

Einführung

In den bisherigen Kurseinheiten lernten wir, Klassen zu implementieren und zu testen. Dabei beschränkten wir uns auf relativ einfache Klassen und Methoden.

Häufig müssen Daten in großen Mengen verwaltet werden. Für solche Aufgaben verwendeten wir bisher Felder. Wir werden uns in dieser Kurseinheit damit beschäftigen, wie wir in Feldern **suchen** und diese **sortieren** können.

Anschließend lernen wir das Prinzip der **Rekursion** kennen, das uns eine andere Herangehensweise für den Algorithmenentwurf bietet. Auch hierbei werden wir wieder auf Suchen und Sortieren von Feldern zurückkommen.

Felder haben den Nachteil, dass ihre Größe während der Erzeugung festgelegt werden muss und später nicht mehr verändert werden kann. In der Realität müssen jedoch häufig Datenmengen verwaltet werden, die mit der Zeit wachsen und auch wieder schrumpfen können. Für die Verwaltung solcher Datenmengen gibt es verschiedene (rekursive) **Datenstrukturen**, wie zum Beispiel **Listen**.

Einige Strukturen, wie zum Beispiel Straßennetze oder Moleküle, lassen sich nicht auf lineare Datenstrukturen abbilden. In solchen Fällen können **Graphen** verwendet werden. **Bäume** sind eine besondere Art von Graphen und ermöglichen ebenfalls die Verwaltung von sich ändernden Datenmengen.

Lernziele

- Das Prinzip der Rekursion kennen.
- Für verschiedenste Probleme rekursive Lösungen formulieren und implementieren können.
- Das Konzept des Methodenrahmens und -stapels verstehen.
- Den Unterschied zwischen linearer und binärer Suche erläutern können und beiden Verfahren anwenden und implementieren können.
- Verschiedene Sortieralgorithmen kennen, anwenden und implementieren können.
- Die Datenstruktur Liste sowie ihre Vor- und Nachteile kennen und erläutern können.
- Den Unterschied zwischen LIFO und FIFO kennen.
- Einfache und doppelt verkettete Listen inklusive gängiger Operationen kennen und in Java implementieren können.
- Graphen und (binäre) Bäume kennen und auf gegebene Probleme anwenden können.
- Verschiedene Arten von Bäumen kennen und diese in Java implementieren können.
- Durchlaufstrategien für Bäume kennen, anwenden und implementieren können.
- Breiten- und Tiefensuche erläutern und auf beliebigen Graphen anwenden können.

33 Suchen und Sortieren

Bisher haben wir uns primär mit einfachen Klassen und ihrem Entwurf beschäftigt und Techniken zur Qualitätssicherung kennen gelernt. Für reale Anwendungen werden jedoch meistens komplexere Daten und komplexeres Verhalten benötigt. Deshalb werden wir uns in diesem Kapitel mit gängigen Problemen und Algorithmen zum Thema Suchen und Sortieren beschäftigen. Diese Verfahren können auf verschiedenen Datenstrukturen angewendet werden; wir werden uns in diesem Kapitel zunächst auf Felder beschränken.

33.1 Suchen in Feldern

Suchen in strukturierten Datensammlungen ist eine häufig auftretende Programmieraufgabe, zu deren Lösung zahlreiche Algorithmen entwickelt wurden. Zunächst wollen wir eine Methode implementieren, die prüft, ob ein bestimmter Wert in einem Feld enthalten ist. Dazu vergleichen wir jedes Element mit dem gesuchten Wert.

Suchen in Feldern

```
boolean istEnthalten(int wert, int[] feld) {
    for (int i : feld) {
        // prüfe ob gesuchter Wert
        if (wert == i) {
            // wenn Wert gefunden, Methode beenden
            return true;
        }
    }
    // Wert wurde nicht gefunden
    return false;
}
```

Wir können die Methode sofort verlassen, wenn wir den gesuchten Wert gefunden haben. Wird das Ende der Schleife erreicht, so wurde der Wert nicht gefunden.

Selbsttestaufgabe 33.1-1:

Implementieren Sie eine Methode `int bestimmeAnzahl(int wert, int[] feld)`, die zählt, wie oft der übergebene Wert in dem Feld enthalten ist.



Selbsttestaufgabe 33.1-2:

Implementieren Sie eine Methode `boolean istEnthalten(String wert, String[] feld)`, die überprüft, ob die übergebene Zeichenkette im Feld enthalten ist. Was müssen Sie hierbei im Gegensatz zur Implementierung für Felder primitiver Typen beachten? ◇

Suche in sortierten
Feldern

Bei der ersten Implementierung von `istEnthalten()` sind wir davon ausgegangen, dass die Elemente des Feldes unsortiert sind. Wenn wir wissen, dass das Feld aufsteigend sortiert ist, können wir die Methode beenden, sobald der aktuelle Wert größer als der gesuchte ist.

```
// Das Feld muss aufsteigend sortiert sein
boolean istEnthalten(int wert, int[] feld) {
    for (int i : feld) {
        // prüfe ob gesuchter Wert
        if (wert == i) {
            // wenn Wert gefunden, Methode beenden
            return true;
        }
        if (wert < i) {
            // der Wert kann im Rest des Feldes nicht
            // mehr vorkommen
            return false;
        }
    }
    // Wert wurde nicht gefunden
    return false;
}
```

Felder können nicht nur auf identische Elemente durchsucht werden, sondern auch auf Objekte, die bestimmte Eigenschaften aufweisen. So könnten beispielsweise alle Rechnungen einer bestimmten Kundin gesucht werden oder der Gesamtbetrag aller Rechnungen eines Kunden bestimmt werden.

Selbsttestaufgabe 33.1-3:

Gegeben seien die folgenden Klassen

```
public class Kunde {
    // ...
}

public class Rechnung {
    private Kunde rechnungsempfaenger;
    // ...

    public Kunde liefereRechnungsempfaenger() {
        return this.rechnungsempfaenger;
    }
}
```

```

    public int bestimmeBetragInCent() {
        // ...
    }

    // ...
}

public class Rechnungssammlung {
    private Rechnung[] rechnungen;

    // ...
}

```

Implementieren Sie eine Methode

```
int bestimmeGesamtbetragAllerRechnungenVon(Kunde k)
```

in der Klasse `Rechnungssammlung`, die die Summe aller Rechnungsbeträge dieses Kunden in Cent bestimmt.



Neben der Suche nach bestimmten Elementen kann auch das Maximum oder Minimum bestimmt werden.

Maximums- und
Minimumssuche

Selbsttestaufgabe 33.1-4:

Implementieren Sie eine Methode `Rechnung findeTeuersteRechnung()` in der Klasse `Rechnungssammlung`, die die Rechnung mit dem größten Betrag bestimmt.



Alle Suchen können auch auf mehrdimensionalen Feldern durchgeführt werden.

Suchen auf mehrdimen-
sionalen Feldern
Suchen von Teilfolgen

Bisher haben wir einzelne Elemente betrachtet, doch auch Teilfolgen können in einem Feld gefunden werden. So kann beispielsweise geprüft werden, ob eine bestimmte Zahlenfolge in einem Feld vorkommt. Man beginnt damit, den ersten Wert des Feldes mit dem ersten Wert der Zahlenfolge zu vergleichen. Stimmen die Werte überein, vergleicht man anschließend die zweiten Werte und so weiter. Stimmen die Werte nicht überein, so weiß man lediglich, dass die Zahlenfolge nicht beim ersten Element des Feldes beginnend gefunden werden kann. Man beginnt dann mit dem zweiten Element des Feldes und vergleicht es mit dem ersten der gesuchten Folge. Stimmen diese überein, so vergleicht man anschließend das dritte mit dem zweiten und so weiter. Abb. 33.1-1 veranschaulicht diese Schritte an einem Beispiel.

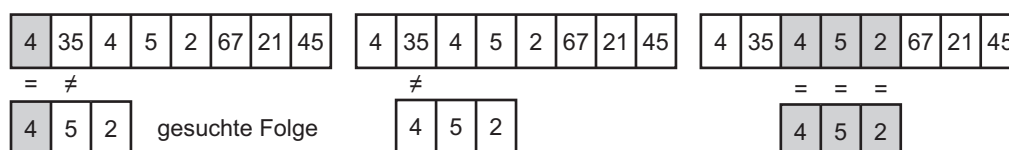


Abb. 33.1-1: Suchen einer Teilfolge in einem Feld

Wurde die gesamte Teilfolge gefunden, so kann die Methode beendet werden. Bei der Implementierung müssen wir aufpassen, dass wir nicht auf ungültige Feldelemente zugreifen.

```
boolean istTeilfolge(int[] feld, int[] gesuchteFolge) {
    // es muss nicht bis zum letzten Element gesucht werden,
    // da dort die Folge nicht mehr vollständig vorkommen kann
    for (int i = 0; i < feld.length
        - gesuchteFolge.length + 1; i++) {
        for (int j = 0; j < gesuchteFolge.length; j++) {
            // Vergleiche entsprechende Einträge
            if (feld[i + j] != gesuchteFolge[j]) {
                // Zahl an Position j stimmt nicht überein
                // suche bei i+1 weiter
                break;
            } else if (j == gesuchteFolge.length - 1) {
                // gesamte Folge wurde gefunden
                return true;
            }
        }
    }
    // Teilfolge konnte nicht vollständig gefunden werden
    return false;
}
```

Selbsttestaufgabe 33.1-5:

Implementieren Sie eine Methode `int bestimmeAnfangdesWorts(char[] feld, String wort)`, die die Position zurück liefert, ab der das Wort vollständig im Feld gefunden wurde. Wird das Wort nicht vollständig gefunden, so soll `-1` zurückgeliefert werden. ◇

Bemerkung 33.1-1: String-Matching

String-Matching *Das in Selbsttestaufgabe 33.1-5 beschriebene Problem wird auch als String-Matching bezeichnet. Zur Lösung dieses Problems existieren diverse Algorithmen.* ┘

Nachdem wir uns bisher mit der Suche in Feldern beschäftigt haben, werden wir uns im nächsten Abschnitt der Sortierung von Feldern widmen. Die beiden Probleme hängen insofern zusammen, als eine Suche auf sortierten Daten meistens effizienter lösbar ist.

33.2 Sortieren von Feldern

Sortieren von Feldern

Das Suchen in großen Datenbeständen wird erheblich beschleunigt, wenn die Datenelemente gemäß einer Ordnungsrelation sortiert wurden. Der Suchalgorithmus kann diese Eigenschaft nutzen, um ein bestimmtes Element in der sortierten Menge zu finden.

Beim Kartenspielen sortieren wir in der Regel unser Blatt nach dem Verfahren „Sortieren beim Einfügen“ (engl. *insertion sort*). Wir nehmen eine Karte auf und fügen sie an der richtigen Stelle des bereits sortierten Blatts ein. Dieses Verfahren können wir auch beim Sortieren der Elemente eines Feldes nachbilden. Wir betrachten das erste Element des Feldes als sortierte und den Rest der Elemente als unsortierte Menge. Nun vergleichen wir, mit dem zweiten Element beginnend, die nicht sortierten Elemente mit den Elementen mit kleinerem Index (die ja bereits sortiert sind) und fügen das betrachtete Element durch Vertauschen (engl. *swap*) von rechts nach links an der richtigen Stelle ein. Das Feld ist sortiert, wenn das Ende des Feldes erreicht ist.

Sortieren beim Einfügen
insertion sort

```
void sortiereAufsteigend(int[] feld) {
    // beginne beim zweiten Element und betrachte
    // die Liste bis zum Index i - 1 als sortiert
    for (int i = 1; i < feld.length; i++) {
        // gehe solange von i nach links
        // bis das Element an die richtige
        // Stelle gerutscht ist
        for (int j = i; j > 0; j--) {
            if (feld[j - 1] > feld[j]) {
                // wenn linkes grösser,
                // vertausche die Elemente
                int temp = feld[j];
                feld[j] = feld[j - 1];
                feld[j - 1] = temp;
            } else {
                // das Element ist
                // an der richtigen Stelle
                break;
            }
        }
    }
}
```

Beachten Sie, dass das übergebene Feld direkt verändert wird und die Methode deshalb auch keinen Ergebnistyp benötigt.


Abb. 33.2-1 veranschaulicht das Verhalten des Sortieralgorithmus.

Selbsttestaufgabe 33.2-1:

Wie viele Vergleichs- und Vertauschungsoperationen benötigt der Algorithmus „Sortieren beim Einfügen“, um das folgende Feld zu sortieren?

```
int[] beispiel = {4, 35, 23, 5, 2, 67, 45, 21};
```

Als Vergleich zählt die Auswertung der `if`-Bedingung.

Wir betrachten die drei Anweisungen im `if`-Block als eine Vertauschungsoperation. 

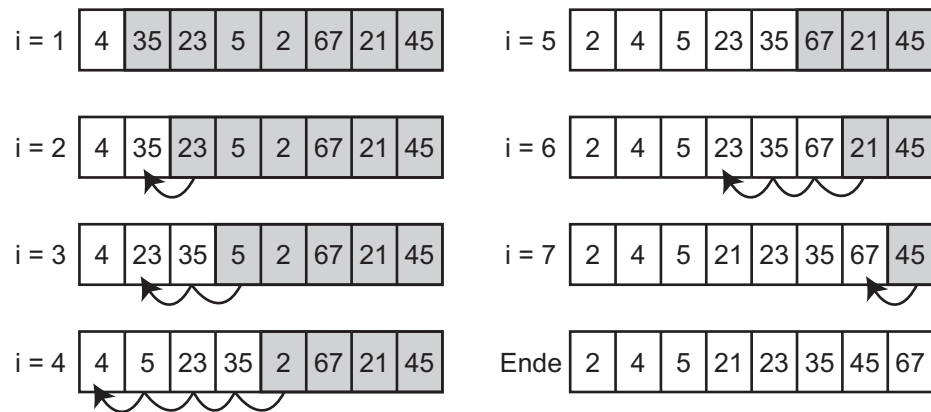


Abb. 33.2-1: Sortieren beim Einfügen (insertion sort)

Selbsttestaufgabe 33.2-2:

Entwerfen Sie analog eine Methode `void sortiereAbsteigend(String[] feld)`, die absteigend ein Feld von Zeichenketten sortiert. ◇

Bubblesort Ein weiterer klassischer Sortieralgorithmus ist Bubblesort. Bubblesort gleicht dem Sortieren durch Einfügen darin, dass benachbarte Elemente vertauscht werden, sofern sie nicht geordnet sind. Bubblesort vollzieht diese Vertauschungen aber in einer anderen Reihenfolge. Der Algorithmus ist der Beobachtung nachgebildet, dass sich verschieden große, in einer Flüssigkeit aufsteigende Blasen von selbst sortieren, weil die kleineren Blasen von den größeren beim Aufsteigen überholt werden. Der Algorithmus vergleicht zwei aufeinander folgende Feldelemente `feld[i]` und `feld[i + 1]` miteinander. Falls `feld[i] > feld[i + 1]` gilt, werden die Werte beider Elemente miteinander vertauscht. Beginnend mit dem ersten Feldelement wird das Feld durchlaufen. Wenn nötig werden die beiden Elemente vertauscht. Dies wird solange von vorne wiederholt, bis keine Vertauschungen mehr notwendig sind, weil kein größerer Wert mehr vor einem kleineren vorkommt.

Das Verfahren ist in Abb. 33.2-2 veranschaulicht.

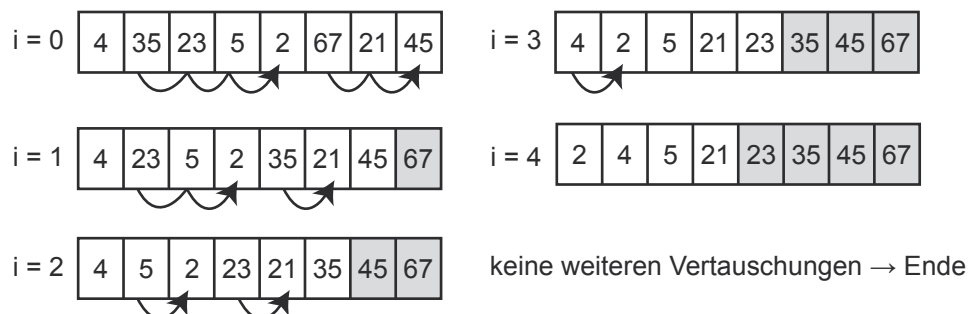


Abb. 33.2-2: Bubblesort

Wir stellen fest, dass nach dem ersten Felddurchlauf das größte Element an der richtigen Stelle steht. Im zweiten Durchlauf müssen wir also das letzte Element

nicht mehr berücksichtigen. Nach dem zweiten Durchlauf steht dann das zweitgrößte Element an der richtigen Position, so dass im dritten Durchlauf die letzten beiden Elemente nicht mehr betrachtet werden müssen.

Sobald in einem Durchlauf keine Vertauschungen mehr durchgeführt wurden, ist das Feld vollständig sortiert.

```
void bubblesort(int[] feld) {
    // es werden maximal feld.length - 1 Durchläufe benötigt
    for (int i = 0; i < feld.length - 1; i++) {
        // solange keine Vertauschungen vorgenommen werden
        // ist das Feld sortiert
        boolean sorted = true;
        // Durchlaufe das Feld, in jedem Durchlauf muss
        // ein Element weniger berücksichtigt werden
        for (int j = 0; j < feld.length - 1 - i; j++) {
            if (feld[j] > feld[j + 1]) {
                // wenn linkes größer
                // dann vertausche
                int temp = feld[j];
                feld[j] = feld[j + 1];
                feld[j + 1] = temp;
                // Feld ist nicht sortiert
                sorted = false;
            }
        }
        if (sorted) {
            // keine Vertauschungen, Feld ist
            // folglich vollständig sortiert
            break;
        }
    }
}
```

Selbsttestaufgabe 33.2-3:

Wie viele Vergleichs- und Vertauschungsoperationen benötigt der Algorithmus Bubblesort, um das folgende Feld zu sortieren?

```
int[] beispiel = {4, 35, 23, 5, 2, 67, 45, 21};
```

Als Vergleich zählt die Auswertung der if-Bedingung.

Wir betrachten die drei Anweisungen im inneren if-Block als eine Vertauschungsoperation.



Selbsttestaufgabe 33.2-4:

Entwerfen Sie mit Hilfe des Bubblesort-Algorithmus eine Methode void sortiereAufsteigend(Rechnung[] rechnungen), die die Rechnungen aufsteigend nach ihren Beträgen sortiert.



Selbsttestaufgabe 33.2-5:

Sortieren durch
Auswählen
selection sort

Der Algorithmus „Sortieren durch Auswählen“ (engl. selection sort) nimmt am Anfang das komplette Feld als unsortiert an, sucht sich das größte Element unter den unsortierten Elementen und vertauscht es, wenn nötig, anschließend mit dem letzten Element des unsortierten Bereichs.

Im nächsten Schritt gehört das letzte Element nun zum sortierten Bereich. Es wird wiederum das größte Element des unsortierten Bereichs mit dem letzten Element vertauscht, so dass es anschließend zum sortierten Bereich gehört. Der Algorithmus terminiert, wenn der unsortierte Bereich aus einem Element besteht.

Wenden Sie den Algorithmus auf das folgende Feld an und zählen Sie die dabei nötigen Vergleichs- und Vertauschungsoperationen.

```
int[] beispiel = {4, 35, 23, 5, 2, 67, 45, 21};
```

**Selbsttestaufgabe 33.2-6:**

Implementieren Sie die Methode `void sortiere(double[] feld)` so, dass sie das übergebene Feld mit Hilfe des Algorithmus „Sortieren durch Auswählen“ sortiert.

**Selbsttestaufgabe 33.2-7:**

Vergleichen Sie die bei den verschiedenen Sortieralgorithmen notwendigen Vergleichs- und Vertauschungsoperationen für die folgenden Felder.

```
int[] beispiel1 = {4, 35, 23, 5, 2, 67, 45, 21};
int[] beispiel2 = {2, 4, 5, 21, 23, 35, 45, 67};
int[] beispiel3 = {67, 45, 35, 23, 21, 5, 4, 2};
```

Was fällt Ihnen dabei auf?

**Weiterführende Literatur**

[Sedgewick03] Robert Sedgewick:

Algorithms in Java

Addison Wesley 2003

Der Autor hat schon zahlreiche Bücher zum gleichen Thema in englischer Sprache verfasst. Das Werk ist didaktisch gut aufgebaut und enthält zahlreiche Illustrationen und Programmcode in Java.

[Cormen09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein:

Introduction to Algorithms, Third Edition

MIT Press 2009

34 Rekursion

Bisher entwickelten wir Algorithmen auf der Basis von Programmstrukturen, die aus Folgen von Anweisungen, aus Verzweigungen und aus Schleifen bestanden. Schleifen wurden insbesondere dann verwandt, wenn Datenstrukturen mit einer variierenden Anzahl von Elementen iterativ zu verarbeiten waren. In diesem Kapitel wollen wir unser Inventar an Programmstrukturen um das Prinzip der Rekursion erweitern.

Rekursion

Definition 34-1: Rekursiver Algorithmus

Ein Algorithmus ist rekursiv, wenn er Methoden (oder Funktionen) enthält, die sich selbst aufrufen.

