Technische Universität Darmstadt
Fachgebiet Theoretische Informatik

Prof. Johannes Buchmann
Nabil Alkeilani Alkadri
Nina Bindel
Patrick Struck

Algorithmen und Datenstrukturen

SoSe 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Lösungsblatt — 30.04.2018

P1 Quicksort

a) Quicksort sortiert die Elemente eines Arrays A, indem das Array in jedem Schritt in drei Teilarrays zerlegt wird:

- Das erste Teilarray enthält alle Elemente, die kleiner oder gleich dem Pivotelement sind (außer dem Pivotelement selbst).
- Das zweite Teilarray enthält das Pivotelement.
- Das dritte Teilarray enthält alle Elemente, die größer als das Pivotelement sind.

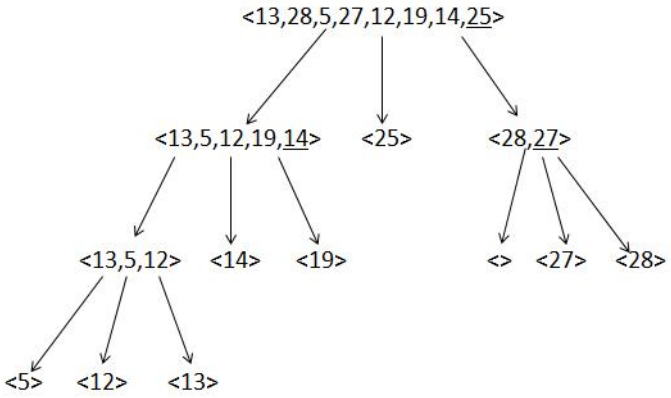
Dies wird solange fortgesetzt, bis alle Teilarrays die Länge 1 haben.

Illustrieren Sie diese Zerlegung anhand des Arrays $\langle 13, 28, 5, 27, 12, 19, 14, 25\rangle$. Verwenden Sie dabei als Pivotelement stets das letzte Element eines Arrays. Geben Sie zusätzlich das fertig sortierte Array an.

b) Geben Sie jeweils Arrays A an, für die die Laufzeit von Quicksort dem besten und schlechtesten Fall entspricht. Dabei soll A die Elemente $2, 13, 21, 9, 7, 17, 15$ enthalten. Das Pivotelement ist stets das letzte Element eines Arrays.
Geben Sie jeweils die Zahl der Vergleiche an, die Quicksort benötigt, um das Array A zu sortieren.

Lösung.

a) Das jeweilige Pivotelement ist unterstrichen.



Das sortierte Array hat die Form $A = \langle 5, 12, 13, 14, 19, 25, 27, 28\rangle$.

b) Bester Fall: $A = \langle 2, 15, 9, 7, 21, 17, 13\rangle$. Es werden 10 Vergleiche benötigt.
Schlechtester Fall: $A = \langle 21, 17, 15, 13, 9, 7, 2\rangle$. Es werden 21 Vergleiche benötigt.

P2 Heap

Die (binäre) Heap-Datenstruktur ist ein Array(-Objekt), welches ein fast vollständiger Binärbaum ist, welches man wie einen fast vollständigen Binärbaum wahrnehmen kann. So können von einem bestimmten Array-Element i respektive sein Eltern-Knoten, sein linker Kind-Knoten und sein rechter Kindknoten mit den Indizes $\lfloor i/2 \rfloor, 2i, 2i+1$ bestimmt werden (siehe Abbildung 1). Die Höhe eines Knotens im Heap wird als die Anzahl der Kanten auf dem längsten Weg zu einem Blatt definiert. Die Höhe des Baumes ist somit die Höhe des Wurzelknotens.

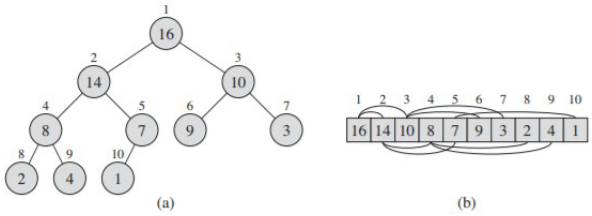


Abbildung 1: Ein Max-Heap, dargestellt als ein (a) Binärbaum und (b) Array.
Quelle: Figure 6.1, Introduction to Algorithms, Third edition.

a) Was ist die maximale und die minimale Anzahl an Elementen in einem Heap der Höhe h?

b) Zeigen Sie, dass ein Heap mit n Elementen eine Höhe von $\lfloor \log(n) \rfloor$ hat.

