Eidgenössische Technische Hochschule Zürich
Professur für Informationstechnologie und Ausbildung
Universitätstrasse 6
CH-8092 Zürich

272-0101-00L Fachdidaktik Informatik 1

Heap, Heapify und Heapsort

Judith Beestermöller

2025

Inhaltsverzeichnis

1 Aufgabe: Vorarbeiten

1.1 Analyse

Leitidee:

Die Informatik stellt uns vor die Herausforderung, Daten so zu strukturieren, dass wir sie schnell und effizient bearbeiten können. Dabei gibt es eine Vielzahl von Datenstrukturen, die je nach Anwendungsfall unterschiedliche Vorteile bieten. Heaps sind eine spezielle Form der binären Baumstruktur, die sich durch ihre Fähigkeit auszeichnen, schnell auf das grösste oder kleinste Element zuzugreifen, was sie besonders wertvoll für Anwendungen wie Prioritätswarteschlangen, Task Scheduling und Graphenalgorithmen macht.

Im Gegensatz zu binären Suchbäumen, die komplexe Ordnungsanforderungen erfüllen, fokussieren sich Heaps auf eine deutlich einfachere Bedingung: Statt einer vollständigen Ordnung genügt eine gezielte Teilordnung – die sogenannte Heap-Eigenschaft –, bei der lediglich das grösste oder kleinste Element an der Wurzel steht. Diese Vereinfachung führt zu einer deutlichen Leistungssteigerung bei spezifischen Operationen wie der Extraktion des grössten oder kleinsten Elements, die in einem Heap in logarithmischer Zeit erfolgen kann, während in einem binären Suchbaum zusätzliche Traversierungen nötig sind.

Ein weiterer Vorteil von Heaps ist die Möglichkeit der effizienten Sortierung durch den Heapsort-Algorithmus, der eine Zeitkomplexität von $\mathcal{O}(n \log n)$ aufweist und Speicherplatz effizient nutzt. Diese Eigenschaften machen Heaps zu einer Schlüsseltechnologie für Szenarien, in denen eine schnelle Priorisierung oder Sortierung von Daten erforderlich ist.

Die Beschäftigung mit Heaps gibt den Schülerinnen und Schülern nicht nur praktische Werkzeuge an die Hand, um Daten effizient zu verarbeiten, sondern fördert auch ihr Verständnis für die Beziehung zwischen Anforderungen und Performance. Durch den Vergleich von Heaps und binären Suchbäumen erkennen die Schülerinnen und Schüler, wie die Vereinfachung von Anforderungen dazu beitragen kann, die Effizienz von Algorithmen zu steigern. Sie lernen, dass die Auswahl einer Datenstruktur massgeblich davon abhängt, welche Anforderungen und Einschränkungen im jeweiligen Anwendungsfall gegeben sind.

Das Lernen über Heaps ist daher nicht nur ein essenzieller Schritt, um algorithmisches Denken und das Verständnis für leistungsfähige Datenstrukturen zu entwickeln, sondern es schärft auch das Bewusstsein der Schülerinnen und Schüler für die Bedeutung einer klaren Problemdefinition in der Informatik.

Dispositionsziel:

Bewusstsein für die Vereinfachung von Anforderungen zur Effizienzsteigerung entwickeln: Die Schülerinnen und Schüler erkennen, dass die Reduktion von Ordnungsanforderungen – etwa durch den Einsatz einer gezielten Teilordnung wie bei Heaps im Vergleich zu binären Suchbäumen – zu einer signifikanten Verbesserung der Performance führen kann. Sie entwickeln ein Verständnis dafür, dass sich bestimmte Anforderungen durch einfachere Strukturen effizienter erfüllen lassen und dass die Wahl der Datenstruktur entscheidend von den spezifischen Zielen und Rahmenbedingungen eines Problems abhängt.

Bewusstsein für die Bedeutung der Heap-Datenstruktur entwickeln: Die Schülerinnen und Schüler entwickeln ein spezifisches Bewusstsein für die Vorteile von Heaps in der Informatik. Sie verstehen, dass Heaps eine effiziente Möglichkeit bieten, ständig auf das grösste oder kleinste Element zuzugreifen, und erkennen die Bedeutung dieser Eigenschaft in Anwendungen wie Prioritätswarteschlangen oder Task Scheduling. Darüber hinaus verstehen sie, dass die zugrundeliegende Teilordnung nicht nur effizient aufgebaut, sondern auch bei laufenden Änderungen (z. B. Einfügen oder Entfernen von Elementen) effizient aufrechterhalten werden kann.

Vergleich von Datenstrukturen kritisch reflektieren: Die Schülerinnen und Schüler entwickeln die Fähigkeit, verschiedene Datenstrukturen hinsichtlich ihrer Effizienz und Eignung für spezifische Aufgaben zu bewerten und zu entscheiden, welche Struktur für ein bestimmtes Problem am besten geeignet ist.

Verständnis für die Effizienz von Heap-Operationen entwickeln: Die Schülerinnen und Schüler verstehen, dass Heaps eine effiziente Zeitkomplexität für Operationen wie Einfügen und Löschen von Elementen bieten, was sie zu einer wichtigeren Datenstruktur für bestimmte Szenarien macht, insbesondere wenn die Daten regelmäßig aktualisiert und sortiert werden müssen. Sie erkennen, dass diese Effizienz im Vergleich zu anderen Datenstrukturen wie binären Suchbäumen oder Arrays deutlich vorteilhafter sein kann, insbesondere bei Operationen wie der Extraktion von Maximal- oder Minimalwerten.

Algorithmisches Verständnis für strukturelle Datenoperationen vertiefen: Durch die Auseinandersetzung mit Heaps entwickeln die Schülerinnen und Schüler ein tieferes Verständnis für die Struktur von Algorithmen, insbesondere bei der Heapify-Methode und Heapsort. Sie sind in der Lage, die Relevanz der Heapstruktur für die schnelle Datenmanipulation zu erkennen und verstehen die zugrunde liegende Logik hinter Algorithmen, die die Struktur aufrechterhalten.

Operationalisiertes Lernziel:

Die Schülerinnen und Schüler können:

1. Heapify-Algorithmus anwenden und erklären:
    - Eine unsortierte Liste in einen Heap umwandeln, indem sie den Heapify-Algorithmus auf eine Baumstruktur anwenden.

- Den Ablauf des Heapify-Algorithmus sowie die zugrunde liegenden Konzepte wie die Heap-Eigenschaft nachvollziehen und erläutern.

2. Heapsort-Algorithmus implementieren und analysieren:
   - Den Heapsort-Algorithmus als Programm umsetzen, einschließlich der Konstruktion und Nutzung eines Heaps zur Sortierung von Daten.
   - Die Zeit- und Speicherkomplexität des Heapsort-Algorithmus analysieren und mit anderen Sortieralgorithmen wie Quicksort oder Mergesort vergleichen.

3. Prioritätswarteschlangen mit Heaps modellieren:
   - Eine Prioritätswarteschlange unter Verwendung eines Heaps erstellen und deren Funktionsweise erklären.
   - Anwendungsbeispiele für Prioritätswarteschlangen benennen und die Vorteile eines Heaps in diesen Kontexten verdeutlichen.

4. Vergleich von Heaps und binären Suchbäumen vornehmen:
   - Die Effizienz von Operationen wie Suchen, Einfügen und Löschen in Heaps und binären Suchbäumen vergleichen.
   - Erklären, warum die Heap-Eigenschaft bei spezifischen Anwendungen effizienter ist als die strikte Ordnungsanforderung eines binären Suchbaums.

Ausgangslage und Vorwissen

Die SuS haben bereits das Kapitel zu Binärsuchbäumen (ausgearbeitet von J. Beestermöller) behandelt. Hier haben sie gelernt, dass die Effizienz von Binärsuchbäumen von der Struktur des Baumes abhängt. Die Laufzeit vieler Methoden ist proportional zur Höhe des Baumes. Daher ist der Binärsuchbaum am effizientesten, wenn der Baum eine Höhe von $\mathcal{O}(\log(n))$ hat, wobei n die Anzahl der Knoten im Baum ist.

Mit der Einführung von Heaps stellen wir den SuS eine Datenstruktur vor, welche garantiert, dass die Höhe des Baumes immer $\mathcal{O}(\log(n))$ ist. Gleichzeitig erfüllen Heaps nicht die gleichen Eigenschaften wie Binärsuchbäume. In dieser Hinsicht zeigt der Vergleich von Binärsuchbäumen und Heaps einen Trade-off: durch das Vereinfachen der Anforderungen an die Datenstruktur kann eine höhere Effizienz erreicht werden.

Anhand von Alltagsbeispielen lernen die SuS zusätzlich, dass die Heap-Datenstruktur viele Anwendungen hat und keinesfalls durch die Vereinfachung „nutzlos“ wird.

