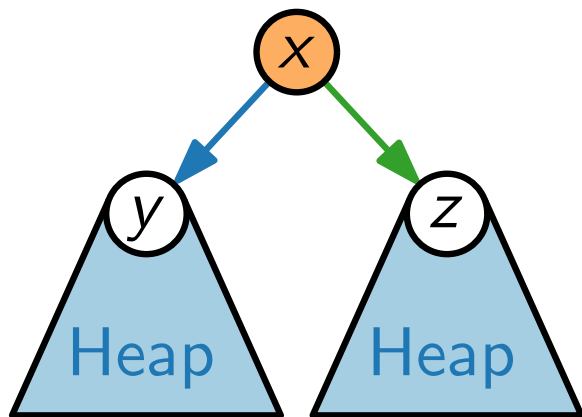
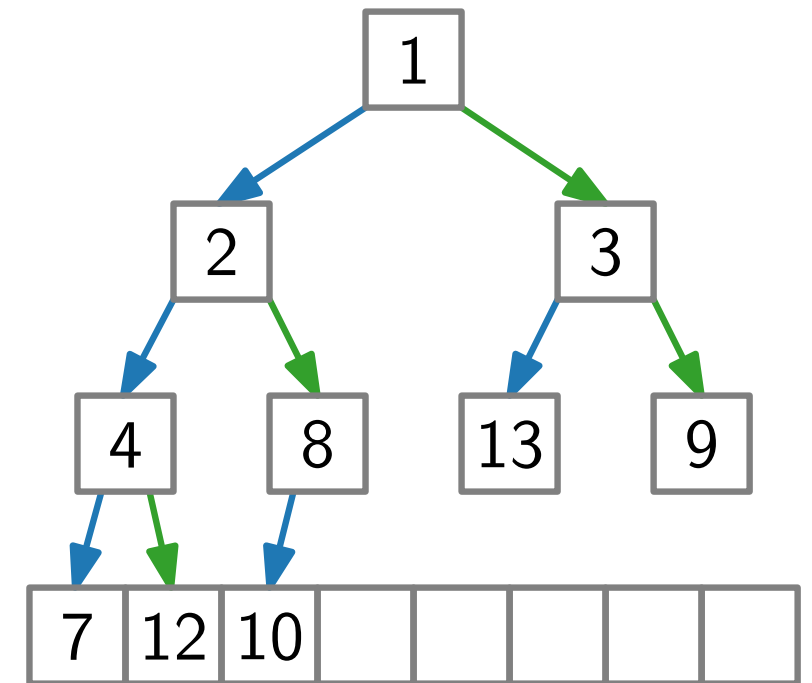


Algorithmen und Datenstrukturen

Vorlesung 6: Heaps und HeapSort



Guten Morgen!

Tipps für unseren ersten Zwischentest am Do, 12. November:

- Lesen Sie die Definitionen der Klassen O , Ω und Θ gaaaaanz 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, mir!
- Haben Sie schon ein Buch??

Guten Morgen!

Tipps für unseren ersten Zwischentest am Do, 12. November:

- Lesen Sie die Definitionen der Klassen O , Ω und Θ gaaaaanz 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, 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, 12. November:

- Lesen Sie die Definitionen der Klassen O , Ω und Θ gaaaaanz 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, 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, 12. November:

- Lesen Sie die Definitionen der Klassen O , Ω und Θ gaaaaanz 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, 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!

Wir bauen eine Datenstruktur

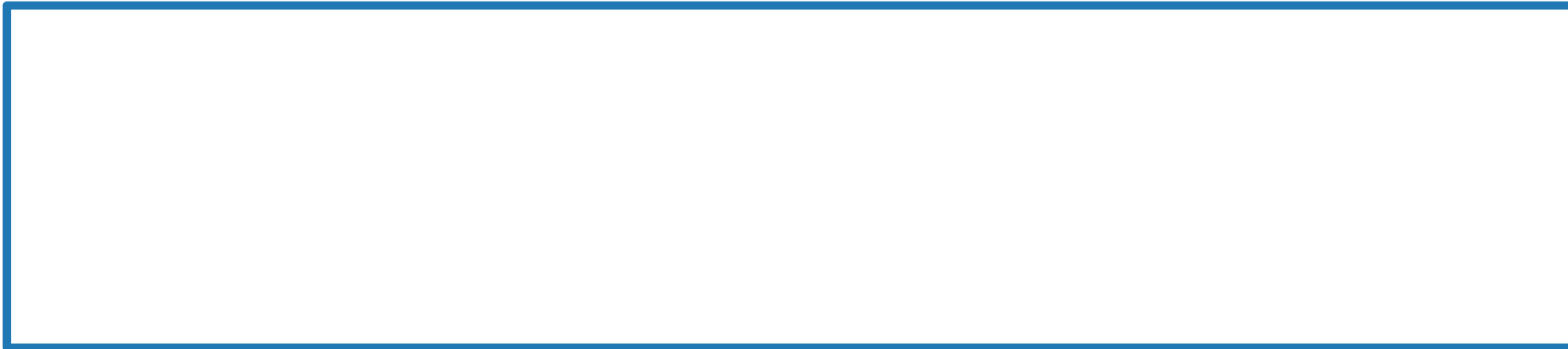
Datenstruktur.

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

Wir bauen eine Datenstruktur

Datenstruktur.

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



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.

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.

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?

Ein simples Kartenspiel

2 mögliche Züge:



Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



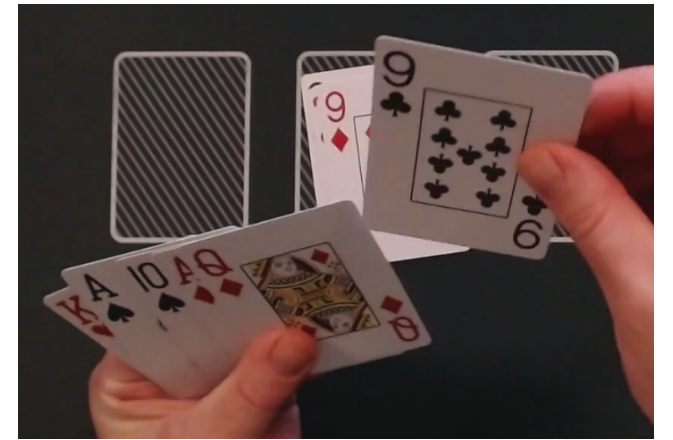
Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



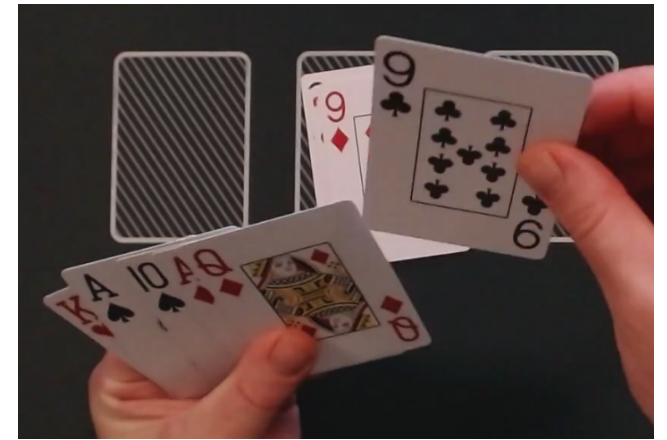
Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

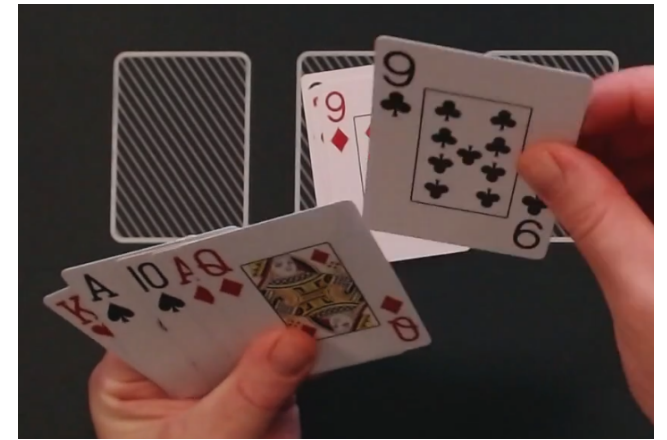
Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Datenstruktur

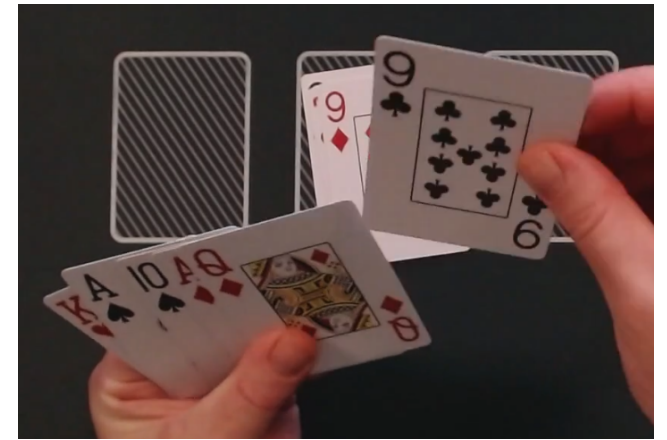
Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Abstrakter Datentyp:

Datenstruktur

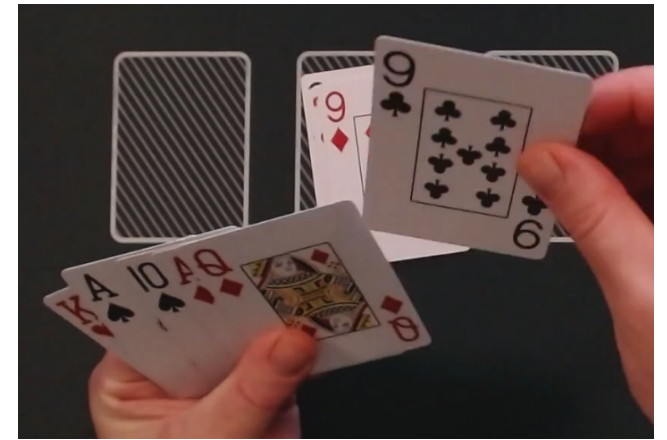
Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Abstrakter Datentyp:

Prioritätsschlange

Datenstruktur

Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Abstrakter Datentyp: **Prioritätsschlange**

verwaltet Elemente einer Menge M ,

Datenstruktur

Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Abstrakter Datentyp: **Prioritätsschlange**

verwaltet Elemente einer Menge M ,

Karten

Datenstruktur

Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Abstrakter Datentyp:

Prioritätsschlange

Karten

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

Datenstruktur

Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen



(2) **kleinste** Karte spielen



Frage:

Welche Laufzeit haben Schritt (1) und (2)?

Antwort:

Kommt drauf an! Wie halten wir die Karten in der Hand?

Abstrakter Datentyp:

Prioritätsschlange

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

Karten

Kartenwert

Datenstruktur

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
<code>INSERT(element x)</code>	
element <code>FINDMIN()</code>	
element <code>EXTRACTMIN()</code>	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	Funktionalität
<code>INSERT(element x)</code>	$M = M \cup \{x\}$
element <code>FINDMIN()</code>	
element <code>EXTRACTMIN()</code>	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	Funktionalität
<code>INSERT(element x)</code>	$M = M \cup \{x\}$
element <code>FINDMIN()</code>	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element <code>EXTRACTMIN()</code>	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	Funktionalität
<code>INSERT(element x)</code>	$M = M \cup \{x\}$
element <code>FINDMIN()</code>	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element <code>EXTRACTMIN()</code>	$x = \text{FINDMIN}(); M = M \setminus \{x\};$ liefere x

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge			

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$		

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert			

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$		

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

müssen wieder kleinste Karte finden

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten in sortierter Reihenfolge			

müssen wieder kleinste Karte finden

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten in sortierter Reihenfolge	$\Theta(n)$		

müssen wieder kleinste Karte finden

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	

müssen wieder kleinste Karte finden

Implementierung

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

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

müssen wieder kleinste Karte finden

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	Funktionalität
<code>INSERT(element x)</code>	$M = M \cup \{x\}$
element <code>FINDMIN()</code>	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element <code>EXTRACTMIN()</code>	$x = \text{FindMin}(); M = M \setminus \{x\};$ liefere x

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 FINDMIN ()	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element EXTRACTMIN ()	$x = \text{FindMin}(); M = M \setminus \{x\};$ liefere x
DECREASEKEY (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 FINDMIN ()	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element EXTRACTMIN ()	$x = \text{FindMin}(); M = M \setminus \{x\};$ liefere x
DECREASEKEY (element x , priorität p)	$x.key = p$

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten als MINHEAP				

Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten als MINHEAP	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

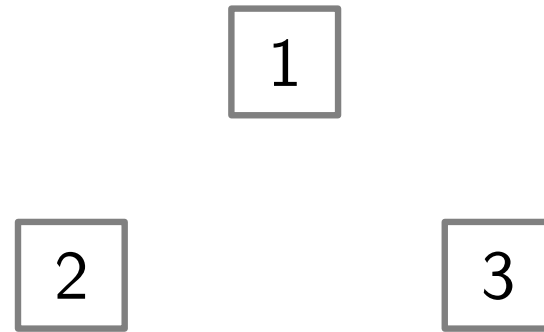
Implementierung

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in sortierter Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten als MINHEAP	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

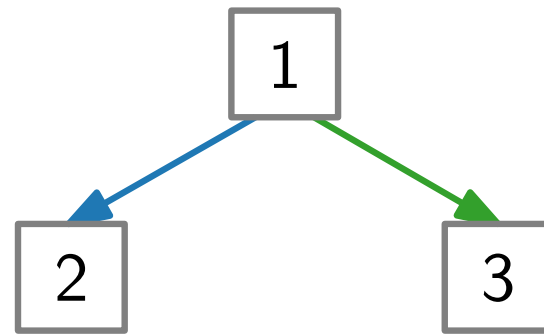
Min-Heaps

1

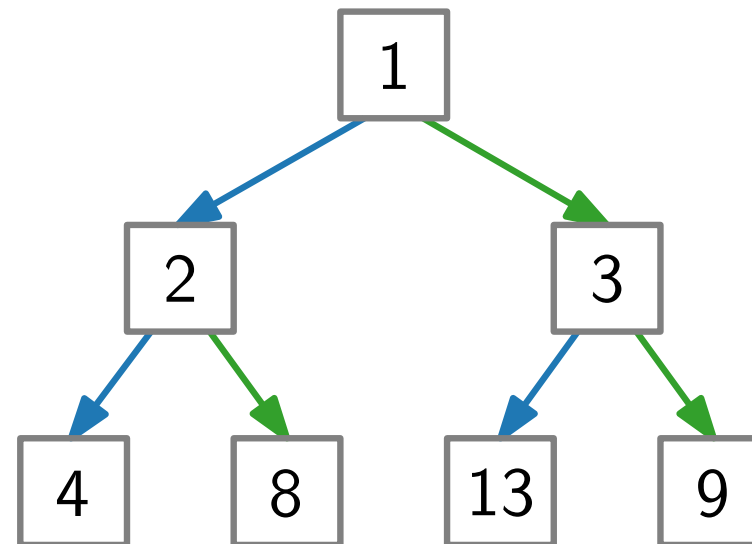
Min-Heaps



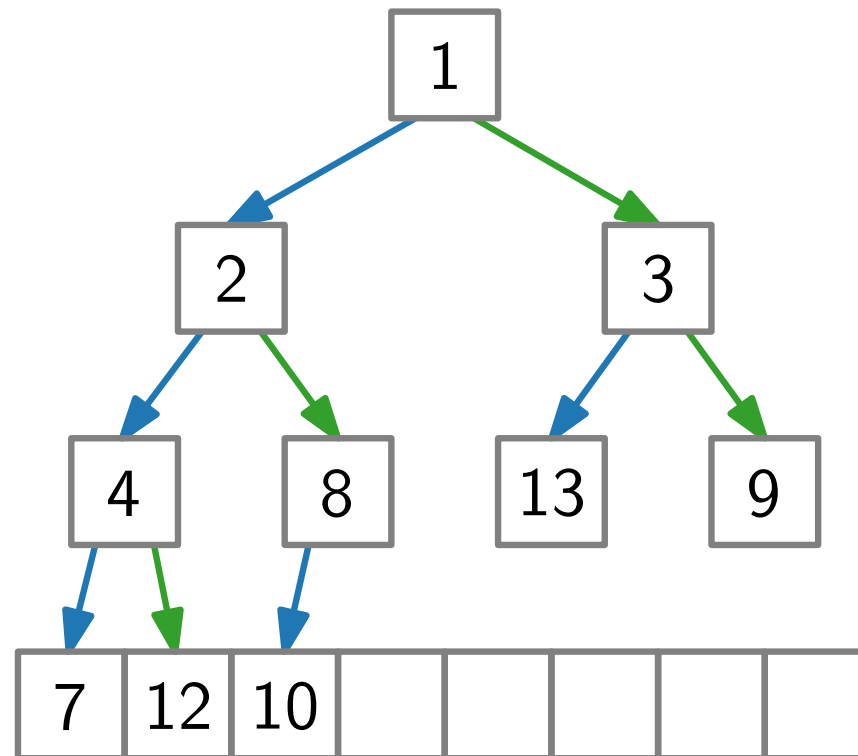
Min-Heaps



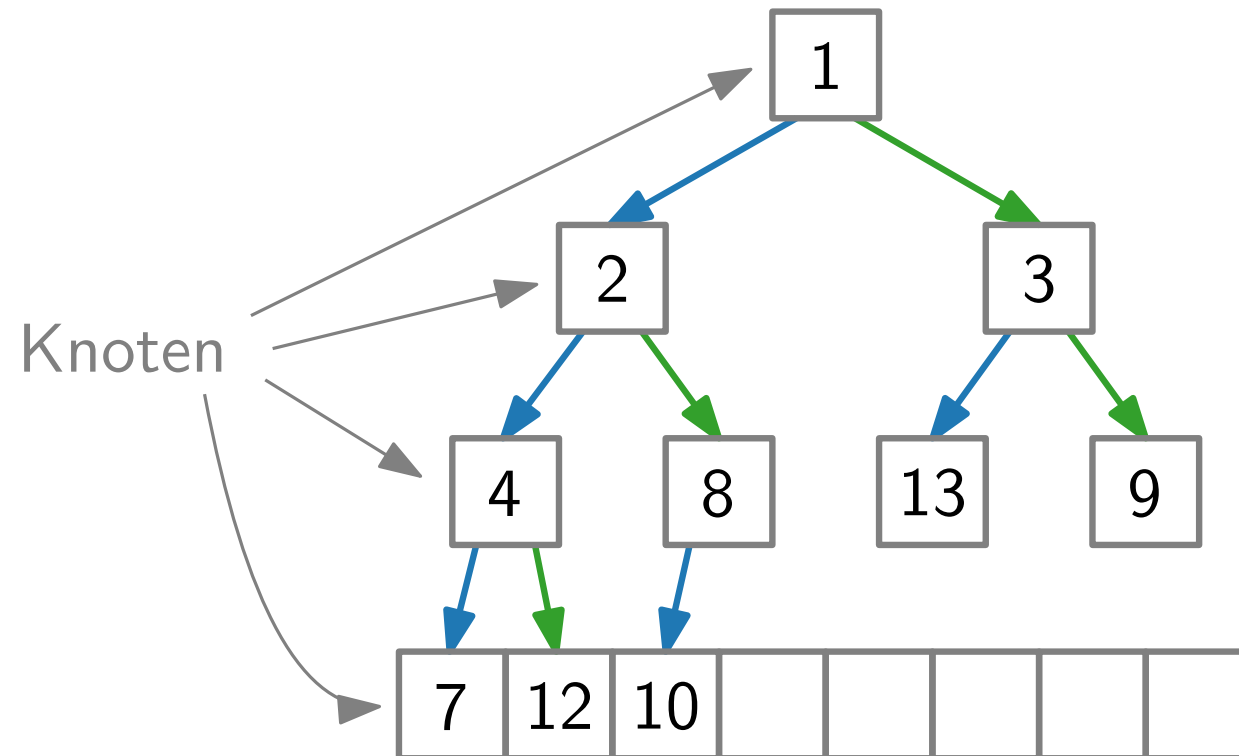
Min-Heaps



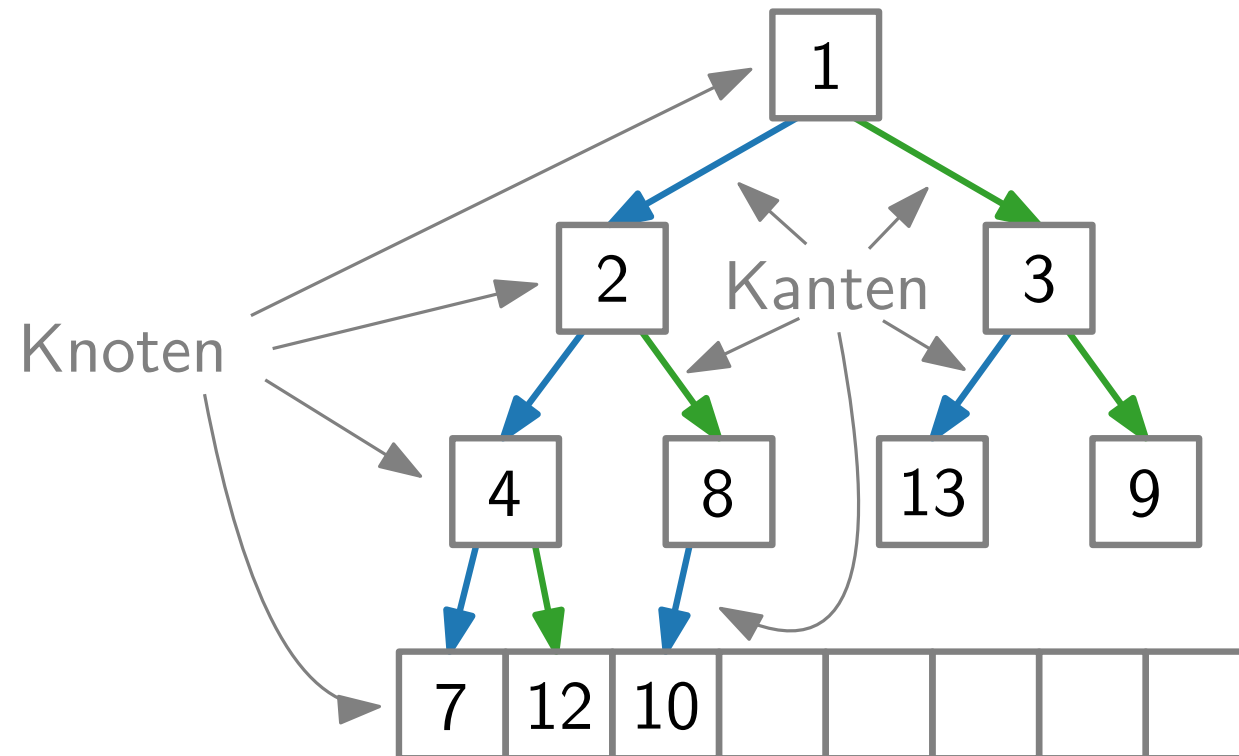
Min-Heaps



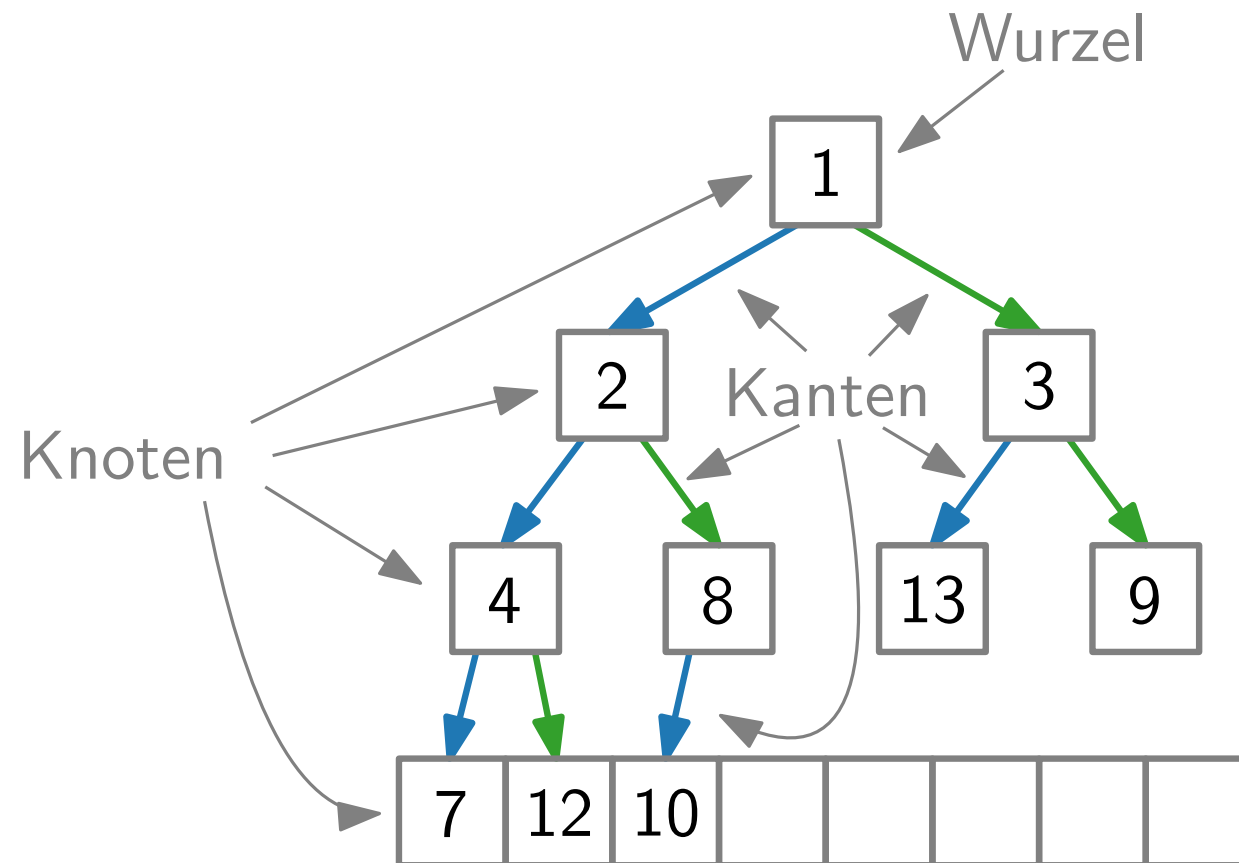
Min-Heaps

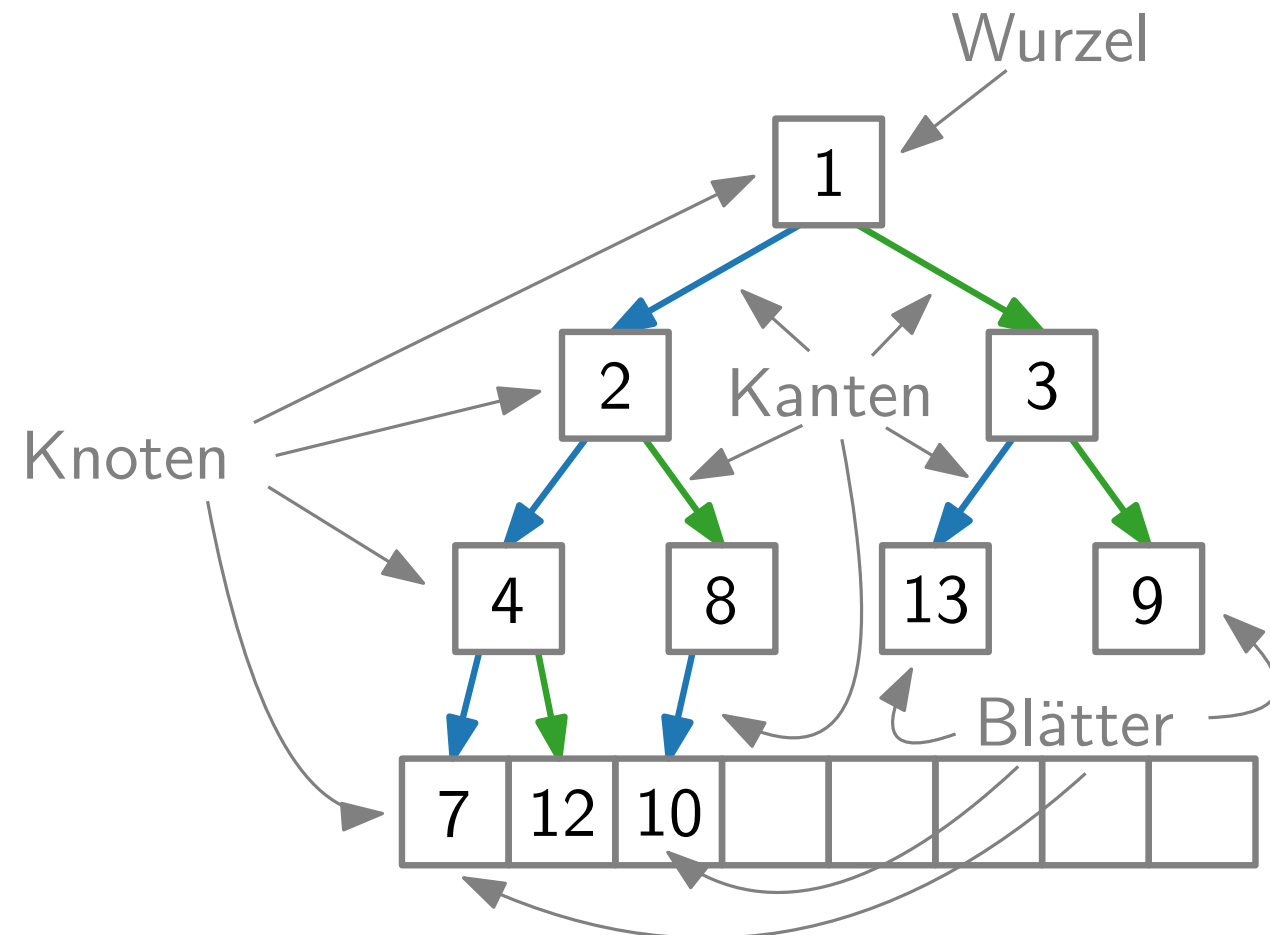


Min-Heaps

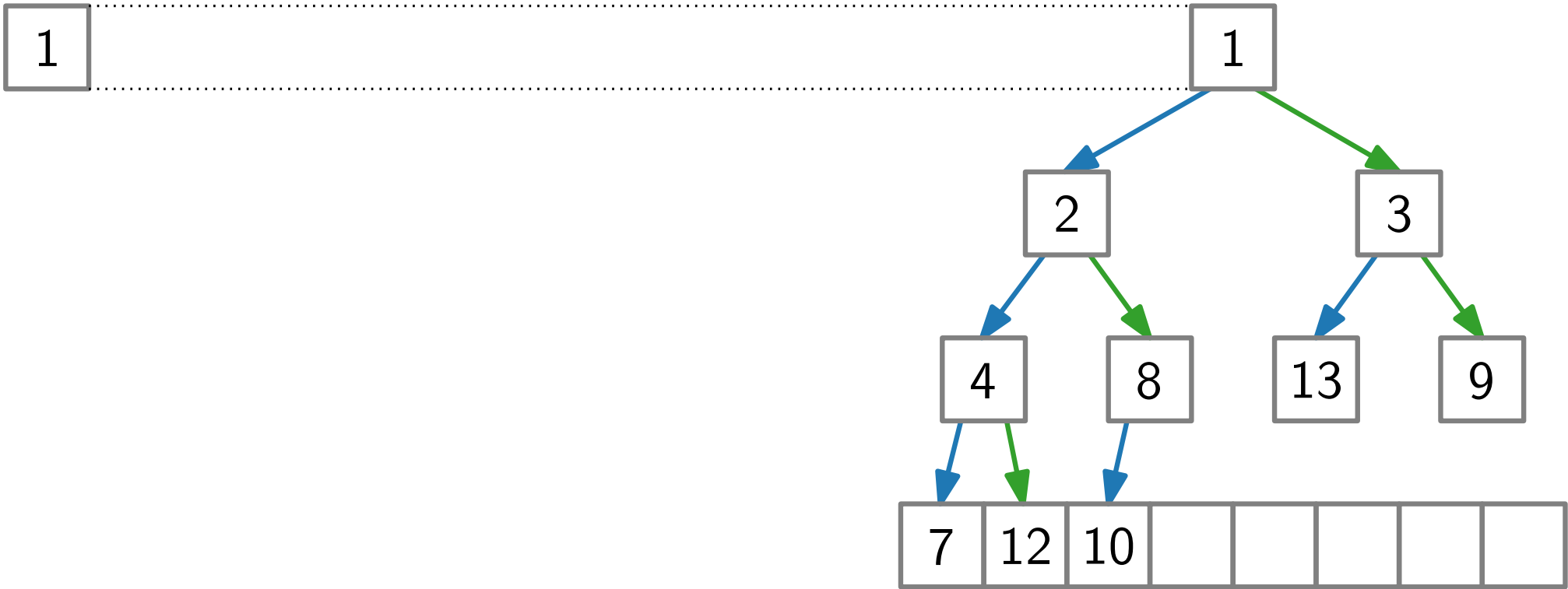


Min-Heaps

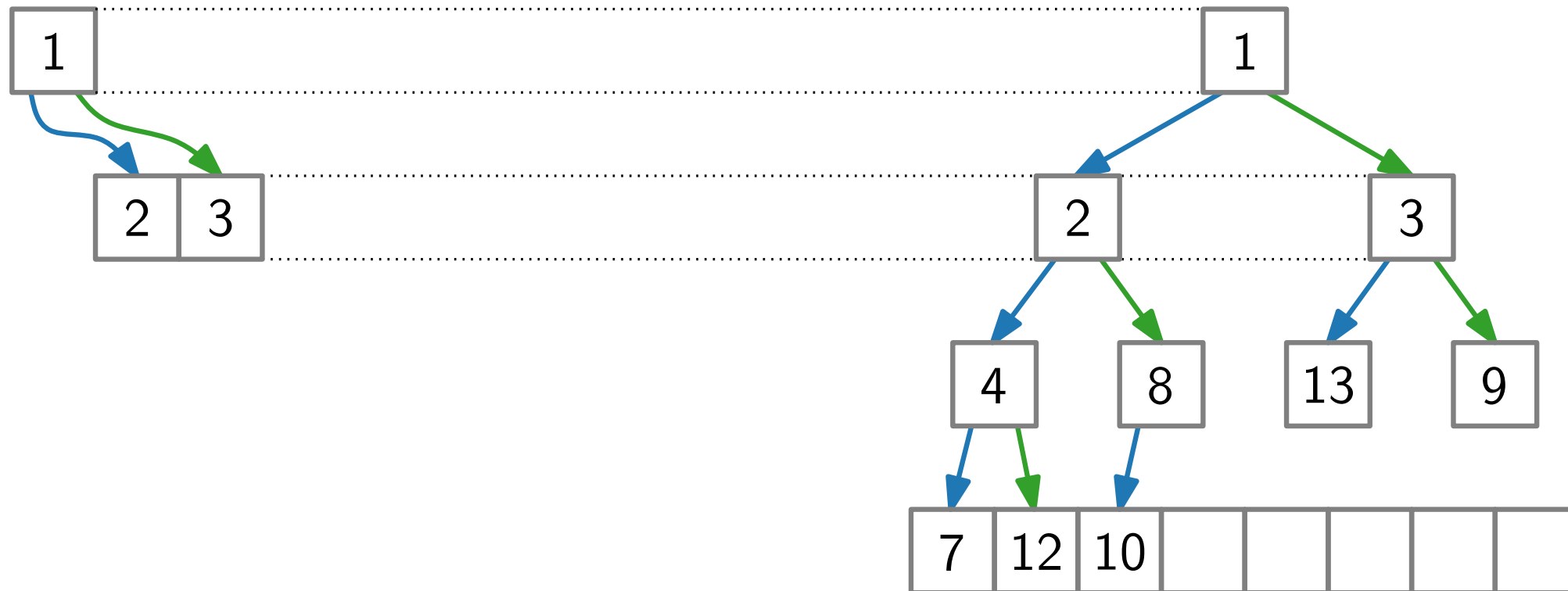




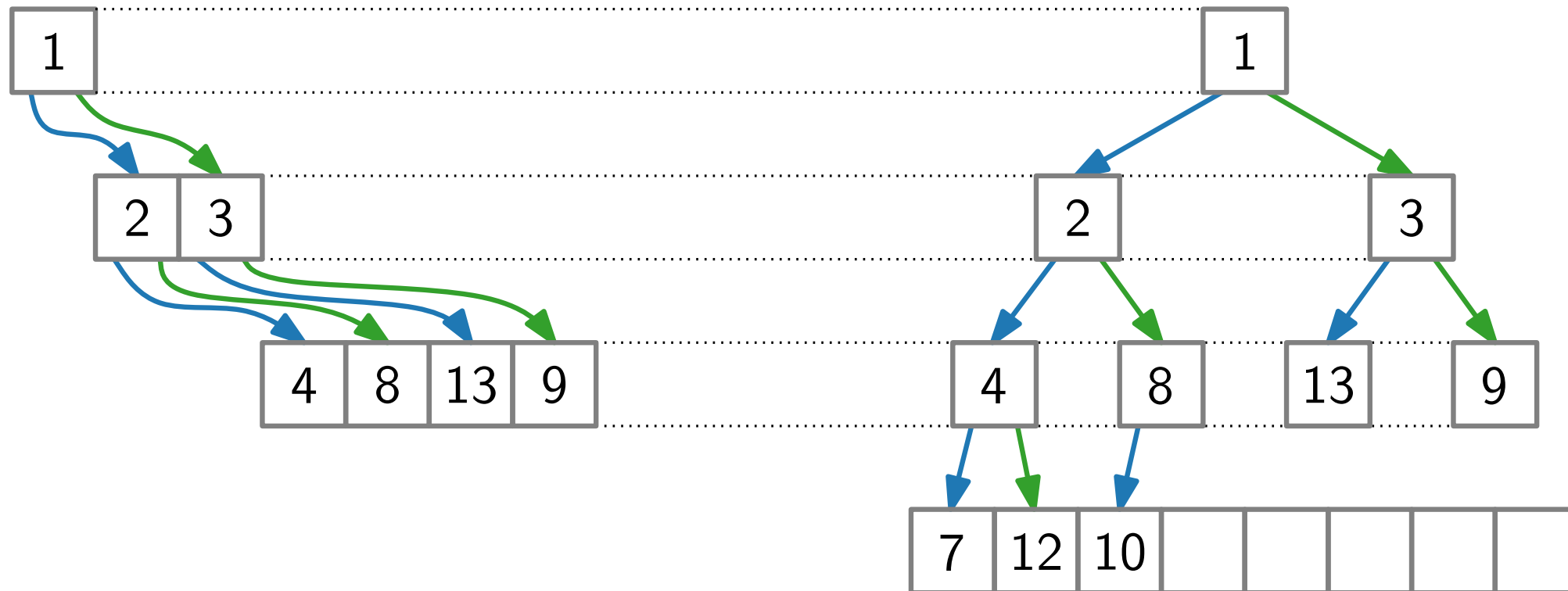
Min-Heaps



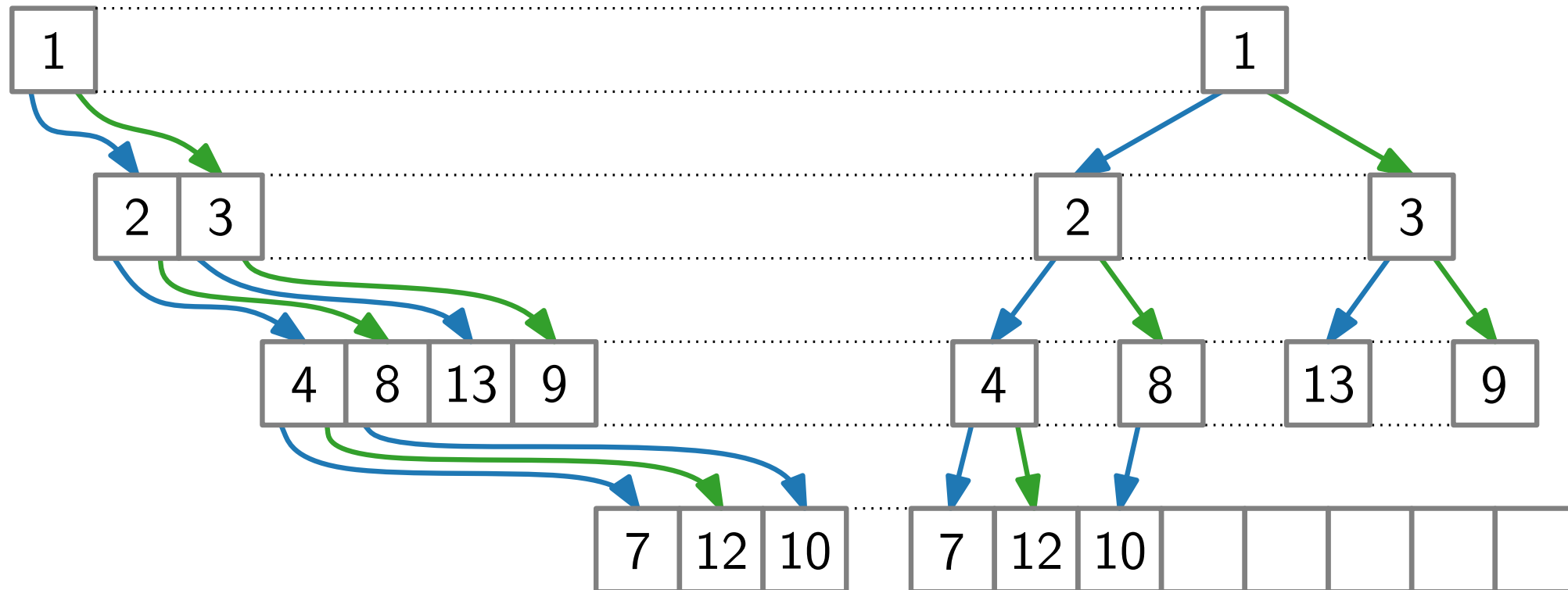
Min-Heaps



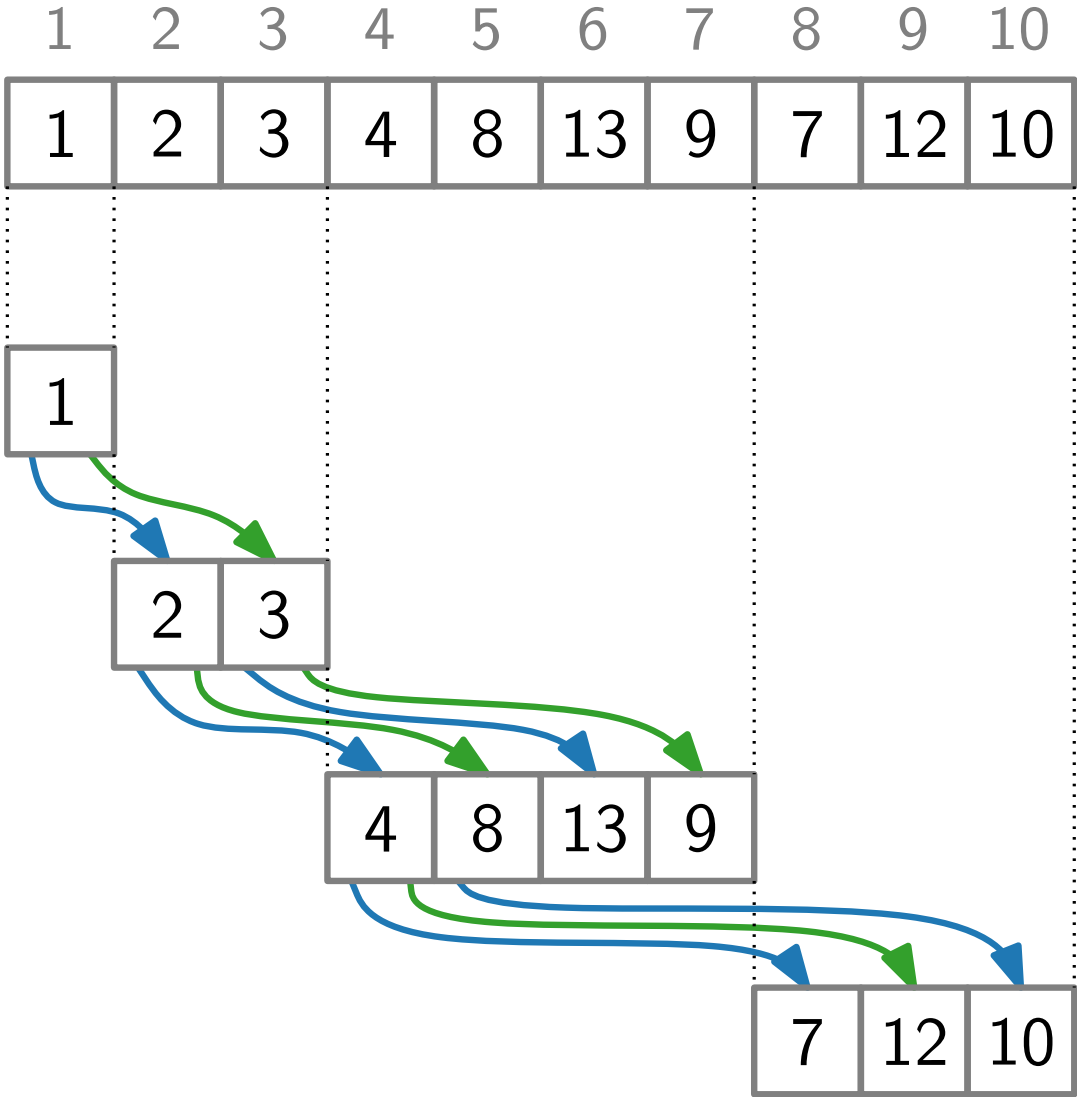
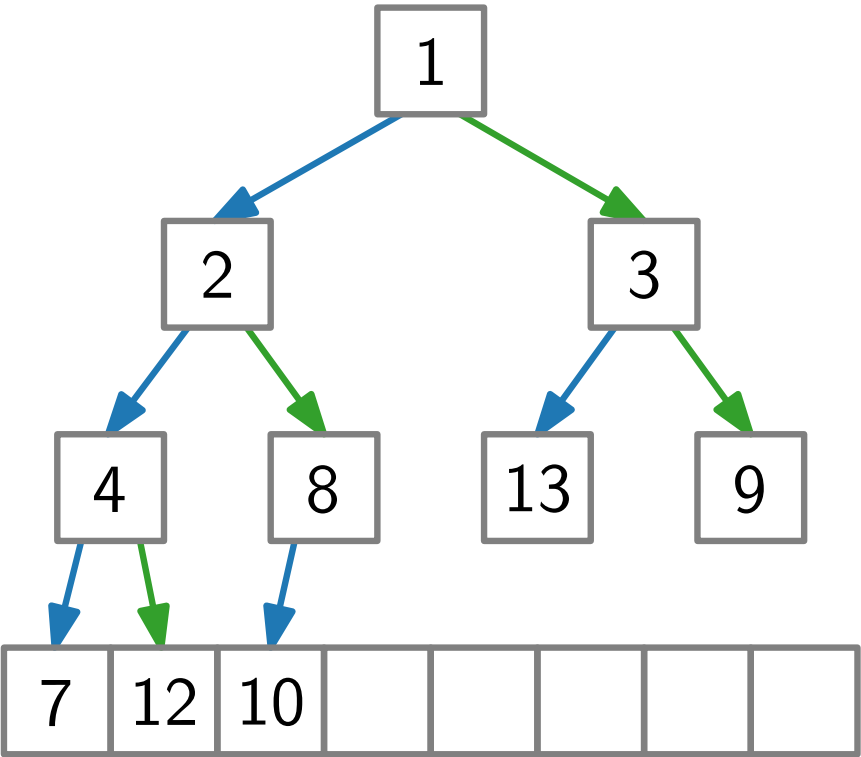
Min-Heaps



