Algorithmen und Datenstrukturen

Herbst 2025

FHDW Fachhochschule der Wirtschaft

Verwendungshinweise

- Dieses Dokument ist nur für den internen Gebrauch an der FHDW durch Studierende und Mitarbeiter der Hochschule bestimmt.
- Es darf Dritten, auch auszugsweise, nur mit vorheriger schriftlicher Genehmigung der FHDW zugänglich gemacht werden.
- Dies gilt insbesondere für eine Weitergabe in elektronischer Form, z.B. im Internet.

Algorithmen und Datenstrukturen

1.	Einführung	6
2.	O-Notation	9
3.	Komplexität	17
4.	Array und Listen	30
5.	Rekursive Algorithmen	50
6.	Bäume	57
7.	Sammlungen	135
8.	Abstrakter Datentyp (Beispiel Stack)	145
9.	Queus	161

Algorithmen und Datenstrukturen

10.	Generalized Queus	168
11.	Anwendungen	183
12.	Grundzüge paralleler Algorithmen	185
13.	Generisches beim Sortieren	193
14.	Sortieralgorithmen – Implementierung und Analyse	203
15.	Such-Algorithmen	232
16.	Hashing und lineares Sondieren	241
17.	Compiler-Algorithmen	267
18.	Algorithmen für Numerik/Grafik	276

Das Formale

Stunden, Aufwand, Rahmenbedingungen

- 40 Kontaktstunden (50% Übungen/studentische Vorträge)
- 85 Stunden im Selbststudium: Nachbereitung, spezielle Übungen, Hausaufgaben
- Gewichtung: 5/180
- Prüfungsleistung als Klausur (90 min):
  - Verständnisfragen (Begriffe, Konzepte)
  - Effizienzanalysen

1. Einführung

FHDW
Fachhochschule der Wirtschaft

Literaturhinweise

- Sedgewick, R. und K. Wayne , 2014: Algorithmen. Algorithmen und Datenstrukturen, 4. Auflage. München: Pearson
- Pomberger, G, 2008: Algorithmen und Datenstrukturen. Eine systematische Einführung in die Programmierung. München: Pearson

Ergänzende Literaturempfehlungen

- Knuth, D., 1994-1998. The Art of Computer Programming. Band 1 – 4, Boston MA: Addison Wesley
- Vöcking, B. et al., 2008. Taschenbuch der Algorithmen. Berlin: Springer

Ziele der Vorlesung

- Verständnis des Entwurfs und der Analyse grundlegender Algorithmen und Datenstrukturen
- Vertiefter Einblick in ein modernes Programmiermodell (mit Java)
- Wissen um Chancen, Probleme und Grenzen des parallelen nebenläufigen Programmierens

Einerseits

Unverzichtbares Grundlagenwissen aus der Informatik

Andererseits

Vorbereitung für Ihr weiteres Studium und die Praxis

2. O-Notation

O-Notation - Allgemeines

- Landau-Symbole (auch O-Notation, englisch big O notation) werden in der Mathematik und in der Informatik verwendet, um das asymptotische Verhalten von Funktionen und Folgen zu beschreiben.
- In der Informatik werden sie bei der Analyse von Algorithmen verwendet und geben ein Maß für die Anzahl der Elementarschritte oder der Speichereinheiten in Abhängigkeit von der Größe des gegebenen Problems an.
- Die Komplexitätstheorie verwendet sie, um Probleme danach zu klassifizieren, wie „schwierig“ oder aufwändig sie zu lösen sind. Zu „leichten“ Problemen existiert ein Algorithmus, dessen Laufzeit sich durch ein Polynom beschränken lässt; als „schwer“ gelten Probleme, für die man keinen Algorithmus gefunden hat, der weniger schnell als exponentiell wächst. Man nennt sie (nicht) polynomiell lösbar.

O-Notation – Landau-Symbole

- Nicht alle Symbole gehen direkt auf Landau zurück, einiges wurde erst von Knuth (1976) in dieser Bedeutung definiert.

Notation	Anschauliche Bedeutung
$f = o(g)$ oder $f \in o(g)$	f wächst (viel) langsamer als g
$f = O(g)$ oder $f \in O(g)$	f wächst höchstens genauso schnell wie g
$f \in \Theta(g)$	f wächst genauso schnell wie g
$f \in \Omega(g)$	f wächst mindestens genauso schnell wie g
$f \in \omega(g)$	f wächst schneller als g

O-Notation – Formale Definition

- Formal lassen sich Landau-Symbole mittels Limes superior und Limes inferior folgendermaßen definieren (wobei in der Mehrheit der Beispiele die beiden Limiten zusammen fallen und hier die Grenzen weggelassen wurden, da sie sich aus dem Zusammenhang ergeben):

Notation	Definition	Mathematische Definition
f ∈ o(g)	asymptotisch gegenüber g vernachlässigbar	$\lim \left\lvert \frac{f(x)}{g(x)} \right\rvert = 0$
f ∈ O(g)	asymptotische obere Schranke	$\limsup \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$
f ∈ Θ(g)	asymptotisch scharfe Schranke	$0 < \liminf \left\lvert \frac{f(x)}{g(x)} \right\rvert \leq \limsup \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$
f ∈ Ω(g)	asymptotisch untere Schranke	$\liminf \left\lvert \frac{f(x)}{g(x)} \right\rvert > 0$
f ∈ ω(g)	asymptotisch dominant	$\lim \left\lvert \frac{f(x)}{g(x)} \right\rvert = \infty$

O-Notation – Beispiele I

Notation	Bedeutung	Erklärung	Beispiele für Laufzeiten
$f \in O(1)$	f ist beschränkt	f überschreitet einen konstanten Wert nicht (ist unabhängig vom Wert des Arguments n).	Feststellen, ob eine Binärzahl gerade ist Nachschlagen des n-ten Eintrags in einem Array
$f \in O(\log \log n)$	f wächst doppel-logarithmisch	Bei Basis 2 erhöht sich f um 1, wenn n quadriert wird.	Interpolationssuche im sortierten Feld mit n gleichförmig verteilten Einträgen
$f \in O(\log n)$	f wächst logarithmisch	f wächst ungefähr um einen konstanten Betrag, wenn sich n verdoppelt.	Binäre Suche im sortierten Feld mit n Einträgen

O-Notation – Beispiele II

Notation	Bedeutung	Erklärung	Beispiele für Laufzeiten
f ∈ O(√n)	f wächst wie die Wurzelfunktion	f wächst ungefähr auf das Doppelte, wenn sich n vervierfacht.	Anzahl der Divisionen des naiven Primzahltest (Teilen durch jede ganze Zahl ≤√n)
f ∈ O(n)	f wächst linear	f wächst ungefähr auf das Doppelte, wenn sich n verdoppelt.	Suche im unsortierten Feld mit n Einträgen (Bsp. lineare Suche)
f ∈ O(n log n)	f hat super-lineares Wachstum		Vergleichsbasierte Algorithmen zum Sortieren von n Zahlen (Mergesort, Heapsort)
f ∈ O(n^2)	f wächst quadratisch	f wächst ungefähr auf das Vierfache, wenn sich n verdoppelt.	Einfache Algorithmen zum Sortieren von n Zahlen (Selectionsort)

O-Notation – Beispiele III

Notation	Bedeutung	Erklärung	Beispiele für Laufzeiten
f ∈ O(n^m)	f wächst polynomiell	f wächst ungefähr auf das 2^m-fache, wenn sich n verdoppelt.	„Einfache“ Algorithmen“
f ∈ O(2^n)	f wächst exponentiell	f wächst ungefähr auf das Doppelte, wenn sich n um 1 erhöht.	Erfüllbarkeitsproblem der Aussagenlogik mittels erschöpfender Suche
f ∈ O(n!)	f wächst faktoriell	f wächst ungefähr aus das (n+1)-fache, wenn sich n um 1 erhöht.	Problem des Handlungsreisenden (mit erschöpfender Suche)

O-Notation - Notationsfallen

Oft wird in der Mathematik bei der Landau-Notation das Gleichheitszeichen verwendet. Es handelt sich dabei aber um eine rein symbolische Schreibweise und nicht um eine Gleichheitsaussage, auf die beispielsweise die Gesetze der Transitivität oder der Symmetrie anwendbar sind: Eine Aussage f(x)=O(g(x)) wie ist **keine** Gleichung und keine Seite ist durch die andere bestimmt. Aus f1(x)=O(g(x)) und f2(x)=O(g(x)) folgt nicht, dass f1 und f2 gleich sind. Genauso wenig kann man aus f(x)=O(g1(x)) und f(x)=O(g2(x)) schließen, dass O(g1(x)) und O(g2(x)) dieselbe Klasse sind oder die eine in der anderen enthalten ist.

Tatsächlich handelt es sich bei O(g(x)) um eine Menge, welche alle diejenigen Funktionen enthält, welche höchstens so schnell wachsen wie g(x). Die Schreibweise $f(x) \in O(g(x))$ ist also formal korrekt.

Die Notation mit dem Gleichheitszeichen wie in f=O(g) wird trotzdem in der Praxis ausgiebig genutzt. Beispielsweise soll der Ausdruck $f(n) = h(n)+ \Theta(g(n))$ besagen, dass es Konstanten c1 und c2 gibt, sodass

$$h(n)+c1.g(n) \leq f(n) \leq h(n) + c2.g(n)$$

für hinreichend große n gilt.

3. Komplexität

FHDW
Fachhochschule der Wirtschaft

Komplexität - Allgemeines

- Die **Komplexitätstheorie** als Teilgebiet der theoretischen Informatik befasst sich mit der Komplexität algorithmisch behandelbarer Probleme auf verschiedenen formalen Rechnermodellen.
- Die Komplexität von Algorithmen wird in deren Ressourcenverbrauch gemessen, meist Rechenzeit oder Speicherplatzbedarf, manchmal auch speziellere Maße wie die Größe eines Schaltkreises oder die Anzahl benötigter Prozessoren bei parallelen Algorithmen.
- Die Komplexität eines Problems ist wiederum die Komplexität desjenigen Algorithmus, der das Problem mit dem geringstmöglichen Ressourcenverbrauch löst.

Komplexität – Berechnungsprobleme als Abbildung I

- Neben Entscheidungsproblemen betrachtet man auch Berechnungsprobleme. Ein solches erfordert eine Antwort, die die Problemlösung beschreibt. Das Multiplikationsproblem beispielsweise stellt sich in der Praxis meist als Berechnungsproblem: Man will das Produkt zweier Zahlen ermitteln. Man versteht ein Berechnungsproblem also als eine Abbildung aus einem Definitionsbereich in einen Lösungsraum, im Fall der Multiplikation von natürlichen Zahlen also als Abbildung
  N x N -> N: (a,b) -> a*b

- Ein anderes Beispiel ist das Problem des Handlungsreisenden. Hier sucht man nach der optimalen Reihenfolge, in der man gegebene Orte besucht, wobei die Gesamtlänge der Route minimal sein soll.

- Viele Optimierungsprobleme sind von großer praktischer Bedeutung. Für die Definition der meisten Komplexitätsklassen wird jedoch die Formulierung durch Entscheidungsprobleme bevorzugt.

Komplexität – Berechnungsprobleme als Abbildung II

- Eine wichtige Unterkategorie der Berechnungsprobleme stellen die Optimierungsprobleme dar. Bei Optimierungsproblemen besteht der funktionale Zusammenhang aus der Forderung, das Maximum bzw. Minimum einer gegebenen Kostenfunktion über alle möglichen Lösungen des Problems zu bestimmen.
- Beim Problem des Handlungsreisenden wäre also die Länge der optimalen Route zu berechnen.

Komplexität – Probleminstanzen I

- Eine Probleminstanz ist nicht mit dem Problem selbst zu verwechseln. Ein Problem stellt in der Komplexitätstheorie eine allgemeine Fragestellung, eine Schablone, dar. Eine Instanz des Problems ist dann eine vollständige Fragestellung, welche die richtige Antwort (ja bzw. nein im Fall eines Entscheidungsproblems) festlegt.
- Eine Instanz des Problems des Handlungsreisenden könnte zum Beispiel die Frage sein, ob eine Route durch die Landeshauptstädte Deutschlands mit einer maximalen Länge von 2000 km existiert. Die Entscheidung über diese Route hat jedoch nur begrenzten Wert für andere Probleminstanzen, wie etwa eine Rundreise durch die Sehenswürdigkeiten Mailands. In der Komplexitätstheorie interessiert man sich daher für Aussagen, die unabhängig von konkreten Instanzen sind.

Komplexität – Probleminstanzen II

- Ein Problem wird so allgemein formuliert, dass es eine unendliche Menge von Probleminstanzen definiert, denn es ist nicht sinnvoll, nach der Komplexität einer endlichen Menge von Instanzen zu fragen; ein Programm könnte eine Liste von vorgefertigten Antworten enthalten und nur durch Tabellenzugriff die richtige Lösung ausgeben, was den Aufwand für die Ermittlung der Antworten nicht widerspiegelt. Interessant wird es erst, wenn eine unendliche Menge von Instanzen gegeben ist und man einen Algorithmus finden will, der für jede Instanz die richtige Antwort berechnet.

Komplexität – Problempräsentationen

- Als formale Sprachen werden Probleme und deren Instanzen über abstrakten Alphabeten definiert. Häufig wird das binäre Alphabet mit den Symbolen 0 und 1 gewählt, da dies der Verwendung von Bits bei modernen Rechnern am nächsten kommt. Eingaben werden dann durch Alphabet-Symbole kodiert. An Stelle von mathematischen Objekten wie Graphen verwendet man möglicherweise eine Bitfolge, die der Adjazenz-Matrix des Graphen entspricht, an Stelle von natürlichen Zahlen zum Beispiel deren Binärdarstellung.
- Auch wenn sich Beweise komplexitätstheoretischer Aussagen in der Regel konkreter Repräsentationen der Eingabe bedienen, versucht man Aussagen und Betrachtung unabhängig von Repräsentationen zu halten. Dies kann etwa erreicht werden, indem man sicherstellt, dass die gewählte Repräsentation bei Bedarf ohne allzu großen Aufwand in eine andere Repräsentation transformiert werden kann, ohne dass sich hierdurch die Berechnungsaufwände insgesamt signifikant verändern. Um dies zu ermöglichen, ist unter anderem die Auswahl eines geeigneten universellen Maschinenmodells von Bedeutung

Komplexität – Problemgröße

- Hat man ein Problem formal definiert (zum Beispiel das Problem des Handlungsreisenden in Form eines Graphen mit Kantengewichten), so möchte man Aussagen darüber treffen, wie sich ein Algorithmus bei der Berechnung der Problemlösung in Abhängigkeit von der Schwierigkeit des Problems verhält. Im Allgemeinen sind bei der Beurteilung der Schwierigkeit des Problems viele verschiedene Aspekte zu betrachten. Dennoch gelingt es häufig, wenige skalare Größen zu finden, die das Verhalten des Algorithmus im Hinblick auf den Ressourcenverbrauch maßgeblich beeinflussen. Diese Größen bezeichnet man als die Problemgröße. In aller Regel entspricht diese der Eingabelänge (bei einer konkret gewählten Repräsentation der Eingabe).
- Man untersucht nun das Verhalten des Algorithmus in Abhängigkeit von der Problemgröße. Die Komplexitätstheorie interessiert sich für die Frage: *Wie viel* Mehrarbeit ist für wachsende Problemgrößen notwendig? Steigt der Aufwand (in Relation zur Problemgröße) zum Beispiel linear, polynomial, exponentiell oder gar überexponentiell?

Komplexität – Analyse der Problemgröße

- Auch innerhalb einer Problemgröße lassen sich verschiedene Verhaltensweisen von Algorithmen beobachten. So hat das Problem des Handlungsreisenden für die 16 deutschen Landeshauptstädte dieselbe Problemgröße **n=16** wie das Finden einer Route durch 16 europäische Hauptstädte. Es ist keineswegs zu erwarten, dass ein Algorithmus unter den unterschiedlichen Bedingungen (selbst bei gleicher Problemgröße) jeweils gleich gut arbeitet. Da es jedoch in der Regel unendlich viele Instanzen gleicher Größe für ein Problem gibt, gruppiert man diese zumeist grob in drei Gruppen: bester, durchschnittlicher und schlechtester Fall. Diese stehen für die Fragen:

  1. Bester Fall: Wie arbeitet der Algorithmus (in Bezug auf die in Frage stehende Ressource) im günstigsten Fall?
  2. Durchschnittlicher Fall: Wie arbeitet der Algorithmus durchschnittlich (wobei die zugrunde gelegte Verteilung für die Berechnung eines Durchschnitts nicht immer offensichtlich ist)?
  3. Schlechtester Fall: Wie arbeitet der Algorithmus im schlimmsten Fall?

Komplexität – Schranken für Probleme I

- Die Betrachtung bester, schlechtester und durchschnittlicher Fälle bezieht sich stets auf eine feste Eingabelänge. Auch wenn die Betrachtung konkreter Eingabelängen in der Praxis von großem Interesse sein kann, ist diese Sichtweise für die Komplexitätstheorie meist nicht abstrakt genug. Welche Eingabelänge als groß oder praktisch relevant gilt, kann sich aufgrund technischer Entwicklungen sehr schnell ändern. Es ist daher gerechtfertigt, das Verhalten von Algorithmen in Bezug auf ein Problem gänzlich unabhängig von konkreten Eingabelängen zu untersuchen. Man betrachtet hierzu das Verhalten der Algorithmen für immer größer werdende, potentiell unendlich große Eingabelängen. Man spricht vom asymptotischen Verhalten des jeweiligen Algorithmus.
- Bei dieser Untersuchung des asymptotischen Ressourcenverbrauchs spielen untere und obere Schranken eine zentrale Rolle. Man möchte also wissen, welche Ressourcen für die Entscheidung eines Problems mindestens und höchstens benötigt werden.

Komplexität – Schranken für Probleme II

- Für die Komplexitätstheorie sind die unteren Schranken von besonderem Interesse: Man möchte zeigen, dass ein bestimmtes Problem *mindestens* einen bestimmten Ressourcenverbrauch beansprucht und es folglich keinen Algorithmus geben kann, der das Problem mit geringeren Ressourcen entscheidet. Solche Ergebnisse helfen, Probleme nachhaltig bezüglich ihrer Schwierigkeit zu separieren. Jedoch sind bisher nur vergleichsweise wenige aussagekräftige untere Schranken bekannt. Der Grund hierfür liegt in der Problematik, dass sich Untersuchungen unterer Schranken stets auf alle denkbaren Algorithmen für ein Problem beziehen müssen; also auch auf Algorithmen, die heute noch gar nicht bekannt sind.
- Im Gegensatz dazu gelingt der Nachweis oberer Schranken in der Regel durch die Analyse konkreter Algorithmen. Durch den Beweis der Existenz auch nur eines Algorithmus, der die obere Schranke einhält, ist der Nachweis bereits erbracht.

Komplexität – Kostenfunktionen

- Zur Analyse des Ressourcenverbrauchs von Algorithmen sind geeignete Kostenfunktionen zu definieren, welche eine Zuordnung der Arbeitsschritte des Algorithmus zu den verbrauchten Ressourcen ermöglichen. Um dies tun zu können, muss zunächst festgelegt werden, welche Art von Arbeitsschritt einem Algorithmus überhaupt erlaubt ist. Diese Festlegung erfolgt in der Komplexitätstheorie über abstrakte Maschinenmodelle – würde man auf reale Rechnermodelle zurückgreifen, so wären die gewonnenen Erkenntnisse bereits in wenigen Jahren überholt. Der Arbeitsschritt eines Algorithmus erfolgt in Form einer Befehlsausführung auf einer bestimmten Maschine. Die Befehle, die eine Maschine ausführen kann, sind dabei durch das jeweilige Modell streng limitiert. Darüber hinaus unterscheiden sich verschiedene Modelle etwa in der Handhabung des Speichers und in ihren Fähigkeiten zur parallelen Verarbeitung, d. h. der gleichzeitigen Ausführung mehrerer Befehle. Die Definition der Kostenfunktion erfolgt nun durch eine Zuordnung von Kostenwerten zu den jeweils erlaubten Befehlen.

Komplexität – Kostenmaße

- Häufig wird von unterschiedlichen Kosten für unterschiedliche Befehle abstrahiert und als Kostenwert für eine Befehlsausführung immer 1 gesetzt. Sind auf einer Maschine beispielsweise Addition und Multiplikation die erlaubten Operationen, so zählt man für jede Addition und jede Multiplikation, die im Laufe des Algorithmus berechnet werden müssen, den Kostenwert von 1 hinzu. Man spricht dann auch von einem *uniformen Kostenmaß*. Ein solches Vorgehen ist dann gerechtfertigt, wenn sich die erlaubten Operationen nicht gravierend unterscheiden und wenn der Wertebereich, auf dem die Operationen arbeiten, nur eingeschränkt groß ist.

- Sollten sich die möglichen Operanden im Laufe eines Algorithmus tatsächlich so gravierend unterscheiden, so muss ein realistischeres Kostenmaß gewählt werden. Häufig wählt man dann das *logarithmische Kostenmaß*. Der Bezug auf den Logarithmus ergibt sich daraus, dass sich eine natürliche Zahl **n** im Wesentlichen durch $\boldsymbol{log_2(n)}$ viele Binärziffern darstellen lässt. Man wählt zur Repräsentation der Operanden Binärziffern aus und definiert die erlaubten booleschen Operationen.

4. Arrays und Listen

FHDW
Fachhochschule der Wirtschaft

Arrays – Allgemeines

- Ein Array repräsentiert eine feste Zahl von Elementen eines bestimmten Typs. Die Elemente in einem Array werden immer in einem zusammenhängenden Speicherbereich gespeichert, wodurch sehr schnell auf sie zugegriffen werden kann.
  Ein Array wird durch eckige Klammern nach dem Typ des Elements definiert. Die folgende Zeile deklariert ein Array mit fünf Zeichen:
  char [ ] vowels = new char[5];

- Eckige Klammern indexieren das Array, wodurch man auf einzelne Elemente des Arrays zugreifen kann
  vowels[0]= ‘a‘; vowels[1]= ‘e‘; vowels[2] = ‘i‘;
  vowels[3] = ‘o‘; vowels[4] = ‘u‘;

- Die Array-Indizes beginnen bei 0 !!!

Arrays – Schleifen

- Die Länge des Arrays wird mit dem Zusatz length dargestellt. Die einzelnen Zeichen können damit mit nachfolgender Schleife ausgegeben werden:

```
for (int i = 0; i < vowels.Length; ++i)
        System.out.println(vowels[i]);
```

- Arrays implementieren das Interface Iterable<T>, daher kann man es auch eleganter schreiben als:

```
for(char c : vowels)
        System.out.println(c);
```

- Die Grenzen im Array sind wichtig, das das Laufzeitsystem dies überprüft und im Fehlerfall eine IndexOutOfBounds-Exception auslöst:

```
vowels[5]= ‘y‘;          //Laufzeitfehler, Index unzulässig
```

Arrays – Initialisierung

- Mit einem Array-Initialisierungsausdruck kann ein Array in einem Schritt deklariert und initialisiert werden:
  char [ ] vowels = new char[ ] = {‘a‘, ‘e‘, ‘i‘, ‘o‘, ‘u‘ };
  oder einfach
  char [ ] vowels = {‘a‘, ‘e‘, ‘i‘, ‘o‘, ‘u‘ };

- Alle Arrays erben von der Klasse java.util.Arrays, in der Methoden und Eigenschaften definiert sind. Dazu gehören Instanzeigenschaften wie l*ength* sowie statische Methoden wie:
  - Elemente unabhängig vom Array-Typ lesen bzw. Schreiben
  - Ein Array sortieren
  - Ein Array kopieren
  - Ein sortiertes Array durchsuchen (indexOf, Find, findIndex, usw.)

Arrays – Standard-Elementinitialisierung

- Beim Erstellen eines Arrays werden dessen Elemente immer durch Standardwerte vorinitialisiert. Der Standardwert ist das Ergebnis eines bitweise Löschen seines Speicherbereichs:
  int [ ] a = new int[1000];
  System.out.println(a[123]); // 0

- Referenztypen werden auf *null* initialisiert.

- Ein Array selbst ist immer ein Referenztyp-Objekt – unabhängig davon, welchen Typ seine Elemente haben. Beispielsweise ist folgendes zulässig:
  int [ ] a = null;

Arrays – Ausgabe von Arrays

- Wir wollen eine Array aus Namen möglichst elegant auf dem Bildschirm ausgeben:
  String [ ] names = { "Hans", "Klaus", "Daniel", " Bastian" };
- Da die Methode toString auf einem Array nicht sinnvoll implementiert ist, liefert ein einfacher Aufruf von println nicht das gewünschte Resultat:
  System.out.println(names); // [java.lang.String;@10b62c9]
- Die statische Methode *Arrays.toString(array)* liefert für unterschiedliche Arrays eine akzeptable String-Darstellung des Arrays (durch eckige Klammern eingeschlossen und durch Kommata getrennt):
  System.out.println(Arrays.toString(names)); // [Hans, Klaus, Daniel, Bastian]

Arrays – Erweiterte for-Schleife

- Ein Array lässt sich auch ohne explizite Angabe der Indizes durchlaufen. Die Syntax ist:

  for (Typ Bezeichner : Array)

  Beispiel:

  String [ ] names = { "Hans", "Klaus", "Daniel", " Bastian" };
  for (String s: names)
  System.out.println(s);

- Rechts vom Doppelpunkt lässt sich auf die Schnelle ein Array aufbauen, über welches das erweiterte *for* dann laufen kann :

  for (int prime: new int [ ] {2,3,5,7,11,13, 17, 19,23, 29, 31})
  System.out.println(prime);

- Möglich ist auch eine Objekterzeugung in einem Methodenaufruf:

  for (String name : Arrays.asList("Hans", "Klaus", "Daniel")
  System.out.println(name);

  Arrays.asList erzeugt kein Array als Rückgabe, sondern baut aus der variable Argumentliste eine Sammlung auf, die von einem speziellen Typ *Iterable* ist.

Arrays – Arrays sortieren

- Die Klasse java.util.Arrays enthält die statische Methode *sort* zum Sortieren von Elementen im Array. Bei primitiven Elementen (außer boolean) gibt es keine Besonderheiten, da sie eine natürliche Ordnung haben. Beispiel:
  int [ ] numbers = {-1, 3, -10, 9, 3};
  String [ ] names = {“Xanten“, “Alpen“, “Wesel“};
  Arrays.sort(numbers);
  Arrays.sort(names);

- Besteht das Array aus Objektreferenzen, müssen die Objekte vergleichbar sein. Das gelingt entweder mit einem Extra-Comparator oder die Klassen implementieren die Schnittstelle Comparable wie zum Beispiel Strings.

- Zusätzlich enthält die Klasse Arrays die Methode parallelSort, die das Sortieren auf mehrere Threads verteilt:
  Arrays.parallelSort(numbers);
  Arrays.parallelSort(number, 1, 3); // Sortieren eines Teilarrays

Arrays – Arrays vergleichen

- Die statische Methode *Arrays.equals( )* vergleicht, ob zwei Arrays den gleichen Inhalt besitzen; dazu ist die überladene Methode für alle wichtigen Typen definiert. Wenn zwei Arrays die gleichen Inhalte besitzen, so ist die Rückgabe der Methode true, sonst false.
  Die beiden Arrays müssen schon die gleiche Anzahl von Elementen haben, sonst liefert die Methode false.
  Im Fall von Objekt-Arrays nutzt *Arrays.equals(…)* nicht die Identitätsprüfung per ==, sondern die Gleichheit per *equals(…)*.
  Beispiel:

```
int[ ] array1 = {1, 2, 3, 4};
int[ ] array2 = {1, 2, 3, 4};
System.out.println(Arrays.equals(array1, array2));  // true
```

- Ein Vergleich von Teil-Arrays ist leider nicht möglich.

Listen – Allgemeines

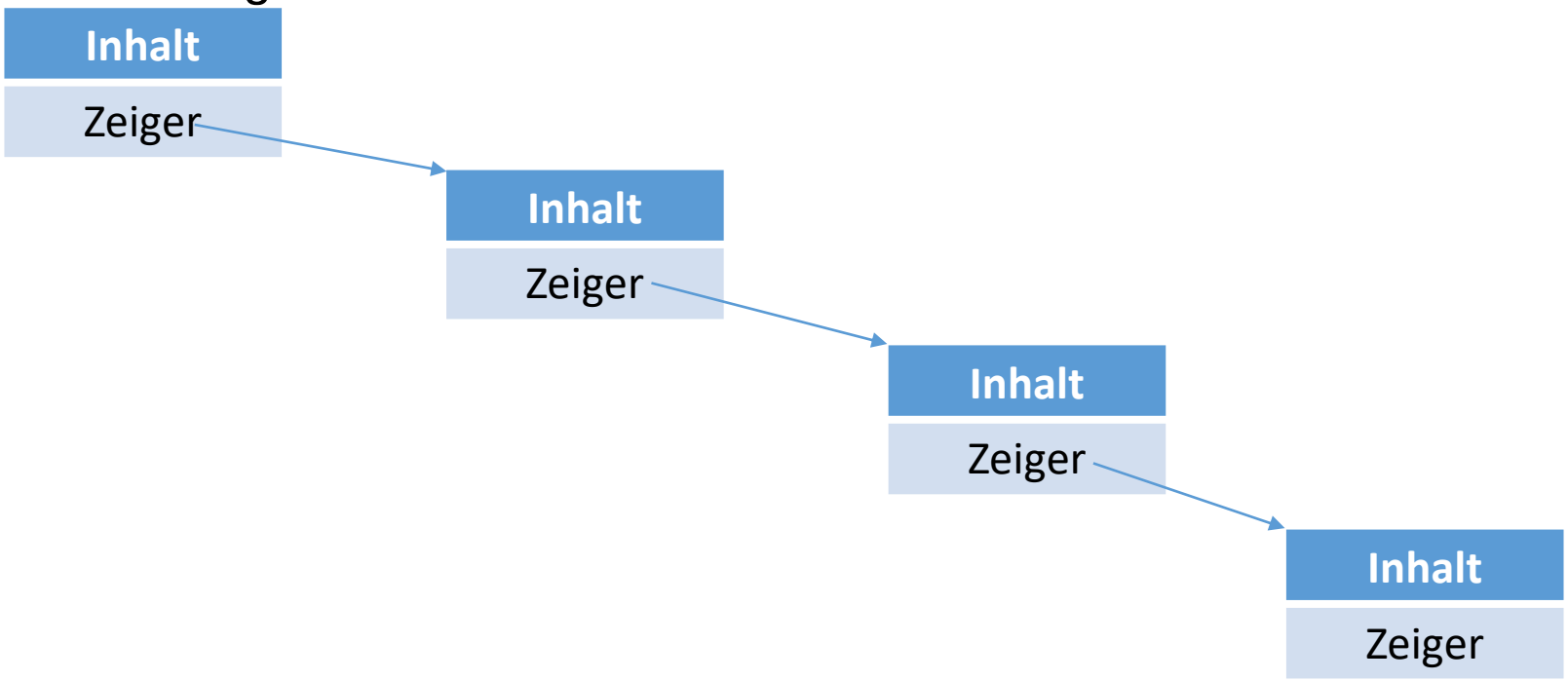
- Eine Liste steht für eine Sequenz, bei der die Elemente eine feste Reihenfolge besitzen. Die *List*-Schnittstelle schreibt Verhalten vor, die alle konkreten Listen implementieren. Interessante Realisierung der *List*-Schnittstelle sind:
  - ArrayList – eine Liste auf Basis eines Feldes
  - LinkedList – eine Liste durch verkettete Elemente
  - CopyOnWriteArrayList – eine schnelle Liste, optimal für häufige nebenläufige Lesezugriffe
- Die Methoden zum Zugriff über die gemeinsame Schnittstelle sind immer die gleichen. So ermöglicht jede Liste einen Punktzugriff über *get(index)* und jede Liste kann alle gespeicherten Elemente sequentiell über einen Iterator hergeben. Die Realisierungen unterscheiden sich in Eigenschaften wie Performance, Speicherplatzbedarf oder Nebenläufigkeit.

Listen – ArrayList versus LinkedList

- Eine *ArrayList* speichert Elemente in einem internem Array. *LinkedList* dagegen speichert die Elemente in einer (doppelt-) verketteten Liste und realisiert die Verkettung in einem eigenen Hilfselement. Es ergeben sich unterschiedliche Einsatzgebiete:
  - Da *ArrayList* intern ein Array benutzt, ist er Zugriff auf ein spezielles Element über die Position in der Liste sehr schnell. Eine *LinkedList* muss aufwändig durchsucht werden.
  - Die verkettete Liste ist dagegen deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden; hier muss einfach nur die Verkettung der Hilfsobjekte an einer Stelle verändert werden. Bei einer *ArrayList* bedeutet das viel Arbeit, es sei denn, das Element kann am Ende gelöscht werden. Andernfalls müssen alle Elemente verschoben werden.
  - Bei einer *ArrayList* kann die Größe des internen Feldes zu klein werden, um ein neues Element aufzunehmen. Dann muss die Laufzeitumgebung ein neues größeres Feld-Objekt anlegen und alle Elemente kopieren.