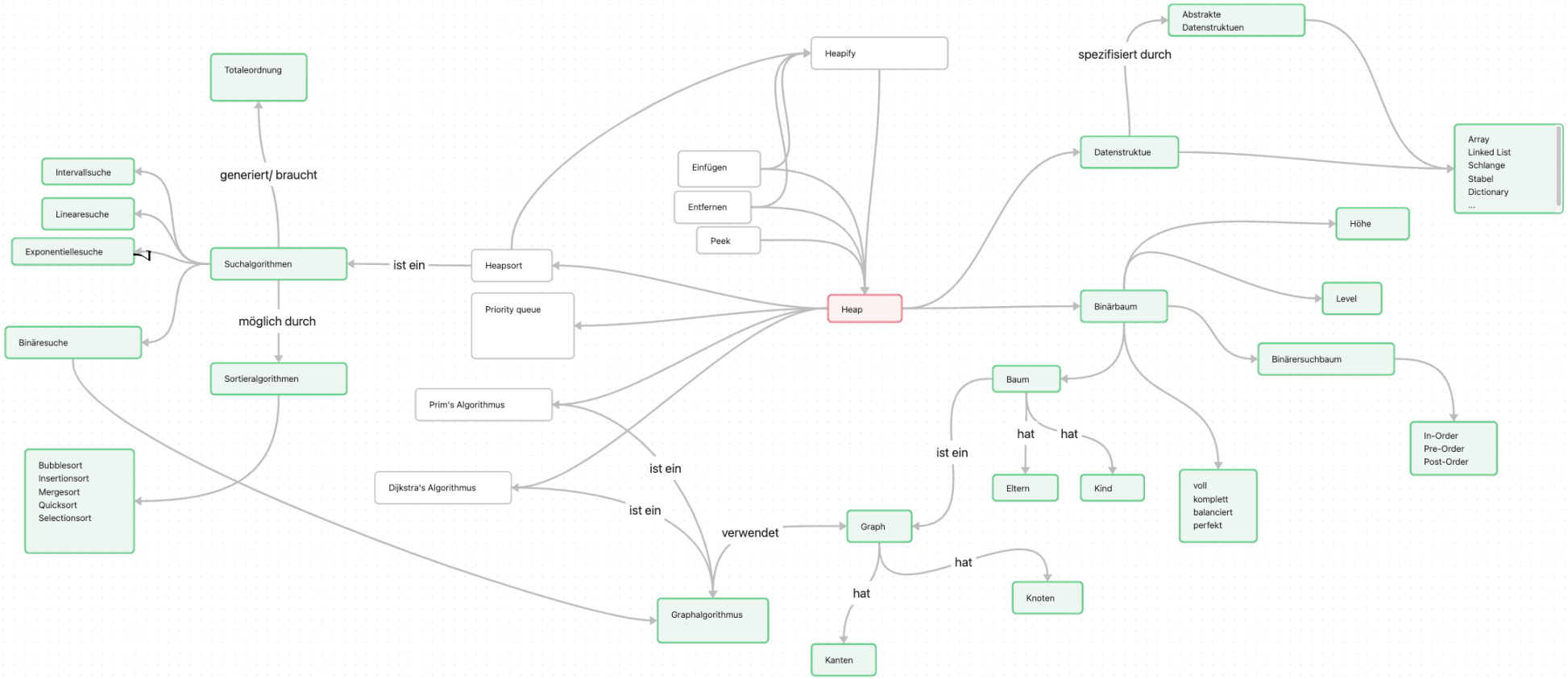
Auf der nachfolgenden Seite haben wir eine Konzeptmap erstellt, die die Konzepte der Lerneinheit darstellt und im Zusammenhang mit vorherigen Lektionen bringt. Konzepte, die bereits behandelt wurden, sind grün hinterlegt.

1.2 Grober Ablauf

Wir stellen die SuS vor die Herausforderung, die abstrakte Datenstruktur für die Notaufnahme eines Krankenhauses zu definieren. Hierbei sollte ihnen auffallen, dass wir nicht die komplette Reihenfolge der Patienten brauchen, sondern lediglich in der Lage sein müssen, den nächsten Patienten aufzurufen. Gleichzeitig sollte ihnen auffallen, dass permanent neue Patienten hinzukommen und andere Patienten entfernt werden.

Nachdem die SuS diese Anforderungen an die Datenstruktur erkannt haben, überlegen wir, welche Laufzeiten wir für die verschiedenen Methoden erwarten können. Zusätzlich erarbeiten wir die Laufzeiten, die wir mit den bereits bekannten Datenstrukturen (z. B. Array, BST) erhalten würden.

Im letzten Abschnitt definieren wir die Heap-Datenstruktur und stellen den Heapify-Algorithmus vor. Hierbei gehen wir insbesondere auf die Laufzeit von Heapify ein. Danach behandeln wir die Methoden Einfügen, Peek und Entfernen. Hier werden auch einige Programmieraufgaben gemacht.


Totaleordnung
generiert/ braucht
Intervallsuche
Linearesuche
Exponentiellesuche
Suchalgorithmen
ist ein
Heapsort
Heapify
Einfügen
Entfernen
Peek
Priority queue
Heap
möglich durch
Binäresuche
Sortieralgorithmen
Bubblesort
Insertionsort
Mergesort
Quicksort
Selectionsort
Prim's Algorithmus
Dijkstra's Algorithmus
ist ein
ist ein
verwendet
Graph
Graphalgorithmus
hat
Kanten
hat
Knoten
spezifisiert durch
Abstrakte
Datenstruktuen
Datenstruktue
Array
Linked List
Schlange
Stabel
Dictionary
...
Höhe
Level
Binärbaum
Binärersuchbaum
In-Order
Pre-Order
Post-Order
Baum
ist ein
hat
hat
Eltern
Kind
voll
komplett
balanciert
perfekt

2 Aufgabe: LPU

2.1 Heaps

Wir haben uns zuletzt mit Binärsuchbäumen beschäftigt. Eine wichtige Beobachtung war, dass diese am effizientesten sind, wenn ihre Höhe $\mathcal{O}(\log(n))$ beträgt, wobei n die Anzahl der Knoten im Binärbaum angibt.

Als Nächstes wollen wir uns mit einer Datenstruktur beschäftigen, die ebenfalls auf einem Binärbaum aufbaut, dabei aber garantiert wird, dass die Höhe des Baumes immer logarithmisch zur Anzahl der Knoten im Baum ist. Dies hat zur Folge, dass alle Operationen, die wir auf der Datenstruktur ausüben, maximal eine Laufzeit von $\mathcal{O}(\log(n))$ haben. Damit dies möglich ist, können wir nicht alle Eigenschaften eines Binärsuchbaums erhalten. Wir wollen zunächst eine Anwendung der Datenstruktur beschreiben, und anhand dieser die Spezifikation der abstrakten Datenstruktur erarbeiten.

Stell dir vor, du befindest dich in der Notaufnahme eines Krankenhauses. Deine Aufgabe ist es, eine Struktur über die Patienten und Patientinnen wie folgt zu erarbeiten:

- Wann immer ein Arzt frei wird, musst du in der Lage sein, die Patientin oder den Patienten mit der höchsten Dringlichkeit aufzurufen. Sobald eine Patientin oder ein Patient von einem Arzt gesehen wird, kannst du sie oder ihn aus deiner Struktur, die eine Warteschlange darstellt, entfernen.
- Gelegentlich hilft auch das Pflegepersonal. Dieses spricht jeweils mit der Patientin oder dem Patienten mit höchster Dringlichkeit. Du musst den Pflegekräften also sagen können, wer die höchste Dringlichkeit hat. Allerdings werden die Patienten danach immer noch von einem Arzt gesehen.
- Es kommen auch ständig neue Patienten und Patientinnen in die Notaufnahme. Diese musst du anhand ihrer Dringlichkeit in die Struktur einfügen. Dabei ist alleine die Dringlichkeit, nicht aber die Ankunftszeit entscheidend. So werden zum Beispiel Patienten, die in Lebensgefahr schweben, immer zuerst behandelt, als solche, die sich einen Arm gebrochen haben.

Übung 2.1 ***Notaufnahme als ADT*** Lösung

Stell dir vor, die Dringlichkeit ist als Zahl bemessen, wobei gilt: Je höher die Zahl, desto höher die Dringlichkeit.
Überlege dir eine Möglichkeit, wie du die Patientinnen und Patienten in einer

Liste so verwalten könntest, dass du jederzeit weisst, wer gerade am dringendsten behandelt werden sollte.

- Wie kannst du sicherstellen, dass die Person mit der höchsten Dringlichkeit immer leicht zu finden ist?
- Wo und wie müsstest du neue Personen in der Liste einfügen?
- Was passiert, wenn du viele neue Patientinnen und Patienten aufnehmen musst – wie aufwendig wird das Einfügen dann?

Entwickle eine erste Version einer Datenstruktur, bei der du die Patientinnen und Patienten nach Dringlichkeit geordnet verwalten kannst. Gib deiner Struktur einen Namen und beschreibe mindestens drei Operationen, die sie können muss (z. B. `fügeEin(...)` oder `behandleDringendste()`). Schreibe auf, was jede Operation genau tun soll.

In Übung 2.1 hast du eine Lösung mit einer sortierten Liste entwickelt. Überlege nun, welche Probleme dabei auftreten können, wenn sehr viele Personen gleichzeitig behandelt oder neu aufgenommen werden müssen.

- Welche Operationen sind besonders langsam oder aufwendig?
- Was würde passieren, wenn du jede neue Person am richtigen Platz einsortieren müsstest – auch bei hundert oder tausend Patientinnen und Patienten?

Es gibt Datenstrukturen, die genau für solche Fälle entwickelt wurden: Eine davon ist der sogenannte Heap. Dabei handelt es sich um eine Struktur, in der stets das Element mit der höchsten Dringlichkeit an der Spitze steht – ohne dass die gesamte Struktur sortiert sein muss.

Übung 2.2 *Notaufnahme als ADT* Lösung

Zeige, dass es für die beschriebene Warteschlange in der Notaufnahme ausreichend ist, wenn die Datenstruktur nur sicherstellt, dass die Person mit der höchsten Dringlichkeit ganz vorne steht – ohne dass alle anderen korrekt einsortiert sind? Wie könnten die folgenden Operationen effizient in einer Liste umgesetzt werden?

- `fügeEin(p)` – fügt eine neue Person mit Dringlichkeit `p` hinzu
- `gibDringendste()` – gibt die Person mit der höchsten Dringlichkeit zurück, entfernt sie aber nicht
- `behandleDringendste()` – gibt die Person mit der höchsten Dringlichkeit zurück und entfernt sie aus der Struktur

Die abstrakte Datenstruktur, die wir uns in der letzten Aufgabe angeschaut haben, wird als Heap bezeichnet. Die genaue Spezifikation lautet wie folgt:

Konzepte und Begriffe:
Die Methoden eines Heap:

- Einfügen: Fügt ein Element hinzu
- Extract: Entfernt das maximale (Max-Heap) oder minimale (Min-Heap) Element
- Peek: Gibt das maximale (Max-Heap) oder minimale (Min-Heap) Element zurück, ohne es zu entfernen.

Die Methoden eines Heaps können auch mit den uns bereits bekannten Datenstrukturen implementiert werden. In der nächsten Aufgabe wirst du aufgefordert, dir zu überlegen, welche Laufzeiten diese Methoden in uns bekannten Datenstrukturen haben.

