Eidgenössische Technische Hochschule Zürich
Professur für Informationstechnologie und Ausbildung
Prof. Dr. Juraj Hromkovič
Universitätstrasse 6
CH-8092 Zürich

Mentorierte Arbeite

Abstrakte Datenstrukturen am Beispiel von binären Suchbäume

Judith Beestermöller

HS 2023

Aufgabe 1 - Konzeption der Unterrichtseinheit

Leitidee:

Eine der wichtigsten Aufgeben von Computern ist das Speichern und Abrufen von Daten. Daher ist es wichtig effiziente Datenstrukturen zu kennen, welche es uns erlauben in einem Datensatz schnell Datenpunkte zu finden, hinzuzufügen, und zu löschen.

Eine abstrakte Datenstruktur erlaubt es uns, die Eigenschaften einer Datenstruktur zu definieren, ohne auf die Implementierung einzugehen. Im obigen Beispiel wurde beispielsweise eine Datenstruktur definiert, welche eine find, add und delete Methode hat. Es wurde aber nicht definiert, wie diese Methoden implementiert sein sollen. Das Konzept der abstrakten Datenstrukturen ist sehr wichtig in der Informatik, da es Separierung der Implementation der Benutzung erlaubt. Dies hat den Vorteil, dass ein Benutzer sich keine Gedanken über die genaue Implementation machen muss, gleichzeitig muss der Benutzer nicht bekannt geben, wofür genau er die Datenstruktur benötigt.

Binäre Suchbäume sind eine einfache Datenstruktur, welche das Finden, Hinzufügen und Löschen effizient ausführen, solange der Binärbaum ausbalanciert ist. Aus diesem Grund sollen die SuS den binären Suchbaum als Datenstruktur kennen. Gleichzeitig ist es wichtig, dass die SuS erkennen, dass die Effizienz des binären Suchbaumes nur besteht, solange die Höhe des Baumes logarithmisch zu der Anzahl Knoten im binäre Suchbaum ist. Ist hingegen die Höhe des binären Suchbaums linear zu der Anzahl Knoten, so verliert der binäre Suchbaum deutlich an seiner Effizienz.

Die Leitidee dieser LPU ist es, dass die Schülerinnen und Schüler durch den Wunsch, eine Datenstruktur zu entwickeln, die effizientes Suchen, Hinzufügen und Löschen ermöglicht, das Konzept abstrakter Datenstrukturen kennenlernen. Darüber hinaus lernen sie binäre Suchbäume als eine Datenstruktur kennen, die diese Anforderungen erfüllt.

Dispositionsziel:

Die SuS verstehen die Grundkonzepte abstrakter Datenstrukturen und sind in der Lage, diese auf verschiedene Problemstellungen anzuwenden. Sie können die Eigenschaften von abstrakten Datenstrukturen analysieren und bewerten, um geeignete Strukturen für spezifische Anforderungen auszuwählen. Darüber hinaus erkennen sie die Bedeutung einer effizienten Implementierung von Datenstrukturen und können bei der Gestaltung neuer Datenstrukturen bereits im Vorfeld überlegen, wie Daten am besten organisiert werden sollten, um eine effiziente Ausführung von Operationen zu ermöglichen. Dabei verstehen sie den Vorteil der Separierung von Implementation und Anwendung, was es erlaubt, die Datenstruktur unabhängig von konkreten Implementierungsdetails zu betrachten und dadurch die Wiederverwendbarkeit und Wartbarkeit von Code zu verbessern.

Die SuS kennen den binären Suchbaum. Sie sind in der Lage bei einem beliebigen binären Suchbaum zu überprüfen, ob ein gegebenes Element vorhanden ist, sowie ein Element hinzuzufügen oder zu löschen. Des Weiteren verstehen die SuS die Wichtigkeit der balancierten Höhe des binären Suchbaumes für die effiziente Ausführung der Operationen. Daher machen sie die SuS in Zukunft bereits beim Anlegen einer neuen

Datenstruktur Gedanken, wie man die Daten einpflegen sollte, damit die Operationen der Datenstruktur effizient ausgeführt werden können.

Operationalisiertes Lernziel:

Die SuS können

- Die Schülerinnen und Schüler können die Konzepte abstrakter Datenstrukturen erklären und anhand von konkreten Beispielen die Vorteile einer Trennung von Implementation und Anwendung aufzeigen.
- können die grundlegenden Operationen eines binären Suchbaums (Finden, Hinzufügen und Löschen von Elementen) korrekt durchführen und verstehen dabei die Bedeutung einer ausbalancierten Baumstruktur für die Effizienz dieser Operationen.
- Die Schülerinnen und Schüler können bei der Gestaltung neuer Datenstrukturen strategisch darüber nachdenken, wie Daten organisiert werden sollten, um eine effiziente Ausführung von Operationen zu ermöglichen, und dabei die Separierung von Implementation und Anwendung berücksichtigen.

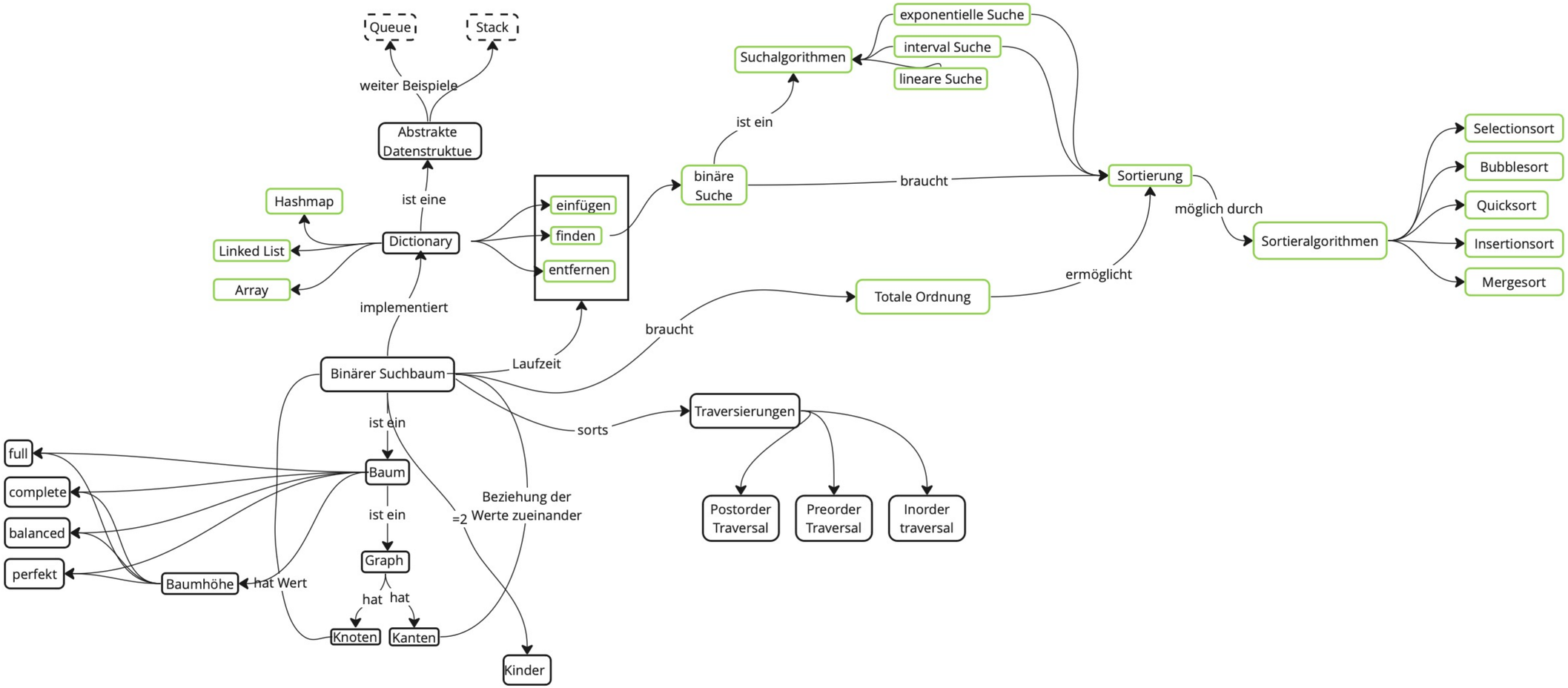
Ausgangslage:

Die SuS haben bereits das Kapitel zu Suchalgorithmen (ausgearbeitet von R. Melzian und J. Beestermöller) erarbeitet und im Anschluss verschiedene Sortieralgorithmen kennengelernt. In diesem Zusammenhang sind die SuS bereits mit Logarithem vertraut. In beiden Lektionen ist man von einem statischen Datensatz ausgegangen. Der Einstieg in diese Einheit wird dadurch motiviert, dass man die SuS darauf aufmerksam macht, dass Datensätze nicht statisch sein müssen. Gleichzeitig besteht der Wunsch des effizienten Findens weiterhin.

Bis jetzt hatten die SuS noch keine Einführung in die Graphentheorie. In dieser Lektion werden die notwendigen Begriffe der Graphentheorie eingeführt (Knoten, Kanten, ...) der Fokus soll aber auf der Datenstruktur liegen. Linked Lists wurden bereits im Unterricht behandelt und dient als Anknüpfungspunkt.

Zusätzlich wird davon ausgegangen, dass die SuS bereits Erfahrungen in der Object Orientierten Programmierung haben. Wir gehen davon aus, dass die SuS in der Lage sind, die behandelten Sortieralgorithmen zu programmieren. Darüber hinaus haben sie auch Erfahrung mit Linked List, Arrays und selbst definierten Datenstrukturen. Auch haben die SuS bereits erfahrung mit rekrusiven Algorithmen, und rekrusiv beschriebenen Datenstrukturen.

Eine Übersicht der zu behandelten Themen ist in der folgenden Mindmap abgebildet. Die den SuS bereit bekannten Themen sind grün markiert.


Queue
Stack
weiter Beispiele
Abstrakte Datenstruktue
ist eine
Hashmap
Linked List
Array
Dictionary
implementiert
einfügen
finden
entfernen
ist ein
Suchalgorithmen
exponentielle Suche
interval Suche
lineare Suche
binäre Suche
braucht
Sortierung
möglich durch
Sortieralgorithmen
Selectionsort
Bubblesort
Quicksort
Insertionsort
Mergesort
Totale Ordnung
ermöglicht
braucht
Laufzeit
Binärer Suchbaum
sorts
Traversierungen
Postorder Traversal
Preorder Traversal
Inorder traversal
ist ein
Baum
full
complete
balanced
perfekt
Baumhöhe
hat Wert
ist ein
Graph
hat
hat
Knoten
Kanten
=2
Beziehung der Werte zueinander
Kinder

Aufgabe 2 - Leitprogrammartige Unterrichtsunterlagen

Einführung

Wir haben zuletzt die Themen Suchen und Sortieren behandelt. Hierbei sind wir davon ausgegangen, dass wir eine statische Datenmenge haben. Das heisst, dass nachdem die Datenstruktur erstellt wurde, keine Datenpunkte entfernt oder hinzugefügt wurden. Dies ist allerdings nicht immer der Fall. Denk beispielsweise an die Bücher, die du im Regal stehen hast. Es kommen regelmässig neue Bücher hinzu, gleichzeitig sortierst du auch immer mal wieder Bücher aus. Das Gleiche gilt, wenn du vergleichst, welche Freunde du zu deiner Geburtstagsparty über die Jahre eingeladen hast. Dein bester Freund/ deine beste Freundin werden jedes Jahr eingeladen, aber es gibt auch Veränderungen auf der Gästeliste. In dieser Lektion wollen wir uns daher mit einer Datenstruktur beschäftigen, welche es uns erlaubt, effizient Datenpunkte zu finden, hinzuzufügen und zu löschen.

Wir wollen zunächst einmal ein real-world Problem betrachten, bei dem wir eine sich ändernde Datenstruktur haben.

Übung 0.1 ***Wie lange braucht die Suche?*** Lösung S. 44

Zu Begin des neuen Schuljahrs möchte die Lehrerin einen Geburtstagskalender ihrer SuS erstellen. Da sie weiss, dass sich die Klassenzusammensetzung noch ändern kann, entscheidet sie sich, Klettstreifen mit den Namen und Geburtstagen ihrer SuS anzufertigen. Diese klebt sie dann, nach dem Geburtstag sortiert, untereinander auf. Sollte sich die Klassenkonstellation ändern, kann sie einfach neue Klettstreifen erstellen, und auf die richtige Position bringen. Das kann bedeuten, dass sie die bestehenden Streifen verschieben muss.
In der Klasse sind 32 SuS. Nach den Herbstferien wechseln zwei der SuS in eine andere Klasse, dafür kommen zwei neue SuS in die Klasse.

(a) Wie kann die Lehrerin möglichst effizient die Klettstreifen der SuS finden, welche die Klasse verlassen?

(b) Wie viele Klettstreifen muss sie hierfür lesen?

(c) Wie kannst die Lehrerin möglichst schnell die Stellen im Kalender finden, an denen die Klettstreifen der neuen SuS stehen sollten?

(d) Wie viele Klettstreifen musst sie im schlechtesten Fall umkleben? (Gehe davon aus, dass es genau Platz für 32 Klettstreifen hat.)

In den nächsten zwei Aufgaben wollen wir uns angucken, ob und wie wir dieses Problem mit den Datenstrukturen, die wir bereits kennen, lösen können.

Übung 0.2 *Pyhton Class SuS* Lösung S. 44

Überlege dir, wie du SuS in der vorherigen Aufgabe in eine Python Class, mit dem Namen 'student', definieren kannst. Welche Eigenschaften braucht jeder SuS? Implementiere zusätzlich eine Vergleichsmethode, welche es dir erlaubt, die SuS nach ihrem Geburtsdatum und, falls dies nicht genügt, nach ihrem Vor- und Nachnamen zu sortieren. Du darfst annehmen, dass es keine zwei SuS gibt, welche dasselbe Geburtsdatum und denselben Namen tragen. Den Skeleton Code haben wir dir bereits zur Verfügung gestellt.

```
    class student:
'''
    Erstellt ein neues student object
    Eingabe: TODO
    Rückgabewert: TODO
'''
def __init__(TODO):
    TODO


'''
    Vergleicht 2 students anhand iheres Geburtstags/ Names
    Eingabe: TODO
    Rückgabewert: TODO
'''
def __lt__(TODO):
    TODO
```

Übung 0.3 *Implementierung Geburtstagskalender* Lösung S. 45

Wie kannst du mittels der student Class aus der vorherigen Aufgabe den Geburtstagskalender implementieren? Überlege dir zwei Datenstrukturen, mit welchen du den Kalender implementieren könntest.
Analysiere für beide Implementationen,

- wie viele Schritte es im schlechtesten Falle braucht, um die SuS zu finden, welche die Klasse verlassen?
- wie viele weitere Schritte es im schlechtesten Fall braucht, um die SuS aus der Datenstruktur zu entfernen, nachdem du sie gefunden hast?
- wie viele Schritte braucht es im schlechtesten Fall, um die Position der neuen

SuS im Kalender zu finden?

- wie aufwendig ist das Hinzufügen der SuS, nachdem du ihrer Position gefunden hast?

Was du gelernt hast:
Wir sind in der Lage, mit den bereits bekannten Datenstrukturen Array und Linked List Datenpunkte in sortierte Datenmengen hinzuzufügen. In einem Array können wir mithilfe der binären Suche die Stelle, an der wir den Datenpunkt hinzufügen wollen, schnell finden. Allerdings müssen im schlechtesten Fall alle Datenpunkte um eine Position verschoben werden. Da wir in der Linked List keinen Random Access haben, können wir die Position, an der der Datenpunkt hinzugefügt wird, nur mittels linearer Suche finden. Wenn wir die Position gefunden haben, können wir eine neue Node erstellen und diese zur Liste hinzufügen, ohne die anderen Datenpunkte verschieben zu müssen.

Abstrakte Datenstrukturen

In der letzten Aufgabe haben wir gesehen, dass man dieselbe Funktion auf zwei unterschiedliche Weisen programmieren kann. Wir hatten zuvor lediglich definiert, was das Programm können soll, nämlich einen SuS in einen Sortierentenkalender hinzufügen oder entfernen. Wie genau man dies implementiert, war nicht Teil der Spezifizierung und wurde dem Programmierer überlassen. Informatiker nennen dies eine *Abstrakte Datenstruktur*.

Konzepte und Begriffe:
Eine Abstrakte Datenstruktur ist ein Konzept in der Informatik, das die strukturierte Organisation von Daten beschreibt, unabhängig von der konkreten Implementierung. Sie definiert, wie Daten organisiert und manipuliert werden können, ohne die internen Details preiszugeben. Abstrakte Datenstrukturen bieten eine standardisierte Möglichkeit, auf Daten zuzugreifen und Operationen darauf auszuführen. Diese Konzepte ermöglichen es Informatikern, Daten effizient zu verwalten und zu analysieren, ohne sich auf die spezifischen Details der Implementierung festlegen zu müssen.

Ein grosser Vorteil von abstrakten Datenstrukturen ist, dass sie die Funktionalität von der Implementierung abstrahieren. Eine Person, welche eine abstrakte Datenstruktur benutzt, braucht nicht zu wissen, wie diese implementiert ist. Die *Spezifikation* einer abstrakten Datenstruktur definiert die strukturellen und operationellen Eigenschaften der Datenstruktur, ohne dabei auf die konkrete Implementierung einzugehen. Sie beschreibt, welche Daten in der Datenstruktur gespeichert werden können und welche Operationen auf diesen Daten möglich sind. Eine solche Spezifikation legt die Schnittstelle fest, über die Programme mit der Datenstruktur interagieren können, ohne sich um deren interne Details kümmern zu müssen. Dadurch ermöglicht sie eine klare Trennung zwischen der Nutzung der Datenstruktur und ihrer konkreten Umsetzung, was die Modellierung, Entwicklung und Wartung von Software erleichtert.

Eine bekannte abstrakte Datenstucktur ist der Stack (Stapel) in der nächsten Aufgabe wirst du aufgefordert den Stack anhand seiner Spezifikation zu implementieren.

Übung 0.4 ***Stapel*** Lösung S. 48

Ein bekannte abstrakte Datenstruktur in der Informatik ist der Stack. Die Spezifikation des Stacks ist wiefolgt:

- Initialization (Initialisierung): Eine leere Stack-Datenstruktur wird erstellt.
- is_empty(): Überprüft, ob der Stack leer ist. Rückgabewert: True, wenn der Stack leer ist, andernfalls False.
- push(item): Fügt ein Element oben auf den Stack hinzu. Parameter: item - Das einzufügende Element.
- pop(): Entfernt das oberste Element vom Stack und gibt es zurück. Rückgabewert: Das entfernte Element.
- peek(): Gibt das oberste Element auf dem Stack zurück, ohne es zu entfernen. Rückgabewert: Das oberste Element.
- size(): Gibt die Anzahl der Elemente im Stack zurück.

Überlege dir, wie du den Methoden des Stacks mit einer Linked List implementieren könntest.

Eine Spezifikation für den Geburtstagskalender könnte beispielsweise wie folgt lauten:

- __init__: Erstellt einen neuen Kalender
- add_student(self, stu): Fügt SuS *stu* an korrekter Stelle in den Kalender hinzu
- remove_student(self, stu): Entferne SuS *stu* vom Kalender

Ein Benutzer könnte anhand des folgenden Codes nun den Kalender nutzen, ohne sich Gedanken darüber zu machen, wie der Kalender implementiert wurde.

```python
# Erstelle Schülerobjekte
student1 = student("Alice", 5, 10)
student2 = student("Bob", 3, 15)
student3 = student("Charlie", 5, 5)

# Erstelle Kalender und füge Schüler hinzu
calendar = BirthdayCalendar()
calendar.add_student(student1)
calendar.add_student(student2)
calendar.add_student(student3)

# Entferne einen Schüler
calendar.remove_student(student2)

# Füge einen neuen Schüler hinzu
new_student = student("David", 4, 20)
calendar.add_student(new_student)
```

Die nächste Aufgabe fordert dich auf, dies anhand der zwei Implementierungen des Geburtstagskalenders in Aufgabe 0.3 selber zu überprüfen.

Übung 0.5 ***Nutzerfreiheit von Abstrakten Datenstuckturen*** Lösung S. 49

Überprüfe, dass der gezeigte Code mit beiden Implementierungen des Geburtstagskalenders in Aufgabe 0.3 benutzt werden kann.

Die abstrakte Datenstruktur des *Dictionary*

Wie bereits zu Beginn des Kapitels erwähnt, ist es unser Ziel eine Datenstruktur kennenzulernen, in der wir effizient Datenpunkte finden, hinzufügen und entfernen können. Eine Datenstucktur bei der sich die Datenmenge verändert wird auch als dynamisch bezeichnet. Wir wollen uns zunächst einmal überlegen, welche Methoden eine solche Datenstruktur braucht. Die Aufgabe ist nicht ganz einfach. Es kann helfen, wenn du dir die Spezifikationen der abstrakten Datenstrukturen in Aufgabe 0.4 noch einmal anschaust.

Übung 0.6 ***Definiere eine abstrakte Datenstruktur*** Lösung S. 49

Überlege dir, welche Methoden eine solche Datenstruktur benötigt. Schreibe eine Spezifikation für eine solche Datenstruktur. Bedenke, dass die Spezifikation einer abstrakten Datenstruktur keine Implementierungsdetails enthalten soll.