Rekursiver Algorithmus

Jede rekursive Lösung umfasst zwei grundlegende Teile:

- *den Basisfall, für den das Problem auf einfache Weise gelöst werden kann, und*
- *die rekursive Definition.*

Basisfall

rekursive Definition

Die rekursive Definition besteht aus drei Facetten:

1. *die Aufteilung des Problems in einfachere Teilprobleme,*
2. *die rekursive Anwendung auf alle Teilprobleme und*
3. *die Kombination der Teillösungen in eine Lösung für das Gesamtproblem.*

Bei der rekursiven Anwendung ist darauf zu achten, dass wir uns mit zunehmender Rekursionstiefe an den Basisfall annähern. Wir bezeichnen diese Eigenschaft als Konvergenz der Rekursion.

Konvergenz

Eine rekursive algorithmische Lösung bietet sich immer dann an, wenn man ein Problem in einfachere Teilprobleme aufspalten kann, die mit dem Originalproblem nahezu identisch sind und die man zuerst nach dem gleichen Verfahren löst.

Nehmen wir an, dass wir die Summe der ersten n natürlichen Zahlen berechnen wollen. Dies können wir mit unserem bisherigen Wissen iterativ mit Hilfe einer Schleife lösen.

```
int summeIterativ(int n) {
    int ergebnis = 0;
    for (int i = 1; i <= n; i++) {
        ergebnis += i;
    }
    return ergebnis;
}
```

Wir können es aber auch als rekursives Problem definieren, denn die Summe der ersten n Zahlen lässt sich berechnen, indem zunächst die Summe der ersten $n - 1$ Zahlen berechnet wird und anschließend die n -te Zahl hinzu addiert wird. Das Problem wird dadurch von n Zahlen auf $n - 1$ Zahlen reduziert. Natürlich müssen wir auch einen Basisfall identifizieren. Dieser ist erreicht, wenn keine Zahl mehr übrig ist. Wir können das Problem folgendermaßen mathematisch beschreiben:

$$\text{summe}(n) = \begin{cases} 0 & \text{wenn } n == 0 \\ \text{summe}(n - 1) + n & \text{sonst} \end{cases}$$

In Java können wir eine solche Lösung formulieren, indem wir die Methode selbst wieder aufrufen:

```
int summeRekursiv(int n) {
    // Basisfall: keine Zahl übrig
    if (n == 0) {
        return 0;
    }
    // sonst: rekursiver Aufruf
    return summeRekursiv(n - 1) + n;
}
```

Wird nun die Methode `summeRekursiv()` mit dem Wert 5 aufgerufen, so finden die folgenden Berechnungen statt:

```
summeRekursiv(5) =
summeRekursiv(5 - 1) + 5 =
(summeRekursiv(4 - 1) + 4) + 5 =
((summeRekursiv(3 - 1) + 3) + 4) + 5 =
(((summeRekursiv(2 - 1) + 2) + 3) + 4) + 5 =
((((summeRekursiv(1 - 1) + 1) + 2) + 3) + 4) + 5 = // Basisfall
((((0 + 1) + 2) + 3) + 4) + 5 =
(((1 + 2) + 3) + 4) + 5 =
((3 + 3) + 4) + 5 =
(6 + 4) + 5 =
10 + 5 =
15
```

Selbsttestaufgabe 34-1:

Was passiert, wenn die Methode `summeRekursiv()` mit negativen Werten aufgerufen wird? Wie kann dieses Problem gelöst werden?



Fakultätsfunktion

Eine weiteres Beispiel ist die Fakultätsfunktion. Sie spielt bei vielen mathematischen Anwendungen eine wichtige Rolle. Sie wird typischerweise durch das Ausrufezeichen „!“ symbolisiert. Die Fakultätsberechnung ist für Eingabewerte $n \geq 0$ wie folgt definiert:

$$n! = \begin{cases} n * (n - 1)! & \text{wenn } n > 1 \\ 1 & \text{wenn } n \leq 1 \end{cases}$$

Selbsttestaufgabe 34-2:

Implementieren Sie eine Methode `fakultaetIterativ(int n)`, die iterativ die Fakultät berechnet und eine Methode `fakultaetRekursiv(int n)`, die die Fakultät rekursiv berechnet. Die Methoden sollen dabei nur Werte $n \geq 0$ akzeptieren. ◇

Selbsttestaufgabe 34-3:

Begründen Sie, warum die Lösung für `fakultaetRekursiv()` konvergiert. ◇

Selbsttestaufgabe 34-4:

Schreiben Sie die Methode `double power(int p, int n)`, die für zwei ganze Zahlen p und n rekursiv den Wert p^n berechnet. Testen Sie Ihr Programm und vergleichen Sie es mit der Lösung von Selbsttestaufgabe 14.1-2, die eine iterative Variante aufzeigt. ◇

Die bisher betrachteten rekursiven Methoden hatten die Eigenschaft, dass sie nur einen rekursiven Aufruf pro Ablaufpfad beinhalten. Allerdings gibt es auch einige Probleme, die mehrere rekursive Aufrufe benötigen. Ein bekanntes Beispiel sind die Fibonacci-Zahlen. Die Fibonacci-Zahlen berechnen sich nach der folgenden Formel:

Fibonacci-Zahlen

$$fib(n) = \begin{cases} n & \text{wenn } 0 \leq n \leq 1 \\ fib(n-1) + fib(n-2) & \text{wenn } n > 1 \end{cases}$$

Selbsttestaufgabe 34-5:

Berechnen Sie alle Fibonacci-Zahlen bis `fib(8)`. ◇

Die Implementierung einer passenden Methode ist nach dem gleichen Muster wie oben möglich. Negative Werte für n sind nicht zulässig; bei negativen Werten werfen wir eine `IllegalArgumentException`.

```
long fibRekursiv(int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Basisfall: n ist 0 oder 1
    if (n == 0 || n == 1) {
        return n;
    }
    // sonst: rekursiver Aufruf
    return fibRekursiv(n - 1) + fibRekursiv(n - 2);
}
```

Selbsttestaufgabe 34-6:

Implementieren Sie eine iterative Lösung für die Berechnung der Fibonacci-Zahlen in der Methode `long fibIterativ(int n)`. ◇

Selbsttestaufgabe 34-7:

Pseudozufallszahlen

In der Programmierung werden bisweilen Zufallszahlen benötigt. Es gibt Formeln zur Erzeugung sogenannter Pseudozufallszahlen, die eine Folge zufälliger Zahlen simulieren. Eine mögliche Formel zur Erzeugung solcher Zahlen ist die folgende:

$$f(n) = \begin{cases} n+1 & \text{wenn } n < 3 \\ 1 + (((f(n-1) - f(n-2)) * f(n-3)) \bmod 100) & \text{sonst} \end{cases}$$

- Implementieren Sie eine Methode `long zufallszahl(int n)`, die rekursiv $f(n)$ berechnet.
- Implementieren Sie eine Methode `void gebeZufallszahlenAus()`, die die Pseudozufallszahlen $f(5)$ bis einschließlich $f(30)$ ausgibt. ◇

Methodenrahmen

Bisher haben wir noch nicht weiter darüber nachgedacht, wie es funktionieren kann, dass eine Methode sich selbst aufruft. In Java wird bei einem Methodenaufruf ein Methodenrahmen erzeugt. In diesem Methodenrahmen sind alle aktuellen Werte der Parameter und lokalen Variablen sowie ein Verweis auf die aufrufende Methode gespeichert (siehe Abb. 34-1). Die Anweisungen einer Methode existieren nur einmal, sie sind nicht in jedem Methodenrahmen gespeichert.

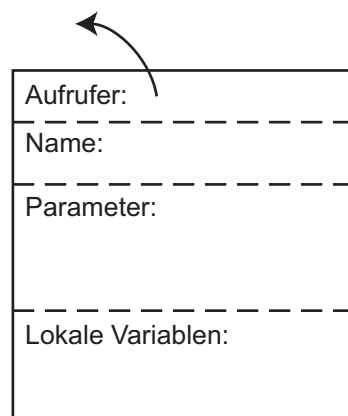


Abb. 34-1: Ein Methodenrahmen

Ruft nun eine Methode eine andere auf, so wird ein neuer Methodenrahmen erzeugt, und die Parameter und lokalen Variablen werden mit ihren Werten initialisiert. Dabei macht es keinen Unterschied, ob eine Methode eine andere oder eben sich selbst aufruft. Ist die aufgerufene Methode beendet, so wird dies dem Aufrufer – ggf. zusammen mit einem Rückgabewert – mitgeteilt.

Gegeben seien die beiden folgenden Methoden:

```
int a(int x) {
    int y = 2 * x;
    int z = 3;
    int w = b(y, z) + x;
    return w;
}

int b(int c, int d) {
    int e = c + 2 * d;
    return e;
}
```

Beim Aufruf der Methode `a()` mit dem Argument 3 wird ein Methodenrahmen für den Aufruf `a(3)` erzeugt (Abb. 34-2).

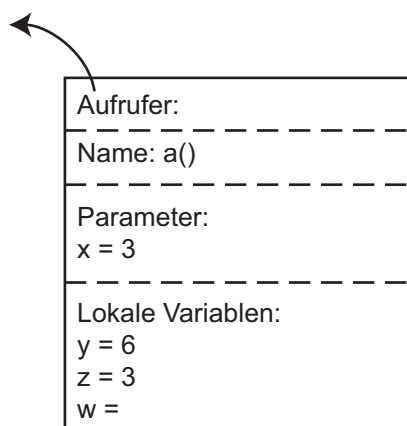


Abb. 34-2: Methodenrahmen für `a(3)`

Erreicht die Ausführung von `a(3)` den Aufruf von `b(6, 3)`, so wird auch dafür ein Methodenrahmen erzeugt (Abb. 34-3).

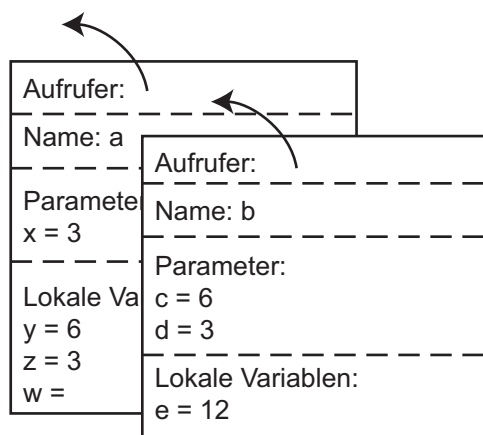


Abb. 34-3: Methodenrahmen für `a(3)` und `b(6, 3)`

Ist die Ausführung von `b(6, 3)` beendet, kann der zugehörige Methodenrahmen wieder gelöscht werden und die Ausführung von `a(3)` wird an der entsprechenden Stelle fortgesetzt.

Bei einer rekursiven Methode passiert das gleiche, nur dass dort die Methodenrahmen alle von der gleichen Methode stammen. Sie werden jeweils mit eigenen Parametern und lokalen Variablen erzeugt. Abb. 34-4 veranschaulicht die Schritte bei der Ausführung von `summeRekursiv(4)`.

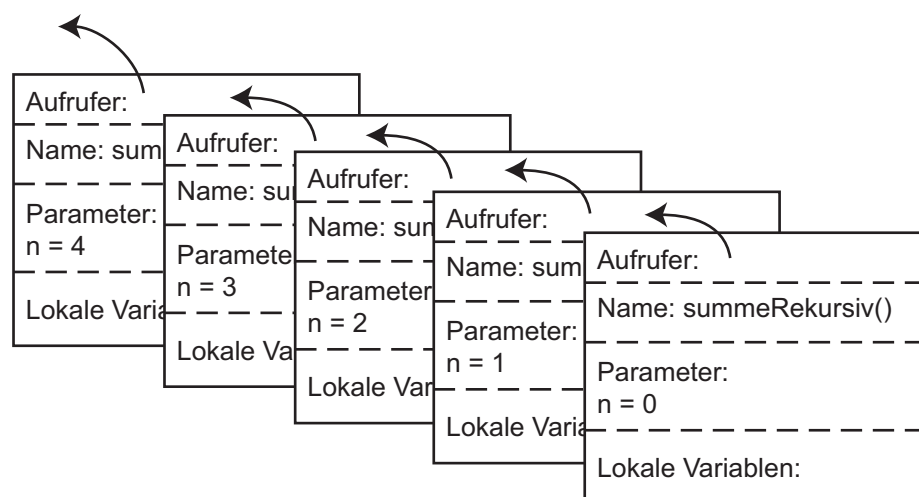


Abb. 34-4: Methodenstapel bei der Ausführung von `summeRekursiv(4)`

Methodenstapel Wenn eine Methode eine andere aufruft, so entsteht ein sogenannter Methodenstapel (engl. *stack*). Vom Methodenstapel wird immer nur die oberste, also die zuletzt aufgerufene Methode ausgeführt. Erst wenn diese beendet ist und wieder vom Stapel entfernt wurde, kann die Methode darunter fortgesetzt werden. Die Datenstruktur des Stapels findet auch in anderen Bereichen Anwendung (siehe Kapitel 35).

Bemerkung 34-1: StackOverflowError

StackOverflow-Error Wenn nicht mehr genug Speicherplatz für neue Methodenrahmen vorhanden ist, erzeugt die virtuelle Maschine in Java einen `StackOverflowError`. Dieser Fall tritt auf, wenn eine Rekursion niemals den Basisfall erreicht oder der Basisfall erst nach zu vielen Schritten erreichen würde.

┘

Rekursionen können nicht nur bei mathematischen Funktionen verwendet werden, sondern auch in vielen anderen Bereichen.

Palindrom Ein Palindrom ist ein Wort, dass sowohl vorwärts als auch rückwärts gelesen das gleiche ist.

Selbsttestaufgabe 34-8:

Implementieren Sie die Methode `boolean istPalindromIterativ(String s)`, die iterativ prüft ob es sich bei der Zeichenkette um ein Palindrom handelt.



Der Begriff Palindrom lässt sich auch rekursiv definieren. Ein Wort aus einem oder keinen Zeichen ist immer ein Palindrom. Ein längeres Wort ist dann ein Palindrom, wenn der erste und letzte Buchstabe identisch sind und der Rest der Zeichenkette, also ohne den ersten und letzten Buchstaben, auch ein Palindrom ist.

Selbsttestaufgabe 34-9:

Implementieren Sie die Methode `boolean istPalindromRekursiv(String s)`, die mit Hilfe der rekursiven Definition prüft, ob es sich bei der Zeichenkette um ein Palindrom handelt.



Auch für das Suchen in und das Sortieren von Feldern gibt es einige rekursive Algorithmen. Bisher haben wir ein sortiertes Feld immer iterativ von einem Ende zum anderen durchsucht. Dieses Verfahren wird auch lineare Suche genannt. Wir können jedoch auch das mittlere Element eines sortierten Feldes betrachten und entscheiden, in welcher der beiden Hälften sich unser gesuchtes Element befindet. Anschließend halbieren wir die Hälfte wieder und treffen die gleiche Entscheidung. Wir gehen solange so vor, bis die zu durchsuchende Menge maximal noch 2 Elemente beinhaltet. Abb. 34-5 veranschaulicht dieses Verfahren.

lineare Suche

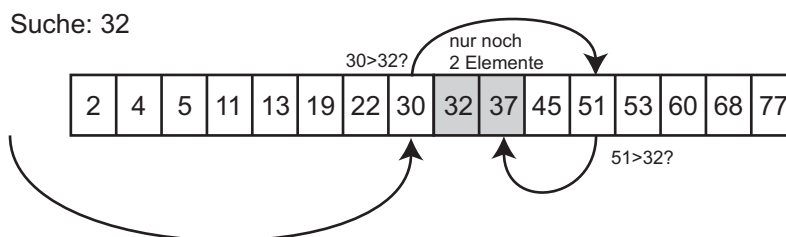


Abb. 34-5: Binäre Suche

Wir können die Methode `boolean istEnthalten(int wert, int[] feld, int start, int ende)`, die im Bereich `start` bis einschließlich `ende` im Feld nach dem Wert sucht, folgendermaßen rekursiv implementieren:

```
// wir gehen von einem sortierten Feld aus
boolean istEnthalten(int wert, int[] feld,
                    int start, int ende) {
    // Basisfall: Bereich enthält maximal 2 Elemente
    if (ende - start <= 1) {
        return feld[start] == wert || feld[ende] == wert;
    }
}
```

```

        // sonst: rekursive Aufteilung
        // Mitte bestimmen
        int mitte = start + (ende - start) / 2;
        if (feld[mitte] == wert) {
            // wert gefunden
            return true;
        }
        if (feld[mitte] > wert) {
            // in linker Hälfte suchen
            return istEnthalten(wert, feld, start, mitte - 1);
        } else {
            // in rechter Hälfte suchen
            return istEnthalten(wert, feld, mitte + 1, ende);
        }
    }
}

```

Wir stellen fest, dass die Methode zwei Parameter hat, die für eine Überprüfung, ob ein Element in einem Feld enthalten ist, nicht wirklich notwendig sind. Wir können die Methode mit vier Parametern als `private` deklarieren und zusätzlich eine öffentliche Methode mit zwei Parametern anbieten, die die private Methode dann aufruft.

```

public class Binaersucher {

    public boolean istEnthalten(int wert, int[] feld) {
        return istEnthalten(wert, feld, 0, feld.length - 1);
    }

    private boolean istEnthalten(int wert, int[] feld,
                                  int start, int ende) {

        // ...
    }
}

```

binäre Suche
binary search

Dieser Algorithmus wird als binäre Suche (engl. *binary search*) bezeichnet. Natürlich könnte auch schon bei größeren Restmengen auf ein anderes, zum Beispiel iteratives Suchverfahren umgeschaltet werden.

Ein ähnliches Verfahren wenden wir auch häufig an, wenn wir zum Beispiel in einem Lexikon nach einem bestimmten Begriff suchen. Wir schlagen eine Seite auf, sehen nach, ob wir uns vor oder nach dem gesuchten Wort im Alphabet befinden, und wählen dann aus, in welchem Teil wir weiter suchen. Dabei lassen wir jedoch bei der Auswahl der nächsten Seite unser Wissen über die Position der Buchstaben im Alphabet mit einfließen, so dass wir nicht immer genau die Mitte des entsprechenden Teils auswählen. Dieses Wissen steht bei der binären Suche nicht zur Verfügung. Wird Wissen über die Verteilung bei der Suche berücksichtigt, so spricht man von einer Interpolationssuche.

Interpolationssuche

Selbsttestaufgabe 34-10:

Führen Sie auf dem folgenden Feld eine binäre und eine lineare Suche nach dem Element 38 aus. Wie viele Vergleiche benötigen Sie, bis Sie das Element gefunden haben?

```
int[] y = {3, 7, 14, 16, 18, 22, 27, 29, 30, 34, 38, 40, 50};
```

Selbsttestaufgabe 34-11:

Ergänzen Sie die Klasse `Binaersucher` um eine Methode `boolean istEnthalten(String s, String[] feld)`, die mit Hilfe einer binären Suche prüft, ob die Zeichenkette in dem Feld enthalten ist. Sie können sich dabei Hilfsmethoden definieren. Hilfsmethoden sollten nach dem Geheimnisprinzip gekapselt sein.

Für das Sortieren von Feldern gibt es beispielsweise den Algorithmus Quicksort. Bei diesem Verfahren wird zufällig ein Element des Feldes als sogenanntes Pivotelement ausgewählt. Anschließend werden die Elemente des Feldes aufgeteilt. In dem einen Teil befinden sich alle Elemente, die kleiner als das Pivotelement sind und im anderen alle, die größer als das Pivotelement sind. Anschließend werden beide Teile wiederum mit dem gleichen Verfahren aufgeteilt. Bei Teilen mit maximal zwei Elementen kann die Sortierung dann direkt vorgenommen werden. Anschließend ist das gesamte Feld sortiert. Abb. 34-6 veranschaulicht dieses Verfahren.

Sortieren von Feldern
Quicksort
Pivotelement

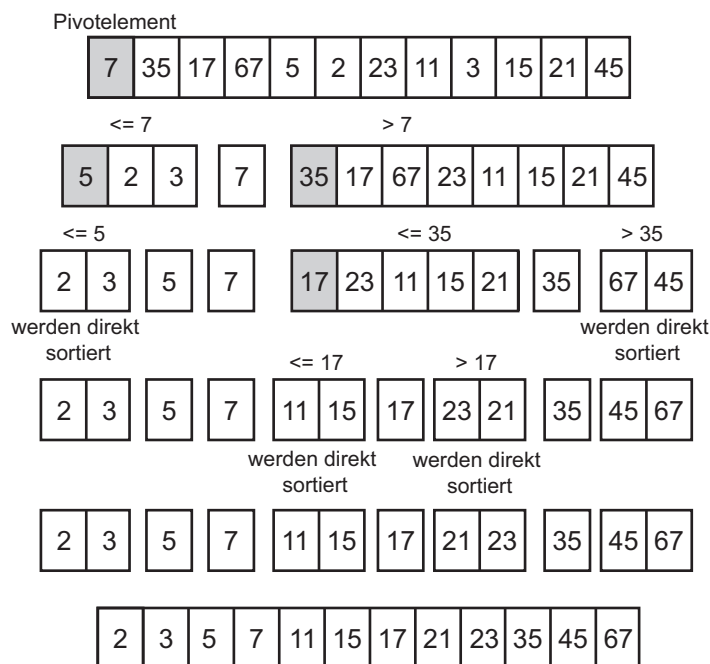


Abb. 34-6: Quicksort

Um dieses Verfahren zu implementieren teilen wir den Algorithmus in zwei Teile. Das Aufteilen des Feldes implementieren wir in der Methode `aufteilen()` und das Sortieren in der Methode `quicksort()`.