Listen – Wiederholung: einfach verkettete Liste I

- Eine einfach-verkettete Liste besteht aus einer Reihe von Knoten. Jeder Knoten enthält einen Inhalt (generischer Typ) und einen Zeiger auf den nächsten Knoten. Der Zeiger im letzten Knoten ist null. Ein Knoten wird visualisiert durch ein Rechteck, das oben genannte Daten beinhaltet.



Listen – Wiederholung: einfach verkettete Liste II

- Wie ändert sich das Bild, falls ein Knoten
  - am Anfang
  - in der Mitte
  - am Ende

  eingefügt wird?

- Wie ändert sich das Bild, falls ein Knoten
  - am Anfang
  - in der Mitte
  - am Ende

  gelöscht wird?

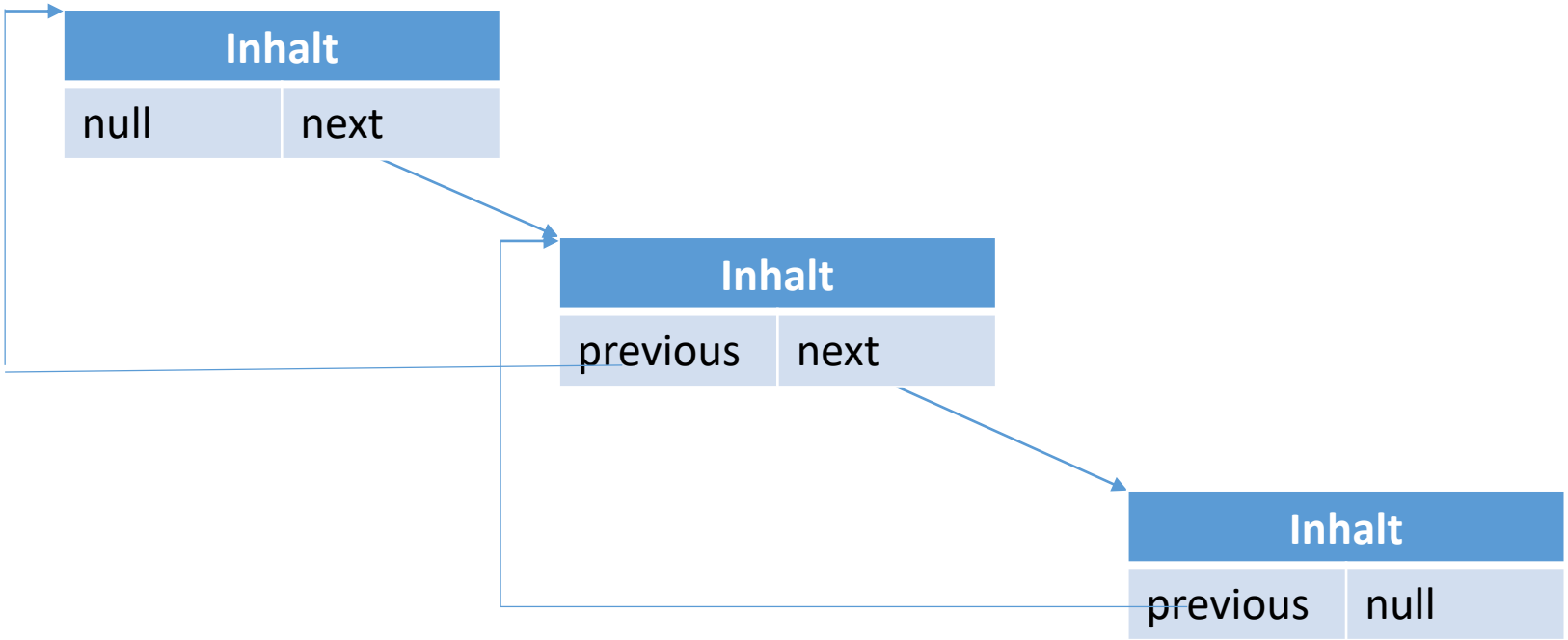
Listen – Wiederholung: einfach verkettete Liste III

```
public class VerketteteListe {
    private class Node  {
        int key;
        Node next;
        public Node(int k) {  key = k; }
    }

    Node root;
    public void add(int k) {
        if (root == null)
            root = new Node( k);
        else {
            Node cn = root;
            while (cn.next != null)
                cn = cn.next;
            cn.next = new Node(k);
        }
    }
```

Listen – Wiederholung: doppelt verkettete Liste I

- Eine doppelt-verkettete Liste besteht aus einer Reihe von Knoten. Jeder Knoten enthält einen Inhalt (generischer Typ) und zwei Zeiger, auf den vorhergehenden (previous) und auf den nächsten (next) Knoten.



Listen – Wiederholung: doppelt verkettete Liste II

- Wie ändert sich das Bild, falls ein Knoten
  - am Anfang
  - in der Mitte
  - am Ende

  eingefügt wird?

- Wie ändert sich das Bild, falls ein Knoten
  - am Anfang
  - in der Mitte
  - am Ende

  gelöscht wird?

Listen – Eigenschaften der Schnittstelle I

- Die Schnittstelle *List* schreibt das allgemeine Verhalten für Listen vor.
- Hinzufügen und Setzen von Elementen
  - Die *add( )*-Methode fügt neue Elemente an, wobei eine Position die Einfügestelle bestimmen kann.
  - Die *addall( )*-Methode fügt fremde Elemente einer anderen Sammlung in die Liste ein.
  - Die *set( )*-Methode setzt ein Element an eine bestimmte Stelle, überschreibt das ursprüngliche Element und verschiebt es auch nicht wie die add( )-Methode.
  - Die *size( )*-Methode liefert die Anzahl der Elemente in der Datenstruktur.
- Listen zu Feldern
  - Die *toArray( )*-Methode wandelt eine Liste in ein Array um.

Listen – Eigenschaften der Schnittstelle II

- Positionsangeben und Suchen
  - Die *isEmpty( )*-Methode liefert die Information, ob die Liste leer ist
  - Mit der *get( )*-Methode kann man eine Element an einer speziellen Stelle abfragen.
  - Die *contains( )*-Methode überprüft, ob ein übergebenes Argument in der Sammlung vorhanden ist. Analog die Methode containsAll( ).
  - Wie bei Strings liefert *indexOf( )* und *lastIndexOf( )* die Fundpositionen.
- Löschen von Elementen
  - Die *remove( )*-Methode löscht aus einer Liste ein Element.
  - Die *removeAll( )*-Methode löscht aus einer Liste alle Elemente eines übergebenen Arrays.
  - Die *clear( )*-Methode löscht eine Liste.

Listen – Einfache Beispiele

```
import java.util.*;

List<String> list1 = new ArrayList<>();
list1.add("Eva");
list1.add(0, "Charisma");
list1.add("Pallas");
System.out.println(list1);                  // [Charisma, Eva, Pallas]
List<String> list2 = Arrays.asList("Tina", „Ute“);
list1.addAll(3, list2);
list1.add("XXX");
System.out.println(list1);                  // [Charisma, Eva, Pallas, Tina, Ute, XXX]
list1.set(5, "Eva");
System.out.println(list1);                  // [Charisma, Eva, Pallas, Tina, Ute, Eva]
System.out.println(list1.size());           // 6
System.out.println("Tina in Liste: " + list1.contains("Tina"));   //Tina in Liste: true
Object o = list1.get(1);
System.out.println(o);                      // Eva
System.out.println("Eva-Index "+ list1.indexOf("Eva"));     //Eva-Index: 1
list1.remove(1);
System.out.println(list1);                  // [Charisma, Pallas, Tina, Ute, Eva]
```

Listen – Konstruktoren

- ArrayList
  - *ArrayList( )* erzeugt eine leere Liste mit positiver Anfangskapazität (10)
  - *ArrayList(int initialCap)* erzeugt eine Liste für initialCap Elemente.
  - *ArrayList(Collection<? extends E> c)* kopiert alle Elemente der Collection c in das neue Array-Objekt.
- LinkedList
  - *LinkedList( )* erzeugt eine neue leere Liste.
  - *LinkedList(Collection<? extends E> c)* kopiert alle Elemente der Collection c in die neue verkettete Liste.
- Ergänzungen bei LinkedList
  - Hier gibt es weitere Hilfsmethoden: addFirst( ), getFirst( ), getLast( ), removeFirst( ) und removeLast( ).

5. Rekursive Algorithmen

FHDW
Fachhochschule der Wirtschaft

Rekursives – Allgemeines

- Knuth schrieb 1974:
  „... the transformation from recursion to iteration is one of the most fundamental concepts of computer science.“
  - *Das war richtig aus Sicht der SW-Technik in den Jahren, als viele Programmiersprachen Rekursion nicht unterstützten.*
  - *Aber: mittlerweile unterstützen alle Programmiersprachen die Rekursion*
  - *Aber: die Einführung höheren Datentypen wie verketteten Listen, Binärbäume und Schlangen führte zur Aufwertung der Rekursion.*
- *Viele Probleme, die durch mehrfache Rekursion gekennzeichnet sind, haben eine exponentielle Komplexität; hier kann auch die Iteration die Komplexität nicht reduzieren, sie steigert aber die Effizienz des Arbeitens.*

Rekursives – vollständige Induktion

- Die Rekursion wurde aus der Mathematik übernommen, wo mit ihrer Hilfe unendliche Mengen durch endliche Aussagen charakterisiert werden. Das Musterbeispiel sind die natürlichen Zahlen, die durch die Axiome von Peano gekennzeichnet sind:
  - *0 ist eine natürliche Zahl (für viele beginnen sie bei 1)*
  - *Mit jeder natürlichen Zahl n ist auch ihr Nachfolger n+1 eine natürliche Zahl*
  - *Enthält eine Menge die Null und mit jeder Zahl auch ihren Nachfolger, so enthält sie alle natürlichen Zahlen.*
- *Das darauf aufbauende Induktionsprinzip oder auch Prinzip der vollständigen Induktion ermöglicht Aussagen über unendliche Mengen von natürlichen Zahlen.*

Rekursives – rekursive Funktion

- Eine rekursive Funktion ist gekennzeichnet durch
  - *Eine Anfangsbedingung (oder auch Abbruchbedingung)*
  - *eine rekursiven Aufruf, bei dem eine Aussage über n+1 aus einer Aussage über n abgeleitet wird*
- *Klassisches Beispiel ist die Fakultätsfunktion:*
  fact(n)=(n+1)!
  für n=0 ist sie definiert als 1
  *für n>0 ist sie definiert als (n-1)!*n*
- *Hier der Beispiel-Code in C/Java*

```
static int fact(int n) {
    if(n>0){
        return n * fact(n-1);
    }
    return 1;
}
```

Rekursives – die Fakultätsfunktion

Auswertung der Funktion `fact` zur Laufzeit:

```
fact(5) = 5 * fact(4)
        = 5 * 4 * fact(3)
        = 5 * 4 * 3 * fact(2)
        = 5 * 4 * 3 * 2 * fact(1)
        = 5 * 4 * 3 * 2 * 1 * fact(0)
        = 5 * 4 * 3 * 2 * 1 * 1
        = 5 * 4 * 3 * 2 * 1
        = 5 * 4 * 3 * 2
        = 5 * 4 * 6
        = 5 * 24
        = 120
```

Rekursives – Beispiele I

- Die Fibonacci-Funktion

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

- *Der Binomialkoeffizient*

$$\binom{n}{k}=\begin{cases} 1 & \text{wenn } k=0 \text{ oder } k=n \\ \binom{n}{k-1}+\binom{n-1}{k-1} & \text{sonst} \end{cases}$$

- *Die ggT-Funktion*

$$ggT(n,m)=\begin{cases} n & \text{wenn } m=0 \\ ggT(m,\ n \bmod m) & \text{sonst} \end{cases}$$

Rekursives – Beispiele II

- Die Grundrechenarten für natürliche Zahlen lassen sich rekursiv beschreiben, indem über das zweite Argument induktiv gerechnet wird:

- Addition ADD(x,y)=x+y

  ADD(x,0) = x

  ADD(x,y+1)=x+y+1=ADD(x,y)+1

- Multiplikation MULT(x,y)=x*y

  MULT(x,0)=0

  MULT(x,y+1)=x*(y+1)=x*y+x=MULT(x,y)+x

- Potenz POW(x,y)=x^y

  POW(x,0)=1

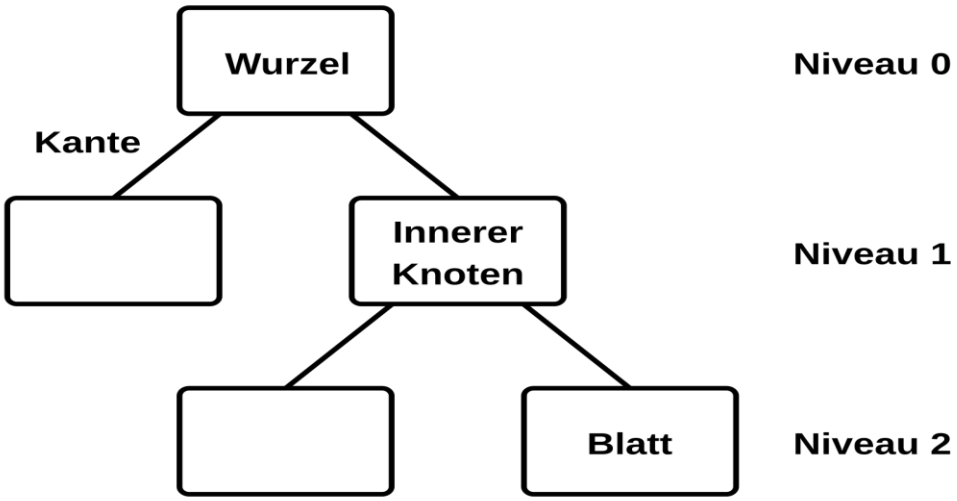
  POW(x,y+1)=POW(x,y)*x

6. Bäume

FHDW
Fachhochschule der Wirtschaft

Bäume – Allgemeines

- Ein **Baum** (engl. **tree**) ist eine spezielle Art von **Datenstruktur**, die eine hierarchische Anordnung von Elementen beschreibt. Ein Baum besteht aus **Knoten** (engl. **nodes**), die miteinander durch **Kanten** (engl. **edges**) verbunden sind. Er wird oft verwendet, um Beziehungen oder Strukturen abzubilden, bei denen eine hierarchische Organisation vorliegt, wie etwa bei Stammbaumstrukturen, Verzeichnisbäumen auf einem Computer oder Entscheidungsbäumen.



Bäume – Eigenschaften

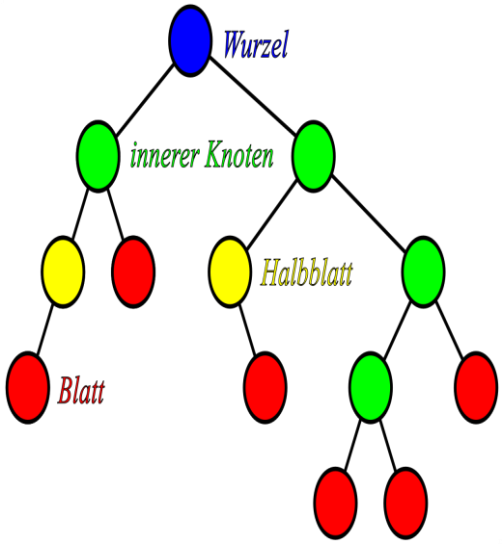
- **Wurzelknoten (root)**: Der oberste Knoten eines Baums. Es gibt genau einen Wurzelknoten, von dem alle anderen Knoten ausgehen.
- **Knoten (nodes)**: Jeder Punkt in der Baumstruktur. Jeder Knoten kann mehrere Kindknoten haben.
- **Eltern- und Kindknoten**: Ein Knoten, der mit einem anderen Knoten verbunden ist, ist entweder ein **Elternknoten** (parent) oder ein **Kindknoten** (child). Ein Elternknoten kann mehrere Kinder haben, aber jedes Kind hat genau einen Elternknoten.
- **Blattknoten (leaf)**: Knoten ohne Kindknoten, d. h. die "Enden" des Baums.
- **Tiefe (depth) eines Knotens**: Die Anzahl der Kanten auf dem Weg von dem Knoten bis zur Wurzel.
- **Höhe des Baums (height)**: Die maximale Tiefe der Knoten eines Baums. Es ist vereinbart, dass ein leerer Baum die Höhe -1 hat. Ein Baum, der nur aus der Wurzel besteht, hat die Tiefe 0.

Bäume – Arten von Bäumen

- **Binärbaum**: Ein Baum, bei dem jeder Knoten höchstens zwei Kindknoten hat (linkes und rechtes Kind).
- **Suchbaum (Binary Search Tree, BST)**: Ein spezieller Binärbaum, bei dem die Knoten so angeordnet sind, dass für jeden Knoten gilt: Alle Werte im linken Teilbaum sind kleiner, alle Werte im rechten Teilbaum sind größer.
- **AVL-Baum**: Ein selbstbalancierender Binärbaum, der sicherstellt, dass die Höhe des linken und rechten Teilbaums jedes Knotens sich um höchstens 1 unterscheidet.
- **Rot-Schwarz-Baum**: Ein selbstbalancierender Binärbaum, der die Balance durch Färben der Knoten unter Berücksichtigung gewisser Regeln sicherstellt.
- **Heap**: Ein binärer Baum, bei dem jedes Eltern-Kind-Paar eine bestimmte Reihenfolge einhält (z. B. bei einem Max-Heap ist der Wert des Elternknotens immer größer oder gleich den Werten seiner Kinder, während bei einem Min-Heap der Wert des Elternknotens immer kleiner oder gleich den Werten seiner Kinder ist).

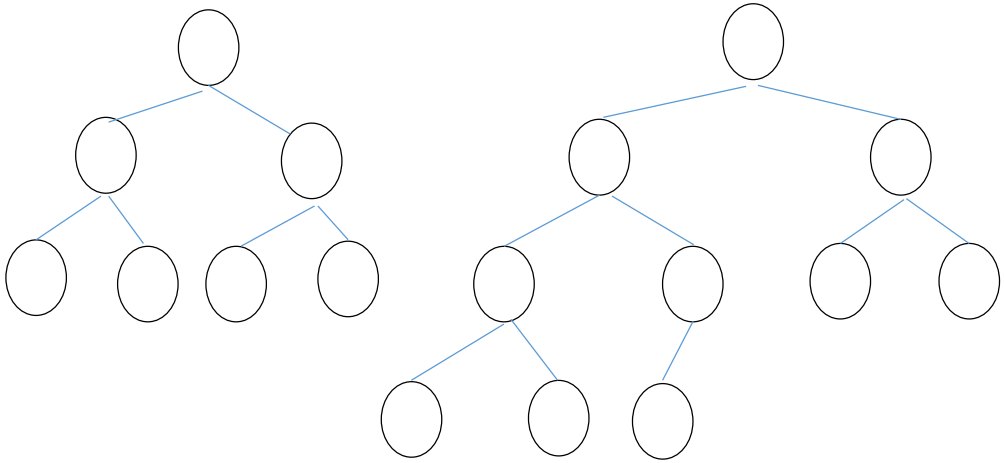
Bäume – Binärbaum I

- Ein Binärbaum ist ein (geordneter) Baum, bei dem jeder Knoten höchstens zwei Kindknoten hat.
- Die beiden Kindern werden linkes und rechtes Kind bezeichnet.
- Hat ein Knoten nur ein Kind, muss es ein linkes oder ein rechtes Kind sein.
- Unter der Höhe versteht man die maximale Anzahl von Kanten auf dem Weg von der Wurzel bis zu einem Blatt. Der Baum im Bild hat die Höhe 4.
- Optional trägt jeder Knoten eine Information – im Bild ist dies die Farbe.
- Optional beinhaltet jeder Knoten einen Vergleichsschlüssel (key).



Bäume – Vollständiger Binärbaum

- Ein vollständiger Binärbaum ist ein Binärbaum, bei der jede Ebene (bis auf die letzte) vollständig gefüllt und die letzte Ebene von links nach rechts gefüllt ist.
- Beispiele:

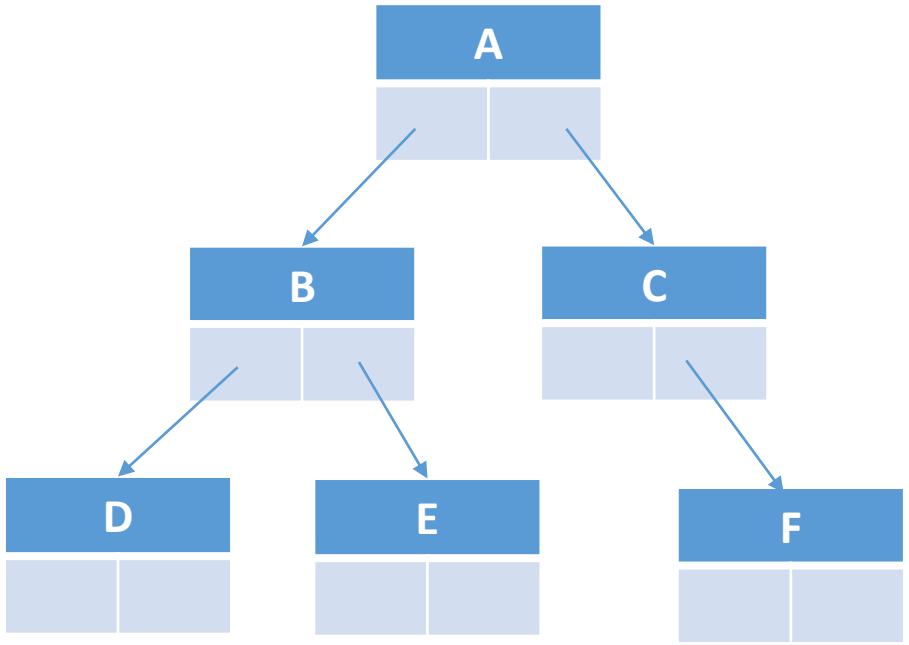


- Ein vollständiger Binärbaum mit N Knoten hat die Höhe $\log_2 N$.
- Bei einer vorgegebenen Knotenzahl sind vollständige Binärbäume solche Binärbäume mit der minimalen Höhe.

Bäume – Implementierung eines Binärbaums I

Implementierung als verkettete Struktur:
Jeder Knoten hat jeweils eine Referenz
für das linke und das rechte Kind.

```
class Node<K, V> {
        K key;
        V value;
        Node<K,V> left;
        Node<K,V> right;
}
```



Es sind im Baum nur die Schlüssel dargestellt.

Bäume – Implementierung eines Binärbaums II

```
public class BinaryTree {
// private Implementierung einer einfachen Node-Klasse
    private class Node {
        int key;
        Node left, right;
        public Node(int k) {          // Konstruktor
            this.key = k;  }
    }

    Node root;
 // Methode zum Hinzufügen eines Schlüssels zum Baum
    public void add(int key) { root = add(key, root); }

    private Node add(int key, Node current) {
        if (current == null) { return new Node(key); }
        if (key < current.key) {
            current.left = add(key, current.left);
        } else if (key > current.key) {
            current.right = add(key, current.right);
        }
        return current;
    }
}
```

Bäume – Implementierung eines Binärbaums III

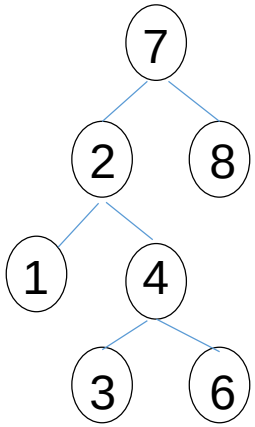
Hier ein kleines Beispiel für die Klasse BinaryTree:

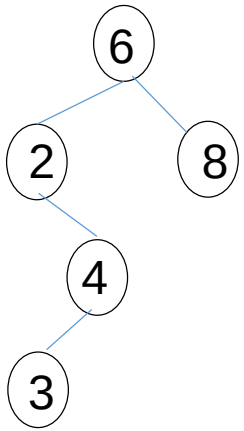
```
 public static void main(String[ ] args) {
        BinaryTree tree = new BinaryTree();
        tree.add(5);
        tree.add(6);
        tree.add(7);
        tree.add(4);
        tree.add(2);
        tree.add(3);
}
```

Wie sieht der Baum nun aus?

Bäume – Binärer Suchbaum I

- Ein binärer **Such**baum ist ein Binärbaum, bei dem für alle Knoten k gilt:
  - Alle Schlüssel im linken Teilbaum sind kleiner als k
  - Alle Schlüssel im rechten Teilbaum sind größer als k.

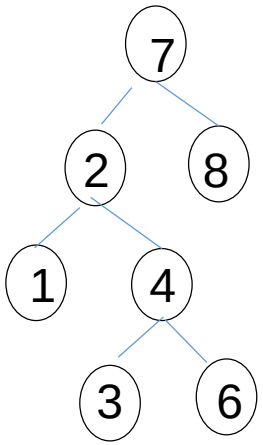




- Das Ziel des binären Suchbaums ist es, die Suche effizient zu gestalten!!

Bäume – Binärer Suchbaum II

- Wie wird ein Knoten gesucht?
  - Wir starten bei der Wurzel und wandern entlang des Baums, wobei wir beachten, dass für einen Entscheidungsknoten k gelten muss:
  - Ist der Schlüssel des gesuchten Knotens kleiner als der von k, so muss der gesuchte Knoten im linken Teilbaum unterhalb von k sein.
  - Ist der Schlüssel des gesuchten Knoten größer als der von k, so muss der gesuchte Knoten im rechten Teilbaum unterhalb von k sein.



Beispiel 6: bei 7 Entscheidung für links

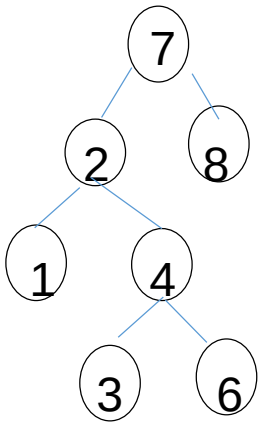
bei 2 Entscheidung für rechts

bei 4 Entscheidung für rechts

bei 6 - > gefunden

Bäume – Binärer Suchbaum III

- Wie wird ein neuer Knoten eingefügt?
- Wir starten bei der Wurzel und wandern entlang des Baums, wobei wir beachten, dass für einen Entscheidungsknoten k gelten muss:
  - Ist der Schlüssel des neuen Knotens kleiner als der von k, so muss der neue Knoten im linken Teilbaum unterhalb von k sein.
  - Ist der Schlüssel des neuen Knotens größer als der von k, so muss der neue Knoten im rechten Teilbaum unterhalb von k sein.
-



Beispiel 5: bei 7 Entscheidung für links

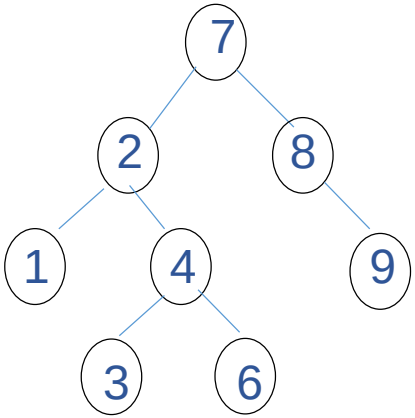
bei 2 Entscheidung für rechts

bei 4 Entscheidung für rechts

bei 6 Entscheidung für links

Bäume – Binärer Suchbaum IIII

- Wie wird ein Knoten gelöscht?
  - Falls der Knoten ein Blatt ist (zum Beispiel 3 oder 6), so wird er einfach entfernt, indem im Vater die Referenz auf das Kind auf null gesetzt.
  - Falls der Knoten nur ein Kind hat (Beispiel 8), wird im Vater die Referenz auf den zu löschenden Konten durch die Referenz auf das Kind ersetzt.
  - Falls der Knoten zwei Kinder hat, wird er durch das kleinste Kind im rechten Teilbaum unterhalb oder das größte Kind im linken Teilbaum unterhalb ersetzt.



Bäume – Binärer Suchbaum Analyse

- **Worst Case**
  - Binärer Suchbaum mit N Knoten kann zu einem Baum der Höhe N-1 entarten.
  - Beispiel: Einfügen einer sortierten Folge 1, 2, 3, 4, …,N.
  - Das Wachstum der Höhe wird mit O(N) beschrieben
- **Average Case**
  - Einfügen einer zufälligen Folge von N Elementen sorgt im Schnitt für eine logarithmische Höhe.
  - Das Wachstum der Höhe wird mit $O(\log_2 N)$ beschrieben.

Bäume – Balancierter Suchbaum

- Das Ziel des binären Suchbaums ist es, die Suche in einer Menge aus N Elementen möglichst effizient zu gestalten. Die Komplexität der Suche hängt linear von der Höhe des Baumes ab.
- Der vorigen Folie ist zu entnehmen, dass N Elemente im schlechtesten Fall in einem binären Suchbaum der Höhe N-1 dargestellt werden könne, im durchschnittlichen Fall durch einen Baum der Höhe ungefähr $\log_2 N$.
- Ziel ist es, den Suchbaum zu balancieren:
  Er heißt balanciert, falls die Höhe garantiert durch $c^* \log_2 N$ beschrieben werden kann, wobei N die Anzahl der Knoten im Baum und c eine von N unabhängige Konstante ist – kurz: die Höhe wächst nur logarithmisch.

6.1 Bäume - Traversieren

FHDW
Fachhochschule der Wirtschaft

Binärer Baum - Traversierungsarten

- Um einen Baum zu traversieren, gibt es verschiedene Traversierungsarten. Die gebräuchlichsten Traversierungsstrategien für **binäre** Bäume sind:
  - **Preorder Traversal** (Vater - Links - Rechts)
  - **Inorder Traversal** (Links - Vater - Rechts)
  - **Postorder Traversal** (Links - Rechts - Vater)
  - **Levelorder Traversal** (Ebene für Ebene)

Traversieren – Implementierung I

- Wir gehen von Klassendefinitionen für einen Knoten Node und eine Klasse BinaryTree wie folgt aus:

```
class Node {
        int key;
        Node left, right;
}
class BinaryTree {
         Node root;
}
```

- Dann lautet die Implementierung von preOrderTraversal:

```
public void preOrderTraversal(Node node) {
        if (node == null) return;
        System.out.print(node.key + " ");       // Vaterknoten
        preOrderTraversal(node.left);           // Linker Teilbaum
        preOrderTraversal(node.right);          // Rechter Teilbaum
    }
```

Traversieren – Implementierung II

- Hier die Implementierung von inOrderTraversal:

```
public void inOrderTraversal(Node node) {
        if (node == null) return;
        inOrderTraversal(node.left);
        System.out.print(node.key + " ");
        inOrderTraversal(node.right);
    }
```

- Und die Implementierung von postOrderTraversal:

```
public void postOrderTraversal(Node node) {
        if (node == null)  return;
        postOrderTraversal(node.left);          // Linker Teilbaum
        postOrderTraversal(node.right);         // Rechter Teilbaum
        System.out.print(node.key + " ");       // Vaterknoten
}
```

Traversieren – Implementierung III a

```
public void levelOrderTraversal(Node root) {
        int height = getHeight(root);
        for (int i = 0; i <= height; ++i)
            print_level(root, i);
    }
    private void print_level(Node n, int level) {
        if (n == null)
            return;
        if (level == 0)
            System.out.printf("%d ", n.key);
        else {
            print_level(n.left,   level - 1);
            print_level(n.right, level - 1);
        }
    }
    private int getHeight(Node n) {
        if (n == null)
            return -1;
        else {
            int left_height = getHeight(n.left);
            int right_height = getHeight(n.right);
            if (left_height >= right_height)
                return left_height + 1;
            else
                return right_height + 1;
        }
```

Traversieren – Implementierung III b

- Eine elegante Implementierung von levelOrderTraversal verwendet Queus und LinkedList :

```
import java.util.*;
public void levelOrderTraversal(Node root) {
        if (root == null)  return;
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while (!queue.isEmpty()) {
            Node current = queue.poll();
            System.out.print(current.key + " "); // Aktueller Knoten
// Füge den linken Knoten zur Queue hinzu
            if (current.left != null) {
                queue.add(current.left);
            }
// Füge den rechten Knoten zur Queue hinzu
            if (current.right != null) {
                queue.add(current.right);
            }
        }
    }
```

6.2 AVL-Baum

FHDW
Fachhochschule der Wirtschaft

AVL-Baum – Einführung I

Der **AVL-Baum** ist nach den sowjetischen Mathematikern Georgi Maximowitsch **A**delson-**V**elski und Jewgeni Michailowitsch **L**andis benannt, die die Datenstruktur im Jahr 1962 vorstellten. Damit ist der AVL-Baum die älteste Datenstruktur für balancierte Bäume.

