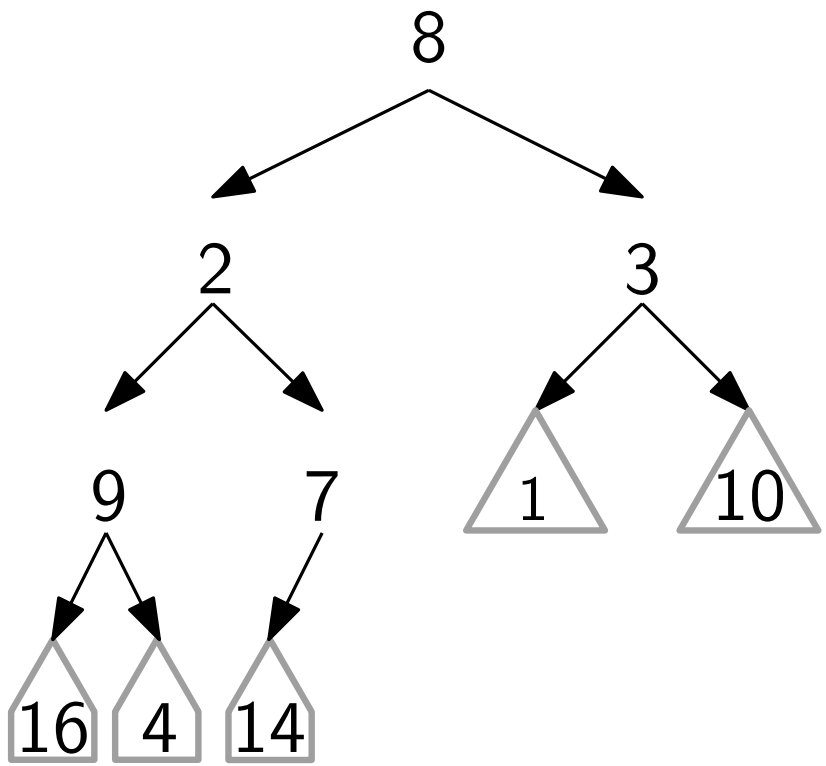


Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

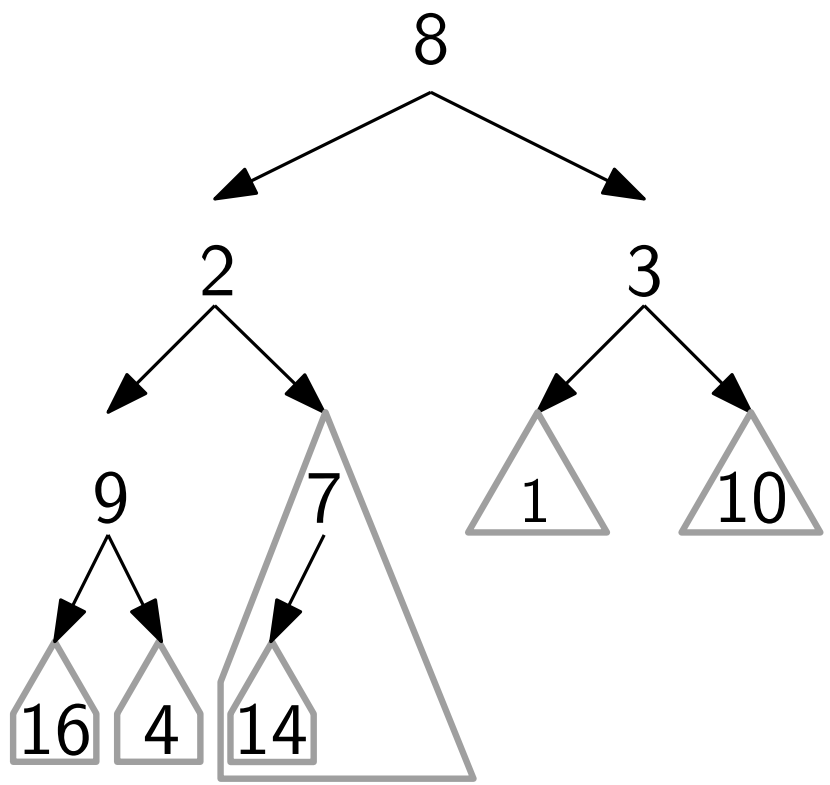
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

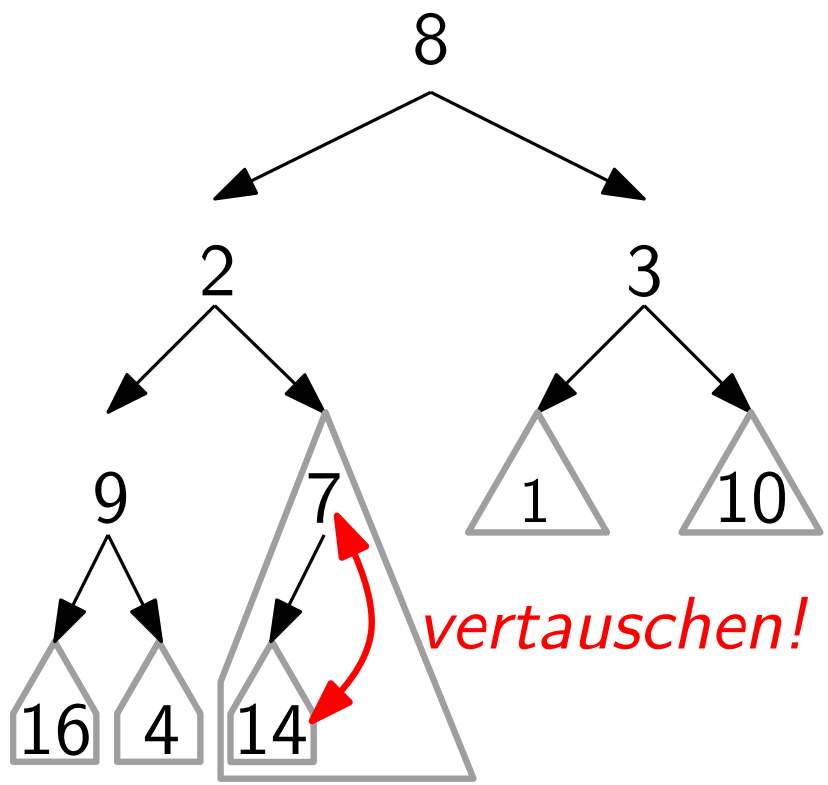
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

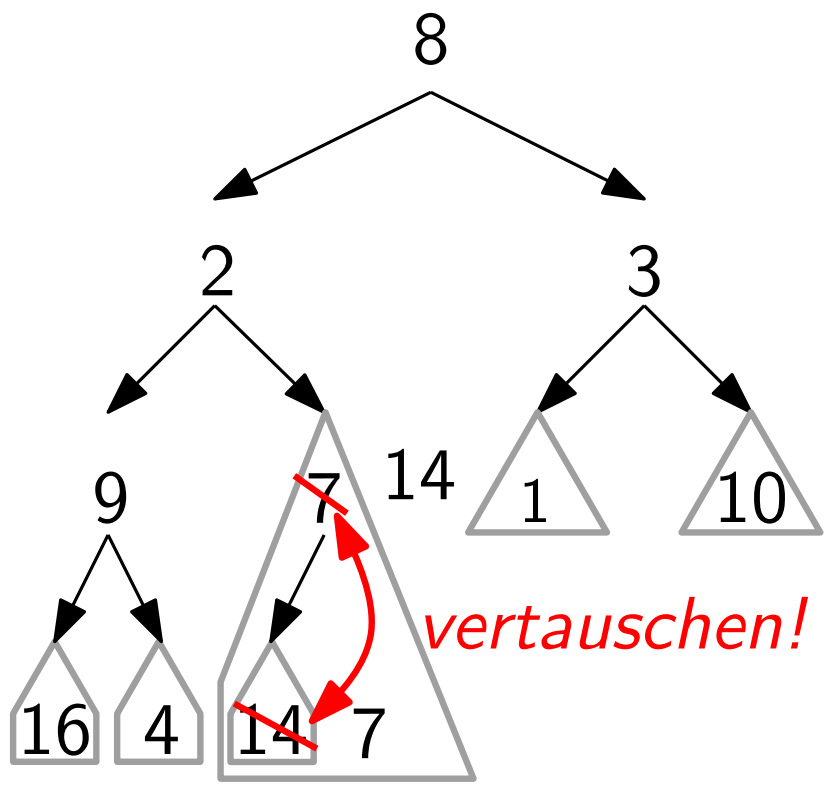
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

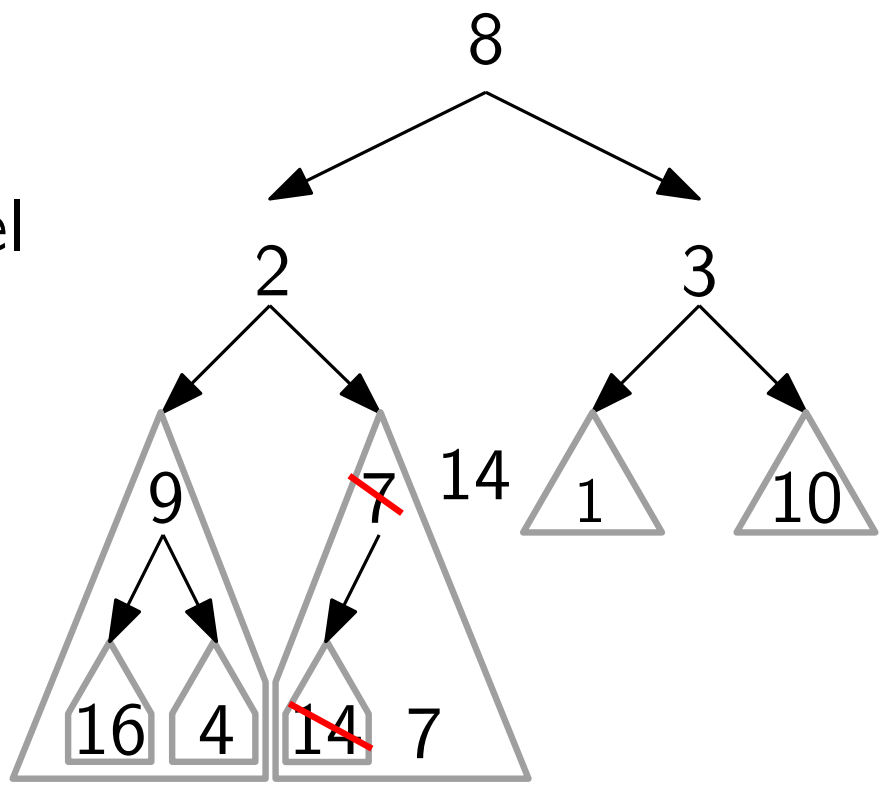
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

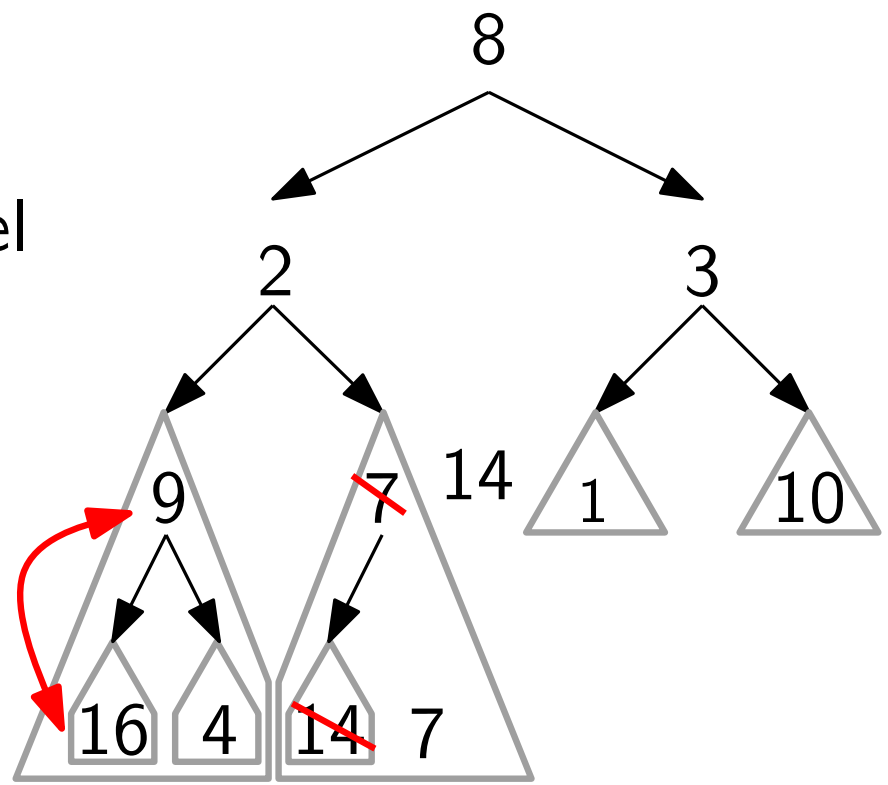
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

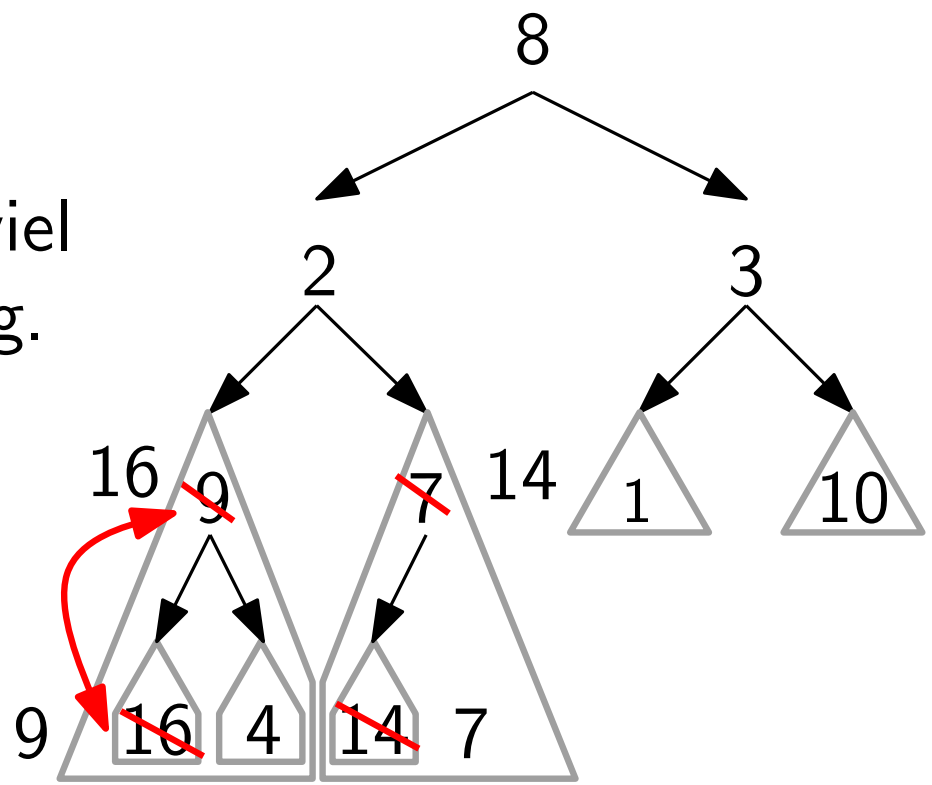
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

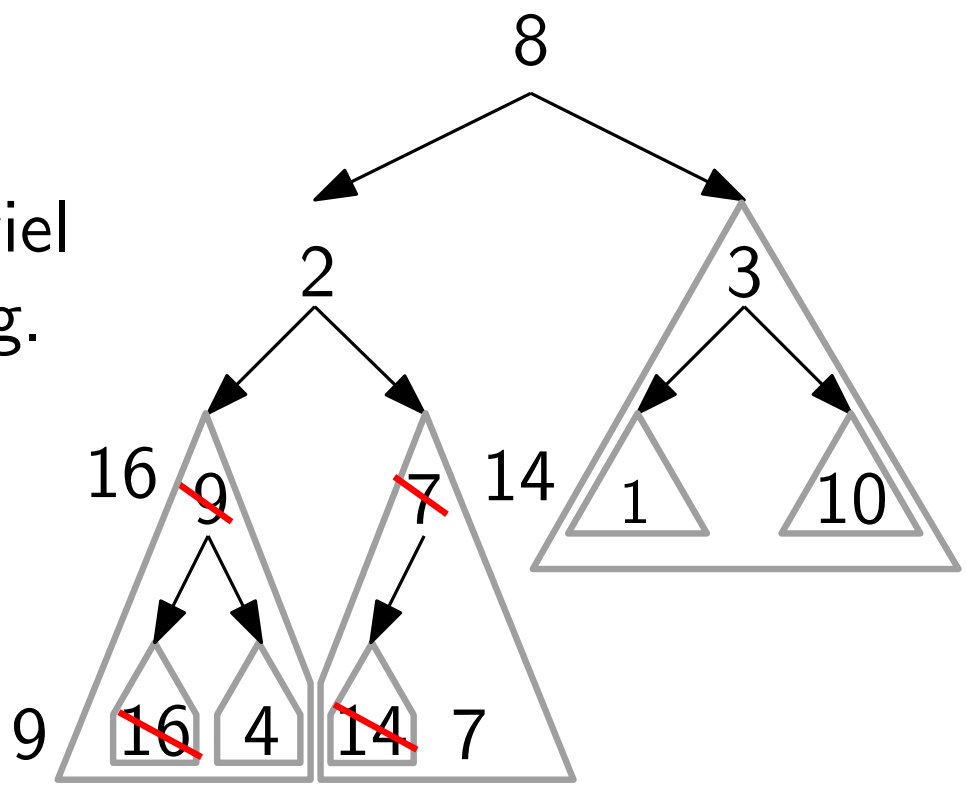
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

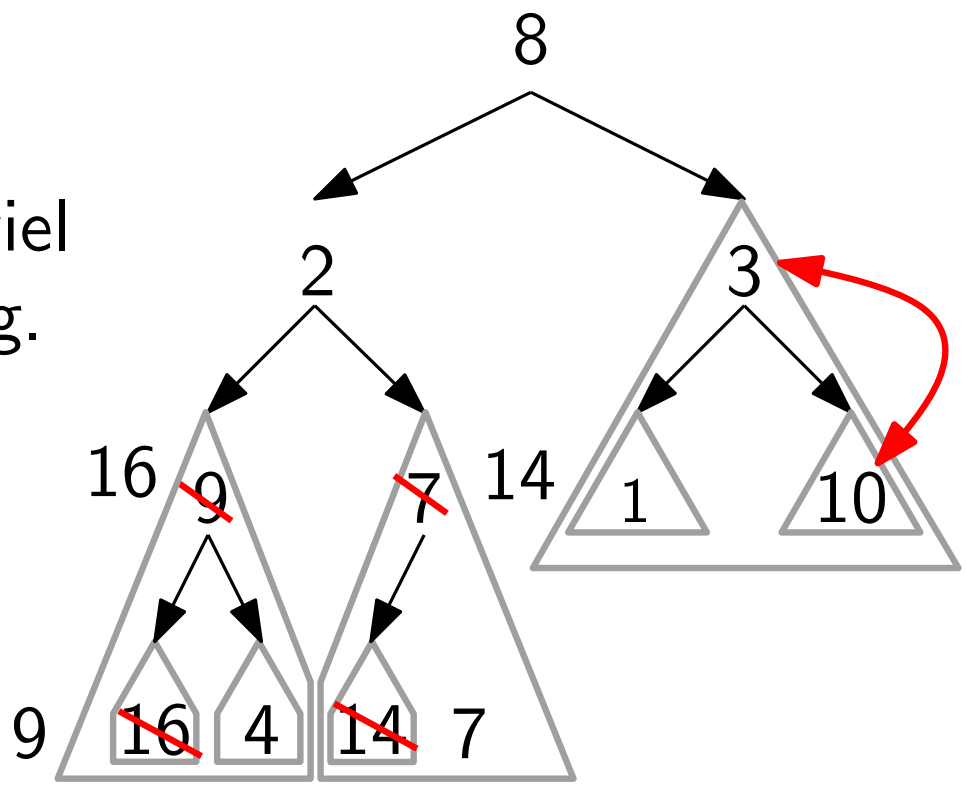
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

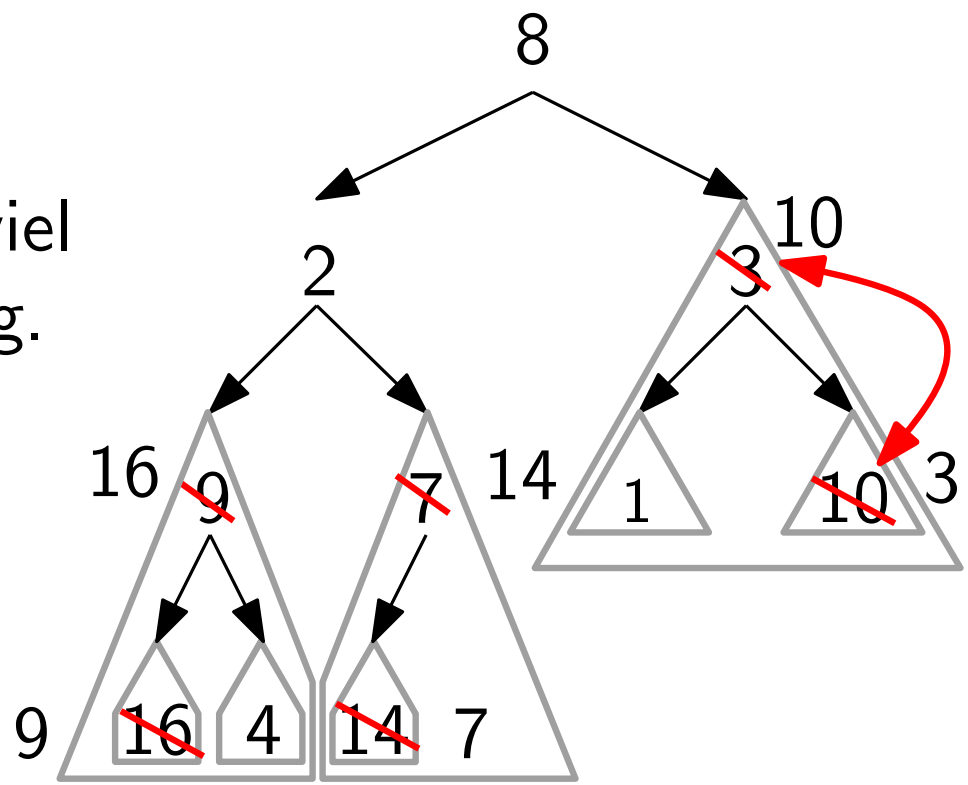
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

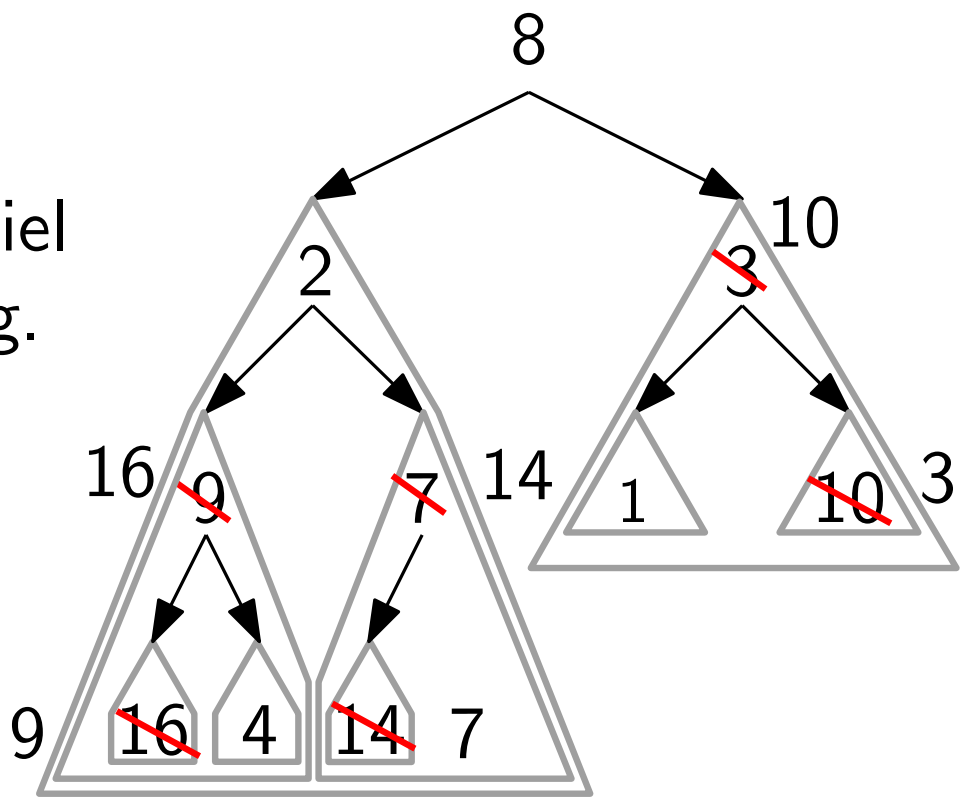
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

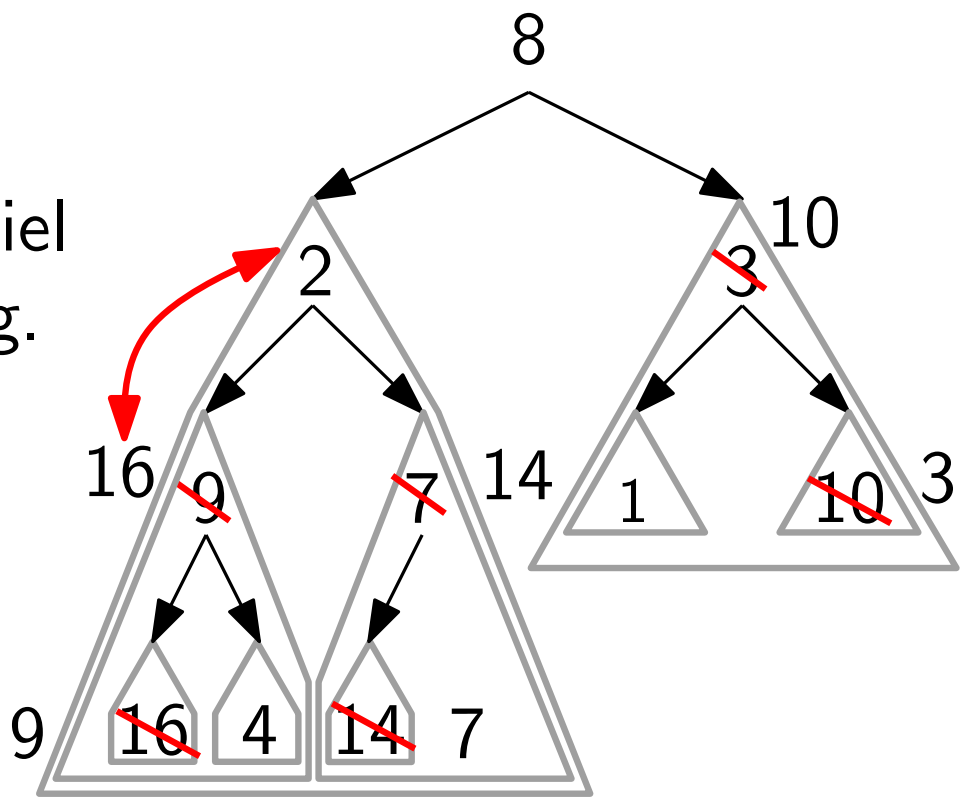
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

