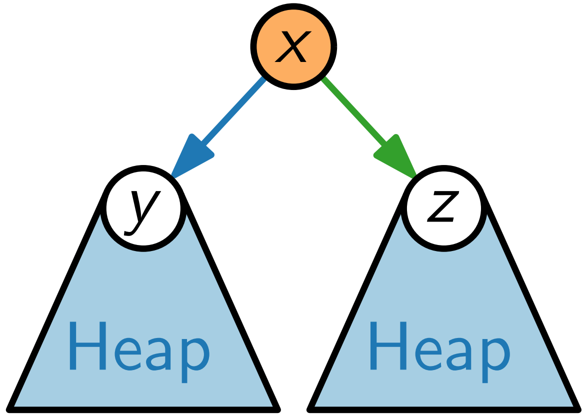
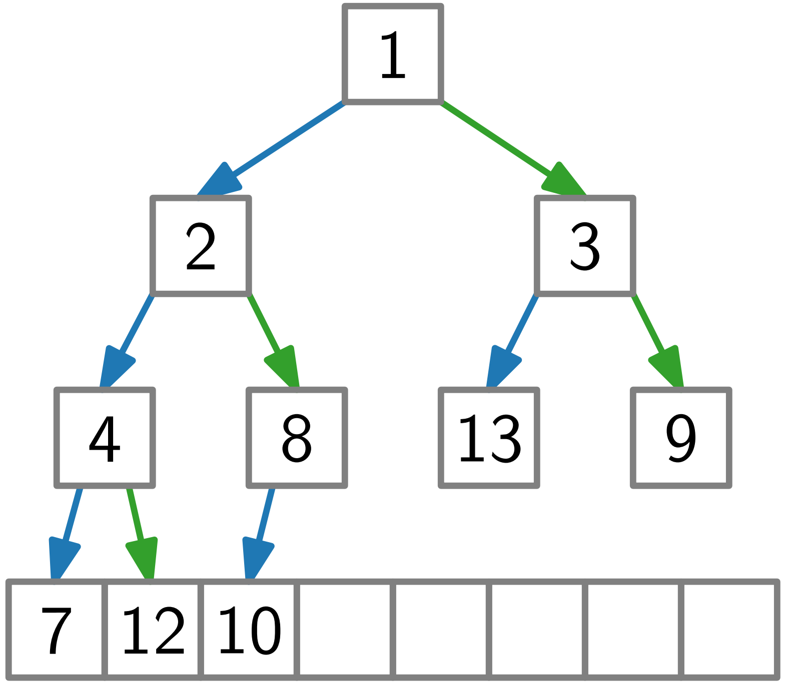
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen



Vorlesung 6:
HeapSort



Alexander Wolff

Wintersemester 2025

Bildquelle: `https://sqlrambling.net/category/sql/heaps`

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

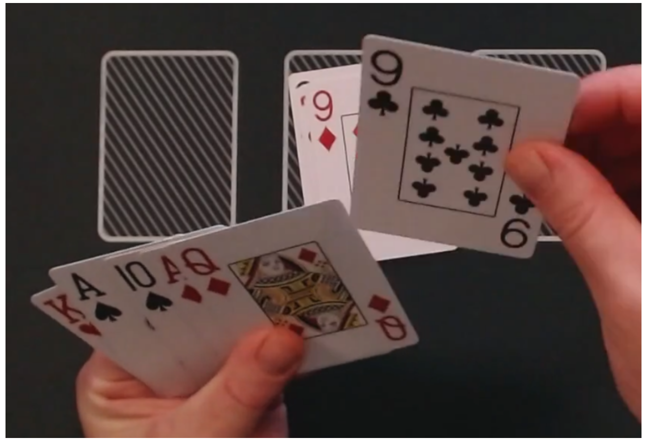
2 mögliche Züge:

(1) Karte ziehen

Ein simples Kartenspiel

2 mögliche Züge:

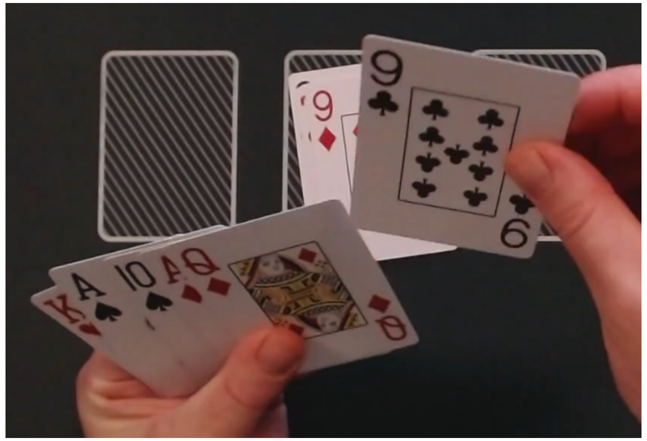
(1) Karte ziehen 

(2) **kleinste** Karte spielen 

Ein simples Kartenspiel

2 mögliche Züge:

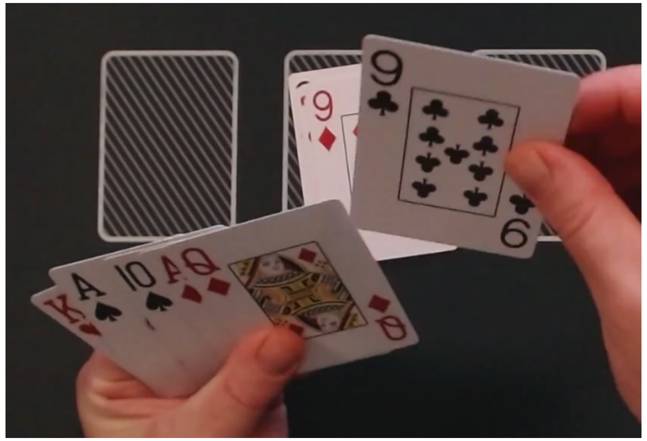
(1) Karte ziehen 

(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?

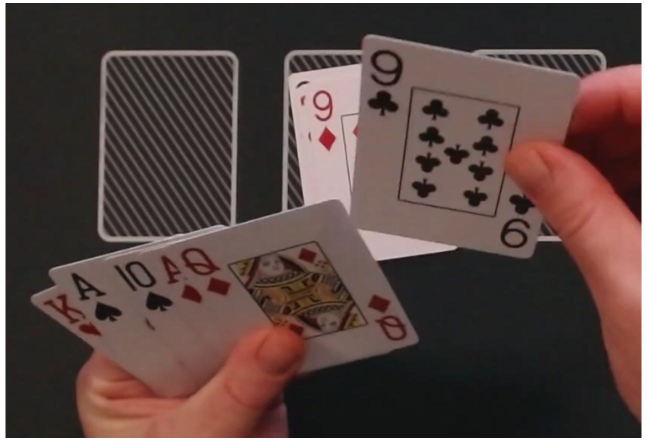
Datenstruktur

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?

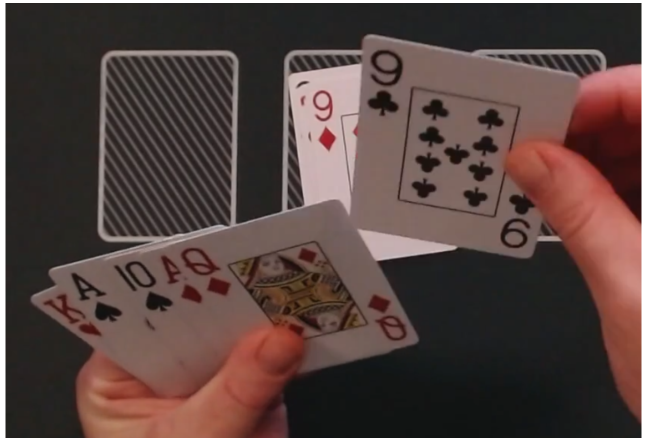
Datenstruktur

Abstrakter Datentyp:

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

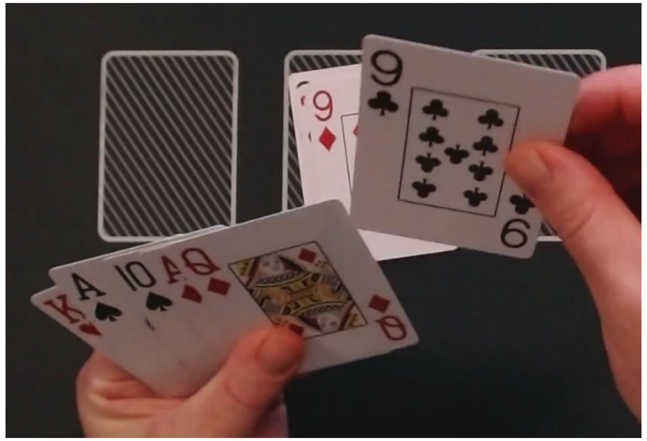
Abstrakter Datentyp: **Prioritätsschlange**

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?

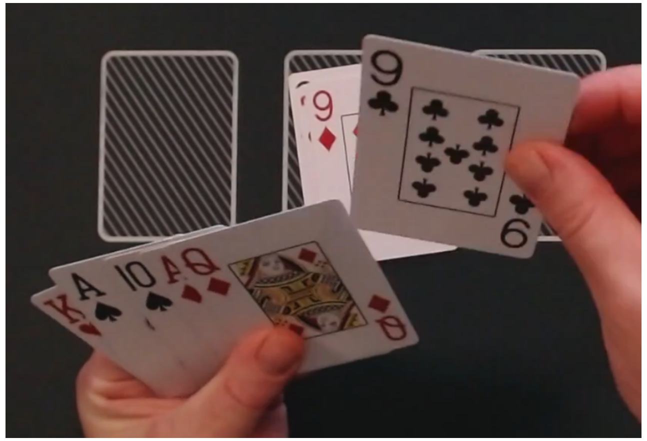
Datenstruktur

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,

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

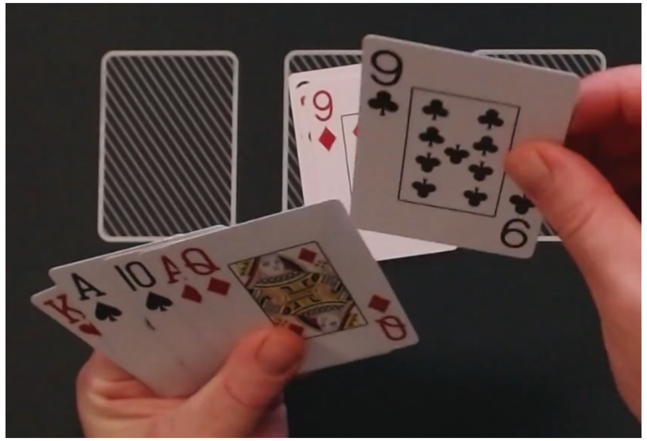
Abstrakter Datentyp: **Prioritätsschlange**

Karten

verwaltet Elemente einer Menge M,

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

Abstrakter Datentyp: **Prioritätsschlange**

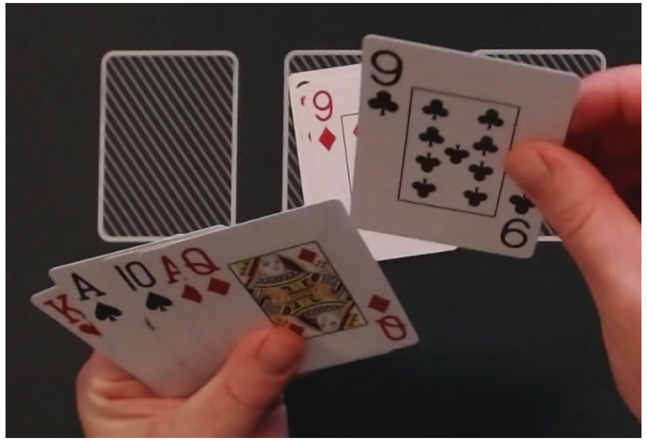
Karten

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

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

Abstrakter Datentyp: **Prioritätsschlange**

Karten

Kartenwert

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

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	**Funktionalität**

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	Funktionalität
$\textsc{Insert}$(element x)	
element $\textsc{FindMin}()$	
element $\textsc{ExtractMin}()$	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	**Funktionalität**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	
element $\textsc{ExtractMin}()$	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	**Funktionalität**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element $\textsc{ExtractMin}()$	

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	**Funktionalität**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	liefere $x \in M$ mit: $x.\mathit{key} = \min\{y.\mathit{key} \mid y \in M\}$
element $\textsc{ExtractMin}()$	$x = \textsc{FindMin}()$; $M = M \setminus \{x\}$; 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:

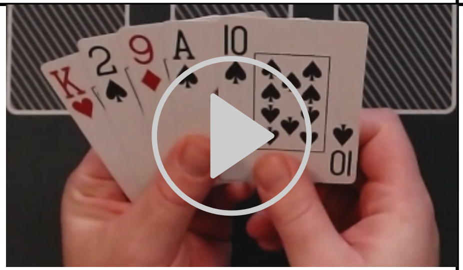
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

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

Implementierung

Aufgabe:

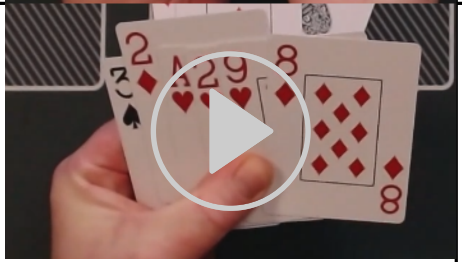
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

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

Implementierung

Aufgabe:

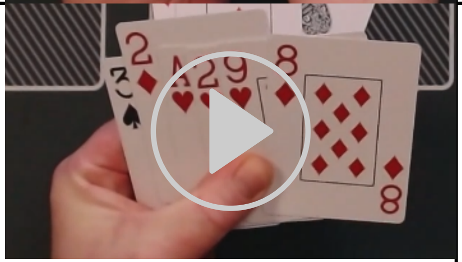
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

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

Implementierung

Aufgabe:

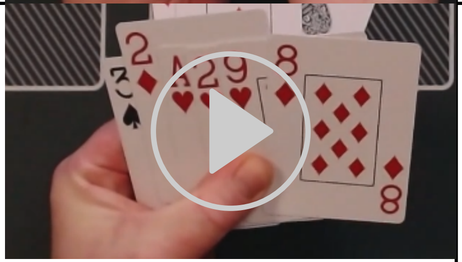
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

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

Implementierung

Aufgabe:

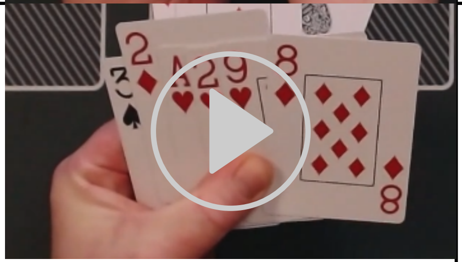
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

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:

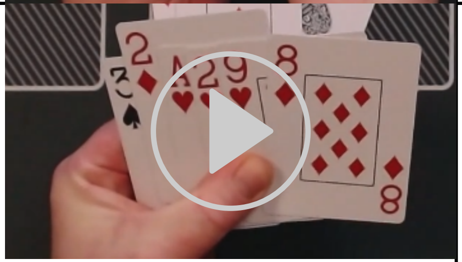
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

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?

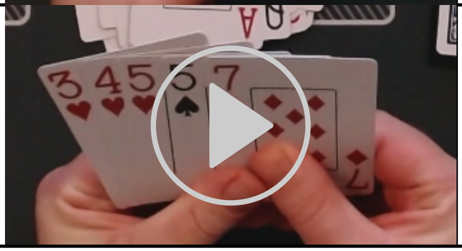
Implementierung		INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge		$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert		$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

müssen wieder kleinste Karte finden

Implementierung

Aufgabe:

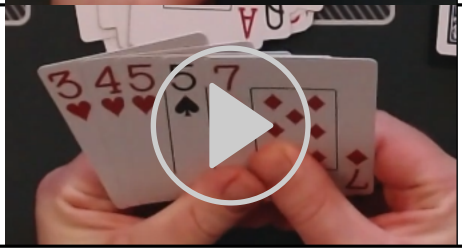
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

Implementierung		Insert	FindMin	ExtractMin
Karten in beliebiger Reihenfolge		$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert		$\Theta(1)$	$\Theta(1)$	müssen wieder kleinste Karte finden $\Theta(n)$
Karten in **sortierter** Reihenfolge				

Implementierung

Aufgabe:

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

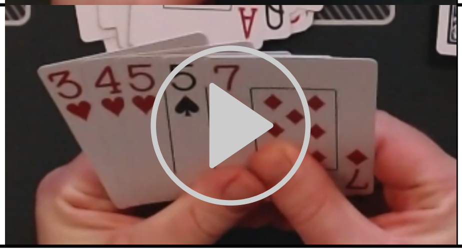
Implementierung		INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge		$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert		$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten in **sortierter** Reihenfolge		$\Theta(n)$		

müssen wieder kleinste Karte finden

Implementierung

Aufgabe:

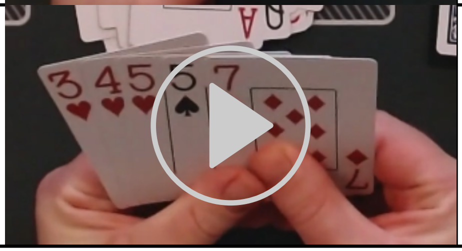
- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

Implementierung		INSERT	FINDMIN	EXTRACTMIN
Karten in beliebiger Reihenfolge		$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert		$\Theta(1)$	$\Theta(1)$	$\Theta(n)$ müssen wieder kleinste Karte finden
Karten in **sortierter** Reihenfolge		$\Theta(n)$	$\Theta(1)$	

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		$\Theta(1)$	$\Theta(1)$	$\Theta(n)$ müssen wieder kleinste Karte finden
Karten in **sortierter** Reihenfolge		$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

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

Operation	**Funktionalität**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element $\textsc{ExtractMin}()$	$x = \text{FindMin}();\ M = M \setminus \{x\};$ liefere x

Prioritätsschlange

Abstrakter Datentyp: **Prioritätsschlange**

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

Operation	**Funktionalität**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element $\textsc{ExtractMin}()$	$x = \text{FindMin}();\ M = M \setminus \{x\};$ liefere x
$\textsc{DecreaseKey}$ (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**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element $\textsc{ExtractMin}()$	$x = \text{FindMin}();\ M = M \setminus \{x\};$ liefere x
$\textsc{DecreaseKey}$ (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				

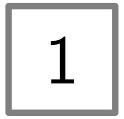
Implementierung

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	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

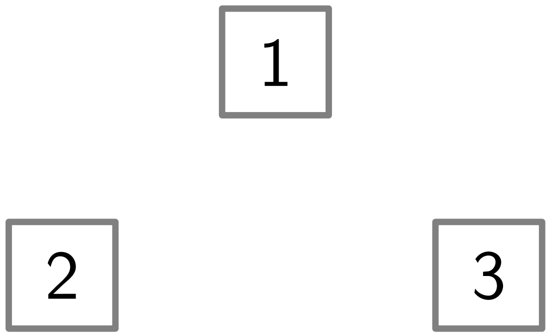
Implementierung

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	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