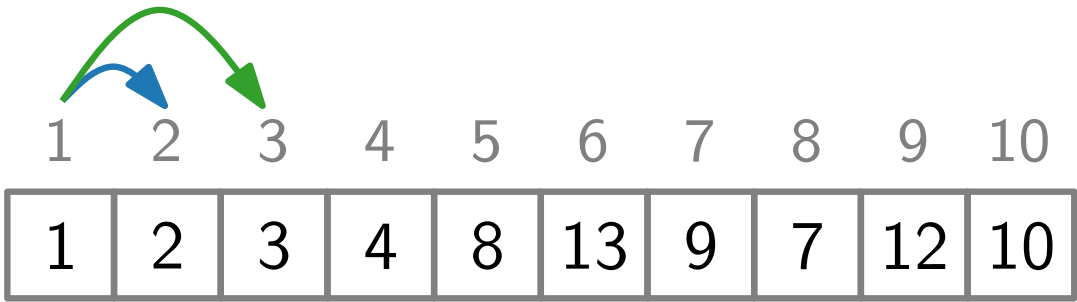
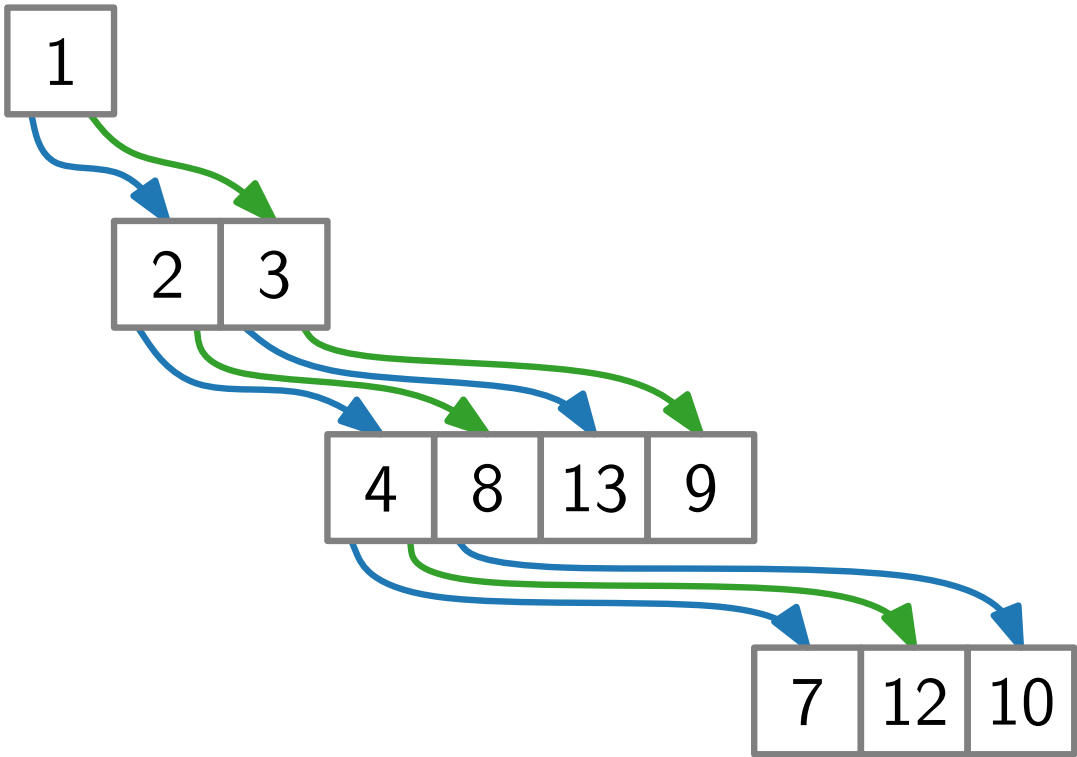
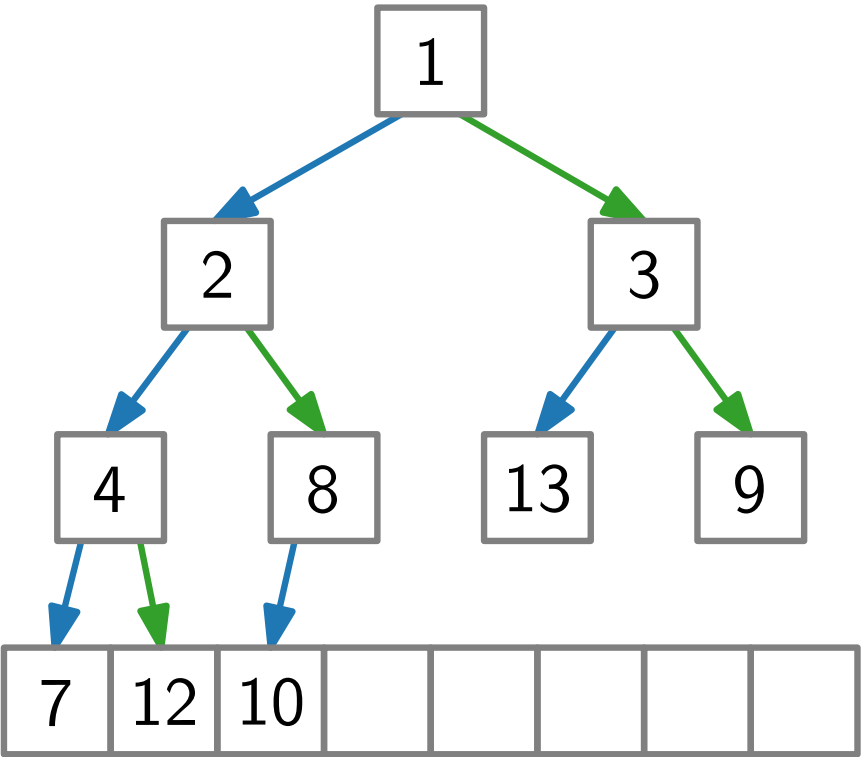
Min-Heaps



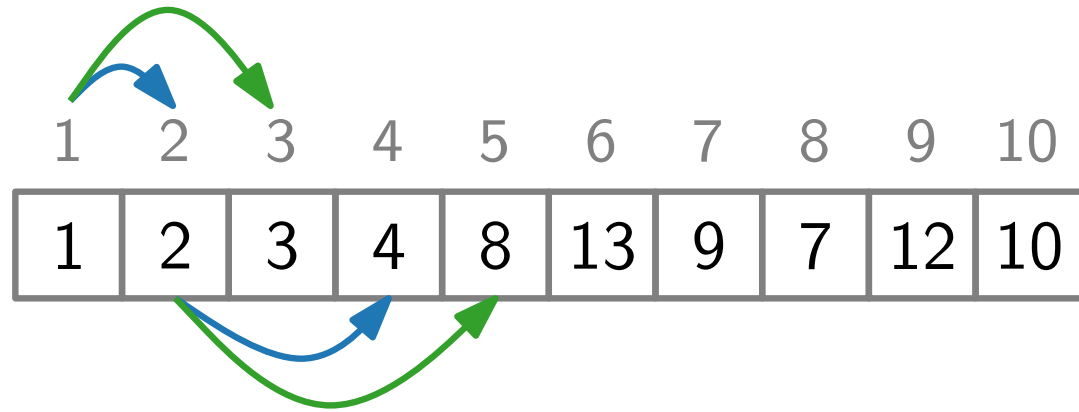
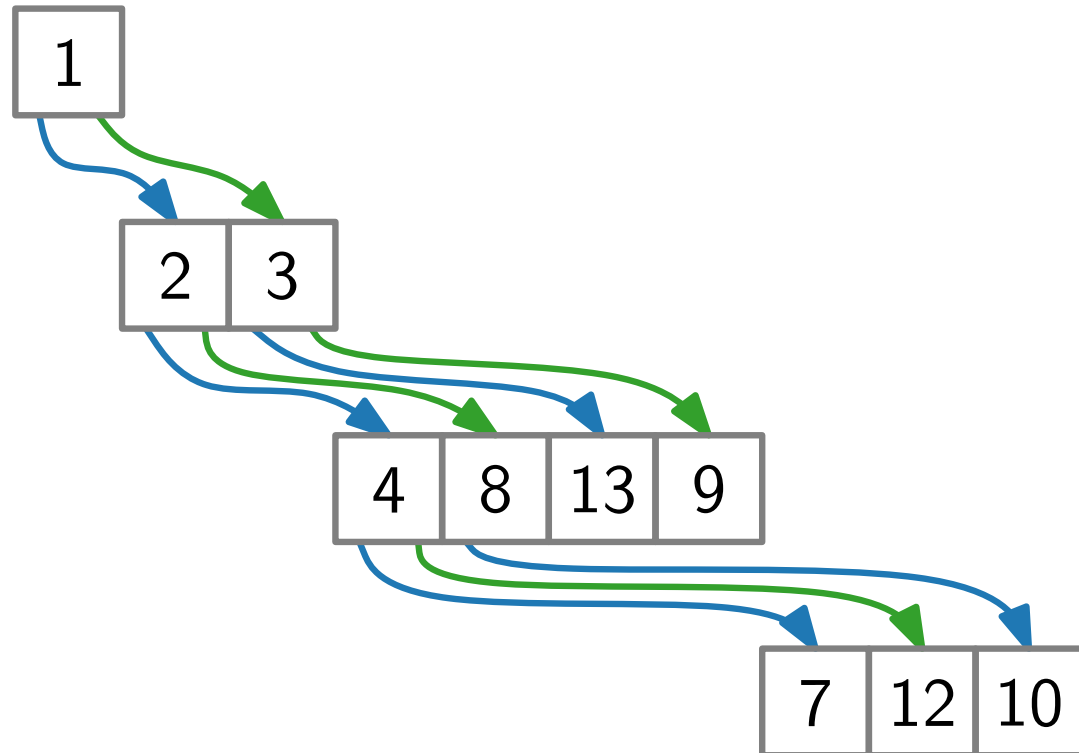
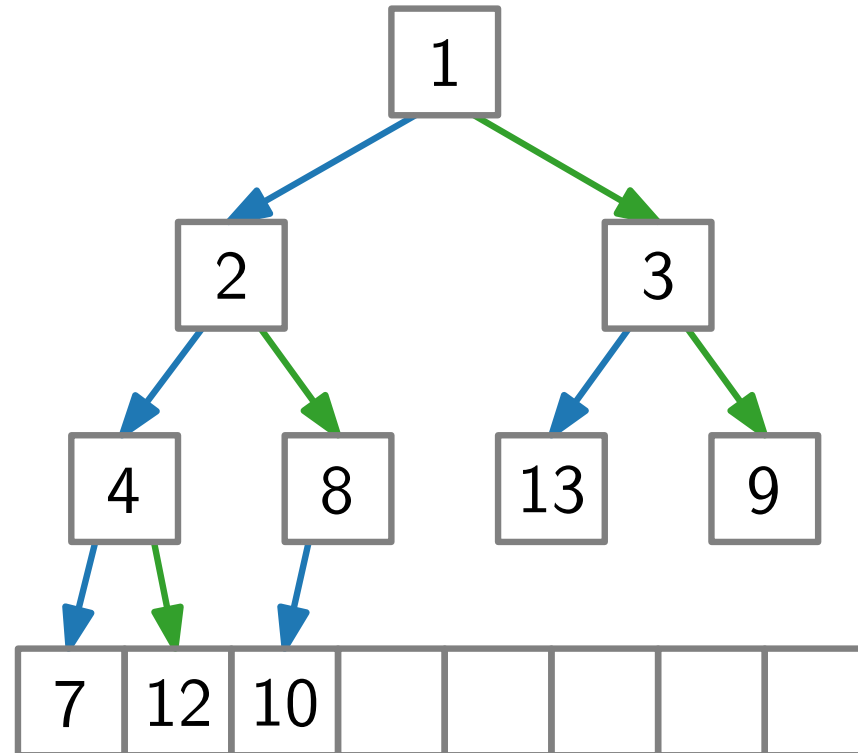
Min-Heaps



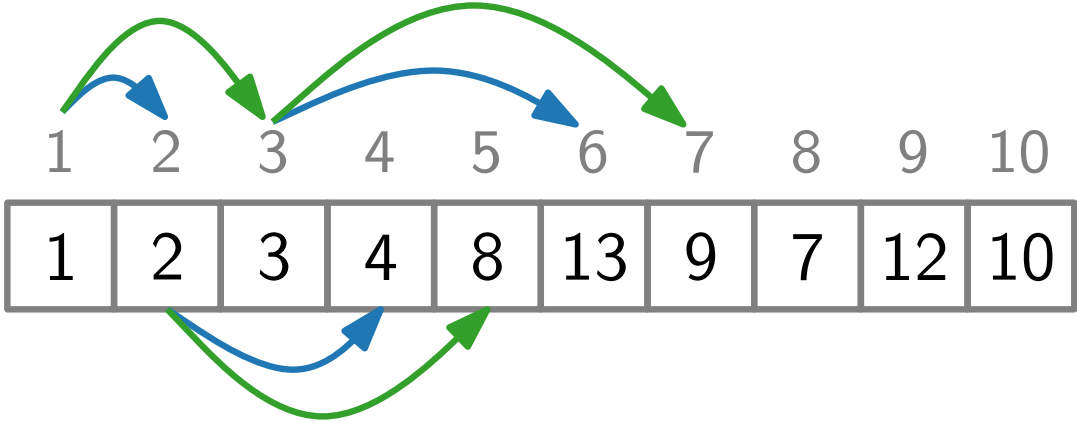
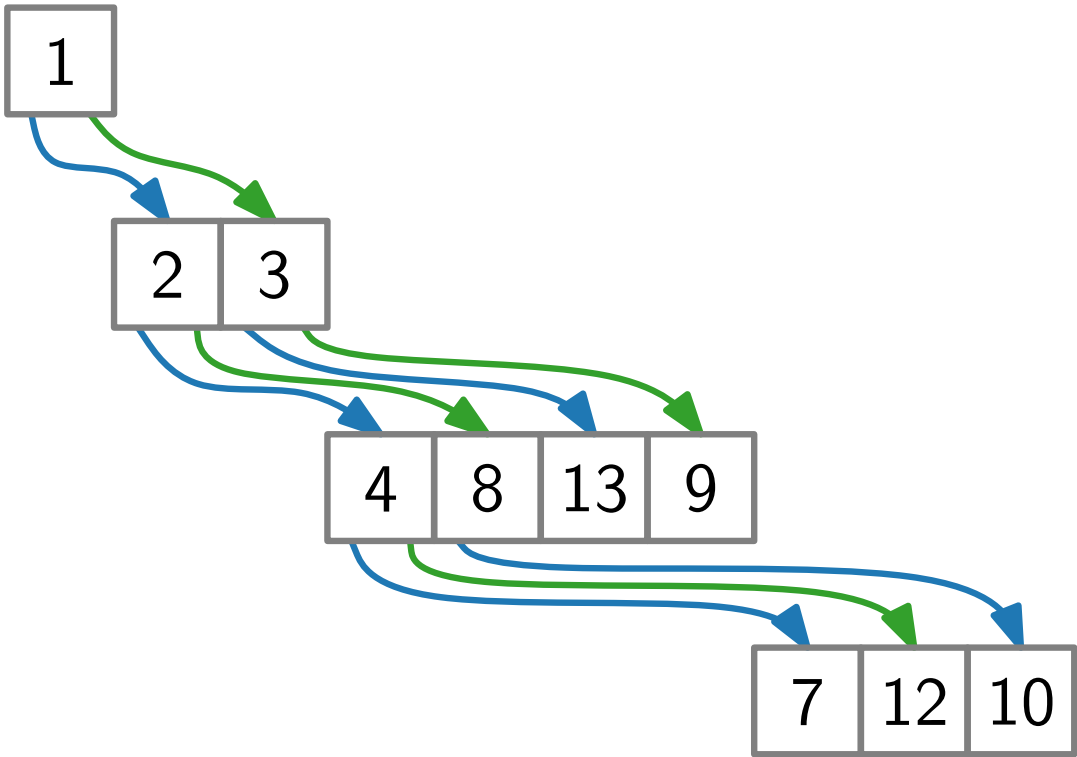
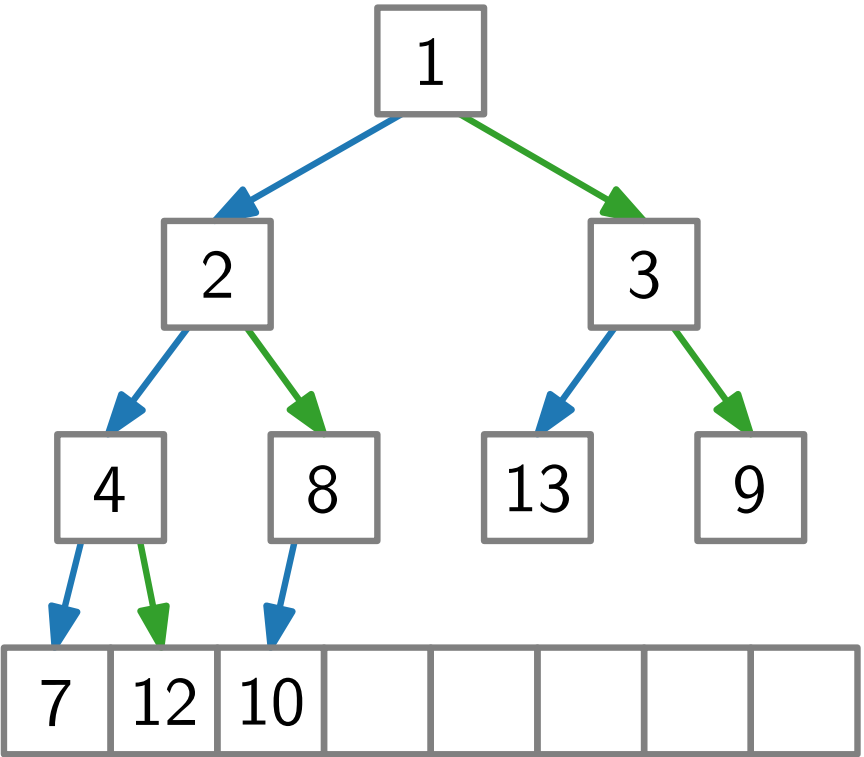
Min-Heaps



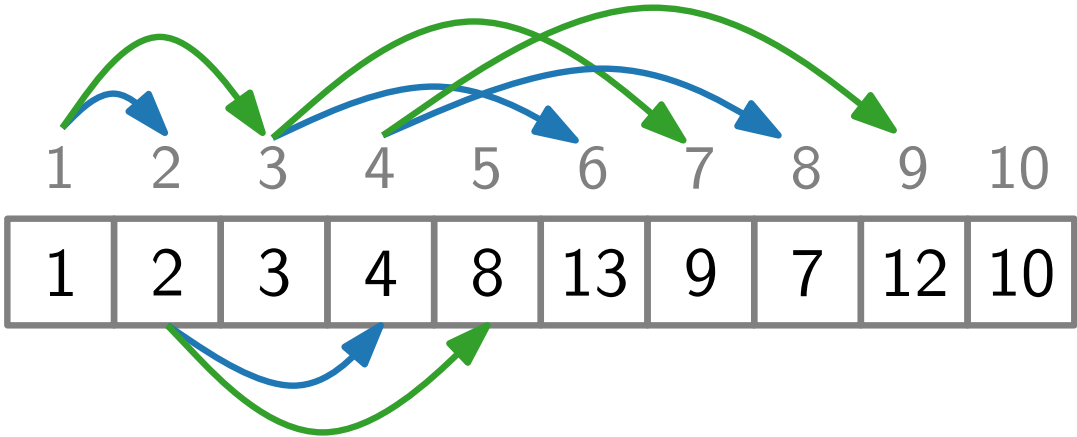
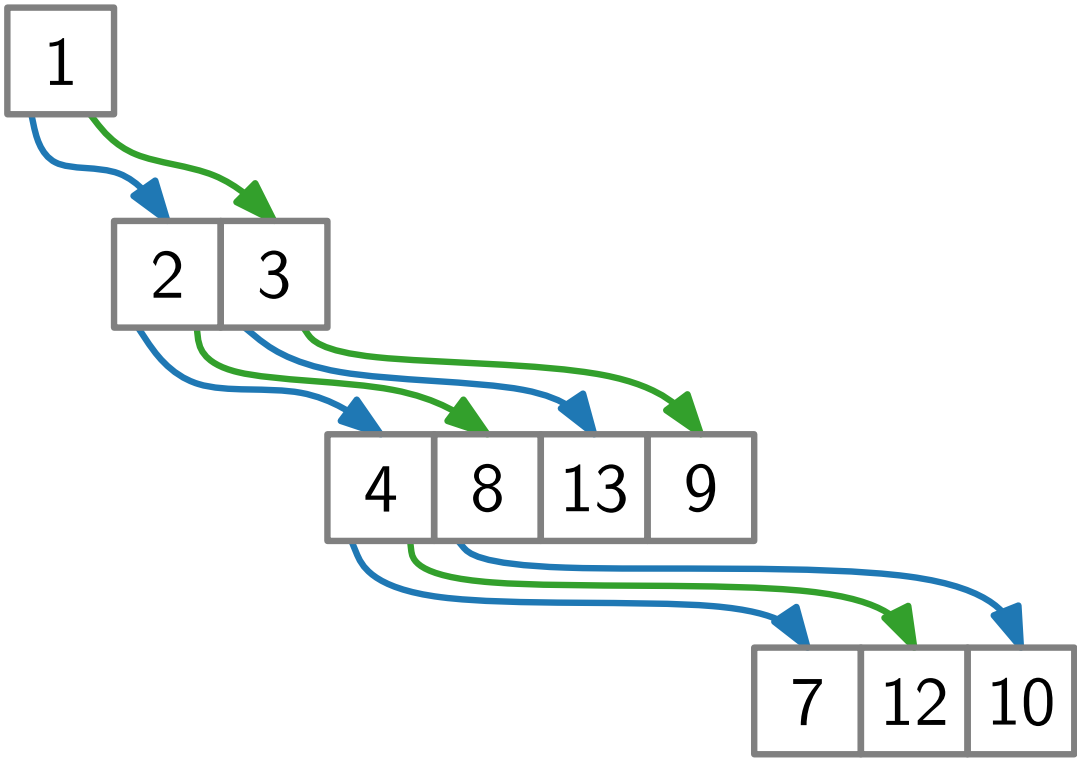
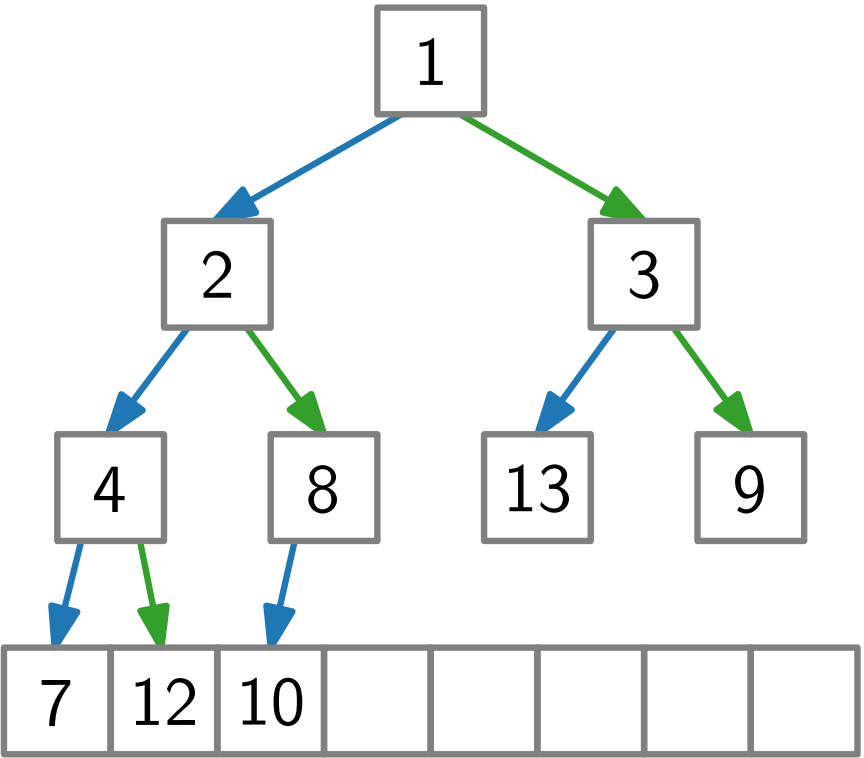
Min-Heaps



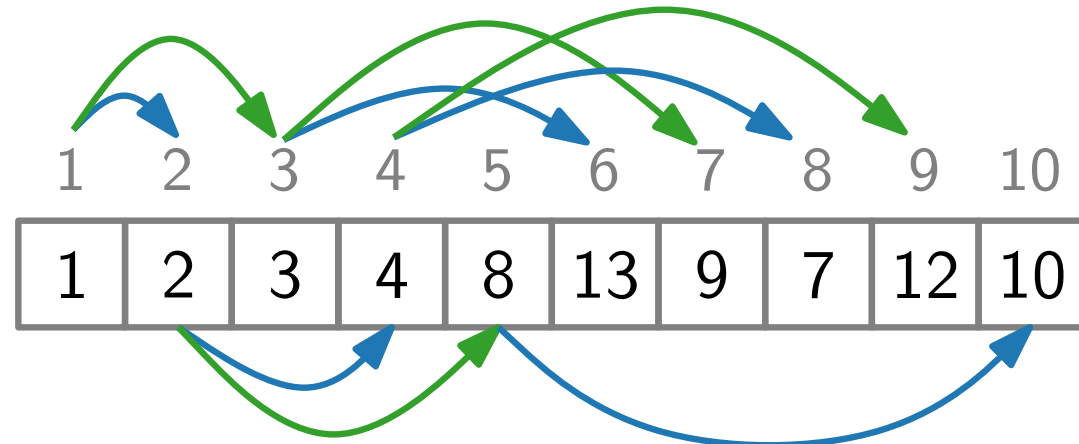
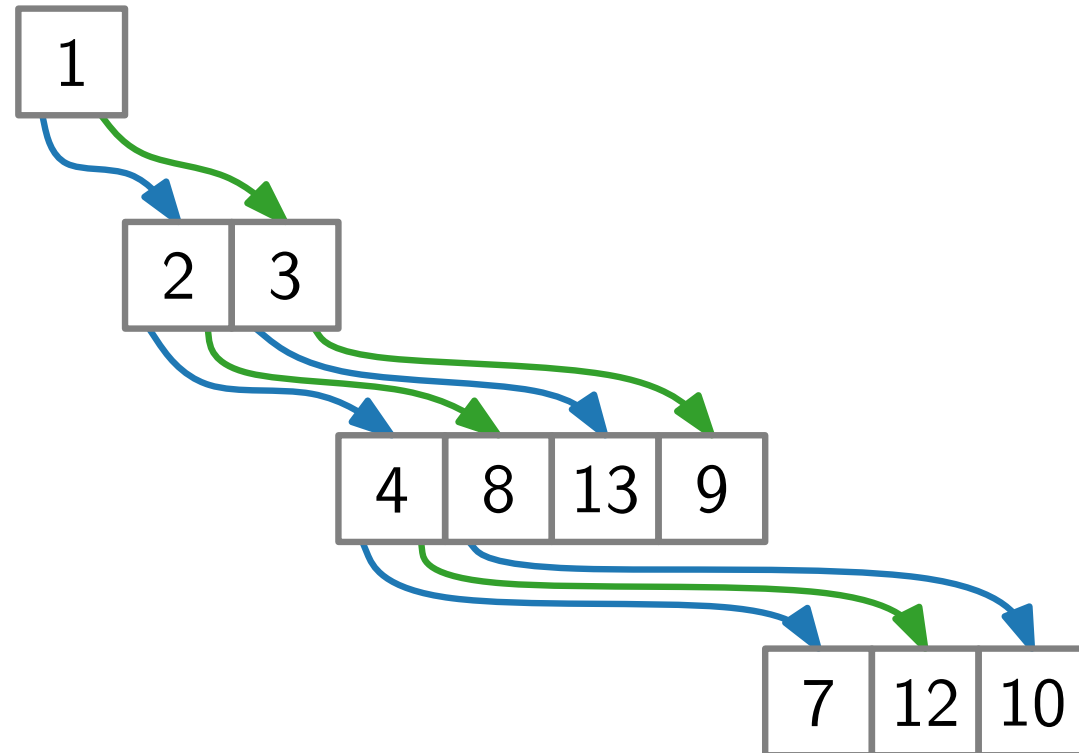
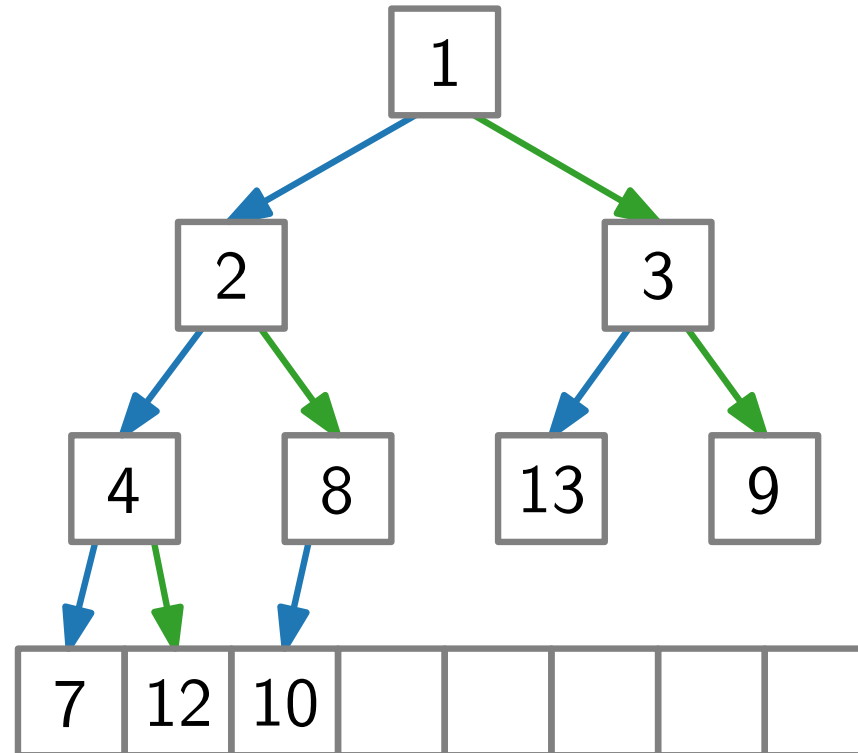
Min-Heaps



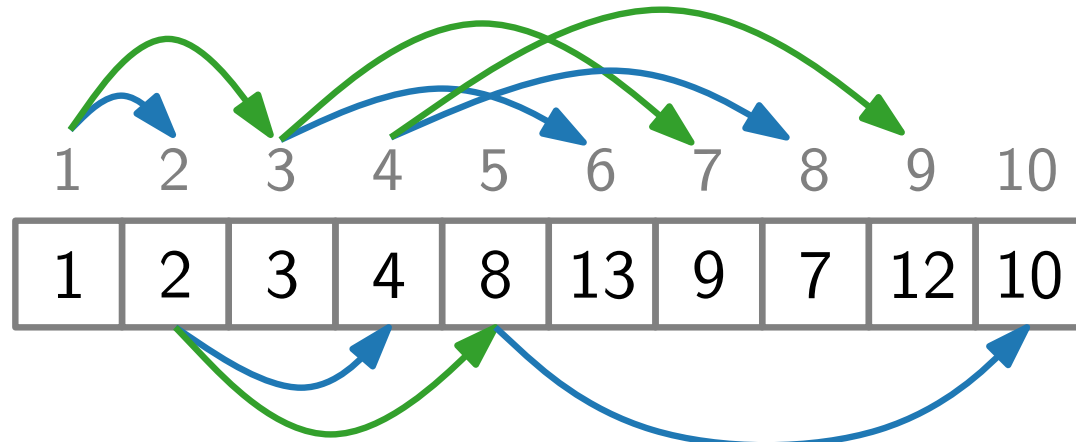
Min-Heaps



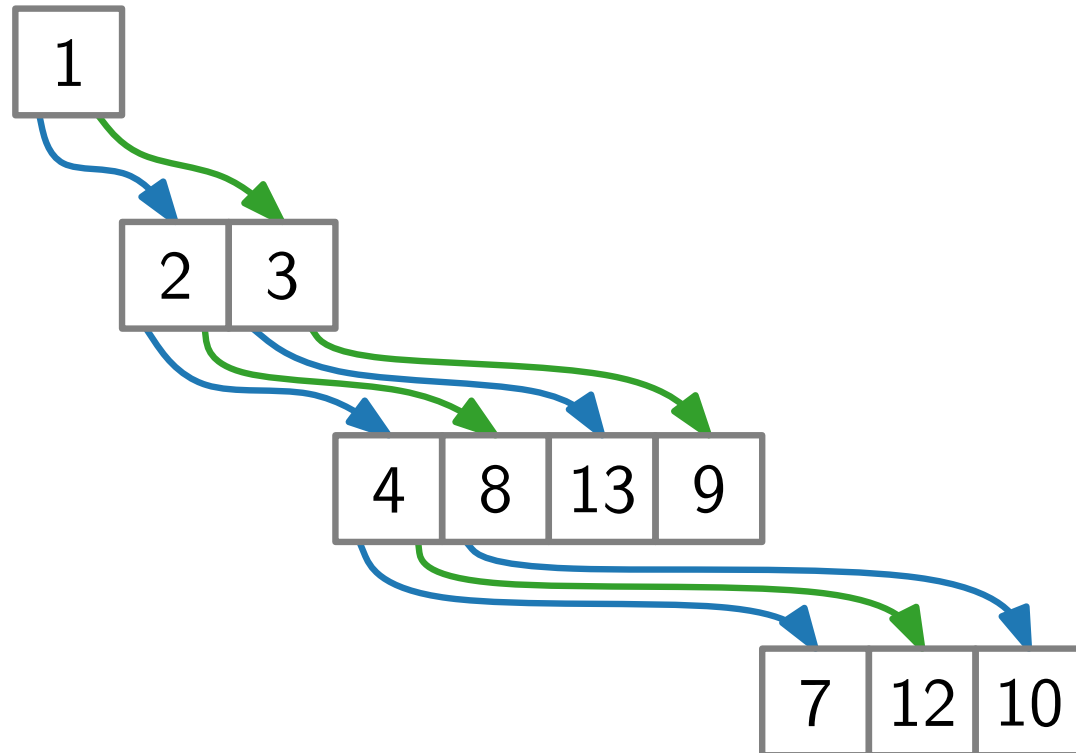
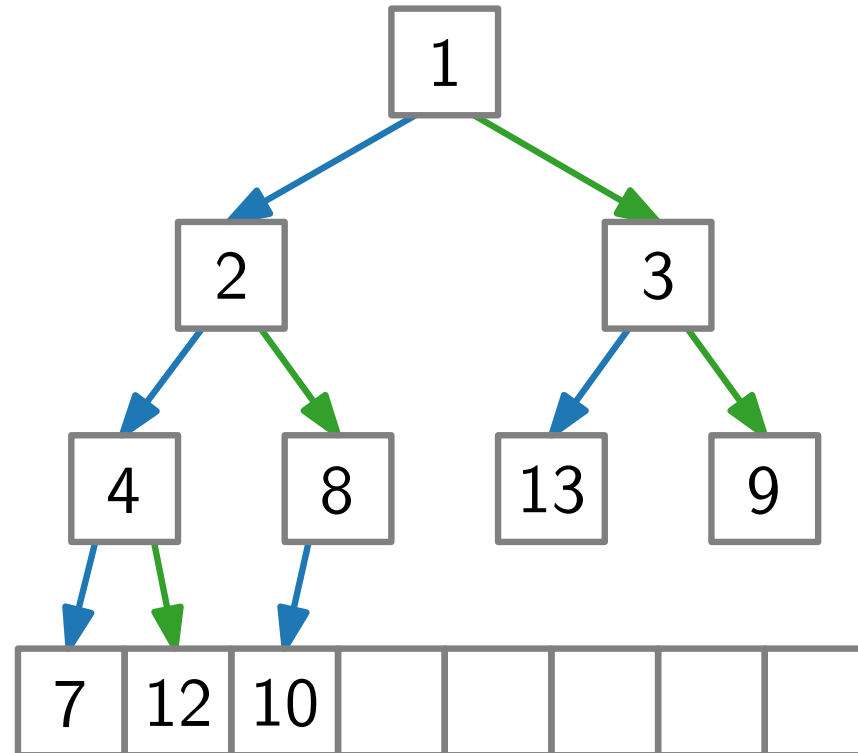
Min-Heaps



