Algorithmen und Datenstrukturen

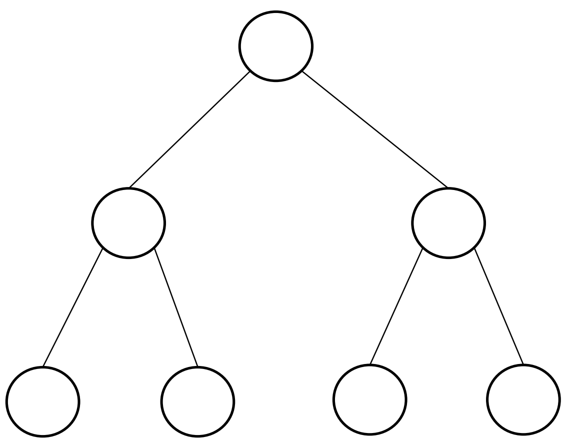
Heapsort ♦ Schlüssel ♦

Datentyp Menge: Hashing

Vollständige Binärbäume

Vollständiger Binärbaum

- Sei $k \geq 0$.
- Ein Binärbaum der Tiefe k heißt **vollständig** gdw.
  1. jedes Blatt hat Tiefe k und
  2. jeder innere Knoten hat zwei Kinder.



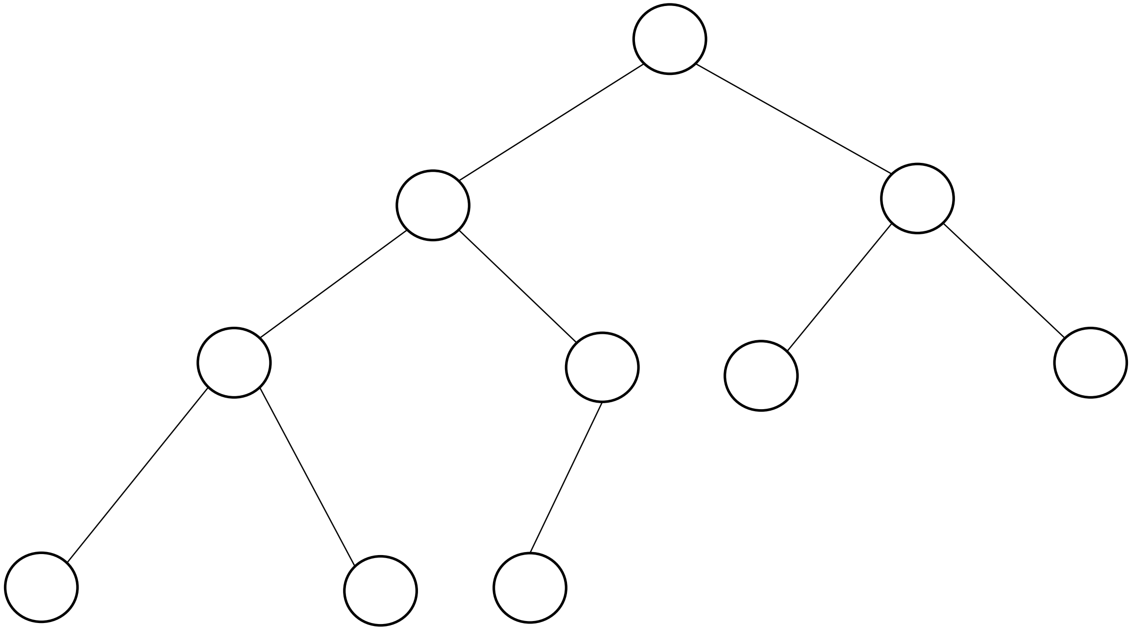
➢ $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$ viele Knoten

Es können also $\Theta(2^k)$ viele Werte gespeichert werden.

Für eine Sequenz der Länge n benötigt man einen Baum der Tiefe $\Theta(\log n)$.