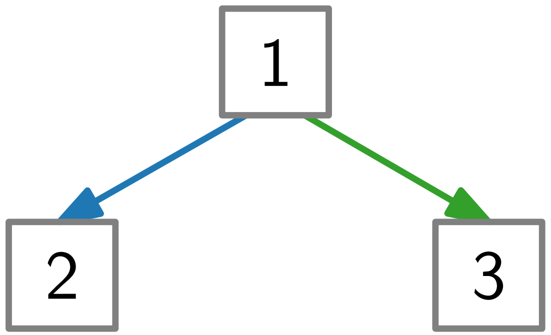
Min-Heaps



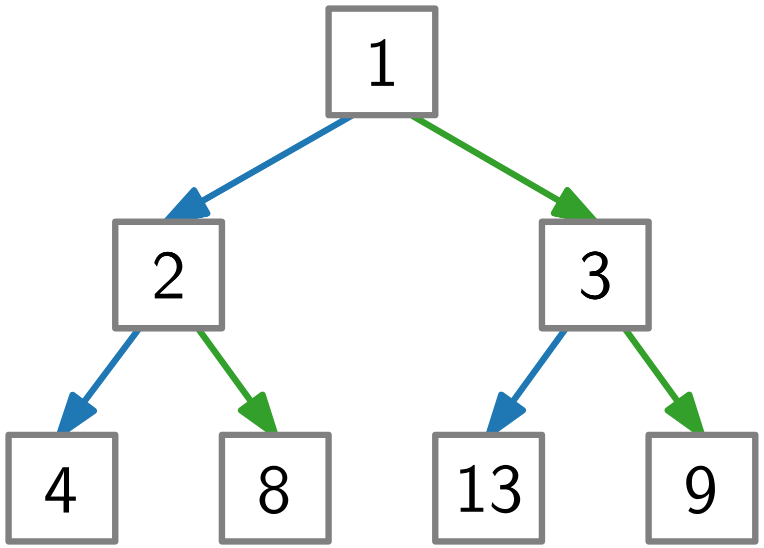
Min-Heaps



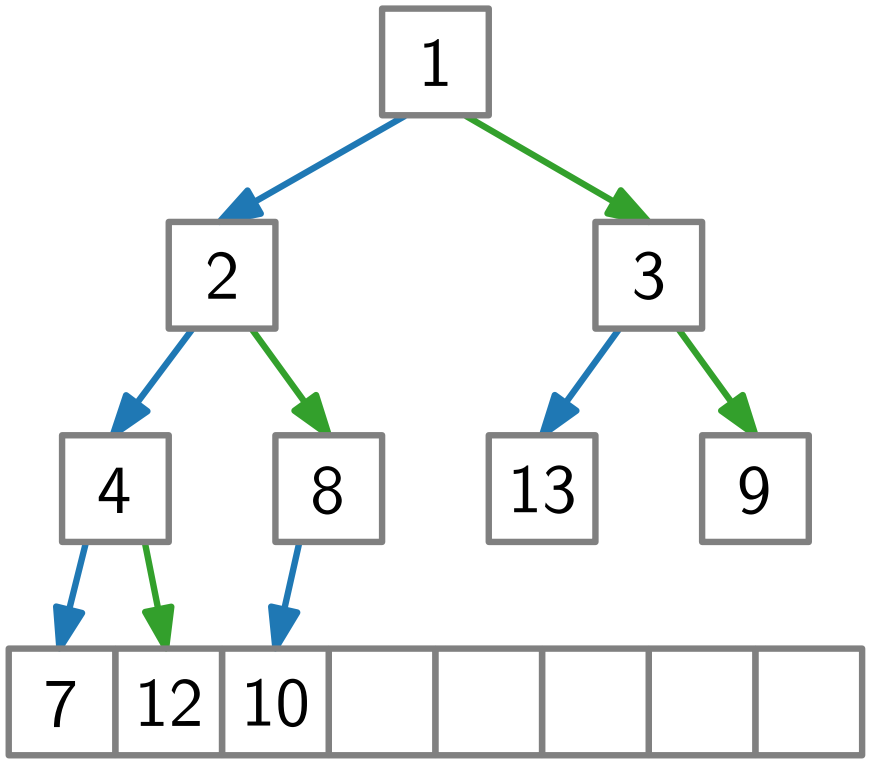
Min-Heaps



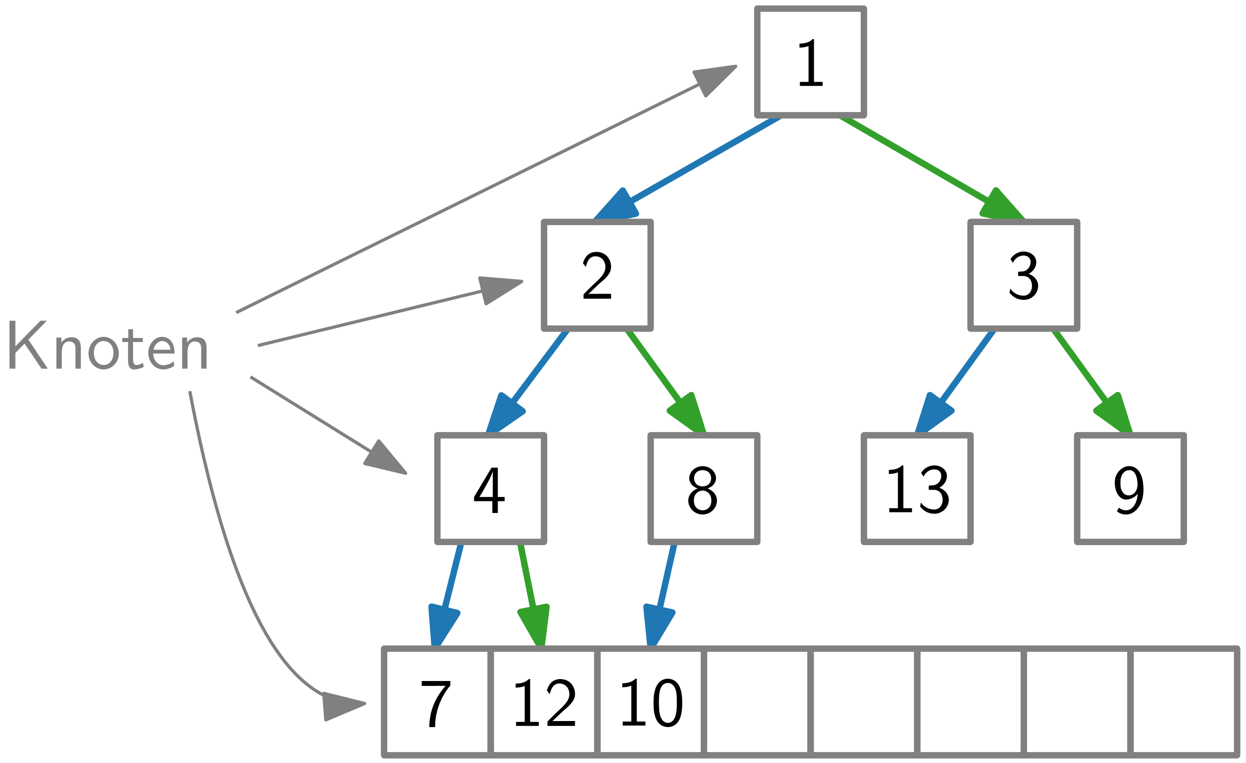
Min-Heaps



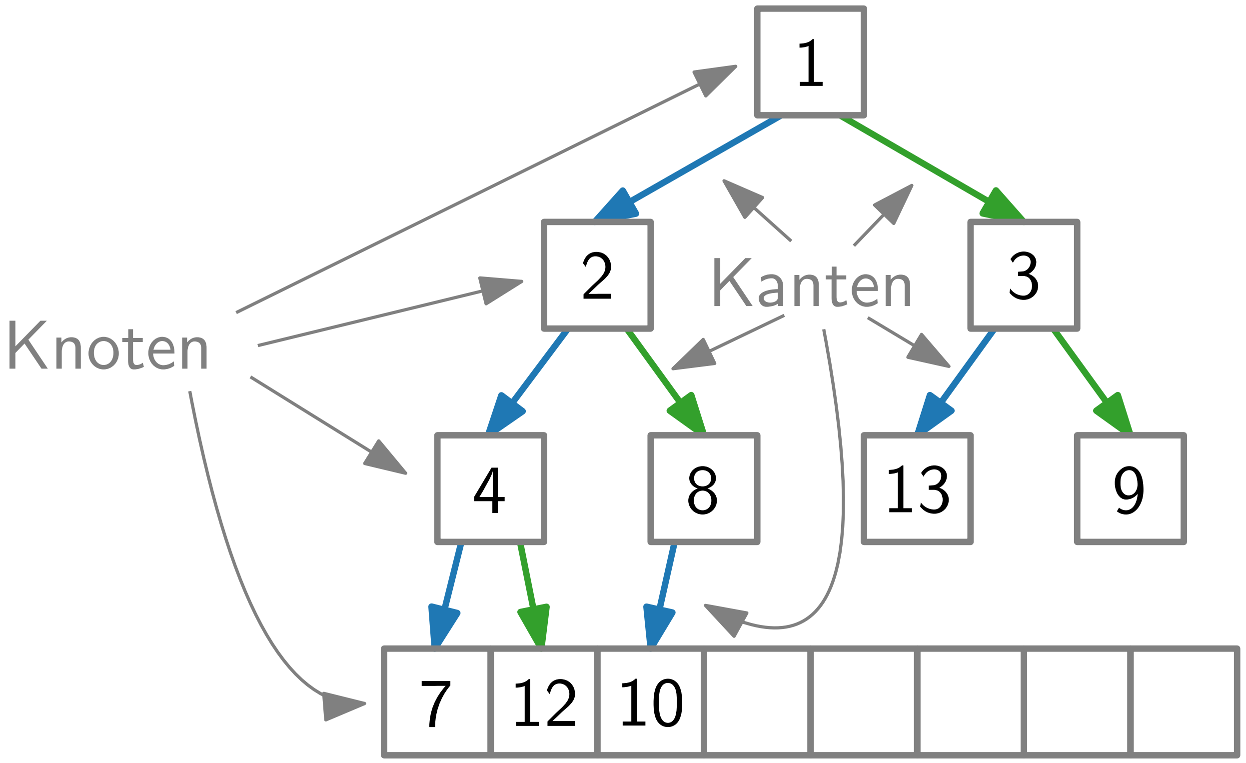
Min-Heaps



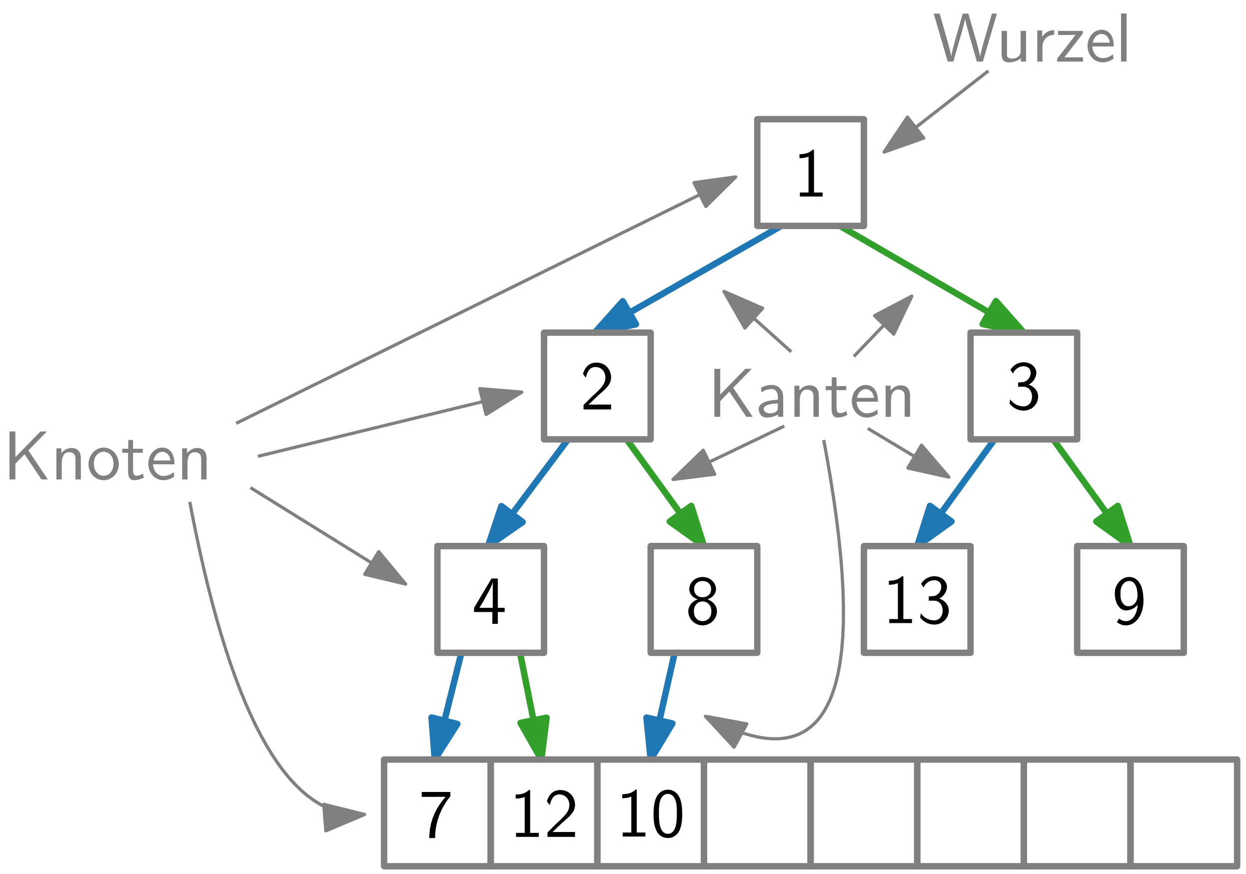
Min-Heaps



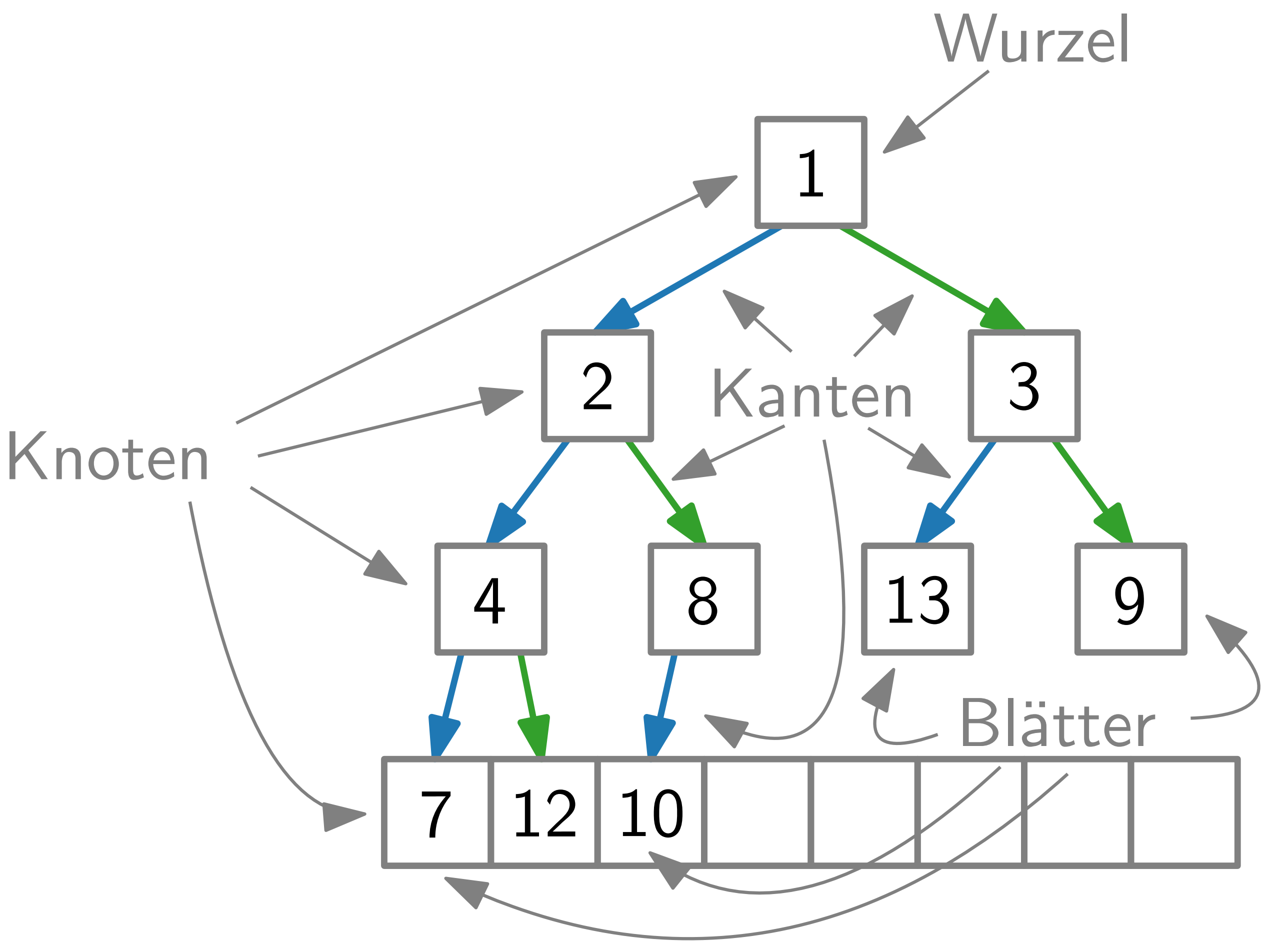
Min-Heaps



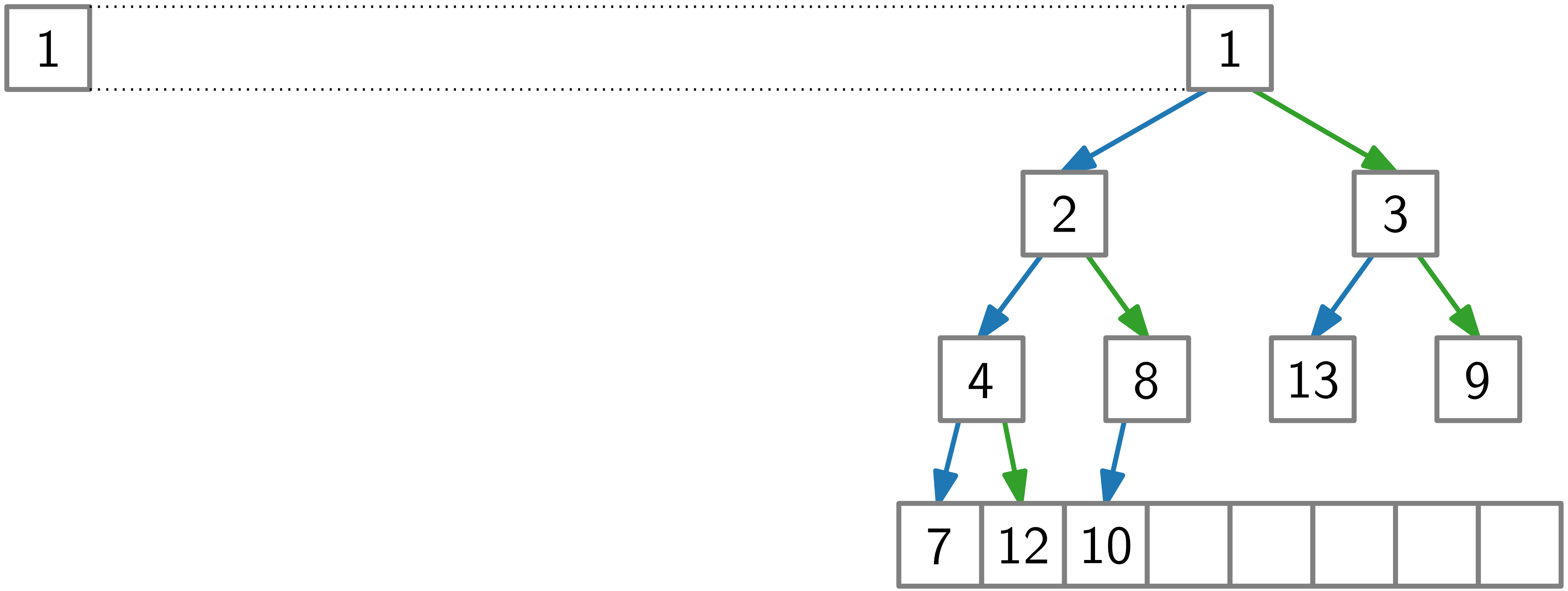
Min-Heaps



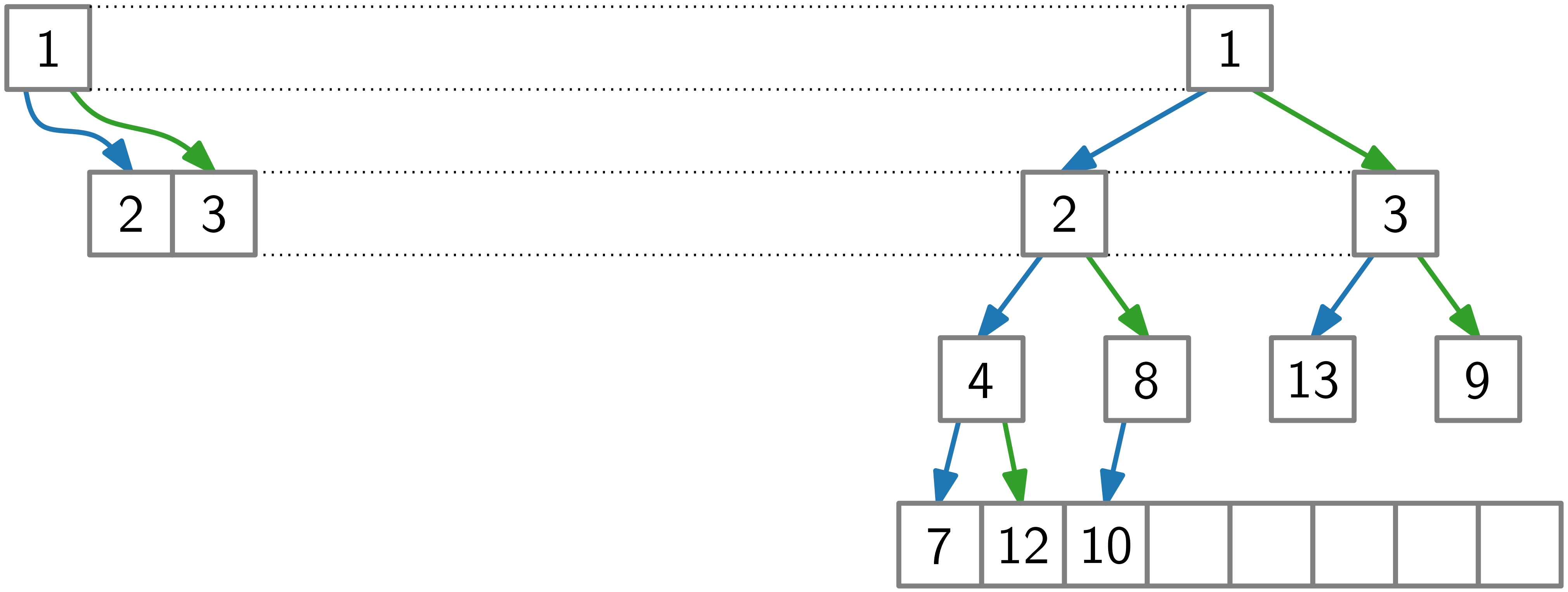
Min-Heaps



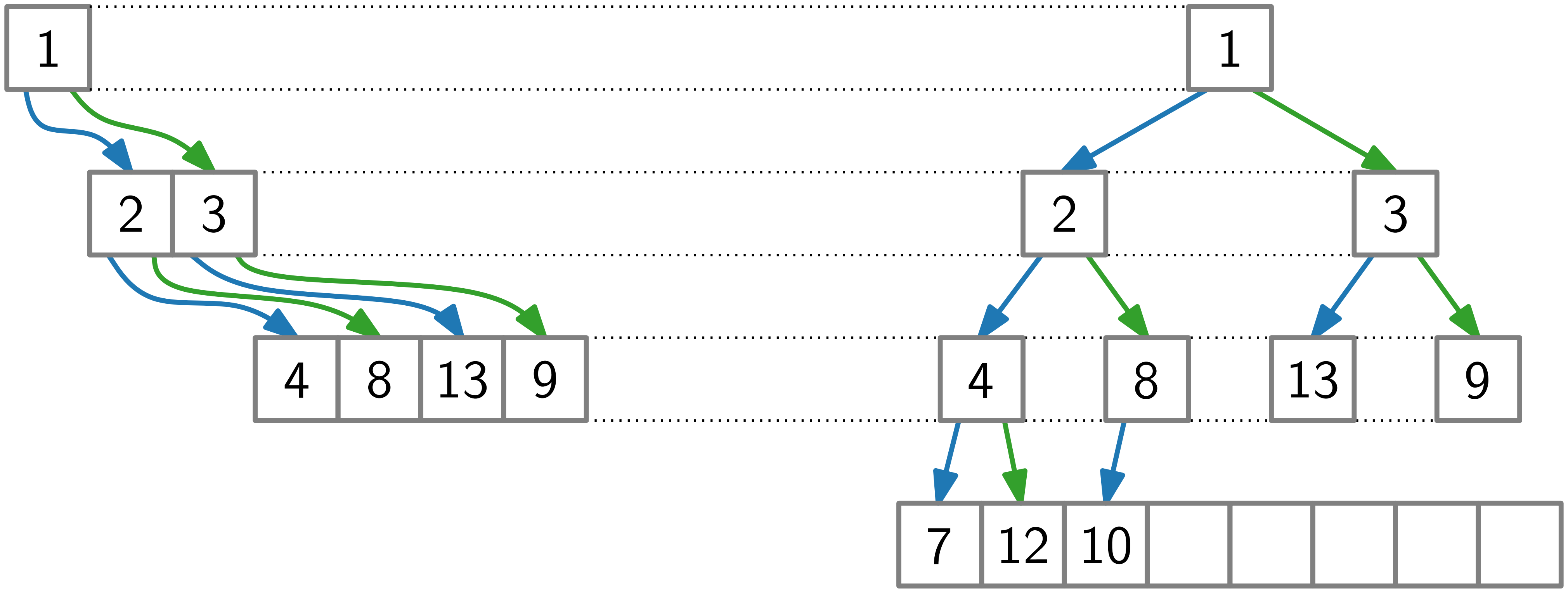
Min-Heaps



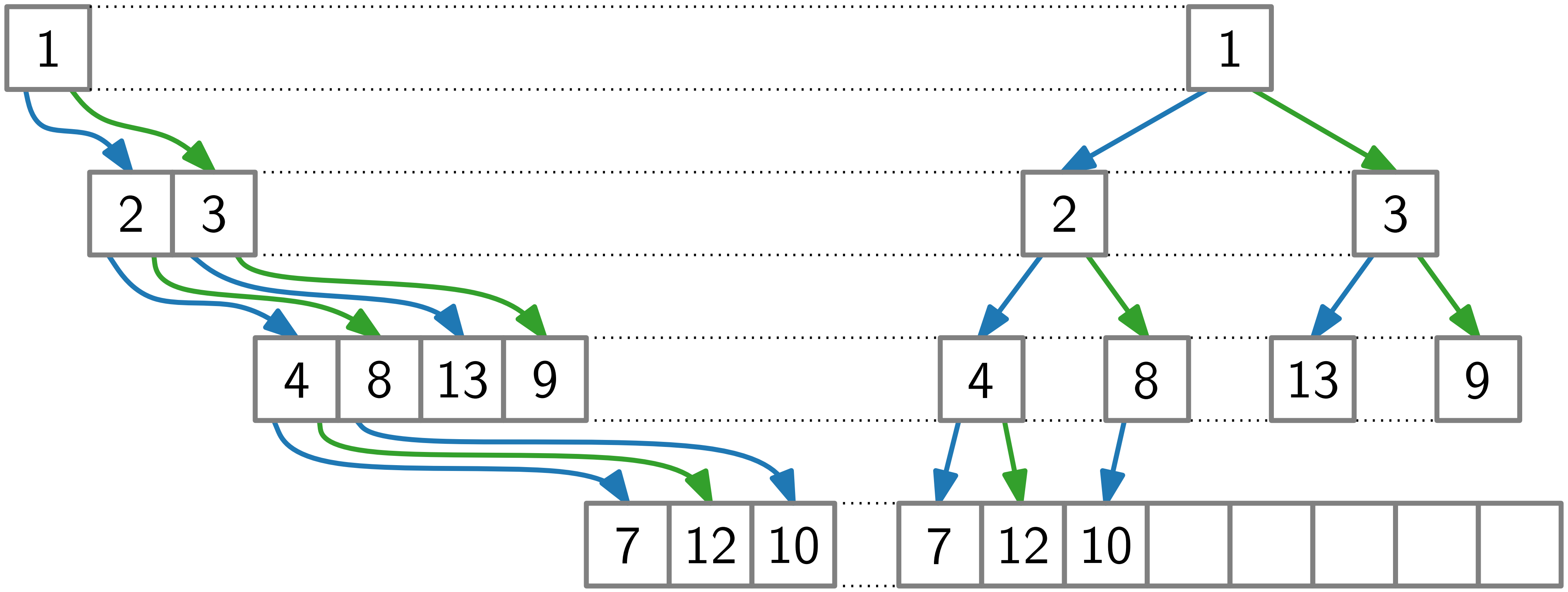
Min-Heaps



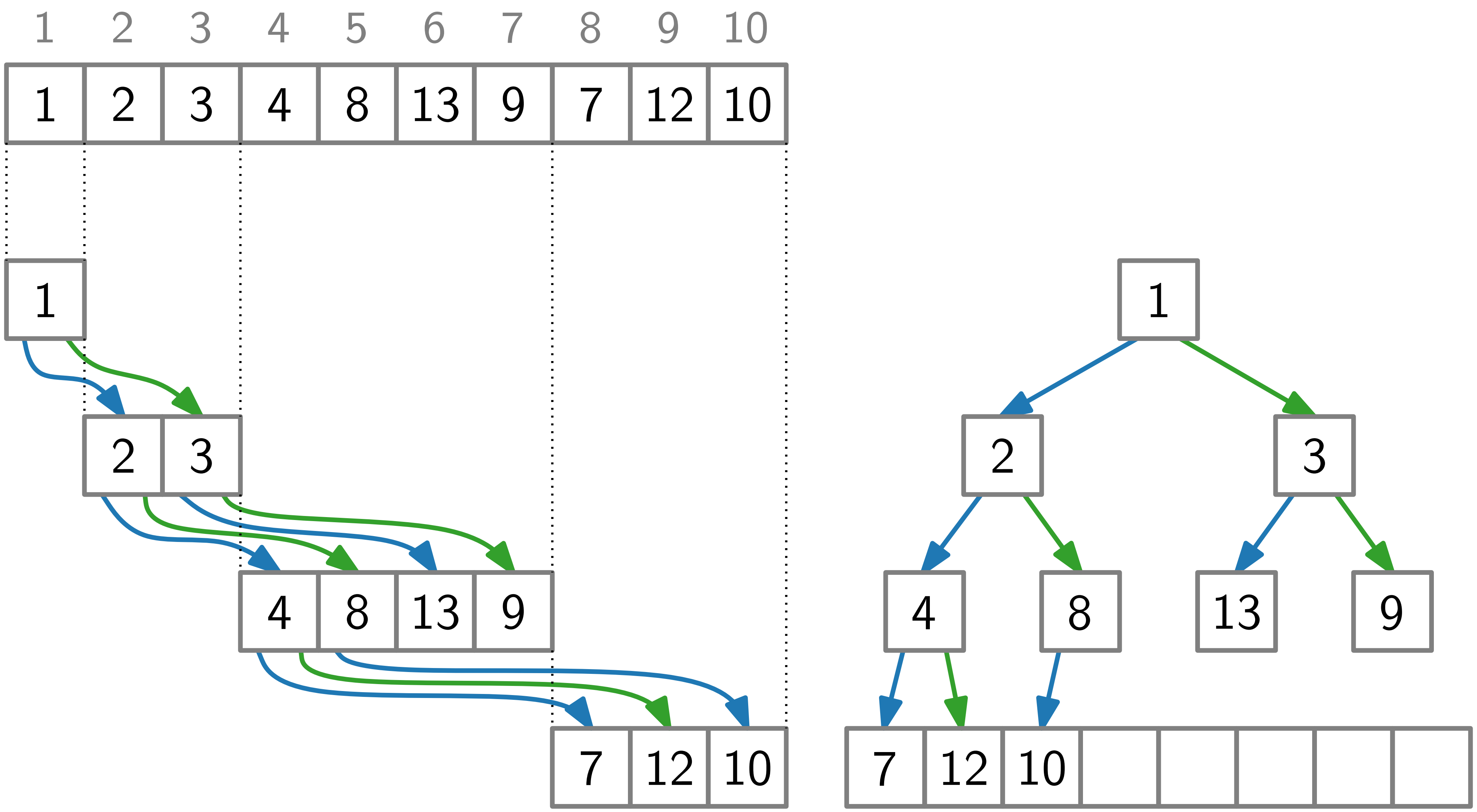
Min-Heaps



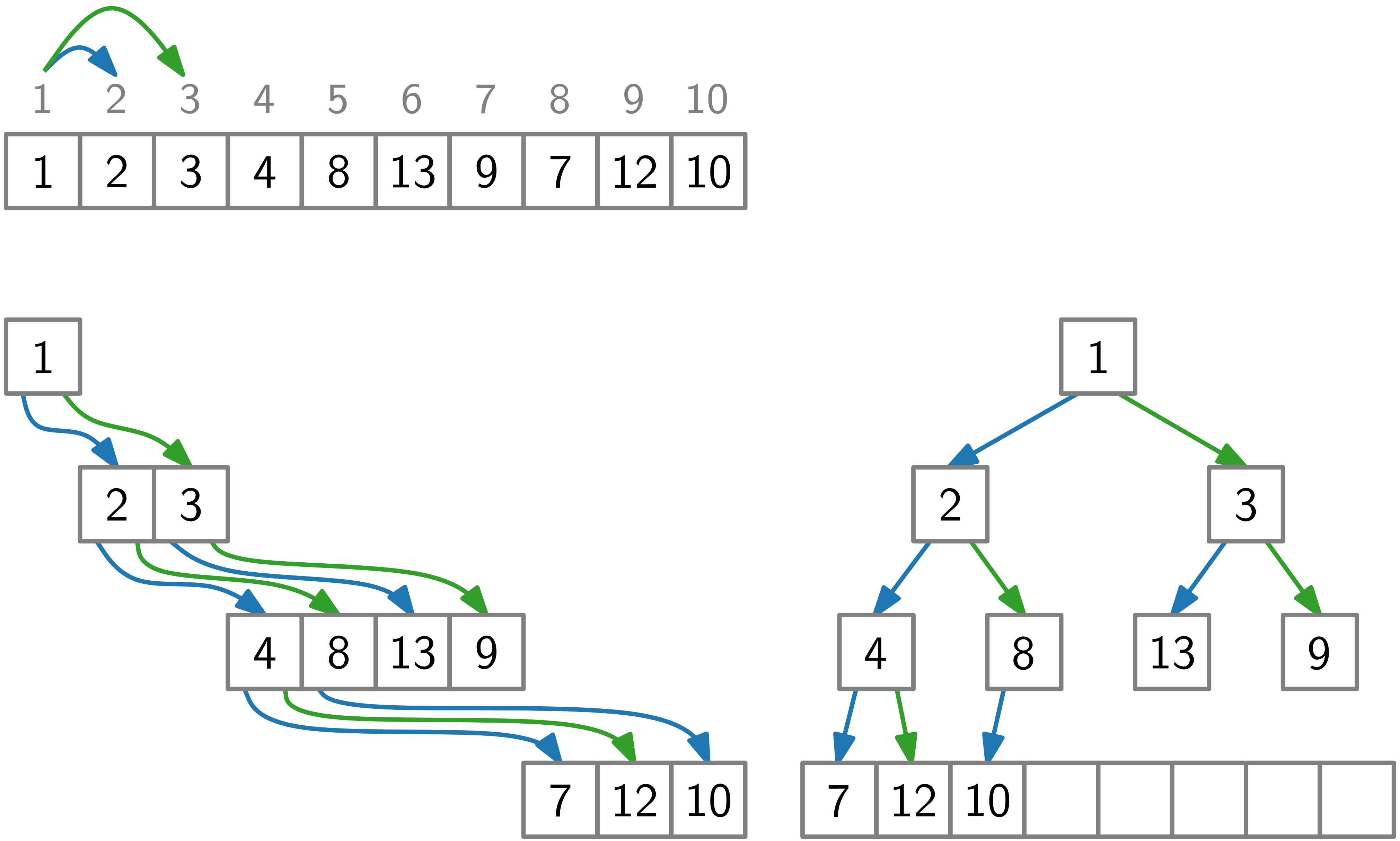
Min-Heaps



Min-Heaps

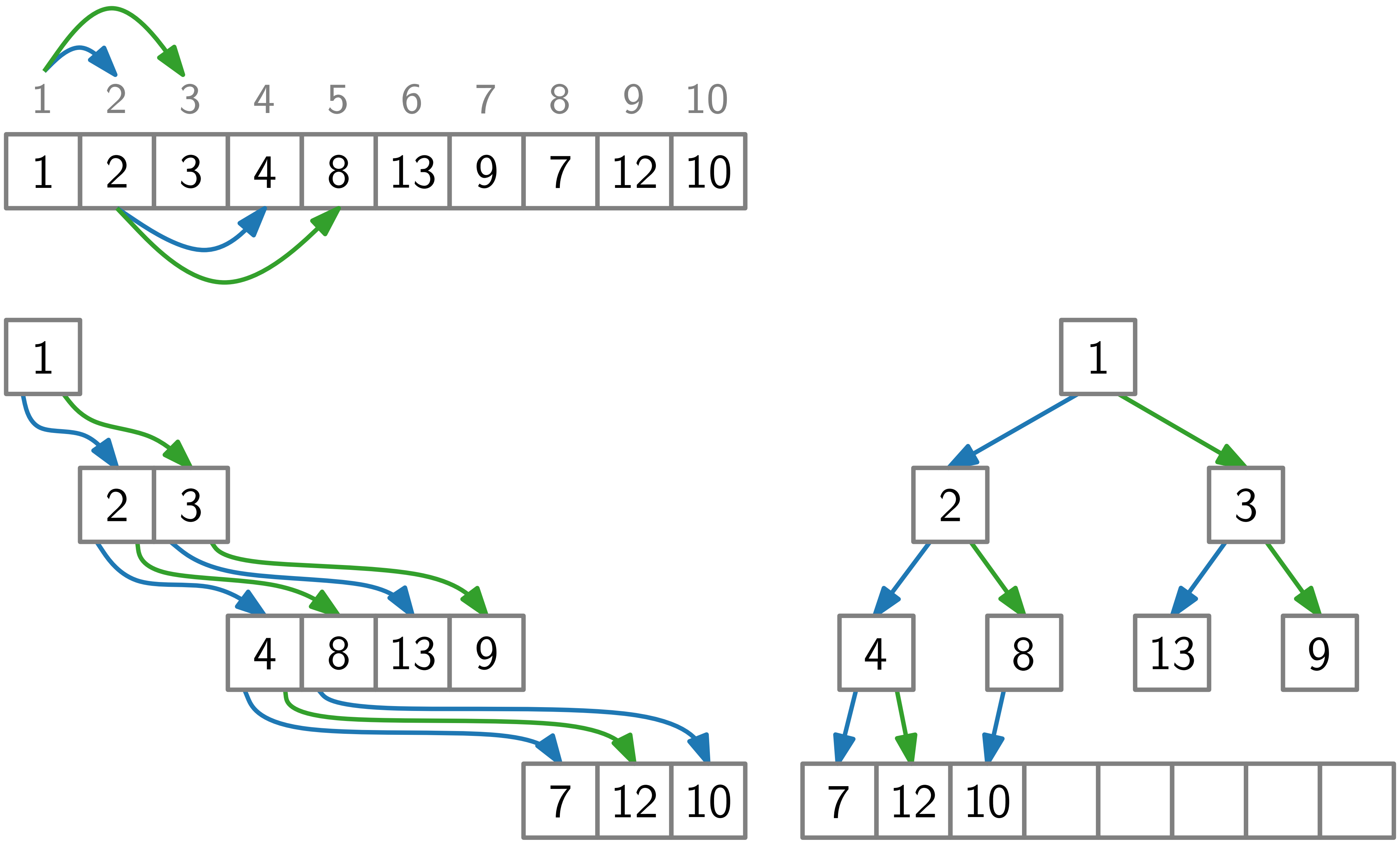

1 2 3 4 5 6 7 8 9 10
1 2 3 4 8 13 9 7 12 10
1
2 3
4 8 13 9
7 12 10
1
2
3
4
8
13
9
7 12 10

Min-Heaps

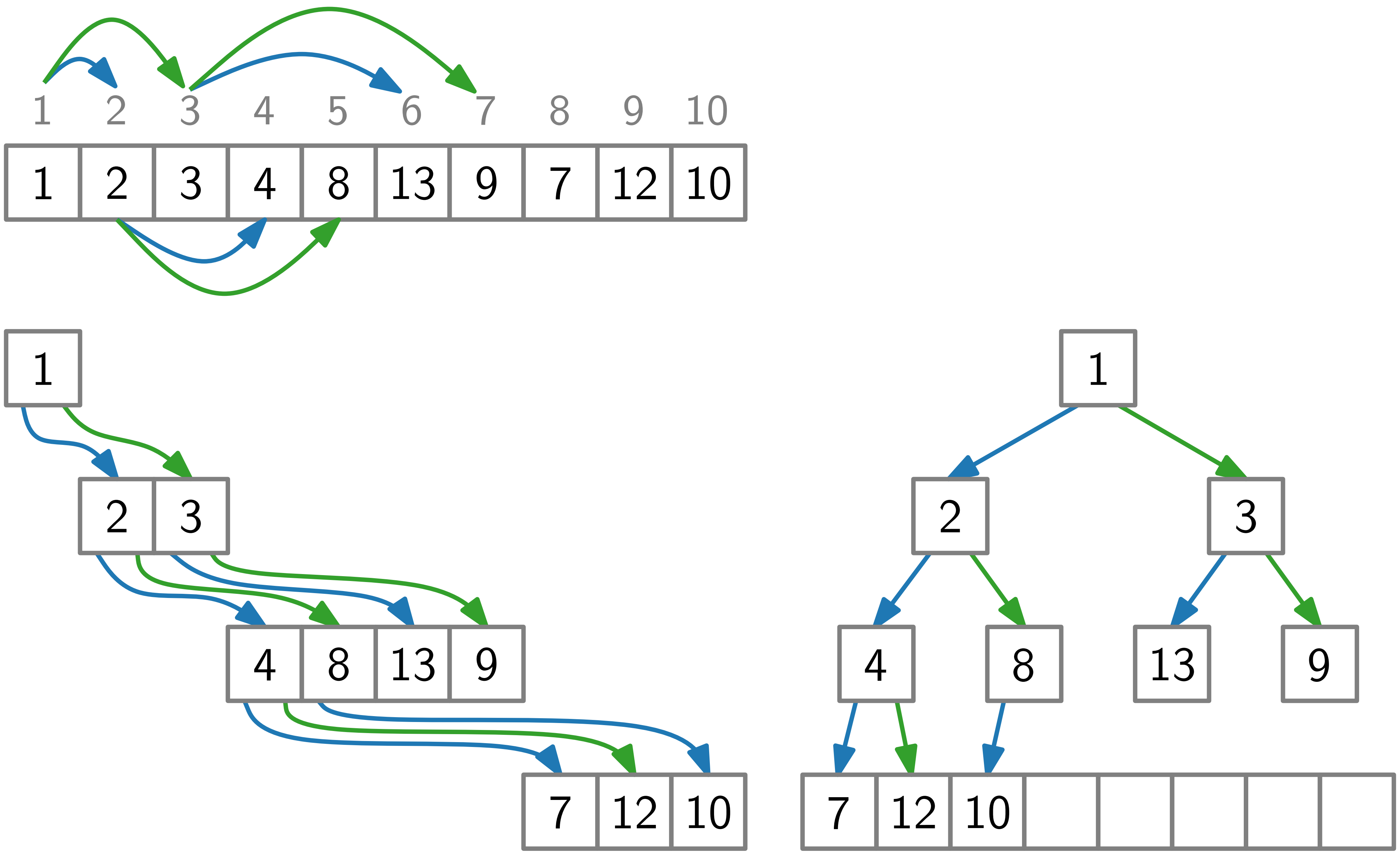


1 2 3 4 5 6 7 8 9 10
1 2 3 4 8 13 9 7 12 10
1
2 3
4 8 13 9
7 12 10
1
2
3
4
8
13
9
7 12 10

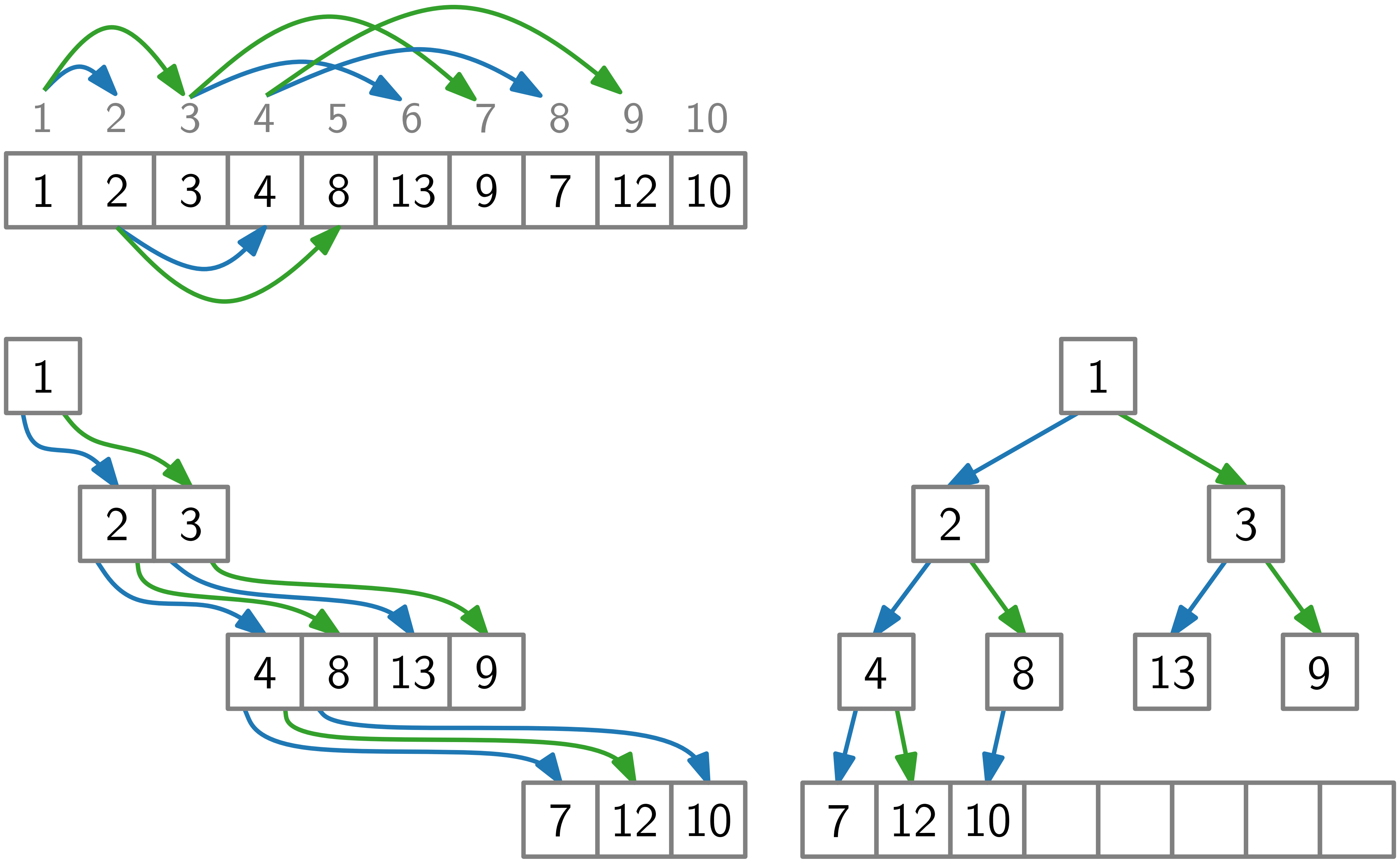
Min-Heaps


1 2 3 4 5 6 7 8 9 10
1 2 3 4 8 13 9 7 12 10
1
2 3
4 8 13 9
7 12 10
1
2
3
4
8
13
9
7 12 10

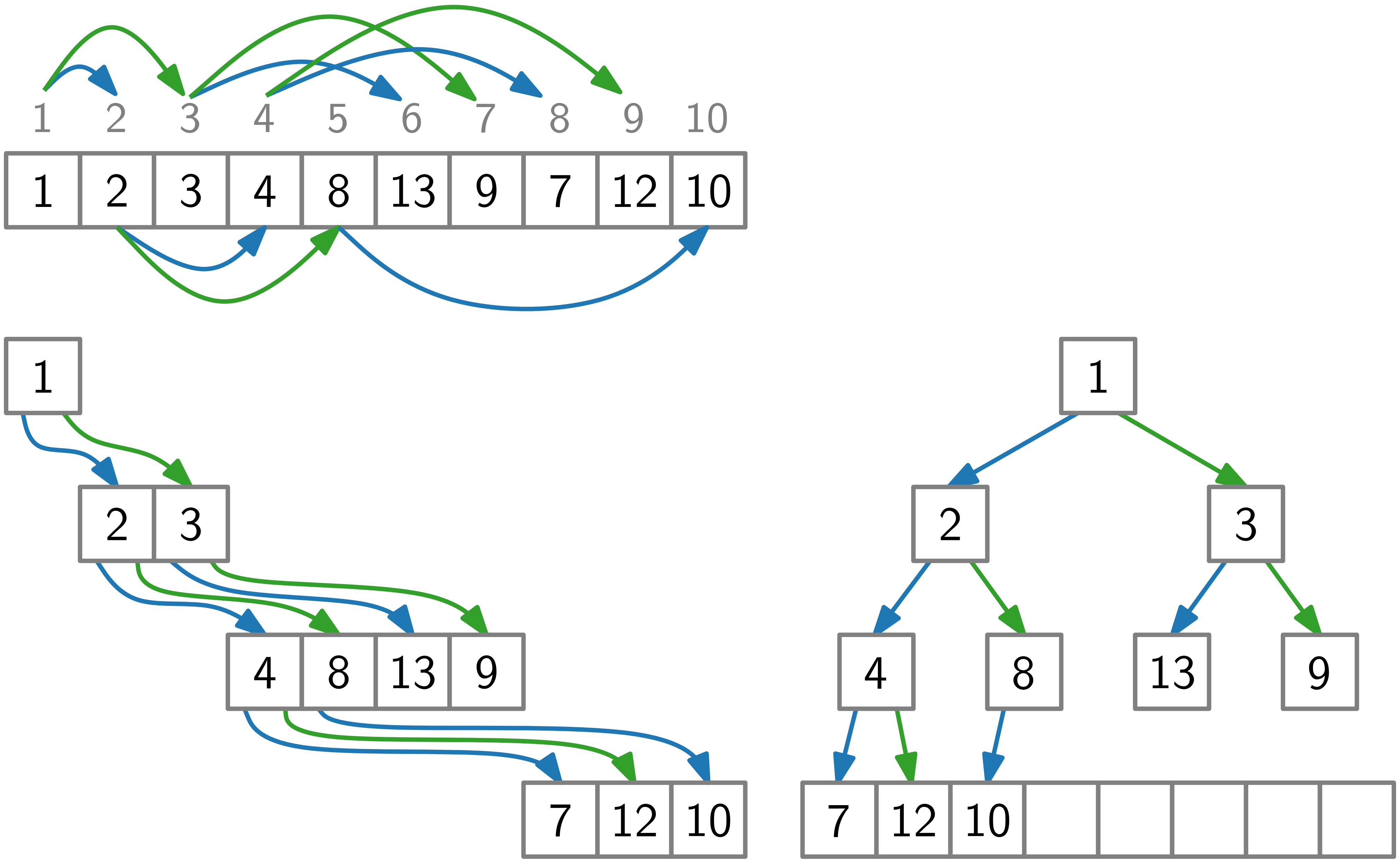
Min-Heaps


1 2 3 4 5 6 7 8 9 10
1 2 3 4 8 13 9 7 12 10
1
2 3
4 8 13 9
7 12 10
1
2
3
4
8
13
9
7 12 10

Min-Heaps

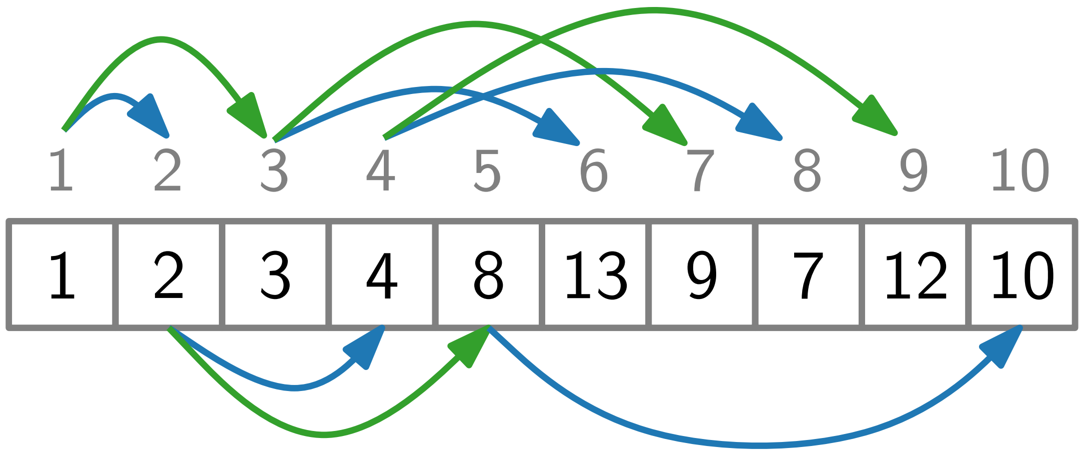

1 2 3 4 5 6 7 8 9 10
1 2 3 4 8 13 9 7 12 10
1
2 3
4 8 13 9
7 12 10
1
2 3
4 8 13 9
7 12 10

Min-Heaps

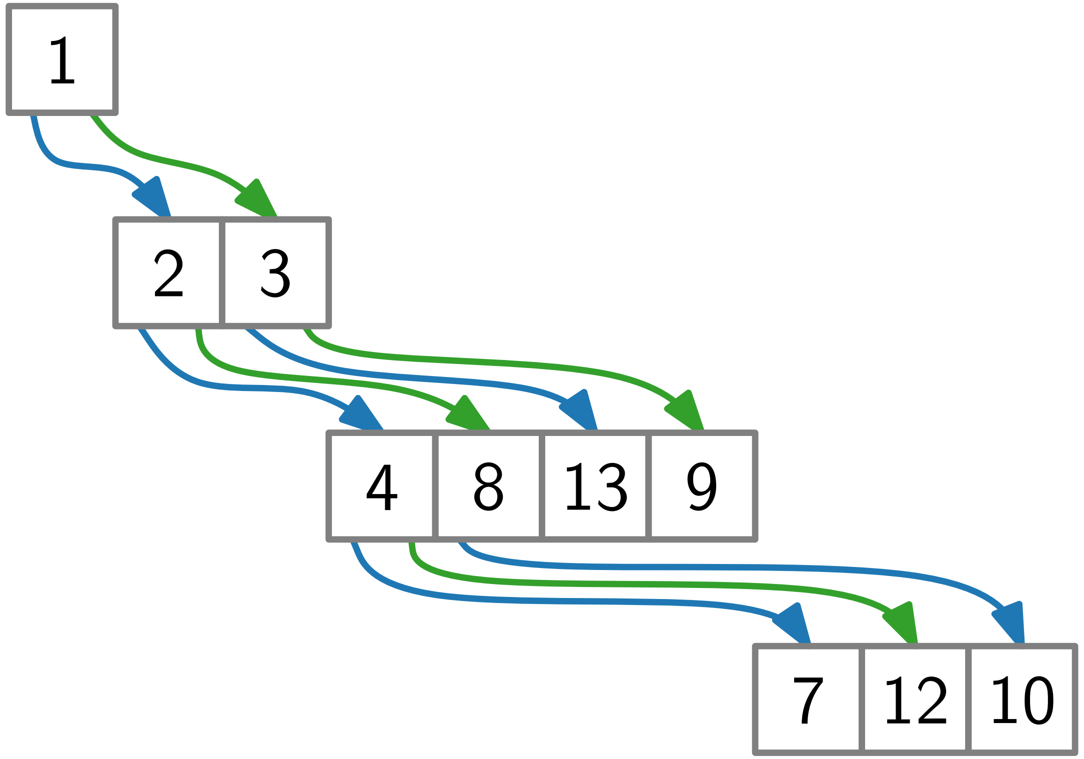


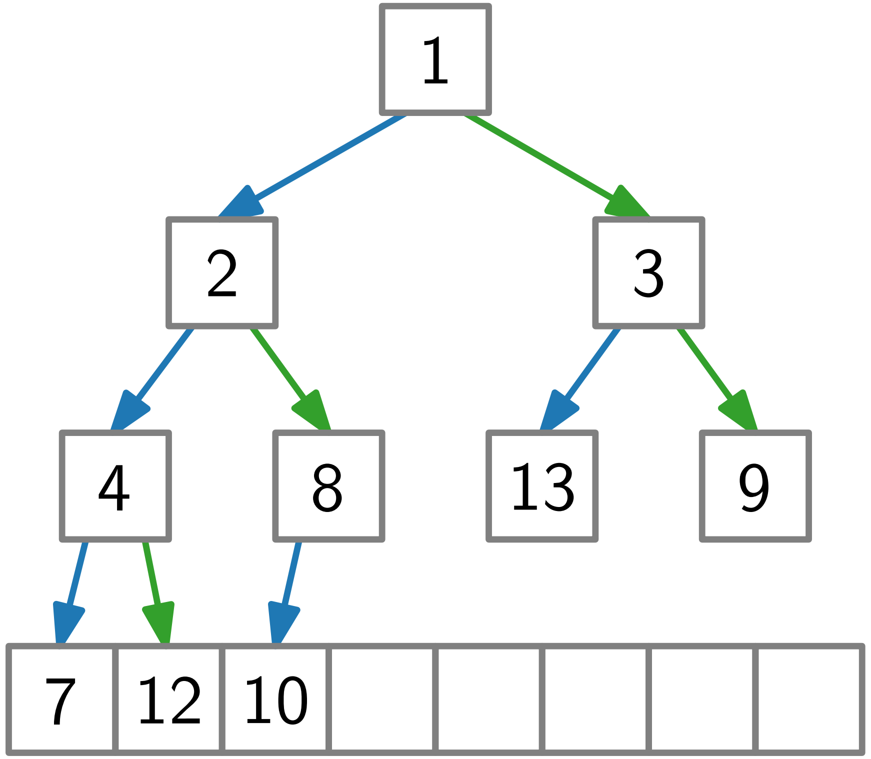
1 2 3 4 5 6 7 8 9 10
1 2 3 4 8 13 9 7 12 10
1
2 3
4 8 13 9
7 12 10
1
2
3
4
8
13
9
7 12 10

Min-Heaps

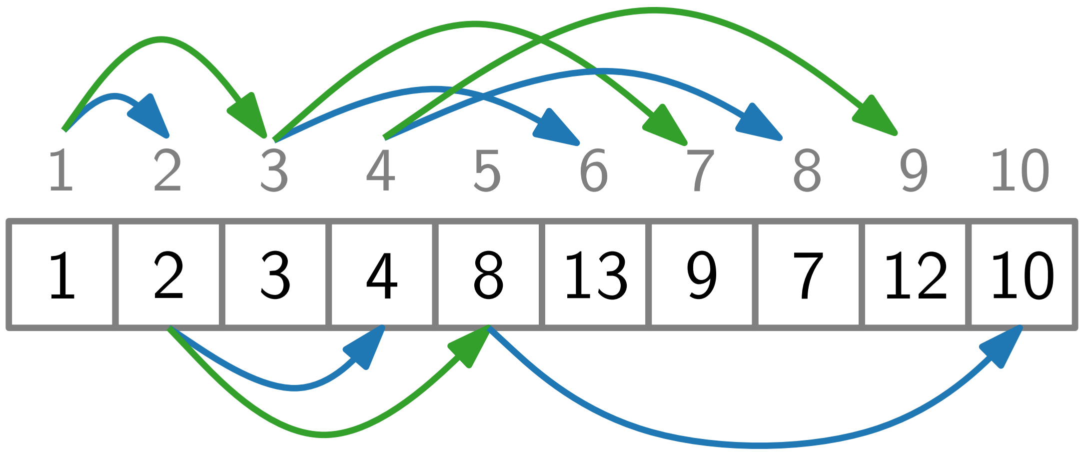


Pfeile implementieren:





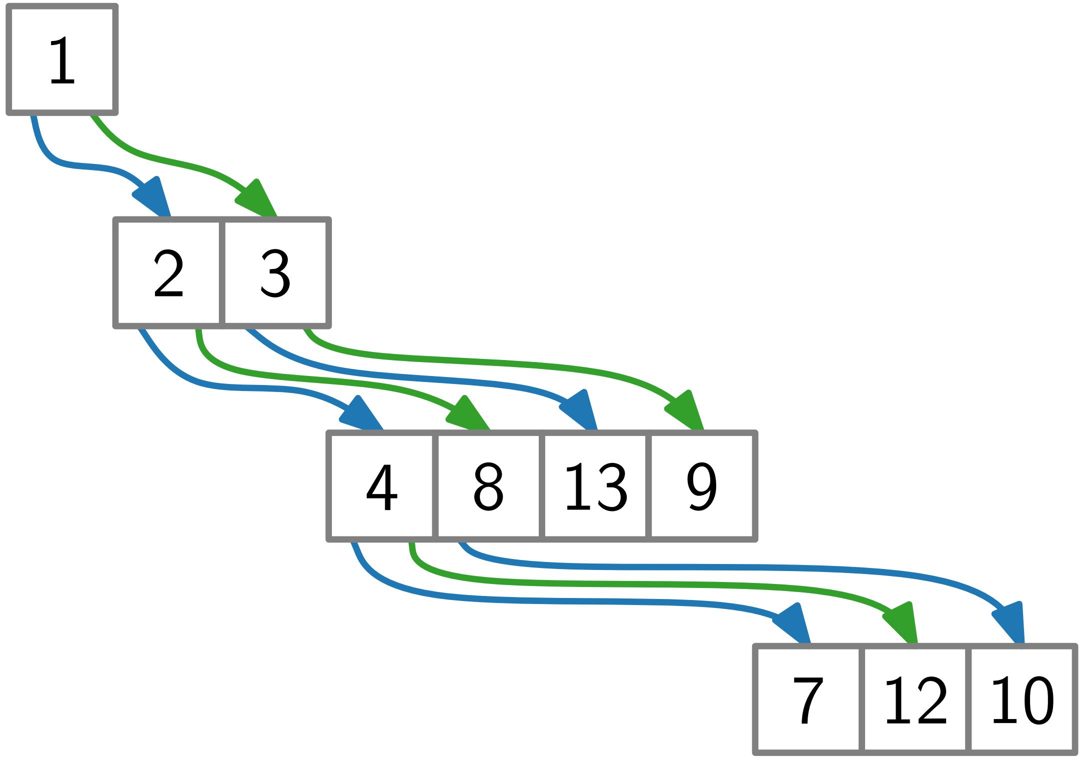
Min-Heaps

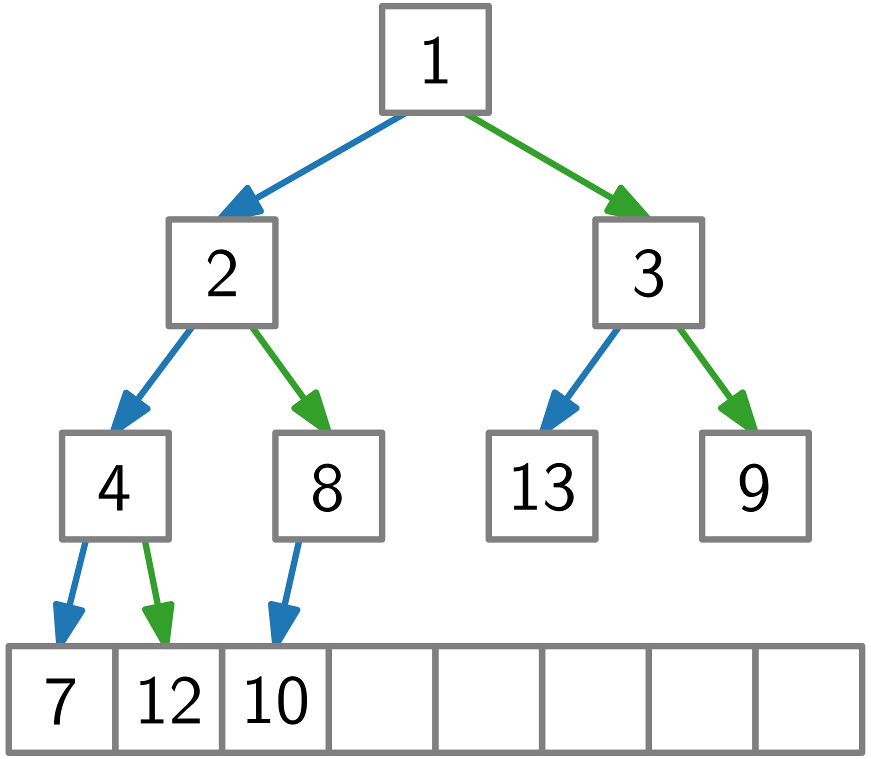


Pfeile implementieren:

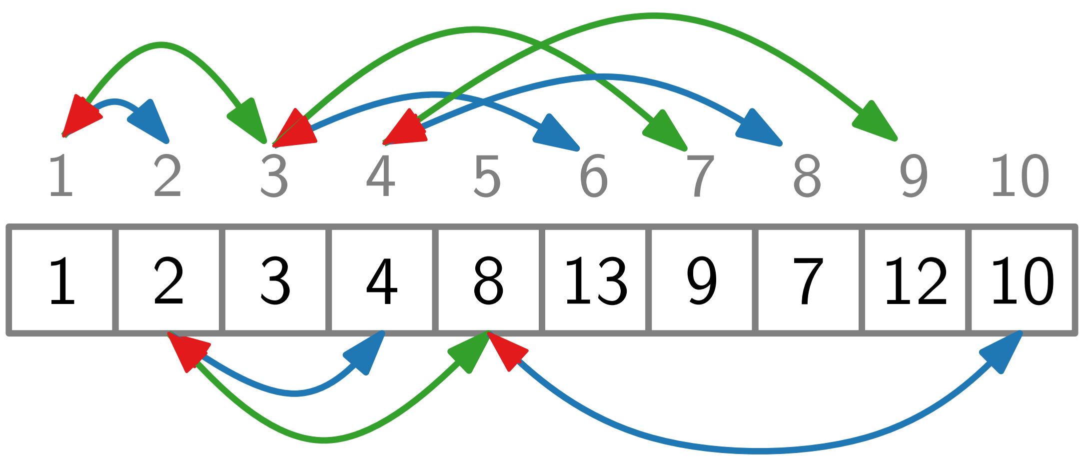
LEFT(index i) **return** . . .