Lösung.

a) Mindestens 2^h und höchstens $2^{h+1}-1$. Ein kompletter Binärbaum der Höhe $h-1$ hat $\sum_{i=0}^{h-1}(2^i) = (2^h)-1$ Elemente und die Anzahl der Elemente in einem Heap der Höhe h liegt zwischen der Zahl für einen vollständigen binären Tiefenbaum der Höhe $h-1$ exklusiv und der Zahl in einem vollständigen Binärbaum der Höhe h inklusiv.

   $f(h) = \sum_{n=1}^{h} 2^i = 2^{h+1}-1$ Also folgt $n \in (f(h-1), f(h)] = (2^h-1, 2^{h+1}-1] = [2^h, 2^{h+1}-1]$

b) Schreibe $n = 2^m - 1 + k$ wobei m so groß wie möglich ist. Dann besteht der Heap aus einem vollständigen binären Baum der Höhe $m-1$, zusammen mit k zusätzlichen Blättern in der niedrigsten Ebene. Die Höhe der Wurzel ist die Länge des längsten einfachen Pfades (Anzahl der Kanten) zu einem dieser k Blätter, welcher die Länge m haben muss. Daher muss nach unserer Definition von m gelten, dass $m = \lfloor \log(n) \rfloor$.

P3 Sortieralgorithmen

Welche der folgenden Aussagen treffen zu? Begründen Sie Ihre Aussage. (Bei wahren Aussagen genügt es dabei, ein Beispielarray mindestens der Länge 8 und das Vorgehen des Sortierverfahrens darauf anzugeben. Bei falschen Aussagen bedarf es einer Begründung.)

a) Es gibt Fälle, in denen die Laufzeit von QuickSort in $O(n^2)$ ist.

b) Es gibt Fälle, in denen die Laufzeit von QuickSort in $O(n)$ ist.

c) Es gibt Fälle, in denen die Laufzeit von MergeSort in $O(n^2)$ ist.

Lösung.

a) Ja, z.B. eine vorsortierte Liste $\langle 1,2,3,4,5,6,7,8 \rangle$ und Pivotelement immer das erste Element (falls der Median nicht in $O(n)$ berechnet wird).

b) Nein. Die untere Grenze für Quicksort ist $O(n \log n)$ im Best Case.

c) Nein. MergeSort ist stabil bezüglich der Laufzeit. Sie liegt also immer in $O(n \log n)$.

P4 Max-Heap

a) In der Vorlesung wurden so genannte Max-Heaps vorgestellt. Benennen Sie die Heapeigenschaft die Max-Heaps erfüllen und bestimmen Sie, ob folgende Arrays Max-Heaps sind. Falls nicht, geben Sie an wo die Heapeigenschaft verletzt ist.

1. $\langle 56, 47, 56, 10, 20, 50, 51, 1, 2, 3, 18\rangle$
2. $\langle 56, 56, 47, 10, 20, 50, 51, 1, 2, 3, 18\rangle$
3. $\langle 56, 47, 56, 10, 51, 20, 50, 1, 2, 3, 18\rangle$
4. $\langle 56, 47, 56, 10, 5, 20, 50, 1, 2, 3, 18\rangle$

b) Um die Max-Heap-Eigenschaft zu erhalten, rufen wir die Prozedur MAX-HEAPIFY auf. Seine Eingaben sind ein Array A und ein Index i in dem Array. Wenn es aufgerufen wird, nimmt MAX-HEAPIFY an, dass die Binärbäume $LEFT(i)$ und $RIGHT(i)$ Max-Heaps sind, aber das $A[i]$ die Max-Heap-Eigenschaft verletzen könnte. MAX-HEAPIFY positioniert den Wert bei $A[i]$ solange neu im Heap bis der Teilbaum, der in Index i verwurzelt ist, die Max-Heap-Eigenschaft erfüllt.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Abbildung 2: Pseudo-Code für MAX-HEAPIFY