Dafür prüfen wir zunächst, ob das Feld weniger als 3 Elemente beinhaltet. Ist dies der Fall, so werden die beiden Elemente, wenn nötig, vertauscht. Bei einem größeren Feld wird das Feld mit Hilfe der Methode `aufteilen()` umsortiert. Anschließend werden die beiden Teile rekursiv mit dem gleichen Verfahren sortiert.

Um zu wissen, in welchem Bereich sie sortieren soll, benötigt die Methode `quicksort()` zwei zusätzliche Parameter, ähnlich wie bei der binären Suche.

```
void quicksort(int[] feld, int start, int ende) {
    // Basisfall: leeres Feld
    if (ende < start) {
        return;
    }
    // Basisfall: maximal 2 Elemente,
    if (ende - start <= 1) {
        // wenn nötig die beiden Werte vertauschen
        if (feld[start] > feld[ende]) {
            int temp = feld[start];
            feld[start] = feld[ende];
            feld[ende] = temp;
        }
        return;
    }
    // Feld aufteilen
    int grenze = aufteilen(feld, start, ende);
    // linken Teil (ohne Pivot) Sortieren
    quicksort(feld, start, grenze - 1);
    // rechten Teil (ohne Pivot) Sortieren
    quicksort(feld, grenze + 1, ende);
}
```

Um das Feld entsprechend aufzuteilen, wählen wir das erste Element als Pivotelement. Anschließend beginnen wir, vom zweiten Element an aufsteigend, ein Element zu suchen, das größer als das Pivotelement ist. Ebenso beginnen wir, vom letzten Element an absteigend, ein Element zu suchen, das kleiner ist als das Pivotelement. Haben wir diese beiden Elemente gefunden, so vertauschen wir sie und suchen von den Positionen aus weiter. Das Vertauschen endet, sobald sich die Suchindizes von links und rechts treffen. Das Element an der Grenze wird anschließend mit dem Pivotelement vertauscht. Dieses Vorgehen ist in Abb. 34-7 dargestellt.

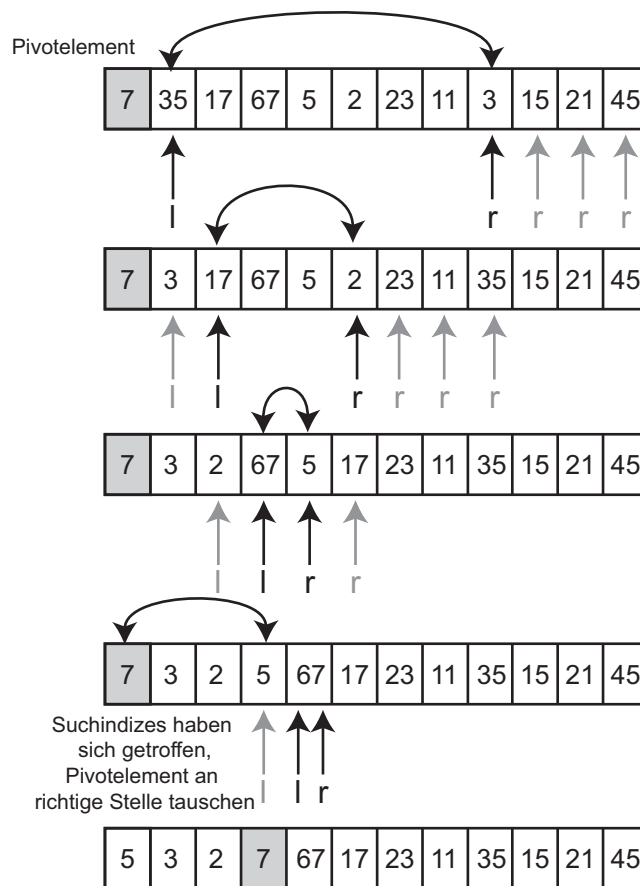


Abb. 34-7: Aufteilen eines Feldes an Hand des Pivotelements

```
// teilt die Elemente auf und liefert die
// Position des Pivotelements zurück
int aufteilen(int[] feld, int start, int ende) {
    // Index von links
    int l = start + 1;
    // Index von rechts
    int r = ende;
    // Pivotelement
    int pivot = feld[start];
    // Umsortierung
    while (l < r) {
        // erstes Element größer als Pivot finden
        while(feld[l] <= pivot && l < r) {
            l++;
        }
        // erstes Element kleiner als Pivot finden
        while(feld[r] > pivot && l < r) {
            r--;
        }
        // Elemente vertauschen
        int temp = feld[l];
        feld[l] = feld[r];
        feld[r] = temp;
    }
}
```

```

// Indizes haben sich getroffen
// prüfen ob Grenze korrekt
if(feld[l] > pivot) {
    // Grenze anpassen
    l--;
}
// Grenze gefunden, Pivot entsprechend vertauschen
feld[start] = feld[l];
feld[l] = pivot;
return l;
}

```

Selbsttestaufgabe 34-12:

Versuchen Sie, die einzelnen Schritte in der Implementierung mit Hilfe des folgenden Beispielaufrufs nachzuvollziehen.

```

int[] x = {12, 2, 6, 1, 8, 34, 10, 7, 20};
quicksort(feld, 0, feld.length - 1);

```

◇

Sortieren durch
Verschmelzen
merge sort

Ein weiteres rekursives Sortierverfahren ist das „Sortieren durch Verschmelzen“ (engl. *merge sort*). Bei diesem Verfahren wird im Gegensatz zu Quicksort nicht beim Aufteilen sortiert, sondern erst wieder beim Zusammenfügen. Das Feld wird in zwei möglichst gleich große Hälften aufgeteilt, die nach dem gleichen Verfahren sortiert werden. Die beiden sortierten Teilfelder werden nun zu einer sortierten Gesamtliste vereint, indem die beiden ersten Elemente verglichen und das kleinere aus seinem Teil entnommen und in das Zielfeld übernommen wird. Das wird solange fortgesetzt, bis beide Teilfelder leer sind. Abb. 34-8 veranschaulicht die Aufteilung und Verschmelzung der Felder.

Das Verschmelzen der beiden Teilfelder $L = \langle l_1, l_2, \dots, l_m \rangle$ und $R = \langle r_1, r_2, \dots, r_n \rangle$ ist in Abb. 34-9 dargestellt und kann folgendermaßen definiert werden:

$$\text{merge}(L, R) = \begin{cases} L & \text{wenn } R \text{ leer ist} \\ R & \text{wenn } L \text{ leer ist} \\ l_1 + \text{merge}(\langle l_2, \dots, l_m \rangle, R) & \text{wenn } l_1 \leq r_1 \\ r_1 + \text{merge}(L, \langle r_2, \dots, r_n \rangle) & \text{sonst} \end{cases}$$

Das Zeichen „+“ soll hier bedeuten, dass das Element vorne in ein Feld eingefügt wird.

Bei einer Implementierung würden wir feststellen, dass das Verschmelzen der Felder sehr mühsam ist, da wir immer wieder neue Felder erzeugen müssten, in die wir die Elemente hinein kopieren. Im folgenden Kapitel 35 werden wir Datenstrukturen kennen lernen, die die dafür notwendigen Operationen, wie zum Beispiel das Einfügen und Löschen von Elementen, das wir beim Verschmelzen benötigen, besser unterstützen.

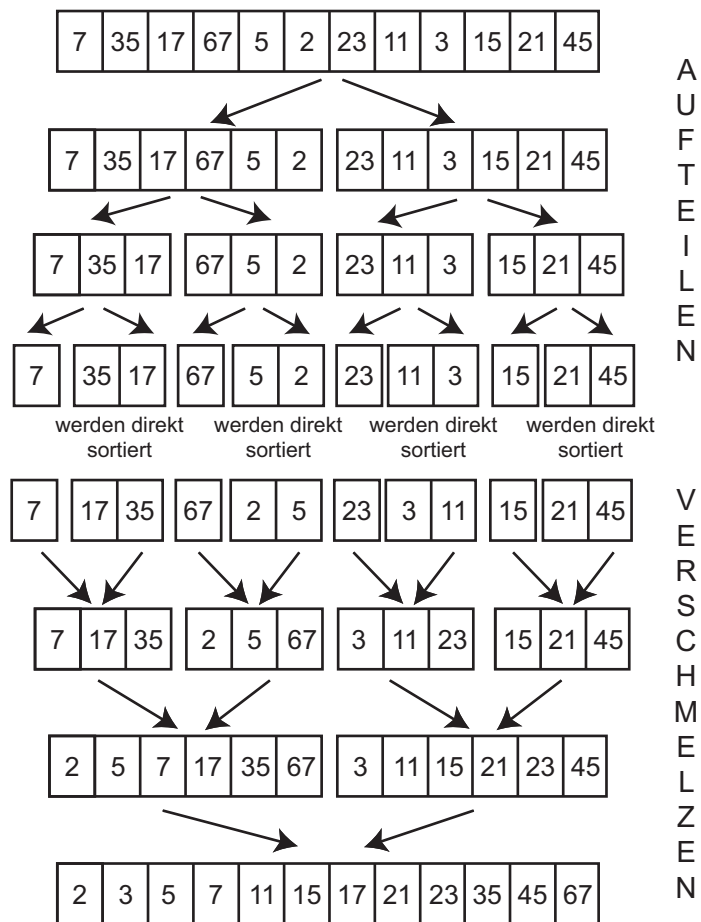


Abb. 34-8: Sortieren durch Verschmelzen (merge sort)

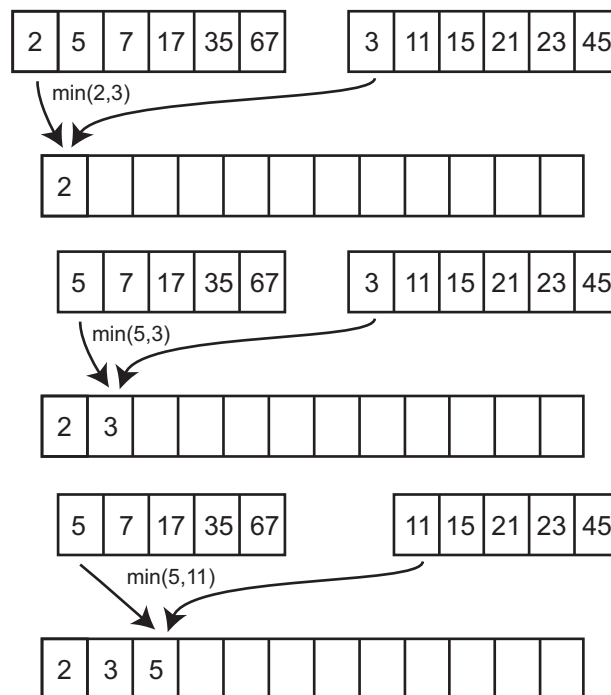


Abb. 34-9: Verschmelzen zweier sortierter Felder

Selbsttestaufgabe 34-13:

Versuchen Sie, das folgende Feld mit Hilfe des Algorithmus „Sortieren durch Verschmelzen“ von Hand zu sortieren.

```
int[] x = {12, 2, 6, 1, 8, 34, 10, 7, 20};
```



35 Listen

Im vorherigen Kapitel haben wir uns mit rekursiven Algorithmen befasst. Im Folgenden lernen wir eine erste rekursive Datenstruktur, die Liste, kennen und entwerfen Algorithmen, die diese Datenstruktur rekursiv bearbeiten. Wir erinnern uns, dass Datenstrukturen eine begriffliche Anschauung mit einer bestimmten Art, Daten im Speicher eines Computers anzuordnen, verbinden.

rekursive Datenstruktur
Liste

Wie rekursive Funktionen erkennt man rekursive Datenstrukturen daran, dass in ihrer Definition ein Selbstbezug auftritt.

35.1 Lineare Datenstrukturen

Die Wahl geeigneter Datenstrukturen ist ein fundamentaler Aspekt des Programm-entwurfs. Die Qualität eines Programmsystems hängt nicht selten von der Festle-gung auf passende Datenstrukturen ab, denn sie bestimmen wesentlich die Struktur der Algorithmen. Zwei Gruppen von Datenstrukturen spielen bei der Programmie-rung eine zentrale Rolle: lineare und graphenartige Datenstrukturen. Graphenartige Datenstrukturen werden wir in Kapitel 36 betrachten.

Lineare Datenstrukturen zeichnen sich dadurch aus, dass die Elemente oder Kno-ten der Datenkollektion in einer Folge angeordnet sind. Den informationstragenden Bestandteil eines Elements nennen wir Eintrag. Eine lineare Datenstruktur kann auch leer sein. Einzelne Einträge können mehrfach in einer linearen Datenstruktur vorkommen.

lineare Datenstruktur
Eintrag

Als typisches Beispiel für eine lineare Datenstruktur haben wir bereits Felder ken-nen gelernt (s. Kapitel 20). Charakteristisch für Felder ist der wahlfreie Zugriff (engl. *random access*), den wir auf die einzelnen Feldelemente haben. Beim wahl-freien Zugriff ist jedes beliebige Element einer Kollektion mit dem gleichen Auf-wand (z. B. Zugriffszeit) erreichbar.

wahlfreier Zugriff

Die Elemente einer linearen Datenstruktur können entweder durch einen wahlfrei-en Zugriff oder durch einen sequenziellen Zugriff (engl. *sequential access*) erreicht werden. Beim sequenziellen Zugriff wird auf die Elemente einer Kollektion in einer vorbestimmten, geordneten Abfolge zugegriffen, so dass der Aufwand für verschie-dene Elemente einer Kollektion variiert.

sequenzieller Zugriff

Weitere Beispiele für linearen Datenstrukturen sind der Stapel (engl. *stack*) und die Warteschlange (engl. *queue*). Sie erlauben keinen wahlfreien, sondern nur einen sequenziellen Zugriff.

Stapel
Warteschlange

Die Metapher des Stapels ist uns in Kapitel 34 im Zusammenhang mit dem Metho-denstapel begegnet. Charakteristisch für die Datenstruktur Stapel ist, dass immer das zuletzt hinzugefügte Element als erstes vom Stapel entnommen werden muss.

Last In First Out	Diese Eigenschaft ist unter dem Namen <i>Last In First Out</i> (LIFO) bekannt. Die Warteschlange hingegen arbeitet nach dem <i>First In First Out</i> -Prinzip (FIFO). Es kann immer das Element aus der Warteschlange entnommen werden, das als erstes eingefügt worden ist, sich also schon am längsten in der Warteschlange befindet.
First In First Out	

35.2 Verkettete Listen

Felder kennen wir bereits als effiziente Datenstrukturen, die Java bereitstellt, um Folgen einfacher Werte oder Objektreferenzen zu organisieren und wahlfrei auf diese zuzugreifen. Der Preis für diese Effizienz ist jedoch, dass die Größe eines Feldes bei seiner Erzeugung festgelegt wird und während der Lebensdauer eines Feldobjekts nicht mehr geändert werden kann. Nun gibt es aber viele Anwendungen, bei denen wir nicht im Vorhinein wissen, wie viele Elemente unsere Kollektion enthalten wird. So wissen wir beispielsweise im Blumengeschäft nicht, wie viele Rechnungen im Laufe eines bestimmten Zeitraums ausgestellt werden oder wieviele Kunden unsere Kundenkartei einmal umfassen könnte. Wir wollen aber dennoch die Gesamtheit der Rechnungen oder Kunden effizient verwalten.

Im täglichen Leben benutzen wir häufig Listen, um z. B. die Teilnehmerinnen eines Kurses zu verwalten, anstehende Einkäufe festzuhalten oder die einzelnen Abschnitte einer Wegbeschreibung zu notieren. Wir können die Liste einfach wachsen lassen, indem wir am Ende oder vor den Anfang der Liste oder aber an beliebiger Stelle zwischen zwei vorhandenen Elementen ein neues hinzufügen, und wir können die Liste schrumpfen lassen, indem wir einfach ein Element streichen.

verkettete Liste	Ein Abbild dieser Struktur in Programmen ist die verkettete Liste (engl. <i>linked list</i>). Eine verkettete Liste ist die einfachste Form einer Sammlung von Datenobjekten, denen eine lineare Ordnung aufgeprägt ist.
------------------	---

Listen sind dynamische Datenstrukturen, die bestimmte Eigenschaften haben:

- | | |
|--------|--|
| Knoten | <ol style="list-style-type: none"> 1. Die einzelnen Knoten (engl. <i>node</i>) oder Elemente der Liste sind geordnet. 2. Listen können beliebig lang werden. 3. Listen können auch leer sein. 4. Man kann Knoten aus einer Liste löschen oder neue Knoten hinzufügen, ohne die bisherigen Knoten umordnen zu müssen. |
|--------|--|

Felder sind am besten geeignet, wenn man eine fest vorgegebene Menge von Dateneinträgen zu speichern hat, auf die man wahlfrei zugreifen möchte. Im Gegensatz dazu hat eine verkettete Liste keine vorherbestimmte, festgelegte Größe. Eine Liste kann dynamisch wachsen oder schrumpfen. Eine Liste verwendet Speicherplatz im Verhältnis zur Anzahl ihrer aktuellen Knoten. Auf die Listenknoten kann nur sequenziell zugegriffen werden. Listen verwendet man besser für Datenkollektionen, deren Größe man nicht vorherbestimmen kann und die sich häufig ändern.

Definition 35.2-1: Verkettete Liste

Eine einfach verkettete Liste ist eine (möglicherweise leere) Menge von Knoten, wobei jeder Knoten einen Eintrag und einen Verweis auf den nächsten Knoten enthält.

einfach verkettete Liste

Der erste Knoten einer Liste wird Kopf (engl. head), der letzte wird Schwanz (engl. tail) genannt. Der Verweis des letzten Elements hat, außer bei einer zirkulären Liste²², den Wert null.

Kopf

Schwanz

┘

Wir können die für das Datenlexikon in Abschnitt 2.7 eingeführte Schreibweise verwenden, um Listen rekursiv zu definieren:

$$\text{Liste} = \text{leereListe} \mid \text{Knoten Liste}$$

Die Knoten einer verketteten Liste können in Java wie folgt definiert werden:

```
public class ListNode {

    private int entry;
    private ListNode next;

    // ...
}
```

Das Attribut `entry` dient zur Speicherung des Eintrags, der in unserer Knoten-Implementierung vom Typ `int` ist. Das Attribut `next` stellt eine Referenz auf den nachfolgenden Listenknoten dar. Abb. 35.2-1 zeigt schematisch einen Knoten mit einem Eintrag und einer Referenz auf einen weiteren Knoten.

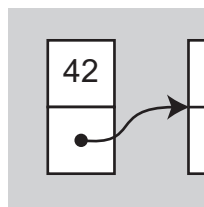


Abb. 35.2-1: Knoten mit Eintrag und Referenz

Um mit verketteten Listen in Java arbeiten zu können, benötigen wir geeignete Methoden, die es uns ermöglichen, in geordneter Weise über die Liste zu laufen und dabei bestimmte Veränderungen zu bewirken oder Information aufzusammeln. Dazu werden wir die Klasse `LinkedList` entwickeln.

Zunächst erweitern wir die Definition der Klasse `ListNode` noch um Setter- und Getter-Methoden und geeignete Konstruktoren.

22 Bei einer zirkulären Liste enthält das Schwanzelement einen Verweis auf das Kopfelement.

```
public class ListNode {

    private int entry;
    private ListNode next;

    public ListNode(int value) {
        this(value, null);
    }

    public ListNode(int value, ListNode nextNode) {
        this.entry = value;
        this.next = nextNode;
    }

    public void setEntry(int value) {
        this.entry = value;
    }

    public void setNext(ListNode nextNode) {
        this.next = nextNode;
    }

    public int getEntry() {
        return this.entry;
    }

    public ListNode getNext() {
        return this.next;
    }
}
```

Für einen Knoten, dem kein weiterer Knoten mehr folgt, ist als Argument nur ein `int`-Wert als Eintrag anzugeben; sein Attribut `next` erhält eine `null`-Referenz. Für alle anderen Knoten kann zusätzlich eine Referenz auf den Nachfolger als Argument übergeben werden.

Selbsttestaufgabe 35.2-1:

Erweitern Sie die Klasse `ListNode` um eine Methode `print()`, die den Eintrag des Knotens ausdrückt.



Nun wenden wir uns der Klasse `LinkedList` zu, die wir nach und nach mit Methoden zur Handhabung von Knoten in verketteten Listen ausrüsten werden.

```
public class LinkedList {

    private ListNode head;

    public LinkedList() {
        this.head = null;
    }
}
```

```

public void add(int value) {
    ListNode newNode = new ListNode(value, this.head);
    this.head = newNode;
}

// ...
}

```

Ein Objekt der Klasse `LinkedList` hat als einziges Attribut einen Listenkopf `head` vom Typ `ListNode`. Nach der Erzeugung eines `LinkedList`-Objekts enthält dieses noch keine Knoten, so dass der Listenkopf `head` eine `null`-Referenz darstellt. Abb. 35.2-2 zeigt schematisch eine leere Liste.

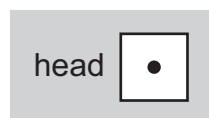


Abb. 35.2-2: Eine leere Liste

Wir können eine verkettete Liste durch das sukzessive Einfügen von Knoten am Anfang der Liste aufbauen. Die Methode `add()`

1. erzeugt einen neuen Listenknoten,
2. trägt den Wert des Eintrags ein,
3. lässt das `next`-Attribut dieses Knotens auf den Kopf der Liste zeigen und
4. setzt den Kopf der Liste auf den neu erzeugten Listenknoten.