Er bildet eine Datenstruktur in Form eines binären Suchbaums mit der zusätzlichen Eigenschaft, dass sich an jedem Knoten die Höhe der beiden Teilbäume um höchstens eins unterscheidet. Diese Eigenschaft lässt seine Höhe nur logarithmisch mit der Zahl der Schlüssel wachsen und macht ihn zu einem balancierten binären Suchbaum. Die maximale (und mittlere) Anzahl der Schritte (Vergleiche), die nötig sind, um An- oder Abwesenheit eines Schlüssels festzustellen, hängt direkt mit der Höhe zusammen. Ferner ist der *maximale* Aufwand für Operationen zum Einfügen und Entfernen eines Schlüssels proportional zur Höhe des Baums und damit ebenfalls logarithmisch in der Zahl der Schlüssel; der *mittlere* Aufwand ist sogar konstant, wenn das Positionieren auf das Zielelement nicht mitgerechnet wird.

AVL-Baum – Einführung II

Viele Operationen, insbesondere die Navigationsoperationen, sind direkt von den binären Suchbäumen zu übernehmen. Bei den modifizierenden Operationen muss jedoch das AVL-Kriterium beobachtet werden, womit auf jeden Fall kleine Anpassungen durchzuführen sind, die bis zu Höhenkorrekturen durch sogenannte Rotationen reichen können.

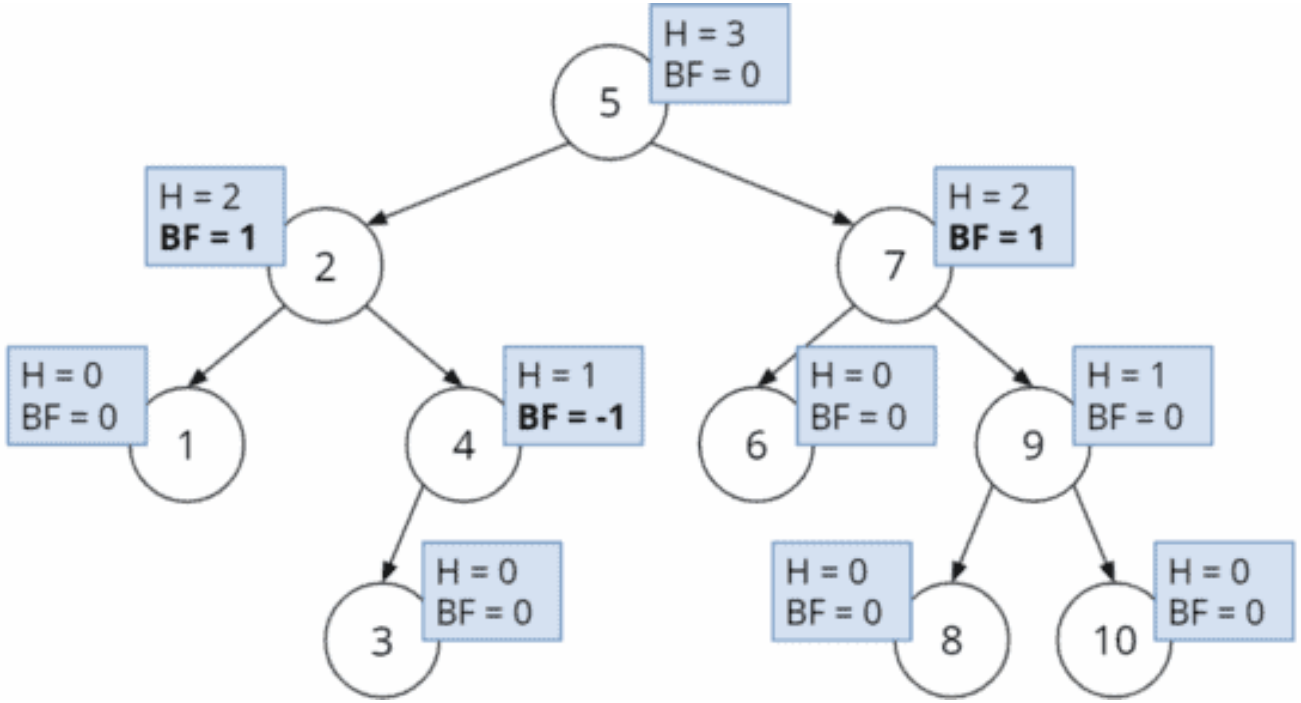
Operation	im Mittel	Worst Case
Suchen	O(log n)	O(log n)
Querschnitt	O(1)	O(log n)
Min, Max	O(log n)	O(log n)
Einfügen	O(1)	O(log n)
Löschen	O(1)	O(log n)
Verketten	O(log n)	O(log n)
Spalten	O(log n)	O(log n)

AVL-Baum - AVL-Kriterium

- Beim AVL-Baum wird die Balance über die Höhe definiert, er ist ein höhen-balancierter binärer Suchbaum.
- Als den Balance-Faktor BF(t) eines Knotens t in einem Binärbaum bezeichnet man die Höhendifferenz
  BF(t) := Height(t_r) – Height(t_l)
  wobei Height(t) die Höhe des Baums t, t_l der linke und t_r der rechte Kindbaum von t ist.
- Ein Knoten t mit BF(t)=0 wird als höhengleich oder ausgewogen bezeichnet, einer mit BF(t)<0 als links-, einer mit BF(t)>0 als rechtslastig bezeichnet. (Das wird nicht einheitlich gehandhabt!)
- Ein binärer Baum ist genau dann ein AVL-Baum, wenn für jeden Knoten t die AVL-Bedingung
  -1 <= BF(t) <= +1
  eingehalten wird.

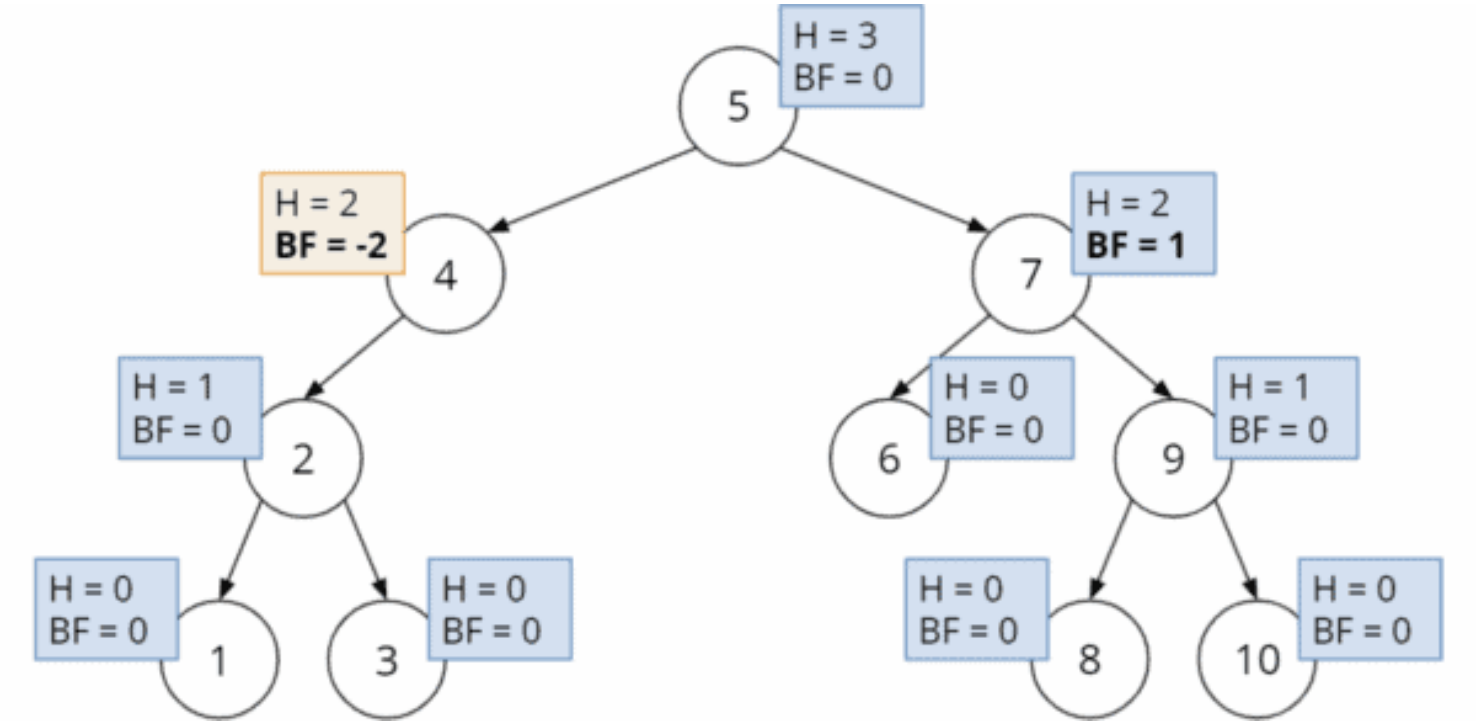
AVL-Baum – Beispiel I

Das folgende Beispiel zeigt einen AVL-Baum mit Angabe von Höhe und Balance-Faktor. an jedem Knoten. Knoten 2 und 7 sind in diesem Beispiel rechtslastig, Knoten 4 ist linkslastig. Alle anderen Knoten sind ausgewogen.



AVL-Baum – Beispiel II

Der folgende Baum ist hingegen kein AVL-Baum, da das AVL-Kriterium (-1 <= BF <= 1) an Knoten 4 nicht erfüllt ist. Dessen linker Teilbaum hat die Höhe 1 und der rechte, leere Teilbaum hat die Höhe -1. Die Differenz ist -2!



AVL-Baum – Java-Code für Knoten I

Die Definition für einen AVL-Knoten erfordert zusätzlich eine Höhenangabe zur Berechnung des Balancefaktors. Bei Abfragen der Höhenangabe durch getHeight() wird die Höhenangabe aktualisiert.

```
public class Node {
    int key;
    Node left;
    Node right;
    int height;

    public Node(int value) {  //Konstruktor
        this.key = value;
    }
```

AVL-Baum – Java-Code für Knoten II

```
    public int getHeight() {
        if (this == null)
            return -1;
        height = -1;
        getHeight(this, height);
        return height;
    }

    private void getHeight(Node tn , int h) {
        if (tn == null) {
            if (h > height)
                height = h;
            return;
        }
        getHeight(tn.left,   h+1);
        getHeight(tn.right, h+1);
    }
}
```

AVL-Baum - Navigieren

- Die *Navigationsoperationen*, das sind die verschiedenen Suchoperationen, das **Traversieren** und **Iterieren**, **Aufsuchen erstes oder letztes Element** und ähnliche, lassen den Baum unverändert und funktionieren im Prinzip auf jedem binären Suchbaum. Die dortigen Angaben zur Komplexität gelten genauso für AVL-Bäume, mit der Präzisierung, dass die Höhe des AVL-Baums sich logarithmisch zur Anzahl der Knoten verhält.
- Das **Suchen** eines Elements anhand seines Schlüssels ist die wichtigste unter den Navigationsoperationen. Die Höhen-Balancierung des AVL-Baums versucht, auf diese Operation hin zu optimieren. Sie ermöglicht einen sogenannten direkten Zugriff (im Gegensatz zum sequentiellen Zugriff der Traversierung). Sie wird in der Regel als vorausgehende Operation sowohl beim Einfügen als auch beim Löschen eingesetzt.

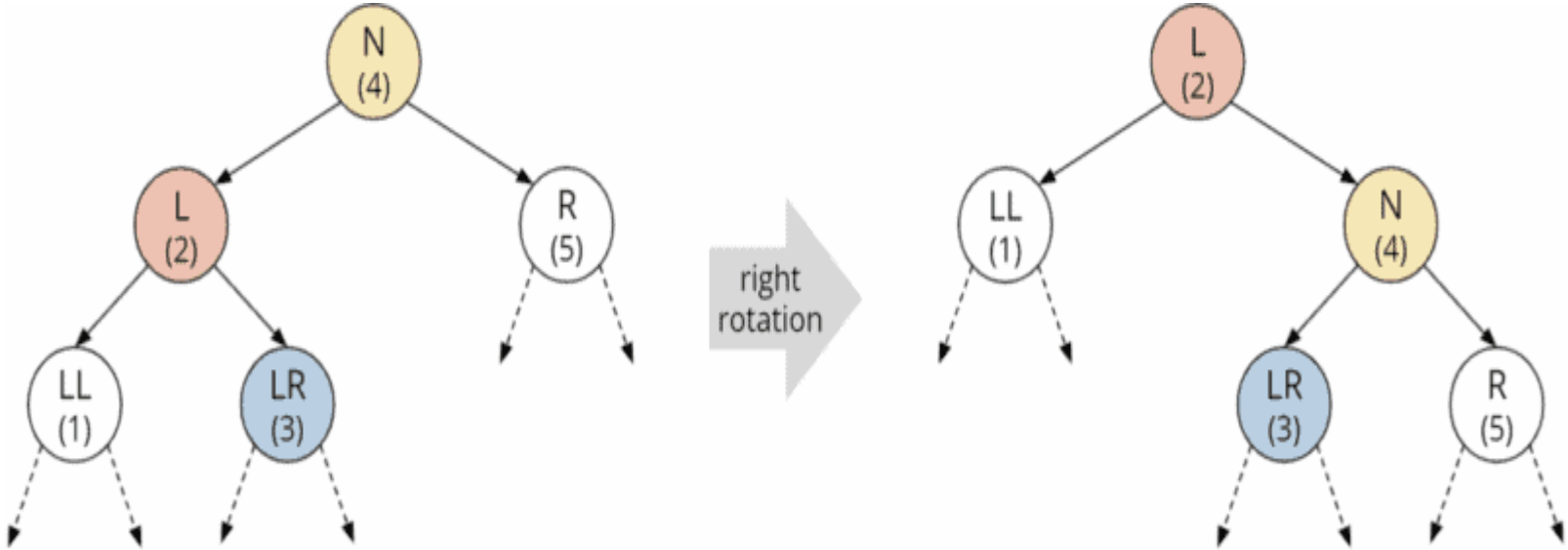
AVL-Baum - Einfügen und Löschen

- Einfügen und Löschen aus einen AVL-Baum funktioniert grundsätzlich erst einmal genau so wie in einem binären Suchbaum!
- Wenn nach einer Einfüge- oder Löschoperation das AVL-Kriterium nicht mehr erfüllt ist, muss der Baum rebalanciert werden Das geschieht durch Rotationen.
- Man unterscheidet zwischen Rechts- und Linksrotationen. In gewissen Fällen reicht eine einfache Rotation nicht aus, so dass einen Kombination aus beiden angewendet werden muss.

AVL-Baum – Rechts-Rotation I

Die folgende Grafik zeigt eine Rechts-Rotation. Der Teilbaum enthält folgende Knoten (mit beispielhaften Knotenwerten):

- N: der Knoten, an dem das Ungleichgewicht festgestellt wurde
- L: der linke Kind-Knoten von N
- LL: der linke Kind-Knoten von L
- LR: der rechte Kind-Knoten von N
- R: der rechte Kind-Knoten von N



AVL-Baum – Rechts-Rotation II

Vor der Rotation gilt folgende In-Order-Reihenfolge:

LL(1) < L(2) < LR(3) < N(4) < R(5)

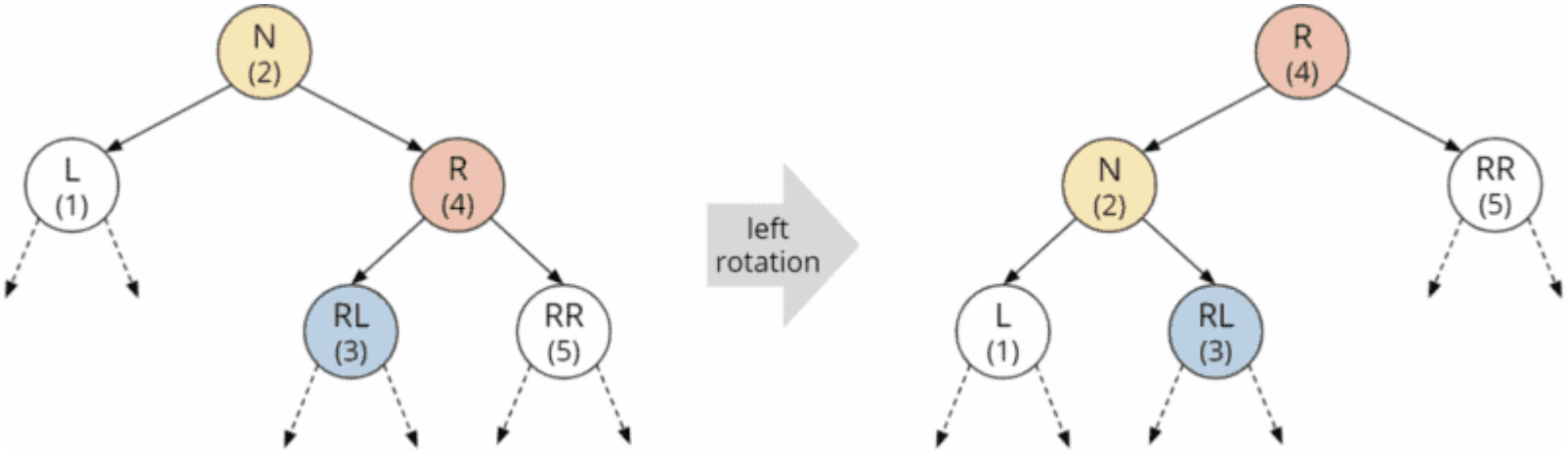
Während der Rotation wandert Knoten *L* an die Wurzel, und die vorherige Wurzel *N* wird zum rechten Kind von *L*. Das vorherige rechte Kind von *L*, *LR* wird zum neuen linken Kind von *N*. Die zwei restlichen Knoten, *LL* und *R* bleiben unverändert relativ zu ihrem Elternknoten.

Durch die Rotation wird die In-Order-Reihenfolge nicht verändert!

```
private Node rotateRight(Node node) {
    Node leftChild = node.left;
    node.left = leftChild.right;
    leftChild.right = node;
    updateHeight(node);
    updateHeight(leftChild);
    return leftChild;
}
```

AVL-Baum – Links-Rotation I

Die Links-Rotation funktioniert analog:



Knoten R wird zur Wurzel, die vorherige Wurzen wird zum linken Kind von R. Das vorherige linke Kind von R (RL) wird zum neuen rechten Kind von N. Die relativen Positionen der Knoten RR und L ändern sich nicht.
Auch bei der Links-Rotation bleibt die In-Order-Reihenfolge der Knoten erhalten .

AVL-Baum – Links-Rotation II

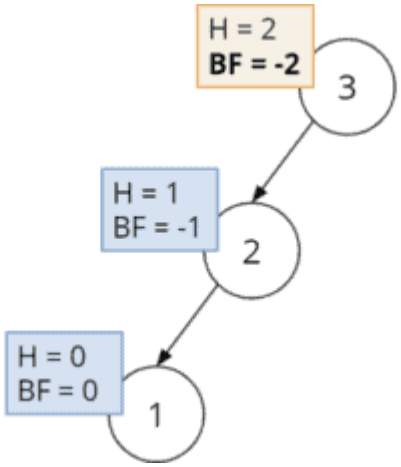
```
private Node rotateLeft(Node node) {
    Node rightChild = node.right;
    node.right = rightChild.left;
    rightChild.left = node;
    updateHeight(node);
    updateHeight(rightChild);
    return rightChild;
}
```

AVL-Baum - Rebalancierung I

- Wenn bei einer Operation ein Höhenunterschied von mehr als 1 zwischen zwei Geschwister-Teilbäumen entsteht, ist beim Elternknoten das AVL-Kriterium verletzt. Eine entsprechende Korrektur heißt „Rebalancierung“. Als Werkzeuge hierfür eignen sich die Rotationen.
- Für Einfügungen und Löschungen, bei denen die temporäre Höhendifferenz absolut nie über 2 hinausgeht, werden zwei Varianten benötigt: Einfach- und Doppelrotation.
- Eine **Einfachrotation** leistet die Rebalancierung, wenn das *innere Kind* des um 2 höheren Geschwisters *nicht höher* ist als sein Geschwister. Dieser Fall wird in der Literatur als Rechts-Rechts- (resp. gespiegelt Links-Links-)Situation bezeichnet

AVL-Baum - Rebalancierung durch Rechts-Rotation I

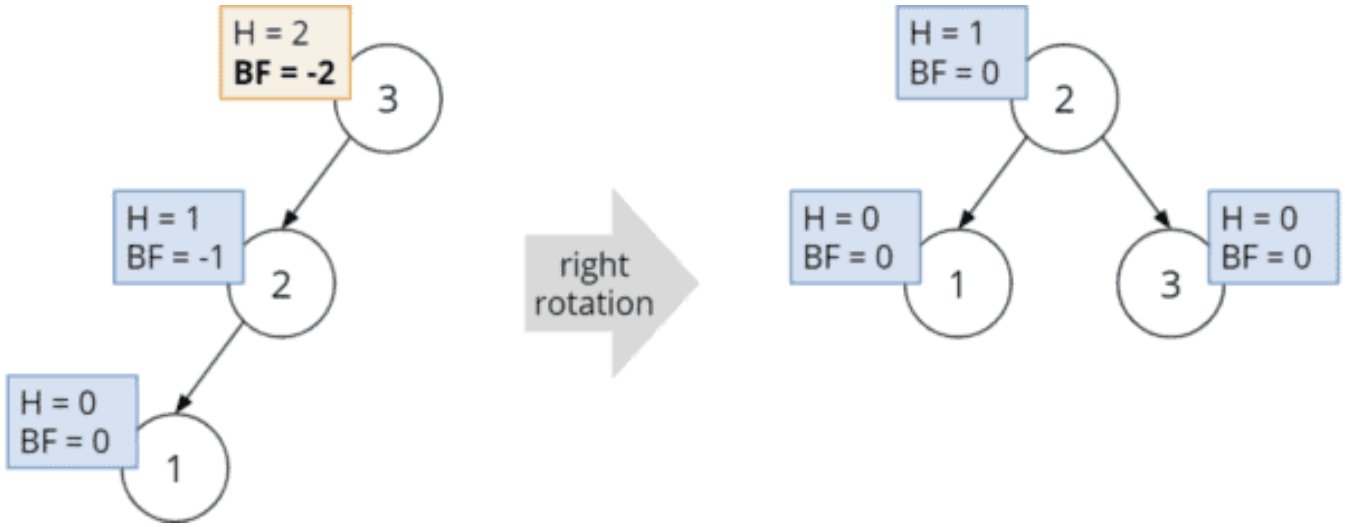
Wir fügen in einen leeren Baum die Knoten 3, 2,1 ein.



An Knoten 3 ist die AVL-Bedingung nicht erfüllt..

AVL-Baum - Rebalancierung durch Rechts-Rotation II

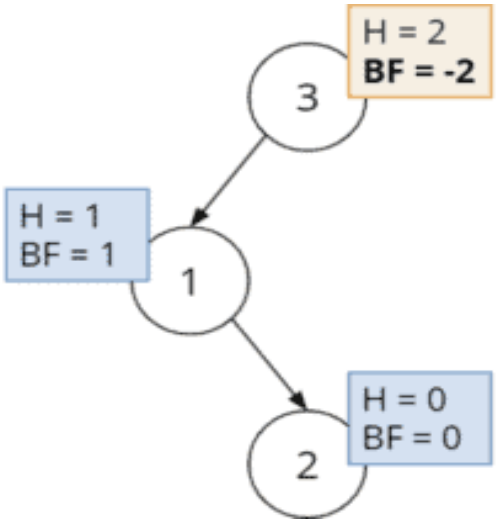
Wir führen eine Rechts-Rotation um Knoten 3 durch:



Die neue Wurzel ist Knoten 2, und dessen Balance-Faktor ist 0. Der Baum erfüllt wieder die AVL-Bedingung.

AVL-Baum - Rebalancierung durch Links-Rechts-Rotation I

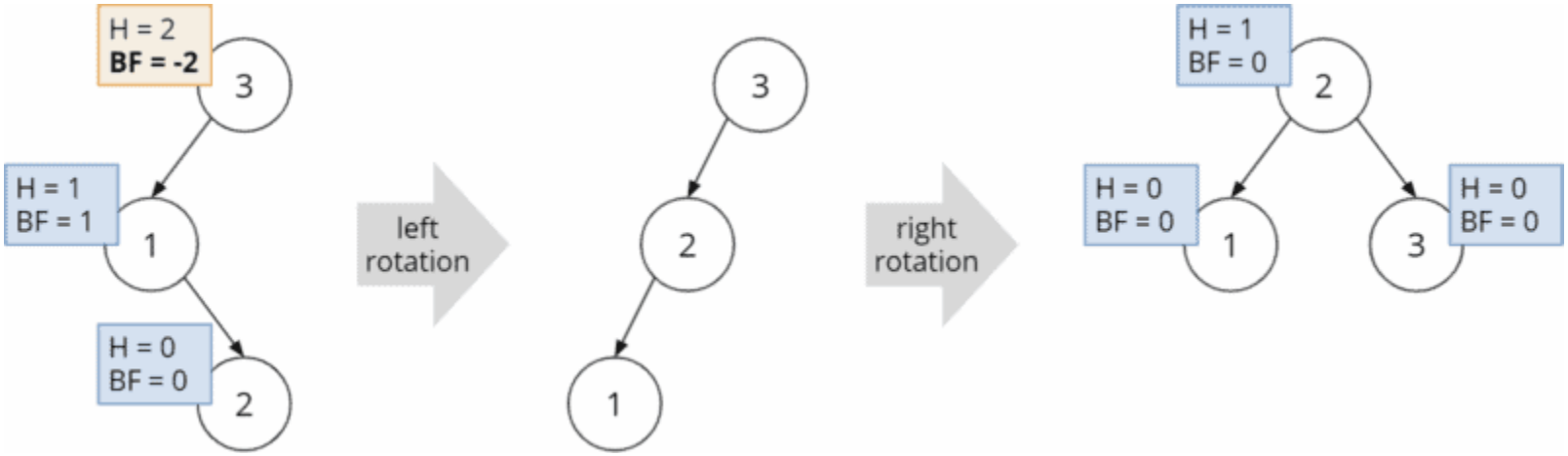
Wir fügen in einen leeren Baum die Knoten 3, 1, 2 ein.



An Knoten 3 ist die AVL-Bedingung nicht erfüllt. In diesem Fall führt eine Rechts-Rotation nicht zum Ziel, da dann der Knoten 1 den Balance-Faktor 2 hat.

AVL-Baum - Rebalancierung durch Links-Rechts-Rotation II

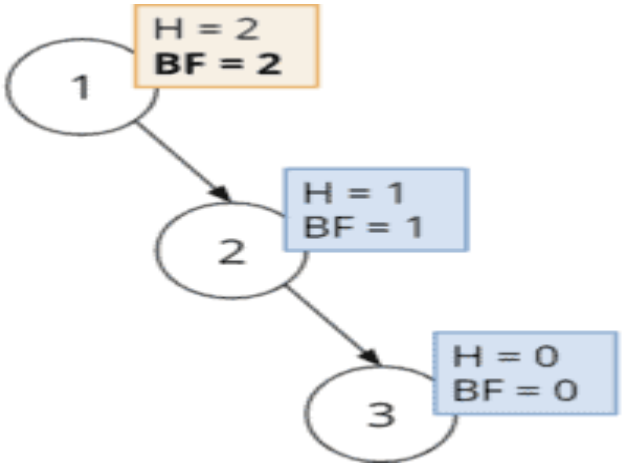
Wir führen eine Links-Rechts-Rotation durch. Dabei rotieren wir zunächst nach links um Knoten 1 und danach nach rechts um Knoten 3:



Man beachte, dass in dieser Problemsituation das linke Kind der Wurzel rechtslastig ist.

AVL-Baum - Rebalancierung durch Links-Rotation I

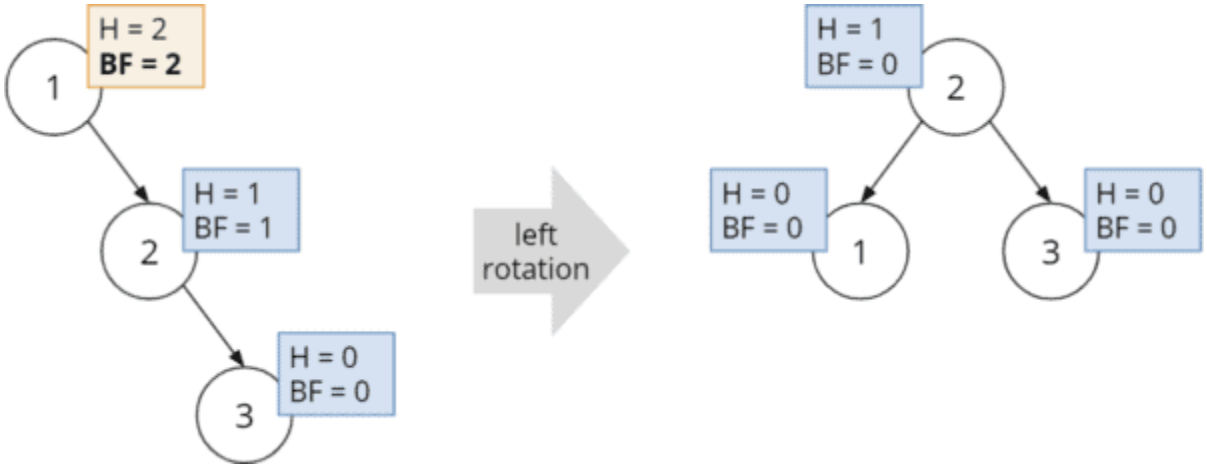
Wir fügen in einen leeren Baum die Knoten 1, 2, 3 ein. Man beachte, dass die Knoten rechtslastig sind.



An Knoten 1 ist die AVL-Bedingung nicht erfüllt..

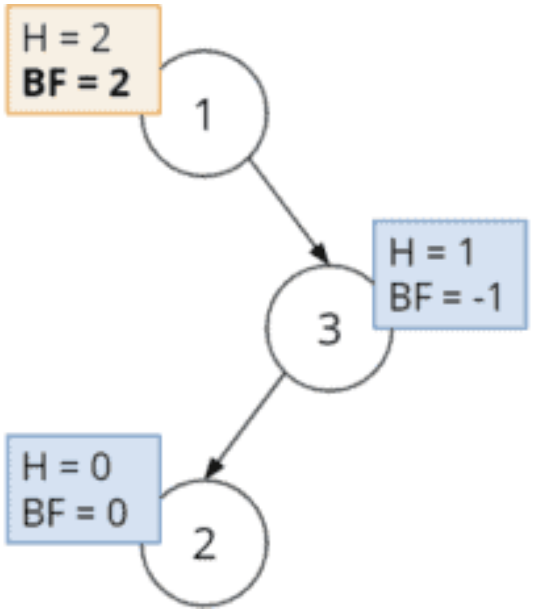
AVL-Baum - Rebalancierung durch Links-Rotation II

Wir führen eine Links-Rotation um den Knoten 1 durch.



