Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Leitprogramm der Informatik

Binäre Suchbäume

Björn Steffen
Timur Erdag
überarbeitet von Christina Class

Binäre Suchbäume

Ein ETH-Leitprogramm für die Informatik

Adressaten und Institutionen Das Leitprogramm richtet sich an Studierende an einer Fachhochschule im 2. Semester des Informatik- oder Elektrotechnikstudiums.

Vorkenntnisse Die Leser und Leserinnen

- beherrschen die wesentlichen Aspekte einer Programmiersprache.
- kennen das Konzept der verketteten Liste.
- verstehen die Rekursion und deren Anwendung.

Dauer Für die vollständige Bearbeitung des Leitprogramms werden ungefähr 6–8 Stunden benötigt.

Betreuerin und Überarbeitung
Christina Class

Autoren
Björn Steffen
Timur Erdag

9. April 2008

Vorwort

Bäume gehören zu den bedeutendsten Datenstrukturen in der Informatik. Dieses Leitprogramm gibt eine Einführung in dieses vielseitige Gebiet und befasst sich im Speziellen mit den binären Suchbäumen.

Das Leitprogramm richtet sich an Studierende an einer Fachhochschule im 2. Semester des Informatik- oder Elektrotechnikstudiums. Im Besonderen ist dieses Leitprogramm auf den Einsatz im Modul «Programmieren II» der Hochschule Luzern Technik & Architektur (HSLU T&A) ausgerichtet.

Es kann auch in anderen Schulstufen verwendet werden, sofern die im Folgenden beschriebenen Vorkenntnisse vorliegen.

Vorkenntnisse

Die Leserinnen und Leser

- beherrschen die wesentlichen Aspekte einer Programmiersprache.
- kennen das Konzept der verketteten Liste.
- verstehen die Rekursion und deren Anwendung.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Bäume gehören zu den bedeutendsten Datenstrukturen in der Informatik. Sie werden nicht nur zur effizienten Speicherung und schnellem Finden von Daten benutzt, sondern sind auch ein nützliches Hilfsmittel zur Strukturierung von Informationen.

Auch die visuelle Aussagekraft der Bäume wird in den verschiedensten Bereichen eingesetzt. Zum Beispiel für die Strukturierung von Menüs graphischer Benutzeroberflächen oder für das Präsentieren von hierarchisch gruppierten Informationen wie bei einem Dateisystem.

Für viele Anwendungen sind spezialisierte Bäume entwickelt worden. Für die Speicherung von Daten spielen zum Beispiel die binären Suchbäume eine wichtige Rolle. Diese Bäume stehen deshalb im Zentrum dieses Leitprogramms.

1.1 Lernziele des Leitprogramms

Nach der Bearbeitung dieses Leitprogramms sollten folgende Lernziele erreicht sein:

- Sie kennen die Möglichkeiten von Bäumen, im Speziellen von Binärbäumen, für den Einsatz in der Informatik.
- Sie können die wesentlichen Algorithmen von binären Suchbäumen wie das Suchen, Einfügen oder das Durchlaufen implementieren.
- Sie können evaluieren, ob ein binärer Suchbaum für ein gegebenes Problem eine geeignete Datenstruktur ist.
- Im Weiteren kennen Sie das Konzept von balancierten Suchbäumen am Beispiel von AVL-Bäumen.
- Am Ende des Leitprogramms sind Sie in der Lage, sich selbständig in weiterführender Literatur über Bäume zu informieren.

1.2 Übersicht

Dieses Leitprogramm führt die Leserinnen und Leser schrittweise in das Gebiet der binären Suchbäume ein.

Im folgenden Kapitel werden die wesentlichen Begriffe und Eigenschaften von Bäumen aufgezeigt, während im dritten Kapitel auf Binärbäume und deren Durchlaufordnungen eingegangen wird. Das Kapitel 4 befasst sich mit den Eigenschaften und der Implementierung von binären Suchbäumen. Im darauf folgenden Kapitel wird auf die Notwendigkeit einer Balancierung für Suchbäume eingegangen und im besonderen die AVL-Bäume vorgestellt.

1.3 Aufbau

Am Anfang jedes Kapitels werden nach einer kurzen Einführung die Lernziele aufgelistet, die in diesem Kapitel erreicht werden sollten.

Einige schwierigere oder weiterführende Abschnitte, Aufgaben oder Abbildungen sind mit einem Stern (⋆) markiert. *Diese Inhalte sind* optional *und für die Erreichung der Lernziele* nicht *notwendig.* Sie dienen den interessierten oder schnelleren Leserinnen und Lesern zur Vertiefung.

1.4 Aufgaben

Jedes Kapitel enthält eine Anzahl von eingestreuten Aufgaben, die direkt an der entsprechenden Stelle gelöst werden sollen. Die Aufgaben dienen nicht nur dem Einüben des Gelernten, sondern enthalten auch weiterführende Informationen und bringen zusätzliche Hinweise.

Im Weiteren befinden sich am Schluss jedes Kapitels zusätzliche Aufgaben zur Lernkontrolle und als Vorbereitung für die Kapiteltests oder allfällige Prüfungen. Für die Selbstkontrolle befinden sich in Anhang A auf Seite 69 die Lösungen zu den Aufgaben.

Es gibt zwei Arten von Aufgaben:

Aufgaben die mit einem Stift gekennzeichnet sind, können auf Papier gelöst werden.

Bei den Aufgaben mit einem CD-ROM Symbol handelt es sich um **Programmieraufgaben**, welche am Computer gelöst werden müssen. Dazu befinden sich auf EducETH weitere Hinweise und Codegerüste. Lösungen finden Sie zu diesen Aufgaben ebenfalls auf EducETH.

1.5 Kapiteltests

Am Ende jedes Kapitels wird ein Kapiteltest abgelegt. Dieser kurze Test umfasst einige Aufgaben zur Kontrolle des Lernfortschritts.

1.6 Programmiersprache und Pseudocode

Die Programmieraufgaben dieses Leitprogramms beziehen sich auf die Programmiersprache JAVA™. Die Aufgaben sollten jedoch ohne grosse Anpassungen auch in anderen Programmiersprachen umgesetzt werden können.

Die Algorithmen sind in Pseudocode verfasst, wie er häufig in der Literatur verwendet wird. Die Beispiele können somit leicht auf die jeweilige Programmiersprache übertragen werden. Ausserdem wird das Lesen und Schreiben von Pseudocode für die Beschreibung von Algorithmen und weitere Nachforschungen geübt.

1.7 Vorgehen

Wurde ein Kapitel fertig bearbeitet und die Aufgaben der Lernkontrolle gelöst, kann der Kapiteltest abgelegt werden. Bei erfolgreichem Bestehen des Tests, wird dann mit dem nächsten Kapitel weitergemacht. Sollten jedoch Schwierigkeiten bei der Lösung der Tests auftauchen, so müssen die entsprechenden Abschnitte des Kapitels nochmals bearbeitet und der Test zur Kontrolle wiederholt werden.

Dieses Vorgehen nennt man auch *Mastery Lernen*. Man fährt erst mit dem Lernen fort, wenn man den bisherigen Stoff beherrscht. Dies erhöht den Lernerfolg deutlich.

2 Bäume

In diesem Kapitel werden Sie mit dem erforderlichen Vokabular im Zusammenhang mit Bäumen vertraut gemacht. Dieses Vokabular ist intuitiv verständlich, da es sich auf bekannte Wörter und deren Bedeutung stützt. Jedoch müssen diese in einem neuen Zusammenhang gesehen werden. Wenn Sie diese einmal verstehen, werden Sie keine Mühe mehr haben, Texten über Bäume in der Informatik zu folgen.

2.1 Lernziele

Um sich mit Algorithmen und Anwendungen im Zusammenhang mit Bäumen zu befassen, ist es erforderlich, dass Sie die grundlegenden Begriffe im Gebiet der Bäume kennen und verstehen.

- Sie können die wesentlichen Elemente eines Baumes identifizieren und umgangssprachlich beschreiben.
- Am Ende dieses Kapitels sind Sie in der Lage, sich selbständig in weiterführender Literatur über Bäume zu informieren.

2.2 Der Baum

Was haben Bäume mit der Informatik zu tun? – werden Sie Sich wahrscheinlich gefragt haben, als Sie zum ersten Mal erfahren haben, dass Bäume auch in der digitalen Welt existieren. Sie werden sehen, dass Bäume in der Informatik einige Gemeinsamkeiten mit den natürlichen Bäumen haben. Zum Beispiel besitzen beide Blätter und Wurzel[1]. Und genau so, wie es in der Natur etliche Arten von Bäumen gibt, existieren auch in der Informatik verschiedene Arten. Jede Art ist spezialisiert und für eine bestimmte Anwendung optimiert.

Es gibt jedoch auch Unterschiede zwischen Bäumen der Informatik und natürlichen Bäumen: Sie „wachsen“ nicht in die gleiche Richtung. Bäume in der Informatik wachsen nach unten und haben dementsprechend die Wurzel oben wie ein natürlicher Baum, den Sie um 180 Grad drehen.

[1] Im Gegensatz zur Natur besitzt ein Baum in der Informatik jedoch nur eine Wurzel.

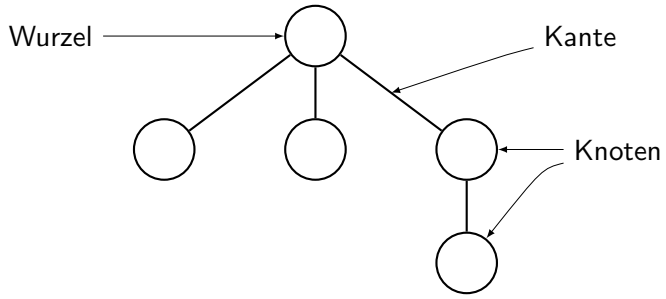


Abbildung 2.1 Ein Baum mit der Wurzel, mehreren Knoten und Kanten.

2.3 Die Komponenten eines Baumes

Knoten u. Kanten Die Knoten enthalten die Daten und werden miteinander durch Kanten verbunden.

Wurzel Jeder nicht leere Baum besitzt genau eine und sie ist ganz oben zu finden.

Vater u. Kind Väter sind mit ihren Kindern von unten mit einer Kante verbunden.

Ein **Baum** (tree) besteht aus verschiedenen Komponenten. Die wohl wichtigste ist der Knoten.

Ein **Knoten** (node) dient zur Speicherung von Daten. Knoten sind untereinander mit **Kanten** (edges) verbunden.

Besitzt ein Baum keine Knoten, so ist der Baum **leer**. Besitzt er hingegen Knoten, so können diese vom Typ Wurzel, innerer Knoten oder Blatt sein. Jeder nicht leere Baum besitzt genau eine **Wurzel** (root), welche normalerweise oben gezeichnet wird. Die verschiedenen Komponenten eines Baumes sind in der Abbildung 2.1 dargestellt.

Da die Wurzel ganz oben ist, wächst der Baum (wie bereits erwähnt) nach *unten*. Ist nun ein Knoten nach unten mit anderen Knoten durch Kanten verbunden, so sind diese unteren Knoten seine **Kinder** (child). Der obere Knoten wird als **Vater** dieser Kinder (vgl. Abbildung 2.2) bezeichnet.

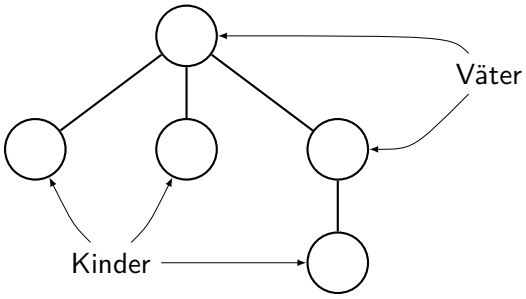


Abbildung 2.2 Die Wurzel ist Vater von drei Kindern und eines ihrer Kinder (ganz rechts) ist selber Vater eines Kindes.

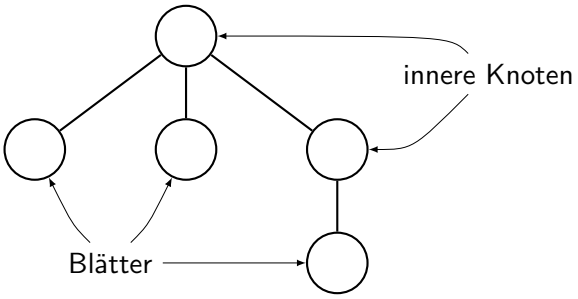


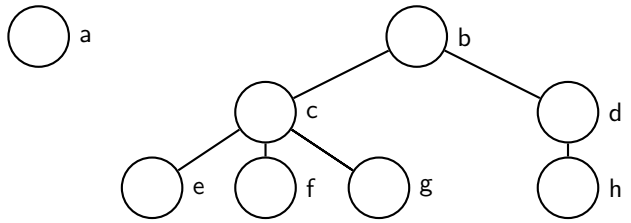
Abbildung 2.3 Ein Baum mit inneren Knoten und Blättern.

Ein **innerer Knoten** (inner node) ist ein Knoten mit mindestens einem Kind. Ist ein Knoten kinderlos, so spricht man von einem **Blatt** (leaf). (vgl. Abbildung 2.3)

Innerer Knoten Dieser Knoten hat mindestens ein Kind.

Blatt Dieser Knoten hat keine Kinder.

Aufgabe 2.1
Bestimmen Sie den Typ aller Knoten in folgender Abbildung. Beachten Sie, dass einzelne Knoten auch von mehreren Typen sein können.



2.4 Masse eines Baumes

Es gibt verschiedene Messgrössen, welche benutzt werden, um Eigenschaften eines Baumes zu beschreiben:

Die **Ordnung** (order) gibt an, wie viele Kinder ein innerer Knoten *höchstens* haben darf. Hat ein Baum zum Beispiel die Ordnung 2, so dürfen seine Knoten höchstens zwei Kinder haben. Ist jedoch nur die grafische Darstellung eines Baumes vorgegeben, so lässt sich seine Ordnung nicht ablesen. An Hand einer Grafik kann nur die Ordnung bestimmt werden, welche der Baum mindestens hat (Abbildung 2.4).

Ordnung Die maximale Anzahl der Kinder, die ein Knoten haben darf.

Der **Grad** (degree) eines Knotens sagt aus, wie viele Kinder ein Knoten effektiv hat.

$$\deg(v) = \text{Anzahl der Kinder von } v$$

Grad Die effektive Anzahl der Kinder, die ein Knoten hat.

Dabei gilt, dass der Grad *jedes* Knotens kleiner oder gleich der Ordnung des Baumes sein muss. In Abbildung 2.4 haben zum Beispiel die Knoten a und b den Grad 2 und der Knoten e hat den Grad 5.

Pfad Der Weg zwischen zwei Knoten.

Ein **Pfad** (path) von Knoten v nach Knoten w gibt an, welche Kanten und Knoten besucht werden, wenn man von v nach w wandert.

Tiefe Die Anzahl der Knoten auf dem direkten Pfad zur Wurzel.

Die **Tiefe** (depth) eines Knotens v gibt an, wie lange der Pfad zur Wurzel ist.

$$\text{depth}(v) = \text{Anzahl der Knoten auf dem Pfad von der Wurzel zum Knoten } v$$

Dabei werden alle Knoten gezählt, welche auf dem direkten Pfad von der Wurzel zum Knoten v liegen – inklusive der Wurzel und dem Knoten v selbst (siehe Abbildung 2.5).

Niveau Die Knoten mit der gleichen Tiefe.

Knoten, welche die gleiche Tiefe haben, werden zu **Niveaus** (level) zusammengefasst, wie Abbildung 2.6 zeigt.

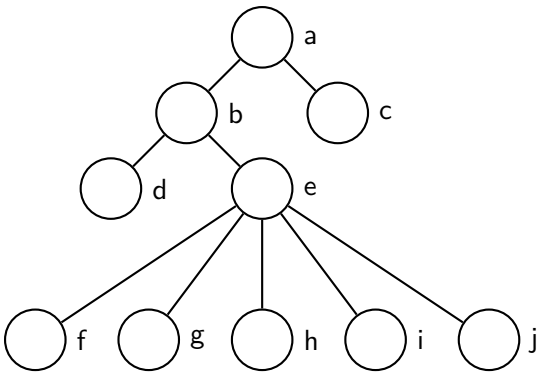


Abbildung 2.4 Ein Baum mit Ordnung ≥ 5.

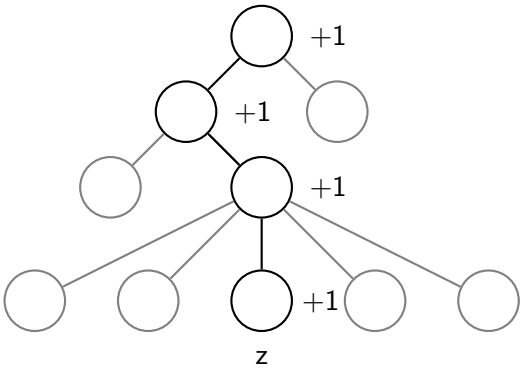


Abbildung 2.5 Die Tiefe vom Knoten z ist $1 + 1 + 1 + 1 = 4$.

Die **Höhe** (height) eines Baumes entspricht der Tiefe des Knoten, welcher am weitesten von der Wurzel entfernt ist.

Höhe Die Höhe ist gleich der Tiefe des tiefsten Knotens im Baum.

$$\text{height}(v) = \text{Die Höhe des Baumes mit Wurzel } v$$

Die Höhe eines Baumes entspricht somit dem grössten Niveau aller Knoten dieses Baumes (siehe Abbildung 2.7).

Ein Baum heisst **ausgefüllt**, wenn alle inneren Knoten die maximale Anzahl Kinder haben. Somit ist der Grad jedes Knotens entweder gleich der Ordnung des Baumes (innerer Knoten) oder 0 (Blatt) (siehe Abbildung 2.8).

Ausgefüllt Jeder innere Knoten besitzt die maximale Anzahl Kinder.

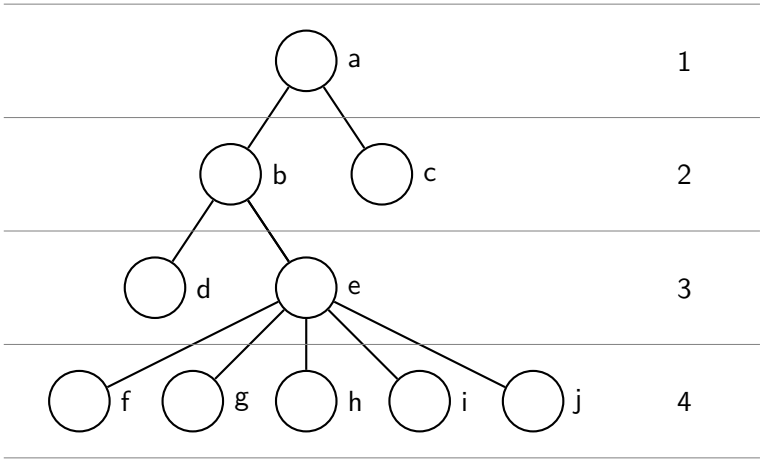


Abbildung 2.6 Die Knoten d und e haben das gleiche Niveau von 3.

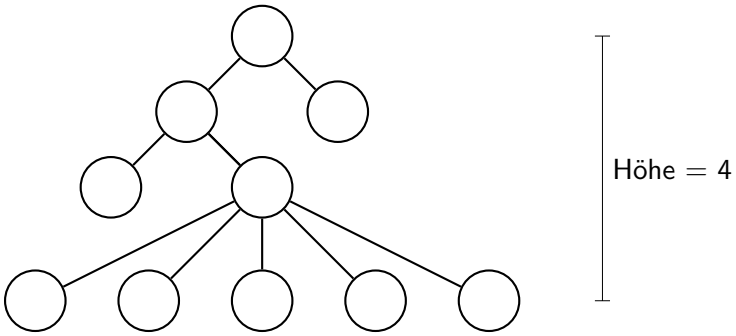


Abbildung 2.7 Die Höhe dieses Baumes beträgt 4.

Vollständig Jedes Niveau hat die maximale Anzahl Knoten.

Ein Baum heisst **vollständig**, wenn jedes Niveau die maximale Anzahl Knoten hat (siehe Abbildung 2.9). Jeder vollständige Baum ist auch ausgefüllt.

Bruder/Geschwister Die Knoten mit dem gleichen Vater.

Im Zusammenhang mit Bäumen, gibt es weitere Begriffe, welche wichtig sind. So wird ein Vater auch **Elternknoten** (parent) genannt. Knoten, welche den gleichen Vater haben, werden auch **Brüder** oder **Geschwister** (sibling) genannt.

Nun kennen Sie alle relevanten Begriffe der Bäume, welche für das weitere Studium dieses Leitprogramms nötig sind und haben damit eine wichtige Grundlage geschaffen.

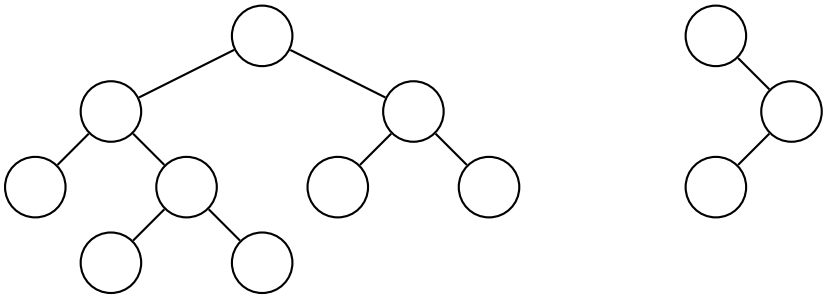


Abbildung 2.8 Wenn beide Bäume Ordnung 2 haben, dann ist nur der Linke ein ausgefüllter Baum.

