

Folien zur Vorlesung *Programmieren 2 (PR2)*

Markus Gumbel

6. Januar 2020

Inhalt

- Exceptions
- Input/Output
- Nebenläufigkeit
- Java Collections-Framework
- Generics
- Datenstrukturen und Abstrakte Datentypen
- Listen, Stacks, Warteschlangen
- Bäume (Suchbäume, Heaps)
- Hashverfahren
- Sortieren
- Graphen
- String-Algorithmen
- Ausblick

Kapitelübersicht

1 Empfohlene Literatur

Empfohlene Literatur (Teil 1)

Zunächst die direkt empfohlenen Bücher:

- Christian Ullenboom. *Java ist auch eine Insel.*
Galileo Press, 2015.
<http://www.tutego.de/javabuch/>.
→ Das deutsche Standardwerk für Java; stets aktuell und als online-Buch verfügbar.
- Michael Kofler. *Java: Der kompakte Grundkurs mit Aufgaben und Lösungen im Taschenbuchformat.*
Rheinwerk Computing, 2 edition, 2017.
→ Ein weiteres Standardwerk für Java.

Empfohlene Literatur (Teil 2)

- Heinz-Peter Gumm and Manfred Sommer. *Einführung in die Informatik*.
Oldenbourg Wissenschaftsverlag.
→ Allgemeines Informatik-Buch. Beispiele in Java.
- Manh Tien Tran and Jörg Hettel. *Nebenläufige Programmierung mit Java: Konzepte und Programmiermodelle für Multicore-Systeme*.
dpunkt. verlag, 2016.
→ Java-Buch speziell für parallele Programmierung. Lohnt sich, wenn das Thema später vertieft werden soll.

Empfohlene Literatur (Teil 3)

- Gunter Saake and Kai-Uwe Sattler. *Algorithmen und Datenstrukturen*. dpunkt.verlag, 5 edition, 2014.
→ Gutes und solides Einsteigerbuch für Algorithmen und Datenstrukturen. Enthält weitestgehend alle Themen, die wir hier behandeln. Beispiele in Java.
- Ralf Hartmut Güting. *Datenstrukturen und Algorithmen*. Leitfäden der Informatik. Springer, 3 edition, 2013.
→ Kompakte aber griffige Darstellung für Algorithmen und Datenstrukturen, sprachneutral. Gut geschrieben.

Empfohlene Literatur (Teil 4)

- Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen*.
Spektrum Akademischer Verlag, 2012.
→ Das deutschsprachige Standardwerk für Algorithmen und Datenstrukturen, sehr umfangreich und theoretisch sehr fundiert. Beispiele in Java.
- Christina Büsing. *Graphen- und Netzwerkoptimierung*.
Spektrum, Akademischer Verlag, Heidelberg, 2010.
→ Ein Buch nur über Graphen. Wir benötigen nur einiges daraus. Als E-Book verfügbar.

Die folgenden Bücher passen ebenfalls sehr gut zur Vorlesung, sind dem Dozenten aber nicht persönlich bekannt.

Empfohlene Literatur (Teil 5)

- Martin Dietzfelbinger, Kurt Mehlhorn, and Peter Sanders.
Algorithmen und Datenstrukturen: Die Grundwerkzeuge.
eXamen.press, 1 edition, 2014.
- Karsten Weicker and Nicole Weicker. *Algorithmen und Datenstrukturen.*
Leitfäden der Informatik. Springer Vieweg.
- Norbert Blum. *Algorithmen und Datenstrukturen.*
Oldenbourg Wissenschaftsverlag, 2 edition, 2012.

Ausgewählte fortgeschrittene Techniken der Programmierung

Kapitelübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Eigene Ausnahmen

Abschnittsübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Eigene Ausnahmen

Motivation: Gängige Fehlersituationen

- Eine Datei soll geladen werden, existiert aber nicht.
- Die Netzwerkverbindung ist nicht verfügbar.
- Der String "a34b" soll in einen Integer umgewandelt werden.
- Eine Methode versucht, einen String auf einen Integer zu casten.
- Das Programm versucht $15 / 0$ zu berechnen.
- Eine Methode greift hinter das letzte Element eines Arrays.
- Es ist kein Speicher mehr auf dem Heap vorhanden.
- Ein Programm gerät in eine unendliche Rekursion und verbraucht den gesamten Stack-Speicher.

Das Problem

Wie transportiert man die Information, dass ein Fehler aufgetreten ist dorthin, wo er korrigiert werden kann?

- Sofortiger Abbruch des Programms?
- Setzen einer globale Fehlervariable?
- Magischer Rückgabewert (-1)?
- Fehler ignorieren und protokollieren?
- Aufruf eines Callbacks zur Fehlerbehandlung?
- Einfach Standardwerte annehmen und weitermachen?

Anforderungen an Fehlerbehandlung

- Fehlerbehandlung und normaler Betrieb sind orthogonal.
- Fehler können nicht ignoriert werden und werden garantiert behandelt.
- Fehlerbehandlung orientiert sich an der Aufrufhierarchie.
- In der Aufrufhierarchie kann an beliebiger Stelle reagiert werden.
- Fehlerbehandlung soll objektorientiert sein.
- Eigene Fehler können definiert und verarbeitet werden.

Abschnittsübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Programmablauf bei Ausnahmen

Behandlung von Ausnahmen

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Eigene Ausnahmen

Beispiel: Ausnahme

```
...
public class ExceptionArrayDemo {
    public static void main(String[] args) {
        String[] namen = {"Franz", "Hans", "Alfons"};

        for (int i = 0; i < 4; i++) {
            System.out.print(namen[i] + ", ");
        }
    }
}
```

Konsole

```
Franz, Hans, Alfons,
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for...
at de.hsmannheim.inf.pr2.exception.ExceptionArrayDemo.main(Excepti...
...
```

Fehlerbehandlung in Java

- Fehler werden durch Ausnahmen (exceptions) signalisiert. Exceptions sind Java-Objekte, die im Fehlerfall erzeugt werden.
- Der try-Block enthält Code, der mit Fehlerbehandlungsroutine(n) versehen werden soll. Tritt ein Fehler auf, wird die Fehlerbehandlungsroutine angesprungen (try-Block wird abrupt beendet). Tritt kein Fehler auf, läuft das Programm nach dem try/catch weiter (try-Block wird normal beendet).
- catch-Blöcke enthalten die Fehlerbehandlungsroutine (exception handler) zum try-Block. Ein Fehler wird als Objekt an *catch* übergeben.

Beispiel: try und catch

...

```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten.
    }
} catch (SQLException ex) {
    // Datenbank Problem behandeln.
    String sqlStatus = ex.getSQLState();
    // ...
    ex.printStackTrace();
}
```

...

Mehrere catch-Blöcke

Merke

- Man kann *beliebig viele* catch-Blöcke angeben.
- Die Laufzeitumgebung wählt den Block aus, bei dem die angegebene Ausnahme am besten zur tatsächlich geworfenen passt.
- Die Auswahl geschieht *anhand der Vererbungshierarchie* der Ausnahmen.
- Die Blöcke müssen in der Reihenfolge *vom Spezifischen zum Allgemeinen* angegeben werden.

Beispiel: Mehrere catch-Blöcke

```
...  
Connection connection = getConnection();  
  
try {  
    ResultSet rs = readData(connection);  
  
    while (rs.next()) {  
        storeToFile(rs.getString(0), rs);  
    }  
} catch (SQLException ex) {  
    // Datenbank-Problem.  
    ex.printStackTrace();  
} catch (FileNotFoundException ex) {  
    // Datei nicht gefunden.  
    System.out.println(ex.getMessage());  
} catch (IOException ex) {  
    // Allgemeiner IO-Fehler.  
    System.out.println(ex.getMessage());  
}
```

Multi-Catch

Merke

- Für einen catch-Block können beliebig viele Ausnahmen (durch | getrennt) angegeben werden.
- Die Ausnahmen dürfen nicht Subklassen voneinander sein.
- Jeder Ausnahme-Typ darf nur einmal genannt werden.

...

```
try {  
    openFile();  
    writeToDatabase();  
} catch (FileNotFoundException | SQLException e) {  
    System.out.println("Ohje, alles kaputt....:" + e);  
}
```

...

So nicht!

```
...
public static void printArray(int[] array) {
    try {
        for (int i = 0; ; i++) {
            System.out.println(array[i]);
        }
    } catch (ArrayIndexOutOfBoundsException ex) {
    }
}
...
```

Abschnittsübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Eigene Ausnahmen

finally-Block

Optional kann ein **finally-Block** angegeben werden.

- Dieser dient für Aufräumarbeiten
 - Schließen von Datenbankverbindungen und Dateien
 - Freigabe von Ressourcen
 - ...
- Wird immer ausgeführt, egal wie try-Block verlassen wurde
 - durch normalen Ablauf
 - durch explizites werfen einer Ausnahme
 - durch fangen und behandeln einer Ausnahme
 - durch eine Ausnahme, die nicht gefangen wird
 - durch ein return-Statement

Beispiel: finally-Block

...

```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten
    }
    return;
} catch (SQLException ex) {
    // Datenbank Problem behandeln.
} finally {
    releaseConnection(connection); // Wird immer ausgeführt!
}
```

...

Ausnahmen im finally-Block

...

```
Connection connection = getConnection();

try {
    ResultSet rs = readData(connection);

    while (rs.next()) {
        // Daten bearbeiten.
    }
    return;
} finally {
    try {
        cleanupConnection(connection);
    } catch (SQLException ex) {
        // Hier kann man nichts mehr machen.
    }
}
```

...

Abruptes Beenden des finally-Blocks

```
...
public class FinallyWrong {

    public static boolean f() {
        try {
            return true;
        } finally {
            return false;
        }
    }

    public static void main(String[] args) {
        System.out.println(f());
    }
}
...
```

Konsole

false

Abschnittsübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Eigene Ausnahmen

Merke: Kategorien von Ausnahmen

- **Errors** - Schwerwiegende Fehler, von denen man sich im Allgemeinen nicht erholen kann
 - Speicher geht aus.
 - VM hat andere Probleme.
- **(Checked) Exceptions** - Fehler, die behandelt werden sollten
 - Datei nicht gefunden.
 - Falsche IP-Adresse angegeben.
 - Falsches SQL-Statement.
- **Runtime Exceptions** - Programmierfehler, die man im Allgemeinen nicht behandelt
 - Zugriff auf null.
 - Division durch 0.

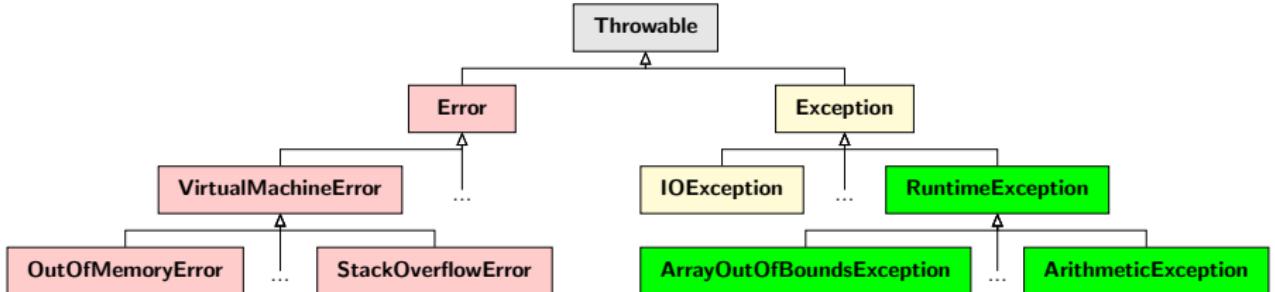


Abbildung 1: Klassenhierarchie der Java-Exceptions. Errors sind hier rot, Checked Exceptions gelb und Runtime Exceptions grün dargestellt.

Throwable

```
+Throwable()  
+Throwable(message: String)  
+Throwable(message: String, cause: Throwable)  
+Throwable(cause: Throwable)  
+getMessage(): String  
+getLocalizedMessage(): String  
+getCause(): Throwable  
+printStackTrace()  
+printStackTrace(s: PrintStream)  
+printStackTrace(s: PrintWriter)  
+fillInStackTrace(): Throwable  
+getStackTrace(): StackTraceElement[]
```

Abbildung 2: Die Klasse Throwable.

Die Klasse Exception

Die Klasse Exception fügt keine neuen Methoden hinzu, sondern enthält nur Konstruktoren, da Konstruktoren in Java nicht vererbt werden. Beispiele für häufige Ausnahmen sind:

- ArithmeticException
- NullPointerException
- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- SecurityException
- IllegalStateException
- ...

Welche Bedeutung diese Exceptions haben kann der Leser in der entsprechenden JavaDoc selbst nachschlagen.

Was gibt dieses Programm aus? Stimmen Sie in Moodle ab.

```
...
public static void main(String[] args) {
    int a = 1, b = 30;
    try {
        System.out.println("Hallo Welt!");
        a--;
        int c = b / a;
        System.out.println("Das Wetter ist schön!");
        try {
            String[] namen = {"Romeo", "Julia"};
            for (int i = 0; i < 3; i++) {
                System.out.println(namen[i] + " ist verliebt.");
            }
        } catch (Exception e) {
            System.out.println("Was war denn hier los?");
        } finally {
            System.out.println("Auf Wiedersehen.");
        }
    } catch (ArithmaticException ex) {
        System.out.println("Huch, wie konnte das passieren?");
    } catch (Exception ex) {
        System.out.println("Hey, was soll das?");
    }
}
...
}
```

Abschnittsübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Call stack (Stacktrace)

„handle or declare“-Regel

Eigene Ausnahmen

Merke: Call-Stack-Mechanismus

- Ausnahmen werden entlang der **Aufrufhierarchie (call stack)** propagiert.
- Wenn eine Ausnahme nicht von einem try-catch-Block behandelt wird, wird sie an den Aufrufer weitergegeben.
- Gelangt die Ausnahme bis zur `main()`-Methode und wird dort nicht behandelt, wird das Programm abgebrochen.
- Hierdurch kann die Ausnahme bis zu demjenigen wandern, der sie beheben kann.

Beispiel: Call-Stack-Mechanismus

```
...
public class StackTraceDemo {
    public static void main(String[] args) throws Exception {
        FileInputStream fis =
            new FileInputStream(new File("/tmp"));
    }
}
```

...

Konsole

```
java.io.FileNotFoundException: \tmp (Das System kann die angegebene...
at java.base/java.io.FileInputStream.open0(Native Method)
at java.base/java.io.FileInputStream.open(FileInputStream.java:213)
at java.base/java.io.FileInputStream.<init>(FileInputStream.java:155)
at de.hsmannheim.inf.pr2.exception.StackTraceDemo.main(StackTraceD...
...
...
```

Beispiel: Call-Stack-Mechanismus (declare)

Programmierer unterstellt nicht vorhandene Datei. Fehler tritt später auf.

```
...
public static void main(String[] args) throws IOException {
    mergeData("/tmp/input.dat", "/tmp/output.dat");
}

public static void mergeData(String inFile, String outFile)
    throws IOException {
    writeData(inFile, outFile);
}

public static void writeData(String inFile, String outFile)
    throws IOException {
    aggregate(5, 5, inFile);
}

public static int aggregate(int a, int b, String inFile)
    throws IOException {
    FileInputStream f = new FileInputStream(inFile);
    return 1;
}
...
```

Beispiel: Call-Stack-Mechanismus (declare)

Benutzer macht falsche Eingabe. Fehler tritt später auf.

```
...
public static void main(String[] args) throws IOException {
    // Der übergebene Dateiname könnte falsch sein.
    mergeData(args[0], "/tmp/output.dat");
}

public static void mergeData(String inFile, String outFile)
    throws IOException {
    writeData(inFile, outFile);
}

public static void writeData(String inFile, String outFile)
    throws IOException {
    aggregate(5, 5, inFile);
}

public static int aggregate(int a, int b, String inFile)
    throws IOException {
    FileInputStream f = new FileInputStream(inFile);
    return 1;
}
...

```

Beispiel: Call-Stack-Mechanismus (handle)

```
...
public static void main(String[] args) throws IOException {
    mergeData("/tmp/output.dat");
}

public static void mergeData(String outFile)
    throws IOException {
    writeData(outFile);
}

public static void writeData(String outFile) throws IOException {
    aggregate(5, 5);
}

public static int aggregate(int a, int b) throws IOException {
    // Eine FileNotFoundException sollte hier behandelt werden:
    FileInputStream f = new FileInputStream("/tmp/input.dat");
    return 1;
}
...
```

„handle or declare“-Regel

Für Ausnahmen gilt die „**handle or declare**“-Regel:

- Die Ausnahme wird durch einen try/catch behandelt *oder*
- Die Ausnahme muss bei der Methode mit dem throws Schlüsselwort angegeben werden.

Ausnahme bei Runtime Exceptions und Errors

Runtime Exceptions und Errors werden nicht deklariert, denn

- Programmierfehler sollen nicht behandelt werden.
- Errors können (eigentlich) nicht behandelt werden.

```
public void storeToFile(String fileName, ResultSet rs)
    throws FileNotFoundException, IOException {
    ...
}
```

Beispiel: Handle or Declare

```
public void userInterface() {  
  
    String dateiName = askUser();  
  
    try {  
        dateiAnlegen(dateiName);  
    }  
    catch (FileNotFoundException ex) {  
        // Benutzer erneut nach Dateinamen Fragen.  
    }  
    catch (IOException ex) {  
        // Benutzer auf Problem hinweisen.  
    }  
}
```

Ausnahmen und Überschreiben

Merke

Überschriebene Methoden können

- Deklarierte Ausnahmen unverändert lassen *oder*
- Deklarierte Ausnahmen weglassen *oder*
- Deklarierte Ausnahmen durch ihre Subklassen ersetzen *oder*
- Ausnahmen hinzufügen, die Subklassen der deklarierten sind.

Sie können auf keinen Fall neue Checked Exceptions hinzufügen. Der Grund ist Polymorphie.

Abschnittsübersicht

② Fehlerbehandlung mit Exceptions

Motivation

Ausnahmen mit try-/catch-Anweisung

Aufräumarbeiten mit finally

Arten von Ausnahmen

Exceptions werden behandelt oder weitergegeben

Eigene Ausnahmen

Ausnahmen selbst schreiben

- Man legt eine Subklasse von Exception an.
- Man gibt ihr einen Namen, der auf "Exception" endet.
- Man fügt Daten hinzu, die für die Ausnahme wichtig sind.
- Man bietet passende Konstruktoren zu den Daten an, mindestens aber zwei Konstruktoren:
 - MyException()
 - MyException(String message)

Ausnahmen selbst werfen

- Nur Subklassen von Throwable können geworfen werden.
- Im Allgemeinen wird man nur Runtime Exceptions und (Checked) Exceptions werfen.
- Errors sind der VM vorbehalten.
- Ausnahmen werden mit dem Schlüsselwort throw geworfen.
- Ausnahmen sollten frisch mit new erzeugt werden.
- Syntax: `throw new WhateverException(...)`

Beispiel: Exceptions werfen

...

```
try {  
    System.out.println("A");  
    if (true) { // Sonst "unreachable code".  
        throw new IOException("Festplatte abgebrannt.");  
    }  
    System.out.println("B");  
} catch (IOException e) {  
    System.out.println("Fehler: " + e.getMessage());  
}
```

...

Konsole

A

Fehler: Festplatte abgebrannt.

Beispiel: Exceptions werfen (2)

```
...
public static void throwExample() {
    try {
        System.out.println("A");
        thrower();
        System.out.println("B");
    } catch (IOException e) {
        System.out.println("Fehler: " + e.getMessage());
    }
}

private static void thrower() throws IOException {
    throw new IOException("Festplatte explodiert");
}
...
```

Konsole

```
A
Fehler: Festplatte explodiert
```

Eigene Exception für einen Server-Status

```
...
public class ServerException extends Exception {

    private int port;
    private String host;

    ...
    public ServerException(String message, String host, int port) {
        super(message);
        this.host = host;
        this.port = port;
    }

    public int getPort() {
        return port;
    }

    public String getHost() {
        return host;
    }
}
...
```

Eigene Server-Exception fangen

```
...
public class Client {

    public void doIt() {

        try {
            Server.connect("server1", 22);
        } catch (ServerException e1) {
            System.err.println("Keine Verbindung zum server "
                + e1.getHost() + " auf Port " + e1.getPort());

            try {
                Server.connect("server2", 22);
            } catch (ServerException e2) {
                System.err.println("Ersatzserver geht auch nicht: "
                    + e2.getHost() + " auf Port " + e2.getPort());
            }
        }
    }
}
...
```

Eigene Server-Exception werfen

```
...
public class Server {

    public static void connect(String host, int port)
        throws ServerException {
        int result = connectInternal(host, port);

        if (result == -1) {
            throw new ServerException("Cannot connect to " +
                host + ":" + port, host, port);
        }
        // ...
    }

    private static native int connectInternal(String host,
                                              int port);
}
...
```

Kapitelübersicht

3 Grundlagen von Input-/Output-Operationen

Einführung

Streams zum Verarbeiten binärer Daten

Reader/Writer zum Verarbeiten von Text

Konsolen-IO

Abschnittsübersicht

3 Grundlagen von Input-/Output-Operationen

Einführung

Streams zum Verarbeiten binärer Daten

Reader/Writer zum Verarbeiten von Text

Konsolen-IO

Motivation

In klassischen Programmiersprachen ist **Eingabe/Ausgabe** (**input/output**, kurz I/O oder IO) Teil der Sprache. In modernen Programmiersprachen wird I/O in Bibliotheken ausgelagert, z. B.:

- Sockets
- Pipes
- Dateien
- HTTP
- ...

Häufig sind die Schnittstellen für die verschiedenen Datenquellen sehr unterschiedlich.

Eingabe/Ausgabe in Java

Java verwendet ein einheitliches Konzept für (nahezu) alle I/O-Operationen: **Streams** (Ströme). Sie liefern bzw. konsumieren die Daten in sequentieller Form. Man unterscheidet:

- Input Streams – Das Programm liest Bytes von einer Quelle.
- Output Streams – Das Programm schreibt Bytes in ein Ziel.
- Reader – Wie InputStream nur für Zeichen statt Bytes.
- Writer – Wie OutputStream nur für Zeichen statt Bytes.

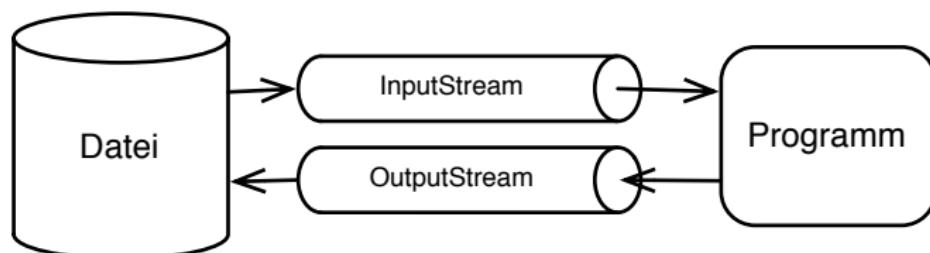


Abbildung 3: Streams

Abschnittsübersicht

③ Grundlagen von Input-/Output-Operationen

Einführung

Streams zum Verarbeiten binärer Daten

Input-Streams

Output-Streams

Filter-Streams

DataStreams

Reader/Writer zum Verarbeiten von Text

Konsolen-IO

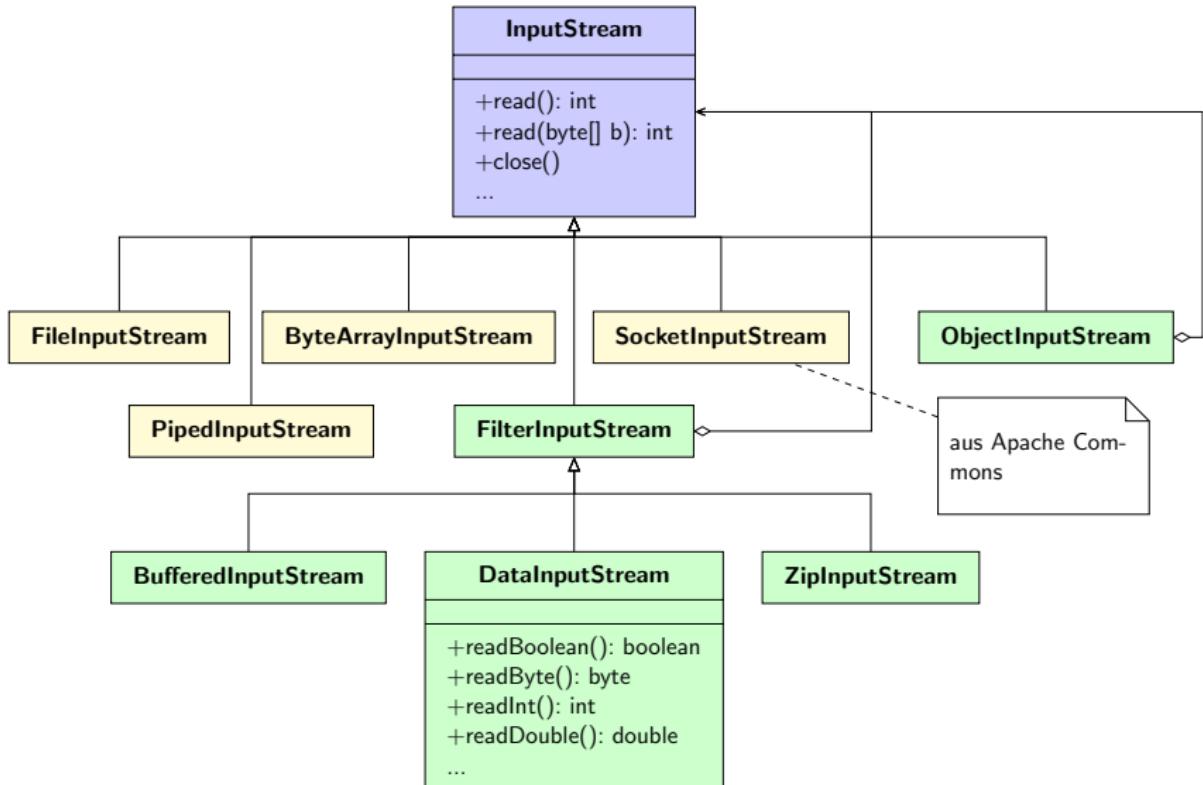


Abbildung 4: Klassenhierarchie von Input-Streams.

Klassenhierarchie: Input Streams

- `FileInputStream` – Dateien
- `ZipInputStream` – Zip-Dateien
- `SocketInputStream` – Netzwerk-Sockets
- `ObjectInputStream` – Serialisierung von Objekten
- `PipedInputStream` – Verbindung von Threads
- `ByteArrayInputStream` – Lesen aus einem Byte-Array.
- `FilterInputStream` – Basisklasse für Filter
- `BufferedInputStream` – Pufferung
- `DataInputStream` – Lesen von Javas primitiven Datentypen.

Die Methoden (Teil 1)

- `abstract int read()` – liest ein einzelnes Byte vom Stream.
- `int read(byte[] b)` – liest vom Stream in das Byte-Array.
- `int read(byte[] b, int off, int len)` – liest maximal `len` Bytes in das Byte-Array ab Position `off` (im Array).
- `long skip(long n)` – überspringt `n` Bytes.
- `int available()` – Anzahl der Bytes, die voraussichtlich gelesen werden können, ohne dass die Leseoperation blockiert.
- `void mark(int readlimit)` – markiert die aktuelle Position im Stream und vergisst sie erst nach `readlimit` Bytes, die daraufhin gelesen wurden.
- `void reset()` – springt zu der mit `mark()` markierten Stelle im Stream zurück.
- `boolean markSupported()` – zeigt an, ob der Stream überhaupt das Setzen einer Marke mit `mark()` unterstützt.
- `void close()` – schließt den Stream.

Bytes von einem Stream lesen

...

```
InputStream fis = new FileInputStream("files/bindata.dat");
int daten;

while ((daten = fis.read()) > -1) {
    byte b = (byte) daten;
    // Jetzt kann man etwas sinnvolles mit den Bytes machen,
    // die aus der Datei gelesen wurden.
}
fis.close();
...
```

In diesem Beispiel wird die `-1` richtig behandelt. Erst nach dem Vergleich erfolgt der cast.

Bytes von einem Stream lesen

Achtung!

```
...
FileInputStream fis = new
FileInputStream("files/bindata.dat");
byte b;

while ((b = (byte) fis.read()) > -1) {
    // Jetzt kann man etwas sinnvolles mit den Bytes machen,
    // die aus der Datei gelesen wurden.
}
fis.close();
...
```

In diesem Beispiel wird die `-1` falsch behandelt. Es wird erst gecastet und dann verglichen. Der Lesevorgang wird potentiell zu früh beendet, weil `-1` ein möglicher Byte-Wert ist und so der Lesevorgang abbricht, wenn dieser Wert in der Quelle auftritt.

Blöcke von einem Stream lesen

```
...
InputStream fis = new FileInputStream("files/bindata.dat");
byte[] daten = new byte[1024];
int bytesRead;

while ((bytesRead = fis.read(daten)) > -1) {
    // Jetzt kann man etwas sinnvolles mit den Bytes machen,
    // die aus der Datei gelesen wurden.
}
fis.close();
...
```

Hier signalisiert der Rückgabewert der Methode, wie viele Bytes gelesen wurden. -1 zeigt das Ende des Streams an. 0 ist ein korrekter Rückgabewert, da es beim Lesen vom Netzwerk vorkommen kann, dass noch keine neuen Daten eingetroffen sind, die Verbindung aber noch steht. Hier muss man einfach erneut read aufrufen.

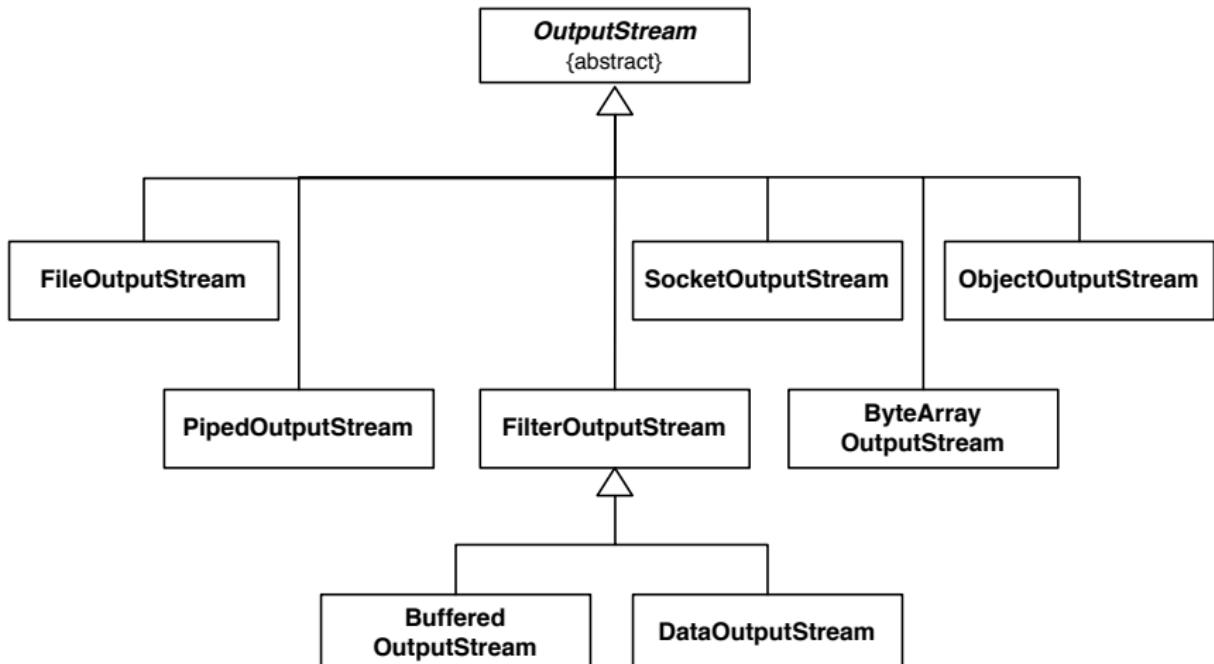


Abbildung 5: Klassenhierarchie Output-Streams.

Klassenhierarchie: Output Streams

- `FileOutputStream` – Dateien
- `SocketOutputStream` – Netzwerk-Sockets
- `ObjectOutputStream` – Serialisierung von Objekten
- `PipedOutputStream` – Verbindung von Threads
- `FilterOutputStream` – Basisklasse für Filter
- `ByteArrayOutputStream` – Schreiben in ein Byte-Array
- `BufferedOutputStream` – Pufferung
- `DataOutputStream` – Schreiben von Daten

Die Methoden von OutputStream

- `abstract void write(int b)` – schreibt ein Byte.
- `void write(byte[] b)` – schreibt das gesamte Byte-Array.
- `void write(byte[] b, int off, int len)` – schreibt len Bytes aus dem Byte-Array beginnend bei off.
- `void flush()` – leert interne Puffer und bringt die Daten auf die Platte.
- `void close()` – schließt den Stream (impliziert ein `flush()`).

Bytes auf einen Stream schreiben

```
...  
OutputStream fos = new  
FileOutputStream("/tmp/cafe.dat");  
  
fos.write(0xca);  
fos.write(0xfe);  
fos.write(0xba);  
fos.write(0xbe);  
  
fos.close();  
...
```

Blöcke auf einen Stream schreiben

```
...  
OutputStream fos = new FileOutputStream("/tmp/cafe.dat");  
byte[] daten = {(byte) 0xca, (byte) 0xfe,  
                (byte) 0xba, (byte) 0xbe};  
  
fos.write(daten);  
fos.write(daten, 0, 2);  
fos.write(daten, 0, 2);  
fos.write(daten, 2, 2);  
fos.write(daten, 2, 2);  
  
fos.close();  
...
```

myfile (hex.)

CA FE BA BE CA FE CA FE BA BE BA BE

FileInputStream

FileInputStream erbt alle Methoden von InputStream und fügt nur zwei Konstruktoren hinzu, um eine Datei zum Öffnen auswählen zu können. Man kann die Datei entweder über den Dateinamen (als String) oder ein spezielles File-Objekt öffnen (das noch später erläutert wird).

- `public FileInputStream(String name) throws FileNotFoundException`
- `public FileInputStream(File file) throws FileNotFoundException`

FileOutputStream

FileOutputStream verhält sich analog zum FileInputStream, nur dass hier über einen weiteren Konstruktorparameter append angegeben werden kann, ob die Datei überschrieben (false) werden soll oder die geschriebenen Daten hinten an die Datei angehängt werden sollen (true).

- `public FileOutputStream(String name) throws FileNotFoundException`
- `public FileOutputStream(String name, boolean append) throws FileNotFoundException`
- `public FileOutputStream(File file) throws FileNotFoundException`
- `public FileOutputStream(File file, boolean append) throws FileNotFoundException`

Beispiel: Anhängen an File-Stream

```
...  
byte[] daten = {(byte) 0xca, (byte) 0xfe,  
                (byte) 0xBA, (byte) 0xbe};  
  
OutputStream fos = new FileOutputStream("/tmp/cafe.dat");  
fos.write(daten);  
fos.close();  
  
OutputStream fos2 =  
    new FileOutputStream("/tmp/cafe.dat", true);  
fos2.write(daten);  
fos2.close();  
...
```

myfile (hex.)

CA FE BA BE CA FE BA BE

Beispiel: Einfaches Kopierprogramm

```
...  
String quelle = "files/ipsumfile.txt";      // Quell und  
String ziel = "files/ipsumfile-copy.txt"; // Zielfile.  
  
// Streams für Quell und Zielfile:  
InputStream in = new FileInputStream(quelle);  
OutputStream out = new FileOutputStream(ziel);  
  
byte[] buffer = new byte[1024]; // Puffer für Dateiinhalt.  
int gelesen;                // Anzahl gelesener Bytes.  
  
// Daten aus Quell- in Zielfile kopieren:  
while ((gelesen = in.read(buffer)) > -1) {  
    out.write(buffer, 0, gelesen);  
}  
  
in.close(); // Streams schließen.  
out.close();  
...
```

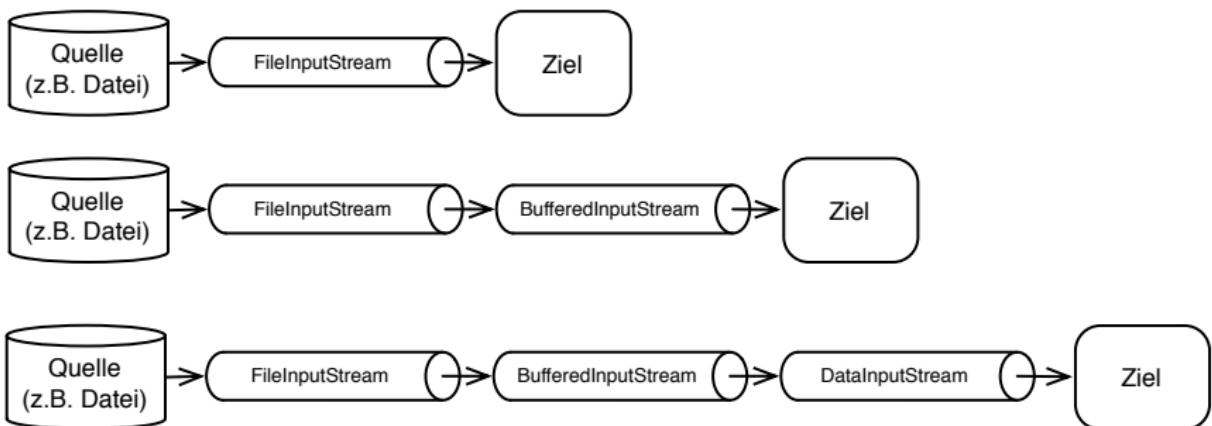


Abbildung 6: Filter-Streams und Decorator-Pattern. Zusätzlich zu den Streams, die direkt mit einer Quelle oder Senke (Datei, Socket etc.) verbunden sind, gibt es die Filter-Streams. Filter-Streams können hinter einen anderen Stream geschaltet werden (Decorator Pattern) und können die Daten entsprechend verändern (daher Filter-Streams).

Beispiel: Buffering

```
...
String quelle = "files/ipsumfile.txt";      // Quell und
String ziel = "files/ipsumfile-copy.txt"; // Zielfile.

InputStream in = new BufferedInputStream(
    new FileInputStream(quelle));

OutputStream out = new BufferedOutputStream(
    new FileOutputStream(ziel));

byte[] buffer = new byte[1024];
int gelesen;

while ((gelesen = in.read(buffer)) > -1) {
    out.write(buffer, 0, gelesen);
}

in.close();
out.close();
```

DataOutputStream / DataInputStream

- **DataOutputStream** schreibt primitive Datentypen und String in einen anderen Stream und erlaubt so den plattformunabhängigen Datenaustausch.
- **DataInputStream** liest die von DataOutputStream geschriebenen Daten wieder ein.

Methoden von DataOutputStream (Teil 1)

- void writeBoolean(boolean v) throws IOException
- void writeByte(int v) throws IOException
- void writeShort(int v) throws IOException
- void writeChar(int v) throws IOException
- void writeInt(int v) throws IOException
- void writeLong(long v) throws IOException
- void writeFloat(float v) throws IOException
- void writeDouble(double v) throws IOException
- void writeBytes(String s) throws IOException
- void writeChars(String s) throws IOException
- void writeUTF(String s) throws IOException

Beispiel: DataOutputStream

...

```
    DataOutputStream out = new DataOutputStream(
        new BufferedOutputStream(
            new FileOutputStream("/tmp/daten.dat")));
```

```
    out.writeUTF("** Datendatei **"); // Unicode Transform.
```

Format

```
    out.writeUTF("Datum");
    out.writeLong(new Date().getTime());
    out.writeUTF("PI");
    out.writeDouble(Math.PI);
```

```
    out.close();
    ...
```

Methoden von DataInputStream (Teil 1)

- boolean readBoolean() throws IOException
- byte readByte() throws IOException
- int readUnsignedByte() throws IOException
- short readShort() throws IOException
- int readUnsignedShort() throws IOException
- char readChar() throws IOException
- int readInt() throws IOException
- long readLong() throws IOException
- float readFloat() throws IOException
- double readDouble() throws IOException
- String readUTF() throws IOException

Beispiel: DataInputStream

...

```
DataInputStream dis = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("/tmp/daten.dat")));  
  
String header = dis.readUTF();  
String datumTag = dis.readUTF();  
Date datum = new Date(dis.readLong());  
String PITag = dis.readUTF();  
double pi = dis.readDouble();  
  
dis.close();  
  
System.out.println(header);  
System.out.println(datumTag + " " + datum);  
System.out.println(PITag + " " + pi);  
...
```

Konsole

```
** Datendatei **  
Datum Mon Jul 24 15:52:59 CEST 2017  
PI 3.141592653589793
```

Abschnittsübersicht

③ Grundlagen von Input-/Output-Operationen

Einführung

Streams zum Verarbeiten binärer Daten

Reader/Writer zum Verarbeiten von Text

Reader

Writer

Wrapper für Streams

Konsolen-IO

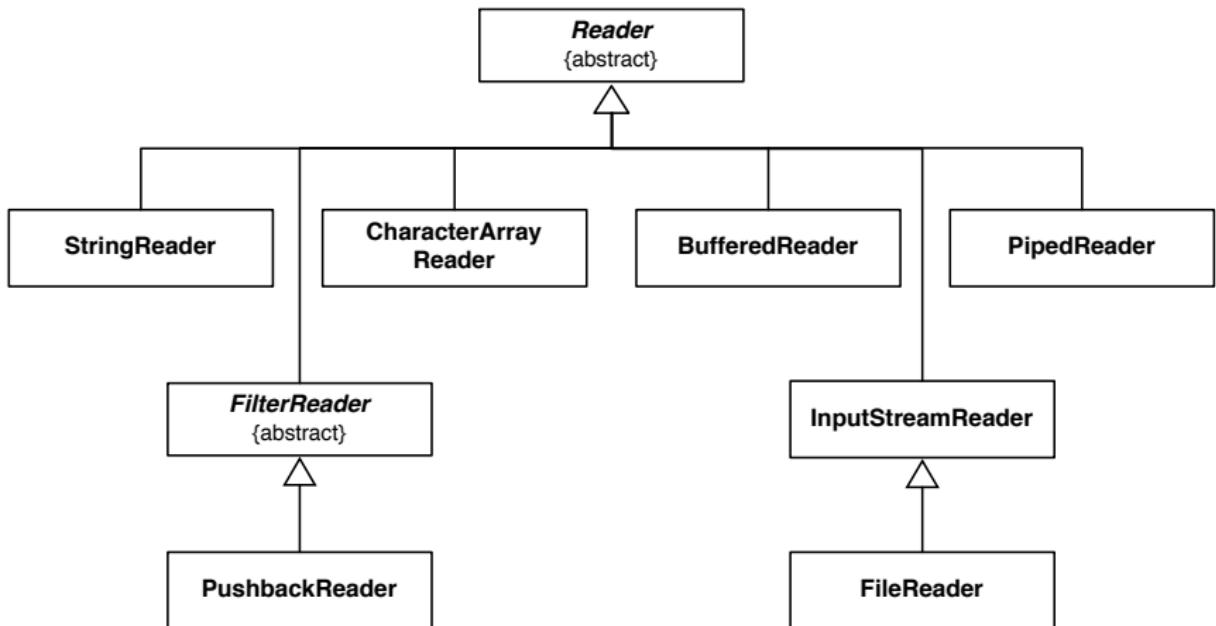


Abbildung 7: Klassenhierarchie Reader.

Klassenhierarchie: Reader

- `StringReader` – aus Strings lesen.
- `CharacterArrayReader` – aus char-Arrays lesen.
- `BufferedReader` – gepuffertes Lesen.
- `PipedReader` – Verbindung von Threads.
- `InputStreamReader` – Verknüpfung von Stream und Reader.
- `FileReader` – aus Dateien lesen.
- `FilterReader` – Filter.
- `PushbackReader` – lesen und zurückstellen von Zeichen.

Methoden von Reader (Teil 1)

- `int read()` – liest ein Zeichen.
- `int read(char[] cbuf)` – liest Zeichen in das char-Array.
- `int read(char[] cbuf, int off, int len)` – liest maximal `len` Zeichen in das char-Array `cbuf` beginnend bei Position `off`.
- `long skip(long n)` – überspringt `n` Zeichen.
- `boolean ready()` – gibt an, ob der nächste `read()` auf dem Reader blockiert oder nicht.
- `void mark(int readlimit)` – markiert die aktuelle Position im Reader und vergisst sie erst nach `readlimit` Bytes, die daraufhin gelesen wurden.
- `void reset()` – springt zu der mit `mark()` markierten Stelle im Reader zurück.

Methoden von Reader (Teil 2)

- boolean markSupported() – zeigt an, ob der Reader überhaupt das setzen einer Marke mit mark() unterstützt.
- void close() – schließt den Reader.

Zeichen von einem Reader lesen

```
...
Reader fr = new FileReader("files/ipsumfile.txt");
int daten;

while ((daten = fr.read()) > -1) {
    char c = (char) daten;
    // Jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden.
}
fr.close();
...
```

So nicht!

```
...
Reader fr = new FileReader("files/ipsumfile.txt");
char b;

while ((b = (char) fr.read()) > -1) {
    // Jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden.
}
fr.close();
...
```

Dieser Code funktioniert nicht. Die Methode `read()` signalisiert das Ende der Daten durch einen Rückgabewert von `-1` als `int`. Wenn man das Ergebnis aber vorher auf `char` castet, wird das Lesen niemals beendet, da `char` ein vorzeichenloser Datentyp ist und daher das Ergebnis eines Casts auf `char` immer ≥ 0 ist.

Blöcke von einem Reader lesen

```
...
Reader fr = new FileReader("files/ipsumfile.txt");
char[] daten = new char[1024];
int charactersRead;

while ((charactersRead = fr.read(daten)) > -1) {
    // Jetzt kann man etwas sinnvolles mit den Zeichen machen,
    // die aus der Datei gelesen wurden.
}
fr.close();
...
```

Beispiel: StringReader

```
...
String text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\n" +
    "Nulla laoreet, sem vel mollis imperdiet, sapien mauris\n" +
    "sollicitudin arcu, sed viverra nulla dui at est.\n" +
    "Nunc est erat, semper id sollicitudin ut, pretium ac eros.\n";
Reader sr = new StringReader(text);
int gelesen;
while ((gelesen = sr.read()) > -1) {
    System.out.print((char) gelesen);
}
sr.close();
...
...
```

Methoden des BufferedReader

- `BufferedReader(Reader in)`
- `BufferedReader(Reader in, int bufferSize)`
- `String readLine()`

Zu beachten ist, dass der BufferedReader die sehr nützliche `readLine()`-Methode einführt, mit der man Textdateien zeilenweise lesen kann. Das Ende der Datei wird bei dieser Methode durch den Rückgabewert `null` signalisiert.

Beispiel: BufferedReader

zum Nummerieren von Zeilen

```
...
String dateiname = "files/ipsumfile.txt";
BufferedReader reader = new BufferedReader(
    new FileReader(dateiname));
String zeile;
int nummer = 1;
while ((zeile = reader.readLine()) != null) {
    System.out.printf("%03d %s%n", nummer, zeile);
    nummer++;
}
reader.close();
...
```

Konsole

```
001 Lorem ipsum dolor sit amet, consectetur adipiscing elit.
002 Nulla laoreet, sem vel mollis imperdiet, sapien mauris
003 sollicitudin arcu, sed viverra nulla dui at est.
004 Nunc est erat, semper id sollicitudin ut, pretium ac eros.
```

Beispiel: Eigener FilterReader

```
...
public classUpperCaseReader extends FilterReader {

    protectedUpperCaseReader(Reader in) {
        super(in);
    }

    @Override
    public int read(char[] cbuf, int off, int len)
        throws IOException {
        int result = super.read(cbuf, off, len);

        for (int i = off; i < off + result; i++) {
            cbuf[i] = Character.toUpperCase(cbuf[i]);
        }
        return result;
    }
}
...
```

Beispiel: Eigener FilterReader

```
...
    String dateiname = "files/ipsumfile.txt";
    BufferedReader reader = new BufferedReader(
        new UpperCaseReader(
            new FileReader(dateiname)));
    String zeile;

    while ((zeile = reader.readLine()) != null) {
        System.out.println(zeile);
    }
    reader.close();
...

```

Konsole

```
LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT.
NULLA LAOREET, SEM VEL MOLLIS IMPERDIET, SAPIEN MAURIS
SOLLICITUDIN ARCU, SED VIVERRA NULLA DUI AT EST.
NUNC EST ERAT, SEMPER ID SOLLICITUDIN UT, PRETIUM AC EROS.
```

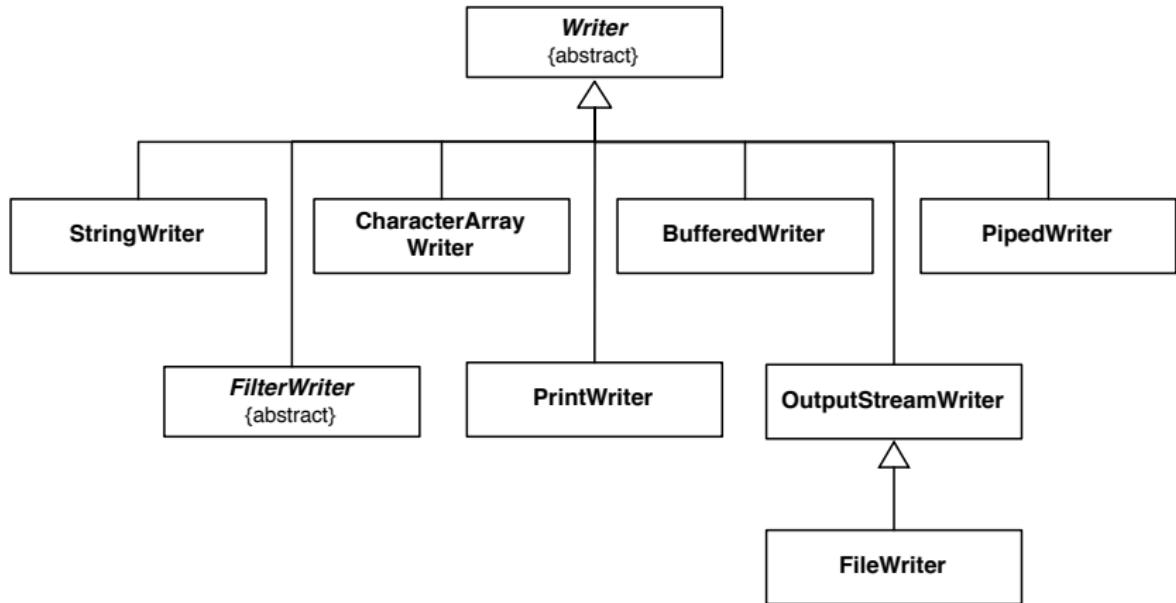


Abbildung 8: Klassenhierarchie: Writer

Klassenhierarchie: Writer

- `StringWriter` – In Strings schreiben.
- `CharacterArrayWriter` – In char-Arrays schreiben.
- `BufferedWriter` – Gepuffertes Schreiben.
- `PipedWriter` – Verbindung von Threads.
- `OutputStreamWriter` – Verbindung von Stream und Writer.
- `FileWriter` – In Datei schreiben.
- `PrintWriter` – Formatierte Ausgabe von Datentypen.
- `FilterWriter` – Filter.

Die Methoden von Writer

- `void write(int c)` – schreibt ein Zeichen.
- `void write(char[] cbuf)` – schreibt alle Zeichen aus dem char-Array.
- `void write(char[] cbuf, int off, int len)` – schreibt len Zeichen aus dem char-Array cbuf beginnend bei off.
- `void write(String str)` – schreibt den String str.
- `void write(String str, int off, int len)` – schreibt len Zeichen aus dem String str, beginnend bei off.
- `Writer append(char c)` – schreibt ein Zeichen.
- `void flush()` – leert interne Puffer und bringt die Daten auf die Platte.
- `void close()` – schließt den Writer (impliziert einen `flush()`).

Zeichen in einen Writer schreiben

```
...  
Writer fw = new FileWriter("files/chartext.txt");  
  
fw.write('T');  
fw.write('e');  
fw.write('x');  
fw.write('t');  
fw.write('\n'); // Newline  
  
fw.close();  
...
```

chartext.txt

Text

Strings in einen Writer schreiben

```
...  
Writer fw = new FileWriter("files/stringtext.txt");  
String daten = "Dies ist ein Text"; // 17 Zeichen  
  
fw.write(daten); // Gesamter String.  
fw.write(daten, 0, 5); // 5 Zeichen ab Pos. 1 ("Dies ").  
fw.write(daten, 0, 5); // nochmal  
fw.write(daten, 12, 5); // 4 Zeichen ab Pos. 13 (" Text").  
fw.write(daten, 12, 5); // nochmal  
  
fw.write("\n"); // Newline  
  
fw.close();  
...
```

stringtext.txt

Dies ist ein Text Dies Text Text

InputStreamReader etc.

Manche APIs liefern nur Streams – manchmal möchte man die Daten aber mit einem Reader/Writer verarbeiten:

- Sockets
- Servlet API

InputStreamReader und OutputStreamWriter dienen dazu, Streams und Reader/Writer miteinander zu verbinden. Verwendung als Filter bzw. Decorator.

Beispiel: Einfacher HTTP-Client

```
...
    Socket socket = new Socket("www.hs-mannheim.de", 80);
    OutputStream os = socket.getOutputStream();
    InputStream is = socket.getInputStream();

    BufferedReader reader = new BufferedReader(new InputStreamReader(is));
    Writer writer = new OutputStreamWriter(os);

    writer.write("GET / HTTP/1.0\r\n\r\n");
    writer.flush();

    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }

    reader.close(); // Alle offenen Ressourcen schließen.
    writer.close();
    socket.close();
...

```

Ausgabe: Einfacher HTTP-Client

Die Antwort des Servers sagt, dass die Anfrage umgeleitet wird.

Konsole

```
HTTP/1.1 301 Moved Permanently
Date: Mon, 07 Aug 2017 13:36:05 GMT
Server: Apache
Location: http://www./
Cache-Control: max-age=86400
Expires: Tue, 08 Aug 2017 13:36:05 GMT
Content-Length: 220
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www./">here</a>.</p>
</body></html>
```

Die Klasse PrintWriter

Die Klasse PrintWriter hat `print(...)`- und `println(...)`-Methoden für alle primitiven Datentypen, `char[]`, `String` und `Object`. Die `print`-Methoden werfen keine `IOExceptions`, Fehler müssen mit `checkError` abgeholt werden. Sie bietet mit `printf(...)` eine mächtige Methode zur formatierten Ausgabe von Daten.

Abschnittsübersicht

3 Grundlagen von Input-/Output-Operationen

Einführung

Streams zum Verarbeiten binärer Daten

Reader/Writer zum Verarbeiten von Text

Konsolen-IO

Konsole IO

Ein Sonderfall von Streams ist das Lesen und Schreiben von Standard-In und Standard-Out. Normalerweise sind diese beiden Streams mit der **Konsole** (Bildschirm/Tastatur) verbunden, durch Ein- und Ausgabeumleitung können die Daten aber mit anderen Quellen und Senken verbunden sein.

- `System.out` - PrintStream, der mit der Konsole verbunden ist (`stdout`).
- `System.err` - PrintStream, der mit der Konsole verbunden ist (`stderr`).
- `System.in` - InputStream, der mit der Konsole verbunden ist. Man kann keine einzelne Zeichen mit `System.in.read()` lesen.

Beispiel: Echo-Befehl

```
...
public class Echo {
    public static void main(String[] args)
        throws IOException {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));

        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    }
}
```

Konsole

```
> cat files/ipsum.txt | java -cp . de.hsmannheim.inf.pr2.io.Echo
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nulla laoreet, sem vel mollis imperdiet, sapien mauris
sollicitudin arcu, sed viverra nulla dui at est.
```

Beispiel: Umleitung von System.out

```
...
PrintStream ps = new PrintStream("files/umleitung.txt");
System.setOut(ps);

System.out.println("Hallo, das kommt ja gar nicht raus...");

ps.close();
...
```

umleitung.txt

Hallo, das kommt ja gar nicht raus...

Kapitelübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

synchronized-Blöcke

Blockieren und Aufwecken von Threads

Abschnittsübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

synchronized-Blöcke

Blockieren und Aufwecken von Threads

Nebenläufigkeit

Beispiele sind:

Webserver Ein Webserver sollte in der Lage sein, mehrere Benutzeranfragen (*requests*) parallel zu verarbeiten.

Nachladen Eine App soll eine Animation anzeigen und gleichzeitig Daten aus dem Netzwerk nachladen.

Rechtschreibprüfung Eine Rechtschreibprüfung soll neben dem normalen Editieren des Dokuments ablaufen.

Parallele Algorithmen Beispielweise lassen sich einige mathematische Algorithmen einfach parallelisieren.

Microprocessor Transistor Counts 1971-2011 & Moore's Law

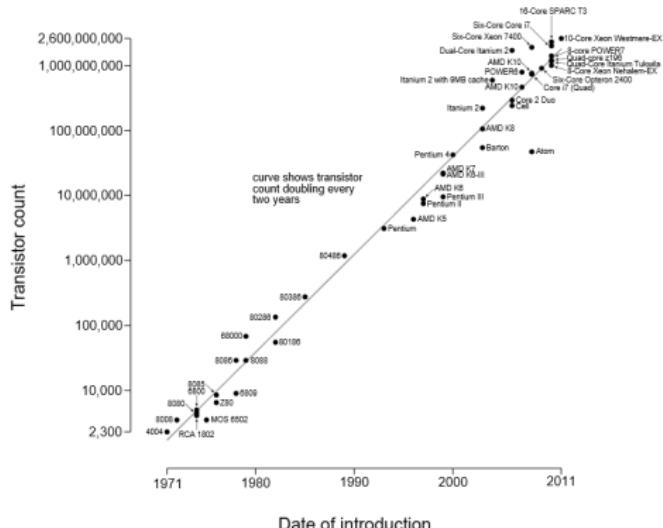


Abbildung 9: Moores Gesetz (Moore's law) [?] lautet: The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [two years]... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.

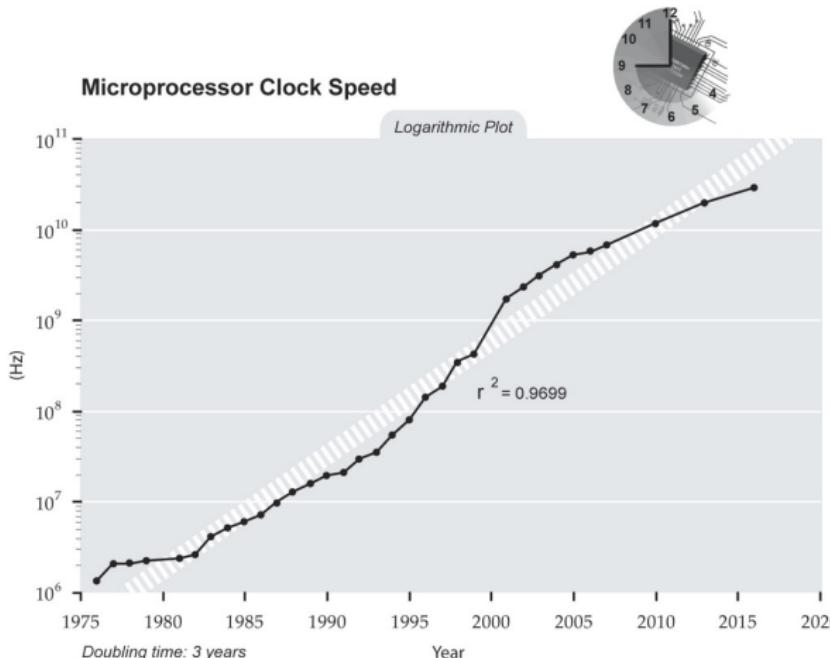


Abbildung 10: Prognostizierte Prozessorgeschwindigkeiten. Ray Kurzweil hat 2005 noch geglaubt, dass die Taktraten weiterhin parallel mit der Anzahl der Transistoren steigen.

