Hochschule Offenburg
University of Applied Sciences

Algorithmen und Datenstrukturen
2. Listen

Prof. Dr. Klaus Dorer

Übersicht

- Listen
  - Array Liste
  - Einfach verkettete Liste
  - Doppelt verkettete Liste
  - Stack
  - Queue
  - Listen in Java

Ziele

- Datenstruktur-Operationen implementieren können
- Komplexität der Operationen kennen

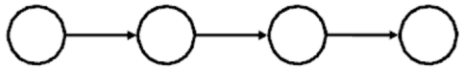
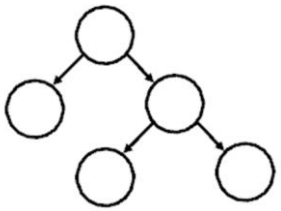
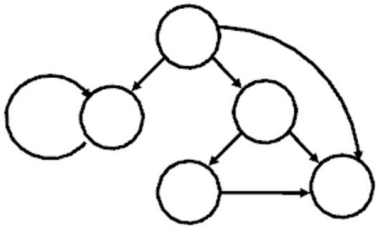
Quellen

- Thomas Ottmann und Peter Widmayer, Algorithmen und Datenstrukturen, 4. Auflage, Spektrum, Berlin, 2002
- www.wikipedia.de
- Daniel Fischer, Algorithmen und Datenstrukturen, 2006
- Stephan Trahasch, AuD, 2019