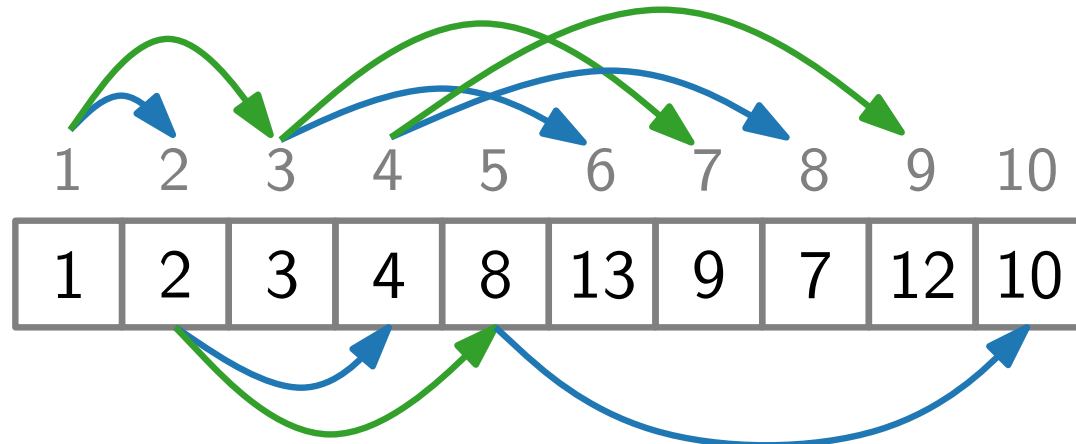
Min-Heaps



Pfeile implementieren:



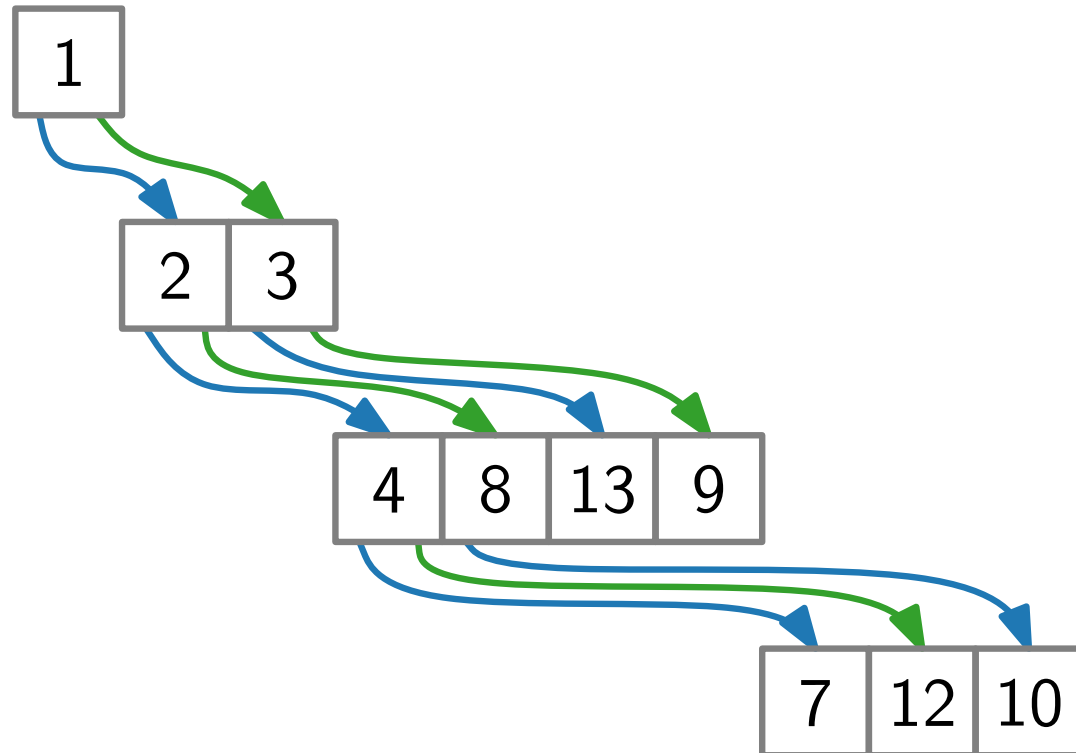
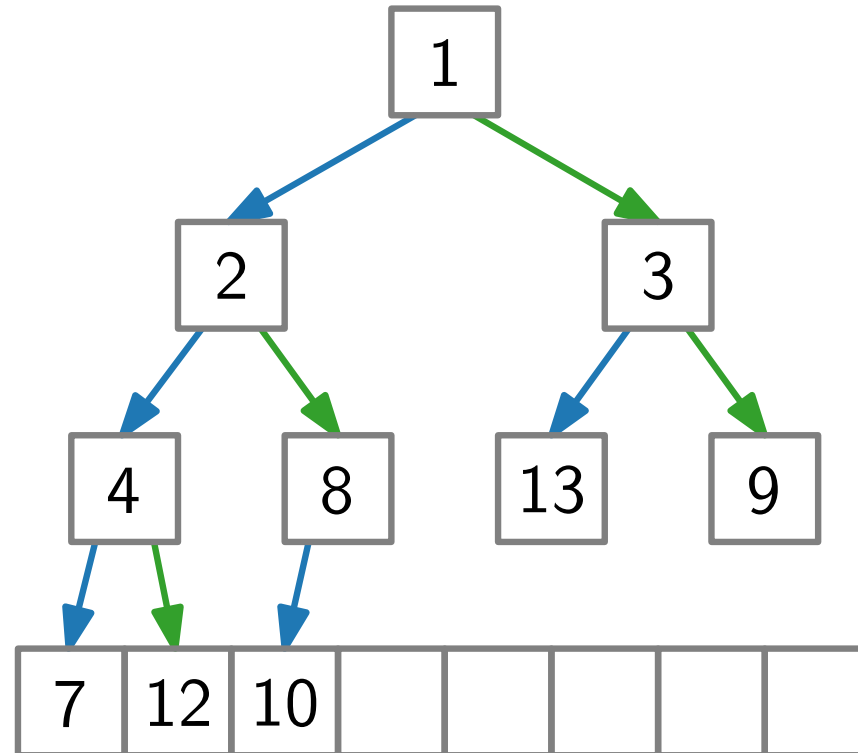
Min-Heaps



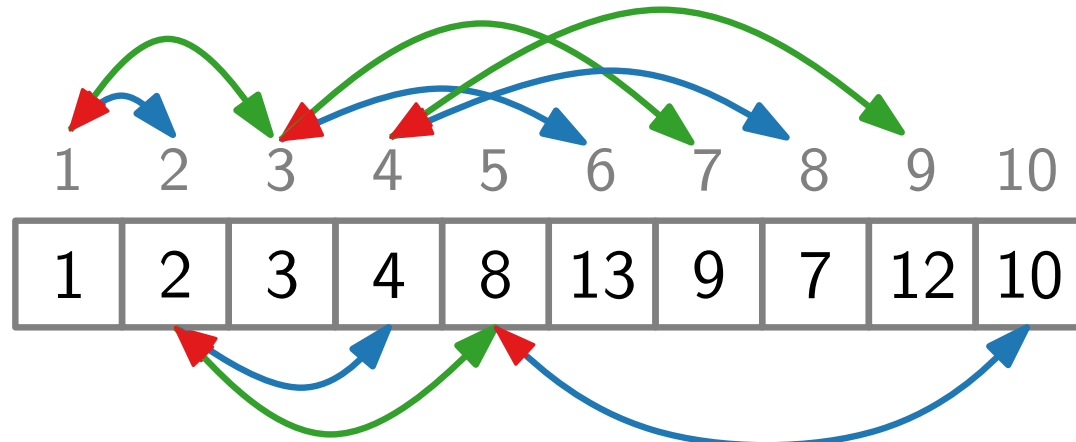
Pfeile implementieren:

`LEFT(index  $i$ )` **return ...**

`RIGHT(index  $i$ )` **return ...**



Min-Heaps

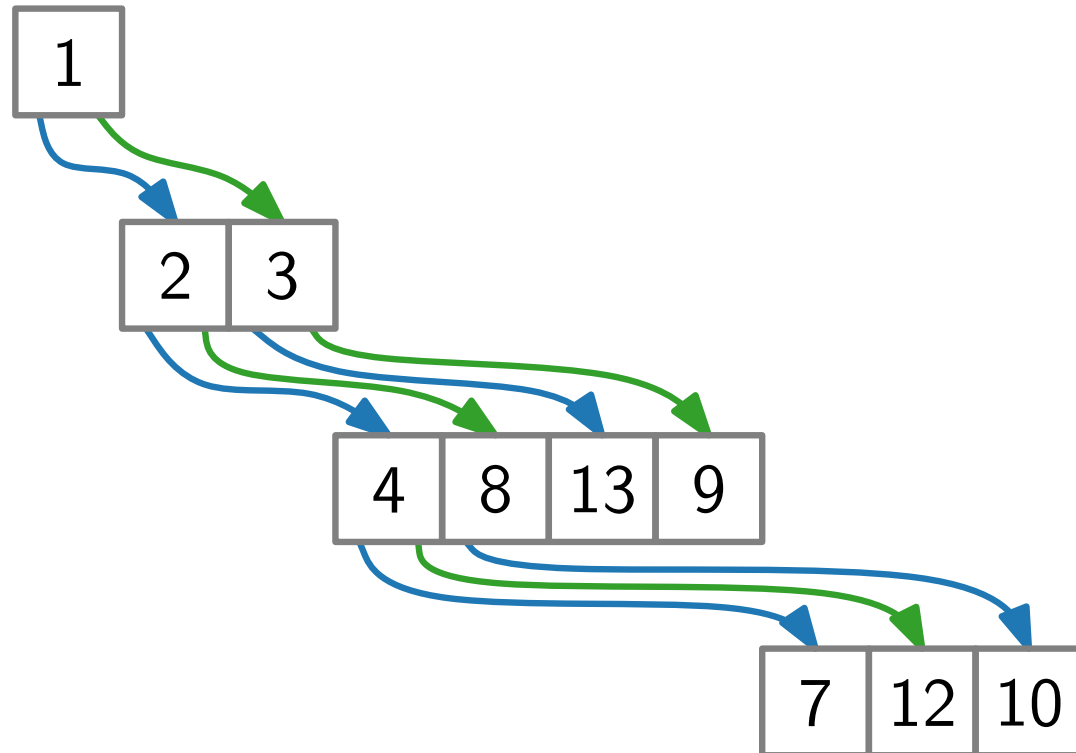
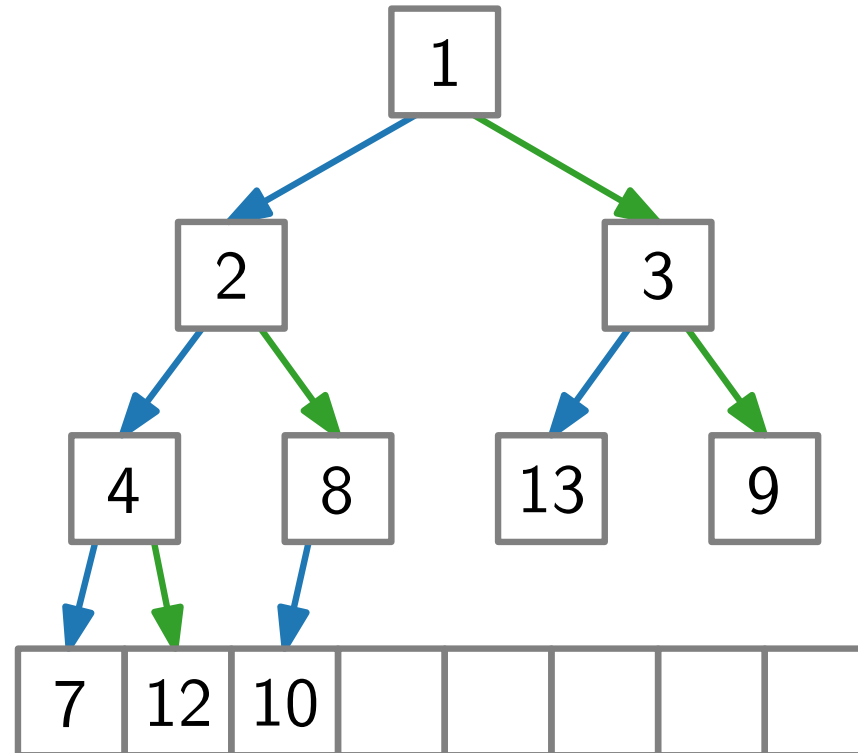


Pfeile implementieren:

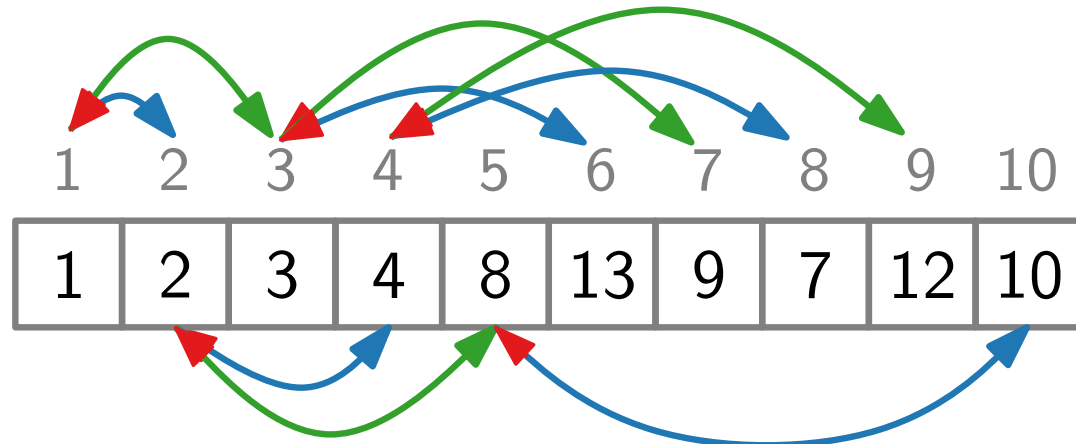
LEFT(index i) **return** ...

RIGHT(index i) **return** ...

PARENT(index i) **return** ...



Min-Heaps

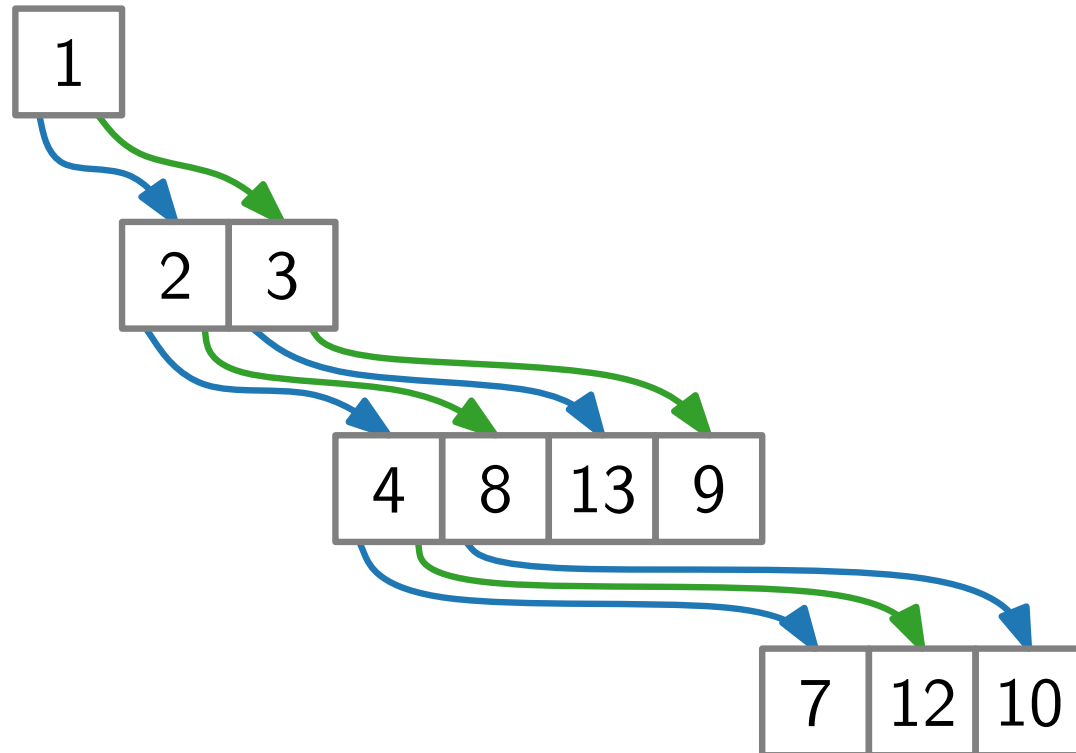
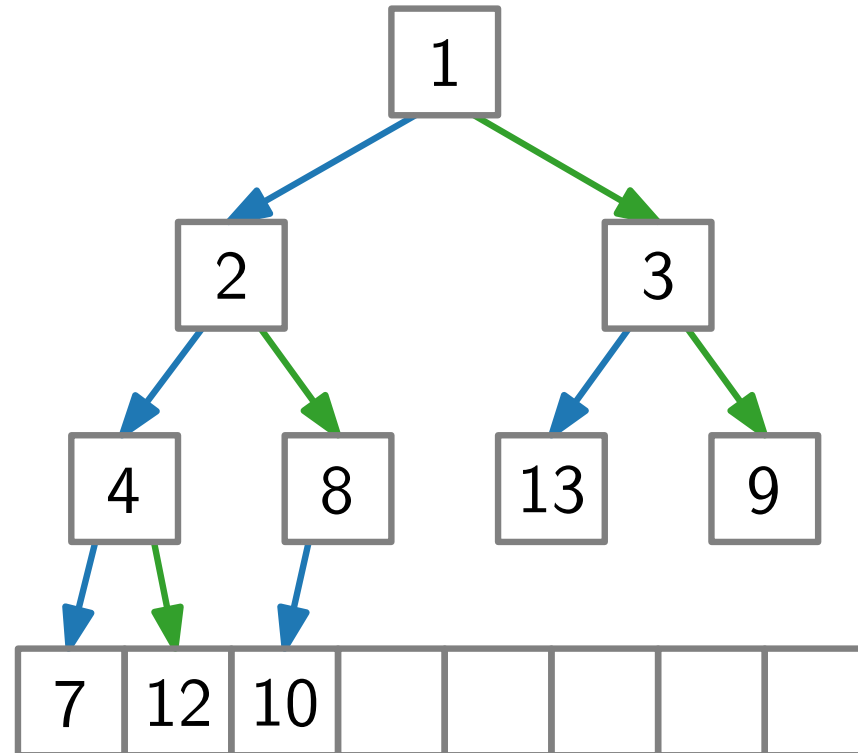


Pfeile implementieren:

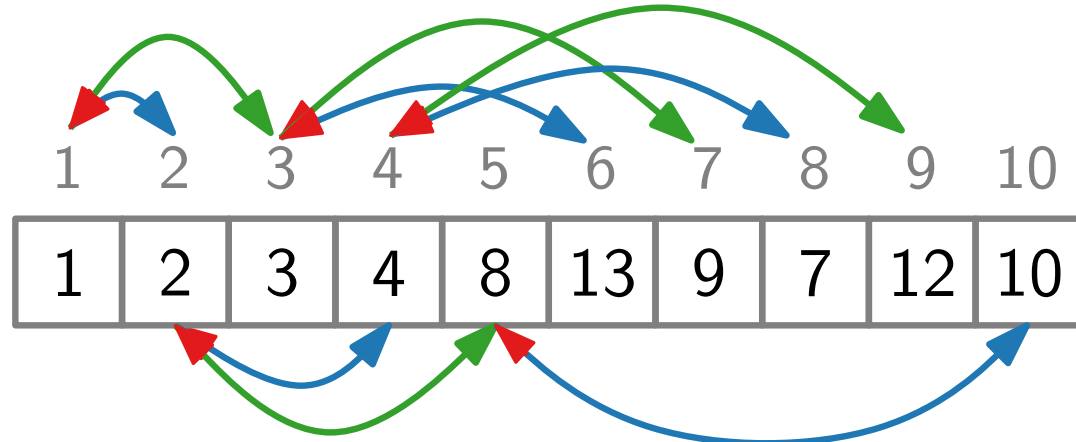
LEFT(index i) **return** $2i$

RIGHT(index i) **return** ...

PARENT(index i) **return** ...



Min-Heaps

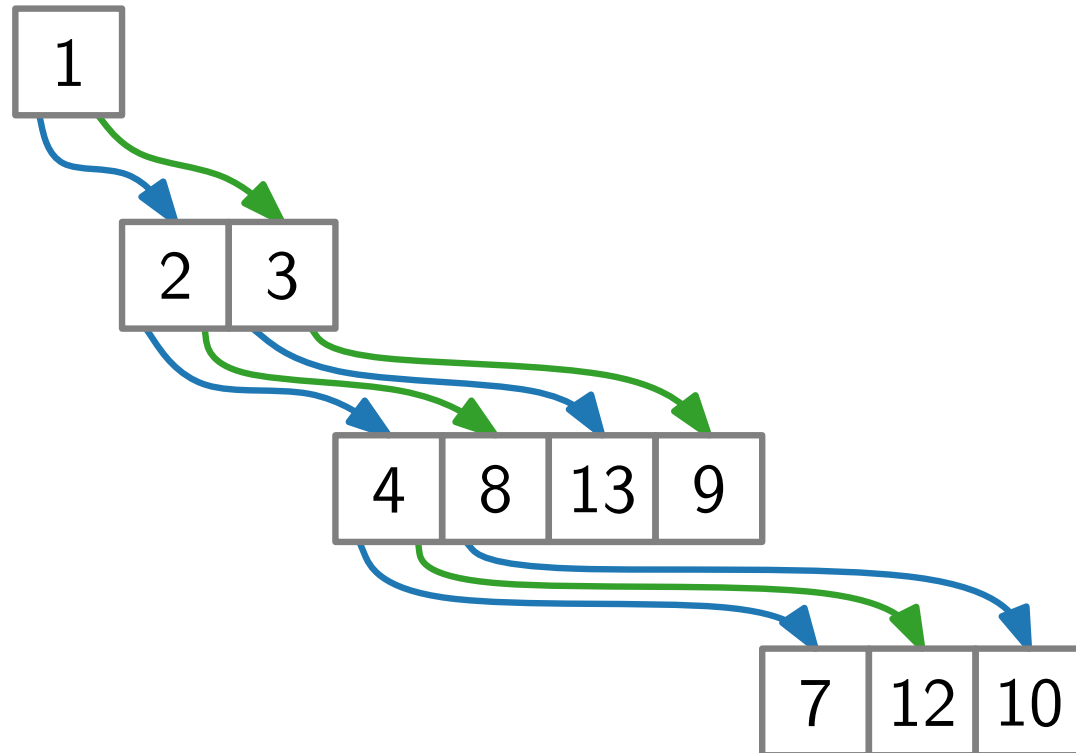
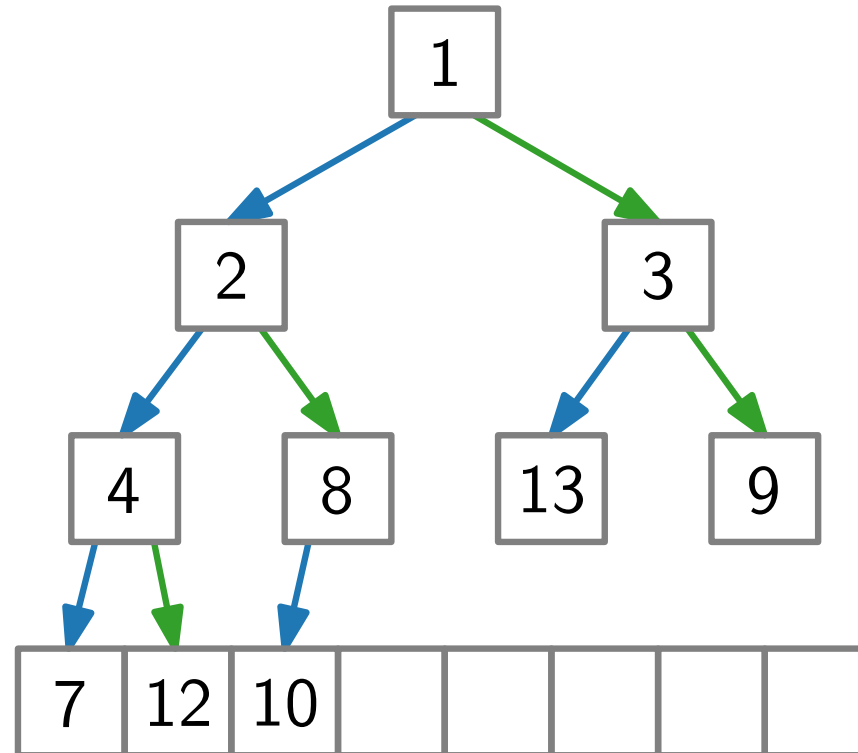


Pfeile implementieren:

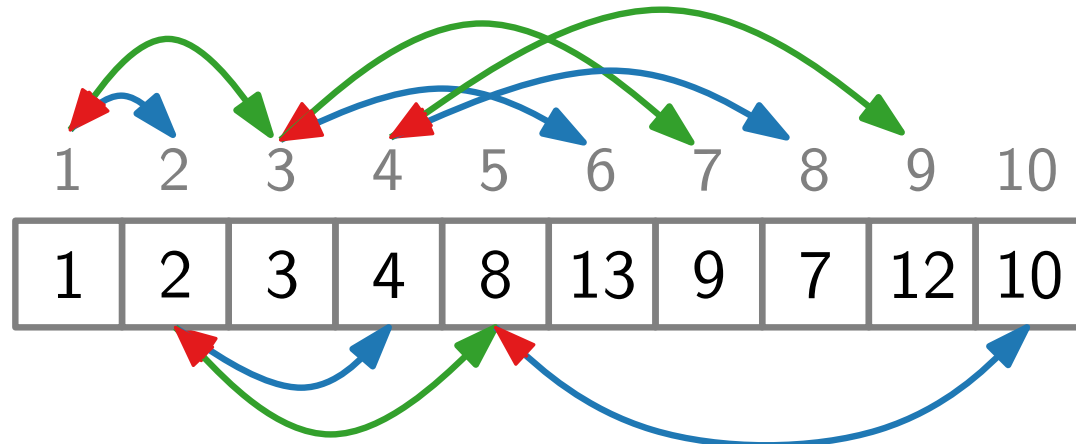
LEFT(index i) **return** $2i$

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

PARENT(index i) **return** ...



Min-Heaps

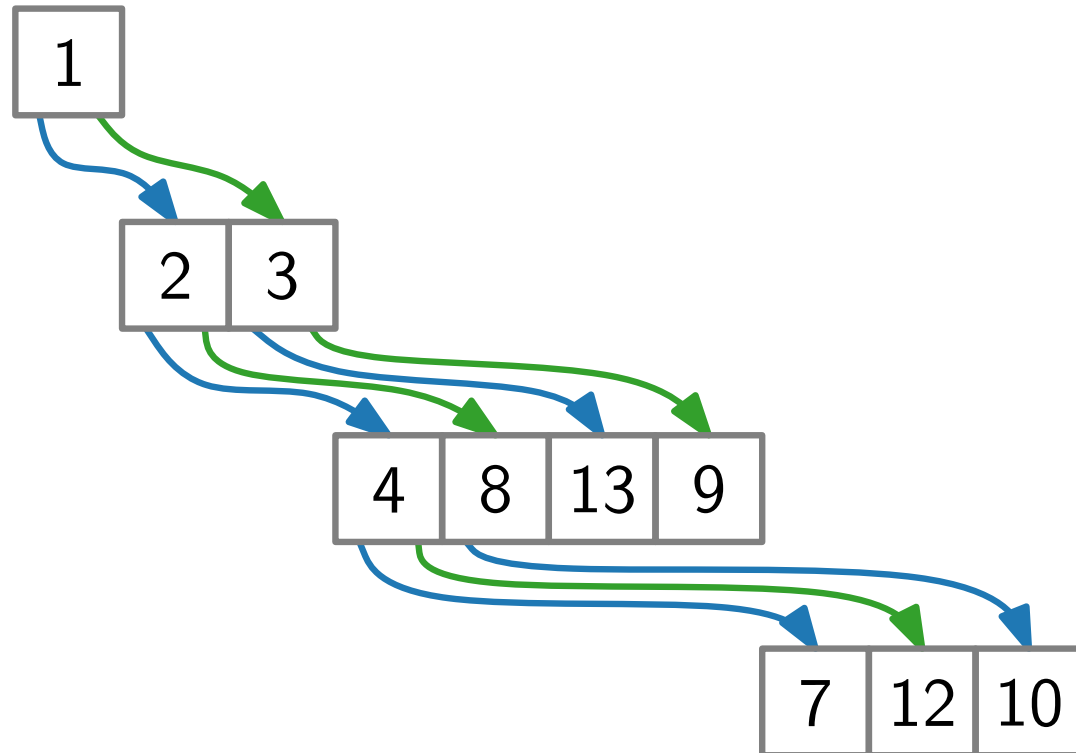
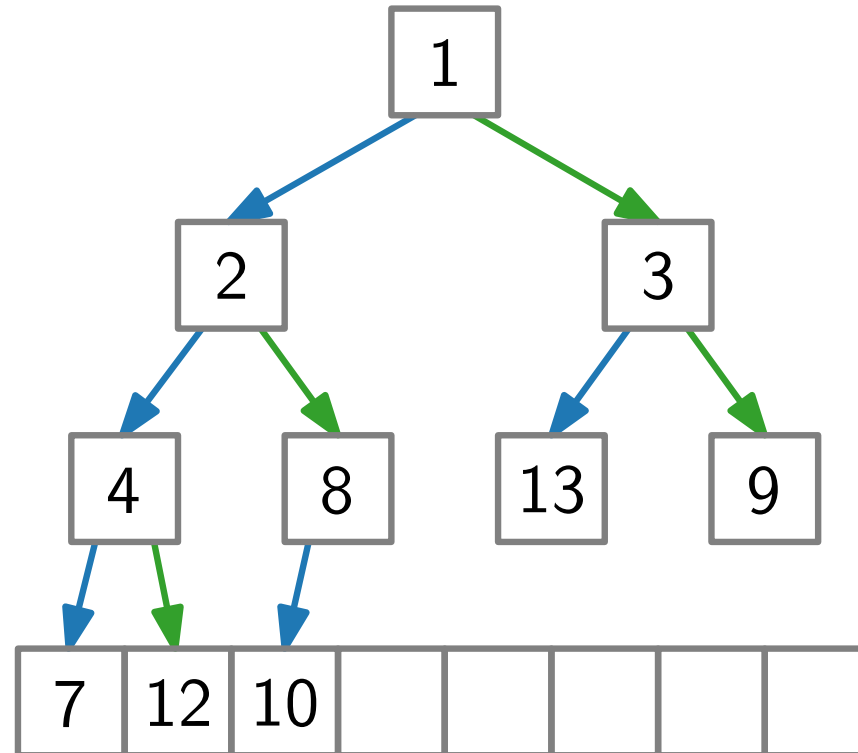


Pfeile implementieren:

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

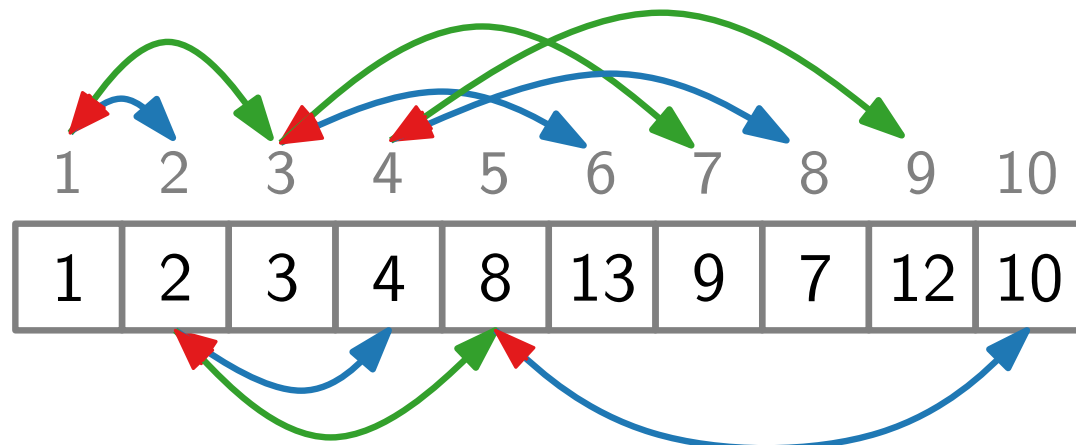
RIGHT(index i) **return** $2i + 1$

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



Min-Heaps

sehr schnelle Rechenoperationen!

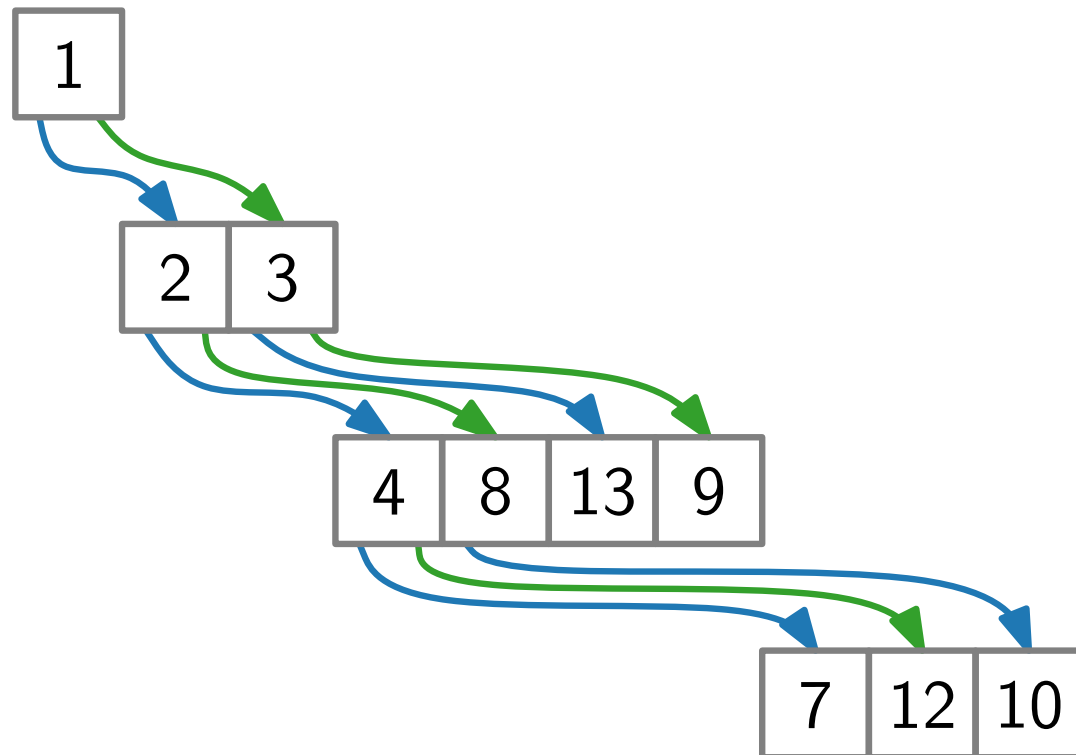
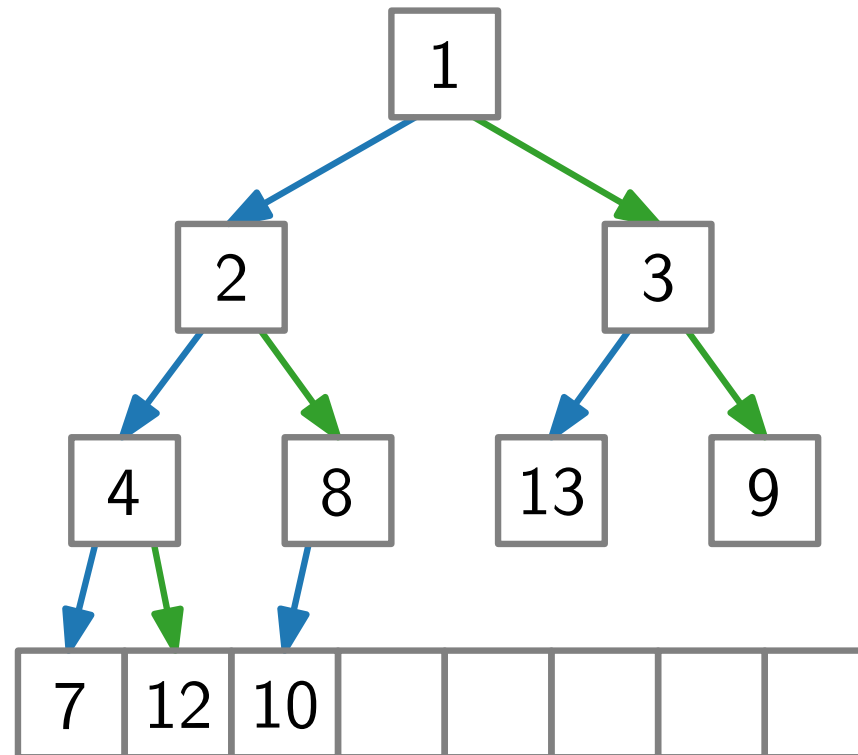


Pfeile implementieren:

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

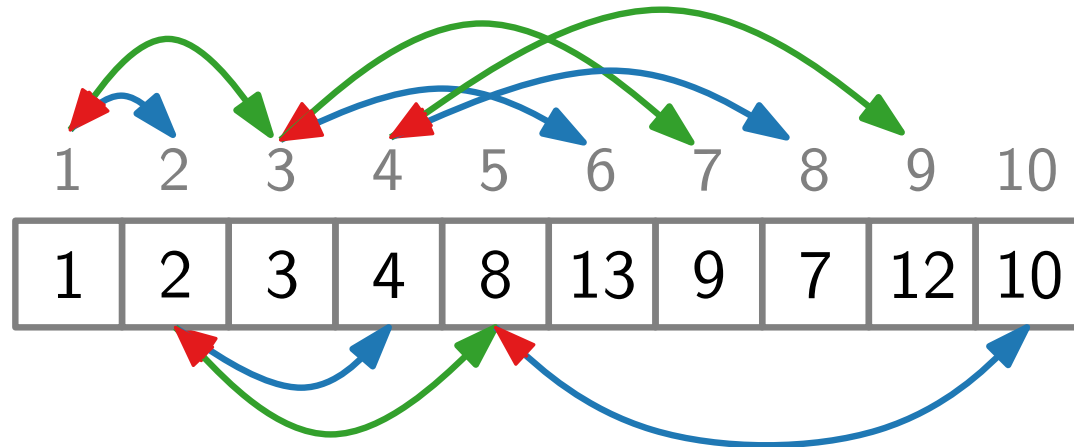
RIGHT(index i) **return** $2i + 1$

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



Min-Heaps

sehr schnelle Rechenoperationen!



Pfeile implementieren:

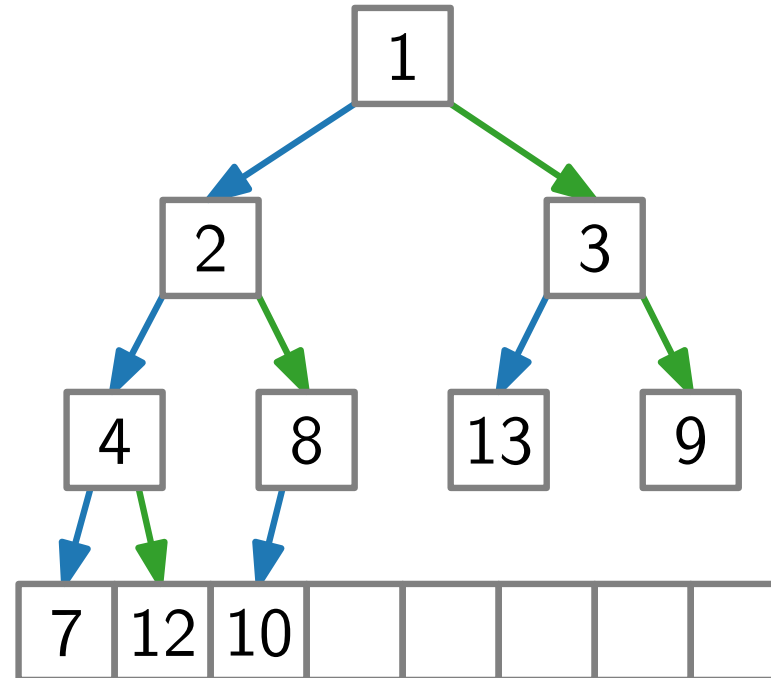
LEFT(index i) **return** $2i$

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

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

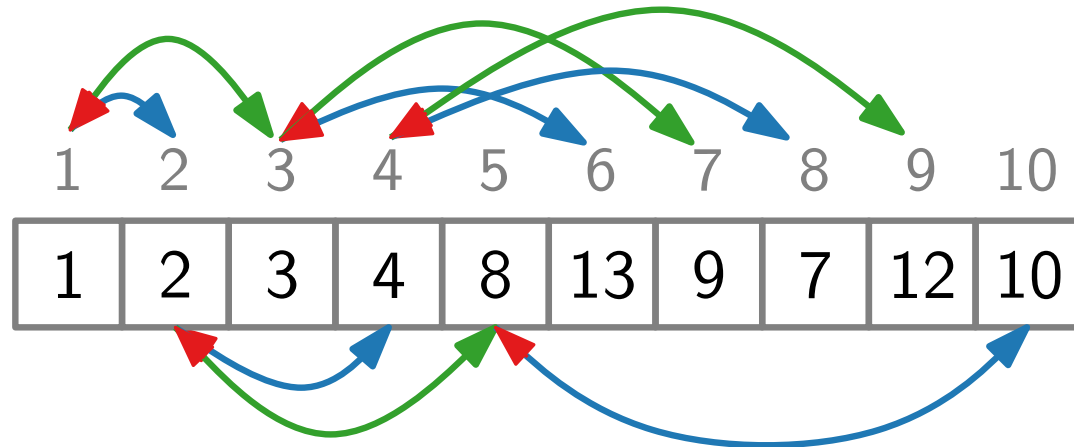
Definition.

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



Min-Heaps

sehr schnelle Rechenoperationen!

