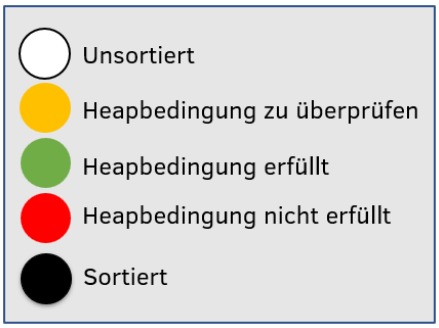
HEAPSORT



GymInf Fachdidaktik II – Frühlingssemester 2023

Studienleistung in der Sequenz von Prof. Dr. Juraj Hromkovič und Regula Lacher

juraj.hromkovic@inf.ethz.ch, regula.lacher@inf.ethz.ch

Eingereicht am 5.6.2023 von:

Patrick Suter

GymInf-Student & Gymnasiallehrperson an der Kantonsschule Zug

patrick.suter@unifr.ch, patrick.suter@ksz.ch

Inhaltsverzeichnis

1. Einleitung

In der Informatik gibt es viele effiziente Sortieralgorithmen, die es ermöglichen, grosse Datenmengen in kurzer Zeit zu sortieren. Einer dieser Algorithmen ist der Heapsort. Heapsort ist ein Sortieralgorithmus, der auf dem Konzept des Binärbaums und des Heaps basiert. Er ist besonders effektiv, wenn es darum geht, grosse Datenmengen zu sortieren, und ist in vielen Anwendungsbereichen weit verbreitet.

In dieser Unterrichtssequenz werden wir uns mit dem Heapsort-Algorithmus auseinandersetzen und lernen, wie er funktioniert. Wir werden uns zunächst mit dem Konzept von Binärbäumen und Heaps als eine Datenstruktur vertraut machen und dann die Schritte des Heapsort-Algorithmus behandeln. Am Ende werden die Schüler:innen in der Lage sein, den Heapsort-Algorithmus anzuwenden und zu verstehen, wie er funktioniert.

Vorwissen aus Informatik und Mathematik

- Die Schüler:innen sind mit grundlegenden Konzepten der Programmierung vertraut, einschliesslich Variablen, Schleifen und Bedingungen.
- Die Schüler:innen können Struktogramme lesen und interpretieren.
- Die Schüler:innen verfügen über grundlegende Kenntnisse der Datenstrukturen.
- Die Schüler:innen verstehen die Algorithmen der linearen und binären Suche sowie den Insertion- und Quicksort.
- Die Schüler:innen sind mit dem Konzept des binären Baums bekannt. Somit können sie einfache binäre Bäume erstellen und traversieren.

Zielsetzungen

- Die Schüler:innen sollen in der Lage sein, den Unterschied zwischen einem Binärbaum und einem Heap zu erklären und einfache binäre Bäume und Heaps zu erstellen und zu manipulieren.
- Die Schüler:innen sollen verstehen, wie der Heapsort-Algorithmus mit seinen Bestandteilen funktioniert.
- Die Schüler:innen sollen die Schritte des Heapsort-Algorithmus verstehen und den Algorithmus selbstständig durchführen.
- Die Schüler:innen sollen die Laufzeitkomplexitäten von Heapsort analysieren können.

Verwendete Symbole


Kontext / Theorie (in normaler Schrift)


Aufgabe (in kursiver Schrift)


Tipps (in blauer Schrift)


Lösungen (in roter Schrift)

2. Unterrichtssequenz

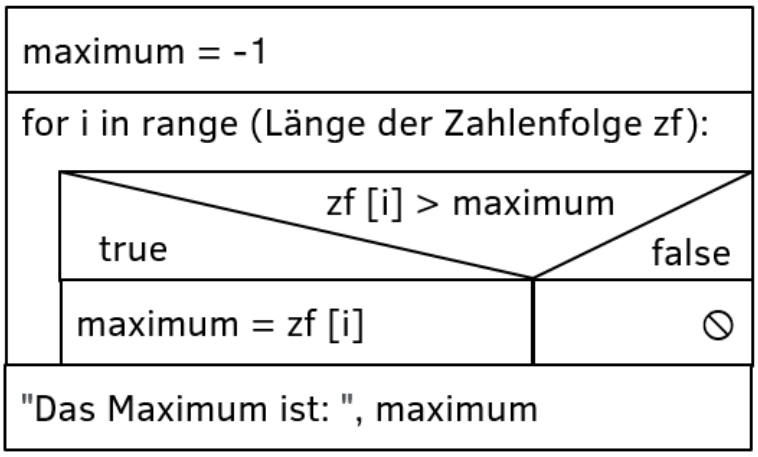
Maximum in Zahlenfolge finden

Als Einstieg möchten wir einen einfachen Algorithmus anschauen, der das Maximum in einer Zahlenfolge sucht und ausgibt. Das untenstehende Struktogramm stellt einen Algorithmus dar, der das Maximum einer Folge ganzzahliger positiver Zahlen sucht und ausgibt.

Studieren Sie den untenstehenden Algorithmus und beantworten Sie folgende Fragen.

Handelt es sich beim Algorithmus um eine lineare oder binäre Suche?	
Von welcher Problemgrösse hängt die Laufzeit ab?	
Welche asymptotische Laufzeitklasse in O-Notation hat dieser Algorithmus?	

Algorithmus zur Suche des Maximums einer Zahlenfolge



Handelt es sich beim Algorithmus um eine lineare oder binäre Suche?	*Es ist eine lineare Suche, da die Liste einmal ganz durchlaufen wird, also nicht wie bei der binären Suche unterteilt wird.*
Von welcher Problemgrösse hängt die Laufzeit ab?	*Sie ist abhängig von der Länge der Zahlenfolge.*
Welche asymptotische Laufzeitklasse in O-Notation hat dieser Algorithmus?	*Es ist O (n), da die gesamte Zahlenfolge einmal gemäss der Schleife durchlaufen wird.*

Zahlenfolge sortieren

In der letzten Aufgabe haben wir einen Algorithmus gesehen, der aus einer Zahlenfolge das Maximum findet.

Nun möchten wir den Algorithmus der letzten Aufgabe weiterentwickeln. Wir schauen uns einen Algorithmus an, der eine Zahlenfolge vom kleinsten zum grössten Element sortiert. Die Zahlenfolge zf ist ein Array. Das jeweils zu sortierende Element befindet sich in der Variable «elementZuSortieren». Der rote Pfeil («roterPfeil») wird fortlaufend auf das nächste Element der Zahlenfolge gesetzt. Der blaue Pfeil zeigt jeweils auf jenes Element links von dem zu sortierenden Element, mit dem das zu sortierende Element verglichen wird. Ein Tausch von benachbarten Elementen geschieht dann, wenn das linke Element grösser ist als das direkt neben ihm liegende rechte Element.

Algorithmus zum Sortieren einer Zahlenfolge

```
zf []
int elementZuSortieren
int roterPfeil, int blauerPfeil
for roterPfeil in range (Länge der Zahlenfolge zf):
    elementZuSortieren = zf [roterPfeil]
    blauerPfeil = roterPfeil
    while (blauerPfeil > 0 und
    elementZuSortieren < zf [blauerPfeil] )
        tausche Elemente aus
        blauerPfeil = blauerPfeil - 1
```

Studieren Sie das obenstehende Struktogramm und beantworten Sie folgende Fragen.

Handelt es sich beim Algorithmus um den Insertion- oder Quicksort?	
Was macht dieser Algorithmus? Erklären Sie die Schritte in eigenen Worten.	
Von welcher Problemgrösse hängt die Laufzeit ab?	
Welche Laufzeitklasse in O-Notation hat dieser Algorithmus im (eher seltenen) Best Case?	
Welche asymptotische Laufzeitklasse in O-Notation hat dieser Algorithmus im Worst Case?	

Handelt es sich beim Algorithmus um den Insertion- oder Quicksort?	*Es handelt sich um Insertionsort.*
Was macht dieser Algorithmus? Erklären Sie die Schritte in eigenen Worten.	*1. Setze den roten Pfeil fortlaufend auf jedes Element.* *2. Setze den blauen Pfeil auf den roten Pfeil und schiebe ihn schrittweise nach links.* *3. Wenn das Element beim blauen Pfeil grösser ist als das Element rechts, neben dem blauen Pfeil, dann tausche die beiden Elemente aus.*
Von welcher Problemgrösse hängt die Laufzeit ab?	*Sie hängt von der Länge der Zahlenfolge ab.*
Welche Laufzeitklasse in O-Notation hat dieser Algorithmus im (eher seltenen) Best Case?	*Der Best Case ist eine bereits sortierte Zahlenfolge. Wenn die Zahlenfolge bereits sortiert ist, finden keine Vertauschungen statt. Die Zeilen in der While-Schleife werden nie ausgeführt.* *Somit wandert einfach der rote Pfeil einmal über die ganze Zahlenfolge, durchläuft also die äussere For-Schleife.* *Das ergibt eine asymptotische Laufzeit von O(n).*
Welche asymptotische Laufzeitklasse in O-Notation hat dieser Algorithmus im Worst Case?	Im *ungünstigsten Fall muss für jedes i-te Element bei einem roten Pfeil jedes der i-1 Elemente davor ausgetauscht werden.* *Bei n Elementen wären das im Worst Case 1+2+… +n−1=n(n−1)/2 Vertauschungen.* *Das ergibt eine asymptotische Laufzeit von $O(n^2)$, also eine quadratische Laufzeit. Verdoppelt man die Länge der Zahlenfolge, so vervierfacht sich die Laufzeit des Algorithmus.*

Verbesserung des Sortieralgorithmus

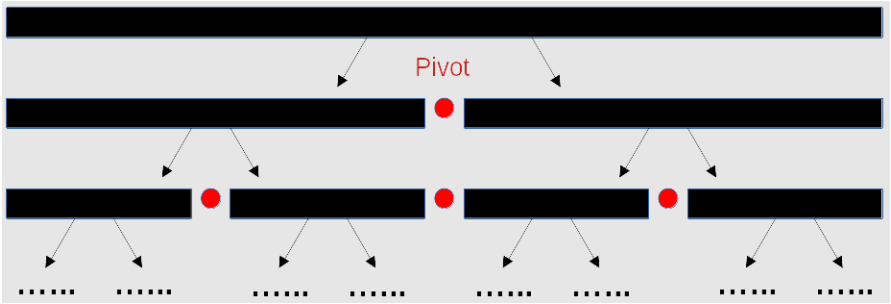
Der Algorithmus der vorhergehenden Aufgabe weist im Best Case eine lineare Laufzeit und im Worst Case eine quadratische Laufzeitkomplexität auf. Wir stellen uns in dieser Aufgabe die Frage, wie der bisher gezeigte Sortieralgorithmus verändert und somit die Laufzeitkomplexität verbessert werden kann. Wir nehmen Bezug auf die bereits bekannte Binärsuche und den Quicksort-Algorithmus.