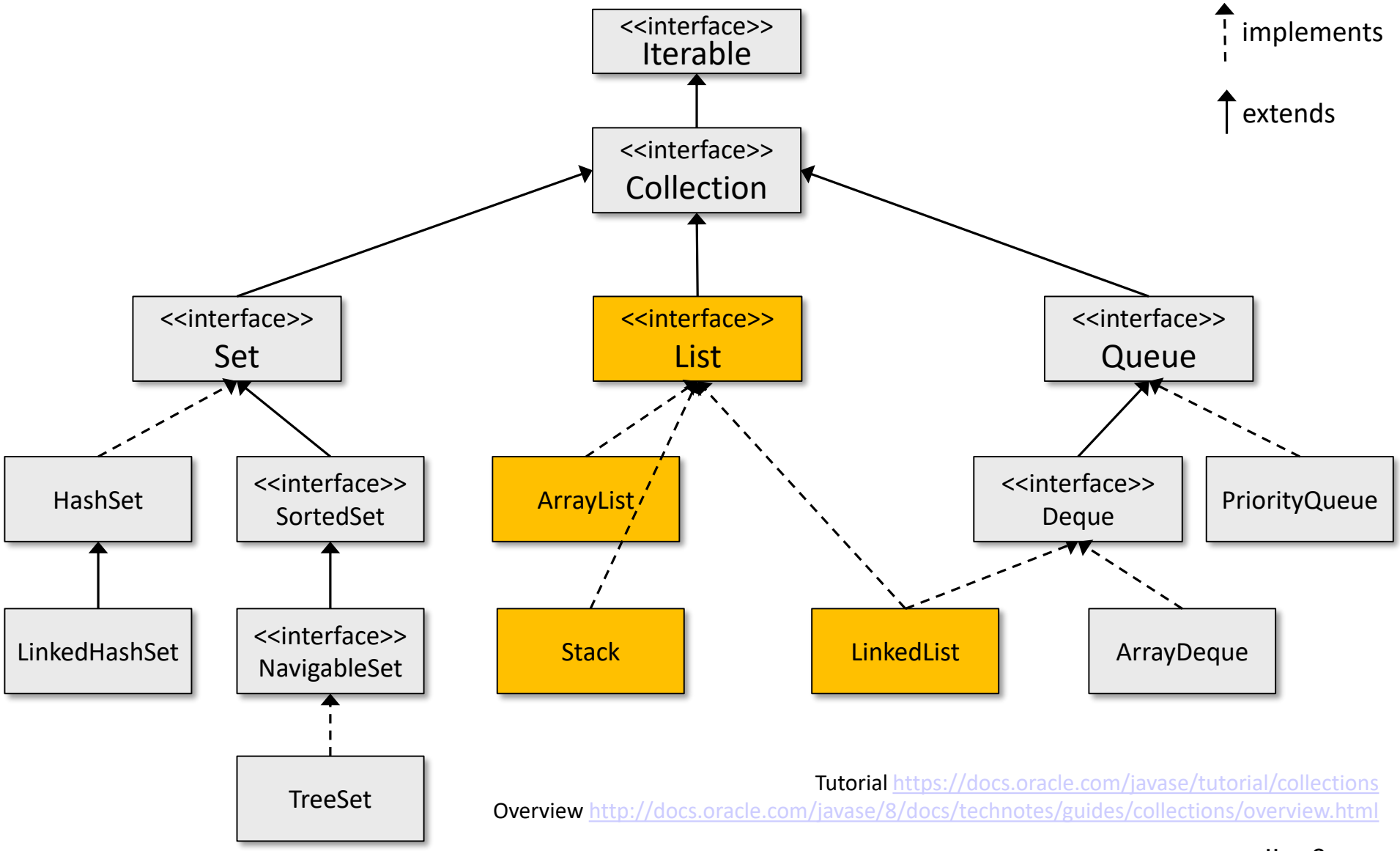
Begriffe

- Datentyp
  - Arten von Objekten, mit denen ein Programm umgehen kann
  - Beispiel
    - Einfache Datentypen: byte, short int, long, float, double, boolean, char
    - Komplexe Datentypen: Arrays, Structs, Klassen
    - Zeiger auf alle Datentypen
- Datenstruktur
  - Kombination von Datentypen und zugehörigen Operationen
  - Beispiel
    - int und +, -, *, /, >, <, ==, …
    - String und equals(String), length(), indexOf(char), substring(int), …
    - Liste und add(Object), remove(Object), get(int), …

Dynamische Datenstrukturen

- Oft weiß man nicht im Vorhinein, wie viele Elemente in einer Datenstruktur gespeichert werden müssen
  - man braucht Datenstrukturen, die beliebig viele Elemente aufnehmen können.
  - Diese Datenstrukturen können wachsen und schrumpfen.
    -> dynamische Datenstrukturen.
- Die wichtigsten dynamischen Datenstrukturen sind:
  - Liste  (Kapitel 2)
  - Baum  (Kapitel 6)
  - Graph  (Kapitel 8)

Java Collection Interface (vereinfacht)



Tutorial https://docs.oracle.com/javase/tutorial/collections

Overview http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Java Collection Interface

`Iterable<E>`:	Über die Datenstruktur kann direkt iteriert werden
`Collection<E>`:	Gruppe von Elementen, Duplikate können erlaubt sein
`Set<E>`:	Menge von Elementen ohne Duplikate
`SortedSet<E>`:	Menge mit einer Totalordnung der Elemente (anhand ihrer natürlichen Ordnung oder anhand eines Vergleichs-Objektes)
`NavigableSet<E>`:	SortedSet mit Methoden, um kleinere oder größere Elemente zu finden
`List<E>`:	Collection mit einer Ordnung, Indexzugriff ist erlaubt
`RandomAccess`:	Marker Schnittstelle, Zugriff in konstanter Zeit möglich
`Queue<E>`:	spezielle Queue-Operationen (FIFO)
`Deque<E>`:	Queue mit Einfüge- und Löschoperationen an Anfang und Ende (FIFO, LIFO)

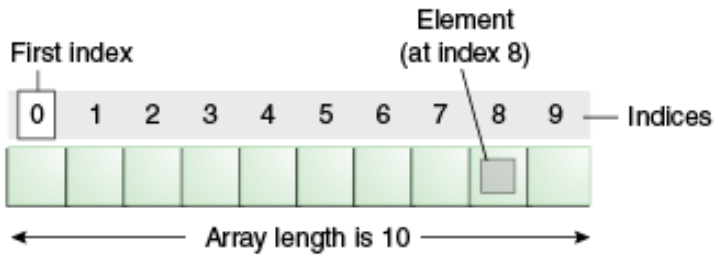
Liste

- Speichert endliche Folge von Elementen eines Grundtyps
  - Reihenfolge ist wichtig
  - Beispiele
    - .
    - .
- Zentrale Operationen
  - Einfügen(position, element) fügt Element an der Stelle position ein
  - Entfernen(position) löscht Element an der Stelle position
  - Zugriff(position) liefert das Element an der Stelle position
  - Suchen(element) sucht das Element in der Liste
  - Abfrage der Anzahl Elemente
- Weitere Operationen
  - Aneinanderhängen von Listen
  - Bilden von Teillisten, …