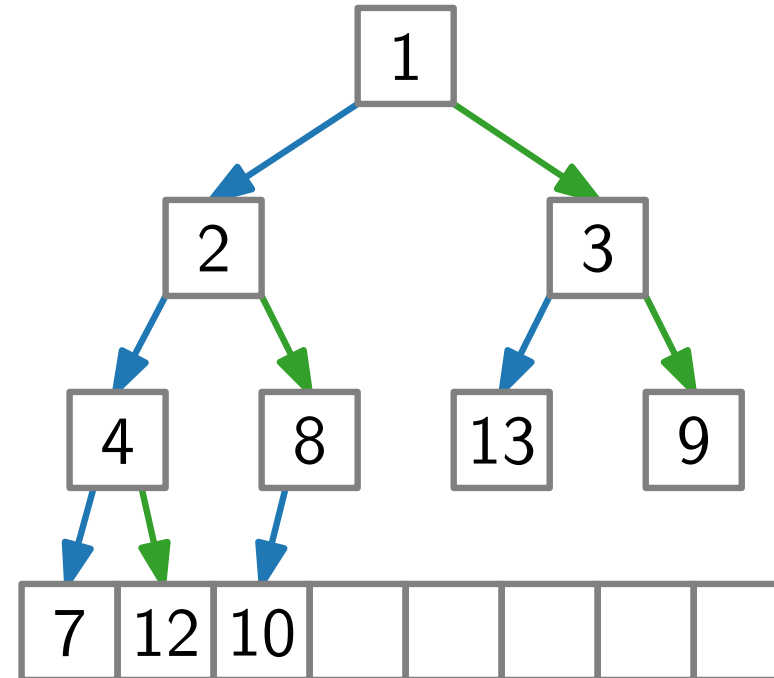


Pfeile implementieren:

`LEFT(index  $i$ )` `return` $2i$
`RIGHT(index  $i$ )` `return` $2i + 1$
`PARENT(index  $i$ )` `return` $\lfloor i/2 \rfloor$

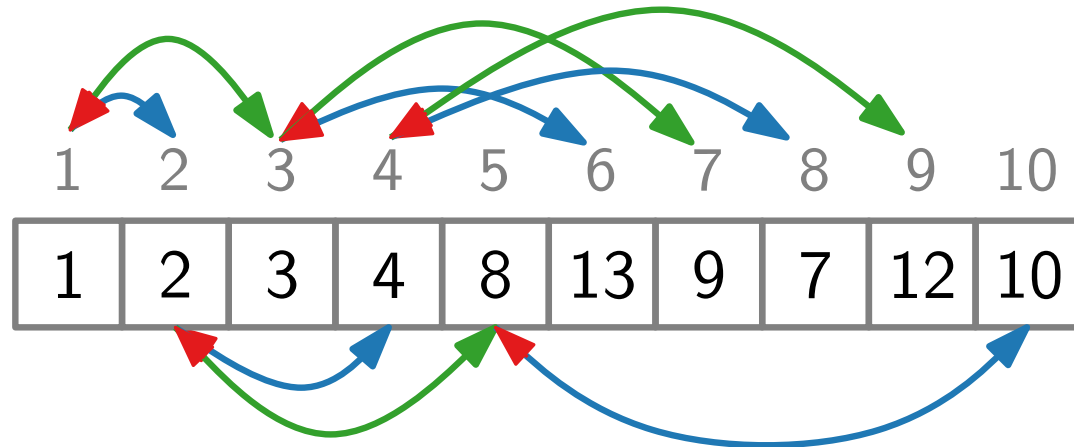
Definition.

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



Min-Heaps

sehr schnelle Rechenoperationen!



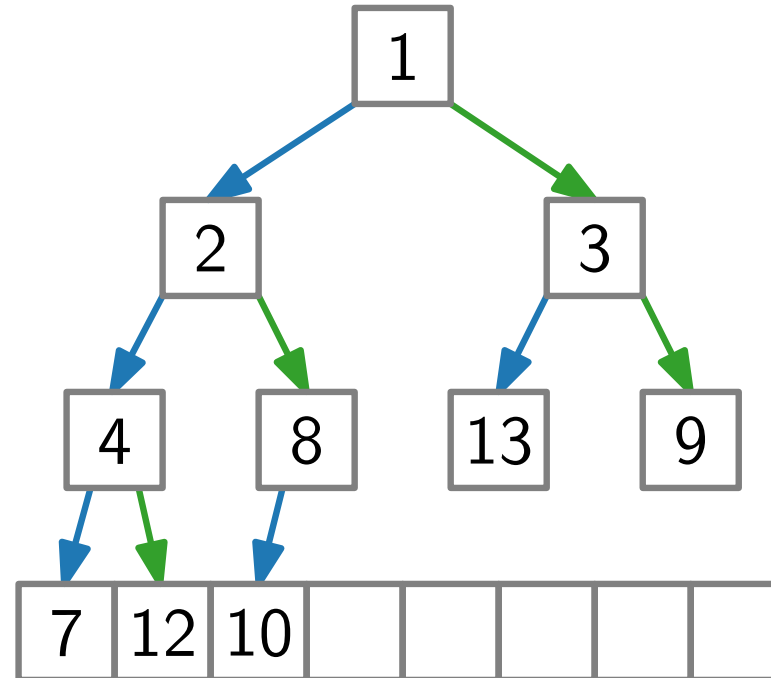
Pfeile implementieren:

`LEFT(index  $i$ )` **return** $2i$
`RIGHT(index  $i$ )` **return** $2i + 1$
`PARENT(index  $i$ )` **return** $\lfloor i/2 \rfloor$

Definition.

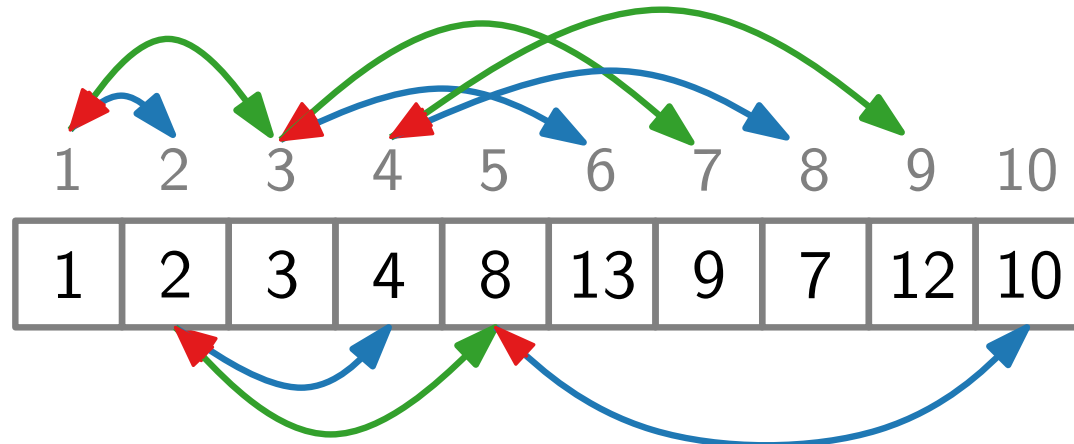
Ein **Heap** ist ein Feld, das einem **binären** Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,



Min-Heaps

sehr schnelle Rechenoperationen!



Pfeile implementieren:

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

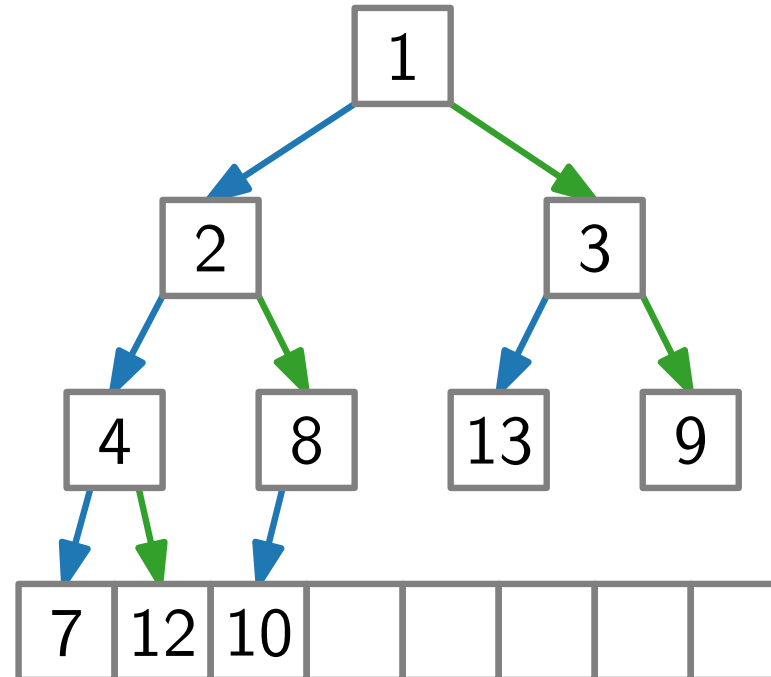
RIGHT(index i) **return** $2i + 1$

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

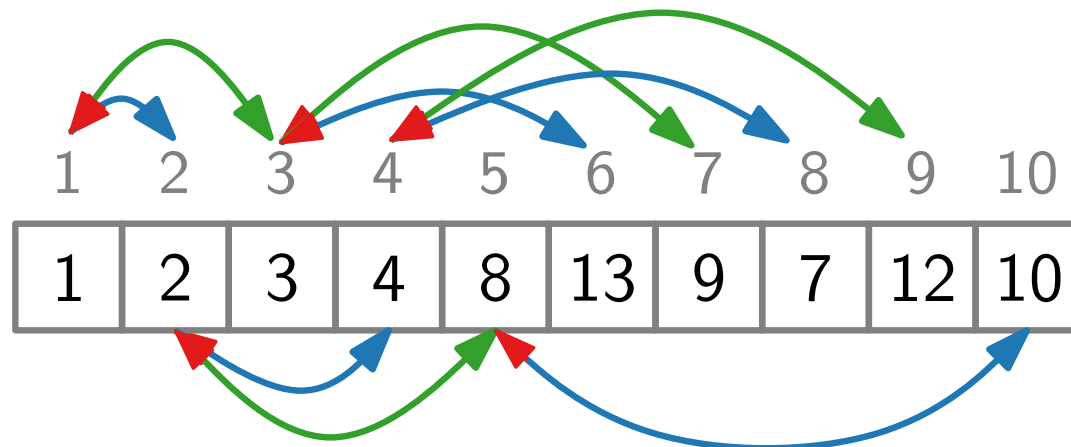
Definition.

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



Min-Heaps



sehr schnelle Rechenoperationen!

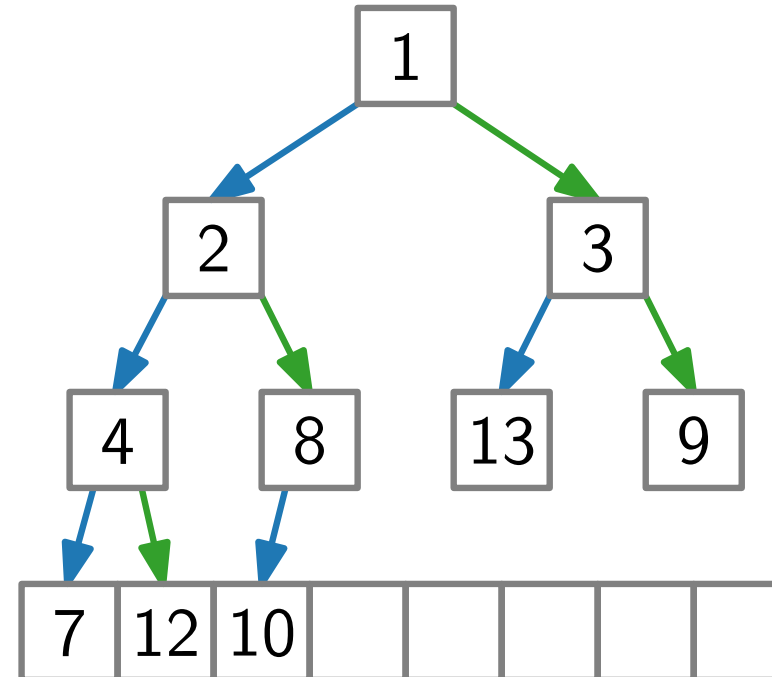
Pfeile implementieren:

`LEFT(index  $i$ )` `return` $2i$
`RIGHT(index  $i$ )` `return` $2i + 1$
`PARENT(index  $i$ )` `return` $\lfloor i/2 \rfloor$

Definition.

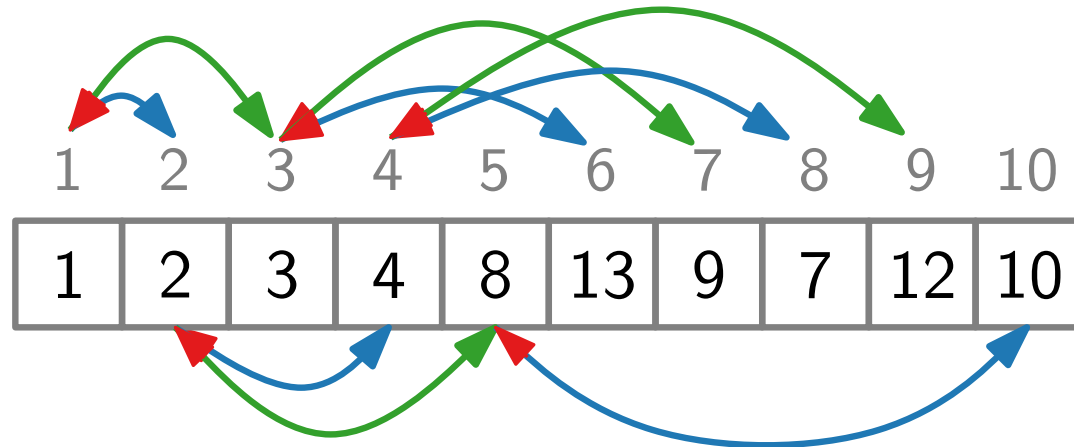
Ein **Heap** ist ein Feld, das einem **binären** Baum entspricht, bei dem

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



Min-Heaps

sehr schnelle Rechenoperationen!



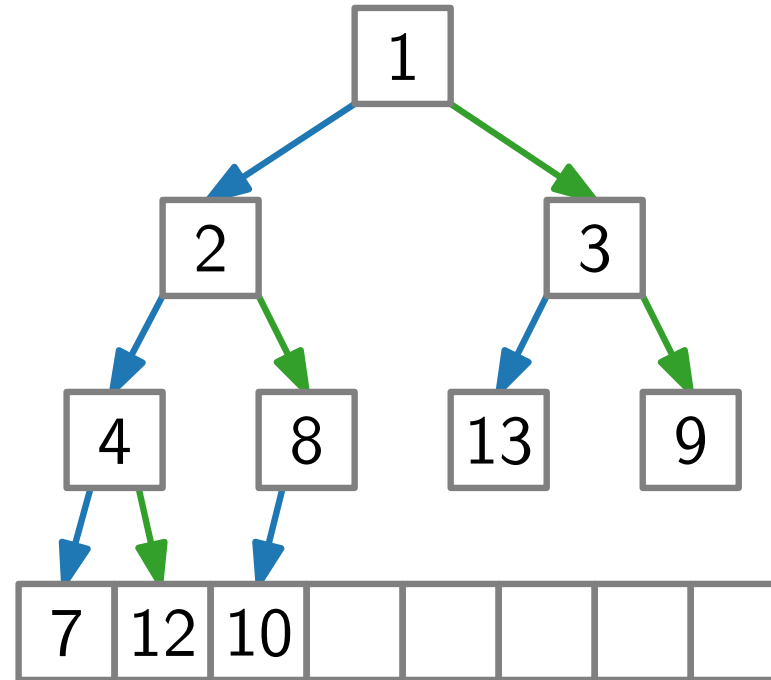
Pfeile implementieren:

`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.

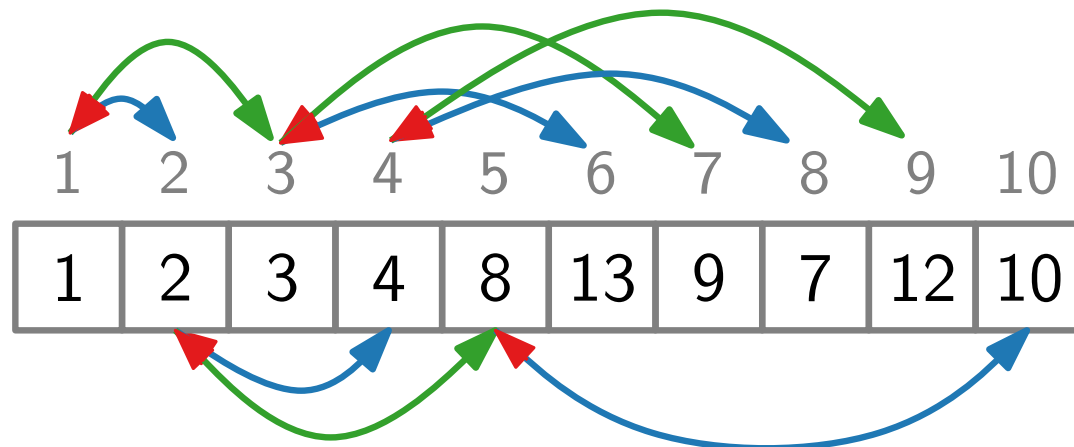


Definition.

Ein Heap hat die **Min-Heap-Eigenschaft**,

Min-Heaps

sehr schnelle Rechenoperationen!



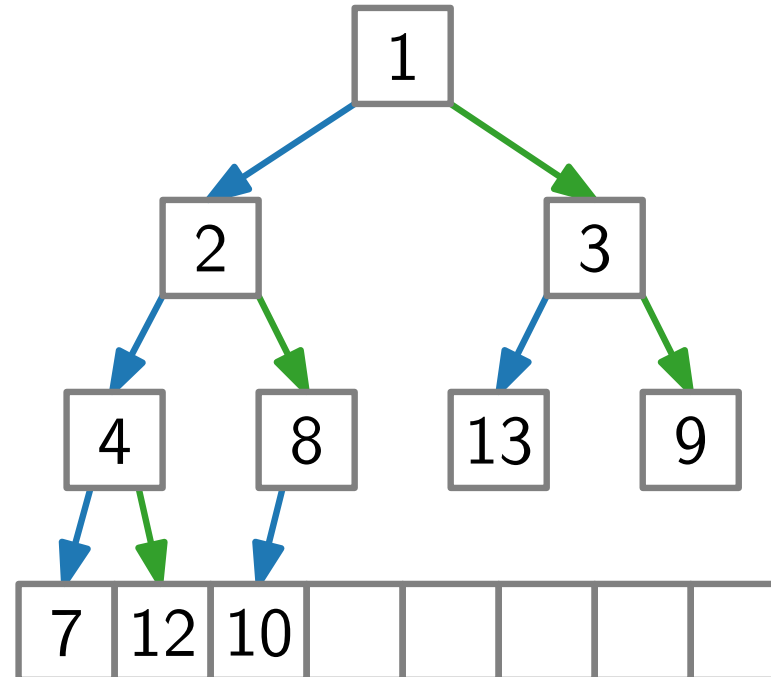
Pfeile implementieren:

$\text{LEFT}(\text{index } i)$ **return** $2i$
 $\text{RIGHT}(\text{index } i)$ **return** $2i + 1$
 $\text{PARENT}(\text{index } i)$ **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.



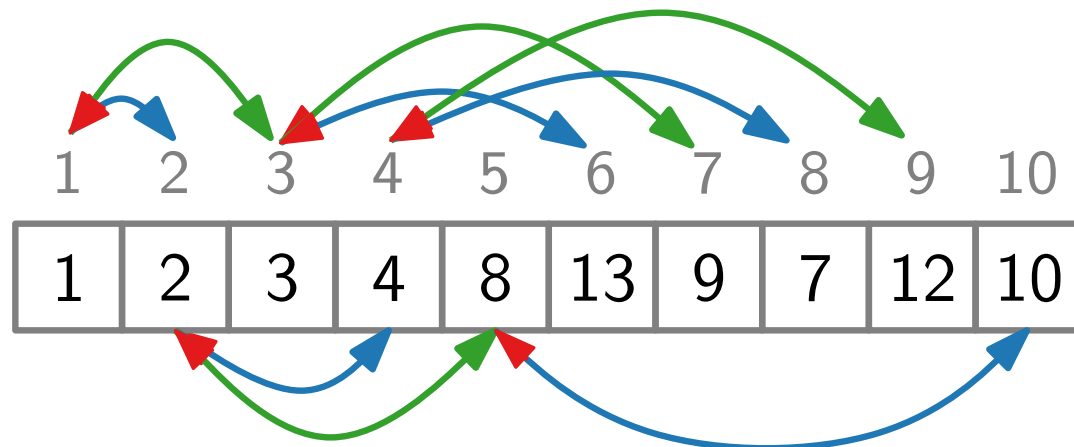
Definition.

Ein Heap hat die **Min-Heap-Eigenschaft**,

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

Min-Heaps

sehr schnelle Rechenoperationen!



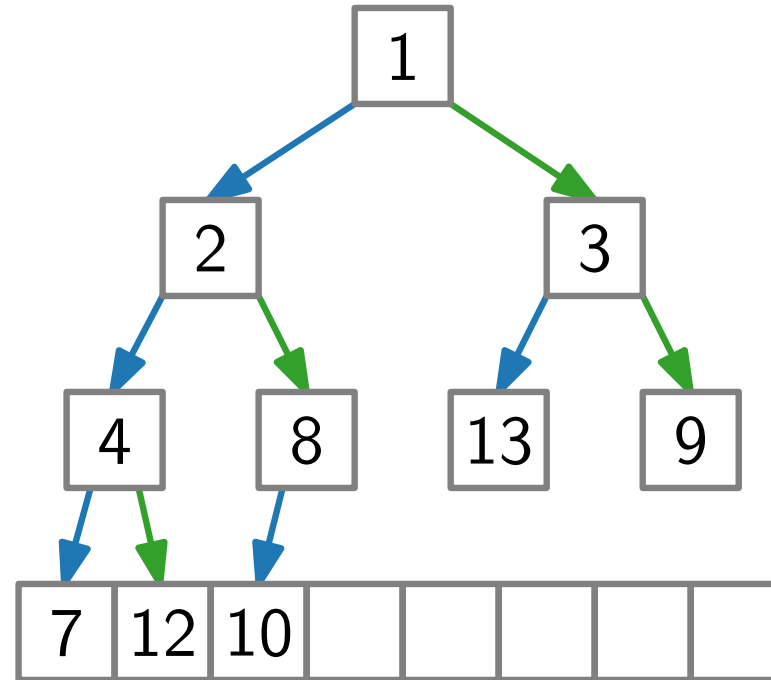
Pfeile implementieren:

$\text{LEFT}(\text{index } i)$ **return** $2i$
 $\text{RIGHT}(\text{index } i)$ **return** $2i + 1$
 $\text{PARENT}(\text{index } i)$ **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.



Definition.

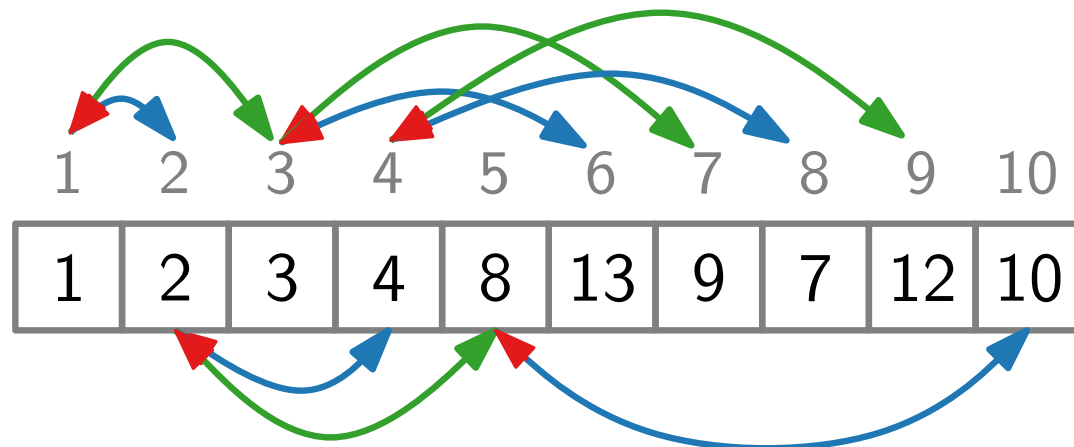
Ein Heap hat die **Min-Heap-Eigenschaft**,

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

So ein Heap heißt **Min-Heap**.

Min-Heaps

sehr schnelle Rechenoperationen!



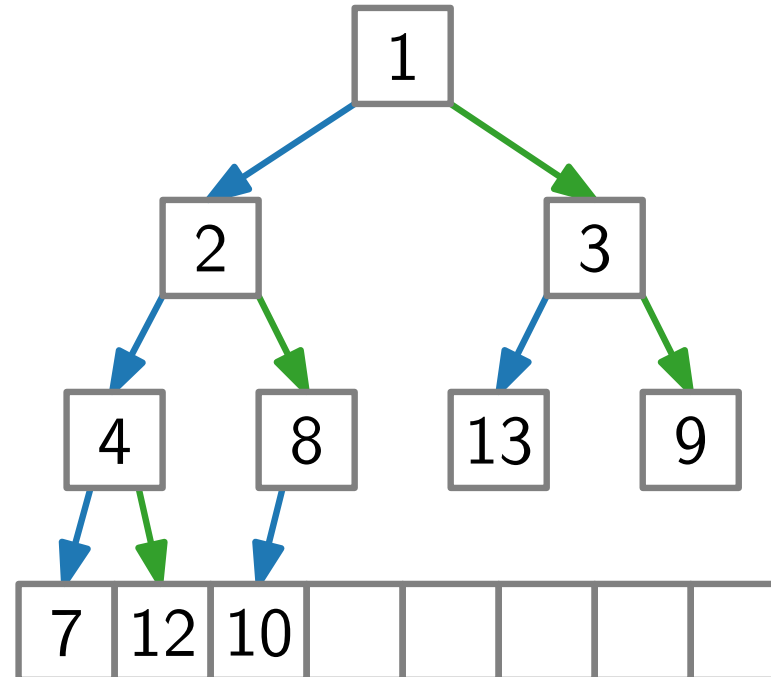
Pfeile implementieren:

$\text{LEFT}(\text{index } i)$ **return** $2i$
 $\text{RIGHT}(\text{index } i)$ **return** $2i + 1$
 $\text{PARENT}(\text{index } i)$ **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.



Definition.

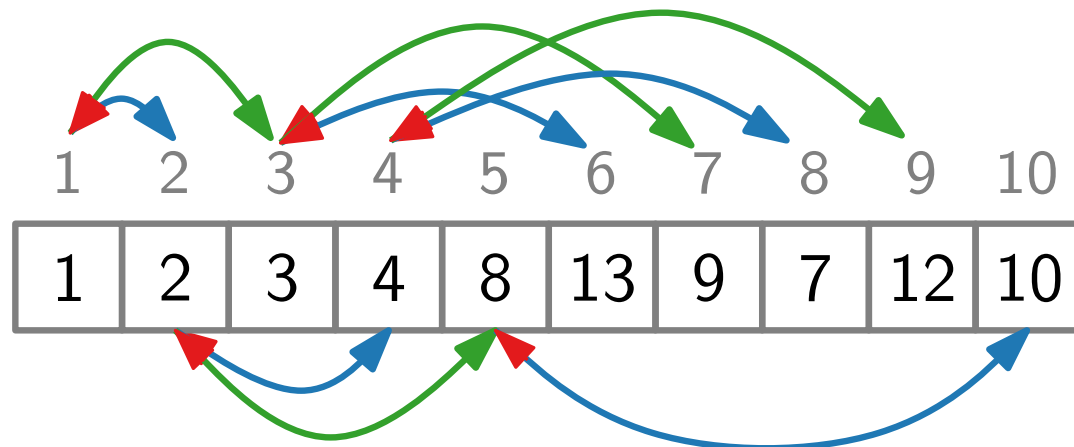
Ein Heap hat die ~~Min-Heap-Eigenschaft~~,

wenn für jeden Knoten $i > 1$ gilt: $A[\text{PARENT}(i)] \not\leq A[i]$.

So ein Heap heißt ~~Min-Heap~~.

Min-Heaps

sehr schnelle Rechenoperationen!



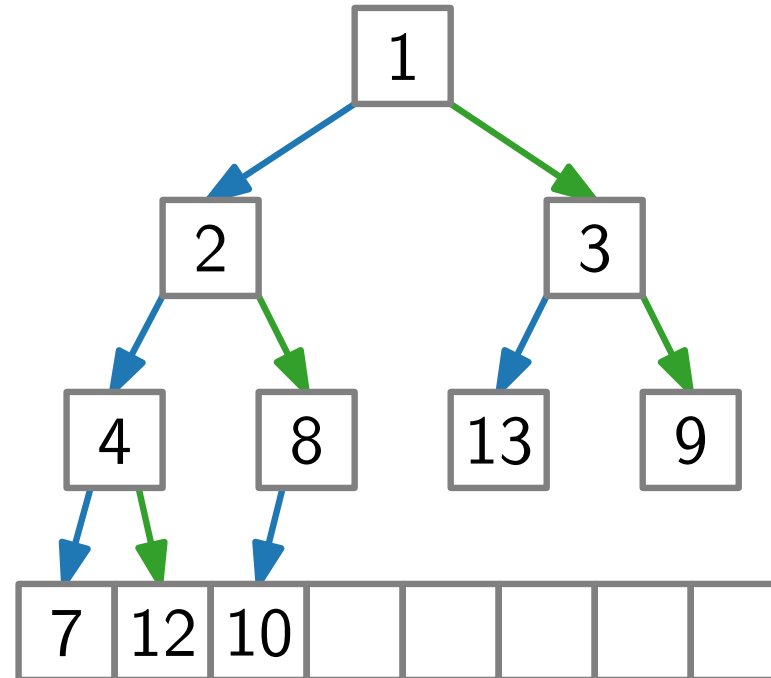
Pfeile implementieren:

$\text{LEFT}(\text{index } i)$ **return** $2i$
 $\text{RIGHT}(\text{index } i)$ **return** $2i + 1$
 $\text{PARENT}(\text{index } i)$ **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.



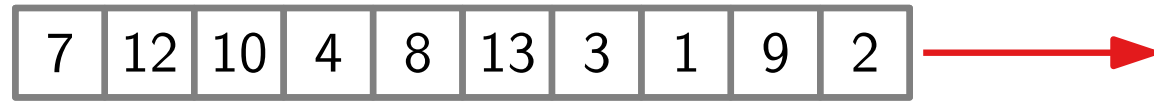
Definition.

Ein Heap hat die ~~Min-Heap-Eigenschaft~~, **Max**

wenn für jeden Knoten $i > 1$ gilt: $A[\text{PARENT}(i)] \not\leq A[i]$.

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

Baustelle



Baustelle



Baustelle

7	12	10	4	8	13	3	1	9	2
---	----	----	---	---	----	---	---	---	---

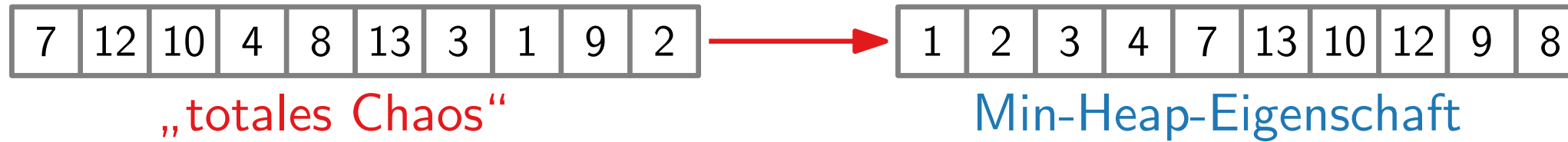
„totales Chaos“



1	2	3	4	7	13	10	12	9	8
---	---	---	---	---	----	----	----	---	---

Min-Heap-Eigenschaft

Baustelle



Aufgabe: Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Baustelle

7	12	10	4	8	13	3	1	9	2
---	----	----	---	---	----	---	---	---	---

„totales Chaos“

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Min-Heap-Eigenschaft

1	2	3	4	7	13	10	12	9	8
---	---	---	---	---	----	----	----	---	---

1	2	3	4	7	8	9	10	12	13
---	---	---	---	---	---	---	----	----	----

aufsteigende Sortierung

Baustelle

7	12	10	4	8	13	3	1	9	2
---	----	----	---	---	----	---	---	---	---

„totales Chaos“

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

1	2	3	4	7	13	10	12	9	8
---	---	---	---	---	----	----	----	---	---

Min-Heap-Eigenschaft

1	2	3	4	7	8	9	10	12	13
---	---	---	---	---	---	---	----	----	----

aufsteigende Sortierung

Baustelle

7	12	10	4	8	13	3	1	9	2
---	----	----	---	---	----	---	---	---	---

„totales Chaos“

1	2	3	4	7	13	10	12	9	8
---	---	---	---	---	----	----	----	---	---

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

1	2	3	4	7	8	9	10	12	13
---	---	---	---	---	---	---	----	----	----

aufsteigende Sortierung

Fertig?

Baustelle



„totales Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Baustelle

7	12	10	4	8	13	3	1	9	2
---	----	----	---	---	----	---	---	---	---

„totales Chaos“

1	2	3	4	7	13	10	12	9	8
---	---	---	---	---	----	----	----	---	---

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

1	2	3	4	7	8	9	10	12	13
---	---	---	---	---	---	---	----	----	----

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

Baustelle

7	12	10	4	8	13	3	1	9	2
---	----	----	---	---	----	---	---	---	---

„totales Chaos“

1	2	3	4	7	13	10	12	9	8
---	---	---	---	---	----	----	----	---	---

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

1	2	3	4	7	8	9	10	12	13
---	---	---	---	---	---	---	----	----	----

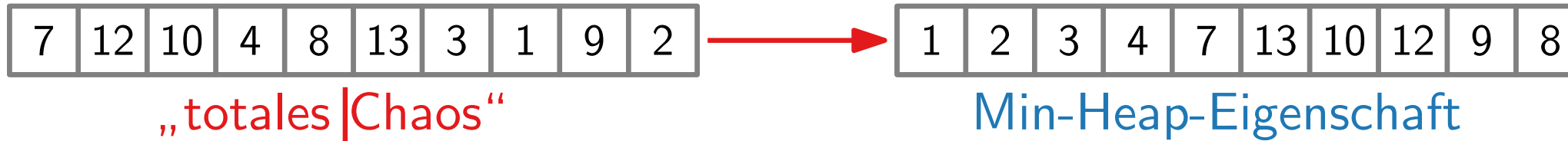
aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!

Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

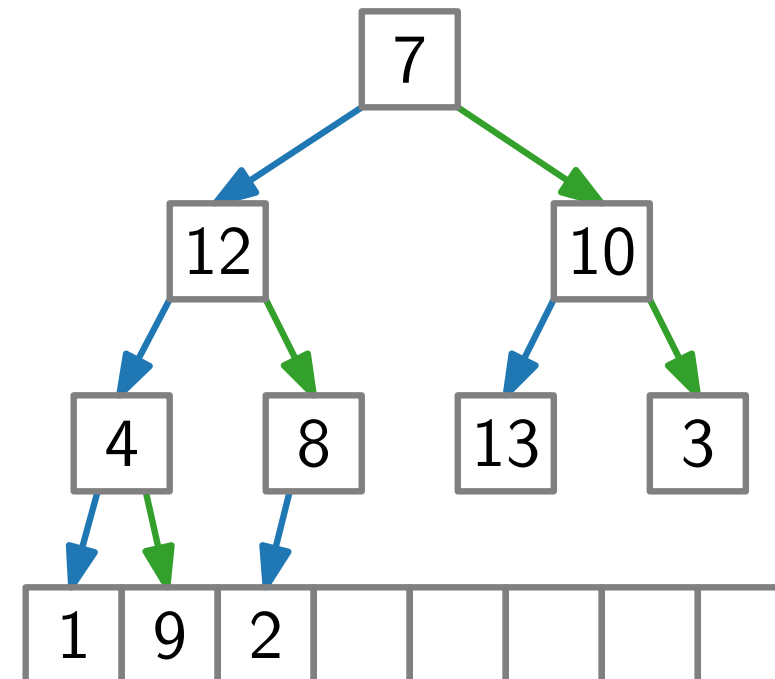
Nimm MERGESORT!



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

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

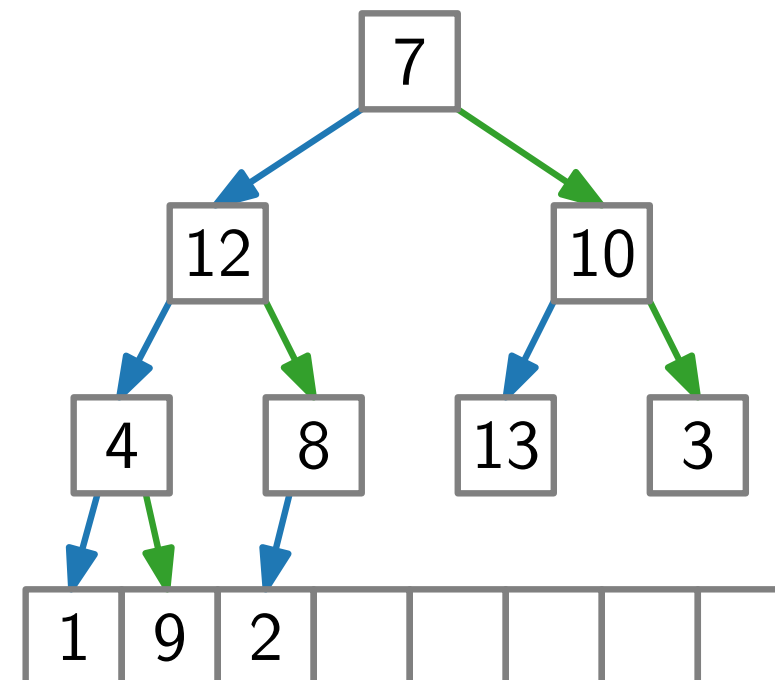
Schnellere Berechnung!

