Prof. Dr. Ir. Joost-Pieter Katoen

Christian Dehnert, Friedrich Gretz, Benjamin Kaminski, Thomas Ströder

Tutoraufgabe 1 (Sortieralgorithmus):

Gegeben sei der folgende Sortieralgorithmus:

```
void sort(int E[]) {
  int i,j,m;
  for (i = 0; i < E.length; i++) {
    m = i;
    for (j = i + 1; j < E.length; j++) {
      if (E[j] <= E[m]) {
          m = j;
      }
    }
    int v = E[i];
    E[i] = E[m];
    E[m] = v;
  }
}
```

a) Nutzen Sie den gegebenen Algorithmus, um das folgende Array zu sortieren. Geben Sie den Zustand des Arrays nach jedem Durchlauf der äußeren Schleife an.

1	3	2	7	0	4	8	5	7	6

b) Geben Sie in wenigen Worten wieder, wie der gegebene Algorithmus funktioniert. In der Vorlesung wurden mehrere Sortierverfahren genannt (siehe Folien). Welcher Name passt zu diesem Algorithmus?

c) Ist der Sortieralgorithmus stabil? Falls dies nicht der Fall ist, geben Sie an, wie er angepasst werden muss, damit er stabil wird!

d) Welche Average-Case Laufzeit besitzt der gegebene Sortieralgorithmus für eine Eingabe der Länge n? Geben Sie die Komplexitätsklasse $\Theta(T_{sort}(n))$ für ein Array E mit Länge `n = E.length` an und begründen Sie Ihre Antwort.

Lösung:

a) In den folgenden Schritten sortiert der Algorithmus das Array:

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

b) Der Algorithmus sortiert das Array von vorne nach hinten indem er jeweils das kleinste Element aus dem noch zu sortierenden Teil sucht und dieses mit dem ersten Element dieses Bereiches tauscht. Dadurch wird von vorne nach hinten ein sortiertes Array aufgebaut.

Selectionsort ist ein geeigneter Name für dieses Verfahren, da jeweils das **kleinste** Element aus den übrigen Elementen **ausgesucht** wird.

c) Der Algorithmus ist nicht stabil. Um einen stabilen Algorithmus zu erhalten können wir das Vertauschen der Elemente durch ein Einfügen ersetzen, wie wir es von Insertionsort kennen. Zusätzlich muss der Vergleich der Elemente in Zeile 6 strikt sein.

d) Unabhängig von den jeweiligen Schlüsseln und ihrer Verteilung im Array wird die äußere Schleife n-fach durchlaufen, die innere Schleife jedesmal von der aktuellen Position bis zum Ende. Das Vertauschen der Werte geschieht in konstanter Zeit, da bei diesem Algorithmus das Aufschieben der Elemente entfällt. Es ergibt sich eine Laufzeitkomplexität von:

$$T_{sort(E)} \in \Theta(\sum_{i=0}^{n} i) = \Theta(n^2)$$

Tutoraufgabe 2 (Mergesort):

Sortieren Sie das folgende Array mithilfe von Mergesort aus der Vorlesung. Geben Sie dazu das Array nach jeder Merge-Operation an.

Sortieren Sie das folgende Array mithilfe von Mergesort. Geben Sie dazu das Array nach jeder Merge-Operation an.

3	2	8	4	1	5	6	7	4

Lösung:

Die grau unterlegten Zeilen dienen nur zur Veranschaulichung, an welchen Stellen das Array aufgeteilt wird. Sie sind zur Lösung der Aufgabe nicht nötig.

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

Tutoraufgabe 3 (Max- und Min-Heaps):

a) Bestimmen Sie, ob folgende Arrays Heaps (Max-Heaps) sind. Falls nicht, geben Sie an wo die Heapeigenschaft verletzt ist.

1.)

56	47	56	10	20	50	51	1	2	3	18

2.)

56	56	47	10	20	50	51	1	2	3	18

3.)

56	47	56	10	51	20	50	1	2	3	18

4.)

56	47	56	10	5	20	50	1	2	3	18

b) In der Vorlesung wurden so genannte Max-Heaps vorgestellt. D.h jedes Element ist größer/gleich seiner Kinder. Ein *Min-Heap* ist ein Heap bei dem jedes Element kleiner/gleich seiner Kinderelemente ist.
Bestimmen sie, ob die folgenden Arrays Min-Heaps sind, und falls nicht, geben sie an, wo die Heapeigenschaft verletzt ist

1.)

1	5	1	8	10	4	15	11	12	14	11

2.)

0	1	5	8	10	4	15	11	12	14	11

3.)

1	5	1	11	8	4	15	11	12	14	10

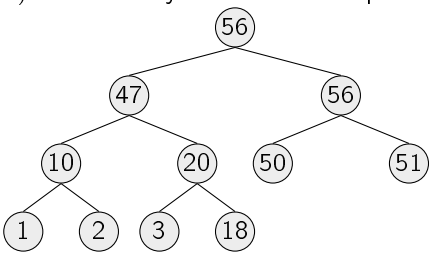
4.)

