# FANTASTISCHE HEAPS

… und wo sie zu finden sind.
Datastructure Love!

**Florian Sihler**

11. November 2022
SP, Universität Ulm

All rights reserved

# FANTASTISCHE HEAPS

… und wo sie zu finden sind.
Datastructure Love!

Florian Sihler

Officially supported by the Pingu-Foundation for Emotional Support. There will be light and there will be cute waddlers!

11. November 2022
SP, Universität Ulm

All rights reserved

# 2.1 Disclaimer

[6]: *Episode-Rekursion* Sihler, 2021

[5]: *LaTeX-Package, tikzpingus* Sihler, 2021

# 2.2 Disclaimer

- Dieser „Heap“ hat nichts mit dem aus der Rekursions-Episode zu tun!

[6]: *Episode-Rekursion* Sihler, 2021

[5]: *LaTeX-Package, tikzpingus* Sihler, 2021

# 2.3 Disclaimer

- Dieser „Heap“ hat nichts mit dem aus der Rekursions-Episode zu tun!
- Wir beschränken uns auf Heap-*Bäume*

[6]: *Episode-Rekursion* Sihler, 2021

[5]: *LaTeX-Package, tikzpingus* Sihler, 2021

# 2.4 Disclaimer

- Dieser „Heap“ hat nichts mit dem aus der Rekursions-Episode zu tun!
- Wir beschränken uns auf Heap-*Bäume*
- Wir ignorieren Rotationen

[6]: *Episode-Rekursion* Sihler, 2021

[5]: *LaTeX-Package, tikzpingus* Sihler, 2021

# 2.5 Disclaimer

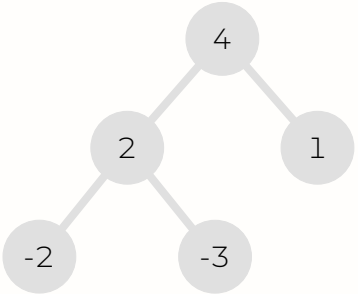
- Dieser „Heap“ hat nichts mit dem aus der Rekursions-Episode zu tun!
- Wir beschränken uns auf Heap-*Bäume*
- Wir ignorieren Rotationen
- Die Folien sind in $\sim 3.5$ Stunden entstanden also reduziert

[6]: *Episode-Rekursion* Sihler, 2021

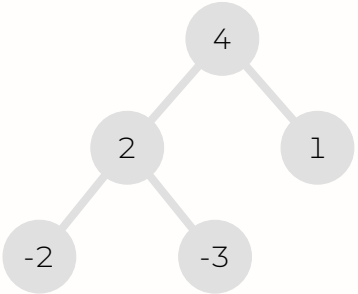
[5]: *LaTeX-Package, tikzpingus* Sihler, 2021

