Einführung in die Heap-Datenstruktur und das darauf basierende Sortierverfahren Heapsort

Manuel Wettstein

22. Juni 2023

In diesen Unterrichtsunterlagen wollen wir uns der so genannten Heap-Datenstruktur widmen. Diese Datenstruktur ermöglicht es, eine gegebene Menge von Datenpunkten zu verwalten (zum Beispiel eine Menge von natürlichen Zahlen wie $\{3, 2, 7, 8\}$), so dass jederzeit neue Elemente effizient hinzugefügt werden können (durch Einfügen der Zahl 4 bekämen wir beispielsweise die neue Menge $\{3, 2, 7, 8, 4\}$) und so dass jederzeit das aktuelle Maximum effizient abgerufen und gelöscht werden kann (nach Löschen des aktuellen Maximums 8 würde also die Menge $\{3, 2, 7, 4\}$ übrig bleiben).

Als Hauptanwendung des Heaps werden wir sehen, wie das bekannte Sortierverfahren Selectionsort von der bisherigen Laufzeit $O(n^2)$ auf $O(n \log n)$ ohne zusätzlichen Speicherbedarf beschleunigt werden kann. Das resultierende effizientere Sortierverfahren ist unter dem passenden Namen Heapsort bekannt.

Folgende Grundlagen und Begriffe werden für die effektive Bearbeitung dieser Unterlagen als Vorwissen vorausgesetzt:

- Asymptotische Notation für Laufzeitanalyse von Algorithmen wie beispielsweise $O(n^2)$ oder $O(n \log n)$.
- Detailliertes Verständnis der Funktionsweise und Analyse des Sortierverfahrens Selectionsort.
- Vertrautheit mit binären Bäumen und deren Grundbegriffen wie Knoten, Kanten, Eltern und Kinder.
- Kontrollstrukturen, Funktionsaufrufe und Arrays in der Programmiersprache Rust.[1]

Wir erarbeiten das Thema mit Hilfe von Aufgaben. Diese sind in drei Kategorien unterteilt und können anhand des Symbols, mit dem sie gekennzeichnet sind, unterschieden werden:

- Durch Bearbeiten dieser *Knobelaufgaben* wird die Schwierigkeit des zu lösenden Problems erkannt, das Interesse für das Problem wird geweckt und wichtige Erkenntnisse für dessen Lösung werden erworben.
- Durch diese *Lern- und Projektaufgaben* wird entweder eine Thematik weiter vertieft oder ein Schritt in Richtung Lösung des Problems wird selbständig gegangen.
- Durch Lösen dieser *Routineaufgaben* wird das erworbene Wissen gefestigt und Verständnislücken können aufgedeckt werden.

Zudem gibt es für die selbständige Auseinandersetzung mit dem Material die folgenden Hilfestellungen:

- Am Ende jedes Abschnitts werden die wichtigsten Konzepte kurz und prägnant zusammengefasst.
- Ausführliche Lösungen zu den Aufgaben befinden sich im Anhang.

[1] Wahlweise kann jede beliebige Programmiersprache verwendet werden. Beachten Sie jedoch, dass sämtlicher Code in Aufgabenstellungen und Musterlösungen in der oben genannten Sprache geschrieben ist.

Inhaltsverzeichnis

1 Motivation und Eigenschaften **2**
1.1 Die Heap-Eigenschaft . . . 3
1.2 Die Höhe und das Maximum eines Heaps . . . 4

2 Effiziente Darstellung **5**
2.1 Die Array-Repräsentation . . . 5
2.2 Das Rechnen mit Indizes . . . 6

3 Grundlegende Operationen **7**
3.1 Einfügen eines zusätzlichen Elements . . . 7
3.2 Entfernen des aktuellen Maximums . . . 8
3.3 Erstellen eines neuen Heaps . . . 9
3.4 Genauere Laufzeitanalyse . . . 10

4 Anwendung: Effizientes Sortieren **11**

5 Kontrollaufgaben **13**

6 Lösungen **15**

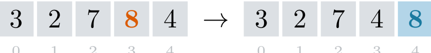
1 Motivation und Eigenschaften

Weil eines unserer Hauptziele die Verbesserung des Sortierverfahrens Selectionsort ist, wollen wir uns als Erstes die Funktionsweise dieses Algorithmus ganz kurz ins Gedächtnis zurückrufen. Der Algorithmus startet mit einem Array von bunt durchmischten natürlichen Zahlen, wie beispielsweise folgender Sequenz:

3	2	7	8	4
0	1	2	3	4

In der Abbildung oben sind die eigentlichen Einträge im Array – also die Elemente, die es jetzt zu sortieren gilt – durch die schwarzen Zahlen in den grauen Kästchen repräsentiert. In kleiner und grauer Schrift sind zusätzlich die jeweiligen Indizes der Einträge mit angegeben.

Der Algorithmus schaut sich nacheinander alle fünf Einträge im Array an und merkt sich in einer Variablen den Index des Eintrags mit der grössten Zahl, welcher er dabei begegnet. In unserem konkreten Beispiel oben wäre das also die Zahl 8 beim Eintrag mit Index 3. Diese gefundene grösste Zahl wird anschliessend mit dem letzten Eintrag vertauscht, also mit der Zahl 4 beim Eintrag mit Index 4:



Mit fetter und roter Schrift heben wir oben links diese grösste gefundene Zahl 8 hervor. Das blaue Kästchen oben rechts deutet hingegen an, dass die Zahl 8 jetzt an der richtigen Stelle steht. Das stimmt tatsächlich, weil 8 als insgesamt grösste Zahl in der aufsteigend sortierten Reihenfolge natürlich erst ganz am Schluss kommt.

Der Algorithmus wiederholt jetzt einfach die gleiche Prozedur, bis das Array sortiert ist. Das heisst, er sucht sich die grösste verbleibende Zahl, indem er sich von Neuem alle grauen Kästchen nacheinander anschaut, und vertauscht dann diese grösste verbleibende Zahl mit dem Eintrag beim letzten grauen Kästchen, welches dann auch blau eingefärbt wird und so weiter:

Aufgabe 1.1

Erinnern Sie sich kurz an die Laufzeitanalyse von Selectionsort. Geben Sie insbesondere an,

- wie oft im schlimmsten Falle zwei Einträge im Array vertauscht werden und
- wie oft sich der Algorithmus insgesamt eines der grauen Kästchen anschaut.

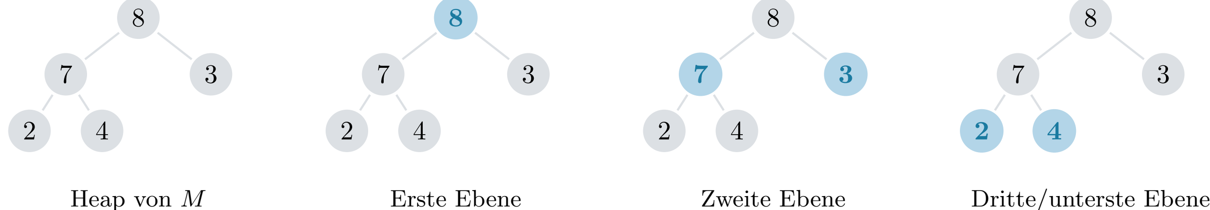
Überlegen Sie sich dann, welcher dieser zwei Punkte mit einer Datenstruktur, so wie sie ganz am Anfang auf der allerersten Seite dieses Dokuments beschrieben wurde, beschleunigt werden könnte.

Lösung auf Seite 15

1.1 Die Heap-Eigenschaft

Die Heap-Datenstruktur verwaltet eine Menge M von Elementen. Bei diesen Elementen kann es sich im Prinzip um beliebige Objekte handeln, solange eine klare Ordnung auf ihnen definiert ist – also solange man sie auf sinnvolle Weise sortieren kann. Der Einfachheit halber nehmen wir aber an, dass es sich bei M immer um eine Menge von natürlichen Zahlen wie zum Beispiel $M = \{3, 2, 7, 8, 4\}$ handelt.

Diese Elemente oder Zahlen werden jetzt in den Knoten eines binären Baumes auf eine sehr spezielle Art und Weise angeordnet. Das könnte zum Beispiel wie in folgender Abbildung ganz links aussehen:



Heap von M | Erste Ebene | Zweite Ebene | Dritte/unterste Ebene

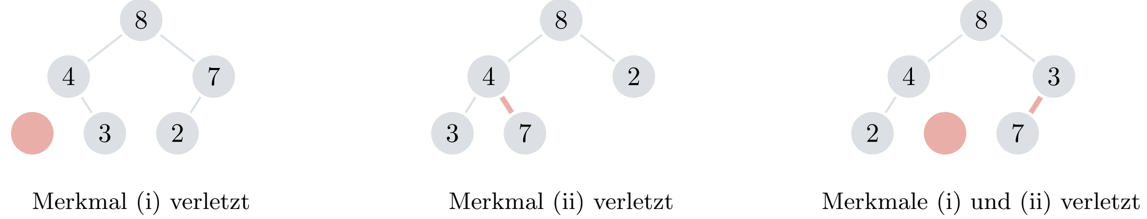
Folgende zwei Merkmale, die wir zusammen als die *Heap-Eigenschaft* bezeichnen wollen, müssen erfüllt sein:

(i) Jede Ebene (ausser der untersten) ist vollständig von links bis rechts mit Knoten besetzt. Die unterste Ebene hingegen ist linksbündig mit den restlichen Knoten aufgefüllt.

(ii) Jeder Knoten (ausser der Wurzel) enthält ein kleineres Element als sein Eltern-Knoten.

Fortan soll ein binärer Baum mit diesen zwei Merkmalen als *Heap von M* oder kürzer, falls die Menge M schon aus dem Kontext gegeben ist, als *Heap* bezeichnet werden.

In den folgenden Illustrationen sehen Sie zusätzlich noch je ein Beispiel für binäre Bäume, die entweder das erste oder das zweite oder beide Merkmale zugleich verletzen. Bei keinem dieser Beispiele handelt es sich also um einen Heap gemäss unserer Definition. Die fehlenden Knoten, deren Absenz Merkmal (i) verletzt, sowie die Kanten, entlang welcher Merkmal (ii) verletzt wird, sind jeweils rot eingezeichnet.

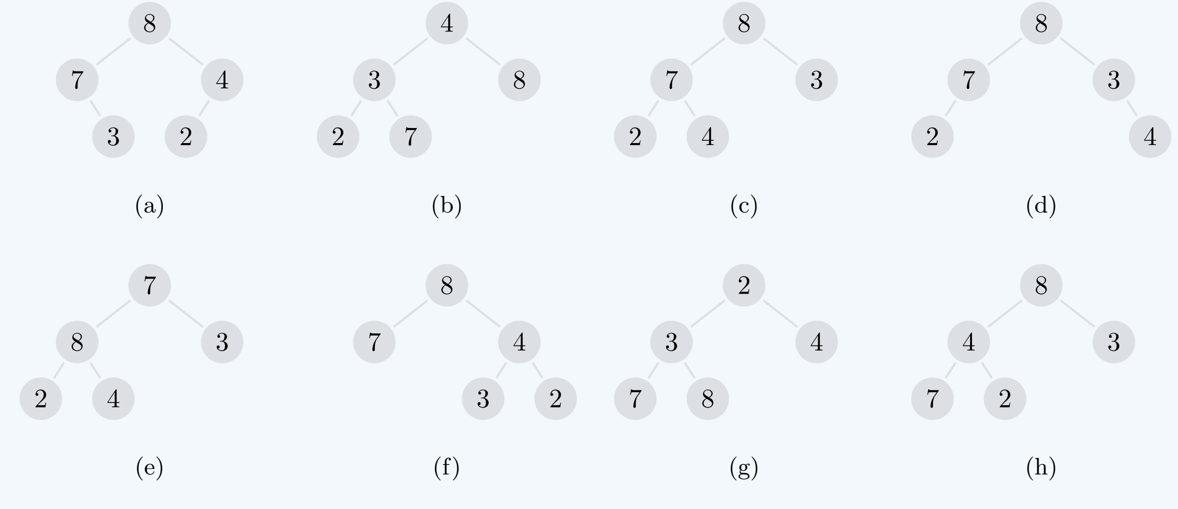


Merkmal (i) verletzt | Merkmal (ii) verletzt | Merkmale (i) und (ii) verletzt

Bitte nutzen Sie die folgenden zwei Aufgaben, um die soeben gelernte Definition der Heap-Eigenschaft zu verinnerlichen und um mögliche Missverständnisse gleich jetzt aufzudecken.

Aufgabe 1.2

Entscheiden Sie, welche der folgenden binären Bäume die Heap-Eigenschaft erfüllen. Bei Nichterfüllung geben Sie bitte zusätzlich an, welches der beiden Merkmale verletzt ist und an welcher Stelle dies passiert.



Lösung auf Seite 15

Aufgabe 1.3

Sei $M = \{3, 2, 7, 8, 4\}$ wie zuvor. Zeichnen Sie alle möglichen binären Bäume mit Knotenmenge M, so dass die Heap-Eigenschaft erfüllt ist. In anderen Worten, bestimmen Sie alle Heaps von M. Wie viele solche Heaps gibt es?

Lösung auf Seite 16

1.2 Die Höhe und das Maximum eines Heaps

Folgende zwei Aufgaben dienen dazu, zwei weitere Eigenschaften herzuleiten, die von jedem Heap erfüllt sind. Auf diese zwei Eigenschaften werden wir in den kommenden Abschnitten immer wieder zurückgreifen.

Aufgabe 1.4

Sei M eine beliebige Menge von natürlichen Zahlen. Stellen Sie sich einen zugehörigen Heap vor. Was ist dann die Höhe dieses Heaps in Abhängigkeit der Anzahl n von Elementen in M? In anderen Worten, wie viele Ebenen gibt es insgesamt? Argumentieren Sie mit Hilfe von Merkmal (i) der Heap-Eigenschaft, dass Ihre Antwort korrekt ist.

Lösung auf Seite 16

Aufgabe 1.5

Sei M eine beliebige Menge von natürlichen Zahlen. Stellen Sie sich einen zugehörigen Heap vor. In welchem Knoten des Heaps findet man dann das Maximum – also die grösste Zahl von M? Argumentieren Sie mit Hilfe von Merkmal (ii) der Heap-Eigenschaft, dass Ihre Antwort korrekt ist.

Lösung auf Seite 17

Zusammenfassung

Bei einem *Heap* von einer Menge M von n natürlichen Zahlen handelt es sich um einen binären Baum mit Knotenmenge M, der die so genannte *Heap-Eigenschaft* erfüllt, welche aus zwei Merkmalen besteht:

(i) Das erste Merkmal besagt, dass alle Ebenen des Baumes bis auf die letzte vollständig mit Knoten besetzt sind, und sorgt dafür, dass der Baum eine Höhe von höchstens $O(\log n)$ hat.

(ii) Das zweite Merkmal besagt, dass alle Knoten kleiner sind als ihre Eltern-Knoten, und sorgt dafür, dass das Maximum von M immer in der Wurzel des Baumes zu finden ist.

2 Effiziente Darstellung

Wir überlegen uns in diesem Abschnitt, wie man einen Heap möglichst platzsparend in einem Computerprogramm darstellen kann. So, wie man das typischerweise auch bei binären Suchbäumen macht, könnte man natürlich für jeden Knoten neben seinem eigentlichen Element zwei zusätzliche Referenzen abspeichern, die auf das linke und rechte Kind verweisen. Dieses Vorgehen scheint auf den ersten Blick vielleicht sogar optimal zu sein, denn wir wollen ja wissen, wie man von einem Knoten zu seinen Kindern kommt.