Eine Standard abstrakte Datenstruktur, die diese Funktionen bereits hat, ist der Dic-

tionary. Dictionaries sind in Python bereits als Datenstruktur für uns implementiert und haben die folgende Spezifikation:

- `update(iterable)` - Fügt Schlüssel-value-Paare aus einem anderen Dictionary oder einer anderen Iterable hinzu oder aktualisiert sie.
- `pop(key, default=None)` - Entfernt den angegebenen Schlüssel und gibt die dazugehörige value zurück. Wenn der Schlüssel nicht vorhanden ist, wird die Standardvalue (standardmäßig None) zurückgegeben.
- `popitem()` - Entfernt und gibt ein beliebiges Schlüssel-value-Paar als Tupel zurück. Nützlich, um Elemente in zufälliger Reihenfolge zu durchlaufen.
- `pop(key)` - Entfernt den angegebenen Schlüssel und gibt die dazugehörigen value zurück. Wenn der Schlüssel nicht vorhanden ist, wird ein Key Error ausgelöst.
- `clear()` - Löscht alle Elemente aus dem Dictionary.
- `copy()` - Gibt eine flache Kopie des Dictionaries zurück.
- `fromkeys(iterable, value=None)` - Gibt ein neues Dictionary zurück, wobei die Elemente aus der übergebenen Iterable mit dem angegebenen value (standardmäßig None) initialisiert werden.
- `get(key, default=None)` - Gibt die value für den angegebenen Schlüssel zurück. Wenn der Schlüssel nicht vorhanden ist, wird die Standardvalue (standardmäßig None) zurückgegeben.
- `items()` - Gibt eine Ansicht mit allen Schlüssel-value-Paaren im Dictionary zurück.
- `keys()` - Gibt eine Ansicht mit allen Schlüsseln im Dictionary zurück.
- `values()` - Gibt eine Ansicht mit allen values im Dictionary zurück.
- `setdefault(key, default=None)` - Wenn der Schlüssel vorhanden ist, gibt diese Methode doe zugehörige value zurück. Andernfalls fügt sie den Schlüssel mit der angegebenen Standardvalue (standardmäßig None) hinzu und gibt diesen zurück.

Überprüfe zunächst einmal, ob du die Spezifikation des Dictionaries verstehst und dich in der Lage fühlst, mit diesem zu arbeiten.

Übung 0.7 ***Wie funktioniert der Dictionary*** Lösung S. 50

Studiere die Definition der Dictionary abstrakten Datenstruktur. Verstehst du, was die jeweiligen Methoden machen? Falls nein, finde dies heraus. Du kannst dich auch mit einem deiner Klassenkameradinnen dazu unterhalten.

Nachdem du die Spezifikation des Dictionaries verstehst, solltest du dir überlegen, wo

und wie du diesen verwenden könntest.

Übung 0.8 ***Anwendungen vom Dictionary*** Lösung S. 50

Überlege dir 3 Anwendungsbeispiele, in denen du einen Dictionary für die Implementierung verwenden kannst. Fallen dir auch 3 Anwendungen auf, bei denen sich der Datentyp der Keys von dem der Values unterscheiden?

Ein Unterschied zwischen der abstracten Datenstruktur, welche wir für den Geburtstagskalender gebaut haben, und der Dictionary Data Struktur ist, dass jedes Element im Geburtstagskalender jeweils ein SuS war, wobei man die SuS anhand ihres Geburtstags und Namens ordnen konnte. Bei einem Dictionary besteht jedes Element aus einem $<$ key, value $>$ Paar, bei dem jeweils die Keys miteinander vergleichbar sein müssen. Das heisst, dass es zwischen den keys eine Totale Ordnung geben muss.

In der nächsten Aufgabe wirst du aufgefordert dir zu überlegen, ob und wie du den Python Dictionary verwenden könntest, um den Geburtstagskalender zu implementieren.

Übung 0.9 ***Zusammenhang Dictionary und Geburtstagskalender*** Lösung S. 50

Überlege dir, wie du die Dictonary Datenstruktur verwenden kannst, um den Geburtstagskalender digital zu implementieren.

Was du gelernt hast:
Abstrakte Datenstrukturen werden in der Informatik benutzt, um die Anwendung einer Datenstruktur und ihre Implementierung zu trennen. Eine abstrakte Datenstruktur sollte so spezifiziert sein, dass ein Nutzer sie verwenden kann, er gleichzeitig aber keine Information über die genaue Implementierung erhält. Zusätzlich sollte ein Informatiker, der eine abstrakte Datenstruktur implementiert, dies tun können, ohne zu wissen, welche Art von Datentyp verwendet wird. Häufig werden aber spezielle Anforderungen an die möglichen Datentypen gestellt. Zum Beispiel ist es beim Dictionary vonnöten, dass es eine totale Ordnung zwischen allen keys im selben Dictionary gibt, da man die Elemente ansonsten nicht unbedingt sortieren kann.

Wie gut können wir sein?

Im letzten Abschnitt haben wir abstrakte Datenstrukturen kennengelernt und gesehen, dass ein Vorteil von abstrakten Datenstrukturen ist, dass ein Benutzer die Implementierung nicht kennen muss. Als Informatiker ist es aber unsere Aufgabe, eine abstrakte Datenstruktur nicht nur zu implementieren, sondern dies auch möglichst effizient zu tun. Das heisst, wenn wir eine abstrakte Datenstruktur implementieren, dann ist es unser Ziel, dies so zu tun, dass die maximale Anzahl an Schritten, die der Computer in der Ausführung der Methoden macht, möglichst gering ist. Manchmal ist es nicht möglich,

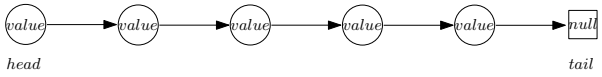


Abbildung 1: Graphische Darstellung einer Linked List

die informationstheoretische minimale Anzahl an Schritten bei allen Methoden gleichzeitig zu erreichen. In diesen Fällen gilt es abzuwägen, welche Methoden am häufigsten gebraucht werden und diese Methoden möglichst effizient zu implementieren. In diesem Unterkapitel wollen wir verstehen, wie gut wir bei den Methoden `update(iterable)`, `pop(key, default=None`, und `get(key, default=none)` sein können. Unser Ziel im folgenden Kapitel wird es sein, den Dictionary entsprechend der hier erklärten unteren Schranken zu implementieren.

Informationstheoretischer Kontext:
Im Kapitel zu Suchalgorithmen haben wir gelernt, dass man auf einer total geordneten Menge bestehend aus n Elementen im schlechtesten Fall nicht schneller als $\log(n)$ suchen kann. Wenn eine Menge nicht sortiert ist, so brauchen wir im schlechtesten Fall sogar $\mathcal{O}(n)$ Schritte, um ein Element zu finden. Daraus können wir schliessen, dass die `get(key, default=None)` Methode mindestens einen Worstcase Lower Bound von $\mathcal{O}(\log(n))$ hat.
In einer Linked List können wir ein Element mit konstant vielen Schritten ($\mathcal{O}(1)$) hinzufügen oder entfernen, sofern wir bereits einen Pointer auf die korrekte Stelle für das Hinzufügen in der Liste haben.
Um eine Laufzeit von $\mathcal{O}(\log(n))$ für die `get(key, default=None)` Methode zu garantieren, benötigen wir eine sortierte Menge. Deshalb ist es wichtig, dass sowohl die `update(iterable)` als auch die `pop(key, default=None)` Methode die Reihenfolge der Menge beibehalten. Das bedeutet, dass wir in beiden Implementierungen zunächst die richtigen Positionen (in höchstens $\mathcal{O}(\log(n))$ Schritten) finden müssen, bevor wir die Menge in $\mathcal{O}(1)$ Schritten aktualisieren. Das führt dazu, dass sowohl die `update(iterable)` als auch die `pop(key, default=None)` Methoden eine untere Grenze von $\mathcal{O}(\log(n))$ haben.

Binärer Suchbaum

In diesem Kapitel lernen wir den binären Suchbaum kennen, der die eben erläuterten Laufzeiten für die *find, add* und *delete* Methode erreicht. In einer Linked List besteht Knoten aus einer *value* und einem Pointer, der wiederum auf einen weiteren Node oder auf null zeigt. Die *Head* Node ist der einzige Knoten, die keinen eingehenden Pointer hat. Abbildung 1 zeigt eine graphische Darstellung einer Linked List. Die Pointer der Knoten werden gerne als Pfeile dargestellt. Die Pfeil deutet an, dass man dem Pointer jeweils nur in einer Richtung folgen kann. Im Computer entspricht ein Pointer einer Referenz auf eine anderen Speicherort. Folgen wir dieser Referenz so gelangen wir zu einem Speicherort, der wieder einen Knoten enthält. Dieser Knoten hat jedoch keine Refenenz zu dem Pointer der auf ihn zeigt. Daher können wir die Linked List nicht einfach Rückwärts ablaufen.

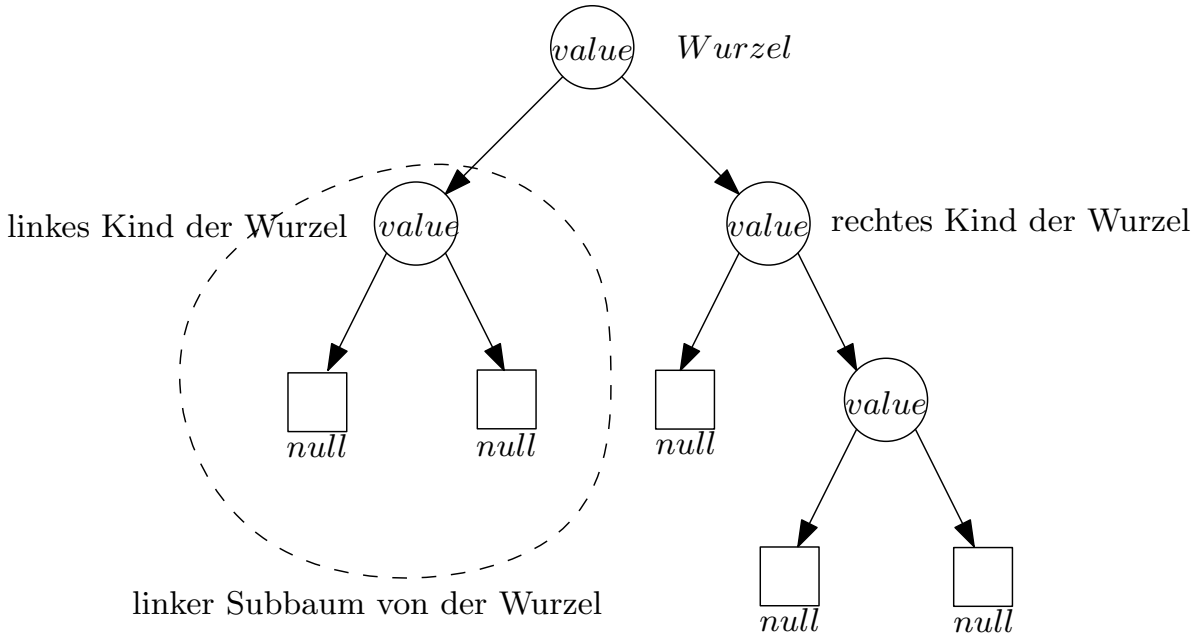


Abbildung 2: Abbildung eines Binärbaums mit Beschriftung einiger seiner Bestandsteile.

Die Datenstucktur eines Binärbaumes besteht auf Knoten welche drei Komponenten haben: eine Value, und zwei Pointer. Wie auch in der Linked List, können die Pointer auf weitere Knoten in der Datenstucktur, oder auf null zeigen.

Genau wie es in der Linked List genau eine Node gibt, auf den kein Pointer zeigt (den head), gibt es auch im Binärbaum genau einen Knoten, auf den kein Pointer zeigt. Dieser wird als *Wurzel* bezeichnet. Zeigt der Pointer eines Knotens x auf einen weitern Knoten y, so wird y als *Kind* von x bezeichnet. Um zwischen den zwei Kindern zu unterscheiden, bezeichnen wir eines als rechtes Kind und den anderen als linkes Kind. Ein Knoten, bei dem beide Pointer auf null zeigen, wird als *Blatt* bezeichnet. Eine grafische Darstellung eines Binärbaumes siehst du in Abbildung2. Manchmal lassen wir die null Pointer der einfachheitshalber in den Abbildungen weg.

In der nächsten Aufgabe wirst du aufgefordert, die beschriebene Datenstruktur in Pythoncode umzuwandeln.

Übung 0.10 ***Binärbäume Programieren*** Lösung S. 51

1. Scheibe eine Knoten Klasse, mit der Methode init, welche einen Knoten eines Binärbaumes erstellt.
2. Schreibe nun die Klasse Binärbaum. Die Klasse sollte 2 Methoden haben:
   - init: welche den Binärbaum initalisiert
   - add: welche einen weitern Knoten zum Baum hinzufügt. Für den Moment kannst du neue Knoten anstelle eine beliebigen null Pointers hin-

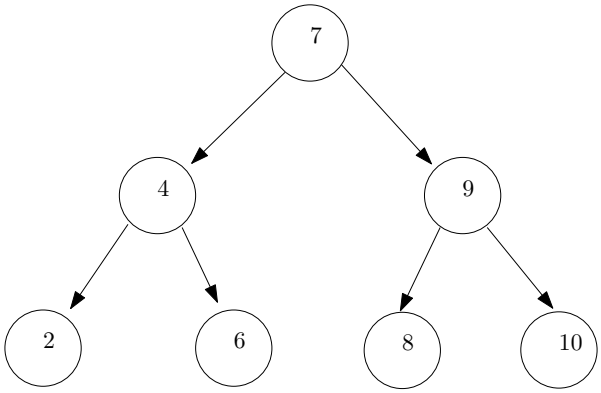


Abbildung 3: Beispiel eines binären Suchbaums.

zufügen.
Extra Challenge: Schaffst du es, den binären Suchbaum so zu implementieren, dass er möglichst verzweigt ist und nicht ein langer Pfad ist?

Dir sollte auffallen, dass Bäume rekursive sind: Sei B ein Baum und x ein beliebiger Knoten in B, der nicht die Wurzel ist. Wenn wir den eingehenden Pointer zu x abschneiden, so ist die Datenstruktur, die x weiterhin enthält wieder einen Baum. Dieser hat dann als Wurzel den Knoten x. Wir nennen diesen Baum Subbaum von B mit Wurzel x. Der Subbaum mit Wurzel $x.links$ wird als linker Subbaum von x bezeichnet, und analog wird der Subbaum mit Wurzel $x.rechts$ wird rechter Subbaum von x bezeichnet.

Jetzt, wo wir Binärbäume kennen, sind wir bereit, die Datenstruktur des binären Suchbaumes kennenzulernen: Ein binärer Suchbaum ist ein Binärbaum, bei dem die Belegung der Knoten mit values folgende Eigenschaften erfüllt:

1. Jede value kommt höchstens in einem Knoten im Suchbaum vor.

2. Für jeden Knoten x gilt, dass seine value grösser ist als alle values von Knoten im linken Subbaum von x.

3. Für jeden Knoten x gilt, dass seine value kleiner ist als alle values von Knoten im rechten Subbaum von x.

Ein Beispiel für einen binären Suchbaum siehst du in Abbildung 3. In Abbildung 4 sehen wir zum Vergleich einen Binärbaum, der kein binärer Suchbaum ist. Dies liegt daran, dass der Knoten mit value 6, als rechtes Kind den Knoten 4 hat. Dies wiederspricht der Anforderung, dass für alle Knoten x, alle Values von Knoten im rechten Subbaum von x grösser sein müssen als die Value von x.

Die Definition eines binären Suchbaums scheint auf den ersten Moment recht abstrakt. Daher wollen wir in der nächsten Aufgabe sicher gehen, dass wir sie so weit verstanden haben, dass wir binäre Suchbäume von Binärbäumen, die keine binären Suchbäume sind, unterscheiden können.

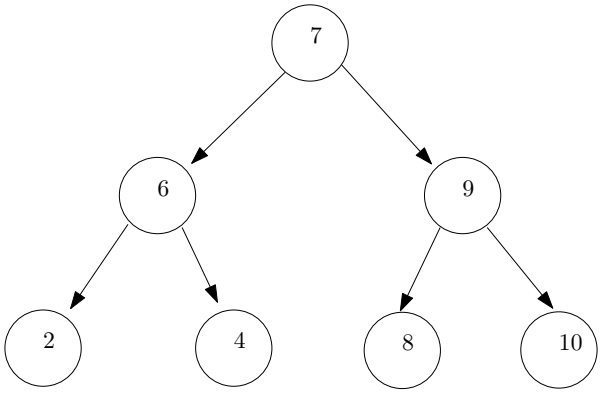
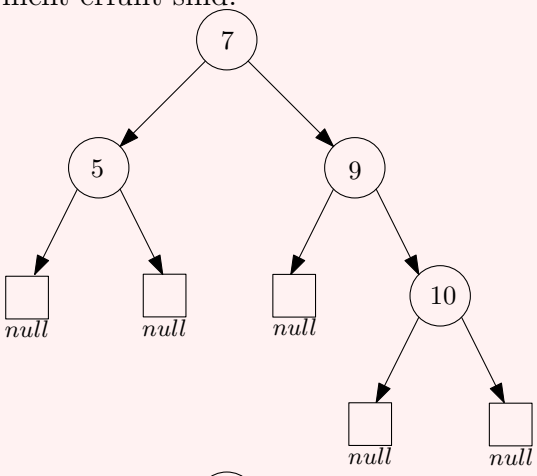
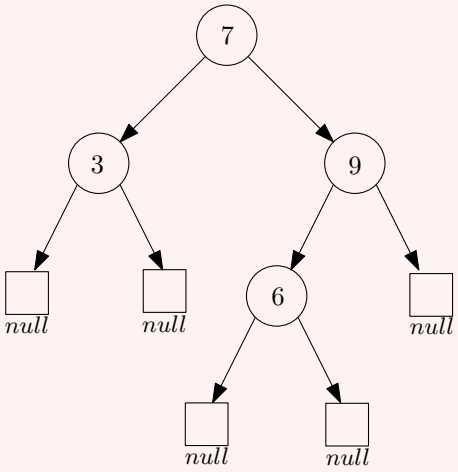


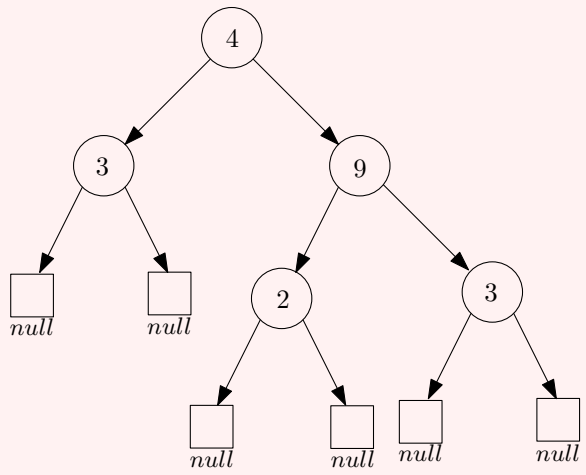
Abbildung 4: Beispiel eines Binärbaumes, der kein binärer Suchbaum ist, da 4 kleiner ist als 6.

Übung 0.11 ***Welche Binärbäume sind korrekt?*** Lösung S. 52

Entscheide für jeden der nachfolgenden Binärbäume, ob es ein binärer Suchbaum ist. Erkläre für die Bäume, die keine binären Suchbäume sind, welche Kriterien nicht erfüllt sind.







Vielleicht fragst du dich auch, ob man die Definition eines binären Suchbaumes nicht auch etwas vereinfachen kann. Dies betrachten wir in der nachfolgenden Aufgabe.

Übung 0.12 ***Andere Definition?*** Lösung S. 52

Tom meint zu Annika, dass eine äquivalente Definition für den binären Suchbaum wie folgt lautet:

1. Jede value kommt höchstens einmal im Baum vor.
2. Die value des linken Kindes ist immer kleiner als die value des Elternknotens.
3. Die value des rechten Kindes immer grösser als die value des Elternknotens.

Stimmst du zu, dass die beiden Definitionen äquivalent sind? Begründe deine Antwort.

Was du gelernt hast:
Der Binärbaum ist eine Datenstruktur, welcher aus Knoten besteht. Jeder Knoten hat drei Komponenten, einer value und zwei Pointer. Die Pointer können auf weitere Knoten im Binärbaum oder auf Null zeigen. Zeigen die Pointer eines Knoten x auf weitere Knoten, so werden diese Knoten als Kinder von x bezeichnet.
Ein binärer Suchbaum ist ein Binärerbaum, bei dem für jeden Knoten gilt, dass die values in seinem linken Subbaum alle kleiner und die values in seinem rechten Subbaum alle grösser sind als die value des Knotens selbst.

Traversierungen von binären Suchbäumen

Häufig ist es interessant, alle Knoten eines binären Suchbaumes in einer geordneten Form zu durchlaufen. Dabei können wir beispielsweise die gesamte Anzahl der Knoten im binären Suchbaum bestimmen. Beim binären Suchbaum haben drei Traversierungen eine

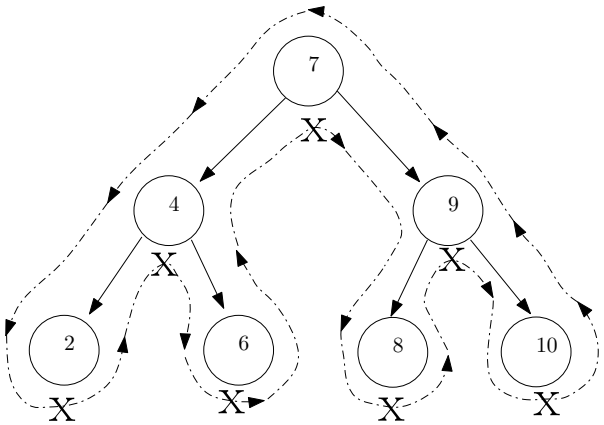


Abbildung 5: Umfahren wir einen Binärbaum gegen den Uhrzeigersinn, so erhalten wir die In-Order Traversierungsreihenfolge indem wir die value immer dann ausgeben wenn wir uns unter dem Knoten befinden.

besondere Bedeutung, die *In-Order Traversireung*, die *Pre-Order Traversierung*, und die *Post-Order Traversierung*. Diese wollen wir in diesem Abschnitt näher betrachten.

Die In-Order Traversierung ist speziell, da sie nur bei Binärbäumen angewendet werden kann, wohin gegen die Pre- und Post-Order Traversierung für alle Bäume verwendet werden kann.

