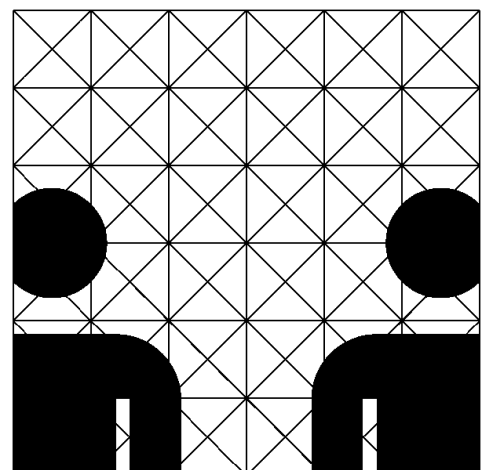


Prüfungsunterlagen
zur Lehrveranstaltung



Teil 4

Universität Hamburg
MIN-Fakultät
Department Informatik
WS 2011 / 2012



Softwareentwicklung I

SE1

Grundlagen objektorientierter Programmierung

Axel Schmolitzky
Heinz Züllighoven
et al.

Teil 4

Verzeichnis der Folien

1. Inhaltliche Gliederung von SE1
- 2. Arrays als speichernahes Sammlungskonstrukt**
3. Ein Beispiel
4. Arrays sind ein speichernahes Konzept
5. Anwendungsbeispiel: Arrays für Bilddaten
6. Anwendungsbeispiel: Spielfelder fester Größe
7. Übersicht: Arrays in Java
8. Arrays sind (fast) Objekte: Array-Variablen
9. Syntax: Array-Deklaration (eindimensional) in Java
10. Erzeugen von Arrays in Java
11. Syntax: Array-Erzeugung (eindimensional) in Java
12. Initialisieren von Array-Zellen in Java
13. Indizierung
14. Schreibender und lesender Zugriff auf Array-Zellen
15. Werte vs. Objekte als Elemente
16. Typischer Fehler: Ungültiger Index (Out of Bounds)
17. Typischer Fehler: Array-Objekt nicht erzeugt
18. Typischer Fehler: Array-Inhalt nicht erzeugt
19. For-Schleifen und Arrays
20. Die erweiterte For-Schleife für Arrays
21. Beispiel: Den maximalen Wert finden
22. Zuweisungen mit Arrays
23. Zuweisungen mit Arrays
24. Kopieren von Array-Objekten
25. Zweidimensionale Arrays
26. Zweidimensionale Arrays erzeugen
27. Zugriff auf zweidimensionale Arrays
28. Kopieren von mehrdimensionalen Arrays
29. Vorteile von Arrays
30. Nachteile von Arrays
31. Vergleich (dynamische) Sammlung und Array
32. Zusammenfassung

33. Klassen und Objekte – revisited

- 34. Klassen in Java sind auch selbst Objekte
- 35. Klassenoperationen
- 36. Klassenoperationen als Dienstleistungen
- 37. Die spezielle Klassenoperation main
- 38. Initialisierung von Klassenobjekten in Java
- 39. Klassenkonstanten
- 40. Klassenoperationen als Dienstleistungen
- 41. Nicht alle Objekte sind Exemplare
- 42. Die Rolle der Klasse in anderen Sprachen
- 43. Klassenmethoden und -variablen in der UML-Notation
- 44. Zusammenfassung

45. Von Sammlungen zu dynamischen Datenstrukturen


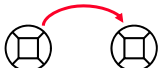

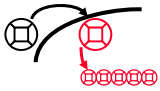
- 46. Implementationen für Sammlungen
- 47. Dynamische Datenstrukturen
- 48. Einteilung dynamischer Datenstrukturen
- 49. Sammlungen implementieren I: Listen
- 50. Zur Erinnerung: der Umgang mit einer Liste
- 51. Listen-Implementationen im Java Collections Framework
- 52. Lineare Datenstrukturen für Listen: Verkettung
- 53. Einfach verkettete Liste
- 54. Doppelt verkettete Liste
- 55. Schema des objektorientierten Entwurfs einer Liste
- 56. Konstruktion eines Kettenglieds
- 57. Definition einer Klasse für Kettenglieder
- 58. Konstruktion einer einfach verketteten Liste
- 59. Einfügen in einer verketteten Liste
- 60. Entfernen aus einer verketteten Liste
- 61. Klassendiagramm einer einfach verketteten Liste
- 62. Doppelt verkettete Liste
- 63. Designalternative Listenenden
- 64. Sonderfälle: Beispiel Listenanfang
- 65. Designalternative Verweise
- 66. Zitat: Good Programmers / Bad Programmierers
- 67. Lineare Strukturen für Listen: „Wachsende“ Arrays
- 68. Wachsende Arrays: Eigenschaften
- 69. Wachsende Arrays: Eigenschaften
- 70. Vergleich der Implementationen
- 71. Einfügen in eine Liste
- 72. Zugriff auf eine beliebige Position; Fazit
- 73. List-Implementationen im Vergleich
- 74. Aufwand für Operationen formalisiert
- 75. Konstanter und variabler Anteil des Aufwandes
- 76. Abschätzungen des Aufwandes
- 77. Ein erster Blick auf die „O-Notation“
- 78. Komplexitäten der Listenoperationen
- 79. Test auf Enthaltensein
- 80. Zusammenfassung

81. Sammlungen implementieren II: Mengen

- 82. Einfügen in Mengen: hoher Aufwand?
- 83. Effiziente Suchverfahren: Anforderungen
- 84. Bäume
- 85. Binäre Suchbäume
- 86. Merkmale von binären Bäumen
- 87. Traversieren von Bäumen
- 88. Breitendurchlauf
- 89. Tiefendurchlauf
- 90. Suchalgorithmus für binäre Suchbäume
- 91. Bäume können entarten
- 92. Balancierte binäre Suchbäume

93. Hash-Verfahren: Die Grundidee
94. Lösungsansatz für Hashing: mehrere Listen statt einer
95. Im Kern: die Hash-Tabelle
96. Ziel: möglichst wenig Überläufe
97. Entscheidend: Die Hash-Funktion
98. Beispiel: Hash-Funktion mit Kollision
99. Hash-Verfahren im Java Collections Framework
100. Indexberechnung für die Hash-Tabelle
101. Hash-Verfahren im Java Collections Framework (II)
102. Set-Implementierungsvarianten im JCF
103. Zusammenfassung
- 104. Mehr zu Sammlungen: Stacks, Queues, Sortieren**
105. SE-I proudly presents: The Stack
106. DAS Standardbeispiel eines dynamischen Datentyps: der Stack
107. Ein Stack-Interface
108. Die Klasse Stack im Java Collections Framework
109. Implementationsskizze: Stack implementiert mit Liste
110. Die Schlange (engl.: Queue)
111. Ein Queue-Interface
112. Anwendung einer Queue: Breitendurchlauf durch Bäume
113. Implementationsskizze für eine Queue als zyklisches Array
114. Ein Interface beschreibt nur Signaturen
115. Ausblick: Abstrakte Datentypen
116. Sortieren
117. Sortierverfahren
118. Komplexität von Sortierverfahren
119. Prinzipien von vergleichsbasierten Sortierverfahren
120. Beispiel Bubble-Sort
121. Implementationsskizze Bubble-Sort
122. Beispiel Quick-Sort
123. Funktionale Definition von Quick-Sort
124. Veranschaulichung Quick-Sort
125. Imperative Konkretisierung: In-Place Quick-Sort
126. Veranschaulichung In-Place Quick-Sort
127. Anmerkungen zu Quick-Sort
128. Lohnt sich die Mühe überhaupt?
129. Sortieren mit linearer Ordnung
130. Parade der Sortierverfahren
131. Zusammenfassung
- 132. Jenseits von Sammlungen: Graphen**
133. Was sind Graphen?
134. Typen von Graphen
135. Graphen: in vielen Anwendungen gebraucht
136. Implementierung von Graphen
137. Implementierung Adjazenzmatrix
138. Implementierung Adjazenzlisten
139. Implementierung Adjazenzlisten
140. Knoten halten eine Menge von Nachbarn
141. Ein Graph hält eine Menge von Knoten
142. Maps: Abbildungen von Schlüssel auf Werte
143. Kantengewichte: Map statt Set in den Knoten
144. Durchläufe in Graphen
145. Durchläufe in Graphen
146. Kürzester Pfad durch einen Graphen
147. Suche nach dem kürzesten Pfad
148. Dijkstras „Wellen“-Algorithmus
149. Dijkstras Algorithmus
150. Beispiel für Dijkstras Algorithmus
151. Zusammenfassung

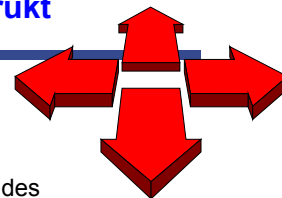
Level 4: Hinter den Kulissen von Sammlungen

Inhaltliche Gliederung von SE1			
Stufe	Titel	Themen u.a.	Woche
1	 „Simple Klasse, simple Objekte“	Klasse, Objekt, Methode, Parameter, Feld, Variable, Sequenz, Zuweisung, Ausdruck, Syntax in EBNF, bedingte Anweisung, primitiver Typ	1 - 4
2	 „Objekte benutzen Objekte“	Typ, Referenz, UML: Klassen- und Objektdiagramme, Schleife, Rekursion, Sichtbarkeit und Lebensdauer, reguläre Ausdrücke	5 - 7
3	 „Schnittstellen mit Interfaces“	Black-Box-Test, Testklasse, Interface, Spezifikation, API, Sammlungen benutzen	8 - 9
4	 „Hinter den Kulissen von Sammlungen“	Arrays, Sammlungen implementieren: Array-Liste, verkettete Liste, Hashing; Sortieren; Stack; Graphen	10 - 14

SE1 – Level 4

1

Arrays als speichernahes Sammlungskonstrukt



- Nachdem wir uns die **Benutzung** von Sammlungen exemplarisch angesehen haben, werden wir für den Rest des Semesters untersuchen, wie dynamische Sammlungen **implementiert** werden können.
- Dazu müssen wir uns zuerst eine grundlegende Datenstruktur zur Implementierung von Sammlungen ansehen: das **Array**.
 - Wir sehen uns zuerst die **Deklaration**, **Erzeugung**, **Initialisierung** und **Benutzung** von **Arrays in Java** an.
 - Wir diskutieren dann die allgemeinen Eigenschaften von Arrays.

SE1 – Level 4

2

Level 4: Hinter den Kulissen von Sammlungen

Ein Beispiel

- Vorgabe: ein Temperaturmessgerät in einer hochseetauglichen Boje.
- Sensoren messen den ganzen Tag über immer wieder die Temperatur.
- Der Speicher ist so begrenzt, dass wir nur 1000 Messwerte speichern können.
- Wir wollen
 - die letzten 1000 Messungen speichern,
 - Maximum und Minimum finden.
- Für die Realisierung in Java bietet sich ein Array an.



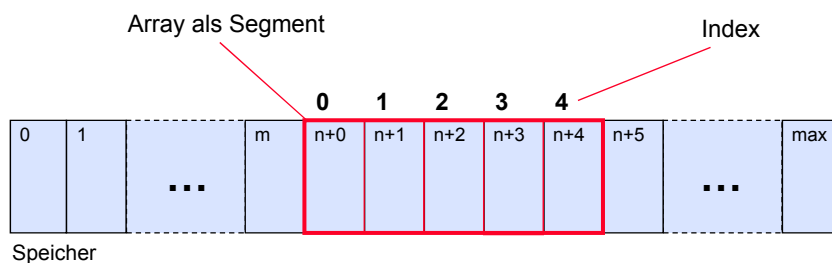
SE1 – Level 4

3

Arrays sind ein speichernahes Konzept



- Arrays sind ein klassisches imperatives Sprachkonzept zur **Sammlung gleichartiger Elemente**. Diese Elemente werden über einen **Index** zugegriffen.
- Arrays dienen meist zur Realisierung von **Listen mit fester Größe**.
- Sie abstrahieren von einem **zusammenhängenden Speicherbereich** samt **indiziertem** Zugriff auf die Speicherzellen.



SE1 – Level 4

4

Level 4: Hinter den Kulissen von Sammlungen

Anwendungsbeispiel: Arrays für Bilddaten



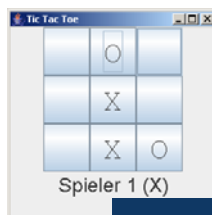
- Bilder sind digital zweidimensionale Strukturen einzelner **Bildpunkte**.
- Die Bildgröße (Bildzeilen und –spalten) ist relativ statisch.
- Informationen über Bildpunkte müssen für Bildverarbeitung **schnell** zugreifbar sein.

SE1 – Level 4

5

Anwendungsbeispiel: Spielfelder fester Größe

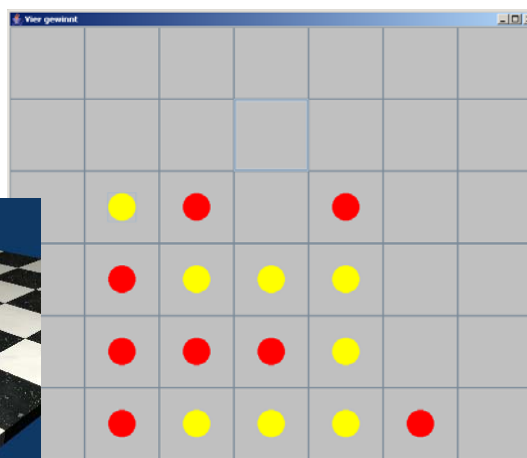
Tic-Tac-Toe: 3x3



Schach: 8x8



Vier gewinnt: 6x7



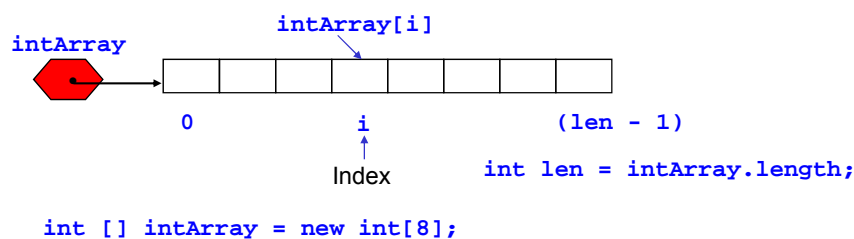
SE1 – Level 4

6

Übersicht: Arrays in Java

Arrays in Java:

- Eine **geordnete Reihung gleichartiger Elemente**.
- Elementtypen können **Basistypen** oder **Referenztypen** sein (auch Referenzen auf andere Arrays).
- Die **Länge eines Array** wird erst beim Erzeugen festgelegt.
- Jeder **Zugriff** über den Index wird **automatisch geprüft**.

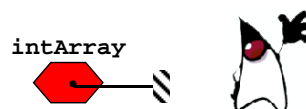


SE1 – Level 4

7

Arrays sind (fast) Objekte: Array-Variablen

- Ein **Array** ist in Java immer ein **Objekt** (Arrays haben alle Eigenschaften, die in der Klasse **Object** definiert sind).
- Eine **Array-Variable** ist immer eine **Referenzvariable**. Der Typ dieser Variablen ist als „**Array von Elementtyp**“ definiert.
- Beispiel einer **Deklaration** eines Arrays von Integer-Werten:
`int[] intArray;`
- Die **Länge/Größe** des Arrays wird in der Deklaration **nicht** angegeben.



SE1 – Level 4

8

Syntax: Array-Deklaration (eindimensional) in Java

Type [] Identifier

Beispiele:

```
int[] numbers; // eine Array-Variable mit primitivem Elementtyp („Array von int“)
```

```
Person[] people; // eine mit einem Objekttyp als Elementtyp („Array von Person“)
```

Erzeugen von Arrays in Java

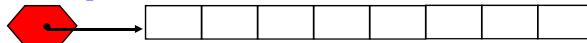
- **Array-Objekte** müssen explizit **erzeugt** werden (wie alle Objekte mit **new**). Dabei kann die Länge definiert oder berechnet werden. Ein einmal erzeugtes Array kann in seiner Länge nicht mehr verändert werden.

```
intArray = new int [8];
```

- Deklaration und Erzeugung können, wie sonst auch, in einem Schritt zusammengefasst werden.

```
int[] intArray = new int [8];
```

intArray



Level 4: Hinter den Kulissen von Sammlungen

Syntax: Array-Erzeugung (eindimensional) in Java

Wie bei einer „normalen“ Objekterzeugung werden auch Arrays durch einen Ausdruck mit dem Schlüsselwort **new** erzeugt.

```
new Type [ LengthExpression ]
```

Beispiele für Ausdrücke:

```
new int[10]           // Erzeugung eines Arrays mit primitivem Elementtyp
```

```
new Person[x]        // Erzeugung eines Arrays mit einem Objekttyp als Elementtyp
```

SE1 – Level 4

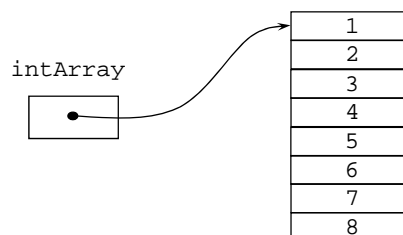
11

Initialisieren von Array-Zellen in Java

- Beim Erzeugen erhalten die **Zellen** eines Arrays in Java die Default-Werte des Elementtyps (für **int** den Wert 0, für **boolean** den Wert **false**, für Referenztypen den Wert **null**, etc.).
- Neben der normalen Zuweisung von Werten kann ein Array auch **implizit erzeugt** und **direkt initialisiert** werden.

```
int[] intArray = {1,2,3,4,5,6,7,8};
```

Diese implizite Erzeugung und Initialisierung mit **geschweiften Klammern** ist ausschließlich bei der Deklaration erlaubt!