1	5	1	11	15	4	8	11	12	14	10

Lösung:

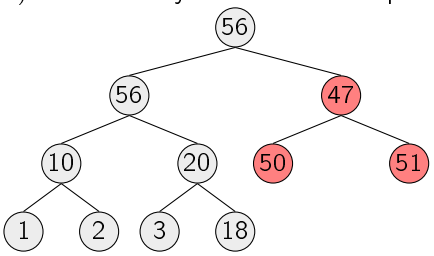
a) Die folgenden Heaps sind in den gegebenen Arrays repräsentiert:

1) Dieses Array ist ein Max-Heap.



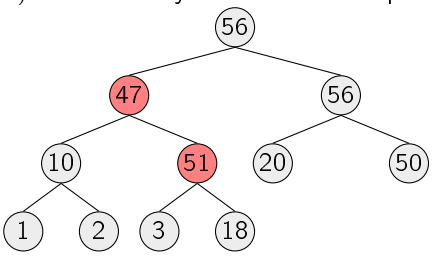
56 47 56 10 20 50 51 1 2 3 18

2) Dieses Array ist kein Max-Heap.



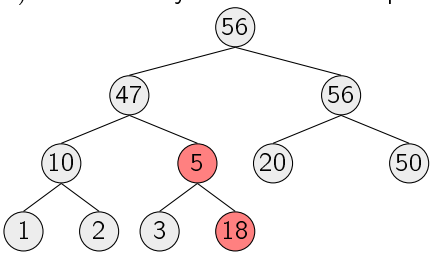
56 56 47 10 20 50 51 1 2 3 18

3) Dieses Array ist kein Max-Heap.



56 47 56 10 51 20 50 1 2 3 18

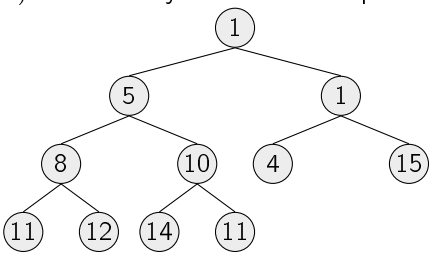
4) Dieses Array ist kein Max-Heap.



56 47 56 10 5 20 50 1 2 3 18

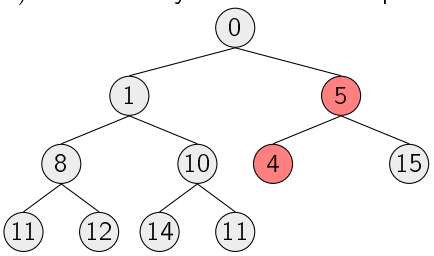
b) Die folgenden Heaps sind in den gegebenen Arrays repräsentiert:

1) Dieses Array ist ein Min-Heap.



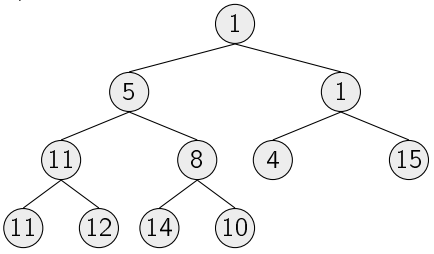
1 5 1 8 10 4 15 11 12 14 11

2) Dieses Array ist kein Min-Heap.



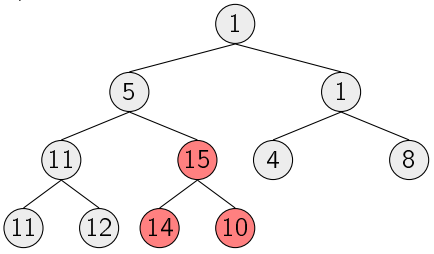
0 1 5 8 10 4 15 11 12 14 11

3) Dieses Array ist ein Min-Heap.



1 5 1 11 8 4 15 11 12 14 10

4) Dieses Array ist kein Min-Heap.

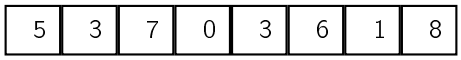


1 5 1 11 15 4 8 11 12 14 10

Tutoraufgabe 4 (Heapsort):

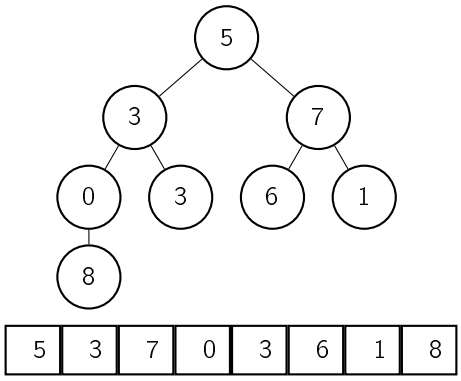
Sortieren Sie das folgende Array mithilfe von Heapsort aus der Vorlesung. Geben Sie dazu das Array nach jeder Swap-Operation an und geben Sie zum jeweils noch unsortierten Arraybereich zusätzlich die grafische Darstellung als Heap an.

Sortieren Sie das folgende Array mithilfe von Heapsort. Geben Sie dazu das Array nach jeder Swap-Operation an.