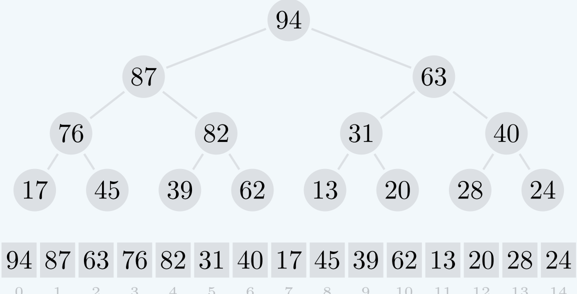
2.1 Die Array-Repräsentation

Um von einem gegebenen Knoten effizient zu seinen Kindern zu gelangen, sind wir jedoch interessanterweise nicht auf zwei zusätzliche Referenzen angewiesen. Das Merkmal (i) der Heap-Eigenschaft erlaubt es uns, die Elemente der Menge M auf die Einträge eines Arrays so zu verteilen, dass für jeden Eintrag nach einer kurzen Rechnung klar sein wird, wo sich die Einträge der entsprechenden Kind-Knoten befinden.

Diese Art, einen Heap mit Hilfe eines Arrays darzustellen, nennen wir *Array-Repräsentation*. In der nächsten Aufgabe finden Sie anhand eines konkreten Beispiels selbständig heraus, wie dieses Array aufgebaut ist.

Aufgabe 2.1

Es folgt ein Beispiel für einen Heap und die zugehörige Array-Repräsentation. Können Sie ein Muster erkennen? Gibt es ein System, mit welchem die Knoten auf die Einträge im Array verteilt worden sind?

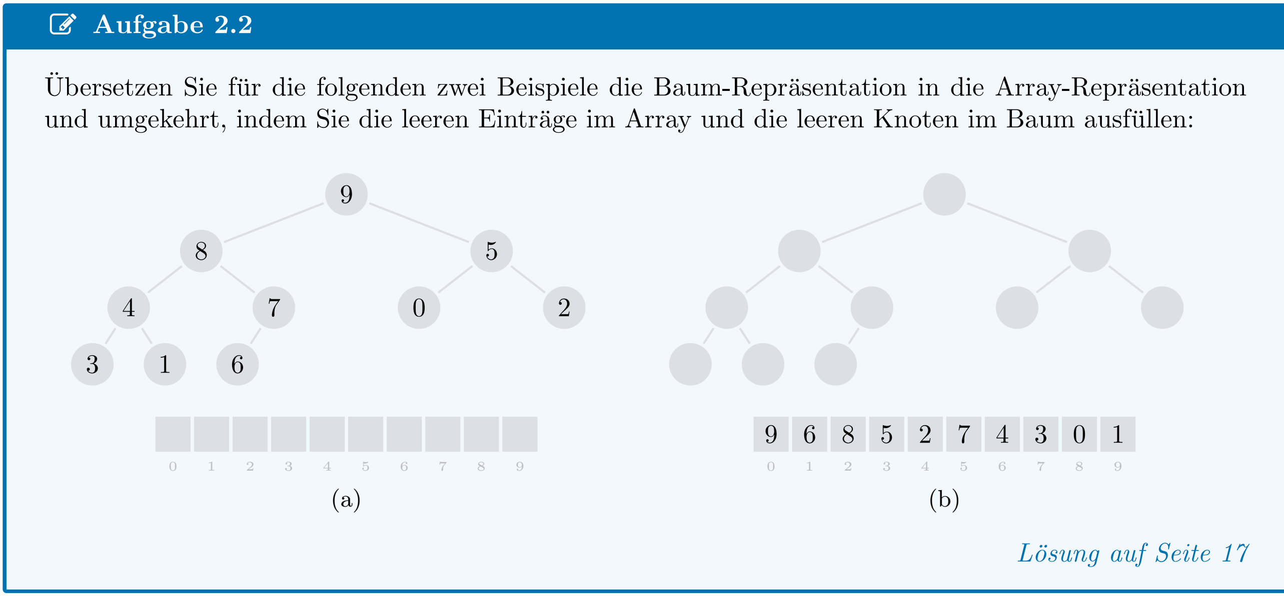


Lösung auf Seite 17

Die Array-Repräsentation ist also ein System, mit dem ein Heap mit n Knoten von einem Array mit n Einträgen repräsentiert wird. Umgekehrt gibt es für ein Array mit n Einträgen nur eine mögliche Gestalt, die der zugehörige Heap haben kann, in der sogleich alle Einträge auf eindeutige Weise auf die Knoten verteilt werden können. Man kann also beliebig zwischen den beiden Darstellungen hin und her wechseln.

Aufgabe 2.2

Übersetzen Sie für die folgenden zwei Beispiele die Baum-Repräsentation in die Array-Repräsentation und umgekehrt, indem Sie die leeren Einträge im Array und die leeren Knoten im Baum ausfüllen:

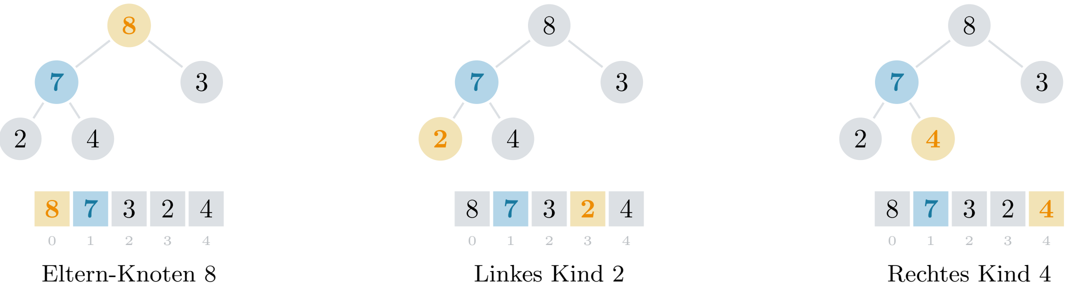


(a) (b)

Lösung auf Seite 17

2.2 Das Rechnen mit Indizes

Was uns jetzt noch fehlt, ist die Fähigkeit, für einen gegebenen Eintrag in der Array-Repräsentation möglichst schnell seine Eltern und Kinder zu finden. Fokussieren Sie sich zu diesem Zweck einmal auf den blau eingefärbten Knoten 7 in folgendem Heap und seine jeweils gelb eingefärbten direkten Vorgänger und Nachfolger:



Eltern-Knoten 8 Linkes Kind 2 Rechtes Kind 4

Für den Knoten mit Index 1 in der Array-Repräsentation scheint es also so zu sein, dass sein Eltern-Knoten den Index 0 hat, dass sein linkes Kind den Index 3 hat und dass sein rechtes Kind den Index 4 hat. Dies mag auf den ersten Blick willkürlich erscheinen, aber es steckt wieder ein System dahinter.

Aufgabe 2.3

Stellen Sie sich einen Knoten eines Heaps vor, der den Index i in der Array-Repräsentation hat. Finden Sie drei allgemein gültige Formeln in Abhängigkeit von i, mit denen man dann den Index des Eltern-Knotens, des linken Kindes und des rechten Kindes berechnen kann.

Lösung auf Seite 18

Aufgabe 2.4

Schreiben Sie eine Funktion, die für eine gegebene Array-Repräsentation die Heap-Eigenschaft überprüft:

```
fn heap_check( array: & [u64] ) -> bool { ... }
```

Lösung auf Seite 19

Zusammenfassung

Die *Array-Repräsentation* eines Heaps ist ein effizientes System, um einen Heap ohne zusätzlichen Speicherbedarf mit einem Array darzustellen. Wenn ein Knoten dem Index i in diesem Array entspricht, dann kann man ganz einfach den Index seiner direkten Vorgänger und Nachkommen berechnen:

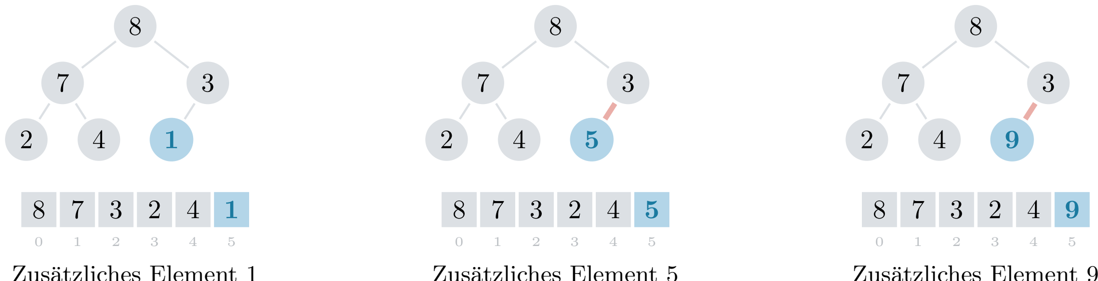
- Der Eltern-Knoten hat den Index $\lfloor (i-1)/2 \rfloor$, falls er existiert.
- Der linke Kind-Knoten hat den Index $2i+1$, falls er existiert.
- Der rechte Kind-Knoten hat den Index $2i+2$, falls er existiert.

3 Grundlegende Operationen

Bis jetzt haben wir Heaps nur als starre Objekte kennengelernt – das heisst als Objekte, die unveränderlich sind und sich nicht dynamisch mit der Zeit wandeln können. Genau dies soll sich in diesem Abschnitt ändern.

3.1 Einfügen eines zusätzlichen Elements

Wir nehmen an, dass der bekannte Heap von $M = \{3, 2, 7, 8, 4\}$ in Array-Repräsentation gegeben ist und dass wir ein zusätzliches Element – wie zum Beispiel 1, 5 oder 9 – einfügen möchten. Ein erster Versuch könnte sein, das Array um einen Eintrag zu vergrössern und das zusätzliche Element genau dort zu positionieren:



Zusätzliches Element 1 Zusätzliches Element 5 Zusätzliches Element 9

Wie Sie sehen, haben wir bei dem zusätzlichen Element 1 Glück und der neue binäre Baum erfüllt bereits die Heap-Eigenschaft. Allerdings sehen Sie bei 5 und 9 auch, dass dieses Vorgehen nicht immer zum Erfolg führt, weil für den zusätzlichen Knoten das Merkmal (ii) der Heap-Eigenschaft jetzt verletzt ist.

Aufgabe 3.1

Betrachten Sie die binären Bäume mit den zusätzlichen Elementen 5 und 9 in der Abbildung oben. Was passiert jeweils, wenn die mit der roten Kante verbundenen Knoten vertauscht werden? Kann auf diese Weise – also durch das wiederholte Vertauschen von benachbarten Knoten entlang einer roten Kante – ein Zustand erreicht werden, in dem die Heap-Eigenschaft erfüllt ist?

Lösung auf Seite 20

Den in der Lösung der Aufgabe oben beschriebenen Vorgang – also das wiederholte Vertauschen des zusätzlichen Knotens mit seinem jeweiligen Eltern-Knoten entlang einer roten Kante – werden wir auf anschauliche Weise mit dem Begriff *Aufsteigen* bezeichnen.

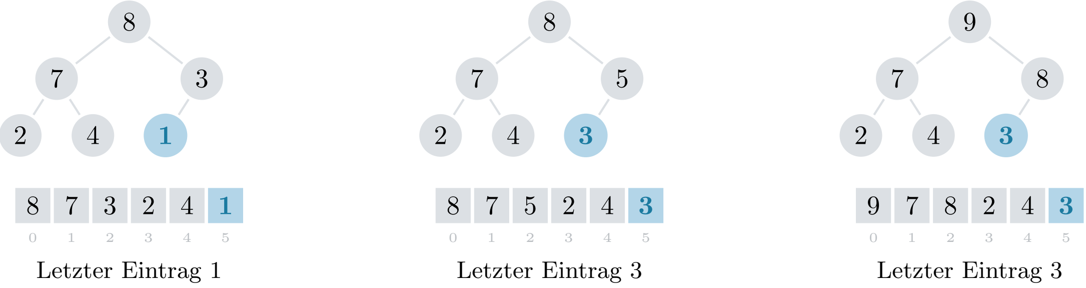
In den zwei betrachteten Situationen hat ebendieses Aufsteigen jeweils zum Erfolg geführt. Es ist aber noch nicht klar, dass der gleiche Vorgang auch in allen anderen erdenklichen Situationen erfolgreich sein wird. Die nächste Aufgabe dient dazu, uns genau davon zu überzeugen.

> **Aufgabe 3.2**
>
> Erklären Sie, wieso es während des Aufsteigens eines zusätzlichen Knotens in einem Heap zu jedem Zeitpunkt maximal eine rote Kante gibt und wo genau diese Kante sich jeweils befindet. Folgern Sie mit Hilfe von Aufgabe 1.4, dass nach höchstens $O(\log n)$ Vertauschungen die Heap-Eigenschaft erfüllt ist.
>
> *Lösung auf Seite 21*

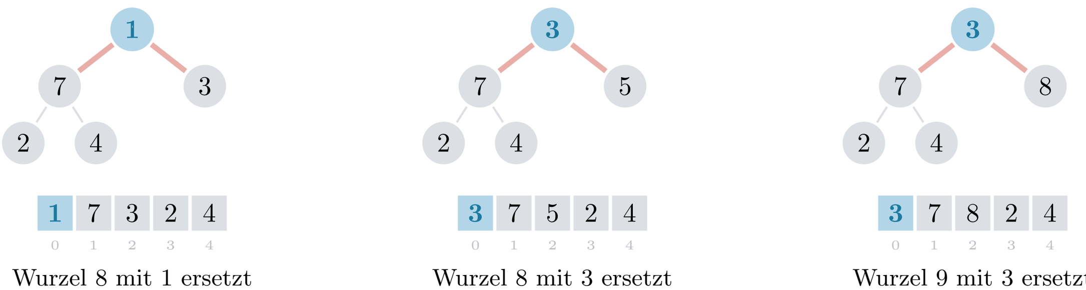
> **Aufgabe 3.3**
>
> Schreiben Sie eine Funktion, die auf der gegebenen Array-Repräsentation eines Heaps – mit einem zusätzlichen Element beim letzten Eintrag – den Vorgang des Aufsteigens ausführt:
>
> ```
> fn heap_ascend( array: & mut [u64] ) { ... }
> ```
>
> *Lösung auf Seite 22*

3.2 Entfernen des aktuellen Maximums

Als Nächstes möchten wir gerne das aktuelle Maximum eines Heaps entfernen. Wir betrachten zu diesem Zweck folgende drei Heaps, die wir schon im Zusammenhang mit Aufgabe 3.1 kennengelernt haben. Der jeweils letzte Eintrag im Array wird dabei eine besondere Rolle spielen und ist deshalb bereits blau eingefärbt:



Von Aufgabe 1.5 wissen wir, dass das zu entfernende Maximum immer in der Wurzel zu finden ist. Ein einfacher erster Versuch könnte also sein, den letzten Eintrag im Array an die erste Stelle zu verschieben – das heisst also, dass wir die Wurzel des Heaps mit ebendiesem letzten Eintrag ersetzen – und dann die Länge des Arrays um einen Eintrag zu verkürzen:



Wie Sie sehen, gibt uns das im Allgemeinen aber keinen Heap, weil das Merkmal (ii) der Heap-Eigenschaft wieder verletzt ist. Im Gegensatz zum letzten Abschnitt, in dem es um das Einfügen eines zusätzlichen Elements ging, gibt es jetzt sogar gleichzeitig zwei rote Kanten anstatt nur eine. Wenn wir mit einer ähnlichen Strategie wie zuvor entlang einer roten Kante zwei Knoten vertauschen wollten, dann wäre es also nicht einmal klar, welche dieser beiden roten Kanten wir wählen sollten.

Aufgabe 3.4

Betrachten Sie die drei binären Bäume mit je zwei roten Kanten in vorhergehender Abbildung. Entlang welcher dieser roten Kanten würden Sie zwei Knoten vertauschen? Begründen Sie ihre Wahl. Kann auf diese Weise – also durch das wiederholte Vertauschen von benachbarten Knoten entlang einer sorgfältig ausgewählten roten Kante – ein Zustand erreicht werden, in dem die Heap-Eigenschaft erfüllt ist?