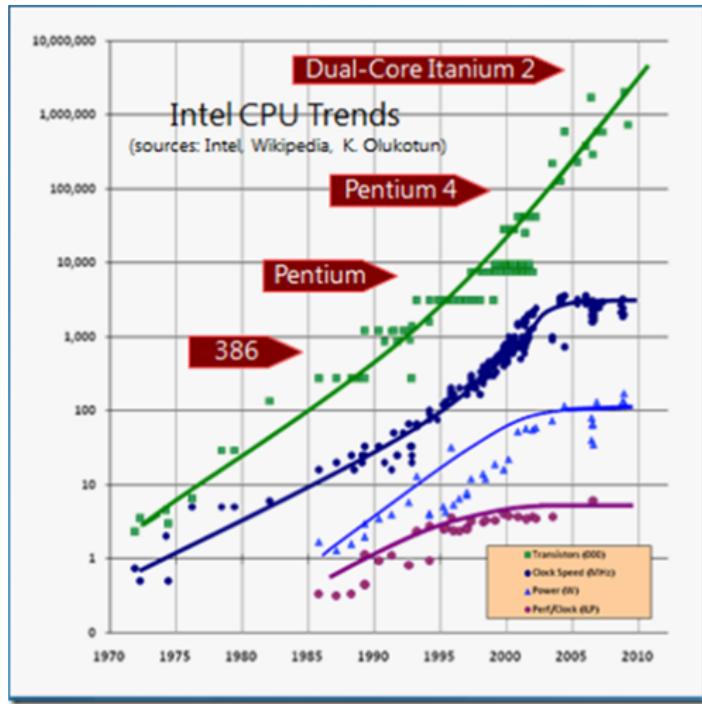


Abbildung 11: Tatsächliche Prozessorgeschwindigkeiten (Stand 2017).
 Grün: Anzahl Transistoren, dunkelblau: CPU-Geschwindigkeit.

Parallele Programmierung

Free lunch is over!

Früher reichte es aus, einfach zu warten und einen neuen Computer zu kaufen, um ein schnelleres Programm zu bekommen. Das ist vorbei. Stattdessen werden die Möglichkeiten von Multi-Core-Prozessoren durch Softwareentwicklungsmethoden unterstützt:

- Programme/Algorithmen werden parallelisiert.
- Dazu eignen sich besonders gut funktionale Programmieransätze.

Wir werden im Folgenden die Grundlagen der nebenläufigen (parallelen) Programmierung kennen lernen. Man unterscheidet zwischen *Threads* und *Prozessen*:

Arten von Nebenläufigkeit

Prozesse haben getrennte Adressräume (getrennter Heap). Die Kommunikation erfolgt nur über Inter Process Communication (IPC). Ein Prozess ist schwergewichtig und hat eigene Ressourcen (geöffnete Dateien, Sockets, Speicher etc.). Innerhalb eines Prozesses gibt es einen oder mehrere *Threads*.

Threads (Fäden) haben einen gemeinsamen Adressraum (gemeinsamer Heap), aber getrennte Stacks. Die Kommunikation erfolgt über den gemeinsamen Speicher. Sie sind leichtgewichtiger als Prozesse.

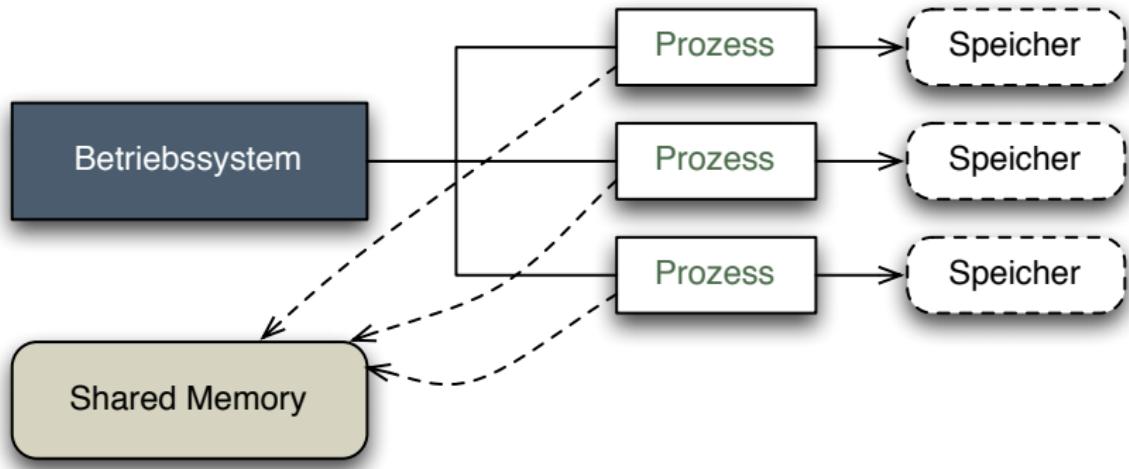


Abbildung 12: Prozesse.

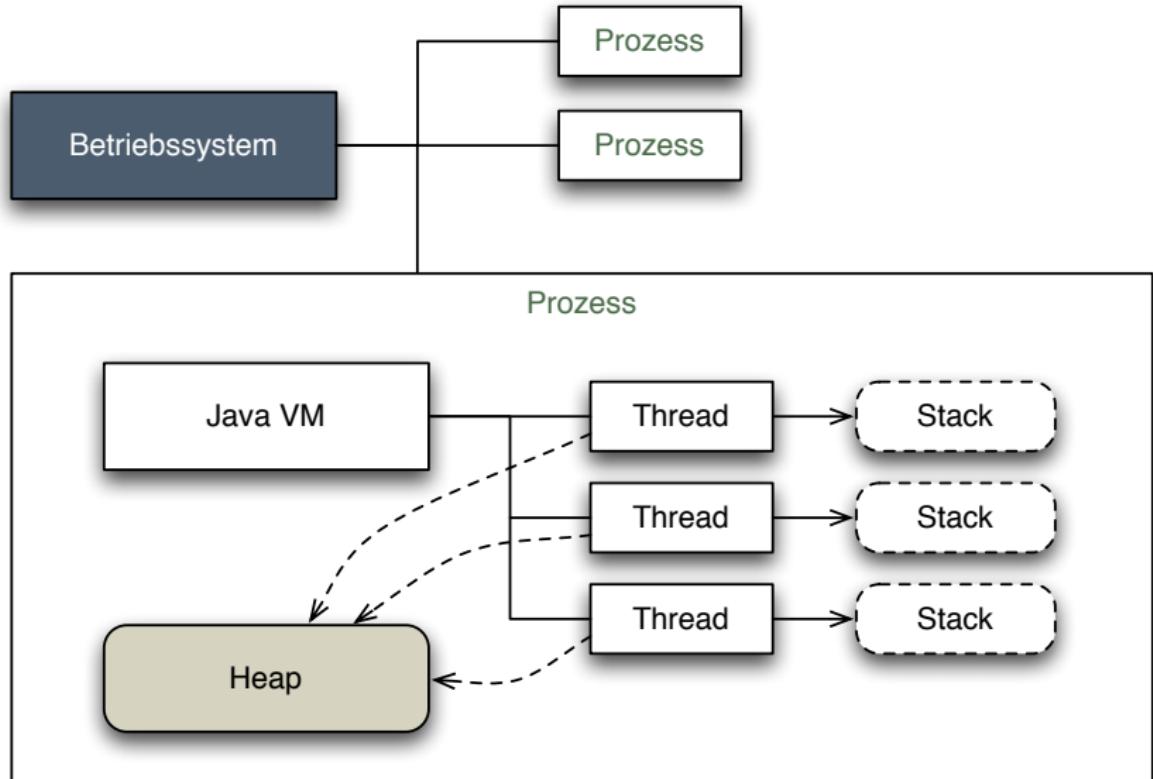


Abbildung 13: Threads.

Abschnittsübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Threads starten

Zustände eines Threads

Beenden von Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

synchronized-Blöcke

Blockieren und Aufwecken von Threads

Threads in Java

Es gibt zwei Möglichkeiten, Threads zu implementieren:

preemptives Scheduling (auch **native threads**): Die Verwaltung der Threads wird vom Betriebssystem übernommen. Wenn ein Thread seine Zeitscheibe aufgebraucht hat, wird er (auch gegen seinen Willen) unterbrochen und ein anderer Thread darf weiterarbeiten.

kooperatives Scheduling (auch **green threading**): Die Threads werden von der Virtuellen Maschine selbst verwaltet und geben sich gegenseitig Rechenzeit ab. Ein Thread läuft solange, bis er selbst der VM mitteilt, dass ein anderer Thread laufen soll.

Realisierung eines Threads

```
...
class Runner implements Runnable {
    public void run() {
        while (true) {
            System.out.println("Paralleler Thread");
        }
    }
}
...
```

Das Beispiel zeigt eine Klasse Runner, die das Runnable-Interface implementiert. Der Code, der in der run-Methode enthalten ist, wird parallel zum Hauptprogramm ausgeführt werden.

Beispiel für einen parallelen Ausführung

...

```
Thread t = new Thread(new Runner());  
t.start(); // Starte diesen Thread.  
  
while (true) {  
    System.out.println("Hauptprogramm");  
}  
...
```

Beispiel für einen parallelen Ausführung

...

```
Thread t = new Thread(new Runner());  
t.start(); // Starte diesen Thread.  
  
while (true) {  
    System.out.println("Hauptprogramm");  
}  
...
```

Konsole

```
Paralleler Thread  
Hauptprogramm  
Hauptprogramm  
Hauptprogramm  
Paralleler Thread  
Paralleler Thread
```

Parallele Ausführung mit Lambdas

...

```
Thread t = new Thread(  
    () -> { // Lambda-Funktion  
        while (true) {  
            System.out.println("Paraller Thread");  
        }  
    });  
t.start();  
  
while (true) {  
    System.out.println("Hauptprogramm");  
}
```

...

Parallele Ausführung mit Lambdas

...

```
Thread t = new Thread(  
    () -> { // Lambda-Funktion  
        while (true) {  
            System.out.println("Paraller Thread");  
        }  
    } );  
t.start();  
  
while (true) {  
    System.out.println("Hauptprogramm");  
}
```

...

Konsole

```
Paralleler Thread  
Hauptprogramm  
Hauptprogramm  
Paralleler Thread  
Paralleler Thread
```

Realisierung eines kooperativen Threads

```
...
class CooperativeRunner implements Runnable {
    public void run() {
        while (true) {
            System.out.println("Paralleler Thread");
            Thread.yield(); // Unterbreche Ausführung.
        }
    }
}
```

Der Unterschied zur preemptiven Variante ist, dass hier der Thread freiwillig Rechenzeit durch die Methode `Thread.yield()` abgibt.

Beispiel für kooperative parallele Ausführung

...

```
Thread t = new Thread(new CooperativeRunner());  
t.start(); // Starte diesen Thread.  
  
while (true) {  
    System.out.println("Hauptprogramm");  
    Thread.yield(); // Unterbreche Ausführung.  
}  
...
```

Beispiel für kooperative parallele Ausführung

...

```
Thread t = new Thread(new CooperativeRunner());  
t.start(); // Starte diesen Thread.  
  
while (true) {  
    System.out.println("Hauptprogramm");  
    Thread.yield(); // Unterbreche Ausführung.  
}
```

...

Konsole

```
Paralleler Thread  
Hauptprogramm  
Paralleler Thread  
Hauptprogramm
```

Wegen des nicht deterministischen Verhalten von `yield()` kann es auch vorkommen, dass viele Male hintereinander eine der beiden Threads Ausgaben machen kann.



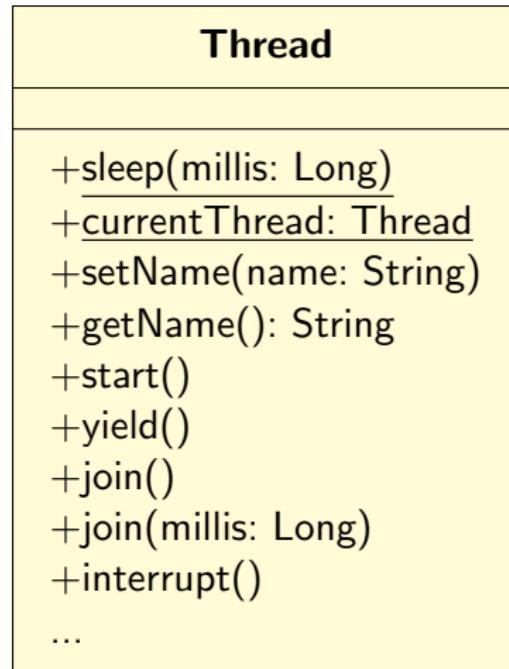


Abbildung 14: Die Klasse `java.lang.Thread`.

Die elementaren Methoden der Klasse Thread

- `Thread(Runnable target)` – Einen neuen Thread anlegen.
- `Thread(Runnable target, String name)` – Einen neuen Thread mit Namen `name` anlegen.
- `void setName(String name), String getName()` – Namen setzen und lesen.
- `void start()` – Thread starten.
- `static void yield()` – Rechenzeit abgeben.
- `static void sleep(long millis)` – Rechenzeit abgeben und für `millis` Millisekunden schlafen.
- `static Thread currentThread()` – Liefert den aktuellen Thread zurück.
- `void run()` – Thread implementiert selbst `Runnable`, man kann also von `Thread` ableiten und dann `run()` überschreiben (nicht empfohlen).

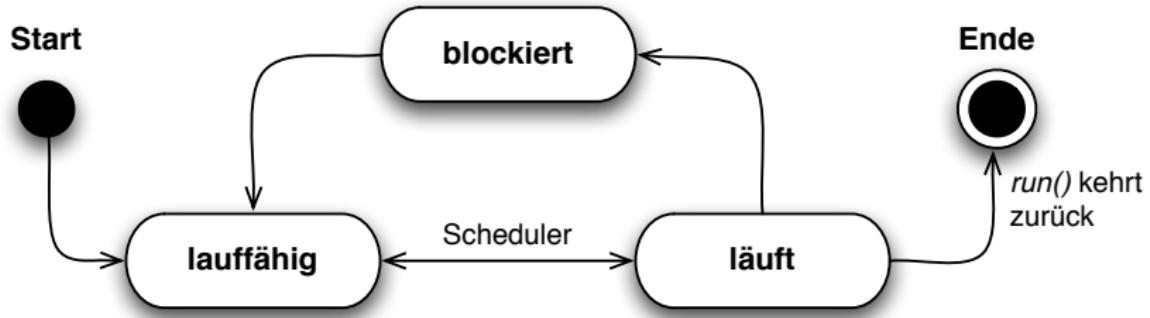


Abbildung 15: Zustnde eines Threads (einfache Version).

Beenden eines Threads

Ein Thread ist beendet, wenn die `run()`-Methode zurückkehrt, z. B. mit `return`. Da nur der Thread seinen internen Zustand kennt, ist die einzige adäquate Methode ihm zu signalisieren, dass er sich bei der nächsten Möglichkeit beenden soll.

Veraltete Methoden

Die Methoden

- `void stop()`, `void stop(Throwable obj)`
- `void destroy()`
- `void suspend()`, `void resume()`

gelten als *überholt (deprecated)* und sollten auf gar keinen Fall benutzt werden. Da sie den Thread einfach an einer beliebigen Stelle stoppen, können unvorhergesehene Probleme auftreten.

Beispiel für Beenden eines Threads mittels Flag

```
...
class StoppableByFlag implements Runnable {
    private volatile boolean cont = true;

    public void requestTermination() {
        cont = false;
    }

    public void run() {
        try {
            while (cont) {
                System.out.println("Thread laeuft");
                Thread.sleep(10); // Schlafe 10 ms.
            }
        } catch (InterruptedException e) {
            System.out.println("StoppableByFlag unterbrochen.");
        }
        System.out.println("** StoppableByFlag gestoppt **");
    }
}
```

Beispiel für Beenden eines Threads mittels Flag

...

```
StoppableByFlag st = new StoppableByFlag();
Thread thread = new Thread(st);
thread.start();
Thread.sleep(30); // Schlafe 30 ms.
st.requestTermination(); // Bitte Thread sich zu beenden.
```

...

Beispiel für Beenden eines Threads mittels Flag

...

```
StoppableByFlag st = new StoppableByFlag();
Thread thread = new Thread(st);
thread.start();
Thread.sleep(30); // Schlafe 30 ms.
st.requestTermination(); // Bitte Thread sich zu beenden.
```

...

Konsole

```
Thread laeuft
Thread laeuft
Thread laeuft
** StoppableByFlag gestoppt **
```

Beenden eines Threads mittels Interrupt

```
...
class StoppableByInterrupt implements Runnable {

    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Thread laeuft");
                Thread.sleep(10); // Schlafe 10 ms.
            }
        } catch (InterruptedException e) {
            System.out.println("StoppableByInterrupt unterbrochen.");
        }
        System.out.println("** StoppableByInterrupt gestoppt **");
    }
}
...
}
```

Beenden eines Threads mittels Interrupt

...

```
Thread thread = new Thread(new StoppableByInterrupt());
thread.start();
Thread.sleep(30); // Schlafe 30 ms.
thread.interrupt(); // Bitte Thread sich zu beenden.
```

...

Beenden eines Threads mittels Interrupt

...

```
Thread thread = new Thread(new StoppableByInterrupt());
thread.start();
Thread.sleep(30); // Schlafe 30 ms.
thread.interrupt(); // Bitte Thread sich zu beenden.
```

...

Konsole

```
Thread laeuft
Thread laeuft
Thread laeuft
StoppableByInterrupt unterbrochen.
** StoppableByInterrupt gestoppt **
```

Abschnittsübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

synchronized-Blöcke

Blockieren und Aufwecken von Threads

Threads vereinigen

Häufig möchte man, dass ein Thread erst weiterläuft, wenn andere Threads zu Ende gelaufen sind. Mit `join()` kann man darauf warten, dass ein anderer Thread fertig ist.

- `void join(long millis)` – Warte maximal `millis` Millisekunden (`0` = ewig).
- `void join(long millis, int nanos)` – Wartet maximal `millis` Milli- und `nanos` Nanosekunden.
- `void join()` – Wartet ewig.

Alle `join()`-Methoden sind unterbrechbar (**interruptible**), d. h. sie kehren mit einer `InterruptedException` zurück, wenn der Thread, der sie aufgerufen hat, unterbrochen wird.

Warten auf Threads

```
...
class RunnerPrinter implements Runnable {
    private String text;

    RunnerPrinter(String text) {
        this.text = text;
    }

    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(text);
            try {
                Thread.sleep((long) (Math.random() * 10));
            } catch (InterruptedException e) {
                System.out.println(text + " ist unterbrochen.");
            }
        }
        System.out.println(text + " ist fertig.");
    }
}
```

Warten auf Threads (Hauptprogramm)

...

```
Thread t1 = new Thread(new RunnerPrinter("Runner 1"));
Thread t2 = new Thread(new RunnerPrinter("Runner 2"));
Thread t3 = new Thread(new RunnerPrinter("Runner 3"));

t1.start(); // Starte alle drei Threads.
t2.start();
t3.start();
t1.join(); // Warte auf t1.
t2.join(); // Warte auf t2.
t3.join(); // Warte auf t3.

System.out.println("Alle fertig.");
...
```

Ausgabe

Konsole

```
Runner 1
Runner 2
Runner 3
Runner 2
Runner 2
Runner 1
Runner 3
Runner 3
Runner 2 ist fertig.
Runner 1
Runner 1 ist fertig.
Runner 3 ist fertig.
Alle fertig.
```

Die Reihenfolge der Ausgaben ist i. A. willkürlich, aber das Hauptprogramm wird erst dann weiter ausgeführt, wenn alle Threads beendet sind.

Prioritäten und Daemon-Threads

Man kann Threads Prioritäten zwischen 1 und 10 zuweisen:

- `void setPriority(int newPriority)`
- `int getPriority()`

Thread-Prioritäten sind nicht portabel und nicht zuverlässig. Ebenso kann man festlegen, was beim Beenden des Main-Threads passieren soll. Die VM beendet sich dann, wenn alle Nicht-Daemon-Threads beendet wurden.

- `void setDaemon(boolean on)`
- `boolean isDaemon()`

Abschnittsübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

Arten von Fehlern durch Nebenläufigkeit

Threadsafety

synchronized-Blöcke

Blockieren und Aufwecken von Threads



Abbildung 16: Das Therac-25-Disaster: Falsche Threadprogrammierung hat drei Patienten getötet, mindestens drei schwer verletzt. Quelle?

Herausforderungen von Nebenläufigkeit

- Threads arbeiten im selben Adressraum, daher müssen Zugriffe auf Speicher koordiniert und geordnet erfolgen.
- Die Probleme im Bereich der Threads sind nur sehr schwer zu reproduzieren und treten häufig erst im Produktivsystem auf.
- Parallelität und gemeinsame Speicher erfordern bestimmte Sprachkonstrukte, die selbst zu neuen Problemen führen.
- Menschen können nur schlecht in parallelen Abläufen denken.

Nebenläufigkeitsprobleme

Safety Hazards sind Probleme, bei denen sich das Programm in Anwesenheit mehrerer Threads nicht mehr korrekt verhält.

- race condition

Liveness Hazards sind Probleme, bei denen ein Programm bei mehreren Threads in einen Zustand gerät, bei dem es keine Fortschritte mehr machen kann.

- deadlock
- starvation
- livelock

Performance Hazards sind Probleme, bei denen ein Programm zwar korrekt funktioniert, die Performance trotz mehrerer Threads jedoch schlecht ist.

Race Condition: Label-Generator

```
...
class LabelGenerator {
    private int number1 = 0; // 1. Teil
    private int number2 = 0; // 2. Teil

    public String next() {
        number1++; // Nächste Zahl des 1. Teils.
        ...
        number2++; // Nächste Zahl des 2. Teils.
        String no = number1 + "-" + number2;
        return no;
    }
}
...
```

Offenbar generiert diese Klasse Label der Art 1-1, 2-2, 3-3 usw.

Race Condition: Benutzung des Label-Generators

```
...
class LabelConsumer implements Runnable {
    private LabelGenerator gen;
    private String consumerId;

    LabelConsumer(String id, LabelGenerator gen) {
        this.gen = gen;
        consumerId = id;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(consumerId + ": " + gen.next());
        }
    }
}
...
...
```

Diese Klasse fordert in der `run()`-Methode mehrere Male einen

Race Condition (Hauptprogramm)

Im Folgenden werden zwei solcher Consumer parallel ausgeführt.

```
...  
    LabelGenerator gen = new LabelGenerator(); // Genau eine  
    Instanz.  
    Thread t1 = new Thread(new LabelConsumer("C1", gen));  
    Thread t2 = new Thread(new LabelConsumer("C2", gen));  
    t1.start();  
    t2.start();  
...
```

Race Condition: Ausgabe

Konsole

```
C2: 1-1  
C2: 2-2  
C2: 3-3  
C2: 4-4  
C2: 5-5  
C1: 5-6  
C1: 6-7  
C1: 7-8  
C1: 8-9  
C1: 9-10
```

Eigentlich müssen beide Zahlenpaare stets gleich sein. Ab und zu ist dies nicht Fall – eine Race Condition liegt vor.

Deadlock: WaitThread

```
...
class WaitThread implements Runnable {
    public Thread other; // Referenz auf anderen Thread.

    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + " Start.");
        try {
            other.join(); // Warte auf Ende des anderen Threads.
        } catch (InterruptedException e) {
            System.out.println(name + " unterbrochen");
        }
        System.out.println(name + " Ende.");
    }
}
...
...
```

Ein WaitThread wartet auf das Ende eines anderen Threads.

Deadlock (Hauptprogramm)

```
...
WaitThread wt1 = new WaitThread();
WaitThread wt2 = new WaitThread();
Thread t1 = new Thread(wt1, "t1");
Thread t2 = new Thread(wt2, "t2");
wt1.other = t2; // Weise die Threads t1 und t2 über Kreuz zu.
wt2.other = t1;
t1.start();
t2.start();
...
System.out.println("Hauptprogramm beendet.");
...
```

Die Programm bleibt hängen.

Deadlock: Auflösung

```
...
WaitThread wt1 = new WaitThread();
WaitThread wt2 = new WaitThread();
Thread t1 = new Thread(wt1, "t1");
Thread t2 = new Thread(wt2, "t2");
wt1.other = t2; // Weise die Threads t1 und t2 über Kreuz zu.
wt2.other = t1;
t1.start();
t2.start();
Thread.sleep(2000);
t1.interrupt(); // Jetzt wird das Deadlock aufgelöst.
System.out.println("Hauptprogramm beendet.");
...
...
```

Sobald ein Thread beendet wird, kann der andere auch weiterlaufen.

Deadlock: Ausgabe

Konsole

```
t1 Start.  
t2 Start.  
Hauptprogramm beendet.  
t1 unterbrochen  
t1 Ende.  
t2 Ende.
```

Hinweis: Vor der Ausgabe „Hauptprogramm beendet.“ wartet das Programm 2000 ms.

Threadsafety

Threadsicher

Eine Klasse ist **threadsicher (threadsafe)**, wenn sie bei der Verwendung mehrerer Threads keine der in Abschnitt ?? genannten Probleme zeigt.

Viele Klassen der Klassenbibliothek sind nicht threadsafe und müssen explizit geschützt werden. Von einigen Klassen gibt es zwei Varianten (`StringBuffer`, `StringBuilder`), die eine threadsafe, die andere nicht.

Korollar: Unveränderliche Objekte

Unveränderliche (immutable) Objekte sind immer threadsicher.

Deshalb ist die funktionale Programmierung (wieder) interessant, da hier keine Variablen verändert werden.

Weitere Kriterien für Threadsafety

- Atomizität** Ist eine Operation atomar (**atomic**), wird sie entweder vollständig oder gar nicht ausgeführt – eine Race Condition kann nicht auftreten. Ist sie nicht atomar, können zwei Threads in Konflikt geraten.
- Reihenfolge** Die Reihenfolge (**ordering**) im Quelltext entspricht nicht immer der Reihenfolge der Ausführung, daher braucht man Regeln, unter welchen Bedingungen die Reihenfolge beachtet wird.
- Sichtbarkeit** Durch Caches und Register gibt es mehrere Kopien der Daten im Speicher. Deswegen ist es wichtig zu definieren, wann eine Änderung in einer lokalen Kopie des Speichers durch einen Thread an andere Threads weitergegeben wird (**visibility**).

Atomare Operationen

Ohne explizite Synchronisation sind nur folgende Operationen in Java atomar:

- Lesen eines volatilen 32-bit Feldes (byte, short, int, float, char, boolean).
- Schreiben eines volatilen 32-bit Felds (byte, short, int, float, char, boolean).

Nicht atomar dagegen sind:

- Lesen oder Schreiben eines 64-bit Feldes (long, double)
- Inkrementieren oder Dekrementieren eines 32-bit Feldes:
`int i = 0; i++;`

Ordering in Java: “within-thread as-if-serial”

Die Programmausführung und der Speicher müssen sich auch in einem Threads so verhalten, als ob die Instruktionen sequentiell ausgeführt würden. Die VM darf beliebige Optimierungen machen, solange diese Bedingung erhalten bleibt.

Quelltext	Alternative
int i, k, m;	int i, k, m;
i = 15;	k = 18;
i++;	i = 15;
k = 18;	k++;
k++;	i++;
m = i + k;	m = i + k;

Tabelle 1: Beispiel für das erlaubte Umsortieren von Anweisungen.

volatile

Das Schlüsselwort `volatile` wird übersprungen.

Abschnittsübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

synchronized-Blöcke

Blockieren und Aufwecken von Threads

Das Schlüsselwort `synchronized`

Wie kann man dafür sorgen, dass ein bestimmtes Code-Segment nur von einem einzigen Thread gleichzeitig durchlaufen werden kann, so dass diese Sequenz sich atomar verhält? Benötigt wird ein Mechanismus, um Bereiche sperren (to lock) zu können, sobald ein Thread eingetreten ist und andere Threads damit auszusperren (**mutual exclusion, Mutex**). Das Schlüsselwort **synchronized** stellt diesen zur Verfügung als:

- **synchronized Methoden**
- **synchronized Blöcke**

Jedes Java-Objekt kann für die **Sperre (lock)** zusammen mit `synchronized` verwendet werden.

Threadsicher: Label-Generator

```
...
class SyncLabelGenerator {
    private int number1 = 0; // 1. Teil
    private int number2 = 0; // 2. Teil

    ...
    public synchronized String next() {
        number1++; // Nächste Zahl des 1. Teils.
        ...
        number2++; // Nächste Zahl des 2. Teils.
        String no = number1 + "-" + number2;
        return no;
    }
}
...

```

Neu ist das synchronized-Statement in der next()-Methode.

Threadsicher: Benutzung des Label-Generators

```
...
class SyncLabelConsumer implements Runnable {
    private SyncLabelGenerator gen;
    private String consumerId;

    SyncLabelConsumer(String id, SyncLabelGenerator gen) {
        this.gen = gen;
        consumerId = id;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(gen.next());
        }
    }
}
...
...
```

Diese Klasse fordert in der `run()`-Methode mehrere Male eine neue
Nummer an

Threadsicher: (Hauptprogramm)

Im Folgenden werden zwei solcher Consumer parallel ausgeführt.

```
...
SyncLabelGenerator gen =
    new SyncLabelGenerator();
Thread t1 = new Thread(new SyncLabelConsumer("C1", gen));
Thread t2 = new Thread(new SyncLabelConsumer("C2", gen));
t1.start();
t2.start();
...
```

Threadsicher: Ausgabe

Konsole

```
1-1  
2-2  
3-3  
4-4  
5-5  
6-6  
7-7  
8-8  
9-9  
10-10
```

Jetzt sind die beiden Labelpaare stets gleich sein.

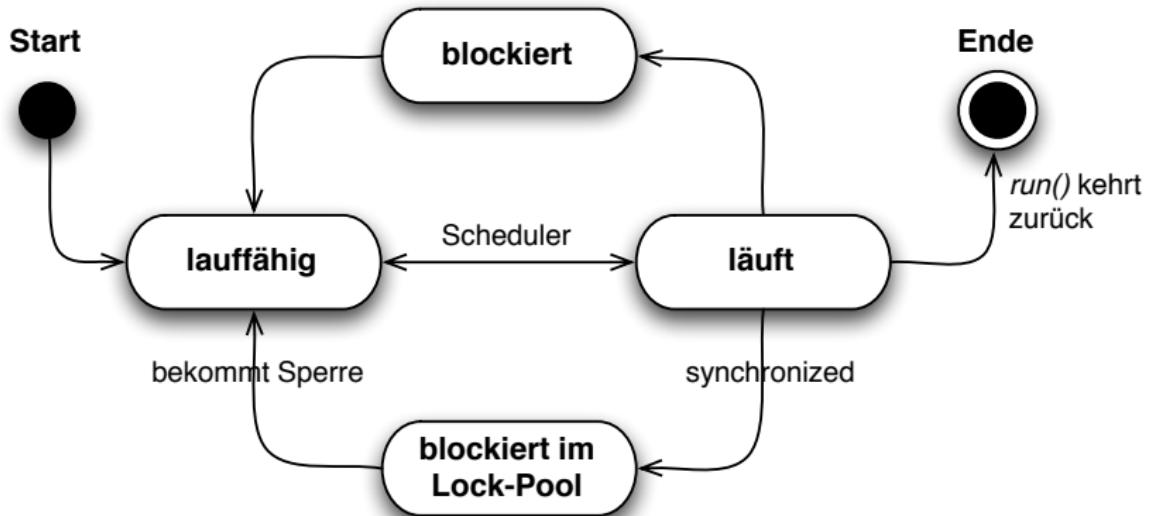


Abbildung 17: Synchronized im Zustandsdiagramm.

Java Locks sind reentrant

Der synchronized-Mechanismus ist reentrant: Hat ein Thread bereits das Token eines bestimmten Objektes, kann er in alle anderen synchronized-Blöcke eintreten, die von dem Objekt geschützt werden. Ansonsten gäbe es permanent **Verklemmungen (deadlocks)**.

```
...
class SyncLabelGenerator {
...
    public synchronized String resetAndNext() {
        number1 = 0; // Setze Zähler zurück.
        number2 = 0;
        return next(); // next() ist ebenfalls synchronized!
    }
...
}
```

Abschnittsübersicht

4 Grundlagen von Nebenläufigkeit

Einführung

Nebenläufigkeit durch Threads

Warten auf das Ende anderer Threads

Seiteneffekte durch parallele Programmausführung

synchronized-Blöcke

Blockieren und Aufwecken von Threads

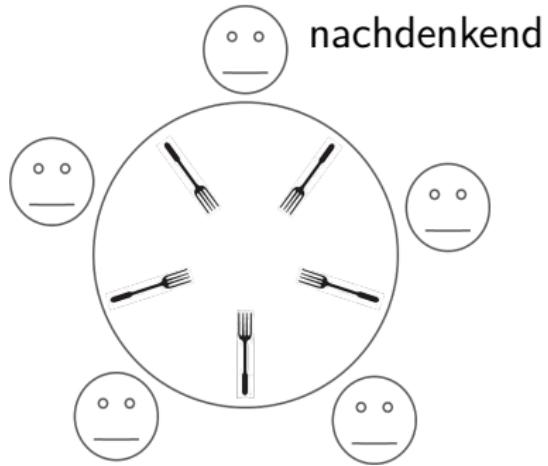
Die speisenden Philosophen

Um einen runden Tisch sitzen fünf Philosophen, die sich mit ihren Nachbarn je eine linke und rechte Gabel teilen. Zum Essen der Mahlzeit in der Mitte des Tisch benötigt man beide Gabeln.

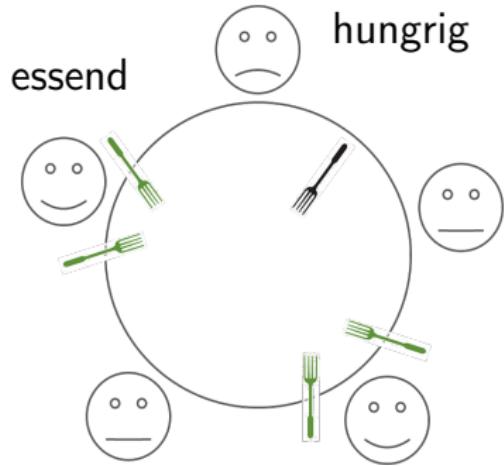
denkend Jeder denkt für eine Weile nach.

hungrig Dann wird er hungrig und versucht, sowohl die rechte wie auch die linke Gabel vom Tisch zu nehmen um zu essen. Möglicherweise muss er warten, da ein oder beide Nachbarn die Gabel selbst benutzen.

essend Hat ein hungriger Philosoph beide Gabeln, isst er eine gewisse Zeit, legt dann die Gabel wieder zurück und wird wieder nachdenklich.



(a) Ausgangssituation



(b) Irgendwann

Abbildung 18: Die speisenden Philosophen.

Aufgabe (10 min.)

Deadlock finden

Bei diesem Philosophen-Problem kann es zu einem Deadlock kommen. Finden Sie dieses und erklären sie es.

Zugriff auf die Gabel muss koordiniert erfolgen

Offenbar ist es wichtig, dass ein Philosoph beide Gabeln auf einmal oder gar keine nimmt. Aus diesem Grund wird der Zugriff auf den Tisch synchronisiert, so dass nur ein Philosoph gleichzeitig Gabel nehmen oder zurücklegen kann.

Nur eine Gabel?

Was aber passiert, wenn ein Philosoph mit exklusivem Zugriff auf dem Tisch erkennt, dass er nur eine Gabel bekommt?

In diesem Fall muss er warten, bis ein anderer Philosoph eine noch benötigte Gabel wieder zurücklegt. In Java gibt es hierfür die Methoden `wait()` und `notifyAll()`.

Philosophen: Tisch

```
...
class Table {
    private boolean[] forks;

    public Table(int N) {
        // Initialisiere Array der Länge N mit true:
        forks = new boolean[N];
        for (int i = 0; i < forks.length; i++) {
            forks[i] = true; // true: Gabel liegt auf dem Tisch.
        }
    }

    public boolean isForkAvailable(int f) {
        return forks[f];
    }

    public void takeFork(int f) {
        forks[f] = false; // false: Gabel hat ein Philosoph.
    }

    public void putFork(int f) {
        forks[f] = true; // Gabel liegt wieder auf dem Tisch.
    }
}
...
```

Philosophen: Hauptprogramm

Im Folgenden werden fünf Philosophen-Threads parallel ausgeführt.

...

```
int N = 5; // Anzahl Philosophen.  
Table table = new Table(N); // Tisch mit N Gabeln.  
for (int i = 0; i < N; i++) { // Aktiviere N Philosophen...  
    // Parameter: linke u. rechte Gabel sowie der Tisch:  
    Philospher p = new Philospher(i, (i + 1) % N, table);  
    Thread t = new Thread(p);  
    t.start();  
}
```

...

Philosophen: Philosoph (Teil 1)

```
...
class Philosopher implements Runnable {

    private int leftFork, rightFork; // Linke u. rechte Gabel.
    private int philId; // Eindeutige Nr. für den Philosophen.
    private Table table; // Sein Tisch.

    Philosopher(int leftFork, int rightFork, Table table) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
        this.table = table;
        this.philId = leftFork + 1; // Id von 1 bis N.
    }

    public void run() {
        while (true) { // Philosophen denken und essen für immer...
            System.out.println(philId + " denkt."); // Nachdenken.
            try {
                Thread.sleep((int) (Math.random() * 500));
            } catch (InterruptedException e) {
            }
            synchronized (table) { // Greife exklusiv auf den Tisch zu.
                while (!(table.isForkAvailable(leftFork) &&
```

Philosophen: Philosoph (Teil 2)

```
        table.isForkAvailable(rightFork))) {
    try {
        // Es sind (noch) nicht beide Gabel verfügbar.
        table.wait(); // Warte, bis sich etwas am Tisch ändert.
    } catch (InterruptedException e) {
    }
}
table.takeFork(leftFork); // Nimm beide Gabeln.
table.takeFork(rightFork);
}
System.out.println(phiId + " isst."); // Essen.
try {
    Thread.sleep((int) (Math.random() * 300));
} catch (InterruptedException e) {
}
synchronized (table) {
    table.putFork(leftFork); // Leg beide Gabeln zurück.
    table.putFork(rightFork);
    table.notifyAll(); // Teile Änderung mit.
}
}
```

Philosophen: Philosoph (Teil 3)

}

...

wait() und notify()

Die Klasse Object bietet die folgenden drei Methoden an:

- `wait()` – legt einen Thread schlafen.
- `notify()` – weckt **einen** der Threads wieder auf, die sich mit `wait()` schlafen gelegt haben.
- `notifyAll()` – weckt alle wartenden Threads wieder auf.

`wait()`, `notify()` und `notifyAll()` dürfen nur auf Objekten aufgerufen werden, für die der Thread das Lock hält.

Test

`wait()` sollte immer in einer Schleife verwendet werden, da der Test noch einmal wiederholt werden muss – der Thread weiß nicht, warum er geweckt wurde.

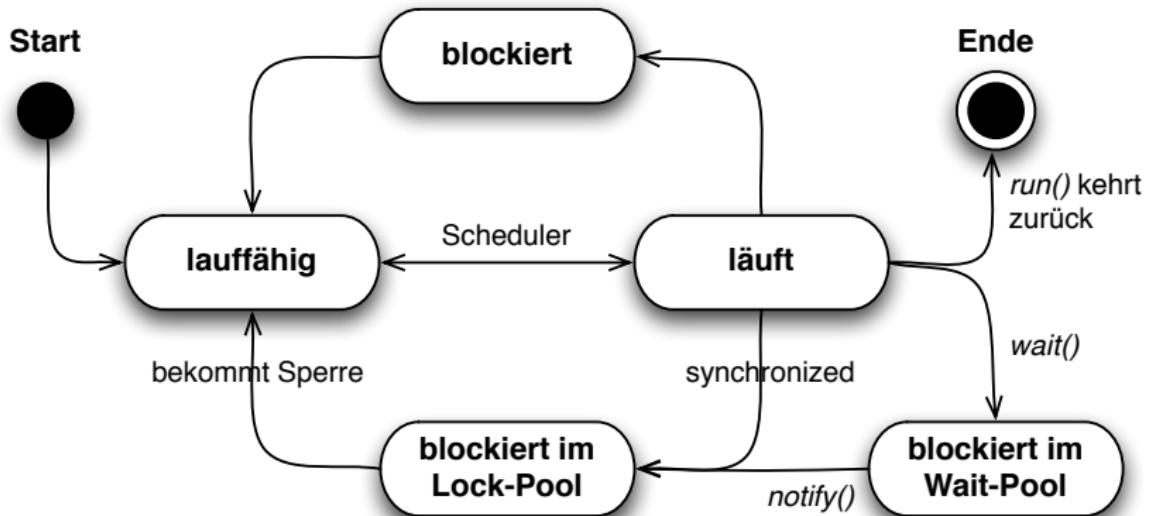


Abbildung 19: Zustandsdiagramm mit Wait-Pool.

Containerklassen in der praktischen Anwendung

Kapitelübersicht

5 Java Collections-Framework

Übersicht über die Datenstrukturen

Iteratoren

Collections und Nebenläufigkeit

Motivation für ein Collections-Framework

Fast immer gibt es in Programmen die Notwendigkeit, Daten strukturiert zu verwalten. Ein Beispiel ist ein Array.

Datenstruktur

Alle Techniken, die helfen Daten strukturiert abzulegen, werden unter dem Begriff Datenstrukturen zusammengefasst.

In objekt-orientierten Sprachen werden Objekte in Datenstrukturen gespeichert, und die Datenstruktur selbst als Objekt realisiert.

Container-Klasse

Eine Container-Klasse ist der Oberbegriff für objektorientierte Datenstrukturen, die Objekte verwaltet.

Dynamische Datenstrukturen

Bisher haben wir nur auf Strukturen fester Länge (Arrays) gearbeitet. Oft weiß man nicht im Vorhinein, wie viele Elemente in einer Datenstruktur (Container) untergebracht werden müssen. Dann braucht man Datenstrukturen, die beliebig viele Elemente aufnehmen können. Diese Datenstrukturen können wachsen und schrumpfen – man nennt sie deshalb auch dynamische Datenstrukturen.

Arrays sind nicht dynamisch

Die Länge eines Arrays kann zwar zur Laufzeit variabel sein, wenn bei der Erzeugung des Arrays eine Variable als Länge genommen wird. Danach ist die Länge aber konstant.

Beispiele für Anforderungen an Datenstrukturen

- Ist der Student Max Müller bereits eingeschrieben? Oder allgemein: Ist ein Element in einer Menge von gleichartigen Elementen enthalten?
- Welcher Druckauftrag wird als nächstes gedruckt?
- In welcher Reihenfolge haben sich Musikfans für Freikarten beworben? Wer steht an Position eins bis zehn?
- Filtere alle Stimmabgaben (bei einer Wahl) heraus, die ungültig sind.
- usw.

Elementare Datenstrukturen (Container)

- Listen** (List) Hier haben Elemente eine Reihenfolge. Varianten davon können über einen Index adressiert werden oder die Datenstrukturen Kellerspeicher (Stack) und Warteschlange (Queue) realisieren.
- Mengen** (Set) Der Container verhält sich wie eine mathematische Menge. Es sind keine Duplikate erlaubt.
- Maps** Maps sind wie eine Menge, wobei die Elemente in einem Schlüssel (key) und den zugehörigen Wert (value) unterteilt sind.

Hinweis

In diesem Abschnitt behandeln wir die Containerklassen aus Sicht der Programmieranwenders, der diese über eine Schnittstelle (Interface) anspricht. Die Implementierungen selbst sind Teil des Kapitels ??.

Abschnittsübersicht

5 Java Collections-Framework

Übersicht über die Datenstrukturen

Collection-Interface

Liste (List)

Mengen (Set)

Maps

Iteratoren

Collections und Nebenläufigkeit

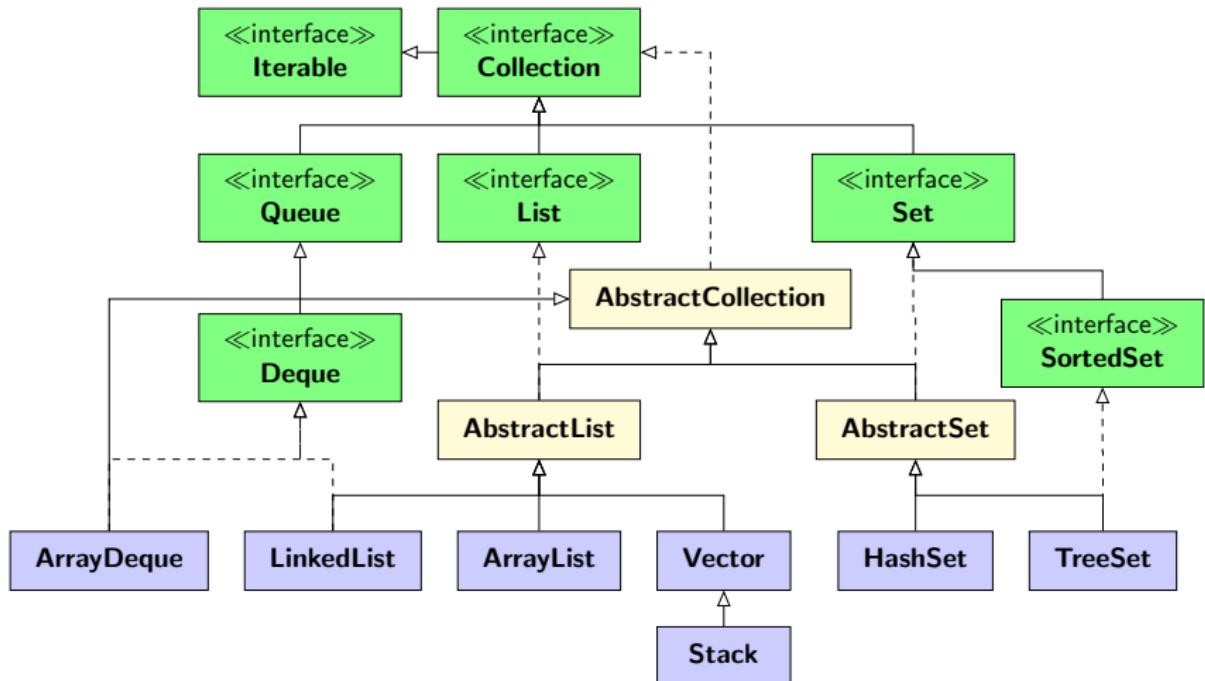


Abbildung 20: Die Interfaces und Klassen des Java-Collections-Framework für Collections.

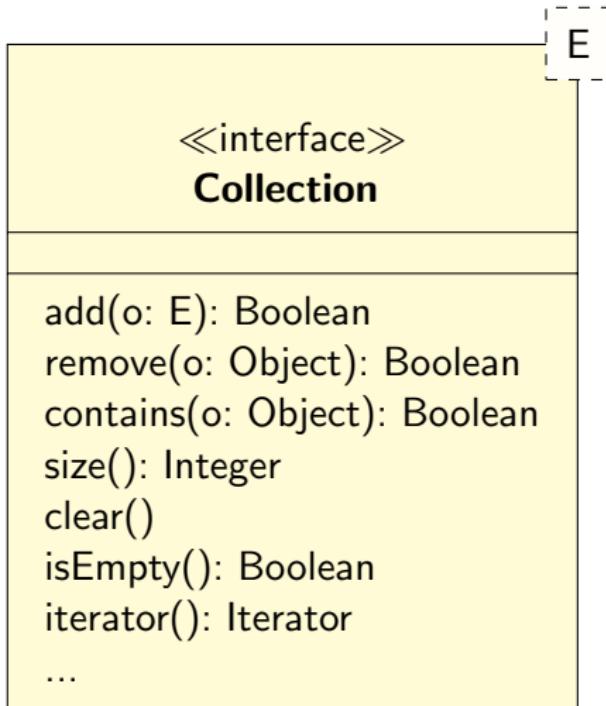


Abbildung 21: Ausschnitt des Interfaces `java.util.Collection`. E ist ein Typ-Parameter (siehe Abschnitt ??).

Elementare Methoden einer Collection

- `boolean add(E e)` – Fügt ein Element e der Containerklasse hinzu und ergibt wahr, wenn dies möglich war, andernfalls falsch.
- `boolean remove(Object o)` – Entfernt das Objekt o aus der Containerklasse. Falls es nicht gefunden wurde, wird falsch zurückgegeben.
- `boolean contains(Object o)` – Sucht ein Element und ergibt wahr, wenn mindestens ein Element gefunden wird, sonst falsch.
- `boolean isEmpty()` – Liefert wahr genau dann, wenn die Containerklasse leer ist, andernfalls falsch.
- `int size()` – Ergibt die Anzahl der Elemente in der Containerklasse.
- `void clear()` – Entfernt alle Elemente aus der Containerklasse.

Listen

Listen sind Containerklassen, die ihre Elemente in einer bekannten Reihenfolge speichern. Damit ist es u. a. möglich, über einen Index auf die Elemente zuzugreifen. Außerdem ist es erlaubt, Duplikate der Datenstruktur hinzuzufügen.

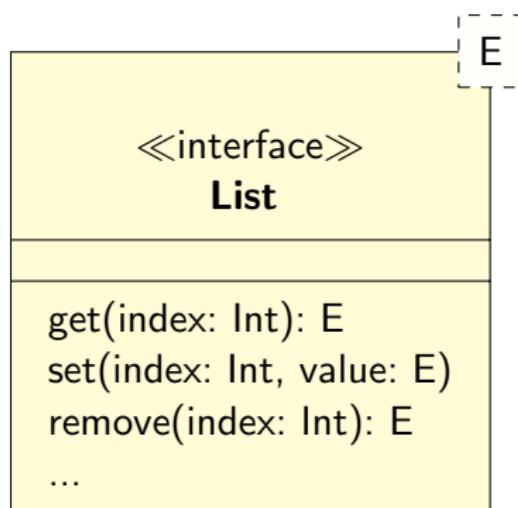


Abbildung 22: Auszug der Methoden des Interfaces `java.util.List`.

Elementare Methoden des Interfaces List

- `E get(int idx)` – Gibt das Element an der Indexposition `idx` zurück.
- `E set(int idx, E e)` – Ersetzt das Element an Position `idx` mit `e`. Das alte Element wird zurückgegeben.
- `E remove(int idx)` – Entfernt das Element an Position `idx` aus der Containerklasse und gibt es zurück.
- uvm.

Achtung

Ist die Indexposition ungültig, wird eine `IndexOutOfBoundsException`-Exception geworfen.

Beispiel

```
...
List<String> list = new LinkedList<>();
list.add("Hallo");
list.add("Welt");
list.add("Welt"); // Duplikat
for (String e : list) { // foreach-Anweisung möglich.
    System.out.print(e + " ");
}
System.out.println("\nAnzahl Elemente: " + list.size());
list.clear(); // Alle Elemente löschen.
System.out.println("Anzahl Elemente: " + list.size());
...

```

Konsole

```
Hallo Welt Welt
Anzahl Elemente: 3
Anzahl Elemente: 0
```

Anmerkungen

Die Liste im vorigen Listing ist eine `LinkedList`. Man sagt, die Datenstruktur `List` wird als `LinkedList` implementiert. Genauso gut wäre es möglich, die Implementierung mit einer `ArrayList` oder einem `Vector` auszutauschen. Welche verschiedene Implementierungen es gibt und was die Unterschiede sowie Vor- und Nachteile sind, sehen wir im Kapitel ??.

Beispiel für spezielle Listenmethoden

```
...
List<Integer> list = new ArrayList<>();
for (int i = 1; i <= 10; i++) {
    list.add(i); // Zahlen 1 bis 10 hinzufügen...
}
for (int e : list) {
    System.out.print(e + " ");
}
System.out.println(""); // Neue Zeile.
list.set(1, 42); // 2. Element austauschen (wie beim Array).
list.remove(8); // Vorletztes Element (9) löschen.
list.remove(Integer.valueOf(5)); // Wert 5 löschen.
for (int e : list) {
    System.out.print(e + " ");
}
...

```

Konsole

```
1 2 3 4 5 6 7 8 9 10
1 42 3 4 6 7 8 10
```

Stack, Queue und Deque

Das Java-Collections-Framework sieht auch Schnittstellen für einen Stack und Warteschlangen vor. Diese Containerklassen finden in erster Linie Anwendungen in Datenstrukturen und Algorithmen und werden deshalb ausführlich in den Abschnitten ?? und ?? behandelt. Einen Stack lernen wir im Abschnitt ?? bereits kennen.

Interface Set

Das Interface Set steht für eine Containerklasse, die Elemente gemäß einer mathematischen Menge verwaltet. Gegenüber dem Interface Collection gibt es keine neuen Methoden – stattdessen ist dieses Interface als ein Marker-Interface gedacht. D. h. es soll dem Programmierer zeigen, dass die implementierende Klasse die Elemente wie eine Menge organisiert.

Beispiel für eine Set

```
...
Set<String> set = new HashSet<>();
String[] chord = new String[]{"E", "G#", "H", "D", "E"}; // E7
for (String note : chord) {
    boolean res = set.add(note);
    if (!res) {
        System.out.println(note + " konnte nicht hinzugefuegt werden.");
    }
}
System.out.println(set.toString()); // HashSet hat toString().
set.remove("D"); // Note D löschen.
System.out.println(set.toString());
...

```

Beispiel für eine Set

```
...
Set<String> set = new HashSet<>();
String[] chord = new String[]{"E", "G#", "H", "D", "E"}; // E7
for (String note : chord) {
    boolean res = set.add(note);
    if (!res) {
        System.out.println(note + " konnte nicht hinzugefuegt werden.");
    }
}
System.out.println(set.toString()); // HashSet hat toString().
set.remove("D"); // Note D löschen.
System.out.println(set.toString());
...
}
```

Konsole

```
E konnte nicht hinzugefuegt werden.
[D, E, H, G#]
[E, H, G#]
```

Allgemein gilt bei Mengen, dass die Reihenfolge, wie die Elemente gespeichert sind, keine Rolle spielt.

Beispiel für eine SortedSet

```
...
Set<String> set = new TreeSet<>(); // Sortierte Menge.
String[] chord = new String[]{"E", "G#", "H", "D", "E"}; // E7
for (String note : chord) {
    boolean res = set.add(note);
    if (!res) {
        System.out.println(note + " konnte nicht hinzugefuegt werden.");
    }
}
System.out.println(set.toString()); // HashSet hat toString().
set.remove("D"); // Note D löschen.
System.out.println(set.toString());
...
}
```

Beispiel für eine SortedSet

```
...
Set<String> set = new TreeSet<>(); // Sortierte Menge.
String[] chord = new String[]{"E", "G#", "H", "D", "E"}; // E7
for (String note : chord) {
    boolean res = set.add(note);
    if (!res) {
        System.out.println(note + " konnte nicht hinzugefuegt werden.");
    }
}
System.out.println(set.toString()); // HashSet hat toString().
set.remove("D"); // Note D löschen.
System.out.println(set.toString());
...

```

Konsole

```
E konnte nicht hinzugefuegt werden.
```

```
[D, E, G#, H]
```

```
[E, G#, H]
```

Es gibt mit der Klasse TreeSet die Möglichkeit, die Elemente sortiert zu speichern.

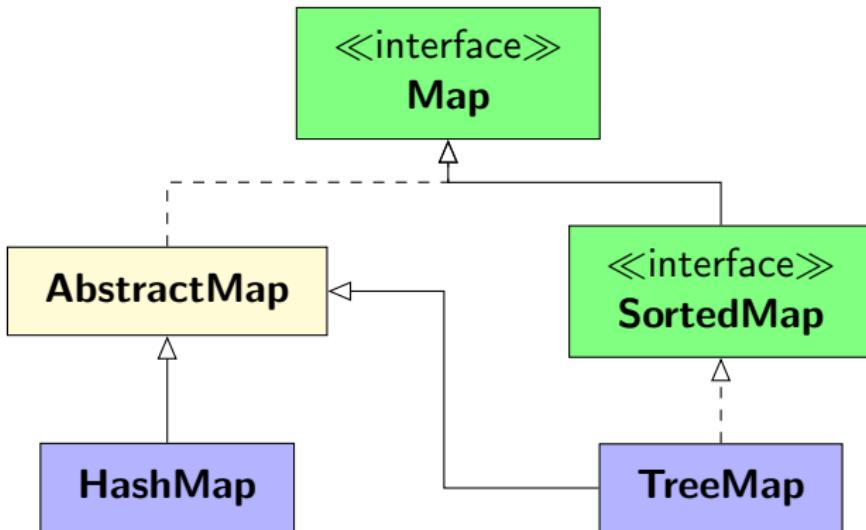


Abbildung 23: Die Interfaces und Klassen für Maps im Java-Collections-Framework.

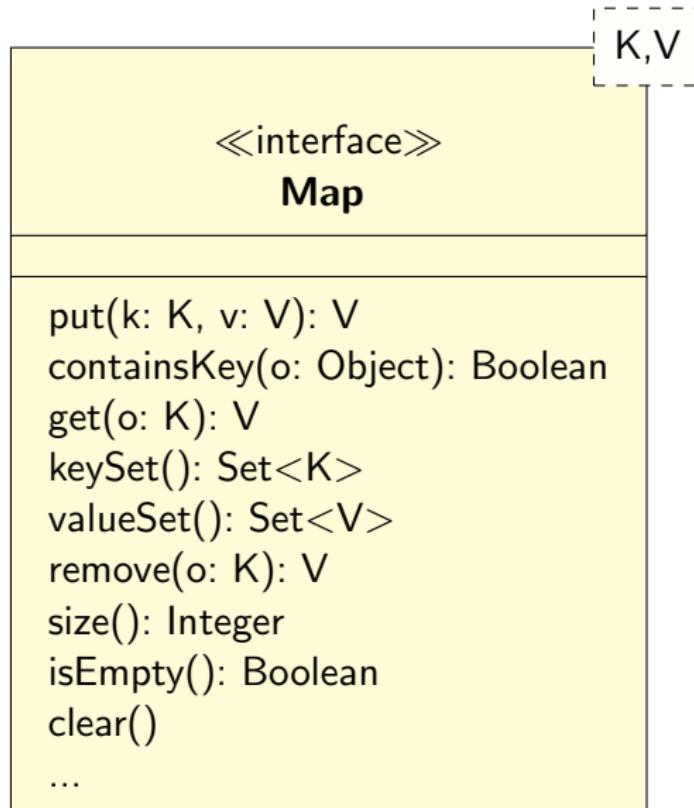


Abbildung 24: Ausschnitt des Interfaces Map. K und V sind Typ-Parameter für die Keys und Values.