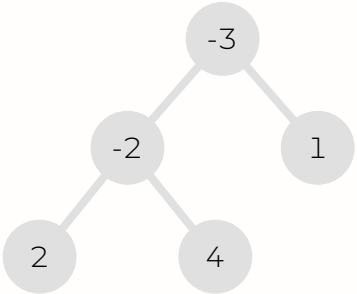
# 3.1 Die Muster-Kinder

# 3.2 Die Muster-Kinder

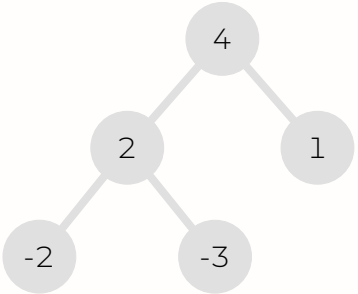


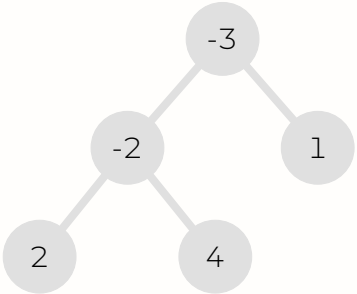
# 3.3 Die Muster-Kinder



# 3.4 Die Muster-Kinder

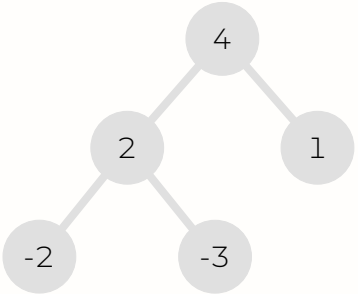



**Max**-Heap

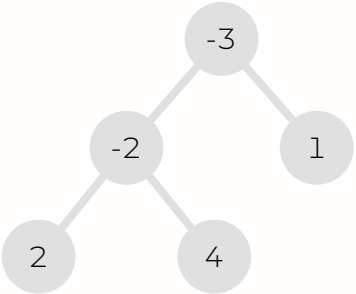
Eltern $\geq$ Kinder

# 3.5 Die Muster-Kinder




**Max**-Heap

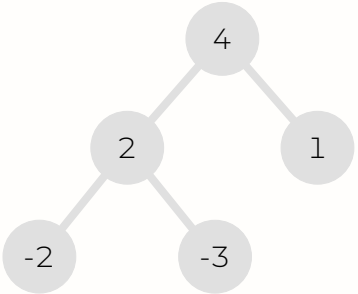
Eltern ≥ Kinder




**Min**-Heap

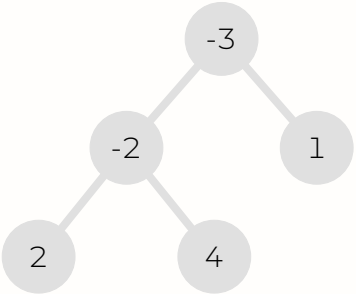
Eltern ≤ Kinder

# 3.6 Die Muster-Kinder




- **Max**-Heap

Eltern ≥ Kinder

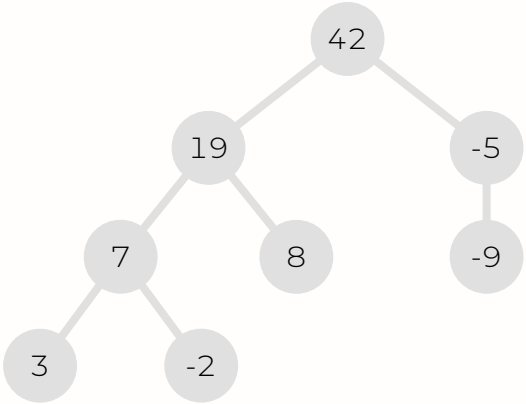



**Min**-Heap

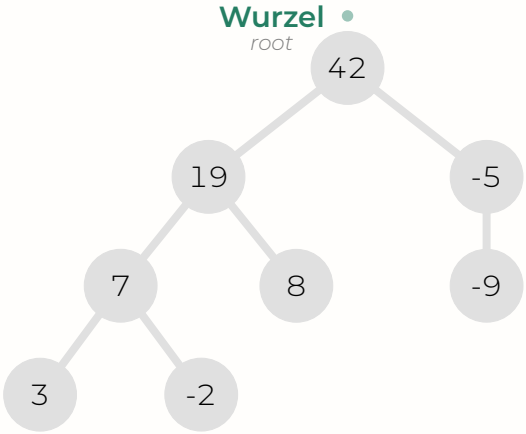
Eltern ≤ Kinder