Der Pseudocode für die In-order Traversierung beginnend von der Wurzel lautet wie folgt:

1. Starte an der Wurzel x und duchlaufe den linken Subbaum von in In-Order Reihenfolge, falls ein linker Subbaum vorhanden ist.

2. Gib die value des Knotens x aus.

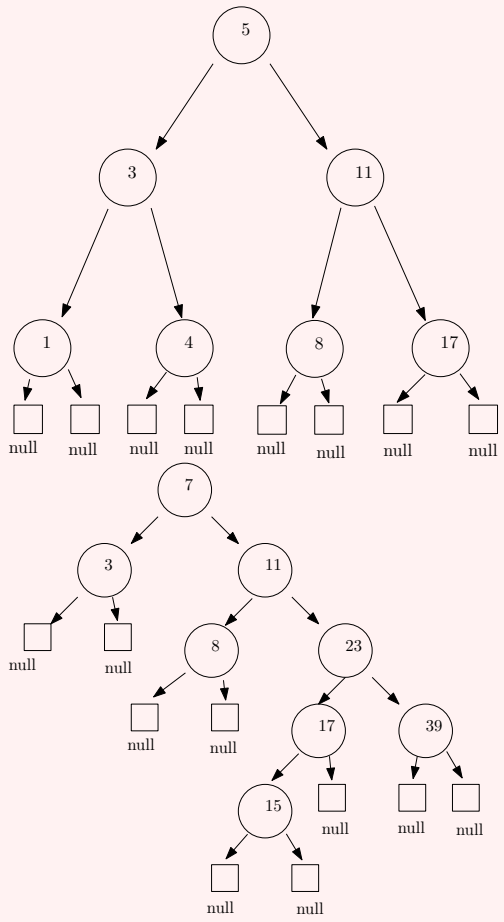
3. Durchlaufe den rechten Subbaum in In-Order Reihenfolge, falls ein rechter Subbaum vorhanden ist.

Umfahren wir den Binärbaum gegen den Uhrzeigersinn, so erhalten wir die In-Order Traversierung wenn wir die values jeweils dann ausgeben wenn wir uns unter dem Knoten befinden. Dies ist in Abbildung 5 dargestellt.

In der nächsten Aufgabe betrachten wir, was das Ergebnis einer solchen In-Order Traversierung ist.

Übung 0.13 ***In-Order Traversierung Durchführung*** Lösung S. 52

Gib die values der nachfolgenden Binärbäume in der Reihenfolge einer In-Order Traversierung an. Was fällt dir auf?



Dir ist vielleicht aufgefallen, dass in der In-Order Traversierung des binären Suchbaumes die Werte der Knoten im sortierter, aufsteigender Reihenfolge waren. Dies gilt generell für die In-Order Traversierung von binären Suchbäumen. Nimm dir einen Augenblick Zeit, um für dich zu bergünden warum dies der Fall ist. In der nächsten Aufgabe wandeln wir den Pseudocode in Python Code um.

Übung 0.14 ***In-Order Code*** **Lösung S. 52**

Schreibe Python Code, der die Knotenwerte eines Binärbaums in In-Order Reihenfolge zurückgibt.

Die In-Order Traversierung werden wir noch später bei der Löschmethode gebrauchen. Hier wollen wir aber zunächst die anderen zwei Traversierungen betrachten.

Der Pseudocode der Pre-Order Traversierung beginnend von der Wurzel ist wie folgt:

1. Starte bei der Wurzel x und gib die value von x aus.
2. Durchlaufe den linken Subbaum in Pre-Order Reihenfolge, falls ein linker Subbaum vorhanden ist.

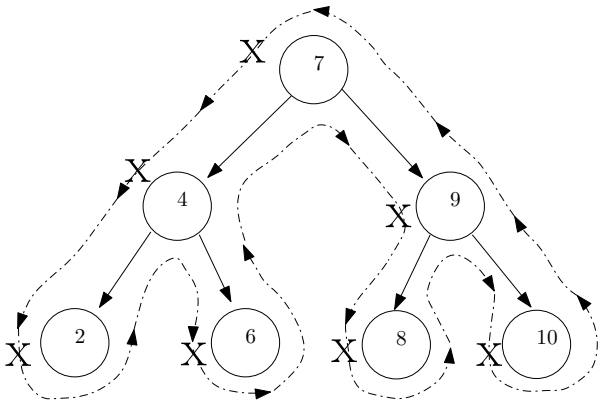


Abbildung 6: Die Pre-Order Reihenfolge erhalten wir, wenn wir den Baum gegen den Uhrzeigersinn umfahren und die values jeweils ausgeben wenn wir uns links von den Knoten befinden.

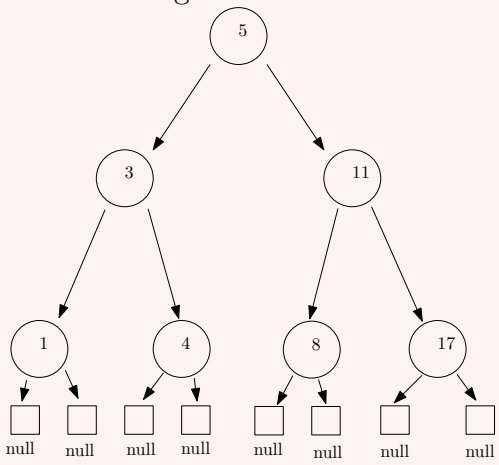
3. Durchlaufe den rechten Subbaum in Pre-Order Reihenfolge, falls ein rechter Subbaum vorhanden ist.

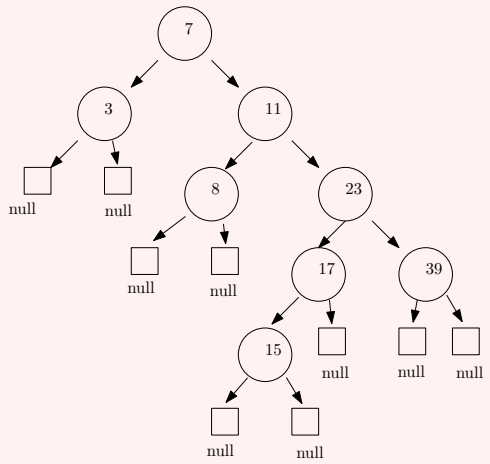
Wie man die Pre-Order Traversal Reihenfolge durch eine Umfahrung des Binärbaumes erhält, ist in Abbildung 6 dergestellt.

Wir schauen uns zunächst wieder die Reihenfolge an, die entsteht, wenn wir einen Binärbaum in einer Pre-Order Traversierung durchlaufen.

Übung 0.15 ***Pre-Order Traversierung Durchführung*** **Lösung S. 53**

Gib die values der nachfolgenden Binärbäume in der Reihenfolge einer Pre-Order Traversierung an.





Auch den Pseudocode der Pre-Order Traversierung wollen wir als in Python Code umwandeln.

Übung 0.16 ***Pre-Order Code*** **Lösung S. 53**

Schreibe Python Code, der die Knotenwerte eines Binärbaums in Pre-Order Reihenfolge zurückgibt.

Die Pre-Order Traversierung ist ein effizientes Tool, wenn man einen binären Suchbaum kopieren möchte. Traversieren wir einen binären Suchbaum B in Pre-Order Reihenfolge und fügen wir sie in dieser Reihenfolge in einen leeren Binärbaum, so ist der entstandene Binärbaum identisch zu B. Wir werden später noch sehen, dass es für jeden Datensatz mehr als nur einen korrekten binären Suchbaum gibt.

Zuletzt wollen wir uns noch die Post-Order Traversierung angucken. Bei dieser Traversierung durchläuft man alle Knoten im rechten und linken Subbaum eines Knotens, bevor man den Knoten selbst betrachtet. Der Pseudocode lautet wie folgt.

Der Pseudocode der Pre-Order Traversierung beginnend von der Wurzel ist wie folgt:

1. Starte bei der Wurzel x und durchlaufe den linken Subbaum in Post-Order Reihenfolge, falls ein linker Subbaum vorhanden ist.

2. Durchlaufe den rechten Subbaum von x in Post-Order Reihenfolge, falls ein rechter Subbaum vorhanden ist.

3. Gib die value des Knotens aus.

Die Post-Order Traversierung erhalten wir, wenn wir bei der Umfahrung des Binärbaumes in Richtung entgegen des Uhrzeigersinns, die Werte jeweils ausgeben, wenn wir uns rechts vom Knoten befinden (Siehe Abbildung 7.

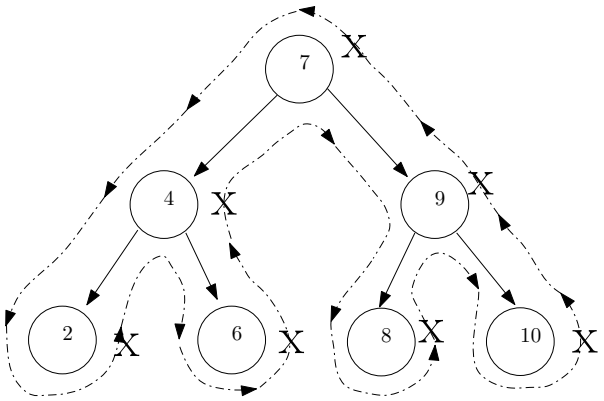
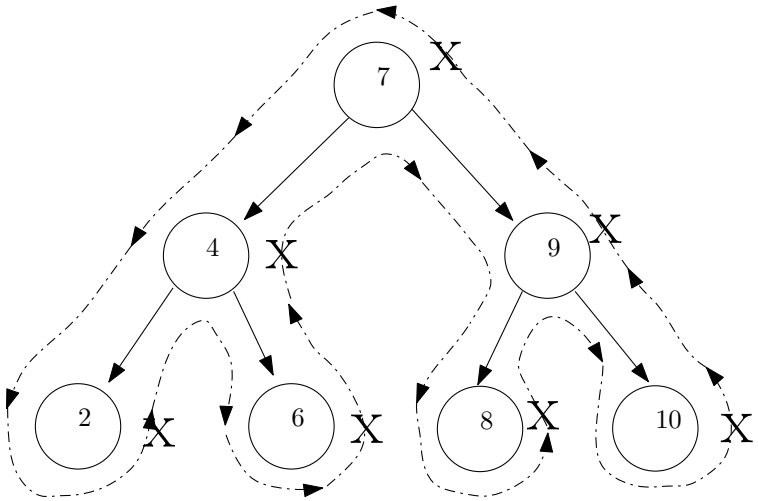


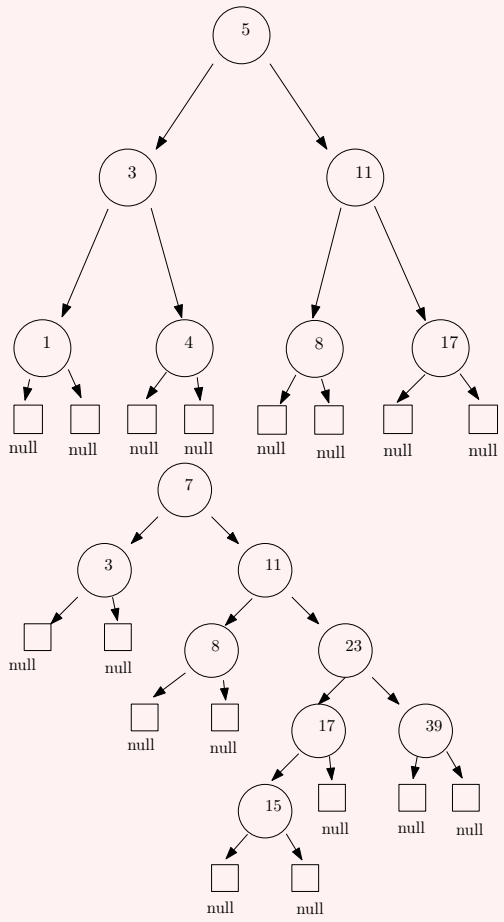
Abbildung 7: Die Post-Order Reihenfolge erhalten wir, wenn wir den Baum gegen den Uhrzeigersinn umfahren und die values jeweils ausgeben wenn wir uns rechts von den Knoten befinden.



Wie auch für die In-Order und Pre-Order Traversierung wollen wir zunächst das Ergebnis einer Post-Order Traversierung betrachten.

Übung 0.17 ***Post-Order Traversierung Durchführung*** Lösung S. 53

Gib die values der nachfolgenden Binärbäume in der Reihenfolge einer Post-Order Traversierung an.



Als Nächstes schreiben wir auch diesen Pseudocode wieder in Python code um.

Übung 0.18 ***Post-Order Code*** Lösung S. 54

Schreibe Python Code, der die Knotenwerte eines Binärbaums in Post-Order Reihenfolge zurückgibt.

Wir können die gelernten Traversierungen nicht nur nutzen, um alle values in einem Binärbaum zu lesen, sondern sie dienen uns auch, um binäre Suchbäume zu beschreiben. Wir werden später sehen, dass es für denselben Datensatz verschiedene binäre Suchbäume gibt. Beschreiben wir aber einen binären Suchbaum mittels einer der Traversierungen, so limitieren wir die möglichen binären Suchbäume. In der nächsten Aufgabe wirst du aufgefordert, binäre Suchbäume so zu zeichnen, dass sie den vorgegebenen Traversierungen entsprechen.

Übung 0.19 ***Konstruiere nach Traversierung*** Lösung S. 54

- Konstruiere einen Binärbaum, dessen Pre-Order Traversierung d, b, a, c, f, e ist.
- Konstruiere einen Binärbaum, dessen In-Order Traversierung (, -, (, -,), (, -,), -,). Das heisst, die Knoten haben values "(", ")ünd ". Gibt er hier mehr als eine Möglichkeit?
- Konstruiere einen Binärbaum, dessen Pre-Order Traversierung (, -, (, -, (,), -,), -,) und In-Order Traversierung (, -, (, -,), (, -,), -,) ist.
- Konstruiere einen binären Suchbaum, dessen Post-Order Traversierung I, IV, III, VIII, XVII, XI, V ist. (Achte darauf, dass wir hier römische Zahlen verwendet haben.)

Was du gelernt hast:
In diesem Unterkapitel haben wir die In-Order, Pre-Order und Post-Order Traversierung kennengelernt, und gesehen wie wir sie aus einer Umfahrung des Binärbaumes entgegen dem Uhrzeigersinn herleiten können.

Finden im binären Suchbaum

Wenn wir eine Menge an Datenpunkten speichern, ist es wichtig, dass wir feststellen können, ob ein gewisser Datenpunkt vorhanden ist oder nicht. Die *find*-Methode, die wir in diesem Abschnitt entwickeln, lässt uns feststellen, ob eine Value x in einem binären Suchbaum vorhanden ist.

Die Spezifikation der *find()* Methode ist wie folgt:

- *find(x)* - Gibt eine Boolean zurück, welche 1 ist, wenn es im binären Suchbaum einen Knoten mit Value x gibt, sonst 0.

In der nächsten Aufgabe wollen wir gemainsam die Implementierung der *find*-Methode entwickeln. Hierzu nutzen wir die Eigenschaften des binären Suchbaums.

Übung 0.20 ***Find Methode entwickeln*** Lösung S. 55

In dieser Aufgabe wollen wir die *find(x)*-Methode gemeinsam entwickeln. Ähnlich zur Linked List haben wir bei einem binären Suchbaum B zunächst Zugriff auf die Wurzel und können dann die Pointer benutzen, um die anderen Elemente im Baum zu erreichen.

a) Angenommen, der Baum ist nicht leer. Wenn wir x und $B.root.value$ vergleichen, gibt es 3 Möglichkeiten. Wie sind diese und was machst du in jedem der 3 Fälle, um x in B zu finden?

b) Welche weitere Möglichkeit entsteht, wenn der Baum auch leer sein kann?

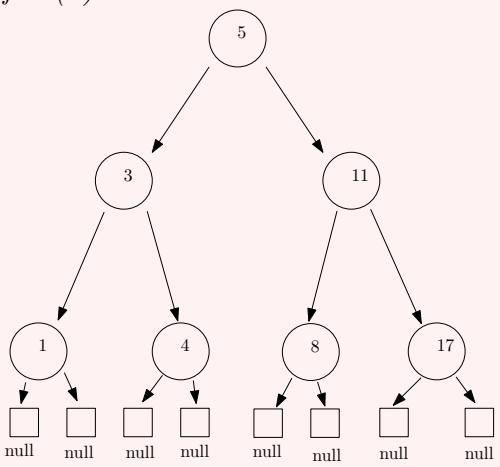
c) Entwickle anhand Teilaufgaben a) und b) einen rekursiven Algorithmus, der die find(x)-Methode implementiert.

In der nächsten Aufgabe kannst du beweisen, dass du verstanden hast, wie die *find*-Methode vorgeht.

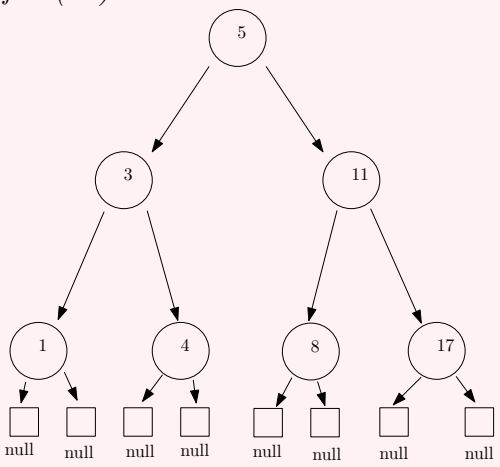
Übung 0.21 ***Pfade des Findes*** Lösung S. 56

Markiere in jeder Teilaufgabe, den Pfad, welcher während der Suche traversiert wird:

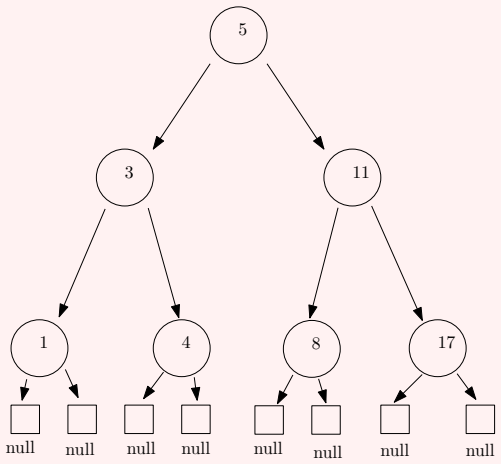
- *find(3)*



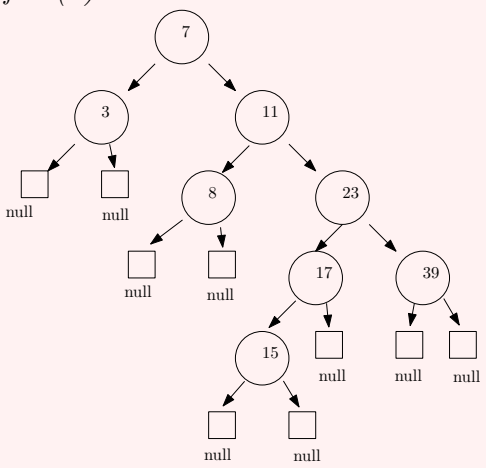
- *find(17)*



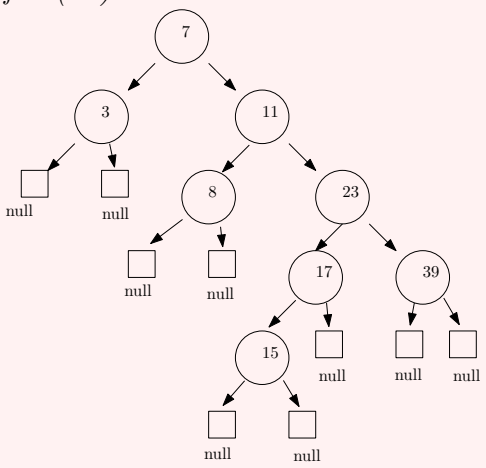
- *find(9)*



- *find(9)*



- *find(15)*

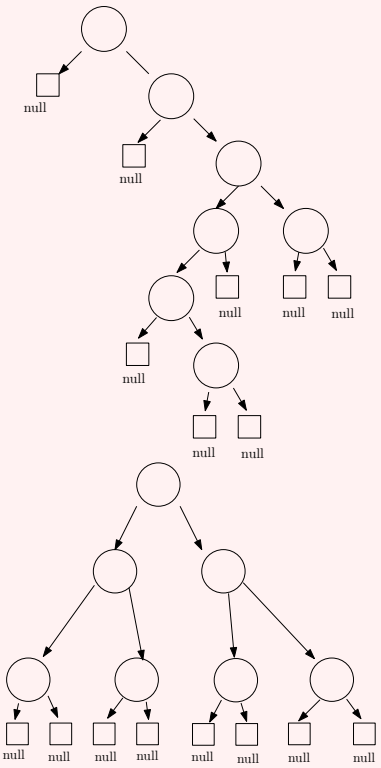


In der Informatik wollen wir in der Regel einen Algorithmus entwerfen, der mit möglichst wenig Schritten eine Lösung zu unserem Problem findet. In der nächsten Aufgabe analysieren wir, in welcher Art von Binärbaum die *find()*-Methode am effizientesten

ist.

Übung 0.22 ***Schnelles finden*** Lösung S. 58

Bestimmen Sie die maximale Anzahl an Vergleichen, die in den beiden dargestellten Suchbäumen erforderlich sind. Hierbei bezeichnet "maximal"die höchste Anzahl an Vergleichen, die im ungünstigsten Fall nötig ist, um ein Element zu finden.



Kannst du eine generelle Regel erkennen?

Konzepte und Begriffe:
Die *Tiefe* (depth) eines Knotens x entspricht der Anzahl Knoten auf dem Pfad von der Wurzel des Baumes bis zum Knoten x. Die *Höhe* (height) eines Baumes B entspricht der maximalen Tiefe eines Knotens

$$height(B) = \max depth(x) \quad \text{wobei } x \text{ ein Knoten von } B \text{ ist.}$$

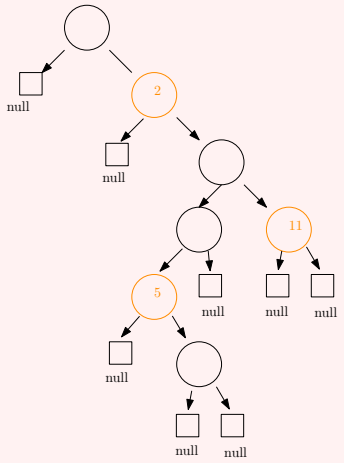
Knoten, welche dieselbe Tiefe haben, werden zu *Level* zusammen gefügt.