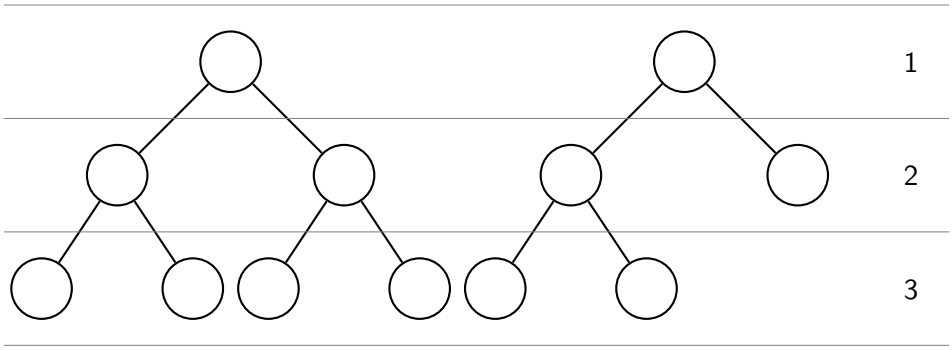


Abbildung 2.9 Wenn beide Bäume Ordnung 2 haben, dann ist der linke Baum vollständig. Der rechte Baum ist zwar ausgefüllt, aber nicht vollständig.

Aufgabe 2.2

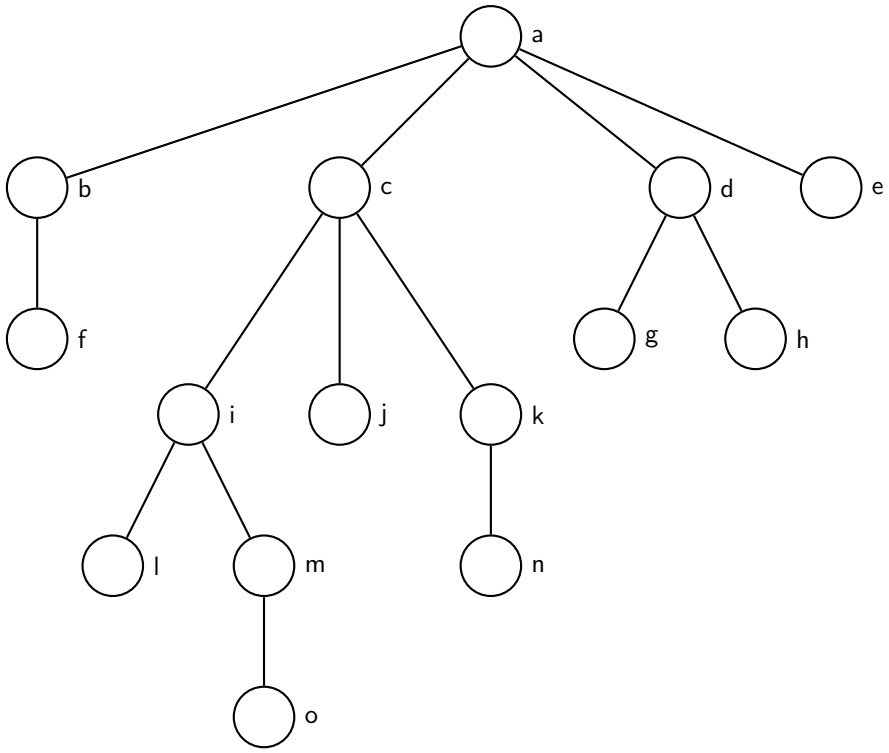
Bestimmen Sie für die zwei Bäume der Aufgabe 2.1 auf Seite 15:

a) die Höhe,
b) die Ordnung,
c) die Niveaus,
d) ob der Baum ausgefüllt ist,
e) den Grad aller Knoten
f) und die Tiefe jedes Knotens.

2.5 Lernkontrolle

Die folgenden Aufgaben dienen dem weiteren Üben und der Lernkontrolle der Inhalte dieses Kapitels. Wenn Sie alle Aufgaben lösen können, kann der Kapiteltest in Angriff genommen werden. Wenn jedoch Schwierigkeiten bestehen, sollten die entsprechenden Abschnitte des Kapitels nochmals bearbeitet werden.

Aufgabe 2.3
Bestimmen Sie im folgenden Baum die Typen der Knoten. Füllen Sie dazu die vorgegebene Tabelle aus.



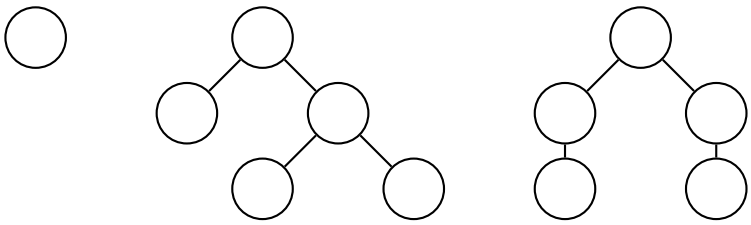
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Wurzel															
Blatt															
Innerer Knoten															

Aufgabe 2.4
Bestimmen Sie für jeden Knoten des Baumes der Aufgabe 2.3 die Tiefe und den Grad.

Aufgabe 2.5
Zeichnen Sie im Baum der Aufgabe 2.3 die Niveaus und die Höhe ein. Überlegen Sie sich,

welche Ordnung dieser Baum haben könnte, und begründen Sie ihre Entscheidung.

Aufgabe 2.6
Gegeben sind die drei folgenden Bäume der Ordnung 2. Entscheiden Sie, welche Bäume vollständig und welche ausgefüllt sind, und begründen Sie Ihre Antwort.



★ **Aufgabe 2.7**
Für viele Anwendungen gibt es verschiedene spezialisierte Bäume. Recherchieren Sie in verschiedenen Quellen (Internet, Bibliothek, ...) und notieren sich drei spezielle Bäume und deren wichtigste Anwendung.
Beachten Sie dabei besondere Eigenschaften der Bäume, wie zum Beispiel die Ordnung der Bäume.

3 Binärbäume

In der Informatik werden Bäume für die Strukturierung von Daten für verschiedene unterschiedliche Zwecke eingesetzt. Für spezifische Anwendungen sind deshalb spezialisierte Bäume entstanden. Von besonderem Interesse ist hierbei der Binärbaum.

Ein **Binärbaum** ist ein Baum der *Ordnung 2*. Somit hat jeder Knoten höchstens zwei Kinder. Durch diese Einschränkung vereinfachen sich viele Algorithmen und Repräsentationen der Bäume. Der Binärbaum wird in der Informatik daher oft eingesetzt und liefert die Grundlage für weitere spezialisiertere Bäume.

Für viele Anwendungen ist es ausserdem notwendig alle Knoten eines Baumes der Reihe nach zu bearbeiten. Da die Knoten jedoch nicht linear angeordnet sind wie zum Beispiel in einer Liste, gibt es verschiedene **Durchlaufordnungen**, um alle Knoten eines Baumes zu durchlaufen.

3.1 Lernziele

Da die Binärbäume eine Grundlage für weitere spezialisiertere Bäume sind, ist es wichtig, dass Sie die Definition der Binärbäume kennen. Im Weiteren ist es für viele Anwendungen unerlässlich, dass alle Knoten eines Baumes in einer bestimmten Reihenfolge durchlaufen werden.

Für dieses Kapitel des Leitprogramms gelten die folgenden Lernziele:

- Sie kennen die Definition und die Besonderheiten der Binärbäume.
- Sie kennen die drei wichtigsten Durchlaufordnungen und deren Eigenschaften.
- Sie können diese Durchlaufordnungen für Binärbäume implementieren.
- Für die drei Durchlaufordnungen können Sie je eine Beispielanwendung angeben und begründen, weshalb diese Reihenfolge verwendet wurde.

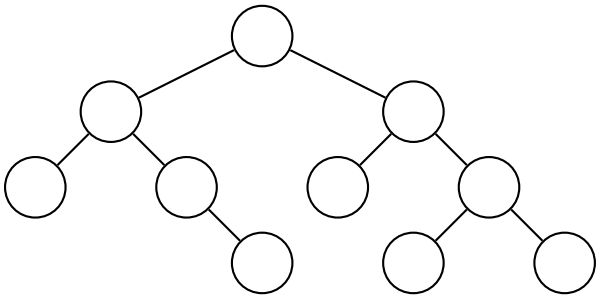
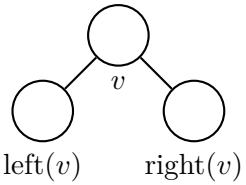


Abbildung 3.1 Ein **Binärbaum**. Jeder Knoten v des Baumes hat *höchstens zwei* Kinder, ein Linkes ($\text{left}(v)$) und ein Rechtes ($\text{right}(v)$).

3.2 Definition von Binärbäumen

Binärbaum Ein Baum der Ordnung 2. Knoten sind entweder linkes oder rechtes Kind ihres Elternknotens.

Ein binärer Baum oder **Binärbaum** (binary tree) ist ein Baum der Ordnung 2. Die inneren Knoten eines Binärbaumes haben somit *höchstens* zwei Kinder. Diese Kinder werden als linkes ($\text{left}(v)$) bzw. rechtes Kind (Sohn) ($\text{right}(v)$) eines Knotens v bezeichnet.

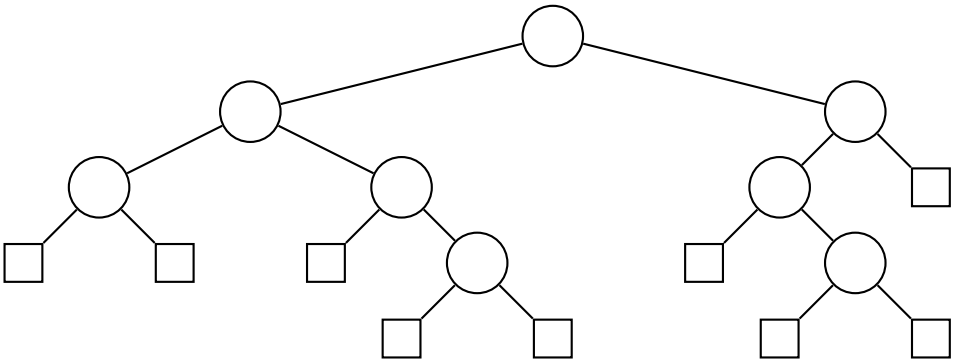


Diese Vereinfachung der Repräsentation von Bäumen erlaubt es, wesentlich einfachere Algorithmen und Datenstrukturen zu definieren (siehe Abbildung 3.1).

⋆ 3.2.1 Alternative Definition von Binärbäumen

In der Literatur existiert eine weitere Definition von Binärbäumen. Beide Definition sind von Bedeutung, entsprechend haben beide Vor- und Nachteile. In diesem Abschnitt wird deshalb die zweite geläufige Definition erläutert.

Nach dieser Definition hat jeder Knoten des Binärbaums *genau zwei* oder *keine* Kinder. Die so definierten Binärbäume sind somit *immer* **ausgefüllt**. Um diese Eigenschaft sicherzustellen, werden in den Binärbaum an bestimmten Stellen *leere* Blätter eingefügt.



★ **Abbildung 3.2** Der Binärbaum nach der alternativen Variante. Durch eingefügte leere Blätter ist dieser Binärbaum immer **ausgefüllt**.

Leere Blätter sind die einzigen Knoten die keine Kinder haben und werden mit dem folgenden Symbol dargestellt:

Ein Binärbaum B wird folgendermassen in die alternative Variante B' umgewandelt:

1. Alle Blätter des Binärbaumes B erhalten zwei leere Blätter und werden somit zu inneren Knoten.

2. Alle inneren Knoten des Baumes B, die noch nicht zwei Kinder besitzen, erhalten an entsprechender Stelle ein leeres Blatt.

Der resultierende Binärbaum B' hat dann die Eigenschaften dieser alternativen Definition. Abbildung 3.2 zeigt einen Binärbaum dieser Definition.

Diese strikte Variante der Definition erlaubt es gewisse Algorithmen einfacher zu definieren, da alle inneren Knoten immer *genau* zwei Kinder haben. Andererseits werden die Abbildungen dadurch überladen und die Beschreibungen und Definitionen unnötig kompliziert. In diesem Leitprogramm wird deshalb die weniger restriktive Variante der Definition eines Binärbaumes verwendet.

3.3 Durchlaufordnungen in Binärbäumen

Oft ist es notwendig, *alle* Knoten eines Baumes der Reihe nach zu besuchen. Solche Anwendungen sind zum Beispiel die Bestimmung der Anzahl Knoten eines Baumes oder seiner Höhe oder das Ausgeben aller Knoten des Baumes.

Durchlaufordnung Die Reihenfolge, in der die Knoten eines Baumes durchlaufen werden.

Bei linearen Datenstrukturen, wie zum Beispiel der verketten Liste, können deren Elemente vom vordersten bis zum hintersten Element der Reihe nach durchlaufen werden. Bei Bäumen sind verschiedene **Durchlaufordnungen** möglich. Die Aktion beim Besuchen eines Knotens ist dabei für verschiedene Anwendungen unterschiedlich und reicht vom einfachen Ausgeben des Knotens bis zu komplizierten Berechnungen für diesen Knoten. Das Durchlaufen eines Baumes wird auch als **Traversierung** bezeichnet.

Für das Durchlaufen von **Binärbäumen** sind drei Reihenfolgen von besonderer Bedeutung: die **Preorder-** (Hauptreihenfolge), die **Postorder-** (Nebenreihenfolge) und die **Inorder-Reihenfolge** (symmetrische Reihenfolge).

Die Bezeichnungen Preorder, Postorder und Inorder verdeutlichen, ob ein Knoten *vor*, *nach* oder *zwischen* seinen Teilbäumen besucht wird.

Die Reihenfolgen können auf einfache Weise rekursiv formuliert werden.

3.3.1 Preorder-Reihenfolge (Hauptreihenfolge)

Preorder-Reihenfolge Bei dieser Reihenfolge wird ein Knoten vor seinen Kindern durchlaufen.

Bei der **Preorder-Reihenfolge** (Hauptreihenfolge) wird ein Knoten jeweils vor seinem linken und rechten Teilbaum durchlaufen. Die **Preorder-Reihenfolge** wird also zuerst die Wurzel des Baumes besuchen. Danach wird *rekursiv* der linke Teilbaum und dann der rechte Teilbaum durchlaufen.

Das Durchlaufen eines Binärbaumes mit Wurzel v in der Preorder-Reihenfolge erfolgt in den folgenden Schritten:

1. Besuche den Knoten v.
2. Durchlaufe den linken Teilbaum des Knotens v in der Preorder-Reihenfolge.
3. Durchlaufe den rechten Teilbaum des Knotens v in der Preorder-Reihenfolge.

Beim Durchlaufen eines Baumes in der Preorder-Reihenfolge entsteht eine lineare Ordnung der Knoten, bei der die Elternknoten immer *vor* ihren Kindern besucht werden, wie Abbildung 3.3 zeigt.

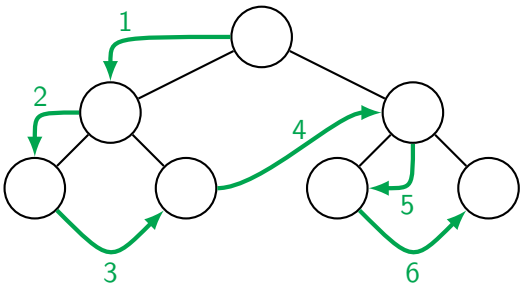


Abbildung 3.3 Durchlaufordnung eines Binärbaumes in der **Preorder-Reihenfolge**. Ein Knoten wird jeweils *vor* seinem linken und rechten Teilbaum durchlaufen.

Der folgende Pseudocode zeigt den rekursiven Algorithmus preorder() für diese Durchlaufordnung.

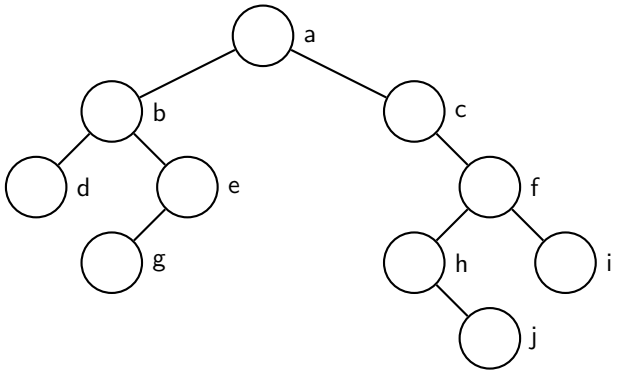
```
1: algorithm preorder(v)
2:     {Durchläuft alle Knoten des Binärbaumes mit Wurzel v in der Preorder-
   Reihenfolge}
3:     if v ≠ null then
4:         Besuche den Knoten v {∗}
5:         preorder( left(v) )
6:         preorder( right(v) )
7:     end if
8: end algorithm
```

Bei der mit {∗} markierten Stelle im Algorithmus findet die eigentliche Bearbeitung des besuchten Knotens statt. Zum Beispiel kann hier der Inhalts des Knotens ausgegeben werden.

Die Preorder-Reihenfolge ist dann nützlich, wenn die Berechnung für einen Knoten vor der Berechnung seiner Kinder stattfinden muss.

★ **Aufgabe 3.1**
Die Preorder-Reihenfolge ist nicht nur für das Durchlaufen von Binärbäumen anwendbar. Erweitern sie den gegebenen Algorithmus preorder() so, dass beliebige Bäume damit durchlaufen werden können.

Aufgabe 3.2
Bestimmen Sie für den folgenden Binärbaum die Durchlaufordnung nach der Preorder-Reihenfolge.



Programmieraufgabe 3.3
Schreiben Sie eine Methode `preorderPrint()` in der Klasse `BinaryNode`. In dieser Methode werden alle Knoten eines Binärbaumes in der Preorder-Reihenfolge ausgegeben.

3.3.2 Postorder-Reihenfolge (Nebenreihenfolge)

Postorder-Reihenfolge
Bei dieser Reihenfolge wird ein Knoten nach seinen Kindern durchlaufen.

Das Durchlaufen eines Binärbaumes in der **Postorder-Reihenfolge** (Nebenreihenfolge) erfolgt ähnlich zur Preorder-Reihenfolge. Bei dieser Durchlaufordnung wird jedoch ein Knoten v *nach* seinem linken und rechten Teilbaum besucht.

1. Durchlaufe den linken Teilbaum des Knotens v in der Postorder-Reihenfolge.

2. Durchlaufe den rechten Teilbaum des Knotens v in der Postorder-Reihenfolge.

3. Besuche den Knoten v.

Die Postorder-Reihenfolge liefert beim Durchlaufen eines Baumes eine lineare Ordnung der Knoten, bei dem die Elternknoten immer *nach* ihren Kindern besucht werden, wie Abbildung 3.4 zeigt.

Das Durchlaufen in der Posterorder-Reihenfolge ist für verschiedene Anwendungen nützlich. Zum Beispiel wenn für jeden Knoten v eine Eigenschaft berechnet werden soll, für deren Bestimmung aber zuerst die Eigenschaften der Kinder von v berechnet werden müssen.

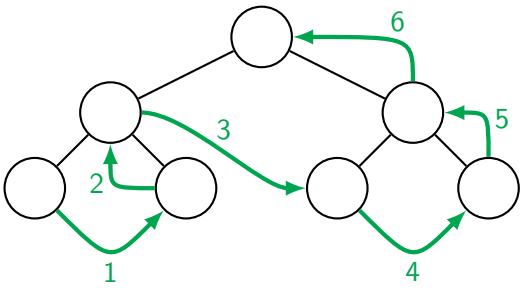
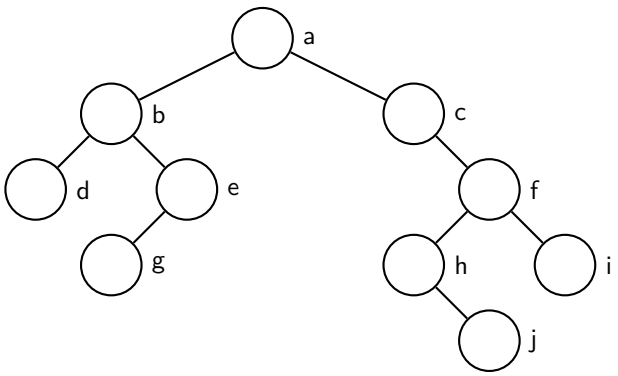


Abbildung 3.4 Durchlaufordnung eines Binärbaumes in der **Postorder-Reihenfolge**. Ein Knoten wird jeweils *nach* seinem linken und rechten Teilbaum durchlaufen.

Aufgabe 3.4
Bestimmen Sie für den folgenden Binärbaum die Durchlaufordnung nach der Postorder-Reihenfolge.



Aufgabe 3.5
Schreiben Sie einen rekursiven Algorithmus postorder() in Pseudocode, der alle Knoten eines Binärbaumes in der Postorder-Reihenfolge besucht.

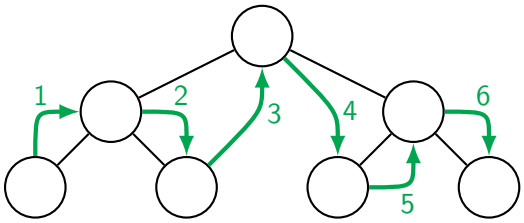


Abbildung 3.5 Durchlaufordnung eines Binärbaumes in der **Inorder-Reihenfolge**. Ein Knoten wird jeweils *zwischen* seinem linken und rechten Teilbaum durchlaufen.