AVL-Baum - Rebalancierung durch Rechts-Links-Rotation I

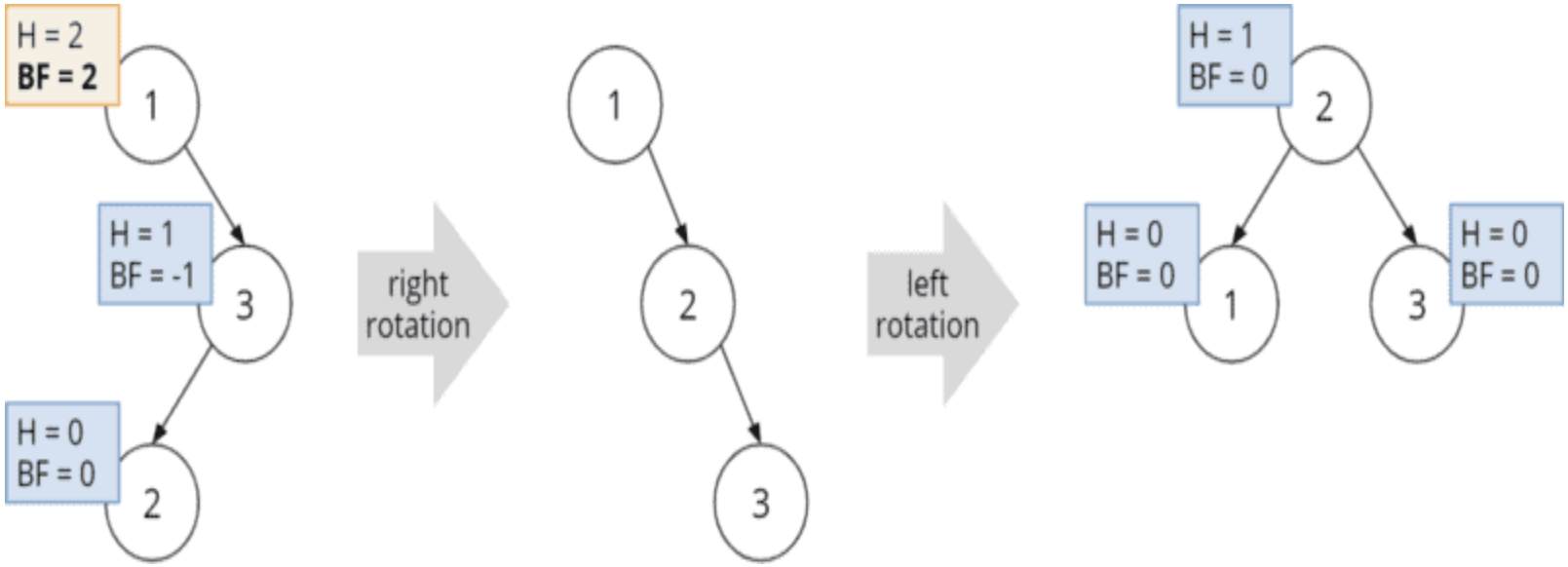
Wir fügen in einen leeren Baum die Knoten 1, 3, 2 ein. Man beachte, dass die Wurzel rechtslastig ist.



An Knoten 1 ist die AVL-Bedingung nicht erfüllt.
Bei einer einfachen Links-Rotation würde 3 zur Wurzel mit Balance-Faktor -2.

AVL-Baum - Rebalancierung durch Rechts-Links-Rotation II

In diesem Fall rotieren wir zuerst um Knoten 3 nach rechts und dann um Knoten 1 nach links.



AVL-Baum - Rebalancierung II

- Es sei N ein problematischer Knoten mit BF(N) > 1 oder BF(N) < -1; ferner sei L das linke und R das rechte Kind von N. Folgende Fälle treten auf und werden wie folgt behandelt:
  - BF(N) < -1 und BF(L)<0
    Abhilfe: Rechts-Rotation um N
  - BF(N) < -1 und BF(L)>0
    Abhilfe: Links-Rotation um R gefolgt von Rechts-Rotation um N
  - BF(N) > 1 und BF(R)>0
    Abhilfe: Links-Rotation um N
  - BF(N) > 1 und BF(R)<0
    Abhilfe: Rechts-Rotation um L gefolgt von Links-Rotation um N
- Dies behandelt einen Teilbaum. Im Vater dieses Teilbaums ist in jedem Fall der Link auf das neue Kind zu aktualisieren.

AVL-Baum - Rebalancierung III

```
private Node rebalance(Node node) {
    int balanceFactor = balanceFactor(node);
    if (balanceFactor < -1) {                       // linkslastig
        if (balanceFactor(node.left) < 0) {         // Fall 1
            node = rotateRight(node);               // Rotation nach rechts
        } else {                                    // Fall 2
            node.left = rotateLeft(node.left);      // Rotation links-rechts
            node = rotateRight(node);
        }
    }
    if (balanceFactor > 1) {                        // rechtslastig
        if (balanceFactor(node.right) > 0) {        // Fall 3
            node = rotateLeft(node);                // Rotation nach links
        } else {                                    // Fall 4
            node.right = rotateRight(node.right);   // Rotation rechts-links
            node = rotateLeft(node);
        }
    }
    return node;
}
```

6.3 Rot-Schwarz-Baum

FHDW
Fachhochschule der Wirtschaft

Rot-Schwarz-Baum – Einführung

Der Rot-Schwarz-Baum ist eine weit verbreitete konkrete Implementierung eines selbst-balancierenden binären Suchbaums, d. h. ein binärer Suchbaum, der automatisch eine gewisse Balance aufrechterhält.

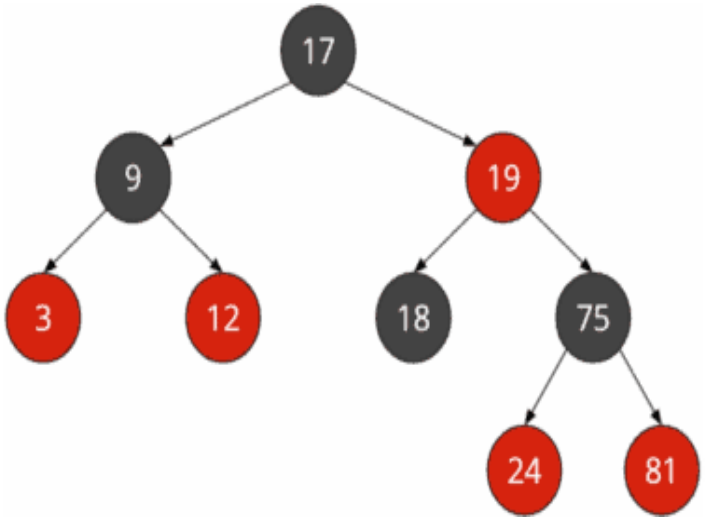
Jedem Knoten ist eine Farbe (rot oder schwarz) zugewiesen. Ein Satz von Regeln legt fest, wie diese Farben angeordnet sein müssen (z. B. darf ein roter Knoten keine roten Kinder haben). Durch diese Anordnung wird sichergestellt, dass der Baum eine gewisse Balance erhält.

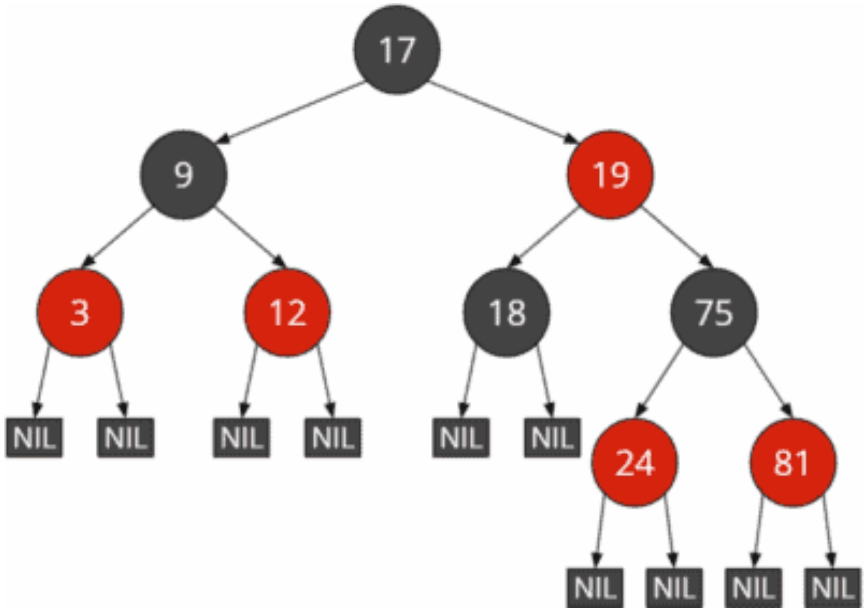
Nach dem Einfügen und Löschen von Knoten werden recht komplexe Algorithmen angewendet, um die Einhaltung der Regeln zu überprüfen – und bei Abweichungen die vorgeschriebenen Eigenschaften durch Umfärben von Knoten und Rotationen wiederherzustellen.

Rot-Schwarz-Bäume werden in der Literatur mit und ohne sogenannte NIL-Knoten dargestellt. Ein NIL-Knoten ist ein Blatt, das keinen Wert enthält. Typischerweise werden sie einfach durch null-Referenzen dargestellt. NIL-Knoten werden im späteren Verlauf für die Algorithmen relevant, um z. B. Farben von Onkel- oder Geschwister-Knoten zu bestimmen.

Rot-Schwarz-Baum – Beispiel

Die erste Grafik zeigt den Baum ohne (d. h. mit impliziten) NIL- Blättern; die zweite Grafik zeigt den Baum mit expliziten NIL-Blättern.





Rot-Schwarz-Baum – Eigenschaften

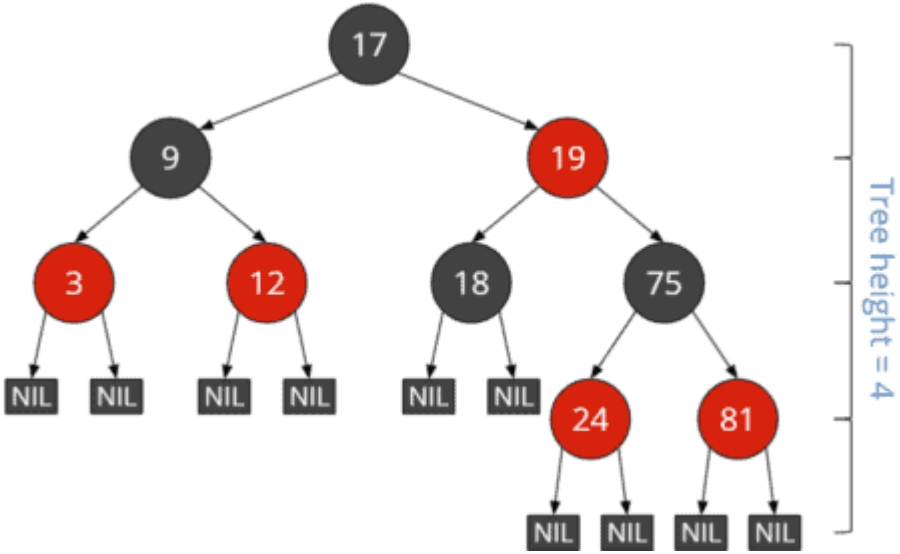
Die Balance des Rot-Schwarz-Baums wird durch folgende Regeln sichergestellt:

1. Jeder Knoten ist entweder rot oder schwarz.
2. (Die Wurzel ist schwarz.)
3. Alle NIL-Blätter sind schwarz.
4. Ein roter Knoten darf keine roten Kinder haben.
5. Alle Pfade von einem Knoten zu den darunterliegenden Blättern enthalten die gleiche Anzahl schwarzer Knoten.

Regel 2 steht in Klammern, da sie auf die Balance des Baumes keinerlei Auswirkung hat. Von daher wird Regel 2 in der Literatur oft weggelassen.
Aus Regeln 4 und 5 folgt, dass ein roter Knoten immer entweder **2** NIL-Blätter hat oder **2** schwarze Kind-Knoten mit Werten. Hätte er **1** NIL-Blatt und **1** schwarzes Kind mit Wert, so würden die Pfade durch dieses Kind mindestens einen schwarzen Knoten mehr haben als der Pfad zum NIL-Knoten.

Rot-Schwarz-Baum – Höhe I

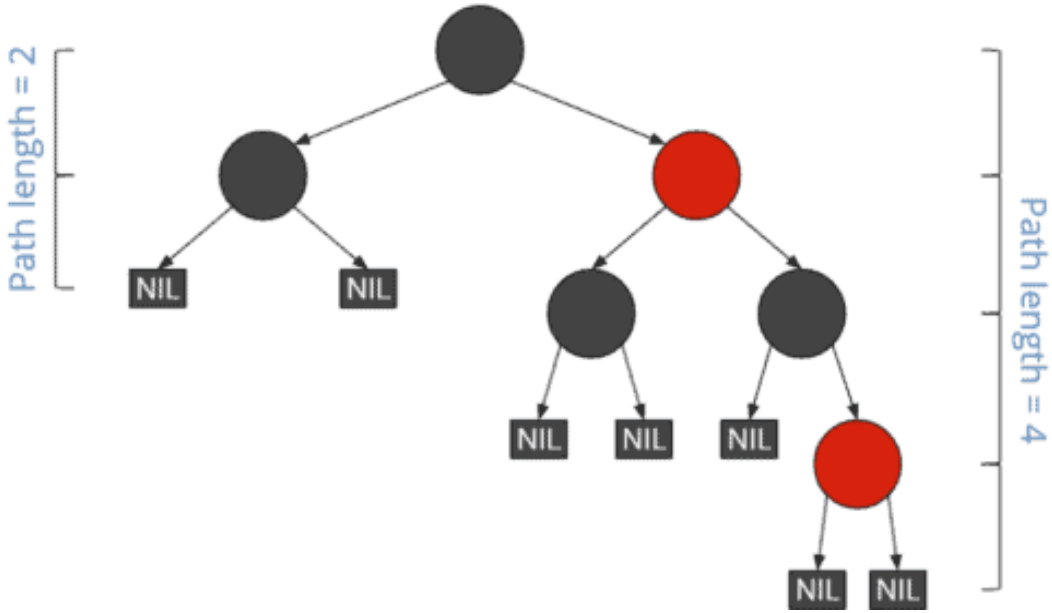
Als Höhe des Rot-Schwarz-Baums bezeichnen wir die maximale Anzahl an Knoten von der Wurzel bis zu einem NIL-Blatt, die Wurzel nicht eingerechnet. Die Höhe des Rot-Schwarz-Baums im Beispiel beträgt 4:



Aus Regel 3 und 4 folgt:
Der *längste* Pfad von der Wurzel zu einem Blatt (die Wurzel nicht eingerechnet) ist maximal doppelt so lang wie der *kürzeste* Pfad von der Wurzel zu einem Blatt.

Rot-Schwarz-Baum – Höhe II

Nehmen wir an, der kürzeste Pfad hat (zusätzlich zur Wurzel) *n* schwarze und keine roten Knoten. Dann könnten wir noch einmal *n* rote Knoten vor jedem schwarzen Knoten einfügen, ohne die Regeln zu brechen.
Das folgende Beispiel zeigt links den kürzest möglichen Pfad und rechts den längst möglichen Pfad durch einen Rot-Schwarz-Baum der Höhe 4:



Rot-Schwarz-Baum – Schwarz-Höhe

Als Schwarz-Höhe bezeichnet man die Anzahl schwarzer Knoten von einem bestimmten Knoten bis zu seinen Blättern. Die schwarzen NIL-Blätter werden dabei mitgezählt, der Startknoten nicht.

Die Schwarz-Höhe des gesamten Baumes ist die Anzahl der schwarzen Knoten von der Wurzel (diese wird nicht mitgezählt) bis zu den NIL-Blättern.

Die Schwarz-Höhe aller bisher gezeigten Rot-Schwarz-Bäume ist 2.

Rot-Schwarz-Baum – Implementation Node

Knoten werden durch die Klasse *Node* dargestellt. Als Knotenwert wird der Einfachheit eine int-Variable verwendet.

Neben den Kind-Knoten *left* und *right* wird für die Implementierung des Rot-Schwarz-Baums eine Referenz auf den Elternkonten *parent* sowie die Farbe des Knotens *isBlack* benötigt. Die Farbe wird in einer Variablen vom Typ *boolean* gespeichert, wobei *rot* als *false* und *schwarz* als *true* festgelegt wird.

```
public class Node {
    int key;
    Node left, right, parent;
    boolean isBlack;

    public Node (int v) {
        this.key = v;
    }
}
```

Rot-Schwarz-Baum – Implementation Baum

Den Rot-Schwarz-Baum wird in der Klasse RedBlackTree implementiert. Diese erweitert die Implementierung eines klassischen Binärbaums.

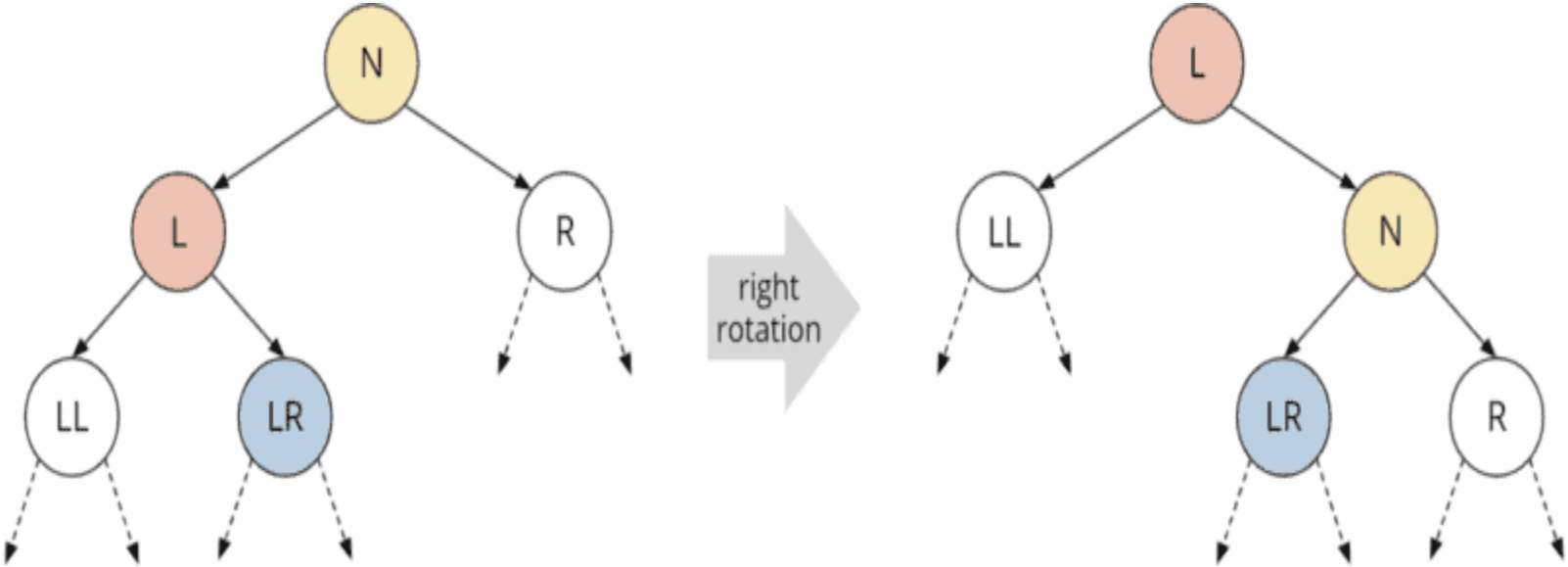
Die Operationen (Einfügen, Suche, Löschen) werden in den folgenden Abschnitten nach und nach hinzugefügt.

Nach dem Einfügen und Löschen werden die Rot-Schwarz-Regeln (s. o.) überprüft. Wenn diese verletzt wurden, müssen sie wiederhergestellt werden. Dies geschieht durch Umfärben von Knoten und durch Rotationen.

Doch zunächst müssen wir einige Hilfsfunktionen definieren – wichtig sind die Rotationen. Sie funktionieren exakt wie bei AVL-Bäumen.

Rot-Schwarz-Baum – Rechts-Rotation I

Der linke Knoten *L* wird zur neuen Wurzel, die Wurzel *N* zu dessen rechtem Kind. Das rechte Kind *LR* des vor der Rotation linken Knotens *L* wird zum linken Kind des nach der Rotation rechten Knotens *N*. Die zwei weißen Knoten *LL* und *R* ändern ihre relative Position nicht.



Rot-Schwarz-Baum – Rechts-Rotation II

Die Implementierung ist etwas aufwändiger, weil:
1. Die parent –Referenzen müssen aktualisiert werden.
2. Die Referenzen von und zum Eltern-Knoten desjenigen Knotens, der zu Beginn der Rotation oben liegt (in der Grafik *N*), muss aktualisiert werden.

```
private void rotateRight(Node node) {
        Node parent = node.parent;
        Node leftChild = node.left;
        node.left = leftChild.right;
        if (leftChild.right != null) {
            leftChild.right.parent = node;
        }
        leftChild.right = node;
        node.parent = leftChild;
        replaceParentsChild(parent, node, leftChild);
    }
```

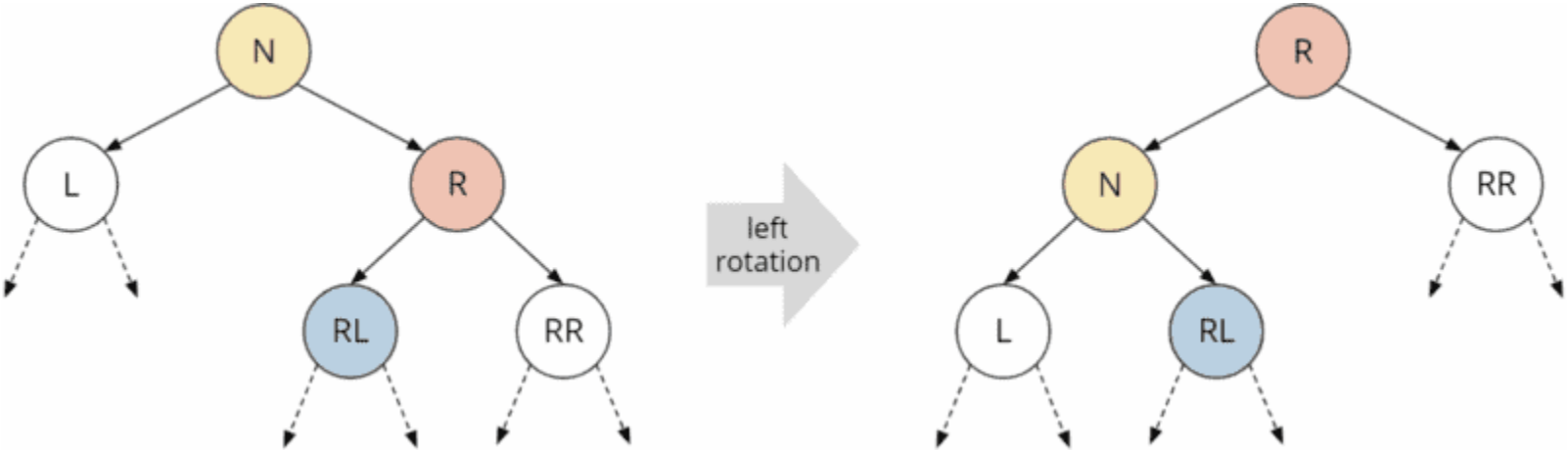
Rot-Schwarz-Baum – Rechts-Rotation III

Die am Ende aufgerufene Methode replaceParentsChild() setzt die Eltern-Kind-Beziehung zwischen dem Eltern-Knoten des ehemaligen Wurzelknotens *N* des rotierten Teilbaums und dessen neuen Wurzelknoten *L*.

```
private void replaceParentsChild(Node parent, Node oldChild, Node newChild) {
        if (parent == null) {
            root = newChild;
        } else if (parent.left == oldChild) {
            parent.left = newChild;
        } else if (parent.right == oldChild) {
            parent.right = newChild;
        } else {
            throw new IllegalStateException("Node is not a child of its parent");
        }
        if (newChild != null) {
            newChild.parent = parent;
      }
    }
```

Rot-Schwarz-Baum – Links-Rotation I

Die Links-Rotation funktioniert analog: Der rechte Knoten *R* wandert hoch an die Spitze. Die Wurzel *N* wird zum linken Kind von *R*. Das linke Kind *RL* des ehemals rechten Knotens *R* wird zum rechten Kind des nach der Rotation linken Knotens *N*. *L* und *RR* ändern ihre relative Position nicht.



Rot-Schwarz-Baum – Links-Rotation II

Hier ist die Implementation in Java:

```
private void rotateLeft(Node node) {
        Node parent = node.parent;
        Node rightChild = node.right;
        node.right = rightChild.left;
        if (rightChild.left != null) {
                rightChild.left.parent = node;
        }
        rightChild.left = node;
        node.parent = rightChild;
        replaceParentsChild(parent, node, rightChild);
    }
```

Rot-Schwarz-Baum – Suchen

Die Suche funktioniert wie in jedem binären Suchbaum: Wir vergleichen den Suchschlüssel zunächst mit der Wurzel. Ist der Suchschlüssel kleiner, setzen wir die Suche im linken Teilbaum fort; ist der Suchschlüssel größer, setzen wir die Suche im rechten Teilbaum fort.

Dies wiederholen wir solange, bis wir entweder den gesuchten Knoten gefunden haben – oder bis wir ein NIL-Blatt (im Java-Code: eine null-Referenz) erreicht haben. Das bedeutet, dass der gesuchte Schlüssel nicht im Baum existiert.

```
public Node searchNode(int key) {
        Node node = root;
        while (node != null) {
                if (key == node.key) {
                        return node;
                } else if (key < node.key) {
                        node = node.left;
                } else {
                        node = node.right;
                }
        }
        return null;
}
```

Rot-Schwarz-Baum – Einfügen I

```
public void insertNode(int key) {
    Node node = root;
    Node parent = null;
// Traversieren den Baum von links nach rechts oder rechts nach links – abhängig vom key
    while (node != null) {
        parent = node;
        if (key < node.key) {
            node = node.left;
        } else if (key > node.key) {
            node = node.right;
        } else {
            throw new IllegalArgumentException("BST contains a node with this key “);
        }
    }
    Node newNode = new Node(key);     // Füge neuen Node ein, newNode.isBlack = false
    if (parent == null) {
            root = newNode;
      } else if (key < parent.key) {
            parent.left = newNode;
      } else {
          parent.right = newNode;
       }
    newNode.parent = parent;
    fixRedBlackPropertiesAfterInsert(newNode);
 }
```

Rot-Schwarz-Baum – Einfügen II

Der neue Knoten wird initial rot gefärbt, so dass Regel 5 erfüllt ist, d. h. dass alle Pfade auch nach dem Einfügen die gleiche Anzahl schwarzer Knoten aufweisen.

Wenn der Elternknoten des eingefügten Knotens allerdings ebenfalls rot ist, wurde die Regel 4 verletzt. Dann muss durch Umfärben und/oder Rotationen den Baum so repariert werden, dass alle Regeln wieder erfüllt sind. Dies geschieht in der Methode *fixRedBlackPropertiesAfterInsert()*, die in der letzten Zeile der insertNode()-Methode aufgerufen wird. Bei der Reparatur müssen unterschiedliche Fälle betrachtet werden:

- Fall 1: Neuer Knoten ist die Wurzel
- Fall 2: Elternknoten ist rot und die Wurzel
- Fall 3: Elternknoten und Onkelknoten sind rot
- Fall 4: Elternknoten ist rot, Onkelknoten ist schwarz, eingefügter Knoten ist "innerer Enkel„
- Fall 5: Elternknoten ist rot, Onkelknoten ist schwarz, eingefügter Knoten ist "äußerer Enkel"

.

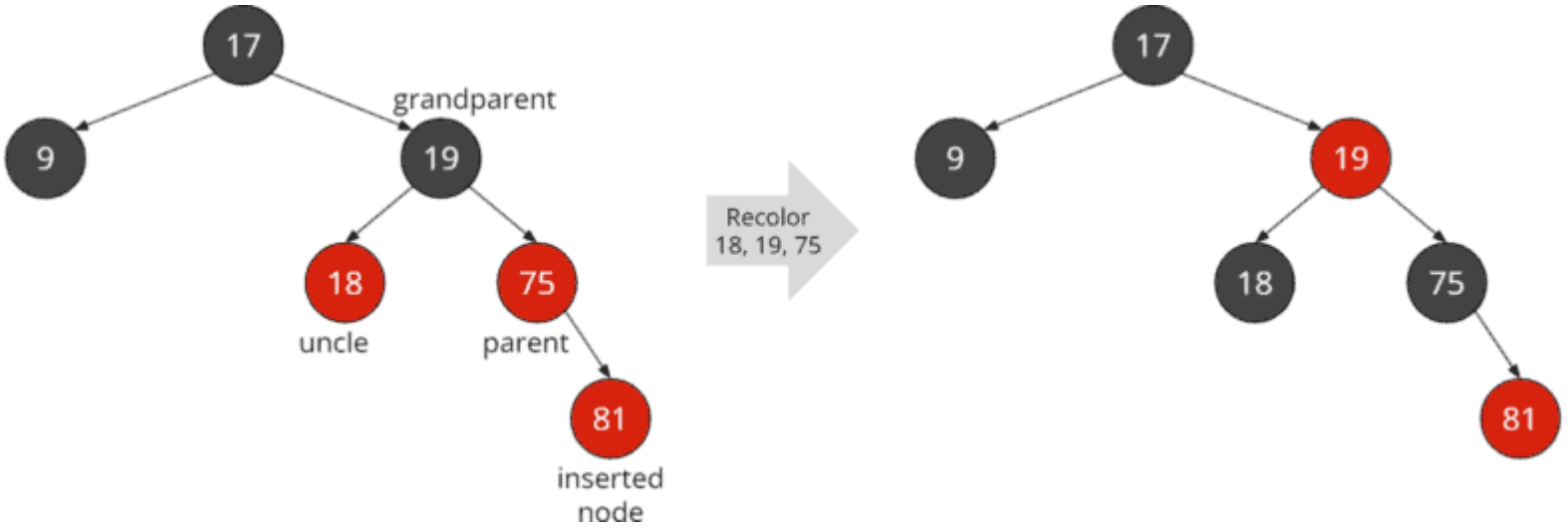
Rot-Schwarz-Baum – Einfügen - Fälle 1,2 3

Fall 1: Neuer Knoten ist die Wurzel
Sollte der neue Knoten die Wurzel sein, muss wir nichts weiter getan werden. Es sein denn, wir arbeiten mit Regel 2 ("die Wurzel ist immer schwarz"). Dann muss die Wurzel schwarz gefärbt werden.

Fall 2: Elternknoten ist rot und die Wurzel
In diesem Fall ist Regel 4 verletzt. Es muss die Wurzel schwarz gefärbt werden. Regel 5?

Fall 3: Elternknoten und Onkelknoten sind rot
Als Onkelknoten werden die Geschwisterknoten des Elternknotens bezeichnet, also das zweite Kind des Großelternknotens neben dem Elternknoten. Die folgende Grafik sollte das verständlich machen: Eingefügt wurde die 81; deren Elternknoten ist die 75, der Großelternknoten die 19 und der Onkel die 18. Sowohl der Elternknoten als auch der Onkelknoten sind rot. In diesem Fall machen wir folgendes:
Wir färben Eltern- und Onkelknoten (im Beispiel 18 und 75) schwarz und den Großelternknoten (im Beispiel 19) rot. Damit ist Regel 4 ("kein rot-rot!") am eingefügten Knoten wieder erfüllt. Die Anzahl schwarzer Knoten pro Pfad ändert sich nicht.

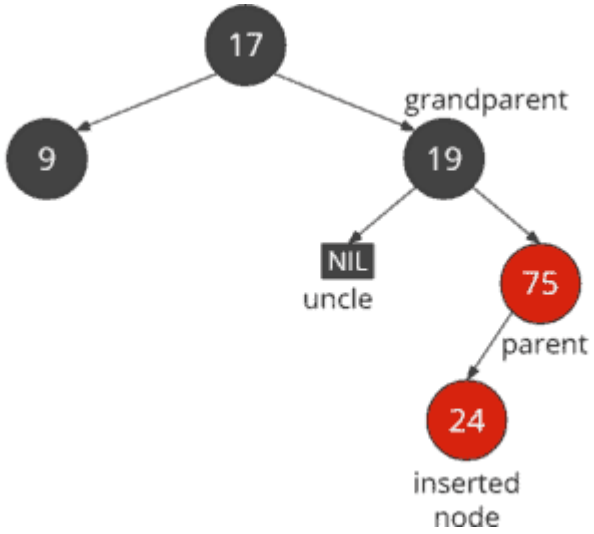
Rot-Schwarz-Baum – Einfügen - Fall 3



Allerdings könnte es nun am Großelternknoten zu zwei roten Knoten in Folge kommen – nämlich dann, wenn auch der Urgroßelternknoten (im Beispiel die 17) rot wäre. In diesem Fall müssten wir weitere Reparaturen vornehmen. Dies würde gemacht werden, indem die Reparaturfunktion rekursiv auf dem Großelternknoten aufgerufen wird.

