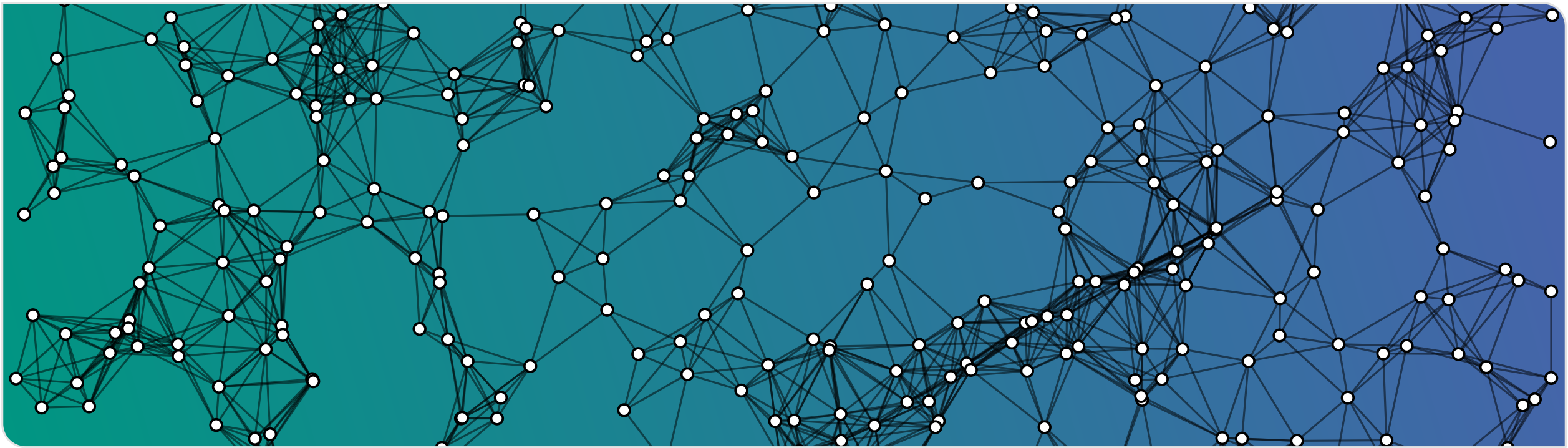
Karlsruher Institut für Technologie

Zusatztutorium III
Sortieren, Heaps, Suchbäume

Algorithmen I, Zusatztutorium

Jonas Seiler, Carina Weber | 15. August 2022



Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste

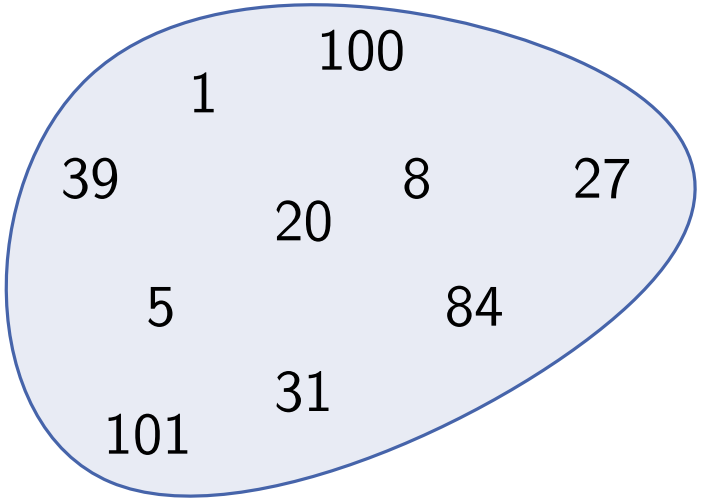
Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste



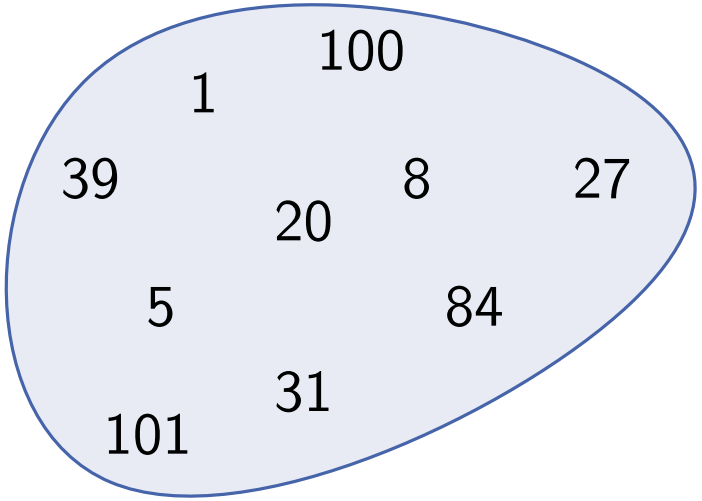
Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

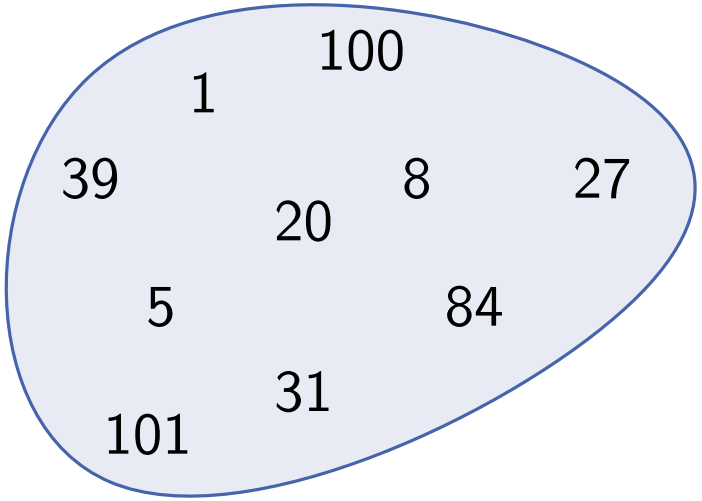
Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Stabilität

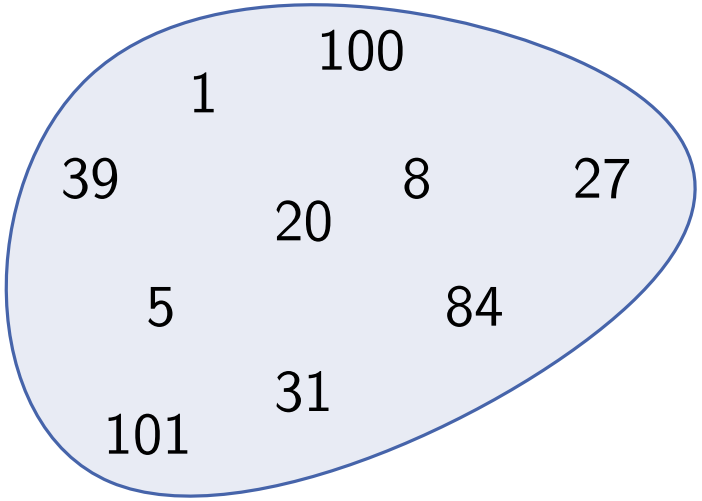
Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Stabilität

Bewahrt Reihenfolge der Elemente mit gleichem Sortierschlüssel

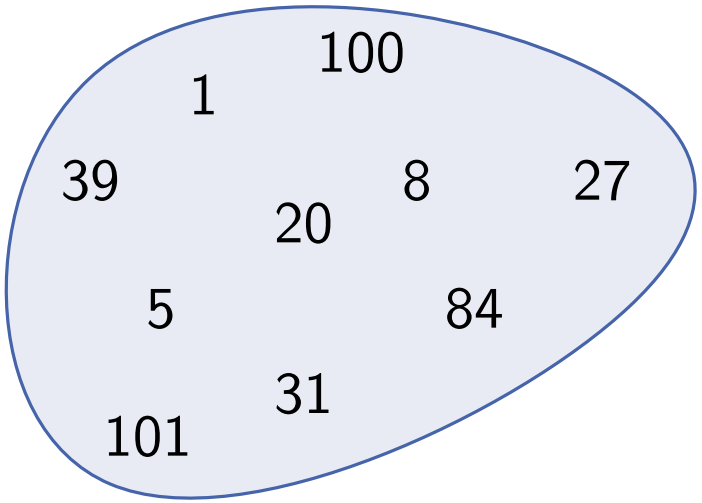
Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Stabilität

Bewahrt Reihenfolge der Elemente mit gleichem Sortierschlüssel

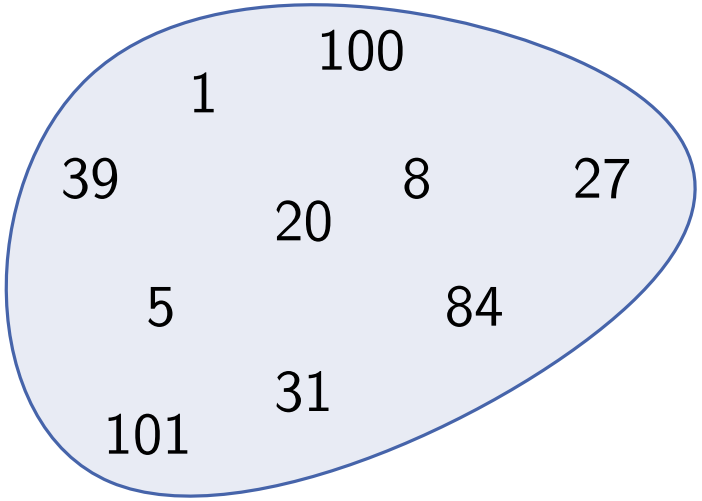
Sortieren

Gegeben

- n Elemente aus geordneter Menge

Gesucht

- sortierte Folge der Elemente
  - Array
  - Liste



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Stabilität

Bewahrt Reihenfolge der Elemente mit gleichem Sortierschlüssel

Insertionsort

- Füge Elemente nacheinander ein

Insertionsort

- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert

Insertionsort

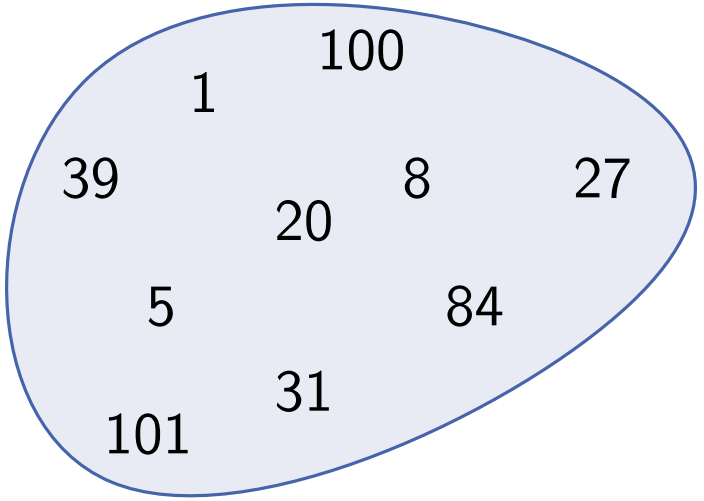
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:

Insertionsort

- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein

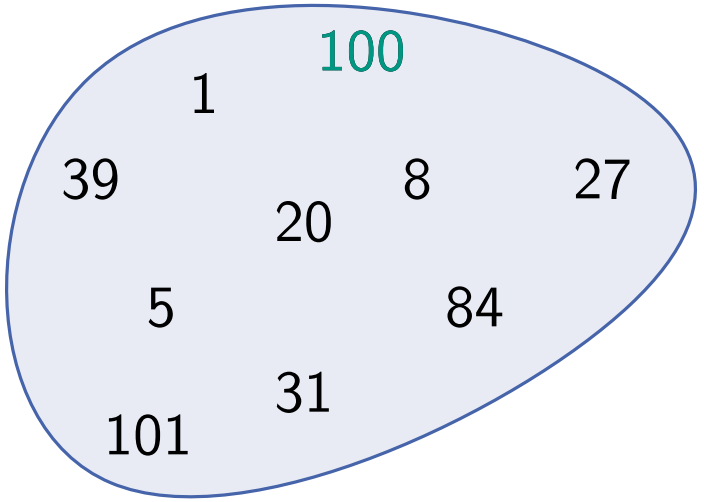
Insertionsort

- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



Insertionsort

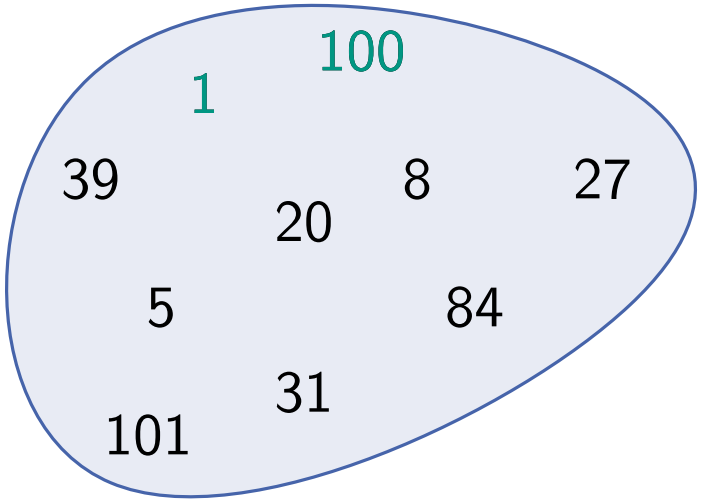
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< \quad 100 \quad >$

Insertionsort

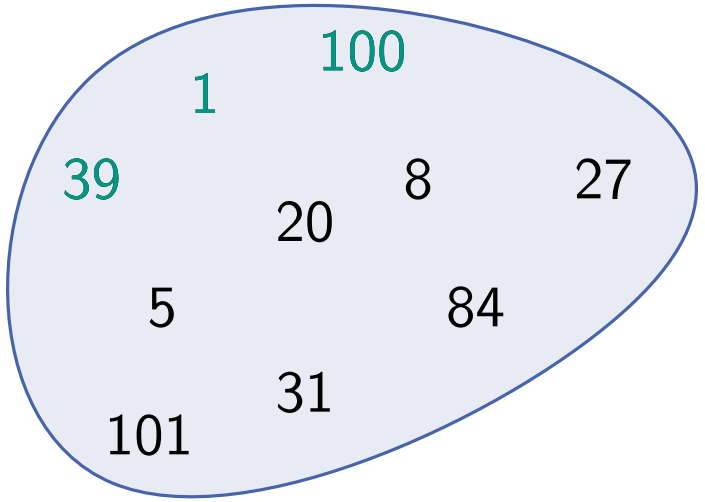
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 100 >$

Insertionsort

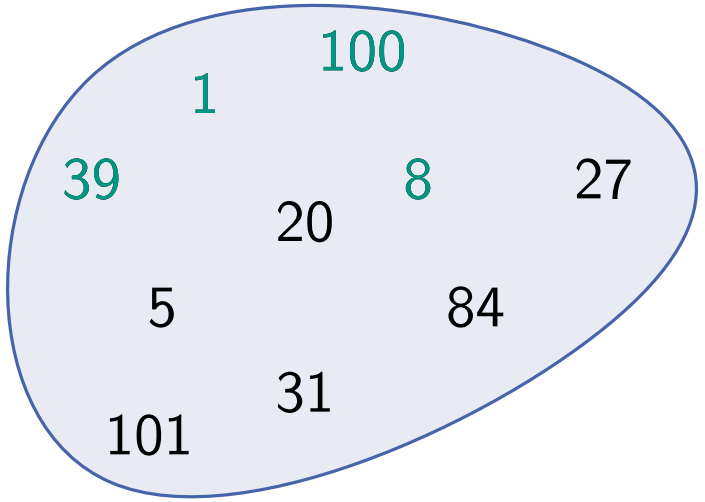
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 39 , 100 >$

Insertionsort

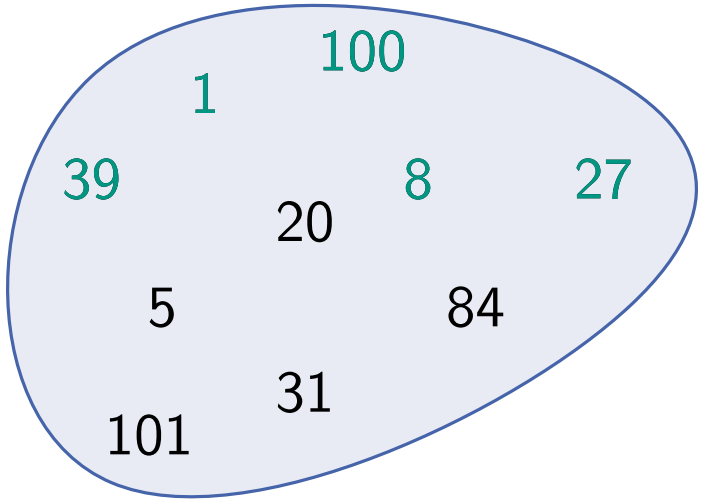
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 8, 39, 100 >$

Insertionsort

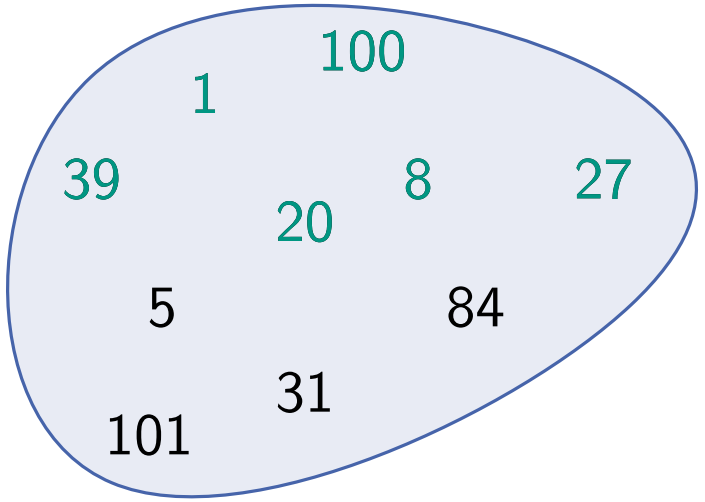
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 8, 27\ , 39, 100 >$

Insertionsort

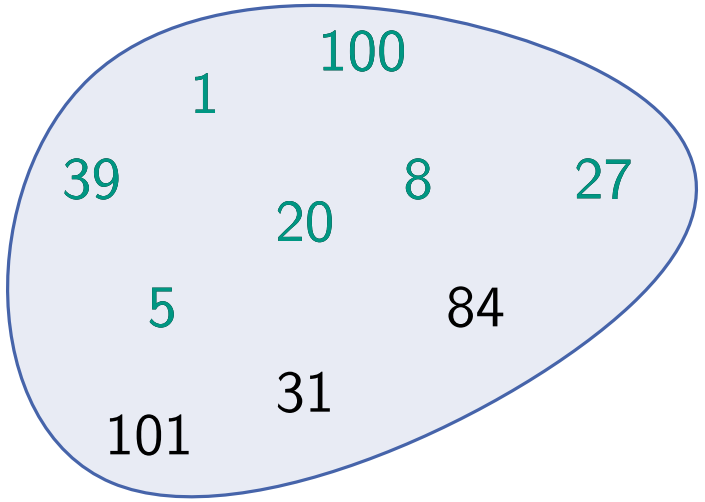
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 8, 20 \ , 27, 39, 100 >$

Insertionsort

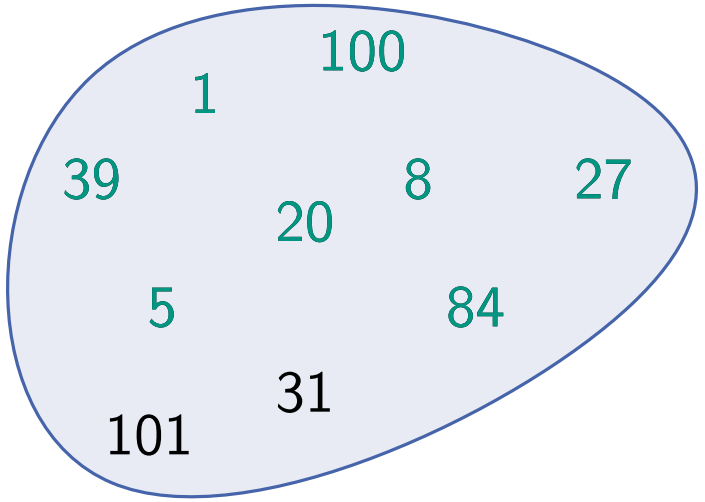
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 5, 8, 20, 27, 39, 100 >$

Insertionsort

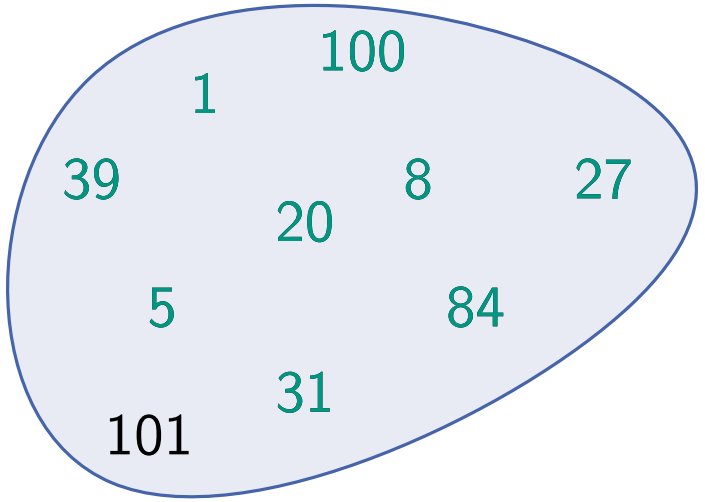
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 5, 8, 20, 27, 39, \ 84 \ , 100 >$

Insertionsort

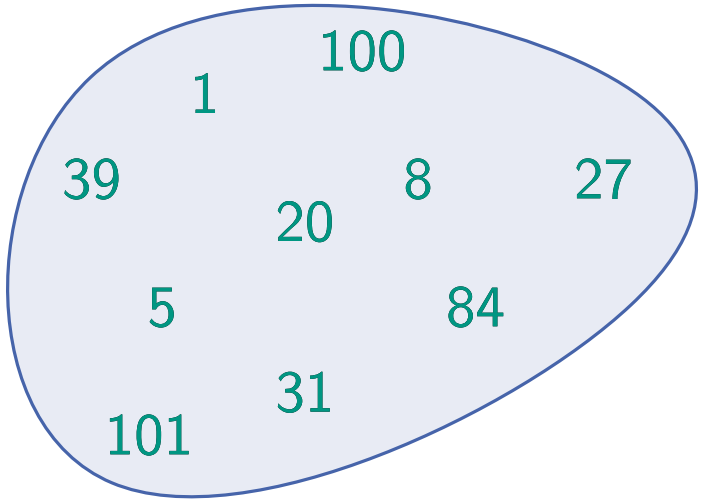
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 5, 8, 20, 27, \ 31 \ , 39, 84, 100 >$

Insertionsort

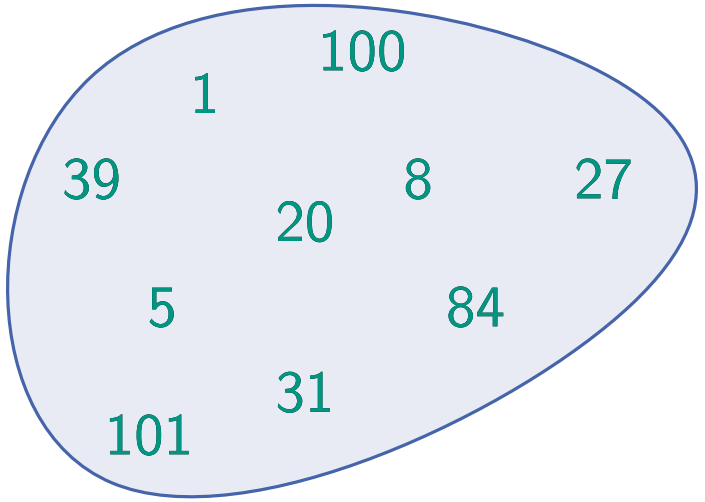
- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Insertionsort

- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



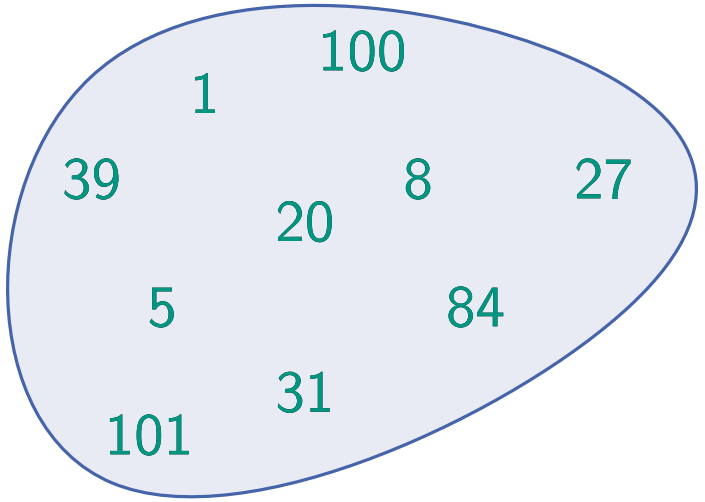
Laufzeit

- best case: $\mathcal{O}(n)$
- worst case: $\mathcal{O}(n^2)$

$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Insertionsort

- Füge Elemente nacheinander ein
- Halte bereits betrachtete Elemente sortiert
- Für i-tes Element:
  - Suche Position des ersten Elements größer (kleiner) i
  - Füge i vor (nach) dem Element ein



$< 1, 5, 8, 20, 27, 31, 39, 84, 100, 101 >$

Laufzeit

- best case: $\mathcal{O}(n)$
- worst case: $\mathcal{O}(n^2)$

stabil

Mergesort

- Zerteile rekursiv die Menge an Elementen

Mergesort

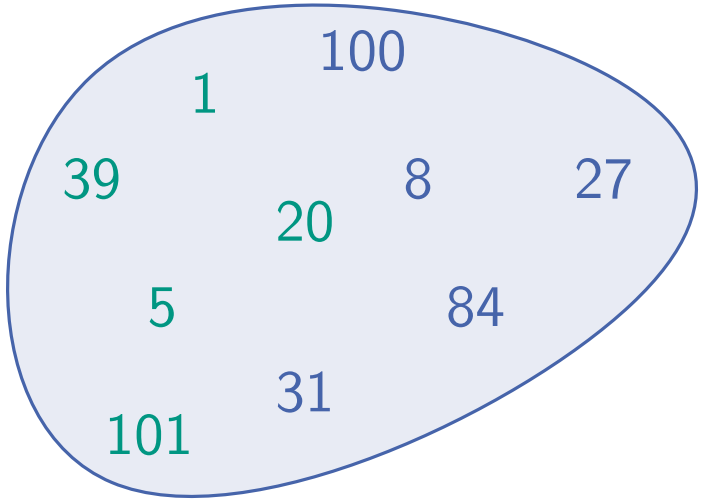
- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält

Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen

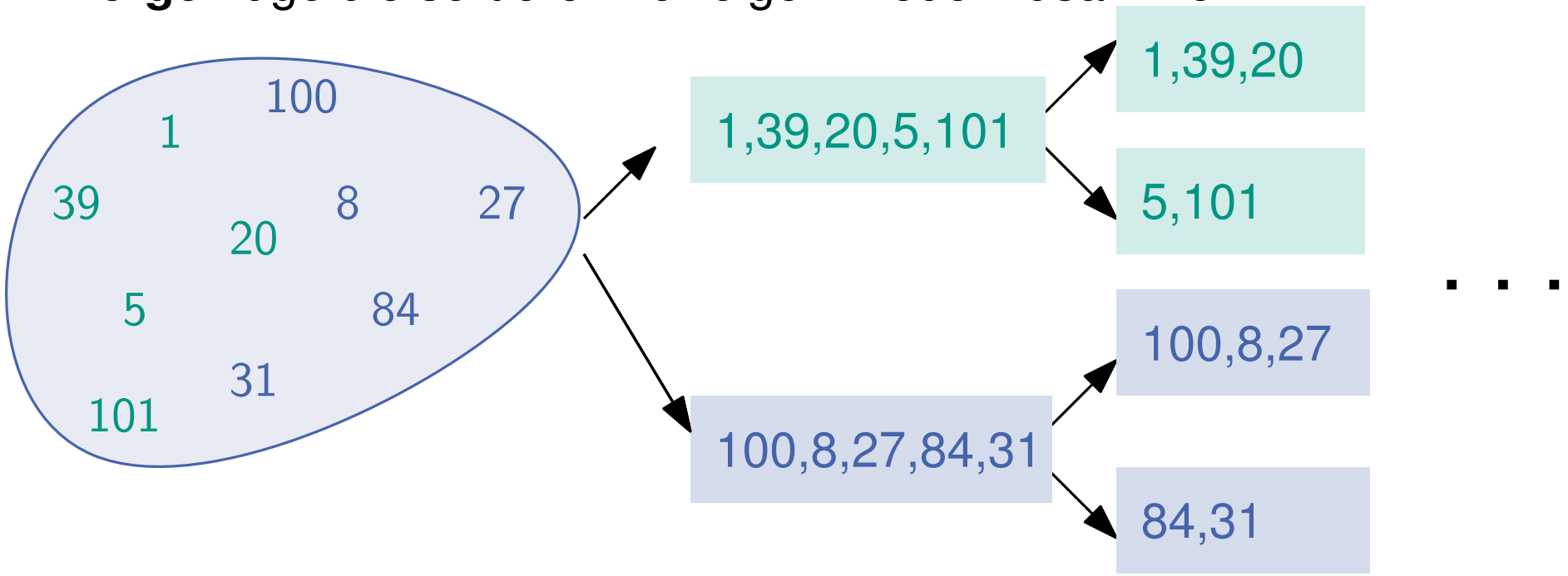
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen

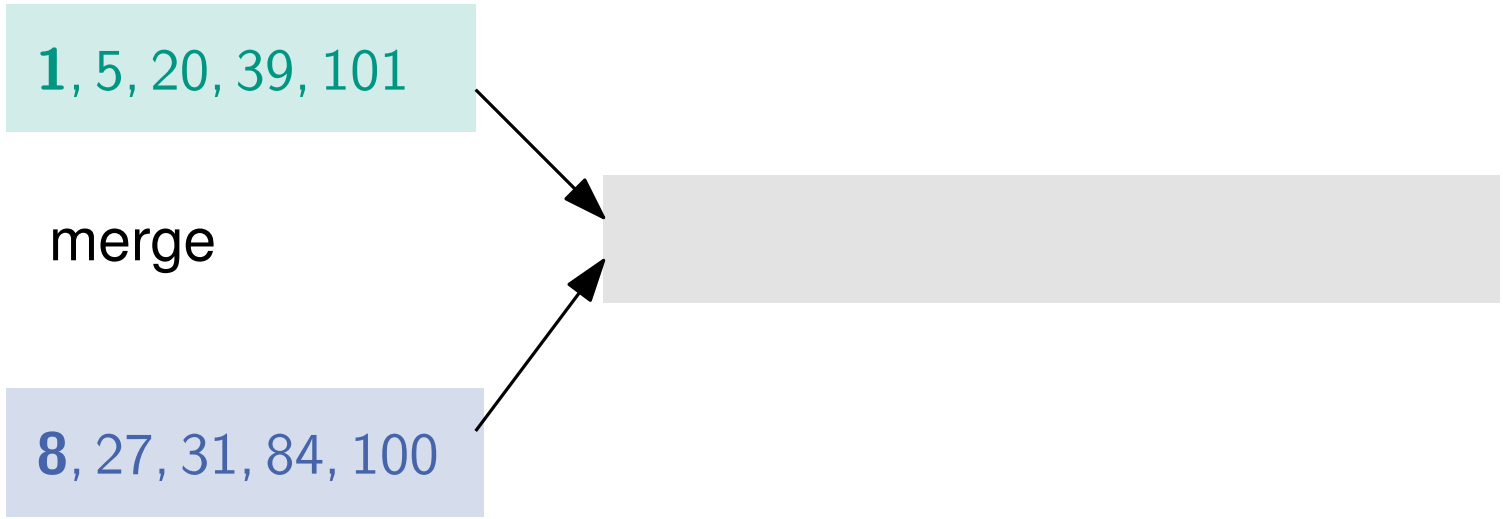
$1, 5, 20, 39, 101$

merge

$8, 27, 31, 84, 100$

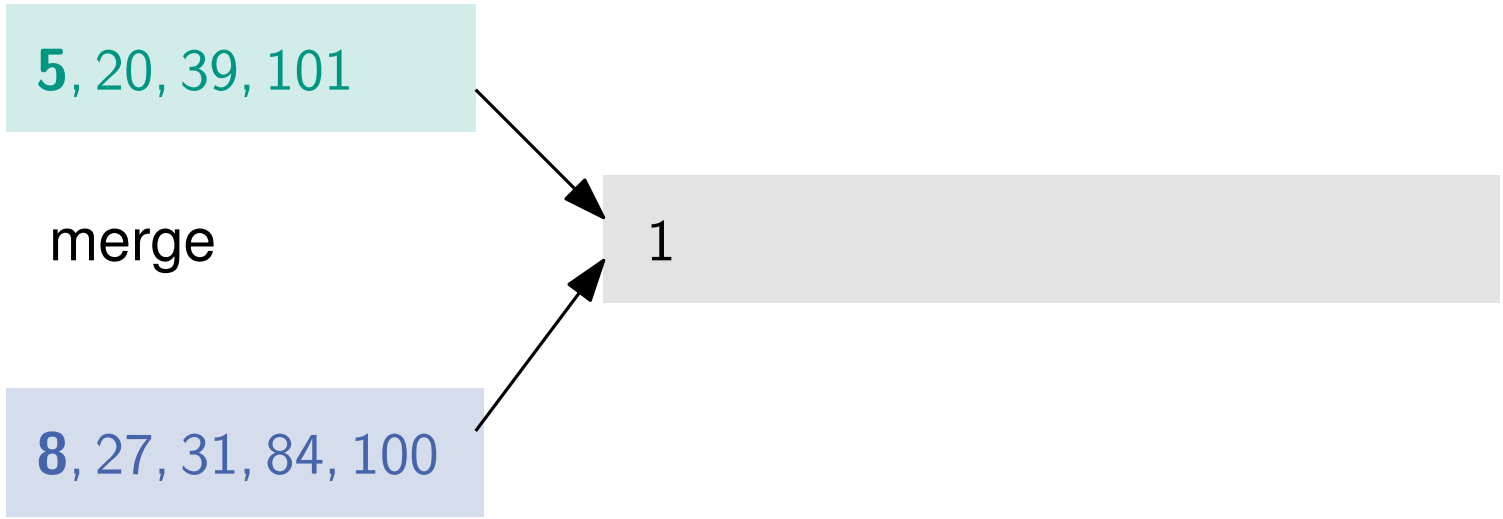
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



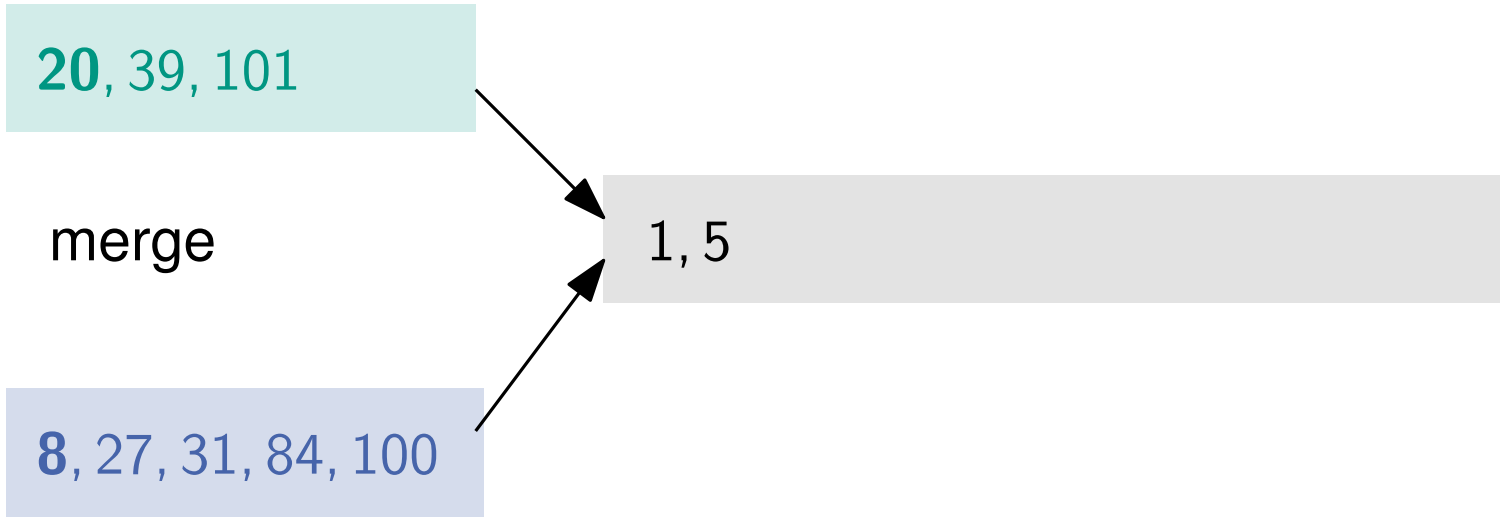
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



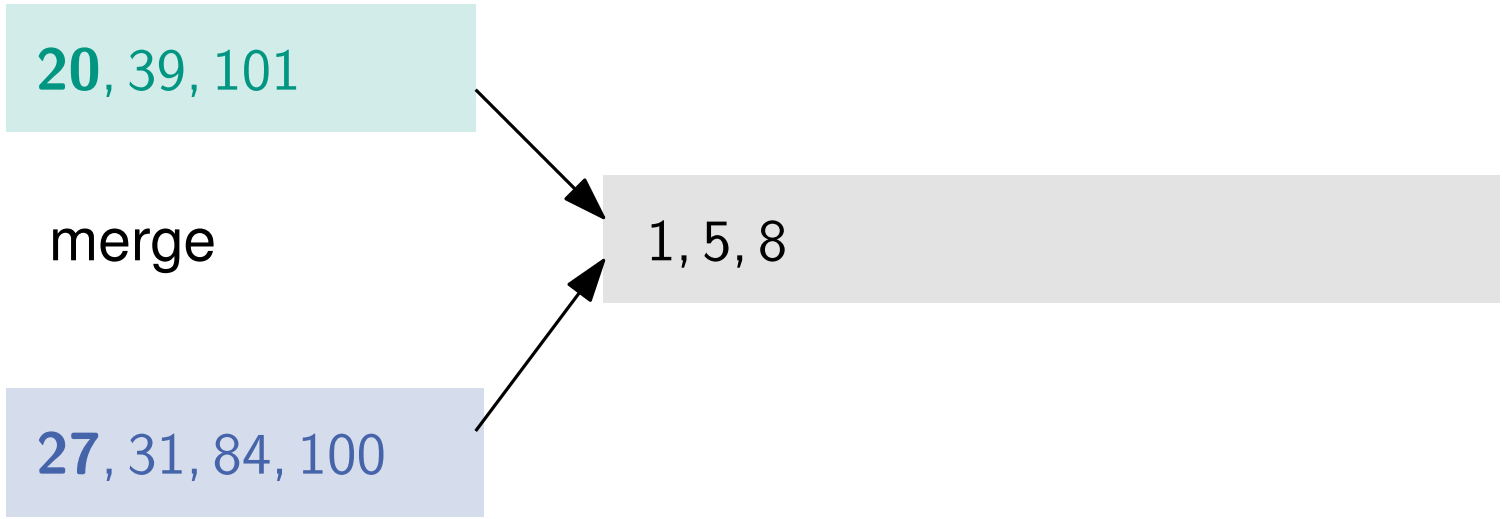
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



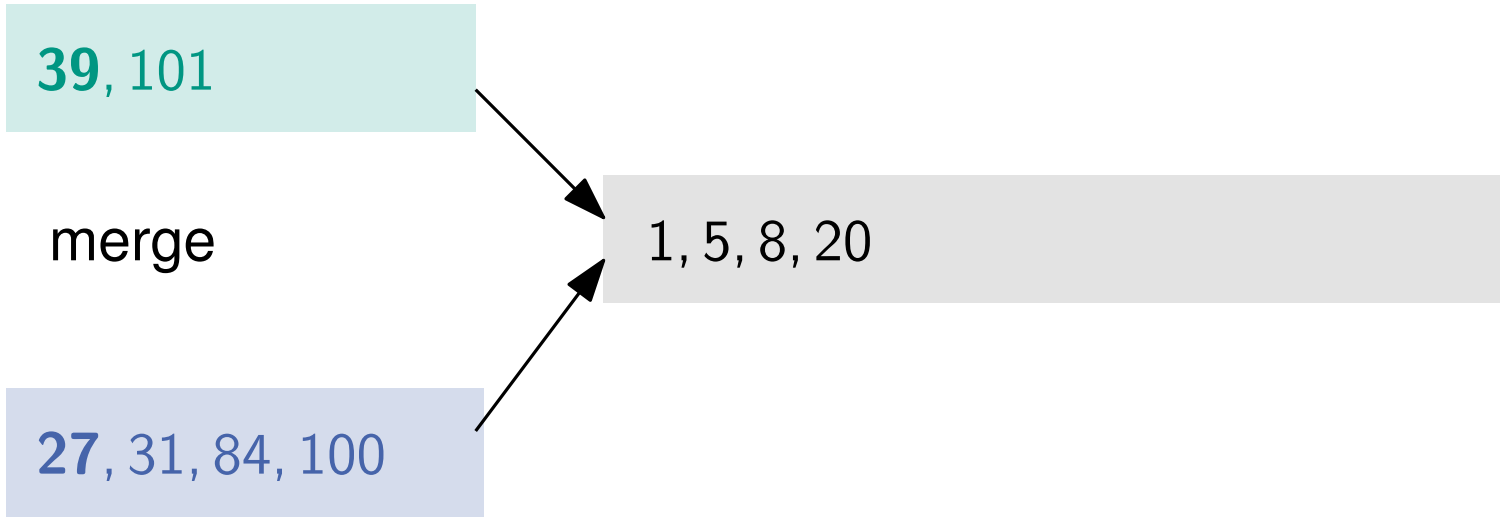
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



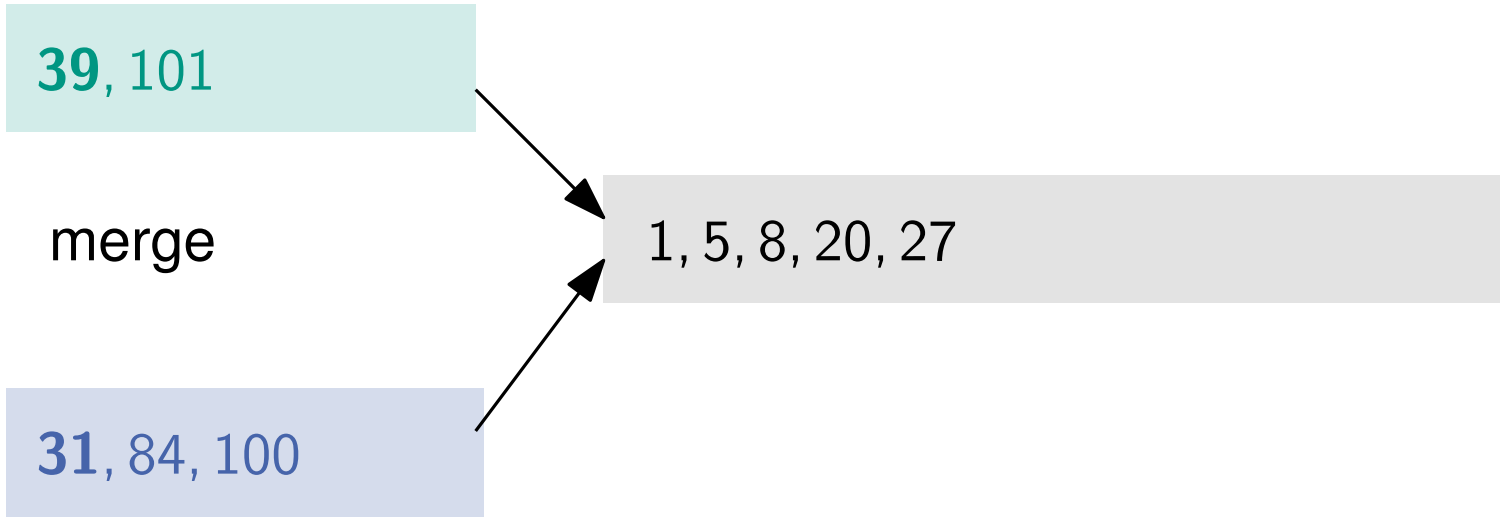
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