Vervollständigen Sie in Partnerarbeit folgende Aussagen:

Das Konzept der Binärsuche kann uns bei der Verbesserung des Sortieralgorithmus helfen, da …	
Wenn die Problemgrösse bei einem Algorithmus mit logarithmischer Laufzeitkomplexität verdoppelt wird, erhöht sich die Laufzeit …	
Quicksort teilt die zu sortierende Zahlenfolge in zwei Teile und sortiert sie einzeln und rekursiv. Das bedeutet, dass auch diese zwei neuen Teile mit Quicksort sortiert werden. Im Vergleich zu Insertionsort ist die Laufzeit bei Quicksort …	
Im Worst-Case-Szenario von Quicksort gemäss Abbildung A ist das Pivot-Element die grösste oder kleinste Zahl der Zahlenfolge. Diese unglückliche Wahl des Pivot-Elementes führt zur Laufzeitkomplexität von …	
Im Best Case (Abbildung B) unterteilt Quicksort die Zahlenfolge in ähnlich grosse Teile. Dies setzt voraus, dass das gewählte Pivot-Element …	
Die Laufzeitkomplexität im Best Case des Quicksort ist …	

Abbildung A:

Abbildung B:



Das Konzept der Binärsuche kann uns bei der Verbesserung des Sortieralgorithmus helfen, da ...	*... das Konzept der Binärsuche eine effizientere Suche nach dem richtigen Einfügepunkt in Insertionsort ermöglicht.* *Durch die Anwendung der Binärsuche auf Insertionsort kann die Anzahl der Vergleiche reduziert werden. Anstatt jedes Element einzeln zu vergleichen, halbiert die Binärsuche den zu durchsuchenden Bereich in jedem Schritt. Dies führt zu einer deutlichen Verbesserung der Laufzeit des Algorithmus, insbesondere bei grossen Datenmengen.* *Beim Insertionsort wird das Array Schritt für Schritt sortiert, indem jedes Element an die richtige Position im bereits sortierten Teil des Arrays eingefügt wird. Durch die Verwendung der Binärsuche kann dieser Vorgang optimiert werden. Anstatt jedes Element sequenziell zu vergleichen, teilt die Binärsuche das Array in der Mitte auf und vergleicht das mittlere Element mit dem einzufügenden Element. Abhängig vom Vergleichsergebnis wird dann entweder der linke oder rechte Teil des Arrays weiter betrachtet. Dieser Prozess wird wiederholt, bis der richtige Einfügepunkt gefunden ist.*
Wenn die Problemgrösse bei einem Algorithmus mit logarithmischer Laufzeitkomplexität verdoppelt wird, erhöht sich die Laufzeit ...	*... lediglich um 1.*
Quicksort teilt die zu sortierende Zahlenfolge in zwei Teile und sortiert sie einzeln und rekursiv. Das bedeutet, dass auch diese zwei neuen Teile mit Quicksort sortiert werden. Im Vergleich zu	*.... um einiges schneller, besonders bei langen Zahlenfolgen.*

Insertionsort ist die Laufzeit bei Quicksort …	
Im Worst-Case-Szenario von Quicksort gemäss Abbildung A ist das Pivot-Element die grösste oder kleinste Zahl der Zahlenfolge. Diese unglückliche Wahl des Pivot-Elementes führt zur Laufzeitkomplexität von …	*… O (n^2), da auf den unterschiedlichen Stufen jeweils n-1 Vergleiche, dann n-2, dann n-3 etc. notwendig sind. Dies ergibt eine Laufzeit T(n) = (n-1) + (n-2) + … + 2 + 1 = n(n-1)/2.*
Im Best Case (Abbildung B) unterteilt Quicksort die Zahlenfolge in ähnlich grosse Teile. Dies setzt voraus, dass das gewählte Pivot-Element …	*… in etwa dem Median der Zahlenfolge entspricht. Dies ist der Best Case.*
Die Laufzeitkomplexität im Best Case von Quicksort ist …	*… O (n * log(n)), da auf jeder der log(n)- Stufen maximal n Vergleiche notwendig sind.*

Motivation für Heapsort

Wir haben bisher gesehen, dass der Insertionsort unter Anwendung der Binärsuche und mit Komplexitätsmass «Anzahl Vergleiche» eine Laufzeitkomplexität von O(n * log(n)) aufweist. Jedoch kann es bei jedem Einfügen zu einer linearen Anzahl von Zuweisungen kommen und wenn man als Komplexitätsmass die «Anzahl Zuweisungen» nimmt, so ist die Laufzeitkomplexität quadratisch, also $O(n^2)$.

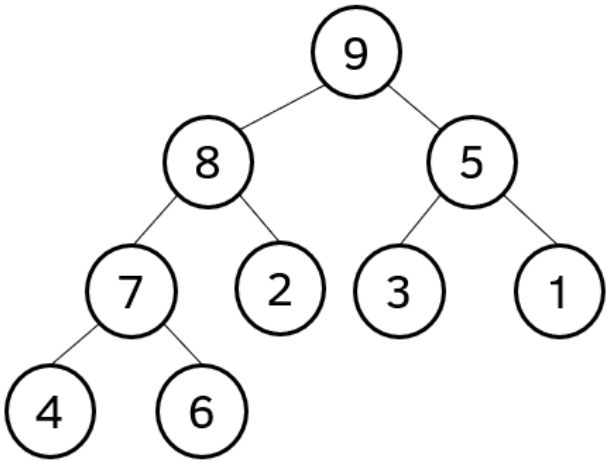
Der Quicksort-Algorithmus weist eine erwartete superlineare Laufzeit von O(n * log(n)) auf. Durch die unglückliche Wahl des Pivot-Elementes kann die Laufzeitkomplexität des Quicksort jedoch auch $O(n^2)$ betragen kann. Lassen Sie uns der Frage nachgehen, wann der Worst Case in der Praxis effektiv vorkommt? Mit Rechenarbeit kann man zeigen, dass der Quicksort-Algorithmus im Average Case eine asymptotische Laufzeit von O(n * log(n)) besitzt. Das ist also eine merkliche Verbesserung im Vergleich zum Insertionsort-Algorithmus mit $O(n^2)$.

Die Motivation zur Weiterentwicklung des Sortieralgorithmus besteht darin, dass wir einen Algorithmus entwickeln möchten, der konstant eine superlineare Laufzeit O (n * log(n)) aufweist, sowohl im Worst Case, im Best Case als auch im Average Case. Ein solcher Algorithmus ist der Heapsort und zählt dabei Vergleiche sowie Zuweisungen.

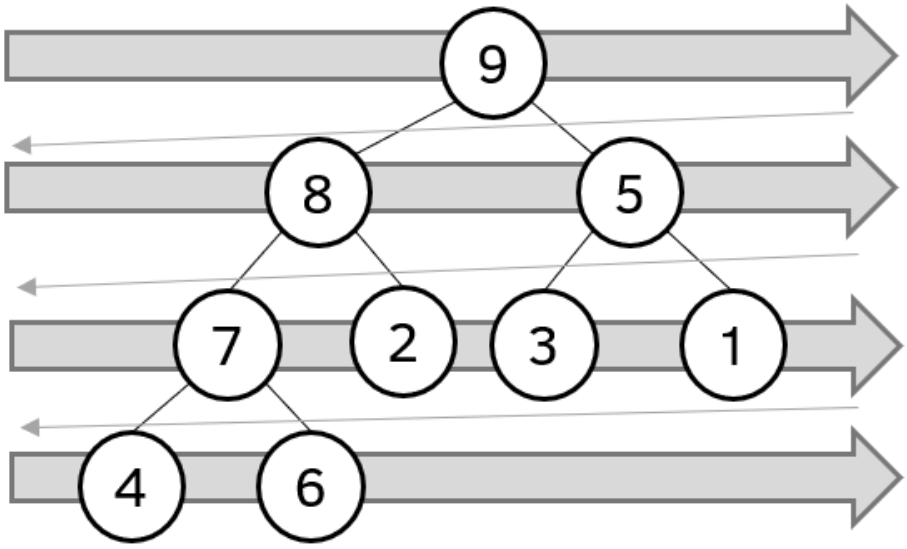
Die Idee des Heapsorts ist es, eine Menge (unsortierte oder teilweise sortierte Zahlenfolge) in einem Binärbaum so zu speichern, dass

a. man auf das maximale Element in der Wurzel sofort zugreifen kann;
b. nachdem man das maximale Element genommen hat, man mit logarithmischem Aufwand das neue Maximum in die Wurzel bekommen kann.

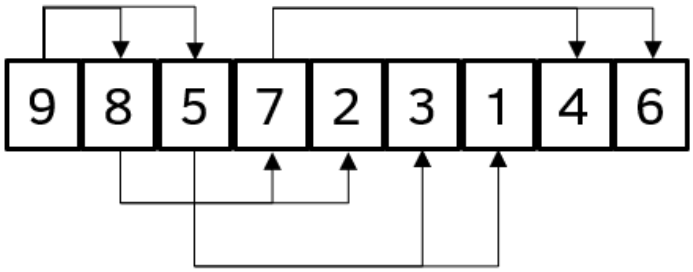
Dabei bezeichnet Heap einen Binärbaum, in dem der Wert eines Knotens entweder grösser oder gleich als der seiner Kinder (Max-Heap) oder kleiner oder gleich als der seiner Kinder (Min-Heap) ist.