Elementare Methoden von Map

- `V put(K key, V value)` – Setzt Schlüssel `key` und Wert `value`.
- `V get(Object key)` – Liest den Wert zum Schlüssel `key`.
- `Set<K> keySet()` – Alle Schlüssel als Set.
- `Set<V> valueSet()` – Alle Werte als Set.
- `boolean containsKey(Object key)` – Testet, ob der Schlüssel `key` vorhanden ist.
- `V remove(Object key)` – Entfernt den Wert zu `key` (und `key` selbst).
- `int size()` – Anzahl der Elemente.
- `boolean isEmpty()` – Überprüft, ob die Map leer ist (`size() == 0`).
- `void clear()` – Löscht die Map.

Beispiel für eine Map

...

```
Map<Integer, String> errorsDE = new HashMap<>();
Map<Integer, String> errorsEN = new HashMap<>();
errorsDE.put(1, "Ungueltige Indexposition!");
errorsDE.put(2, "Teilen durch 0 nicht erlaubt!");
errorsEN.put(1, "Invalid index position!");
errorsEN.put(2, "Division by zero error!");

for (Integer id : errorsDE.keySet()) {
    System.out.println(errorsDE.get(id) + " <=> " +
        errorsEN.get(id));
}
```

...

Beispiel für eine Map

...

```
Map<Integer, String> errorsDE = new HashMap<>();
Map<Integer, String> errorsEN = new HashMap<>();
errorsDE.put(1, "Ungueltige Indexposition!");
errorsDE.put(2, "Teilen durch 0 nicht erlaubt!");
errorsEN.put(1, "Invalid index position!");
errorsEN.put(2, "Division by zero error!");

for (Integer id : errorsDE.keySet()) {
    System.out.println(errorsDE.get(id) + " <=> " +
        errorsEN.get(id));
}
```

...

Konsole

```
Ungueltige Indexposition! <=> Invalid index position!
Teilen durch 0 nicht erlaubt! <=> Division by zero error!
```

Abschnittsübersicht

5 Java Collections-Framework

Übersicht über die Datenstrukturen

Iteratoren

Collections und Nebenläufigkeit

Iterable und Iterator

Iterable-Instanzen liefern einen Iterator zurück. Iterable und Iterator sind außerdem die Grundlage für die foreach-Schleife. Die Methoden auf Iterator<E> sind:

- boolean `hasNext()` - Gibt an, ob noch Elemente vorhanden sind.
- E `next()` - Geht zum nächsten Element und gibt es zurück.
- `remove()` - Entfernt das aktuelle Element (muss nicht unterstützt werden).

Ein Iterator kann nur vorwärts laufen – für bestimmte List-basierte Collections gibt es den ListIterator, der auch rückwärts navigieren kann.

Beispiel für einen Iterator

```
...
List<String> list = Arrays.asList( // Hilfsmethode
    "Hallo", "Welt!", "Oh", "schoene", "Welt!");
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    System.out.print(it.next() + " ");
}
System.out.println();
for (String word : list) { // Alternative Syntax.
    System.out.print(word + " ");
}
System.out.println();
Set<String> set = new HashSet<>(list); // Elemente werden eingefügt.
for (String word : set) {
    System.out.print(word + " ");
}
...
...
```

Beispiel für einen Iterator

```
...
List<String> list = Arrays.asList( // Hilfsmethode
    "Hallo", "Welt!", "Oh", "schoene", "Welt!");
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    System.out.print(it.next() + " ");
}
System.out.println();
for (String word : list) { // Alternative Syntax.
    System.out.print(word + " ");
}
System.out.println();
Set<String> set = new HashSet<>(list); // Elemente werden eingefügt.
for (String word : set) {
    System.out.print(word + " ");
}
...

```

Konsole

```
Hallo Welt! Oh schoene Welt!
Hallo Welt! Oh schoene Welt!
Hallo Welt! Oh schoene
```

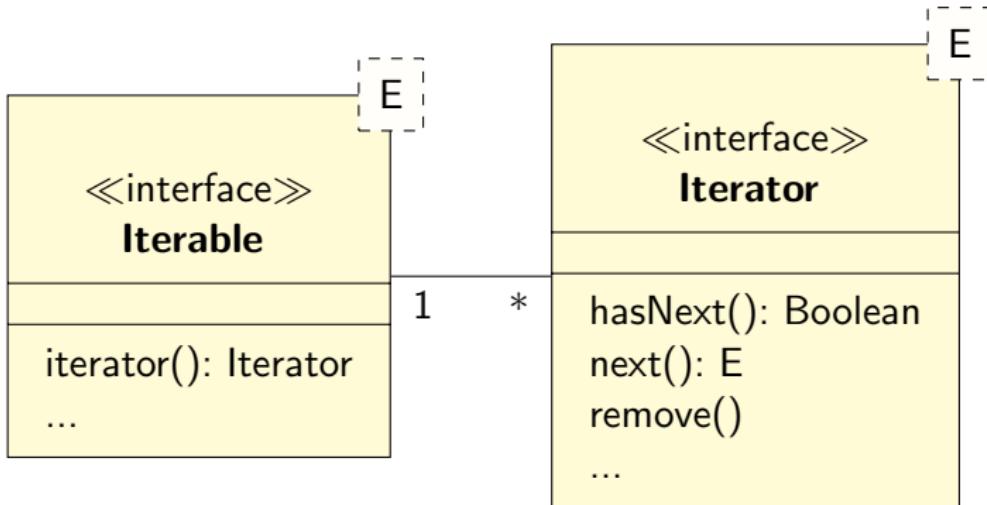


Abbildung 25: Iterable und Iterator.

Abschnittsübersicht

5 Java Collections-Framework

Übersicht über die Datenstrukturen

Iteratoren

Collections und Nebenläufigkeit

Thread-Safety

Die modernen Collection-Implementierungen (`ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`) sind nicht thread-safe, die alten Implementierungen (`Vector`, `Hashtable`) jedoch sind thread-safe, da alle Methoden synchronisiert sind. Man kann jedes Collection-Objekt mit `Collections.synchronized...()` in ein threadsicheres umwandeln.

`java.util.concurrent`

Im Package `java.util.concurrent` gibt es alternative Implementierungen für Collections-Klassen, die eine gute Performance bei Nebenläufigkeit bieten.

Kapitelübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Abschnittsübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Liste mit Objekten vom Typ Object

Vor Java 5 haben alle Containerklassen ausschließlich Objekte vom Typ Object verwaltet.

...

```
// Alle verwalteten Objekte sind vom Typ Object:  
List list = new ArrayList(); // Liste von Object-Objekten.  
list.add("Hallo");  
list.add(42);  
String text = (String) list.get(0);  
int zahl = (int) list.get(0); // Laufzeitfehler!
```

...

Liste mit Objekten vom Typ Object

Vor Java 5 haben alle Containerklassen ausschließlich Objekte vom Typ Object verwaltet.

...

```
// Alle verwalteten Objekte sind vom Typ Object:  
List list = new ArrayList(); // Liste von Object-Objekten.  
list.add("Hallo");  
list.add(42);  
String text = (String) list.get(0);  
int zahl = (int) list.get(0); // Laufzeitfehler!
```

...

Konsole

```
java.lang.ClassCastException: java.lang.String cannot be cast to ja...  
at de.hsmannheim.inf.pr2.generics.JavaParticularitiesDemo.rawTypes...  
at de.hsmannheim.inf.pr2.generics.JavaParticularitiesDemo.main(Jav...  
...
```

Motivation für Generics

Die Verwendung von `Object` als Datentyp in Containerklassen ist unbefriedigend, da

- die Klassen dann nicht typsicher sind – man könnte jedes beliebige Objekt hinzufügen.
- man beim Herausholen der Objekte ständig casten muss.

Gesucht ist also ein Mechanismus, der es erlaubt, diese Probleme zu umgehen.

Eigenschaften von Generics

Generics erlauben vom konkreten Typ zu abstrahieren, d. h.:

- Der Typ einer Variable, eines Parameters, eines Rückgabewertes etc. ist selbst ein Variable, eine **Typ-Variable (type variable)**.
- Klassen und Methoden haben zusätzliche **Typ-Parameter (type parameter)**, welche die Typ-Variablen setzen.
- Der Verwender kann Typ-Parameter setzen und damit die Klassen/Methoden parametrieren.
- Klassen mit Typ-Parametern sind **generische Typen (generic types)** bzw. generische Klassen (**generic classes**).
- Methoden mit Typ-Parametern sind **generische Methoden (generic methods)**.

Abschnittsübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Kellerspeicher (Stack)

Ein Stack ist eine Datenstruktur mit beschränktem Zugriff auf die gespeicherte Elemente. Sie arbeitet nach dem LIFO-Prinzip (Last-In-First-Out). Beim Auslesen eines Elements wird das jeweils zuletzt gespeicherte Element zuerst ausgelesen, danach das vorletzte etc. Der Kellerspeicher wird auch Stapel genannt (bzw. im englischen Stack). Elemente werden übereinander gestapelt und dann wieder in umgekehrter Reihenfolge vom Stapel genommen.

Schnittstelle eines Stacks

- `public void push(E e)` – legt ein neues Element `e` auf den Stapel.
- `public E pop()` – entfernt das oberste Element aus dem Stapel und gibt es zurück.
- `public E peek()` – liest das oberste Element vom Stapel, ohne es zu entfernen.

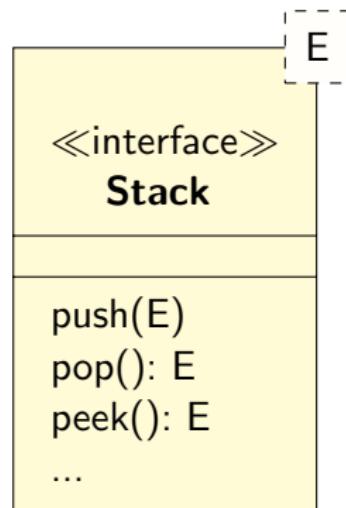
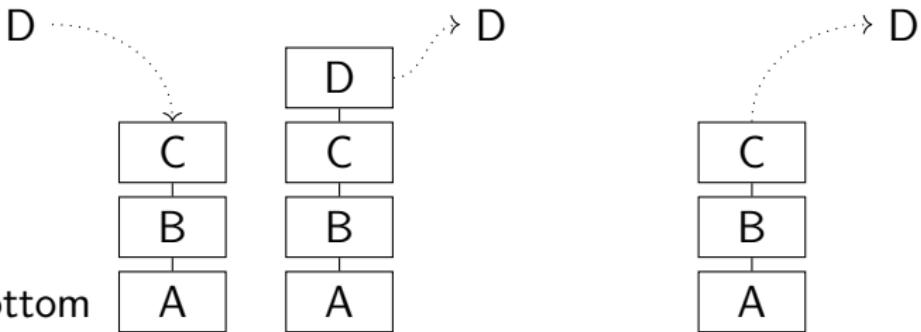


Abbildung 26: Interface Stack
(nicht Teil des Java Collections-Frameworks).



(a) Einfügen eines
Elements mittels
push().

(b) Lesen mittels
peek().

(c) Verlassen
eines Elements
mittels *pop()*.

Abbildung 27: Funktionsweise eines Stacks.

Stack-Implementierung mit internem Array

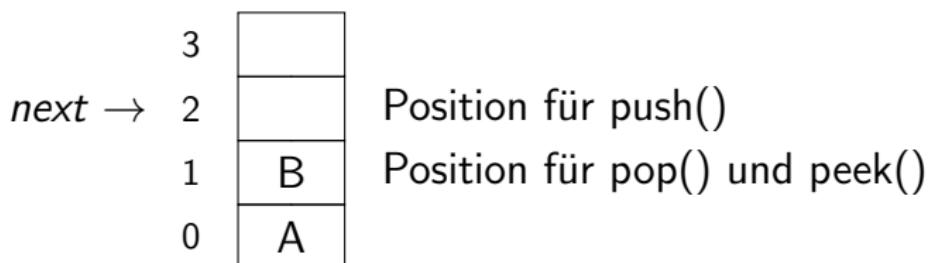


Abbildung 28: Funktionsweise eines Stack mittels internem Array. Die Indexvariable *next* zeigt auf den nächsten freien Speicherplatz. Zeigt *next* über das Array-Ende hinaus, gibt es eine Fehlermeldung.

ArrayStack (Teil 1)

```
...
public class ArrayStack<E> implements Stack<E> {

    protected E[] stack;
    private int next = 0; // Nächster freier Platz (= Anzahl).

    public ArrayStack(int capacity) {
        stack = (E[]) new Object[capacity];
    }
...
    public void push(E value) {
        if (next >= stack.length) {
            throw new ContainerException("Stack overflow");
        } else {
            stack[next++] = value;
        }
    }

    public E pop() {
        if (isEmpty()) {
            throw new ContainerException("Stack underflow");
        } else {
            return stack[--next];
        }
    }
}
```

ArrayStack (Teil 2)

```
        }
    }

public E peek() {
    if (isEmpty()) {
        throw new ContainerException("Stack is empty");
    } else {
        return stack[next - 1];
    }
}

...
}
```

Anwendung des Stacks

```
...  
ArrayStack<String> stack = new ArrayStack<>(10);  
stack.push("Hallo");  
stack.push("Welt!");  
// stack.push(42); // Compilerfehler!  
  
String s1 = stack.pop();  
String s2 = stack.pop();  
System.out.println(s1 + " " + s2);  
// int s3 = stack.pop(); // Compilerfehler!  
...
```

Anwendung des Stacks

```
...  
ArrayStack<String> stack = new ArrayStack<>(10);  
stack.push("Hallo");  
stack.push("Welt!");  
// stack.push(42); // Compilerfehler!  
  
String s1 = stack.pop();  
String s2 = stack.pop();  
System.out.println(s1 + " " + s2);  
// int s3 = stack.pop(); // Compilerfehler!  
...
```

Konsole

Welt! Hallo

Verbesserte Typ-Inferenz seit Java 7

Seit Java 7 ist die Typ-Inferenz für Generics deutlich verbessert worden:

- In den meisten Situationen muss man den Typ-Parameter nur noch bei der Deklaration setzen.
- Bei der Objekterzeugung kann die Angabe des Typ-Parameters durch den **Diamond <>** ersetzt werden.

```
List<String> list = new ArrayList<>();  
Map<Integer, String> map = new HashMap<>();  
Set<?> set = new HashSet<>();
```

Abschnittsübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Typlösung

Ko- und Invarianz

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Generics und Vererbung

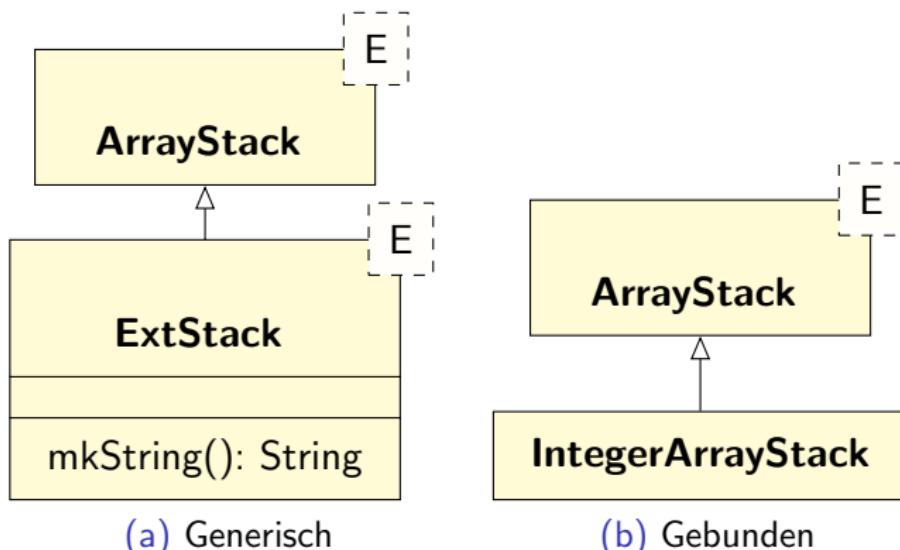


Abbildung 29: Möglichkeiten der Vererbung bei generischen Klassen: a) Der Subtyp ist selbst ein generischer Typ oder b) man setzt beim Erben den Typ-Parameter, sodass die Subklasse nicht mehr generisch ist.

ExtStack

```
...
public class ExtArrayList<E> extends ArrayList<E> {
...
/** 
 * @return Der Stack als String. Unterstes Element ist links.
 */
public String mkString() {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < size() - 1; i++) {
        sb.append(stack[i] + ", ");
    }
    return "[" + sb.toString() + stack[size() - 1] + "]";
}
...
}
```

Anwendung von ExtStack

```
...
ExtArrayList<String> stack = new ExtArrayList<>(10);
stack.push("Hallo");
stack.push("Welt!");
System.out.println(stack.mkString());
...

```

Konsole

[Hallo, Welt!]

IntegerArrayList

```
...
public class IntegerArrayList extends ArrayList<Integer> {

    public IntegerArrayList(int capacity) {
        super(capacity);
    }
}
```

Sowie die Anwendung:

```
...
    IntegerArrayList stack = new IntegerArrayList(10);
    stack.push(1);
    stack.push(2);
...
```

Generics als Compile-Zeit-Konstrukt

Java Generics existieren nur zur Compile-Zeit.

- **Typlöschung (type erasure)**: Typ-Informationen werden entfernt.
 - Typ-Parameter ist zur Laufzeit vom Typ Object.
 - Typ-Parameter kann nicht in statischen Variablen oder Methoden verwendet werden.
 - Typ-Parameter kann nicht verwendet werden, um Objekte zu erzeugen.
- Es gibt nur eine einzige Klasse pro generischem Typ, den **Raw type**.
- Generics verhalten sich damit grundlegend anders als C++-Templates.

Raw Types

```
...
ArrayStack<String> stringStack = new ArrayStack<>(4);
ArrayStack<Integer> intStack = new ArrayStack<>(4);
ArrayStack rawStack = new ArrayStack(4);

System.out.println(stringStack.getClass());
System.out.println(intStack.getClass());
System.out.println(rawStack.getClass());
...

```

Konsole

```
class de.hsmannheim.inf.pr2.ads.ArrayStack
class de.hsmannheim.inf.pr2.ads.ArrayStack
class de.hsmannheim.inf.pr2.ads.ArrayStack
```

Kovarianz und Invarianz

Sei A eine Oberklasse von B . Die Art der Vererbung einer zusammengesetzten Datenstruktur G mit Typ-Parameter wird **kovariant** (covariant) genannt, wenn die $G<A>$ ebenfalls als Oberklasse von G betrachtet wird. Andernfalls heißt die Art der Vererbung **invariant** (invariant).

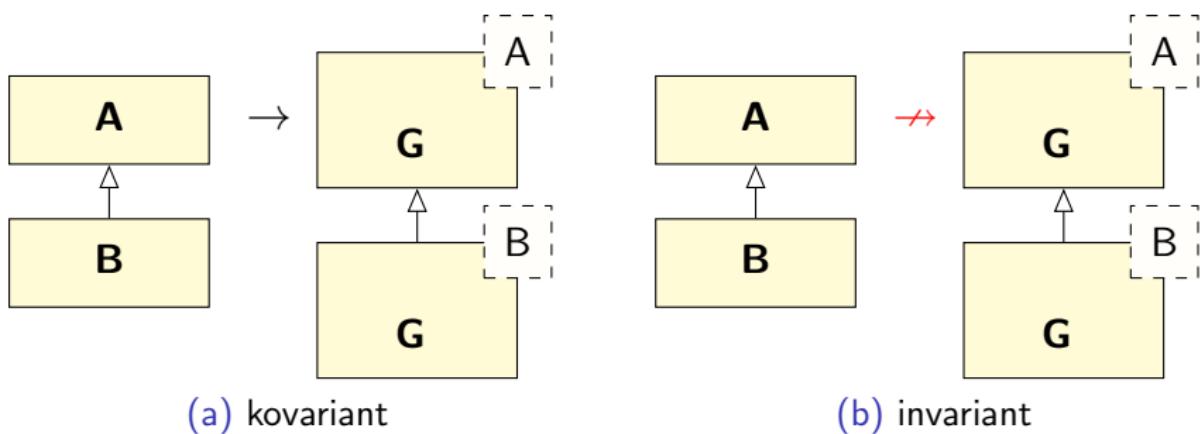


Abbildung 30: Ko- und invariante generische Klassen.

Arrays sind kovariant und reified

```
...
Object[] array = new String[10];
array[0] = "Hallo";
array[1] = Integer.valueOf(15); //
```

Laufzeitfehler!

```
...
```

Konsole

```
java.lang.ArrayStoreException: java.lang.Integer
at de.hsmannheim.inf.pr2.generics.JavaParticularitiesDemo.arrayRei...
at de.hsmannheim.inf.pr2.generics.JavaParticularitiesDemo.main(Jav...
...
...
```

Nachteile von Kovarianz

Diese Kovarianz von Arrays bedingen Laufzeitfehler. Bei Generics wollte man solche Fehler vermeiden und zur Compilezeit erkennen. Dann müssen Generics invariant sein (siehe Abb. 30).

Kovarianz- oder Auslese-Widerspruch

```
List<Number> numberList = new ArrayList<>();  
numberList.add(42); // Auto-Boxing: new Integer(42)  
numberList.add(3.14); // new Double(3.14)
```

```
List<Object> objList = numberList; // Wenn dies erlaubt wäre...  
objList.add("Hallo");  
Number n = numberList.get(2); // "Hallo" ist keine Number.
```

Generics sind invariant

```
ArrayList<Object> list =  
    new ArrayList<String>(); // Compilerfehler!  
list.add(new Integer(15));
```

Konsole

```
Error: java: incompatible types:  
java.util.ArrayList<java.lang.String> cannot be converted to  
java.util.ArrayList<java.lang.Object>
```

Generics vs. Arrays

Generics und Arrays unterscheiden sich grundsätzlich und harmonieren im Allgemeinen schlecht.

- Arrays sind **kovariant (covariant)** und führen daher die Prüfung zur Laufzeit durch (**reified**).
- Generics sind **invariant (invariant)** und verlagern alle Prüfungen auf die Compilezeit (**erasure**).

Es ist nicht möglich ein Array zu erstellen aus:

- einem generischen Typ (`List<E>[]`)
- einem parametrierten Typ (`List<String>[]`)
- einem Typ-Parameter (`E[]`)

Merke!

$G<A>$ bedeutet, dass die generische Klasse G Objekte der Klasse A und deren abgeleiteten Klassen verarbeiten kann.

Abschnittsübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Wildcards

Wildcards `<?>` zeigt an, dass jede beliebige Ausprägung eines generischen Typs möglich ist. `<?>` ist die Kurzform von `<? extends Object>`. Wildcards können nur bei der Deklaration von Parametern und Variablen auftauchen.

Bound Wildcards `<? extends T>` bzw. `<? super T>` stellen sicher, dass nicht jede beliebige Ausprägung des generischen Typs, sondern nur bestimmte möglich sind.

- `? super T`
der Typ-Parameter muss vom Typ T oder einer *Superklasse* (Oberklasse) von T sein.
- `? extends T`
der Typ-Parameter muss vom Typ T oder einer *Subklasse* (Unterklasse) von T sein.

Beispiel für ein Wildcard

```
...
public class StackPrinter {
    // <?> bedeutet: Referenz auf irgendeinen ArrayStack:
    public void printStack(ArrayStack<?> stack) {
        for (int i = stack.size(); i > 0; i--) {
            System.out.printf("%d %s%n", i, stack.pop());
        }
    }
}
...
...
ArrayStack<String> stack = new ArrayStack<>(10);
stack.push("!");
stack.push("World");
stack.push("Hello");

StackPrinter printer = new StackPrinter();
printer.printStack(stack);

ArrayStack<?> wildcard1 = new ArrayStack<String>(10);
// ArrayStack<?> wildcard2 =
//   new SimpleStack<?>(10); // Compilerfehler!
...

```

Beispiel für ein Bound Wildcard mit extends

```
...
public class StackSum {
    // Stack muss mindestens Elemente vom Typ Number enthalten:
    public double sum(ArrayStack<? extends Number> numbers) {
        double sum = 0;

        while (!numbers.isEmpty()) { // solange Elemente vorhanden...
            Number n = numbers.pop(); // numbers liefert mind. Number.
            sum += n.doubleValue();
        }
        return sum;
    }
}
...

...
ArrayStack<Integer> stack = new ArrayStack<>(10);
stack.push(1);
stack.push(2);
stack.push(3);

StackSum sum = new StackSum();
System.out.println(sum.sum(stack));
...
```

Beispiel für ein Bound Wildcard mit super

```
...
public class ExtArrayList<E> extends ArrayList<E> {
...
    public void popAll(Collection<? super E> collection) {
        while (!isEmpty()) {
            collection.add(pop());
        }
    }
}
...
...
ExtArrayList<Number> stack = new ExtArrayList<>(10);
stack.push(Integer.valueOf(42));
stack.push(4711); // Geht auch wg. Auto-Boxing.
stack.push(Double.valueOf(3.14));
stack.push(2.71);

// Object ist Oberklasse von Number:
List<Object> list = new ArrayList<>();
stack.popAll(list);
System.out.println(list);
...

```

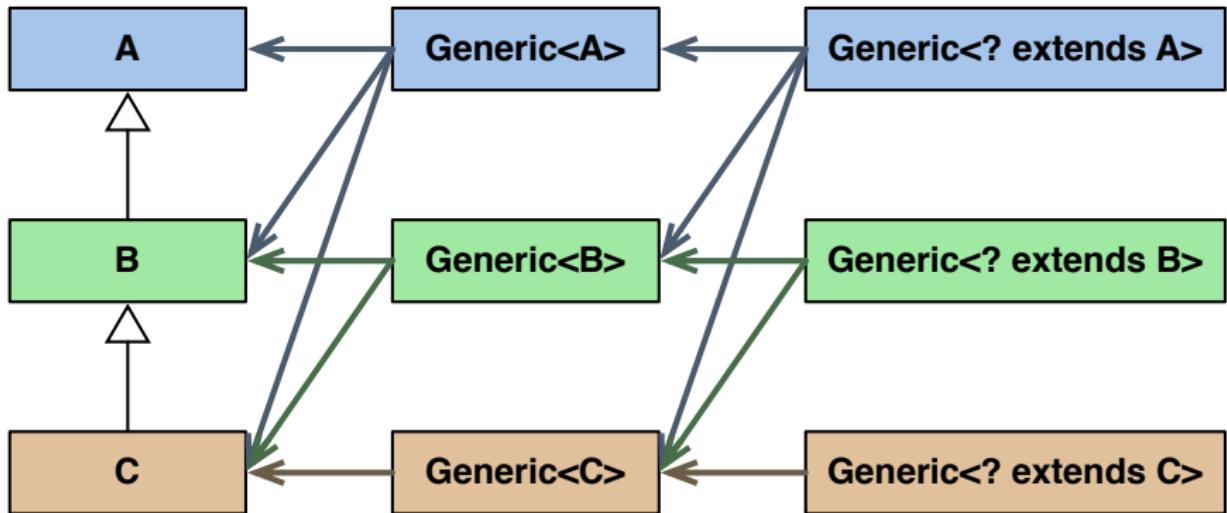


Abbildung 31: Wirkungsweise von `? extends` bei Generics. Diese Grafik zeigt, welche Klasse welche Objekte enthalten kann bzw. welche Referenzen auf welche Typen zeigen können. So kann eine Referenz vom Typ `Generic<? extends B>` auf die generische Klassen `Generic` und `Generic<C>` zeigen. `Generic` kann dann Objekte vom Typ `B` und `C` enthalten.

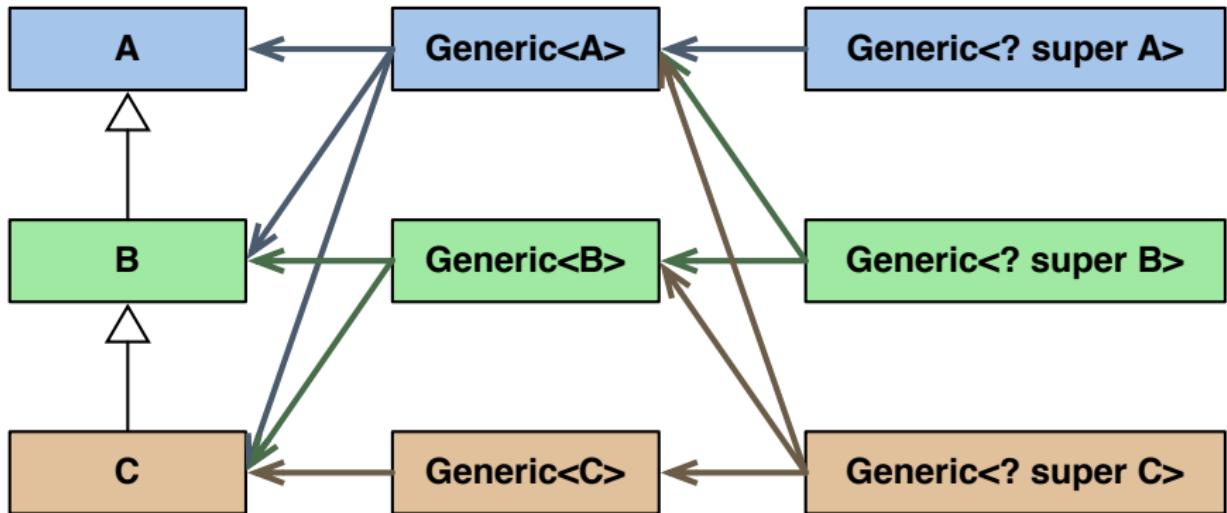


Abbildung 32: Wirkungsweise von `? super` bei Generics. Diese Grafik zeigt, welche Klasse welche Objekte enthalten kann bzw. welche Referenzen auf welche Typen zeigen können. So kann eine Referenz vom Typ `Generic<? super B>` auf die generische Klassen `Generic` und `Generic<A>` zeigen. `Generic` kann dann Objekte vom Typ B und C enthalten.

Abschnittsübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Generics bei statischen Methoden

Statische Methoden oder Attribute werden von allen Instanzen einer Klasse geteilt.

Statische Attribute

Da es von einer generischen Klasse Objekte mit verschiedenen Typ-Parametern gibt, kann es **keine** statischen Attribute geben.

Statische Methoden hingegen sind möglich: Hier muss der Typ-Parameter für jede statische Methode angegeben werden. Hier gibt wieder alle Kombinationsmöglichkeiten:

- einfacher Typ-Parameter bzw. Kurzschreibweise mit ?
- Typ-Parameter mit Wildcards

Einfacher Typ-Parameter

Der Typ-Parameter steht syntaktisch hinter static und vor dem Rückgabetyp.

```
...
    ArrayStack<Integer> as = new ArrayStack<>();
    as.push(1); // Füge ein paar Werte ein.
    as.push(2);
    as.push(3);
    readDataTypeParameter(as); // Rufe statische Methoden auf.
    readDataAnonymousWildcard(as);
    readDataNumber(as);
...
public static <E> void readDataTypeParameter(ArrayStack<E> stack) {
    while (!stack.isEmpty()) { // Gibt es noch Werte im Stack?
        E e = stack.pop(); // Lies Element aus.
        System.out.println("Element " + e);
    }
}
...
```

? als Typ-Parameter

Da die Eigenschaften von E unbekannt sind, kann es nur als eine Instanz der Klasse Objekt behandelt werden. Deshalb kann auch die Kurzschreibweise für ? extends Object benutzt werden:

```
...
public static void readDataAnonymousWildcard(ArrayStack<?> stack) {
    while (!stack.isEmpty()) { // Gibt es noch Werte im Stack?
        Object e = stack.pop(); // Lies Element aus.
        System.out.println("Element " + e);
    }
}
...
```

Der Source-Code wird dadurch etwas kürzer.

Gebundene Wildcards (extends oder super)

Schließlich lassen sich Typ-Parameter, die gebundene Wildcards nutzen, ebenfalls in statischen Methoden nutzen.

```
...
public static <E extends Number>
void readDataNumber(ArrayStack<E> stack) {
    double summe = 0; // Summe zunächst 0.
    while (!stack.isEmpty()) { // Gibt es noch Werte im Stack?
        E e = stack.pop(); // Lies Element aus.
        summe = summe + e.doubleValue(); // Bilde Summme.
    }
    System.out.println("Summe: " + summe);
}
...
```

Abschnittsübersicht

6 Generics

Motivation

Generische Klassen am Beispiel eines Stacks

Generics und Vererbung

Wildcards und gebundene Parameter

Generics bei statischen Methoden

Zusammenfassung

Kapitelübersicht

7 Collections und elementare Klassen und Interfaces aus java.lang

Verändern von Elementen in Containerklassen

Gleichheit und Reihenfolge von Objekten

Hashcode

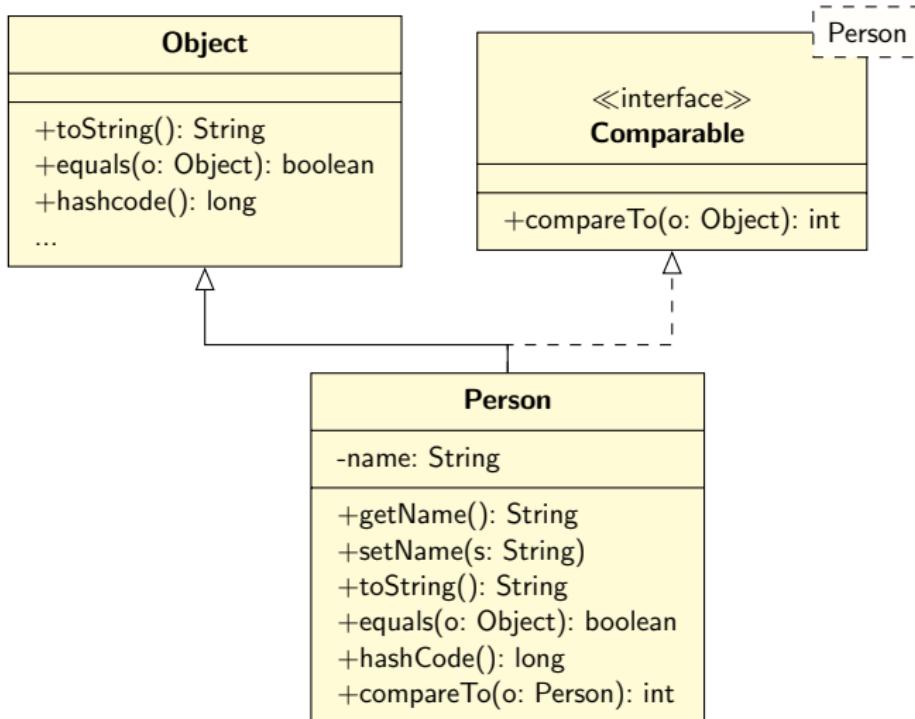


Abbildung 33: Wichtige Methoden der Klassen `Object` und des Interfaces `Comparable` am Beispiel der Klasse `Person`.

Abschnittsübersicht

7 Collections und elementare Klassen und Interfaces aus java.lang

Verändern von Elementen in Containerklassen

Gleichheit und Reihenfolge von Objekten

Hashcode

Person

```
...
public class Person implements Comparable<Person> {

    private String name;

    public Person(String name) {
        setName(name);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

...
}
```

Seiteneffekt durch Verändern der Objekte

```
...  
List<Person> list = new ArrayList<>();  
list.add(new Person("Homer"));  
Person march = new Person("March");  
list.add(march);  
System.out.println(list.toString());  
  
march.setName("Marge"); // Name war falsch!  
System.out.println(list.toString());  
...
```

Seiteneffekt durch Verändern der Objekte

```
...  
List<Person> list = new ArrayList<>();  
list.add(new Person("Homer"));  
Person march = new Person("March");  
list.add(march);  
System.out.println(list.toString());  
  
march.setName("Marge"); // Name war falsch!  
System.out.println(list.toString());  
...
```

Konsole

```
[Homer, March]  
[Homer, Marge]
```

Referenzen

Collections halten nur Referenzen auf die verwalteten Objekte. Wird ein Objekt nachträglich verändert, so ändert sich die Containerklasse indirekt mit.

Sortierreihenfolge falsch

```
...
Set<Person> set = new TreeSet<>();
set.add(new Person("Homer"));
Person march = new Person("Garge");
set.add(march);
System.out.println(set.toString()); // Sortiert!

march.setName("Marge"); // Name war falsch!
System.out.println(set.toString());
...
...
```

Sortierreihenfolge falsch

```
...
Set<Person> set = new TreeSet<>();
set.add(new Person("Homer"));
Person march = new Person("Garge");
set.add(march);
System.out.println(set.toString()); // Sortiert!

march.setName("Marge"); // Name war falsch!
System.out.println(set.toString());
...
```

Konsole

```
[Garge, Homer]
[Marge, Homer]
```

Eigenschaft der Collection verletzt

Das nachträgliche Verändern hat die Sortierreihenfolge verletzt.

Verändern von Elementen

Alle Elemente, die in den Containerklassen gespeichert werden, sind Objekte. Auf sie wird mittels einer Referenz verwiesen. Damit ist es u. a. möglich, ein gespeichertes Objekt zu verändern, wenn es bereits in die Containerklasse eingefügt wurde.

Vorsicht!

Wenn für die interne Organisation der Wert eines Elements benutzt wird, kann es zu Laufzeitfehlern kommen. Insbesondere bei Containerklassen, die Elemente sortieren (`TreeSet`, `TreeMap`) und solche, die auf Hashverfahren basieren (`HashSet`, `HashMap`), sollte das Verändern von Elementen vermieden werden.

Abschnittsübersicht

7 Collections und elementare Klassen und Interfaces aus java.lang

Verändern von Elementen in Containerklassen

Gleichheit und Reihenfolge von Objekten

Gleichheit von Objekten

Reihenfolge von Objekten

Hashcode

Gleichheit von Objekten

...

```
Set<Person> set = new HashSet<>();  
Person homer = new Person("Homer");  
set.add(homer);  
set.add(new Person("Marge"));  
  
Person homer2 = new Person("Homer");  
set.add(homer2); // Noch ein Homer wird hinzugefügt.  
  
System.out.println(set.toString());  
System.out.println(homer == homer2);  
System.out.println(homer.equals(homer2));  
...
```

Gleichheit von Objekten

```
...  
Set<Person> set = new HashSet<>();  
Person homer = new Person("Homer");  
set.add(homer);  
set.add(new Person("Marge"));  
  
Person homer2 = new Person("Homer");  
set.add(homer2); // Noch ein Homer wird hinzugefügt.  
  
System.out.println(set.toString());  
System.out.println(homer == homer2);  
System.out.println(homer.equals(homer2));  
...
```

Konsole

```
[Marge, Homer]  
false  
true
```

Gleichheit von Objekten mit equals()

Der Vergleichsoperator `==` vergleicht die Referenzen, d. h. ob eine Variable auf das gleiche Objekt auf dem Heap zeigt. In der `equals()`-Methode hingegen kann frei definiert werden, wann zwei Objekte als gleich angesehen werden.

Objekte werden stets mit equals() verglichen

Alle Objekte (Elemente) in den Containerklassen werden mit `equals()` auf Gleichheit überprüft. Ohne Überschreiben wird die Methode `equals()` in der Klasse `Object` genommen – dort wird auf Referenzgleichheit `==` verglichen. Sollte dies nicht gewünscht sein, muss `equals()` unbedingt überschrieben werden.

Beispiel für equals() in Person

```
...
public class Person implements Comparable<Person> {

    private String name;
    ...

    /**
     * Überprüft, ob ein anderes Objekt dieselbe Person ist.
     *
     * @param other Anderes Objekt.
     * @return Wahr, falls der Name der anderen Person gleich ist,
     * in allen anderen Fällen falsch.
     */
    @Override
    public boolean equals(Object other) {
        // Üblich: auf null und gleiche Klasse überprüfen:
        if (other != null && other instanceof Person) {
            Person person2 = (Person) other;
            return getName().equals(person2.getName());
        }
        return false;
    }
    ...
}
```

Reihenfolge von Objekten mit compareTo()

Für ein Objekt o_1 kann mit der Methode `compareTo()` des Interfaces `java.lang.Comparable` festgestellt werden, ob ein anderes Objekt o_2 kleiner, gleich oder größer ist. Für den Rückgabewert von `compareTo()` gilt:

- <0 falls o_1 kleiner als das andere Objekt o_2 ist ($o_1 < o_2$).
Üblich ist der Rückgabewert -1 .
- 0 falls beide Objekte gleich sind. Dann muss auch $o1.equals(o2)$ gelten.
- >0 falls Objekt o_1 größer als o_2 ist ($o_1 > o_2$). Üblich ist $+1$.

In `compareTo()` lassen sich also Kriterien definieren, wie eine Reihenfolge für beliebige Objekte bestimmt wird.

Beispiel für compareTo() in Person

```
...
public class Person implements Comparable<Person> {

    private String name;
    ...
    /**
     * Vergleicht diese Person mit einer anderen.
     *
     * @param other Andere Person (nicht Objekt!)
     * @return <0, falls der Name anderen Person lexikographisch
     * kleiner ist, 0 falls gleicher Name oder >0 sonst.
     */
    public int compareTo(Person other) {
        // Strings haben natürlich auch eine compareTo()-Methode:
        return getName().compareTo(other.getName());
    }
}
...
```

Comparator

Vermutlich alle Klassen im JDK, deren Objekte in eine Reihenfolge gebracht werden können, implementieren das Interface Comparable. Beispiele sind String, Date, Integer, Double uvm.

Comparator

Die Klasse `java.util.Comparator` ermittelt nach dem gleichen Prinzip wie Comparable die Reihenfolge zweier Objekte. Sie kann genutzt werden, wenn Objekte das Interface Comparable nicht implementieren.

Beispiel für Sortieralgorithmus und Comparator

```
...  
List<Person> list = new ArrayList<>();  
list.add(new Person("Homer"));  
list.add(new Person("Marge"));  
list.add(new Person("Bart"));  
list.add(new Person("Lisa"));  
list.add(new Person("Maggie"));  
  
Collections.sort(list); // Kriterium: Person.compareTo()  
System.out.println(list);  
  
Collections.sort(list, new Comparator<Person>() {  
    public int compare(Person p1, Person p2) {  
        return p1.getName().length() - p2.getName().length();  
    }  
});  
System.out.println(list);  
...
```

Konsole

```
[Bart, Homer, Lisa, Maggie, Marge]  
[Bart, Lisa, Homer, Marge, Maggie]
```

Beispiel für Sortieralgorithmus mit Lambda

```
...
List<Person> list = new ArrayList<>();
list.add(new Person("Homer"));
list.add(new Person("Marge"));
list.add(new Person("Bart"));
list.add(new Person("Lisa"));
list.add(new Person("Maggie"));

// Java 8 Lambda:
Collections.sort(list, (p1, p2) ->
    p1.getName().length() - p2.getName().length());
System.out.println(list);
...
```

Abschnittsübersicht

- 7 Collections und elementare Klassen und Interfaces aus `java.lang`
 - Verändern von Elementen in Containerklassen
 - Gleichheit und Reihenfolge von Objekten
 - Hashcode**

Hashcode

Die Klasse `Object` hat eine `hashCode()`-Methode. Die Bedeutung von Hashcodes werden wir im Abschnitt ?? genauer besprechen.

Elementare Algorithmen und Datenstrukturen

Kapitelübersicht

8 Einführung in Datenstrukturen

Knoten und Kanten

Elemente einer dynamischen Datenstruktur werden oft als **Knoten** (nodes oder vertices) und die Verbindungen zwischen ihnen als **Kanten** (edges) bezeichnet. Knoten werden in Java zur Laufzeit (dynamisch) mittels new-Operator erzeugt und dann verkettet. Solange genügend Hauptspeicher vorhanden ist, können neue Elemente erzeugt und in die Datenstruktur eingefügt werden.

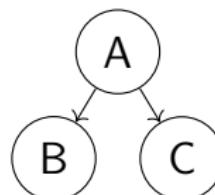
Wichtigsten Datenstrukturen (Teil 1)

Die wichtigsten dynamischen Datenstrukturen sind:

Listen: Knoten werden in einer Folge gespeichert. Der häufigste Vertreter ist die verkettete Liste. Varianten von Listen sind Stacks und Warteschlangen.

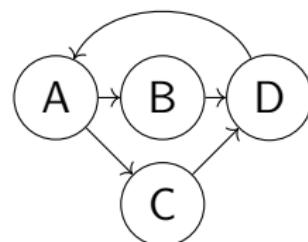


Bäume: Elemente von Bäumen haben i. A. mehr als einen Nachfolger aber stets genau einen Vorgänger. Wichtige Baumvarianten sind binäre Suchbäume, B-Bäume oder Heaps. Schreibweise:



Wichtigsten Datenstrukturen (Teil 2)

Graphen: Graphen sind die allgemeinste Datenstruktur, in der die Elemente beliebig viele Vorgänger oder Nachfolger haben können. Schreibweise:



Hashtabellen: Hashtabellen sind Arrays, in denen Elemente über eine sogenannte Hashfunktion sehr effizient verwaltet werden können.

Kapitelübersicht

9 Definition von dynamischen Datenstrukturen

Abstrakte Datentypen

Schnittstellen (Interfaces)

Prozedural vs. objekt-orientiert

Gebiet	Methode/Ansatz	Bemerkung
Mathematik	Abstrakter Datentyp (Algebra)	
Algorithmik	Datenstruktur / Schnittstellenbeschreibung	
Programmierung	Klasse (Implementierung)	Oft mehrere mögliche Implementie- rungen

Tabelle 2: Definitionen von dynamischen Datenstrukturen

Abschnittsübersicht

9 Definition von dynamischen Datenstrukturen

Abstrakte Datentypen

Schnittstellen (Interfaces)

Prozedural vs. objekt-orientiert

Abstrakte Datentypen

Ein abstrakter Datentyp (ADT) besteht aus einer Reihe von Mengen (Sorten), die für die Werte stehen, die Variablen annehmen können. Weiterhin werden Operationen definiert, die beschreiben, wie sich die Werte der Sorten verändern lassen. Ein solches Konstrukt wird auch Algebra genannt – sie „beschreibt“ bzw. charakterisiert den Datentyp. Die Bedeutung dieser Datenstruktur (die Semantik) ist nur indirekt über die Werte und Operationen gegeben. Da ADTs über Funktionen definiert sind, wird als Rückgabewert einer Funktion immer (!) eine neue Datenstruktur erzeugt, d. h. der Datentyp ist unveränderlich und passt somit sehr gut zur funktionalen Programmierung (siehe Abschnitt ??).

Abschnittsübersicht

9 Definition von dynamischen Datenstrukturen

Abstrakte Datentypen

Schnittstellen (Interfaces)

Prozedural vs. objekt-orientiert

Datentypdefinition über Schnittstellenbeschreibung

Eine Schnittstelle (engl. *interface*) enthält eine Reihe von Funktionen oder Prozeduren, die das Verhalten und die Eigenschaften einer Datenstruktur – oder ganz allgemein eines Programms – festlegen. Durch Aufrufe der Unterprogramme lässt sich eine neue Datenstruktur erzeugen. Man unterscheidet Schnittstellen nach der Art:

funktional: Es werden keine bestehenden Datenstrukturen verändert, sondern stets neue Datenstrukturen erzeugt, wenn etwas verändert werden soll. Jede Datenstruktur für sich ist unveränderlich (engl. *immutable*).

imperativ: Der Zustand (Belegung der Variablen usw.) kann durch Schnittstellenaufrufe verändert werden. Dies entspricht dem imperativem Ansatz.

Schreibweise von Schnittstellendefinitionen

Es wird im Folgenden eine sprachneutrale Schreibweise für Schnittstellen verwendet, die sich an die UML-Notation anlehnt. Beispielsweise wird die Java-Funktion

```
int findeMaximum(int [] f) {  
    ...  
}
```

geschrieben als

```
findeMaximum(f: IntArray): Int
```

Hauptunterschied ist, dass zuerst die Variable und dann der Datentyp kommt. Üblicherweise wird der Datentyp ebenfalls sprachneutral gehalten. Ebenso wird bei einer Schnittstelle niemals die Implementierung angegeben, sondern eben nur die Schnittstelle (die sogenannte Signatur) der Funktion. (→ Java-Interface)

Abschnittsübersicht

9 Definition von dynamischen Datenstrukturen

Abstrakte Datentypen

Schnittstellen (Interfaces)

Prozedural vs. objekt-orientiert

Prozedural vs. objekt-orientiert

Sowohl funktionale wie auch imperative Schnittstellen lassen sich über zwei verbreitete Ansätze beschrieben:

prozedural: In jeder Schnittstellenoperation wird die Datenstruktur als Parameter übergeben. Die prozedurale Operation zum Einfügen einer Zahl in die verkettete Liste sieht wie folgt aus:

- `add(z: Int, l: List)`
mit `z` als einzufügende Zahl und `l` als Liste.

objekt-orientiert: Hier gehören alle Funktionen bzw. Prozeduren (sog. Methoden) zu einer Klasse, und die Klasse entspricht der Datenstruktur:

- `l.add(z: Int)`
mit `l` als Objekt der Klasse List und `z` als Zahl.

Datenstrukturen werden imperativ/objekt-orientiert entwickelt.

Die Ansätze funktional/imperativ sowie prozedural/objekt-orientiert lassen sich prinzipiell frei kombinieren. Wir verwenden in der Vorlesung – sofern nicht anders angegeben – stets den Ansatz imperativ/objekt-orientiert.

Java Collections-Framework

Im Java Collections-Framework sind die Container stets imperativ/objekt-orientiert realisiert.

Eigenes Container-Framework

Im Folgenden werden Datenstrukturen und Algorithmen mit eigenen Container-Klassen beschrieben, die absichtlich nicht 100% kompatibel zum Java Collections-Framework sind.

Kapitelübersicht

10 Listen: Elemente mit bestimmter Reihenfolge

Verkettete Liste (Linked List)

Warteschlange (Queue)

Kellerspeicher (Stack)

Listen

Alle Listen-Datenstrukturen haben die Eigenschaft, dass die enthaltenen Elemente eine wohldefinierte Reihenfolge haben. Anders als im Array sind die Elemente nicht notwendigerweise über einen Index adressierbar.

Abschnittsübersicht

10 Listen: Elemente mit bestimmter Reihenfolge

Verkettete Liste (Linked List)

Aufbau einer verketteten Liste

ADT

Funktionale verkettete Liste

Objekt-orientierte verkettete Liste

Doppelt verkettete Liste

Aufwand

Warteschlange (Queue)

Kellerspeicher (Stack)

Aufbau einer verketteten Liste (linked list)

Jeder Knoten, außer dem letzten, hat genau einen Nachfolger. Das erste Element einer Liste wird Kopf (Head) und die übrigen Elemente Rest (Tail) genannt. Offenbar ist der Rest einer Liste wieder eine Liste bestehend aus Head and Tail. Eine einelementige Liste hat als Tail eine leere Liste.

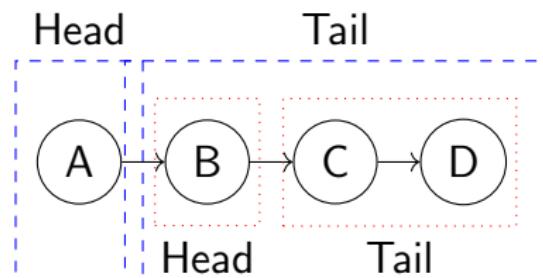


Abbildung 34: Rekursive Aufbau einer verketteten Liste.

Operationen für den ADT Liste

Aus dem selbstähnlichen Aufbau einer verketteten Liste lässt sich direkt ein abstrakter Datentyp ableiten.

Algebra:	List		
Sorten:	List, Elem, Boolean, Int		
Operationen:			
nil:		→ List	Erzeugt eine leere Liste.
addFirst:	Elem × List	→ List	Fügt einen Wert vorne an.
head:	List	→ Elem	Wert des ersten Knotens.
tail:	List	→ List	Gibt die Restliste zurück.
isEmpty:	List	→ Boolean	Überprüft, ob die Liste leer ist.
size:	List	→ Int	Ermittelt die Länge der Liste.
...

Signatur

Tabelle 3: Operationen des abstrakten Datentyps Liste

Axiome für den ADT Liste

Das Verhalten der Datenstruktur wird durch Axiome, also elementaren Eigenschaften, beschrieben:

- $\text{isEmpty}(\text{nil}()) = \text{true}$
- $\text{isEmpty}(\text{addFirst}(h, \text{nil}())) = \text{false}$
- $\text{head}(\text{addFirst}(h, t)) = h$
- $\text{tail}(\text{addFirst}(h, t)) = t$
- $\text{size}(\text{nil}()) = 0$
- $\text{size}(\text{addFirst}(h, t)) = 1 + \text{size}(t)$
- ...

Funktionale, objekt-orientierte verkettete Liste

Die rekursive und selbstähnliche Struktur einer verketteten Liste kann direkt in eine objekt-orientierte Datenstruktur überführt werden.

Funktionale, objekt-orientierte verkettete Liste

Bei einer funktionalen verketteten Liste wird die Liste ausschließlich als eine Folge von Listen-Knoten (ListNode) dargestellt.

Die in der Vorlesung besprochene Klasse ListNode implementiert einige wichtige Funktionen einer Liste. Eine leere Liste entspricht dem leeren Knoten, dargestellt durch ein null-Objekt. Die funktionale Liste wird objekt-orientiert realisiert, d. h. die Operationen sind nicht prozedurale Funktionen sondern Methoden.

Schnittstelle einer funktionalen, objekt-orientierten verketteten Liste

Methode	Funktion
<code>l.getHead(): E</code> (auch: <code>l.getFirst(): E</code>)	Liefert den Inhalt des ersten Knotens der Liste <code>l</code> .
<code>l.getTail(): List</code>	Gibt die Restliste ohne Kopf (head) zurück.
<code>l.addFirst(e: E): List</code>	Fügt ein neues Element <code>e</code> am Anfang der Liste ein und gibt die neue Liste zurück. Die alte Liste <code>l</code> wird nicht verändert.
<code>l.size(): Int</code>	Liefert die Länge der Liste, d. h. die Anzahl der Elemente.

Tabelle 4: Schnittstelle einer funktionalen verketteten Liste.

Implementierung einer funktionalen verketteten Liste

Die Listenknoten sind nur für die Verknüpfung zuständig, nicht aber für die Elemente (*value*) selbst, die in der Liste verwaltet werden.

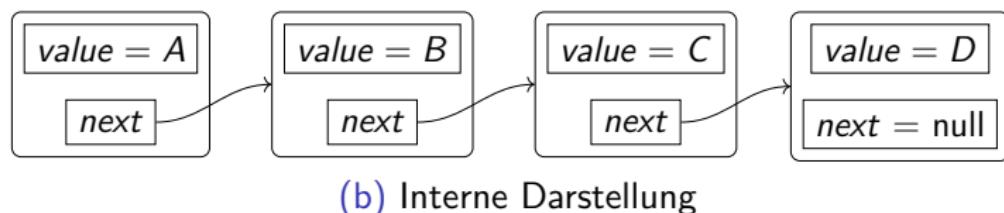
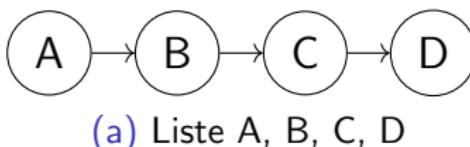


Abbildung 35: Aufbau einer verketteten Liste mittels Knoten, die aufeinander verweisen. Die Variable *next* verweist entweder auf den Nachfolgerknoten oder ist null, was bedeutet, dass es keinen Nachfolger gibt.

Die Klasse ListNode

```
...
public class ListNode<E> {
    /**
     * Wert, den dieser Knoten aufnimmt.
     */
    public E value;

    /**
     * Nachfolger dieses Knotens. null bedeutet: kein Nachfolger.
     */
    public ListNode<E> next = null;
}
...
```

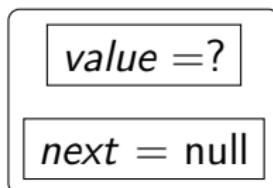


Abbildung 36: Listenknoten. Der Default-Wert von `value` hängt vom Datentyp ab.

Namenskonventionen

Die Klassennamen von Datenstrukturen folgen den Regeln:

- Zuerst wird der Datentyp genannt, der verwaltet wird (meist Integer-Zahlen oder Strings), sofern keine Generics zum Einsatz kommen.
- Am Ende steht der Typ der Datenstruktur oder der Hilfsklasse.
- Dazwischen stehen weitere Angaben wie z. B. die konkrete Implementierung, wenn es mehrere Möglichkeiten gibt.

Methoden der Klasse ListNode (Teil 1)

```
...
public class ListNode<E> {
...
/** 
 * Erzeugt einen neuen Knoten (neue Liste).
 *
 * @param value int-Wert dieses Knotens
 */
public ListNode(E value) {
    this.value = value;
}

/** 
 * @return Den Wert des ersten Knotens der Liste.
 */
public E getHead() {
    return value;
}

/** 
 * @return Die Restliste.
 */
public ListNode<E> getTail() {
    return next;
}
...
}
```

Methoden der Klasse ListNode (Teil 2)

```
...
public class ListNode<E> {
...
/** 
 * Erzeugt einen neuen Listenknoten und lässt ihn auf diese
 * Liste zeigen.
 *
 * @param value Wert des neuen Listenknotens.
 * @return Die neue Liste mit neuem Knoten am Anfang.
 */
public ListNode<E> addFirst(E value) {
    return new ListNode<>(value, this);
}

/** 
 * @return Die Anzahl der Elemente in der Liste.
 */
public int size() {
    if (getTail() == null) { // Kein Nachfolger-Knoten vorhanden?
        return 1; // Liste hat genau ein Element.
    } else {
        return 1 + getTail().size();
    }
}
...
}
```

Einige Operationen mit einer verketteten Liste

...

```
ListNode<Integer> list = null;    // Leere Liste.  
list = new ListNode<Integer>(1); // ergibt 1.  
list = list.addFirst(2);      // ergibt 2, 1.  
list = list.addFirst(3);      // ergibt 3, 2, 1.  
int value = list.getTail().getHead(); // ergibt 2.  
list.addFirst(4);            // ergibt 4, 3, 2, 1.  
int size = list.size();      // ergibt 3.
```

...

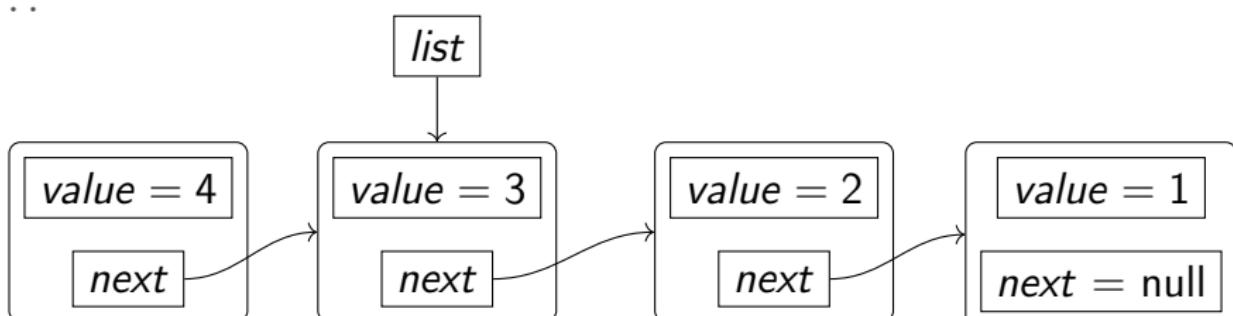


Abbildung 37: Beispiel einer durch verknüpfte ListNode-Objekte realisierten Liste.