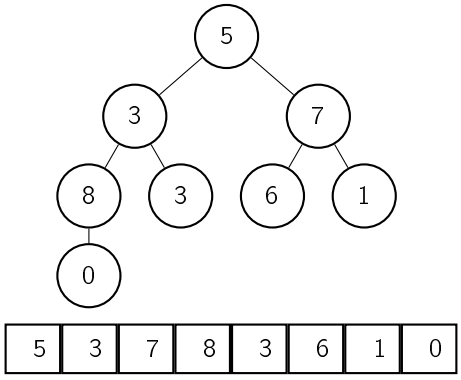


Lösung:

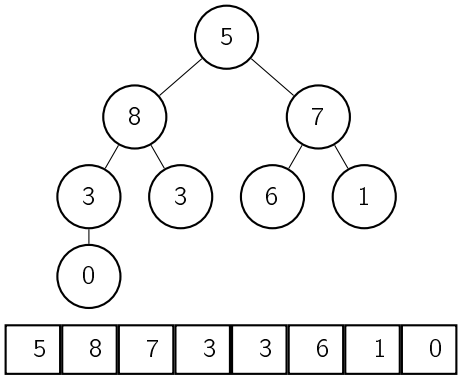
Schritt 0:



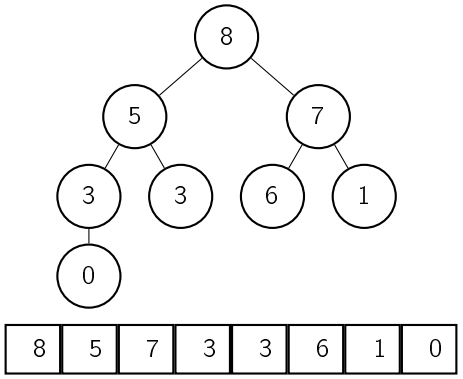
Schritt 1:



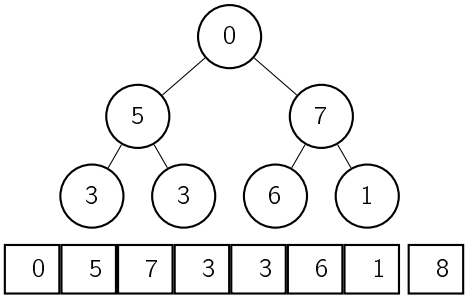
Schritt 2:



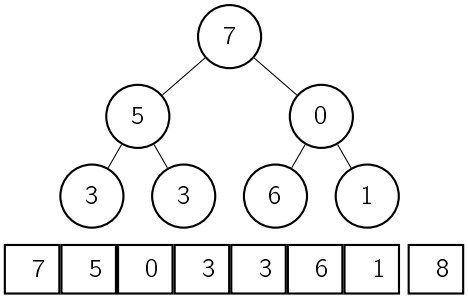
Schritt 3:



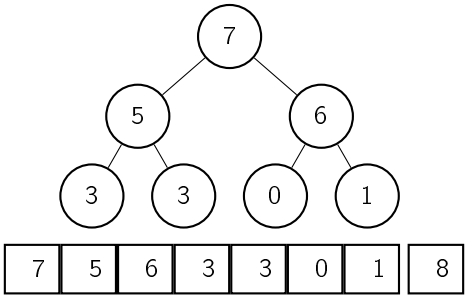
Schritt 4:



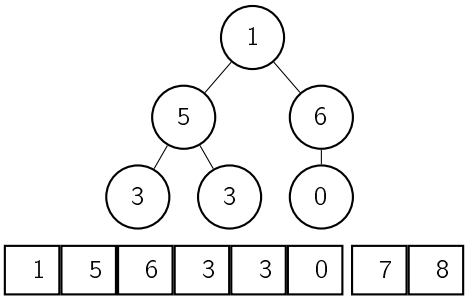
Schritt 5:



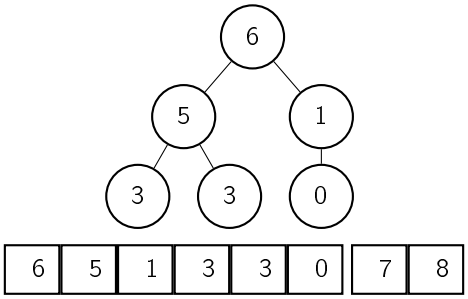
Schritt 6:



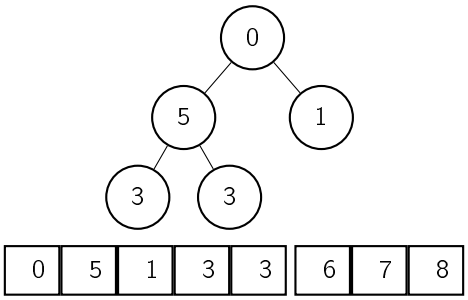
Schritt 7:



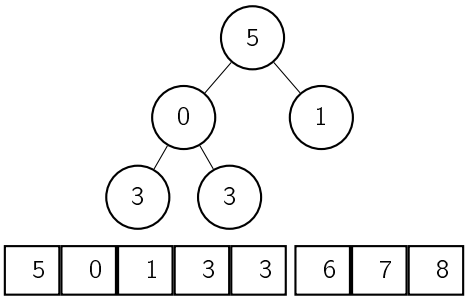
Schritt 8:



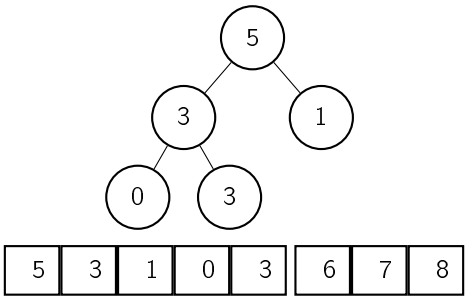
Schritt 9:



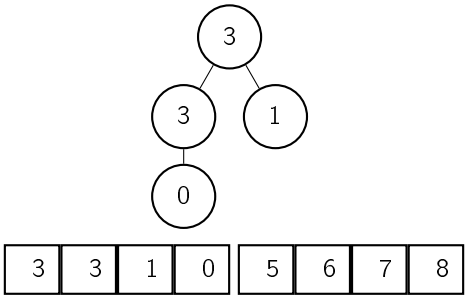
Schritt 10:



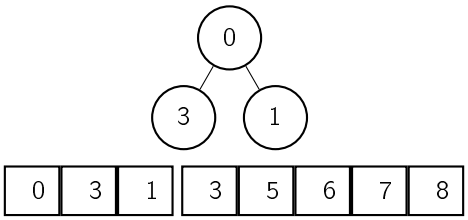
Schritt 11:



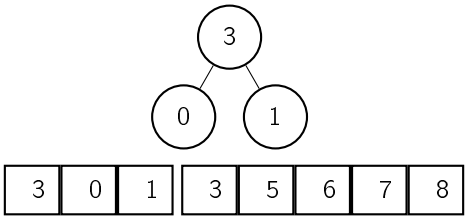
Schritt 12:



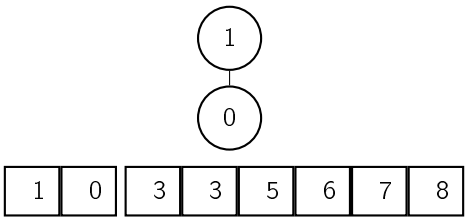
Schritt 13:



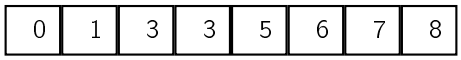
Schritt 14:



Schritt 15:



Schritt 16:



Tutoraufgabe 5 (Quicksort):

Sortieren Sie das folgende Array mithilfe von Quicksort aus der Vorlesung. Geben Sie dazu das Array nach jeder Partition-Operation an.

Sortieren Sie das folgende Array mithilfe von Quicksort. Geben Sie dazu das Array nach jeder Partition-Operation an und markieren Sie das jeweils verwendete Pivot-Element.

8	2	4	7	5	6	1	3

Lösung:

Die jeweils verwendeten Pivot-Elemente sind grau unterlegt.

8	2	4	7	5	6	1	3

1	2	**3**	7	5	6	8	4

1	**2**	3	7	5	6	8	4

1	2	3	**4**	5	6	8	7

1	2	3	4	5	6	**7**	8

1	2	3	4	5	**6**	7	8

Aufgabe 6 (Rekursiver Sortieralgorithmus): (2 + 3 = 5 Punkte)

```
sort(int [ ] A, int l, int r) {
  if (A[l] > A[r]){
    exchange(A[l], A[r]);
  }
  if (l < r-1){
    k:= (r-l+1) div 3;
    sort(A, l, r-k);
    sort(A, l+k, r);
    sort(A, l, r-k);
  }
}
```

a) Bestimmen Sie für den gegebenen Sortieralgorithmus die Komplexitätsklasse Θ im Best-, Worst- und Average-Case für den Aufruf `sort(A,0,n-1)` in Abhängigkeit von `n`, der Anzahl der zu sortierenden Elemente aus dem Array `A`. Dabei verursacht ein Vergleich zwischen Arrayelementen eine Kosteneinheit, alle anderen Operationen sind "kostenlos".

b) Der vorgestellte Algorithmus ist nicht stabil. Ändern Sie den Algorithmus so ab, dass er stabil wird.

Lösung:

a) Die Laufzeit von `sort` ist im Best-, Worst- und Average-Case gleich. Sie kann mit folgender Rekursionsgleichung beschrieben werden:

$$T(n) = 3 \cdot T(\frac{2}{3}n) + 1$$

Dabei steht $3 \cdot T(\frac{2}{3}n)$ für die Komplexität der Rekursionsaufrufe und 1 für die Komplexität der Vergleiche.

Zur Bestimmung der Komplexitätsklasse verwenden wir das Mastertheorem.