# 4.1 Die Anatomie

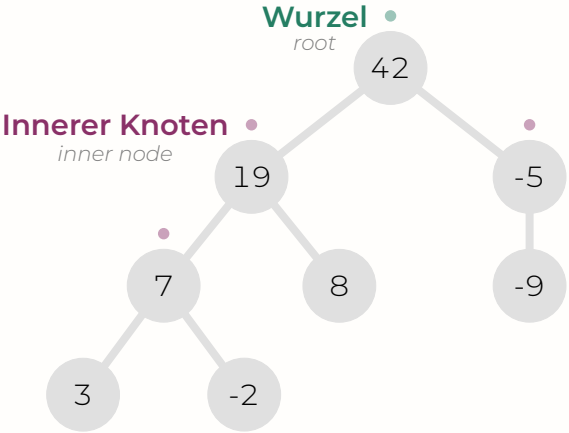
# 4.2 Die Anatomie



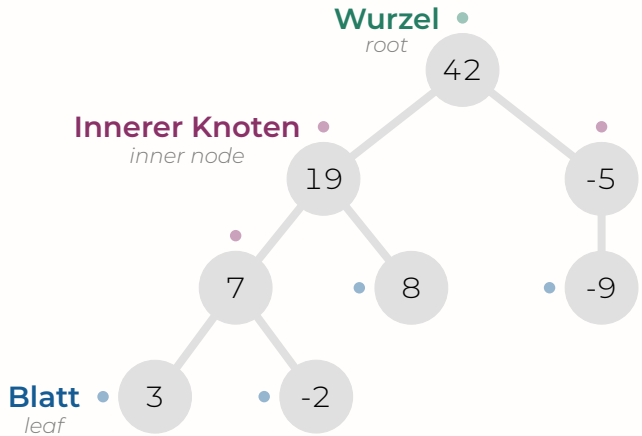
# 4.3 Die Anatomie



# 4.4 Die Anatomie

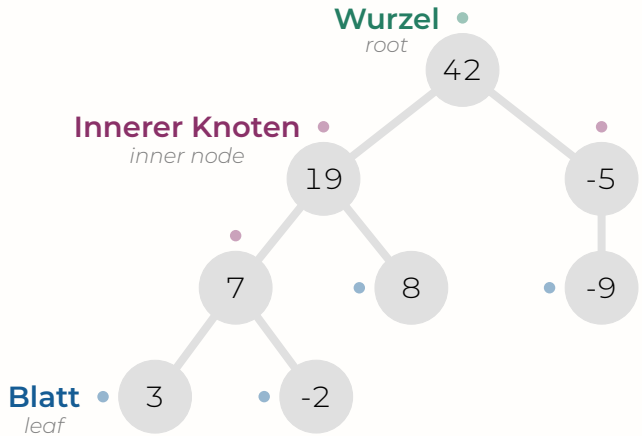


# 4.5 Die Anatomie



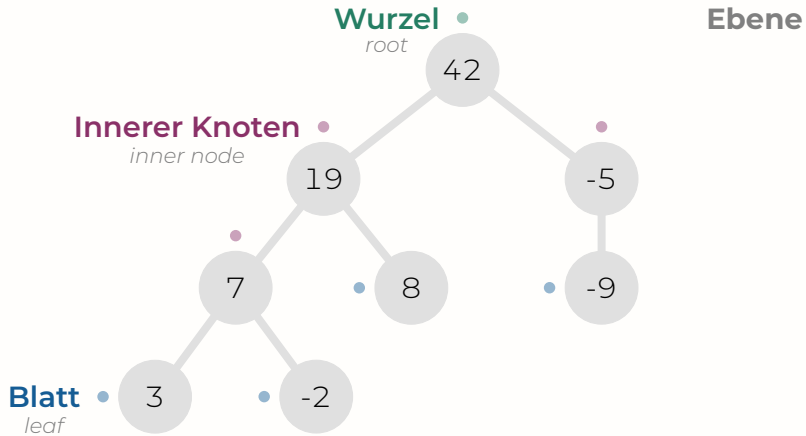
# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.



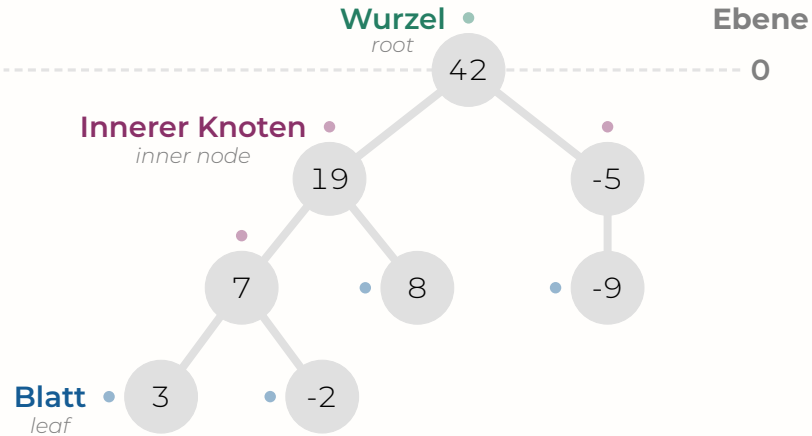
# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.



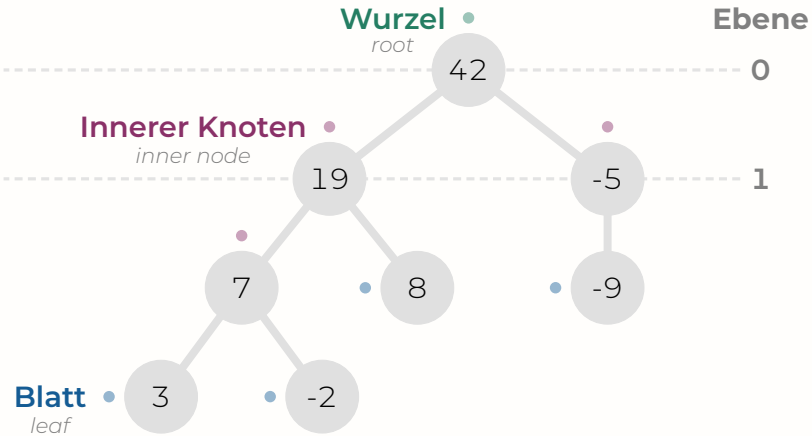
# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.



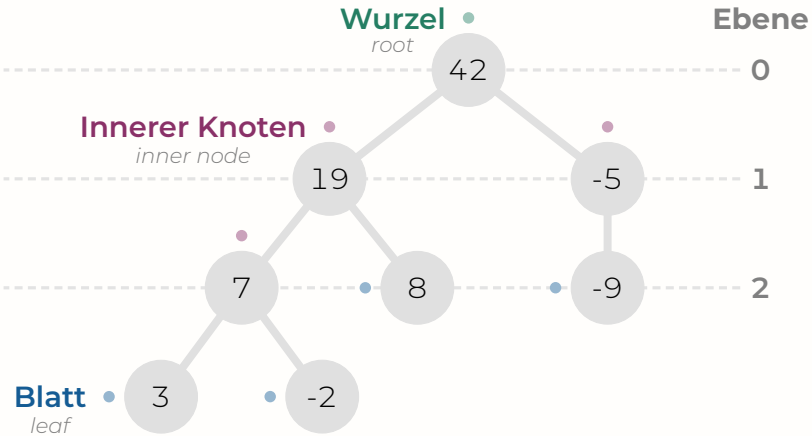
# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.



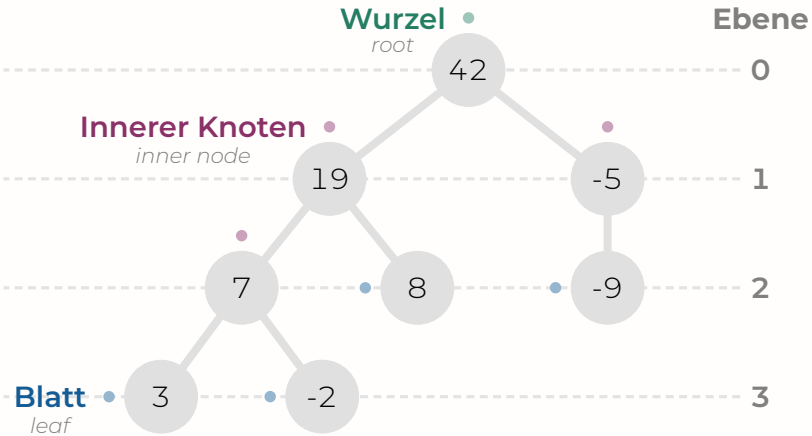
# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.