SE1 – Level 4

12

Level 4: Hinter den Kulissen von Sammlungen

Indizierung

- Arrays werden **beginnend bei 0** indiziert!!!
- Gültige Indizes eines Arrays sind: 0 ... length - 1
- Beispiel: Ein Array der Größe 8 hat als gültige Indizes 0..7

Gültige Namen für die Elemente des Arrays sind:

- `temperatures[0]`
- `temperatures[1]`
- `temperatures[2]`
- `temperatures[3]`
- ...

temperatures



12.3	0
12.7	1
13.4	2
14.9	3
16.2	4
15.1	5
14.6	6
12.9	7

SE1 – Level 4

13

Schreibender und lesender Zugriff auf Array-Zellen

- Auf die Array-Zellen wird mit eckigen Klammern `[]` zugegriffen:

Schreibender Zugriff

Lesender Zugriff

```
float element = temperatures[0];  
temperatures[3] = 21.2f;  
temperatures[0] = temperatures[1];
```

SE1 – Level 4

14

Level 4: Hinter den Kulissen von Sammlungen

Werte vs. Objekte als Elemente

The diagram illustrates two types of array elements:

- temperatures:** An array where each cell contains a numerical value (e.g., 12.3, 12.7, 13.4, 14.9, 16.2, 15.1, 14.6, 12.9).
- personen:** An array where each cell contains a reference (dot) pointing to a 'Person' object.

• Array-Zellen speichern **Werte** oder **Referenzen auf Objekte** genau wie andere Variablen auch.

SE1 - Level 4 15

Typischer Fehler: Ungültiger Index (Out of Bounds)

• Angenommen, wir haben ein Array der Größe 8.

• Dann schreiben wir:

```
System.out.println(temperatures[8]);
```

temperatures

12.3	0
12.7	1
13.4	2
14.9	3
16.2	4
15.1	5
14.6	6
12.9	7

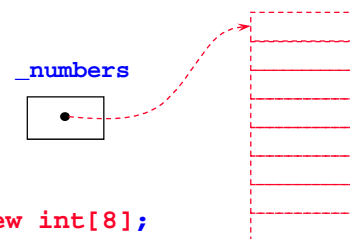
• Es kommt zu einer Fehlermeldung:

ArrayIndexOutOfBoundsException: 8

SE1 - Level 4 16

Typischer Fehler: Array-Objekt nicht erzeugt

- Angenommen, wir haben ein Array als Exemplarvariable deklariert: `private int[] _numbers;`
- Im Konstruktor schreiben wir als Erstes: `_numbers[0] = 42;`
- Es kommt zu einer Fehlermeldung: `NullPointerException`



- Es fehlt die **Erzeugung** des Arrays:

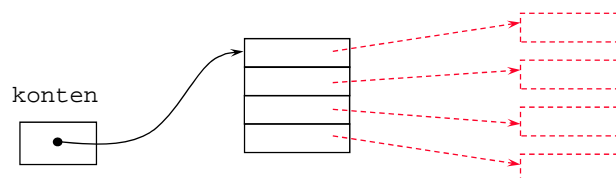
```
_numbers = new int[8];
```

SE1 – Level 4

17

Typischer Fehler: Array-Inhalt nicht erzeugt

- Angenommen, wir haben ein Array mit einem Objekttyp als Elementtyp deklariert und initialisiert: `Konto[] konten = new Konto[4];`
- Unmittelbar danach schreiben wir: `konten[0].einzahlen(123);`
- Es kommt zu einer Fehlermeldung: `NullPointerException`



- Es fehlt die **Erzeugung der Elemente** des Arrays, also der Konto-Objekte.

SE1 – Level 4

18

For-Schleifen und Arrays

- Typischerweise werden For-Schleifen eingesetzt, um alle Elemente eines Arrays zu bearbeiten. Dabei wird die **öffentliche Exemplarkonstante** `length` benutzt.
- Beispiel: Das Ausgeben der Werte eines Arrays.

```
public void printArray(int[] intArray)
{
    for (int i = 0; i < intArray.length; i++)
    {
        System.out.println(intArray[i]);
    }
}
```

Eine solche Standardbenutzung einer For-Schleife für Arrays wird auch als **Programmiermuster** bezeichnet. Im Englischen wird oft der Begriff **idiom** verwendet.

SE1 – Level 4

19

Die erweiterte For-Schleife für Arrays

- Die erweiterte For-Schleife kann auch für Arrays verwendet werden. Dies erspart den Zugriff auf `length`.
- Gleiches Beispiel: Das Ausgeben der Werte eines Arrays.

```
public void printArray(int[] intArray)
{
    for (int k : intArray)
    {
        System.out.println(k);
    }
}
```

Lies auch hier: für jeden int k im intArray tue...

Diese Schleifenart ist nicht geeignet, wenn **am Array selbst** (also an der Belegung der Zellen) etwas geändert werden soll. Außerdem steht im Schleifenrumpf **kein Schleifenindex** zur Verfügung.

SE1 – Level 4

20

Beispiel: Den maximalen Wert finden

```
/**
 * Liefere den maximalen Wert im gegebenen Array.
 */
public int findeMaximum(int[] intArray)
{
    int max = Integer.MIN_VALUE;
    for (int i=0; i < intArray.length; i++)
    {
        if (intArray[i] > max)
        {
            max = intArray[i];
        }
    }
    return max;
}
```

Alternativ: neue For-Schleife

```
{
    int max = Integer.MIN_VALUE;
    for (int k : intArray)
    {
        if (k > max)
        {
            max = k;
        }
    }
    return max;
}
```

Benutzung:

```
int[] myArray = new int[10];
myArray[0] = 20;
myArray[1] = 40;
myArray[2] = 30;
int maxi = findeMaximum(myArray);
System.out.println(maxi);
```

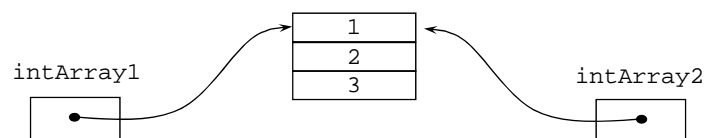
SE1 – Level 4

21

Zuweisungen mit Arrays

- Die Zuweisung einer Array-Variablen kopiert **nur eine Referenz!**
- Beispiel:

```
int[] intArray1 = { 1, 2, 3 };
int[] intArray2 = intArray1;
```
- Beide Referenzen verweisen nun auf **dasselbe Array-Objekt**:



- Wenn man eine **Kopie des Array-Objektes** benötigt, dann gibt es zwei Möglichkeiten:
 - Elementweise in ein neues Array-Objekt kopieren.
 - Die Operation **clone** verwenden.

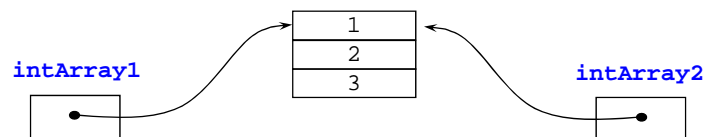
SE1 – Level 4

22

Zuweisungen mit Arrays

- Die Zuweisung einer Array-Variablen kopiert **nur eine Referenz!**
- Beispiel:

```
int[] intArray1 = { 1, 2, 3 };  
int[] intArray2 = intArray1;
```
- Beide Referenzen verweisen nun auf **dasselbe Array-Objekt**:



- Wenn man eine **Kopie des Array-Objektes** benötigt, dann gibt es zwei Möglichkeiten:
 - Elementweise in ein neues Array-Objekt kopieren.
 - Die Operation **clone** verwenden.

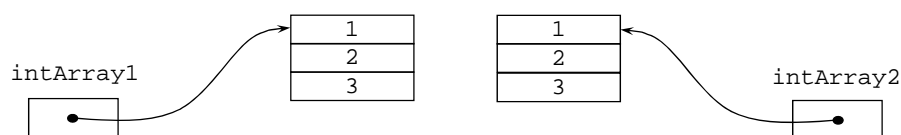
Kopieren von Array-Objekten

- Neues Array-Objekt selbst erzeugen und elementweise kopieren:

```
int[] intArray1 = { 1, 2, 3 };  
int[] intArray2 = new int[intArray1.length];  
for (int i=0; i < intArray1.length; i++)  
{  
    intArray2[i] = intArray1[i];  
}
```

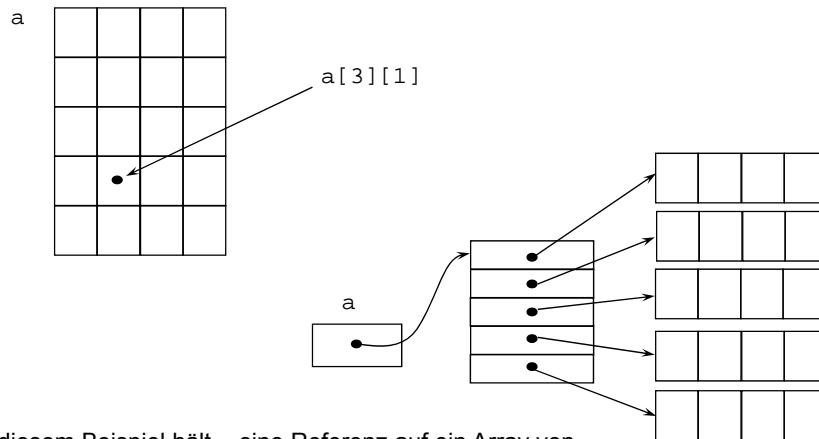
- Die Operation **clone** verwenden:

```
int[] intArray1 = { 1, 2, 3 };  
int[] intArray2 = intArray1.clone();
```



Zweidimensionale Arrays

Zweidimensionale Arrays in Java sind Arrays von eindimensionalen Arrays.



In diesem Beispiel hält **a** eine Referenz auf ein Array von Zeilen; der erste Index benennt die Zeile, der zweite die Spalte.

SE1 – Level 4

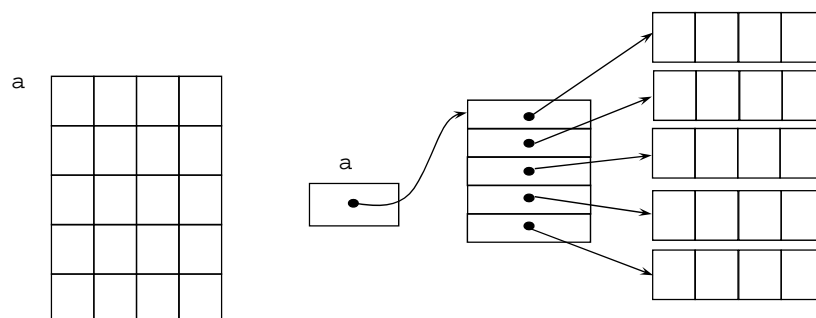
25

Zweidimensionale Arrays erzeugen

```
int[][] a;  
a = new int[5][];  
for (int i=0; i<5; i++)  
{  
    a[i] = new int[4];  
}
```

Mit **new** ist es auch möglich, ein zweidimensionales Arrays direkt zu erzeugen.

Also: **new int[5][4]**



SE1 – Level 4

26

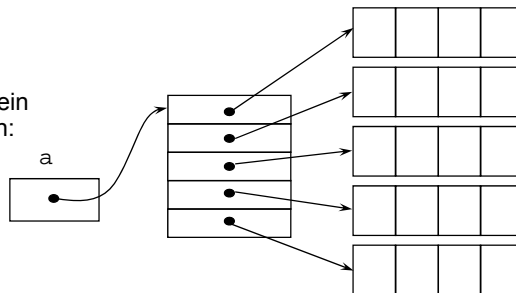
Zugriff auf zweidimensionale Arrays

```
// Alle Array-Elemente auf den Wert 7 setzen

for (int row=0; row < a.length; row++)
{
    for (int column=0; column < a[row].length; column++)
    {
        a[row][column] = 7;
    }
}
```

Mit Hilfe von Initializern kann auch direkt ein zweidimensionales Array angelegt werden:

```
int[][] a = { {7,7,7,7},
               {7,7,7,7},
               {7,7,7,7},
               {7,7,7,7},
               {7,7,7,7} };
```

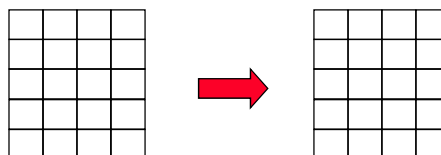


SE1 – Level 4

27

Kopieren von mehrdimensionalen Arrays

- Bei mehrdimensionalen Arrays ist es prinzipiell wie bei eindimensionalen:
 - Es kann „von Hand“ kopiert werden, indem für jede Dimension die notwendigen Arrays erzeugt und deren Inhalte kopiert werden.
 - Oder es wird die Operation `clone` benutzt. Dabei ist zu beachten:
 - `clone` ist für Arrays nicht rekursiv implementiert!
 - D.h., bei einem Aufruf von `clone` auf einer Array-Variablen für ein mehrdimensionales Array wird nur das Array auf der obersten Ebene kopiert.
 - Wenn eine vollständige Kopie gewünscht ist, dann muss „von Hand“ für alle Dimensionen geklont werden.
- In jedem Fall muss die Objektstruktur von Arrays gut verstanden sein!



SE1 – Level 4

28

Level 4: Hinter den Kulissen von Sammlungen

Vorteile von Arrays

- Aufgrund der speichernahen Modellierung eines zusammenhängenden Speicherbereichs haben Arrays vor allem **Effizienzvorteile**:
 - Der Zugriff auf ein Element kann direkt auf einen **Index-Zugriff** der unterliegenden Rechnerarchitektur abgebildet werden; dies ist sehr effizient.
 - Auch das **Kopieren eines ganzen Arrays** kann direkt auf unterliegende Maschinenbefehle abgebildet werden und ist damit sehr effizient.
- **Arrays in Java** haben gegenüber den dynamischen Sammlungen außerdem den Vorteil, dass sie **auch elementare Typen** als Elementtyp zulassen.



SE1 – Level 4

29

Nachteile von Arrays

- In klassischen imperativen Sprachen (wie Pascal) ist die Größe eines Arrays bereits **zur Übersetzungszeit festgelegt**. Dies muss kein Nachteil sein, denn bei etlichen Anwendungen ist die Größe einer Datenstruktur fachlich festgelegt (Beispiel: **Schachbrett**).
- Für ein Array in Java muss die Größe erst zur Laufzeit festgelegt werden; dennoch bilden Java-Arrays gegenüber dem Typ **List** aus dem JCF eine schwächere Realisierung des Listenkonzeptes:
 - Ein Array, einmal erzeugt, hat eine **feste Maximalkapazität**.
 - Auf einem Array gibt es außer dem indizierten Zugriff **keine höherwertigen Operationen** wie beispielsweise (zwischen zwei Elementen) **Einfügen**, **Entfernen** (mit Aufrücken), am Ende **Anfügen** oder den **Test auf Enthaltensein**.



SE1 – Level 4

30

Vergleich (dynamische) Sammlung und Array

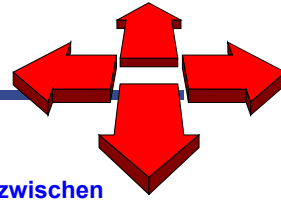
- Die Java-Sammlungen wie **List** oder **Set** sind beschränkt auf **Sammlungen von Referenzen**, d.h. es können nur Referenztypen als Elementtypen definiert werden; Arrays erlauben hingegen beide Typfamilien als Elementtyp.
- Nach seiner Erzeugung kann ein Array seine **Größe nicht mehr verändern**; **List** und **Set** hingegen sind dynamische Sammlungen und können **beliebig viele Elemente** aufnehmen.
- Ein Array modelliert **zusammenhängende Speicherzellen**; der schreibende Zugriff auf eine Position verschiebt nicht alle nachfolgenden Elemente, sondern ersetzt das Element an der Position. Ein Einfügen muss durch ein Verschieben „von Hand“ realisiert werden.
- Arrays werden in der Implementierung von dynamischen Sammlungen verwendet; sie stehen auf einem **niedrigeren Abstraktionsniveau**.

Zusammenfassung



- Ein **Array** ist eine elementare imperative Datenstruktur, die sehr speichernah konzipiert ist.
- Arrays sind geordnete Reihungen gleichartiger Elemente, auf die über einen Index zugegriffen wird.
- Arrays stehen auf einem **niedrigeren Abstraktionsniveau** als Listen und Mengen.
- Für Java gilt:
 - Die Elemente können von elementarem Typ oder Referenztypen sein (auch Referenzen auf andere Arrays).
 - Die Größe eines Arrays wird erst beim Erzeugen festgelegt.
 - Jeder Zugriff über den Index wird zur Laufzeit geprüft.

Klassen und Objekte – revisited



- Im klassischen objektorientierten Modell ist der **Unterschied zwischen Klasse und Objekt** klar:
 - **Klassen** sind die Einheiten des **statischen Programmtextes**. Sie beschreiben die Erzeugung und das Verhalten von Objekten.
 - **Objekte** sind die Einheiten des **laufenden Programms**. Sie haben einen veränderbaren Zustand und ein prinzipiell festgelegtes Verhalten. Alle Operationen beziehen sich immer auf ein Objekt.
- Wenn **Klassen** selbst vollständig **im Laufzeitsystem verfügbar** sind, verschiebt sich diese klare Unterteilung:
 - Über den Zugriff auf eine Klasse kann zur Laufzeit das Verhalten ihrer Objekte verändert werden.
 - Klassen werden zu eigenständigen Objekten mit einem eigenen Zustandsraum.
- Java wählt einen „Mittelweg“ zwischen diesen beiden Positionen.

SE1 – Level 4

33

Klassen in Java sind auch selbst Objekte



- Klassen in Java definieren nicht nur das Verhalten und die Struktur ihrer Exemplare zur Laufzeit; sie existieren auch selbst als Objekte zur Laufzeit.
- Ein solches **Klassenobjekt** kann wie alle Objekte einen Zustand haben (über **Klassenvariablen**) und Methoden anbieten (**Klassenmethoden**).
- Klassenvariablen und Klassenmethoden werden mit dem Modifikator **static** deklariert.

```
class Konto
{
    private static int exemplarzaehler = 0;

    public Konto()
    {
        exemplarzaehler++;
    }

    public static int anzahlErzeugteExemplare()
    {
        return exemplarzaehler;
    }
}
```

Klassenvariable

Auch hier gilt: Klassenvariablen sollten privat deklariert werden.