Wiederholung: Array in Java

- Deklaration eines Arrays in Java
  - Typ[] name = new Typ[anzahl];
- Zugriff
  - Ergebnis = name[index];
- Eigenschaften
  - Wahlfreier Zugriff
  - Größe statisch, zur Deklarationszeit festgelegt

```
1. public class MyClass
2. {
3.   public static void main(String[] args)
4.   {
5.     int[] array1 = new int[10];
6.     String[] array2 = new String[20];
7.     int[] zahlArray = {5, 2, 3, 4};
8.   }
9. }
```

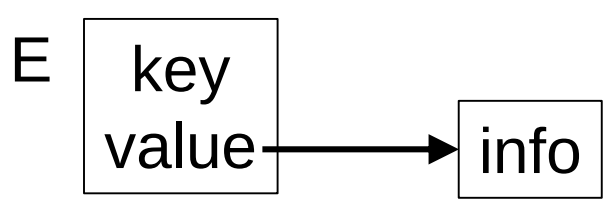


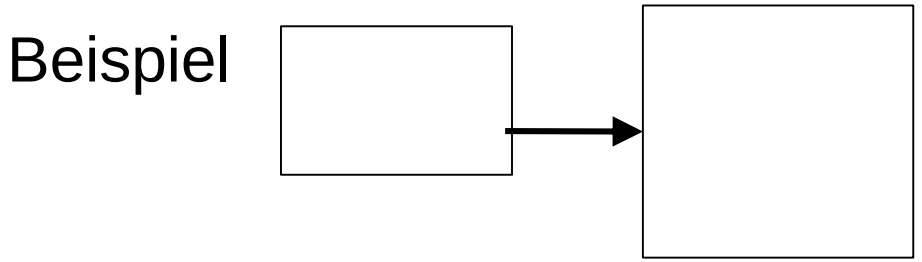
Arraybasierte Liste

- Die Datenelemente werden in einem Array abgelegt, das den direkten Zugriff auf ein Element an der Stelle i erlaubt

```
public class ArrayList<E>
{
        private E[] entry;     // array for the element
        private int size;      // the number of elements stored
        …
}
```

- Die Datenelemente können selbst wieder strukturiert sein





Arraybasierte Liste

- Einfügen(i, element) – `void add(int index, E element);`
  - Verschiebe alle Elemente ab i um eins nach rechts
  - Füge neues Element an Position i ein
  - size++
  - Komplexität:
  - Beispiel: list.add(1, -5)

i	0	1	2	3	4	5	6
entry[i] (vorher)	15	22	4	-2	16		
entry[i] (nachher)							

- Entfernen(i) - `E remove(int index);`
  - Verschiebe alle Elemente ab i+1 um eins nach links
  - Lösche vorige Position des letzten Elements, size--
  - Komplexität:
  - Beispiel: list.remove(0) -> 15

i	0	1	2	3	4	5	6
entry[i] (vorher)	15	22	4	-2	16		
entry[i] (nachher)							

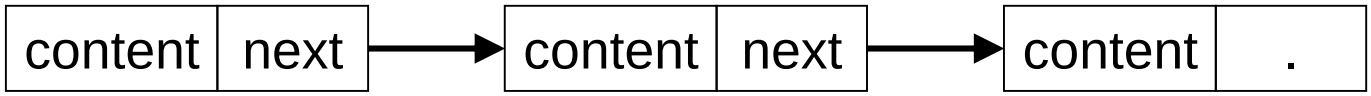
Arraybasierte Liste

- Zugriff(i) - `E get(int index);`
  - Direkter Zugriff durch Adressrechnung möglich
  - Komplexität:
- Suchen(element) - `int indexOf(Object o);`
  - Sequentielle Suche
  - Komplexität:
- Nachteile
  - Größe des Arrays muss vorab festgelegt werden
    - Legt man das Array zu groß an, wird Speicherplatz verschwendet
    - Legt man es zu klein an, kann es beim Hinzufügen nötig werden, ein neues, größeres Array anzulegen und den Inhalt des alten ins neue zu kopieren
  - Alle Elemente müssen an einem zusammenhängenden Platz im Speicher stehen

Einfach verkettete Liste

- Eintrag speichert
  - ein Element
  - Referenz auf das nächste
    - null, falls kein nächstes
    - Alternativ: Referenz auf ein dummy Element (siehe später)

