

## Aussagen:

- Anmeldung zu Tutorien bis Mittwoch über Moses
- Bis Mittwoch: 3er-Gruppen für HA/PA bilden.
- Modulklausur: 22. Juli, 4. Okt.
- Schein-Kriterien:
  - 50% Punkte HA
  - 3 PA, davon jeder einmal vorstellen
  - Projekt

## • Rückgespräch

• 1. HA / PA: diese Woche

HA-Abgabe: Di, 14:15 Uhr

PA-Abgabe: Do / Fr

---

Inhalt: Algorithmen, Datenstrukturen,  
Grundlagen aus Theor. Informatik

---

## Sortieren

bislang: Bubble sort, Selection Sort,  
Insertion Sort: Aufwand  $\Theta(n^2)$   
Merge Sort: Aufwand  $\Theta(n \log n)$   
Quick Sort: Aufwand  
- worst case  $\Theta(n^2)$   
- average case  $\Theta(n \log n)$

---

Heap Sort  $\Theta(n \log n)$  Vergleiche im  
worst case  
empirisch vergleichbar mit Merge Sort.

Verwende wenn dynamische Datenstruk-  
tur: Heap

Ein Heap ist eine spezielle Variante einer Priority Queue

Wertebereich: homogenen Kompon. Typ  
(wie bei array)

alle Komponenten haben u.a.  
einen Schlüssel.

Eine Priority Queue unterstützt die folgenden Operationen:

1) Einfügen einer Komponente  
insert ()

2) Zugriff auf Komponente mit  
größtem Schlüssel  
maximum ()

3) Entfernen Komponente mit dem  
größten Schlüssel  
extractMax ()

[ 4) Ändern eines Schlüssels  
changeKey () ]

All diese Operationen sollen möglichst  
effizient (schnell) durchführbar sein.

Im Falle eines Heaps:

insert():  $O(\log n)$

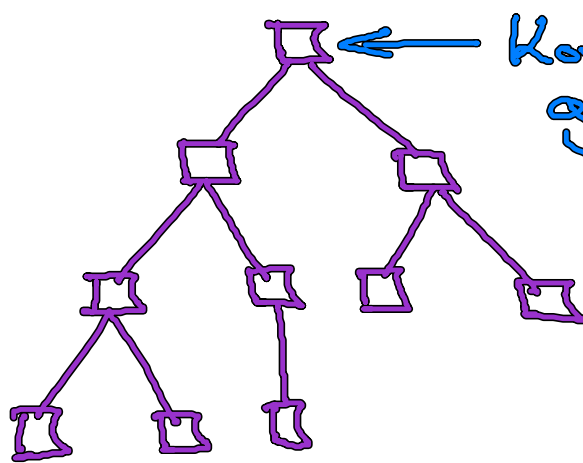
maximum():  $O(1)$

extractMax():  $O(\log n)$

changeKey():  $O(\log n)$

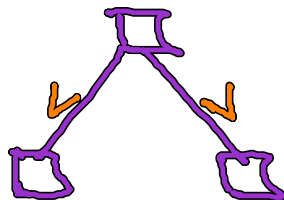
grobe Vorstellung eines Heaps:

voller binärer Baum



Komponente mit  
größtem Schlüssel  
sitzt in Wurzel.

Bedingung: Für jeden Knoten gilt:  
Die Schlüssel seiner Kinder sind  
kleiner als der eigene Schlüssel