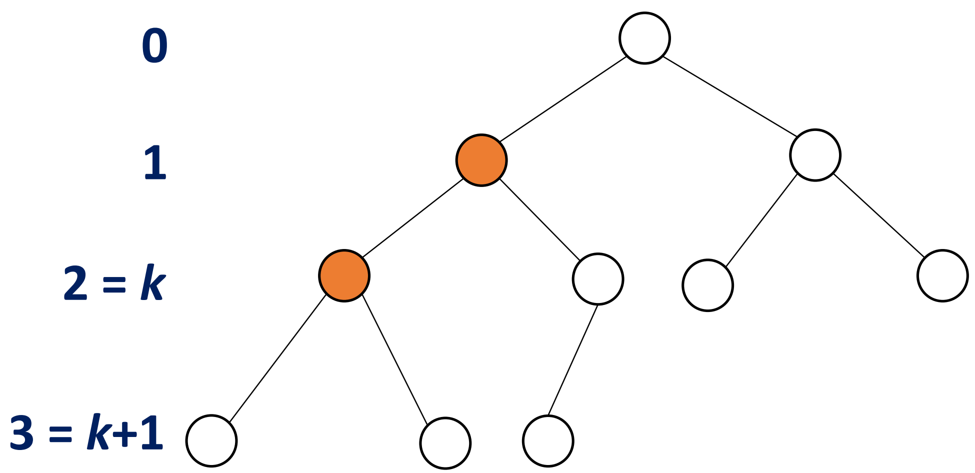
Fast vollständiger Binärbaum – Intuition

- Ein vollständiger Binärbaum, dem „ein paar Blätter fehlen dürfen“.
- Die Ebene größter Tiefe ist „von links her gefüllt“.



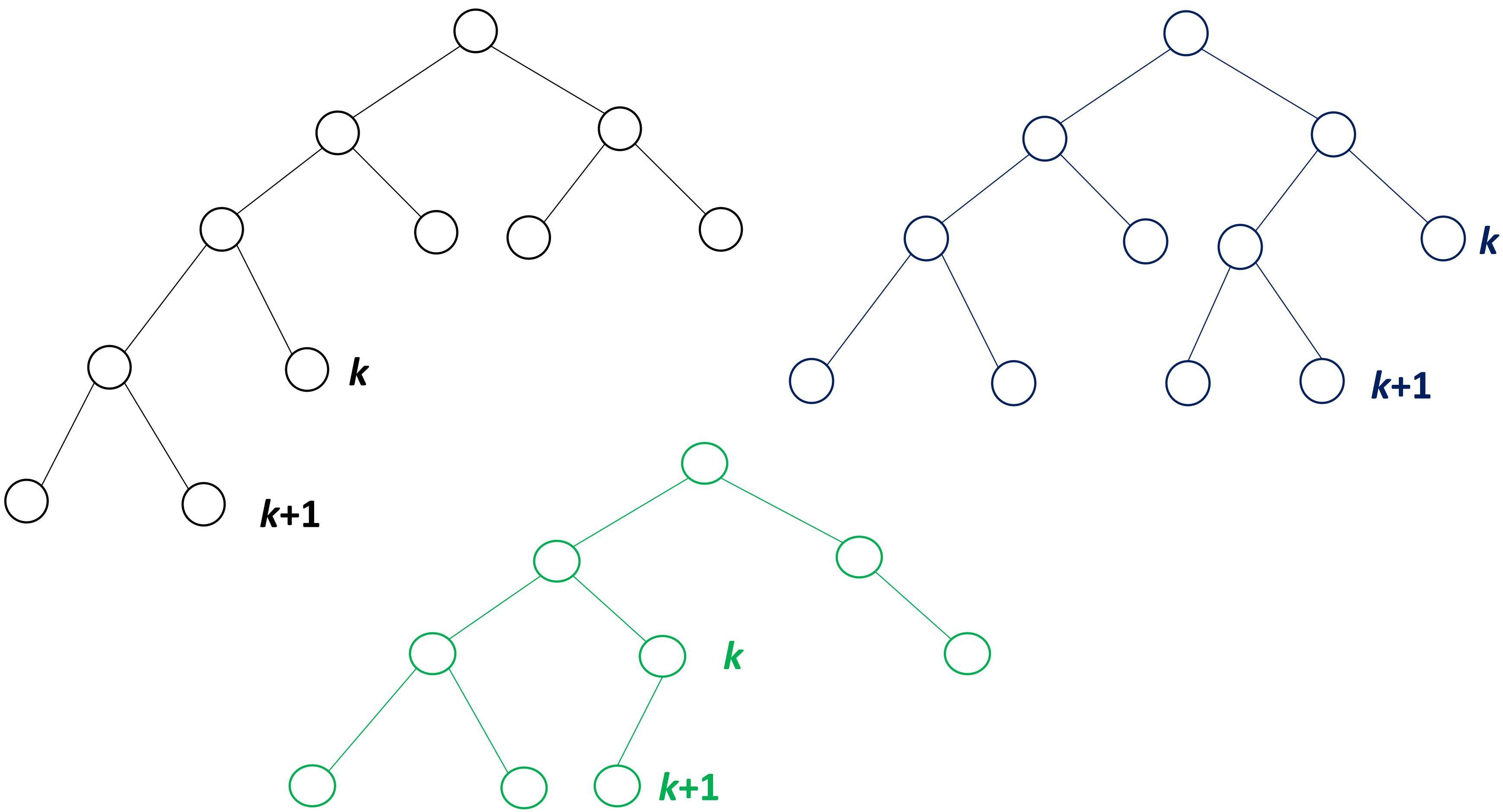
Fast vollständiger Binärbaum – Definition

