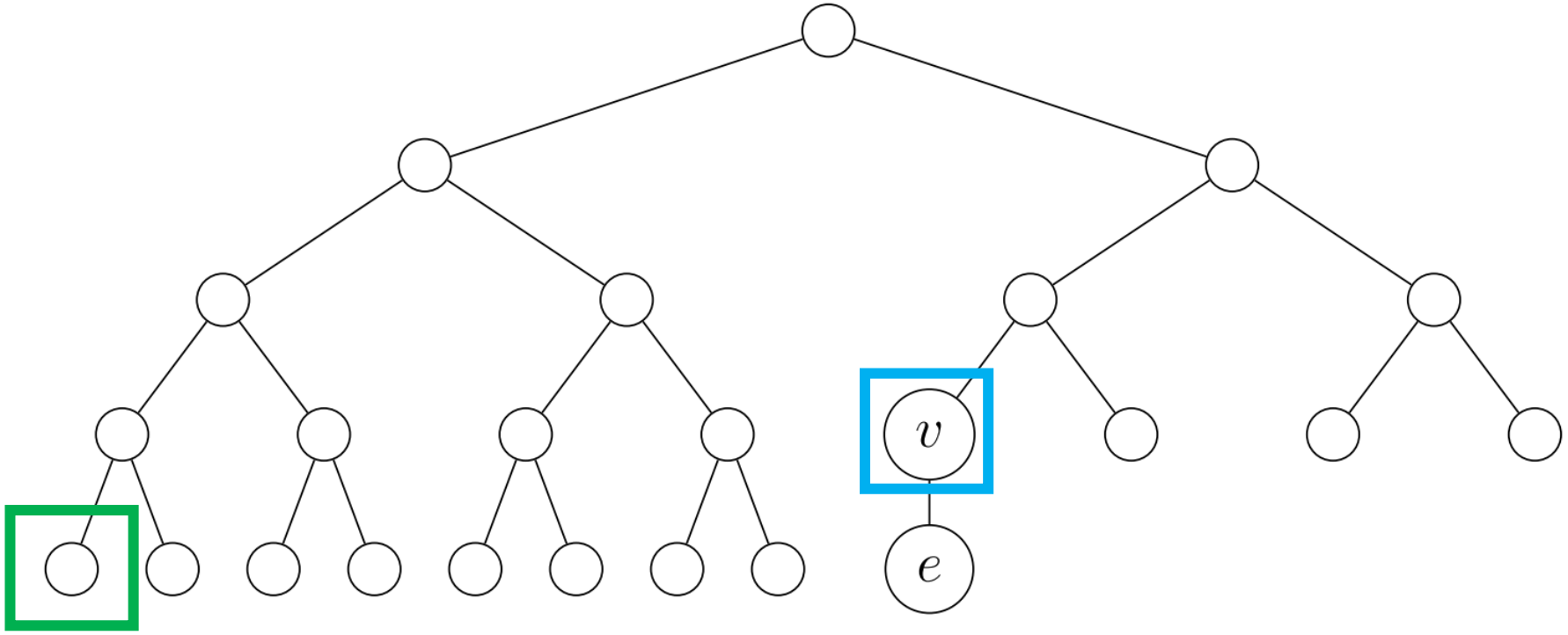
Tutorium
Grundlagen: Algorithmen und Datenstrukturen

Übungsblatt Woche 9

Wiederholung: Definitionen

Keine Kinder = Blatt (für uns auch Wurzel)

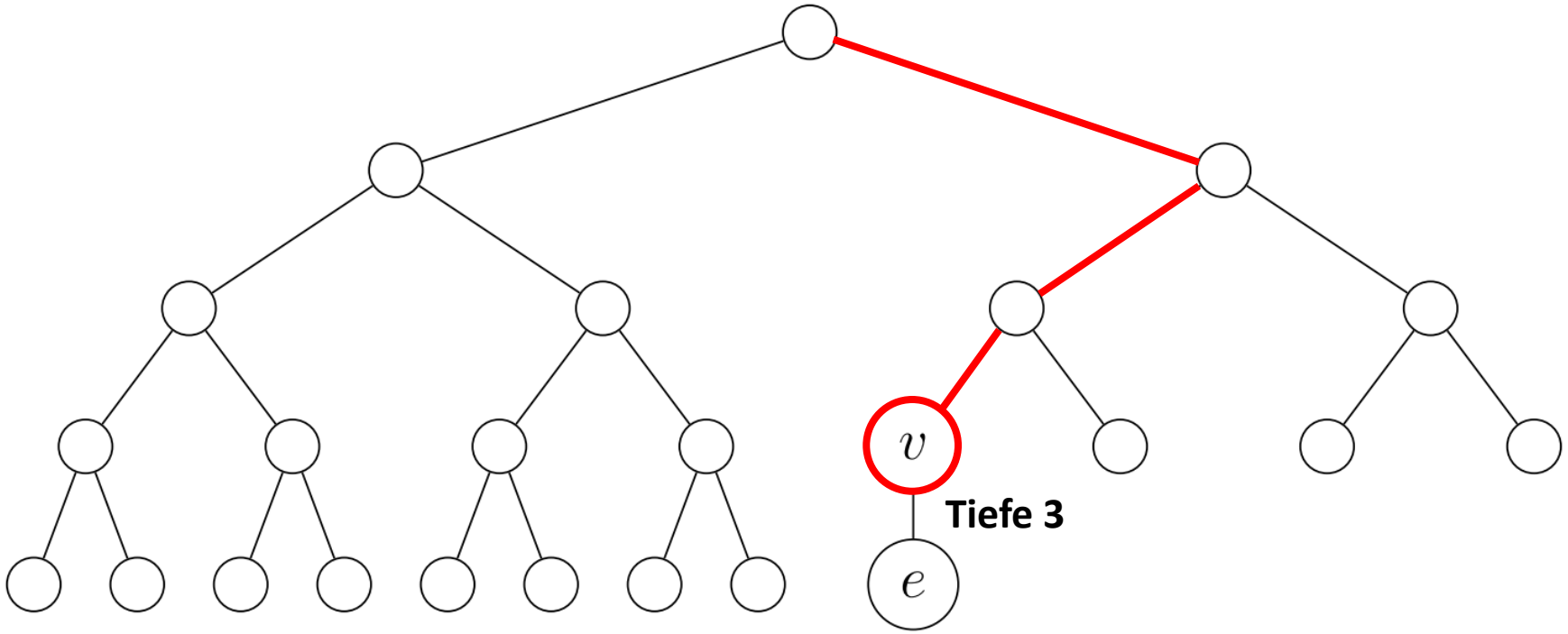
Sonst = innerer Knoten



Wiederholung: Definitionen

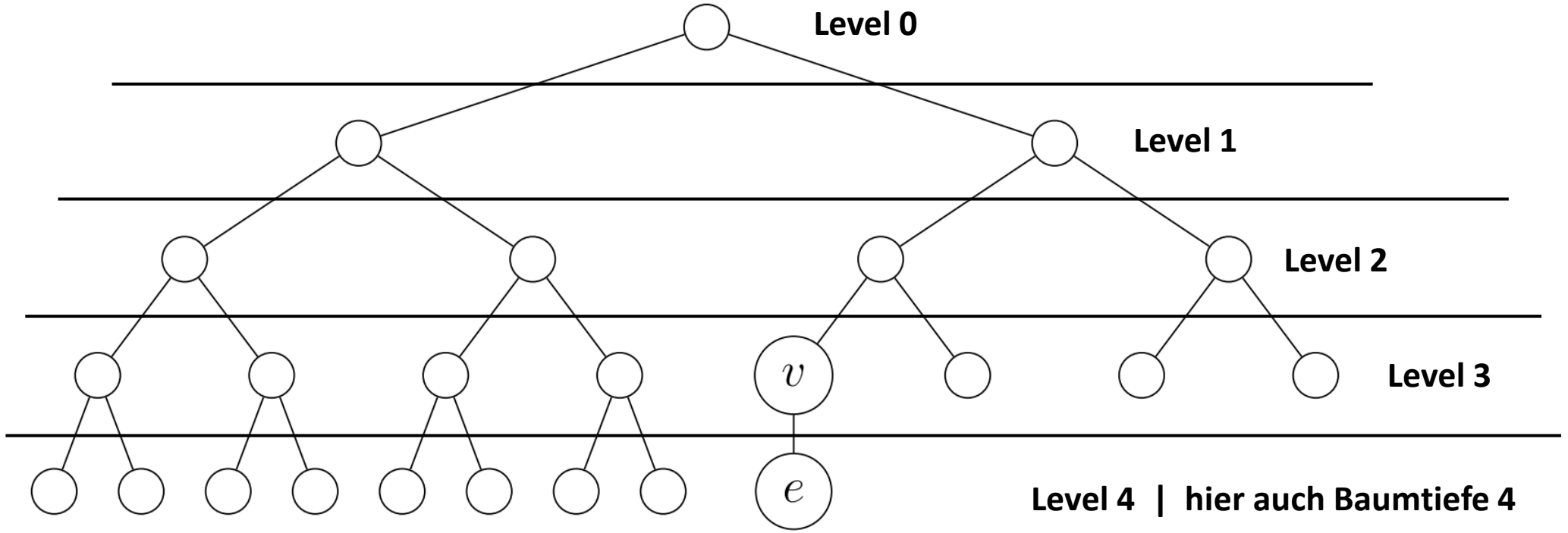
Tiefe eines Knotens = Entfernung von Wurzel gemessen in Kanten
(insbesondere hat die Wurzel Tiefe 0)

Achtung: Unterschiede in Literatur!!!