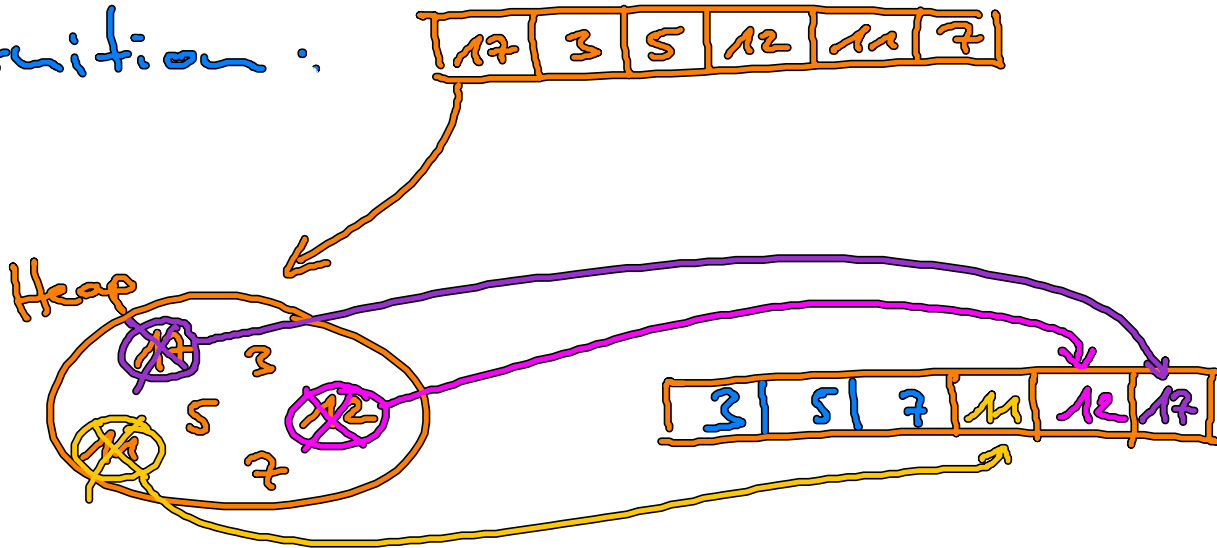


# Großstruktur von Heapsort:

## Grobstruktur von Heapsort

1. Gegeben sei das Array  $vec[]$  mit  $n$  Komponenten.
2. Initialisiere den Heap mit den Komponenten von  $vec$ .
3. **for**  $i := n - 1$  **downto** 0 **do**
  - 3.1 Greife auf das größte Element des Heaps zu.
  - 3.2 Weise diesen Wert der Arraykomponente  $vec[i]$  zu.
  - 3.3 Entferne das größte Element aus dem Heap.

Intuition:



Schleifeninvariante: Nach  $j$  Durchläufen der Schleife stehen die  $j$  größten Elemente sortiert in den letzten  $j$  Einträgen des Arrays.

Aufwand:

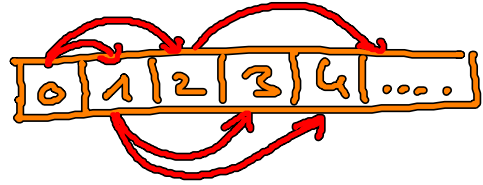
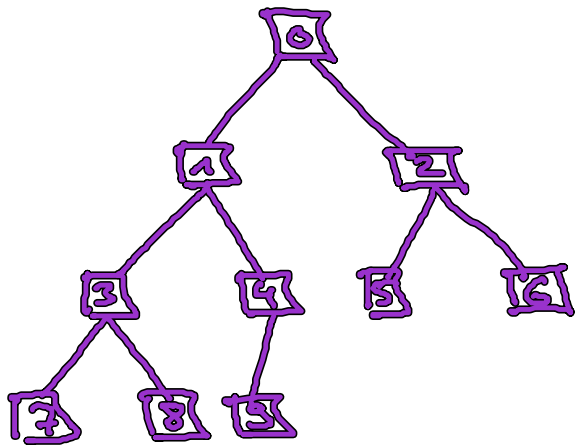
$$\text{Aufbau des Heaps: } \sum_{i=1}^n O(\log i) \\ \in O(n \log n)$$

$n$ -maliges Extrahieren des Maximums:

$$\sum_{i=0}^{n-1} O(\log(n-i)) \in O(n \log n).$$

## Implementierung des Heaps

in Array: Interpretiere Array als vollen binären Baum:



Die Kinder von Knoten  $i$  haben die Nummern  $2i+1, 2i+2$

Def: Ein Array erfüllt die Heapeigenschaft genau dann wenn jeder Knoten in dem zugeh. vollen binären Baum einen größeren Schlüssel besitzt als seine Kinder.

$$\forall i : \text{vec}[i].\text{key} > \text{vec}[2i+1].\text{key} \\ \text{und} \quad \quad \quad > \text{vec}[2i+2].\text{key}$$

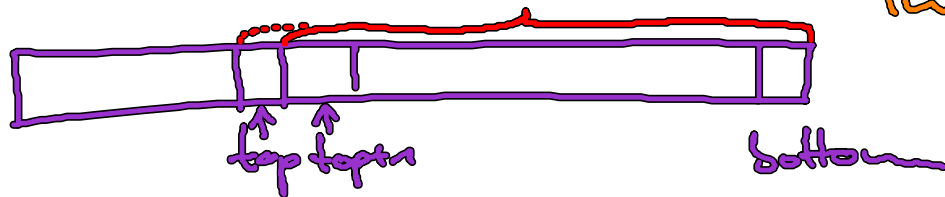
(sofern  $2i+1, 2i+2 < n$  (Länge des Arrays)).