- Ein Binärbaum der Tiefe k+1 heißt **fast vollständig** gdw.
  1. jedes Blatt hat Tiefe k oder k+1,
  2. jeder Knoten mit Tiefe $<k$ hat zwei Kinder und
  3. falls der rechte Teilbaum eines Knotens der Tiefe $m \leq k$ ein Blatt der Tiefe k+1 enthält, dann ist das linke Kind Wurzel eines vollständigen Binärbaums der Tiefe k-m.



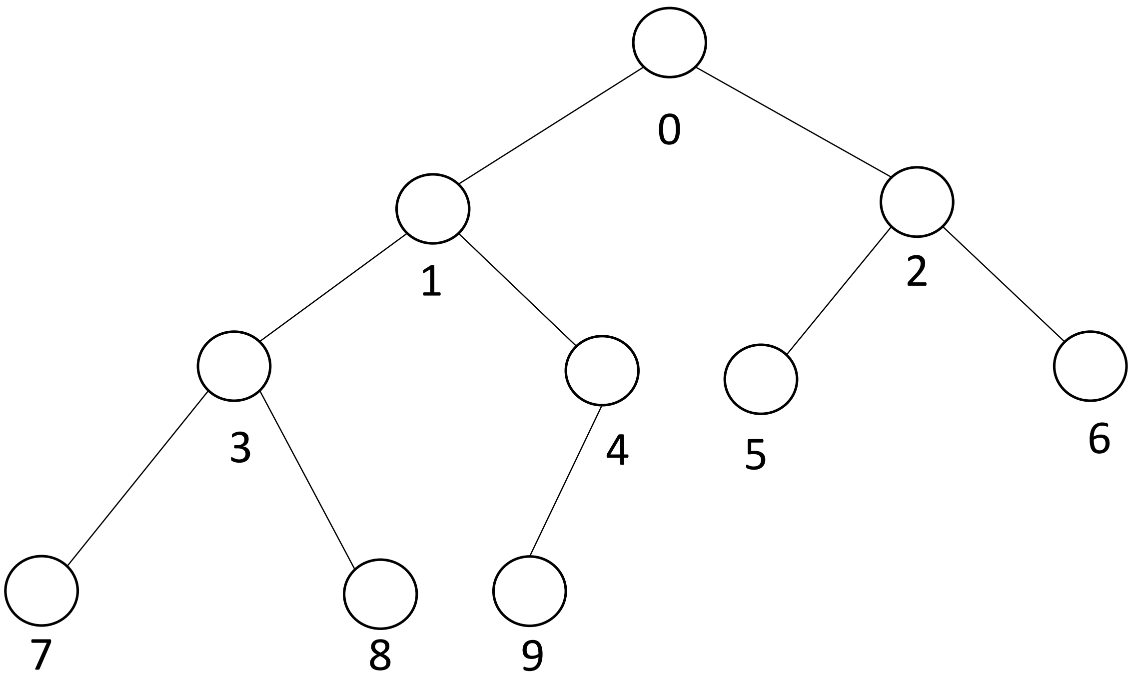
➢ **Vollständige Binärbäume sind spezielle fast vollständige Binärbäume.**

Binärbäume, nicht fast vollständig

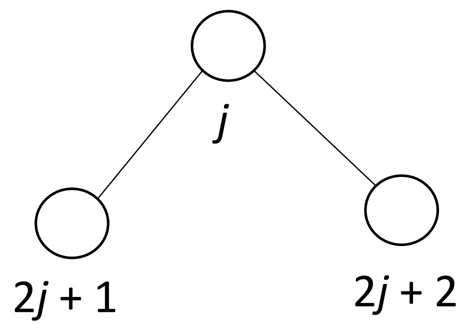


Fast vollständiger Binärbaum – Indizierung

- Indizierung in der Reihenfolge von Levelorder



allgemein:



Falls $i = 2j + 1$
oder $i = 2j + 2$,
dann gilt $j = \lfloor (i - 1)/2 \rfloor$.