3.3.3 Inorder-Reihenfolge (Symmetrische Reihenfolge)

Die Preorder- und die Postorder-Reihenfolgen sind für alle Bäume anwendbar, die Inorder-Reihenfolge ist jedoch nur für Binärbäume definiert.

Inorder-Reihenfolge
Bei dieser Reihenfolge wird ein Knoten zwischen seinen Kindern durchlaufen.

Bei der **Inorder-Reihenfolge** (symmetrischen Reihenfolge) wird ein Knoten v *zwischen* dem Durchlaufen seines linken und rechten Teilbaumes besucht.

1. Durchlaufe den linken Teilbaum des Knotens v in der Inorder-Reihenfolge.

2. Besuche den Knoten v.

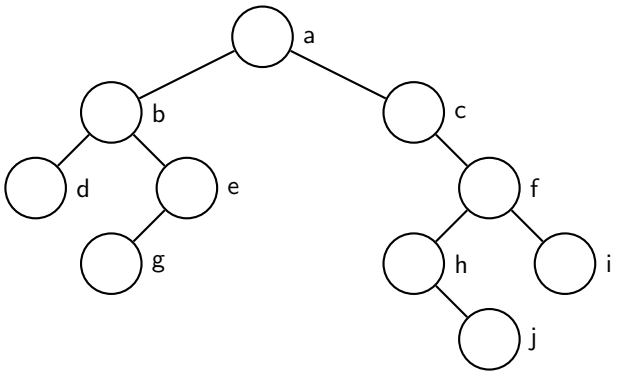
3. Durchlaufe den rechten Teilbaum des Knotens v in der Inorder-Reihenfolge.

Beim Durchlaufen eines Binärbaumes in der Inorder-Reihenfolge ergibt sich eine lineare Ordnung der Knoten. In dieser Ordnung erscheint ein Knoten immer *zwischen* seinen Kindern (siehe Abbildung 3.5). Die Knoten des Binärbaumes werden quasi von links nach rechts besucht.

Die Inorder-Reihenfolge hat ebenfalls verschiedene Anwendungen. Diese Durchlaufordnung spielt zum Beispiel eine wichtige Rolle bei den binären Suchbäumen des nächsten Kapitels.

Aufgabe 3.6
Schreiben Sie einen Algorithmus inorderNext(v) in Pseudocode der für einen Binärbaum den nächsten Knoten nach v in der Inorder-Reihenfolge liefert. Gehen Sie dabei davon aus, dass ein solcher Knoten existiert. (Sie können also annehmen, dass right(v) $\neq$ **null**)

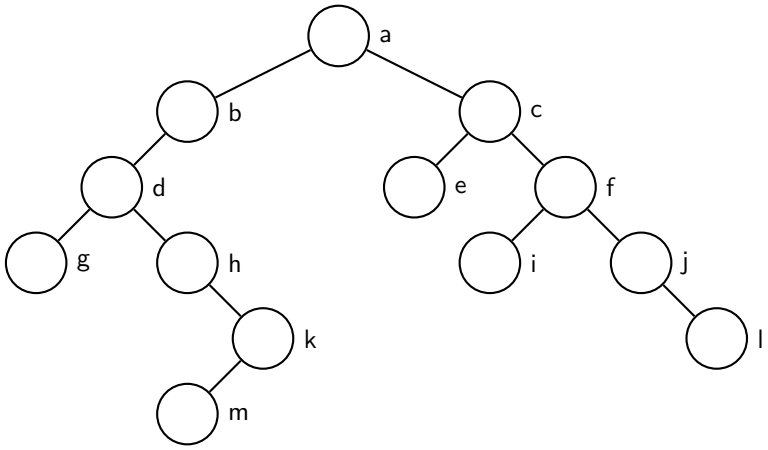
Aufgabe 3.7
Bestimmen Sie für den folgenden Binärbaum die Durchlaufordnung nach der Inorder-Reihenfolge.



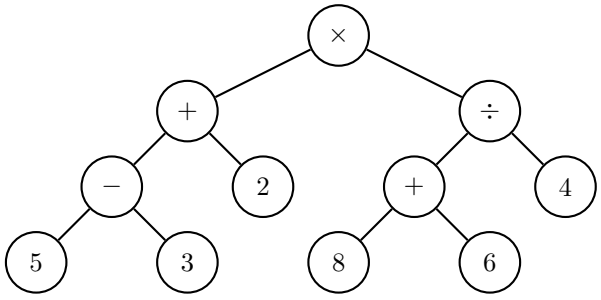
3.4 Lernkontrolle

Die folgenden Aufgaben dienen dem weiteren Üben und der Lernkontrolle der Inhalte dieses Kapitels. Wenn Sie alle Aufgaben lösen können, kann der Kapiteltest in Angriff genommen werden. Wenn jedoch Schwierigkeiten bestehen, sollten die entsprechenden Abschnitte des Kapitels nochmals bearbeitet werden.

Aufgabe 3.8
Schreiben Sie für den folgenden Binärbaum die Durchlaufordnungen für die Preorder-, Postorder- und die Inorder-Reihenfolge auf.



Aufgabe 3.9
Um mathematische Ausdrücke durch Programme auszuwerten, muss der Ausdruck in eine geeignete Repräsentation gebracht werden. Eine mögliche Repräsentation ist der abstrakte Syntaxbaum, wie er für einen gegebenen Ausdruck in der folgenden Abbildung dargestellt ist.



Das Erstellen von Syntaxbäumen aus Texten ist ein wesentlicher Bestandteil von Compilern und Interpretern.

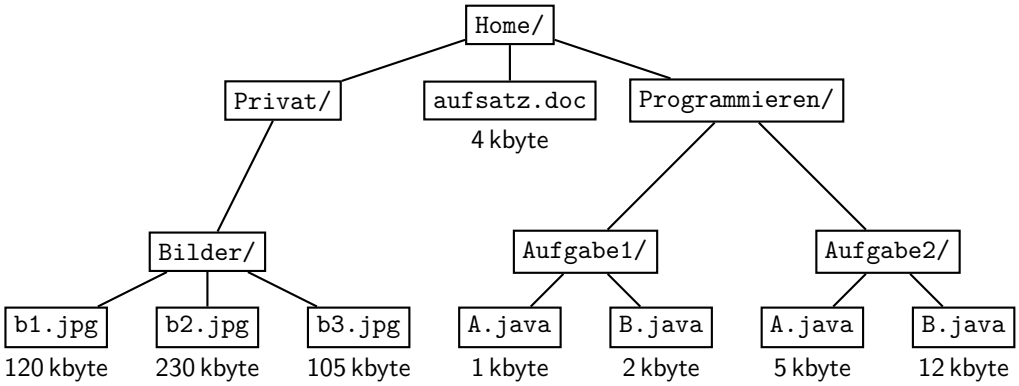
Wichtig ist das richtige Setzen der Klammern, denn die beiden Teilbäume eines Knotens müssen vor diesem Knoten ausgewertet werden. Dies kann man erreichen, indem man standardmässig jeden Ausdruck eines Teilbaumes mit einer Klammer umschliesst. Mit einer passenden Traversierung des Syntaxbaumes erhält man den mathematischen Ausdruck, der im Baum dargestellt ist.

Welche Durchlaufordnung wird dabei angewandt?

Wie lautet der mathematische Ausdruck, der durch den Syntaxbaum repräsentiert wird?

Aufgabe 3.10

Ein Dateisystem lässt sich einfach als Baum darstellen. Die Blätter repräsentieren die Dateien. Die inneren Knoten entsprechen den Verzeichnissen oder Ordnern. Die folgende Grafik zeigt einen Ausschnitt aus einem Dateisystem mit Angabe der Dateigrössen. Die Verzeichnisse selbst benötigen dabei keinen Speicherplatz.



Beschreiben Sie in Stichworten, wie Sie vorgehen können, um den verwendeten Speicherplatz der Inhalte aller Verzeichnisse zu berechnen.

Geben Sie einen Algorithmus an, der das gegebene Problem in möglichst wenigen Schritten löst.

★ **Aufgabe 3.11**

B sei ein *vollständiger* Binärbaum der Höhe h. Bestimmen Sie die folgenden Werte für B in Abhängigkeit von h:

a) Die Anzahl der Blätter im Binärbaum B.

b) Die Anzahl der Knoten im Binärbaum B.

★ **Aufgabe 3.12**

Sei B ein Baum mit mehr als einem Knoten.

a) Ist es möglich, dass beim Durchlaufen des Baumes B in der Preorder-Reihenfolge die gleiche Reihenfolge der Knoten entsteht, wie wenn B in der Postorder-Reihenfolge durchlaufen wird?

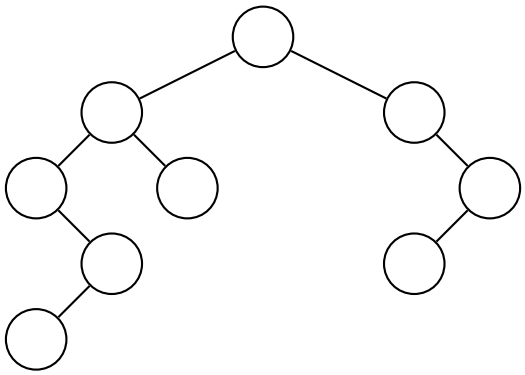
Geben Sie ein Beispiel, falls dies zutrifft, oder argumentieren Sie, wieso dies nicht eintreten kann.

b) Ist es möglich, dass beim Durchlaufen des Baumes B in der Preorder-Reihenfolge die *umgekehrte* Reihenfolge der Knoten entsteht, wie wenn B in der Postorder-Reihenfolge durchlaufen wird?

Geben Sie ein Beispiel, falls dies zutrifft, oder argumentieren Sie, wieso dies nicht eintreten kann.

★ **Aufgabe 3.13**

Erweitern Sie den folgenden Binärbaum so, dass er der alternativen Definition für Binärbäume aus Abschnitt 3.2.1 entspricht.



4 Binäre Suchbäume

In den vorherigen zwei Kapiteln haben Sie Bäume im Allgemeinen sind und Binärbäume im Speziellen kennengelernt. Nun nutzen Sie dieses Wissen, um eine erweiterte Form der Binärbäume zu verstehen. Hierbei handelt es sich um eine Baumstruktur, welche insbesondere auf das Suchen und Entfernen von Knoten spezialisiert ist. Diese erweiterten Binärbäume werden **Binäre Suchbäume** genannt.

4.1 Lernziele

Um binäre Suchbäume korrekt in einem Programm als Datenstruktur anzuwenden, ist es erforderlich, dass Sie deren Aufbau verstehen.

- Sie können einen binären Suchbaum als Datenstruktur anwenden.
- Sie kennen Unterschiede und Gemeinsamkeiten zwischen Binärbäumen und binären Suchbäumen.
- Sie können die Operationen **Suchen** und **Einfügen** eines binären Suchbaumes auf Papier skizzieren und implementieren.
- Sie können das Prinzip der Operation **Entfernen** auf Papier aufzeichnen.

4.2 Definition von binären Suchbäumen

Ein **binärer Suchbaum** (binary search tree) unterscheidet sich in seiner Struktur kaum von einem Binärbaum. Binäre Suchbäume beziehungsweise deren Knoten brauchen jedoch ein zusätzliches Erkennungsmerkmal. Dieses Erkennungsmerkmal wird Schlüssel genannt.

Ein **Schlüssel** (key) kennzeichnet die Daten und erlaubt es, sie so abzulegen, dass diese schnell wieder gefunden werden. Dabei ist es wichtig, entweder in den Daten selber einen geeigneten Schlüssel zu finden oder diesen basierend auf (Teil-)Daten zu berechnen. In diesem Leitprogramm werden zur Vereinfachung nur kleine ganze Zahlen als Schlüssel betrachtet, welche dann zugleich auch die

Binärer Suchbaum Ein geordneter Binärbaum, wobei jeder Knoten einen Schlüssel beinhaltet.

Schlüssel Der Schlüssel kennzeichnet die Daten die in einem Knoten abgelegt sind.

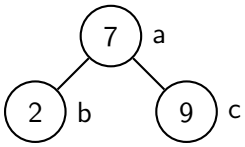


Abbildung 4.1 Knoten a besitzt Schlüssel 7, Knoten b hat Schlüssel 2 und Knoten c besitzt Schlüssel 9.

Daten darstellen. Die Abbildung 4.1 zeigt einen Suchbaum bei dem die Schlüssel der Knoten eingetragen sind.

Der Schlüssel, der einem Knoten v zugeordnet ist, wird als $\text{key}(v)$ bezeichnet. In der Abbildung 4.2 gilt beispielsweise $\text{key}(\text{f}) = 3$.

$$\text{key}(v) = \text{Schlüssel vom Knoten } v$$

Basierend auf den Schlüsseln können die Eigenschaften definiert werden, die einen Binärbaum zu einem **binären Suchbaum** machen:

- Jeder Schlüssel im linken Teilbaum eines Knotens ist kleiner als der Schlüssel im Knoten selbst.
- Jeder Schlüssel im rechten Teilbaum eines Knotens ist grösser als oder gleich dem Schlüssel im Knoten selbst.

Aus den zwei eben genannten Eigenschaften folgt, dass bei der Inorder-Reihenfolge der Knoten (siehe Abschnitt 3.3.3) die Schlüssel in aufsteigender Reihenfolge betrachtet werden. Um dies zu verdeutlichen zeigt Abbildung 4.3 die Ausgabe der Schlüssel der Bäume aus Abbildung 4.2 in der Inorder-Reihenfolge.

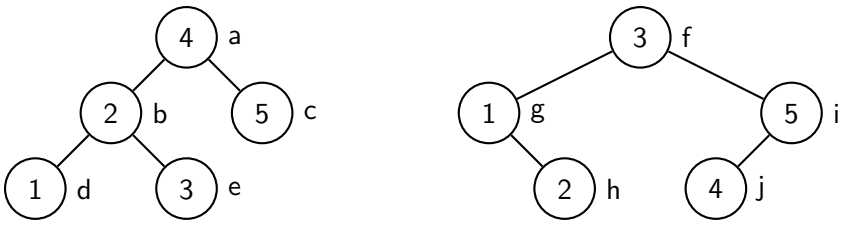


Abbildung 4.2 Zwei binäre Suchbäume mit den gleichen Schlüsseln. Je nachdem in welcher Reihenfolge die Knoten eingefügt wurden, sehen die Suchbäume anders aus.

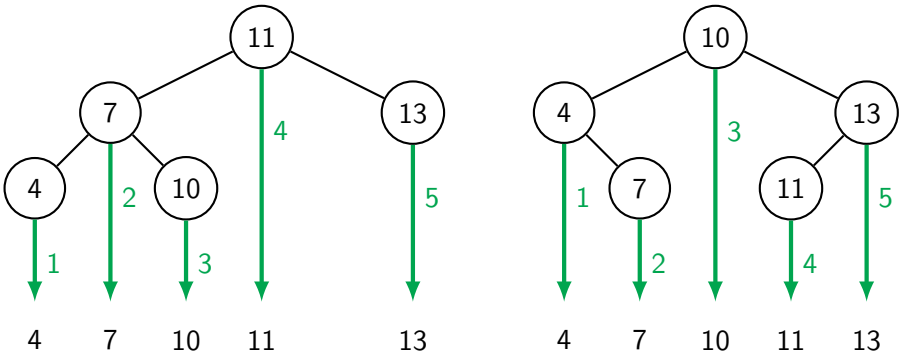


Abbildung 4.3 Die beiden binären Suchbäume haben die gleichen Schlüssel. Die Inorder-Reihenfolge der Schlüssel ist somit für beide Suchbäume identisch.

Aufgabe 4.1
Bestimmen Sie die Werte in Abbildung 4.2 von key(a), key(b), key(e) und key(g), key(i).

★ **Aufgabe 4.2**
In diesem Leitprogramm werden, wie bereits erwähnt, nur ganzzahlige Schlüssel verwendet. Suchen Sie nach geeigneten Schlüsseln für die folgenden Daten, welche in einem binären Suchbaum gespeichert werden sollen: Personen, Bücher, Adressen. Begründen Sie Ihre Wahl.

4.3 Suchen, Einfügen und Entfernen von Knoten

Sie kennen nun den Aufbau eines binären Suchbaums. In diesem Abschnitt werden die Operationen eines binären Suchbaums vorgestellt. Dabei wird zuerst auf das Suchen und Einfügen von Knoten eingegangen. Anschliessend wird das Prinzip betrachtet, wie ein Knoten entfernt wird.

4.3.1 Suchen eines Knotens

Ein Knoten in einem binären Suchbaum zu **suchen** (search) ist sehr effizient. Dabei muss der zu suchende Schlüssel k bekannt oder berechenbar sein.

Der folgende Algorithmus search($root, key$) zeigt in Pseudocode den rekursiven Algorithmus für das Suchen eines Knotens mit dem Schlüssel k.

```
1:  algorithm search(v, k)
2:      {Im Baum mit Wurzel v wird der Schlüssel k gesucht}
3:      if v ≠ null then
4:          if k < key(v) then
5:              search(left(v), k) {Suche im linken Teilbaum left(v)}
6:          else
7:              if k > key(v) then
8:                  search(right(v), k) {Suche im rechten Teilbaum right(v)}
9:              else
10:                 Beende Suche {Suche war erfolgreich}
11:             end if
12:         end if
13:     else
14:         Beende Suche {Suche war erfolglos}
15:     end if
16: end algorithm
```

Dieser Algorithmus wird nur den *ersten* Knoten mit dem Schlüssel k suchen und auch finden, falls er vorhanden ist. Hat es jedoch weitere Knoten mit diesem Schlüssel, werden diese nicht gefunden.

Programmieraufgabe 4.3
Setzen Sie die Theorie in die Praxis um. Implementieren Sie den Algorithmus search($root, key$). Hierzu steht Ihnen die Klasse `BinSearchTree` zur Verfügung.

★ **Aufgabe 4.4**
Ändern Sie den Pseudocode von search(v, k) so, dass nun alle Knoten mit dem gleichen Schlüssel ausgegeben werden.

★ **Programmieraufgabe 4.5**
Implementieren Sie den Algorithmus search($root, key$) aus Aufgabe 4.4 in der Klasse `BinSearchTree`.

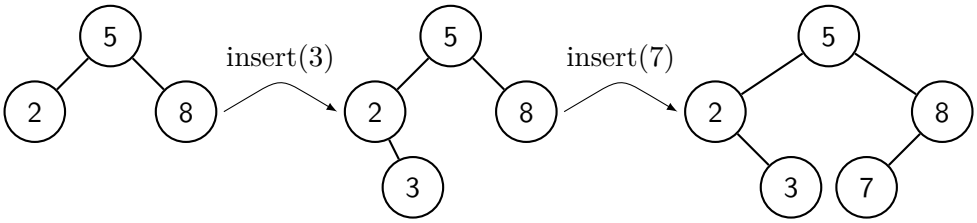


Abbildung 4.4 Das Einfügen eines Knotens mit Schlüssel 3 und 7 in den binären Suchbaum.

4.3.2 Einfügen eines neuen Knotens

Das **Einfügen** (insert) eines Knotens erfordert das Suchen der richtigen Einfügestelle im Baum. Dabei wird ähnlich vorgegangen wie beim Algorithmus search(). In jedem Knoten wird eine Vergleichsoperation ausgeführt und entschieden, ob der einzufügende Knoten im linken oder rechten Teilbaum gespeichert werden soll. Dieser Vergleich wird rekursiv ausgeführt bis der jeweilige Teilbaum leer[1] ist. Hier ist nun der freie Platz, an dem der Knoten mit den Daten eingefügt werden kann. (vgl. Abbildung 4.4).

Der folgende Algorithmus insert($root, key$) in Pseudocode fgt einen Knotens mit dem Schlüssel k in den binären Suchbaum ein.

```
1:  algorithm insert(v, k)
2:      {Im Baum mit Wurzel v wird nach einem Platz gesucht, an dem der
    neue Knoten mit Schlüssel k gespeichert werden kann.}
3:      if v ≠ null then
4:          if k < key(v) then
5:              insert(left(v), k) {Knoten muss im linken Teilbaum gespeichert
    werden}
6:          else
7:              insert(right(v), k) {Knoten muss im rechten Teilbaum gespeichert
    werden}
8:          end if
9:      else
10:         Füge hier den neuen Knoten mit Schlüssel k ein.
11:     end if
12: end algorithm
```

Die Reihenfolge in der Knoten mit ihren Daten und Schlüsseln in einen binären Suchbaum einfügt werden, bestimmt die Struktur des Baumes (vgl. Aufgabe 4.6).

[1] Leer bedeutet, dass der Teilbaum keine Knoten hat, also **null** ist.

Aufgabe 4.6
Ausgangslage ist jeweils ein leerer Baum. Fügen Sie die folgenden neun Knoten in den drei verschiedenen vorgegebenen Reihenfolgen nacheinander in den Baum ein.

- Reihenfolge a: 1, 2, 3, 4, 5, 6, 7, 8, 9
- Reihenfolge b: 5, 3, 7, 4, 6, 2, 8, 1, 9
- Reihenfolge c: 8, 2, 7, 3, 6, 4, 5, 9, 1

Aufgabe 4.7
In Aufgabe 4.6 haben Sie gesehen, dass die Einfügereihenfolge wichtig ist. Nun wird der umgekehrte Weg eingeschlagen. Verwenden Sie Ihren in Aufgabe 4.6 gezeichneten Baum (Reihenfolge b). Nennen Sie mindestens zwei weitere Reihenfolgen, welche den gleichen binären Suchbaum erzeugen.

Programmieraufgabe 4.8
Implementieren Sie den Algorithmus insert($root, key$) zum Einfügen eines neuen Schlüssels in der Klasse `BinSearchTree`.

