Binärer Heap und Heapsort[*]

Hans Jürgen Ohlbach

22. März 2018

Keywords: Heaps, Heapsort

Empfohlene Vorkenntnisse: Bäume, siehe Baeume.pdf

Inhaltsverzeichnis

1 Was ist ein Binärer Heap? 2

2 Tiefe des Heap-Baumes 2

3 Heapify 3

4 Speicherung im Array 4

5 Einfügen und Löschen 5

6 Anwendungen von Heaps 6

6.1 Prioritätsschlangen (engl. Priority Queues) 6

6.2 Heapsort 7

6.3 Eigenschaften von Heapsort 11

[*]Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

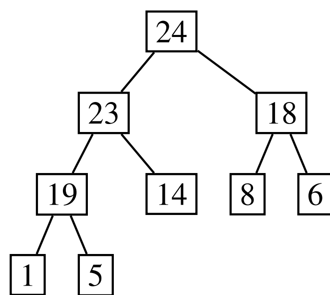
1 Was ist ein Binärer Heap?

Ein *Binärer Heap* ist ein *Binärbaum*, dessen Knoten Daten mit einer totalen Ordnung tragen, und die der ***Heap-Bedingung*** genügen:

- Der Wert eines Knotens ist größer als der Wert seiner Kindknoten.
  Die Kindknoten können dabei in einer beliebigen Reihenfolge sein.
- Alle Schichten bis zur letzten Schicht sind aufgefüllt.
  Die letzte Schicht muss linksbündig aufgefüllt sein.
  Das garantiert, dass der Baum *balanciert* ist.

Typische solche Daten mit totaler Ordnung sind die Zahlen mit der $\leq$- oder $<$-Ordnung.

Als Beispiel zeigen wir einen Binären Heap für die Zahlen 23,1,6,19,14,18,8,24,5.
Der Baum sieht dann so aus:



2 Tiefe des Heap-Baumes

Da der Baum ein balancierter Binärbaum ist, verdoppelt sich in jeder Schicht die Anzahl der Kindknoten. Hat man n Blattknoten, dann kann es höchstens $\lceil \log_2(n) \rceil$ Schichten geben[1]. Daher ist die Größe der Baumtiefe $\mathcal{O}(\log(n))$.

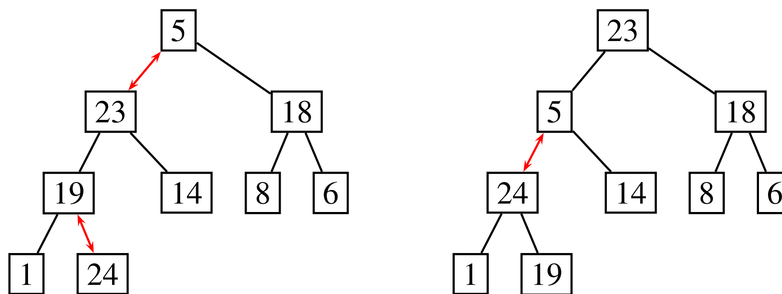
[1] Für eine Zahl x ist $\lceil x \rceil$ die kleinste ganze Zahl $\geq x$, und $\lfloor x \rfloor$ die größte ganze Zahl $\leq x$.

3 Heapify

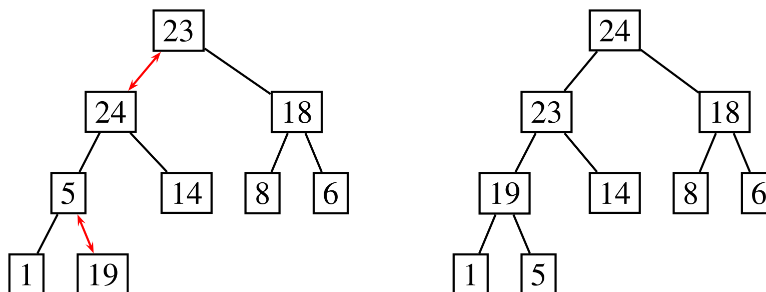
Die wichtigste Operation ist das sog. *Heapify*. Das bedeutet die Herstellung der Heap-Bedingung, wenn der Baum ihr nicht genügt. Das ist dann der Fall, wenn ein Knoten im Baum einen Elternknoten hat, der kleiner ist.