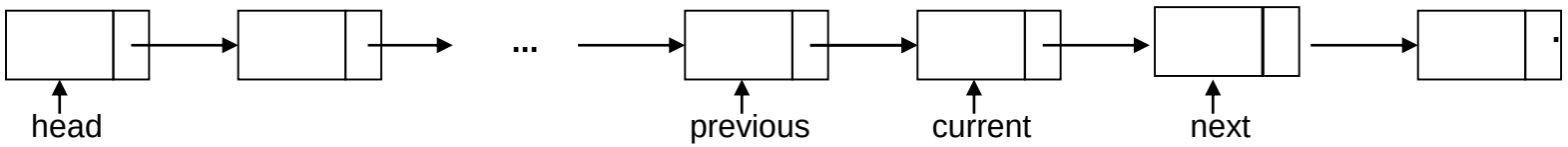
```
class Entry<E>
{
     E content;      // value to represent
     Entry<E> next; // Reference to next element
     …
}
```



Einfach verkettete Liste

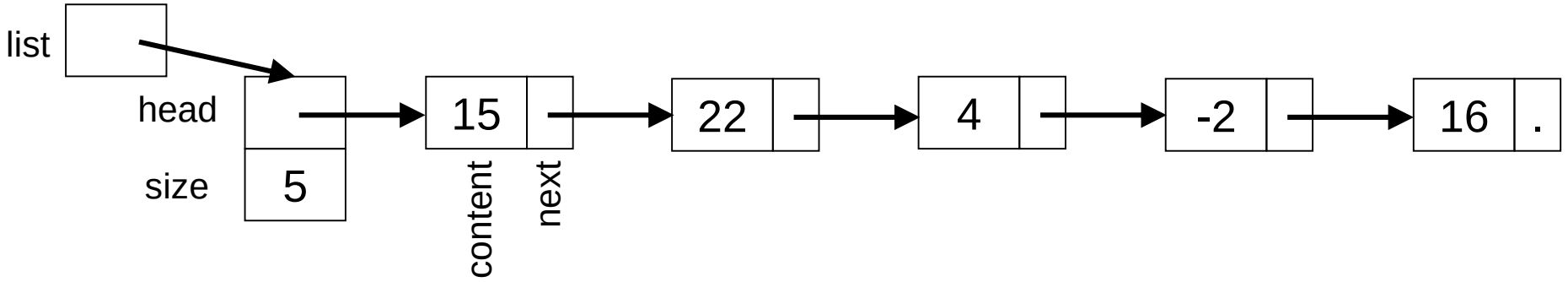
- Die Liste besteht aus
  - einer Referenz auf das erste Element
    - null, wenn die Liste leer ist
    - Alternativ: Referenz auf ein dummy Element (siehe später)
  - der Anzahl Elemente in der Liste

```
public class SingleLinkedList<E>
{
    private Entry<E> head; // Reference to first element
    private int size;
    …
    class Entry<E> {…}     // inner class
}
```



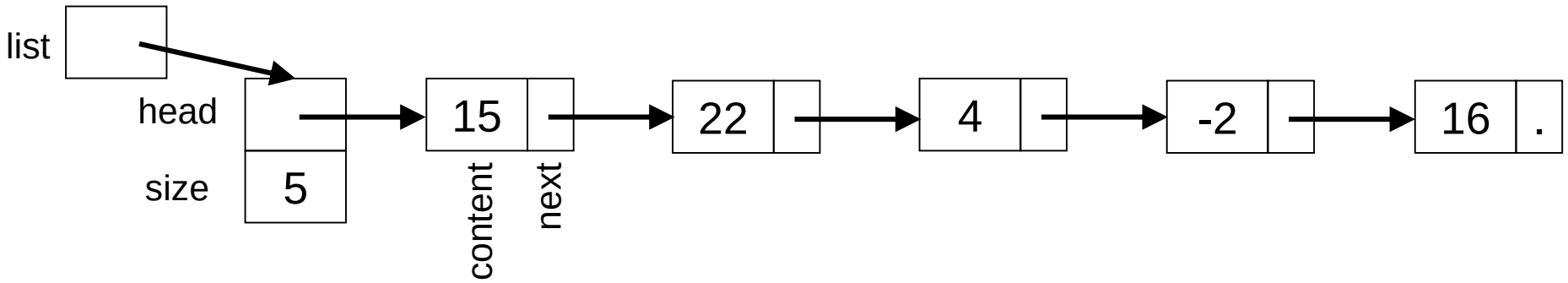
Einfach verkettete Liste

- Einfügen(i, wert)
  - Gehe zum vorigen (i-1.) Element
  - Erzeuge neuen Entry
  - Verbiege Referenzen
  - Passe size an
  - Komplexität:
  - Beispiel: list.add(3, -5)