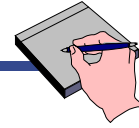
Klassenmethode



SE1 – Level 4

34

Klassenoperationen



- Die öffentlichen Klassenmethoden bilden die Operationen eines Klassenobjektes.
- Die Operationen eines Klassenobjektes sind für Klienten in der Punktnotation aufrufbar:

`<Klassenname>.<Klassenoperation> (<aktuelle Parameter>);`



```
class Kontenverwalter
{
    public void statusPruefen()
    {
        int anzahlKonten = Konto.anzahlErzeugteExemplare();
        ...
    }
}
```

Klassenoperationen als Dienstleistungen

- Die statischen Methoden in Java werden häufig zur Realisierung von (einfachen) **Dienstleistungen** benutzt, die sich nicht auf den Zustand des gerufenen Klassenobjekts beziehen, sondern **ausschließlich auf den übergebenen Parametern** arbeiten.
- Vorteil: Zum Abrufen dieser Dienstleistungen muss kein Exemplar einer Klasse erzeugt werden; das Klassenobjekt steht unmittelbar zur Verfügung.
- Beispiele:
 - Die Klasse **Arrays** aus dem Paket **java.util**, die ausschließlich statische Methoden anbietet, mit denen Arrays manipuliert werden können. Alle Arrays werden dabei als Parameter an die Methoden übergeben.
 - Die Klassenoperation **arraycopy** in der Klasse **java.lang.System**. Sie bietet eine dritte Möglichkeit zum Kopieren von Array-Inhalten (neben dem expliziten Traversieren und **clone**).
 - Mathematische Funktionen in **java.lang.Math**.



Die spezielle Klassenoperation main



- Eine Klasse kann eine Klassenmethode mit einer ganz speziellen Signatur anbieten:

```
public static void main(String[] args)
```



- Die auf **genau diese Weise** definierte Klassenoperation wird in der Laufzeitumgebung von Java gesondert behandelt: Sie bildet die Schnittstelle zum Betriebssystem, indem nur genau sie von außerhalb der Java-Welt aufgerufen werden kann.
- Sie bildet somit den **Einstiegspunkt für Java-Programme**: In dieser Methode werden üblicherweise die ersten Exemplare erzeugt, mit denen eine Java-Anwendung gestartet wird.
- Die Möglichkeit, beliebige Objekte interaktiv erzeugen und manipulieren zu können, wird ausschließlich von BlueJ geboten. Andere IDEs bieten einen **Startknopf**, mit dem eine main-Methode aufgerufen wird.

Initialisierung von Klassenobjekten in Java

- Jede Klasse in Java definiert nur genau ein Klassenobjekt.
- Dieses Klassenobjekt wird nicht über einen Konstruktoraufwurf erzeugt, sondern wird automatisch erzeugt, sobald eine Klasse in die Virtual Machine geladen wird (weil eine Klassenmethode aufgerufen wird oder weil ein Exemplar der Klasse erzeugt wird).
- Es gibt deshalb auch keine aufrufbaren Konstruktoren für Klassenobjekte. In einer Klassendefinition können aber **Klassen-Initialisierer** angegeben werden, die nach dem Laden der Klasse ausgeführt werden.



```
class Konto
{
    static {
        exemplarzaehler = 42;
        ...
    }
}
```

Level 4: Hinter den Kulissen von Sammlungen

Klassenkonstanten



- Auch Konstanten (gekennzeichnet durch den Modifikator `final`) können mit dem Modifikator `static` deklariert werden.
- Sie werden dadurch zu **Klassenkonstanten**.
- Klassenkonstanten werden öffentlich (`public`) deklariert, wenn sie als globale Konstanten dienen sollen.
- Beispiele:

```
public static final int TAGE_PRO_WOCHE = 7;  
public static final float PI = 3.141592654f;  
public static final int ANZAHL_SPALTEN = 80;
```



Hinweis zur Pragmatik: Fast immer sollten im Quelltext solche **symbolischen Konstanten** verwendet werden, anstatt an allen benutzenden Stellen jeweils das gewünschte Literal direkt anzugeben.

SE1 – Level 4

39

Klassenoperationen als Dienstleistungen

- Die statischen Methoden in Java werden häufig zur Realisierung von (einfachen) **Dienstleistungen** benutzt, die sich nicht auf den Zustand des gerufenen Klassenobjekts beziehen, sondern **ausschließlich auf den übergebenen Parametern** arbeiten.
- Vorteil: Zum Abrufen dieser Dienstleistungen muss kein Exemplar einer Klasse erzeugt werden; das Klassenobjekt steht unmittelbar zur Verfügung.
- Beispiele:
 - Die Klasse `Arrays` aus dem Paket `java.util`, die ausschließlich statische Methoden anbietet, mit denen Arrays manipuliert werden können. Alle Arrays werden dabei als Parameter an die Methoden übergeben.
 - Die Klassenoperation `arraycopy` in der Klasse `java.lang.System`. Sie bietet eine dritte Möglichkeit zum Kopieren von Array-Inhalten (neben dem expliziten Traversieren und `clone`).
 - Mathematische Funktionen in `java.lang.Math`.



SE1 – Level 4

40

Level 4: Hinter den Kulissen von Sammlungen

Nicht alle Objekte sind Exemplare



- Nach der Einführung von Klassenobjekten können wir eine Unterscheidung zwischen **Exemplar** und **Objekt** für Java vornehmen:
 - Alle Exemplare einer Klasse sind Objekte.
 - Auch eine Klasse ist ein Objekt, sie ist aber in Java nicht das Exemplar einer weiteren Klasse (es gibt keine so genannten „Metaklassen“).
 - Exemplare werden explizit mit **new** erzeugt, während Klassen automatisch geladen und initialisiert werden, sobald sie benutzt werden.



SE1 – Level 4

41

Die Rolle der Klasse in anderen Sprachen

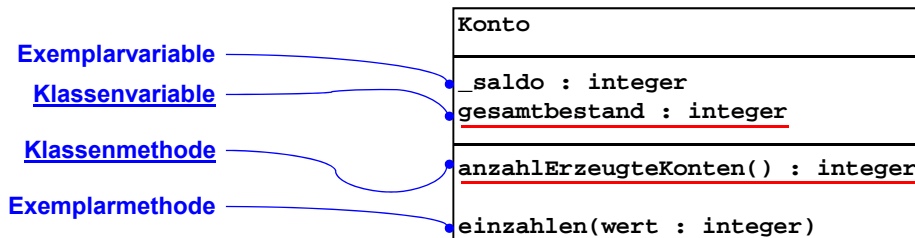


- Eine Klasse wird generell verstanden als eine Schablone, ein Bauplan für Objekte.
- **Smalltalk** geht weiter als Java: In Smalltalk ist eine Klasse selbst wieder ein Exemplar einer Metaklasse. Es ist in Smalltalk sogar möglich, die Methoden einer Klasse „im laufenden Betrieb“ zu verändern.
- In **Eiffel** ist eine Klasse hingegen ein Konzept der Organisation von Programmtexten und zu deren Übersetzung. Zur Laufzeit gibt es kein Klassenobjekt.
- Es gibt auch objektorientierte Programmiersprachen ohne eingebautes Klassenkonzept (**Self**, **Cecil**). In diesen so genannten **exemplarbasierten Sprachen** werden Einzelobjekte definiert, weitere Objekte können durch **Klonen** bereits bestehender Objekte erzeugt werden.

SE1 – Level 4

42

Klassenmethoden und -variablen in der UML-Notation



Klassenvariablen und Klassenmethoden werden in den Klassen-Diagrammen der UML **unterstrichen**, um sie von Exemplarvariablen und -methoden zu unterscheiden.

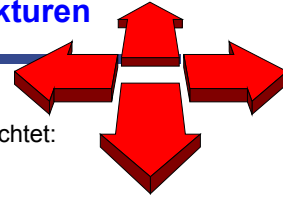
Zusammenfassung



- **Klassen** sind in verschiedenen Sprachen sehr unterschiedlich modelliert.
- In Java sind **Klassen auch Objekte** mit einem eigenen Zustand, der zur Laufzeit verändert werden kann; die dazu notwendigen **Klassenvariablen** werden mit dem Schlüsselwort **static** deklariert.
- Die Operationen eines Klassenobjektes werden mit **Klassenmethoden** realisiert, die ebenfalls mit dem Schlüsselwort **static** deklariert werden. Für ihren Aufruf muss kein Exemplar erzeugt werden.
- Öffentliche **Klassenkonstanten** (**public static final**) werden verwendet, um global gültige Konstanten zu definieren.

Level 4: Hinter den Kulissen von Sammlungen

Von Sammlungen zu dynamischen Datenstrukturen



- Wir haben bisher **Sammlungen** wie Mengen und Listen betrachtet:
 - Elemente können hinzugefügt und entnommen werden.
 - Es gibt unterschiedliche Organisationsprinzipien.
- Wir betrachten nun die **Implementationen** dieser Sammlungen und thematisieren damit erstmals **dynamische Datenstrukturen**, die in der Informatik eine große Tradition haben.
- Wir klären im Weiteren die Begriffe
 - „dynamisch“
 - „Datenstruktur“
- Wir geben einen Einstieg in die wichtigsten dynamischen Datenstrukturen mit ihren spezifischen Stärken und Schwächen.



SE1 – Level 4

45

Implementationen für Sammlungen

- Das Java Collections Framework (JCF) stellt für seine Collection-Interfaces (wie **List**, **Set**, etc.) einige Implementationen zur Verfügung.
- Alle Implementationen basieren auf zwei grundlegenden Programmierkonstrukten:
 - **Arrays**
 - **verkettete Strukturen**
- Einige Implementationen machen nur von dem einen oder dem anderen Konzept Gebrauch, andere kombinieren sie. Insgesamt können im JCF vier Implementierungskonzepte unterschieden werden:
 - **verkettete Listen**
 - **wachsende Arrays**
 - **balancierte Bäume**
 - **Hash-Verfahren**

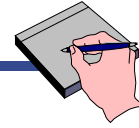
Dies sind Beispiele für
dynamische Datenstrukturen!

SE1 – Level 4

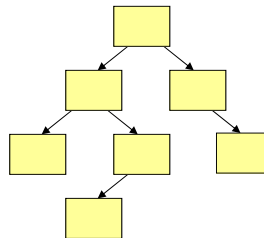
46

Level 4: Hinter den Kulissen von Sammlungen

Dynamische Datenstrukturen



- **Dynamische Datenstrukturen** bezeichnen die Organisationsform von veränderbaren Sammlungen von Objekten.
- Eine **Struktur** ist ein
 - Gebilde aus Elementen (Objekten)
 - mit Beziehungen (Relationen)
- Dynamische Datenstrukturen sind meist **gleichartig rekursiv** aufgebaut.
- **Ändern einer Struktur** bedeutet
 - Hinzufügen, Modifizieren und Löschen von Elementen
 - Ändern von Beziehungen
- Als **dynamisch** werden Datenstrukturen dann bezeichnet, wenn sie durch das Einfügen und Entfernen ihrer Elemente wachsen und schrumpfen.



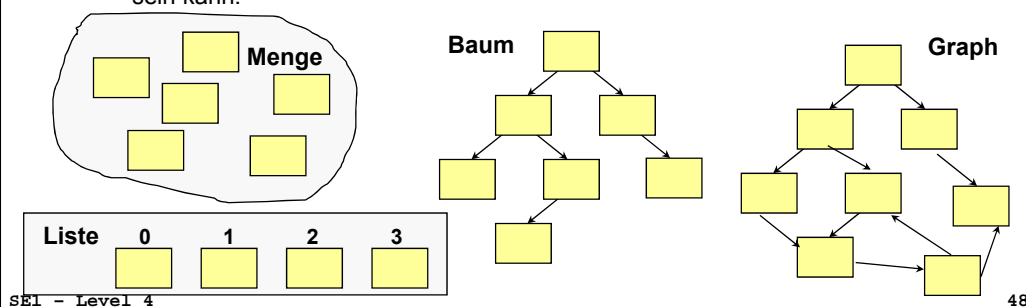
SE1 – Level 4

nach © Neumann

47

Einteilung dynamischer Datenstrukturen

- Üblicherweise werden dynamische Datenstrukturen nach den Eigenschaften ihrer grundlegenden Struktur eingeteilt.
- Wir unterscheiden:
 - Strukturen von Elementen **ohne Relation** zueinander (z.B. für Mengen)
 - Lineare oder **sequenzielle Strukturen** (z.B. für Listen)
 - **Bäume**, in denen ein Element ein Vorgängerelement aber mehr als ein Nachfolgerelement haben kann.
 - **Graphen**, in denen ein Element beliebig mit anderen Elementen verbunden sein kann.



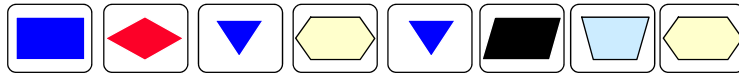
SE1 – Level 4

48

Level 4: Hinter den Kulissen von Sammlungen

Sammlungen implementieren I: Listen

- Listen sind die grundlegendste (elementare) sequenzielle Struktur.
- Sobald wir dynamische Datenstrukturen zu ihrer Implementierung beherrschen, können wir auch weitere lineare Sammlungsarten wie Stacks und Queues implementieren.
- Wir werden zwei Implementationsformen betrachten:
 - eine basierend auf Verkettung
 - eine basierend auf Arrays



SE1 – Level 4

49

Zur Erinnerung: der Umgang mit einer Liste

- Aus Sicht des Klienten einer Liste sind relevant:
 - Eine Liste kann **beliebig viele Elemente** enthalten.
 - Über den Index kann direkt auf **beliebige Positionen** in der Liste zugegriffen werden.
 - Das **Einfügen** eines Elements **erhöht** den Index der nachfolgenden Elemente.
 - Das **Entfernen** eines Elements **verringert** den Index der nachfolgenden Elemente.
 - Häufig wird die Information benötigt, ob ein gegebenes Element bereits in der Liste enthalten ist („**Test auf Enthaltensein**“).

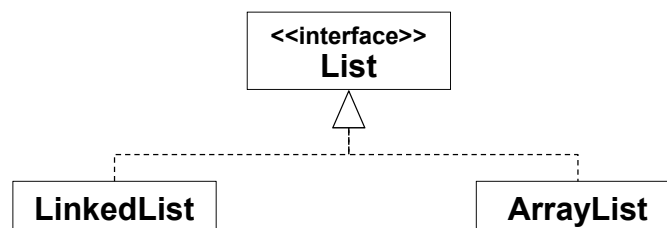


SE1 – Level 4

50

Listen-Implementationen im Java Collections Framework

- Der Umgang mit einer Liste ist im JCF mit dem Interface `List` modelliert.
- Das JCF bietet zwei Implementierungen für dieses Interface:
 - `LinkedList`
 - `ArrayList`
- `LinkedList` basiert auf dem Konzept **verkettete Liste**.
- `ArrayList` basiert auf dem Konzept **wachsender Arrays**.

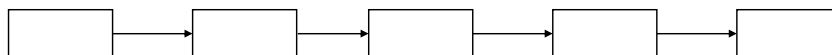


SE1 – Level 4

51

Lineare Datenstrukturen für Listen: Verkettung

- Eine **verkettete Liste** ist das bekannteste Beispiel für eine sequenzielle dynamische Datenstruktur:
- Wir haben die Liste als Sammlung kennen gelernt, deren Wertemenge Elemente als **endliche Folgen eines Grundtyps** umfasst:
 - Listenelemente besitzen eine **Reihenfolge**
 - Elemente des Grundtyps können **mehrfach enthalten** sein (Duplikate)
- Eine verkettete Liste ist grundsätzlich als Struktur betrachtet eine **Sequenz** ihrer Elemente:
 - Jedes Listenelement ist **mit dem nächsten verbunden**.
 - Um vom Anfang zum Ende der Liste zu gelangen, muss **jedes Element traversiert** werden.



SE1 – Level 4

52

Einfach verkettete Liste

- Eine Liste als Referenzkette zwischen ihren Elementen kann auf zwei Arten realisiert werden. Wir unterscheiden:
 - **Einfach verkettete Liste:**
 - jedes Listenelement hat nur eine Referenz auf sein **Nachfolgerelement**.
 - Die Liste kann nur elementweise vom Anfang zum Ende traversiert werden.

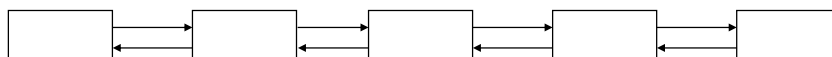


SE1 – Level 4

53

Doppelt verkettete Liste

- **Doppelt verkettete Liste:**
 - jedes Listenelement hat eine Referenz auf sein **Nachfolger**- und sein **Vorgängerelement**.
 - Die Liste kann elementweise in beide Richtungen traversiert werden.



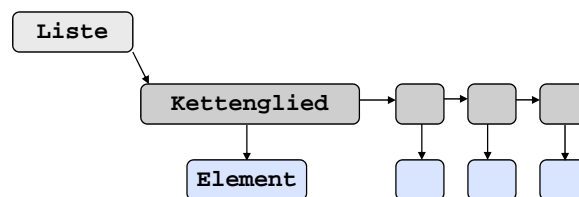
SE1 – Level 4

54

Level 4: Hinter den Kulissen von Sammlungen

Schema des objektorientierten Entwurfs einer Liste

- Üblicherweise besteht der objektorientierte Entwurf einer Liste aus verschiedenen Objekten:
 - Ein Objekt, das die **Liste insgesamt** für ihre Klienten repräsentiert.
 - Objekte als **Kettenglieder**, die die Verkettung der Liste realisieren. Sie sind für die Klienten nicht sichtbar.
 - Objekte, die als **Elemente** in der Liste gespeichert sind. Sie werden von den Klienten über die Umgangsformen der Liste verwaltet.



