7. Sortieren

Lernziele:

- Die wichtigsten Sortierverfahren kennen und einsetzen können,
- Aufwand und weitere Eigenschaften der Sortierverfahren kennen,
- das Problemlösungsparadigma Teile-und-herrsche verstehen und zur Konstruktion von Algorithmen verwenden können und
- Algorithmen für Probleme kennen, die mit Sortieren verwandt sind.

Einfache Sortierverfahren

Siehe "Einführung in die Programmierung":

- Sortieren durch Auswählen (S. 424–426)
- Sortieren durch Einfügen (S. 427–429)

Weiteres bekanntes Verfahren: Sortieren durch Vertauschen: Bubblesort (s.u.)

Bemerkung:

- Um die Darstellung zu vereinfachen, werden alle Sortierverfahren am Beispiel der aufsteigenden Sortierung von **int**-Werten (**int**-Feld) vorgestellt.
- Abstraktere Konzepte für den Vergleich haben wir in den vorangegangenen Kapiteln kennengelernt.

Bubblesort: Grundidee

- Gegeben ist ein zu sortierendes **int**-Feld a.
- In jeder Iteration durchlaufen wir das Feld von links nach rechts.
- Wenn `a[i] > a[i+1]` gilt, dann vertauschen wir die beiden Werte.
- Wenn in einer Iteration keine Vertauschung mehr notwendig ist, ist das Feld sortiert.

```
Beginn:                  5 4 2 3 7 8 6
Nach der 1. Iteration: 4 2 3 5 7 6 8
Nach der 2. Iteration: 2 3 4 5 6 7 8
Nach der 3. Iteration: 2 3 4 5 6 7 8
```

Bubblesort: Bemerkungen

- Nach der ersten Iteration ist garantiert, dass das größte Element ganz rechts steht (Index `length-1`).
- Spätestens nach der zweiten Iteration befindet sich das zweitgrößte Element an der zweiten Stelle von rechts.
- Somit ist das Feld spätestens nach `length` Iterationen sortiert.

Optimierung:

- Wenn in einer Iteration die letzte Vertauschung an Position `i` (mit `i+1`) stattgefunden hat, dann enthält das Feld ab Position `i+1` die `length-i-1` größten Werte des Feldes.
- Konsequenz: Wir merken uns die Position der letzen Vertauschung und gehen in einer Iteration nur noch bis zu dieser Stelle.

Bubblesort: Algorithmus

```
static void bubblesort(int[] a) {
    int lastSwap=a.length-1;       // Position des letzten Tausches
    int iright;                    // rechte Grenze fuer Iteration

    while (lastSwap > 0) {         // solange in voriger Iteration Vertauschung
        iright = lastSwap;         // lege rechte Grenze fuer Iteration fest
        lastSwap = 0;              // noch gab es keine Vertauschung
        for(int i=0 ; i<iright ; i++) {
            if ( a[i] > a[i+1] ) { // wenn Tausch notwendig
                swap(a,i,i+1);     // dann tausche
                lastSwap = i;      // und merke Position
            }
        }
    }
}
```

Bubblesort: Aufwand im Best und Worst Case

Das Feld a habe die Länge n.

- Best Case:
  Das Feld a ist schon aufsteigend sortiert. Dann ist in der ersten Iteration keine Vertauschung notwendig und der Algorithmus terminiert.
  Aufwand: $O(n)$
- Worst Case:
  Das Feld a ist absteigend sortiert. Dann sind n Iterationen notwendig mit $n-1, n-2, \ldots$ Vertauschungen.
  Aufwand: $O(n^2)$

Bubblesort: Aufwand im Mittel (Average Case)

- Wenn zu Beginn der Sortierung für `i < j` gilt `a[i] > a[j]` (wir bezeichnen dies als Fehlstand), dann müssen die beiden Elemente `a[i]` und `a[j]` irgendwann einmal getauscht werden.
- Wie viele Fehlstände hat eine zufällig generierte Folge der Länge n im Mittel?
- Für das erste Element wird es unter den Elementen 2 bis n im Mittel $\frac{1}{2}(n-1)$ Fehlstände geben, für das zweite Element im Mittel $\frac{1}{2}(n-2)$, usw.
- Wir haben also im Mittel

$$\frac{1}{2}\sum_{i=1}^{n-1}(n-i) = \frac{1}{2}\sum_{i=1}^{n-1} i = \frac{1}{2}\cdot\frac{n(n-1)}{2} = O(n^2)$$

  Fehlstände.
- Damit sind im Mittel $O(n^2)$ Vertauschungen notwendig und der Aufwand beträgt auch im Mittel $O(n^2)$.

Zusammenfassung: Einfache Sortierverfahren

- Alle bisher betrachteten Verfahren haben im Mittel eine Laufzeit von $O(n^2)$.
- Ist Sortieren nicht schneller möglich als in Zeit $O(n^2)$?
- Doch!
- Und im Prinzip wissen wir auch schon wie! ☞ ausgeglichene Bäume

Sortieren mit Hilfe von ausgeglichenen Bäumen

Erste Idee: Wir bauen mit den n zu sortierenden Werten einen AVL-Baum auf.