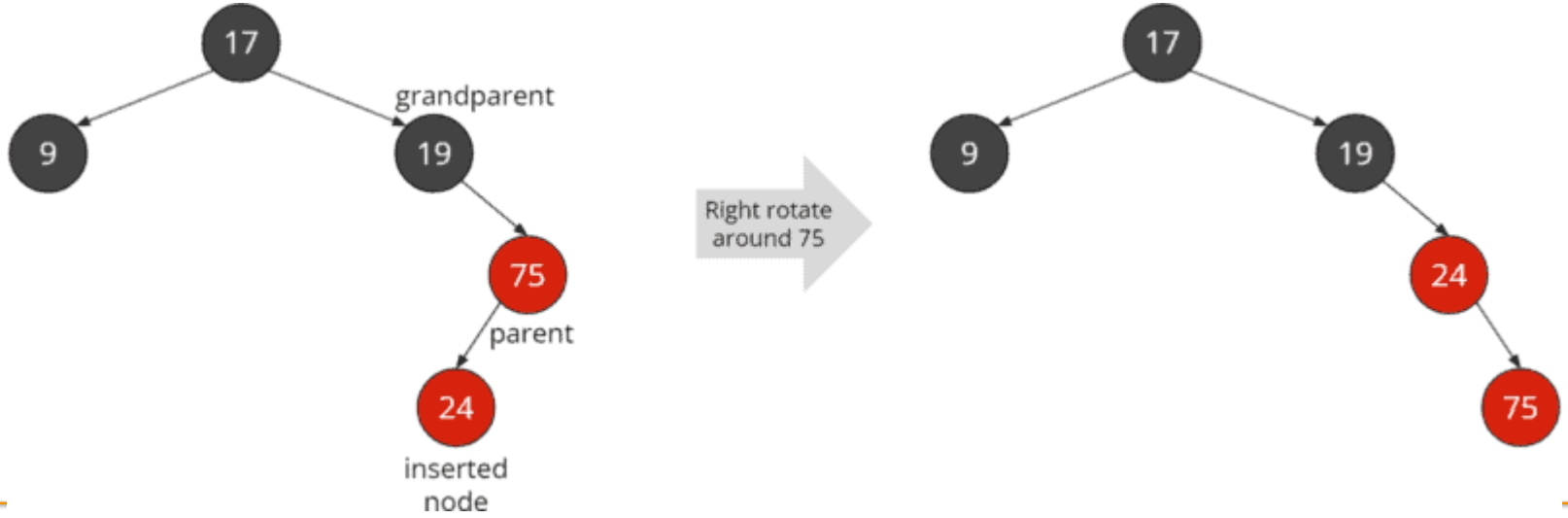
Rot-Schwarz-Baum – Einfügen – Fall 4,Teil a

Fall 4: Elternknoten ist rot, Onkelknoten ist schwarz, eingefügter Knoten ist „innerer Enkel“
Zu Klärung: "innerer Enkel" bedeutet, dass der Weg vom Großeltern-Knoten zum eingefügten Knoten ein Dreieck bildet, wie in folgender Grafik anhand der 19, 75 und 24 gezeigt. In diesem Beispiel sieht man außerdem, dass auch ein NIL-Blatt als schwarzer Onkelknoten angesehen wird (entsprechend Regel 3).



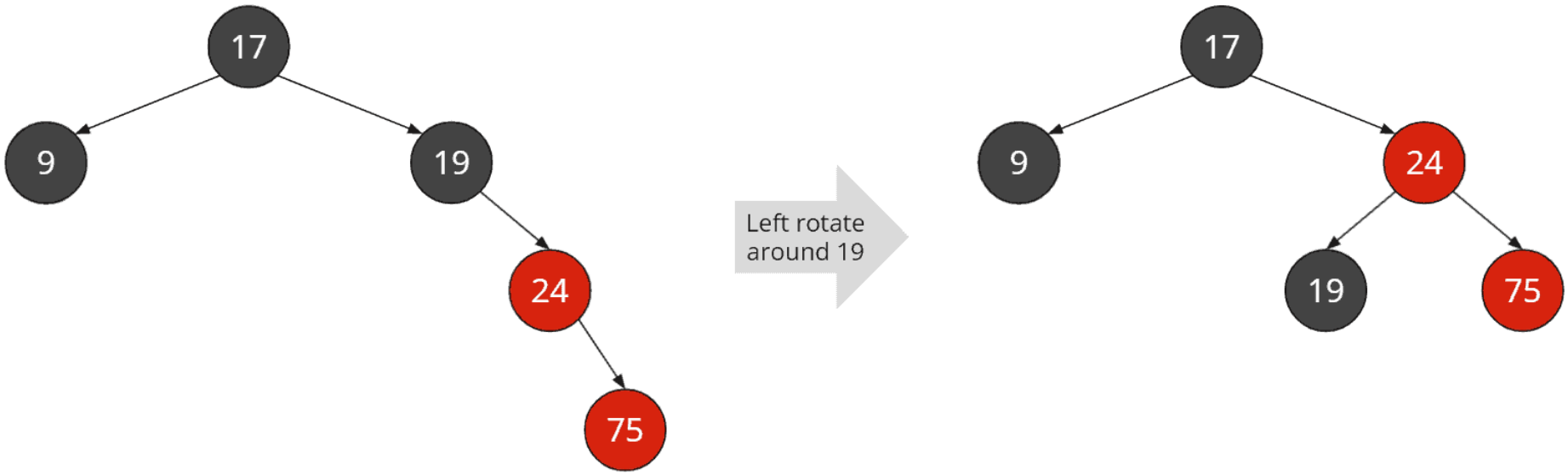
Rot-Schwarz-Baum – Einfügen – Fall 4,Teil b

In diesem Fall rotieren wir zunächst am Elternknoten in *entgegengesetzter Richtung des eingefügten Knotens*.
Was bedeutet das?
Wenn der eingefügte Knoten *linkes* Kind seines Elternknotens ist, rotieren wir am Elternknoten nach *rechts*. Wenn der eingefügte Knoten *rechtes* Kind ist, dann rotieren wir nach *links*.
Im Beispiel ist der eingefügte Knoten (die 24) linkes Kind, also rotieren wir am Elternknoten (im Beispiel 75) nach rechts:



Rot-Schwarz-Baum – Einfügen – Fall 4, Teil c

Als zweites rotieren wir am Großelternknoten in *entgegengesetzter* Richtung zur vorherigen Rotation (im Beispiel an der 19 links herum):



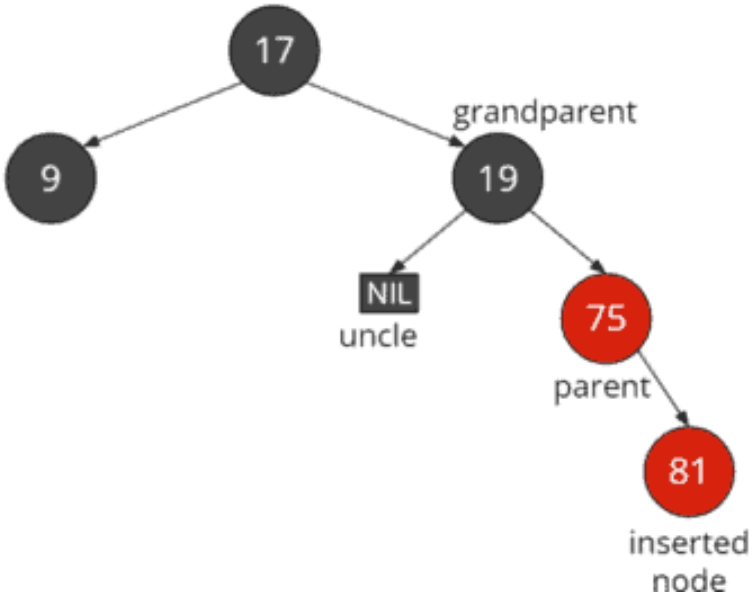
Rot-Schwarz-Baum – Einfügen – Fall 4, Teil d

Zuletzt färben wir den gerade eingefügten Knoten (im Beispiel die 24) schwarz und den ursprünglichen Großelternknoten (im Beispiel die 19) rot:

Da an der Spitze des zuletzt rotierten Teilbaumes jetzt ein schwarzer Knoten steht, kann es dort nicht zu einer Verletzung von Regel 4 ("kein rot-rot!") kommen.
Auch das Rotfärben des ursprünglichen Großelternknotens (19) kann nicht zu einer Verletzung von Regel 4 führen. Denn dessen linkes Kind ist der Onkelknoten, der per Definition dieses Falles schwarz ist. Und das rechte Kind ist als Folge der zweiten Rotation das linke Kind des eingefügten Knotens, also ein schwarzes NIL-Blatt.
Die eingefügte rote 75 hat als Kinder zwei NIL-Blätter, daher kein Verstoß gegen Regel 4.

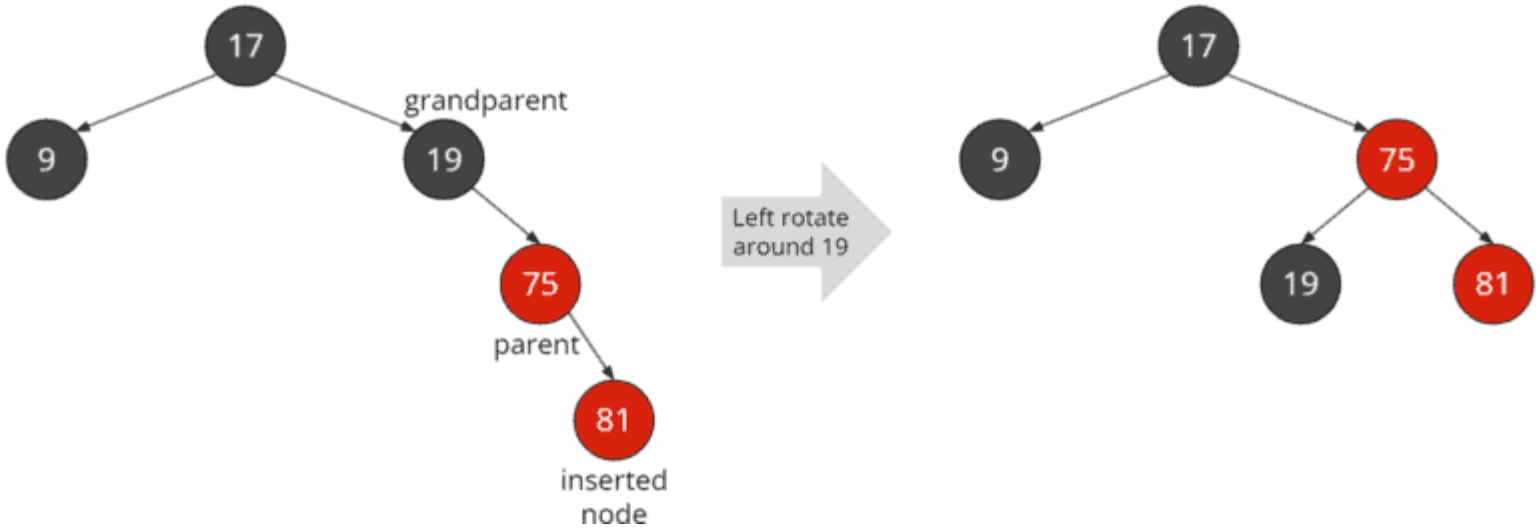
Rot-Schwarz-Baum – Einfügen – Fall 5, Teil a

Fall 5: Elternknoten ist rot, Onkelknoten ist schwarz, eingefügter Knoten ist „Äußerer Enkel“:
Zur Klärung: "Äußerer Enkel" bedeutet, dass der Weg vom Großelternknoten zum eingefügten Knoten eine Linie bildet, wie in folgendem Beispiel die 19, die 75 und die gerade eingefügte 81:



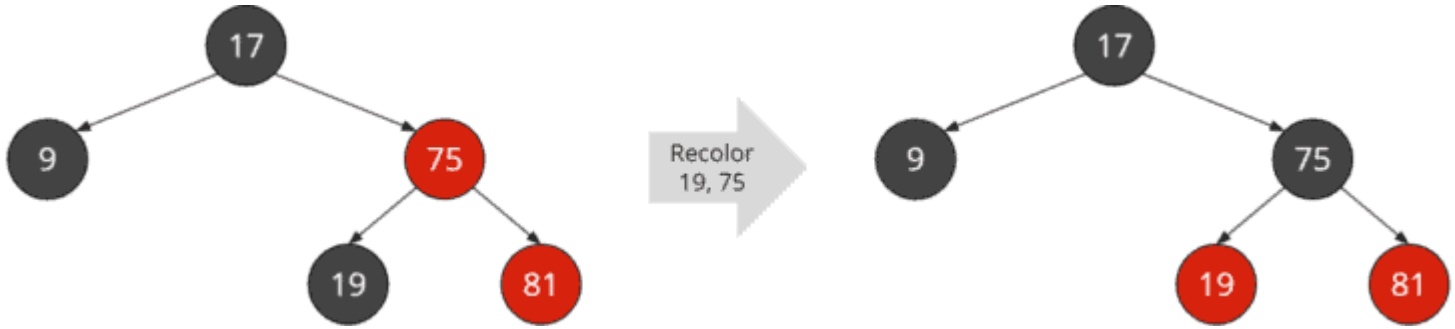
Rot-Schwarz-Baum – Einfügen – Fall 5,Teil b

In diesem Fall rotieren wir am Großelternknoten (im Beispiel 19) in *entgegengesetzter Richtung des Eltern- und eingefügten Knotens* (beide gehen in diesem Fall ja in die gleiche Richtung). Im Beispiel sind Eltern- und eingefügter Knoten jeweils *rechtes* Kind, also rotieren wir am Großelternknoten nach *links*:



Rot-Schwarz-Baum – Einfügen – Fall 5,Teil c

Danach wird der ehemalige Elternknoten (im Beispiel die 75) schwarz und der ehemaligen Großelternknoten (die 19) rotgefärbt:



An der Spitze der Rotation ist ein schwarzer Knoten, so dass es dort zu keiner Verletzung von Regel 4 ("kein rot-rot!") kommen kann.
Das linke Kind der 19 ist nach der Rotation der ursprüngliche Onkelknoten, also per Fall-Definition schwarz; das rechte Kind der 19 ist das ursprünglich linke Kind des Elternknotens (75), welches ebenfalls ein schwarzes NIL-Blatt sein muss, da sonst der rechte Platz, an dem wir die 81 eingefügt haben, nicht frei gewesen wäre (denn ein roter Knoten hat immer entweder zwei schwarze Kinder mit Wert oder zwei schwarze NIL-Kinder).
Die rote 81 ist der eingefügte Knoten und hat daher ebenfalls zwei schwarze NIL-Blätter.

Rot-Schwarz-Baum – Einfügen – Java I

```
 private void fixRedBlackPropertiesAfterInsert(Node node) {
    Node parent = node.parent;
// Fall 1: Parent ist null, wir haben root erreicht --> Ende der Rekursion
    if (parent == null) {
// Nur erforderlich, falls wir mit  Regel 2 arbeiten:
        node.isBlack = true;
        return;
    }
// Parent ist schwarz --> nichts tun
   if (parent.isBlack == true) {
        return;
    }
// Ab hier ist parent rot
    Node grandparent = parent.parent;

// Fall 2:
// Die Nicht-Existenz von grandparent bedeutet, dass  parent  gleich root ist. Falls wir mit
// Regel 2 arbeiten, so ist grandparent nie null, und der folgende if-then Block kann entfallen.
     if (grandparent == null) {
// Dieser Teil wird nur aufgerufen auf roten nodes (entweder auf neu eingefügten - oder
// rekursiv auf roten grandparents). Daher muss nur root schwarz gefärbt werden.
            parent.isBlack = true;
            return;
     }
```

Rot-Schwarz-Baum – Einfügen – Java II

```
// Hole Onkel uncle (kann null/nil sein, in dem Fall ist seine Farbe schwarz)
    Node uncle = getUncle(parent);
// Fall 3: uncle ist rot -> ändere die Farbe für parent, grandparent und uncle
    if (uncle != null && uncle.isBlack == false) {
        parent.isBlack = true;
         grandparent.isBlack = false;
         uncle.isBlack = true;
// Rekursiver Aufruf für grandparent, welcher nun rot ist
// Es kann root sein oder roten parent haben, in den Fällen muss mehr repariert werden
         fixRedBlackPropertiesAfterInsert(grandparent);
    }
    else if (parent == grandparent.left) {   // Parent ist linkes Kind von grandparent
// Fall 4a: Uncle is black and node is left->right "inner child" von grandparent
            if (node == parent.right) {
                rotateLeft(parent);
// "parent" zeigt auf den neuen Wurzelknoten des rotierten Teilbaums.
// und wird im nächsten Schritt umgefärbt.
                parent = node;
            }
// Fall 5a: Uncle ist schwarz und node is left->left "outer child" von grandparent
            rotateRight(grandparent);
// Färbe ursprüngliche parent und grandparent um
            parent.isBlack = true;
            grandparent.isBlack = true;
        }
```

Rot-Schwarz-Baum – Einfügen – Java III

```
// Parent ist rechtes Kind von grandparent
        else {
 // Fall 4b: Uncle ist schwarz and node ist right->left "inner child" von grandparent
            if (node == parent.left) {
                rotateRight(parent);
// "parent" zeigt auf den neuen Wurzelknoten des rotierten Teilbaums.
// und wird im nächsten Schritt umgefärbt .
                parent = node;
            }
// Fall 5b: Uncle ist schwarz und node is right->right "outer child" von grandparent
            rotateLeft(grandparent);
// Färbe ursprüngliche parent und grandparent um
            parent.isBlack = true;
            grandparent.isBlack = false;
        }
    }

private Node getUncle(Node parent) {
        Node grandparent = parent.parent;
        if (grandparent.left == parent) {
            return grandparent.right;
        } else if (grandparent.right == parent) {
            return grandparent.left;
        } else {
            throw new IllegalStateException("Parent is not a child of its grandparent");
        }
    }
```

6.4. Vergleich

FHDW
Fachhochschule der Wirtschaft

Vergleich – Allgemeines

Der Rot-Schwarz-Baum sowie der AVL-Baum sind selbstbalancierende binäre Suchbäume.

Beim Rot-Schwarz-Baum ist der längste Pfad zur Wurzel maximal doppelt so lang wie der kürzeste Pfad zur Wurzel. Beim AVL-Baum hingegen unterscheidet sich die Tiefe keiner zweier Teilbäume um mehr als 1.

Im Rot-Schwarz-Baum wird die Balance durch die Farben der Knoten, ein Set von Regeln, und durch Rotationen und Umfärben der Knoten sichergestellt. Im AVL-Baum hingegen werden die Höhen der Teilbäume verglichen und bei Bedarf Rotationen durchgeführt.

Vergleich – Unterschiede und Fazit

- Diese Unterschiede in den Eigenschaften der zwei Baumarten führen zu folgenden Unterschieden bzgl. Performance und Speicherbedarf:
  - Aufgrund der gleichmäßigeren Balancierung des AVL-Baums erfolgt die Suche in diesem in der Regel schneller als im Rot-Schwarz-Baum. Von der Größenordnung her liegen aber beide im Bereich *O(log n)*.
  - Auch für Einfügen und Löschen beträgt die Zeitkomplexität in beiden Bäumen *O(log n)*. Im direkten Vergleich ist allerdings der Rot-Schwarz-Baum schneller, da dieser seltener rebalanciert wird.
  - Beide Bäume benötigen zusätzlichen Speicher:
    - pro Knoten speichert der AVL-Baum zusätzlich die Höhe des an einem Knoten beginnenden Teilbaums
    - der Rot-Schwarz-Baum speichert zusätzlich die Farbinformation sowie die Referenz auf den Vaterknoten.
    In der Praxis sind diese zusätzlichen Hauptspeicheranforderungen unerheblich!
- Erwartet man viele Einfüge-/Löschoperationen, dann sollte man also eher einen Rot-Schwarz-Baum einsetzen. Geht man dagegen von mehr Suchoperationen aus, dann sollte die Wahl auf den AVL-Baum fallen.

7. Sammlungen

FHDW
Fachhochschule der Wirtschaft

Sammlungen – Allgemeines I

- Manche grundlegende Datentypen sind mit Sammlungen von Objekten verbunden. Sammlungen sind ein allgemeines Konzept, das in der Programmierung üblicherweise als „Collection“ bezeichnet wird. Insbesondere Wertemengen sind Sammlungen von Objekten, und ihre Operationen haben die Aufgabe, der Sammlung Objekte hinzuzufügen, Objekte zu entfernen oder Objekte in der Sammlung zu untersuchen. In diesem Abschnitt betrachten wir drei solcher Datentypen: Multimenge (bag), Warteschlange (queue) und Stapel (stack), die sich darin unterscheiden, welches Objekt als Nächstes entfernt oder untersucht wird.

- Multimengen, Warteschlangen und Stapel sind elementare, vielseitig einsetzbare Datenstrukturen, die häufig eingesetzt werden. Der Anwendungs- und Implementierungscode soll mit der allgemeinen Herangehensweise an die Entwicklung von Datenstrukturen und Algorithmen vertraut machen.

Sammlungen – Allgemeines II

- Eines der Ziele ist es zu zeigen, dass die Art, wie wir Sammlungen von Objekten repräsentieren, direkte Auswirkungen auf die Effizienz der verschiedenen Operationen hat. Für Collections werden daher Datenstrukturen entworfen, die genau die Sammlung von Objekten repräsentieren, die eine effiziente Implementierung der benötigten Operationen unterstützt.

- Das zweite Ziel ist es, die grundlegenden Konzepte für Generics und Iteration vorzustellen, die den Code stark vereinfachen. Beides sind fortgeschrittene Mechanismen, die für das Verständnis der Algorithmen nicht unbedingt erforderlich sind. Da man aber mit ihrer Hilfe Anwendungs-Code (und Implementierungen von Algorithmen) entwickeln kann, der klarer, kompakter und eleganter ist, als es ohne diese Konzepte möglich wäre, soll hier nicht auf sie verzichtet werden.

Sammlungen – Allgemeines III

- Und als drittes Ziel werden in diesem Abschnitt die Vorteile und die Bedeutung verketteter Datenstrukturen vor Augen geführt. Vor allem die klassische Datenstruktur der verketteten Liste erlaubt die Implementierung von Multimengen, Warteschlangen und Stapel von einer Effizienz, die anders nicht zu erreichen wäre. Verkettete Listen zu verstehen, ist daher ein wichtiger erster Schritt zum Studium der Algorithmen und Datenstrukturen.
- Zu jedem der drei Datentypen betrachten wir APIs und Beispielanwendungen und stellen dann mögliche Repräsentationen der Datentypwerte und Implementierungen der Datentypoperationen vor – eine Vorgehensweise, die von nun an (für kompliziertere Datenstrukturen) beibehalten werden soll. Die vorgestellten Implementierungen sind Modelle für die noch folgenden Implementierungen.

Sammlungen – Grundlegendes für API

- Wie beginnen die Diskussion der abstrakten Datentypen für Collections mit der Definition ihrer Methoden. Jedes API enthält
  - einen Konstruktor, der keine Argumente übernimmt,
  - eine Methode zum Hinzufügen eines Elements zur Collection
  - eine Methode, um zu testen, ob die Collection leer ist,
  - eine Methode, die die Größe der Collection zurückliefert
- Stack und Queue haben zusätzlich eine Methode, um ein Element aus der Collection zu löschen. Darüber hinaus nutzen diese APIs spezielle Konzepte der Objekt-orientierten Programmierung: Generics und iterierbare Collections.

Sammlungen – API Teil 1

Public class Bag<Item> implements Iterable<Item>		
	Bag()	Erzeugt eine leere Multimenge
void	add(Item item)	Fügt ein Element hinzu
boolean	isEmpty()	Ist die Multimenge leer?
int	size()	Anzahl der Elemente in der Multimenge

Public class Queue<Item> implements Iterable<Item>		
	Queue()	Erzeugt eine leere Multimenge
void	enqueue(Item item)	Fügt ein Element hinzu
Item	dequeue()	Entfernt das am frühesten hinzugefügte Element
boolean	isEmpty()	Ist die Warteschlange leer?
int	size()	Anzahl der Elemente in der Warteschlange

Sammlungen – API Teil 2

Public class Stack<Item> implements Iterable<Item>		
	Stack()	Erzeugt einen leeren Stapel
void	push(Item item)	Fügt ein Element hinzu
Item	pop()	Entfernt das zuletzt hinzugefügte Element
boolean	isEmpty()	Ist der Stapel leer?
int	size()	Anzahl der Elemente im Stapel

Sammlungen – Generics I

- Ein wesentliches Merkmal der abstrakten Datentypen für Collections ist, dass man sie grundsätzlich für jede Art von Daten verwenden kann. Ein spezielles Konzept der Objekt-orientierten Programmierung, Generics oder auch parametrisierte Typen, genannt, ermöglicht es, diese Eigenschaft in die Implementierung zu übernehmen. Die Umsetzung von Generics hat so weitreichende Auswirkungen auf die Programmiersprachen, dass in vielen Sprachen darauf verzichtet wird. Für die Art und Weise, wie sie im vorliegenden Kontext verwendet wird, muss man sich aber glücklicherweise mit nur wenigen neuen Syntaxformen auseinandersetzen.

Sammlungen – Generics II

- Die Notation <T> nach dem Klassennamen definiert den Namen Item als einen Typparameter, einen symbolischen Platzhalter für einen tatsächlichen Typ, der vom Client angegeben wird. Sie können Stack<Item> lesen als „Stapel von Elementen“. Wenn wir Stack implementieren, kennen wir den eigentlichen Typ Stack nicht, aber eine Anwendung kann den Stapel für jeden Datentyp seiner Wahl verwenden, einschließlich einem, der lange nach der Entwicklung der Implementierung definiert wurde. Der Anwendungs-Code gibt bei der Erzeugung des Stapels den gewünschten Typ vor: Wir können Item mit dem Namen jedes beliebigen Referenzdatentyps ersetzen (konsequent überall dort, wo er steht). Das ist genau das, was wir wünschen. Hier ein wenig Anwendungscode:

```
Stack<String> stack = new Stack<String>();
stack.push(“Test“);
…
String next = stack.pop();
```

Sammlungen – Generics III

- Wenn man versucht, in vorhergehendem Beispiel ein Integer-Objekt (oder Daten irgendeines anderen Typs als String) zu *stack* hinzuzufügen, erhält man einen Compilerfehler. Ohne Generics müsste man für jeden Datentyp, den wir in einer Collection verwahren wollen, eine eigene API definieren; mit Generics reicht eine API (und eine Implementierung) für alle Datentypen, inklusive denen, die erst noch entwickelt werden.
  Generische Typen führen zu klaren, übersichtlichem Code, der leicht zu verstehen und debuggen ist, sodass diese Datentypen im restlichen Verlauf verwendet werden.

- Zur Erinnerung: anders als bei C# müssen die generischen Typen zur Laufzeit durch **Referenztypen** geschlossen sein. Das impliziert, dass anstelle der Basistypen wie int die entsprechenden Referenztypen wie **Integer** benutzt werden müssen.

8. Stack

Stack – Allgemeines I

- Ein Stapel (Stack) ist eine Collection, die auf dem LIFO-Prinzip aufbaut. Post auf dem Schreibtisch wird meist in Form eines Stapels abgelegt. Die neu eingegangene Post wird obenauf gelegt und dann zum Lesen herunter geholt. Heutzutage ist nicht mehr so viel Papier im Umlauf wie früher, aber das hier beschriebene Prinzip liegt einer Reihe von Anwendungen zugrunde, die man regelmäßig auf dem Computer nutzt. So ist zum Beispiel auch die E-Mail-Ablage häufig als Stapel organisiert, wobei die neuesten Nachrichten obenauf angezeigt werden und von dort entnommen werden. Der Nachteil ist, dass wir den Stapel vielleicht niemals ganz abarbeiten, so dass alte E-Mails ungelesen bleiben.

Stack – Allgemeines II

- Wenn man in einem Browser einen Hyperlink anklickt, zeigt der Browser eine neue Seite an (und legt sie auf einem Stapel in der Historie ab). Man kann weitere Links anklicken, um neue Seiten zu besuchen, aber man kann auch immer zur vorherigen Seite zurückkehren, indem man auf den Zurück-Button klickt und dabei den obersten Eintrag vom Stapel entnimmt.
- Die Last-In-first-Out-Strategie eines Stapels ist genau das Verhalten, das hier beschrieben wurde. Wenn eine Anwendung mit dem for-each-Konzept über die Elemente des Stapels iteriert, werden die Elemente in der umgekehrten Reihenfolge verarbeitet, in der sie hinzugefügt wurde.

Stack – Stapel fester Größe I

- Als Beispiel betrachten wir einen abstrakten Datentyp für einen Stapel fester Größer von Strings. Die API lässt sich zunächst nur auf Strings anwenden, die Anwendung muss zudem eine Größe angeben und unterstützt keine Iteration. Für FixedCapacityStackOfStrings führt dies zu nachstehender einfacher Implementierung. Die Instanzvariablen bestehen aus einem Array a[ ] für die Elemente in dem Stapel und einem Integer N, der die Anzahl der Elemente in dem Stapel festhält. Um ein Element zu entfernen, wird N dekrementiert und a[N] zurückgeliefert. Um ein neues Element einzufügen, wird a[N] auf das neue Element gesetzt und N inkrementiert. Damit ist garantiert, dass
  - Die Elemente in dem Stapel in der Reihenfolge stehen wie sie eingefügt werden.
  - Der Stapel ist leer, wenn N gleich 0 ist.
  - Das oberste Element im Stapel befindet sich in a[N-1], wenn der Stapel nicht leer ist.

Stack – Stapel fester Größe II

Die Anwendung liest Zeichenkette von der Tastatur und speichert sie auf dem Stapel. Wenn als Zeichenkette “-“ eingegeben wird, wird die letzte auf dem Stapel gespeicherte Zeichenkette vom Stapel geholt und ausgegeben.

```
public class FixedCapacityOfStrings
{
   public static void main(String[ ] args)  {
      FixedCapacityStackOfStrings s = new FixedCapacityStackOfStrings(100);
      System.out.println("Gib Zeichenketten ein, Abbruch mit -:");
      while (true) {
         String item = new Scanner(System.in).nextLine();
         if (item.equals("-"))
            break;
         else s.push(item);
      }
      System.out.printf("Last element on stack: %s\n", s.pop());
     System.out.printf("%d left on Stack\n", s.size());
   }
}
```

Stack – Stapel fester Größe III

Nachstehend die Implementation der Klasse FixedCapacityStackOfStrings:

```
public class FixedCapacityStackOfStrings
{
   private String [ ] a;
   private int N;

   public FixedCapacityStackOfStrings(int cap)
   { a = new String[cap]; }

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

   public void push(String item)     { a[N++] = item; }
   public String pop()               { return a[--N]; }
}
```

Stack – Stapel fester Größe mit Generics I

- Ein Nachteil der Implementierung ist, dass nur Zeichenketten aufgenommen werden können. Wenn also statt Zeichenketten double-Werte aufgenommen werden sollen, muss eine ähnliche Klasse entwickelt werden, in der String durch double ersetzt wird. Das geht eleganter, indem Generics verwendet werden. Dazu wir eine Klasse FixedCapacityStack implementiert, die sich von FixedCapacityOfStrings im Wesentlichen dadurch unterscheidet, dass String durch <T> ersetzt wird.
  Der Name T ist ein Typparameter, ein symbolischer Platzhalter für irgendeinen Referenztyp. Bei der Implementierung dieser Klasse kennt man den expliziten Typ von T nicht. Leider hat die Sache den Haken, dass generische Arrays in Java nicht erlaubt sind, wohl aber schon in C#. Stattdessen müssen wir eine Umgehung vornehmen, indem wir ein Object-Array allokieren und auf ein Typ-Array casten:

  a = (T[ ]) new Object(cap);

Stack – Stapel fester Größe mit Generics II

In der Anwendung muss nur eine Zeile geringfügig geändert werden:

FixedCapacityStack<String> s = new FixedCapacityStack<String>(100);

Nachstehend die Implementation der Klasse FixedCapacityStack:

```
public class FixedCapacityStack<T>
{
   private T[ ] a;
   private int N;

   public FixedCapacityStack(int cap)
   { a = (T[ ]) new Object[cap]; }

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

   public void push(String item)     { a[N++] = item; }
   public String pop()               { return a[--N]; }
}
```

Stack – Größenanpassung des Arrays I

- Wenn in der Implementation ein Array fester Länge verwendet wird, muss die maximale Größe des Stapels vorher bekannt sein. Das kann dadurch realisiert werden, dass er ausreichend groß dimensioniert wird, was zu einer Verschwendung von Speicherplatz führt. Trotzdem kann es zu einem Überlauf kommen, was zum Absturz der Anwendung führt. Um das zu verhindern, kann bei der push-Methode überprüft werden, ob noch Platz im Array ist, und gegebenenfalls muss die push-Operation scheitern. Das ist aber ebenfalls unerwünscht, da sich die Anwendung nicht mit dem Problem eines überlaufenden Stapels beschäftigen soll.

- Die Lösung ist, den Speicher für den Stapel zu verdoppeln, falls die push-Methode feststellt, dass kein freier Eintrag mehr vorhanden ist. Die Verdopplung kann mit nachstehender Methode durchgeführt werden.