4.3.3 Entfernen eines Knotens

Das **Entfernen** (delete) von Daten bzw. Knoten ist komplexer als die beiden Algorithmen search() und insert(), weshalb wir hier auf deren Implementierung verzichten.

Die Schwierigkeit beim Entfernen liegt darin, die Eigenschaften des binären Suchbaums zu erhalten. Hierzu wird zwischen 3 Fällen unterschieden:

Fall 1 Der zu löschende Knoten v ist ein Blatt und hat demzufolge keine Kinder.

Fall 2 Der zu löschende Knoten v ist ein innerer Knoten mit Grad 1.

Fall 3 Der zu löschende Knoten v ist ein innerer Knoten mit Grad 2.

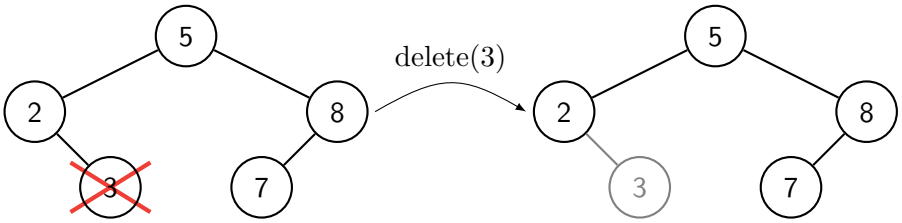


Abbildung 4.5 Fall 1: Das Entfernen des Blattes mit Schlüssel 3.

Fall 1

Fall 1 ist am Einfachsten: Es wird nach dem zu entfernenden Knoten gesucht und dieser dann einfach entfernt (vgl. Abbildung 4.5).

Fall 2

Fall 2 ist auch relativ einfach: Es wird nach dem zu entfernenden Knoten gesucht. Dieser wird gelöscht und mit seinem Kind ersetzt (Abbildung 4.6).

Die nächste Aufgabe veranschaulicht diese zwei Fälle. Danach wird Fall 3 erläutert.

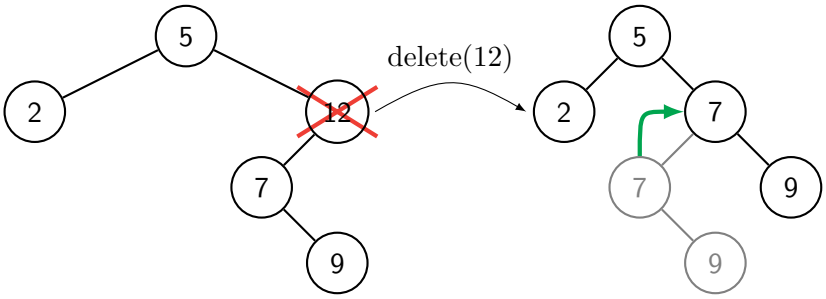
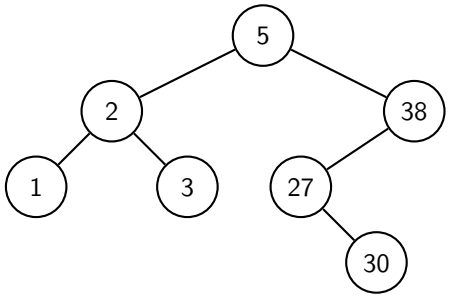


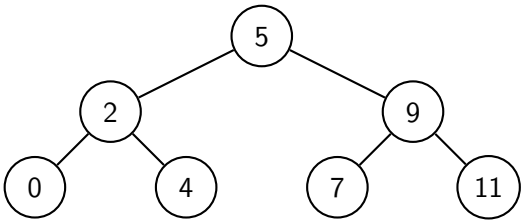
Abbildung 4.6 Fall 2: Das Entfernen vom Knoten mit dem Schlüssel 12.

Aufgabe 4.9
Entfernen Sie im folgenden Baum die Knoten in der vorgegebenen Reihenfolge: 3, 27, 30, 2, 38, 5. Notieren Sie bei jedem Knoten, den Sie entfernen, ob es sich um Fall 1 oder Fall 2 handelt.



Fall 3

Fall 3 liegt dann vor, wenn der zu entfernende Knoten Grad 2 hat. Dieser Fall ist etwas komplexer, da der zu entfernende Knoten nicht einfach durch eines seiner Kinder ersetzt werden kann. Die Gründe sind in folgender Abbildung ersichtlich. Wenn hier nun die Wurzel (Schlüssel 5) entfernt wird, kann nicht einfach das linke Kind (Wert 2) oder das rechte Kind (Wert 9) als neue Wurzel eingesetzt werden. In beiden Fällen wären die Eigenschaften eines binären Suchbaumes verletzt.



Um dieses Problem zu lösen, muss zuerst ein geeigneter Nachfolger bestimmt werden. Dieser befinden sich im rechten Teilbaum des zu entfernenden Knotens. Es ist derjenige Knoten, welcher im rechten Teilbaum am weitesten links steht (in der Abbildung von vorhin ist es der Knoten mit dem Schlüssel 7). Sobald der Nachfolgeknoten gefunden wurde, muss dieser von seinem alten Platz entfernt werden und den zu löschenden Knoten ersetzen (vgl. Abbildung 4.7).

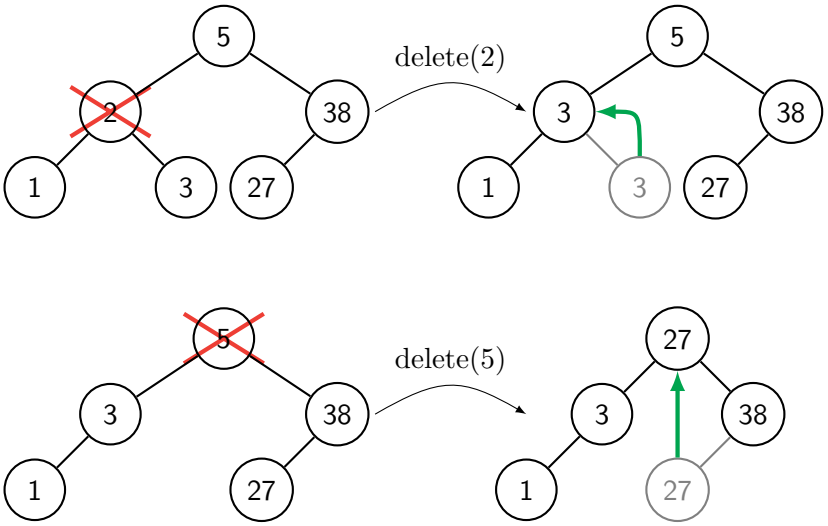
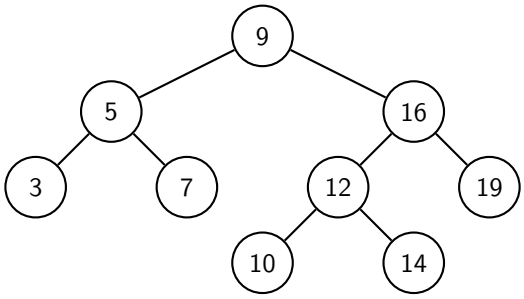


Abbildung 4.7 Fall 3: Der Knoten mit dem Schlüssel 2 wird entfernt. Danach der Knoten mit dem Schlüssel 5.

Aufgabe 4.10
Entfernen Sie im folgenden Baum die Knoten in der Reihenfolge: 5, 7, 16, 14, 9, 10. Notieren Sie bei jedem Knoten, den Sie entfernen, um welchen Fall es sich handelt.



Sie haben bereits gelesen, dass die Inorder-Reihenfolge der Knoten eines binären Suchbaumes eine aufsteigend geordnete Schlüsselreihe ergibt. Dies kann nun genutzt werden, um den Nachfolger zu bestimmen. Der Schlüssel des Nachfolgers steht in der Inorder-Reihenfolge rechts neben dem Schlüssel des zu entfernenden Knotens (vgl. Abbildung 4.8).

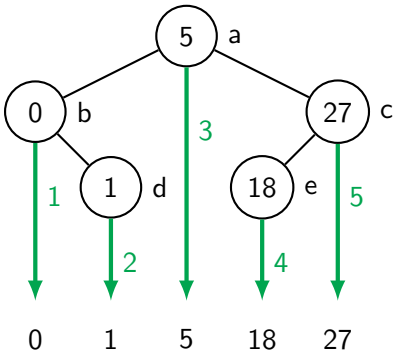
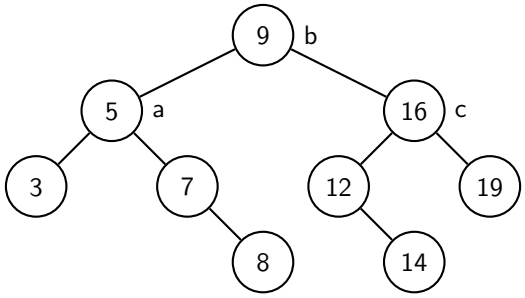
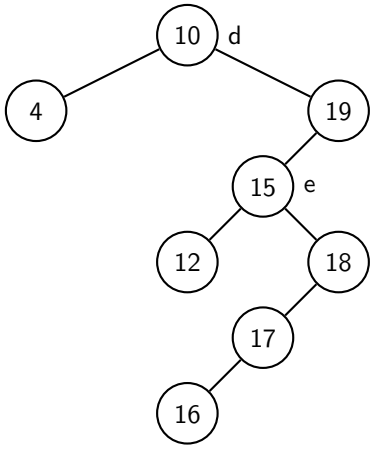


Abbildung 4.8 Der Nachfolger von Knoten a ist der Knoten e, da e in der Inorder-Reihenfolge auf a folgt.

Aufgabe 4.11
Gegeben sind die beiden folgenden binären Suchbäume. Geben Sie für den linken Baum die Inorder-Reihenfolge aus und bestimmen Sie zusätzlich die Nachfolger der Knoten a, b und c mit Hilfe der Inorder-Reihenfolge. Bestimmen Sie dann für den rechten binären Suchbaum die Nachfolger der Knoten d und e ohne Hilfe der Inorder-Reihenfolge.





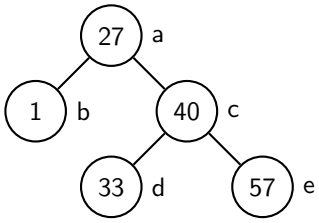
★ **Aufgabe 4.12**
Schreiben Sie den Algorithmus delete(*root*, *key*) in Pseudocode. Berücksichtigen Sie dabei alle drei Fälle. Löschen Sie alle Knoten mit dem Schlüssel *key*.

★ **Programmieraufgabe 4.13**
Implementieren Sie den Algorithmus delete(*root*, *key*) in der Klasse `BinSearchTree`.

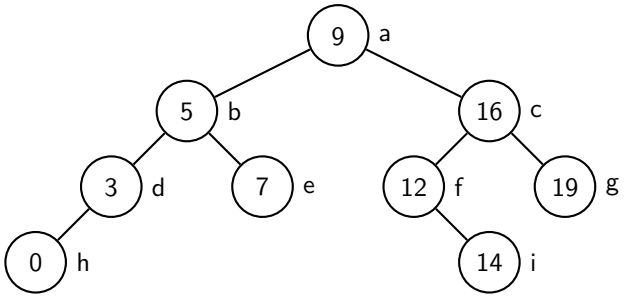
4.4 Lernkontrolle

Die folgenden Aufgaben dienen dem weiteren Üben und der Lernkontrolle der Inhalte dieses Kapitels. Wenn Sie alle Aufgaben lösen können, kann der Kapiteltest in Angriff genommen werden. Wenn jedoch Schwierigkeiten bestehen, sollten die entsprechenden Abschnitte des Kapitels nochmals bearbeitet werden.

Aufgabe 4.14
Notieren Sie für jeden Knoten den Schlüssel in der Schreibweise: $\text{key}(x) = y$.



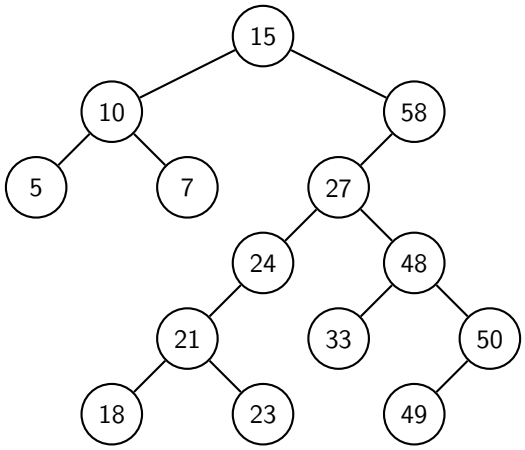
Aufgabe 4.15
Wenden Sie die Operationen search(a, 12), search(a, 9) und search(a, 15) auf den folgenden Baum an. Notieren Sie sich dabei für jede Operation die Knoten, die besucht werden. Geben Sie auch an, ob ein Knoten mit dem entsprechenden Schlüssel gefunden wird.



Aufgabe 4.16
Gegeben ist ein leerer binärer Suchbaum. Fügen Sie nun die folgenden Knoten in der gegebenen Reihenfolge ein:

Reihenfolge	Knoten	Schlüssel
1.	a	27
2.	b	7
3.	c	10
4.	d	33
5.	e	27
6.	f	31
7.	g	27
8.	h	0

Aufgabe 4.17
Gegeben ist folgender Baum:



Entfernen Sie nun die Knoten mit den folgenden Schlüsseln in der gegebenen Reihenfolge: 7, 49, 10, 15, 58, 27, 33, 18. Geben Sie auch hier zuerst an, welchen Fall Sie benutzen.

★ **Aufgabe 4.18**
In diesem Kapitel wurde der Begriff Nachfolger erklärt. Überlegen Sie sich, wie ein Vorgänger definieren werden könnte ohne die Inorder-Reihenfolge zu verwenden.

★ **Aufgabe 4.19**
In Aufgabe 4.18 haben Sie den Vorgänger definiert ohne die Hilfe der Inorder-Reihenfolge. Definieren Sie nun den Vorgänger mit Hilfe der Inorder-Reihenfolge.

5 Balancierte Suchbäume

Solange ein Suchbaum eine **ausgeglichene** Struktur hat, ist das Suchen, Einfügen und Entfernen eines Knotens effizient möglich ($O(\log n)$). Im schlechtesten Fall kann es jedoch vorkommen, dass der Suchbaum zu einer Liste entartet. Dann haben die erwähnten Operationen nicht mehr die gewünschte Effizienz. In diesem Fall gilt dann $O(n)$. Um zu verhindern, dass ein Suchbaum eine unvorteilhafte Struktur annimmt, müssen zusätzliche Bedingungen an die Struktur des Suchbaumes gestellt werden. Durch die Einhaltung dieser Bedingungen ist dann die Effizienz der Suchbäume gewährleistet.

Es gibt eine grosse Anzahl von solchen Bedingungen an die Struktur der Bäume, bei deren Einhaltung das Suchen, Einfügen und Entfernen von Knoten immer effizient möglich ist. Werden diese Bedingungen durch das Einfügen oder Entfernen eines Knotens verletzt, müssen Umstrukturierungen vorgenommen werden, um sicherzustellen, dass diese Bedingungen weiterhin gelten. Eine Art solcher sich selbst *ausbalancierenden* Suchbäume sind die **AVL-Bäume**, die in diesem Kapitel vorgestellt werden.

5.1 Lernziele

Für die effiziente Anwendung von binären Suchbäumen als Datenstruktur ist es notwendig, dass die Suchbäume ausgeglichen sind. Somit ist die leistungsfähige Struktur der binären Suchbäume über die gesamte genutzte Zeit gewährleistet.

- Sie kennen Gründe für eine Balancierung eines binären Suchbaumes.
- Sie kennen die Definition der AVL-Bäume und können diese von unausgeglichen Bäumen unterscheiden.
- Sie können für die Knoten von Binärbäumen die Balancefaktoren bestimmen.
- Sie sind in der Lage, sich selbständig in weiterführender Literatur über weitere Bedingungen zur Balancierung von Suchbäumen zu informieren.

5.2 Laufzeitbetrachtungen von binären Suchbäumen

Im vorhergehenden Kapitel hat sich gezeigt, dass die Einfügereihenfolge bei der Erzeugung eines binären Suchbaumes eine wesentliche Rolle für seine Struktur spielt (vgl. Aufgabe 4.6). Werden keine weiteren Bedingungen an die Struktur von binären Suchbäumen gestellt, so kann es im ungünstigen Fall vorkommen, dass der Baum zu einer verketten Liste entartet. Der linke binäre Suchbaum in Abbildung 5.1 entsteht zum Beispiel, wenn die Knoten in aufsteigender Reihenfolge ihrer Schlüssel in den Baum eingefügt werden.

Um in diesem Suchbaum nach dem Knoten mit Schlüssel 5 zu suchen, müssen *alle* Knoten des Baumes der Reihe nach besucht werden. Das Suchen hat somit die gleiche Laufzeit wie bei einer verketten Liste ($O(n)$).

Im Kontrast dazu hat der rechte binäre Suchbaum in Abbildung 5.1 eine ausbalancierte Struktur. Um den Knoten mit Schlüssel 5 zu finden, müssen *nur* drei Elemente besucht werden.

Laufzeit Die Anzahl der Schritte, die ein Algorithmus für die Bearbeitung benötigt.

Die Suche folgt im *schlechtesten* Fall dem Pfad von der Wurzel zum tiefsten Blatt. Somit ist die **Laufzeit** des Suchens durch die Höhe des Baumes bestimmt ($O(\log_2 n)$).

Um für binäre Suchbäume auch im schlechtesten Fall eine schnelle Laufzeit zu erhalten, muss also die Höhe des Suchbaumes möglichst klein gehalten werden. Dies kann durch eine zusätzliche Balancierung der Struktur eines Suchbaumes garantiert werden.

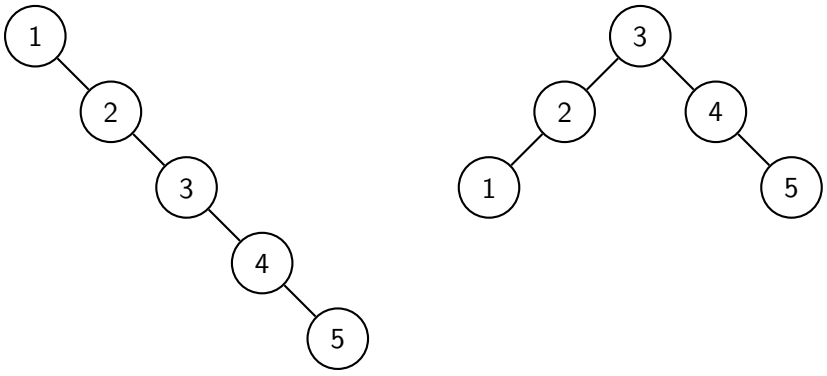


Abbildung 5.1 Der linke stark entartete Suchbaum ist durch das Einfügen der Knoten in aufsteigender Reihenfolge ihrer Schlüssel entstanden. Der rechte Suchbaum mit den gleichen Schlüsseln hat hingegen eine ausbalancierte Struktur.

5.3 AVL-Bäume

AVL-Bäume sind binäre Suchbäume, welche durch Balancierungen eine ausgeglichene Struktur garantieren. Diese Bäume werden nach ihren Erfindern Adelson-Velskij und Landis benannt. Die AVL-Bäume sind **höhenbalancierte** (height-balanced) binäre Suchbäume, das heisst die Balancierung wird erreicht, indem eine Bedingung an die Höhen von Teilbäumen gestellt wird.

AVL-Baum Ein binärer Suchbaum, bei dem sich die Höhen der Teilbäume eines Knotens um höchstens 1 unterscheiden.

Ein binärer Suchbaum ist ein **AVL-Baum** oder **AVL-ausgeglichen**, wenn für *alle* Knoten v des Baumes gilt, dass sich die Höhe des linken Teilbaumes von v *höchstens* um 1 von der Höhe des rechten Teilbaumes unterscheidet (siehe Abbildung 5.2).

Aus der Definition des AVL-Baumes folgt direkt, dass jeder Teilbaum eines AVL-Baumes ebenfalls ein AVL-Baum ist. Die Höhenbalancierung erwirkt, dass der Baum in jedem Fall eine möglichst kleine Höhe hat, womit eine effiziente Laufzeit der Suchbaum-Operationen gewährleistet ist.

Balancefaktor Die Differenz zwischen der Höhe des rechten und des linken Teilbaumes eines Knotens.

Da sich ein AVL-Baum durch das Einfügen und Entfernen von Knoten verändert, müssten nach jeder Ver alle Knoten die Höhen der beiden Teilbäume neu bestimmt werden. Dies wäre jedoch sehr aufwändig. Durch die Verwendung von **Balancefaktoren** $\text{bal}(v)$ an jedem Knoten v, kann effizienter bestimmt werden, ob ein AVL-Baum nach einer Veränderung noch ein AVL-Baum ist..

$$\begin{aligned}\text{bal}(v) = &\text{ Höhe des rechten Teilbaumes von } v \\ &- \text{Höhe des linken Teilbaumes von } v.\end{aligned}$$

Fehlende Teilbäume müssen dabei mit einer Höhe von 0 berücksichtigt werden (vgl. Abbildung 5.3).

