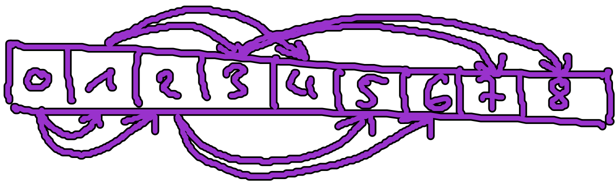
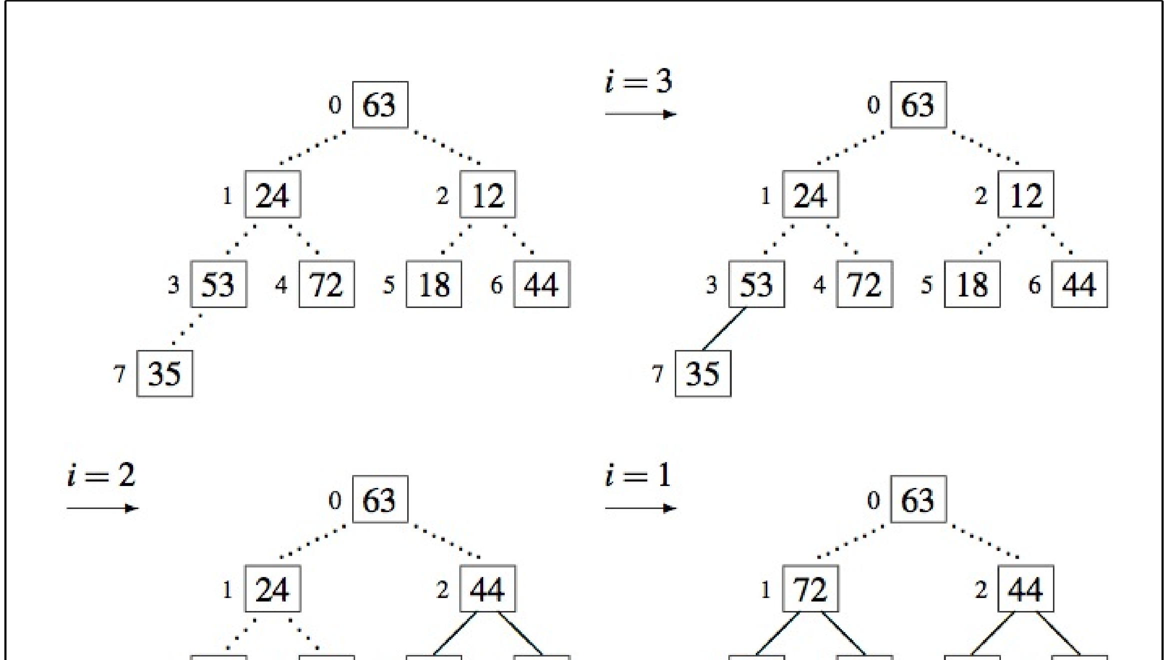

0 1 2 3 4 5 6 7 8


0 63
1 24
2 12
3 53
4 72
5 18
6 44
7 35
$i = 3$
0 63
1 24
2 12
3 53
4 72
5 18
6 44
7 35
$i = 2$
0 63
1 24
2 44
$i = 1$
0 63
1 72
2 44

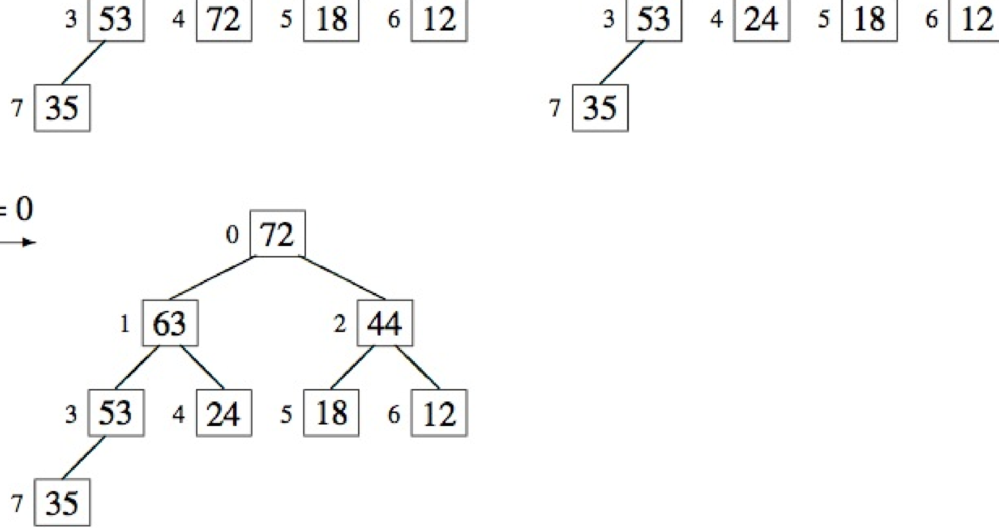
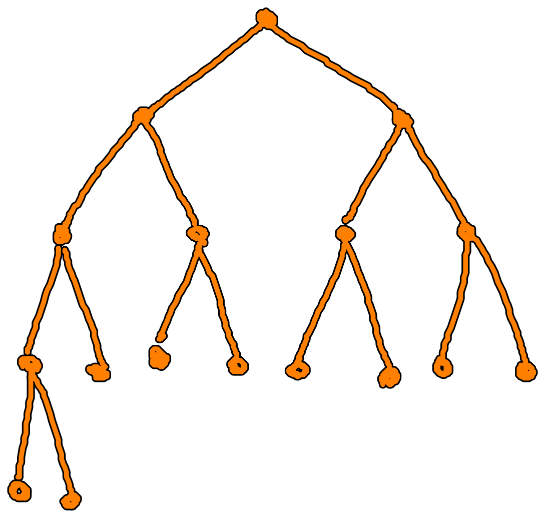


