# Algorithmen und Datenstrukturen

Vorrangwarteschlangen

Prof. Justus Piater, Ph.D.

7. April 2025

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

## Inhaltsverzeichnis

**1 Abstrakter Datentyp** **2**

**2 Implementierung mittels Liste** **7**

**3 Heap** **14**

**4 Implementierung mittels Heap** **18**

**5 *Bottom-Up Heap Construction*** **25**

**6 Sortieren mit einer Vorrangwarteschlange** **32**

**7 Zusammenfassung** **36**

Einführung

Aus einer Warteschlange (english *queue*) werden Elemente in der Reihenfolge ihres *Eintreffens* abgerufen, also *first-in, first-out* (FIFO). Aus einer ***Vorrangwarteschlange*** (englisch *priority queue*) werden die Elemente in der Reihenfolge ihrer *Priorität* abgerufen, im Allgemeinen unabhängig von der Reihenfolge ihres Eintreffens. Damit ist das Verhalten einer Vorrangwarteschlange weder FIFO noch LIFO.

Genau wie Warteschlangen funktionieren Vorrangwarteschlangen asynchron in dem Sinne, dass jederzeit und unkoordiniert Elemente eingefügt und abgerufen werden können. Ein Beispiel ist die Vergabe von Flugzeugsitzen an Stand-By-Passagiere. Interessenten können jederzeit eintreffen, und jeder freigegebene Sitzplatz wird an die Passagierin mit der höchsten Priorität vergeben, die z.B. von ihrem *Frequent-Flyer*-Status abhängt.

Ein anderes Beispiel ist die Notaufnahme eines Krankenhauses. Notfallpatienten können jederzeit eingeliefert werden. Immer, wenn Behandlungskapazität vorhanden ist, kommt die augenscheinlich dringendste behandlungsbedürftige Person an die Reihe.

Vorrangwarteschlangen lassen sich effizient auf Basis einer speziellen Baumstruktur implementieren, die sich *Halde* nennt, besser bekannt unter der englischen Bezeichnung *heap*.

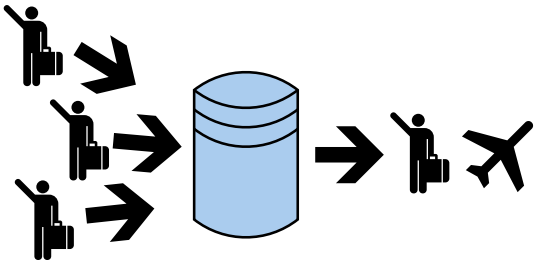
Es liegt nahe, Vorrangwarteschlangen zum Sortieren zu nutzen. Erst fügen wir alle zu sortierenden Elemente in eine Vorrangwarteschlange ein, und dann holen wir sukzessive jeweils das Element mit der höchsten Priorität wieder heraus. Verwendet man für

die Vorrangwarteschlange einen *Heap*, dann ergibt sich hiermit der bekannte HeapSort-Algorithmus, der zu den effizientesten generischen Sortieralgorithmen gehört.

# 1 Abstrakter Datentyp

Video 1 beginnt hier.

### Vorrangwarteschlangen *(priority queues)* [Slide 1]



Funktionsweise:

- Priorisierte Elemente kommen jederzeit an.
- Das Element mit der höchsten Priorität kann jederzeit abgerufen werden.
- Weder LIFO noch FiFO.
- Aus Nutzersicht: keine Ordnung, Positionen, Ränge, etc.

Weitere Beispiele?

- Stand-By-Passagiere
- Flugzeuge, die auf ihre Landung warten
- Notfallpatienten

### ADT: Vorrangwarteschlange [Slide 2]

```
insert(k, v)  // Creates an entry with key k and value v.

min()         // Returns (but does not remove) an entry (k, v)
              // having minimal k, or null if the priority queue is empty.

removeMin()   // Returns and removes an entry (k, v)
              // having minimal k, or null if the priority queue is empty.

size()        // Returns the number of elements in the list .

isEmpty()     // Returns a boolean indicating whether the list is empty.
```

Definition

Sehen wir hier die Definition des abstrakten Datentyps ***Vorrangwarteschlange***. Die beiden entscheidenden Methoden sind `insert()` und `removeMin()`. `insert(k, v)` fügt den Wert `v` unter dem Schlüssel `k` in die Vorrangwarteschlange ein. Mit anderen Worten, die Vorrangwarteschlange enthält anschließend ein neues *Element*, das aus dem Schlüssel `k` und dem Wert `v` besteht.

`removeMin()` entfernt das Element mit *minimalem* Schlüssel aus der Vorrangwarteschlange, und liefert es zurück.

Ferner gibt es noch die Methode `min()`, die das Element mit minimalem Schlüssel zurückliefert, ohne es aus der Vorrangwarteschlange zu entfernen. Wiederholte Aufrufe von `min()`, ohne Aufrufe von `removeMin()` oder `insert()` dazwischen, liefern also jedesmal dasselbe Element zurück.

Vervollständigt wird unser abstrakter Datentyp wie immer durch die Methoden `size()` und `isEmpty()`.

## Java-Interfaces [Slide 3]

```
public interface Entry<K,V> {
  K getKey();   // Returns the key stored in this entry.
  V getValue(); // Returns the value stored in this entry.
}

public interface PriorityQueue<K,V> {
  int size();       // Returns the number of items in the priority queue.
  boolean isEmpty(); // Tests whether the priority queue is empty.

  /* Inserts a key-value pair and returns the entry created;
   * returns the entry storing the new key-value pair:        */
  Entry<K,V> insert(K key, V value) throws IllegalArgumentException;

  // Returns (but does not remove) an entry with minimal key:
  Entry<K,V> min();

  // Removes and returns an entry with minimal key:
  Entry<K,V> removeMin();
}
```

## Schlüssel [Slide 4]

Die Prioritäten der Elemente werden jeweils durch ihren ***Schlüssel*** festgelegt.

Der Schlüssel kann vom Element selbst abgeleitet oder völlig separat definiert werden, je nach Anwendung.

Beispiele?

- (Ankunftszeit, Frequent-Flyer-Status, bezahlter Flugpreis)
- (Ankunftszeit, verbliebene Treibstoffmenge, Verspätung)
- (Ankunftszeit, Verletzungsgrad)

Erklärung

Wie ermittelt eine Vorrangwarteschlange die Priorität eines Elements? Zu diesem Zweck bestehen die Elemente jeweils aus einem Schlüssel-Wert-Paar. Der Schlüssel ist eine Datenstruktur, anhand derer die Prioritäten verschiedener Elemente untereinander verglichen werden können.

Im Beispiel der Stand-By-Passagiere kann der Schlüssel beispielsweise aus dem Frequent-Flyer-Status, dem bezahlten Ticketpreis und der Ankunftszeit bestehen. In der Notaufnahme beinhaltet der Schlüssel sicher die Schwere einer Verletzung und wiederum auch die Ankunftszeit.

## Totalordnung [Slide 5]

**Definition:** Eine binäre Relation $\leq$ mit den folgenden Eigenschaften für jedes $k$:

**Totalität:** $k_1 \leq k_2$ oder $k_2 \leq k_1$ *(impliziert Reflexivität $k \leq k$).*

**Antisymmetrie:** Wenn $k_1 \leq k_2$ und $k_2 \leq k_1$, dann $k_1 = k_2$.

**Transitivität:** Wenn $k_1 \leq k_2$ und $k_2 \leq k_3$, dann $k_1 \leq k_3$ *(impliziert die Existenz von $k_{\text{min}} \leq k$).*

Definition

Damit sich aus paarweisen Vergleichen von Schlüsseln eine widerspruchsfreie Rangfolge ergibt, muss unsere Vergleichsoperation eine sogenannte ***Totalordnung*** darstellen. Eine Totalordnung ist per Definition eine binäre Relation, hier $\leq$ genannt, mit den drei Eigenschaften ***Totalität***, ***Antisymmetrie*** und ***Transitivität***.

Totalität besagt, dass zwei gegebene Schlüssel $k_1$ und $k_2$ immer miteinander verglichen werden können und dass entweder $k_1 \leq k_2$ oder $k_2 \leq k_1$ ist, oder beides. Dies impliziert insbesondere, dass ein Schlüssel mit sich selbst verglichen werden kann und dass dieser Vergleich affirmativ ausfällt.

Antisymmetrie besagt, dass $k_1$ und $k_2$ aus der Sicht der Relation $\leq$ identisch sind, falls sowohl $k_1 \leq k_2$ als auch $k_2 \leq k_1$ ist.

Transitivität besagt, dass, wenn $k_1 \leq k_2$ und $k_2 \leq k_3$ sind, dann muss auch $k_1 \leq k_3$ sein. Die Transitivität sorgt dafür, dass die Ordnung keine Widersprüche enthalten kann, wie z.B. Zykel von $\leq$-Relationen.

Wenn man nun alle Elemente einer Menge paarweise und widerspruchsfrei miteinander vergleichen kann, dann folgt daraus, dass diese Menge ein Minium enthält – oder ggf. mehrere minimale Elemente, die dann jedoch aus Sicht des verwendeten Vergleichsoperators identisch sind.

Eine Totalordnung über den Schlüsseln garantiert also, dass wir einen minimalen Schlüssel in der Vorrangwarteschlange identifizieren können.

## Vergleichen von Schlüsseln [Slide 6]

- Vorrangwarteschlange separat für jeden Schlüssel-Datentyp implementieren?
- Schlüssel-Datentyp mit Vergleichs-Methode ausstatten?

  Numerischer oder lexikografischer Vergleich?

  Vergleich von $x$ oder $y$?

Erklärung

Wie implementieren wir einen solchen Vergleichsoperator für unsere Vorrangwarteschlange?

Wir könnten ihn z.B. fest in unsere Vorrangwarteschlange einbauen. Dann müssten wir allerdings für jeden Schlüssel-Datentyp eine separate Vorrangwarteschlange implementieren.

Also bietet es sich vielleicht an, einen abstrakten Datentyp Schlüssel zu definieren und mit einer Vergleichsmethode auszustatten. Damit könnte man Schlüssel-Datenstrukturen und zugehörige Vergleichsmethoden gemeinsam implementieren.

Dies hat allerdings den Nachteil, dass eine bestimmte Schlüssel-Datenstruktur nicht unbedingt einen bestimmten Vergleichsoperator impliziert. Es ist durchaus denkbar, dass man dieselbe Schlüssel-Datenstruktur in verschiedenen Vorrangwarteschlangen mit *unterschiedlichen* Vergleichsoperatoren verwenden möchte.

Nehmen wir z.B. an, unser Schlüssel ist eine Zeichenkette, was in der Praxis sicher häufig vorkommt. Wie vergleicht man zwei Zeichenketten? Lexikografisch vielleicht, aber soll

die Groß- und Kleinschreibung ignoriert werden oder nicht? Sollen Leer- und Satzzeichen berücksichtigt werden oder nicht? Die Antwort wird in verschiedenen Anwendungen unterschiedlich ausfallen.

Oder verschiedene Fluggesellschaften priorisieren ihre Stand-By–Passagiere anhand derselben Kriterien, aber gewichten diese Kriterien unterschiedlich.

## ADT: Komparator [Slide 7]

```
compare(a, b) // Returns an integer i with
              // i < 0 if a < b,
              // i = 0 if a = b,
              // i > 0 if a > b.
              // Error if a and b cannot be compared.
```

Siehe auch java.util.Comparator.