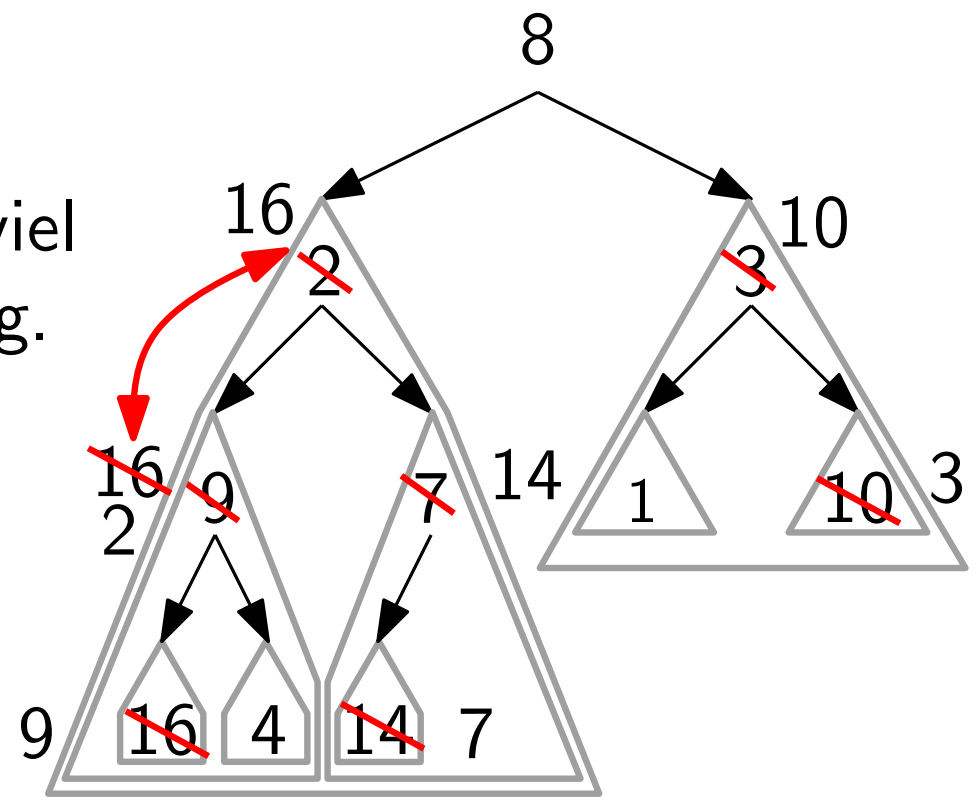
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

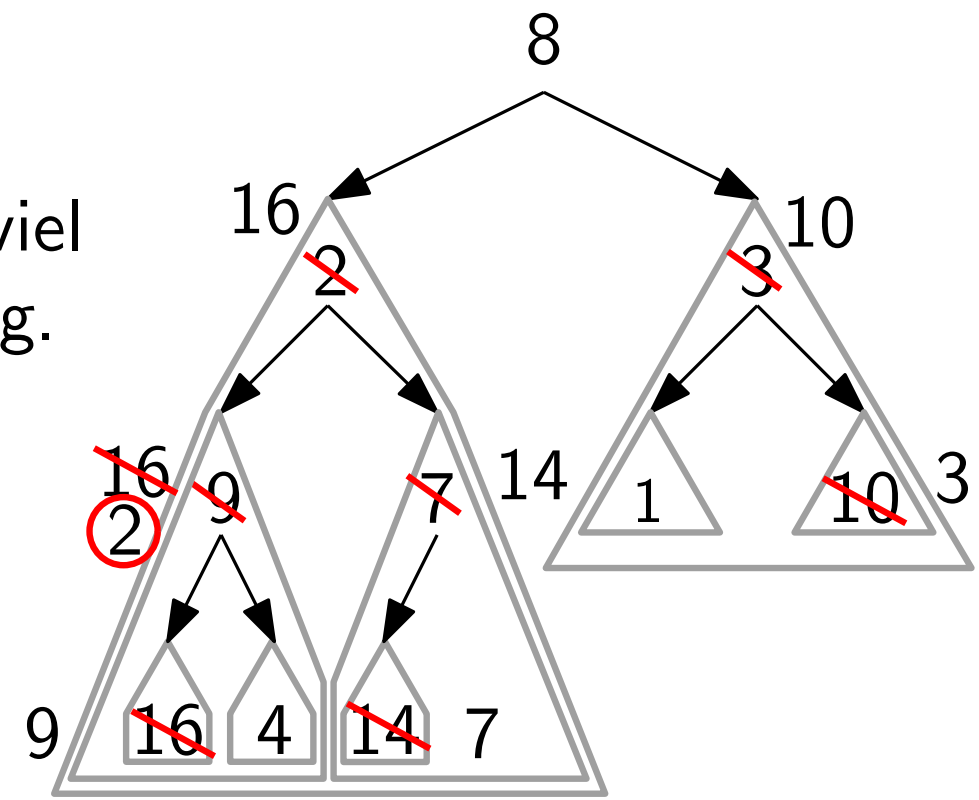
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

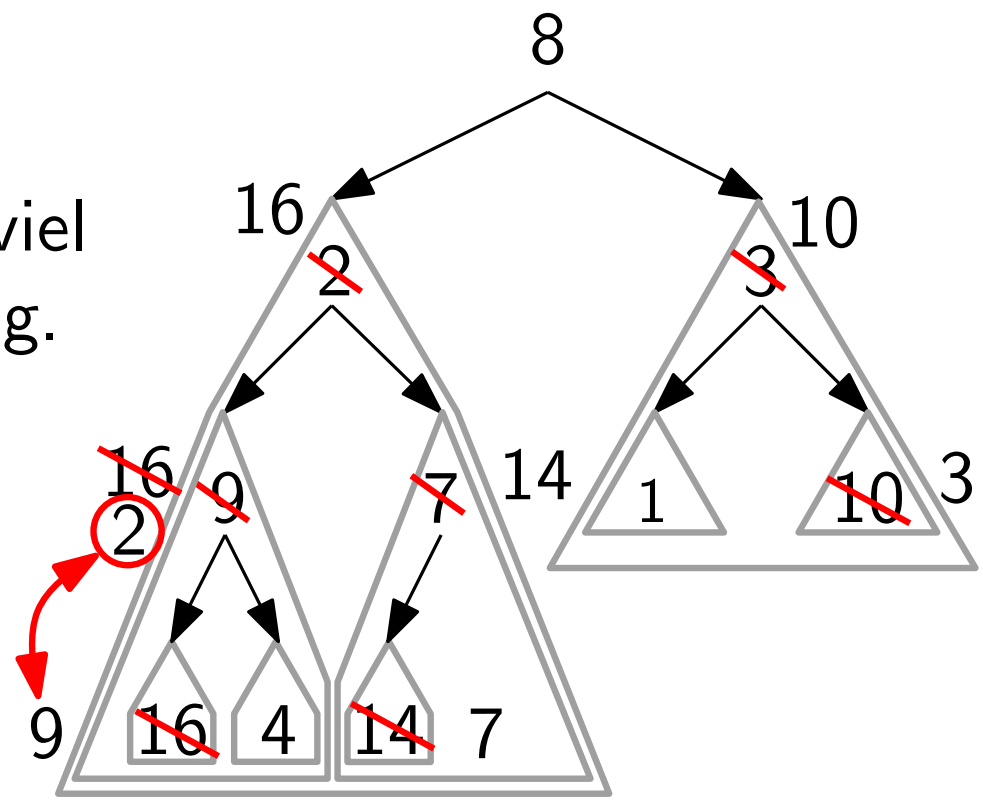
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

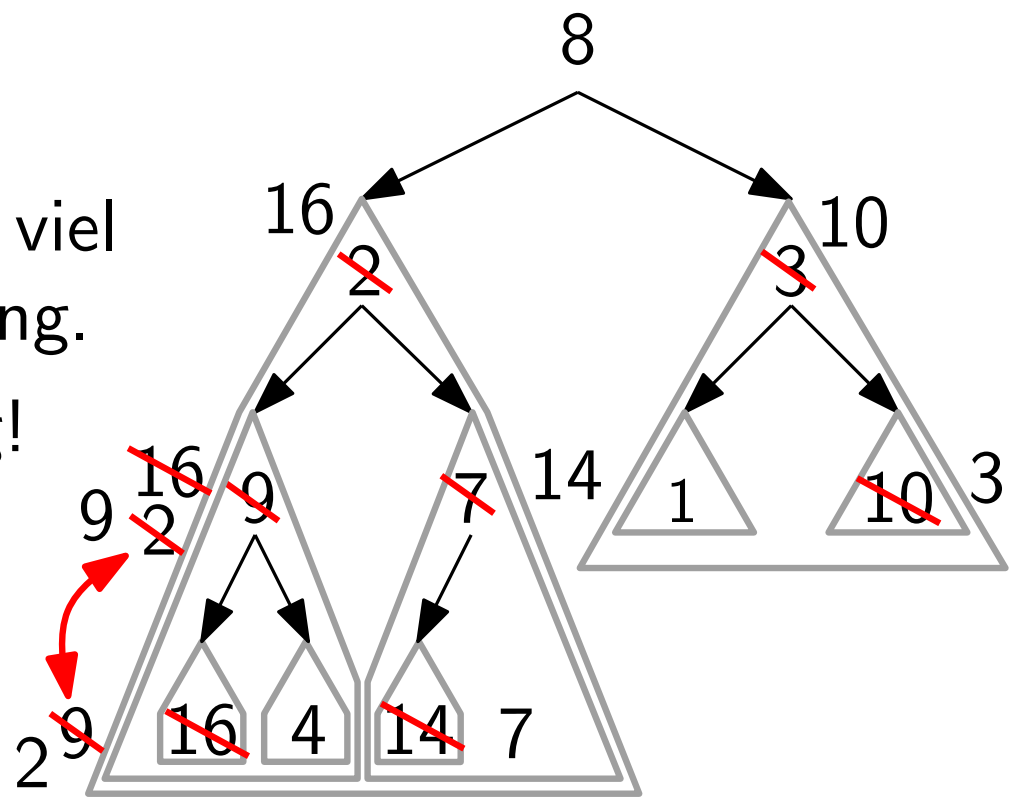
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

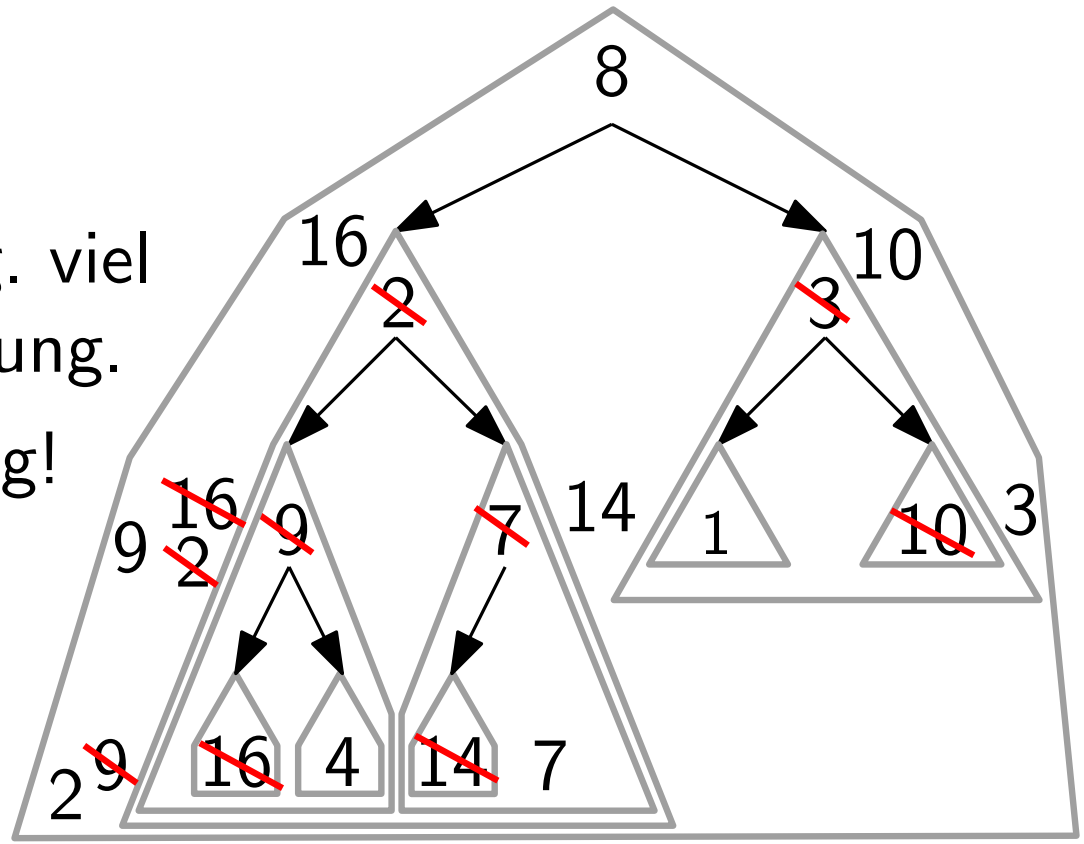
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

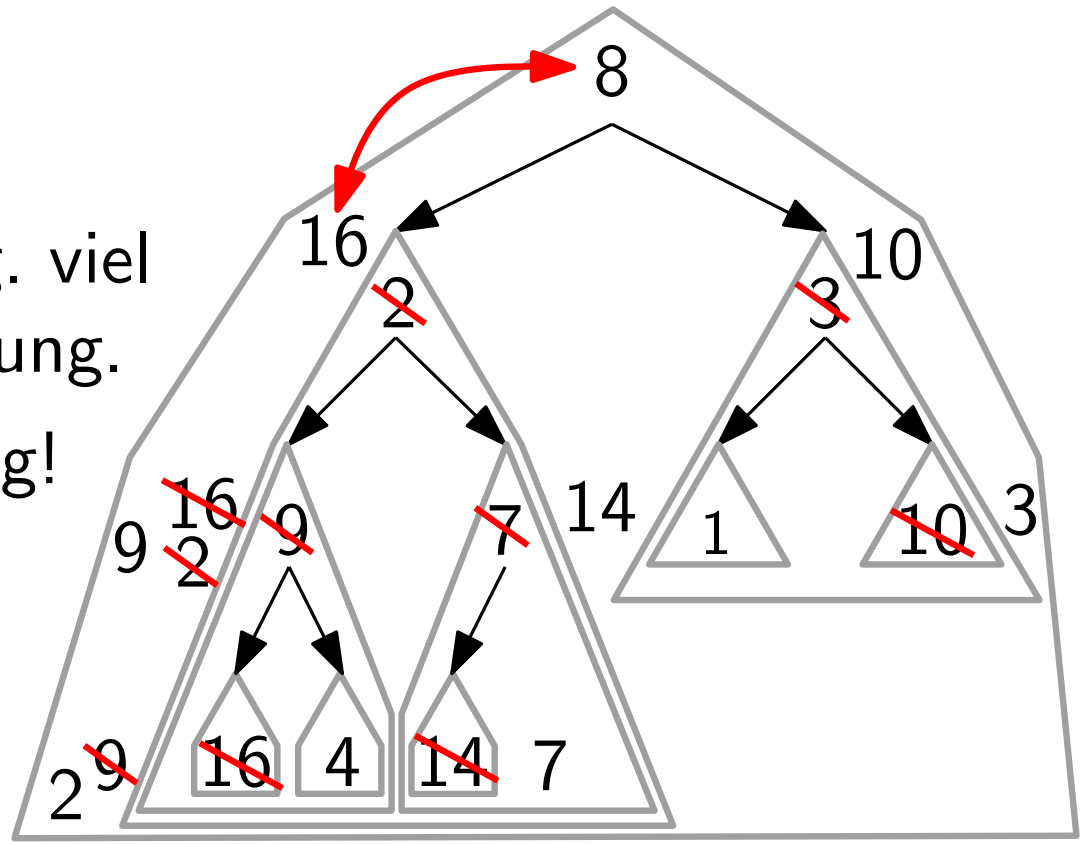
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

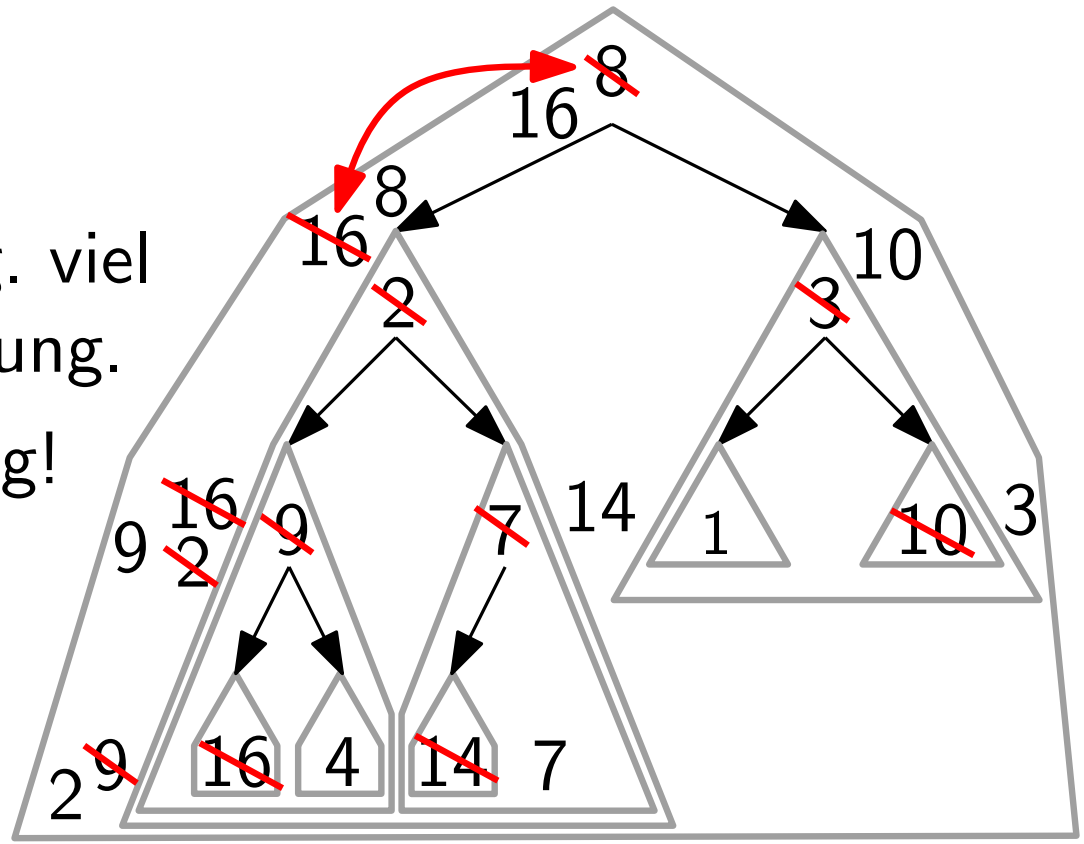
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

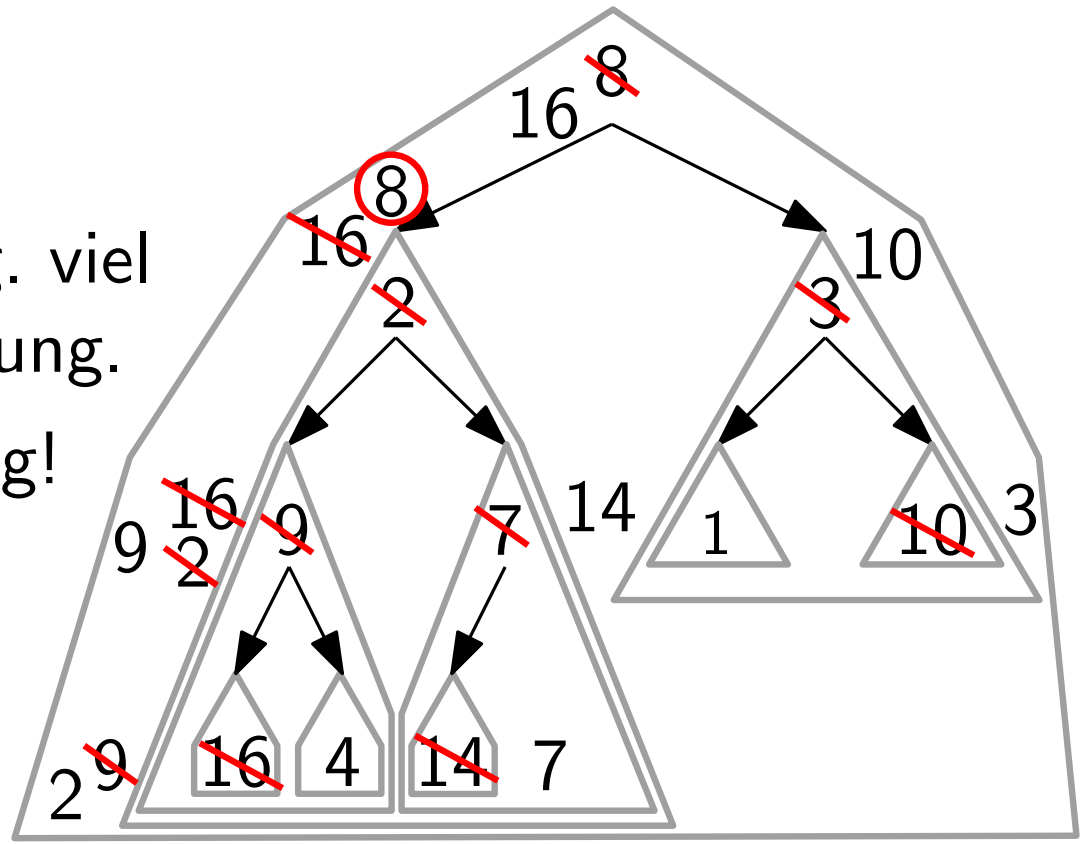
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

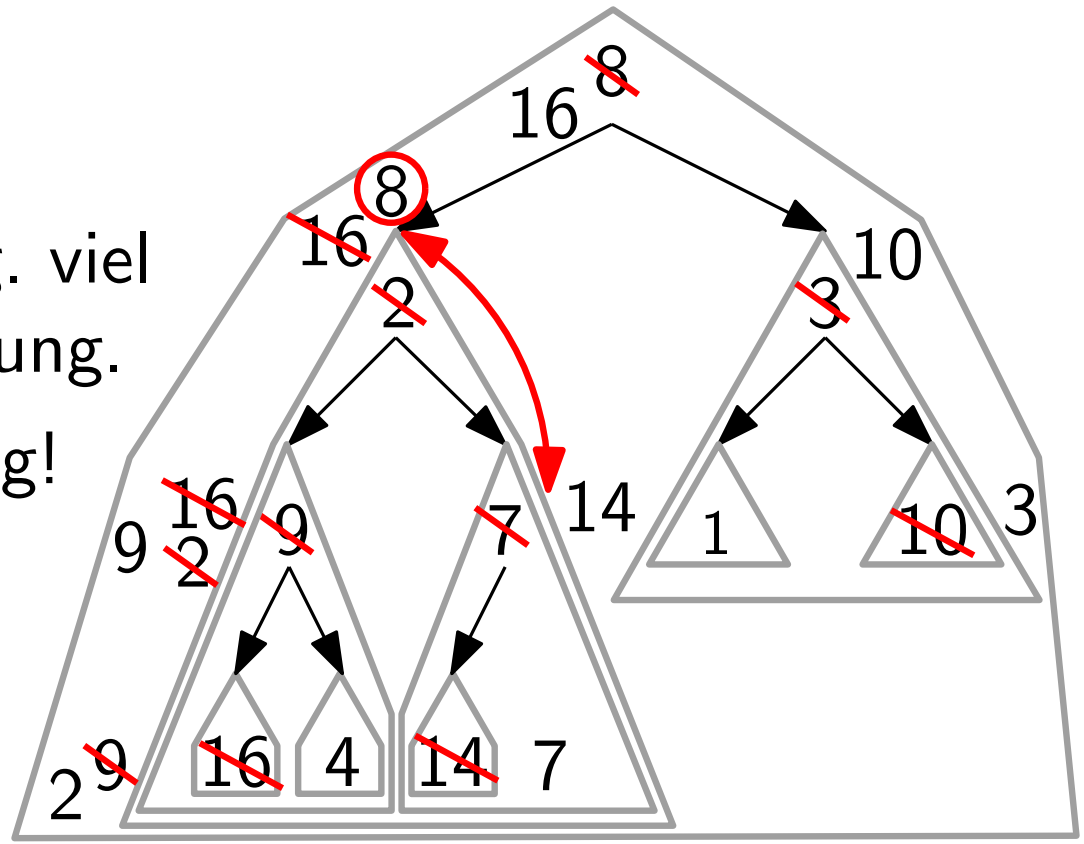
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