Übung 2.3 ***Heap Methoden in anderen Datenstrukturen*** Lösung

Überlege dir die Worstcaselaufzeit der drei Heap Methoden, wenn du sie mit den folgenden Datenstrukturen umsetzt.

- Array (unsortiert)
- Array (sortiert)
- Linked List (unsortiert)
- Linked List (sortiert)
- Binärersuchbaum

Die Datenstruktur, die wir in dieser Lektion kennenlernen wollen, heisst Heap, und hat die folgenden Eigenschaften:

Konzepte und Begriffe:
Ein **Heap** ist eine spezialisierte, baumbasierte Datenstruktur, die zwei zentrale Eigenschaften erfüllt:

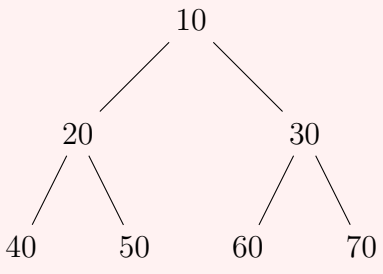
1. Heap-Eigenschaft: Der Wert jedes Knotens muss bestimmten Regeln im Verhältnis zu seinen Kindknoten genügen:
   - **Max-Heap**: Der Wert eines Elternknotens ist grösser oder gleich den Werten seiner Kindknoten.
   - **Min-Heap**: Der Wert eines Elternknotens ist kleiner oder gleich den Werten seiner Kindknoten.
2. Eigenschaft eines vollständigen Binärbaums: Ein Heap ist ein vollständiger Binärbaum, das bedeutet:
   - Alle Ebenen des Baums sind vollständig gefüllt, mit Ausnahme der letzten Ebene.
   - In der letzten Ebene sind alle Knoten von links nach rechts angeordnet.

In der nächsten Aufgabe wird dein Verständnis der obigen Definition getestet.

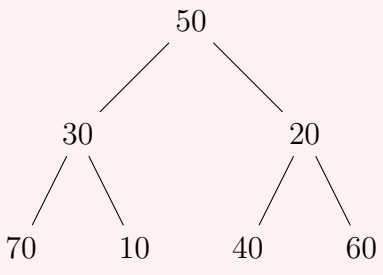
Übung 2.4 ***Heap Methoden in anderen Datenstrukturen*** Lösung

Entscheide für jeden der nachfolgenden Graphen, ob es sich um eine Heap handelt. Begründe deine Antwort. Hinweis, bei den Graphen kann es sich sowohl um Min- als auch um Max-Heaps handeln.

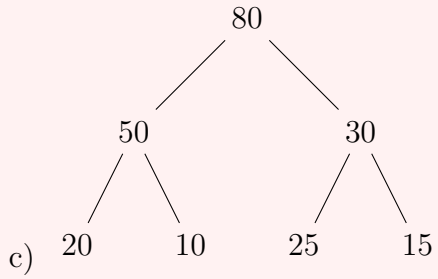
a)



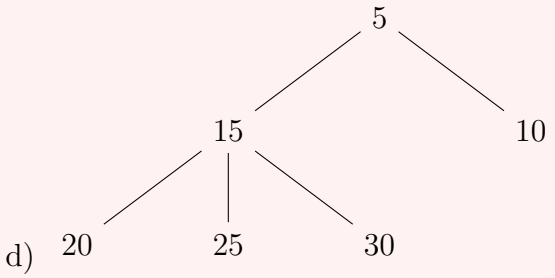
b)



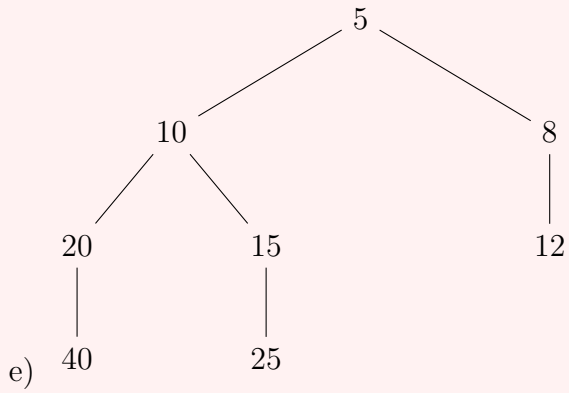
c)



d)



e)



Bei sehr grossen Bäumen, kann es mühsam werden zu überprüfen, ob es sich um einen Heap handelt. Wir wollen daher einen Algorithmus schreiben, der für uns überprüft, ob ein gegebener Binärbaum die Eigenschaften eines Heap erfüllt.

Um es uns etwas einfacher zu machen, werden wir zwei Funktionen in unserem Code implementieren: Eine, die die Heap-Eigenschaft überprüft, und eine, die überprüft, ob es sich bei dem Binärbaum um einen vollständigen Binärbaum handelt.

Um die Heap-Eigenschaft zu überprüfen, muss jeder einzelne Knoten des Baums kontrolliert werden. Die Überprüfung eines Knotens ist nicht allzu schwierig: Es genügt, den Knoten und seine direkten Kinder zu betrachten.

Übung 2.5 ***Überprüfen der Heap Eigenschaft*** **Lösung**

Schreibe die Methode checkMinHeapProperty die als Input einen Knoten x nimmt und ausgibt, ob der Binärbaum mit Wurzel x die Min-Heap-Eigenschaft erfüllt oder nicht.

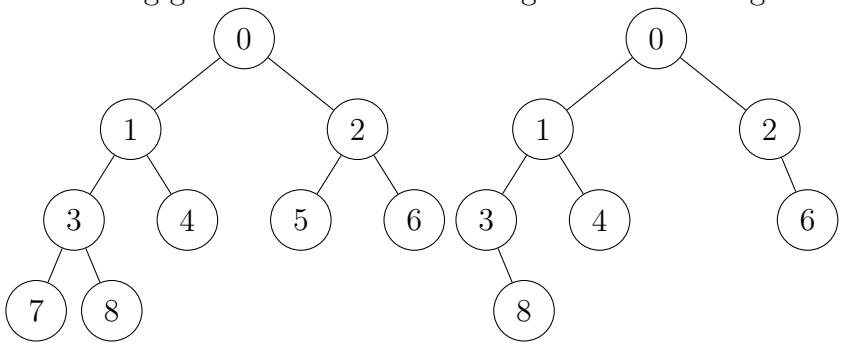
```python
class Node:
    """
    Die Node-Klasse stellt einen Knoten in einem Binärbaum
    ↪ dar.

    Attribute:
    - value: Der Wert des Knotens (der Wert des Knotens im
    ↪ Heap).
    - left: Verweis auf das linke Kind des Knotens (None, wenn
    ↪ kein linkes Kind existiert).
    - right: Verweis auf das rechte Kind des Knotens (None,
    ↪ wenn kein rechtes Kind existiert).
    """
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def checkMinHeapProperty(x):
        """
        Überprüft, ob der Binärbaum mit Wurzel x die
        ↪ Heap-Eigenschaft erfüllt.

        Parameter:
        - x: Der Knoten, der die Wurzel des zu überprüfenden
        ↪ Baums darstellt.

        Rückgabewert:
        - True, wenn der Baum die Heap-Eigenschaft erfüllt,
        ↪ andernfalls False.
        """
```

Als Zweites müssen wir überprüfen, ob es sich bei dem gegebenen Binärbaum um einen vollständigen Binärbaum handelt. In der nächsten Aufgabe erhältst du einen Hinweis, wie du überprüfen kannst, ob ein gegebener Binärbaum ein vollständiger Binärbaum ist. Wenn du bereits eine Idee für die Implementierung hast, kannst du aber auch direkt zur Programmieraufgabe gehen.

Bei einer Level-Order-Traversierung werden die Knoten eines Baumes nach ihrer Höhe (Level) besucht. Zuerst wird die Wurzel auf Level 0 betrachtet, dann folgen die Knoten auf Level 1 (die direkten Kinder der Wurzel), dann alle Knoten auf Level 2 und so weiter. Innerhalb eines Levels werden die Knoten in der Reihenfolge besucht, in der sie im Baum von links nach rechts erscheinen.

Jeder Knoten im Baum hat einen Index, der seine Position in der Reihenfolge angibt, wie er bei der Traversierung besucht wird. Die Wurzel hat den Index 0, ihr linkes Kind den Index 1 und das rechte Kind den Index 2. Die Kinder eines Knotens erhalten Indizes basierend auf der Position ihres Elternteils: Das linke Kind eines Knotens hat den Index $2 * Elternindex + 1$ und das rechte Kind den Index $2 * Elternindex + 2$. Wenn jedoch Knoten fehlen, werden die entsprechenden Indizes übersprungen, aber die Reihenfolge der Indizes bleibt unverändert. Das bedeutet, dass, auch wenn ein Knoten an einer bestimmten Stelle im Baum fehlt, die Indizes der verbleibenden Knoten weiterhin der Formel folgen.

In den zwei folgenden Bäumen haben wir die Indices der Level-Order-Traversierung angezeigt. Im linken, vollständigen Baum wird einfach hochgezählt, im nicht vollständigen Baum werden die Indices der fehlenden Knoten übersprungen. Die Level-Order-Traversierung gibt die Knoten in aufsteigender Reihenfolge wieder.



Übung 2.6 ***Vollständiger Baum und Level Order Traversierung*** Lösung

Zeige, dass in einem vollständigen Baum der Index eines Knotens nie grösser ist als $n - 1$ wobei n die Anzahl Knoten im Baum zählt.

Die Level-Order-Traversierung kann uns helfen zu überprüfen, ob ein Binärbaum vollständig ist. Kannst du erkennen, wie?