Definition

Daher wird üblicherweise der Vergleichsoperator eigenständig definiert, außerhalb von Schlüssel und Vorrangwarteschlange. Hierzu führen wir den abstrakten Datentyp Komparator ein, der lediglich die Methode `compare()` enthält. `compare(a, b)` vergleicht die Schlüssel `a` und `b`, und liefert einen Wert <0 zurück falls der Schlüssel `a` kleiner ist als der Schlüssel `b`, einen Wert >0 falls `a` größer ist als `b`, oder den Wert 0 falls beide Schlüssel gleichwertig sind.

Nun haben wir alle Komponenten einer Vorrangwarteschlange beisammen. Unsere Daten bestehen aus Schlüssel-Wert-Paaren, wobei der Wert uns hier nicht näher interessiert. Für unsere Schlüssel-Datenstruktur benötigen wir einen Komparator, der eine Totalordnung über unseren Schlüsseln definiert. Dieser Komparator wird dann von der Vorrangwarteschlange verwendet. Welche Methoden den Komparator in welcher Weise verwenden, hängt von der Datenstruktur ab, mittels derer wir unsere Vorrangwarteschlange implementieren.

## Zwei Komparatoren für Zeichenketten [Slide 8]

```java
public class StringLengthComparator implements Comparator<String> {
  /** Compares two strings according to their lengths. */
  public int compare(String a, String b) {
    if (a.length() < b.length()) return -1;
    else if (a.length() == b.length()) return 0;
    else return 1;
  }
}

public class StringLexComparator implements Comparator<String> {
  /** Compares two strings lexicogrpahically . */
  public int compare(String a, String b) {
    return a.compareTo(b);
  }
}
```

## AbstractPriorityQueue [Slide 9]

```
/**
 * An abstract base class to ease the implementation of the PriorityQueue interface.
 *
 * The base class provides four means of support:
 * 1) It defines a PQEntry class as a concrete implementation of the
 *    entry interface
 * 2) It provides an instance variable for a general Comparator and a
 *    protected method, compare(a, b), that makes use of the comparator.
 * 3) It provides a boolean checkKey method that verifies that a given key
 *    is appropriate for use with the comparator
 * 4) It provides an isEmpty implementation based upon the abstract size() method.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public abstract class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {
  //------------------ nested PQEntry class ------------------
  /**
   * A concrete implementation of the Entry interface to be used within
   * a PriorityQueue implementation.
   */
  protected static class PQEntry<K,V> implements Entry<K,V> {
    private K k; // key
    private V v; // value

    public PQEntry(K key, V value) {
      k = key;
      v = value;
    }

    // methods of the Entry interface
    public K getKey() { return k; }
    public V getValue() { return v; }

    // utilities not exposed as part of the Entry interface
    protected void setKey(K key) { k = key; }
    protected void setValue(V value) { v = value; }
  } //------------- end of nested PQEntry class -------------

  // instance variable for an AbstractPriorityQueue
  /** The comparator defining the ordering of keys in the priority queue. */
  private Comparator<K> comp;

  /**
   * Creates an empty priority queue using the given comparator to order keys.
   * @param c comparator defining the order of keys in the priority queue
   */
  protected AbstractPriorityQueue(Comparator<K> c) { comp = c; }
```

```java
  /** Creates an empty priority queue based on the natural ordering of its keys. */
  protected AbstractPriorityQueue() { this(new DefaultComparator<K>()); }

  /** Method for comparing two entries according to key */
  protected int compare(Entry<K,V> a, Entry<K,V> b) {
    return comp.compare(a.getKey(), b.getKey());
  }

  /** Determines whether a key is valid. */
  protected boolean checkKey(K key) throws IllegalArgumentException {
    try {
      return (comp.compare(key,key) == 0); // see if key can be compared to itself
    } catch (ClassCastException e) {
      throw new IllegalArgumentException("Incompatible␣key");
    }
  }

  /**
   * Tests whether the priority queue is empty.
   * @return true if the priority queue is empty, false otherwise
   */
  @Override
  public boolean isEmpty() { return size() == 0; }
}
```

### Quiz [Slide 10]

Eine Vorrangwarteschlange darf mehrere identische Schlüssel enthalten.

A: wahr

B: falsch

D: weiß nicht

## 2 Implementierung mittels Liste

Video 2 beginnt hier.

### Implementierung mittels Liste [Slide 11]

Verkettete Liste oder ArrayList, unsortiert oder sortiert: Komplexität der Methoden?

## Implementierung mittels Liste [Slide 12]

Verkettete Liste oder ArrayList, unsortiert oder sortiert: Komplexität der Methoden?

| **Methode** | **Unsortiert** | **Sortiert** |
|---|---|---|
| `size()` | $O(1)$ | $O(1)$ |
| `isEmpty()` | $O(1)$ | $O(1)$ |
| `insert()` | $O(1)$ | $O(n)$ |
| `min()` | $\Theta(n)$ | $O(1)$ |
| `removeMin()` | $\Theta(n)$ | $O(1)$ |

Erklärung

Einfache Datenstrukturen für den abstrakten Datentyp Vorrangwarteschlange sind Listen. Was sind die asymptotischen Laufzeiten der 5 Methoden der Vorrangwarteschlange, abhängig von der Anzahl $n$ der Elemente in der Vorrangwarteschlange, wenn wir sie auf Basis einer *verketteten* Liste implementieren?

`size()` und `isempty()` können wir wie immer leicht in konstanter Zeit implementieren, indem wir eine Variable $n$ vorsehen, die von `insert()` inkrementiert und von `removeMin()` dekrementiert wird.

Betrachten wir zunächst eine *unsortierte* verkettete Liste. Hier muss `insert()` das neue Element lediglich am Kopf oder am Ende der Liste anhängen, was sich in konstanter Zeit machen lässt.

Die Methode `min()` muss nun jedoch die gesamte Liste durchgehen, um den minimalen Schlüssel zu finden. Dies benötigt eine Zeit linear in $n$. Also ist die asymptotische Laufzeit von `min()` $\Theta(n)$.

`removeMin()` muss ebenfalls den minimalen Schlüssel finden, und zusätzlich das zugehörige Element aus der Liste entfernen. Dieses Entfernen nimmt bei einer verketteten Liste lediglich konstante Zeit in Anspruch. Daher ist `removeMin()` ebenfalls $\Theta(n)$.

Was sind die Laufzeiten bei einer *sortierten* verketteten Liste? Hier muss die Methode `insert(k, v)` die Liste traversieren, bis ein Schlüssel gefunden wird, der größer oder gleich `k` ist, und das neue Element vor diesem in die Liste einfügen. Die Suche wird im Schnitt etwa die halbe Liste traversieren müssen. Also ist `insert()` $O(n)$.

Im Gegenzug können `min()` und `removeMin()` jederzeit den minimalen Schlüssel in konstanter Zeit am Kopf der Liste abgreifen und, im Falle von `removeMin()`, entfernen.

Damit verhalten sich die unsortierte und die sortierte verkettete Liste komplementär: Bei der unsortierten Liste ist das Einfügen effizient und das Herausholen ineffizient, und bei der sortierten Liste ist umgekehrt das Einfügen ineffizient und das Herausholen effizient.

Eine Sequenz von $n$ Einfüge- und Herausholoperationen ist also in beiden Fällen $nO(n)$, also $O(n^2)$.

Was sind die Laufzeiten dieser Methoden, wenn wir statt einer verketteten Liste eine ArrayList verwenden?

`insert()` in eine *unsortierte* ArrayList bleibt konstant. Ebenso bleibt das Finden des minimalen Schlüssels per sequenzieller Suche $\Theta(n)$. Das Entfernen eines Elements zieht das Aufrücken im Schnitt der halben Anzahl der Elemente nach sich, dauert also $O(n)$ Zeit. Da jedoch bereits das Auffinden des minimalen Elements $\Theta(n)$ Zeit benötigt, bleibt die asymptotische Laufzeit von `removeMin()` $\Theta(n)$.

Bei der *sortierten* ArrayList muss die Zelle gefunden werden, in die das neue Element einsortiert werden muss. Im Gegensatz zur verketteten Liste erlaubt die ArrayList den direkten Zugriff auf jedes Element über seinen Index. Daher können wir die sequenzielle Suche durch eine Binärsuche ersetzen und die Laufzeit der Suche von $O(n)$ auf $O(\log n)$ reduzieren. Allerdings muss für das neue Element Platz geschaffen werden, indem im Schnitt die Hälfte der Elemente im Array verschoben wird. Dieses Verschieben nimmt also $O(n)$ Zeit in Anspruch. Daher bleibt die Gesamtlaufzeit von `insert()` $O(n)$.

`removeMin()` können wir in konstanter Zeit implementieren, indem wir ein zyklisches

Array verwenden.

Wir schlussfolgern also, dass die asymptotischen Laufzeiten bei verketteter Liste und ArrayList dieselben sind. Dies ist der Grund, warum auf dieser Slide lediglich von einer Liste die Rede ist und offen bleibt, durch welche Datenstruktur Instanzen dieses abstrakten Datentyps implementiert werden, durch verkettete Listen oder Arrays.

UnsortedPriorityQueue [Slide 13]

```java
/**
 * An implementation of a priority queue with an unsorted list .
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public UnsortedPriorityQueue() { super(); }

  /**
   * Creates an empty priority queue using the given comparator to order keys.
   * @param comp comparator defining the order of keys in the priority queue
   */
  public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /**
   * Returns the Position of an entry having minimal key.
   * This should only be called on a nonempty priority queue
   * @return Position of entry with minimal key
   */
  private Position<Entry<K,V>> findMin() { // only called when nonempty
    Position<Entry<K,V>> small = list.first();
    for (Position<Entry<K,V>> walk : list.positions())
      if (compare(walk.getElement(), small.getElement()) < 0)
        small = walk;    // found an even smaller key
    return small;
  }

  /**
   * Inserts a key-value pair and returns the entry created.
   * @param key     the key of the new entry
   * @param value   the associated value of the new entry
   * @return the entry storing the new key-value pair
   * @throws IllegalArgumentException if the key is unacceptable for this queue
   */
  @Override
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key); // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    list.addLast(newest);
    return newest;
  }

  /**
   * Returns (but does not remove) an entry with minimal key.
```

```java
   * @return entry having a minimal key (or null if empty)
   */
  @Override
  public Entry<K,V> min() {
    if (list.isEmpty()) return null;
    return findMin().getElement();
  }

  /**
   * Removes and returns an entry with minimal key.
   * @return the removed entry (or null if empty)
   */
  @Override
  public Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(findMin());
  }

  /**
   * Returns the number of items in the priority queue.
   * @return number of items
   */
  @Override
  public int size() { return list.size(); }
}
```

## SortedPriorityQueue [Slide 14]

```java
/**
 * An implementation of a priority queue with a sorted list.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public SortedPriorityQueue() { super(); }

  /**
   * Creates an empty priority queue using the given comparator to order keys.
   * @param comp comparator defining the order of keys in the priority queue
   */
  public SortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /**
   * Inserts a key-value pair and returns the entry created.
   * @param key     the key of the new entry
   * @param value   the associated value of the new entry
   * @return the entry storing the new key-value pair
   * @throws IllegalArgumentException if the key is unacceptable for this queue
   */
  @Override
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key); // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    Position<Entry<K,V>> walk = list.last();
    // walk backward, looking for smaller key
    while (walk != null && compare(newest, walk.getElement()) < 0)
      walk = list.before(walk);
    if (walk == null)
      list.addFirst(newest);              // new key is smallest
    else
      list.addAfter(walk, newest);        // newest goes after walk
    return newest;
  }

  /**
   * Returns (but does not remove) an entry with minimal key.
   * @return entry having a minimal key (or null if empty)
   */
  @Override
  public Entry<K,V> min() {
    if (list.isEmpty()) return null;
    return list.first().getElement();
```

```java
  }

  /**
   * Removes and returns an entry with minimal key.
   * @return the removed entry (or null if empty)
   */
  @Override
  public Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(list.first());
  }

  /**
   * Returns the number of items in the priority queue.
   * @return number of items
   */
  @Override
  public int size() { return list.size(); }
}
```

## Quiz [Slide 15]

Warum durchläuft `insert()` die Liste rückwärts und nicht vorwärts?