Mit einer einfachen Operation kann man dieses Problem beheben:
Man vertauscht Kindknoten mit Elternknoten.
Falls beide Kindknoten größer sind als der Elternknoten, dann vertauscht man den größeren Kindknoten mit dem Elternknoten.

In folgendem Baum stehen 5 und 24 an der falschen Position. Wir vertauschen zunächst 5 mit 23 und 19 mit 24:



Im rechten Baum hat man noch mehrere Vertauschungsnotwendigkeiten. Die Kinder des Knotens 5 sind beide größer als 5. In diesem Fall vertauschen wir den größeren Kindknoten mit den Elternknoten. Also vertauschen wir zunächst 5 und 24. Dann bleibt noch, 23 mit 24 und 19 mit 5 zu vertauschen.



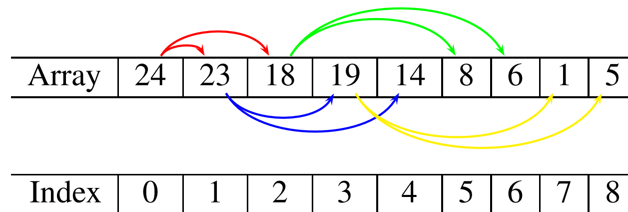
Jetzt die Heap-Bedingung für den Baum wieder hergestellt.

Komplexität von Heapify: Da die maximale Baumtiefe $\mathcal{O}(\log(n))$ ist, ist der Aufwand für das Vertauschen ebenfalls $\mathcal{O}(\log(n))$.

4 Speicherung im Array

Ein binärer Baum kann linear in einem Array gespeichert werden, indem man die Schichten nacheinander abspeichert.

Für den obige Baum sieht das so aus:



Wenn das Array ab Index 0 gezählt wird, sind die Indices für die Kindknoten eines Knotens bei Arrayposition k: $2k+1$ und $2k+2$. Für das obige Beispiel bedeutet das:

Arrayindex	Arrayinhalt	Index der Kindknoten	Kindknoten
0	24	1,2	23,18
1	23	3,4	19,14
2	18	5,6	8,6
3	19	7,8	1,5

Von dem Arrayindex i eines Kindknotens kann man den Index des Elternknotens berechnen:

$$\mathit{Elternknotenindex}(i) = \lfloor (i-1)/2 \rfloor$$

Im Beispiel ergibt sich dabei:

Index des Kindknotens	0	1	2	3	4	5	6	7	8
Index des Elternknotens	0	0	0	1	1	2	2	3	3

Für die meisten Betrachtungen über Heaps ist die Baumdarstellung jedoch instruktiver. Alle Manipulationen der Bäume kann man jedoch direkt in Manipulationen auf den Arrays übersetzen.

Konsequenz: Implementiert man Heapify auf einem Array, dann kann man mit $\mathcal{O}(\log(n))$ Aufwand das größte Element an die erste Position im Array bringen. Allerdings lohnt sich die Speicherung im Array nur dann wenn man die maximal benötigte Größe des Arrays bei der Erzeugung des Arrays abschätzen kann.