Übung 2.7 ***Überprüfen vollständiger Binärbaum*** Lösung

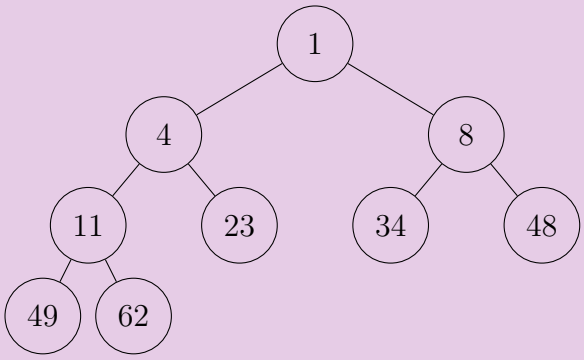
Schreibe die Methode isCompleteBinaryTree die als Input einen Knoten x nimmt und ausgibt, ob der Binärbaum mit Wurzel x die Min-Heap-Eigenschaft erfüllt oder nicht.

```
class Node:
    """
    Die Node-Klasse stellt einen Knoten in einem Binärbaum
    ↪  dar.

    Attribute:
```

```
        - value: Der Wert des Knotens (der Wert des Knotens im
        ↪  Heap).
        - left: Verweis auf das linke Kind des Knotens (None, wenn
        ↪  kein linkes Kind existiert).
        - right: Verweis auf das rechte Kind des Knotens (None,
        ↪  wenn kein rechtes Kind existiert).
        """
        def __init__(self, value):
            self.value = value
            self.left = None
            self.right = None

        def isCompleteBinaryTree(                              ):
            """
            Überprüft rekursiv, ob der Baum ein vollständiger
            ↪  Binärbaum ist.

            Parameter:




            Rückgabewert:
            - True, wenn der Baum vollständig ist.
            - False, wenn der Baum nicht vollständig ist.
            """
```

Wenn wir mit Heaps arbeiten, wissen wir, dass es sich um vollständige Binärbäume handelt. Man kann in diesem Fall den Baum auch als Array darstellen, anstatt einer Menge von Knoten. Dies hat insbesondere den Vorteil, dass man auf jedes Element im Baum direkt zugreifen kann und nicht den Baum von der Wurzel aus traversieren muss.

Konzepte und Begriffe:
Sei $a = [1, 4, 8, 11, 23, 34, 48, 49, 62]$ ein Array. Der dazugehörige vollständige Binärbaum ist

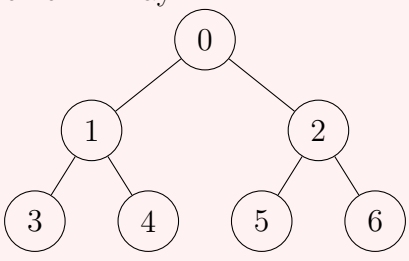


Wichtig ist zu bemerken, dass wir die Kanten des Baumes nicht mehr explizit speichern, sondern dass sich diese daraus ergeben, dass der Baum vollständig ist und wir die Werte der Knoten in Level-Order-Reihenfolge in dem Array speichern.

Für den Rest dieser Lektion gehen wir von der Array-Darstellung der Binärbäume und Heaps aus.

Übung 2.8 ***Kinder finden*** Lösung

In der Node-Darstellung eines Binärbaumes können wir die Kinder eines Knotens durch das Folgen der Pointer `left` bzw. `right` erreichen. Bei der Array-Darstellung sind diese Pointer nicht mehr vorhanden. Stattdessen können wir anhand des Index direkt auf die Elemente des Arrays zugreifen.
Ihre Aufgabe ist es, die Formeln $l(i)$ und $r(i)$ zu entwickeln, mit denen Sie anhand des Index i des Elternknotens auf das linke und das rechte Kind zugreifen können. Wenn beispielsweise $0 \leq i < n$ der Index des Elternknotens ist, dann soll $l(i)$ den Index des linken Kindes im Array angeben und $r(i)$ den Index des rechten Kindes.
Veranschaulichung und Gesetzmässigkeit
Um eine Gesetzmässigkeit für die Formeln $l(i)$ und $r(i)$ zu finden, betrachten wir zunächst ein konkretes Beispiel eines Binärbaumes und dessen Darstellung in einem Array.



Obige Figur zeigt ein Beispiel eines Binärbaumes mit Knoten-Indizes in einer Array-Darstellung.

Betrachten Sie den obigen Binärbaum, bei dem die Zahlen in den Knoten die Indizes repräsentieren, unter der Annahme, dass der Baum Level-für-Level in ein Array gespeichert wird (beginnend bei Index 0). Füllen Sie die folgende Tabelle aus, indem Sie die Indizes der linken und rechten Kinder für die gegebenen Elternknoten ermitteln:

Elternknoten-Index (i)	**Index des linken Kindes ($l(i)$)**	**Index des rechten Kindes ($r(i)$)**
0		
1		
2		
3		
...	...	...

Analysieren Sie die Beziehungen zwischen i, $l(i)$ und $r(i)$ in der ausgefüllten Tabelle, um die allgemeinen Formeln für $l(i)$ und $r(i)$ abzuleiten.

Bei der Datenstruktur des Binärsuchbaums haben wir gelernt, dass es mehr als einen Binärsuchbaum für den gleichen Datensatz gibt. Gleichzeitig gab es für jeden Datensatz jeweils nur einen einzigen Binärsuchbaum, der vollständig war. Wir wollen nun untersuchen, ob es verschiedene Heaps für den gleichen Datensatz gibt.

Übung 2.9 ***Wie viele Heaps?*** Lösung

Zeichne alle Min-Heaps für die Bäume mit Knoten

- 1
- 1,2
- 1,2,3
- 1,2,3,4
- 1,2,3,4,5

Kannst du eine Regelmässigkeit erkennen?

In der nächsten Aufgabe sollst du berechnen, wie viele Heaps es für n Datenpunkte gibt.

Was du gelernt hast:
Heaps sind eine baumbasierte Datenstruktur, bei der jeweils das Element mit dem höchsten (Max-Heap) oder dem niedrigsten (Min-Heap) Wert die Wurzel bildet. Heaps sind immer vollständige Binärbäume. Anders als bei einem Binärsuchbaum, für den es nur einen vollständigen Binärsuchbaum pro Datensatz gibt, kann es viele Heaps für den gleichen Datensatz geben.

2.2 Heap Methoden

Nachdem wir verstanden haben, was Heaps sind, wollen wir als Nächstes verstehen, wie die Methoden Einfügen, Extract und Peek durchgeführt werden können. Die Methoden Einfügen und Extract werden uns insbesondere dabei helfen, Heaps aus einem gegebenen Datensatz zu bauen.

2.2.1 Peek

Die wahrscheinlich einfachste der drei Methoden ist die Peek-Methode. Die Spezifikation der Peek-Methode ist

- *peek()* - Gibt das Element mit dem höchsten (Max-Heap) bzw. kleinsten (Min-Heap) Wert zurück.

Bei der Definition ist der Knoten mit dem höchsten/kleinsten Wert immer in der Wurzel zu finden. Bei der Array-Darstellung hat die Wurzel den Index 0. Somit können wir einfach $A[0]$ zurückgeben.

```
class Heap:
    def __init__(self):
        self.heap = []

    def peek(self):
        """Returns the top element of the heap without removing it."""
        if len(self.heap) == 0:
            raise IndexError("Heap is empty")
        return self.heap[0]  # The root element in the array
        ↪ representation
```

Die Methoden Einfügen und Extract sind etwas schwieriger zu implementieren, da wir sicherstellen müssen, dass die Heap-Eigenschaften nach dem Ausführen der Methode weiterhin Bestand haben.

2.2.2 Einfügen

Wir wollen uns zunächst mit dem Einfügen in einen Heap beschäftigen.

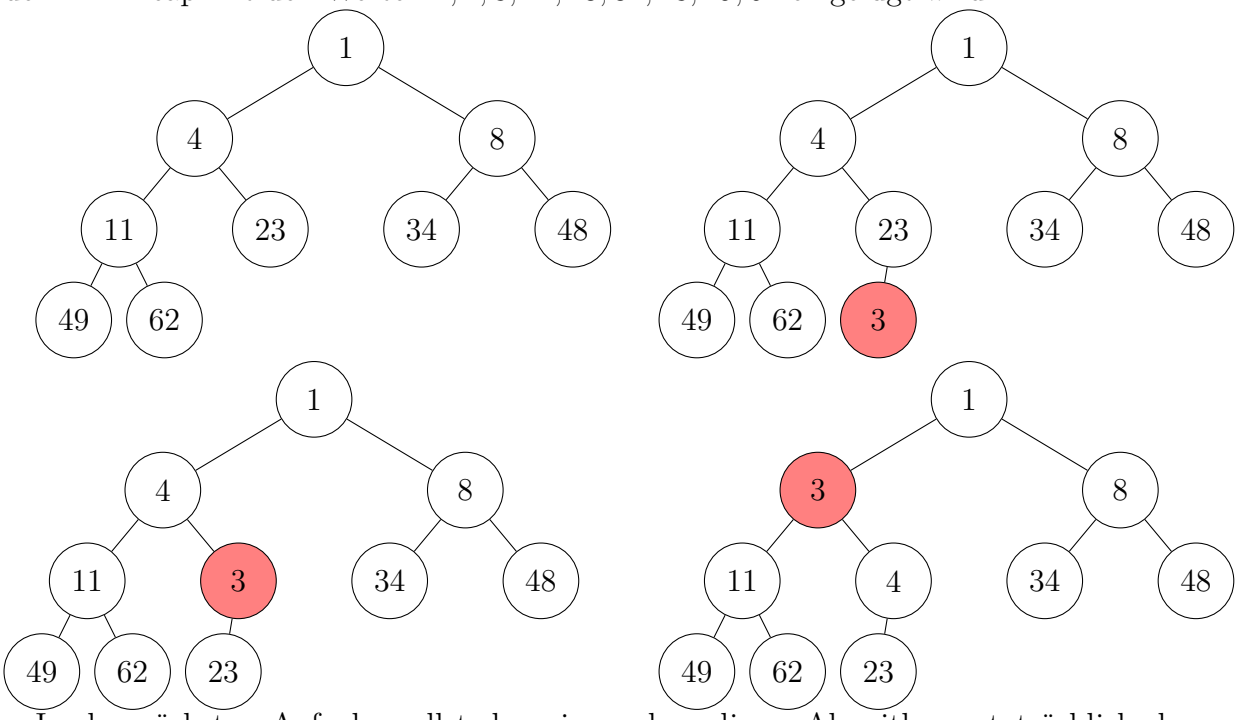
- *add(x)* - Fügt das Element mit Wert x in die Heap ein.