MAX-HEAPIFY$(A, 3)$ wird auf dem Array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$ aufrufen. Zeigen Sie die Zwischenzustände des Arrays, die während dem Ablauf des Aufrufes vorkommen sowie den Endzustand nach Ende der Prozedur.

c) Zeigen Sie, dass die Worst-Case-Laufzeit von MAX-HEAPIFY auf einem Heap der Größe n in $\Omega(\log(n))$ liegt.
Hinweis: Geben Sie für einen Heap mit n Knoten Knotenwerte an, sodass MAX-HEAPIFY rekursiv auf jedem Knoten in einem einfachen Pfad von der Wurzel zu einem Blatt aufgerufen wird.

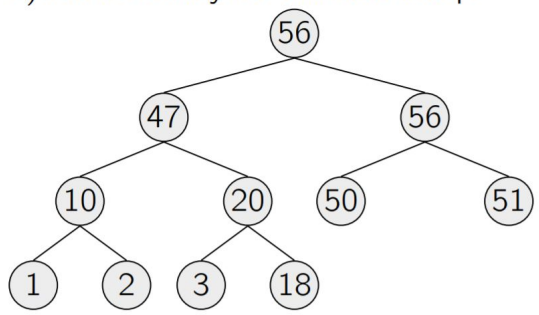
Lösung.

a) Siehe Abbildung 3. Die Heapeigenschaft der Max-Heaps beschreibt die Bedingung, dass der Wert eines Knotens höchstens der Wert seines Elternknotens ist. Damit gilt, dass der größte Wert in einem Max-Heap sich an der Wurzel befindet.

b) Siehe Abbildung 4.

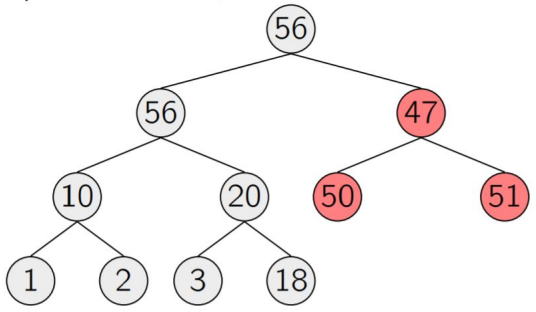
c) Betrachten Sie den Heap A mit $A[1] = 1$ und $A[i] = 2$ für $2 \leq i \leq n$. Da 1 das kleinste Element des Heaps ist, muss es durch jede Ebene des Heaps getauscht werden, bis es ein Blattknoten ist. Da der Heap die Höhe $\lfloor \log(n) \rfloor$ hat, läuft MAX-HEAPIFY in der Worst-Case-Laufzeit $\Omega(\log(n))$.

1) Dieses Array ist ein Max-Heap.



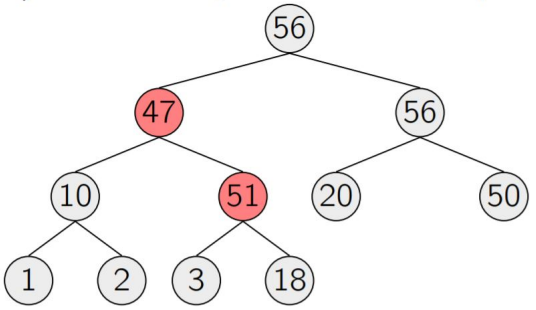
56	47	56	10	20	50	51	1	2	3	18

2) Dieses Array ist kein Max-Heap.