Die folgende Aufgabe überprüft, dass du die Begriffe Höhe und Tiefe korrekt verwenden kannst.

Übung 0.23 *Höhen messen*

Lösung S. 58

Bestimme die Tiefe der drei markierten Knoten. Was ist die Höhe des Baumes?



Wenn du Lust hast, kannst du in der nächsten Aufgabe überlegen, wie du einen Algorithmus programmierst, der die Höhe eines Binärbaumes bestimmt.

Übung 0.24 *Höhe rekrusiv bestimmen*

Lösung S. 58

Wie kannst du die Höhe eines Binärbaums rekursiv bestimmen?

Die Höhe eines binären Suchbaumes entspricht der maximalen Anzahl an Vergleichen, die wir in einer *find(x)*-Operation machen müssen. In Aufgabe 0.22 haben beide Bäume dieselbe Anzahl Knoten, dennoch haben sie unterschiedliche Höhen.

Die nächste Aufgabe fordert dich auf, dir zu überlegen, wie ein binärer Suchbaum am besten aufgebaut ist, damit die Worstcase Laufzeit der *find*-Methode möglichst gering ist.

Übung 0.25 *Optimale Strukturen*

Lösung S. 59

Überlege dir die optimale Struktur eines binären Suchbaumes, um die Anzahl Vergleich im schlechtesten Fall zu minimieren.

Schaffst du es schon, die minimale und die maximale Höhe eines binären Suchbaumes für eine fixe Anzahl Knoten zu bestimmen? Dies kannst du in der nächsten Aufgabe unter Beweis stellen.

Übung 0.26 ***Grosse und kleine Bäume*** Lösung S. 59

Was ist die minimale Höhe $h_{\min}$ eines Binärbaumes, der 23 Knoten enthält? Was ist die maximale $h_{\max}$ Höhe eines Baumes, der 23 Knoten enthält? Kannst du eine generelle Formel entwickeln für die minimale und die maximale Höhe eines Baumes, der n Knoten enthält?

Was du gelernt hast:
In diesem Kapitel haben wir die *find*-Methode selbstständig entwickelt. Zusäzlich haben wir gesehen, dass die Laufzeit der *find*-Methode stark von der Höhe des Baumes abhängt, und wir haben überlegt, welche Baumstrucktur es uns erlaubt, effizient die *find*-Methode auszuführen.

Einfügen im binären Suchbaum

Wie jede Datenstruktur ist auch der binäre Suchbaum nach seiner Initialisierung zunächst leer. Um ihn zu nutzen, müssen wir Datenpunkte hinzufügen und entfernen können. In diesem Abschnitt untersuchen wir, wie man Datenpunkte in einen binären Suchbaum hinzufügt.

Die Spezifikation der *add*-Methode ist wie folgt:

- *add(x)* - Fügt einen Knoten mit value x in den binären Suchbaum hinzu, sofern dieser nicht bereits vorhanden ist.

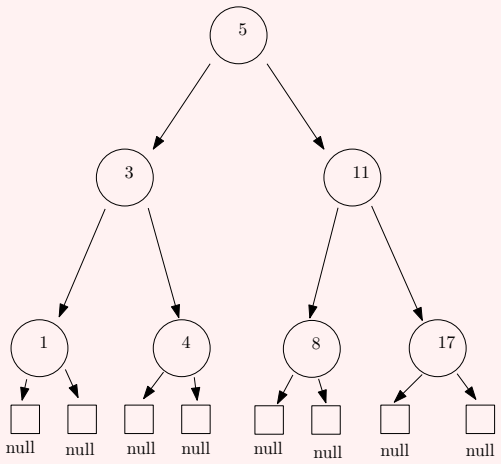
Damit beim Hinzufügen einer value in einen binären Suchbaum möglichst wenig Aufwand entsteht, wird eine neue value immer als Blatt in den binären Suchbaum eingefügt. Dadurch muss nur ein Knoten seinen Pointer von null auf den neuen Knoten ändern und wir haben die Garantie, dass der Baum weiterhin verbunden ist.

Versuche in der nächsten Aufgabe jeweils die geforderte value so hinzuzufügen, dass der binäre Suchbaum bestehen bleibt.

Übung 0.27 ***Add Methode anwenden*** Lösung S. 59

Alle Teilaufgaben in dieser Aufgabe verwenden denselben binären Suchbaum als Ausgangspunkt.

- Füge im abgebildeten Baum die value 7 hinzu.
- Füge im abgebildeten Baum die value 11 hinzu.
- Füge im abgebildeten Baum die value 2 hinzu.



Dir ist in der letzten Aufgabe vielleicht bereits aufgefallen, dass es immer genau eine Stelle im binären Suchbaum gibt, an dem wir eine neue value hinzufügen können. Dies wollen wir in der folgenden Aufgabe mathematisch beweisen.

Übung 0.28 ***Einzigartigegs Einfügen*** Lösung S. 60

Zeige, dass unter der Annahme, dass x derzeit nicht im Baum ist, es genau eine Stelle im binären Suchbaum gibt, an dem die value x als Blatt hinzugefügt werden kann, und die Eigenschaften des binären Suchbaums erhalten bleiben.

Mit diesen Erkentnissen sind wir in der Lage, die *add*-Methode zu entwickeln.

Übung 0.29 ***Add Methode entwickeln*** Lösung S. 60

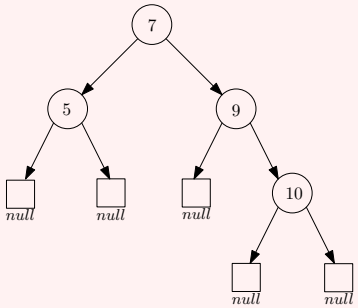
Bevor wir eine value in den binären Suchbaum hinzufügen können, müssen wir überprüfen, ob die value bereits vorhanden ist. Dies ähnelt der *find()*-Methode.

- Inwiefern unterscheiden sich die Basecases der *find(x)*- und der *add(x)*-Methode?
- Entwickle anhand dieser Erkenntnisse die *add(x)*-Methode.

In der nächsten Aufgabe wollen wir unser gelerntes Wissen testen und gleich mehrere values hintereinander in den binären Suchbaum einfügen.

Übung 0.30 ***Values hinzufügen*** Lösung S. 61

- Füge im abgebildeten Baum nacheinander die values 4, 3, 8, und 11 hinzu.
- Füge im abgebildeten Baum nacheinander die values 3, 8, 11 und 4 hinzu.
- Was fällt dir auch



Was du gelernt hast:
In diesem Abschnitt haben wir die *add*-Methode des binären Suchbaumes kennengelernt. Wir haben gezeigt, dass es in einem binären Suchbaum immer genau eine Stelle gibt, an der man eine noch nicht vorhandene value als Blatt hinzufügen kann und die Eigenschaften des binären Suchbaumes erhalten bleiben.

Bau eines binären Suchbaums

Im letzten Abschnitt haben wir gelernt, wie man values in einen binären Suchbaum hinzufügt. Somit sind wir bereits in der Lage binäre Suchbäume für einen gegebenen Datensatz zu bauen: Wir fügen sukzessive die values in einen zu Begin leeren binären Suchbaum ein. Wir haben in Aufgabe 0.28 gezeigt, dass es für jede value immer genau eine Stelle im binären Suchbaum gibt, an dem man die value hinzufügen kann und die Eigenschaften eines binären Suchbaumes erhalten bleiben. Die nächste Aufgabe zeigt, dass es für dieselben values unterschiedliche binäre Suchbäume gibt.

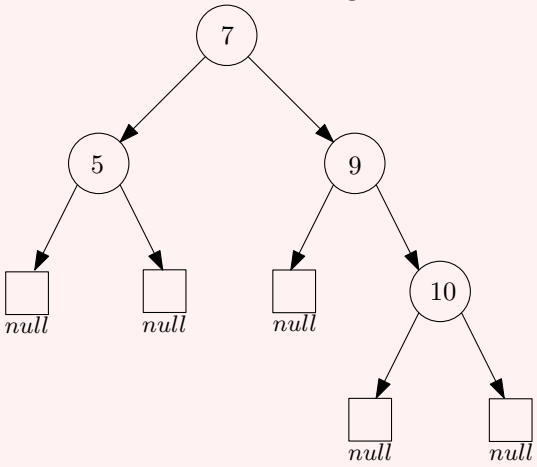
Übung 0.31 ***3-Knoten Bäume*** Lösung S. 62

Zeichne alle Suchbäume für die values 1, 2, 3. Macht es einen Unterschied, was die eigentlichen values sind, oder gibt es für jedes beliebige Tripel an drei values die gleiche Anzahl binärer Suchbäume?

Die Struktur eines binären Suchbaumes hängt davon ab, in welcher Reihenfolge die values in den binären Suchbaum eingefügt werden. Jedoch führt nicht jede Reihenfolge zwingend zu einem anderen binären Suchbaum, wie die nächste Aufgabe zeigt.

Übung 0.32 ***Wie ist der Baum entstanden?*** Lösung S. 62

In welcher Reihenfolge wurden die values in diesen binären Suchbaum eingefügt? Gibt es mehr als eine mögliche Reihenfolge?



Wir wissen also, dass die Anzahl der binären Suchbäume für n verschiedene values, höchstens $n!$, die Anzahl an verschiedenen Reihenfolgen ist. Gleichzeitig ist n eine untere Schranke für die Anzahl an binären Suchbäumen mit n Knoten, da es für jede value immer mindestens einen binären Suchbaum mit dieser value als Wurzel gibt.

Eine Frage, die du dir jetzt vielleicht stellst, ist, wie viele unterschiedliche binäre Suchbäume es für ein n values gibt. Dies leiten wir in der folgenden Aufgabe her.

Übung 0.33 ***Bäume zählen*** Lösung S. 62

Angenommen wir bauen einen binären Suchbaum für die values $1, 2, \ldots n$.

- Wie viele Möglichkeiten haben wir, die Wurzel auszuwählen?
- Wie viele Möglichkeiten haben wir, die Wurzel, das recht Kind der Wurzel und das linke Kind der Wurzel auszuwählen?
- Wie viele binäre Suchbäume für die values $1, 2, \ldots, n$ gibt es insgesamt?

Konzepte und Begriffe:
Die Zahlenfolge $C(N) = \frac{(2n)!}{(n+1)!n!}$ ist bekannt als *Catalan-Zahl*. Die catalanschen Zahlen bilden eine Folge natürlicher Zahlen, die nicht nur die Anzahl an binären Suchbäumen zählen, sondern auch darüber hinaus häufig in der Kombinatorik vorkommen. So beschreibt $C(n)$, die n-te catalansche Zahl, etwa auch die Anzahl an monotonen Pfade entlang der Ränder eines Quadratgitters mit der Grösse $n \times n$, die keinen Punkt oberhalb der Diagonale enthalten, oder die der Anzahl der Klammerungen eines Produktes, in dem n Multiplikationen vorkommen. Die ersten 10 Catalan Zahlen $C_0, C_1, \ldots C_9$ lauten $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862$. Bist du froh, dass du nicht alle binären Suchbäume mit 6 values zeichnen musstest?

Wenn es also viele verschiedene valide binäre Suchbäume für den gleichen Datensatz gibt, dann fragst du dich vielleicht, welcher binäre Suchbaum der Beste wäre.

In der Informatik sind wir interessiert daran, dass die einzelnen Methoden eines Algorithmus möglichst wenig Schritte benötigen. Als wir die *find(x)*-Methode kennengelernt haben, haben wir gesehen, dass die maximale Anzahl an Vergleichen, die während eines Aufrufs von *find* gemacht werden, gleich der Höhe eines Baumes ist. In der nächsten Aufgabe betrachten wir den Zusammenhang zwischen der Anzahl Schritte während des Einfügens und der Struktur des binären Suchbaumes.

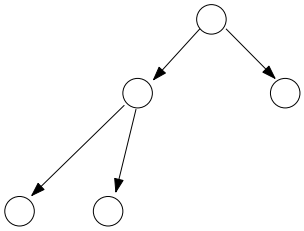
Übung 0.34 ***Performance Einfügen*** Lösung S. 63

Was ist die Worstcase Anzahl an Vergleichen, die wir beim Einfügen in einen binären Suchbaum machen müssen?

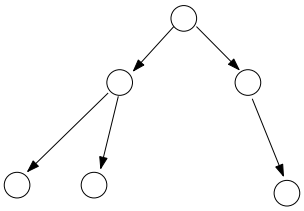
Die Laufzeit beim Einfügen hängt also wie die Laufzeit beim Finden von der Struktur des binären Suchbaumes ab. Es gibt einige Strukturen von Binärbäumen, die ihren eigenen Namen haben, welche wir als Nächstes kennenlernen.

Konzepte und Begriffe:
Ein Binärbaum, bei dem jeder Knoten entweder auf zwei andere Knoten oder zweimal auf null zeigt, wird als *Voller Binärbaum* bezeichnet. In einem Binärbaum, bei dem jedes Level, bis auf das unterste, die maximale Anzahl Knoten hat, und bei dem das unterste Level von links nach rechts befüllt ist, wird als *Kompletter Binärbaum* bezeichnet. Ein Binärbaum, bei dem jeder Knoten, der kein Blatt ist, genau zwei Kinder hat, wird als *Perfekter Binärbaum* bezeichnet. In einem *Balancierter Binärbaum* unterscheiden sich die Höhen von rechten und linken Subbaum eines jeden Knoten um maximal 1.
Beispiele für die drei verschiedenen Typen von Binärbäumen siehst du in Abbildung 8. Der einfacheithalber haben wir die values der Knoten nicht angezeigt.

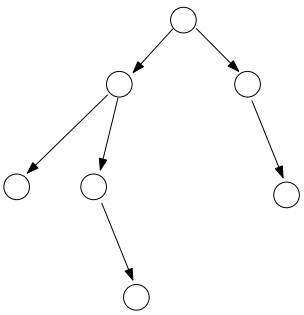
Die gegebenen Definitionen erscheinen auf den ersten Eindruck sehr abstrakt. Daher wollen wir in den nächsten zwei Aufgaben die Definitionen konkreter machen.



(a) Beispiel eines vollen Binärbaums.



(b) eispiel eines kompletten Binärbaums.



(c) eispiel eines perfekten Binärbaums.

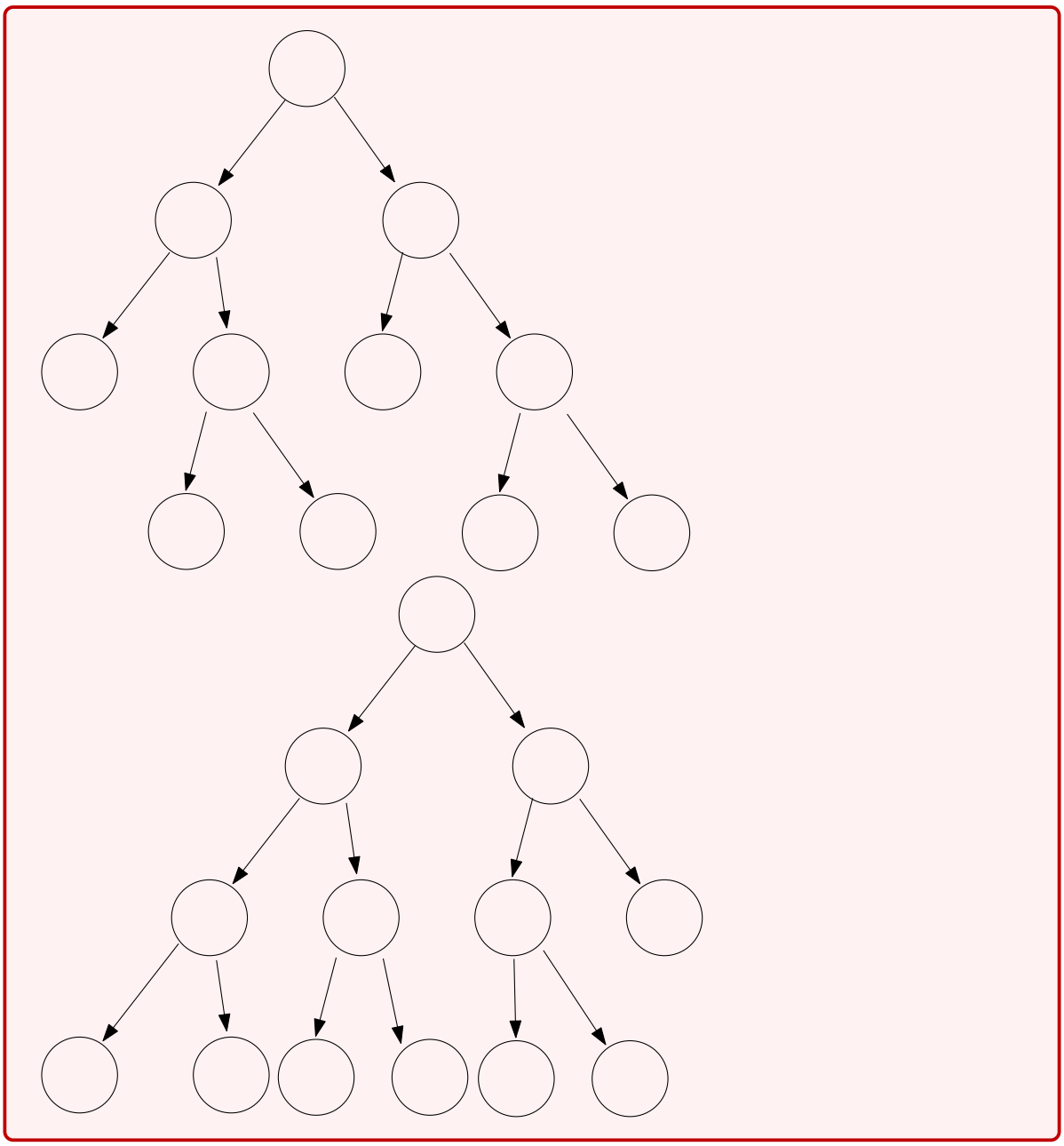
Abbildung 8: Beispiele der drei Baumtypen.

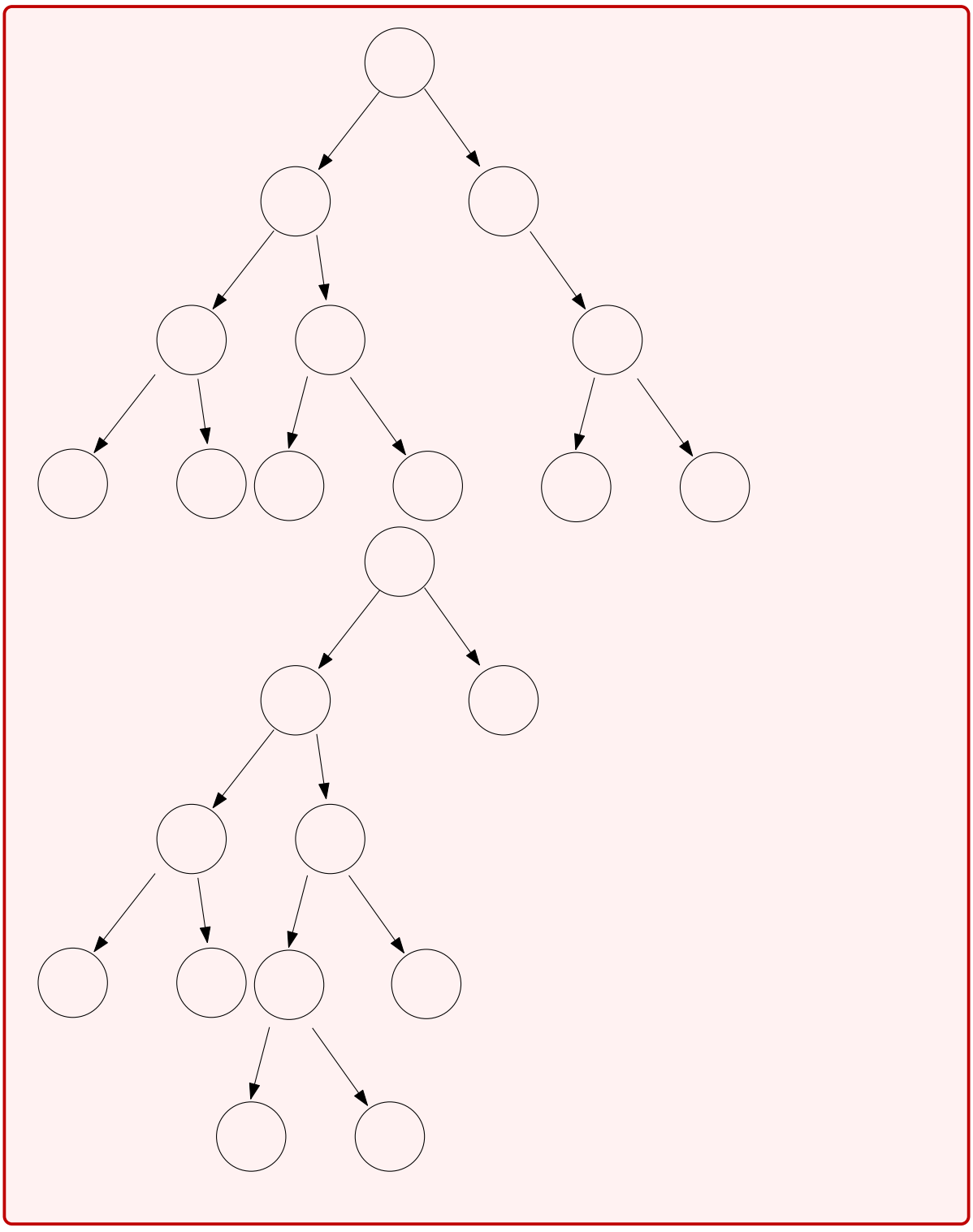
Übung 0.35 ***Bäume zeichnen*** Lösung S. 63

- Zeichne (wenn möglich) einen vollen Binärbaum bestehend aus 10 Knoten. Falls es nicht möglich ist, begründe warum.
- Zeichne (wenn möglich) einen kompletten Binärbaum bestehend aus 12 Knoten. Falls es nicht möglich ist, begründe warum.
- Zeichne (wenn möglich) einen perfekten Binärbaum bestehend aus 7 Knoten. Falls es nicht möglich ist, begründe warum.
- Zeichne (wenn möglich) einen balancierten Binärbaum bestehend aus 8 Knoten.