Idee:

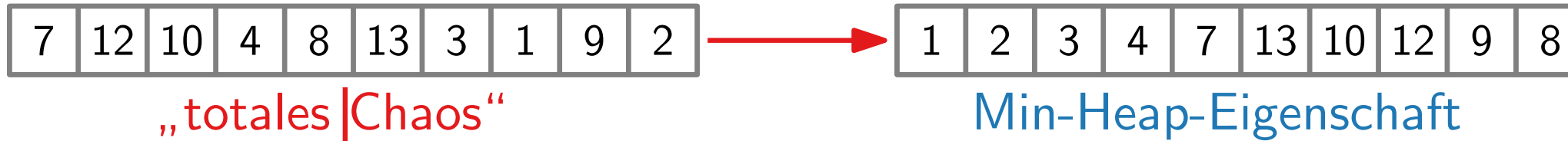
Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

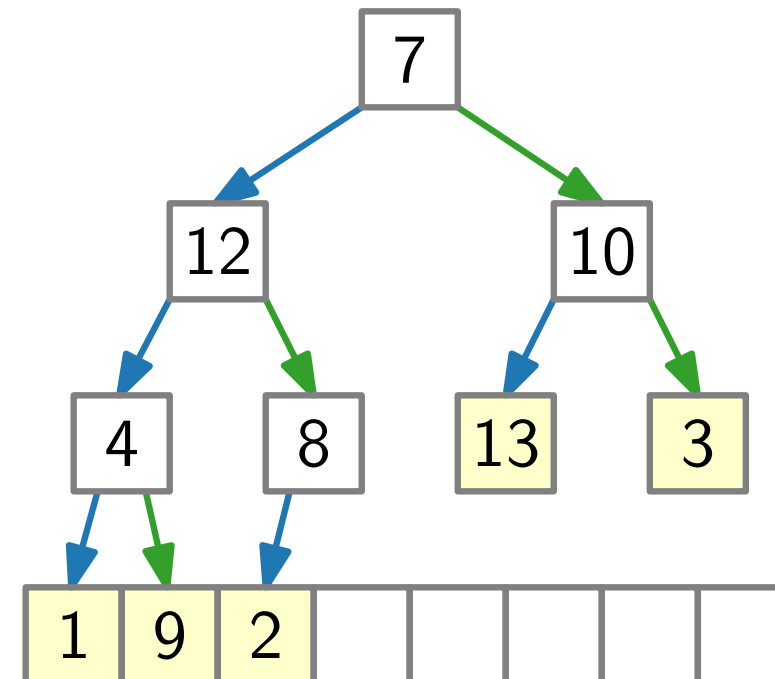
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

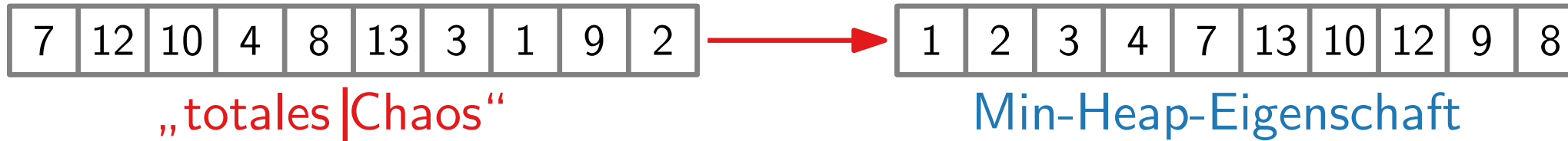
Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

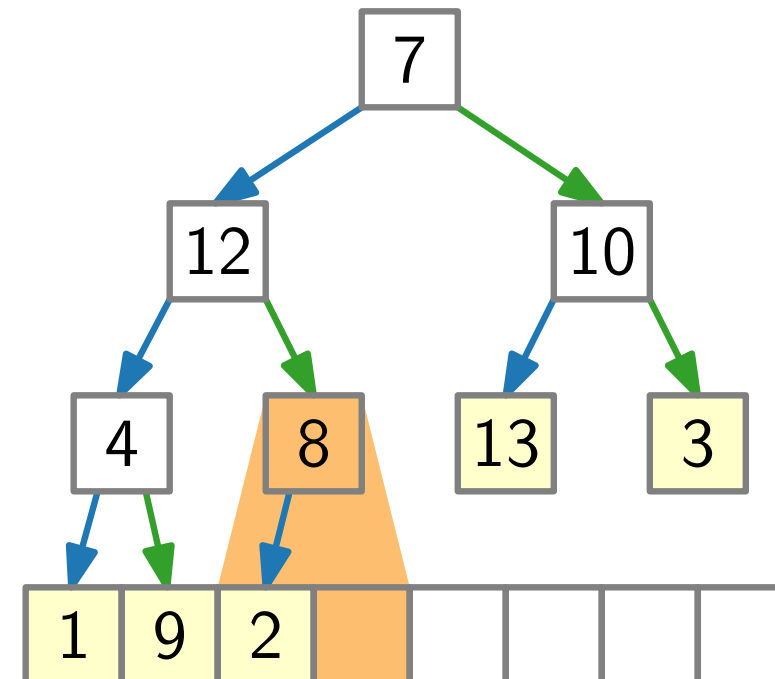
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

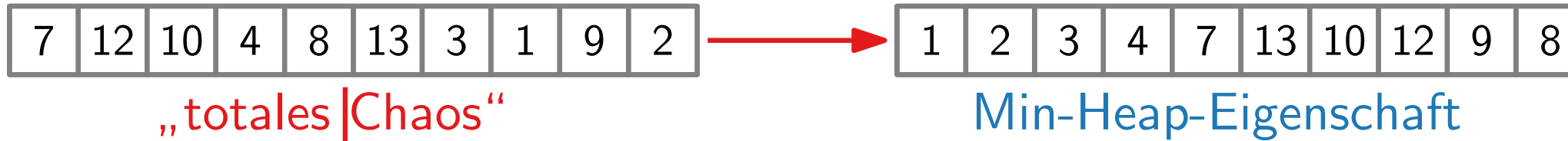
Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

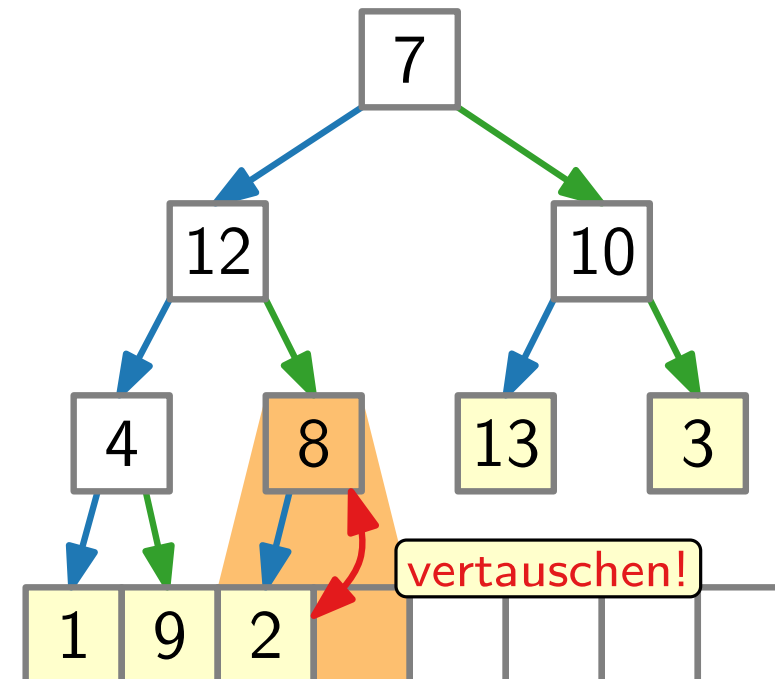
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

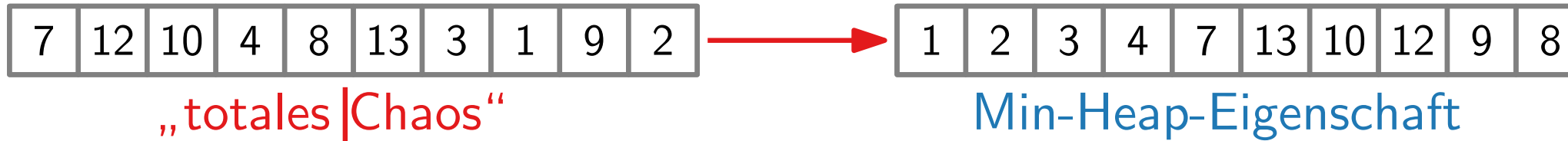
Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

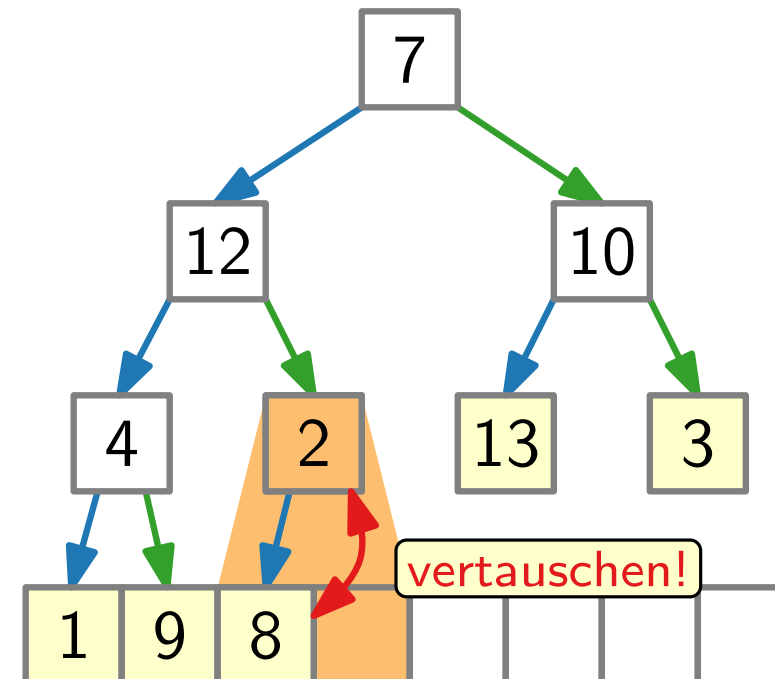
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

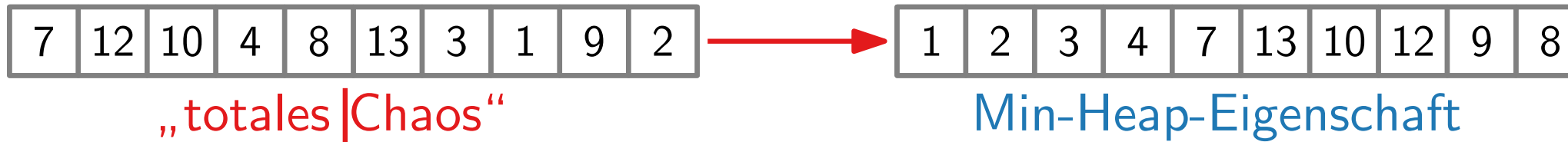
Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

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

Hoffen:

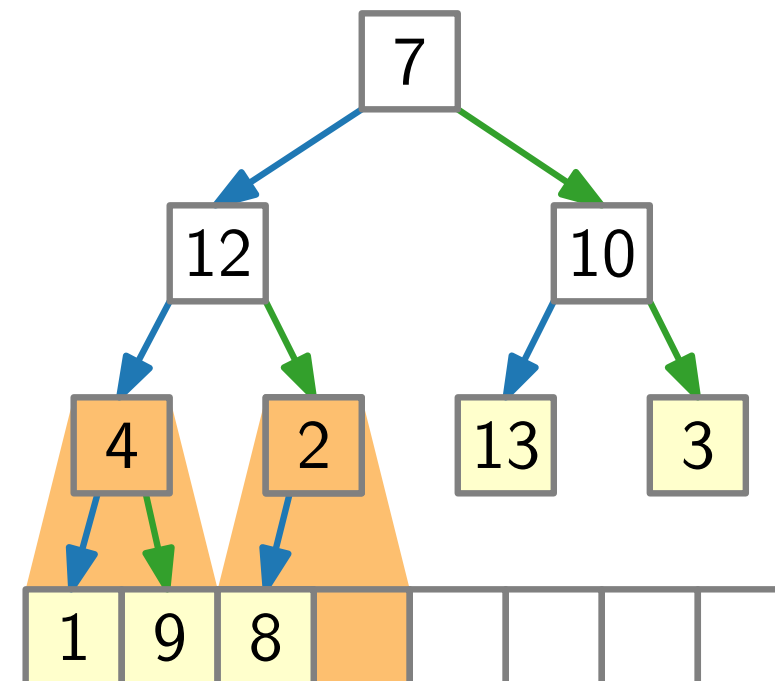
Schnellere Berechnung!

Idee:

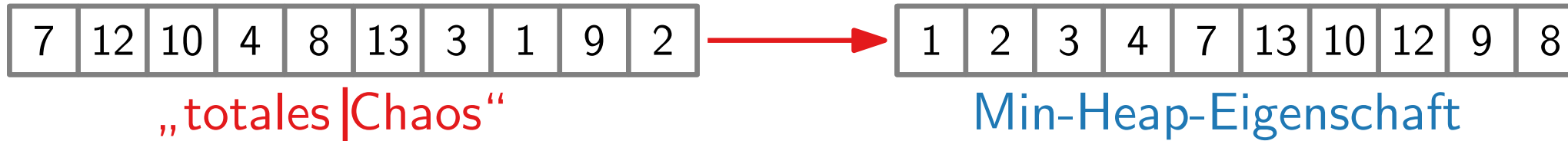
Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

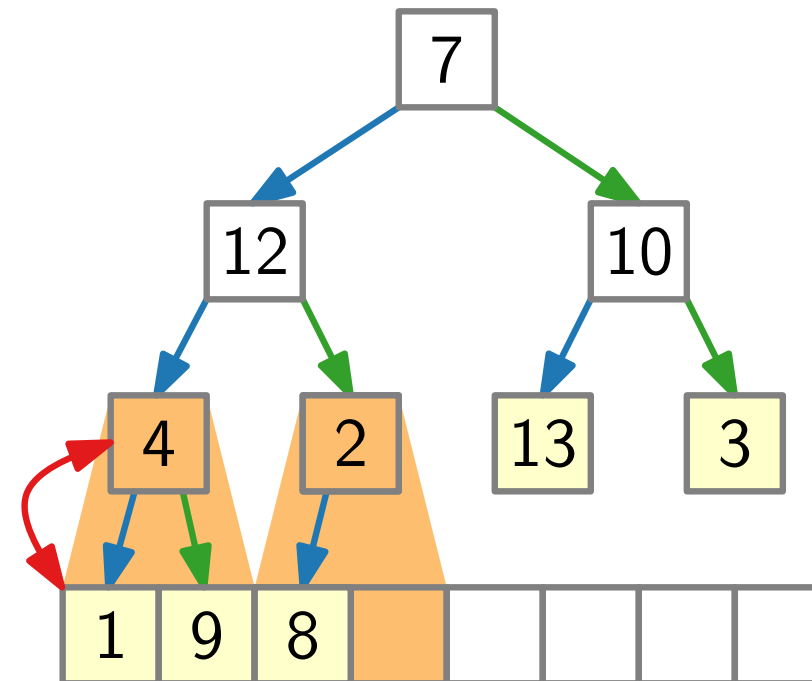
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

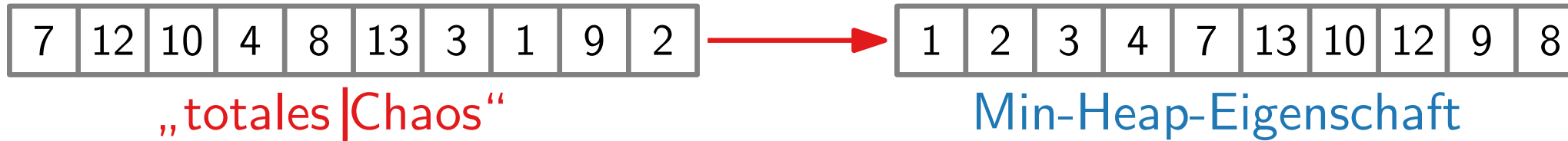
Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

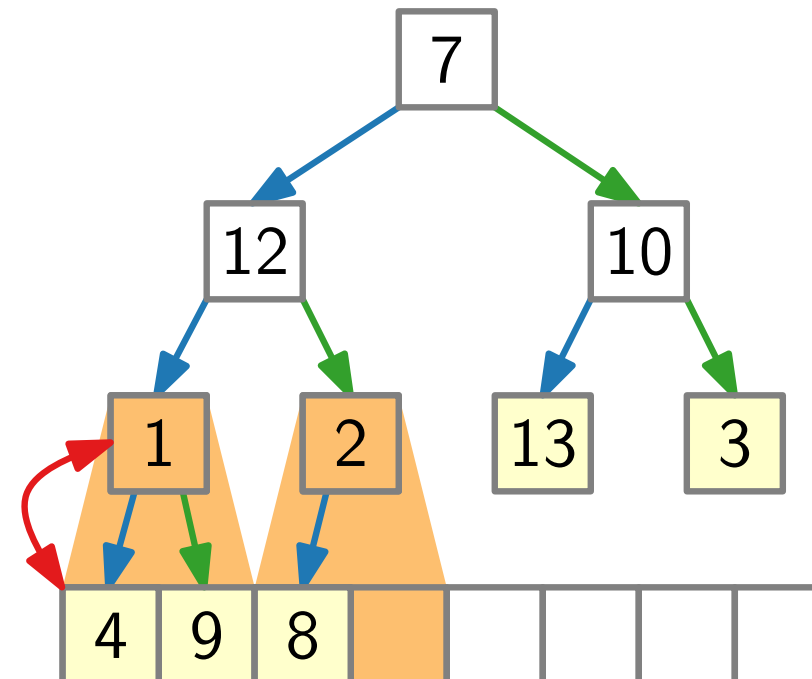
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

Schnellere Berechnung!

Idee:

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



Baustelle



„**total**es Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

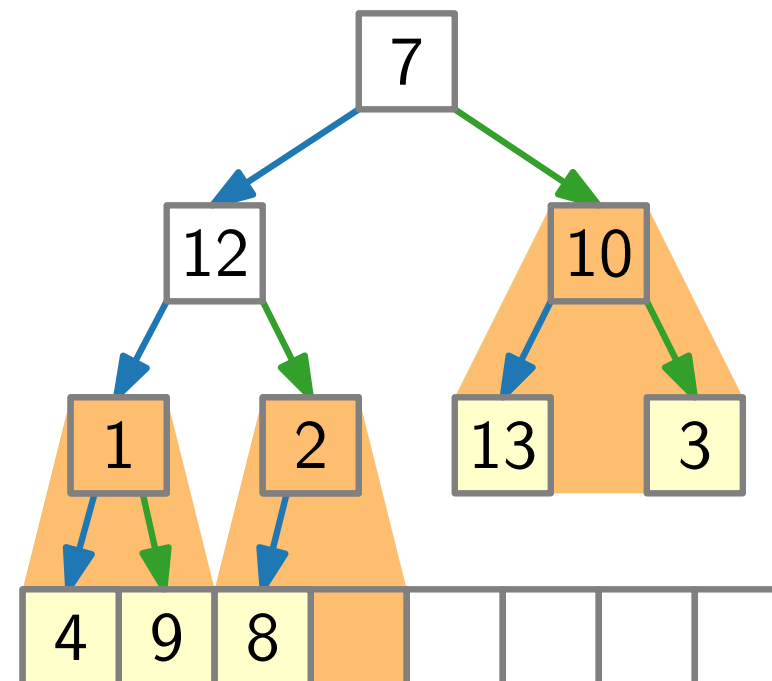
Schnellere Berechnung!

Idee:

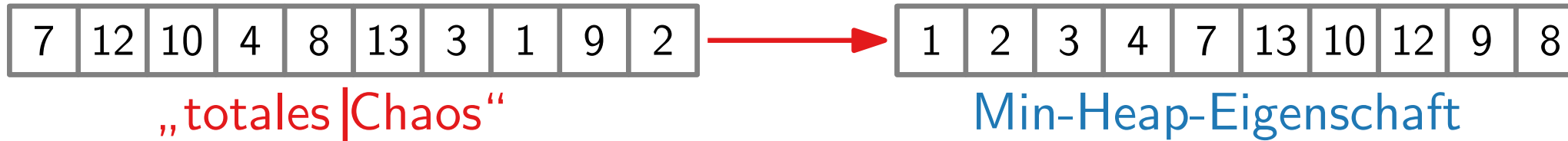
Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

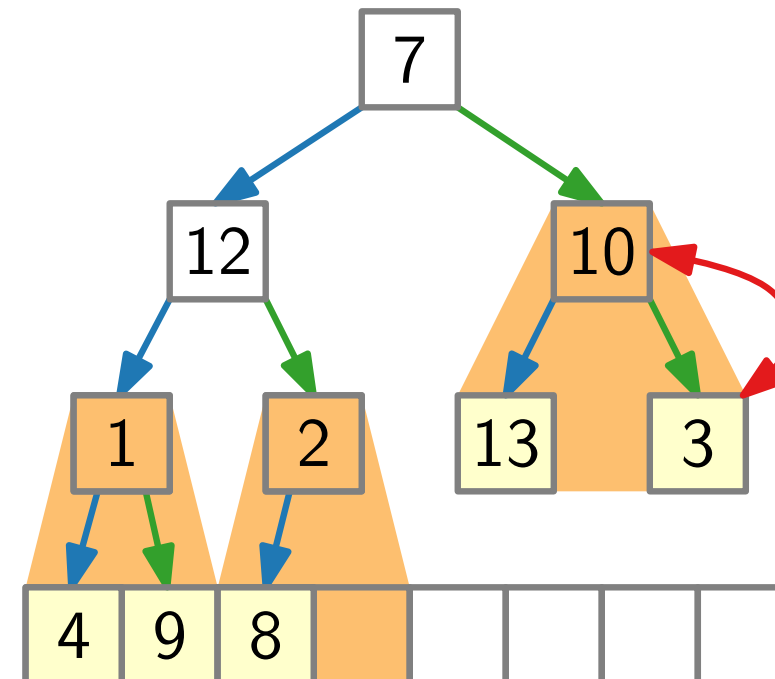
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

Schnellere Berechnung!

Idee:

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



Baustelle

7 12 10 4 8 13 3 1 9 2

„totales Chaos“

1 2 3 4 7 13 10 12 9 8

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

1 2 3 4 7 8 9 10 12 13

aufsteigende Sortierung

Fertig?

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

Hoffen:

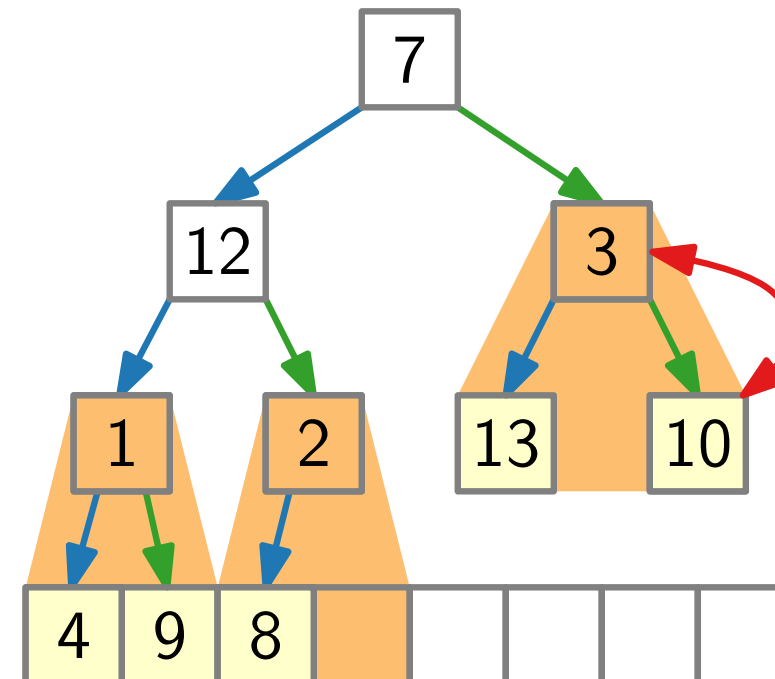
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



„**total**es Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

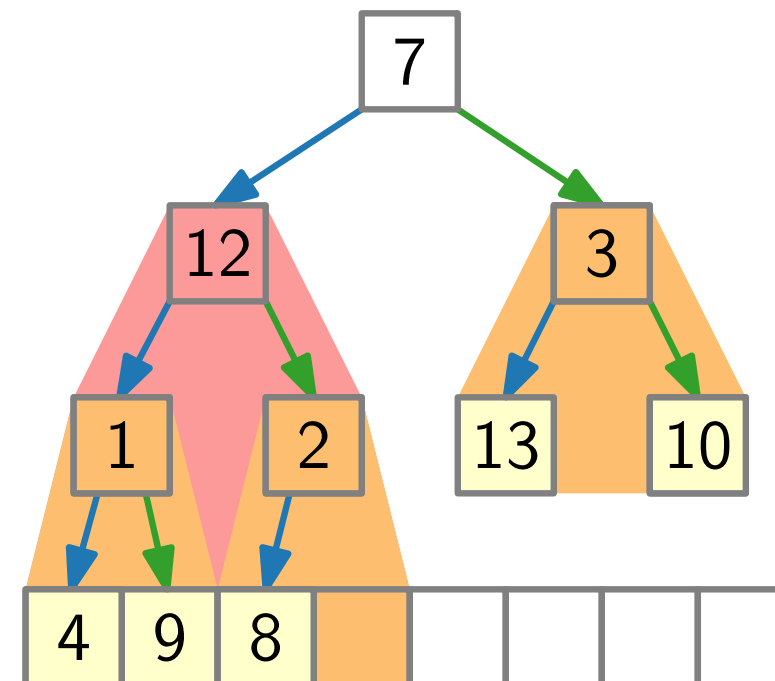
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

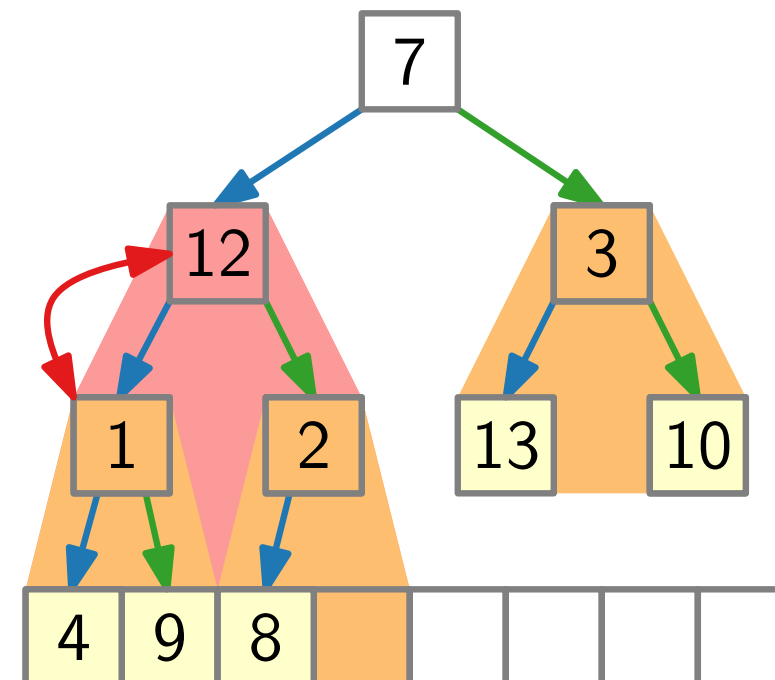
Schnellere Berechnung!

Idee:

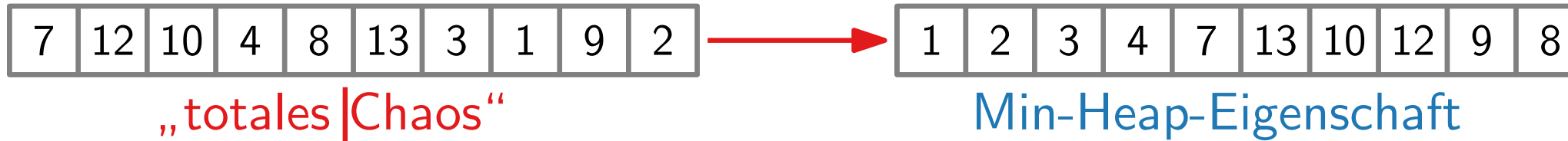
Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

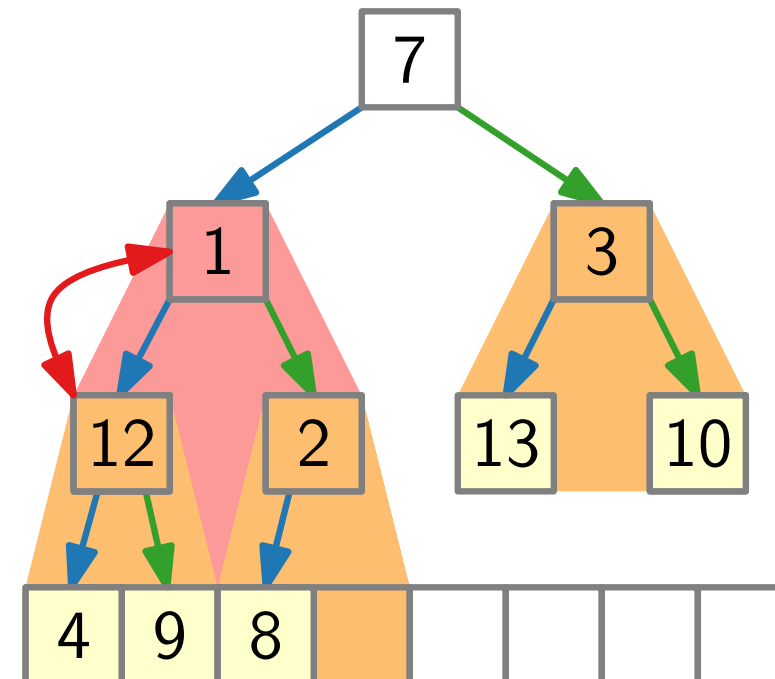
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

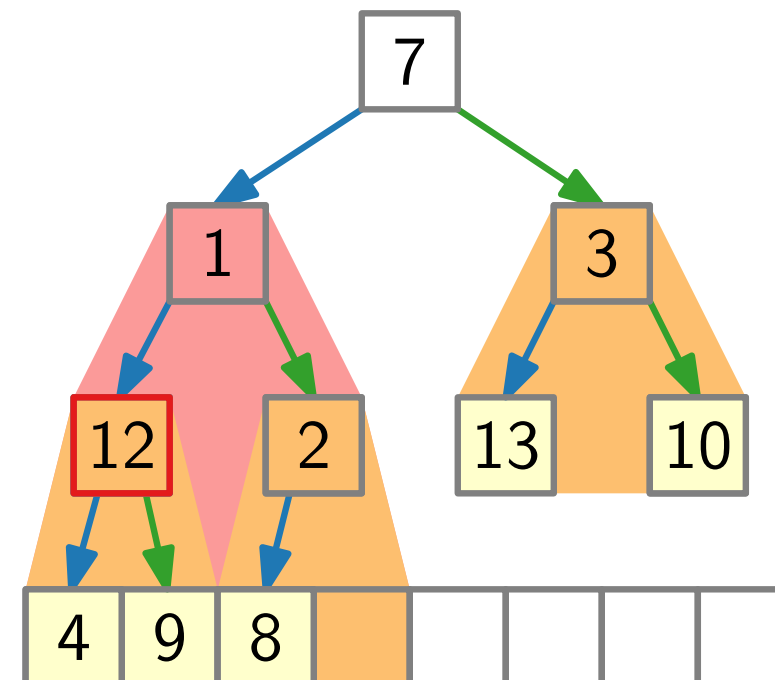
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



„**totales Chaos**“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

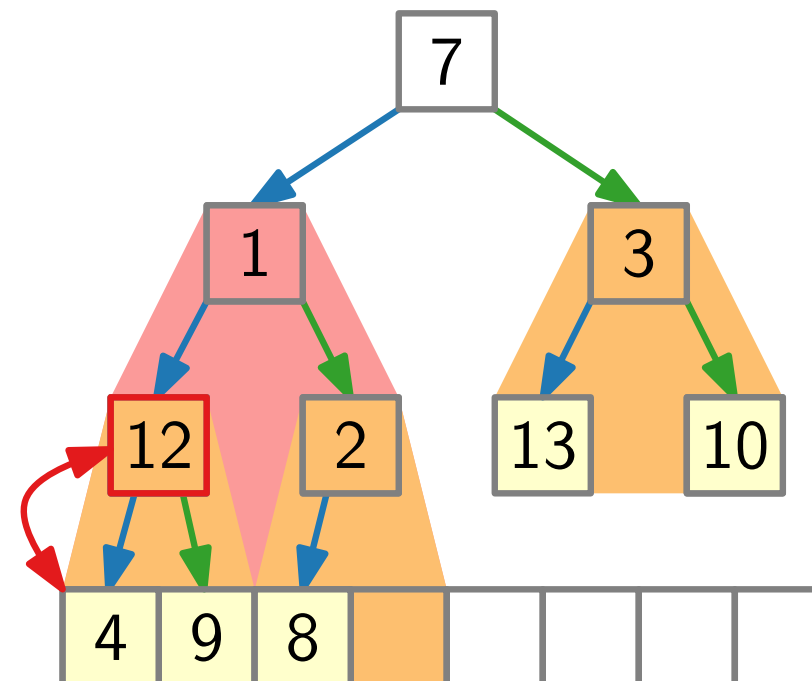
Schnellere Berechnung!

Idee:

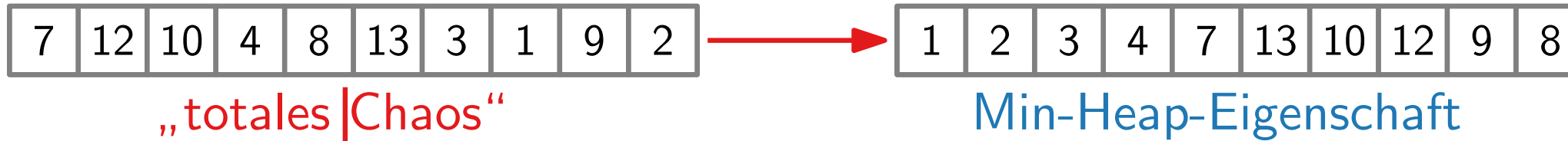
Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

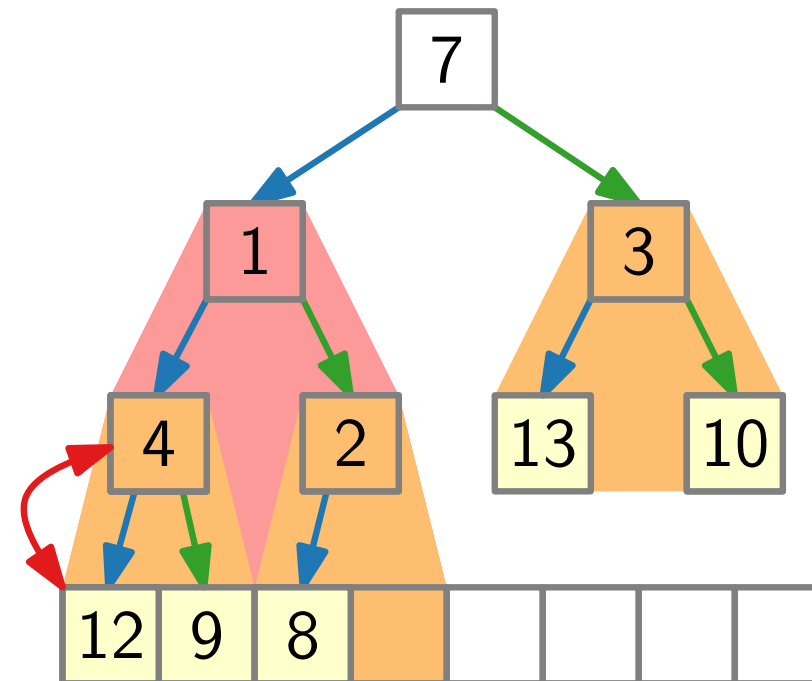
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

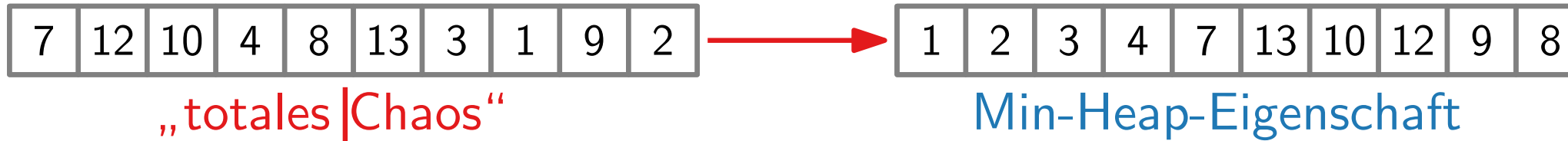
Schnellere Berechnung!

Idee:

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



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



Fertig?

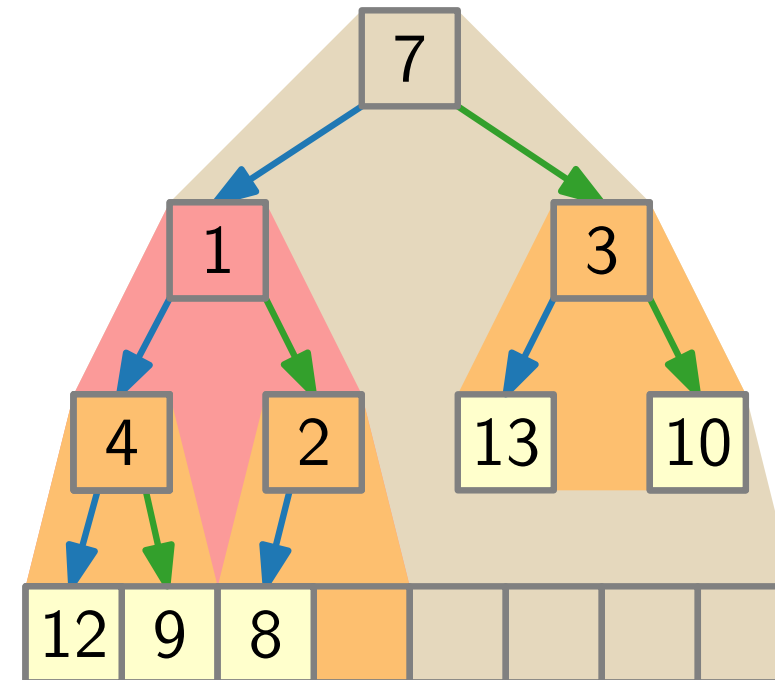
Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen:

Schnellere Berechnung!

Idee:

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



Baustelle



„**total**es Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

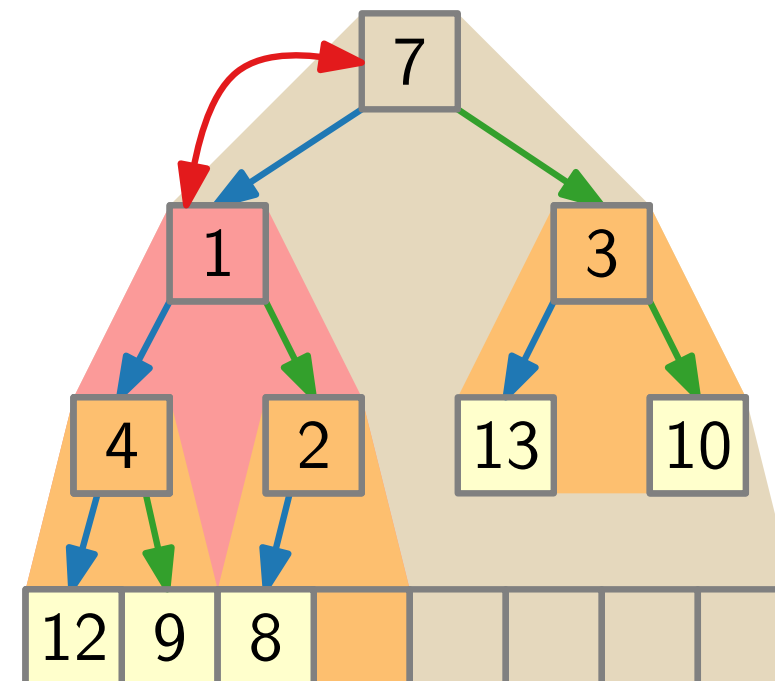
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



„**total**es Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

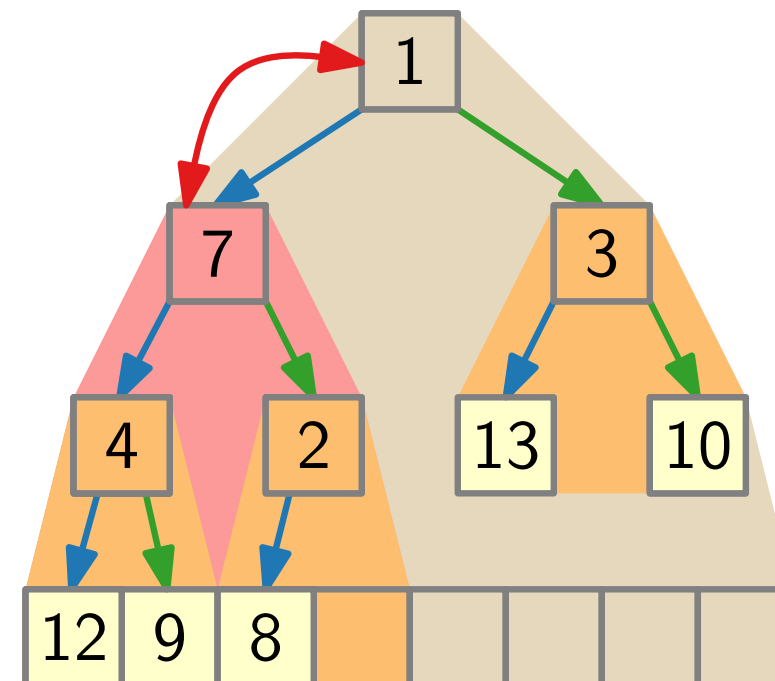
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



„**total**es Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

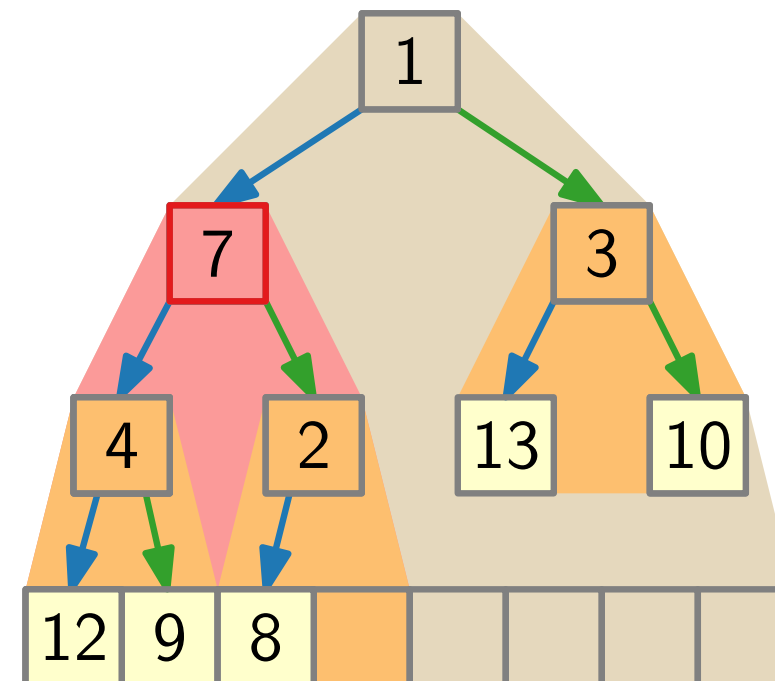
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



„totales Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

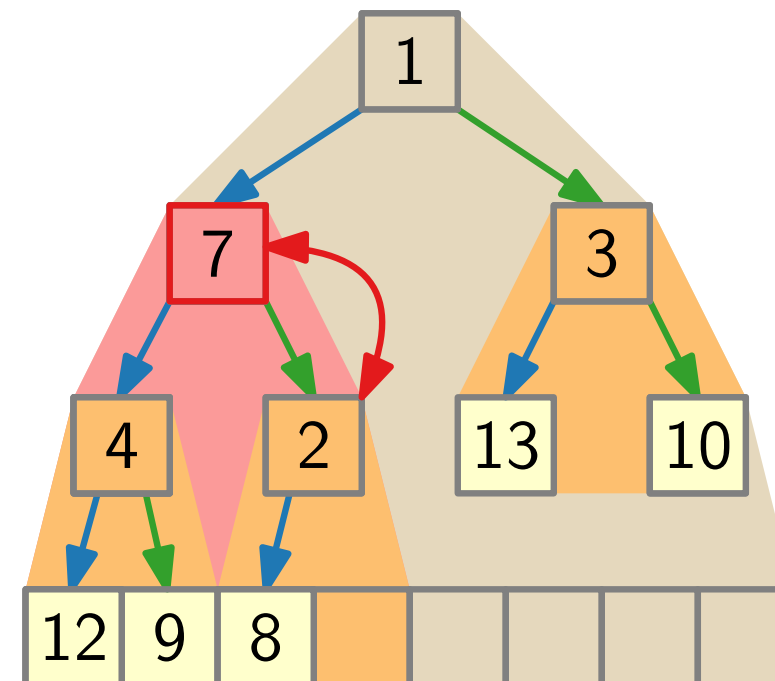
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



„**total**es Chaos“

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

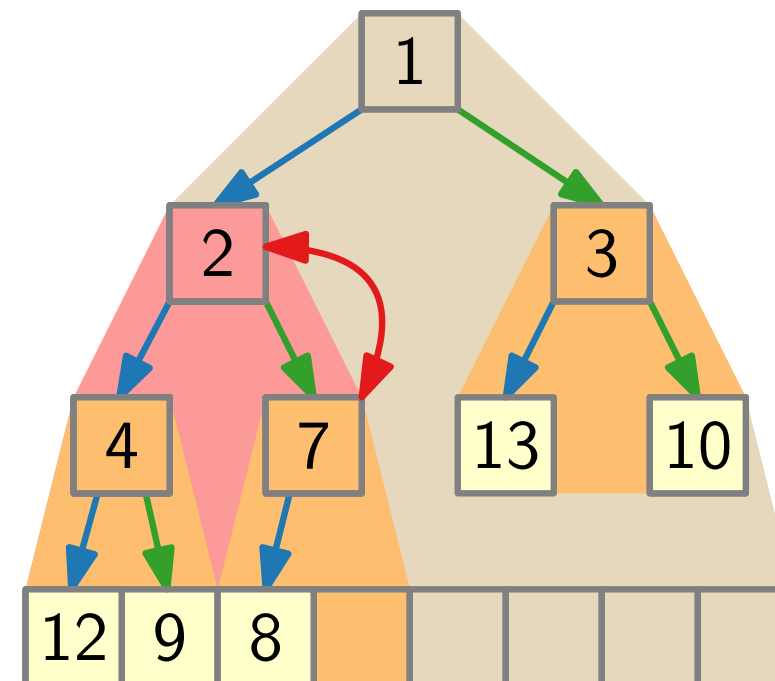
Schnellere Berechnung!