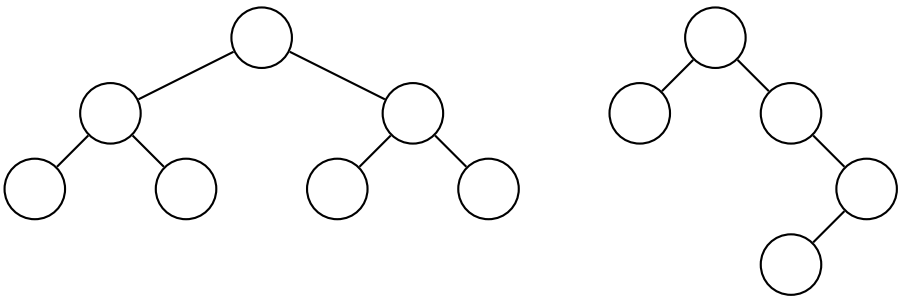


Abbildung 5.2 Der linke Binärbaum ist AVL-ausgeglichen und somit ein AVL-Baum. Der rechte Binärbaum ist hingegen *kein* AVL-Baum.

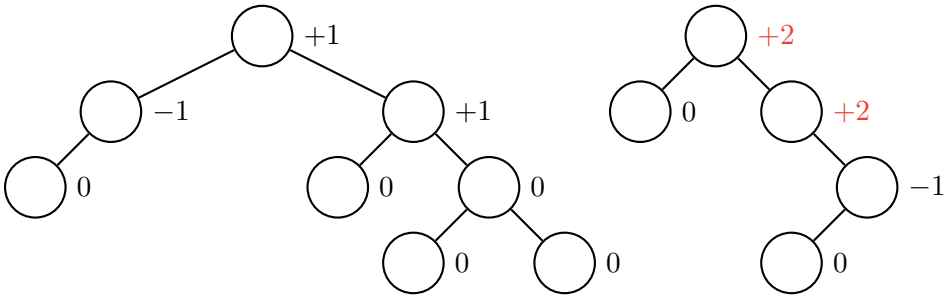


Abbildung 5.3 Zwei Binärbäume mit eingetragenen Balancefaktoren. Der linke Baum ist ein AVL-Baum, da alle Knoten ausgeglichen sind. Der Rechte ist hingegen kein AVL-Baum, da zwei Knoten unausgeglichen sind.

Ausgeglichen Ein Knoten ist ausgeglichen, wenn sein Balancefaktor -1, 0 oder $+1$ ist.

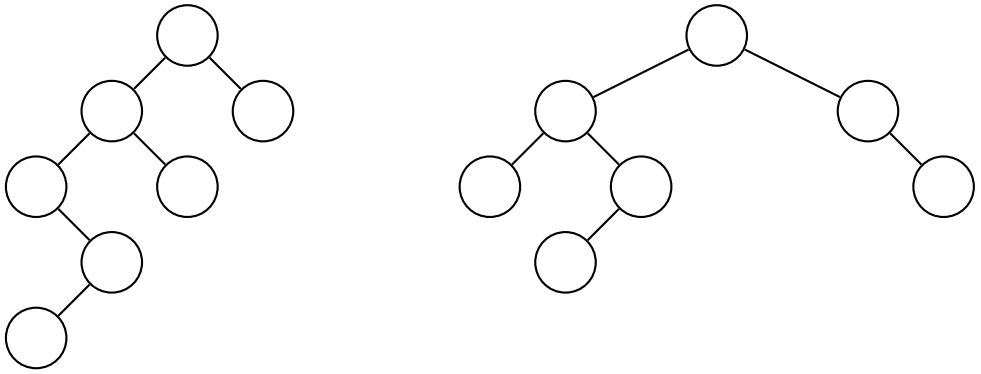
Ein Knoten v heisst **ausgeglichen** (balanced), wenn für ihn die Bedingung

$$\text{bal}(v) \in \{-1, 0, +1\}$$

erfüllt ist. Gilt die Bedingung nicht, ist der Knoten v **unausgeglichen** (unbalanced).

Ein binärer Suchbaum ist somit ein AVL-Baum, wenn *alle* Knoten des Baumes ausgeglichen sind. Durch die Balancefaktoren für jeden Knoten des AVL-Baumes müssen bei einer Veränderung der Struktur des Baumes nur diese Werte entsprechend angepasst werden.

Aufgabe 5.1
Bestimmen Sie die Balancefaktoren bal() aller Knoten der folgenden zwei Bäume. Welcher Baum ist AVL-ausgeglichen?



★ 5.4 Operationen in AVL-Bäumen

Aufgrund der zusätzlichen Balancierung ist die Umsetzung von AVL-Bäumen aufwändiger als die der binären Suchbäume. Die folgenden Abschnitte zeigen, wann die Balancierung von AVL-Bäumen notwendig wird und wie diese funktioniert.

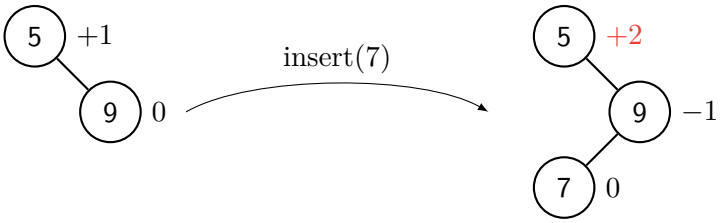
★ 5.4.1 Suchen, Einfügen und Entfernen von Knoten

Da AVL-Bäume auch binäre Suchbäume sind, werden Knoten gleich gesucht, eingefügt oder entfernt werden. Nach dem Einfügen oder Entfernen eines Knoten kann jedoch ein Suchbaum entstehen, der nicht mehr AVL-ausgeglichen ist. Für das Einfügen eines Knotens ist dies in Abbildung 5.4 dargestellt. Das gleiche kann auch beim Entfernen eines Knotens geschehen.

Nach dem Einfügen und Entfernen eines Knotens muss also möglicherweise eine Umstrukturierung des Suchbaumes stattfinden, um wieder einen AVL-Baum zu erhalten. Dies geschieht unter Berücksichtigung der folgenden Punkte:

- Es wird dem Pfad vom eingefügten bzw. entfernten Knoten bis zur Wurzel gefolgt und dabei die neuen Balancefaktoren aller besuchten Knoten bestimmt.
- Bei jedem unausgeglichenen Knoten auf dem Pfad wird eine Umstrukturierung durchgeführt.

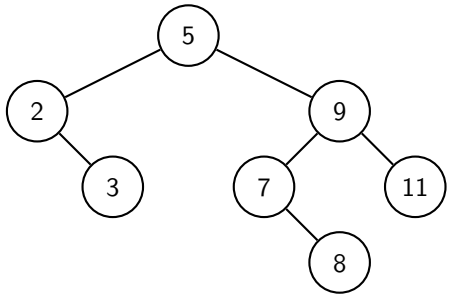
Da sich nur die Balancefaktoren der Knoten auf dem Pfad zur Wurzel geändert haben können, ist gewährleistet, dass der so umstrukturierte Baum wieder AVL-ausgeglichen ist.



★ **Abbildung 5.4** Der linke AVL-Baum wird durch das Einfügen des Knotens mit Schlüssel 7 zum unausgeglichenen rechten Suchbaum. Der rechte Suchbaum ist nicht mehr höhenbalanciert, da der Balancefaktor des Knotens mit Schlüssel 5 nun +2 ist.

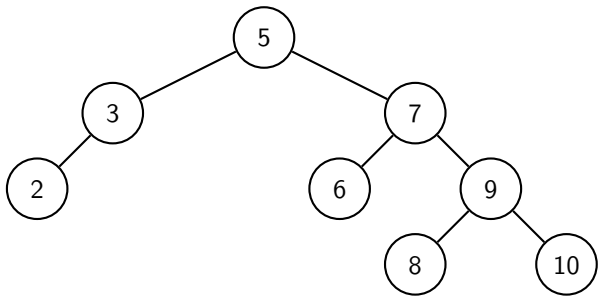
★ **Aufgabe 5.2**

Entfernen Sie den Knoten mit Schlüssel 2 aus dem folgenden AVL-Baum und berechnen Sie die Balancefaktoren des neuen Baumes. Ist der resultierende Baum noch AVL-ausgeglichen?



★ **Aufgabe 5.3**

Fügen Sie im folgenden AVL-Baum zuerst einen Knoten mit Schlüssel 4 ein. Entfernen Sie danach den Knoten mit Schlüssel 9. Bestimmen Sie nach den einzelnen Operationen jeweils die Balancefaktoren des neu enstehenden Baumes. Sind Umstrukturierungen notwendig?



★ 5.4.2 Rebalancierung durch Baumrotationen

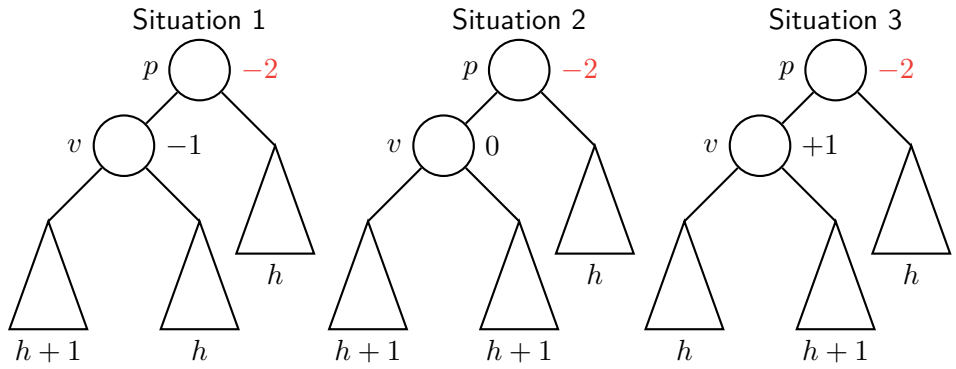
Rebalancierung Eine Umstrukturierung, bei der ein unausgeglichener Baum wieder balanciert wird.

Baumrotation Eine Methode zur Rebalancierung eines Suchbaumes, bei der die Eigenschaften des Suchbaumes erhalten bleiben.

Eine Umstrukturierung oder **Rebalancierung** eines Suchbaumes der unausgeglichen ist, kann durch Baumrotationen erreicht werden.

Eine **Baumrotation** (tree rotation) ist eine Operation auf einem Suchbaum, wleche die Struktur des balancierten Suchbaumes wieder ausbalanciert. Nach der Rotation gelten die geforderten Eigenschaften des Suchbaumes. Die Inorder-Reihenfolge der Knoten wird durch eine Baumrotation nicht verändert.

Baumrotationen sind immer dann nötig, wenn ein Knoten unausgeglichen ist. Insgesamt können die folgenden drei Situationen auftreten, in denen eine Rebalancierung notwendig ist. Zu jeder Situation gibt es zusätzlich eine spiegelbildliche Variante.



★ **Abbildung 5.5** Die drei möglichen Situationen eines unausgeglichenen Baumes.

Bei den folgenden Situationen sei p der unausgeglichene Knoten. Einer der beiden Teilbäume von p hat eine grössere Höhe als der andere. Die Wurzel dieses höheren Teilbaumes sei der Knoten v.

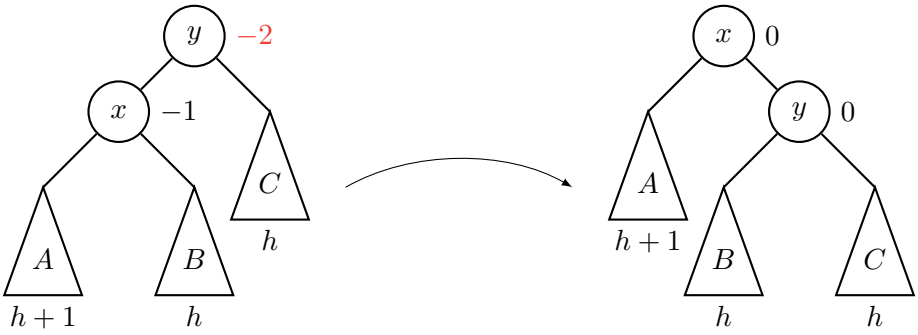
Situation 1 Die Balancefaktoren $\text{bal}(p)$ und $\text{bal}(v)$ haben das *gleiche* Vorzeichen. In Abbildung 5.5 sind beide Vorzeichen negativ. Bei der spiegelbildlichen Varianten sind beide Vorzeichen positiv.

Situation 2 Der Balancefaktor $\text{bal}(v)$ ist 0. In Abbildung 5.5 gilt $\text{bal}(p) = -2$, bei der spiegelbildlichen Variante ist $\text{bal}(p) = +2$.

Situation 3 Die Balancefaktoren $\text{bal}(p)$ und $\text{bal}(v)$ haben ein *unterschiedliches* Vorzeichen. In Abbildung 5.5 gilt $\text{bal}(p) = -2$, bei der spiegelbildlichen Variante ist $\text{bal}(p) = +2$.

Durch eine Baumrotation wird das Niveau von einigen Knoten verkleinert und von anderen vergrössert. Somit wandern erstere im Baum nach oben und letztere nach unten, um die Balance des Baumes wieder herzustellen.

Nach dem Einfügen eines Knotens kann die Balance immer mit *genau* einer Baumrotation wiederhergestellt werden. Beim Löschen können jedoch *mehrere* Baumrotationen notwendig sein, vom jeweiligen Knoten aus bis hinauf zur Wurzel.



★ **Abbildung 5.6** Eine einfache Rotation nach *rechts* um den unausgeglichenen Knoten mit Schlüssel y wieder in die Balance zu bringen (Situation 1). Neben den Balancefaktoren sind auch die Höhen der Teilbäume eingetragen.

Einfache Rotation

Entsteht nach dem Einfügen oder Entfernen eines Knotens ein unausgeglichener Knoten der Situation 1 oder 2, kann dieser Knoten durch eine einfache Rotation wieder ausgeglichen werden.

Einfache Rotation Die einfache Rotation bringt einen unausgeglichenen Knoten in Situation 1 oder 2 wieder in die Balance.

Die **einfache Rotation** nach *rechts* wird angewendet, wenn der linke Teilbaum eines Knotens im Vergleich zum rechten Teilbaum zu hoch ist. Durch eine Baumrotation nach rechts wird dieser Teilbaum angehoben und der rechte weniger hohe Teilbaum abgesenkt, um den unausgeglichenen Knoten wieder auszugleichen.

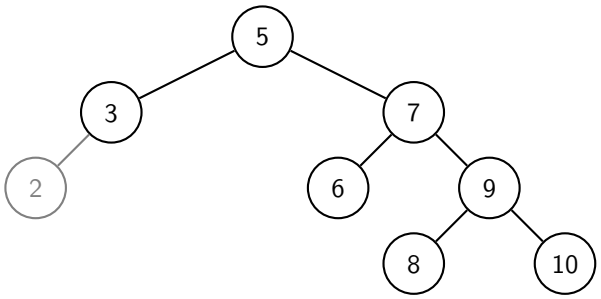
Abbildung 5.6 zeigt den Vorgang der einfachen Rotation nach rechts an einem unausgeglichenen Knoten mit Schlüssel y. Der hohe Teilbaum A und der Knoten mit Schlüssel x werden durch die Rechtsrotation um ein Niveau angehoben, während der Teilbaum C und der Knoten mit Schlüssel y abgesenkt werden. Im resultierenden Baum ist die Balance wieder hergestellt.

Bei der einfachen Rotation nach *links* findet die Baumrotation in der entgegengesetzten Richtung statt. Dabei wird der rechte Teilbaum angehoben und der linke Teilbaum abgesenkt.

★ **Aufgabe 5.4**
Zeichnen Sie eine Grafik analog zur Abbildung 5.6 die eine einfache Rotation nach links illustriert.

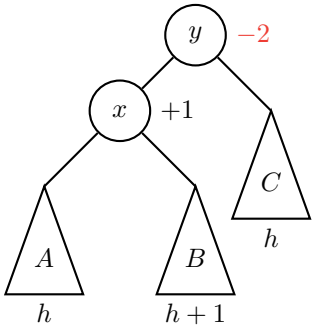
★ **Aufgabe 5.5**

Der folgende unausgeglichene Suchbaum ist durch Entfernen des Knotens mit Schlüssel 2 entstanden. Zeichnen Sie die Balancefaktoren ein und Führen Sie auf dem Suchbaum die notwendige einfache Rotation aus, um ihn wieder auszugleichen.



★ **Aufgabe 5.6**

Bei folgendem Baum liegt die Situation 3 vor. Die Balancefaktoren der Knoten mit den Schlüsseln x und y haben also unterschiedliche Vorzeichen. In diesem Fall reicht eine einfache Baumrotation nicht aus, um den Baum wieder auszugleichen. Zeigen Sie dies, indem Sie eine einfache Rotation nach rechts ausführen und danach prüfen ob der resultierende Suchbaum ausgeglichen ist.



Doppelrotation

Tritt die Situation 3 auf, reicht eine einfache Rotation nicht aus, um den unausgeglichenen Knoten zu balancieren (vgl. Aufgabe 5.6). Denn eine einfache Rotation würde das Ungleichgewicht nur auf die andere Seite verlagern. Es ist deshalb eine Doppelrotation nötig.

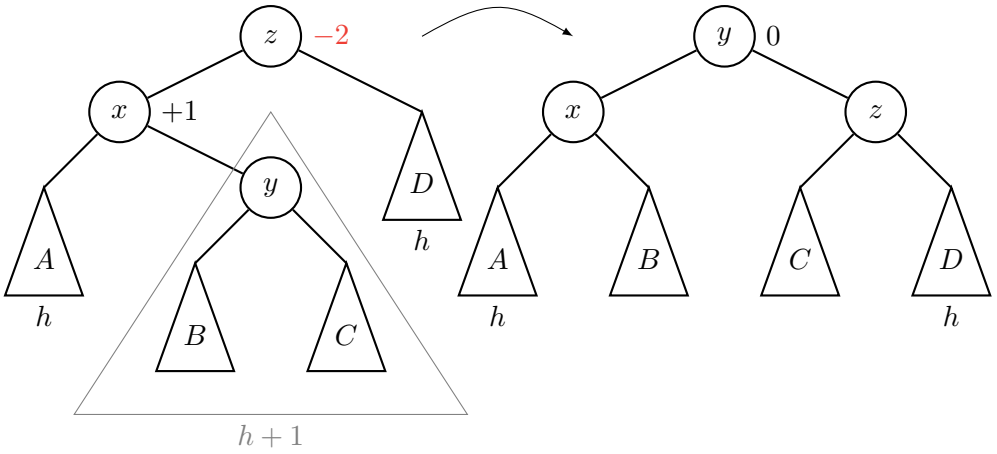
Doppelrotation Die Doppelrotation bringt einen unausgeglichenen Knoten in Situation 3 wieder in die Balance.

Durch eine **Doppelrotation** *links-rechts* wird der Baum in Abbildung 5.7, der sich in Situation 3 befindet, ausgeglichen. Der mittlere Teilbaum mit Wurzel y (grau) ist im Vergleich zum Teilbaum D des Knotens z zu hoch. Die Doppelrotation hebt den Knoten y um zwei Niveaus nach oben und er wird somit Vaterknoten von x und z. Der Teilbaum D wird dabei abgesenkt. Die beiden Teilbäume B und C werden zu rechten bzw. linken Teilbäumen der Knoten x und z. Im resultierenden Baum ist die Balance wieder hergestellt.

Die Doppelrotation *rechts-links* ist die spiegelbildliche Variante bei der die mittleren Teilbäume ebenfalls angehoben werden, dafür aber der linke Teilbaum absenkt wird.

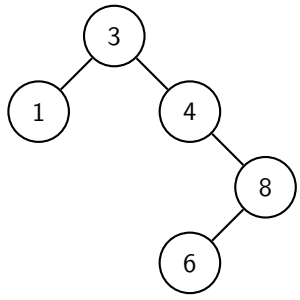
★ **Aufgabe 5.7**
Zeichnen Sie eine Grafik analog zur Abbildung 5.7, die eine Doppelrotation *rechts-links* illustriert.

★ **Aufgabe 5.8**
Der folgende unausgeglichene Suchbaum ist durch das Einfügen des Knotens mit Schlüssel 6 entstanden. Führen Sie auf dem Suchbaum die notwendige Doppelrotation



★ **Abbildung 5.7** Eine **Doppelrotation** *links-rechts* um den unausgeglichenen Knoten mit Schlüssel z wieder in die Balance zu bringen (Situation 3). Neben den Balancefaktoren sind auch die relevanten Höhen der Teilbäume eingetragen.

aus um ihn wieder auszugleichen.



⋆ 5.5 Implementierung von AVL-Bäumen

Dieser Abschnitt beschreibt zusätzliche Hinweise und Algorithmen, die für eine Implementierung eines AVL-Baumes benötigt werden. Daher richtet er sich an die schnellen und geübten Leser und Leserinnen. Das Ziel dieses Abschnitts ist die vollständige Implementierung eines AVL-Baumes in Java.

Die nachfolgenden Beschreibungen stellen eine Skizze dar und müssen ergänzt werden, um eine vollständige Anleitung zu erhalten. Auch sind für die Aufgaben dieses Abschnitts keine Lösungen gegeben. Es sind daher weitere Nachforschungen in zusätzlicher Literatur notwendig.

Der folgende Algorithmus zeigt eine mögliche Implementierung einer einfachen Rotation nach rechts in Pseudocode.