Konsequenzen:

- Jede Einfügeoperation hat Aufwand $O(\log n)$.
- Gesamtaufwand für den Aufbau des Baums daher: $O(n \log n)$
- Eine Inorder-Traversierung des AVL-Baums liefert die sortierte Reihenfolge in Zeit $O(n)$.
- Gesamtaufwand Sortierung: $O(n \log n)$

Kleinere Probleme:

- relativ hoher konstanter Faktor, zur Erinnerung: Höhe von AVL-Bäumen bis zu 1.5 mal minimale Höhe, "aufwendige" Ausgleichsalgorithmen
- keine In-Place-Sortierung (in situ): Es wird zusätzlich $O(n)$ Speicher benötigt.

Heap

Definition 7.1. Es sei T ein binärer Baum, wobei an jedem inneren Knoten w ein Wert $val(w)$ gespeichert wird. Für einen Knoten w bezeichne w_l den linken und w_r den rechten Sohn (Wurzel des linken bzw. rechten Unterbaums).

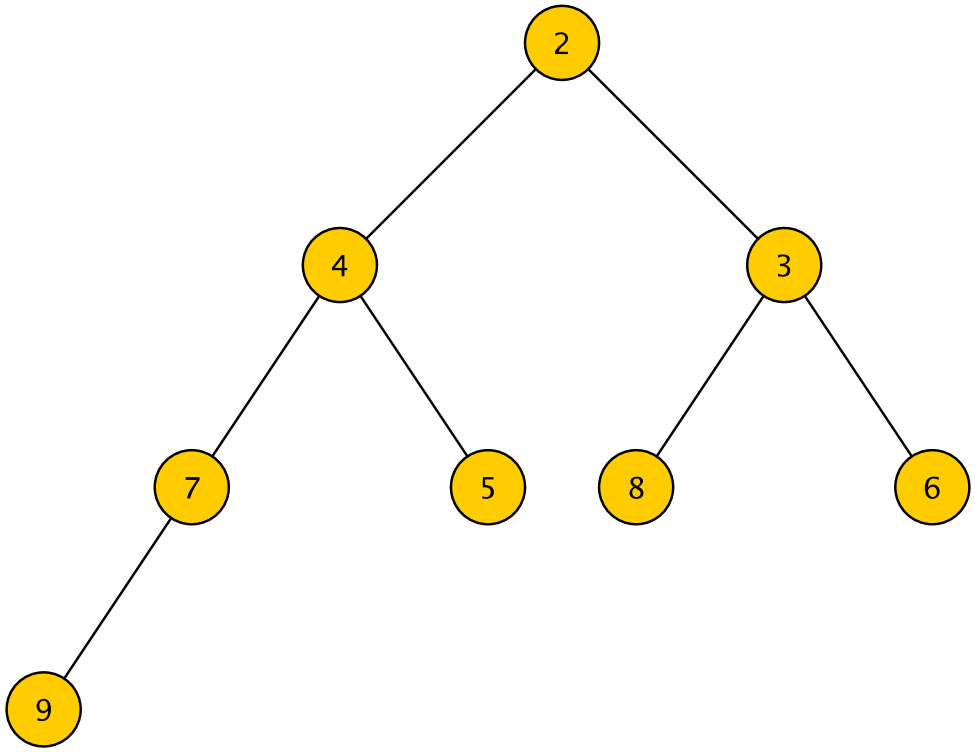
T ist ein *Heap*, gdw. für alle Knoten w gilt:

- w_l nicht leer $\Rightarrow val(w) \leq val(w_l)$ und
- w_r nicht leer $\Rightarrow val(w) \leq val(w_r)$

Anschaulich: Der Wert an einem Knoten ist stets kleiner oder gleich den Werten an den Söhnen.

Wir sprechen in diesem Fall von einem Min-Heap, gilt $\geq$ dann liegt ein Max-Heap vor.