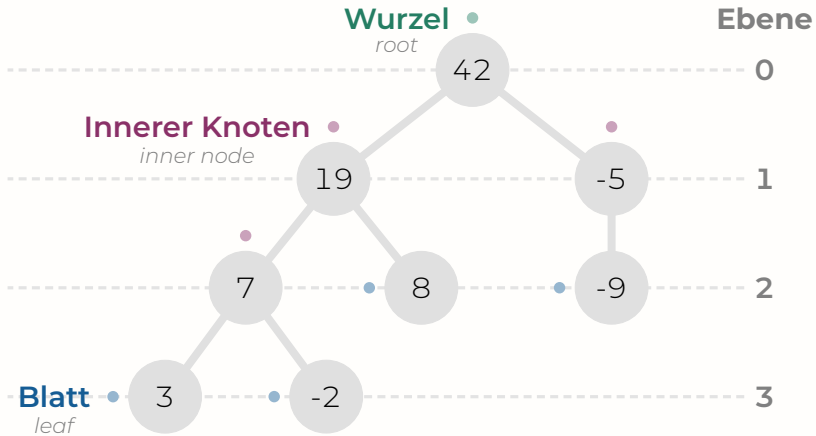
# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.



# Die Anatomie

Blätter sind also Knoten ohne weitere Kindknoten.




Die Linien zwischen den Knoten heißen einfach „Kante“ (*edge*).

# 5.1 Formales

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# 5.2 Formales

- Für uns ein Baum

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# 5.3 Formales

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# Formales

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten
  - $V$ und $E$ sind endlich

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# 5.5 Formales

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten
  - $V$ und $E$ sind endlich
  - Azyklisch & zusammenhängend („maximal azyklisch“)

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# 5.6 Formales

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten
  - $V$ und $E$ sind endlich
  - Azyklisch & zusammenhängend („maximal azyklisch“)
- Sogar ein Binärbaum!

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# 5.7 Formales

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten
  - $V$ und $E$ sind endlich
  - Azyklisch & zusammenhängend („maximal azyklisch“)
- Sogar ein Binärbaum!
  - Jeder Knoten hat maximal zwei Nachfolger

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# Formales

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten
  - $V$ und $E$ sind endlich
  - Azyklisch & zusammenhängend („maximal azyklisch“)
- Sogar ein Binärbaum!
  - Jeder Knoten hat maximal zwei Nachfolger
  - Wir konzentrieren uns auf ausgewogene

[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# Formales

Eine sträflich oberflächliche Betrachtung etlicher cooler formaler Grundlagen.

- Für uns ein Baum
  - Graph $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten
  - $V$ und $E$ sind endlich
  - Azyklisch & zusammenhängend („maximal azyklisch“)
- Sogar ein Binärbaum!
  - Jeder Knoten hat maximal zwei Nachfolger
  - Wir konzentrieren uns auf ausgewogene
- Bäume lassen sich als rekursive Struktur auffassen

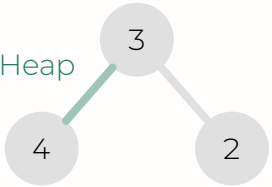
[6]: *Episode-Rekursion* Sihler, 2021

[2]: *The average height of binary trees and other simple trees* Flajolet, 1982

# 6.1 Heapify

# 6.2 Heapify

Max-Heap



# 6.3 Heapify

# Heapify


- *Up*-Heapify: Prüft Blätter → Wurzel

# Heapify


- *Up*-Heapify: Prüft Blätter → Wurzel
- *Down*-Heapify: Prüft Wurzel → Blätter

# Heapify


Herstellung der Heap-Eigenschaft durch Vertauschungen!

- *Up*-Heapify: Prüft Blätter → Wurzel
- *Down*-Heapify: Prüft Wurzel → Blätter

# 7.1 Insert / Put

# 7.2 Insert / Put

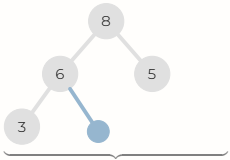
- Auf der niedersten Ebene, so weit links wie möglich