Übung 0.36 ***Strukturen erkennen*** Lösung S. 65

Gib für jeden der nachfolgenden Binärbäume an, ob er voll, komplett, perfekt oder balanciert ist. (Es kann mehr als eine Antwort richtig sein.)





Als Nächstes wollen wir verstehen, ob sich eine dieser Baumstrukturen besonders für einen binären Suchbaum eignet. Hierfür wollen wir als Erstes untersuchen, ob man überhaupt für jede Anzahl an Knoten eine solche Baumstruktur bauen kann. Dies ist wichtig, da es beim binären Suchbaum keine Beschränkungen gibt bezüglich der An-

zahl Knoten. Daher müssen wir sicherstellen, dass wir uns nicht auf einen Baumtypen festlegen, welcher dies nicht ermöglicht.

Übung 0.37 ***Knotenzahlen*** Lösung S. 65

Gibt für jede der nachfolgenden Aussagen an, ob sie wahr oder falsch sind. Wenn eine Aussage falsch ist, dann überlege dir, was die richtige Aussage wäre.

- Volle binär Bäume haben immer einer gerade Anzahl Knoten.
- Für jedes $n \in \mathbb{N}_{\geq 0}$ gibt es einen vollen Binärbaum, der genau n Knoten enthält.
- Es gibt nur dann einen perfekten Binärbaum mit n Knoten, wenn $n = 2^k$ für $k \in \mathbb{N}$, oder $n = 0$ ist.
- Balancierte Binärbäume haben immer eine gerade Anzahl Knoten.

Aus Aufgabe 0.37 geht hervor, dass man für jedes n einen kompletten oder balancierten binären Suchbaum bauen kann. Es gibt aber n für welche es keinen vollen und keinen perfekten binären Suchbaum gibt.

Als Nächstes wollen wir uns anschauen, was die minimalen und maximalen Höhen für eine fixe Anzahl an Knoten sind. Da wir wissen, dass die Worstcase Laufzeit vom Suchen und Finden im binären Suchbaum proportional zur Höhe des Baumes ist, wäre es interessant zu wissen, was die maximale Höhe für eine fixe Anzahl an Knoten in den jeweiligen Baumtypen ist.

Übung 0.38 ***Minimale Höhen*** Lösung S. 65

Zeige, dass alle 4 Binärbaumtypen (voll, komplett, perfekt, balanciert) eine Minimalhöhe von $h = \log_2(n+1)$ haben (wobei wir annehmen, dass $n = 2^k - 1$ für $k \in \mathbb{N}$ ist).

Übung 0.39 ***Maximale Höhen*** Lösung S. 66

Für $n = 15$, was ist die maximale Höhe eines vollen, kompletten, perfekten und balancierten Binärbaumes? Kannst du deine Beobachtung verallgemeinern? Was ist die maximale Höhe der jeweiligen Bäume für ein beliebiges n? Beschreibe, wie du diese konstruieren kannst.

Aus Aufgabe 0.39 folgt, dass komplette, perfekte und balancierte Binärbäume eine Höhe haben, die logarithmisch zu der Anzahl Knoten im Baum ist. In Aufgabe 0.26 haben

wir zusätzlich gesehen, dass die minimale Höhe eines Binärbaums auch logarithmisch zur Anzahl der Knoten im Baum ist. Das heisst, dass komplette, perfekte und balancierte Binärbäume quasi die minimale mögliche Höhe eines Binärbaumes erreichen.

In der nächsten Aufgabe erarbeiten wir, wie wir aus einem gegebenen Datensatz einen kompletten binären Suchbaum erstellen.

Übung 0.40 ***binäre Suchbäume bauen*** Lösung S. 66

Angenommen, wir haben einen Datensatz, der aus den Werten $1, 2, \ldots, n$ besteht.

- Welche Höhe hat ein kompletter Binärbaum, der aus n Knoten besteht?
- Wie viele Knoten sind auf dem untersten Level?
- Wie hoch ist der rechte und der linke Subbaum der Wurzel?
- Welchen Knoten solltest du als Wurzel für den kompletten binären Suchbaum verwenden? (Hinweis: Überlege dir, wie viele Knoten jeweils im rechten und linken Subbaum sein müssen.)
- Überlege dir, wie du einen kompletten binären Suchbaum aus den Werten $1, 2, \ldots, n$ bauen kannst.

In der nächsten Aufgabe kannst du deinen Algorithmus aus der letzten Aufgabe testen.

Übung 0.41 ***Baue komplette binäre Suchbäume*** Lösung S. 68

Baue für jeden der folgenden Datensätze einen kompletten binären Suchbaum.

- $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$
- $3, 6, 9, 10, 14, 17, 19$
- $4, 2, 7, 11, 10, 14$

Was du gelernt hast:
Wir haben gesehen, dass man mit der *add*-Methode einen binären Suchbaum für einen beliebigen Datensatz bauen kann. Für jeden Datensatz gibt es aber viele verschiedene binäre Suchbäume. Die Anzahl der binären Suchbäume wird mit der catalansche Zahl beschrieben. Jedoch gibt es einige binäre Suchbäume, die deutlich grösser sind als andere. Um möglichst effizient mit den binären Suchbäumen arbeiten zu können, sollten wir versuchen, die Höhe gering zu halten. Eine Möglichkeit, dies zu tun, ist, den Suchbaum als kompletten binären Suchbaum zu bauen.

Löschen im binären Suchbaum

Bis jetzt haben wir uns damit beschäftigt, wie Datenpunkte in einem binären Suchbaum findet oder hinzufügt. Manchmal möchte man aber auch Datenpunkte aus einem Datensatz entfernen. Dies wollen wir in diesem Abschnitt lernen.

Je nachdem, welcher Knoten im Baum entfernt wird, hat dies unterschiedliche Auswirkungen auf den binären Suchbaum. Unsere Aufgabe ist es, die Datenstruktur wieder so zu reparieren, dass das Resultat wieder ein binärer Suchbaum ist.

Als Erstes wollen wir untersuchen, wie die Position des Knotens im Baum, das Resultat des Löschens beeinflusst.

Übung 0.42 ***Bäume zerlegen*** Lösung S. 69

Was ist das Resultat, wenn ...

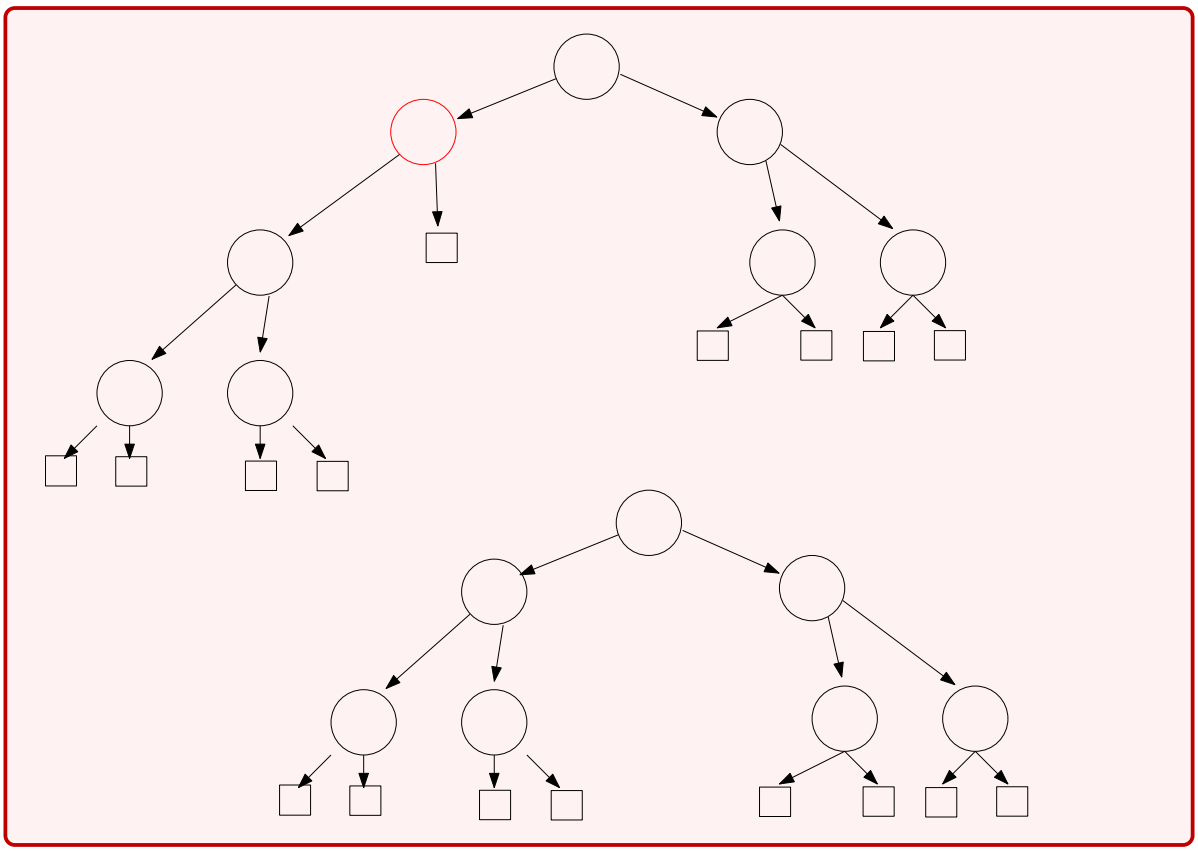
- ... wir die Wurzel von einem binären Suchbaum entfernen?
- ... ein Blatt vom binären Suchbaum entfernen?
- ... einen Knoten, der weder Wurzel noch Blatt ist, vom binären Suchbaum entfernen?

Aus Aufgabe 0.42 geht hervor, dass wenn wir ein Blatt aus einem binären Suchbaum entfernen, wir weiterhin einen binären Suchbaum haben. Wenn wir aber einen Knoten aus dem binären Suchbaum entfernen, der kein Blatt ist, dann zerfällt der binäre Suchbaum in mehrere Teile. Wir können diesen Fall noch einmal in zwei Unterfälle unterteilen, je nachdem ob der entfernte Knoten ein oder zwei Kinder hat.

Wir betrachten zunächst den Fall, dass der entfernte Knoten genau ein Kind hat.

Übung 0.43 ***Löschen einfach*** Lösung S. 69

Zeige, dass wenn wir einen Knoten, der nur ein Kind hat, mit seinem Kind ersetzen, das Resultat weiterhin ein binärer Suchbaum ist. Die Konstruktion ist auch in der unteren Abbildung dargestellt.



In Aufgabe 0.43 haben wir gesehen, dass wenn der Knoten, den wir entfernen, nur ein Kind hat, so können wir den Knoten mit seinem Kind ersetzen und erhalten wieder einen binären Suchbaum. Dies, wie auch der Fall wo der zu löschende Knoten keine Kinder hat, wird ein wichtiger Teil der generellen Löschmethode sein. In der nächsten Aufgabe wollen wir zunächst diese beiden Unterfälle der Löschmethode implementieren.

Übung 0.44 ***Einfaches Löschen Implementieren*** Lösung S. 70

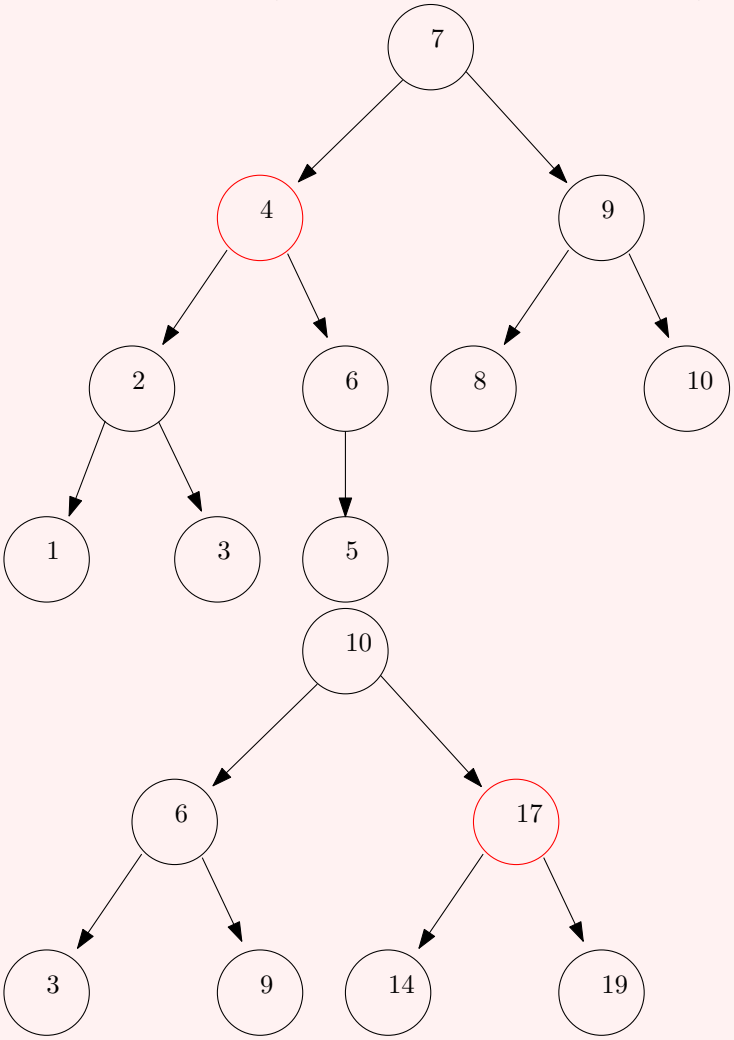
Schreibe die Python Methoden *delSimpel* welche in der Lage ist, Blätter und Knoten, die nur ein Kind haben zu entfernen.
Baue diese Methode so in Python Code ein, dass sie zunächst den zu löschenden Knoten finden.

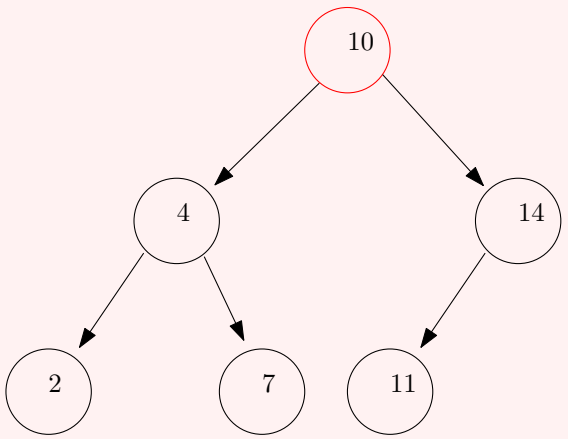
Als Nächstes wollen wir den komplizierteren Fall betrachten, bei dem der zu entfernende Knoten zwei Kinder hat. Damit der binäre Suchbaum nicht in mehrere Teile zerfällt, werden wir ihn mit einem anderen Knoten im Baum ersetzen. Wir wissen bereits, wie man Knoten, die höchsten ein Kind haben, entfernt. Unser Ziel ist es daher, den Knoten, der zwei Kinder hat, mit einem Knoten x, der höchsten ein Kind hat, zu ersetzen. In einem zweiten Schritt löschen wir dann Knoten x an seiner ursprünglichen Position. Um herauszufinden, welchen Knoten wir an die Stelle des zu löschenden Knotens setzen, analysieren wir zunächst, welche values an die Position des zu löschenden Knotens setzen

können.

Übung 0.45 ***Ersatzraum*** Lösung S. 70

In den abgebildeten Bäumen ist jeweils ein Knoten rot markiert. Welche values können an diese Postion geschrieben werden, sodass das Ergebnis weiterhin ein binärer Suchbaum ist? Markiere alle Knoten im linken und rechten Subbaum des markierten Knotens, die höchstens ein Kind haben, die in dieser Range liegen.





In Aufgabe 0.45 gab es immer genau zwei Knoten, welche man an die Position des markierten Knotens setzen könnte, sodass die Struktur des binären Suchbaumes erhalten geblieben ist. In der nächsten Aufgabe zeigen, dass es immer genau zwei Knoten gibt, die man an die Position setzen kann, und dass diese Knoten immer höchstens ein Kind haben.

Übung 0.46 ***Vorgänger und Nachfolger*** Lösung S. 72

Zeige, dass es für jeden Knoten x der zwei Kinder hat, immer genau zwei Knoten im Subbaum mit Wurzel x gibt, die man an die Position von x setzen kann, sodass das Resultat weiterhin ein binärer Suchbaum ist.

Als Nächstes wollen wir uns überlegen, wie wir die zwei Knoten, die wir an die Position des zu löschenden Knotens setzen, im binären Suchbaum finden können.

Übung 0.47 ***Löschen Implementieren*** Lösung S. 72

Sei x ein Knoten im binären Suchbaum, der zwei Kinder hat. Schreibe Pseudocode, um den Knoten mit der grössten value im rechten Subbaum von x zu finden, und den Knoten mit der kleinsten value im rechten Subbaum von x zu finden.

Bei der Einführung der In-Order Traversierung haben wir angekündigt, dass wir diese noch einmal betrachten, werden wir die Löschmethode einführen. Dies wollen wir in der nächsten Aufgabe tun.

Übung 0.48 ***Verhältnisse erkennen*** Lösung S. 73

Sei x ein Knoten im binären Suchbaum, der zwei Kinder hat. Wenn wir den gesamten binären Suchbaum betrachten, stehen die zwei Knoten, die an die Position von x treten können, in welchem Verhältnis zu x?
Hinweis: An welchen Postionen tauchen sie in der In-Order Traversierung auf.

Mit diesem Wissen sind wir in der Lage die *delete(x)*-Methode selber zu implementieren. Die Spezifikation der Methode lauter wie folgt:

- *delete(x))* - Entfernt (falls vorhanden) den Knoten mit value x auf eine Art, dass die übrige Datenstruktur weiterhin ein binärer Suchbaum ist.

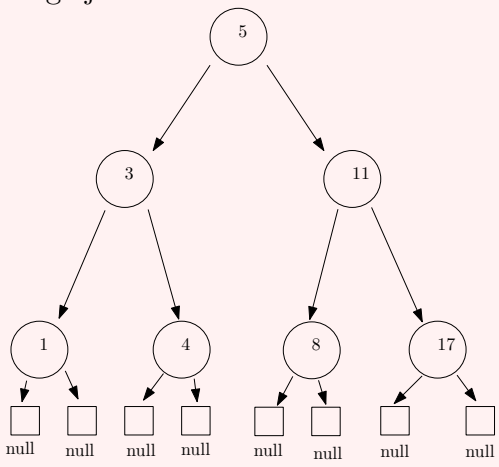
Übung 0.49 ***Implementiere delete(x)*** Lösung S. 73

Implementiere die *delete(x)*-Methode. Du darfst dabei selber entscheiden, ob du x mit der nächst grösseren, oder der nächst kleineren value ersetzen möchtest.
Hinweis: Es empfiehlt sich zunächst auf Papier einen binären Suchbaum zu zeichen. Zeichne zusäzlich das Resultat wenn du einen beliebigen Knoten entfernst.

Obwohl es beim Löschen eines Knotens aus dem binären Suchbaum zwei Möglichkeiten gibt, mit welchem Knoten der zu löschende Knoten ersetzt wird (sofern der Knoten 2 Kinder hat), wird der Einfachheit halber in der Regel nur eine der Methoden implementiert. Dies hat zur Folge, dass, selbst wenn wir mit einem balancierten Binärbaum beginnen, der Baum durch wiederholtes Löschen sehr aus der Balance geraten kann. Dies zeigt auch die nächste Aufgabe.

Übung 0.50 ***Delete Methode anwenden*** Lösung S. 75

Lösche im abgebildeten binären Suchbaum nacheinander die Knoten 3, 11, 17, 8. Zeige jeden Zwischenschritt.



Egal, ob wir den zu entfernenden Knoten mit seinem Vorgänger oder seinem Nachfolger in der In-Order Reihenfolge ersetzen, kann es passieren, dass ein balancierter Baum durch wiederholtes Löschen nicht mehr balanciert ist. Es gibt Erweiterungen vom binären Suchbaum, die darauf achten, dass der Baum immer ausbalanciert bleibt. Dies geht aber über den Inhalt dieses Kapitels hinaus.

Was du gelernt hast:
In diesem Abschnitt haben wir gelernt, Datenpunkte aus einem binären Suchbaum zu entfernen. Einen Knoten aus einem binären Suchbaum zu löschen, erfordert verschiedene Schritte, je nachdem, wie viele Kinder dieser Knoten hat.
Es ist am einfachsten, wenn der Knoten keine Kinder hat. In diesem Fall handelt es sich um einen Blattknoten, den du einfach entfernen kannst. Da er keine weiteren Verbindungen hat, verändert sich der Rest des Baums nicht.
Wenn der Knoten hingegen ein Kind hat, wird der Löschvorgang etwas komplexer. Hier ersetzt du den Knoten durch sein einziges Kind. Das bedeutet, dass das Kind an die Stelle des zu löschenden Knotens rückt und so die Struktur des Baums beibehalten wird.
Am anspruchsvollsten ist der Fall, wenn der Knoten zwei Kinder hat. In diesem Szenario musst du einen Ersatzknoten finden, der die Eigenschaften des binären Suchbaums weiterhin erfüllt. Eine Möglichkeit ist es, den kleinsten Wert im rechten Teilbaum des zu löschenden Knotens zu finden. Dieser Wert ist gleichzeitig der „Nachfolger“ des Knotens, da er grösser ist als alle Knoten im linken Teilbaum, aber kleiner als alle anderen Knoten im rechten Teilbaum. Du ersetzt den zu löschenden Knoten durch diesen Nachfolger und löschst dann den Nachfolger an seiner ursprünglichen Position. Eine alternative Methode ist, den größten Wert im linken Teilbaum zu nehmen, der als „Vorgänger“ fungiert.
Durch das wiederholte Löschen von Knoten kann es passieren, dass ein Baum, der zunächst balanciert war, sehr aus der Balance gerät. Dies führt über kurz oder lang zu Performance Einbussen, da Operationen am binären Suchbaum nicht mehr in optimaler Zeit ausgeführt werden können.