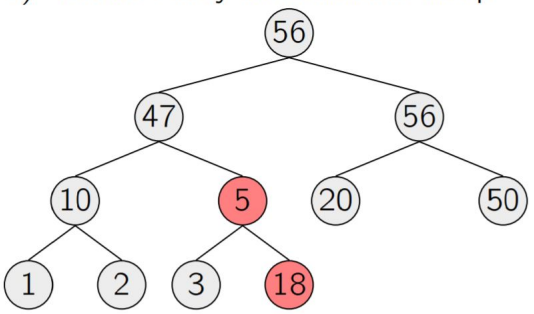
56	56	47	10	20	50	51	1	2	3	18

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

Abbildung 3: Max-Heaps

27	17	3	16	13	10	1	5	7	12	4	8	9	0
27	17	10	16	13	3	1	5	7	12	4	8	9	0
27	17	10	16	13	9	1	5	7	12	4	8	3	0

Abbildung 4: Zustände des Arrays A

H1 Quicksort

Zeigen Sie mit Hilfe des Mastertheorems, dass die Laufzeit von QUICKSORT im besten Fall in $\Omega(n \log n)$ $(n \geq 2)$ liegt. Gehen Sie dabei wie folgt vor.

a) Ergänzen Sie folgende Rekursionsgleichung für die Best-Case-Laufzeit von QUICKSORT.

$$T(n) = \min_{1 \leq q \leq n-1} T(\ldots) + T(\ldots) + \Theta(\ldots)$$

b) Beweisen Sie, dass die Funktion $f(q) = q \log q + (n-q-1)\log(n-q-1)$ ihr Minimum im Intervall $[1, n-1]$ für $q = \frac{n-1}{2}$ annimmt.

c) Beweisen Sie mit Hilfe des Mastertheorems, dass für die Laufzeit von QUICKSORT im besten Fall $T(n) = \Omega(n \log n)$ gilt. Setzen Sie dazu das in b) berechnete Minimum in die Rekursionsgleichung aus a) ein.

Lösung.

a) $T(n) = \min_{1 \le q \le n-1} T(q) + T(n-q-1) + \Theta(n)$.

b) Um zu zeigen, dass die Funktion $f(q) = q \log q + (n-q-1)\log(n-q-1)$ ihr Minimum im Intervall $1 \le q \le n-1$ für $q = (n-1)/2$ annimmt, setzen wir die 1. Ableitung der Funktion $f(q)$ gleich 0.

$$\begin{aligned} & 0 = \frac{\ln q - \ln(n-q-1)}{\ln 2} = f'(q) \\ \Leftrightarrow \quad & 0 = \ln q - \ln(n-q-1) \\ \Leftrightarrow \quad & 1 = \frac{q}{n-q-1} \\ \Leftrightarrow \quad & q = n-q-1 \\ \Leftrightarrow \quad & q = \frac{n-1}{2} \end{aligned}$$

Durch Einsetzen in die 2. Ableitung $f''(q) = 1/(\ln 2)\,(1/q + 1/(n-q-1))$ verifizieren wir, dass $(n-1)/2$ tatsächlich ein Minimum der Funktion $f(q)$ ist.

$$\begin{aligned} f''\left(\frac{n-1}{2}\right) &= \frac{1}{\ln 2}\left(\frac{2}{n-1} + \frac{2}{n-1}\right) \\ &= \frac{1}{\ln 2} \cdot \frac{4}{n-1} \\ &> 0 \quad (\text{da } n \ge 2) \end{aligned}$$

c) Die Rekursionsgleichung lautet $T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n)$. Diese Gleichung unterscheidet sich nur um $O(1)$ von der Gleichung $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ (siehe unten).
Mit $a = b = 2$ und $f(n) = \Theta(n)$ erhalten wir $f(n) = \Theta(n^{\log_b a})$ und somit nach Teil 2 des Mastertheorems $T(n) = \Theta(n \log n)$. Insbesondere gilt also $T(n) = \Omega(n \log n)$.

Wir zeigen, dass die Rekursionsgleichungen $T(n) = 2T(\frac{n-1}{2}) + \Theta(n)$ und $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ der gleichen Komplexitätsklasse angehören.
Seien g, h Funktionen mit $g(n) = \lfloor \frac{n-1}{2} \rfloor$ und $h(n) = \lfloor \frac{n}{2} \rfloor$[1], dann folgt mit der Eigenschaft, dass diese Funktionen monoton fallend sind $\forall n_1, n_2 \in \mathbb{N}\, \forall f \in \{g, h\} : n_1 < n_2 \Rightarrow f(n_1) \le f(n_2)$.