Beispiel: Min-Heap



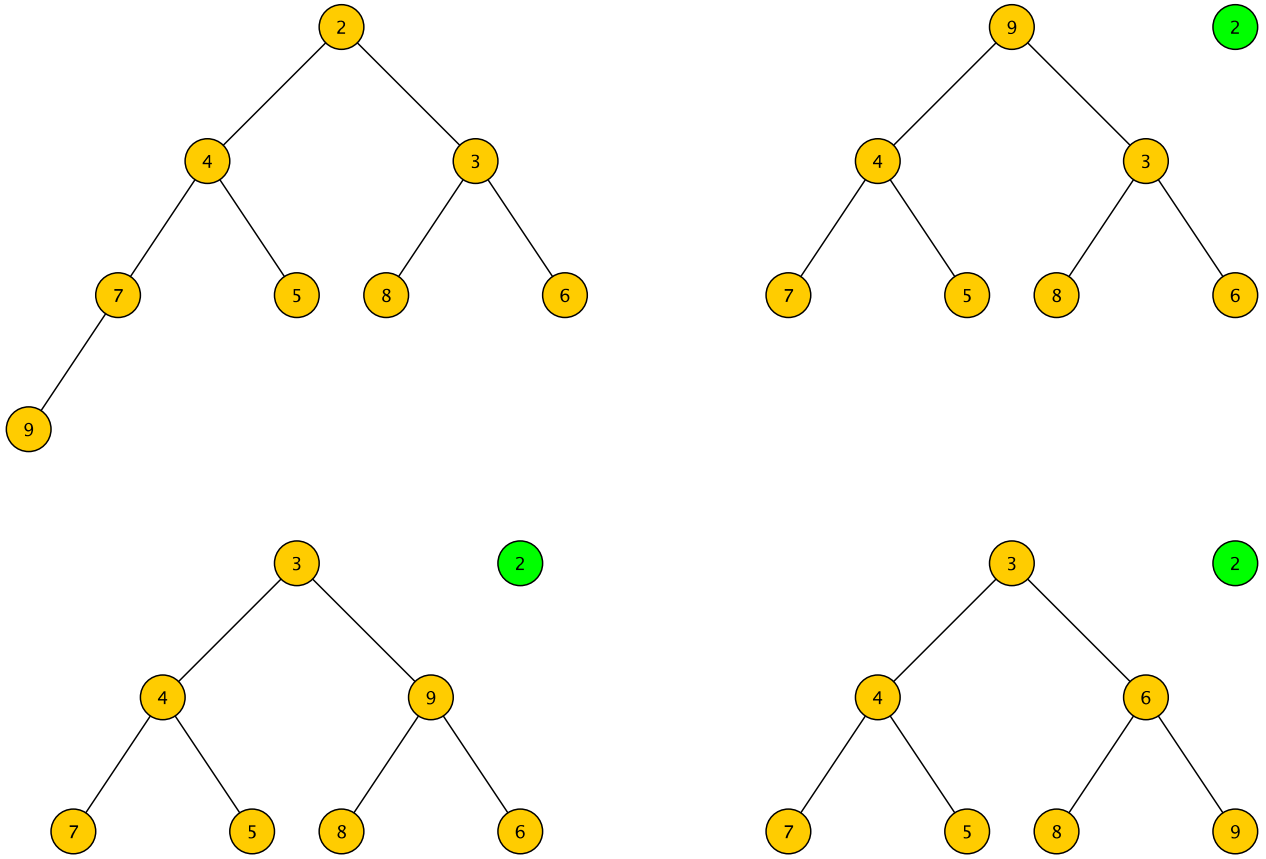
Sortieren eines Heap

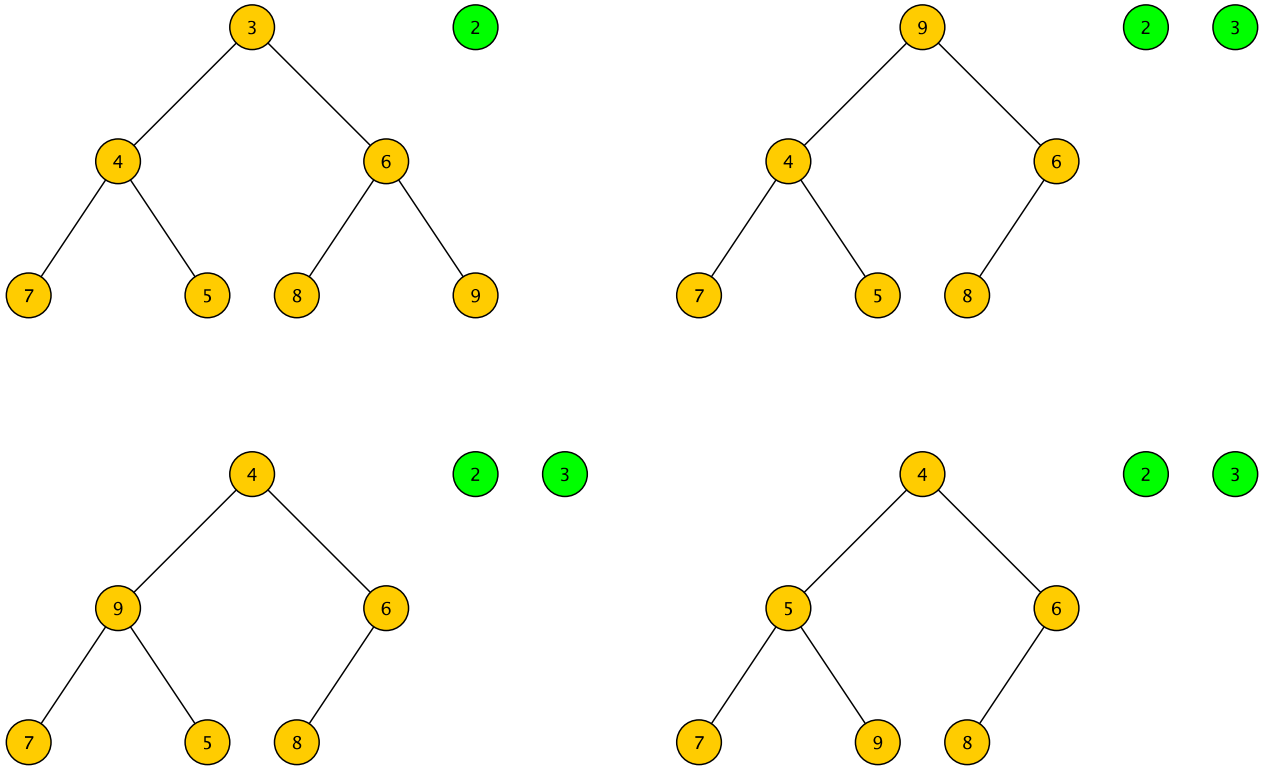
- An der Wurzel steht (bei einem Min-Heap) der kleinste Wert.
- Wir nehmen den Wert der Wurzel, nehmen ihn in die sortierte Folge auf und löschen ihn aus dem Heap.
- Um die Lücke an der Wurzel zu füllen, verschieben wir den am weitesten rechts stehenden Wert der untersten Ebene in die Wurzel und löschen den dazugehörigen Knoten.

