

Sarah Ertel
Patrick Greher
Eugen Ljavin

1	2	3	4	Σ

Übungsblatt Nr. 3

(Abgabetermin 10.05.2018)

Aufgabe 1

a)

Algorithm 1: Insertion Sort Algorithmus

```

1 function insertionSort(toSort [ ])
2   for  $i \leftarrow 1; i < toSort.length; i \leftarrow i + 1$  do
3      $j \leftarrow i;$ 
4     while  $(j > 0) \wedge (toSort[j - 1] > toSort[j])$  do
5        $tmp \leftarrow toSort[j - 1];$ 
6        $toSort[j-1] \leftarrow toSort[j];$ 
7        $toSort[j] \leftarrow tmp;$ 
8        $j \leftarrow j-1;$ 
9     end
10  end
```

Algorithm 2: Minimumsuche + Austausch Algorithmus

```

1 function minimumSwapSort(toSort [ ])
2   for  $i \leftarrow 0; i < toSort.length - 1; i \leftarrow i + 1$  do
3     for  $j \leftarrow i + 1; j < toSort.length; i \leftarrow j + 1$  do
4       if  $toSort[i] > toSort[j]$  then
5          $tmp \leftarrow toSort[i];$ 
6          $toSort[i] \leftarrow toSort[j];$ 
7          $toSort[j] \leftarrow tmp;$ 
8       end
9     end
10  end
```

b)

c)

	Minimumsuche + Austausch Algorithmus	Insertion Sort
Vertauschungen	0	0
Vergleiche	maximal: $\frac{n^2}{2} - \frac{n}{2}$	$n - 1$

d)

$n \in \mathbb{N}$

	Minimumsuche + Austausch Algorithmus	Insertion Sort
Array	$A = \langle n, n + 1, n + 2, \dots \rangle$	$A = \langle n, n - 1, n - 2, \dots \rangle$
Anzahl Vergleiche	$\frac{n^2}{2} - \frac{n}{2}$	$\frac{n^2 - n}{2}$

e)

 $n \in \mathbb{N}$

	Minimumsuche + Austausch Algorithmus	Insertion Sort
Array	$A = \langle n, n - 1, n - 2, \dots \rangle$	$A = \langle n, n - 1, n - 2, \dots \rangle$
Anzahl Vertauschungen	$\frac{n^2}{2} - \frac{n}{2}$	$\frac{n^2 - n}{2}$

Aufgabe 2

$A = \langle 4, 2, 12, 10, 18, 14, 6, 16, 8 \rangle$

a)

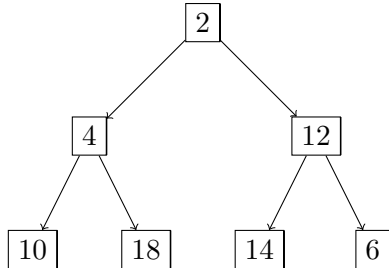
4

1) Die erste Zahl aus dem Array nehmen und als Wurzel einsetzen

2

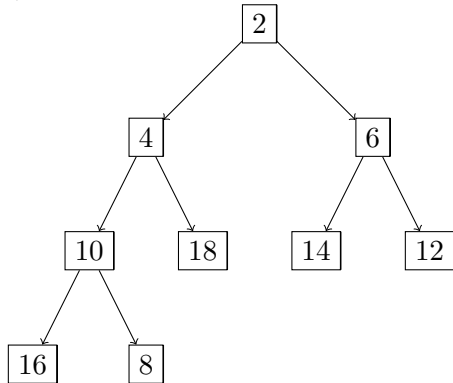
4

3) 4 und 2 vertauschen, da $2 < 4$



5) Die nächsten vier Elemente werden angefügt

(10,18,14 sind größer als die jeweiligen Parent Elemente)

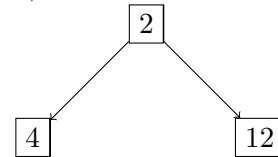


6) Die letzten beiden Elemente werden angefügt

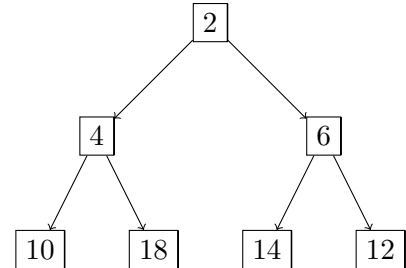
4

2

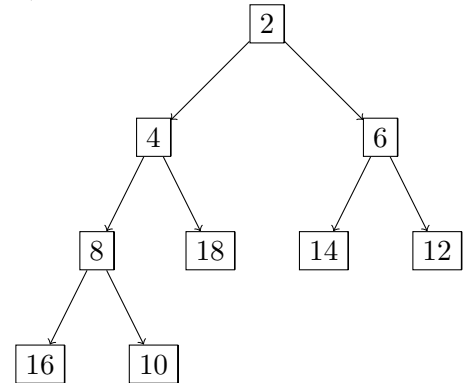
2) Die nächste Zahl aus dem Array als Child a



4) nächste Zahl als Child anfügen

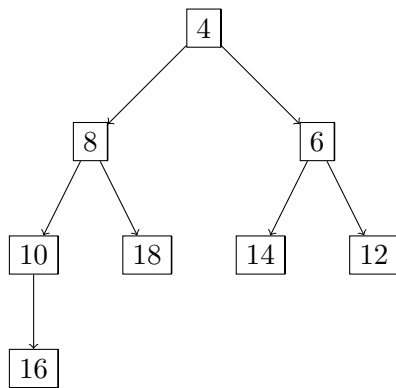


6) Da $6 < 12$ müssen die beiden Elemente vert

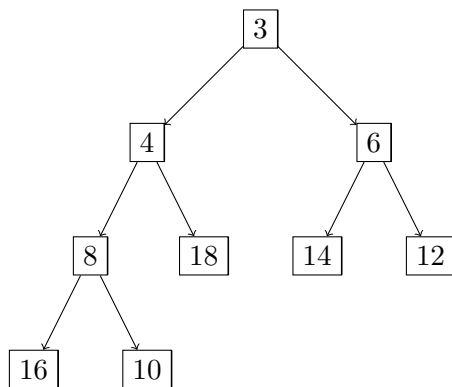


7) 8 und 10 müssen vertauscht werden

b)



c)



d)