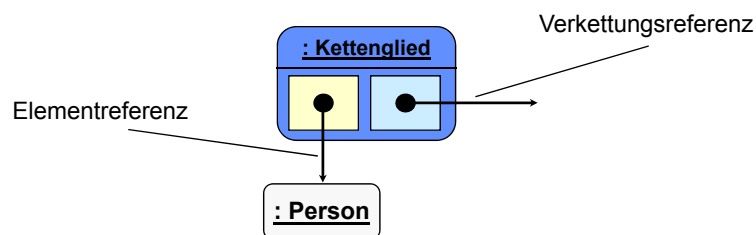
SE1 – Level 4

55

Konstruktion eines Kettenglieds

Ein Kettenglied kann objektorientiert so entworfen werden:

- Jedes **Kettenglied** wird als ein eigenes Objekt modelliert. Dazu wird eine eigene Klasse für die Kettenglieder definiert, etwa **Kettenglied**.
- Jedes Kettenglied hält eine Referenz auf das eigentliche **Element der Sammlung** (beispielsweise in einer Liste von Personen eine Referenz auf ein Exemplar der Klasse **Person**).
- Ein Kettenglied hält außerdem mindestens eine Referenz auf ein weiteres Kettenglied (das Nachfolgerelement) als **Verkettungsreferenz**.



SE1 – Level 4

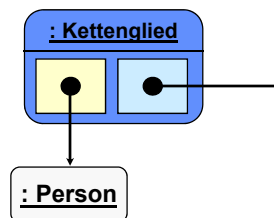
56

Level 4: Hinter den Kulissen von Sammlungen

Definition einer Klasse für Kettenglieder

Hier wird eine Exemplarvariable deklariert, die den gleichen Typ hat wie die definierende Klasse!
Dies wird auch **strukturelle Rekursion** genannt.

```
class Kettenglied {
    private Kettenglied _nachfolger;
    private Person _element;
    ...
}
```

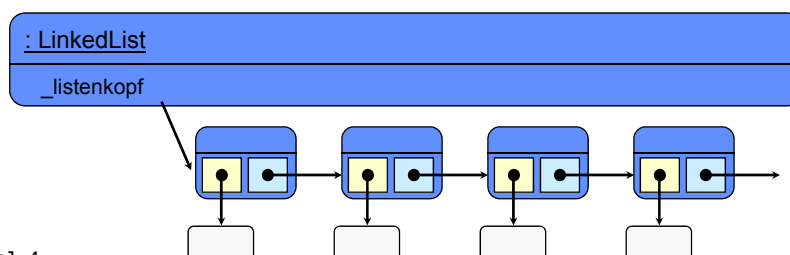


SE1 - Level 4

57

Konstruktion einer einfach verketteten Liste

- Die Liste wird für ihre Klienten als Exemplar einer eigenen Klasse realisiert, etwa **LinkedList**.
- Die Kettenglieder (Exemplare von **Kettenglied**) bilden die innere Struktur der Liste.
- Die referenzierten Elemente werden üblicherweise **nicht** als Teil der Liste angesehen (beispielsweise wird beim Entfernen aus einer Liste nicht automatisch das Element selbst gelöscht).
- In der Klasse **Liste** wird üblicherweise die Referenz auf das erste **Listenelement** gehalten („**Listenkopf**“).



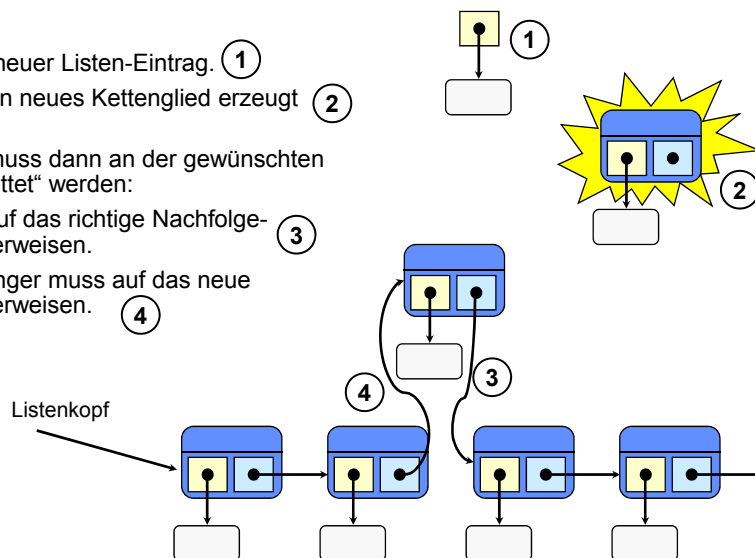
SE1 - Level 4

58

Level 4: Hinter den Kulissen von Sammlungen

Einfügen in einer verketteten Liste

- Gegeben: ein neuer Listen-Eintrag. ①
- Zuerst muss ein neues Kettenglied erzeugt werden. ②
- Dieses Glied muss dann an der gewünschten Stelle „eingekettet“ werden:
 - Es muss auf das richtige Nachfolge-Element verweisen. ③
 - Der Vorgänger muss auf das neue Element verweisen. ④

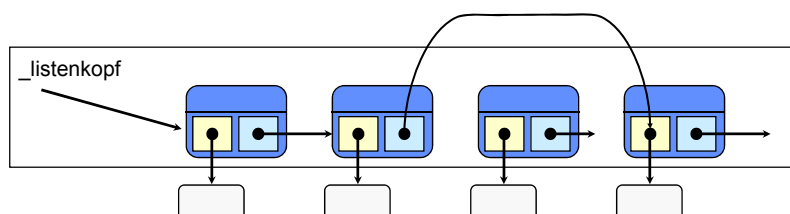


SE1 – Level 4

59

Entfernen aus einer verketteten Liste

- Die Referenz des Vorgängers wird einfach auf den Nachfolger umgebogen.
- Das Kettenglied wird dann vom Garbage-Collector entfernt, sobald keine Referenz mehr darauf existiert.



SE1 – Level 4

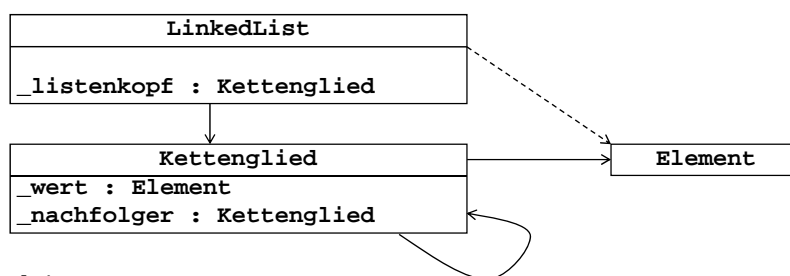
60

Level 4: Hinter den Kulissen von Sammlungen

Klassendiagramm einer einfach verketteten Liste

Das Klassendiagramm zeigt wesentliche objektorientierte Konstruktionsmerkmale einer einfach verketteten Liste:

- Die Klasse **LinkedList** hält einen Verweis auf die Klasse **Kettenglied** im Attribut **_listenkopf**.
- Sie benutzt die Klasse **Element** in den Parametern ihrer Methoden.
- Die Klasse **Kettenglied** speichert jeweils ein Exemplar der Klasse **Element** im Attribut **_wert**.
- **Kettenglied** verweist auf sich selbst, um im Attribut **_nachfolger** das nächste Kettenglied referenzieren zu können.

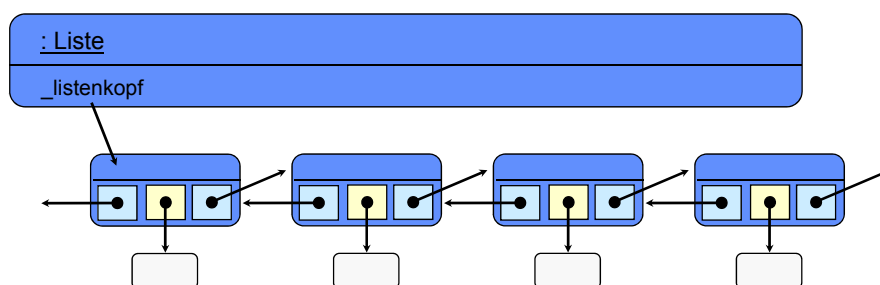


SE1 - Level 4

61

Doppelt verkettete Liste

- Bei einer doppelt verketteten Liste hat ein Kettenglied außerdem eine Referenz auf ein weiteres Kettenglied (das "vorige"). Dies ermöglicht ein effizientes Durchlaufen der Liste in beide Richtungen.
- Einfügen und Entfernen werden vereinfacht.
- Die JCF-Implementation **LinkedList** basiert auf diesem Konzept.



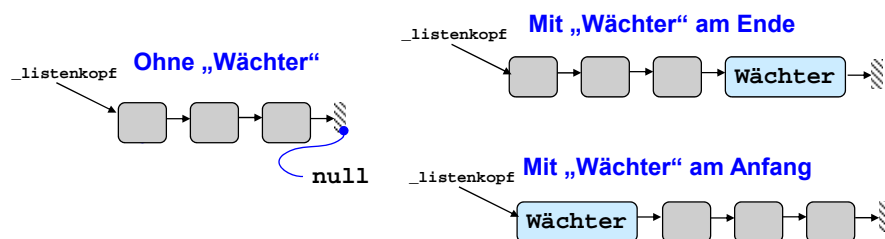
SE1 - Level 4

62

Level 4: Hinter den Kulissen von Sammlungen

Designalternative Listenenden

- An den **Listenenden** (Anfang und/oder Ende) können explizit leere Kettenelemente stehen, sog. **Wächter** (engl. sentinel).
- Vorteil eines Wächter-Objekts:
 - Es gibt weniger Sonderfälle zu programmieren (Beim Einfügen, Entfernen etc.).



SE1 – Level 4

© Budd

63

Designalternative Verweise

- Neben einem Verweis auf den Anfang kann auch ein Verweis auf das **Ende** einer Liste gehalten werden.
- Vorteil:
 - Der Zugriff auf das Listende ist genau so schnell wie auf den Listenanfang (gut für Operationen wie **Anfügen**).



SE1 – Level 4

© Budd

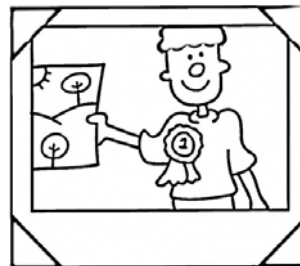
65

Level 4: Hinter den Kulissen von Sammlungen

Zitat: Good Programmers / Bad Programmiers

- “Good programmers plan before they write code, especially when there are pointers involved. For example, if you ask them to reverse a linked list, good candidates will always make a little drawing on the side and draw all the pointers and where they go. They have to. **It is humanly impossible to write code to reverse a linked list without drawing little boxes with arrows between them.** Bad programmers will start writing code right away.” (Blog: Joel on Software, March 2000)

<http://www.joelonsoftware.com/articles/fog0000000073.html>



SE1 – Level 4

66

Lineare Strukturen für Listen: „Wachsende“ Arrays

- Arrays sind als direkte Implementation für Listen ungeeignet, weil sie eine festgelegte Größe haben.
- Stattdessen wird das Konzept von **wachsenden Arrays** benutzt:
 - Erzeuge ein Array mit einer Anfangsgröße;
 - Befülle es mit den einzufügenden Elementen;
 - Wenn dieses Array voll ist, erzeuge ein größeres und kopiere alle Elemente des alten Arrays in das neue.
- Dies führt zur Unterscheidung von logischer Größe der Liste (Anzahl der Elemente, **Kardinalität**) und physikalischer Größe (**Kapazität**) des implementierenden Arrays.
- Es muss immer gelten: **Kapazität ≥ Kardinalität**

zulässige Indexe
(immer: < Kardinalität)

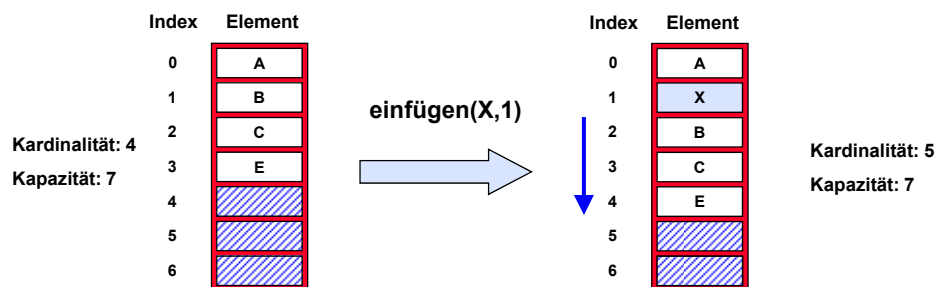
Index	Element
0	A
1	B
2	C
3	E
4	
5	
6	

SE1 – Level 4

67

Wachsende Arrays: Eigenschaften

- Der Zugriff auf eine beliebige Indexposition ist sehr einfach und **schnell**: Es wird ein **indexbasierter Zugriff** auf das Array ausgeführt.
- Bei jedem Einfügen oder Löschen müssen die nachfolgenden Elemente innerhalb des Arrays **verschoben** werden. Im Extremfall (Operation am Listenanfang) müssen alle Elemente um eine Position verschoben werden.

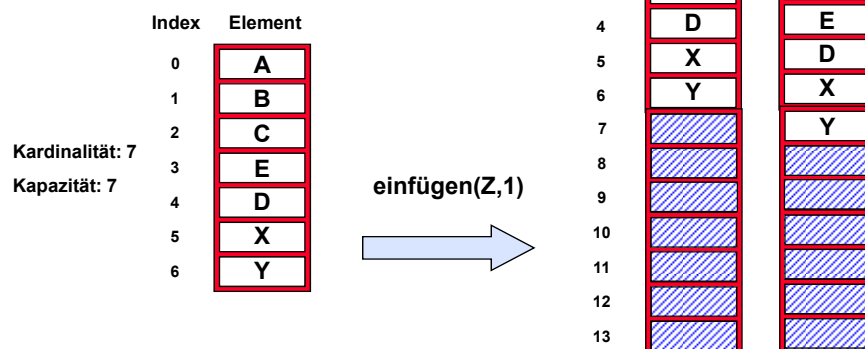


SE1 – Level 4

68

Wachsende Arrays: Eigenschaften

- Wenn die Kapazität des Arrays für ein neu einzufügendes Element nicht ausreicht, muss ein größeres Array (beispielsweise mit doppelter Kapazität) angelegt werden, in das anschließend alle Elemente umkopiert werden.
- Die Klasse **ArrayList** des JCF basiert auf diesem Konzept.



SE1 – Level 4

69

Vergleich der Implementationen

- Beide Implementationen für Listen haben ihre Stärken und Schwächen.
- Um dies zu verdeutlichen, vergleichen wir die beiden Implementationen des JCF:
 - **LinkedList** (verkettete Liste)
 - **ArrayList** (wachsende Arrays)
- Wir betrachten den Aufwand für zwei typische Operationen:
 - **Einfügen** eines Elementes
 - **Zugriff** auf ein Element an einer bestimmten Position



Einfügen in eine Liste

- **LinkedList:**
 - Nachteil: Position, an der eingefügt werden soll, ist erst durch **Traversieren** der Liste zu erreichen. Im Durchschnitt wird dabei die halbe Liste abgelaufen.
 - Nachteil: **Objekterzeugung** für jede Einfügung.
 - Vorteil: Das **Einfügen** ist sehr einfach (einfaches Umketten, konstanter Aufwand).
- **ArrayList:**
 - Nachteil: Alle Elemente nach der Einfügeposition müssen um eine Position verschoben werden. Im Durchschnitt wird dabei die **halbe Liste umkopiert**.
 - Nachteil: Wenn die Kapazität ausgeschöpft ist, muss ein **neues Array** angelegt und alle Elemente müssen umkopiert werden.
 - Vorteil: die Position zum Einfügen kann **direkt angesprochen** werden (konstanter Aufwand).

Zugriff auf eine beliebige Position; Fazit

- Beim Zugriff auf eine beliebige Position spielt die `ArrayList` ihre Stärke voll aus:
 - Der Zugriff erfolgt in konstanter Zeit, während bei der `LinkedList` durchschnittlich die halbe Liste durchlaufen werden muss.
- Insgesamt zeigt sich, dass die Implementierungen für unterschiedliche Benutzungsprofile einer Liste unterschiedlich geeignet sind:
 - Für **relativ konstante** Listen, bei denen **häufig** wahlfrei auf beliebige Positionen **zugegriffen** wird, ist die `ArrayList` besser geeignet.
 - Für sehr **dynamische große** Listen, bei denen **viel eingefügt** und entfernt wird (insbesondere am Listenanfang), ist die `LinkedList` eventuell die bessere Wahl.
- **Pragmatik für Java:** Für die meisten Anwendungen mit eher kleinen Listen ist die `ArrayList` die Implementation der Wahl.

SE1 – Level 4

72

List-Implementierungen im Vergleich

LinkedList

- Knoten, die mit einander verbunden werden und eine **Kette** bilden
- Indizierung durch „Abzählen“
- Einfügen legt ein neues Objekt an, das eingekettet wird.
- Löschen kettet lediglich ein Kettenglied aus der Kette aus.

ArrayList

- Dynamisch „wachsendes“ **Array**
- Direkt indizierbar
- Einfügen erfordert Verschieben von Folgeelementen und eventuelle Neuerzeugung eines kompletten Arrays plus Umkopieren.
- Löschen erfordert Verschieben von Folgeelementen. Es findet keine Verkleinerung des Arrays statt!

SE1 – Level 4

73

Aufwand für Operationen formalisiert

- Unter **Aufwand** verstehen wir die Menge an **elementaren Schritten**, die für eine zusammengesetzte Operation ausgeführt werden müssen.
 - Bsp.: Die zusammengesetzte Operation **Einfügen an Position i** auf einer verketteten Liste erfordert mehrere elementare Schritte, die sich aus dem Durchlaufen der Liste bis zur Position i, dem Erzeugen eines neuen Kettenglieds und dem Setzen der Verkettungen ergeben.
- Elementare Schritte sind:
 - **Zuweisungen** ($x = y, i++, \dots$)
 - **Vergleiche** ($a \leq b, \text{next} \neq \text{null}, \dots$)
 - **Aufrufe mit konstantem Zeitbedarf** (Objekterzeugung kleiner Objekte, sondierende Methoden, ...)