  Hierdurch kann die Heap-Definition an der Wurzel verletzt werden.
- Sollte die Heap-Definition verletzt sein, vertauschen wir den Wert der Wurzel mit dem kleineren der beiden Söhne.

  Damit ist an der Wurzel die Heapbedingung wieder erfüllt.
- Falls jetzt am ausgetauschten Sohn die Heap-Bedingung verletzt ist, verfahren wir dort wie an der Wurzel.
- Wir setzen dieses Verfahren fort, bis keine Verletzung mehr vorliegt. Dies ist spätestens dann der Fall, wenn wir die unterste Ebene des Heap erreichen.

Beispiel: Sortieren eines Heap





✎ Beispiel weiter an Tafel

Vollständiger Binärbaum

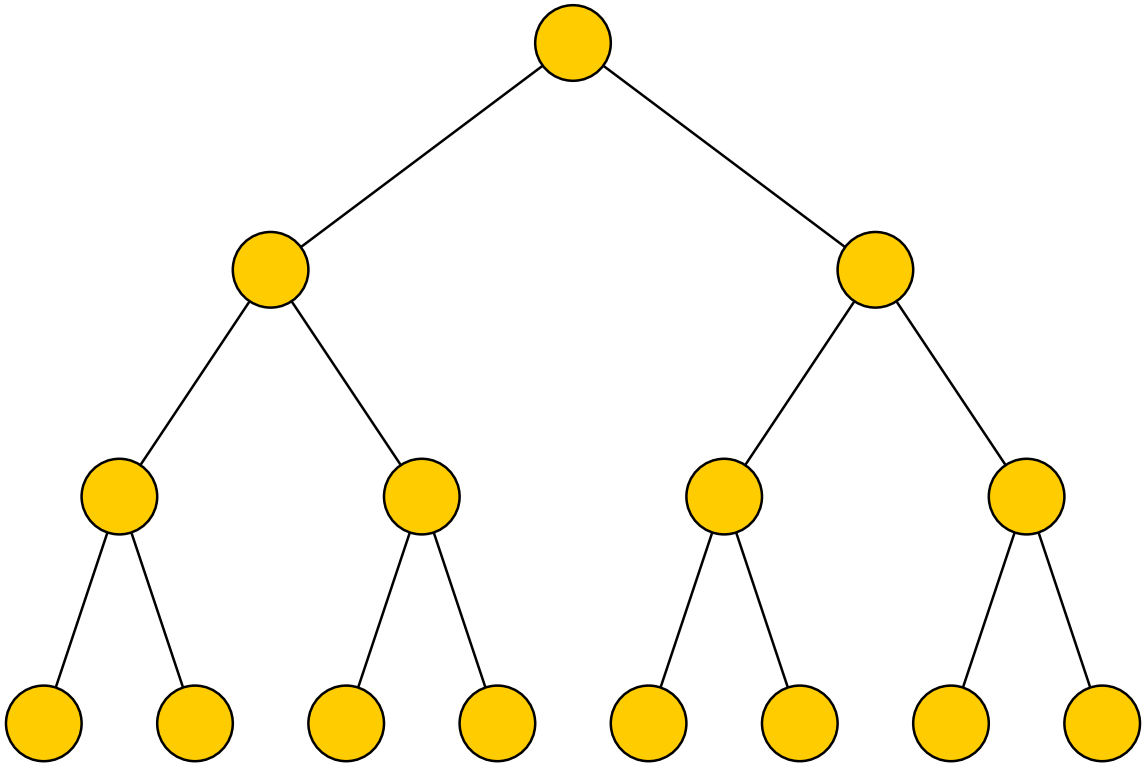
Definition 7.2. Ein binärer Baum der Höhe h heißt *vollständig*, wenn er $2^h - 1$ (innere) Knoten hat.

Bemerkung: vgl. Lemma 6.2:

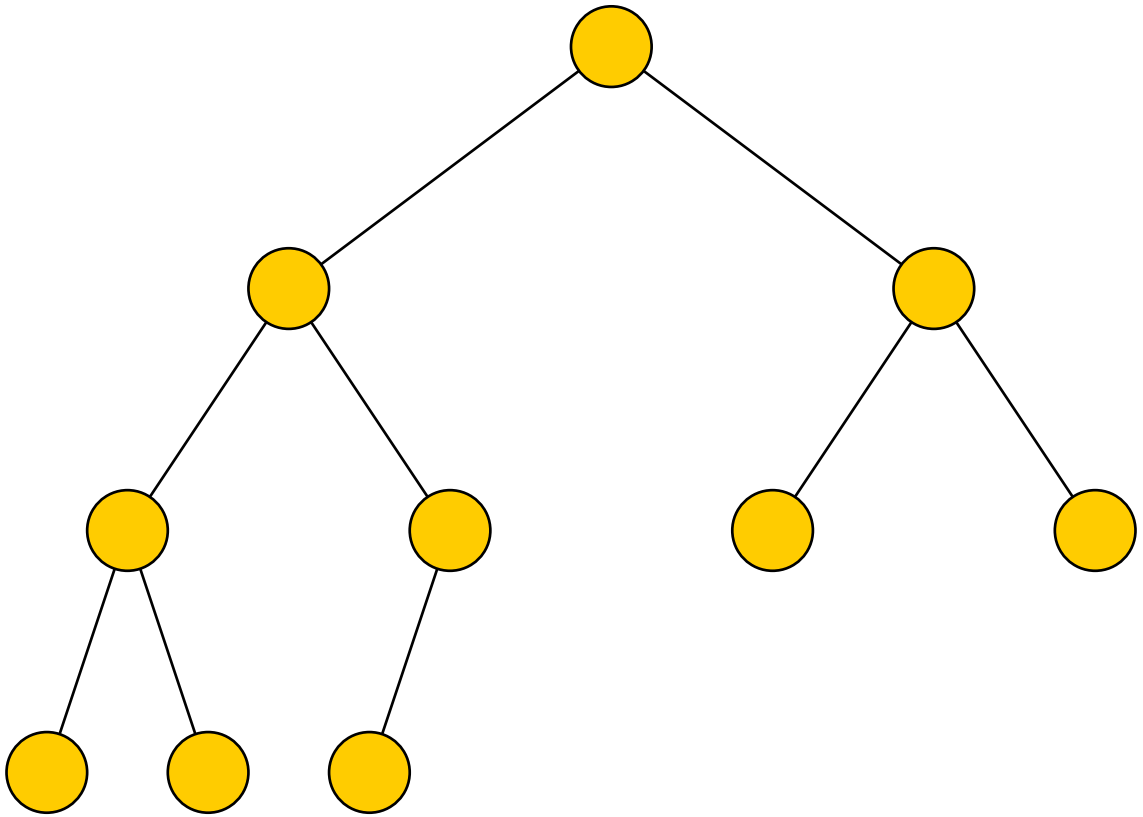
- Ein vollständiger Baum hat für seine Höhe maximal viele Knoten n. Für n ist die Höhe minimal.
- Es gilt: $n = 2^h - 1$.
- Daraus folgt: $h = \log_2(n + 1)$

Definition 7.3. Ein binärer Baum der Höhe h mit $2^{h-1} \leq n \leq 2^h - 1$ Knoten, bei dem die Knoten der Ebene h alle möglichst weit links stehen, heißt *links vollständiger binärer Baum*.