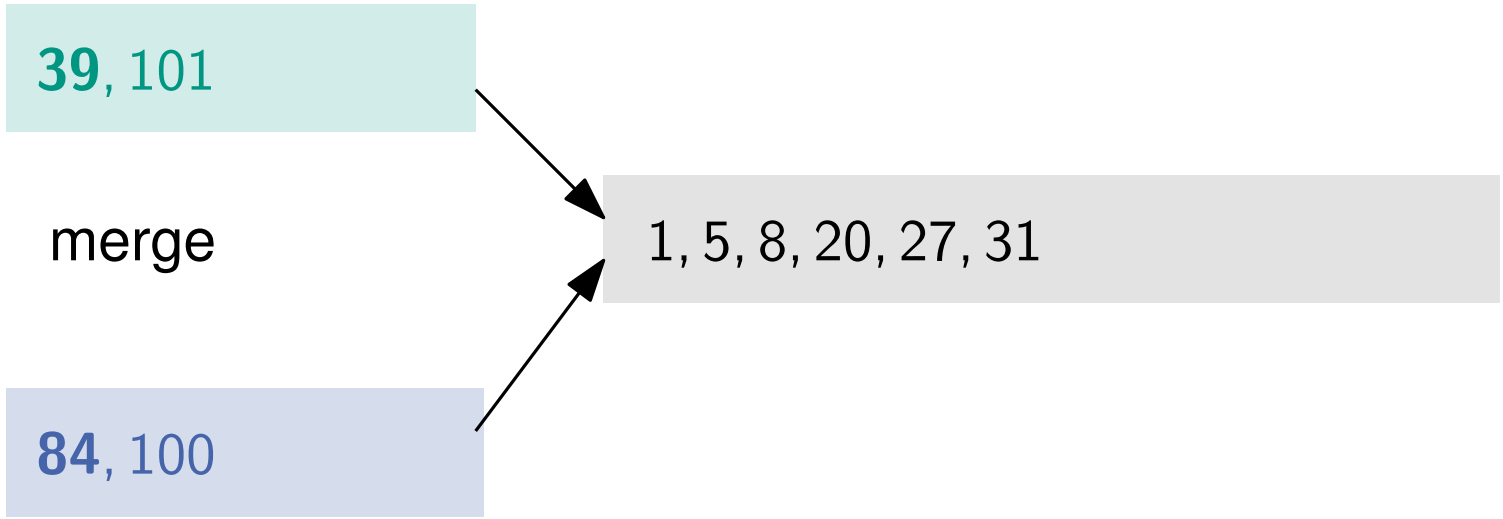
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



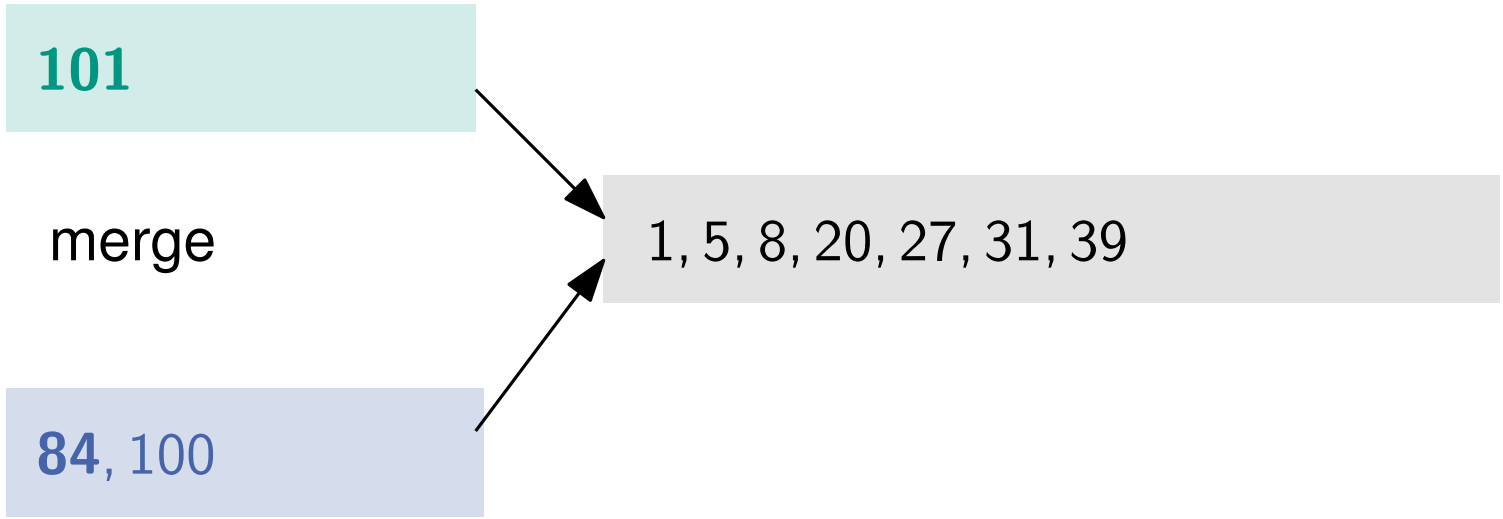
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



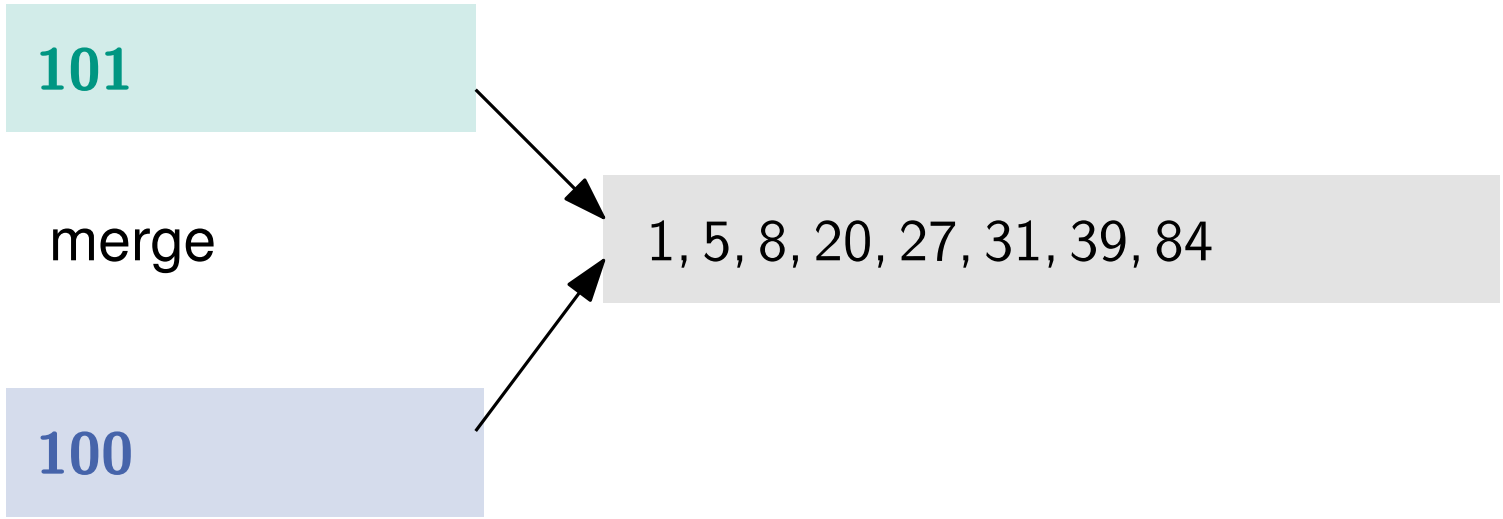
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



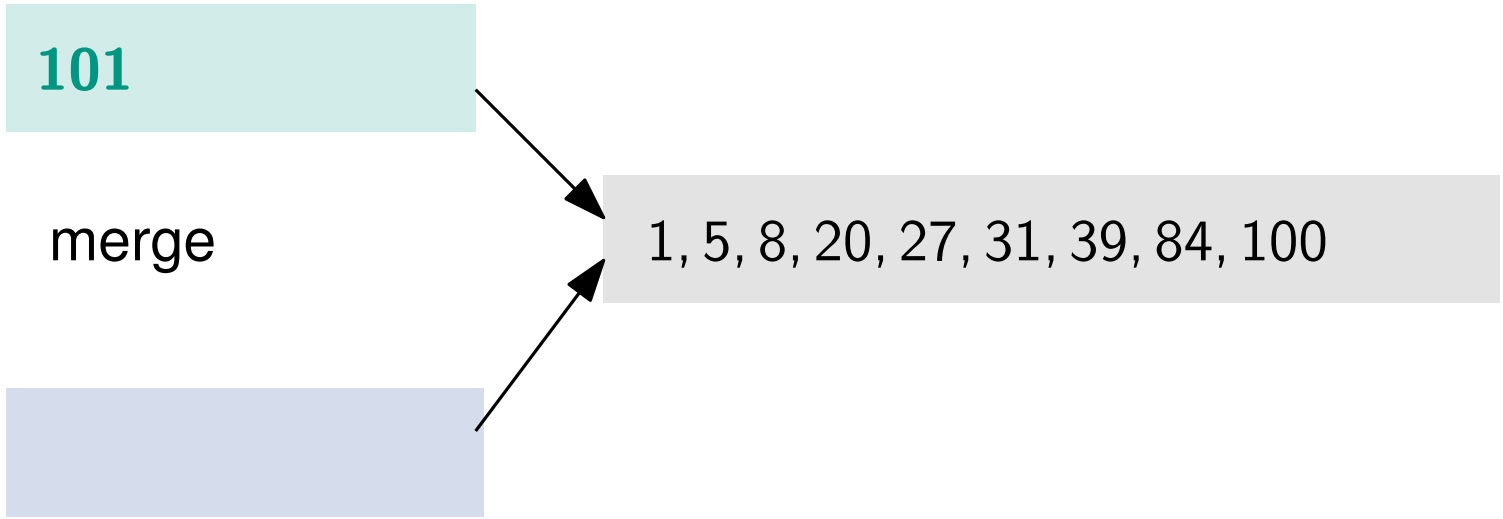
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



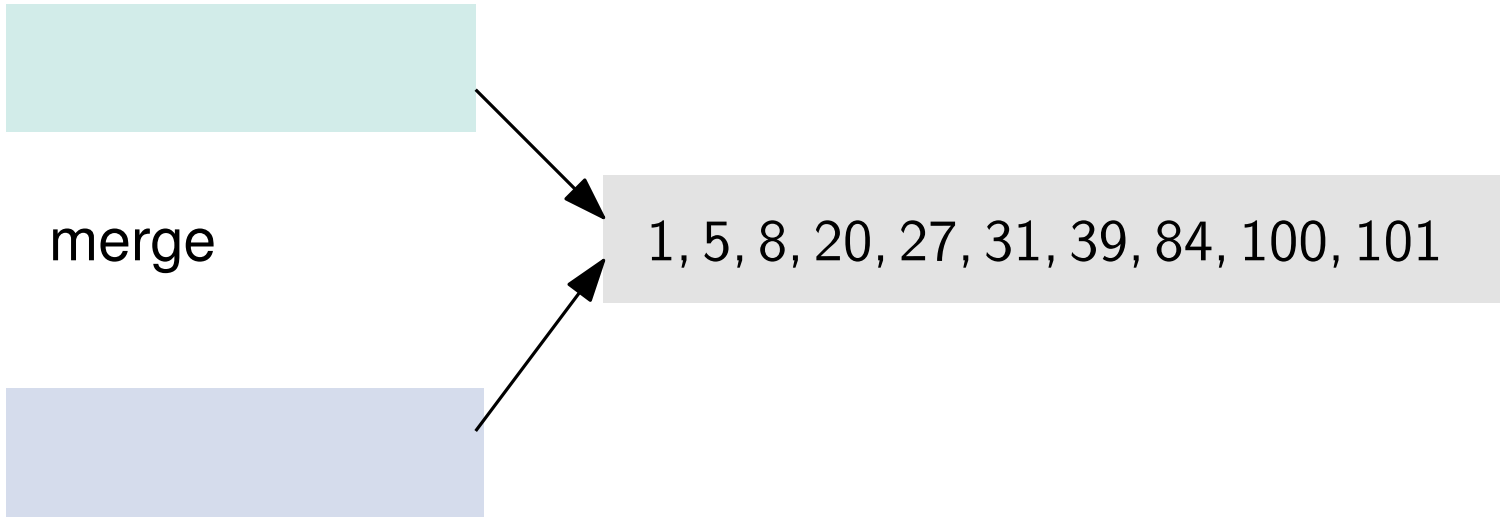
Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen



Mergesort

- Zerteile rekursiv die Menge an Elementen
  - solange, bis jede Menge nur ein Element enthält
  - **merge**: füge die sortieren Teilfolgen wieder zusammen

Laufzeit

- best case: $\mathcal{O}(n \log n)$
- worst case: $\mathcal{O}(n \log n)$

stabil

Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot

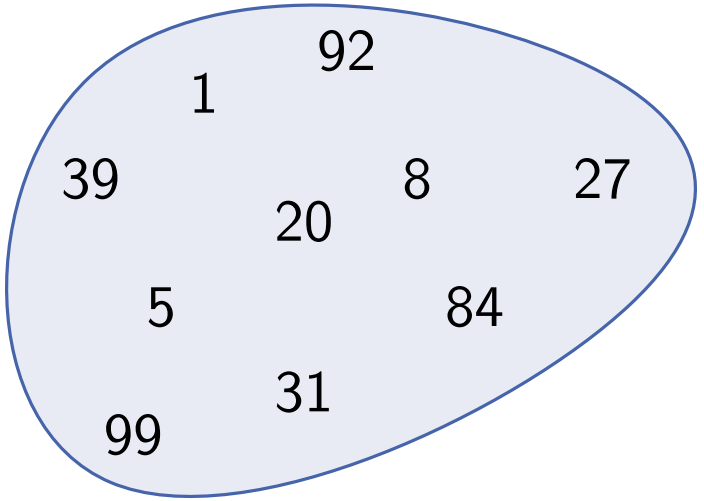
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



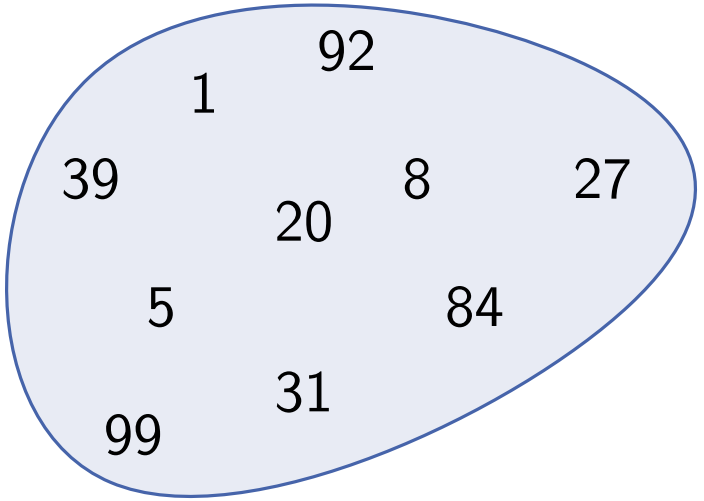
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



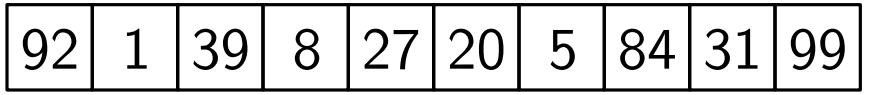
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



Quicksort

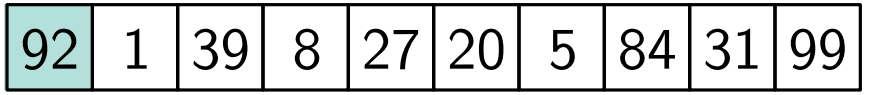
Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot

$k = 0$



Quicksort

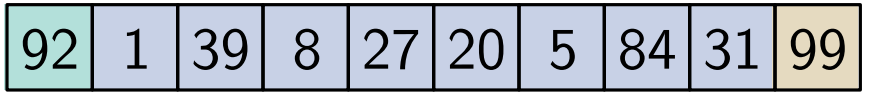
Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot

$k = 0$



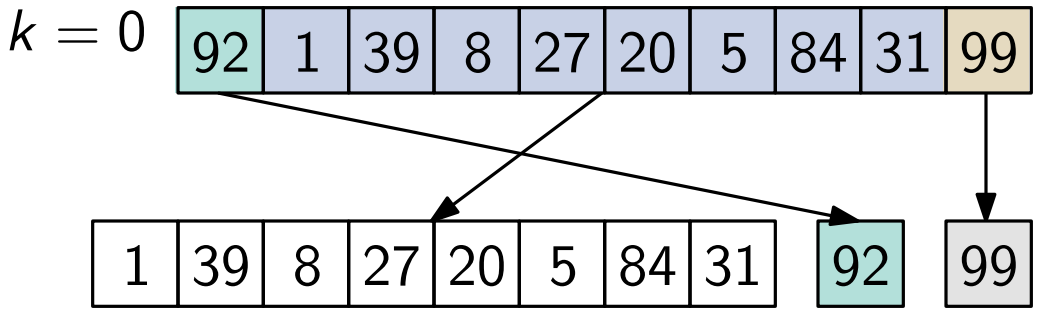
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



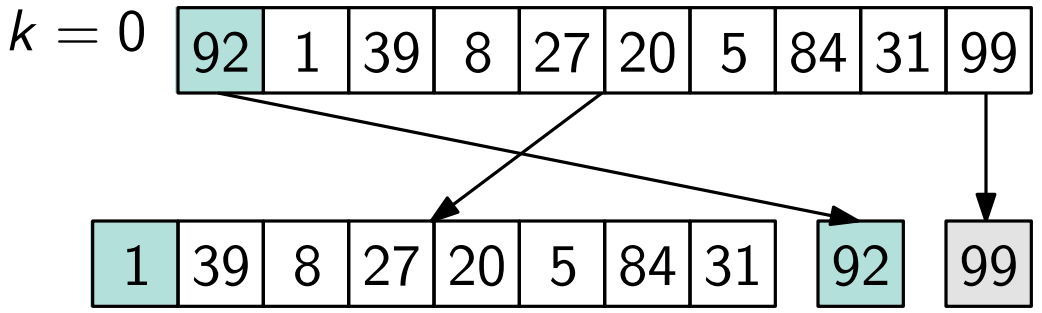
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



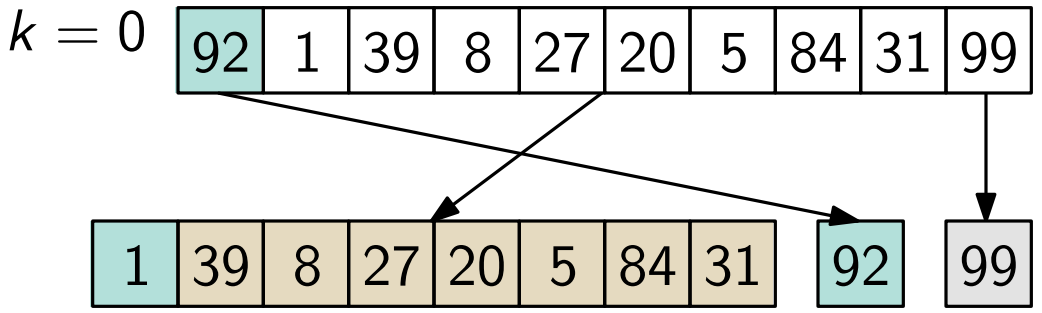
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



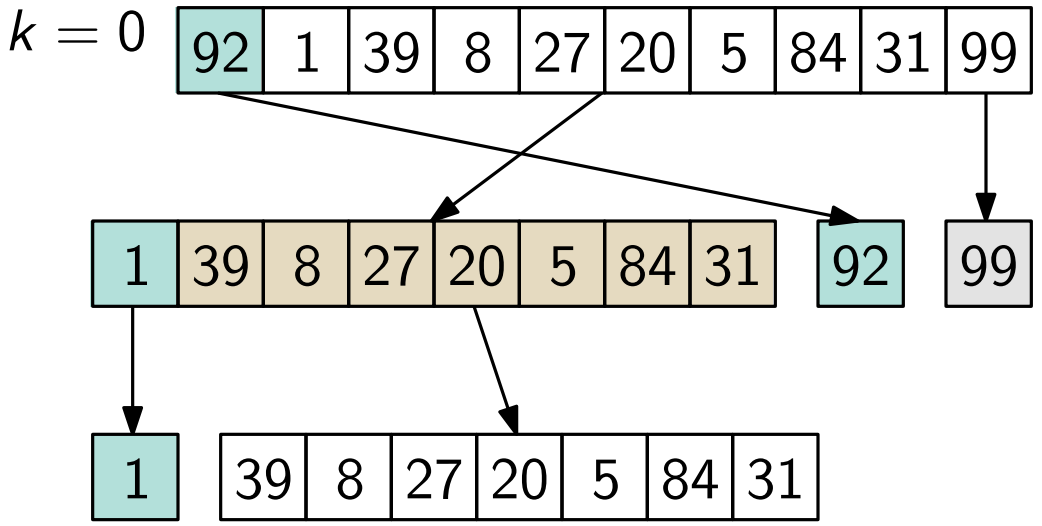
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



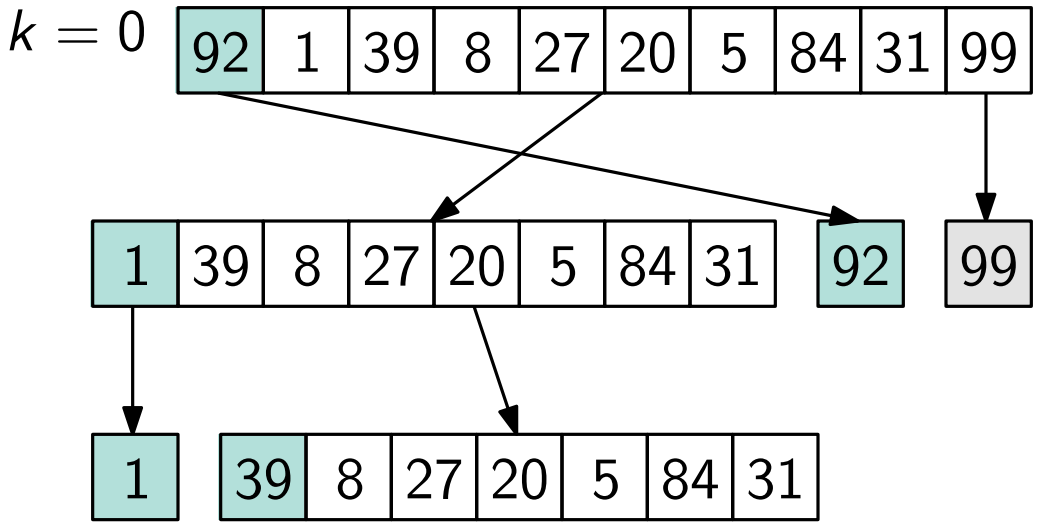
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



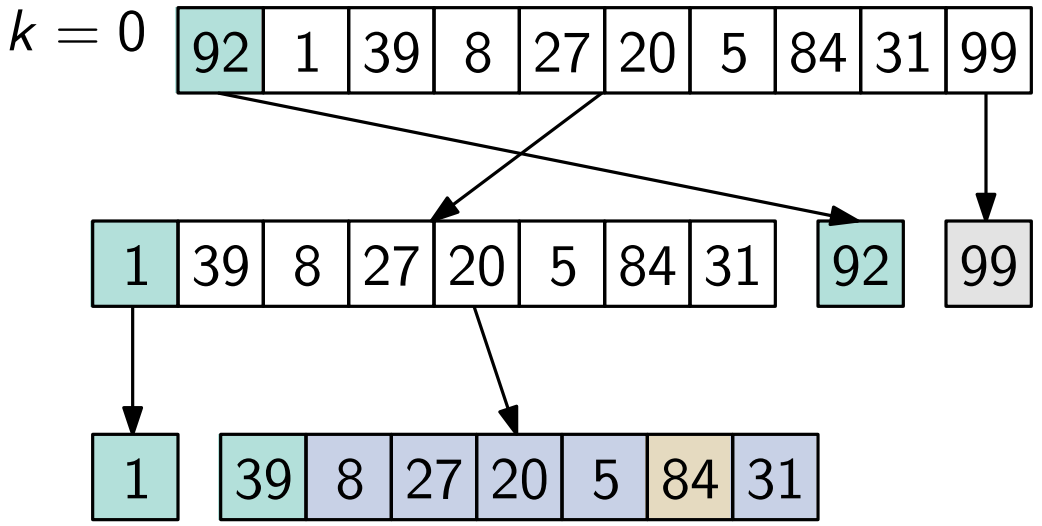
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



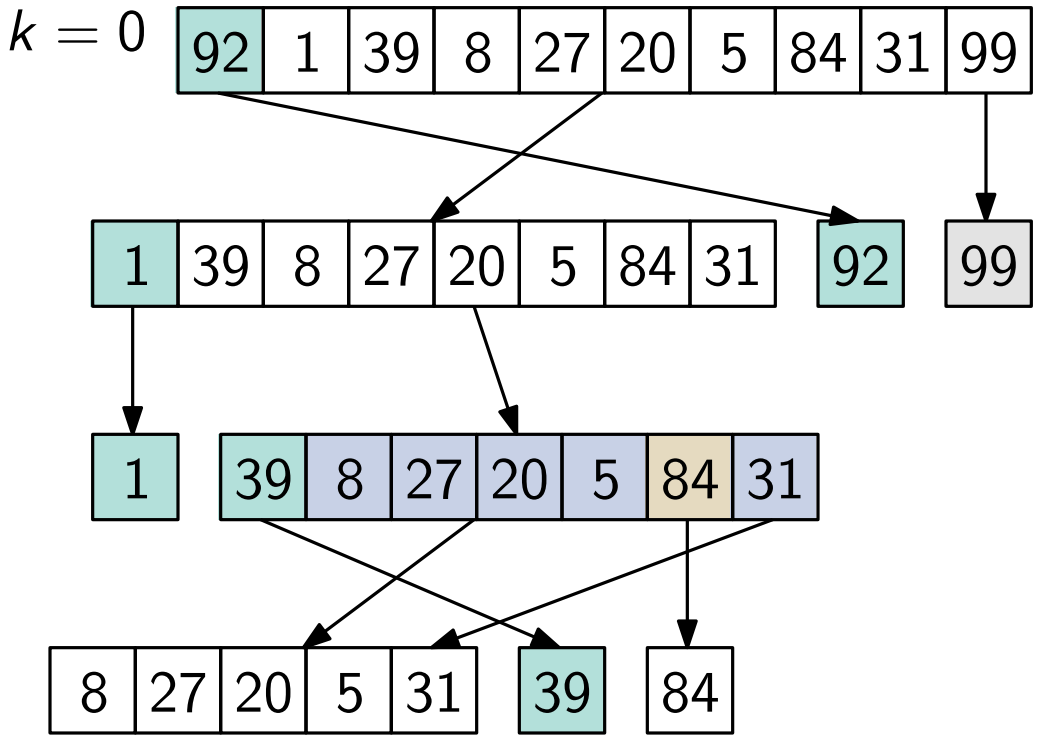
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



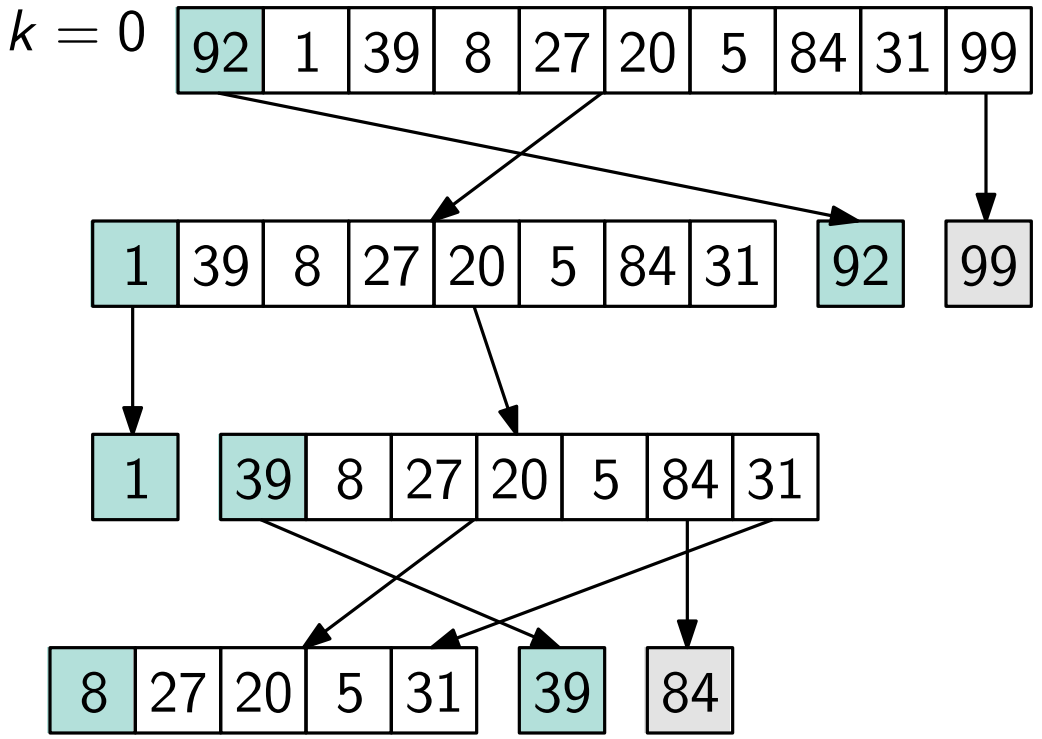
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



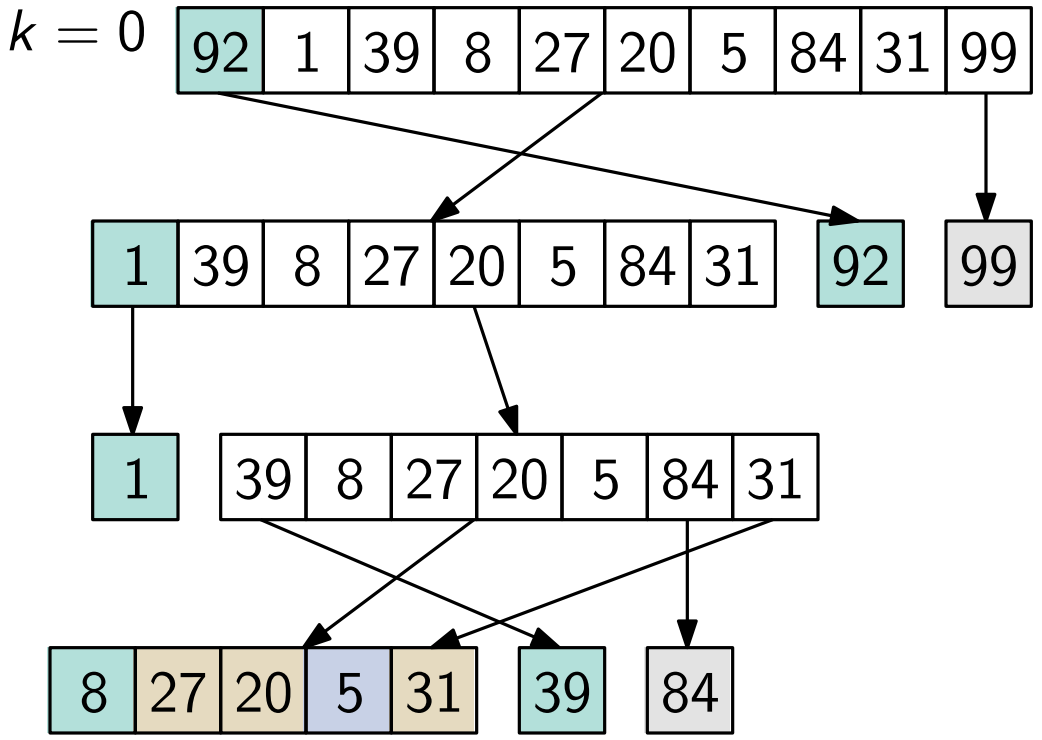
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



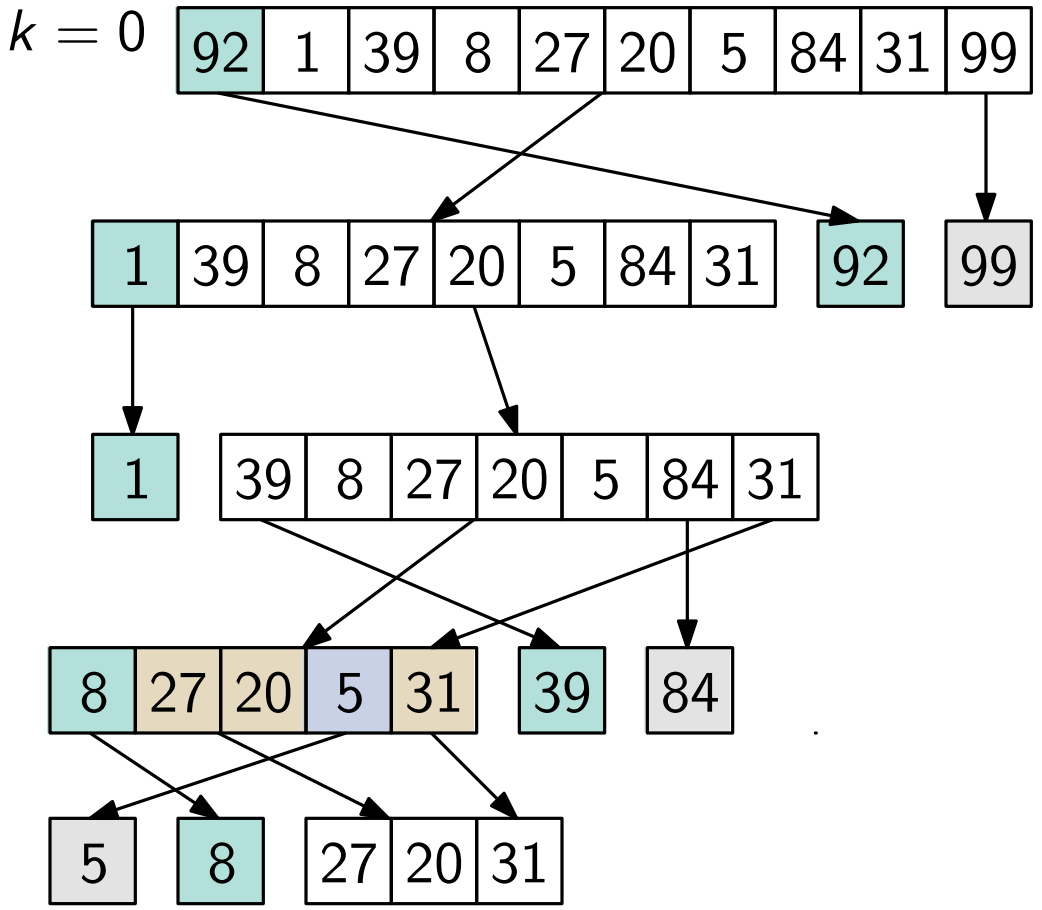
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



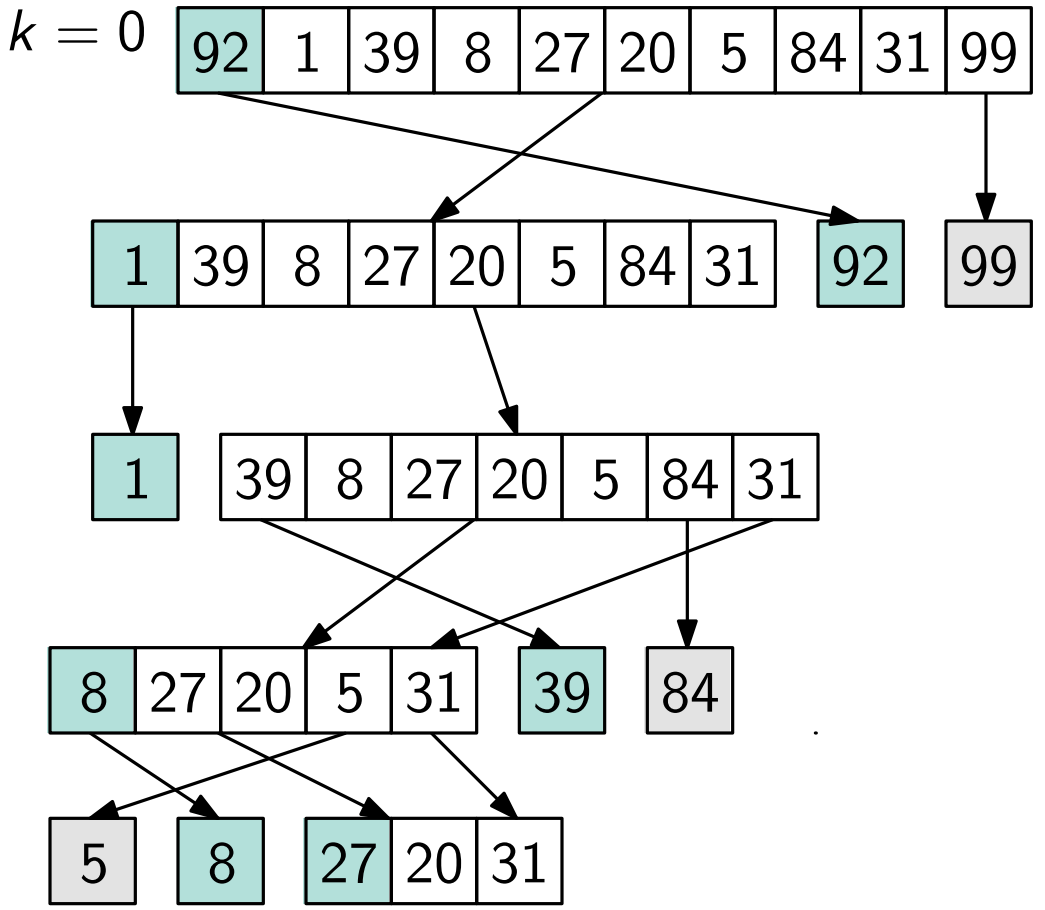
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



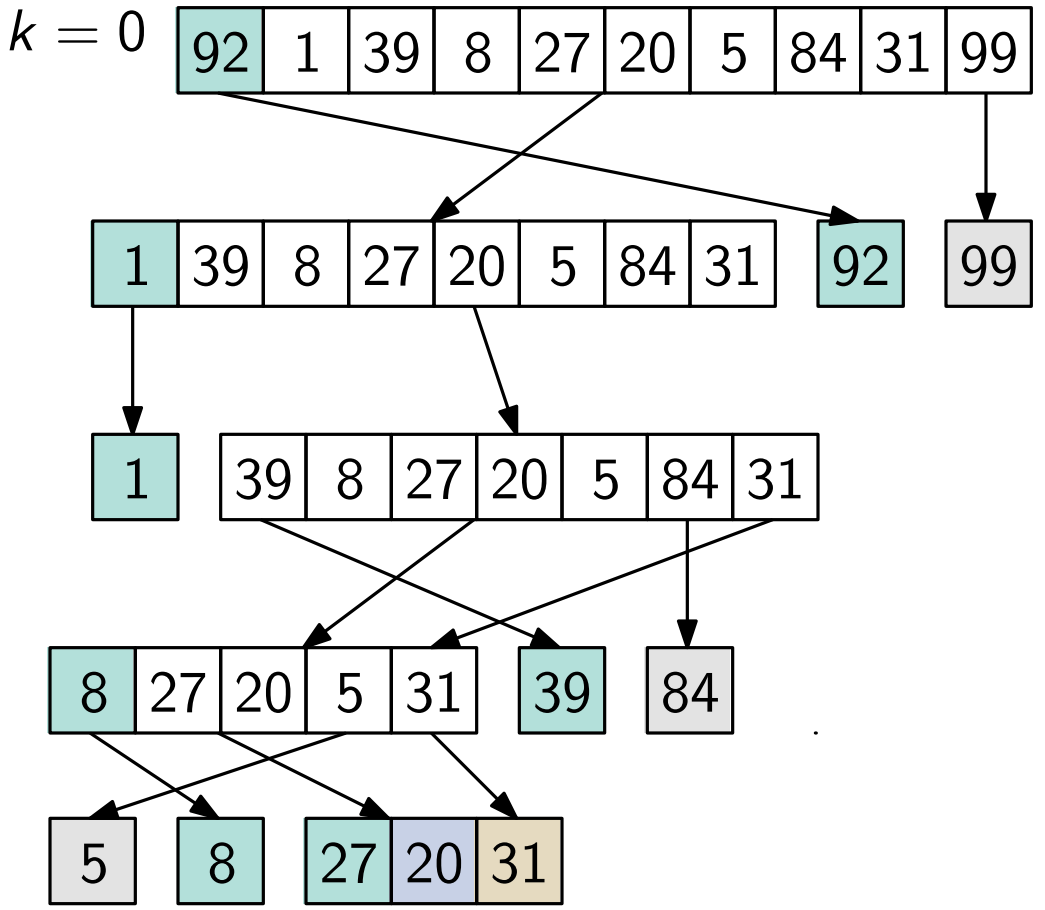
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



Quicksort

Zerlegung in Teilarrays

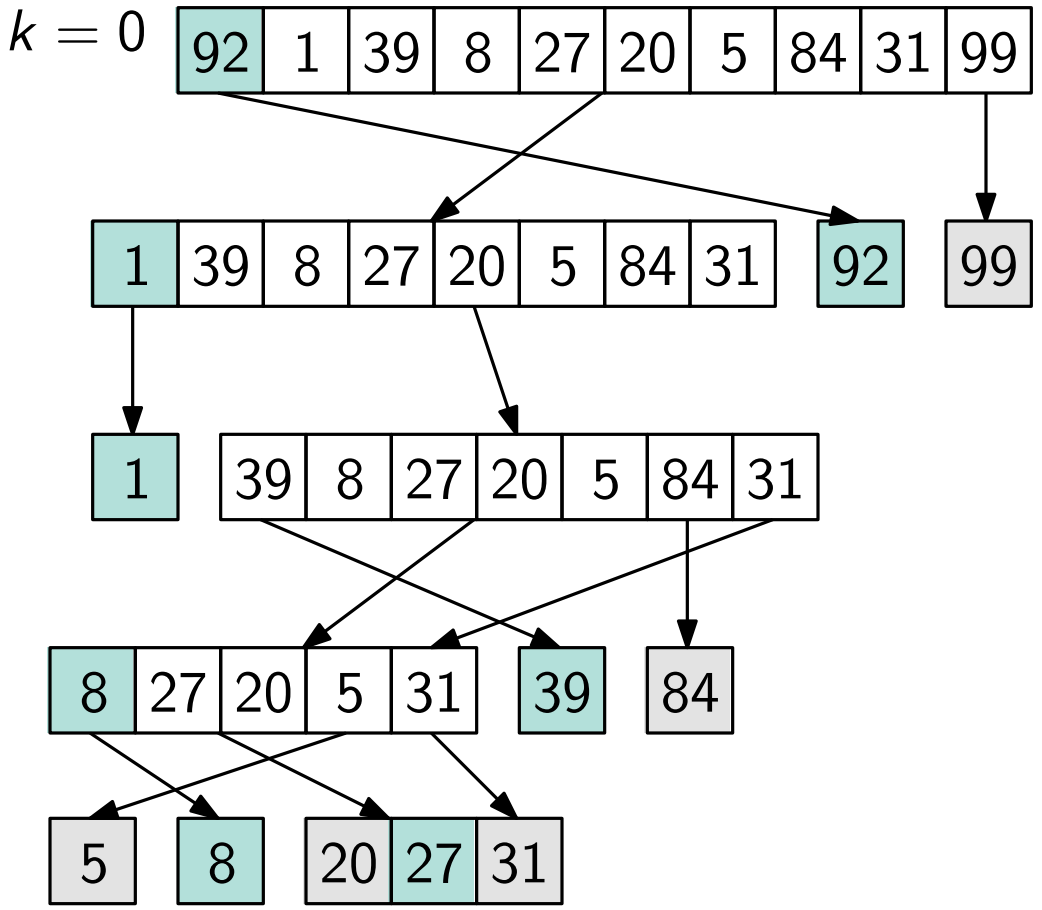
- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot

Best Case

- Pivot ist der Median
- Teilarrays werden halbiert
  - $\log(n)$ Ebenen
- Laufzeit: $\mathcal{O}(n \log n)$