Lösung auf Seite 23

Den in der Lösung der Aufgabe oben beschriebenen Vorgang – also das wiederholte Vertauschen des blauen Knotens mit dem grösseren seiner zwei Kinder entlang einer roten Kante – werden wir, in Anlehnung an den Begriff des Aufsteigens, mit dem Begriff *Absickern* bezeichnen.

Genau wie beim Aufsteigen geht es jetzt darum, uns zu überzeugen, dass dieser Vorgang immer zum Erfolg führt und nicht nur in den drei vorher betrachteten speziellen Situationen.

Aufgabe 3.5

Erklären Sie, wieso es während des Absickerns eines Knotens zu jedem Zeitpunkt maximal zwei rote Kanten gibt und wo genau diese Kanten sich jeweils befinden. Folgern Sie mit Hilfe von Aufgabe 1.4, dass nach höchstens $O(\log n)$ Vertauschungen die Heap-Eigenschaft erfüllt ist.

Lösung auf Seite 24

Aufgabe 3.6

Schreiben Sie eine Funktion, die auf der gegebenen Array-Repräsentation eines Heaps – mit einem deplatzierten Element beim ersten Eintrag – den Vorgang des Absickerns ausführt:

```
fn heap_descend( array: & mut [u64] ) { ... }
```

Lösung auf Seite 25

3.3 Erstellen eines neuen Heaps

Wir sind jetzt in der Lage, einen Heap auf dynamische Weise zu verwenden. Wir können jederzeit neue Elemente effizient hinzufügen und das aktuelle Maximum effizient entfernen – und zwar so, dass die Heap-Eigenschaft stets erfüllt bleibt.

Doch was tun wir, wenn wir eine ganze Menge M von Elementen auf einen Schlag bekommen? Wie konstruieren wir dann am besten einen entsprechenden Heap? Interessanterweise haben wir mit den beiden beschriebenen Vorgängen des Aufsteigens und des Absickerns zwei Werkzeuge in der Hand, mit denen wir dieses Problem auf zwei unterschiedliche Arten bereits lösen können. Überlegen Sie es sich selbst in folgender Aufgabe.

Aufgabe 3.7

Sei M eine beliebige Menge von n natürlichen Zahlen. Sei weiter A ein gegebenes Array mit Einträgen aus der Menge M in beliebiger Reihenfolge. Überlegen Sie sich zwei unterschiedliche Algorithmen, die durch das wiederholte Vertauschen von Einträgen in A das Array so verändern, dass die Heap-Eigenschaft am Schluss erfüllt ist – also so, dass A die Array-Repräsentation eines Heaps von der Menge M ist.

1. Der erste Algorithmus soll n mal den Vorgang des Aufsteigens auf Teilbäumen ausführen.
2. Der zweite Algorithmus soll n mal den Vorgang des Absickerns auf Teilbäumen ausführen.

Argumentieren Sie für beide Fälle, dass höchstens $O(n \log n)$ mal zwei Einträge vertauscht werden.

Lösung auf Seite 26

Bei den zwei beschriebenen Algorithmen der vorhergehenden Aufgabe verwenden wir die Vorgänge des Aufsteigens und des Absickerns auf eine leicht andere Weise als bisher. Die Knoten, auf welchen die Vorgänge ausgeführt werden, befinden sich zu Beginn nämlich nicht mehr unbedingt beim letzten oder beim ersten Eintrag des Arrays. Diese Einträge können jetzt einen beliebigen Index haben.

Bevor wir diese zwei Algorithmen implementieren können, müssen wir deshalb zuerst unseren bereits existierenden Code für das Aufsteigen und das Absickern derart anpassen, dass beide Vorgänge auch bei beliebigem Index starten können.

Aufgabe 3.8

Überarbeiten Sie Ihren Code der Funktionen `heap_ascend` und `heap_descend`, so dass der entsprechende Vorgang bei beliebigem Index i gestartet werden kann. Implementieren Sie also folgende Funktionen:

```
fn heap_ascend_at_index( array: & mut [u64], i: usize ) { ... }
fn heap_descend_at_index( array: & mut [u64], i: usize ) { ... }
```

Lösung auf Seite 27

Aufgabe 3.9

Basierend auf Ihrer Lösung der vorhergehenden Aufgabe schreiben Sie jetzt zwei weitere Funktionen, welche die beiden Algorithmen von Aufgabe 3.7 implementieren:

```
fn heap_create_with_ascend( array: & mut [u64] ) { ... }
fn heap_create_with_descend( array: & mut [u64] ) { ... }
```

Lösung auf Seite 28

3.4 Genauere Laufzeitanalyse

Wir haben jetzt also zwei Algorithmen, die das gleiche Problem – das Erstellen eines neuen Heaps – auf sehr ähnliche Art und Weise lösen. Beide Methoden sind auch ziemlich schnell – höchstens $O(n \log n)$ Vertauschungen für n Elemente, wie wir in Aufgabe 3.7 bereits gesehen haben. Trotzdem sollten wir uns die Frage stellen, welchen der beiden Algorithmen wir in der Praxis tatsächlich verwenden wollen.

Aufgabe 3.10

Sei M eine Menge von n natürlichen Zahlen. Nehmen Sie der Einfachheit halber an, dass es sich dabei um die konkrete Menge $M = \{1, 2, 3, \ldots, n\}$ handelt. Konstruieren Sie jetzt ein Array A mit Einträgen aus der Menge M, so dass beide Algorithmen bei jedem Zwischenschritt – also beim Aufsteigen oder Absickern jedes Knotens – die grösstmögliche Anzahl von Vertauschungen von Einträgen durchführen.

Lösung auf Seite 29

Aufgabe 3.11

Nehmen Sie an, dass $n = 2^h - 1$ gilt für eine natürliche Zahl h. Das bedeutet insbesondere, dass jeder Heap von der Menge $M = \{1, 2, 3, \ldots, n\}$ genau h vollständig besetzte Ebenen hat – dass es sich also um einen vollständigen binären Baum der Höhe h handelt. Sei weiter A das Array aus der Aufgabe zuvor. Finden Sie für beide Algorithmen je eine Formel in Abhängigkeit von h, welche die Anzahl Vertauschungen beschreibt für den Fall, dass der entsprechende Algorithmus auf das Array A angewendet wird.

Lösung auf Seite 29

Die zwei in Aufgabe 3.11 berechneten Formeln für die Anzahl von Vertauschungen sind die Schlüssel zur Beantwortung der Frage, welchen der beiden Algorithmen wir bevorzugen sollten. Es wird sich herausstellen, dass eine der beiden Formeln eine signifikant kleinere Zahl beschreibt und dass der entsprechende Algorithmus deshalb viel effizienter ist als der andere. Unsere letzte Herausforderung in diesem Abschnitt ist es, die kleinere dieser zwei Zahlen zu identifizieren.

Bevor Sie weiter lesen, versuchen Sie sich doch zuerst selbständig zu überlegen, welcher der beiden Algorithmen der effizientere sein könnte. Ist für die meisten Startpositionen Aufsteigen oder Absickern besser? Oder anders gefragt, befinden sich in einem Heap die meisten Knoten eher auf einer weiter oberen Ebene in der Nähe der Wurzel oder eher auf einer weiter unteren Ebene in der Nähe der Blätter?

Aufgabe 3.12

Versuchen Sie zu zeigen, dass die berechnete Formel für die Anzahl von Vertauschungen beim Erstellen eines Heaps durch Aufsteigen eine Zahl beschreibt, die *mindestens* so gross wie $\frac{n}{2}\log_2(\frac{n}{2})$ ist.

Lösung auf Seite 29

Aufgabe 3.13

Versuchen Sie zu zeigen, dass die berechnete Formel für die Anzahl von Vertauschungen beim Erstellen eines Heaps durch Absickern eine Zahl beschreibt, die *höchstens* so gross wie $n + 1$ ist. Die Verwendung der Gleichung $\sum_{j=0}^{\infty} \frac{j}{2^j} = 2$ könnte sich dabei als hilfreich herausstellen.

Lösung auf Seite 29

Zusammenfassung

Basierend auf den beiden Vorgängen *Aufsteigen* und *Absickern* können folgende drei grundlegende Operationen auf der Array-Repräsentation eines Heaps mit n Elementen realisiert werden:

- Nach dem Einfügen eines zusätzlichen Elements kann durch $O(\log n)$ Vertauschungen die Heap-Eigenschaft wiederhergestellt werden.
- Nach dem Entfernen des aktuellen Maximums kann durch $O(\log n)$ Vertauschungen die Heap-Eigenschaft wiederhergestellt werden.
- Ein beliebig angeordnetes Array kann durch $O(n)$ Vertauschungen so präpariert werden, dass die Heap-Eigenschaft am Schluss erfüllt ist.

4 Anwendung: Effizientes Sortieren

Wir sind soweit und können endlich die Früchte unserer harten Arbeit ernten. Denken Sie an den Anfang dieser Unterrichtsunterlagen und insbesondere an Aufgabe 1.1 zurück. Wir hatten uns damals überlegt, wie das Sortierverfahren Selectionsort mit Hilfe einer Heap-Datenstruktur beschleunigt werden kann. Das daraus resultierende Sortierverfahren *Heapsort*, das wir nachfolgend implementieren wollen, hat neben seiner Effizienz aber noch eine weitere sehr interessante Eigenschaft. Es besitzt die Fähigkeit, ein gegebenes Array mit n Einträgen *in situ* (lateinisch für *am Ort*) zu sortieren. Das bedeutet, dass der zusätzliche Speicherplatzbedarf neben dem schon gegebenen Array durch eine Konstante abgeschätzt werden kann – also insbesondere so, dass der zusätzliche Speicherplatzbedarf in keiner Weise von der Eingabegrösse n abhängt.

Aus den Aufgaben 3.7 und 3.13 wissen wir bereits, wie ein Heap mit höchstens $O(n)$ Vertauschungen und ohne zusätzlichen Speicherplatzbedarf auf einem gegebenen Array mit n Einträgen aufgebaut wird. Die Funktionsweise von Heapsort ist nun so, dass in einem zweiten Schritt dieser Heap mit höchstens $O(n \log n)$ zusätzlichen Vertauschungen und wieder ohne zusätzlichen Speicherplatzbedarf in ein sortiertes Array umgewandelt wird.

Aufgabe 4.1

Betrachten Sie folgendes Array mit fünf Einträgen, welches Sie bereits vom allerersten Abschnitt dieser Unterrichtsunterlagen kennen:

3	2	7	8	4
0	1	2	3	4

1. Erstellen Sie von Hand und mit Hilfe von Absickern zuerst die Array-Repräsentation eines entsprechenden Heaps.

2. Entfernen Sie dann wiederholt das aktuelle Maximum, indem Sie es mit dem letzten Knoten vertauschen und dann die Heap-Eigenschaft auf dem kleiner werdenden Heap wiederherstellen.

Diese zwei Schritte führen dazu, dass das Array am Schluss sortiert ist. Zeichnen Sie bitte alle Zwischenschritte auf – das heisst, immer dann, wenn Sie zwei Einträge im Array vertauschen.

Lösung auf Seite 30

Bei der Prozedur bestehend aus den zwei Teilen, die Sie in der vorhergehenden Aufgabe ausgeführt haben, handelt es sich um nichts anderes als Heapsort. Da ausser dem Vertauschen von Einträgen im Array im Wesentlichen nichts anderes passiert, ist auch klar, dass diese Prozedur in situ arbeitet.

In den folgenden abschliessenden Programmieraufgaben implementieren Sie zuerst den noch fehlenden zweiten Teil von Aufgabe 4.1. Zusammen mit dem schon bestehenden Code für den ersten Teil lässt sich danach ganz einfach eine vollständige Implementierung für Heapsort angeben.

Aufgabe 4.2

Schreiben Sie eine Funktion, welche die gegebene Array-Repräsentation eines Heaps von natürlichen Zahlen in ein sortiertes Array umwandelt, indem die Zahlen der Grösse nach aus dem gegebenen Heap entfernt werden und das Array mit diesen Zahlen gleichzeitig von hinten wieder aufgefüllt wird:

```
fn heap_arrange_inorder( array: & mut [u64] ) { ... }
```

Lösung auf Seite 31

Aufgabe 4.3

Schreiben Sie eine letzte Funktion, die ein gegebenes Array von natürlichen Zahlen sortiert, indem sie zuerst die Array-Repräsentation eines entsprechenden Heaps konstruiert und danach die Zahlen der Grösse nach aus ebendiesem Heap entfernt:

```
fn heap_sort( array: & mut [u64] ) { ... }
```

Lösung auf Seite 32

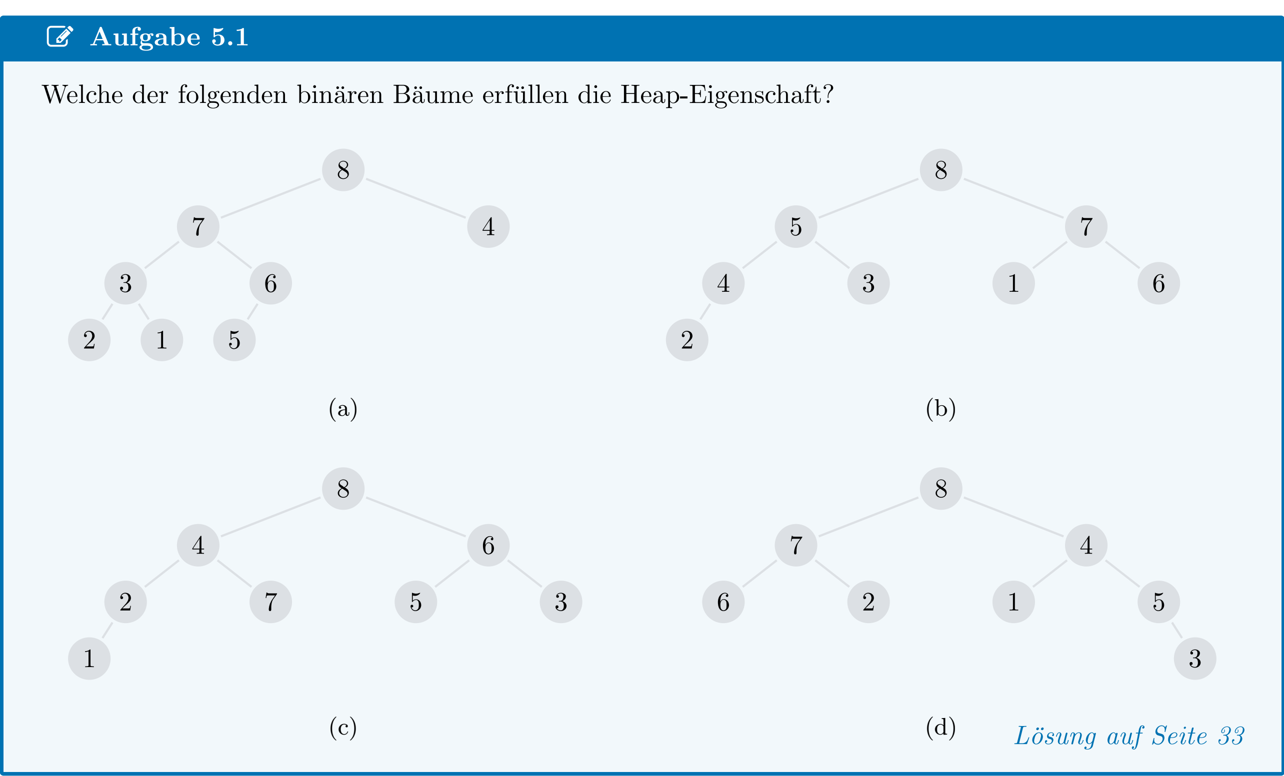
Zusammenfassung

Das Sortierverfahren *Heapsort* ist in der Lage, ein Array mit n Einträgen mit höchstens $O(n \log n)$ Vertauschungen *in situ* zu sortieren – also so, dass kein zusätzlicher Speicherplatz verwendet wird.