Rein objekt-orientierte verkettete Liste

Um „Geisterlisten“ beim Einfügen eines Elements am Anfang ohne Zuweisung einer Variablen zu vermeiden, wird eine weitere Klasse eingeführt, die

- intern einen Verweis auf den aktuellen Kopf-Knoten führt und
- außerdem alle Operationen der imperativen Schnittstelle enthält.

Objekt-orientierte verkettete Liste

Bei einer rein objekt-orientierten verketteten Liste besteht die Liste aus einem Objekt der Klasse List und einer Folge von Listen-Knoten (ListNode). Viele Operationen auf diese Liste verändern die Elemente.

Objekt-orientierte verkettete Liste (Klasse List)

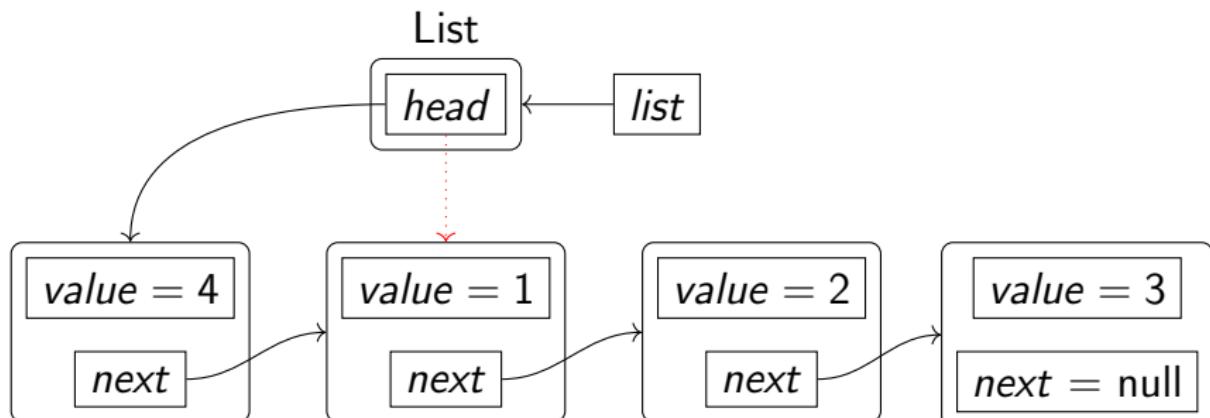


Abbildung 38: Eine rein objekt-orientierte Liste. Die Variable `head` der Klasse `List` zeigt immer auf den Anfang der Liste. Beim Einfügen des Elements 4 am Anfang wird die Referenz `head` von Knoten 1 (rot gestrichelt) auf Knoten 4 versetzt.

Die Klasse List incl. addFirst()-Methode

```
...
public class List<E> implements Container<E> {
    /**
     * Listenkopf.
     */
    protected ListNode<E> head = null;
...
    public void addFirst(E e) {
        if (isEmpty()) {
            head = new ListNode(e); // neuer Head-Knoten.
        } else {
            head = head.addFirst(e);
        }
    }
...
}
```

Das Interface Container

Abweichung vom Java Collections-Framework

Anders als im Java Collections-Framework führen wir das Interface Container für objekt-orientierte, veränderliche Container ein.

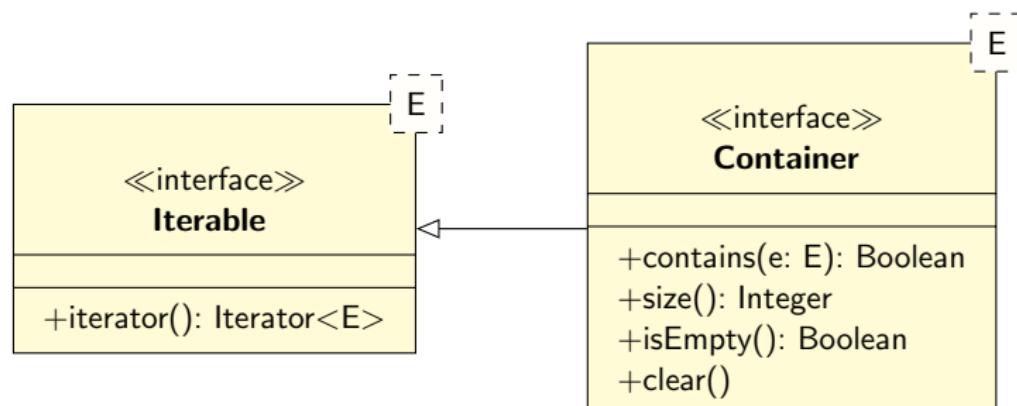


Abbildung 39: Das Interface Container für alle objekt-orientierten, veränderlichen Datenstrukturen.

Operationen einer veränderlichen, objekt-orientierten verketteten Liste (Teil 1)

Methode	Funktion
<code>I.addFirst(e: E)</code>	Fügt ein neues Element e am Anfang der Liste I ein.
<code>I.getFirst(): E</code>	Liefert den Wert den ersten Knotens, ohne den Knoten zu löschen.
<code>I.removeFirst(): E</code>	Löscht den ersten Knoten der Liste und gibt dessen Wert zurück.
<code>I.addLast(e: E)</code>	Hängt ein neues Element e am Ende der Liste I an.
<code>I.getLast(): E</code>	Liefert den Inhalt des letzten Elements der Liste I.
<code>I.removeLast(): E</code>	Löscht den letzten Knoten der Liste und gibt dessen Wert zurück.

Operationen einer veränderlichen, objekt-orientierten verketteten Liste (Teil 2)

Methode	Funktion
<code>l.remove(e: E)</code>	Entfernt alle Elemente mit dem Wert e aus der Liste, sofern vorhanden.
<code>l.setAt(i: int, e: E)</code>	Ersetzt das Element an Position i in der Liste, sofern es diese Position gibt. Andernfalls wird ein Fehler gemeldet.
<code>l.getAt(i: int): E</code>	Liefert das Element an Position i in der Liste, sofern es diese Position gibt. Andernfalls wird ein Fehler gemeldet.
<code>l.removeAt(i: int): E</code>	Entfernt den Knoten an Position i und gibt dessen Wert zurück, sofern es diese Position gibt. Andernfalls wird ein Fehler gemeldet.
<code>l.concat(m: List)</code>	Fügt m an das Ende von l durch Verlinkung.

Anwendungsbeispiele der List

```
...
List<Integer> list = new List<>(); // Erzeuge leere Liste.
list.addLast(1); // Füge Elemente ein. Ergibt 1.
list.addLast(2); // Ergibt 1, 2.
list.addLast(3); // Ergibt 1, 2, 3.
list.addFirst(4); // Ergibt 4, 1, 2, 3.

System.out.println("list = " + list); // möglich wg. toString().
System.out.println("Groesse = " + list.size());
System.out.println("leer? " + list.isEmpty());

System.out.println("Loesche alle Elemente aus list.");
list.clear();
...
```

Konsole

```
list = ( 4  1  2  3  )
list = ( 4  1  2  3  )
Groesse = 4
leer? false
Loesche alle Elemente aus list.
list = ( )
Groesse = 0
leer? true
```

Die addLast()-Methode

```
...
public class List<E> implements Container<E> {
...
/*
 * Fügt ein Element an das Ende der Liste an.
 *
 * @param e Element, das eingefügt werden soll.
 */
public void addLast(E e) {
    ListNode<E> node = new ListNode<>(e); // Erzeuge Knoten.
    // Das Ende der Liste suchen:
    ListNode<E> p = head; // Hilfsvariable.
    if (p == null) { // Leere Liste?
        head = node; // Head ist jetzt der neue Knoten.
    } else { // Liste enthält Elemente.
        while (p.getTail() != null) { // p am Ende?
            p = p.getTail(); // p wandert weiter.
        }
        p.next = node; // Füge am Ende an.
    }
}
...
}
```

Objekt-orientierte doppelt verkettete Liste

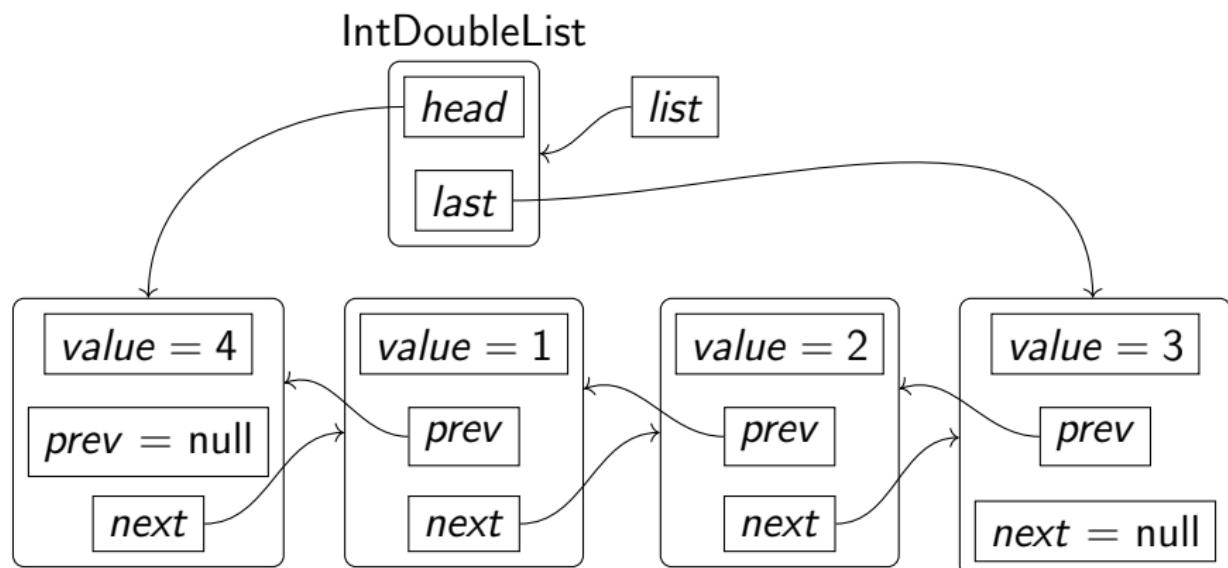


Abbildung 40: Eine objekt-orientierte doppelt verkettete Liste. Neben der Variablen `head` gibt es noch die Variable `last`, die auf das letzte Element der Liste zeigt.

Operation	einfach	doppelt
addFirst	$O(1)$	$O(1)$
getFirst	$O(1)$	$O(1)$
removeFirst	$O(1)$	$O(1)$
addLast	$O(n)$	$O(1)$
getLast	$O(n)$	$O(1)$
removeLast	$O(n)$	$O(1)$
...		
contains	$O(n)$	$O(n)$
remove	$O(n)$	$O(n)$
getAt	$O(n)$	$O(n)$
clear	$O(1)$	$O(1)$

Tabelle 5: Laufzeitanalyse für verkettete Listen. Es sind nicht alle Operationen gezeigt. Die Operationen *Last sind bei einer doppelt verketteten Liste schneller, da über den Zeiger *last* ein direkter Zugriff auf den letzten Knoten erfolgt.

Abschnittsübersicht

10 Listen: Elemente mit bestimmter Reihenfolge

Verkettete Liste (Linked List)

Warteschlange (Queue)

Warteschlange mit interner verketteten Liste

Warteschlange basierend auf einem zirkulären Array

Aufwand

Kellerspeicher (Stack)

Warteschlange (Queue)

Eine Warteschlange (engl. queue) ist eine Datenstruktur mit beschränktem Zugriff auf die gespeicherten Elemente, die nach dem FIFO-Prinzip (First-In-First-Out) funktioniert. Wie in einer Warteschlange im Supermarkt werden neue Elemente hinten an die Warteschlange eingefügt und vorne entnommen. Anwendungsbeispiele sind

- Prozessverwaltung in Betriebssystemen
- Speicherverwaltung von Warteschlangen bei Druckaufträgen
- Nachrichtenpuffer in der Kommunikationssoftware
- Reservierungen von Sitzplätzen uvm.

Schnittstelle einer Warteschlange

Methode	Funktion
$q.\text{enter}(e: E)$	Fügt ein neues Element e ans Ende der Queue q ein.
$q.\text{leave}(): E$	Entfernt das erste Element aus der Queue q und gibt es zurück.
$q.\text{front}(): E$	Liest das erste Element aus der Queue q , ohne es zu entfernen.

Tabelle 6: Schnittstelle einer (veränderlichen und objekt-orientierten) Warteschlange.

Außerdem können auf eine Warteschlange noch die generischen Operationen für alle Datenstrukturen angewandt werden.

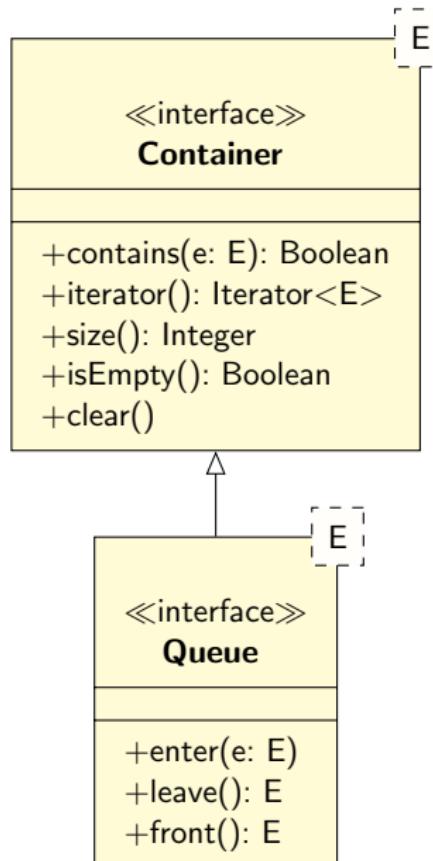
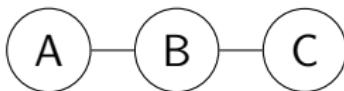


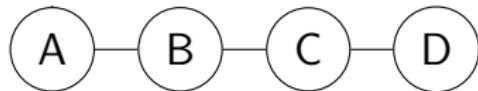
Abbildung 41: Das Interface Queue.

Front



- (a) Einfügen eines Elements mittels *enter()*.

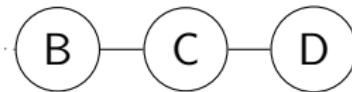
D



- (b) Lesen mittels *front()*.

A

Front



- (c) Verlassen eines Elements mittels *leave()*.

Abbildung 42: Funktionsweise einer Warteschlange.

Queue-Implementierung mit interner Liste

Der Anfang einer Warteschlange (front) entspricht dem Anfang einer Liste. Die Methode `leave()` kann durch `removeFirst()` und `enter()` mit `addLast()` realisiert werden. Die Klasse `ListQueue` kapselt die Operationen und delegiert an die intern verwendete Klasse `List` weiter. Dies ist ein übliches Entwurfsmuster in der objekt-orientierten Programmierung.

Wichtig!

Hier müssen die Instanzvariablen, die auf die interne Liste verweisen, geschützt werden, damit die interne Datenstruktur nicht von „außen“ verändert wird und so die Warteschlangen-Eigenschaft – d. h. das Einfügen am Ende und Verlassen am Anfang – umgangen wird.

Einige Methoden der Klasse ListQueue

```
...
public class ListQueue<E> implements Queue<E> {
    ...
    // Direkten Zugriff auf die Liste verbieten:
    private List<E> list = new List<>();
    ...

    public void enter(E e) {
        list.addLast(e); // Füge Element ans Ende der int. Liste an.
    }

    ...
    public E leave() {
        if (isEmpty()) {
            throw new ContainerException("Queue underflow");
        } else {
            return list.removeFirst(); // Front ist der Anfang der int. Liste.
        }
    }

    ...
    public boolean isEmpty() {
        return list.isEmpty(); // Delegiere dies weiter an die Liste.
    }

    ...
}
```

Anwendungsbeispiel der Klasse ListQueue

```
...
Queue<Integer> queue = new ListQueue<>();

System.out.println("Größe = " + queue.size()); // = 0.
queue.enter(1);
System.out.println("Größe = " + queue.size()); // = 1.
queue.enter(2);
System.out.println("Größe = " + queue.size()); // = 2.
queue.enter(3);
System.out.println("Größe = " + queue.size()); // = 3.

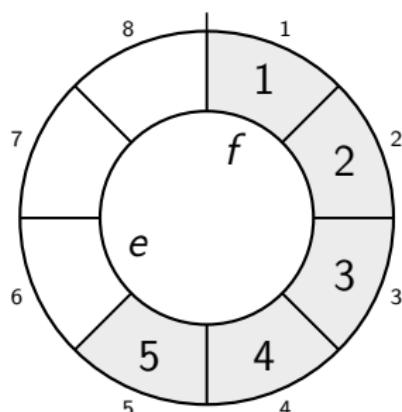
System.out.println("Nächstes: " + queue.leave()); // 1.
System.out.println("Größe = " + queue.size()); // = 2.

queue.enter(4); // Ergibt Elemente: 2, 3, 4.
System.out.println("Größe = " + queue.size()); // = 3.

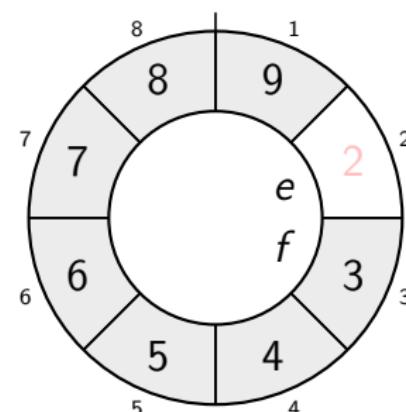
while (!queue.isEmpty()) {
    int i = queue.leave(); // ... in der Reihenfolge 2, 3, 4.
    System.out.println("Nächstes: " + i);
}
queue.leave(); // Fehlermeldung, weil die Warteschlange leer ist.
...
```

Warteschlange basierend auf einem zirkulären Array

f und e geben die Index-Position des ersten Elements (front) bzw. des nächsten freien Platzes zum Einfügen (enter) an. Ist das Ende des Arrays erreicht, findet ein Überlauf statt.



(a) 5 Elemente eingefügt.



(b) Überlauf

Abbildung 43: Warteschlange mit zirkulärem Array.

Kapazität des zirkulären Arrays

Zu Beginn zeigen f und e auf die gleiche Index-Position 1. $f = e$ bedeutet allgemein, dass das Array leer ist, unabhängig, welchen Wert f und e annehmen. Den Abstand d , den f und e haben dürfen, ist durch diese Vereinbarung $d = |e - f| \leq n - 1$. Insgesamt kann das Array also maximal $n - 1$ Elemente aufnehmen. Mittels

$$d = (e - f + n) \bmod n$$

lässt sich der Abstand d auch im Falle eines Überlaufs berechnen. Es ist es nicht nötig, Array-Elemente beim Entfernen zu löschen, da bereits über die Index-Positionen der Zustand klar ist.

Einige Methoden der Klasse ArrayQueue (Teil 1)

```
...
public class ArrayQueue<E> implements Queue<E> {

    // Die Elemente der Warteschlange:
    private Object[] queue = new Object[101];
    // Index des aktuellen Elements für front():
    private int f = 0;
    // Index des nächsten Elements für enter():
    private int e = 0;

    ...
    public void enter(E value) {
        // Hat die Queue bereits n-1-Elemente?
        if (size() == queue.length - 1) {
            throw new ContainerException("Queue overflow");
        } else {
            queue[e] = value;
            e = (e + 1) % queue.length;
        }
    }
    ...
}
```

Einige Methoden der Klasse ArrayQueue (Teil 2)

```
...
public class ArrayQueue<E> implements Queue<E> {
...
    public boolean isEmpty() {
        return f == e;
    }

    public void clear() {
        f = e = 0;
    }

    public int size() {
        return ((e - f) + queue.length) % queue.length;
    }

}
...
}
```

Operation	Aufwand
enter	$O(1)$
leave	$O(1)$
front	$O(1)$

Tabelle 7: Aufwand der Operationen einer Warteschlange. Der Aufwand gilt für alle Implementierungen und Kategorien (bester, schlechtester und durchschnittlicher Fall).

Abschnittsübersicht

10 Listen: Elemente mit bestimmter Reihenfolge

Verkettete Liste (Linked List)

Warteschlange (Queue)

Kellerspeicher (Stack)

Stack-Implementierung mit interner Liste

Stack-Implementierung mit internem Array

Anwendungsbeispiele für einen Stack

Aufwand

Kellerspeicher (Stack)

Ein Stack ist eine Datenstruktur mit beschränktem Zugriff auf die gespeicherte Elemente. Sie arbeitet nach dem LIFO-Prinzip (Last-In-First-Out). Beim Auslesen eines Elements wird das jeweils zuletzt gespeicherte Element zuerst ausgelesen, danach das vorletzte etc. Der Kellerspeicher wird auch Stapel genannt (bzw. im englischen Stack). Elemente werden übereinander gestapelt und dann wieder in umgekehrter Reihenfolge vom Stapel genommen.

Schnittstelle eines Stack

Methode	Funktion
<code>s.push(e: E)</code>	Legt ein neues Element e auf den Stapel s .
<code>s.pop(): E</code>	Entfernt das oberste Element aus dem Stapel s und gibt es zurück.
<code>s.peek(): E</code>	Liest das oberste Element vom Stapel, ohne es zu entfernen.

Tabelle 8: Schnittstelle eines (veränderlichen, objekt-orientierten) Stacks.

Außerdem können auf einen Stack noch die generischen Operationen für alle Datenstrukturen angewandt werden.

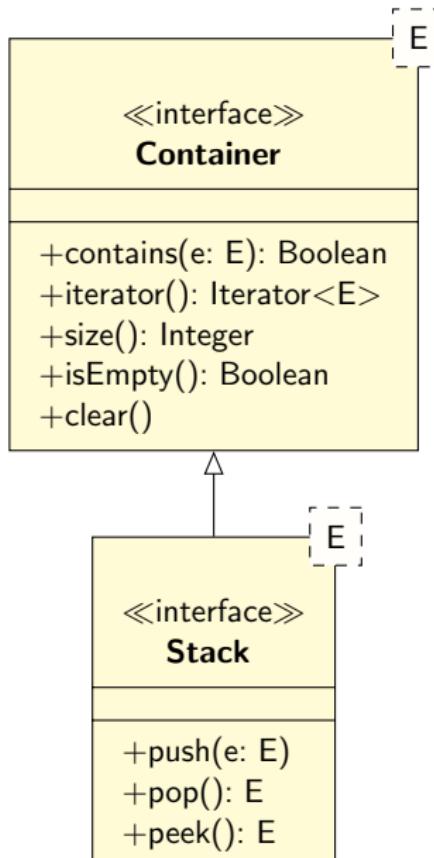
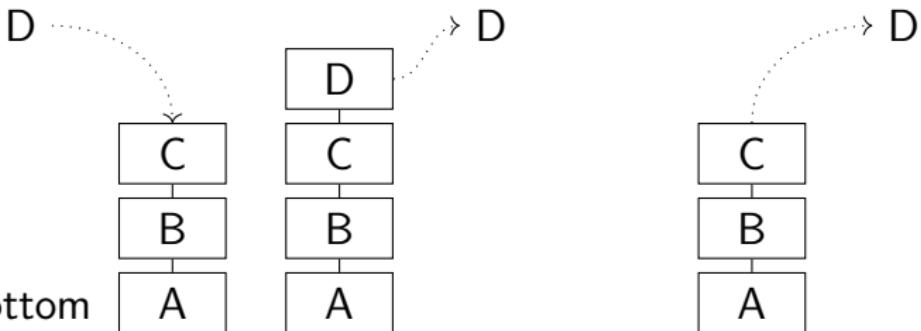


Abbildung 44: Das Interface Stack.



(a) Einfügen eines
Elements mittels
push().

(b) Lesen mittels
peek().

(c) Verlassen
eines Elements
mittels *pop()*.

Abbildung 45: Funktionsweise eines Stacks.

Stack-Implementierung mit interner Liste

ListStack realisiert einen Stack, indem intern eine List-Liste genutzt wird. push() entspricht dem Einfügen am Anfang der Liste mittels addFirst() und pop() dem Entfernen mittels removeFirst(). Auch hier wird das Entwurfsprinzip angewandt, das bereits bei ListQueue vorgestellt wurde.

Einige Methoden der Klasse ListStack

```
...
public class ListStack<E> implements Stack<E> {

    protected List<E> list = new List<>();

    ...
    public E pop() {
        if (isEmpty()) {
            throw new ContainerException("Stack Underflow");
        } else {
            return list.removeFirst();
        }
    }

    ...
    public void push(E value) {
        list.addFirst(value);
    }

    ...
    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int size() {
        return list.size();
    }
}
```

Anwendungsbeispiel der Klasse ListStack

```
...
Stack<Integer> stack = new ListStack<>();
stack.push(1); // 1| (<- unten)
stack.push(2); // 2, 1|
stack.push(3); // 3, 2, 1|  
  
while (!stack.isEmpty()) { // Ausgabe: 3, 2, 1
    int i = stack.pop();
    System.out.println(i);
}
...
...
```

Stack-Implementierung mit internem Array

Auch hier wird ein Entwurfsprinzip angewandt, das so ähnlich bei ArrayQueue vorgestellt wurde. Abb. 46 zeigt die Funktionsweise.

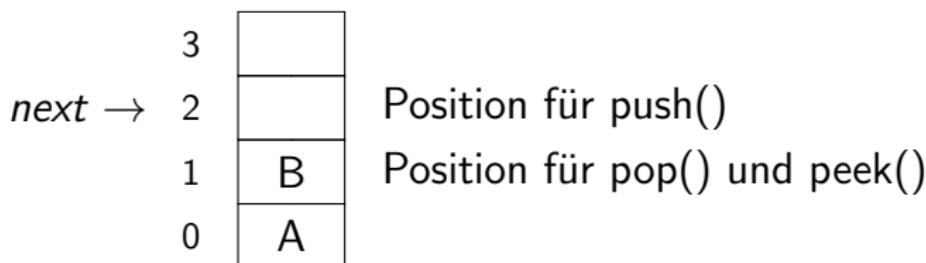


Abbildung 46: Funktionsweise eines Stack mittels internem Array. Die Indexvariable *next* zeigt auf den nächsten freien Speicherplatz. Zeigt *next* über das Array-Ende hinaus, gibt es eine Fehlermeldung.

Stack für Unterprogramme

Ein Stack kann genutzt werden, um die lokalen Variablen sowie die Parameter und Rückgabewerte einer Funktion zwischenzuspeichern. Mit jedem Unterprogrammaufruf werden folglich alle Parameter sowie die lokalen Variablen auf dem Stack gelegt. Gibt es zu viele Unterprogrammaufrufe, kann der Stack überlaufen (in Java: `StackOverflowError`).

Java-Beispiel zur Fakultätsberechnung (Teil 1)

```
...
Stack<Integer> stack = new ListStack<>();
public int n, f; // Globale Variablen.

public SubroutinesWithStack() {
    n = 4;
    stack.push(n);          // Schreibe neuen Parameter.
    facWithoutParameter();
    f = stack.pop();        // Entferne Ergebnis und
    stack.pop();            // Parameter.
    System.out.println("fak(" + n + ") = " + f);
}

public void facWithoutParameter() {
    n = stack.peek();       // Lies (1.) Parameter.
    if (n <= 1) {
        stack.push(1);     // Schreibe Ergebnis.
    } else {
```

Java-Beispiel zur Fakultätsberechnung (Teil 2)

```
    stack.push(n - 1);    // Schreibe neuen Parameter.  
    facWithoutParameter();  
    f = stack.pop();      // Entferne Ergebnis und  
    stack.pop();          // Parameter.  
    n = stack.peek();     // Parameter für diesen Aufruf (n ist  
global).  
    stack.push(n * f);    // Schreibe Ergebnis dieses Aufrufs.  
}  
}  
...
```

Protokoll (Teil 1)

Hinweis: Nachfolgend ist ein Stack in der Form (1 2 3) dargestellt.
Hierbei ist rechts unten, d. h. 3 ist das unten liegende Element.

```
main() ↓  
- stack = ( ) // Stack ist leer.  
- stack = ( 4 ) // Oberstes Element ist neuer Parameter n.  
- facWithoutParameter() ↓  
- - stack = ( 3 4 ) // Oberstes Element ist neuer Parameter.  
- - facWithoutParameter() ↓  
- - - stack = ( 2 3 4 ) // Oberstes Element ist neuer Parameter.  
- - - facWithoutParameter() ↓  
- - - - stack = ( 1 2 3 4 ) // Oberstes Element ist neuer Parameter.  
- - - - facWithoutParameter() ↓  
- - - - - stack = ( 1 1 2 3 4 ) // Oberstes Element ist Rueckgabewert.  
- - - - - facWithoutParameter() ↓  
- - - - - f = 1  
- - - - - n = 2  
- - - - - stack = ( 2 3 4 ) // Parameter und Ergebnis entfernt.  
- - - - - stack = ( 2 2 3 4 ) // Oberstes Element ist Rueckgabewert.  
- - - - - facWithoutParameter() ↓  
- - - - - f = 2  
- - - - - n = 3  
- - - - - stack = ( 3 4 ) // Parameter und Ergebnis entfernt.
```

Protokoll (Teil 2)

```
- - - stack = ( 6 3 4 ) // Oberstes Element ist Rueckgabewert.  
- - facWithoutParameter() ✓  
- - f = 6  
- - n = 4  
- - stack = ( 4 ) // Parameter und Ergebnis entfernt.  
- - stack = ( 24 4 ) // Oberstes Element ist Rueckgabewert.  
- facWithoutParameter() ✓  
- stack = ( 24 4 ) // Oberstes Element ist Rueckgabewert.  
- stack = ( ) // Stack ist wieder leer.  
main() ✓
```

Operation	Aufwand
push	$O(1)$
pop	$O(1)$
peek	$O(1)$

Tabelle 9: Aufwand der Operationen eines Stacks. Der Aufwand gilt für alle Implementierungen und Kategorien (bester, schlechtester und durchschnittlicher Fall).

Kapitelübersicht

11 Bäume

Eigenschaften von Bäumen

Binärbaum

Binäre Suchbäume

Ausgeglichene Suchbäume

Abschnittsübersicht

11 Bäume

Eigenschaften von Bäumen

Binärbaum

Binäre Suchbäume

Ausgeglichene Suchbäume

Baum (tree)

Ein Baum in der Informatik meint ein hierarchisches Strukturierungs- und Organisationsprinzip. Ein Baum ist eine Menge von Knoten und Kanten. Jeder Baum besitzt einen ausgezeichneten Knoten, die **Wurzel** bzw. Wurzelknoten (engl. root oder root node). Jeder Knoten, außer der Wurzel, ist durch genau eine Kante mit seinem **Vaterknoten** (parent node) verbunden. Ein Knoten, der einen Elternknoten hat, wird als dessen **Kindknoten** (child node) bezeichnet.

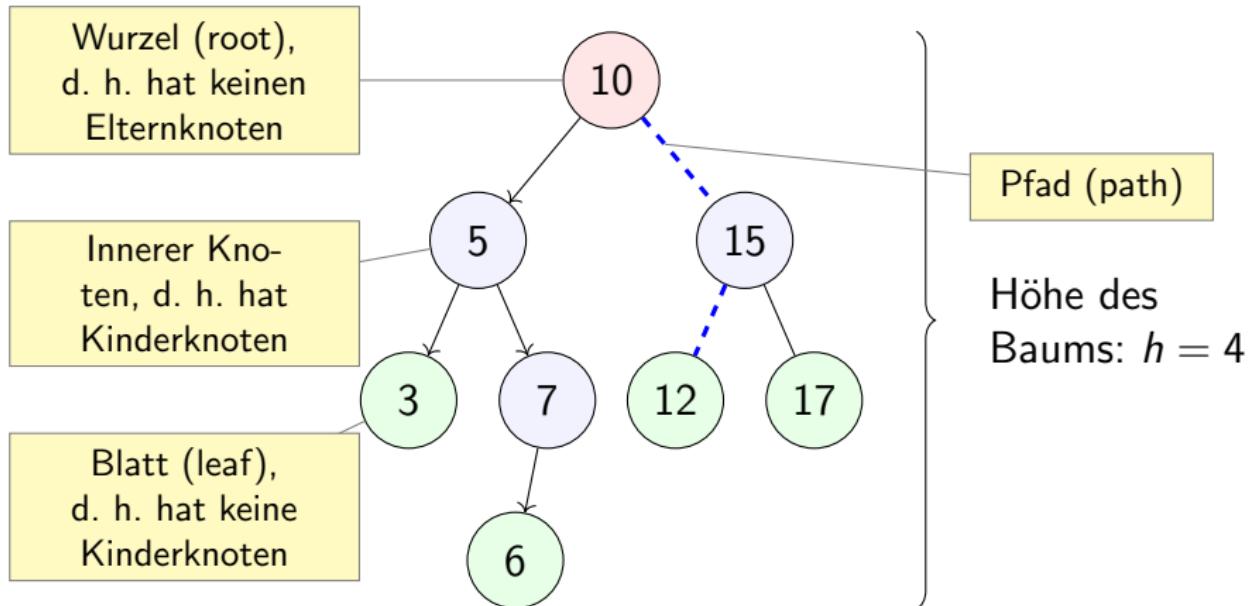


Abbildung 47: Wichtige Begriffe von Bäumen.

Höhe eines Baums

Das Niveau (auch Ebene oder engl. level) eines Knotens ist die Länge des Pfades von der Wurzel zu diesem Knoten. Die Höhe (height) eines Baumes entspricht dem größten Level eines Blattes + 1.

Verschiedene Arten von Bäumen

Ein Baum mit einer maximalen Anzahl von n Kindern ($n \geq 2$) heißt n -ärer Baum (n -ary tree). Ein wichtiger und häufig eingesetzter Spezialfall sind binäre Bäume, die maximal zwei Kinderknoten haben ($n = 2$). Wenn die Reihenfolge der Kinder eines Knotens von Bedeutung ist, spricht man von einem geordneten Baum (ordered tree).

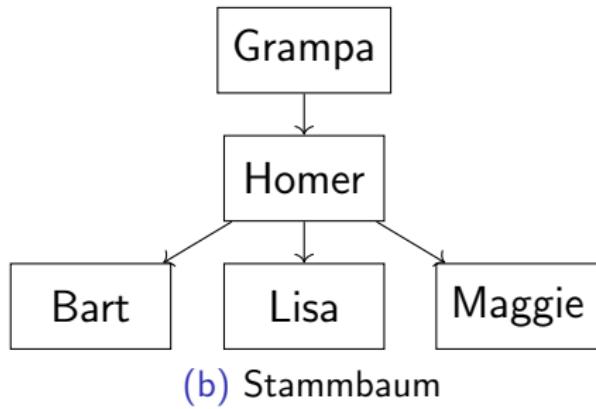
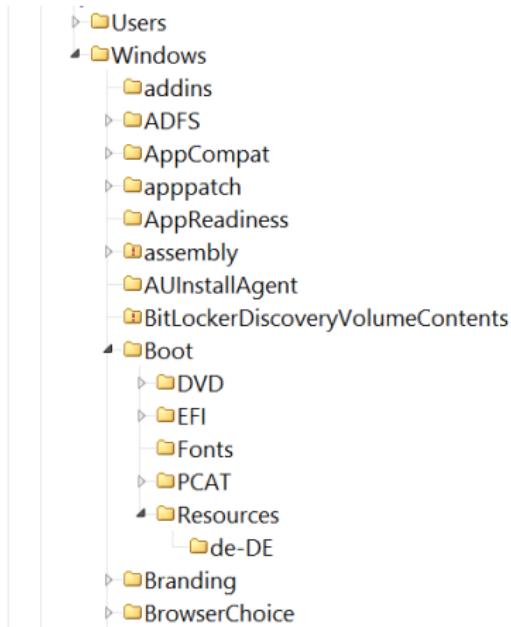


Abbildung 48: Beispiele für Bäume im täglichen Leben.

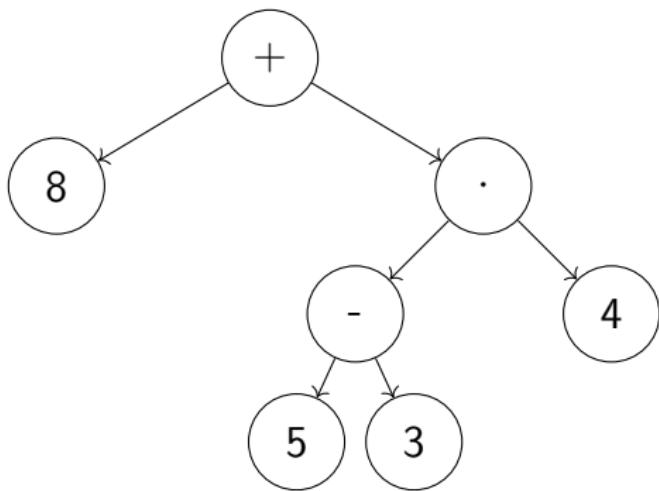


Abbildung 49: Darstellung des arithmetischen Ausdrucks $8 + (5 - 3) \cdot 4$ als geordneter Binärbaum.

Abschnittsübersicht

11 Bäume

Eigenschaften von Bäumen

Binärbaum

Funktionaler Binärbaum

Traversierung von Elementen in einem Binärbaum

Binäre Suchbäume

Ausgeglichene Suchbäume

Aufbau eines binären Baums (binary tree)

Jeder Knoten kann als Nachfolger (Kind) einen Teilbaum haben. Offenbar ist ein Teilbaum von der Struktur her wieder ein vollständiger Baum. Ein leerer Baum wird durch einen Null-Knoten repräsentiert.

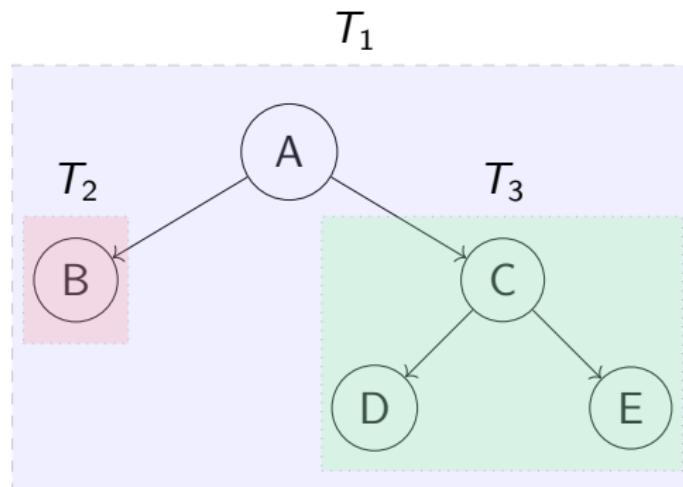


Abbildung 50: Selbstähnlicher Aufbau eines Binärbaums.

Schnittstelle eines funktionalen, objekt-orientierten Binärbaums (Teil 1)

Methode	Funktion
<code>t.value()</code>	Liefert den Wert des Wurzknotens.
<code>t.left(): TreeNode</code>	Liefert den linken Teilbaum.
<code>t.right(): TreeNode</code>	Liefert den rechten Teilbaum.
<code>t.size(): Int</code>	Liefert die Anzahl der Elemente des Baums.
<code>t.height(): Int</code>	Ergibt die Höhe des Baums.
<code>tree(e: E, l: TreeNode, r: TreeNode): TreeNode</code>	Konstruiert einen binären Baum aus einem linken und rechten Teilbaum und einem Element e als gemeinsame Wurzel.

Tabelle 10: Schnittstelle eines (funktionalen und objekt-orientierten) Binärbaums.

Implementierung eines funktionalen Binärbaums

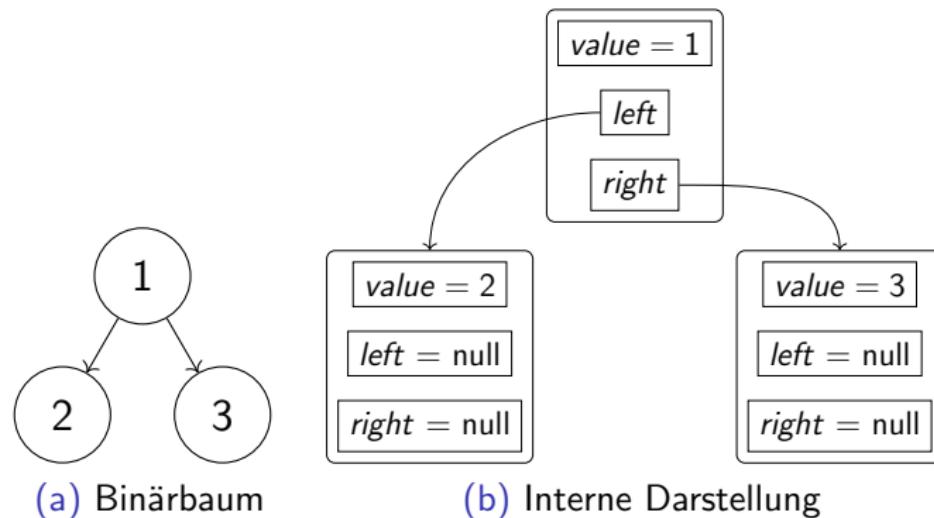


Abbildung 51: Aufbau eines Binärbaums mittels Knoten, die aufeinander verweisen. Die Variablen *left* und *right* verweisen entweder auf einen Teilbaum oder sind null, was bedeutet, dass es keinen Teilbaum gibt.

Die Klasse TreeNode

```
...
public class TreeNode<E> {

    protected E value = null;           // Wert des Knotens.
    protected TreeNode<E> left = null; // Linker Teilbaum.
    protected TreeNode<E> right = null; // Rechter Teilbaum.

}
...
}
```

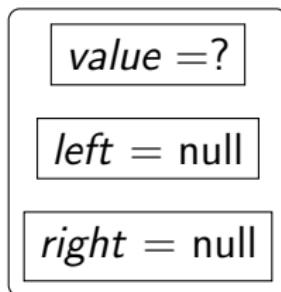


Abbildung 52: Binärbaumknoten mit Default-Werten.

```
...
public class TreeNode<E> {
...
    /**
     * Erzeuge einen neuen Binärbaum(-Knoten), entspricht tree().
     * @param value Der Wert des Knotens.
     * @param left Der linke Teilbaum.
     * @param right Der rechte Teilbaum.
     */
    public TreeNode(E value, TreeNode<E> left, TreeNode<E> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public E getValue() {
        return value;
    }

    public TreeNode<E> getLeft() {
        return left;
    }

    public TreeNode<E> getRight() {
        return right;
    }
...
}
```

Anwendung eines funktionalen Binärbaums

...

```
TreeNode<Integer> t2 = new TreeNode<>(2);
TreeNode<Integer> t3 = new TreeNode<>(3);
TreeNode<Integer> t1 = new TreeNode<>(1, t2, t3);
int value = t1.getLeft().getValue(); // = 2.
System.out.println("Value = " + value);
```

...

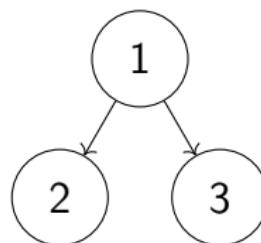


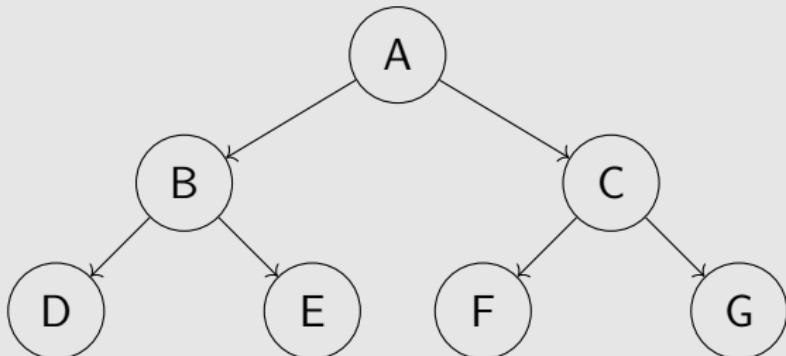
Abbildung 53: Binärbaum t1 aus Listing.

Algorithmen zur Traversierung von Binärbäumen

Das Abarbeiten (Durchlaufen, Traversierung) von Knoten in einer bestimmten Reihenfolge ist ein grundlegendes Konzept für Binärbäume.

- Inorder** Zuerst wird der linke Teilbaum besucht, dann der Knoten selbst und anschließend der rechte Teilbaum.
- Preorder** Zuerst wird der Knoten selbst besucht, dann der linke Teilbaum und anschließend der rechte Teilbaum.
- Postorder** Zuerst wird der linke Teilbaum besucht, dann der rechte Teilbaum und anschließend der Knoten selbst.
- Levelorder** Zuerst werden alle Knoten auf demselben Niveau von links nach rechts besucht, dann wird auf das nächste Niveau gewechselt.

Beispiel



(a) Baum

Durchlauf	Reihenfolge
Preorder	A, B, D, E, C, F, G
Inorder	D, B, E, A, F, C, G
Postorder	D, E, B, F, G, C, A
Levelorder	A, B, C, D, E, F, G

(b) Traversierung

Abbildung 54: Beispiele für Traversierungen von Binäräbäumen.

Algorithmus 1 Inorder-Durchlauf

```
proc Inorder(node)                                // Inorder-Durchlauf ab Knoten node.
    if node ≠ null then                          // Teilbaum nicht leer?
        Inorder(Linker Teilbaum von node);
        Verarbeite Wert des Knotens node;      // Verarbeitung in der Mitte.
        Inorder(Rechter Teilbaum von node);
    fi
corp
```

Zeichen in Inorder-Reihenfolge ausgeben

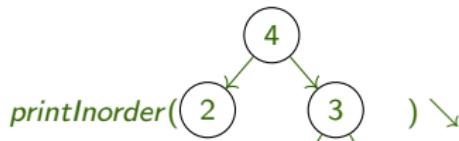
In `printInorder()` wird der aktuelle Knoten (`this`) an die zweite Variante der Methode übergeben, die einen Knoten als Parameter hat.

```
...
public class TreeNode<E> {
...
    public void printInorder() {
        printInorder(this);
        System.out.println(); // Am Ende noch eine neue Zeile.
    }
...
    private void printInorder(TreeNode<E> node) {
        if (node != null) { // Ist der Baum nicht leer?
            // Dann kann weiter gemacht werden.
            printInorder(node.getLeft());
            System.out.println(node.toString());
            printInorder(node.getRight());
        }
    }
...
}
```

Hinweis

Im Abschnitt ?? wird beschrieben, wie sich Algorithmen protokollieren lassen.

Protokoll für Inorder (Teil 1)



// Linker Teilbaum von 4:

- *printInorder(2)* ↘

// Linker Teilbaum von 2:

- - *printInorder(null)* ↘

- - *printInorder(null)* ↘

- - Ausgabe "2"

// Rechter Teilbaum von 2:

- - *printInorder(null)* ↘

- - *printInorder(null)* ↘

- *printInorder(2)* ↗

- Ausgabe "4"

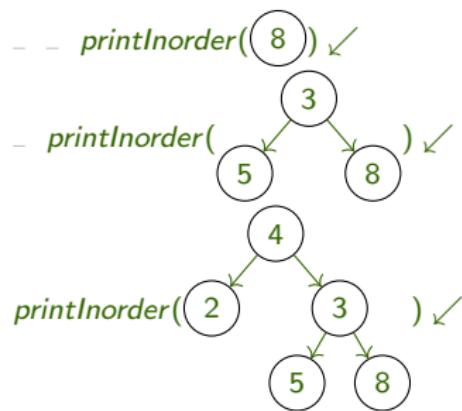
// Rechter Teilbaum von 4:

- *printInorder(3)* ↗

Protokoll für Inorder (Teil 2)

```
// Linker Teilbaum von 3:  
-- printInorder(5) ↓  
// Linker Teilbaum von 5:  
--- printInorder(null) ↓  
--- printInorder(null) ↗  
--- Ausgabe "5"  
// Rechter Teilbaum von 5:  
--- printInorder(null) ↓  
--- printInorder(null) ↗  
--- printInorder(5) ↗  
-- Ausgabe "3"  
// Rechter Teilbaum von 3:  
-- printInorder(8) ↓  
// Linker Teilbaum von 8:  
--- printInorder(null) ↓  
--- printInorder(null) ↗  
--- Ausgabe "8"  
// Rechter Teilbaum von 8:  
--- printInorder(null) ↓  
--- printInorder(null) ↗
```

Protokoll für Inorder (Teil 3)



Algorithmus 2 Preorder-Durchlauf

```
proc Preorder(node)                                // Preorder-Durchlauf ab Knoten node.  
    if node ≠ null then                            // Teilbaum nicht leer?  
        Verarbeite Wert des Knotens node;          // Verarbeitung zu Beginn.  
        Preorder(Linker Teilbaum von node);  
        Preorder(Rechter Teilbaum von node);  
    fi  
corp
```

Algorithmus 3 Postorder-Durchlauf

```
proc Postorder(node)                                // Postorder-Durchlauf ab Knoten node.  
    if node ≠ null then                            // Teilbaum nicht leer?  
        Postorder(Linker Teilbaum von node);  
        Postorder(Rechter Teilbaum von node);  
        Verarbeite Wert des Knotens node;          // Verarbeitung am Ende.  
    fi  
corp
```

Levelorder

Algorithmus 4 Levelorder-Durchlauf

```
proc Levelorder(node)                                // Levelorder-Durchlauf ab Knoten node.
    if node ≠ null then                            // Teilbaum nicht leer?
        Sei q eine leere Warteschlange;
        Füge node in q ein;                      // q.enter(node)
        while q ist nicht leer do
            Hole in n das nächste Element aus q;      // n := q.leave()
            Verarbeite Wert des Knotens n;
            if n hat linken Nachfolger then
                Füge linken Nachfolger in q ein;          // q.enter(n.getLeft())
                fi
            if n hat rechten Nachfolger then
                Füge rechten Nachfolger in q ein;         // q.enter(n.getRight())
                fi
        od
    fi
corp
```

Schnittstelle eines funktionalen, objekt-orientierten Binärbaums (Teil 2)

Methode	Funktion
<code>t.preorder()</code>	Durchläuft alle Elemente in Preorder-Reihenfolge.
<code>t.inorder()</code>	Durchläuft alle Elemente in Inorder-Reihenfolge.
<code>t.postorder()</code>	Durchläuft alle Elemente in Postorder-Reihenfolge.
<code>t.levelorder()</code>	Durchläuft alle Elemente in Levelorder-Reihenfolge.

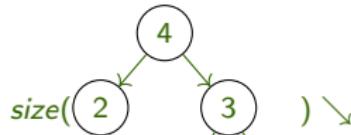
Tabelle 11: Weitere Operationen eines (funktionalen und objekt-orientierten) Binärbaums.

Anzahl Elemente in einem Binärbaum

Nun soll die Methode `size()` eines Baums mit Hilfe einer Postorder-ähnlichen Traversierung implementiert werden.

```
...
public class TreeNode<E> {
...
    public int size() {
        return size(this);
    }
...
    private int size(TreeNode<E> node) {
        if (node == null) { // Leerer Baum?
            return 0; // Anzahl Elemente ist 0.
        } else { // Echter Teilbaum.
            // Größe ist dieser Knoten plus solche in den Teilbäumen:
            int sizeLeft = size(node.getLeft());
            int sizeRight = size(node.getRight());
            return 1 + sizeLeft + sizeRight;
        }
    }
...
}
```

Protokoll für size() (Teil 1)



```
size((2)) →  
-- size(null) →  
-- size(null) = 0 ✓  
-- size(null) →  
-- size(null) = 0 ✓  
size((2)) = 1 ✓
```

```
size((3)) →  
-- size(5) →  
-- -- size(null) →  
-- -- size(null) = 0 ✓  
-- -- size(null) →  
-- -- size(null) = 0 ✓
```

Protokoll für size() (Teil 2)

```
- - size(5) = 1 ✓  
- - size(8) ↴  
- - - size(null) ↴  
- - - size(null) = 0 ✓  
- - - size(null) ↴  
- - - size(null) = 0 ✓  
- - size(8) = 1 ✓  
- size(3  
  |  
  5  8 ) = 3 ✓  
- size(4  
  |  
  2  3 ) = 5 ✓  
  |  
  5  8
```

Abschnittsübersicht

11 Bäume

Eigenschaften von Bäumen

Binärbaum

Binäre Suchbäume

Eigenschaften eines binären Suchbaums

Veränderlicher binärer Suchbaum

Aufwand

Ausgeglichene Suchbäume

Suchbaum

Bisher wurden Bäume in erster Linie für die hierarchische Repräsentation und Organisation von Daten genutzt. Das wichtigste Einsatzgebiet ist jedoch die effektive **Suche von Elementen**. Wir werden in diesem Abschnitt speziell binäre Suchbäume betrachten.

Suchbäume realisieren Mengen

Suchbäume sollen ihre Elemente wie in einer Menge im mathematischen Sinne verwalten. Duplikate sind nicht erlaubt und die Reihenfolge der Elemente spielt keine Rolle.

Operationen eines veränderlichen, objekt-orientierten binären Suchbaums

Methode	Funktion
<code>s.add(e: E): Boolean</code>	Fügt ein neues Element e in die Menge (in den Suchbaum) s ein und gibt wahr zurück, wenn dies möglich wahr oder falsch, falls ein solches Element bereits enthalten ist.
<code>s.contains(e: E): Boolean</code>	Überprüft, ob ein Element in der Menge (im Suchbaum) vorhanden ist.
<code>s.remove(e: E): Boolean</code>	Entfernt ein Element e aus der Menge (dem Suchbaum) und gibt wahr zurück, falls ein solches Element enthalten war, andernfalls falsch. Hinweis: Es kann maximal ein solches Element geben.

Tabelle 12: Schnittstelle eines binären Suchbaums. Die Operationen entsprechen dem Verwalten von Elementen in einer Menge.

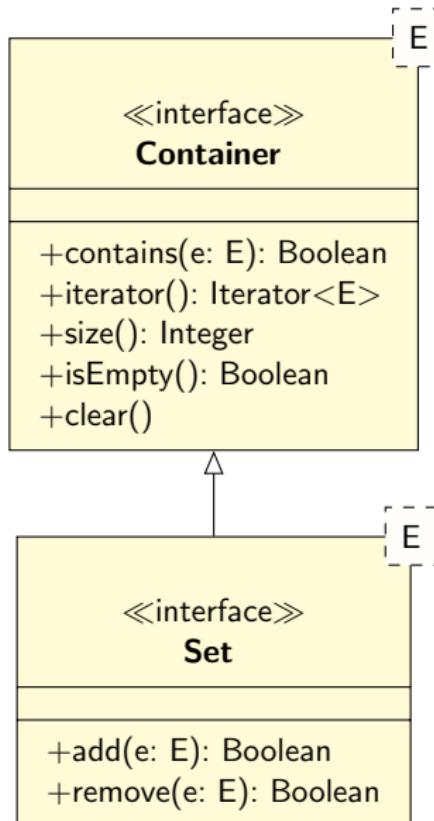


Abbildung 55: Das Interface Set.

Eigenschaften binärer Suchbäume

Die Voraussetzung für die Suche ist, dass die Schlüsselwerte in den Knoten eine **Rangfolge** haben, d. h. der Größe nach sortierbar sind. Wenn wir Objekte nehmen, die das Comparable-Interface erben, ist diese Voraussetzung erfüllt.

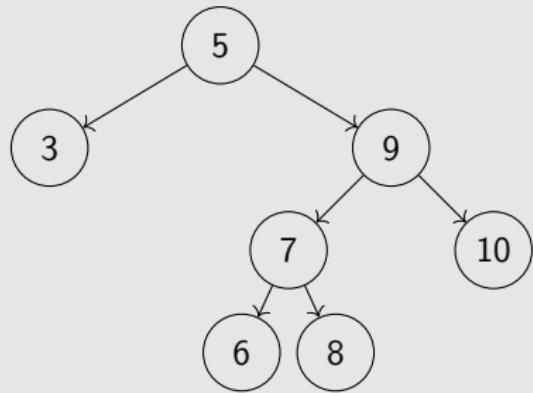
Definition binärer Suchbaum

Für jeden inneren^a Knoten n gilt: Alle Schlüsselwerte

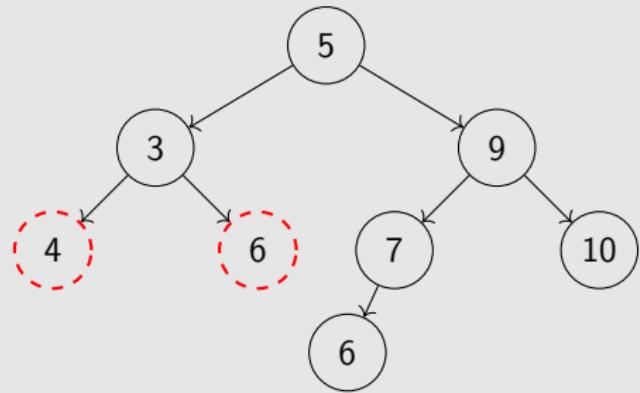
- im linken Teilbaum sind **kleiner** als der Wert vom Knoten n sowie alle Schlüsselwerte
- im rechten Teilbaum sind **größer** als der Wert vom Knoten n .

^aeinschließlich Wurzel

Beispiele



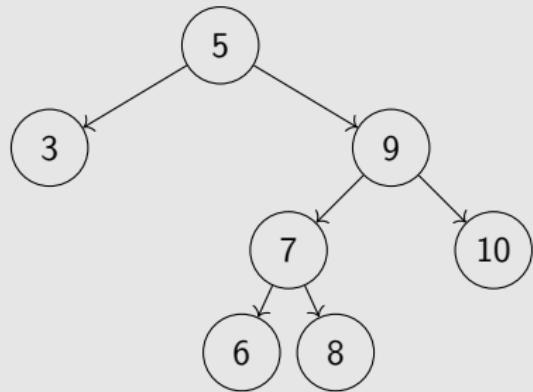
(a) Korrekter Suchbaum



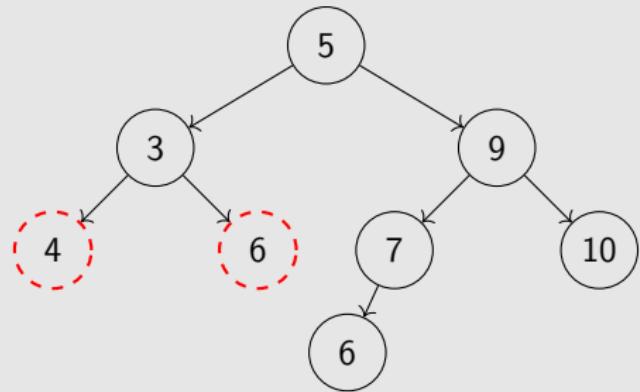
(b) Kein Suchbaum

Abbildung 56: Beispiele für binäre Suchbäume. In b) ist Element 4 nicht kleiner relativ zum Element 3 und Element 6 ist nicht kleiner relativ zur Wurzel 5.