Konstanter und variabler Anteil des Aufwandes

- Der Aufwand für eine Operation setzt sich üblicherweise aus einem **konstanten Anteil** und einem **variablen Anteil** zusammen.
 - Der konstante Anteil ist für jede Ausführung der Operation gleich.
 - Der variable Anteil hängt von der Menge der zu verarbeitenden Daten ab.

Bsp.: Beim Einfügen in eine verkettete Liste ist der Aufwand für Erzeugen und Verketteten immer gleich, das Durchlaufen hingegen ist abhängig von dem gewünschten Index. Im schlechtesten Fall muss die gesamte Liste durchlaufen werden.

Level 4: Hinter den Kulissen von Sammlungen

Abschätzungen des Aufwandes

- Bei Abschätzungen des Aufwandes wird häufig vom **schlechtesten Fall** (engl.: worst case) ausgegangen. Ebenfalls verbreitet ist die Abschätzung des **Aufwandes im Mittel**.
- Außerdem wird üblicherweise von **großen Datenmengen** ausgegangen, so dass der konstante Anteil irrelevant wird und vernachlässigt werden kann.
- Der Aufwand wird dadurch zu einer **Funktion**, die von der Anzahl **N** der zu verarbeitenden Datenelemente abhängt:
 - **Aufwand = $f(N)$**
- Im Zusammenhang von Aufwandsbetrachtungen für **Algorithmen** wird auch von ihrer **Komplexität** gesprochen. Die zugehörige **Komplexitätstheorie** ist ein Teilgebiet der theoretischen Informatik.

SE1 – Level 4

76

Ein erster Blick auf die „O-Notation“



- Wir betrachten hier die sog. „**O-Notation**“ oder auch *Landau-Notation*, nach dem deutschen Zahlentheoretiker Edmund Landau. Sie wird in der Mathematik und in der Informatik verwendet, um das asymptotische Verhalten von Funktionen und Folgen zu beschreiben.
- In der Informatik finden wir die O-Notation insbesondere in der **Komplexitätstheorie**, um verschiedene Probleme und Algorithmen danach zu vergleichen, wie "schwierig" oder aufwändig sie zu berechnen sind. Mit der O-Notation können Aufwände für algorithmische Probleme in so genannte **Komplexitätsklassen** eingeteilt werden.
- Typische Komplexitätsklassen sind:

$O(1)$	konstanter Aufwand (u.a. alle elementaren Schritte)
$O(\log n)$	logarithmischer Aufwand (u.a. Baumsuche)
$O(n)$	linearer Aufwand (u.a. Suche in Listen)
$O(n \cdot \log n)$	(u.a. gute Sortierverfahren)
$O(n^2)$	quadratischer Aufwand (u.a. einfache Sortierverfahren)
$O(n^k)$ mit $k \geq 0$	polynomialer Aufwand
$O(2^n)$	exponentieller Aufwand
- Diese Klassen (außer der ersten) geben eine **Größenordnung** für den Aufwand **in Abhängigkeit von der zu verarbeitenden Datenmenge n**.
- Diese Notation wird ausführlich in FGI 1 sowie in Algorithmen und Datenstrukturen thematisiert.

SE1 – Level 4

77

Level 4: Hinter den Kulissen von Sammlungen

Komplexitäten der Listenoperationen

- Einfügen in eine Liste:
 - **LinkedList: $O(n)$** (linearer Aufwand, da bis zu n Elemente durchlaufen werden müssen)
 - **ArrayList: $O(n)$** (linear Aufwand, da bis zu n Elemente verschoben werden müssen)
- Zugriff auf ein Element über einen Index:
 - **LinkedList: $O(n)$** (linearer Aufwand, siehe oben)
 - **ArrayList: $O(1)$** (konstanter Aufwand, da direkte Abbildung auf indizierten Zugriff der unterliegenden Rechnerarchitektur)

SE1 – Level 4

78

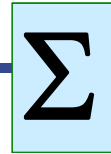
Test auf Enthaltensein

- Aufwand für Suchen in einer Liste: **$O(n)$**
 - Die vorige Diskussion hat gezeigt, dass sowohl einfache verkettete Strukturen als auch Array-Implementationen keine günstigen Voraussetzungen für diesen Test haben: Bei beiden muss im Durchschnitt die halbe Liste durchsucht werden.
- Insbesondere bei Sets (Mengen) spielt der Test auf Enthaltensein in der Praxis häufig eine wichtige Rolle.
- Deshalb werden wir für Mengen Implementationen betrachten, die bei diesem Test erheblich effizienter sind.
- Möglich sind Realisierungen mit **$O(\log n)$** und sogar mit **$O(1)$** , also konstantem Aufwand!

SE1 – Level 4

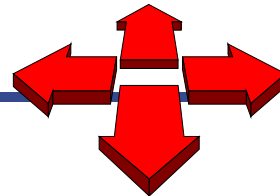
79

Zusammenfassung



- Für Listen gibt es zwei klassische imperative Implementationen: **verkettete Listen** und **wachsende Arrays**.
- Verkettete Listen können **einfach** oder **doppelt verkettet** sein.
- Wachsende Arrays basieren auf Arrays, die bei Bedarf mit größerer Kapazität angelegt werden.
- Beide Implementationen haben Stärken und Schwächen. Mit Hilfe der **O-Notation** können wir diese Unterschiede formal greifbar machen.

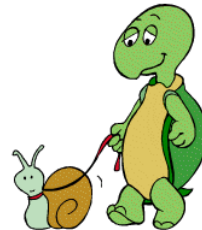
Sammlungen implementieren II: Mengen



- Aufwand für Operationen auf Mengen?
- Bäume als Realisierung für Mengen:
 - binäre Bäume
 - Suchbäume
 - balancierte Bäume
- Hash-Verfahren als Realisierung für Mengen:
 - Hash-Tabelle
 - Hash-Funktion

Einfügen in Mengen: hoher Aufwand?

- Wenn ein Element in eine Menge eingefügt werden soll, muss die Implementierung der Menge prüfen, ob das Element ein Duplikat ist.
- Es ist also bei jedem Einfügen ein Test auf Enthaltensein nötig.
- Würde die Menge intern als einfache Liste implementiert, würde der Aufwand für das Einfügen linear von der Größe der Menge abhängen ($O(n)$).
- Bei großen Mengen ist das nicht akzeptabel.
 - Wir brauchen etwas Schnelleres.



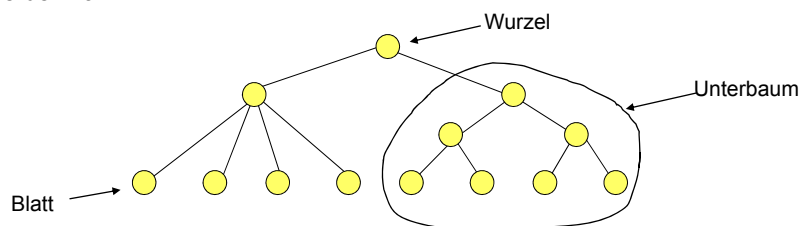
Effiziente Suchverfahren: Anforderungen

- Der **Test auf Enthaltensein** wird häufig auch als **Suche** bezeichnet: Wir suchen ein Element in einer Sammlung; wenn wir es finden, ist das Testergebnis positiv.
- Um nicht jedes Element in der Menge mit dem zu suchenden Element vergleichen zu müssen (linearer Aufwand, $O(n)$), müssen wir die Elemente geeignet strukturieren.
- Voraussetzung dafür ist, dass die Elemente bestimmte Eigenschaften haben. Zwei typische Anforderungen an Elemente sind, dass sie
 - **sortierbar** oder
 - **kategorisierbar** sind.
- Sortierbare Elemente ermöglichen eine **binäre Suche** oder eine Realisierung mit einem **Suchbaum**.
- Kategorisierbare Elemente ermöglichen **Hash-Verfahren**.

Level 4: Hinter den Kulissen von Sammlungen

Bäume

- Ein **Baum** (engl.: tree) ist eine Struktur, in der Knoten miteinander durch (gerichtete) Kanten verbunden sind.
- Jeder Knoten kann beliebig viele **Kindknoten** (Nachfolger) haben, mit denen er über Kanten verbunden ist. Ein Knoten hat aber immer **maximal einen** Vorgängerknoten.
- Ein Knoten ohne Vorgänger heißt **Wurzel** des Baumes.
- Ein Knoten ohne Kindknoten wird als **Blatt** bezeichnet.
- Bäume sind **rekursiv**: Ein Kindknoten kann wieder als ein Wurzelknoten eines kleineren Baumes angesehen werden, der als **Unterbaum** bezeichnet werden kann.

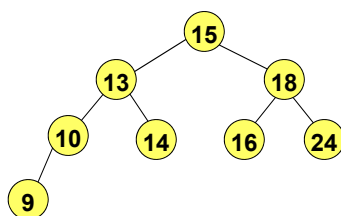


SE1 – Level 4

84

Binäre Suchbäume

- In einem **binären Baum** hat ein Knoten maximal zwei Kindknoten.
- Wenn jeder Knoten ein sortierbares Element enthält und gilt, dass im linken Unterbaum alle „kleineren“ und im rechten Unterbaum alle „größeren“ Element liegen, dann wird der Baum zu einem **binären Suchbaum**.



Anordnung der Menge
 $M = \{ 9 \ 10 \ 13 \ 14 \ 15 \ 16 \ 18 \ 24 \}$
 als binärer Suchbaum

SE1 – Level 4

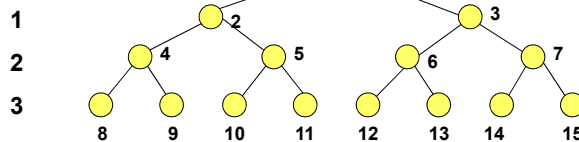
85

Level 4: Hinter den Kulissen von Sammlungen

Merkmale von binären Bäumen



- Viele Eigenschaften von (binären) Bäumen beziehen sich auf die Anzahl der Knoten und Blätter sowie die Höhe eines Baums.
- Beispiele für Eigenschaften:
 - Ein voller binärer Baum mit Höhe h hat 2^h Blätter.
 - Die Zahl der Knoten eines vollen binären Baums mit Höhe h ist $2^{h+1} - 1$

Höhe $h=0$  2^0 Knoten $2^0 + 2^1$ $2^0 + 2^1 + 2^2$ $2^{h+1} - 1$

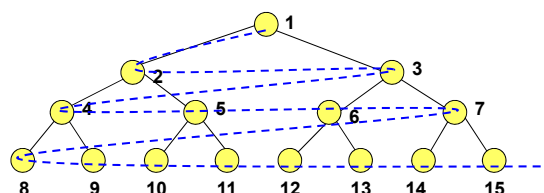
SE1 – Level 4

nach © Neumann

86

Traversieren von Bäumen

- Für viele Anwendungen müssen alle Knoten eines Baumes der Reihe nach bearbeitet werden. Dazu muss ein Ordnungsprinzip gewählt werden, um die Knoten eines Baumes zu „linearisieren“.
- Die bekanntesten Traversierungsstrategien sind:
 - Breitendurchlauf
 - Tiefendurchlauf



?

SE1 – Level 4

nach © Neumann

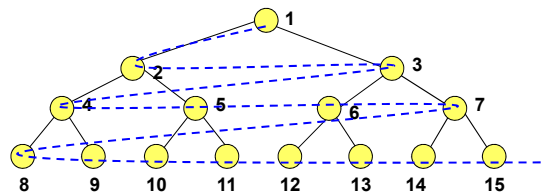
87

Breitendurchlauf



Breitendurchlauf (level-order tree traversal)

- Die Idee:
Verfahren, um einen Baum "schichtenweise" abzuarbeiten,
z.B. bei der Suche nach Zielknoten mit minimalem Abstand zur Wurzel.
- Gilt auch für allgemeine Bäume.
- Strategie:
 - Breite vor Tiefe,
 - links vor rechts



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

SE1 – Level 4

nach © Neumann

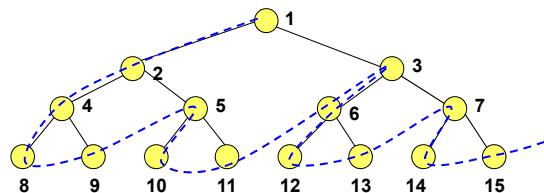
88

Tiefendurchlauf



Tiefendurchlauf

- Die Idee:
Allgemeines Verfahren, um einen Baum "astweise" in Richtung seiner Blätter abzuarbeiten,
z.B. bei der Suche nach Planschritten, die zu einem Ziel führen sollen.
- Gilt auch für allgemeine Bäume.
- Strategie:
 - Tiefe vor Breite,
 - links vor rechts



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

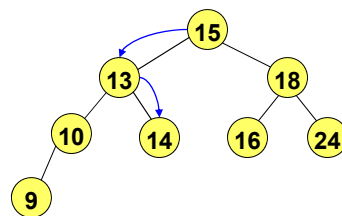
SE1 – Level 4

nach © Neumann

89

Suchalgorithmus für binäre Suchbäume

- Die Suche nach einem Element ist (rekursiv) folgendermaßen möglich:
 - Ist der Baum leer?
 - Ja → **Suche erfolglos**
 - Nein: Enthält der Wurzelknoten das gesuchte Element?
 - Ja → **Suche erfolgreich**
 - Nein: Ist das gesuchte Element kleiner als das aktuelle Element?
 - Ja: **Weitersuchen** im linken Teilbaum
 - Nein: **Weitersuchen** im rechten Teilbaum



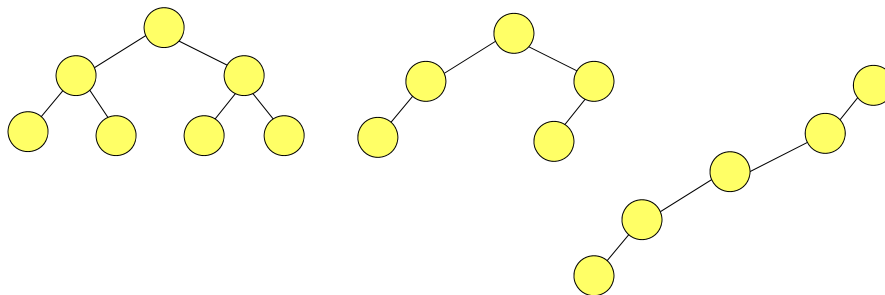
Suche nach $k = 14$

SE1 – Level 4

90

Bäume können entarten

- Damit sind wir fast am Ziel: Bei einem „gutmütigen“ Baum ist die Suche recht schnell.
- Ein binärer Baum kann jedoch „entarten“: Eine verkettete Liste kann gesehen werden als ein verkümmerter binärer Baum mit jeweils einem Kindknoten.
- Es fehlt noch eine entscheidende Bedingung...



SE1 – Level 4

91

Level 4: Hinter den Kulissen von Sammlungen

Balancierte binäre Suchbäume



- In einem **balancierten binären Baum** gilt für jeden Knoten, dass die **Höhen** seiner beiden Unterbäume sich **maximal um eins** unterscheiden.
- Die Höhe **h** eines solchen Baumes berechnet sich dann logarithmisch aus der Anzahl **n** der Knoten im Baum:
 - **$h = \lg(n)$** (\lg ist hier der Logarithmus Dualis, also der Logarithmus zur Basis 2)
- Unsere **Suche** muss von der Wurzel bis zu jedem Blatt dann höchstens **$\lg(n)$ Vergleiche** vornehmen; der Aufwand ist damit in seiner Größenordnung **$O(\lg(n))$** .
- Die Baum-Implementationen des JCF (**TreeSet** und **TreeMap**) benutzen binäre, balancierte Bäume. Sie setzen jedoch voraus, dass die **Elemente sortierbar** sind (sie müssen das Interface **Comparable** implementieren).
- Es geht aber noch schneller; und das sogar ohne die Notwendigkeit, dass die Elemente sortierbar sind!

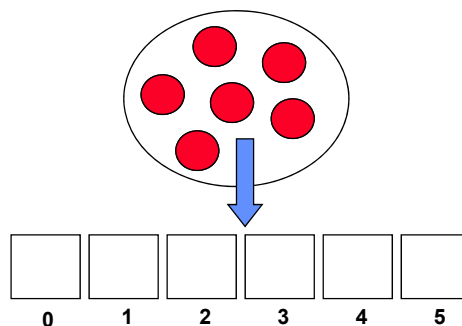
SE1 – Level 4

92

Hash-Verfahren: Die Grundidee



- Die Grundidee von **Hash-Verfahren** (auch: Hashing) ist, Elemente auf eine Indexstruktur abzubilden. Dabei soll sich aus einem Element **unmittelbar** sein Index **berechnen** lassen.



nach © Budd

SE1 – Level 4

93

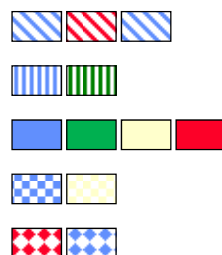
Level 4: Hinter den Kulissen von Sammlungen

Lösungsansatz für Hashing: mehrere Listen statt einer

- Statt eine Liste komplett zu durchsuchen:



- Mehrere Listen + Wissen, in welcher gesucht werden muss!