5 Kontrollaufgaben

Aufgabe 5.1

Welche der folgenden binären Bäume erfüllen die Heap-Eigenschaft?



Lösung auf Seite 33

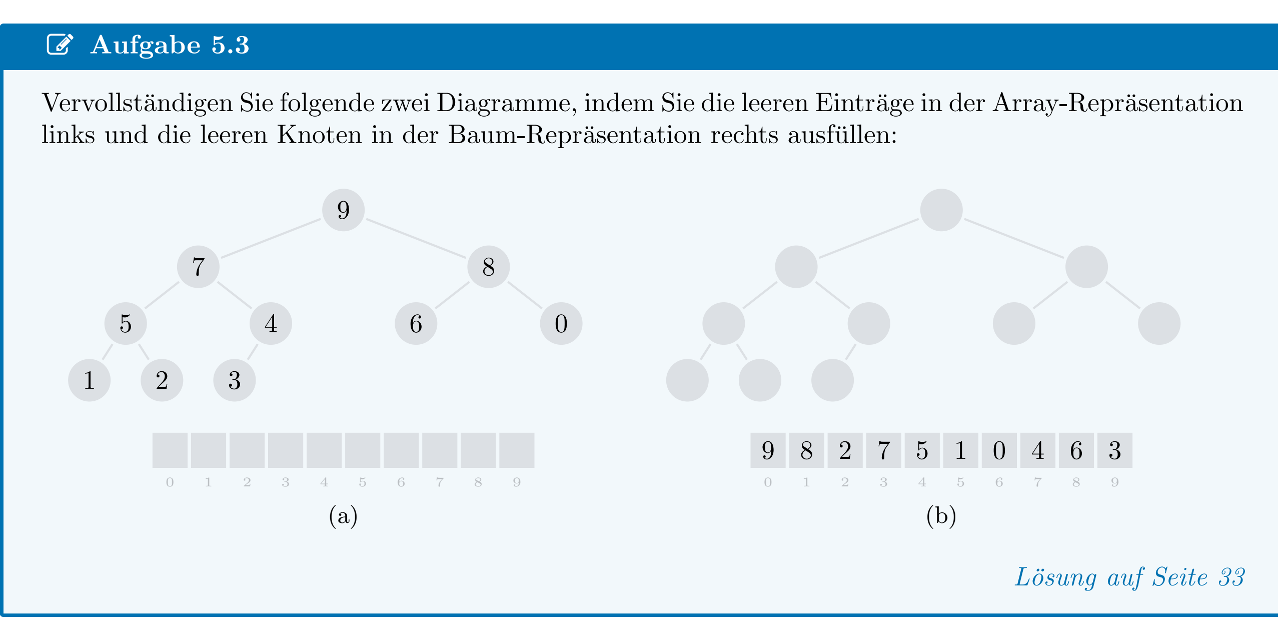
Aufgabe 5.2

Der Heap – so wie er in diesen Unterrichtsunterlagen vorgestellt wurde – gibt uns die Möglichkeit, für eine dynamisch veränderbare Menge M von natürlichen Zahlen zu jedem Zeitpunkt effizient das Maximum von M abrufen zu können. Was wäre jetzt aber, wenn wir stattdessen am Minimum von M interessiert wären? Geht das direkt oder müssten wir dafür zuerst gewisse Anpassungen vornehmen?

Lösung auf Seite 33

Aufgabe 5.3

Vervollständigen Sie folgende zwei Diagramme, indem Sie die leeren Einträge in der Array-Repräsentation links und die leeren Knoten in der Baum-Repräsentation rechts ausfüllen:



Lösung auf Seite 33

Aufgabe 5.4

Stellen Sie sich einen Heap mit $n = 8$ Knoten vor. Sei nun i einer der Indizes 0, 2, 3 oder 8. Berechnen Sie dann jeweils den Index (a) des Eltern-Knotens, (b) des linken Kindes und (c) des rechten Kindes von dem Knoten mit Index i – oder erklären Sie, wieso diese Frage vielleicht gar keinen Sinn macht.

Lösung auf Seite 34

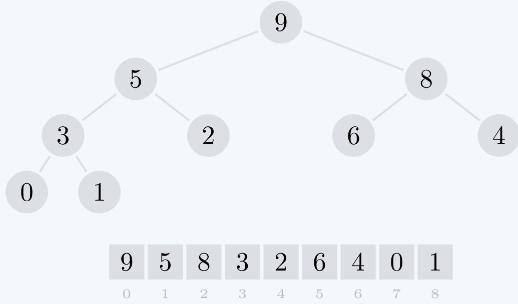
Aufgabe 5.5

Erklären Sie auf intuitive Weise, welcher der zwei vorgestellten Vorgänge – Aufsteigen oder Absickern – für das Erstellen eines neuen Heaps der effizientere ist.

Lösung auf Seite 34

Aufgabe 5.6

Fügen Sie bei folgendem Heap zuerst mit Hilfe von Aufsteigen das zusätzliche Element 7 ein und entfernen Sie dann – in einem darauf folgenden Schritt – mit Hilfe von Absickern das aktuelle Maximum in der Wurzel des Heaps. Zeichnen Sie bitte alle Zwischenschritte auf.



Lösung auf Seite 35

6 Lösungen

Lösung zu Aufgabe 1.1 auf Seite 3

Sei n die Anzahl Einträge in dem zu sortierenden Array. Beim konkreten Beispiel vor der Aufgabe hatten wir also $n = 5$. Im Allgemeinen gilt dann, dass der Algorithmus

- im schlimmsten Falle n mal zwei Einträge vertauscht und
- genau $n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = O(n^2)$ mal ein graues Kästchen anschaut.

Man könnte, wenn man es genau nimmt, bei beiden Werten noch 1 abziehen. Denn beim letzten Schritt des Algorithmus – wenn also alle Kästchen bis auf eines bereits blau sind – gibt es eigentlich nichts mehr zu tun, weil dann die Zahl beim ersten Eintrag auch schon an der richtigen Stelle steht.

Beim ersten Punkt von oben können wir keine essentielle Beschleunigung erwarten. Es kann ja durchaus sein, dass alle Zahlen in dem zu sortierenden Array am Anfang an der falschen Stelle stehen.

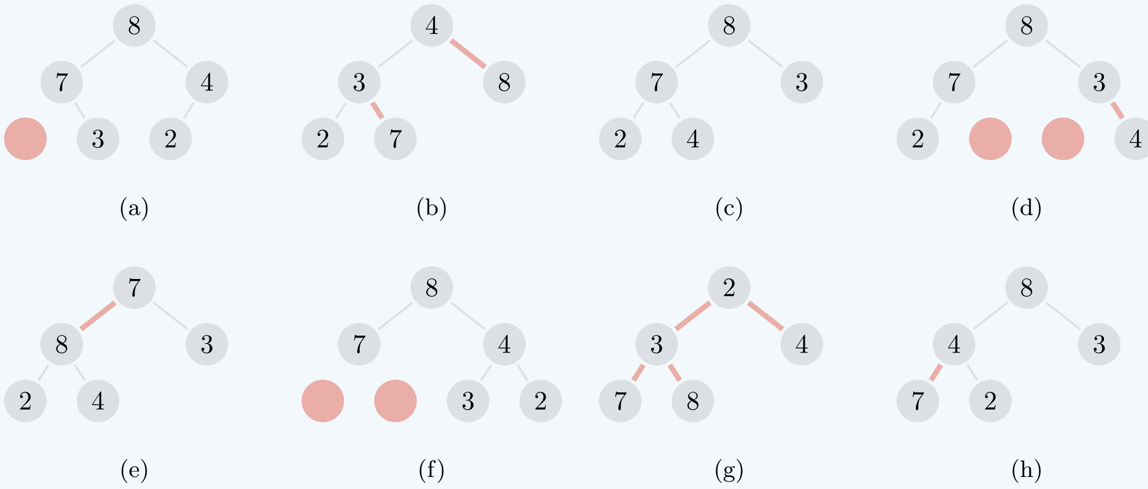
Beim zweiten Punkt von oben hingegen könnte man vermuten, dass da unnötigerweise Zeit verschwendet wird. Um die insgesamt grösste Zahl zu finden, schaut sich der Algorithmus alle n grauen Kästchen an. Um anschliessend die zweitgrösste Zahl zu finden, schaut er sich nochmals alle $n - 1$ verbleibenden grauen Kästchen an. Für die drittgrösste Zahl dann nochmals $n-2$ graue Kästchen und so weiter und so fort. Das bedeutet insbesondere, dass der Algorithmus sich sehr viele Einträge im Array immer wieder von Neuem anschaut und die dabei erhaltene Information gleich wieder vergisst.

Die beschriebene Heap-Datenstruktur löst genau dieses Teilproblem auf effizientere Weise. Sie erlaubt es, aus einer Menge wie $\{3, 2, 7, 8, 4\}$ das Maximum 8 zu extrahieren, ohne dabei alle Elemente einzeln anschauen zu müssen. Dabei wird eine neue Menge $\{3, 2, 7, 4\}$ ohne die Zahl 8 erstellt, aus welcher dann in einem zweiten Schritt das neue Maximum 7 effizient extrahiert werden kann und so weiter.

Das Ziel dieser Unterrichtsunterlagen ist es, genauer zu verstehen, wie so etwas überhaupt möglich sein kann.

Lösung zu Aufgabe 1.2 auf Seite 4

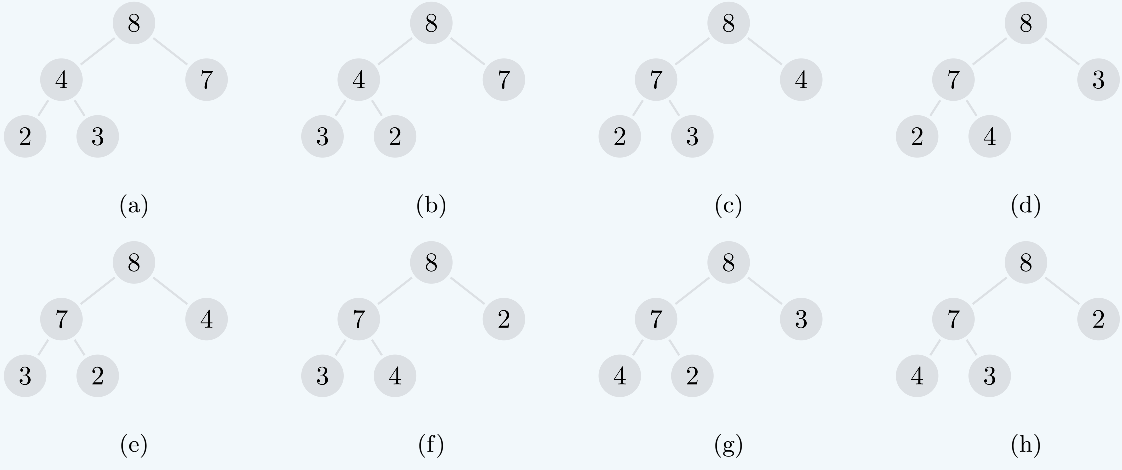
In den folgenden Illustrationen haben wir die unterschiedlichen Stellen, an denen die beiden Merkmale der Heap-Eigenschaft verletzt sind, rot eingezeichnet:



Insbesondere erfüllt nur Beispiel (c) die Heap-Eigenschaft. Dabei handelt es sich übrigens um den bereits bekannten Heap von $M = \{3, 2, 7, 8, 4\}$. Eine natürliche Frage ist nun, ob (c) vielleicht sogar die einzige Möglichkeit ist, um die Heap-Eigenschaft für dieses M zu erfüllen. Sehen Sie dazu die nächste Aufgabe.

Lösung zu Aufgabe 1.3 auf Seite 4

Folgende acht Möglichkeiten existieren, wobei es sich bei (d) um den uns schon bekannten Heap handelt:



(a) (b) (c) (d) (e) (f) (g) (h)

Lösung zu Aufgabe 1.4 auf Seite 4

Vollständig: Wir berechnen zuerst die Höhe eines Heaps, der die Gestalt eines vollständigen binären Baums hat, bei dem also alle h Ebenen bis und mit der h-ten und letzten Ebene vollständig mit Knoten besetzt sind. Das bedeutet nichts anderes, als dass die Anzahl n von Knoten in der Form

$$n = 1 + 2 + 4 + \cdots + 2^{h-1} = \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

geschrieben werden kann. Das stimmt tatsächlich, denn ein vollständiger binärer Baum hat 1 Knoten auf der ersten Ebene, 2 Knoten auf der zweiten Ebene, 4 Knoten auf der dritten Ebene, 2^{i-1} Knoten auf der i-ten Ebene und insbesondere 2^{h-1} Knoten auf der h-ten und letzten Ebene.

Die Gleichung von oben lässt sich jetzt ganz einfach nach h – also nach der Anzahl Ebenen oder nach der Höhe des Heaps – auflösen:

$$n = 2^h - 1 \quad \Longrightarrow \quad n + 1 = 2^h \quad \Longrightarrow \quad \log_2(n+1) = h$$

Unvollständig: Für den Fall, dass der Heap nicht die Gestalt eines vollständigen binären Baums hat, wissen wir wegen Merkmal (i) der Heap-Eigenschaft trotzdem, dass die ersten $h-1$ Ebenen vollständig mit Knoten besetzt sind und dass nach der unvollständigen h-ten und letzten Ebene keine weiteren Knoten mehr kommen. Also lässt sich jetzt die Anzahl n von Knoten in der Form

$$n = 1 + 2 + 4 + \cdots + 2^{h-2} + s = \sum_{i=1}^{h-1} 2^{i-1} + s = 2^{h-1} - 1 + s$$

schreiben, wobei s die Anzahl Knoten auf der h-ten Ebene ist. Wenn wir dieses s aus der Gleichung streichen, dann bekommen wir eine Ungleichung, die wir auf gleiche Weise wie vorher nach h auflösen:

$$n > 2^{h-1} - 1 \quad \Longrightarrow \quad n + 1 > 2^{h-1} \quad \Longrightarrow \quad \log_2(n+1) > h - 1 \quad \Longrightarrow \quad \log_2(n+1) + 1 > h$$

Eine Ungleichung in die andere Richtung bekommen wir, wenn wir das s mit 2^{h-1} ersetzen. Diese zweite Ungleichung liefert dann den gleichen Ausdruck wie im vollständigen Fall als untere Schranke für h.

Zusammenfassung: Die Höhe eines Heaps liegt zwischen $\log_2(n+1)$ und $\log_2(n+1) + 1$. Etwas konkreter ausgedrückt ist die Höhe eines Heaps mit n Knoten gleich $\lceil \log_2(n+1) \rceil$. In asymptotischer Notation schreiben wir dann auch, dass die Höhe eines Heaps gleich $O(\log n)$ ist.

☑ Lösung zu Aufgabe 1.5 auf Seite 4

Man findet das Maximum von M immer in der Wurzel des zugehörigen Heaps, also ganz oben. Mit einem Widerspruchsbeweis können wir uns auf folgende Weise von dieser Tatsache überzeugen.

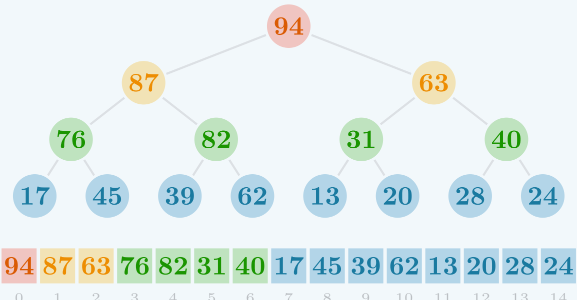
Nehmen wir doch einmal an, dass sich das Maximum *nicht* in der Wurzel des Heaps befindet. Dann hätte derjenige Knoten, der das Maximum enthält, gezwungenermassen einen Eltern-Knoten. In diesem Eltern-Knoten befände sich dann aber wiederum ein Element der Menge M. Und dieses Element müsste natürlich kleiner sein als das Maximum von M. Also bestünde zwischen diesen beiden Knoten ein Widerspruch zu der Annahme, dass Merkmal (ii) der Heap-Eigenschaft erfüllt ist.