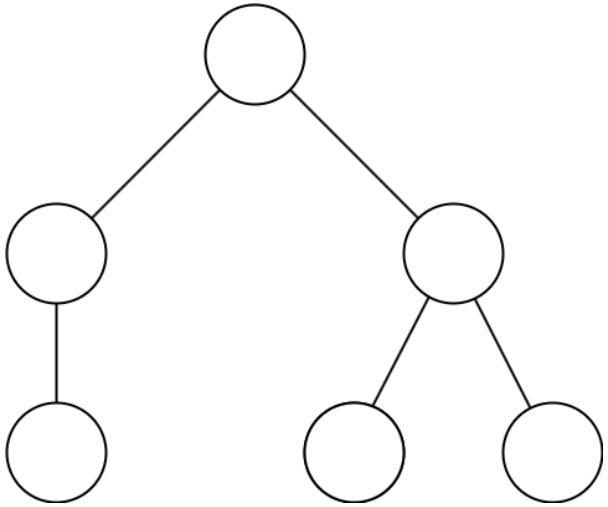
Wiederholung: Definitionen

Tiefe eines Baumes = Höchste Knotentiefe im Baum
i-tes Level eines Baumes = Menge aller Knoten mit Tiefe i

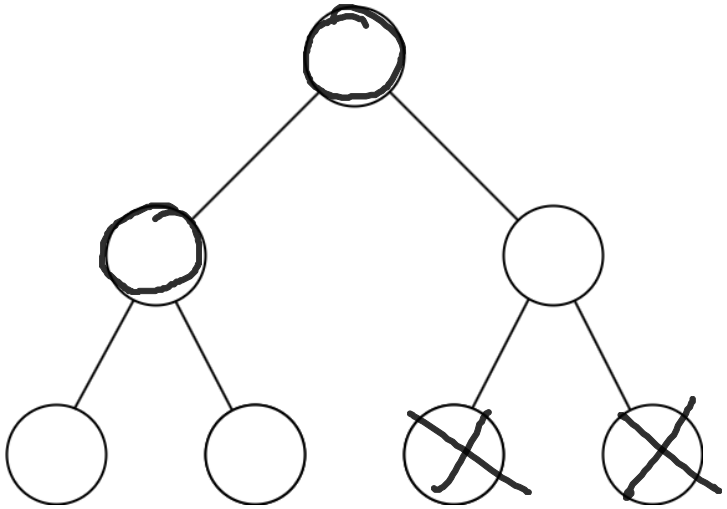


Wiederholung: Definitionen

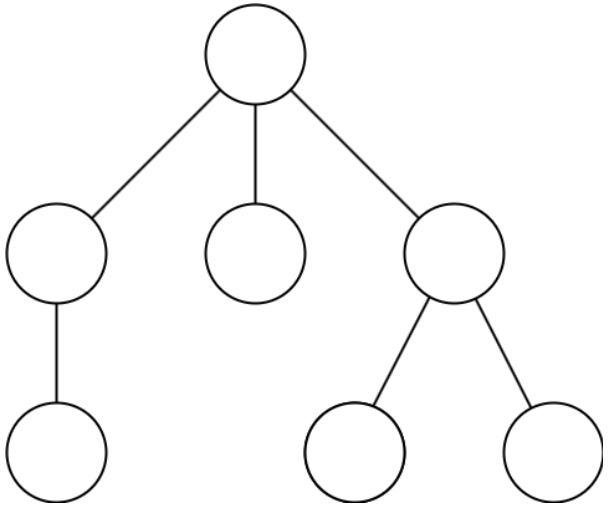
Binärbaum = Jeder innere Knoten hat ***maximal*** 2 Kinder
Echter Binärbaum = Jeder innere Knoten hat ***genau*** 2 Kinder



Binärbaum



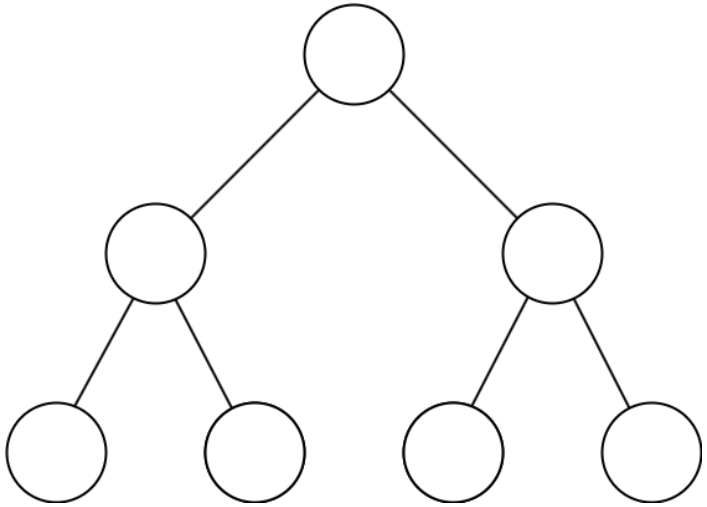
Echter Binärbaum



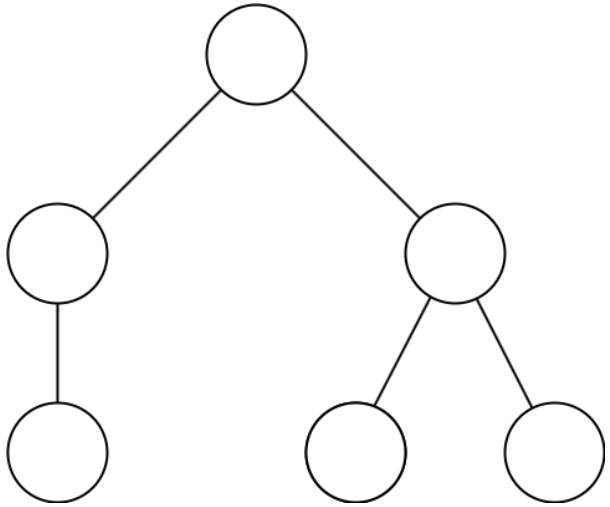
Kein Binärbaum

Wiederholung: Definitionen

Vollständiger Binärbaum = Alle Level des Baumes sind vollständig gefüllt
Fast vollständiger Binärbaum = Alle bis auf das tiefste Level des Baumes sind vollständig gefüllt und Umordnung ist möglich



Vollständiger Binärbaum

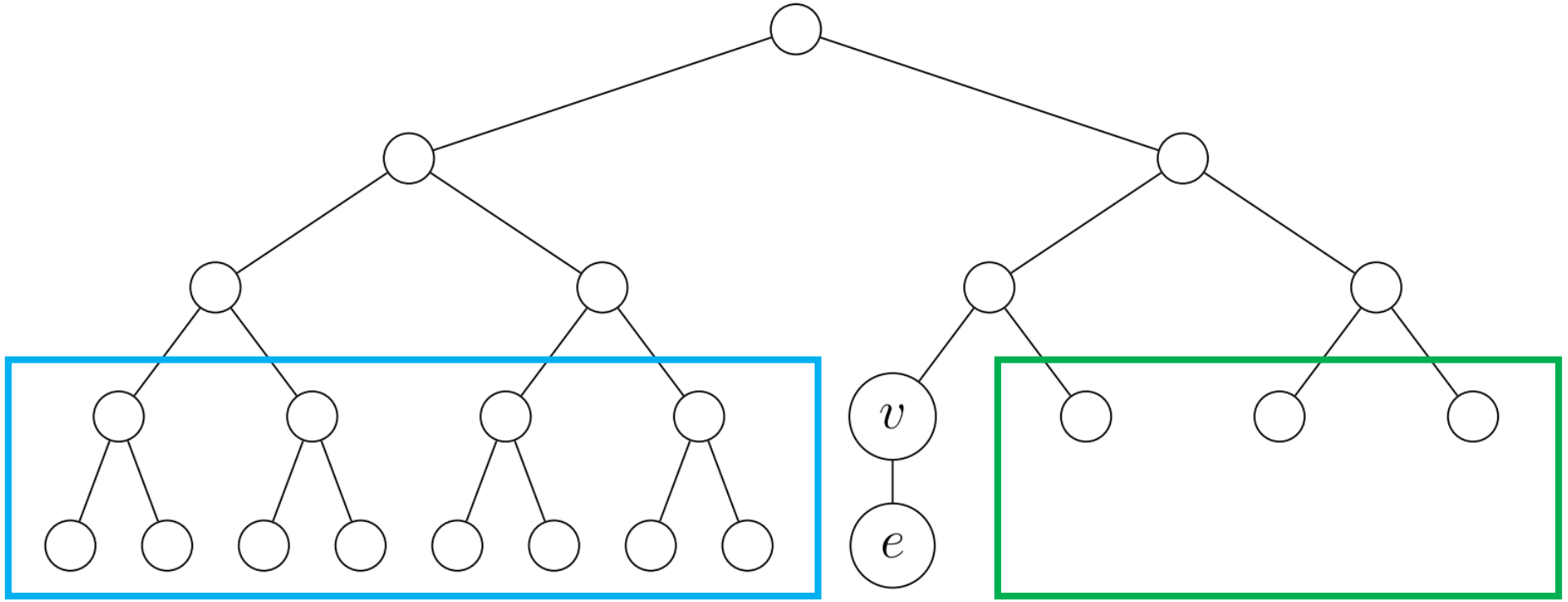


Fast vollständiger Binärbaum

Wiederholung: Definitionen

Vollständiger Binärbaum = Alle Level des Baumes sind vollständig gefüllt

Fast vollständiger Binärbaum = Alle bis auf das tiefste Level des Baumes sind vollständig gefüllt und Umordnung ist möglich