Glossary

0.1 Konzepte und Begriffe

Abstrakte Datenstruktur Eine Abstrakte Datenstruktur ist ein Konzept in der Informatik, das die strukturierte Organisation von Daten beschreibt, unabhängig von der konkreten Implementierung. Sie definiert, wie Daten organisiert und manipuliert werden können, ohne die internen Details preiszugeben. Abstrakte Datenstrukturen bieten eine standardisierte Möglichkeit, auf Daten zuzugreifen und Operationen darauf auszuführen. Diese Konzepte ermöglichen es Informatikern, Daten effizient zu verwalten und zu analysieren, ohne sich auf die spezifischen Details der Implementierung festlegen zu müssen. 6

Balancierter Binärbaum Ein balancierter Binärbaum ist ein Binärbaum bei dem sich

die Höhen der zwei Subbäume von jedem Knoten um maximal 1 unterscheiden. 31

Blatt Ein Knoten im gereichteten Baum, der auf keine weitern Knoten zeigt. 12

Catalan-Zahl Die n-te Catalan-Zahl $C(n)$ bemisst die Anzahl an binären Suchbäumen die es für n Knoten gibt. 31

Head Der Knoten in einer Linked List, der keinen eingehenden Pointer hat. 11

Höhe Die Höhe eines Baumes ist die maximale Tiefe eines Knotens in einem Binärbaum 25

In-Order Traversireung Eine Traversierung des Binärbaumes, bei der zunächst alle values im linken Subbaum, dann die value des Knotens und zuletzt alle values im rechten Subbaum durchlaufen werden. 16

Kind Die Knoten, auf die ein Knoten x zeigt, werden als Kinder von x bezeichnet. 12

Kompletter Binärbaum Ein Binärbaum ist komplett, wenn jedes Level ausser das unterste die maximale Anzahl Knoten hat, und das unterste Level von links nach rechts gefüllt wird. 31

Level Level i eines Binärbaums bezeichnet das Set aller Knoten mit Tiefe i. 25

Perfekter Binärbaum Ein binärer Suchbaum in dem jedes Level, die maximale Anzahl Knoten enthält, wird als perfekt bezeichnet. 31

Post-Order Traversierung Eine Traversierung des Binärbaumes, bei der zunächst alle values im linken Subbaum, dann alle values im rechten Subbaum, und zuletzt die value des Knotens durchlaufen wird. 16

Pre-Order Traversierung Eine Traversierung des Binärbaumes, bei der zunächst der values des Knotens, dann alle values im linken Subbaum und zuletzt alle values im rechten Subbaum durchlaufen werden. 16

Spezifikation Die Spezifikation einer abstrakten Datenstruktur definiert die strukturellen und operationellen Eigenschaften einer Datenstruktur, ohne dabei auf die konkrete Implementierung einzugehen. Sie beschreibt, welche Daten in der Datenstruktur gespeichert werden können und welche Operationen auf diesen Daten möglich sind. 6

Tiefe Die Tiefe eines Knotens x ist die Anzahl an Knoten, die auf dem Pfad von der Wurzel zum Knoten x liegen, inklusive der Wurzel und x. 25

Voller Binärbaum Ein Binärbaum ist voll, wenn jeder Knoten entweder genau zwei, oder null Kinder hat. 31

Wurzel Der Knoten im binären Suchbaum, der keinen eingehendern Pointer hat. 12

Lösungen zu den Übungen

Lösung 0.1 ***Wie lange braucht die Suche?*** Übung S. 4

Da die Klettstreifen nach dem Geburtstag sortiert sind, können wir die binäre Suche verwenden, um die Klettstreifen der SuS zu finden, welche die Klasse verlassen. Dies braucht maximal $\log(32) = 5$ Vergleiche. Um die Stellen in der Liste zu finden, an welcher die neuen SuS hinzugefügt werden sollten, verwenden wir eine modifizierte binäre Suche, in der wir immer zwei aufeinander folgende SuS betrachten. Unser Ziel ist es zwei Einträge zu finden, bei dem einer vor und der andere nach dem Geburtstag des neuen Schülers liegt. Da das Jahr zirkulär ist (nach dem letzten Kind ist identisch zu vor dem ersten Kind), haben wir wieder 32 mögliche Stellen an denen der/die neue Schüler/in hinzugefügt werden kann und brauchen somit wieder höchsten 5 Vergleiche.

Lösung 0.2 ***Pyhton Class SuS*** Übung S. 4

```
class student:
'''
    Erstellt ein neues student object
    Eingabe:
        - name: name des Schülers
        - month/day: Monat/ Tag des Geburtstags
'''
def __init__(self, name, month, day):
    self.name = name
    self.month = month
    self.day = day

'''
    Vergleicht 2 students anhand iheres Geburtstags/ Names
    Eingabe:
        - self
        - other - student mit dem verglichen wird
    Rückgabe: bool
        - 1 wenn Geburtstag von self im Jahr vor dem
        ↪ Geburtstag von other liegt, oder self und other
        ↪ den selben Geburtstag haben und selfs Name
        ↪ alphabetisch vor other liegt
        - 0 sonst
'''
def __lt__(self, other):
```

```
        if self.month < other.month:
            return True
        elif self.month == other.month:
            if self.day < other.day:
                return True
            elif self.day == other.day:
                return self.name < other.name
        return False
```

Lösung 0.3 ***Implementierung Geburtstagskalender*** Übung S. 5

Wir können den Geburtstagskalender mittels einer Linked List oder mittels der Python Array implementieren. Um bei der Linked List einen SuS hinzuzufügen, müssen wir mittels linearer Suche die korrekte Position finden. Das eigentliche Hinzufügen kann dann durch das Umsetzen von zwei Pointern vollzogen werden. Um einen SuS zu entfernen, müssen wir ebenfalls die lineare Suche verwenden, um sie/ihn zu finden. Das Entfernen selbst wird durch das Umsetzen eines einzelnen Pointers vollzogen.
In der Array-Implementierung können wir die binäre Suche verwenden, um die Position des/der Schüler/in zu finden, welche hinzugefügt/ entfernt wird. Das Hinzufügen/ Entfernen wird dann mittels der Array-Operationen insert() und pop() vollzogen, hierbei werden jeweils alle nachfolgenden SuS um eine Position verschoben.
Implementierung mittels Linked List:

```
class BirthdayCalendarNode:
'''
    Erstellt ein neues BirthdayCalendarNode object
    Eingabe:
        - student: Schüler/in zu welchem wir
        ↪  BirthdayCalendarNode object erstellen
'''
    def __init__(self, student):
        self.student = student
        self.next = None

class BirthdayCalendar:
'''
    Erstellt neuen BirthdayCalendar als Linked list
'''
    def __init__(self):
        self.head = None
```

```
    '''
        Fügt Schüler/in dem Kalender hinzu
        Eingabe:
            - new_student: Schüler/in, welche/r hinzugefügt wird
    '''
        def add_student(self, new_student):
            new_node = BirthdayCalendarNode(new_student)

            if not self.head or new_student < self.head.student:
                new_node.next = self.head
                self.head = new_node
                return

            current = self.head
            while current.next and current.next.student <
            ↪  new_student:
                current = current.next

            new_node.next = current.next
            current.next = new_node

    '''
        Entfernt Schüler/in aus dem Kalender
        Eingabe:
            - target_student: Schüler/in welche/r entfernt wird
        Rückgabe: bool
            - 1 : Schüler/in wurde gefunden und entfernt
            - 0 : sonst
    '''
        def remove_student(self, target_student):
            current = self.head
            previous = None

            while current:
                if current.student == target_student:
                    if previous:
                        previous.next = current.next
                    else:
                        self.head = current.next
                    return True
                previous = current
```

```
                current = current.next

            return False
```

Implementierung mittels Python Array:

```
    class BirthdayCalendar:
    '''
        Erstellt neuen BirthdayCalendar
    '''
    def __init__(self):
        self.students = []

    '''
        Fügt Schüler/in dem Kalender hinzu
        Eingabe:
            - new_student: Schüler/in, welche/r hinzugefügt wird
    '''
    def add_student(self, new_student):
        # Verwende binäre Suche, um die Position des neuen
        ↪  Schülers zu finden
        left, right = 0, len(self.students) - 1

        while left <= right:
            mid = (left + right) // 2
            if new_student < self.students[mid]:
                right = mid - 1
            else:
                left = mid + 1

        self.students.insert(left, new_student)


    '''
        Entfernt Schüler/in aus dem Kalender
        Parameter:
            - target_student: Schüler/in welche/r entfernt wird
        Rückgabe: bool
            - 1 : Schüler/in wurde gefunden und entfernt
            - 0 : sonst
    '''
    def remove_student(self, target_student):
```

```
        left, right = 0, len(self.students) - 1

        while left <= right:
            mid = (left + right) // 2
            if target_student < self.students[mid]:
                right = mid - 1
            elif target_student == self.students[mid]:
                del self.students[mid]
                return True
            else:
                left = mid + 1

        return False
```

Lösung 0.4 ***Stapel*** Übung S. 6

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        """
        Initialization: Eine leere Stack-Datenstruktur wird
        ↪ erstellt.
        """
        self.head = None
        self.count = 0

    def is_empty(self):
        """
        is_empty(): Überprüft, ob der Stack leer ist.
        Rückgabewert: True, wenn der Stack leer ist, andernfalls
        ↪ False.
        """
        return self.head is None

    def push(self, item):
        """
```

```
        push(item): Fügt ein Element oben auf den Stack hinzu.
        Parameter: item - Das einzufügende Element.
        """
        new_node = Node(item)
        new_node.next = self.head
        self.head = new_node
        self.count += 1

    def pop(self):
        """
        pop(): Entfernt das oberste Element vom Stack und gibt es
        ↪  zurück.
        Rückgabewert: Das entfernte Element.
        """
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        popped_node = self.head
        self.head = self.head.next
        self.count -= 1
        return popped_node.data

    def peek(self):
        """
        peek(): Gibt das oberste Element auf dem Stack zurück,
        ↪  ohne es zu entfernen.
        Rückgabewert: Das oberste Element.
        """
        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.head.data

    def size(self):
        """
        size(): Gibt die Anzahl der Elemente im Stack zurück.
        """
        return self.count
```

Lösung 0.5 ***Nutzerfreiheit von Abstrakten Datenstuckturen*** Übung S. 8

Siehe Lösung von Aufgabe 0.3.

Lösung 0.6 ***Definiere eine abstrakte Datenstruktur*** Übung S. 8

class meine_Datenstruktur

- __init__: Initialisiere Datenstruktur
- insert(data): Füge einen Datenwert *data* hinzu
- remove(data): Entferne Datenpunkt *data*
- find(data): Überprüfe, ob Datenpunkt *data* in der Datenstruktur vorhanden ist

Lösung 0.7 ***Wie funktioniert der Dictionary*** Übung S. 9

Ein Dictionary ist eine Datenstruktur in Python, die es ermöglicht, Daten in Form von Schlüssel-value-Paaren zu speichern und effizient darauf zuzugreifen. Wesentliche Merkmale von Dictionaries:

- Schlüssel-Wert-Paare: Jedes Element in einem Dictionary besteht aus einem Schlüssel (key) und einem dazugehörigen Wert (value). Schlüssel müssen unveränderlich (immutable) und eindeutig innerhalb des Dictionaries sein, während Werte beliebige Datentypen annehmen können, und durch die update Methode geändert werden können.
- Zugriff: Die get Methode erlaubt den Zugriff auf spezifische Schlüssel-value-Paaren ohne, dass die Datenstruktur sortiert sein muss. Die Methoden items, keys, und values erlauben es eine Übersicht der Schüller-value-Paare, oder auch nur der Schlüssel oder values zu erlangen.
- Dynamische Größe: Dictionaries können dynamisch wachsen und schrumpfen, je nachdem, wie viele Schlüssel-value-Paare hinzugefügt (update) oder entfernt (pop) werden.

Lösung 0.8 ***Anwendungen vom Dictionary*** Übung S. 10

Viele Antworten sind hier möglich.

Lösung 0.9 ***Zusammenhang Dictionary und Geburtstagskalender*** Übung S. 10

Im Geburtstagskalender besteht jedes Element aus einem SuS. Um den Dictionary zu verwenden, müssen wir jedes Element in ein < key, value > Paar verwandeln. Da es zwischen den SuS eine totale Ordnung gibt und wir den Kalender nach dieser Ordnung sortieren wollen, müssen wir für jeden SuS ein < key, value > Paar im

Dictionary erstellen, bei dem jeweils der SuS den Key darstellt. Als Value kann ein beliebiger Wert gewählt werden, z.B. None.

Lösung 0.10 ***Binärbäume Programieren*** Übung S. 12

```
import random

class Knoten:
'''
    Erstellt einen neuen Knoten
    Eingabe:
        - value: Wert des Knotens
'''
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class Bbaum:
    '''
        Initialisiert einen Binärbaum
    '''
    def __init__(self):
        self.root = None

    '''
        Fügt einen neuen Knoten in den Binärbaum hinzu, sofern der
        ↪ Wert noch nicht vorhanden ist
        Eingabe: - value: Wert des neuen Knotens
    '''
    def add(self, value):
        new_node = Knoten(value)
        if not self.root:
            self.root = new_node
        else:
            self._add_recursive(self.root, new_node)

    def _add_recursive(self, node, new_node):
        if random.choice([True, False]):
            if node.left is None:
                node.left = new_node
            else:
```

```
                self._add_recursive(node.left, new_node)
        else:
            if node.right is None:
                node.right = new_node
            else:
                self._add_recursive(node.right, new_node)
```

Lösung 0.11 ***Welche Binärbäume sind korrekt?*** Übung S. 13

1. Ja
2. Nein, da $6 < 7$, aber 6 ist im rechten Subbaum von 7.
3. Nein, da $3 < 9$ und 3 ist das rechte Kind von 9. Zusätzlich, tritt die value 3 zweimal im Baum auf.

Lösung 0.12 ***Andere Definition?*** Übung S. 15

Nein, die Definitionen sind nicht äquivalent. Der 2 Baum der letzten Aufgabe erfüllt die Definition von Tom, aber nicht die vorherige Definition.

Lösung 0.13 ***In-Order Traversierung Durchführung*** Übung S. 16

- 1, 3, 4, 5, 8, 11, 17
- 3, 7, 8, 11, 15, 17, 23, 39

Die In-Order Traversierung gibt die values der Knoten in aufsteigender Reihenfolge zurück.

Lösung 0.14 *In-Order Code* Übung S. 17

```
'''
    Druckt die Werte des Binärbaumes mit Wurzel node in In-Order
    ↪ Reihenfolge
    Eingabe:
        - node: derzeitiger Knoten in der Traversierung
'''
def Inorder(node):
    if node is None:
        return
    Inorder(node.left)
    print(node.value, end=' ')
    Inorder(node.right)
```

Lösung 0.15 *Pre-Order Traversierung Durchführung* Übung S. 18

- 5, 3, 1, 4, 11, 8, 17
- 7, 3, 11, 8, 23, 17, 15, 39

Lösung 0.16 *Pre-Order Code* Übung S. 19

```
'''
    Druckt die Werte des Binärbaumes mit Wurzel node in Pre-Order
    ↪ Reihenfolge
    Eingabe:
        - node: derzeitiger Knoten in der Traversierung
'''
def Preorder(node):
    if node is None:
        return
    print(node.value, end=' ')
    Preorder(node.left)
    Preorder(node.right)
```

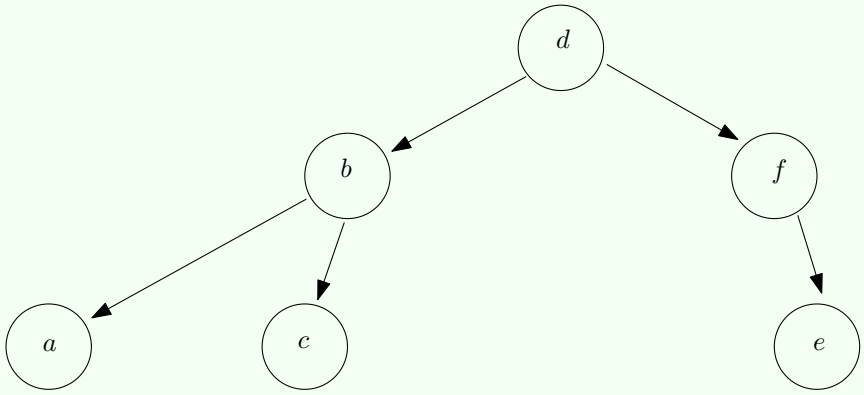
Lösung 0.17 *Post-Order Traversierung Durchführung* Übung S. 20

- 1, 4, 3, 8, 17, 11, 5
- 3, 8, 15, 17, 39, 23, 11, 7

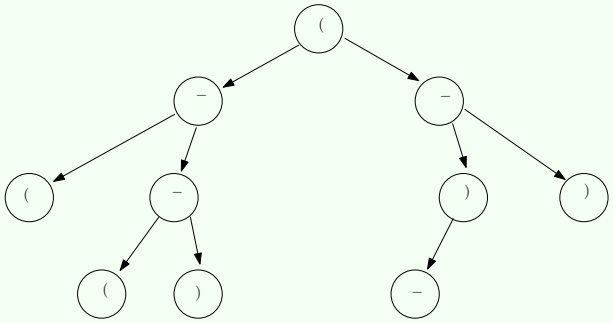
Lösung 0.18 *Post-Order Code* Übung S. 21

```
'''
    Druckt die Werte des Binärbaumes mit Wurzel node in Post-Order
    ↪ Reihenfolge
    Eingabe:
        - node: derzeitiger Knoten in der Traversierung
'''
def Postorder(node):
    if node is None:
        return
    Postorder(node.left)
    Postorder(node.right)
    print(node.value, end=' ')
```

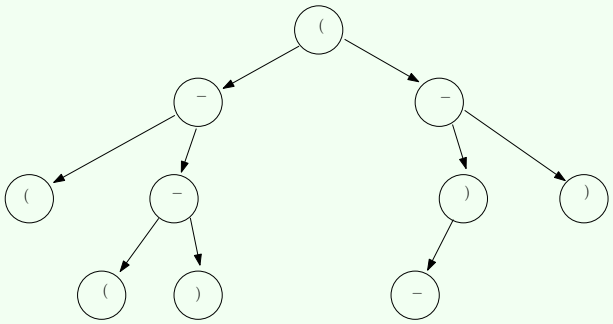
Lösung 0.19 *Konstruiere nach Traversierung* Übung S. 21

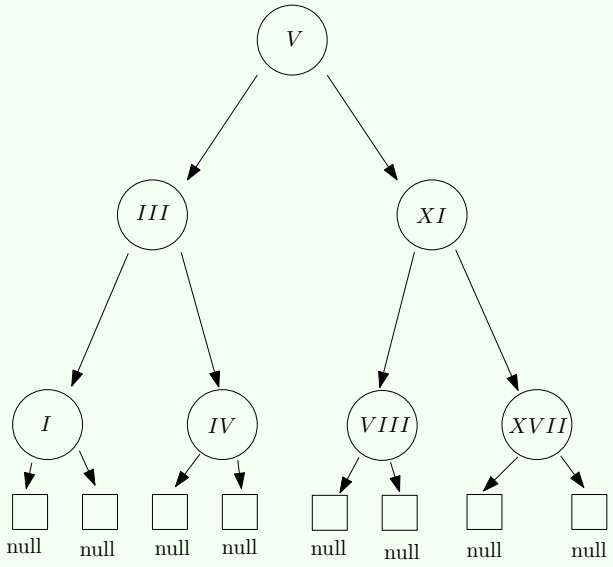
- 

  Hinweis, da wir nicht explizit nach einem binären Suchbaum gefragt haben, ist hier mehr als eine Antwort möglich!

- 

  Ja, wir können jeden beliebigen Binärbaum nehmen und ihn mit der Beschriftung labeln.

- 

- 

Lösung 0.20 ***Find Methode entwickeln*** Übung S. 22

a)
- $x = B.root.value$: in diesem Falle haben wir die value x in B gefunden und können 1 zurückgeben
- $x < B.root.value$: in diesem Falle suchen wir im linken Subbaum von x weiter.

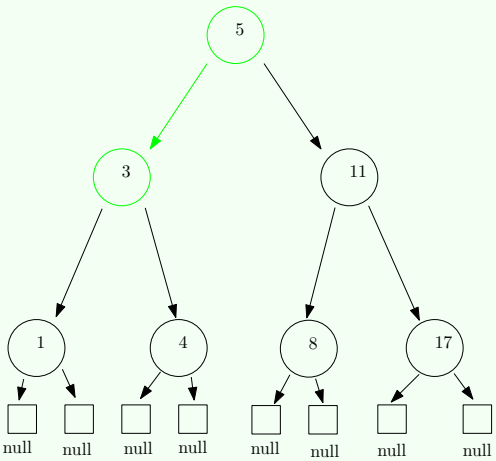
- $x > B.root.value$: in diesem Falle suchen wir im rechten Subbaum von x weiter.

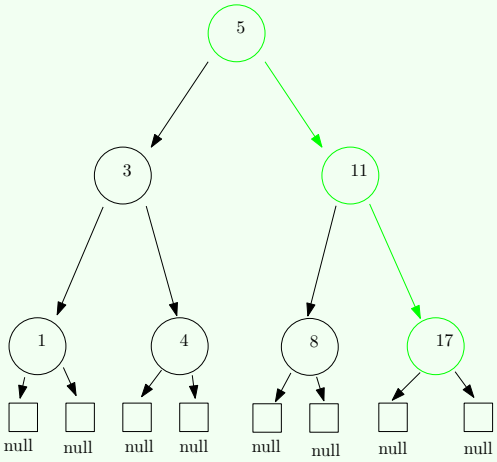
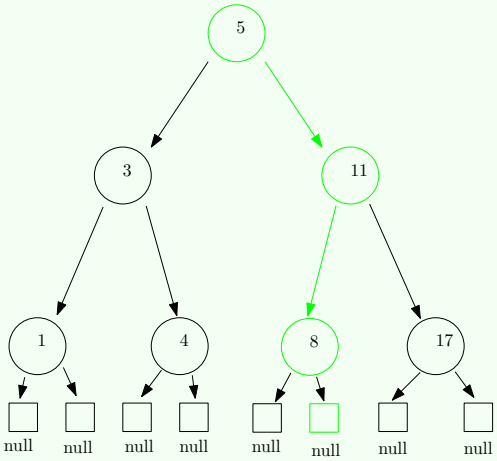
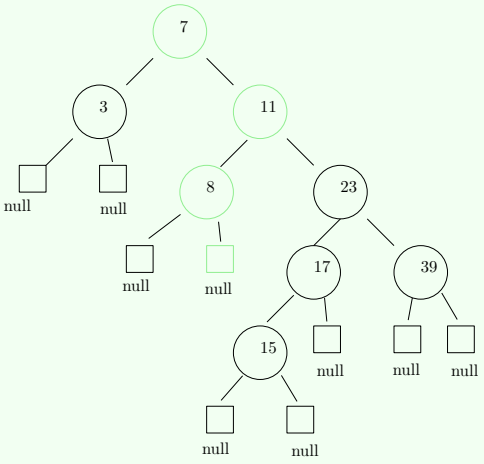
b) $B.root.value = null$: wenn der Baum leer ist, dann ist x kein Element im Baum. In diesem Falle geben wir 0 zurück.

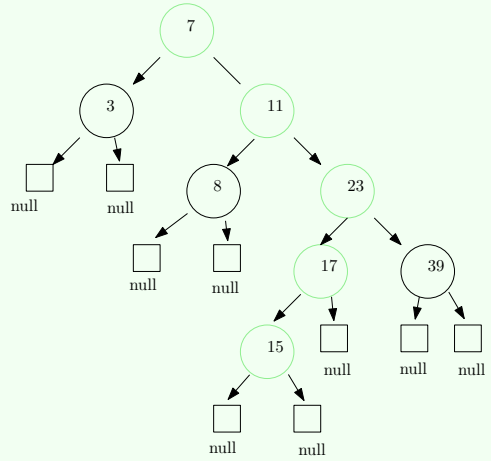
c)
```
'''
    Bestimmt ob Knoten mit Wert value im binären Suchbaum
    ↪ vorhanden ist
    Eingabe:
        - value: Wert des zu Suchenden Knotens
    Rückgabe: bool
        - 1: wenn value vorhanden
        - 0: sonst
'''
def find(self, value):
    return self._find_recursive(self.root, value)