Beispiele



(a) Korrekter Suchbaum



(b) Kein Suchbaum

Abbildung 56: Beispiele für binäre Suchbäume. In b) ist Element 4 nicht kleiner relativ zum Element 3 und Element 6 ist nicht kleiner relativ zur Wurzel 5.

Was fällt noch auf?

Elemente sind sortiert

In einem binären Suchbaum werden die Elemente sortiert gespeichert.
Die Inorder-Traversierung gibt die Elemente aufsteigend sortiert aus.

Implementierung eines veränderlichen, objekt-orientierten binären Suchbaums

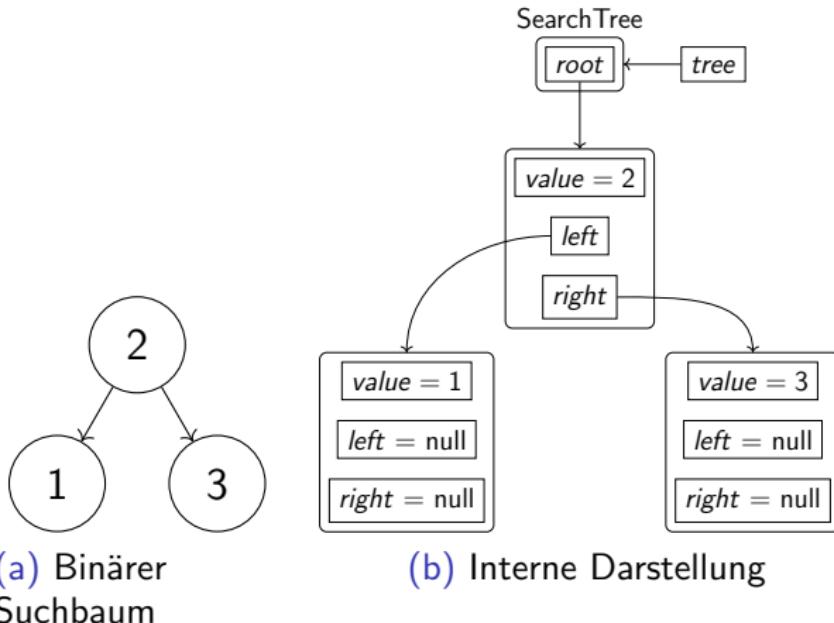


Abbildung 57: Aufbau eines binären Suchbaums mittels Knoten, die aufeinander verweisen.

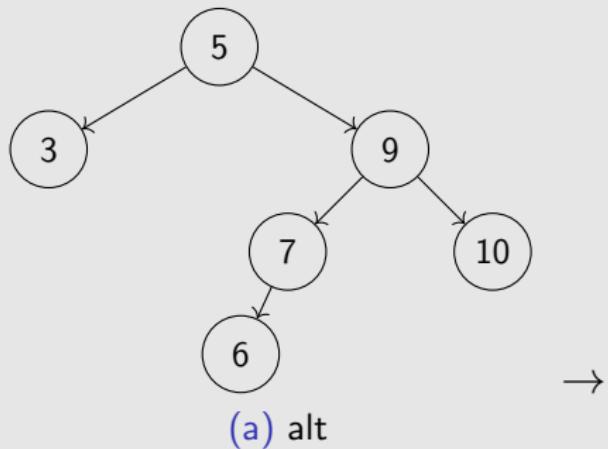
Die Klasse SearchTree

```
...
public class SearchTree<E extends Comparable<E>> implements Set<E> {

    protected SearchTreeNode<E> root; // Wurzelknoten dieses Suchbaums.
...
    public int size() {
        if (isEmpty()) { // Sonderfall leerer Baum beachten!
            return 0; // Leerer Baum hat Größe 0.
        } else {
            return root.size(); // Nimm Größe des fkt. Baums.
        }
    }
...
    public void printInorder() {
        if (!isEmpty()) {
            root.printInorder();
        }
    }
...
}
```

Etliche Methoden können direkt an die Klasse TreeNode delegiert werden.

Beispiel für das Hinzufügen des Elements 8



Beispiel für das Hinzufügen des Elements 8

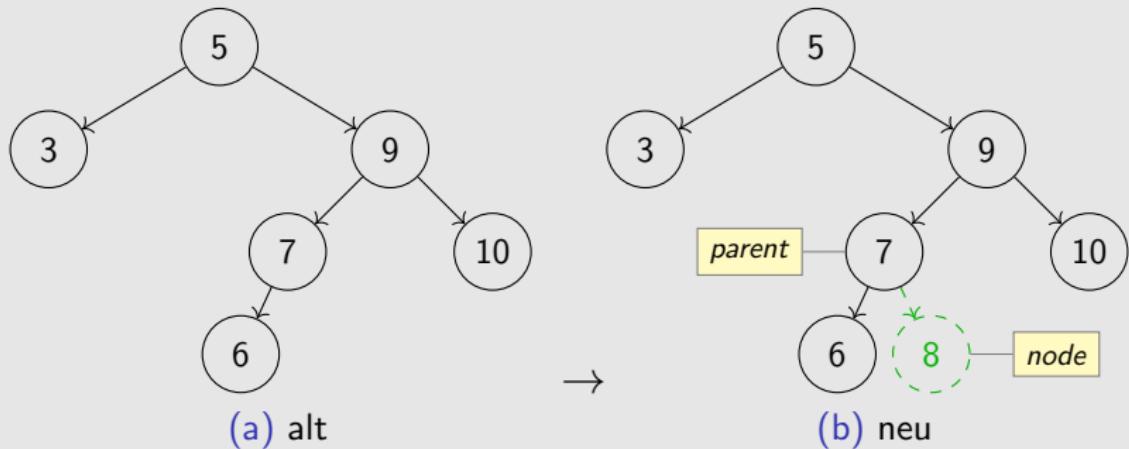


Abbildung 58: Hinzufügen eines Elements in einen binären Suchbaum. Ein neues Element (hier die 8) wird stets als Blatt hinzugefügt. Hierzu wird von der Wurzel ausgehend der Baum knotenweise durchlaufen und – ähnlich wie bei der binären Suche – in dem Teilbaum weitergesucht, in dem das Element gehört.

Hinzufügen von Elementen (Teil 1)

```
...
public class SearchTree<E extends Comparable<E>> implements Set<E> {
...
/** 
 * Füge einen Wert in den Baum hinzu. Das geht nur, wenn dieser
 * Wert noch nicht enthalten ist.
 *
 * @param o Einzufügendes Objekt.
 * @return True, wenn Wert hinzugefügt wurde, oder false, wenn nicht,
 * da dieser schon vorhanden war.
 */
public boolean add(E o) {
    // Strategie: Wir suchen den Knoten, der nach dem Einfügen des neuen
    // Elements der Elternknoten ist.

    TreeNode<E> parent = null; // (Vorläufiger) Elternknoten
    TreeNode<E> node = root; // (Vorläufiger) Kindknoten

    // Solange der aktuelle Kindknoten nicht null ist...
    while (node != null) {
        // Setze alten Kindknoten als neuen Elternknoten:
        parent = node;
        // Ist der Wert bereits in diesem Knoten gespeichert?
        if (node.getValue().equals(o)) {
            return false; // Ja, also kann er nicht nochmal eingefügt werden.
        } else if (o.compareTo(node.getValue()) < 0) {
            // Der einzufügende Wert ist kleiner als der aktuelle Knoten.
            node = node.getLeft(); // D.h. suche im linken Teilbaum weiter.
        } else {
            // Der einzufügende Wert ist größer als der aktuelle Knoten.
```

Hinzufügen von Elementen (Teil 2)

```
    node = node.getRight(); // D.h. suche im rechten Teilbaum weiter.  
}  
}  
// Erzeuge den neuen Knoten für den einzufügenden Wert:  
SearchTreeNode<E> newNode = new SearchTreeNode<>(o);  
if (parent == null) {  
    // Kein Elternknoten gefunden. D.h. Baum ist leer.  
    root = newNode; // Neuer Knoten wird Wurzelement.  
} else if (o.compareTo(parent.getValue()) < 0) {  
    // Hier wird festgestellt, ob der neue Knoten der rechte oder linke  
    // Nachfolger des Elternknotens ist.  
    parent.left = newNode; // In diesem Fall links.  
} else {  
    parent.right = newNode; // Dito für rechts.  
}  
return true; // Neuer Wert konnte erfolgreich eingefügt werden.  
}  
...  
}
```

Löschen von Elementen

Das Entfernen eines Knotens ist komplizierter als das Einfügen, da evtl. Lücken geschlossen werden müssen. Zunächst wird der zu löschende Knoten *node* gesucht. Wird er gefunden, kann er gelöscht werden, andernfalls gibt es nichts zu tun. Ein Spezialfall ist, wenn der Wurzel-Knoten gelöscht wird – hier muss die Wurzel neu gesetzt werden. Folgende Eigenschaften des Knotens bedingen unterschiedliche Löschaktionen:

- Der Knoten ist ein **Blatt**, d. h. er hat keine Kinder. Es braucht lediglich der Elternknoten (*parent*) bestimmt werden und dort die Referenz *node* entfernt zu werden.

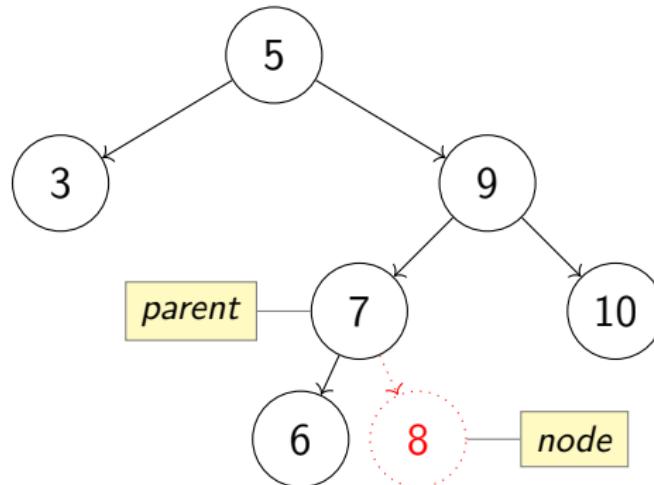
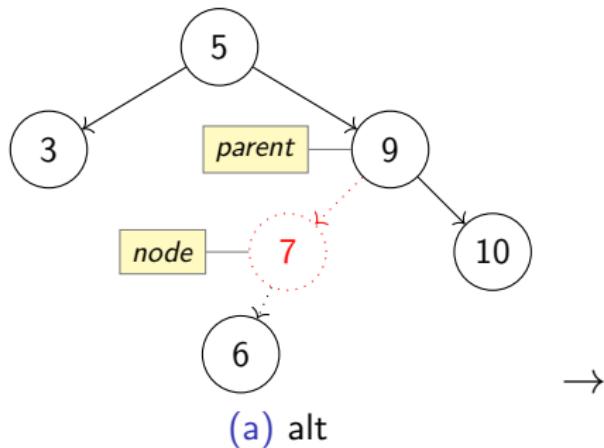


Abbildung 59: Löschen eines Blatts in einem binären Suchbaum.

- Der Knoten besitzt nur **einen** Nachfolger (Kind). Im Elternknoten wird die Referenz auf *node* ersetzt durch die Referenz auf das Kind von *node*.



- Der Knoten besitzt nur **einen** Nachfolger (Kind). Im Elternknoten wird die Referenz auf *node* ersetzt durch die Referenz auf das Kind von *node*.

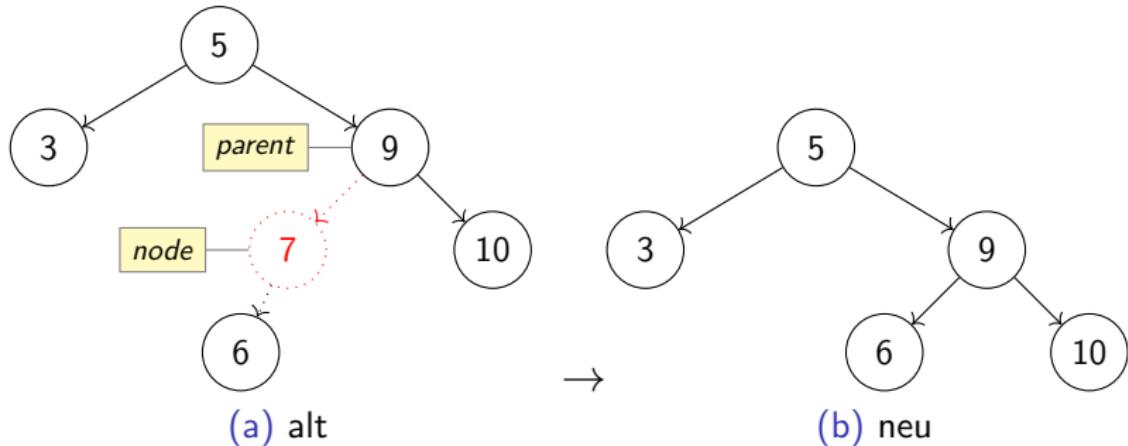
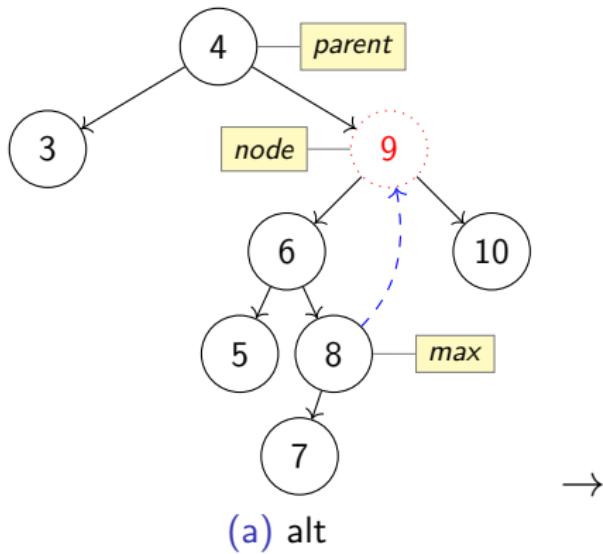


Abbildung 60: Löschen eines inneren Knotens mit nur einem Nachfolger in einem binären Suchbaum.

- Der Knoten *node* besitzt **zwei** Nachfolger (Kinder). Er wird mit dem größten Element (Knoten *max*) des linken Teilbaums oder dem kleinsten Element (Knoten *min*) des rechten Teilbaums ersetzt. Die Knoten *max* und *min* haben stets nur einen Nachfolger.



- Der Knoten *node* besitzt **zwei** Nachfolger (Kinder). Er wird mit dem größten Element (Knoten *max*) des linken Teilbaums oder dem kleinsten Element (Knoten *min*) des rechten Teilbaums ersetzt. Die Knoten *max* und *min* haben stets nur einen Nachfolger.

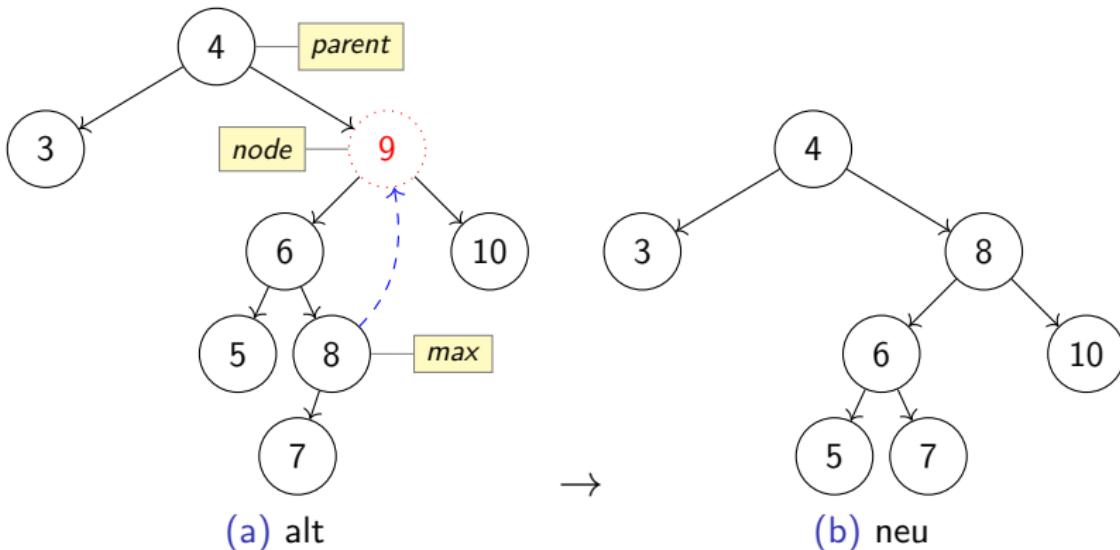


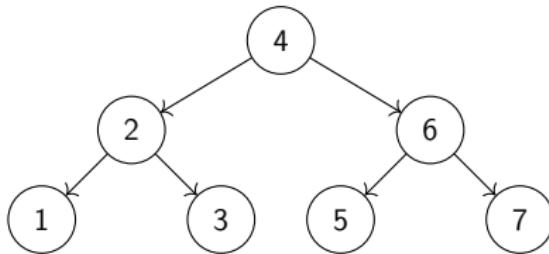
Abbildung 61: Löschen eines inneren Knotens mit zwei Nachfolgern in einem binären Suchbaum.

Löschen von Elementen im Pseudocode

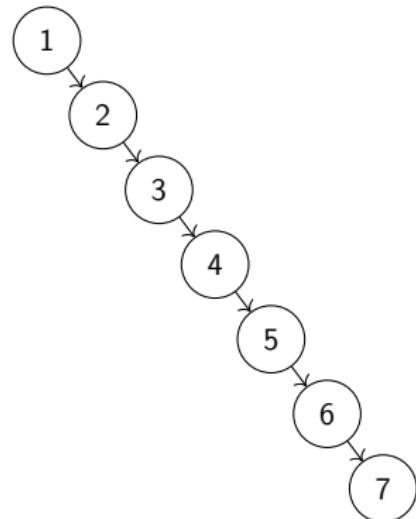
Siehe Skript! GAS-Demo zeigen.

Aufwand

Die Algorithmen für das Suchen, Einfügen oder Löschen von Elementen in einem binären Suchbaum hängen offenbar von der Länge des Pfads von der Wurzel bis zum entsprechenden Knoten ab. Das Umhängen von Baumknoten ist unabhängig von der Anzahl Knoten konstant. Der Aufwand wird folglich allein durch die Höhe des Baums bestimmt. Ist die maximale Höhe h , so ist die Komplexität der Operationen gleich $O(h)$.



(a) ausgeglichen



(b) entartet

Abbildung 62: Die Einfügereihenfolge der Elemente bestimmt das Aussehen des Baums, d. h. dieselbe Menge von Elementen führt bei unterschiedlicher Eingabereihenfolge zu unterschiedlichen Bäumen. a) zeigt einen ausgeglichenen Baum, der bei der Eingabereihenfolge 4, 2, 6, 1, 3, 5, 7 entsteht, und b) einen zur Liste entarteten Baum mit der Eingabefolge 1, 2, 3, 4, 5, 6, 7.

Ausgeglichener Baum

Ein Baum heißt ausgeglichen (oder auch balanciert), wenn bei einer gegebenen Zahl von Elementen dessen Höhe minimal ist.

Die Höhe eines Baums und dessen Topologie

Im schlechtesten Fall ist der Baum zu einer Liste der Höhe $h = n$ entartet. Im besten Fall hat jeder innere Knoten immer zwei Nachfolger. Dies bedeutet, dass es auf Ebene (Level) 0 einen, auf Ebene 1 zwei, auf Ebene 2 vier Knoten gibt bzw. allgemein auf Ebene i 2^i Knoten gibt. Ein Baum mit größtem Niveau k hat die Höhe $h = k + 1$ und fasst somit maximal

$$n = \sum_{i=0}^k 2^i = 2^{k+1} - 1 = 2^h - 1$$

Knoten. Auflösen nach h ergibt:

$$h = \log_2(n + 1)$$

Laufzeitverhalten für binäre Suchbäume

Operation	ausgeglichen	entartet
add	$O(\log n)$	$O(n)$
contains	$O(\log n)$	$O(n)$
remove	$O(\log n)$	$O(n)$
Traversierung	$O(n)$	$O(n)$

Tabelle 13: Aufwand bei binären Suchbäumen.

Anmerkung: Mittels binären Suchbäumen lassen sich n Elemente in $O(n \cdot \log n)$ sortieren. Hierzu werden diese zunächst eingefügt und dann über Traversieren sortiert ausgegeben. Dies ergibt salopp $n \cdot (O(\log n)) + O(n) = O(n \cdot \log n)$.

Abschnittsübersicht

11 Bäume

Eigenschaften von Bäumen

Binärbaum

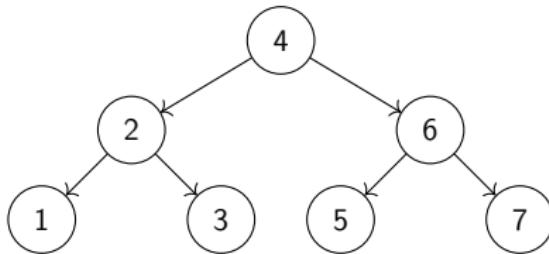
Binäre Suchbäume

Ausgeglichene Suchbäume

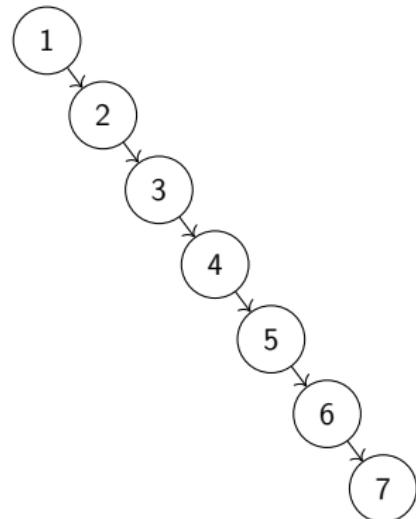
Einführung

AVL-Bäume

B-Bäume



(a) ausgeglichen



(b) entartet

Abbildung 62: Die Einfügereihenfolge der Elemente bestimmt das Aussehen des Baums, d. h. dieselbe Menge von Elementen führt bei unterschiedlicher Eingabereihenfolge zu unterschiedlichen Bäumen. a) zeigt einen ausgeglichenen Baum, der bei der Eingabereihenfolge 4, 2, 6, 1, 3, 5, 7 entsteht, und b) einen zur Liste entarteten Baum mit der Eingabefolge 1, 2, 3, 4, 5, 6, 7.

Ausgeglichene Bäume

Die Idee ist, bei jeder Einfüge- oder Löschoperation zu versuchen, den Baum wieder auszugleichen. Hierzu gibt es mehrere Datenstrukturen und Ansätze:

AVL-Baum Ein AVL-Baum – benannt nach G. M. Adelson-Velskii und E. M. Landis – ist ein höhenbalancierter binärer Suchbaum, der beim Ausgleichen einen konstanten Aufwand hat.

B-Baum Ein B-Baum (B wie balanciert, breit, buschig oder Bayer) ist ein höhenbalancierter n -ärer Baum mit unausgeglichenem Verzweigungsgrad, der vor allem im Datenbank-Bereich zum Einsatz kommt.

Rot-Schwarz-Baum Von den B-Bäumen abgeleitet sind die so genannten Rot-Schwarz-Bäume, die einen B-Baum als binären Suchbaum emulieren.

AVL-Kriterium

Ein AVL-Baum ist binärer Suchbaum, der stets dem AVL-Kriterium genügt und damit ausgeglichenen (balanciert) ist.

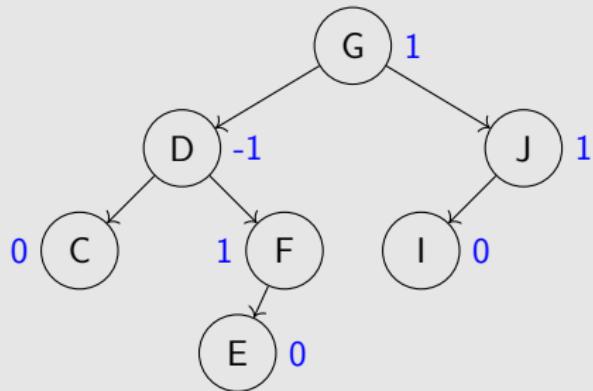
AVL-Kriterium (AVL-Eigenschaft)

Für jeden inneren Knoten incl. Wurzel ist der Betrag der Differenz der Höhen des linken und des rechten Teilbaums maximal 1.

Wenn $b(t)$ die Balance eines Teilbaums bezeichnet, gilt also

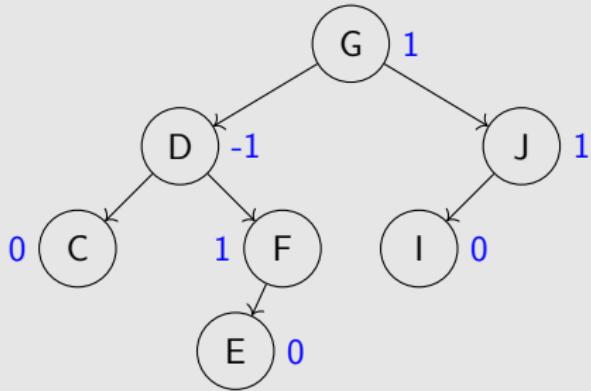
$$b(t) = \text{height}(t.\text{left}()) - \text{height}(t.\text{right}()) \in \{-1, 0, 1\}$$

Beispiel für AVL-Bäume

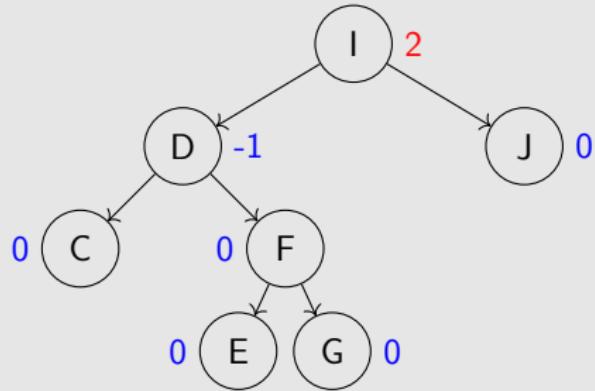


(a) AVL-Baum

Beispiel für AVL-Bäume



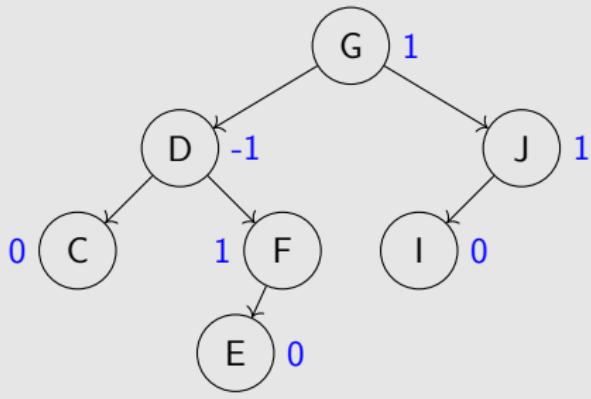
(a) AVL-Baum



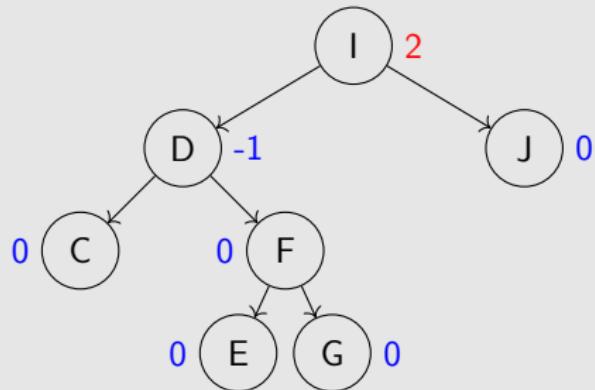
(b) kein AVL-Baum

Abbildung 63: AVL-Bäume: (a) zeigt einen AVL-Baum, (b) hingegen nicht, da Knoten I die Balance -2 hat. Neben den Knoten ist die Balance gezeigt.

Beispiel für AVL-Bäume



(a) AVL-Baum



(b) kein AVL-Baum

Abbildung 63: AVL-Bäume: (a) zeigt einen AVL-Baum, (b) hingegen nicht, da Knoten I die Balance -2 hat. Neben den Knoten ist die Balance gezeigt.

GAS-Demo!

Einfügen von Knoten

Wir wollen die Elemente 3, 2, 1 einen leeren Suchbaum einfügen.

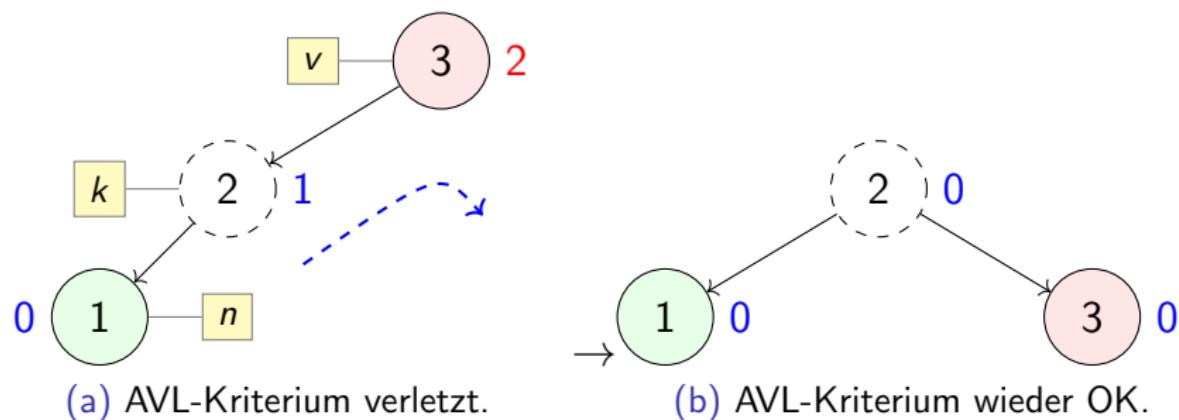


Abbildung 64: Einfache (Rechts-)Rotation über den Knoten v führt die AVL-Eigenschaft wieder herbei.

Einfügen von weiteren Knoten

Wir wollen jetzt noch die Elemente 8, 9 einfügen.

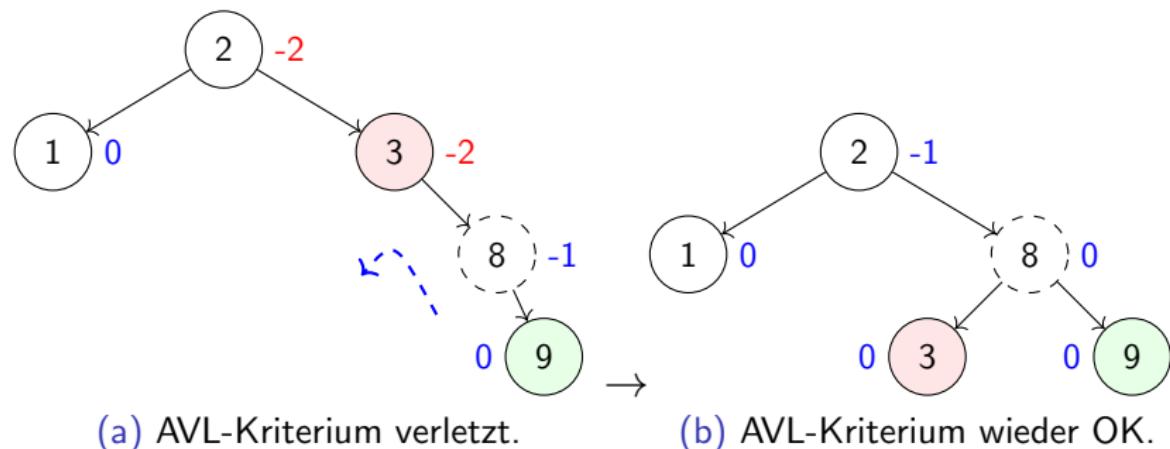


Abbildung 65: Einfache (Links)Rotation über den Knoten v führt die AVL-Eigenschaft wieder herbei.

AVL-Baum: Einfache Rotation

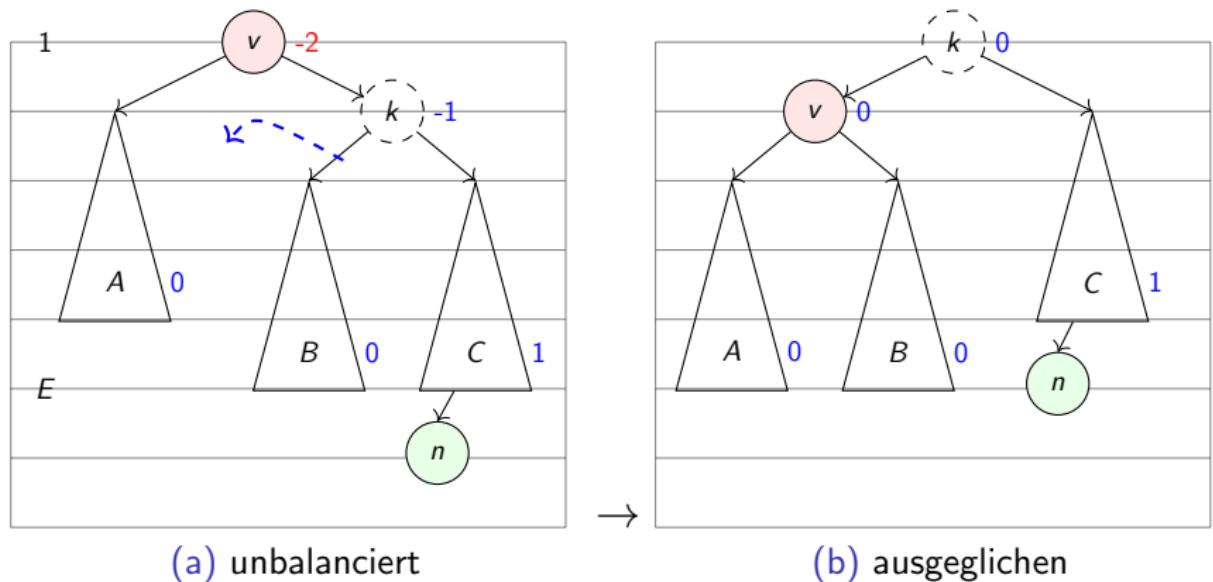
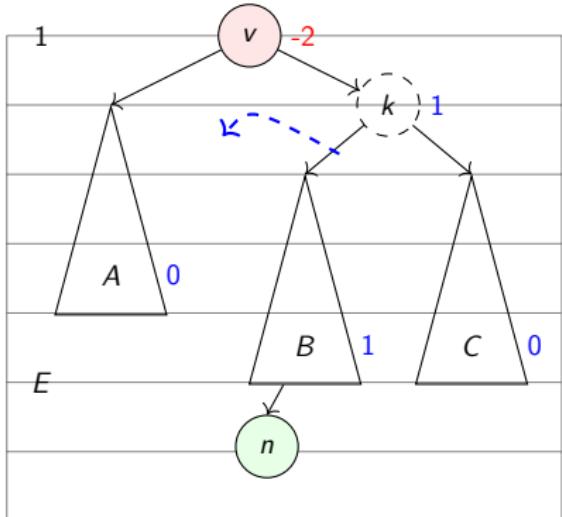
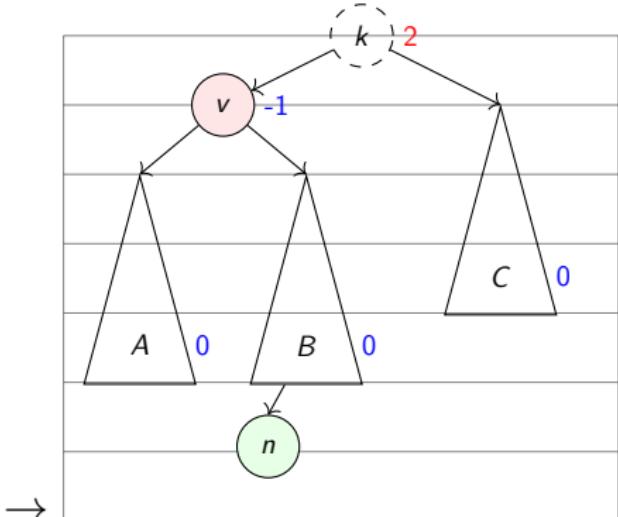


Abbildung 66: Einfache Links-Rotation über den Knoten v .



(a) unbalanciert



(b) immer noch unbalanciert

Abbildung 67: Verläuft der Pfad v, k, n zick-zack, reicht eine einfache Rotation nicht aus.

Doppel-Rotation

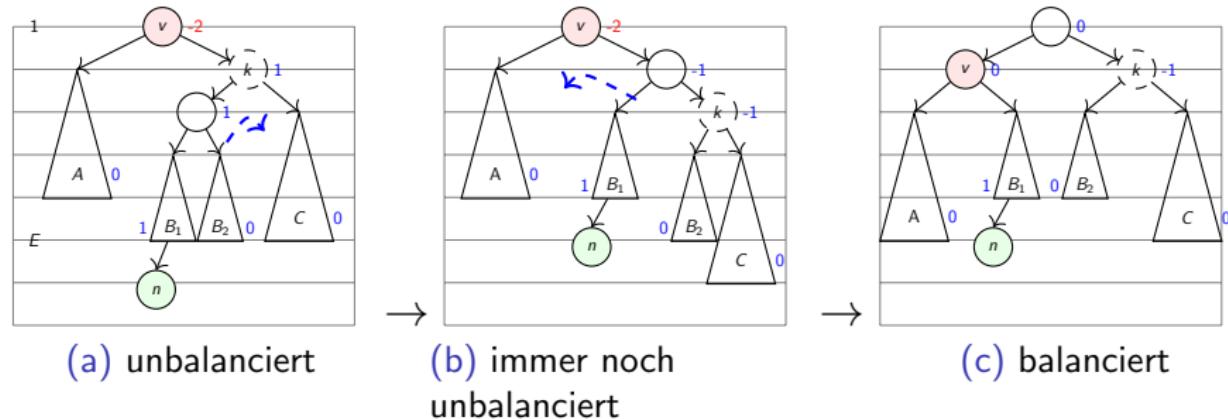


Abbildung 68: Doppelrotation: Zunächst wird über den Knoten k eine Rechts-Rotation ausgeführt. Anschließend wird eine Links-Rotation über v gemacht.

Einfügen von weiteren Knoten

Wir wollen jetzt noch das Element 5 in den Baum aus Abb. 65b einfügen.

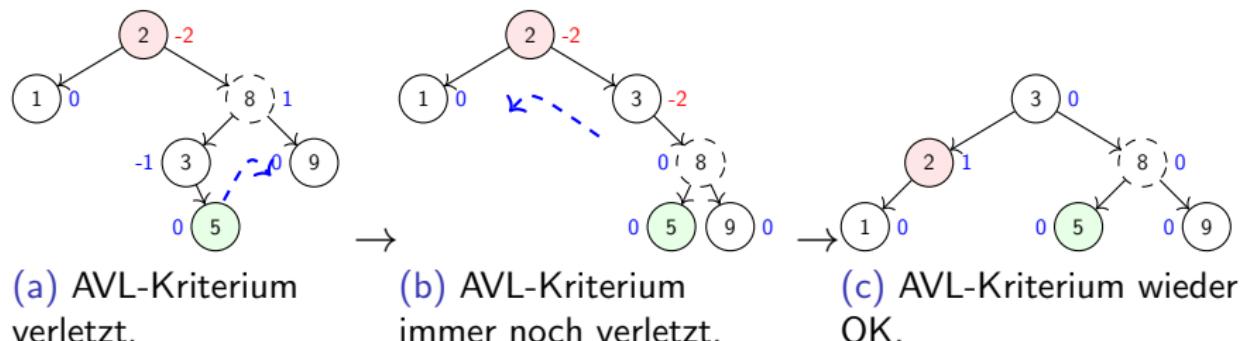


Abbildung 69: Beispiel für eine Doppelrotation im AVL-Baum.

AVL-Regeln beim Einfügen von Elementen

Algorithmus 5 Ausbalancieren nach dem Einfügen in einem AVL-Baum.

Suche in v den ersten Knoten ausgehend vom eingefügten Knoten n , der die AVL-Eigenschaft verletzt;

```
if  $v$  existiert then // Ist der Baum überhaupt unbalanciert?  
    Sei  $k$  der Nachfolger von  $v$ , dessen Teilbaum  $n$  enthält;  
    if  $k$  ist der linke Nachfolger von  $v$  then  
        if  $n$  ist im linken Teilbaum von  $k$  then  
            Führe Rechts-Rotation über  $v$  aus;  
        else //  $n$  ist im rechten Teilbaum von  $k$   
            Führe Links-Rechts-Doppelrotation aus;  
        fi  
    else //  $k$  ist der rechte Nachfolger von  $v$   
        if  $n$  ist im rechten Teilbaum von  $k$  then  
            Führe Links-Rotation über  $v$  aus;  
        else //  $n$  ist im linken Teilbaum von  $k$   
            Führe Rechts-Links-Doppelrotation aus;  
        fi  
    fi  
fi
```

AVL-Regeln beim Löschen von Elementen

Auch beim Löschen von Elementen kann ein AVL-Baum unbalanciert werden. Hier sind vergleichbare Rotationen nötig, um den Baum wieder ins Gleichgewicht zu bringen. Dies wird jedoch in dieser Vorlesung nicht besprochen.

Aufwand

Beim Einfügen (und Löschen) von Elementen sind ggf. Rotationen notwendig. Diese ändern jedoch die Gesamlaufzeitkomplexität der Operationen nicht. Einfügen hat einen Aufwand von $O(\log n)$. Das Finden des ersten verletzten Knotens v benötigt ebenfalls $O(\log n)$ Schritte, da der Aufwand ebenfalls von der Höhe des Baums abhängig ist. Die Rotationen (einfach oder doppelt) sind von der Anzahl der Elemente n im Baum unabhängig. Insgesamt bleibt es somit bei $O(\log n)$.

B-Baum

Ein B-Baum der Ordnung m ist ein Vielweg-Suchbaum mit folgenden Eigenschaften:

- ① Die Anzahl der Schlüssel in jedem Knoten – außer der Wurzel – liegt zwischen m und $2m$. Die Wurzel enthält mindestens 1 und maximal $2m$ Schlüssel.
- ② Alle Pfadlängen von der Wurzel zu einem Blatt sind gleich.
- ③ Jeder innere Knoten mit s Schlüsseln hat genau $s + 1$ Söhne.
- ④ Es gibt keine Duplikate (wie im binären Suchbaum).

Aufbau eines B-Baums

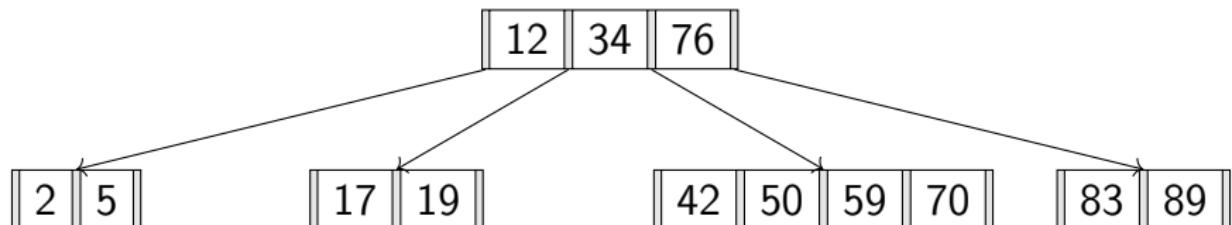


Abbildung 70: Beispiel für einen B-Baum der Ordnung $m = 2$. Die Knoten enthalten eine sortierte Liste mit Elementen sowie Verweise auf die Kinder. Ein Verweis zwischen den Elementen e_1 und e_2 genügt der Bedingung: Alle Elemente e in den Kinderknoten liegen im Bereich $e_1 < e < e_2$.

Anwendung eines B-Baums in der DB-Technologie

B-Bäume kommen in Datenbank-Systemen zum Einsatz. Jeder Knoten, der in diesem Kontext Seite (engl. page) genannt wird, wird aus einer Datei im Dateisystem geladen, um die Schlüssel zu durchsuchen. Ein Knoten enthält Verweise auf die eigentlichen Daten (oft in sog. Tabellen einer relationalen DB). Ziel ist es, mit möglichst wenigen Seitenzugriffen jeden Schlüssel und somit jeden Datensatz zu finden. Da Dateizugriffe sehr langsam sind (ca. Faktor 1.000) gegenüber Hauptspeicherzugriffen, sollen nur wenige Knoten geladen werden.

Einfügen von Elementen

Algorithmus 6 Einfügen von Elementen in einem B-Baum.

```
funct BTTree.add(v)
    if Baum ist leer then
        Lege einen neuen Knoten mit v an;
    else
        Suche vom Wurzelknoten beginnend den Schlüssel v;
        if v nicht gefunden then
            Sei p das Blatt, an dem die Suche endete;
            Füge v in das Blatt p sortiert ein;
            if p hat jetzt mehr als  $2 \cdot m$  Elemente then
                overflow(p);
            fi
            return true;
        else
            return false;                                // Da Element bereits vorhanden ist.
        fi
    fi
tcnuf
```

Algorithmus 7 Überlauf beim Einfügen in einen B-Baum.

```
proc BTTree.overflow(p) // Knoten, an dem Überlauf stattgefunden hat.  
    Teile p in der Mitte (Element e an Position  $m + 1$ ) in die  
    Knoten  $p_1$  und  $p_2$  ( $p_1$  hat Elemente der Positionen  $1, \dots, m$   
    und  $p_2$  Elemente von  $m + 2, \dots, 2m + 1$ );  
    if p hat Elternknoten q then  
        Füge Element e in den Vaterknoten q ein und verweise auf die  
        zwei neu erzeugten Nachfolgerknoten  $p_1$  und  $p_2$ ;  
        if q hat jetzt mehr als  $2m$  Elemente then  
            overflow(q);  
        fi  
    else  
        Erzeuge neuen Wurzelknoten mit Element e, der auf die zwei  
        neu erzeugten Nachfolgerknoten  $p_1$  und  $p_2$  verweist;  
    fi  
corp
```

B-Baum: Löschen eines Elements

Zunächst wird das Element bzw. dessen Knoten gesucht und das Element aus dem Knoten gelöscht. Es können nun Unterläufe (Underflows) entstehen, wenn nach dem Entfernen des Elements weniger als m Schlüssel im Knoten übrig geblieben sind. Dann gibt es prinzipiell zwei Möglichkeiten:

Balance Falls ein Nachbarknoten mehr als m Elemente hat, kann ein Element aus dem Nachbarknoten verschoben werden, um den Unterlauf zu kompensieren.

Merge Falls ein Nachbarknoten lediglich m Elemente hat, können beide Knoten zusammengefügt werden.

Beim Löschen eines Elements aus einem inneren Knoten müssen die Nachfolger neu geordnet werden. Auch hier kann der entfernte Wert durch balancieren oder mergen ausgeglichen werden.

Löschen der 83 mit Balance

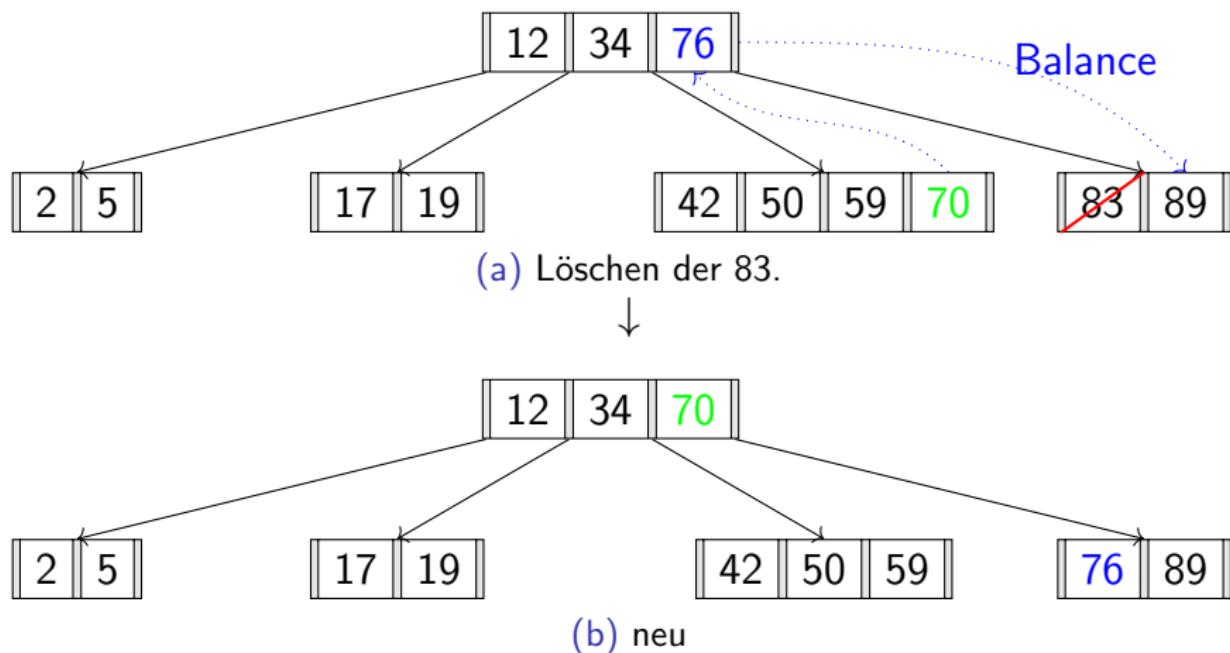


Abbildung 71: Löschen des Elements 83 mit anschließendem Ausgleich durch Balancieren.

Löschen der 17 mit Merge

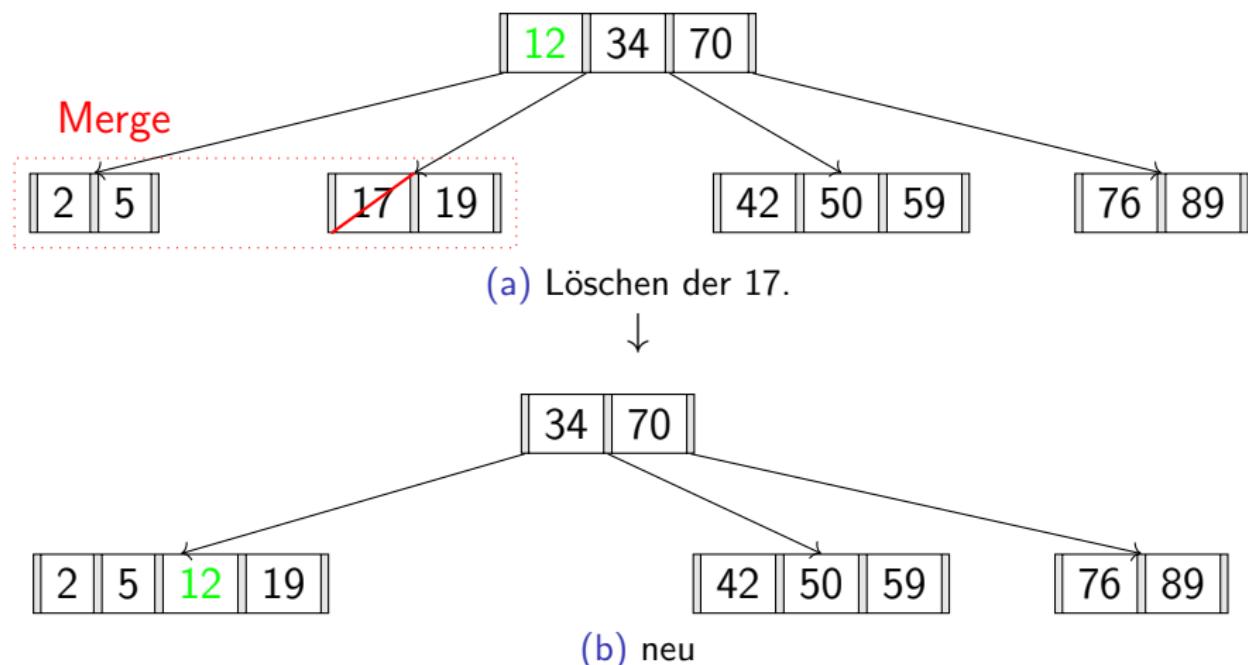


Abbildung 72: Löschen des Elements 17 mit anschließendem Ausgleich durch Mergen.

Löschen der 34 im inneren Knoten

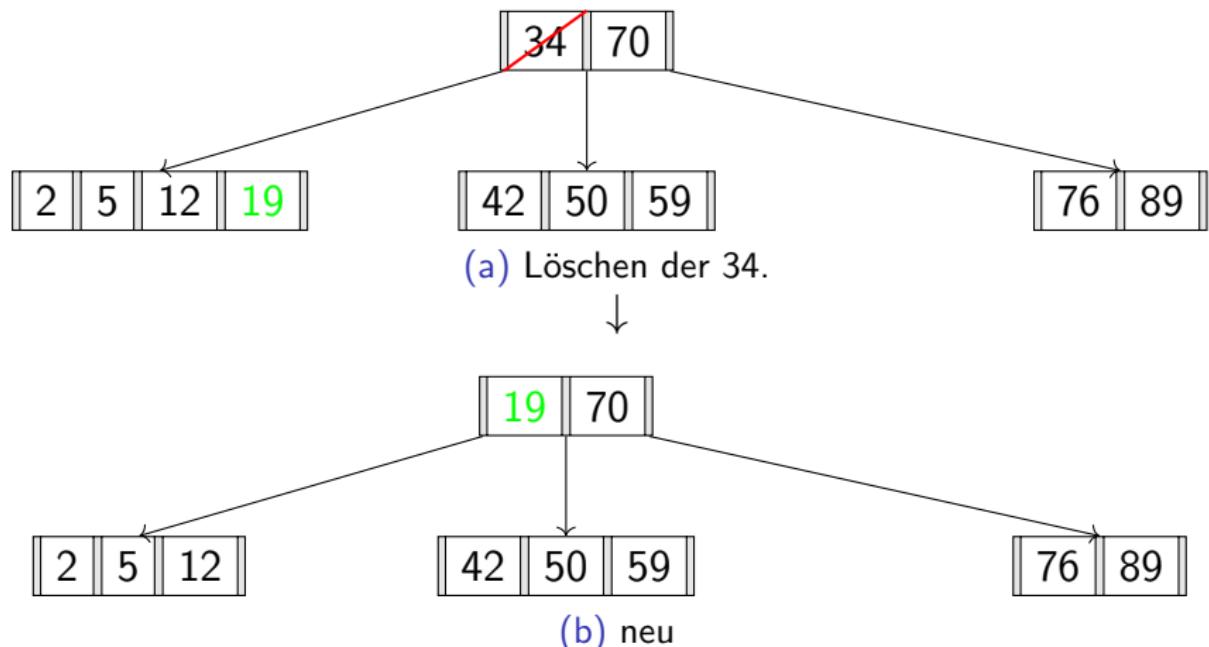


Abbildung 73: Löschen inneren des Elements 34 mit anschließendem Ausgleich durch Balancieren.

Minimal gefüllter B-Baum

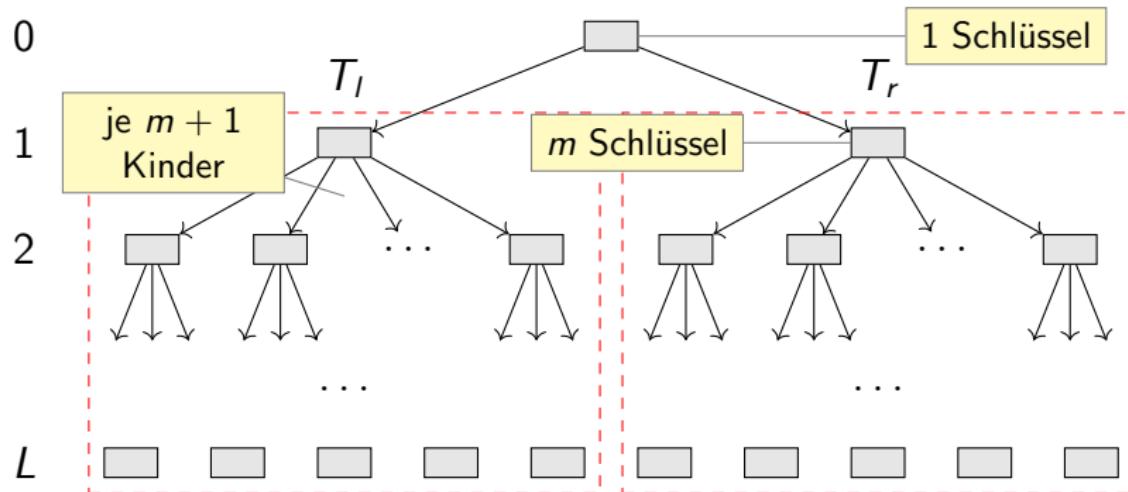


Abbildung 74: Aufbau eines minimal gefüllten B-Baums der Höhe $h = L + 1$. Der Wurzelknoten enthält minimal ein Schlüssel, die anderen Knoten m Schlüssel, was bedingt, dass jeder innere Knoten $m + 1$ Kinder hat.

Minimal gefüllter B-Baum: Anzahl Schlüssel

Die Anzahl der Knoten im linken Baum T_L ist:

$$|\text{Knoten } T_L| = 1^0 + (m+1)^1 + (m+1)^2 + \dots + (m+1)^{L-1}$$

bzw.

$$|\text{Knoten } T_L| = \frac{(m+1)^L - 1}{m}$$

Da ein Knoten m Schlüssel hat, erhalten wir:

$$|\text{Schlüssel } T_L| = m \frac{(m+1)^L - 1}{m} = (m+1)^L - 1$$

Die Gesamtanzahl der Schlüssel ist damit:

$$n = 1 + 2 \cdot ((m+1)^L - 1) = 2 \cdot (m+1)^L - 1$$

Höhe eines minimal gefüllten B-Baums

Ein nicht minimal gefüllter B-Baum mit gleicher Höhe hat sogar mehr Schlüssel, d. h. die Kapazität für Bäume der Höhe $h = L + 1$ ist allgemein:

$$n \geq 2 \cdot (m + 1)^L - 1$$

Da $h = L + 1$ ist, wird zunächst nach L umgestellt, um h in Abhängigkeit von n zu erhalten: $(m + 1)^L \leq (n + 1)/2$ bzw. $L \leq \log_{m+1}(n + 1)/2$. Somit gilt:

$$h(n) \leq \log_{m+1}(n + 1)/2 + 1 \in O(\log n)$$

Selbst ein minimal gefüllter Baum hat eine Höhe, die logarithmisch von der Anzahl der Elemente abhängt.

Laufzeitverhalten für B-Bäume

Operation	Aufwand
add	$O(\log n)$
contains	$O(\log n)$
remove	$O(\log n)$

Tabelle 14: Aufwand beim B-Baum.

Die Kosten für alle Operationen sind proportional der Höhe des Baums.

Kapitelübersicht

12 Heaps

Aufbau eines Heaps (Halde)

Heaps in einem Array

Sortieren mit Heaps

Aufwand

Abschnittsübersicht

12 Heaps

Aufbau eines Heaps (Halde)

Heaps in einem Array

Sortieren mit Heaps

Aufwand

Heap

Ein Min-Heap (Max-Heap) ist ein ausgeglichener binärer Baum, bei dem für jeden (Teil-)Baum gilt, dass alle Elemente unterhalb des Wurzelknotens **größer (kleiner)** oder gleich dem Wurzelknoten sind.