- erlaubt Implementierung als Array

Heapsort

Heapsort – Idee (1)

- Williams und Floyd (1964)
- Anordnung der Elemente der zu sortierenden Sequenz in einem fast vollständigen Binärbaum (nach Levelorder)
- *Für aufsteigendes Sortieren:*
  Veränderung der Zuordnung schrittweise, bis jedem Vaterknoten ein Element zugeordnet ist, das nicht kleiner als die Elemente in seinen Kindern ist (**Heap**-Eigenschaft)

➢ Wurzel speichert das größte Element der Sequenz (**Max-Heap**)

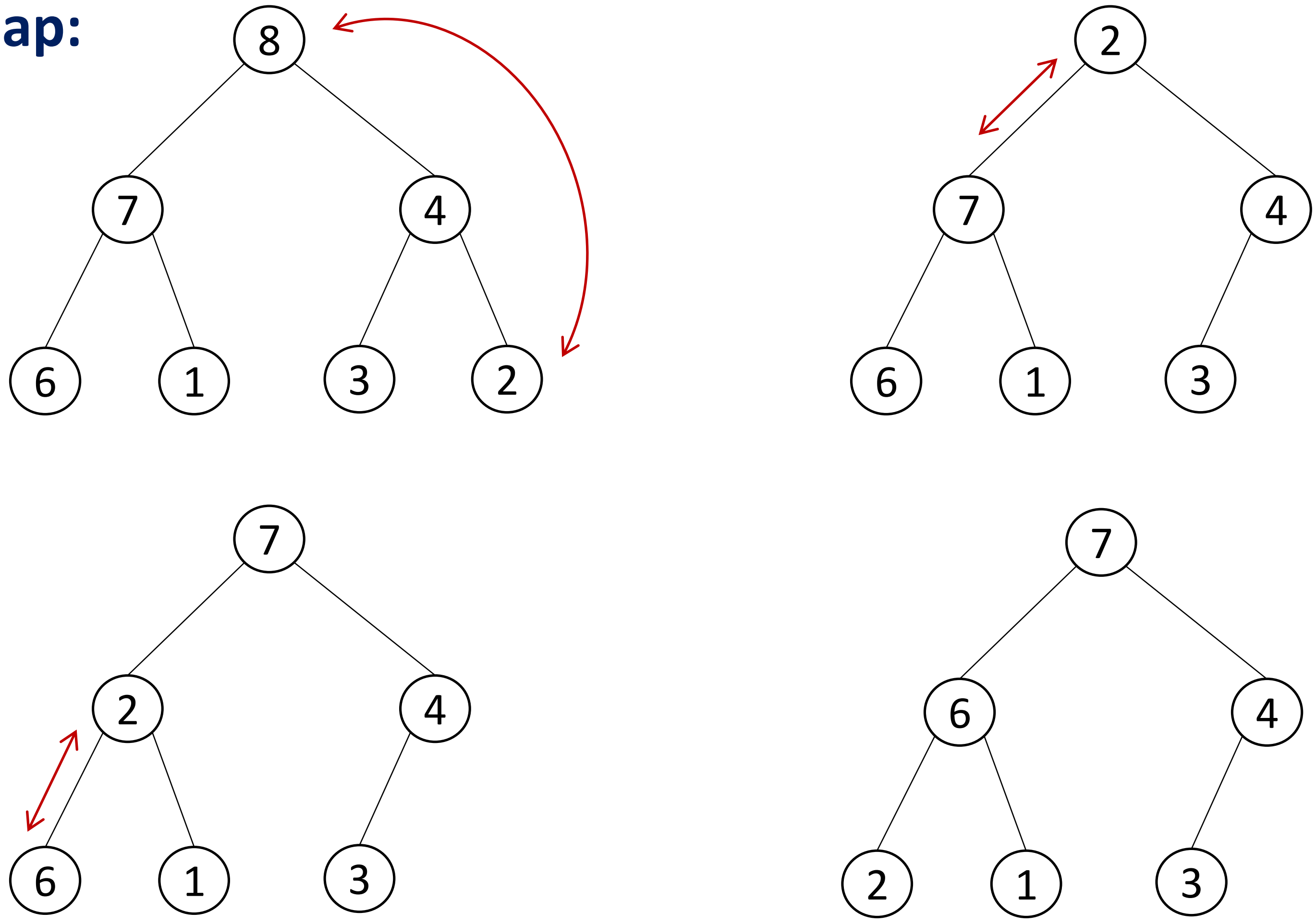
- *Für absteigendes Sortieren:* analog mit **Min-Heap**

Heapsort – Idee (2)