# 7.3 Insert / Put

- Auf der niedersten Ebene, so weit links wie möglich
- Anschließend: *heapify*

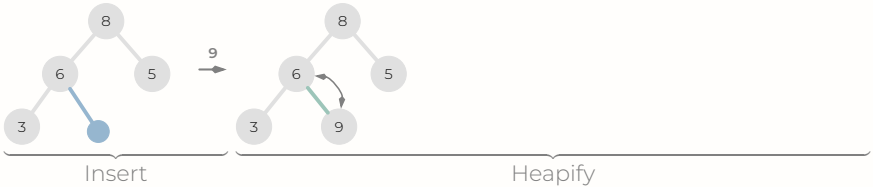
# Insert / Put

- Auf der niedersten Ebene, so weit links wie möglich
- Anschließend: *heapify*



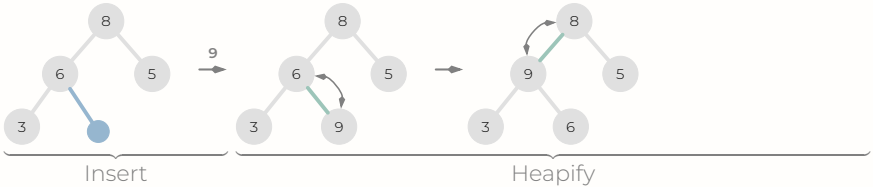
# Insert / Put

- Auf der niedersten Ebene, so weit links wie möglich
- Anschließend: *heapify*



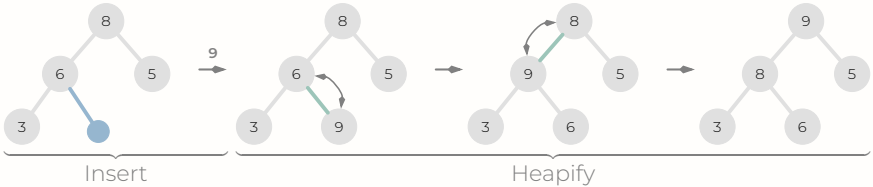
# Insert / Put

- Auf der niedersten Ebene, so weit links wie möglich
- Anschließend: *heapify*



# Insert / Put

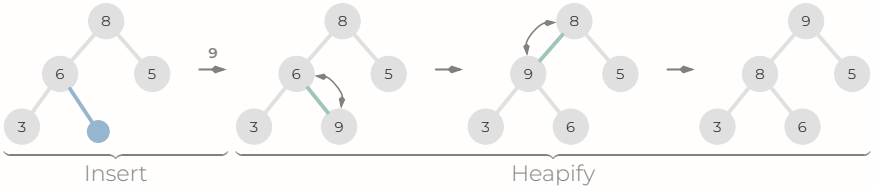
- Auf der niedersten Ebene, so weit links wie möglich
- Anschließend: *heapify*



# Insert / Put

Ist die unterste Ebene voll, so erschaffen wir einfach eine Neue!

- Auf der niedersten Ebene, so weit links wie möglich
- Anschließend: *heapify*



# 8.1 Remove / Get

# 8.2 Remove / Get

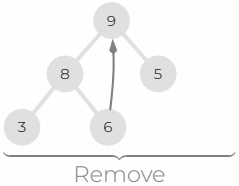
- Ersetze oberstes Element durch „letztes“

# 8.3 Remove / Get

- Ersetze oberstes Element durch „letztes“
- Anschließend: *heapify*

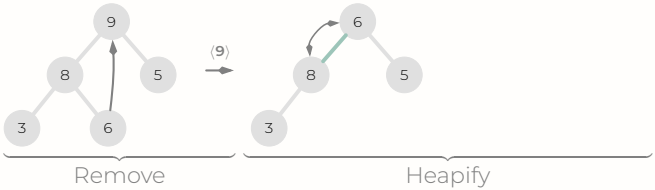
# 8.4 Remove / Get

- Ersetze oberstes Element durch „letztes“
- Anschließend: *heapify*



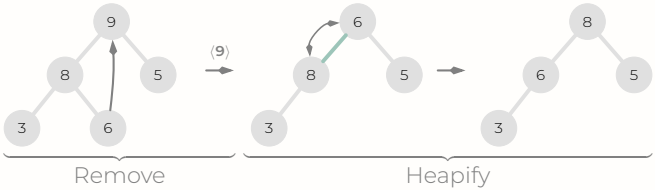
# Remove / Get

- Ersetze oberstes Element durch „letztes“
- Anschließend: *heapify*



# Remove / Get

- Ersetze oberstes Element durch „letztes“
- Anschließend: *heapify*



# 9.1 Weitere Operationen

[3]: *Refined complexity analysis for heap operations* Fredman, 1987

[7]: *Heap : A Data Structure for Efficient Programs* Singh, 2013

# 9.2 Weitere Operationen

- Find / Extract Min / Max

[3]: *Refined complexity analysis for heap operations* Fredman, 1987

[7]: *Heap : A Data Structure for Efficient Programs* Singh, 2013

# 9.3 Weitere Operationen

- Find / Extract Min / Max
- Replace (one node with another)

[3]: *Refined complexity analysis for heap operations* Fredman, 1987

[7]: *Heap : A Data Structure for Efficient Programs* Singh, 2013

# 9.4 Weitere Operationen

- Find / Extract Min / Max
- Replace (one node with another)
- Meld / Join

[3]: *Refined complexity analysis for heap operations* Fredman, 1987

[7]: *Heap : A Data Structure for Efficient Programs* Singh, 2013

# 10.1 Erstaunliche Erkentnisse

# 10.2 Erstaunliche Erkentnisse

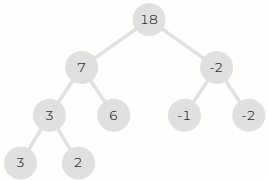
*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

# 10.3 Erstaunliche Erkentnisse

*Annahmen*:

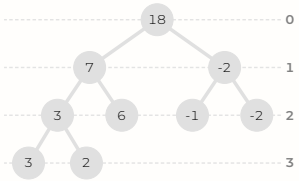
Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.