Ein Heap wird auf ein Array projiziert, indem dessen Elemente von oben links zeilenweise nach rechts unten in das Array übertragen werden:



Der oben gezeigte Heap sieht als Array also so aus:

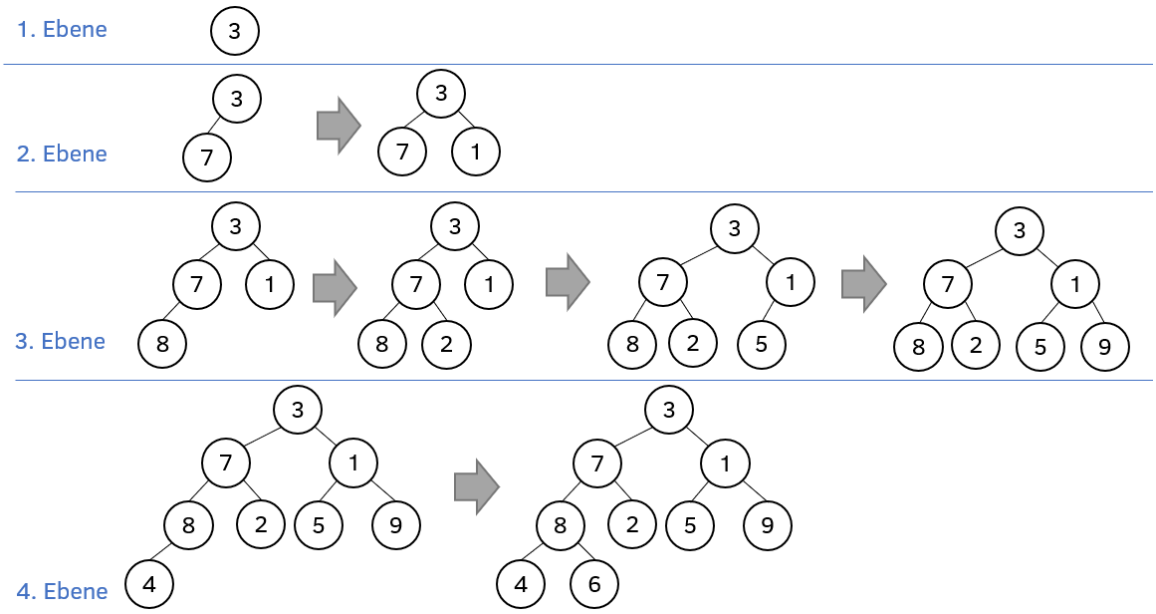


Bei einem Max-Heap ist das grösste Element immer ganz oben – in der Array-Form ist es folglich ganz links.

Binärbaum erstellen

Ein Binärbaum ist eine Art von Baumstruktur, mittels der wir eine Zahlenfolge als Gedankenkonstrukt grafisch darstellen können. Er ist also keine separate Datenstruktur. Die Zahlenfolge liegt weiterhin als Array im Speicher. Der Elternknoten steht über dem/den Kindknoten. Wir beachten, dass jeder Elternknoten höchstens zwei Kinder haben kann und dass die Anzahl linker Knoten im Unterbaum grösser oder gleich der Anzahl Knoten im rechten Unterbaum ist.

Schauen Sie sich folgende Illustration zur Erstellung eines Binärbaums für die Zahlenfolge [3, 7, 1, 8, 2, 5, 9, 4, 6] an und notieren Sie die diesbezüglich notwendigen Schritte.

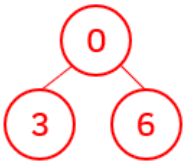
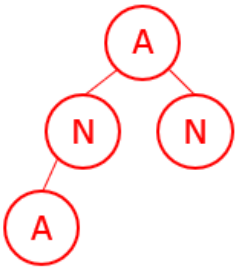
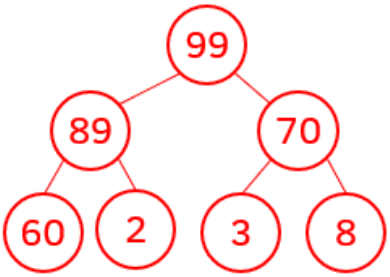


1. Schritt (Start)	
2. Schritt (Schleife)	

1. Schritt (Start)	*Ich erstelle einen ersten Elternknoten mit dem Wert des ersten Elements der Zahlenfolge.*
2. Schritt (Schleife)	*Solange es ein nächstes Element in der Zahlenfolge gibt, nehme ich dieses und füge es als Kindknoten dem Binärbaum hinzu. Dabei achte ich darauf, dass je Elternknoten max. zwei Kindknoten erlaubt sind und die Knoten innerhalb einer Ebene von links nach rechts hinzugefügt werden.*

Erstellen Sie für folgende Zahlenfolgen die entsprechenden Binärbäume.

Zahlenfolge	***Binärbaum***
[0, 3, 6]	
[A, N, N, A]	
[99, 89, 70, 60, 2, 3, 8]	

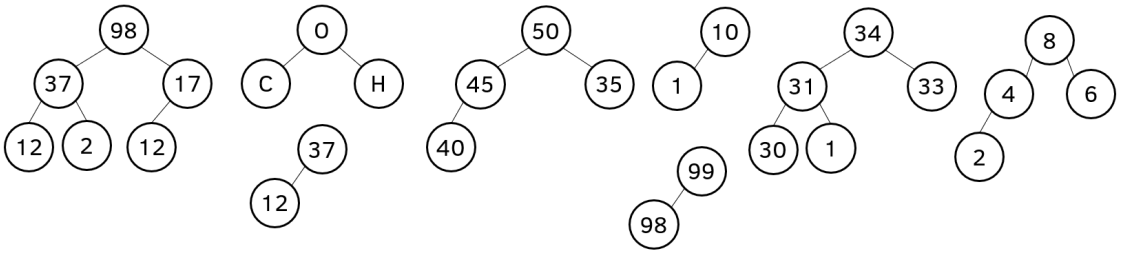
[0, 3, 6]	
[A, N, N, A]	
[99, 89, 70, 60, 2, 3, 8]	

Heapbedingung erkennen

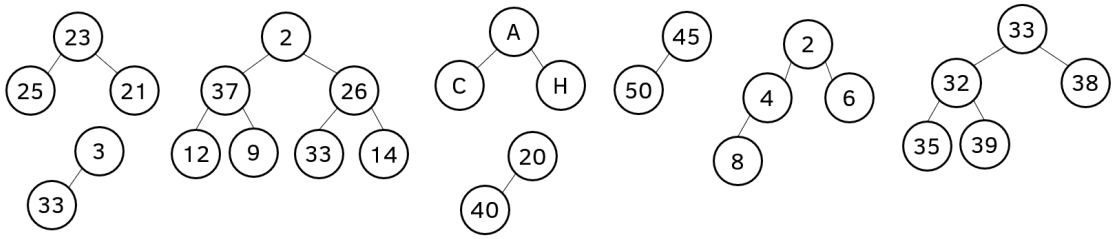
Der Heapsort kann nur auf Binärbäume angewandt werden, die eine bestimmte Heapbedingung erfüllen. In dieser Aufgabe wollen wir diese gemeinsam entdecken.

Schauen Sie sich die folgenden Binärbäume an und überlegen Sie, welche Heapbedingung abgeleitet werden könnte. Notieren Sie diese Heapbedingung.

Heapbedingung erfüllt



Heapbedingung nicht erfüllt



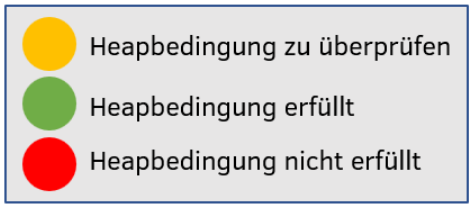
Heapbedingung:

Tipp: Vergleichen Sie die Zahlen auf den unterschiedlichen Ebenen der Binärbäume.

Heapbedingung:
Max-Heap: Wert im Elternknoten >= Wert im Kindknoten

Max-Heap mittels heapify-Funktion erstellen

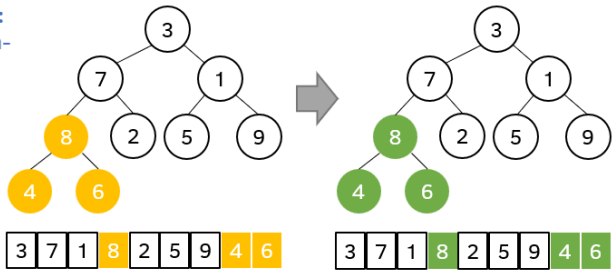
Eine zentrale Aufgabe im Heapsort-Algorithmus führt die Funktion heapify() aus. Diese strukturiert den Heap. Startpunkt der heapify-Funktion ist der konstruierte Binärbaum einer unsortierten Zahlenfolge, welcher die Heapbedingung eines Max-Heaps nicht erfüllt. Die heapify-Funktion besucht alle Elternknoten vom letzten zum ersten und sorgt dafür, dass die Heapbedingung erfüllt wird und somit ein Max-Heap entsteht. Wie man in den nachfolgenden Illustrationen sieht, arbeitet der hier vorgestellte Algorithmus von unten nach oben, also von den Blättern in Richtung Wurzel.



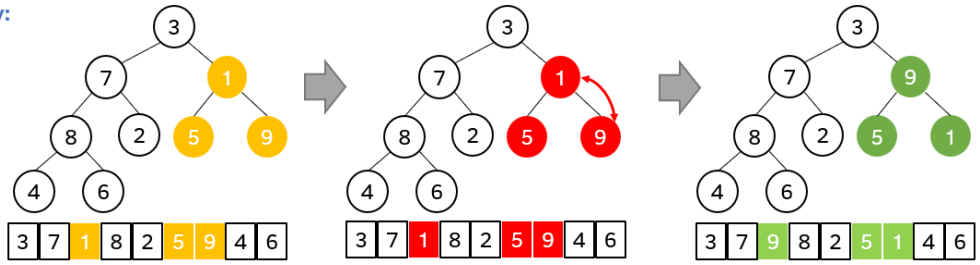
a) Schauen Sie sich die folgenden Illustrationen an und beschreiben Sie die heapify-Funktion in eigenen Worten.

b) Überlegen Sie sich auch, ob der Algorithmus zur Erstellung des Heaps auch funktionieren würde, wenn er von oben nach unten, also von der Wurzel zu den Blättern verlaufen würde?