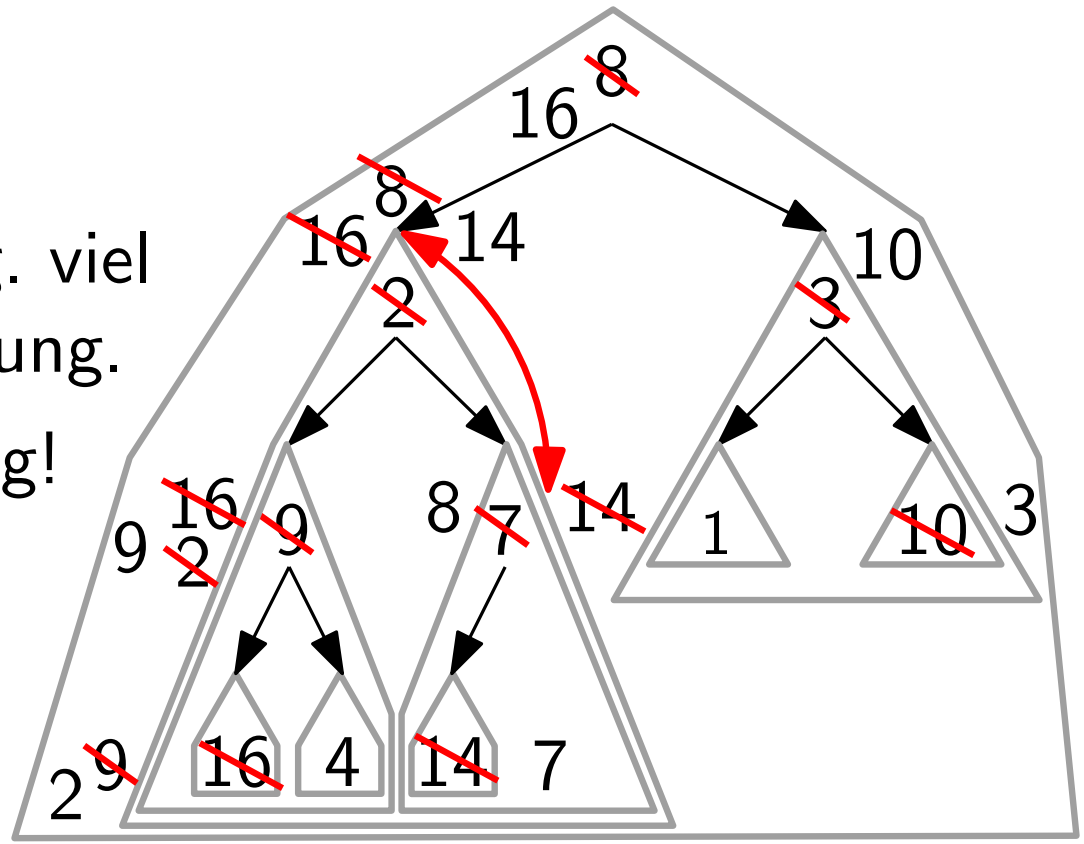
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

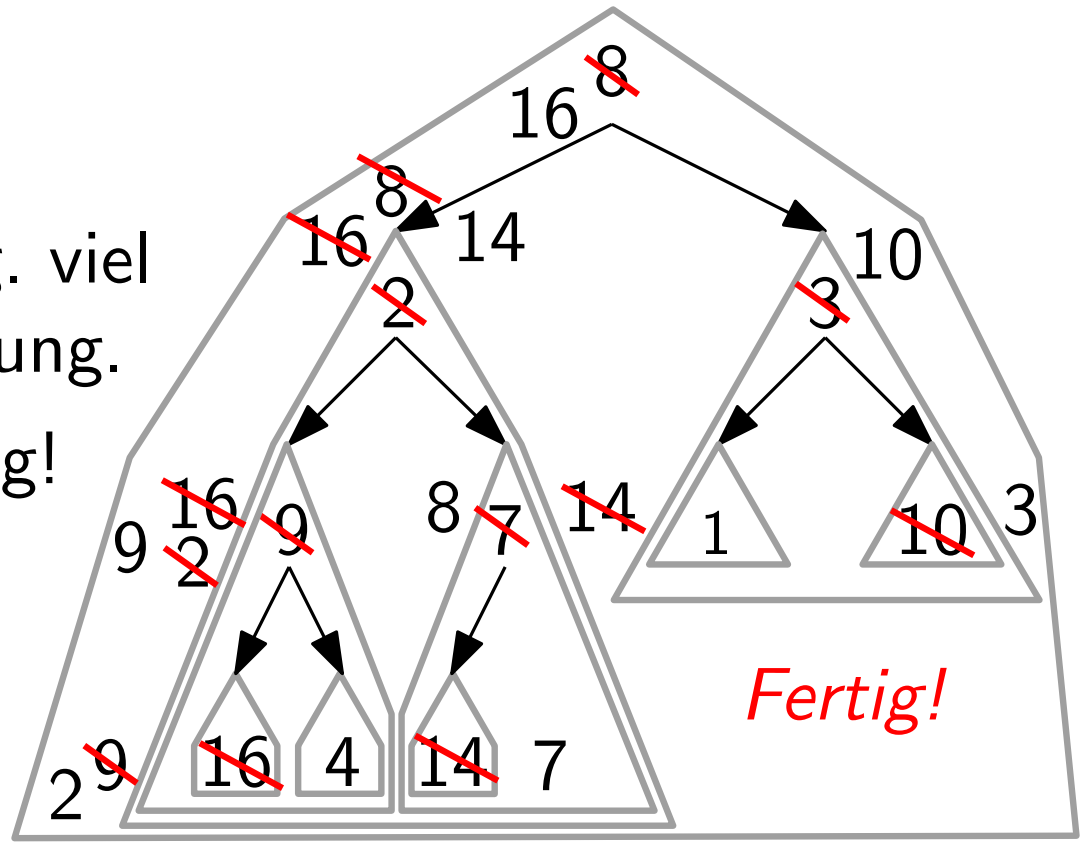
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

16	14	10	9	8	7	4	3	2	1

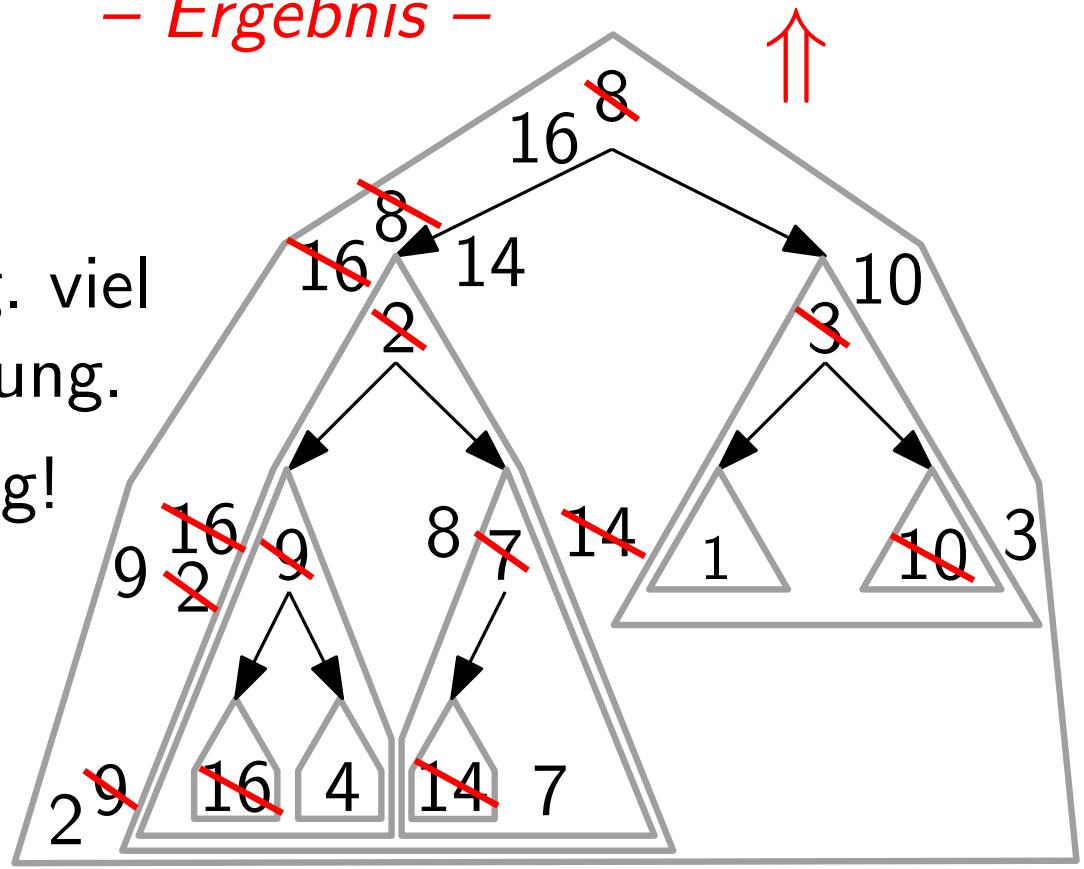
Absteigende Sortierung

– Ergebnis –

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

↓

16	14	10	9	8	7	4	3	2	1

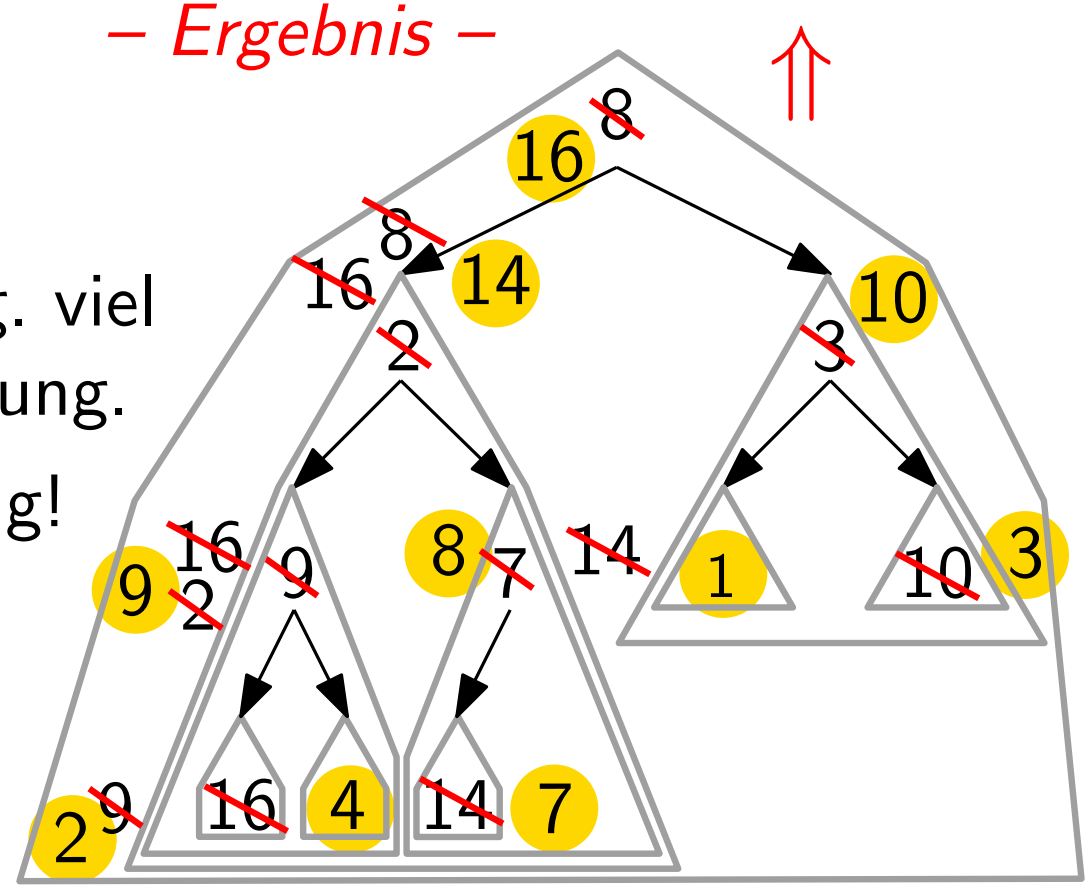
Absteigende Sortierung

– Ergebnis –

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

– Ergebnis –

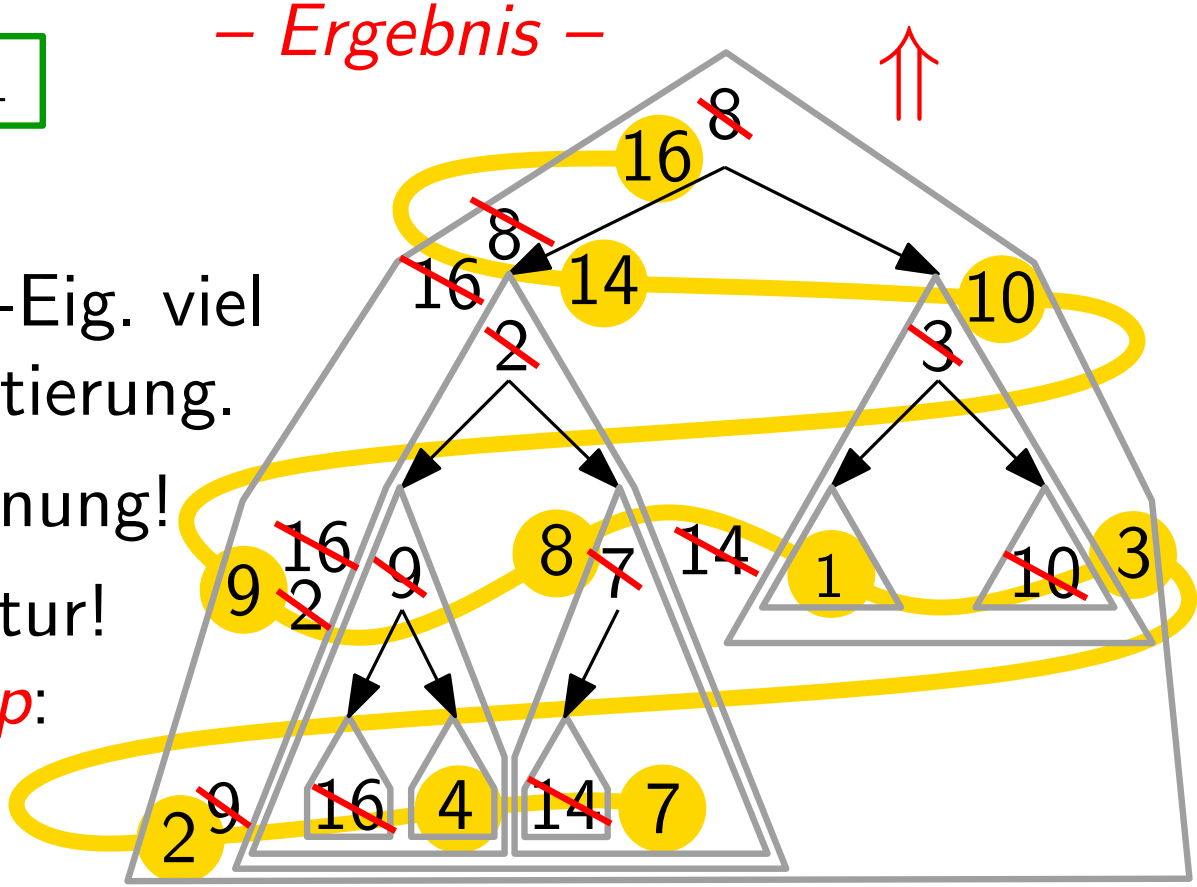
16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle

8	2	3	9	7	1	10	16	4	14

„totales Chaos“

→

16	14	10	8	7	3	9	2	1	4

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

16	14	10	9	8	7	4	3	2	1

Absteigende Sortierung

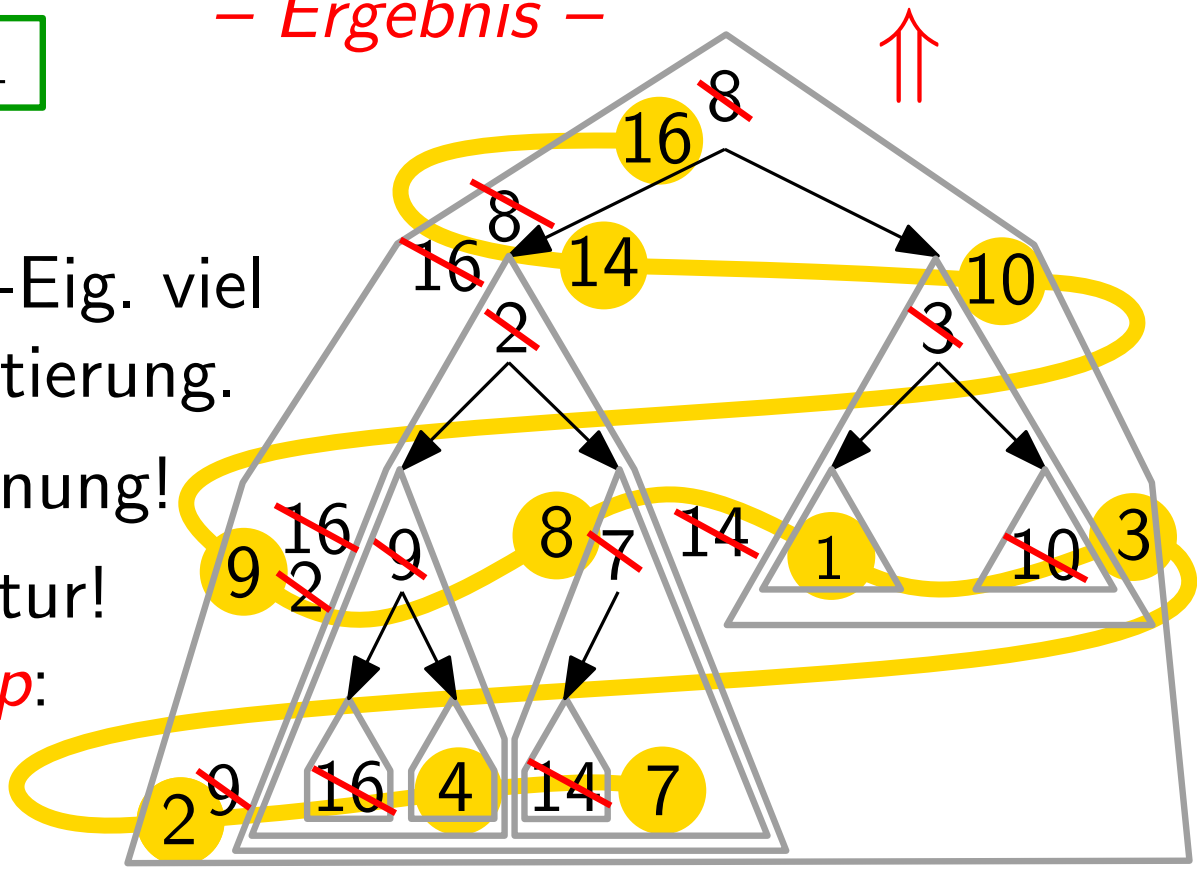
16	14	10	9	8	1	3	2	4	7

– Ergebnis –

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

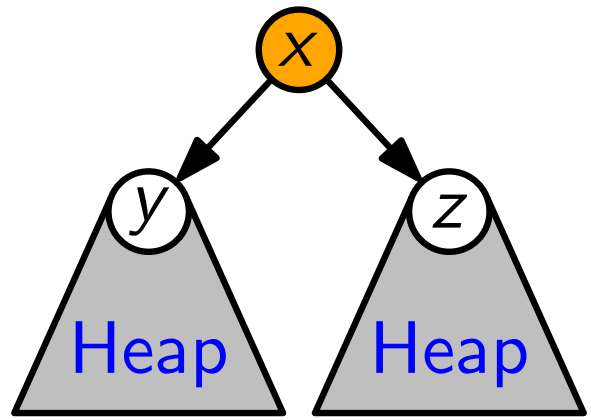
Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



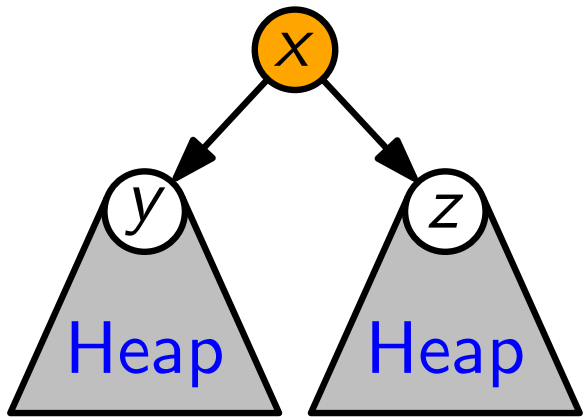
Elementaroperation

„Versickere“ x, falls x zu klein



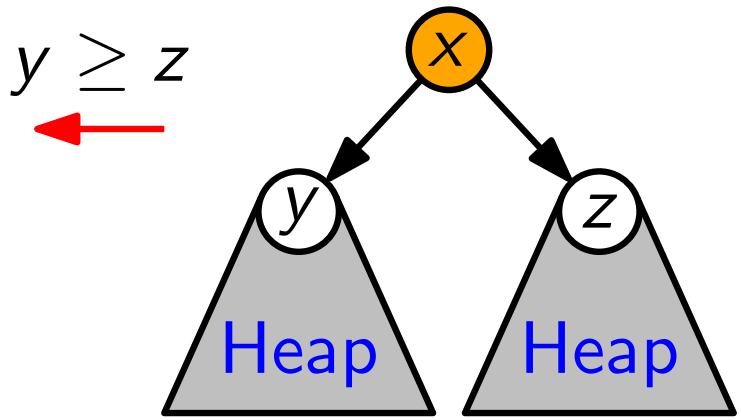
Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



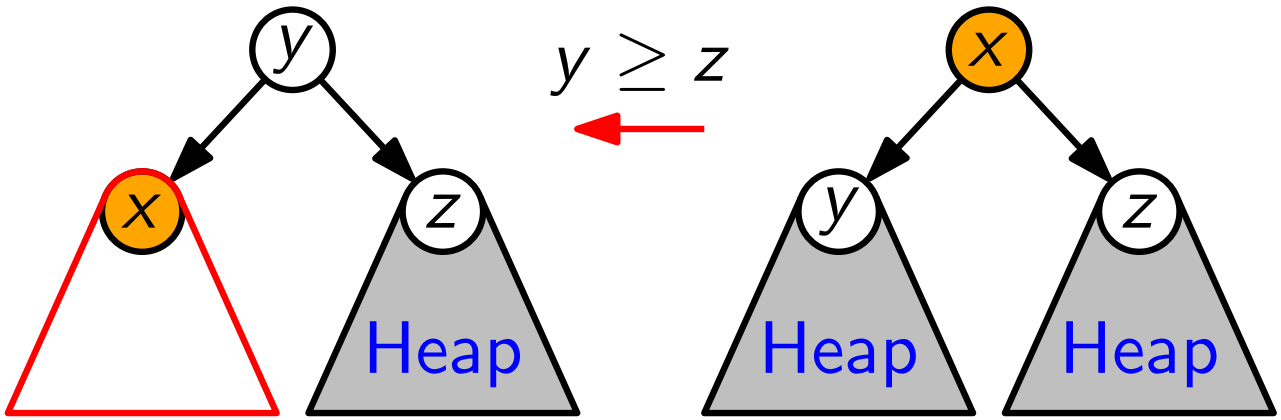
Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



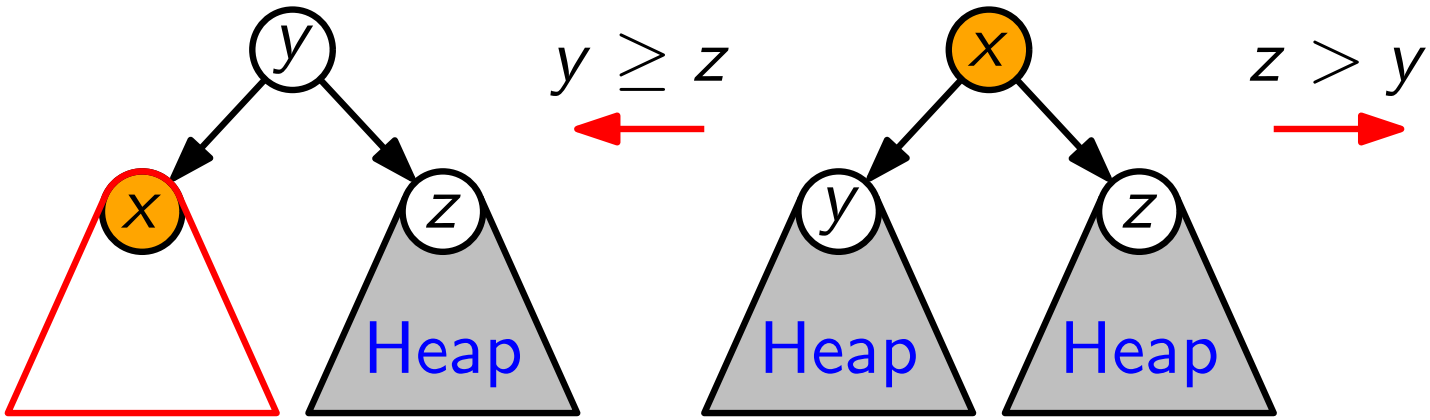
Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



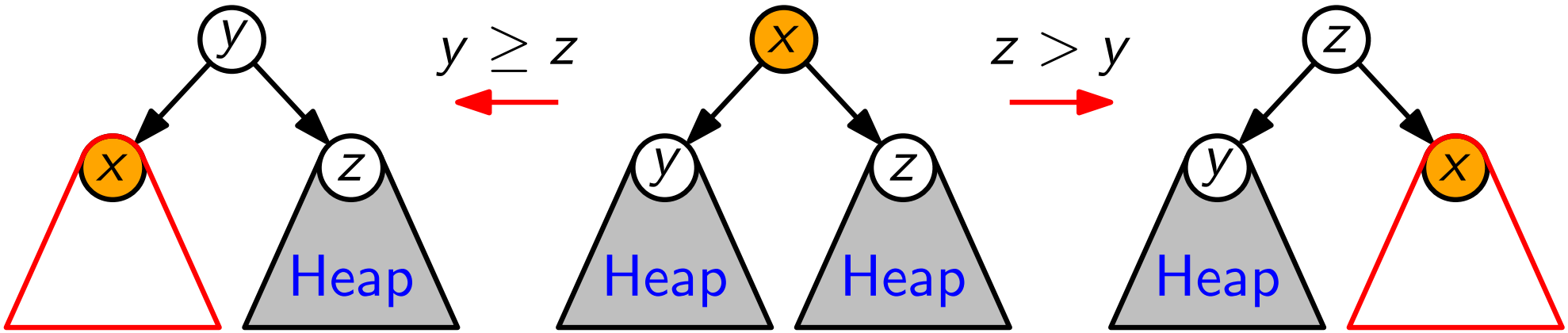
Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



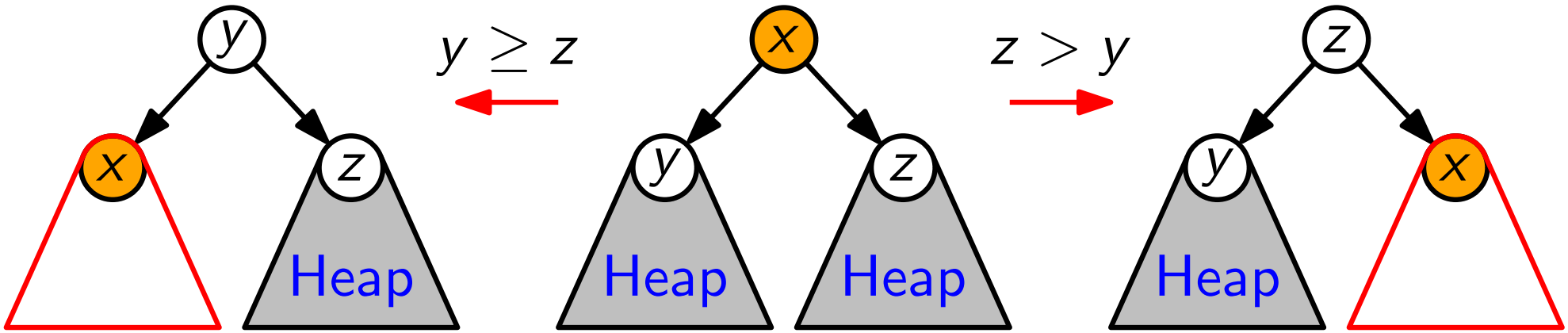
Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