Stack –Größenanpassung des Arrays II

```
private void resize(int max)
{
    T[ ] temp = (T[ ])new Object[max];
    for (int i = 0; i < N; ++i)
        temp[i] =a[i];
    a  =  temp;
}
```

Die push-Methode sieht nun so aus:

```
public void push(T item)
{
    if (N == a.length)
        resize(2*a.length);
    a[N++] = item;
}
```

Stack – Größenanpassung des Arrays III

- In ähnlicher Weise kann das Array verkleinert werden. Es hat sich herausgestellt, dass das Sinn macht, falls weniger als ein Viertel des Arrays belegt ist; dann kann man die Größe des Arrays halbieren, was ebenfalls mit Hilfe der Methode resize innerhalb der pop-Methode durchgeführt werden kann.
- Es ist zu beachten, dass der Speicher des frei gewordenen Arrays freigegeben werden muss, um die gewünschte Entlastung des Hauptspeichers wirklich zu gewährleisten.

Stack – Iteration I

- Wie bereits früher erwähnt, ist eine der grundlegenden Operationen auf Collections die Java-Anweisung foreach, mit der alle Elemente durchlaufen werden. Ein Anwendungsbaustein kann so aussehen:

```
Stack<String> collection = new Stack<String>();
…
for (String s : collection)
        System.out.println(s);
```

- Diese foreach-Anweisung ist eine Kurzform eines while-Konstrukts:

```
Iterator<String> i  = collection.iterator();
while (i.hasNext())
{
        String s = i.next();
        System.out.println(s);
}
```

Stack – Iteration II

- Dieser Code weist die Bestandteile auf, die in jeder Collection implementiert werden muss:
  Die Collection muss eine iterator()-Methode implementieren, die ein Iterator-Objekt zurück liefert.
  Die Klasse Iterator muss über zwei Methoden verfügen:
  - hasNext(), die einen Boolschen Returnwert hat
  - next(), die ein generische Element aus der Collection zurückliefert.

- Dies wird in der Objekt-orientierten Programmierung durch ein Interface realisiert. In Java ist dazu die Klausel „implements Iterable<T>“ bei der Klassendefinition zu ergänzen.
  Dies soll genutzt werden, um über die Elemente in der Arrayrepräsentation unseres Stacks in umgekehrter Reihenfolge zu iterieren, sodass wir den Iterator ReverseArrayIterator nennen und folgende Methode hinzufügen:
  public Iterator<T> iterator()
  { return new ReverseArrayIterator(); }

Stack – Iteration III

Dazu müssen die durch die Schnittstelle vorgegebenen Methoden implementiert werden:

```
private call ReverseArrayIterator implments Iterator<T>
{
    private int i = N;
    public boolean hasNext()    { return i > 0  }
    public T next()             { return a[--i]; }
    public void remove()        {                }
}
```

Die vorgeschriebene Methode remove() bleibt in unserem Fall leer!

Zusätzlich muss am Anfang des Programms die Zeile
import java.util.Iterator;
eingefügt werden.

Nachfolgend der Code für einen Iterierbaren Stack:

Stack – Iteration IIII

```
import java.util.Iterator;

public class Stack<T> implements Iterable<T>{
    private T[ ] a;
    private int N;

    public Stack(int cap)          { a = (T[ ]) new Object[cap]; }
    public Stack()                 { a = (T[ ]) new Object[1]; }

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

    public void push(T item)
    {
        if (N == a.length)
            resize(2*a.length);
        a[N++] = item;
    }

    public T pop()                  { return a[--N]; }
```

Stack – Iteration V

```
    private void resize(int max) {
        T[ ] temp = (T[ ])new Object[max];
         for (int i = 0; i < N; ++i)
                temp[i] =a[i];
         a  =  temp;
    }

    public Iterator<T> iterator()  { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<T> {
        private int i = N;
        public boolean hasNext()    { return i > 0; }
        public T next()             { return a[--i]; }
        public void remove()        {   }
    }
}
```

9. Queues

Queus – Queue mit verketteter Liste I

- Die Implementierung einer Queue auf der Basis einer verketteten ist ebenfalls nicht sonderlich kompliziert. Sie enthält die Klasse Node, die bereits für den Inhalt eine generische Variable enthält, wie folgt:

```
class Node {
    T item;
    Node next;
}
```

- Der Anfang der Warteschlange wird von der Instanzvariable *first* referenziert und das Ende von der Instanzvariablen last.
  Mit *enqueue()* wird ein Element hinten in der Liste eingefügt – die dafür erforderlichen Anweisungen wurden bereits im Kapitel über Listen besprochen. Dabei werden die beiden Variable first so gesetzt, dass sie auf den neuen Knoten zeigt, falls die Liste leer ist, und die Variable last wird immer auf den neuen Knoten gesetzt.

Queus – Queue mit verketteter Liste II

- Mit dequeue() wird das erste Element der Liste entfernt, wobei first auf das zweite Element der Liste gesetzt wird; falls die Liste nur ein Element enthält, wird first auf null gesetzt. Es wird last nur dann aktualisiert, falls die Liste jetzt leer ist.
- Die Implementierung für isEmpty( ) und size( ) entsprechen denjenigen von Stack.
- Zusätzlich (und optional) wurde die Schnittstelle Iterable implementiert.
- Hier der Code für Queue():

Queus –Queue mit verketteter Liste III

```
import java.util.Iterator;
public class Queue<T> implements Iterable<T>
{
   private Node first;
   private Node last;
   private int N;
   private class Node {
      T item;
      Node next;
   }
   public boolean isEmpty()     { return first == null; }
   public int size()            { return N; }
   public void enqueue(T item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty())            { first = last; }
         else                   {oldlast.next = last;}
      N++;
   }
```

Stack –Queue mit verketteter Liste IIII

```
public T dequeue()
{
    T item = first.item;
    first = first.next;
    if (isEmpty() )
        last = null;
    N--;
    return item;
}
public Iterator<T> iterator()  { return new LinkedListIterator();}

private class LinkedListIterator implements Iterator<T>
{
    private Node current = first;
    public boolean hasNext()    {return current != null;}
    public void remove()          { }
    public T next()
    {
        T item = current.item;
        current = current.next;
        return item;
    }
}
```

Queus – Queue mit verketteter Liste V

Das Anwendungsprogramm, das den Einsatz der Queue zeigt, nutzt jetzt keinen Benutzer-Dialog, sondern ein vorgegebenes Integer-Array. Man beachte, dass das Array nicht aus primitiven Datentypen, sondern aus Referenztypen bestehen muss, um Generics nutzen zu können.

```
public class Programm {
    public static void main(String[] args) {
        Queue<Integer> q = new Queue<Integer>();
        Integer [ ] s1 = {1,2,3,4,5 };
        Integer [ ] s2 = {6,7,8,9,10 };
        for (Integer item: s1)
            q.enqueue(item);
        System.out.println(q.dequeue());
        for (Integer item: s2)
            q.enqueue(item);
        System.out.println("Size: "+ q.size());
        System.out.println(q.dequeue());
//        for (Integer item : q)
//            System.out.println(item);
    }
```

Queus – Queue mit LinkedList

Abschließend folgt eine Implementierung der Queue mittels LinkedList. Diese Implementierung ist leichter. Obwohl die LinkedList die Schnittstelle Iterable implementiert, gilt das jetzt nicht automatisch für Queue. Dies zu ergänzen bleibt als Aufgabe.

```
import java.util.*;
public class Queue<T> {
    private List<T>  q = new LinkedList<T>();
    public boolean isEmpty()      { return q.isEmpty(); }
    public int size()             { return q.size(); }

    public void enqueue(T item)  { q.add(item); }
    public T dequeue()
    {
        T item = q.get(0);
        q.remove(0);
        return item;
    }
}
```

10. Generalized Queus

FHDW
Fachhochschule der Wirtschaft

Generalized Queus – Allgemeines I

- Eine **verallgemeinerte Warteschlange** (auch *Generalized Queue* genannt) ist eine Erweiterung des klassischen Warteschlangenmodells, in dem die Einhaltung der Reihenfolge (FIFO – First In, First Out) aufgehoben oder modifiziert wird, um spezielle Anforderungen zu erfüllen. In verallgemeinerten Warteschlangenmodellen können unterschiedliche Regeln für das Einfügen und Entfernen von Elementen gelten. Diese Modelle werden in vielen Bereichen verwendet, um komplexere Prozesse und Systeme abzubilden, bei denen einfache FIFO-Regeln nicht ausreichend sind..

- Hier sind einige häufige Formen verallgemeinerter Warteschlangen:

Generalized Queus – Allgemeines II

Prioritätswarteschlangen:

Elemente haben Prioritäten, und diejenigen mit der höchsten Priorität werden bevorzugt bedient, unabhängig von ihrer Ankunftszeit. Dies ist eine gängige Struktur in Betriebssystemen oder Notaufnahmen, wo kritische Aufgaben zuerst bearbeitet werden müssen.

Double-Ended Queues (Deque):

Eine Warteschlange, bei der Elemente an beiden Enden eingefügt oder entfernt werden können. Dies ermöglicht flexiblere Operationen und ist eine Verallgemeinerung der einfachen FIFO-Warteschlange.

Generalized Queus – Allgemeines II

Rundpuffer (Circular Queue):

In einem begrenzten Speicherbereich organisiert, wird der letzte Platz des Puffers mit dem ersten Platz verbunden, um eine kreisförmige Struktur zu bilden. Diese wird in Situationen verwendet, in denen Ressourcen effizient genutzt werden müssen, wie z. B. in der Netzwerkkommunikation.

Multilevel-Warteschlangen:

Mehrere Warteschlangen existieren, wobei jede Warteschlange unterschiedliche Prioritätsstufen oder andere Unterscheidungsmerkmale haben kann. Ein Beispiel ist das Prozessmanagement in Betriebssystemen, wo Prozesse je nach Ressourcenbedarf in verschiedenen Warteschlangen verarbeitet werden.

Generalized Queus – Allgemeines III

Verlustsysteme:

Hier können neue Anfragen verworfen werden, wenn die Warteschlange voll ist oder ein bestimmtes Zeitlimit überschritten wird. Diese Warteschlangenform kommt häufig in Telekommunikationssystemen oder Netzwerken zum Einsatz, um Überlastungen zu vermeiden.

Dynamische Warteschlangen:

In dynamischen Warteschlangen kann die Reihenfolge der Elemente auf Basis von externen Bedingungen, wie z. B. Wartezeit oder Dringlichkeit, angepasst werden. Solche Modelle werden in Echtzeitsystemen verwendet, in denen sich die Bedingungen schnell ändern können.

Generalized Queus – Die Priority Queue I

- In Java ist die Klasse **PriorityQueue** eine Implementierung einer Warteschlange, bei der die Elemente nach einer bestimmten Reihenfolge, basierend auf ihrer "Priorität", sortiert werden. Im Gegensatz zu einer normalen Warteschlange, in der das Prinzip **First In, First Out (FIFO)** gilt, werden bei einer PriorityQueue die Elemente in einer bestimmten Reihenfolge aus der Warteschlange entfernt, die durch die natürliche Reihenfolge oder einen benutzerdefinierten **Comparator** definiert wird.
- Wichtige Eigenschaften der PriorityQueue:

Generalized Queus – Die Priority Queue II

Automatische Sortierung:

Elemente in einer PriorityQueue werden in einer aufsteigenden (natürlichen) Reihenfolge oder anhand eines benutzerdefinierten Vergleichs sortiert. Das heißt, das Element mit der "höchsten Priorität" wird an die Spitze der Warteschlange gesetzt und beim Abruf zuerst entfernt.

Kein garantiertes FIFO:

Anders als bei einer normalen Warteschlange wird nicht das erste eingefügte Element zuerst entfernt, sondern das "wichtigste" Element. Wichtigkeit wird durch die Priorität definiert.

Implementierung als Min-Heap:

Intern verwendet die PriorityQueue einen Min-Heap, was bedeutet, dass das Element mit dem **kleinsten** Wert oder der **höchsten** Priorität an der Spitze steht und zuerst entfernt wird.

Generalized Queus – Die Priority Queue II

Keine geordnete Iteration:

Obwohl die PriorityQueue Elemente nach Priorität sortiert, garantiert sie keine vollständige Sortierung der Elemente, wenn man durch die Warteschlange iteriert. Nur das oberste Element (das zuerst entfernt werden kann) ist garantiert das kleinste (oder das mit der höchsten Priorität).

Thread-Sicherheit:

PriorityQueue ist nicht thread-sicher. Für den gleichzeitigen Zugriff von mehreren Threads muss eine Synchronisation, z. B. durch PriorityBlockingQueue, verwendet werden.

Null-Werte nicht erlaubt:

In einer PriorityQueue sind null-Elemente nicht erlaubt, da sie bei der Bestimmung der Reihenfolge Probleme verursachen würden.

Generalized Queus – Priority Queue Methoden

add(E e) und **offer(E e)**: Fügen ein Element zur Warteschlange hinzu. offer wird bevorzugt, da es keinen Fehler wirft, wenn die Warteschlange voll ist.

peek(): Gibt das Element mit der höchsten Priorität zurück, entfernt es aber nicht. Gibt null zurück, wenn die Warteschlange leer ist.

poll(): Gibt das Element mit der höchsten Priorität zurück und entfernt es. Gibt null zurück, wenn die Warteschlange leer ist.

remove(): Entfernt ein bestimmtes Element aus der Warteschlange, falls es existiert.

size(): Gibt die Anzahl der Elemente in der Warteschlange zurück.

isEmpty(): Gibt true zurück, wenn die Warteschlange leer ist.

Generalized Queus – Priority Queue Beispiel

```
import java.util.PriorityQueue;
public class PriorityQueueExample {
    public static void main(String [ ] args) {
 // Erstellen einer PriorityQueue mit natürlicher Reihenfolge (aufsteigend)
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
// Hinzufügen von Elementen
        priorityQueue.add(10);
        priorityQueue.add(20);
        priorityQueue.add(15);
// Abrufen und Entfernen des Elements mit der höchsten Priorität
        System.out.println("Peek: " + priorityQueue.peek()); // Gibt das kleinste Element
zurück, entfernt es aber nicht
        System.out.println("Poll: " + priorityQueue.poll()); // Gibt das kleinste Element
zurück und entfernt es
       System.out.println("Nach dem Poll:");
        while (!priorityQueue.isEmpty()) {
            System.out.println(priorityQueue.poll()); // Entfernt und gibt die
verbleibenden Elemente zurück
        }
    }
}
```

Generalized Queus – Implementierung I

- Hier ist eine Java-Implementierung einer **verallgemeinerten Warteschlange**. In diesem Beispiel verwenden wir eine **Prioritätswarteschlange** als verallgemeinerte Warteschlange. Bei einer Prioritätswarteschlange hat jedes Element eine Priorität, und die Elemente mit der höchsten Priorität werden zuerst verarbeitet, unabhängig davon, wann sie in die Warteschlange eingefügt wurden.
- Die Klasse verwendet eine Kombination aus einer benutzerdefinierten Node-Klasse (die sowohl den Wert als auch die Priorität speichert) und einer Prioritätsregel, die durch die Implementierung des Interfaces Comparable in der Node-Klasse festgelegt wird.
- Die Prioritätswarteschlange entfernt die Elemente in der Reihenfolge ihrer Priorität, unabhängig davon, wann sie hinzugefügt wurden.

Generalized Queus – Implementierung II

Erklärung:
Node-Klasse: Diese generische Klasse speichert das Element (value) und seine Priorität (priority). Durch die Implementierung des Interfaces Comparable<Node<T>> können wir die Priorität von zwei Node-Objekten vergleichen, um die Reihenfolge in der Warteschlange zu bestimmen.

GenralizedQueue--Klasse: Diese Klasse verwaltet eine PriorityQueue<Node<T>>, die Elemente nach ihrer Priorität sortiert. Niedrigere Prioritätswerte werden als "höher" angesehen, d. h. sie werden bevorzugt. Methoden wie enqueue und dequeue ermöglichen das Hinzufügen und Entfernen von Elementen.

main-Methode: In der main-Methode wird die GeneralizedQueue getestet, indem Aufgaben mit unterschiedlichen Prioritäten hinzugefügt werden. Beim Entfernen wird die Prioritätsreihenfolge berücksichtigt, sodass "Aufgabe 2" (Priorität 1) vor den anderen entfernt.

Generalized Queus – Implementierung III

```
import java.util.PriorityQueue;
// Klasse für den generischen Knoten, der das Element und dessen Priorität
speichert
class Node<T> implements Comparable<Node<T>> {
    private T value;
    private int priority;

// Konstruktor
    public Node(T value, int priority) {
        this.value = value;
        this.priority = priority;
    }

   public T getValue()          { return value; }    // Getter für den Value
   public int getPriority()     { return priority; } // Getter für die Priorität

// Vergleichsmethode: Knoten mit niedriger Prioritätsnummer werden bevorzugt
    @Override
    public int compareTo(Node<T> other) {
        return Integer.compare(this.priority, other.priority);
    }
```

Generalized Queus – Implementierung IIII

```
@Override
    public String toString() {  return "Value: " + value + ", Priority: " + priority;   }
}
// Verallgemeinerte Warteschlange mit Prioritätsregel
class GeneralizedQueue<T> {
    private PriorityQueue<Node<T>> queue;
    public GeneralizedQueue() {// Prioritätswarteschlange mit natürlicher Reihenfolge
        queue = new PriorityQueue<>();
    }
// Methode zum Hinzufügen eines Elements mit einer bestimmten Priorität
    public void enqueue(T value, int priority) {
        Node<T> node = new Node<>(value, priority);
        queue.add(node);
    }
// Methode zum Entfernen des Elements mit der höchsten Priorität
    public T dequeue() {
        if (queue.isEmpty()) {
            throw new IllegalStateException("Die Warteschlange ist leer");
        }
        return queue.poll().getValue();
    }
```

Generalized Queus – Implementierung V

```
    // Methode zur Überprüfung des nächsten Elements (ohne es zu entfernen)
    public T peek() {
        if (queue.isEmpty()) {
                throw new IllegalStateException("Die Warteschlange ist leer");
        }
        return queue.peek().getValue();
    }
// Überprüfen, ob die Warteschlange leer ist
    public boolean isEmpty() {  return queue.isEmpty(); }
}
// Testklasse
public class GeneralizedQueueExample {
    public static void main(String[] args) {
        GeneralizedQueue<String> queue = new GeneralizedQueue<>();
// Hinzufügen von Elementen mit unterschiedlichen Prioritäten
        queue.enqueue("Aufgabe 1", 3);
        queue.enqueue("Aufgabe 2", 1);
        queue.enqueue("Aufgabe 3", 2);
// Entfernen und Anzeigen der Elemente basierend auf der Priorität
        while (!queue.isEmpty()) {
            System.out.println("Entferntes Element: " + queue.dequeue());
        }
    }
}
```

11. Anwendungen

FHDW
Fachhochschule der Wirtschaft

Anwendungen

Anwendungsbereiche verallgemeinerter Warteschlangen:

- Betriebssysteme (z. B. Task-Scheduling).
- Netzwerkverkehr (z. B. Routing von Paketen).
- Produktionsplanung (z. B. Bearbeitung von Aufträgen in Abhängigkeit von Ressourcen und Prioritäten).
- Kundenservice (z. B. VIP-Kunden haben Vorrang).

Verallgemeinerte Warteschlangen helfen dabei, flexiblere und anwendungsorientierte Lösungen zu finden, wenn die Standard-FIFO-Struktur nicht ausreicht.

12. Grundzüge paralleler Algorithmen

FHDW
Fachhochschule der Wirtschaft

Parallele Algorithmen – Einführung

- **Parallele Algorithmen** sind Algorithmen, die darauf ausgelegt sind, Aufgaben gleichzeitig (parallel) auf mehreren Prozessoren oder Recheneinheiten auszuführen. Sie sind eine Schlüsseltechnologie für die Nutzung moderner Mehrkernprozessoren und verteilte Systeme wie Supercomputer und Cloud-Umgebungen. Im Gegensatz zu sequenziellen Algorithmen, die eine Aufgabe Schritt für Schritt abarbeiten, teilen parallele Algorithmen die Arbeit in mehrere Teile auf, die unabhängig voneinander verarbeitet werden können.

Parallele Algorithmen – Parallelität, Granularität

- **Parallelität** bedeutet, dass mehrere Berechnungen gleichzeitig ausgeführt werden können. Diese Berechnungen können entweder auf einem Mehrkernprozessor, einer GPU (Graphics Processing Unit) oder in einem verteilten System (Cluster, Cloud) ablaufen.

- **Granularität** beschreibt das Verhältnis zwischen der Anzahl der Rechenschritte und der Kommunikation zwischen den parallel arbeitenden Prozessen. Man unterscheidet:
  - **Feingranulare Parallelität**: Viele kleine Aufgaben, die oft miteinander kommunizieren müssen.
  - **Grobgranulare Parallelität**: Weniger, größere Aufgaben, bei denen weniger Kommunikation nötig ist.

Parallele Algorithmen – Partitionierung

- Eine der wichtigsten Aufgaben bei parallelen Algorithmen ist die **Partitionierung** der Daten und der Berechnungen in unabhängige Teilprobleme, die parallel ausgeführt werden können. Man unterscheidet zwei Hauptarten:.
  - **Datenparallelität**: Die Daten werden in mehrere Teile aufgeteilt, und auf jedem Teilstück wird der gleiche Algorithmus angewendet. Ein Beispiel wäre die parallele Verarbeitung von Elementen eines Arrays wie die Matrizenmultiplikation.
  - **Aufgabenparallelität**: Unterschiedliche Aufgaben (oder Berechnungsschritte) werden unabhängig voneinander auf mehreren Prozessoren ausgeführt.

Parallele Algorithmen – Synchronisation

- In vielen parallelen Algorithmen müssen die verschiedenen Prozesse miteinander kommunizieren, um Ergebnisse auszutauschen. In einem verteilten System geschieht dies über Nachrichtenübermittlung (Message Passing), während in Mehrkernsystemen gemeinsam genutzter Speicher (Shared Memory) verwendet werden kann.
- **Synchronisation** ist der Prozess, bei dem parallele Aufgaben aufeinander abgestimmt werden. Beispielsweise müssen oft alle Prozesse auf ein bestimmtes Ergebnis warten, bevor sie fortfahren können. Häufig verwendete Mechanismen für die Synchronisation sind Sperren (Locks), Barrieren oder Semaphoren.

Parallele Algorithmen – Load Balancing und Korrekheit

- Bei parallelen Algorithmen ist es wichtig, die Arbeit so gleichmäßig wie möglich auf alle Prozessoren zu verteilen, um die Rechenleistung optimal zu nutzen. Ein Ungleichgewicht (z. B. ein Prozess ist schneller fertig als die anderen) führt zu ineffizienter Auslastung. Lastverteilungstechniken sollen dies vermeiden.

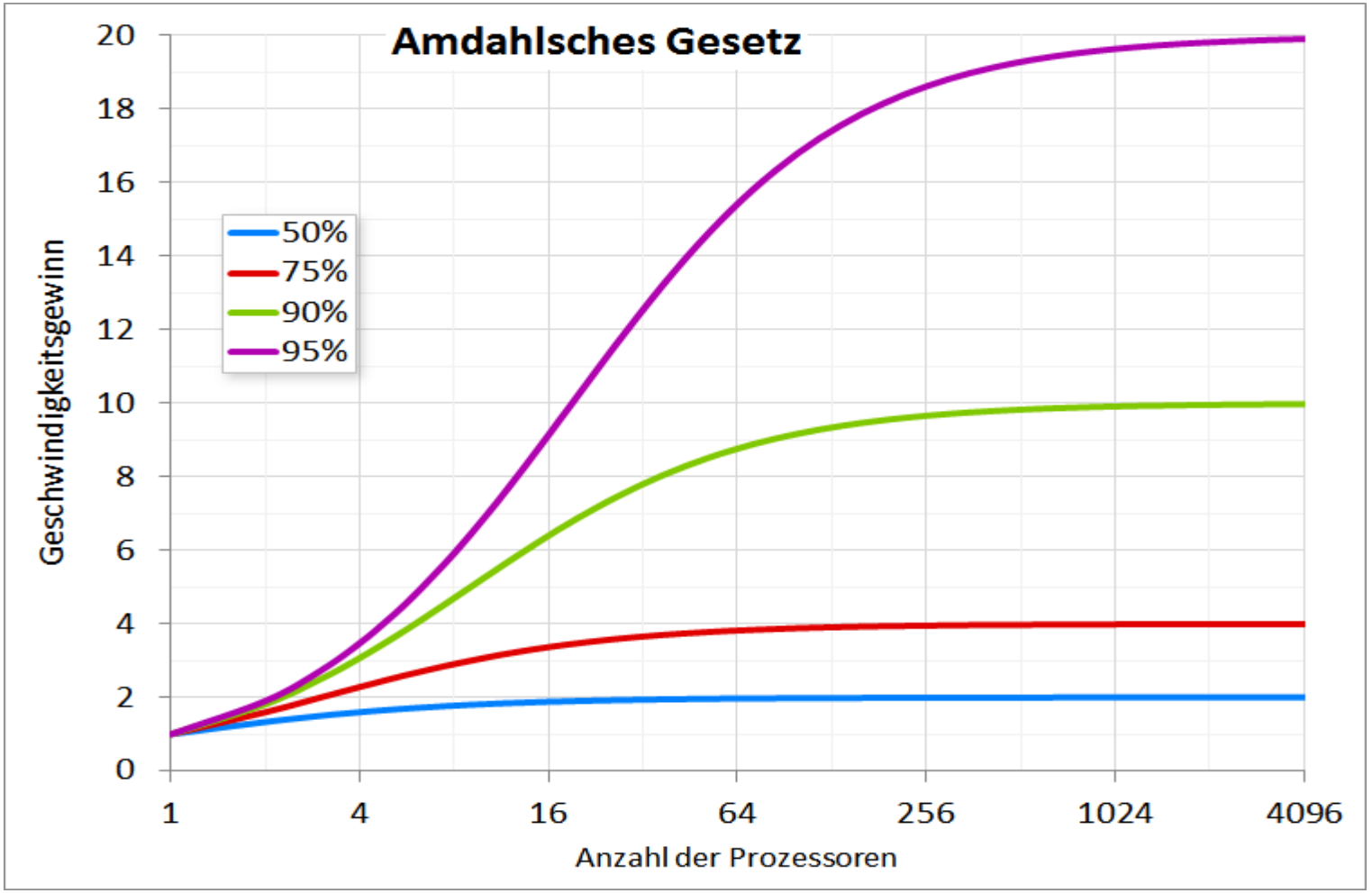
- Bei parallelen Algorithmen ist es entscheidend, sicherzustellen, dass mehrere Prozesse nicht gleichzeitig auf dieselben Ressourcen (z. B. Variablen) zugreifen, ohne korrekt synchronisiert zu sein. Solche fehlerhaften Zugriffe können zu **Race Conditions** führen, bei denen das Ergebnis davon abhängt, welcher Prozess zuerst auf eine Ressource zugreift. Um dies zu verhindern, verwendet man Synchronisationsmechanismen wie Sperren.

Parallele Algorithmen – Speedup

- **Speedup** beschreibt, wie viel schneller ein paralleler Algorithmus im Vergleich zu einem sequentiellen Algorithmus ist. Idealerweise sollte ein Algorithmus auf N Prozessoren N-mal schneller sein; dies wird jedoch selten erreicht.

- **Amdahl‘s Gesetzt** beschreibt die theoretische Grenze des Speedups durch Parallelisierung. Es besagt, dass der maximale Speedup durch den nicht-parallelisierbaren Anteil des Algorithmus begrenzt wird. Wenn ein Teil des Algorithmus nicht parallel ausgeführt werden kann, wird der Gesamtspeedup eingeschränkt, selbst wenn viele Prozessoren zur Verfügung stehen.

Parallele Algorithmen – Amdahls Gesetz



13. Generisches beim Sortieren

FHDW
Fachhochschule der Wirtschaft

Generisches Sort – Allgemeines I

- Das Hauptaugenmerk in diesem Kapitel liegt auf Algorithmen zum Umordnen von Elementen, bei denen die Elemente über einen Schlüssel (key) verfügen. Ziel des Sortieralgorithmus ist es, die Elemente so anzuordnen, dass deren Schlüssel einer wohldefinierten Sortierregel genügen. Das heißt, es soll das Array so umgeordnet werden, dass der Schlüssel für jedes Element mit einem niedrigeren Index nicht größer ist als der Schlüssel jedes Elementes mit einem größeren Index. Die speziellen Eigenschaften der Schlüssel und Elemente können sich von Anwendung zu Anwendung stark unterscheiden. Die Unterscheidung der Schlüssel wird durch eine spezielle Methode (typischerweise basierend auf Comparable) definiert.
- Der Vorteil dieser Vorgehensweise ist es, dass wir in den verschiedene Beispielen nicht auf verschiedene Arten des Vergleichens und Vertauschens eingehen müssen. Nachteile?

Generisches Sort – Allgemeines II

- Die Methode *sort* übernimmt das Sortieren des Arrays. Es benutzt die beiden Hilfsmethoden *less()* und *exch()* (zusammen mit möglicherweise weiteren Hilfsmethoden). Die Anwendung mit der Methode *main( )* holt Daten durch Eingabe von der Standardeingabe oder gibt das Array einfach vor. Das Array wird mit *sort( )* sortiert und mit der privaten Methode *show( )* (zumindest partiell) ausgegeben.
  Es werden in der Methode *sort* unter anderem die Sortiermethoden BubbleSort, SelectionSort, InsertionSort, MergeSort, QuickSort implementiert.

- Bis auf wenige Ausnahmen greift der Sortieralgorithmus nur über zwei Methoden auf die Daten zu. Die Methode *less( )* vergleicht zwei Elemente und die Methode *exch( )* vertauscht sie. Die beiden Methoden sind dank der Schnittstelle Comparable leicht zu implementieren.

Generisches Sort – Java-Code

```
 public static void main( String [ ] args) {
        Integer [ ] a = new Integer[50000];
// hier kommt eine Initialisierung des zu sortierenden Arrays
        long start = System.currentTimeMillis();
        Sort(a);
        long elapsed = System.currentTimeMillis() – start;
        System.out.println("Verstrichene Zeit(Millisekunden): " + elapsed);
        System.out.println("Sortiert: "+ isSorted(a));
        show(a);
 }
public static void sort(Comparable[ ] a) {     }
private static boolean less(Comparable v, Comparable w)
{ return v.compareTo(w) < 0; }  // liefert true, falls v < w
private static void exch(Comparable[ ] a, int i, int j)
{ Comparable t = a[i]; a[i] = a[j]; a[j] = t; }
private static void show(Comparable[ ] a)
{  for (int i = 0; i < Math.min(15, a.length); ++i)
            System.out.printf(a[i] + " ");  }
private static boolean isSorted(Comparable[] a)
{  for (int i = 1; i < a.length; ++i)
            if (less(a[i], a[i-1]) == true)
                return false;
        return true;
 }
```

Generisches Sort – Bemerkungen I

- Diese Klasse veranschaulicht die Konventionen von Sortieralgorithmen für Arrays. Zu jedem Sortieralgorithmus wird eine sort( )-Methode definiert in einer Klasse, die auf den Methodennamen hinweist.

- **Verifizierung**
  Es ist sinnvoll zu überprüfen, ob das Array wirklich so sortiert ist, wie es gewünscht ist. Dazu dient in der Anwendung die Methode isSorted( ), die genau darüber informiert. Es wird empfohlen , diese Methode in der Entwicklungsphase der Software aufzunehmen.

- **Laufzeit**
  Zusätzlich beschäftigen wir uns mit der Performance der Algorithmen. Dazu werden der Startzeitpunkt und der Endzeitpunkt des Algorithmus gemessen; die Laufzeit als Differenz beider Zeiten wird ausgegeben.

Generisches Sort – Bemerkungen II

- **Kostenmodell für das Sortieren**
  Bei der Analyse der Sortieralgorithmen wird die Anzahl der grundlegenden Operationen (Vergleichs- und Tauschoperationen) oder vielleicht die Häufigkeit der Schreib- und Lesezugriffe auf das Array ermittelt. Anschließend werden diese Werte mit der Hypothesen über das jeweilige Laufzeitverhalten vergleichen. Mit diesen Konventionen wird die Überprüfung der Hypothesen zur Performance erleichtert.

- **Zusätzlicher Speicherbedarf**
  Der zusätzliche Speicherbedarf ist oft genauso wichtig wie die Ausführungszeit. Es wird grundsätzliche zwischen zwei Arten von Sortieralgorithmen unterschieden, die in-place sortieren und keine zusätzliche Speicherbedarf benötigen und Algorithmen, die so viel zusätzlichen Speicher benötigen, da sie eine Kopie des zu sortierenden Arrays anlegen.