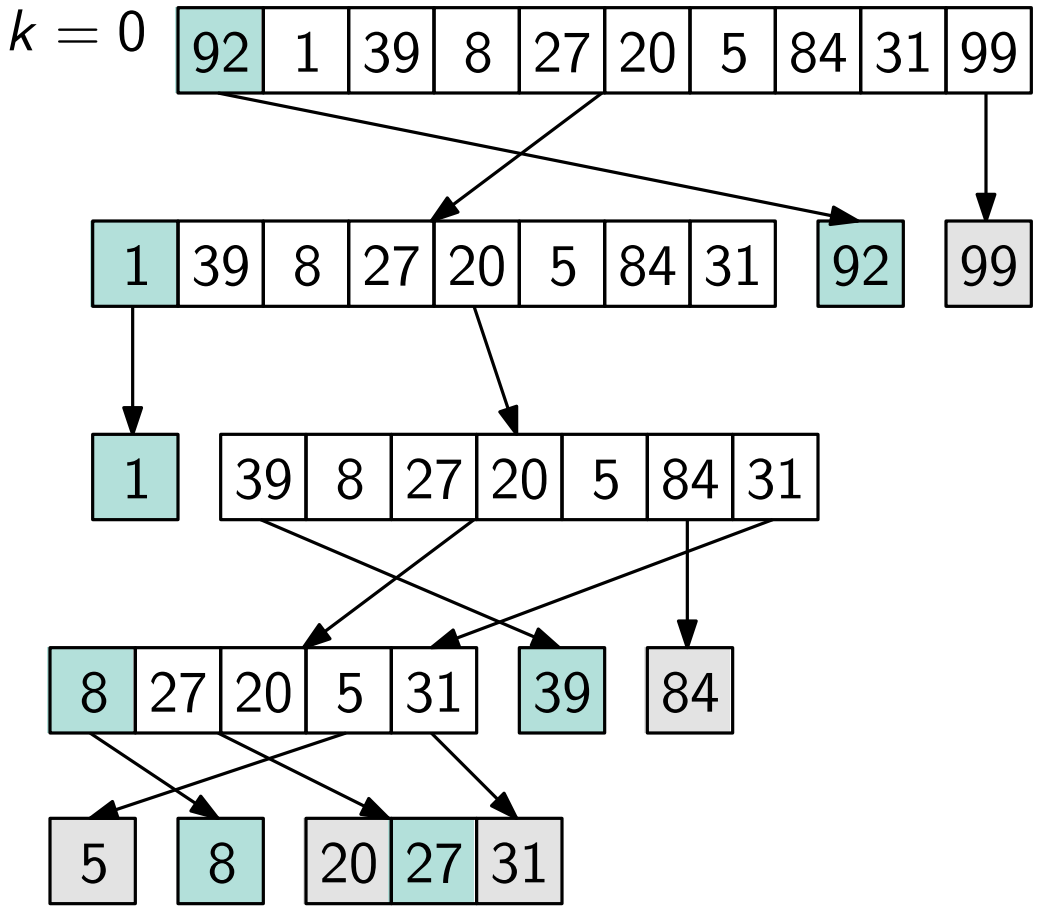
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



Best Case

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

Worst Case

- Element am Rand wird gewählt
- Je ein Element pro Schritt sortiert
- Laufzeit: $\mathcal{O}(n^2)$

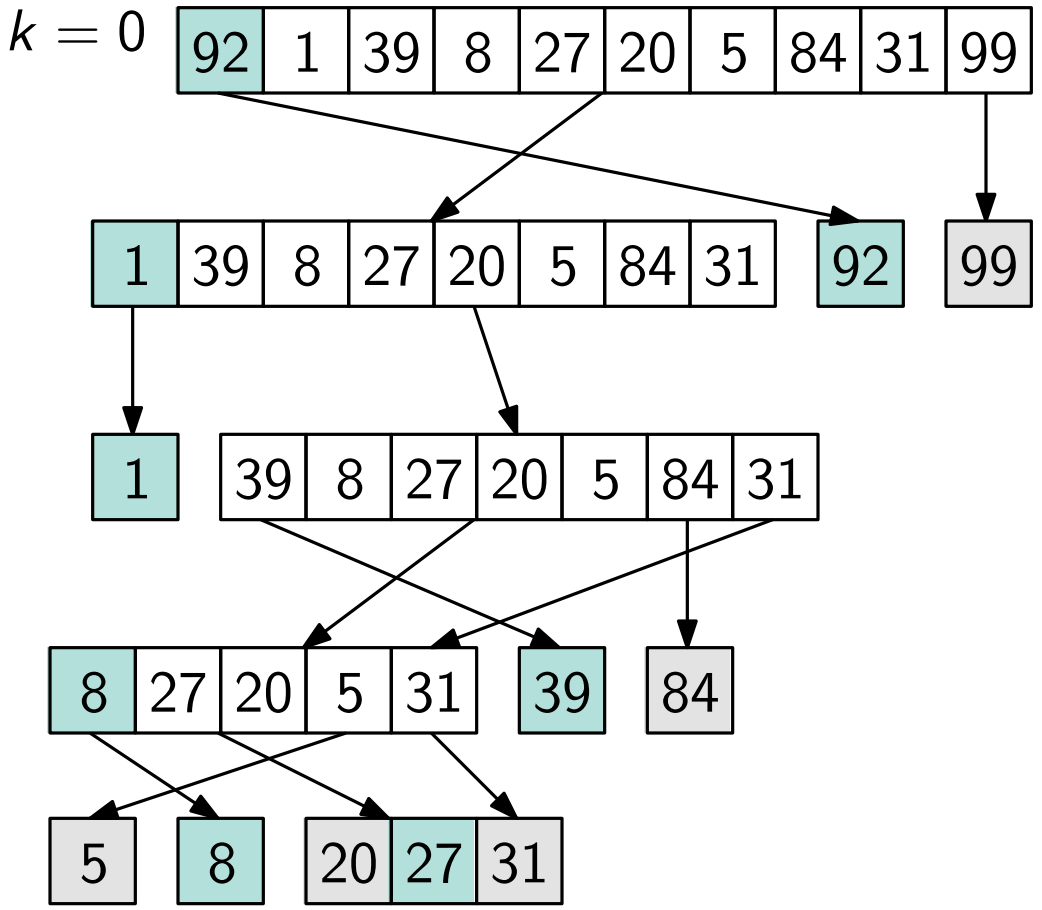
Quicksort

Zerlegung in Teilarrays

- wähle ein Element als **Pivot Element**
  - oftmals das k-te Element

Links: Elemente, die kleiner sind als das Pivot

Rechts: Elemente, die größer sind als das Pivot



Best Case

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

Worst Case

- Laufzeit: $\mathcal{O}(n^2)$

Average Case

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

Quickselect

Gesucht

- das k-kleinste (größte) Element

Quickselect

Gesucht

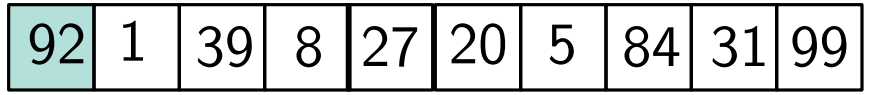
- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab

Quickselect

Gesucht

- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab

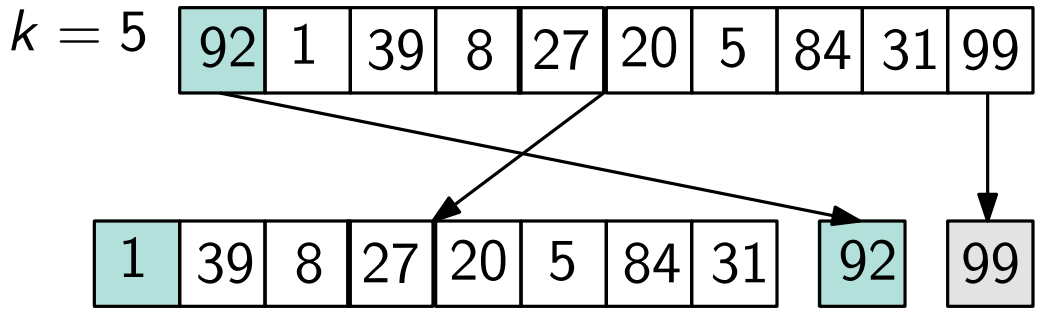
$k = 5$



Quickselect

Gesucht

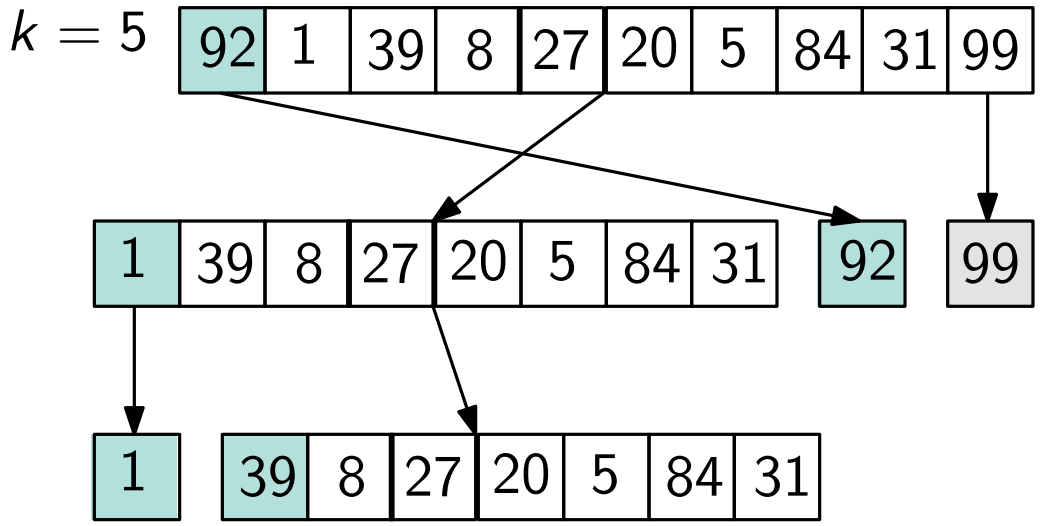
- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab



Quickselect

Gesucht

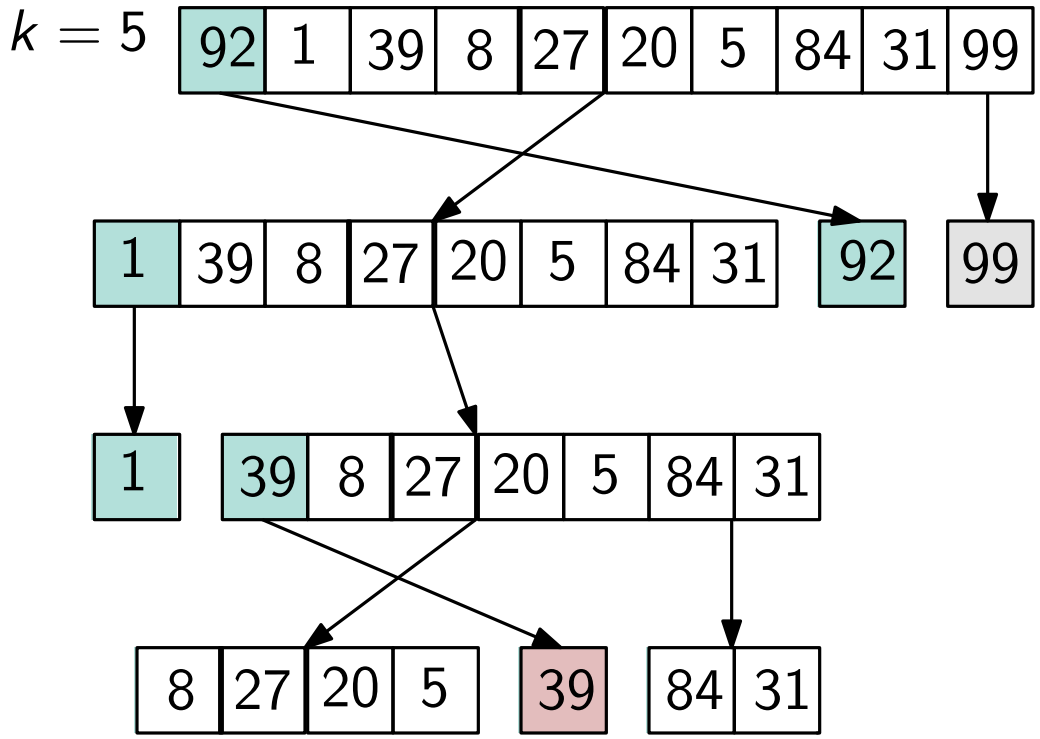
- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab



Quickselect

Gesucht

- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab



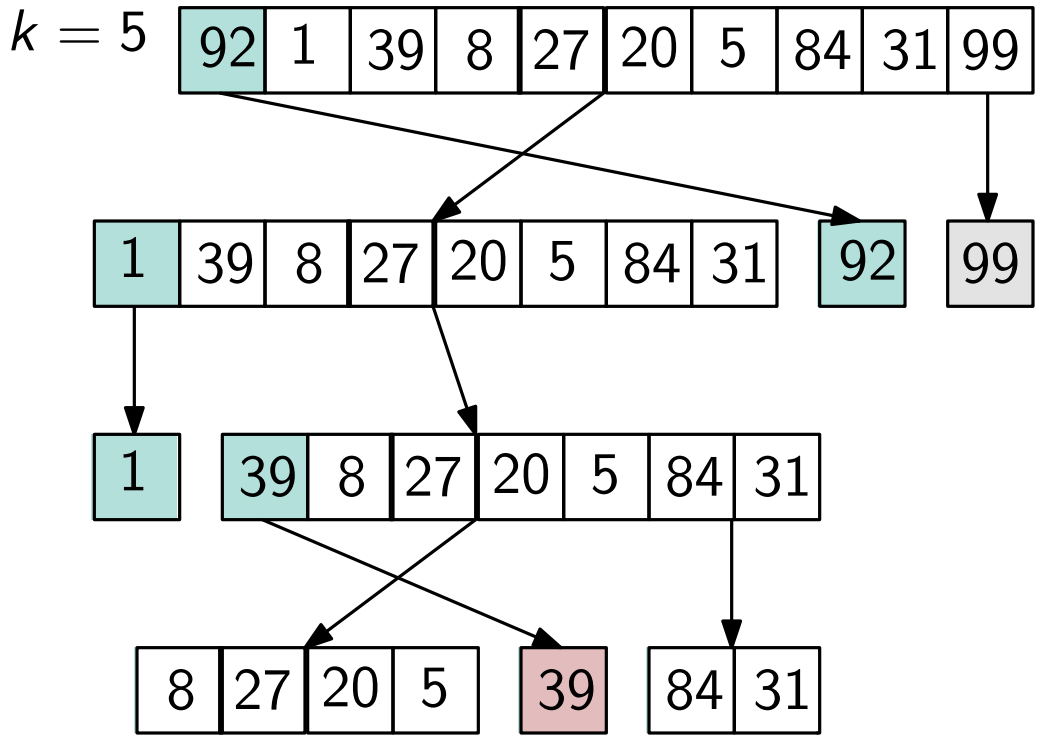
Quickselect

Gesucht

- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab

Best Case

- Pivot verringert Suchsatz sinnvoll
- in $\mathcal{O}(n)$



Quickselect

Gesucht

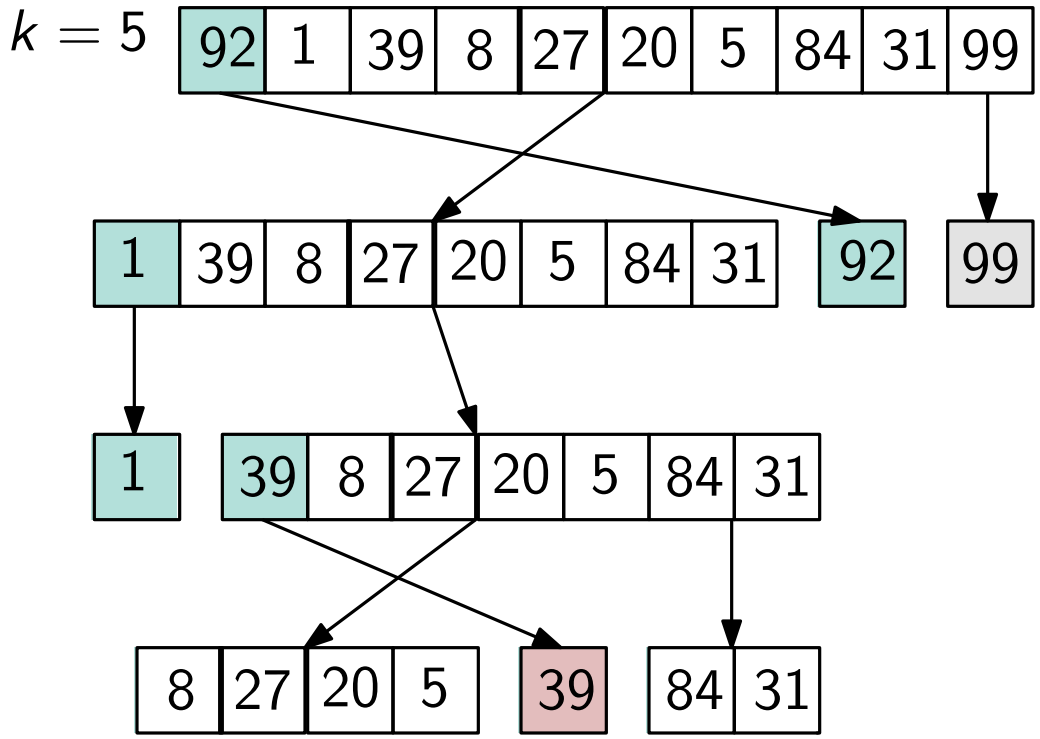
- das k-kleinste (größte) Element
- funktioniert ähnlich zu Quicksort
  - wähle Pivot-Element
  - sortiere ob Elemente größer/kleiner sind
  - steige nur in relevante Teilmenge ab

Best Case

- Pivot verringert Suchsatz sinnvoll
- in $\mathcal{O}(n)$

Worst Case

- Pivot verringert Suchsatz nur um 1
- in $\mathcal{O}(n^2)$



Zusammenfassung vergleichsbasiertes Sortieren

untere Schranke: vergleichsbasierte Sortieralgorithmen benötigen $\Omega(n \log n)$ Vergleiche

Zusammenfassung vergleichsbasiertes Sortieren

untere Schranke: vergleichsbasierte Sortieralgorithmen benötigen $\Omega(n \log n)$ Vergleiche

- **Mergesort:** arbeite beim Zusammenfügen der Teillösungen
- **Quicksort:** arbeite beim Zerlegen in Teilprobleme

Zusammenfassung vergleichsbasiertes Sortieren

untere Schranke: vergleichsbasierte Sortieralgorithmen benötigen $\Omega(n \log n)$ Vergleiche

- **Mergesort:** arbeite beim Zusammenfügen der Teillösungen
- **Quicksort:** arbeite beim Zerlegen in Teilprobleme

	Best Case	Average Case	Worst Case	Stabilität
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	stabil
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	stabil
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	stabil
Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	i.A. nicht stabil

Zusammenfassung vergleichsbasiertes Sortieren

untere Schranke: vergleichsbasierte Sortieralgorithmen benötigen $\Omega(n \log n)$ Vergleiche

- **Mergesort:** arbeite beim Zusammenfügen der Teillösungen
- **Quicksort:** arbeite beim Zerlegen in Teilprobleme

	Best Case	Average Case	Worst Case	Stabilität
Bubble Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	stabil
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	stabil
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	stabil
Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	i.A. nicht stabil

Geht es besser?

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]

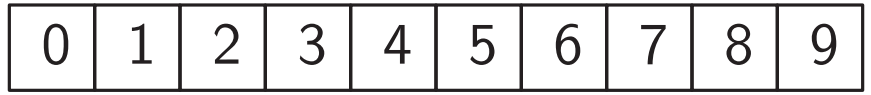
Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m

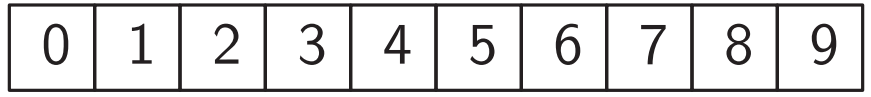
hier $m = 10$

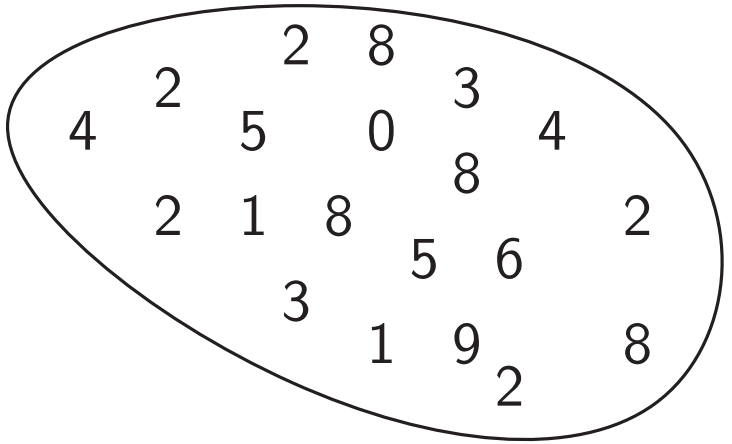


Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein

hier $m = 10$

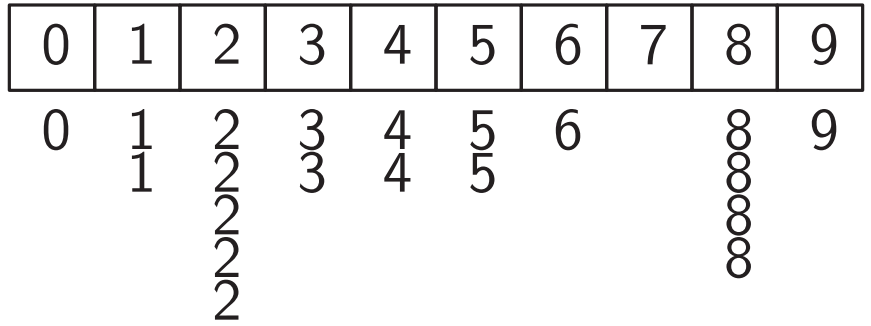




Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein

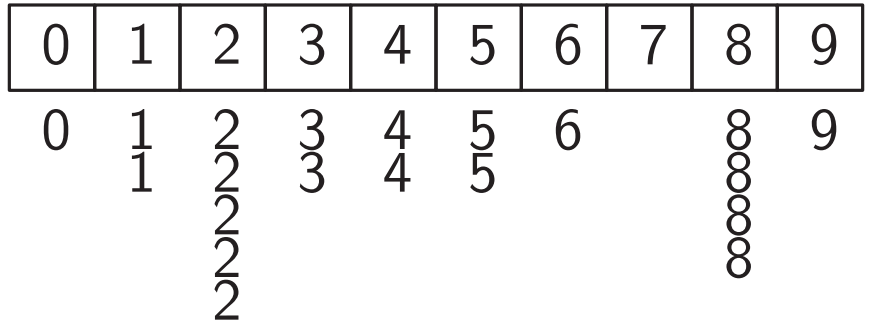
hier $m = 10$



Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

hier $m = 10$



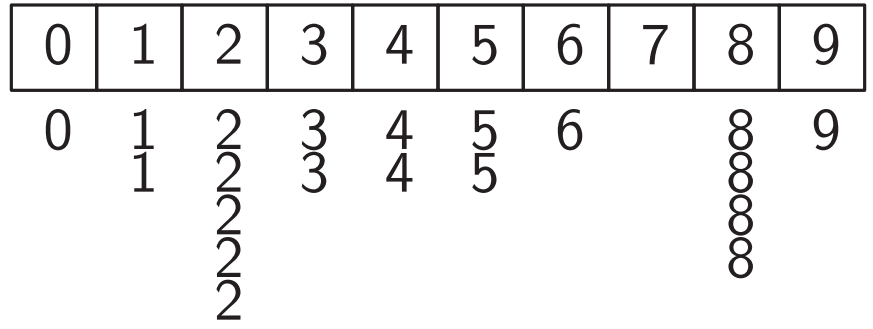
$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$



Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

PINGO: Wann ist Bucketsort sinnvoll?

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4 >$

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

PINGO: Wann ist Bucketsort sinnvoll?

- Personen nach Alter?

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

PINGO: Wann ist Bucketsort sinnvoll?

- Personen nach Alter?

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

PINGO: Wann ist Bucketsort sinnvoll?

- Personen nach Alter?
- Knoten nach Grad?

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

PINGO: Wann ist Bucketsort sinnvoll?

- Personen nach Alter?
- Knoten nach Grad?

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

PINGO: Wann ist Bucketsort sinnvoll?

- Personen nach Alter?
- Knoten nach Grad?
  - bei Mehrfachkanten?

Bucketsort

- Sinnvoll, bei gewisser Werteverteilung
- Elemente haben keys (können auch Elemente selbst sein)
  - ganzzahlige keys liegen im Wertebereich (0,m]
- m Buckets: Erstelle Array B der Größe m
  - sortiere Elemente anhand ihres Keys in B ein
  - Lies sortierte Folge aus B aus

Laufzeit: in $\mathcal{O}(n)$, wenn $m \in \mathcal{O}(n)$

hier $m = 10$

0	1	2	3	4	5	6	7	8	9
0	1 1	2 2 2 2 2	3 3	4 4	5 5	6		8 8 8 8 8	9

$< 0, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 8, 8, 9 >$

PINGO: Wann ist Bucketsort sinnvoll?

- Personen nach Alter?
- Knoten nach Grad?
  - bei Mehrfachkanten?

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern

LSD (Least Significant Digit)

- Sortiere Ziffern von "rechts nach links"

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern

LSD (Least Significant Digit)

- Sortiere Ziffern von "rechts nach links"

MSD (Most Significant Digit)

- Sortiere Ziffern von "links nach rechts"

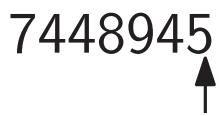
LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

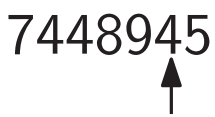


LSD

- sortiere nach Ziffer x_0

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$



LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2
- bis nach allen c Ziffern sortiert wurde

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2
- bis nach allen c Ziffern sortiert wurde

Aufgabe

Sortiert die folgende Sequenz mit LSD (Angeben nach jeder Ziffer)

001, 101, 102, 520, 032, 045, 097, 231, 981, 273, 186, 020, 100

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2
- bis nach allen c Ziffern sortiert wurde

Aufgabe

Sortiert die folgende Sequenz mit LSD (Angeben nach jeder Ziffer)

001, 101, 102, 520, 032, 045, 097, 231, 981, 273, 186, 020, 100

Ziffer 0

52**0**, 02**0**, 10**0**, 00**1**, 10**1**, 23**1**, 98**1**, 10**2**, 03**2**, 27**3**, 04**5**, 18**6**, 19**7**

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2
- bis nach allen c Ziffern sortiert wurde

Aufgabe

Sortiert die folgende Sequenz mit LSD (Angeben nach jeder Ziffer)

001, 101, 102, 520, 032, 045, 097, 231, 981, 273, 186, 020, 100

Ziffer 1

1**0**0, 0**0**1, 1**0**1, 1**0**2, 5**2**0, 0**2**0, 2**3**1, 0**3**2, 0**4**5, 2**7**3, 9**8**1, 1**8**6, 1**9**7

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2
- bis nach allen c Ziffern sortiert wurde

Aufgabe

Sortiert die folgende Sequenz mit LSD (Angeben nach jeder Ziffer)

001, 101, 102, 520, 032, 045, 097, 231, 981, 273, 186, 020, 100

Ziffer 2

001, **0**20, **0**32, **0**45, **1**00, **1**01, **1**02, **1**86, **1**97, **2**31, **2**73, **5**20, **9**81

LSD-/MSD Radix Sort

- Bucketsort auf Ziffern
- Eingabe: ganze Zahlen aus $(0, n^c]$

7448945

LSD

- sortiere nach Ziffer x_0
- sortiere **stabil** nach Ziffer x_1
- sortiere **stabil** nach Ziffer x_2
- bis nach allen c Ziffern sortiert wurde

Laufzeit

$\mathcal{O}(c \cdot n)$

- Kann sich je nach Basis ändern

vergleichsbasiertes vs ganzzahliges Sortieren

Vorteile ganzzahliges Sortieren

- lineare Laufzeit

vergleichsbasiertes vs ganzzahliges Sortieren

Vorteile ganzzahliges Sortieren

- lineare Laufzeit

Vorteile vergleichsbasiertes Sortieren

- weniger Annahmen
  - Man braucht keinen beschränkten Schlüssel

vergleichsbasiertes vs ganzzahliges Sortieren

Vorteile ganzzahliges Sortieren

- lineare Laufzeit

Vorteile vergleichsbasiertes Sortieren

- weniger Annahmen
  - Man braucht keinen beschränkten Schlüssel
- robust gegen beliebige Eingabeverteilung

vergleichsbasiertes vs ganzzahliges Sortieren

Vorteile ganzzahliges Sortieren

- lineare Laufzeit

Vorteile vergleichsbasiertes Sortieren

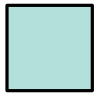
- weniger Annahmen
  - Man braucht keinen beschränkten Schlüssel
- robust gegen beliebige Eingabeverteilung
- Eingabe muss nur lexikografisch sortierbar sein
  - bei ganzzahlig: numerische Eingabe

(binäre) Heaps

- Baum, das heißt es gibt

(binäre) Heaps

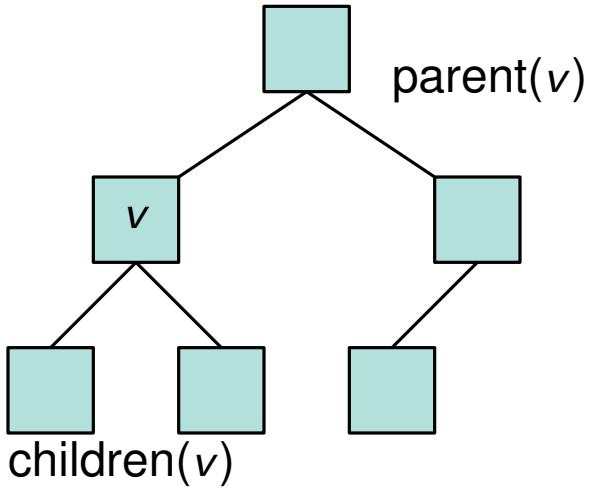
- Baum, das heißt es gibt
  - genau einen Wurzelknoten



Wurzel

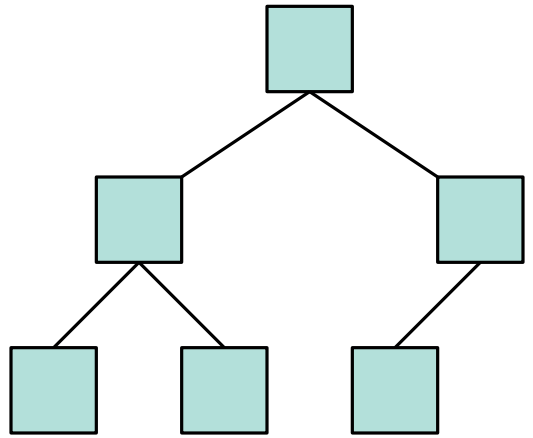
(binäre) Heaps

- Baum, das heißt es gibt
  - genau einen Wurzelknoten
  - innere Knoten
    - haben sowohl einen **Elternknoten** als auch **Kindknoten**



(binäre) Heaps

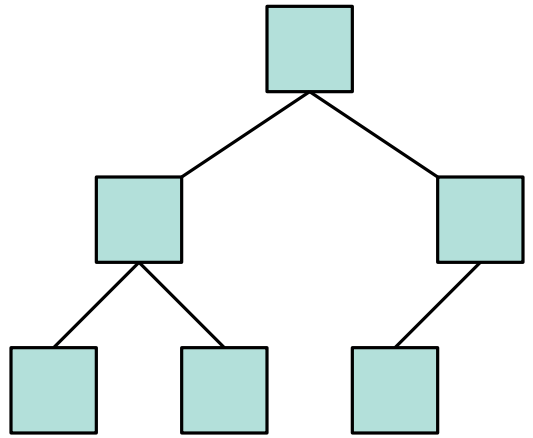
- Baum, das heißt es gibt
  - genau einen Wurzelknoten
  - innere Knoten
    - haben sowohl einen **Elternknoten** als auch **Kindknoten**
  - Blattknoten
    - haben keine Kinder



Blattknoten

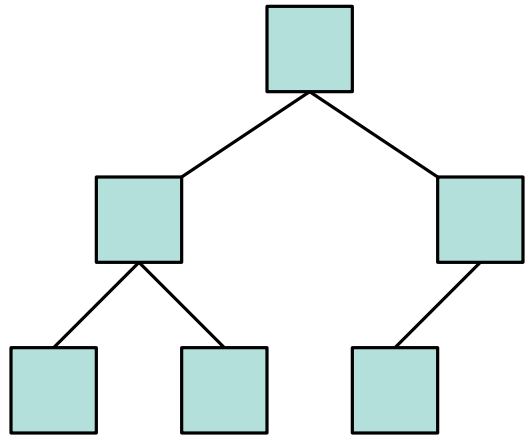
(binäre) Heaps

- Baum, das heißt es gibt
  - genau einen Wurzelknoten
  - innere Knoten
    - haben sowohl einen **Elternknoten** als auch **Kindknoten**
  - Blattknoten
    - haben keine Kinder
- Heap hat $\Theta(\log n)$ viele Layer
  - wobei fehlende Knoten im letzten Layer rechts unten sind



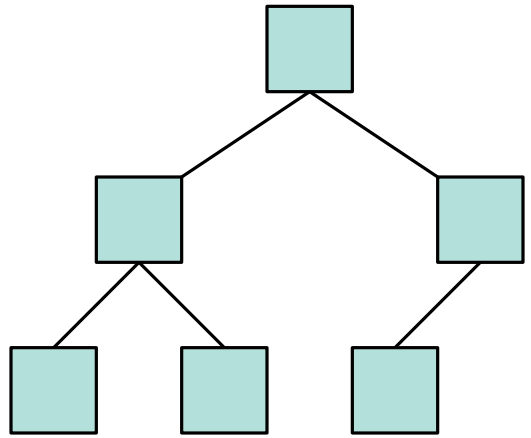
(binäre) Heaps

- Baum, das heißt es gibt
  - genau einen Wurzelknoten
  - innere Knoten
    - haben sowohl einen **Elternknoten** als auch **Kindknoten**
  - Blattknoten
    - haben keine Kinder
- Heap hat $\Theta(\log n)$ viele Layer
  - wobei fehlende Knoten im letzten Layer rechts unten sind
- Knoten können viele Dinge speichern
  - zur Vereinfachung: Knoten wird durch seine Priorität dargestellt



(binäre) Heaps

- Baum, das heißt es gibt
  - genau einen Wurzelknoten
  - innere Knoten
    - haben sowohl einen **Elternknoten** als auch **Kindknoten**
  - Blattknoten
    - haben keine Kinder
- Heap hat $\Theta(\log n)$ viele Layer
  - wobei fehlende Knoten im letzten Layer rechts unten sind
- Knoten können viele Dinge speichern
  - zur Vereinfachung: Knoten wird durch seine Priorität dargestellt



Heap-Eigenschaft:parent(v) $\leq$ v

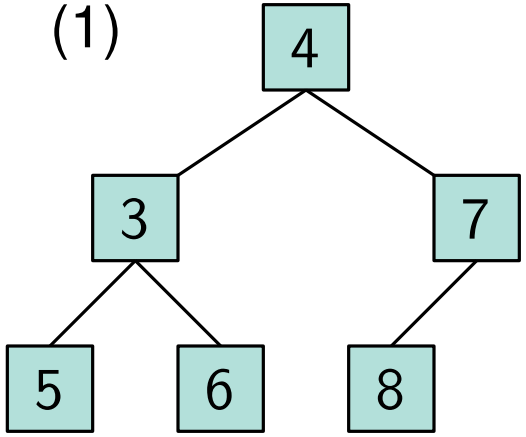
(binäre) Heaps

PINGO: was ist ein Heap?

(binäre) Heaps

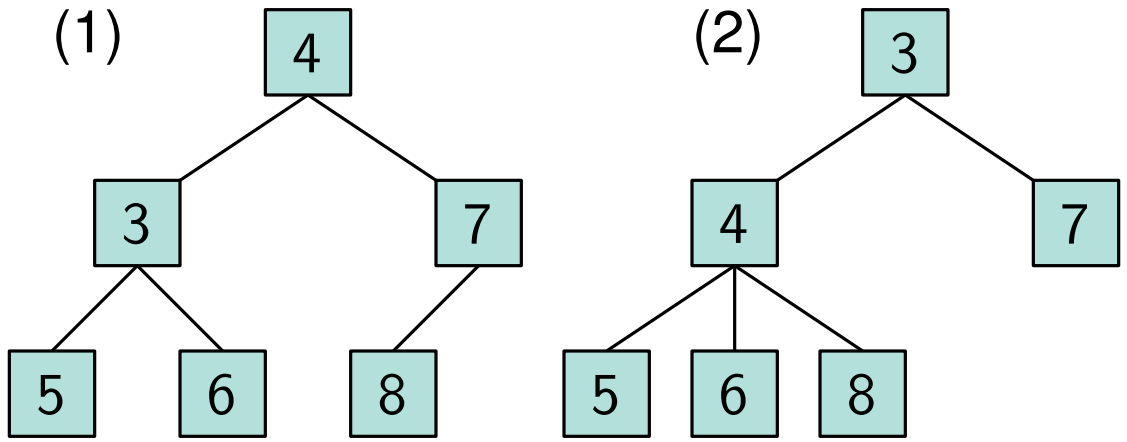
PINGO: was ist ein Heap?

(1)



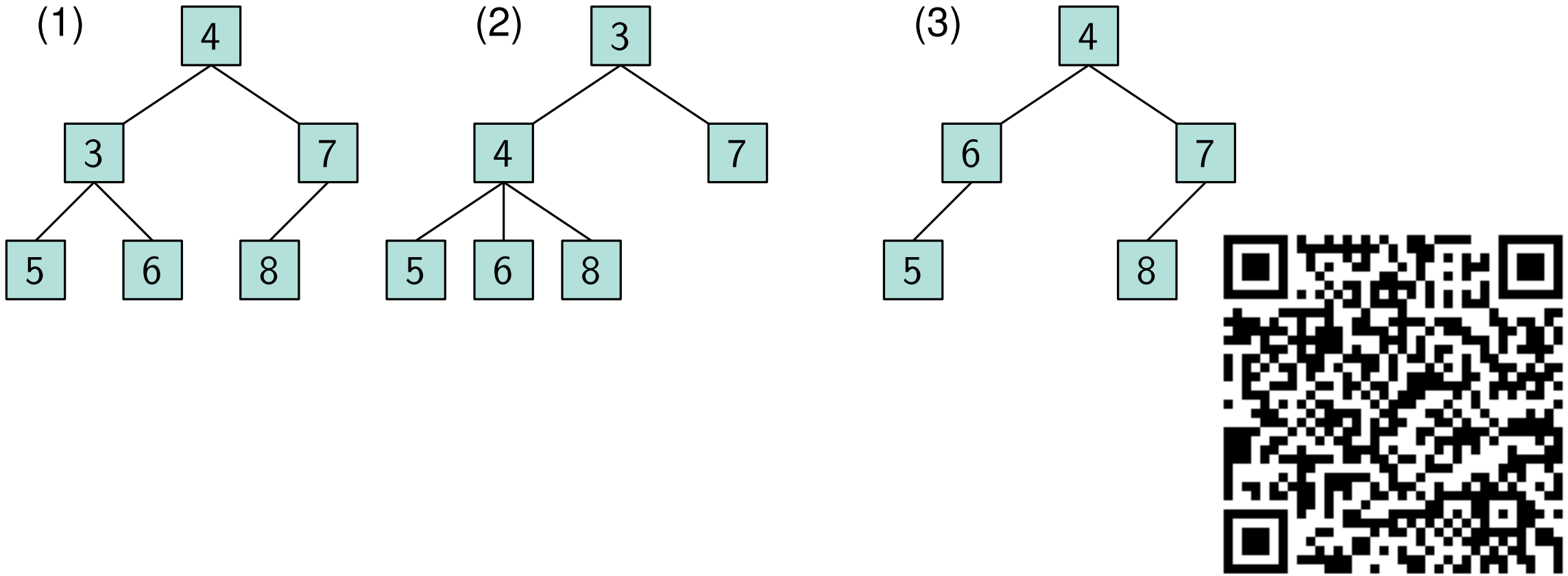
(binäre) Heaps

PINGO: was ist ein Heap?



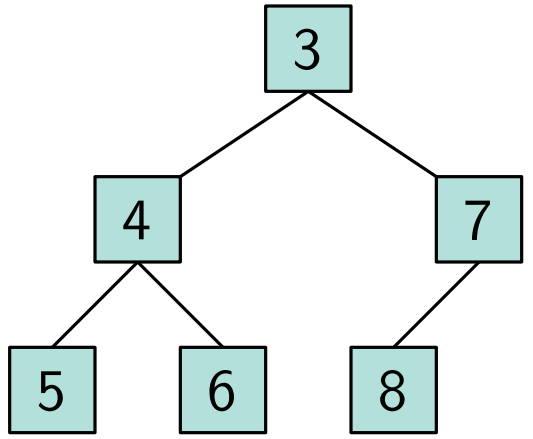
(binäre) Heaps

PINGO: was ist ein Heap?