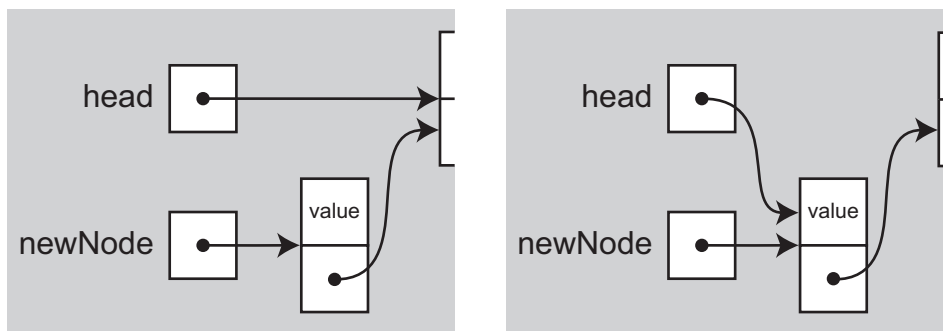


Abb. 35.2-3: Einfügen eines Knotens

Abb. 35.2-3 zeigt das Einfügen eines Knotens am Anfang einer nicht-leeren Liste. Das Attribut `head` verweist immer auf den ersten Knoten der Liste. Um eine Liste mit zwei Einträgen zu erhalten, gehen wir wie folgt vor:

```

LinkedList liste = new LinkedList(); // 1
liste.add(47); // 2
liste.add(11); // 3

```

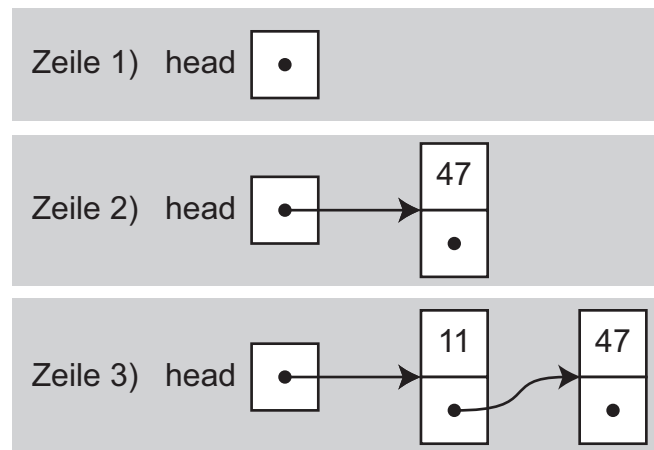


Abb. 35.2-4: Liste nach Ausführung der Zeilen 1, 2 und 3

Abb. 35.2-4 zeigt schematisch den Zustand der Liste jeweils nach Ausführung der Zeilen 1, 2 und 3.

Um die Länge einer verketteten Liste zu ermitteln, ergänzen wir die Methode `size()`.

```
public int size() {
    ListNode current = this.head;
    int count = 0;
    while (current != null) {
        count++;
        current = current.getNext();
    }
    return count;
}
```

Die Methode `size()` zeigt ein typisches Muster, um über eine verkettete Liste zu iterieren. Solange nicht das Ende der Liste erreicht ist (`current != null`), wird eine Aktion auf den betrachteten Knoten (`current`) angewendet, und anschließend der nächste Knoten (`current.getNext()`) aufgesucht.

Wird die Methode `size()` auf unsere oben erstellte zweielementige Liste angewendet, so verweist die lokale Variable `current` zunächst auf den Listenkopf (der das Element 11 enthält) und handelt sich im ersten Durchlauf der `while`-Schleife zum zweiten Listenknoten (der das Element 47 enthält) vor. Dessen `null`-Referenz übernimmt sie im zweiten Durchlauf der `while`-Schleife, worauf diese beendet wird.

Selbsttestaufgabe 35.2-2:

Formulieren Sie die Methode `size()` mit Hilfe einer `for`-Schleife.



Nach demselben Muster können wir beispielsweise auch eine Suche auf verketteten Listen definieren:

```

public boolean contains(int value) {
    ListNode current = this.head;
    while (current != null) {
        if (current.getEntry() == value) {
            return true;
        }
        current = current.getNext();
    }
    return false;
}

```

Bemerkung 35.2-1:

Beachten Sie die Ähnlichkeit dieser Methode mit der Suche in Feldern (Abschnitt 33.1). Die lineare Suche lässt sich auf Listen einfach realisieren. ┘

Selbsttestaufgabe 35.2-3:

Überlegen sie, ob Listen auch für die binäre Suche geeignet sind. ◇

Selbsttestaufgabe 35.2-4:

Schreiben Sie eine Methode `sum()`, die iterativ die Einträge der Listenknoten aufsummiert und die Summe zurück gibt.

Lösungshinweis: Orientieren Sie sich am Muster der `size()`-Methode. ◇

Wir implementieren nun die Methode `size()` erneut, diesmal rekursiv:

```

public int size() {
    return size(this.head);
}

private int size(ListNode node) {    // 1
    if (node == null) {              // 2
        return 0;                    // 3
    }                                 // 4
    return size(node.getNext()) + 1; // 5
}

```

Beachten Sie das Zusammenspiel der öffentlichen und der privaten `size()`-Methoden. Die öffentliche Methode ruft mit dem Listenkopf die private Methode auf, die rekursiv die eigentliche Arbeit erledigt. Dieses Muster ist uns im Zusammenhang mit rekursiven Methoden bereits bekannt (Kapitel 34) und wird uns auch bei rekursiven Datenstrukturen oft begegnen.

Wir verwenden wieder die oben erstellte zweielementige Liste, um uns zu vergegenwärtigen, was nach dem Aufruf von

```
liste.size();
```

geschieht. Zunächst ruft die öffentliche Methode die private Methode auf und übergibt den (von head referenzierten) Listenkopf als Parameter. Die private Methode prüft zunächst, ob der Basisfall vorliegt (Zeile 2). Da der aktuelle Parameter auf einen Knoten verweist, erfolgt in Zeile 5 der nächste Aufruf der `size()`-Methode mit dessen Nachfolger. Der Parameter ist nun der zweite Knoten der Liste. Wieder liegt kein Basisfall vor, da auch dieser Parameter auf einen Knoten verweist. Es erfolgt ein weiterer Aufruf der `size()`-Methode, doch nun wird als Parameter eine `null`-Referenz übergeben. Der Basisfall ist erreicht, und die zuletzt aufgerufene `size()`-Methode liefert den Rückgabewert 0 an den Aufrufer zurück. Während der Rückabwicklung der Rekursion addiert jede zuvor aufgerufene `size()`-Methode den Wert 1, bis schließlich die Gesamtlänge 2 an die öffentliche `size()`-Methode und von dieser an ihren Aufrufer zurückgeliefert wird.

Selbsttestaufgabe 35.2-5:

Schreiben Sie ein Methodenpaar `sum()`, das die Einträge in den Listenknoten rekursiv aufsummiert:

```
public int sum() {
    // ...
}

private int sum(ListNode node) {
    // ...
}
```



Die folgende rekursive Methode gibt die Einträge der Listenknoten am Bildschirm aus:

```
public void printList() {
    printList(this.head);
}

private void printList(ListNode node) {
    if (node == null) {
        return;
    }
    node.print();
    System.out.print(" ");
    printList(node.getNext());
}
```


Selbsttestaufgabe 35.2-6:

Schreiben Sie ein Methodenpaar `printReverseList()`, das die Einträge der Listenknoten in umgekehrter Reihenfolge druckt:

```
public void printReverseList() {
    // ...
}

private void printReverseList(ListNode node) {
    // ...
}
```

Lösungshinweis: Sie müssen die Methode rekursiv aufrufen, bis Sie den Schwanz der Liste erreicht haben. Während die rekursiven Aufrufe rückabgewickelt werden, können Sie die Werte ausdrucken. ◇

Eine Liste ist ein dynamisches Objekt, das es uns auch erlaubt, Knoten zu löschen oder neue Knoten an einer bestimmten Position einzufügen. Wenn wir einen bestimmten Knoten entfernen wollen, müssen wir zwei Aufgaben lösen:

1. Wir müssen den Knoten, der entfernt werden soll, in der verketteten Liste finden, und
2. wir müssen dafür sorgen, dass die dem gesuchten Knoten benachbarten Knoten wieder so miteinander verkettet sind, dass ihre ursprüngliche Reihenfolge erhalten bleibt.

Abb. 35.2-5 deutet an, dass wir den Listenknoten mit dem Eintrag 5 aus der Liste entfernen möchten.

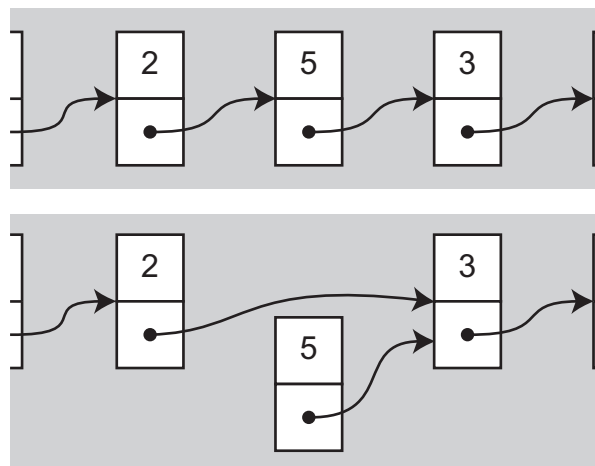


Abb. 35.2-5: Entfernen eines Listenknotens

Als Ergebnis des Löschvorgangs muss das `next`-Attribut des ehemaligen Vorgängers des entfernten Knotens auf den Nachfolger des entfernten Knotens verweisen.

Wir definieren eine rekursive Methode `remove()`, die einen Knoten `node` und einen ganzzahligen Wert `value` als Argumente akzeptiert. Die prinzipielle Idee ist die, dass die Methode rekursiv durch die Liste läuft, um den Knoten mit dem Eintrag `value` zu finden. Wenn solch ein Knoten gefunden werden kann, wird er entfernt und sein Nachfolger wird zurückgegeben. Falls der Wert in keinem Knoten vorkommt, bleibt die Liste unverändert.

```
public void remove(int value) {
    this.head = remove(this.head, value);
}

private ListNode remove(ListNode node, int value) { // 1
    if (node == null) { // 2
        return null; // 3
    } // 4
    if (node.getEntry() == value) { // 5
        return node.getNext(); // 6
    } // 7
    node.setNext(remove(node.getNext(), value)); // 8
    return node; // 9
}
```

Erneut haben wir eine öffentliche Methode und einen nicht-öffentlichen rekursiven Algorithmus definiert. Beachten Sie, dass hier zwei Basisfälle vorliegen:

1. Ein Basisfall tritt auf, wenn kein passender Knoten in der Liste gefunden wurde oder wenn die Liste leer ist (Zeile 2).
2. Der zweite Basisfall liegt vor, wenn der Eintrag eines Knotens in der Liste mit dem gesuchten Wert, der als zweiter Parameter (`value`) beim Aufruf der Methode übergeben wurde, übereinstimmt (Zeile 5).

Falls keiner dieser Basisfälle auftritt, wird die Methode `remove()` rekursiv auf den Rest der Liste angewendet (Zeile 8). Das Ergebnis jedes rekursiven Aufrufs ist entweder die ursprüngliche Restliste oder die um den gesuchten Knoten verringerte Restliste. Wir müssen aber auf jeden Fall das Ergebnis mit dem Knoten, der auf die ursprüngliche Restliste verwies, über das `next`-Attribut verknüpfen (Zeile 8), um bei der Rückabwicklung der rekursiven Aufrufe die sich aus den Basisfällen ergebende Restliste mit dem unveränderten Vorspann der Ursprungsliste Knoten für Knoten wieder zu verknüpfen.

Abb. 35.2-6 veranschaulicht die rekursive Zerlegung einer Liste.

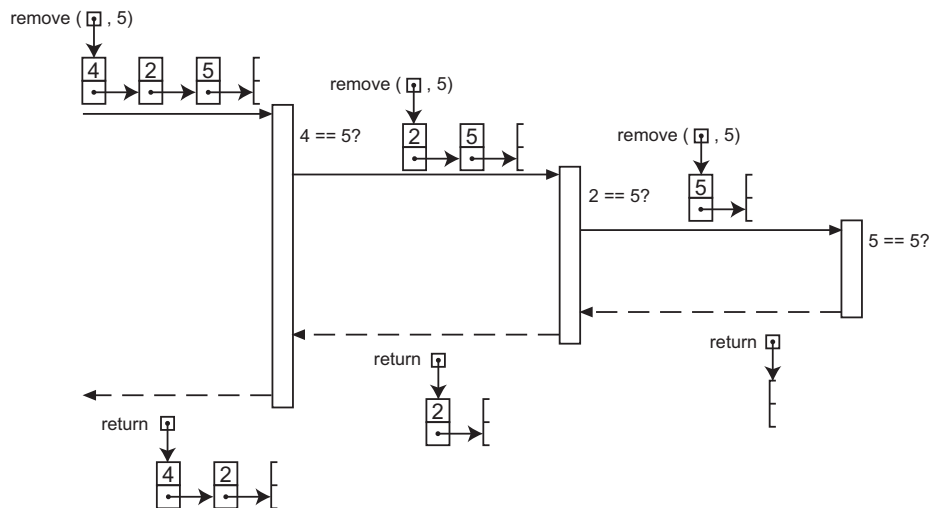


Abb. 35.2-6: Rekursives Löschen

Selbsttestaufgabe 35.2-7:

Was würde passieren, wenn wir Zeile 8 wie folgt:

```
remove(node.getNext(), value); // 8
```

änderten und

1. Methode `remove(5)` auf die Liste $l = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$ anwenden?
 - a) Die Liste l bleibt unverändert.
 - b) Die Liste l enthält nur noch die Elemente $\langle 6, 7, 8 \rangle$.
 - c) Die Liste l ist leer.
2. Was wäre das Ergebnis des Aufrufs `remove(0)`?

**Selbsttestaufgabe 35.2-8:**

Schreiben Sie ein Methodenpaar `contains()`, das rekursiv prüft, ob in der Liste ein Knoten mit einem bestimmten Wert vorhanden ist:

```
public boolean contains(int value) {
    // ...
}

private boolean contains(...) {
    // ...
}
```



Selbsttestaufgabe 35.2-9:

Beschreiben Sie den Basisfall in Selbsttestaufgabe 35.2-8 und erörtern Sie, warum die Rekursion konvergiert.



Nicht immer möchte man neue Einträge am Anfang einer Liste einfügen, wie es bisher durch die Methode `add()` geschieht. Wir wollen die `add()`-Methode überladen, um die Möglichkeit zu schaffen, einen Knoten mit einem neuen Eintrag an einer gewählten Stelle `index` in Listen einzufügen.

```
public void add(int index, int value) {
    if (index < 0 || index > size()) {
        throw new IndexOutOfBoundsException("index muss im "
            + "Bereich von 0 bis " + size() + " liegen.");
    }
    this.head = add(this.head, index, value);
}

private ListNode add(ListNode node,
                    int steps, int value) {           // 1
    if (steps == 0) {                                 // 2
        return new ListNode(value, node);             // 3
    }                                                  // 4
    node.setNext(add(node.getNext(), steps - 1, value)); // 5
    return node;                                     // 6
}
```

Analog zu Abb. 35.2-6 zeigt Abb. 35.2-7 das Einfügen eines Listenknotens. Das `next`-Attribut des neuen Knotens verweist auf seinen zukünftigen Nachfolger (Schritt 1). Der bisherige Vorgänger des Nachfolgers wird zum Vorgänger des neuen Knotens (Schritt 2).

Die private rekursive `add()`-Methode bewältigt mehrere Aufgaben:

- Zunächst muss sie sich rekursiv an die Einfügestelle heran zählen (`steps - 1` in Zeile 5).
- Ist die Einfügestelle erreicht (Zeile 2), muss ein neuer Knoten mit dem erwünschten Eintrag und einer Referenz auf den Nachfolgerknoten erzeugt werden (Zeile 3). Dies ist der Basisfall.
- Das Ergebnis jedes rekursiven Aufrufs ist eine Teilliste, die mit dem Knoten, der auf die ursprüngliche Teilliste verwies, durch `setNext()` verknüpft wird (Zeile 5). Hier erfolgt auch die Verkettung des Vorgängerknotens mit dem neuen Knoten.

Die öffentlichen `add()`-Methode hat vor dem Aufruf der privaten `add()`-Methode noch die wichtige Aufgabe, zu prüfen, ob die im Parameter `index` übergebene Einfügestelle zulässig ist.

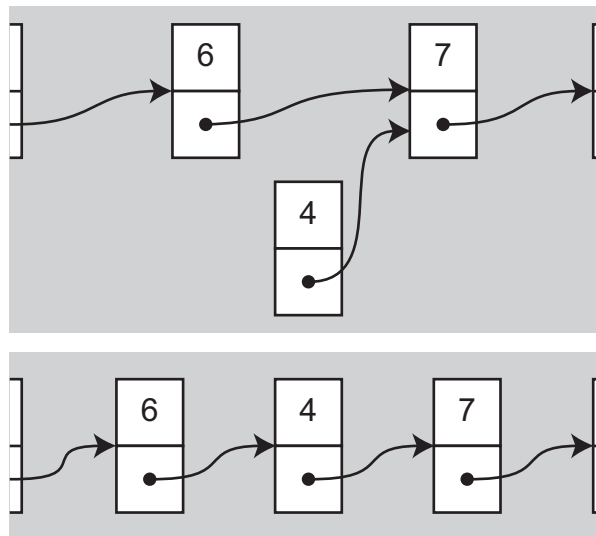


Abb. 35.2-7: Einfügen eines Listenknotens

Verkettete Listen können viele weitere Methoden haben. So könnte man z. B. den Eintrag an einer bestimmten Position abrufen, die Liste leeren, eine Teilliste erzeugen wollen und vieles mehr.

Selbsttestaufgabe 35.2-10:

Implementieren sie eine Methode, um den Eintrag an einer bestimmten Position abzurufen (ohne die Liste zu verändern):

```
public int get(int index) {
    // ...
}
```

Bedenken Sie den Fall, dass die Methode mit einem ungültigen Index als Parameter aufgerufen werden könnte. Entwerfen Sie je eine iterative und eine rekursive Lösung.



Selbsttestaufgabe 35.2-11:

Implementieren Sie eine Methode, um die Liste zu leeren.

```
public void clear() {
    // ...
}
```



Selbsttestaufgabe 35.2-12:

Implementieren Sie eine Methode, die angibt, ob die Liste leer ist.

```
public boolean isEmpty() {
    // ...
}
```



Selbsttestaufgabe 35.2-13:

Implementieren Sie eine Methode, die eine Teilliste zurück liefert.

```
public LinkedList subList(int fromIndex, int toIndex) {
    // ...
}
```

Die Teilliste soll alle Einträge zwischen fromIndex (einschließlich) und toIndex (ausschließlich) in der richtigen Reihenfolge enthalten. Sind fromIndex und toIndex identisch, so ist die resultierende Liste leer. Bedenken Sie den Fall, dass die Methode mit ungültigen Indizes als Parameter aufgerufen werden könnte. Als ungültig ist auch der Fall fromIndex > toIndex zu werten.

Lösungshinweis: Verwenden Sie Kopien der Einträge im gewünschten Listenabschnitt und erzeugen Sie mit diesen eine neue Liste. (Implementierungen, die ohne Kopien arbeiten, sind möglich, aber wesentlich aufwändiger zu realisieren.) ◇

Zum Abschluss unserer Betrachtungen über einfach verkettete Listen kommen wir auf den Sortier-Algorithmus „Merge-Sort“ (vgl. Kapitel 34) zurück. Nehmen wir an, uns liegt eine umfangreiche unsortierte Liste vom Typ `LinkedList` vor. Diese soll in eine sortierte Liste vom Typ `LinkedList` überführt werden. Der Algorithmus „Merge-Sort“ lässt sich für unseren Typ `LinkedList` wie folgt implementieren:

```
public LinkedList mergeSort() {
    int length = this.size();
    if (length <= 1) {
        return this;
    }
    LinkedList leftSorted
        = this.subList(0, length / 2).mergeSort();
    LinkedList rightSorted
        = this.subList(length / 2, length).mergeSort();
    leftSorted.mergeWith(rightSorted);
    return leftSorted;
}

private void mergeWith(LinkedList otherList) {
    if (otherList.isEmpty()) {
        return;
    }
    if (this.isEmpty()) {
        this.head = otherList.head;
        return;
    }
}
```

```

        if (otherList.head.getEntry() <= this.head.getEntry()) {
            int first = otherList.removeFirst();
            this.mergeWith(otherList);
            this.add(first);
            return;
        }
        int first = this.removeFirst();
        otherList.mergeWith(this);
        otherList.add(first);
        this.head = otherList.head;
    }

    private int removeFirst() {
        if (this.head == null) {
            throw new IndexOutOfBoundsException("leere Liste");
        }
        int first = this.head.getEntry();
        this.head = this.head.getNext();
        return first;
    }

```

Vergleichen Sie die Implementierung mit der Beschreibung des Sortier-Algorithmus „Merge-Sort“ in Kapitel 34.

35.3 Spezielle Listen

Für viele Verwendungen sind sortierte Listen besonders effizient. Beispielsweise kann die Suche nach einem Eintrag in einer sortierten Liste oft vor dem Erreichen des Listenendes abgebrochen werden, was insbesondere bei langen Listen ins Gewicht fällt. Stellen wir uns eine aufsteigend sortierte Liste mit `int`-Einträgen vor. Wenn wir beispielsweise feststellen möchten, ob der Wert 123 in der Liste vorkommt, können wir unsere Suche abbrechen, sobald wir einen Eintrag mit einem Wert größer als 123 antreffen. sortierte Liste

Um eine aufsteigend sortierte Liste zu erhalten, können wir einen Sortier-Algorithmus wie „Merge-Sort“ anwenden, oder aber einen Listentyp konzipieren, der von vornherein und jederzeit sortiert ist.