Beispiel: Vollständiger Binärbaum der Höhe 4



Beispiel: Links vollständiger Baum

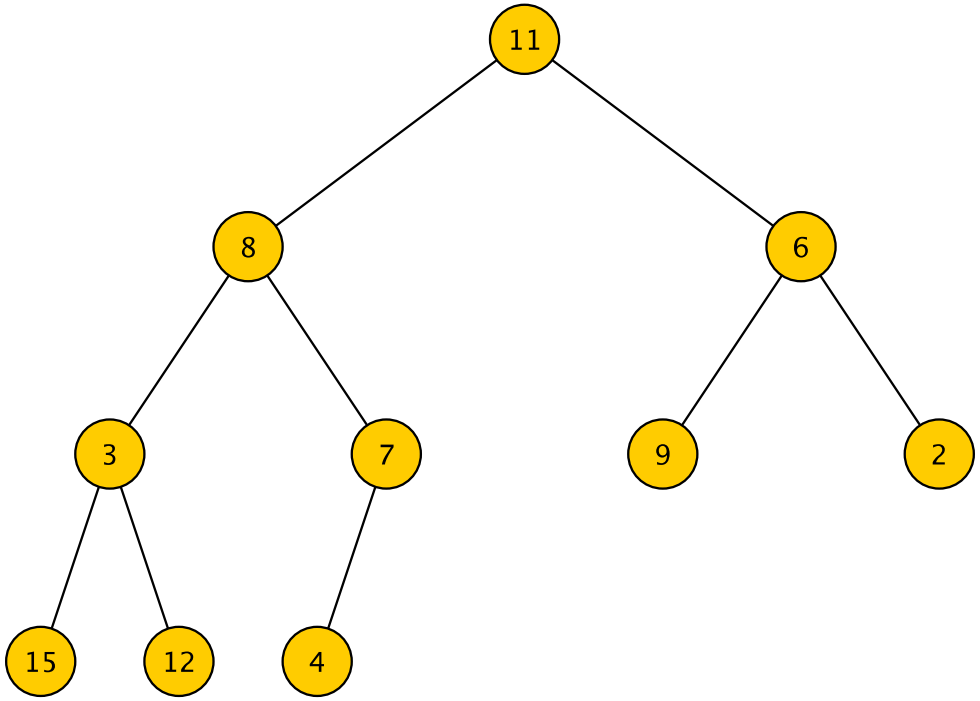


Konstruktion eines Heap

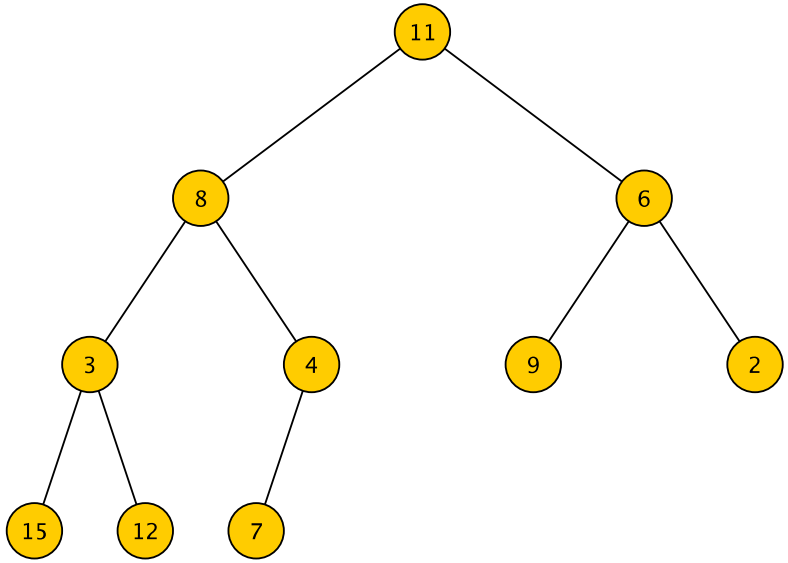
- Wir bauen einen links vollständigen Binärbaum mit n Knoten auf und
- platzieren die zu sortierenden Werte in den Knoten von oben nach unten und in jeder Ebene von links nach rechts.
- Wir überprüfen die Heap-Definition von unten nach oben und in jeder Ebene von rechts nach links.
- Wenn die Heap-Definition an einem Knoten verletzt ist, dann tauschen wir den Wert am Knoten mit dem kleineren der beiden Söhne. Wenn notwendig, setzen wir dieses Verfahren am ausgetauschten Sohn fort (vgl. Folie 300).

Beispiel: Konstruktion eines Heap

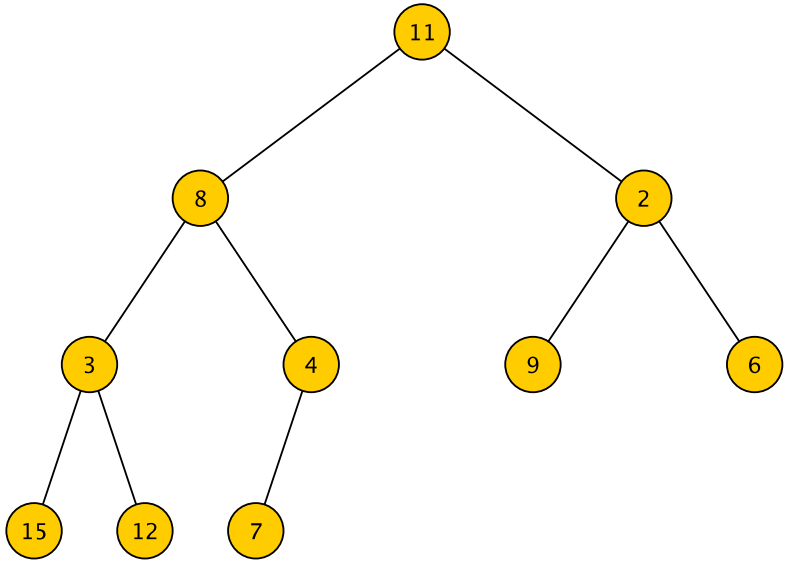
links vollständiger Binärbaum zur Folge 11 8 6 3 7 9 2 15 12 4