def _find_recursive(self, node, value):
    if node is None:
        return 0
    elif node.value == value:
        return 1
    elif value < node.value:
        return self._search_recursive(node.left, value)
    return self._search_recursive(node.right, value)
```

Lösung 0.21 ***Pfade des Findes*** Übung S. 23

- 

- 
5
3
11
1
4
8
17
null
null
null
null
null
null
null
null
- 
5
3
11
1
4
8
17
null
null
null
null
null
null
null
null
- 
7
3
11
null
null
8
23
null
null
17
39
15
null
null
null
null
null



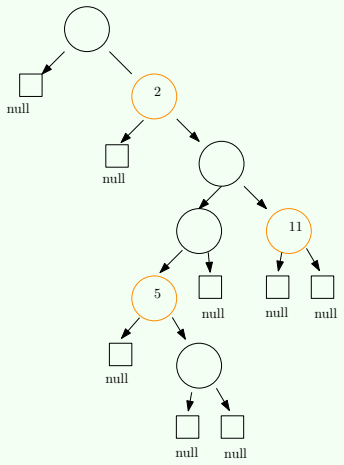
Lösung 0.22 *Schnelles finden*

Übung S. 25

Im ersten Baum müssen maximal 7 Vergleiche gemacht werden, im zweiten Baum maximal 4 Vergleiche. Generell ist die maximale Anzahl an Vergleichen gleich der maximalen Länge eines Pfades von der Wurzel zu einem Null-Knoten.

Lösung 0.23 *Höhen messen*

Übung S. 25



Die Höhe des Baumes ist 6.

Lösung 0.24 *Höhe rekrusiv bestimmen*

Übung S. 26

$$height(B) = \begin{cases} 0 & \text{if not } B.root \\ 1 + \max\{height(Subtree(B.root.right)), \\ \qquad height(Subtree(B.root.left))\} \end{cases}$$

Lösung 0.25 *Optimale Strukturen*

Übung S. 26

Um die Anzahl der Vergleiche zu minimieren, müssen wir die Höhe des Baumes minimieren. Dies erreichen wir, in dem wir die Höhe des Baumes erst dann erhöhen, wenn es keine Möglichkeit mehr gibt, einen Knoten im Baum hinzuzufügen, ohne die Höhe zu erhöhen.

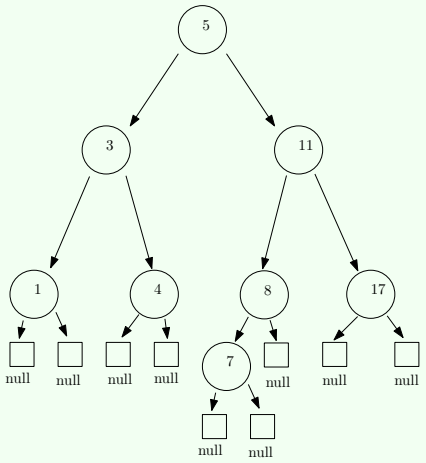
Lösung 0.26 *Grosse und kleine Bäume*

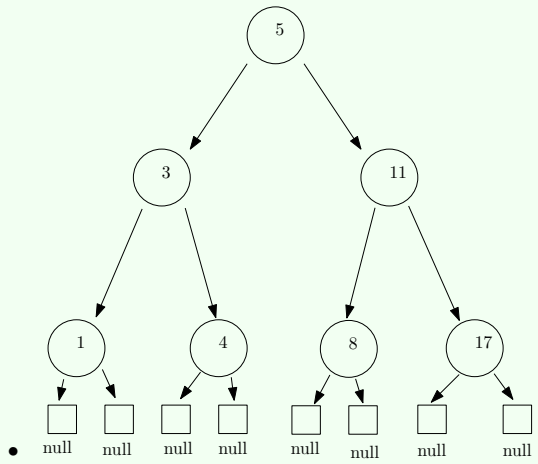
Übung S. 26

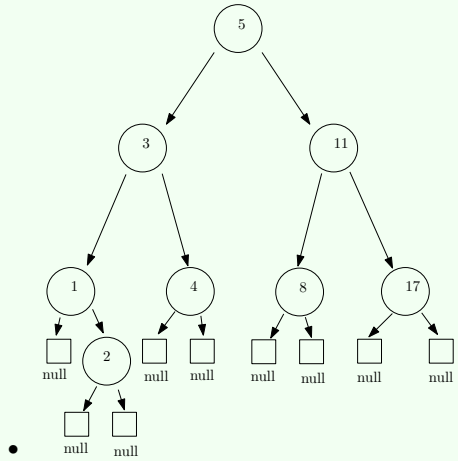
Ein Binärbaum mit 23 Knoten hat mindestens eine Höhe von $h_{\min} = 5$ und maximal eine Höhe von $h_{\max} = 23$. Generell kann ein Baum mit Höhe h bis zu $2^h - 1$ Knoten enthalten. Daher hat ein Baum mit n Knoten mindestens Höhe $2^h - 1 \geq n \Leftrightarrow \log_2(n+1) \geq h$. Die maximal Höhe wird erreicht, wenn der Baum genau ein Knoten auf jeder Höhe hat. Somit ist die maximal Höhe eines Baumes mit n Knoten n.

Lösung 0.27 *Add Methode anwenden*

Übung S. 27







Lösung 0.28 ***Einzigartigegs Einfügen*** Übung S. 28

Wenn der binäre Suchbaum leer ist, so kann x nur als Wurzel hinzugefügt werden, da wir ansonsten keinen Baum haben. Wenn bereits Knoten im Baum vorhanden sind, so muss x als ein Kind von einem Blatt im Baum hinzugefügt werden. Betrachten wir alle Kinder eines binären Suchbaums von links nach rechts, so ist die Reihenfolge strikt aufsteigend. Es gibt genau zwei Blätter B_1 und B_2 in dieser Reihenfolge, für die gilt $B_1 < x < B_2$. Dies bedeutet, dass x entweder als rechtes Kind von B_1 oder linkes Kind von B_2 hinzugefügt werden muss. Da B_1 und B_2 beide Blätter sind, muss es einen gemeinsamen Vorfahren (Alle Knoten, welche auf dem Pfad von einem Knoten y zur Wurzel des Baumes liegen, werden alle Vorfahre von y bezeichnet.) B_3 geben, wo gilt $B_1 < B_3 < B_2$. Da entweder $x < B_3$ oder $B_3 < x$ steht fest, auf welcher Seite von B_3 x im Baum angehängt werden muss. Dies entscheidet also, ob x ein rechtes Kind von B_1 oder ein linkes Kind von B_3 ist.

Lösung 0.29 ***Add Methode entwickeln*** Übung S. 28

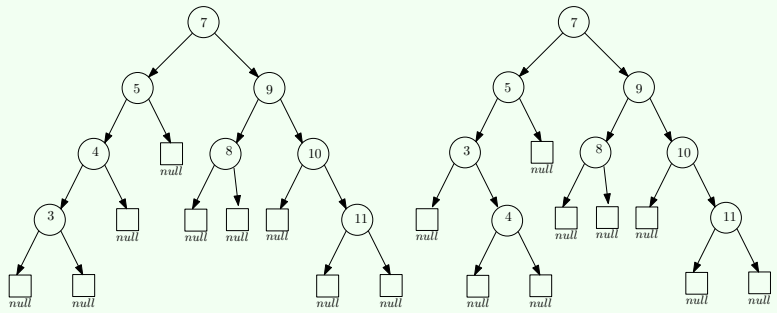
- The *find(x)*-Methode hat zwei Basecases. Der erste ist, dass wir einen Knoten mit value x finden. Der andere, dass wir einen null-Knoten finden. Finden wir Knoten mit value x in der *Add(x)*-Methode, so sind wir fertig und brauchen den binären Suchbaum nicht weiter zu verändern. Erreichen wir einen null-Knoten, so wollen wir an genau dieser Stelle einen Knoten mit value x hinzufügen.

- ```
  '''
      Fügt Knoten mit Wert value im binären Suchbaum hinzu
      Eingabe:
          - value: Wert des neuen Knotens
  '''
  def add(self, value):
      if not self.root:
          self.root = Knoten(value)
      self._add_recursive(self.root, value)

  def _add_recursive(self, node, value):
      if value < node.value:
          if node.left:
              self._add_recursive(node.left, value)
          else:
              node.left = Knoten(value)
      elif value > node.value:
          if node.right:
              self._add_recursive(node.right, value)
          else:
              node.right = Knoten(value)
  ```

Lösung 0.30 ***Values hinzufügen*** Übung S. 28

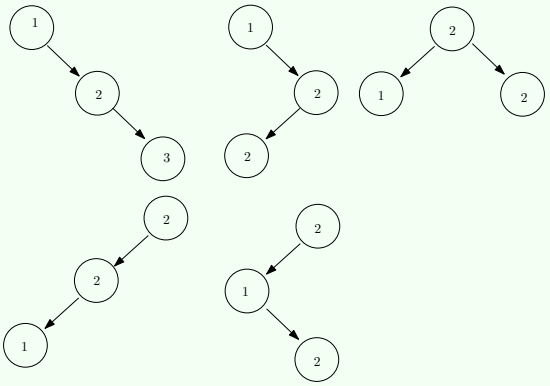
Je nachdem ob wir die value 3 oder 4 zuerst hinzufügen gibt es zwei verschiedene Lösungen.
  ```



Lösung 0.31 *3-Knoten Bäume*

Übung S. 29

Es gibt 5 Bäume für die values 1, 2, 3.



Die Anzahl an binären Suchbäumen ist für jedes Tripel gleich, da es immer genau eine totale Ordnung der values gibt.

Lösung 0.32 *Wie ist der Baum entstanden?*

Übung S. 29

Es gibt drei mögliche Reihenfolgen.

- 7,5,9,10
- 7,9,5,10
- 7,9,10,5

Wir wissen, dass die 7 als erstes eingefügt werden muss. Zusätzlich muss die 9 vor der 10 eingefügt werden.

Lösung 0.33 ***Bäume zählen*** Übung S. 30

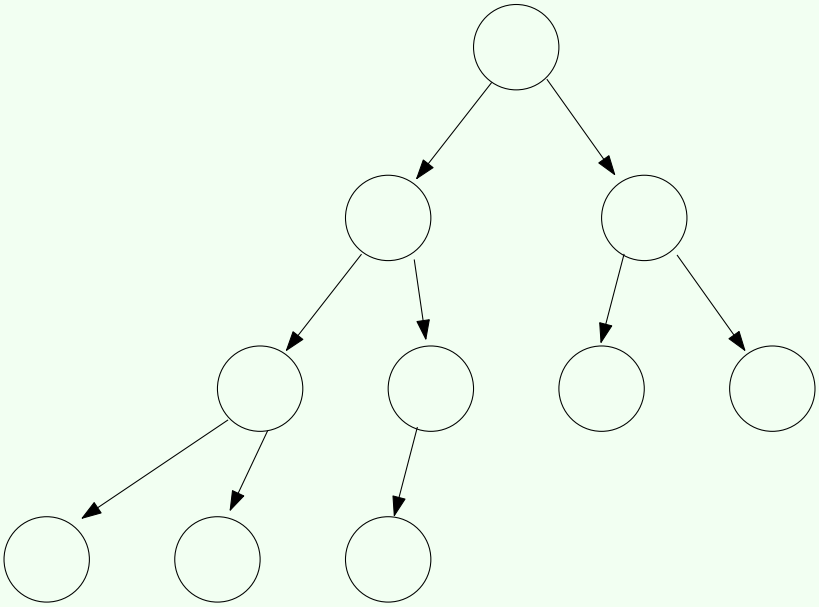
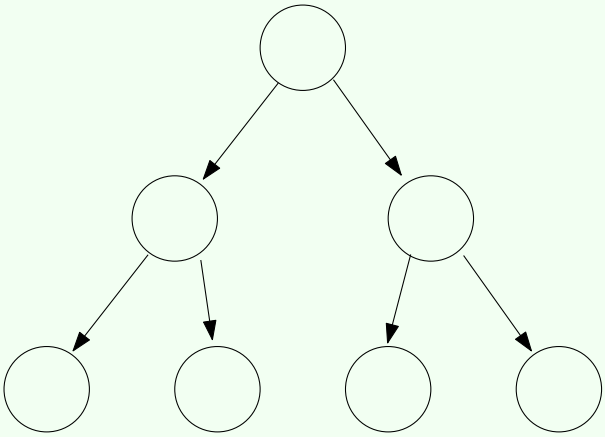
- Es gibt n Möglichkeiten, die Wurzel auszuwählen.
- Nachdem wir eine value i, wobei $1 \leq i \leq n$, für die Wurzel ausgewählt haben, wissen wir, dass die $i-1$ kleineren values im linken und die $n-i-1$ grösseren values im rechten Subbaum der Wurzel sein müssen. Jede der values, die kleiner sind als i kann Wurzel des linken Subbaumes sein, und jede der values die grösser sind als i, kann Wurzel des rechten Subbaumes sein. Somit gibt es $\sum_{i=1}^{n}(i-1)(n-i-1)+2(n-1)$ Möglichkeiten, die Wurzel und ihre 2 Kinder auszuwählen. Der letzte Summand ist für die Fälle $i=1$ und $i=n$, da in diesem Falle einer der 2 Multiplikatoren 0 ergibt.
- Sei $C(n)$ die Anzahl der binären Suchbäume für die values $1, 2, \ldots n$. Nachdem wir eine value i für die Wurzel ausgewählt haben, wissen wir für jeden andere value, ob es im rechten oder im linken Subbaum der Wurzel ist. Somit sind der rechte und der linke Subbaum der Wurzel unabhängig voneinander. Zusätzlich ist der linke Subbaum ein binärer Suchbaum der values $1, 2, \ldots, i-1$ und der rechte Subbaum ein binärer Suchbaum der values $i+1, i+2, \ldots n$. Daher gibt es $C(i-1)$ Möglichkeiten den linken und $C(n-i-i)$ Möglichkeiten den rechten Subbaum der Wurzel zu bauen. Somit erhalten wir $C(n) = \sum_{i=1}^{n} C(i-1)C(n-i-1)$. Um die Rekursion zu lösen, müssen wir noch die Basecases bestimmen. Es gibt jeweils genau einen binären Suchbaum, der keine bzw. genau einen value enthält. Somit ist $C(0) = C(1) = 1$. Wenn wir dies in die obige Rekursion einfügen, erhalten wir
$$C(n) = \frac{(2n)!}{(n+1)!n!}.$$

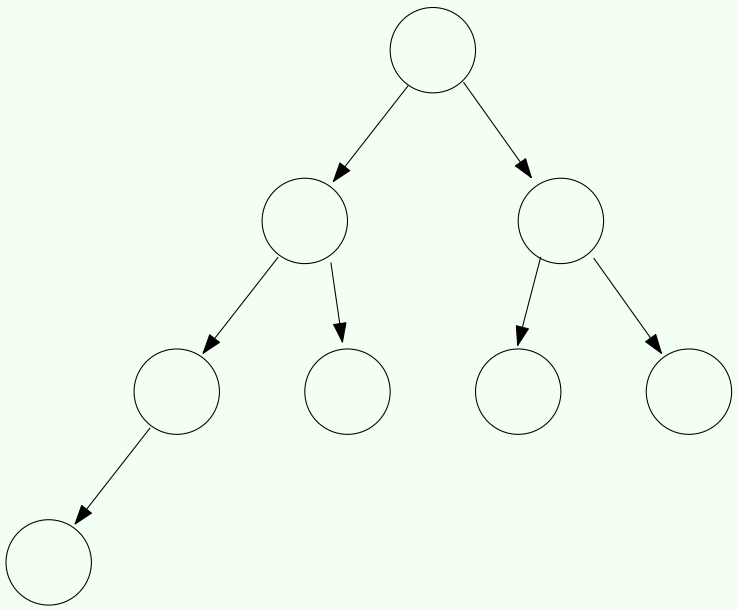
Lösung 0.34 ***Performance Einfügen*** Übung S. 31

Die Worstcase Anzahl an Vergleichen ist immer gleich der Höhe des Baumes, da wir im Worstcase den Pfad von der Wurzel zum am weitesten entfernten Kind ablaufen müssen. Aus Aufgabe 0.26 wissen wir, dass ein Baum mit n Knoten mindestens Höhe $\log_2(n+1)$ und maximal Höhe n hat. Somit ist die Zahl der Vergleiche im Worstcase mindestens $\log_2(n+1)$.

Lösung 0.35 ***Bäume zeichnen*** Übung S. 31

- Es ist nicht möglich, einen vollen Binärbaum mit 10 Knoten zu zeichnen, da ein voller Binärbaum entweder leer ist, oder eine ungerade Anzahl an Konten enthält. Dies folgt daraus, dass jeder Knoten entweder 0 oder 2 Kinder hat, die Wurzel aber keinen Geschwisterknoten hat.

- 
- 
- Hier ist mehr als eine Antwort möglich. Eine Möglichkeit wäre die folgende:



Lösung 0.36 ***Strukturen erkennen*** Übung S. 32

Die Binärbäume sind von oben nach unten

- voll, balanciert
- voll, balanciert und komplett
- nichts
- komplett

Lösung 0.37 ***Knotenzahlen*** Übung S. 35

- Die Aussage ist falsch. Ein voller Binärbaum hat entweder Null Knoten, oder eine ungerade Anzahl an Knoten.
- wahr
- Die Aussage ist falsch. Perfekte Binärbäume gibt es nur für $n = 2^k - 1$.
- Die Aussage ist falsch. Man kann für jedes n einen balancierten Binärbaum bauen.

Lösung 0.38 ***Minimale Höhen*** Übung S. 35

Ein perfekter Binärbaum hat Höhe $h = \log_2(n+1)$. Da ein perfekter Binärbaum immer auch voll, komplett, balanciert ist, gilt dieselbe minimale Höhe auch für diese Bäume.