Abbildung 10.14: Ein Beispiel für createHeap().

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

Beweis: Betrachte Knoten des Baumes und schätze ab, wie tief das entsprechende Element absinken kann



Schicht	#Knoten	#Schichten, um die man absinken kann
0	1	h
1	2	$h-1$
2	4	$h-2$
3	8	$h-3$
4	16	$h-4$
$\vdots$		
i	2^i	$h-i$
$\vdots$		
$h-1$	2^{h-1}	1
h	$\leq 2^h$	0

Aufwand des Alg. entspricht Anzahl der insgesamt möglichen Absinkschritte

$$\sum_{i=0}^{h} 2^i \cdot (h-i) = \sum_{i=1}^{h} 2^{h-i} \cdot i = 2^h \cdot \sum_{i=1}^{h} \frac{i}{2^i}$$

$$\leq 2^h \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2 \cdot 2^h = 2^{h+1}$$

Für einen vollen binären Baum gilt:

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

$\Longrightarrow$ Aufwand $\in \mathcal{O}(n)$ □

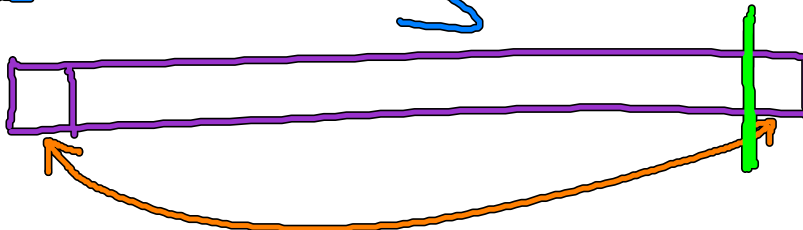
Zurück zu Heapsort:

Grobstruktur von Heapsort

1. Gegeben sei das Array `vec[]` mit n Komponenten.
2. Initialisiere den Heap mit den Komponenten von `vec`. ← $\mathcal{O}(n)$
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.

} Aufwand?

Genaue Beschreibung von 3.2, 3.3:



Idee: Tausche im ersten Durchlauf das größte Element an die letzte Stelle des Arrays, im zweiten Schritt an die vorletzte....