A: Das ist effizienter.

B: Dadurch liefert `removeMin()` identische Schlüssel in der Reihenfolge ihres Eintreffens.

C: Das vereinfacht den Code.

D: weiß nicht

# 3 Heap

Video 3 beginnt hier.

Erklärung

Mit einer Liste als Datenstruktur laufen die Methoden des abstrakten Datentyps Vorrangwarteschlange in einer Zeit von entweder $O(1)$ oder $O(n)$. Damit nimmt eine Sequenz von $n$ Einfüge- und Entfernungsoperationen eine Gesamtzeit von $O(n^2)$ in Anspruch.

Können wir vielleicht durch eine andere Datenstruktur einen Kompromiss erzielen, so dass sowohl Einfüge- als auch Entfernungsoperationen jeweils unterhalb einer Laufzeit von $O(n)$ bleiben, beispielsweise $O(\log n)$? In diesem Fall könnten wir die Gesamtlaufzeit von $n$ Einfüge- und Entfernungsoperationen von $O(n^2)$ auf $O(n \log n)$ reduzieren!

Dies ist in der Tat möglich, und zwar mit Hilfe einer speziellen Baumstruktur, die als ***Heap*** bekannt ist, zu deutsch *Haufen* oder *Halde*.

## Kompromiss der Laufzeiten [Slide 16]

Mit einer Liste laufen die Methoden in einer Zeit von entweder $O(1)$ oder $O(n)$.

Können wir einen Kompromiss finden, z.B. eine Laufzeit von jeweils $O(\log n)$?

Ja – indem wir die Liste durch eine bestimmte Baumstruktur ersetzen!

## *Heap* (Haufen, Halde) [Slide 17]

**Definition:** Ein (binärer) ***Heap*** ist ein *Binärbaum* $T$ mit den folgenden Eigenschaften:

**Heap-Ordnung:** Der *Schlüssel* jedes Knotens (außer der Wurzel) ist *größer oder gleich* dem seines Elternknotens.

***Anmerkung***

- Die Schlüssel entlang jedes von der Wurzel ausgehenden Pfades bilden eine nicht-absteigende Folge.
- Die Wurzel enthält einen minimalen Schlüssel.

**Vollständigkeit:** Ein Binärbaum mit Höhe $h$ ist ***vollständig***, wenn er die folgenden Eigenschaften besitzt:

- Die Zeilen $0, 1, \ldots, h-1$ enthalten jeweils eine *maximale* Anzahl von Knoten ($2^i$ Knoten in Zeile $i$).
- In Zeile $h-1$ befinden sich alle internen Knoten *links* der externen Knoten (d.h. eine Inorder-Traversierung besucht die internen vor den externen Knoten in Zeile $h-1$).
- In Zeile $h-1$ existiert höchstens ein Knoten mit *einem Kind*. Falls er existiert, liegen alle anderen internen Knoten dieser Zeile links von ihm, und sein Kind ist ein linkes Kind.

Ein Baum mit $n$ Knoten ist *vollständig* genau dann, wenn seine *zeilenweise Nummerierung* lückenlos ist, d.h., wenn seine Positionen von 0 bis $n-1$ nummeriert sind.

**Definition:** Der ***letzte Knoten*** eines Haufens ist der am weitesten rechts liegende Knoten in Zeile $h$.

Definition

Wir speichern unsere Schlüssel und zugehörigen Werte in einem Binärbaum. Jeder Knoten des Binärbaums enthält ein Schlüssel-Wert-Paar. Diese Schlüssel-Wert-Paare sind im Baum so angeordnet, dass kein Schlüssel kleiner ist als der Schlüssel seines Elternknotens. Diese Eigenschaft bezeichnet man als ***Heap-Ordnung***.

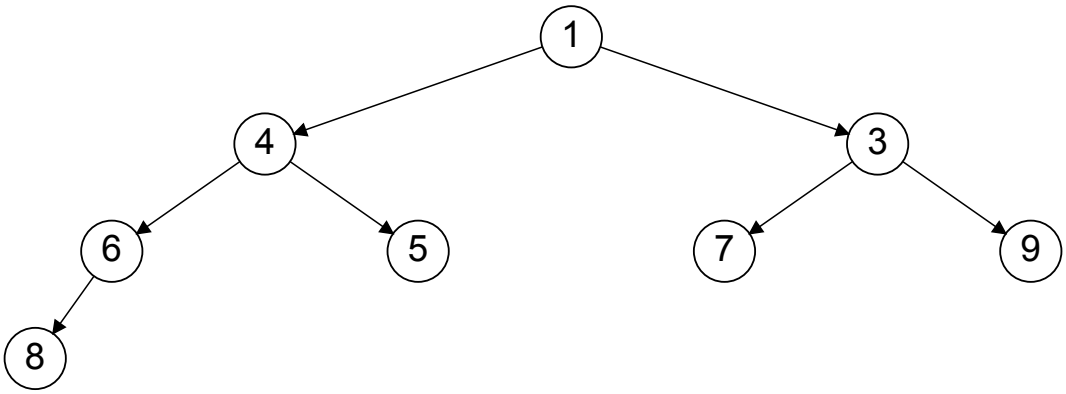
Aus der Heap-Ordnung folgt, dass sich ein minimaler Schlüssel in der Wurzel befindet, und dass entlang jedes Pfades von der Wurzel die Schlüssel eine nicht-absteigende Sequenz bilden.

Unser Binärbaum muss eine zweite wichtige Eigenschaft erfüllen, nämlich die sogenannte ***Vollständigkeit***. Ein Binärbaum ist vollständig, wenn jede seiner Zeilen maximal gefüllt ist, bis auf die letzte, und wenn in der letzten Zeile alle Knoten linksbündig angeordnet sind. Mit anderen Worten, ein Baum mit $n$ Knoten ist vollständig genau dann, wenn seine zeilenweise Nummerierung lückenlos ist, d.h., wenn seine Positionen von 0 bis $n-1$ durchnummeriert sind.

Einen Binärbaum mit diesen beiden Eigenschaften, der Heap-Ordnung und der Vollständigkeit, bezeichnet man als ***Heap*** oder genau genommen als *binären Heap*. Die populärsten Datenstrukturen für Vorrangwarteschlangen sind Heaps.

Für die Beschreibung unserer Algorithmen auf Heaps benötigen wir noch eine weitere Definition: Der ***letzte Knoten*** eines Heaps ist der am weitesten rechts liegende Knoten seiner letzten Zeile, d.h., Knoten $n-1$ in seiner zeilenweisen Nummerierung.

## Beispiel eines Heaps [Slide 18]




Beispiel

Hier sehen wir ein Beispiel eines Heaps. In jedem Knoten steht der dort gespeicherte Schlüssel. Auf die Darstellung des jeweils zugehörigen Wertes verzichten wir hier und im Folgenden, da die Werte für uns ohne Belang sind und immer mit dem Schlüssel zu einem Element verbunden bleiben.

Wir erkennen sofort die *Vollständigkeit* dieses Heaps: Jede Zeile ist voll bis auf die letzte, und in der letzten liegt der einzige Knoten ganz links.

Die *Heap*-Ordnung können wir einfach überprüfen: Für jeden Knoten ist der Schlüssel mindestens so groß wie der seines Elternknotens.

In diesem Beispiel sind alle Schlüssel verschieden. Eine Vorrangwarteschlange kann jedoch problemlos mehrere gleiche Schlüssel enthalten. Daher ist für die Heap-Ordnung ausreichend, dass der Schlüssel jedes Knotens größer als *oder gleich* dem des Elternknotens sein muss.

## Quiz [Slide 19]

Ein Heap ist ein binärer Suchbaum.

A: wahr

B: falsch

D: weiß nicht

## Die Höhe eines Heaps [Slide 20]

**Proposition:** Ein Heap $T$ mit $n$ Elementen hat die Höhe $h = \lfloor \log_2 n \rfloor$.

**Beweis:** Da $T$ vollständig ist, enthalten die Zeilen 0 bis $h-1$ genau $1+2+4+\cdots+2^{h-1} = 2^h - 1$ Knoten, und die Anzahl der Knoten in Zeile $h$ ist *mindestens* 1 und *höchstens* $2^h$. Daher,

$$\begin{array}{rcccl} 2^h - 1 + 1 & = & 2^h & \leq & n \leq 2^{h+1} - 1 = 2^h - 1 + 2^h \\ \log_2(n+1) - 1 & & & \leq & h \leq \log_2 n \end{array}$$

### *Anmerkung*

Schaffen wir es, `insert()` und `removeMin()` in linearer Zeit in der Höhe des Heaps zu implementieren, dann ist die Gesamtlaufzeit der Heap-Operationen logarithmisch in $n$!

Erklärung

Die Laufzeiten unserer Algorithmen für `insert()` und `removeMin()` werden von der Höhe unseres Heaps bestimmt werden. Hier weisen wir nach, dass die Höhe $h$ eines Heaps tatsächlich logarithmisch in der Anzahl $n$ seiner Knoten ist.

Bei unserer früheren Betrachtung allgemeiner Eigenschaften von Binärbäumen haben wir bereits bewiesen, dass ein Binärbaum der Höhe $h$ maximal $2^{h+1}-1$ Knoten enhalten kann. Wegen der Vollständigkeitseigenschaft des Heaps enthalten seine ersten $h-1$ Zeilen also genau $2^h - 1$ Knoten.

Damals ebenfalls bereits bewiesen haben wir die Tatsache, dass die letzte Zeile eines Binärbaums der Höhe $h$ zwischen 1 und $2^h$ Knoten enthalten kann.

Ein Heap enthält also mindestens $2^h - 1 + 1 = 2^h$ Knoten, und höchstens $2^h - 1 + 2^h = 2^{h+1} - 1$ Knoten.

Wenn wir diese beiden Ungleichungen durch Logarithmieren nach $h$ auflösen, dann stellen wir fest, dass $h$ höchstens $\log_2 n$ ist, und mindestens $\log_2(n+1) - 1$.

Da das Argument des linken Logarithmus *größer* ist als das Argument des rechten Logarithmus, ist klar, dass der linke Wert nach Abzug der 1 um *weniger als 1* unter dem rechten liegen muss. Da die Höhe $h$ eine ganze Zahl sein muss, folgt daraus unsere Proposition: Ein Heap $T$ mit $n$ Elementen hat genau die Höhe $h = \log_2 n$, abgerundet auf die nächste ganze Zahl.