Elementaroperation

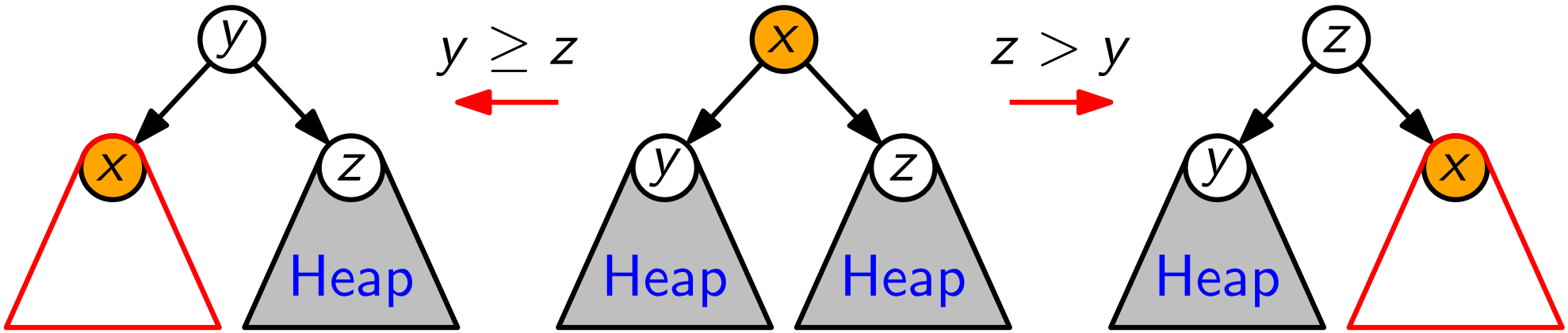
„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$

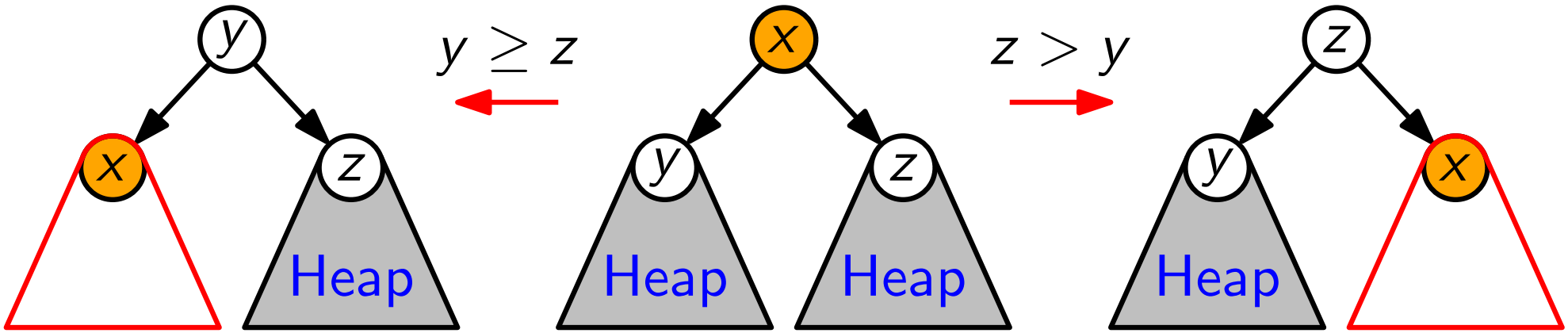



```
MaxHeapify(int A[ ], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



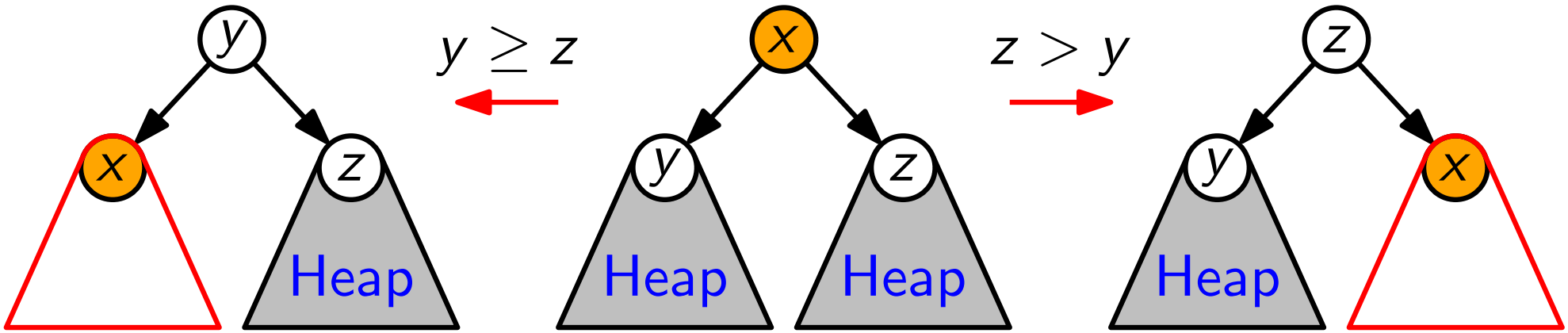

```
MaxHeapify(int A[ ], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit?

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$



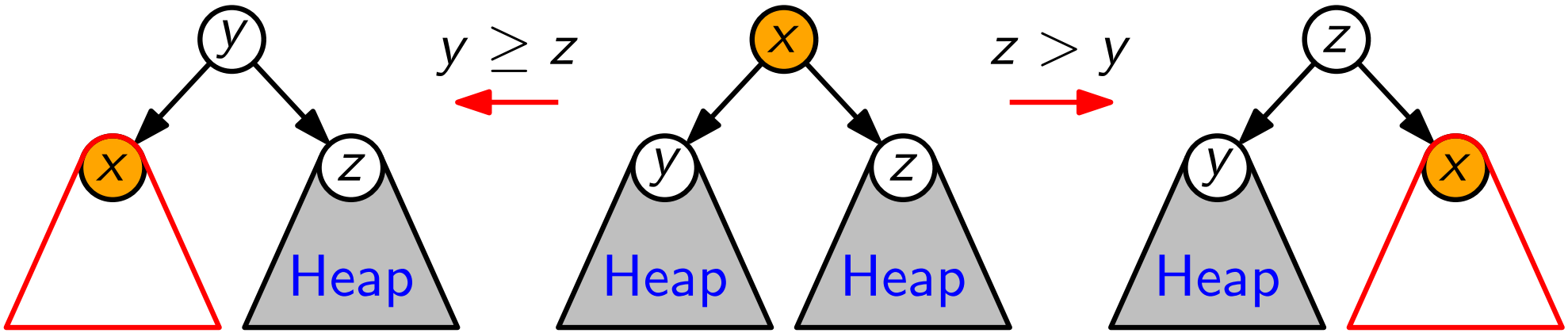
MaxHeapify(int $A[\,]$, index i)
  $\ell = \text{left}(i)$; $r = \text{right}(i)$
  if $\ell \leq A.heap\text{-}size$ **and** $A[\ell] > A[i]$ **then**
    $largest = \ell$
  else $largest = i$
  if $r \leq A.heap\text{-}size$ **and** $A[r] > A[largest]$
    $largest = r$
  if $largest \neq i$ **then**
    swap$(A, i, largest)$
    MaxHeapify$(A, largest)$

Lokale Strategie: *top-down*

Laufzeit? $T_{\text{MH}}(n, i)$

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

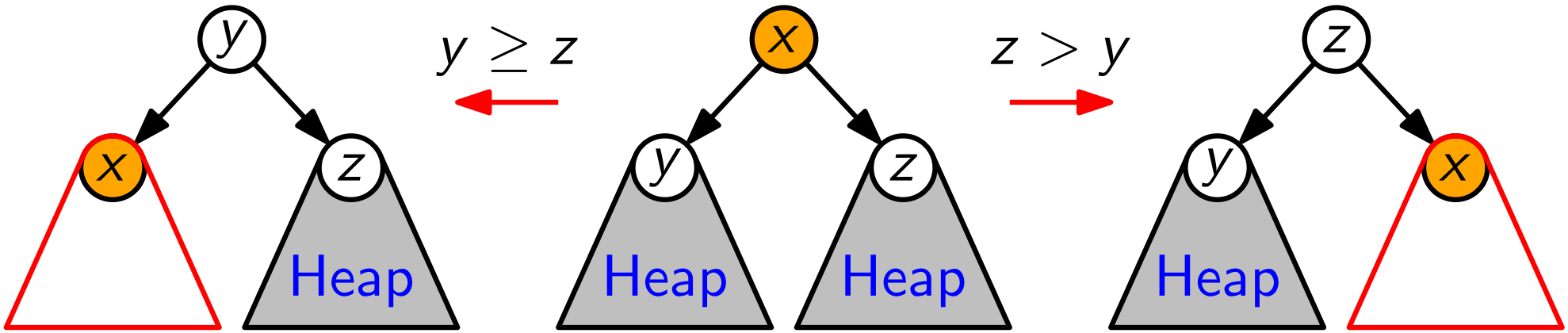
Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Swaps

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

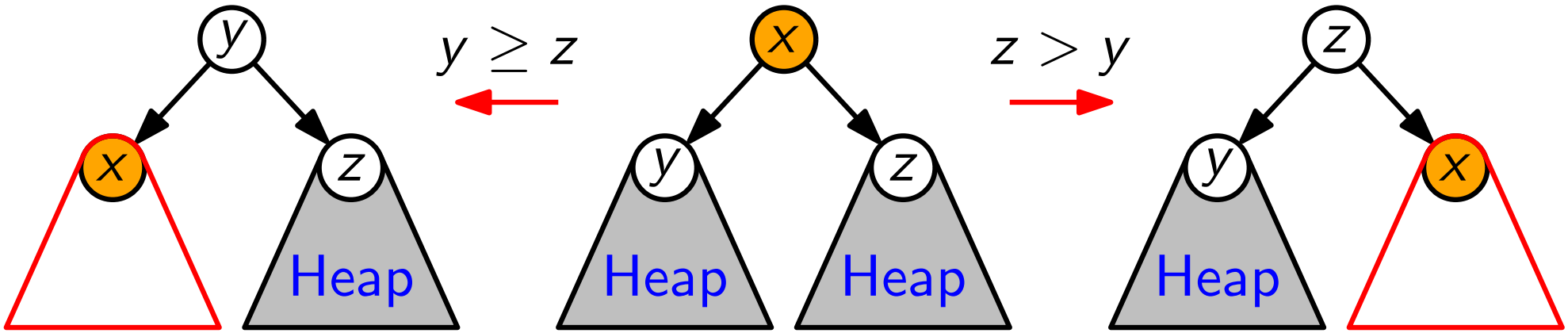
Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

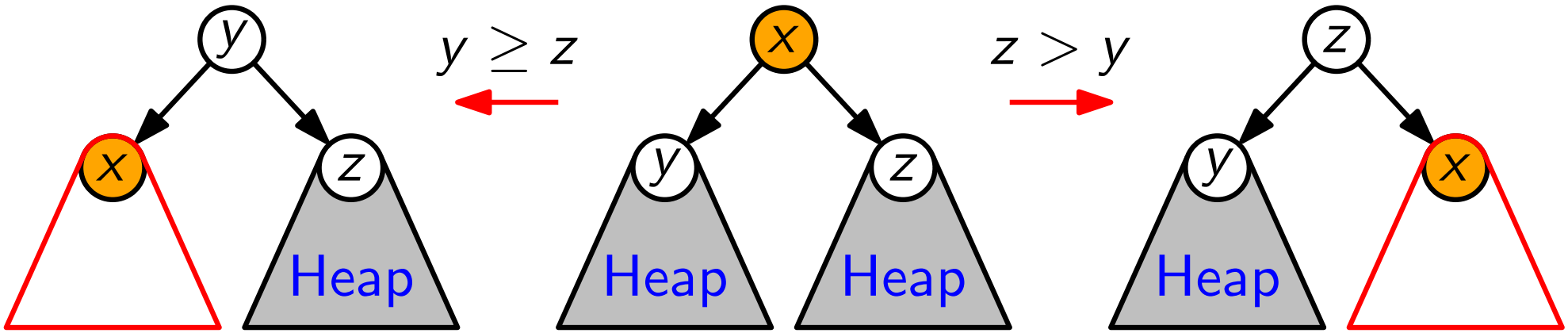
$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

$\leq$ Höhe von i im Heap

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Swaps

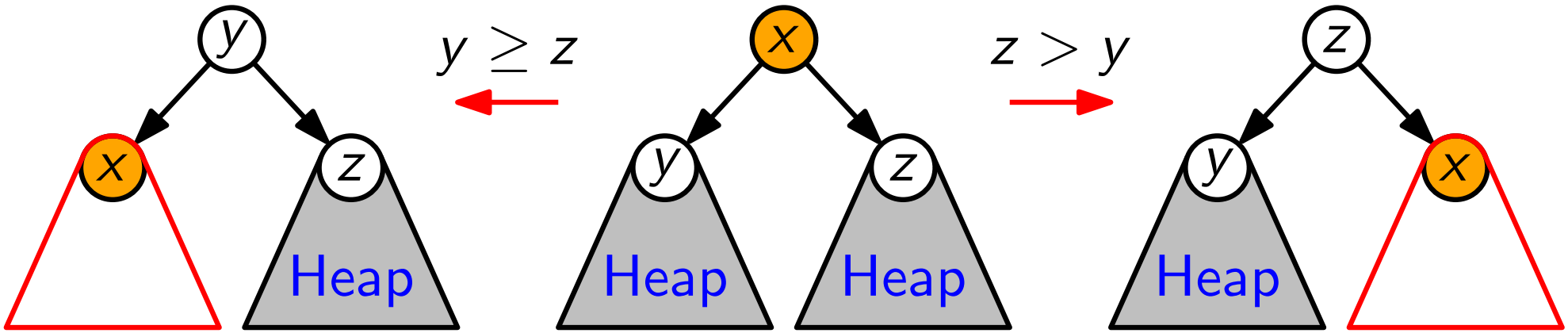
$\leq$ Länge des Weges von Knoten i zu einem Blatt

$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

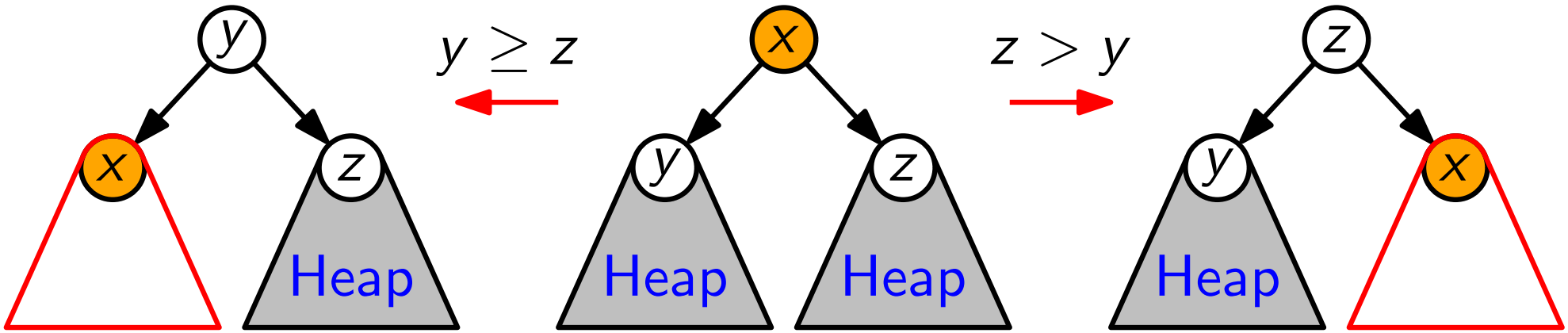
$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

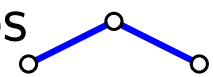
Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

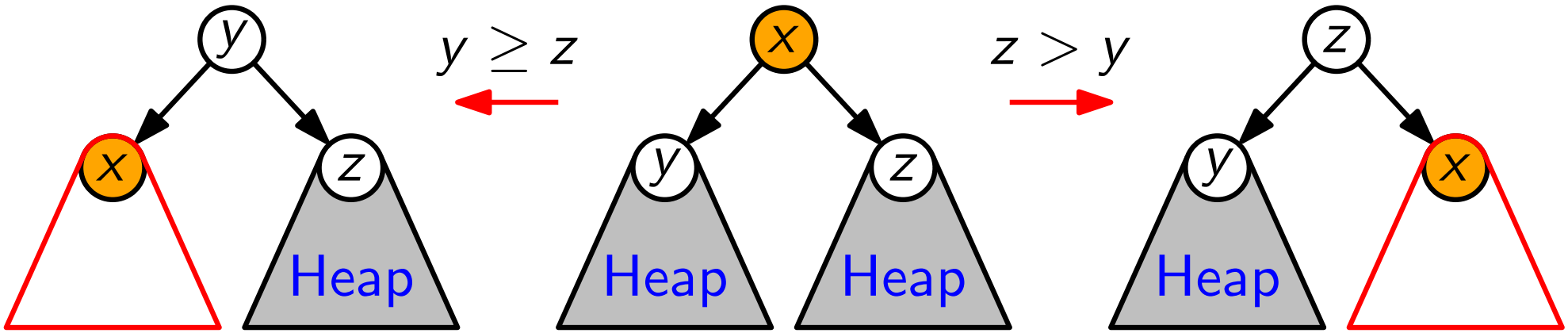
$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps 

$\leq \lfloor \log_2 n \rfloor$

Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

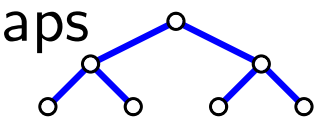
$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

$\leq$ Höhe von i im Heap

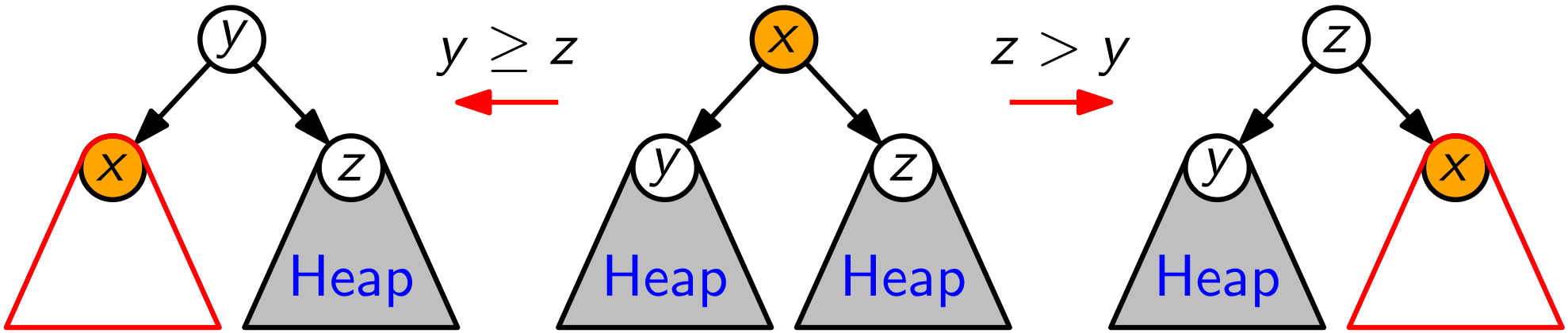
$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$



Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

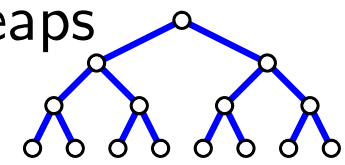
$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

$\leq$ Höhe von i im Heap

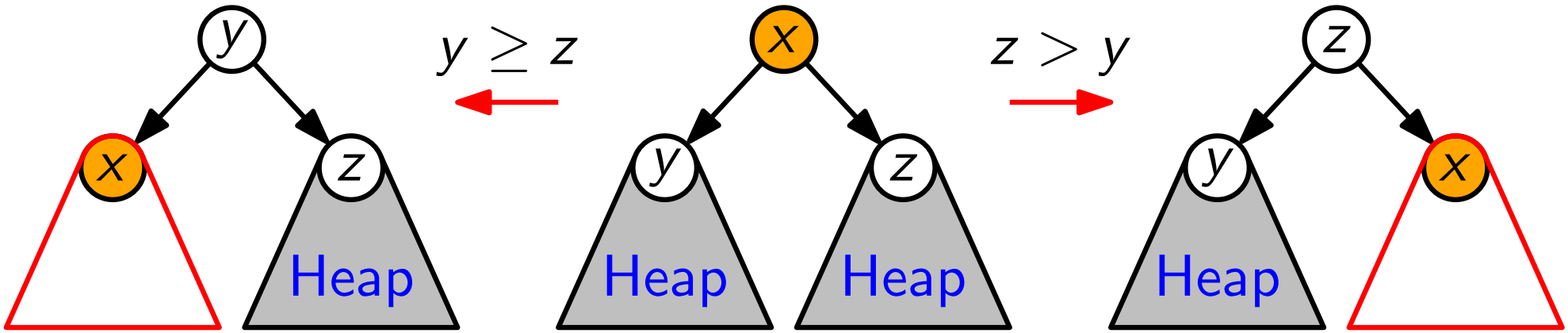
$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$