Eine aufsteigend sortierte Liste für `int`-Werte können wir ähnlich wie die Klasse `LinkedList` definieren. Um zu gewährleisten, dass die Liste jederzeit sortiert ist, müssen wir dafür sorgen, dass Einträge von Anfang an gemäß der gewählten Ordnung in die Liste eingefügt werden.

```

public class SortedList {

    private ListNode head;

    public SortedList() {
        this.head = null;
    }

```

```

public void add(int value) {
    this.head = add(this.head, value);
}

private ListNode add(ListNode node, int value) {
    if (node == null) {
        return new ListNode(value, node);
    }
    if (node.getEntry() > value) {
        return new ListNode(value, node);
    }
    node.setNext(add(node.getNext(), value));
    return node;
}

// ...
}

```

Die Methode `add()` fügt jeden neuen Eintrag an der richtigen Stelle in die sortierte Liste ein.

Selbsttestaufgabe 35.3-1:

Zeichnen Sie für die sortierte Liste $sorted = \langle 2, 3, 8 \rangle$ ein Aufrufdiagramm für den Aufruf `add(5)` im Stil von Abb. 35.2-6.



Einige Methoden der Klasse `LinkedList` aus dem vorherigen Abschnitt könnten wir identisch für die Klasse `SortedList` definieren, beispielsweise `printList()` oder `size()`. Andere typische Listenmethoden sollten wir besser nicht identisch von der Klasse `LinkedList` übernehmen, da wir sonst nicht vom möglichen Effizienzvorteil sortierter Listen profitieren. Dies betrifft die Methoden `remove()` und `contains()`.

Selbsttestaufgabe 35.3-2:

Welche Methoden der Klasse `LinkedList` könnten wir ohne Nachteil für `SortedList` identisch übernehmen?



Selbsttestaufgabe 35.3-3:

Was hält uns davon ab, die Klasse `LinkedList` zu spezialisieren?



Um den Effizienzvorteil sortierter Listen nutzen zu können, wollen wir nun eine Methode `remove()` speziell für eine aufsteigend sortierte Liste definieren.


```

public void remove(int value) {
    this.head = remove(this.head, value);
}

private ListNode remove(ListNode node, int value) { // 1
    if (node == null) { // 2
        return null; // 3
    } // 4
    if (node.getEntry() > value) { // 5
        return node; // 6
    } // 7
    if (node.getEntry() == value) { // 8
        return node.getNext(); // 9
    } // 10
    node.setNext(remove(node.getNext(), value)); // 11
    return node; // 12
}

```

Die Effizienz ergibt sich aus dem zweiten Basisfall (Zeile 5). Sobald bei der rekursiven Ausführung ein Eintrag angetroffen wird, der größer als `value` ist, steht fest, dass ein Eintrag mit dem Wert `value` auch in der restlichen Liste nicht mehr gefunden wird.

Selbsttestaufgabe 35.3-4:

Welche weiteren Unterschiede bestehen zwischen den `remove()`-Methoden der Klassen `LinkedList` und `SortedList`?



Selbsttestaufgabe 35.3-5:

Implementieren Sie eine effiziente Methode `contains()` für `SortedList`, die prüft, ob in der Liste ein Knoten mit einem bestimmten Wert vorhanden ist.



Zum Abschluss dieses Kapitels betrachten wir noch einen weiteren speziellen Listentyp, die doppelt verkettete Liste. Für einige Abstraktionen ist eine einfach verkettete Liste nicht das optimale Modell. Wenn z. B. eine Warteschlange mit Hilfe einer verketteten Liste implementiert werden soll, die das Einfügen am Ende und das Entfernen am Anfang der Schlange unterstützt, können diese Methoden effizienter implementiert werden, wenn die Elemente der Liste in beide Richtungen, also vorwärts und rückwärts, verkettet sind.

doppelt verkettete Liste

Eine Knoten einer doppelt verketteten Liste kann ähnlich wie der uns bekannte `ListNode` definiert werden. Für die Einträge wählen wir diesmal den Typ `String`.

```
public class DoublyLinkedListNode {  
    private String entry;  
    private DoublyLinkedListNode next;  
    private DoublyLinkedListNode prev;  
  
    // ...  
}
```

Eine Warteschlange stellt typischerweise zwei Methoden `enqueue()` zum Hinzufügen eines Eintrags und `dequeue()` zum Entfernen eines Eintrags bereit. Sie arbeitet nach dem *First In First Out*-Prinzip, so dass mit `dequeue()` immer der Eintrag aus der Warteschlange zurückgegeben wird, der sich am längsten in der Warteschlange befindet.

Selbsttestaufgabe 35.3-6:

Ergänzen Sie den Typ `DoublyLinkedListNode` um die benötigten Konstruktoren und Methoden. Entwerfen Sie eine Klasse `DoublyLinkedList` mit den Methoden `enqueue()` zum Hinzufügen eines String-Eintrags am Ende der Liste und `dequeue()` zum Entfernen eines String-Eintrags am Anfang der Liste. ◇

36 Graphen und Bäume

Bei einigen Problemen reichen lineare Datenstrukturen nicht mehr aus. Wollen wir beispielsweise ein Straßennetz oder die Struktur eines Moleküls nachbilden, so benötigen wir Elemente, die mit mehreren anderen Elementen verbunden sein können.

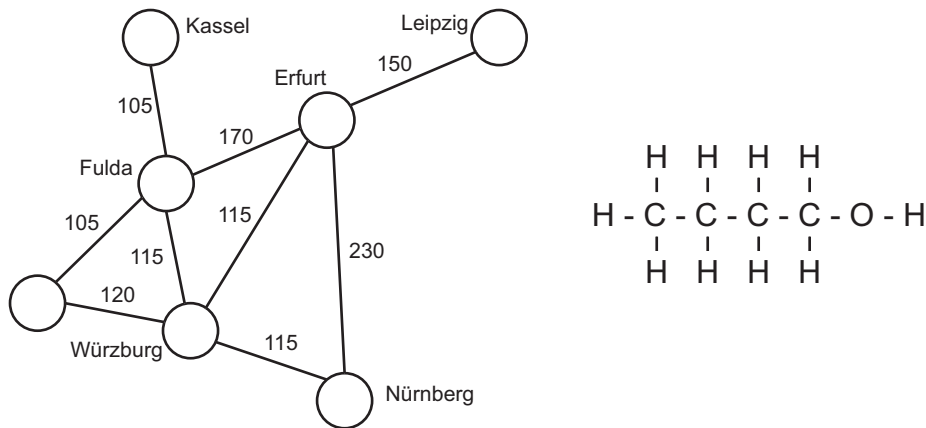


Abb. 36-1: Straßenkarte und Molekül

Für solche Probleme werden häufig Graphen, insbesondere verschiedene Varianten von Bäumen, eingesetzt. Zunächst werden wir Graphen kennen lernen und uns anschließend den Besonderheiten von Bäumen und ihren Implementierungen zuwenden. Zum Abschluss werden wir uns mit der Suche in Graphen beschäftigen.

36.1 Graphen

In der Mathematik, genauer in der Graphentheorie, ist ein Graph eine Struktur, die aus einer Menge von Knoten (engl. *node*) und einer Menge von Kanten (engl. *edge*) besteht. Eine Kante verbindet ein Paar von Knoten.

Graph
Knoten
Kante

Definition 36.1-1: Graph

Ein Graph G ist ein Tupel:

$$G = (V, E)$$

wobei

- V die Menge der Knoten und
- $E \subseteq V \times V$ die Menge der Kanten ist, wobei eine Kante ein Paar von Knoten ist.

┘

gerichteter Graph Man unterscheidet zwischen gerichteten Graphen, wo Kanten ein geordnetes Paar (a, b) sind und eine Kante vom Startknoten a zum Zielknoten b verläuft, und ungerichteten Graphen, wo die Kanten keine Richtung aufweisen. Gerichtete Kanten werden mit Pfeilspitzen dargestellt. Berücksichtigt man bei einem gerichteten Graphen die Richtung der Kanten nicht, so erhält man den zugehörigen ungerichteten Graphen (siehe Abb. 36.1-1).

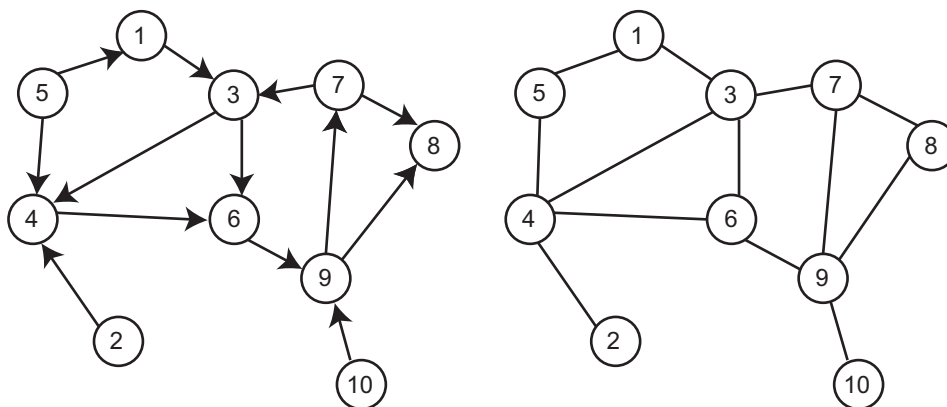


Abb. 36.1-1: Ein gerichteter Graph (links) und der zugehörige ungerichtete Graph (rechts)

eingehende Kanten Bei einem gerichteten Graphen spricht man von ein- und ausgehenden Kanten. Die eingehenden Kanten eines Knotens v sind diejenigen Kanten, die v als Zielknoten besitzen. Die ausgehenden Kanten besitzen v als Startknoten. Die Anzahl der eingehenden Kanten eines Knoten wird als Eingangsgrad bezeichnet, die Anzahl der ausgehenden Kanten als Ausgangsgrad.

Grad Bei ungerichteten Graphen spricht man lediglich vom Grad eines Knotens, der identisch mit der Anzahl allen Kanten ist, die diesen Knoten enthalten.

Knoten und Kanten können zusätzliche Eigenschaften aufweisen. So können in Knoten zum Beispiel Namen oder Werte und bei Kanten die Länge oder das Gewicht gespeichert werden. Bei einem Straßennetz würde in den Knoten beispielsweise der Name der Stadt und bei Kanten die Entfernung zwischen den beiden Städten, die durch diese Kante verbunden sind, gespeichert werden. Die gespeicherten Daten eines Knotens oder einer Kante werden auch als Label bezeichnet.

Selbsttestaufgabe 36.1-1:

Gegeben sei der folgende gerichtete Graph:

$$G = (\{1, 2, 4, 6, 9, 11\}, \{(1, 11), (1, 6), (2, 4), (4, 6), (9, 11)\})$$

Die Knoten sind jeweils durch ihr Label dargestellt. Zeichnen Sie den zugehörigen Graphen.



gerichteter Pfad Ein gerichteter Pfad zwischen zwei Knoten ist eine Folge von Kanten, wobei der

Zielknoten jeder Kante mit dem Startknoten der nachfolgenden Kante identisch ist. Der Startknoten der ersten Kante ist der Anfang des Pfades und der Zielknoten der letzten Kante das Ende des Pfades. Berücksichtigt man bei einem Pfad die Richtung der Kanten nicht, erhält man einen ungerichteten Pfad.

ungerichteter Pfad

Selbsttestaufgabe 36.1-2:

Bestimmen Sie den Pfad mit der minimalen Entfernung zwischen Dresden und Karlsruhe an Hand der Abb. 36.1-2.

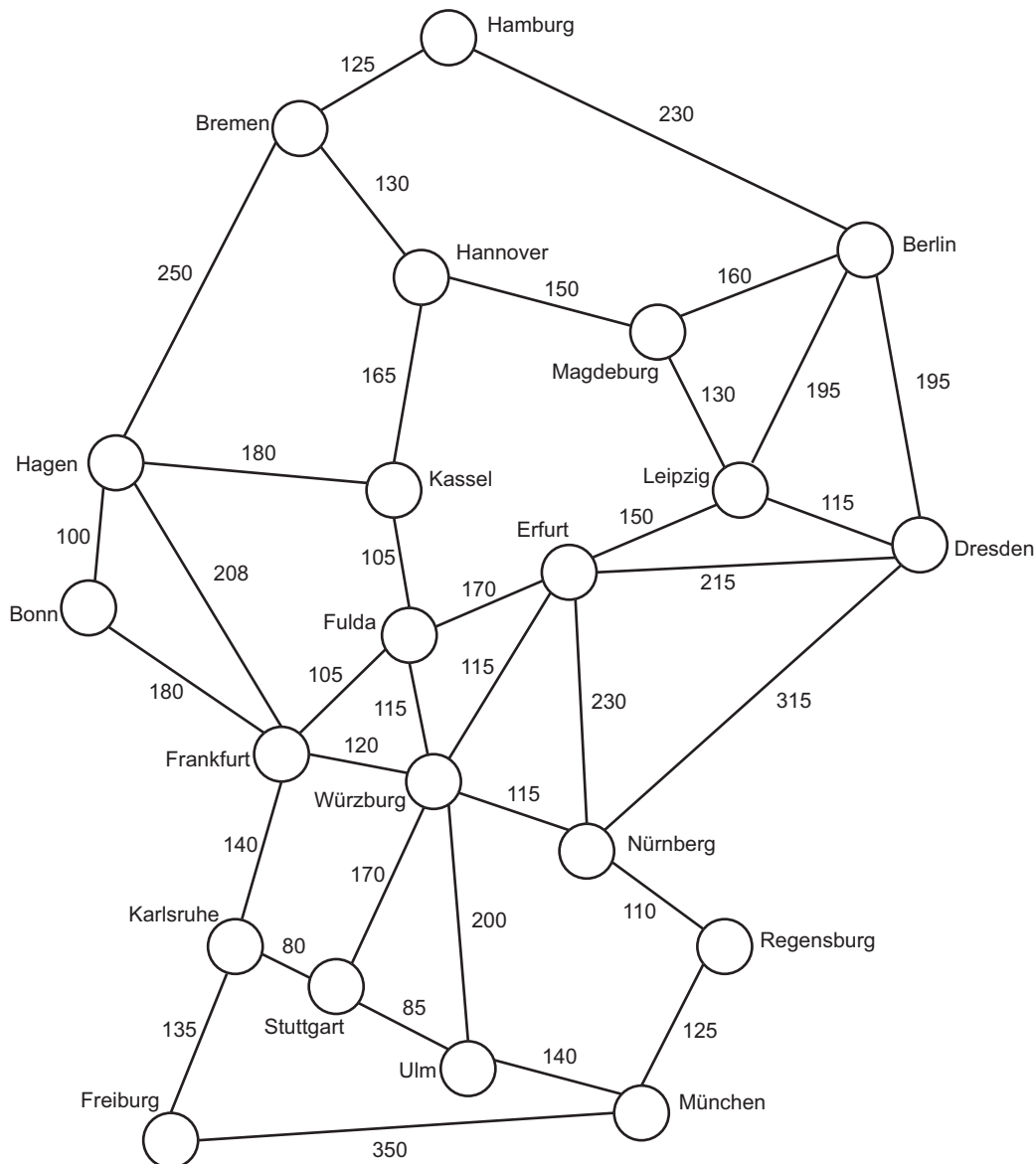


Abb. 36.1-2: Straßenkarte



Ein Pfad, dessen Start- und Zielknoten identisch ist, wird als Zyklus bezeichnet.

Zyklus

Gerichtete Graphen ohne Zyklen werden als gerichtete, azyklische Graphen (engl. *directed acyclic graph*, kurz: DAG) bezeichnet.

gerichteter, azyklischer Graph

Wichtige Algorithmen auf Graphen sind zum Beispiel die Suche von Elementen und Pfaden, insbesondere von kürzesten Wegen.

zusammenhängender
Graph

Ein ungerichteter Graph wird als zusammenhängend bezeichnet, wenn es von jedem Knoten einen Pfad zu jedem anderen Knoten gibt. Ein gerichteter Graph wird als stark zusammenhängend bezeichnet, wenn es zwischen allen Knoten einen gerichteten Pfad gibt. Ein gerichteter Graph wird als schwach zusammenhängend bezeichnet, wenn es zwischen allen Knoten einen ungerichteten Pfad gibt.

stark zusammen-
hängender Graph
schwach zusammen-
hängender Graph

Einer besonderen Gruppe von Graphen, den Bäumen, werden wir uns im nächsten Abschnitt zuwenden.

36.2 Bäume

Bei der Programmierung werden Bäume zum Beispiel als Datenstrukturen zur hierarchischen Speicherung von Daten verwendet.

Im täglichen Leben benutzen wir Baumdarstellungen, um z. B. die Personal- oder Abteilungshierarchie einer Organisation zu beschreiben oder einen Familienstammbaum übersichtlich zu präsentieren. Auch das Dateisystem Ihres Computers ist als Baumstruktur organisiert.

Definition 36.2-1: Baum

Baum

Ein Baum (engl. tree) ist ein gerichteter, schwach zusammenhängender, azyklischer Graph. Jeder Knoten besitzt einen maximalen Eingangsgrad von 1.

┘

Wurzel

Jeder nicht leere Baum hat einen speziellen Knoten, die Wurzel (engl. *root*). Die Wurzel hat als einziger Knoten einen Eingangsgrad von 0.

direkter
Vorgängerknoten
Elternknoten

Der Startknoten der eingehenden Kante wird als direkter Vorgänger- oder Elternknoten bezeichnet. Alle Knoten, außer der Wurzel, haben genau einen direkten Vorgängerknoten.

direkter
Nachfolgerknoten
Kindknoten

Jeder Knoten kann eine endliche Anzahl von direkten Nachfolger- oder Kindknoten besitzen.

Blatt

Knoten, die keine Nachfolger, also einen Ausgangsgrad von 0, besitzen, werden Blätter (engl. *leaf*) genannt.

Höhe eines Baums

Die Höhe eines Baums ist das Maximum der Knoten, die auf dem Weg von einem Blatt zur Wurzel liegen.

Häufig werden bei der Programmierung Baumdatenstrukturen mit bestimmten Eigenschaften benutzt:

Binärer Baum

- Ein Binärer Baum ist ein Baum, bei dem jeder Knoten höchstens zwei Nachfolgerknoten besitzt. Binäre Bäume werden bevorzugt als effiziente Suchstrukturen verwendet.

- Bei einem balancierten Baum (engl. *balanced tree*) B haben alle Unterbäume von B möglichst die gleiche Tiefe, d. h. zwischen der Wurzel und jedem Blatt von B bis auf eine vorgegebene Differenz (meist 1) gleich viele innere Knoten. balancierter Baum
- Ein AVL-Baum ist ein binärer Baum, bei dem sich die Höhen der Unterbäume jedes Knotens höchstens um 1 unterscheiden und jeder Unterbaum ein AVL-Baum ist. AVL-Baum

Abb. 36.2-1 veranschaulicht die Begriffe Wurzel, Knoten und Blatt. In der Informatik wächst ein Baum üblicherweise von oben nach unten, so dass sich seine Wurzel oben und seine Blätter unten befinden. Die Höhe des Baum in Abb. 36.2-1 ist 4.

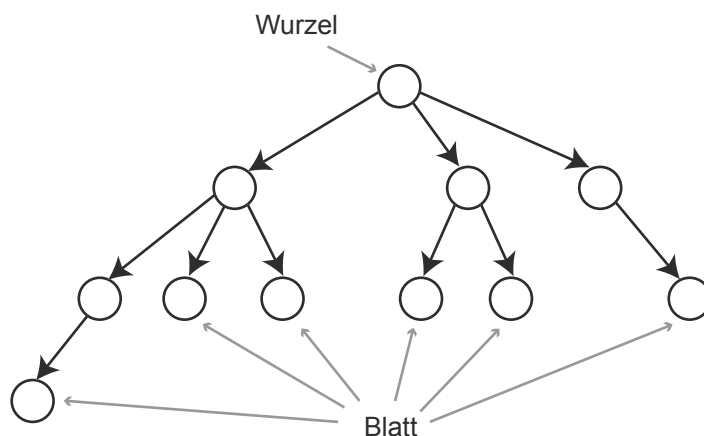


Abb. 36.2-1: Ein Baum

Aus dem bisher Gesagten können wir die folgenden Konzepte ableiten:

Die Vorgänger eines Knotens u sind alle Knoten bis zur Wurzel, die direkt oder indirekt Vorgänger von u sind. Vorgänger

Die Nachfolger von u sind seine direkten Nachfolger, die Nachfolger ihrer Nachfolger usw. bis hin zu den Blättern. Nachfolger

Der durch einen Knoten und alle seine Nachfolgerknoten aufgespannte Baum wird als Unterbaum bezeichnet. Unterbaum

Knoten, die den gleichen direkten Vorgängerknoten haben, werden Geschwisterknoten (engl. *sibling*) genannt. Geschwisterknoten

Selbsttestaufgabe 36.2-1:

Angenommen, wir haben ein Buch mit 4 Kapiteln 1, ..., 4, wobei

- Kapitel 1 aus drei Unterkapiteln 1.1, ..., 1.3 besteht,
- Kapitel 2 zwei Unterkapitel 2.1 und 2.2 aufweist,
- Kapitel 3 vier Unterkapitel besitzt und
- Kapitel 4 drei Unterkapitel hat.