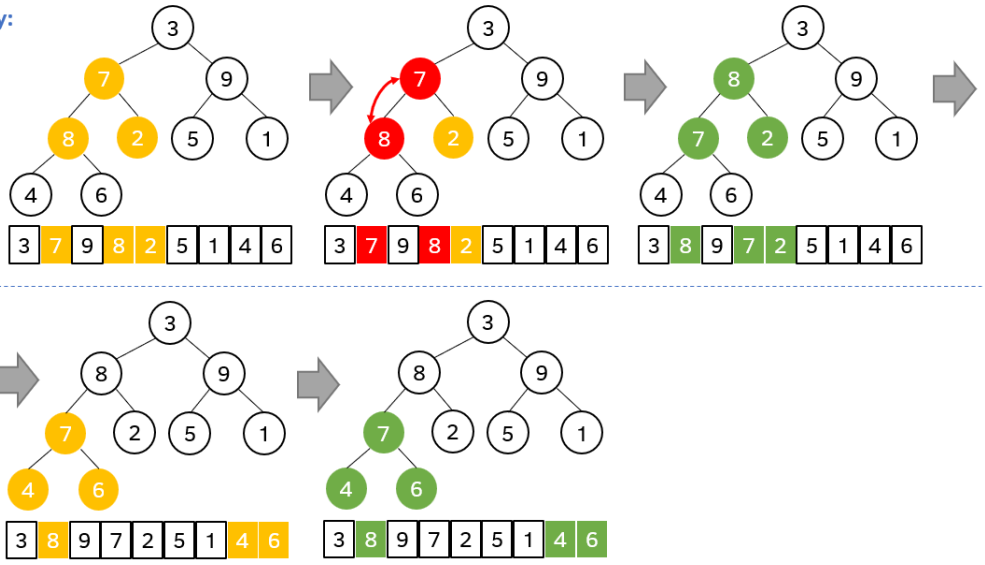
1. Aufruf heapify:
auf letztem Elternknoten (8)



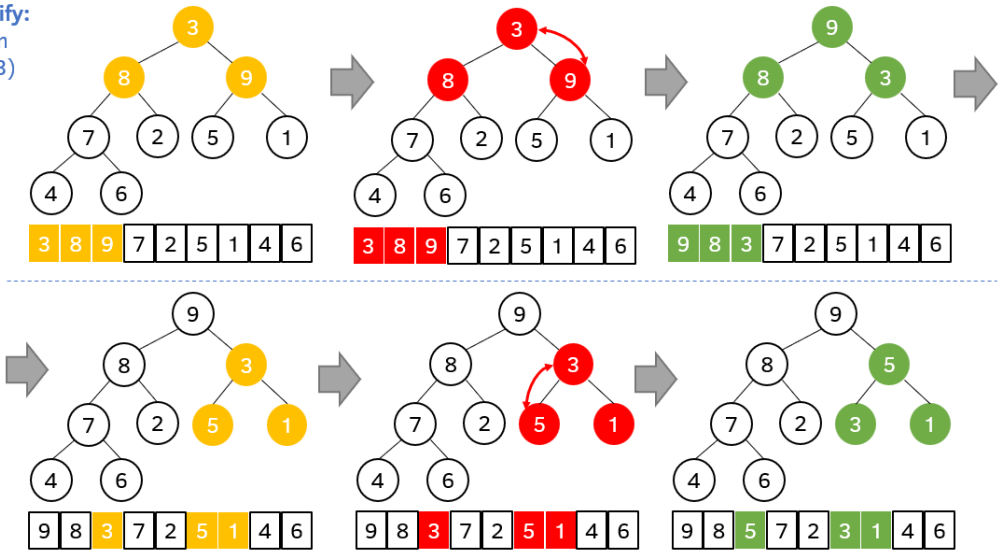
2. Aufruf heapify:
auf zweitletztem Elternknoten (1)



3. Aufruf heapify:
auf drittletztem
Elternknoten (7)



4. Aufruf heapify:
auf viertletztem
Elternknoten (3)



a) Beschreibung der heapify-Funktion in eigenen Worten:

b) Würde der Algorithmus zur Erstellung des Heaps auch funktionieren, wenn er von oben nach unten, also von der Wurzel zu den Blättern, verlaufen würde?

a) heapify-Funktion in eigenen Worten:

- Zuerst überprüft die heapify-Funktion durch Wertevergleich beim letzten Elternknoten, ob die Kindknoten kleiner sind als der Elternknoten.
  - Falls die Heapbedingung erfüllt ist, geht der Algorithmus zum nächsten Elternknoten (Richtung Wurzel) und überprüft wieder die Heapbedingung.
  - Falls die Heapbedingung nicht erfüllt ist, wird der Elternknoten mit dem grösseren der beiden Kindknoten vertauscht. Dieser kann auch Auswirkungen auf die darunter liegenden Ebenen haben. Falls der vertauschte Kindknoten selbst auch Kindknoten hat, wird darauf wieder die heapify-Funktion aufgerufen. Dies ist somit ein rekursiver Aufruf, allerdings nur für den Kindknoten, den wir ausgetauscht haben.
- Wenn alle Elternknoten besucht sind und bei allen Kindern die Heapbedingung erfüllt ist, ist die heapify-Funktion beendet.

b) Würde der Algorithmus zur Erstellung des Heaps auch funktionieren, wenn er von oben nach unten, also von der Wurzel zu den Blättern, verlaufen würde?

Ja, ein Algorithmus zur Erstellung des Heaps kann auch funktionieren, wenn er von oben nach unten, also von der Wurzel zu den Blättern, verläuft. Es handelt sich um den sogenannten "top-down-heapify"-Algorithmus. Dieser überprüft die Heap-Eigenschaft für jeden Knoten und passt diese gegebenenfalls an, während er von der Wurzel zu den Blättern durch den Baum geht. Die einzelnen Schritte im Falle eines Max-Heap sind dann wie folgt:

1. Beginne bei der Wurzel des Baumes.
2. Vergleiche den Wert des aktuellen Knotens mit den Werten seiner Kinder.
3. Wenn der aktuelle Knoten kleiner ist als einer seiner Kinder, tausche den Wert des Knotens mit dem Wert des grösseren Kindes.
4. Gehe zum grösseren Kindknoten und wiederhole die Schritte 2-3, bis der aktuelle Knoten die Heap-Eigenschaft erfüllt oder ein Blatt erreicht wird.
5. Gehe zum nächsten Knoten in Richtung der Blätter und wiederhole die Schritte 2-4, bis der gesamte Baum überprüft wurde.

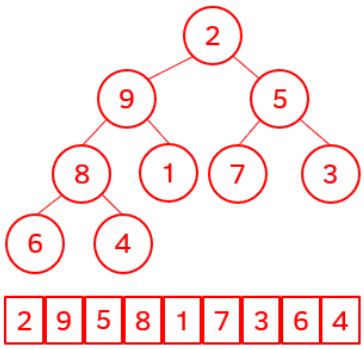
Der top-down-heapify-Algorithmus ist somit etwas komplizierter als der bottom-up-heapify-Algorithmus.

Binärbaum und heapify-Funktion

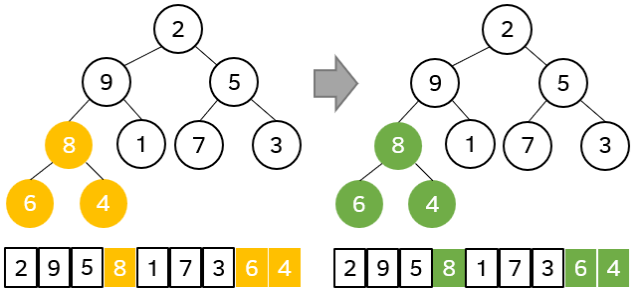
Erstellen Sie für die Zahlenfolge 2, 9, 5, 8, 1, 7, 3, 6, 4 einen Binärbaum und wenden Sie darauf die heapify-Funktion an, damit die Heapbedingung für einen Max-Heap erfüllt ist.

Binärbaum
1. Aufruf heapify-Funktion
2. Aufruf heapify-Funktion
3. Aufruf heapify-Funktion
4. Aufruf heapify-Funktion