☑ Lösung zu Aufgabe 2.1 auf Seite 5

Das gesuchte System ist das folgende:

> Man liest alle Knoten des Heaps zeilenweise von links nach rechts und von oben nach unten gehend nacheinander ab und fügt sie in der gleichen Reihenfolge in das Array ein.

Das heisst konkret, dass man in der ersten Ebene startet, auf der sich nur der Knoten 94 befindet, den man sogleich am Anfang des Arrays beim Index 0 einfügt. Dann geht man zur zweiten Ebene, auf der sich die Knoten 87 und 63 befinden, die man in dieser Reihenfolge bei den nächsten zwei Einträgen mit Index 1 und 2 einfügt. Für die dritte Ebene mit vier Knoten 76, 82, 31 und 40 verwendet man der Reihe nach die Einträge mit Index 3, 4, 5 und 6. Für die acht Knoten 17, 45, 39, 62, 13, 20, 28 und 24 in der untersten Ebene verwendet man schliesslich noch die verbleibenden Einträge mit Index 7 bis 14:



In der Abbildung oben heben wir die unterschiedlichen Ebenen hervor. Sowohl die Knoten des Heaps als auch die entsprechenden Einträge im Array sind mit unterschiedlichen Farben markiert.

☑ Lösung zu Aufgabe 2.2 auf Seite 6

Das sind die vervollständigten Diagramme mit der bereits bekannten Einfärbung:

(a)

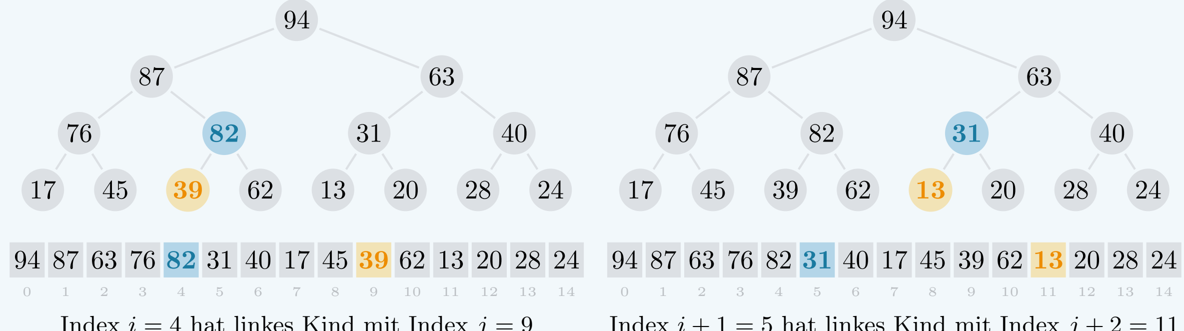
(b)

Lösung zu Aufgabe 2.3 auf Seite 6

Linkes Kind: Wir wollen uns zuerst auf das linke Kind konzentrieren und überlegen uns zu diesem Zweck den folgenden Zusammenhang:

> Falls ein Knoten mit Index i ein linkes Kind mit Index j hat, dann hat der Knoten mit Index $i+1$ ein linkes Kind mit Index $j+2$.

Dieser Zusammenhang wird mit Hilfe der folgenden Illustration veranschaulicht:



Index $i = 4$ hat linkes Kind mit Index $j = 9$ — Index $i + 1 = 5$ hat linkes Kind mit Index $j + 2 = 11$

Falls wir also auf einer bestimmten Ebene des Heaps vom Knoten mit Index i zum Knoten mit Index $i+1$ schreiten, dann muss auf der nächsttieferen Ebene immer ein Knoten übersprungen werden, um vom linken Kind mit Index j zum nächsten linken Kind mit Index $j+2$ zu gelangen.

Wir folgern, dass der Knoten mit Index i ein linkes Kind mit Index $2i+1$ hat. Der Faktor 2 in dieser Formel stammt von dem oben erklärten Zusammenhang. Der zusätzliche Summand 1 dient als Verankerung, so dass die Formel für die Wurzel (also für den Fall $i = 0$) das richtige Resultat liefert.

Rechtes Kind: Das rechte Kind befindet sich immer in der gleichen Ebene wie das linke Kind, und zwar genau eine Stelle weiter rechts. Also muss der Index des rechten Kindes um genau 1 grösser sein als der Index des linken Kindes. Somit ist der Index des rechten Kindes gleich $2i+2$.

Eltern-Knoten: Sei j der Index des Eltern-Knotens von unserem gegebenen Knoten mit Index i. Es gibt die zwei möglichen Fälle, dass der gegebene Knoten mit Index i selbst entweder ein linkes oder ein rechtes Kind ist. In ersterem Fall erhalten wir die Gleichung $i = 2j+1$ und in zweiterem Fall die Gleichung $i = 2j+2$. Beide Gleichungen lassen sich nach j auflösen und wir erhalten:

$$\text{Fall 1: } j = \frac{i-1}{2} \qquad\qquad \text{Fall 2: } j = \frac{i}{2} - 1$$

Wir könnten diese zwei Formeln gleich so als Endresultat für gerades i (Fall 2) als auch für ungerades i (Fall 1) stehen lassen. Wir können die zwei Fälle aber auch zusammenfassen, indem wir bemerken, dass durch Ab- und Aufrunden bei beiden Formeln das gleiche Resultat herauskommt:

$$\text{Fall 1 und Fall 2: } j = \left\lfloor \frac{i-1}{2} \right\rfloor = \left\lceil \frac{i}{2} - 1 \right\rceil$$

Lösung zu Aufgabe 2.4 auf Seite 6

Beachten Sie bitte, dass wir zur besseren Lesbarkeit in der folgenden Musterlösung die Berechnung des Index des Eltern-Knotens in eine separate Funktion `parent` ausgelagert haben. Gleichzeitig stellen wir auch die entsprechenden Funktionen `left` und `right` für das linke und rechte Kind bereit, welche in späteren Aufgaben Verwendung finden werden.

Falls Sie Ihre eigene Lösung auf Korrektheit überprüfen wollen, dann können Sie Ihren Code mit der Funktion `test_heap_check` testen.

```rust
/// Berechnet den Index des Eltern-Knotens fuer einen gegebenen Index
/// (bei Ganzzahldivision mit Rest wird automatisch abgerundet)
fn parent( i: usize ) -> usize { (i-1) / 2 }

/// Berechnet den Index des linken Kindes fuer einen gegebenen Index
fn left( i: usize ) -> usize { 2*i + 1 }

/// Berechnet den Index des rechten Kindes fuer einen gegebenen Index
fn right( i: usize ) -> usize { 2*i + 2 }

/// Ueberprueft die Heap-Eigenschaft fuer ein gegebenes Array
fn heap_check( array: & [u64] ) -> bool {
  // iteriere durch alle Knoten ausser der Wurzel
  for i in 1 .. array.len() {
    // ueberpruefe, ob der Knoten groesser ist als sein Eltern-Knoten
    if array[i] > array[parent( i )] {
      // falls ja, gib FALSCH zurueck
      return false;
    }
  }
  // andernfalls, gib WAHR zurueck
  return true;
}

/// Testet die Korrektheit von heap_check fuer eine Reihe von Arrays
#[test]
fn test_heap_check() {
  assert!(   heap_check( & [] ) );
  assert!(   heap_check( & [7] ) );
  assert!(   heap_check( & [7,6] ) );
  assert!( ! heap_check( & [6,7] ) );
  assert!(   heap_check( & [7,6,5] ) );
  assert!( ! heap_check( & [6,5,7] ) );
  assert!( ! heap_check( & [5,7,6] ) );
  assert!(   heap_check( & [7,6,5,4,3] ) );
  assert!( ! heap_check( & [7,4,3,6,5] ) );
  assert!(   heap_check( & [7,6,5,4,3,2,1] ) );
  assert!(   heap_check( & [7,5,6,1,2,3,4] ) );
  assert!(   heap_check( & [7,3,6,1,2,4,6] ) );
  assert!( ! heap_check( & [6,7,5,4,3,2,1] ) );
  assert!( ! heap_check( & [5,6,7,4,3,2,1] ) );
  assert!( ! heap_check( & [7,4,5,6,3,2,1] ) );
  assert!( ! heap_check( & [7,3,5,4,6,2,1] ) );
  assert!( ! heap_check( & [7,6,2,4,3,5,1] ) );
  assert!( ! heap_check( & [7,6,1,4,3,2,5] ) );
  assert!( ! heap_check( & [1,2,3,4,5,6,7] ) );
}
```

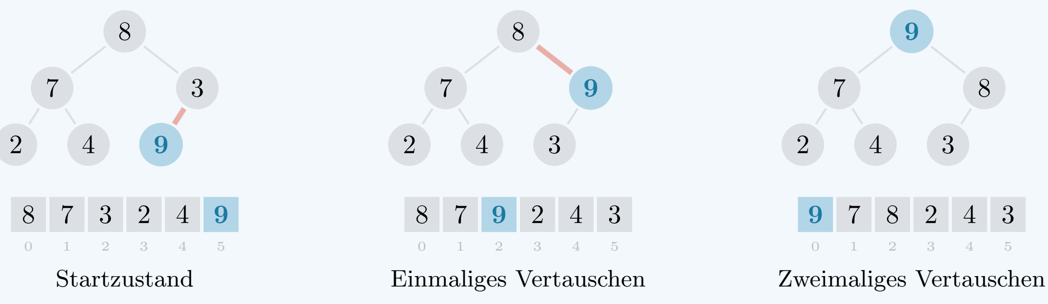
Lösung zu Aufgabe 3.1 auf Seite 7

Für das zusätzliche Element 5 ist nach einmaligem Vertauschen der zwei benachbarten Knoten entlang der roten Kante die Heap-Eigenschaft bereits erfüllt:

Startzustand

Einmaliges Vertauschen

Für das zusätzliche Element 9 stimmt dies hingegen nicht. Der zusätzliche Knoten verletzt nach einmaligem Vertauschen nach wie vor Merkmal (ii) der Heap-Eigenschaft. Jedoch erreichen wir auch hier nach zweimaligem Vertauschen einen Zustand, in dem die Heap-Eigenschaft erfüllt ist:



Startzustand

Einmaliges Vertauschen

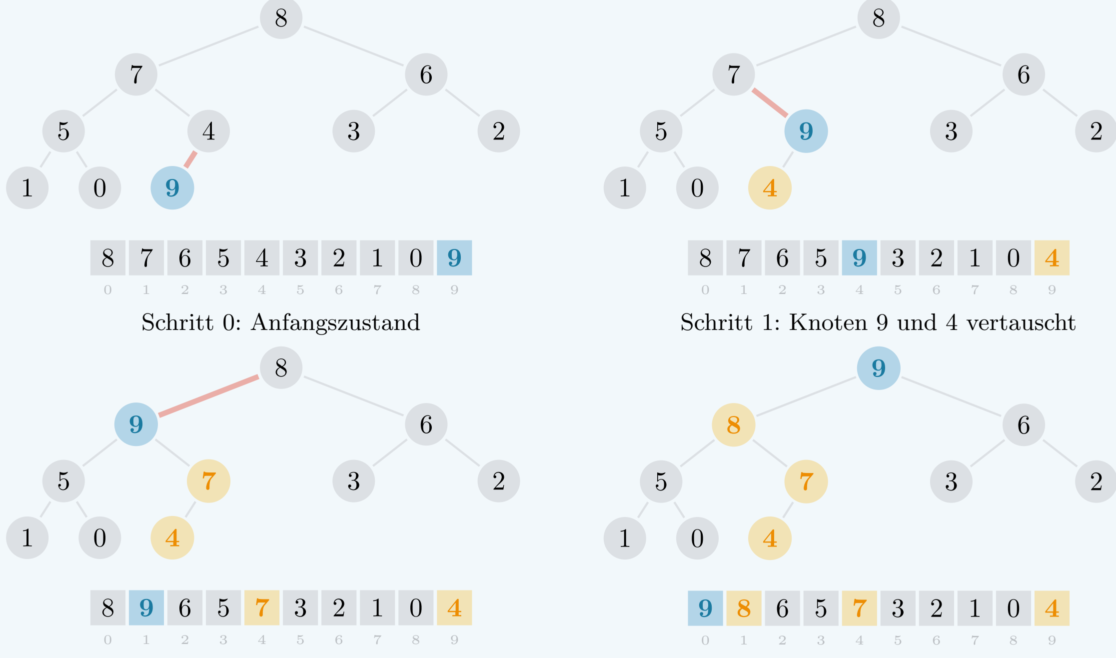
Zweimaliges Vertauschen

Lösung zu Aufgabe 3.2 auf Seite 8

Wir behaupten, dass folgende Aussage während des gesamten Vorgangs des Aufsteigens wahr bleibt:

> Zu jedem Zeitpunkt existiert maximal eine rote Kante, welche den zusätzlichen Knoten mit seinem aktuellen Eltern-Knoten verbindet.

Um uns davon zu überzeugen, müssen wir verstehen, was genau in jedem einzelnen Schritt des Vorgangs passiert – also bei der Vertauschung des zusätzlichen Knotens mit seinem aktuellen Eltern-Knoten entlang einer roten Kante. Wir betrachten dafür folgendes Beispiel mit 9 als zusätzlichen Knoten:



Schritt 0: Anfangszustand

Schritt 1: Knoten 9 und 4 vertauscht

Schritt 2: Knoten 9 und 7 vertauscht

Schritt 3: Knoten 9 und 8 vertauscht

Alle Knoten, die dabei jemals mit dem zusätzlichen Knoten vertauscht worden sind, haben wir gelb eingefärbt. Der zusätzliche Knoten selbst ist wie gewohnt blau eingefärbt.

Die gelben und der blaue Knoten formen nach jedem Schritt einen aufsteigenden Pfad mit dem blauen Knoten auf der jeweils obersten Ebene. Weiter existiert maximal eine rote Kante zwischen dem blauen Knoten und seinem grauen Eltern-Knoten. Wir sehen auf folgende Weise, dass dies kein Zufall ist:

- Jede Kante zwischen zwei grauen Knoten ist nicht rot, weil sie in gleicher Form schon im ursprünglichen Heap existiert hat.
- Jede Kante zwischen zwei gelben Knoten ist nicht rot, weil die gleiche Kante zwischen zwei ehemals grauen Knoten im ursprünglichen Heap existiert hat.
- Jede Kante zwischen einem grauen Knoten und seinem gelben Eltern-Knoten ist nicht rot, weil der ursprüngliche graue Eltern-Knoten jetzt als gelber Knoten eine Ebene tiefer sitzt und durch einen grösseren gelben Knoten ersetzt worden ist.
- Die einzige Kante zwischen einem grauen Knoten und seinem blauen Eltern-Knoten ist nicht rot, weil der ursprüngliche Eltern-Knoten im aktuellen Schritt gerade eben entlang einer roten Kante vertauscht worden ist – und zwar gerade deshalb, weil der ursprüngliche graue Eltern-Knoten kleiner war als der neue blaue Eltern-Knoten.
- Die einzige Kante zwischen einem gelben Knoten und seinem blauen Eltern-Knoten ist nicht rot, weil diese beiden Knoten im aktuellen Schritt gerade eben entlang einer roten Kante vertauscht worden sind – und zwar gerade deshalb, weil der blaue Knoten grösser war als der gelbe Knoten.

Dieser aufsteigende Pfad kann nur so lang sein wie die Höhe des Heaps – also höchstens $O(\log n)$.