- Tauschen den Wert des letzten Blattes größter Tiefe mit dem der Wurzel und löschen dieses Blatt
  - ➢ letztes Feld im zugehörigen Array ist jetzt mit dem größten Element der Sequenz belegt
  - ➢ letztes Feld im Array gehört nicht mehr zum Baum mit den noch zu sortierenden Elementen
- Heap-Eigenschaft wieder herstellen
  - wenn Wurzel bereits Heap-Eigenschaft: Stopp
  - sonst: vertausche Wurzelelement mit maximalem Element an den Kindern
  - iteriere für alle so veränderten Knoten

Beispiel: Ein Heapsort-Schritt

Heap:

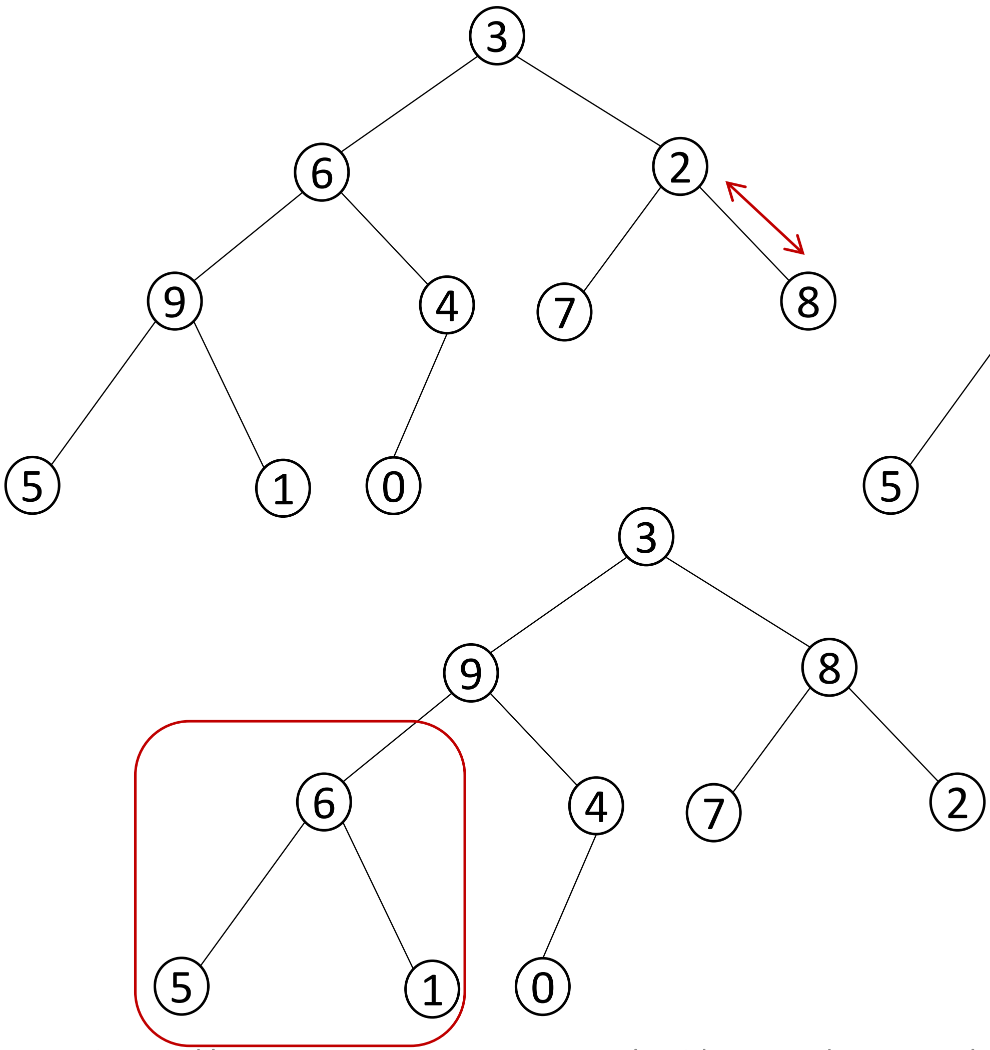


Binärbaum in Heap transformieren

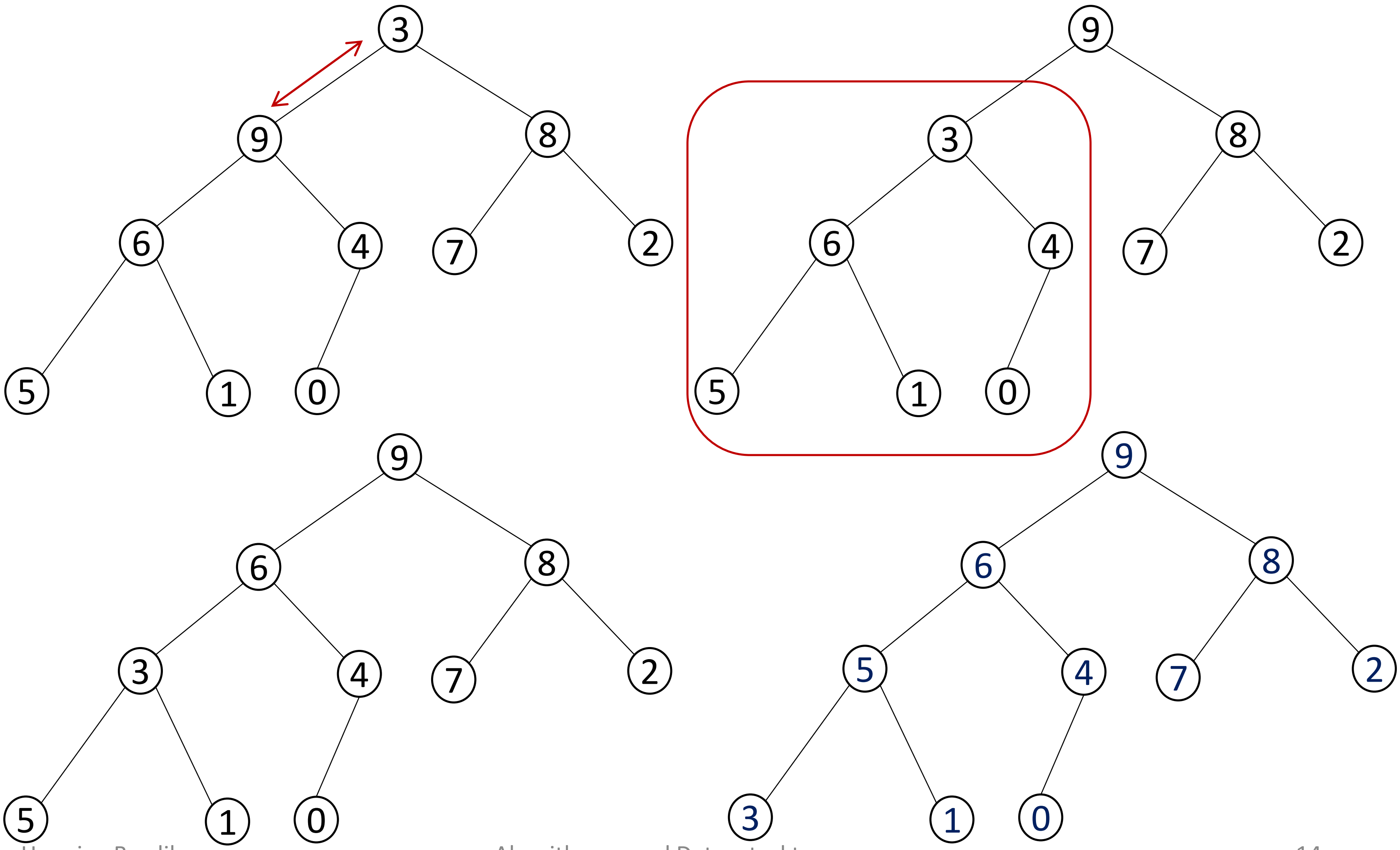
- **Basis:** Jedes Blatt ist ein Heap.
- **Schritt:** Sind der linke und der rechte Teilbaum eines Knotens Heaps der Tiefe m (oder rechts m-1), dann entsteht daraus ein Heap der Tiefe m+1 wie folgt:
  1. Sei x der Wert am Vater beider Teilbäume und y das Maximum der Werte seiner Kinder.
     Falls $x < y$, dann vertausche diese Werte.
  2. Stelle die Heapeigenschaft im veränderten Teilbaum wieder her.