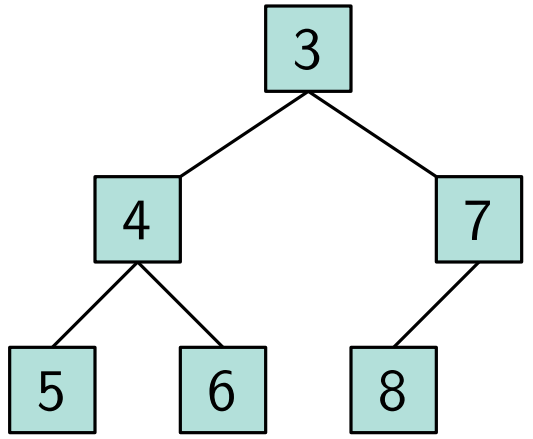
Heap Repräsentation

- mittels Array



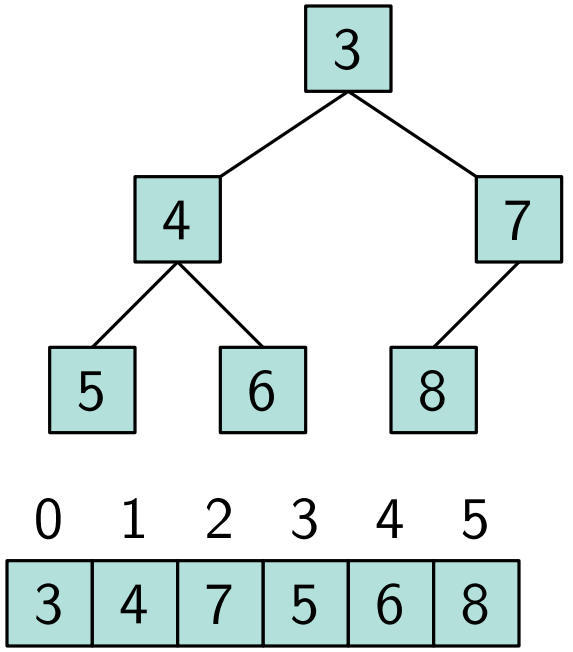
Heap Repräsentation

- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab



Heap Repräsentation

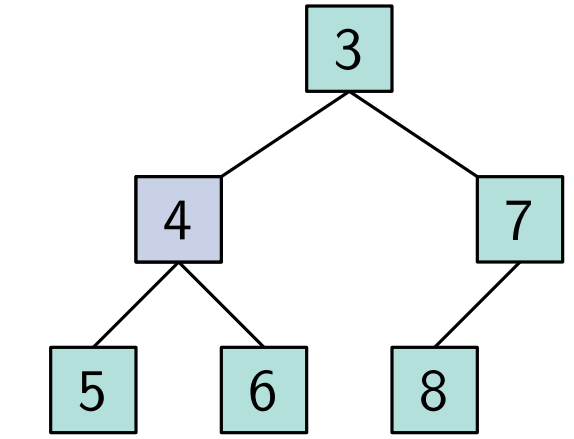
- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab



Heap Repräsentation

- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab

Kind- und Elternknoten



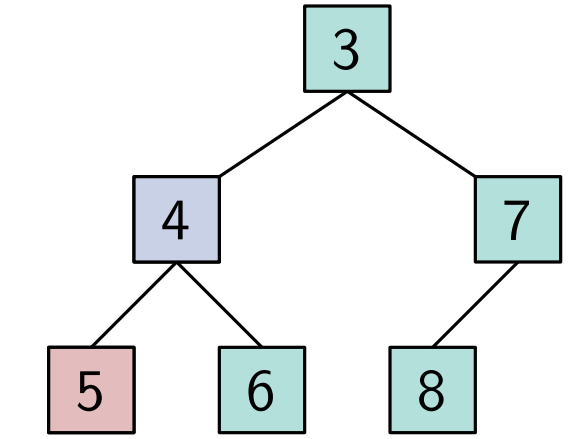
4 ist an Stelle 1

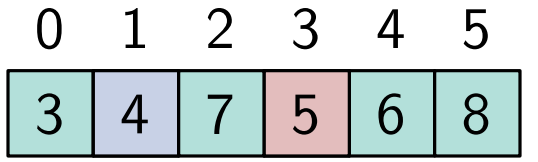
Heap Repräsentation

- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab

Kind- und Elternknoten

- linkes Kind: $2v + 1$





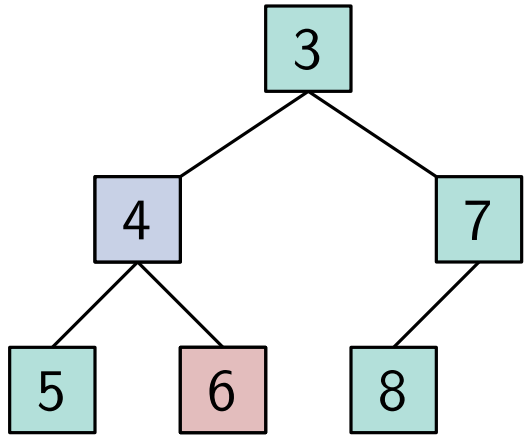
linkes Kind: $2 \cdot 1 + 1 = 3$

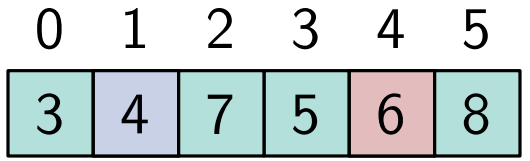
Heap Repräsentation

- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab

Kind- und Elternknoten

- linkes Kind: $2v + 1$
- rechtes Kind: $2v + 2$





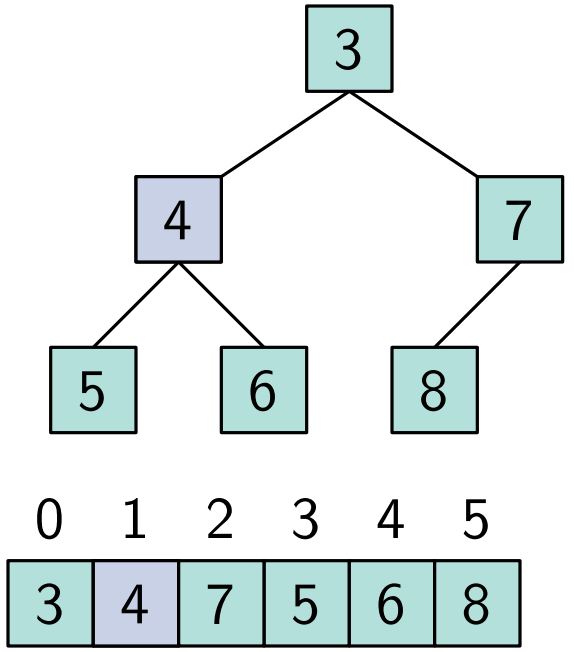
rechtes Kind: $2 \cdot 1 + 2 = 4$

Heap Repräsentation

- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab

Kind- und Elternknoten

- linkes Kind: $2v + 1$
- rechtes Kind: $2v + 2$
- Elternknoten: $\lfloor \frac{v-1}{2} \rfloor$

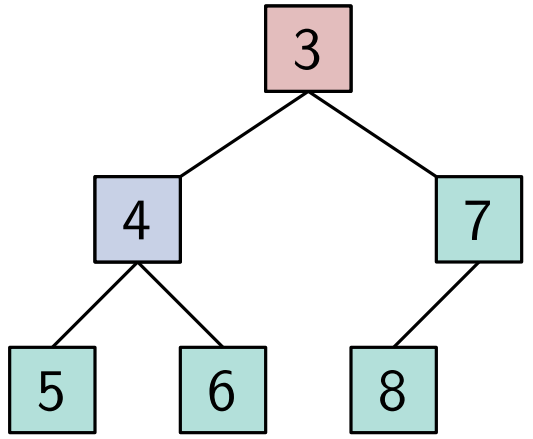


Heap Repräsentation

- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab

Kind- und Elternknoten

- linkes Kind: $2v + 1$
- rechtes Kind: $2v + 2$
- Elternknoten: $\lfloor \frac{v-1}{2} \rfloor$



0	1	2	3	4	5
3	4	7	5	6	8

Elternknoten: $\lfloor \frac{1-1}{2} \rfloor = 0$

Heap Repräsentation

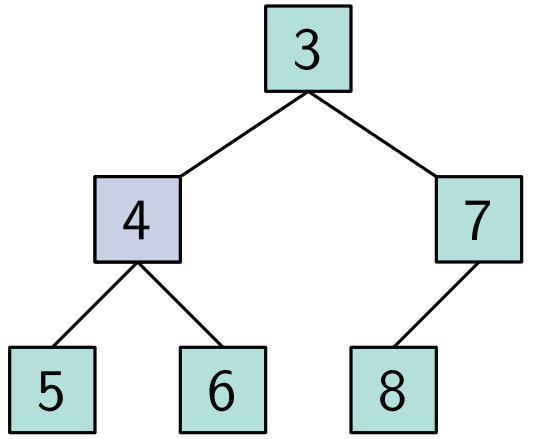
- mittels Array
  - starte bei Wurzelknoten
  - lese Knoten Layer-weise von links nach rechts ab

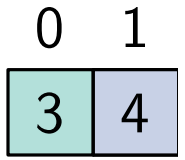
Kind- und Elternknoten

- linkes Kind: $2v + 1$
- rechtes Kind: $2v + 2$
- Elternknoten: $\lfloor \frac{v-1}{2} \rfloor$

PINGO: Wie viele binäre Heaps gibt es für

$A = \{9, 5, 3, 4\}$



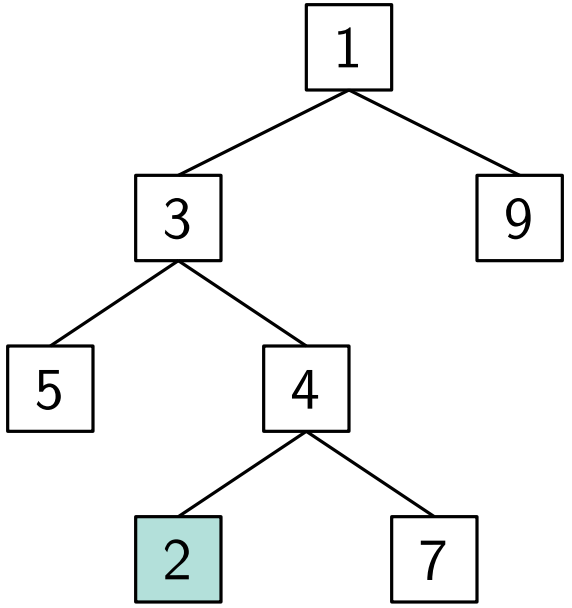


Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt

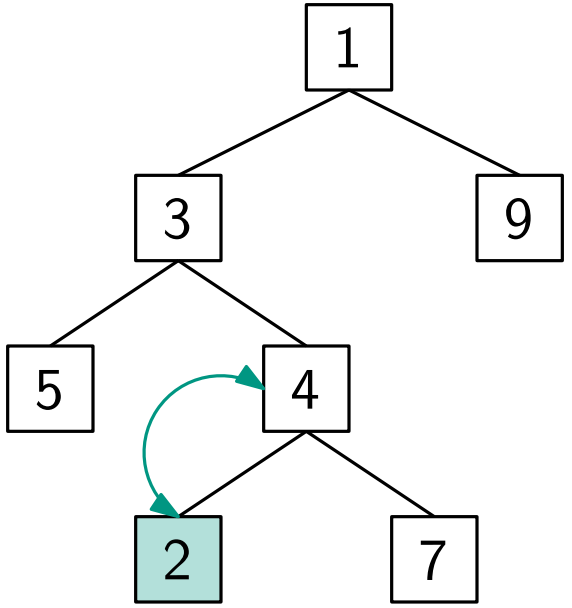
Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt



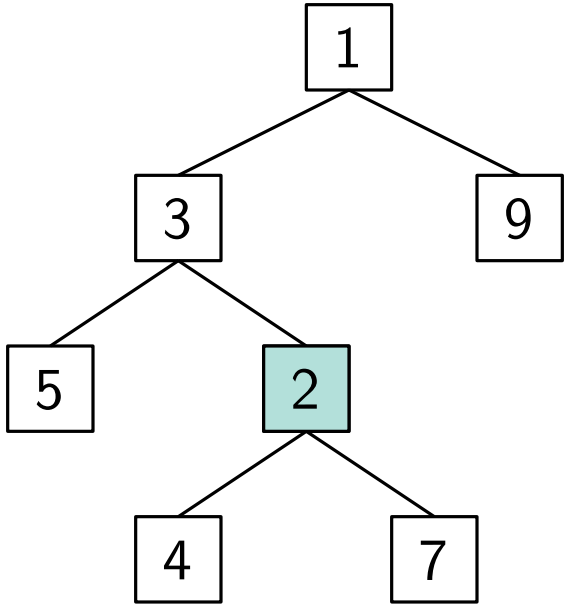
Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt



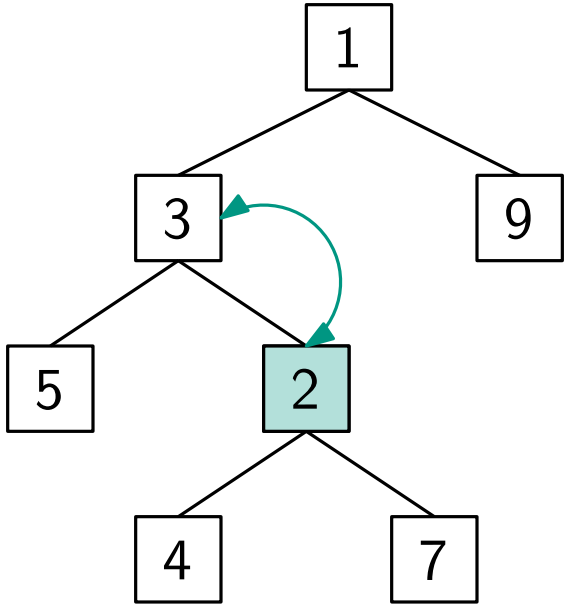
Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt



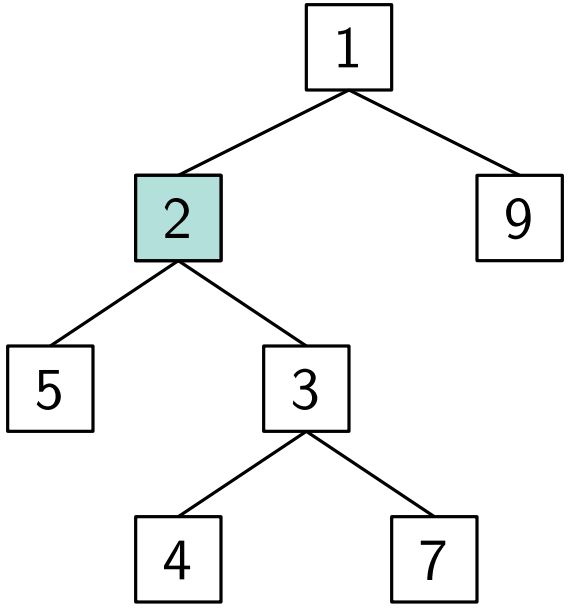
Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt



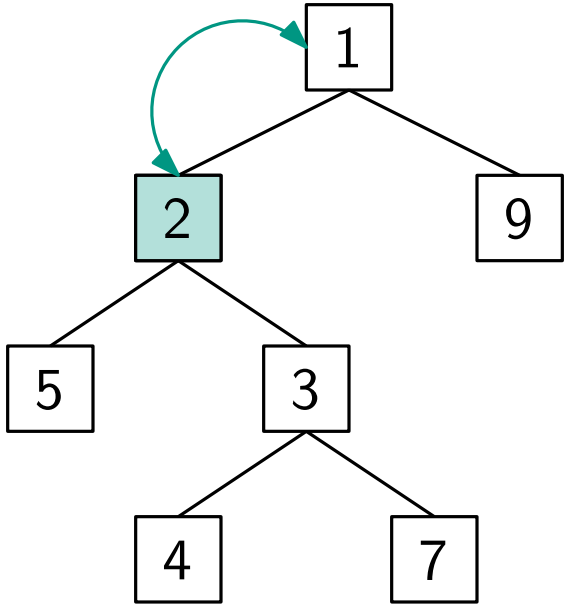
Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt



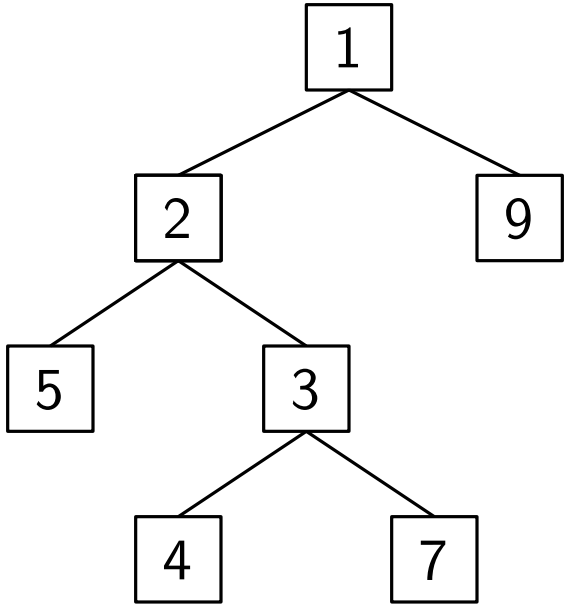
Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt



Hilfsoperation: bubbleUp

- Falls Element Heap-Eigenschaft nicht erfüllt
  - Vergleiche Element mit seinem Parent
  - tausche, falls Parent größer
  - Wiederhole solange bis dies nicht mehr gilt

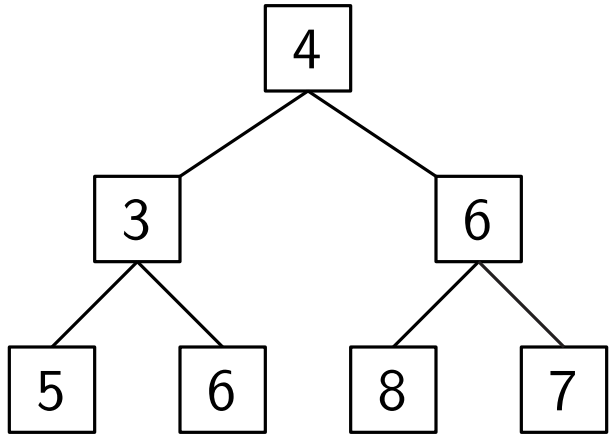


Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig

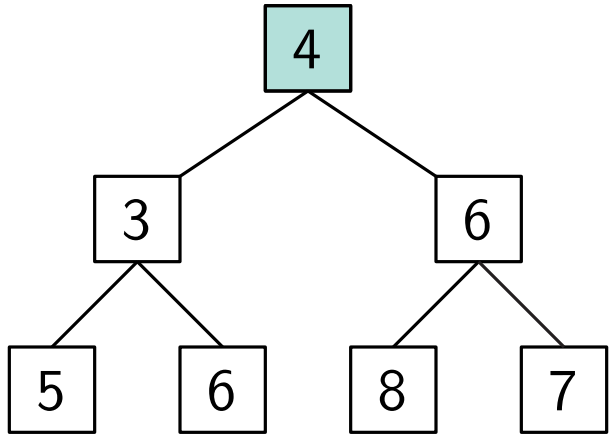
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



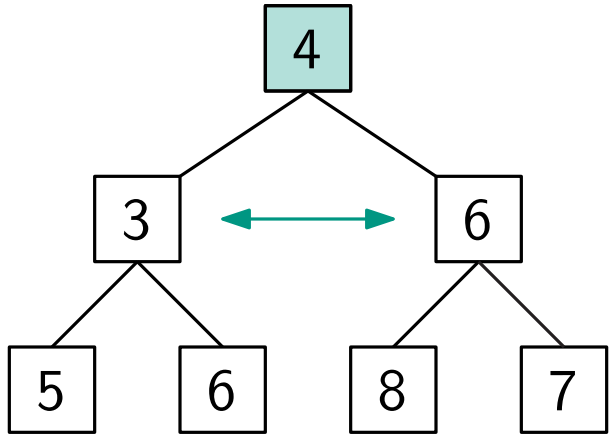
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



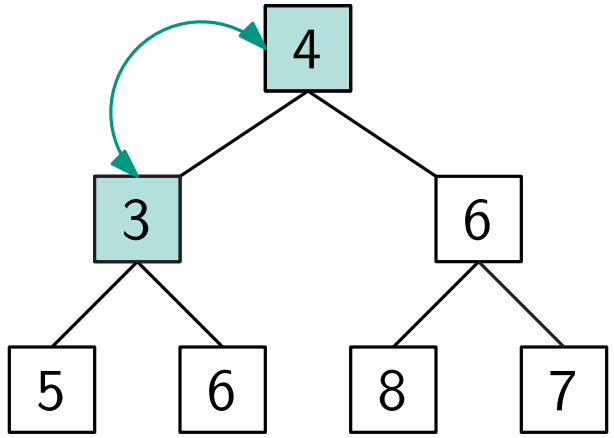
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



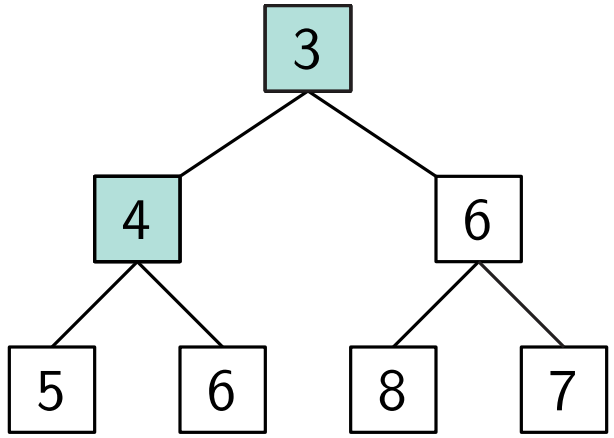
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



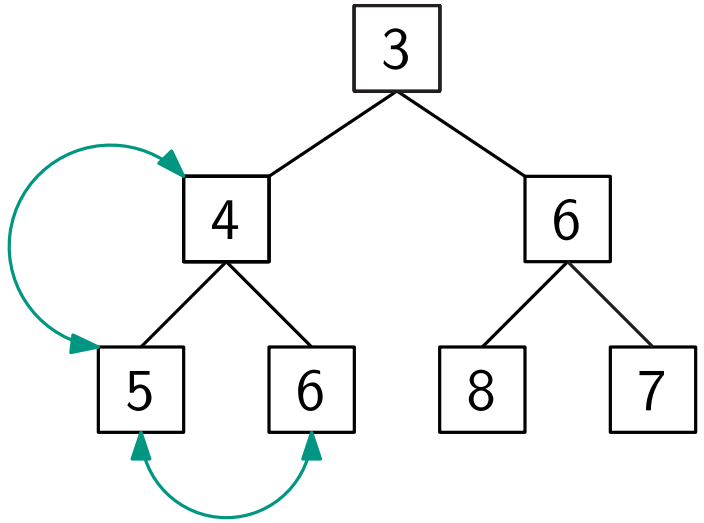
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



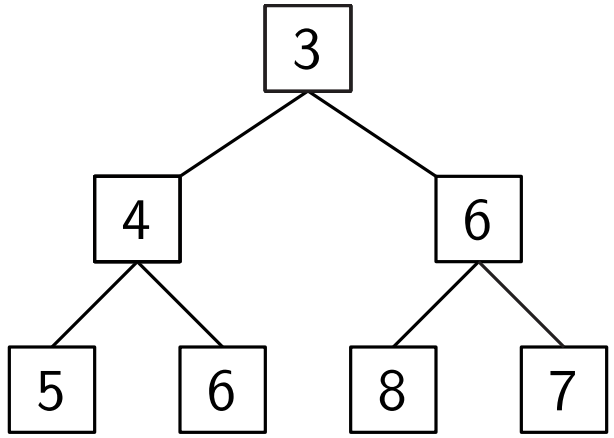
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



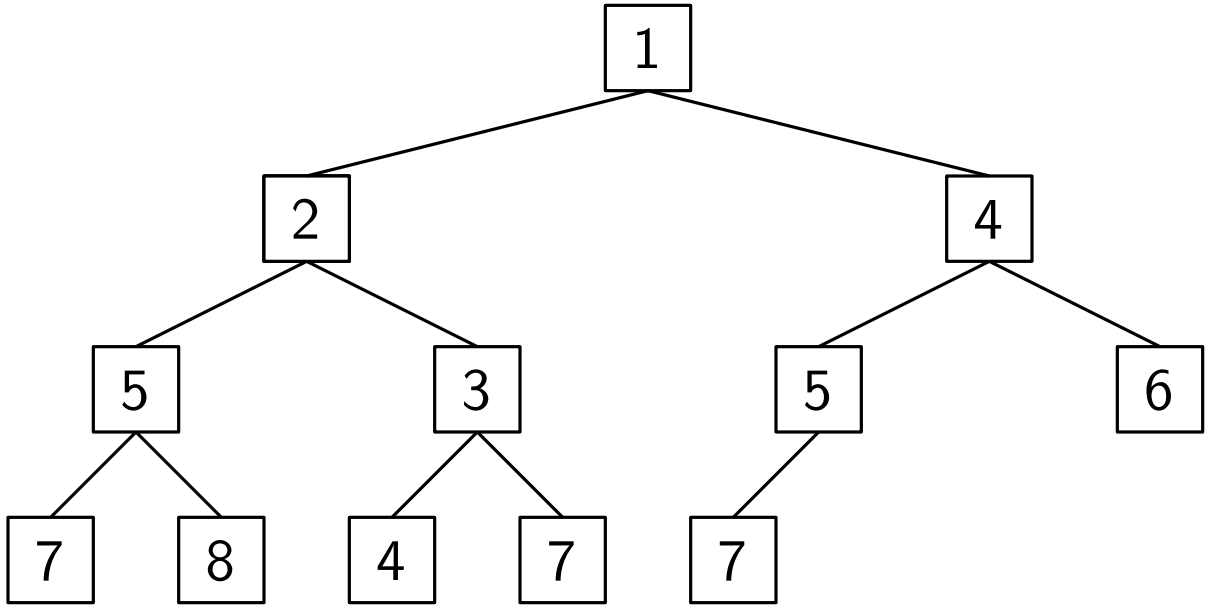
Hilfsoperation: sinkDown

- Falls Knoten v Blatt
  - wir sind fertig
- Falls Knoten v mindestens ein Kind hat
  - Bestimme das Minimum des Knotens v und seiner Kinder
  - tausche v mit dem Minimum
  - Falls v selbst Minimum war
    - wir sind fertig



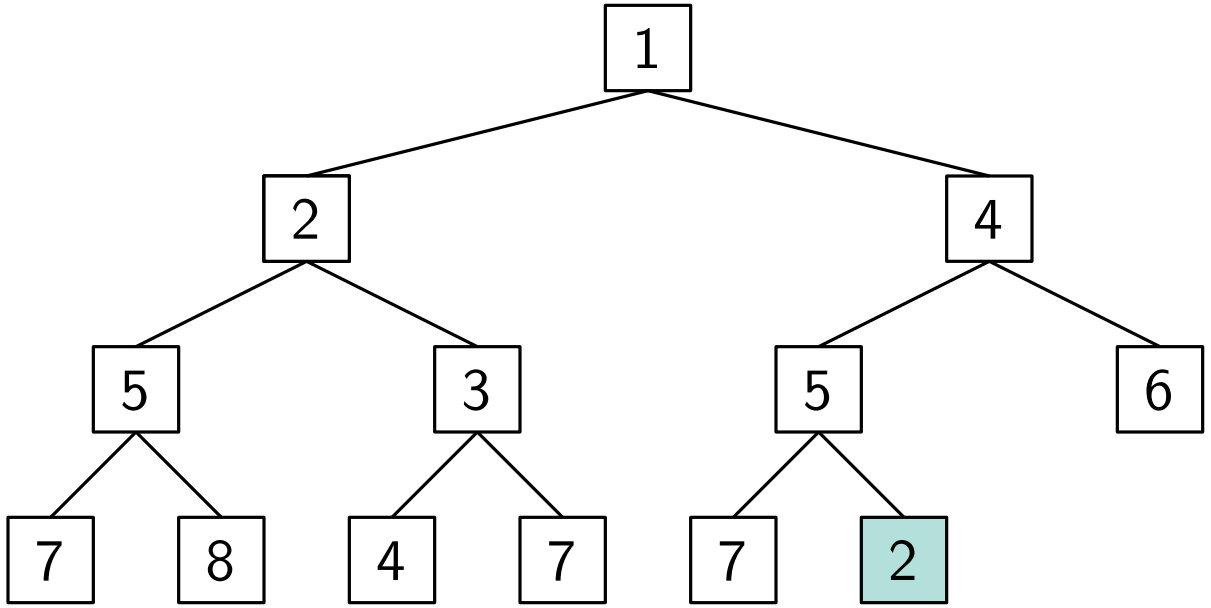
push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**



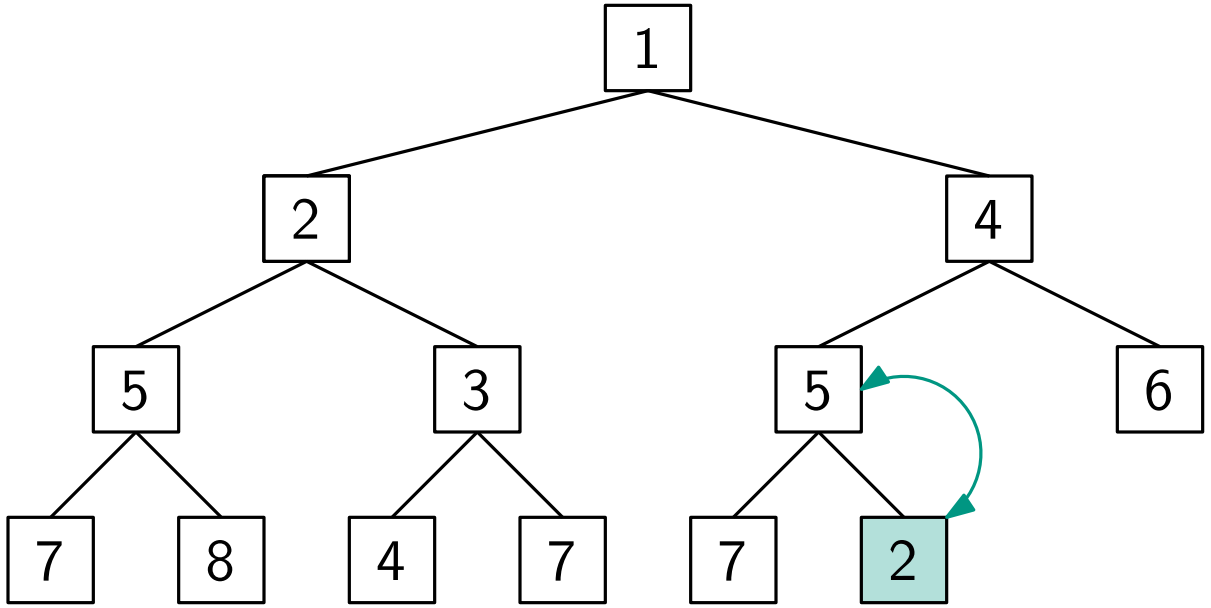
push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**



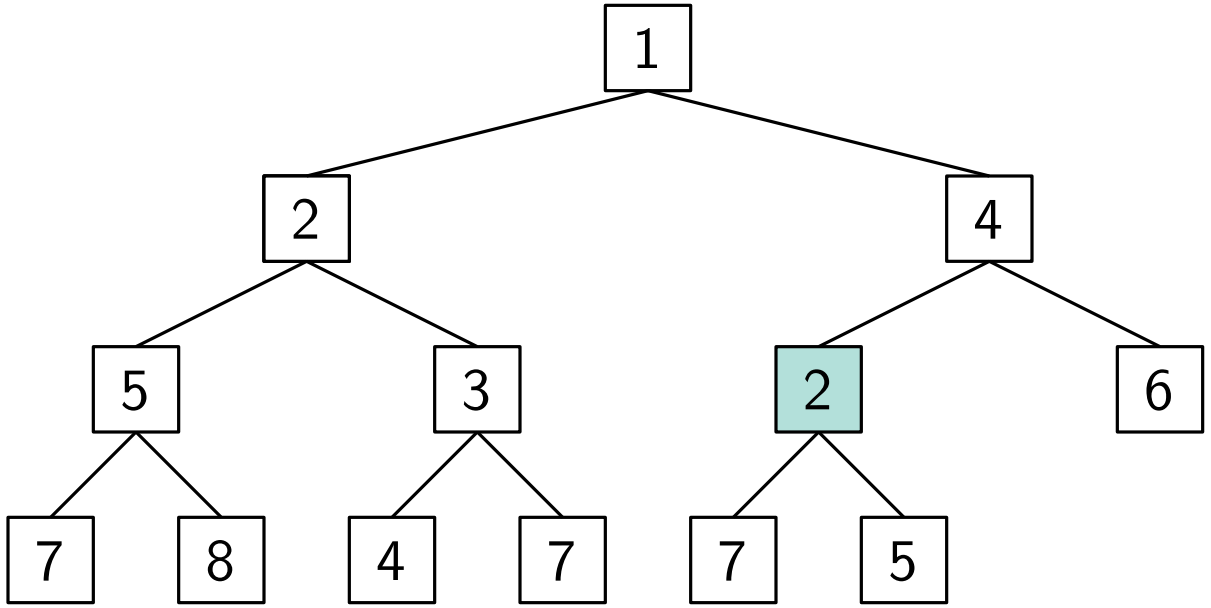
push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**



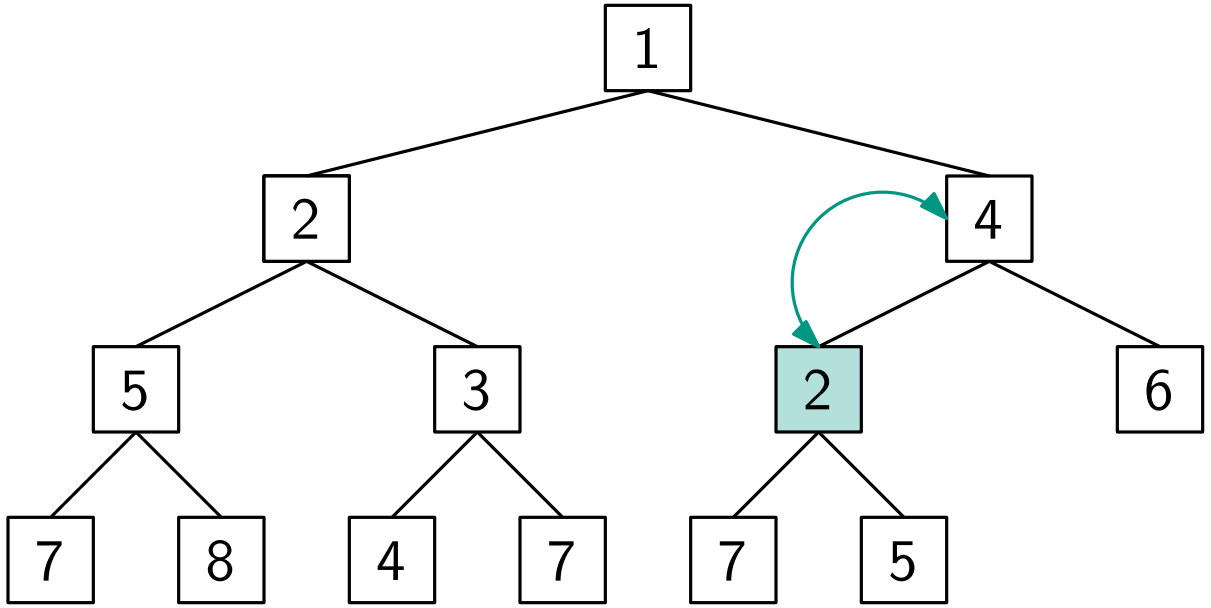
push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**



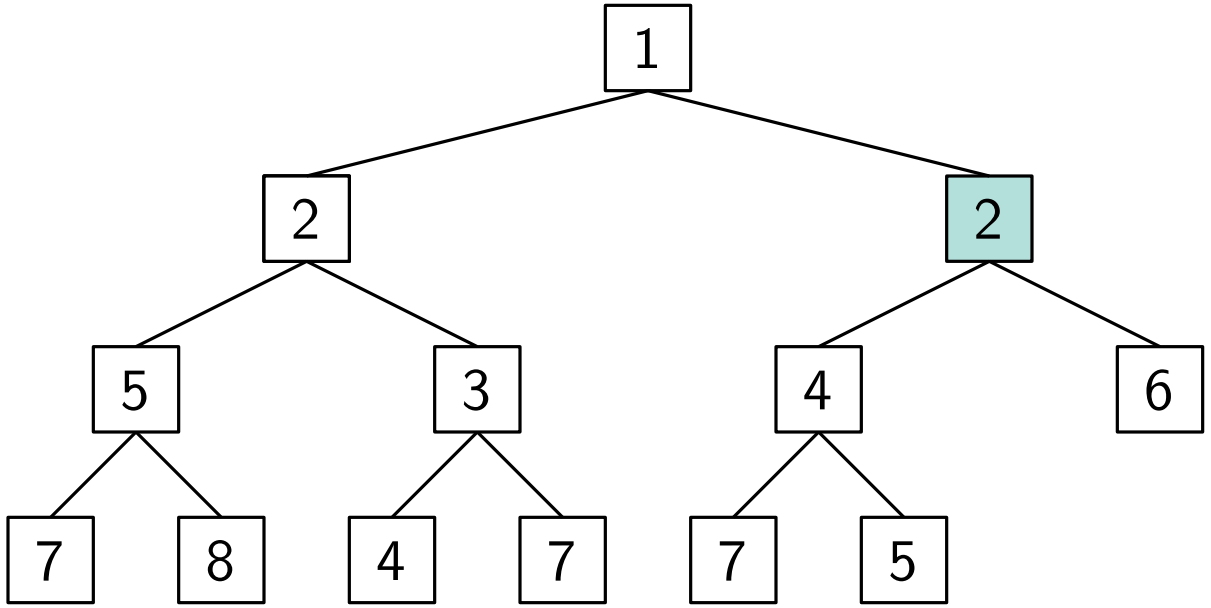
push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**



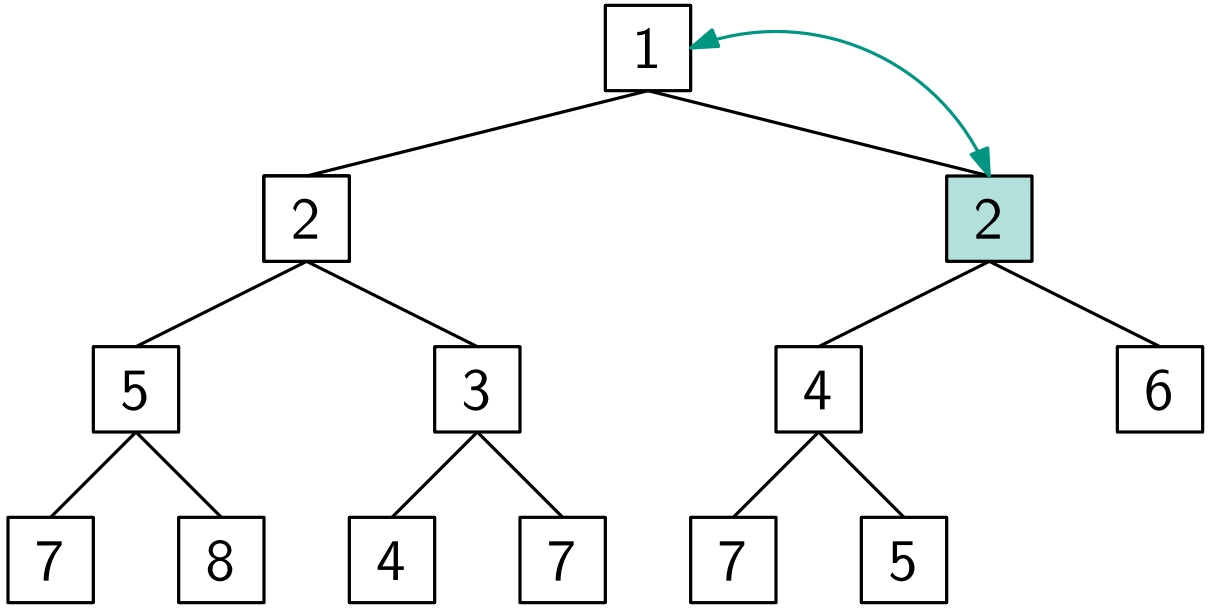
push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**



push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**

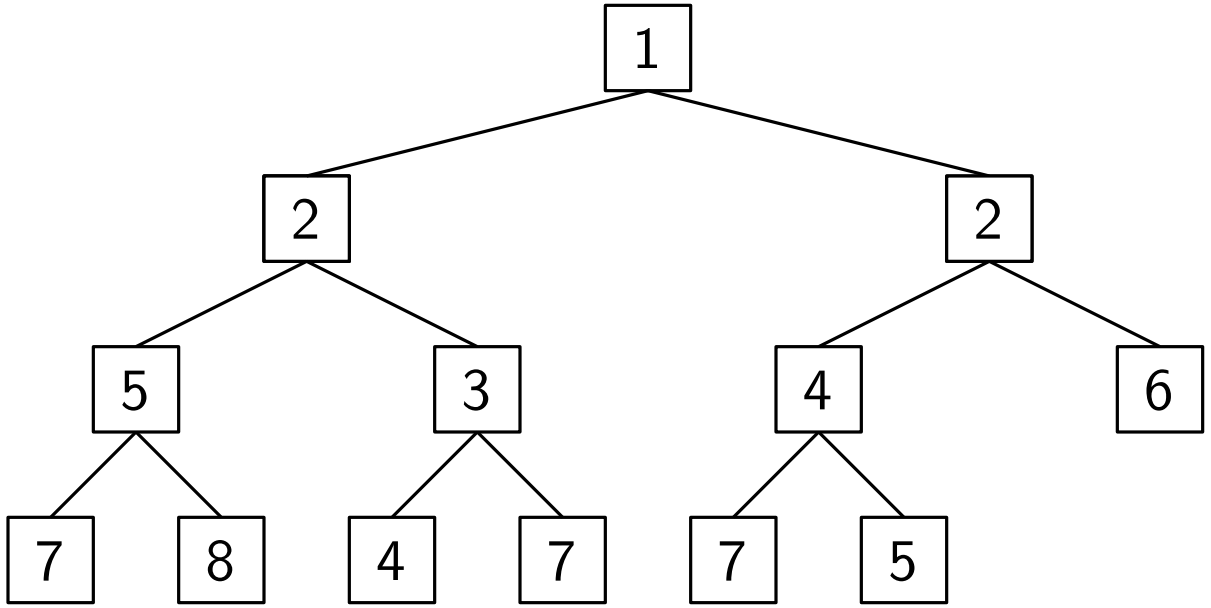


push

- Füge Element als Blatt rechts unten ein
- Schieb es an die richtige Stelle
- Dafür Hilfsprozedur **bubbleUp**

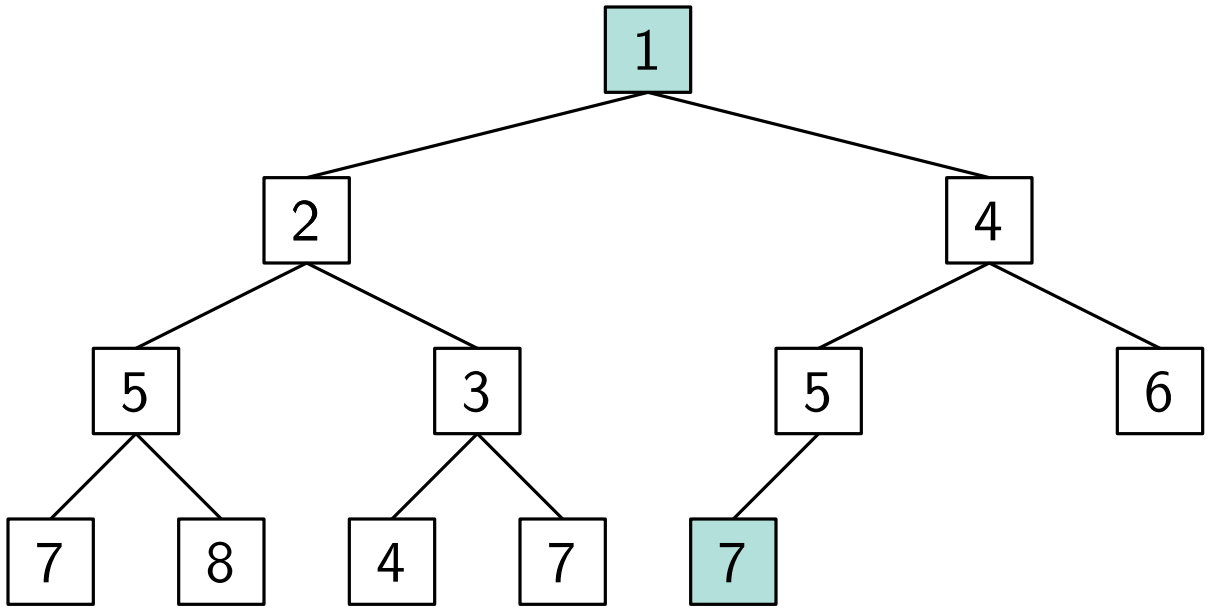
Laufzeit:

$\mathcal{O}(\log n)$



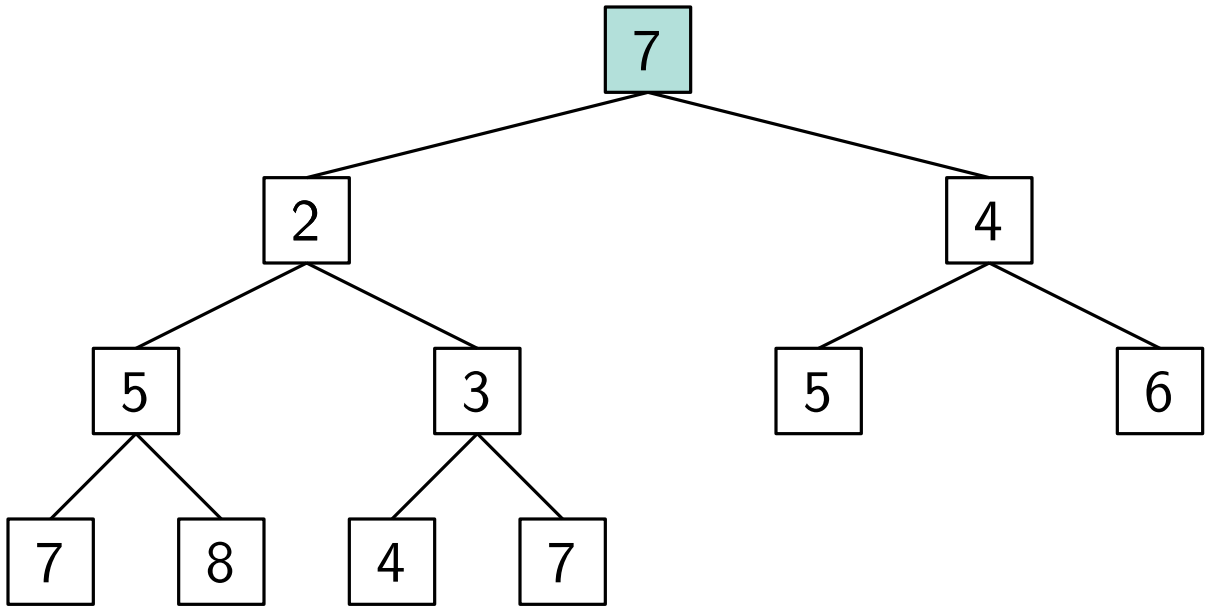
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