## Folgerung:

- größtes Element steht in  $vec[0]$
- entlang eines Weges von der Wurzel zu einem Blatt fallen die Schlüssel streng monoton.

## Methode zur Herstellung der Heapeigensch.

void heapify (Item[] vec, int top, int bottom)



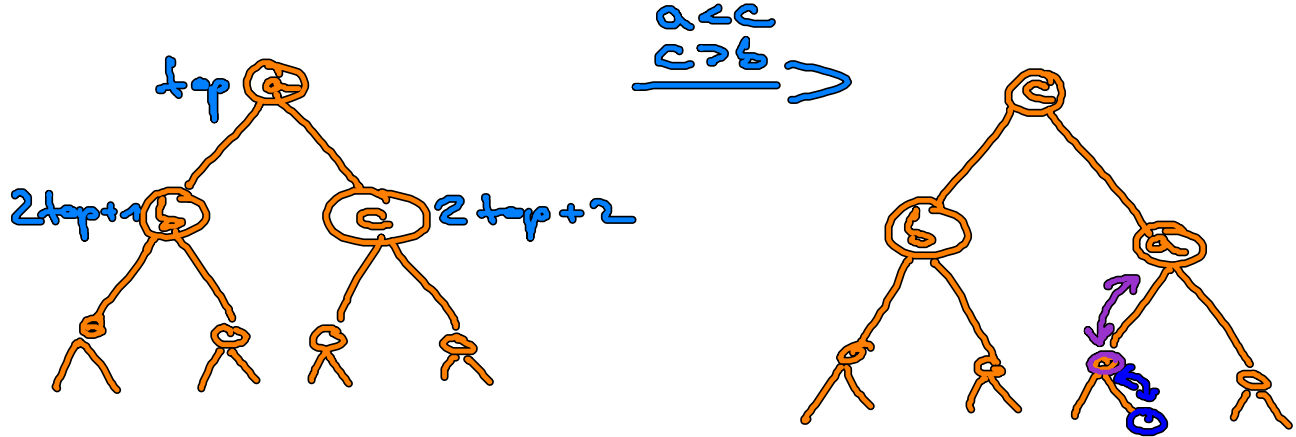
- setze voraus, dass Heapeigenschaft erfüllt ist in Teilarray von  $top+1$  bis  $bottom$
- stelle dann sicher, dass Heapeigenschaft erfüllt ist von  $top$  bis  $bottom$

1) Ermittle das größere der beiden Kinder "child" (falls Kinder existieren und im vorgegeb. Bereich liegen)

2) vergleiche  $vec[top].key$  mit  $vec[child].key$   
Falls  $vec[top].key > vec[child].key$ , fertig!  
Sonst: vertausche Einträge von  $top$  und  $child$ .

3) Wende heapify rekursiv an auf den Bereich  $child, \dots, bottom$ .





Korrektheit der Methode folgt aus Austausch + rekursiven Aufruf.

$\Rightarrow$  Heapeigenschaft kann also durch wiederholtes Aufrufen von `heapify` her gestellt werden.

```
/**
 * turns array into a heap
 * @param vec the array to which this happens
 */
private static void createHeap(Item[] vec) {
    for (int i = vec.length/2 - 1; i >= 0; i--) {
        heapify(vec, i, vec.length - 1);
    }
}
```

```
/**
 * establishes heap property in a certain range
 * @param vec the array in which this happens
 * @param top start of the range
 * @param bottom end of the range
 */
private static void heapify(Item[] vec, int top, int bottom) {
    Item temp;
    int child;

    if (2*top+1 > bottom) return; // nothing to do

    if (2*top+2 > bottom) {
        // vec[2*top+1] is only child of vec[top]
        child = 2*top+1;
    } else {
        // 2 sons, determine bigger one
        if (vec[2*top+1].key > vec[2*top+2].key) {
            child = 2*top+1;
        } else {
            child = 2*top+2;
        }
    }
}
```