Generisches Sort – Beispiel BubbleSort I

- Start: unsortiertes Array
- Es werden immer zwei benachbarte Elemente verglichen (blau) und vertauscht, falls nötig
- Nach jedem Durchlauf ist ein Element mehr sortiert (grün)

- Nach dem ersten Durchlauf ist (mindestens) das letzte Element korrekt (*Bubble* = hochgestiegene Luftblase)
- Nach spätestens N –1 Durchläufen sind die letzten N – 1 Elemente korrekt, also alle (!)
- Im i-ten Durchlauf muss man von der 0ten bis zur (N – 1 – i)-ten Stelle laufen ➔ wichtig für C-Code

	0	1	2	3	4	
0	4	3	1	5	2	
1	3	4	1	5	2	1. Durchlauf
2	3	1	4	5	2	
3	3	1	4	5	2	
4	3	1	4	2	5	
5	1	3	4	2	5	2. Durchlauf
6	1	3	4	2	5	
7	1	3	2	4	5	
8	1	3	2	4	5	3.
9	1	2	3	4	5	
10	1	2	3	4	5	4.

Generisches Sort – Beispiel BubbleSort II

```
private static void BubbleSort(Comparable [ ] a)
{
        int i, j;
 //       boolean swapped;
        for (i = 1; i < a.length; i++) {
 //           swapped = false;
            for(j = 0; j < a.length - i; j++) {
                if(! less(a[j],a[j+1])) {
                    exch(a, j, j+1);
//                      swapped = true;
                }
            }
//            if ( ! swapped )
//                break;
        }
}
```

Generisches Sort – Beispiel BubbleSort III

- Wenn bei einem Durchlauf keine Vertauschung mehr vorgenommen wurde, ist kein weiterer Durchlauf nötig
- Bei großen Arrays tritt dieser Fall häufig auf, da mit jedem Durchlauf die Ordnung der vorderen Elemente „verbessert“ wird

➔ Abbruchkriterium in die Schleifen einfügen
(wie als Kommentar bereits eingefügt)

Generisches Sort – Beispiel BubbleSort IIII

Beispiel: 5 Elemente

1. Durchlauf: 4 Vergleiche
2. Durchlauf: 3 Vergleiche
3. Durchlauf: 2 Vergleiche
4. Durchlauf: 1 Vergleich

➔ Also $1+2+3+4=10$ Vergleiche

Für N Elemente: $1+2+3+\ldots+(N-1) = \frac{N \cdot (N-1)}{2}$ Vergleiche.

Ebenso viele Vertauschungsoperationen können im worst case benötigt werden.

Der average case kommt nahe an den worst case heran.

Im *best case* (alles schon sortiert) brauchen wir $N-1$ Vergleiche.

Mit der O-Notation beschreiben wir das Wachstum durch $O(N^2)$.

14. Sortieralgorithmen

FHDW
Fachhochschule der Wirtschaft

14.1 SelectionSort

FHDW
Fachhochschule der Wirtschaft

SelectionSort - Idee

Bei SelectionSort wird zwischen einem bereits sortierten und einen noch unsortierten Teil des Arrays unterschieden.

In jedem Schritt wird das kleinste Element des unsortierten Teils gesucht und am Ende der bereits sortierten Elemente eingefügt.

Das dort stehende Element kommt an die Stelle, wo das gerade gefundene Minimum stand.

SelectionSort - Ablauf

In jedem Durchlauf wird der unsortierte Teil (rot) durchsucht.

Die blau markierten Elemente werden vertauscht.

Die grünen Elemente sind bereits sortiert.

	0	1	2	3	4
0	4	3	1	5	2
1	1	3	4	5	2
2	1	2	4	5	3
3	1	2	3	5	4
4	1	2	3	4	5

SelectionSort - Implementierung

```
public static void SelectionSort(Comparable [ ] a)
{
        int N = a.length;
        for (int i = 0; i < N; ++i) {
            int min = i;
            for (int j=i+1; j < N; ++j)
                if (less(a[j], a[min]))
                    min = j;
            if (i != min)
                exch(a, i, min);
        }
   }
```

SelectionSort - Analyse

Man benötigt für N Elemente $N-1$ Durchläufe.
In i-ten Durchlauf muss man das Minimum aus $N+1-i$ Elementen finden, was $N-i$ Vergleiche kostet .
Also insgesamt

$$(N-1)+(N-2)+\ldots+(N-(N-1))$$

$$= \quad (N-1)\cdot N - (1+2+\ldots+(N-1))$$

$$= \quad (N-1)\cdot N - (N-1)\cdot N/2$$

$$= \quad \frac{N\cdot(N-1)}{2} \quad \text{Vergleiche}$$

SelectionSort benötigt also in jedem Fall so viele Vergleiche wie BubbleSort im *worst case ;* es führt allerdings höchstens N-1 Vertauschungen durch.
Mit der O-Notation beschreiben wir die Komplexität durch $O(N^2)$.

SelectionSort - Bemerkungen

- Ausführungszeit in Abhängigkeit von der Eingabe:
  Die Technik, das kleinste Element durch einmaliges Durchlaufen des Arrays zu finden, verrät nicht viel darüber, wo das nächstkleinere Element beim nächsten Durchlauf zu finden ist. Diese Eigenschaft kann in einigen Situationen von Nachteil sein. Zu Beispiel kann man darüber überrascht sein, dass das Sortieren eines bereits sortierten Arrays (best case) genau so lange dauern kann wie eines Arrays mit lauter gleichen Schlüsseln oder eines Arrays mit zufällig angeordneten Schlüsseln!

14.2 InsertionSort

FHDW
Fachhochschule der Wirtschaft

InsertionSort – Idee und Ablauf

Auch bei InsertionSort unterscheiden wir einen bereits sortierten und einen noch unsortierten Teil des Arrays
Statt das nächst größere Element zu suchen (wie bei SelectionSort), nehmen wir irgendein nächstes und fügen es an der richtigen Stelle ein

- Der grüne Teil ist bereits sortiert
- Das jeweils nächste Element wird an der richtigen Stelle einsortiert
- Alle dahinter liegenden, bereits sortierten, müssen verschoben werden

	0	1	2	3	4
0	4	3	1	5	2
1	3	4	1	5	2
2	1	3	4	5	2
3	1	3	4	5	2
4	1	2	3	4	5

InsertionSort - Implementierung

```
public static void InsertionSort(Comparable [ ] a)
{
        int N = a.length;
        for (int i = 0; i < N; ++i)
        {
            for   (int j = i; j > 0 && less(a[j], a[j-1]); --j)
                exch(a, j, j-1);
        }
    }
```

InsertionSort - Analyse

Die äußere Schleife wird immer $N-1$ mal durchlaufen.

Beim i-ten Durchlauf benötigt man maximal i Vergleiche.

Also $1+2+3+\ldots+(N-1) = \frac{N \cdot (N-1)}{2}$ Vergleiche im *worst case.*

Im *best case* hat man jedes Mal nur einen Vergleich: $1+1+1+\ldots+1 = N-1$ und keine Vertauschoperationen.

Der *average case* liegt bei $\frac{N \cdot (N-1)}{2}$

Mit der O-Notation beschreiben wir das Wachstum durch $O(N^2)$.

InsertionSort - Bemerkungen

Verbesserungen:
Für die Suche kann man die binäre Suche verwenden statt der linearen (auf den bereits sortierten Elementen)

- Das führt zu weniger Vergleichen, aber die Verschiebung der dahinter liegenden, schon sortierten Elemente muss noch durchgeführt werden

Besondere Eignung für:

- Arrays, bei denen jedes Element nicht weit von seiner endgültigen Position entfern ist;
- Arrays, die durch Anhängen von einem kleinen Array an ein großes sortiertes Array entstanden sind;
- Arrays, bei denen nur einige wenige Elemente nicht an der richtigen Position stehen.

Sortieralgorithmen – Vergleich I

Welcher Algorithmus ist nun schneller? Im Allgemeinen vergleichen wir Algorithmen, indem wir:

- sie implementieren und debuggen,
- ihre grundlegenden Eigenschaften analysieren,
- Experimente durchführen, um die Hypothese zu validieren,
- eine Hypothese über vergleichende Performance formulieren.

Wenn eine Implementierung erfolgt ist, bleibt die Frage nach den Eingabemodellen. Typisch sind:

- sortiertes Array,
- umgekehrt sortiertes Array
- zufällig sortiertes Array

Um Messergebnisse vergleichen zu können, müssen die Hardware vergleichbar und die Implementierungen identisch sein.

Sortieralgorithmen – Vergleich II

Bei allen bisher vorgestellten Algorithmen sieht man, dass der einfache BubbleSort-Algorithmus in den meisten Fällen der langsamste ist.

Dennoch zeigt das Wachstum von Vergleichs- und Tauschoperationen ein ähnliches Verhalten – es wächst quadratisch mit der Größe des Arrays, was wir mit der mathematischen Formel $O(N^2)$ bezeichnen.

Das bedeutet:
Die Laufzeiten von je zwei der vorgestellten Algorithmen unterscheidet sich nur durch einen konstanten Faktor!

14.3 QuickSort

QuickSort - Idee

QuickSort beruht auf dem Prinzip *divide-and-conquer* (lat. divide et impera): Teile und herrsche

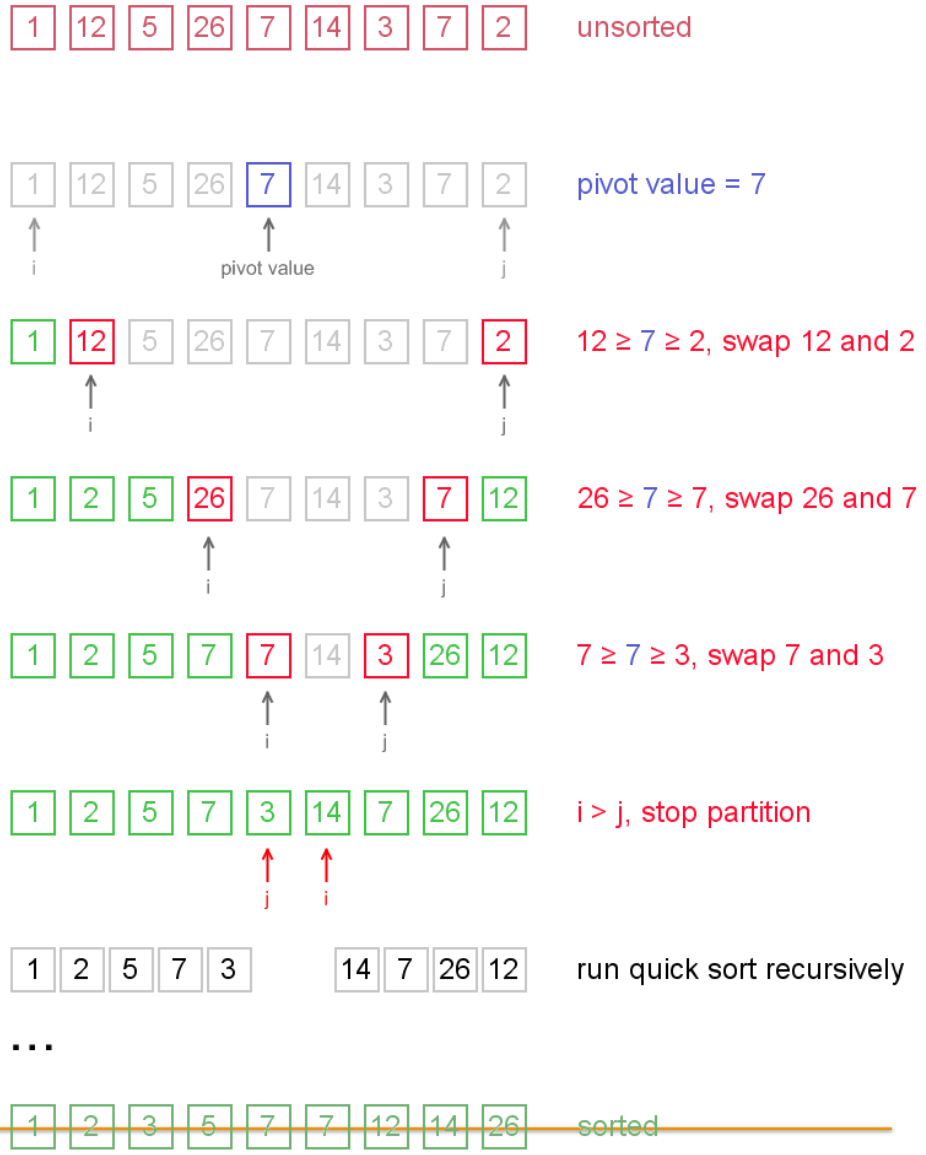
- Man wählt willkürlich ein Element x (Pivot)
- Das Array wird in zwei Teile zerlegt:
  - Im ersten Teil sind alle Elemente kleiner als x
  - Im zweiten Teil sind alle Elemente größer als x
- Diese Teile werden immer wieder weiter nach diesem Muster zerlegt, bis alles sortiert ist.
  ➔ rekursive Funktion!

QuickSort - Ablauf

Zeiger i und j laufen auf einander zu.

In jedem Schritt Vergleich mit Pivot und evtl. Vertauschung

Abbruch, wenn die Zeiger einander überholt haben



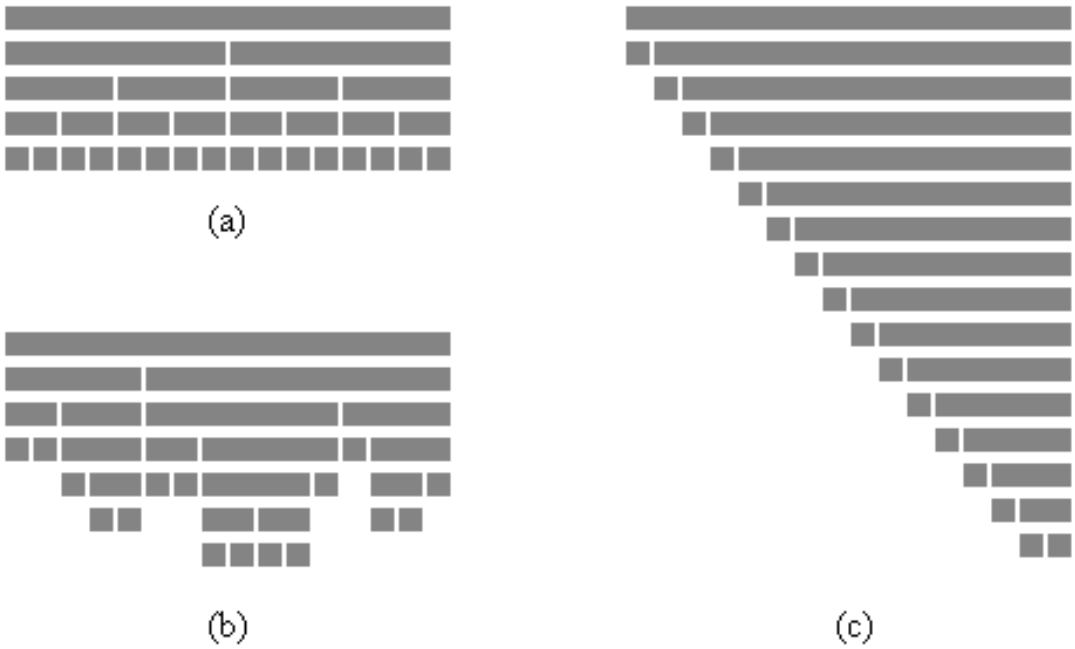
QuickSort - Implementierung

```
public static void QuickSort(Comparable [ ] a, int low, int high)
{
    int i = low, j = high;
    Comparable z = a[(low + high) / 2]; // immer die Mitte

    while(i <= j) {
        while(less(a[i], z) ) i++;          // Index i wandert nach rechts
        while( less(z, a[j]) ) j--;         // Index j wandert nach links
        if (i <= j) {
            exch(a, i, j);                  // Vertauschen
            i++;
            j--;
        }
    }
   // Rekursive Aufrufe
      if(low < j)     QuickSort(a, low, j);
      if(i < high)    QuickSort(a, i, high);
      return;
   }
```

QuickSort - Rekursionstiefe

(a) Best case: $\log_2 N$ Unterteilungen
(b) Average case: $C \cdot \log_2 N$ Unterteilungen
(c) Worst case: N Unterteilungen



QuickSort – Analyse

Auf jeder „Etage“ des Rekusionsbaums müssen alle Elemente durchgegangen werden, also $C \cdot N$ Operationen durchgeführt werden.

Damit muss die Zahl der Etagen (= Zahl der Unterteilungen) multipliziert werden, also

Best case: $C_1 \cdot N \cdot \log_2 N$

Average case: $C_2 \cdot N \cdot \log_2 N$

Worst case: $C_3 \cdot N^2$

(C_i ist dabei jeweils eine passende Konstante)

Mit der O-Notation beschreiben wir das Wachstum mit O(N · log N).

QuickSort – Wahl des Pivotelements

Verschiedene mögliche Strategien:

- das erste oder das letzte:
  - bei unbekannten Daten die einfachste Strategie
  - führt bei sortierten Daten zum worst-case (!)

- das mittlere
  - bei vorsortierten Daten am besten

- den Median des ersten, mittlere und letzten Elements (also das mittlere von der Größe her)

- ein zufälliges
  - vermeidet konstruierte worst-case-Szenarien

QuickSort – Zusammenfassung

- QuickSort ist ein effizientes, schnelles Sortierverfahren mit einer mittleren Komplexität von $C \cdot N \cdot \log_2 N$

- Es hat aber eine sehr schlechte Komplexität im worst case
  - Worst case tritt bei naiver Pivot-Wahl sogar im Fall vorsortierter Elemente auf
  - Worst case ist bei kluger Pivot-Wahl aber sehr unwahrscheinlich

14.4 ShellSort

FHDW
Fachhochschule der Wirtschaft

ShellSort – Idee I

- Dieser Algorithmus basiert auf InsertionSort. InsertionSort ist bei großen ungeordneten Arrays ineffizient, da bei jedem Tausch nur nebeneinanderliegende Elemente vertauscht werden, so dass die Elemente bei jedem Durchgang nur um eine Position im Array verschoben werden. Wenn sich beispielsweise ein Element mit dem kleinsten Schlüssel zufällig am Ende des Arrays befindet, werden N-1 Tauschoperationen benötigt, bis das Element an der richtigen Position steht.
- Schneller geht es mit ShellSort, wenn auch Elemente vertauscht werden, die weiter auseinander liegen. Mit diesem Verfahren lassen sich Arrays so vorsortieren, dass sie am Ende durch InsertionSort effizient sortiert werden können.

ShellSort – Idee II

- Die Idee ist, dem Array durch Umsortieren die Eigenschaft zu verleihen, dass die Folge der jeweils h-ten Elemente eine sortierte Teilsequenz ergibt (h ist eine natürliche Zahl). Eine solche Menge wird als h-sortiert bezeichnet. Durch h-Sortieren für große Werte von h kann man Elemente im Array über große Entfernungen verschieben und damit das Sortieren für kleine Werte von h erleichtern. Mit der Technik kann durch eine Folge absteigender Werte von h – mit 1 endend – ein sortiertes Array erzeugt werden, da aus dem ShellSort bei Abstandswert h=1 nichts anderes als ein InsertionSort wird.
- Solch eine Folge heißt Abstandsfolge. Eine mögliche Abstandsfolge ist $\frac{1}{2}*(3^k-1)$; sie kann bei N/3 beginnen, sie **muss** bei 1 enden. Solch eine Abstandfolge kann in einem Array gespeichert werden.

ShellSort – Idee III

- Beim h-Sortieren des Arrays fügen wir jedes Element unter die vorherigen Elemente seiner h-Teilsequenz ein – einfach indem wir es mit den Elementen vertauschen, die größere Schlüssel haben. (d. h. wir verschieben sie ihrer Teilsequenz jeweils um eine Position nach rechts). Für diese Aufgabe wird der Code von InsertionSort verwendet, nur dass der Abstand bei jedem Durchlauf durch das Array um h und nicht um 1 dekrementiert wird.
- ShellSort verdankt seine Effizienz dem Kompromiss zwischen Größe und teilweiser Ordnung der Teilsequenzen. Am Anfang sind die Teilsequenzen kurz; später sind sie bereits teilweise sortiert. In beiden Fällen ist InsertionSort das Verfahren der Wahl.

ShellSort - Implementierung

```
public static void ShellSort(Comparable [ ] a, int low, int high)
{
      int N = a.length;
       int i, j, k, h;
       Comparable t;
       int [ ] spalten = {21523360, 7174453, 2391484, 797161, 265720, 88573,
29524, 9841, 3280, 1093, 364, 121, 40, 13, 4, 1};
     for (k = 0; k < spalten.length; k++) {
          h = spalten[k];
          for (i = h; i < N; i++)  {  // Sortiere die "Spalten" mit InsertionSort
              t = a[i];
              j = i;
              while (j >= h && less(t, a[j-h])) {
                  exch(a, j, j-h);
                  j = j - h;
              }
              a[j] = t;
          }
       }
       return;
}
```

ShellSort – Ablauf

h,i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
13,13	2	14	13	12	11	10	9	8	7	6	5	4	3	15	1
13,14	2	1	13	12	11	10	9	8	7	6	5	4	3	15	14
4,6	2	1	9	12	11	10	13	8	7	6	5	4	3	15	14
4,7	2	1	9	8	11	10	13	12	7	6	5	4	3	15	14
4,8	2	1	9	8	7	10	13	12	11	6	5	4	3	15	14
4,9	2	1	9	8	7	6	13	12	11	10	5	4	3	15	14
4,10							5				13				
4,10	2	1	5	8	7	6	9	12	11	10	13	4	3	15	14
4,11								4				12			
4,11	2	1	5	4	7	6	9	8	11	10	13	12	3	15	14
4,12									3				11		
4,12	2	1	5	4	3	6	9	8	7	10	13	12	11	15	14

ShellSort – Analyse

- ShellSort hängt stark von der verwendeten Abstandsfolge ab. Aber bis heute gibt es keine wissenschaftlich belegte Formel für das Wachstum bei einem zufällig geordneten Array. Das Wachstum scheint vergleichbar zu dem von QuickSort zu sein.

- Probieren Sie selbst verschiedene Abstandsfolgen aus!

15. Such-Algorithmen

Suchen – Überblick I

- Es gibt eine Reihe von Suchalgorithmen. Hier die gängigen:

1. Lineare Suche (Linear Search):
   Bei der linearen Suche wird jedes Element einer Liste nacheinander überprüft, bis das gesuchte Element gefunden wird oder das Ende der Liste erreicht ist.
   Komplexität: O(n), wobei n die Anzahl der Elemente in der Liste ist. Diese Methode ist einfach, aber ineffizient für große Datenmengen.
2. Binäre Suche (Binary Search):
   Die binäre Suche funktioniert nur auf sortierten Listen. Sie teilt die Liste wiederholt in zwei Hälften und vergleicht das gesuchte Element mit dem mittleren Element. Je nach Ergebnis wird die Suche in der linken oder rechten Hälfte fortgesetzt.
   Komplexität: O(log n). Diese Methode ist viel effizienter als die lineare Suche, erfordert jedoch, dass die Daten vorher sortiert sind.

Suchen – Überblick II

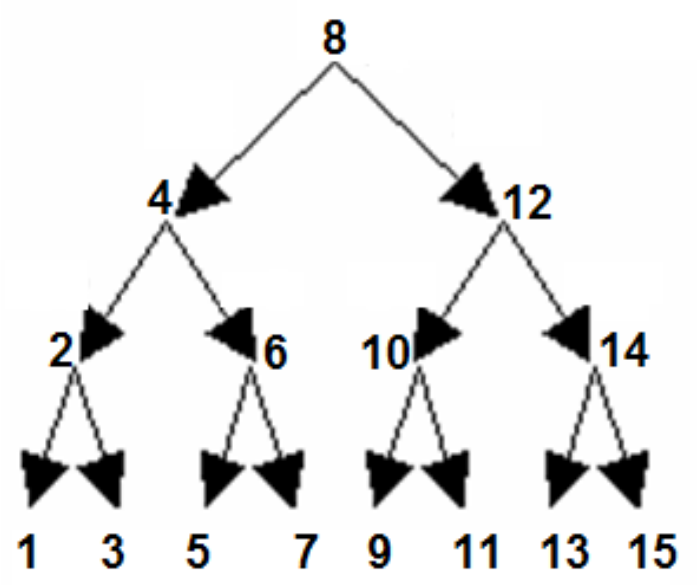
3. Sprungsuche (Jump Search):
   Bei der Sprungsuche wird die Liste in gleich große Blöcke unterteilt. Man springt dann von Block zu Block, bis man einen Block findet, der das gesuchte Element enthalten könnte, und führt dann eine lineare Suche innerhalb dieses Blocks durch.
   Komplexität: $O(\sqrt{n})$. Diese Methode ist nützlich für große, sortierte Listen.
4. Interpolation Search:
   Diese Methode ist eine Verbesserung der binären Suche, die die Position des gesuchten Elements basierend auf dem Wert des Elements schätzt. Sie funktioniert am besten bei gleichmäßig verteilten Daten.
   Komplexität: O(log log n) im besten Fall, O(n) im schlechtesten Fall.

Suchen – Überblick III

5. Exponential Search:
   Diese Methode kombiniert die binäre Suche mit einer exponentiellen Suche, um die Position des gesuchten Elements in einer sortierten Liste zu finden. Zuerst wird die Größe des Suchbereichs exponentiell erhöht, bis das Element gefunden wird oder der Bereich überschritten wird.
   Komplexität: O(log n).
6. Fibonacci-Suche:
   Diese Methode nutzt Fibonacci-Zahlen, um die Suche in einer sortierten Liste durchzuführen. Sie ist ähnlich wie die binäre Suche.

Suche – Beispiel für binäre Suche

- In der Mitte ist Eintrag 8



- Maximal 4 Versuche nötig

Suchen - Anzahl Vergleiche bei binärer Suche

- Das echte Telefonbuch hat vielleicht 200.000 Einträge.
- Wie oft muss man halbieren, um von 200.000 auf 1 zu kommen?
- Gleichwertig: Wie oft muss man verdoppeln, um von 1 auf 200.000 (oder darüber) zu kommen?

  $1 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 \cdot 2 \geq 200.000$

  $2^n \geq 200.000$

  $n \geq \log_2 200.000 = 17{,}6 \approx 18$ (aufgerundet)

- Komplexität: Für N Elemente $\log_2 N$ Schritte (jeweils aufrunden!) Die Komplexität wird mit O(log N) beschrieben.

Suchen - Binäre Suche als Algorithmus

- Wir gehen von einem sortierten Array aus.
- Die binäre Suche kann rekursiv oder durch eine Schleife implementiert werden.
- Die Suchfunktion bekommt ein Array übergeben und verwendet und einen Suchbereich [l..r] (l linke Grenze, r rechte Grenze)
- Ist der Wert in der Mitte (m) größer als der gesuchte, muss man links weiter suchen: Neuer Suchbereich: [l..m-1]
- Ist er kleiner, muss man rechts weitersuchen: Neuer Suchbereich [m+1..r]
- Dies tut man so lange, bis das Element gefunden ist oder der Suchbereich leer ist, also r<l gilt.

Suchen - Binäre Suche (mit Schleife)

```
int binarySearch(int []a){
    int m;
    int  l = 0;
    int r = a.length - 1;
    do {
        m = (l + r)/2;
        if (wert == a[m])
            return m;
        if (wert < a[m])
            r = m-1;
        if (wert > a[m])
            l = m + 1;
    } while (l <= r);
    return -1;
}
```

Suchen - Binäre Suche (rekursiv)

```
public static int binarySearch(int [] a,int wert){
    return binarySearch(a, wert, 0, a.length-1);
}

public static int binarySearch(int []a,int wert,int l,int r)
{
    int m;
    if(r < l)
        return -1;                              //Nichts gefunden
    m = (l + r)/2;
    if(a[m] > wert)
        return binarySearch(a, wert, l, m-1);//links
weitersuchen
    else if(a[m] < wert)
        return binarySearch(a, wert, m+1, r); //rechts
weitersuchen
    else
       return m;              //Wert an der Stelle m gefunden
]
```

16. Hashing

Hashing – Einführung I

- Hashing-Verfahren sind Algorithmen, die Daten beliebiger Größe in einen festen, kompakten Hash-Wert umwandeln. Diese Verfahren sind in der Informatik weit verbreitet und haben verschiedene Anwendungen. Hier sind einige wichtige Aspekte und gängige Hashing-Verfahren:
- **Funktionsweise**
  Ein Hash-Algorithmus nimmt eine Eingabe (z. B. eine Datei, einen Text oder ein Passwort) und wandelt sie in einen Hash-Wert um. Dieser Hash-Wert hat eine feste Länge, unabhängig von der Größe der Eingabedaten.
- **Kollisionsresistenz**
  Ein gutes Hashing-Verfahren sollte sicherstellen, dass es extrem unwahrscheinlich ist, dass zwei unterschiedliche Eingaben denselben Hash-Wert erzeugen (Kollision). Dies ist wichtig für die Integrität der Daten.

Hashing – Einführung II

- **Deterministisch**
  Das bedeutet, dass dieselbe Eingabe immer denselben Hash-Wert erzeugt. Dies ist entscheidend für die Konsistenz und Überprüfbarkeit.
- **Schnelligkeit**
  Hashing-Verfahren sollten schnell sein, um die Effizienz bei der Verarbeitung großer Datenmengen zu gewährleisten.
- **Beispiele** für Hashing-Verfahren
  - MD5 (Message Digest 5): weit verbreitet, erzeugt 128-Bit-Hash-Werte, mittlerweile als unsicher geltend Verfahren
  - SHA-1 (Secure Hash Algorithm 1): Erzeugt 160-Bit-Hash-Werte, wird jedoch ebenfalls als unsicher angesehen.
  - SHA-256: Teil der SHA-2-Familie, erzeugt 256-Bit-Hash-Werte und gilt als sicherer. Verwendung in der Kryptographie
  - SHA-3: Die neueste Hash-Funktion, die eine höhere Sicherheit und Flexibilität bietet.

Hashing – Einführung III

- **Anwendungen**
  - in Datenbanken zur schnellen Suche,
  - in der Kryptographie zur Sicherung von Passwörtern
  - in der Datenübertragung zur Überprüfung der Datenintegrität.

- **Sicherheit/Umkehrbarkeit**
  Hashing ist in der Regel nicht umkehrbar ist, was bedeutet, dass man aus dem Hash-Wert nicht auf die ursprünglichen Daten schließen kann. Dies trägt zur Sicherheit bei, insbesondere bei sensiblen Informationen.

Hashing – Hashtabellen I

- Wenn die Schlüssel kleine Integer-Werte sind, kann man für die Implementierung einer ungeordneten Symboltabelle ein Array verwenden mit dem Schlüssel als Arrayindex. In einer solchen Datenstruktur kann der mit dem Schlüssel i verbundene Wert in dem Arrayeintrag i gespeichert, wo er jederzeit direkt abgreifbar ist.

- Zunächst wird *Hashing* beschrieben - eine Erweiterung dieses einfachen Prinzips, mit der sich auch kompliziertere Typen von Schlüsseln handhaben lassen. Grundlage für den Zugriff auf Schlüssel-Wert- Paare über Arrays sind passende arithmetische Operationen, die die Schlüssel in Arrayindizes umwandeln.

Hashing – Hashtabellen II

- Suchalgorithmen, die Hashing verwenden, bestehen aus zwei Teilen. Im ersten Teil wird eine Hashfunktion berechnet, die den Suchschlüssel in einen Arrayindex umwandelt. Im Idealfall lassen sich voneinander verschiedene Schlüssel auf voneinander verschiedene Indizes abbilden. Dieses Ideal ist im Allgemeinen allerdings nicht zu erreichen, sodass wir mit der Möglichkeit rechnen müssen, dass zwei oder mehr Schlüssel auf den gleichen Arrayindex abgebildet werden. Deshalb besteht der zweite Teil einer Hashsuche aus der sogenannten Kollisionsauflösung, die sich dieser Situation annimmt. Zuerst werden wir beschreiben, wie sich Hashfunktionen berechnen lassen, und danach werden verschiedene Ansätze zur Kollisionsauflösung vorgestellt: lineare Sondierung (und Hashing mit Verkettung).