# 10.4 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

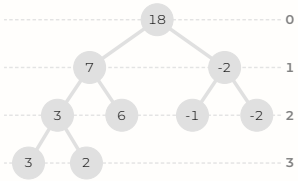


# 10.5 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene *i* enthält $2^i$ Elemente

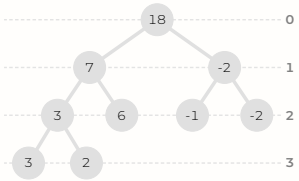


# 10.6 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene *i* enthält $2^i$ Elemente
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!

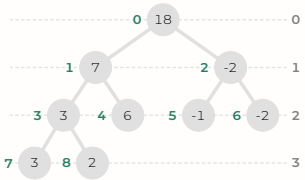


# 10.7 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene *i* enthält $2^i$ Elemente
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!

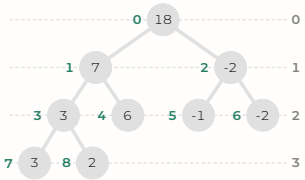


# 10.8 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene $i$ enthält $2^i$ Elemente
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von $n$ ist $2 \cdot n + 1$, das rechte Kind $2 \cdot n + 2$

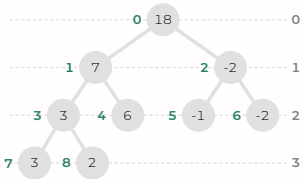


# 10.9 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene $i$ enthält $2^i$ Elemente
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von $n$ ist $2 \cdot n + 1$, das rechte Kind $2 \cdot n + 2$
- Der Elternknoten von $n$ ist $\lfloor \frac{n-1}{2} \rfloor$

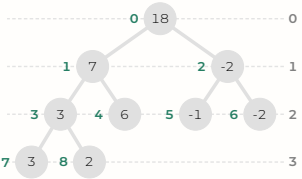


# Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene $i$ enthält $2^i$ Elemente
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von $n$ ist $2 \cdot n + 1$, das rechte Kind $2 \cdot n + 2$
- Der Elternknoten von $n$ ist $\lfloor \frac{n-1}{2} \rfloor$



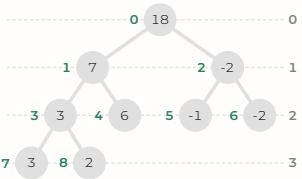

Hmmmm...

# 10.11 Erstaunliche Erkentnisse

*Annahmen*:

Heap ist bis auf letzte Ebene komplett gefüllt.
Letzte Ebene füllt sich von links an.

- Jede gefüllte Ebene $i$ enthält $2^i$ Elemente
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von $n$ ist $2 \cdot n + 1$, das rechte Kind $2 \cdot n + 2$
- Der Elternknoten von $n$ ist $\lfloor \frac{n-1}{2} \rfloor$




Hmmmm… Dat is ne Array-Nummerierung!

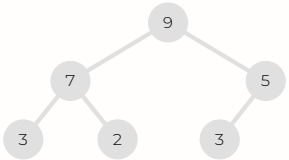
# 11.1 Ein Heap als Array

# 11.2 Ein Heap als Array

- Heaps sind meist „nur“ Arrays

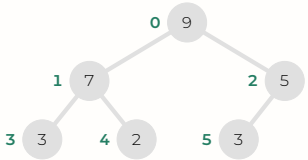
# 11.3 Ein Heap als Array

- Heaps sind meist „nur“ Arrays



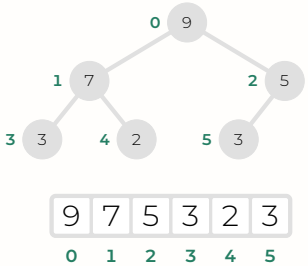
# 11.4 Ein Heap als Array

- Heaps sind meist „nur“ Arrays



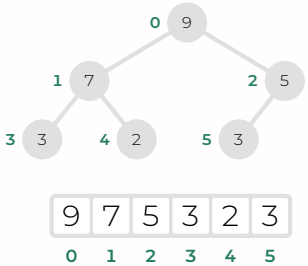
# 11.5 Ein Heap als Array

- Heaps sind meist „nur“ Arrays



# 11.6 Ein Heap als Array

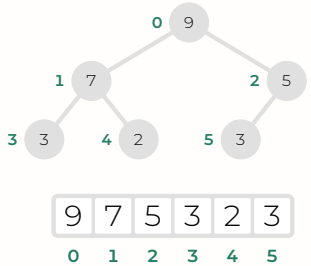
- Heaps sind meist „nur“ Arrays
- Hilfsroutinen erlauben die angenehme Arbeit:



# 11.7 Ein Heap als Array

- Heaps sind meist „nur“ Arrays
- Hilfsroutinen erlauben die angenehme Arbeit:

```
int left(int n) { return 2 * n + 1; }
int right(int n) { return 2 * n + 2; }
int parent(int n) { return (n - 1) / 2; }
```

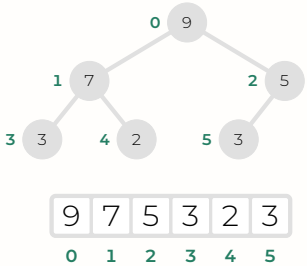


# 11.8 Ein Heap als Array

- Heaps sind meist „nur“ Arrays
- Hilfsroutinen erlauben die angenehme Arbeit:

```
int left(int n) { return 2 * n + 1; }
int right(int n) { return 2 * n + 2; }
int parent(int n) { return (n - 1) / 2; }
```

- Checks schaden natürlich nicht!