Beim Einfügen in einen Binärsuchbaum haben wir zunächst überprüft, ob das Element, welches hinzugefügt werden soll, bereits im Baum vorhanden ist. Dafür haben wir die Position, an der das Element stehen würde, bestimmt. Wenn das Element nicht vorhanden war, haben wir es an dieser Position hinzugefügt. Wir haben dabei gesehen, dass, wenn ein Element hinzugefügt wird, es immer als Blatt hinzugefügt wird.

Bei einem Heap darf der gleiche Wert auch mehrmals vorkommen, daher müssen wir nicht überprüfen, ob der Wert bereits vorhanden ist. Dennoch müssen wir feststellen, an welche Stelle das neue Element gehört. Um sicherzugehen, dass der Binärbaum nach dem Einfügen weiterhin vollständig ist, fügen wir das Element zunächst als Blatt ein, so dass der Baum weiterhin vollständig ist.

Im nächsten Schritt vergleichen wir den Wert des neuen Knotens x mit dem Wert seines Elternknotens $p(x)$. Haben wir einen Min-Heap und ist $value(x) < value(p(x))$, so tauschen wir x und $p(x)$. Handelt es sich bei dem Heap um einen Max-Heap, so tauschen wir x und $p(x)$ immer dann, wenn $value(x) > p(x)$ ist. Diesen Prozess wiederholen wir so lange, bis x entweder die Wurzel des Baumes ist oder x und $p(x)$ nicht mehr getauscht werden sollten.

Die nächste Abbildung zeigt eine Visualisierung des Prozesses, wenn der Wert 3 in den Min-Heap mit den Werten $1, 4, 8, 11, 23, 34, 48, 49, 62$ eingefügt wird.



In der nächsten Aufgabe sollst du zeigen, dass dieser Algorithmus tatsächlich das gewünschte Resultat liefert.

Übung 2.10 ***Einfügen in Heaps*** Lösung

Man kann den Prozess des Einfügens in eine Min-Heap mit folgendem Pseudocode beschreiben.

Füge x als letztes Element in die Array hinzu.
While $value(x) < value(p(x))$:
    Swap x and p(x)

Beweise, dass dieser Pseudocode korrekt ist, das heisst, dass das Ergebnis eine Min-Heap ist, die zusätzlich zu den ursprünglichen Werten auch den Wert x enthält.

Nun sind deine Programmierkünste gefragt, um die Methode zu implementieren.

Übung 2.11 ***Einfügen in Heaps*** Lösung

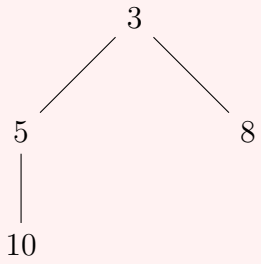
Implementiere die Einfüge Methode.

```
class MinHeap:
    def \_\_init\_\_(self):
        """
        Initializes an empty Min-Heap.
        """
        self.heap = []

    def add(self, value):
        """
        Adds a new value to the Min-Heap while maintaining the
        ↪  heap property.

        Parameters:
        value: The value to be added to the heap.
        """
```

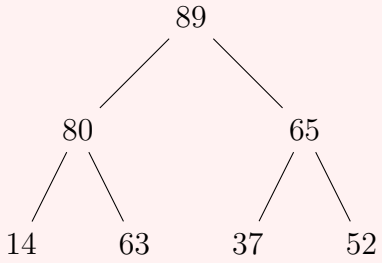
Du kannst dein Verständnis der Methode noch mehr üben, indem du die nächste Aufgabe löst.

Übung 2.12 ***Einfügen in Heaps*** Lösung

a) Füge die Werte 7, 4, 1 nacheinander in folgende Heap ein. Was ist die Finale Heap?



b) Füge die Werte 3, 21, 84 nacheinander in folgende Heap ein. Was ist die Finale Heap? Macht es einen Unterschied in welcher Reihenfolge du die Werte hinzufügst?



2.2.3 Extract

Als Nächstes wollen wir uns die Extract-Methode des Heaps genauer ansehen. Die Spezifikation ist wie folgt:

- *remove()* - Entfernt den maximalen (Max-Heap) bzw. minimalen (Min-Heap) Wert aus dem Heap und gibt diesen zurück.

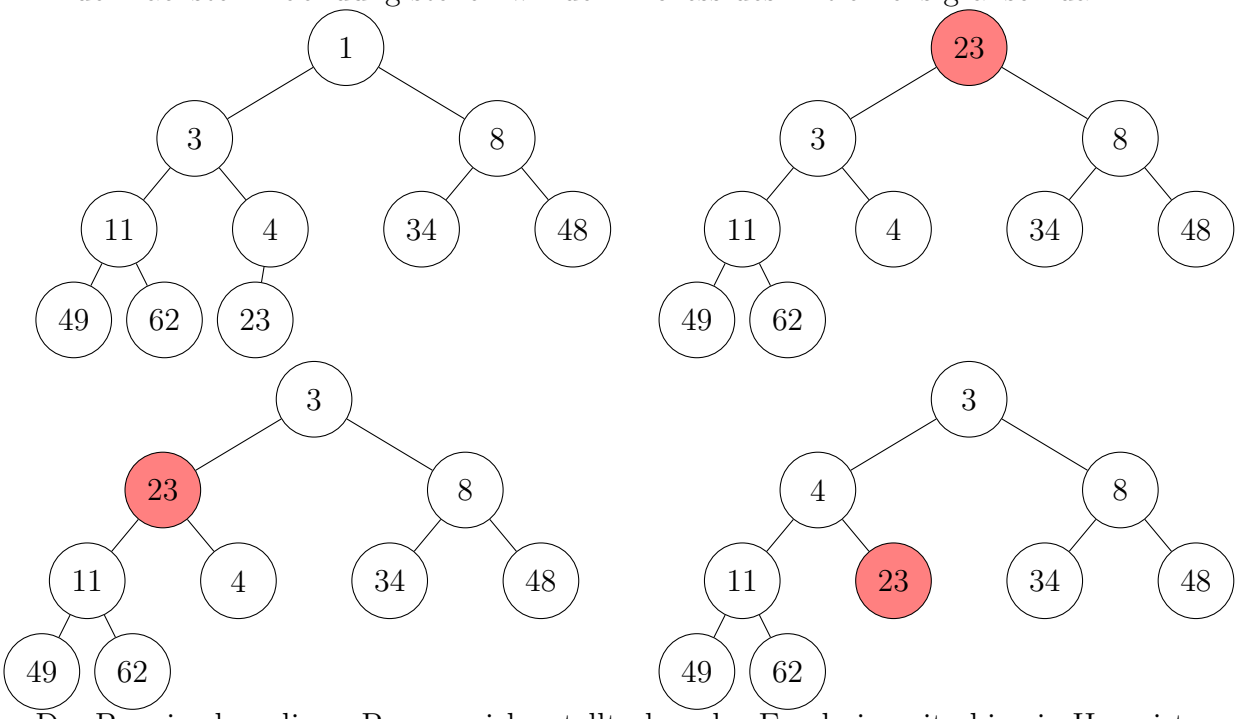
Wir haben bereits bei der Peek-Methode gelernt, dass das Element, welches wir suchen, in der Wurzel zu finden ist. Die Schwierigkeit besteht darin, die Eigenschaften des Heaps wiederherzustellen, nachdem wir diesen Wert entfernt haben.

Wir gehen wieder ähnlich vor, wie auch beim Entfernen in einem Binärsuchbaum. Wir ersetzen den Knoten, den wir entfernen, zunächst mit einem Blatt. Um sicherzugehen, dass der Heap ein vollständiger Baum bleibt, ersetzen wir die Wurzel immer mit dem am weitesten rechts stehenden Blatt auf dem untersten Level.

Im nächsten Schritt müssen wir die Heap-Eigenschaft wiederherstellen. Dieser Prozess ähnelt dem Wiederherstellen der Heap-Eigenschaft nach dem Einfügen eines neuen Elements in den Heap. Nach dem Einfügen eines neuen Elements als Blatt stand dieses Element zunächst an einer falschen Position im Heap, und wir mussten es von unten nach oben an die richtige Stelle verschieben. Beim Ersetzen der Wurzel mit einem Blattknoten beim Extracten steht die Wurzel zunächst an einer falschen Position, und wir verschieben sie von oben nach unten, bis die Heap-Eigenschaft wiederhergestellt ist. Hierzu vergleichen wir die Wurzel mit ihren Kindern (sofern zwei vorhanden) und wählen das grössere (Max-Heap) bzw. kleinere (Min-Heap) Kind und tauschen die Wurzel mit diesem Knoten. Diesen Prozess wiederholen wir so lange, bis der Knoten entweder

wieder ein Blatt im Baum ist oder grösser (Max-Heap) bzw. kleiner (Min-Heap) als seine Kinder ist.

In der nächsten Abbildung stellen wir den Prozess des Entfernens grafisch dar.



Der Beweis, dass dieser Prozess sicherstellt, dass das Ergebnis weiterhin ein Heap ist, ist ähnlich zum Korrektheitsbeweis des Hinzufügens. Es lohnt sich, wenn du den Beweis noch einmal für diesen Algorithmus anpasst.

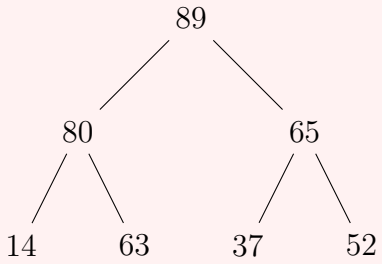
Die nächsten zwei Aufgaben überprüfen dein Verständnis der Entfern-Methode und von Heaps generell.

Übung 2.13 ***Entfernen in Heaps*** Lösung

Tommy behauptet, dass nach dem Wiederherstellen der Heap beim Entfernen, der Blattknoten welcher zunächst die Wurzel ersetzt, am Ende wieder ein Blatt Knoten ist. Muss dies tatsächlich so sein?