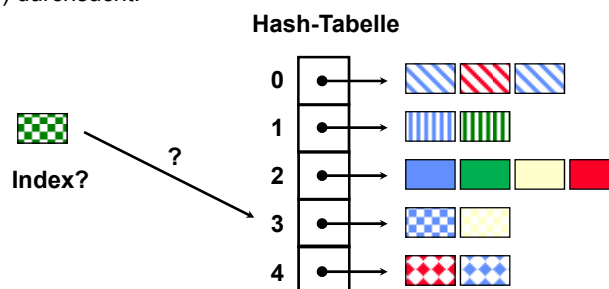
SE1 – Level 4

94

Im Kern: die Hash-Tabelle



- Im Kern eines Hash-Verfahrens steht die sogenannte **Hash-Tabelle** (meist als **Array** realisiert). Sie kann verstanden werden als eine Tabelle von (möglichst kurzen) Listen.
- Für ein zu suchendes Element wird zuerst der Index der Liste in der Tabelle ermittelt, in der Elemente der gleichen Kategorie liegen.
- Nach einem (schnellen) indexbasierten Zugriff auf die Liste wird diese dann (linear) durchsucht.



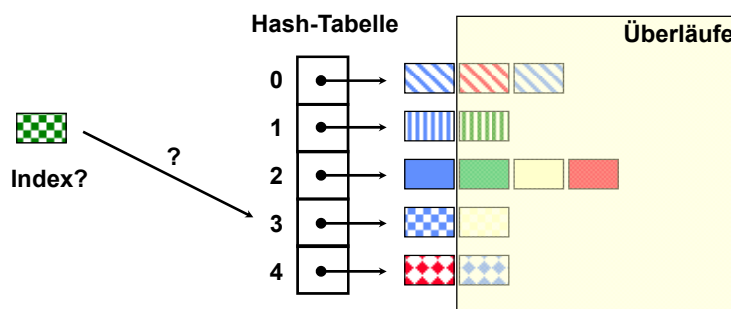
SE1 – Level 4

95

Level 4: Hinter den Kulissen von Sammlungen

Ziel: möglichst wenig Überläufe

- Ideal für ein Hash-Verfahren ist, wenn die Listen maximal ein Element enthalten; nach der Indexberechnung ist dann für eine Suche **maximal ein Vergleich** notwendig!
- Enthält eine Liste mehr als ein Element, so werden die überschüssigen Elemente als **Überläufe** bezeichnet.
 - In einigen Darstellungen werden die Listen deshalb auch als Überlaufbehälter (engl.: bucket) bezeichnet.

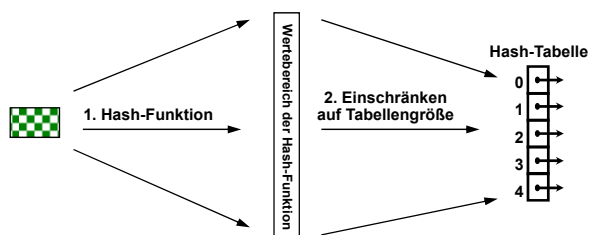


SE1 – Level 4

96

Entscheidend: Die Hash-Funktion

- Die Basis des Verfahrens ist die **Hash-Funktion**:
 - Sie bildet ein **Element** auf einen **ganzzahligen Wert** ab, also auf einen Integer-Wert.
 - Der berechnete Wert bildet eine **künstliche Kategorie**: Alle Elemente mit demselben Wert fallen in dieselbe Kategorie.
 - Wenn die Hash-Funktion zwei verschiedene Elemente auf denselben Wert abbildet, wird dies als **Kollision** bezeichnet.
 - Der berechnete Wert muss in einem zweiten Schritt auf einen **Index** in der Hash-Tabelle abgebildet werden.
- Entscheidend ist die **Güte** der Hash-Funktion: Je gleichmäßiger sie die Elemente in der Tabelle verteilt, desto schneller ist das Verfahren.



SE1 – Level 4

97

Beispiel: Hash-Funktion mit Kollision

Beispiel:

Einfügen von Monatsnamen in eine Hash-Tabelle mit 12 Positionen.

$c_1 \dots c_k$ Monatsname als Zeichenkette

$N(c_k)$ Binärdarstellung eines Zeichens

Hashfunktion h mit $m=12$ bildet Zeichenketten in Indizes $0 \dots 11$ ab:

$$h(c_1 \dots c_k) = \sum_{i=1}^k N(c_i) \bmod m$$

Verteilung der Namen mit Kollisionen:

0	November	6	Mai, September
1	April, Dezember	7	Juni
2	März	8	Januar
3	-	9	Juli
4	August	10	-
5	Oktober	11	Februar

Eine **ideale Hash-Funktion** bildet alle Schlüssel eins-zu-eins auf unterschiedliche Integerwerte ab, die fortlaufend sind und bei Null beginnen.

nach © Neumann

SE1 – Level 4

98

Hash-Verfahren im Java Collections Framework

- Die Implementation **HashSet** für das Interface **Set** im JCF basiert auf einem Hash-Verfahren.
- Als Basis für die Hash-Funktion wird das Ergebnis der Operation **hashCode** verwendet, die in der Klasse **Object** und damit für alle Objekte definiert ist.
- Vorsicht:** Wenn für eine Klasse die Operation **equals** redefiniert wird, dann muss garantiert sein, dass für zwei Exemplare dieser Klasse, die nach der neuen Definition gleich sind, auch die Operation **hashCode** den gleichen Wert liefert!
 - Es besteht sonst die Möglichkeit, dass in ein **HashSet** Duplikate eingetragen werden können, weil die beiden Elemente in verschiedenen Überlaufbehältern landen.
 - Die Spezifikation von **Set** würde damit nicht eingehalten, weil das Implementationsverfahren zufällig auf Hashing basiert!



SE1 – Level 4

99

Indexberechnung für die Hash-Tabelle



- **Aufgepasst:** Der Wertebereich einer Hash-Funktion erlaubt üblicherweise auch negative Werte; in der Hash-Tabelle (einem Array) kann aber nur mit positiven Werten indiziert werden!
- Beim Abbilden des Hash-Wertes auf einen gültigen Index muss deshalb ein eventuell vorhandenes **negatives Vorzeichen entfernt** werden. Dazu gibt es mehrere Möglichkeiten:
 - Maskieren des Vorzeichens über eine Bitmaske
 - Rechtsschieben der Bits des Hash-Wertes (Eliminieren des LSB)
 - In Java: Anwenden von **Math.abs()**
 - Sonderfall beachten: Im Zweierkomplement ist der Betrag der kleinsten negativen Zahl immer um Eins größer als die größte positive Zahl. Math.abs liefert bei der kleinsten negativen Zahl deshalb das Argument als Ergebnis, also wieder eine negative Zahl!
 - Der String „hochenwiseler“ beispielsweise liefert in Java **Integer.MIN_VALUE** als Hash-Wert.



SE1 – Level 4

100

Hash-Verfahren im Java Collections Framework (II)

- Dynamische Größenanpassung
 - Die Hash-Verfahren im JCF passen die Größe der Hash-Tabelle dynamisch an (ähnlich wie bei wachsenden Arrays).
 - Auf diesen Prozess kann mit zwei Konstruktorparametern Einfluss genommen werden:
 - der **Anfangskapazität** (initial capacity) als Größe der Tabelle
 - dem **Befüllungsgrad** (load factor)
 - Die Hash-Tabelle wird mit der Anfangskapazität angelegt. Sobald mehr Elemente eingefügt sind als die aktuelle Kapazität multipliziert mit dem Befüllungsgrad, wird die Kapazität erhöht (Aufruf der privaten Methode **rehash**).

SE1 – Level 4

101

Set-Implementierungsvarianten im JCF

TreeSet

- **Balancierter Binärer Suchbaum**
- Einfügen und Entfernen durch Baumsuche in $O(\log n)$.
- Reorganisation bei Ungleichgewichten durch Knotenumordnung ist akzeptabel schnell, kommt aber oft vor.

HashSet

- **Hash-Verfahren** mit dynamischer Anpassung der Hash-Tabelle
- Einfügen und Entfernen in **konstanter Zeit**.
- Reorganisation nach Erreichen des Load-Factors ist durch komplettes Rehashen teuer!
- Geschwindigkeit hängt auch von der Güte der Hash-Werte ab.

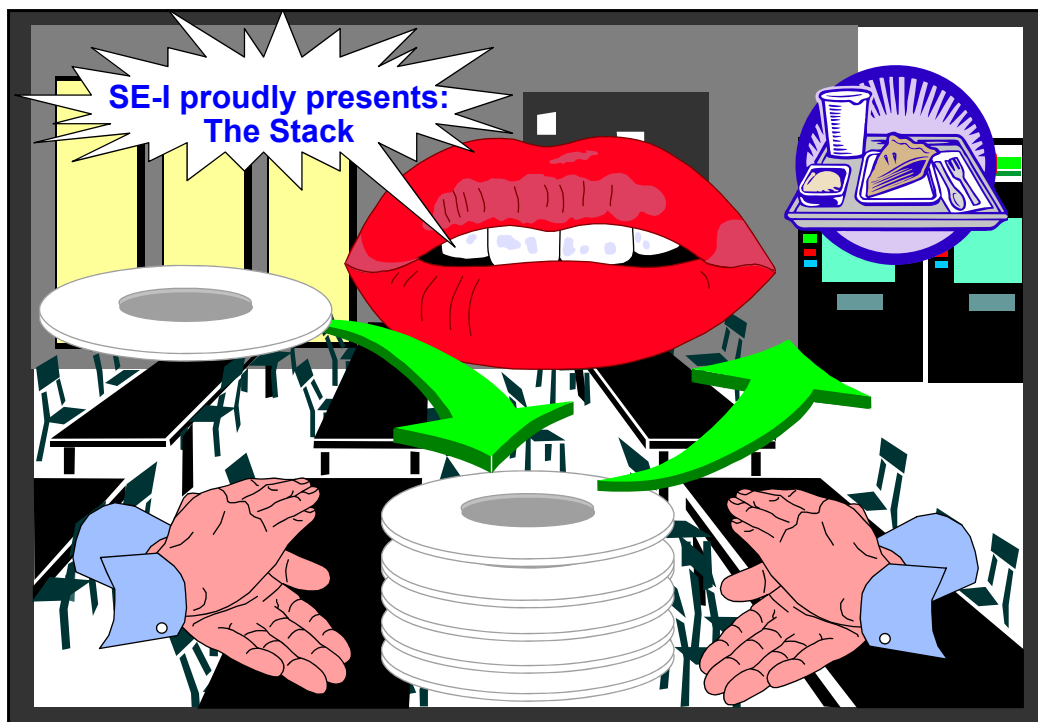
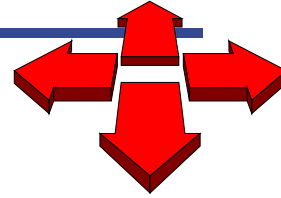
Zusammenfassung



- Bei **Mengen** ist der Test auf Enthaltensein (**Suche**) typischerweise die wichtigste Operation.
- Mit einer naiven Implementation ist der Such-Aufwand $O(n)$.
- Bei einem **balancierten binären Suchbaum** reduziert sich der Such-Aufwand auf $O(\log(n))$.
- Die **set**-Implementation **TreeSet** des JCF erfordert, dass die Elemente das Interface **Comparable** implementieren.
- Mit einem guten **Hash-Verfahren** kann eine Suche in $O(1)$ durchgeführt werden.
- Die **set**-Implementation **HashSet** des JCF implementiert ein dynamisches Hash-Verfahren.

Mehr zu Sammlungen: Stacks, Queues, Sortieren

- Neben Listen und Mengen gibt es weitere, in Theorie und Praxis wichtige Sammlungstypen, z.B.:
 - **Stack** (oder Keller)
 - **Queue** (oder Schlange)
- Stack und Queue haben die gleiche sequenzielle Struktur wie eine Liste, unterscheiden sich aber durch ihre Umgangsformen und durch die Organisationsprinzipien ihrer Elemente.
- **Sortieren** ist eine Standardaufgabe für Sammlungen. Wir geben einen Einstieg.



DAS Standardbeispiel eines dynamischen Datentyps: der Stack

Hier die klassische Definition:

- **Keller** (engl. *stack*): Folge von Elementen eines gegebenen Datentyps mit eingeschränkten Einfüge- und Ausfügeoperationen.
- Eine **Datenstruktur** über einem Datentyp *T* bezeichnet man als Keller, Stack oder Stapel, ... wenn es zwei Zugriffsoperationen gibt, von denen die eine ein Element von *T* stets *an das Ende der Folge* einfügt und die andere stets *das letzte Element der Folge entfernt* und als Ergebnis liefert. Die Einfügeoperation nennt man **push**, die Ausfügeoperation **pop**.
- Das Prinzip, dass stets das zuletzt eingefügte Element eines Kellers als erstes wieder entfernt werden muss, bezeichnet man als **LIFO-Prinzip** (vom engl. **Last In First Out**).

[nach Informatik-Duden]



Stacks spielen bei der syntaktischen Analyse und beim Übersetzerbau eine große Rolle (wir erinnern uns an den Aufruf-Stack). Sie gehören zu den wichtigen „Behältern“ der Informatik.

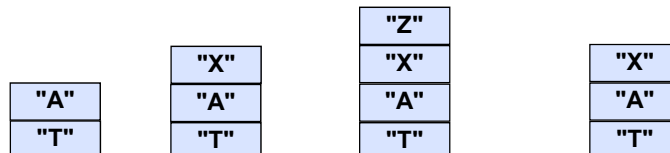
SE1 – Level 4

106

Ein Stack-Interface

```
interface Stack<E>
{
    void push(E element);
    E pop();
    boolean isEmpty();
}
```

gegeben: `Stack<String> s`
mit zwei Elementen.



```
s.push("X");    s.push("Z");    String t = s.pop();
// t == "Z"
```

Das Interface **Stack** spezifiziert zwar die Signaturen der Stack-Operationen, legt aber nicht die gewünschte Organisationsform der Elemente (LIFO) fest.



SE1 – Level 4

107

Level 4: Hinter den Kulissen von Sammlungen

Die Klasse Stack im Java Collections Framework

Constructor Summary

Stack()
Creates an empty Stack.

Method Summary

boolean	empty() Tests if this stack is empty.
E	peek() Looks at the object at the top of this stack without removing it from the stack.
E	pop() Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item) Pushes an item onto the top of this stack.
int	search(Object o) Returns the 1-based position where an object is on this stack.



SE1 – Level 4

108

Implementationsskizze: Stack implementiert mit Liste

```
class StackImpl<E> implements Stack<E>
{
    private List<E> implList = new LinkedList<E>();
    public void push(E el)
    {
        implList.add(0,el); // vorn einfügen
    }

    /** pop is valid only if not isEmpty */
    public E pop()
    {
        if (!isEmpty())
        {
            return implList.remove(0); // vorn entfernen
        }
        else
            // Fehlerbehandlung
    }
    public boolean isEmpty()
    {
        return implList.isEmpty();
    }
}
```

Dies ist eine generische
Implementation auf Basis
der generischen Liste
aus dem JCF.

SE1 – Level 4

109

Level 4: Hinter den Kulissen von Sammlungen

Die Schlange (engl.: Queue)



Schlange (engl.: queue): Folge von Elementen eines gegebenen Datentyps, bei der ein Element stets **an das Ende der Folge** eingefügt und stets **das erste Element der Folge entfernt** und als Ergebnis geliefert wird.

- Das Prinzip, dass stets das zuerst eingefügte Element einer Schlange als erstes wieder entfernt werden muss, bezeichnet man als **FIFO-Prinzip** (vom engl. **First In First Out**).
- Es gibt neben den klassischen FIFO-Schlangen auch „gewichtete Warteschlangen“ (*priority queues*), bei denen die Elemente nach Priorität (dem Gewicht) sortiert werden.



Schlangen spielen bei vielen technischen Anwendungen und für Systemsoftware eine große Rolle. So wird z.B. die Tastatureingabe zeichenweise in eine Schlange (input buffer) geschrieben und ausgelesen.

SE1 – Level 4

110

Ein Queue-Interface

```
interface Queue<E>
{
    void enqueue(E element);
    E dequeue();
    boolean isEmpty();
}
```

"A"	"T"
-----	-----

gegeben: `Queue<String> q`
mit zwei Elementen.

"A"	"T"	"X"
-----	-----	-----

```
q.enqueue("X");
```

"A"	"T"	"X"	"Z"
-----	-----	-----	-----

```
q.enqueue("Z");
```

"T"	"X"	"Z"
-----	-----	-----

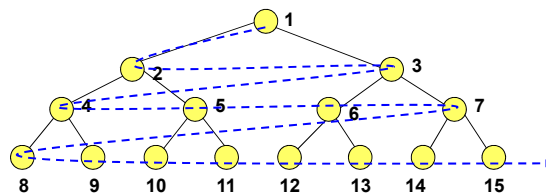
```
String s = q.dequeue(); // s == "A"
```

SE1 – Level 4

111

Anwendung einer Queue: Breitendurchlauf durch Bäume

- Gegeben: Ein **binärer Baum**.
- Gewünscht: Ein **Breitendurchlauf** durch die Knoten.

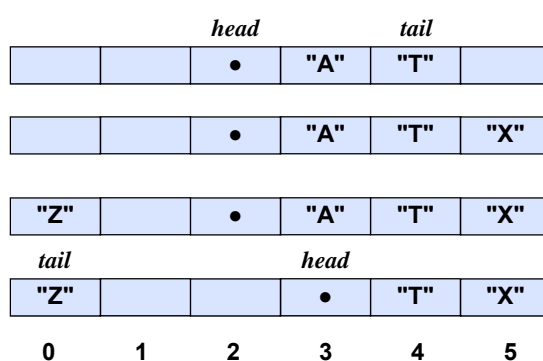


- Lösungsskizze
 - Initialisierung: füge das Wurzelement in eine Queue ein.
 - Schleife – solange die Queue noch Elemente enthält:
 - Entnimm das erste Element (verarbeitete es) und füge seine Nachfolger (falls vorhanden) in die Queue ein.

SE1 – Level 4

112

Implementationsskizze für eine Queue als zyklisches Array



Intern:

```
String[] _implArray =
    new String[6];
```

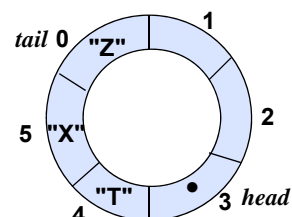
```
q.enqueue("X");
```

```
q.enqueue("Z");
```

```
String s = q.dequeue();
```



Dieser Entwurf als sogen. **Ringpuffer** ist speichereffizient für begrenzte Schlangen.