- *Unterkapitel 3.2 bestehe aus zwei Unterkapiteln 3.2.1 und 3.2.2.*

Zeichnen Sie einen Baum, der die Struktur des Buches repräsentiert. Die Knoten des Baumes sollen die Nummern der Kapitel und Unterkapitel tragen, und die Geschwisterknoten sollen in aufsteigender Reihenfolge von links nach rechts angeordnet sein.



36.3 Durchlaufstrategien für Bäume

Wie bei linearen Datenstrukturen müssen auch die Elemente eines Baumes oft durchlaufen werden, zum Beispiel um alle Elemente auszugeben oder ein bestimmtes Element zu finden.

Bei linearen Datenstrukturen, zum Beispiel einer verketteten Liste, beginnen wir mit dem Kopf (bei einer doppelt verketteten Liste möglicherweise auch mit dem Schwanz) und folgen der linearen Struktur der Liste, um Knoten für Knoten den Wert des Eintrags zu verarbeiten. Wenn wir einen Baum durchlaufen wollen, um alle Knoten zu erreichen, gibt es verschiedene Strategien. Jede Strategie beginnt bei der Wurzel und führt für jeden Knoten eine einheitliche Routine aus.

Durchlaufstrategie	Es wird zwischen vier möglichen Strategien, den Baum systematisch zu durchlaufen, unterschieden:
pre-order	<ul style="list-style-type: none"> • pre-order: wir verarbeiten zuerst den Eintrag des aktuellen Knotens, anschließend besuchen wir den linken Nachfolger, danach den rechten Nachfolger;
post-order	<ul style="list-style-type: none"> • post-order: wir besuchen zuerst den linken Nachfolger, danach den rechten Nachfolger, und zuletzt verarbeiten wir den Eintrag des aktuellen Knotens;
in-order	<ul style="list-style-type: none"> • in-order: wir besuchen zuerst den linken Nachfolger, anschließend verarbeiten wir den Eintrag des aktuellen Knotens, danach besuchen wir den rechten Nachfolger;
level-order	<ul style="list-style-type: none"> • level-order: zuerst werden alle Knoten auf einer Ebene untersucht und ihre Einträge verarbeitet, anschließend werden alle Nachfolgerknoten jedes Knotens der betrachteten Ebene von links nach rechts in gleicher Weise untersucht.

Diese Durchlaufstrategien sind in Abb. 36.3-1 veranschaulicht. Die Zahlenangaben verdeutlichen die Reihenfolge, in der die Einträge verarbeitet werden.

Die in-order-Strategie kann lediglich bei binären Bäumen verwendet werden. Alle anderen Strategien können auch bei anderen Bäumen angewendet werden.

Die von uns gezeigten Durchlaufstrategien bevorzugen ein links-nach-rechts-Verfahren. Wir hätten ebenso ein rechts-nach-links-Verfahren vorschlagen können, doch der Durchlauf von links nach rechts ist der Standard.

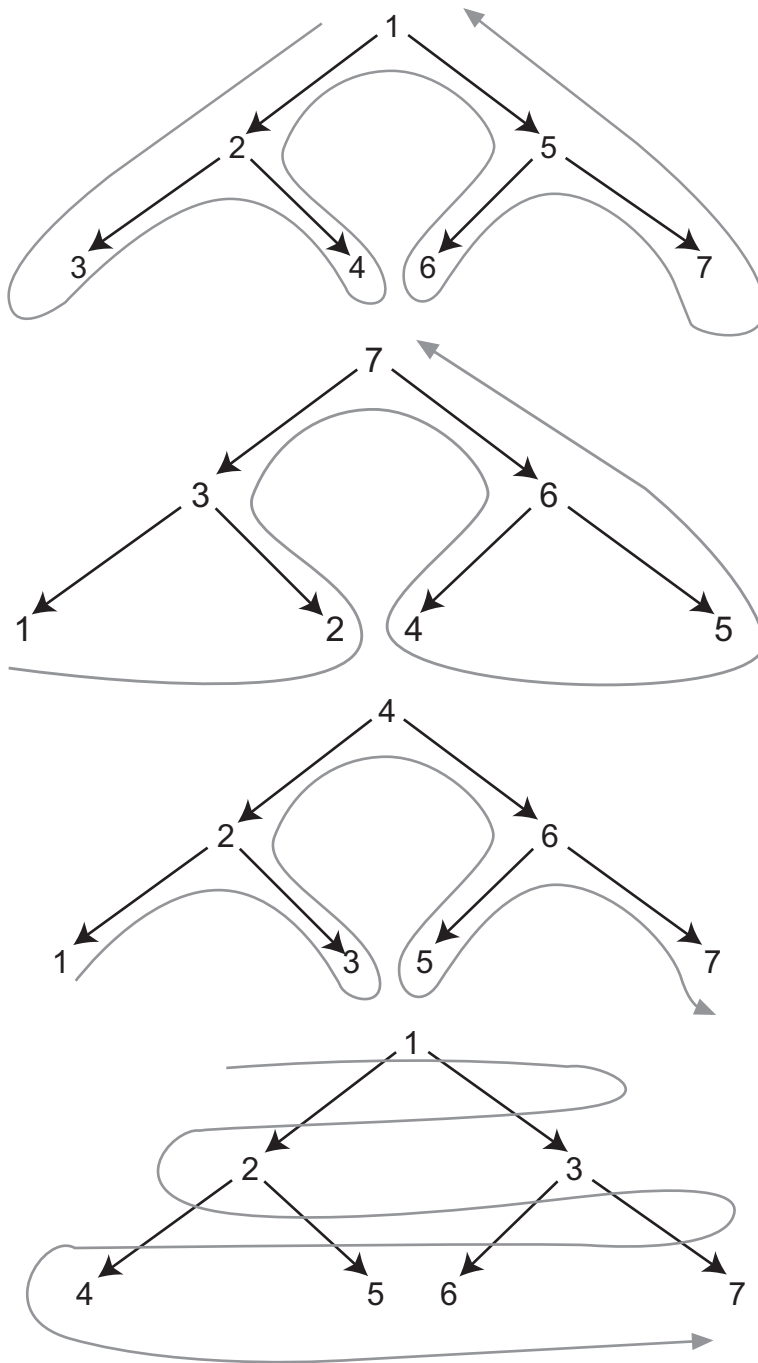


Abb. 36.3-1: Durchlaufstrategien: Pre-order, post-order, in-order und level-order

Selbsttestaufgabe 36.3-1:

Durchlaufen Sie die beiden in Abb. 36.3-2 dargestellten Bäume mit der pre-order-, post-order- und level-order-Strategie und geben Sie jeweils an, in welcher Reihenfolge die Knoten-Einträge verarbeitet werden. Durchlaufen Sie den binären Baum außerdem auch mit Hilfe der in-order-Strategie.



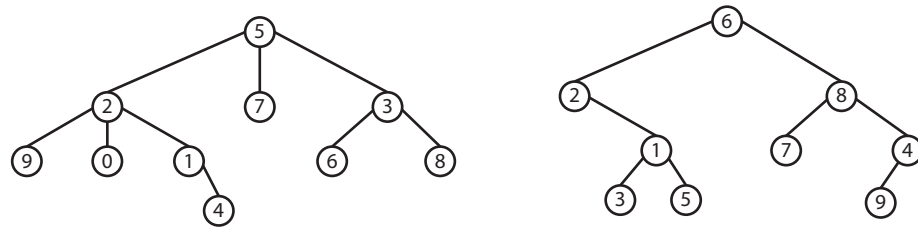


Abb. 36.3-2: Zwei Bäume

36.4 Binäre Bäume

Binäre Bäume sind Datenstrukturen, die in jedem Knoten einen Eintrag speichern und deren Knoten einen maximalen Ausgangsgrad von 2 aufweisen (siehe Abb. 36.4-1).

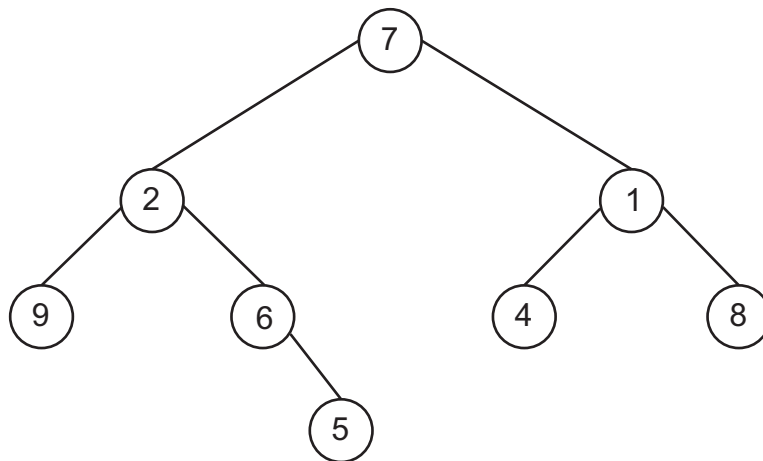


Abb. 36.4-1: Ein binärer Baum

Ein binärer Baum kann in Java ähnlich wie eine Liste implementiert werden. Bei einem binären Baum muss jeder Knoten jedoch nicht nur einen, sondern zwei Verweise auf seine Nachfolger speichern können. Wir definieren die Klasse `BinaryTreeNode` für unsere Baumknoten folgendermaßen:

```
public class BinaryTreeNode {
    private int entry;
    private BinaryTreeNode leftChild;
    private BinaryTreeNode rightChild;

    public BinaryTreeNode(int e) {
        this.entry = e;
    }
}
```

```
public BinaryTreeNode(int e, BinaryTreeNode left,
                      BinaryTreeNode right) {
    this(e);
    this.leftChild = left;
    this.rightChild = right;
}

// Getter und Setter
// ...
}
```

Die Klasse `BinaryTree` ist ähnlich wie eine Liste implementiert. Sie besitzt lediglich einen Verweis auf den Wurzelknoten.

```
public class BinaryTree {
    private BinaryTreeNode root;

    public BinaryTree() {
    }

    public BinaryTree(BinaryTreeNode root) {
        this.root = root;
    }

    // ...
}
```

Mit Hilfe der folgenden Anweisungen erzeugen wir den in Abb. 36.4-2 dargestellten Baum.

```
BinaryTreeNode a = new BinaryTreeNode(4);
BinaryTreeNode b = new BinaryTreeNode(7);
BinaryTreeNode c = new BinaryTreeNode(3, a, b);
BinaryTree t = new BinaryTree(c);
```

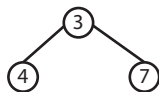


Abb. 36.4-2: Ein kleiner binärer Baum

Selbsttestaufgabe 36.4-1:

Entwickeln Sie Anweisungen, um den Baum aus Abb. 36.4-1 darzustellen. ◇

Rekursive Methoden sind gut geeignet, einen Baum zu durchsuchen und zu verändern, wenn man eine der ersten drei Durchquerungsstrategien pre-, post-, oder in-order benutzt. Diese Verfahren spiegeln die rekursive Struktur eines Baumes, der aus linken und rechten Unterbäumen besteht, wider.

Im Gegensatz dazu ist die level-order-Durchquerung für ein rekursives Verarbeitungsschema weniger gut geeignet, da sie den Verknüpfungen eines Baums nicht folgt, sondern sie quer durchschneidet.

Wir wollen nun eine Methode `printPreorder()` in der Klasse `BinaryTree` implementieren, die die Knoten in der Reihenfolge ausgibt, in der sie mit Hilfe der pre-order-Strategie besucht werden. Auch hier benutzen wir wieder eine private Hilfsmethode.

```
public class BinaryTree {
    private BinaryTreeNode root;

    // ...

    public void printPreorder() {
        // start with root
        printPreorder(root);
    }

    private void printPreorder(BinaryTreeNode tn) {
        // base case: empty subtree
        if (tn == null) {
            return;
        }
        // visit current node
        System.out.print(tn.getEntry() + " ");
        // visit left child
        printPreorder(tn.getLeftChild());
        // visit right child
        printPreorder(tn.getRightChild());
    }
}
```

Selbsttestaufgabe 36.4-2:

Entwickeln Sie analog zur Methode `printPreorder()` die Methoden `printInorder()` und `printPostorder()` für die Klasse `BinaryTree`. ◇

Selbsttestaufgabe 36.4-3:

Entwickeln Sie eine Methode `public boolean contains(int x)`, die überprüft, ob der Baum den Wert x enthält. Geben Sie an, welche Suchstrategie Ihre Methode verwendet.



Bisher haben wir Bäume durch einzelne Anweisungen willkürlich aufgebaut. Zumeist sollen Daten in Bäumen jedoch mit einer gewissen Systematik abgelegt werden. Häufig werden die Elemente nach folgendem Prinzip gespeichert: Der Wert, der im linken Kind gespeichert ist, ist immer kleiner als der Wert des Elternknotens, und der Wert des rechten Kindes ist immer größer als der Wert des Elternknotens. Ein Baum, dessen Elemente diese Eigenschaft aufweisen, wird auch Suchbaum genannt. In einem solchen Suchbaum kann jedes Element maximal einmal vorkommen.

Suchbaum

Das Hinzufügen eines Knoten nach dieser Vorgabe können wir wie folgt implementieren: Wenn ein Baum leer ist, wird das Element als Wurzel gespeichert. Soll einem nichtleeren Baum ein neuer Wert hinzugefügt werden, so wird der einzufügende Wert mit der Wurzel verglichen und dann entschieden, ob im linken oder rechten Teilbaum weiter nach einem freien Platz gesucht werden muss. Ist der entsprechende Teilbaum leer, so wird das Element dort eingefügt. Ansonsten wird mit dem gleichen Verfahren in dem entsprechenden Teilbaum weiter gesucht.

Selbsttestaufgabe 36.4-4:

Zeichnen Sie den Suchbaum, der entsteht, wenn Sie der Reihe nach die Werte 7, 12, 20, 3, 5, 1, 6, 9 einfügen.

**Selbsttestaufgabe 36.4-5:**

Implementieren Sie eine Klasse `BinarySearchTree`, die einen Suchbaum repräsentiert. Ergänzen Sie die Klasse um die Methoden `void add(int x)`, die den neuen Wert an entsprechender Stelle einfügt, `boolean contains(int x)`, die überprüft ob der Wert in dem Baum gespeichert ist und `printInorder()`, die die Elemente in in-order-Reihenfolge ausgibt. Die Methode `contains()` sollte bei ihrer Suche das Wissen über die Struktur eines Suchbaums ausnutzen. Was fällt bei der Ausgabe der Elemente mit Hilfe der Methode `printInorder()` auf?



36.5 Suche in Graphen

Beim Entwurf von Algorithmen ist es ratsam, ein gegebenes Anwendungsproblem auf bekannte Strukturen abzubilden, für die man bereits Lösungsverfahren kennt.

Felder bilden dabei nur eine der vielen möglichen Strukturen. Listen, Warteschlangen, Bäume und Graphen sind weitere mögliche Strukturen. Wegesuchprobleme im Straßenverkehr oder in Kommunikationsnetzen kann man z. B. auf entsprechende Fragestellungen auf Graphen reduzieren. Graphen sind mathematisch und algorithmisch gut untersuchte Strukturen, für die man bereits effektive Lösungsverfahren kennt. Auch viele Suchstrategien arbeiten auf Graphen und können in zwei Suchklassen eingeteilt werden: die Breiten- (engl. *breadth-first*) und die Tiefensuche (engl. *depth-first search*).

Breitensuche

Die beiden Verfahren unterscheiden sich darin, in welcher Reihenfolge sie die Knoten durchsuchen. Beide Verfahren müssen speichern, welche Knoten schon besucht wurden, um nicht im Kreis zu laufen. Die Breitensuche betrachtet zunächst alle direkten Nachbarn des aktuellen Knotens. Wenn das gesuchte Element nicht dabei ist, so werden als nächstes die Nachbarn der Nachbarn betrachtet und so weiter. In welcher Reihenfolge die direkten Nachbarn untersucht werden, spielt keine Rolle. Man tastet sich bei der Suche also in der vollen Breite immer weiter vor. Abb. 36.5-1 veranschaulicht dieses Vorgehen. Die Nummern zeigen an, in welcher Reihenfolge die Knoten besucht werden.

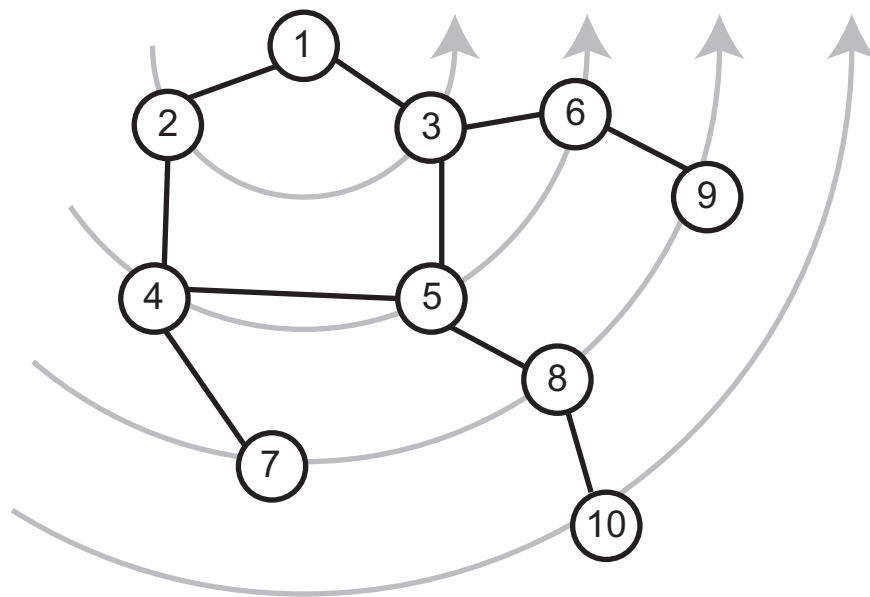


Abb. 36.5-1: Besuchsreihenfolge bei einer Breitensuche

Tiefensuche

Bei der Tiefensuche hingegen wird erst ein Nachbar gewählt und anschließend von diesem Nachbar wieder ein Nachbar ausgewählt. Es wird solange diese Strategie angewendet, bis ein Knoten keine Nachbarn hat oder all seine Nachbarn schon besucht wurden. Ist dies der Fall, geht man einen Schritt zurück und schaut, ob dieser Knoten noch andere unbesuchte Knoten aufweist. Diese Strategie des Zurückgehens wird auch als Backtracking bezeichnet. Bei der Tiefensuche geht man ähnlich wie in einem Labyrinth so lange weiter, bis man in eine Sackgasse gelangt, arbeitet sich dann wieder ein Stück zurück und versucht eine andere Abzweigung. Man sucht al-

Backtracking

so zunächst in der Tiefe des Graphen. Abb. 36.5-2 veranschaulicht dieses Vorgehen. Die Nummern zeigen an, in welcher Reihenfolge die Knoten besucht werden.

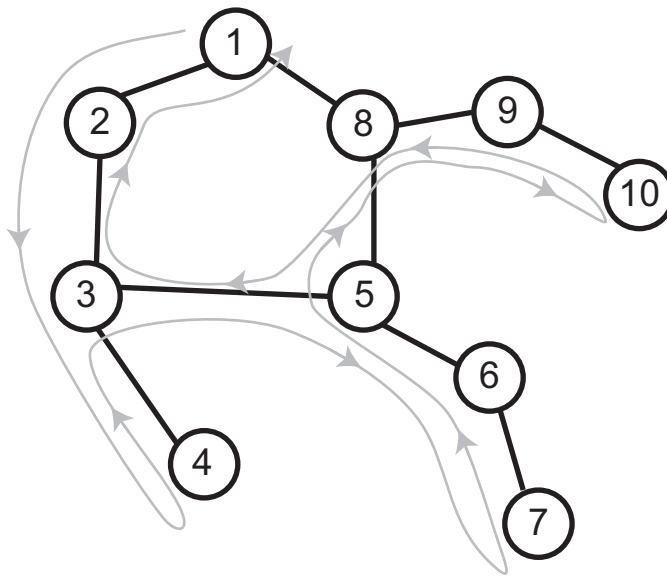


Abb. 36.5-2: Besuchreihenfolge bei einer Tiefensuche

Um eine Breitensuche systematisch durchführen zu können, benötigen wir eine Warteschlange.

1. Füge den Startknoten in die Warteschlange ein.
2. Entnehme solange den ersten, also ältesten Knoten aus der Warteschlange, bis du einen unbesuchten erhältst.
3. Vergleiche den Wert des Knotens mit dem gesuchten Knoten.
4. Sind sie identisch, so ist der gesuchte Knoten gefunden und die Suche kann beendet werden.
5. Sind sie nicht identisch, so wird der Knoten als besucht gekennzeichnet.
6. Füge alle noch nicht besuchten Nachbarn des Knotens in die Warteschlange ein.
7. Ist die Warteschlange leer, so konnte der Knoten nicht gefunden werden und die Suche kann beendet werden.
8. Fahre mit Schritt 2 fort.

Für die Tiefensuche können wir das folgende systematische Verfahren verwenden:

1. Überprüfe ob der aktuelle Knoten der gesuchte Knoten ist.
2. Sind die Knoten identisch, so ist der gesuchte Knoten gefunden und die Suche kann beendet werden.
3. Sind sie nicht identisch, so wird der Knoten als besucht gekennzeichnet.
4. Hat der Knoten keine unbesuchten Nachbarn, so muss die Suche hier erfolgreich abgebrochen werden und zum Vorgänger zurückgekehrt werden.