Elementaroperation

„Versickere“ x, falls x zu klein, d.h. falls $x < \max(y, z)$




```
MaxHeapify(int A[ ], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Lokale Strategie: *top-down*

Laufzeit? $T_{\mathrm{MH}}(n, i)$

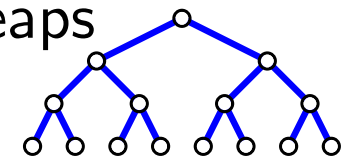
$:=$ Anzahl der Swaps

$\leq$ Länge des Weges von Knoten i zu einem Blatt

$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$



Das große Ganze

Lokale Strategie: *top-down*

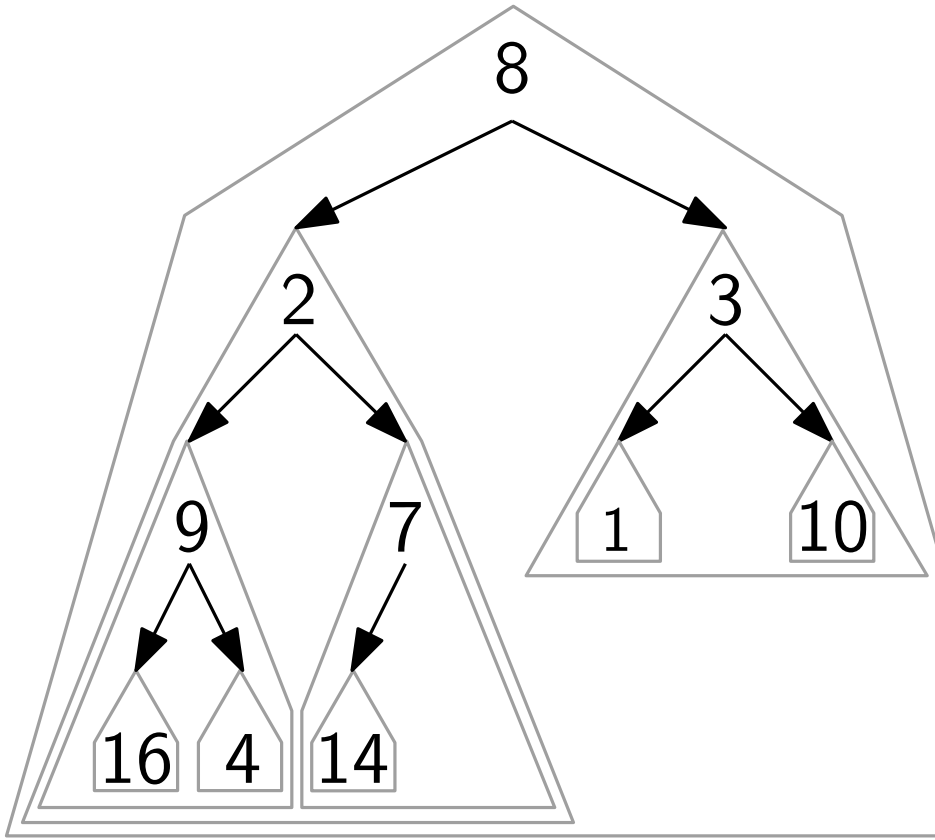
Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*



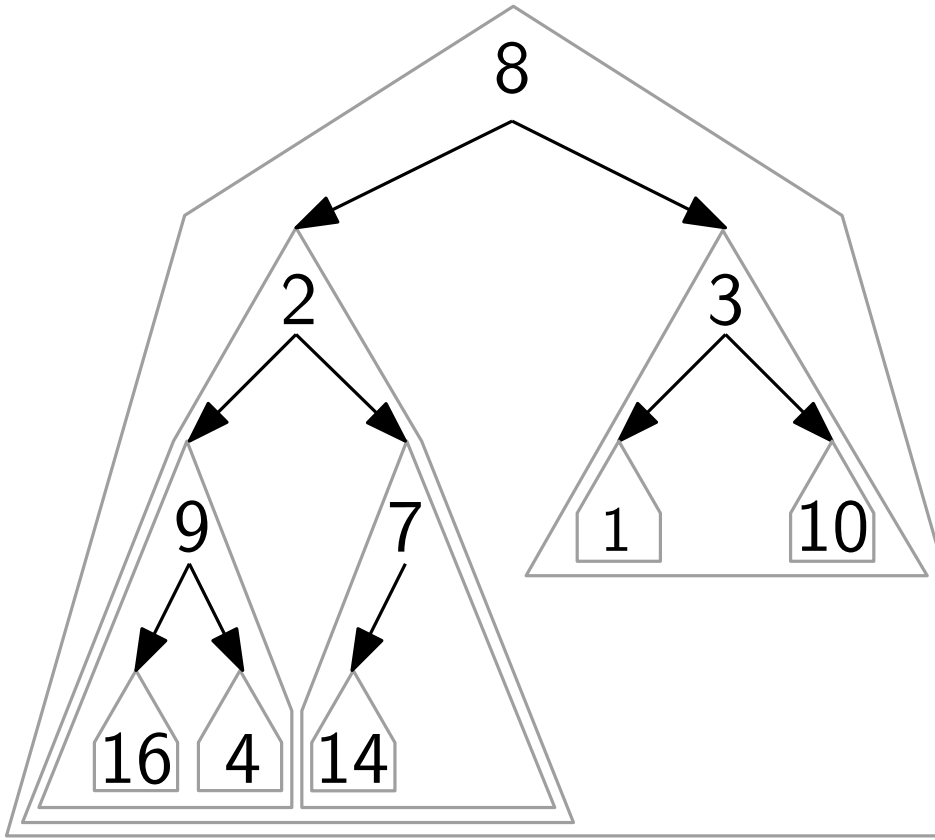
Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```



Das große Ganze

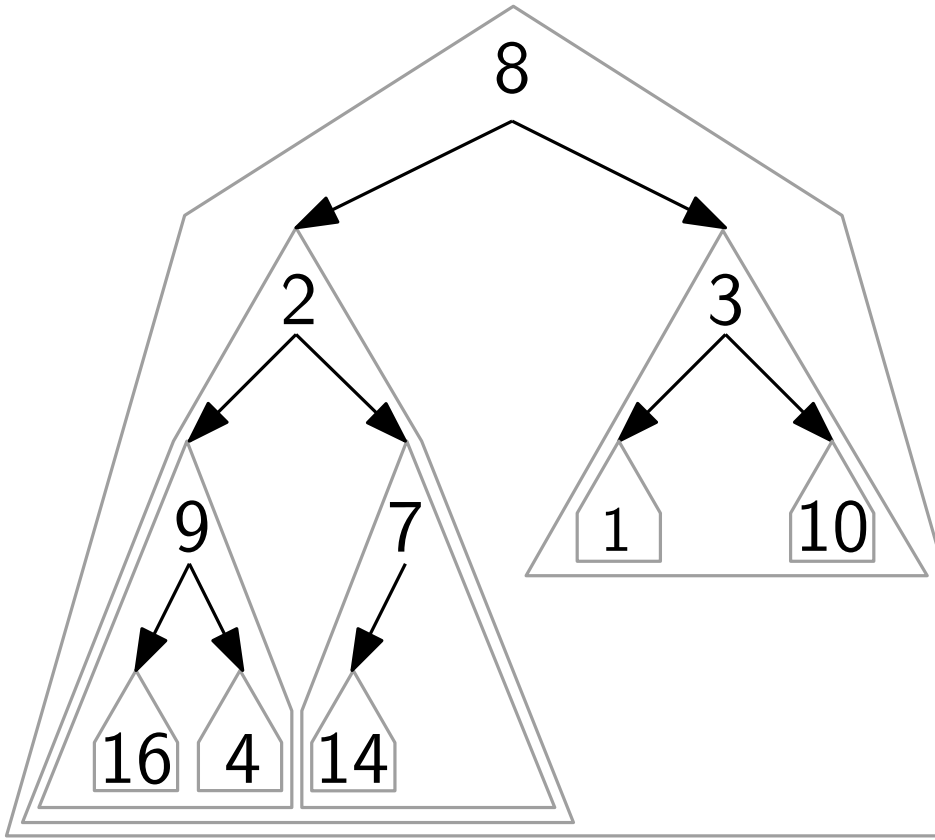
Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit.



Das große Ganze

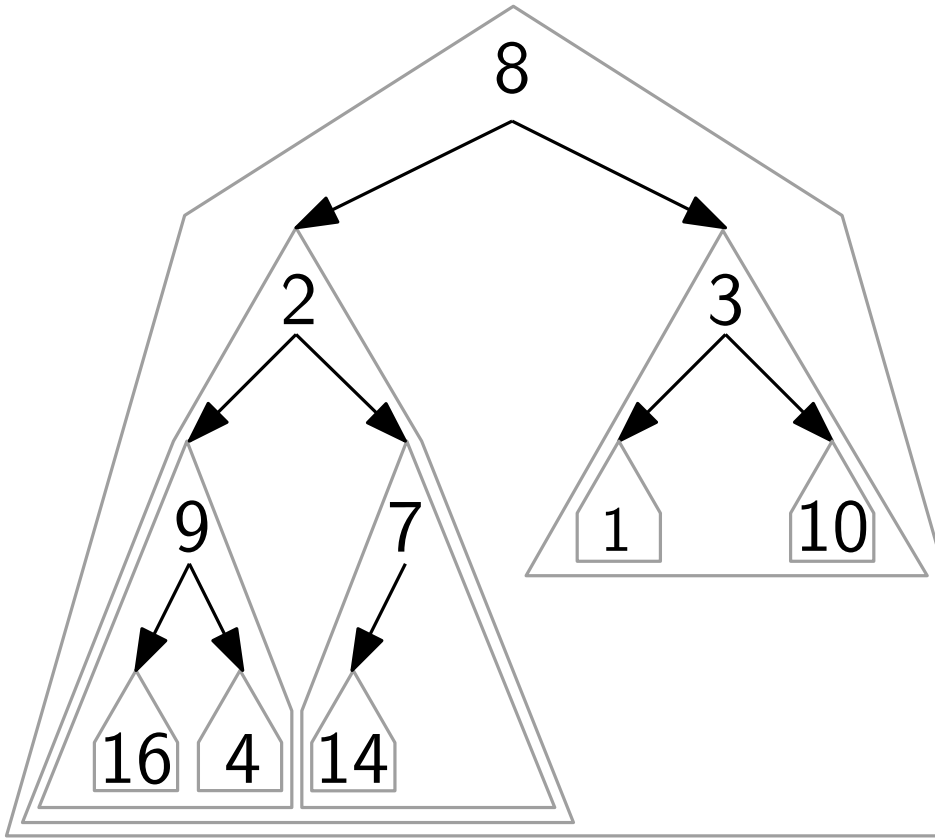
Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$



Das große Ganze

Lokale Strategie: *top-down*

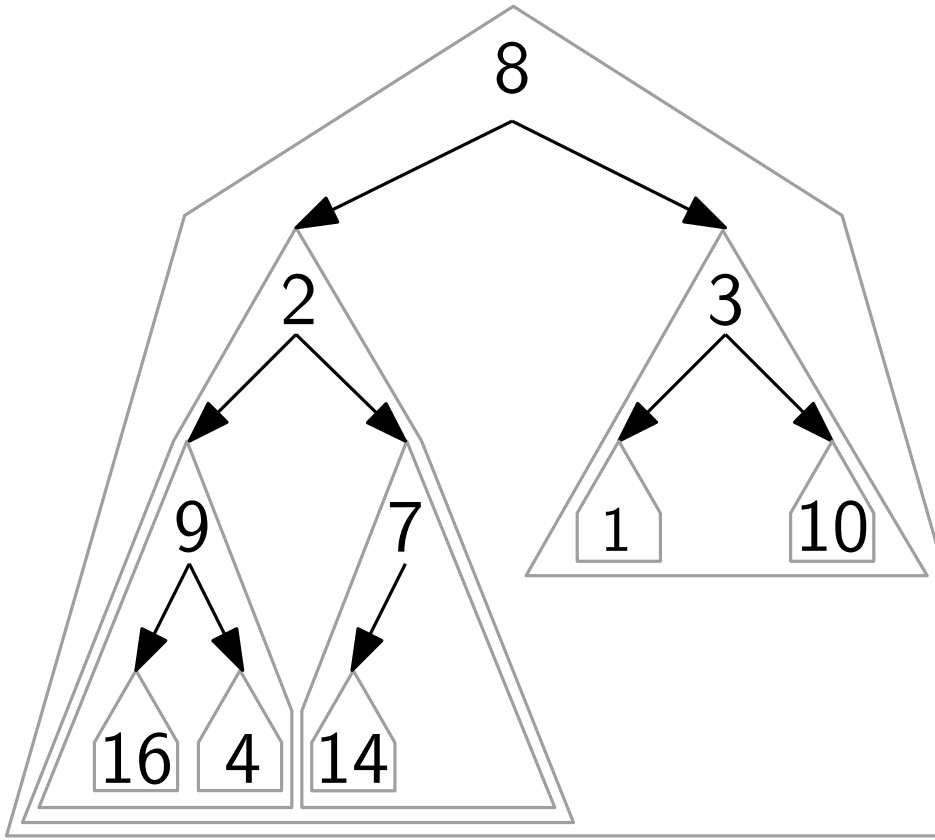
Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

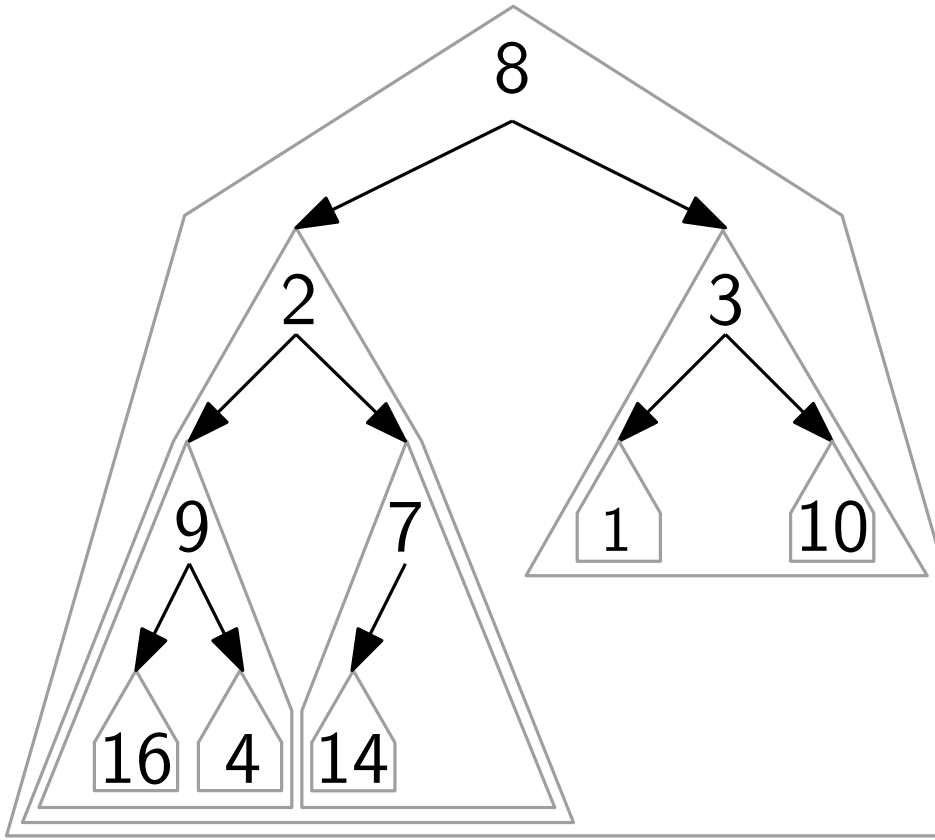
Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

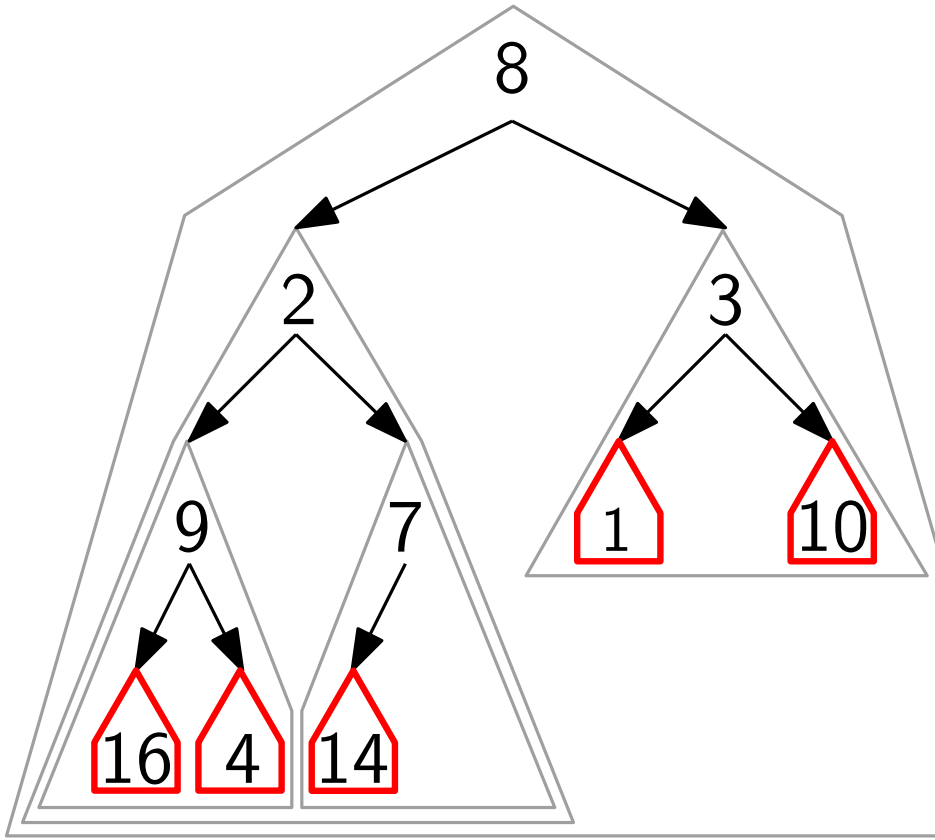
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

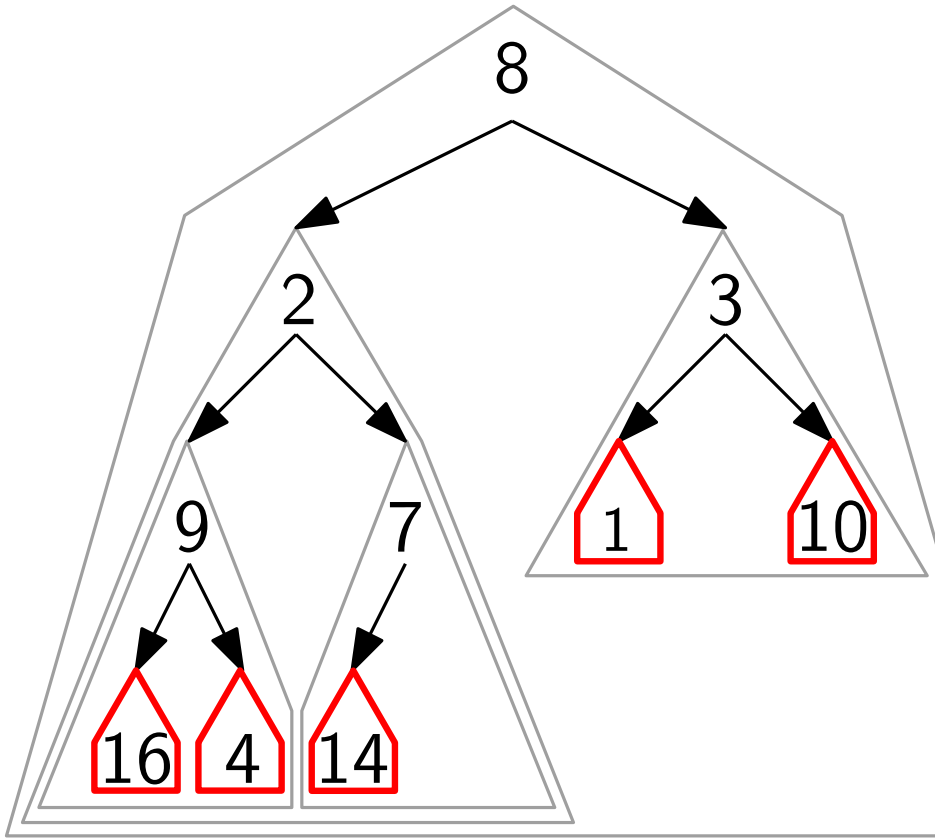
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 +$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

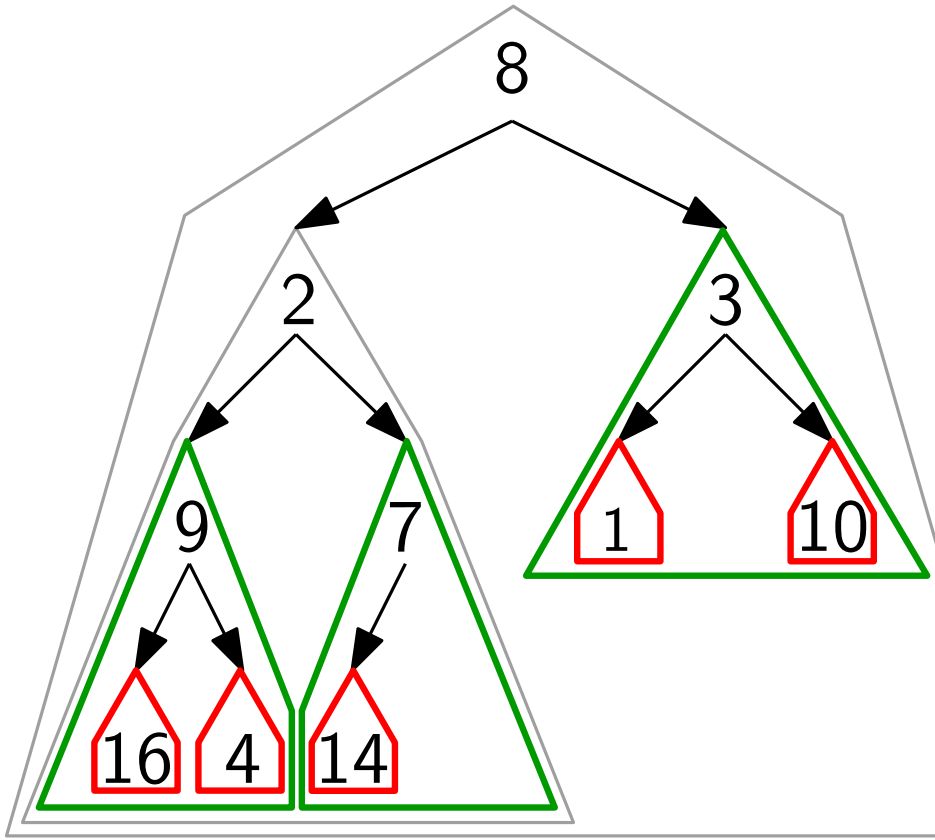
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 +$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

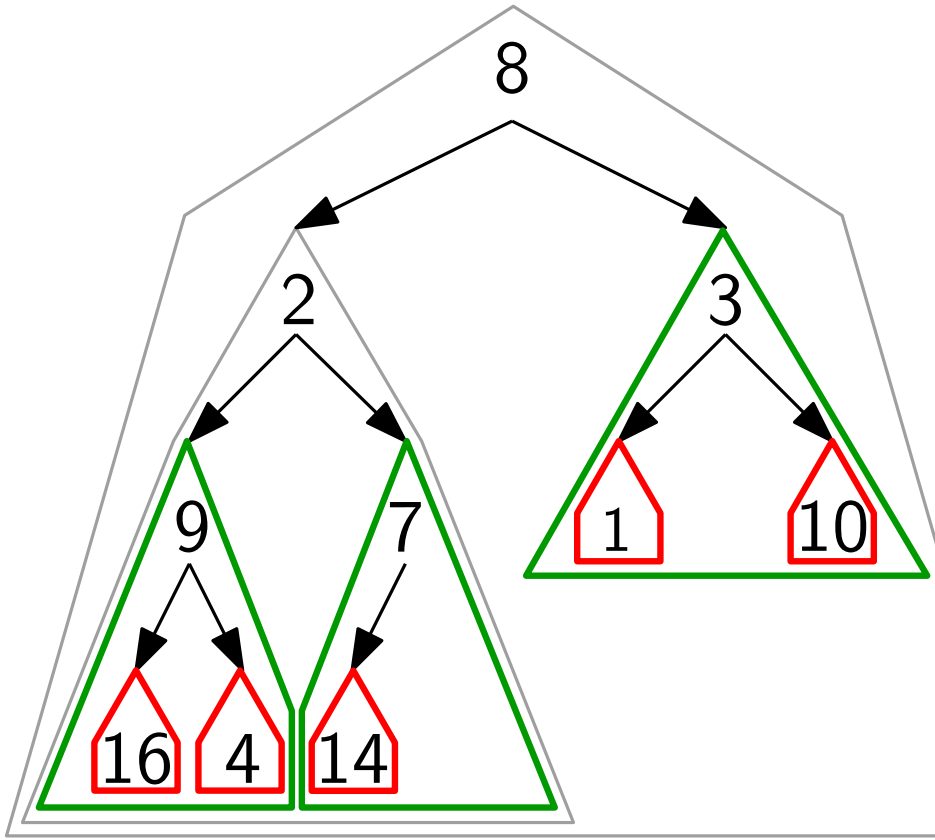
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 +$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

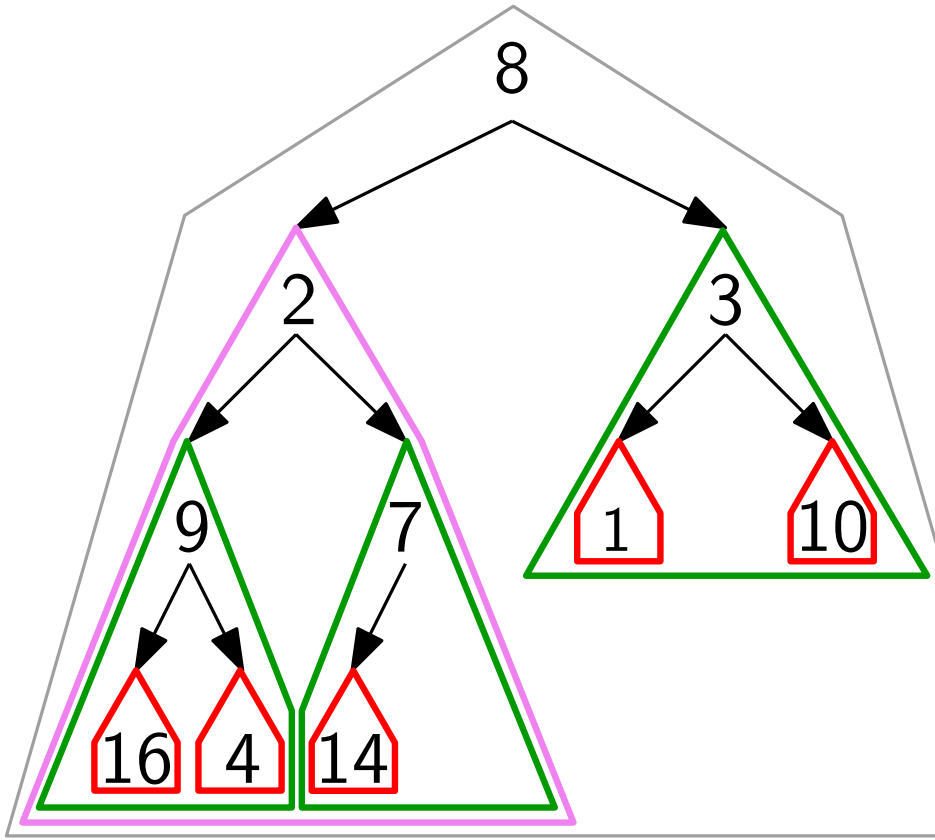
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 +$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

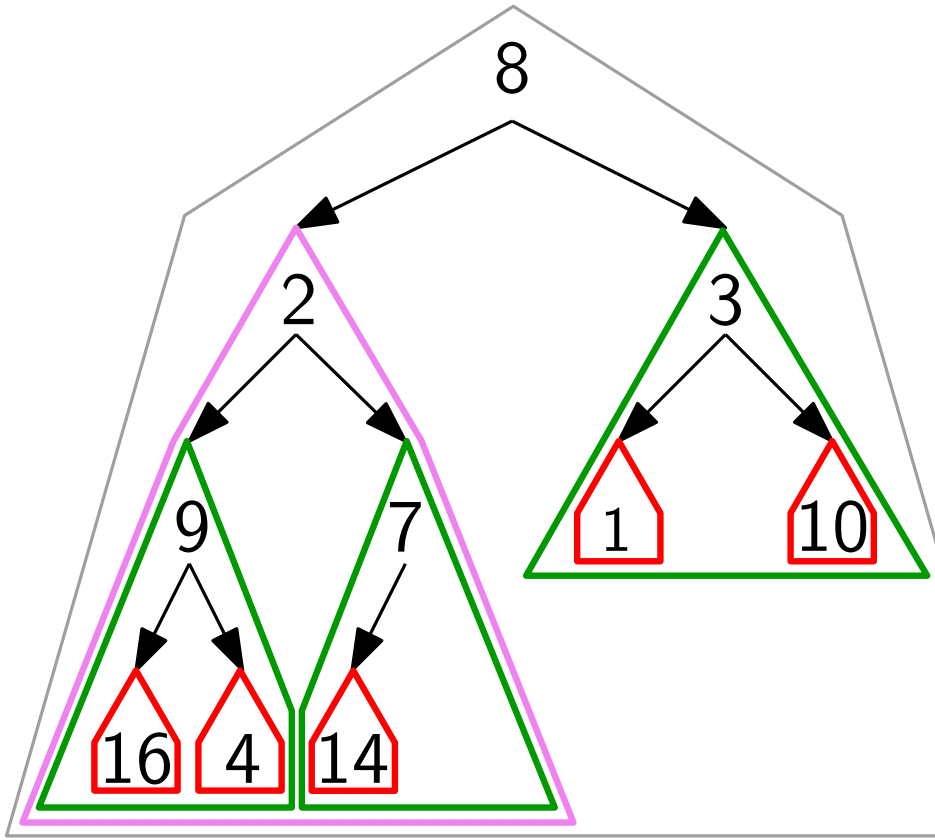
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 +$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

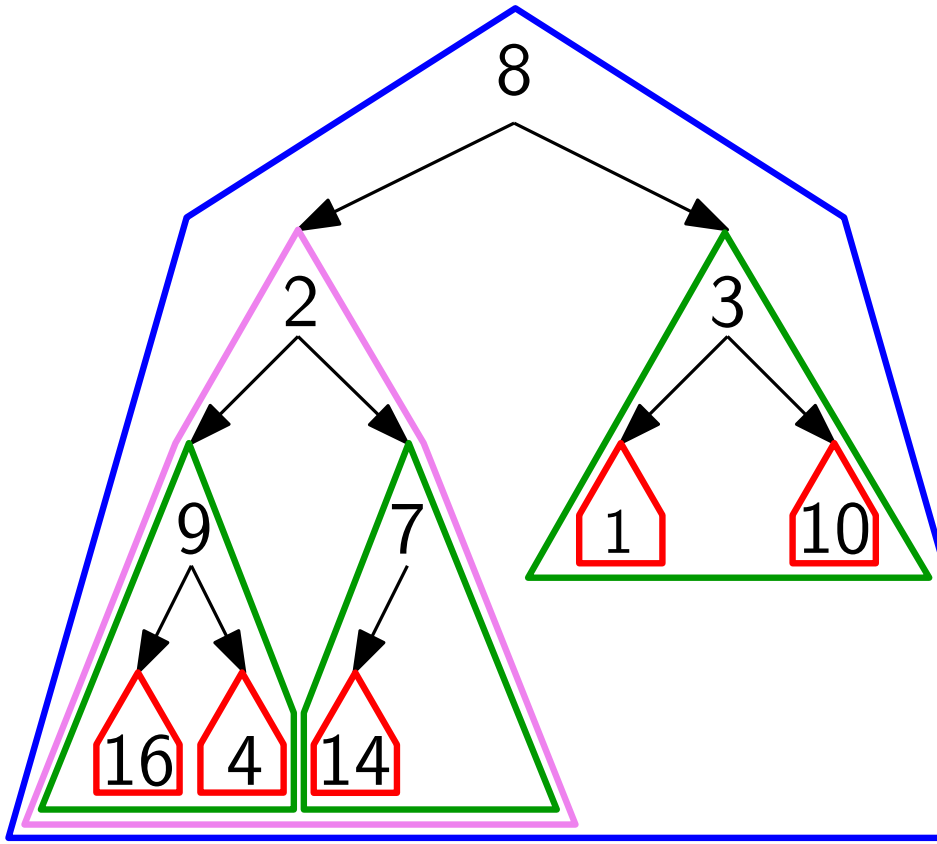
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 +$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathsf{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

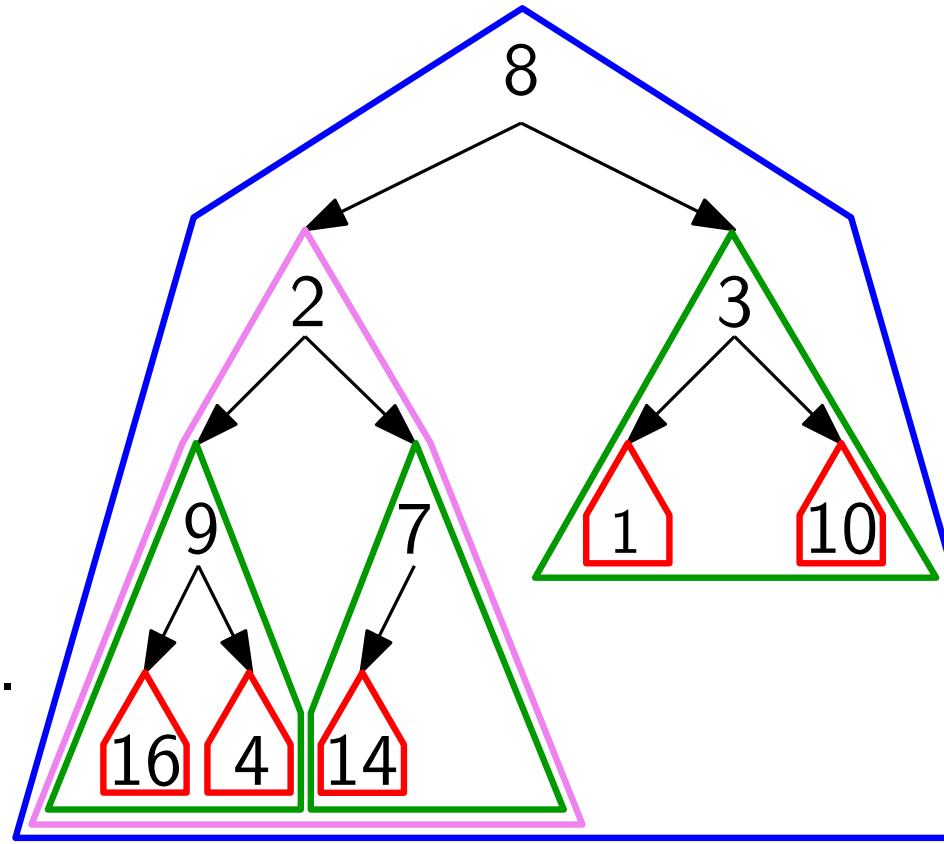
```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathsf{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathsf{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \ldots$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathrm{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