Lösung 0.39 ***Maximale Höhen*** Übung S. 35

Ein perfekter Binärbaum mit 15 Knoten hat immer Höhe 4. Es gibt für jedes n immer höchstens einen perfekten Binärbaum und dieser hat immer Höhe $h = \log_2(n+1)$.
Ein kompletter Binärbaum mit 15 Knoten hat ebenfalls Höhe 4. Es gibt für jedes n einen kompletten Binärbaum, und dieser hat immer Höhe $h = \lceil \log_2(n+1) \rceil$. Dies liegt daran, dass alle Level bis auf das unterste die maximale Anzahl Knoten haben müssen.
Die maximale Höhe eines vollen Binärbaums mit 15 Knoten ist 8. Eine Möglichkeit, um einen vollen Binärbaum mit maximaler Höher für beliebiges ungerades n zu bauen, ist, mit der Wurzel zu starten, auf jedem Level immer dem am weitesten linken Knoten genau zwei Kinder zu geben und alle anderen Knoten auf dem Level keine Kinder zu geben. In diesem Falle hat der entstandene volle Binärbaum Höhe $\frac{n-1}{2}$.
Ein balancierter Binärbaum mit maximaler Höhe zu erhalten, haben wir immer genau einen Knoten auf der untersten Stufe. Sei B_h ein balancierter Suchbaum der Höhe h mit der minimalen Anzahl an Knoten. Bevor wir die Höhe erhöhen können, müssen alle anderen Subbäume von B_h ihre Höhe um 1 erhöhen. Das heisst, alle Knoten mit Grad 1 und Grad 0 bekommen jeweils ein weiteres Kind. Somit ist $|B_{h+1}| = |B_h| + (|B_h| - |B_{h-1}|) + (|B_{h-1}| - |B_{h-2}|)$, wobei der zweite Summand die Knoten mit Grad 0 und der dritte Summand die Knoten mit Grad 1 beschreibt. Wir wissen zusätzlich, dass $B_0 = 0$, $B_1 = 1$ und $B_2 = 2$ ist. Wenn wir diese Gleichungen auflösen, so erhalten wir, dass $B_h = \frac{1}{10}(-10 + (5 - 3\sqrt{(5)})(\frac{1}{2}(1 - \sqrt{(5)}))^h + (\frac{1}{2}(1 + \sqrt{(5)}))^h(5 + 3\sqrt{(5)}))$ Knoten enthält. Dies impliziert, dass ein balancierter Binärbaum mit 15 Knoten maximal eine Höhe von 5 hat.

Lösung 0.40 ***binäre Suchbäume bauen*** Übung S. 36

- $h = \lceil \log_2(n+1) \rceil$
- Sei $j = \lfloor \log_2(n+1) \rfloor$. Dann sind auf dem untersen Level $k = n - 2^j - 1$ Knoten.
- Der linke Subbaum der Wurzel hat Höhe $\lceil \log_2(n+1) \rceil - 1$. Wenn $n - 2^j - 1 > 2^{h-1}$ (d.h. wenn das unterste Level zu mehr als der Hälfte gefüllt ist) dann hat der rechte Subbaum ebenfalls Höhe $\lceil \log_2(n+1) \rceil - 1$, andernfalls hat er

Höhe $\lceil \log_2(n+1) \rceil - 2$.

- Wir unterscheiden 2 Fälle: Fall 1 ist, dass $n - 2^{h-1} - 1 > 2^{h-2}$ und Fall 2 ist, dass $n - 2^{h-1} - 1 \leq 2^{h-2}$. Im ersten Fall ist das unterste Level zu mehr als der Hälfte gefüllt. In diesem Fall ist der rechte Subbaum der Wurzel ein perfekter Binärbaum der Höhe $h-1$. Das heisst, der rechte Subbaum enthält genau $2^{h-1}-1$ Knoten. Da diese alle einen kleinere value haben als die Wurzel, hat die Wurzel die value 2^{h-1}. Im zweiten Falle ist das unterste Level zu weniger als die Hälfte gefüllt, und der rechte Subbaum der Wurzel ist ein perfekter binärer Suchbaum der Höhe $h-2$. Dieser enthält somit $2^{h-1}-1$ Knoten, deren values alle grösser sein müssen als die Wurzel. Somit hat in diesem Falle die Wurzel die value $n - 2^{h-1} + 1$.

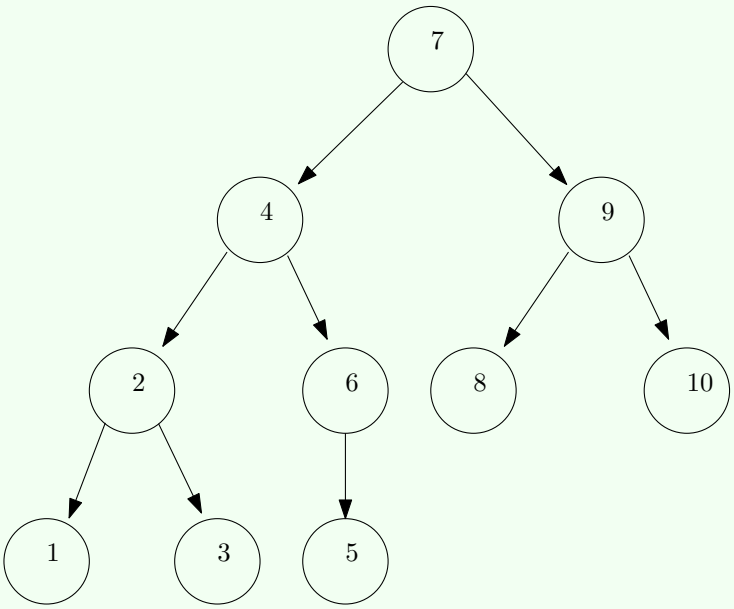
- Wir können rekursiv vorgehen, nachdem wir die Wurzel bestimmt haben, wissen wir, welche values im rechten und im linken Subbaum der Wurzel sein müssen. Diese formen wieder einen kompletten binären Suchbaum, wir können also das rechte und das linke Kind der Wurzel auf dieselbe Art und Weise bestimmen, wie wir die Wurzel des Baumes bestimmt haben. Dies wiederholen wir, bis alle Knoten ihren Platz gefunden haben.

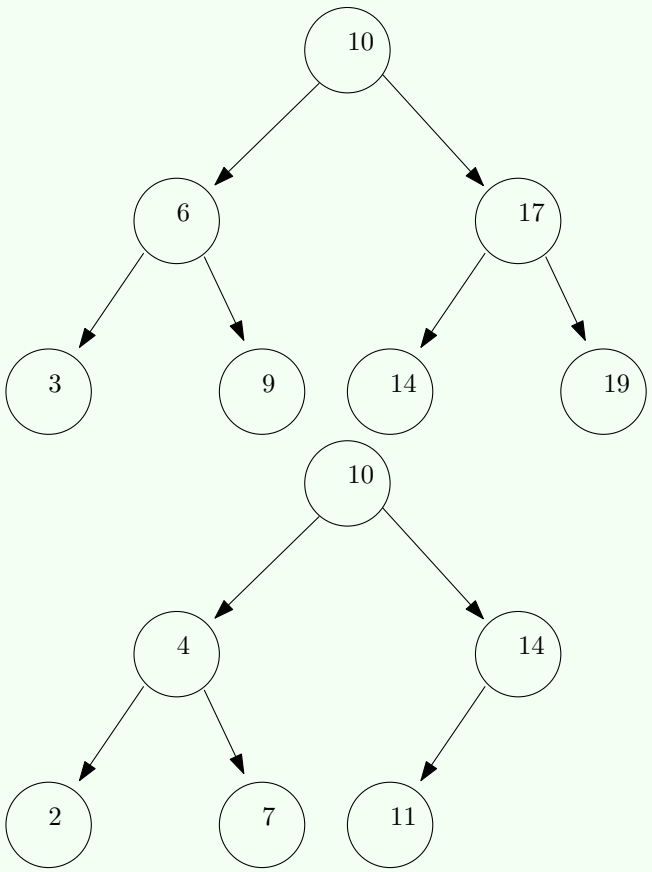
```
class Knoten:
    '''
        Initislisiert einen neuen Knoten
        Eingabe:
            - value: Wert des neuen Knotens
    '''
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    '''
        Baut einen balancieten binären Suchbaum bestehend aus
        ↪  n Knoten
        Eingabe: - n: Anzahl der Knoten im balancierten
        ↪  binären Suchbaum
    '''
    def buildBalancedBST(n):
        h = math.ceil(math.log2(n + 1))  # Calculate height h
        if n == 0:
            return None
        if h == 1:
            return Knoten(1)
```

```
        if n - (1 << (h - 1)) - 1 > (1 << (h - 2)):
            root_value = 1 << (h - 1)
            root = Knoten(root_value)
            root.left = buildBalancedBST((1 << (h - 1)) - 1)
            root.right = buildBalancedBST(n - (1 << (h - 1)) -
            ↪  1)
        else:
            root_value = n - (1 << (h - 1)) + 1
            root = Knoten(root_value)
            root.left = buildBalancedBST((1 << (h - 1)) - 1 +
            ↪  (n - (1 << (h - 1)) - 1 - (1 << (h - 2))))
            root.right = buildBalancedBST((1 << (h - 1)) - 1)
            height h-2
        return root
```

Lösung 0.41 ***Baue komplette binäre Suchbäume*** Übung S. 36





Lösung 0.42 ***Bäume zerlegen*** Übung S. 37

- Sofern die Wurzel ein linkes und ein rechtes Kind hat, ist das Resultat, wenn wir die Wurzel vom binären Suchbaum entfernen, dass zwei binäre Suchbäume übrig bleiben. Diese entsprechen dem rechten und linken Subbaum der Wurzel. Wenn die Wurzel nur ein Kind hatte, so bleibt ein binärer Suchbaum übrig. Und wenn die Wurzel keine Kinder hatte, so erhalten wir einen leeren Suchbaum.
- Wenn wir ein Blatt vom binären Suchbaum entfernen, so ist das Resultat ein binärer Suchbaum.
- Sei x der Knoten, den wir entfernen. Sofern x zwei Kinder hat, erhalten wir drei binäre Suchbäume, die jeweils dem rechten Subbaum von x, dem linken Subbaum von x und dem ursprünglichen binären Suchbaum ohne den Subbaum mit Wurzel x entsprechen. Wenn x nur ein Kind hat, so ist jeweils der rechte oder der linke Subbaum von x leer.

Lösung 0.43 ***Löschen einfach*** Übung S. 37

Dies folgt aus der Definition des binären Suchbaums: Sei x der Knoten, den wir entfernen, und y sein Kind. Wir wissen, dass der Subbaum mit Wurzel y ein binärer Suchbaum ist. Sei z der Knoten, welcher auf x zeigt, und ohne Beschränkung der Allgemeinheit, sei x das linke Kind von z. Dann sind die values aller Knoten im Subbaum mit Wurzel x kleiner als die value von z. Wenn wir also Knoten y mit seinem Subbaum an die Stelle von x setzen und ansonsten nichts an der Struktur des binären Suchbaumes ändern, haben wir weiterhin einen binären Suchbaum.

Lösung 0.44 ***Einfaches Löschen Implementieren*** Übung S. 38

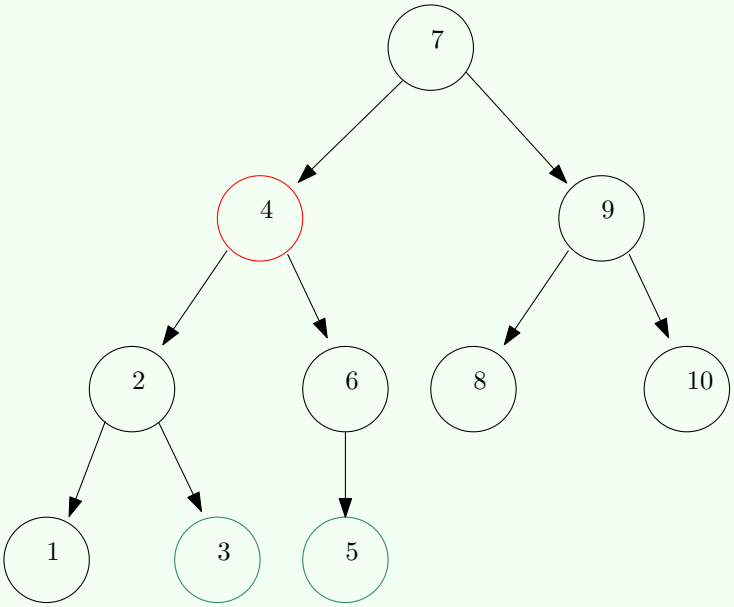
```
'''
    Löscht Knoten mit Wert x sofern dieser höchstens ein Kind hat
    Eingabe
        - x: Wert des zu löschenden Knotens
'''
def delSimple(root, x):
    if root is None:
        return root

    if root.value > x:
        root.left = deleteNode(root.left, x)
        return root
    elif root.value < x:
        root.right = deleteNode(root.right, x)
        return root

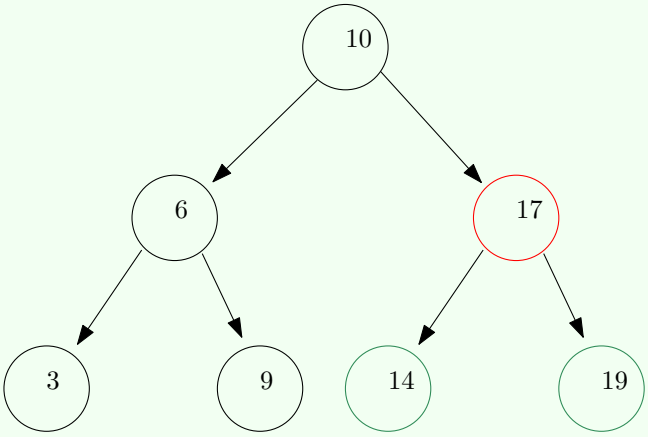
    if root.left is None and root.right is None:
        return Null
    if root.left is None:
        temp = root.right
        del root
        return temp
    elif root.right is None:
        temp = root.left
        del root
        return temp
```

Lösung 0.45 ***Ersatzraum*** Übung S. 39

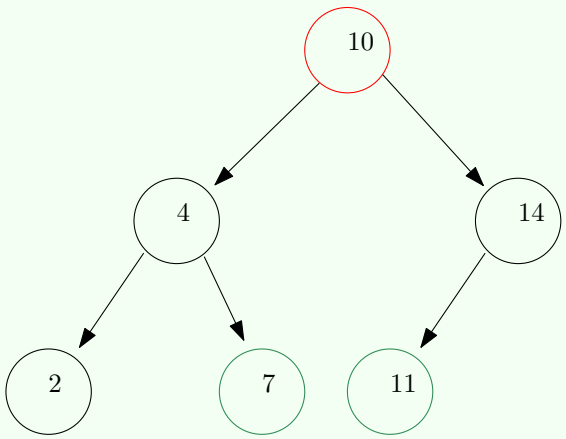
Damit wir weiterhin einen binären Suchbaum haben, muss die value des neuen Knotens zwischen der minimalen value im rechten Subbaum und der maximalen value im linken Subbaum des markierten Knotens liegen.
Für den erten abgebildeten Baum ist die Range demnach $[3, 5]$.



Für den zweiten abgebildeten Baum ist die Range $[14, 19]$.



Für den dritten abgebildeten Baum ist die Range $[7, 11]$.



Lösung 0.46 ***Vorgänger und Nachfolger*** Übung S. 40

Damit der resultierende Binärbaum ein binärer Suchbaum ist, muss der Knoten, der an die Postion von x kommt, mindestens so gross sein, wie die grösste value im linken Subbaum von x, und höchstens so gross sein wie die kleinste value im rechten Subbaum von x. Diese Eigenschaft wird jeweils nur vom Knoten mit der höchsten value im linken Subbaum von x und dem Knoten mit der kleinsten value im rechten Subbaum von x erfüllt.
Betrachten wir zunächst den Knoten y mit der höchsten value im linken Subbaum von x. Wir behaupten, dass y kein rechtes Kind haben kann. Ansonsten müsste das rechte Kind von y eine grössere value haben als y, was der Annahme, dass y die grösste value im linken Subbaum von x ist, widerspricht. Ohne Beschränkung der Allgemeinheit kann man auf gleiche Art zeigen, dass der Knoten mit der kleinsten value im rechten Subbaum kein linkes Kind hat.

Lösung 0.47 ***Löschen Implementieren*** Übung S. 40

```
'''
    Findet den Vorgänger vom Knoten node in In-Order Reihenfolge
    Eingabe:
        - node: Knoten, dessen Vorgänger wir finden
'''
def maxLeft(node):
    curr = node.right
    while curr.right:
        curr = curr.right
    return curr

'''
```

```
    Findet den Nachfolger vom Knoten node in In-Order Reihenfolge
    Eingabe:
        - node: Knoten, dessen Nachfolger wir finden
'''
def minRight(node):
    curr = node.left
    while curr.left:
        curr = curr.left
    return curr
```

Lösung 0.48 ***Verhältnisse erkennen*** Übung S. 40

Betrachten wir die totale Ordnung aller values, so sind die zwei Knoten, die an die Postion von x treten können, die value, der vor x in der totalen Ordnung steht, und die value, der nach x in der totalen Ordnung steht. Da die In-Order Traversierung die totale Ordnung wieder gibt, stehen diese values vor und nach x in der In-Order Traversierung.

Lösung 0.49 ***Implementiere delete(x)*** Übung S. 41

```
Lösung in der x durch nächst grösseren Knoten ersetzt wird

'''
    Löscht Knoten mit Wert x und ersetzt seine Position mit dem
    ↪ nächst grösseren Wert
    Eingabe:
        - root: Wurzel des binären Suchbaums
        - x: Wert des zu Löschenden Knotens
'''
def deleteSuc(root, x):
        if root is None:
                return root

        if root.value > x:
                root.left = deleteSuc(root.left, x)
                return root
        elif root.value < x:
                root.right = deleteSuc(root.right, x)
                return root

        if root.left is None:
```

```
                temp = root.right
                del root
                return temp
        elif root.right is None:
                temp = root.left
                del root
                return temp

        else:
                succParent = root
                succ = root.right
                while succ.left:
                        succParent = succ
                        succ = succ.left

                if succParent != root:
                        succParent.left = succ.right
                else:
                        succParent.right = succ.right
                root.value = succ.value
                del succ
                return root

# Lösung in der x durch nächst kleinere Knoten ersetzt wird

'''
    Löscht Knoten mit Wert x und ersetzt seine Position mit dem
    ↪ nächst kleineren Wert
    Eingabe:
        - root: Wurzel des binären Suchbaums
        - x: Wert des zu Löschenden Knotens
'''
def deletePre(root, x):
        if root is None:
                return root

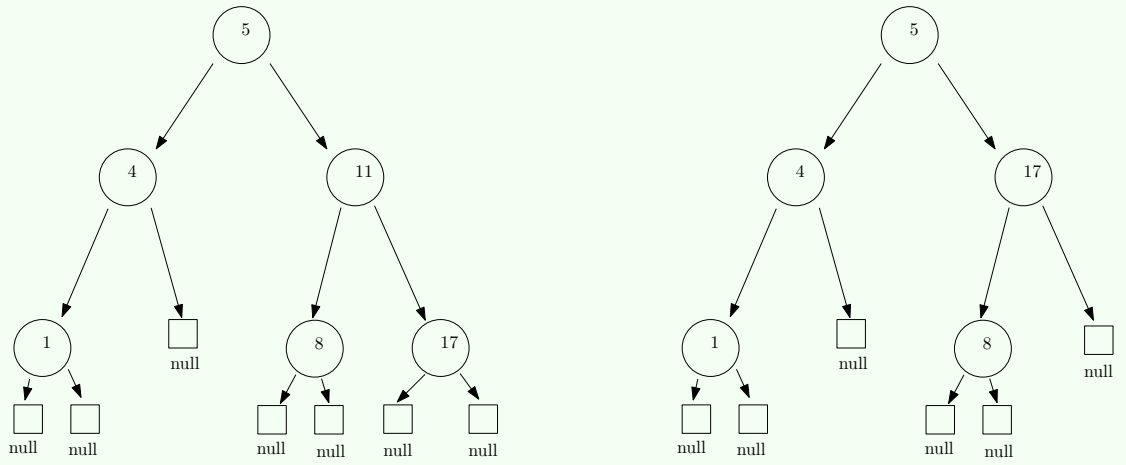
        if root.value > x:
                root.left = deletePre(root.left, x)
                return root
        elif root.value < x:
                root.right = deletePre(root.right, x)
                return root
```

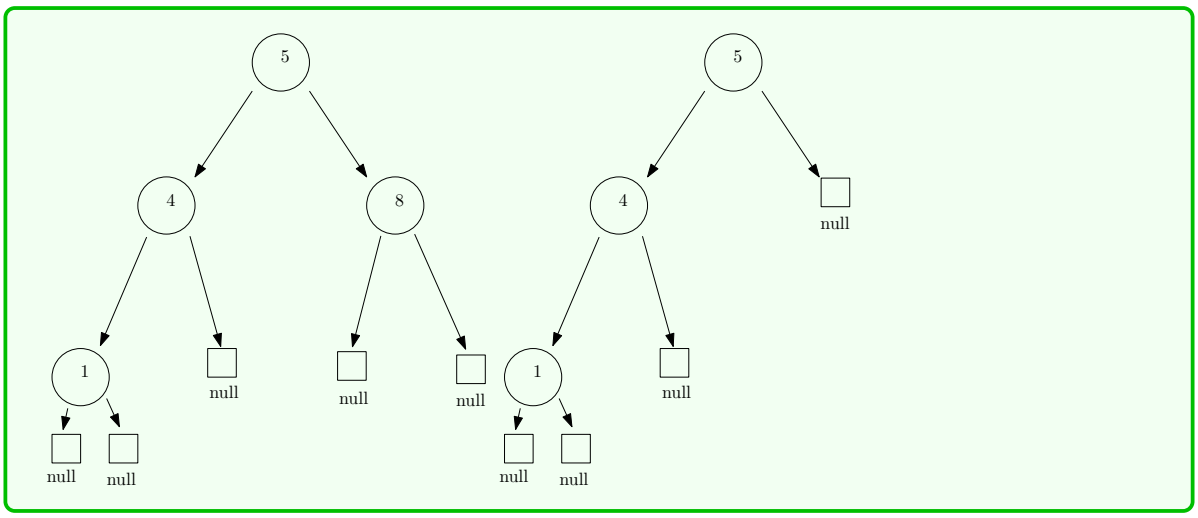
```
        if root.left is None:
                temp = root.right
                del root
                return temp
        elif root.right is None:
                temp = root.left
                del root
                return temp

        else:
                preParent = root
                pre = root.left
                while pre.right:
                        preParent = pre
                        pre = pre.right

                if preParent != root:
                        preParent.right = pre.left
                else:
                        preParent.left = pre.left
                root.value = pre.value
                del pre
                return root
```

Lösung 0.50 ***Delete Methode anwenden*** Übung S. 41




5
4
8
1
null
null
null
null
null
null
5
4
null
1
null
null
null
null