1: **algorithm** rightrotation(p)
2: $v \leftarrow \text{left}(p)$
3: Vertausche die Inhalte und Schlüssel von p mit v
4: $\text{left}(p) \leftarrow \text{left}(v)$
5: $\text{left}(v) \leftarrow \text{right}(v)$
6: $\text{right}(v) \leftarrow \text{right}(p)$
7: $\text{right}(p) \leftarrow v$

8: $bv \leftarrow 1 + \max(-\text{bal}(v), 0) + \text{bal}(p)$ $\{*\}$
9: $bp \leftarrow 1 + \text{bal}(v) + \max(0, bv)$ $\{*\}$
10: $\text{bal}(p) \leftarrow bp$
11: $\text{bal}(v) \leftarrow bv$
12: **end algorithm**

Die beiden mit $\{*\}$ markierten Zeilen berechnen die neuen Balancefaktoren der Knoten *nur* anhand der vorherigen Balancefaktoren. Bei der Funktion leftrotation(p) lauten die beiden Zeilen daher anders:

1: **algorithm** leftrotation(p)
2: $v \leftarrow \text{right}(p)$
3: Vertausche die Inhalte und Schlüssel von p mit v
4: Rotiere die Knoten und Teilbäume entsprechend
5: $bv \leftarrow -1 - \max(\text{bal}(v), 0) + \text{bal}(p)$
6: $bp \leftarrow -1 + \text{bal}(v) - \max(0, -bv)$
7: $\text{bal}(p) \leftarrow bp$
8: $\text{bal}(v) \leftarrow bv$
9: **end algorithm**

Das Einfügen und Entfernen von Knoten erfolgt wie im Abschnitt 5.4.1 erläutert. Es wird bewusst kein Pseudocode vorgegeben.

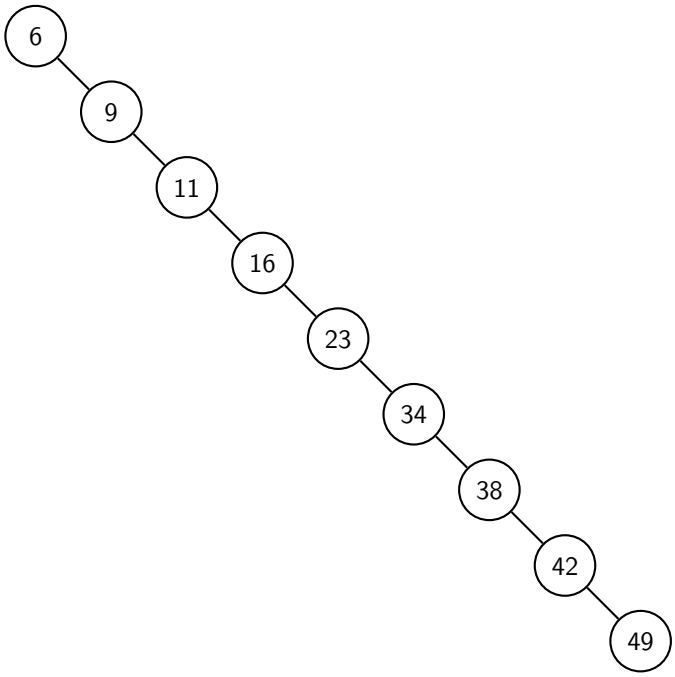
★ **Programmieraufgabe 5.9**
Vervollständigen Sie die beiden Methoden `rightrotation()`, `leftrotation()` in der gegebenen Klasse `AVLNode`. Diese beiden Methoden sollen die einfachen Baumrotationen gemäss dem Pseudocode durchführen.

★ **Programmieraufgabe 5.10**
Erweitern Sie die Klasse `AVLNode` so, dass nach dem Einfügen oder Entfernen eines Knotens, die unausgeglichenen Knoten mit Hilfe von Baumrotationen ausbalanciert werden. Nutzen Sie dabei die Erläuterungen im Abschnitt 5.4.1 auf Seite 51.

5.6 Lernkontrolle

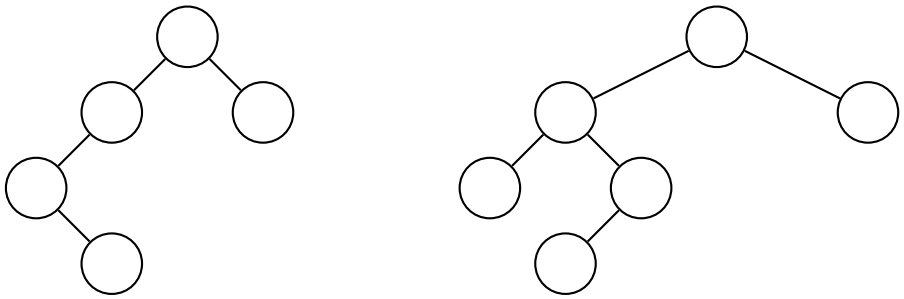
Die folgenden Aufgaben dienen dem weiteren Üben und der Lernkontrolle der Inhalte dieses Kapitels. Wenn Sie alle Aufgaben lösen können, kann der letzte Kapiteltest gemacht werden. Wenn jedoch Schwierigkeiten bestehen, sollten die entsprechenden Abschnitte des Kapitels nochmals bearbeitet werden.

Aufgabe 5.11
Der folgende binäre Suchbaum ist zu einer verketten Liste entartet.

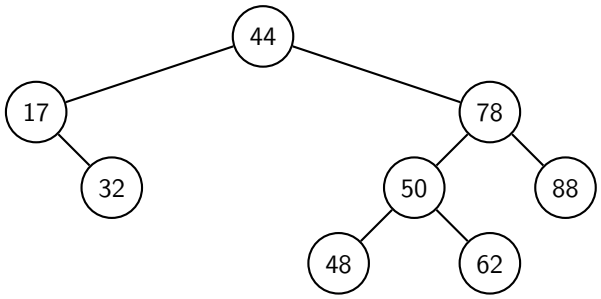


a) Wie viele Schritte werden im *schlechtesten* Fall für die Suche eines Knotens benötigt?
b) Zeichnen Sie einen binären Suchbaum mit den gleichen Schlüsseln der möglichst ausgeglichen ist. Wie viele Schritte werden nun im schlechtesten Fall für die Suche benötigt?

Aufgabe 5.12
Zeichnen Sie bei den folgenden Binärbäumen die Balancefaktoren aller Knoten ein und bestimmen Sie, ob diese ausgeglichen sind.



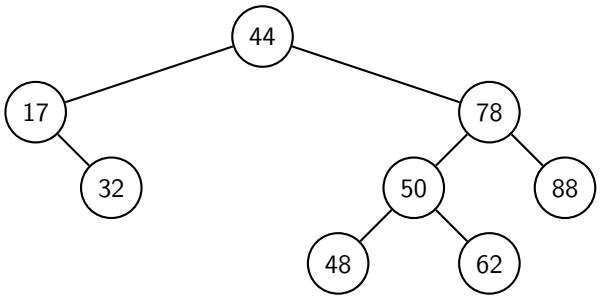
Aufgabe 5.13
Bestimmen Sie die Balancefaktoren aller Knoten des folgenden binären Suchbaumes. Handelt es sich um einen AVL-Baum?



★ **Aufgabe 5.14**
Schreiben Sie eine Funktion balanced(v) in Pseudocode, die **true** zurück gibt, falls der Binärbaum mit Wurzel v höhenbalanciert ist. Die Funktion height($\cdot$) darf dabei verwendet werden.

★ **Aufgabe 5.15**

Entfernen Sie den Knoten mit Schlüssel 78 aus dem folgenden Suchbaum und führen Sie danach die notwendigen Rebalancierungen durch, um wieder einen AVL-Baum zu erhalten.



★ **Aufgabe 5.16**

Zeichnen Sie den AVL-Baum, der durch Einfügen der Schlüssel

$$9, 14, 12, 4, 8, 6, 3, 1, 13$$

der Reihe nach in den anfangs leeren Baum entsteht. Führen Sie dabei die benötigten Rebalancierungen durch.

6 Zusammenfassung

In diesem Leitprogramm wurden die wichtigsten Elemente von Bäumen, wie sie in der Informatik verwendet werden, aufgezeigt. Sie haben somit alle in der Einleitung aufgelisteten Lernziele erreicht.

Bäume sind ein mächtiges Konzept in der Informatik und werden in vielen Gebieten eingesetzt. Für besondere Zwecke wurden unter anderem spezialisierte Bäume entwickelt. Darunter gehören zum Beispiel die abstrakten Syntaxbäume, die in Aufgabe 3.9 vorgestellt wurden. In der Algorithmischen Geometrie oder Computergrafik sind die Quad- und Oct-trees verbreitet. Diese sind, wie der Name vermuten lässt, Bäume der Ordnung 4 bzw. 8. Eine weitere verbreitete Form von Bäumen sind die B-Bäume, die bei der Implementierung von Datenbanken eingesetzt werden.

6.1 Weiterführende Literatur

Die interessierten Leser und Leserinnen werden auf die Bücher von Ottmann und Widmayer (2002) und Goodrich und Tamassia (2004) verwiesen. Auch auf der Online-Enzyklopädie Wikipedia (2006) finden sich viele weitere Anwendungen von Bäumen. Ausserdem haben die Autoren Sedgewick und Schidlowsky (2003a,b,c) drei Bücher herausgegeben, welche die Programmiersprachen Java, C++ und C abdecken.

Literaturverzeichnis

Class, Christina (2006). “Modul Programmierung II”.
SW3 – DAT, HTA Luzern.

Goodrich, Michael T. und Roberto Tamassia (2004).
Data Structures and Algorithms in JAVA. 3. Auflage.
Hoboken: John Wiley & Sons, Inc.

Ottmann, Thomas und Peter Widmayer (2002).
Algorithmen und Datenstrukturen. 4. Auflage.
Heidelberg, Berlin: Spektrum, Akadademischer Verlag.

Sedgewick, Robert und Michael Schidlowsky (2003*a*). *Algorithms in C++, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*.
Addison Wesley Professional.

Sedgewick, Robert und Michael Schidlowsky (2003*b*). *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*.
Addison Wesley Professional.

Sedgewick, Robert und Michael Schidlowsky (2003*c*). *Algorithms in Java, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*.
Addison Wesley Professional.

Wikipedia (2006). *Binary search tree — Wikipedia, The Free Encyclopedia*.
URL: `http://en.wikipedia.org/w/index.php?title=Binary_search_tree&oldid=74720615` (besucht am 19. 09. 2006).

Index

A Lösungen zu den Aufgaben

In diesem Anhang sind die Musterlösungen zu den Aufgaben dieses Leitprogramms aufgeführt. Diese sind für die Selbstkontrolle gedacht.

A.1 Lösungen zum Kapitel 2

Lösung zur Aufgabe 2.1
Die Knoten der Bäume haben folgende Eigenschaften:

Wurzel Knoten a und b.

Blatt Knoten a, e, f, g und h sind Blätter, da sie keine Kinder haben.

Innerer Knoten Knoten b, c und d, da sie mindestens ein Kind haben.

Lösung zur Aufgabe 2.2
Die Bäume und ihre Knoten haben folgende Masse:

a) Die Höhe beträgt im linken Baum 1 und im rechten Baum 3.
b) Der Baum links kann jede beliebige Ordnung haben. Baum rechts hingegen muss mindestens Ordnung 3 besitzen.
c) Das Niveau sehen Sie unten in der Tabelle.
d) Der linke Baum besteht nur aus der Wurzel und hat damit keine inneren Knoten. Damit ist er ausgefüllt. Der rechte Baum ist nicht ausgefüllt (die Wurzel hat nur zwei Kinder, die Ordnung des Baumes ist aber mind. 3).
e) Der Grad und die Tiefe der Knoten sehen Sie in der folgenden Tabelle:

Niveau	Knoten	Grad	Tiefe
1	a	0	1
	b	2	
2	c	3	2
	d	1	
3	e	0	3
	f	0	
	g	0	
	h	0	

Lösung zur Aufgabe 2.3
Speziell zu erwähnen ist hier die Wurzel, welche sowohl vom Typ Wurzel als auch vom Typ innerer Knoten ist. Bedenken Sie, eine Wurzel ohne Kinder vom Typ Blatt ist.

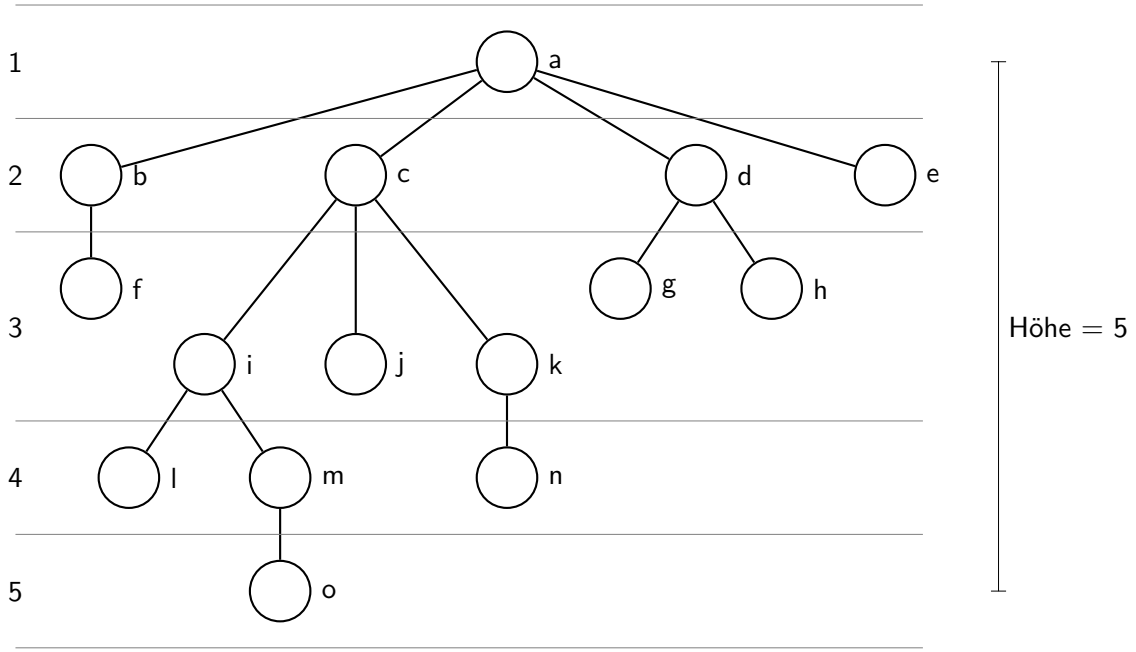
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Wurzel	x														
Blatt					x	x	x	x		x		x		x	x
Innerer Knoten	x	x	x	x					x		x		x		

Lösung zur Aufgabe 2.4

Knoten	Grad	Tiefe
a	4	1
b	1	2
c	3	
d	2	
e	0	
f	0	3
g	0	
h	0	
i	2	
j	0	
k	1	
l	0	4
m	1	
n	0	
o	0	5

Lösung zur Aufgabe 2.5
Aus Aufgabe 2.4 sehen wir, dass der Knoten mit dem höchsten Grad Knoten a ist. Dieser hat Grad 4. Somit muss dieser Baum eine Ordnung ≥ 4 haben.

Unten sehen Sie den Baum mit den Niveaus auf der linken Seite. Rechts steht die Höhe.



Lösung zur Aufgabe 2.6
Der Baum ganz links besteht nur aus der Wurzel. Hier ist die Wurzel vom Typ Blatt. Damit ist der Baum ausgefüllt, da es keine inneren Knoten mit Grad kleiner 2 gibt. Dieser Baum ist auch vollständig. Der mittlere Baum ist wiederum ausgefüllt, da auch hier keine inneren Knoten existieren, welche zu wenig Kinder haben. Dieser Baum ist jedoch nicht vollständig, da es auf dem Niveau 3 nur 2 Knoten hat (die maximale Anzahl Knoten auf diesem Niveau ist 4). Der dritte und letzte Baum ist nicht ausgefüllt, da die beiden Knoten mit Tiefe 1 nicht genug Kinder haben. Daraus folgt auch, dass dieser Baum nicht vollständig ist (nur ein ausgefüllter Baum kann vollständig sein).

Lösung zur Aufgabe 2.7
Für diese Aufgabe erhalten Sie hier keine Lösung. Besprechen Sie die Ergebnisse Ihrer Recherche doch einfach mit einem/einer Mitstudierenden oder mit dem Dozenten oder der Dozentin.

A.2 Lösungen zum Kapitel 3

Lösung zur Aufgabe 3.1
Die Eigenschaft, dass ein Knoten des Binärbaumes nur zwei Kinder haben kann, ist für die Durchlaufordnung in der Preorder-Reihenfolge nicht relevant.

Der Algorithmus preorder() kann daher folgendermassen erweitert werden.

```
1: algorithm preorder(v)
2:     {Durchläuft alle Knoten des Baumes mit Wurzel v in der Preorder-
   Reihenfolge}
3:     if v ≠ null then
4:         Besuche den Knoten v
5:         for all c ← Kind von v do
6:             preorder( c )
7:         end for
8:     end if
9: end algorithm
```

Lösung zur Aufgabe 3.2
Die Durchlaufordnung nach der Preorder-Reihenfolge ist:
a, b, d, e, g, c, f, h, j, i.

Lösung zur Aufgabe 3.4
Die Durchlaufordnung nach der Postorder-Reihenfolge ist:
d, g, e, b, j, h, i, f, c, a.

Lösung zur Aufgabe 3.5

```
algorithm postorder(v)
    {Durchläuft alle Knoten des Binärbaumes mit Wurzel v in der Postorder-
Reihenfolge}
    if v ≠ null then
        postorder( left(v) )
        postorder( right(v) )
        Besuche den Knoten v
    end if
end algorithm
```

Lösung zur Aufgabe 3.6
Der Algorithmus inorderNext(v) für einen Knoten v sieht folgendermassen aus:

```
1: algorithm inorderNext(v)
2:     Require right(v) ≠ null
3:     v ← right(v)
```

```
4:     while left(v) ≠ null do
5:         v ← left(v)
6:     end while
7:     return v
8: end algorithm
```

Lösung zur Aufgabe 3.7
Die Durchlaufordnung nach der Inorder-Reihenfolge ist:
d, b, g, e, a, c, h, j, f, i.

Lösung zur Aufgabe 3.8
Die Preorder-Reihenfolge ist: a, b, d, g, h, k, m, c, e, f, i, j, l.
Die Postorder-Reihenfolge ist: g, m, k, h, d, b, e, i, l, j, f, c, a.
Und schliesslich die Inorder-Reihenfolge: g, d, h, m, k, b, a, e, c, i, f, j, l.

Lösung zur Aufgabe 3.9
Die Durchlaufen nach der Inorder-Reihenfolge führt zur richtigen Ausgabe des Ausdrucks.

Der mathematische Ausdruck lautet:

$$((5-3)+2) \times ((8+6) \div 4)\,.$$

Lösung zur Aufgabe 3.10
Um den verwendeten Speicherplatz eines Verzeichnisses zu berechnen, muss zuerst der benötige Speicher für alle Unterverzeichnisse und Dateien des Verzeichnisses bekannt sein. Der notwendige Speicherplatz wird berechnet, indem bei den Blättern des Baumes begonnen und sukzessive zur Wurzel vorgearbeitet wird. Dies kann auf elegante Weise mit einer Traversierung nach der Postorder-Reihenfolge erreicht werden.

Der benötigte Speicherplatz beträgt: 479 kbyte.

Lösung zur Aufgabe 3.11
a) Die Anzahl der Blätter im Binärbaum B beträgt:

$$2^{h-1}$$

b) Die Anzahl der Knoten im Binärbaum B ist:

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

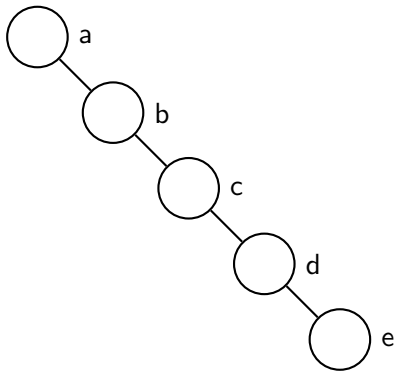
Lösung zur Aufgabe 3.12

a) Es ist nicht möglich, dass die Preorder- der Postorder-Reihenfolge entspricht. Da der Baum B mindestens zwei Knoten hat, gibt es neben der Wurzel a einen weiteren Knoten v.

Angenommen die Preorder-Reihenfolge der Knoten des Baumes lautet:
$a, \ldots, v, \ldots$

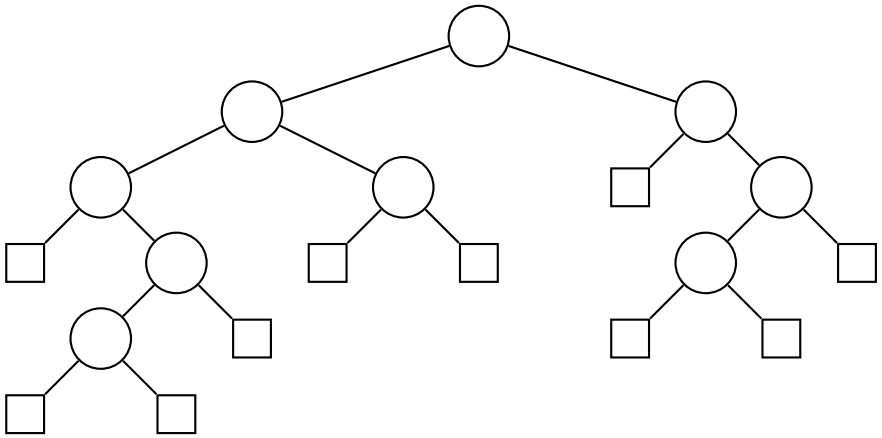
Die entsprechende Postorder-Reihenfolge
$\ldots, v, \ldots, a$
kann deshalb niemals die gleiche sein.

b) Es ist möglich, dass die Reihenfolge der besuchten Knoten bei der Preorder-- Reihenfolge der umgekehrten Reihenfolge der besuchten Knoten bei Postorder entspricht. Dies gilt zum Beispiel für den folgenden Baum:



Lösung zur Aufgabe 3.13

Der erweiterte Binärbaum nach der alternativen Definition für Binärbäume sieht so aus:



A.3 Lösungen zum Kapitel 4

Lösung zur Aufgabe 4.1
key(a) = 4, key(b) = 2, key(e) = 3, key(g) = 1 und key(i) = 5.