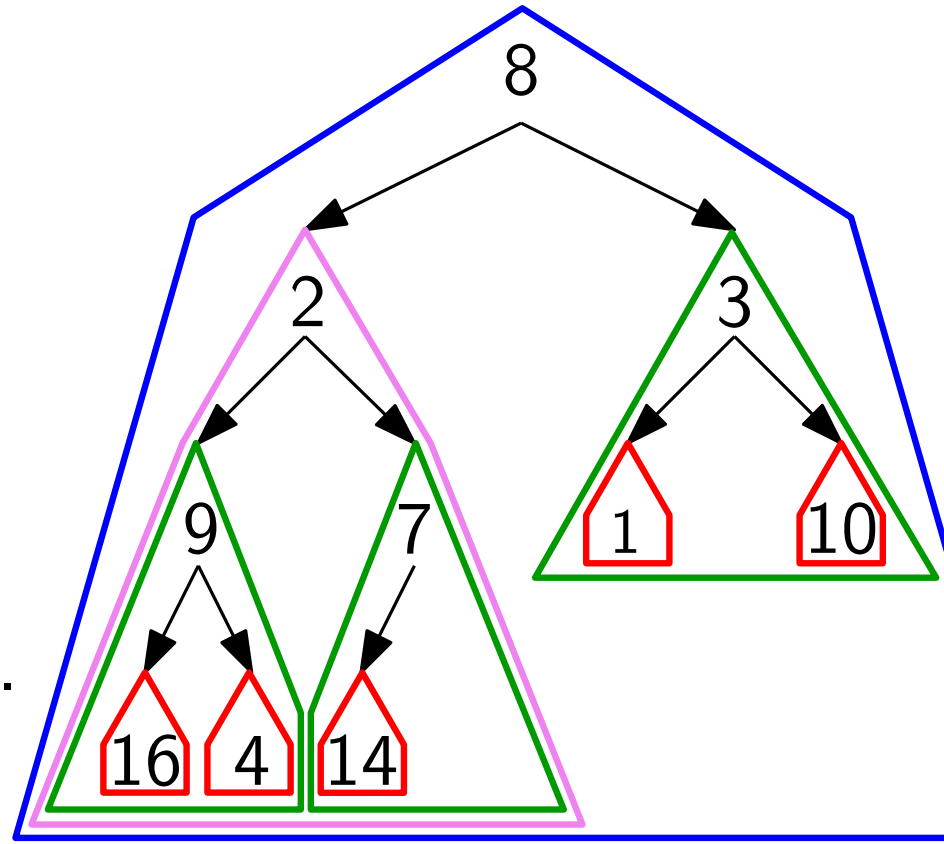
Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathrm{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \ldots$

$= n \sum_{i=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^{i+1} \cdot i$



Das große Ganze

Lokale Strategie: *top-down*

Laufzeit: $T_{\mathrm{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

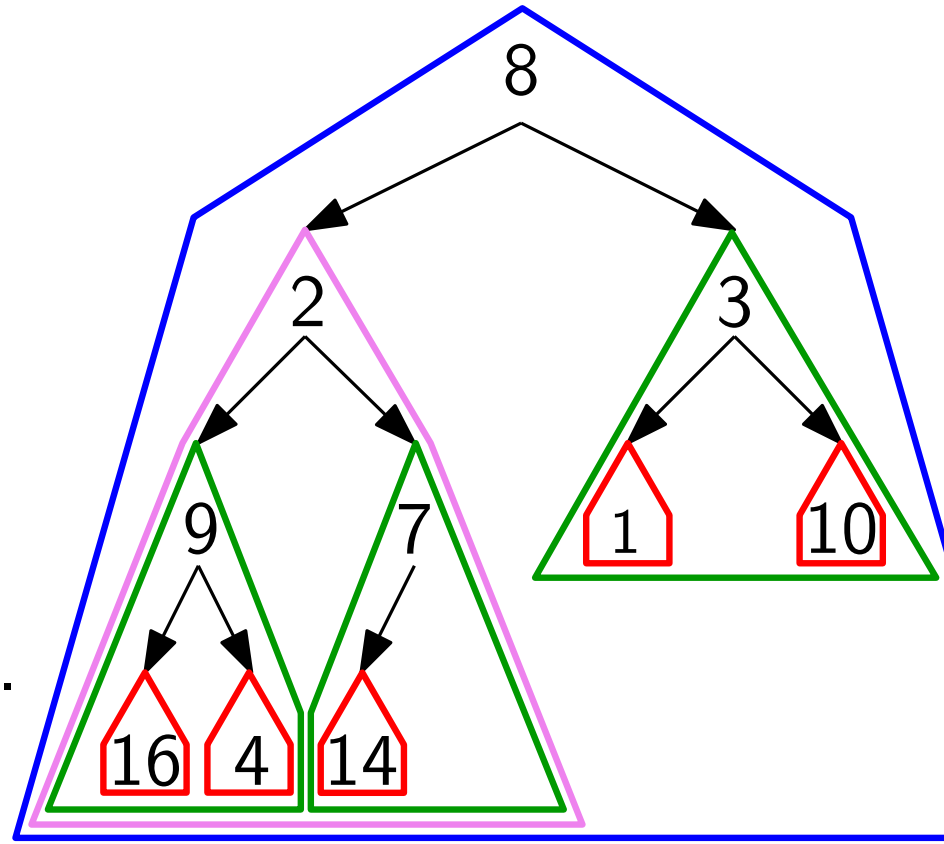
Laufzeit. grob: $O(n \log n)$

genauer: $T_{\mathrm{BMH}}(n) =$

$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n, i)$

$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \ldots$

$= n \sum_{i=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^{i+1} \cdot i \; = \; ?$



Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1}$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} = \tfrac{n}{4} \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\frac{1}{2}\right)^{i+1} \leq \frac{n}{4} \sum_{i=1}^{\overset{\infty}{\cancel{\lfloor \log n \rfloor}}} i \cdot \left(\frac{1}{2}\right)^{i-1}$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

Wir hätten gerne:

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$ **?**

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$ **?**

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$ **?**

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$ **?**

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten! *ableiten!*

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$ **?**

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \frac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} =$ **?**

ableiten!

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten!

ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten! *ableiten!*

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

$$\Rightarrow \quad T_{\mathrm{BMH}}(n) \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten! *ableiten!*

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

$$\Rightarrow T_{\mathrm{BMH}}(n) \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1} =$$

Forts. Laufzeitanalyse

$$T_{\text{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten! *ableiten!*

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

$$\Rightarrow T_{\text{BMH}}(n) \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1} = \frac{n}{4} \cdot \frac{1}{\left(\frac{1}{2}\right)^2}$$

Forts. Laufzeitanalyse

$$T_{\text{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten! *ableiten!*

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

$$\Rightarrow T_{\text{BMH}}(n) \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1} = \frac{n}{4} \cdot \frac{1}{\left(\frac{1}{2}\right)^2} = n$$

Forts. Laufzeitanalyse

$$T_{\mathrm{BMH}}(n) \approx n \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\tfrac{1}{2}\right)^{i+1} \leq \tfrac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1}$$

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten! *ableiten!*

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

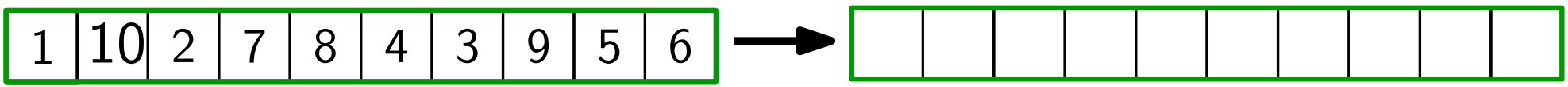
$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

$$\Rightarrow\ T_{\mathrm{BMH}}(n) \leq \tfrac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\tfrac{1}{2}\right)^{i-1} = \frac{n}{4} \cdot \frac{1}{\left(\frac{1}{2}\right)^2} = n$$

Satz. Ein Heap von n Elementen kann in $\Theta(n)$ Zeit berechnet werden.

Übung Heap-Aufbau

Aufgabe: Bauen Sie einen Heap mit BuildMaxHeap!

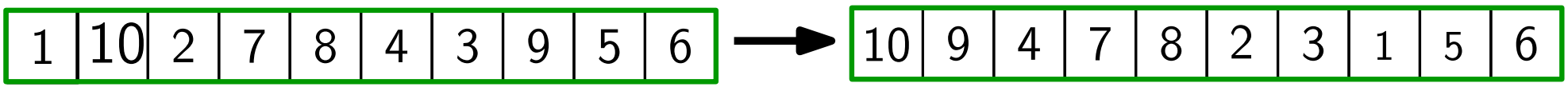



```
BuildMaxHeap(int A[])
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1 do
        MaxHeapify(A, i)
```

```
MaxHeapify(int A[], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Übung Heap-Aufbau

Aufgabe: Bauen Sie einen Heap mit BuildMaxHeap!




```
BuildMaxHeap(int A[])
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1 do
        MaxHeapify(A, i)
```

```
MaxHeapify(int A[], index i)
    ℓ = left(i); r = right(i)
    if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
        largest = ℓ
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i then
        swap(A, i, largest)
        MaxHeapify(A, largest)
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

```
return A[1]
```

ExtractMax()

```
if A.heap-size < 1 then
    error "Heap underflow"
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

```
return A[1]
```

ExtractMax()

```
if A.heap-size < 1 then
    error "Heap underflow"
max = A[1]
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"

$max = A[1]$

$A[1] = A[A.heap\text{-}size]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

```
return A[1]
```

ExtractMax()

```
if A.heap-size < 1 then
    error "Heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size − −
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"

$max = A[1]$

$A[1] = A[A.heap\text{-}size]$

$A.heap\text{-}size --$

MaxHeapify$(A, 1)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

```
return A[1]
```

ExtractMax()

```
if A.heap-size < 1 then
    error "Heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size − −
MaxHeapify(A, 1)
return max
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size --$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size --$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size --$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size --$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size --$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p)

$A.heap\text{-}size ++$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p)

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p)

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax()

return $A[1]$

ExtractMax()

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p)

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p)

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$
IncreaseKey$(A.heap\text{-}size, p)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax() $O(\quad)$

return $A[1]$

Laufzeiten?

ExtractMax() $O(\quad)$

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

IncreaseKey(index i, prio. p) $O(\quad)$

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p) $O(\quad)$

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$
IncreaseKey$(A.heap\text{-}size, p)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax() $O(1)$

return $A[1]$

ExtractMax() $O(\quad)$

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

Laufzeiten?

IncreaseKey(index i, prio. p) $O(\quad)$

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p) $O(\quad)$

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$
IncreaseKey$(A.heap\text{-}size, p)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax() $O(1)$

return $A[1]$

ExtractMax() $O(\log n)$

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

Laufzeiten?

IncreaseKey(index i, prio. p) $O(\quad)$

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p) $O(\quad)$

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$
IncreaseKey$(A.heap\text{-}size, p)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax() $O(1)$

return $A[1]$

ExtractMax() $O(\log n)$

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size - -$
MaxHeapify$(A, 1)$
return max

Laufzeiten?

IncreaseKey(index i, prio. p) $O(\log n)$

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p) $O(\quad)$

$A.heap\text{-}size + +$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$
IncreaseKey$(A.heap\text{-}size, p)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax() $O(1)$

return $A[1]$

ExtractMax() $O(\log n)$

if $A.heap\text{-}size < 1$ **then**
  error "Heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size --$
MaxHeapify$(A, 1)$
return max

Laufzeiten?

IncreaseKey(index i, prio. p) $O(\log n)$

if $p < A[i]$ **then error** "prio. too small"
$A[i] = p$
while $i > 1$ **and** $A[\text{parent}(i)] < A[i]$
  swap$(A, i, \text{parent}(i))$
  $i = \text{parent}(i)$

Insert(priorität p) $O(\log n)$

$A.heap\text{-}size ++$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = -\infty$
IncreaseKey$(A.heap\text{-}size, p)$

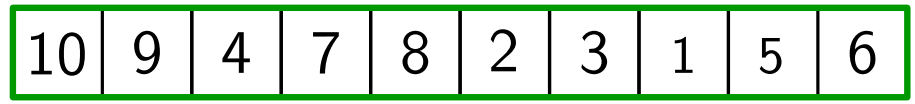
Vom Heap zur Sortierung

Idee:

Vom Heap zur Sortierung

Idee:

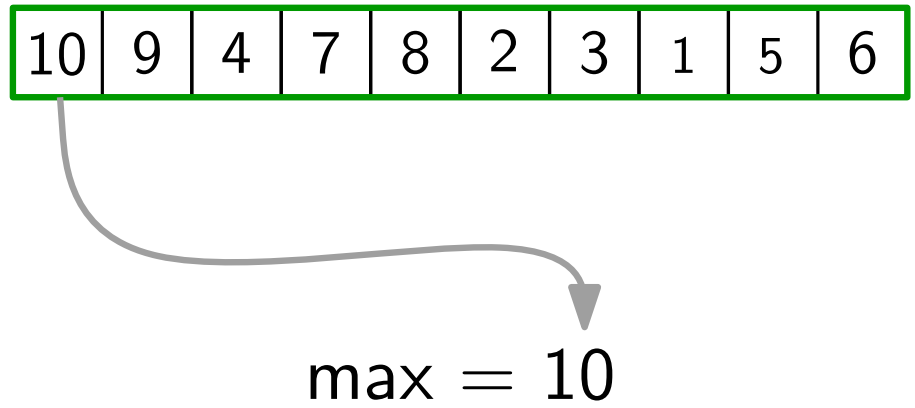
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

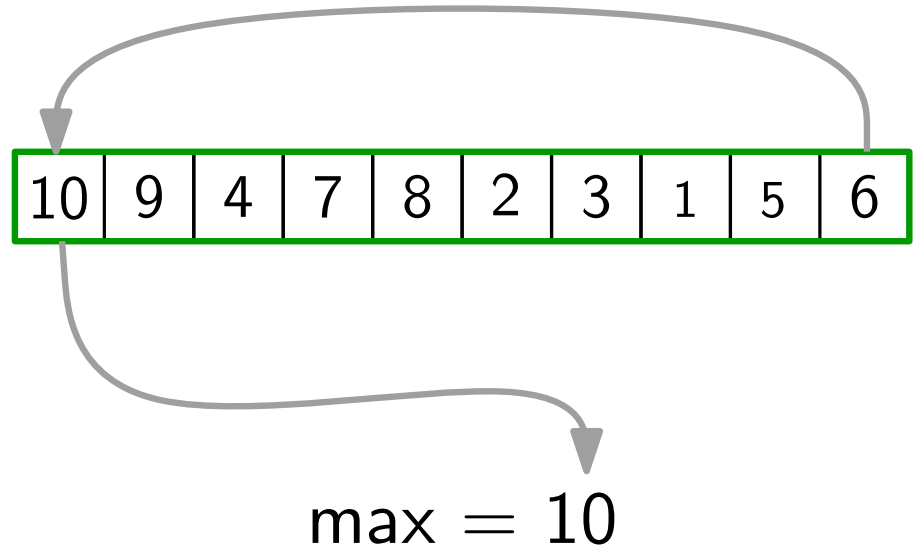
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

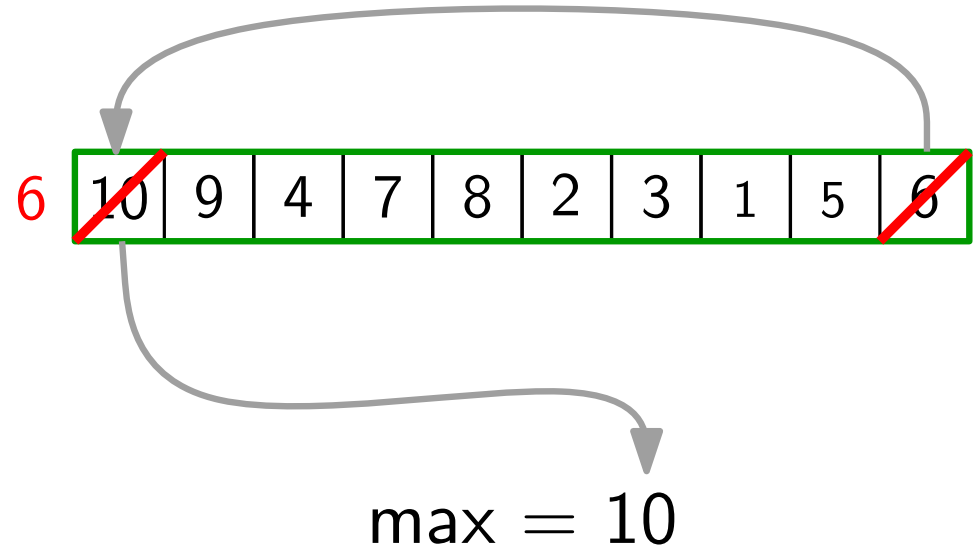
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

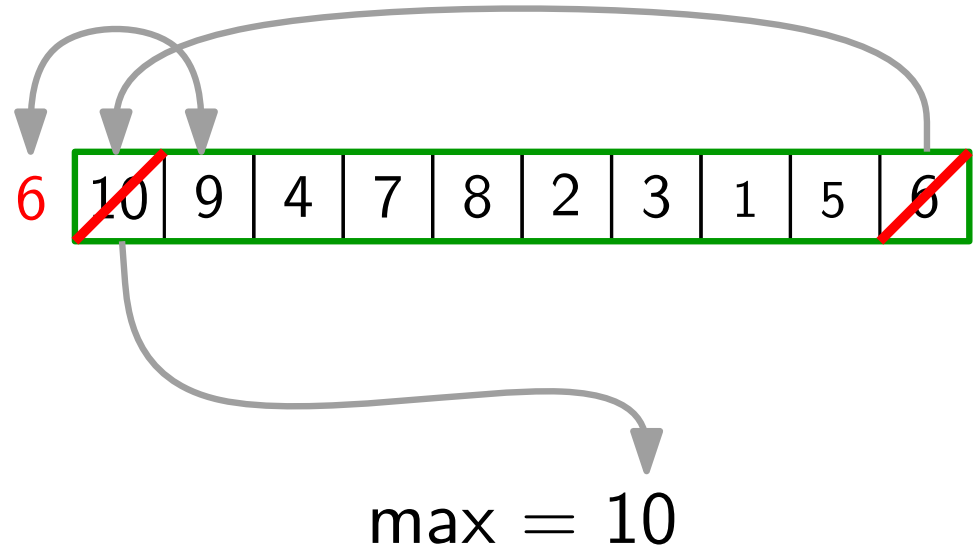
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

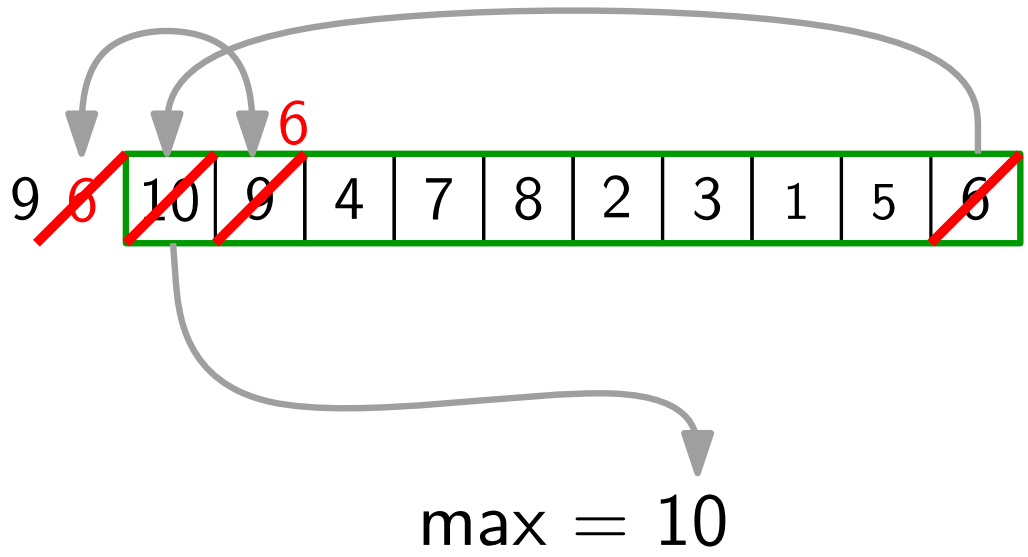
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

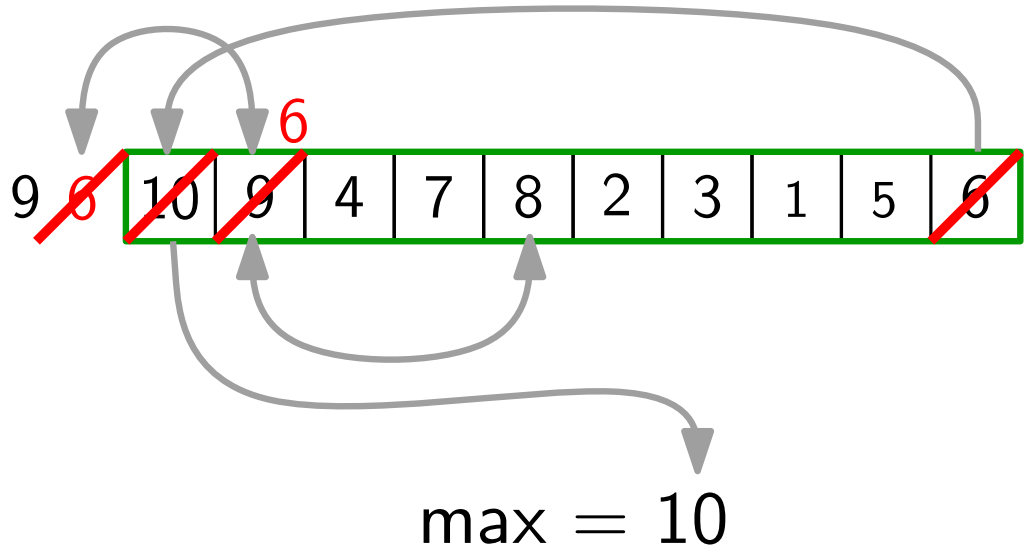
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

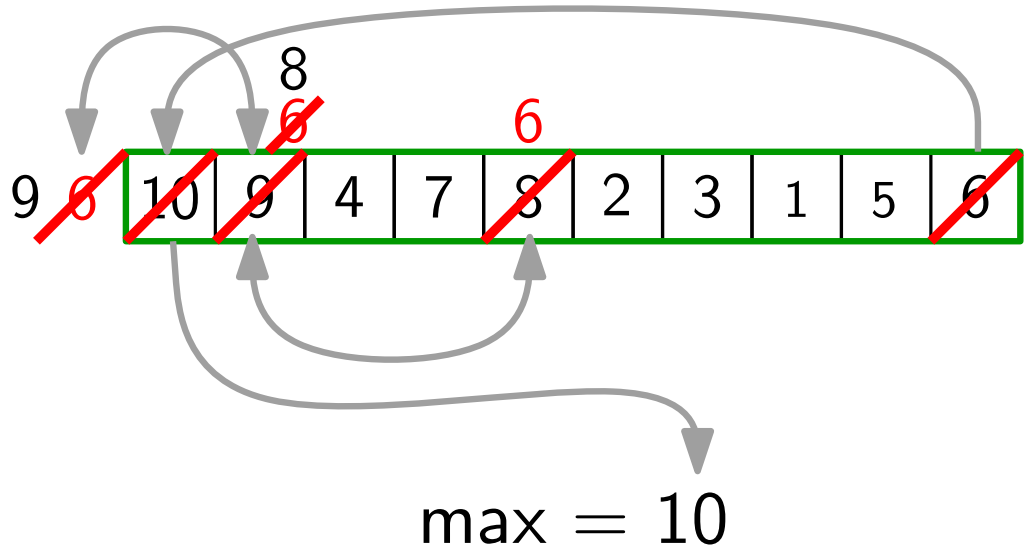
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

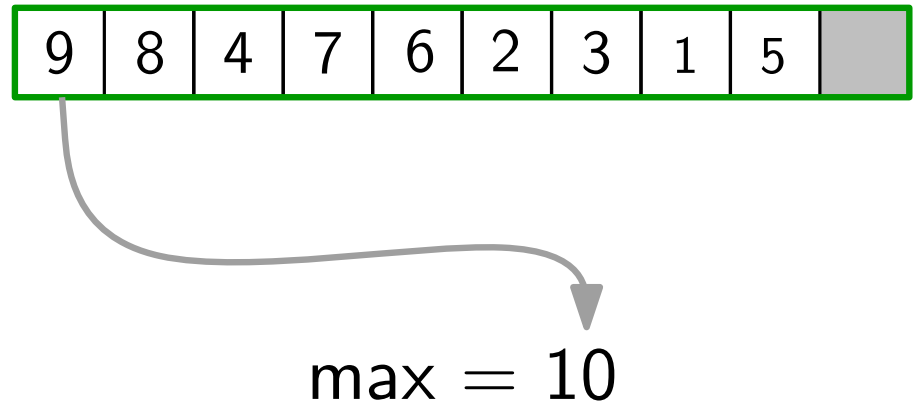
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

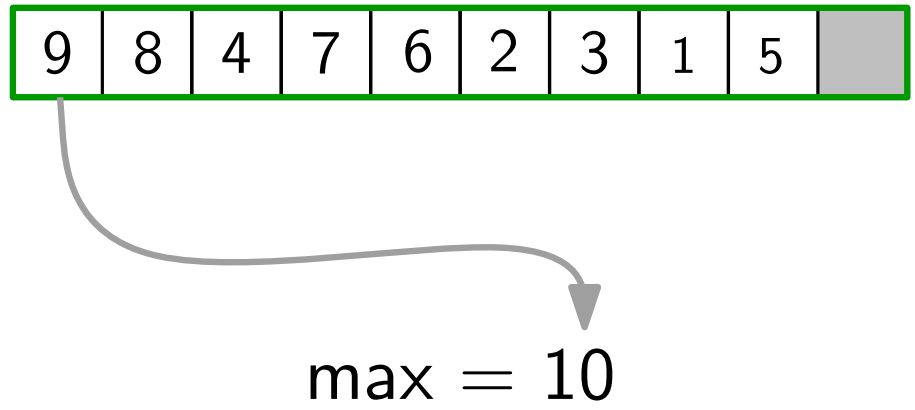
- ExtractMax() gibt rechtestes Heap-Element frei.