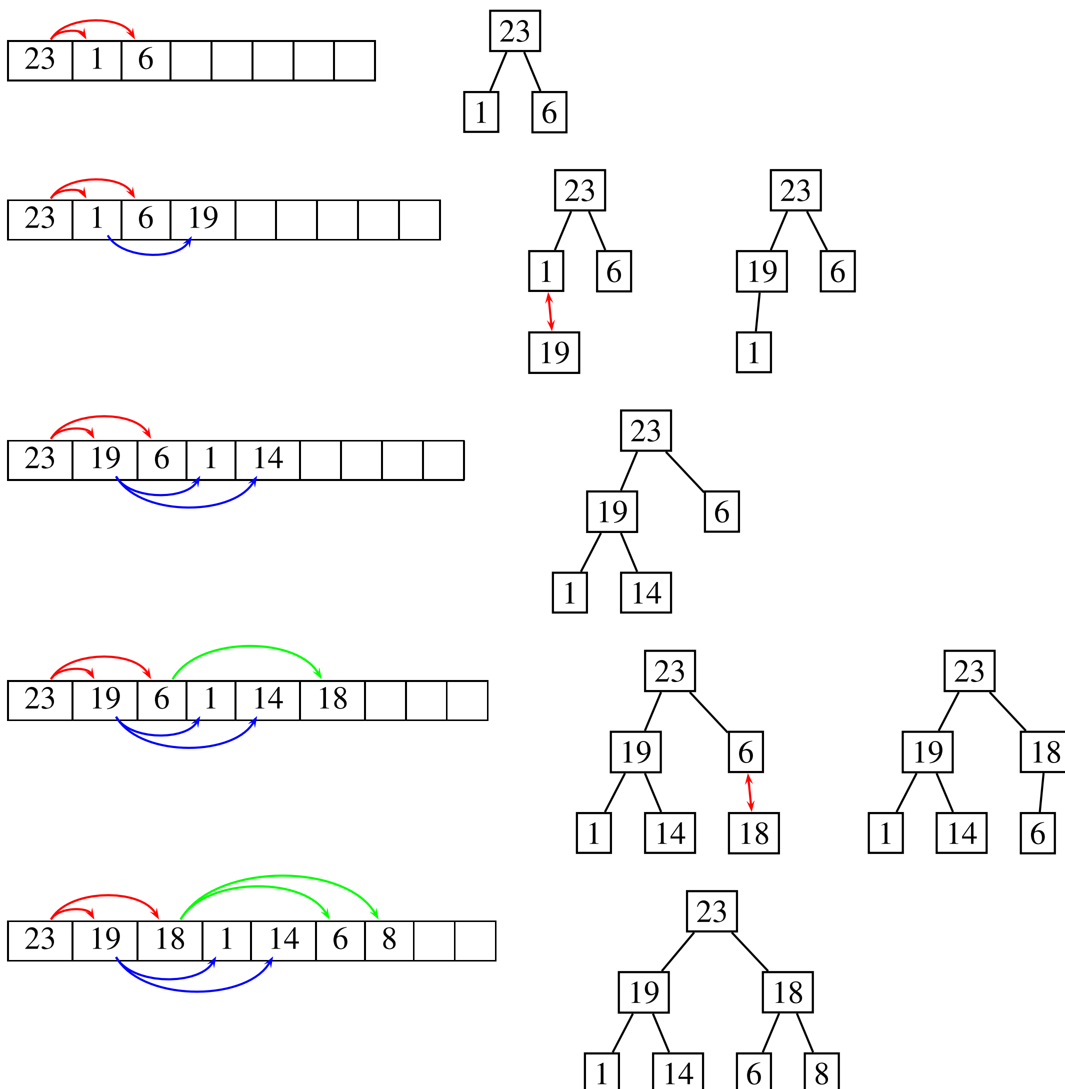
5 Einfügen und Löschen

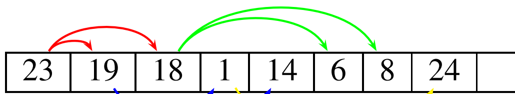
Speichert man den Heap im Array, dann kann ein neues Element *eingefügt* werden, indem man es ans Ende des Arrays setzt, und dann die Heapify-Funktion aufruft.

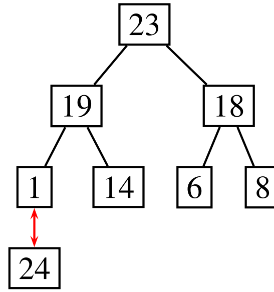
Mit dieser Methode kann man ein Heap aufbauen, indem man die Elemente nacheinander einfügt.

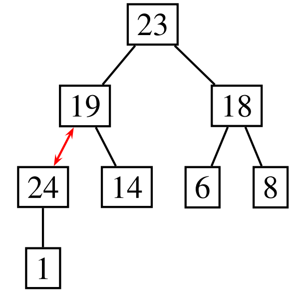
Um ein Element aus der Mitte zu löschen, überschreibt man es mit dem letzten Element im Array, und ruft wiederum die Heapify-Funktion auf.

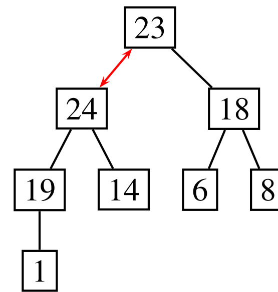
Der Aufbau eines Heaps mittels eines Arrays sorgt automatisch dafür, dass das Heap balanciert ist. Das liegt daran, dass im Array das Heap ebenenweise gespeichert wird. Hängt man ein neues Element ans Ende des Arrays, dann wird die aktuelle Ebene aufgefüllt. Wir illustrieren das an dem Beispiel von oben: 23,1,6,19,14,18,8,24,15. Die ersten drei Zahlen sind vorerst in der richtigen Reihenfolge.