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





Min-Heaps

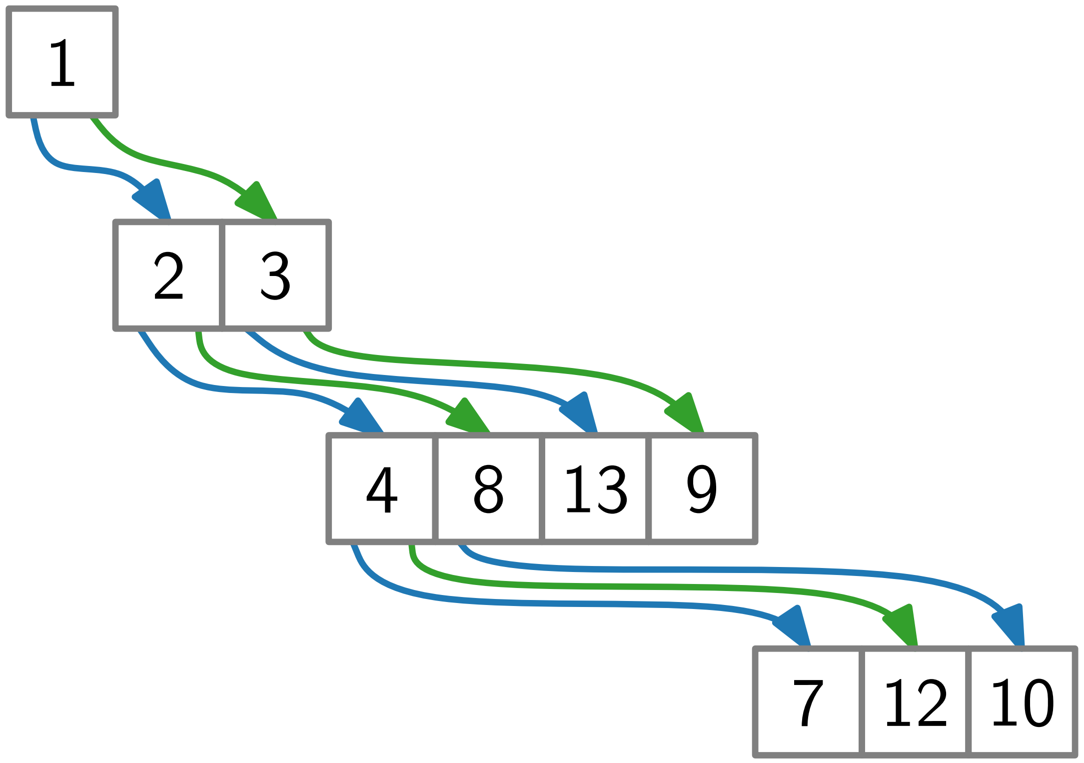


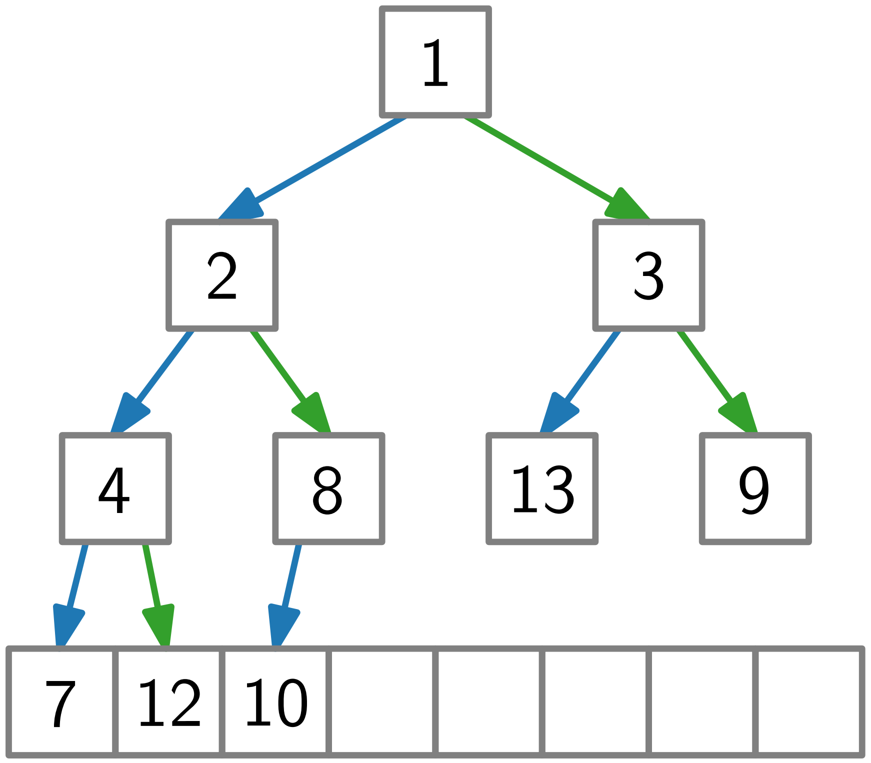
Pfeile implementieren:

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

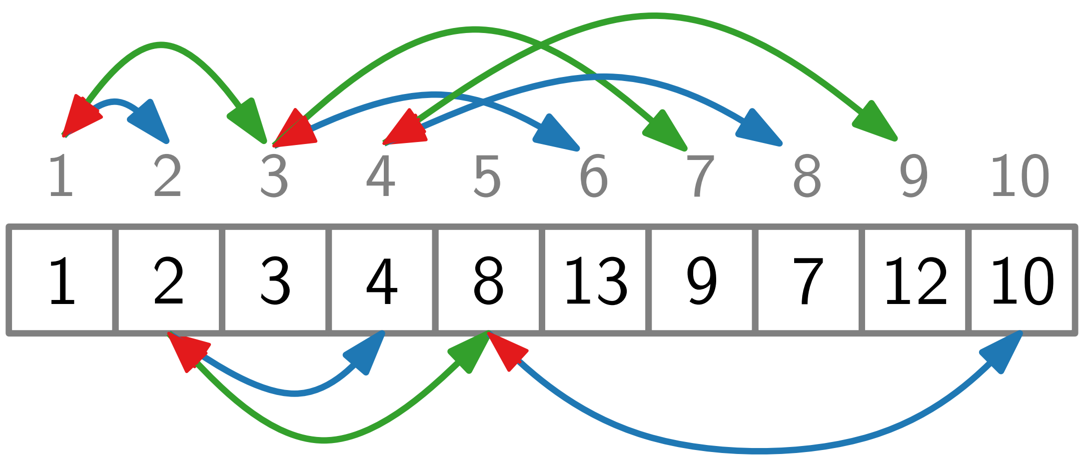
RIGHT(index i) **return** ...

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





Min-Heaps

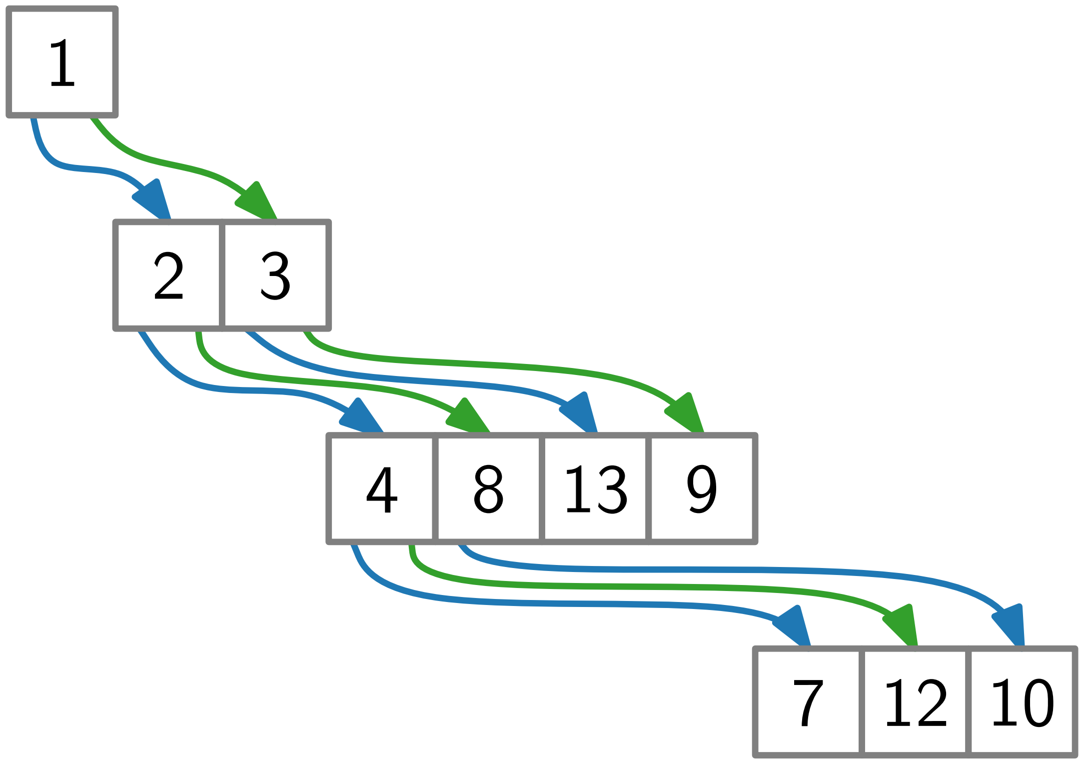


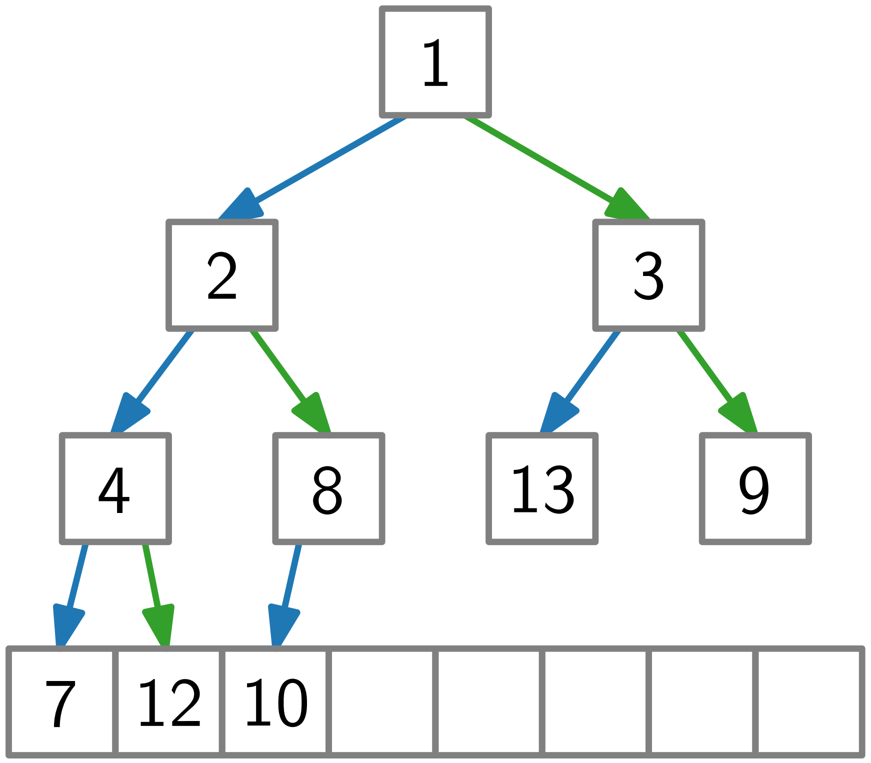
Pfeile implementieren:

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

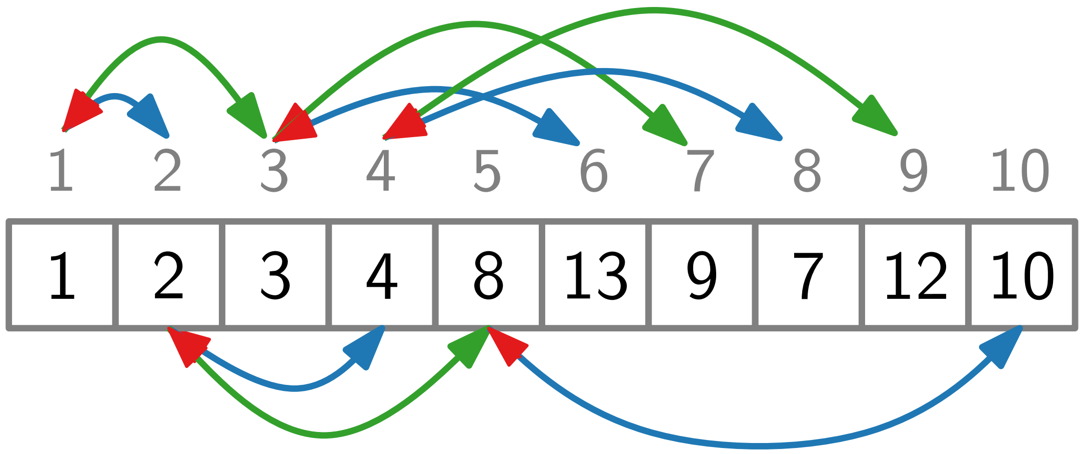
RIGHT(index i) **return** $\ldots$

PARENT(index i) **return** $\ldots$



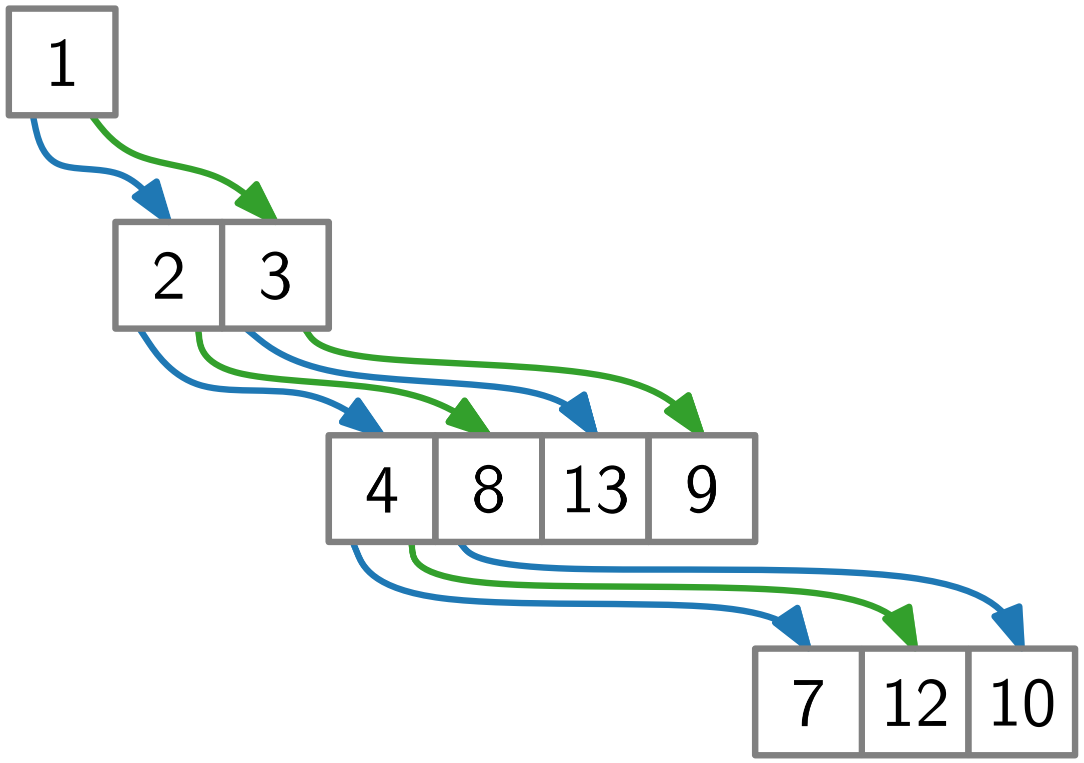


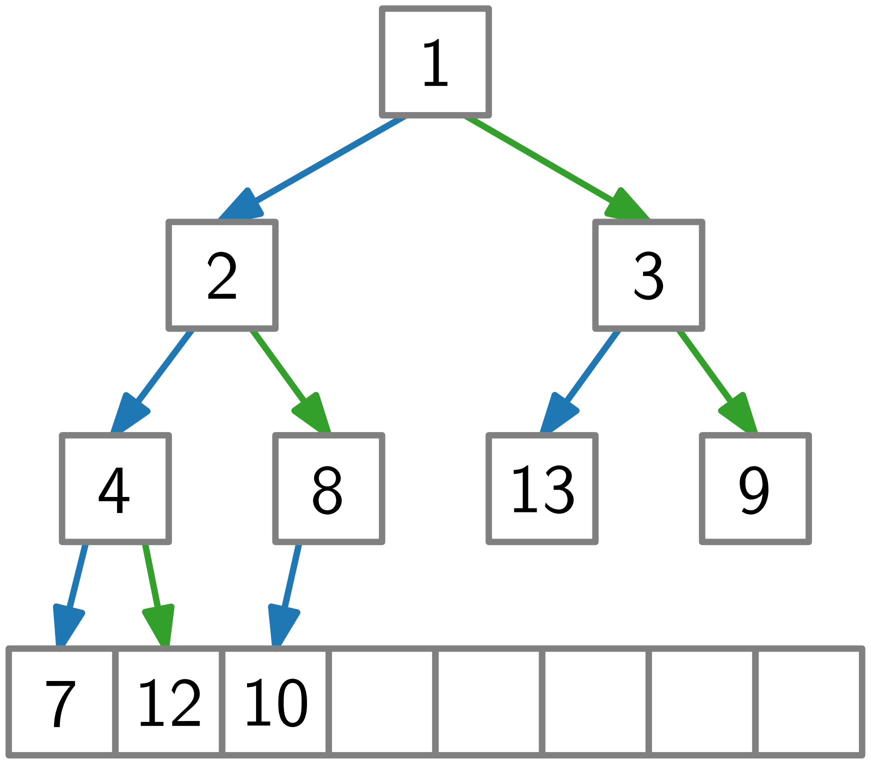
Min-Heaps



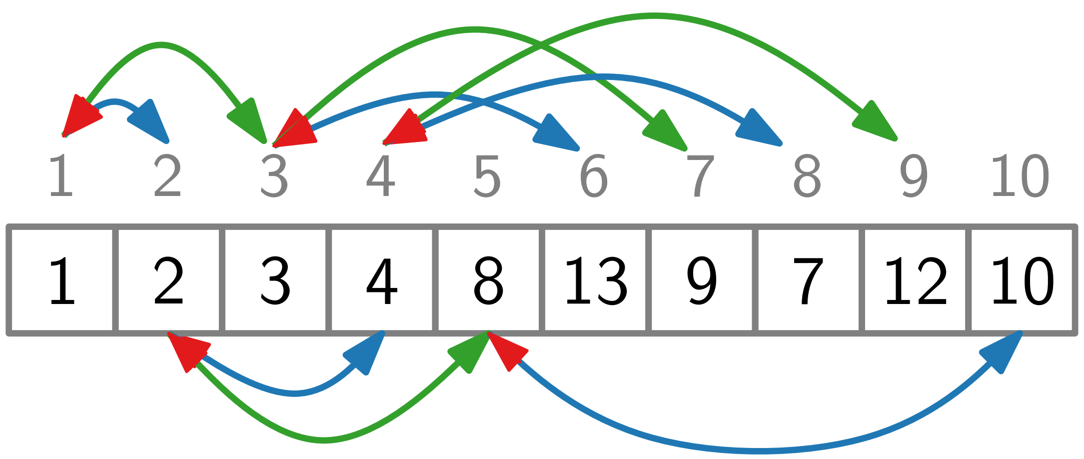
Pfeile implementieren:

LEFT(index i)	**return** $2i$
RIGHT(index i)	**return** $2i + 1$
PARENT(index i)	**return** ...



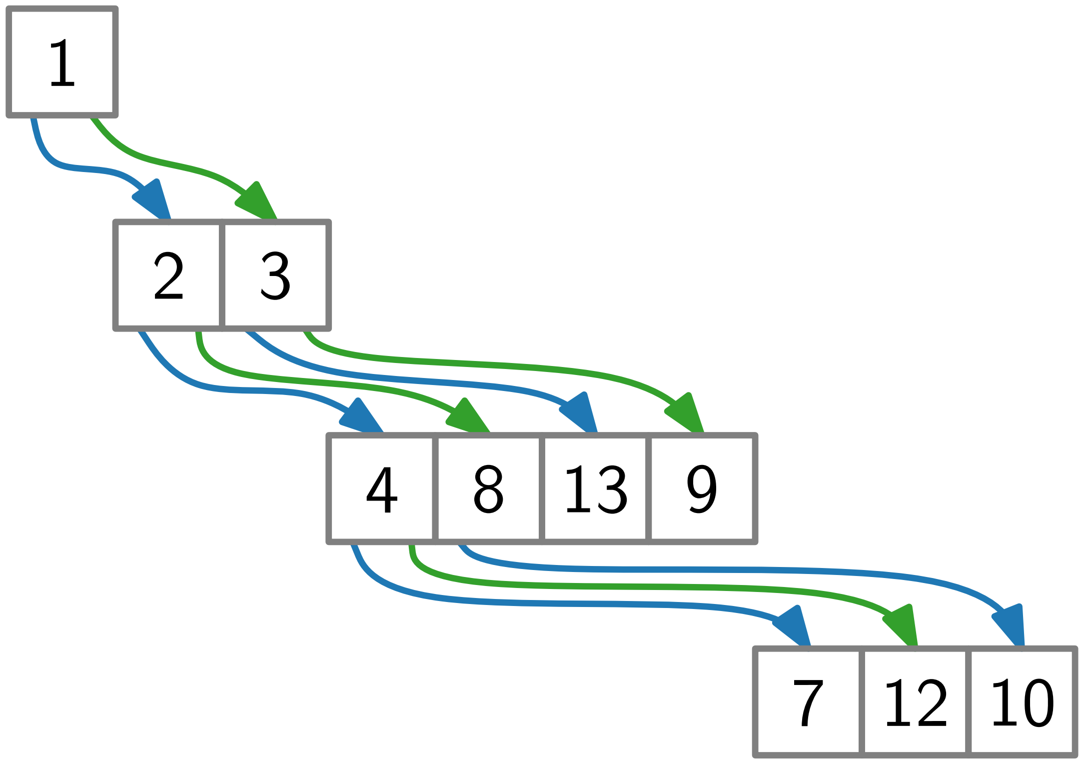


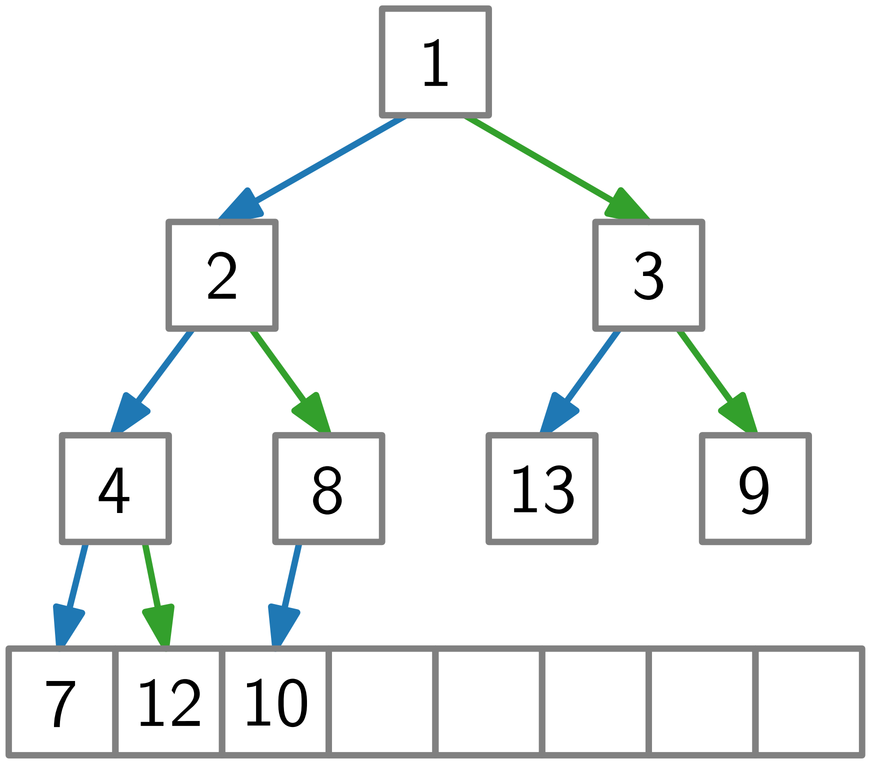
Min-Heaps



Pfeile implementieren:

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



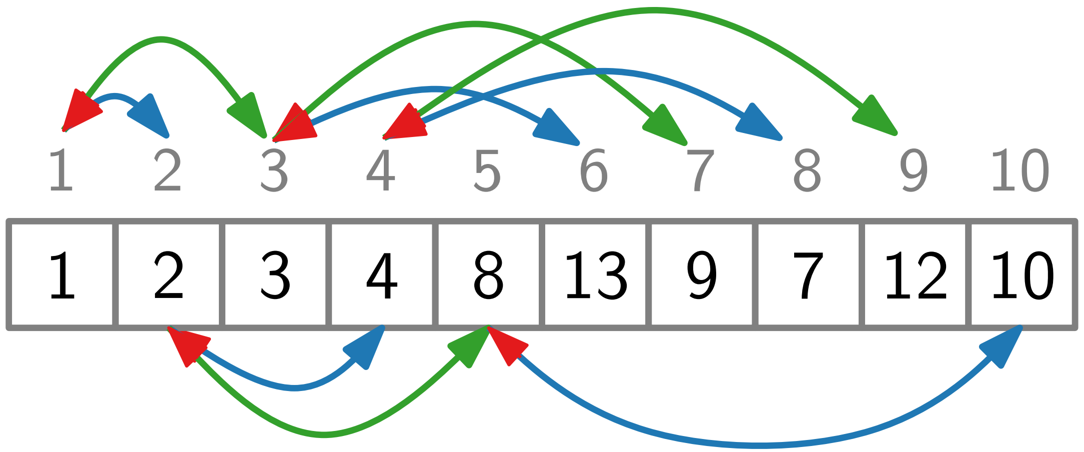


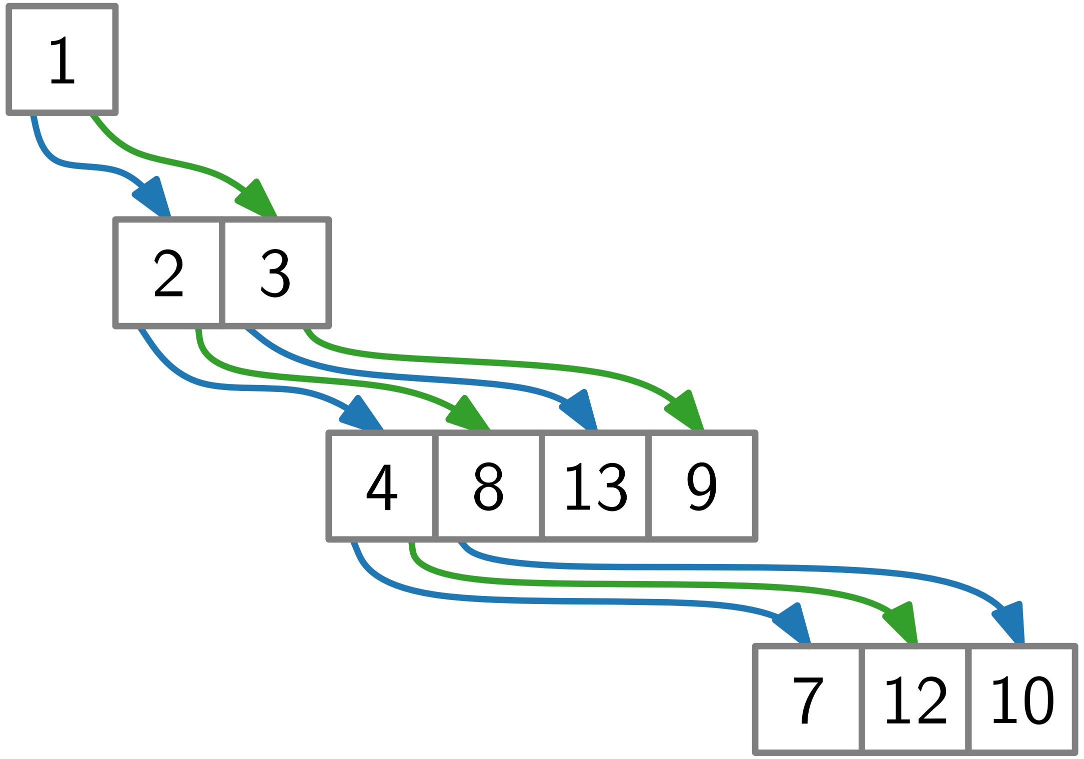
Min-Heaps

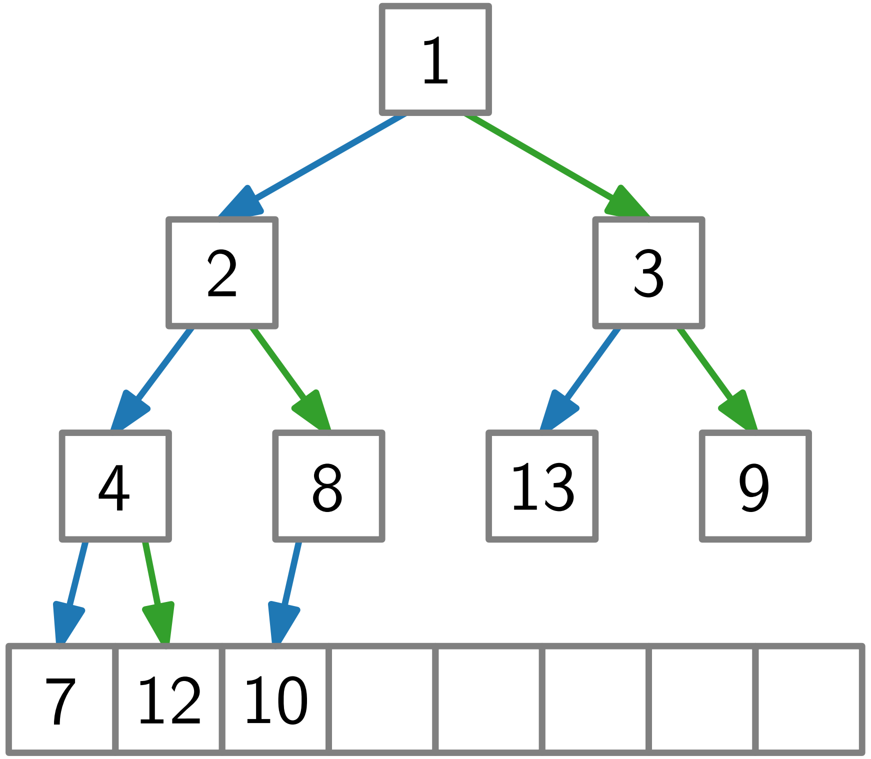
sehr schnelle Rechenoperationen!

Pfeile implementieren:

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

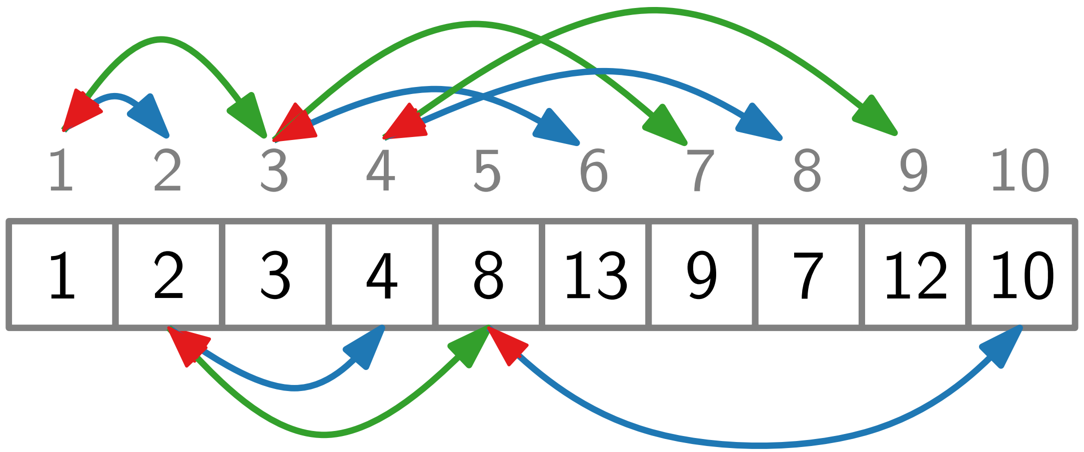






Min-Heaps

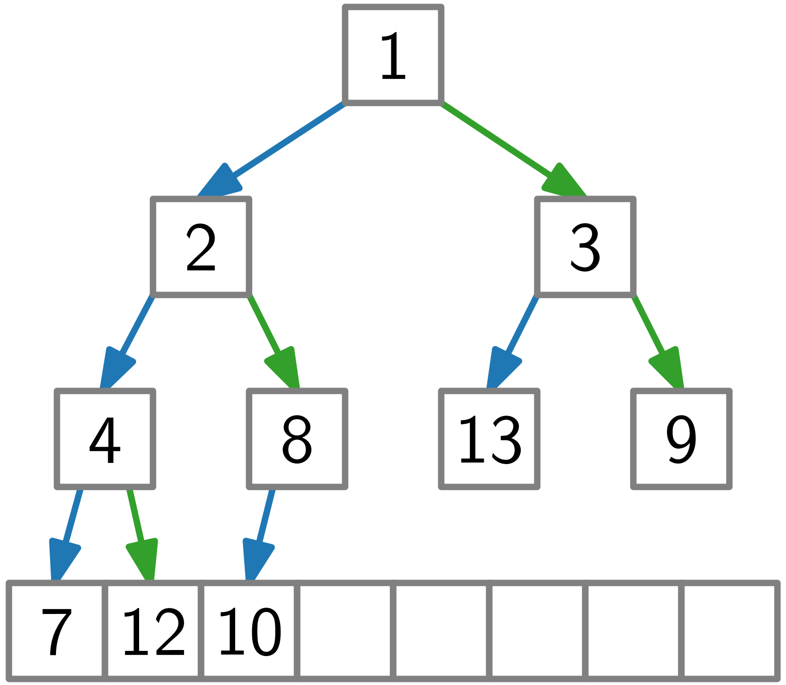
sehr schnelle Rechenoperationen!



Pfeile implementieren:

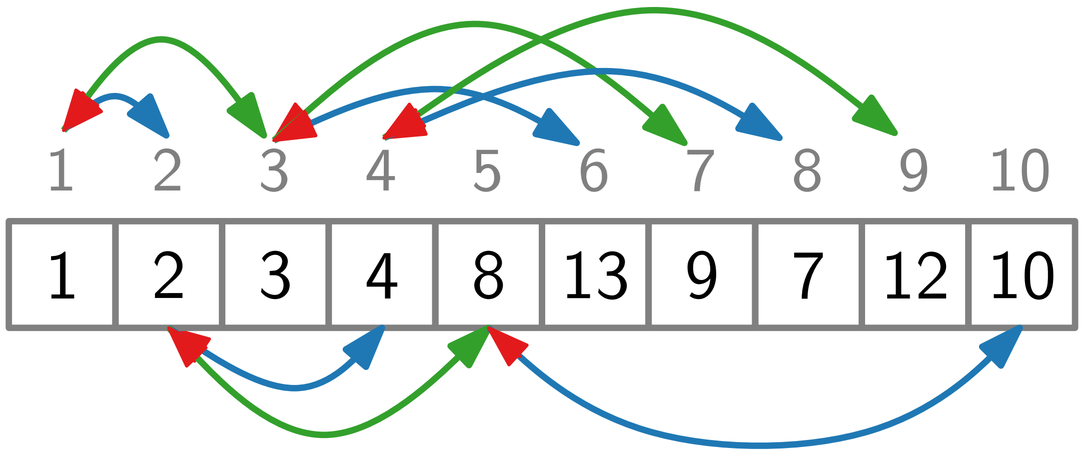
LEFT(index i)	**return**	$2i$
RIGHT(index i)	**return**	$2i + 1$
PARENT(index i)	**return**	$\lfloor i/2 \rfloor$

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



Min-Heaps

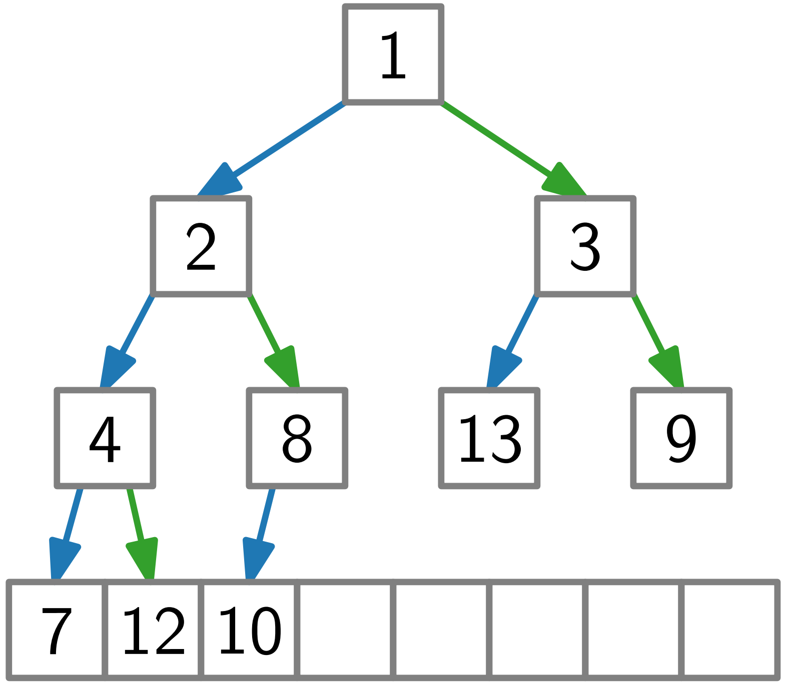
sehr schnelle Rechenoperationen!



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

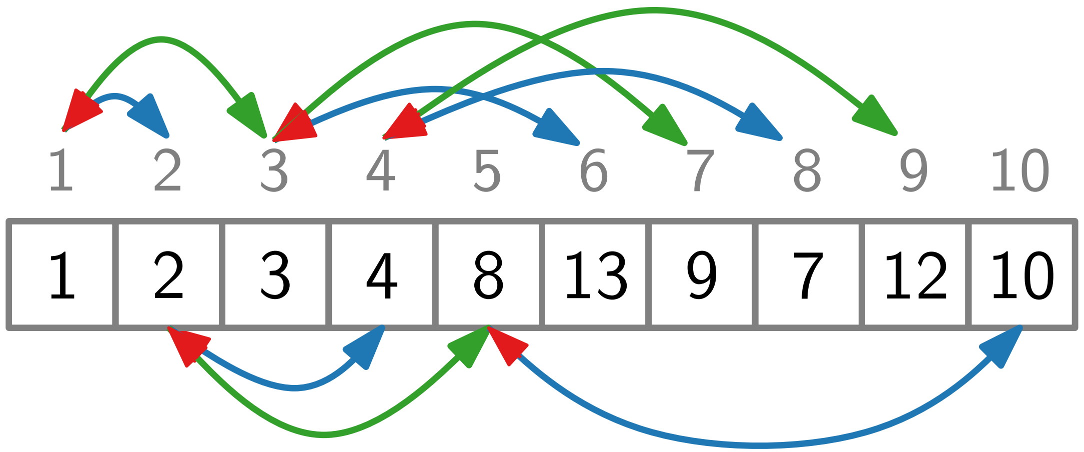


Min-Heaps

sehr schnelle Rechenoperationen!