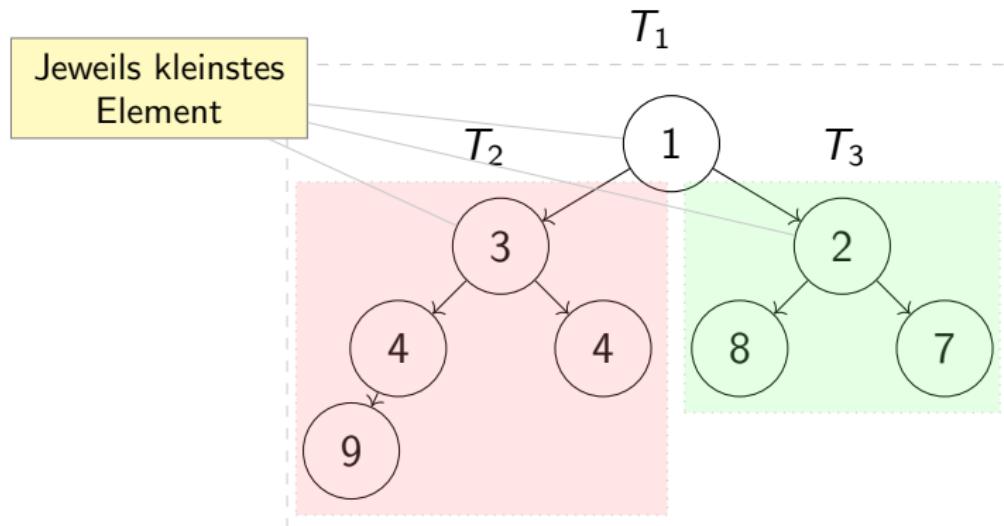


Abbildung 75: Aufbau eines Min-Heaps.

Anwendungsszenarien für einen Heap (Halde)

Die Datenstruktur Heap eignet sich gut für Warteschlangen mit Prioritäten (engl. priority queues). Die Elemente werden in einen Heap eingesortiert und das wichtigste Element steht stets vorne. Bei solchen Warteschlangen gilt das FIFO-Prinzip jedoch nicht mehr.

Methode	Funktion
$q.\text{enter}(e: E)$	Fügt ein neues Element e in die Queue q ein.
$q.\text{leave}(): E$	Entfernt das Element mit höchster Priorität aus der Queue q und gibt es zurück.
$q.\text{front}(): E$	Liest das Element mit höchster Priorität aus der Queue q , ohne es zu entfernen.

Tabelle 15: Schnittstelle einer (veränderlichen und objekt-orientierten) Prioritäten-Warteschlange.

Abschnittsübersicht

12 Heaps

Aufbau eines Heaps (Halde)

Heaps in einem Array

Einfügen von Elementen

Löschen von Elementen

Sortieren mit Heaps

Aufwand

Heap im Array: Berechnung der Position

Die Indexpositionen i_1 und i_2 der Kinderknoten von Index i ist :

$$i_1 = 2 \cdot i \text{ für Kind 1}$$

$$i_2 = 2 \cdot i + 1 \text{ für Kind 2}$$

Der Index des Elternknotens ist für $i > 1$:

$$i = \begin{cases} i/2 & \text{falls } i \text{ gerade} \\ (i-1)/2 & \text{sonst} \end{cases}$$

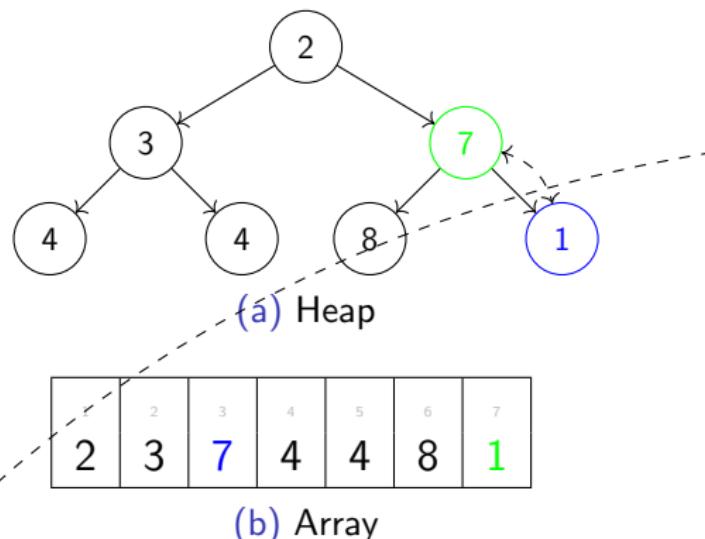


Abbildung 76: Min-Heap als Array.

Das neue Element wird an letzte Position einfügt und solange mit Elternknoten vertauscht, bis die relative Ordnung wieder stimmt.

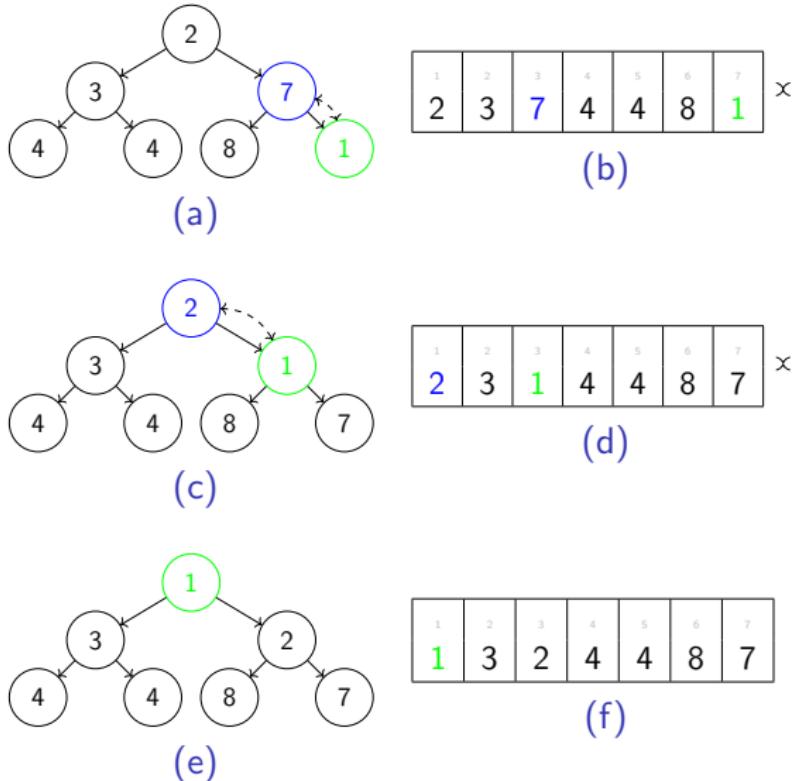


Abbildung 77: Einfügen eines Elements.

Einfügen eines Elements

```
...
public class ArrayPriorityQueue<E extends Comparable<E>> implements Queue<E>
{
    protected E[] array;
    protected int next = 0; // Indexposition für neue Elemente.

    public ArrayPriorityQueue() {
        this(100);
    }
...
    public void enter(E value) {
        if (next >= array.length) { // Voll?
            throw new ContainerException("Heap full.");
        } else {
            int idx = next; // Nächster freier Platz.
            array[next++] = value; // Element wird zunächst hinten angefügt.
            int parentIdx = calcParentIdx(idx);
            while (idx > 0 && array[parentIdx].compareTo(array[idx]) > 0) {
                swap(parentIdx, idx);
                idx = parentIdx;
                parentIdx = calcParentIdx(idx);
            }
        }
    }
...
}
```

Das Wurzelement wird durch das letzte Element ersetzt.

Dieses sinkt solange nach unten, bis die Heap-Eigenschaft wieder hergestellt ist, wobei jeweils das kleinste Nachfolgerelement nach oben durchgereicht wird.

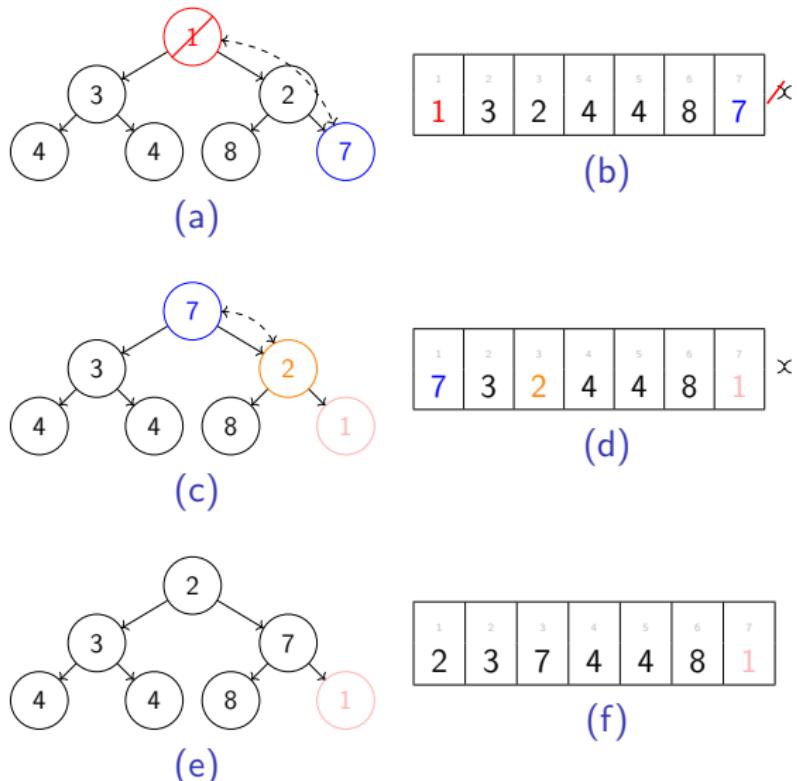


Abbildung 78: Löschen eines Elements.

Abschnittsübersicht

12 Heaps

Aufbau eines Heaps (Halde)

Heaps in einem Array

Sortieren mit Heaps

Aufwand

HeapSort

Die Elemente werden zuerst alle in einen Heap eingelesen und anschließend aus ihm „entfernt“. Aufgrund der Heap-Eigenschaft erfolgt das Auslesen sortiert. Wenn die Elemente im Array jedoch nicht gelöscht werden, bleibt ein sortiertes Array übrig. Ein Max-Heap sortiert das Array aufsteigend, ein Min-Heap hingegen absteigend.

Protokoll HeapSort

Der grüne Bereich entspricht den gelöschten Elementen.

sort() ↘

— $a =$

0	1	2	3
1	2	4	3

— $a =$

0	1	2	3
2	3	4	1

— $a =$

0	1	2	3
3	4	2	1

— $a =$

0	1	2	3
4	3	2	1

— $a =$

0	1	2	3
4	3	2	1

sort() = a ↘

Abschnittsübersicht

12 Heaps

Aufbau eines Heaps (Halde)

Heaps in einem Array

Sortieren mit Heaps

Aufwand

Aufwand Heap

Der Heap ist stets ausgeglichen, so dass es beim Einfügen oder Löschen maximal $\log n$ Vertauschungen gibt.

Operation	Aufwand
enter	$O(\log n)$
leave	$O(\log n)$
front	$O(1)$

Tabelle 16: Aufwand der Operationen einer Prioritäten-Warteschlange. Der Aufwand gilt für alle Implementierungen und Kategorien (bester, schlechtester und durchschnittlicher Fall).

Aufwand HeapSort

HeapSort ist ein schnelles Sortierverfahren mit dem Laufzeit-Aufwand:

$$O(n \log n)$$

Es werden n mal Elemente eingefügt, was eine Komplexität von (salopp) $n \cdot O(\log n) = O(n \log n)$ hat. Anschließend werden alle „gelöscht“, was ebenfalls $n \cdot O(\log n) = O(n \log n)$ ergibt. Das ist insgesamt $2 \cdot O(n \log n) = O(n \log n)$.

Optimierung

Dieses Verfahren lässt sich speziell für den HeapSort noch verbessern, die Komplexitätsklasse bleibt aber gleich.

Kapitelübersicht

13 Hashverfahren

Überblick

Offenes Hashing mittels Listen

Geschlossenes Hashing mittels Sondierung

Aufwand

Abschnittsübersicht

13 Hashverfahren

Überblick

Offenes Hashing mittels Listen

Geschlossenes Hashing mittels Sondierung

Aufwand

Operationen eines veränderlichen, objekt-orientierten binären Suchbaums

Methode	Funktion
<code>s.add(e: E): Boolean</code>	Fügt ein neues Element e in die Menge (in den Suchbaum) s ein und gibt wahr zurück, wenn dies möglich wahr oder falsch, falls ein solches Element bereits enthalten ist.
<code>s.contains(e: E): Boolean</code>	Überprüft, ob ein Element in der Menge (im Suchbaum) vorhanden ist.
<code>s.remove(e: E): Boolean</code>	Entfernt ein Element e aus der Menge (dem Suchbaum) und gibt wahr zurück, falls ein solches Element enthalten war, andernfalls falsch. Hinweis: Es kann maximal ein solches Element geben.

Tabelle 12: Schnittstelle eines binären Suchbaums. Die Operationen entsprechen dem Verwalten von Elementen in einer Menge.

Hashverfahren – Alternative für Suchbäume

Hashverfahren ermöglichen ebenfalls das effiziente Einfügen, Suchen und Löschen von Elementen in eine Datenstruktur Menge. Sie sind somit neben dem binären Suchbaum oder dem B-Baum ein weiterer Datentyp, der eine Menge realisiert. Im besten Fall haben die Mengen-Operationen sogar eine Laufzeit von $O(1)$.

Operationen einer Hashtabelle

Methode	Funktion
<code>s.add(e: E): Boolean</code>	Fügt ein neues Element e in die Menge (in die Hashtabelle) s ein und gibt wahr zurück, wenn dies möglich wahr oder falsch, falls ein solches Element bereits enthalten ist.
<code>s.contains(e: E): Boolean</code>	Überprüft, ob ein Element in der Menge (in der Hashtabelle) vorhanden ist.
<code>s.remove(e: E): Boolean</code>	Entfernt ein Element e aus der Menge (der Hashtabelle) und gibt wahr zurück, falls ein solches Element enthalten war, andernfalls falsch. Hinweis: Es kann maximal ein solches Element geben.

Tabelle 17: Schnittstelle einer Hashtabelle. Die Operationen entsprechen dem Verwalten von Elementen in einer Menge.

Idee von Hashverfahren

Eine Hashtabelle hat ein intern verwendetes Array mit fester Länge m . Eine **Hashfunktion** berechnet für jedes Element eine Indexposition in der Hashtabelle. Die Arrayplätze werden auch als **Buckets** (für engl. Eimer, Gefäß) und die Größe m als **Kapazität** der Hashtabelle bezeichnet.

- ① Jedes Element muss zuvor in eine Zahl, dem sogenannten **Hashwert** (hashcode), umgewandelt werden: $z : E \rightarrow \mathbb{N}$
- ② Anschließend wird der Zahlenwert auf die Arraygrenzen 0 bis $m - 1$ mittels der eigentlichen Hashfunktion $h : \mathbb{N} \rightarrow \mathbb{N}$, meist einer Modulo-Funktion, abgebildet.

Wichtig ist, dass die Hashfunktion die Elemente gleichmäßig auf die Indexpositionen verteilt („to hash“ für engl. „zerhacken“).

Beispiel: Hashcode für einen String

„The hash code for a String object is computed as $s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \dots + s[n-1]$ using int arithmetic, where $s[i]$ is the i -th character of the string, n is the length of the string. (The hash value of the empty string is zero.)“

aus der Java-Dokumentation über die Klasse String.

Hashverfahren

Die (grobe) Strategie zum Manipulieren von Hashtables ist

Einfügen: Berechne den Index und füge das Element an dieser Position in das Array ein.

Suchen: Berechne den Index und schaue, ob ein Element dort vorhanden ist.

Löschen: Berechne den Index und entferne das Element dort.

Verwaltung von Monatsnamen

Es sollen Monatsnamen in einer Hashtabelle der Kapazität $m = 17$ verwaltet werden. Zur Berechnung des Hashwert eines Monatsnamen $c = c_1 c_2 \dots c_k$ werden die ersten drei Buchstaben in Großbuchstaben umgewandelt und davon der Ordinalwert gebildet. Ihre Summe ist der Hashwert:

$$z(c) = \text{ord}(c_1) + \text{ord}(c_2) + \text{ord}(c_3)$$

Beispiel: $z(\text{Juni}) = \text{ord}(\text{J}) + \text{ord}(\text{U}) + \text{ord}(\text{N}) = 10 + 21 + 14 = 45$.
Die Hashfunktion sei:

$$h(z) = z \bmod m$$

Somit ist $h(\text{Juni}) = z(\text{Juni}) \bmod 17 = 45 \bmod 17 = 11$.

A	1	N	14
B	2	O	15
C	3	P	16
D	4	Q	17
E	5	R	18
F	6	S	19
G	7	T	20
H	8	U	21
I	9	V	22
J	10	W	23
K	11	X	24
L	12	Y	25
M	13	Z	26

Tabelle 18:
Ordinalwerte

0	Nov.	9	Juli
1	Apr., Dez.	10	
2	Mrz.	11	Juni
3		12	Aug., Okt.
4		13	Feb.
5		14	
6	Mai, Sep.	15	
7		16	
8	Jan.		

Abbildung 79: Hashtabelle mit 12 Monatsnamen. Offenbar kommt es zu Kollisionen (rot).

Der Füllgrad α bezeichnet den Anteil der belegten Plätze in %, d. h. $\alpha = n/m$ (n : Anzahl der Elemente und m : Arraygröße).

Wahrscheinlichkeit für Kollisionen

Die Wahrscheinlichkeit, n Elemente in eine Hashtabelle der Kapazität m ohne Kollision einzufügen ist ($1 \leq n \leq m$):

$$P = 1 - \frac{m!}{(m-n)!m^n}$$

Beispiele

m	n	P
17	12	0,99

(a) Monatsnamen

m	n	P
365	22	0,48
365	23	0,51
365	50	0,97

(b) Geburtstage

Tabelle 19: Beispiele für die Wahrscheinlichkeit a) von Kollisionen von Monatsnamen und dass b) zwei oder mehr Schüler am gleichen Tag Geburtstag haben.

Abschnittsübersicht

13 Hashverfahren

Überblick

Offenes Hashing mittels Listen

Geschlossenes Hashing mittels Sondierung

Aufwand

Offenes Hashing mittels Listen

Um Überläufe handhaben zu können, kann in einer Hashtabelle jeder Bucket durch eine verkettete Liste realisiert werden. Elemente werden an den Anfang der Liste angefügt bzw. darin gesucht oder daraus gelöscht. Eine leere Liste entspricht einem nicht belegten Bucket. Dieses Verfahren wird als offenes Hashing bezeichnet, da prinzipiell beliebig viele Überläufe stattfinden können. Achtung: Manchmal wird dieses Verfahren auch als geschlossen bzgl. der Indexpositionen bezeichnet.

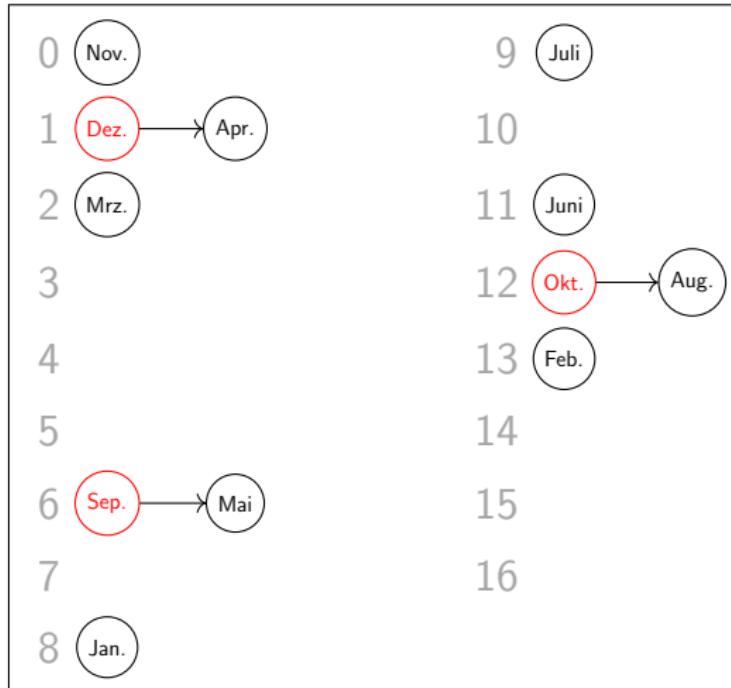


Abbildung 80: Hashtabelle mit verketteter Liste für Überläufe. Neue Elemente (rot) werden vorne eingefügt.

Abschnittsübersicht

13 Hashverfahren

Überblick

Offenes Hashing mittels Listen

Geschlossenes Hashing mittels Sondierung

Lineares Sondieren

Quadratisches Sondieren

Löschen von Elementen beim Sondieren

Aufwand

Kollisionsbehandlung mit Sondieren

Solange noch Buckets frei sind, können Überläufe in einer Hashtabelle auch durch das Suchen von alternativen Arrayplätzen realisiert werden. Dieses Verfahren nennt man Sondieren bzw. geschlossenes Hashing. Achtung: Auch hier wird dieses Verfahren manchmal als offen bzgl. der Indexpositionen bezeichnet. Die Idee ist wie folgt: Ist ein Bucket belegt, wird eine „neue“ Hashfunktion h_i solange ($i = 1, 2, 3, \dots, m$) probiert, bis ein freier Platz gefunden ist.

Fehlerfall

Tritt während der Suche der Fall $h(z) = h_i(h(z))$ ein, d. h. der ursprüngliche Indexplatz wird wieder erreicht, so können fortan keine freien Buckets mehr gefunden werden und die Suche wird abgebrochen.

Lineares Sondieren

Beim linearen Sondieren ist die neue Hashfunktion:

$$h_i(z) = (h(z) + c \cdot i) \bmod m$$

mit $1 \leq i \leq m$. Abb. 81 demonstriert für das Monatsnamenbeispiel, wie die alternativen Arrayplätze gefunden werden.

Achtung

Damit auch alle möglichen Plätze gesucht werden, sollten c und m für $c > 1$ teilerfremd sein.

0	Nov.	9	Juli
1	Apr.	10	
2	Mrz.	11	Juni
3	Dez. ↙ +(1 · 1)	12	Aug. ↙ +(1 · 2)
4		13	Feb. ↙
5		14	Okt. ↙
6	Mai ↙	15	
7	Sep. ↙	16	
8	Jan.		

Abbildung 81: Lineares Sondieren mit $c = 1$. Der Monat Dezember gehört ursprünglich an Position 1, die jedoch belegt ist. Mit zwei Sondierungsschritten ($i = 1$ und $i = 2$) wird an Position 3 ein freier Platz gefunden.

Die Klasse Hashtable

```
...
public class Hashtable<E> implements Set<E> {

    protected HashtableEntry<E>[] hashtable;
    private int c = 1; // Faktor für die Sondierung.
...
/*
 * Erzeugt eine Hashtabelle mit der Kapazität capacity,
 * die lineares Sondieren mit Schrittweite c verwendet.
 *
 * @param capacity Die interne Größe des genutzten Arrays.
 * @param c          Schrittweite
 */
public Hashtable(int capacity, int c) {
    hashtable = (HashtableEntry<E>[] ) new HashtableEntry[capacity];
    clear();
    this.c = c;
}
...
}
```

add()-Methode der Hashtable

```
...
public class Hashtable<E> implements Set<E> {
...
    public boolean add(E element) {
        int oIdx = hash(element.hashCode(), 0); // Erstes versuchtes Bucket.
        int idx = oIdx;                      // Lauf-Index.
        int i = 1;
        while (hashtable[idx].isEnganged() && !hashtable[idx].isSame(element)) {
            idx = hash(oIdx, i);           // Alternativplatz.
            if (idx == oIdx) { // Erster Indexplatz wieder erreicht?
                throw new ContainerException("Hashtable full!");
            }
            i++;
        }
        if (hashtable[idx].isSame(element)) { // Element bereits enthalten?
            return false;      // Einfügen wird abgelehnt.
        } else {
            hashtable[idx].setElement(element);
            return true;
        }
    }
...
}
```

hash()-Methode der Hashtable

```
...
public class Hashtable<E> implements Set<E> {
...
    protected int hash(int idx, int i) {
        return (idx + c * i) % hashtable.length;
    }
}
...
}
```

Anwendung der Hashtable

```
...
Hashtable<Integer> hashtable = new Hashtable<>(10);
int[] elements = {10, 21, 10, 2, 6, 9, 20, 29};
for (int i = 0; i < elements.length; i++) {
    boolean result = hashtable.add(elements[i]);
    if (result == false) {
        System.out.println(elements[i] +
            " konnte nicht hinzugefuegt werden.");
    }
}
System.out.println(hashtable.mkString());

int[] test = {21, 7, 2};
for (int i = 0; i < test.length; i++) {
    boolean contains = hashtable.contains(test[i]);
    if (contains == true) {
        System.out.println(test[i] + " ist enthalten.");
    } else {
        System.out.println(test[i] + " ist nicht enthalten.");
    }
}
...
}
```

Konsole

10 konnte nicht hinzugefuegt werden.

0: 10

1: 21

2: 2

3: 20

4: 29

5: 0

6: 6

7: 0

8: 0

9: 9

21 ist enthalten.

7 ist nicht enthalten.

2 ist enthalten.

Quadratisches Sondieren

Beim quadratischen Sondieren ist die neue Hashfunktion:

$$h_i(z) = (h(z) + i^2) \bmod m$$

mit $1 \leq i \leq m$. Im Gegensatz zum linearen Sondieren werden die Sprünge durch die quadratische Funktion immer größer, d. h. neue Elemente werden durchaus weit entfernt von der ursprünglichen Position eingefügt.

0	Nov.	9	Juli
1	Apr.	10	
2	Mrz.	11	Juni
3		12	Aug.
4		13	Feb.
5	Dez.	14	
6	Mai	15	
7	Sep.	16	Okt.
8	Jan.		

Abbildung 82: Quadratisches Sondieren. Der Monat Dezember gehört ursprünglich an Position 1, die jedoch belegt ist. Mit zwei Sondierungsschritten ($+1^2$ und $+2^2$) wird an Position 5 ein freier Platz gefunden.

Löschen von Elementen beim Sondieren

Beim Löschen eines Elements in Hashtables basierend auf verketteten Listen kann das Element einfach aus der Liste ausgeklinkt werden. Bei Hashtables mit Sondierung kann das Element nicht einfach aus dem Bucket entfernt werden, da sonst die Sondierungskette unterbrochen wäre und beim Suchen ein Element möglicherweise nicht mehr gefunden wird (siehe Abb. 83 auf der nächsten Folie). Eine Lösung ist es, den Bucket-Platz nach dem Entfernen des Elements nicht leer zu lassen, sondern zum Überschreiben zu markieren.

0	Nov.	9	Juli
1	Apr.	10	
2	Mrz.	11	Juni
3	Dez. ↙	12	Aug. ↘
4		13	Feb. ↗
5		14	Okt. ↙
6	Mai ↘	15	
7	Sep. ↗	16	
8	Jan.		

Abbildung 83: Löschen beim linearen Sondieren. Würde z. B. der Februar beim Löschen entfernt, verursacht das eine Lücke und der Oktober würde nicht mehr gefunden.

Abschnittsübersicht

13 Hashverfahren

Überblick

Offenes Hashing mittels Listen

Geschlossenes Hashing mittels Sondierung

Aufwand

Laufzeitverhalten

Das Suchen, Einfügen und Löschen von Elementen in einer Hashtabelle kann (fast) gleich behandelt werden und gilt sowohl für Hashtables basierend auf Listen wie auch für das Sondieren. Bei allen drei Operationen muss zunächst der Bucket-Platz ermittelt werden. Im besten Fall haben diese Operationen $O(1)$, d. h. es kommt zu keiner Kollision. Im schlechtesten Fall gilt $O(n)$, d. h. es kommt ausschließlich zu Kollisionen. Im Mittel wird das Verhalten vom Füllgrad $\alpha = n/m$ abhängig sein. Außerdem spielen die Eigenschaften der Hashfunktion h und die Erzeugung der Hashcodes z eine Rolle.

Operation	Fall	Liste oder Sondieren
add	bester	$O(1)$
	durchschnittlich	$O\left(\frac{1}{1-\alpha}\right)$
	schlechtester	$O(n)$
contains	bester	$O(1)$
	durchschnittlich erfolgreich/-los	$O\left(\frac{1}{\alpha} \log \frac{1}{1-\alpha}\right) / O\left(\frac{1}{1-\alpha}\right)$
	schlechtester	$O(n)$
remove	bester	$O(1)$
	durchschnittlich	$O\left(\frac{1}{\alpha} \log \frac{1}{1-\alpha}\right)$
	schlechtester	$O(n)$

Tabelle 20: Übersicht Laufzeitverhalten bei Hashtabellen unter der Annahme, dass die Hashcodes optimal „zerstreut“ sind. $1/(1 - \alpha)$ gibt die Anzahl der Versuche an, bis ein Platz gefunden wird (Füllgrad $\alpha < 1$ vorausgesetzt).

Optimaler Füllgrad α

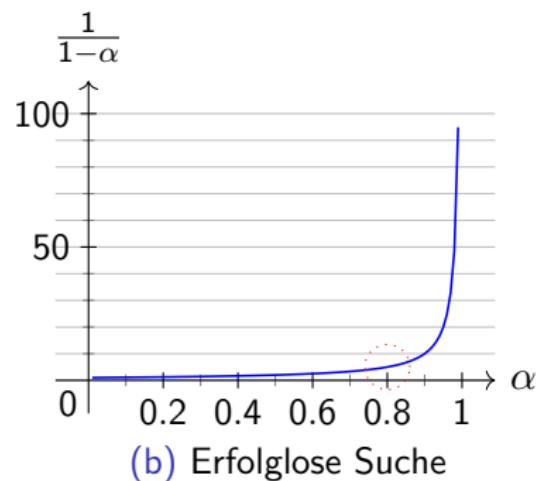
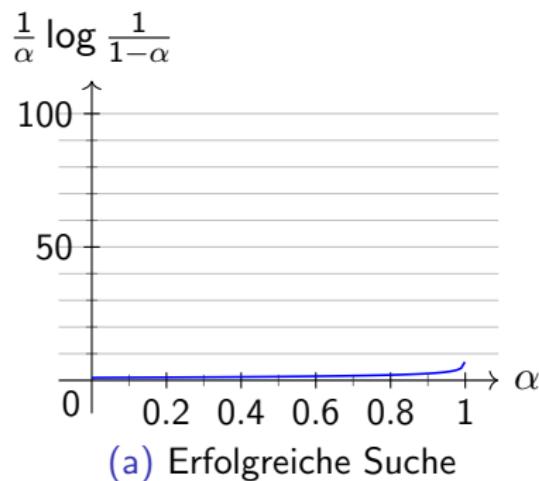


Abbildung 84: Anzahl der Schritte in Abhängigkeit des Füllfaktors. a) Erfolgreiche Suche. b) Erfolglose Suche, Einfügen und Löschen. Ab einem Füllgrad von ca. 80 % (roter Kreis) ist das Laufzeitverhalten schlecht.

Rehashing / Ausblick

Ist der Füllgrad zu groß oder das Array voll, kann ein **Rehashing** durchgeführt werden. Hierbei wird das Array vergrößert und alle Elemente werden neu eingefügt. Ein vorteilhafter Seiteneffekt ist, dass alle zum Löschen markierte Plätze ebenfalls freigegeben werden können. Falls oft Elemente gelöscht werden, muss das Rehashen häufig erfolgen, d. h. eine Hashtabelle ist nicht wirklich eine dynamisch Datenstruktur. Das sogenannte dynamische Hashen passt die Arraygröße automatisch an.

Die **Reihenfolge** der Elemente in Hashtables ist quasi **zufällig**, da die Hashwert-Berechnung möglichst zerstreut erfolgen soll. Anders als im binären Suchbaum liegen die Elemente nicht sortiert vor.

Kapitelübersicht

14 Einleitung

Eigenschaften von Sortieralgorithmen

Visualisierung von Sortierverfahren

Sortieren

Das Sortieren von Datensätzen ist in der Praxis eine häufig vorkommende Aufgabe. Viele Anwendungen setzen – aus unterschiedlichen Gründen – sortierte Datenmengen voraus:

- Prüfungsnoten nach Matrikelnummer
- Bücherliste sortiert nach Namen
- uvm.

Aber auch für andere Algorithmen ist es oft wichtig, dass Daten sortiert vorliegen. Beispielsweise geht das Suchen dann schneller.

Abschnittsübersicht

14 Einleitung

Eigenschaften von Sortieralgorithmen

Visualisierung von Sorterverfahren

Interne und externe Verfahren

Es werden zwei grundlegende Verfahren unterschieden:

Interne Verfahren Beim Sortieren werden alle Daten (z. B. Arrays) im Hauptspeicherstrukturen gehalten. Folglich können nicht mehr Daten sortiert werden als der Hauptspeicher groß ist.

Externe Verfahren Wenn der Hauptspeicher zum Sortieren nicht ausreicht, kann mit Hilfe eines externen Speichermediums sortiert werden. Das ist aufgrund der langsameren Zugriffszeiten langsamer und sollte vermieden werden.

Stabilität eines Sortierverfahrens

Ein Sortieralgorithmus wird **stabil** genannt, wenn zwei gleiche Schlüssel nach dem Sortieren in der gleichen Reihenfolge stehen. Andernfalls heißt es **instabil**.

Name	Größe (cm)
Bill Gates	179
Albert Einstein	173
Steve Jobs	184
Alan Turing	181
Konrad Zuse	179

(a) unsortiert

Name	Größe (cm)
Albert Einstein	173
Bill Gates	179
Konrad Zuse	179
Alan Turing	181
Steve Jobs	184

(b) stabil sortiert nach
Größe

Name	Größe (cm)
Albert Einstein	173
Konrad Zuse	179
Bill Gates	179
Alan Turing	181
Steve Jobs	184

(c) instabil sortiert nach
Größe

Tabelle 21: Stabilität von Sortierverfahren

Abschnittsübersicht

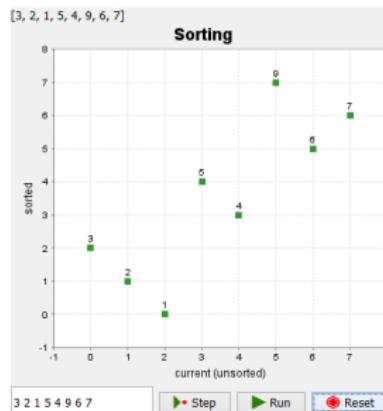
14 Einleitung

Eigenschaften von Sortieralgorithmen

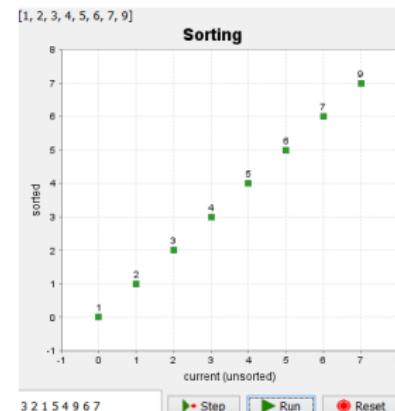
Visualisierung von Sortierverfahren

Visualisierung von Sortierverfahren

Das Sortieren kann visualisiert werden, indem die Ist-Indexposition aller Elemente gegenüber der Soll-Position aufgetragen wird. Liegen die Punkte auf einer Diagonalen ist die Folge sortiert.



(a) unsortiert



(b) sortiert

Abbildung 85: Visualisierung von Suchverfahren

Kapitelübersicht

15 Einfache Sortierverfahren

BubbleSort: Sortieren durch Austauschen

InsertionSort: Sortieren durch Einfügen

SelectionSort: Sortieren durch Austauschen

Schnittstelle für alle Sortieralgorithmen

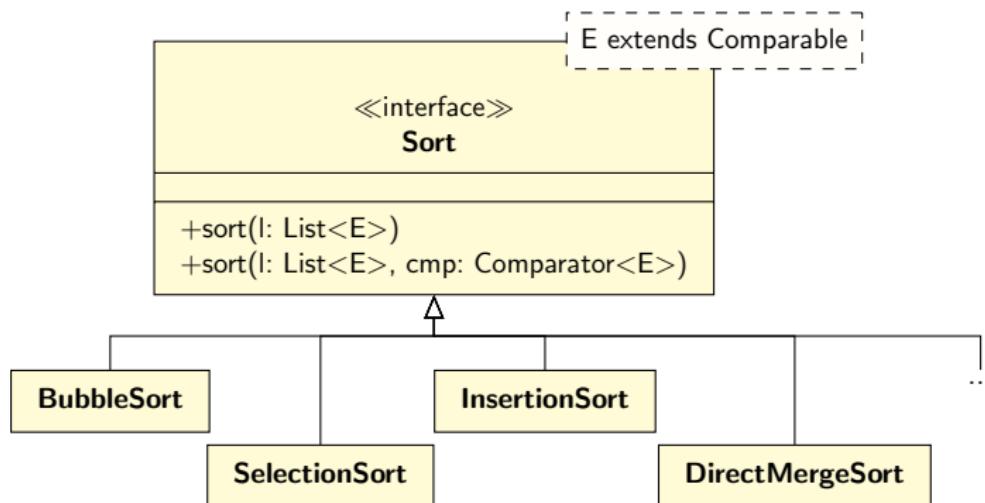


Abbildung 86: Alle Sortieralgorithmen implementieren diese Schnittstelle Sort.

Abschnittsübersicht

15 Einfache Sortierverfahren

BubbleSort: Sortieren durch Austauschen

Algorithmus

Aufwand und Stabilität

Visualisierung

InsertionSort: Sortieren durch Einfügen

SelectionSort: Sortieren durch Austauschen

Sortieren durch Austauschen: BubbleSort

Beim BubbleSort vergleicht man paarweise von links nach rechts zwei benachbarte Elemente. Entsprechen diese Elemente nicht der Sortierreihenfolge, vertauscht man sie. Elemente, die größer als ihre Nachfolger sind, überholen diese und bewegen sich zum Ende der Folge hin.

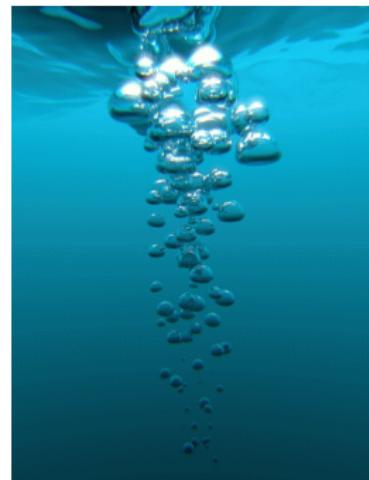


Abbildung 87: Größere Luftblasen in einer Flüssigkeit steigen schneller auf als kleinere
(© mathcs.emory.edu).

BubbleSort als Algorithmus

Mit dem ersten Durchlauf wird das insgesamt größte Element an Position n gebracht. Beim zweiten Durchlauf wird das zweitgrößte Element an Position $n - 1$ eingesortiert usw. Steht das kleinste Element am Ende der Folge, rutscht es mit jedem Durchlauf nur um eine Position nach vorne. Folglich sind in diesem Fall weitere Wiederholungen nötig, damit es an Position 1 rutscht (insgesamt $n - 1$ Iterationen).

Da nach einem Durchlauf ein *relatives größtes* Element nach hinten gewandert ist, kann in der darauffolgenden Iteration der Vergleich eine Position vorher abgebrochen werden, da hier kein kleineres Element mehr stehen kann.

BubbleSort im Pseudocode

Algorithmus 8 Bubblesort (Variante 1)

```
proc BubbleSort1( $F$ )                                // Folge  $F$  der Länge  $n$ 
    for  $j$  von 1 bis  $n - 1$  do                      // Wende  $n - 1$ -mal an.
        for  $i$  von 1 bis  $n - j$  do                  // Vergleiche nur bis zur Position  $n - j$ .
            if  $F[i] > F[i + 1]$  then                // Reihenfolge zweier Elemente falsch?
                Vertausche beide Elemente;
            fi
        od
    od
corp
```

BubbleSort als Java-Programm

...

```
/**  
 * @param list Die zu sortierende Liste (wird verändert).  
 */  
public void sort(List<E> list, Comparator<E> cmp) {  
    for (int j = 1; j <= list.size() - 1; j++) {  
        for (int i = 0; i <= (list.size() - 1) - j; i++) {  
            if (cmp.compare(list.get(i), list.get(i + 1)) > 0) {  
                swap(list, i, i + 1); // Tausche Elemente.  
            }  
        }  
    }  
}
```

...

Hilfsfunktionen

...

```
/**  
 * Hilfsprozedur zum Vertauschen zweier Listen-Elemente.  
 *  
 * @param list List mit Elementen, die vertauscht werden sollen.  
 * @param idx1 1. Element.  
 * @param idx2 2. Element.  
 */  
private void swap(List<E> list, int idx1, int idx2) {  
    E tmp = list.get(idx1);  
    list.set(idx1, list.get(idx2));  
    list.set(idx2, tmp);  
}  
...
```

Protokoll von Sortieralgorithmen

Im Folgenden werden bei einem Ablauf eines Sortieralgoritmus nicht mehr alle Schritte (v. a Variablen) protokolliert, sondern – wenn nicht anders angegeben – nur noch die aktuellen Werte des zu sortierenden Arrays.

Protokoll für BubbleSort (Teil 1)

// Durchlauf bis Position 9

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	11	5
0	1	2	3	4	5	6	7	8	9
4	9	5	1	6	7	8	9	11	5
0	1	2	3	4	5	6	7	8	9
4	5	9	1	6	7	8	9	11	5
0	1	2	3	4	5	6	7	8	9
4	5	1	9	6	7	8	9	11	5
0	1	2	3	4	5	6	7	8	9
4	5	1	6	9	7	8	9	11	5
0	1	2	3	4	5	6	7	8	9
4	5	1	6	7	9	8	9	11	5
0	1	2	3	4	5	6	7	8	9
4	5	1	6	7	8	9	9	11	5
0	1	2	3	4	5	6	7	8	9
4	5	1	6	7	8	9	9	11	5

Protokoll für BubbleSort (Teil 2)

0	1	2	3	4	5	6	7	8	9
4	5	1	6	7	8	9	9	<u>11</u>	5

// Durchlauf bis Position 8

0	1	2	3	4	5	6	7	8	9
4	5	1	6	7	8	9	9	5	11

0	1	2	3	4	5	6	7	8	9
4	5	1	<u>6</u>	7	8	9	9	5	11

0	1	2	3	4	5	6	7	8	9
4	1	<u>5</u>	<u>6</u>	7	8	9	9	5	11

0	1	2	3	4	5	6	7	8	9
4	1	<u>5</u>	<u>6</u>	<u>7</u>	8	9	9	5	11

0	1	2	3	4	5	6	7	8	9
4	1	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	9	9	5	11

0	1	2	3	4	5	6	7	8	9
4	1	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	9	5	11

0	1	2	3	4	5	6	7	8	9
4	1	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>9</u>	5	11

Protokoll für BubbleSort (Teil 3)

0	1	2	3	4	5	6	7	8	9
4	1	5	6	7	8	9	9	5	11

// Durchlauf bis Position 7

0	1	2	3	4	5	6	7	8	9
4	1	5	6	7	8	9	5	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	5	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	5	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	5	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	5	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	5	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	5	9	11

Protokoll für BubbleSort (Teil 4)

// Durchlauf bis Position 6

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	5	9	9	11
0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	5	9	9	11
0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	5	9	9	11
0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	5	9	9	11
0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	5	9	9	11
0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	5	9	9	11

// Durchlauf bis Position 5

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	5	8	9	9	11

Protokoll für BubbleSort (Teil 5)

0	1	2	3	4	5	6	7	8	9
1	<u>4</u>	<u>5</u>	6	7	5	<u>8</u>	<u>9</u>	<u>9</u>	11
0	1	2	3	4	5	6	7	8	9
1	4	<u>5</u>	<u>6</u>	7	5	<u>8</u>	<u>9</u>	<u>9</u>	11
0	1	2	3	4	5	6	7	8	9
1	4	5	<u>6</u>	<u>7</u>	5	<u>8</u>	<u>9</u>	<u>9</u>	11
0	1	2	3	4	5	6	7	8	9
1	4	5	6	<u>7</u>	<u>5</u>	<u>8</u>	<u>9</u>	<u>9</u>	11

// Durchlauf bis Position 4

0	1	2	3	4	5	6	7	8	9
1	<u>4</u>	<u>5</u>	6	5	<u>7</u>	<u>8</u>	<u>9</u>	<u>9</u>	11
0	1	2	3	4	5	6	7	8	9
1	<u>4</u>	<u>5</u>	6	5	<u>7</u>	<u>8</u>	<u>9</u>	<u>9</u>	11
0	1	2	3	4	5	6	7	8	9
1	4	<u>5</u>	<u>6</u>	5	<u>7</u>	<u>8</u>	<u>9</u>	<u>9</u>	11
0	1	2	3	4	5	6	7	8	9
1	4	5	<u>6</u>	<u>5</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>9</u>	11

Protokoll für BubbleSort (Teil 6)

// Durchlauf bis Position 3

0	1	2	3	4	5	6	7	8	9
<u>1</u>	<u>4</u>	5	5	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9
1	<u>4</u>	<u>5</u>	5	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9
1	4	<u>5</u>	<u>5</u>	6	7	8	9	9	11

// Durchlauf bis Position 2

0	1	2	3	4	5	6	7	8	9
<u>1</u>	<u>4</u>	5	5	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9
1	<u>4</u>	<u>5</u>	5	6	7	8	9	9	11

// Durchlauf bis Position 1

0	1	2	3	4	5	6	7	8	9
<u>1</u>	<u>4</u>	5	5	6	7	8	9	9	11

// Fertig

Protokoll für BubbleSort (Teil 7)

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

Weitere Optimierung

Wird einem Durchlauf nichts mehr (paarweise) vertauscht, ist die Folge sortiert. Dies kann als in den Algorithmus eingebaut werden und bleibt Ihnen als Übung überlassen.

Absteigend sortieren

```
...
    // Erzeuge einen aufsteigenden Comparator für Strings:
    Comparator<Integer> iCmp = Comparator.naturalOrder();
    doDescSort(new BubbleSort<>(), getIntList(), iCmp);
...
/**
 * @return Eine unsortierte Liste von Zahlen.
 */
public static List<Integer> getIntList() {
    return Arrays.asList(9, 15, 2, 5, 10, 8, 9, 1);
}
...
public static <E> void doDescSort(Sort sortAlg, List<E> list,
                                    Comparator<E> cmp) {
    // Erzeuge absteigenden Comparator:
    Comparator<E> desc = cmp.reversed();

    System.out.println(sortAlg + " desc:");
    System.out.println("vorher: " + list.toString());
    sortAlg.sort(list, desc);
    System.out.println("nachher: " + list.toString());
}
...

```

Laufzeit-Aufwand (Variante 1): $O(n^2)$

Im BubbleSort gemäß Alg. 8 sollen

- ① die Anzahl der Vergleiche v und
- ② die Vertauschungen t gezählt werden.

Das Vergleichen ist eine dominante Anweisung, die unabhängig von den Elementen der Folge ist, so dass BubbleSort mit

$$f(n) = \frac{n \cdot (n - 1)}{2}$$

die Laufzeitkomplexität $O(n^2)$ hat. BubbleSort erhält die relative Ordnung der Elemente bei und ist somit stabil.

[10, 13, 15, 18, 18, 19, 27, 39, 41, 10, 46, 20, 43, 55, 9, 17, 50, 60, 61, 63, 25, 70, 26, 75, 58, 22, 82, 37, 47, 52...]

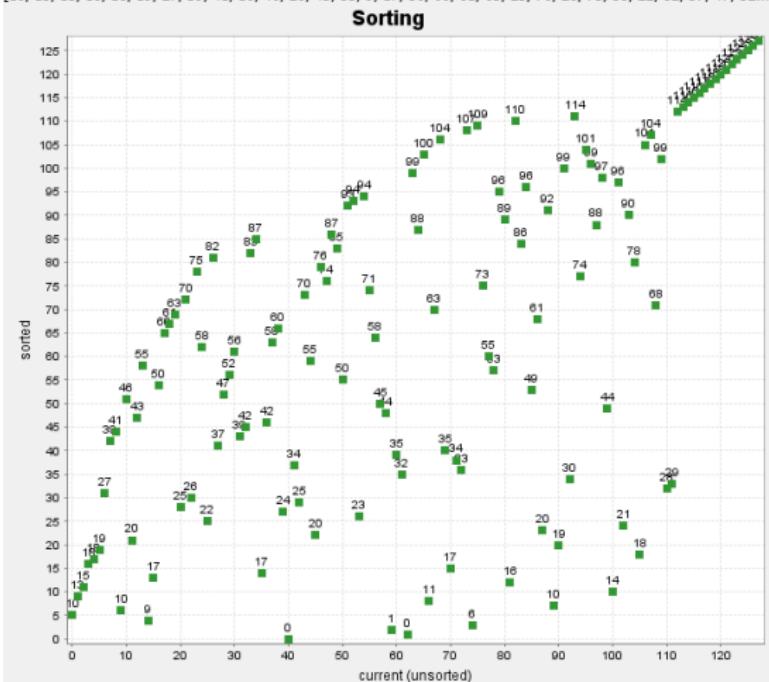


Abbildung 88: Visualisierung von BubbleSort.

Abschnittsübersicht

15 Einfache Sortierverfahren

BubbleSort: Sortieren durch Austauschen

InsertionSort: Sortieren durch Einfügen

Algorithmus

Aufwand und Stabilität

Visualisierung

SelectionSort: Sortieren durch Austauschen

Algorithmusidee von InsertionSort

InsertionSort ist mit dem Einsortieren von Karten eines Stapels auf dem Tisch in die Karten auf der Hand vergleichbar. In einer Folge F wird das erste Element an Position 1 als sortiert angenommen.

Beginnend ab dem zweiten Element wird an der passenden Stelle in der Folge F dieses Element eingefügt. Muss ein Element an einer Stelle dazwischen geschoben werden, werden die rechts davon liegenden Elemente jeweils um eine Position nach rechts geschoben.

Maximum-Variante

In der Maximum-Variante kann auch mit dem am weitesten rechts stehenden Element anfangen und absteigend sortiert eingefügt werden.

InsertionSort im Pseudocode

Algorithmus 9 InsertionSort (Minimum-Variante) → InsertionSort.java

```
proc insertionSort(F)                                // Eingabe: zu sortierende Folge F der Länge n
    for i von 2 bis n do                            // Element an Pos. i füge ich ein.
        m := F[i];                                    // Merke einzufügendes Element.
        j := i;                                       // Beginne ab dieser Position und gehe rückwärts...
        while j > 1 und F[j - 1] > m do      // ... solange das linke Element
            größer ist.
            F[j] := F[j - 1];   // Kopiere linkes Element eine Position nach rechts.
            Ziehe eins von j ab;                  // Zähle rückwärts.
        od                                         // An Pos. j ist eine Lücke entstanden.
        F[j] := m;                                  // Gemerkt Element dort einfügen.
    od
corp
```

InsertionSort

...

```
/*
 * @param list Die zu sortierende Folge
 */
public void sort(List<E> list, Comparator<E> cmp) {
    for (int i = 1; i < list.size(); i++) {
        int j = i;
        E m = list.get(i); // Einzufügender Wert.
        // Verschiebe alle größeren Elemente nach rechts:
        while (j > 0 && cmp.compare(list.get(j - 1), m) > 0) {
            list.set(j, list.get(j - 1)); // Vorgänger auf Nachfolger kopiert.
            j--; // j zeigt auf das zu überschreibende Feld.
        }
        list.set(j, m); // Setze m auf das freie Feld.
    }
}
```

...

Protokoll für InsertionSort (Teil 1)

// Einfuegen an Position 1 (Element 4)

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	11	5

0	1	2	3	4	5	6	7	8	9
4	9	5	1	6	7	8	9	11	5

// Einfuegen an Position 2 (Element 5)

0	1	2	3	4	5	6	7	8	9
4	9	5	1	6	7	8	9	11	5

0	1	2	3	4	5	6	7	8	9
4	5	9	1	6	7	8	9	11	5

// Einfuegen an Position 3 (Element 1)

0	1	2	3	4	5	6	7	8	9
4	5	9	1	6	7	8	9	11	5

0	1	2	3	4	5	6	7	8	9
1	4	5	9	6	7	8	9	11	5

// Einfuegen an Position 4 (Element 6)

0	1	2	3	4	5	6	7	8	9
1	4	5	9	6	7	8	9	11	5

Protokoll für InsertionSort (Teil 2)

0	1	2	3	4	5	6	7	8	9
1	4	5	6	9	7	8	9	11	5

// Einfügen an Position 5 (Element 7)

0	1	2	3	4	5	6	7	8	9
1	4	5	6	9	7	8	9	11	5
0	1	2	3	4	5	6	7	8	9

// Einfügen an Position 6 (Element 8)

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	9	8	9	11	5
0	1	2	3	4	5	6	7	8	9

// Einfügen an Position 7 (Element 9)

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	9	11	5
0	1	2	3	4	5	6	7	8	9

// Einfügen an Position 8 (Element 11)

Protokoll für InsertionSort (Teil 3)

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	9	<u>11</u>	5

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	9	<u>11</u>	5

// Einfuegen an Position 9 (Element 5)

0	1	2	3	4	5	6	7	8	9
1	4	5	6	7	8	9	9	<u>11</u>	<u>5</u>

0	1	2	3	4	5	6	7	8	9
1	4	5	<u>5</u>	6	7	8	9	9	11

Laufzeit-Aufwand: $O(n^2)$

Es sollen zunächst drei Anweisungen gezählt werden:

- ① Anzahl der Einfügeaktionen e ($F[j] := m$)
- ② Anzahl der Vergleiche v (while $j > 1$ und $F[j - 1] > m$) und
- ③ Anzahl der Shift-Aktionen s ($F[j] := F[j - 1];$)

Die Anzahl der Vergleiche v ist eine dominierende Aktion, denn keine wird häufiger ausgeführt. Im schlechtesten Fall gilt:

$$f(n) = v = \frac{n \cdot (n - 1)}{2}$$

Also hat InsertionSort die Laufzeit-Komplexität von $O(n^2)$. Die Speicher-Komplexität ist $O(n)$.

Stabilität von InsertionSort

Das Verfahren ist stabil, keine Vertauschungen sondern nur Verschiebungen stattfinden. Dadurch werden die relative Reihenfolgen beibehalten.

```
[1, 5, 6, 6, 6, 7, 8, 10, 10, 10, 12, 17, 22, 33, 33, 41, 47, 48, 57, 58, 60, 62, 67, 68, 70, 79, 79, 82, 87, 98, 99, 10...
```

> **Sorting**

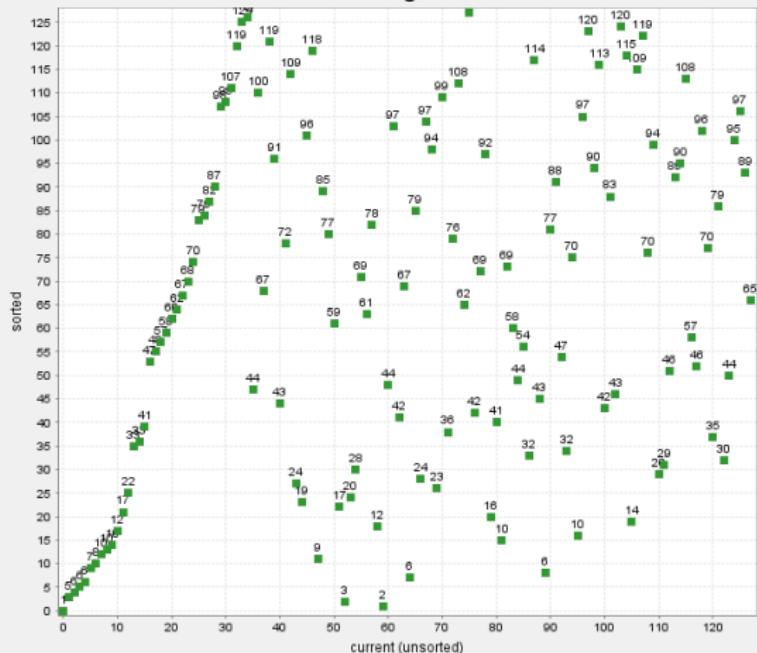


Abbildung 89: Visualisierung von InsertionSort.

Abschnittsübersicht

15 Einfache Sortierverfahren

BubbleSort: Sortieren durch Austauschen

InsertionSort: Sortieren durch Einfügen

SelectionSort: Sortieren durch Austauschen

Algorithmus

Aufwand und Stabilität

Visualisierung

Algorithmusidee von SelectionSort

Die Idee von SelectionSort ist, dass man im ersten Durchlauf das größte Element an die letzte Position bringt, indem das größte Element ans Ende getauscht wird. Jetzt ist an Position n das größte Element, und das gleiche Prinzip kann für die Restfolge von 1 bis $n - 1$ angewandt werden. Mit jedem Schritt wird die Restfolge kleiner, und es wird solange nach diesem Prinzip weiter verfahren, bis die Restfolge nur die ersten beiden Element umfasst. Hier muss ein letzter Tausch überprüft werden, danach ist die Folge sortiert.

Minimum-Variante

Eine Variante des SelectionSorts kann auch die kleinsten Element nach vorne bringen (Minimum-Variante).

SelectionSort im Pseudocode

Algorithmus 10 SelectionSort (Maximum-Variante) → SelectionSort.java

```
proc SelectionSort(F) // Folge F der Länge n
    for p von n bis 2 rückwärts do // Laufe rückwärts bis zum zweiten Element
        Ermittle die Indexposition max des größten Elements im
        Bereich 1 bis p; // Entspricht in etwa MaxSearch().
        Tausche (größtes) Element an Position max mit dem an
        Position p;
    od
corp
```

Maximum-Suche im Pseudocode

Diese Maximumssuche basiert auf Alg. ??, welcher um einen Suchbereich i_1 bis i_2 erweitert wird.

Algorithmus 11 Suche der Indexposition des maximalen Elements im Intervall

```
funct MaxSearch( $F, i_1, i_2$ ) // Folge  $F$  der Länge  $n$ , Bereichsgrenzen  $i_1, i_2$ 
    Vermute vorläufiges Maximum an Indexposition  $m := i_1$ ;
    for  $i$  von  $i_1 + 1$  bis  $i_2$  do // Durchlaufe die restlichen Elemente.
        if  $F[i] > F[m]$  then // Neues Maximum gefunden?
             $m := i$ ; // Indexposition des aktuellen Maximums.
        fi
    od
    return  $m$ ;
```

tcnuf

SelectionSort in Java

```
...
public void sort(List<E> list, Comparator<E> cmp) {
    // Durchlaufe Positionen n-1 bis 1 rückwärts...
    for (int p = list.size() - 1; p > 0; p--) {
        // Suche größtes Element im Bereich:
        int max = maxSearch(list, 0, p, cmp);
        swap(list, p, max); // Tausche beide Elemente.
    }
}
...
}
```

Die swap-Funktion wurde bereits beim BubbleSort genutzt.