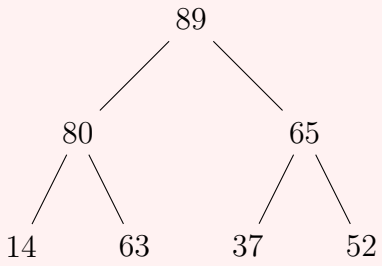
Übung 2.14 ***Heap Methoden anwenden*** Lösung

a) Wie sieht folgende Heap aus, wenn folgende Operationen nacheinander ausgeführt werden. remove(), remove(), remove()



b) Wie sieht die Finale Heap aus, wenn du nacheinander folgende Methoden aufrufst:

- add(3)
- remove()
- add(26)
- remove()
- remove()



Als Nächstes sind deine Programmierkünste gefragt: Schaffst du es, die Entfern-Methode selbständig zu implementieren?

Übung 2.15 ***Entfernen in Heaps*** Lösung

Implementiere die Extrakt Methode.

```
class Heap:

    def __init__(self, elements=None):
        """
        Initialisiert den Heap mit einer Liste von Elementen.

        HINWEIS: Die übergebene Liste muss bereits die
        ↪  Eigenschaften eines Heaps erfüllen.
```

```
        :param elements: (list) Eine Liste von Zahlen zur
        ↪  Initialisierung des Heaps.
        """
        self.heap = elements if elements else []

    def remove(self):
        """
        Entfernt das größte Element (die Wurzel) aus dem Max-Heap
        und stellt die Heap-Eigenschaft wieder her.

        :return: (int) Das entfernte größte Element oder None,
        ↪  falls der Heap leer ist.
        """
```

2.2.4 Laufzeiten

Die Methoden peek(), remove() und add() in einem Heap haben unterschiedliche Laufzeiten, die sich aus der Struktur des Heaps ergeben.

Versuche, die Laufzeiten der drei Methoden selbst herzuleiten.

Übung 2.16 ***Laufzeiten*** Lösung

Was ist die Worst-Case-Laufzeit, (das heisst die asymptotische maximale Anzahl an Operationen gemessen in der Anzahl Knoten in der Heap)

a) der Peek()-Methode?

b) der Remove()-Methode?

c) der Add()-Methode?

Begründe deine Antwort.

Zu diesem Zeitpunkt lohnt es sich, den Heap noch einmal mit anderen Datenstrukturen zu vergleichen. Wir haben bereits in Übung 2.3 die Laufzeiten anderer Datenstrukturen analysiert, die man für Heap-Operationen nutzen kann. Dir sollte auffallen, dass der Heap deutlich schnellere Worst-Case-Laufzeiten hat als diese anderen Datenstrukturen.

Was du gelernt hast: Wir haben gelernt, die Peek-, Einfügen- und Entfernen-Methoden zu implementieren und ihre Laufzeiten zu analysieren. Beim Vergleich mit anderen Datenstrukturen haben wir gesehen, dass der Heap für diese drei Methoden sehr effizient ist. Gleichzeitig dürfen wir nicht vergessen, dass einige Datenstrukturen weitere Methoden bieten, die mit einem Heap nicht oder nur schwer umsetzbar sind. Daher ist es wichtig, die Anforderungen an unser Programm genau zu analysieren, um die effizienteste Datenstruktur auszuwählen.

2.3 Heaps bauen - Heapify

Zu diesem Zeitpunkt sind wir in der Lage, einen bestehenden Heap zu verwenden und zu erhalten. Wir können auch einen Heap bauen, indem wir die Datenpunkte nacheinander mittels der Einfügemethode in einen zunächst leeren Heap hinzufügen. Im letzten Abschnitt wollen wir uns mit einer Methode beschäftigen, die es uns erlaubt, einen Heap auf noch effizientere Weise zu bauen. Wir wollen zunächst analysieren, mit welcher Laufzeit wir uns vergleichen.

Übung 2.17 ***Laufzeiten*** Lösung

Angenommen du hast einen Datensatz bestehend aus n Datenpunkten (unsortiert). Was ist die asymptotische Laufzeit, wenn du einen Heap erstellst, indem du die Datenpunkte nacheinander in einen zunächst leeren Heap hinzufügst?

Eine Alternative zu der eben beschriebenen Methode besteht darin, zunächst den gesamten Datensatz unsortiert in einen vollständigen Baum einzufügen und dann den Heap von unten nach oben wiederherzustellen.

Dieser Ansatz benutzt die Strategie, von den Blättern aus nach oben zu arbeiten. Dabei garantieren wir, dass jeder Teilbaum bereits ein Heap ist, wenn wir ihn verarbeiten.

Der Schlüsselgedanke ist dabei, dass Blattknoten per Definition schon gültige Heaps sind. Daher müssen wir nur sicherstellen, dass die Elternknoten die Heap-Eigenschaft erfüllen. Dies geschieht durch eine abwärtsgerichtete Heapify-Funktion.

Somit beginnt der Algorithmus, indem er den letzten Nicht-Blattknoten bestimmt. Alle Knoten mit einem Index grösser als dieser Wert sind bereits Blätter und müssen nicht bearbeitet werden. Um den Heap zu erstellen, wenden wir Heapify auf alle Nicht-Blatt-Knoten an, beginnend beim letzten und endend bei der Wurzel. Die Heapify-Methode ähnelt dem Wiederherstellen des Heaps nach dem Entfernen der Wurzel und funktioniert für den Knoten mit Index i wie folgt:

1. Vergleiche den Knoten mit Index i mit seinen Kindern.
2. Falls eines der Kinder grösser ist (für einen Max-Heap), tausche den Knoten mit dem grössten Kind.
3. Wende Heapify rekursiv auf den betroffenen Teilbaum an.

In der nächsten Aufgabe wird dein Verständnis des Heapify-Algorithmus getestet.

Übung 2.18 ***Laufzeiten*** Lösung

Sei $[28, 3, 14, 39, 21, 1, 31, 23]$ ein unsortiertes Array. Verwende den Heapify-Algorithmus, um einen Min-Heap zu erstellen. Zeichne alle Zwischenschritte.

Reicht dein Verständnis des Algorithmus bereits aus, damit du ihn implementieren kannst?

Übung 2.19 ***Heapify programmieren*** Lösung

Implementiere den Heapify Algorithmus.

```
def heapify(                    ):
```

Mit diesem Verständnis des Heapify-Algorithmus wollen wir nun die Laufzeit dieses Algorithmus analysieren.

Übung 2.20 ***Laufzeiten*** Lösung

Was ist die Worst-Case Laufzeit des Heapify Algorithmus bei einem Datensatz der Grösse n? Tip: Überlege die Anzahl an Tausche ein Knoten auf Level i maximal machen muss. Wie viele Knoten hat es auf Level i.

Was du gelernt hast: Wir haben zwei Methoden zum Bauen eines Heaps kennengelernt. Aus Aufgabe 2.20 geht hervor, dass wir mit dem zweiten Ansatz den Heap in linearer Laufzeit bauen können. Dabei vermeidet die Bottom-Up-Heap-Konstruktion überflüssige Operationen und nutzt die natürliche Struktur des Binärbaums, um ein unsortiertes Array effizient in einen Heap umzuwandeln.

Lösungen zu den Übungen

Lösung 2.1 ***Notaufnahme als ADT*** Übung

Eine einfache Lösung ist die Verwaltung der Patientinnen und Patienten in einer **sortierten Liste**, die stets absteigend nach Dringlichkeit geordnet ist.
Die Liste wird nach dem Einfügen jeder neuen Person so sortiert, dass die dringendste Person ganz vorne steht.
Operationen:

- `fügeEin(p)`: Die Person `p` wird an der korrekten Stelle (nach Dringlichkeit) in die Liste eingefügt. Dafür muss die gesamte Liste durchsucht werden, was im schlimmsten Fall $\mathcal{O}(n)$ Zeit kostet.
- `behandleDringendste()`: Die erste Person in der Liste (Index 0) wird entfernt und zurückgegeben. Das geht in $\mathcal{O}(1)$.
- `gibDringendste()`: Gibt die erste Person in der Liste zurück, ohne sie zu entfernen. Ebenfalls in $\mathcal{O}(1)$.

Beobachtung: Diese Lösung ist funktional korrekt, aber ineffizient bei häufigem Einfügen, da jedes Einfügen potenziell eine komplette Neuordnung der Liste erfordert. Die Zeitkomplexität liegt bei:

- Einfügen: $\mathcal{O}(n)$
- Zugriff/Entfernen der dringendsten Person: $\mathcal{O}(1)$

Lösung 2.2 ***Notaufnahme als ADT*** Übung

Wenn wir zu jedem Zeitpunkt wissen, welche Person die höchste Dringlichkeit hat, so können wir den Ärzten/ Ärztinnen und dem Pflegepersonal zu jedem Zeitpunkt sagen, wer als nächstes gesehen werden muss.
Es genügt, dass das dringendste Element immer an einer definierten Stelle (z.B. am Anfang der Liste) steht. Eine vollständige Sortierung ist nicht nötig.

- `fügeEin(p)`: Neue Patienten werden an das Ende der Liste angehängt. Überprüfung auf höchste Dringlichkeit in $\mathcal{O}(1)$. Wir müssen den neuen Patienten nur mit dem derzeit ersten Vergleichen danach kommt der Patient entweder nach ganz vorne oder ganz hinten.
- `gibDringendste()`: Zugriff auf die Person mit höchster Dringlichkeit in $\mathcal{O}(1)$, da wir wissen, wo sich diese befindet.
- `behandleDringendste()`: Entfernen des dringendsten Elements, was in

$\mathcal{O}(n)$ erfolgt, da wir die nächst dringendste Person im Anschluss finden müssen.

Lösung 2.3 ***Heap Methoden in anderen Datenstrukturen*** Übung

	Einfügen	Extract	Peek
Array (unsortiert)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Array (sortiert)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Linked List (unsortiert)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Linked List (sortiert)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Binärsuchbaum	$\mathcal{O}(n)$	height of tree $\mathcal{O}(n)$	height of tree $\mathcal{O}(n)$

Lösung 2.4 ***Heap Methoden in anderen Datenstrukturen*** Übung

a) Ja, der Baum stellt eine Min-Heap dar.

b) Nein, da 50 grösser ist als 20, 20 aber kleiner als 60.

c) Ja, der Baum stellt eine Max-Heap dar.

d) Nein, da der Baum kein Binärbaum ist.

e) Nein, da der Baum kein vollständiger Binärbaum ist.