Lösung zur Aufgabe 4.2
Im Prinzip kann man fast alles als Schlüssel wählen. Eine mögliche Lösung:

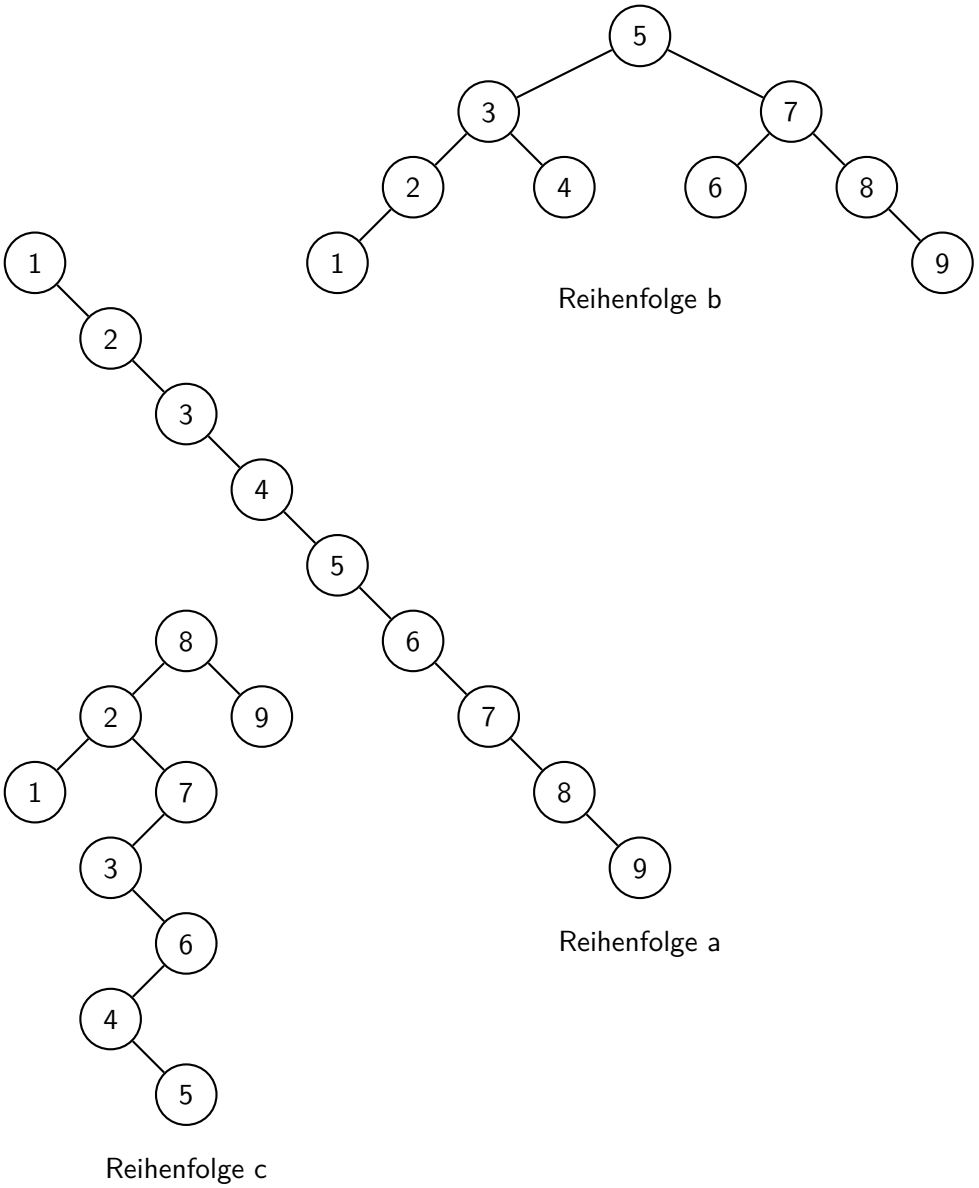
- Personen: Hier wäre eine AHV-Nummer das Ideale. Oder Nachname und Vorname welche dann alphabetisch geordnet werden (was nicht gerade für die Performance spricht).
- Bücher: Die ISBN-Nummer.
- Adressen: Man könnte hier einen Hash-Wert über den Strassennamen und Strassennummer berechnen und diesen dann als Schlüssel verwenden.

Sie sehen, dass es hier darum geht, einen möglichst eindeutigen und schnellen Schlüssel zu finden. Ausserdem muss unbedingt darauf geachtet werden, dass der gewählte Schlüssel keine änderungen erfährt, da sonst die Ordnung nicht mehr stimmt. Hierfür schlechte Beispiele wären: Alter einer Person, aktuelle Lagermengen eines bestimmten Produkts, usw.

Lösung zur Aufgabe 4.4
Die gewünschte Änderung finden Sie in der Zeile 13.

```
1:  algorithm search(v, k)
2:      {Im Baum mit Wurzel v werden alle Knoten mit Schlüssel k gesucht}
3:      if v ≠ null then
4:          if k < key(v) then
5:              {Suche im linken Teilbaum left(v)}
6:              search(left(v), k)
7:          else
8:              if k > key(v) then
9:                  {Suche im rechten Teilbaum right(v)}
10:                 search(right(v), k)
11:             else
12:                 Suche erfolgreich
13:                 search(right(v), k) {Rechts nachsehen, ob noch ein weiterer
    gleicher Schlüssel vorhanden ist}
14:                 Beende Suche {Suche war erfolgreich}
15:             end if
16:         end if
17:     else
18:         Beende Suche {Suche war erfolglos}
19:     end if
20: end algorithm
```

Lösung zur Aufgabe 4.6
Im folgenden Bild sind die gesuchten Bäume:



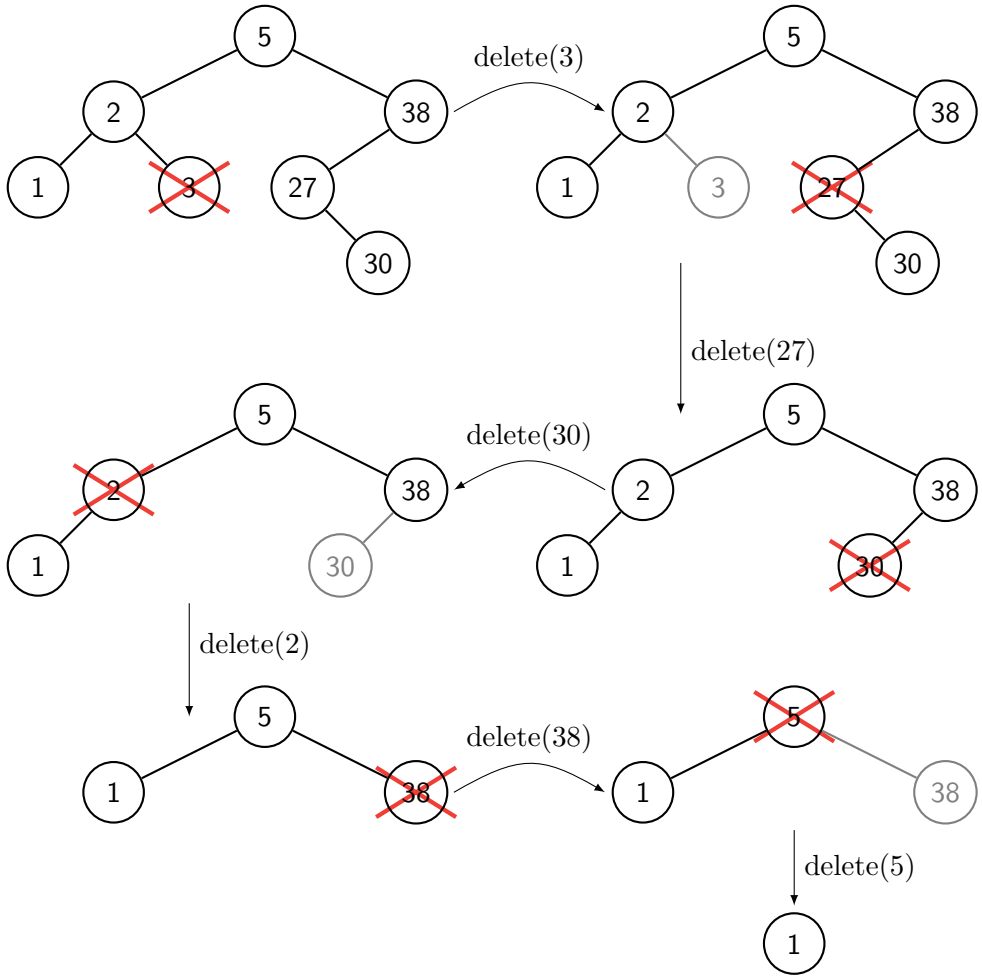
Lösung zur Aufgabe 4.7
Es gibt mehrere Reihenfolgen, welche den gleichen binären Suchbaum erzeugen würden. Zwei mögliche Lösungen sind:

1. Möglichkeit: 5, 3, 4, 2, 1, 7, 6, 8, 9

2. Möglichkeit: 5, 7, 8, 9, 6, 3, 2, 1, 4

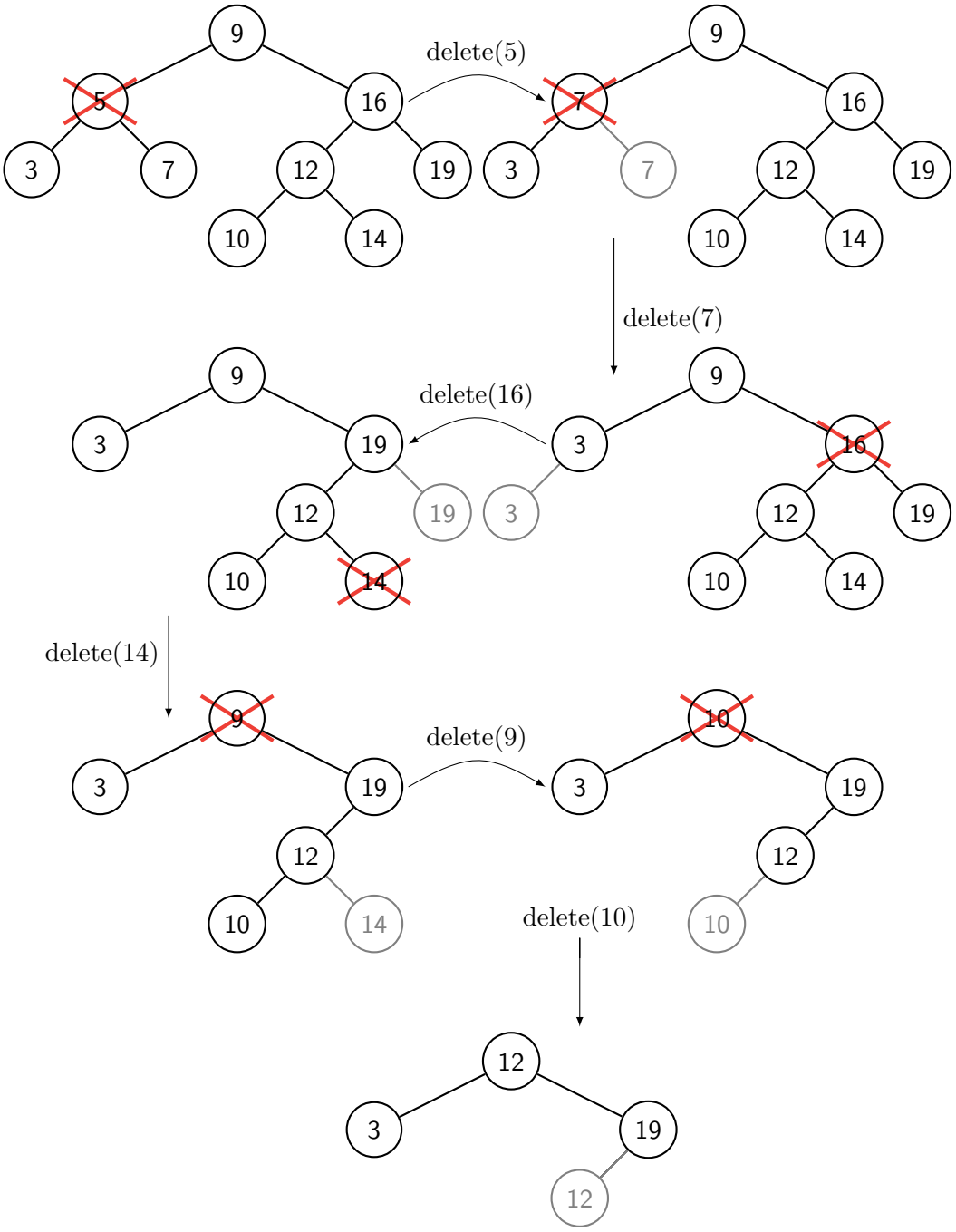
Lösung zur Aufgabe 4.9
Welcher Fall beim Entfernen angewendet wurde, entnehmen Sie der Tabelle. Die Bäume unten zeigen auf, wie der Baum nach jedem Schritt aussieht.

Knoten	Fall
3	1
27	2
30	1
2	2
38	1
5	2



Lösung zur Aufgabe 4.10
Welcher Fall beim Entfernen angewendet wurde, entnehmen Sie der Tabelle. Die Bäume unten zeigen auf, wie der Baum nach jedem Schritt aussieht.

Knoten	Fall
5	3
7	2
16	3
14	1
9	3
10	3



Lösung zur Aufgabe 4.11

- Baum links:

  - Die Inorder-Reihenfolge lautet: 3, 5, 7, 8, 9, 12, 14, 16, 19.
  - Der Inorder-Nachfolger von a ist 7, der von b ist 12 und der von c ist die 19.

- Baum rechts: Der Inorder-Nachfolger von d ist die 12 und von e ist die 16.

Lösung zur Aufgabe 4.12
In den Zeilen 24-27 wurde nun auch Fall 3 berücksichtigt.

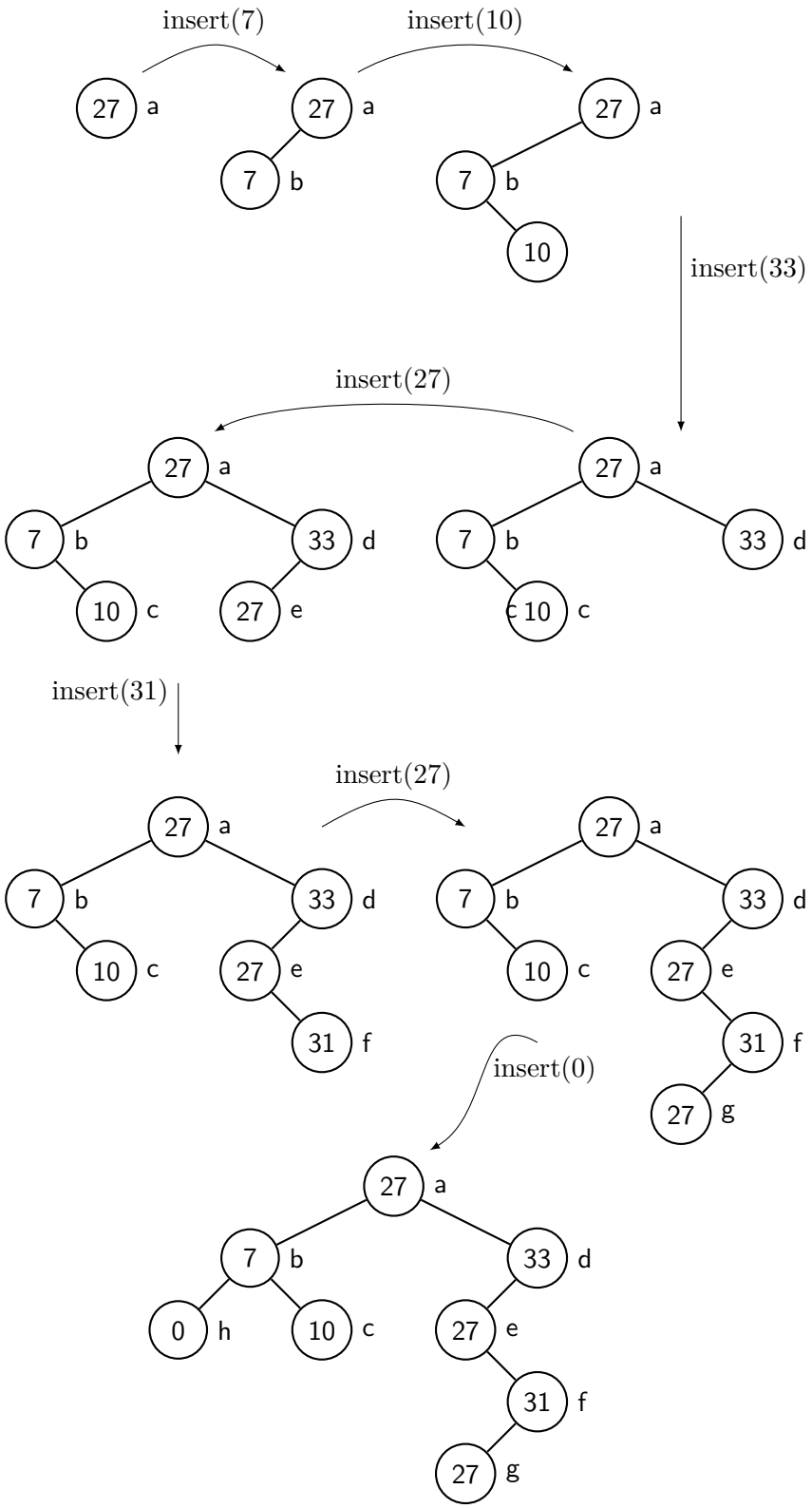
```
1: algorithm delete(v, k)
2:     if v ≠ null then
3:         if k < key(v) then
4:             delete(left(v),k) {Knoten muss im linken Teilbaum left(v) gesucht und
   entfernt werden}
5:         else
6:             if k > key(v) then
7:                 delete(right(v), k) {Knoten muss im rechten Teilbaum right(v) ge-
   sucht und entfernt werden}
8:             else
9:                 {Knoten v gefunden}
10:                Suche erfolgreich
11:                if left(v) = null then
12:                    if right(v) = null then
13:                        {Fall 1 Der Knoten zum Entfernen ist ein Blatt}
14:                        Entferne den Knoten v
15:                    else
16:                        {Fall 2 Der zu entfernende Knoten ist ein innerer Knoten
   mit einem Kind rechts}
17:                        v = right(v) {Ersetze Knoten v mit seinem Kind}
18:                    end if
19:                else
20:                    if right(v) = null then
21:                        {Fall 2 Der zu entfernende Knoten ist ein innerer Knoten
   mit einem Kind links}
22:                        v = left(v) {Ersetze Knoten v mit seinem Kind}
23:                    else
24:                        {Fall 3 Der zu entfernende Knoten ist ein innerer Knoten
   mit zwei Kindern}
25:                        Suche Nachfolger
26:                        Lösche Nachfolger von seiner alten Position
27:                        Ersetze den zu entfernenden Knoten mit seinem Nachfolger
28:                    end if
29:                end if
30:                delete(v,k) {Suche im verbleibenden Baum weitere Vorkommen zum
   Löschen}
31:            end if
32:        end if
33:    end if
34: end algorithm
```

Lösung zur Aufgabe 4.14
key(a) = 27, key(b) = 1, key(c) = 40, key(d) = 33 und key(e) = 57.