Es gilt $b = 3$, $c = 1{,}5$ und $f(n) = c \in \mathcal{O}(n^0)$. Daraus ergibt sich $E = \log_{1,5} 3$, so dass $f(n) \in \mathcal{O}(n^{E-\epsilon})$. Dies ist der 1. Fall und wir erhalten:

$$T(n) \in \Theta(n^{\log_{1,5} 3})$$

b) Der Suchalgorithmus `sort` ist nicht stabil. Vor dem rekursiven Aufrufen, also bevor die Elemente im gegebenen Bereich sortiert wurden, werden gegebenenfalls bereits die beiden äußeren Einträge vertauscht. Hierbei kann es passieren, dass ein Element über eins mit gleichem Schlüssel, das sich innerhalb des Bereichs befindet, hinweg bewegt wird - die ursprüngliche Sortierung ist dann nichtmehr gewährleistet. Es reicht jedoch, nur im Basisfall, d.h. wenn sich nur zwei Elemente im zu sortierenden Bereich befinden, diese per vertauschen zu sortieren. Wird der Sortieralgorithmus entsprechen implementiert (siehe den folgenden `sort2`) kann ein Element nicht über eins mit gleichem Schlüssel hinweg vertauscht werden, der Algorithmus ist somit stabil.

```
sort2(int [ ] A, int l, int r) {

  if (l < r-1) {
    k:= (r-l+1) div 3;
    sort2(A, l, r-k);
    sort2(A, l+k, r);
    sort2(A, l, r-k);

  } else if (A[l] > A[r]) {
     exchange(A[l], A[r]);
  }
}
```

Aufgabe 7 (Mergesort): (3 Punkte)

Sortieren Sie das folgende Array mithilfe von Mergesort aus der Vorlesung. Geben Sie dazu das Array nach jeder Merge-Operation an.

Sortieren Sie das folgende Array mithilfe von Mergesort. Geben Sie dazu das Array nach jeder Merge-Operation an.

4	1	2	6	8	3	7

Lösung:

4	1	2	6	8	3	7

1	4	2	6	8	3	7

1	4	2	6	8	3	7

1	2	4	6	8	3	7

1	2	4	6	3	8	7

1	2	4	6	3	7	8

1	2	3	4	6	7	8

Aufgabe 8 (Quicksort): (3 Punkte)

Sortieren Sie das folgende Array mithilfe von Quicksort aus der Vorlesung. Geben Sie dazu das Array nach jeder Partition-Operation an.

Sortieren Sie das folgende Array mithilfe von Quicksort. Geben Sie dazu das Array nach jeder Partition-Operation an und markieren Sie das jeweils verwendete Pivot-Element.

7	0	4	9	8	6	5	3	1	2

Lösung:

Die jeweils verwendeten Pivot-Elemente sind grau unterlegt.

7	0	4	9	8	6	5	3	1	2

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

0	1	2	9	8	6	5	3	7	4

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

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

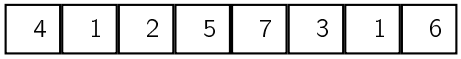
0	1	2	3	4	6	5	**7**	8	9

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

Aufgabe 9 (Heapsort): (5 Punkte)

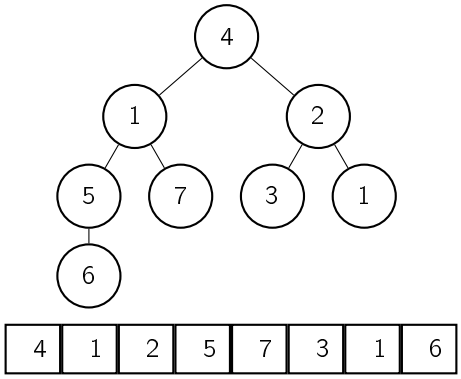
Sortieren Sie das folgende Array mithilfe von Heapsort aus der Vorlesung. Geben Sie dazu das Array nach jeder Swap-Operation an und geben Sie zum jeweils noch unsortierten Arraybereich zusätzlich die grafische Darstellung als Heap an.

Sortieren Sie das folgende Array mithilfe von Heapsort. Geben Sie dazu das Array nach jeder Swap-Operation an.

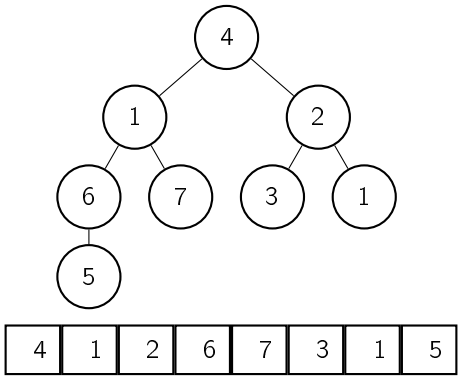


Lösung:

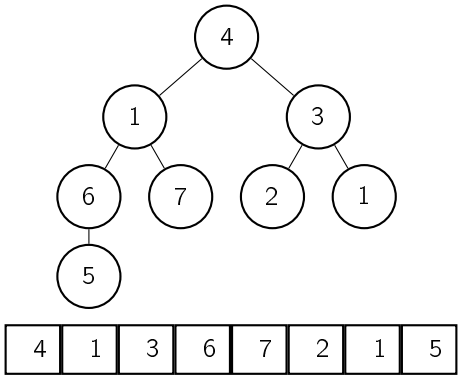
Schritt 0:



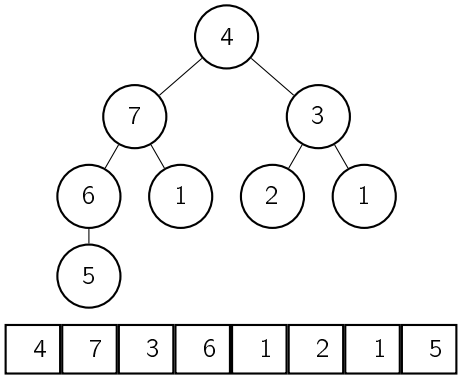
Schritt 1:



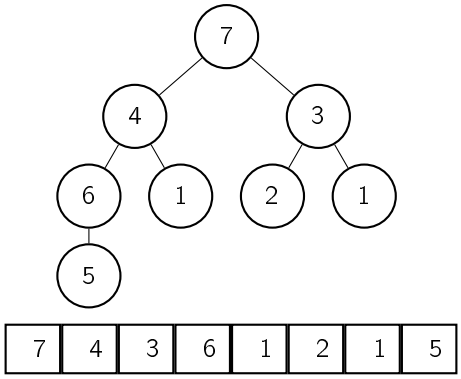
Schritt 2:



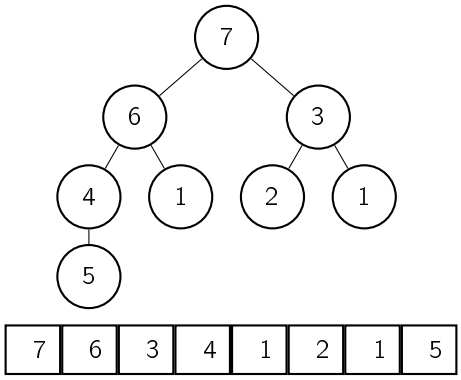
Schritt 3:



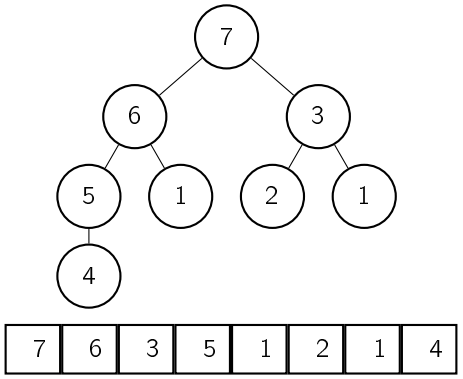
Schritt 4:



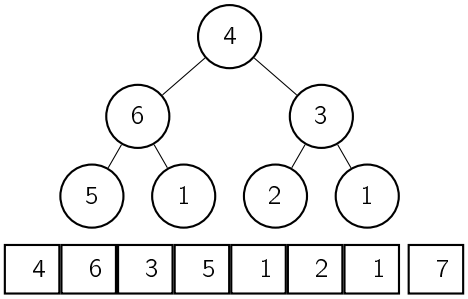
Schritt 5:



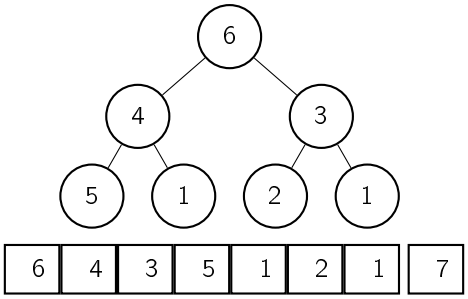
Schritt 6:



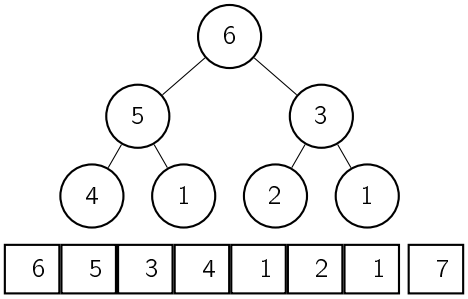
Schritt 7:



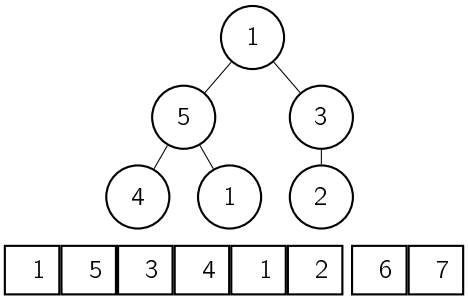
Schritt 8:



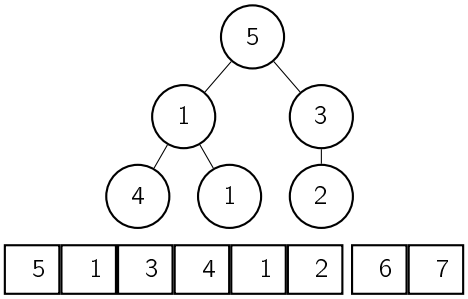
Schritt 9:



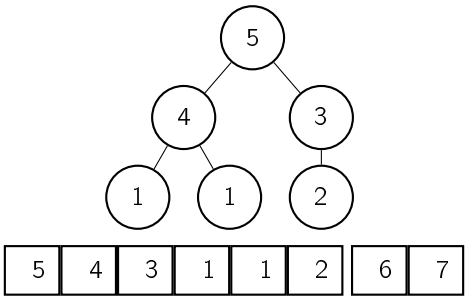
Schritt 10:



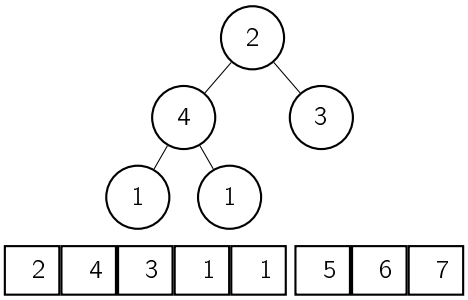
Schritt 11:



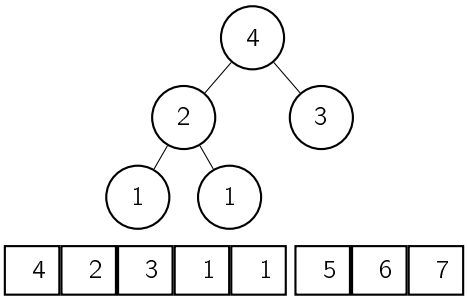
Schritt 12:



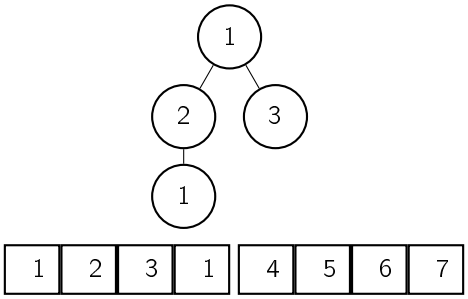
Schritt 13:



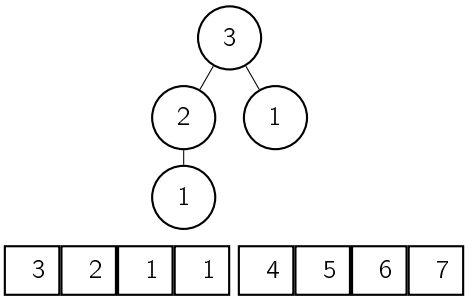
Schritt 14:



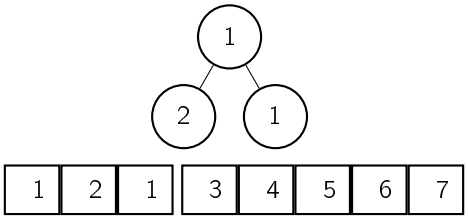
Schritt 15:



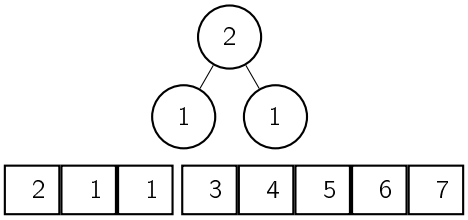
Schritt 16:



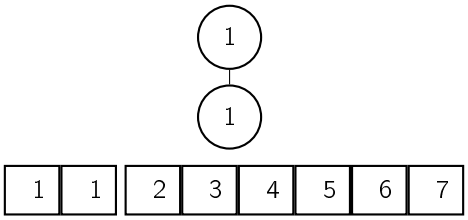
Schritt 17:



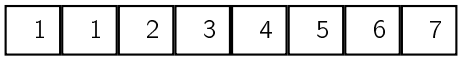
Schritt 18:



Schritt 19:



Schritt 20:



Aufgabe 10 (Beweis der Lemmata aus der Vorlesung): (2 + 4 + 3 = 9 Punkte)

a) Beweisen Sie, dass ein Heap mit n Elementen die Höhe $\lfloor \log_2 n \rfloor$ hat.
Hinweis: Sie dürfen als bekannt voraussetzen, dass die minimale Höhe eines **Binärbaumes** *mit n Elementen* $h = \lceil \log_2(n+1) \rceil - 1$ *ist.*

b) Beweisen Sie, dass ein Heap mit n Elementen $\lceil \frac{n}{2} \rceil$ Blätter besitzt.
Was sagt das über die Anzahl der inneren Knoten des Heaps?

c) Beweisen Sie, dass ein Heap maximal $\lceil n/2^{h+1} \rceil$ Knoten mit der Höhe h hat.
Hinweis: Die Höhe h des Knotens entspricht der Länge des längsten Pfades zu einem Blatt des Heaps.