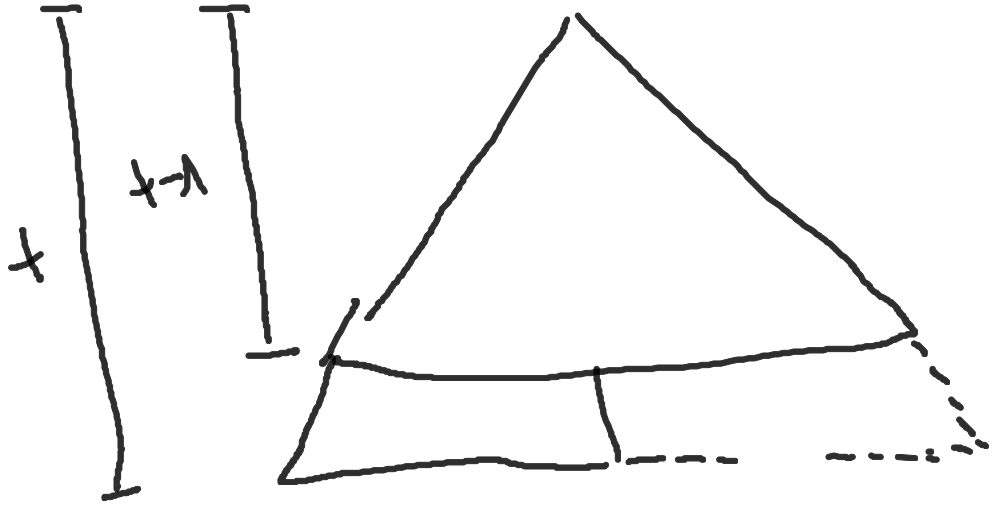
2 Kinder pro Knoten

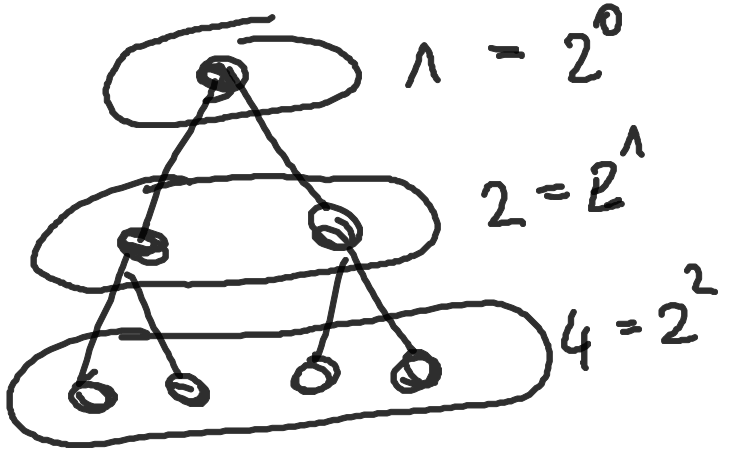
Keine Kindknoten

Aufgabe 9.1 – Vollständig

In der Vorlesung wurde die folgende Aussage verwendet: Jeder fast vollständige Binärbaum mit $n \geq 1$ Knoten und Baumtiefe $t \geq 0$ erfüllt $2^t \leq n \leq 2^{t+1} - 1$.

Beweisen Sie diese Aussage.





$$\sum_{i=0}^{t} 2^i = 2^{t+1} - 1$$

$$\sum_{i=0}^{t-1} 2^i = 2^t - 1$$

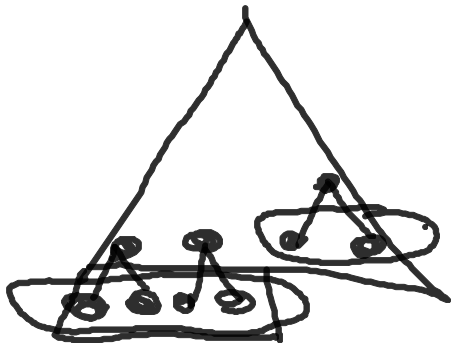
$$2^t - 1 < n \leq 2^{t+1} - 1$$

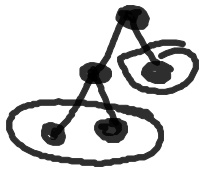
$$2^t \leq n \leq 2^{t+1} - 1$$

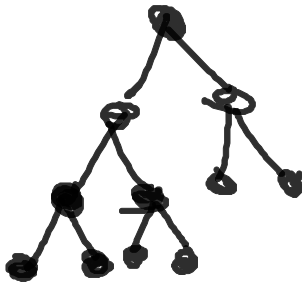
Aufgabe 9.1 – Vollständig

Aufgabe 9.2 – Binärbaum mit b-Blättern

Zeigen Sie, dass es für jedes $b \geq 1$ einen fast vollständigen echten Binärbaum mit b Blättern gibt.







$b = 2^i \Rightarrow t = \log_2(b)$

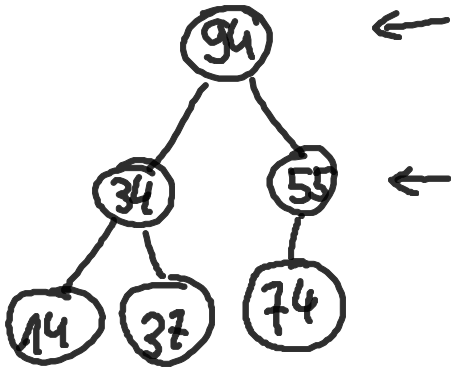
- Vollständiger Baum mit Tiefe t

$t = \lfloor \log_2(b) \rfloor$

- Vollständiger Baum
  - ↳ Hänge von links jeweils 2 Blätter an erhöht Blattzahl um 1 bis b erreicht

Aufgabe 9.3 – Heapsort

Prioritätswarteschlangen, wie beispielsweise durch Heaps implementiert, lassen sich auch problemlos zum Sortieren von Sequenzen verwenden. Insbesondere haben wir in der Vorlesung gelernt, dass sich binäre Heaps ohne zusätzlichen Speicherbedarf gut auf einem Feld umsetzen lassen. Dabei muss allerdings beachtet werden, dass die Funktion `deleteMin()` das jeweils kleinste Element zuerst mit dem letzten vertauscht, und dann aus dem Heap entfernt (bevor die Heap-Invariante per `siftDown` wiederhergestellt wird). Das sorgt dafür, dass man eine Sequenz, die in einem Feld gespeichert ist, mit einem normalen min-Heap *absteigend* sortiert, statt wie sonst *aufsteigend*.

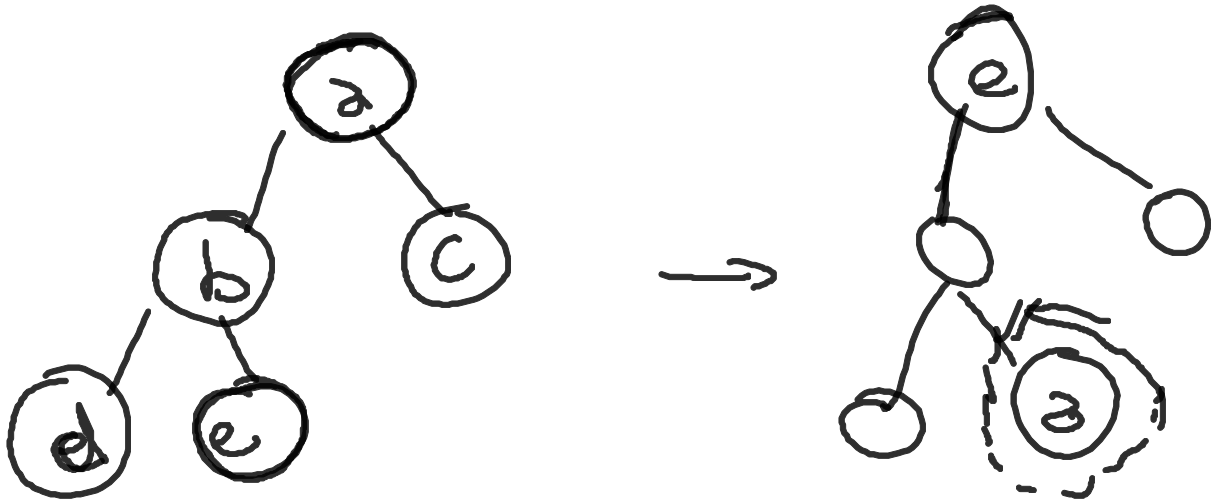


Gegeben ist die folgende Sequenz, die bereits so in einem Feld im Speicher vorliegt:

```
[94, 34, 55, 14, 37, 74]
```

Sie können davon ausgehen, dass jede Zahl immer genau ein Speicherfeld belegt. Sortieren Sie das Feld nun wie gewohnt *aufsteigend* (also mit dem kleinsten Element an erster Stelle), indem Sie Heapsort auf Basis eines Max-Heaps ausführen. Ein Max-Heap verhält sich genauso wie ein Min-Heap, mit dem Unterschied, dass die Priorität eines Knotens immer *größer oder gleich* seiner Kindknoten sein muss.

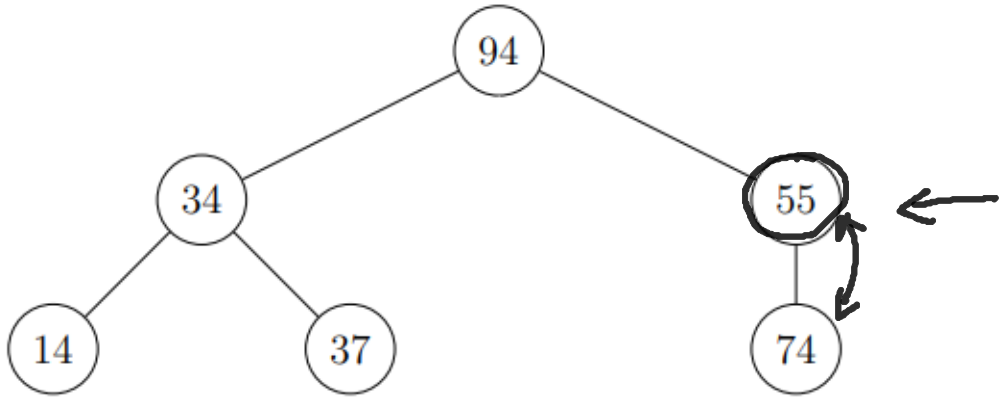
Aufgabe 9.3 – Heapsort



Aufgabe 9.3 – Heapsort

a) Führen Sie zunächst die Operation `build` auf dem Feld aus, und stellen Sie die Max-Heap-Invariante sicher, indem Sie passende `siftDown`-Operationen ausführen.