Vom Heap zur Sortierung

Idee:

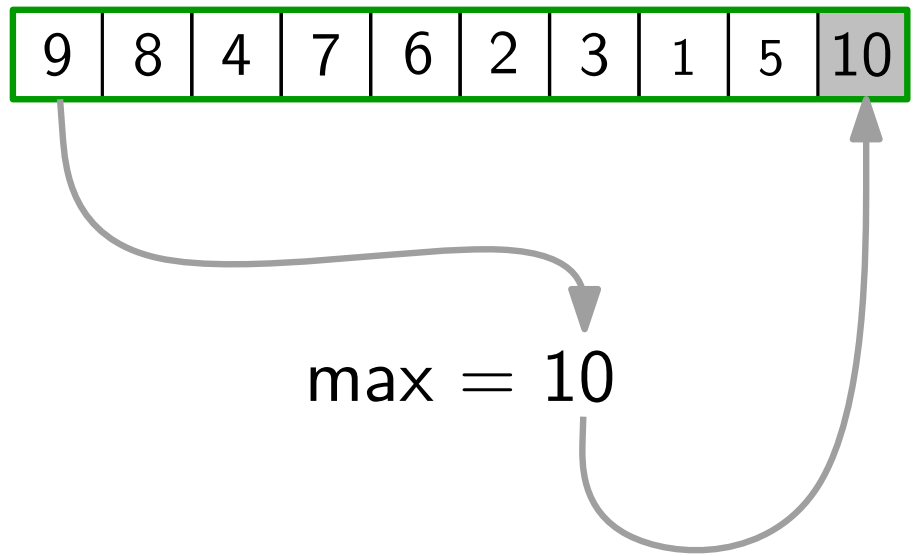
- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



Vom Heap zur Sortierung

Idee:

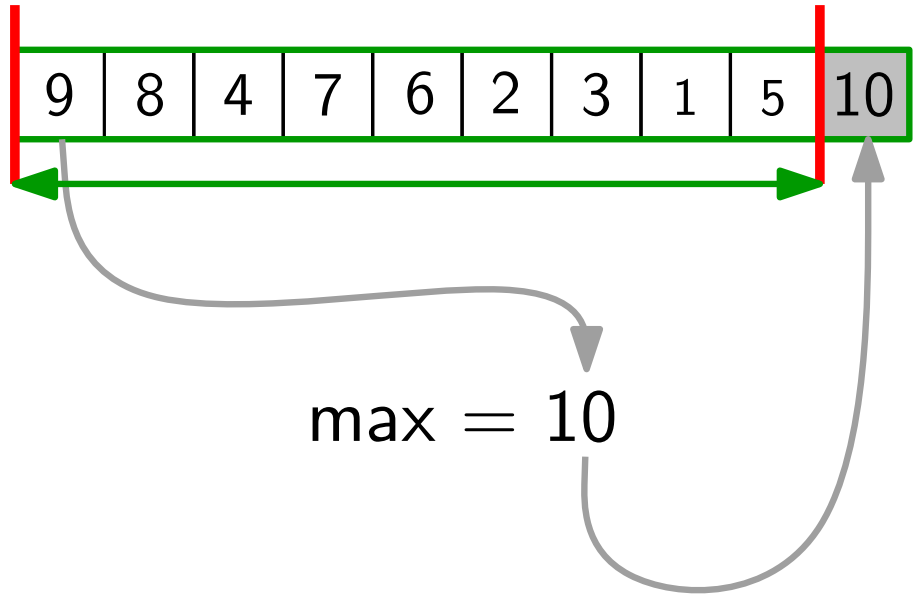
- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



Vom Heap zur Sortierung

Idee:

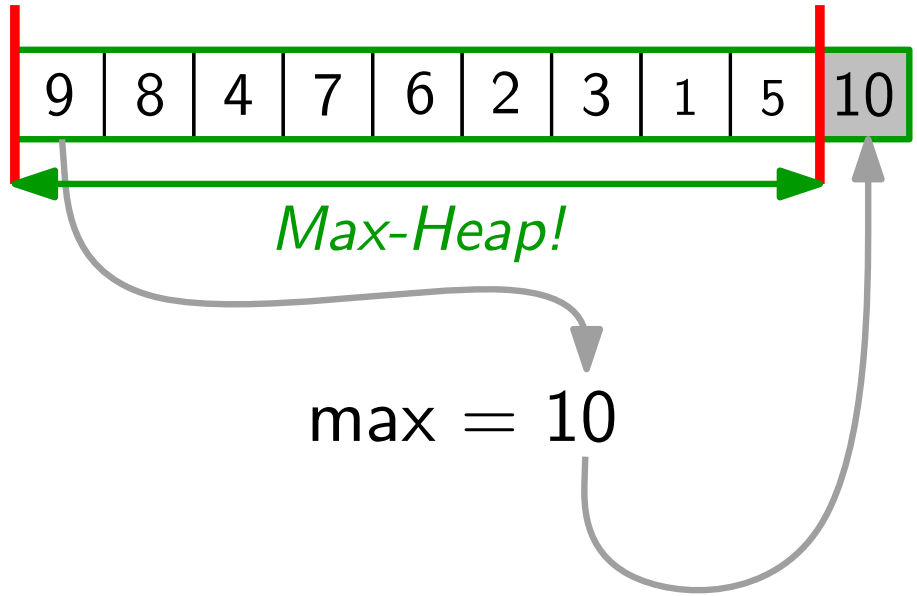
- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



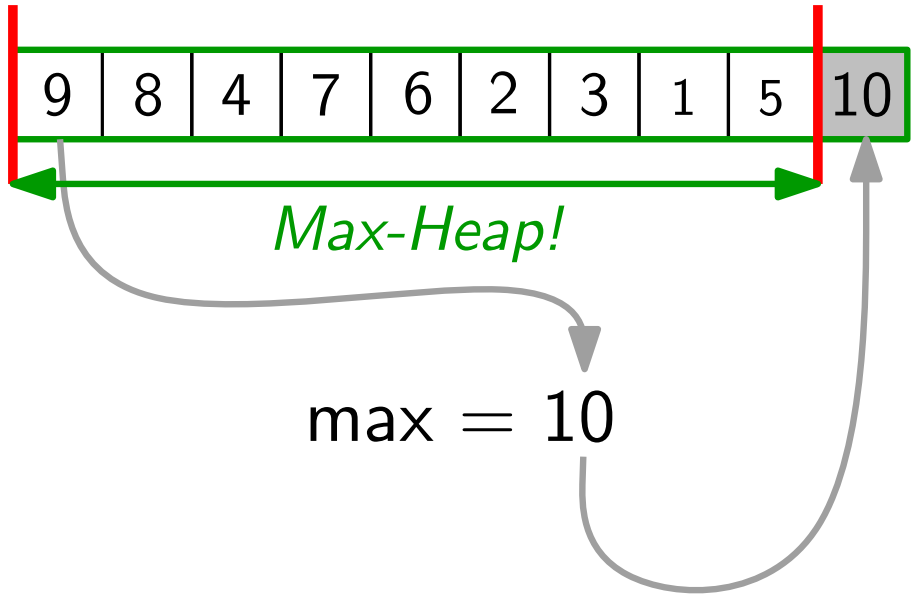
Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

HeapSort(int[] *A*)

Schreiben *Sie* den Pseudocode.
Verwenden Sie BuildMaxHeap
und ExtractMax.

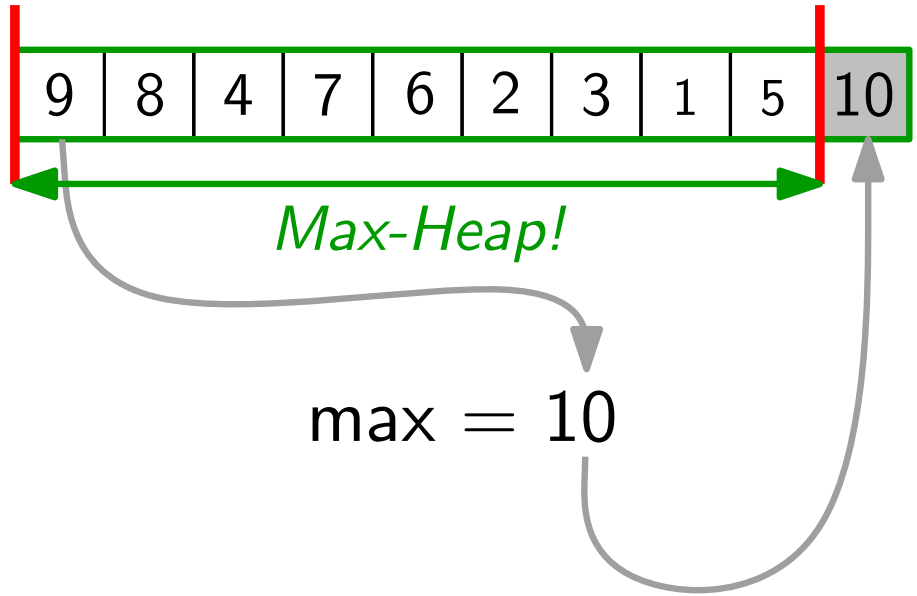


Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

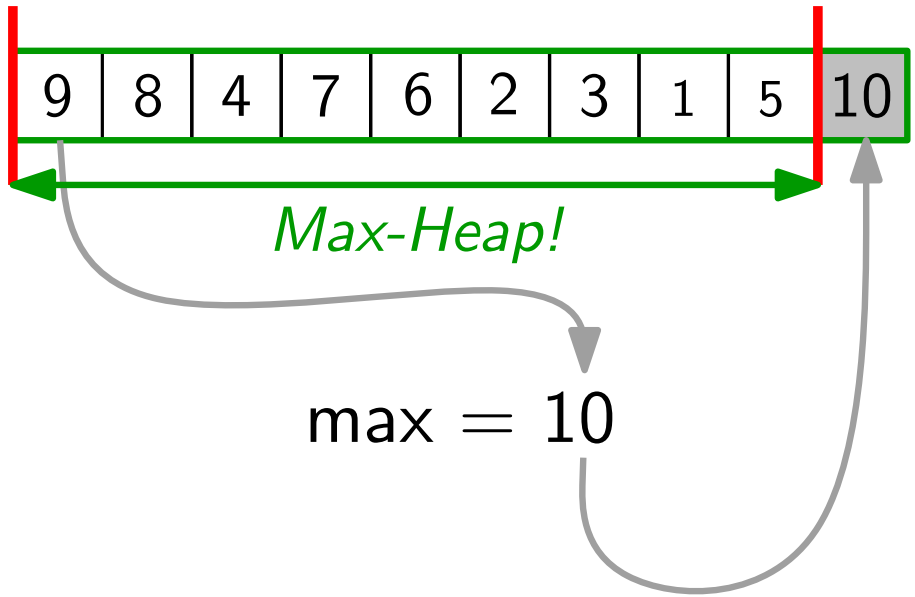


Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```



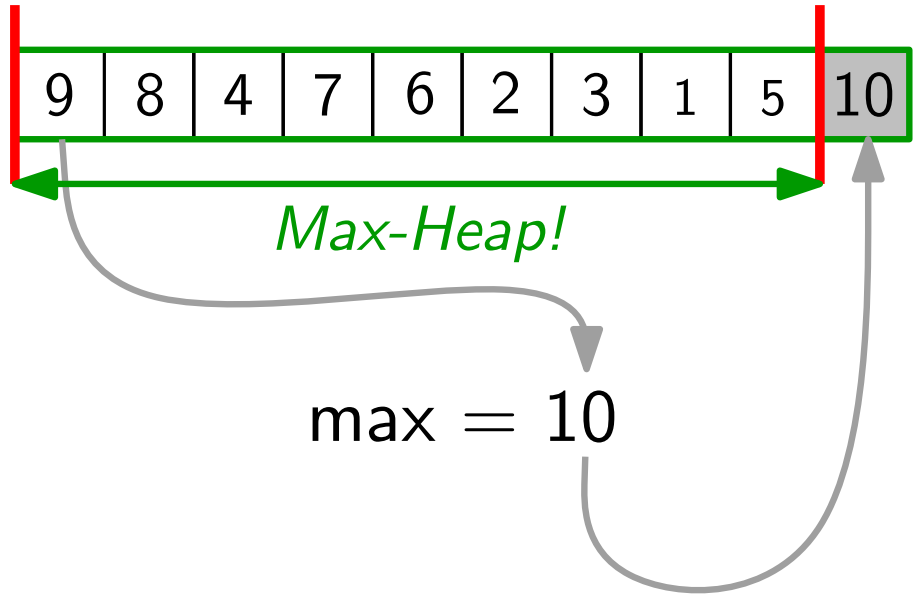
Laufzeit: $T_{\mathsf{HS}}(n)$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```



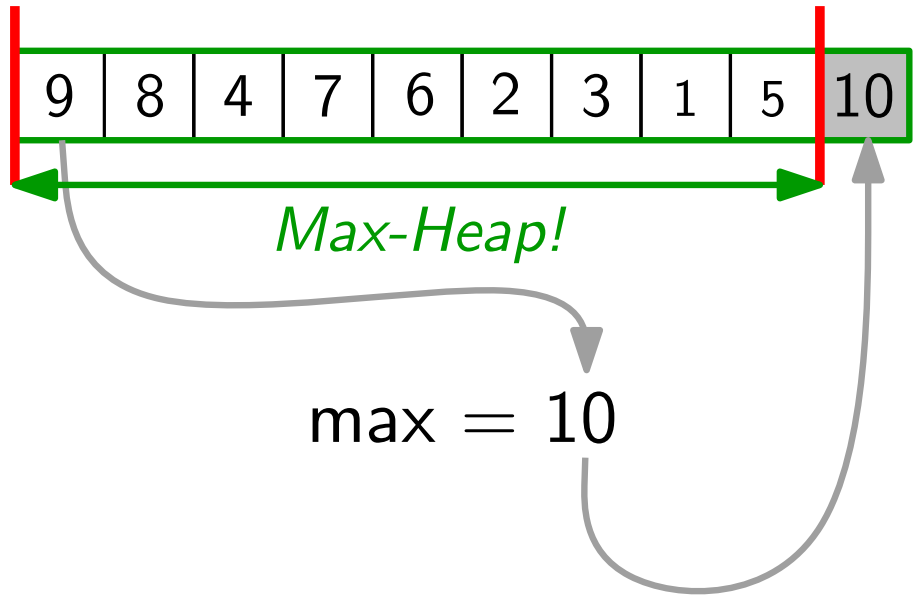
Laufzeit: $T_{\mathsf{HS}}(n) \in$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```



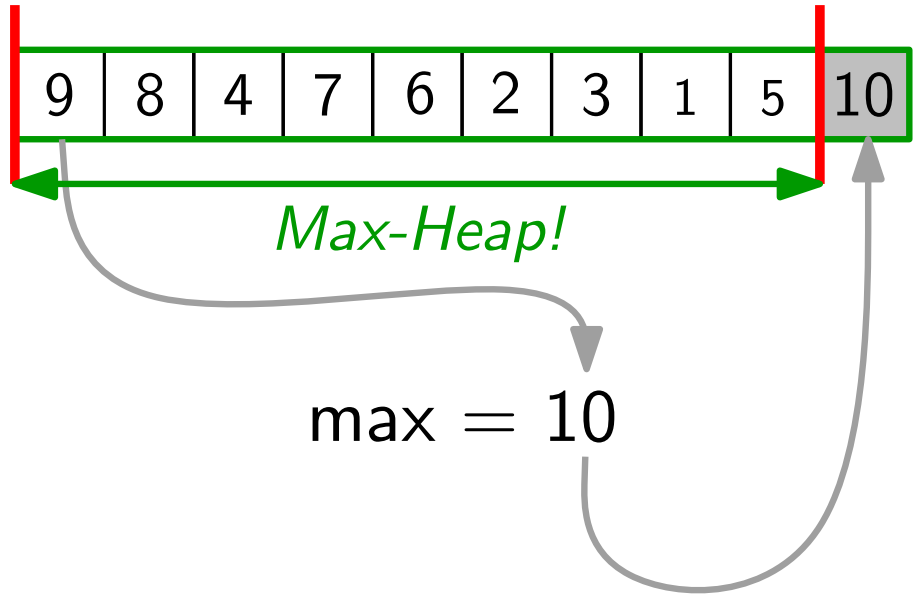
Laufzeit: $T_{\mathsf{HS}}(n) \in$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

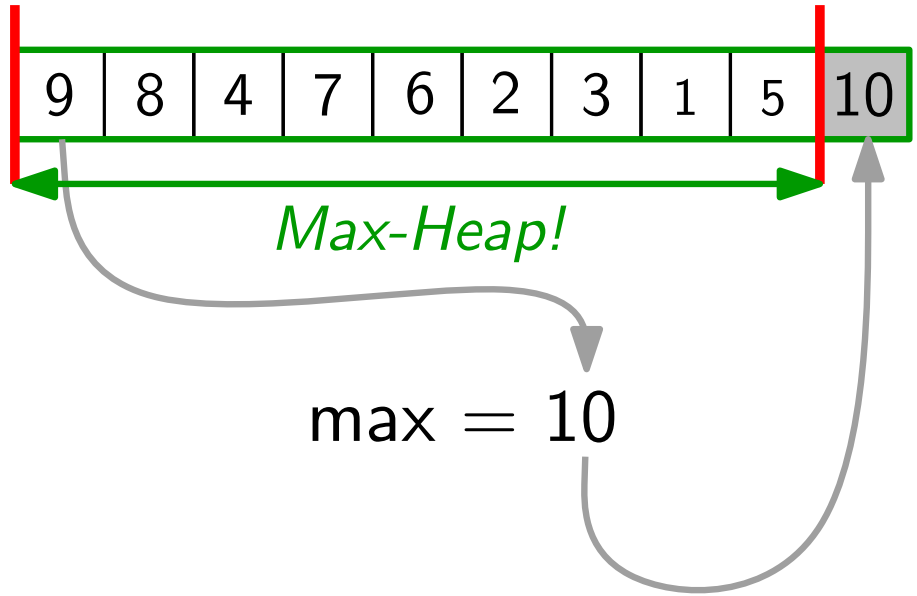


Laufzeit: $T_{\mathsf{HS}}(n) \in O(n)$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



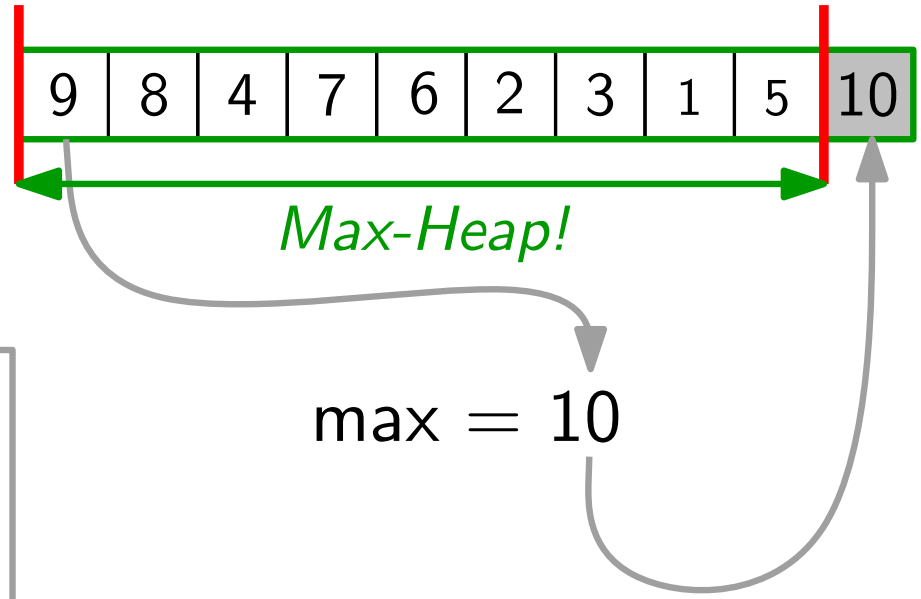

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) +$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



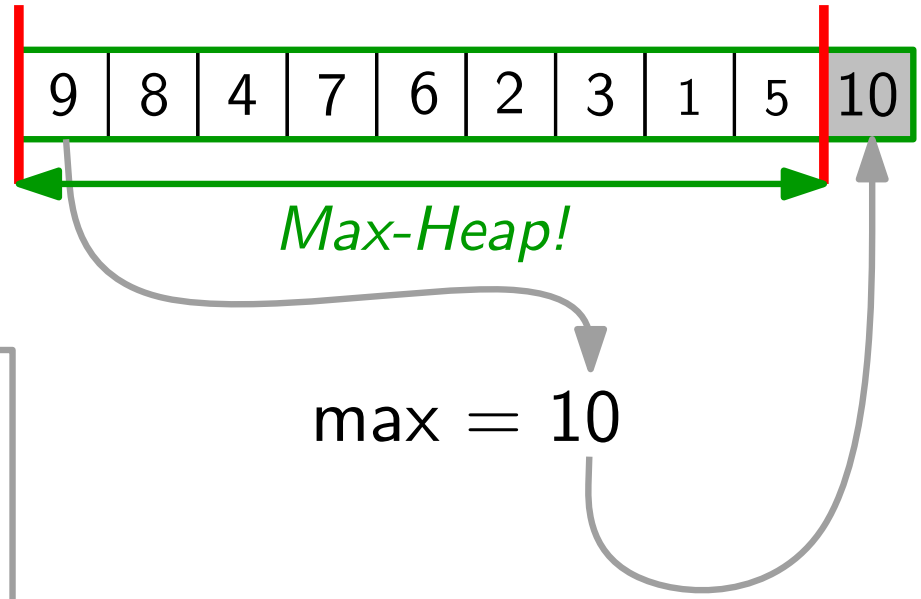

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) +$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



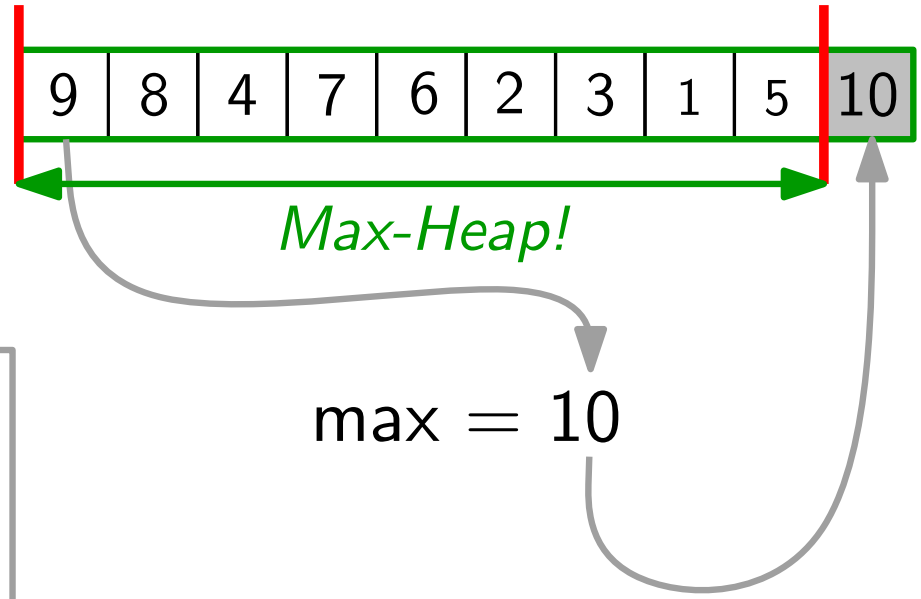

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1)$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