5. Wähle einen unbesuchten Nachbarn des aktuellen Knotens aus und suche von dort rekursiv mit diesem Verfahren weiter.
6. Wird ohne Erfolg vom besuchten Nachbarn hierher zurückgekehrt, so fahre mit Schritt 4 fort.

Speichert man bei jedem Knoten, von welchem Vorgänger aus er eingefügt wurde, so kann nachträglich auch der Weg vom Start- zum Zielknoten rekonstruiert werden.

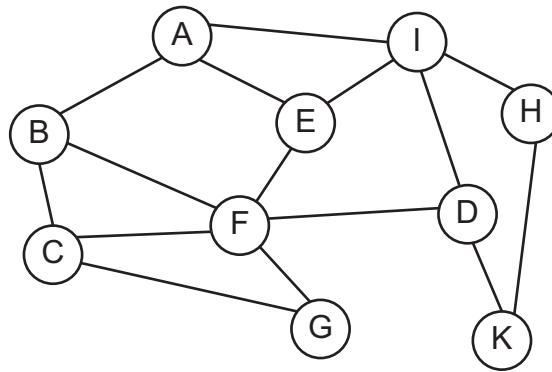


Abb. 36.5-3: Ein Graph

Betrachten wir den in Abb. 36.5-3 gegebenen Graphen. Wir legen fest, dass wir die Nachbarn eines Knoten immer in alphabetischer Reihenfolge besuchen wollen. Wir starten bei Knoten A und wollen einen Weg zu Knoten K finden.

Bei der Breitensuche fügen wir zunächst A in die Warteschlange ein. Wir entnehmen den ersten Knoten, also A, und da es sich dabei nicht um den gesuchten Knoten handelt, markieren wir ihn als besucht. Dann fügen wir die Nachbarn B, E und I ein. Wir entnehmen wieder den ersten Knoten, jetzt also B, und nachdem es nicht der gesuchte Knoten ist, markieren wir ihn und fügen seine Nachbarn C und F ein. A wird nicht mit eingefügt, weil es schon markiert ist. Alle Schritte auf diesem Graphen sind in Abb. 36.5-4 veranschaulicht.

Bei der Tiefensuche stellen wir fest, dass A nicht der gesuchte Knoten ist und markieren ihn als besucht. Als nächstes besuchen wir seinen Nachbarn B. Wir stellen fest, dass es nicht der gesuchte ist, markieren ihn und wählen seine nächsten Nachbarn aus. A kommt nicht in Frage, weil er schon als besucht markiert ist. Also wählen wir C und besuchen diesen. Die weiteren Schritte sind in Abb. 36.5-5 veranschaulicht.

Selbsttestaufgabe 36.5-1:

Welchen Weg von Hamburg nach Dresden (siehe Abb. 36.1-2) finden Sie, wenn Sie die Breitensuche anwenden und dabei die Städte aufsteigend nach ihrer Entfernung sortiert besuchen. Welchen Weg finden Sie, wenn Sie die Tiefensuche mit der gleichen Sortierung benutzen?



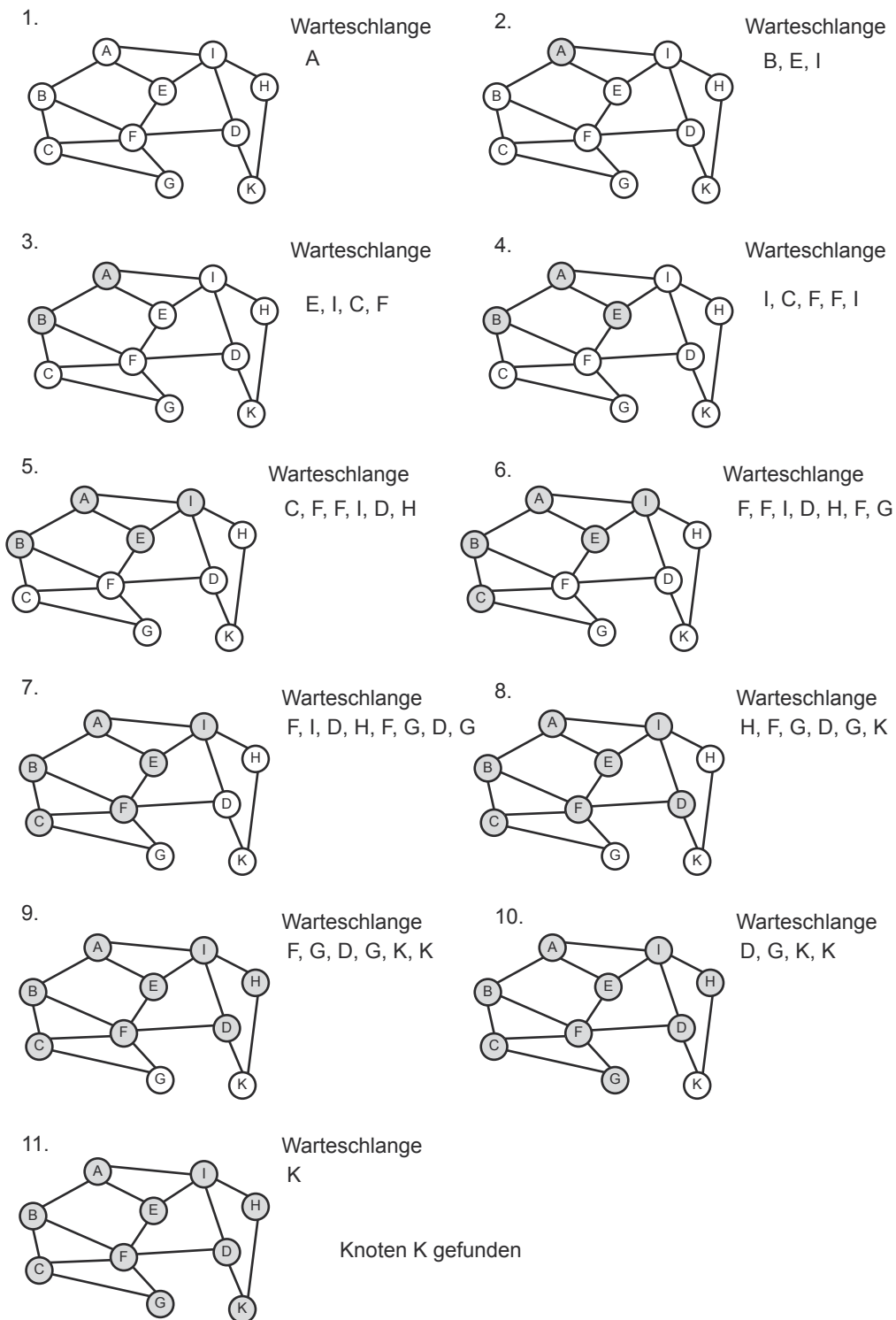


Abb. 36.5-4: Breitensuche auf dem Graph aus Abb. 36.5-3

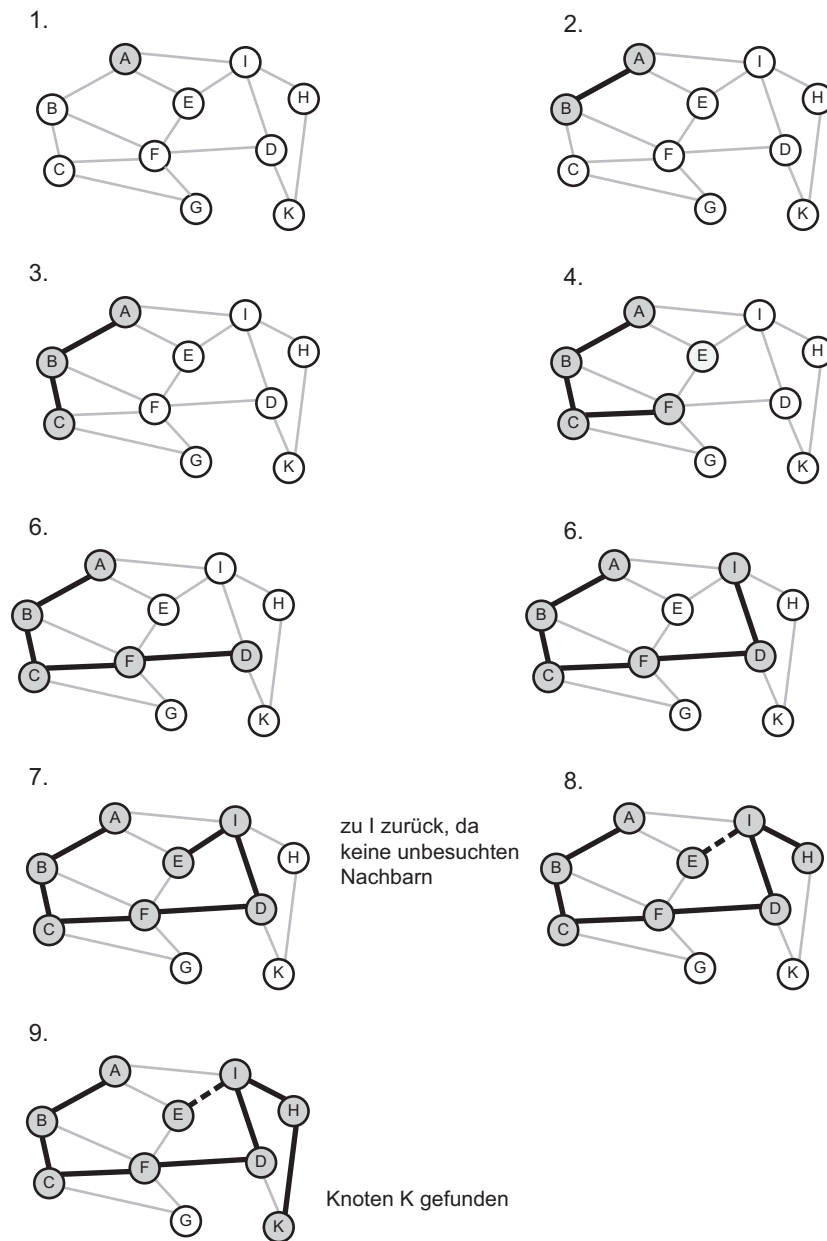


Abb. 36.5-5: Tiefensuche auf dem Graph aus Abb. 36.5-3

37 Zusammenfassung

In dieser Kurseinheit lernten wir sowohl die **lineare** als auch die **binäre Suche** kennen.

Für die **Sortierung** von Datenmengen kennen wir nun sowohl nicht rekursive Algorithmen, wie **Bubblesort**, **Sortieren beim Einfügen** und **Sortieren durch Auswählen**, als auch rekursive Algorithmen, wie **Quicksort** und **Sortieren durch Verschmelzen**.

Rekursive Algorithmen bestehen aus **Basisfällen**, bei denen das Problem trivial lösbar ist, **rekursiven Aufrufen**, die das Problem verkleinern und es mit dem gleichen Verfahren lösen sowie der **Kombination der Teillösungen**. Wichtig dabei ist, dass eine Rekursion **konvergiert**.

Neben Suchen und Sortieren können rekursive Algorithmen auch bei vielen anderen Problemen angewendet werden, insbesondere bei verschiedenen Datenstrukturen.

Für die Verwaltung sich verändernder Datenmengen können **verkettete Listen** verwendet werden, die einige Vor- aber auch Nachteile gegenüber Feldern bieten. Bei beiden handelt es sich jedoch um **lineare Datenstrukturen**. Wir unterscheiden dabei zwischen **wahlfreiem** und **sequenziellem** Zugriff und der Reihenfolge, in der Elemente hinzugefügt und wieder entfernt werden können.

Viele gängige Operationen auf Listen können mit Hilfe rekursiver Ansätze elegant gelöst werden. Für manche Anwendungsfälle werden **doppelt verkettete Listen** benötigt.

Um nicht lineare Strukturen abbilden zu können, verwenden wir **Graphen**. Eine wichtige Unterart der Graphen sind **Bäume**. **Binäre Bäume**, insbesondere **Suchbäume**, eignen sich zur (sortierten) Speicherung von Daten. Bäume können mit Hilfe verschiedener **Strategien** durchlaufen werden.

Graphen können mit Hilfe der **Breiten-** und **Tiefensuche** durchsucht werden.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 33.1-1:

```
int bestimmeAnzahl(int wert, int[] feld) {
    if (feld == null) {
        return 0;
    }
    int anzahl = 0;
    for (int i : feld) {
        // Wert gefunden?
        if (i == wert) {
            anzahl++;
        }
    }
    return anzahl;
}
```

Lösung zu Selbsttestaufgabe 33.1-2:

Da uns hier die lexikalische Gleichheit zweier Zeichenketten interessiert (und nicht die Identität zweier Objekt-Verweise), sollten die Elemente mit Hilfe der Methode `equals()` (und nicht mit `==`) verglichen werden. `equals()` ist in der Klasse `String` so überschrieben, dass genau die Zeichenketten verglichen werden (vgl. Kapitel 26).

```
boolean istEnthalten(String wert, String[] feld) {
    if (feld == null) {
        return false;
    }
    for (String s : feld) {
        // Wert gefunden?
        if (s.equals(wert)) {
            return true;
        }
    }
    return false;
}
```

Lösung zu Selbsttestaufgabe 33.1-3:

```
int bestimmeGesamtbetragAllerRechnungenVon(Kunde k) {
    // wenn kein Kunde oder keine Rechnungen vorhanden
    if (k == null || rechnungen == null) {
        return 0;
    }
    int betrag = 0;
    // alle Rechnungen betrachten
    for (Rechnung r : rechnungen) {
        // wenn gesuchter Kunde
        if (r.liefereRechnungsempfaenger() == k) {
            // Betrag dazu addieren
            betrag += r.bestimmeBetragInCent();
        }
    }
    return betrag;
}
```

Lösung zu Selbsttestaufgabe 33.1-4:

```
Rechnung findeTeuersteRechnung() {
    // wenn keine Rechnungen vorhanden
    if (rechnungen == null) {
        return null;
    }
    Rechnung max = null;
    // alle Rechnungen betrachten
    for (Rechnung r : rechnungen) {
        // wenn noch kein Maximum gefunden
        // oder aktuelle Rechnung teurer
        if (max == null || max.bestimmeBetragInCent()
            < r.bestimmeBetragInCent()) {
            // Maximum anpassen
            max = r;
        }
    }
    return max;
}
```

Lösung zu Selbsttestaufgabe 33.1-5:

```

int bestimmeAnfangdesWorts(char[] feld, String wort) {
    for (int i = 0; i < feld.length - wort.length() + 1; i++) {
        for (int j = 0; j < wort.length(); j++) {
            // Vergleiche entsprechende Buchstaben
            if (feld[i + j] != wort.charAt(j)) {
                // Buchstabe an Position j stimmt nicht überein
                // suche bei i+1 weiter
                break;
            } else if (j == wort.length() - 1) {
                // gesamtes Wort wurde gefunden
                return i;
            }
        }
    }
    // Wort konnte nicht vollständig gefunden werden
    return -1;
}

```

Lösung zu Selbsttestaufgabe 33.2-1:

Es werden 18 Vergleiche und 12 Vertauschungen benötigt.

Lösung zu Selbsttestaufgabe 33.2-2:

```

void sortiereAbsteigend(String[] feld) {
    // beginne beim zweiten Element und betrachte
    // die Liste bis zum Index i - 1 als sortiert
    for (int i = 1; i < feld.length; i++) {
        // gehe solange von i nach links
        // bis das Element an die richtige
        // Stelle gerutscht ist
        for (int j = i; j > 0; j--) {
            if (feld[j - 1].compareTo(feld[j]) < 0) {
                // wenn linkes kleiner,
                // vertausche die Elemente
                String temp = feld[j];
                feld[j] = feld[j - 1];
                feld[j - 1] = temp;
            } else {
                // sonst, ist das Element
                // an der richtigen Stelle
                break;
            }
        }
    }
}

```

Lösung zu Selbsttestaufgabe 33.2-3:

Es werden 25 Vergleiche und 12 Vertauschungen benötigt.

Lösung zu Selbsttestaufgabe 33.2-4:

```
void sortiereAufsteigend(Rechnung[] rechnungen) {
    // es werden maximal length - 1 Durchläufe benötigt
    for (int i = 0; i < rechnungen.length - 1; i++) {
        // solange keine Vertauschungen vorgenommen werden
        // ist das Feld sortiert
        boolean sorted = true;
        // Durchlaufe das Feld, in jedem Durchlauf muss
        // ein Element weniger berücksichtigt werden
        for (int j = 0; j < rechnungen.length - 1 - i; j++) {
            if (rechnungen[j].bestimmeBetragInCent()
                > rechnungen[j + 1].bestimmeBetragInCent()) {
                // wenn linkes größer dann vertausche
                Rechnung temp = rechnungen[j];
                rechnungen[j] = rechnungen[j + 1];
                rechnungen[j + 1] = temp;
                // Feld ist nicht sortiert
                sorted = false;
            }
        }
        if (sorted) {
            // keine Vertauschungen, Feld ist
            // folglich vollständig sortiert
            break;
        }
    }
}
```

Lösung zu Selbsttestaufgabe 33.2-5:

Es werden 28 Vergleiche und 6 Vertauschungen benötigt.

Lösung zu Selbsttestaufgabe 33.2-6:

```

void sortiere(double[] feld) {
    // suche length-1 mal das Maximum
    for (int i = 0; i < feld.length - 1; i++) {
        // Position des Maximums
        int max = 0;
        // Suche das Maximum
        for (int j = 1; j < feld.length - i; j++) {
            // wenn aktueller Wert größer als Maximum
            if (feld[j] > feld[max]) {
                // speichere aktuelle Position
                max = j;
            }
        }
        // setze das Maximum an die letzte der unsortierten
        // Positionen, wenn es dort nicht schon ist
        if (max != feld.length - 1 - i) {
            double temp = feld[max];
            feld[max] = feld[feld.length - 1 - i];
            feld[feld.length - 1 - i] = temp;
        }
    }
}

```

Lösung zu Selbsttestaufgabe 33.2-7:

Sortieren durch Einfügen:		Bubblesort:	
Vergleiche	Vertauschungen	Vergleiche	Vertauschungen
18	12	25	12
7	0	7	0
28	28	28	28

Sortieren durch Auswählen:	
Vergleiche	Vertauschungen
28	6
28	0
28	4

Es fällt auf, dass Sortieren durch Auswählen weniger Vertauschungsoperationen als die anderen Verfahren benötigt, jedoch immer gleich viele Vergleiche. Die anderen beiden Verfahren schneiden dafür bei schon sortierten Feldern besser ab, sind jedoch bei umgekehrt sortierten Feldern deutlich langsamer.

Lösung zu Selbsttestaufgabe 34-1:

Es tritt ein `StackOverflowError` auf. Um dies zu verhindern, kann bei negativen Werten entweder eine `IllegalArgumentException` geworfen oder die Abfrage `n == 0` durch `n <= 0` ersetzt werden.

Lösung zu Selbsttestaufgabe 34-2:

```
int fakultaetIterativ(int n) {
    // ungültige Werte abfangen
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Ergebnis initialisieren
    int erg = 1;
    for (int i = 2; i <= n; i++) {
        erg *= i;
    }
    return erg;
}

int fakultaetRekursiv(int n) {
    // ungültige Werte abfangen
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Basisfall
    if (n == 0) {
        return 1;
    }
    // rekursiver Aufruf
    return n * fakultaetRekursiv(n - 1);
}
```

Lösung zu Selbsttestaufgabe 34-3:

In jedem Schritt wird `n` um eins verringert. Somit nähert sich `n` immer weiter an den Basisfall 0 an.

Lösung zu Selbsttestaufgabe 34-4:

```

public double power(int p, int n) {
    if (n < 0) {
        if (p == 0) {
            System.out.println("Dieses Ergebnis " +
                               "kann nicht berechnet werden!");
        } else {
            return 1.0 / power(p, -n);
        }
    }
    if (n == 0) {
        return 1;
    }
    return p * power(p, n - 1);
}

```

Lösung zu Selbsttestaufgabe 34-5:

0, 1, 1, 2, 3, 5, 8, 13, 21

Lösung zu Selbsttestaufgabe 34-6:

```

long fibIterativ(int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    }
    // Basisfall: n ist 0 oder 1
    if (n == 0 || n == 1) {
        return n;
    }
    // die ersten beiden Zahlen
    int a = 0; // fib(0)
    int b = 1; // fib(1)
    // berechne die nächsten Zahlen
    for (int i = 2; i <= n; i++) {
        // fib(i) = fib(i - 1) + fib(i - 2);
        int temp = a + b;
        // Werte für nächste Berechnung
        // speichern
        a = b;
        b = temp;
    }
    return b;
}

```

Lösung zu Selbsttestaufgabe 34-7:

```
long zufallszahl(int n) {
    // Basisfall n < 3
    if (n < 3) {
        return n + 1;
    }
    // rekursive Aufrufe
    long f1 = zufallszahl(n - 1);
    long f2 = zufallszahl(n - 2);
    long f3 = zufallszahl(n - 3);
    // berechne Ergebnis
    return 1 + (((f1 - f2) * f3) % 100);
}

void gebeZufallszahlenAus() {
    for (int i = 5; i <= 30; i++) {
        System.out.println(zufallszahl(i));
    }
}
```

Lösung zu Selbsttestaufgabe 34-8:

```
boolean istPalindromIterativ(String s) {
    int laenge = s.length();
    // betrachte erste Haelfte des Wortes
    for (int i = 0; i < s.length() / 2; i++) {
        // und prüfe ob korrespondierendes
        // Zeichen identisch ist
        if (s.charAt(i) != s.charAt(laenge - 1 - i)) {
            return false;
        }
    }
    // alle Zeichen passen
    return true;
}
```

Lösung zu Selbsttestaufgabe 34-9:

```
boolean istPalindromRekursiv(String s) {
    // Basisfall
    if (s.length() <= 1) {
        return true;
    }
    // erstes und letztes Zeichen vergleichen
    if (s.charAt(0) != s.charAt(s.length() - 1)) {
        return false;
    }
    return istPalindromRekursiv(s.substring(
        1, s.length() - 1));
}
```

Lösung zu Selbsttestaufgabe 34-10:

Bei der linearen Suche sind genau 11 Vergleiche nötig, bis der Wert gefunden wird.