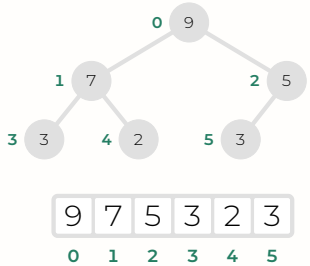


# 11.9 Ein Heap als Array

- Heaps sind meist „nur“ Arrays
- Hilfsroutinen erlauben die angenehme Arbeit:

```
int left(int n) { return 2 * n + 1; }
int right(int n) { return 2 * n + 2; }
int parent(int n) { return (n - 1) / 2; }
```

- Checks schaden natürlich nicht!




Tipp: Bei Arrays ist meist auch *swap* äußerst hilfreich!

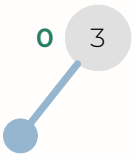
# 12.1 Ein absteigendes Beispiel

# 12.2 Ein absteigendes Beispiel

$$[3, 9, 8, 2, 12]$$

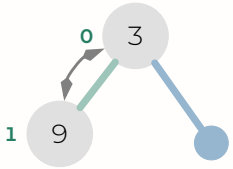
# 12.3 Ein absteigendes Beispiel

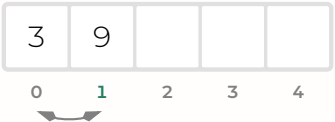
[3, 9, 8, 2, 12]




# 12.4 Ein absteigendes Beispiel

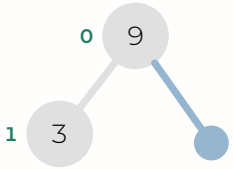
[3, 9, 8, 2, 12]

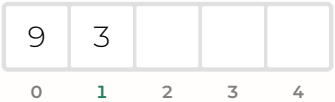


# 12.5 Ein absteigendes Beispiel

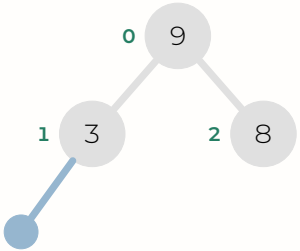
[3, 9, 8, 2, 12]

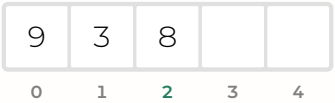


# 12.6 Ein absteigendes Beispiel

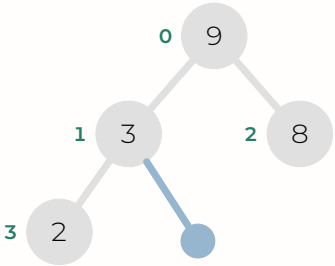
[3, 9, 8, 2, 12]



# 12.7 Ein absteigendes Beispiel

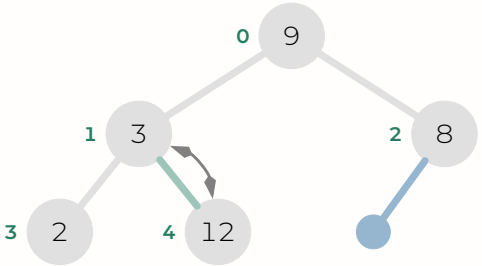
[3, 9, 8, 2, 12]

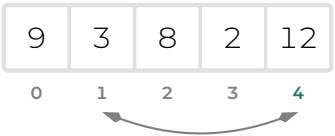



| 9 | 3 | 8 | 2 | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

# 12.8 Ein absteigendes Beispiel

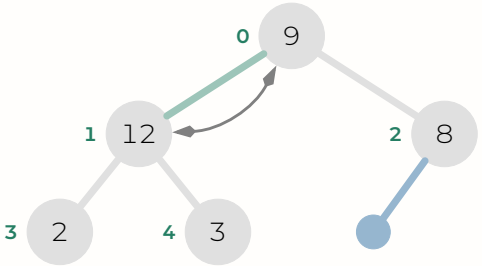
$[3, 9, 8, 2, 12]$



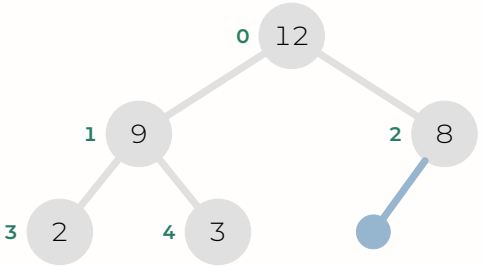
# 12.9 Ein absteigendes Beispiel

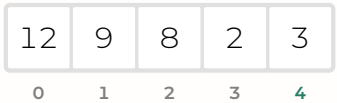
[3, 9, 8, 2, 12]




# Ein absteigendes Beispiel

[3, 9, 8, 2, 12]

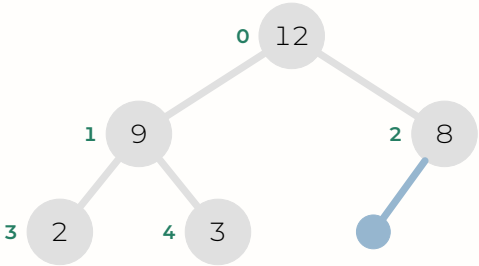


# Ein absteigendes Beispiel

Konstruiert man den Heap nur durch *insert*, verletzt stets max. das neue Element die Heap-Eigenschaft. Weiter ist er schön ausgewogen.