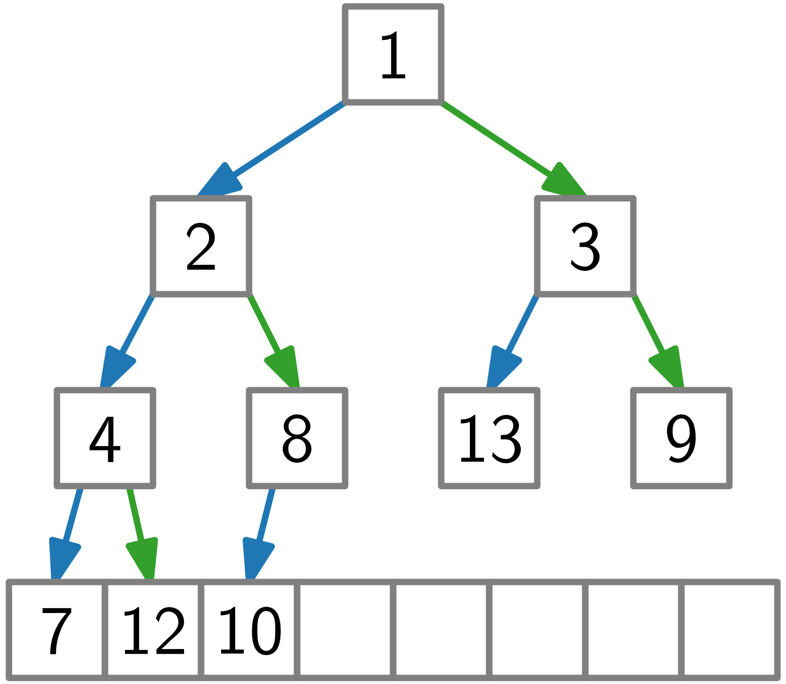
Pfeile implementieren:

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



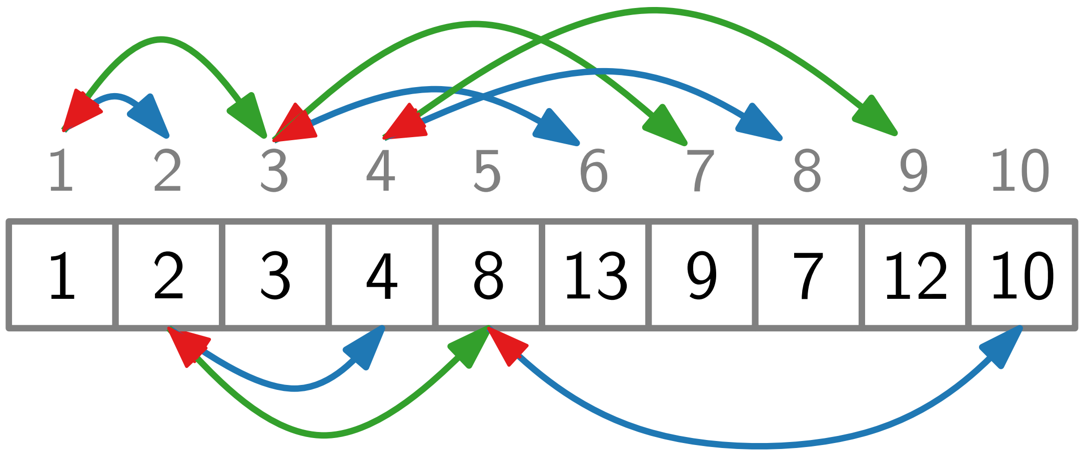
Definition.
Ein **Heap** ist ein Feld, das einem $\boxed{\text{binären}}$ Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,



Min-Heaps

sehr schnelle Rechenoperationen!

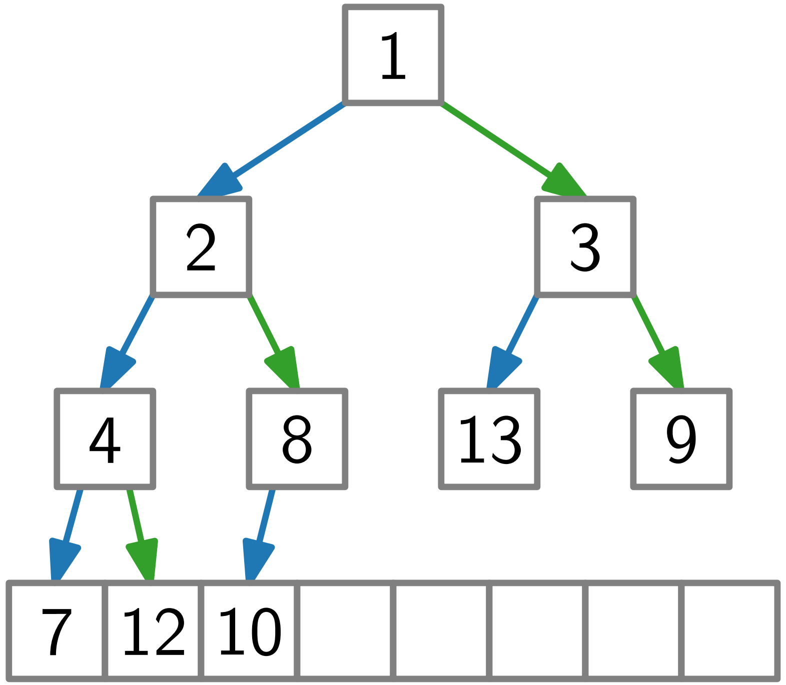


Pfeile implementieren:

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

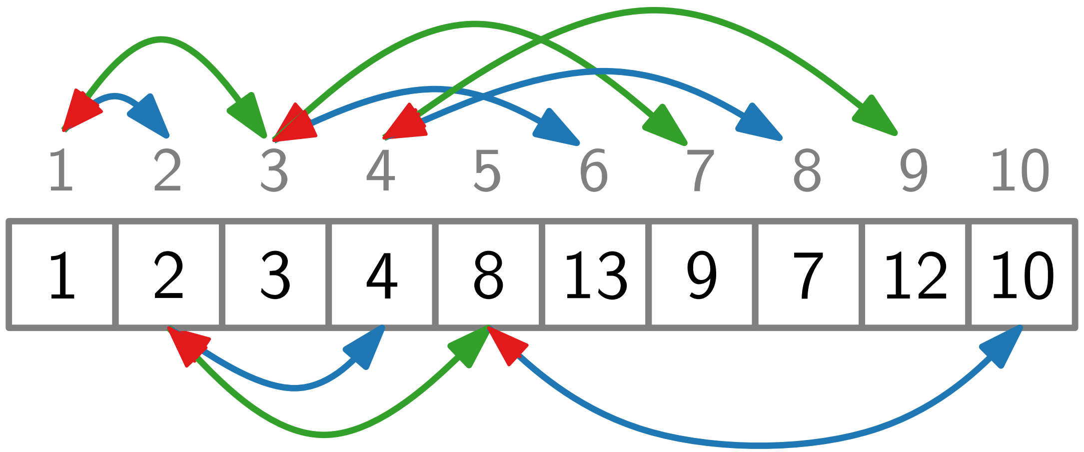
Definition.
Ein **Heap** ist ein Feld, das einem binären Baum entspricht, bei dem

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



Min-Heaps

sehr schnelle Rechenoperationen!

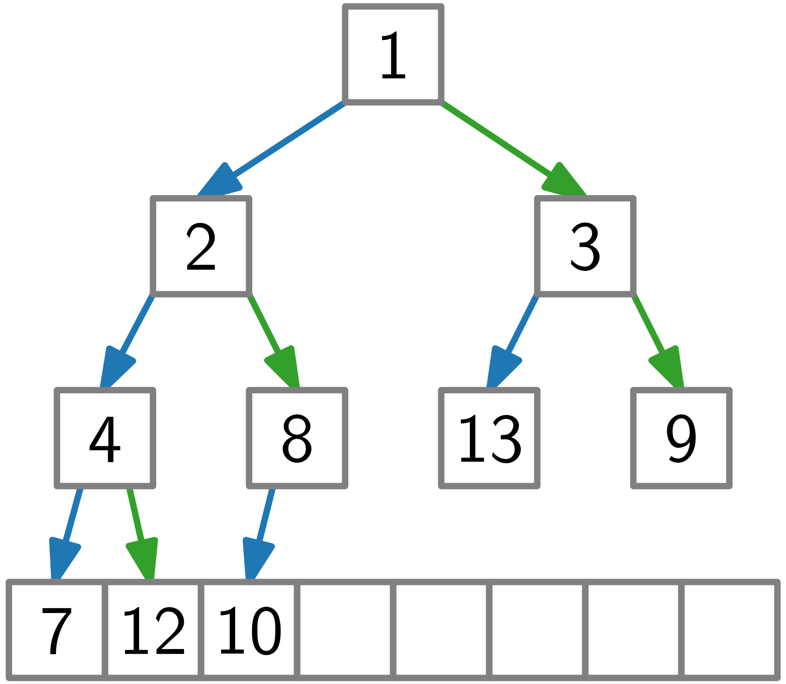


Pfeile implementieren:

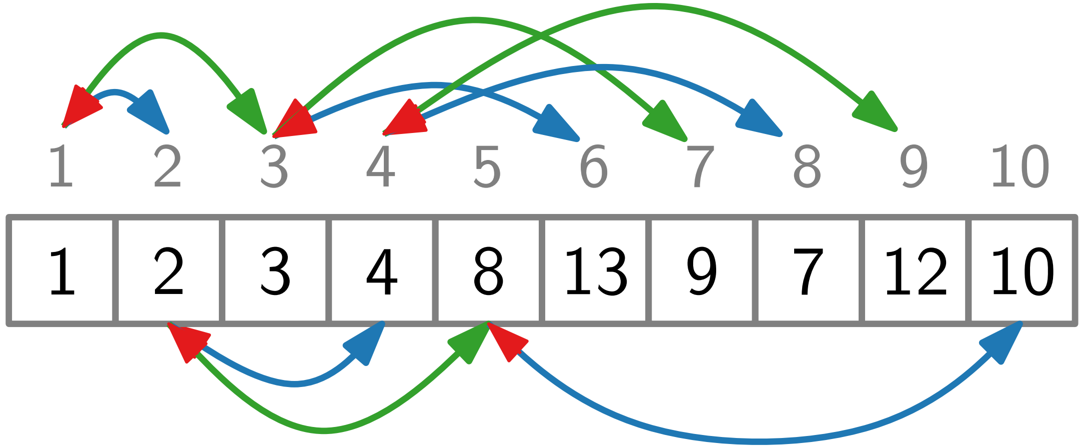
LEFT(index i)	**return**	$2i$
RIGHT(index i)	**return**	$2i + 1$
PARENT(index i)	**return**	$\lfloor i/2 \rfloor$

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



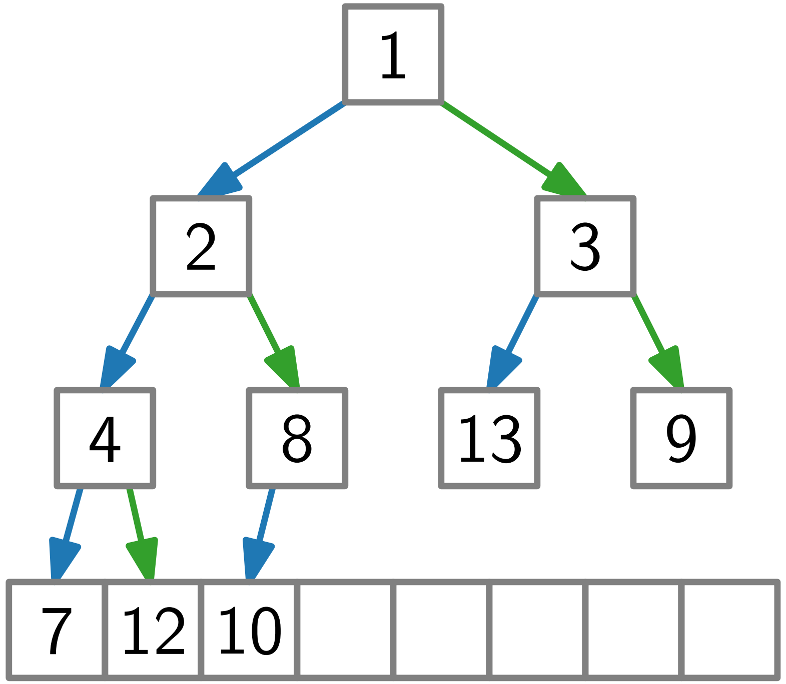
sehr schnelle Rechenoperationen!

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

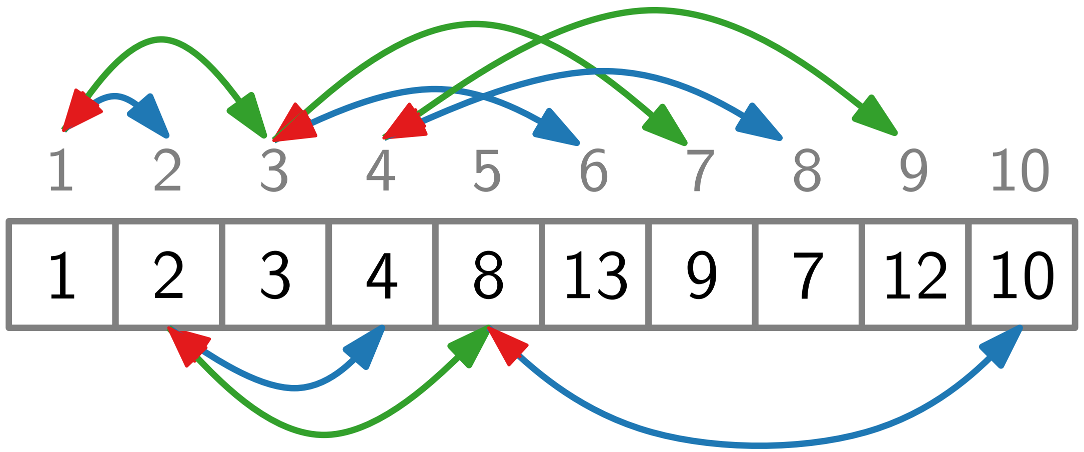
- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und
- die **Heap-Eigenschaft** gilt.



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

Min-Heaps

sehr schnelle Rechenoperationen!



Pfeile implementieren:

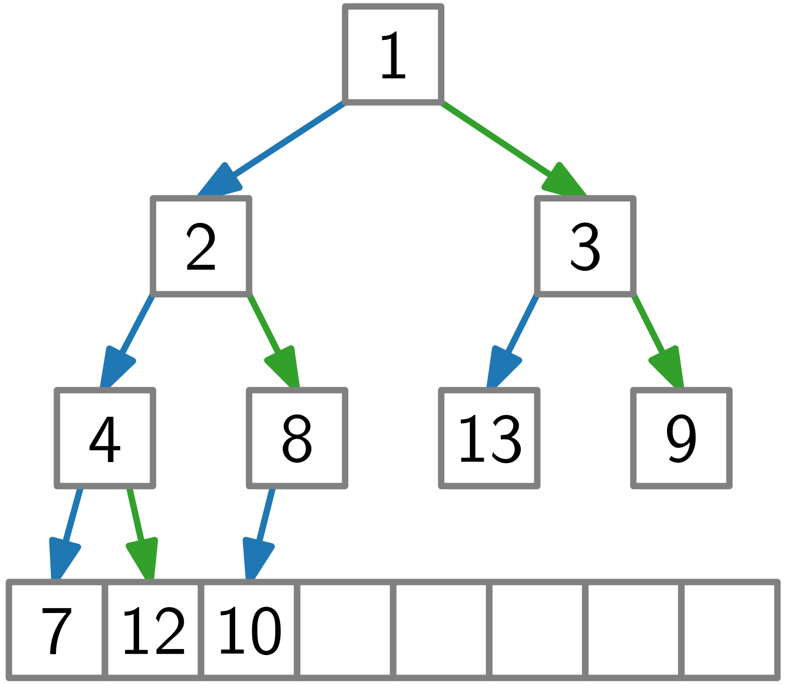
LEFT(index i) **return** $2i$

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.

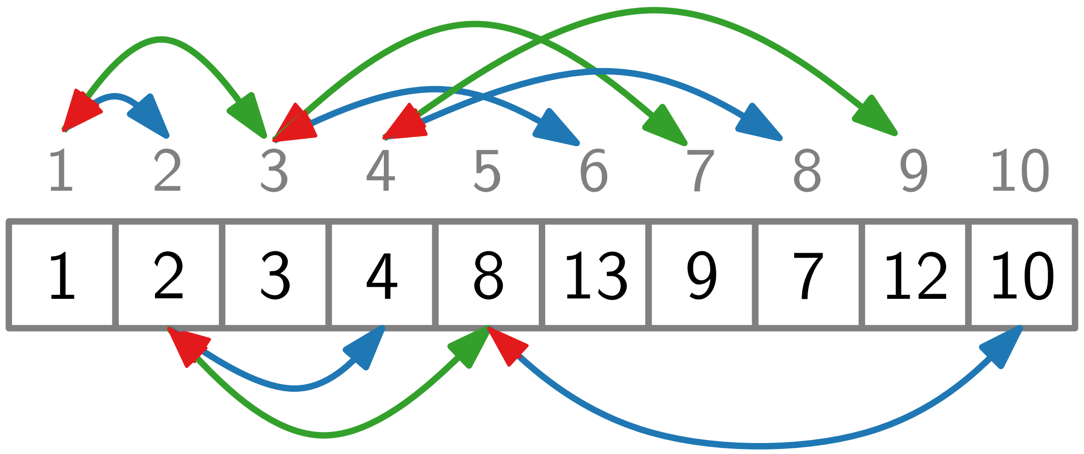


Definition.
Ein Heap hat die **Min-Heap-Eigenschaft**,

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

Min-Heaps

sehr schnelle Rechenoperationen!

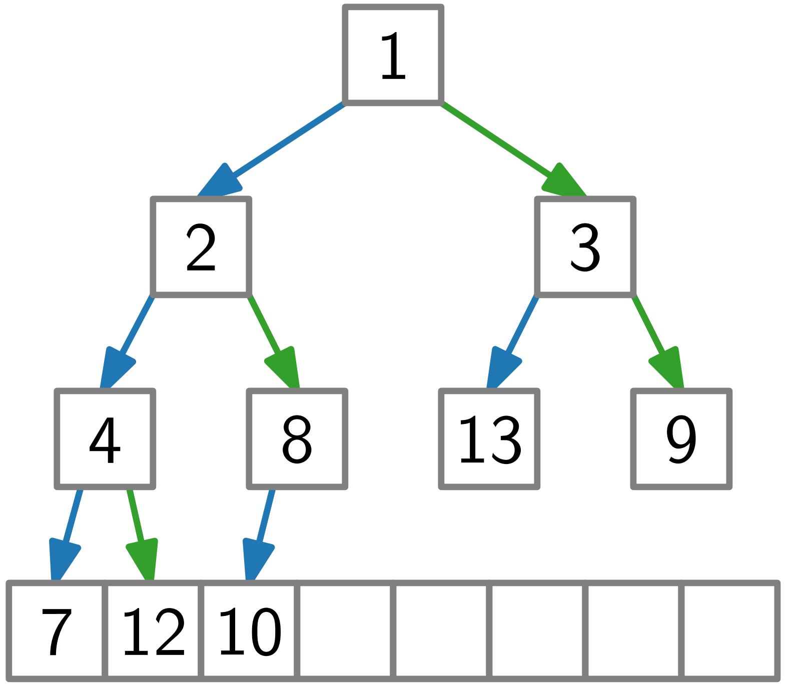


Pfeile implementieren:

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.

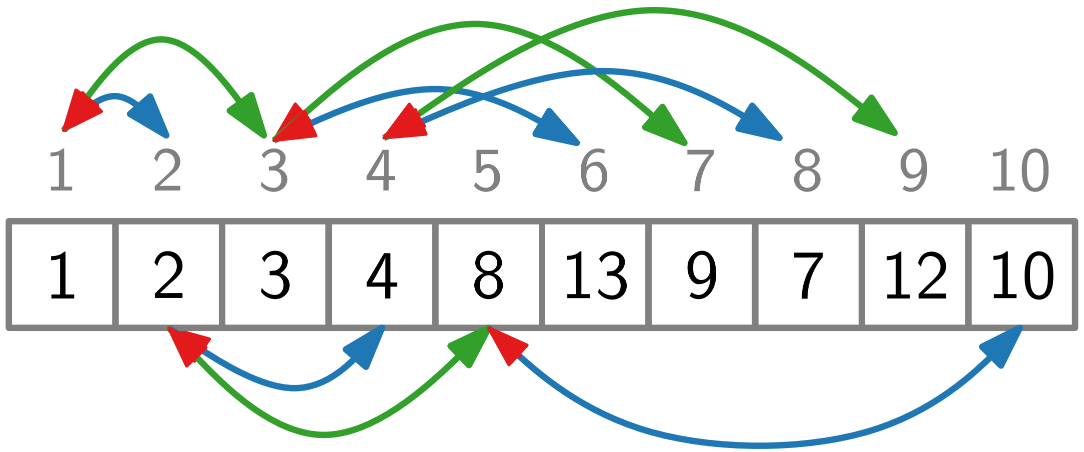


Definition.
Ein Heap hat die **Min-Heap-Eigenschaft**,

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

So ein Heap heißt **Min-Heap**.

Min-Heaps



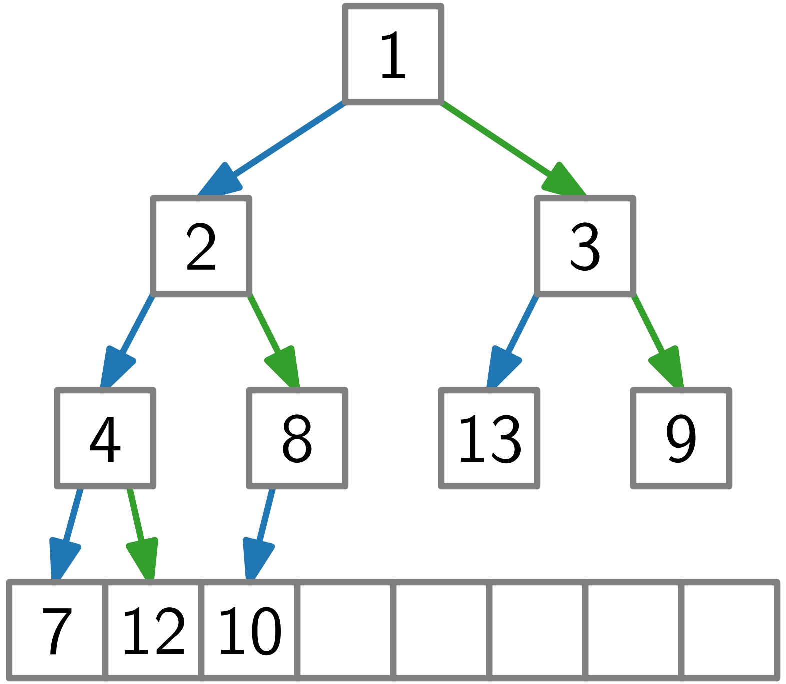
sehr schnelle Rechenoperationen!

Pfeile implementieren:

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.

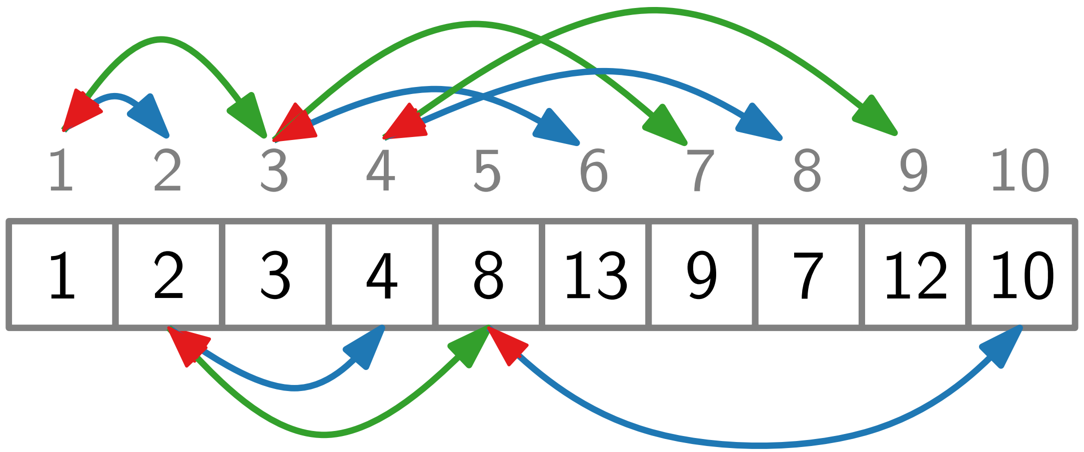


Definition.
Ein Heap hat die **~~Min~~-Heap-Eigenschaft**,

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

So ein Heap heißt **~~Min~~-Heap**.

Min-Heaps



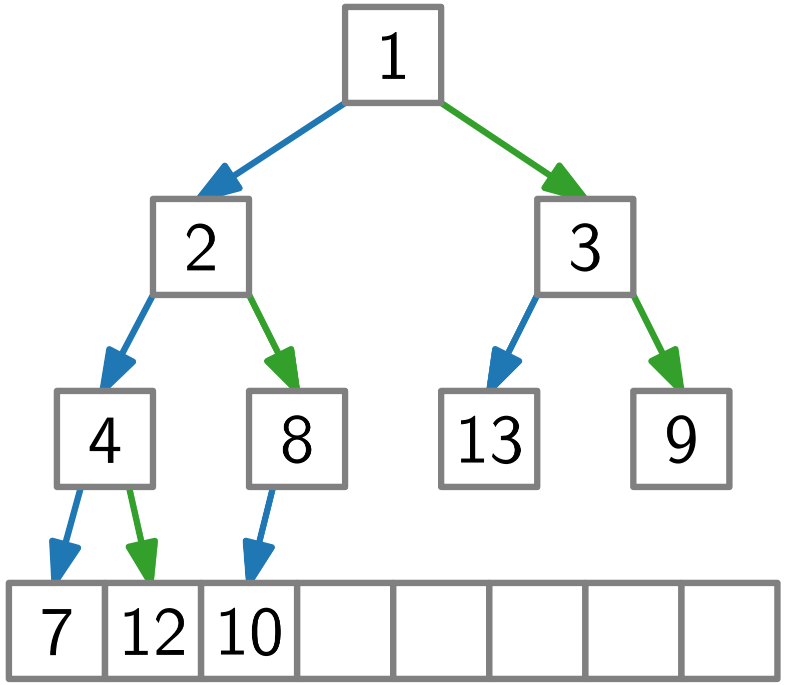
sehr schnelle Rechenoperationen!

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 $\boxed{\text{binären}}$ 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: $A[\text{PARENT}(i)] \cancel{\leq} A[i]$.
$\geq$

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

Baustelle

7	12	10	4	8	13	3	1	9	2

Baustelle

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 7 & 12 & 10 & 4 & 8 & 13 & 3 & 1 & 9 & 2 \\ \hline\end{array} \longrightarrow \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 3 & 4 & 7 & 13 & 10 & 12 & 9 & 8 \\ \hline\end{array}$$

Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

$\longrightarrow$

1	2	3	4	7	13	10	12	9	8

Min-Heap-Eigenschaft

Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

$\longrightarrow$

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!

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!

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

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

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.

Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

$\longrightarrow$

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

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!

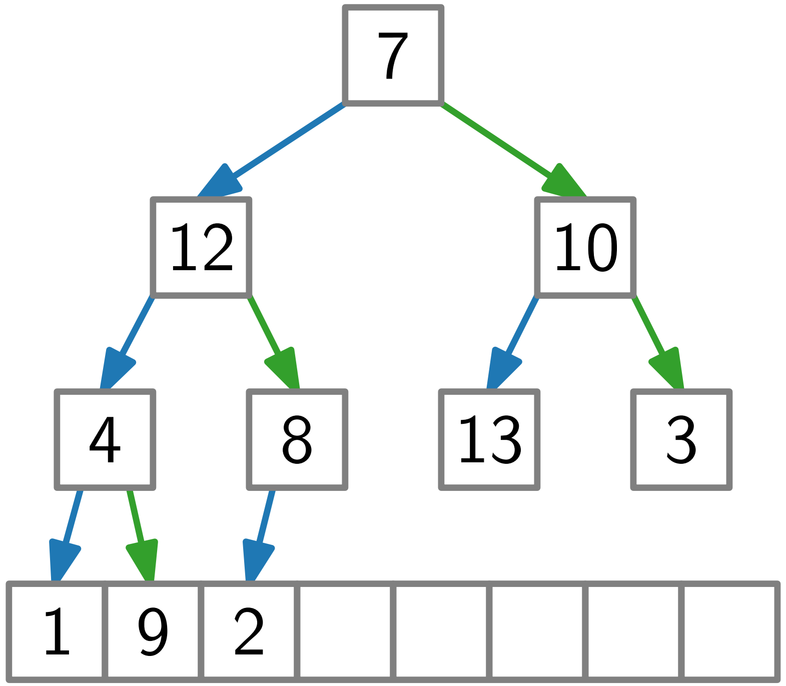
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!



Baustelle

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!

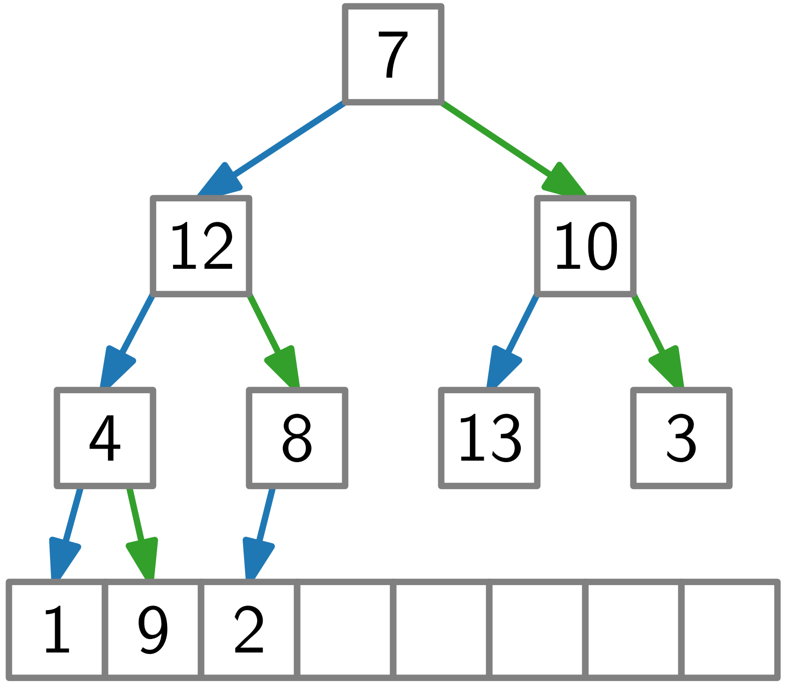
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

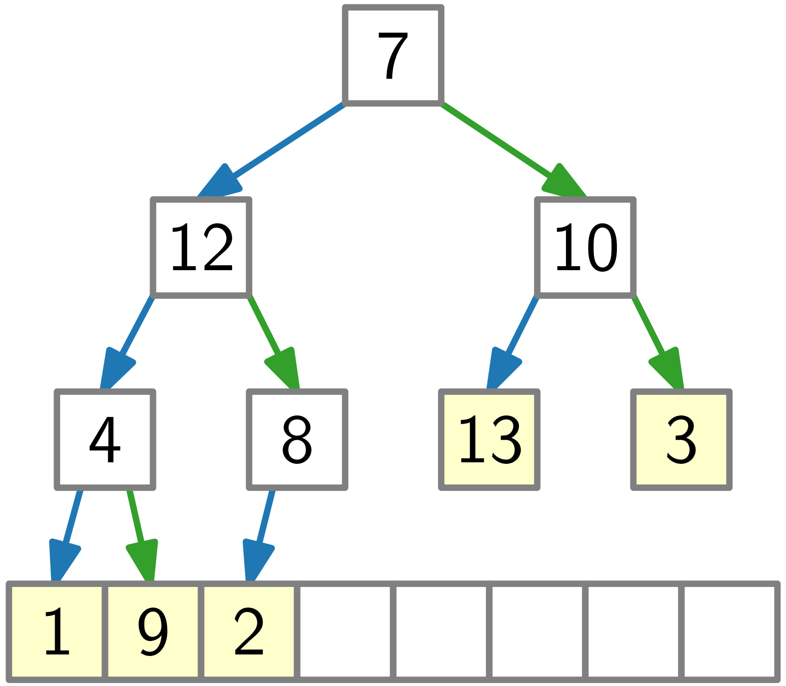
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

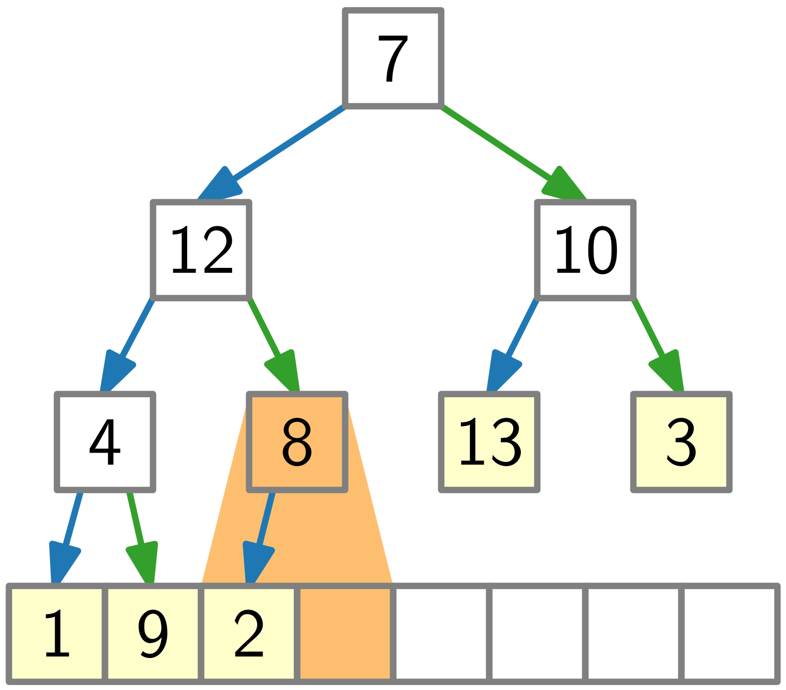
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

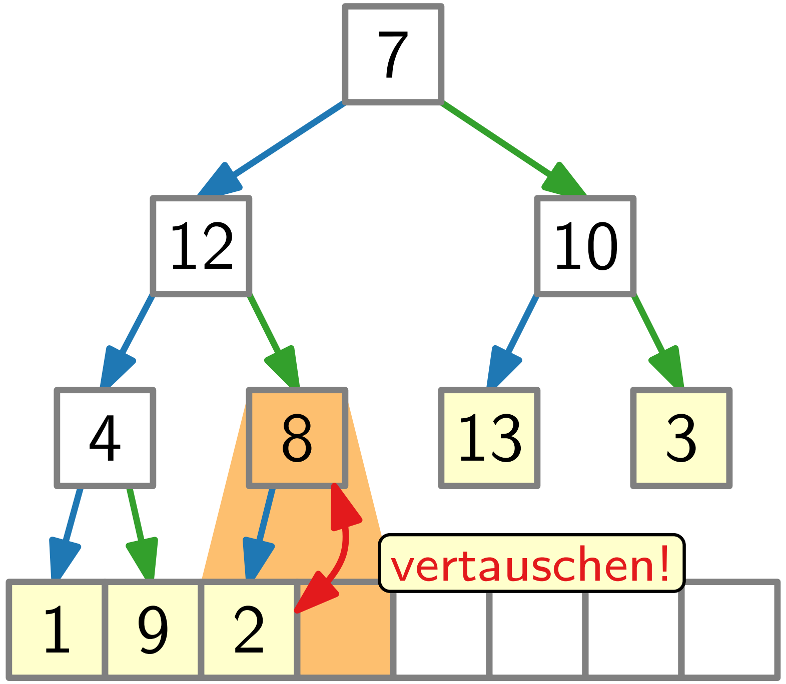
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

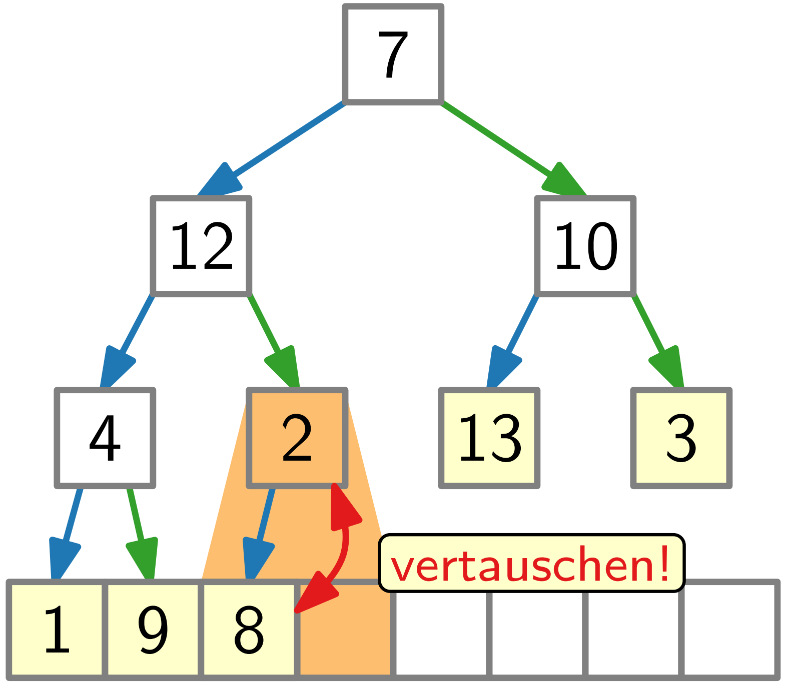
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

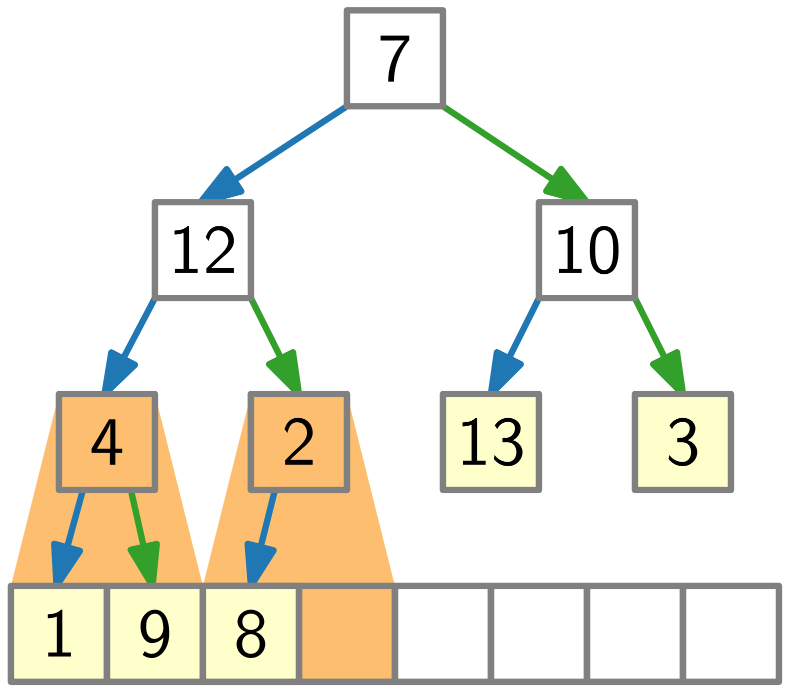
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

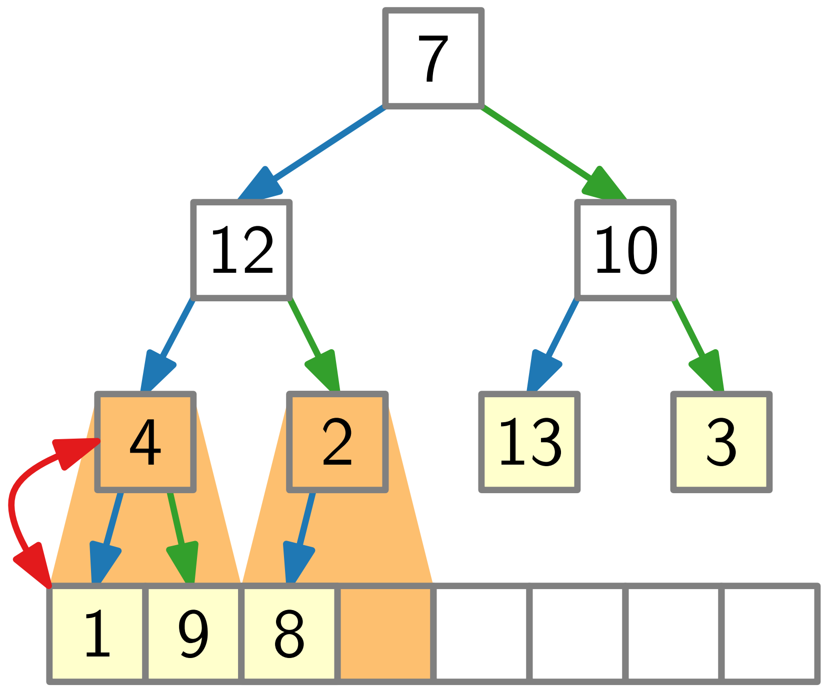
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

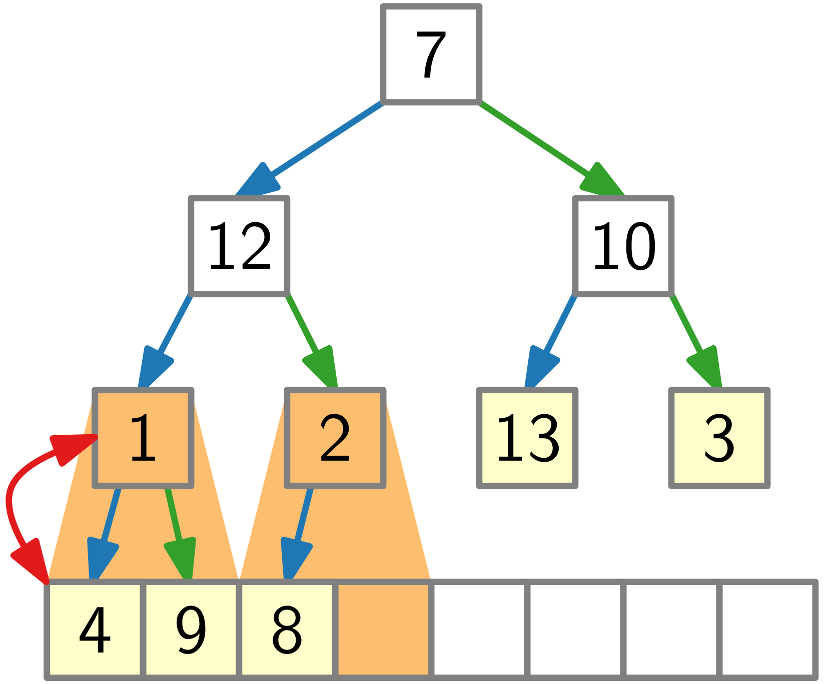
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