Wenn nun diese Funktionen wie in einer Rekursion wiederholt angewendet werden, erreichen sie schließlich den Rekursionsanker (im Falle von Quicksort 1-elementige Listen). Sei $m_{\{g,h\}}(n)$ die Funktion, die angibt wie oft die jeweilige Funktion (g bzw. h) auf n angewendet werden muss, um 1 zu erreichen. Seien nun a und b Folgen innerhalb der natürlichen Zahlen mit:

$$a_0 := 1 \text{ und } a_{n+1} := 2a_n + 1$$

$$b_0 := 1 \text{ und } b_{n+1} := 2b_n + 2$$

sodass die Folge a_n den maximalen Wert k beinhaltet, bei dem $m_h(k) = n$ gilt. b_n entspricht dem Äquivalent für m_g. Dieses Verhalten lässt sich durch Induktion zeigen, z.B. für a_n:
Für $n = 0$ ist $a_0 = 1$ sodass der Induktionsanfang gilt. Für den Induktionsschritt nehme man an a_{n+1} sei nicht der maximale Wert mit $m_h(a_{n+1}) = n+1$, sodass $m_h(a_{n+1}+1) = n+1$ gelte. Dann folgt aber $h(a_{n+1}+1) = h(2a_n+2) = a_n+1$ für welches laut Induktionsvoraussetzung $m_h(a_n + 1) = n + 1$ gilt, und mit dieser weiteren Anwendung von h einen Widerspruch zur Annahme liefert. Analog lässt sich auch für b_n argumentieren.

Dann folgen für a und b die expliziten Darstellungen:

$$a_n = \sum_{i=0}^{n} 2^i$$

$$b_n = 2^n + \sum_{i=1}^{n} 2^i$$

[1] Für eine Erläuterung zur Anwendung mit floor innerhalb der Rekurrenz siehe "Introduction to Algorithms, Chapter 4.6.2".

Mit $b_n - a_n = 2^n - 1 < 2^{n+1} - 1 = a_{n+1}$ folgt $a_n \leq b_n < a_{n+1}$ und zusammen mit der oben beschriebenen Monotonie das Resultat, dass die Rekursionstiefe der beiden Funktionen sich maximal um 1 unterschiedet, demnach (und mit einem ähnlichen Argument für die Länge der betrachteten Teillisten) also die Komplexität der Funktionen sich nur durch vernachlässigbare konstante Werte unterscheidet.

H2 Heapsort

Der Heapsort-Algorithmus bildet zunächst mithilfe von BUILD-MAX-HEAP einen Max-Heap aus dem Eingabearray $A[1, ..., n]$, wobei $n = A.length$ ist. Da das maximale Element des Arrays an der Wurzel $A[1]$ gespeichert ist, können wir es in seine korrekte endgültige Position bringen, indem wir es mit A[n] tauschen. Wenn wir jetzt den n-ten Knoten aus dem Heap verwerfen - indem wir einfach $A.heapsize$ dekrementieren - beobachten wir, dass die Kinder von der Wurzel Max-Heaps bleiben, aber das neue Wurzelelement die Max-Heap-Eigenschaft verletzen könnte. Alles was wir tun müssen, um die Max-Heap-Eigenschaft wiederherzustellen, ist MAX-HEAPIFY auf der Wurzel aufzurufen. Der Heapsort Algorithmus wiederholt dann diesen Prozess für den Max-Heap der Größe $n-1$ bis zu der Größe 2.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

a) Beweisen Sie die Korrektheit von HEAPSORT mit der folgenden Schleifeninvariante:

   Zu Beginn jeder Iteration der for-Schleife (Zeilen 2-5) ist das Teilarray $A[1, ..., i]$ ein Max-Heap, der die kleinsten Elemente von $A[1, ..., n]$ enthält, und das Teilarray $A[i + 1, ..., n]$ enthält das $n - i$ größten Elemente von $A[1, ..., n]$ aufsteigend sortiert.

b) Zeigen Sie, dass die Worst-Case-Laufzeit von HEAPSORT $\Omega(n \log(n))$ ist.
   Hinweis: Betrachten Sie den Fall, dass das Eingabe Array in absteigender Reihenfolge sortiert ist.