$[3, 9, 8, 2, 12]$

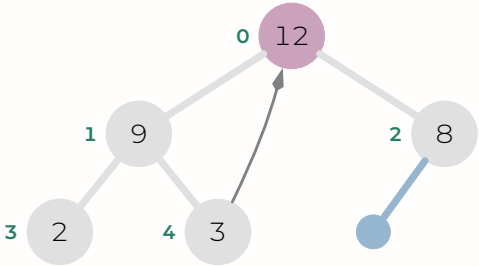


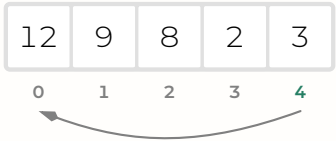

| 12 | 9 | 8 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

# Ein absteigendes Beispiel

Das Entfernen zur Abwechslung nun mit „down-heapify“.

$$[3, 9, 8, 2, 12]$$

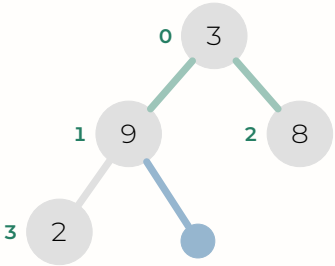


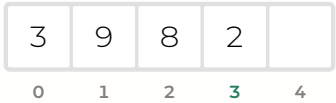



# Ein absteigendes Beispiel

Das Entfernen zur Abwechslung nun mit „down-heapify“.
Bei mehreren Konflikten: tausche mit *extremerem* Element.

[3, 9, 8, 2, 12]



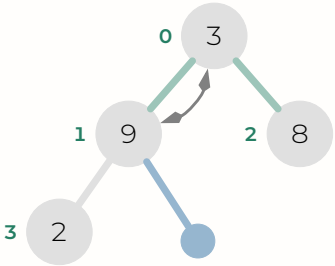



# Ein absteigendes Beispiel

Das Entfernen zur Abwechslung nun mit „down-heapify“.

Bei mehreren Konflikten: tausche mit *extremerem* Element.

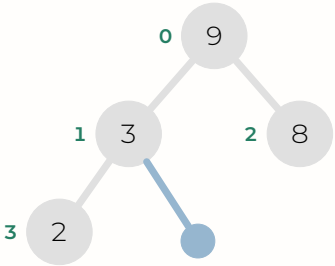
$$[3, 9, 8, 2, 12]$$




# Ein absteigendes Beispiel

Das Entfernen zur Abwechslung nun mit „down-heapify“.
Bei mehreren Konflikten: tausche mit *extremerem* Element.

$$[3, 9, 8, 2, 12]$$




| 9 | 3 | 8 | 2 | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

# 13.1 Zur Übung

[4]: *Suchbäume*
GeeksforGeeks, 2020

[1]: *Exercises for Algorithms and Data Structures* Carzaniga, 2016

# 13.2 Zur Übung

- Visualisiere *insert* $[3, 5, 8, 3, -5, 6]$ (in der Reihenfolge) im *Min*-Heap. Am Besten simultan im Baum und im Array.

[4]: *Suchbäume* GeeksforGeeks, 2020

[1]: *Exercises for Algorithms and Data Structures* Carzaniga, 2016

# 13.3 Zur Übung

- Visualisiere *insert* $[3, 5, 8, 3, -5, 6]$ (in der Reihenfolge) im *Min*-Heap. Am Besten simultan im Baum und im Array.
- Visualisiere nun das Entfernen, bis der Heap leer ist

[4]: *Suchbäume* GeeksforGeeks, 2020

[1]: *Exercises for Algorithms and Data Structures* Carzaniga, 2016

# 13.4 Zur Übung

- Visualisiere *insert* $[3, 5, 8, 3, -5, 6]$ (in der Reihenfolge) im *Min*-Heap. Am Besten simultan im Baum und im Array.
- Visualisiere nun das Entfernen, bis der Heap leer ist

- Bei diesen und anderen Datenstrukturen hilft die *praktische* Auseinandersetzung!

[4]: *Suchbäume* GeeksforGeeks, 2020

[1]: *Exercises for Algorithms and Data Structures* Carzaniga, 2016

[1] Antonio Carzaniga. „Exercises for Algorithms and Data Structures“. 2016

[2] Philippe Flajolet und Andrew Odlyzko. „The average height of binary trees and other simple trees“. 1982

[3] Michael L. Fredman und Thomas H. Spencer. „Refined complexity analysis for heap operations“. 1987

[4] GeeksforGeeks. *Suchbäume*. 2020

[5] Florian Sihler. *LaTeX-Package, tikzpingus*. 2021

[6] Florian Sihler. *Episode-Rekursion*. 2021

[7] Sanjay Singh. „Heap : A Data Structure for Efficient Programs“. 2013



**Florian Sihler**

Ulm, 11. November 2022


Slides and LaTeX-sources on GitHub!