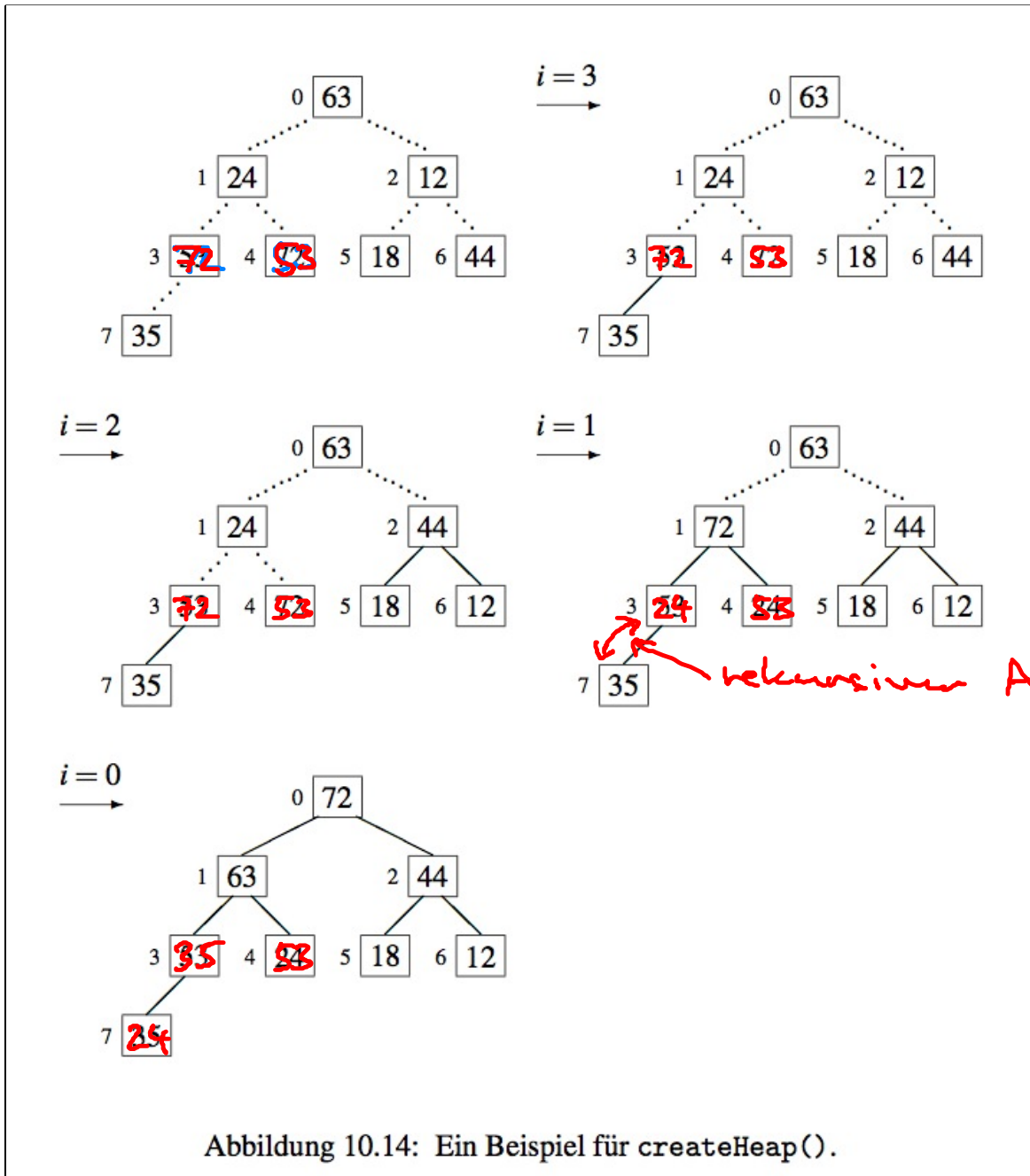
```

} //endif

// check if exchange is necessary
if (vec[top].key < vec[child].key) {
    temp = vec[top];
    vec[top] = vec[child];
    vec[child] = temp;
    // recursive call for possible further exchanges
    heapify(vec, child, bottom);
} //endif
}

```

## Beispiel:



Theorem: Das Herstellen der Heapeigenschaft funktioniert mit Laufzeit  $O(n)$ .