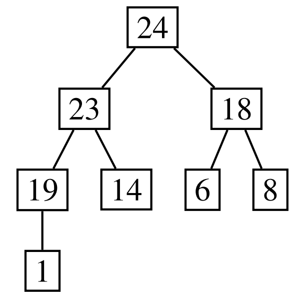


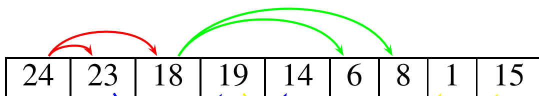


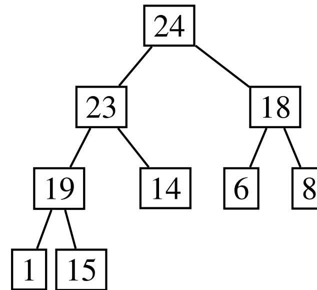












6 Anwendungen von Heaps

Die Eigenschaft, das größte Element sehr effizient nach oben bzw. vorne zu bringen, wird insbesondere bei Prioritätsschlangen und Heapsort ausgenutzt.

6.1 Prioritätsschlangen (engl. Priority Queues)

Eine wichtige Anwendung der Heap-Datenstruktur sind Prioritätsschlangen, bei denen man jeweils nur das größte Element abgreifen will, und die Sortierung der anderen Elemente unwichtig ist.

Prioritätsschlangen sind Arrays, mit den zwei wichtigsten Operationen:
- **Einfügen** eines neuen Elements
- **Abfragen** und **extrahieren** des aktuell größten Elements.

Da man bei der Abfrage nur an dem größten Element interessiert ist, und die Sortierung der restlichen Elemente egal ist, eignen sich binäre Heaps in Array-Implementierung am besten. Die Einfüge-Operation mit der Herstellung der Heap-Bedingung ist mit $\mathcal{O}(\log(n))$ Aufwand möglich. Danach ist das größte Element am Anfang des Arrays. Die Abfrage des größten Element geht dann mit konstantem Aufwand. Entfernt man das größte Element, dann muss man die Heap-Bedingung wieder herstellen, was wieder mit $\mathcal{O}(\log(n))$ Aufwand möglich ist.

6.2 Heapsort

Heapsort ist eine optimierte Variante von *Selectionsort*.
Für eine Array mit Elementen $a_0 \ldots a_n$ funktioniert Selectionsort folgendermaßen:

- Vertausche das größte Element mit a_n
- Vertausche das zweitgrößte Element mit a_{n-1},
- usw.

Man muss also im Teilarray $a_0 \ldots a_k$ das größte Element suchen, und mit a_k vertauschen.

Das größte Element in einer Liste zu suchen ist genau die Stärke des binären Heaps. Allerdings muss zu Beginn die Heap-Bedingung hergestellt werden.

Heapsort funktioniert auf einem Array daher folgendermaßen:

- Stelle die Heap-Bedingung her. Das macht man, indem man nacheinander die Elemente hinzufügt und die Heapify-Operation ausführt. Jetzt ist das größte Element am Anfang des Arrays.
- Vertausche das erste Element mit dem letzten im Array.
- Stelle die Heapify-Bedingung für die Elemente $a_0 \ldots a_{n-1}$ her.
  Jetzt ist das zweitgrößte Element am Anfang des Arrays.
- etc.

Da der Aufwand, die Heapify-Bedingung wieder herzustellen jedesmal $\mathcal{O}(\log(n))$ ist, ergibt sich ein Gesamtaufwand für die Sortierung der n Elemente, von $\mathcal{O}(n \log(n))$.