Lösung zu Aufgabe 3.3 auf Seite 8

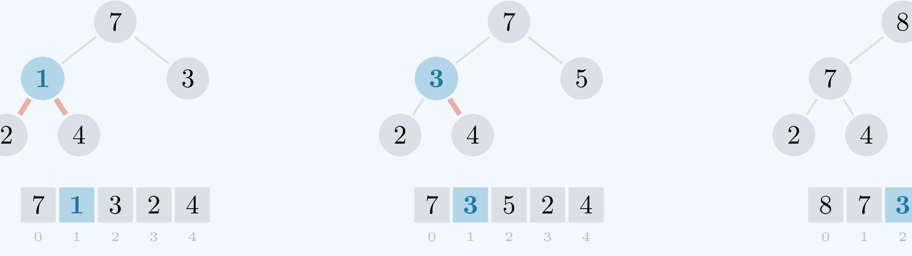
Beachten Sie bitte, dass in der folgenden Musterlösung die bereits in Aufgabe 2.4 ausgearbeiteten Funktionen `parent` und `heap_check` verwendet werden. Ihren eigenen Code können Sie falls erwünscht wieder mit Hilfe der Funktion `test_heap_ascend` auf Korrektheit überprüfen.

```
/// Fuehrt Aufsteigen auf gegebener Array-Repraesentation
/// mit zusaetzlichem letzten Eintrag aus
fn heap_ascend( array: & mut [u64] ) {
  // Vorbedingung: Heap nicht leer
  assert!( ! array.is_empty() );
  // starte beim letzten Eintrag
  let mut i = array.len() - 1;
  // solange wir nicht bei der Wurzel sind und
  // solange der Knoten groesser ist als sein Eltern-Knoten
  while i != 0 && array[i] > array[parent( i )] {
    // vertausche die zwei Knoten
    array.swap( i, parent( i ) );
    // steige eine Ebene auf
    i = parent( i );
  }
}

/// Testet die Korrektheit von heap_ascend fuer eine Reihe von Arrays
#[test]
fn test_heap_ascend() {
  let test_heap_ascend_once =
  |array: & mut [u64], expected: & [u64]| {
    heap_ascend( array );
    assert_eq!( array, expected );
  };
  test_heap_ascend_once( & mut [7],             & [7] );
  test_heap_ascend_once( & mut [7,6],           & [7,6] );
  test_heap_ascend_once( & mut [6,7],           & [7,6] );
  test_heap_ascend_once( & mut [7,6,5],         & [7,6,5] );
  test_heap_ascend_once( & mut [6,5,7],         & [7,5,6] );
  test_heap_ascend_once( & mut [7,6,5,4,3],     & [7,6,5,4,3] );
  test_heap_ascend_once( & mut [7,5,4,3,6],     & [7,6,4,3,5] );
  test_heap_ascend_once( & mut [6,5,4,3,7],     & [7,6,4,3,5] );
  test_heap_ascend_once( & mut [7,6,5,4,3,2,1], & [7,6,5,4,3,2,1] );
  test_heap_ascend_once( & mut [7,6,4,3,2,1,5], & [7,6,5,3,2,1,4] );
  test_heap_ascend_once( & mut [6,5,4,3,2,1,7], & [7,5,6,3,2,1,4] );
}
```

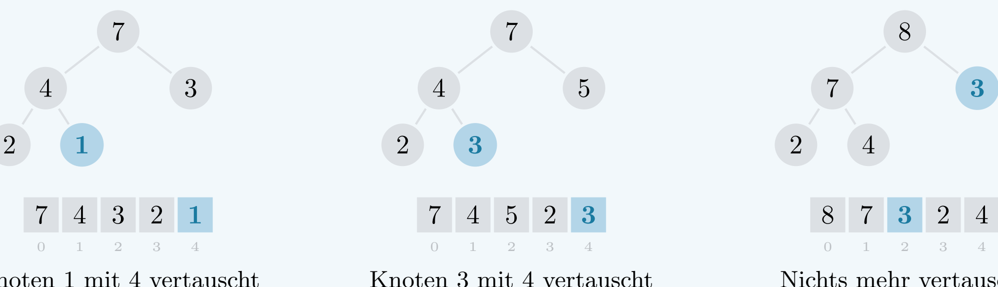
☑ Lösung zu Aufgabe 3.4 auf Seite 9

Es ist immer möglich, die rote Kante so auszuwählen, dass das Vertauschen der zwei entsprechenden Knoten entlang dieser Kante dazu führt, dass beide roten Kanten gleichzeitig repariert werden – also so, dass sie nach dem Vertauschen nicht mehr rot sind. Wir erreichen dies, wenn wir den blauen Knoten mit dem jeweils grösseren seiner zwei Kinder vertauschen:



Knoten 1 mit 7 vertauscht | Knoten 3 mit 7 vertauscht | Knoten 3 mit 8 vertauscht

Wie Sie sehen, entsteht durch dieses Vorgehen eine Reihe von unterschiedlichen Situationen. Im ersten Fall entsteht wieder die gleiche Situation mit zwei roten Kanten zu den beiden Kindern des blauen Knotens; im zweiten Fall ist lediglich eine dieser zwei Kanten rot; und im dritten Fall gibt es gar keine roten Kanten mehr. Wir können also das gleiche Vorgehen im ersten Fall wiederholen; im zweiten Fall wählen wir einfach die einzige rote Kante; und im dritten Fall gibt es gar nichts mehr zu tun:



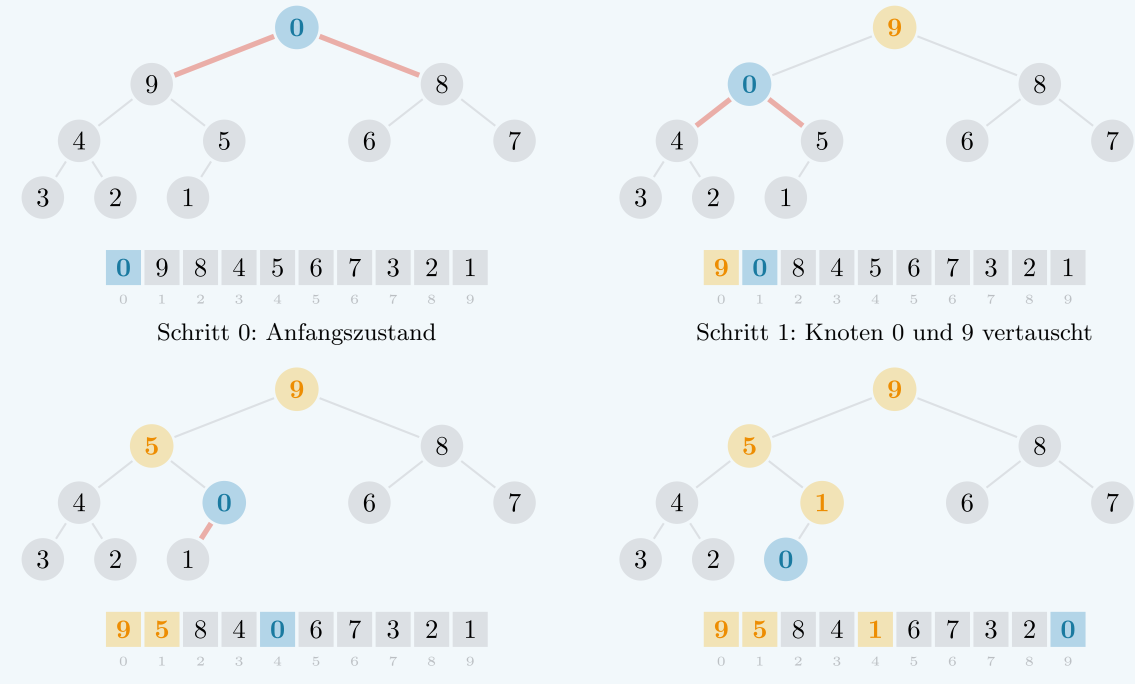
Knoten 1 mit 4 vertauscht | Knoten 3 mit 4 vertauscht | Nichts mehr vertauscht

Lösung zu Aufgabe 3.5 auf Seite 9

Wir behaupten, dass folgende Aussage während des gesamten Vorgangs des Absickerns wahr bleibt:

> Zu jedem Zeitpunkt existieren maximal zwei rote Kanten, welche den absickernden Knoten mit je einem seiner zwei Kinder verbinden.

Um uns davon zu überzeugen, müssen wir wieder verstehen, was genau in jedem einzelnen Schritt des Vorgangs passiert – also bei der Vertauschung des absickernden Knotens mit dem grösseren seiner zwei Kinder entlang einer roten Kante. Wir betrachten dafür folgendes Beispiel mit 0 in der Rolle des absickernden Knotens:



Schritt 0: Anfangszustand

Schritt 1: Knoten 0 und 9 vertauscht

Schritt 2: Knoten 0 und 5 vertauscht

Schritt 3: Knoten 0 und 1 vertauscht

Alle Knoten, die dabei jemals mit dem absickernden Knoten vertauscht worden sind, haben wir gelb eingefärbt. Der absickernde Knoten selbst ist wie gewohnt blau eingefärbt.

Die gelben und der blaue Knoten formen nach jedem Schritt einen absteigenden Pfad mit dem blauen Knoten auf der jeweils untersten Ebene. Weiter existieren maximal zwei rote Kanten zwischen dem blauen Knoten und seinen Kindern. Wir sehen auf folgende Weise, dass dies kein Zufall ist:

- Jede Kante zwischen zwei grauen Knoten ist nicht rot, weil sie in gleicher Form schon im ursprünglichen Heap existiert hat.
- Jede Kante zwischen zwei gelben Knoten ist nicht rot, weil die gleiche Kante zwischen zwei ehemals grauen Knoten im ursprünglichen Heap existiert hat.
- Jede Kante zwischen einem grauen Knoten und seinem gelben Eltern-Knoten ist nicht rot, weil der gelbe Eltern-Knoten in einem früheren Schritt des Vorgangs genau deshalb in diese Position gebracht worden war, weil er das grössere von den zwei damaligen Kindern war.
- Die einzige Kante zwischen dem blauen Knoten und seinem gelben Eltern-Knoten ist nicht rot, weil diese beiden Knoten im aktuellen Schritt gerade eben entlang einer roten Kante vertauscht worden sind – und zwar insbesondere deshalb, weil der gelbe Knoten grösser war als der blaue Knoten.

Dieser absteigende Pfad kann nur so lang sein wie die Höhe des Heaps – also höchstens $O(\log n)$.

☑ Lösung zu Aufgabe 3.6 auf Seite 9

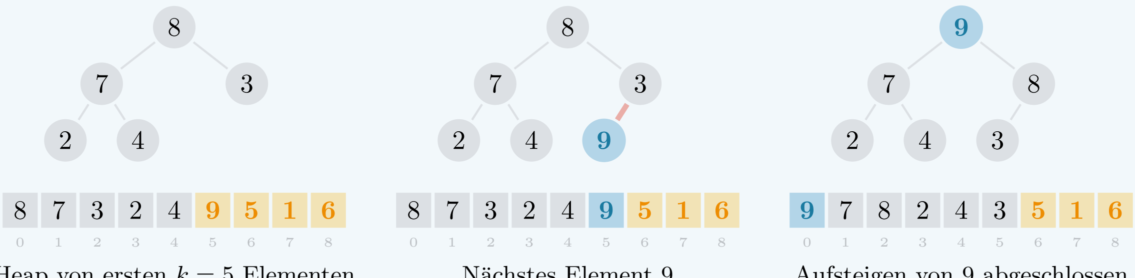
Legen Sie in folgender Musterlösung Ihr Augenmerk bitte auf die spezielle Logik bei der Berechnung der Indizes `l` und `r`. Für den Fall, dass beim letzten Schritt entweder kein linkes oder rechtes Kind mehr existiert, erlaubt uns dies die Ausführung des restlichen Codes ohne weitere Fallunterscheidung und ohne fehlerhaften Zugriff ausserhalb der Grenzen der Array-Repräsentation.

```
/// Fuehrt Absickern auf gegebener Array-Repraesentation
/// mit deplatziertem Element beim ersten Eintrag aus
fn heap_descend( array: & mut [u64] ) {
  // Vorbedingung: Heap nicht leer
  assert!( ! array.is_empty() );
  // starte bei der Wurzel und wiederhole folgenden Vorgang
  let mut i = 0;
  loop {
    // berechne validen Index fuer beide Kinder
    let l = if  left( i ) < array.len() {  left( i ) } else { i };
    let r = if right( i ) < array.len() { right( i ) } else { i };
    // falls das linke Kind am groessten ist
    if array[l] > array[i] && array[l] >= array[r] {
      // vertausche mit linkem Kind und gehe weiter
      array.swap( i, l ); i = l;
    }
    // ansonsten, falls das rechte Kind am groessten ist
    else if array[r] > array[i] && array[r] >= array[l] {
      // vertausche mit rechtem Kind und gehe weiter
      array.swap( i, r ); i = r;
    }
    // ansonsten, breche den Vorgang ab
    else { break; }
  }
}

/// Testet die Korrektheit von heap_descend fuer eine Reihe von Arrays
#[test]
fn test_heap_descend() {
  let test_heap_descend_once =
  |array: & mut [u64], expected: & [u64]| {
    heap_descend( array );
    assert_eq!( array, expected );
  };
  test_heap_descend_once( & mut [7],             & [7] );
  test_heap_descend_once( & mut [7,6],           & [7,6] );
  test_heap_descend_once( & mut [6,7],           & [7,6] );
  test_heap_descend_once( & mut [7,6,5],         & [7,6,5] );
  test_heap_descend_once( & mut [6,5,7],         & [7,5,6] );
  test_heap_descend_once( & mut [5,7,6],         & [7,5,6] );
  test_heap_descend_once( & mut [7,6,5,4,3],     & [7,6,5,4,3] );
  test_heap_descend_once( & mut [5,7,6,4,3],     & [7,5,6,4,3] );
  test_heap_descend_once( & mut [3,7,6,5,4],     & [7,5,6,3,4] );
  test_heap_descend_once( & mut [7,6,5,4,3,2,1], & [7,6,5,4,3,2,1] );
  test_heap_descend_once( & mut [1,7,6,5,4,3,2], & [7,5,6,1,4,3,2] );
  test_heap_descend_once( & mut [1,6,7,2,3,4,5], & [7,6,5,2,3,4,1] );
  test_heap_descend_once( & mut [1,7,6,2,3,4,5], & [7,3,6,2,1,4,5] );
  test_heap_descend_once( & mut [1,6,7,5,4,3,2], & [7,6,3,5,4,1,2] );
}
```

Lösung zu Aufgabe 3.7 auf Seite 9

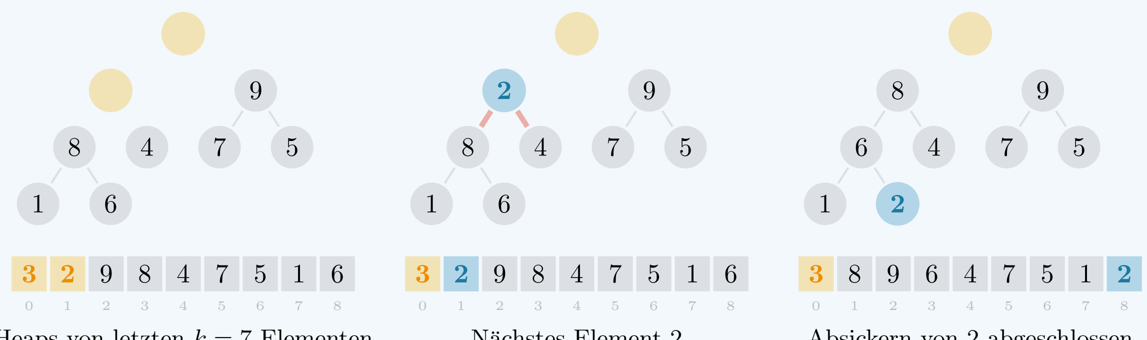
Aufsteigen: Auch wenn wir alle Elemente der Menge M auf einen Schlag bekommen, können wir trotzdem alle Elemente schön nacheinander in einen zu Beginn leeren Heap einfügen. Wir verwenden dafür die ersten k Einträge von A als die Array-Repräsentation des Heaps von den ersten k auf diese Weise eingefügten Elementen. Die restlichen $n - k$ Einträge von A hingegen dienen als Warteschlange für alle anderen Elemente, die noch nicht eingefügt worden sind. Mit folgendem Beispiel illustrieren wir einen Schritt dieses Vorgehens:



Heap von ersten $k = 5$ Elementen   Nächstes Element 9   Aufsteigen von 9 abgeschlossen

Ganz links haben wir die Situation, in der die ersten $k = 5$ Elemente bereits eingefügt sind – das heisst also, dass die ersten $k = 5$ Einträge von A die Array-Repräsentation eines Heaps darstellen. In der Mitte haben wir die direkt darauf folgende Situation, in der das nächste Element 9 nun auch als Teil des binären Baums verstanden wird, der jetzt aber die Heap-Eigenschaft nicht mehr erfüllt. Ganz rechts haben wir die Situation, nachdem der Vorgang des Aufsteigens auf dem binären Baum mit jetzt $k = 6$ Elementen abgeschlossen worden ist – das heisst also, dass die Heap-Eigenschaft jetzt wieder erfüllt ist. Zur besseren Übersicht sind die noch nicht eingefügten Einträge in der Warteschlange jeweils gelb eingefärbt.