Die Heap-Ordnung erlaubt es uns, für `insert()` und `removeMin()` Algorithmen zu entwickeln, die bei jedem Aufruf lediglich einen Pfad zwischen Wurzel und Blatt betrachten. Damit können wir beide in $O(h)$, also in $O(\log n)$ implementieren!

# 4 Implementierung mittels Heap

Video 4 beginnt hier.

## `insert()` [Slide 21]

1. Füge das neue Element am *Ende* des Baums (so dass er *vollständig* bleibt) als Position $p$ an.
2. `upheap(`$p$`)`: Lasse das neue Element Richtung Wurzel ***aufsteigen*** (durch sukzessives Vertauschen mit dem Elternelement), bis die Heap-Ordnung wieder hergestellt ist.

Laufzeit?

Animation

Schauen wir uns nun Algorithmen für den abstrakten Datentyp Vorrangwarteschlange basierend auf der Heap-Datenstruktur an.

Der Algorithmus für die Methode `insert()` ist ganz einfach. Zuerst fügen wir das neue Element in einem neuen Knoten in den Baum ein, in unserem Beispiel mit dem Schlüssel 2. Wegen der Vollständigkeitseigenschaft ist es klar, wo dieser neue Knoten $p$ angehängt wird.

Damit haben wir möglicherweise die Heap-Ordnung verletzt. Um diese wieder herzustellen, folgen wir einer Prozedur, die wir `upheap(`$p$`)` nennen.

Da die Heap-Ordnung eine Relation zwischen den Schlüsseln eines Knotens und seines Elternknotens definiert, schauen wir uns den Schlüssel des Elternknotens von $p$ an. Ist dieser größer, dann stellen wir die Heap-Ordnung lokal wieder her, indem wir die beiden Elemente vertauschen. Hier ist die 6 größer als die 2; also vertauschen wir diese beiden Elemente.

Dadurch verkleinert sich der Schlüssel des Elternknotens, womit dieser nun möglicherweise kleiner ist als der Schlüssel *seines* Elternknotens. Ist das der Fall, so wie hier mit der 4, dann vertauschen wir wiederum diese beiden Elemente.

Dies setzen wir fort, bis entweder die Wurzel erreicht ist oder keine Vertauschung notwendig ist. In diesem Fall können wir aufhören, denn oberhalb des zuletzt vertauschten Elements haben wir den Baum ja nicht verändert.

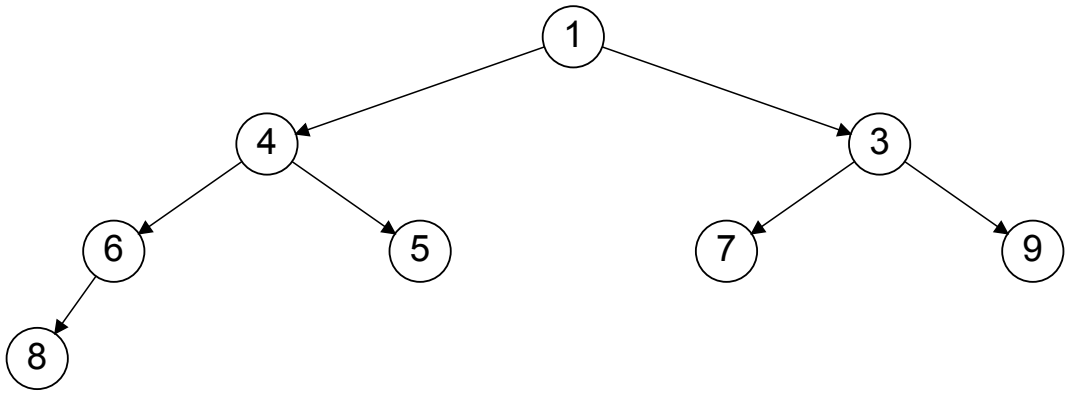
Erklärung

Da der Algorithmus lediglich einen einzigen Pfad von einem Blatt Richtung Wurzel beschreitet, kann er höchstens $\log n$ Knoten besuchen. Da jede einzelne Operation inklusive des Vertauschens von Elementen in konstanter Zeit durchgeführt werden kann, ist die asymptotische Laufzeit von `insert()` $O(\log n)$.

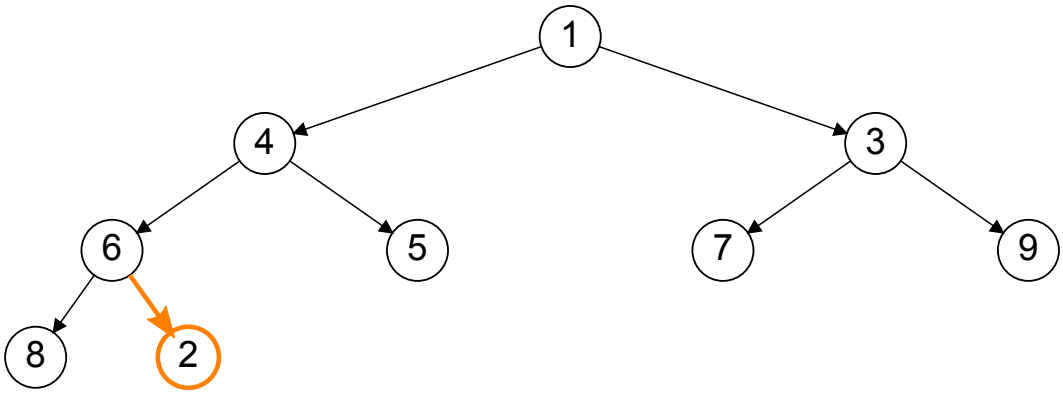
Obwohl der Algorithmus lediglich einen einzigen Pfad von einem Blatt Richtung Wurzel beschreitet, stellt er die Heap-Ordnung *global* wieder her. Der Effekt einer Sequenz von Vertauschungen ist, dass das neue Element nach oben ***aufsteigt***. Die Reihenfolge aller anderen Schlüssel entlang dieses Pfades ändert sich dabei jedoch nicht. Daher ist es unmöglich, dass eine Vertauschung die Heap-Ordnung *unterhalb* der vertauschten Elemente verletzt.

Bei einer Vertauschung wird das aufsteigende Element zum neuen Elternknoten seines ehemaligen Elternknotens. Er wird damit auch zum neuen Elternknoten seines *anderen* Kindknotens. Kann dies *dort* die Heap-Ordnung verletzen? Nein, denn durch die Vertauschung hat sich der Schlüsselwert seines Elternknotens weiter vermindert. Nur eine Vergrößerung kann eine bestehende Heap-Ordnung verletzen.

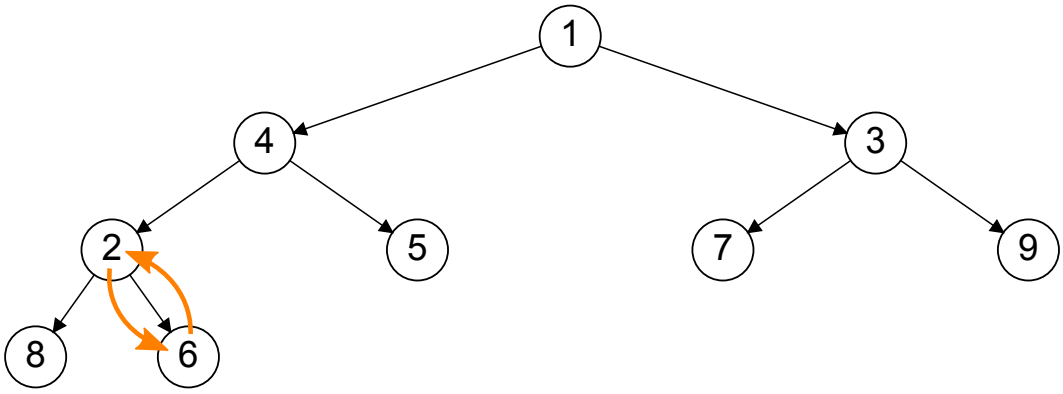
**[Slide 22]**



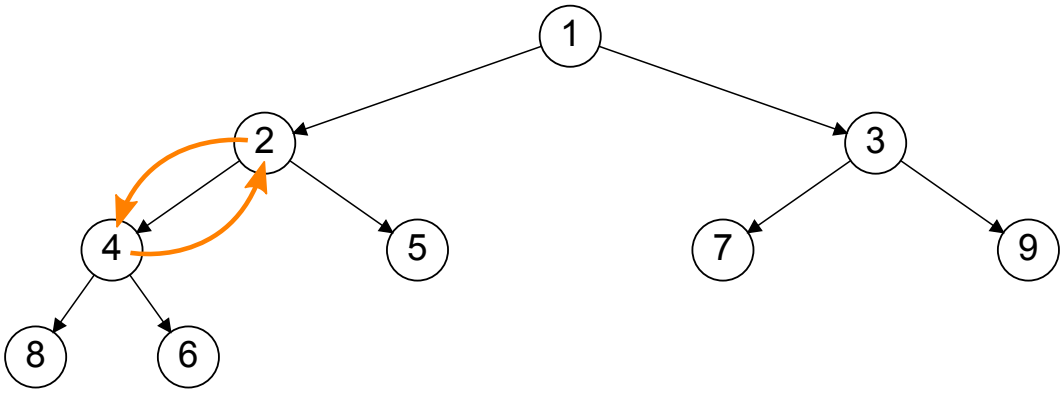

**[Slide 23]**




**[Slide 24]**



**[Slide 25]**




**`removeMin()` [Slide 26]**

1. Entferne und liefere das Element der Wurzel.
2. Verschiebe das Element des *letzten* Knotens in die Wurzel $r$, und eliminiere den letzten Knoten.
3. `downheap(`$r$`)`: Lasse das Element des Knotens $r$ ***absinken***, bis die Heap-Ordnung wieder hergestellt ist:
   (a) Hat $r$ kein rechtes Kind, sei $s$ das linke Kind von $r$.
   Andernfalls sei $s$ das Kind von $r$ mit *niedrigerem Schlüssel*.
   (b) Ist $k(r) > k(s)$, vertausche $r$ und $s$, und lasse $r$ mittels `downheap(`$r$`)` weiter absinken.

Laufzeit?

Animation

Die Idee von `removeMin()` folgt dem gleichen Prinzip wie `insert()`. `removeMin()` entfernt zunächst das Element aus der Wurzel $r$, ersetzt es durch das Element des letzten Knotens des Baums, und entfernt den nun leeren letzten Knoten. Anschließend wird mittels einer Prozedur `downheap(`$r$`)` die Heap-Ordnung global wiederhergestellt, indem das neue Element sukzessive mit dem Element eines Kindknotens vertauscht wird.

Hier stellt sich eine neue Frage. Beim Aufsteigen in `insert()` gab es nur einen Elternknoten, aber beim ***Absinken*** hier in `removeMin()` haben wir meist zwei Kindknoten zur Auswahl. In unserem Beispiel ist die 8 größer als sowohl die 4 als auch die 3. Mit welchem vertauschen wir das Element? Oder ist das egal?