Lösung:

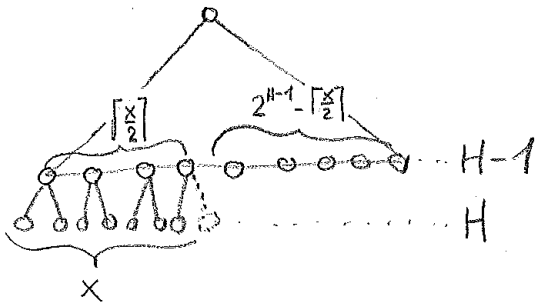
a) Da die Ebenen des Heaps, bis auf die letzte, immer maximal besetzt sind, hat ein Heap immer die minimale Höhe eines Binärbaumes. Wir "wissen", dass die minimale Höhe eines Binärbaumes mit n Elementen $h = \lceil \log_2(n+1) \rceil - 1$ ist. Dies lässt sich wie folgt in $h = \lfloor \log_2 n \rfloor$ umformen:

Gegeben $h = \lceil \log_2(n+1) \rceil - 1$ wir unterscheiden ob es ein m mit $n = 2^m$ gibt oder nicht, d.h. ob es sich bei n um eine Zweierpotenz handelt:

$n = 2^m$: $\lceil \log_2(n+1) \rceil - 1 = \lceil \underbrace{\log_2(n)}_{m \in \mathbb{N}} + \underbrace{\varepsilon}_{\leq 1} \rceil - 1 = \lfloor \log_2(n) \rfloor + 1 - 1 = \lfloor \log_2(n) \rfloor$

$n \neq 2^m$: Durch das hinzuaddieren von 1 könnte eine zweier Potenz erreicht werden, jedoch kann keine überschritten werden. Somit gilt: $\lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2(n) \rfloor$.

b) Da alle, bis auf die letzte Ebenen eines Heaps voll besetzt sind können sich Blätter nur in den letzten beiden Ebenen befinden. Sei H die Höhe des Heaps, d.h. der Heap besitzt H Ebenen. Alle Blätter befinden sich auf den Ebenen H und $H-1$. Ebene $H-1$ ist, wie bereits bemerkt, voll besetzt. Die letzte Ebene H jedoch muss nicht voll besetzt sein. Sei x die Anzahl der Knoten auf Ebene H. Da es $2^H - 1$ viele Knoten auf den Ebenen 0 bis $H-1$ gibt gilt dann, dass es $n = 2^H - 1 + x$ Knoten im Heap gibt. Betrachte hierzu auch die folgende Grafik:



Wir betrachten die Ebene $H-1$. Diese Ebene ist voll besetzt. Somit gibt es 2^{H-1} Knoten, von denen $\lceil \frac{x}{2} \rceil$ Eltern Knoten der x Knoten auf der Ebene H sind und somit innere Knoten. Somit ergibt sich eine Anzahl von $2^{H-1} - \lceil \frac{x}{2} \rceil$ Blätter in Ebene $H-1$ und die Gesamtzahl von Blättern ergibt sich als:

$$\underbrace{x + 2^{H-1}}_{ganzzahlig} - \lceil \frac{x}{2} \rceil = \lfloor x + 2^{H-1} - \frac{x}{2} \rfloor = \lfloor \frac{2x + 2^H - x}{2} \rfloor = \lfloor \frac{\overbrace{2^H + x}^{n+1}}{2} \rfloor = \lfloor \frac{n+1}{2} \rfloor \overset{*}{=} \lceil \frac{n}{2} \rceil$$

Für die Gleichheit (*) ist eine Fallunterscheidung nötig:

1. n ist gerade, d.h. es gibt ein m mit $n = 2m$ und es gilt:
   $\lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{2m+1}{2} \rfloor = m + \lfloor \frac{1}{2} \rfloor = m = \lceil m \rceil = \lceil \frac{n}{2} \rceil$
2. n ist ungerade, d.h. es gibt ein m mit $n = 2m + 1$ und es gilt:
   $\lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{2m+2}{2} \rfloor = m + \lfloor 1 \rfloor = m + 1 = \lceil m + 1 - \frac{1}{2} \rceil = \lceil \frac{2m+2}{2} - \frac{1}{2} \rceil = \lceil \frac{2m+1}{2} \rceil = \lceil \frac{n}{2} \rceil$

q.e.d.

Aus diesem Lemma folgt unmittelbar das folgende Lemma (Vorlesung), das wir im Weiteren nutzen werden:

Ein Heap mit n Elementen besitzt $\lfloor \frac{n}{2} \rfloor$ innere Knoten.

c) Wir beweisen nun die Aussage per Induktion über die Höhe h:

Induktionsverankerung für $h = 0$:

Die Knoten mit der Höhe $h = 0$ sind die Blätter des Heaps. Nach oben bewiesenem Lemma gibt es in dem Heap $\lceil \frac{n}{2} \rceil = \lceil \frac{n}{2^{0+1}} \rceil$ Blätter.

Induktionsschritt $h - 1 \mapsto h$:

Sei T' der Baum, den man nach dem Entfernen aller Blätter aus einem Baum T erhält. Somit besteht T' aus den inneren Knoten von Baum T. T' hat also $n' = \lfloor n/2 \rfloor$ Knoten.

Da sich für die inneren Knoten aus T die Höhe h, durch das Entfernen der Blätter, um eins verringert, haben sie in T' die Höhe $h - 1$. Sei n_h die Anzahl der Knoten mit Höhe h im Baum T und entsprechend n'_h die Anzahl im Baum T' dann gilt: $n_h = n'_{h-1}$

Durch Induktion folgt nun:
$n_h = n'_{h-1} \leq \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \leq \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$.

q.e.d