Absickern: Wir können auf ganz ähnliche Weise einen zweiten Algorithmus entwerfen, der aber auf dem Vorgang des Absickerns beruht. Dazu müssen wir das gegebene Array A nicht wie vorher von links nach rechts, sondern jetzt einfach von rechts nach links abarbeiten. Die Warteschlange ist jetzt am Anfang des Arrays und nicht mehr am Ende. Dies bedeutet auch, dass die bereits eingefügten Elemente gleich mehrere Teil-Heaps formen, die erst in einem späteren Schritt verbunden werden. Folgende Illustration eines Schrittes macht diese Eigenheit des Algorithmus hoffentlich klar:



Heaps von letzten $k = 7$ Elementen   Nächstes Element 2   Absickern von 2 abgeschlossen

Ganz links haben wir die Situation, in der die letzten $k = 7$ Elemente bereits eingefügt sind – insbesondere formen die entsprechenden Knoten momentan drei separate Teil-Heaps. In der Mitte haben wir die direkt darauf folgende Situation, in der das nächste Element 2 jetzt auch als Knoten dieser Teil-Heaps verstanden wird und es deshalb nur noch zwei separate Teil-Heaps gibt. Ganz rechts haben wir die Situation, nachdem der Vorgang des Absickerns auf dem betroffenen Teil-Heap abgeschlossen worden ist. Zur besseren Übersicht sind die noch fehlenden Knoten mit gelber Farbe angedeutet.

Laufzeit: Beide Algorithmen führen je n mal einen der Vorgänge Aufsteigen oder Absickern aus. Für beide dieser Vorgänge wissen wir aber bereits aus den Aufgaben 3.2 und 3.5, dass sie je höchstens $O(\log n)$ mal zwei Einträge des Arrays vertauschen. Für beide Algorithmen ist also die Anzahl Vertauschungen insgesamt höchstens $O(n \log n)$.

Lösung zu Aufgabe 3.8 auf Seite 10

Aufsteigen: Folgender Code ist identisch zu Aufgabe 3.3 bis auf den kleinen Unterschied, dass der Vorgang jetzt beim gegebenen Index `i` startet anstatt bei `array.len()-1`:

```
/// Fuehrt Aufsteigen auf gegebener Array-Repraesentation
/// bei gegebenem Index aus
fn heap_ascend_at_index( array: & mut [u64], i: usize ) {
  // Vorbedingung: Valider Index
  assert!( array.len() > i );
  // starte beim gegebenen Eintrag
  let mut i = i;
  // solange wir nicht bei der Wurzel sind und
  // solange der Knoten groesser ist als sein Eltern-Knoten
  while i != 0 && array[i] > array[parent( i )] {
    // vertausche die zwei Knoten
    array.swap( i, parent( i ) );
    // steige eine Ebene auf
    i = parent( i );
  }
}
```

Absickern: Folgender Code ist identisch zu Aufgabe 3.6 bis auf den kleinen Unterschied, dass der Vorgang jetzt beim gegebenen Index `i` startet anstatt bei `0`:

```
/// Fuehrt Absickern auf gegebener Array-Repraesentation
/// bei gegebenem Index aus
fn heap_descend_at_index( array: & mut [u64], i: usize ) {
  // Vorbedingung: Valider Index
  assert!( array.len() > i );
  // starte bei gegebenem Eintrag und wiederhole folgenden Vorgang
  let mut i = i;
  loop {
    // berechne validen Index fuer beide Kinder
    let l = if  left( i ) < array.len() {  left( i ) } else { i };
    let r = if right( i ) < array.len() { right( i ) } else { i };
    // falls das linke Kind am groessten ist
    if array[l] > array[i] && array[l] >= array[r] {
      // vertausche mit linkem Kind und gehe weiter
      array.swap( i, l ); i = l;
    }
    // ansonsten, falls das rechte Kind am groessten ist
    else if array[r] > array[i] && array[r] >= array[l] {
      // vertausche mit rechtem Kind und gehe weiter
      array.swap( i, r ); i = r;
    }
    // ansonsten, breche den Vorgang ab
    else { break; }
  }
}
```

☑ Lösung zu Aufgabe 3.9 auf Seite 10

Machen Sie sich bitte mit Hilfe der Testfunktionen in folgender Musterlösung bewusst, dass die beiden Algorithmen bei gleicher Eingabe nicht unbedingt immer dieselbe Array-Repräsentation eines Heaps konstruieren:

```
/// Erstellt einen Heap fuer ein gegebenes Array von natuerlichen Zahlen
/// mit Hilfe des Vorgangs Aufsteigen
fn heap_create_with_ascend( array: & mut [u64] ) {
  // iteriere ueber alle Eintraege
  for i in 0 .. array.len() {
    // fuehre Aufsteigen auf gegebenem Eintrag aus
    heap_ascend_at_index( array, i );
  }
  // Nachbedingung: Heap-Eigenschaft erfuellt
  assert!( heap_check( array ) );
}

/// Testet die Korrektheit von heap_create_with_ascend
/// fuer eine Reihe von Arrays
#[test]
fn test_heap_create_with_ascend() {
  let test_heap_create_once =
  |array: & mut [u64], expected: & [u64]| {
    heap_create_with_ascend( array );
    assert_eq!( array, expected );
  };
  test_heap_create_once( & mut [1,2,3,4,5,6,7], & [7,4,6,1,3,2,5] );
  test_heap_create_once( & mut [1,3,5,7,2,4,6], & [7,5,6,1,2,3,4] );
  test_heap_create_once( & mut [6,4,2,7,5,3,1], & [7,6,3,4,5,2,1] );
  test_heap_create_once( & mut [1,7,2,6,3,5,4], & [7,6,5,1,3,2,4] );
}

/// Erstellt einen Heap fuer ein gegebenes Array von natuerlichen Zahlen
/// mit Hilfe des Vorgangs Absickern
fn heap_create_with_descend( array: & mut [u64] ) {
  // iteriere ueber alle Eintraege in umgekehrter Reihenfolge
  for i in (0 .. array.len()).rev() {
    // fuehre Absickern auf gegebenem Eintrag aus
    heap_descend_at_index( array, i );
  }
  // Nachbedingung: Heap-Eigenschaft erfuellt
  assert!( heap_check( array ) );
}

/// Testet die Korrektheit von heap_create_with_descend
/// fuer eine Reihe von Arrays
#[test]
fn test_heap_create_with_descend() {
  let test_heap_create_once =
  |array: & mut [u64], expected: & [u64]| {
    heap_create_with_descend( array );
    assert_eq!( array, expected );
  };
  test_heap_create_once( & mut [1,2,3,4,5,6,7], & [7,5,6,4,2,1,3] );
  test_heap_create_once( & mut [1,3,5,7,2,4,6], & [7,3,6,1,2,4,5] );
  test_heap_create_once( & mut [6,4,2,7,5,3,1], & [7,6,3,4,5,2,1] );
  test_heap_create_once( & mut [1,7,2,6,3,5,4], & [7,6,5,1,3,2,4] );
}
```

Lösung zu Aufgabe 3.10 auf Seite 10

Konstruktion: Das Array $A = [1, 2, 3, \dots, n]$ aller Zahlen von 1 bis n in aufsteigend sortierter Reihenfolge erfüllt den gewünschten Zweck. Wir argumentieren separat für beide Algorithmen wie folgt.

Aufsteigen: Wenn beim Erstellen des Heaps durch Aufsteigen beim k-ten Schritt die Zahl k eingefügt wird, dann handelt es sich dabei um die *grösste* bis zu diesem Zeitpunkt eingefügte Zahl. Der Knoten k wird deshalb bis ganz nach oben in die Wurzel des Heaps aufsteigen.

Absickern: Beim Erstellen des Heaps durch Absickern hingegen fügen wir beim k-ten Schritt die Zahl $n - k + 1$ von rechts nach links gehend hinzu. Dabei handelt es sich um die *kleinste* bis zu diesem Zeitpunkt betrachtete Zahl. Der Knoten $n - k + 1$ wird also bis in die unterste Ebene absickern.

Lösung zu Aufgabe 3.11 auf Seite 10

Aufsteigen: Das Element 1, das zuerst eingefügt wird, kommt direkt in die Wurzel des Heaps und wird nicht weiter vertauscht. Die zwei Elemente 2 und 3 danach steigen mit je einer Vertauschung von der zweiten Ebene in die Wurzel auf. Die vier Elemente 4, 5, 6 und 7 benötigen dann bereits zwei Vertauschungen. Im Allgemeinen gibt es 2^i Elemente, welche, wenn sie eingefügt werden, genau i Vertauschungen verursachen. Wir erhalten also folgende Formel für die Anzahl von Vertauschungen:

$$\sum_{i=0}^{h-1} 2^i \cdot i$$

Absickern: Die Überlegung hier ist ähnlich wie oben, allerdings mit dem Unterschied, dass die Elemente jetzt abwärts in die unterste Ebene absickern. Wir folgern, dass es 2^i Elemente gibt, welche, wenn sie eingefügt werden, genau $(h - i - 1)$ Vertauschungen verursachen. Wir erhalten also folgende Formel:

$$\sum_{i=0}^{h-1} 2^i \cdot (h - i - 1)$$

Lösung zu Aufgabe 3.12 auf Seite 11

Wir erhalten die gewünschte Abschätzung, wenn wir uns auf den letzten Summanden in der Formel beschränken und die beiden Gleichungen $2^h = n + 1$ und $h = \log_2(n + 1)$ einsetzen:

$$\sum_{i=0}^{h-1} 2^i \cdot i \geq 2^{h-1} \cdot (h-1) = \frac{2^h}{2} \cdot (h-1) = \frac{n+1}{2} \cdot \log_2\left(\frac{n+1}{2}\right) \geq \frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right)$$

Aus dieser Abschätzung lernen wir auch gleich, dass die Laufzeitanalyse von Aufgabe 3.2 mit höchstens $O(n \log n)$ Vertauschungen in asymptotischer Notation nicht weiter verbessert werden kann.

Lösung zu Aufgabe 3.13 auf Seite 11

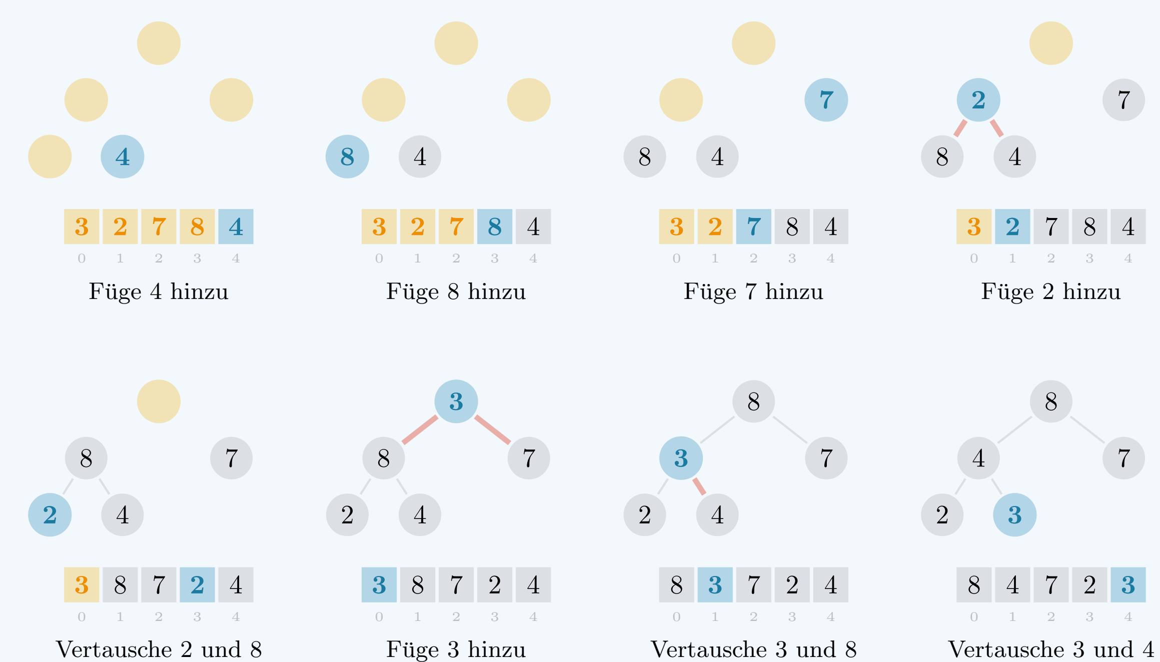
Wir erhalten die gewünschte Abschätzung auf folgende Weise, indem wir die in der Aufgabenstellung erwähnte Gleichung als obere Schranke einsetzen und ganz am Schluss noch $2^h = n + 1$ verwenden:

$$\sum_{i=0}^{h-1} 2^i \cdot (h-i-1) = \sum_{i=0}^{h-1} \frac{h-i-1}{2^{-i}} = 2^{h-1} \cdot \sum_{i=0}^{h-1} \frac{h-i-1}{2^{h-i-1}} = 2^{h-1} \cdot \underbrace{\sum_{j=0}^{h-1} \frac{j}{2^j}}_{\leq 2} \leq 2^h = n + 1$$

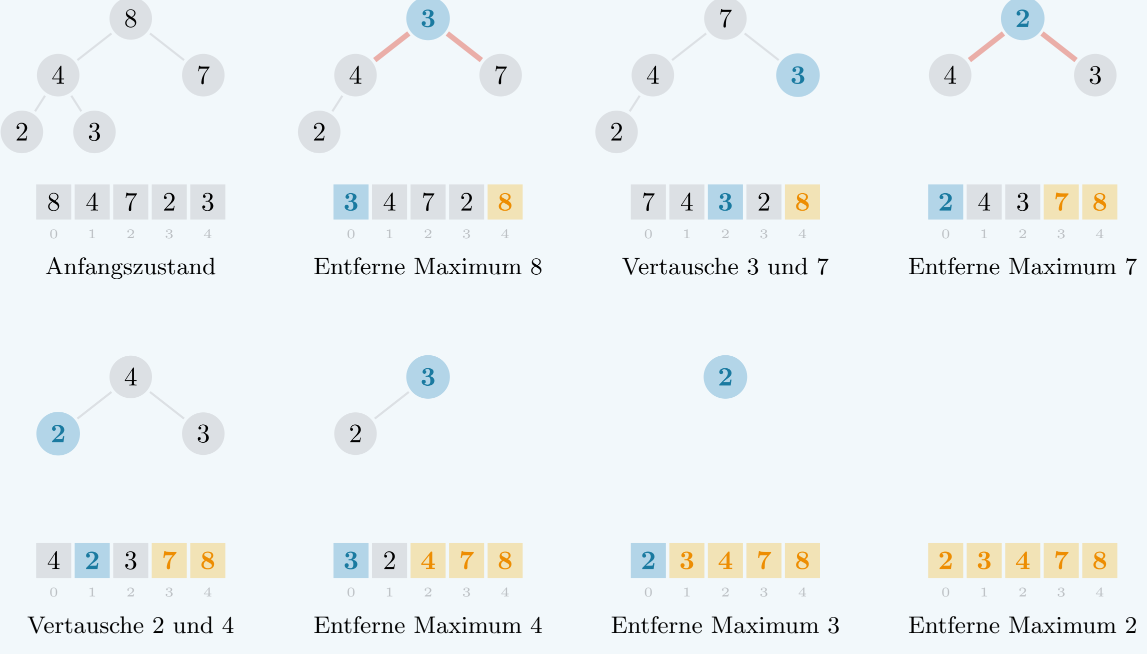
Mit dieser Abschätzung haben wir gerade gezeigt, dass es im schlimmsten Fall höchstens $n + 1$ Vertauschungen gibt beim Erstellen eines Heaps von n Elementen durch Absickern. Dieser Algorithmus hat also die asymptotische Laufzeit $O(n)$ und ist damit effizienter als die Alternative mit Aufsteigen.