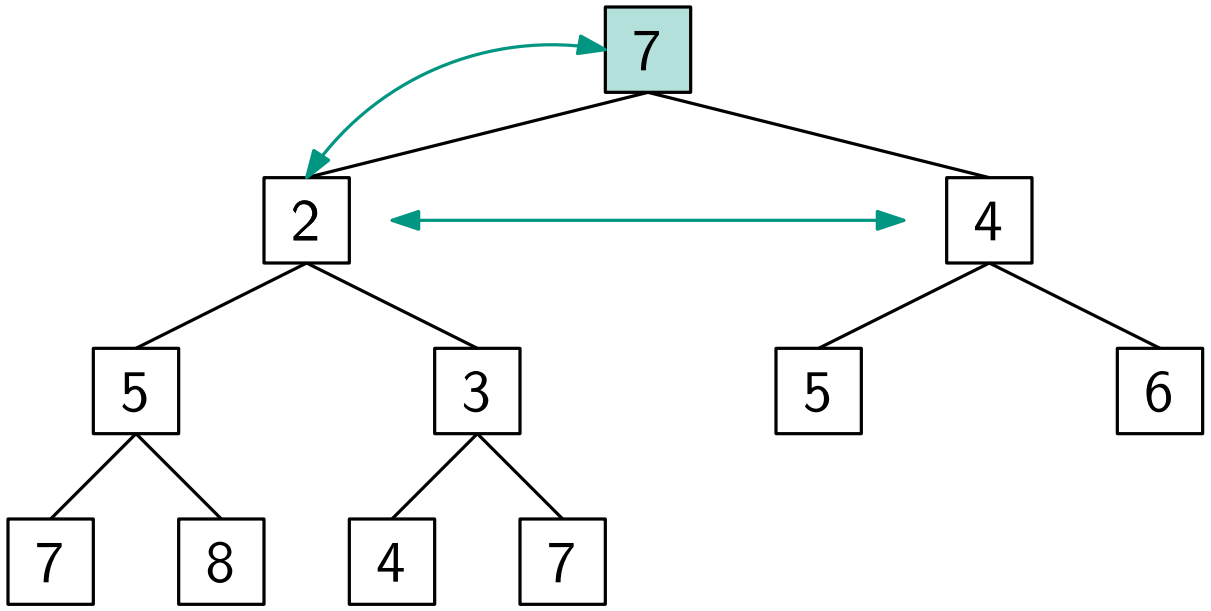
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



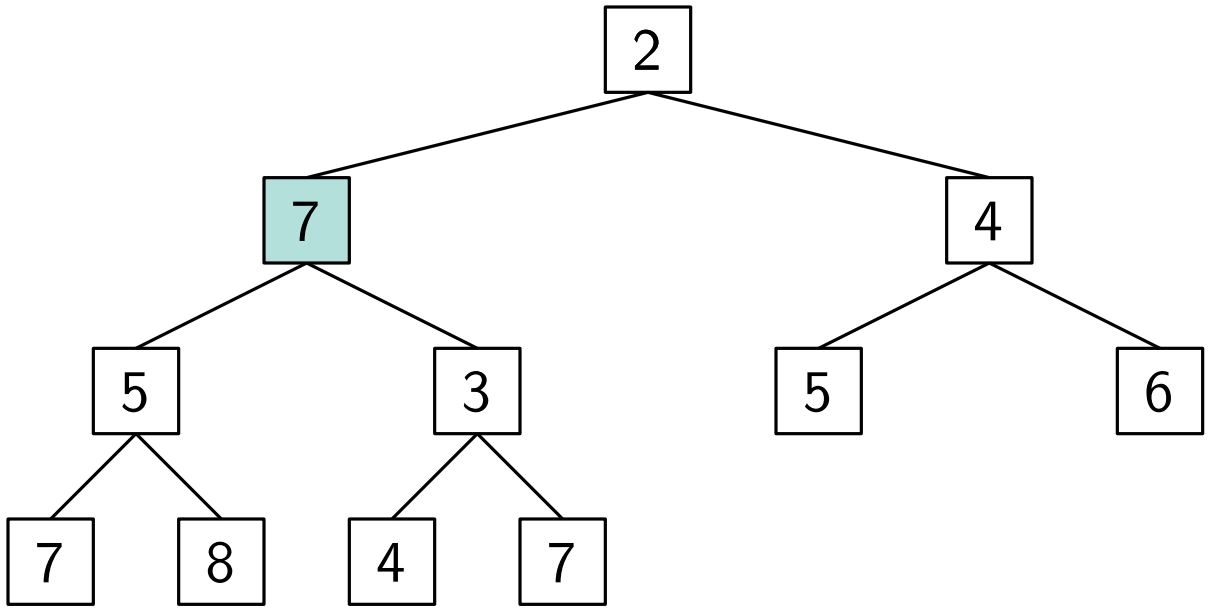
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



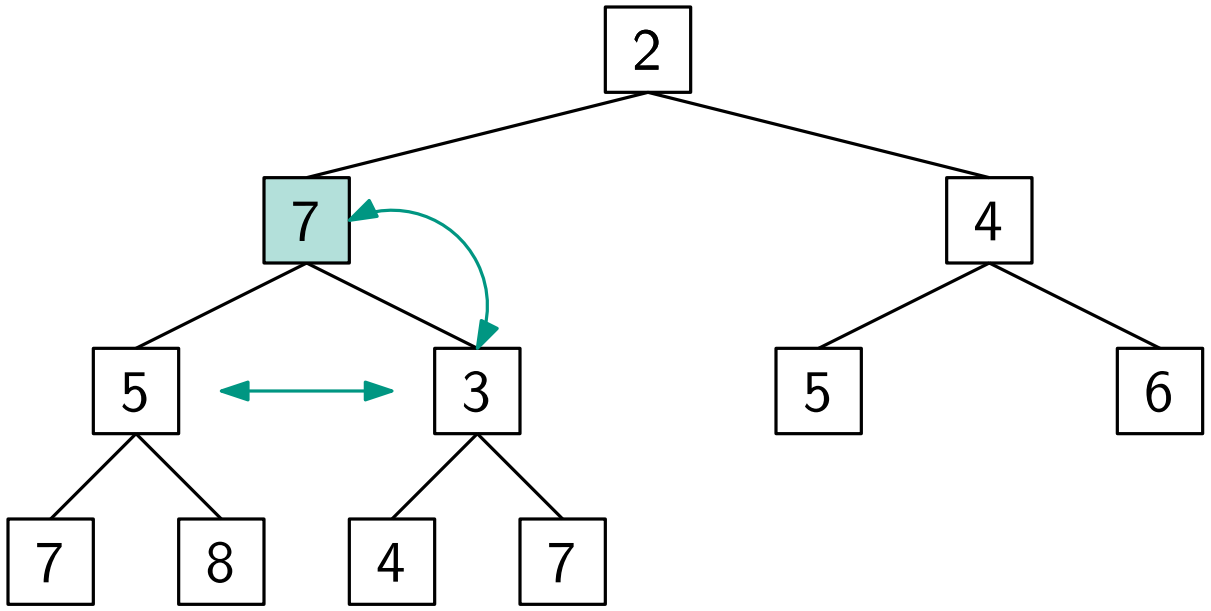
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



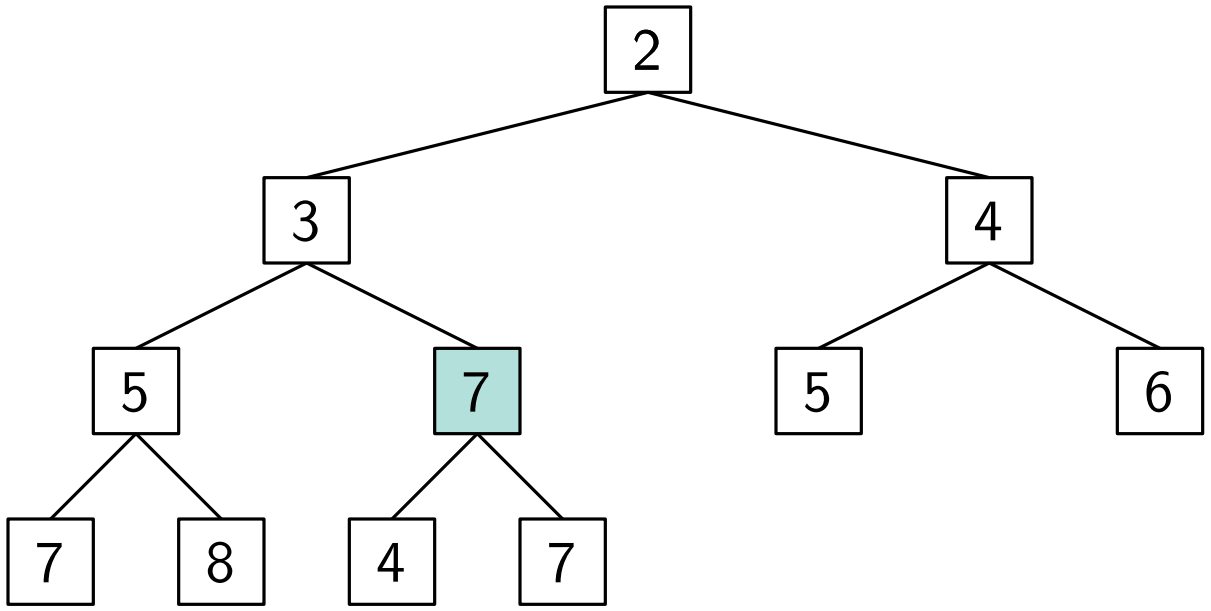
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



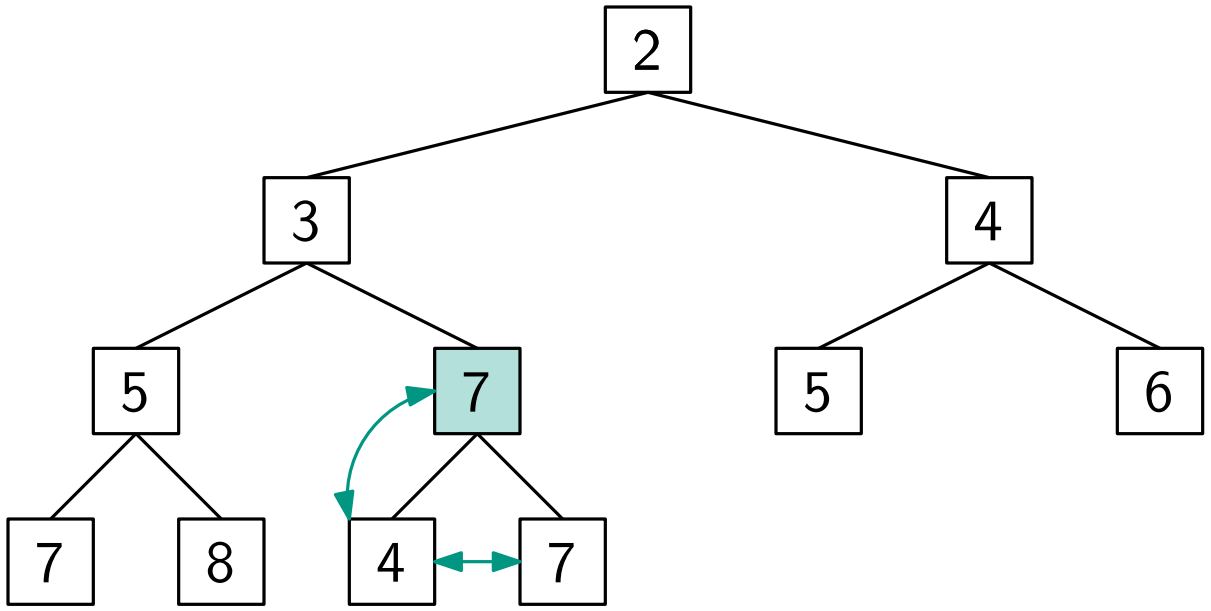
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



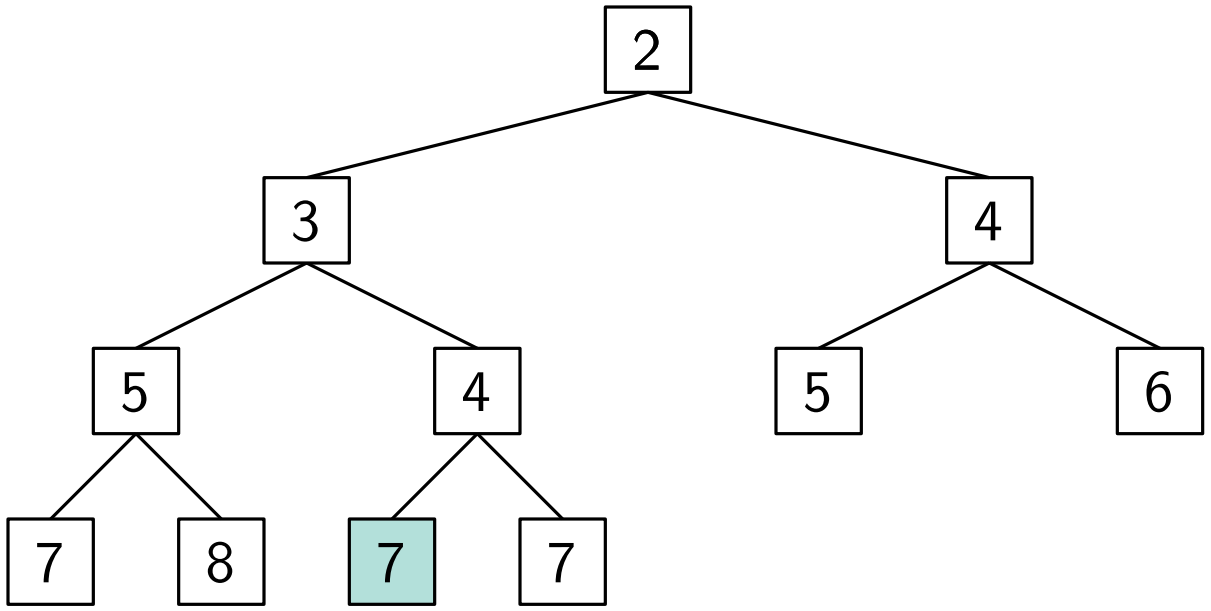
popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**



popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**

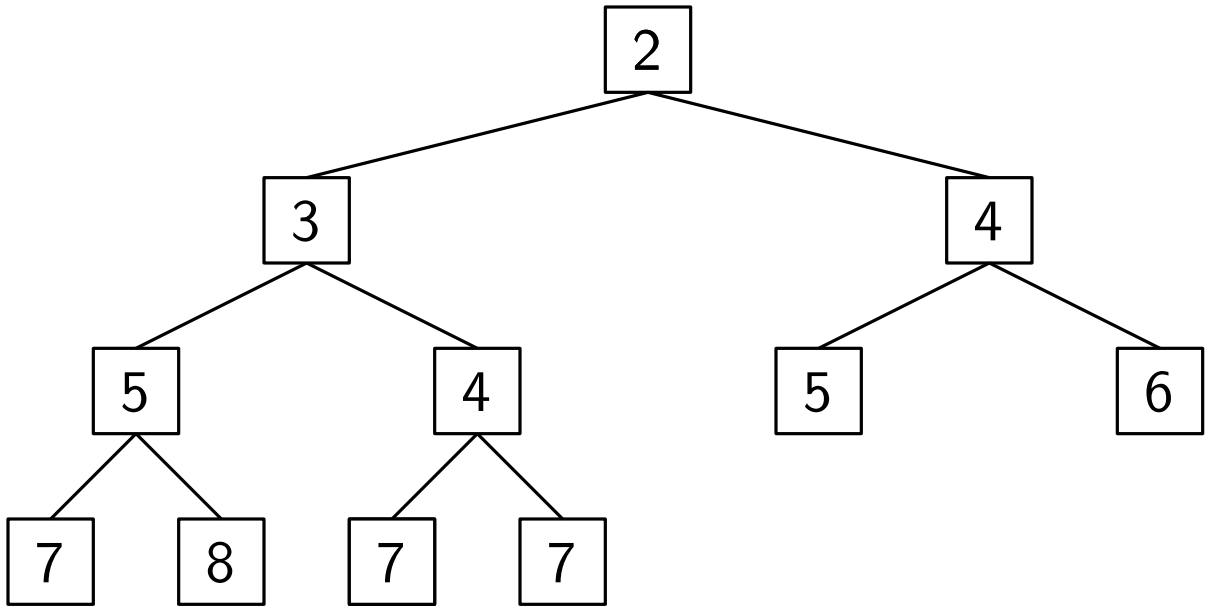


popMin

- Entfernt Wurzel und gibt sie aus
- Kopiert Element rechts unten an die Wurzel
- Räumt danach den Heap wieder auf
  - Benutzt dazu Hilfsprozedur **sinkDown**

Laufzeit:

$\mathcal{O}(\log n)$

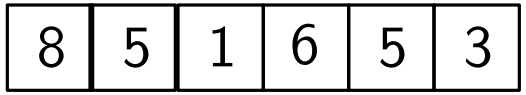


build

- Bringt das gespeicherte Array von Zahlen in Heap Form
  - Stellt Heap Eigenschaft her!
- Benutzt **sinkDown**

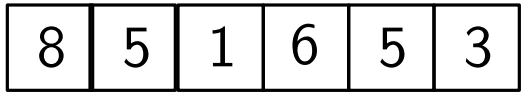
build

- Bringt das gespeicherte Array von Zahlen in Heap Form
  - Stellt Heap Eigenschaft her!
- Benutzt **sinkDown**



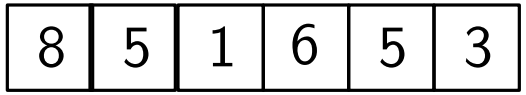
build

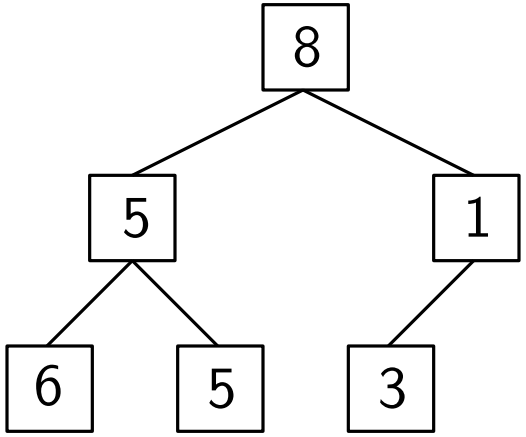
- Erstelle Heap aus Array



build

- Erstelle Heap aus Array

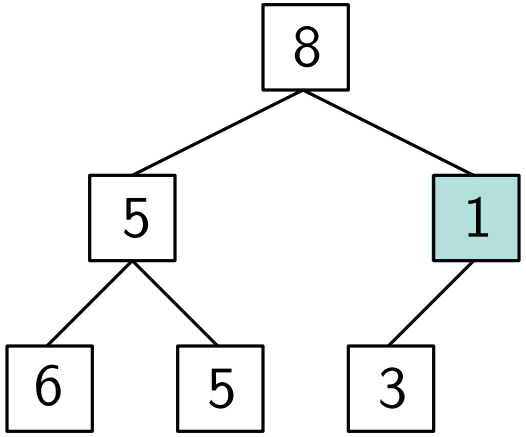




build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel

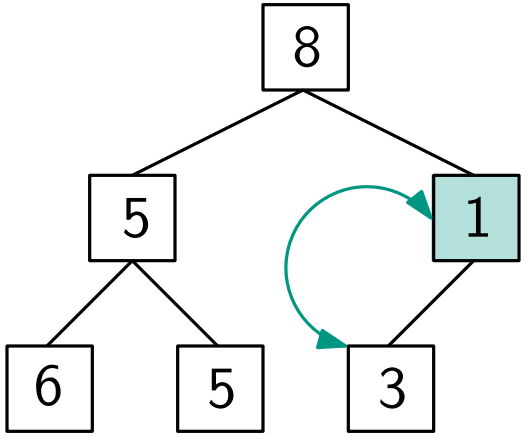
8	5	1	6	5	3



build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel
- Wende **sinkDown** an

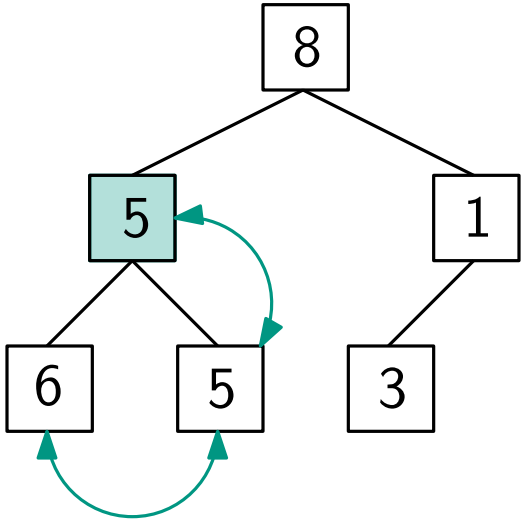
8	5	1	6	5	3



build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel
- Wende **sinkDown** an

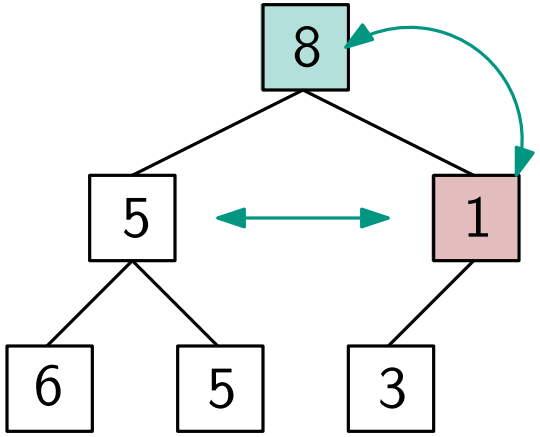
8	5	1	6	5	3



build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel
- Wende **sinkDown** an

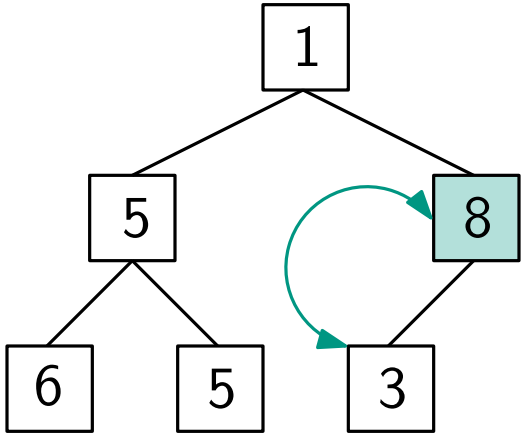
8	5	1	6	5	3



build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel
- Wende **sinkDown** an

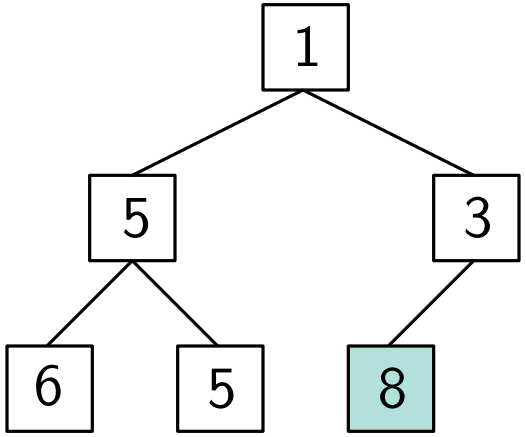
8	5	1	6	5	3



build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel
- Wende **sinkDown** an

8	5	1	6	5	3

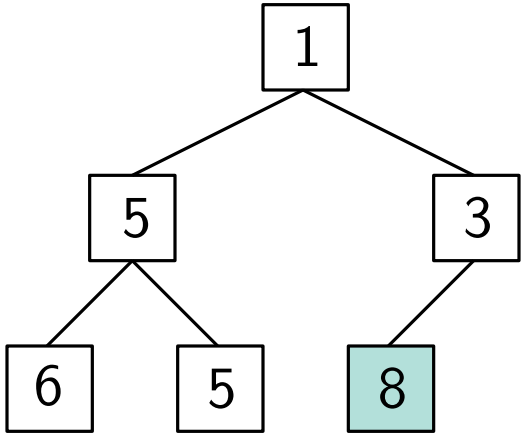


build

- Erstelle Heap aus Array
- Starte bei letztem Knoten der vorletzten Ebene
  - Bis zur Wurzel
- Wende **sinkDown** an

8	5	1	6	5	3

Laufzeit:
- in $\mathcal{O}(n)$

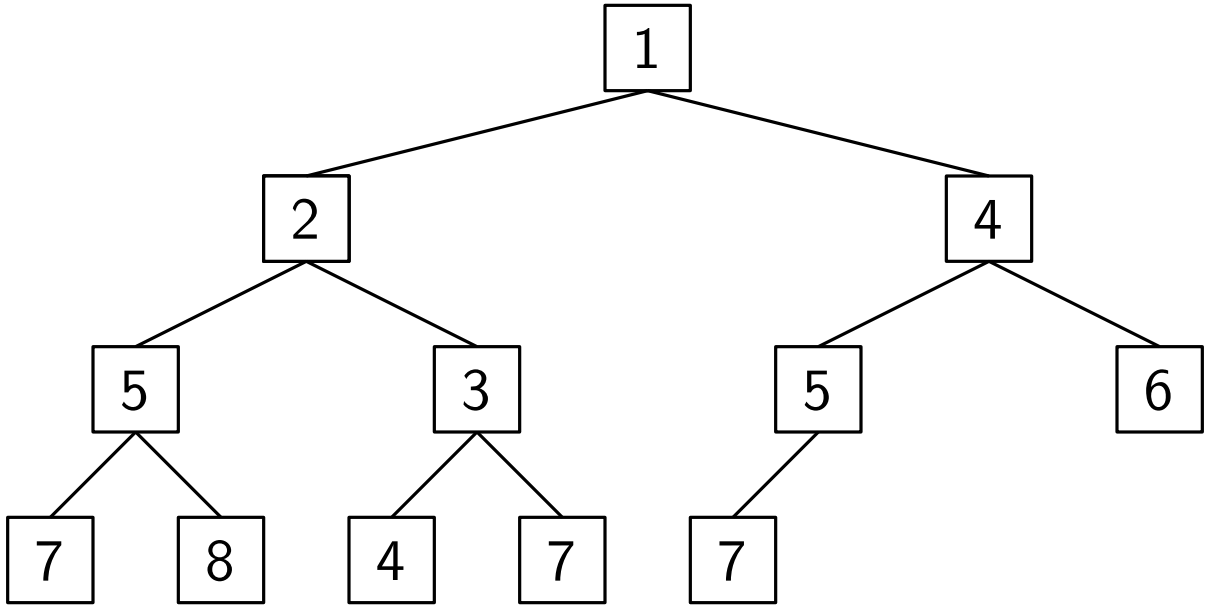


decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation

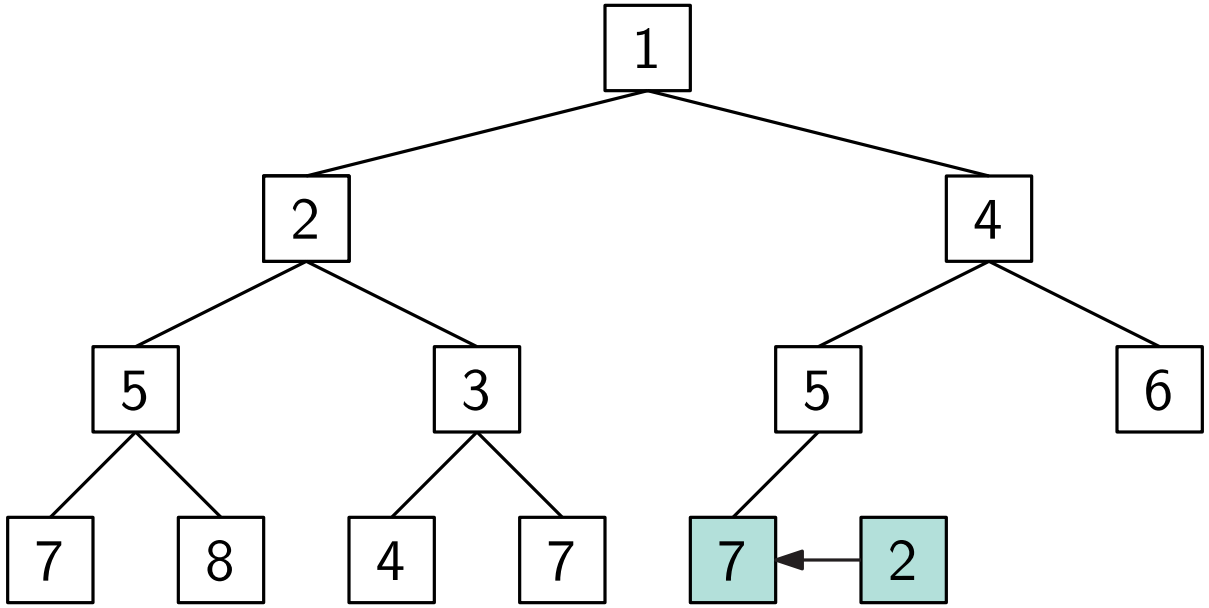
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



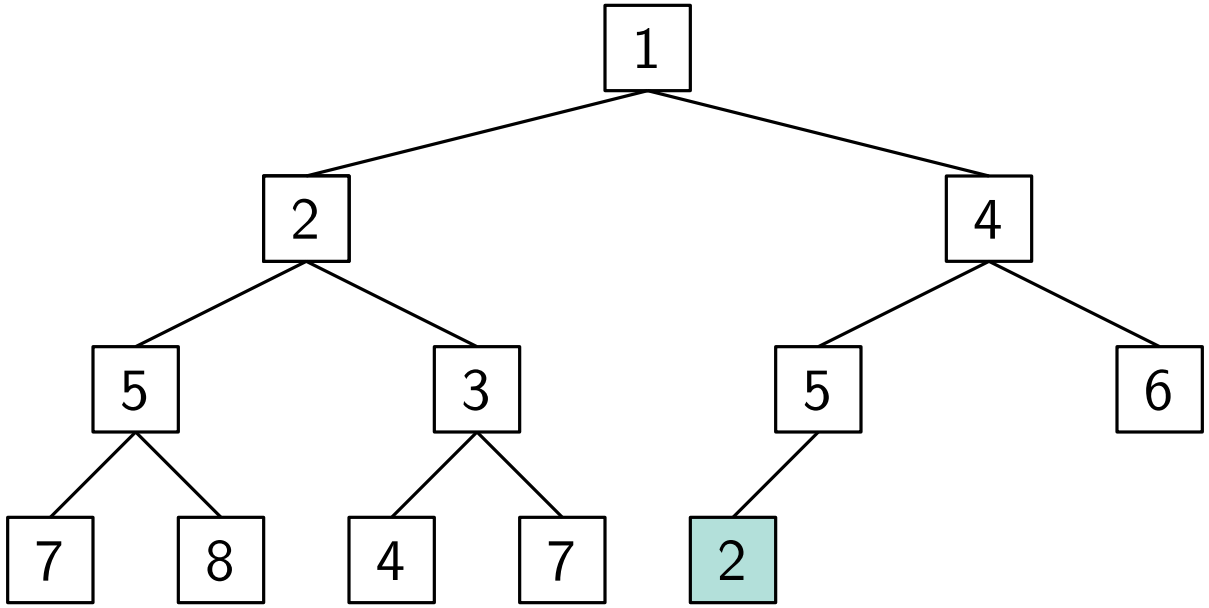
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



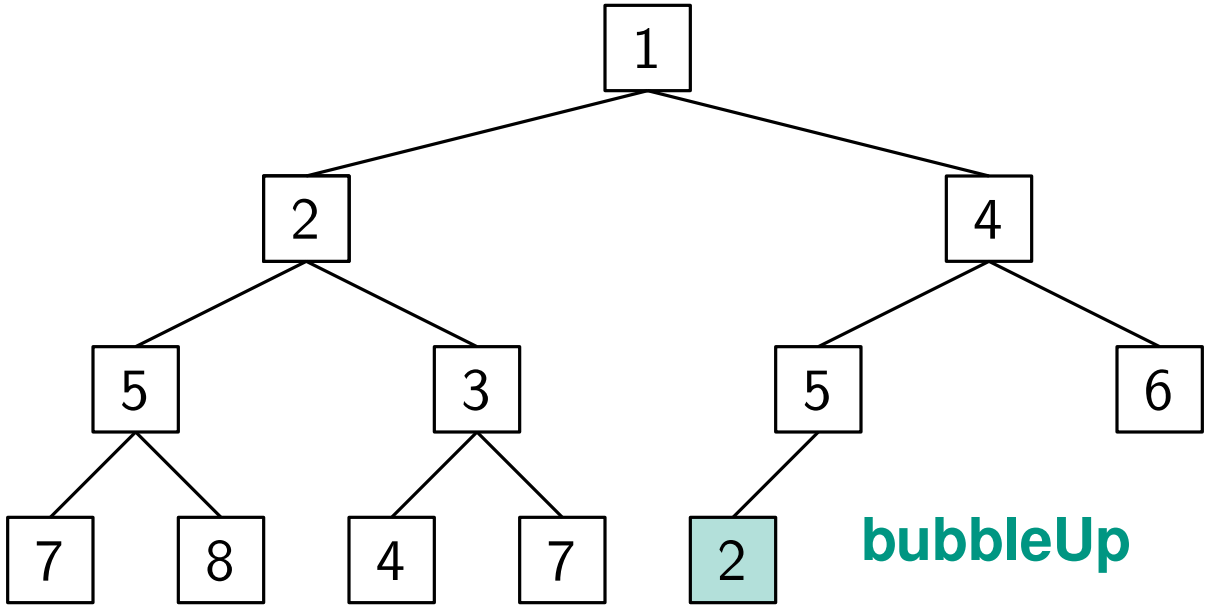
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



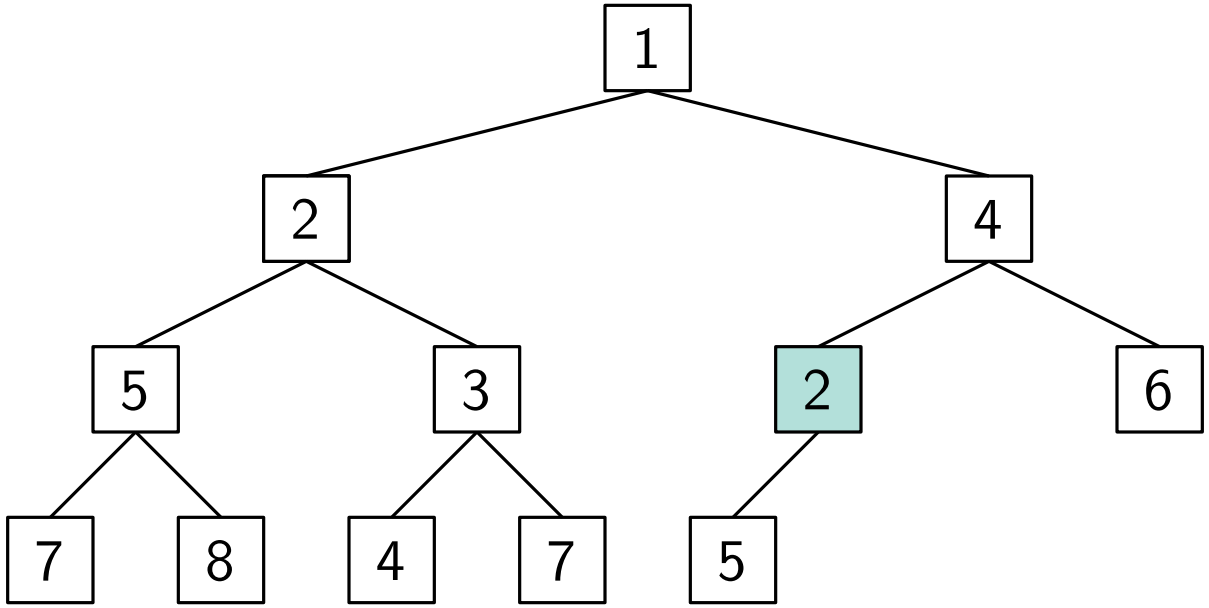
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



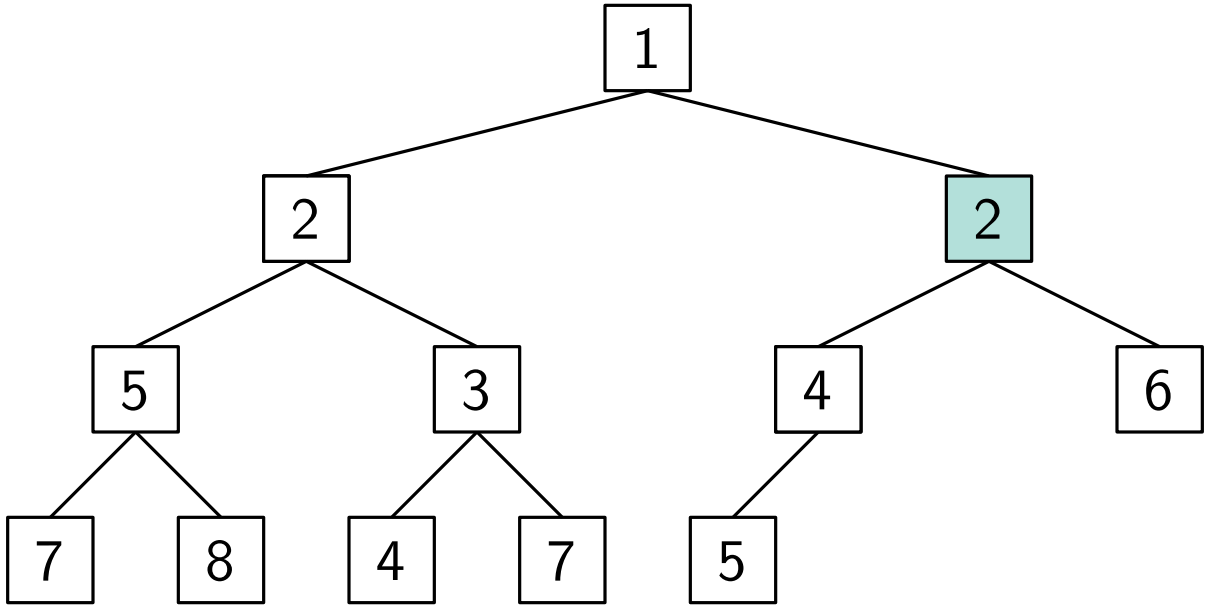
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



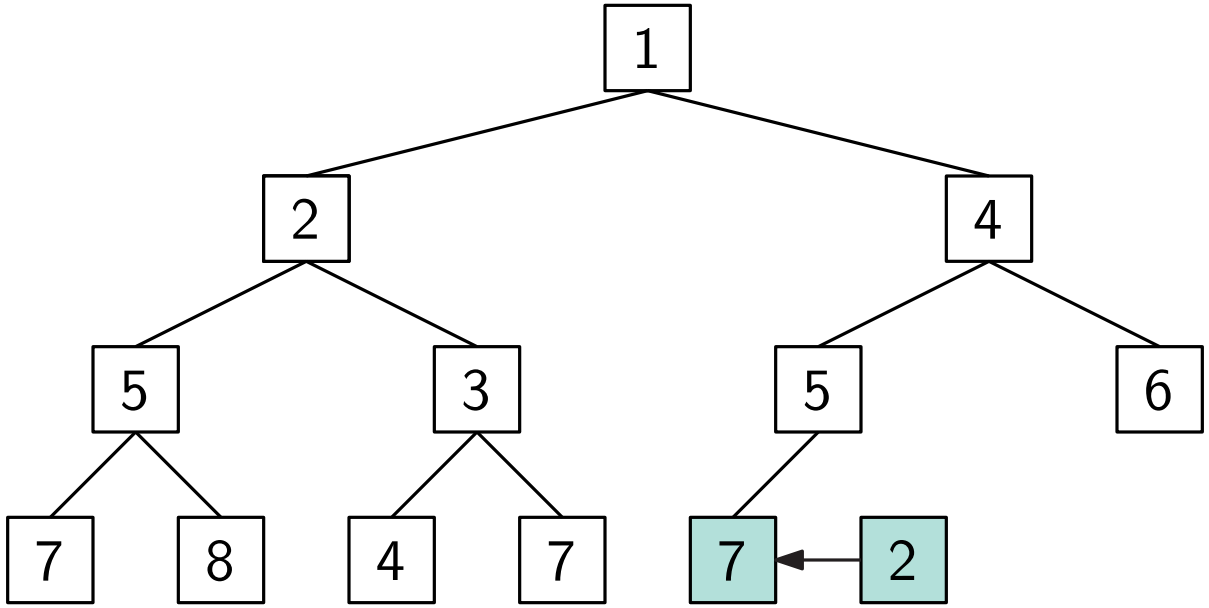
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



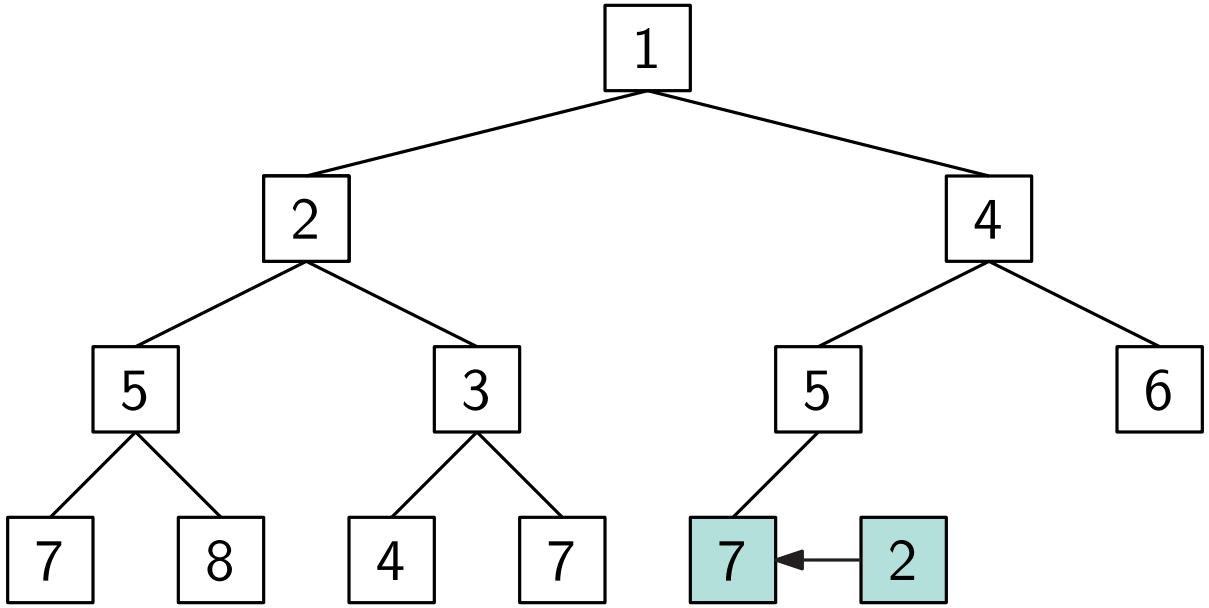
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



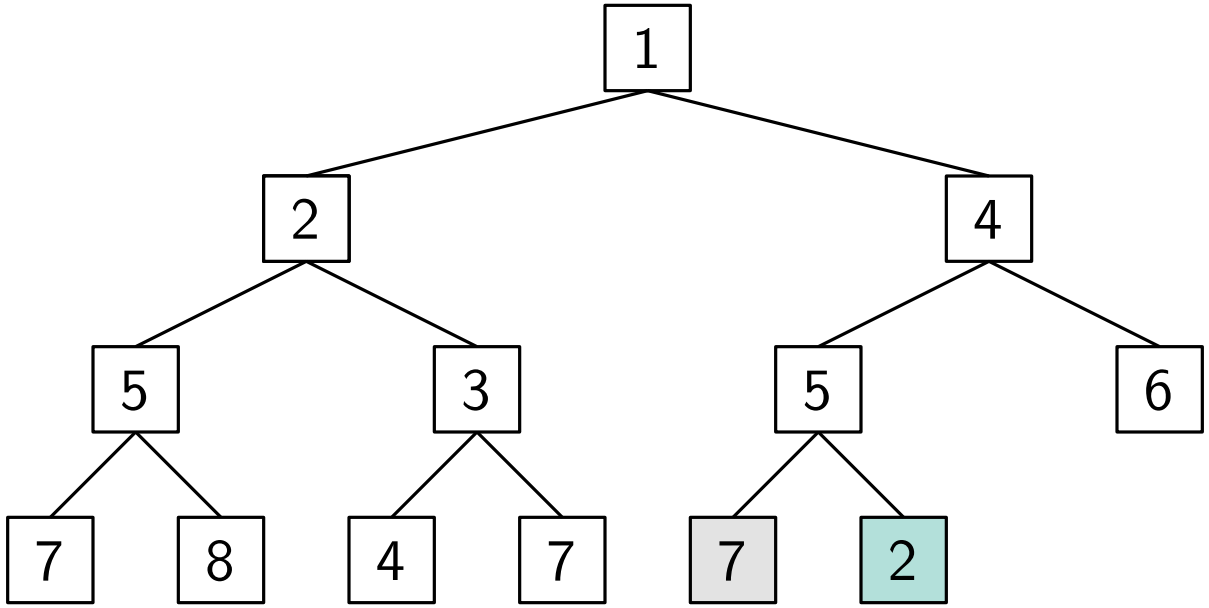
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