→ Heapeigenschaft (wieder) herstellen ab Vaterknoten

Binärbaum in Heap transformieren



Binärbaum in Heap transformieren

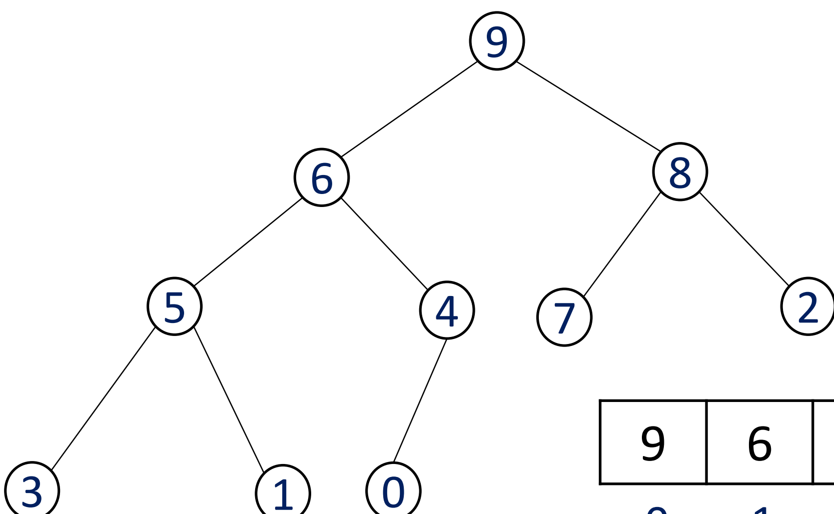


Datenstruktur Heap

- Ein Array A der Länge n heißt **Heap** wenn für alle $1 \leq k < n$ gilt:

$$A[\lfloor (k\text{-}1)/2 \rfloor] \geq A[k]\ .$$

Falls $i = 2j + 1$
oder $i = 2j + 2$,
dann gilt $j = \lfloor (i-1)/2 \rfloor$.



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

Array in Heap transformieren (1)

- benutzen:

```
# Vertauschen der Werte an Positionen i und j in Liste L
# pre: 0 <= i, j <= len(L)-1
def chg(L,i,j):
      tmp = L[i]
      L[i] = L[j]
      L[j] = tmp
```

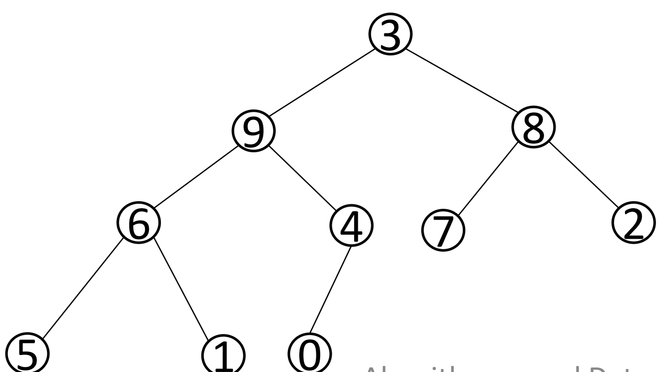
Array in Heap transformieren (2)

Eingabe: Array A von Zahlen

Ausgabe: Heap mit den Elementen von A

Falls $i = 2j + 1$
oder $i = 2j + 2$,
dann gilt $j = \lfloor (i-1)/2 \rfloor$.

```
def buildHeap(A):
    last = len(A) - 1
    # fuer alle inneren Knoten bottom-up
    for v in range((last-1)//2,-1,-1):
        # Heapeigenschaft herstellen
        heapify(A,v,last)
```



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

Heapeigenschaft wieder herstellen

```
def heapify(A,v,last):
   w = 2 * v + 1  # linkes Kind von v
   while (w <= last):  # w ist Knoten des Baums
      if (w+1 <= last):  # Gibt es ein rechtes Kind?
         if (A[w] < A[w+1]): w = w+1
         # w ist Kind mit maximalem Element
      if (A[v] >= A[w]): return  # Heapeigenschaft o.k.
      # sonst:
      chg(A,v,w)
      v = w
      w = 2 * v + 1
```

Heapsort

```
def heapsort(A):
   buildHeap(A)
   last = len(A) - 1
   while (last >= 1):
      # Wurzel mit letztem Blatt tauschen
      chg(A,last,0)
      # letzten Knoten aus Baum entfernen
      last = last - 1
      # Heapeigenschaft wieder herstellen
      heapify(A,0,last)
```

Heapsort – Laufzeitanalyse (1)

- **`heapify`:**
  - **`while`**-Schleife steigt „Ast“ zu einem Blatt ab
    → $O(\log n)$ Schleifendurchläufe
  - je Schleifendurchlauf konstant viele Operationen

    → $\boldsymbol{O(\log n)}$

- **`buildHeap`:** $\Theta(n)$ viele Aufrufe von **`heapify`** → $\boldsymbol{O(n \log n)}$

```
def buildHeap(A):
    n = len(A) - 1
    for v in range(n//2,-1,-1):
        heapify(A,v,n)
```

Heapsort – Laufzeitanalyse (2)