Lösung zur Aufgabe 4.15

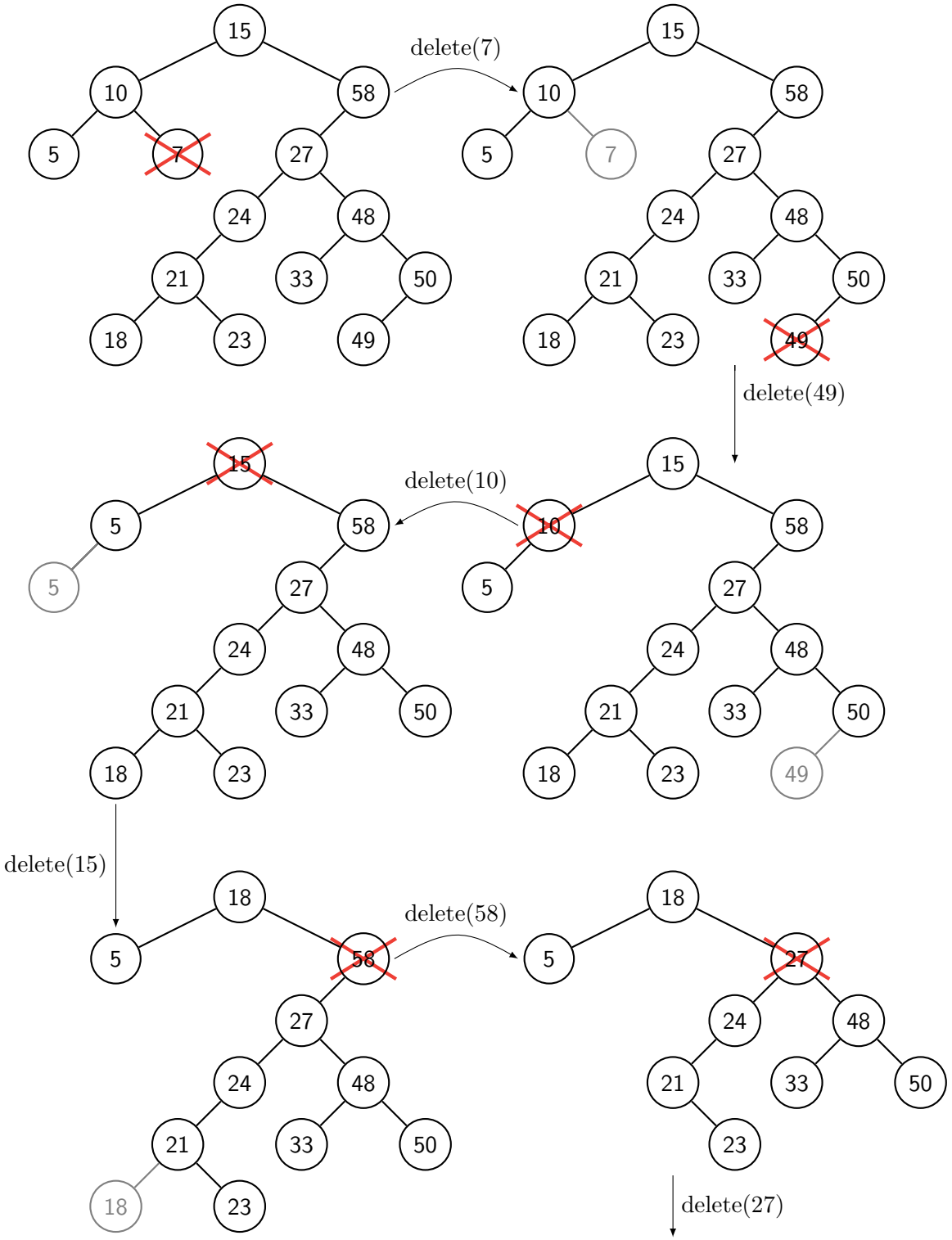
Aufruf	Besuchte Knoten	Erfolgreich?
search(a, 12)	a, c, f	Ja
search(a, 9)	a	Ja
search(a, 15)	a, c, f, i	Nein

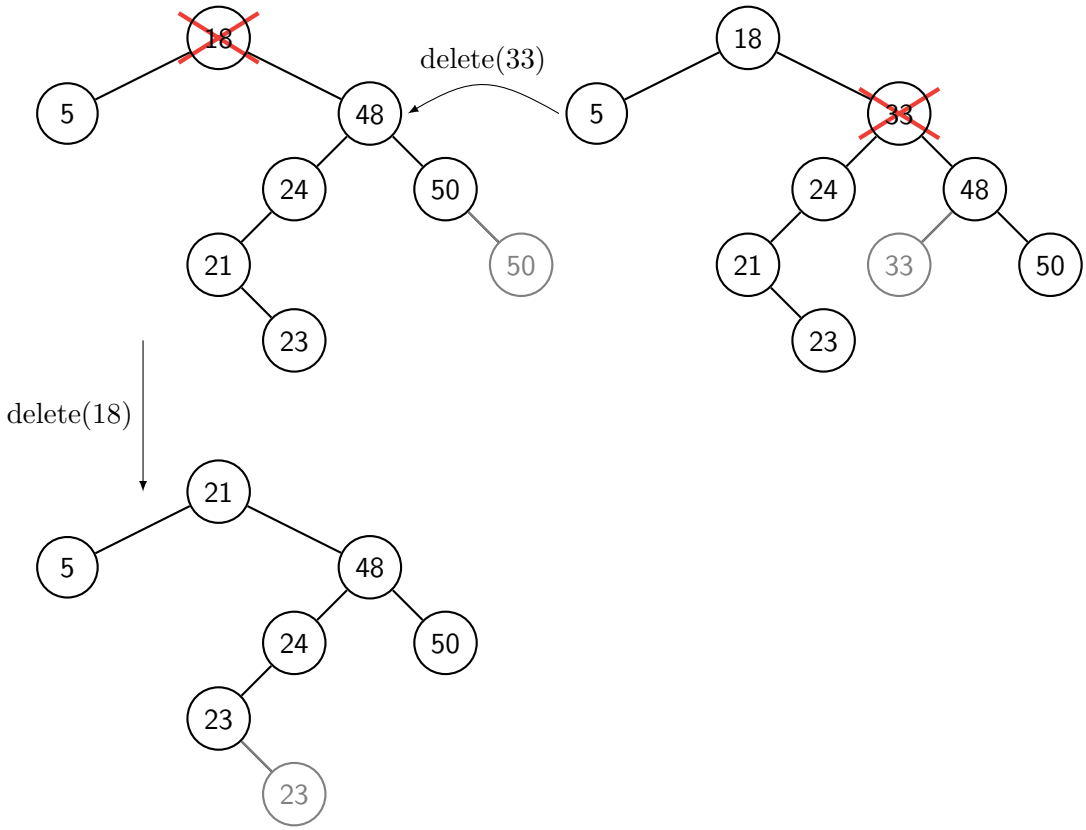
Lösung zur Aufgabe 4.16
Die einzelnen Schritte sind auf der folgenden Seite ersichtlich:


insert(7)
insert(10)
insert(33)
insert(27)
insert(31)
insert(27)
insert(0)
27 a
7 b
10
33 d
10 c
27 e
31 f
27 g
0 h

Lösung zur Aufgabe 4.17
Welcher Fall beim Entfernen angewendet wurde, entnehmen Sie der Tabelle. Die Bäume unten zeigen auf, wie der Baum nach jedem Schritt aussieht.

Knoten	Fall
7	1
49	1
10	2
15	3
58	2
27	3
33	3
18	3


15
10
58
5
7
27
24
48
21
33
50
18
23
49
delete(7)
delete(49)
delete(10)
delete(15)
delete(58)
delete(27)

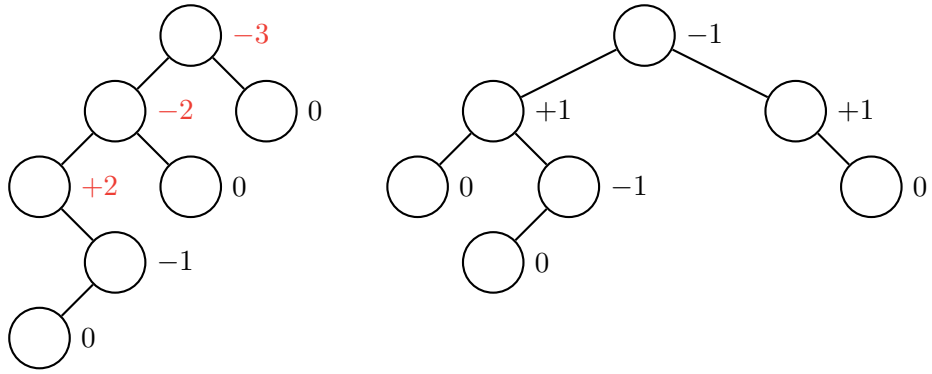


Lösung zur Aufgabe 4.18
Ein Vorgänger ist vom Prinzip her das gleiche wie der Nachfolger - nur spiegelverkehrt:
Nachfolger : Knoten, das am weitesten links steht im rechten Teilbaum des zu löschenden Knotens
Vorgänger : Knoten, das am weitesten rechts steht im linken Teilbaum des zu löschenden Knotens

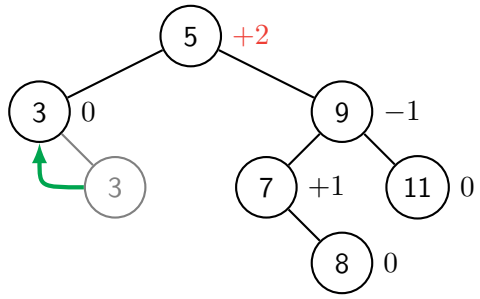
Lösung zur Aufgabe 4.19
Anstatt den rechten Nachbarn in der Inorder-Reihenfolge zu wählen, nimmt man den linken Nachbarn.

A.4 Lösungen zum Kapitel 5

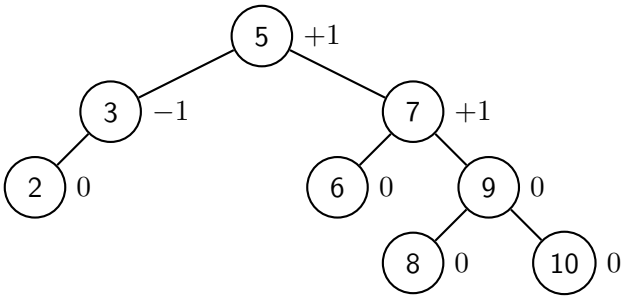
Lösung zur Aufgabe 5.1
Die folgende Zeichnung zeigt die beiden Bäume mit den eingetragenen Balancefaktoren. Der linke Baum ist *kein* AVL-Baum. Der rechte Baum ist ein AVL-Baum.



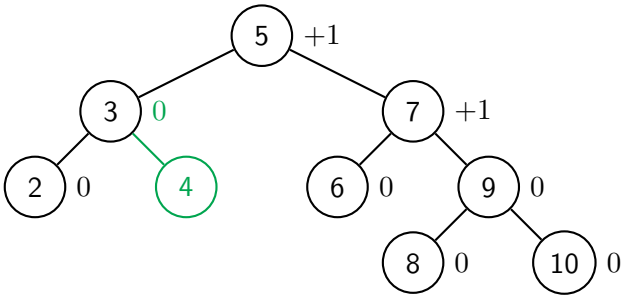
Lösung zur Aufgabe 5.2
Der resultierende Baum ist nicht mehr AVL-ausgeglichen, da der Knoten mit Schlüssel 5 mit einem Balancefaktor von +2 unausgeglichen ist.



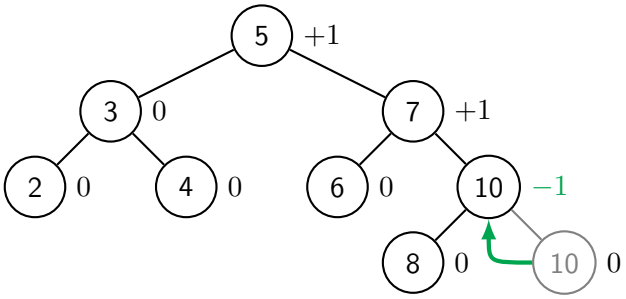
Lösung zur Aufgabe 5.3
Der AVL-Baum vor den beiden Operationen:



Der Baum nach dem Einfügen eines Knotens mit Schlüssel 4:

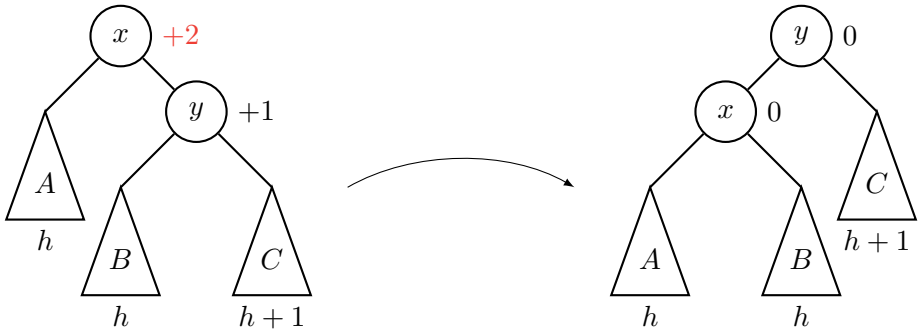


Der Baum nach dem Entfernen des Knotens mit Schlüssel 9:

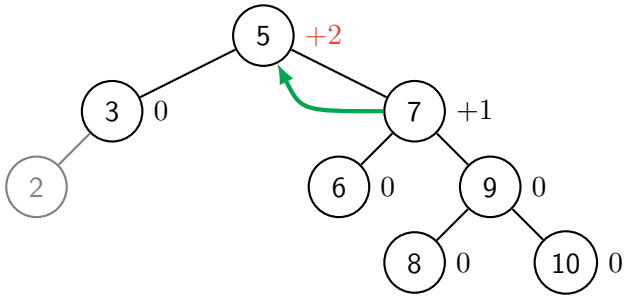


Es muss nach beiden Operationen keine Umstrukturierung stattfinden, da keiner der Knoten unausgeglichen wird.

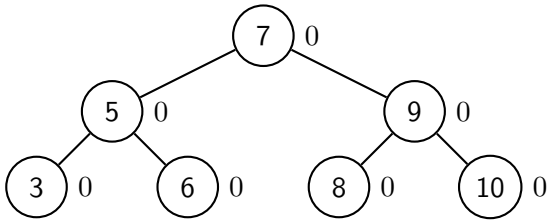
Lösung zur Aufgabe 5.4
Die folgende Zeichnung zeigt die einfache Rotation nach *links*.



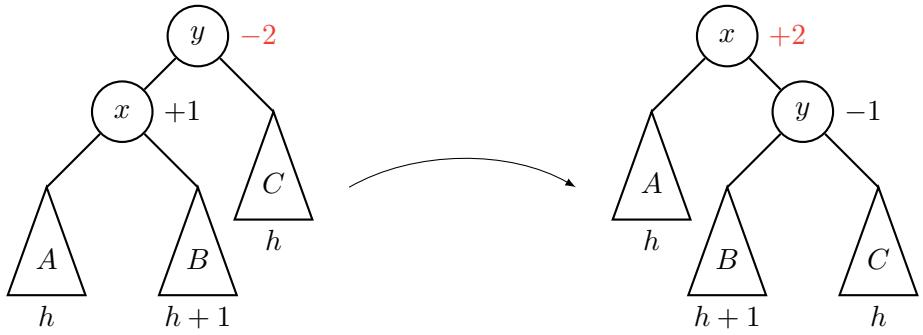
Lösung zur Aufgabe 5.5
Im gegebenen Baum liegt nach dem Entfernen des Knotens die Situation 1 vor. Der unausgeglichene Knoten mit Schlüssel 5 und sein Sohn im höheren Teilbaum haben das *gleiche* Vorzeichen.



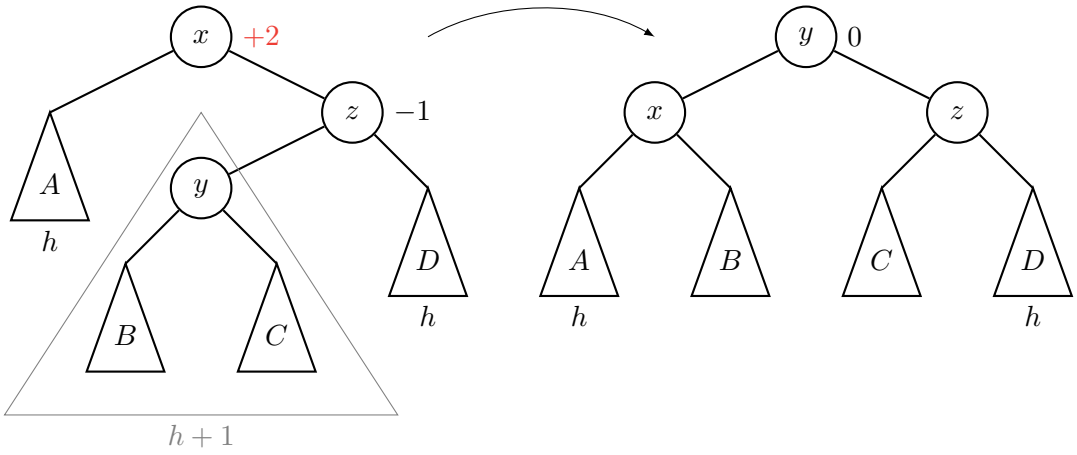
Eine einfache Rotation reicht aus um den folgenden AVL-ausgeglichenen Suchbaum zu erhalten:



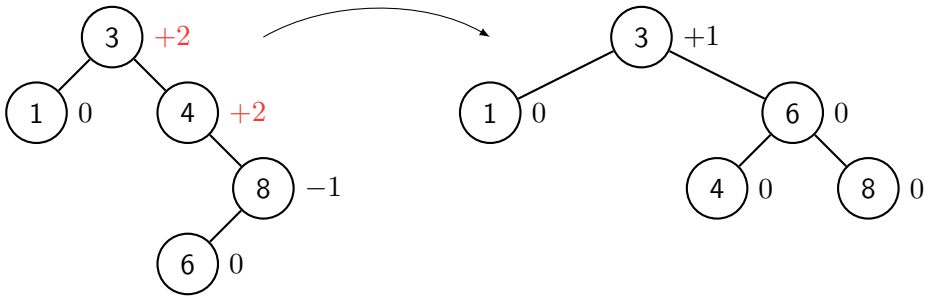
Lösung zur Aufgabe 5.6
Wird eine einfache Rotation bei Situation 3 angewendet, spiegelt sich das Problem auf die andere Seite und es ergibt sich wieder die gleiche, unausgeglichene Situation.



Lösung zur Aufgabe 5.7
Die folgende Zeichnung zeigt die Doppelrotation *rechts-links*.



Lösung zur Aufgabe 5.8
Der erste unausgeglichene Knoten auf dem Pfad vom eingefügten Knoten bis zur Wurzel ist der Knoten mit Schlüssel 4. Es liegt die Situation 3 vor um den Knoten mit Schlüssel 4 zu wieder auszugleichen. Es wird deshalb eine Doppelrotation rechts-links durchgeführt.

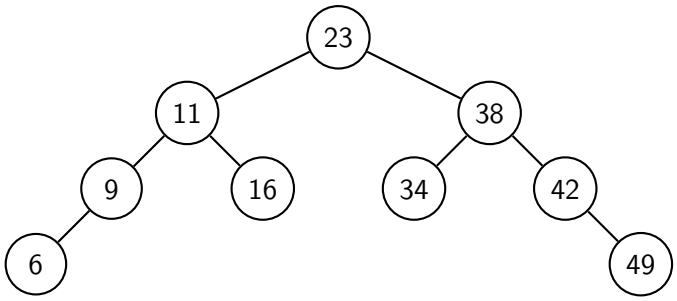


Nach dem Ausgleich des Knotens mit Schlüssel 4 wird dem Pfad weiter zur Wurzel gefolgt. Nun ist aber kein Knoten mehr unausgeglichen und somit ist die Rebalancierung beendet.

Lösung zur Aufgabe 5.11

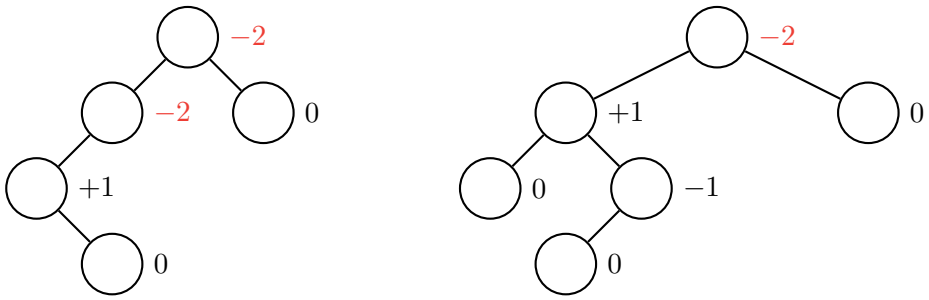
a) Im schlechtesten Fall werden neun Schritte benötigt.

b) Ein ausgeglichener binärer Suchbaum mit den gleichen Schlüsseln könnte folgendermassen aussehen. Es werden maximal vier Schritte zum Suchen benötigt.

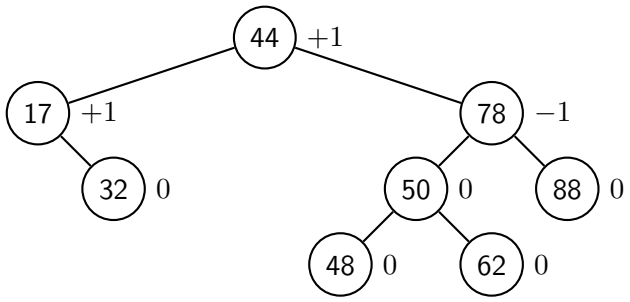


Lösung zur Aufgabe 5.12

Beide Bäume sind nicht AVL-ausgeglichen.



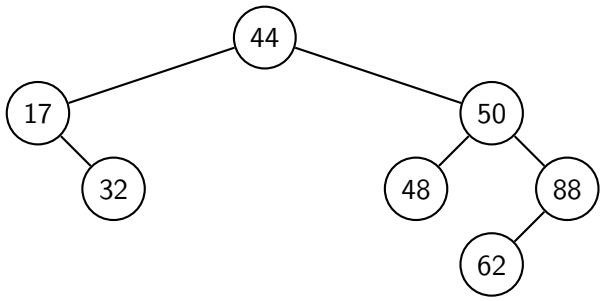
Lösung zur Aufgabe 5.13
Die folgende Zeichnung zeigt den Suchbaum mit den eingetragenen Balancefaktoren. Es ist ein AVL-Baum, da alle Knoten ausgeglichen sind.



Lösung zur Aufgabe 5.14
Die Funktion balanced(v) könnte folgendermassen aussehen:

```
1:  algorithm balanced(v)
2:      if v ≠ null then
3:          if −2 < height(right(v)) − height(left(v)) < 2 then
4:              return balanced(left(v)) ∧ balanced(right(v))
5:          else
6:              return false
7:          end if
8:      else
9:          return true
10:     end if
11: end algorithm
```

Lösung zur Aufgabe 5.15
Nachdem der Knoten mit Schlüssel 78 entfernt wurde und die notwendigen Umstrukturierungen stattgefunden haben, erhält man den folgenden AVL-Baum.



Lösung zur Aufgabe 5.16
Die folgende Abbildung zeigt den resultierenden AVL-Baum.