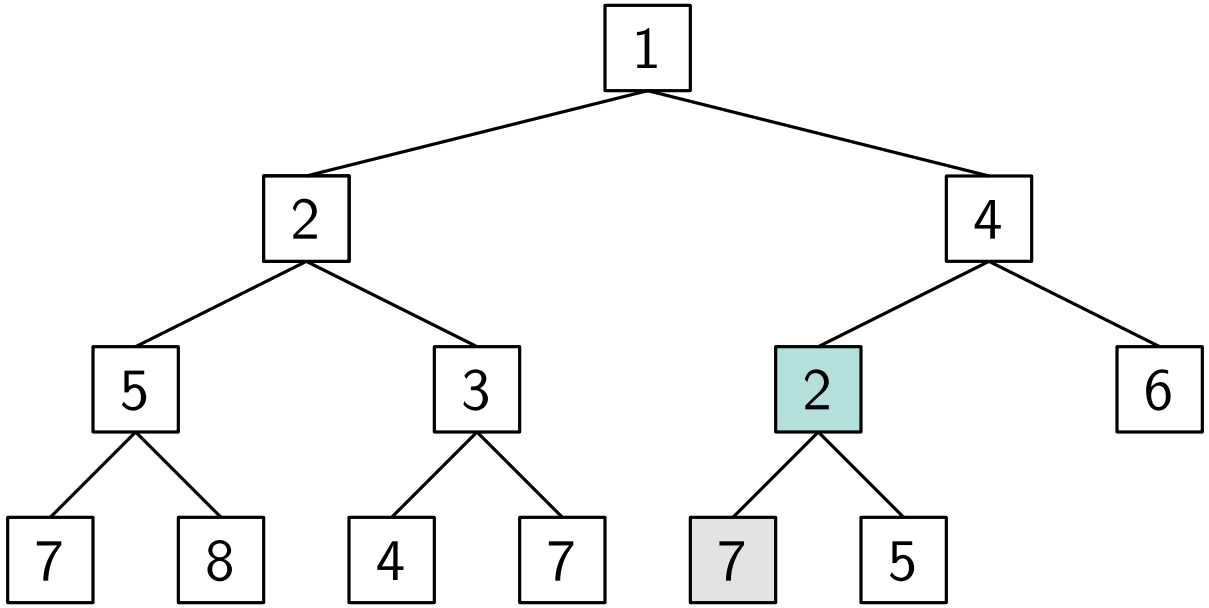
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



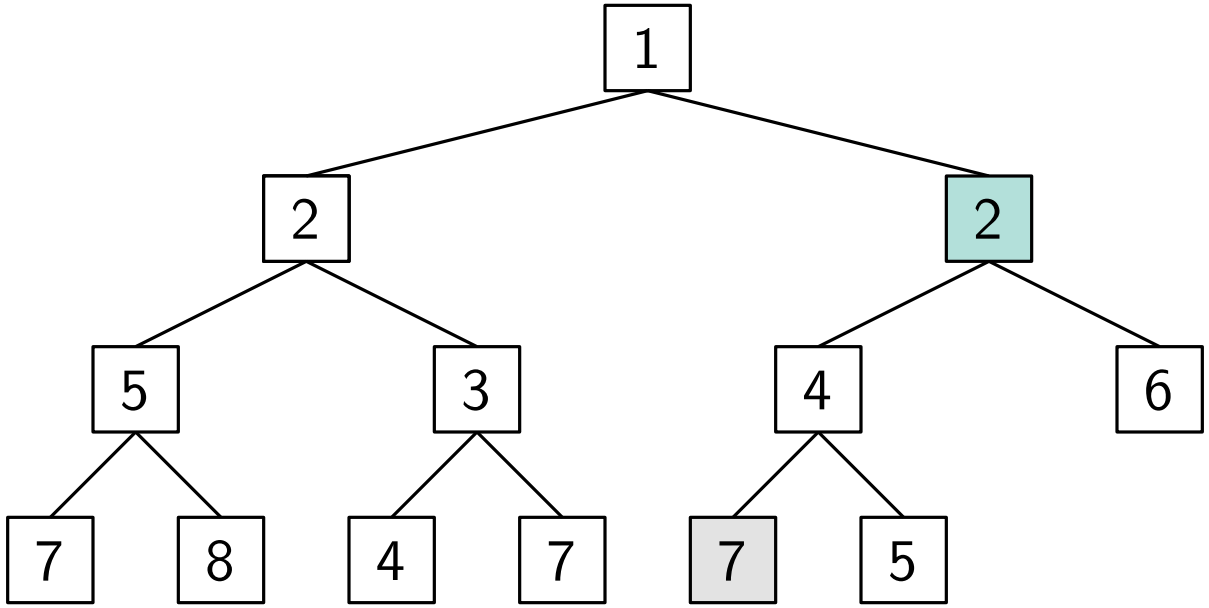
decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



decPrio

- mehrere Möglichkeiten
  - Element updaten und Heap-Eigenschaft wiederherstellen
  - Element löschen und mit neuer Priorität wieder einfügen
  - Lazy Evaluation



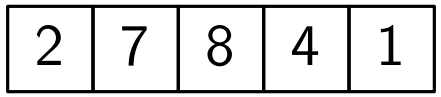
Heap Sort

- Sortiert absteigend

Heap Sort

- Sortiert absteigend

Funktionsweise

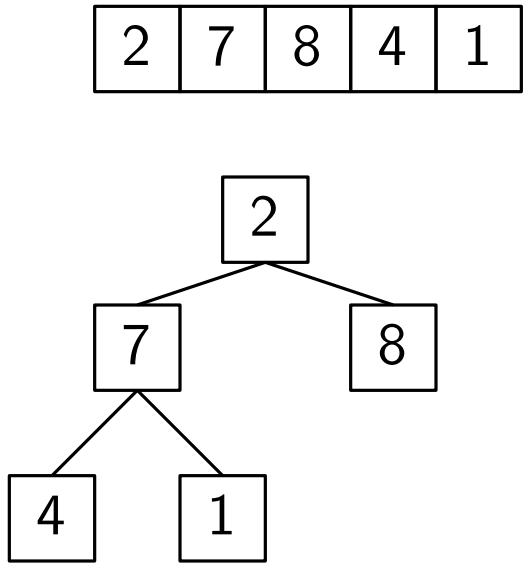


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap

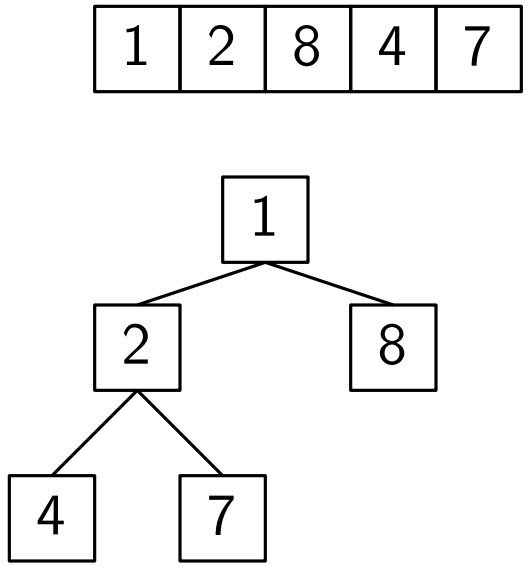


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap

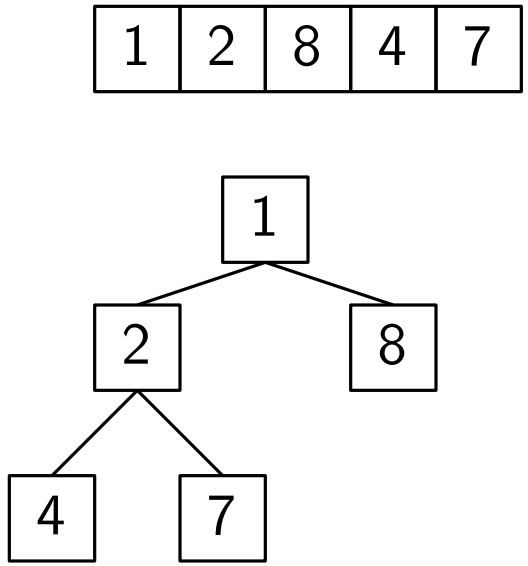


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays

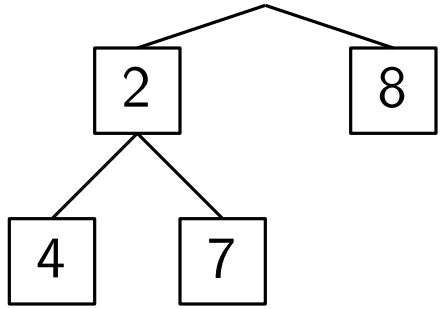


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays

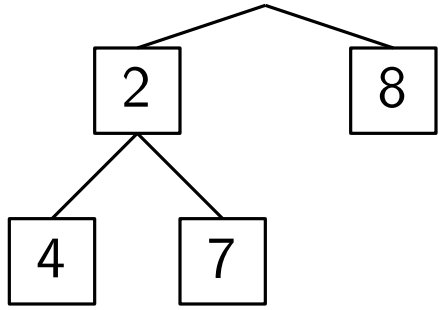


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her

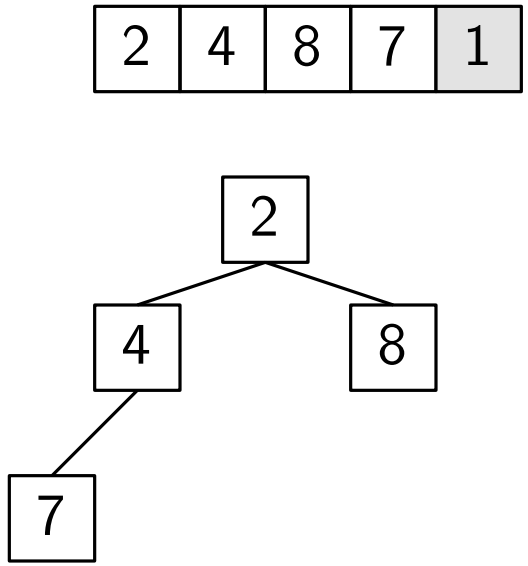


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her

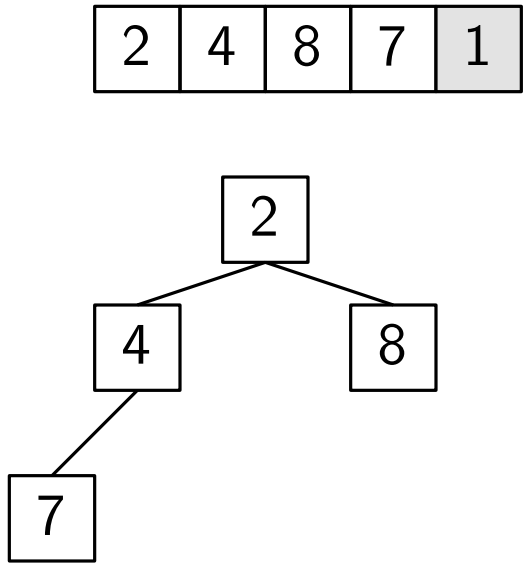


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her
  - Wiederhole bis nur noch eine Zahl in Heap

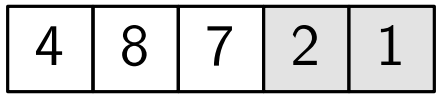


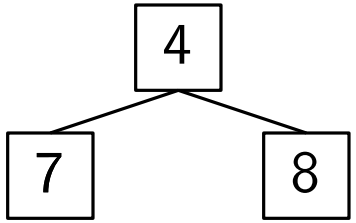
Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her
  - Wiederhole bis nur noch eine Zahl in Heap



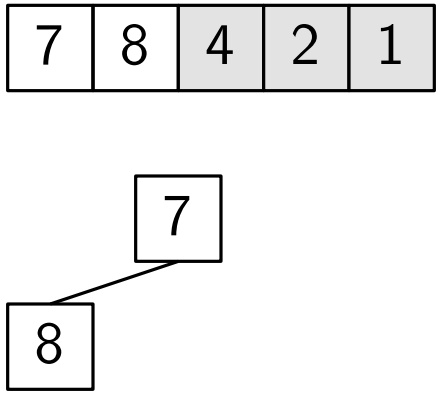


Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her
  - Wiederhole bis nur noch eine Zahl in Heap



Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her
  - Wiederhole bis nur noch eine Zahl in Heap

Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her
  - Wiederhole bis nur noch eine Zahl in Heap

Laufzeit

Heap Sort

- Sortiert absteigend

Funktionsweise

- **build:** Baut aus gegebenen Zahlen Heap
- **popMin:** Löscht Wurzel
  - speichert Element an letzter Stelle des Arrays
  - Stellt Heap-Eigenschaft wieder her
  - Wiederhole bis nur noch eine Zahl in Heap

Laufzeit

- $\mathcal{O}(n \log n)$
- nicht stabil

Zusammenfassung Heaps

werden als **Priority Queue** verwendet

Zusammenfassung Heaps

werden als **Priority Queue** verwendet

- **push(x):** Füge ein Objekt mit Priorität x ein
- **popMin():** Erhalte Objekt mit minimaler Priorität
- **decPrio(x,x'):** Verringere Priorität x zu x'
- **build(Array A):** Erstelle Priority Queue aus gegebenem Array

Zusammenfassung Heaps

werden als **Priority Queue** verwendet

- **push(x):** Füge ein Objekt mit Priorität x ein
- **popMin():** Erhalte Objekt mit minimaler Priorität
- **decPrio(x,x'):** Verringere Priorität x zu x'
- **build(Array A):** Erstelle Priority Queue aus gegebenem Array
  - Hilfsprozeduren zur Umsetzung: **bubbleUp()** und **sinkDown()**

Zusammenfassung Heaps

werden als **Priority Queue** verwendet

Laufzeiten in Binären Heaps:

- **push(x):** Füge ein Objekt mit Priorität x ein — $\mathcal{O}(\log n)$
- **popMin():** Erhalte Objekt mit minimaler Priorität — $\mathcal{O}(\log n)$
- **decPrio(x,x'):** Verringere Priorität x zu x' — $\mathcal{O}(\log n)$
- **build(Array A):** Erstelle Priority Queue aus gegebenem Array — $\mathcal{O}(n)$
  - Hilfsprozeduren zur Umsetzung: **bubbleUp()** und **sinkDown()**

Zusammenfassung Heaps

werden als **Priority Queue** verwendet

Laufzeiten in Binären Heaps:

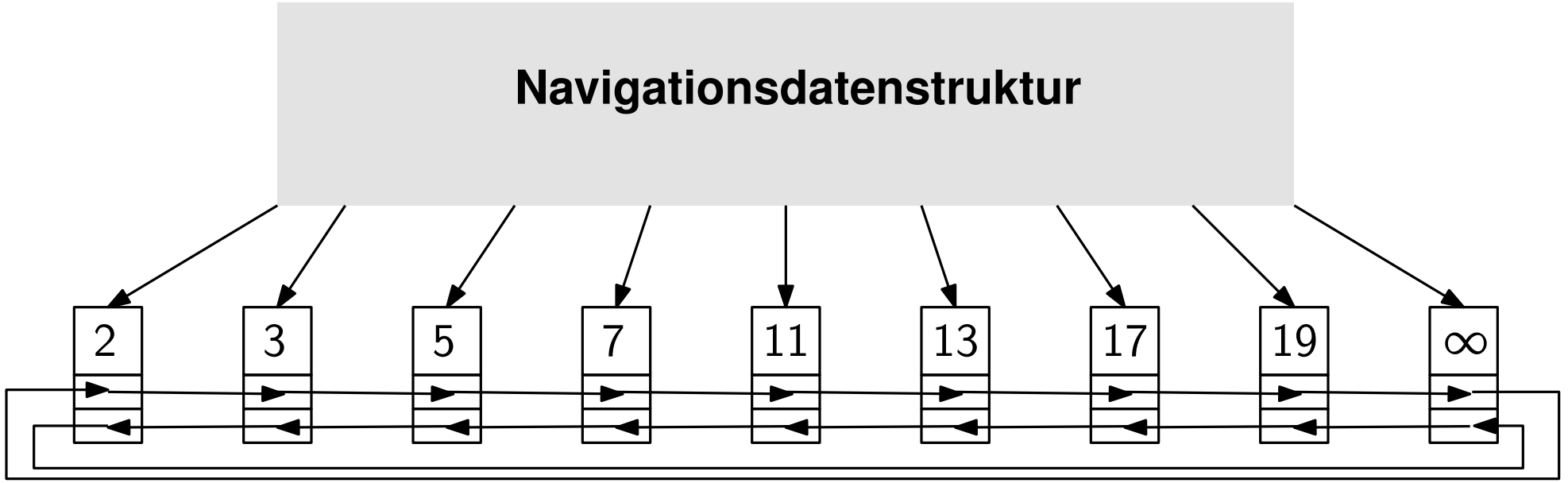
- **push(x):** Füge ein Objekt mit Priorität x ein — $\mathcal{O}(\log n)$
- **popMin():** Erhalte Objekt mit minimaler Priorität — $\mathcal{O}(\log n)$
- **decPrio(x,x'):** Verringere Priorität x zu x' — $\mathcal{O}(\log n)$
- **build(Array A):** Erstelle Priority Queue aus gegebenem Array — $\mathcal{O}(n)$
  - Hilfsprozeduren zur Umsetzung: **bubbleUp()** und **sinkDown()**

Alternative Heaps

- Binomial-Heaps
  - push(x) amortisiert in $\mathcal{O}(1)$
- Fibonacci-Heaps
  - decPrio(x,x') zusätzlich amortisiert in $\mathcal{O}(1)$

Grundlagen Sortierte Folgen

- Folge $< e_1, \ldots, e_n >$ mit $e_1 \leq \cdots \leq e_n$
  - mit aufgesetzter Navigationsdatenstruktur M
  - oftmals mit Dummyelement ∞
    - zum Aufrechterhalten von Invarianten



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

binärer Suchbaum

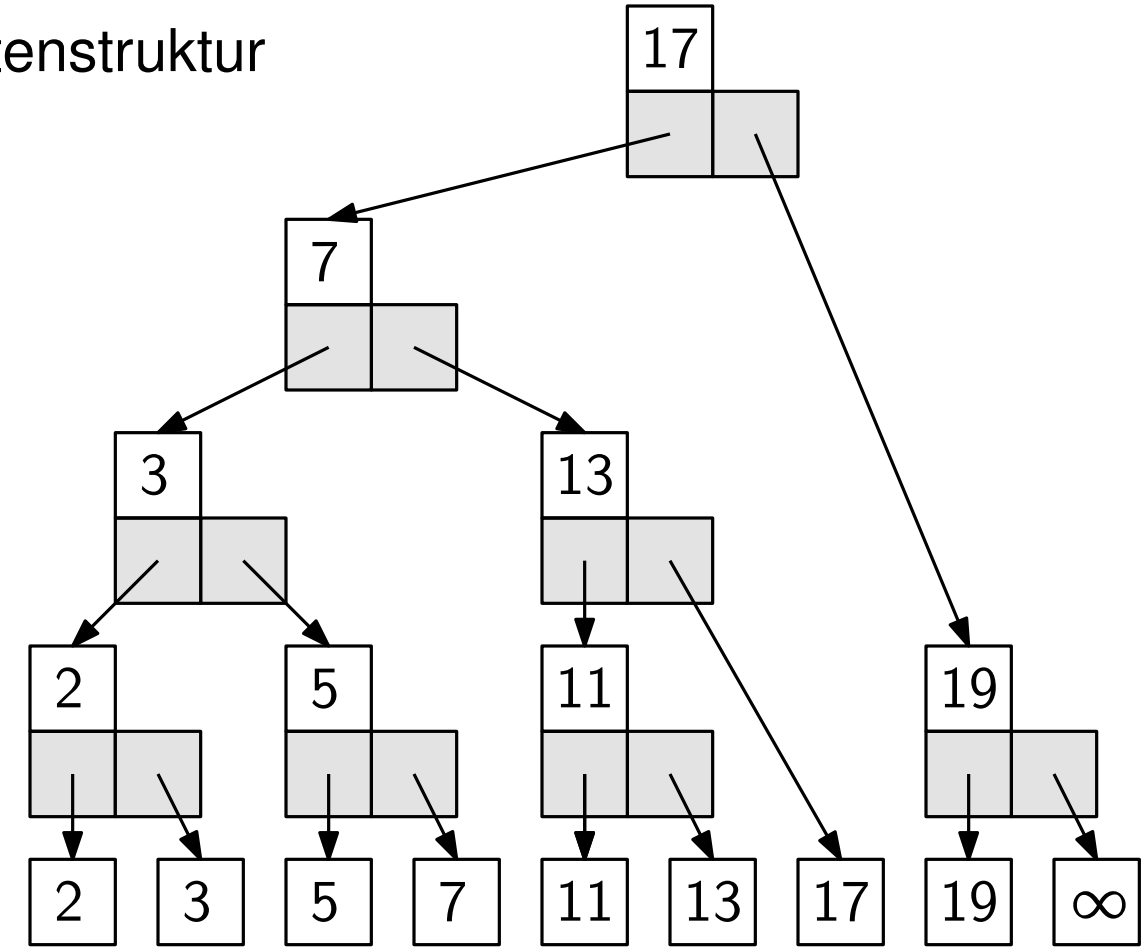
Idee: Verwende Baum als Navigationsdatenstruktur

- **Blätter:** Elemente der Folge

binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

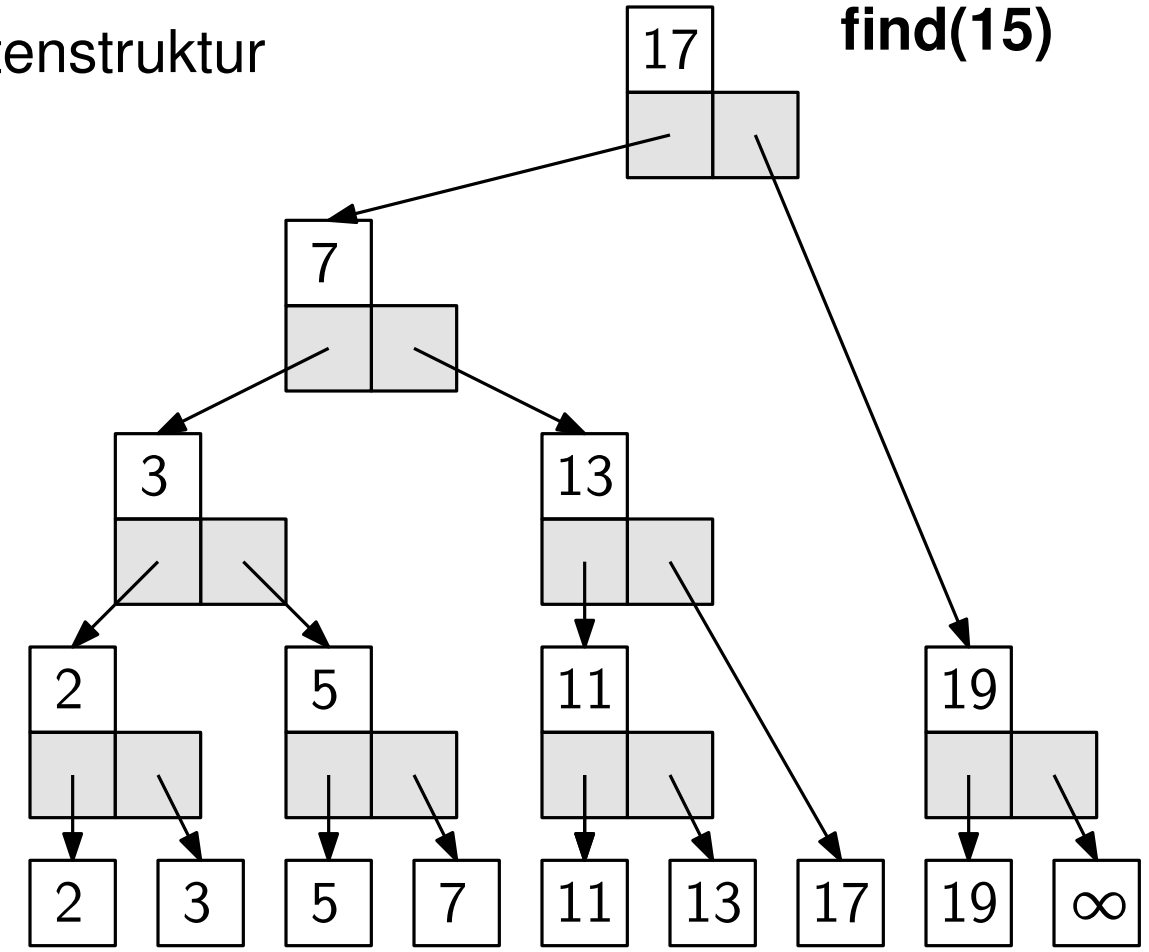
- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

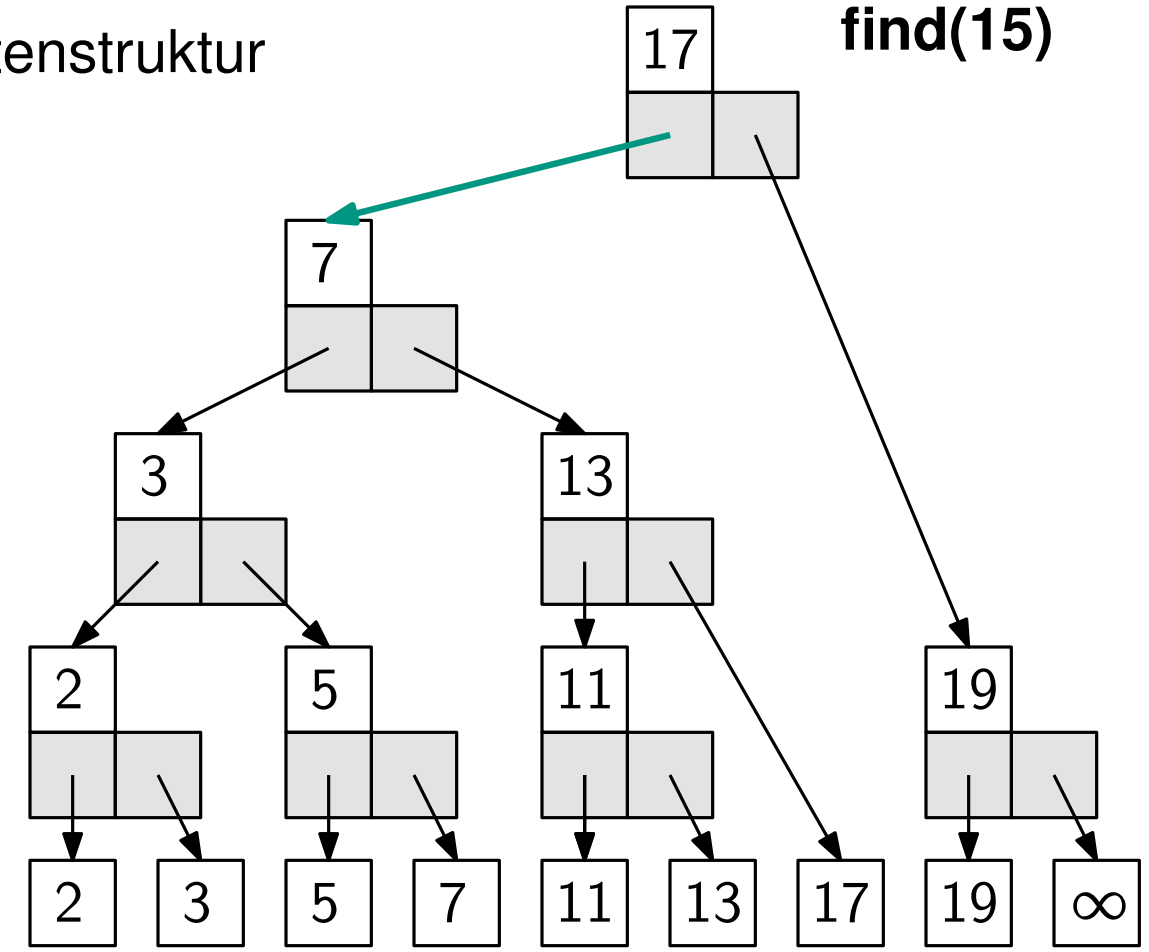
- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$
- **find:** Element (oder nächstgrößeres)
  - steige dazu in richtigenTeilbaum ab



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

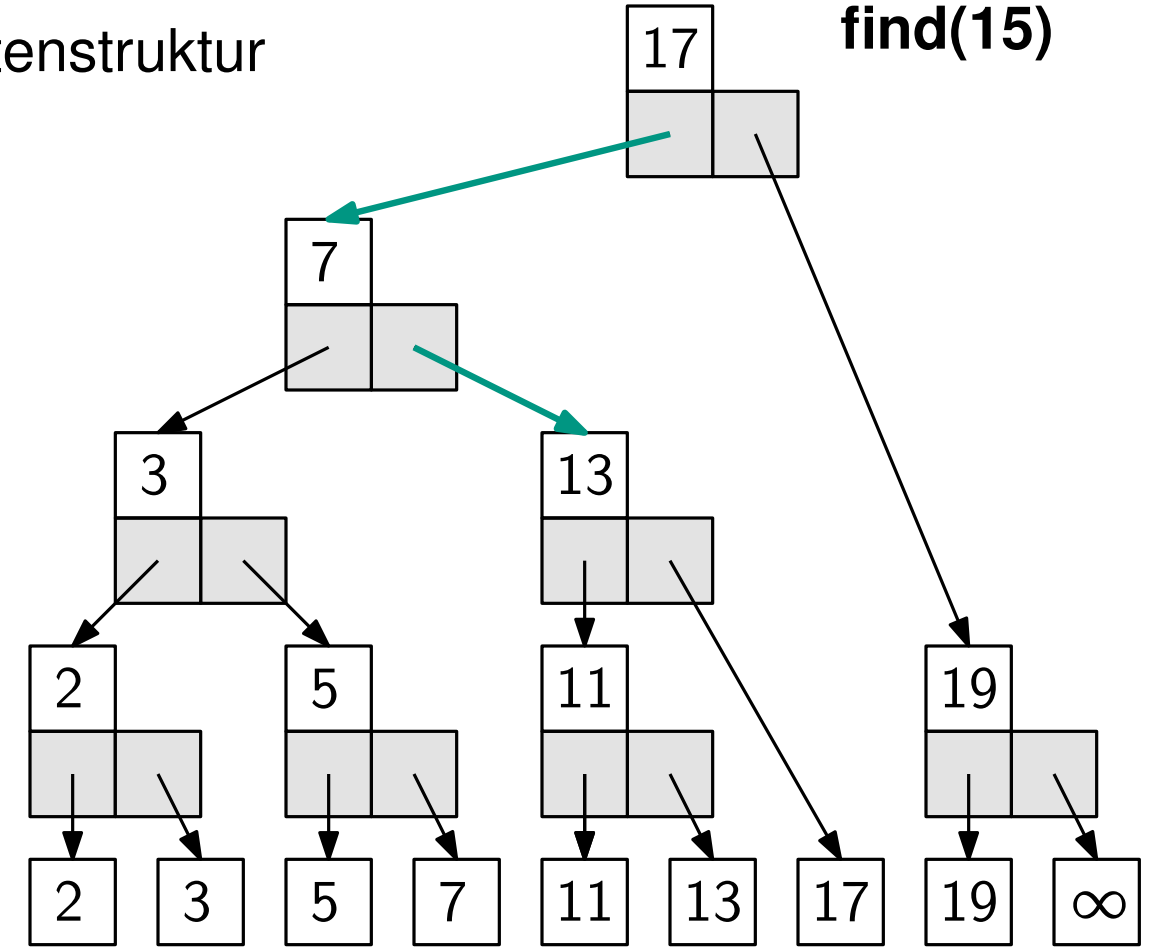
- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$
- **find:** Element (oder nächstgrößeres)
  - steige dazu in richtigenTeilbaum ab



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

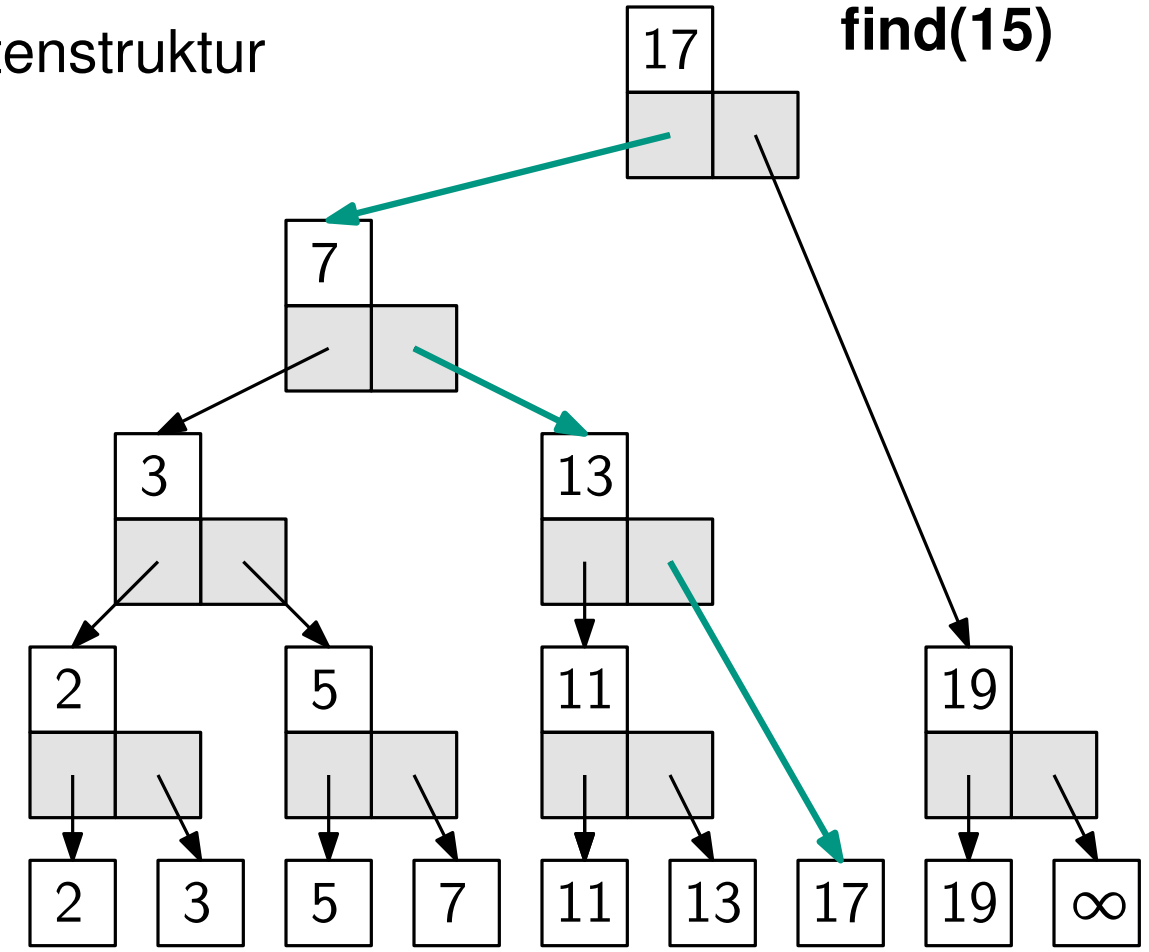
- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$
- **find:** Element (oder nächstgrößeres)
  - steige dazu in richtigenTeilbaum ab



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

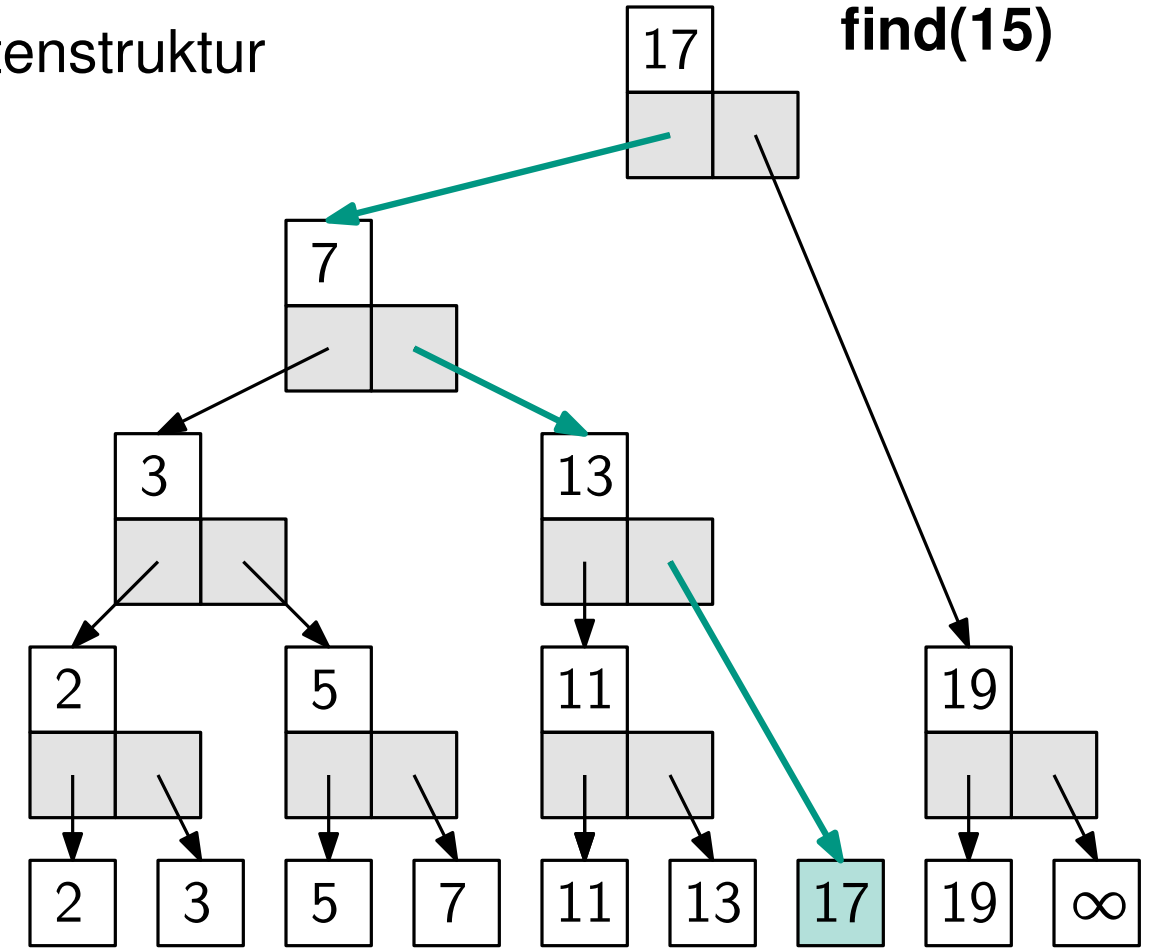
- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$
- **find:** Element (oder nächstgrößeres)
  - steige dazu in richtigenTeilbaum ab



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

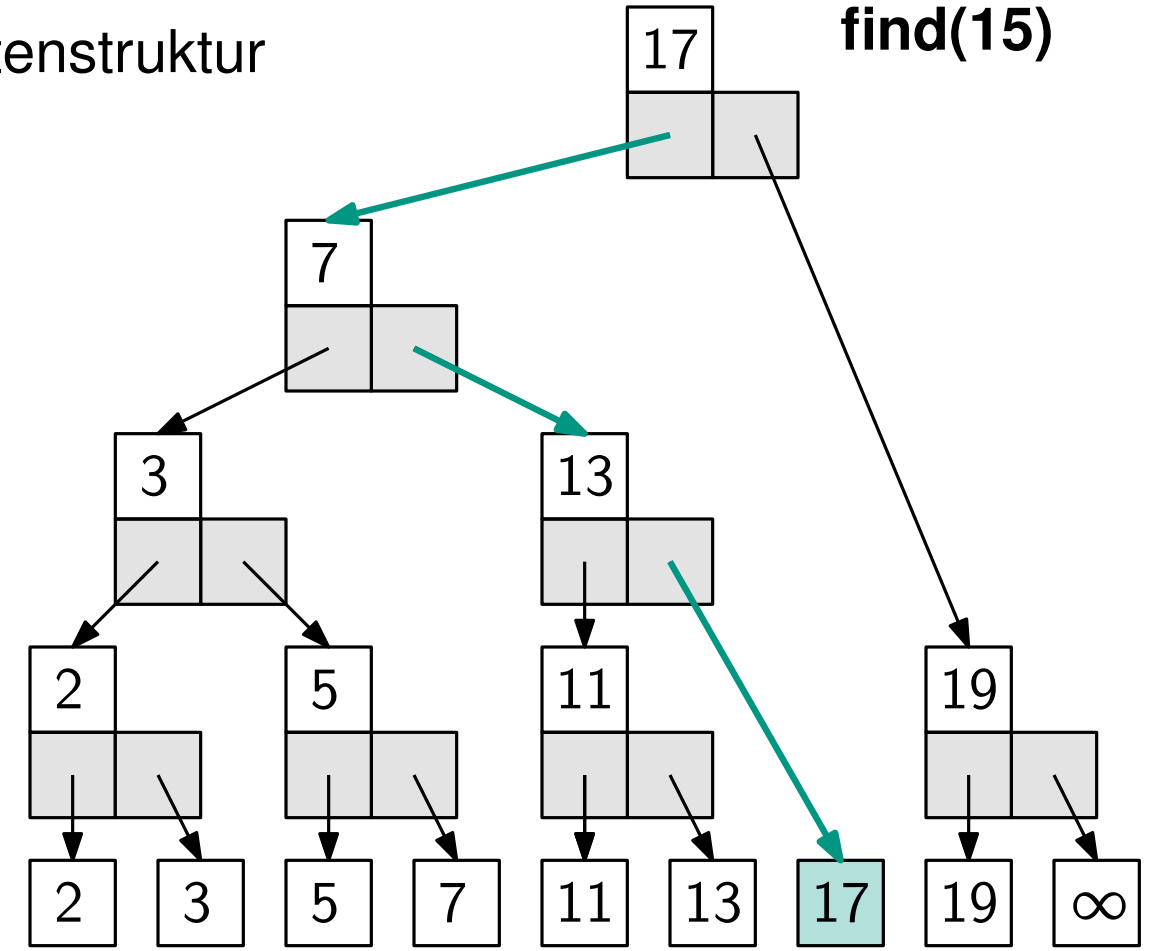
- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$
- **find:** Element (oder nächstgrößeres)
  - steige dazu in richtigenTeilbaum ab



binärer Suchbaum

Idee: Verwende Baum als Navigationsdatenstruktur

- **Blätter:** Elemente der Folge
- **innere Knoten:** (k, l, r)
  - k : Spaltschlüssel
  - l : linker Teilbaum
    - Blätter haben Schlüssel $\leq k$
  - r : rechter Teilbaum
    - Blätter haben Schlüssel $> k$
- **find:** Element (oder nächstgrößeres)
  - steige dazu in richtigenTeilbaum ab



kann durch ungeschicktes Einfügen/ Löschen zu Liste ausarten

binärer Suchbaum

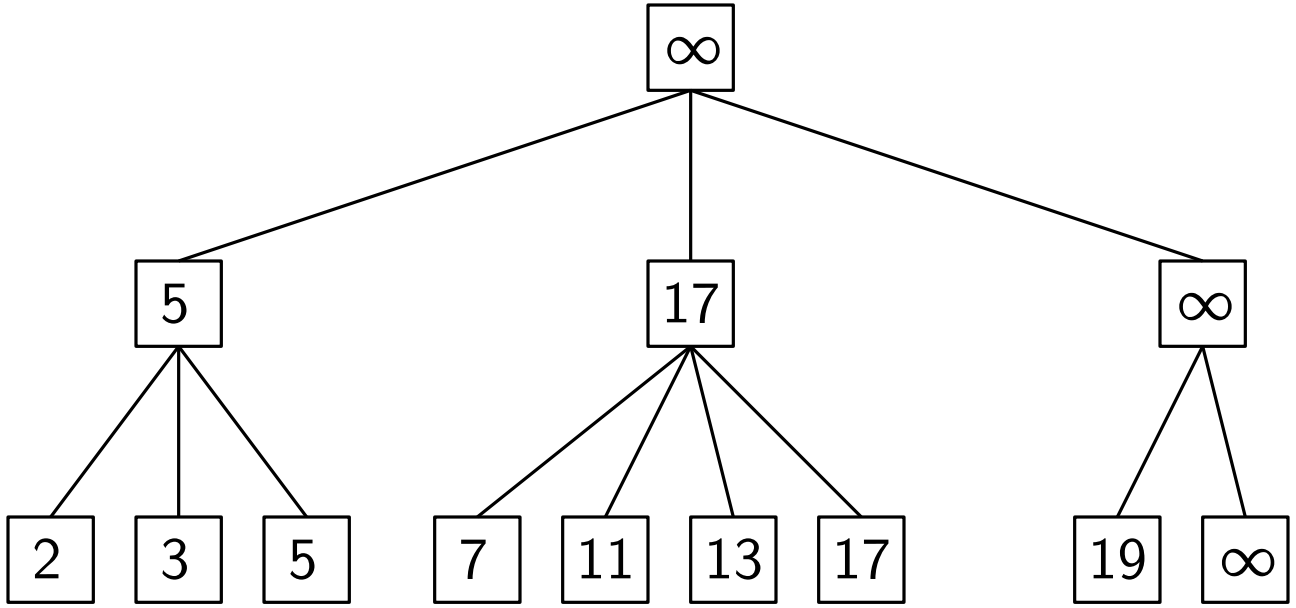
PINGO:

- Gebeben: Zahlen von 1 bis 1000
- Gesucht: 412

Welche Sequenz von Schlüsselwerten kann existieren?