Bei der binären Suche werden die folgenden Schritte und Vergleiche durchgeführt:

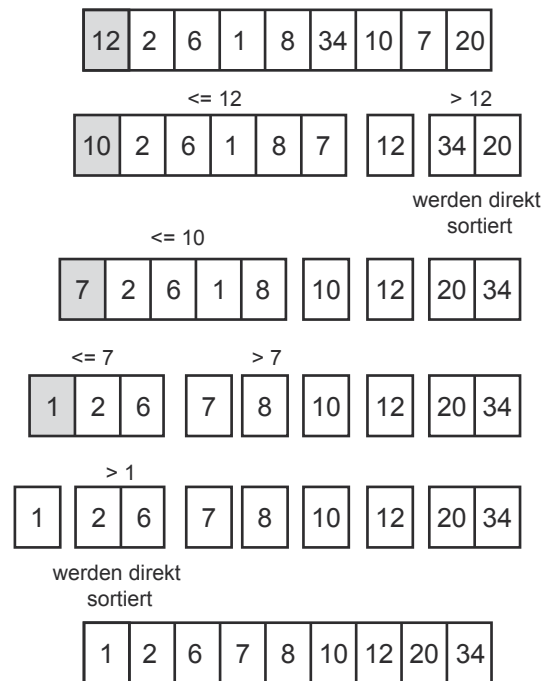
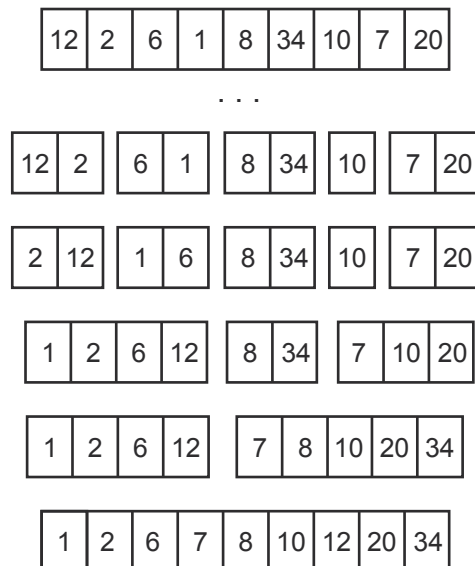
```
start = 0 und ende = 12 => mitte = 6 => Vergleich mit 27
start = 7 und ende = 12 => mitte = 9 => Vergleich mit 34
start = 10 und ende = 12 => mitte = 11 => Vergleich mit 40
start = 10 und ende = 10 => Basisfall => Vergleich mit 38
```

Es finden insgesamt 4 Vergleiche mit den Werten 27, 34, 40 und 38 statt.

Lösung zu Selbsttestaufgabe 34-11:

```
public boolean istEnthalten(String s, String[] feld) {
    return istEnthalten(s, feld, 0, feld.length - 1);
}

// wir gehen von einem sortierten Feld aus
private boolean istEnthalten(String s, String[] feld,
                             int start, int ende) {
    // Basisfall: Bereich enthält maximal 2 Elemente
    if (ende - start <= 1) {
        return feld[start].equals(s) || feld[ende].equals(s);
    }
    // sonst: rekursive Aufteilung
    // Mitte bestimmen
    int mitte = start + (ende - start) / 2;
    System.out.println(mitte + " " + feld[mitte]);
    if (feld[mitte].equals(s)) {
        // wert gefunden
        return true;
    }
    if (feld[mitte].compareTo(s) > 0) {
        // in linker Hälfte suchen
        return istEnthalten(s, feld, start, mitte - 1);
    } else {
        // in rechter Hälfte suchen
        return istEnthalten(s, feld, mitte + 1, ende);
    }
}
```

Lösung zu Selbsttestaufgabe 34-12:**Abb. ML 19:** Sortieren eines Beispielfeldes mit Quicksort**Lösung zu Selbsttestaufgabe 34-13:****Abb. ML 20:** Sortieren eines Beispielfeldes mit Sortieren durch Verschmelzen

Lösung zu Selbsttestaufgabe 35.2-1:

```
public void print() {
    System.out.print(this.entry);
}
```

Lösung zu Selbsttestaufgabe 35.2-2:

```
public int size() {
    int count = 0;
    for (ListNode current = this.head; current != null;
        current = current.getNext()) {
        count++;
    }
    return count;
}
```

Lösung zu Selbsttestaufgabe 35.2-3:

Die binäre Suche setzt zunächst eine Ordnung unter den zu durchsuchenden Elementen voraus. Selbst auf einer sortierten Liste lässt sie sich jedoch nicht effizient implementieren, da Listen auf sequenziellem Zugriff beruhen. Die binäre Suche auf linearen Datenstrukturen erfordert wahlfreien Zugriff.

Lösung zu Selbsttestaufgabe 35.2-4:

```
public int sum() {
    ListNode current = this.head;
    int sum = 0;
    while (current != null) {
        sum += current.getEntry();
        current = current.getNext();
    }
    return sum;
}
```

Lösung zu Selbsttestaufgabe 35.2-5:

```
public int sum() {
    return sum(this.head);
}

private int sum(ListNode node) {
    if (node == null) {
        return 0;
    }
    return node.getEntry() + sum(node.getNext());
}
```

Lösung zu Selbsttestaufgabe 35.2-6:

```
public void printReverseList() {
    printReverseList(this.head);
}

private void printReverseList(ListNode node) {
    if (node == null) {
        return;
    }
    printReverseList(node.getNext());
    node.print();
    System.out.print(" ");
}
```

Lösung zu Selbsttestaufgabe 35.2-7:

Antwort a beantwortet die erste Frage richtig. Da keine neue Liste konstruiert wird, liefert die private, rekursive `remove()`-Methode gemäß Zeile 7 als Ergebnis den Wert des ersten Aufrufs, das `head`-Attribut, zurück.

Die Antwort auf Frage 2 ist die gleiche wie auf Frage 1.

Lösung zu Selbsttestaufgabe 35.2-8:

```
public boolean contains(int value) {
    return contains(this.head, value);
}

private boolean contains(ListNode node, int value) {
    if (node == null) {
        return false;
    }
    if (node.getEntry() == value) {
        return true;
    }
    return contains(node.getNext(), value);
}
```

Lösung zu Selbsttestaufgabe 35.2-9:

Hier gibt es zwei Basisfälle: der erste ist das Ende der Liste bzw. eine leere Liste, der zweite ist der Fall, dass ein passendes Element gefunden wird. Falls der gesuchte Eintrag gefunden wird, endet das Verfahren nach endlich vielen Schritten. Falls der gesuchte Eintrag nicht gefunden wird, erreichen wir nach endlich vielen Schritten über die `next`-Attribute das Ende der Liste, das durch den `null`-Verweis gekennzeichnet ist, und das Verfahren konvergiert durch den ersten Basisfall.

Lösung zu Selbsttestaufgabe 35.2-10:

Iterativ:

```
public int get(int index) {
    if (index < 0 || index >= size()) {
        throw new IndexOutOfBoundsException(
            "ungültiger index");
    }
    ListNode current = this.head;
    for (int i = 0; i < index; i++) {
        current = current.getNext();
    }
    return current.getEntry();
}
```

Rekursiv:

```
public int get(int index) {
    if (index < 0 || index >= size()) {
        throw new IndexOutOfBoundsException(
            "ungültiger index");
    }
    return get(this.head, index).getEntry();
}

private ListNode get(ListNode node, int steps) {
    if (steps == 0) {
        return node;
    }
    return get(node.getNext(), steps - 1);
}
```

Lösung zu Selbsttestaufgabe 35.2-11:

```
public void clear() {
    this.head = null;
}
```

Lösung zu Selbsttestaufgabe 35.2-12:

```
public boolean isEmpty() {
    return (size() == 0);
}
```


Lösung zu Selbsttestaufgabe 35.2-13:

```

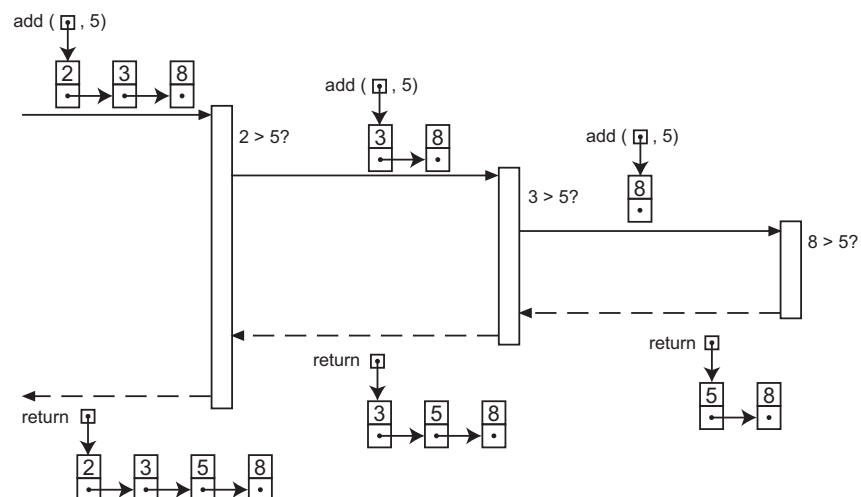
public LinkedList subList(int fromIndex, int toIndex) {
    if (fromIndex < 0 || toIndex > size()
        || fromIndex > toIndex) {
        throw new IndexOutOfBoundsException(
            "ungültiger index");
    }
    LinkedList list = new LinkedList();
    list.head = subList(this.head, fromIndex,
                        toIndex - fromIndex);

    return list;
}

private ListNode subList(ListNode node, int steps, int size) {
    if (size == 0) {
        return null;
    }
    if (steps == 0) {
        ListNode subListNode = new ListNode(node.getEntry());
        subListNode.setNext(subList(node.getNext(), 0,
                                    size - 1));

        return subListNode;
    }
    return subList(node.getNext(), steps - 1, size);
}

```

Lösung zu Selbsttestaufgabe 35.3-1:**Abb. ML 21:** Rekursives Einfügen eines Elements

Lösung zu Selbsttestaufgabe 35.3-2:

`printList()`, `printReverseList()`, `size()`, `sum()` (alle Varianten), `get()`, `clear()`, `subList()`. Methoden, die als Parameter Indexpositionen erwarten (`get()`, `subList()`), werden allerdings für sortierten Listen selten verwendet.

Lösung zu Selbsttestaufgabe 35.3-3:

Wir hatten in der Klasse `LinkedList` im vorherigen Abschnitt die Methode `add()` überladen. Die zweite `add()`-Methode dient dazu, einen Knoten mit einem neuen Eintrag an einer gewählten Stelle `index` in eine Liste einzufügen. Für eine sortierte Liste wäre eine solche Funktionalität fatal.

Es wäre sehr schlechter Programmierstil, die Methode entgegen ihrer Zweckbestimmung mit einer unschädlichen Funktionalität zu überschreiben wie etwa

```
public void add(int index, int value) {
    this.add(value); // Tun Sie so etwas nie!
}
```

Eher wäre zu überlegen, die für beide Klassen nützlichen Methodendefinitionen in eine gemeinsame (ggf. abstrakte) Oberklasse zu verlagern und diese einmal mit `LinkedList` und einmal mit `SortedList` zu spezialisieren.

Lösung zu Selbsttestaufgabe 35.3-4:

Es bestehen keine weiteren Unterschiede.

Lösung zu Selbsttestaufgabe 35.3-5:

```
public boolean contains(int value) {
    return contains(this.head, value);
}

private boolean contains(ListNode node, int value) {
    if (node == null) {
        return false;
    }
    if (node.getEntry() > value) {
        return false;
    }
    if (node.getEntry() == value) {
        return true;
    }
    return contains(node.getNext(), value);
}
```

Lösung zu Selbsttestaufgabe 35.3-6:

```
public class DoublyLinkedListNode {
    private String entry;
    private DoublyLinkedListNode next;
    private DoublyLinkedListNode prev;

    public DoublyLinkedListNode(String value,
                                DoublyLinkedListNode prevNode) {
        this.entry = value;
        this.prev = prevNode;
        this.next = null;
    }

    public DoublyLinkedListNode getNext() {
        return next;
    }

    public void setNext(DoublyLinkedListNode nextNode) {
        this.next = nextNode;
    }

    public String getEntry() {
        return entry;
    }
}

public class DoublyLinkedList {

    private DoublyLinkedListNode head;
    private DoublyLinkedListNode tail;

    public DoublyLinkedList() {
        this.head = null;
        this.tail = null;
    }

    public void enqueue(String value) {
        DoublyLinkedListNode node
            = new DoublyLinkedListNode(value, this.tail);
        if (this.tail != null) {
            this.tail.setNext(node);
        }
        this.tail = node;
        if (this.head == null) {
            this.head = node;
        }
    }
}
```

```

public String dequeue() {
    if (this.head == null) {
        return null;
    }
    String value = this.head.getEntry();
    this.head = this.head.getNext();
    return value;
}
}

```

Lösung zu Selbsttestaufgabe 36.1-1:

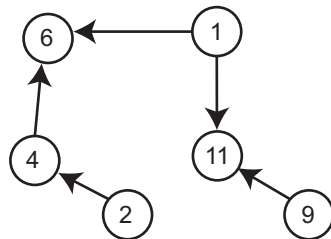


Abb. ML 22: Ein gerichteter Graph

Lösung zu Selbsttestaufgabe 36.1-2:

Der kürzeste Weg von Dresden nach Karlsruhe geht über Erfurt, Würzburg und Stuttgart und hat eine Länge von 580 Kilometern.

Lösung zu Selbsttestaufgabe 36.2-1:

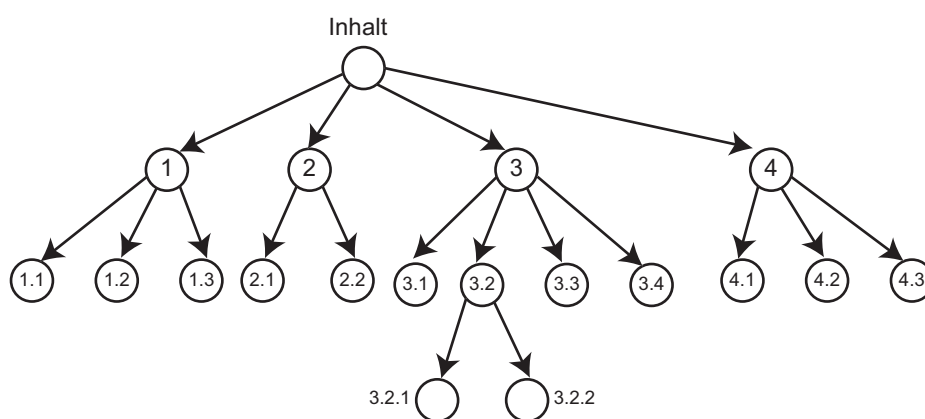


Abb. ML 23: Kapitelstruktur

Lösung zu Selbsttestaufgabe 36.3-1:

linker Baum:

pre-order: 5, 2, 9, 0, 1, 4, 7, 3, 6, 8

post-order: 9, 0, 4, 1, 2, 7, 6, 8, 3, 5

level-order: 5, 2, 7, 3, 9, 0, 1, 6, 8, 4

rechter Baum:

pre-order: 6, 2, 1, 3, 5, 8, 7, 4, 9

post-order: 3, 5, 1, 2, 7, 9, 4, 8, 6

level-order: 6, 2, 8, 1, 7, 4, 3, 5, 9

in-order: 2, 3, 1, 5, 6, 7, 8, 9, 4

Lösung zu Selbsttestaufgabe 36.4-1:

```
BinaryTreeNode a
    = new BinaryTreeNode(6, null, new BinaryTreeNode(5));
BinaryTreeNode b
    = new BinaryTreeNode(2, new BinaryTreeNode(9), a);
BinaryTreeNode c
    = new BinaryTreeNode(1, new BinaryTreeNode(4),
                          new BinaryTreeNode(8));
BinaryTreeNode root = new BinaryTreeNode(7, b, c);
BinaryTree bt = new BinaryTree(root);
```

Lösung zu Selbsttestaufgabe 36.4-2:

```
public class BinaryTree {
    private BinaryTreeNode root;

    // ...

    public void printInorder() {
        // start with root
        printInorder(root);
    }

    private void printInorder(BinaryTreeNode tn) {
        // base case: empty subtree
        if (tn == null) {
            return;
        }
        // visit left child
        printInorder(tn.getLeftChild());
        // visit current node
        System.out.print(tn.getEntry() + " ");
    }
}
```

```

        // visit right child
        printInorder(tn.getRightChild());
    }

    public void printPostorder() {
        // start with root
        printPostorder(root);
    }

    private void printPostorder(BinaryTreeNode tn) {
        // base case: empty subtree
        if (tn == null) {
            return;
        }
        // visit left child
        printPostorder(tn.getLeftChild());
        // visit right child
        printPostorder(tn.getRightChild());
        // visit current node
        System.out.print(tn.getEntry() + " ");
    }
}

```

Lösung zu Selbsttestaufgabe 36.4-3:

```

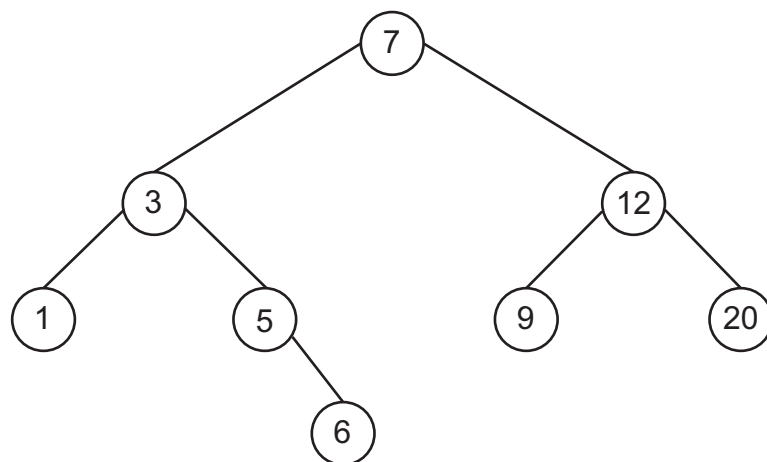
public class BinaryTree {
    private BinaryTreeNode root;

    // ...

    public boolean contains(int x) {
        return contains(root, x);
    }

    private boolean contains(BinaryTreeNode tn, int x) {
        // base case 1: empty subtree
        if (tn == null) {
            return false;
        }
        // search in preorder
        // compare x with current node
        if (tn.getEntry() == x) {
            // base case 2: x found
            return true;
        }
        // search in left and in right subtree
        return contains(tn.getLeftChild(), x)
            || contains(tn.getRightChild(), x);
    }
}

```

Lösung zu Selbsttestaufgabe 36.4-4:**Abb. ML 24:** Suchbaum**Lösung zu Selbsttestaufgabe 36.4-5:**

Durchläuft man einen Suchbaum in in-order-Reihenfolge, so erhält man die Elemente in sortierter Reihenfolge.

```

public class BinarySearchTree {
    private BinaryTreeNode root;

    public void add(int x) {
        // Tree is empty
        if (root == null) {
            root = new BinaryTreeNode(x);
        } else {
            // insert recursive
            add(root, x);
        }
    }

    private void add(BinaryTreeNode tn, int x) {
        // compare current entry
        if (x < tn.getEntry()) {
            // add in left subtree
            if (tn.getLeftChild() == null) {
                // base case 1: insert as left child
                tn.setLeftChild(new BinaryTreeNode(x));
            } else {
                // insert recursive
                add(tn.getLeftChild(), x);
            }
        } else {
            // add in right subtree
            if (tn.getRightChild() == null) {
                // base case 2: insert as right child

```

```

        tn.setRightChild(new BinaryTreeNode(x));
    } else {
        // insert recursive
        add(tn.getRightChild(), x);
    }
}

}

public boolean contains(int x) {
    // search recursive
    return contains(root, x);
}

private boolean contains(BinaryTreeNode tn, int x) {
    // base case 1: no more nodes
    if (tn == null) {
        return false;
    }
    if (tn.getEntry() == x) {
        // base case 2: value found
        return true;
    }
    if (x < tn.getEntry()) {
        // search recursive
        return contains(tn.getLeftChild(), x);
    } else {
        // search recursive
        return contains(tn.getRightChild(), x);
    }
}

public void printInorder() {
    // start with root
    printInorder(root);
}

private void printInorder(BinaryTreeNode tn) {
    // base case: empty subtree
    if (tn == null) {
        return;
    }
    // visit left child
    printInorder(tn.getLeftChild());
    // visit current node
    System.out.print(tn.getEntry() + " ");
    // visit right child
    printInorder(tn.getRightChild());
}
}

```


Lösung zu Selbsttestaufgabe 36.5-1:

Bei der Breitensuche finden Sie den Weg:

Hamburg, Berlin, Dresden

Bei der Tiefensuche finden Sie den Weg:

Bremen, Hannover, Magdeburg, Leipzig, Dresden