Lösung 2.5 ***Überprüfen der Heap Eigenschaft*** Übung

```
class Node:
    """
    Die Node-Klasse stellt einen Knoten in einem Binärbaum
    ↪ dar.

    Attribute:
    - value: Der Wert des Knotens (der Wert des Knotens im
    ↪ Heap).
    - left: Verweis auf das linke Kind des Knotens (None, wenn
    ↪ kein linkes Kind existiert).
    - right: Verweis auf das rechte Kind des Knotens (None,
    ↪ wenn kein rechtes Kind existiert).
    """
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

```python
def checkMinHeapProperty(x):
    """
    Überprüft, ob der Binärbaum mit Wurzel x die
    ↪ Min-Heap-Eigenschaft erfüllt.

    Parameter:
    - x: Der Knoten, der die Wurzel des zu überprüfenden
    ↪ Baums darstellt.

    Rückgabewert:
    - True, wenn der Baum die Heap-Eigenschaft erfüllt,
    ↪ andernfalls False.
    """
    if x is None:
        return True

    if x.left is not None:
        if x.value > x.left.value:
            return False
        if not checkHeapProperty(x.left):
            return False

    if x.right is not None:
        if x.value > x.right.value:
            return False
        if not checkHeapProperty(x.right):
            return False
    return True
```

Lösung 2.6 ***Vollständiger Baum und Level Order Traversierung*** Übung

Sei $n = 2^h - 1 + k$, wobei $0 \leq k < 2h$. In einem vollständigen Baum ist die Höhe h wenn $k = 0$ und $h + 1$ wenn $k > 0$. Zusätzlich ist die Anzahl der Knoten auf Höhe i gleich 2^{i-1}, für $1 \leq i \leq h$.
Somit sind die Indizes auf Level i zwischen $2^{i-1} - 1$ und $2^i - 2$, was eindeutig nicht grösser ist als $n - 1$. Für das Level $h + 1$ sind die Indexe bei einem vollständigen Baum zwischen $2^h - 1$ und $2^h - 2 + k$. Das heisst, dass der maximale Index gleich $n - 1$ ist.

Lösung 2.7 ***Überprüfen vollständiger Binärbaum*** Übung

```
class Node:
    """
    Die Node-Klasse stellt einen Knoten in einem Binärbaum
    ↪ dar.

    Attribute:
    - value: Der Wert des Knotens (der Wert des Knotens im
    ↪ Heap).
    - left: Verweis auf das linke Kind des Knotens (None, wenn
    ↪ kein linkes Kind existiert).
    - right: Verweis auf das rechte Kind des Knotens (None,
    ↪ wenn kein rechtes Kind existiert).
    """
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def isCompleteBinaryTree(root, index=0, total_nodes=None):
        """
        Überprüft rekursiv, ob der Baum ein vollständiger
        ↪ Binärbaum ist.

        Parameter:
        - root: Die Wurzel des Binärbaums.
        - index: Der aktuelle Index des Knotens in einer
        ↪ hypothetischen Level-Order
        - total_nodes: Die Gesamtanzahl der Knoten im Baum
        ↪ (wird nur beim ersten Aufruf berechnet).

        Rückgabewert:
        - True, wenn der Baum vollständig ist.
        - False, wenn der Baum nicht vollständig ist.
        """
        if root is None:
            return True  # Ein leerer Baum ist vollständig

        if total_nodes is None:
            total_nodes = countNodes(root)

        if index >= total_nodes:
```

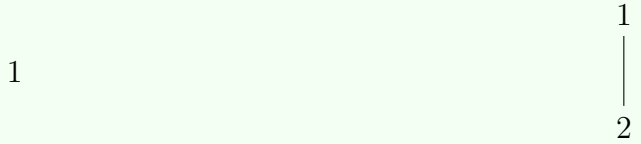
```
            return False

        return (isCompleteBinaryTree(root.left, 2 * index + 1,
        ↪  total_nodes) and
                isCompleteBinaryTree(root.right, 2 * index + 2,
                ↪  total_nodes))


    def countNodes(root):
        """
        Hilfsmethode, die die Anzahl der Knoten im Baum
        ↪  berechnet.
        """
        if root is None:
            return 0
        return 1 + countNodes(root.left) +
        ↪  countNodes(root.right)
```

Lösung 2.8 ***Kinder finden*** Übung

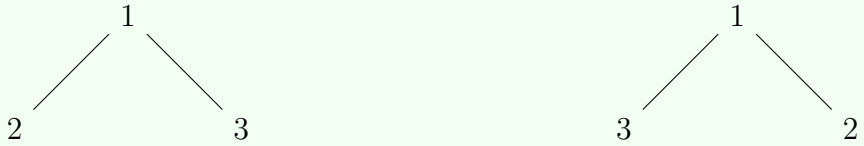
$$l(i) = 2i + 1$$
$$r(i) = 2i + 2$$

Lösung 2.9 ***Wie viele Heaps?*** Übung

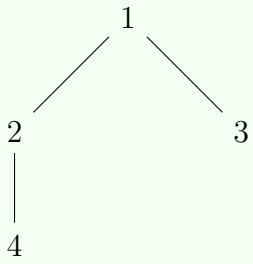
Es gibt eine Heap für einen Knoten und eine Heap für zwei Knoten:

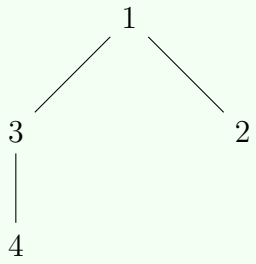


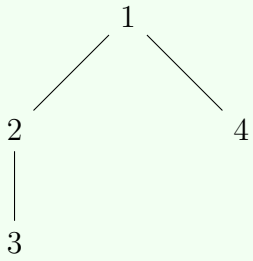
Für 3 Knoten gibt es 2 Min-Heaps



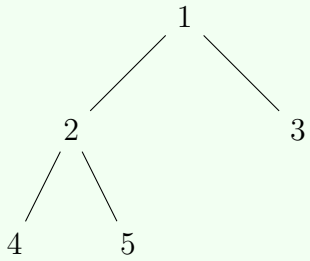
Für 4 Knoten gibt es drei Heaps:

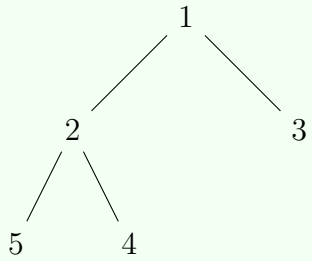


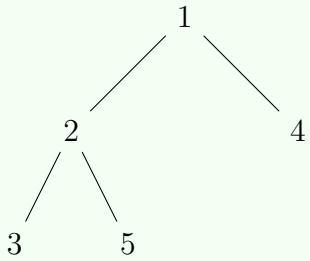


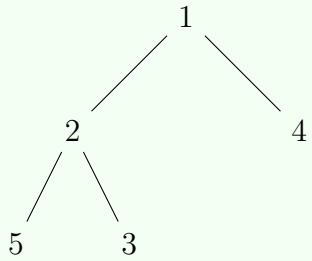


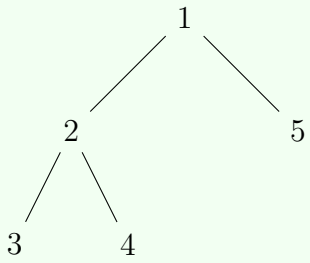
Es gibt 8 Min-Heaps mit 5 Knoten:

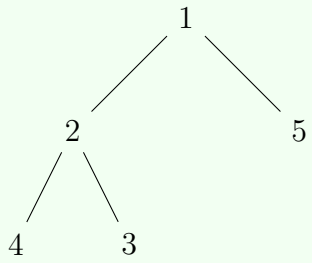


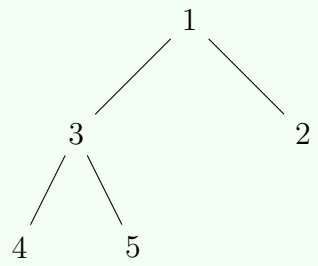


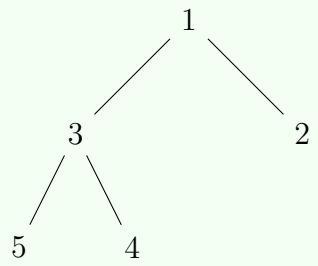












Lösung 2.10 ***Einfügen in Heaps*** Übung

Da wir nach dem Einfügen des neuen Elementes einen vollständigen Binärbaum haben und danach nur noch Positionen tauschen, ist das Ergebnis ein vollständiger Baum.
Wir zeigen, dass zu jedem Zeitpunkt gilt, dass der Subbaum mit Wurzel x eine Min-Heap bildet.
Dies gilt eindeutig, wenn x als Blatt hinzugefügt wird, da ein einzelner Knoten immer eine Min-Heap bildet.
Falls x bereits größer oder gleich dem Wert seines Elternknotens ist $value(x) \leq value(p(x))$ dann bleibt die Min-Heap-Eigenschaft erhalten, da alle anderen Knoten weiterhin die Min-Heap-Bedingung erfüllen. In diesem Fall wird der while-Loop nie ausgeführt, und der Algorithmus endet sofort. Die Heap-Eigenschaft bleibt somit erhalten.
Falls x kleiner als sein Elternknoten $p(x)$ ist, wird ein Swap durchgeführt. Danach befindet sich x eine Stufe höher im Baum und $p(x)$ eine Stufe tiefer. Da $x < p(x)$ und $p(x)$ kleiner als seine Kinder war bevor x eingefügt wurde, muss nach der Transitivität x kleiner sein als alle seine Kinder. Dies gilt für jeden Swap und daher gilt, dass x immer kleiner gleich seiner Kinder ist. Da keine anderen Swaps durchgeführt werden, muss der Subbaum mit Wurzel x eine Min-Heap formen. Da wir beim Einfügen keinen anderen Teil des Baumes verändern muss der Baum auch nach dem Einfügen eine Min-Heap formen.

Lösung 2.11 ***Einfügen in Heaps*** Übung

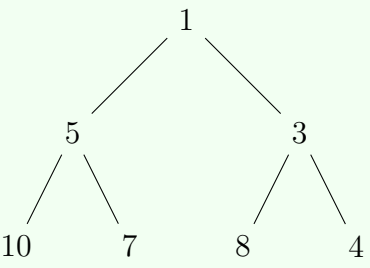
Wir zeigen die Implementierung für eine Min-Heap. Für eine Min-Heap muss die Tauschbedingung in der While-Loop angepasst werden.