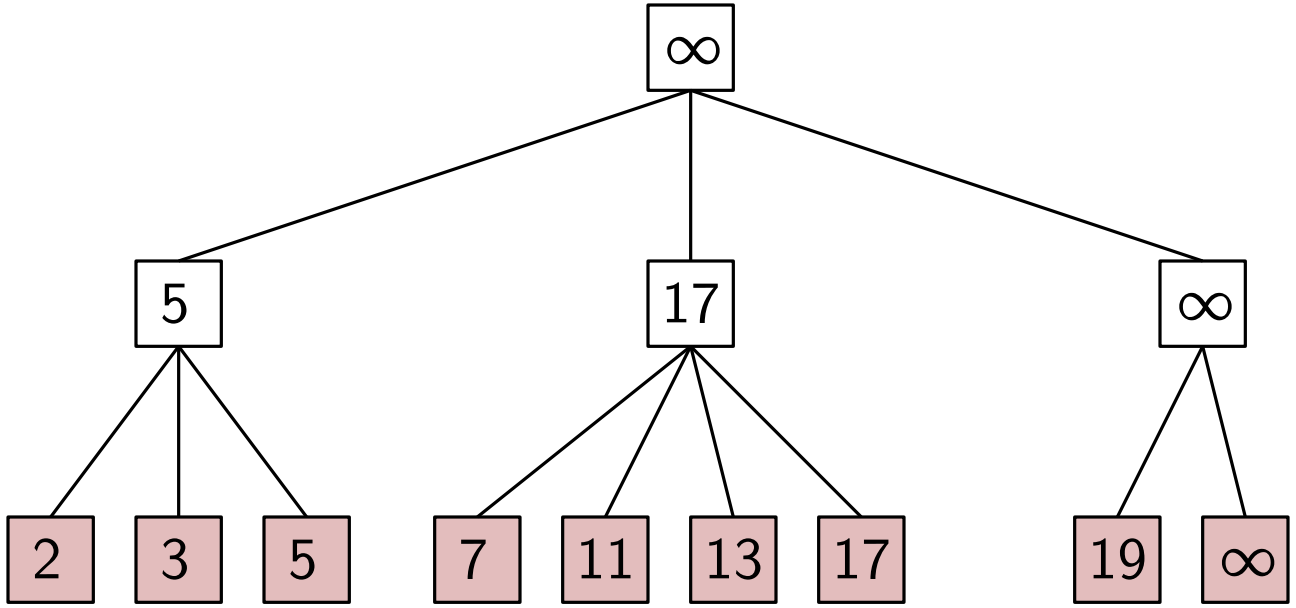
- (1) 998, 14, 36, 512, 247, 309, 499, 412
- (2) 565, 501, 181, 673, 500, 480, 427, 412
- (3) 2, 837, 547, 137, 230, 453, 402, 412
- (4) 765, 699, 643, 555, 270, 315, 411, 412
- (5) 666, 245, 598, 301, 572, 365, 500, 412
- (6) 666, 182, 575, 194, 483, 317, 595, 412
- (7) 2, 781, 776, 633, 215, 545, 214, 412

Aufbau (a,b)-Baum



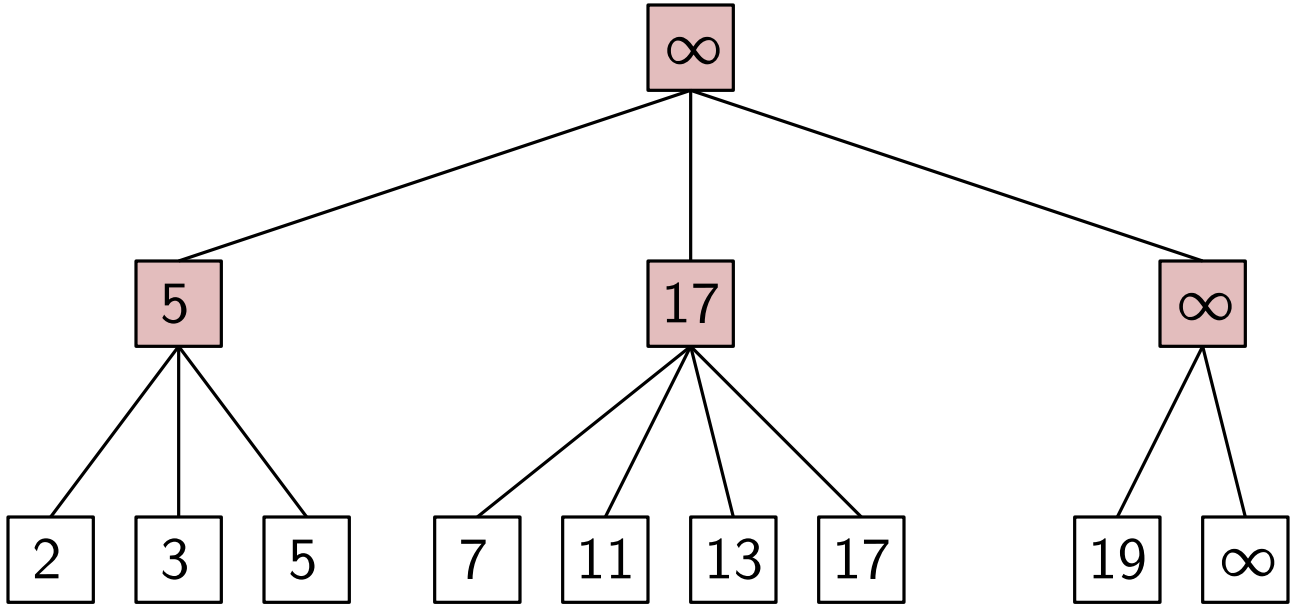
Aufbau (a,b)-Baum

- Blätter sind Elemente der Folge



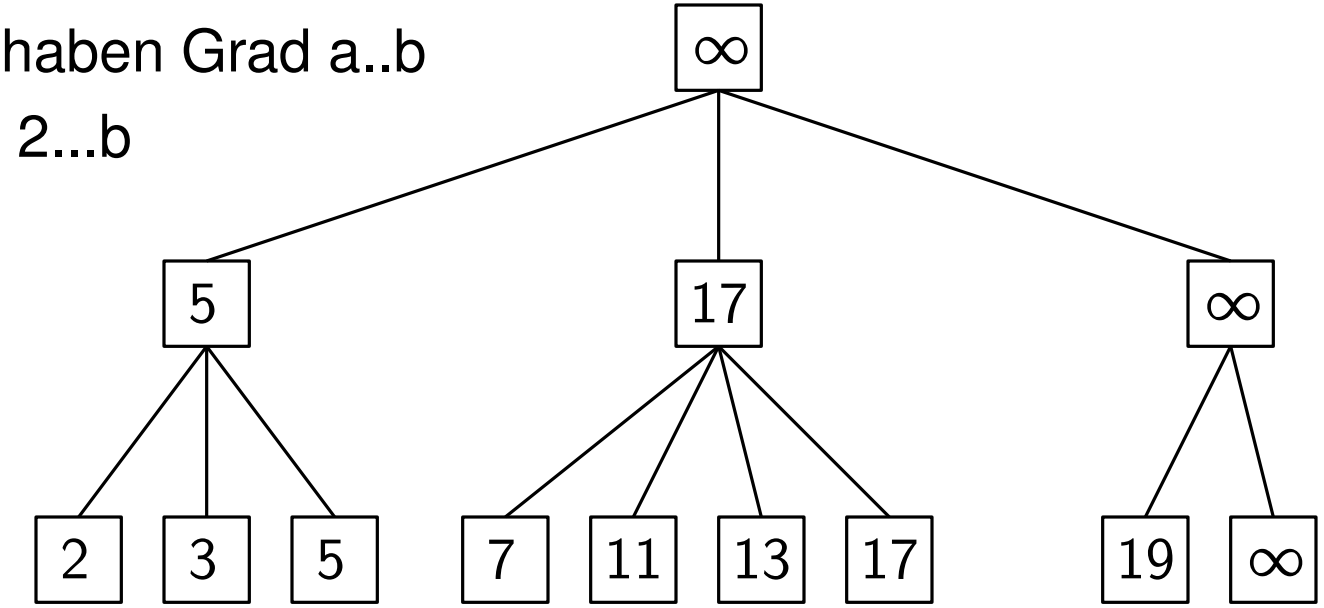
Aufbau (a,b)-Baum

- Blätter sind Elemente der Folge
- Andere Knoten sind Suchstruktur
  - Knotenlabel: größter Key seiner Kinder



Aufbau (a,b)-Baum

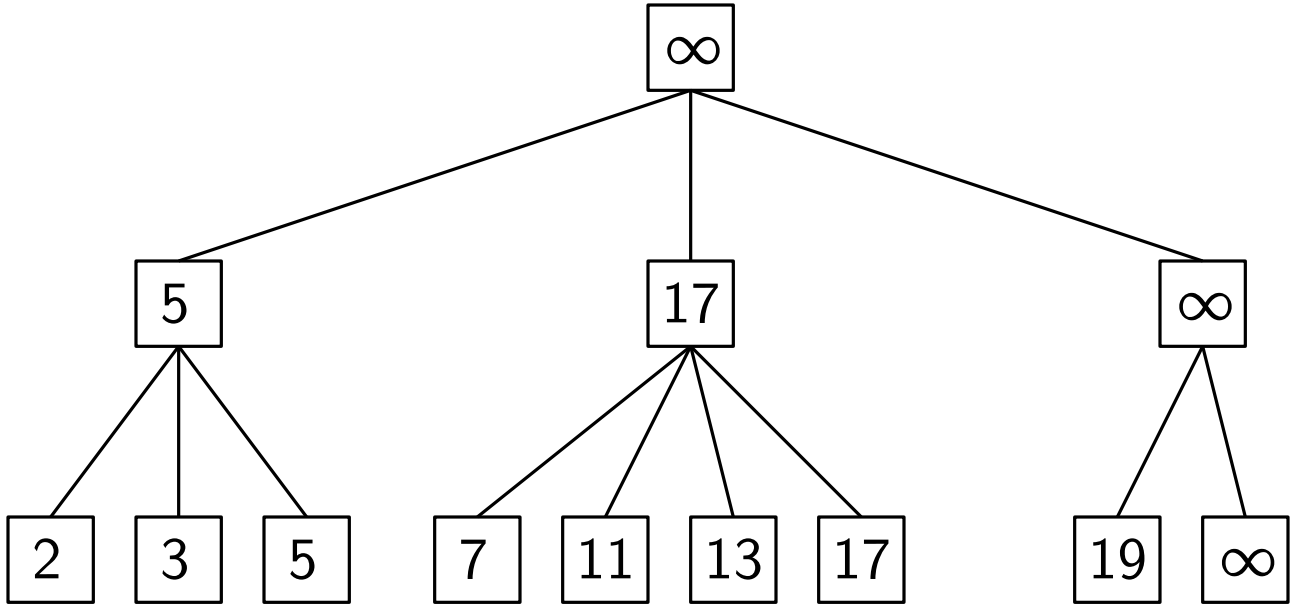
- Blätter sind Elemente der Folge
- Andere Knoten sind Suchstruktur
  - Knotenlabel: größter Key seiner Kinder
- Innere Knoten haben Grad a..b
  - Wurzel: Grad 2...b



Aufbau (a,b)-Baum

find(k)

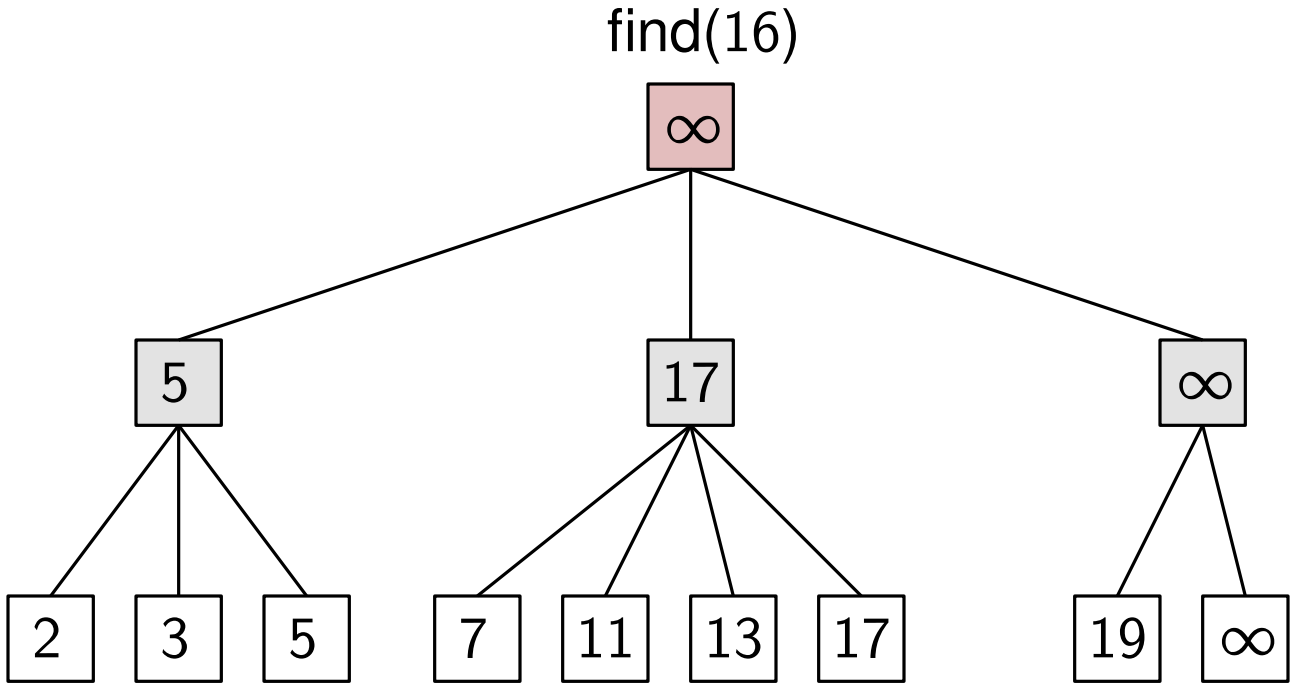
- Find(k): Wähle kleinstes Kind mit Label >k



Aufbau (a,b)-Baum

find(k)

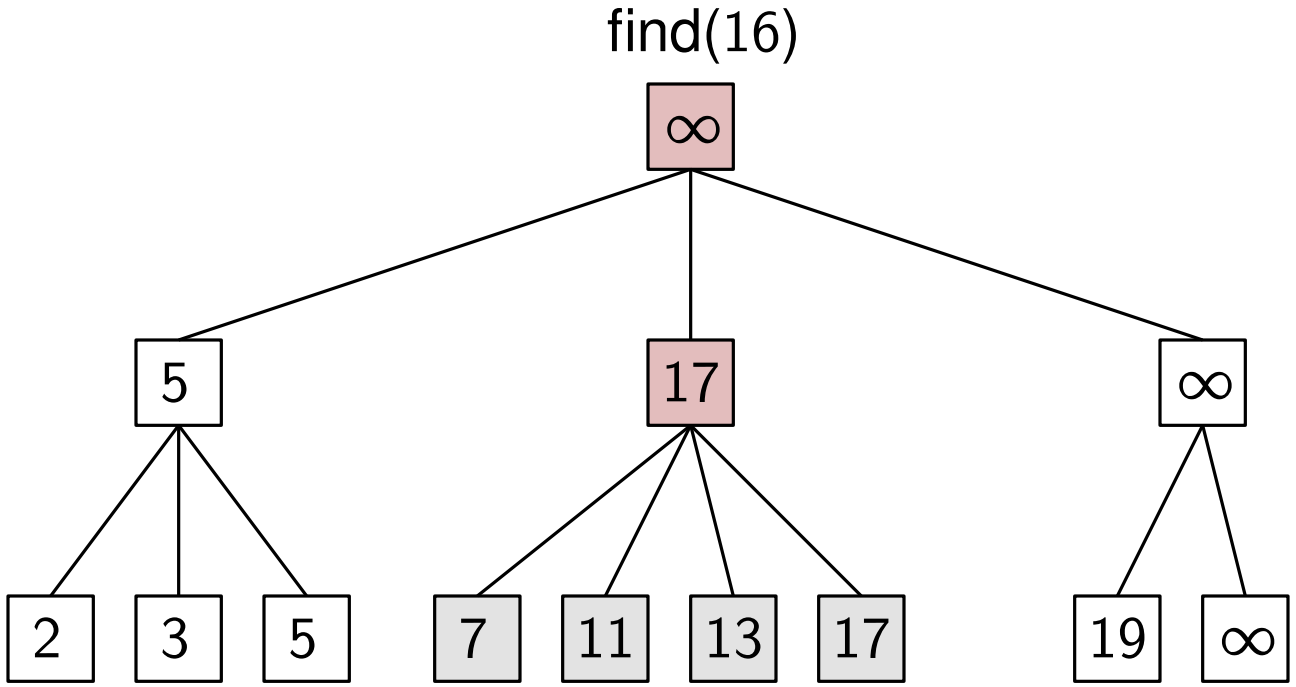
- Find(k): Wähle kleinstes Kind mit Label >k



Aufbau (a,b)-Baum

find(k)

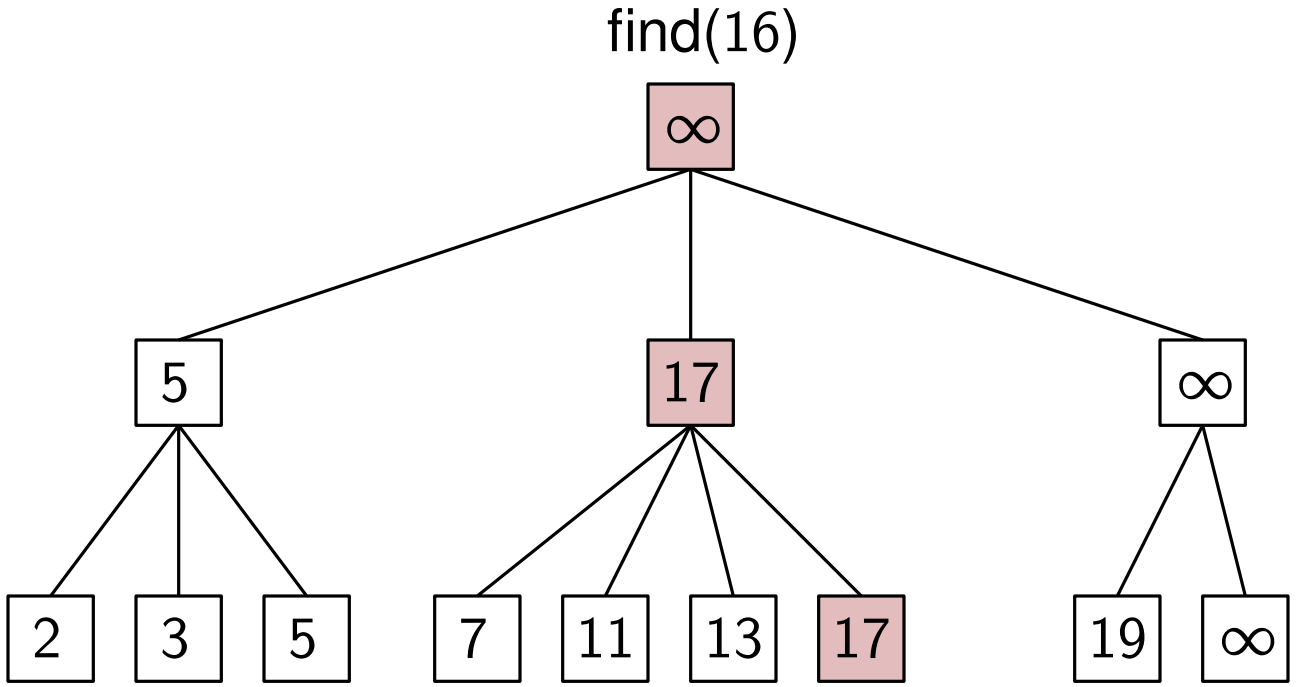
- Find(k): Wähle kleinstes Kind mit Label >k



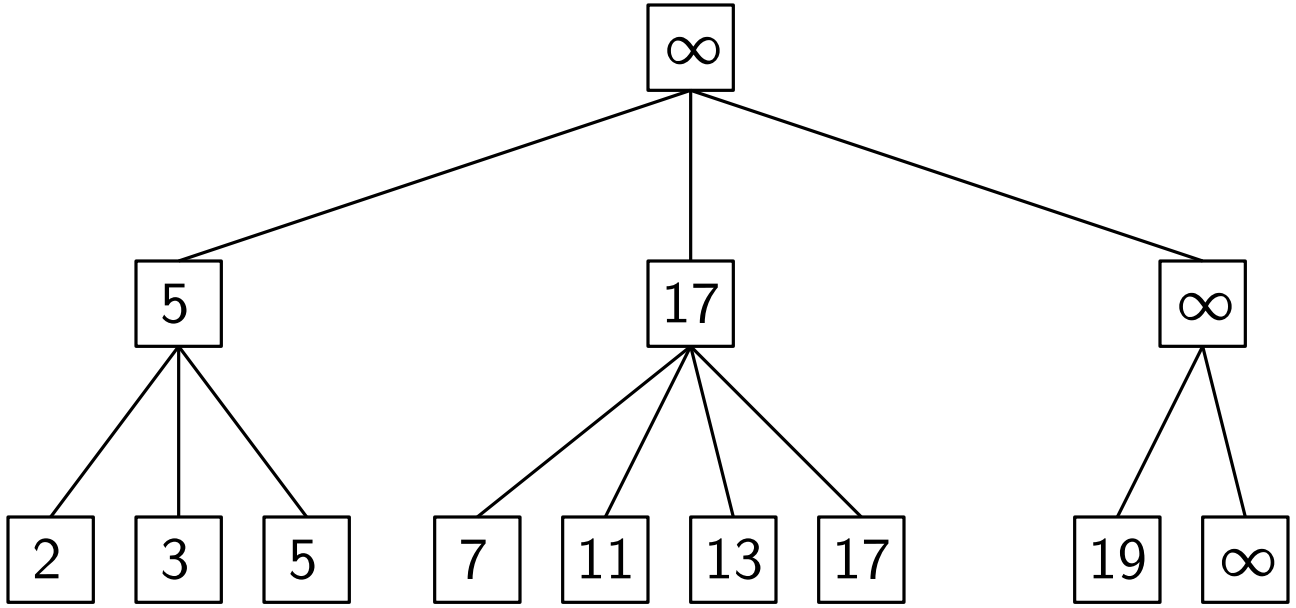
Aufbau (a,b)-Baum

find(k)

- Find(k): Wähle kleinstes Kind mit Label >k



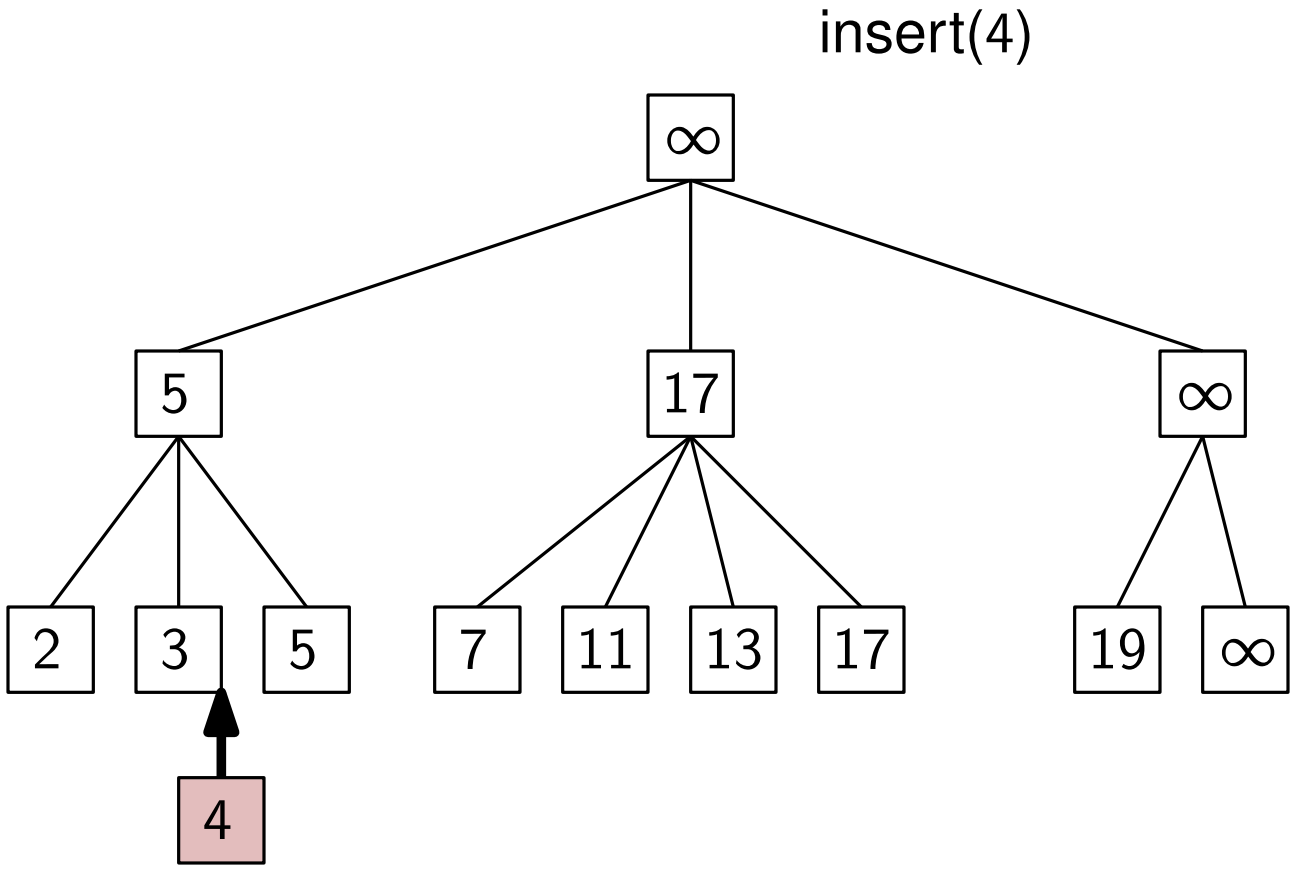
insert



insert

Fall 1

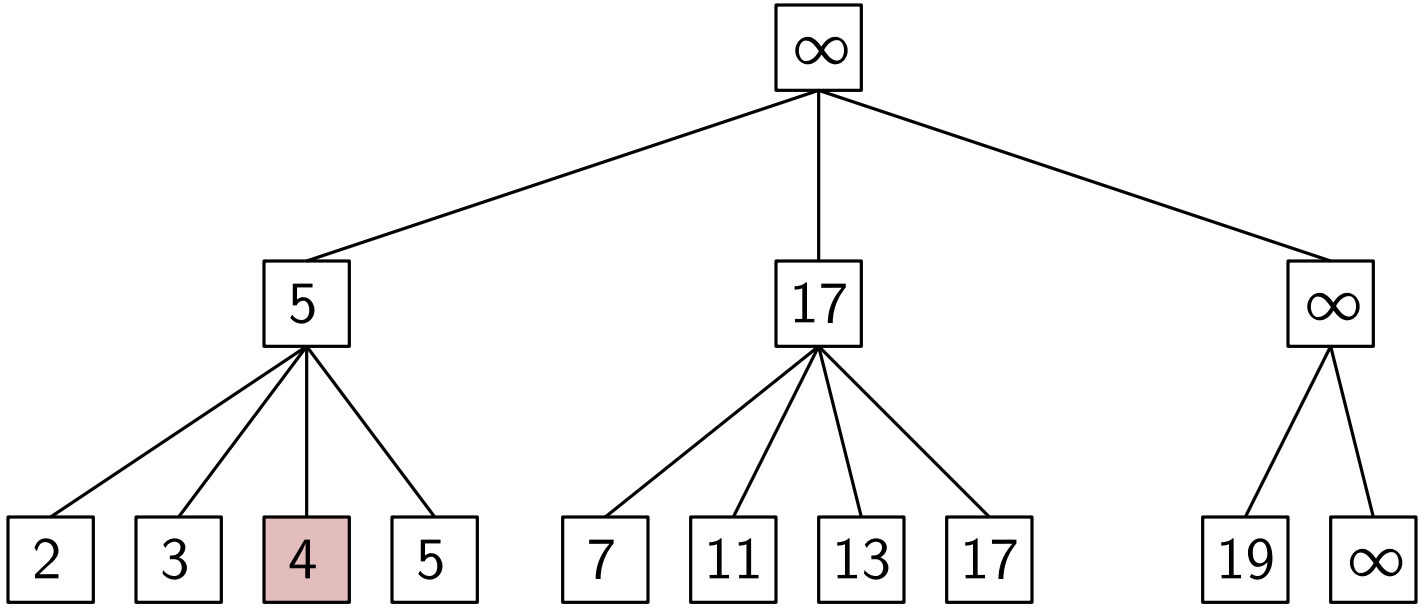
- Elternknoten hat weniger als **b** Kinder



insert

Fall 1

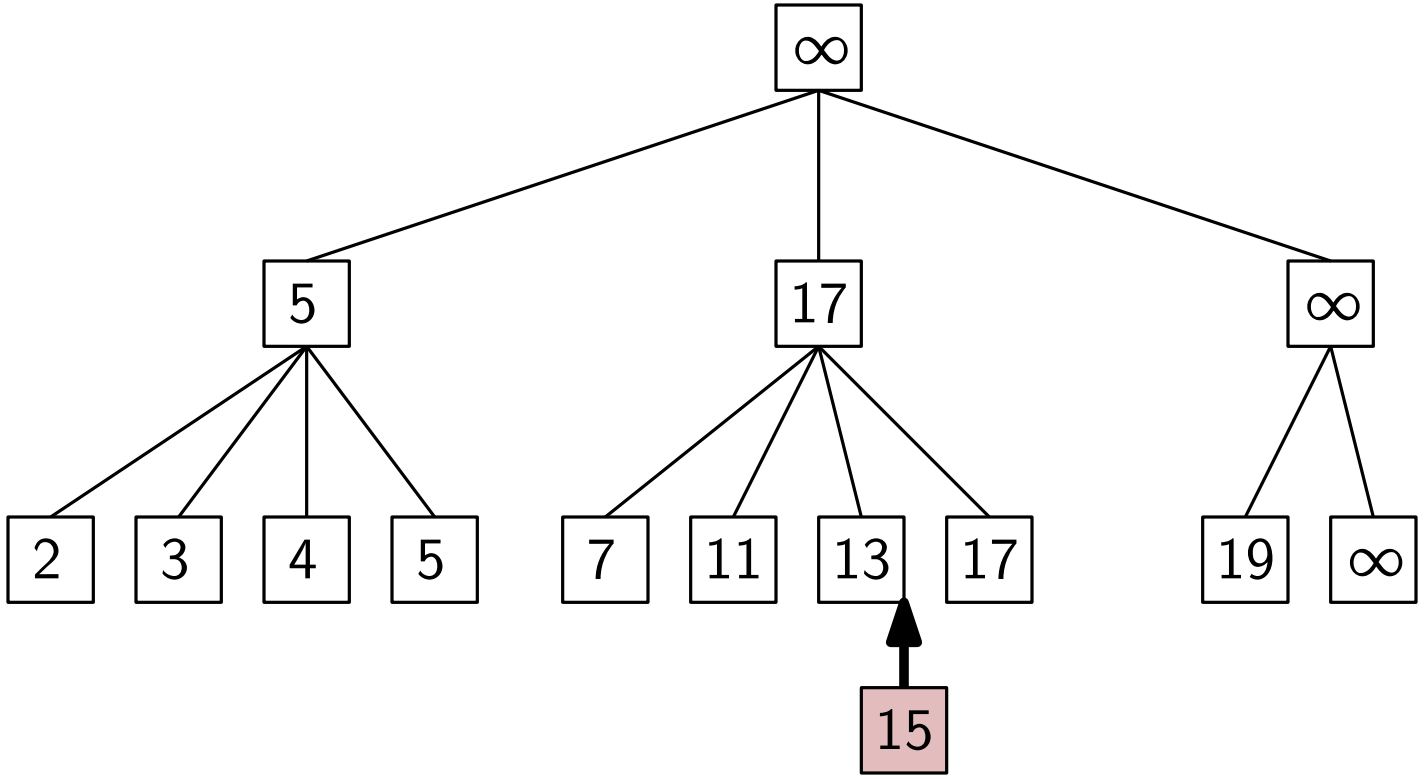
- Elternknoten hat weniger als **b** Kinder
- Element einfach einfügen



insert

Fall 2

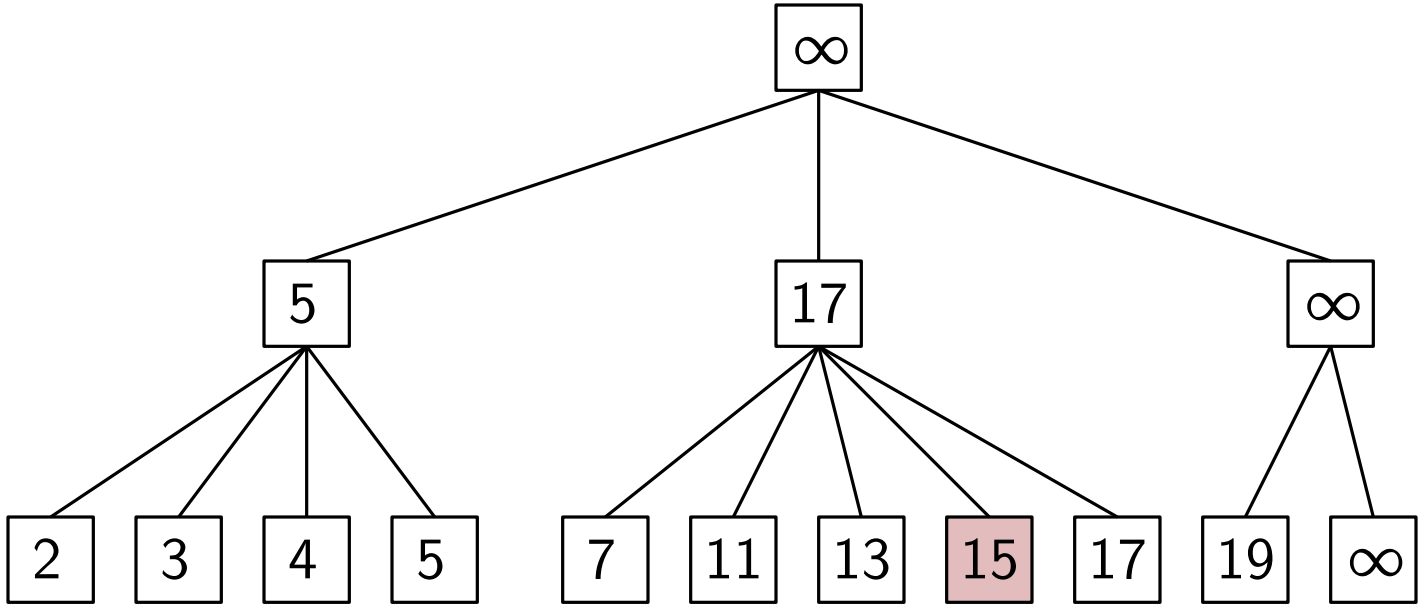
- Elternknoten hat schon **b** Kinder



insert

Fall 2

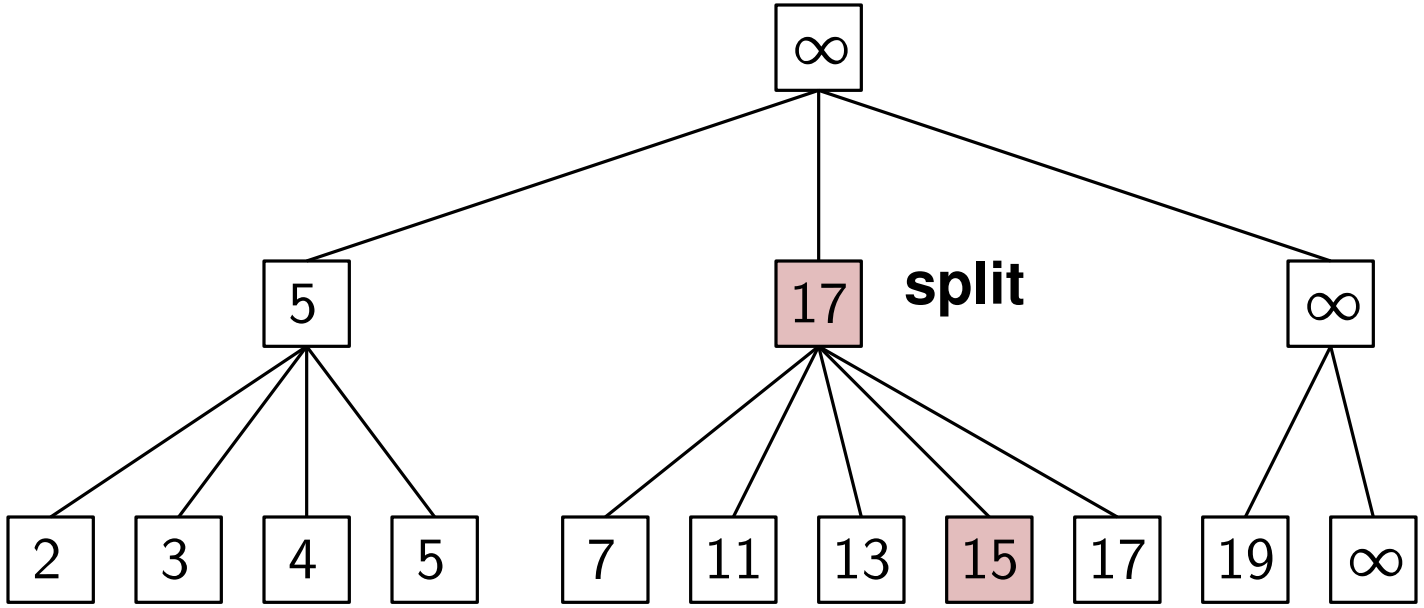
- Elternknoten hat schon **b** Kinder



insert

Fall 2

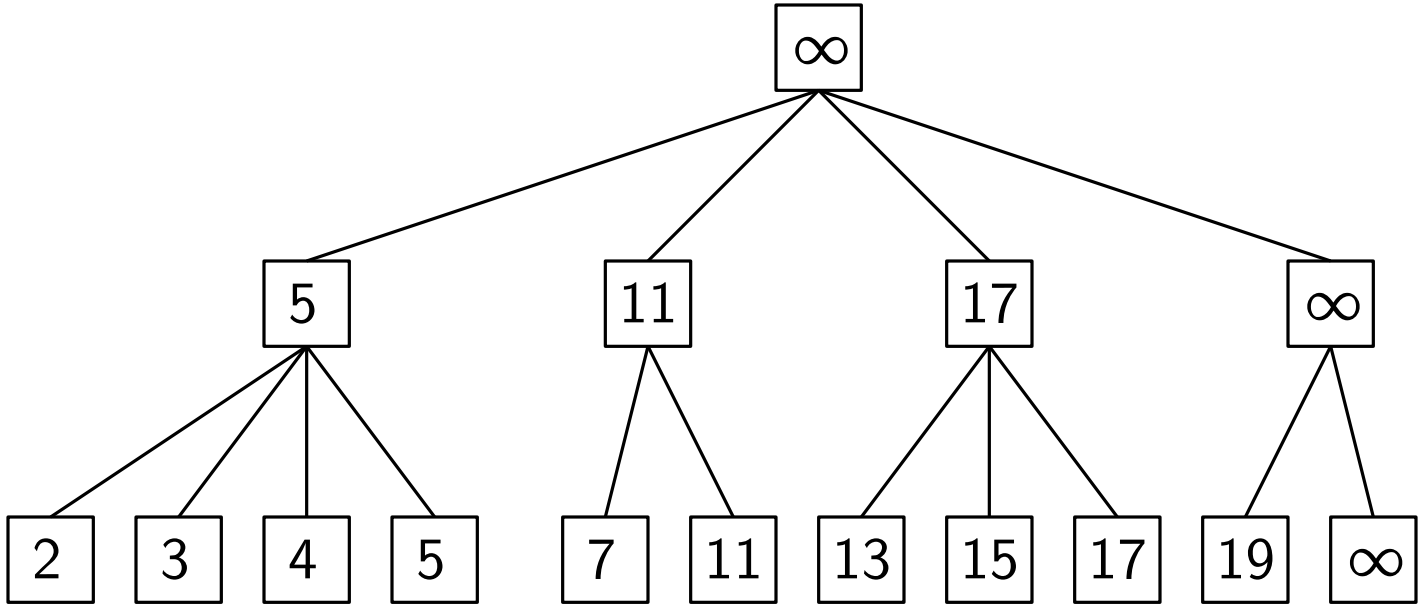
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei



insert

Fall 2

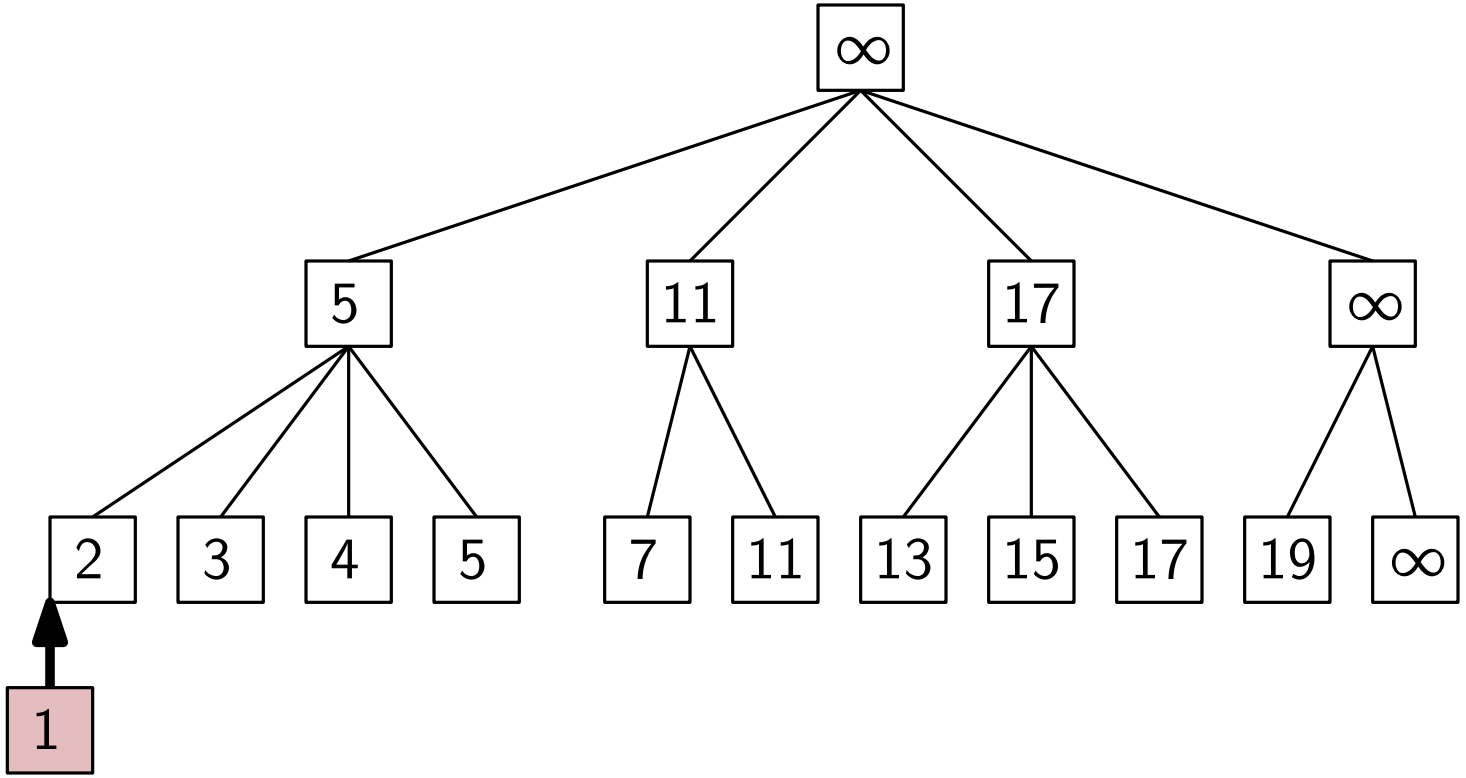
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei



insert

Fall 2

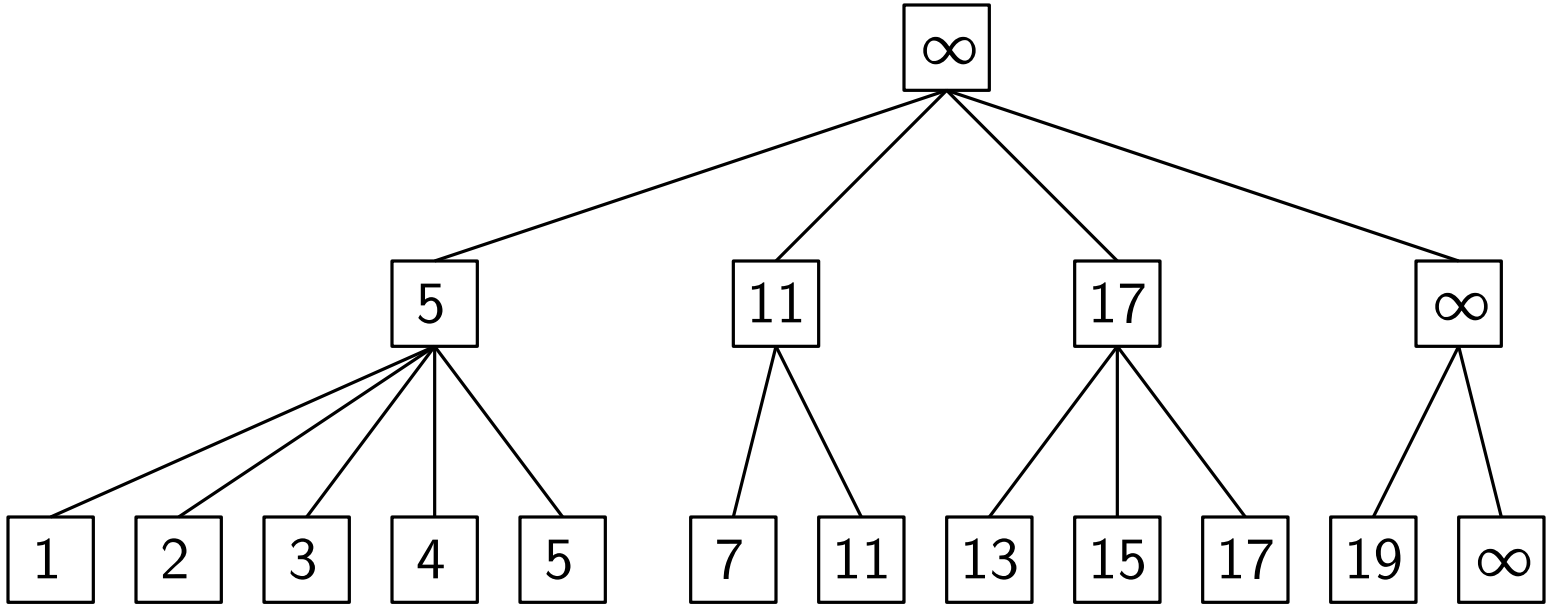
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei



insert

Fall 2

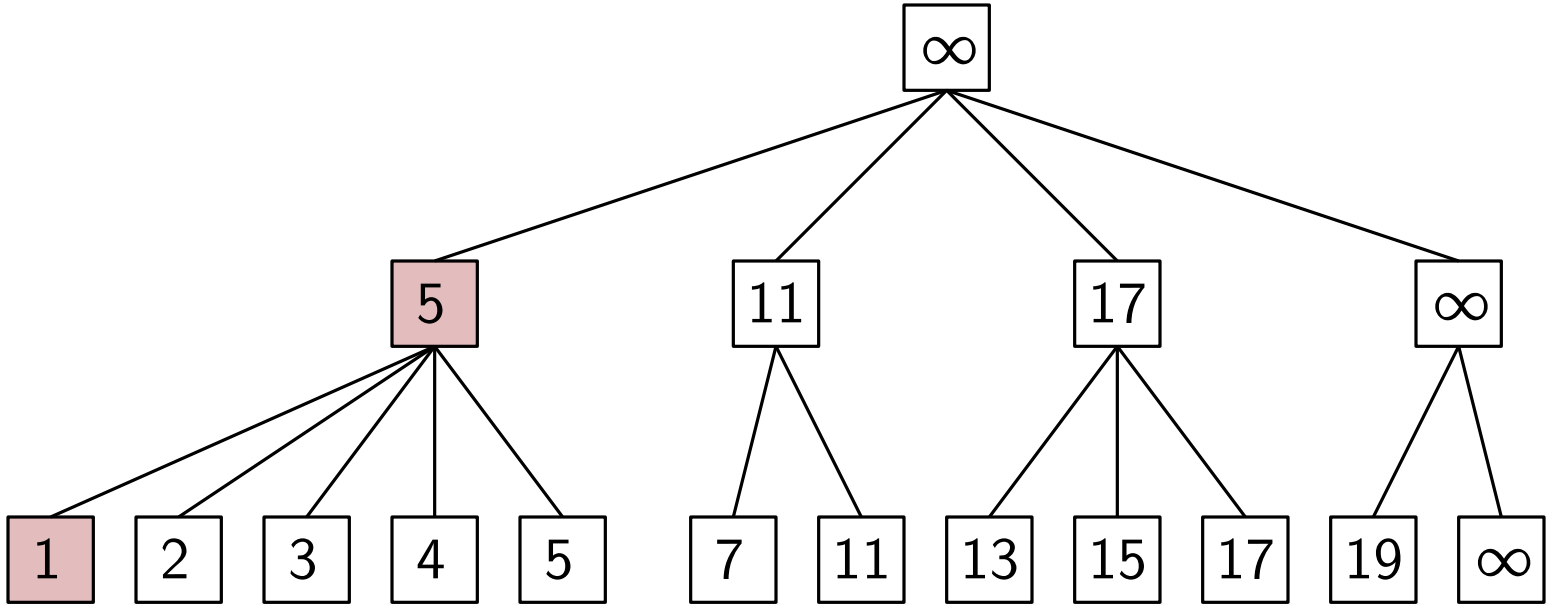
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei



insert

Fall 2

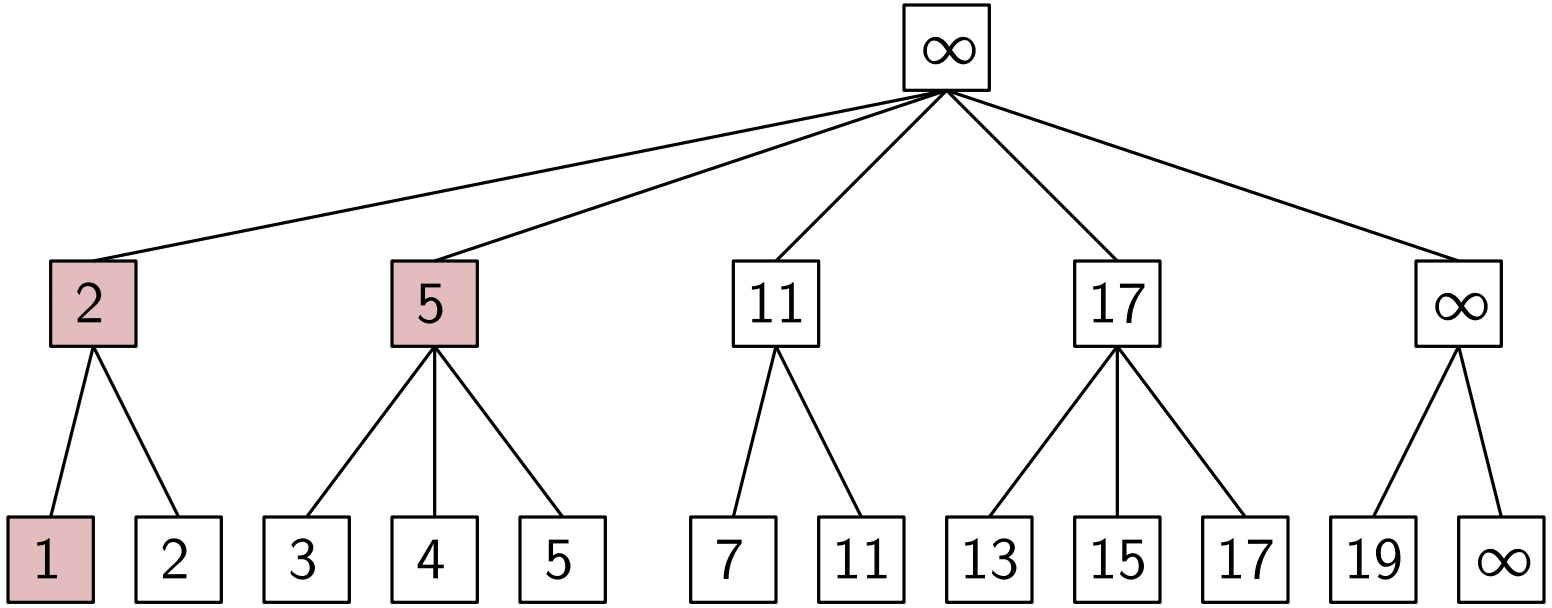
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei



insert

Fall 2

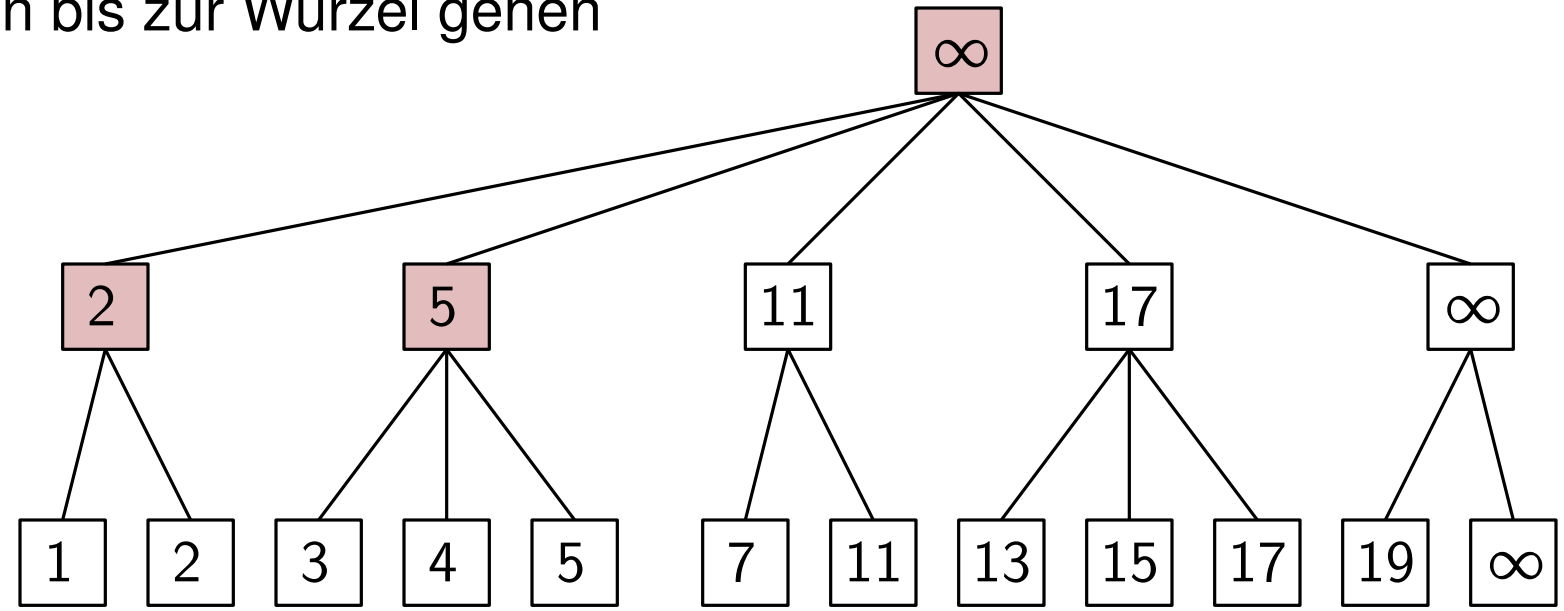
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei



insert

Fall 2

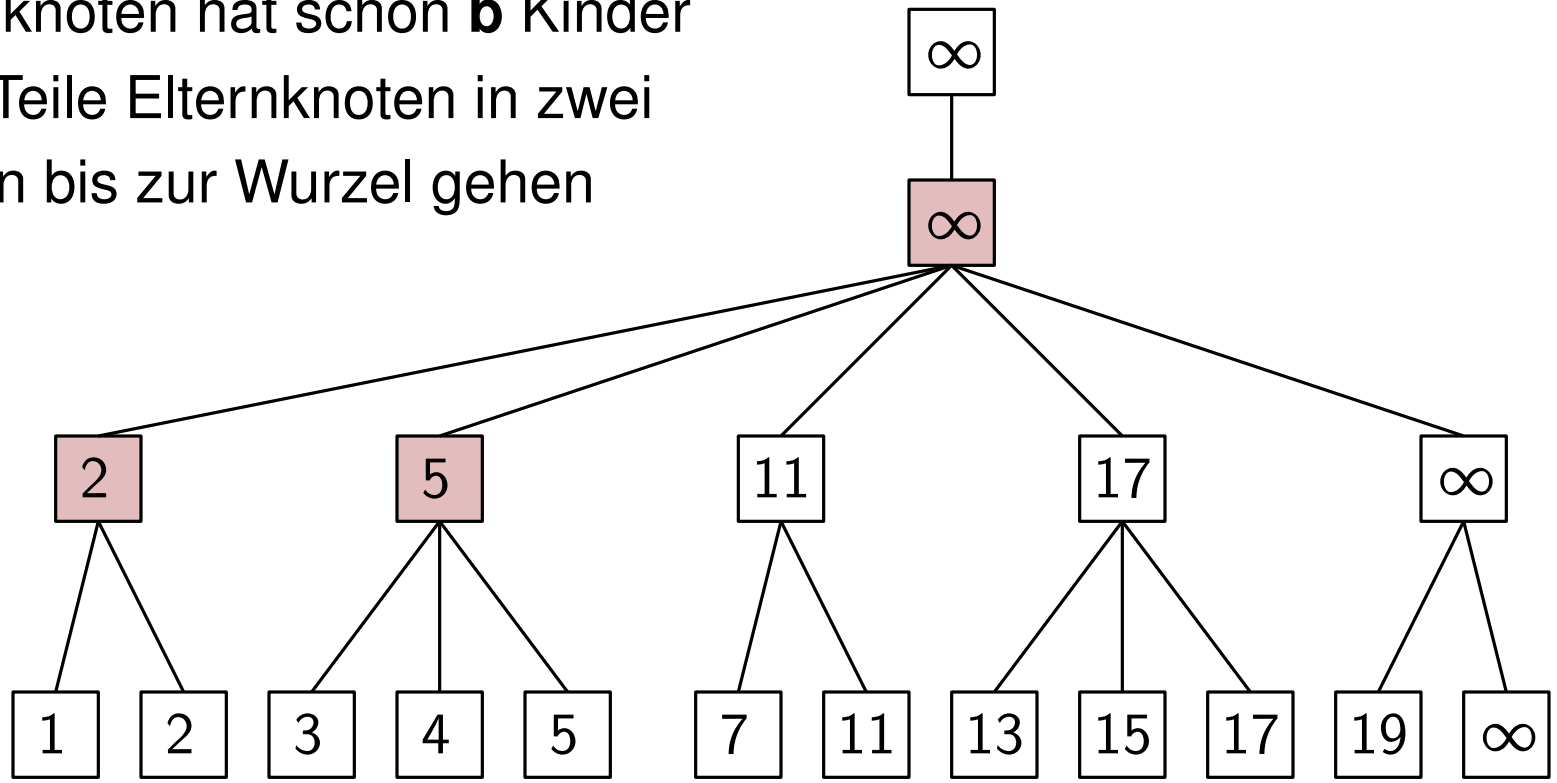
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei
  - Kann bis zur Wurzel gehen



insert

Fall 2

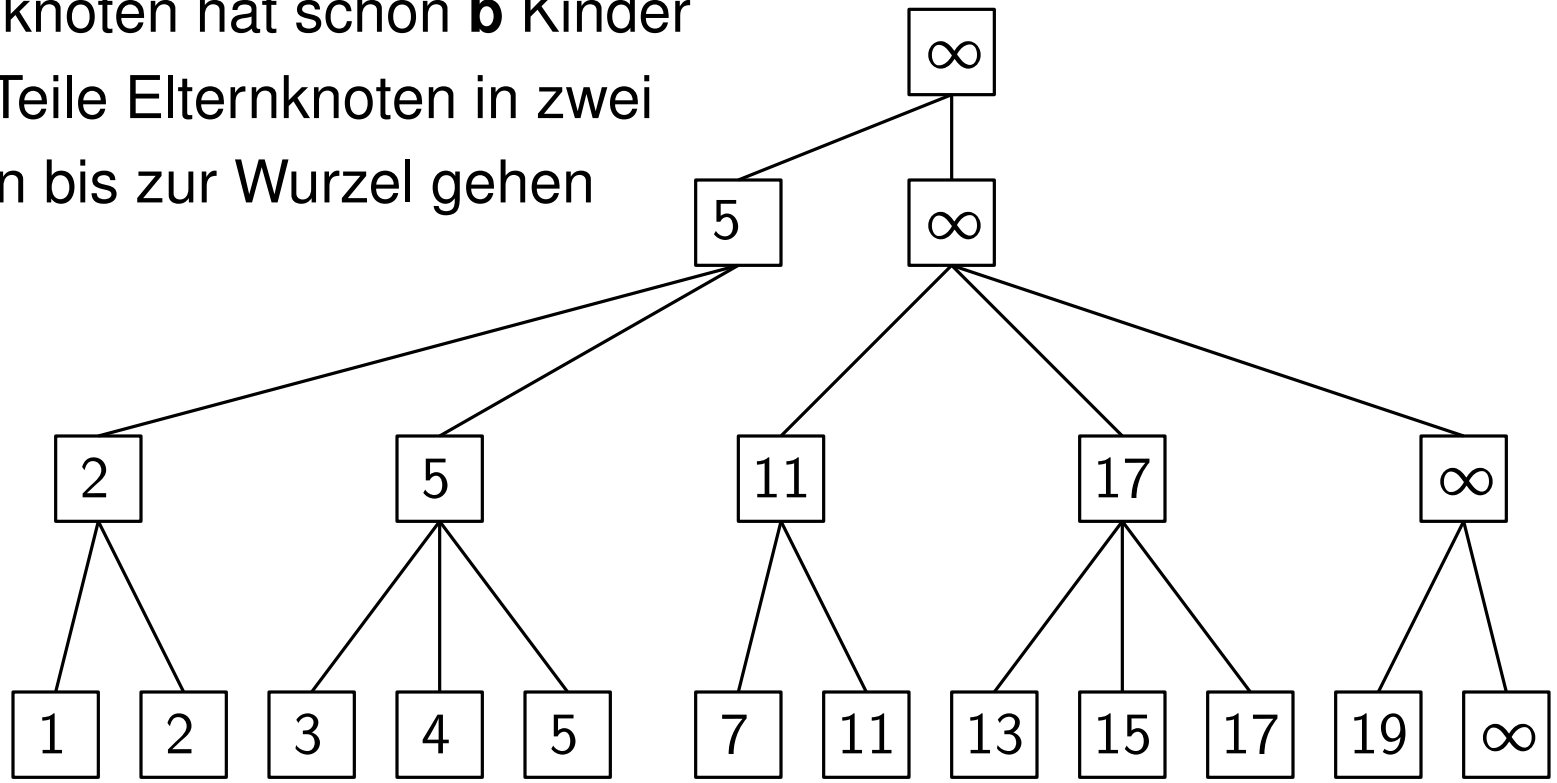
- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei
  - Kann bis zur Wurzel gehen



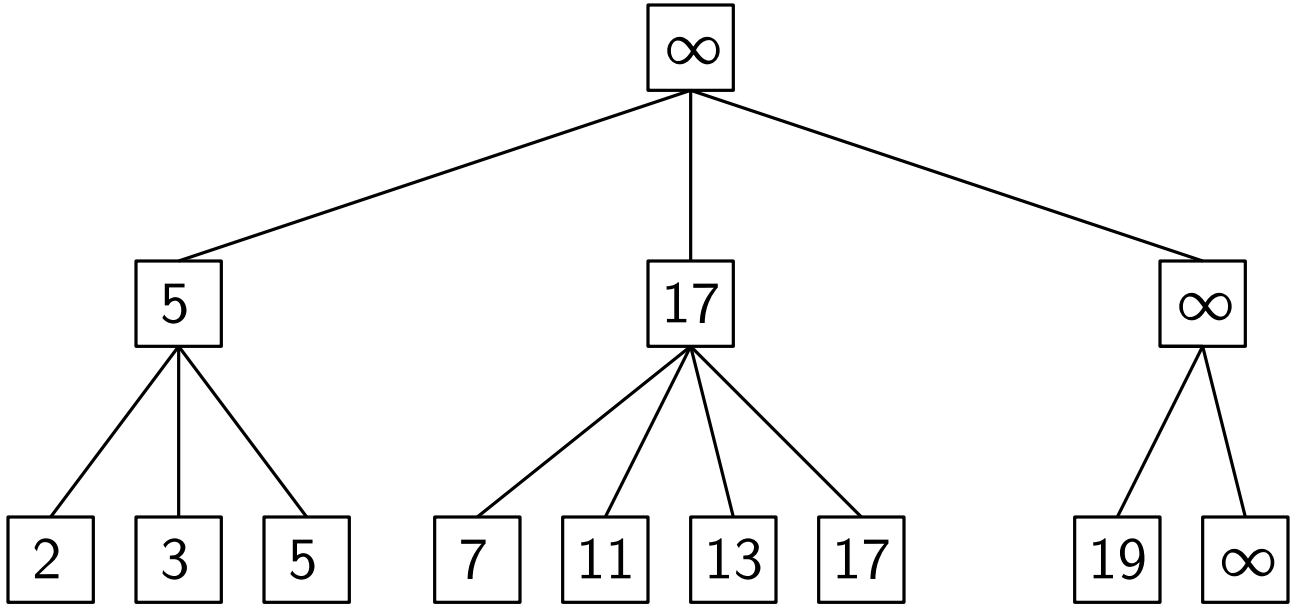
insert

Fall 2

- Elternknoten hat schon **b** Kinder
- **split:**Teile Elternknoten in zwei
  - Kann bis zur Wurzel gehen



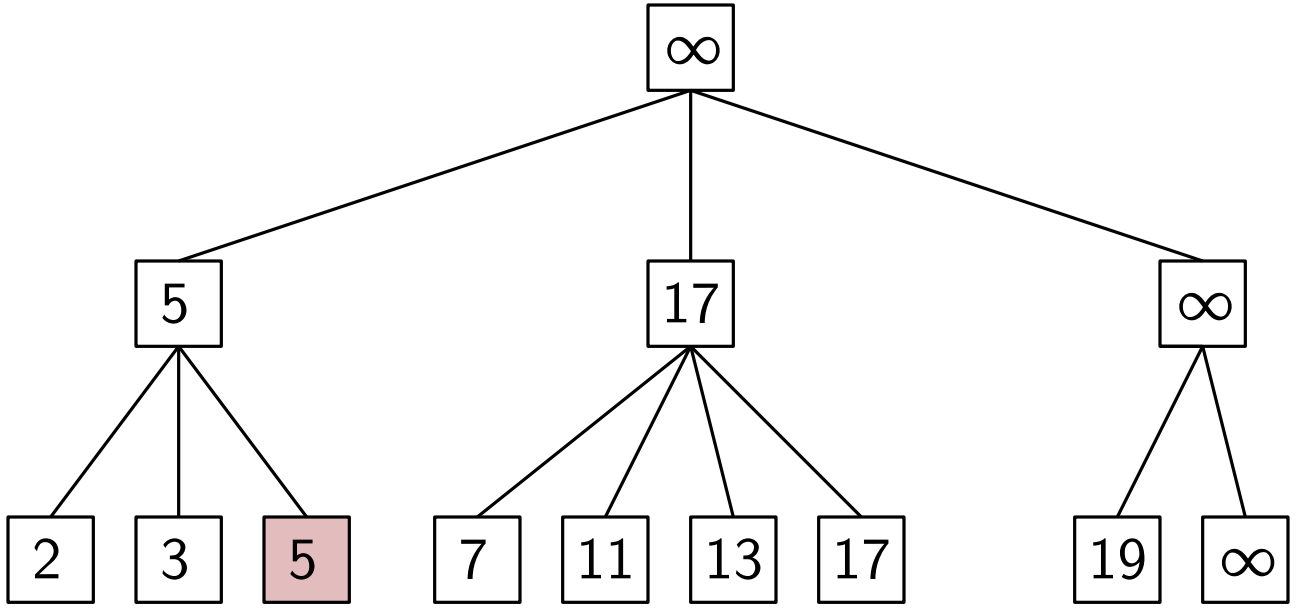
remove



remove

Fall 1

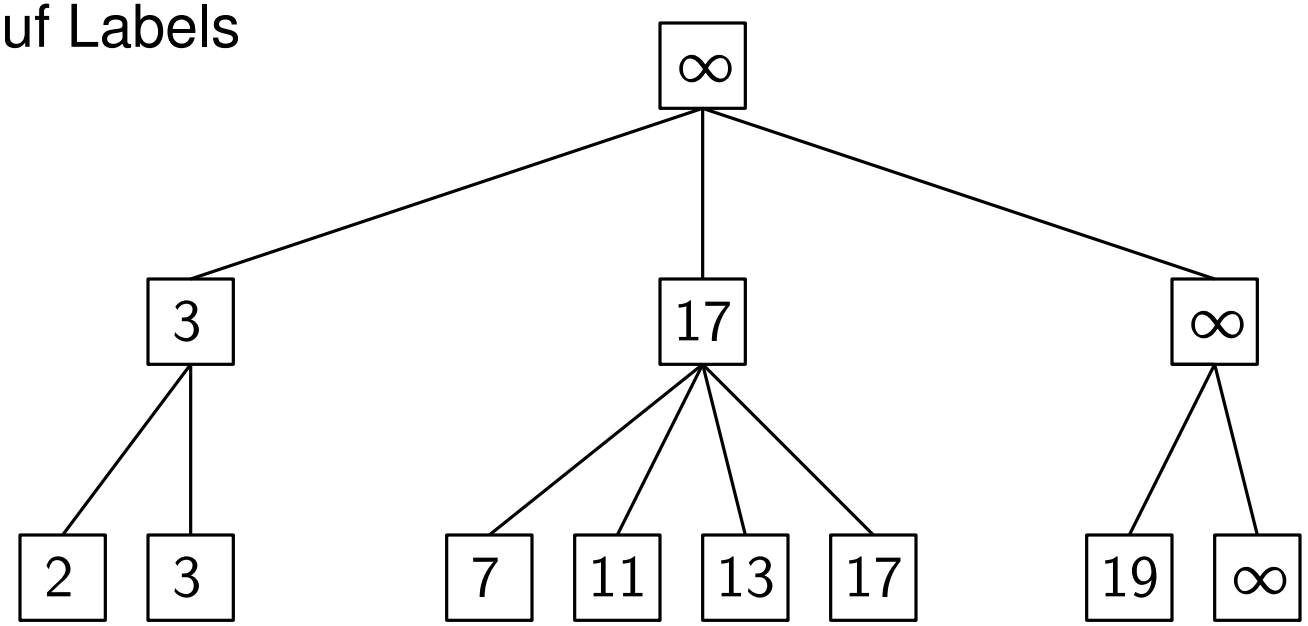
- Elternknoten hat mehr als **a** Kinder



remove

Fall 1

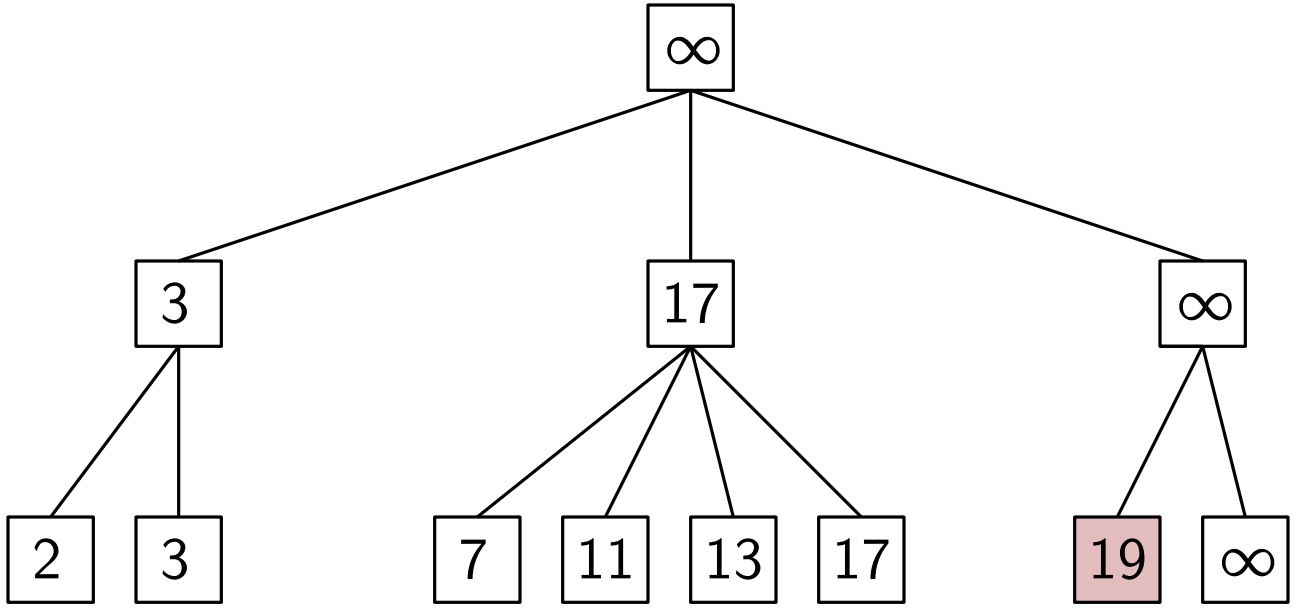
- Elternknoten hat mehr als **a** Kinder
- Entferne Element
  - Achte dabei auf Labels



remove

Fall 2.1

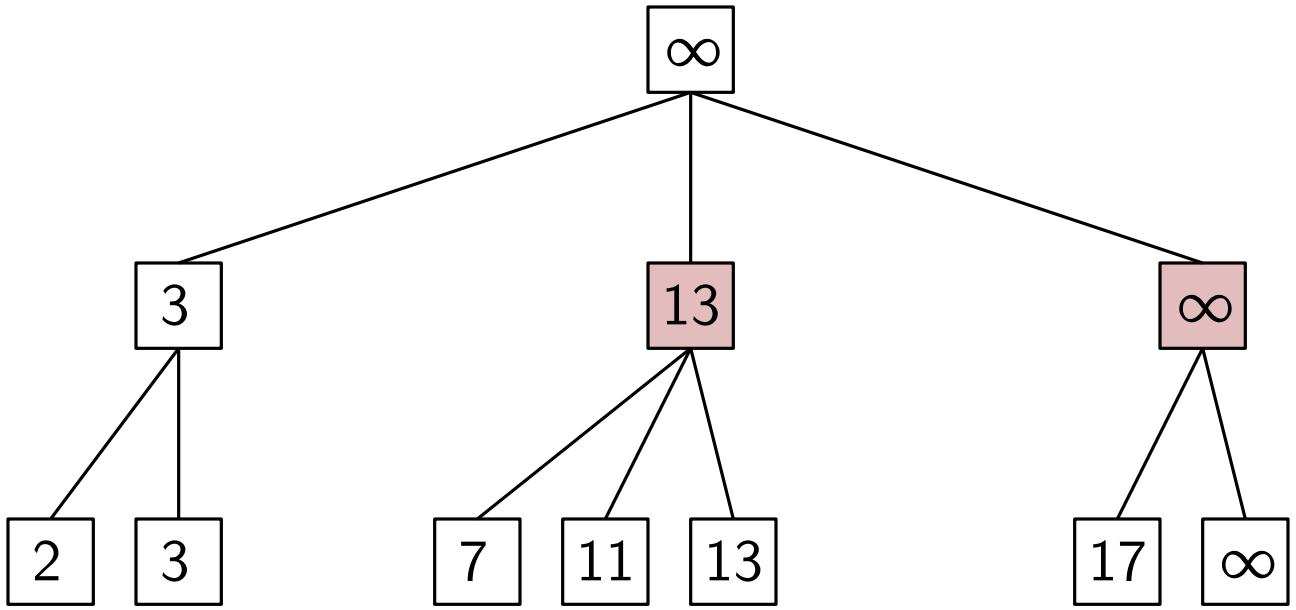
- Elternknoten p hat weniger als a Kinder



remove

Fall 2.1

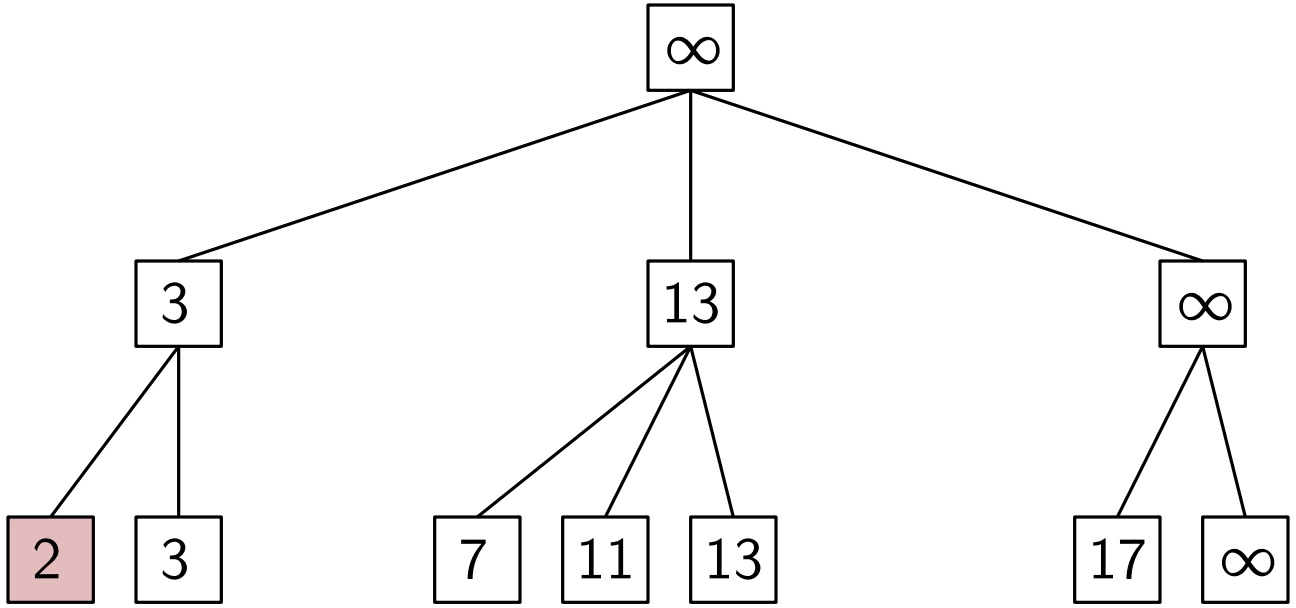
- Elternknoten p hat weniger als a Kinder
- Falls p und Geschwisterknoten p' mehr als b Kinder
  - **rebalance**



remove

Fall 2.2

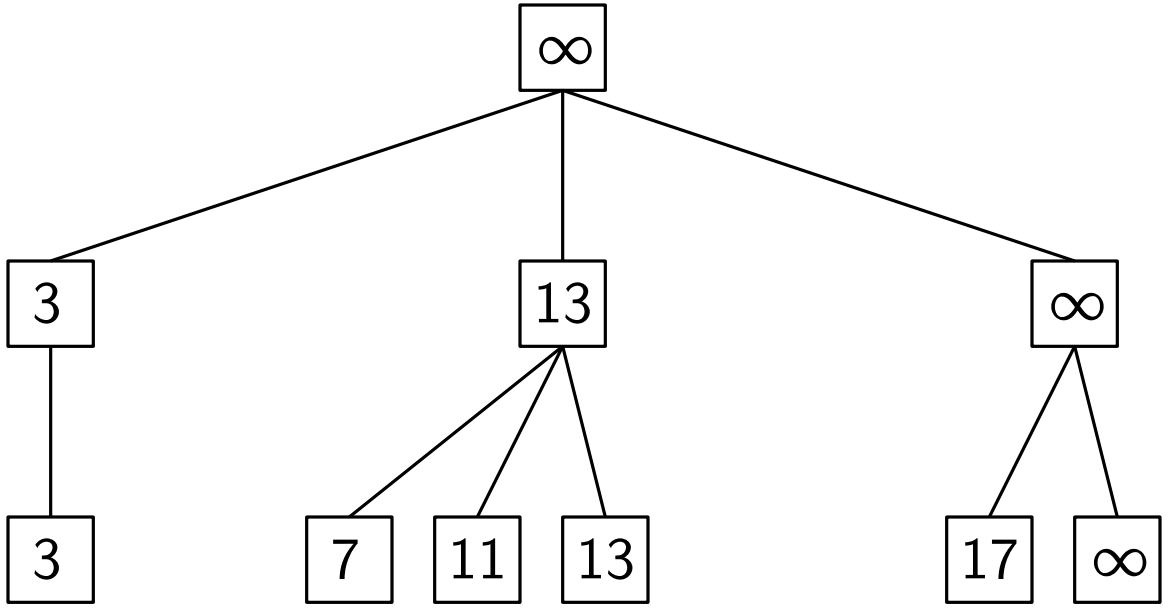
- Elternknoten p hat weniger als a Kinder



remove

Fall 2.2

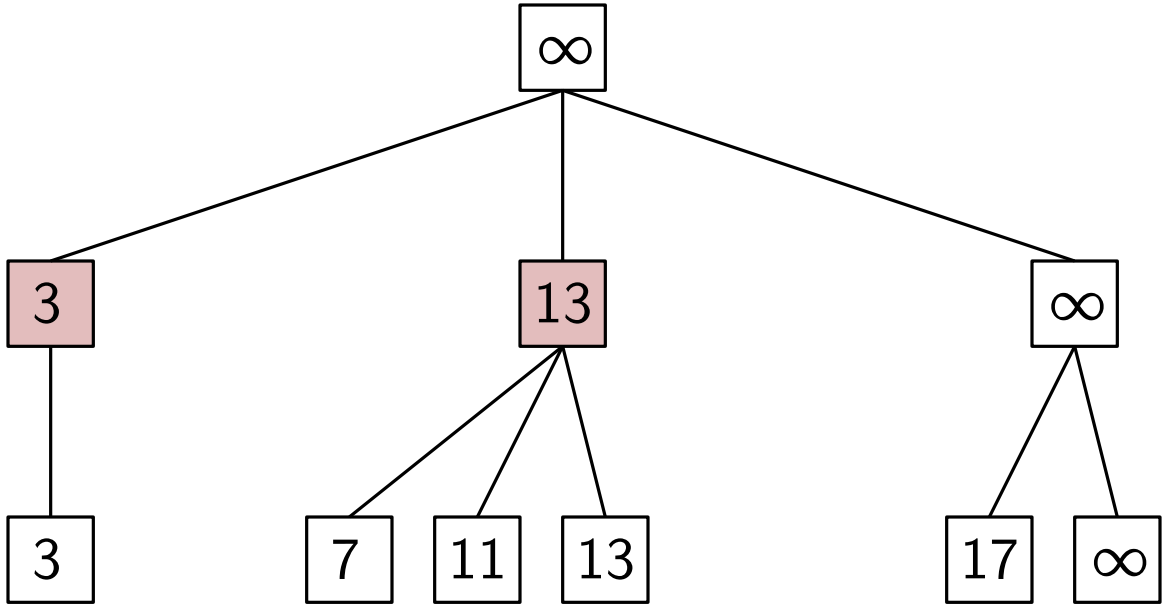
- Elternknoten p hat weniger als a Kinder



remove

Fall 2.2

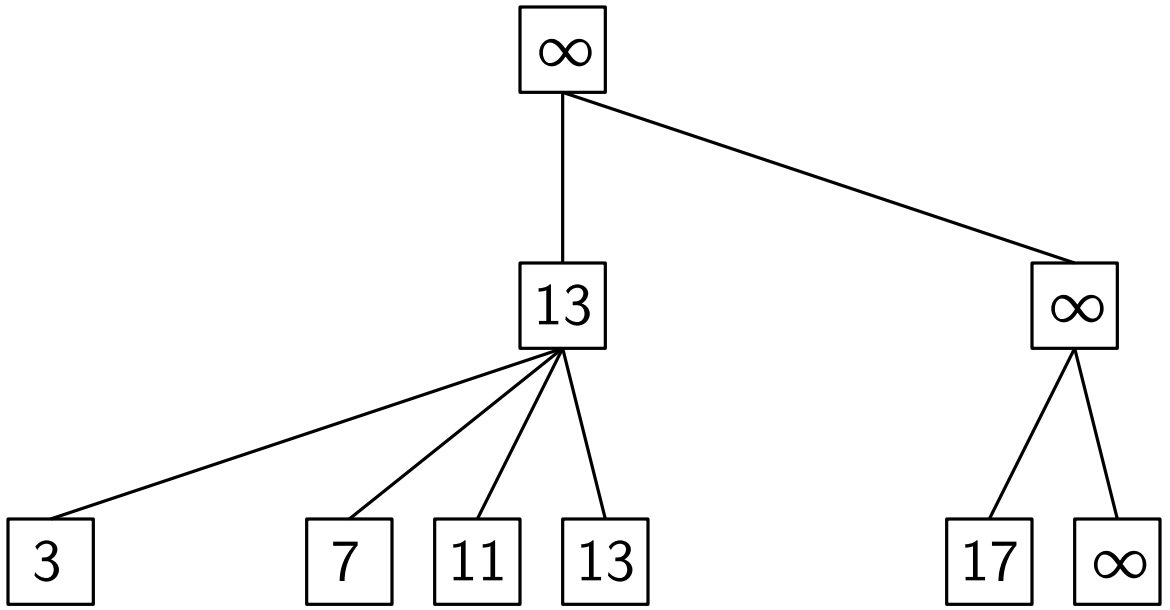
- Elternknoten p hat weniger als a Kinder
- Falls p und Geschwisterknoten $p' \leq$ b Kinder
  - **fuse**



remove

Fall 2.2

- Elternknoten p hat weniger als a Kinder
- Falls p und Geschwisterknoten $p' \leq$ b Kinder
  - **fuse**



Zusammenfassung (a,b)-Bäume

- Navigationsdatenstrukur auf einer sortierten Folge
- Binärer Suchbaum kann zu Liste ausarten
  - Damit find in $\mathcal{O}(n)$
- (a,b)-Baum ist balanciert
  - Jeder Knoten hat mindestens a bis maximal b Kinder
  - find, insert, remove in $\mathcal{O}(\log n)$
    - Bei zu wenig Kind-Knoten: **rebalance** oder **fuse**
    - Bei zu vielen Kind-Knoten: **split**