SE1 – Level 4

© Universität München, Hans-Peter Kriegel und Thomas Seidl

113

Level 4: Hinter den Kulissen von Sammlungen

Ein Interface beschreibt nur Signaturen

```
interface Stack<E>
{
    void push(E element);
    E pop();
    boolean isEmpty();
}
```

```
interface Queue<E>
{
    void enqueue(E element);
    E dequeue();
    boolean isEmpty();
}
```

```
interface Geblubber<E>
{
    void bla(E element);
    E bloe();
    boolean blub();
}
```



Interfaces sind nur **semi-formale Spezifikationen**. Die Signaturen sind formal festgelegt, aber es fehlt die **Semantik** der Implementationen.

Wir können die Semantik lediglich in Form von **Kommentaren** angeben (also nur informell).

Dies ändert jedoch nichts am Wert von sprechenden Namen: Quelltexte (insbesondere von Klienten) werden so lesbarer!

SE1 – Level 4

114

Ausblick: Abstrakte Datentypen



- Die theoretische Informatik gibt mit den sog. **abstrakten Datentypen** eine **formale Spezifikation** eines Datentyps vor.
- Die Grundidee ist, einen Datentyp durch sein **Verhalten** und nicht durch seine **Implementation** zu definieren. Wir lernen Beispiele in SE II kennen.
- Für SE I gelten aber bereits die zentralen Konstruktionsprinzipien:
 - **Trenne** die **Spezifikation** einer Konstruktionseinheit von ihrer **Implementation** (Repräsentation).
 - **Verberge Implementationen** von Konstruktionseinheiten und mache sie so austauschbar.

•TYPES

- *STACK [G]*

•FUNCTIONS

- *push: STACK [G] → G → STACK [G]*
- *pop: STACK [G] → STACK [G]*
- *top: STACK [G] → G*
- *empty: STACK [G] → BOOLEAN*
- *new: STACK [G]*

•AXIOMS

For any $x, s: \text{STACK } [G]$

- A1 • *top (push (s, x)) = x*
- A2 • *pop (push (s, x)) = s*
- A3 • *empty (new)*
- A4 • *not empty (push (s, x))*

•PRECONDITIONS

- *pop (s: STACK [G]) require not empty (s)*
- *top (s: STACK [G]) require not empty (s)*

SE1 – Level 4

115

Level 4: Hinter den Kulissen von Sammlungen

Sortieren

- **Sortieren** ist eine klassische Informatikaufgabe für Sammlungen: Eine Menge von Objekten soll in eine **geordnete Reihenfolge** gebracht werden.
- Voraussetzung zum Sortieren ist daher, dass es eine **Ordnungsrelation** $<$ (genau: eine strenge schwache Ordnung) für ein Merkmal der Objekte gibt. Dieses Merkmal heißt auch **Sortierschlüssel**; die darauf bezogene Ordnungsrelation heißt **Sortierkriterium**.
- Die Algorithmen, nach denen Sammlungen sortiert werden, heißen **Sortierverfahren**.
- Im Alltag finden wir sortierte Sammlungen beispielsweise als
 - Telefonverzeichnisse (sortiert nach lexikographischer Ordnung der Nachnamen)
 - Fahrpläne (sortiert nach der Abfahrtszeit)

$$L = [a_1, a_2, \dots, a_n]$$

aufsteigende Anordnung:	$\bigwedge_{i=1}^{n-1} a_{i+1} \geq a_i$
absteigende Anordnung:	$\bigwedge_{i=1}^{n-1} a_{i+1} \leq a_i$

SE1 – Level 4

116

Sortierverfahren



- **Sortierverfahren** lassen sich nach unterschiedlichen Kriterien klassifizieren. Üblich sind:
 - **Komplexität:**
Wieviele Schritte (Operationen) sind notwendig, um n Elemente zu ordnen (oft unterschieden nach im besten / durchschnittlichen / schlechtesten Fall, abhängig von der Vorsortierung der Elemente).
 - **Speicherbedarf:**
Sortiert das Verfahren die Elemente ohne zusätzlichen Speicherbedarf (**in situ** bzw. **in place**) oder nicht.
 - **Stabilität:**
Ein Sortierverfahren ist **stabil**, wenn es die ursprüngliche Reihenfolge zweier gleicher Elemente beim Sortieren nicht verändert, sonst ist es instabil.
 - **Art des Algorithmus:**
Die meisten Sortierverfahren sind **vergleichsbasiert**, d.h. die Elemente werden paarweise verglichen. Einige Verfahren (z.B. Bucketsort, Radixsort) berechnen aus den Sortierschlüsseln direkt die Reihenfolge der Elemente.

SE1 – Level 4

117

Komplexität von Sortierverfahren

Komplexität

- Gesucht wird eine obere Schranke für die Anzahl von Schlüsselvergleichen, die im schlechtesten Fall notwendig sind, um n Objekte zu sortieren.
- Für vergleichsbasierte Sortierverfahren gilt, dass die Schranke C_{max} im besten Fall $C_{max} n = O(n \cdot \log n)$ ist.
- Allgemein ist ein Sortierverfahren von der **Ordnung** $O(g(n))$, wenn für die Zahl der Operationen $f(n)$ beim Sortieren einer Liste mit n Elementen gilt:

$$\exists c, \exists n_0 : \bigwedge_{n > n_0} f(n) \leq c \cdot g(n)$$

Typische Komplexitätseigenschaften von Sortierverfahren:

$O(n \log n)$	gute Verfahren
$O(n^2)$	einfache Verfahren

Prinzipien von vergleichsbasierten Sortierverfahren

- Sortierverfahren arbeiten nach verschiedenen Prinzipien:
 - **Vertauschen:**
Vertausche Einträge, wenn sie wechselseitig falsch angeordnet sind (Bubble-Sort, Shaker-Sort).
 - **Einfügen:**
Stelle eine sortierte Teilfolge her und füge die verbleibenden Elemente an der richtigen Stelle ein (Insertion-Sort, Shell-Sort).
 - **Auswahl:**
Suche das Element, das an Platz 1 gehört, an Platz 2 usw. (Selection-Sort, Heap-Sort).
- In allen Fällen sind die Grundoperationen **Vergleich zweier Elemente** und **Bewegen eines Elementes** erforderlich.

Level 4: Hinter den Kulissen von Sammlungen

Beispiel Bubble-Sort

Bubble-Sort: das Prinzip

- Der Algorithmus vergleicht der Reihe nach zwei benachbarte Elemente und vertauscht sie, falls sie in der falschen Reihenfolge vorliegen. Dieser Vorgang wird solange wiederholt, bis keine Vertauschungen mehr nötig sind. Hierzu sind in der Regel mehrere Durchläufe erforderlich.
- Je nachdem, ob auf- oder absteigend sortiert wird, steigen die kleineren oder größeren Elemente wie Blasen im Wasser (daher der Name) immer weiter nach oben, d.h. an das Ende der Reihe. Auch werden immer zwei Zahlen miteinander in "Bubbles" vertauscht.

55	07	78	12	42	1. Durchlauf
07	55	78	12	42	
07	55	78	12	42	
07	55	12	78	42	
07	55	12	42	78	2. Durchlauf
07	55	12	42	78	
07	12	55	42	78	
07	12	42	55	78	3. Durchlauf
07	12	42	55	78	
07	12	42	55	78	4. Durchlauf
07	12	42	55	78	Ergebnis

SE1 – Level 4

nach © wikipedia.de

120

Implementationsskizze Bubble-Sort

Für eine Liste **L** mit **n** Elementen:Schleife über **durchlauf = 1 .. n-1**

```

{
  Schleife über position = 1 .. n-1 - durchlauf
  {
    Falls L[position] > L[position+1] ...
    {
      ... vertausche L[position] und L[position+1]
    }
  }
}

```

Hinweis: Hier werden 1-basierte Indizes verwendet.

Komplexität:

- Der Algorithmus besitzt eine quadratische Komplexität ($O(n^2)$) und ist daher langsamer als viele andere Sortieralgorithmen.

Vorteile:

- Geringer Speicherbedarf (in-place-Verfahren)
- Einfach und für kleine Elementmengen gut geeignet

SE1 – Level 4

121

Level 4: Hinter den Kulissen von Sammlungen

Beispiel Quick-Sort

Quick-Sort (Hoare 1962):

- ist ein schneller, rekursiver, nicht-stabiler Sortieralgorithmus, der nach dem Prinzip **Teile und Herrsche** (engl. *divide and conquer*) arbeitet.
- Prinzip:
 - Ein (prinzipiell beliebiger) Schlüssel x wird aus den Elementen der zu sortierenden Reihe ausgewählt (**Pivot-Element**).
 - Die Reihe wird anschließend in eine Reihe mit Schlüsseln $\geq x$ und eine mit Schlüsseln $< x$ zerlegt (Divide-Schritt).
 - Die beiden resultierenden Teilreihen werden rekursiv bis auf Elementebene in gleicher Weise behandelt (Conquer).
 - Die mit einander verketteten Teilreihen bilden insgesamt eine sortierte Gesamtreihe.

SE1 – Level 4

nach © Universität München, Institut für Informatik, Hans-Peter Kriegel

122

Funktionale Definition von Quick-Sort

ALGORITHM Quicksort(S)

IF $|S| = 1$

THEN RETURN S

ELSE

Divide: Wähle irgendeinen Schlüsselwert $x = s_i.\text{key}$ aus S aus.
Berechne eine Teilfolge S1 aus S mit den Elementen,
deren Schlüsselwert $< x$ ist
und eine Teilfolge S2 mit Elementen $\geq x$.

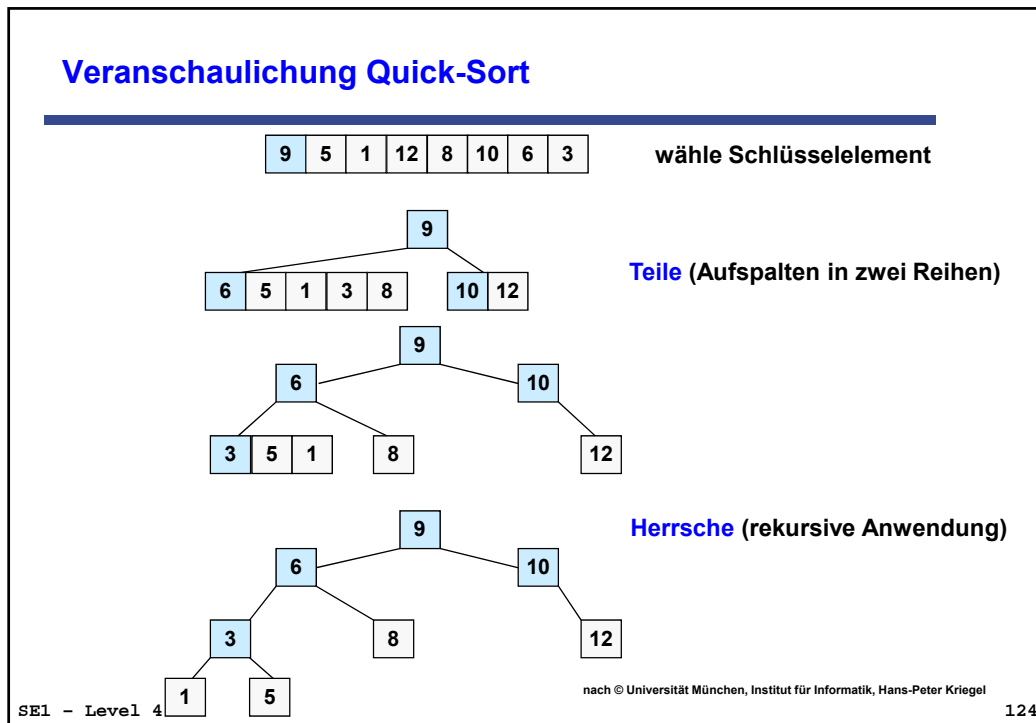
Conquer: $S1' = \text{Quicksort}(S1)$; $S2' = \text{Quicksort}(S2)$;

Merge: RETURN Concat(S1', S2')

END;

SE1 – Level 4

nach © Universität München, Institut für Informatik, Hans-Peter Kriegel 123



Imperative Konkretisierung: In-Place Quick-Sort

InPlaceQuicksort:

wähle ein Element der Folge als Pivotelement x ;

setze den Index i auf das erste und j auf das letzte Element der Folge;

wiederhole solange $i \leq j$

erhöhe Index i bis zum nächsten Element a_i mit $a_i \geq x$;

erniedrige Index j bis zum nächsten Element a_j mit $a_j \leq x$;

falls $i \leq j$

vertausche a_i und a_j ;

erhöhe i und erniedrige j um eine Position;

Wende InPlaceQuicksort auf resultierende Teilfolgen mit Länge > 1 an.

Veranschaulichung In-Place Quick-Sort

<table border="1"><tr><td>9</td><td>5</td><td>1</td><td>12</td><td>8</td><td>10</td><td>6</td><td>3</td></tr></table>	9	5	1	12	8	10	6	3	wähle Pivotelement: $x =$ <table border="1"><tr><td>9</td></tr></table>	9
9	5	1	12	8	10	6	3			
9										
i j	setze Indizes									
i j	Indizes werden nicht verschoben: $9 \geq 9$ und $3 \leq 9$									
<table border="1"><tr><td>3</td><td>5</td><td>1</td><td>12</td><td>8</td><td>10</td><td>6</td><td>9</td></tr></table>	3	5	1	12	8	10	6	9	weil $i \leq j$: vertausche Elemente...	
3	5	1	12	8	10	6	9			
$\xrightarrow{1} i$ $j \xleftarrow{1}$... und versetze i und j jeweils um eine Position									
i j	verschiebe Indizes (nur i wird verschoben)									
<table border="1"><tr><td>3</td><td>5</td><td>1</td><td>6</td><td>8</td><td>10</td><td>12</td><td>9</td></tr></table>	3	5	1	6	8	10	12	9	weil $i \leq j$: vertausche Elemente...	
3	5	1	6	8	10	12	9			
$\xrightarrow{1} i$ $j \xleftarrow{1}$... und versetze i und j jeweils um eine Position									
j i	verschiebe Indizes (beide werden verschoben)									
<table border="1"><tr><td>3</td><td>5</td><td>1</td><td>6</td><td>8</td><td>10</td><td>12</td><td>9</td></tr></table>	3	5	1	6	8	10	12	9	weil $i > j$: zwei Teilfolgen gebildet, rekursiv weiter...	
3	5	1	6	8	10	12	9			

SE1 – Level 4 126

Anmerkungen zu Quick-Sort

- Die **Wahl des Pivotelements** hat Einfluss auf die Effizienz:
 - Idealerweise sollte das Element gewählt werden, das bei einer sortierten Liste in der Mitte stünde (also der **Median**).
 - Wenn bei einer **bereits sortierten Reihe** immer das linke (oder immer das rechte) Element als Pivot gewählt wird, wird in jedem Rekursionsschritt die Reihe nur um ein Element verkürzt.
 - Der Algorithmus besitzt deshalb zwar im durchschnittlichen Fall die Komplexität $O(n \cdot \log n)$; im schlechtesten Fall degeneriert er jedoch zu $O(n^2)$.
 - Wenn die Implementierung es zulässt (schneller Indexzugriff), sollte deshalb das räumliche mittlere Element gewählt werden, um eine Degeneration auszuschließen.
- Trotz dieser Eigenschaften gilt Quick-Sort als eines der schnellsten Sortierverfahren und ist zusätzlich recht einfach zu implementieren.

Lohnt sich die Mühe überhaupt?

- Sortierverfahren können recht anspruchsvoll in ihrer Implementation sein, sowohl in der Erstellung als auch in der Wartung. Warum also nicht einfach immer mit Bubble-Sort sortieren?
- Zur Veranschaulichung vergleichen wir an einem konkreten Beispiel zwei Komplexitätsklassen für Sortierverfahren: $O(n^2)$ und $O(n \log n)$.
- Für **Klimaberechnungen** müssen teilweise sehr große Datenmengen verarbeitet werden. Wir nehmen an, dass ein Geophysiker **10 Millionen** Zahlenwerte sortieren muss, auf einem Rechner, der **10^8** elementare Schritte pro Sekunde ausführen kann.
- Er startet den Sortieralgorithmus und geht einen Kaffee trinken. Als er zurückkommt, ist der Rechner noch nicht fertig. Und wird es auch so schnell nicht...
- $O(n^2)$: $10 \text{ Millionen}^2 = 10^7 * 10^7 = 10^{14} / 10^8 = 10^6 \text{ Sekunden} \approx \mathbf{11 \text{ Tage}}$...
- $O(n \log n)$: $10^7 * \mathbf{\text{ld}(10^7)} = 10^7 * 24 / 10^8 \approx \mathbf{2,4 \text{ Sekunden}}$!
- **Ergo: Die Mühe lohnt sich für große Datenmengen sehr!**

Sortieren mit linearer Ordnung



Voraussetzungen:

- Die Menge aller Schlüssel ist hinreichend klein
- Es muss nicht ‚in situ‘ sortiert werden

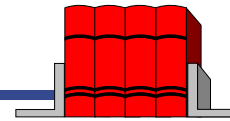
Sortieren mit **Bucket-Sort**:

- Lege für jeden Schlüssel eine Liste an.
- Mache die Listen zu Elementen eines Arrays, das mit dem Schlüssel indiziert werden kann.
- Gehe die zu sortierenden Elemente einzeln durch und hänge jedes an die zu seinem Schlüssel gehörige Liste an.
- Verbinde die Listen zu einer Ergebnisliste.

Das Verfahren ist **linear in n**, da der Aufwand für das Durchlaufen der Ausgangsliste und des Schlüssel-Arrays linear von der Zahl der Listenelemente abhängt.