Idee:

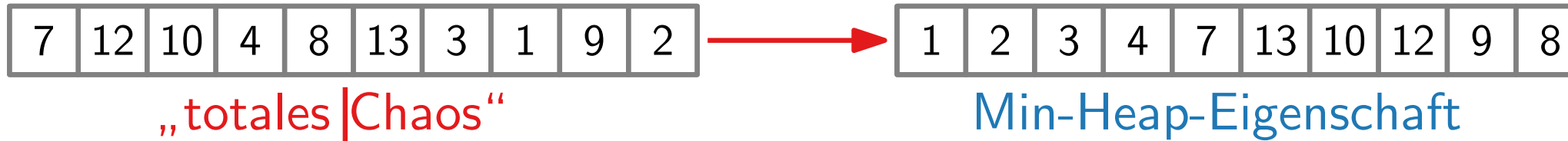
Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle



Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!



aufsteigende Sortierung

Fertig?

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

Hoffen:

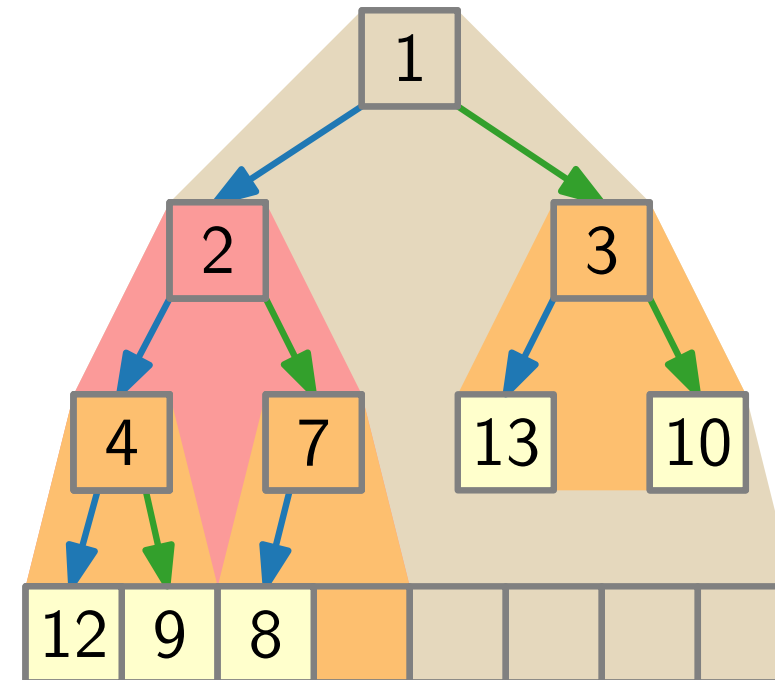
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Fertig!

Baustelle

7 12 10 4 8 13 3 1 9 2

„totales Chaos“

1 2 3 4 7 13 10 12 9 8

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

– Ergebnis –

1 2 3 4 7 8 9 10 12 13

aufsteigende Sortierung

Fertig?

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

Hoffen:

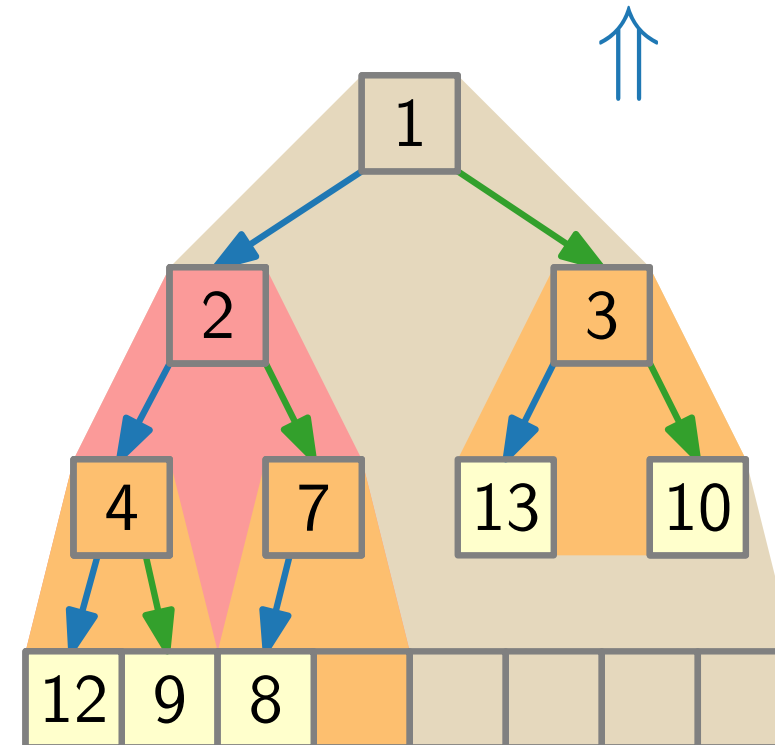
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Fertig!

Baustelle

7 12 10 4 8 13 3 1 9 2

„**totales Chaos**“

1 2 3 4 7 13 10 12 9 8

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

– Ergebnis –

1 2 3 4 7 8 9 10 12 13

aufsteigende Sortierung

Fertig?

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

Hoffen:

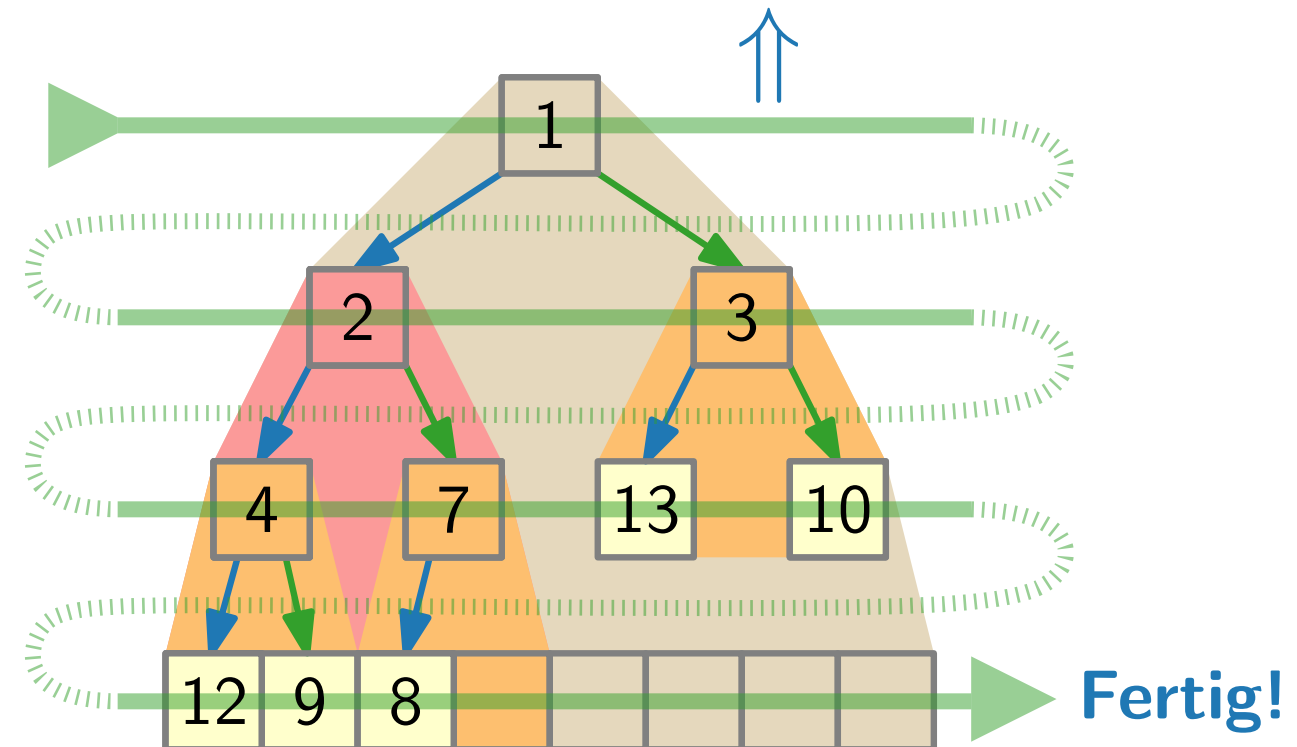
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

Arbeite **bottom-up**:

Erst die Blätter...



Baustelle

7 12 10 4 8 13 3 1 9 2

„totales Chaos“

1 2 3 4 7 13 10 12 9 8

Min-Heap-Eigenschaft

Aufgabe:

Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MERGESORT!

1 2 3 4 7 13 10 12 9 8

– Ergebnis –

1 2 3 4 7 8 9 10 12 13

aufsteigende Sortierung

Fertig?

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

Hoffen:

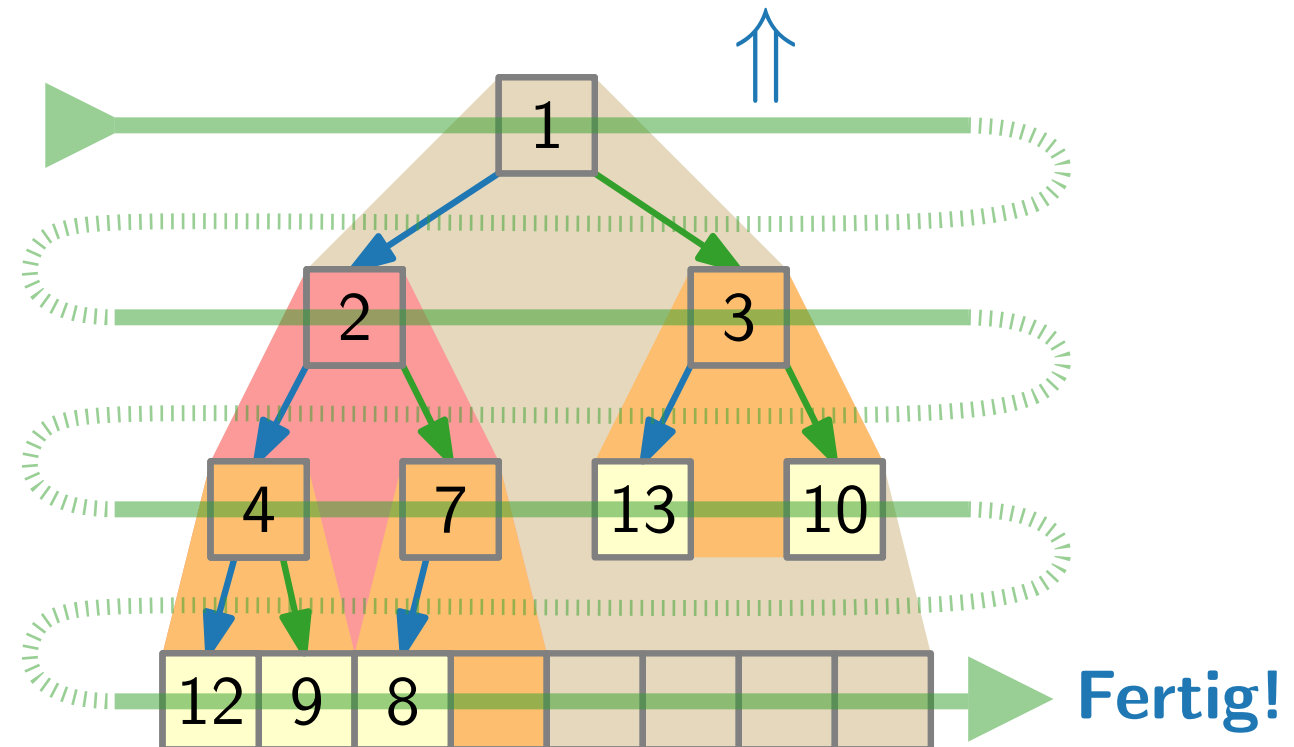
Schnellere Berechnung!

Idee:

Nutze Baumstruktur!

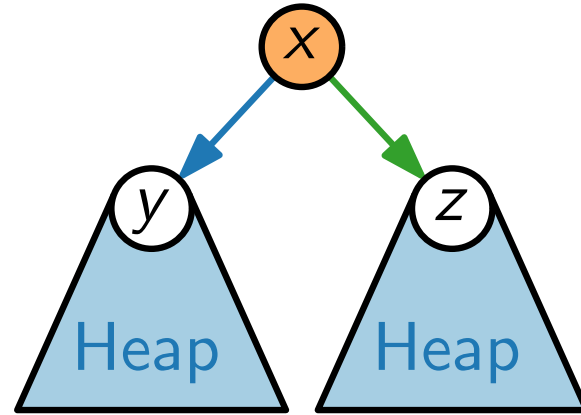
Arbeite **bottom-up**:

Erst die Blätter...



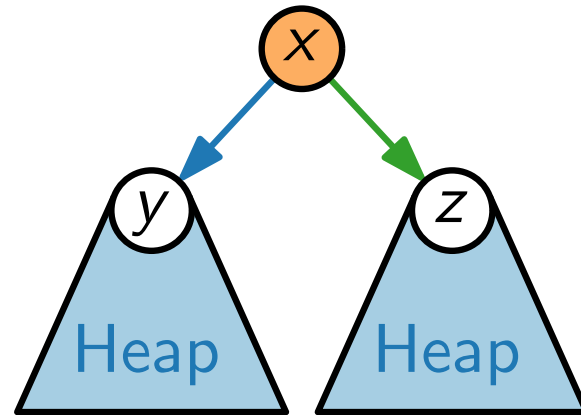
Elementaroperation

„**Versickere**“ x , falls x zu groß



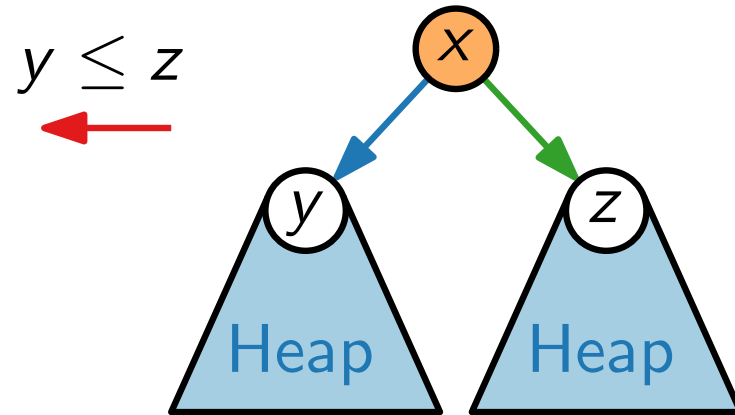
Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



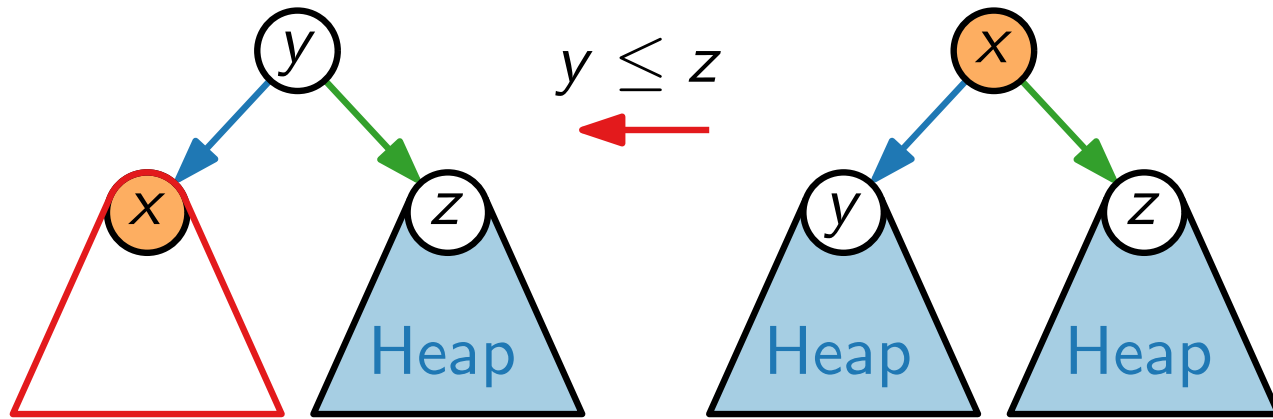
Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



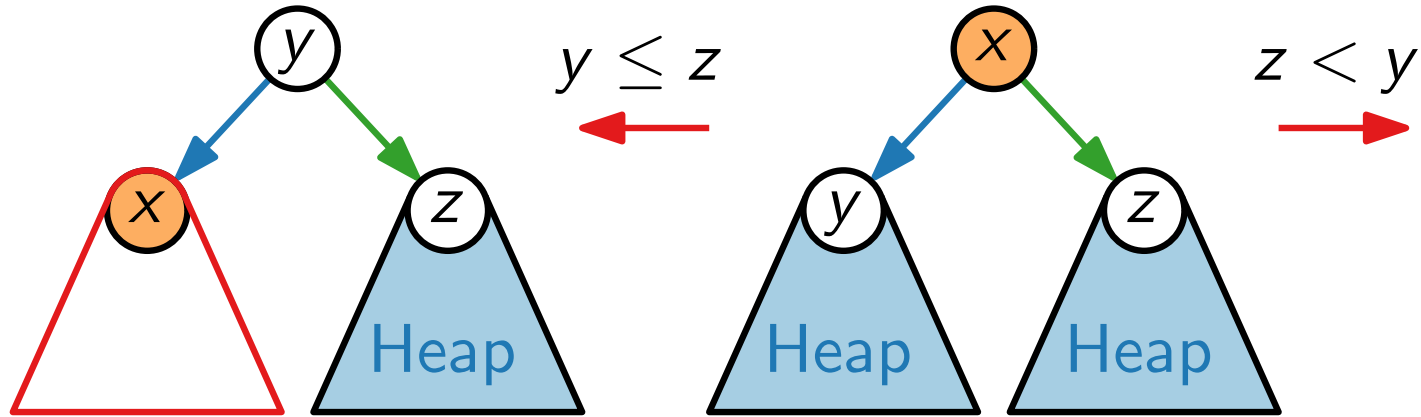
Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



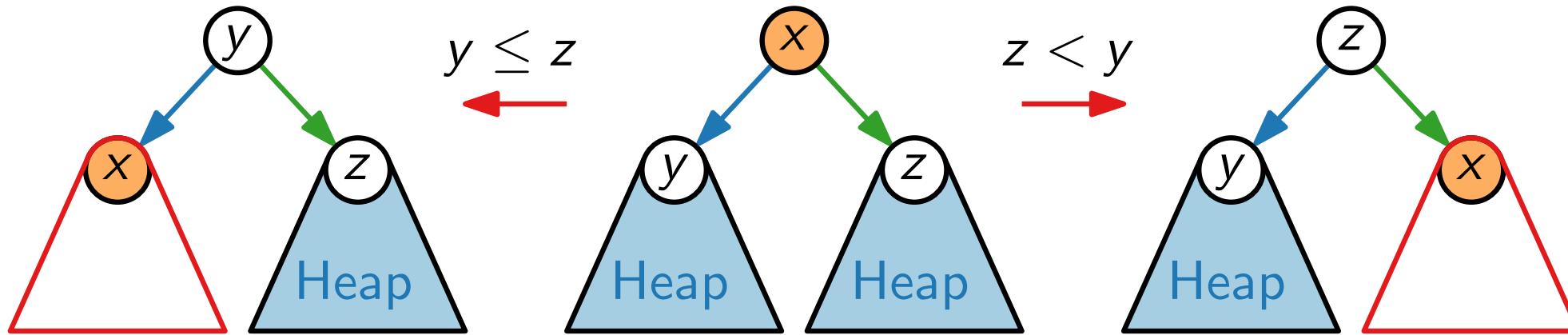
Elementaroperation

„Versickere“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



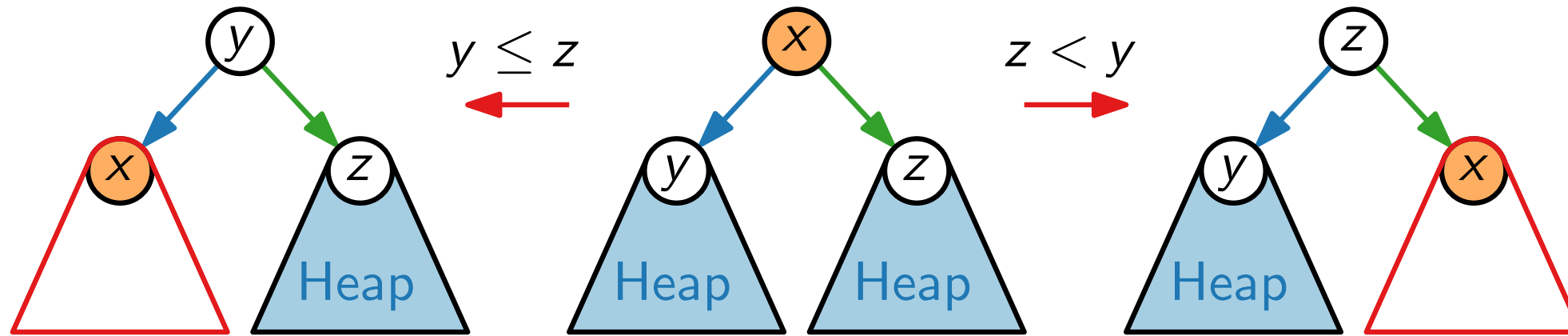
Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



Elementaroperation

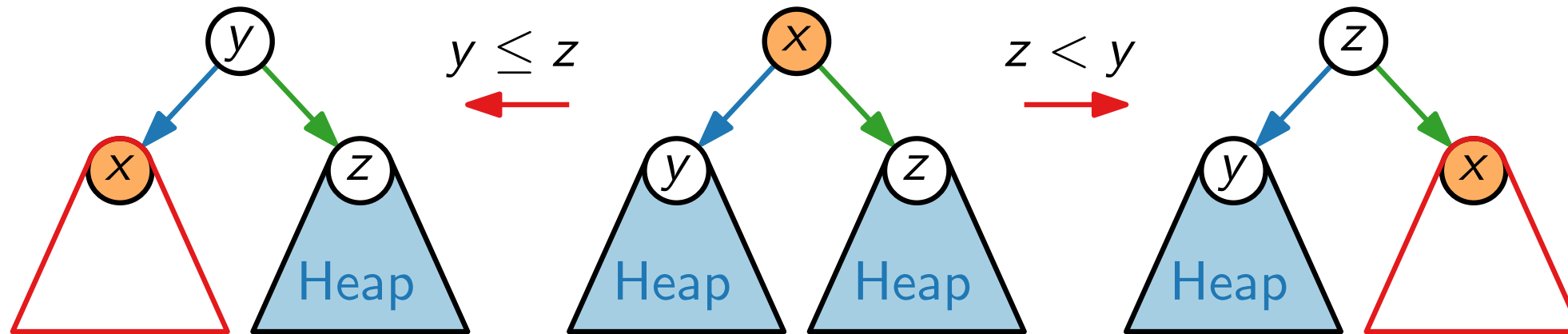
„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



MINHEAPIFY(int[] A , index i)

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$

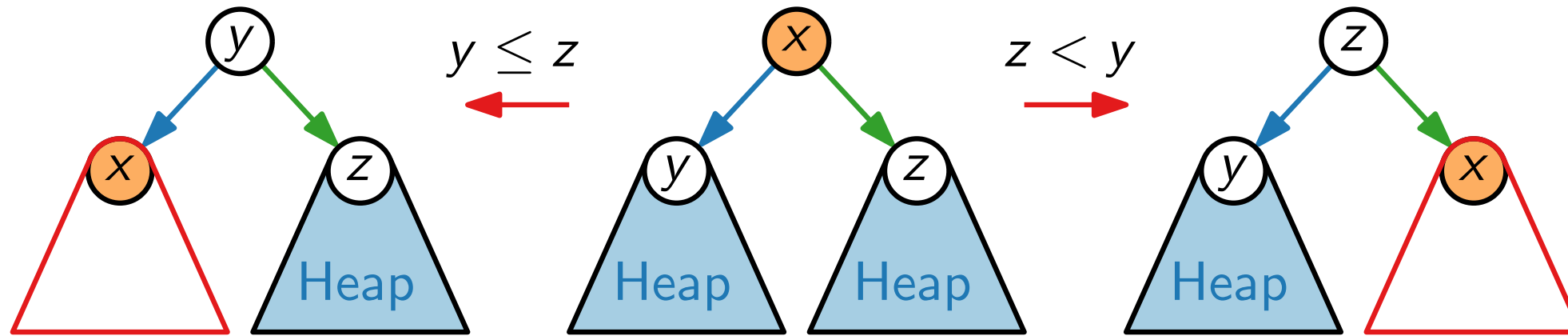


```
MINHEAPIFY(int[] A, index i)
```

```
   $\ell = \text{LEFT}(i); r = \text{RIGHT}(i)$ 
```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



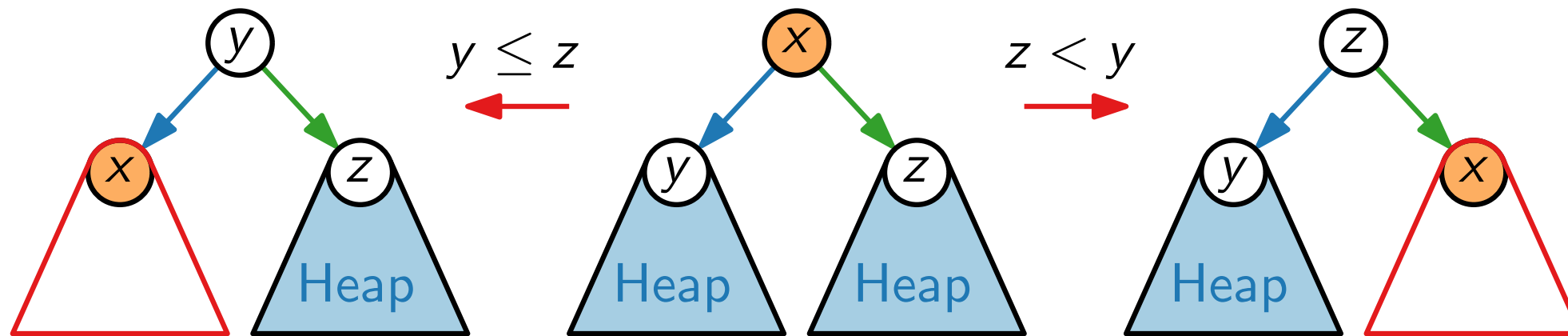
```
MINHEAPIFY(int[] A, index  $i$ )
```

```
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
```

```
   $\text{min} = i$ 
```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```
MINHEAPIFY(int[] A, index  $i$ )
```

```
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
```

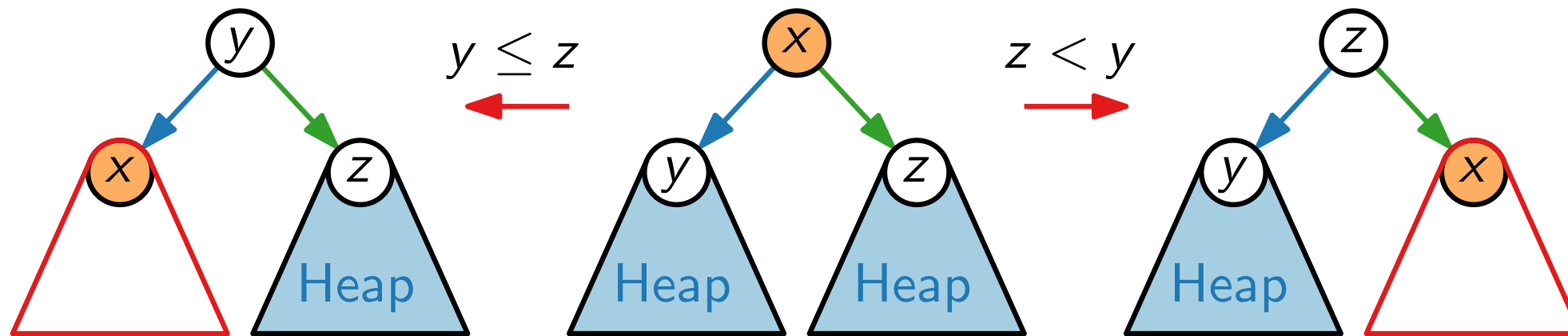
```
   $\text{min} = i$ 
```

```
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
```

```
     $\text{min} = \ell$ 
```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```
MINHEAPIFY(int[] A, index  $i$ )
```

```
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
```

```
   $\text{min} = i$ 
```

```
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
```

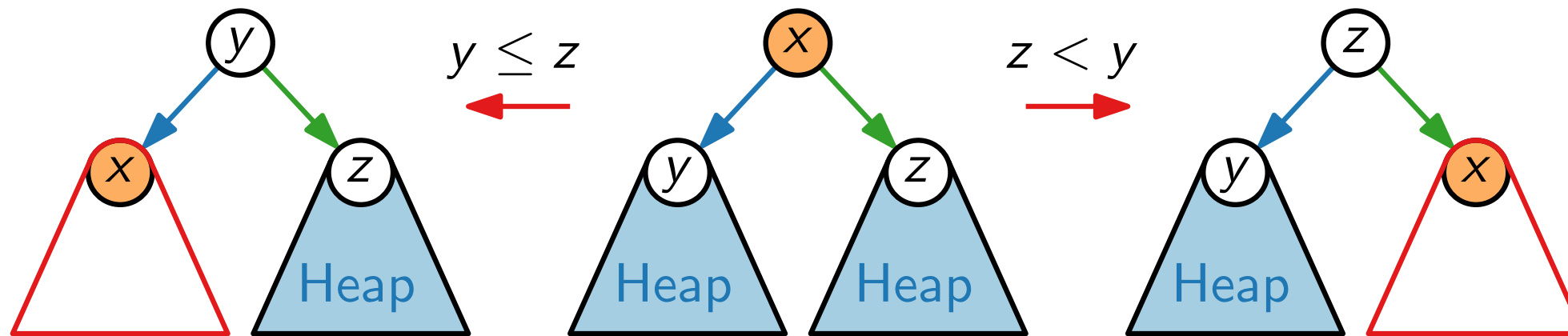
```
     $\text{min} = \ell$ 
```

```
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
```

```
     $\text{min} = r$ 
```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



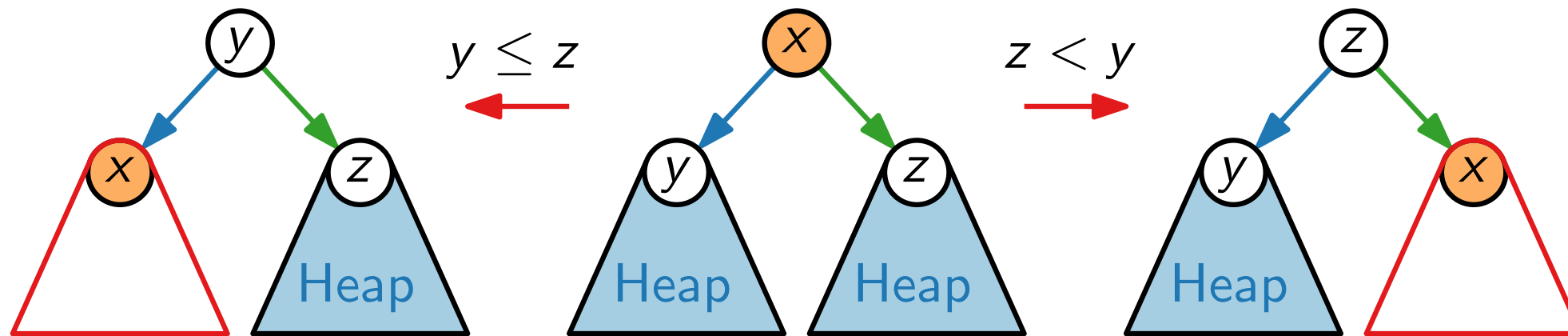
```

MINHEAPIFY(int[] A, index  $i$ )
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
    |

```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



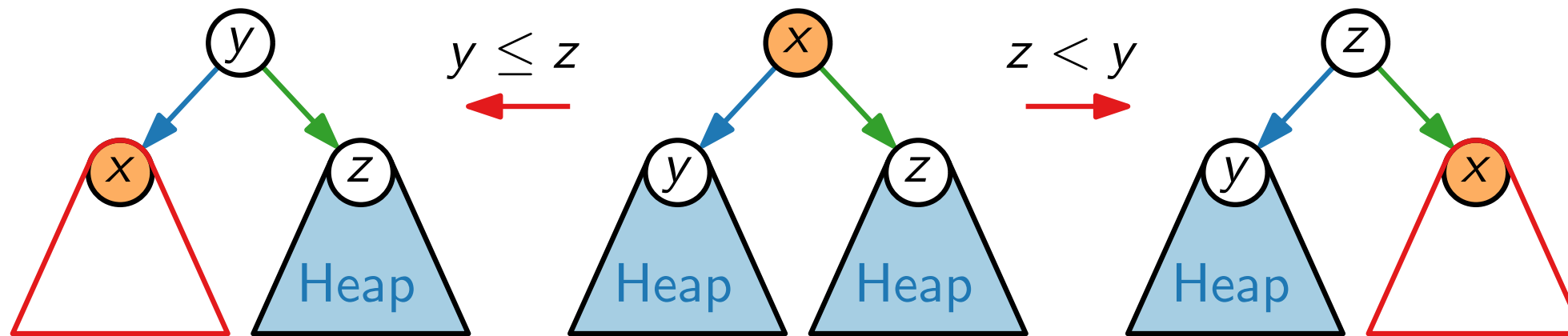
```

MINHEAPIFY(int[] A, index i)
   $\ell = \text{LEFT}(i); r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
     $A[i] \leftrightarrow A[\text{min}]$ 

```

Elementaroperation

„Versickere“ x , falls x zu groß, d.h. falls $x > \min(y, z)$

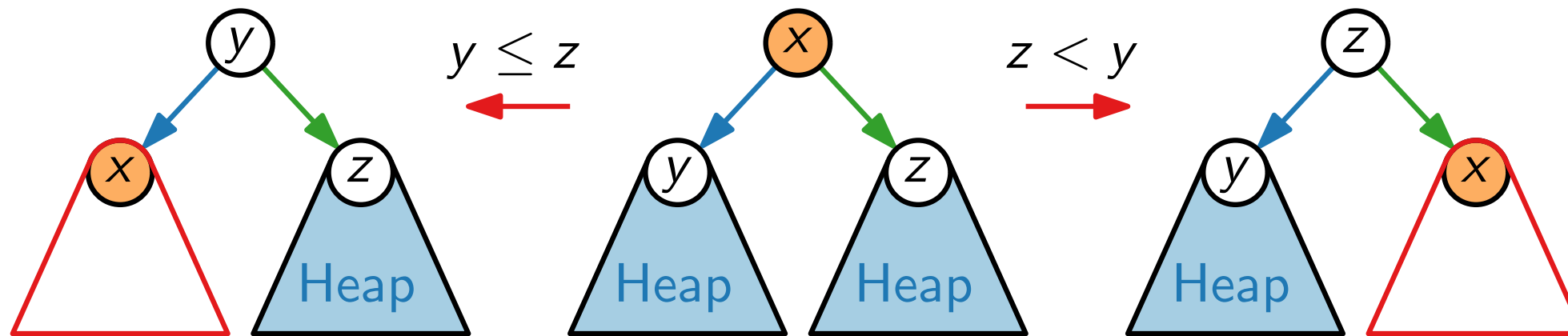


```

MINHEAPIFY(int[] A, index i)
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$ 
    Tausche  $A[i]$  und  $A[\text{min}]$ :
     $\text{temp} = A[i]$ 
     $A[i] = A[\text{min}]$ 
     $A[\text{min}] = \text{temp}$ 
     $A[i] \leftrightarrow A[\text{min}]$ 
  
```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$

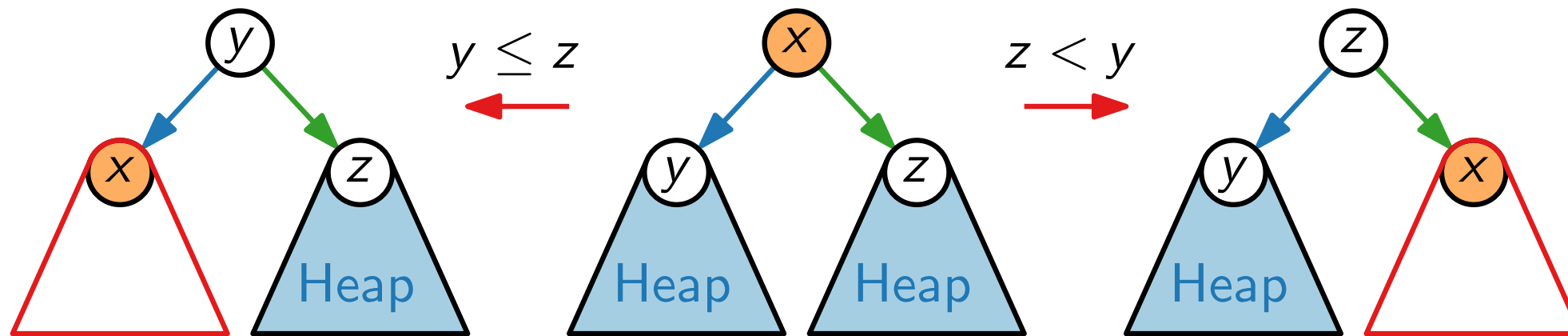


```

MINHEAPIFY(int[] A, index  $i$ )
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
     $A[i] \leftrightarrow A[\text{min}]$ 
    MINHEAPIFY( $A, \text{min}$ )
  
```

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



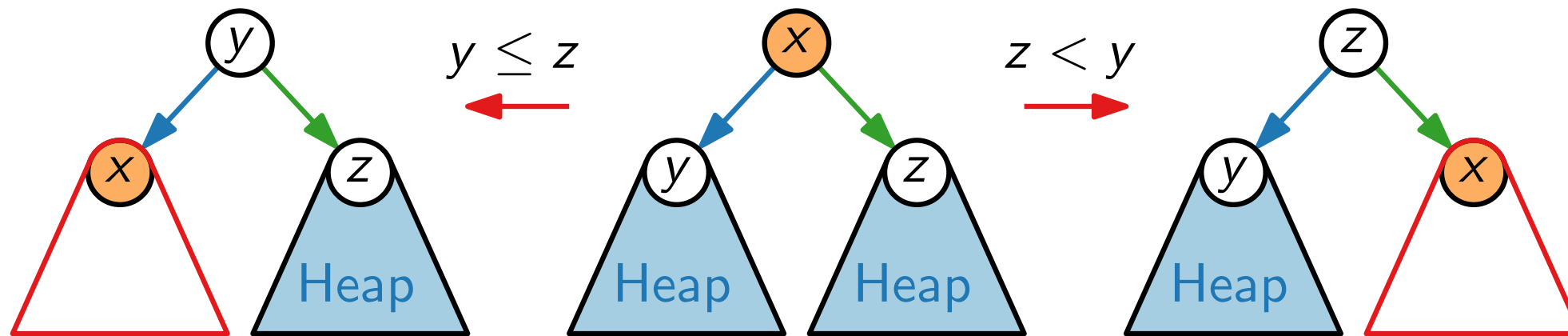
```

MINHEAPIFY(int[] A, index i)
  ℓ = LEFT(i); r = RIGHT(i)
  min = i
  if ℓ ≤ A.heap-size and A[ℓ] < A[i] then
    min = ℓ
  if r ≤ A.heap-size and A[r] < A[min] then
    min = r
  if min ≠ i then
    A[i] ↔ A[min]
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

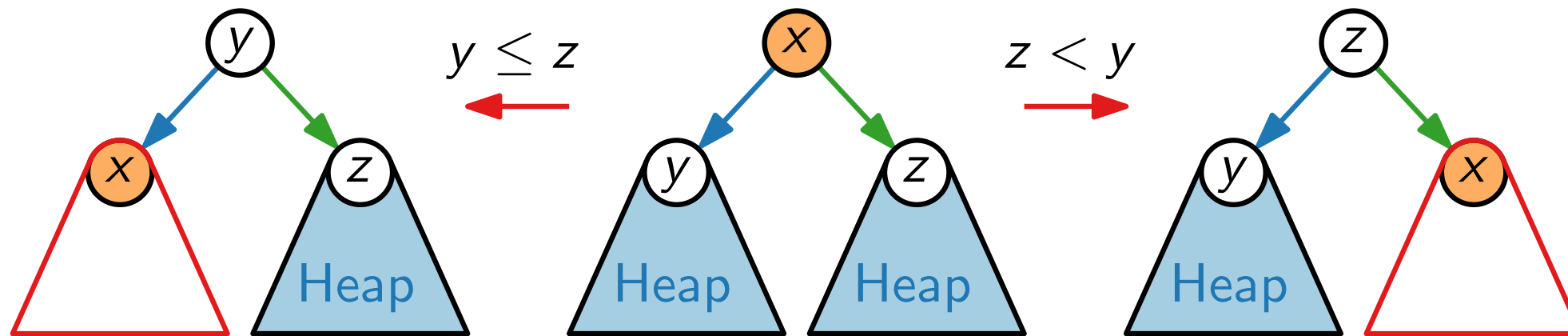
MINHEAPIFY(int[] A, index i)
  ℓ = LEFT(i); r = RIGHT(i)
  min = i
  if ℓ ≤ A.heap-size and A[ℓ] < A[i] then
    min = ℓ
  if r ≤ A.heap-size and A[r] < A[min] then
    min = r
  if min ≠ i then
    A[i] ↔ A[min]
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

Laufzeit?

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

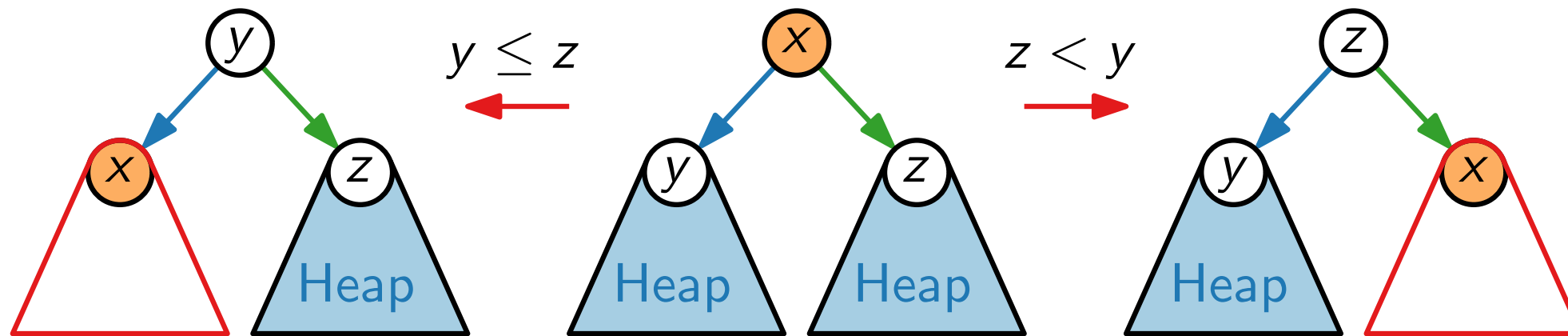
MINHEAPIFY(int[] A, index i)
   $\ell = \text{LEFT}(i); r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
     $A[i] \leftrightarrow A[\text{min}]$ 
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

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

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

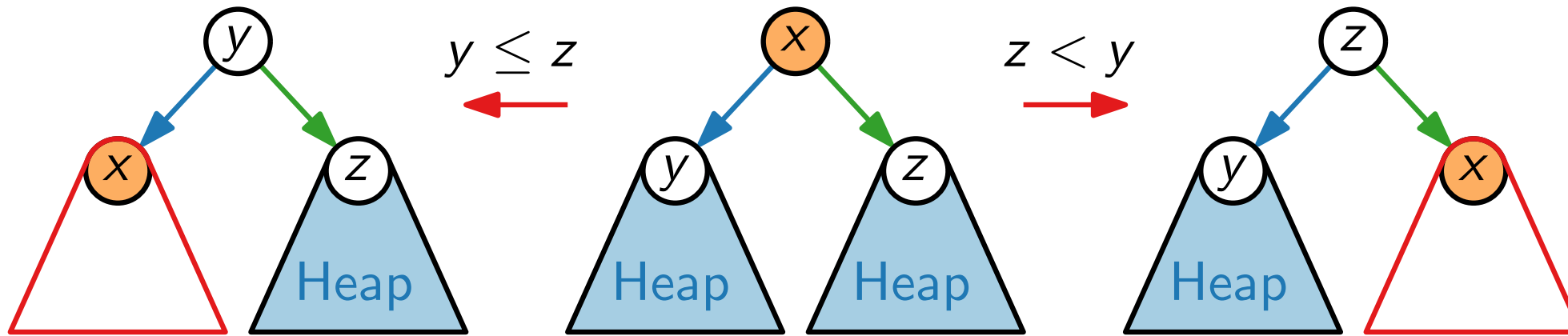
MINHEAPIFY(int[] A, index i)
  ℓ = LEFT(i); r = RIGHT(i)
  min = i
  if ℓ ≤ A.heap-size and A[ℓ] < A[i] then
    min = ℓ
  if r ≤ A.heap-size and A[r] < A[min] then
    min = r
  if min ≠ i then
    A[i] ↔ A[min]
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

Laufzeit? $T_{MH}(n, i)$
 := Anzahl der Tauschoperationen

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

MINHEAPIFY(int[] A, index i)
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
     $A[i] \leftrightarrow A[\text{min}]$ 
    MINHEAPIFY( $A, \text{min}$ )
  