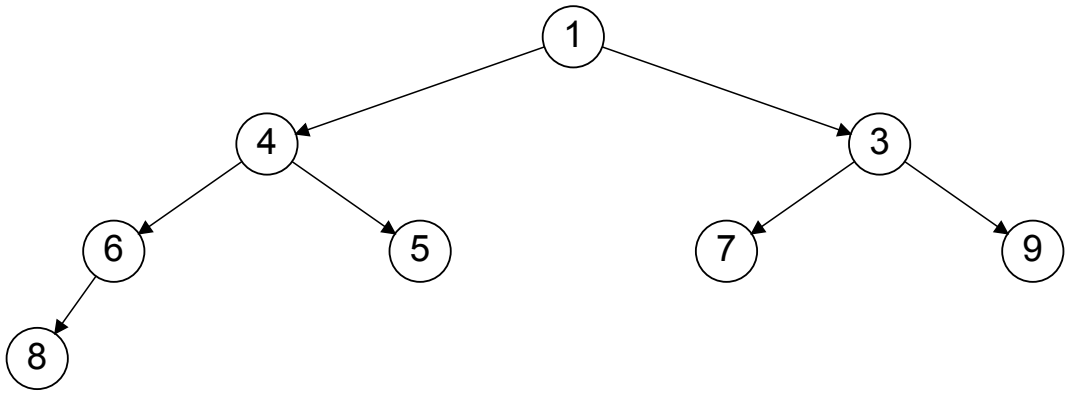
Falls die Schlüssel der beiden Kindknoten identisch sind, dann ist es tatsächlich egal. Sinde diese jedoch verschieden, so wie hier, dann müssen wir aufpassen, denn einer dieser Geschwisterschlüssel wird durch die Vertauschung zum Elternschlüssel des anderen. Da die Heap-Ordnung vorschreibt, dass Elternschlüssel nicht größer sein dürfen als die Schlüssel ihrer Kinder, müssen wir also beim Absinken immer das Kind mit dem *kleineren* Schlüssel zum Elternknoten des anderen machen, hier die 3.

Auf diese Weise vertauschen wir sukzessive unser absinkendes Element mit dem Kindelement kleineren Schlüssels, bis kein Kind mehr einen kleineren Schlüssel hält als das absinkende Element. An diesem Punkt ist die Heap-Ordnung global wieder hergestellt.

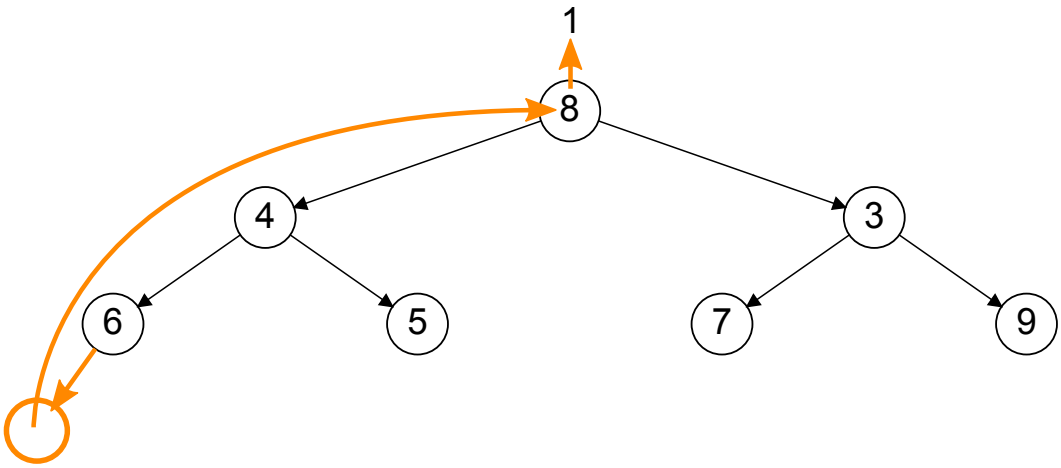
Erklärung

Die Laufzeit von `removeMin()` ist wiederum durch die Höhe des Baums begrenzt und beträgt $O(\log n)$.

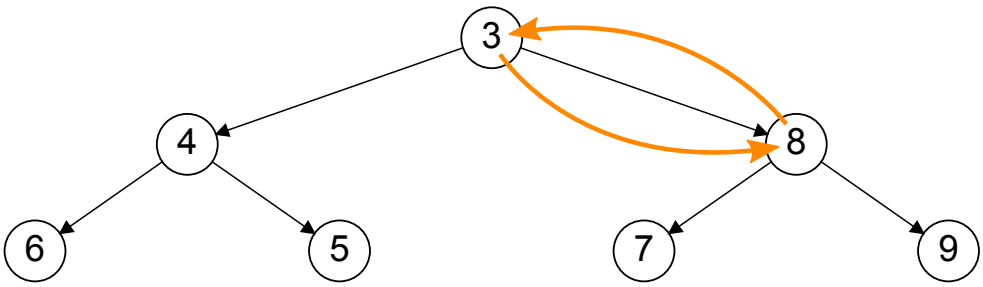
[Slide 27]



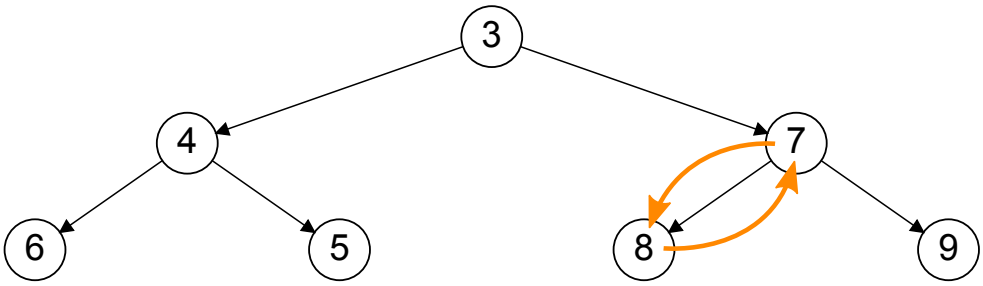

[Slide 28]




[Slide 29]



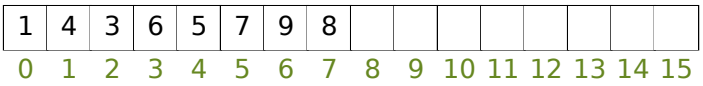
**[Slide 30]**

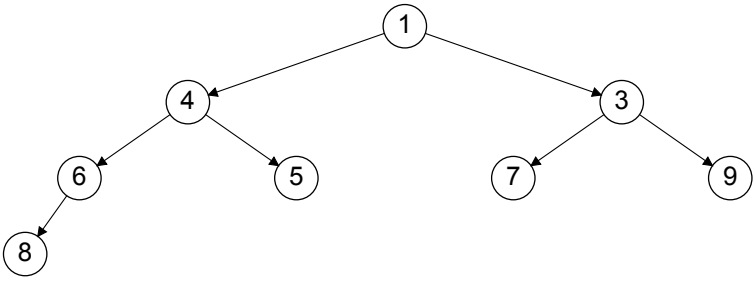



## Repräsentation mittels Array-basierter Liste [Slide 31]

Zeilenweise Nummerierung; dynamisches Array.

Laufzeit der Methoden?




Erklärung

Welche der möglichen Datenstrukturen für Bäume sollten wir für einen Heap wählen? Die Eigenschaften des Heap und seiner Algorithmen legen eine sehr eindeutige Antwort nahe.

Ein Heap ist ein vollständiger Binärbaum. Daher ist eine Array-basierte Datenstruktur besonders platzeffizient. Die Algorithmen des Heap erfordern lediglich das Einfügen am Ende des Baums, also auch des Arrays, sowie das Vertauschen von Elementen. Es werden keine Unterbäume umgehängt. Damit ist die ArrayList eine ideale Datenstruktur für Heaps.

HeapPriorityQueue [Slide 32]

```java
/**
 * An implementation of a priority queue using an array-based heap.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public HeapPriorityQueue() { super(); }

  /**
   * Creates an empty priority queue using the given comparator to order keys.
   * @param comp comparator defining the order of keys in the priority queue
   */
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }

  // protected utilities
  protected int parent(int j) { return (j-1) / 2; } // truncating division
  protected int left(int j) { return 2*j + 1; }
  protected int right(int j) { return 2*j + 2; }
  protected boolean hasLeft(int j) { return left(j) < heap.size(); }
  protected boolean hasRight(int j) { return right(j) < heap.size(); }

  /** Exchanges the entries at indices i and j of the array list. */
  protected void swap(int i, int j) {
    Entry<K,V> temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
  }

  /** Moves the entry at index j higher, if necessary, to restore the heap property. */
  protected void upheap(int j) {
    while (j > 0) {          // continue until reaching root (or break statement)
      int p = parent(j);
      if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
      swap(j, p);
      j = p;                              // continue from the parent's location
    }
  }

  /** Moves the entry at index j lower, if necessary, to restore the heap property. */
  protected void downheap(int j) {
    while (hasLeft(j)) {            // continue to bottom (or break statement)
      int leftIndex = left(j);
      int smallChildIndex = leftIndex; // although right may be smaller
      if (hasRight(j)) {
```

```
        int rightIndex = right(j);
        if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
          smallChildIndex = rightIndex; // right child is smaller
    }
    if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
      break;                          // heap property has been restored
    swap(j, smallChildIndex);
    j = smallChildIndex;              // continue at position of the child
  }
}

// public methods

/**
 * Returns the number of items in the priority queue.
 * @return number of items
 */
@Override
public int size() { return heap.size(); }

/**
 * Returns (but does not remove) an entry with minimal key.
 * @return entry having a minimal key (or null if empty)
 */
@Override
public Entry<K,V> min() {
  if (heap.isEmpty()) return null;
  return heap.get(0);
}

/**
 * Inserts a key-value pair and return the entry created.
 * @param key     the key of the new entry
 * @param value   the associated value of the new entry
 * @return the entry storing the new key-value pair
 * @throws IllegalArgumentException if the key is unacceptable for this queue
 */
@Override
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);    // auxiliary key-checking method (could throw exception)
  Entry<K,V> newest = new PQEntry<>(key, value);
  heap.add(newest);                   // add to the end of the list
  upheap(heap.size() - 1);            // upheap newly added entry
  return newest;
}

/**
 * Removes and returns an entry with minimal key.
 * @return the removed entry (or null if empty)
 */
@Override
public Entry<K,V> removeMin() {
```

```
    if (heap.isEmpty()) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size() - 1);          // put minimum item at the end
    heap.remove(heap.size() - 1);      // and remove it from the list ;
    downheap(0);                       // then fix new root
    return answer;
  }

  /** Used for debugging purposes only */
  private void sanityCheck() {
    for (int j=0; j < heap.size(); j++) {
      int left = left(j);
      int right = right(j);
      if (left < heap.size() && compare(heap.get(left), heap.get(j)) < 0)
        System.out.println("Invalid␣left␣child␣relationship");
      if (right < heap.size() && compare(heap.get(right), heap.get(j)) < 0)
        System.out.println("Invalid␣right␣child␣relationship");
    }
  }
}
```

# 5 *Bottom-Up Heap Construction*

Video 5 beginnt hier.

## *Bottom-Up Heap Construction* [Slide 33]

Aufbau eines Heaps durch iterativen Aufruf von `insert()` dauert $O(n \log n)$.

Sind alle Elemente auf einmal vorhanden, kann ein Heap in $O(n)$ aufgebaut werden!

Zur einfacheren Darstellung nehmen wir $n = 2^{h+1} - 1$ an, d.h. der Baum ist *voll* mit Höhe $h = \log(n+1) - 1$.