Lösung.

a) Wir werden die Schleifeninvariante von HEAPSORT durch vollständige Induktion über i beweisen:

   Induktionsanfang: Zu Beginn der ersten Iteration der for-Schleife haben wir $i = A.length$. Das Teilarray $A[1, ..., n]$ ist ein Max-Heap, da auf diesen vorher BUILD-MAX-HEAP(A) aufgerufen wurde. Es enthält die n kleinsten Elemente und das leere Teilarray $A[n + 1, ..., n]$ enthält trivialerweise die 0 größten Elemente von A in sortierter Reihenfolge.

   Induktionsbehauptung: Zu Beginn jeder i-ten Iteration der for-Schleife ist das Teilarray $A[1, ..., n]$ ein Max-Heap, der die kleinsten Elemente von $A[1, ..., n]$ enthält, und das Teilarray $A[i + 1, ..., n]$ enthält das $n - i$ größten Elemente von $A[1, ..., n]$ aufsetigend sortiert.

   Induktionsschritt $i -> i - 1$: Angenommen, zu Beginn der Iteration i der for-Schleife ist das Teilarray $A[1, ... ,i]$ ein Max-Heap, der die kleinsten Elemente von $A[1, ... ,n]$ enthält und das Teilarray $A[i + 1, ..., n]$ enthält die größten Elemente von $A[1, ..., n]$ in sortierte Reihenfolge.

   Da $A[1, ..., i]$ ein Max-Heap ist, ist $A[1]$ das größte Element in $A[1, ..., i]$. Somit ist es das $(n - i + 1)$-größte Element aus dem ursprünglichen Array, da von den $n - i$ größten Elementen angenommen wird, dass sie sich am Ende des Arrays befinden. In Zeile 3 wird $A[1]$ mit $A[i]$ getauscht, sodass $A[i, ..., n]$ die $n - i + 1$ größten Elemente des Arrays enthält und $A[1, ..., i - 1]$ enthält die kleinsten $i - 1$ Elemente.
   Da $A[1, ..., i]$ vor der Iteration ein Max-Heap war und nur die Elemente $A[1]$ und $A[i]$ getauscht wurden sind die

linken und rechten Teilbäume von Knoten 1, bis Knoten $i-1$ immernoch Max-Heaps.
Deshalb kann schließlich MAX-HEAPIFY$(A, 1)$ aufgerufenwerden, sodass das Element, das sich jetzt an Knoten 1 befindet, in die richtige Position gebracht wird und die Max-Heap Eigenschaft für die nächste Iteration auf $A[1, ..., i-1]$ wieder hergestellt ist.

Korrektheit des Endergebnisses: Nach der letzten Iteration gilt nach der Schleifeninvariante, dass das Teilarray $A[2, ..., n]$ die $n-1$-größten Elemente von $A[1, ..., n]$ sortiert enthält. Da $A[1]$ wegen der nach wie vor geltenden Max-Heap-Eigenschaft das n-größte Element sein muss, ist das gesamte Array wie gewünscht sortiert.

b) BUILD-MAX-HEAP liegt in $\Theta(n)$ und ist hierfür also vernachlässigbar. Nehmen wir an, dass HEAPSORT auf einem Array aufgerufen wird, welches in absteigender Reihenfolge sortiert ist und $2^h - 1$ Elemente beinhaltet (volle Blattebene). Man betrachte die Blätter ($\lceil n/2 \rceil$ Elemente) im rechten Teilbaum ($\lceil n/4 \rceil$ Elementen) der Wurzel. Jedes Mal, wenn hier A[1] mit A[i] getauscht wird, enthält der Root-Knoten danach einen der n/4 kleinsten Werte im Heap (aufgrund der Sortierung), da die einzige Möglichkeit wie sich hier ein Eintrag verändert haben könnte, nur durch noch kleinere Werte ist. Diese müssen beim Wiederherstellen der Heap-Eigenschaft, jedoch stets den gesamten Baum, bis zu einem Blatt durchlaufen ($\geq \log(n) - 1$), sodass zusammen $\lceil n/4 \rceil \cdot (\log(n) - 1) \in \Omega(n \log n)$ folgt.