`[94, 34, 55, 14, 37, 74]`

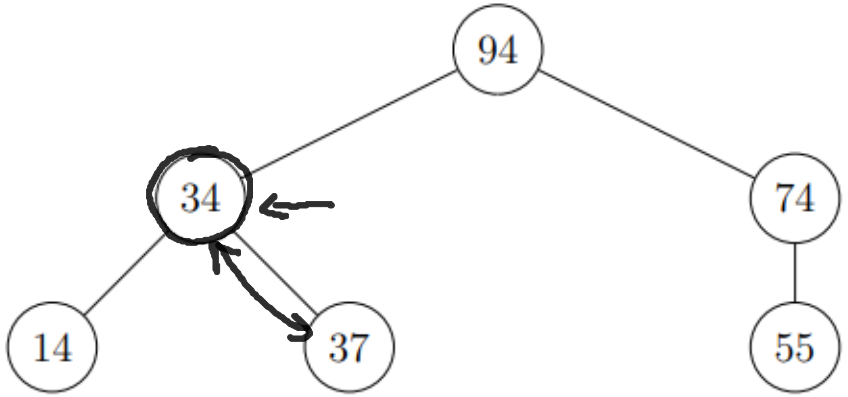


Aufgabe 9.3 – Heapsort

a) Führen Sie zunächst die Operation `build` auf dem Feld aus, und stellen Sie die Max-Heap-Invariante sicher, indem Sie passende `siftDown`-Operationen ausführen.

`[94, 34, 55, 14, 37, 74]`

- `siftDown(55)`

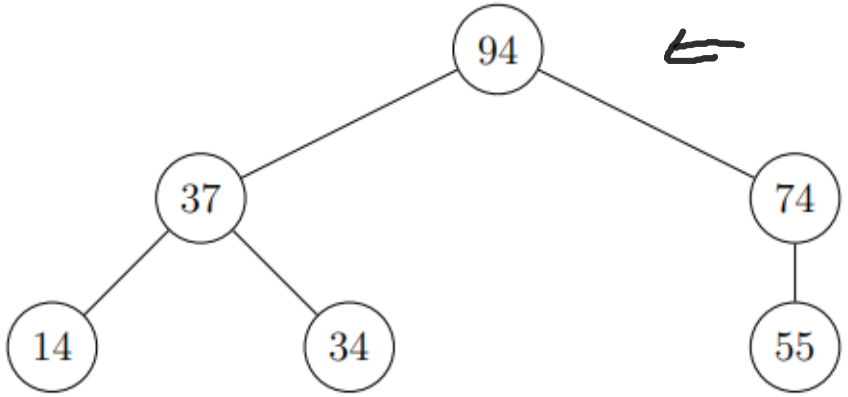


Aufgabe 9.3 – Heapsort

a) Führen Sie zunächst die Operation `build` auf dem Feld aus, und stellen Sie die Max-Heap-Invariante sicher, indem Sie passende `siftDown`-Operationen ausführen.

`[94, 34, 55, 14, 37, 74]`

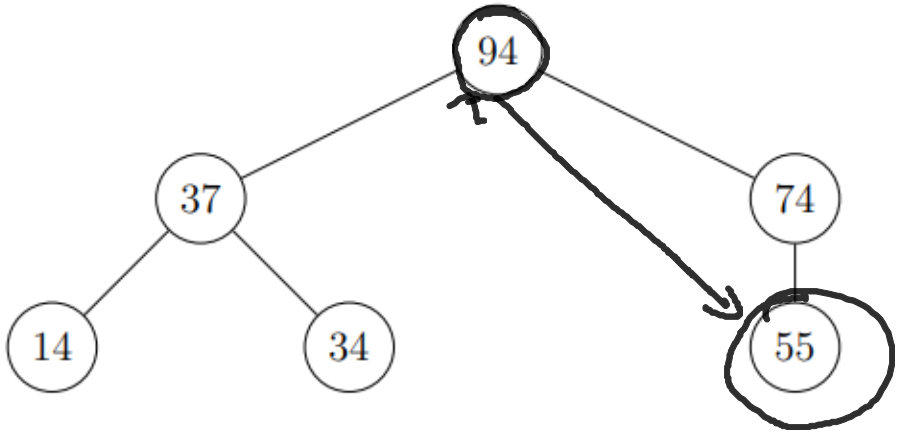
- `siftDown(34)`



Aufgabe 9.3 – Heapsort

a) Führen Sie zunächst die Operation `build` auf dem Feld aus, und stellen Sie die Max-Heap-Invariante sicher, indem Sie passende `siftDown`-Operationen ausführen.

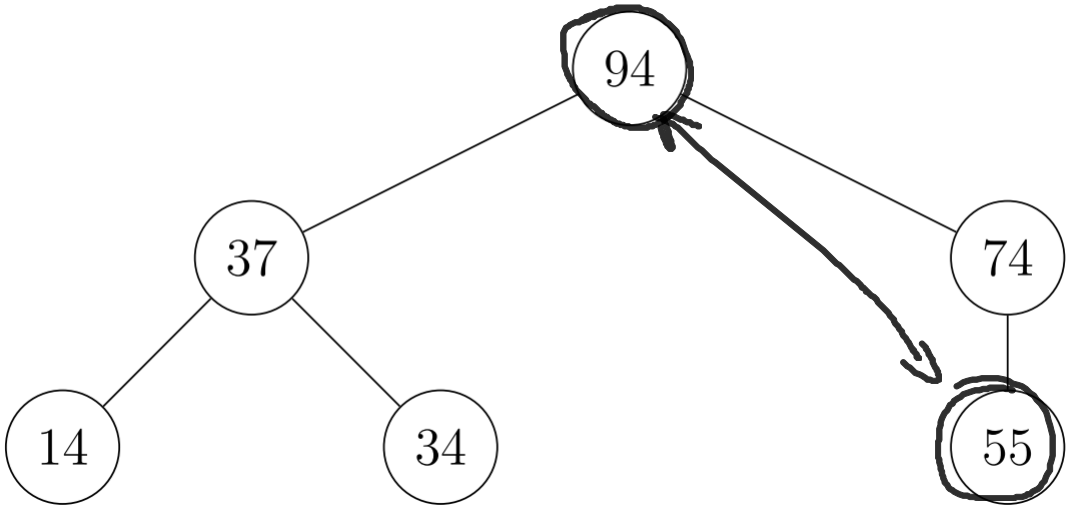
`[94, 34, 55, 14, 37, 74]`



- `siftDown(94)`: Es ändert sich nichts.

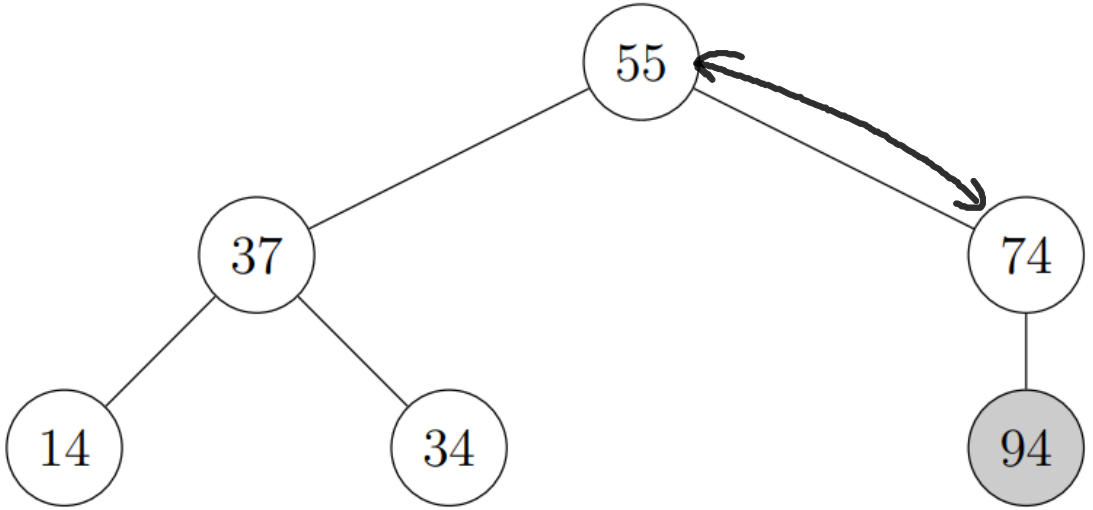
Aufgabe 9.3 – Heapsort

b) Führen Sie nun so lange `deleteMax` aus, bis kein Element mehr im Heap ist. Vertauschen Sie dazu wie gelernt das erste (hier: maximale!) Element mit dem letzten im Heap. Anstatt nun das letzte Element zu löschen, ignorieren Sie für alle weiteren Operationen den Speicher von diesem und allen nachfolgenden Elementen. Mit dieser Einschränkung können Sie nun wie gewohnt `siftDown` auf dem obersten Element ausführen, bis die Heap-Invariante wiederhergestellt wurde.



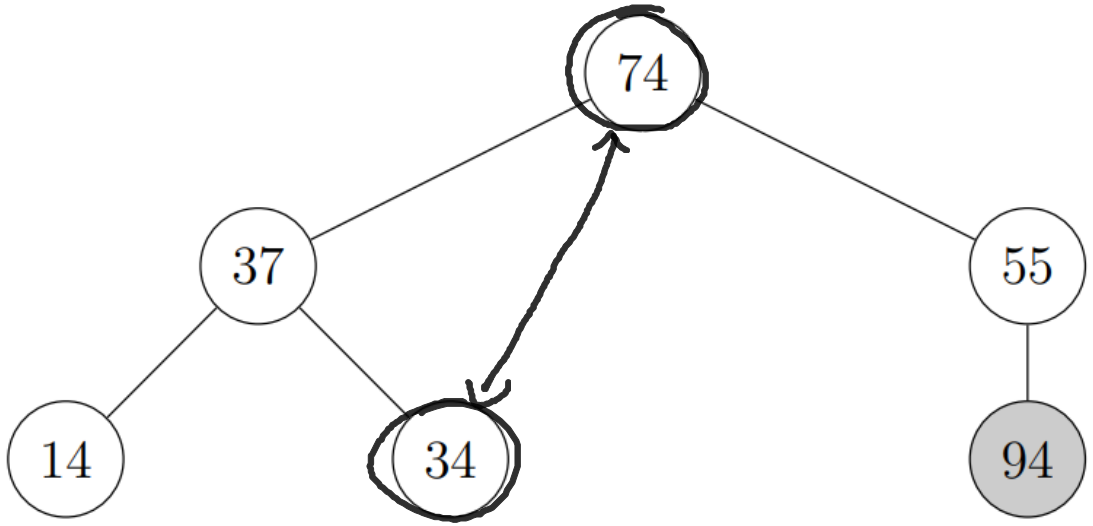
Aufgabe 9.3 – Heapsort

- Vertauschen von 94 mit letzten Element 55, gefolgt von `deleteMax(94)`:



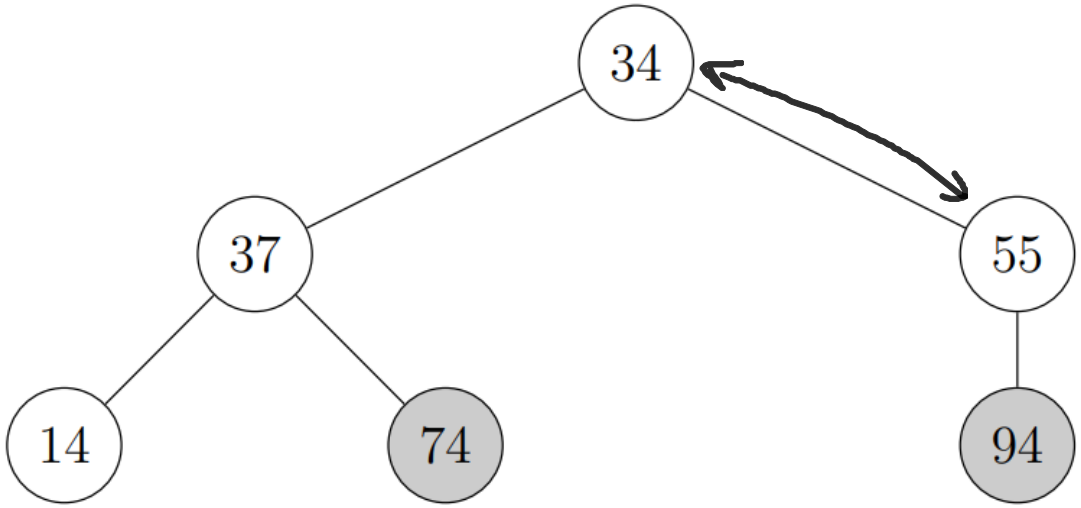
Aufgabe 9.3 – Heapsort

- `siftDown(55)`:



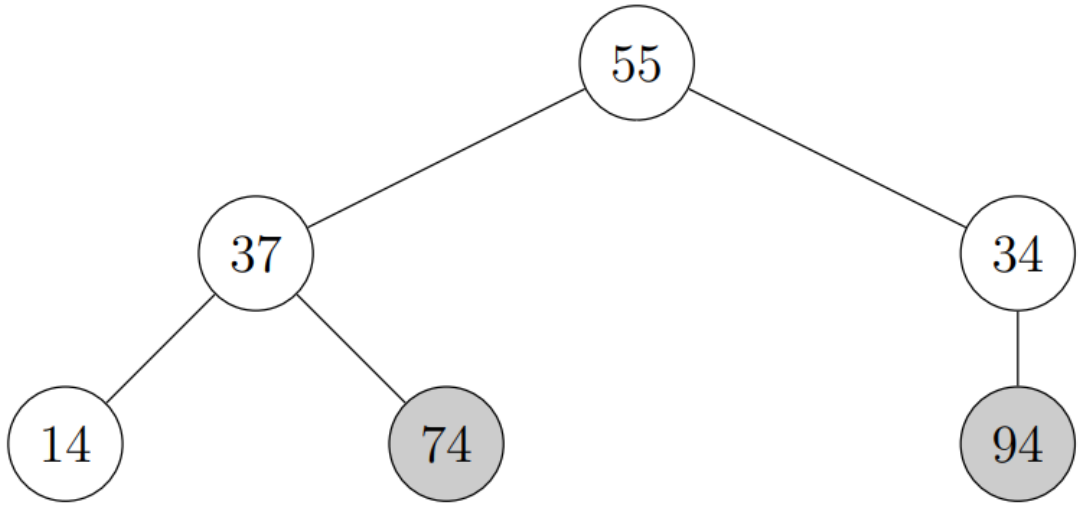
Aufgabe 9.3 – Heapsort

- Vertauschen von 74 mit letzten Element 34, gefolgt von `deleteMax(74)`:



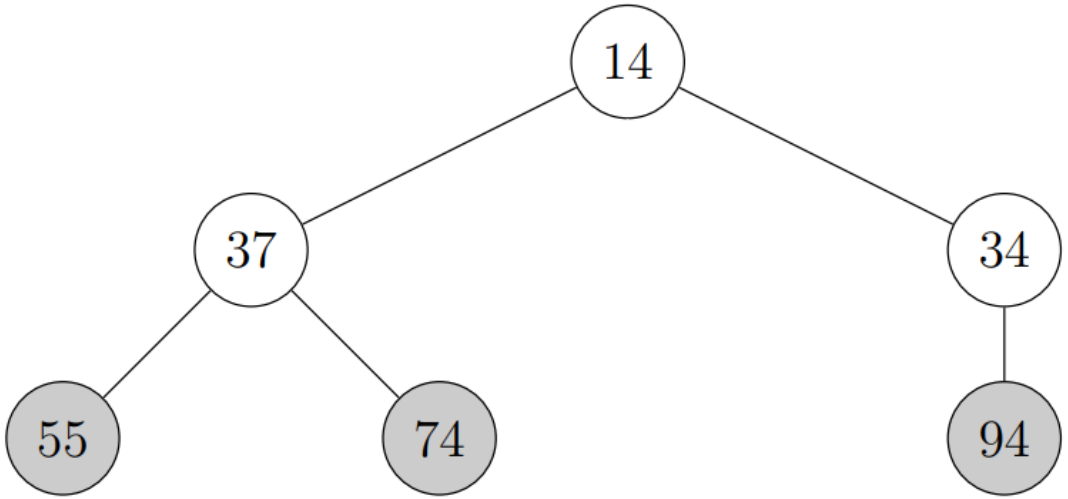
Aufgabe 9.3 – Heapsort

- `siftDown(34)`:



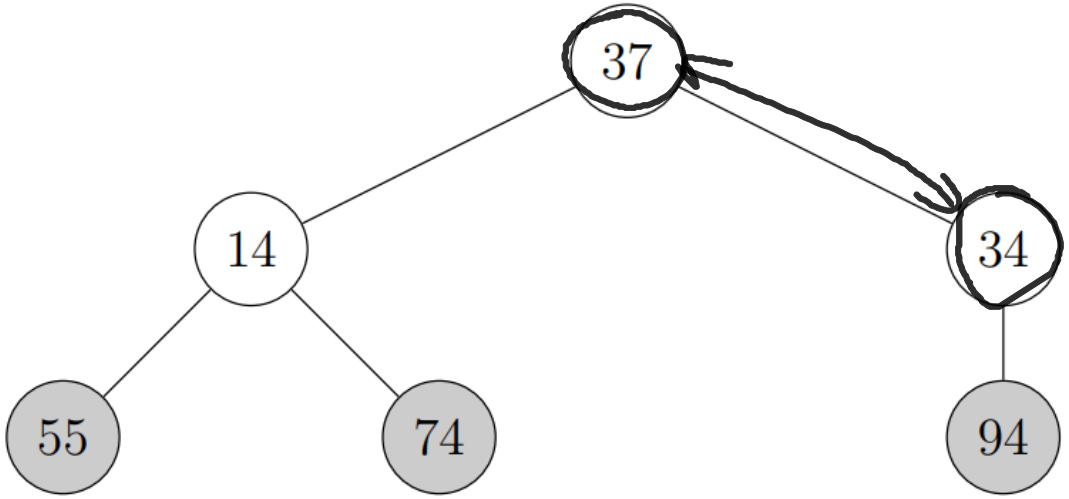
Aufgabe 9.3 – Heapsort

- Vertauschen von 55 mit letzten Element 14, gefolgt von `deleteMax(55)`:



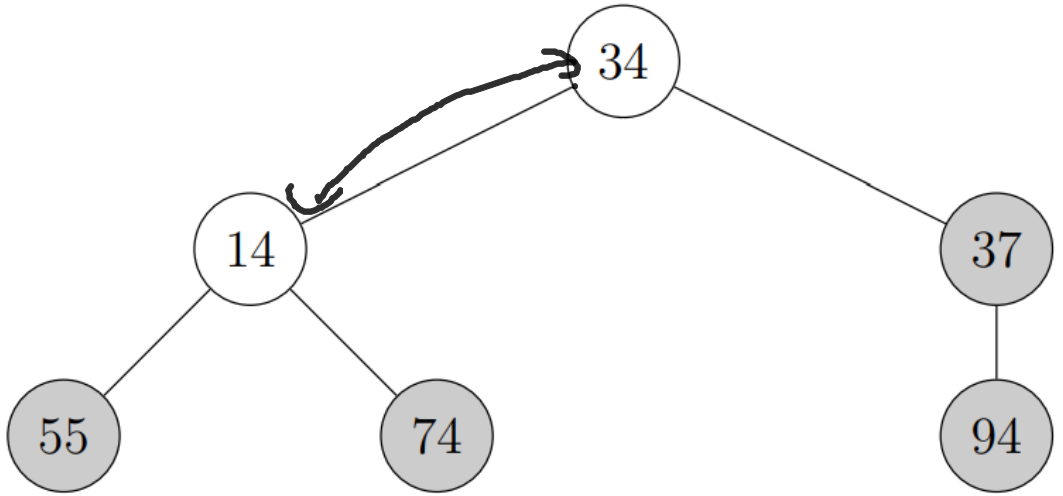
Aufgabe 9.3 – Heapsort

- `siftDown(14)`:



Aufgabe 9.3 – Heapsort

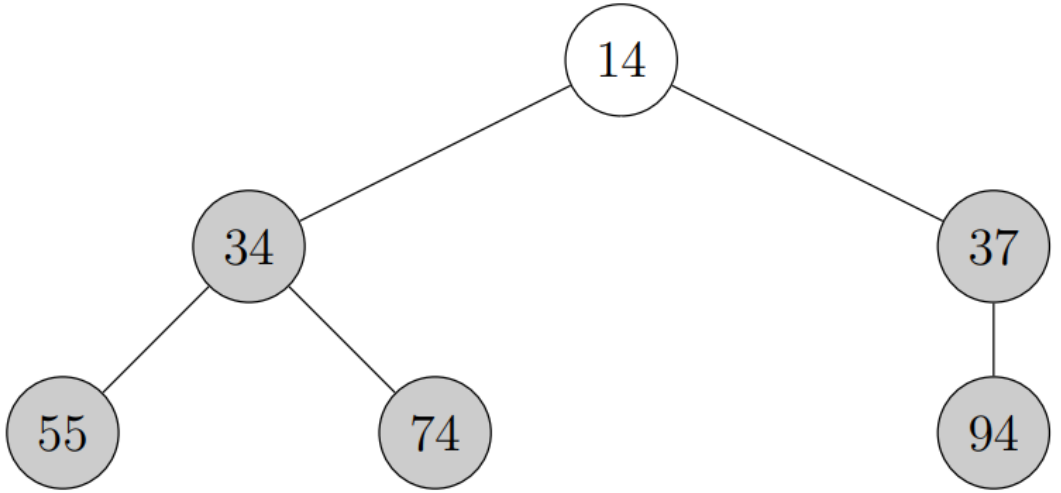
- Vertauschen von 37 mit letzten Element 34, gefolgt von `deleteMax(37)`:



- `siftDown(34)`: Es ändert sich nichts.