1. Erstelle $(n+1)/2$ Heaps, die aus jeweils einem einzigen Knoten bestehen.
2. Für $i = 2, \dots, h+1$:
   (a) Fusioniere Heaps (jeder enthält $2^{i-1} - 1$ Knoten) paarweise durch Hinzufügen einer gemeinsamen Wurzel.
   (b) Propagiere das neue Wurzelelement abwärts, bis die Heap-Ordnung wieder hergestellt ist.

   Das Ergebnis sind $(n+1)/2^i$ neue Heaps von jeweils $2^i - 1$ Knoten.

Erklärung

Wie lange dauert es, eine Heap-basierte Vorrangwarteschlange durch $n$ Aufrufe von `insert()` mit $n$ Elementen zu füllen? Wir haben $n$ Aufrufe, von denen jeder eine Laufzeit von $O(\log n)$ hat, also ist die Gesamtlaufzeit $O(n \log n)$.

In der Praxis hat man jedoch oft alle $n$ Elemente bereits in einem Array parat. Unseren Heap speichern wir ebenfalls in einem Array. Können wir das nicht irgendwie zu unserem Vorteil nutzen?

*Yes We Can!* Wir können nicht nur das bestehende Array nutzen und uns die eigene Datenstruktur für den Heap sparen. Wir können sogar die Erstellung des Heaps von $O(n \log n)$ auf $O(n)$ beschleunigen!

Dies ist in der Tat ein erstaunliches Ergebnis! Schauen wir uns uns an, wie das funktioniert.

Animation

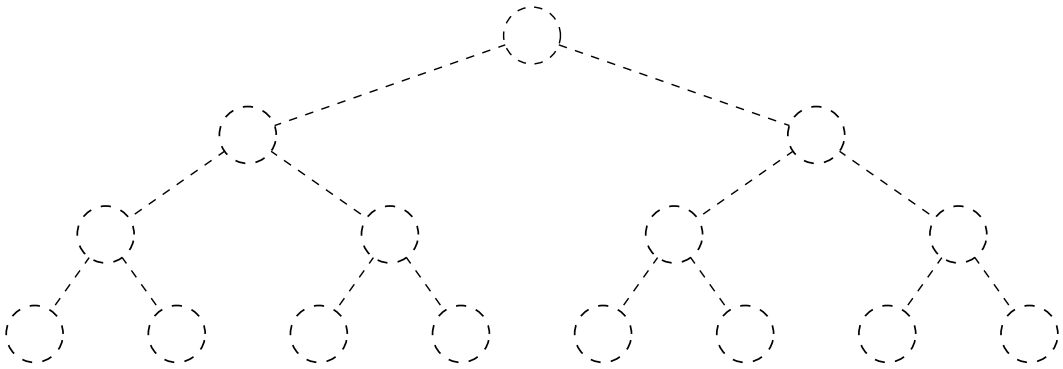
Um uns die Beschreibung des Konzepts zu vereinfachen, gehen wir davon aus, dass unser Array so viele Elemente enthält, dass der fertige Baum *voll* ist, d.h., seine letzte Zeile enthält die maximale Anzahl Knoten.

Wir beginnen damit, die letzte Zeile unseres Heaps aufzufüllen. Das sind genau $(n+1)/2$ Knoten.

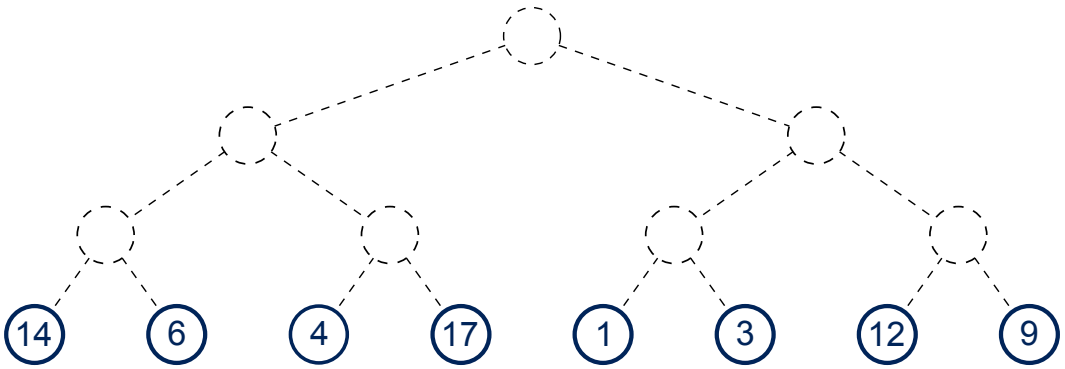
Nun füllen wir die vorletzte Zeile auf, indem wir jeweils zwei benachbarte Knoten der letzten Zeile mit einer gemeinsamen Wurzel zu einem Baum verbinden. Jeden dieser $(n+1)/4$ Bäume verwandeln wir in einen Heap, indem wir durch Absinken des jeweiligen Wurzelelements die Heap-Ordnung herstellen. Dies leistet unsere bereits besprochene Methode `downheap()` genau wie bei `insert()`: Die linken und rechten Unterbäume der neuen Wurzel sind Heaps; nur die neue Wurzel verletzt möglicherweise die Heap-Ordnung.

Diese Prozedur iterieren wir Zeile für Zeile nach oben, bis wir die Wurzel eingefügt und sein Element haben absinken lassen, bis die Heap-Ordnung hergestellt ist.