```

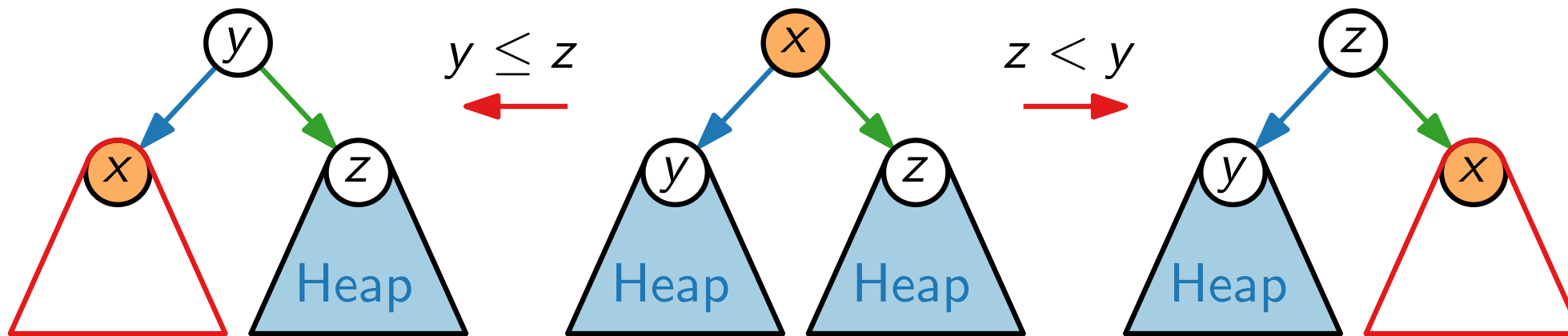
Lokale Strategie: top-down

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

:= Anzahl der Tauschoperationen
 $\leq$ Länge des Weges von
 Knoten i zu einem Blatt

Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

MINHEAPIFY(int[] A, index i)
  ℓ = LEFT(i); r = RIGHT(i)
  min = i
  if ℓ ≤ A.heap-size and A[ℓ] < A[i] then
    min = ℓ
  if r ≤ A.heap-size and A[r] < A[min] then
    min = r
  if min ≠ i then
    A[i] ↔ A[min]
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

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

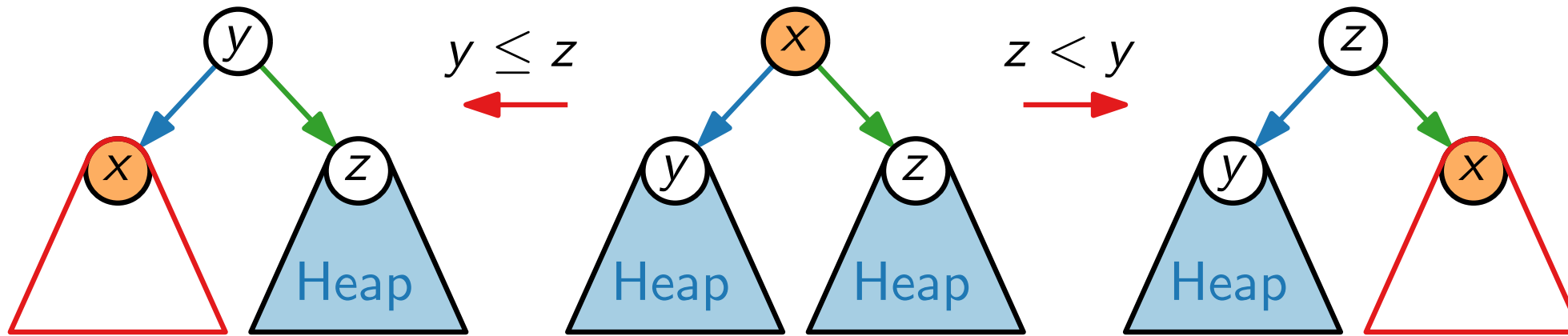
$\quad \quad \quad :=$ Anzahl der Tauschoperationen

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

$\quad \quad \quad \leq$ Höhe von i im Heap

Elementaroperation

„Versickere“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

MINHEAPIFY(int[] A, index i)
  ℓ = LEFT(i); r = RIGHT(i)
  min = i
  if ℓ ≤ A.heap-size and A[ℓ] < A[i] then
    min = ℓ
  if r ≤ A.heap-size and A[r] < A[min] then
    min = r
  if min ≠ i then
    A[i] ↔ A[min]
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

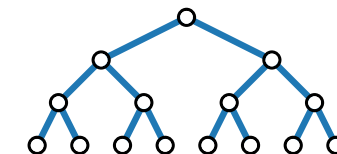
Laufzeit? $T_{MH}(n, i)$

$\coloneqq$ Anzahl der Tauschoperationen

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

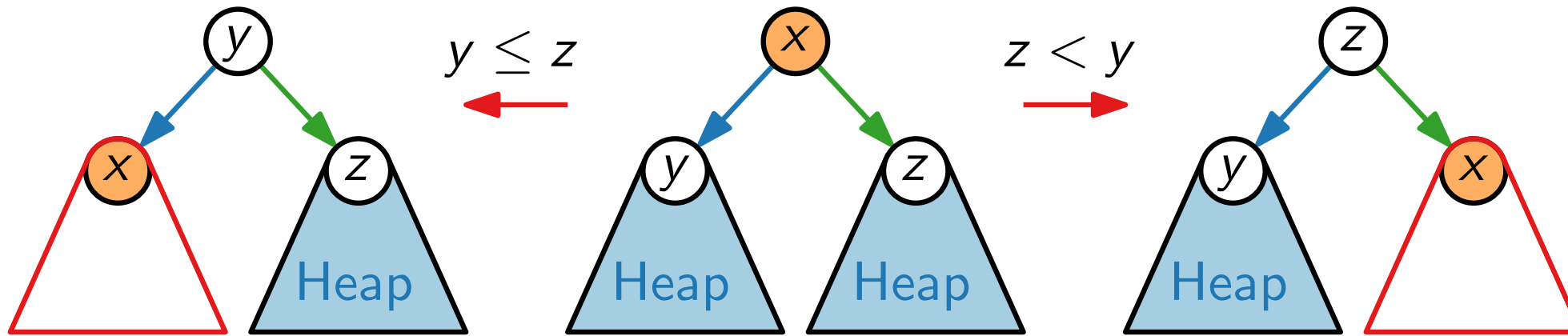
$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps



Elementaroperation

„**Versickere**“ x , falls x zu groß, d.h. falls $x > \min(y, z)$



```

MINHEAPIFY(int[] A, index i)
  ℓ = LEFT(i); r = RIGHT(i)
  min = i
  if ℓ ≤ A.heap-size and A[ℓ] < A[i] then
    min = ℓ
  if r ≤ A.heap-size and A[r] < A[min] then
    min = r
  if min ≠ i then
    A[i] ↔ A[min]
    MINHEAPIFY(A, min)
  
```

Lokale Strategie: top-down

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

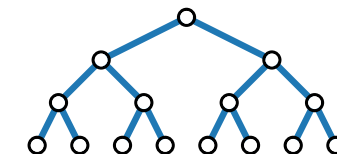
$\coloneqq$ Anzahl der Tauschoperationen

$\leq$ 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**

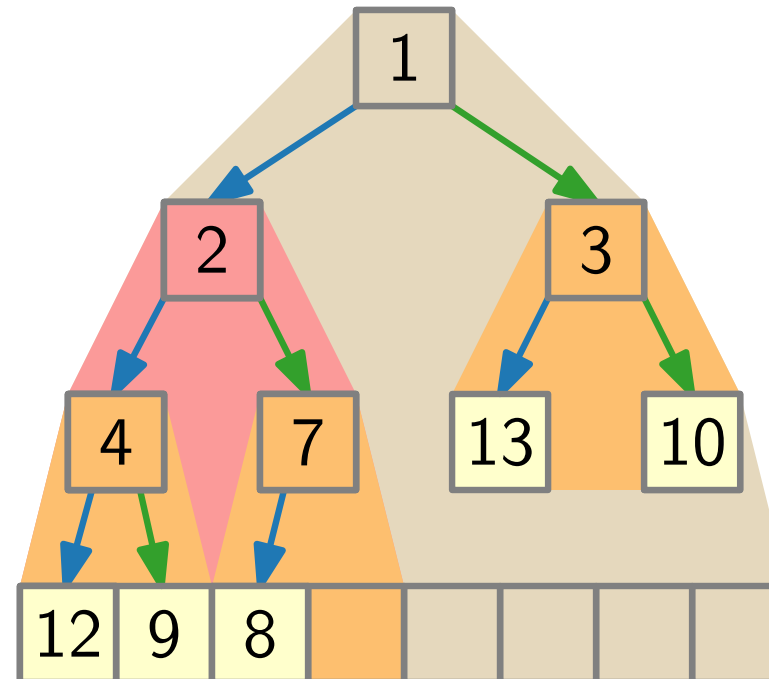
Laufzeit: $T_{MH}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up



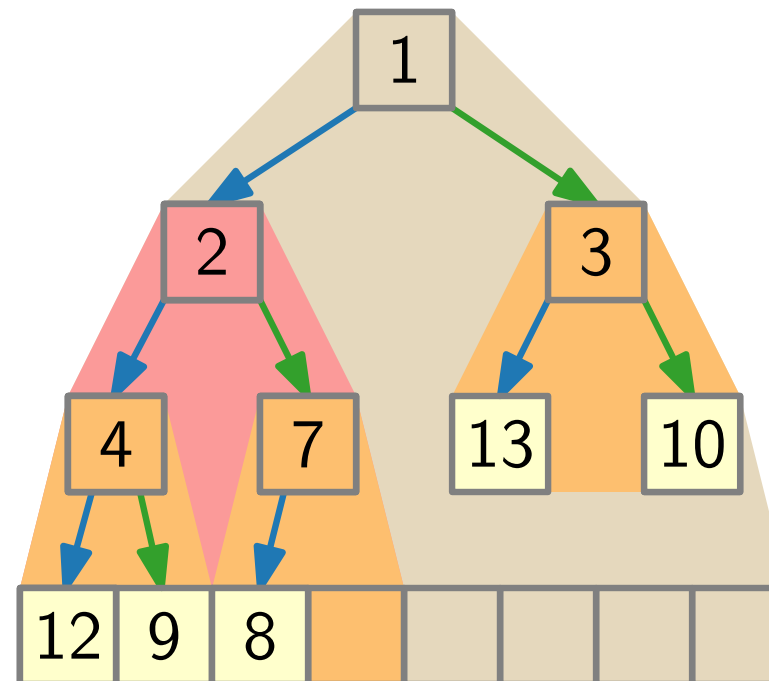
Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for  $i = \lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```



Das große Ganze

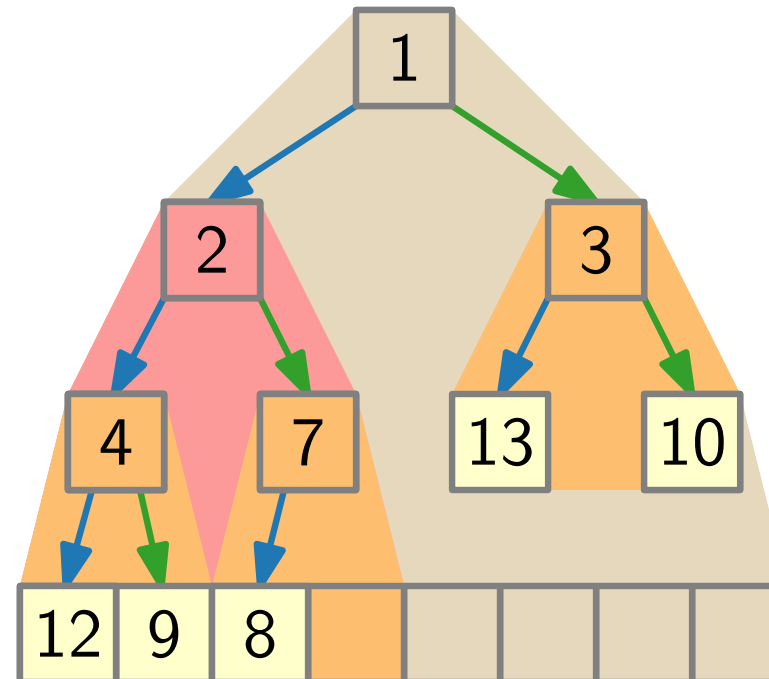
Lokale Strategie: top-down

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

Globale Strategie: bottom-up

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

Laufzeit.



Das große Ganze

Lokale Strategie: top-down

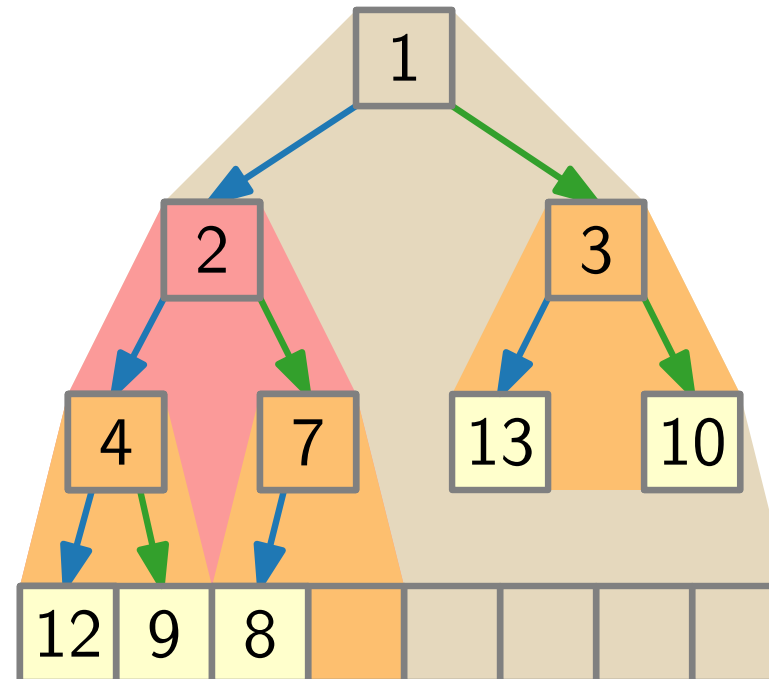
Laufzeit: $T_{MH}(n, i) \leq \text{Höhe von Knoten } i \text{ im Heap der Größe } n$

Globale Strategie: bottom-up

```

BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for  $i = \lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
  
```

Laufzeit. grob: $\mathcal{O}(n \log n)$



Das große Ganze

Lokale Strategie: top-down

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

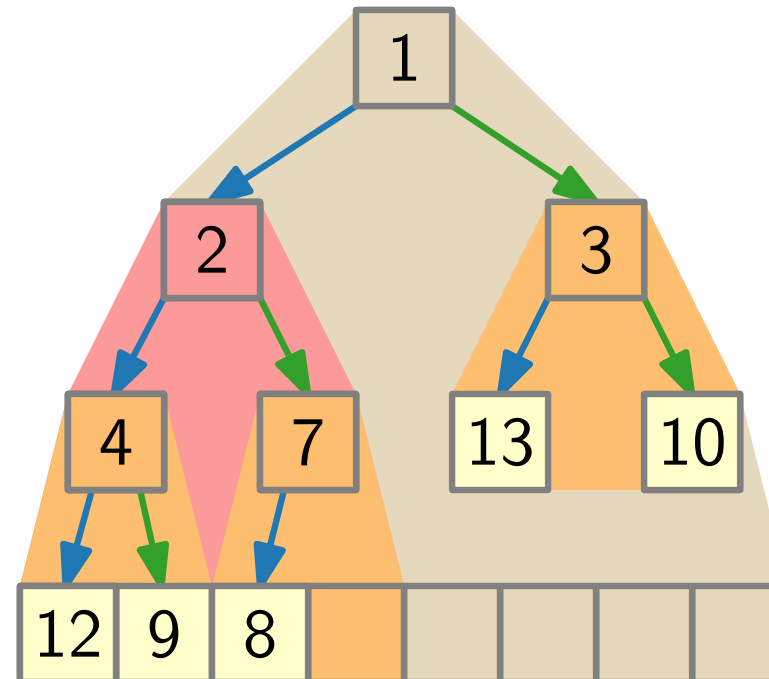
Globale Strategie: bottom-up

```

BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
  
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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



Das große Ganze

Lokale Strategie: top-down

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

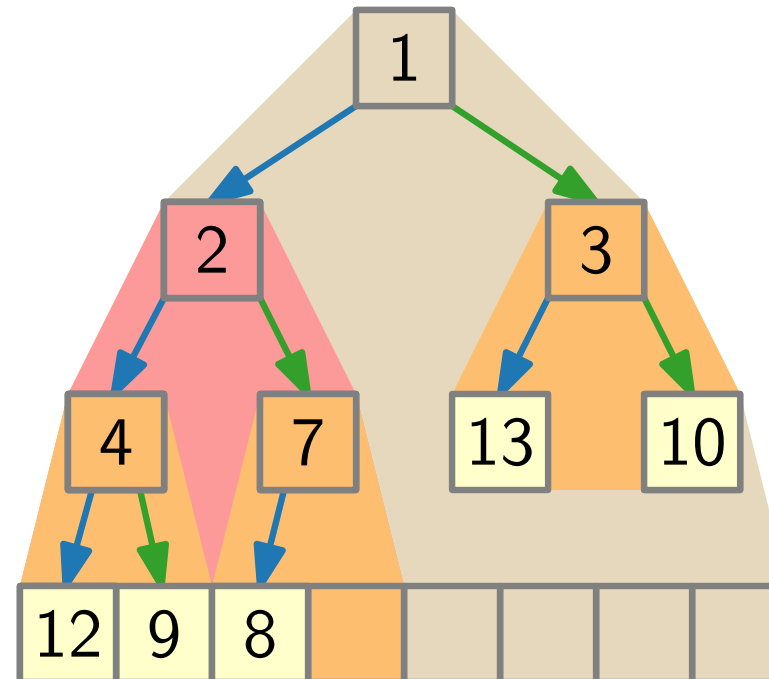
Globale Strategie: bottom-up

```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

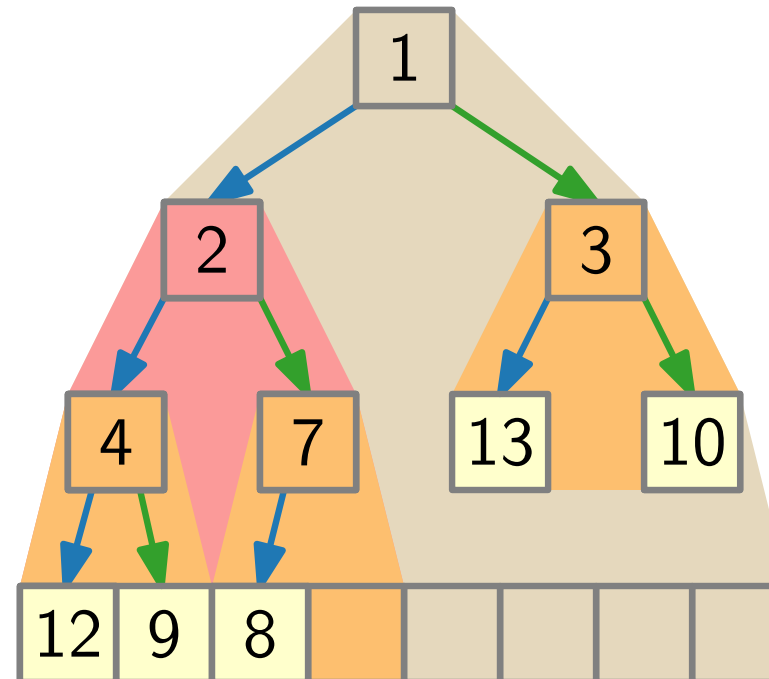
```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

$\approx$



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

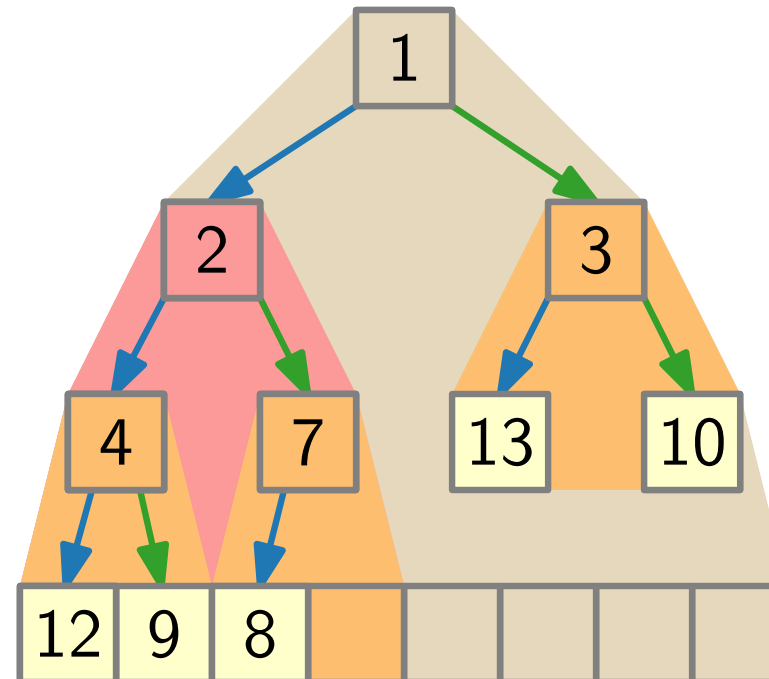
```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```

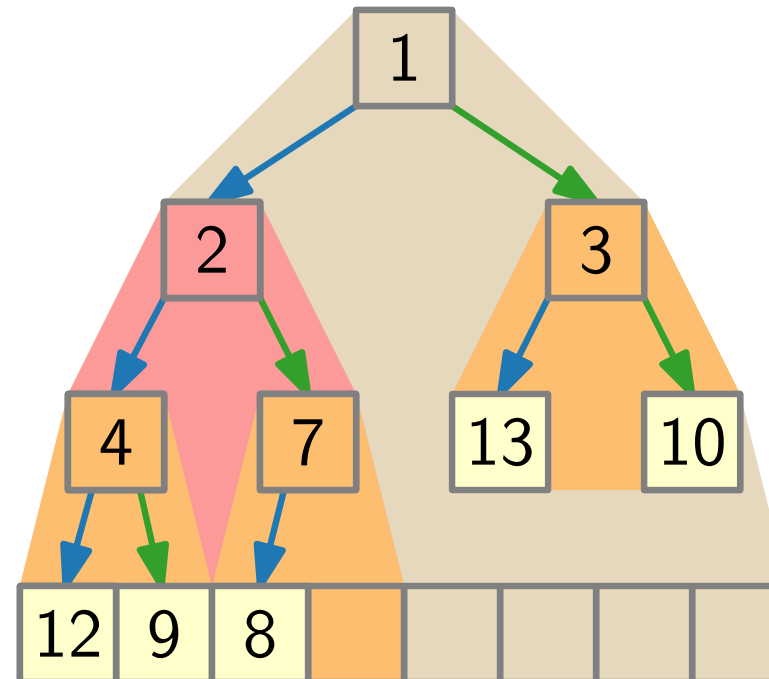
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
  
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

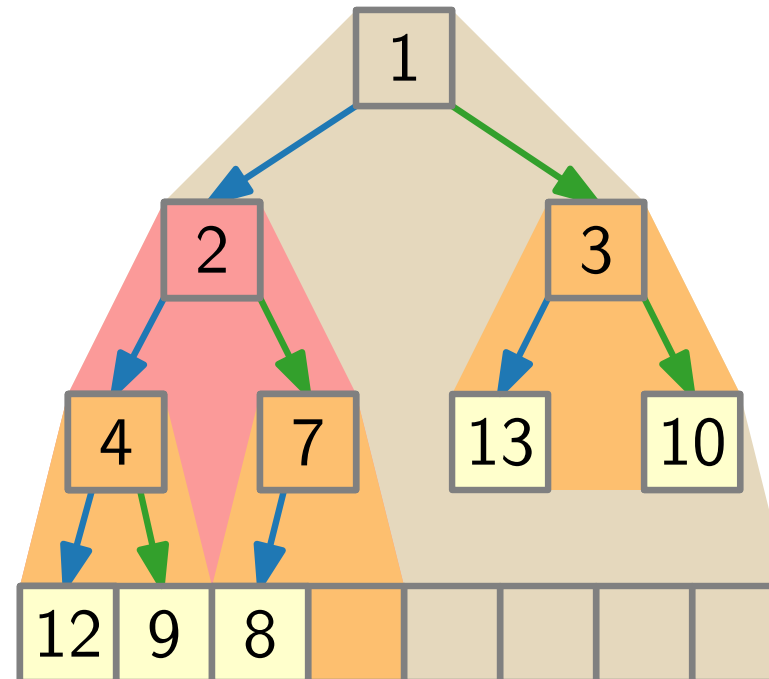
```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```

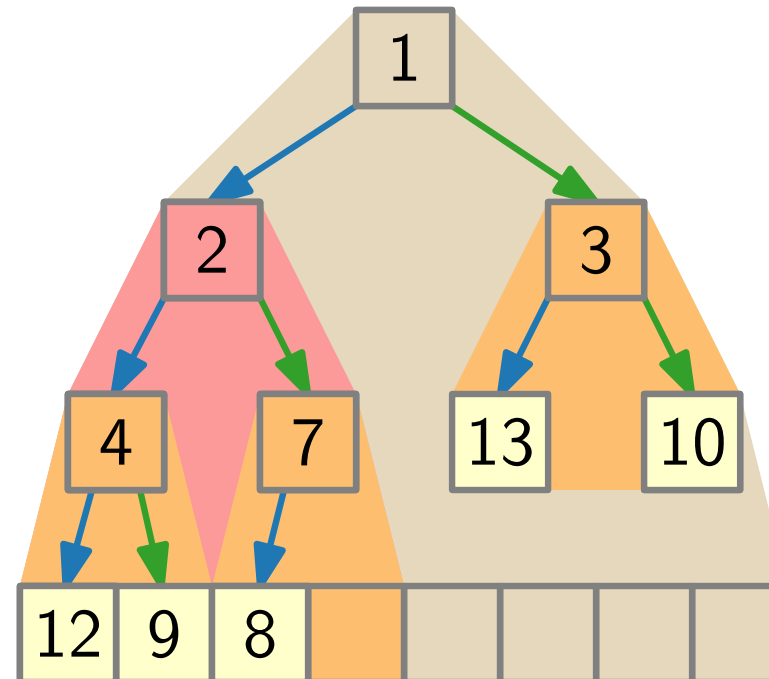
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
  
```

Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```

BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
  
```

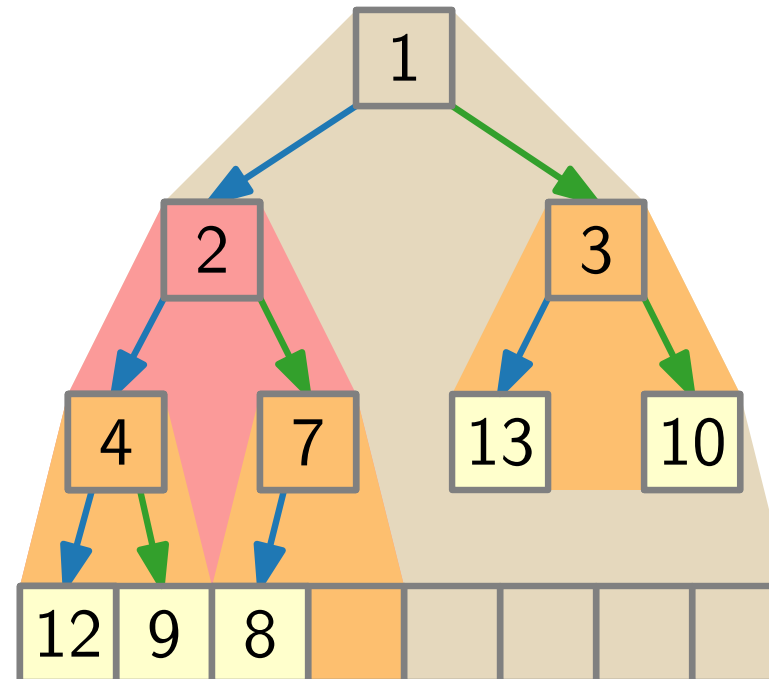
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

$$= n \cdot$$



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

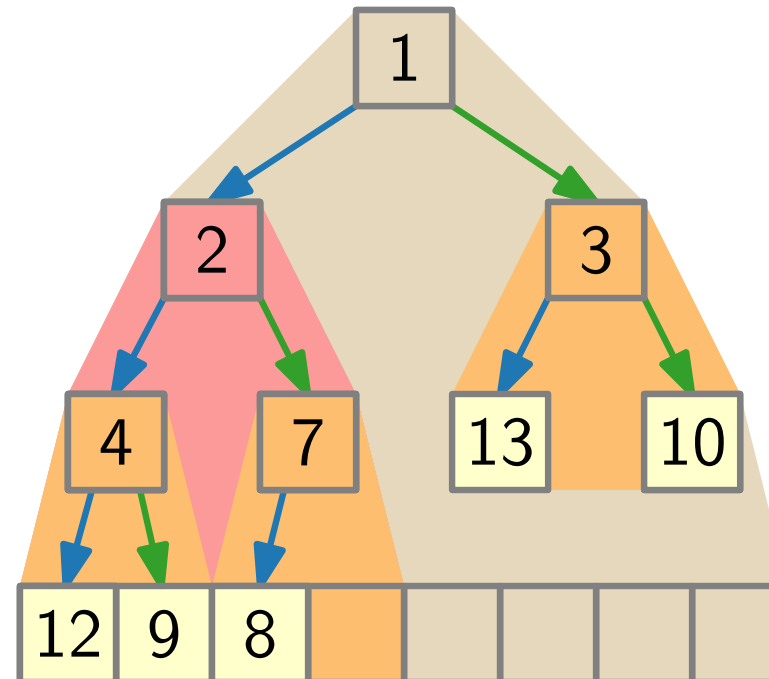
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

$$= n \cdot \sum_{i=1}^{\lfloor \log n \rfloor}$$



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

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

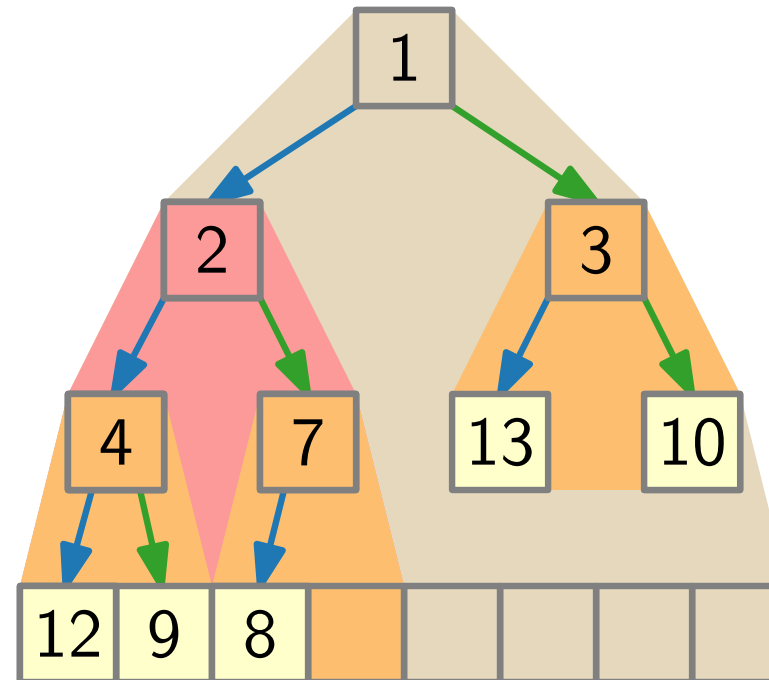
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

$$= n \cdot \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot$$



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

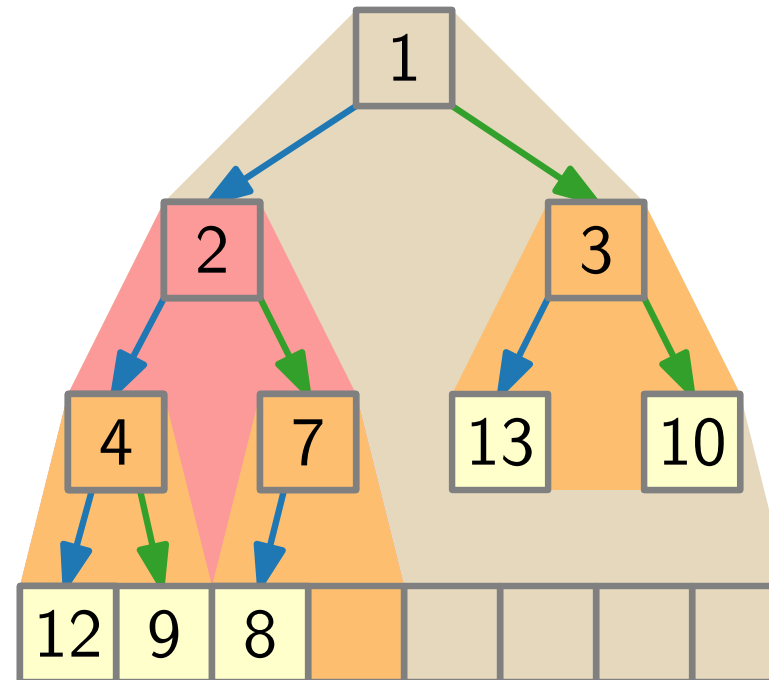
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

```
BUILDMINHEAP(int[] A)
  A.heap-size = A.length
  for i =  $\lfloor A.length/2 \rfloor$  downto 1 do
    MINHEAPIFY(A, i)
```

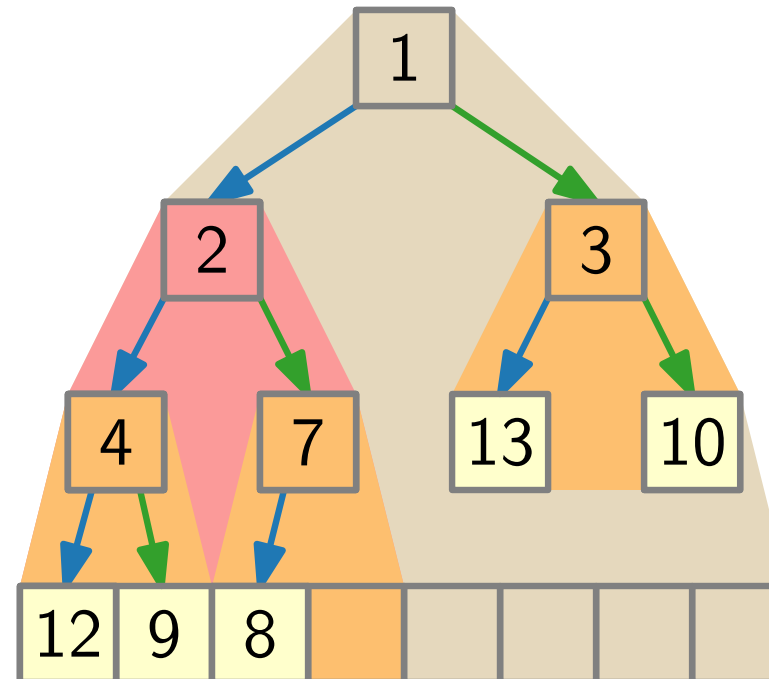
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

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



Fortsetzung Laufzeitanalyse

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

Fortsetzung Laufzeitanalyse

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

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \text{ geometrische Reihe}$$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \quad \text{geometrische Reihe}$$

$$2') \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$$

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \quad \text{geometrische Reihe}$$

$$2') \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$$

Wir hätten gerne:

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \quad \text{geometrische Reihe}$$

$$2') \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$$

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} =$$

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \quad \text{geometrische Reihe}$$

$$2') \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$$

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = ?$$

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \quad \text{geometrische Reihe}$$

$$2') \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$$

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = ?$$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$

ableiten!

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = ?$$

Fortsetzung Laufzeitanalyse

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

$$2) \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \quad \text{geometrische Reihe}$$

$$2') \sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$$

ableiten!

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i \right)' =$$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$

ableiten!

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i \right)' =$$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$

ableiten!

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i \right)' = \left(\frac{1}{1-q} \right)' =$$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i \right)' = \left(\frac{1}{1-q} \right)' =$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' =$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$

ableiten!

Quotientenregel:

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

ableiten!

Wir hätten gerne:

$$\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2}$$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2} = \frac{1}{(1-q)^2}$

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2}$

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

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2}$

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

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2}$

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

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2}$

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

Fortsetzung Laufzeitanalyse

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

2) $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ ableiten!

Quotientenregel:

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

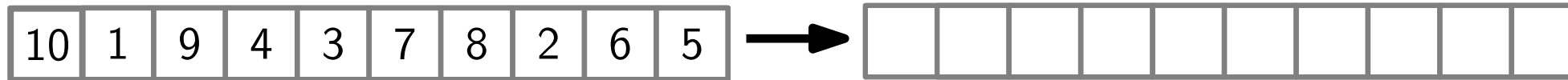
Wir hätten gerne: ableiten! $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q) \cdot 0 - (-1) \cdot 1}{(1-q)^2}$

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

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

Übung Heap-Aufbau

Aufgabe. Bauen Sie einen Heap mit BUILDMINHEAP!



```

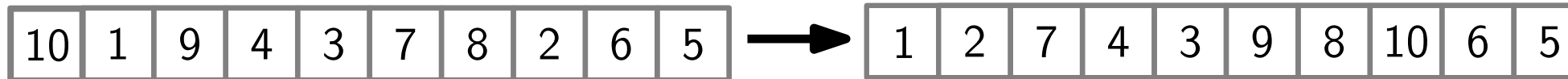
MINHEAPIFY(int[] A, index  $i$ )
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
     $A[i] \leftrightarrow A[\text{min}]$ 
    MINHEAPIFY( $A, \text{min}$ )
  
```

```

BUILDMINHEAP(int[] A)
   $A.\text{heap-size} = A.\text{length}$ 
  for  $i = \lfloor A.\text{length}/2 \rfloor$  downto 1 do
    MINHEAPIFY( $A, i$ )
  
```

Übung Heap-Aufbau

Aufgabe. Bauen Sie einen Heap mit BUILDMINHEAP!



```

MINHEAPIFY(int[] A, index  $i$ )
   $\ell = \text{LEFT}(i)$ ;  $r = \text{RIGHT}(i)$ 
   $\text{min} = i$ 
  if  $\ell \leq A.\text{heap-size}$  and  $A[\ell] < A[i]$  then
     $\text{min} = \ell$ 
  if  $r \leq A.\text{heap-size}$  and  $A[r] < A[\text{min}]$  then
     $\text{min} = r$ 
  if  $\text{min} \neq i$  then
     $A[i] \leftrightarrow A[\text{min}]$ 
    MINHEAPIFY( $A, \text{min}$ )
  
```

```

BUILDMINHEAP(int[] A)
   $A.\text{heap-size} = A.\text{length}$ 
  for  $i = \lfloor A.\text{length}/2 \rfloor$  downto 1 do
    MINHEAPIFY( $A, i$ )
  
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ then
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ then
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size --$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[\text{PARENT}(i)] > A[i]$

└

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[\text{PARENT}(i)] > A[i]$

$A[i] \leftrightarrow A[\text{PARENT}(i)]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[PARENT(i)] > A[i]$

$A[i] \leftrightarrow A[PARENT(i)]$

$i = PARENT(i)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[PARENT(i)] > A[i]$

$A[i] \leftrightarrow A[PARENT(i)]$

$i = PARENT(i)$

INSERT(priorität p)

$A.heap-size++$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[PARENT(i)] > A[i]$

$A[i] \leftrightarrow A[PARENT(i)]$

$i = PARENT(i)$

INSERT(priorität p)

$A.heap-size++$

if $A.heap-size > A.length$ **then error...**

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[PARENT(i)] > A[i]$

$A[i] \leftrightarrow A[PARENT(i)]$

$i = PARENT(i)$

INSERT(priorität p)

$A.heap-size++$

if $A.heap-size > A.length$ **then error...**

$A[A.heap-size] = \infty$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

return $A[1]$

EXTRACTMIN()

if $A.heap-size < 1$ **then**
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p)

if $p > A[i]$ **then error** „prio. too large“

$A[i] = p$

while $i > 1$ **and** $A[PARENT(i)] > A[i]$

$A[i] \leftrightarrow A[PARENT(i)]$

$i = PARENT(i)$

INSERT(priorität p)

$A.heap-size++$

if $A.heap-size > A.length$ **then error...**

$A[A.heap-size] = \infty$

DECREASEKEY($A.heap-size, p$)

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(\quad)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\quad)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\quad)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(1)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\quad)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\quad)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN()

$\mathcal{O}(1)$

return $A[1]$

EXTRACTMIN() $\mathcal{O}(\quad)$

if $A.heap-size < 1$ then
 error „Heap underflow“

$min = A[1]$

$A[1] = A[A.heap-size]$

$A.heap-size--$

MINHEAPIFY($A, 1$)

return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\quad)$

if $p > A[i]$ then error „prio. too large“

$A[i] = p$

while $i > 1$ and $A[PARENT(i)] > A[i]$

$A[i] \leftrightarrow A[PARENT(i)]$

$i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$

$A.heap-size++$

if $A.heap-size > A.length$ then error...