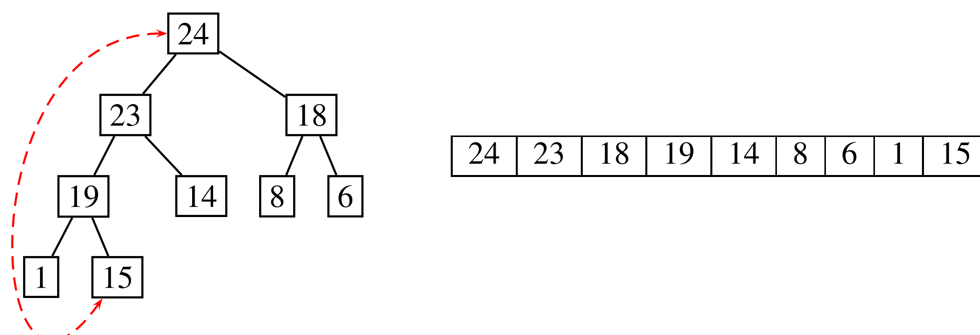
Wir illustrieren das Verfahren mit folgendem Beispiel:

23	1	6	19	14	18	8	24	15

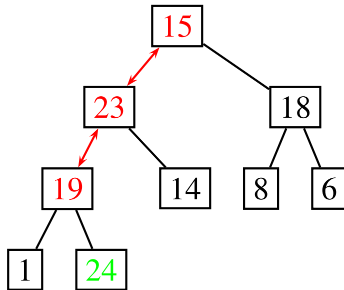
Im allerersten Schritt werden die Elemente in ein Heap umgebaut. Das geschieht, indem man nacheinander die Elemente hinzunimmt und immer die Heapify-Operation ausführt. Das geht mit $\mathcal{O}(n \log(n))$ Aufwand.

Links ist immer der Heap als Baum dargestellt, und rechts die tarsächliche Repräsentation im Array.

Das Ergebnis von Heapify mit dem größten Element 24 an der Spitze ist dann:

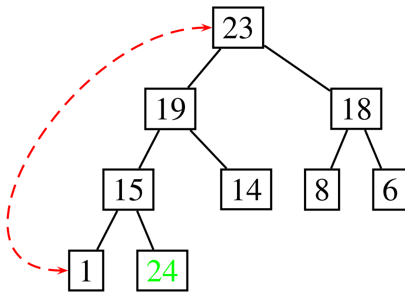


Die gestrichelte Linie deutet an, dass jetzt das erste (größte) Element mit dem letzten Element vertauscht werden muss. Die Zahl 24 ist dann schon an der richtigen Position im Array. Die restlichen Operationen betreffen jetzt nur noch das Array vor der 24. Da jetzt die Heap-Bedingung verletzt wird, muss die Heapify-Operation wieder aufgerufen werden. Die roten Pfeile deuten an, welche Elemente vertauscht werden müssen.



15	23	18	19	14	8	6	1	24

Die zweitgrößte Zahl ist jetzt vorne und muss mit dem zweitletzten Element im Array vertauscht werden.



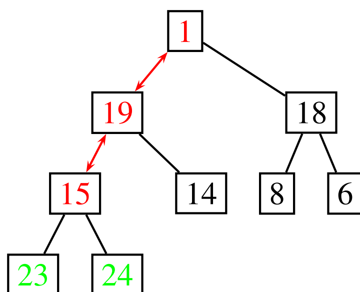
23	19	18	15	14	8	6	1	24

Die 23 ist jetzt an der richtigen Position.

Die folgenden Operationen sind Wiederholungen von
- Heapify
- Vertauschen des ersten Elements mit dem letzten des unsortierten Arrayteils.

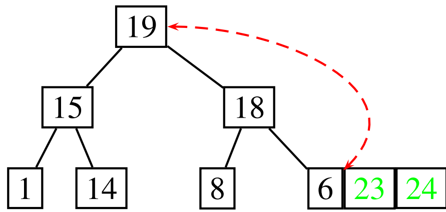
Die schon fertig einsortieren Elemente gehören eigentlich nicht mehr zum Baum. Zur Illustration werden sie noch mitgeführt, aber grün gekennzeichnet.