Level 4: Hinter den Kulissen von Sammlungen

Parade der Sortiervverfahren



Bubble-Sort	$O(n^2)$
Shaker-Sort	$O(n^2)$
Selection-Sort	$O(n^2)$
Insertion-Sort	$O(n^2)$
Shell-Sort	$O(n^{1.2})$
Merge-Sort	$O(n \log n)$
Quick-Sort	$O(n \log n)$
Heap-Sort	$O(n \log n)$
Bucket-Sort	$O(n)$

Siehe dazu die reichhaltige Literatur, z.B.

N. Wirth: Algorithmen und Datenstrukturen, Teubner

G. Saake, K.-U. Sattler: Algorithmen und Datenstrukturen: Eine Einführung mit Java, dpunkt

R.H. Güting, S. Dieker: Datenstrukturen und Algorithmen, Teubner

Die Angaben beziehen sich auf die Durchschnittskomplexität.

Demo-Beispiele der UBC :

<http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html>

SE1 – Level 4

© Bernd Neuman, L. Dreschler-Fischer, Uni HH, FBI 2005/06

130

Zusammenfassung

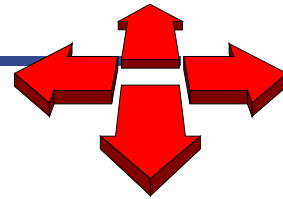


- Ein **Stack** oder Stapel ist ein Sammlungstyp, bei dem Elemente nur nach dem **LIFO-Prinzip** entfernt und hinzugefügt werden können: das zuletzt eingefügte Element wird als erstes zurück geliefert.
- Eine **Queue** oder Schlange ist ein Sammlungstyp, bei dem Elemente nur nach dem **FIFO-Prinzip** entfernt und eingefügt werden können: das zuerst eingefügte Element wird auch als erstes zurück geliefert.
- **Sortieren** ist eine Standardaufgabe in der Informatik. Es gibt unterschiedliche Sortiervverfahren mit spezifischen Stärken und Schwächen hinsichtlich **Komplexität**, **Speicheraufwand** und **Stabilität**.
- **Bubble-Sort** ist ein sehr einfach zu implementierender Algorithmus mit schlechter Effizienz; er ist höchstens für kleine Sammlungen einsetzbar.
- **Quick-Sort** ist einer der schnellsten In-Place-Sortieralgorithmen, er kann jedoch bei bereits sortierten Reihen degenerieren.

SE1 – Level 4

131

Jenseits von Sammlungen: Graphen



- Was sind Graphen?
- Wofür Graphen?
- Implementierung von Graphen
- Durchläufe durch Graphen
- Kürzester Pfad durch einen Graphen

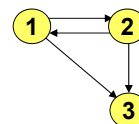
Dank an Norbert Ritter für die Folien-Vorlagen zum Thema Graphen!

SE1 – Level 4

132

Was sind Graphen?

- Ein **Graph** stellt eine Menge von Objekten mit einer Relation auf diesen Objekten dar.
- Formal: Ein (gerichteter) Graph G ist ein Paar $G = (V, E)$, wobei gilt:
 - V ist eine endliche Menge von **Knoten** (engl.: vertex, daher V)
 - E ist eine endliche Menge von **Kanten** (engl.: edge, daher E).
 - formal: E ist eine zweistellige Relation auf V , $E \subseteq V \times V$.
- Bildliche Darstellung
 - eines Knotens durch einen kleinen Kreis
 - einer gerichteten Kante durch einen Pfeil



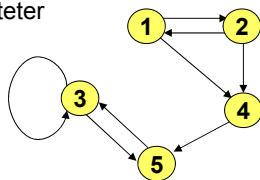
SE1 – Level 4

133

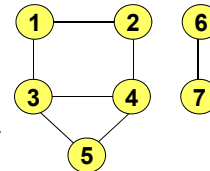
Level 4: Hinter den Kulissen von Sammlungen

Typen von Graphen

Gerichteter Graph

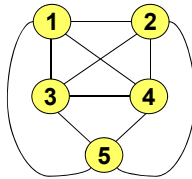


Ungerichteter Graph
(mit symmetrischen Beziehungen)

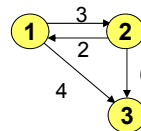


nicht zusammenhängend

Vollständig zusammenhängender Graph



Bewerteter
(gewichteter)
Graph



Alle vier Graphen sind **zyklisch**: Es gibt mindestens einen Pfad, auf dem ein Knoten zweimal vorkommt.

SE1 – Level 4

134

Graphen: in vielen Anwendungen gebraucht

- Graphen spielen in vielen Kontexten eine Rolle, u.a.:
 - Streckennetze öffentlicher Verkehrsmittel
 - Vernetzte Rechner
 - Logistik
- Konkretes Beispiel:
 - Eine Fluggesellschaft soll prüfen können, ob zwischen zwei Flughäfen ausschließlich mit den Flügen der eigenen Gesellschaft verkehrt werden kann.
 - Flughäfen sind dann Knoten, eigene Flugverbindungen sind Kanten in einem Graphen.



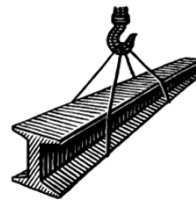
reales Beispiel
(Deutsche BA, 2005)

SE1 – Level 4

135

Implementierung von Graphen

- Es gibt mehrere Möglichkeiten, Graphen zu implementieren.
- Zwei beliebte Methoden:
 - Adjazenzmatrix
 - Adjazenzlisten
- Zwei Knoten u, v sind **adjazent** (=benachbart), wenn sie durch eine Kante verbunden sind.

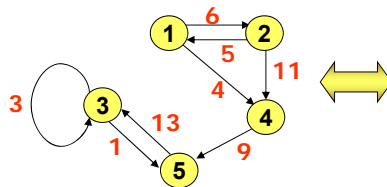


SE1 – Level 4

136

Implementierung Adjazenzmatrix

Für ungewichtete Graphen:
 $A_{ij} = \text{true (1), falls Kante von } i \text{ nach } j$



j =	1	2	3	4	5
i = 1	0	1	0	1	0
2	1	0	0	1	0
3	0	0	1	0	1
4	0	0	0	0	1
5	0	0	1	0	0

j =	1	2	3	4	5
i = 1	∞	6	∞	4	∞
2	5	∞	∞	11	∞
3	∞	∞	3	∞	1
4	∞	∞	∞	∞	9
5	∞	∞	13	∞	∞

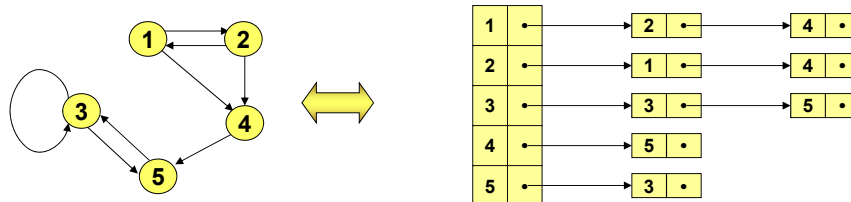
Für gewichtete Graphen können z.B. die „Kosten“ für Zustandsübergang von i nach j eingetragen werden. Besondere Markierung (hier: „unendlich“) für „keine Kante“.

SE1 – Level 4

137

Implementierung Adjazenzlisten

- Jeder Knoten verfügt über eine Liste seiner Nachfolgerknoten.
- Zusätzlich möglich: Alle Knoten sind über ein Array direkt zugreifbar.



- Geringer Platzbedarf, Zugriff auf alle Nachfolgerknoten in Linearzeit.
- Bei Bedarf können auch **inverse Adjazenzlisten** (Listen von Vorgängerknoten) vorgesehen werden.

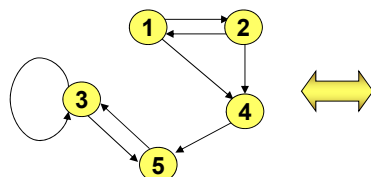
SE1 – Level 4

138

Implementierung Adjazenzlisten (objektorientiert)

Menge

- Jeder Knoten verfügt über eine ~~Liste~~ seiner Nachfolgerknoten.
 - Zusätzlich möglich: Alle Knoten sind über ~~ein Array~~ direkt zugreifbar.
- eine Menge**



Objektorientierte Modellierung

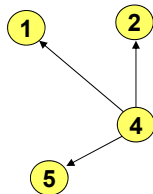
- Eine Klasse modelliert die **Knoten**.
- Mehrere Möglichkeiten für **Kanten**:
 - Implizit durch Referenzen auf Nachbarn;
 - explizit als eigene Klasse modelliert.
- Eine Klasse modelliert **Graphen**:
 - Hält eine Menge von Knoten.

SE1 – Level 4

139

Knoten halten eine Menge von Nachbarn

- Wenn jeder Knoten eine Menge seiner Nachbarknoten hält, ist ein gerichteter Graph automatisch realisierbar.



- Die **Kanten** werden so direkt mit **Java-Referenzen** realisiert.

```

public class Knoten
{
    private Set<Knoten> _nachbarn;

    public Knoten() {
        _nachbarn = new HashSet<Knoten>();
    }

    public boolean hatVerbindungZu(Knoten k)
    {
        return _nachbarn.contains(k);
    }

    public void verbindeMit(Knoten k)
    {
        _nachbarn.put(k);
    }

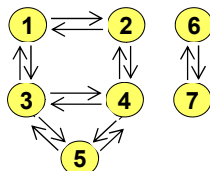
    public void trenneVerbindungZu(Knoten k)
    {
        _nachbarn.remove(k);
    }
}
  
```

SE1 – Level 4

140

Ein Graph hält eine Menge von Knoten

- Für einen **ungerichteten** Graphen muss garantiert sein, dass für eine Verbindung zweier Knoten beide Knoten wechselseitig als Nachbarn eingetragen sind.



```

class Graph
{
    private Set<Knoten> _knotenmenge;

    public Graph()
    {
        _knotenmenge = new HashSet<Knoten>();
    }

    public void neuerKnoten(Knoten k)
    {
        _knotenmenge.add(k);
    }

    public void verbindeDirekt(Knoten k1, Knoten k2)
    {
        k1.verbindeMit(k2);
        k2.verbindeMit(k1);
    }

    public boolean enthaelt(Knoten k)
    {
        return _knotenmenge.contains(k);
    }
}
  
```

SE1 – Level 4

141

Maps: Abbildungen von Schlüssel auf Werte

- **Anforderung:** Für jede Kante soll ein **Gewicht** (bspw. ein int-Wert) gehalten werden.
- Da wir die Kanten nicht als eigenen Objekttyp modelliert haben, stellt sich die Frage, wo die Gewichte gespeichert werden sollen.
- Java bietet eine elegante Lösungsmöglichkeit:
 - Eine **Abbildung** (engl.: **Map**) ist eine **Menge von Schlüssel-Wert-Paaren**:
Schlüssel 1 → Wert 1
Schlüssel 2 → Wert 2
...
Schlüssel n → Wert n
- Der **Schlüssel** ist (wie die Elemente in einer Menge) **eindeutig**.
- Der **Wert** kann ein **beliebiges Objekt** sein (muß nicht eindeutig sein).

SE1 – Level 4

142

Kantengewichte: Map statt Set in den Knoten

Lösungsansatz:

- Ein Knoten enthält nicht nur eine Menge seiner Nachbarn, sondern eine Abbildung von Nachbarn auf Werte.
- Die Nachbarknoten sind also die Schlüssel für die Abbildung, als Wert wird ein Integer-Objekt verwendet.

```
class Knoten
{
    private Map<Knoten,Integer> _nachbarn;

    public Knoten() {
        _nachbarn = new HashMap<Knoten,Integer>();
    }

    public boolean hatVerbindungZu(Knoten k)
    {
        return _nachbarn.keySet().contains(k);
    }

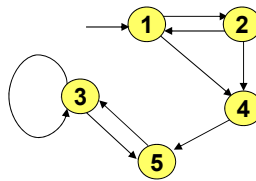
    public void verbindeMit(Knoten k, int gewicht)
    {
        _nachbarn.put(k,new Integer(gewicht));
    }

    public int gibVerbindungsgewicht(Knoten k)
    {
        Integer gewicht = _nachbarn.get(k);
        if (gewicht == null)
            return 0;
        else
            return gewicht.intValue();
    }
}
```

SE1 – Level 4

Durchläufe in Graphen

- Oft soll jeder Knoten nur einmal bearbeitet werden.
- Wiederholtes Erreichen von Knoten möglich:
 - mehrere Pfade
 - Zyklen



Buchführung erforderlich

SE1 – Level 4

144

Durchläufe in Graphen

- Prinzip für **Durchlauf**:
 - Wende **Breitendurchlauf** oder **Tiefendurchlauf** auf die erreichbaren Knoten des Graphen an.
 - **Markiere** dabei besuchte Knoten und vermeide Mehrfachbesuche.
- Merke:
 - Tiefendurchlauf lässt sich rekursiv einfach implementieren.
 - Markierungen „Schon besucht“ müssen vor neuem Durchlauf zurückgesetzt werden. Alternative: Markierungen mit jedem Durchlauf umkehren (Veranschaulichung: Brettspiel Reversi).
 - Statt die Knoten selbst zu markieren, kann die Buchhaltung auch beim Besucher erfolgen.



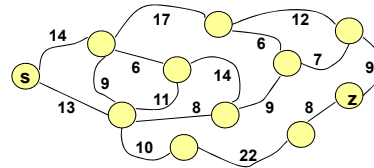
SE1 – Level 4

145

Kürzester Pfad durch einen Graphen



- In **gewichteten Graphen** sind die Kanten mit einem **Gewicht** markiert. Dieses Gewicht kann beispielsweise die Entfernung zwischen Knoten modellieren oder allgemeiner die Kosten für das Verfolgen einer Kante.
- Häufig stellt sich dann die Frage nach dem kürzesten/günstigsten Weg zwischen zwei Knoten **s** und **z**.



- Beispiele:
 - Suche nach dem kürzesten Weg von A nach B in einem Straßennetz,
 - in einem Netz von Versorgungsleitungen oder
 - zum Routing im Internet

SE1 – Level 4

146

Suche nach dem kürzesten Pfad



- Viele Algorithmen beruhen auf dem **Optimalitätsprinzip**:
 - Ist $p = (u_0, u_1, \dots, u_z)$ ein kürzester Pfad von Knoten u_0 nach Knoten u_z ,
 - so ist $p' = (u_i, \dots, u_j)$, $0 \leq i < j \leq z$, auch immer ein kürzester Pfad von u_i nach u_j .



**Um einen kürzesten Pfad zu finden,
kann man von kürzesten Teilpfaden ausgehen.**

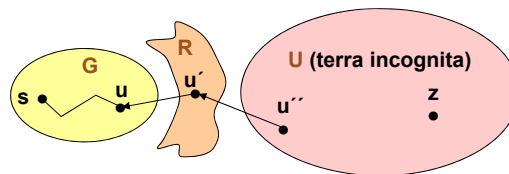
SE1 – Level 4

147

Dijkstras „Wellen“-Algorithmus



- Dijkstras Idee (1959):
 - Vom Startknoten eine „äquidistante Welle“ aussenden, die sukzessiv weitere Knoten erfasst, bis Zielknoten erreicht ist.
- Dabei werden drei Knotenmengen unterschieden:
 - für „gewählte Knoten“ **G** ist **kürzester** Weg vom Startknoten **s** bekannt.
 - für „Randknoten“ **R** ist **ein** Weg von **s** bekannt.
 - für „unerreichte Knoten“ **U** kennt man noch **keinen** Weg.



SE1 – Level 4

148

Dijkstras Algorithmus



1. Initialisierung

Pfadlänge aller Knoten außer **s** ist ∞ , $G = \{s\}$, $R = \{\text{Nachfolger von } s\}$

2. Berechne Wege ab **s**

- Falls **R** leer ist, ist Zielknoten nicht von **s** erreichbar, brich ab.
- Entferne **nächstgelegenen** Randknoten **u** aus **R** und füge ihn in **G** ein.
- **Falls u Zielknoten** ist, gib **Vorgängerliste** von **u** bis **s** aus und brich ab.
- Füge alle **Nachfolger** von **u** in **R** ein, **die nicht in G** sind.
- Vermerke **Pfadlänge und Vorgänger** für kürzesten Pfad zu allen Nachfolgern von **u** in **R**.
- Wiederhole 2.

SE1 – Level 4

149

Level 4: Hinter den Kulissen von Sammlungen

Beispiel für Dijkstras Algorithmus

Gewählt	Randknotenliste (sortiert)
(s 0 s)	(c 13 s) (a 14 s)
(c 13 s)	(a 14 s) (f 21 c) (g 23 c) (b 24 c)
(a 14 s)	(b 20 a) (f 21 c) (g 23 c) (d 31 a)
(b 20 a)	(f 21 c) (g 23 c) (d 31 a)
(f 21 c)	(g 23 c) (e 30 f) (d 31 a)
(g 23 c)	(e 30 f) (d 31 a) (i 45 g)
(e 30 f)	(d 31 a) (h 37 e) (i 45 g)
(d 31 a)	(h 37 e) (i 45 g)
(h 37 e)	(i 45 g) (z 46 h)
(i 45 g)	(z 46 h)
(z 46 h)	

Kürzerer Teilpfad nach b wird entdeckt, Vorgänger und Pfadlänge werden gemerkt.

Ergebnis: s, c, f, e, h, z

Notation: (Name, Pfadlänge, Vorgänger)

SE1 - Level 4
150

Zusammenfassung

- Graphen** und die **Algorithmen** auf ihnen bilden ein wichtiges Teilgebiet der Informatik.
- Ein Graph besteht aus **Knoten** und **Kanten**.
- Kanten können **gerichtet** oder **ungerichtet** sein.
- In einem **gewichteten Graphen** haben die Kanten Gewichte.
- Graphendurchläufe** besuchen jeden erreichbaren Knoten (mindestens) einmal.
- Ein häufig zu lösendes Problem ist, den **kürzesten Weg** in einem Graphen von einem Knoten zu einem anderen zu finden.

SE1 - Level 4
151