```
class MinHeap:
    def __init__(self):
        """
        Initializes an empty Min-Heap.
        """
        self.heap = []
```

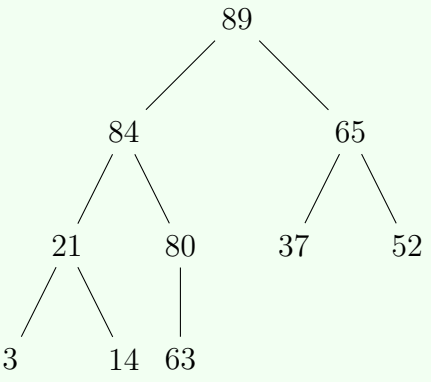
```python
def add(self, value):
    """
    Adds a new value to the Min-Heap while maintaining the
    ↪ heap property.

    Parameters:
    value: The value to be added to the heap.
    """
    self.heap.append(value)

    current_index = len(self.heap) - 1
    while current_index > 0:
        parent_index = (current_index - 1) // 2
        if self.heap[current_index] < self.heap[parent_index]:
            self.heap[current_index], self.heap[parent_index] =
            ↪ self.heap[parent_index],
            ↪ self.heap[current_index]
            current_index = parent_index
        else:
            break
```

Lösung 2.12 ***Einfügen in Heaps*** Übung

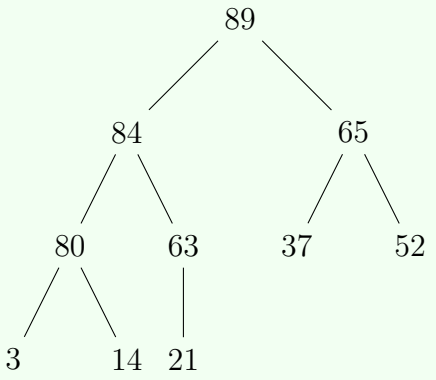
a)



b)

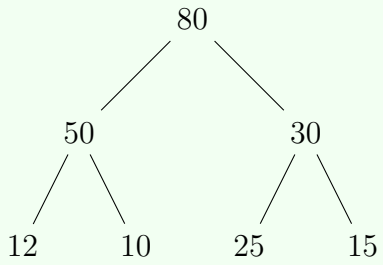


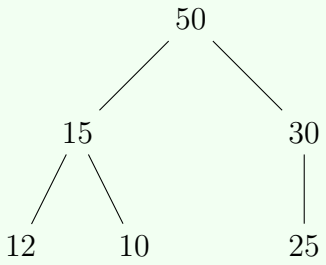
Ja, die Reihenfolge des Einfügens ist wichtig. Hätten wir beispielsweise die Elemente in der Reihenfolge 3, 84, 21 hinzugefügt, so wäre das Resultat



Lösung 2.13 *Entfernen in Heaps* Übung

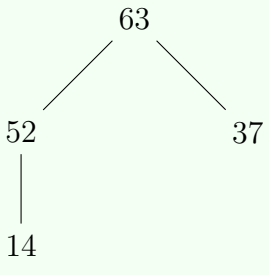
Nein, dies muss nicht der Fall sein, wie das nächste Beispiel zeigt.



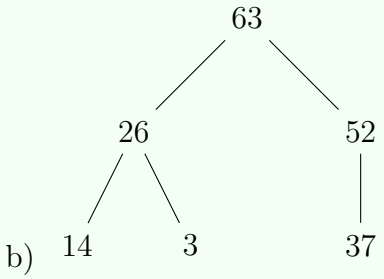


Lösung 2.14 *Heap Methoden anwenden* Übung

a)



b)



Lösung 2.15 ***Entfernen in Heaps*** Übung

Wir zeigen die Implementierung für eine Max-Heap. Für eine Min-Heap muss die Tauschbedingung in der While-Loop angepasst werden.

```
class Heap:

    def __init__(self, elements=None):
        """
        Initialisiert den Heap mit einer Liste von Elementen.

        HINWEIS: Die übergebene Liste muss bereits die
        ↪  Eigenschaften eines Heaps erfüllen.

        :param elements: (list) Eine Liste von Zahlen zur
        ↪  Initialisierung des Heaps.
        """
        self.heap = elements if elements else []

    def remove(self):
        """
        Entfernt das größte Element (die Wurzel) aus dem Max-Heap
        und stellt die Heap-Eigenschaft wieder her.

        :return: (int) Das entfernte größte Element oder None,
        ↪  falls der Heap leer ist.
        """
        if len(self.heap) == 0:
            return None

        root = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
```

```
        index = 0
        size = len(self.heap)

        while index < size:
            left = 2 * index + 1
            right = 2 * index + 2
            largest = index

            if left < size and self.heap[left] >
            ↪  self.heap[largest]:
                largest = left
            if right < size and self.heap[right] >
            ↪  self.heap[largest]:
                largest = right

            if largest == index:
                break

            self.heap[index], self.heap[largest] =
            ↪  self.heap[largest], self.heap[index]
            index = largest

        return root
```

Lösung 2.16 ***Laufzeiten*** Übung

a) $\mathcal{O}(1)$, da sich der gesuchte Wert immer am Index 0 befindet.

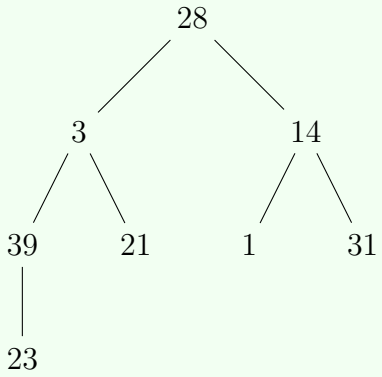
b) $\mathcal{O}(\log(n))$ die maximal Anzahl an Tausche der Höhe des Heaps entspricht.

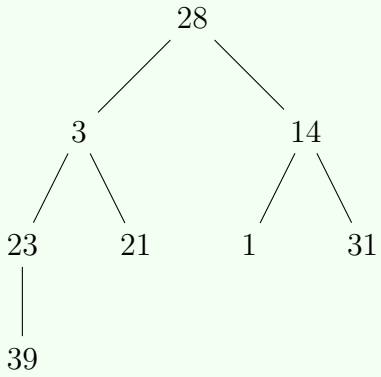
c) $\mathcal{O}(\log(n))$ die maximal Anzahl an Tausche der Höhe des Heaps entspricht.

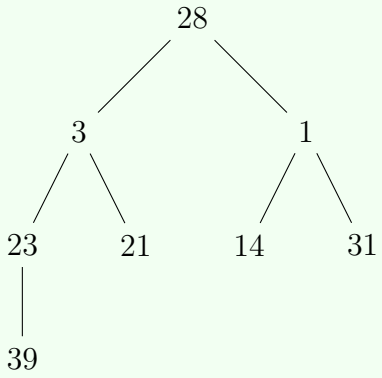
Lösung 2.17 ***Laufzeiten*** Übung

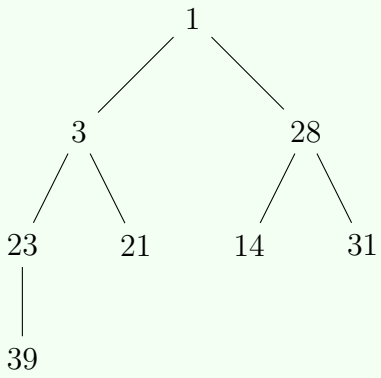
Die Worst-Case-Laufzeit für das Hinzufügen des i-ten Datenpunktes beträgt $(\log(i))$. Daher wissen wir, dass die gesamte Laufzeit $\mathcal{O}(n \log(n))$ ist, da wir n Summanden haben, welche alle höchstens $\log(n)$ sind. Gleichzeitig haben wir $\frac{n}{2}$ Summanden, welche alle mindestens $\log(\frac{n}{2})$ sind, weshalb die Worst-Case-Laufzeit mindestens $\Omega(\frac{n}{2} \log(\frac{n}{2})$ ist. Daher erhalten wir eine asymptotische Laufzeit von $\Theta(n \log(n))$.

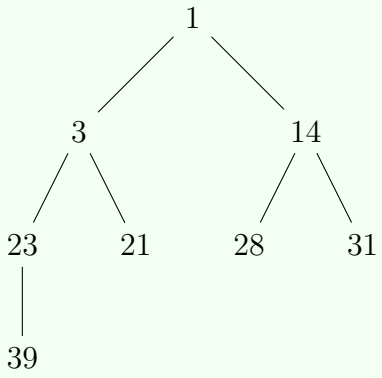
Lösung 2.18 *Laufzeiten* Übung











Lösung 2.19 *Heapify programmieren* Übung

```
def heapify(arr, n, i):
"""
Stellt sicher, dass der Teilbaum mit Wurzel an Index i ein
↪  Max-Heap ist.

:param arr: Liste, die den Heap repräsentiert
:param n: Größe des Heaps
:param i: Index des Knotens, der heapifiziert wird
```

```
    """
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def build_heap(arr):
    """
    Wandelt eine unsortierte Liste in einen Max-Heap um.

    :param arr: Liste der Elemente
    """
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
```

Lösung 2.20 ***Laufzeiten*** **Übung**

Sei h die Höhe ded vollständigen Binärbaumes. Dann gilt. $2^h \leq n \leq 2^{h+1} - 1$. Im level i gibt es höchstens 2^{i-1} Knoten. Da der Knoten am Ende nicht tiefer als Level h liegen kann wird er maximal $h - i$ mal getauscht. Summieren wir dies über alle Level so sehen wir das die gesammte Anzahl Tausche maximal $\sum_{i=0}^{h} 2^i(h-i) = -h + 2^{h+1} - 2 = \mathcal{O}(n)$.