Heapify:



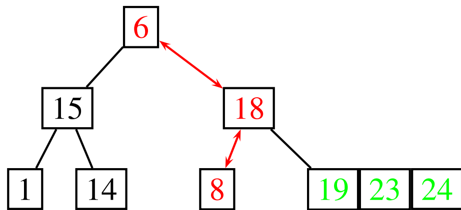
1	19	18	15	14	8	6	23	24

Vertauschen:



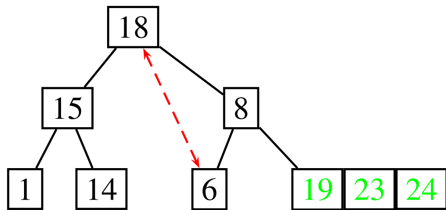
19	15	18	1	14	8	6	23	24

Heapify:



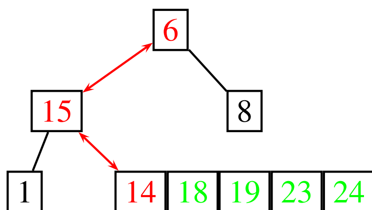
6	15	18	1	14	8	19	23	24

Vertauschen:



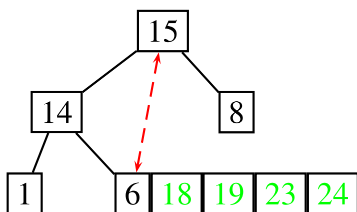
18	15	8	1	14	6	19	23	24

Heapify:



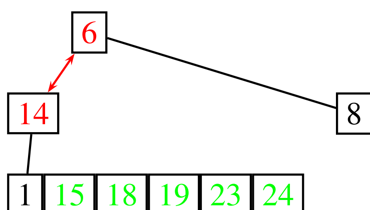
6	15	8	1	14	18	19	23	24

Vertauschen:



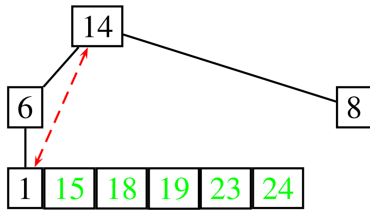
15	14	8	1	6	18	19	23	24

Heapify:



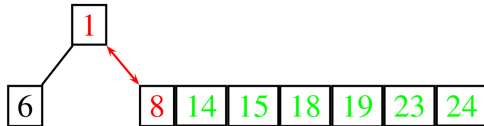
6	14	8	1	15	18	19	23	24

Vertauschen:



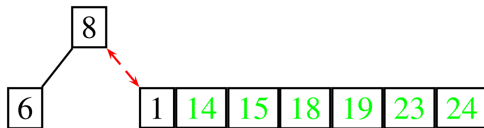
14	6	8	1	15	18	19	23	24

Heapify:



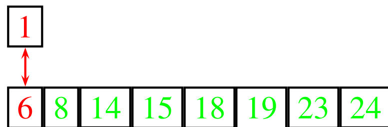
1	6	8	14	15	18	19	23	24

Vertauschen:



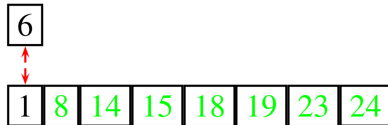
1	6	8	14	15	18	19	23	24

Heapify:



1	6	8	14	15	18	19	23	24

Vertauschen:



1	6	8	14	15	18	19	23	24

Ende:

1	6	8	14	15	18	19	23	24

6.3 Eigenschaften von Heapsort

Speicherstruktur: Die Heapify-Operation ist effizient nur auf Arrays implementierbar. Daher arbeitet Heapsort effizient nur auf Arrays. Allerdings genügt das Ausgangsarray. Heapsort ist daher in-place.

Stabilität: durch die Heapify-Operation werden die Elemente unkontrollierbar vertauscht, HeapSort ist daher *nicht stabil*.

Zeitkomplexität:
Best Case: Wenn die Daten umgekehrt sortiert sind, dann ist bei der Heapify-Operation keine Vertauschung notwendig. In diesem Fall ist die Komplexität linear: $\mathcal{O}(n)$.

Worst Case: dafür ist der Aufwand, wie oben dargelegt $\mathcal{O}(n \log(n))$. Damit gehört Heapsort zu den schnellsten Sortierverfahren.

Ein **Average Case:** ist schwer zu quantifizieren, da die Anzahl der Vertauschungen bei der Heapify-Operation sehr von den Daten abhängt.

Parallelisierung: Die nächste Heapify-Operation ergibt sich erst, wenn die letzte Heapify-Operation fertig ist. Daher müssen sie sequentiell durchgeführt werden. Die Vertauschungen zu parallelisieren wäre im Prinzip möglich, erfordert aber höchstwahrscheinlich mehr Synchronisationsaufwand, als es Geschwindigkeit bringt.