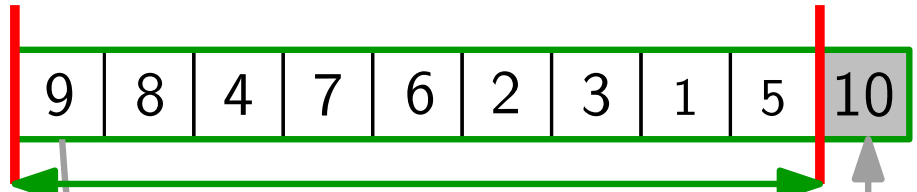

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

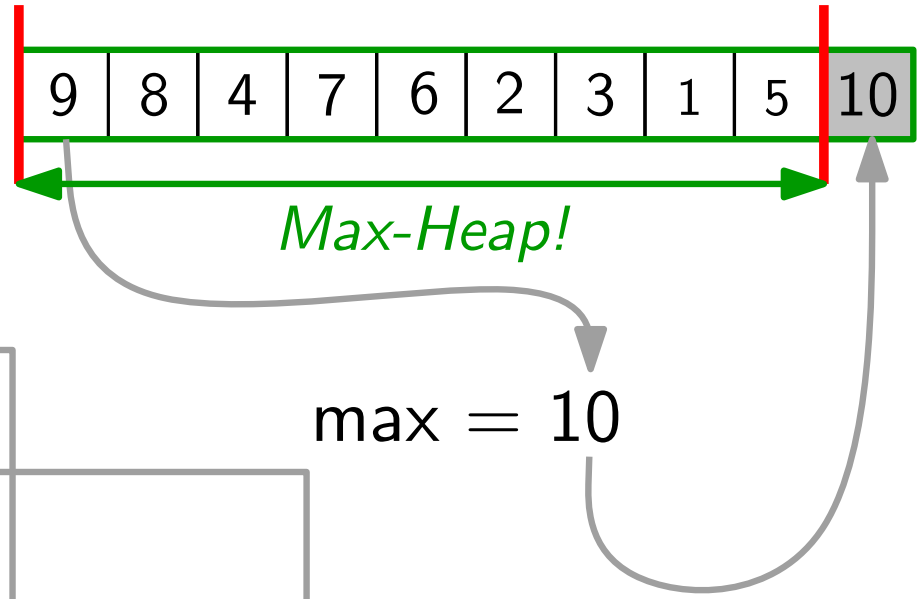
max = 10

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



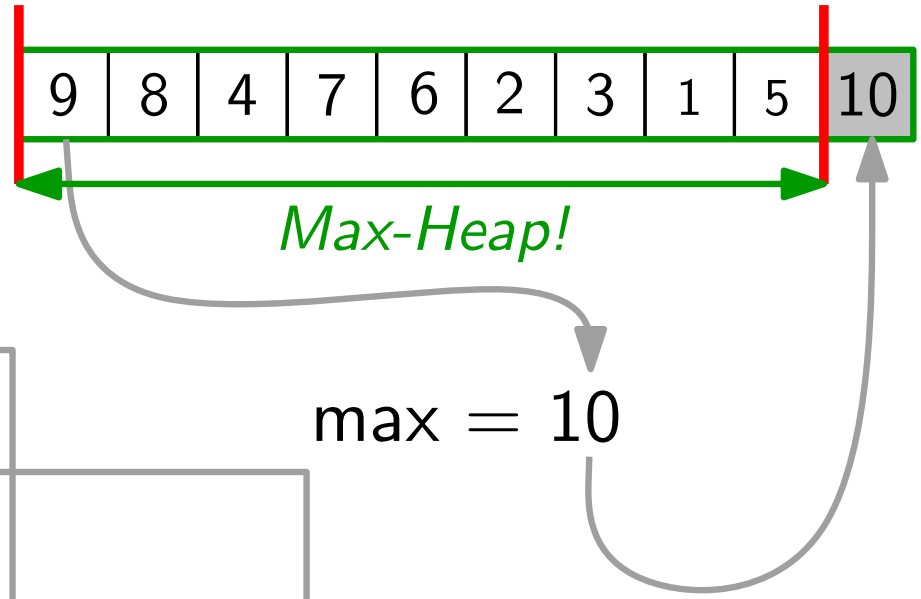

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n)$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



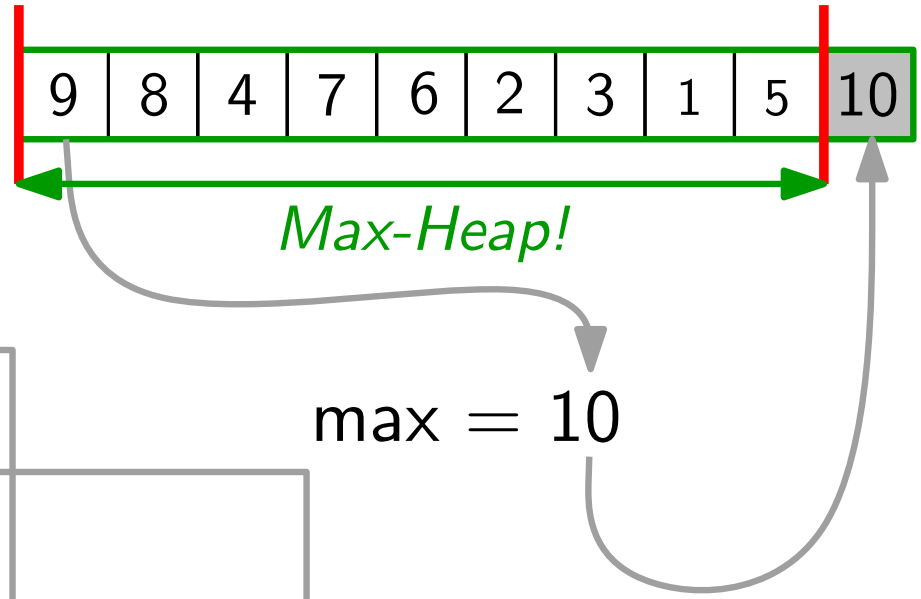

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) =$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.



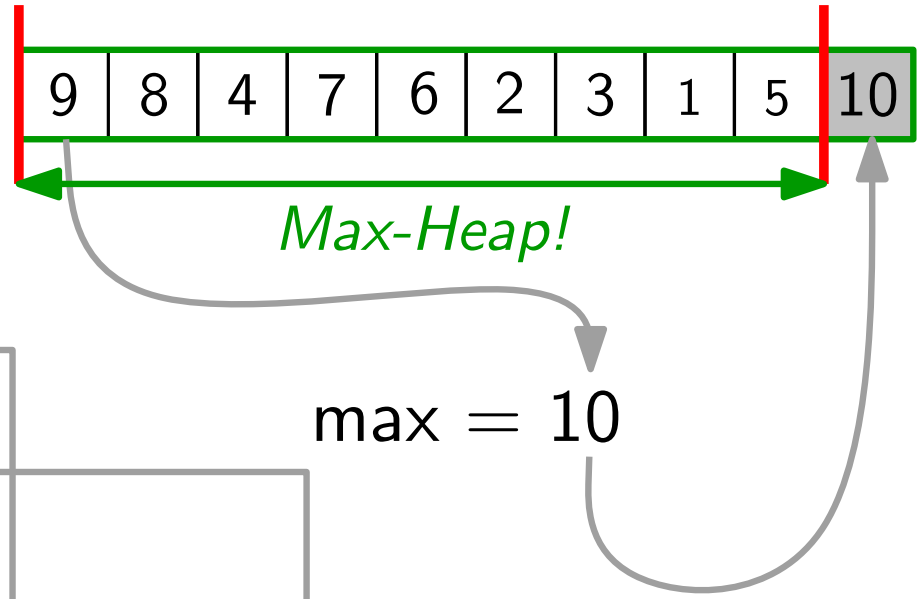

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

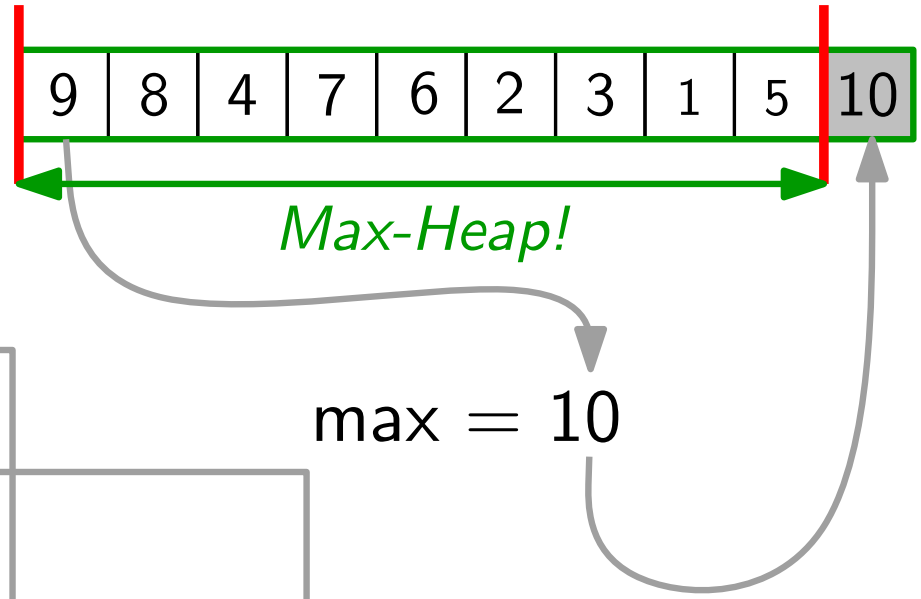
Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

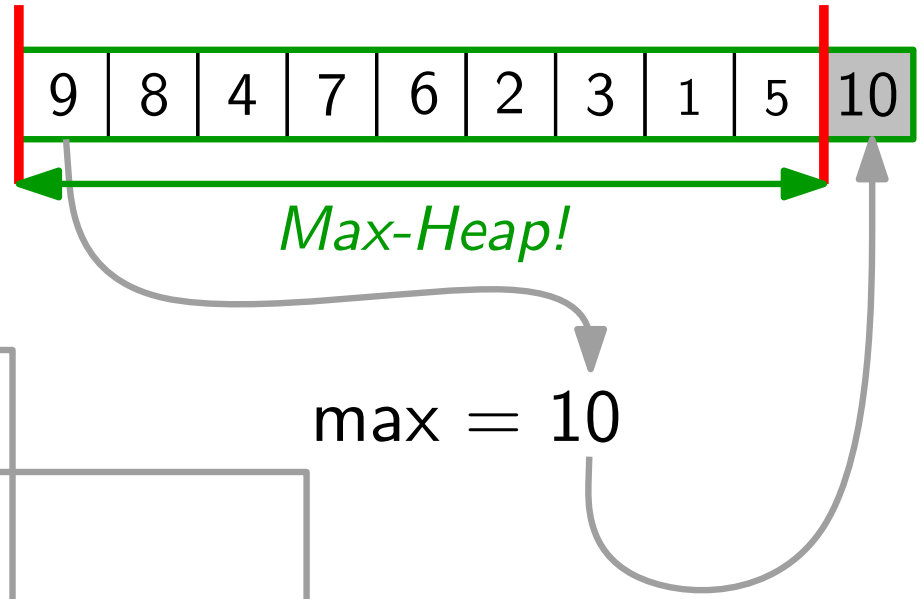
Genauer: $c \cdot n \; +$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

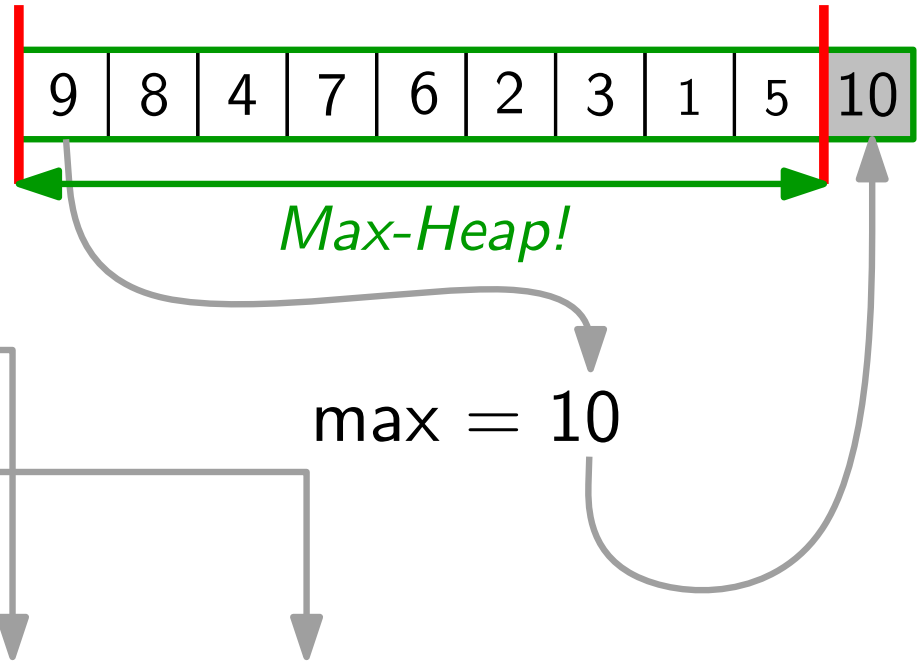
Genauer: $c \cdot n + \sum_{i=2}^{n}$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

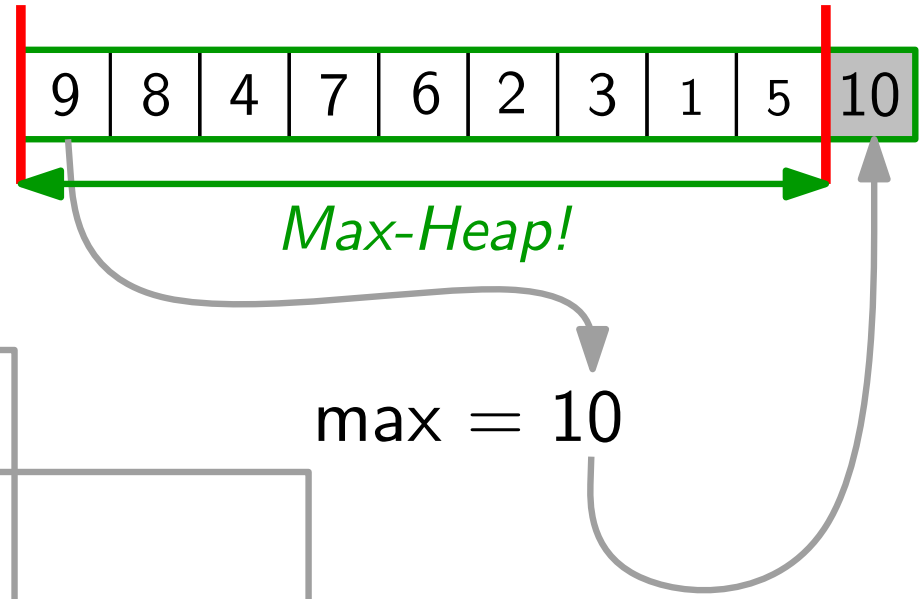
Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

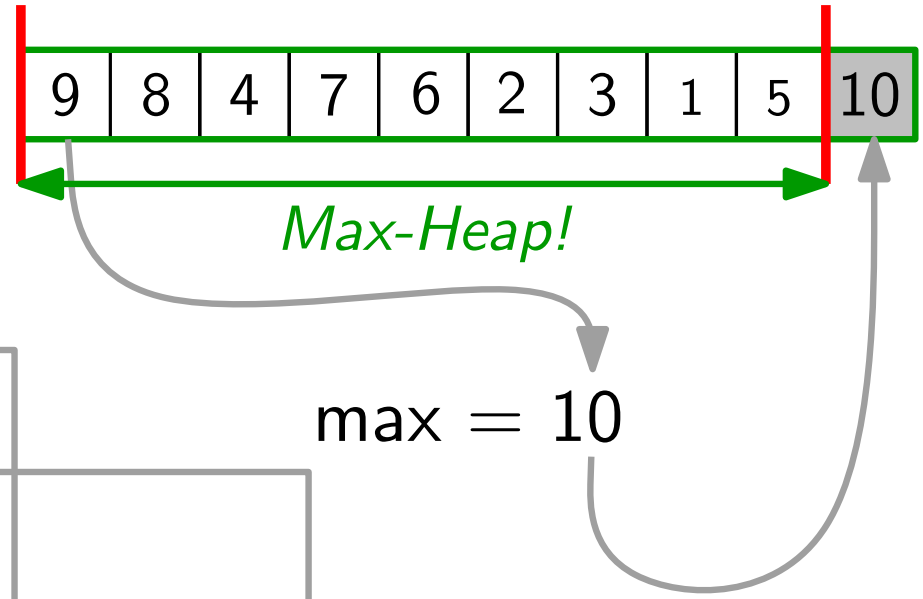
Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n}$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

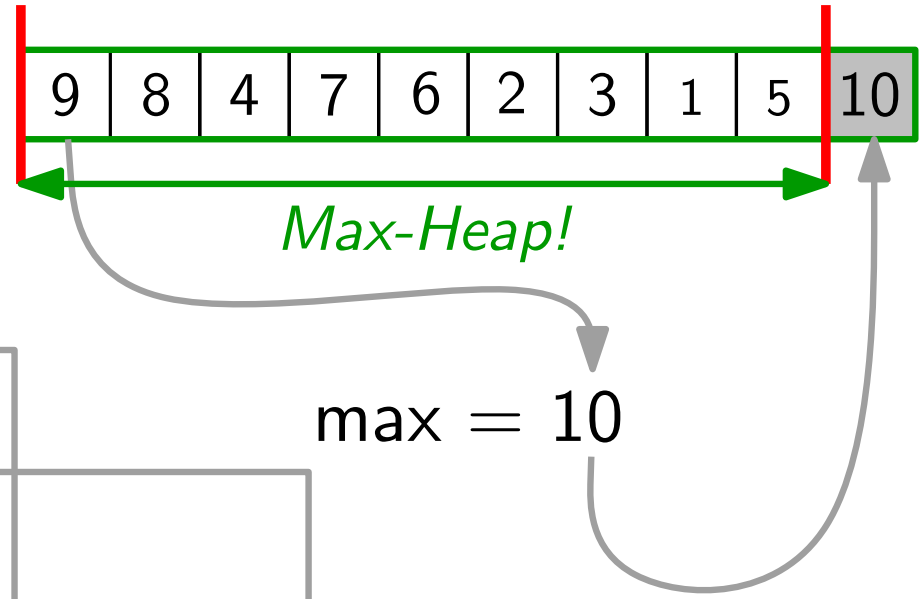
Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2}$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2}$

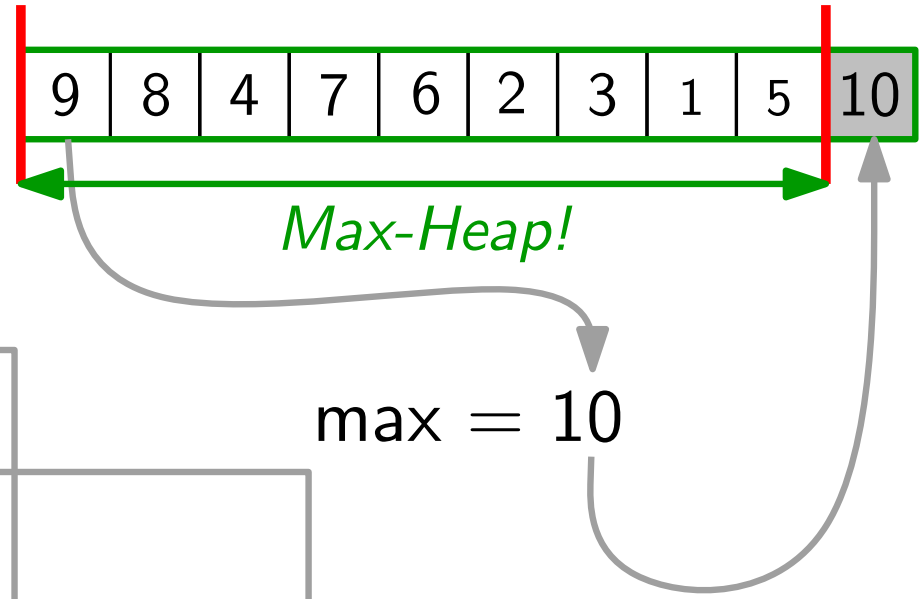
$\in \Omega(\quad)$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2}$

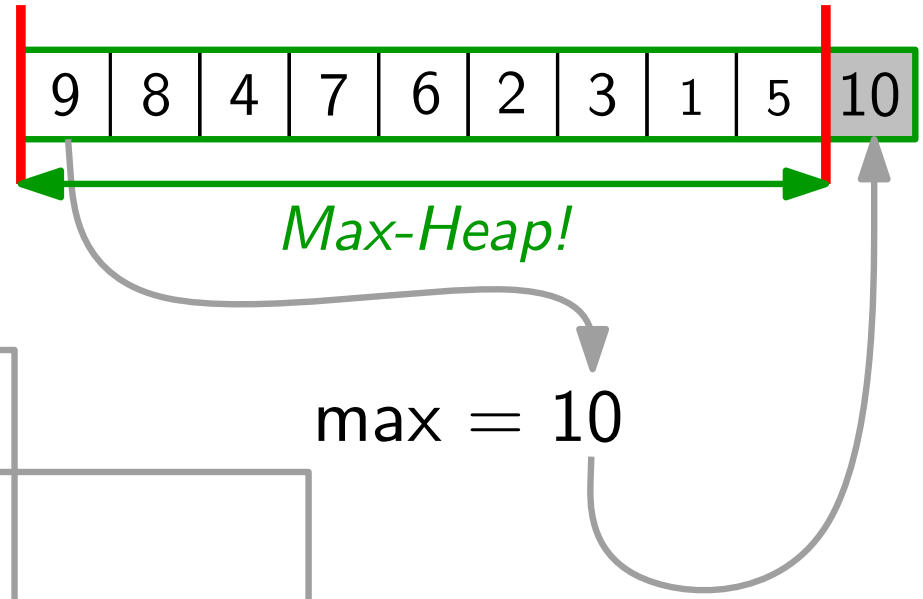
$\in \Omega(n \log n)$

Satz. HeapSort sortiert n Schlüssel in $O(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathsf{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

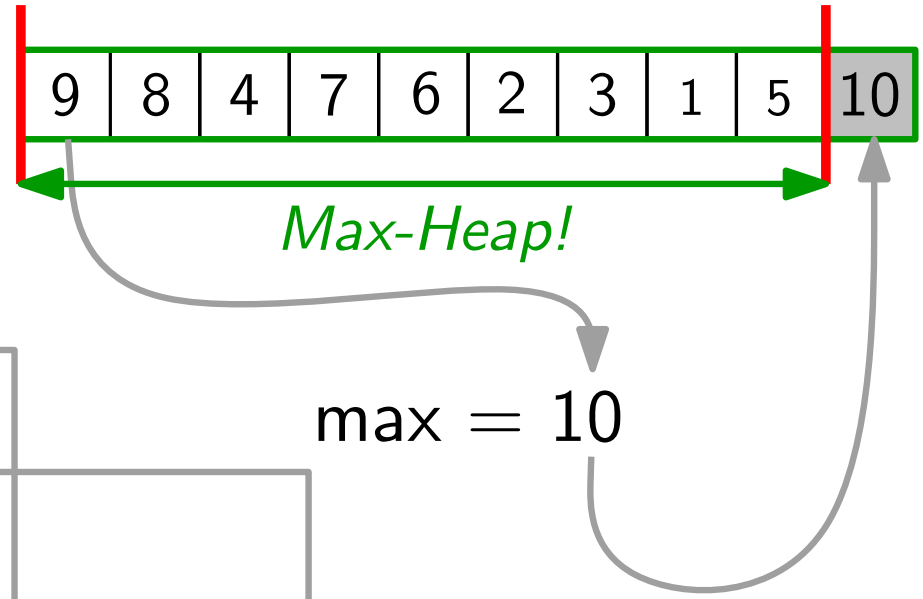
Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2} \in \Omega(n \log n)$

Satz. HeapSort sortiert n Schlüssel in $\Theta(n \log n)$ Zeit.

Vom Heap zur Sortierung

Idee:

- ExtractMax() gibt rechtestes Heap-Element frei.
- Speichere dort das extrahierte Maximum.




```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

Laufzeit: $T_{\mathrm{HS}}(n) \in O(n) + (n-1) \cdot O(\log n) = O(n \log n)$

Genauer: $c \cdot n + \sum_{i=2}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2} \in \Omega(n \log n)$

Satz. HeapSort sortiert n Schlüssel in $\Theta(n \log n)$ Zeit.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit				
Avg.-Case-Laufzeit				
Best-Case-Laufzeit				
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit				
Best-Case-Laufzeit				
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit				
Best-Case-Laufzeit				
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit				
Best-Case-Laufzeit	$\Theta(n)$			
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	**?**			
Best-Case-Laufzeit	$\Theta(n)$			
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	$\Theta(n^2)$			
Best-Case-Laufzeit	$\Theta(n)$			
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*			
Best-Case-Laufzeit	$\Theta(n)$			
in situ (*in place*)				
stabil				

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*			
Best-Case-Laufzeit	$\Theta(n)$			
in situ[1] (*in place*)				
stabil				

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*			
Best-Case-Laufzeit	$\Theta(n)$			
in situ[1] (*in place*)	✓			
stabil				

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*			
Best-Case-Laufzeit	$\Theta(n)$			
in situ[1] (*in place*)	✓			
stabil[2]				

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$			
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*			
Best-Case-Laufzeit	$\Theta(n)$			
in situ[1] (*in place*)	✓			
stabil[2]	✓			

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$		
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*			
Best-Case-Laufzeit	$\Theta(n)$			
in situ[1] (*in place*)	✓			
stabil[2]	✓			

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$		
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$		
Best-Case-Laufzeit	$\Theta(n)$			
in situ[1] (*in place*)	✓			
stabil[2]	✓			

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$		
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$		
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$		
in situ[1] (*in place*)	✓			
stabil[2]	✓			

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$		
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$		
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$		
in situ[1] (*in place*)	✓	✗		
stabil[2]	✓			

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$		
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$		
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$		
in situ[1] (*in place*)	✓	✗		
stabil[2]	✓	✓		

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$		
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$		
in situ[1] (*in place*)	✓	✗		
stabil[2]	✓	✓		

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$	$\Theta(n \log n)$	
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$		
in situ[1] (*in place*)	✓	✗		
stabil[2]	✓	✓		

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$	$\Theta(n \log n)$	
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
in situ[1] (*in place*)	✓	✗		
stabil[2]	✓	✓		

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$	$\Theta(n \log n)$	
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
in situ[1] (*in place*)	✓	✗	✓	
stabil[2]	✓	✓		

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$	$\Theta(n \log n)$	
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
in situ[1] (*in place*)	✓	✗	✓	
stabil[2]	✓	✓	✗	

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort	
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Avg.-Case-Laufzeit	$\Theta(n^2)$ *Warum?*	$\Theta(n \log n)$	$\Theta(n \log n)$	
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
in situ[1] (*in place*)	✓	✗	✓	
stabil[2]	✓	✓	✗	

[1]) Ein *in-situ*-Algorithmus benötigt nur $O(1)$ extra Speicher.

[2]) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenf. belässt.