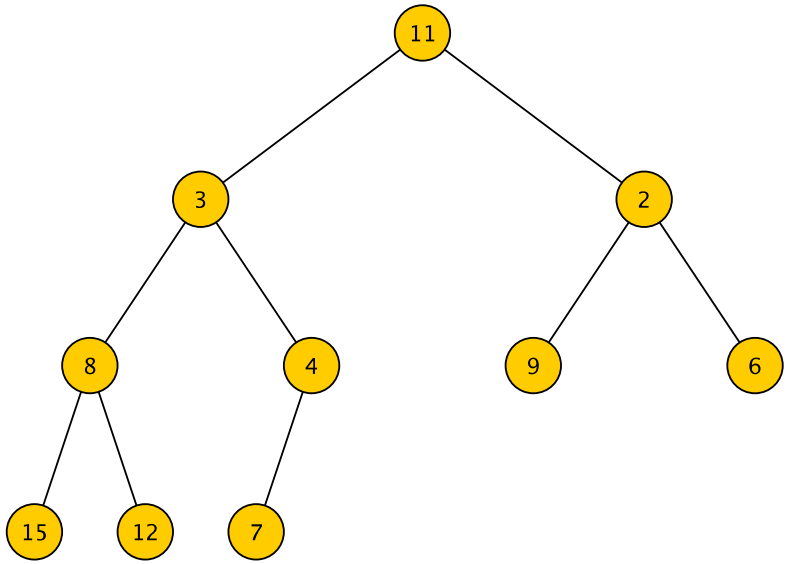
4 12 15 2 9 ok, Vertauschung bei 7



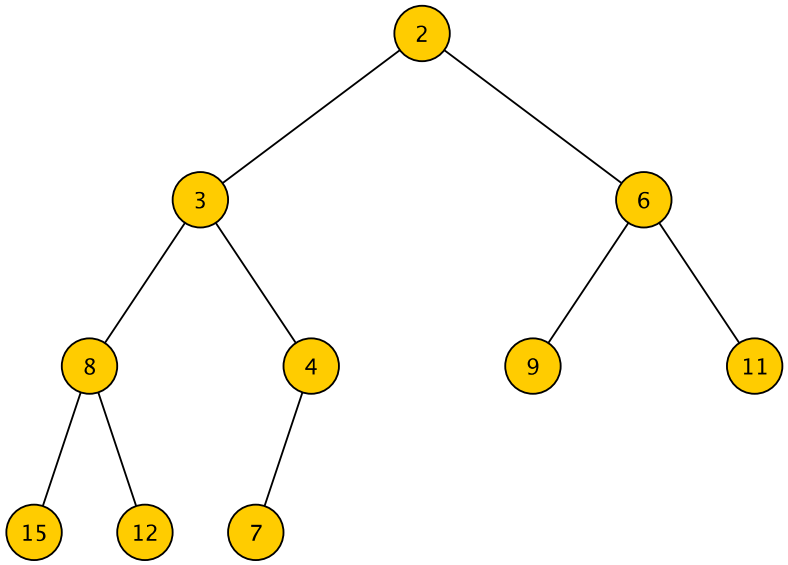
3 ok, Vertauschung bei 6



Vertauschung bei 8



Vertauschung bei 11 mit weiterer Vertauschung



Damit ist der Heap fertig!

Heapsort

(1) Heap konstruieren (Folie 306)

(2) Heap sortieren (Folie 300)

Aufwand im Worst Case:

- Aufbau des links vollständigen Binärbaums: $O(n)$
- Konstruktion des Heap aus dem links vollständigen Baum: $O(n \log n)$, weil für jeden Knoten höchsten $h = O(\log n)$ Vertauschungen anfallen.
- Sortieren des Heap: $O(n \log n)$, weil nach jedem "Abpflücken" der Wurzel höchstens $h = O(\log n)$ Vertauschungen anfallen.
- Gesamtaufwand: $O(n \log n)$

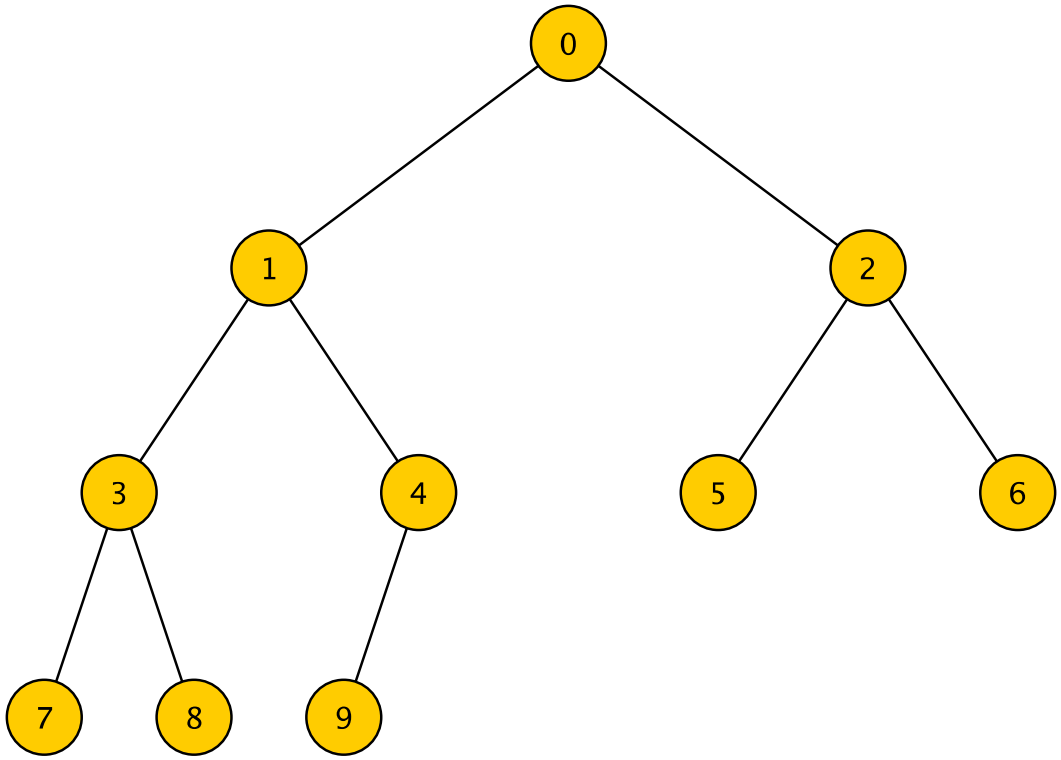
Abbildung eines Heap auf ein Feld

Wir können einen vollständigen Binärbaum mit n Knoten wie folgt auf ein Feld mit Index von 0 bis $n-1$ abbilden:

- Die Wurzel liegt bei Index 0.
- Der Knoten mit Index i hat seine Söhne bei Index $2i+1$ (linker Sohn) und $2i+2$ (rechter Sohn).
- Die Söhne existieren genau dann, wenn $2i+1 \leq n-1$ bzw. $2i+2 \leq n-1$ gilt.