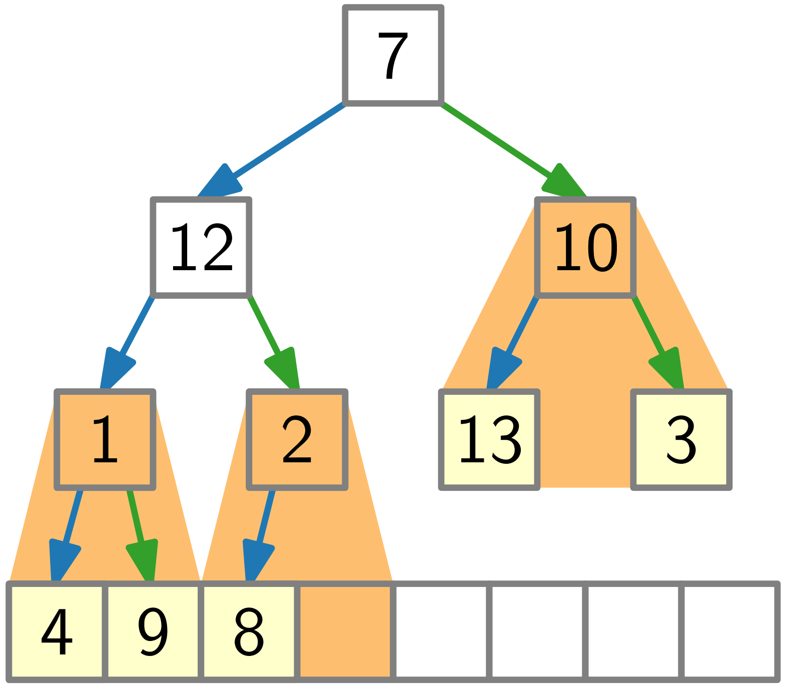
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

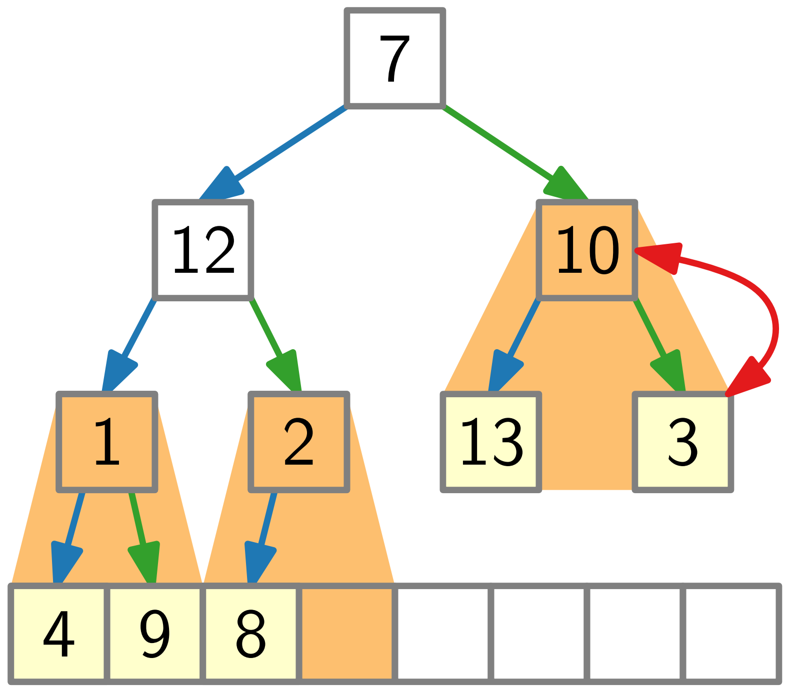
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

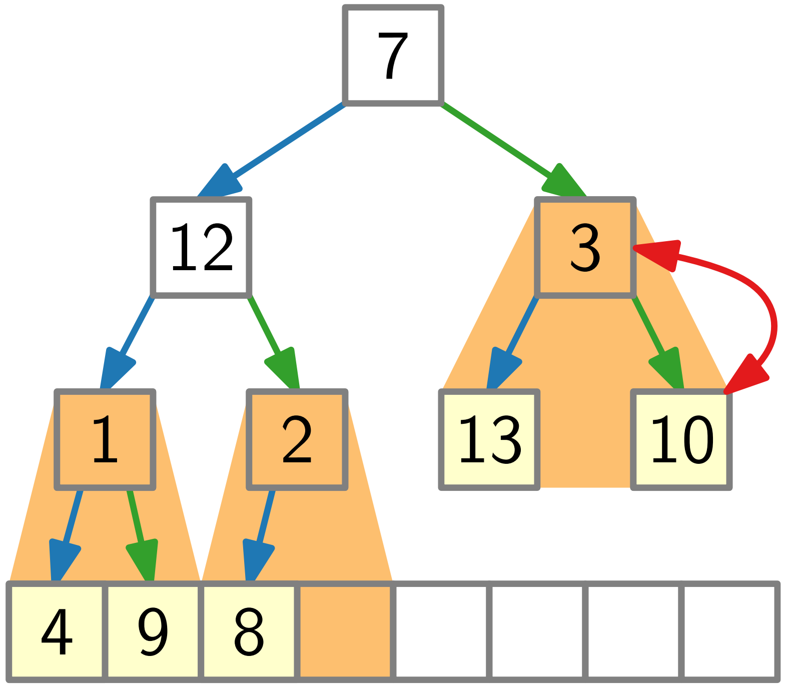
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

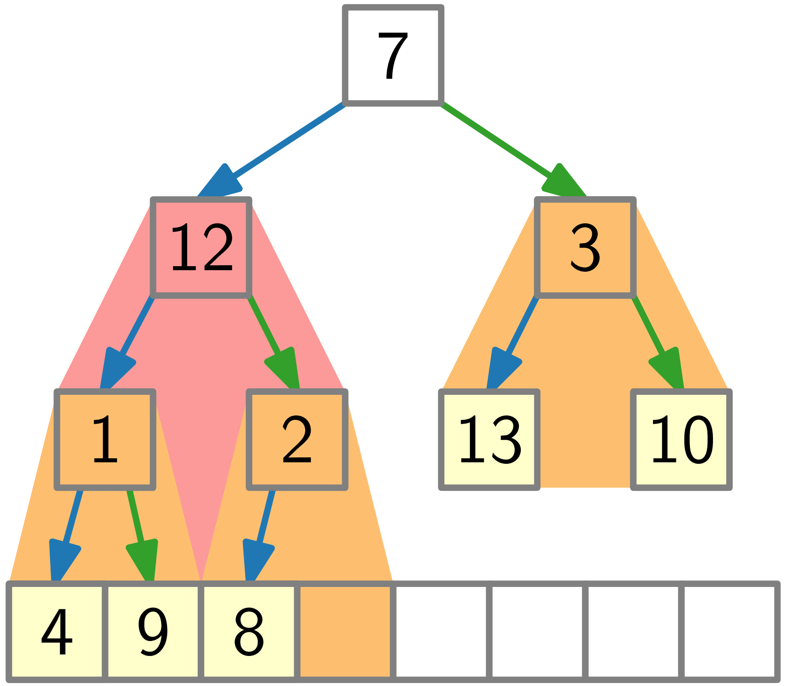
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

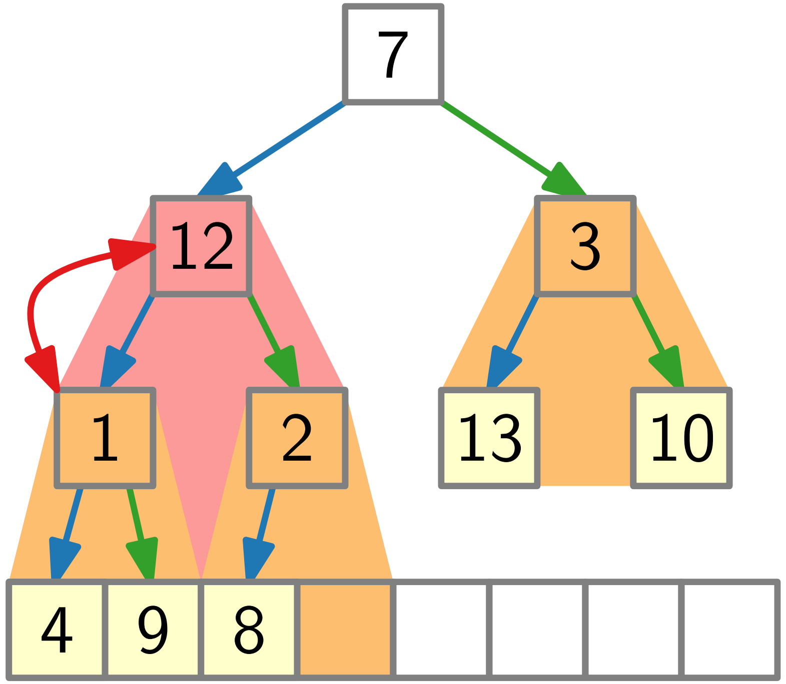
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

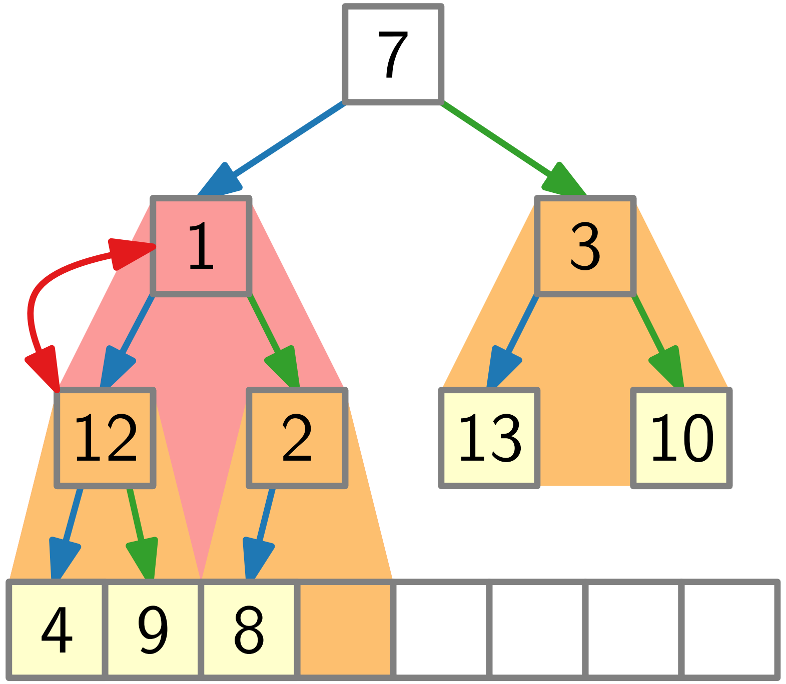
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

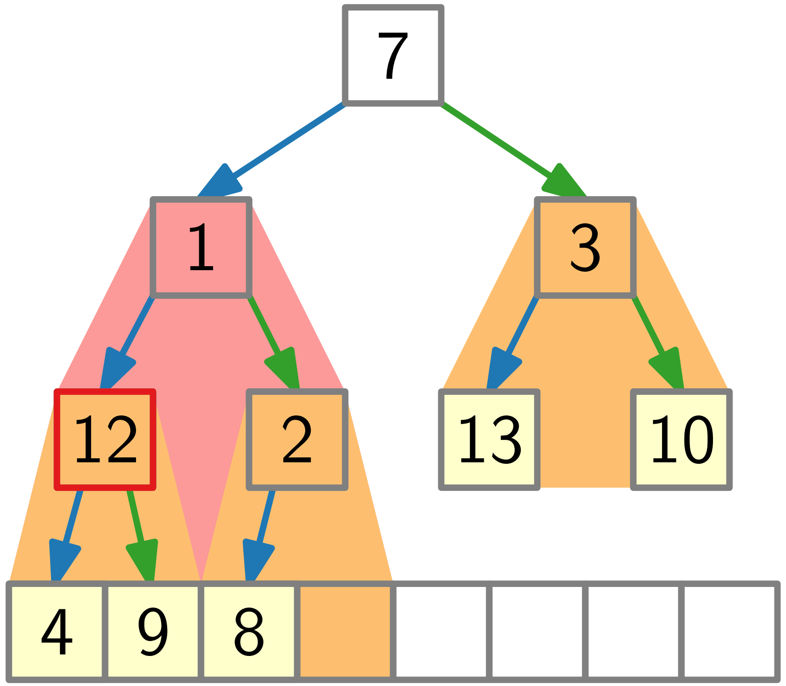
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

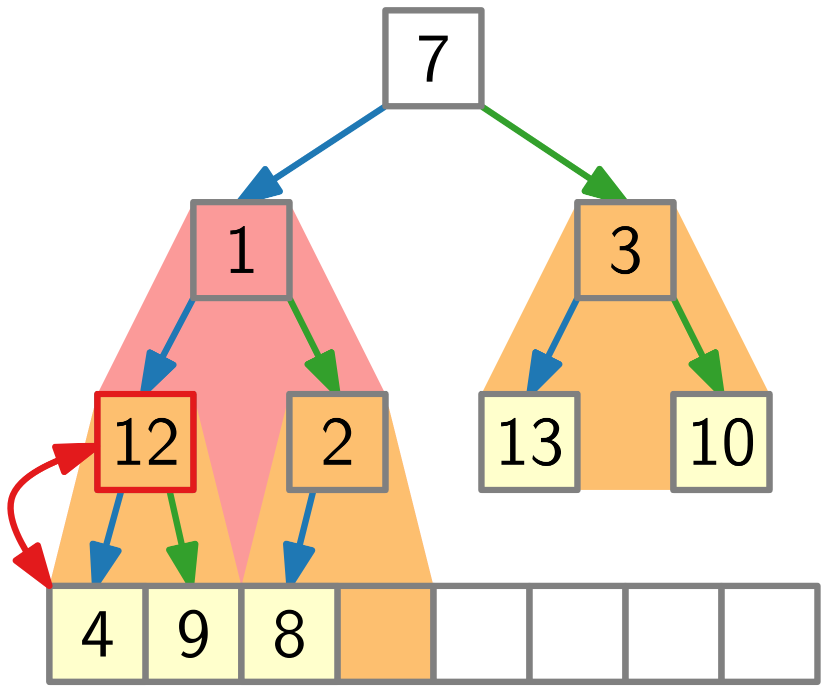
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

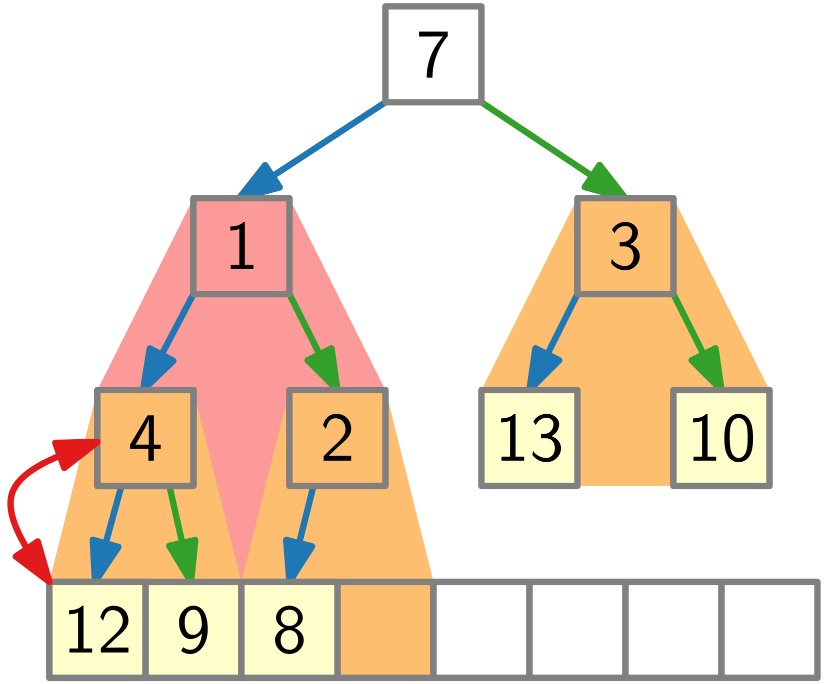
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

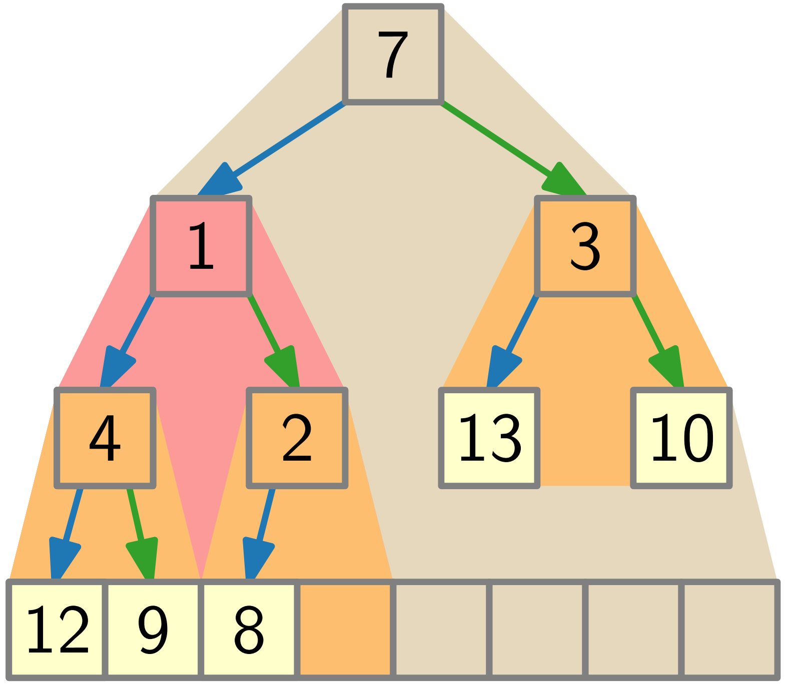
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

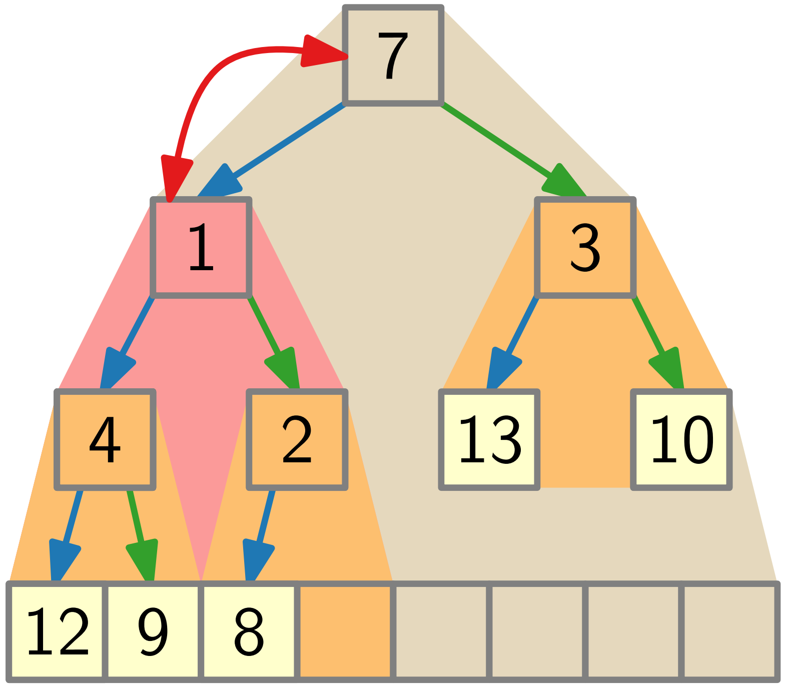
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

$\longrightarrow$

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!

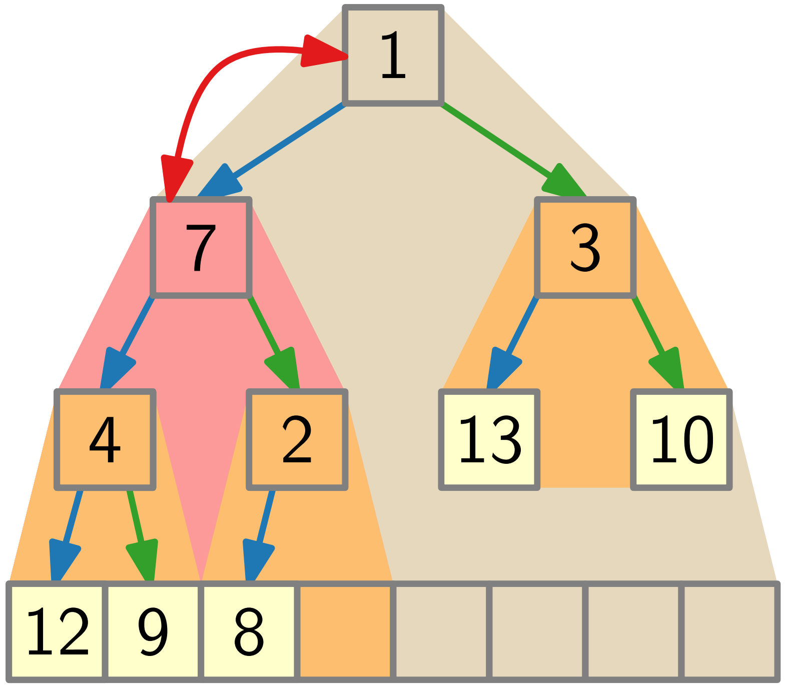
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

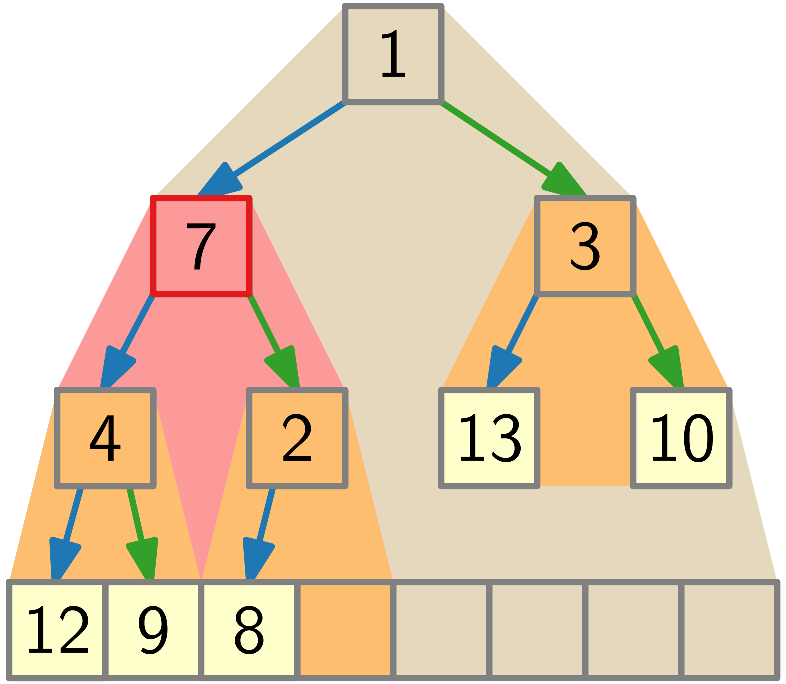
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

$\longrightarrow$

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!

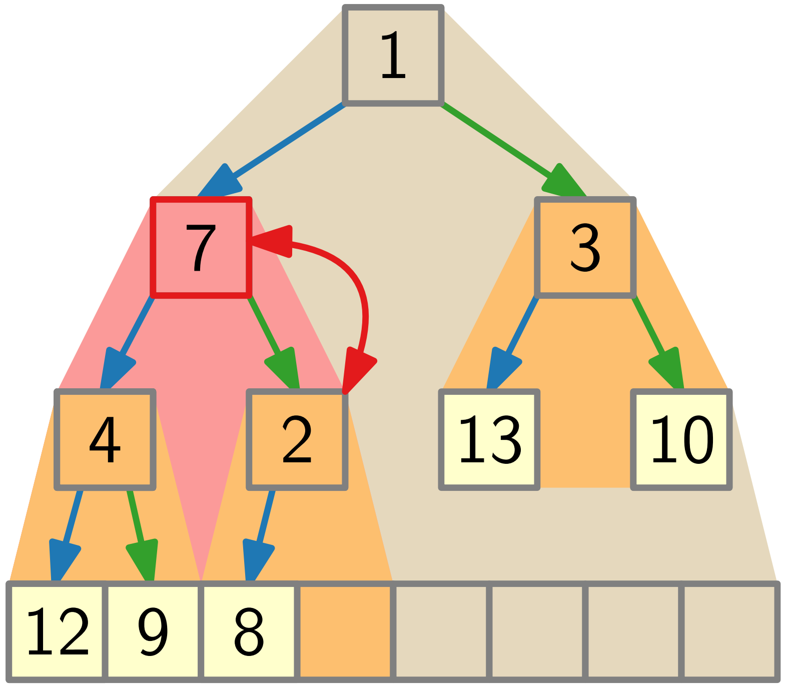
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

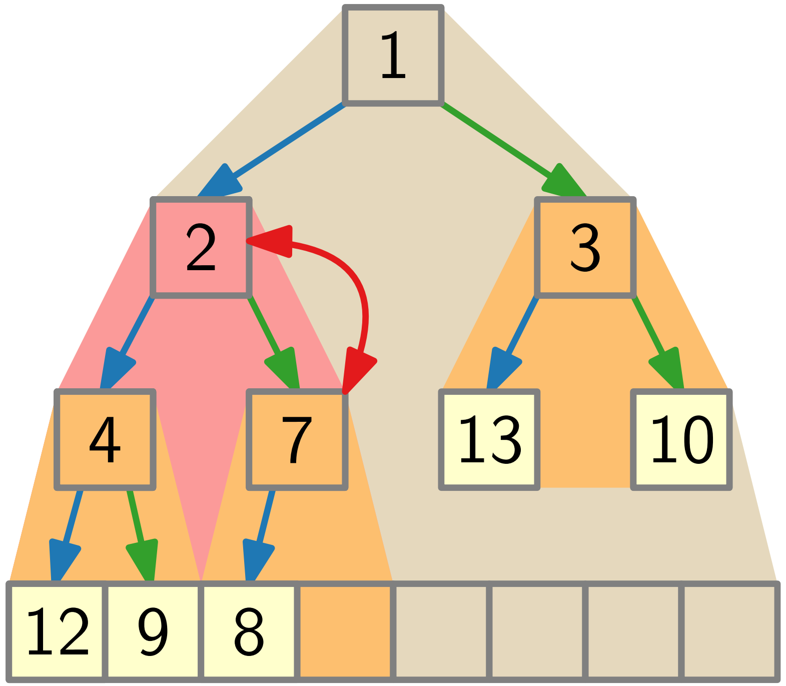
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



Baustelle

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!

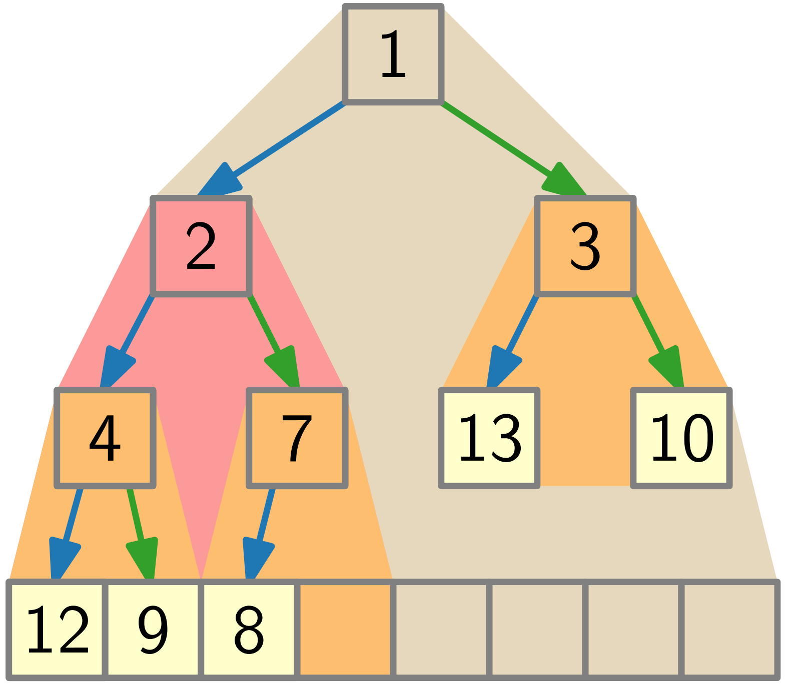
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

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

Hoffen: Schnellere Berechnung!

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



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

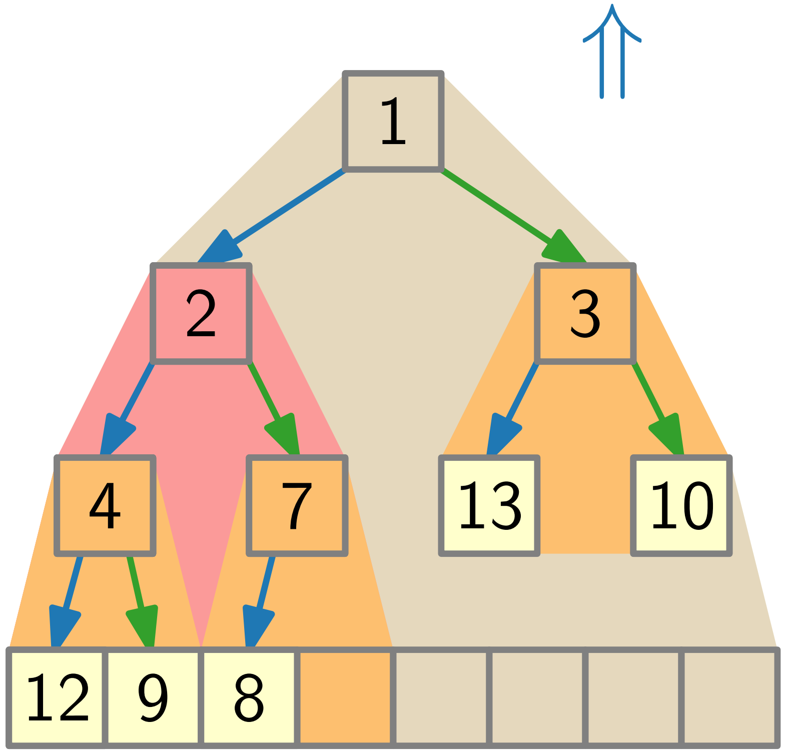
Aufgabe: Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

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

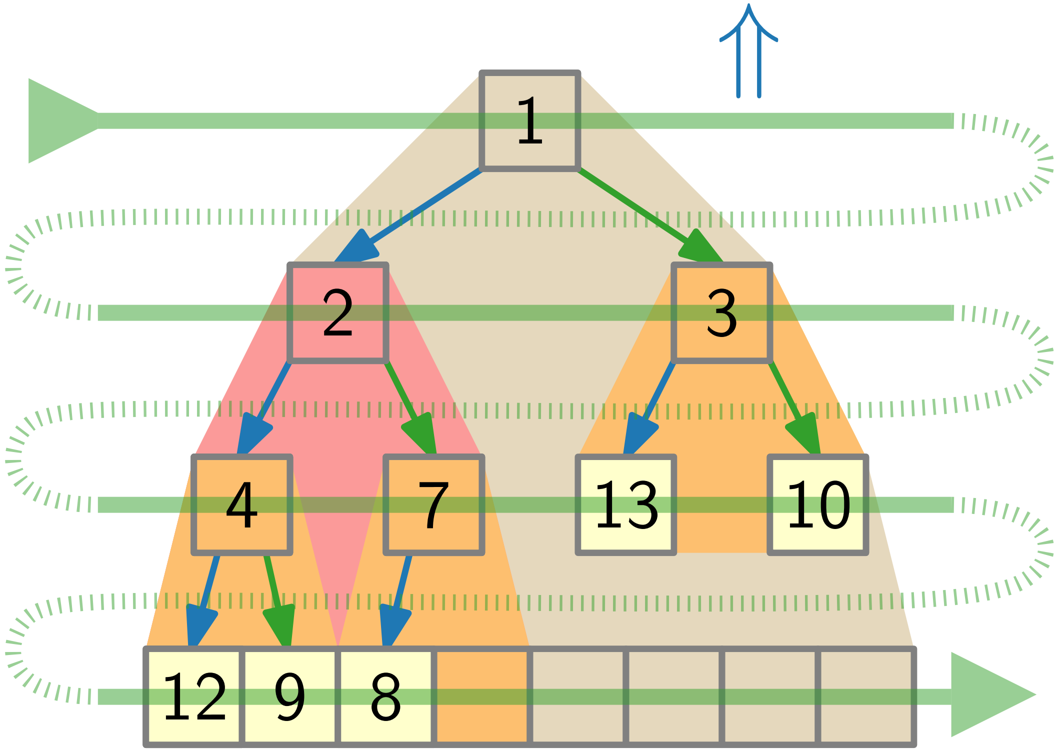
Aufgabe: Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MergeSort!

1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

– Ergebnis –



Fertig!

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!

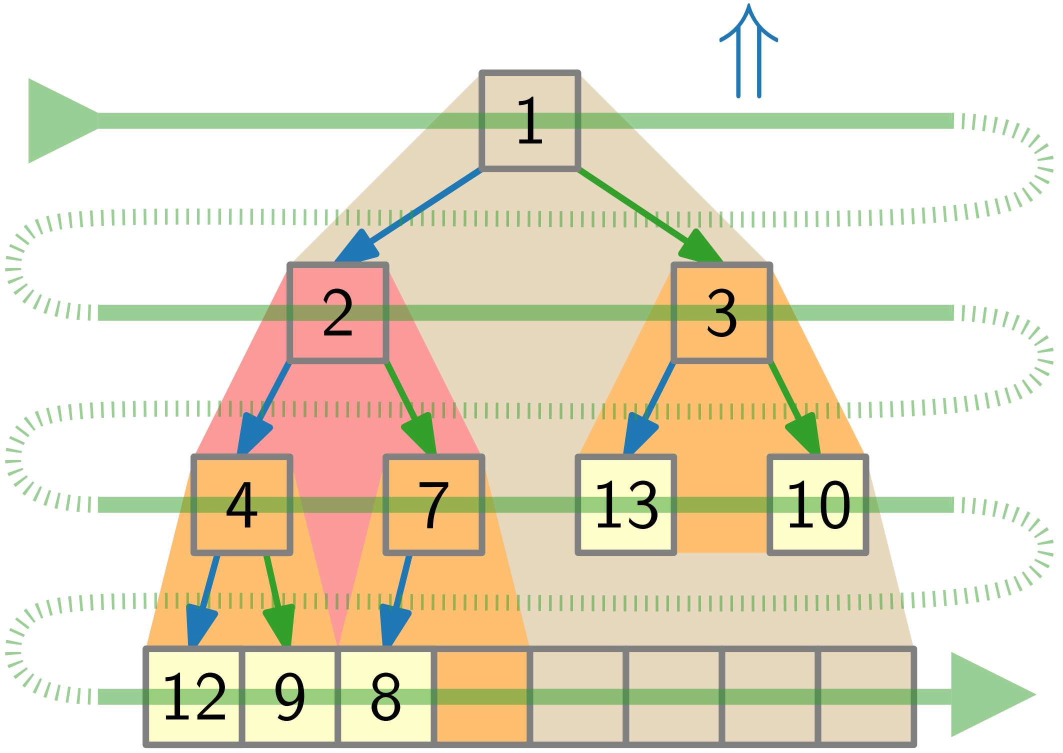
Nimm MergeSort!

1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

1	2	3	4	7	13	10	12	9	8

– Ergebnis –



Fertig!