Maximumsuche in Java

```
...
private int maxSearch(List<E> list, int i1, int i2,
                      Comparator<E> cmp) {
    int m = i1; // Bis jetzt "größtes" Element.
    for (int i = i1 + 1; i <= i2; i++) {
        if (cmp.compare(list.get(i), list.get(m)) > 0) {
            m = i; // Größeres Element gefunden.
        }
    }
    return m;
...
}
```

Protokoll für SelectionSort (Teil 1)

-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>9</td><td>4</td><td>5</td><td>1</td><td>6</td><td>7</td><td>8</td><td>9</td><td><u>11</u></td><td>5</td></tr></table>	0	1	2	3	4	5	6	7	8	9	9	4	5	1	6	7	8	9	<u>11</u>	5
0	1	2	3	4	5	6	7	8	9													
9	4	5	1	6	7	8	9	<u>11</u>	5													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>9</td><td>4</td><td>5</td><td>1</td><td>6</td><td>7</td><td>8</td><td>9</td><td>5</td><td><u>11</u></td></tr></table>	0	1	2	3	4	5	6	7	8	9	9	4	5	1	6	7	8	9	5	<u>11</u>
0	1	2	3	4	5	6	7	8	9													
9	4	5	1	6	7	8	9	5	<u>11</u>													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>5</td><td>4</td><td>5</td><td>1</td><td>6</td><td>7</td><td>8</td><td><u>9</u></td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	5	4	5	1	6	7	8	<u>9</u>	9	11
0	1	2	3	4	5	6	7	8	9													
5	4	5	1	6	7	8	<u>9</u>	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>5</td><td>4</td><td>5</td><td>1</td><td>6</td><td>7</td><td><u>8</u></td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	5	4	5	1	6	7	<u>8</u>	9	9	11
0	1	2	3	4	5	6	7	8	9													
5	4	5	1	6	7	<u>8</u>	9	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>5</td><td>4</td><td>5</td><td>1</td><td><u>6</u></td><td>7</td><td>8</td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	5	4	5	1	<u>6</u>	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9													
5	4	5	1	<u>6</u>	7	8	9	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>5</td><td>4</td><td>5</td><td>1</td><td><u>6</u></td><td>7</td><td>8</td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	5	4	5	1	<u>6</u>	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9													
5	4	5	1	<u>6</u>	7	8	9	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>5</td><td>4</td><td>5</td><td><u>1</u></td><td>6</td><td>7</td><td>8</td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	5	4	5	<u>1</u>	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9													
5	4	5	<u>1</u>	6	7	8	9	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>1</td><td>4</td><td><u>5</u></td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	1	4	<u>5</u>	5	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9													
1	4	<u>5</u>	5	6	7	8	9	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>1</td><td><u>4</u></td><td>5</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	1	<u>4</u>	5	5	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9													
1	<u>4</u>	5	5	6	7	8	9	9	11													
-	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>1</td><td>4</td><td>5</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>9</td><td>11</td></tr></table>	0	1	2	3	4	5	6	7	8	9	1	4	5	5	6	7	8	9	9	11
0	1	2	3	4	5	6	7	8	9													
1	4	5	5	6	7	8	9	9	11													

Rot unterstrichen

sind 1) das Element an der Position, das mit dem 2) größten Element ausgetauscht wird.
Grün markiert ist der bereits sortierte Array-Bereich.

Vollständiges Protokoll für SelectionSort (Teil 1)

main() ↴

_ list =

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	11	5

_ selectionSort(list) ↴

_ _ p = 9

_ _ _ maxSearch(list, 0, 9) ↴

_ _ _ _ m = 0

_ _ _ _ i = 1

_ _ _ _ i = 2

_ _ _ _ i = 3

_ _ _ _ i = 4

_ _ _ _ i = 5

_ _ _ _ i = 6

_ _ _ _ i = 7

_ _ _ _ i = 8

_ _ _ _ m = 8

_ _ _ _ i = 9

_ _ _ maxSearch(list, 0, 9) = 8 ↴

_ _ max = 8

_ _ list =

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	11	5

_ _ swap(list, 9, 8) ↴

_ _ _ tmp = 5

_ _ _ list =

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	11	11

_ _ _ list =

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	5	11

_ _ swap(list, 9, 8) ↴

_ _ p = 8

_ _ _ maxSearch(list, 0, 8) ↴

_ _ _ m = 0

Vollständiges Protokoll für SelectionSort (Teil 2)

```
- - - - i = 1
- - - - i = 2
- - - - i = 3
- - - - i = 4
- - - - i = 5
- - - - i = 6
- - - - i = 7
- - - - i = 8
- - maxSearch(list, 0, 8) = 0 ↘
- - max = 0
```

list =	0	1	2	3	4	5	6	7	8	9	11
	9	4	5	1	6	7	8	9	5	11	

```
- - swap(list, 8, 0) ↘
- - - tmp = 5
```

list =	0	1	2	3	4	5	6	7	8	9	11
	9	4	5	1	6	7	8	9	9	11	

```
- - list =
- - - list =
- - swap(list, 8, 0) ↘
- - p = 7
- - maxSearch(list, 0, 7) ↘
- - - m = 0
```

```
- - - i = 1
- - - i = 2
- - - i = 3
- - - i = 4
- - - m = 4
- - - i = 5
- - - m = 5
- - - i = 6
```

Vollständiges Protokoll für SelectionSort (Teil 3)

- - - $m = 6$

- - - $i = 7$

- - - $m = 7$

- - $\text{maxSearch}(\text{list}, 0, 7) = 7 \checkmark$

- - $\text{max} = 7$

- - $\text{list} =$

0	1	2	3	4	5	6	7	8	9
5	4	5	1	6	7	8	9	9	11

- - $\text{swap}(\text{list}, 7, 7) \checkmark$

- - - $\text{tmp} = 9$

- - - $\text{list} =$

0	1	2	3	4	5	6	7	8	9
5	4	5	1	6	7	8	9	9	11

- - - $\text{list} =$

0	1	2	3	4	5	6	7	8	9
5	4	5	1	6	7	8	9	9	11

- - $\text{swap}(\text{list}, 7, 7) \checkmark$

- - $p = 6$

- - $\text{maxSearch}(\text{list}, 0, 6) \checkmark$

- - - $m = 0$

- - - $i = 1$

- - - $i = 2$

- - - $i = 3$

- - - $i = 4$

- - - $m = 4$

- - - $i = 5$

- - - $m = 5$

- - - $i = 6$

- - - $m = 6$

- - $\text{maxSearch}(\text{list}, 0, 6) = 6 \checkmark$

- - $\text{max} = 6$

- - $\text{list} =$

0	1	2	3	4	5	6	7	8	9
5	4	5	1	6	7	8	9	9	11

Vollständiges Protokoll für SelectionSort (Teil 4)

```
- - - swap(list, 6, 6) ↴
- - - - tmp = 8
- - - list =


|   |   |   |   |   |   |   |   |   |    |   |
|---|---|---|---|---|---|---|---|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 9 |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |   |


- - - list =


|   |   |   |   |   |   |   |   |   |    |   |
|---|---|---|---|---|---|---|---|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 9 |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |   |


- - swap(list, 6, 6) ↵
- - p = 5
- - maxSearch(list, 0, 5) ↴
- - - m = 0
- - - i = 1
- - - i = 2
- - - i = 3
- - - i = 4
- - - m = 4
- - - i = 5
- - - m = 5
- - maxSearch(list, 0, 5) = 5 ↵
- - max = 5
- - list =


|   |   |   |   |   |   |   |   |   |    |   |
|---|---|---|---|---|---|---|---|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 9 |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |   |


- - swap(list, 5, 5) ↴
- - - tmp = 7
- - - list =


|   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 11 |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |    |


- - - list =


|   |   |   |   |   |   |   |   |   |    |   |
|---|---|---|---|---|---|---|---|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 9 |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |   |


- - swap(list, 5, 5) ↵
- - p = 4
```

Vollständiges Protokoll für SelectionSort (Teil 5)

```
- - maxSearch(list, 0, 4) ↴  
- - - m = 0  
- - - i = 1  
- - - i = 2  
- - - i = 3  
- - - i = 4  
- - - m = 4  
- - maxSearch(list, 0, 4) = 4 ↵  
- - max = 4  
- - list = 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |

  
- - swap(list, 4, 4) ↴  
- - - tmp = 6  
- - - list = 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |

  
- - - list = 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |

  
- - swap(list, 4, 4) ↵  
- - p = 3  
- - maxSearch(list, 0, 3) ↴  
- - - m = 0  
- - - i = 1  
- - - i = 2  
- - - i = 3  
- - maxSearch(list, 0, 3) = 0 ↵  
- - max = 0  
- - list = 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| 5 | 4 | 5 | 1 | 6 | 7 | 8 | 9 | 9 | 11 |

  
- - swap(list, 3, 0) ↴  
- - - tmp = 1
```

Vollständiges Protokoll für SelectionSort (Teil 6)

```
-- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- swap(list, 3, 0) ↘
-- p = 2
-- maxSearch(list, 0, 2) ↘
-- m = 0
-- i = 1
-- m = 1
-- i = 2
-- m = 2
-- maxSearch(list, 0, 2) = 2 ↗
-- max = 2
-- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- swap(list, 2, 2) ↘
-- tmp = 5
-- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- swap(list, 2, 2) ↗
-- p = 1
-- maxSearch(list, 0, 1) ↘
-- m = 0
-- i = 1
-- m = 1
-- maxSearch(list, 0, 1) = 1 ↗
-- max = 1
```

Vollständiges Protokoll für SelectionSort (Teil 7)

```
-- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- swap(list, 1, 1) ↴
-- -- tmp = 4
-- -- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- -- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- -- swap(list, 1, 1) ↵
-- -- list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
-- selectionSort(list) ↵
main() ↵
```

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

Laufzeit-Aufwand: $O(n^2)$

Es sollen zunächst drei Anweisungen gezählt werden:

- ① Anzahl der Tauschaktionen t (Tausche (größtes) Element an Position max mit dem an Position p;)
- ② Anzahl der Vergleiche v in der Funktion MaxSearch (if $F[i] > F[m]$ then) und
- ③ Anzahl der Zuweisungen z ($m := i;$) in dieser Funktion.

Die Anzahl der Vergleiche v ist eine dominierende Aktion, denn keine wird häufiger ausgeführt. In allen drei Fällen (besten, mittlerer, schlechtesten) gilt:

$$f(n) = v = \frac{n \cdot (n - 1)}{2}$$

Damit hat SelectionSort die Laufzeit-Komplexität von $O(n^2)$. Die Speicher-Komplexität ist $O(n)$.

Stabilität von SelectionSort

Das Verfahren ist instabil wegen der Vertauschungen, dadurch werden die relative Reihenfolgen nicht beibehalten.

[20, 79, 30, 43, 70, 36, 49, 37, 58, 37, 41, 103, 77, 84, 86, 37, 101, 24, 88, 51, 16, 8, 87, 20, 103, 81, 11, 25, 33, ...]

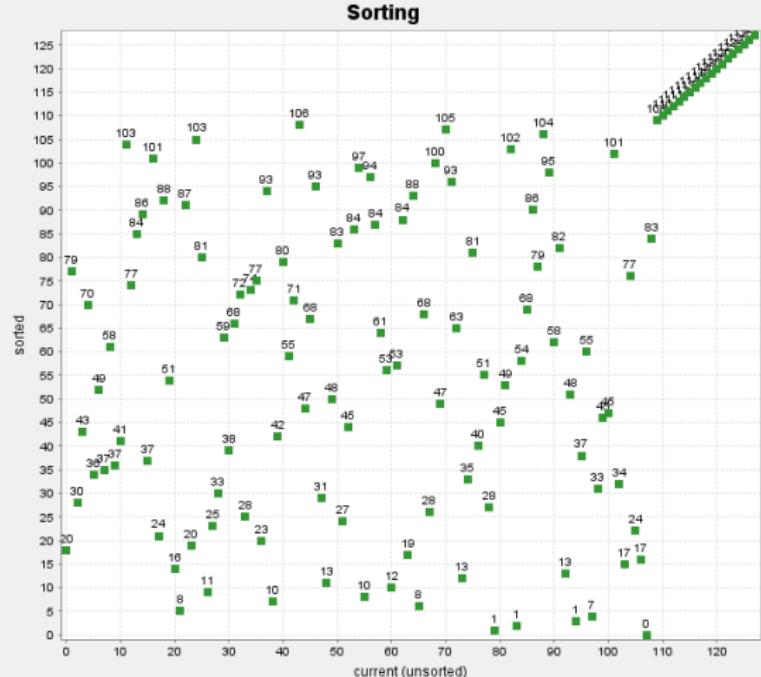


Abbildung 90: Visualisierung von SelectionSort.

Kapitelübersicht

16 Fortgeschrittene Sortierverfahren

MergeSort: Sortieren durch Zusammenfügen

QuickSort: Sortieren durch Zerlegen

Zusammenfassung und Ausblick

Abschnittsübersicht

16 Fortgeschrittene Sortierverfahren

MergeSort: Sortieren durch Zusammenfügen

Direkter MergeSort

Natürlicher MergeSort

Rekursiver In-Memory MergeSort

Aufwand und Stabilität

Visualisierung

QuickSort: Sortieren durch Zerlegen

Zusammenfassung und Ausblick

MergeSort

MergeSort ist externes Verfahren. Alle Varianten basieren auf den beiden Schritten Split (Aufteilen) und Merge (Zusammenführen):

Split Die Folge F wird in Teile zerlegt, die jeweils in den Hauptspeicher passen und daher getrennt voneinander mit internen Verfahren sortiert werden können. Diese sortierten Teilfolgen werden wieder in Dateien F_i ($i = 2, 3, \dots$) ausgelagert.

Merge Anschließend werden die Teilfolgen gleichzeitig eingelesen und zusammengeführt, indem jeweils das **kleinste** Element aller Teilfolgen gelesen und in die ursprüngliche Folge F geschrieben wird. F wird dabei überschrieben.

Keine Dateien

Wir werden das Verfahren mit einem Array anstelle von Dateien simulieren.

MergeSort im Pseudocode

Algorithmus 12 Direkter 2-Wege MergeSort

```
proc DirectMergeSort( $F$ ) // Folge  $F$  der Länge  $n$ 
    Setze aktuelle Lauflänge auf  $r := 1$ ;
    while  $r < n$  do // Solange ist  $F$  möglicherweise noch unsortiert.
        Schreibe von der Folge  $F$  jeweils einen Lauf alternierend auf  $F_1$  und  $F_2$  bis alle
        Läufe von  $F$  nach  $F_1$  und  $F_2$  kopiert sind;
        Setze  $F$  auf die Ausgangsposition zurück (Überschreiben);
        for 1. bis letztes Laufpaar do // Letzes Laufpaar muss nicht vollständig besetzt sein.
            while kein Lauf abgearbeitet ist do
                Entferne das kleinere Anfangselement aus Lauf auf  $F_1$  bzw.  $F_2$  und füge dieses
                an  $F$  an;
            od
            Füge den Rest des nichtleeren Laufs an  $F$  an;
        od
        Verdopple Lauflänge für nächsten Durchlauf:  $r := r \cdot 2$ ;
    od
corp
```

Direkter 2-Wege MergeSort in Java

```
...
public void sort(List<E> list, Comparator<E> cmp) {
    if (list.size() <= 1) {
        return; // Höchstens ein Element, es gibt nichts zu sortieren.
    }
    List<E> f1 = new ArrayList<>(); // Erzeuge die Bänder F1
    List<E> f2 = new ArrayList<>(); // und F2.

    int r = 1; // Lauflänge zu Beginn.
    // Solange die Lauflänge kleiner als n ist, ist die Folge
    // möglicherweise noch unsortiert:
    while (r < list.size()) {
        split(list, f1, f2, r); // Teile list in F1 und F2 auf.
        merge(f1, f2, list, r, cmp); // Füge F1 und F2 in F zusammen.
        r = r * 2; // Verdoppele Lauflänge für nächsten Durchlauf.
    }
}
...

```

```
...
private void split(List<E> list, List<E> f1, List<E> f2, int r) {
    f1.clear(); // Zunächst werden die Bänder gelöscht.
    f2.clear();

    // true für Band F1 bzw. false für F2:
    boolean firstTape = false; // zunächst F2, wird sofort gewechselt.
    int i = 0; // Zähler zum Bandwechseln.

    for (E e : list) { // Durchlaufe das Band F...
        if (i++ % r == 0) { // Laufende erreicht?
            firstTape = !firstTape; // Ja, d.h. wir wechseln das Band.
        }
        // Füge zu lesendes Element auf F an das aktuelle Band an:
        if (firstTape) {
            f1.add(e);
        } else {
            f2.add(e);
        }
    }
}
...

```

```
...
private void merge(List<E> f1, List<E> f2, List<E> list,
                  int r, Comparator<E> cmp) {
    // Anzahl Laufpaare (aufgerundet):
    int noRunPairs = (int) Math.ceil(list.size() / (2.0 * r));
    list.clear(); // Originalband wieder löschen.

    for (int rp = 1; rp <= noRunPairs; rp++) { // Alle Laufpaare...
        int remaining1 = Math.max(0, f1.size() - r);
        int remaining2 = Math.max(0, f2.size() - r);
        while (f1.size() > remaining1 && f2.size() > remaining2) {
            if (cmp.compare(f1.get(0), f2.get(0)) <= 0) {
                list.add(f1.remove(0)); // Element auf F1 kleiner gleich -> angefügt.
            } else {
                list.add(f2.remove(0)); // Element auf F2 größer -> angefügt.
            }
        }
        // Füge den Rest eines etwaigen nicht-leeren Laufs an F an:
        while (f1.size() > remaining1) { // Band F1
            list.add(f1.remove(0));
        }
        while (f2.size() > remaining2) { // Band F2
            list.add(f2.remove(0));
        }
    }
}
...
```

Protokoll für direkten 2-Wege-MergeSort (Teil 1)

$f =$	0	1	2	3	4	5	6	7	8	9
	9	4	5	1	6	7	8	9	11	5

// Split (Lauflaenge 1):

$f_1 =$	0	1	2	3	4
	9	5	6	8	11

$f_2 =$	0	1	2	3	4
	4	1	7	9	5

// Merge (Lauflaenge ist nun 2):

$f =$	0	1	2	3	4	5	6	7	8	9
	4	9	1	5	6	7	8	9	5	11

// Split (Lauflaenge 2):

$f_1 =$	0	1	2	3	4	5
	4	9	6	7	5	11

$f_2 =$	0	1	2	3
	1	5	8	9

// Merge (Lauflaenge ist nun 4):

$f =$	0	1	2	3	4	5	6	7	8	9
	1	4	5	9	6	7	8	9	5	11

Protokoll für direkten 2-Wege-MergeSort (Teil 2)

// Split (Lauflaenge 4):

$f_1 =$	0	1	2	3	4	5
	1	4	5	9	5	11

$f_2 =$	0	1	2	3
	6	7	8	9

// Merge (Lauflaenge ist nun 8):

$f =$	0	1	2	3	4	5	6	7	8	9
	1	4	5	6	7	8	9	9	5	11

// Split (Lauflaenge 8):

$f_1 =$	0	1	2	3	4	5	6	7
	1	4	5	6	7	8	9	9

$f_2 =$	0	1
	5	11

// Merge (Lauflaenge ist nun 16):

$f =$	0	1	2	3	4	5	6	7	8	9
	1	4	5	5	6	7	8	9	9	11

Eigenschaften des direkten MergeSorts

Werden Folgen beim MergeSort in zwei Teilstufen zerlegt, nennt man das Verfahren **2-Wege-MergeSort**. Prinzipiell kann ein MergeSort auch mit $m > 2$ Dateien bzw. Folgen realisiert werden, er heißt dann allgemein m -Wege-MergeSort.

Das eben vorgestellte Verfahren wird als reines oder **direktes** 2-Wege-MergeSort (engl. straight 2-way mergesort) bezeichnet, da die Lauflänge immer mit jedem Durchlauf verdoppelt wird. Diese Variante ist **ineffizient**, da immer mit ein-elementigen Teilstufen begonnen und keine Vorsortierung berücksichtigt wird. Ebenso sind die Lauflängen nicht variabel.

Definition: Maximaler Lauf

Ein maximaler Lauf ist eine aufsteigende oder gleichbleibende Schlüsselfolge, die von einem größeren Element links im Vergleich zum ersten Element oder dem linken Rand bzw. von einem kleineren Element rechts im Vergleich zum letzten Element oder dem rechten Rand eingeschlossen wird. Sei $f_i = f[i]$ aus F :

li. Rand $f_1 \leq f_2 \leq \dots \leq f_m > f_{m+1}$

Mitte $f_{i-1} > f_i \leq f_{i+1} \leq \dots \leq f_{i+m-1} > f_{i+m}$

re. Rand $f_{n-m-1} > f_{n-m} \leq f_{n-1} \leq \dots \leq f_n$

Beispiel

$F = [4], [3, 5, 5, 7], [2, 4], [1, 6, 8], [2]$. Läufe sind durch Klammern eingeschlossen.

Natürlicher MergeSort im Pseudocode

Algorithmus 13 Natürlicher MergeSort

```
proc NatMergeSort( $F$ ) // Folge  $F$  der Länge  $n$ 
    do
        Teile  $F$  alternierend in Folgen  $F_1$  und  $F_2$  mit jeweils maximalen
        Lauflängen auf;
        if beide Folgen  $F_1$  und  $F_2$  benutzt then
            Führe  $F_1$  und  $F_2$  zusammen mit NatMerge( $F, F_1, F_2$ );
        fi
        while es mehr als einen Lauf in  $F$  gibt od
    corp
```

Algorithmus 14 Merge-Funktion des natürlichen MergeSorts

```
proc NatMerge( $F, F_1, F_2$ )
```

Setze F auf die Ausgangsposition zurück (leere Folge);

do

if erstes Element in $F_1 <$ erstes Element in F_2 then

Entferne erstes Element aus F_1 und füge es an F an;

if Lauf F_1 ist zu Ende then

Füge restlichen Lauf F_2 an F an;

fi

else

// Dieser Teil ist symmetrisch zum obigen:

Entferne erstes Element aus F_2 und füge es an F an;

if Lauf F_2 ist zu Ende then

Füge restlichen Lauf F_1 an F an;

fi

fi

while F_1 nicht leer und F_2 nicht leer od

Füge restlichen Elementen einer möglicherweise nicht-leeren Folge (F_1 oder F_2) an F an;

corp

Protokoll für NaturalMergeSort (Teil 1)

$f =$

0	1	2	3	4	5	6	7	8	9
9	4	5	1	6	7	8	9	11	5

// Split:

$f_1 =$

0	1	2	3	4	5	6
9	1	6	7	8	9	11

$f_2 =$

0	1	2
4	5	5

// Merge (ergibt 2 Lauf/Laeufe):

$f =$

0	1	2	3	4	5	6	7	8	9
4	5	5	9	1	6	7	8	9	11

// Split:

$f_1 =$

0	1	2	3
4	5	5	9

$f_2 =$

0	1	2	3	4	5
1	6	7	8	9	11

// Merge (ergibt 1 Lauf/Laeufe):

$f =$

0	1	2	3	4	5	6	7	8	9
1	4	5	5	6	7	8	9	9	11

Rekursiver In-Memory MergeSort

Die Idee des reinen MergeSort-Algorithmus lässt sich auch gut für ein internes Sortierverfahren übertragen, indem alle Elemente in einem Array gehalten werden. Dazu wird das Array zunächst rekursiv in gleichgroße Hälften (Läufe) unterteilt, solange bis die Intervalllänge 1 ist. Danach werden die benachbarten Läufe sortiert zusammengefügt. Dies entspricht dem Merge-Schritt (Zusammenführen). Die so entstandenen teilsortierten Läufe werden nach dem gleichen Prinzip zu längeren Teilfolgen zusammengefügt, bis schließlich das gesamte Array zu einem Lauf zusammengeführt wurde.

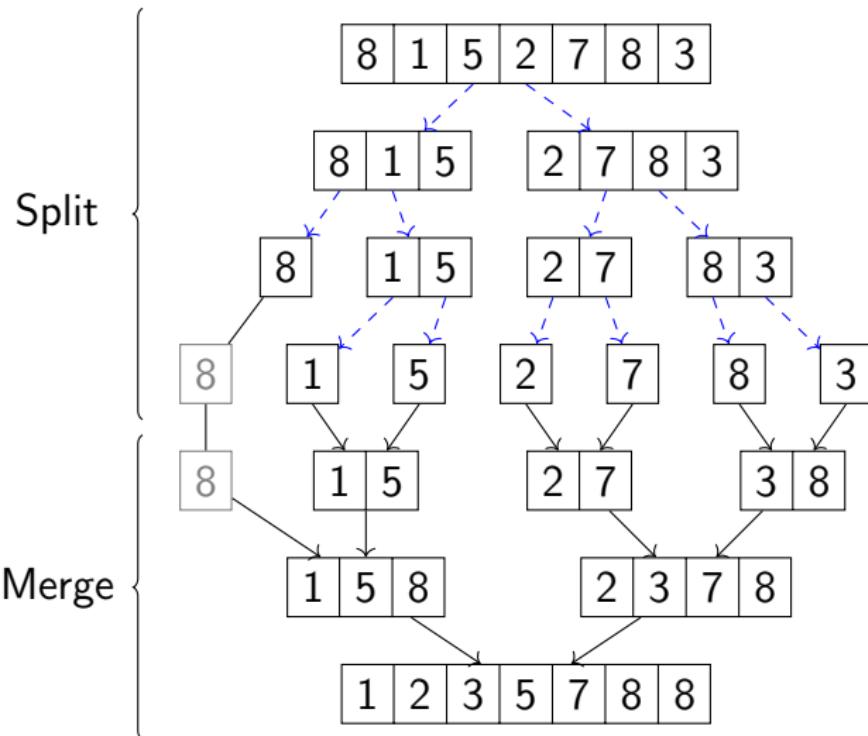


Abbildung 91: Ablauf des rekursiven In-Memory MergeSorts.

Laufzeit von MergeSort

Im Folgenden untersuchen wir die Laufzeit vom reinen 2-Wege MergeSort, und wir unterstellen, dass die Anzahl der Elemente eine Zweierpotenz ist ($n = 2^i$ für $i = 0, 1, 2, \dots$). Wir haben gesehen, dass die Anzahl der Durchläufe (das Verteilen und Mischen) konstant ist. Unter der Annahme der Zweierpotenz ist dies:

$$d = \log_2 n$$

Laufzeit

Pro Durchlauf werden alle n Elemente von F gleichmäßig auf die beiden Hilfsbänder kopiert. Die Anzahl der Kopieraktionen beim Splitten ist insgesamt:

$$k_s = n \cdot d$$

Laufzeit

Pro Durchlauf werden alle n Elemente von F gleichmäßig auf die beiden Hilfsbänder kopiert. Die Anzahl der Kopieraktionen beim Splitten ist insgesamt:

$$k_s = n \cdot d$$

Beim Zurückschreiben (Mergen) auf F gibt es ebenfalls genau n Kopieroperationen pro Durchlauf, also insgesamt:

$$k_m = n \cdot d$$

Laufzeit

Für ein Laufpaar mit Läufen der Länge r kann es höchstes $2r - 1$ Vergleiche geben. Bei einer Lauflänge r gibt es $\frac{n/2}{r}$ Laufpaare. Für die Gesamt-Vergleiche gilt mit $r = 2^{i-1}$:

$$v = \sum_{i=1}^d (2r - 1) \cdot \frac{n/2}{r} = \sum_{i=1}^d \underbrace{\frac{2r - 1}{2r}}_{< 1} n < \sum_{i=1}^d n = n \cdot d$$

Laufzeit

Für ein Laufpaar mit Läufen der Länge r kann es höchstes $2r - 1$ Vergleiche geben. Bei einer Lauflänge r gibt es $\frac{n/2}{r}$ Laufpaare. Für die Gesamt-Vergleiche gilt mit $r = 2^{i-1}$:

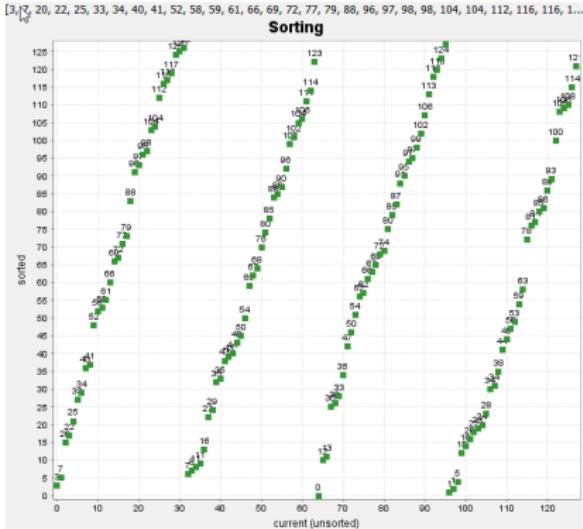
$$v = \sum_{i=1}^d (2r - 1) \cdot \frac{n/2}{r} = \sum_{i=1}^d \underbrace{\frac{2r - 1}{2r}}_{< 1} n < \sum_{i=1}^d n = n \cdot d$$

Somit ist der Gesamtaufwand in allen Fällen (beste, schlechteste, durchschnittlicher):

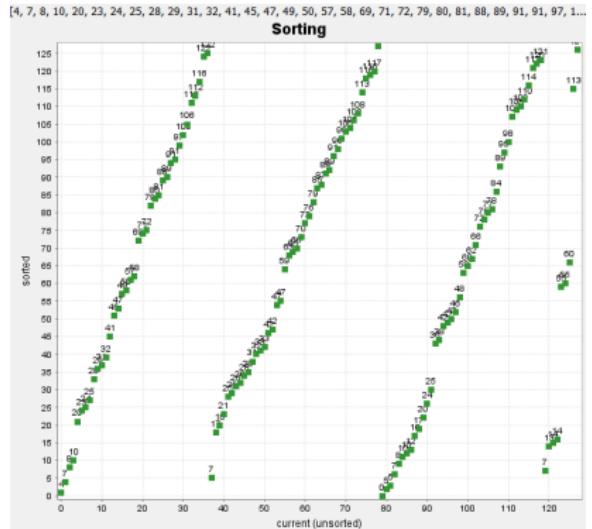
$$f(n) = k_s + k_m + v < nd + nd + nd = 3n \log_2 n \in O(n \cdot \log n)$$

Stabilität

Der reine 2-Wege MergeSort ist stabil. Beim natürlichen MergeSort kann es vorkommen, dass die relative Reihenfolge gleicher Elemente nicht beibehalten wird.



(a) Direkter 2-Wege MergeSort



(b) Natürlicher MergeSort

Abbildung 92: Visualisierung der beiden MergeSort-Varianten.

Abschnittsübersicht

16 Fortgeschrittene Sortierverfahren

MergeSort: Sortieren durch Zusammenfügen

QuickSort: Sortieren durch Zerlegen

QuickSort mit einfacher Zerlegung durch Kopieren des Arrays

QuickSort ohne Hilfsarray

Nachteile

Aufwand und Stabilität

Visualisierung

Zusammenfassung und Ausblick

Idee von QuickSort

QuickSort ist ein so genannter Divide-And-Conquer-Algorithmus.

- Zerlegung der zu sortierenden Folge F in zwei Teile bzw. Teilfolgen mit der Eigenschaft:
- Alle Elemente der einen Folge sind kleiner als ein Referenzelement – das sog. *Pivot-Element* – und alle Elemente der anderen Folge sind größer (oder gleich) als das Referenzelement.
- Das Pivot-Element kann beliebig gewählt werden, sollte aber in der Folge vorkommen.
- Für die Sortierung der Teilfolgen wird dasselbe Prinzip rekursiv angewendet.
- Das Verfahren endet (Rekursionsende), wenn die zu sortierende Teilfolge die Länge 1 hat oder leer ist. Dann ist die gesamte Folge sortiert.

QuickSort allgemein

Algorithmus 15 Grundlegende Idee von QuickSort

```
proc QuickSort( $F$ )
  if (Teil-)Folge  $F$  hat mehr als ein Element then
    Zerlege die Folge in die Folgen (Partitionen)  $F_1$  und  $F_2$  ;
    QuickSort( $F_1$ );                                // Wende das Verfahren ...
    QuickSort( $F_2$ );                                // rekursiv auf die Teilstufen an.
  fi
  corp
```

Die Teilstufen werden oft als (verkettete) Listen dargestellt. Wir simulieren dies durch Index-Intervalle i_1 bis i_2 in einem Array.

Zerlegung (Partitionierung)

Es gibt mehrere Verfahren für die Zerlegung einer Folge F in zwei oder drei Teile (F_1, F_2, F_3). Hier gilt mit p als Pivotwert:

Zwei Teilfolgen: Es gilt $\forall e \in F_1 : e \leq p$ und $\forall e \in F_2 : e > p$

Drei Teilfolgen: Es gilt $\forall e \in F_1 : e < p$, $\forall e \in F_2 : e > p$ und die dritte Folge F_3 enthält die Pivot-Elemente.

Bei array-basierten Verfahren kann es außerdem sein, dass die Partitionierung das oder die Pivot-Element(e) bereits an die richtige Stelle im Array einsortiert.

Einfache Zerlegung durch Kopieren des Arrays

Algorithmus 16 QuickSort mit simpler Zerlegung

```
proc QuickSortSimple( $F$ ,  $i_1$ ,  $i_2$ ) // Teifolge im Intervall  $i_1$  bis  $i_2$ .
    if  $i_1 < i_2$  then // Intervall mind. ein Element?
        ( $i, j$ ) := PartitionSimple( $F$ ,  $i_1$ ,  $i_2$ );
        QuickSortSimple( $F$ ,  $i_1$ ,  $i - 1$ ); // Rekursion!
        QuickSortSimple( $F$ ,  $j + 1$ ,  $i_2$ );
    fi
corp
```

Bei PartitionSimple stehen die Pivot-Elemente im Bereich i bis j nach der Zerlegung bereits an der richtigen Stelle im Array und brauchen nicht mehr berücksichtigt zu werden.

Algorithmus 17 Zerlegefunktion mit simpler Zerlegung durch Array-Kopien)

funct PartitionSimple(F , i_1 , i_2)

Erzeuge Hilfsarray h der Länge $i_2 - i_1 + 1$;

$piv := F[i_1]$; // Irgendein Pivot-Wert, hier der linke Wert

Füge alle Elemente im Bereich i_1 bis i_2 , die kleiner als piv sind, in h ein;
 k ist hierbei der Index in h mit dem nächsten freien Platz;

Merke den unteren Index $i := i_1 + k - 1$ für das erste Pivot-Element;

Füge alle Elemente im Bereich i_1 bis i_2 , die gleich piv sind, in h ein;
 k ist hierbei der Index in h mit dem nächsten freien Platz;

Merke den obereren Index $j := i_1 + k - 2$ für das letzte Pivot-Element;

Füge alle Elemente im Bereich i_1 bis i_2 , die größer als piv sind, in h ein;

Kopiere Elemente in h nach F ab Position i_1 ;

return (i, j) ; // Von i bis j stehen die Pivot-Elmente

tcnuf

Detaillierter Ablauf (Teil 1)

`quickSortSimple(f, 0, 9) ↴`

-	$f =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>10</td><td>4</td><td>5</td><td>1</td><td>3</td><td>8</td><td>7</td><td>9</td><td>11</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>5</td></tr></table>	0	1	2	3	4	5	6	7	8	9		10	4	5	1	3	8	7	9	11										5
0	1	2	3	4	5	6	7	8	9																							
	10	4	5	1	3	8	7	9	11																							
									5																							

`partitionSimple(f, 0, 9) ↴`

- -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>4</td><td>5</td><td>1</td><td>3</td><td>8</td><td>7</td><td>9</td><td>-1</td><td>-1</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		4	5	1	3	8	7	9	-1	-1										
0	1	2	3	4	5	6	7	8	9																							
	4	5	1	3	8	7	9	-1	-1																							

- -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>4</td><td>5</td><td>1</td><td>3</td><td>8</td><td>7</td><td>9</td><td>5</td><td>-1</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		4	5	1	3	8	7	9	5	-1										
0	1	2	3	4	5	6	7	8	9																							
	4	5	1	3	8	7	9	5	-1																							

- -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>4</td><td>5</td><td>1</td><td>3</td><td>8</td><td>7</td><td>9</td><td>5</td><td>10</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		4	5	1	3	8	7	9	5	10										
0	1	2	3	4	5	6	7	8	9																							
	4	5	1	3	8	7	9	5	10																							

`partitionSimple(f, 0, 9) = (8, 8) ↴`

`quickSortSimple(f, 0, 7) ↴`

- - -	$f =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>4</td><td>5</td><td>1</td><td>3</td><td>8</td><td>7</td><td>9</td><td>5</td><td>10</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		4	5	1	3	8	7	9	5	10										
0	1	2	3	4	5	6	7	8	9																							
	4	5	1	3	8	7	9	5	10																							

`partitionSimple(f, 0, 7) ↴`

- - - -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>1</td><td>3</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		1	3	-1	-1	-1	-1	-1	-1	-1										
0	1	2	3	4	5	6	7	8	9																							
	1	3	-1	-1	-1	-1	-1	-1	-1																							

- - - -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>1</td><td>3</td><td>4</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		1	3	4	-1	-1	-1	-1	-1	-1										
0	1	2	3	4	5	6	7	8	9																							
	1	3	4	-1	-1	-1	-1	-1	-1																							

- - - -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>1</td><td>3</td><td>4</td><td>5</td><td>8</td><td>7</td><td>9</td><td>5</td><td>10</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		1	3	4	5	8	7	9	5	10										
0	1	2	3	4	5	6	7	8	9																							
	1	3	4	5	8	7	9	5	10																							

`partitionSimple(f, 0, 7) = (2, 2) ↴`

`quickSortSimple(f, 0, 1) ↴`

- - - - -	$f =$	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td></td><td>1</td><td>3</td><td>4</td><td>5</td><td>8</td><td>7</td><td>9</td><td>5</td><td>10</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9		1	3	4	5	8	7	9	5	10										
0	1	2	3	4	5	6	7	8	9																							
	1	3	4	5	8	7	9	5	10																							

`partitionSimple(f, 0, 1) ↴`

- - - - -	$tmp =$	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>-1</td><td>-1</td></tr></table>	0	1	-1	-1
0	1					
-1	-1					

Detaillierter Ablauf (Teil 2)

- - - - - $tmp =$

0	1
1	-1

- - - - - $tmp =$

0	1
1	3

- - - $partitionSimple(f, 0, 1) = (0, 0)$ ↗

- - - $quickSortSimple(f, 0, -1)$ ↘

// Leeres Intervall ($0 \not\leq -1$), da voriges Pivotelement am Rand:

- - - - - $f =$

0	1	2	3	4	5	6	7	8	9
1	3	4	5	8	7	9	5	10	11

- - - $quickSortSimple(f, 0, -1)$ ↗

- - - $quickSortSimple(f, 1, 1)$ ↘

- - - - - $f =$

0	1	2	3	4	5	6	7	8	9
1	3	4	5	8	7	9	5	10	11

- - - $quickSortSimple(f, 1, 1)$ ↗

- - $quickSortSimple(f, 0, 1)$ ↗

- - $quickSortSimple(f, 3, 7)$ ↘

- - - - - $f =$

0	1	2	3	4	5	6	7	8	9
1	3	4	5	8	7	9	5	10	11

- - - $partitionSimple(f, 3, 7)$ ↘

- - - - - $tmp =$

0	1	2	3	4
-1	-1	-1	-1	-1

- - - - - $tmp =$

0	1	2	3	4
5	5	-1	-1	-1

- - - - - $tmp =$

0	1	2	3	4
5	5	8	7	9

- - - $partitionSimple(f, 3, 7) = (3, 4)$ ↗

- - - $quickSortSimple(f, 3, 2)$ ↘

// Leeres Intervall ($3 \not\leq 2$), da voriges Pivotelement am Rand:

Detaillierter Ablauf (Teil 3)

```
- - - - - f = [ 0 | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 | 9 | 10 | 11 ]
- - - - - quickSortSimple(f, 3, 2) ↘
- - - - - quickSortSimple(f, 5, 7) ↘
- - - - - f = [ 0 | 1 | 2 | 3 | 5 | 5 | 8 | 6 | 7 | 9 | 10 | 11 ]
- - - - - partitionSimple(f, 5, 7) ↘
- - - - - - tmp = [ 0 | 1 | 2 ]
- - - - - - - 7 | -1 | -1
- - - - - - tmp = [ 0 | 1 | 2 ]
- - - - - - - 7 | 8 | -1
- - - - - - tmp = [ 0 | 1 | 2 ]
- - - - - - - 7 | 8 | 9
- - - - - partitionSimple(f, 5, 7) = (6, 6) ↘
- - - - - quickSortSimple(f, 5, 5) ↘
- - - - - f = [ 0 | 1 | 2 | 3 | 4 | 5 | 7 | 6 | 8 | 9 | 10 | 11 ]
- - - - - quickSortSimple(f, 5, 5) ↘
- - - - - quickSortSimple(f, 7, 7) ↘
- - - - - f = [ 0 | 1 | 2 | 3 | 4 | 5 | 5 | 7 | 6 | 8 | 9 | 10 | 11 ]
- - - - - quickSortSimple(f, 7, 7) ↘
- - - - - quickSortSimple(f, 5, 7) ↘
- - - - - quickSortSimple(f, 3, 7) ↘
- - - - - quickSortSimple(f, 0, 7) ↘
- - - - - quickSortSimple(f, 9, 9) ↘
- - - - - f = [ 0 | 1 | 2 | 3 | 4 | 5 | 5 | 7 | 8 | 9 | 10 | 11 ]
- - - - - quickSortSimple(f, 9, 9) ↘
```

Detaillierter Ablauf (Teil 4)

quickSortSimple(f, 0, 9) ↴

Kompakter Ablauf (Teil 1)

$f =$	0	1	2	3	4	5	6	7	8	9
	10	4	5	1	3	8	7	9	11	5

$f =$	0	1	2	3	4	5	6	7	8	9
	4	5	1	3	8	7	9	5	10	11

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	8	7	9	5	10	11

// Leeres Intervall ($0 \not\leq -1$), da voriges Pivotelement am Rand:

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	8	7	9	5	10	11

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	8	7	9	5	10	11

// Leeres Intervall ($3 \not\leq 2$), da voriges Pivotelement am Rand:

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	10	11

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	7	8	9	10	11

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	7	8	9	10	11

$f =$	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	7	8	9	10	11

Die aktuelle Teilfolge ist grün dargestellt (Klausur: Bereich mit [] klammern) und das Pivot-Element rot und unterstrichen.

QuickSort ohne Hilfsarray

In Alg. 16 wurden Hilfsarrays benötigt, um die Folge F zerlegen zu können. Der Pseudocode in Alg. 18 zeigt eine QuickSort-Variante, bei der die Zerlegung der Folge kein Hilfsarray benötigt, da die Elemente im Originalarray vertauscht werden.

Algorithmus 18 QuickSort mit Zerlegung ohne Hilfsarray.

```
proc QuickSortSaake( $F, i_1, i_2$ )                                // Teifolge im Intervall  $i_1$  bis  $i_2$ .
    if  $i_1 < i_2$  then                                              // Intervall mind. ein Element?
         $i := \text{PartitionSaake}(F, i_1, i_2);$ 
        QuickSortSaake( $F, i_1, i - 1$ );                               // Rekursion!
        QuickSortSaake( $F, i + 1, i_2$ );
    fi
corp
```

Algorithmus 19 Zerlegefunktion durch Austausch von Elementen.

funct PartitionSaake(F, i_1, i_2)

Wähle als Pivot das rechte Element, $pivot := F[i_2]$;

Sei $index := i_1$ die Index-Position, an die das nächste Element kleiner oder gleich als das Pivot-Element hingetauscht wird;

for i von i_1 bis $i_2 - 1$ **do**

if $F[i] \leq pivot$ **then**

 Vertausche Elemente an Position $index$ und i ;

 Erhöhe $index$ um 1;

fi

od

Vertausche Elemente an Position $index$ und i_2 ;

return $index$;

tcnuf

Sortfunktion in QuickSortSaake

```
...
public void sort(List<E> list, Comparator<E> cmp) {
    // Starte mit vollem Intervall:
    quickSortSaake(list, 0, list.size() - 1, cmp);
}

private void quickSortSaake(List<E> list, int i1, int i2,
                           Comparator<E> cmp) {
    // Nur solange der linke Index kleiner als der rechte ist
    // gibt es etwas zu tun:
    if (i1 < i2) {
        int pn = partitionSaake(list, i1, i2, cmp);
        quickSortSaake(list, i1, pn - 1, cmp);
        quickSortSaake(list, pn + 1, i2, cmp);
    }
}
...
```

Zerlegefunktion in QuickSortSaake

```
...
private int partitionSaake(List<E> list, int i1, int i2,
                         Comparator<E> cmp) {
    // Pivot-Element ist das am weitesten rechts stehende Element:
    E pivot = list.get(i2);
    int index = i1;

    // In dem Durchlauf werden alle Elemente <= Pivot-Element nach links
    // getauscht. Ähnlich wie Selection-Sort. Die Reihenfolge muss nicht
    // stimmen.
    for (int i = i1; i < i2; i++) {
        if (cmp.compare(list.get(i), pivot) <= 0) {
            swap(list, index, i); // Static included.
            index++;
        }
    }
    // index bleibt eine Position rechts neben dem zuletzt getauschten
    // Element stehen, das kleiner als das Pivot-Element war. Deshalb kann
    // im letzten Schritt diese Position mit dem Pivot-Element vertauscht
    // werden. Dann sind alle links davon <= als das Pivot-Element.
    swap(list, i2, index);
    return index;
}
...
```

Ablauf QuickSortSaake (Teil 1)

list =	0	1	2	3	4	5	6	7	8	9
	10	4	5	1	3	8	7	9	11	5

list =	0	1	2	3	4	5	6	7	8	9
	4	5	1	3	5	8	7	9	11	10

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	11	10

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	11	10

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	11	10

// Leeres Intervall ($4 \not\leq 3$), da voriges Pivotelement am Rand:

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	11	10

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	11	10

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	10	11

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	8	7	9	10	11

// Leeres Intervall ($5 \not\leq 4$), da voriges Pivotelement am Rand:

list =	0	1	2	3	4	5	6	7	8	9
	1	3	4	5	5	7	8	9	10	11

Ablauf QuickSortSaake (Teil 2)

list =

0	1	2	3	4	5	6	7	8	9	10	11
1	3	4	5	5	7	8	9	10	11		

// Leeres Intervall ($8 \not\leq 7$), da voriges Pivotelement am Rand:

list =

0	1	2	3	4	5	6	7	8	9	10	11
1	3	4	5	5	7	8	9	10	11		

list =

0	1	2	3	4	5	5	6	7	8	9	11
1	3	4	5	5	7	8	9	10	11		

Vollständiger Ablauf QuickSortSaake (Teil 1)

Hinweis: In `partitionSaake()` sind das Pivot-Element und die Position `index` **rot unterstrichen**. Die Indexposition `i` ist der rechte Rand des grünen Bereichs.

*quickSortSaake(list, 0, 9) *

0	1	2	3	4	5	6	7	8	9
10	4	5	1	3	8	7	9	11	5

- partitionSaake(list, 0, 9)

0	1	2	3	4	5	6	7	8	9
10	4	5	1	3	8	7	9	11	5

0	1	2	3	4	5	6	7	8	9
10	4	5	1	3	8	7	9	11	5

1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

4	10	5	1	3	8	7	9	11	5
0	1	2	3	4	5	6	7	8	9

4	5	10	1	3	8	7	9	11	15
0	1	2	3	4	5	6	7	8	9

- - 115L -

4	5	1	10	3	8	7	9	11	5
0	1	2	3	4	5	6	7	8	9

-- list =

4	5	1	3	10	8	7	9	11	5
0	1	2	3	4	5	6	7	8	9

-- *list* =

4	5	1	3	10	8	7	9	11	5
0	1	2	3	4	5	6	7	8	9

- - *list* =

4 5 1 3 10 8 7 9 11 5

Vollständiger Ablauf QuickSortSaake (Teil 2)

- `partitionSaake(list, 0, 9) = 4 ↴`

- `quickSortSaake(list, 0, 3) ↴`

- - `list =`

0	1	2	3	4	5	6	7	8	9
4	5	1	3	5	8	7	9	11	10

- - `partitionSaake(list, 0, 3) ↴`

- - - `list =`

0	1	2	3	4	5	6	7	8	9
4	5	1	3	5	8	7	9	11	10

- - - `list =`

0	1	2	3	4	5	6	7	8	9
4	5	1	3	5	8	7	9	11	10

- - `partitionSaake(list, 0, 3) = 1 ↴`

- - `quickSortSaake(list, 0, 0) ↴`

- - - `list =`

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	8	7	9	11	10

- - `quickSortSaake() ↴`

- - `quickSortSaake(list, 2, 3) ↴`

- - - `list =`

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	8	7	9	11	10

- - - `partitionSaake(list, 2, 3) ↴`

- - - `partitionSaake(list, 2, 3) = 3 ↴`

- - - `quickSortSaake(list, 2, 2) ↴`

- - - `list =`

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	8	7	9	11	10

- - - `quickSortSaake() ↴`

- - - `quickSortSaake(list, 4, 3) ↴`

Vollständiger Ablauf QuickSortSaake (Teil 3)

// Leeres Intervall ($4 \not< 3$), da voriges Pivotelement am Rand:

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	5	6	7	8	9	1	3	4	5	5	8	7	9	11	10		
0	1	2	3	4	5	5	5	6	7	8	9															
1	3	4	5	5	8	7	9	11	10																	

- - - quickSortSaake() ↘

- - - quickSortSaake() ↘

- - - quickSortSaake() ↘

- - - quickSortSaake(list, 5, 9) ↗

- - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	5	6	7	8	9	1	3	4	5	5	8	7	9	11	10		
0	1	2	3	4	5	5	5	6	7	8	9															
1	3	4	5	5	8	7	9	11	10																	

- - - partitionSaake(list, 5, 9) ↗

- - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	11	10	1	3	4	5	5	8	7	9	11	10		
0	1	2	3	4	5	5	8	7	9	11	10															
1	3	4	5	5	8	7	9	11	10																	

- - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	11	10	1	3	4	5	5	8	7	9	11	10		
0	1	2	3	4	5	5	8	7	9	11	10															
1	3	4	5	5	8	7	9	11	10																	

- - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>11</td><td>10</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	11	10	1	3	4	5	5	8	7	9	11	10		
0	1	2	3	4	5	5	8	7	9	11	10															
1	3	4	5	5	8	7	9	11	10																	

- - - partitionSaake(list, 5, 9) = 8 ↗

- - - quickSortSaake(list, 5, 7) ↗

- - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	10	11	1	3	4	5	5	8	7	9	10	11		
0	1	2	3	4	5	5	8	7	9	10	11															
1	3	4	5	5	8	7	9	10	11																	

- - -	partitionSaake(list, 5, 7) ↗	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	10	11	1	3	4	5	5	8	7	9	10	11		
0	1	2	3	4	5	5	8	7	9	10	11															
1	3	4	5	5	8	7	9	10	11																	

- - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	10	11	1	3	4	5	5	8	7	9	10	11		
0	1	2	3	4	5	5	8	7	9	10	11															
1	3	4	5	5	8	7	9	10	11																	

- - - partitionSaake(list, 5, 7) = 7 ↗

Vollständiger Ablauf QuickSortSaake (Teil 4)

- - - quickSortSaake(list, 5, 6) ↴

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>8</td><td>10</td><td>9</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	8	10	9	1	3	4	5	5	8	7	9	10	11			
0	1	2	3	4	5	5	8	7	9	8	10	9																
1	3	4	5	5	8	7	9	10	11																			

- - - partitionSaake(list, 5, 6) ↴

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>8</td><td>10</td><td>9</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>8</td><td>7</td><td>9</td><td>10</td><td>11</td><td></td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	8	7	9	8	10	9	1	3	4	5	5	8	7	9	10	11			
0	1	2	3	4	5	5	8	7	9	8	10	9																
1	3	4	5	5	8	7	9	10	11																			

- - - partitionSaake(list, 5, 6) = 5 ↴

- - - quickSortSaake(list, 5, 4) ↴

// Leeres Intervall (5 ⪻ 4), da voriges Pivotelement am Rand:

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9	10	11	1	3	4	5	5	7	8	9	10	11		
0	1	2	3	4	5	6	7	8	9	10	11															
1	3	4	5	5	7	8	9	10	11																	

- - - quickSortSaake() ↴

- - - quickSortSaake(list, 6, 6) ↴

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9	10	11	1	3	4	5	5	7	8	9	10	11		
0	1	2	3	4	5	6	7	8	9	10	11															
1	3	4	5	5	7	8	9	10	11																	

- - - quickSortSaake() ↴

- - - quickSortSaake() ↴

- - - quickSortSaake(list, 8, 7) ↴

// Leeres Intervall (8 ⪻ 7), da voriges Pivotelement am Rand:

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	6	7	8	9	10	11	1	3	4	5	5	7	8	9	10	11		
0	1	2	3	4	5	6	7	8	9	10	11															
1	3	4	5	5	7	8	9	10	11																	

- - - quickSortSaake() ↴

- - - quickSortSaake() ↴

- - - quickSortSaake(list, 9, 9) ↴

- - - -	list =	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr><tr><td>1</td><td>3</td><td>4</td><td>5</td><td>5</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td></td><td></td></tr></table>	0	1	2	3	4	5	5	7	8	9	10	11	1	3	4	5	5	7	8	9	10	11		
0	1	2	3	4	5	5	7	8	9	10	11															
1	3	4	5	5	7	8	9	10	11																	

- - - quickSortSaake() ↴

- - - quickSortSaake() ↴

Vollständiger Ablauf QuickSortSaake (Teil 5)

quickSortSaake() ↵

Nachteile von QuickSort

Im Extremfall ist in jedem Schritt das Pivot-Element ein Randwert der zu sortierenden Teilfolge. Das führt dazu, dass mit jedem Zerlegungsschritt nur ein Element abgespalten wird. Besser wäre jedoch, dass die Zerlegung die Elemente in etwa zwei gleich große Hälften teilt.

Ablauf mit der QuickSort-Variante von Saake:

list =	0	1	2	3
	2	3	4	1

// Leeres Intervall ($0 \not\leq -1$), da voriges Pivotelement am Rand:

list =	0	1	2	3
	1	3	4	2

list =	0	1	2	3
	1	3	4	2

// Leeres Intervall ($1 \not\leq 0$), da voriges Pivotelement am Rand:

list =	0	1	2	3
	1	2	4	3

list =	0	1	2	3
	1	2	4	3

// Leeres Intervall ($2 \not\leq 1$), da voriges Pivotelement am Rand:

list =	0	1	2	3
	1	2	3	4

list =	0	1	2	3
	1	2	3	4

Laufzeit von QuickSort

Bei Rekursion ist es nicht einfach, die Anzahl der Schritte zu ermitteln. Deshalb unterstellen wir jetzt folgende Annahmen:

- Die Zahl der Elemente n ist eine Zweier-Potenz.
- Das Pivot-Element ist **nicht** Teil der Folge.
- Die Folge wird immer so aufgeteilt, dass die Zerlegung optimal (bester Fall) in zwei gleich große Hälften erfolgt.

D. h. die Teilstücke werden jeweils halbiert, solange bis sie nur noch ein Element enthalten. Die Anzahl der Halbierungsschritte (Durchläufe) ist dann:

$$d = \log_2 n$$

Laufzeit von QuickSort

Im Teil Zerlege werden Kopieraktionen vorgenommen. Die Teilfolgen werden immer kürzer, ihre Länge ist $t = n/2^i$ für $i = 1, 2, \dots, d$. Gleichzeitig gibt es davon immer mehr, ihre Anzahl ist $r = 2^i$. Insgesamt werden immer alle

$$t \cdot r = n \frac{2^i}{2^i} = n$$

Elemente pro Durchlauf berücksichtigt.

Pro Durchlauf werden alle Elemente genau zweimal kopiert (auf Hilfsarray h und zurück auf F) und dreimal verglichen. Dies ergibt:

$$f_b(n) = (2 + 3)n \cdot d = 5n \log_2 n \in O(n \log n)$$

Auch im durchschnittlichen Fall hat QuickSort eine Komplexität von $O(n \log n)$.

Stabilität

Die Stabilität von QuickSort hängt von der Zerlegefunktion ab. Bei einfacher Zerlegung mit Kopieren in ein Hilfsarray ist er stabil. Durch das Vertauschen von Elementen in der Variante ohne Hilfsarray kann das Verfahren instabil werden (ähnlich wie beim SelectionSort).

[0, 0, 1, 1, 2, 2, 3, 3, 20, 66, 30, 13, 9, 7, 15, 55, 42, 39, 27, 13, 64, 71, 27, 57, 29, 62, 11, 72, 40, 33, 53, 17, 28..

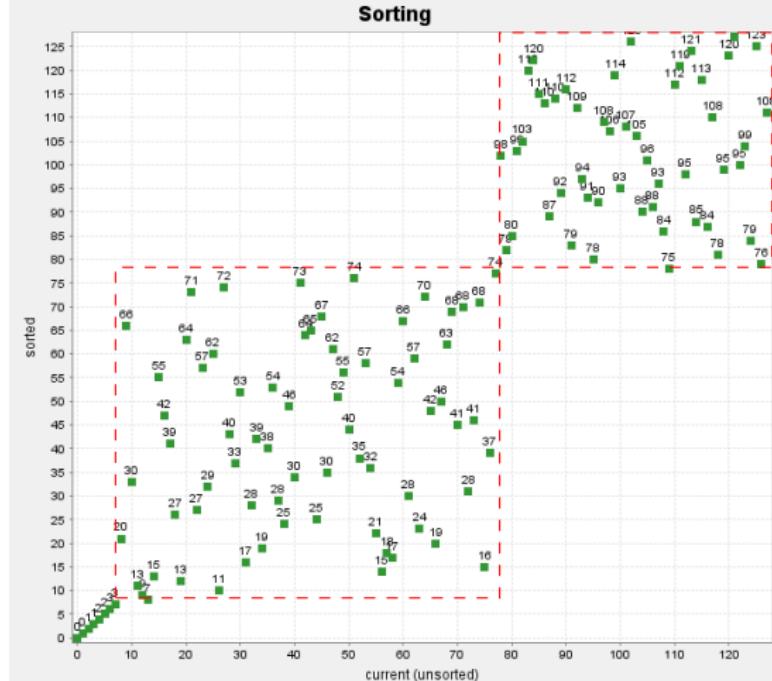


Abbildung 93: Visualisierung von QuickSort in der Variante von Saake.

Abschnittsübersicht

16 Fortgeschrittene Sortierverfahren

MergeSort: Sortieren durch Zusammenfügen

QuickSort: Sortieren durch Zerlegen

Zusammenfassung und Ausblick

Zusammenfassung und Ausblick

Verfahren	Stabilität	Aufwand (Durchschnitt)
BubbleSort	stabil	$O(n^2)$
InsertionSort	stabil	$O(n^2)$
SelectionSort	instabil	$O(n^2)$
MergeSort	(in)stabil	$O(n \cdot \log n)$
QuickSort	(in)stabil	$O(n \cdot \log n)$

Tabelle 22: Übersicht der Eigenschaften der behandelten Sortierverfahren.