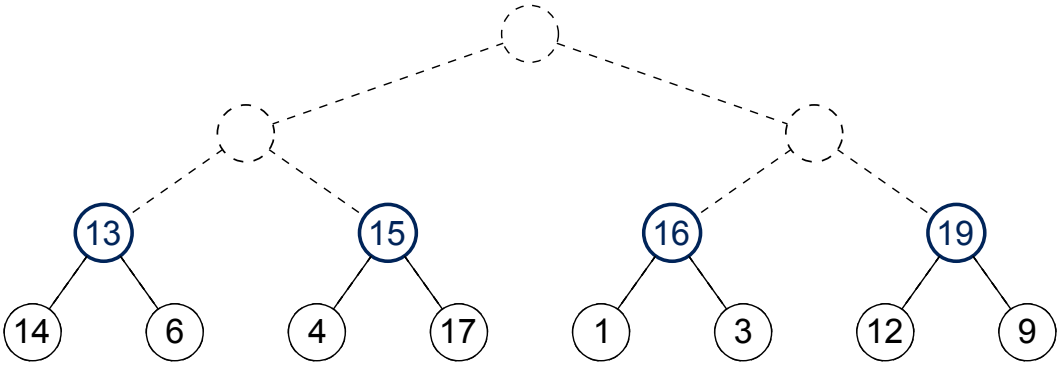
**[Slide 34]**



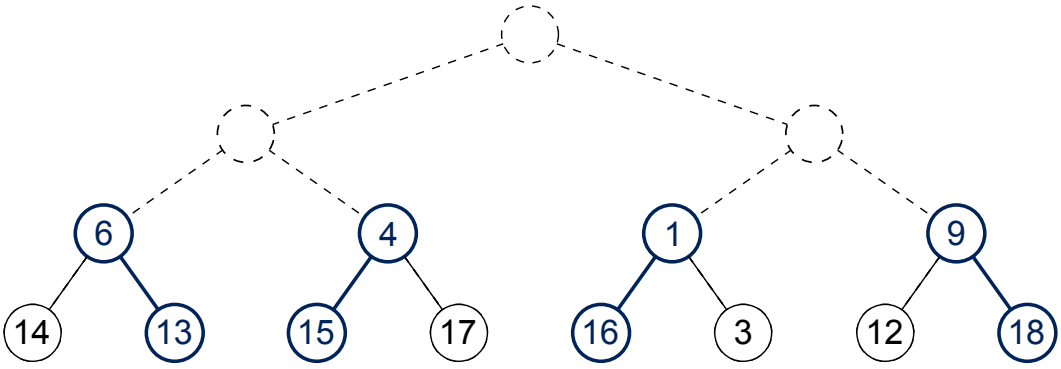
**[Slide 35]**



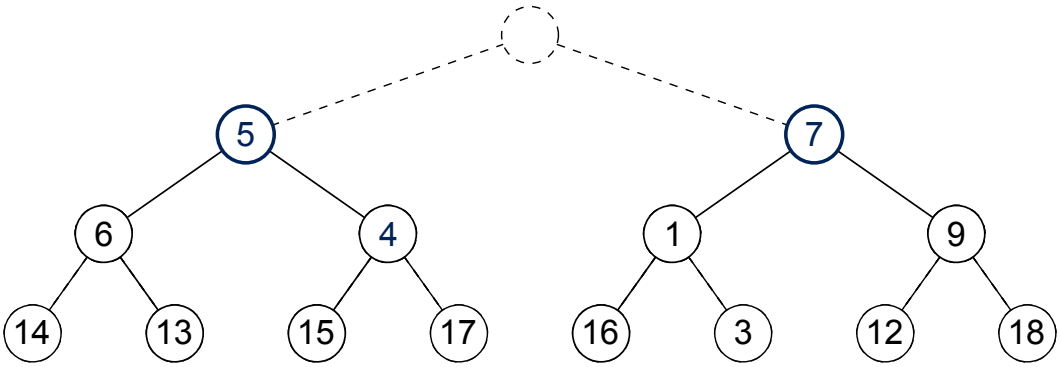
**[Slide 36]**



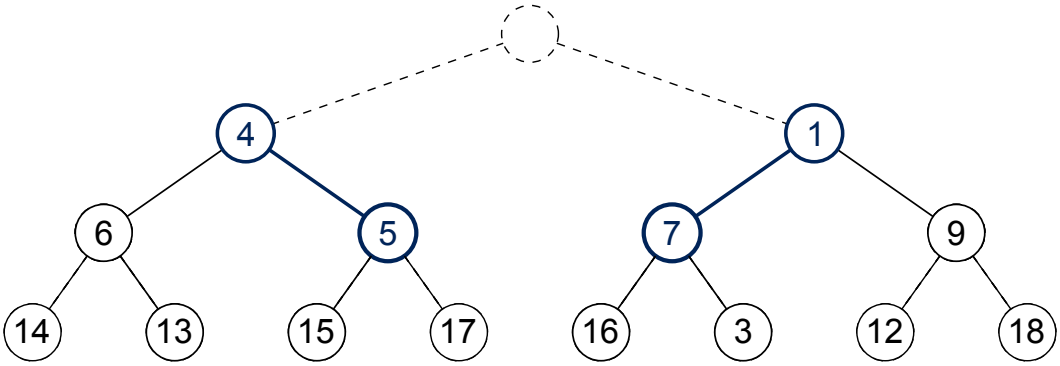

**[Slide 37]**



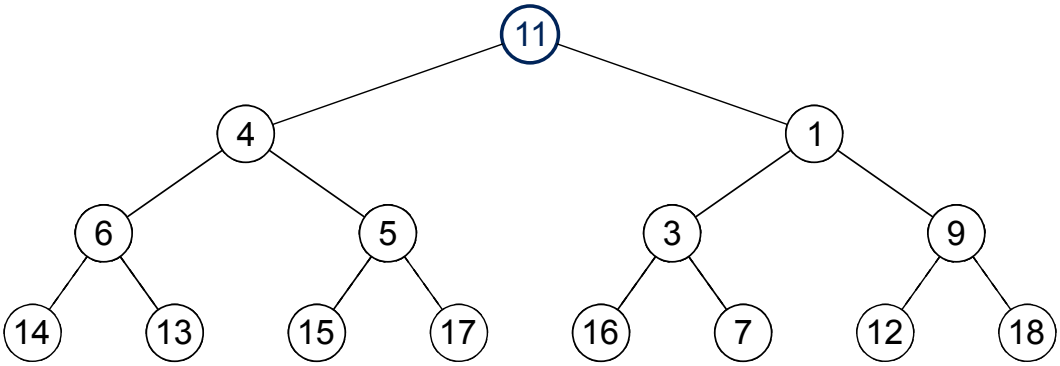

**[Slide 38]**



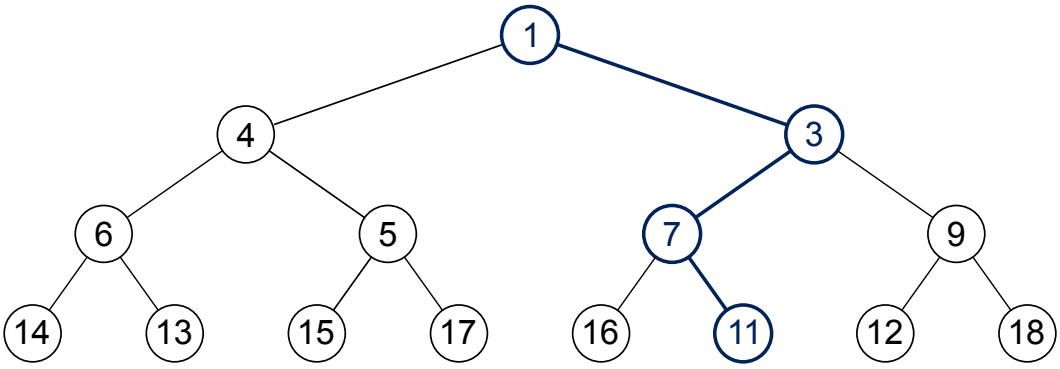
**[Slide 39]**




**[Slide 40]**




**[Slide 41]**



### Algorithmus [Slide 42]

Alle Elemente sind unsortiert im Array gegeben.

1. Für alle internen Knoten $e$ von unten nach oben
   (a) $e$ absinken lassen

```
Algorithm heapify(A, n):
  Require: An array A storing n keys.
  Ensure:  A contains the same keys arranged as a heap.
  for j ← ⌊(n−1)/2⌋ downto 0
    downheap(j)
```

***Wichtig***

`heapify()` zeigt den Algorithmus der *bottom-up heap construction*! Die obigen Grafiken illustrieren lediglich die Baum-Operationen, die `heapify()` ausführt.

Erklärung

Wir haben soeben das Funktionsprinzip der *bottom-up heap construction* grafisch illustriert. Nun formulieren wir es als Algorithmus `heapify()` auf unserer Array-Datenstruktur.

Erinnern wir uns, dass in der Array-Datenstruktur per zeilenweiser Nummerierung die Blätter unseres zukünftigen Heaps nebeneinander hinten im Array liegen. Um das letzte Blatt mit seinem Elternknoten und ggf. seinem linken Geschwisterknoten zu einem Heap zu formen, müssen wir lediglich diesen Elternknoten am Index `j` finden und lokal die Heap-Ordnung herstellen, indem wir `downheap(j)` aufrufen.

Genauso verfahren wir mit dem linken Nachbar-Heap des letzten Knotens, d.h., wir rufen `downheap(j-1)` auf. Am Anfang der vorletzten Zeile angekommen, besteht diese aus lauter Wurzeln kleiner Heaps.

Um diese wiederum paarweise zu Heaps zusammenzufassen, verfahren wir exakt genauso: Wir rufen `downheap()` auf dem Elternknoten des letzten Knotens der vorletzten Zeile auf. In der zeilenweisen Nummerierung finden wir diesen als Vorgänger des ersten Knotens der folgenden Zeile.

Wir können also die Zeilen komplett ignorieren und sukzessive `downheap()` auf allen internen Knoten unseres Heaps aufrufen, vom Elternknoten des letzten Blatts bis zur Wurzel.

All dies leistet unser kleiner `heapify()`-Algorithmus auf Basis der `downheap()`-Prozedur!

Dass dieser Algorithmus korrekt ist, ist leicht einzusehen. Aber läuft er tatsächlich in $O(n)$, wie versprochen? Schließlich rufen wir $O(n)$-mal `downheap()` auf, und jeder dieser Aufrufe beschreitet Pfade in unserem Baum!

## An Array Constructor for our `HeapPriorityQueue` [Slide 43]

```
/* Creates a priority queue initialized with the given key-value pairs. */
public HeapPriorityQueue(K[] keys, V[] values) {
  super();
  for (int j=0; j < Math.min(keys.length, values.length); j++)
    heap.add(new PQEntry<>(keys[j], values[j]));
  heapify();
}

/* Performs a bottom-up construction of the heap in linear time. */
protected void heapify() {
  int startIndex = parent(size()-1); // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--) // loop until processing the root
    downheap(j);
}
```

## Laufzeitanalyse [Slide 44]

**Proposition:** *Bottom-up construction* eines Heaps mit $n$ Elementen hat eine Laufzeit von $O(n)$ unter der Annahme, dass zwei Schlüssel in konstanter Zeit verglichen werden können.

**Beweisskizze:** Die Zeit wird durch die Abwärtspropagation dominiert. Wie verhält sich die Anzahl der Propagationsschritte zur Anzahl der Kanten?

- Sei $\pi_v$ der Pfad von Knoten $v$ zu seinem Inorder-Nachfolger.
- Da die Propagation von $v$ maximal $\|\pi_v\|$ Schritte erfordert, ist die Gesamtzeit aller Propagationen $O\left(\sum_v \|\pi_v\|\right)$.
- Jeder Pfad $\pi_v$ besteht aus genau einer rechten Kante gefolgt von einer Sequenz linker Kanten.
- Jede rechte Kante $(v, w)$ ist die erste Kante von $\pi_v$.
- Da keine zwei Knoten denselben Inorder-Nachfolger haben können, ist jede linke Kante Teil genau eines solchen Pfades.

Erklärung

Skizzieren wir hier einen Beweis für die Proposition, dass unsere *Bottom-Up Heap Construction* einen Heap von $n$ Elementen in einer asymptotischen Laufzeit von $O(n)$ aufbaut.

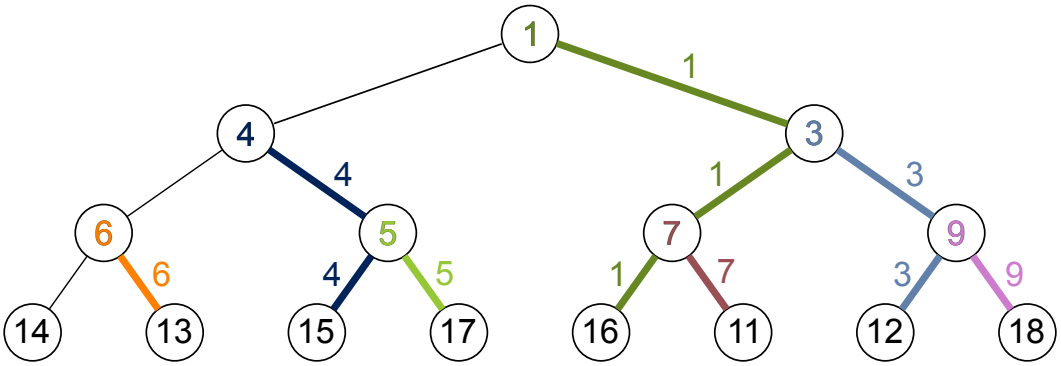
Alle einzelnen Schritte des Algorithmus lassen sich in konstanter Laufzeit implementieren. Da unsere Elemente bereits in einem Array vorliegen, ergibt sich die Gesamtlaufzeit aus der Gesamtzahl der Vertauschungen beim Absinken der Elemente der zwischenzeitlichen Wurzeln.

Alle internen Elemente des finalen Heaps liegen zwischendurch in Wurzeln und können absinken. Wir haben bereits gezeigt, dass die Hälfte der Knoten eines vollen Binärbaums intern ist. Das sind also $O(n)$ interne Knoten. Das Absinken eines Elements kann einen Pfad der Länge $O(\log n)$ ablaufen. Damit sieht es so aus, als könnten wir keine bessere Laufzeit garantieren als $O(n \log n)$!

Wir haben hier nach der maximalen Pfadlänge beim Absinken eines *beliebigen* Elements gefragt. Die entscheidende Frage ist jedoch: Was ist die maximale *Gesamtlänge aller Pfade*, die beim Absinken *aller* Elemente auftreten kann?

## Pfade zu Inorder-Nachfolgern [Slide 45]

Jede Kante ist mit $v$ markiert, zu deren $\pi_v$ sie gehört:




Beispiel

Um uns hierzu einen Überblick zu verschaffen, betrachten wir ein Beispiel und verfolgen für jedes einzelne interne Element, wie weit es maximal absinken kann. O.B.d.A. nehmen wir an, dass jedes Element bis zu seinem Inorder-Nachfolger absinkt. (Tiefer kann es nicht sinken!) Mit anderen Worten, es wird zuerst mit dem Element seines rechten Kinds vertauscht, und danach immer mit dem linken Kind, bis es in der letzten Zeile ankommt.

Damit wir die Wege der einzelnen Elemente verfolgen können, bitten wir sie, die Kanten, die sie dabei traversieren, mit ihrem Schlüssel zu markieren. Das Ergebnis sehen wir hier: Jede Kante wird von genau einem Element traversiert, bis auf den Pfad von der Wurzel bis zum ersten Knoten der letzten Zeile.

Wir können uns leicht davon überzeugen, dass tatsächlich keine Kante auf mehreren solcher Absinkpfaden liegen kann: Die rechte Kante wird durch diese Absinkstrategie ausschließlich durch das Element ihres Ausgangsknotens traversiert. Aber auch alle folgenden linken Kanten können nur von demselben Element traversiert werden, da keine zwei verschiedenen Knoten denselben Inorder-Nachfolger haben können!

Damit ist die Gesamtzahl aller Vertauschungen begrenzt durch die Anzahl der Kanten unseres Heaps. Diese ist jedoch, wie wir leicht zeigen können, um 1 geringer als die Anzahl $n$ der Knoten. Damit ist die Gesamtzahl der Vertauschungen und folglich die asymptotische Laufzeit der *Bottom-Up Heap Construction* $O(n)$!

# 6 Sortieren mit einer Vorrangwarteschlange

Video 6 beginnt hier.

## Sortieren mit einer Vorrangwarteschlange [Slide 46]

```
Algorithm PriorityQueueSort(Q, P):
  Require: A queue Q storing elements with a total order,
             and a priority queue P that uses the same total order relation.
  Ensure: The queue Q sorted by the total order relation.
  while not Q.isEmpty() do
    k ← Q.dequeue()
    P.insert(k, 0)     // just key (= value), no extra data
  while not P.isEmpty() do
    (k, v) ← P.removeMin()
    Q.enqueue(k)
```

Erklärung

Machen wir nun einen kleinen Ausflug zum Sortieren. Da sukzessive Aufrufe von `removeMin()` die Elemente in aufsteigender Reihenfolge ihrer Schlüssel zurückgeben, bietet sich dieser zweistufige Sortieralgorithmus an: Zuerst werden alle Elemente nacheinander mittels `insert()` in die Vorrangwarteschlange eingefügt. Anschließend werden sie nacheinander mittels `removeMin()` wieder herausgeholt.