Hashing – Hashtabellen III

- Hashing ist ein klassisches Beispiel für einen Kompromiss zwischen Rechenzeit und Speicherbedarf . Gäbe es keine Speicherbeschränkungen, könnte man jede Suche mit nur einem Speicherzugriff ausführen, indem wir einfach den Schlüssel als Index in ein (potenziell riesiges) Array verwenden. Dieser Idealfall ist jedoch nicht oft gegeben, da der erforderliche Speicherbedarf nicht vertretbar ist, wenn die Anzahl der möglichen Schlüsselwerte riesig ist. Wenn es andererseits keine Beschränkung hinsichtlich der Rechenzeit gäbe, wäre bei einer sequenziellen Suche in einem ungeordneten Array der Speicherbedarf eventuell nur minimal. Hashing bietet eine Möglichkeit, Rechenzeit und Speicherbedarf in einem vertretbaren Rahmen zu halten, um ein Gleichgewicht zwischen diesen beiden Extremen herzustellen. Um in Hashing-Algorithmen Zeit gegen Speicher abzuwägen, müssen lediglich die Parameter anpasst werden, ohne den Code neu zu schreiben. Dafür wird auf Ergebnisse der Wahrscheinlichkeitstheorie zurückgegriffen.

Hashing – Hashfunktionen I

- Das erste zu lösende Problem ist die Berechnung der Hashfunktion, die die Schlüssel in Arrayindizes umwandelt. Wenn wir ein Array haben, das M Schlüssel-Wert-Paare aufnehmen kann, dann wird eine Hashfunktion benötigt, die jeden gegebenen Schlüssel in einen Arrayindex umwandelt – und zwar in ein Integer im Bereich [0, M-1].

- Gesucht ist eine Hashfunktion, die sowohl leicht zu berechnen ist als auch die Schlüssel gleichmäßig verteilt: für jeden Schlüssel sollte jeder Integer zwischen 0 und M-1 gleich wahrscheinlich sein – ein Idealzustand, der sich nicht unbedingt gleich erschließt.

- Um Hashing zu verstehen sollten man am deshalb zuerst sorgfältig darüber nachdenken, wie eine solche Funktion zu implementieren ist.

Hashing – Hashfunktionen II

- Die Hashfunktion hängt vom Schlüsseltyp ab. Genau genommen benötigt man für jeden Schlüsseltyp eine eigene Hashfunktion.
  Wenn der Schlüssel aus einer Zahl besteht, könnte man mit dieser Zahl direkt loslegen.
  Wenn der Schlüssel aus einem String besteht (beispielsweise aus dem Namen einer Person), muss der String in eine Zahl umgewandelt werden.
  Und wenn der Schlüssel aus mehreren Teilen besteht (zum Beispiel eine Mail-Adresse), müssen die Teile irgendwie kombiniert werden.
  Für viele häufiger benutzte Schlüsseltypen kann man auf Java-Standardimplementierungen zurückgreifen. Dennoch wird kurz auf potentielle Implementierungen für diverse Schlüsseltypen eingegangen.

Hashing – Hashfunktionen - III

- **Positive Integer**
  Das am häufigsten verwendet Verfahren zum Hashen von Integer-Werte wird modulares Hashing genannt. Man wählt für die Arraygröße M eine Primzahl und berechnet für jeden positiven Integer-Schlüssel k den Rest der Division von k durch M (k % M). Wenn M keine Primzahl ist, kann es passieren, dass nicht alle Bits des Schlüssels berücksichtigt werden; damit werden die Werte nicht gleichmäßig verteilt. Angenommen, die Zahlen sind Potenzen von 10 und M sei 10^k, dann werden nur die k niedrigstwertigen Ziffern verwendet. Eine solche Wahl kann Probleme bereiten, wenn zum Beispiel die Schlüssel Telefonvorwahlen sind und M=100 ist. Die meisten Vorwahlen haben in der Mitte eine 0 oder 1, so dass diese Entscheidung die Werte kleiner 20 begünstigt. Ähnliches Beispiel sind die IP-Adressen.
  Daher ist eine Primzahl meistens eine bessere Wahl.

Hashing – Hashfunktionen - IIII

- **Gleitkommazahlen**
  Wenn die Schlüssel reelle Zahlen zwischen 0 und 1 sind, kann einfach mit M multiplizieren und auf die nächste kleinere natürliche Zahl abrunden. Dieser Ansatz hat den Nachteil, dass er den höchstwertigen Bits mehr Gewicht verleiht. Diese Situation lässt sich durch die Anwendung des modularen Hashing auf die Binärdarstellung das Schlüssels lösen.

- **Strings**
  Man behandelt Strings wie riesige Integer – siehe nachstehend: Wenn R größer ist als irgendein Zeichenwert, so besteht die Berechnung darin, den String als N-stelligen Integer zur Basis R zu behandeln und den Rest der Division dieser Zahl durch M zu berechnen.

```
int hash = 0;
for (int i = 0; i < s.length; ++i)
   hash = (R*hash + s.charAt()) % M;
```

Hashing – Hashfunktionen V

Zusammengesetzte Schlüssel

Wenn der Schlüsseltyp aus mehreren Integer-Feldern besteht, kann diese normalerweise so mischen wie gerade bei Strings beschrieben. Angenommen die Suchschlüssel sind vom Typ Date, der drei Integer-Felder aufweist, wobei day 2-stellig, month 2-stellig und year 4-stellig ist. Dann sieht unsere Hashfunktion wie folgt aus:

```
int hash = (((day * R + month) % M)*R +year) % M;
```

Wenn der Wert von R klein genug gewählt ist, so dass kein Überlauf auftritt, ist das Ergebnis wie gewünscht ein Integer zwischen 0 und M-1. In dem Fall kann man die innere Modulo-Operation sparen, indem man eine kleine Primzahl (z. B. 31) wählt.

Hashing – Hashfunktionen in Java I

- Java hilft, das grundlegende Problem zu lösen, dass jeder Datentyp eine Hash-Funktion benötigt, indem sichergestellt ist, dass jeder Datentyp eine Methode namens *hashCode()* erbt, die einen 32-Bit-Integer zurückliefert.
  Dabei muss die Implementierung von *hashCode()* für einen Datentyp konsistent mit *equals* sein. Das heißt, wenn *a.equals(b)* den Wert *true* liefert, dann muss *a.hashCode()* den gleichen numerischen Wert wie *b.hashCode()* liefern.
  Dies erlaubt den Umkehrschluss, dass bei unterschiedlichen *hashCode()*-Werten die Objekte nicht gleich sein können. Wenn die *hashCode()*-Werte dagegen gleich sind, können die Objekte sowohl gleich oder ungleich sein, und man muss mit *equals()* testen, welcher Fall vorliegt.

Hashing – Hashfunktionen in Java II

- Diese Konvention ist die Grundlage dafür, dass Anwendungen eine *hashCode()*-Methode für Symboltabellen verwenden können. Dies bedeutet, dass sowohl *hashCode()* als auch *equals()* überschrieben werden müssen, wenn Hashing auf benutzerdefinierte Typen angewendet werden sollen.
  Die Standardimplementierung liefert die Maschinenadresse des Schlüsselobjekts zurück.
  Erfreulicherweise bietet Java bereits einige vordefinierte *hasCode()*-Implementierungen an, die die Standardimplementierungen für etliche häufig verwendete Typen überschreiben (z. B. String, Integer, Double, etc.).

Hashing – Hashfunktionen in Java III

Das Ziel: ein Array-Index

Da das Ziel ein Arrayindex und kein 32-bit-Integer ist, wird die vorgestellte Implementierung von *hashCode()* durch modulares Hashing kombiniert, um Integer zwischen 0 und M-1 zu erzeugen:

```
private int hash(Key x) {
        return (x.hashCode() & 0x7fffffff) % M;}
```

Der Code maskiert das Vorzeichenbit und berechnet dann den Rest der Division durch M.
Typischerweise werden dabei Primzahlen in der Größenordnung der Hashtabelle M verwendet.

Hashing – Benutzerdefinierter Hashcode I

Es wird erwartet, dass *hashCode()* die Schlüssel gleichmäßig unter den möglichen 32-Bit-Ergebnissen verteilt. Für jedes Objekt kann man demnach x.hashCode() schreiben und erwarten, einen der 2^{32}-möglichen 32-Bit-Werte mit gleicher Wahrscheinlichkeit zu erhalten. Dies ist jedenfalls das Ziel der *hashCode()*-Implementierungen für gängige Datentypen. Für eigene Datentypen muss man selbst dafür sorgen. Das Date-Beispiel veranschaulicht eine mögliche Vorgehensweise: die Instanzvariablen zu Integer umzuwandeln und darauf modulares Hashing anzuwenden.
Die Java-Konvention, dass alle Datentypen eine *hashCode()*-Methode erben, erlaubt noch einen einfacheren Ansatz: die *hashCode()*-Methode auf die Instanzvariablen anwenden, um jede in eine 32-Bit-Integer-Wert umzuwandeln, und dann die arithmetischen Operationen Transaction ausführen.

Man beachte, dass die Instanzvariablen eines primitiven Datentyps in einen Wrappertyp umgewandelt werden müssen, um auf die Methode *hashCode()* zuzugreifen.

Hashing – Benutzerdefinierter Hashcode II

```
public class Transaction
{
        private final String who;
        private final Date when;
        private final double amount;
        public int hashcode()
        {
                int hash = 17;
                hash = 31 * hash – who.hashCode();
                hash = 31 * hash + when.hashCode();
                hash = 31 * hash +((Double)amount).hash.Code();
                return hash;
        }
        …
}
```

Lineares Sondieren – Überblick

- **Lineares Sondieren** ist eine Methode zur **Kollisionsbehandlung** in **Hash-Tabellen**. Eine Kollision tritt auf, wenn zwei unterschiedliche Schlüssel auf denselben Speicherplatz (Index) in der Hash-Tabelle abgebildet werden. Beim linearen Sondieren wird versucht, eine Kollision zu lösen, indem die nächste freie Position in der Hash-Tabelle gesucht wird.

- **Funktionsweise**
  Die einfachste Möglichkeit zur Definition einer solchen Folge besteht darin, so lange den jeweils nächsten Behälter zu prüfen, bis man auf einen leeren Behälter trifft. Die Definition der Folge von Hashfunktionen sieht dann so aus:

  $h_i(x) = (h(x) + i) \bmod m$

Lineares Sondieren – Funktionsweise I

- Die Anwendung des Modulo-Operator hat mit der begrenzten Zahl von Behältern zu tun: Wurde der letzte Behälter geprüft, so beginnt man wieder beim ersten Behälter. Das Problem dieser Methode ist, dass sich so schnell Ketten oder Cluster bilden und die Zugriffszeiten im Bereich solcher Ketten schnell ansteigen. Das lineare Sondieren ist daher wenig effizient. Sein Vorteil ist jedoch, dass – im Gegensatz zu anderen Sondierungsverfahren – alle Behälter der Tabelle benutzt werden.
- Um ein Element x zu finden, berechnet man h(x) und beginnt dort mit der Suche. Man durchläuft das Array, bis entweder das Element gefunden oder ein leerer Behälter erkannt wird. Löschungen sind etwas schwieriger als beim Hashing mit Verkettung, denn man kann nicht einfach eine Suche durchführen und das Element dort löschen, wo man es findet.

Lineares Sondieren – Funktionsweise II

- Löschungen werden häufig mithilfe von sogenannten *Tombstones* umgesetzt. Beim Löschen eines Elements wird markiert, dass der Behälter leer ist und zuvor belegt war. Beim Suchen bleibt man nicht bei einem *Tombstone* stehen. Stattdessen setzt man die Suche fort. Man muss dabei auf das Ende des Arrays achten und gegebenenfalls wieder am Anfang beginnen. Beim Einfügen kann jeder *Tombstone*, auf den man stößt durch einen leeren Behälter ersetzt werden.
- In der Praxis ist lineares Sondieren eine der schnellsten Hashing-Algorithmen. Vorteilhaft ist der geringe Speicheraufwand. Man benötigt lediglich ein Array und eine sehr einfache Hashfunktion. Hervorragende Lokalität: Bei Kollisionen suchen wir nur an benachbarten Orten im Array. Bei der linearen Abtastung kommt es zu erheblichen Leistungseinbußen, wenn der Lastfaktor hoch wird. Die Anzahl der Kollisionen nimmt tendenziell mit der Anzahl der bestehenden Kollisionen zu. Dies wird als primäres Clustering bezeichnet.

Lineares Sondieren – Beispiel I

Nehmen wir an, wir haben eine Hash-Tabelle der Größe 10 und möchten die Schlüssel *12, 22* und *32* speichern. Unsere Hash-Funktion ist:
$h(k) = k \bmod 10$

1. Für den Schlüssel *12* ergibt sich der Index $h(12) = 12 \bmod 10 = 2$. Dieser Platz ist frei, daher wird *12* an Position 2 gespeichert.

2. Für den Schlüssel *22* ergibt sich der Index $h(22) = 22 \bmod 10 = 2$. Dieser Platz ist bereits durch 12 belegt. Beim linearen Sondieren wird der nächste Platz überprüft, d. h. Index 3. Da dieser frei ist, wird 22 an Position 3 gespeichert.

3. Für den Schlüssel *32* ergibt sich der Index $h(32) = 32 \bmod 10 = 2$. Wieder gibt es eine Kollision. Beim linearen Sondieren wird erst Position 3 (bereits belegt) und dann Position 4 überprüft, wo 32 gespeichert wird.

Lineares Sondieren – Beispiel II

Vorteile:

- Einfach zu implementieren.
- Es werden keine zusätzlichen Datenstrukturen benötigt, um Kollisionen zu behandeln.

Nachteile:

- Primäre Clusterbildung: Wenn mehrere Schlüssel auf denselben oder benachbarten Positionen kollidieren, entstehen sogenannte Cluster, also aufeinanderfolgende belegte Positionen. Dies führt zu längeren Suchen, da das lineare Sondieren immer mehr aufeinanderfolgende Plätze prüfen muss.
- Mit der Zunahme der Füllrate der Hash-Tabelle (Anteil belegter Plätze) wird das lineare Sondieren ineffizienter, da immer mehr Kollisionsprüfungen stattfinden.

Lineares Sondieren – Implementierung I

```
import java.util.Arrays;

public class HashTable {
    private int[ ] table;                    // Array zur Speicherung der Werte
    private int size;                        // Größe der Hash-Tabelle

// Konstruktor zur Initialisierung der Hash-Tabelle
    public HashTable(int size) {
        this.size = size;
        table = new int[size];
        Arrays.fill(table, -1);              // -1 bedeutet, dass der Platz frei ist
    }

private int hash(int key) { return key % size; } // Hash-Funktion

public void display() {                      // Methode zum Anzeigen der Hash-Tabelle
        System.out.println("Hash-Tabelle:");
        for (int i = 0; i < size; i++) {
            System.out.println("Index " + i + ": " + (table[i] == -1 ? "leer" : table[i]));
        }
    }
```

Lineares Sondieren – Implementierung II

```
// Methode zum Einfügen eines Schlüssels

    public void insert(int key) {
        int hashValue = hash(key);  // Berechne den Startindex
        int initialHash = hashValue; // Merke den Startindex für den Fall eines Zyklus
// Finde den nächsten freien Platz (mit linearem Sondieren)
        while (table[hashValue] != -1) {
            hashValue = (hashValue + 1) % size; // Verschiebe linear
            if (hashValue == initialHash) { // Wenn wir zurück beim Start sind, ist die
Tabelle voll
                System.out.println("Hash-Tabelle ist voll. Schlüssel " + key + " kann nicht
eingefügt werden.");
                return;
            }
        }
// Füge den Schlüssel an der freien Stelle ein
        table[hashValue] = key;
    }
```

Lineares Sondieren – Implementierung III

```
//  Methode zum Suchen eines Schlüssels

    public boolean search(int key) {
        int hashValue = hash(key);
        int initialHash = hashValue;
 // Suche linear in der Hash-Tabelle nach dem Schlüssel
        while (table[hashValue] != -1) {
            if (table[hashValue] == key) {
                return true; // Schlüssel gefunden
            }
            hashValue = (hashValue + 1) % size;
            if (hashValue == initialHash) { // Wenn wir zurück beim Start sind, beenden
wir die Suche
                return false; // Schlüssel nicht gefunden
            }
        }
        return false; // Schlüssel nicht in der Hash-Tabelle
    }
```

Lineares Sondieren – Implementierung IIII

```
//  Hauptprogramm

    public static void main(String[] args) {
        HashTable hashTable = new HashTable(10);

        hashTable.insert(12);
        hashTable.insert(22);
        hashTable.insert(32);
        hashTable.insert(42); // Erzeugt mehrere Kollisionen
        hashTable.display();
       // Suche nach Schlüsseln
        System.out.println("Suche nach 22: " + (hashTable.search(22) ? "gefunden" :
"nicht gefunden"));
        System.out.println("Suche nach 50: " + (hashTable.search(50) ? "gefunden" :
"nicht gefunden"));
    }
}
```

17. Compiler-Algorithmen

Compiler-Algorithmen– Übersicht I

- Compiler-Algorithmen sind wesentliche Bestandteile eines Compilers, der Quellcode in Maschinensprache übersetzt. Hier sind die Hauptphasen und einige gängige Algorithmen, die in diesen Phasen verwendet werden:

1. Lexikalische Analyse:
   In diesem ersten Schritt wird der Quellcode in Token zerlegt. Token sind die kleinsten bedeutungstragenden Einheiten, wie Schlüsselwörter, Operatoren und Bezeichner. Diese Phase wird oft von einem sogenannten Lexer oder Scanner durchgeführt.

2. Syntaktische Analyse:
   Hier wird die Struktur der Token überprüft, um sicherzustellen, dass sie den grammatikalischen Regeln der Programmiersprache entsprechen. Dies geschieht durch einen Parser, der einen Syntaxbaum (Parse Tree) erstellt, der die hierarchische Struktur des Codes darstellt.

Compiler-Algorithmen– Übersicht II

3. Semantische Analyse:
   In dieser Phase wird überprüft, ob der Code semantisch korrekt ist. Das bedeutet, dass die Bedeutung der Anweisungen und Ausdrücke überprüft wird, z. B. ob Variablen deklariert sind, bevor sie verwendet werden, und ob die Datentypen kompatibel sind.

4. Optimierung:
   Der Compiler versucht, den Code zu optimieren, um die Effizienz zu verbessern. Dies kann sowohl auf der Ebene des Quellcodes als auch auf der Ebene des Zielcodes geschehen. Optimierungen können die Laufzeit oder den Speicherverbrauch des Programms reduzieren.

Compiler-Algorithmen– Übersicht III

5. Codegenerierung:
   In diesem Schritt wird der optimierte Code in die Zielprogrammiersprache übersetzt, oft in Maschinensprache oder eine Zwischensprache. Der Compiler erzeugt den endgültigen Code, der auf der Zielarchitektur ausgeführt werden kann, um die Leistung des generierten Codes weiter zu verbessern.

6. Codeoptimierung:
   Nach der Codegenerierung kann eine weitere Optimierung stattfinden, um die Leistung des generierten Codes weiter zu verbessern.

Compileralgorithmen sind komplex und erfordern ein tiefes Verständnis der Programmiersprachen, der Computerarchitektur und der Algorithmen.

Compiler-Algorithmen– Konkreter Syntaxbaum I

- Ein konkreter Syntaxbaum (auch Parse-Baum genannt) ist eine Datenstruktur, die die vollständige syntaktische Struktur eines Quellcodes oder einer Ausdrucksform darstellt. Im Gegensatz zum abstrakten Syntaxbaum (AST) enthält der konkrete Syntaxbaum alle Details der Syntax, einschließlich der spezifischen Regeln und der Reihenfolge der Elemente, die im Quellcode vorkommen.

- Jeder Knoten im konkreten Syntaxbaum repräsentiert ein sprachliches Element, wie zum Beispiel:

  - Terminale: Die grundlegenden Symbole der Sprache, wie Schlüsselwörter, Operatoren und Literale.

  - Nicht-Terminale: Abstrakte Symbole, die Gruppen von Terminalen oder andere Nicht-Terminale darstellen.

Compiler-Algorithmen– Konkreter Syntaxbaum II

- Die Kanten zwischen den Knoten zeigen die hierarchischen Beziehungen und die Struktur der Sprache an. Der konkrete Syntaxbaum spiegelt die Grammatikregeln wider, die zur Analyse des Quellcodes verwendet werden.

- Ein Beispiel für einen konkreten Syntaxbaum könnte eine einfache mathematische Ausdrucksform wie „3 + 5 * 2“ sein. Der Baum würde die Struktur des Ausdrucks darstellen, wobei der Plus-Operator als Wurzelknoten und die Zahlen sowie der Multiplikationsoperator als untergeordnete Knoten angeordnet sind. In diesem Fall würde der Baum die Priorität der Operationen berücksichtigen, indem er den Multiplikationsknoten höher anordnet als den Additionsknoten.

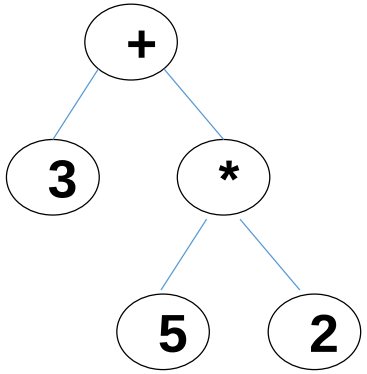
Compiler-Algorithmen– Abstrakter Syntaxbaum (AST) I

- Ein abstrakter Syntaxbaum (AST) ist eine Datenstruktur, die die syntaktische Struktur eines Programms oder einer Ausdrucksform in einer hierarchischen Form darstellt. Im Gegensatz zu einem konkreten Syntaxbaum, der alle Details der Syntax eines Programms enthält, konzentriert sich der AST auf die wesentlichen Elemente und deren Beziehungen zueinander.

- Jeder Knoten im Baum repräsentiert ein sprachliches Element, wie zum Beispiel einen Operator, eine Variable oder eine Funktion, während die Kanten die Beziehungen zwischen diesen Elementen darstellen. Der AST wird häufig in der Compiler-Entwicklung verwendet, um den Quellcode zu analysieren und zu verarbeiten, da er eine vereinfachte und abstrahierte Sicht auf den Code bietet, die für verschiedene Analysen und Transformationen nützlich ist.

Compiler-Algorithmen– Abstrakter Syntaxbaum (AST) II

- Beispiel für die Formel 3 + 5 * 2:
  - Der Wurzelknoten ist der Operator `+`, was bedeutet, dass die gesamte Berechnung mit der Addition beginnt.
  - Der linke Kindknoten ist die Zahl 3.
  - Der rechte Kindknoten ist der Operator `*`, was bedeutet, dass die Multiplikation als nächstes ausgeführt wird.
  - Der linke Kindknoten des Multiplikationsoperators ist die Zahl 5.
  - Der rechte Kindknoten des Multiplikationsoperators ist die Zahl 2.



18. Algorithmen für Numerik/Graphik

FHDW
Fachhochschule der Wirtschaft

Numerische Algorithmen – Überblick I

- Es gibt unzählige Algorithmen. Hier nur einige sehr bekannte:

1. Intervallschachtelung:
   Dieses Verfahren approximiert eine Nullstelle einer stetigen Funktion. Es beruht auf dem Zwischenwertsatz der Analysis.

2. Newton-Verfahren:
   Dieses Verfahren wird verwendet, um die Nullstellen einer differenzierbaren Funktion zu finden. Es basiert auf der Idee, dass man mit einer Näherung für die Nullstelle beginnt und dann iterativ eine bessere Näherung berechnet, indem man die Tangente an die Funktion an der aktuellen Näherung betrachtet.

Numerische Algorithmen – Überblick II

3. Gradientenabstieg:
   Ein Optimierungsalgorithmus, der verwendet wird, um das Minimum einer Funktion zu finden. Der Algorithmus bewegt sich in Richtung des steilsten Abstiegs (d.h. des negativen Gradienten) der Funktion, um die Werte zu minimieren.

4. Gauss-Eliminationsverfahren:
   Ein Algorithmus zur Lösung von linearen Gleichungssystemen. Er transformiert das Gleichungssystem in eine obere Dreiecksform, sodass die Lösungen durch Rücksubstitution leicht gefunden werden können.

5. Euler-Verfahren:
   Ein Verfahren zur numerischen Lösung einer gewöhnlichen Differentialgleichung erster Ordnung. Die Methodik erinnert stark an das Newton-Verfahren.

Numerische Algorithmen – Überblick III

6. Runge-Kutta-Verfahren:
   Eine Familie von Methoden zur numerischen Lösung von gewöhnlichen Differentialgleichungen. Diese Verfahren bieten eine Möglichkeit, die Lösung einer Differentialgleichung schrittweise zu approximieren.

7. Monte-Carlo-Simulation:
   Eine Methode zur statistischen Analyse, die Zufallszahlen verwendet, um Probleme zu lösen, die deterministische Algorithmen schwer handhaben können. Sie wird häufig in der Finanzmathematik, Physik und anderen Bereichen eingesetzt, um Wahrscheinlichkeiten und Risiken zu bewerten.

Numerische Algorithmen – Überblick IIII

8. Fast Fourier Transform (FFT):
   Ein Algorithmus zur effizienten Berechnung der diskreten Fourier-Transformation (DFT) und ihrer Inversen. FFT wird häufig in der Signalverarbeitung und Bildbearbeitung verwendet, um Frequenzanalysen durchzuführen.

9. Interpolationsalgorithmen:
   Verfahren wie die Lagrange-Interpolation oder die spline-Interpolation, die verwendet werden, um Werte zwischen bekannten Datenpunkten zu schätzen. Diese Algorithmen sind nützlich in der Datenanalyse, der numerischen Simulation und beim Computer Aided Design (CAD).
   Einfachstes Beispiel für die Lagrange-Interpolation ist die Bestimmung einer Geraden durch zwei vorgegebene Punkte.

Numerik – Intervallschachtelung

- Intervallschachtelung ist ein Algorithmus, um eine Nullstelle einer stetigen Funktion zu berechnen.
- Zwischenwertsatz der Analysis:
  Es sei f eine stetige reell-wertige Funktion auf einem Intervall [a,b].
  Falls f(a)<0 und f(b)>0 gilt, so existiert ein x aus dem Intervall [a,b] mit f(x)=0.
  Analog: Falls f(a)>0 und f(b)<0 gilt, so existiert ein x aus dem Intervall [a,b] mit f(x)=0.
- Die Intervallschachtelung testet den Funktionswert von f in der Mitte des Intervalls m=(a+b)/2. Wenn bei m nicht die Nullstelle ist, so wird die Suche im Intervall [a,m] bzw. [m,b] fortgesetzt (abhängig vom Vorzeichen von f(m)).

Numerik –Intervallschachtelung Beispiel

```
public static double intervall(double left, double right, double accuracy)  {
   double m = 0.0;
   while((right - left) > accuracy) {
         m = (left + right )/2;
         if (f(m) == 0.0)
            return m;
         if (f(left) < 0.0) {
            if (f(m) < 0.0 )
               left = m;
            else
               right = m;
         }
         if (f(right) < 0.0) {
            if (f(m) < 0.0)
               right = m;
            else
               left = m;
         }
      }
      return m;
}
```

Numerik – Newton-Verfahren I

- Es ist anwendbar zur Bestimmung eine stetig differenzierbaren Funktion f von den reellen Zahlen in die reellen Zahlen. Die Idee ist, dass man die Funktion f durch eine Tangente in einem Punkt approximiert und deren Nullstelle berechnet. Mit dieser Nullstelle der Tangente wird das Verfahren wiederholt, bis die gewünschte Genauigkeit erzielt ist. Die Iterationsformel lautet:
  $x_{n+1} = x_n - f(x_n)/f'(x_n)$
- Wenn man das Verfahren in der Nähe einer Nullstelle der Funktion startet, erzielt man mit wenigen Iterationsschritten ein gutes Ergebnis.
- Problematisch ist es, wenn man in die Nähe eines lokalen Extremums gerät – dann führt die Iteration typischerweise weit von der gesuchten Nullstelle weg.

Numerik – Newton-Verfahren II

- Für eine positive reelle Zahl a wird die Quadratwurzel bestimmt, in dem man die Nullstelle der Funktion f(x) = x*x – a mittels des Newtonverfahrens bestimmt. Wegen f‘(x)=2x erhält man aus als Iterationsformel:
  $x_{n+1} = x_n – f(x_n)/f‘(x_n)=1/2*(x_n+a/x_n)$
  Das Verfahren konvergiert für a>0, sofern x_0 ungleich 0 gewählt wird.

- Das Verfahren kann auch genutzt werden, um Nullstellen von Funktionen zwischen mehrdimensionalen Räumen zu berechnen. Dabei werden partielle Ableitungen sowie die Inverse der Jacobi-Matrix genutzt.

Numerik – Euler-Verfahren

- Zur Lösung des Anfangswertproblems
  $y'(x)=f(x,y)$, $y(x_0)=y_0$
  für eine gewöhnliche Differentialgleichung wähle man die Schrittweite $h>0$ und betrachte die diskreten Stellen
  $x_k = x_0 + kh$ mit $k=0,1,2,3,4\ldots$
  und berechne die Werte
  $y_{k+1}=y_k+hf(x_k,y_k)$, $k=0,1,2,3,4\ldots$

- Die berechneten Werte stellen Approximationen an die tatsächlichen Werte $y(t_k)$ der exakten Lösung des Anfangswertproblems dar. Je kleiner die Schrittweite h gewählt wird, desto mehr Rechenarbeit ist nötig, aber desto genauer werden auch die approximierten Werte.

Numerik – Euler-Verfahren Beispiel

- Als Beispiel wird die explizite DGL erster Ordnung y‘ = x - y mit der Anfangsbedingung y(1)=2 betrachtet. Die Lösung soll im Intervall von 1,0 bis 1,4 mit der Schrittweite h=0,1 approximativ berechnet werden.
  Die Lösung der homogenen DGL lautet C*exp(-x), eine spezielle Lösung der inhomogenen DGL ist x-1. Die Anfangswertbedingung liefert als exakte Lösung der DGL
  y(x)=x-1+2*exp(1-x).
  Die Approximation wird mit Hilfe von Excel durchgeführt:

x	1,0	1,1	1,2	1,3	1,4
y	2,0	1,9	1,82	1,758	1,7122
y‘	-1	-0,8	-0,62	-0,458	0,3122

Numerik – Runge-Kutta-Verfahren

- Wir betrachten ein komplexes Anfangswertproblem mit einer Funktion y:R -> R^d, wobei R^d der kanonische d-dimensionale Vektorraum über den reellen Zahlen ist. Die Funktion y ist also ein d-Tupel aus Funktionen.
  Das Anfangswertproblem
  $y'(x)=f(x,y(x)),\ y(x_0)=y_0,$
  hat typischerweise keine Lösung, die explizit exakt angegeben werden kann.

- Die Schrittweite ist wieder h. Dann ist die Iterationsformel für das s-stufige Runge-Kutta-Verfahren:

$$y_{n+1} = y_n + h \sum_{j=1}^{s} b_j k_j$$

  Die b‘s definieren das jeweilige Verfahren (Integrale über das Intervall), die Zwischenschritte der k‘s entsprechen Auswertungen von f an der rechten Seite des Intervalls.

Numerik – Gauss-Verfahren

- Das Gaußsche Eliminationsverfahren (oder einfach Gauß-Verfahren) ist ein Algorithmus aus der linearen Algebra und Numerik. Es ist ein wichtiges Verfahren zum Lösen von linearen Gleichungssystemen und beruht auf Zeilen- und Spaltenumformungen. Damit wird ein LGS auf Stufenform gebracht, an der die Lösungsmenge durch sukzessive Elimination der Unbekannten gelöst werden kann.
- Die Anzahl der benötigten Operationen ist bei einer nxn-Matrix in der Größenordnung n^3.
- In seiner Grundform ist es aus numerischer Sicht anfällig für Rundungsfehler, aber mit kleinen Modifikationen (Pivotisierung) stellt es ein Standardverfahren zum Lösen eines LGS dar.

Numerik – Gauss-Verfahren Beispiel

- Gelöst werden soll das LGS

  $x_1 + 2x_2 + 3x_3 = 2$

  $x_1 + x_2 + x_3 = 2$

  $3x_1 + 3x_2 + x_3 = 0$

  Das wird durch eine Matrix wie folgt dargestellt:

$$\begin{pmatrix} 1 & 2 & 3 & 2 \\ 1 & 1 & 1 & 2 \\ 3 & 3 & 1 & 0 \end{pmatrix}$$

  Indem man von der zweiten Zeile die erste und von der dritten Zeile das 3-fache der ersten Zeile abzieht, erhält man

$$\begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & -1 & -2 & 0 \\ 0 & -3 & -8 & -6 \end{pmatrix}$$

  Von der dritten Zeile wird das 3-fache der zweiten abgezogen:

$$\begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & -1 & -2 & 0 \\ 0 & 0 & -2 & -6 \end{pmatrix}$$

- Aus der dritten Zeile: $x_3=3$. Aus der zweiten Zeile folgt $x_2=-6$.