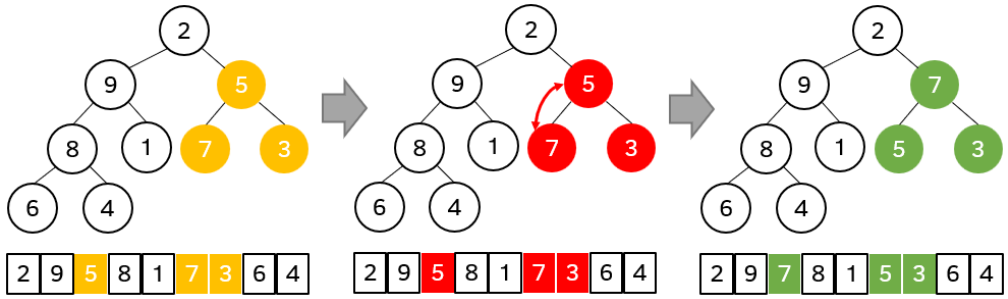
Binärbaum



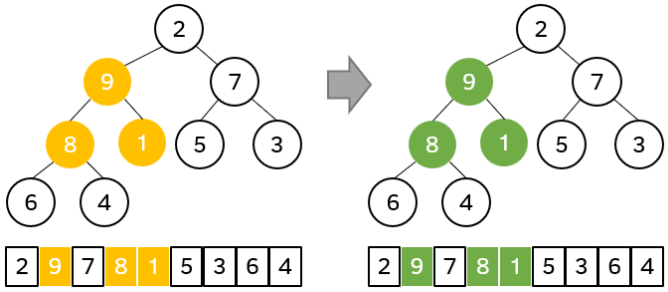
1. Aufruf heapify-Funktion



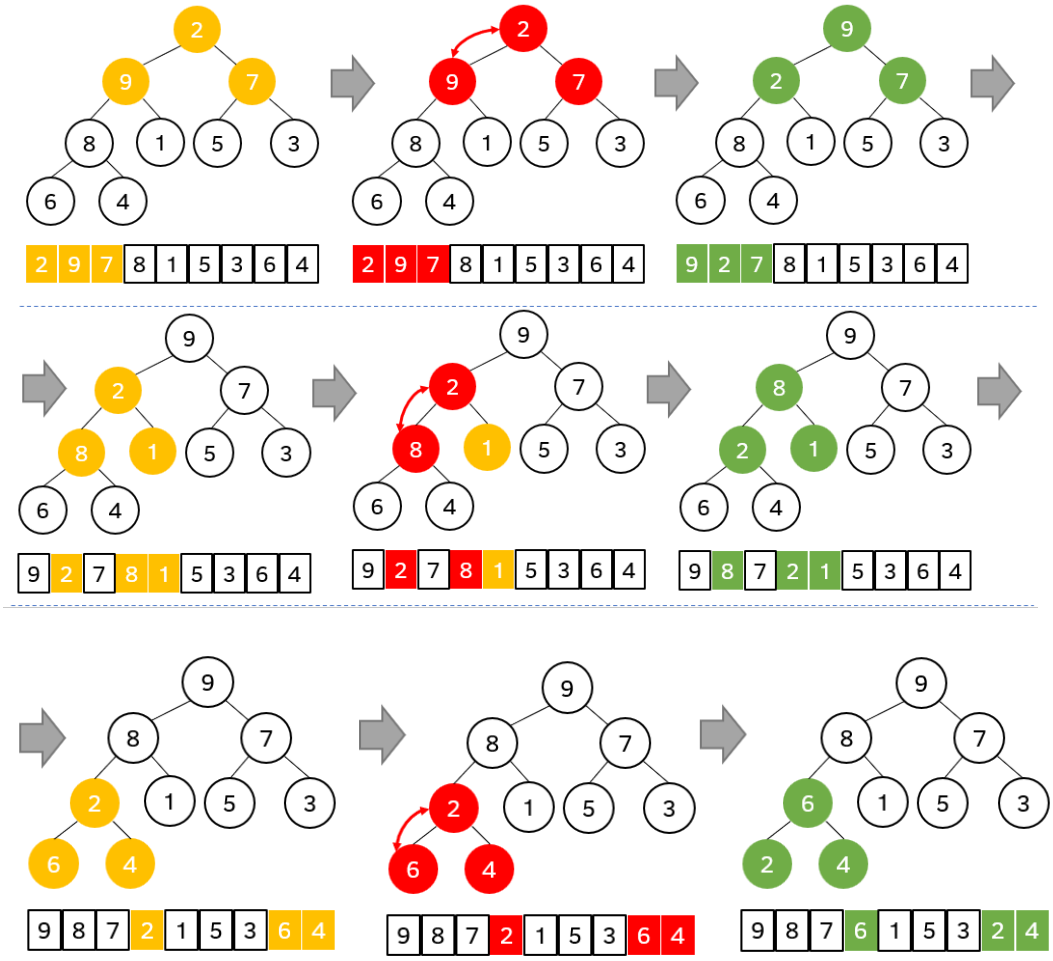
2. Aufruf heapify-Funktion



3. Aufruf heapify-Funktion



4. Aufruf heapify-Funktion



Heapsort-Laufzeitkomplexität analysieren

Abschliessend möchten wir die Laufzeitkomplexität des Heapsort-Algorithmus analysieren. Dafür schauen wir uns die beiden Funktionen separat an:

a) heapify-Funktion zur Erstellung der Max-Heap-Eigenschaft (u.a. nach der Wurzelentfernung)
b) Build-Heap-Funktion zur Erstellung des Heaps
c) Gesamte Laufzeitkomplexität von Heapsort

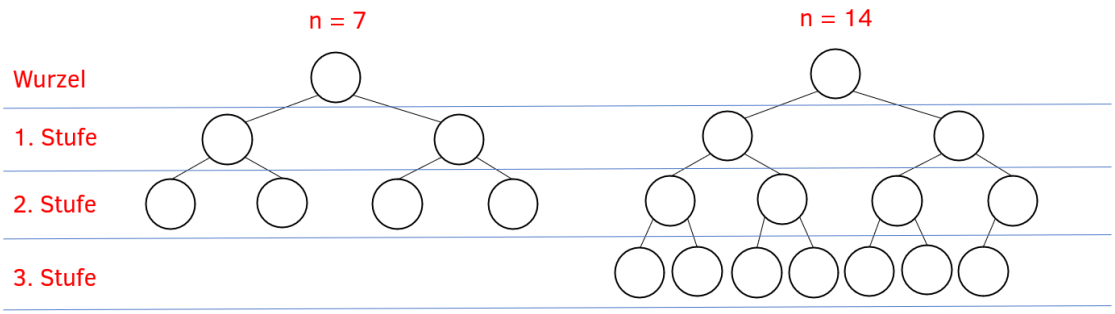
a) heapify-Funktion

Analysieren wir zuerst die heapify-Funktion, welche auch für den initialen Aufbau des Heaps benötigt wird. In der heapify-Funktion hangeln wir uns einmal von oben nach unten durch den Baum. Die Höhe eines Binärbaumes wird ohne Wurzel gezählt.

1. *Zeichnen Sie einen Binärbaum der Grösse 7 (also 7 Knoten).*
2. *Zeichnen Sie einen zweiten Binärbaum mit doppelter Anzahl Knoten.*
3. *Wie viele Stufen (also Ebenen) werden durch die Verdoppelung der Knoten (oder Elemente) hinzugefügt? Um welche Laufzeitklasse handelt es sich bei dieser Funktion?*

1. Binärbaum mit n=7
2. Binärbaum mit n=14
3. Laufzeitkomplexität?

Da die Höhe eines Binärbaumes der Grösse n maximal $\log_2 n$ ist, wird der Binärbaum bei einer Verdopplung der Anzahl Knoten lediglich eine Ebene tiefer:



Die Komplexität der heapify-Funktion ist demzufolge logarithmisch, also *O(log n)*.

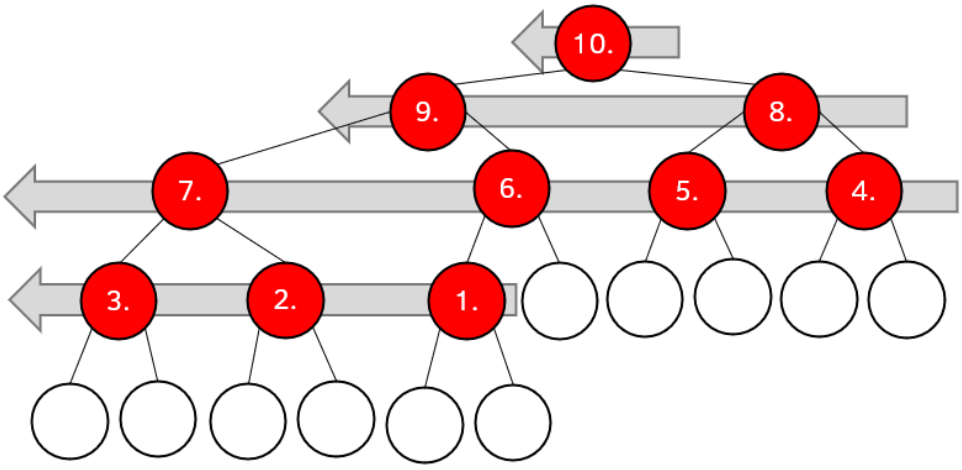
b) Build-Heap-Funktion zur Erstellung des Heap

Zur Erstellung des Heaps wird für jeden Elternknoten die heapify-Funktion aufgerufen. Gestartet wird beim letzten Elternknoten, rückwärts laufend bis zur Wurzel.

1. *Zeichnen Sie einen Binärbaum der Grösse 21 (also 21 Knoten).*
2. *Wie viele Elternknoten sind vorhanden?*
3. *Um welche Laufzeitklasse handelt es sich bei dieser Funktion?*

1. Binärbaum mit n=21
2. Anzahl Elternknoten?
3. Laufzeitkomplexität?

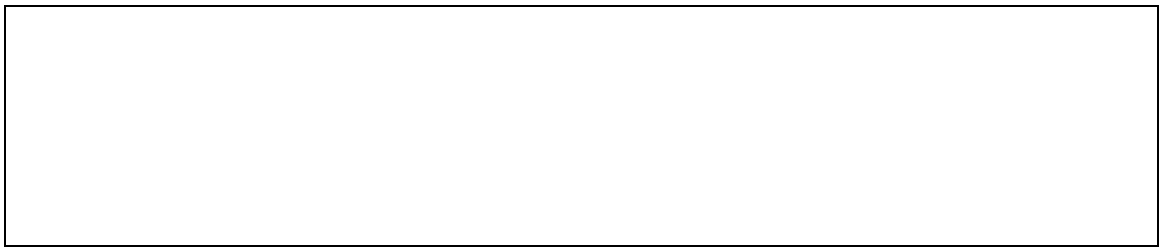
Ein Heap der Grösse n hat abgerundet n/2 Elternknoten. In einem Binärbaum mit n=21 Knoten gibt es 10 Elternknoten.