- `heapsort:` $\Theta(n \log n)$

```
def heapsort(A):
    buildHeap(A)  ...................... O(n log n)
    last = len(A) - 1 ........... O(1)
    while (last >= 1): .......... O(n) mal:
        chg(A,last,0)   .....................  O(1)
        last = last - 1 .....................  O(1)
        heapify(A,0,last) ................  O(log n)
```

Mergesort *versus* Heapsort

- beide mit optimaler Laufzeit
- Heapsort sortiert **in-place**
- Mergesort sortiert <u>nicht</u> **in-place**
  (*Mischen erfordert Kopieren zwischen zwei Arrays*)

Verwalten komplexer Daten

Daten und Schlüssel

- Daten können beliebig komplex strukturiert sein (Einträge in Datenbanken, Dateien, Listen, ...)
- Solche Daten werden dann meist eindeutig **Schlüsseln** zugeordnet
  - (ganze) Zahlen (s. z.B. Kontonummer),
  - Strings (s. z.B. Telefonbuch),

  für die eine lineare Ordnung definiert ist.
- Behandelt werden dann meist nur die Schlüssel
  - als Werte der Knoten eines Baums, einer Liste, ...
  - beim Sortieren, Suchen, ...

Datentyp Menge

Menge als ADT

<u>type</u> set =

<u>sorts</u> T, boolean, set

<u>functions</u>

empty: $\rightarrow$ set	liefert $\varnothing$
isEmpty : set $\rightarrow$ boolean	$M = \varnothing$?
contains: T $\times$ set $\rightarrow$ boolean	$x \in M$?
add: T $\times$ set $\rightarrow$ boolean	$M = M \cup \{x\}$
remove: T $\times$ set $\rightarrow$ boolean	$M = M \setminus \{x\}$
...	

<u>end</u>.

Verwalten von Mengen

- **Besonderheit:** Eine Menge kann ein Element aus dem Grundbereich **T** höchstens einmal enthalten (**Element-Beziehung!!!**)

➢ vor dem Einfügen eines Elements muss immer überprüft werden, ob es bereits enthalten ist

➢ in Sequenzen (in jeder Implementierung): $\Theta(n)$

➢ *Effizienter?!*

Verwalten von Mengen durch Hashing

Hashtabellen

- Verwalten der Daten in einer Tabelle,
  je Datensatz (Element der Menge) eine Tabellenzeile
- Berechne die Tabellenzeile aus (Teilen der) Daten:
  aus dem Schlüssel

  → **Hashfunktion** (Schlüssel ⟼ Hashwert)
- Ziel: Tabelle fester (ausreichender) Größe benutzen

  → Suchen, Einfügen, Löschen in (fast) konstanter Zeit
  (*im Wesentlichen Zeit für Berechnung der Hashfunktion*)
- In der Praxis: Bei Überschreiten einer Grenze für die Anzahl der eingetragenen Elemente: *Vergrößerung* der Tabelle

Kollisionen

- Menge kann (theoretisch) unbegrenzt anwachsen
- Tabelle soll endliche Größe haben
- **Kollision:** Zuordnung desselben Hashwertes zu verschiedenen Elementen der Menge
- **Strategien zur Kollisionsbehandlung:**
  - **lineare Listen** von Elementen als Tabelleneinträge (Hashwert referenziert *Teilmenge*)
  - **Sondieren:** In bestimmter Schrittweite die nächste freie Tabellenzeile suchen (**lineares/quadratisches Sondieren)**
  - **doppeltes Hashing:** zweite Hashfunktion bestimmt Schrittweite beim Sondieren

Hashfunktion - Beispiele

- Schlüssel sind ganze Zahlen, Tabelle der Größe M
- Hashfunktion $\boldsymbol{h : \mathbb{Z} \longrightarrow \{0, \ldots, M\text{-}1\}}$
- *Beispiel:* $h(n) = n \bmod M$
- **lineares Sondieren:**

```
for k in range(0,M): (h(n) + k) % M
```

- **quadratisches Sondieren:**

```
for k in range(0,M): (h(n) + k**2) % M
```

(*bewirkt oft gleichmäßigere Verteilung der Tabelleneinträge*)

Bemerkungen zur Laufzeit

- bei kollisionsfreien Einträgen:
  - ***O*(1)** für Suchen, Einfügen, Löschen
  - dann aber viel freier Speicherplatz

- bei steigender Wahrscheinlichkeit von Kollisionen:
  - „Entartung“ bis zu ***O*(*n*)** bei linearer Sondierung oder Listen als Tabelleneinträge
  - Besonders vorteilhaft hier: doppeltes Hashing
    - ➢ zweiter Hashwert muss ungleich 0 und sollte teilerfremd zur Tabellengröße sein