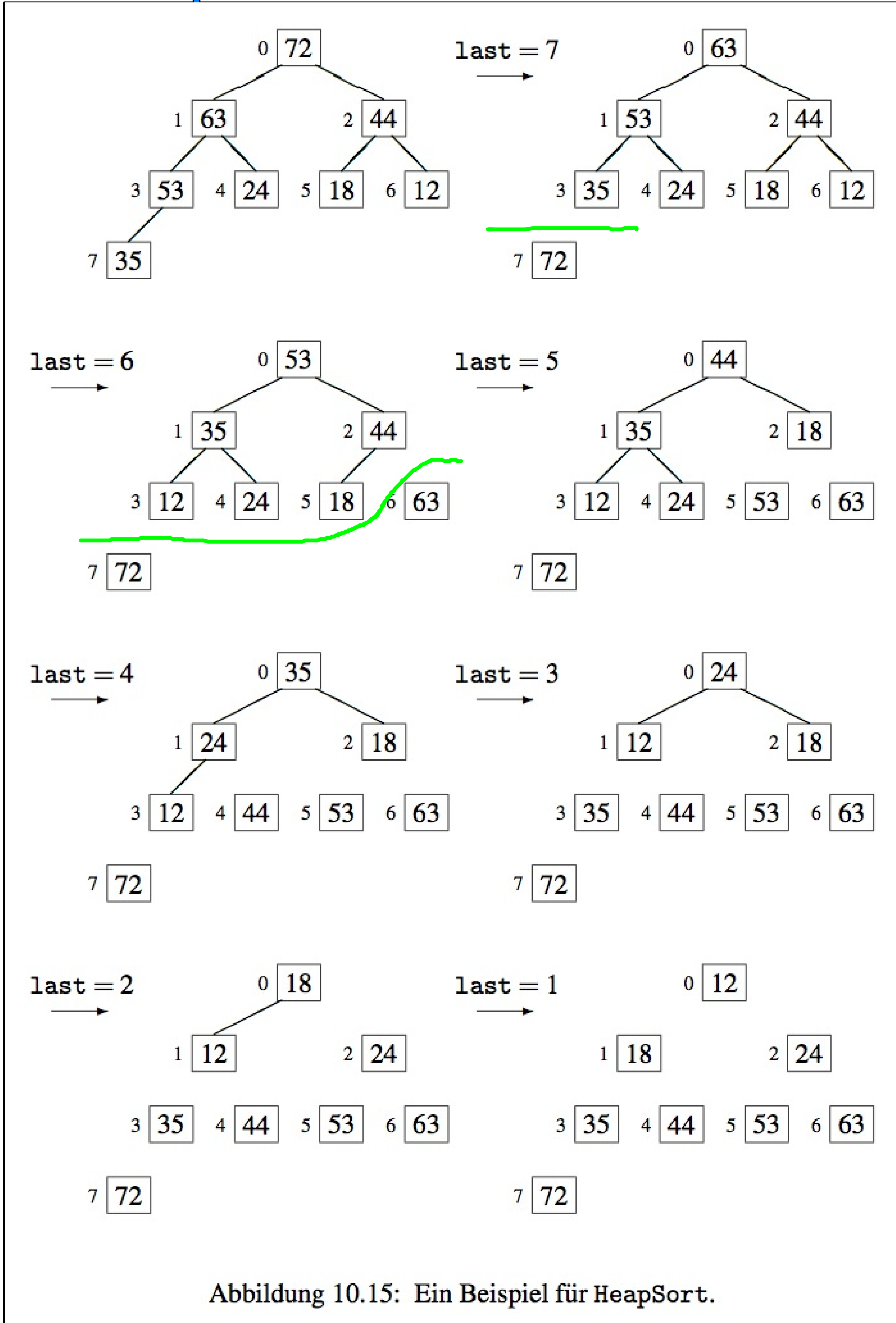
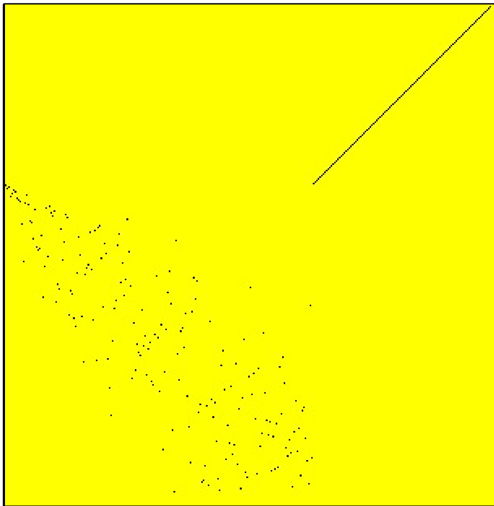


Abbildung 10.15: Ein Beispiel für HeapSort.

```java
/**
 * sorts array by heapsort in a certain range
 * @param vec the array in which this happens
 */
public static void heapSort(Item[] vec)
                        throws NullPointerException {
        if (vec == null) throw new NullPointerException();

        Item temp;
        int last;
        int n = vec.length;

        createHeap(vec);
        for (last = n-1; last > 0; last--) {
                // exchange top component with
                // current last component of vec
                temp = vec[0];
                vec[0] = vec[last];
                vec[last] = temp;
                // call heapify to to reestablish heap property
                heapify(vec, 0, last-1);
        }//endfor
}
```



Aufwand für Heapsort:

$O(n)$ + n Aufrufe von Heapify für Element an Stelle 0.