Da die Komplexität der heapify-Funktion O(log (n)) ist, ist die Komplexität für die Build-Heap-Funktion maximal O(n * log (n)).

c) Gesamtlaufzeitkomplexität von Heapsort

Überlegen Sie sich basierend auf a) und b) die gesamte Laufzeitkomplexität von Heapsort und notieren Sie Ihre Überlegungen:



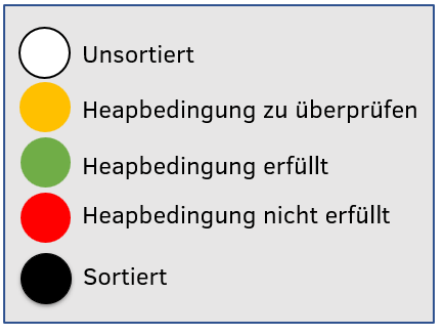
Beide Teilalgorithmen haben gleiche Zeitkomplexität.

- Die Laufzeit für die initiale Erstellung des Heaps ist O(n * log (n)).
- Die Gesamtlaufzeitkomplexität für das Reparieren des Heaps nach der Wurzelentfernung ist auch O(n * log (n)).

Die Zeitkomplexität von Heapsort beträgt demzufolge: O(n * log (n)).

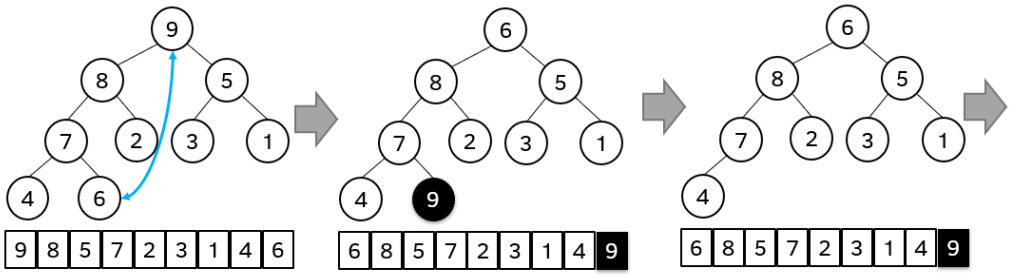
Heapsort-Algorithmus verstehen

Im eigentlichen Heapsort-Algorithmus machen wir uns die Tatsache zunutze, dass das grösste Element des Max-Heaps immer an dessen Wurzel (also zuoberst im Baum beziehungsweise ganz links im Array) steht.

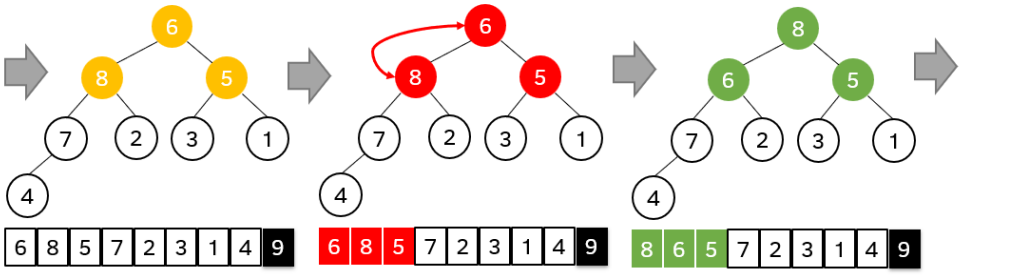


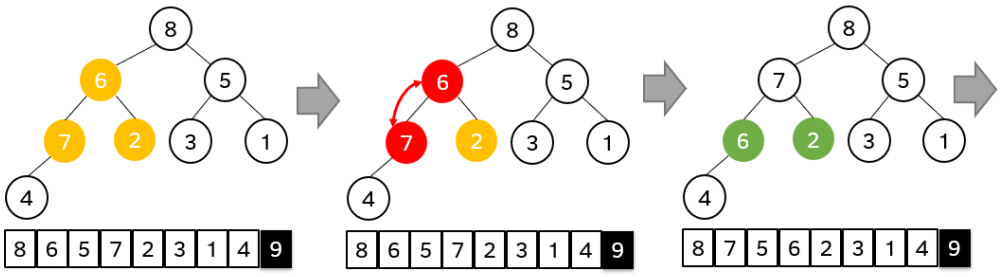
Schauen Sie sich die folgenden Illustrationen an und beschreiben Sie den Heapsort-Algorithmus in eigenen Worten.