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

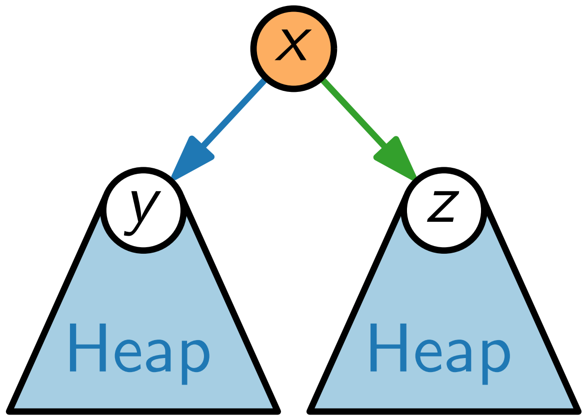
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

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

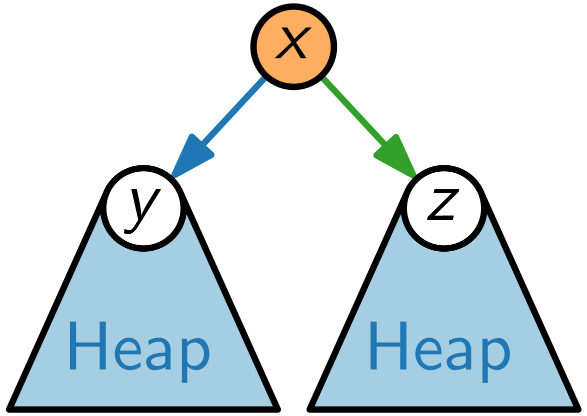
Elementaroperation

„Versickere“ x, falls x zu groß



Elementaroperation

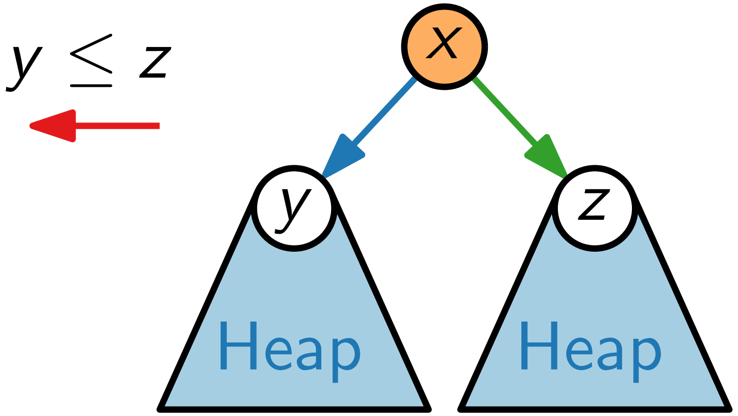
„Versickere“ x, falls x zu groß, d.h. falls $x > \min(y, z)$



Elementaroperation

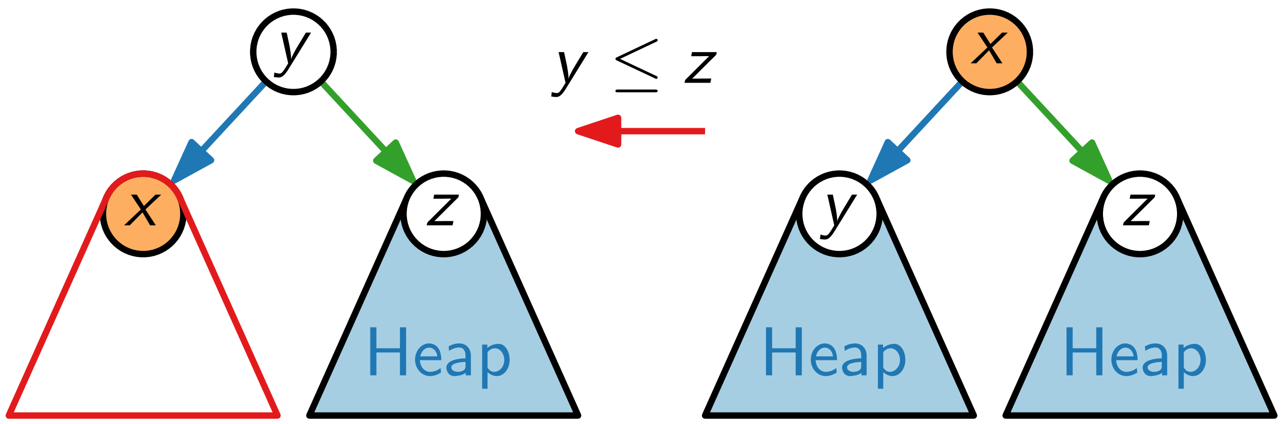
„Versickere" x, falls x zu groß, d.h. falls $x > \min(y, z)$

$y \leq z$



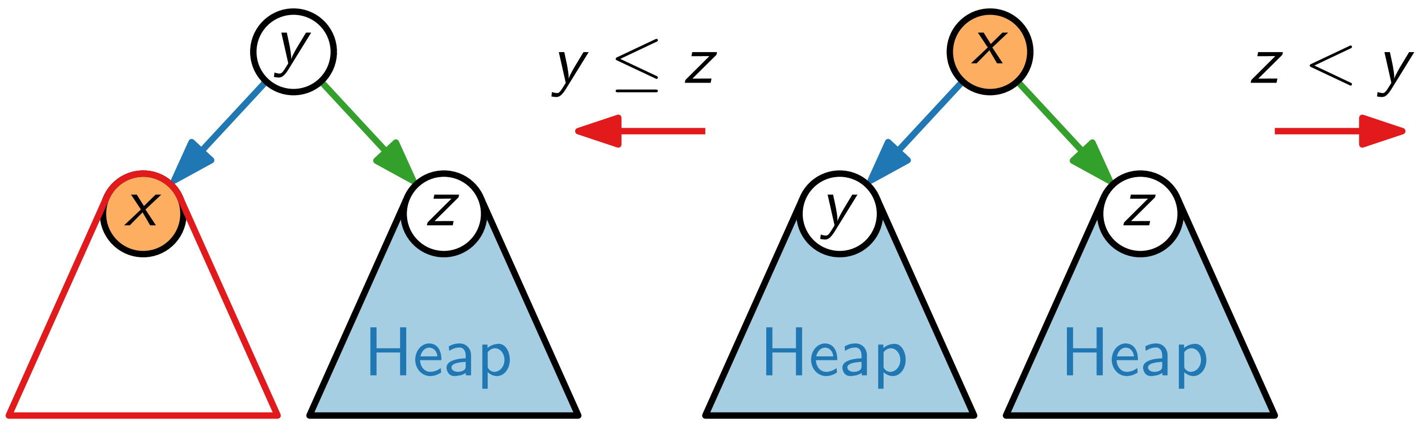
Elementaroperation

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



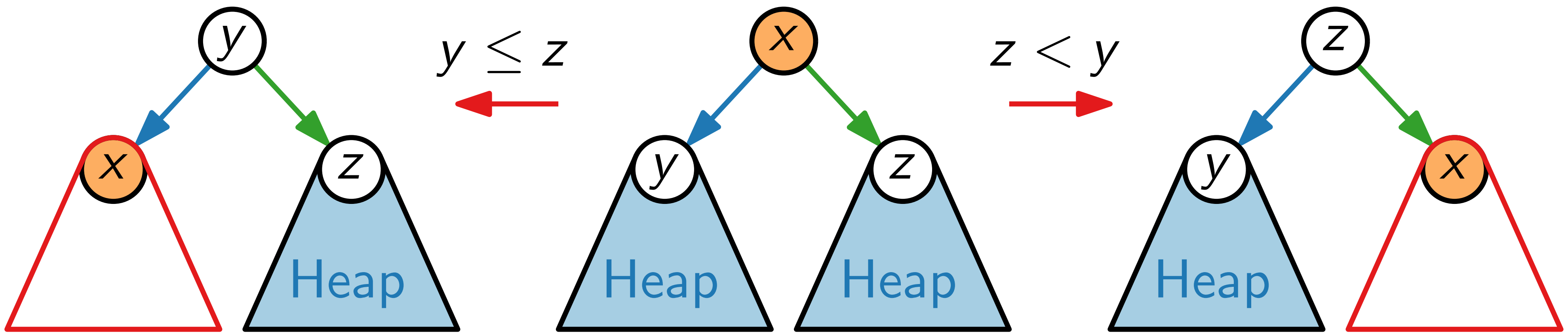
Elementaroperation

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



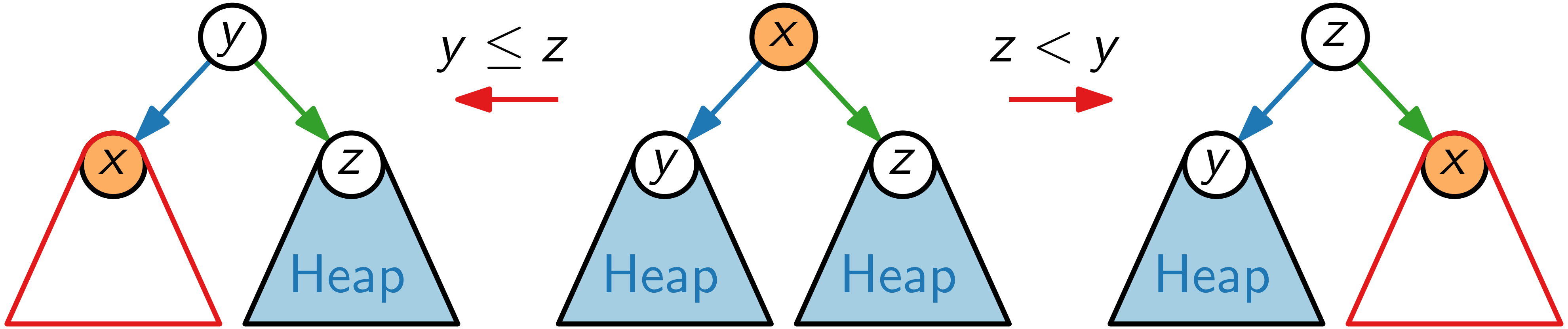
Elementaroperation

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



Elementaroperation

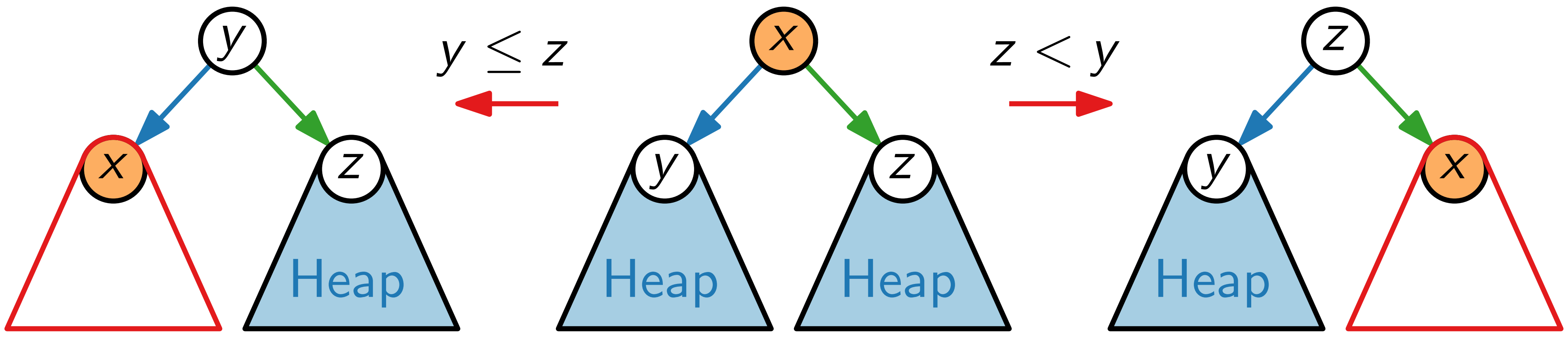
„Versickere" x, falls x zu groß, d.h. falls $x > \min(y, z)$



MinHeapify(int[] A, index i)

Elementaroperation

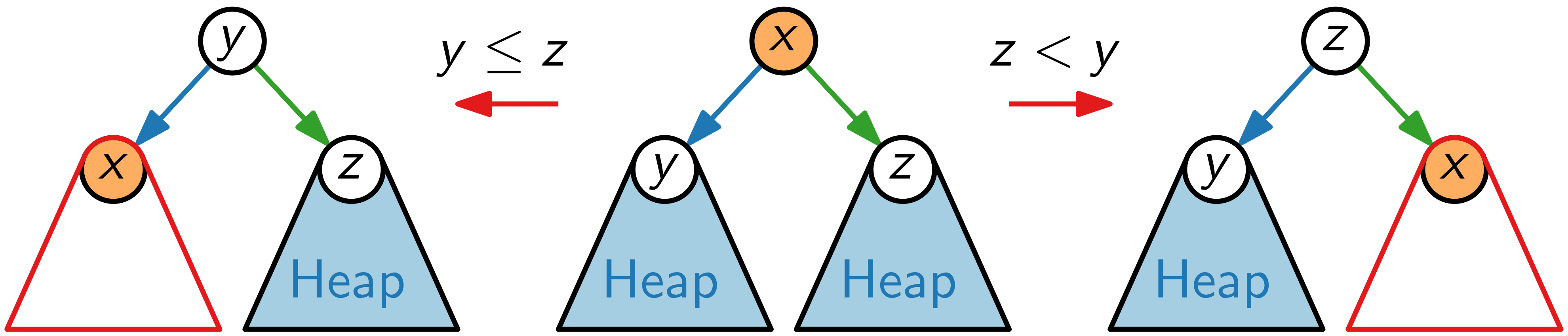
„Versickere“ x, falls x zu groß, d.h. falls $x > \min(y, z)$



MinHeapify(int[] A, index i)
ℓ = left(i); r = right(i)

Elementaroperation

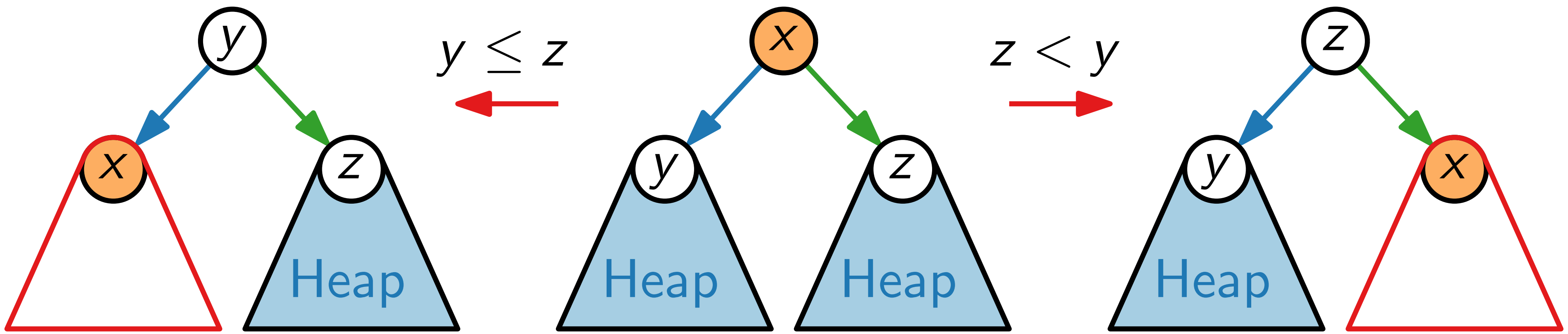
„Versickere" x, falls x zu groß, d.h. falls $x > \min(y, z)$



MinHeapify(int[] A, index i)
  $\ell = \text{LEFT}(i)$; $r = \text{RIGHT}(i)$
  $min = i$

Elementaroperation

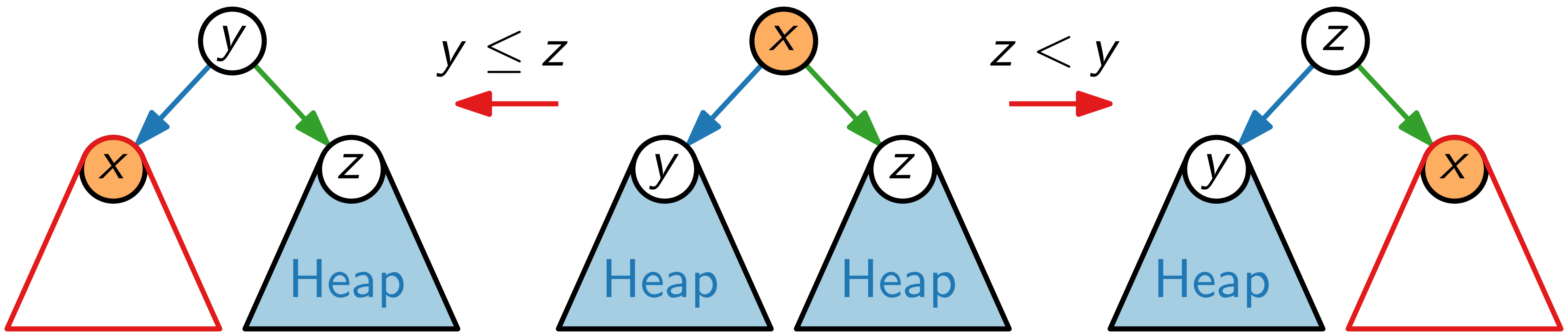
„Versickere“ x, falls x zu groß, d.h. falls $x > \min(y, z)$



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

Elementaroperation

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



$\textsc{MinHeapify}(\text{int[] } A, \text{index } i)$

$\ell = \textsc{left}(i)$; $r = \textsc{right}(i)$

$min = i$

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

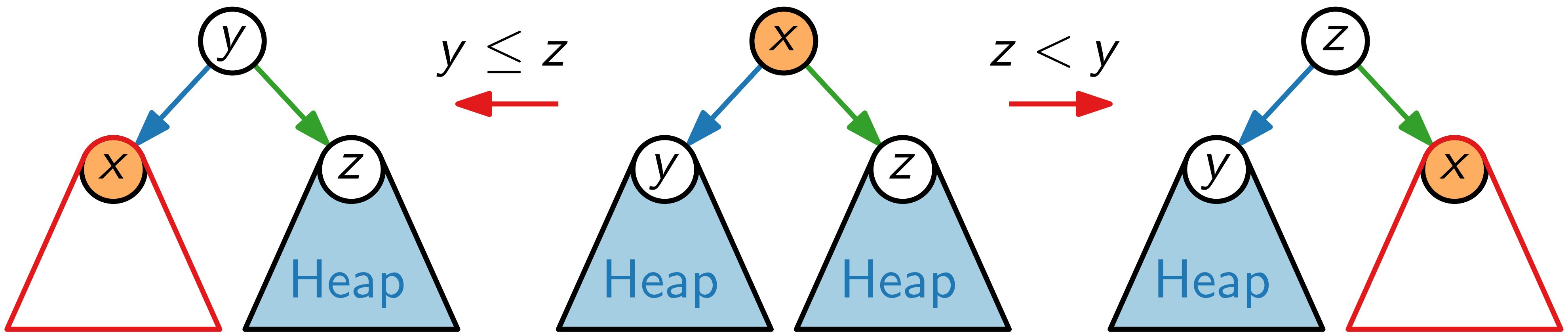
$\quad min = \ell$

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

$\quad min = r$

Elementaroperation

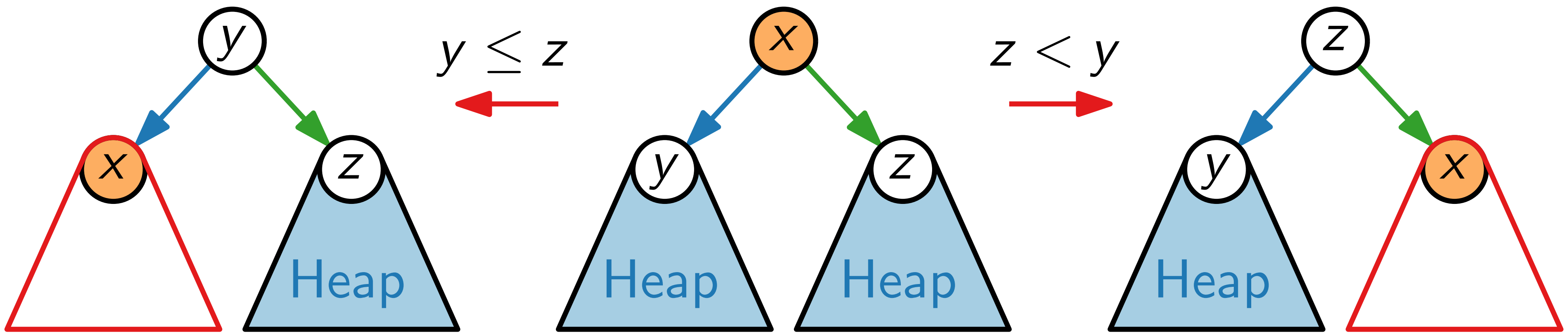
„Versickere" x, falls x zu groß, d.h. falls $x > \min(y, z)$



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

Elementaroperation

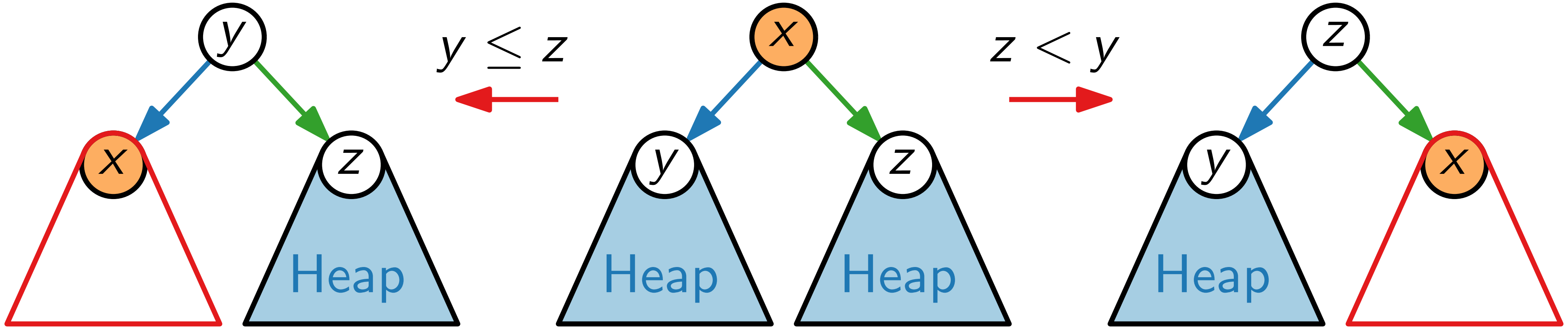
„Versickere" x, falls x zu groß, d.h. falls $x > \min(y, z)$



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

Elementaroperation

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

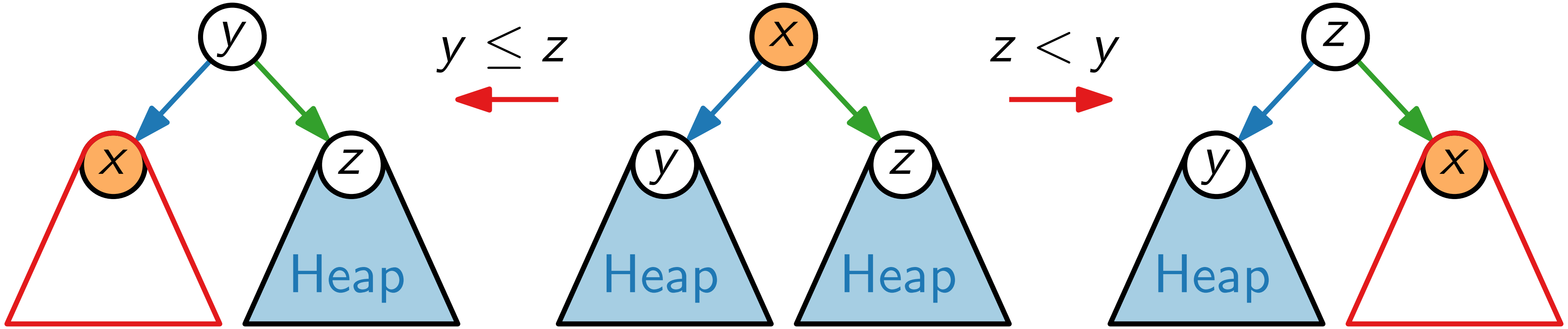


$\textsc{MinHeapify}(\text{int}[]\ A, \text{index}\ i)$
  $\ell = \text{LEFT}(i)$; $r = \text{RIGHT}(i)$
  $min = i$
  if $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \leq A.$ … $A[min]$ **then**
    min …
  if $min \neq$ …
    $A[i] \leftrightarrow A[min]$

Tausche $A[i]$ und $A[min]$:
$temp = A[i]$
$A[i] = A[min]$
$A[min] = temp$

Elementaroperation

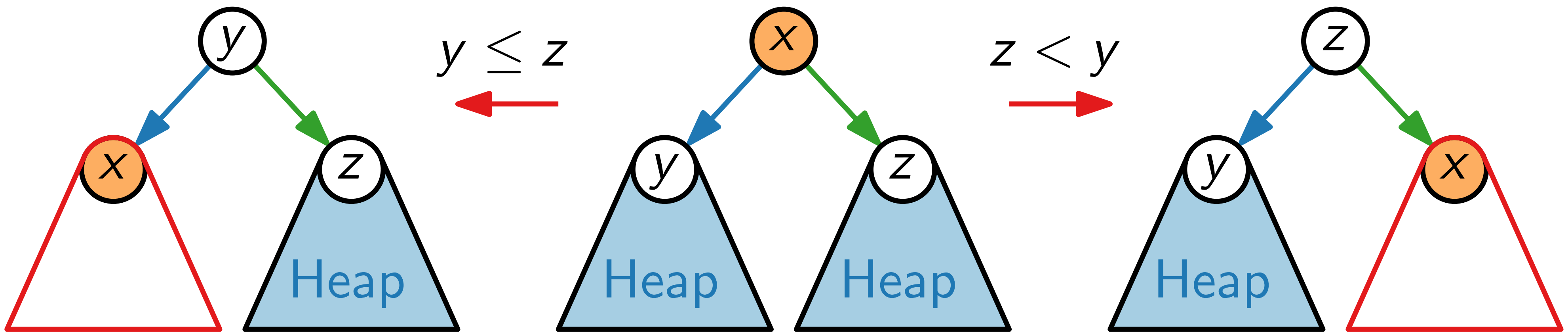
„Versickere" x, falls x zu groß, d.h. falls $x > \min(y, z)$



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

Elementaroperation

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

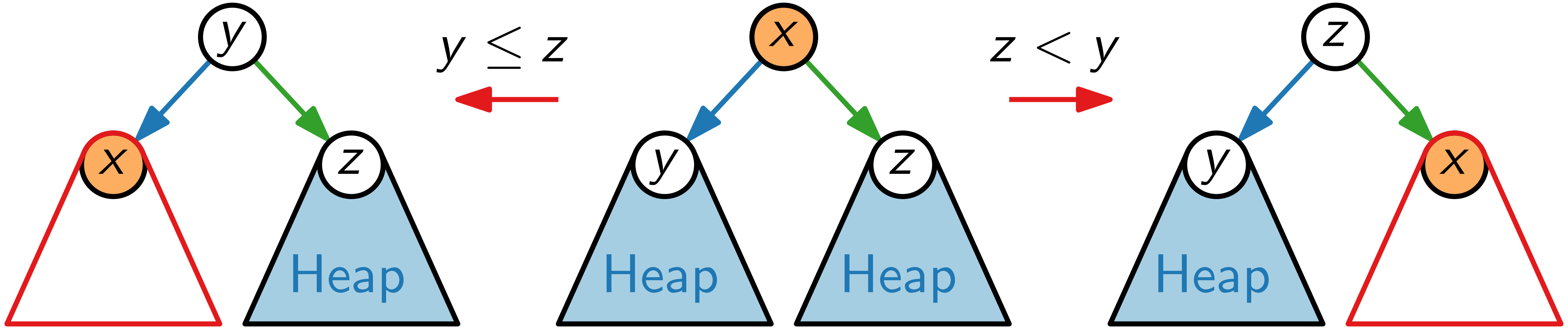


MinHeapify(int[] A, index i)
  ℓ = Left(i); r = Right(i)
  $min = i$
  if $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \leq A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
    $min = r$
  if $min \neq i$ **then**
    $A[i] \leftrightarrow A[min]$
    MinHeapify(A, min)

Lokale Strategie: top-down

Elementaroperation

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



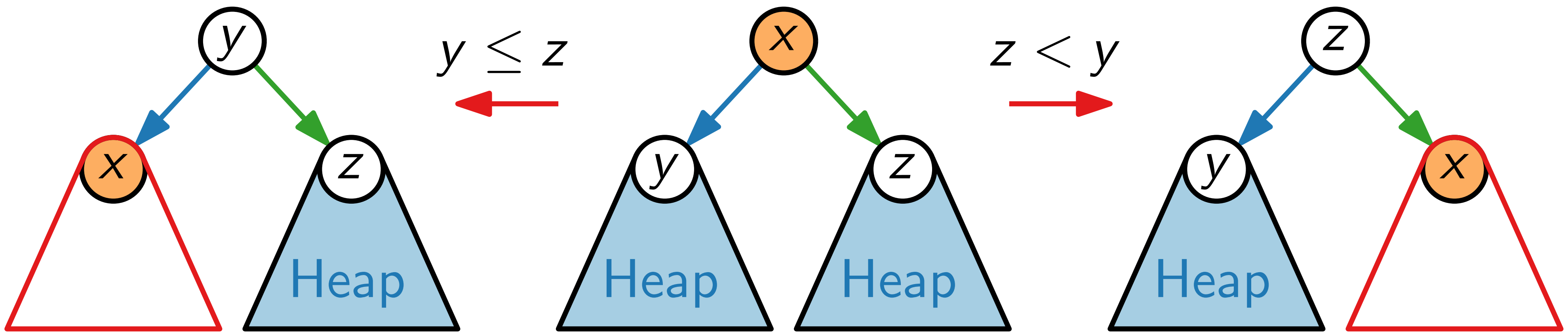
MinHeapify(int[] A, index i)
  $\ell = \text{LEFT}(i)$; $r = \text{RIGHT}(i)$
  $min = i$
  if $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \leq A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
    $min = r$
  if $min \neq i$ **then**
    $A[i] \leftrightarrow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

Lokale Strategie: top-down

Laufzeit?

Elementaroperation

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



$\textsc{MinHeapify}(\text{int}[]\ A, \text{index } i)$

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

  $min = i$

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

    $min = \ell$

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

    $min = r$

  if $min \neq i$ **then**

    $A[i] \leftrightarrow A[min]$

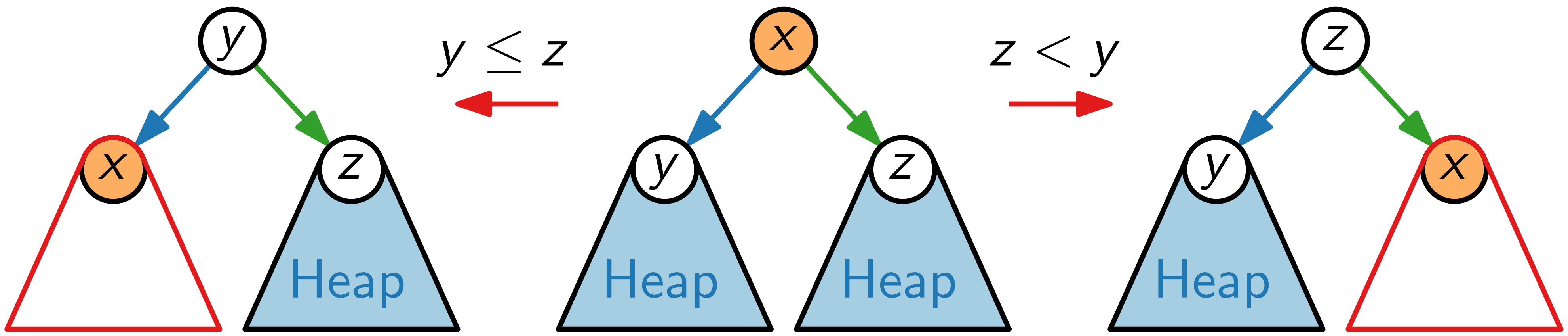
    $\textsc{MinHeapify}(A, min)$

Lokale Strategie: top-down

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

Elementaroperation

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



$\textsc{MinHeapify}(\text{int}[\,]\ A, \text{index } i)$
  $\ell = \textsc{left}(i)$; $r = \textsc{right}(i)$
  $min = i$
  if $\ell \le A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \le A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
    $min = r$
  if $min \ne i$ **then**
    $A[i] \leftrightarrow A[min]$
    $\textsc{MinHeapify}(A, min)$

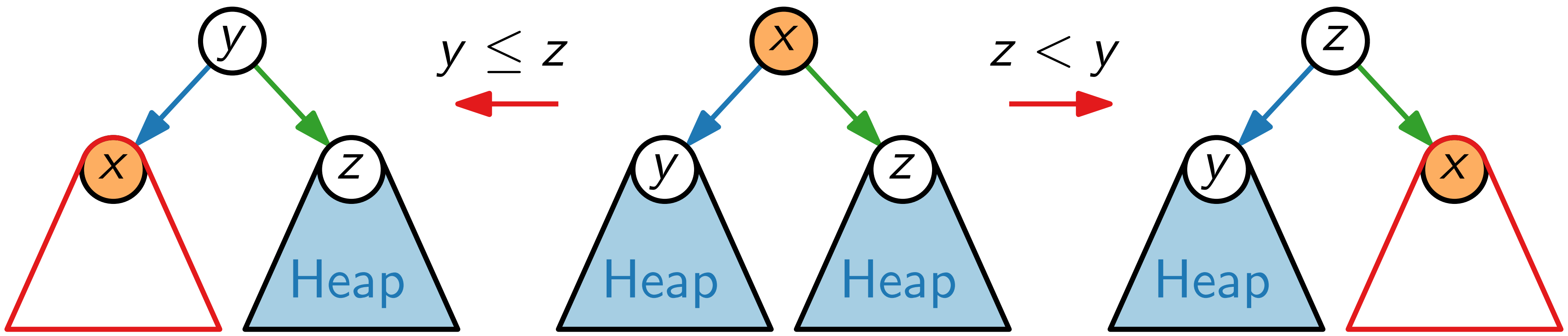
Lokale Strategie: top-down

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

$:=$ Anzahl der Tauschoperationen

Elementaroperation

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




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

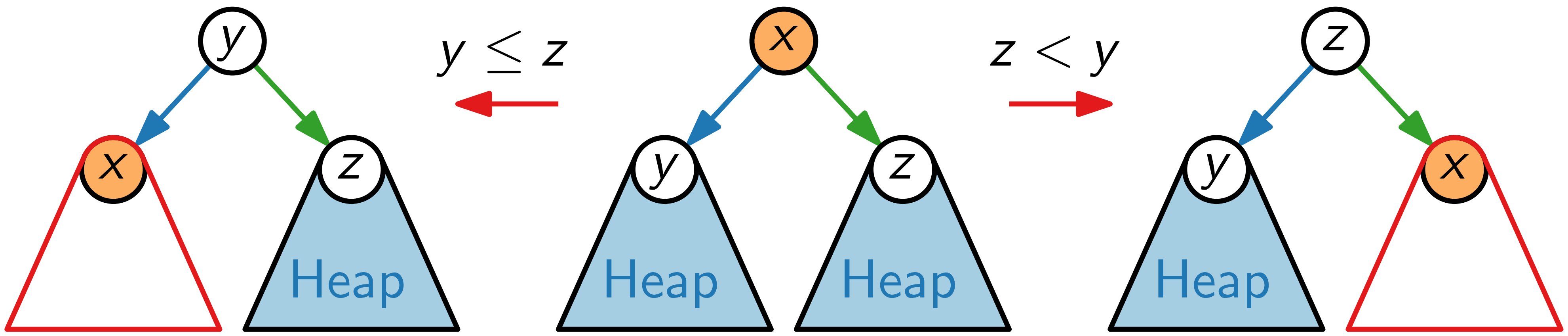
Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ 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 $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \leq A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
    $min = r$
  if $min \neq i$ **then**
    $A[i] \leftrightarrow A[min]$
    MinHeapify(A, min)

Lokale Strategie: top-down

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

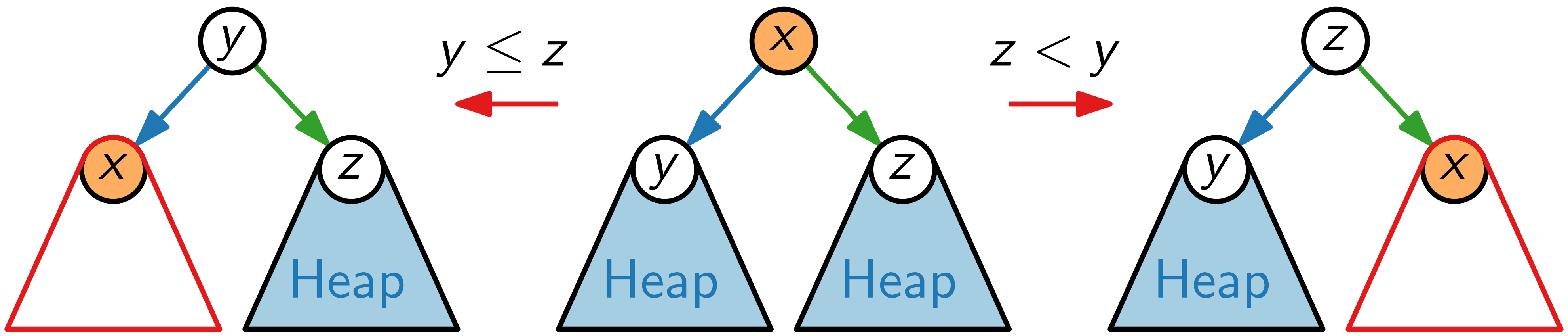
$:=$ Anzahl der Tauschoperationen

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

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

Lokale Strategie: top-down

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

$:=$ Anzahl der Tauschoperationen

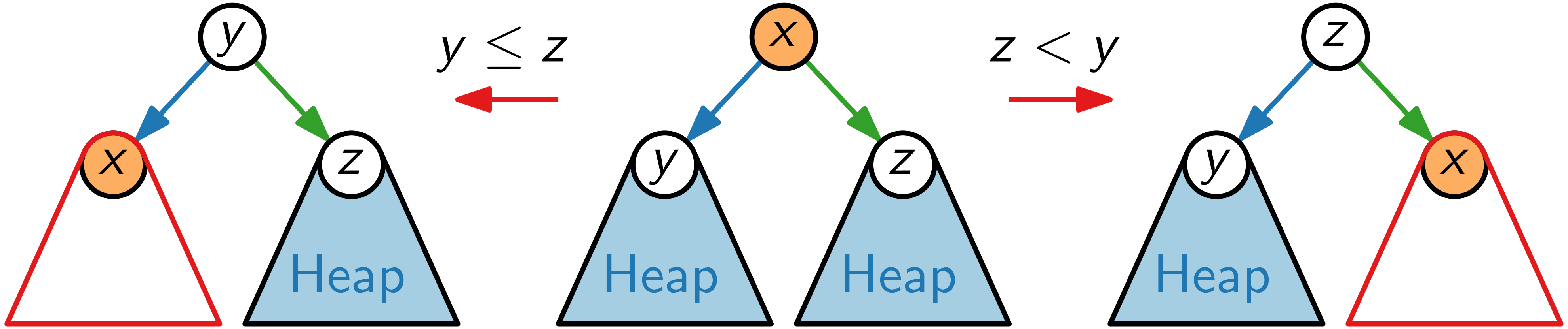
$\leq$ Länge des Weges von Knoten i zu einem Blatt

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

$:=$ Anzahl der Tauschoperationen

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

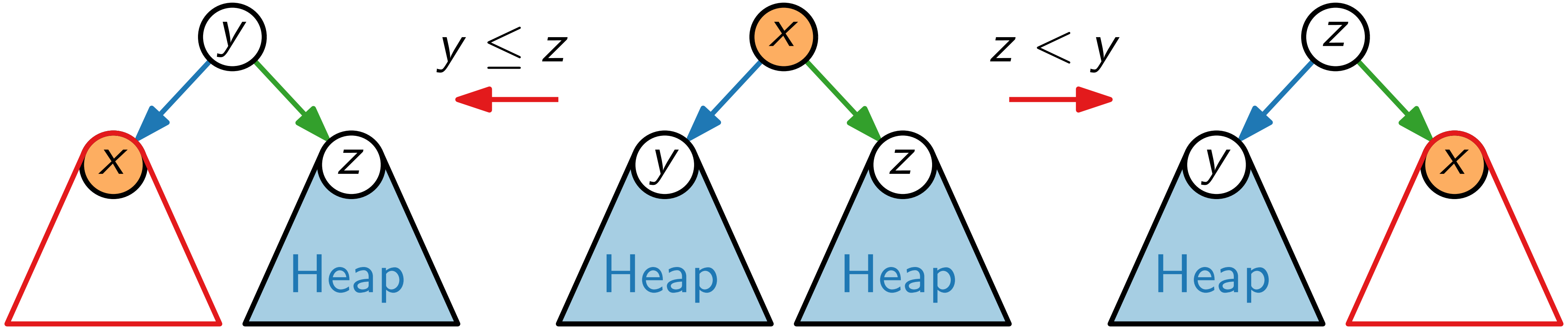
$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$

Elementaroperation

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



$\textsc{MinHeapify}(\text{int}[]\ A, \text{index } i)$
- $\ell = \textsc{left}(i)$; $r = \textsc{right}(i)$
- $min = i$
- **if** $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
  - $min = \ell$
- **if** $r \leq A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
  - $min = r$
- **if** $min \neq i$ **then**
  - $A[i] \leftrightarrow A[min]$
  - $\textsc{MinHeapify}(A, min)$

Lokale Strategie: top-down

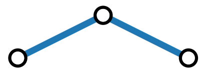
Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Tauschoperationen

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

$\leq$ Höhe von i im Heap

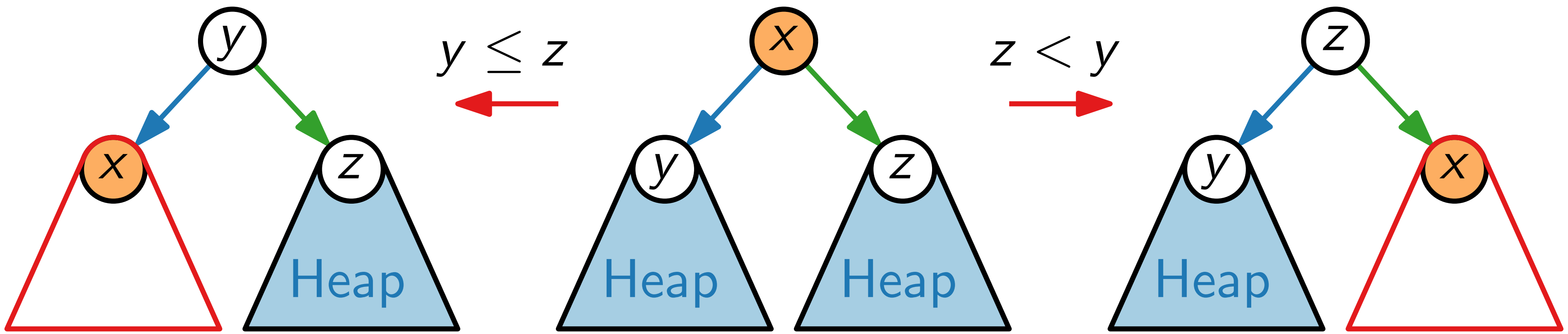
$\leq$ Höhe des Heaps



$\leq \lfloor \log_2 n \rfloor$

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

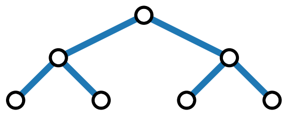
Laufzeit? $T_{\mathrm{MH}}(n, i)$

$:=$ Anzahl der Tauschoperationen

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

$\leq$ Höhe von i im Heap

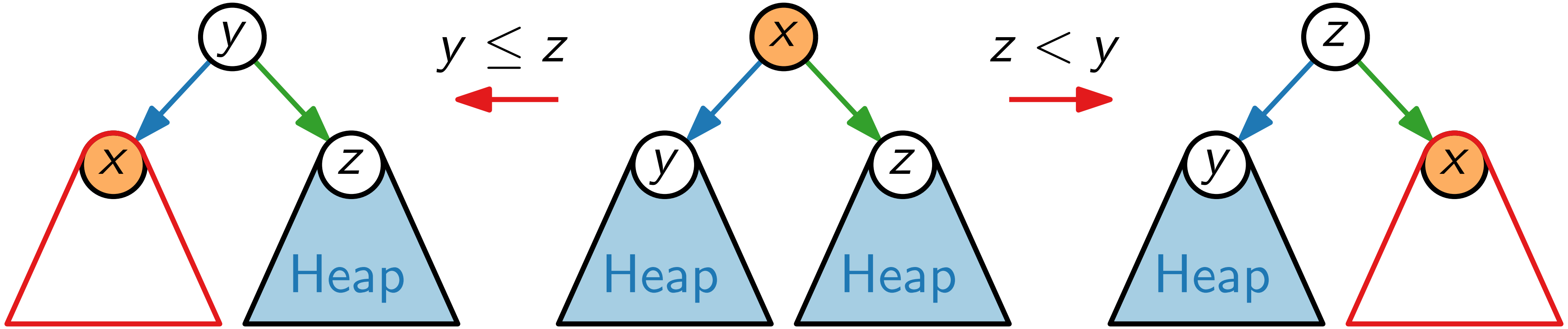
$\leq$ Höhe des Heaps



$\leq \lfloor \log_2 n \rfloor$

Elementaroperation

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



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

Lokale Strategie: top-down

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

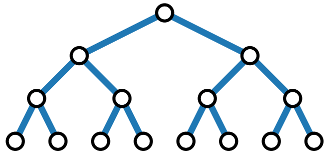
$:=$ Anzahl der Tauschoperationen

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

$\leq$ Höhe von i im Heap

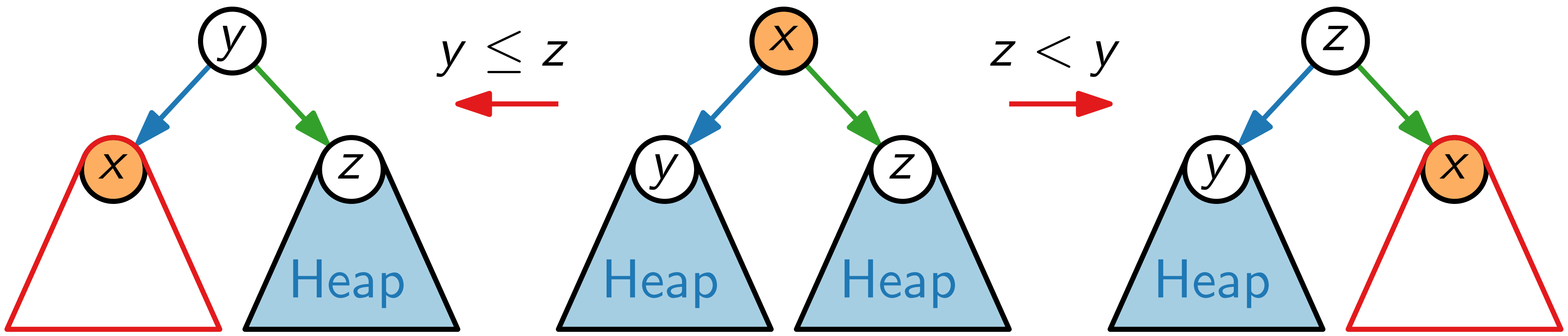
$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$



Elementaroperation

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



MinHeapify(int[] A, index i)

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

$min = i$

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

$min = \ell$

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

$min = r$

if $min \neq i$ **then**

$A[i] \leftrightarrow A[min]$

MinHeapify(A, min)

Lokale Strategie: top-down

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

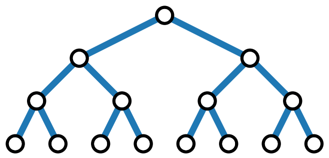
$:=$ Anzahl der Tauschoperationen

$\leq$ 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

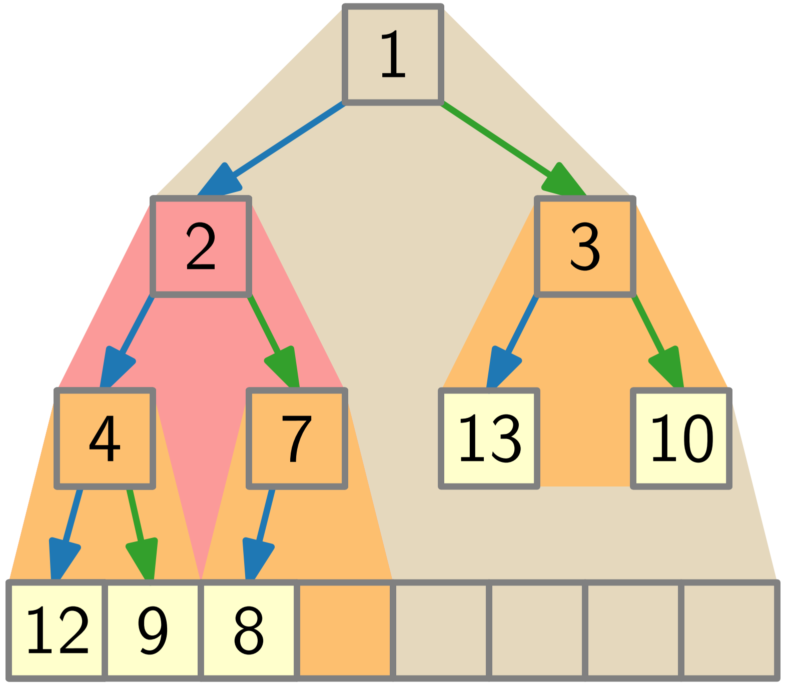
Laufzeit: $T_{\mathrm{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Das große Ganze

Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**



Das große Ganze

Lokale Strategie: **top-down**

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

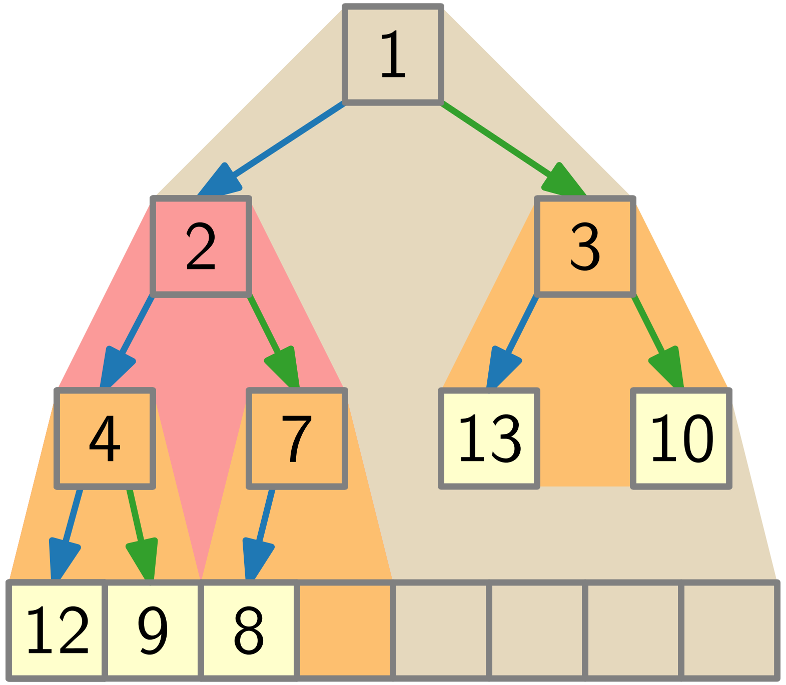
Globale Strategie: **bottom-up**

$\textsc{BuildMinHeap}(\text{int}[]\ A)$

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

for $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**

$\textsc{MinHeapify}(A, i)$



Das große Ganze

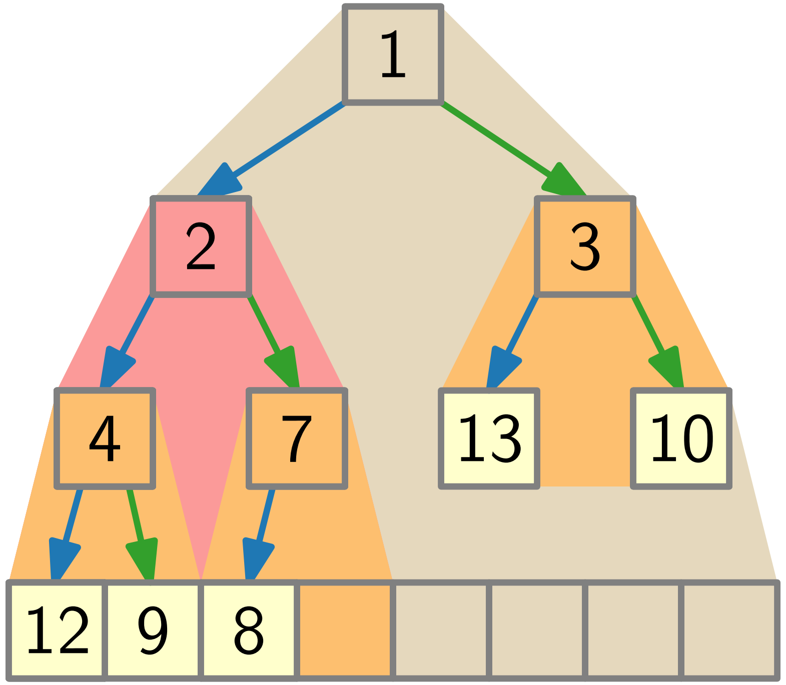
Lokale Strategie: top-down

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

Globale Strategie: bottom-up

BuildMinHeap(int[] A)
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

Laufzeit.