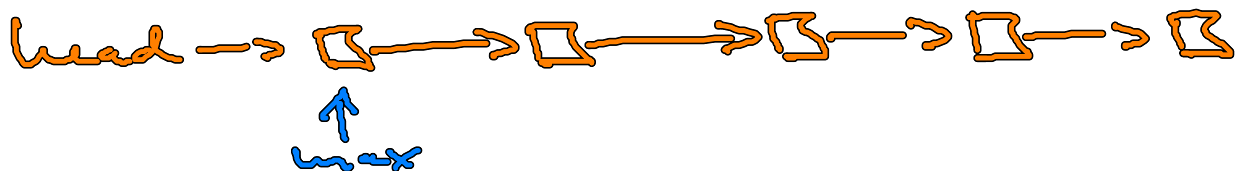
$\leq O(n) + n \cdot O(\log n) \in O(n \log n)$.

Priority Queues: <u>dynamische</u> Datenstrukt.

insert()
maximum()
extractMax()
changeKey()

Implementation auf unterschiedliche Arten möglich

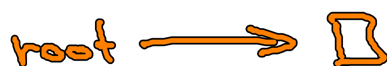
<u>Bsp:</u> Sortierte Liste (absteigend)

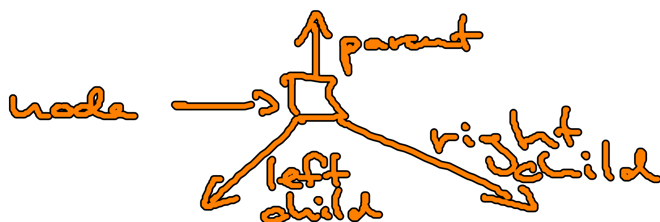


Aufwand für 4 Operationen:

insert(): $O(n)$
maximum(): $O(1)$
extractMax(): $O(1)$
changeKey(): $O(n)$

<u>Bsp:</u> Implementation durch vollen Binären Baum:





Aufwand für 4 Operationen:

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

Skizze

maximum(): $O(1)$

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

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

Untere Komplexitätsschranken für das Sortieren:

bisher: MergeSort, HeapSort, QuickSort — sortieren in $O(n \log n)$ Zeit im worst case bzw. average case

Frage: geht's auch schneller?

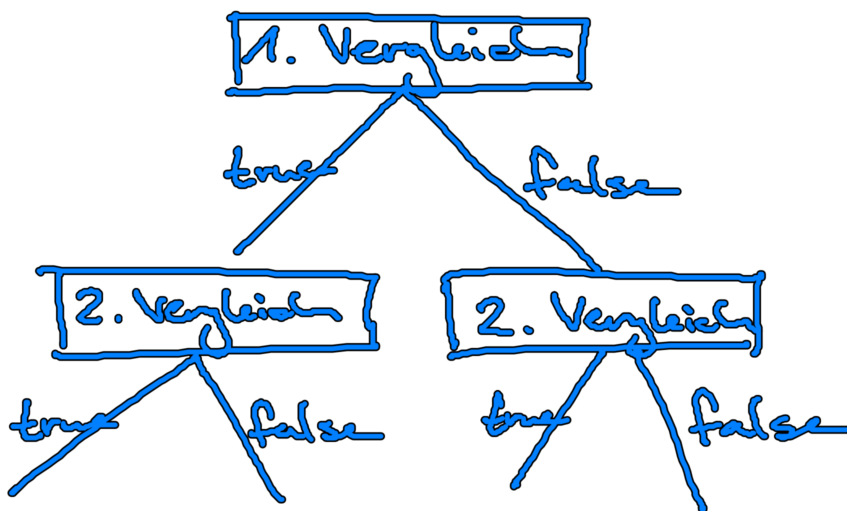
Antwort: Nein!

Theorem: Jeder Sortieralgorithmus, der auf paarweisem Vergleich beruht, braucht $\Omega(n \log n)$ Vergleiche im worst case und im average case.

(Anderes Rechenmodell: z.B. „Spaghetti-Computer" sortiert in Linearzeit)

Beweis: (nur für deterministische Algorithmen)

Bei deterministischen Algorithmen steht der erste Vergleich fest. Jeder weitere Vergleich ist eindeutig durch die Ergebnisse der bisherigen Vergleiche bestimmt:



$\Longrightarrow$ Jeder determin. Alg., der auf paarw. Vergleich beruht, erzeugt einen

Entscheidungsbaum.

- Wurzelknoten ≙ 1. Vergleich
- alle Knoten, außer Blättern, entspr. Vergleichen. Ist der Vergleich "true", so geht der Alg. weiter zum linken Kind, sonst zum rechten.
- Ein Lauf des Alg. entspricht also einem absteigenden Pfad von der Wurzel zu einem Blatt.
- Bätter ≙ Permutationen der Eingabefolge, insbesondere hat der Baum genau n! Blätter.