Aufgabe 9.3 – Heapsort

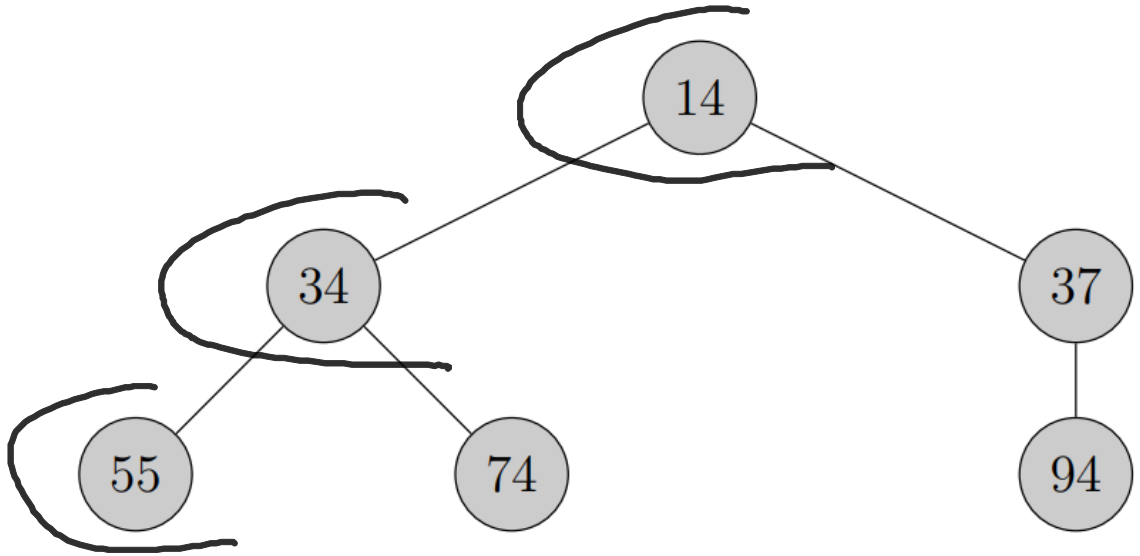
- Vertauschen von 34 mit letzten Element 14, gefolgt von `deleteMax(34)`:



- `siftDown(14)`: Es ändert sich nichts.

Aufgabe 9.3 – Heapsort

- `deleteMax(14)`:



14 34 37 55 74 94

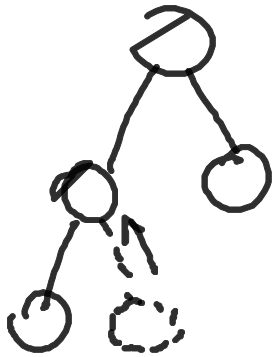
Aufgabe 9.4 – Binäre Heaps

Führen Sie auf einem anfangs leeren Binären Heap folgende Operationen aus und stellen Sie die Zwischenergebnisse graphisch dar:

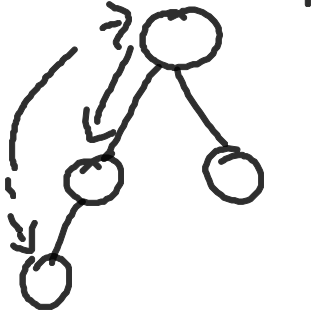
- build({15, 20, 9, 1, 11, 8, 4, 13, 17}),
- insert(7),
- delMin(),
- replaceKey(20, 3),
- delete(8).

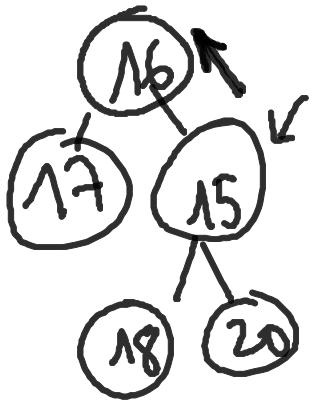
Min-Heap

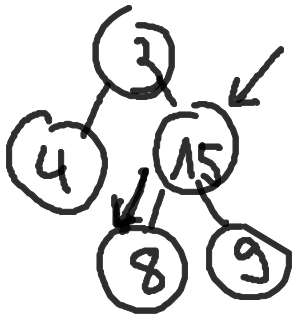
insert



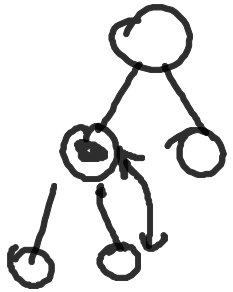
delMin()







delete(_)

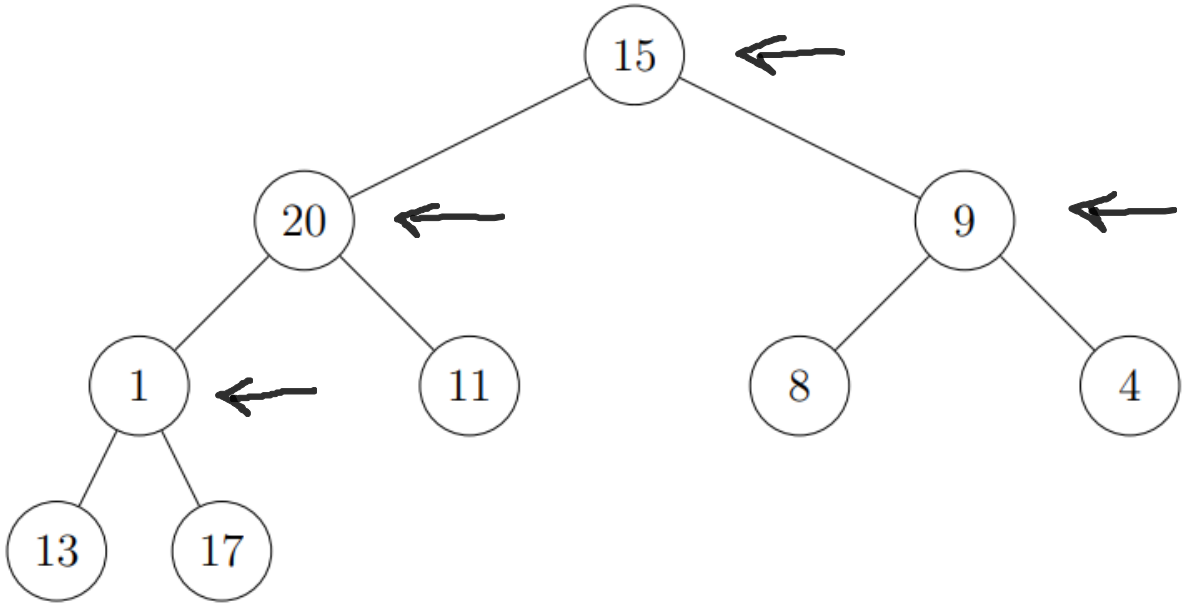


tauschen

- siftUp bis nicht mehr möglich
- siftDown bis nicht mehr möglich

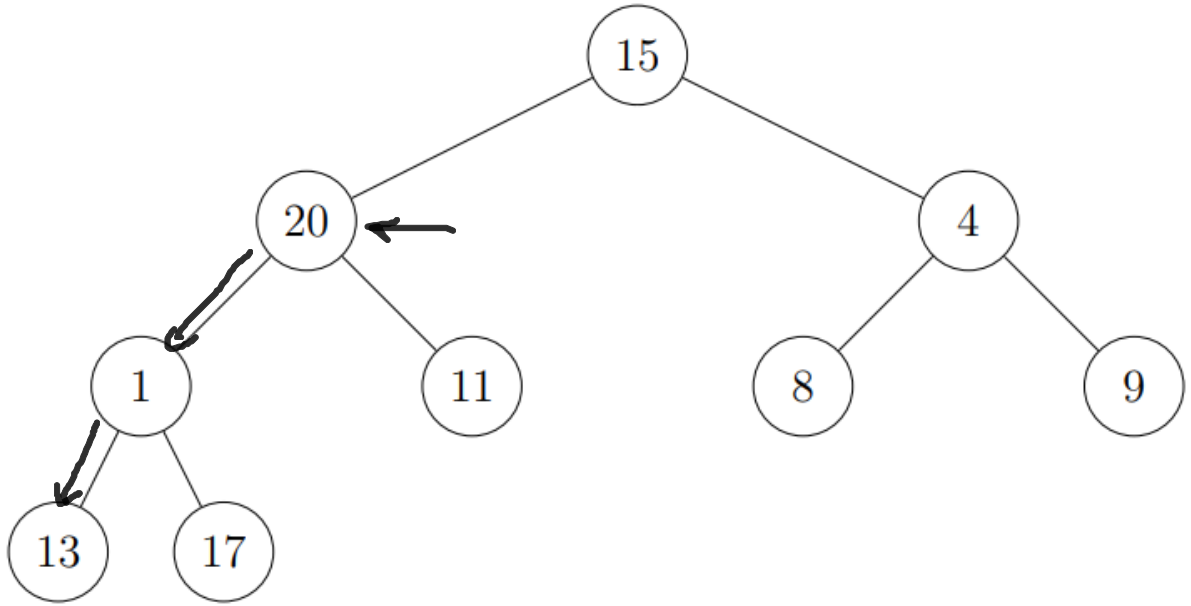
Aufgabe 9.4 – Binäre Heaps

Anfang von build:



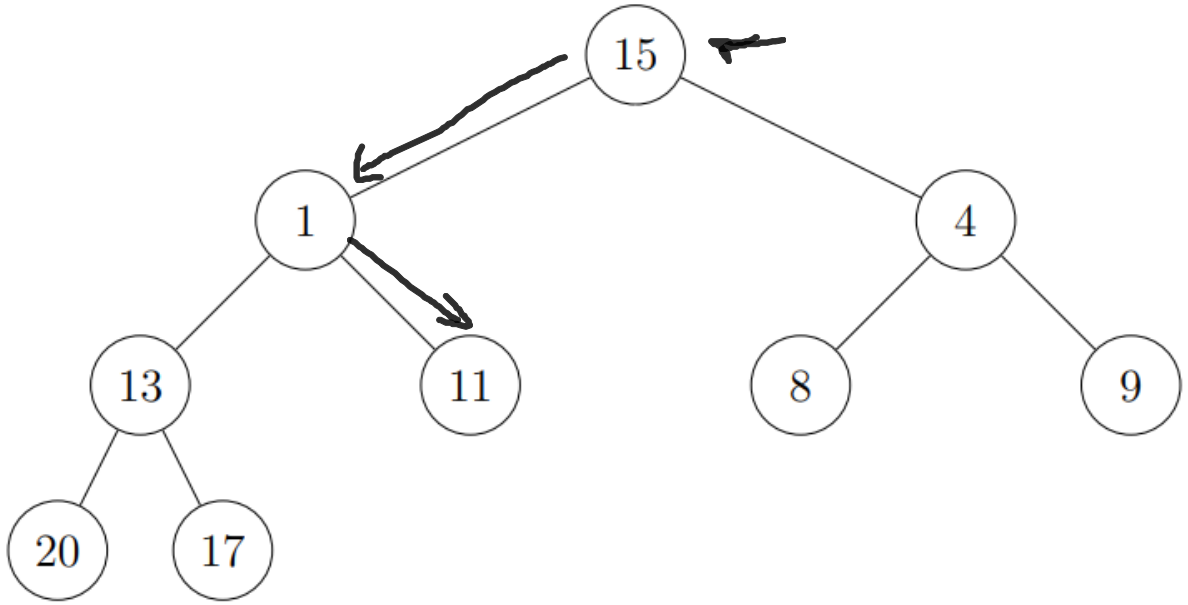
Aufgabe 9.4 – Binäre Heaps

Bei siftDown von 1 passiert nichts. Daher gleich siftDown von 9:



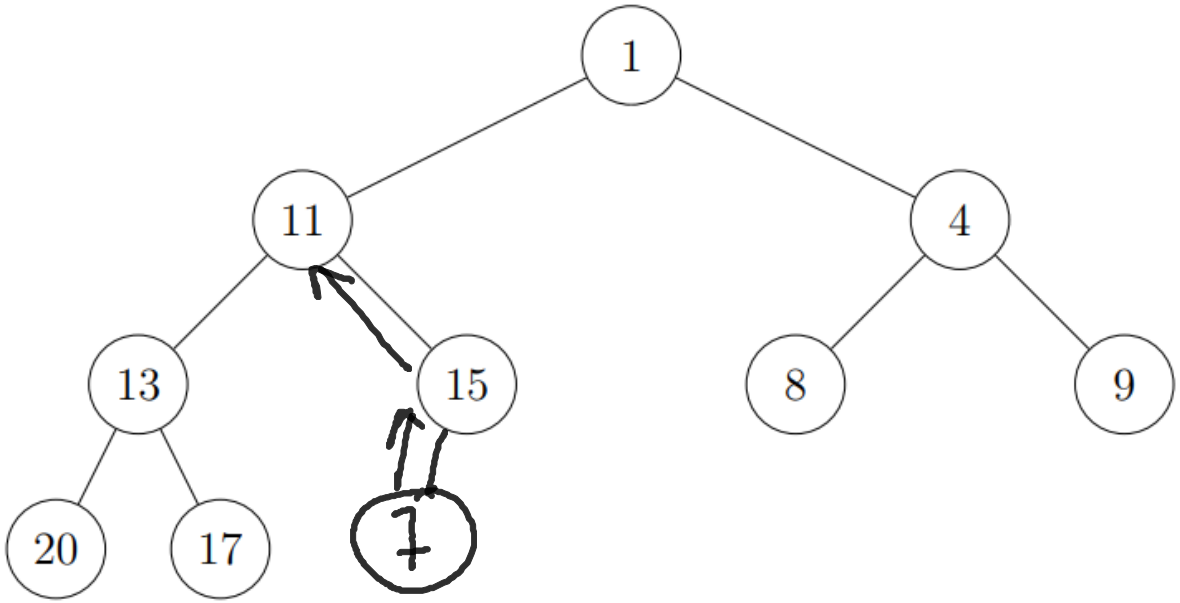
Aufgabe 9.4 – Binäre Heaps

siftDown von 20:



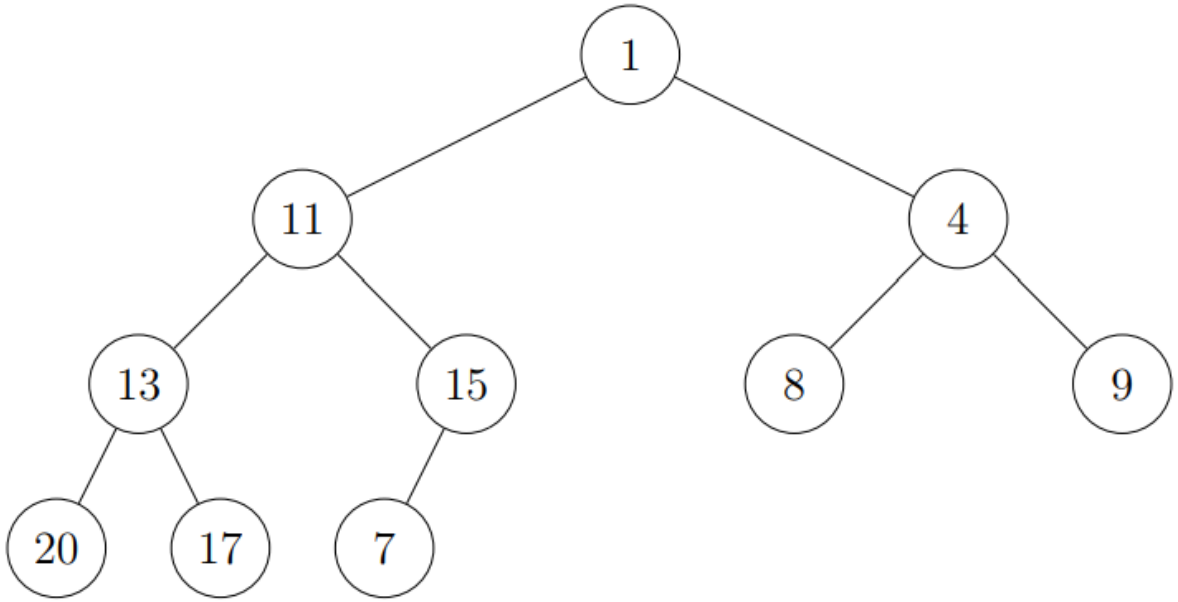
Aufgabe 9.4 – Binäre Heaps

siftDown von 15 und damit das Endresultat für build:



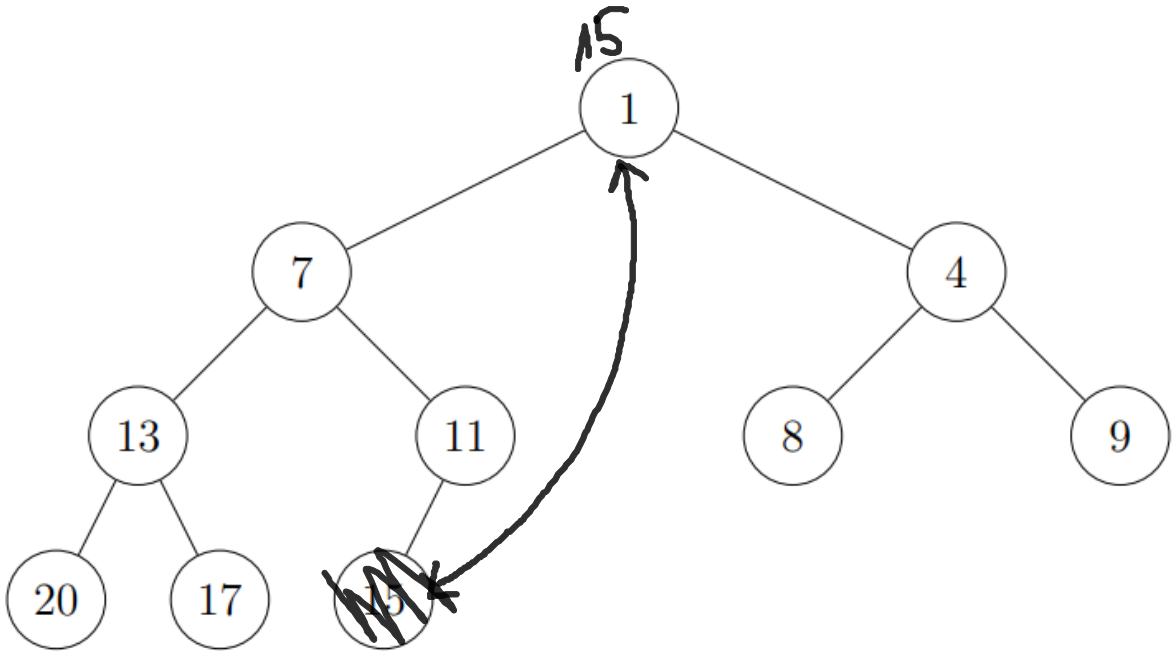
Aufgabe 9.4 – Binäre Heaps

Einfügen der 7. Zuerst hinten anhängen.