Es stellt sich heraus, dass dieser allgemeine Sortieralgorithmus auf Basis einer Vorrangwarteschlange exakt drei typischen Sortieralgorithmen entpricht, je nachdem, welche Datenstruktur der Vorrangwarteschlange zugrunde liegt.

## Selection Sort [Slide 47]

Implementation der Sortierung mit einer unsortierten Liste.

Laufzeit-Komplexität?

Animation

Implementieren wir die Vorrangwarteschlange mit einer *unsortierten* ArrayList, dann besteht `removeMin()` darin, dieses unsortierte Array der Länge nach zu durchlaufen und dabei das kleinste Element zu suchen. In dieser Animation durchläuft die dunkelblaue Kreismarkierung das unsortierte Array, und das jeweils kleinste bisher gefundene Element wird rot markiert.

Am Ende des Arrays angekommen, wird das kleinste Element entnommen und an das sortierte Ausgabearray angefügt. Da sich die Gesamtanzahl der Elemente nicht ändert, bringen wir sowohl das sortierte als auch das unsortierte Array in derselben ArrayList unter. Der sortierte Teil des Arrays wächst von links nach rechts, während sich der unsortierte Teil entsprechend nach rechts zurückzieht.

Haben wir also das kleinste Element gefunden, stellen wir es an den Anfang des Arrays. Um dafür Platz zu schaffen, setzen wir das dort befindliche Element einfach an den freiwerdenden Platz, wo sich vorher das kleinste Element befand. Der sortierte Teil des Arrays ist in der Animation grün hinterlegt.

Anschließend folgt das nächste `removeMin()`, per sequenzieller Suche über dem unsortierten Teil des Arrays.

Da die Hauptarbeit dieses Algorithmus darin besteht, das kleinste Element zu extrahieren, wird er als *Selection Sort* bezeichnet.

Erklärung

Was ist die Laufzeit von *Selection Sort*? In der ersten Phase wird $n$-mal `insert()` aufgerufen. Da jedes Element lediglich an das Ende des Arbeitsarrays angefügt werden muss, läuft jeder Aufruf in konstanter Zeit. Die erste Phase hat also eine asymptotische Laufzeit von $O(n)$. Bekommen wir die Elemente bereits in einer ArrayList übergeben, dann können wir uns diese Phase ggf. komplett sparen.

In der zweiten Phase wird $n$-mal `removeMin()` aufgerufen. Jeder Aufruf sucht den unsortierten Teil des Arrays komplett ab, was jedesmal eine Zeit von $\Theta(n)$ beansprucht. Die Gesamtdauer der zweiten Phase hat also eine asymptotische Laufzeit von $\Theta(n^2)$. Da dies die dominierende Laufzeit der beiden Phasen ist, ist dies auch die Laufzeit des kompletten *Selection Sort*-Algorithmnus.

## Insertion Sort [Slide 48]

Implementation der Sortierung mit einer sortierten Liste.

Laufzeit-Komplexität?

Animation

Implementieren wir die Vorrangwarteschlange mit einer *sortierten* ArrayList, dann besteht `insert()` darin, für jedes Element den bereits sortierten Teil des Arrays der Länge nach zu durchlaufen und dabei das kleinste Element an der korrekten Stelle einzusortieren. Auf diese Weise wächst der sortierte Teil des Arrays von links nach rechts, wobei sich der unsortierte Teil entsprechend zurückzieht, genau wie bei *Selection Sort*.

Bei jedem `insert()` wird das Zielarray, der linke Teil der ArrayList, um das nächste Element erweitert. Anschließend wird diese neue, letzte Element an seinen Platz in der sortierten Sequenz bugsiert, indem es sukzessive mit seinem linken Nachbarn vertauscht wird, bis sein linker Nachbar nicht mehr größer ist als er selbst. In der Animation werden diese paarweisen Vergleiche durch die beiden dunkelblauen Kreismarkierungen dargestellt.

Da die Hauptarbeit dieses Algorithmus darin besteht, das nächste Element an seiner korrekten Stelle einzufügen, wird er als *Insertion Sort* bezeichnet.

Erklärung

Was ist die Laufzeit von *Insertion Sort*? In der ersten Phase wird $n$-mal `insert()` aufgerufen. Bei jedem Aufruf muss dabei ein Element an seinen Platz gebracht werden. Hierzu sind $O(n)$ Vergleiche und Vertauschungen notwendig. Daher hat die erste Phase eine Gesamtlaufzeit von $O(n^2)$.

In der zweiten Phase wird $n$-mal `removeMin()` aufgerufen. Jeder Aufruf muss lediglich das nächste Element des bereits sortierten Arrays zurückliefern, was sich in konstanter Zeit machen lässt. Die zweite Phase hat also eine Laufzeit von $O(n)$, und die Gesamtlaufzeit von *Insertion Sort* ist die Laufzeit der ersten Phase, also $O(n^2)$.

Hier können wir uns ggf. die zweite Phase komplett sparen, was jedoch genau wie bei *Selection Sort* keinen Einfluss auf die asymptotische Gesamtlaufzeit hat.

## Heap Sort [Slide 49]

Implementation der Sortierung mit einem Heap.

Laufzeit-Komplexität?

Erklärung

Implementieren wir die Vorrangwarteschlange mit einem Heap, dann hat jeder Aufruf von `insert()` und jeder Aufruf von `removeMin()` eine Laufzeit von $O(\log n)$. Da jede dieser beiden Methoden jeweils $n$-mal aufgerufen wird, ist die Gesamtlaufzeit dieses sogenannten *Heap-Sort*-Algorithmus $O(n \log n)$. *Heap-Sort* ist also effizienter als *Selection Sort* und *Insertion Sort*.

## In-Place Heap Sort [Slide 50]

Man verwendet einen *inversen Komparator* (so dass `min()` den *maximalen* Schlüssel liefert).

1. **Erstellung des Heaps** Im Array wächst der *Heap* von links nach rechts, während die unsortierte Sequenz entsprechend schrumpft (oder: *Bottom-Up Heap Construction*).

2. **Extraktion der sortierten Sequenz** Im Array wächst die *sortierte Sequenz* von rechts nach links, während der Heap entsprechend schrumpft.

Erklärung

Implementieren wir, wie üblich, unseren Heap auf Basis einer ArrayList, dann können wir *Heap-Sort in place* implementieren, das heißt, das übergebene Array direkt sortieren, ohne weitere Datenstrukturen, deren Größe von $n$ abhängt, genau wie bei *Selection Sort* und *Insertion Sort*.

Bei *Selection Sort* haben wir gesehen, dass die *erste* Phase des generischen Sortierens mit einer Vorrangwarteschlange wegfallen kann, nämlich das Einfügen der Elemente in die Vorrangwarteschlange. Bei *Insertion Sort* kann die *zweite* Phase wegfallen, nämlich das Extrahieren der Elemente aus der Vorrangwarteschlange. *Heap-Sort* benötigt dagegen beide Phasen.

In der ersten Phase wird der Heap erstellt. Dies kann in einer Zeit von $O(n \log n)$ durch $n$ sukzessive Aufrufe von `insert()` geschehen, oder in einer Zeit von $O(n)$ durch *Bottom-Up Heap Construction*.

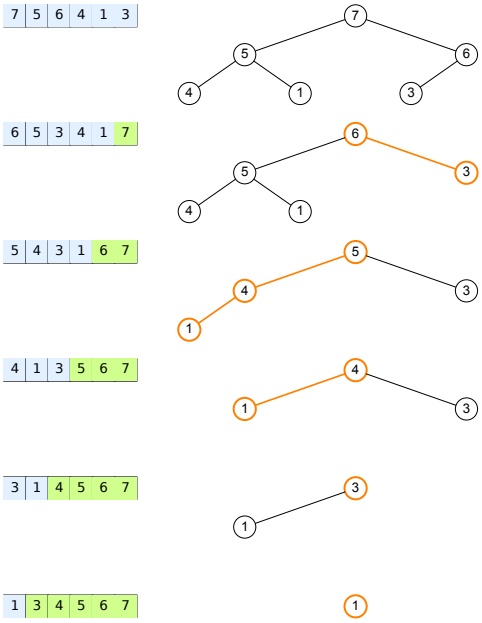
Anschließend wird die sortierte Ausgabesequenz erzeugt, indem $n$-mal `removeMin()` aufgerufen wird.

Wie bei *Selection Sort* und *Insertion Sort* besteht das Array aus einem unsortierten und einem sortierten Teil. Bei *Heap-Sort* ist der unsortierte Teil allerdings nicht willkürlich angeordnet, sondern hat nach der ersten Phase die Form eines Heaps in zeilenweiser Nummerierung.

In einer typischen Formulierung von *Heap-Sort* verwendet der Heap einen *inversen* Komparator und liefert die Elemente in umgekehrter Reihenfolge zurück. Ein solcher Heap wird auch als *Max*-Heap bezeichnet, im Gegensatz zu unserem bisher verwendeten *Min*-Heap.

Da `removeMin()` hier das *größte* Element zurückliefert, muss der sortierte Teil des Arrays in diesem Fall von rechts nach links wachsen, während sich der unsortierte Teil nach links zurückzieht.

## [Slide 51]




Beispiel

In diesem Beispiel sortieren wir die Sequenz 6,5,7,4,1,3. Das oberste Bild zeigt das Array nach der ersten Phase von *Heap-Sort*, der *Bottom-Up Heap Construction*. Links sehen wir die Array-Repräsentation des Heaps blau hinterlegt, und rechts die äquivalente graphische Darstellung als Baum. Wie Sie sehen, entspricht die Anordnung der Elemente im Array der zeilenweisen Nummerierung der Elemente im Baum.

Nun beginnt die zweite Phase mit dem ersten Aufruf von `removeMin()`. Das Wurzelelement 7 wird entnommen, und das letzte Element des Heaps, die 3, nimmt dessen Platz in der Wurzel ein. Die dadurch frei gewordene Zelle ganz rechts im Array wird zum letzten Element der sortierten Sequenz, hier grün hinterlegt. Hier wird also die entnommene 7 platziert.

Anschließend wird durch Absinken der 3 von der Wurzel die Heap-Ordnung wieder hergestellt. Hier vertauschen wir immer mit dem *größeren* Kindelement, da es sich um einen Max-Heap handelt. Der Pfad der absinkenden 3 ist hier rot hervorgehoben.

Es folgt der nächste Aufruf von `removeMin()`, und so weiter. Auf diese Weise wächst die sortierte Sequenz von rechts nach links, während der Heap sich nach links zurückzieht. Am Ende besteht der Heap nur noch aus seiner Wurzel. Da sein Element ein Kleinstes sein muss, ist damit die Sequenz komplett sortiert.

# 7 Zusammenfassung

## Zusammenfassung [Slide 52]

**ADT:**

- Vorrangwarteschlange

**DS:**

- Liste (verkettet oder ArrayList; unsortiert oder sortiert)
- Heap

**Algorithmen:**

- aufsteigen; absinken
- *Bottom-Up Heap Construction*

**Anwendung:**

- Sortierung