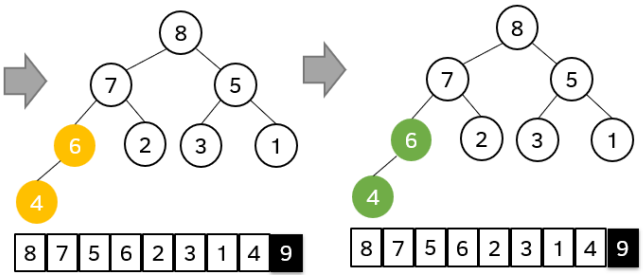
1. Aufruf, Schritt 1

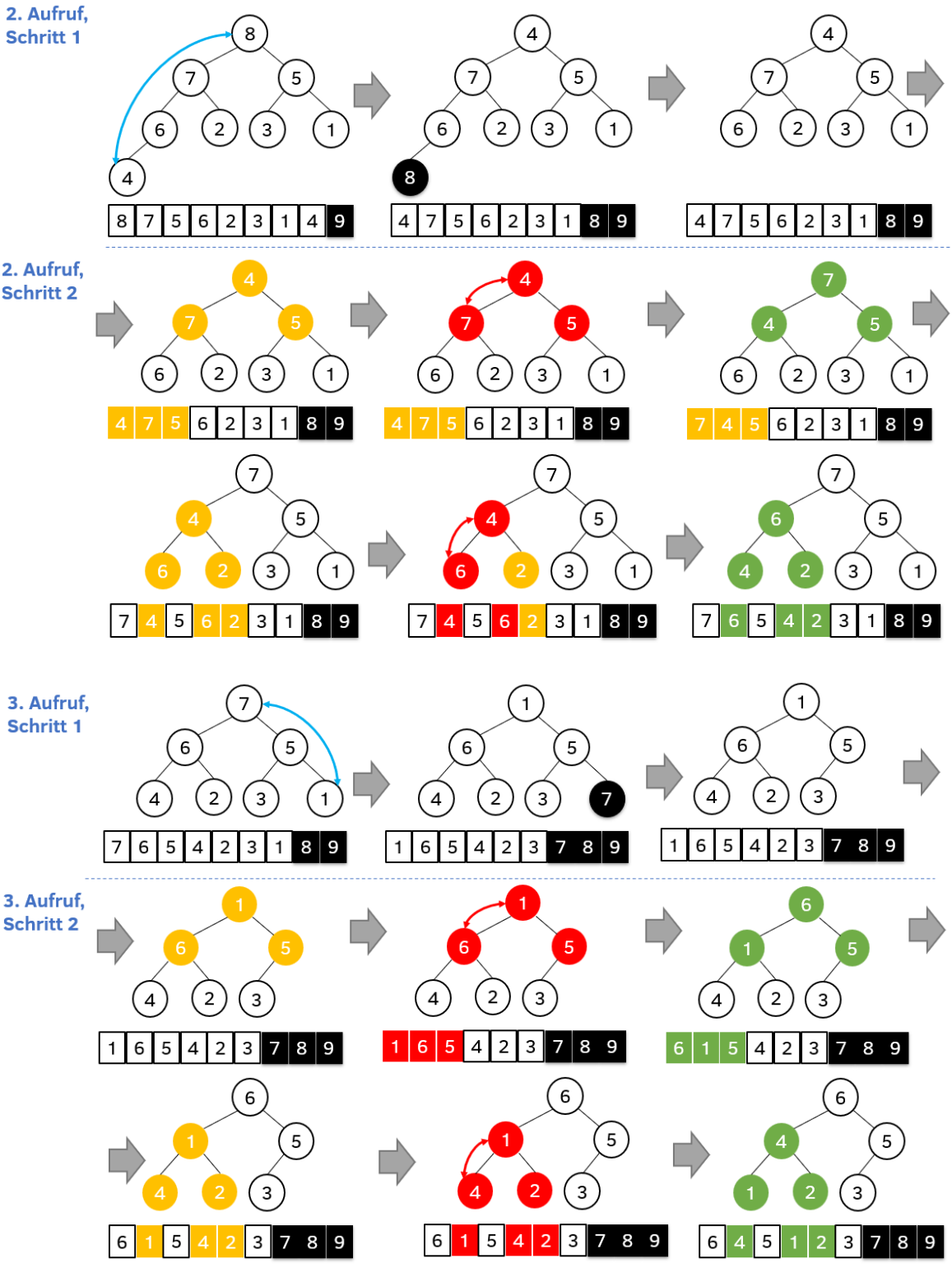


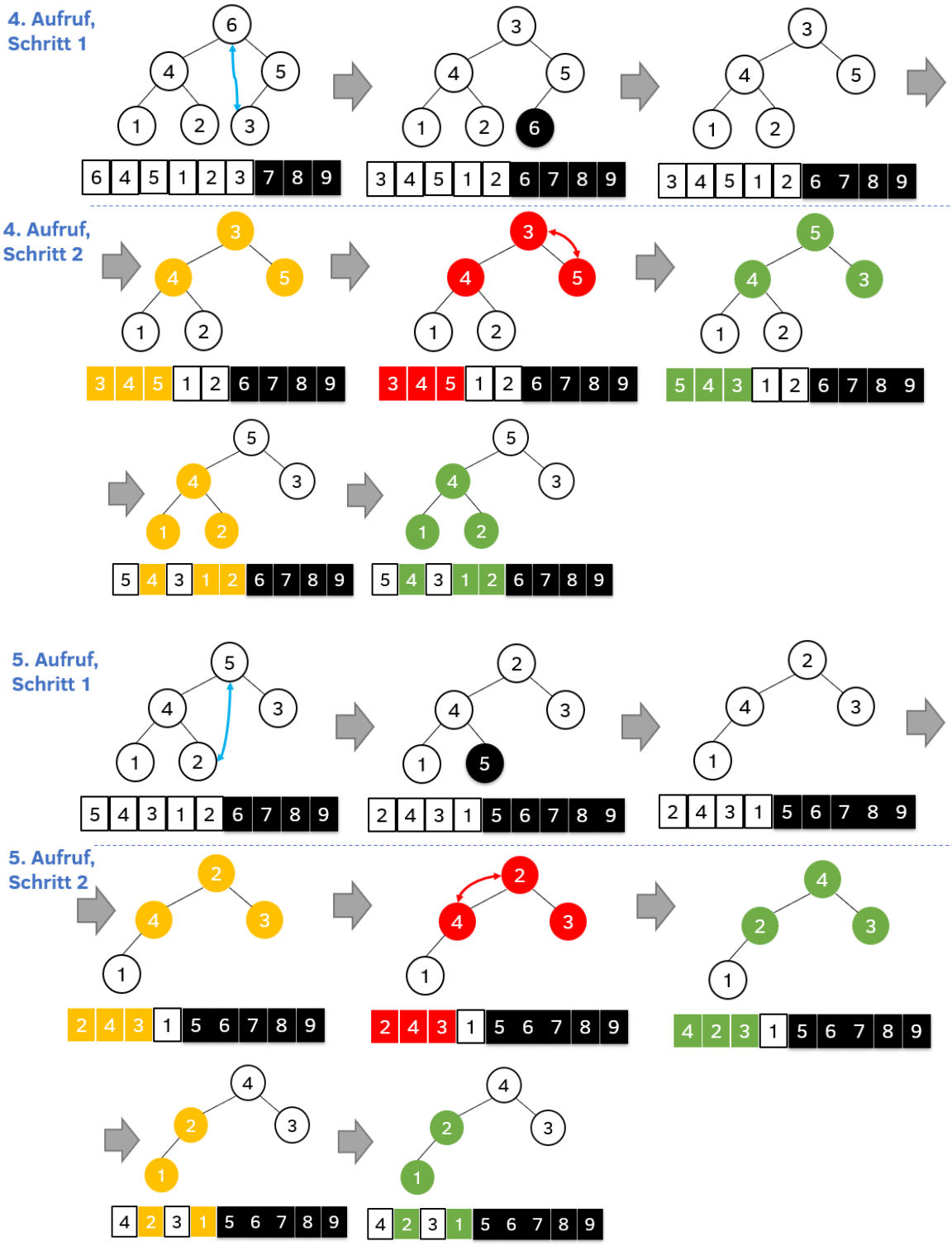
1. Aufruf, Schritt 2

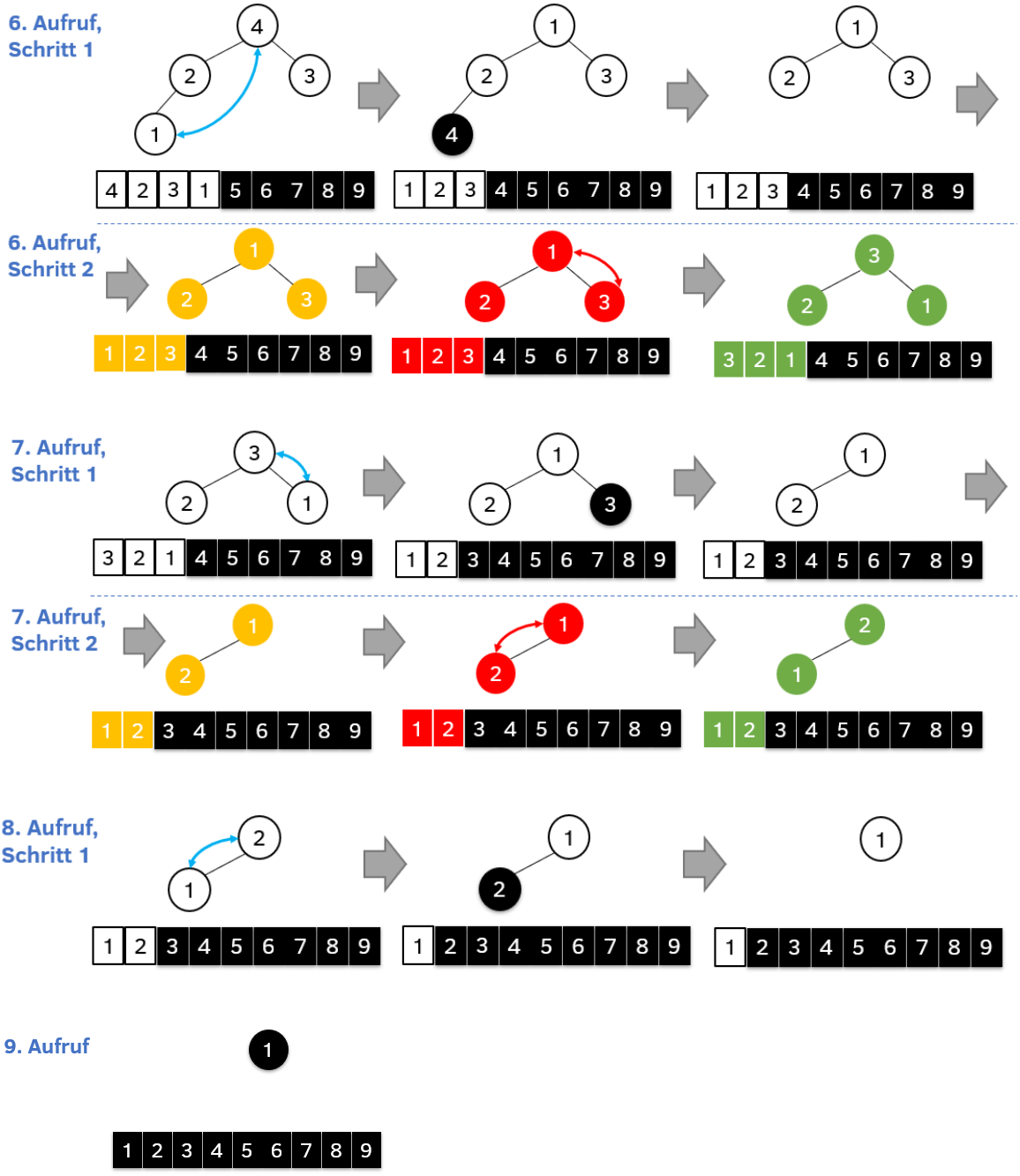




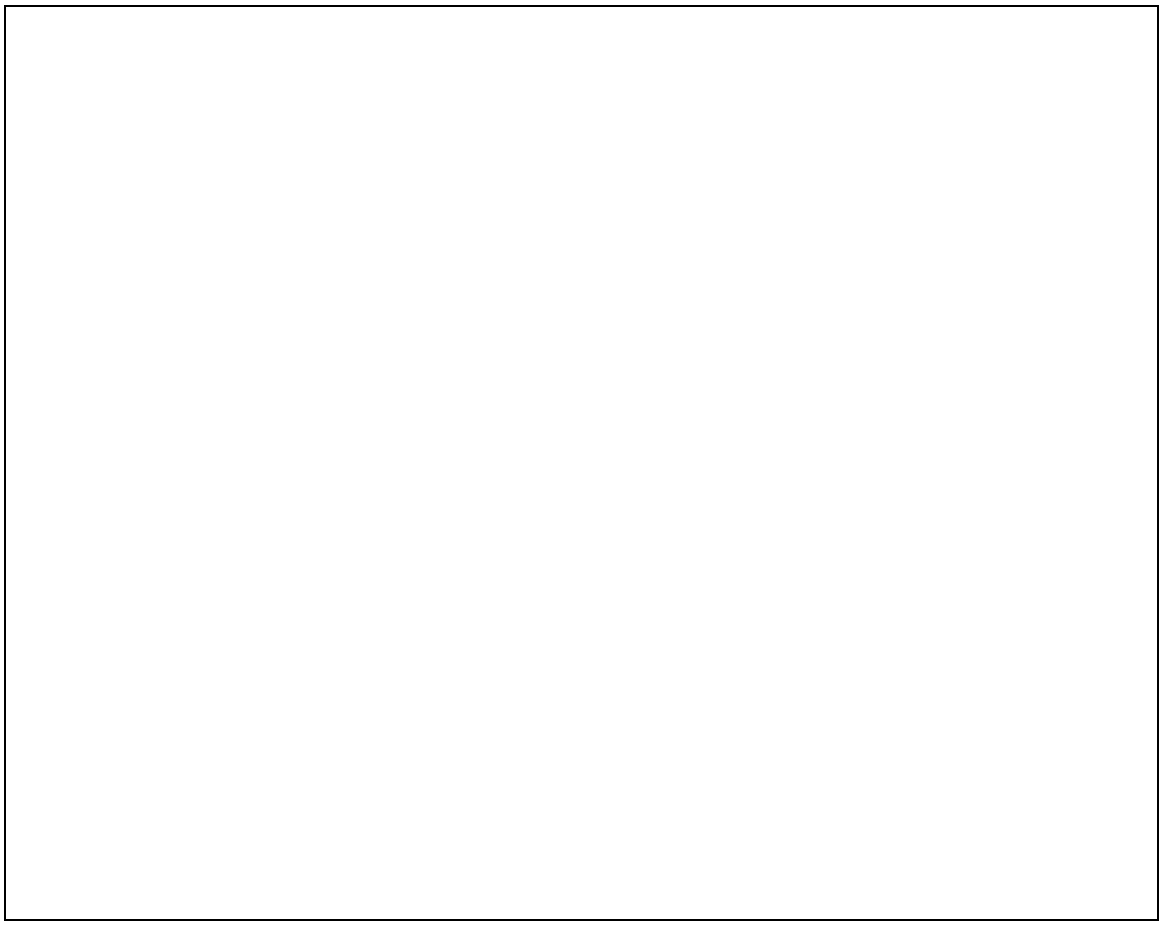



2. Aufruf, Schritt 1
2. Aufruf, Schritt 2
3. Aufruf, Schritt 1
3. Aufruf, Schritt 2


4. Aufruf, Schritt 1
4. Aufruf, Schritt 2
5. Aufruf, Schritt 1
5. Aufruf, Schritt 2



Beschreibung des Heapsort-Algorithmus in eigenen Worten:



Schritt 1: Das Wurzelelement als jeweils höchste Zahl wird mit dem letzten Element getauscht, so dass die höchste Zahl an ihrer finalen Position sortiert im Array ist. Gedanklich wird das höchste Element aus dem Baum entfernt und in der sortierten Teilfolge an die richtige Position gebracht.