$A[A.heap-size] = \infty$

DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(1)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\log n)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\quad)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(1)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\log n)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\quad)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(1)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\log n)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\log n)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(1)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\log n)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\log n)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\quad)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FINDMIN() $\mathcal{O}(1)$
return $A[1]$

EXTRACTMIN() $\mathcal{O}(\log n)$
if $A.heap-size < 1$ then
 error „Heap underflow“
 $min = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size--$
MINHEAPIFY($A, 1$)
return min

DECREASEKEY(index i , prio. p) $\mathcal{O}(\log n)$
if $p > A[i]$ then error „prio. too large“
 $A[i] = p$
while $i > 1$ and $A[PARENT(i)] > A[i]$
 $A[i] \leftrightarrow A[PARENT(i)]$
 $i = PARENT(i)$

INSERT(priorität p) $\mathcal{O}(\log n)$
 $A.heap-size++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = \infty$
DECREASEKEY($A.heap-size, p$)

Laufzeiten?

HeapSort

Idee:

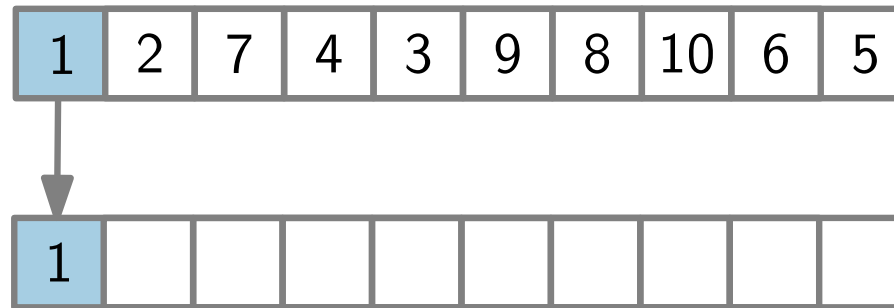
HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

1	2	7	4	3	9	8	10	6	5
---	---	---	---	---	---	---	----	---	---

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

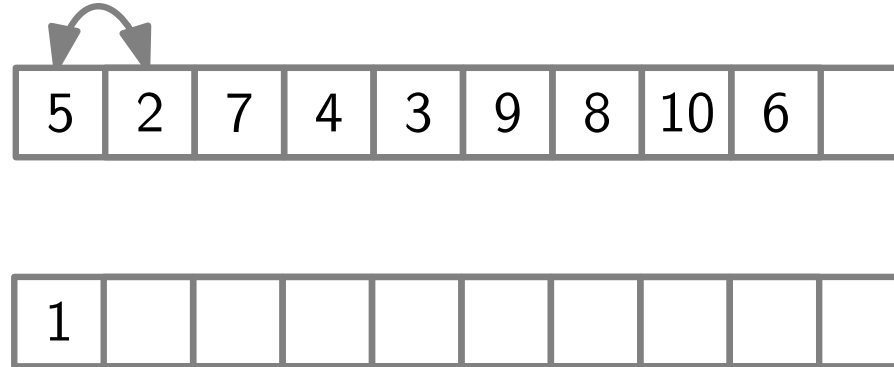


The diagram shows a horizontal sequence of 10 boxes, each containing a number. The numbers are 1, 2, 7, 4, 3, 9, 8, 10, 6, and 5. A curved arrow originates from the right side of the last box (5) and points to the top of the first box (1), indicating a circular linked list structure.

The diagram illustrates the merging step of a merge sort algorithm. It consists of two horizontal arrays of boxes. The top array contains the numbers 5, 2, 7, 4, 3, 9, 8, 10, 6, followed by an empty box. The bottom array contains the number 1, followed by eight empty boxes. A curved arrow originates from the right side of the top array and points to the first empty box in the bottom array, indicating the process of merging the two sorted sub-arrays.

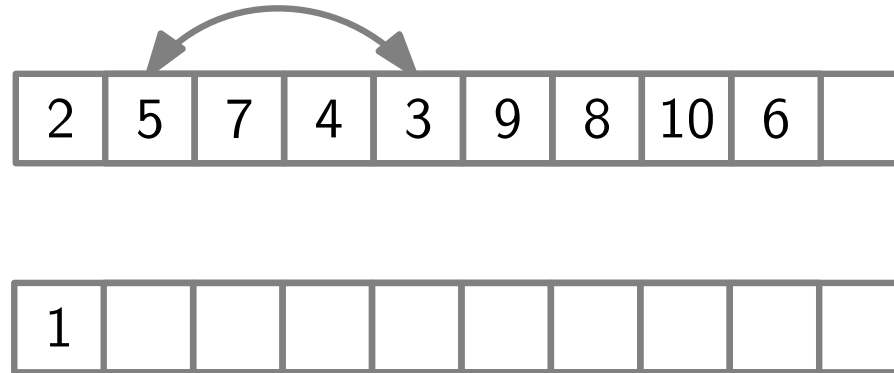
HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.



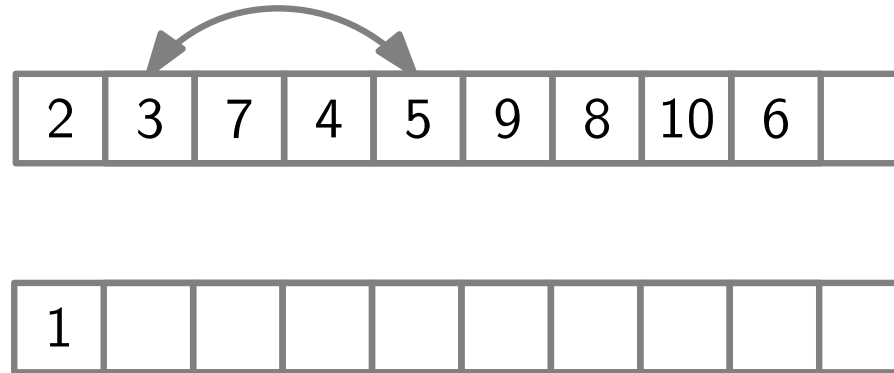
HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.



HeapSort

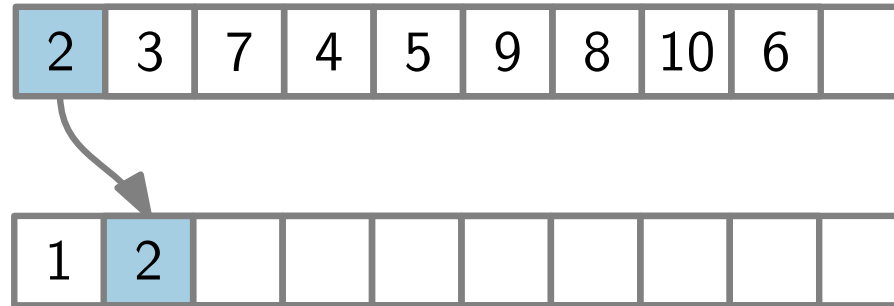
Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.



HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

Min-Heap



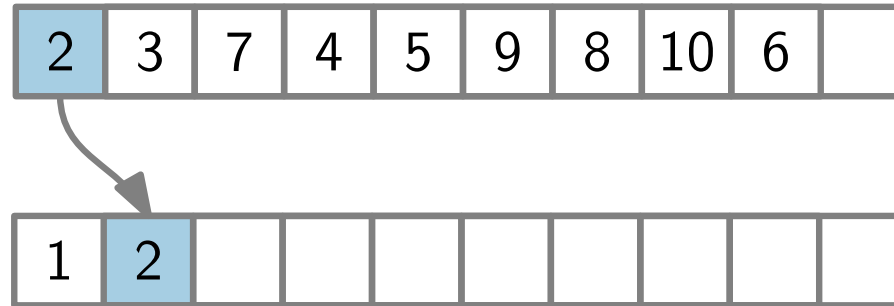
HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
```

Schreiben *Sie* den Pseudocode.
Verwenden Sie
`BUILDMINHEAP` und
`EXTRACTMIN`.

Min-Heap

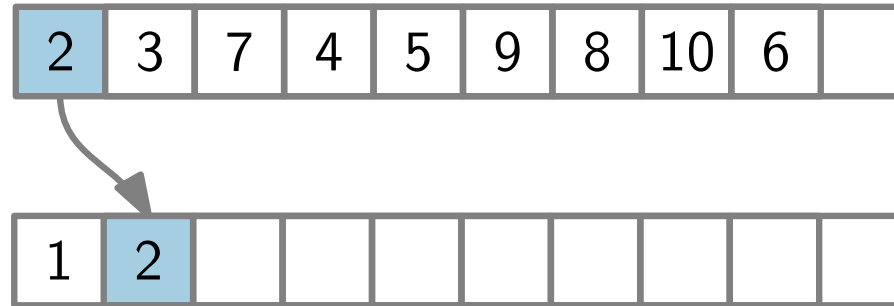


HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)  
    BUILDMINHEAP(A)
```

Min-Heap



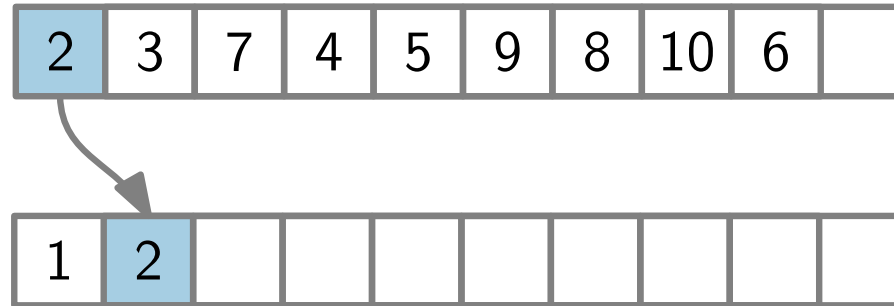
HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]

  return B
```

Min-Heap

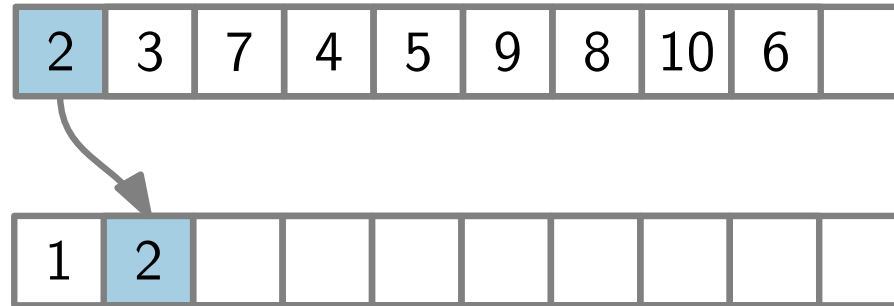


HeapSort

Idee: ■ EXTRACTMIN() gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    |
  return B
```

Min-Heap

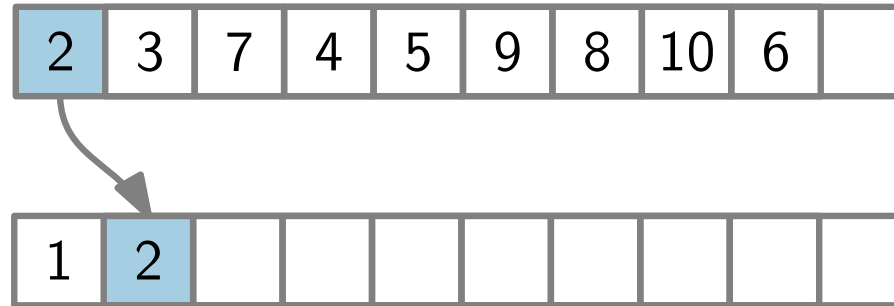


HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap

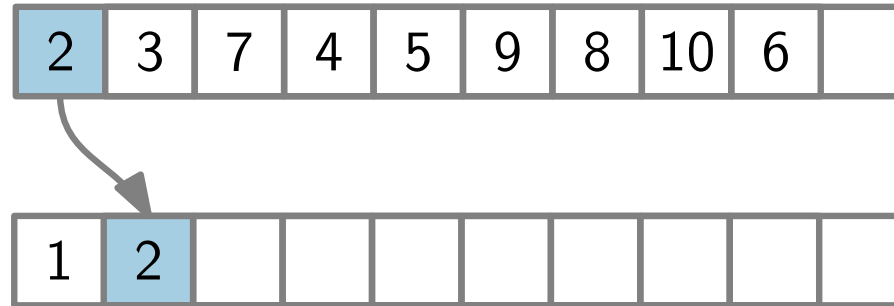


HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

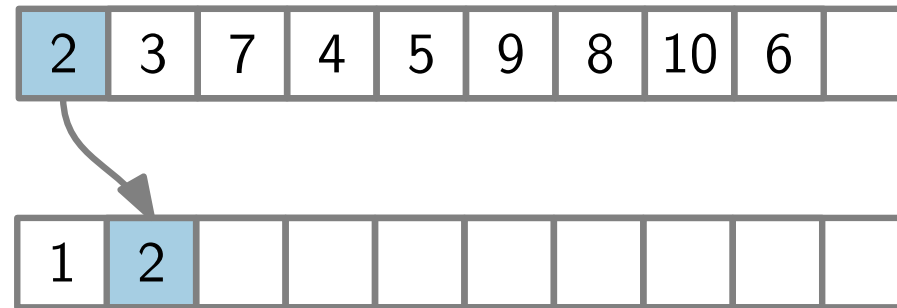
Obere Schranke: $T_{HS}(n)$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

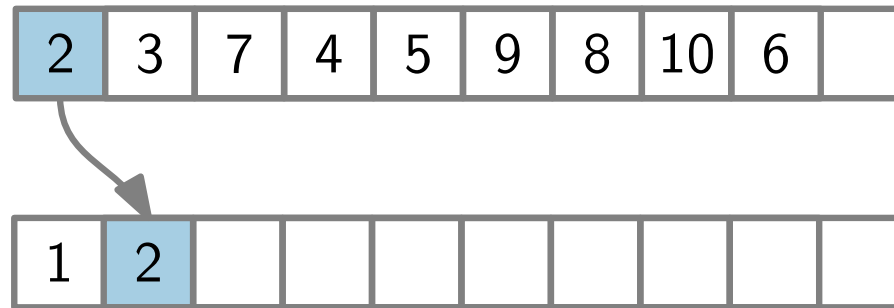
Obere Schranke: $T_{\text{HS}}(n) \in$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

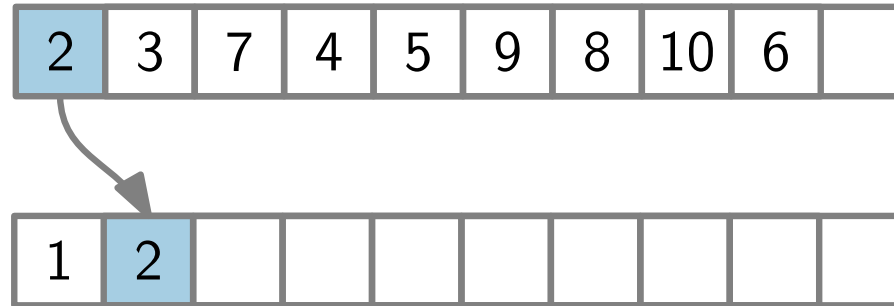
Obere Schranke: $T_{HS}(n) \in$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

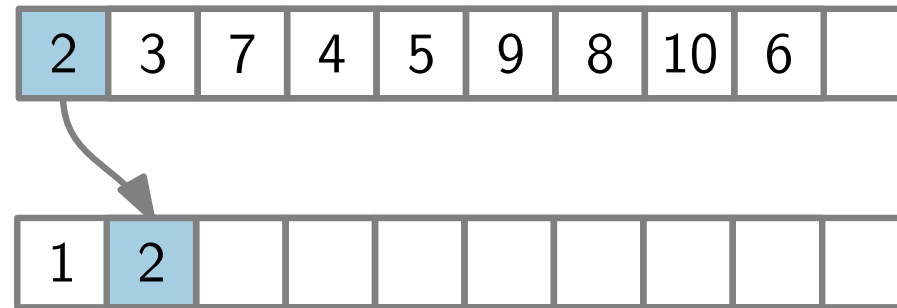
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n)$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

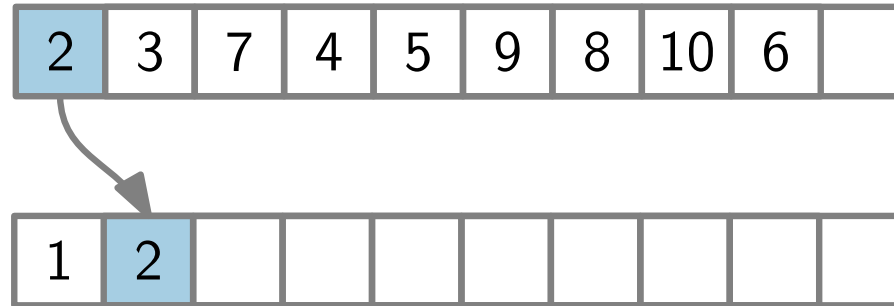
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) +$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

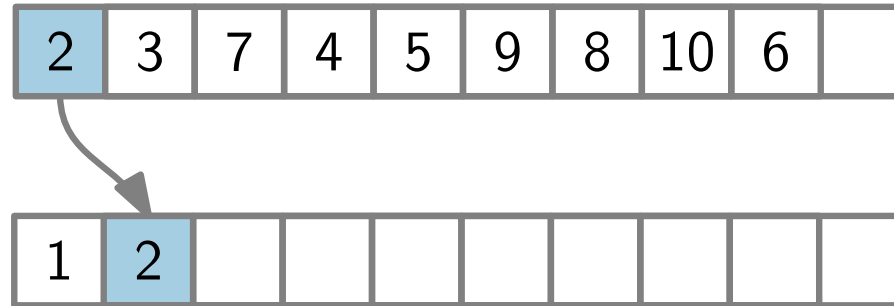
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) +$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

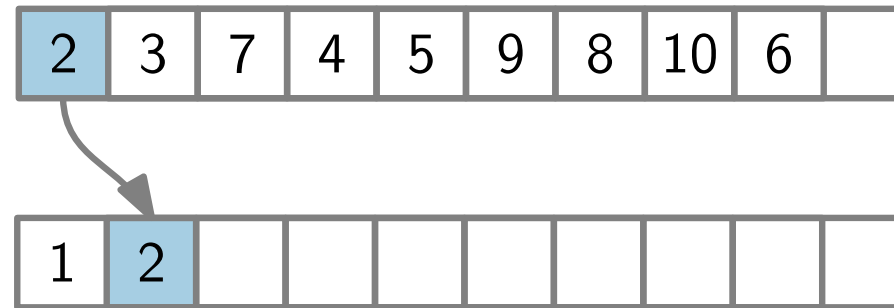
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

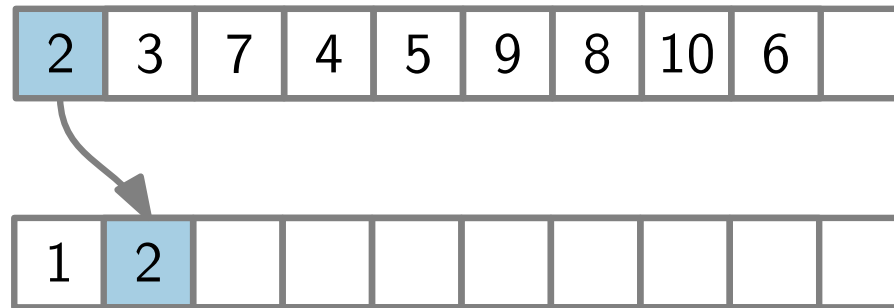
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

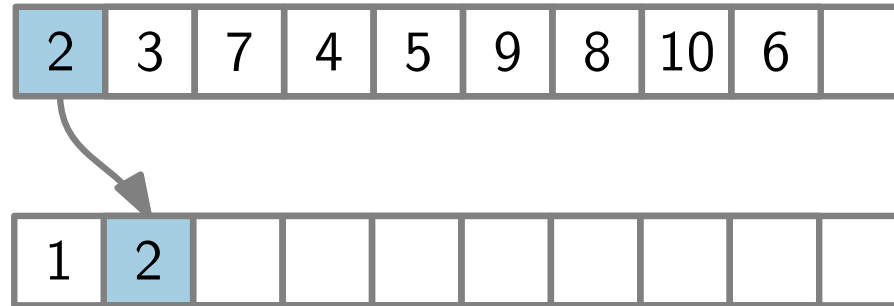
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

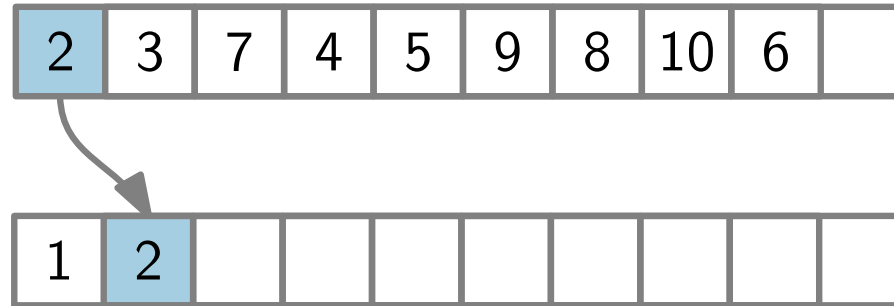
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n)$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

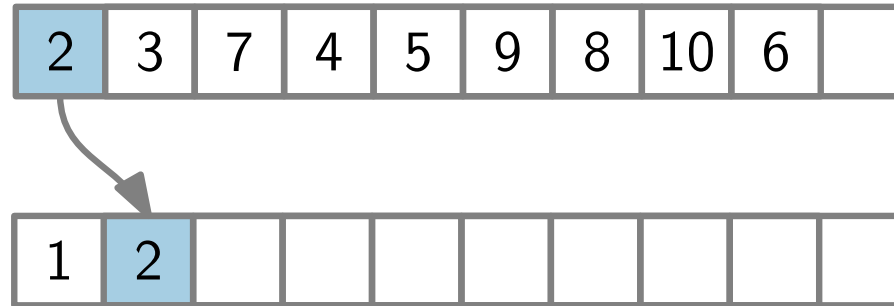
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) =$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

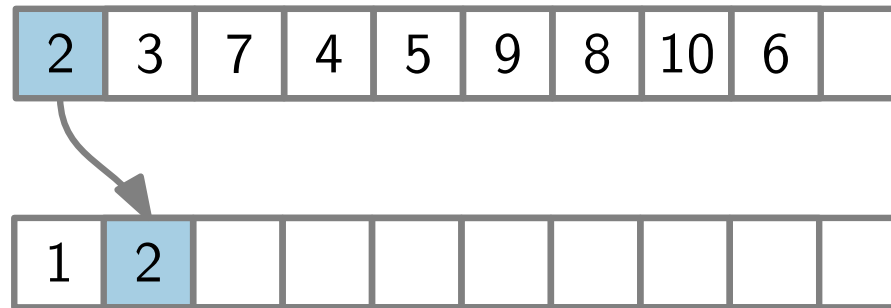
Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

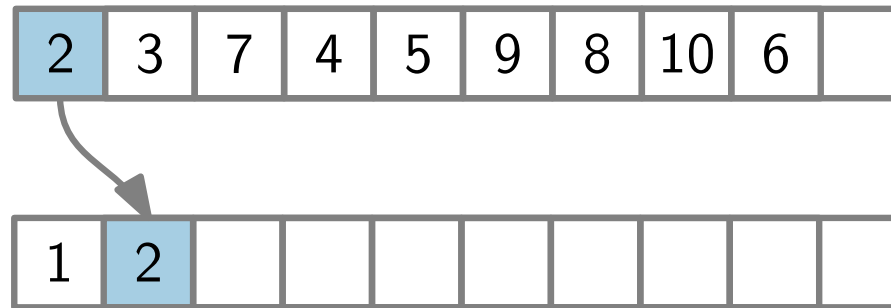
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n +$

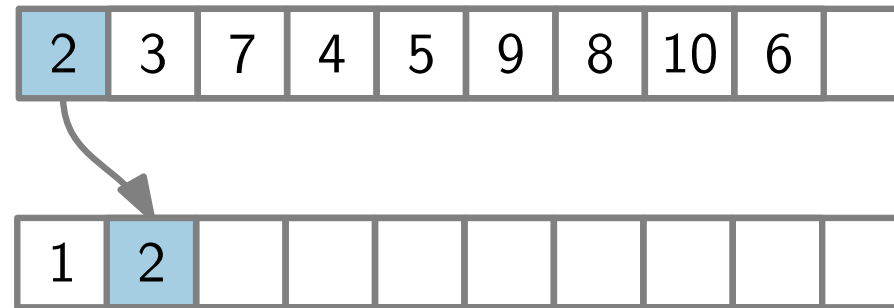
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n$

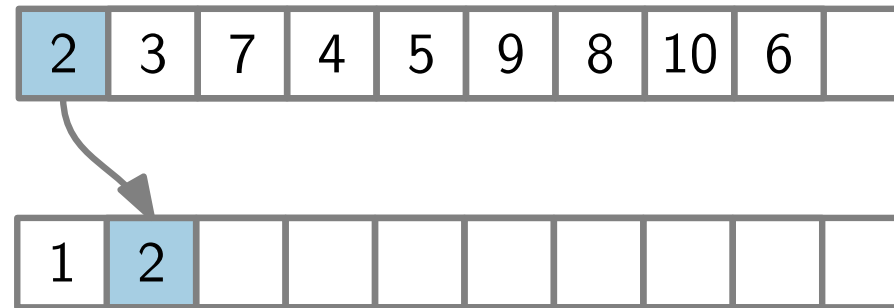
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n c' \cdot \log_2 i \geq$

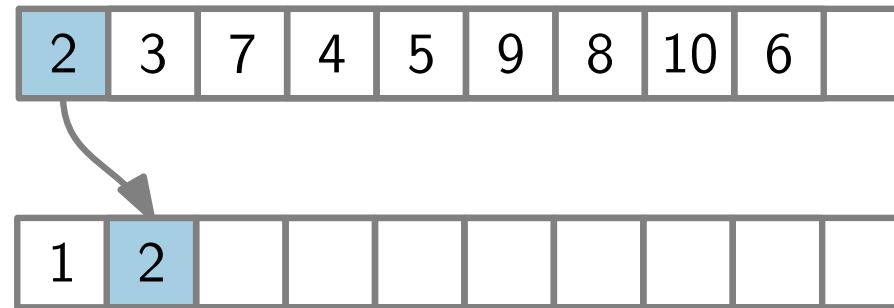
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^n$

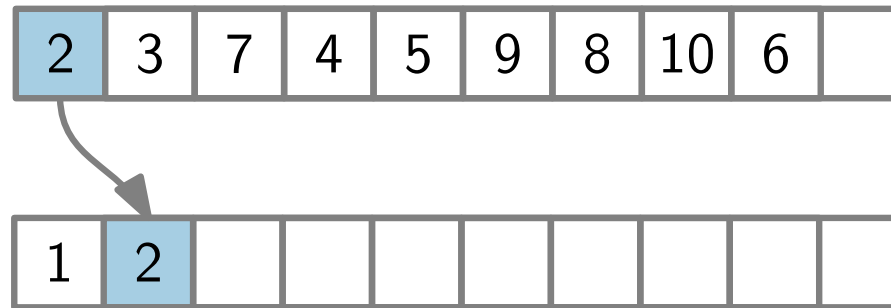
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
    BUILDMINHEAP(A)
    B = new int[A.length]
    for i = 1 to A.length do
        B[i] = EXTRACTMIN()
    return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^n \log_2 \frac{n}{2}$

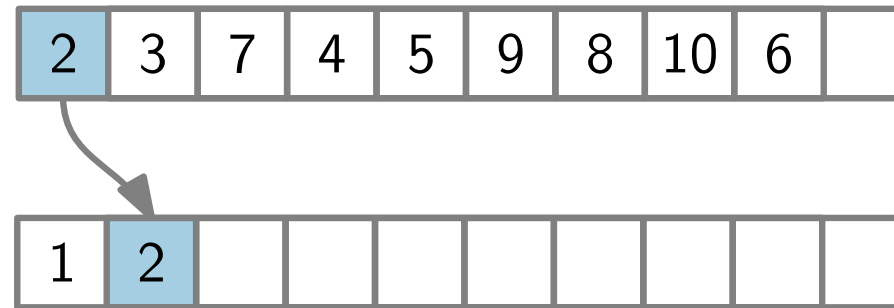
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^n \log_2 \frac{n}{2} \in \Omega(\quad)$

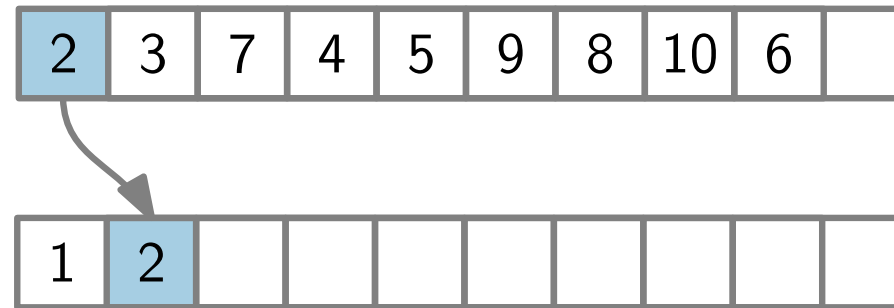
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^n \log_2 \frac{n}{2} \in \Omega(n \log n)$

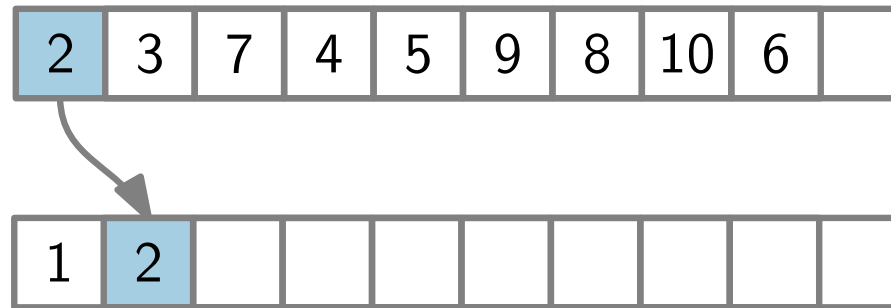
Satz. HEAPSORT sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

```
int[] HEAPSORT(int[] A)
    BUILDMINHEAP(A)
    B = new int[A.length]
    for i = 1 to A.length do
        B[i] = EXTRACTMIN()
    return B
```

Min-Heap



Laufzeit:

Obere Schranke: $T_{\text{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^n c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^n \log_2 \frac{n}{2} \in \Omega(n \log n)$

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

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$
HEAPSORT		

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$
HEAPSORT		$\Theta(n \log n)$

Vergleich Laufzeiten

	Bester Fall	Schlechtester Fall
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

	Bester Fall	Schlechtester Fall	in-situ	
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$		
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$		
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$		
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$		
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$		

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

	Bester Fall	Schlechtester Fall	in-situ	
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$		
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$		
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$		
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$		

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

	Bester Fall	Schlechtester Fall	in-situ	
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$		
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$		
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$		

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

	Bester Fall	Schlechtester Fall	in-situ	
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$		
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$		

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

	Bester Fall	Schlechtester Fall	in-situ	
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$		

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

	Bester Fall	Schlechtester Fall	in-situ	
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✗

Vom Heap zur Sortierung (in-situ)

Idee:

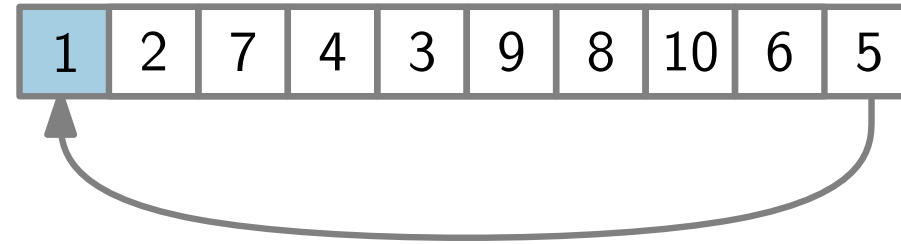
Vom Heap zur Sortierung (in-situ)

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.

1	2	7	4	3	9	8	10	6	5
---	---	---	---	---	---	---	----	---	---

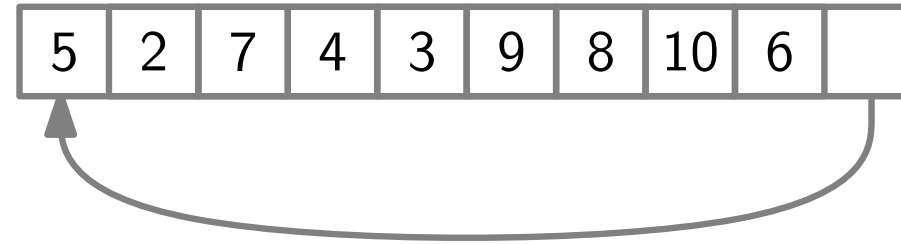
Vom Heap zur Sortierung (in-situ)

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.



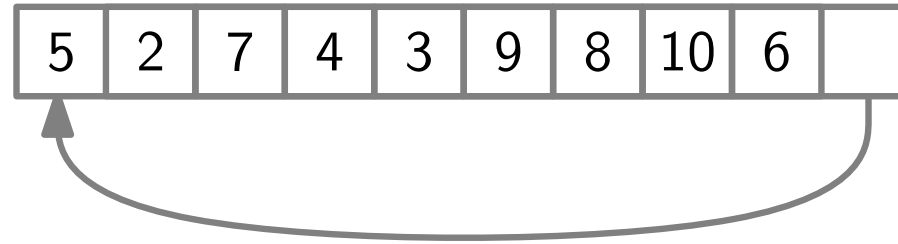
Vom Heap zur Sortierung (in-situ)

Idee: ■ `EXTRACTMIN()` gibt kleinstes Heap-Element aus.



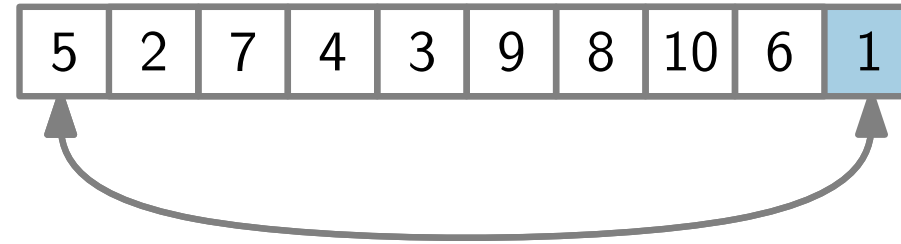
Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei



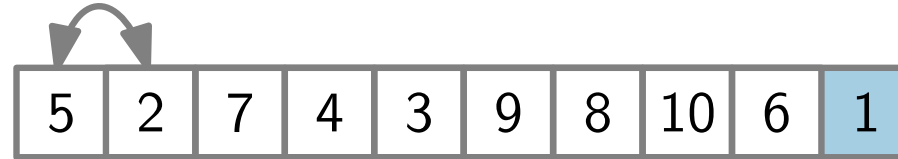
Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei



Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei



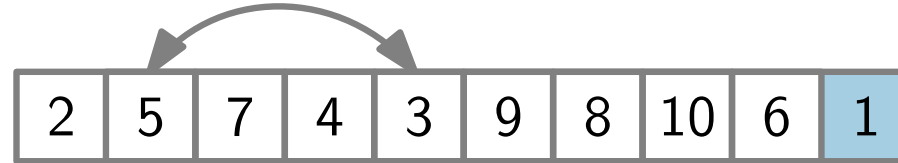
Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

2	5	7	4	3	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

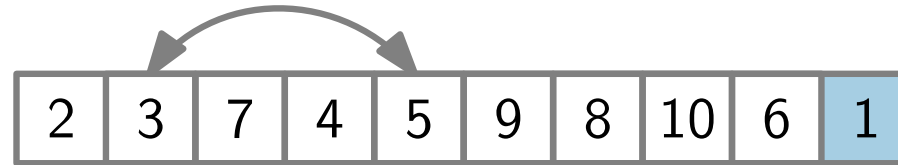
Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei



Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei



Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

`HEAPSORT(int[] A)`

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```
HEAPSORT(int[] A)  
  BUILDMINHEAP(A)
```

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```
HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for  $i = A.length$  downto 2 do
    |
```

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

Vom Heap zur Sortierung (in-situ)

- Idee:**
- EXTRACTMIN() gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```
HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for  $i = A.length$  downto 2 do
     $A[i] = \text{EXTRACTMIN}()$ 
```

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

Vom Heap zur Sortierung (in-situ)

- Idee:**
- EXTRACTMIN() gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```
HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for  $i = A.length$  downto 2 do
     $A[i] = \text{EXTRACTMIN}()$ 
```

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

10	9	8	7	6	5	4	3	2	1
----	---	---	---	---	---	---	---	---	---

Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```
HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for  $i = A.length$  downto 2 do
     $A[i] = \text{EXTRACTMIN}()$ 
  for  $i = 1$  to  $\lfloor A.length/2 \rfloor$  do
    .
```

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

10	9	8	7	6	5	4	3	2	1
----	---	---	---	---	---	---	---	---	---

Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```
HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for  $i = A.length$  downto 2 do
     $A[i] = \text{EXTRACTMIN}()$ 
  for  $i = 1$  to  $\lfloor A.length/2 \rfloor$  do
     $A[i] \leftrightarrow A[n - i]$ 
```

2	3	7	4	5	9	8	10	6	1
---	---	---	---	---	---	---	----	---	---

Min-Heap

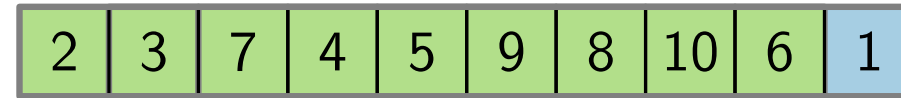
10	9	8	7	6	5	4	3	2	1
----	---	---	---	---	---	---	---	---	---

Vom Heap zur Sortierung (in-situ)

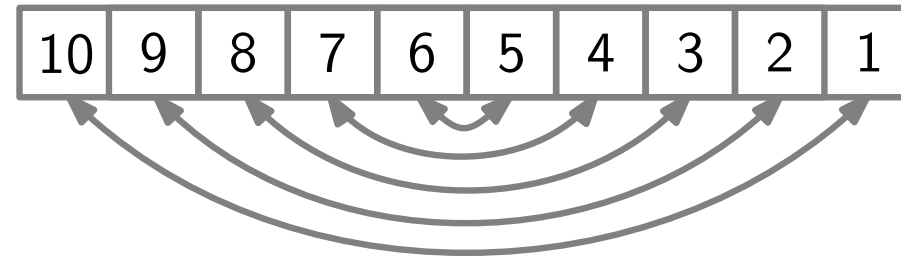
- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```

HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for i = A.length downto 2 do
    A[i] = EXTRACTMIN()
  for i = 1 to ⌊A.length/2⌋ do
    A[i] ↔ A[n - i]
  
```



Min-Heap



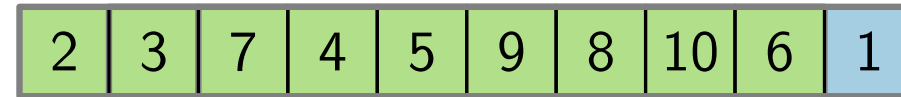
Vom Heap zur Sortierung (in-situ)

- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```

HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for  $i = A.length$  downto 2 do
     $A[i] = \text{EXTRACTMIN}()$ 
  for  $i = 1$  to  $\lfloor A.length/2 \rfloor$  do
     $A[i] \leftrightarrow A[n - i]$ 

```



Min-Heap



Vom Heap zur Sortierung (in-situ)

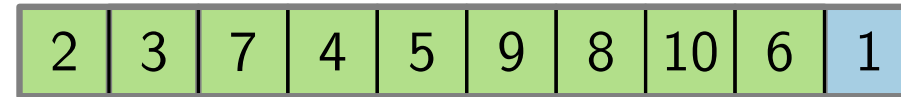
- Idee:**
- `EXTRACTMIN()` gibt kleinstes Heap-Element aus.
 - Letztes Feldelement wird dadurch frei

```

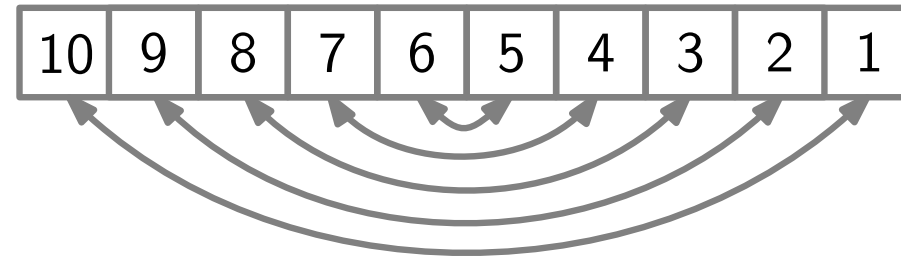
HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  for i = A.length downto 2 do
    | A[i] = EXTRACTMIN()
  for i = 1 to ⌊A.length/2⌋ do
    | A[i] ↔ A[n - i]
  
```

```

HEAPSORT(int[] A)
  BUILDMAXHEAP(A)
  for i = A.length downto 2 do
    | A[i] = EXTRACTMAX()
  
```



Min-Heap



Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✗

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortialgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
INSERTIONSORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SELECTIONSORT	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BUBBLESORT	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MERGESORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HEAPSORT	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