Das größte Element könnte sich an jedem Knoten des Min-Heap Baumes befinden. Man muss also den Baum traversieren, bis man den maximalen Knoten gefunden hat. Anschließend muss die Min-Heap Eigenschaft wiederhergestellt werden.

- Um den maximalen Knoten zu finden muss jeder Knoten betrachtet werden $\Rightarrow \mathcal{O}(n)$
- Der gefundene Knoten muss mit dem letzten Element des Min-Heap Baumes ersetzt werden $\Rightarrow \mathcal{O}(1)$
- Die Min-Heap Eigenschaft muss wiederhergestellt werden $\Rightarrow \mathcal{O}(\log n)$

Es ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(n) + \mathcal{O}(1) + \mathcal{O}(\log n) = \mathcal{O}(n)$.

Aufgabe 3

a)

Ein k-Heap kann wie ein Binärer-Heap als Array dargestellt werden.

Der Parent node eines Elementes i lässt sich mit $\lfloor (i-1)/k \rfloor$, die Child nodes mit $i * k + 1$ bis $i * k + k$ berechnen.

b)

Die Höhe eines k -Heaps der Größe n ist $\lfloor \log_k(n) + 1 \rfloor$

c)**Algorithm 3: Insert**

```

1 function insertElement(Element e, k-nary k, Heap data)
2   positionElement, heapSize  $\leftarrow$  heapSize+1
3   data[heapSize-1]  $\leftarrow$  e
4   for  $parent = \lfloor (positionElement - 1)/k \rfloor$ ,  $data[positionElement] > parent$ ; do
5     | vertausche (data[positionElement] mit data[parent] positionElement  $\leftarrow$  parent
6   end

```

Algorithm 4: ExtractMin

```

1 function ExtractMin(Heap data, k-nary k)
2   position  $\leftarrow$  0
3   lösche data[position]
4   data[position]  $\leftarrow$  data[heapSize-1]
5   while true do
6     | children[]  $\leftarrow$  data[position*k+1] - data[position*k+k]
7     | for  $index \leftarrow 0$ ,  $index < k$ ,  $index+1$  do
8       |   if  $children[index] < data[position]$  then
9         |     | vertausche(children[index],data[position])
10        |     | position  $\leftarrow$  index index  $\leftarrow$  k
11      |   end
12    | end
13 end

```

Aufgabe 4**a)**

Für das Array der Länge $n = 2$ sortiert der Algorithmus korrekt, da die Elemente ggfs. vertauscht werden um sie zu sortieren und der Algorithmus terminiert. Bei der Länge $n = 1$ terminiert der Algorithmus direkt.

Für $n > 2$ gilt:

Nach dem Ausführen der Zeile 8 ist der Bereich $A[1 \dots n - k]$ sortiert, sodass die größten Elemente im hinteren Bereich des zweiten Drittels liegen. Nach dem Ausführen der Zeile 9 ist der Bereich $A[1 + k \dots n]$ sortiert, sodass die größten Elemente am Ende des Arrays stehen. Das zweite Dittel (mittlerer Bereich) ist nun nicht mehr sortiert. Nach dem Ausführen der Zeile 10 ist der Bereich $A[1 \dots n - k]$ wieder sortiert. Da die größten Elemente durch Ausführen der Zeile 9 in das letzte Drittel des Arrays gebracht wurden ist folglich der Bereich $A[1 \dots n]$ und somit das gesamte Array sortiert.

b)

Der Algorithmus ruft sich rekursiv drei mal auf und betrachtet dabei $\frac{2}{3}$ der Arraylänge n . Des Weiteren werden zwei Vergleiche durchgeführt, um zu prüfen, ob zwei Elemente getauscht werden müssen sowie für die Abbruchbedingung. Es ergibt sich daraus die Rekursionsvorschrift $T(n) = 3 \cdot T\left(\frac{2}{3} \cdot n\right) + 1$

Mit Hilfe des Mastertheorems lässt sich folgende Komplexität ermitteln:

$$T(n) = 3 \cdot T\left(\frac{2}{3} \cdot n\right) + 1 \Rightarrow 3 \cdot T\left(\frac{2}{3} \cdot n\right) + \mathcal{O}(1)$$

Nach dem Mastertheorem gilt: $a = 3$; $b = \frac{3}{2}$; $f(n) = 1 = \mathcal{O}(n^c)$ mit $c = 0$

Es ist: $c < \log_{\frac{3}{2}} 3$

Nach Fall 1 des Mastertheorems gilt: $T(n) = \mathcal{O}(n^{\log_{\frac{3}{2}} 3}) \approx \mathcal{O}(n^{2,7})$

c)

Sowohl Quick-Sort, als auch Minimumsuche + Austausch sowie Insertion Sort haben im worst case eine Komplexität von $\mathcal{O}(n^2)$.

Setzt man in dieser Teilaufgabe Komplexität mit Effizienz gleich, gilt $\mathcal{O}(n^2) < \mathcal{O}(n^{2,7})$.

Damit ist Zwei-Drittel-Sortieren im worst case **nicht** effizienter als die drei obenstehenden Sortieralgorithmen.