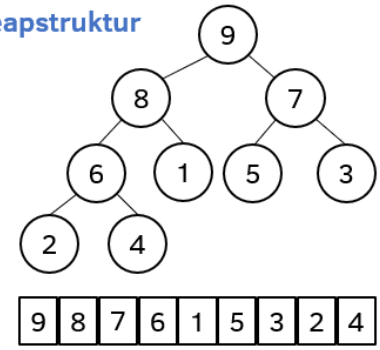
Schritt 2: Die heapify-Funktion wird auf dem neuen Wurzelknoten aufgerufen und auf dem grösseren der beiden Kinder wiederholt, bis der ganze Baum wieder die Heapbedingung für einen Max-Heap erfüllt.
Achtung: Die heapify-Funktion wird nur auf den Knoten entlang des Weges von der neuen Wurzel zu einem Blatt angewendet. Es werden also nicht alle Elternknoten besucht.

Die Schritte 1 und 2 werden so lange wiederholt, bis der Baum nur noch aus einem Element besteht. Dann ist das Array sortiert.

Heapsort-Algorithmus anwenden

Führen Sie den Heap-Algorithmus auf folgendem Binärbaum in Heapstruktur aus.

Binärbaum in Heapstruktur

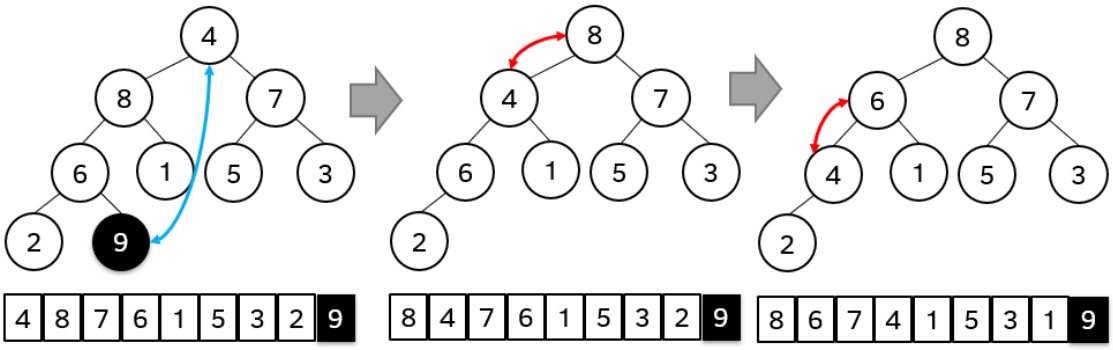


1. Aufruf
2. Aufruf
3. Aufruf
4. Aufruf

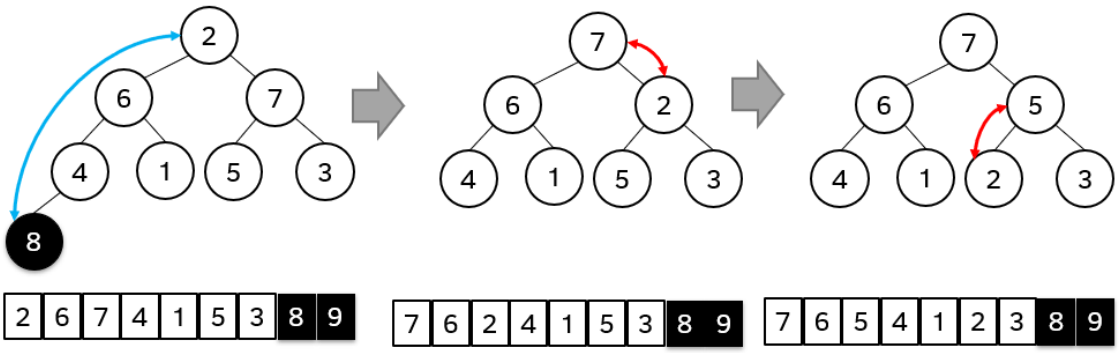
5. Aufruf
6. Aufruf
7. Aufruf
8. Aufruf
9. Aufruf

Die Musterlösung zu obiger Aufgabe zeigt nur die notwendigen Vertauschungen. Die detaillierten Zwischenschritte werden aus Effizienzgründen nicht gezeigt.

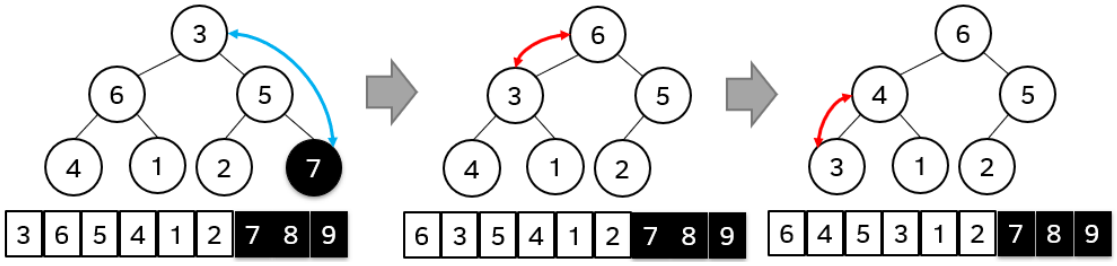
1. Aufruf



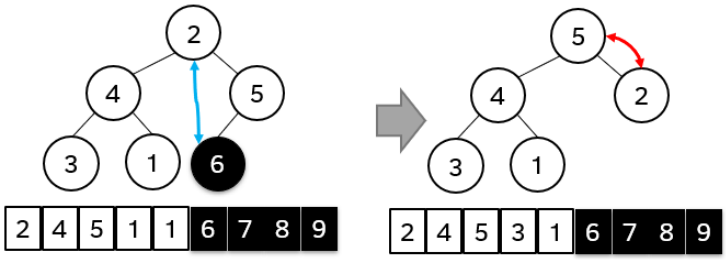
2. Aufruf



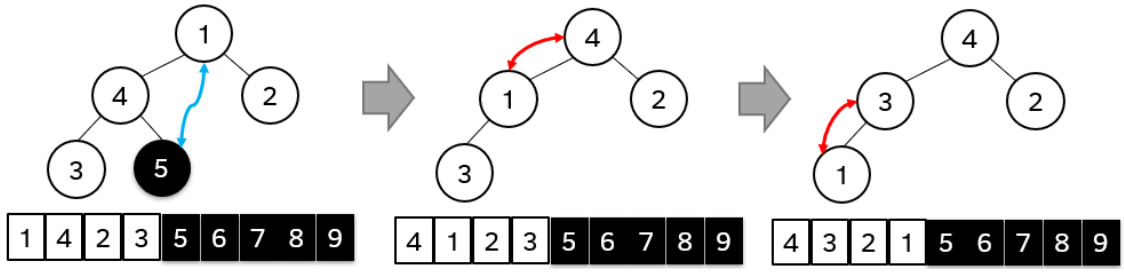
3. Aufruf



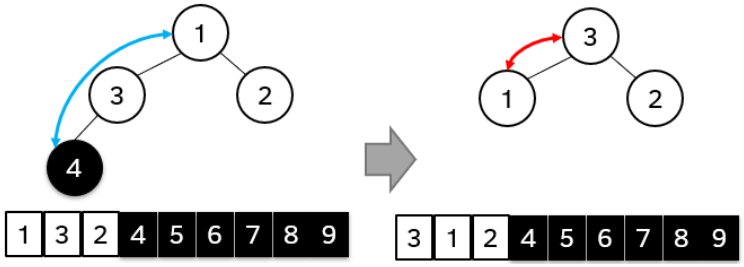
4. Aufruf



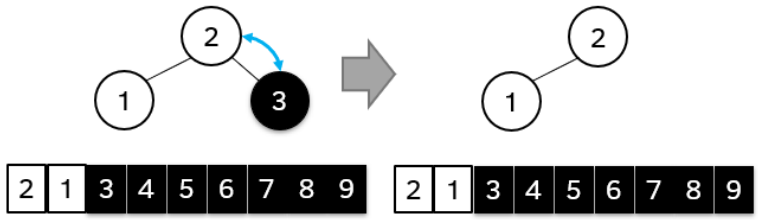
5. Aufruf



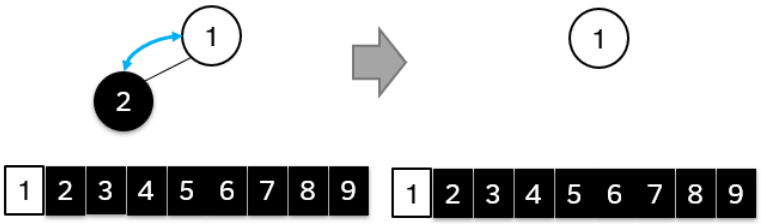
6. Aufruf



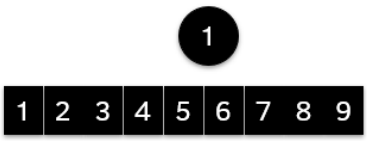
7. Aufruf



8. Aufruf



9. Aufruf



3. Fazit

In dieser Unterrichtssequenz zum Thema Heapsort haben die Schüler:innen die grundlegenden Konzepte der Binärbäume, der heapify-Funktion und des Heapsort-Algorithmus kennengelernt. Sie haben gelernt, wie man Binärbäume und Heaps erstellt und manipuliert und wie der Heapsort-Algorithmus funktioniert. Auch haben die Schüler:innen die Laufzeitkomplexität des Algorithmus analysiert.

Durch die Übungsaufgaben haben die Schüler:innen ihr Verständnis des Heapsort-Algorithmus vertieft und können nun eigenständig den Algorithmus durchführen.

Besonders hervorzuheben ist die fachdidaktische Komponente des Ampelsystems, das den Schüler:innen grafisch einfach die Funktionsweise des Heap-Algorithmus erklärt.