Das große Ganze

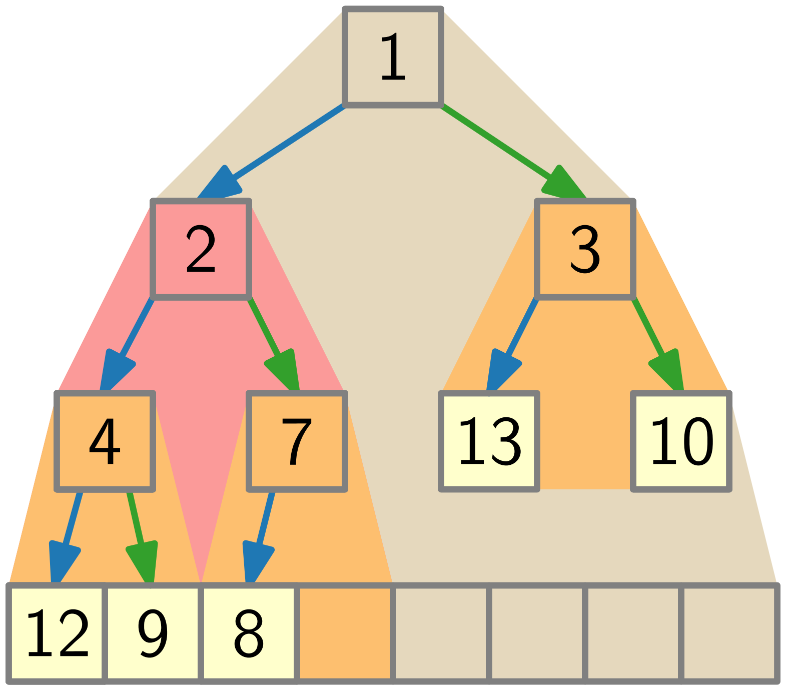
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

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

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



Das große Ganze

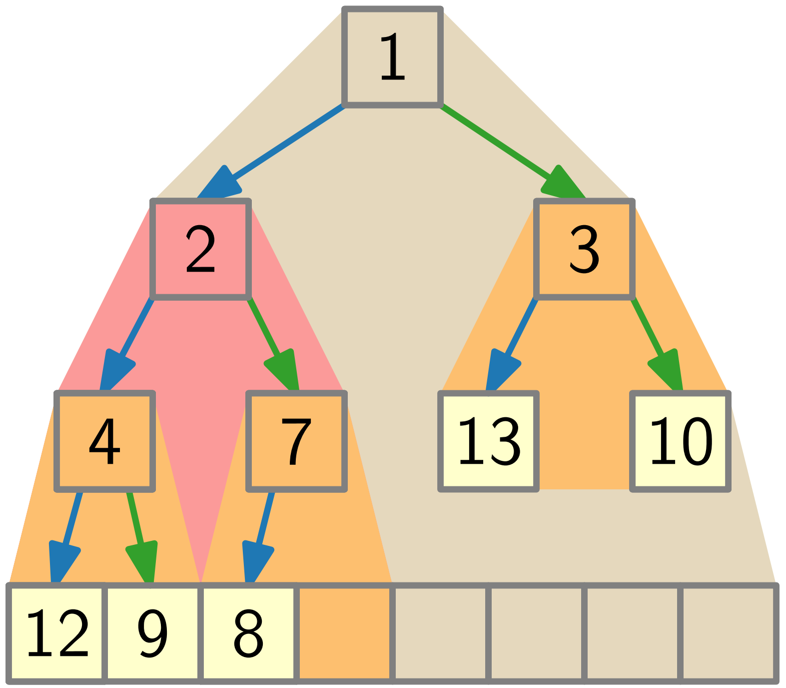
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

$\textsc{BuildMinHeap}(\text{int}[]\ A)$
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    $\textsc{MinHeapify}(A, i)$

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$



Das große Ganze

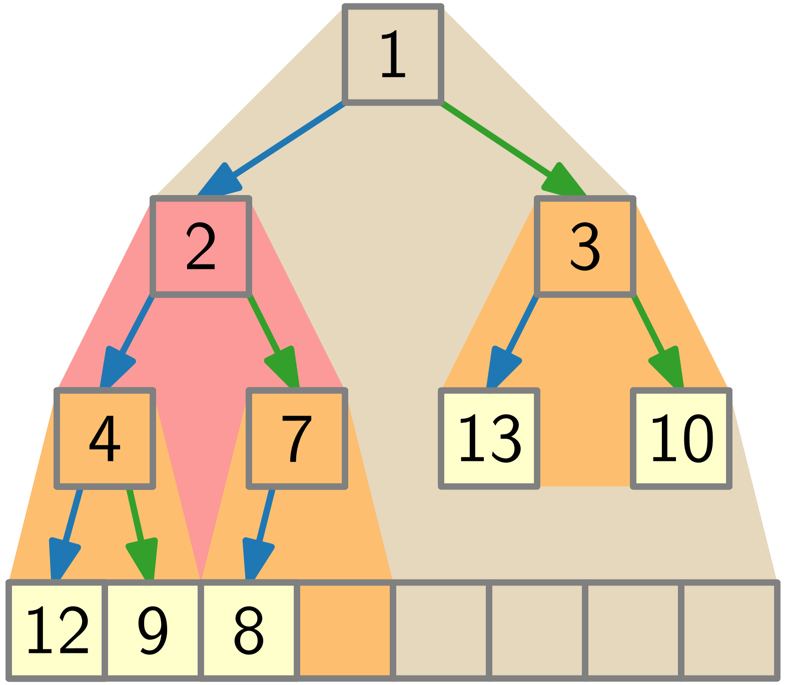
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

BuildMinHeap(int[] A)
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$
$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n, i)$



Das große Ganze

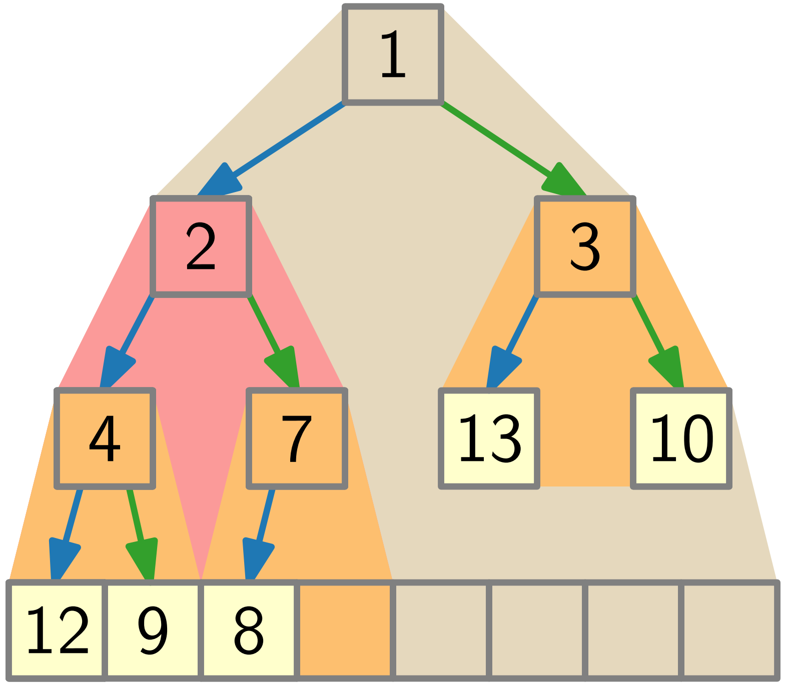
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

BuildMinHeap(int[] A)
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$
$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n, i)$
$\approx$



Das große Ganze

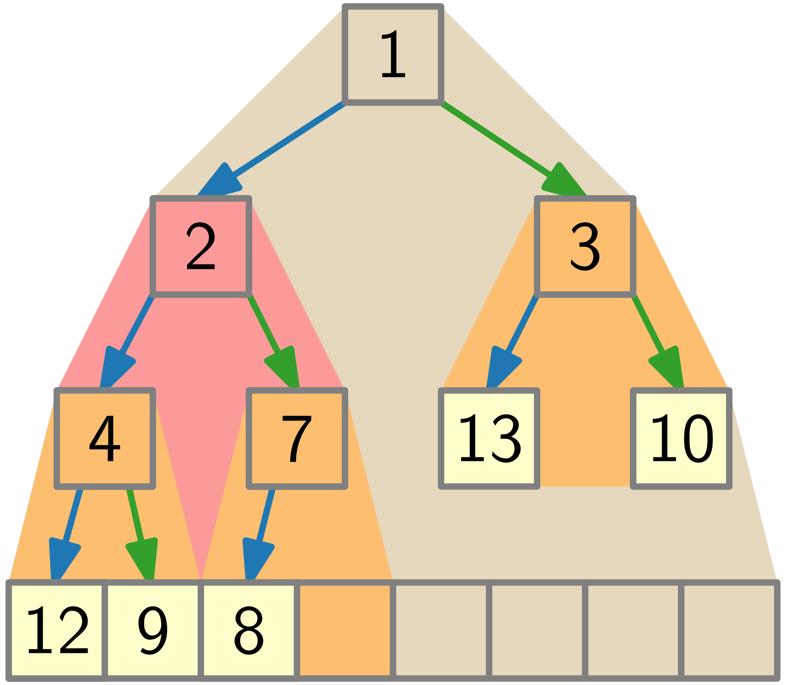
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

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

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$
$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n, i)$
$\approx \frac{n}{2} \cdot 0 +$



Das große Ganze

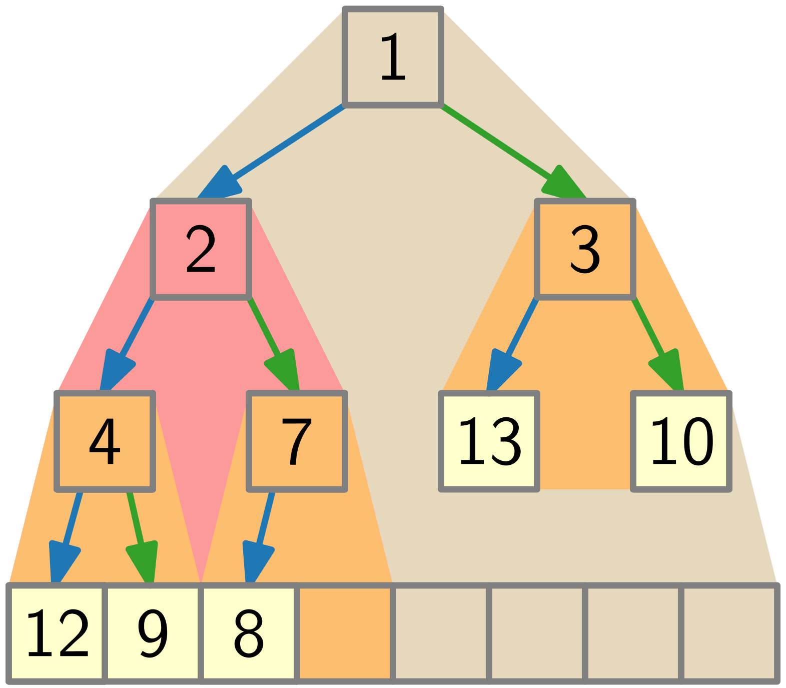
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

$\textsc{BuildMinHeap}(\text{int[] } A)$
  $A.\textit{heap-size} = A.\textit{length}$
  for $i = \lfloor A.\textit{length}/2 \rfloor$ **downto** 1 **do**
    $\textsc{MinHeapify}(A, i)$

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$
$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n,i)$
$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 +$



Das große Ganze

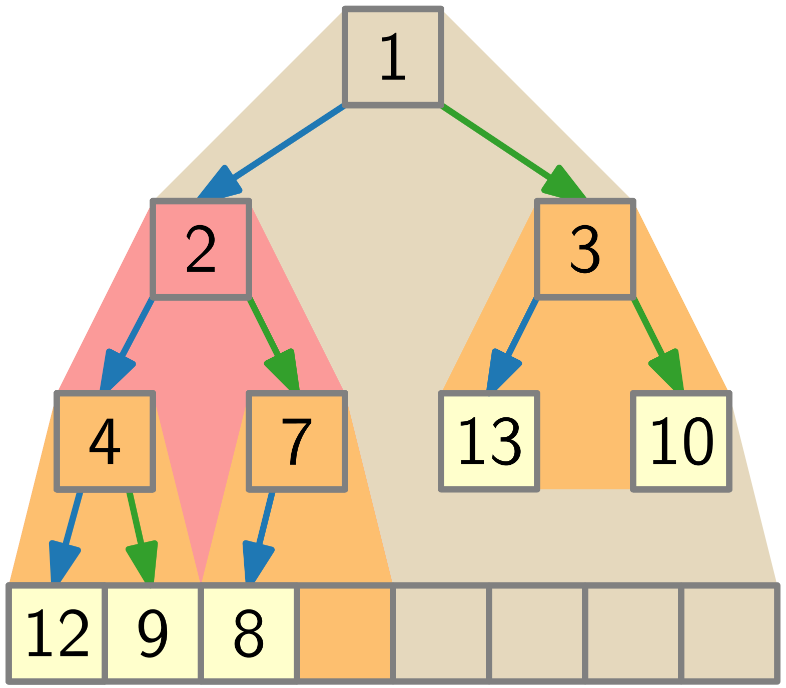
Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

$\textsc{BuildMinHeap}(\mathrm{int}[]\ A)$
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    $\textsc{MinHeapify}(A, i)$

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$
$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\mathrm{MH}}(n, i)$
$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 +$



Das große Ganze

Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

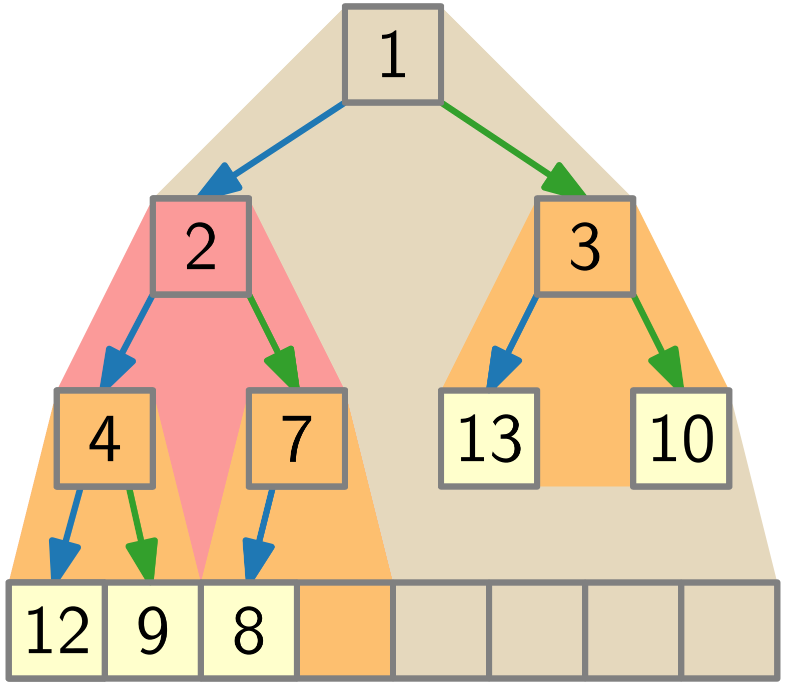
BuildMinHeap(int[] A)
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

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

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

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

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



Das große Ganze

Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

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

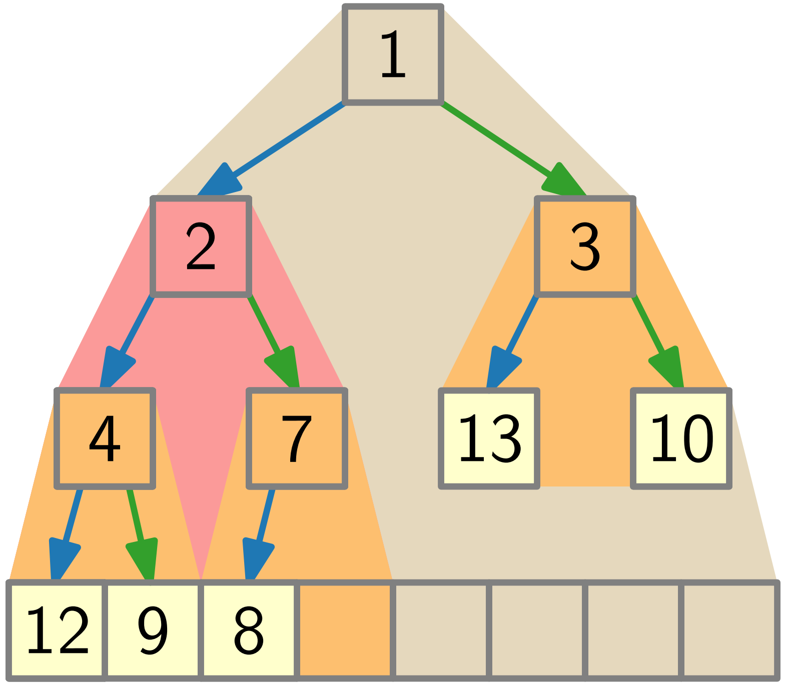
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

$$= n \cdot$$



Das große Ganze

Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

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

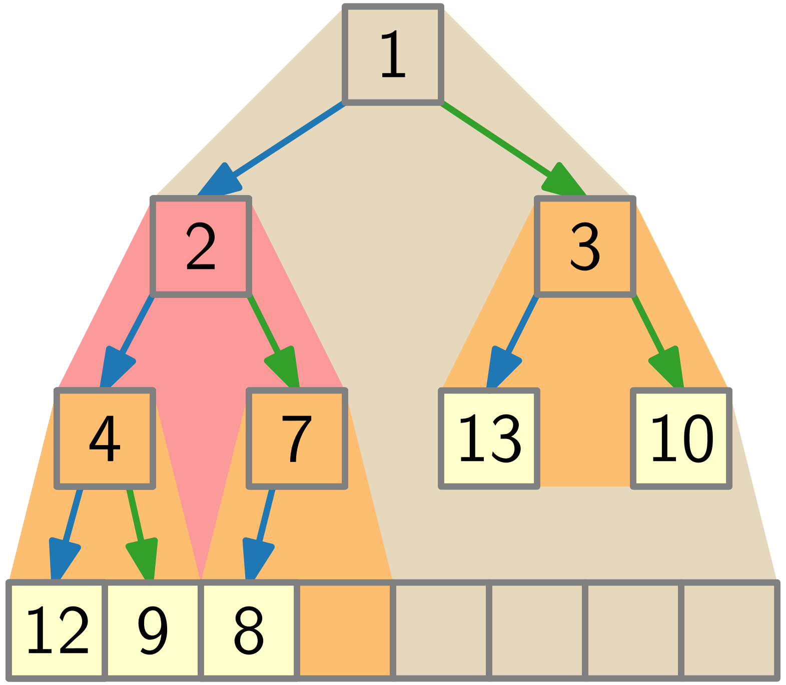
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

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



Das große Ganze

Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

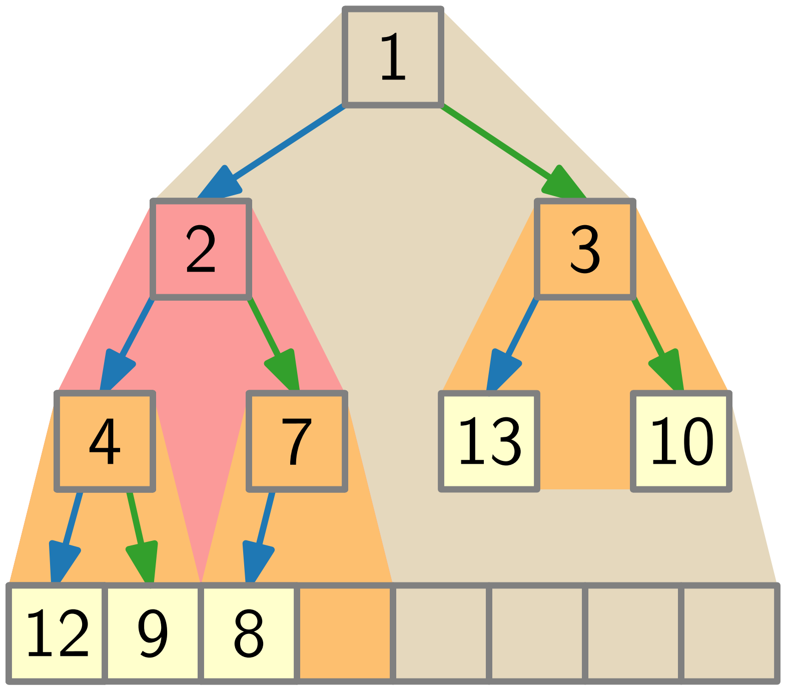
BuildMinHeap(int[] A)
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\text{BMH}}(n) =$

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

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

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



Das große Ganze

Lokale Strategie: top-down

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

Globale Strategie: bottom-up

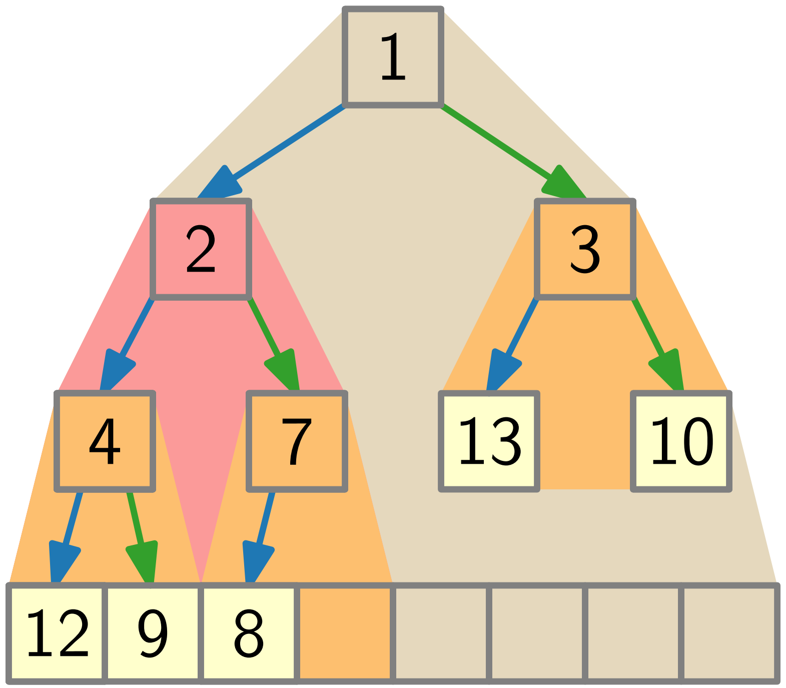
BuildMinHeap(int[] A)
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

Laufzeit. grob: $\mathcal{O}(n \log n)$
genauer: $T_{\mathrm{BMH}}(n) =$

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

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

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



Das große Ganze

Lokale Strategie: **top-down**

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

Globale Strategie: **bottom-up**

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

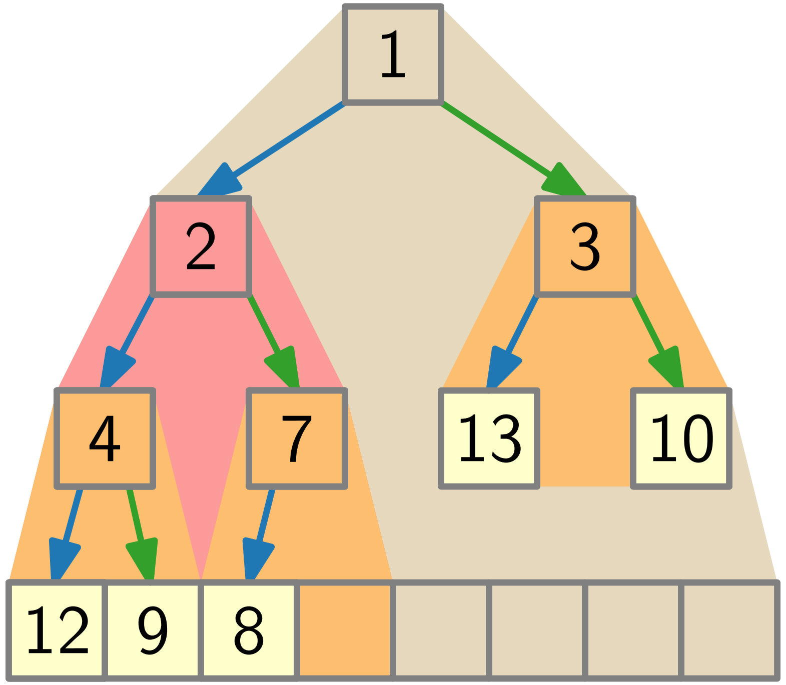
Laufzeit. grob: $\mathcal{O}(n \log n)$

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

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

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

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



Fortsetzung Laufzeitanalyse

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

Fortsetzung Laufzeitanalyse

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

Fortsetzung Laufzeitanalyse

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

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

Fortsetzung Laufzeitanalyse

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Wir hätten gerne:

Fortsetzung Laufzeitanalyse

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

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

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Wir hätten gerne: $$\sum_{i=1}^{\infty} i q^{i-1} = \quad \textbf{?}$$

Fortsetzung Laufzeitanalyse

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

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

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

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

Fortsetzung Laufzeitanalyse

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

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

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

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

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

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

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

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

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

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!

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

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Quotientenregel:

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

ableiten!

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

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

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

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

Quotientenregel:

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

ableiten!

ableiten!

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

$$= \frac{1}{(1-q)^2}$$

Fortsetzung Laufzeitanalyse

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

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

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

ableiten!

Quotientenregel:

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

ableiten!

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

$$= \frac{1}{(1-q)^2}$$

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

Fortsetzung Laufzeitanalyse

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

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

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

Quotientenregel:

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

ableiten!

ableiten!

Wir hätten gerne:

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Quotientenregel:

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

ableiten!

ableiten!

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Quotientenregel:

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

ableiten!

ableiten!

Wir hätten gerne:

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

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

Fortsetzung Laufzeitanalyse

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

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

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

Quotientenregel:

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

ableiten!

ableiten!

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

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

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

Übung Heap-Aufbau

Aufgabe. Bauen Sie einen Heap mit BuildMinHeap!

10	1	9	4	3	7	8	2	6	5

$\longrightarrow$

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

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

Übung Heap-Aufbau

Aufgabe. Bauen Sie einen Heap mit BUILDMINHEAP!

10	1	9	4	3	7	8	2	6	5

$\longrightarrow$

1	2	7	4	3	9	8	10	6	5

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

BUILDMINHEAP(int[] A)
  $A.heap\text{-}size = A.length$
  for $i = \lfloor A.length/2 \rfloor$ **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.

$\textsc{FindMin}()$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: **Prioritätsschlange**

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

$\textsc{FindMin}()$
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.

$\textsc{FindMin}()$
return $A[1]$

$\textsc{ExtractMin}()$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

$\textsc{FindMin}()$
  return $A[1]$

$\textsc{ExtractMin}()$
  if $A.heap\text{-}size < 1$ **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.

$\textsc{FindMin}()$
return $A[1]$

$\textsc{ExtractMin}()$
if $A.heap\text{-}size < 1$ **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 $A.heap\text{-}size < 1$ **then**
  error „Heap underflow“
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$

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 $A.heap\text{-}size < 1$ **then**
    error „Heap underflow"
  $min = A[1]$
  $A[1] = A[A.heap\text{-}size]$
  $A.heap\text{-}size{-}{-}$

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 $A.heap\text{-}size < 1$ **then**
    error „Heap underflow“
  $min = A[1]$
  $A[1] = A[A.heap\text{-}size]$
  $A.heap\text{-}size{-}{-}$
  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.

$\textsc{FindMin}()$
return $A[1]$

$\textsc{ExtractMin}()$
if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow“
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size--$
$\textsc{MinHeapify}(A, 1)$
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 $A.heap\text{-}size < 1$ **then**
  error „Heap underflow"
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size--$
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.

$\textsc{FindMin}()$
  return $A[1]$

$\textsc{ExtractMin}()$
  if $A.\mathit{heap\text{-}size} < 1$ **then**
    error „Heap underflow“
  $\mathit{min} = A[1]$
  $A[1] = A[A.\mathit{heap\text{-}size}]$
  $A.\mathit{heap\text{-}size}{-}{-}$
  $\textsc{MinHeapify}(A, 1)$
  return min

$\textsc{DecreaseKey}$(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.

$\textsc{FindMin}()$
return $A[1]$

$\textsc{ExtractMin}()$
if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow"
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}$(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.

$\textsc{FindMin}()$
  return $A[1]$

$\textsc{ExtractMin}()$
  if $A.heap\text{-}size < 1$ **then**
    error „Heap underflow"
  $min = A[1]$
  $A[1] = A[A.heap\text{-}size]$
  $A.heap\text{-}size{-}{-}$
  $\textsc{MinHeapify}(A, 1)$
  return min

$\textsc{DecreaseKey}$(index i, prio. p)
  if $p > A[i]$ **then error** „prio. too large"
  $A[i] = p$
  while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

$\textsc{FindMin}()$
  return $A[1]$

$\textsc{ExtractMin}()$
  if $A.heap\text{-}size < 1$ **then**
    error „Heap underflow"
  $min = A[1]$
  $A[1] = A[A.heap\text{-}size]$
  $A.heap\text{-}size{-}{-}$
  $\textsc{MinHeapify}(A, 1)$
  return min

$\textsc{DecreaseKey}$(index i, prio. p)
  if $p > A[i]$ **then error** „prio. too large"
  $A[i] = p$
  while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
    $A[i] \leftrightarrow A[\textsc{parent}(i)]$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: **Prioritätsschlange**

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

$\textsc{FindMin}()$
  return $A[1]$

$\textsc{ExtractMin}()$
  if $A.heap\text{-}size < 1$ **then**
    error „Heap underflow“
  $min = A[1]$
  $A[1] = A[A.heap\text{-}size]$
  $A.heap\text{-}size{-}{-}$
  $\textsc{MinHeapify}(A, 1)$
  return min

$\textsc{DecreaseKey}$(index i, prio. p)
  if $p > A[i]$ **then error** „prio. too large“
  $A[i] = p$
  while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
    $A[i] \leftrightarrow A[\textsc{parent}(i)]$
    $i = \textsc{parent}(i)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

$\textsc{FindMin}()$

return $A[1]$

$\textsc{ExtractMin}()$

if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow“
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}$(index i, prio. p)

if $p > A[i]$ **then error** „prio. too large“
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}$(priorität p)

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: **Prioritätsschlange**

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

$\textsc{FindMin}()$
  return $A[1]$

$\textsc{ExtractMin}()$
  if $A.\mathit{heap\text{-}size} < 1$ **then**
    error „Heap underflow“
  $\mathit{min} = A[1]$
  $A[1] = A[A.\mathit{heap\text{-}size}]$
  $A.\mathit{heap\text{-}size}{-}{-}$
  $\textsc{MinHeapify}(A, 1)$
  return min

$\textsc{DecreaseKey}$(index i, prio. p)
  if $p > A[i]$ **then error** „prio. too large“
  $A[i] = p$
  while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
    $A[i] \leftrightarrow A[\textsc{parent}(i)]$
    $i = \textsc{parent}(i)$

$\textsc{Insert}$(priorität p)
  $A.\mathit{heap\text{-}size}{+}{+}$
  if $A.\mathit{heap\text{-}size} > A.\mathit{length}$ **then error...**

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

$\textsc{FindMin}()$
return $A[1]$

$\textsc{ExtractMin}()$
if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow“
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}$(index i, prio. p)
if $p > A[i]$ **then error** „prio. too large“
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

FindMin()
  return $A[1]$

ExtractMin()
  if $A.heap\text{-}size < 1$ **then**
    error „Heap underflow"
  $min = A[1]$
  $A[1] = A[A.heap\text{-}size]$
  $A.heap\text{-}size{-}{-}$
  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** $A[\text{Parent}(i)] > A[i]$
    $A[i] \leftrightarrow A[\text{Parent}(i)]$
    $i = \text{Parent}(i)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Laufzeiten?

$\textsc{FindMin}()$ — $\mathcal{O}(\quad)$

return $A[1]$

$\textsc{ExtractMin}()$ — $\mathcal{O}(\quad)$

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

$\textsc{DecreaseKey}$(index i, prio. p) — $\mathcal{O}(\quad)$

if $p > A[i]$ **then error** „prio. too large"
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}$(priorität p) — $\mathcal{O}(\quad)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Laufzeiten?

$\textsc{FindMin}()$ — $\mathcal{O}(1)$

return $A[1]$

$\textsc{ExtractMin}()$ — $\mathcal{O}(\quad)$

if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow“
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}(\text{index } i, \text{prio. } p)$ — $\mathcal{O}(\quad)$

if $p > A[i]$ **then error** „prio. too large“
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}(\text{priorität } p)$ — $\mathcal{O}(\quad)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Laufzeiten?

$\textsc{FindMin}()$ $\mathcal{O}(\ 1\ )$

return $A[1]$

$\textsc{ExtractMin}()$ $\mathcal{O}(\quad)$

if $A.\mathit{heap\text{-}size} < 1$ **then**
  error „Heap underflow“
$\mathit{min} = A[1]$
$A[1] = A[A.\mathit{heap\text{-}size}]$
$A.\mathit{heap\text{-}size}{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

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

if $p > A[i]$ **then error** „prio. too large“
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}$(priorität p) $\mathcal{O}(\quad)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Laufzeiten?

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] ↔ 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] = ∞
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.

Laufzeiten?

$\textsc{FindMin}()$ — $\mathcal{O}(1)$

return $A[1]$

$\textsc{ExtractMin}()$ — $\mathcal{O}(\log n)$

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

$\textsc{DecreaseKey}(\text{index } i, \text{prio. } p)$ — $\mathcal{O}(\quad)$

if $p > A[i]$ **then error** „prio. too large"
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}(\text{priorität } p)$ — $\mathcal{O}(\quad)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: **Prioritätsschlange**

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

Laufzeiten?

$\textsc{FindMin}()$ — $\mathcal{O}(1)$

return $A[1]$

$\textsc{ExtractMin}()$ — $\mathcal{O}(\log n)$

if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow“
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}$(index i, prio. p) — $\mathcal{O}(\log n)$

if $p > A[i]$ **then error** „prio. too large“
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}$(priorität p) — $\mathcal{O}(\quad)$

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

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Laufzeiten?

$\textsc{FindMin}()$ $\boxed{\mathcal{O}(\;1\;)}$
return $A[1]$

$\textsc{ExtractMin}()$ $\boxed{\mathcal{O}(\log n)}$
if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow"
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}$(index i, prio. p) $\boxed{\mathcal{O}(\log n)}$
if $p > A[i]$ **then error** „prio. too large"
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}$(priorität p) $\boxed{\mathcal{O}(\quad)}$
$A.heap\text{-}size{+}{+}$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = \infty$
$\textsc{DecreaseKey}(A.heap\text{-}size, p)$

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

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

Laufzeiten?

$\textsc{FindMin}()$ — $\mathcal{O}(1)$

return $A[1]$

$\textsc{ExtractMin}()$ — $\mathcal{O}(\log n)$

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

$\textsc{DecreaseKey}(\text{index } i, \text{prio. } p)$ — $\mathcal{O}(\log n)$

if $p > A[i]$ **then error** „prio. too large"
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}(\text{priorität } p)$ — $\mathcal{O}(\log n)$

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

HeapSort

Idee:

HeapSort

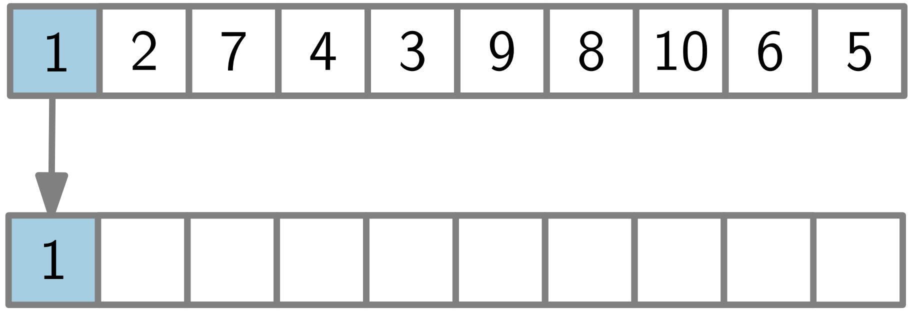
Idee:

- ExtractMin() gibt kleinstes Heap-Element aus.