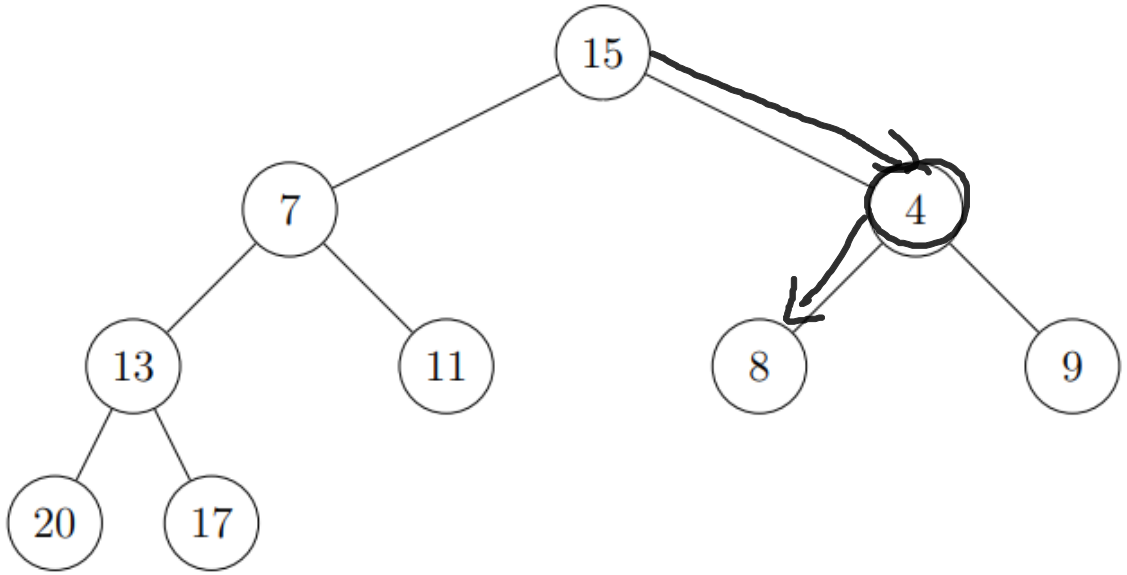
Aufgabe 9.4 – Binäre Heaps

siftUp von 7:



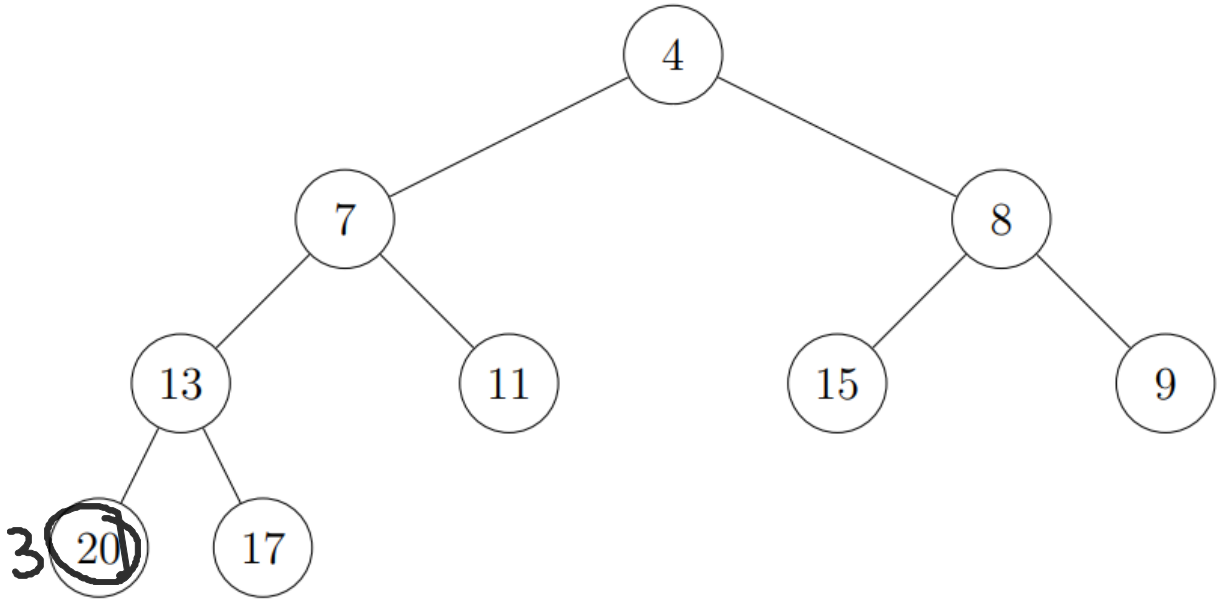
Aufgabe 9.4 – Binäre Heaps

Vorbereitung für deleteMin (d.h. Vertauschen der Wurzel mit dem letzten Element und anschließendes Löschen der *ehemaligen* Wurzel):



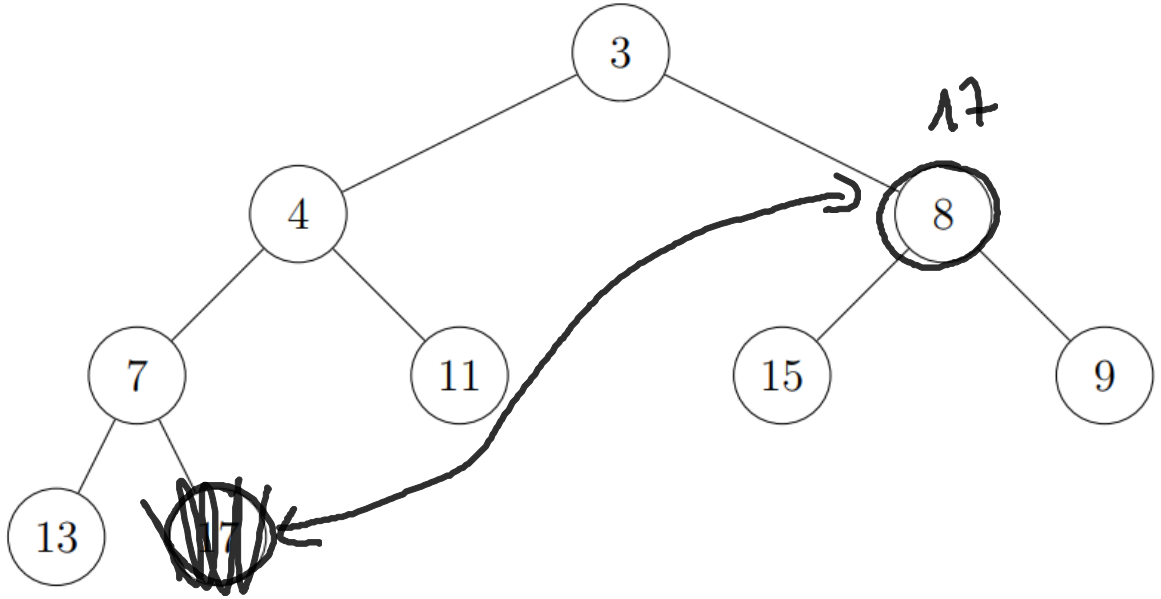
Aufgabe 9.4 – Binäre Heaps

siftDown von 15:



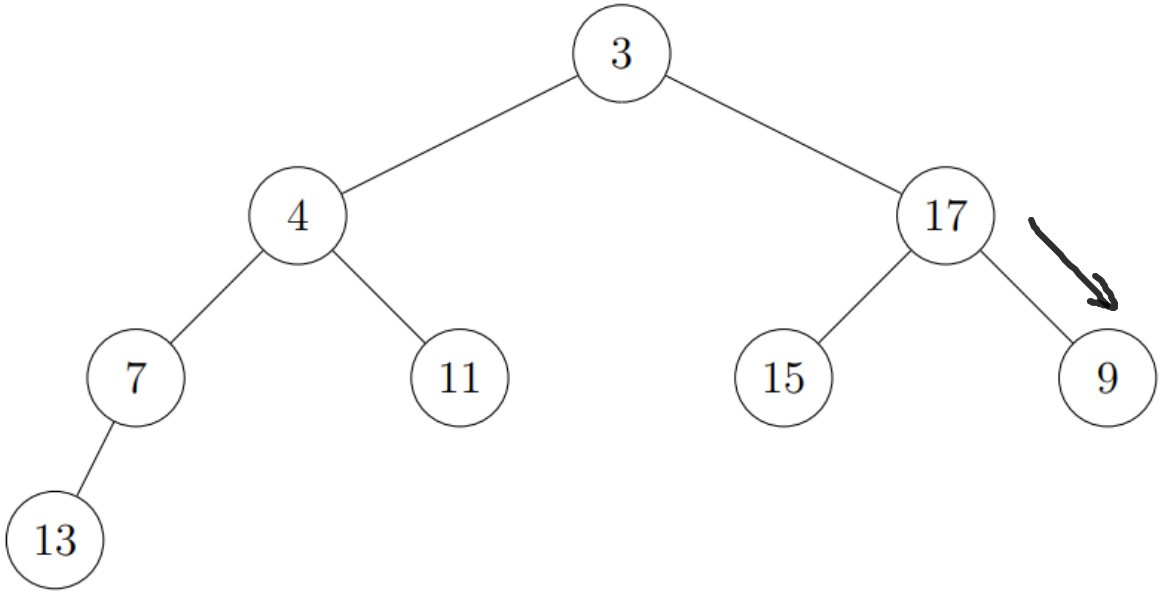
Aufgabe 9.4 – Binäre Heaps

siftUp von 3:



Aufgabe 9.4 – Binäre Heaps

delete von 8 vorbereiten (d.h. Vertauschen von 8 mit dem letzten Element, und anschließendes Löschen von 8):



Aufgabe 9.4 – Binäre Heaps

Nun muss die 17, die eventuell die Heap-Invariante verletzt, an die korrekte Position gebracht werden. Zunächst siftUp von 17, was in diesem Fall den Heap nicht verändert. Anschließend siftDown von 17.