☑ Lösung zu Aufgabe 4.1 auf Seite 12

Erstellen des Heaps: In den folgenden Abbildungen sind wie gewohnt der absickernde Knoten jeweils blau und die Einträge in der Warteschlange jeweils gelb eingefärbt:



Entfernen der Maxima: In den folgenden Abbildungen ist wiederum der absickernde Knoten blau eingefärbt; gelbe Farbe hingegen wird jetzt für die sortierte Teilfolge von bereits entfernten Elementen verwendet:



Lösung zu Aufgabe 4.2 auf Seite 12

Beachten Sie, dass in folgender Musterlösung die praktische Subslice-Notation `array[0..i]` verwendet wird. Dies erlaubt uns ohne weitere Umstände einen kürzeren Bereich des gegebenen Arrays als Repräsentation eines immer kleiner werdenden Heaps zu verwenden.

```rust
/// Ordnet die Elemente eines gegebenen Heaps von natuerlichen Zahlen
/// in sortierter Reihenfolge an
fn heap_arrange_inorder( array: & mut [u64] ) {
  // Vorbedingung: Heap-Eigenschaft erfuellt
  assert!( heap_check( array ) );
  // iteriere ueber alle Eintraege in umgekehrter Reihenfolge
  for i in (1 .. array.len()).rev() {
    // vertausche die Wurzel mit dem gegebenen Eintrag
    array.swap( 0, i );
    // sickere die neue Wurzel im kleineren Heap ab
    heap_descend_at_index( & mut array[0..i], 0 );
  }
}

/// Testet die Korrektheit von heap_arrange_inorder
/// fuer eine Reihe von Heaps
#[test]
fn test_heap_arrange_inorder() {
  let test_heap_arrange_once =
  |array: & mut [u64], expected: & [u64]| {
    heap_arrange_inorder( array );
    assert_eq!( array, expected );
  };
  test_heap_arrange_once( & mut [],              & [] );
  test_heap_arrange_once( & mut [7],             & [7] );
  test_heap_arrange_once( & mut [7,6],           & [6,7] );
  test_heap_arrange_once( & mut [7,6,5],         & [5,6,7] );
  test_heap_arrange_once( & mut [7,5,6],         & [5,6,7] );
  test_heap_arrange_once( & mut [7,6,5,4,3],     & [3,4,5,6,7] );
  test_heap_arrange_once( & mut [7,6,3,4,5],     & [3,4,5,6,7] );
  test_heap_arrange_once( & mut [7,5,6,3,4],     & [3,4,5,6,7] );
  test_heap_arrange_once( & mut [7,6,5,4,3,2,1], & [1,2,3,4,5,6,7] );
  test_heap_arrange_once( & mut [7,5,6,1,2,3,4], & [1,2,3,4,5,6,7] );
  test_heap_arrange_once( & mut [7,3,6,1,2,4,5], & [1,2,3,4,5,6,7] );
  test_heap_arrange_once( & mut [7,6,3,5,4,2,1], & [1,2,3,4,5,6,7] );
}
```

☑ Lösung zu Aufgabe 4.3 auf Seite 12

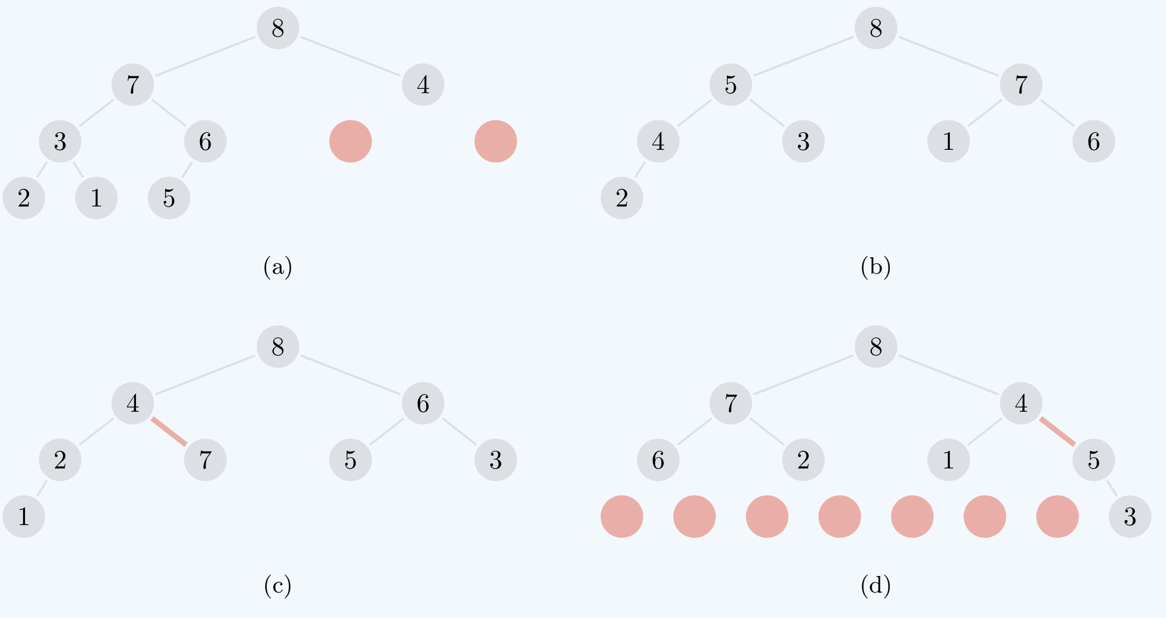
Mit den schon gegebenen Funktionen für die Erstellung eines Heaps von Aufgabe 3.9 und für die Umwandlung eines Heaps in ein sortiertes Array von Aufgabe 4.2 ist es jetzt ein Leichtes, eine Implementierung für Heapsort anzugeben:

```rust
/// Sortiert ein gegebenes Array von natuerlichen Zahlen
fn heap_sort( array: & mut [u64] ) {
  // erstelle einen Heap durch Absickern
  heap_create_with_descend( array );
  // ordne die Elemente in sortierter Reihenfolge an
  heap_arrange_inorder( array );
}

/// Testet die Korrektheit von heap_sort fuer eine Reihe von Arrays
#[test]
fn test_heap_sort() {
  let test_heap_sort_once =
  |array: & mut [u64], expected: & [u64]| {
    heap_sort( array );
    assert_eq!( array, expected );
  };
  test_heap_sort_once( & mut [],              & [] );
  test_heap_sort_once( & mut [7],             & [7] );
  test_heap_sort_once( & mut [6,7],           & [6,7] );
  test_heap_sort_once( & mut [7,6],           & [6,7] );
  test_heap_sort_once( & mut [5,6,7],         & [5,6,7] );
  test_heap_sort_once( & mut [5,7,6],         & [5,6,7] );
  test_heap_sort_once( & mut [6,5,7],         & [5,6,7] );
  test_heap_sort_once( & mut [6,7,5],         & [5,6,7] );
  test_heap_sort_once( & mut [7,5,6],         & [5,6,7] );
  test_heap_sort_once( & mut [7,6,5],         & [5,6,7] );
  test_heap_sort_once( & mut [3,4,5,6,7],     & [3,4,5,6,7] );
  test_heap_sort_once( & mut [7,6,5,4,3],     & [3,4,5,6,7] );
  test_heap_sort_once( & mut [3,5,7,6,4],     & [3,4,5,6,7] );
  test_heap_sort_once( & mut [6,4,3,5,7],     & [3,4,5,6,7] );
  test_heap_sort_once( & mut [1,2,3,4,5,6,7], & [1,2,3,4,5,6,7] );
  test_heap_sort_once( & mut [7,6,5,4,3,2,1], & [1,2,3,4,5,6,7] );
  test_heap_sort_once( & mut [1,3,5,7,6,4,2], & [1,2,3,4,5,6,7] );
  test_heap_sort_once( & mut [6,4,2,1,3,5,7], & [1,2,3,4,5,6,7] );
}
```

Lösung zu Aufgabe 5.1 auf Seite 13

Nur der binäre Baum (b) erfüllt beide Merkmale der Heap-Eigenschaft. In den folgenden Illustrationen heben wir die fehlenden Knoten für Merkmal (i) und die defekten Kanten für Merkmal (ii) der Heap-Eigenschaft mit roter Farbe hervor:



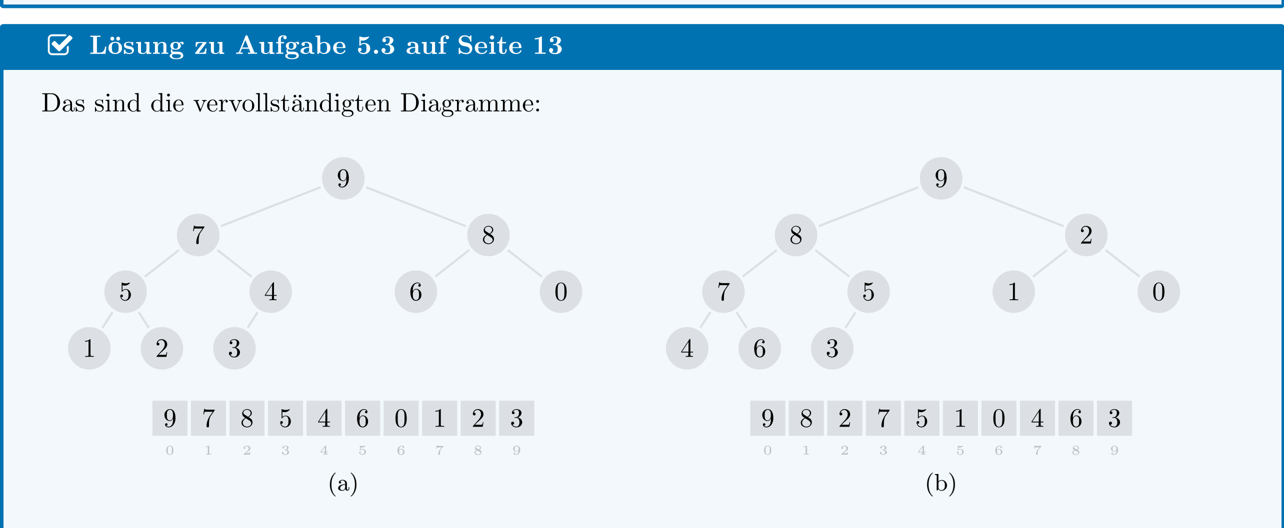
(a) (b) (c) (d)

Lösung zu Aufgabe 5.2 auf Seite 13

Die wichtigste Anpassung, die wir zuerst machen müssten, liegt natürlich bei Merkmal (ii) der Heap-Eigenschaft. Hier müssten wir verlangen, dass jeder Knoten ein grösseres Element enthält als sein Eltern-Knoten. Mit dieser angepassten Heap-Eigenschaft würden wir dann immer das Minimum von M in der Wurzel des Heaps finden. Anschliessend müssten wir an jeder Stelle – in Beispielen, Algorithmen und Programmcode – in diesen Unterrichtsunterlagen, an der zwei Zahlen verglichen werden, den entsprechenden Vergleich umdrehen.

Lösung zu Aufgabe 5.3 auf Seite 13

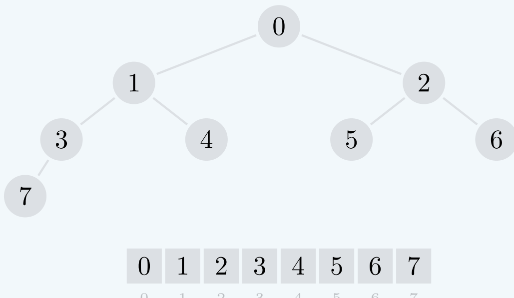
Das sind die vervollständigten Diagramme:



(a) (b)

Lösung zu Aufgabe 5.4 auf Seite 14

Zur besseren Übersicht ist es hilfreich, einen binären Baum mit den entsprechenden Indizes in den acht Knoten aufzuzeichnen. Beachten Sie aber bitte, dass es sich dabei streng genommen gar nicht um einen Heap handelt, weil das Merkmal (ii) der Heap-Eigenschaft überall verletzt ist:



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

Knoten $i = 0$: Der Knoten mit Index 0 entspricht immer der Wurzel und hat deshalb gar keinen Eltern-Knoten. Mit den Formeln von Aufgabe 2.3 oder mit Hilfe der Abbildung oben sehen wir weiter, dass (b) das linke Kind den Index 1 und (c) das rechte Kind den Index 2 hat.

Knoten $i = 2$: Die korrekten Indizes sind (a) für den Eltern-Knoten 0, (b) für das linke Kind 5 und dann noch (c) für das rechte Kind 6.

Knoten $i = 3$: Hier haben wir eine Situation, in der zwar kein rechtes, aber trotzdem ein linkes Kind existiert. Die korrekten Indizes sind (a) für den Eltern-Knoten 1 und (b) für das linke Kind 7.

Knoten $i = 8$: Es gibt gar keinen Knoten mit Index 8. Deshalb macht auch keine einzige der drei Fragen (a), (b) oder (c) in diesem Fall Sinn.

Lösung zu Aufgabe 5.5 auf Seite 14

Beim Erstellen eines Heaps durch Absickern werden im schlimmsten Fall nur $O(n)$ Vertauschungen durchgeführt. Das Erstellen des Heaps durch Aufsteigen hingegen benötigt $O(n \log n)$ Vertauschungen und ist somit weniger effizient.

Der intuitive Grund für diesen Unterschied liegt in der Tatsache, dass die unteren Ebenen in einem Heap tendenziell viel mehr Knoten enthalten als die Ebenen weiter oben. Deshalb gibt es auch viel mehr Knoten, für welche der Weg nach unten bis zu einem Blatt viel kürzer ist als der Weg nach oben bis zur Wurzel. Das bedeutet dann aber auch, dass der Vorgang des Absickerns für diese Knoten viel weniger Vertauschungen benötigt als der Vorgang des Aufsteigens.

Lösung zu Aufgabe 5.6 auf Seite 14

Einfügen des Elements 7: Der zusätzliche Knoten 7 steigt vom letzten Eintrag ausgehend auf:

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

Schritt 0

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

Schritt 1

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

Schritt 2

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

Schritt 3

Entfernen des Maximums: Der letzte Knoten 2 sickert von der Wurzel ausgehend ab:

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

Schritt 4

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

Schritt 5

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

Schritt 6

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

Schritt 7