1	2	7	4	3	9	8	10	6	5

HeapSort

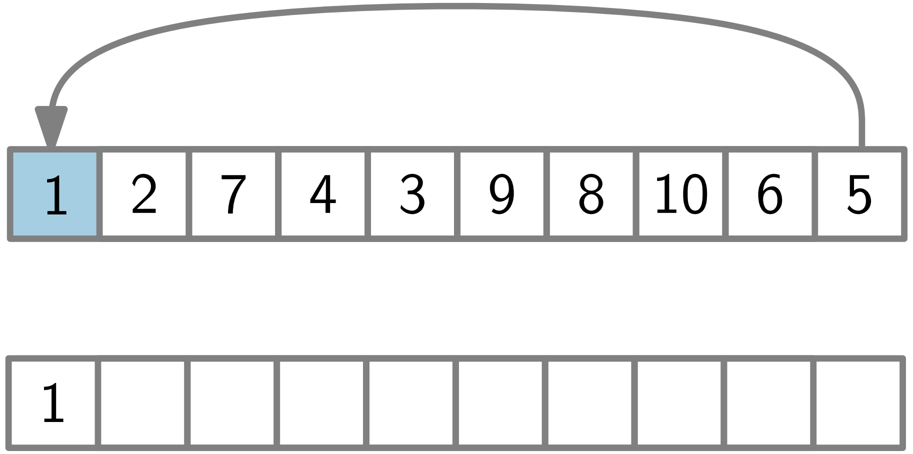
Idee: ■ ExtractMin() gibt kleinstes Heap-Element aus.



HeapSort

Idee:

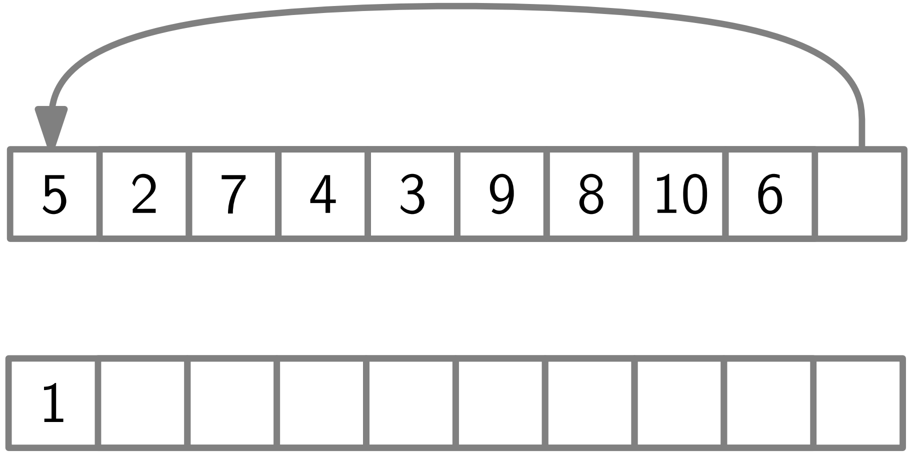
- ExtractMin() gibt kleinstes Heap-Element aus.



HeapSort

Idee:

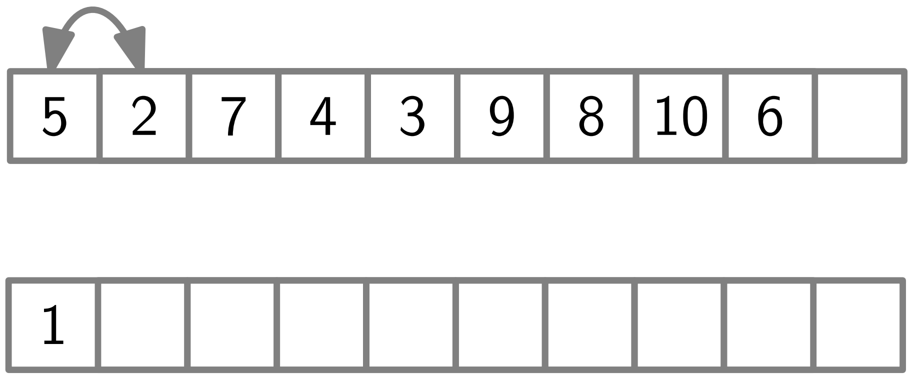
- ExtractMin() gibt kleinstes Heap-Element aus.



HeapSort

Idee:

- ExtractMin() gibt kleinstes Heap-Element aus.



HeapSort

Idee: ▪ ExtractMin() gibt kleinstes Heap-Element aus.

2	5	7	4	3	9	8	10	6	

1									

HeapSort

Idee:

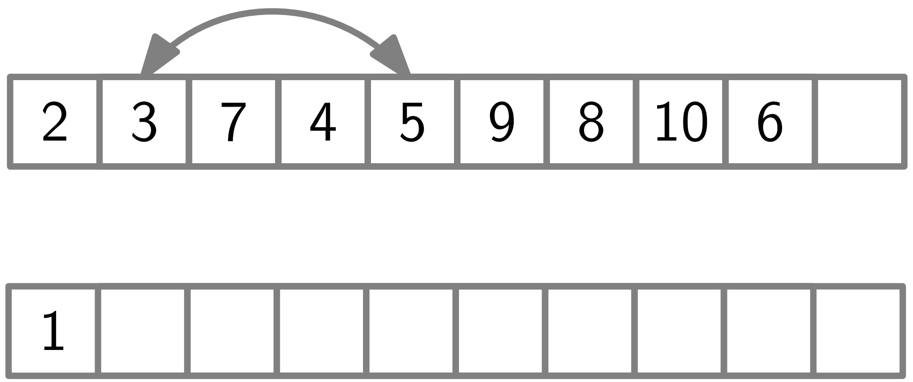
- EXTRACTMIN() gibt kleinstes Heap-Element aus.

2	5	7	4	3	9	8	10	6	

1									

HeapSort

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



HeapSort

Idee: ▪ ExtractMin() gibt kleinstes Heap-Element aus.

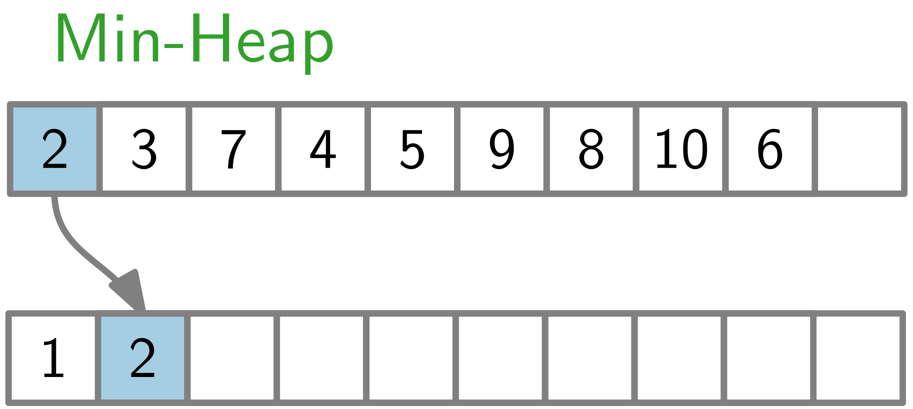
Min-Heap

2	3	7	4	5	9	8	10	6	

1									

HeapSort

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

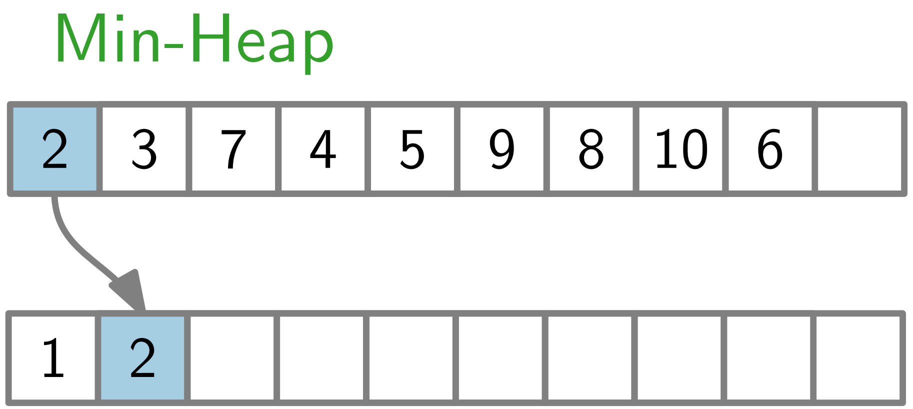


HeapSort

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

int[] HEAPSORT(int[] A)

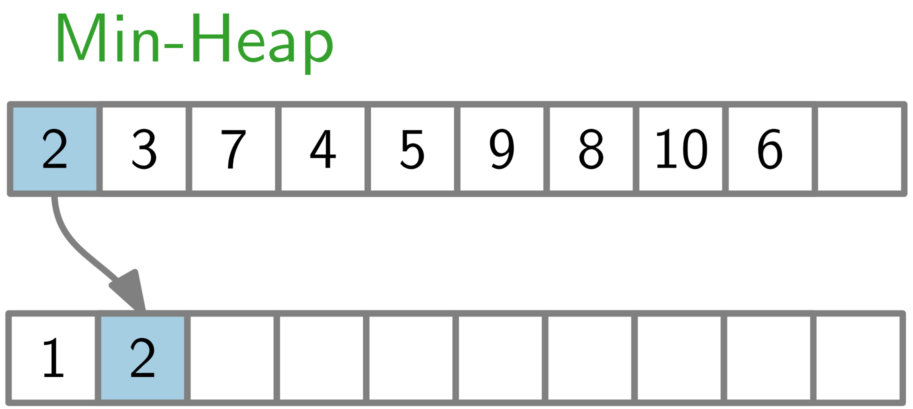
Schreiben *Sie* den Pseudocode.
Verwenden Sie
BUILDMINHEAP und
EXTRACTMIN.



HeapSort

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

int[] HeapSort(int[] A)
  BuildMinHeap(A)

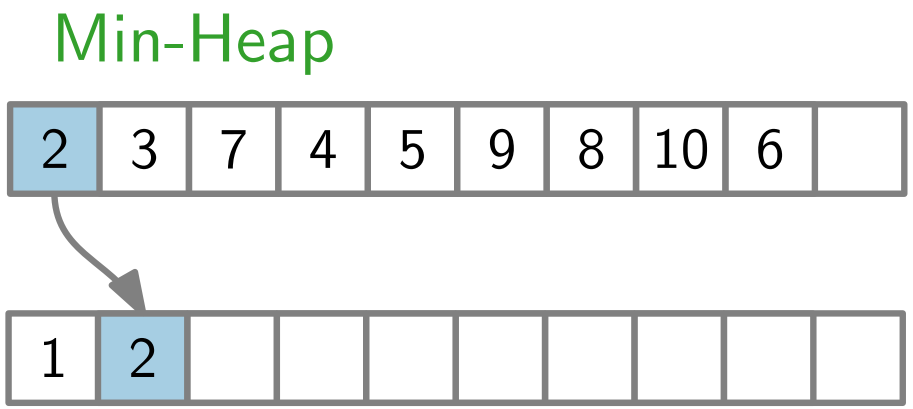


HeapSort

Idee:

- ExtractMin() gibt kleinstes Heap-Element aus.

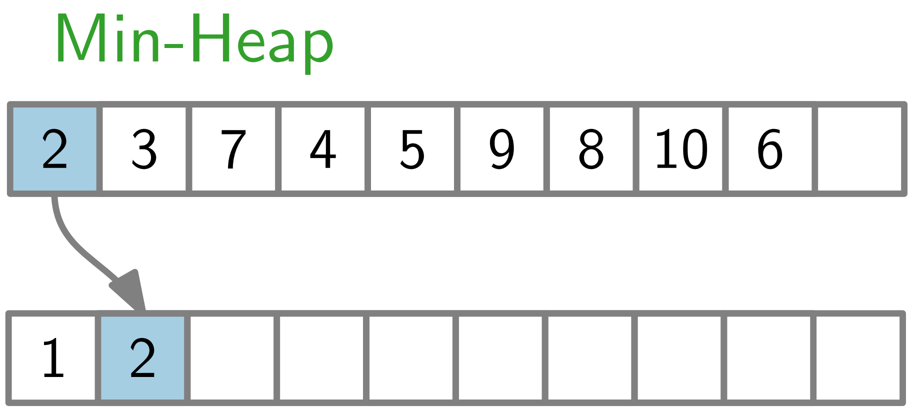
```
int[] HeapSort(int[] A)
  BuildMinHeap(A)
  B = new int[A.length]


  return B
```



HeapSort

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

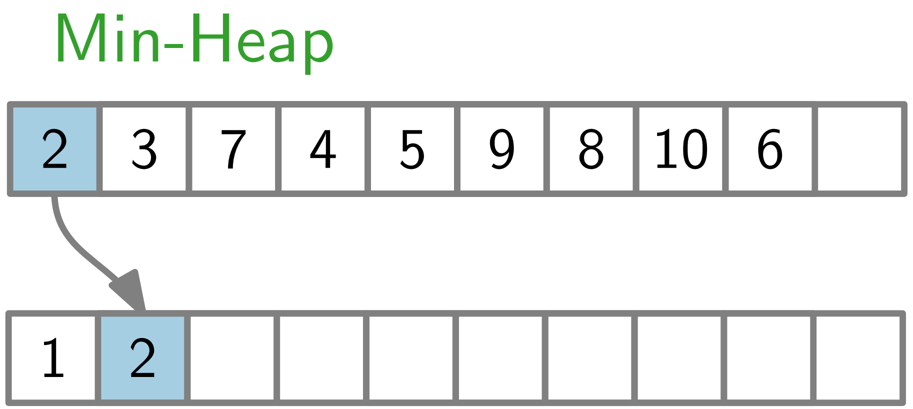
```
int[] HeapSort(int[] A)
    BuildMinHeap(A)
    B = new int[A.length]
    for i = 1 to A.length do
     └
    return B
```



HeapSort

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

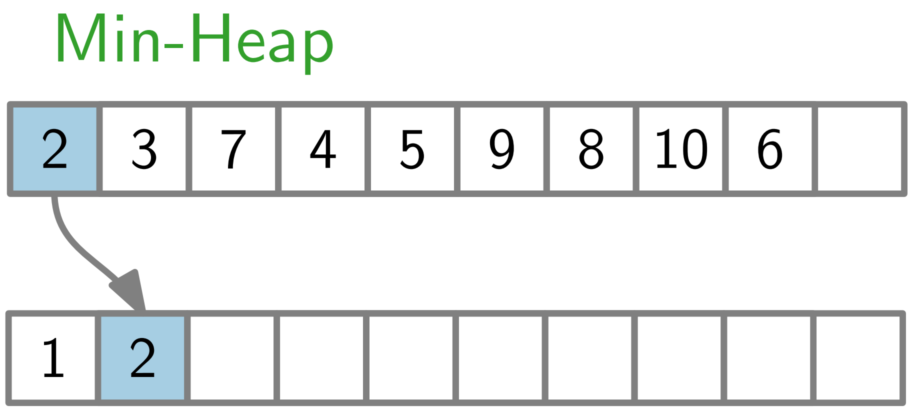
```
int[] HeapSort(int[] A)
  BuildMinHeap(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = ExtractMin()
  return B
```



HeapSort

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

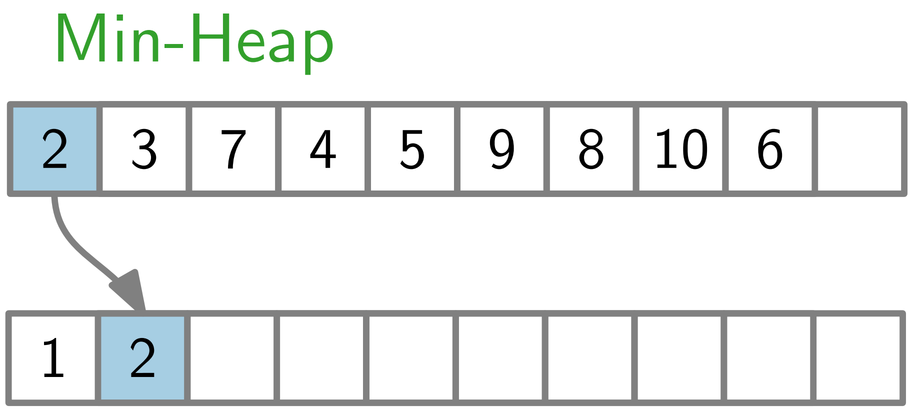
```
int[] HEAPSORT(int[] A)
  BUILDMINHEAP(A)
  B = new int[A.length]
  for i = 1 to A.length do
    B[i] = EXTRACTMIN()
  return B
```



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
```



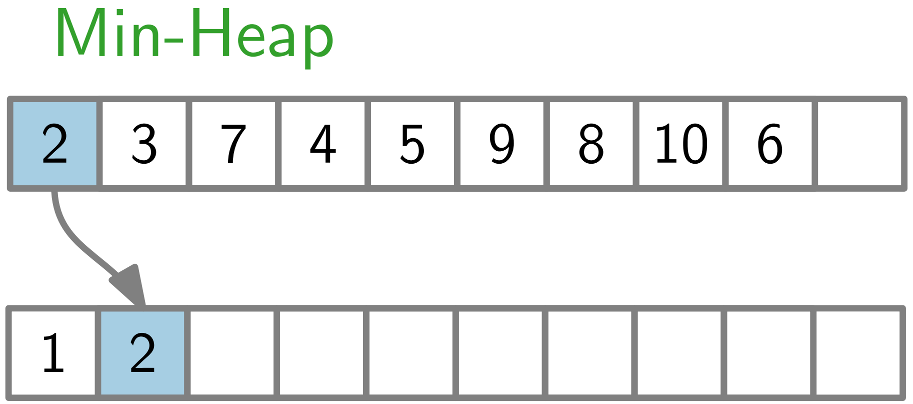
Laufzeit:

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

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
```



Laufzeit:

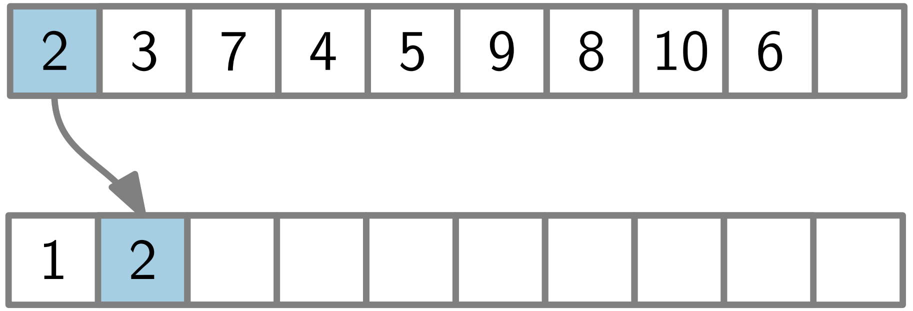
Obere Schranke: $T_{\mathrm{HS}}(n) \in$

HeapSort

Idee: ▪ $\textsc{ExtractMin}()$ gibt kleinstes Heap-Element aus.

$\text{int}[\,]\ \textsc{HeapSort}(\text{int}[\,]\ A)$
  $\textsc{BuildMinHeap}(A)$
  $B = \textbf{new int}[A.length]$
  for $i = 1$ **to** $A.length$ **do**
    $B[i] = \textsc{ExtractMin}()$
  return B

Min-Heap



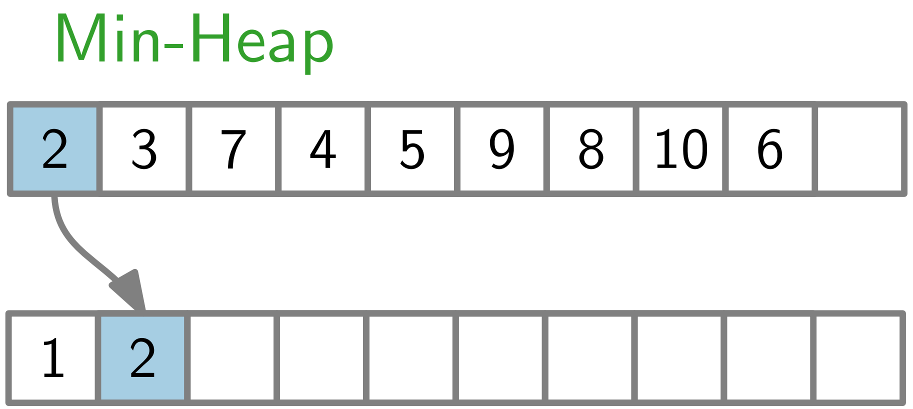
Laufzeit:

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

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
```



Laufzeit:

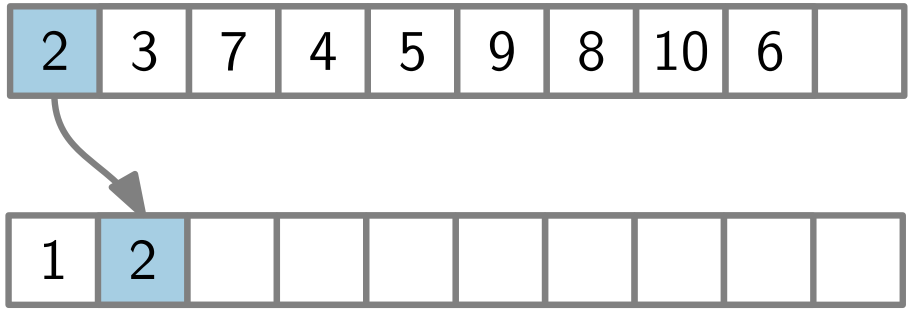
Obere Schranke: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n)$

HeapSort

Idee: ■ $\textsc{ExtractMin}()$ 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:

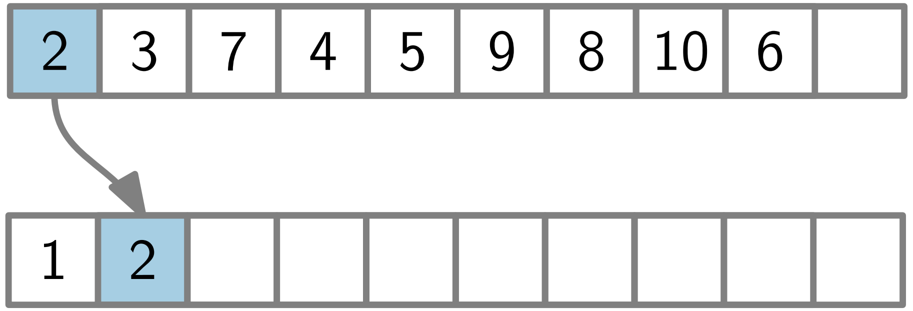
Obere Schranke: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n) +$

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:

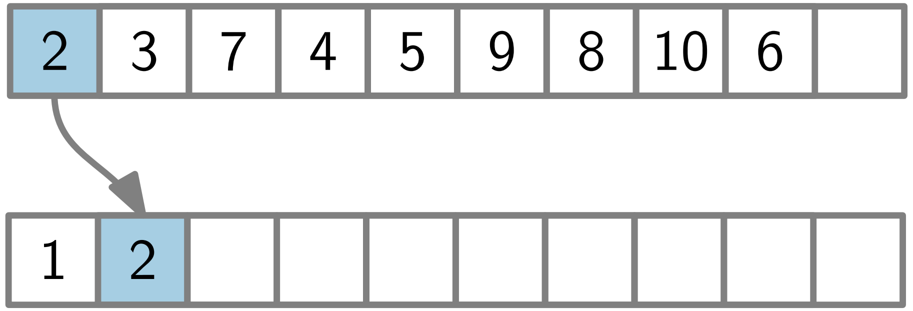
Obere Schranke: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n) +$

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



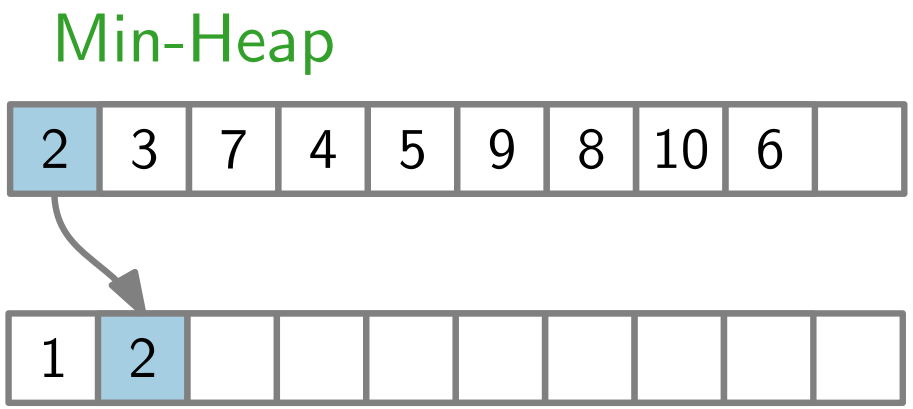
Laufzeit:

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

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
```



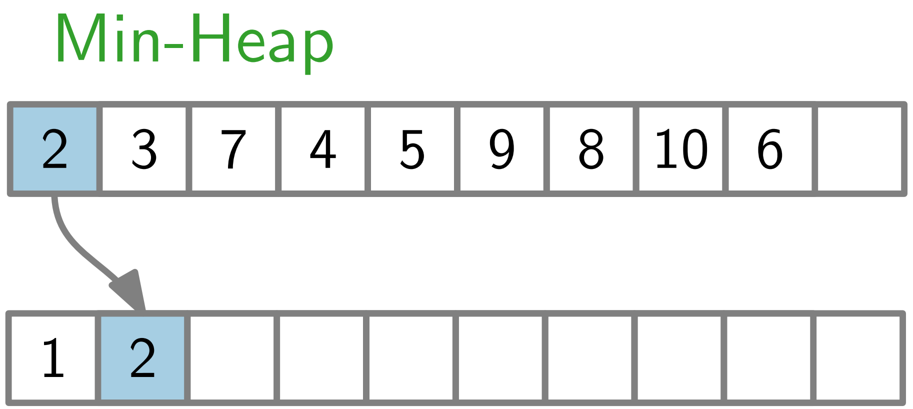
Laufzeit:

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

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
```



Laufzeit:

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

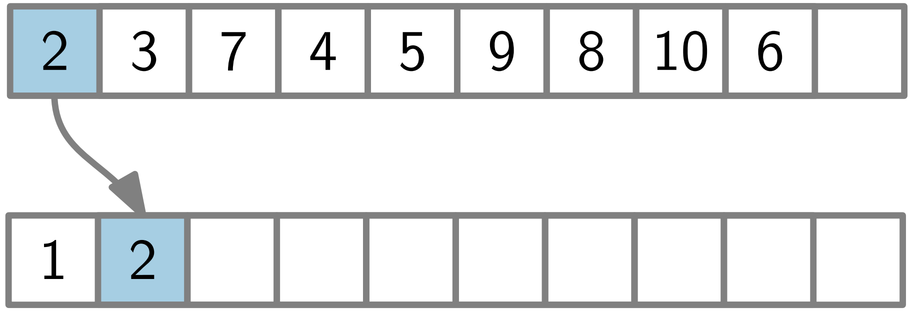
HeapSort

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:

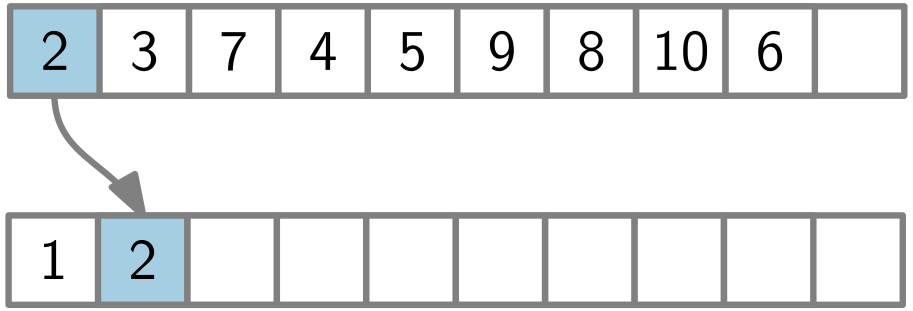
Obere Schranke: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n)$

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:

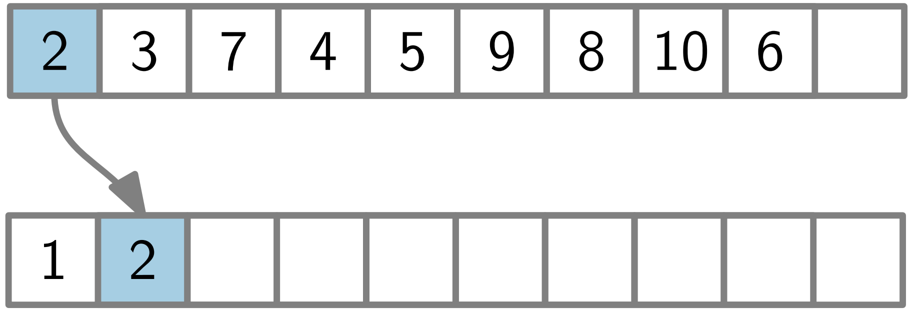
Obere Schranke: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) =$

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

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

2	3	7	4	5	9	8	10	6	

1	2								

Laufzeit:

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

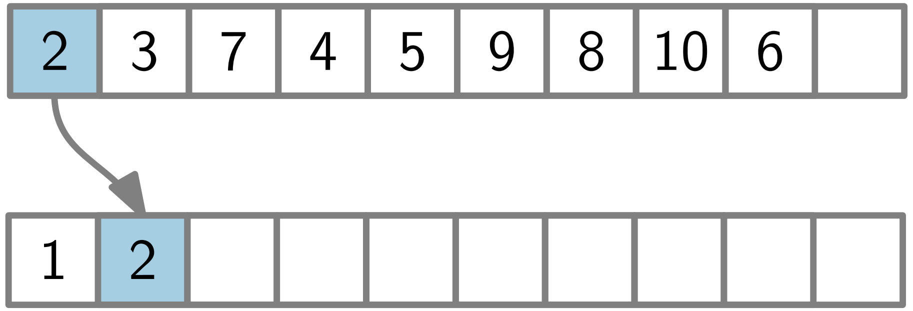
Satz. HeapSort sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

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

Untere Schranke: $c \cdot n +$

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

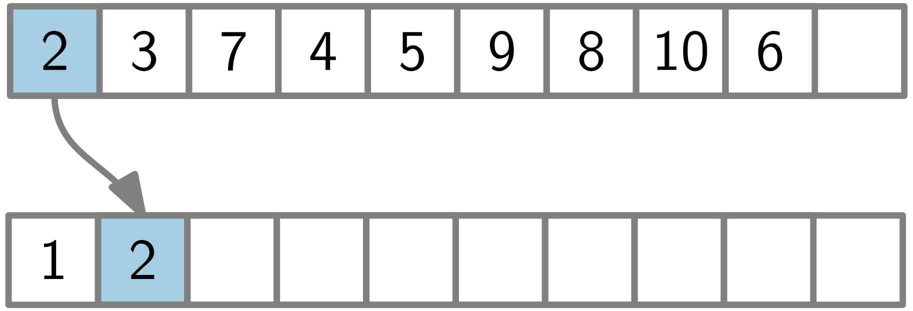
HeapSort

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

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

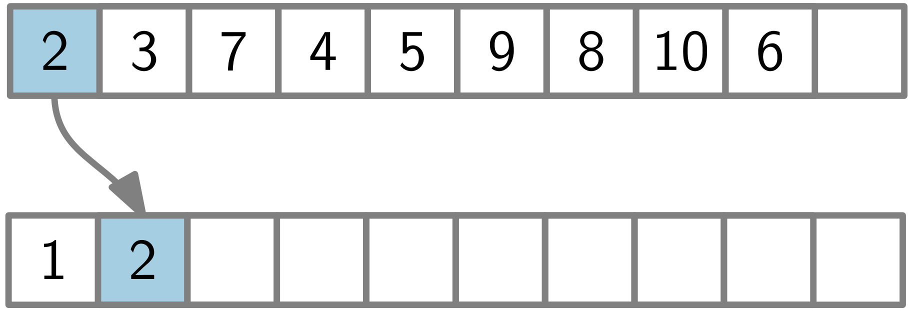
Satz. HeapSort sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

HeapSort

Idee: ■ $\textsc{ExtractMin}()$ 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: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

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

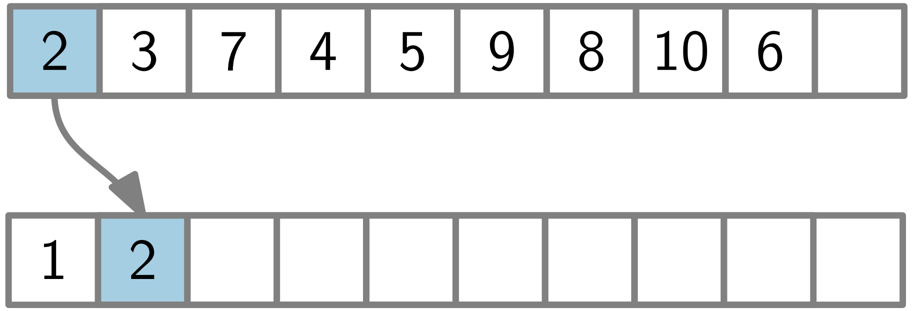
Satz. $\textsc{HeapSort}$ sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

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

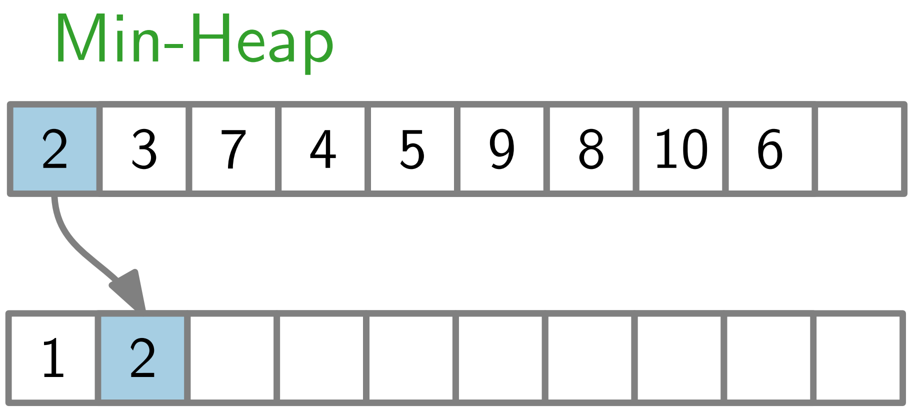
Untere Schranke: $c \cdot n + \sum_{i=1}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n}$

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
```



Laufzeit:

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

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

2	3	7	4	5	9	8	10	6	

1	2								

Laufzeit:

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

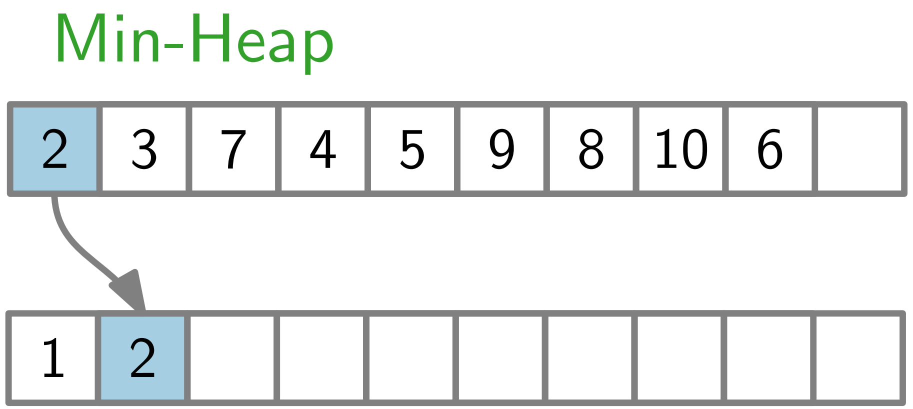
Untere Schranke: $c \cdot n + \sum_{i=1}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2} \in \Omega(\qquad)$

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



Laufzeit:

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

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

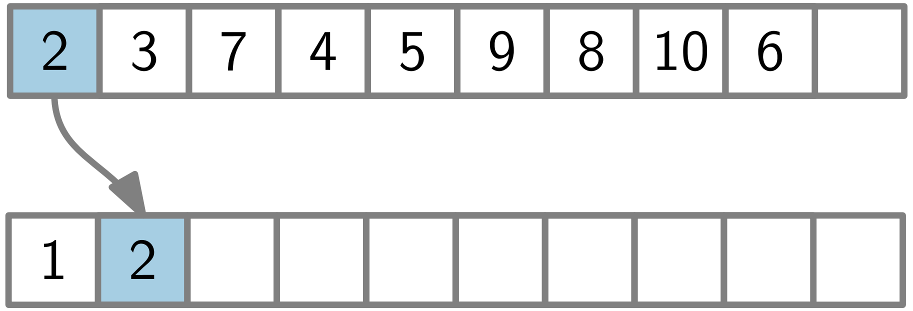
Satz. HeapSort sortiert n Schlüssel in $\mathcal{O}(n \log n)$ Zeit.

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 Sortieralgorithmus 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 Sortieralgorithmus 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 Sortieralgorithmus 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 Sortieralgorithmus 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 Sortieralgorithmus 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 Sortieralgorithmus 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 Sortieralgorithmus 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)

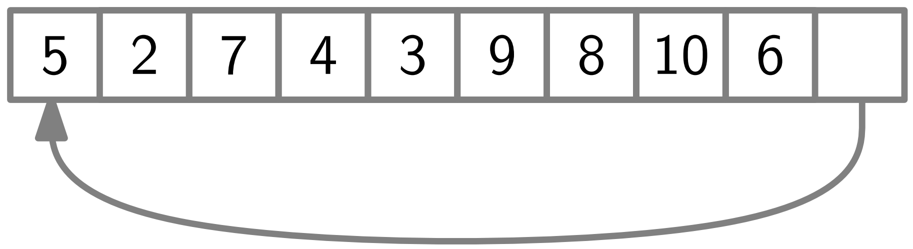
Idee:

- EXTRACTMIN() gibt kleinstes Heap-Element aus.

1	2	7	4	3	9	8	10	6	5

Vom Heap zur Sortierung (in-situ)

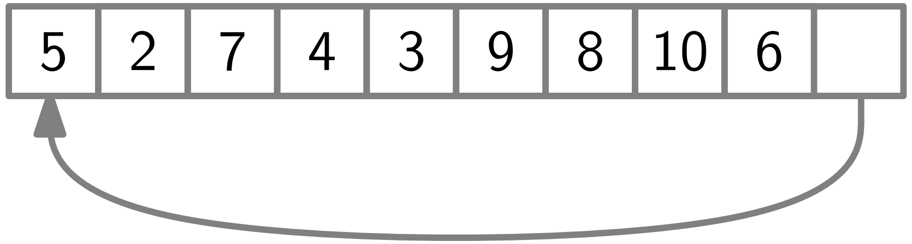
Idee:
- ExtractMin() gibt kleinstes Heap-Element aus.



Vom Heap zur Sortierung (in-situ)

Idee:

- ExtractMin() gibt kleinstes Heap-Element aus.
- Letztes Feldelement wird dadurch frei



Vom Heap zur Sortierung (in-situ)

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

5	2	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

5	2	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

2	5	7	4	3	9	8	10	6	1

Vom Heap zur Sortierung (in-situ)

Idee:

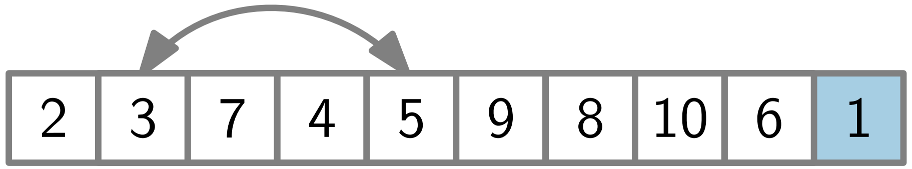
- ExtractMin() gibt kleinstes Heap-Element aus.
- Letztes Feldelement wird dadurch frei

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

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:
- $\textsc{ExtractMin}()$ gibt kleinstes Heap-Element aus.
- Letztes Feldelement wird dadurch frei

$\textsc{HeapSort}(\text{int}[]\ A)$
  $\textsc{BuildMinHeap}(A)$
  for $i = A.length$ **downto** 2 **do**
    $A[i] = \textsc{ExtractMin}()$

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] = 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:
- $\textsc{ExtractMin}()$ gibt kleinstes Heap-Element aus.
- Letztes Feldelement wird dadurch frei

$\textsc{HeapSort}(\text{int}[]\ A)$
  $\textsc{BuildMinHeap}(A)$
  for $i = A.length$ **downto** 2 **do**
    $A[i] = \textsc{ExtractMin}()$
  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] = $ 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

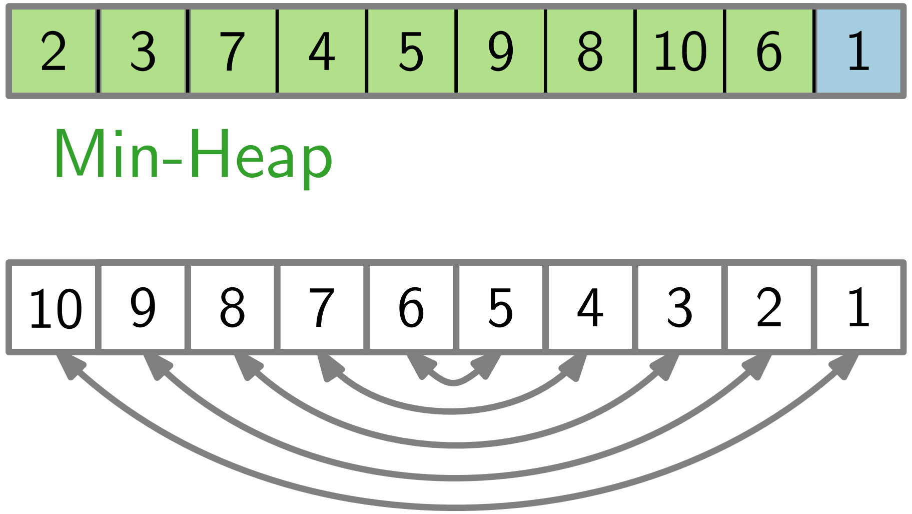
Min-Heap

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**
    $A[i] = \text{ExtractMin}()$
  for $i = 1$ **to** $\lfloor A.length/2 \rfloor$ **do**
    $A[i] \leftrightarrow A[n-i]$



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]
```

2	3	7	4	5	9	8	10	6	1

Min-Heap

10	9	8	7	6	5	4	3	2	1

1	2	3	4	5	6	7	8	9	10

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** $\lfloor A.length/2 \rfloor$ **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

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

1	2	3	4	5	6	7	8	9	10

Vergleich Laufzeiten

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

Ein Sortieralgorithmus 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 Sortieralgorithmus 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	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