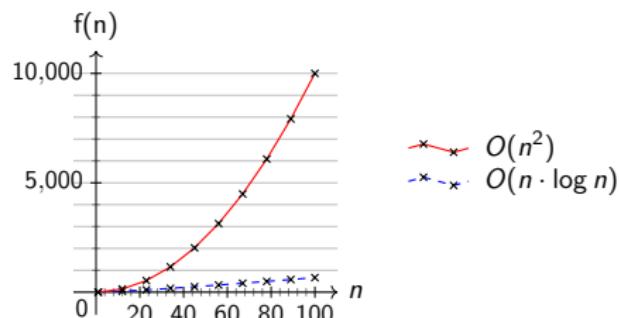


Abbildung 94: Graph für den Laufzeitvergleich.

Ausblick (Teil 1)

Es gibt eine Reihe weiterer Sortierverfahren, die teilweise Varianten der hier vorstellten sind:

RadixSort RadixSort ist ein (neues) Verfahren, das angewandt werden kann, wenn die Elemente in ein Array passen, die Folge keine Duplikate hat und die Schlüssel nummerisch sind. Dann wird mittels einer Fachverteilung (so genannte Buckets) und index-basiertem Zugriff sortiert. RadixSort hat eine Laufzeit von $O(n)$.

HeapSort HeapSort nutzt die Eigenschaft von Heaps zum Sortieren. Allgemein gilt, dass mit Suchbäumen auch Folgen in $O(n \cdot \log n)$ sortiert werden können.

Ausblick (Teil 2)

ShakerSort Eine Erweiterung von BubbleSort ist ShakerSort. Dabei wird abwechselnd ein Durchlauf aufwärts und dann ein Durchlauf abwärts durchgeführt, bei dem jeweils das größte Element hinten und das jeweils kleinste vorne in der Folge landet. Die Laufzeit ist $O(n^2)$.

ShellSort ShellSort ist ein InsertionSort-Variante. Mit ShellSort wird der vermeintliche Nachteil von InsertionSort ausgeglichen, dass große Elemente, die früh eingefügt werden, nur langsam um je eine Position bis ans Ende des Feldes geschoben werden. Die Laufzeit ist $O(n^2)$.

Ausgewählte Datenstrukturen und Algorithmen

Kapitelübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Traversierung

Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Traversierung

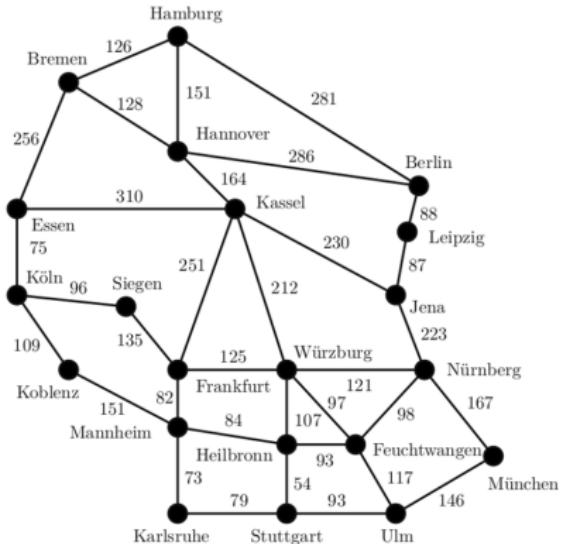
Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume



(a)



(b)

Abbildung 95: Straßennetz als Graph, das die Grundlage für jede Navigationssoftware ist. (a) Kartenausschnitt von Google-Maps. (b) Einfaches Beispiel-Straßennetz aus [?].

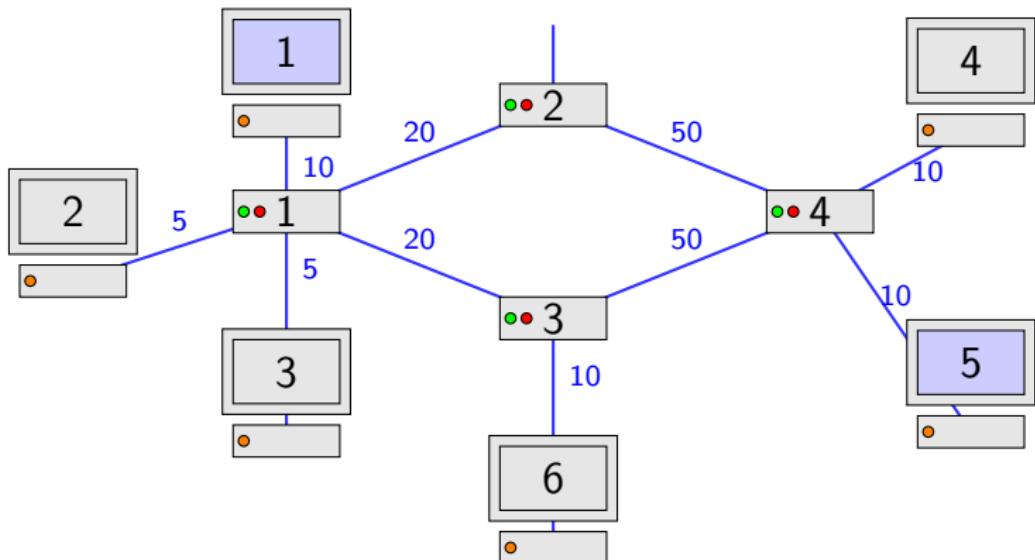
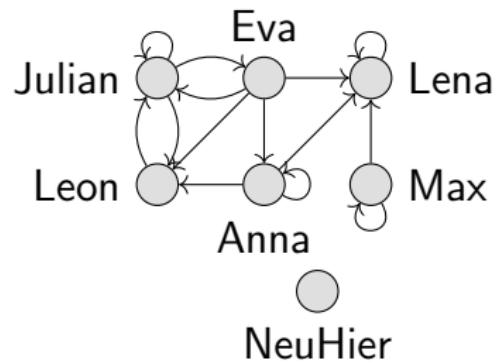


Abbildung 96: Die Topologie eines Computernetzwerks lässt sich gut als Graph darstellen. Es kann ermittelt werden, welche Wege ein IP-Paket nehmen kann und welche Auswirkungen es hätte, wenn ein Router ausfällt. Sind die Verbindungen mit einer Kapazität versehen (bits/s), kann errechnet werden, was der maximale Datendurchsatz von einem Rechner zu einem anderen ist.

mag	Max	Anna	Leon	Eva	Neu-Hier	Lena	Julian
Max	✓				✓		
Anna		✓	✓		✓		
Leon						✓	
Eva		✓	✓		✓	✓	
NeuHier							
Lena						✓	
Julian			✓	✓			✓

(a) Matrix-Darstellung



(b) Graph

Abbildung 97: Freundschaftsbeziehungen als Graph. Welche Personen bilden eine Clique?

Statische Graphen

Die Datenstruktur Graph ist üblicherweise weniger „dynamisch“ als die bisher behandelten Datentypen. Das Einfügen, Suchen und Löschen von Elementen ist möglich, der Schwerpunkt liegt jedoch oft auf Auswertungen eines bestehenden und für die Zeit der Auswertung unveränderlichen Graphen.

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Ungerichteter Graph

Gerichteter Graph

Graphen mit Kanten-Gewichten

Spezielle Graphen

Implementierungen

Traversierung

Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume

Graph

Definition Graph

Ein Graph $G = (V, E)$ besteht aus einer Menge V von Knoten (engl. *Vertices*) dargestellt durch Punkte oder Kreise und einer Menge E von Kanten (engl. *Edges*) als Verbindungen zweier Knoten dargestellt durch Linien. Es bezeichnen $n = |V|$ die Anzahl von Knoten und $m = |E|$ die Anzahl der Kanten.

Die Art der Kanten charakterisiert einen Graphen weiter. Kanten können

- gerichtet oder ungerichtet
- gewichtet oder ungewichtet

sein. Hierbei sind alle vier Kombinationen erlaubt.

Ungerichteter Graph

Bei einem ungerichtetem Graphen sind die Knoten miteinander verbunden, ohne dass eine Richtung vorgegeben ist. Die Kanten werden durch eine Zweiermenge dargestellt $\{v_1, v_2\}$ mit $v_1, v_2 \in V$ und $v_1 \neq v_2$. D. h. es ist kein Selbstbezug eines Knoten möglich.

Beispiel

$$V = \{1, 2, 3, 4, 5, 6, 7\} \text{ und}$$
$$E = \{\{1, 2\}; \{1, 3\}; \{2, 4\};$$
$$\{3, 4\}; \{3, 5\}; \{4, 5\}; \{6, 7\}\}$$

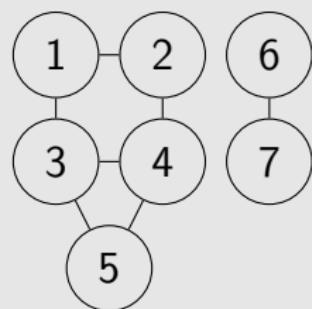


Abbildung 98:
Ungerichteter Graph.

Fragestellungen

Typische Fragestellungen für einen ungerichteten Graphen sind:

Layout Wie könnte der Graph in der Ebene gezeichnet werden, ohne dass sich die Kantenlinien überschneiden?

Nächste Nachbarn Was sind die x nächsten Elemente des Knotens a ?

Zyklen Hat der Graph Rundwege (Zyklen)?

Gruppen Sind alle Elemente untereinander erreichbar oder gibt es Gruppen (Cliquen)?

Eulerkreis Wie sieht ein Weg durch den Graphen aus, bei dem alle Kanten genau einmal durchlaufen werden?

Hamilton-Pfad Wie lässt sich ein Graph vollständig durchlaufen, ohne dass ein Knoten zweimal besucht wird?

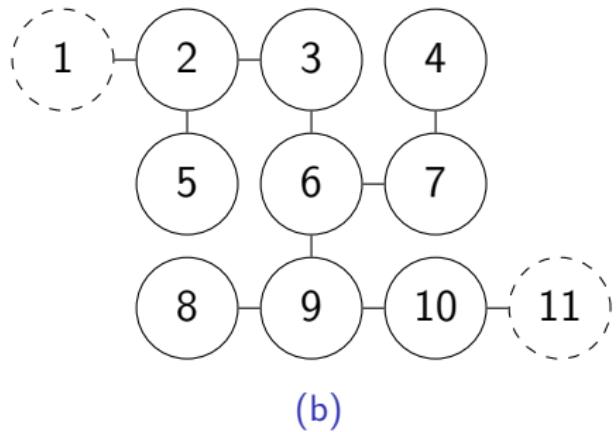
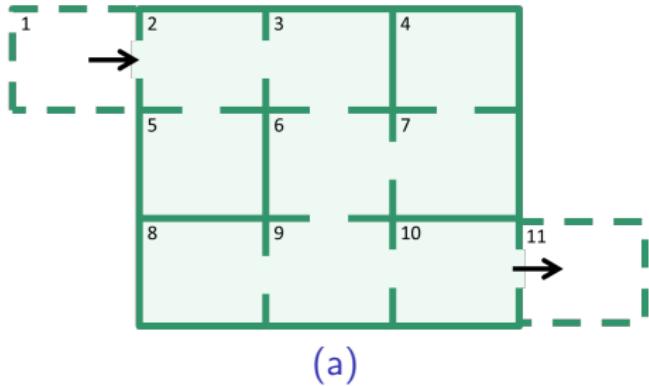


Abbildung 99: Labyrinth als Graph. Auch das Labyrinth in Abb. ?? im Abschnitt ?? lässt sich als Graph modellieren.

Gerichteter Graph

Bei einem gerichteten Graphen (engl. *directed graph*, auch Digraph) sind die Knoten mit einem Richtungssinn verbunden. Die Ecken werden durch ein Zweiertupel (d. h. geordnet) dargestellt: (v_1, v_2) mit $v_1, v_2 \in V$. Somit ist ein Selbstbezug eines Knoten möglich.

Beispiel

$$V = \{1, 2, 3, 4, 5, 6, 7\} \text{ und}$$
$$E = \{(1, 2), (2, 1), (2, 4), (3, 1), (3, 4), (3, 5), (5, 3), (5, 4), (7, 7), (7, 6)\}$$

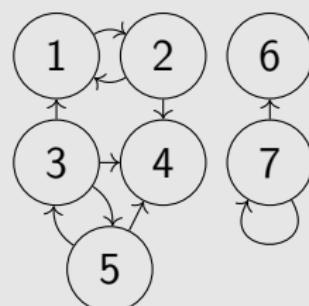
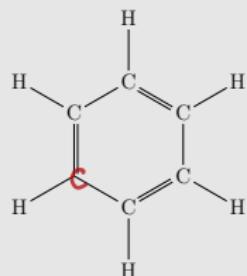


Abbildung 100:
Gerichteter Graph.

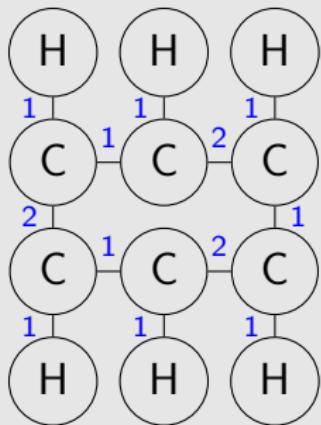
Kanten-Gewichte

Bei einem gerichteten oder ungerichteten Graphen können Kanten zusätzlich noch ein Gewicht erhalten. Ein gewichteter Graph ist ein Dreier-Tupel $G = (V, E, \gamma)$ mit $\gamma : E \rightarrow \mathbb{R}$.

Beispiel



(a) Benzolring



(b) Graph

Abbildung 101: Benzolring als Beispiel für einen gewichteten Graph.

Fragestellungen

Typische Fragestellungen für einen (oft gerichteten) Graphen mit Kanten-Gewichten sind:

Minimaler Abstand Die Berechnung des kürzesten Wegs von einem Knoten zu einem anderen wird sehr häufig benötigt, z. B. in jeder Navigationsssoftware.

Maximaler Durchfluss Wie groß ist die maximale Kapazität der Kanalisation oder eines Netzwerkes allgemein?

Kritischer Pfad Welcher Pfad durch einen Projektplan ist kritisch, d. h. hat Aufgaben ohne Zeitpuffer?

Minimal aufspannende Bäume Wie viele Kilometer umfasst das kürzeste Straßennetz Deutschlands, bei dem immer noch alle Orte erreichbar sind?

Spannbaum

Sei $G = (V, E)$ ein verbundener, ungerichteter Graph. Ein Spannbaum ist ein freier Baum, der alle Knoten aus V enthält und dessen Kanten eine Teilmenge aus E sind.

Der Baum heißt frei, da er keine dedizierte Wurzel hat.

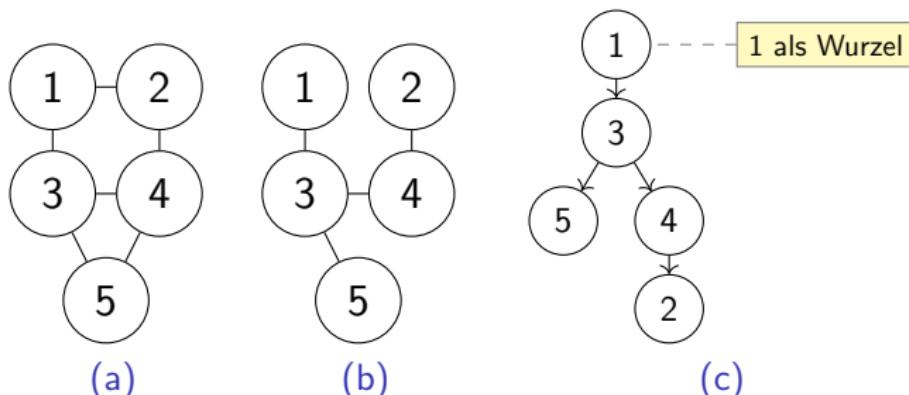


Abbildung 102: Spannbaum. (a) Graph, (b) ein daraus abgeleiteter Spannbaum, der als freier Baum ohne Wurzel dargestellt ist, und (c) der gleiche Baum wie (b) mit Knoten 1 als Wurzel.

Directed Acyclic Graph

Ein (englisch) Directed Acyclic Graph (DAG) ist ein Spezialfall von einem gerichteten Graphen, bei dem es keinen geschlossenen Rundweg entlang der gerichteten Kanten gibt.

Beispiele:

- Stammbaum.
- Feed forward Neuronales Netz.
- Arithmetische Ausdrücke, wenn Variablen wiederverwendet werden sollen.

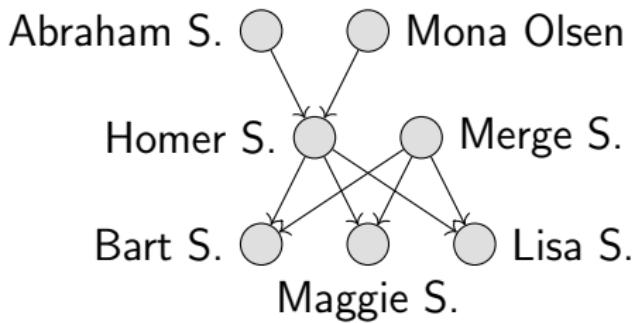


Abbildung 103: Auszug des Stammbaums der Simpsons als Directed Acyclic Graph.

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Kanten- und Knotenliste für ungewichtete Graphen

Adjazenzliste

Adjazenzmatrix

Aufwand (Einfügen/Löschen)

Traversierung

Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume

Kanten- und Knotenliste für ungewichtete Graphen

Die Speicherung eines Graphen als Kanten- und Knotenliste ist eher eine klassische, maschinennahe Form. Hierzu werden die Knoten von 1 bis n nummeriert.

Kantenliste Am Anfang der Liste wird vermerkt, wie viele Knoten bzw. Kanten es gibt. Danach werden die Kanten als Paare von Knoten-Indizes aufgelistet.

Knotenliste Am Anfang der Liste wird wie bei der Kantenliste die Anzahl Knoten bzw. Kanten vermerkt. Danach folgt eine Auflistung von Knoten, wobei für jeden Knoten die Anzahl und die Indizes der ausgehenden Kanten gespeichert wird.

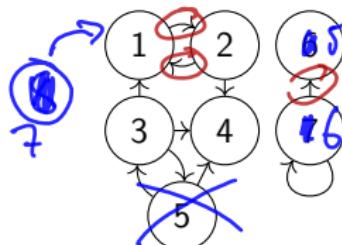
Anzahl		Kanten										
Knoten	Kanten	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
X 8 7	10 m	1	2	2	1		2	4	3	1	3	4
							3	5	5	3	5	4

Kanten

... 6. 7. 8. 9. 10. 11. 8/1

Tabelle 23: Kantenliste für den Beispielgraph. Das Array speichert neben der Anzahl von Knoten und Kanten eine sortierte Liste von Kanten.

Löschen
knoten $O(n)$



Einfügen
knoten $O(1)$
Kante $O(1)$

Abbildung 104: $E = \{(1, 2), (2, 1), (2, 4), (3, 1), (3, 4), (3, 5), (5, 3), (5, 4), (7, 7), (7, 6)\}$

Anzahl	Knoten 1		Knoten 2		Knoten 3		
Knoten	Kanten		#	#	#	#	
7	10	1	2	2	1	4	3
							1 4 5
Knoten 4		Knoten 5		Knoten 6		Knoten 7	
...	#	#		#	#	#	
0	2	3	4	0	2	7	6

Tabelle 24: Knotenliste für den Beispielgraph. # bedeutet Anzahl von Kanten.

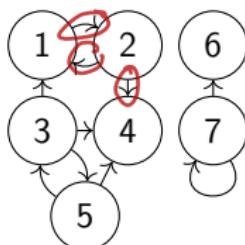


Abbildung 105: $E = \{(1, 2), (2, 1), (2, 4), (3, 1), (3, 4), (3, 5), (5, 3), (5, 4), (7, 7), (7, 6)\}$

Adjazenzliste

Eine Adjazenzliste ist eine weitere dynamische Datenstruktur für einen Graphen, bei der sowohl die Knoten wie auch die Kanten pro Knoten in einer verketteten Liste verwaltet werden.

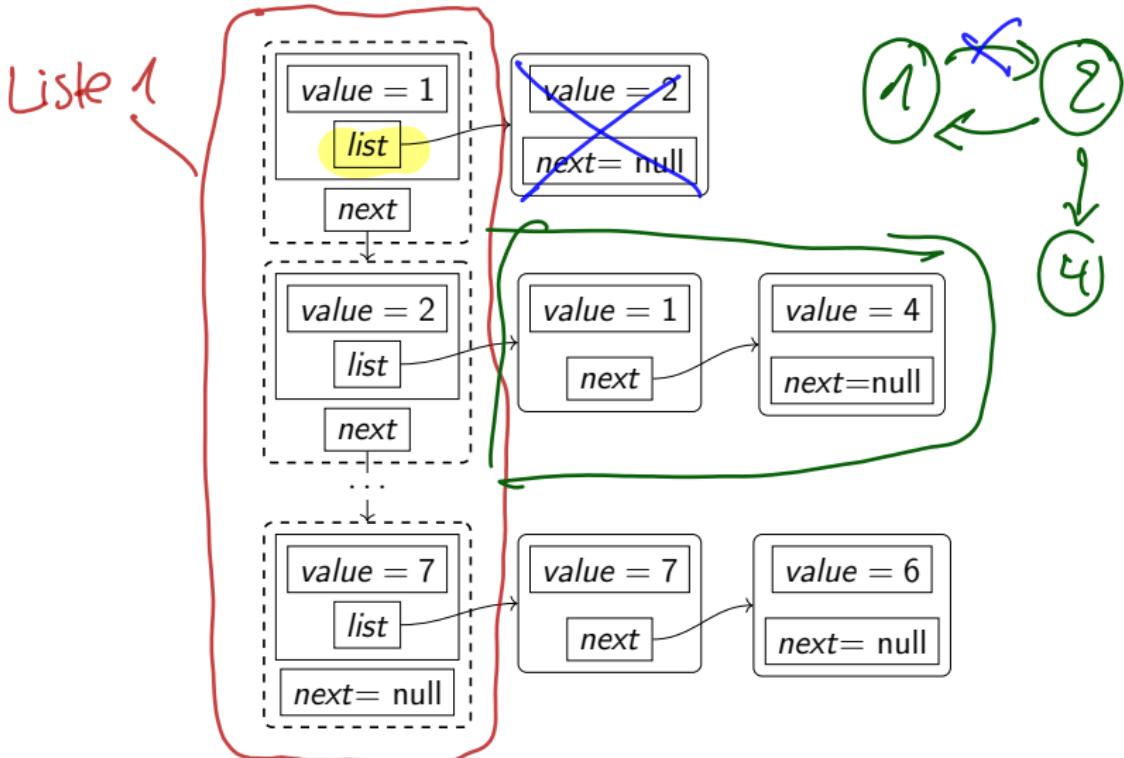
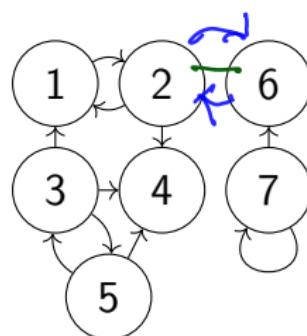


Abbildung 106: Adjazenzliste. Links sind vertikal alle Knoten zu sehen, rechts davon die ausgehenden Kanten. *value* entspricht der Knotennummer.

Adjazenzmatrix

Hier werden die Kanten in einer Matrix gespeichert. Die Zellen bestehen aus binären Werten, falls der Graph ungewichtet ist, ansonsten aus Zahlen. Die Matrix ist symmetrisch, falls der Graph ungerichtet ist. Diese Implementierung ist sinnvoll, wenn der Graph viele Kanten hat.



nach

$$M = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

von

(b) Adjazenzmatrix

Abbildung 107: Graph als Adjazenzmatrix.

Die Klasse WeightedMatrixGraph

```
...
public class WeightedMatrixGraph<E> {

    private E[][] matrix;
    private boolean isDirected = true;

    /**
     * Erzeuge einen Graph mit <code>numberNodes</code> Knoten.
     *
     * @param numberNodes
     * @param directed      Wahr, wenn dies ein gerichteter Graph
     *                      sein soll.
     */
    public WeightedMatrixGraph(int numberNodes, boolean directed) {
        matrix = (E[][] ) new Object[numberNodes][numberNodes];
        isDirected = directed;
    }
    ...
}
```

Erzeugen von Kanten (Teil 1)

```
...
public class WeightedMatrixGraph<E> {
...
    /**
     * @return Anzahl von Knoten (Vertices).
     */
    public int getNumberOfVertices() {
        return matrix.length;
    }

    /**
     * @param from
     * @param to
     * @return null, falls keine Kante zwischen Knoten from
     * und to besteht, sonst das Gewicht der Kante.
     */
    public E getEdge(int from, int to) {
        return matrix[from][to];
    }

    /**
     * Fügt eine gewichtete Kante zwischen Knoten from und to ein.
     */
}
```

Erzeugen von Kanten (Teil 2)

```
*  
* @param from  
* @param to  
* @param weight  
*/  
public void addDirectedEdge(int from, int to, E weight) {  
    matrix[from][to] = weight;  
}  
  
/**  
 * Fügt eine gewichtete, aber ungerichtete Kante zwischen  
 * Knoten i und j ein.  
 *  
 * @param i  
 * @param j  
 * @param weight  
 */  
public void addEdge(int i, int j, E weight) {  
    matrix[i][j] = weight;  
    matrix[j][i] = weight;  
}  
...  
...
```

Erzeugen von Kanten (Teil 3)

}

...

Anwendung

Das nachfolgende Listing zeigt die Erzeugung des Beispielgraphen.
Die Knoten beginnen bei 0 zu zählen.

...

```
UnweightedMatrixGraph g = new UnweightedMatrixGraph(7, true);
g.addDirectedEdge(0, 1);
g.addDirectedEdge(1, 0);
g.addDirectedEdge(1, 3);
g.addDirectedEdge(2, 0);
g.addDirectedEdge(2, 3);
g.addDirectedEdge(2, 4);
g.addDirectedEdge(4, 2);
g.addDirectedEdge(4, 3);
g.addDirectedEdge(6, 5);
g.addDirectedEdge(6, 6);
```

...

Operation		Kanten-liste	Knoten-liste	Adjazenz-liste	Adjazenz-matrix
Einfügen	Knoten	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
Löschen		$O(m)$	$O(n+m)$	$O(n+m)$	$O(n^2)$
Einfügen	Kante	$O(1)$	$O(n+m)$	$O(n)$	$O(1)$
Löschen		$O(m)$	$O(n+m)$	$O(n)$	$O(1)$

Tabelle 25: Komplexität der Einfüge- und Löschoperationen bei verschiedenen Graph-Implementierungen. $n = |V|, m = |E|$.

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Traversierung

Tiefensuche

Breitensuche

Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume

Traversierung von Graphen

Die Idee, alle Knoten eines Graphen zu bearbeiten, ist, den Graph zunächst in einen Baum zu überführen und dann die beim Baum bekannten Traversierungen zu nutzen.

Expansion eines Graphen

Die Expansion $X(v)$ eines Graphen G im Knoten v ist ein Baum mit folgenden Eigenschaften:

- Falls v keine Nachfolger hat: Baum mit einzigm Knoten v .
- Falls der Knoten v die Nachfolger v_1, \dots, v_k hat: Baum mit v als Wurzel und $X(v_1), \dots, X(v_k)$ als Kinder.

Die Expansion erzeugt einen unendlich großen Baum, falls G Zyklen hat — diese müssen bei den Baum-Traversierungsalgorithmen berücksichtigt werden.

Beispiel

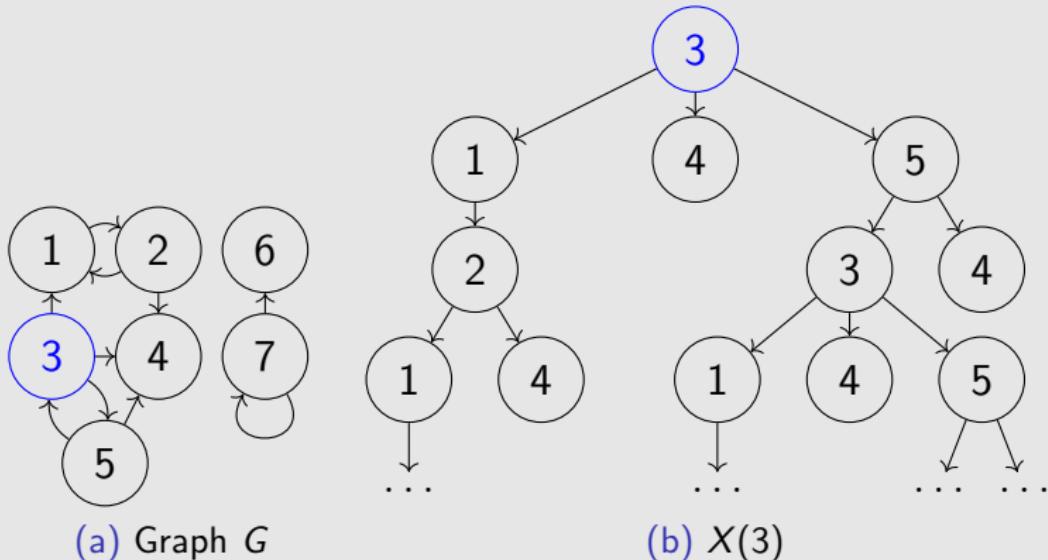


Abbildung 108: Beispiel für die Expansion $X(3)$ des Graphen G . Aufgrund der Zyklen im Graphen entsteht ein unendlich großer Baum.

Tiefensuche

Bei der so genannten Tiefensuche werden zunächst alle Knoten entlang eines Pfads verfolgt, solange bis der Knoten ein Blatt ist oder ein Zyklus erreicht wurde. Danach wird ein weiterer Pfad abgesucht. Somit werden weit entfernte Knoten zuerst erreicht.

Tiefensuche (Depth-First-Traversal)

Der Tiefendurchlauf ab Knoten v entspricht dem Preorder-Durchlauf der Expansion $X(v)$.

Beispiel

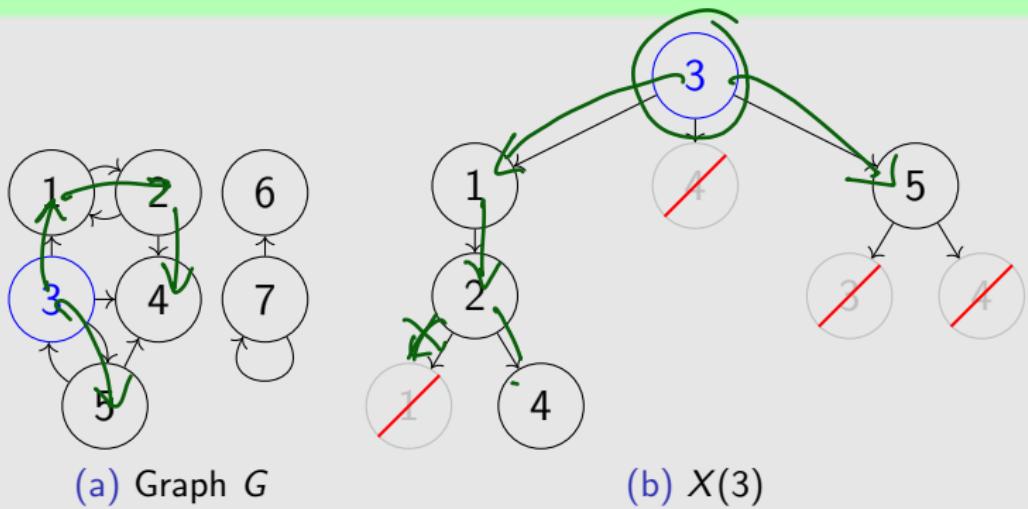


Abbildung 109: Beispiel für die Tiefensuche im Graph G ausgehend von Knoten 3. Die sich ergebende Reihenfolge ist 3, 1, 2, 4, 5.

Algorithmus 20 Tiefensuche. Vgl. Alg. 2.

```
proc Graph.depthFirst(v) // Tiefensuche ab Knoten v.  
    if v noch nicht besucht then  
        Markiere v als besucht;  
        Verarbeite Knoten v; ← Pre  
        for jeden Nachfolger  $v_i$  von v do  
            depthFirst( $v_i$ );  
    od  
fi  
corp
```

Tiefensuche (Teil 1)

```
...
public class WeightedMatrixGraph<E> {
...
    /**
     * Durchlaufe den Graph in Tiefensuche und gib die Knoten
     * in einer Liste zurück.
     *
     * @param v Startknoten
     * @return Liste mit den Knoten-Indizes.
     */
    public java.util.List getDepthFirst(int v) {
        boolean[] visited = new boolean[getNumberOfVertices()];
        var list = new ArrayList<Integer>();
        // Das besucht-Feld wird als Parameter übergeben:
        getDepthFirstRec(v, visited, list);
        return list;
    }
}
```

Tiefensuche (Teil 2)

```
/**  
 * Durchlaufe den Graph in Tiefensuche und fülle dabei die  
Liste.  
 *  
 * @param v          Aktueller Knoten  
 * @param visited Array das angibt, ob ein Knoten bereits  
besucht wurde.  
 * @param list Liste, die nach und nach gefüllt wird.  
 */  
private void getDepthFirstRec(int v, boolean[] visited,  
                               java.util.List<Integer> list) {  
    if (!visited[v]) {  
        visited[v] = true;  
        list.add(v); // Verarbeite Knoten v.  
        // Für jeden Nachfolger von v...  
        for (int j = 0; j < getNumberOfVertices(); j++) {  
            if (j != v && getEdge(v, j) != null) {  
                getDepthFirstRec(j, visited, list);  
            }  
        }  
    }  
}
```

Tiefensuche (Teil 3)

```
    }  
}  
}  
}  
}  
...  
}  
...
```

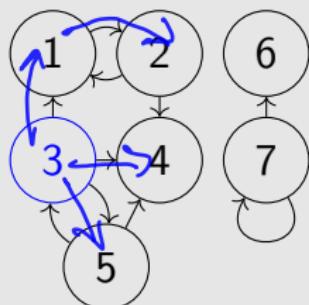
Breitensuche

Bei der so genannten Breitensuche werden zunächst alle direkt benachbarten Knoten, dann die Nachbar-Knoten zweiten Grades, drittes Grades usw. besucht.

Breitensuche (Breadth-First-Traversal)

Die Breitensuche ab Knoten v entspricht dem Levelorder-Durchlauf der Expansion $X(v)$.

Beispiel



(a) Graph G

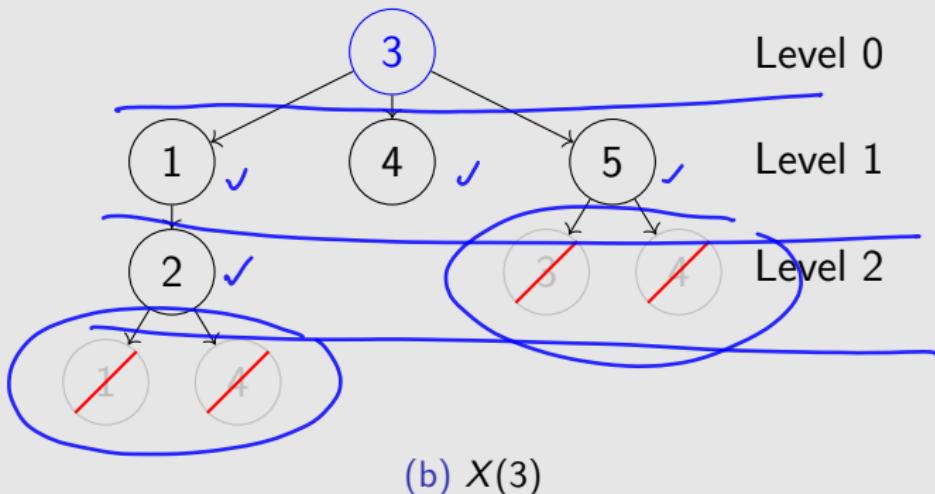


Abbildung 110: Beispiel für die Breitensuche im Graph G ausgehend von Knoten 3. Die sich ergebende Reihenfolge ist 3, 1, 4, 5, 2

Algorithmus 21 Breitensuche. Vgl. Alg. 4.

proc Graph.breadthFirst(v)

// Breitensuche ab Knoten v .

Erzeuge Warteschlange q und füge v in q ein;

Markiere v als besucht;

while q nicht leer ist **do**

Entnehme vorderstes Element aus q und weise es w zu;

Verarbeite w ;

for jeden Nachfolger w_i von w **do**

if w_i noch nicht besucht **then**

Füge w_i in q ein;

Markiere w_i als besucht;

fi

od

od

corp

Breitensuche (Teil 1)

```
...
public class WeightedMatrixGraph<E> {
...
/** 
 * Durchlufe den Graph in Breitensuche und gib die Knoten
 * in einer Liste zurück.
 *
 * @param v Startknoten
 * @return Liste mit den Knoten-Indizes.
 */
public java.util.List<Integer> getBreadthFirst(int v) {
    boolean[] visited = new boolean[getNumberOfVertices()];
    var list = new ArrayList<Integer>();
    // ArrayQueue ist voll ausprogrammiert, ListQueue nicht.
    Queue<Integer> queue = new ArrayQueue<>();
    queue.enter(v);
    visited[v] = true;
```

Breitensuche (Teil 2)

```
while (!queue.isEmpty()) {  
    int w = queue.leave();  
    list.add(w); // Verarbeite Element.  
    for (int wi = 0; wi < getNumberOfVertices(); wi++) {  
        if (wi != w && getEdge(w, wi) != null) {  
            if (!visited[wi]) {  
                queue.enter(wi);  
                visited[wi] = true;  
            }  
        }  
    }  
    return list;  
}  
...  
}
```

Breitensuche (Teil 3)

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Traversierung

Kürzeste Entfernung von einem Knoten zu einem anderen

Algorithmus von Dijkstra

Bellmann-Ford-Algorithmus

Maximaler Durchfluss

Minimale Spannbäume

Kürzeste Entfernung von einem Knoten zu einem anderen

Diese Fragestellung wird auch *Single-source shortest path*-Problem genannt. Bekannte Algorithmen hierfür sind von:

Dijkstra Dies ist ein Greedy-Algorithmus, der nur für Distanzen ≥ 0 funktioniert [?].

Bellmann-Ford ist für alle Kantengewichte anwendbar [?].

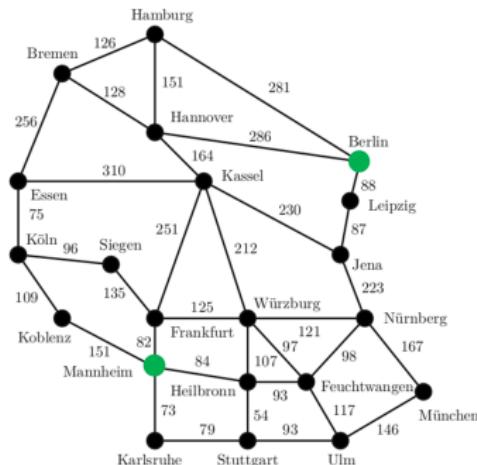


Abbildung 111: Entfernungen zwischen Städten in Deutschland. Wie weit ist es von Mannheim nach Berlin?

Algorithmus von Dijkstra

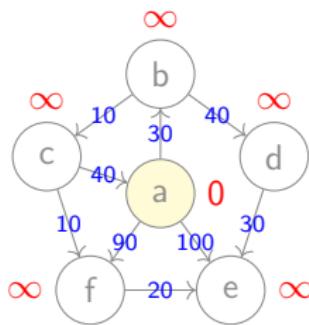
Hinweis

Um den kürzesten Abstand von Knoten v zum Knoten z zu bestimmen, muss ohnehin jeder Knoten untersucht werden. Indirekt wird so der kürzeste Abstand von v zu allen Knoten im Graph berechnet.

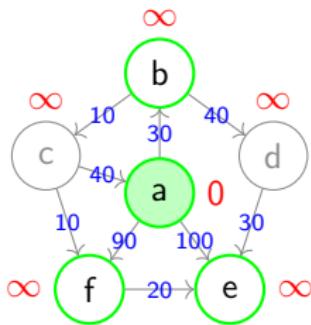
Idee: Ausgehend vom Startknoten v werden nach und nach Knoten mit kürzesten Abstand besucht. Der Abstand von diesem Knoten (w) zum Startknoten wird gemerkt (Variable $\text{dist}(w)$). Hat ein (neuer) Pfad zu einem Knoten einen geringeren Abstand als der bisher kürzeste, wird dieser Pfad gewählt. Dabei wird ein Spannbaum mit Startknoten als Wurzel erzeugt (Vorgänger $\text{pred}(w)$ wird gemerkt).

Algorithmus 22 Dijkstra-Algorithmus.

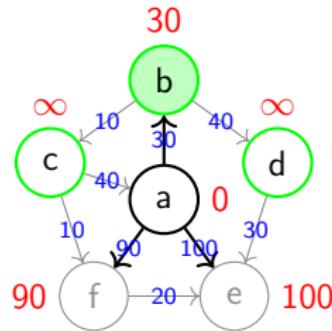
```
proc Graph.dijkstra(v)           // Knoten v, ab dem Entferungen bestimmt werden.  
    Setze für jeden Knoten w die vorläufige Distanz  $\text{dist}(w) = \infty$   
    und den Vorgänger  $\text{pred}(w)$  auf null;  
    Setze  $\text{dist}(v) := 0$  und  $\text{pred}(v) := v$ ;  
    Setze die Menge der unbesuchten Knoten  $V' := V$ ;  
    while es gibt ein Knoten  $w \in V'$  mit  $\text{dist}(w) < \infty$  do  
        Wähle ein  $v' \in V'$  mit  $\text{dist}(v')$  minimal;          // Zu Beginn ist das v.  
        Sei  $N$  die Menge aller Nachfolger von  $v'$ , die auch in  $V'$  sind;  
        for alle  $n$  in  $N$  do  
            if  $\text{dist}(v') + \gamma((v', n)) < \text{dist}(n)$  then // Kürzeren Weg gefunden?  
                 $\text{dist}(n) := \text{dist}(v') + \gamma((v', n))$ ;  
                 $\text{pred}(n) := v'$ ;                         // Merke neuen Vorgänger.  
            fi  
        od  
        Entferne  $v'$  aus  $V'$ ;  
    od  
corp
```



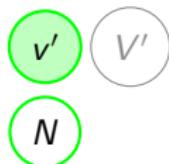
(a) Start



(b) 1a

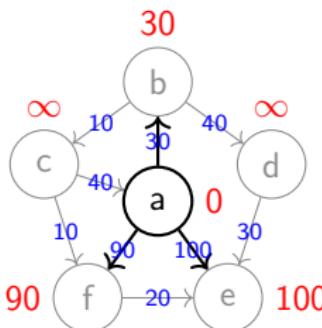


(c) 2a

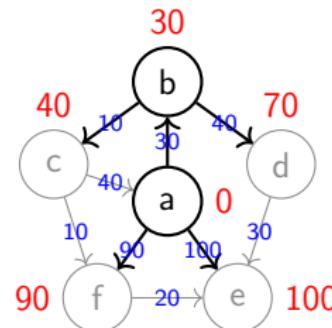


→ prev.

(d) Legende

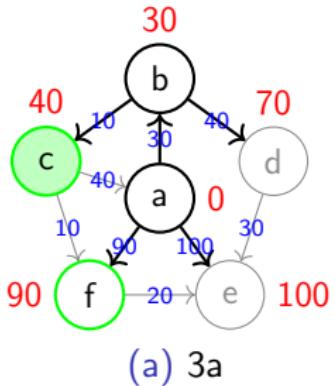


(e) 1b

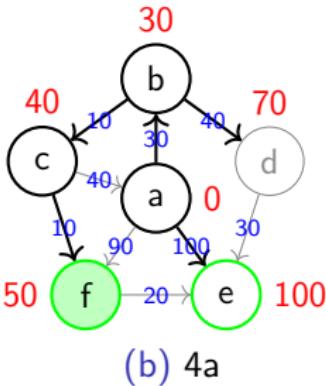


(f) 2b

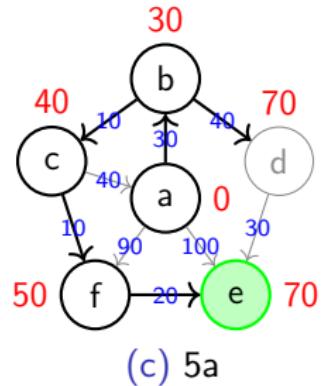
Abbildung 112: Ablauf Dijkstra-Algorithmus (Teil 1). Berechnet werden die minimalen Abstände zu Knoten a.



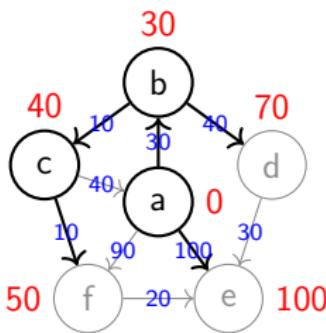
(a) 3a



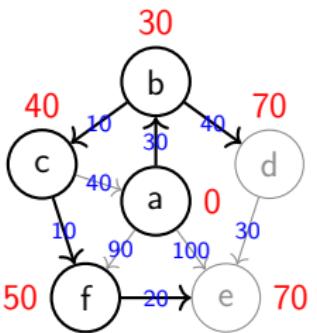
(b) 4a



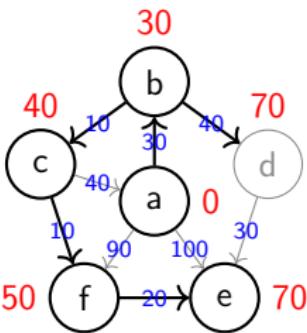
(c) 5a



(d) 3b

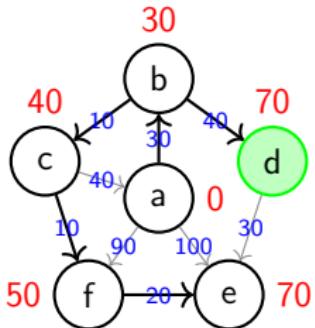


(e) 4b

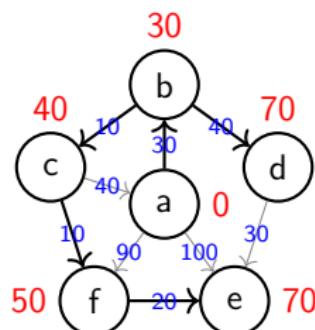


(f) 5b

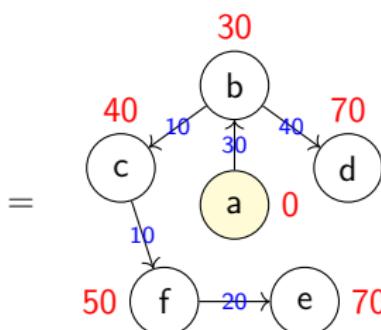
Abbildung 113: Ablauf Dijkstra-Algorithmus (Teil 2).



(a) 6a



(c) 6b



(d) Spannbaum

Abbildung 114: Ablauf Dijkstra-Algorithmus (Teil 3).

Eigenschaften des Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist ein Greedy-Algorithmus (engl. *gierig*): Es wird stets der Knoten v' mit geringstem Abstand als nächster gewählt, damit die Abstände möglichst langsam wachsen. Er funktioniert dadurch nur für Distanzen größer 0, da aufgrund des gierigen Verhaltens lokale Minima nicht erkannt werden.

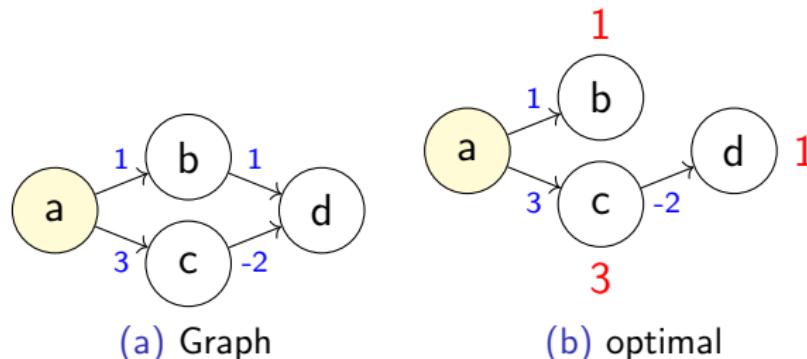
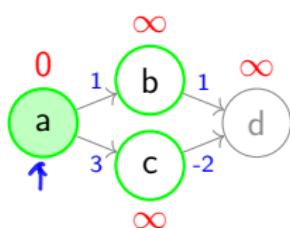
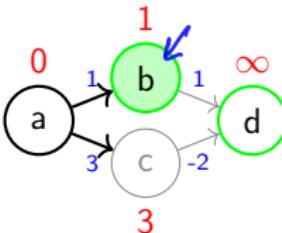


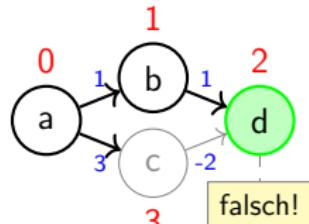
Abbildung 115: (a) Graph mit negativen Kantengewichten. (b) zeigt die optimale Lösung.



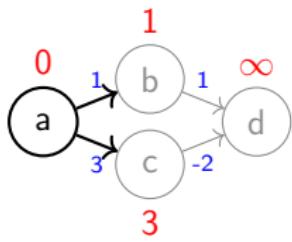
(a) 1a



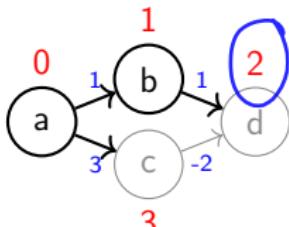
(b) 2a



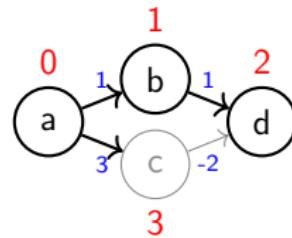
(c) 3a



(d) 1b



(e) 2b



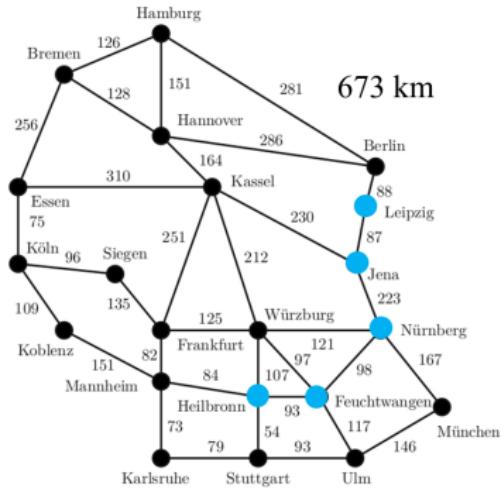
(f) 3b

Abbildung 116: Beispiel, wie der Dijkstra-Algorithmus bei negativen Kantengewichten fehlschlägt. Im dritten Schritt wählt der Algorithmus den Knoten d aus V' aus, da $\text{dist}(d) = 2 < 3 = \text{dist}(c)$ ist. Diese Entscheidung ist falsch.

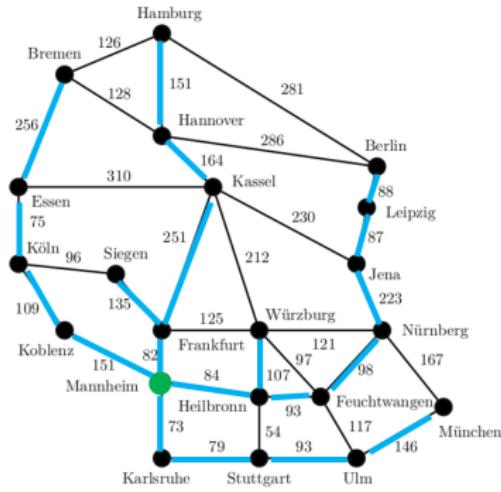
Aufwand

Implementierung	Aufwand
wie in Alg. 22	$O(n^2)$
mit Priority Queue	$O(m + n \cdot \log n)$

Tabelle 26: Aufwand des Dijkstra-Algorithmus in allen Fällen. $n = |V|$ und $m = |E|$.



(a)



(b)

Abbildung 117: Wie weit ist es von Mannheim nach Berlin? (a) zeigt den Streckenverlauf und die Distanz von 673 km. In (b) ist der Spannbaum (blau) gezeigt, der alle Pfade zu allen anderen Städten angibt.

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Traversierung

Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume

Abschnittsübersicht

17 Graphen

Einführung

Arten von Graphen

Implementierungen

Traversierung

Kürzeste Entfernung von einem Knoten zu einem anderen

Maximaler Durchfluss

Minimale Spannbäume

Programmabläufe

Für das Verständnis der Programmierung ist es sehr wichtig, genau verstanden zu haben, wie ein Programm ausgeführt wird. Was genau bedeutet der verschachtelte Funktionsaufruf von eben? In diesem Abschnitt wollen wir systematisch überlegen, wie die Ausführung eines Programms, d. h. Pseudocode, auf dem Papier notiert werden kann. Später überlassen wir das natürlich dem Computer.

Debugger

In Entwicklungsumgebungen wie IntelliJ oder Eclipse kann die Programmausführung auch sehr gut mit einem sogenannten Debugger verfolgt werden.

Darstellung von Programmabläufen

Für die Protokollierung eines Programmablaufs verwenden wir folgende Konventionen:

- Der Aufruf des (einzigsten) Hauptprogramms wird mit `main()` protokolliert.
- Anweisungen innerhalb eines Unterprogrammaufrufs werden eingerückt dargestellt und durch den aufrufenden Programmteil mit `\` und `/` geklammert.
- Jede Zuweisung einer Variablen wird protokolliert (sofern nicht anders angegeben).
- Parameter einer Funktion werden als Wert angeben, ggf. auch mit Variablenname.
- Ausdrücke in Bedingungen werden nicht notiert.
- Ausgaben werden mit „Ausgabe“ protokolliert.

Programmablauf – Basiselemente

Algorithmus

23
Hauptprogramm, das die ersten n geraden Zahlen ausgibt → TestGerade.java

```
Gib aus "n eingeben:";  
Lies n ein;  
for z von 1 bis n do  
    if z mod 2 = 0 then  
        Gib z aus;  
    fi  
od  
Gib aus "Fertig!";
```

Der Ablauf des Hauptprogramms für die Eingabe $n = 7$ ist:

main() ↘

– Ausgabe "n eingeben: "

– $n = 7$ (von Eingabe)

– $z = 1$

– $z = 2$

– Ausgabe "2"

– $z = 3$

– $z = 4$

– Ausgabe "4"

– $z = 5$

– $z = 6$

– Ausgabe "6"

– $z = 7$

– Ausgabe "Fertig!"

main() ↗

Programmablauf – Funktionen

Bei Funktionen werden der Name und die Parameter sowie der Rückgabewert verschachtelt notiert.

Algorithmus 24

Potenzfunktion a^b (für $b \geq 0$)

→ Pow.java

funct pow(a, b)

$r := 1;$

for i von 1 bis b **do**

$r := r \cdot a;$

od

return $r;$

tcnuf

Der Ablauf ist:

main() ↘

– *pow(a = 2, b = 4)* ↘

– – $r = 1$

– – $i = 1$

– – $r = 2$

– – $i = 2$

– – $r = 4$

– – $i = 3$

– – $r = 8$

– – $i = 4$

– – $r = 16$

– *pow(2, 4) = 16* ↘

main() ↘

Programmablauf – Prozeduren

Bei Prozeduren fehlt der Rückgabewert.

Algorithmus 25 Rekursives Hallo

```
proc h(n)
    if n > 0 then
        Gib aus "Hallo";
        h(n - 1);
    fi
corp
```

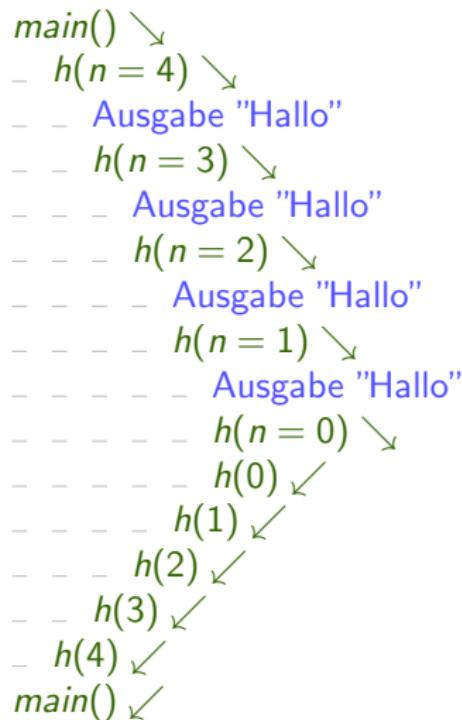
Programmablauf – Prozeduren

Bei Prozeduren fehlt der Rückgabewert.

Algorithmus 26 Rekursives Hallo

```
proc h(n)
    if n > 0 then
        Gib aus "Hallo";
        h(n - 1);
    fi
corp
```

Der Ablauf von Aufruf $h(4)$ ist:



Gleches Protokoll bedeutet deterministisch

Ist bei gleicher Eingabe das Ablaufprotokoll stets dasselbe, liegt ein deterministischer Algorithmus vor.

- [Bü10] Christina Büsing.
Graphen- und Netzwerkoptimierung.
Spektrum, Akademischer Verlag, Heidelberg, 2010.
- [Bel58] Richard Bellman.
On a routing problem.
Quart. Appl. Math., 16:87–90, 1958.
- [Blu12] Norbert Blum.
Algorithmen und Datenstrukturen.
Oldenbourg Wissenschaftsverlag, 2 edition, 2012.
- [Dij59] Edsger W. Dijkstra.
A note on two problems in connexion with graphs.
Numerische Mathematik, 1:269–271, 1959.
- [DMS14] Martin Dietzfelbinger, Kurt Mehlhorn, and Peter Sanders.
Algorithmen und Datenstrukturen: Die Grundwerkzeuge.
eXamen.press, 1 edition, 2014.

- [Gü13] Ralf Hartmut Güting.
Datenstrukturen und Algorithmen.
Leitfäden der Informatik. Springer, 3 edition, 2013.
- [GS] Heinz-Peter Gumm and Manfred Sommer.
Einführung in die Informatik.
Oldenbourg Wissenschaftsverlag.
- [Kof17] Michael Kofler.
Java: Der kompakte Grundkurs mit Aufgaben und Lösungen im Taschenbuchformat.
Rheinwerk Computing, 2 edition, 2017.
- [Moo65] G Moore.
Cramming more components onto integrated circuits.
electronics 38 (8): 114–117, 1965.
- [OW12] Thomas Ottmann and Peter Widmayer.
Algorithmen und Datenstrukturen.
Spektrum Akademischer Verlag, 2012.

- [SS14] Gunter Saake and Kai-Uwe Sattler.
Algorithmen und Datenstrukturen.
dpunkt.verlag, 5 edition, 2014.
- [TH16] Manh Tien Tran and Jörg Hettel.
Nebenläufige Programmierung mit Java: Konzepte und Programmiermodelle für Multicore-Systeme.
dpunkt. verlag, 2016.
- [Ull15] Christian Ullenboom.
Java ist auch eine Insel.
Galileo Press, 2015.
<http://www.tutego.de/javabuch/>.
- [WW] Karsten Weicker and Nicole Weicker.
Algorithmen und Datenstrukturen.
Leitfäden der Informatik. Springer Vieweg.