Beispiel: Feldabbildung eines links vollständigen Binärbaums

$n = 10$

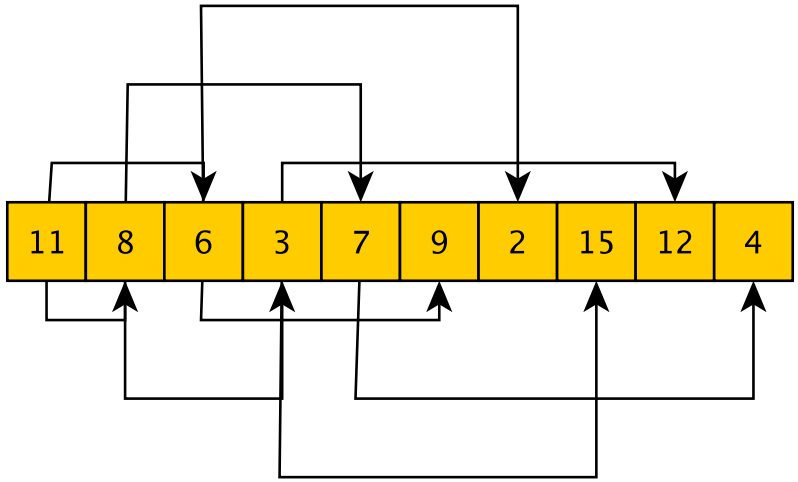


In-Place Heapsort

- Heap-Konstruktion: Stelle im Feld von rechts nach links die Heap-Definition für einen Max-Heap her!
- Heap-Sortierung: analog zu Folie 300:
  - Nehme in jeder Iteration das Maximum aus dem Heap (Index 0),
  - verschiebe den am weitesten rechts stehenden Wert der untersten Ebene in die Wurzel,
  - platziere das soeben gelöschte Maximum an die Stelle, wo der am weitesten rechts stehende Wert der untersten Ebene stand und
  - stelle Heap-Definition (für Max-Heap) wieder her.

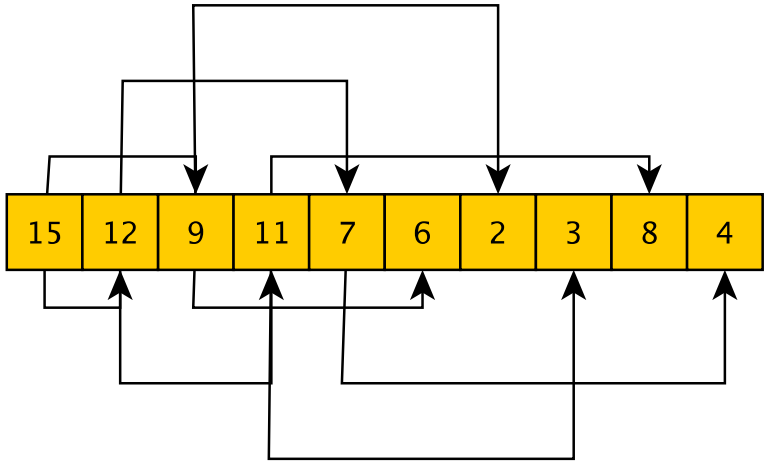
Beispiel: In-Place Heapsort

Zu sortierende Folge als Feld:

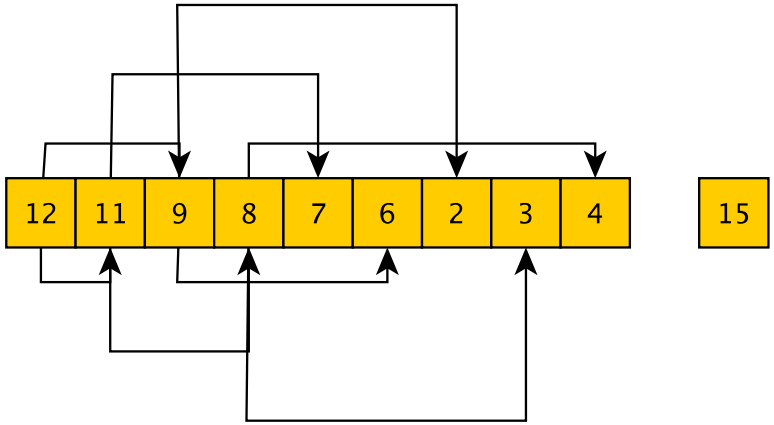


Die unteren Pfeile verweisen auf die linken Söhne, die oberen auf die rechten Söhne (nur zur Veranschaulichung).

Die Konstruktion des Max-Heap liefert:



Wir ziehen die 4 auf die Wurzel. Die 15 wird im Feld der 4 abgelegt, dies gehört jetzt aber nicht mehr zum Heap. Ausgehend von der 4 in der Wurzel stellen wir durch Vertauschungen wieder einen Heap her.



✎ weiter an der Tafel

Fazit Heapsort

- effizient auch im Worst Case: Zeit $O(n \log n)$
- In-Place Implementierung möglich: $O(1)$ zusätzlicher Speicher