Einfach verkettete Liste

- Entfernen(i)
  - Gehe wie oben zum i-1. Element
  - Verbiege Referenzen
  - Passe size an
  - Komplexität:
  - Beispiel: list.remove(2)



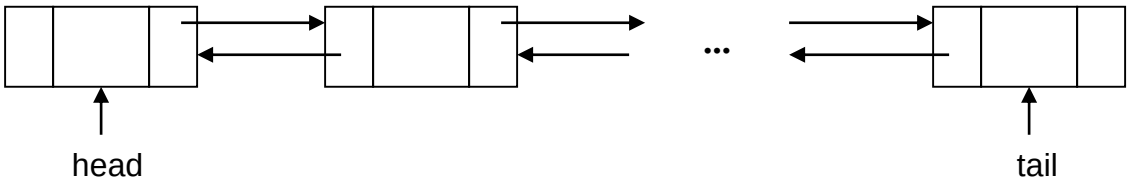
Einfach verkettete Liste

- Zugriff(i)
  - Gehe wie oben zum i. Element
  - Komplexität:
- Suche(wert)
  - Sequentielle Suche beginnend beim Listenkopf
  - Komplexität:
- Vorteile gegenüber Array
  - Größe muss nicht vorab festgelegt werden
  - Elemente müssen nicht hintereinander im Speicher liegen
  - Hinzufügen und Löschen des ersten Elements ist O(1)
- Nachteile
  - Lesen des i. Elements ist O(n) (beim Array O(1))
  - Höherer Speicherverbrauch durch Referenzen?

Doppelt verkettete Liste

- Eintrag referenziert Vorgänger und Nachfolger

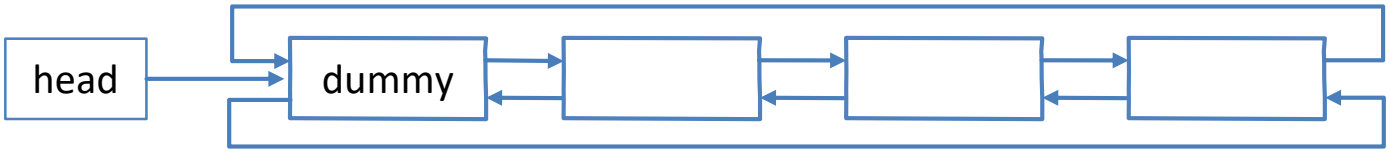
```
public class DoubleLinkedList<E>
{
     private Entry<E> head;    // Reference to first element
     private int size;
     …
     class Entry<E>
     {
          E content;           // value to represent
          Entry<E> previous;   // reference to previous
          Entry<E> next;       // reference to next
          …
     }
}
```



Trick zur Vereinfachung

- Leere Liste besitzt bereits ein „dummy“ Element
  - Dadurch keine Sonderbehandlung für Operationen am Listenanfang / Listenende
  - Einfach verkettete Liste

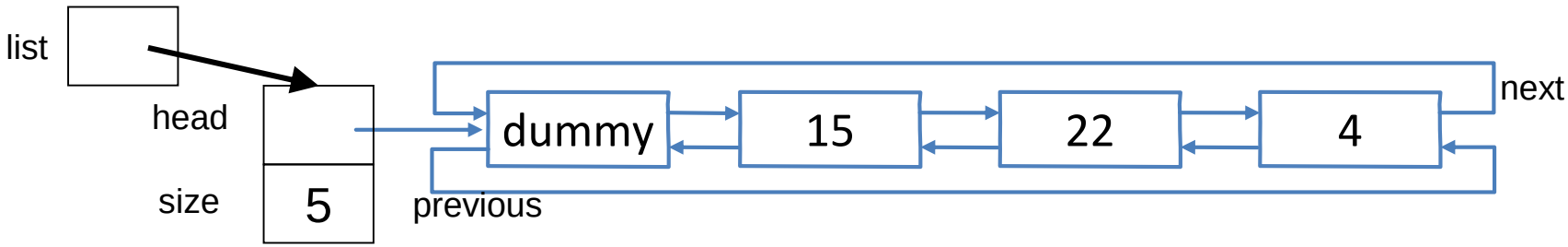
  - Doppelt verkettete Liste



  - Erstes Element ist dann
    - Entry currentElement = head.next;
  - Erkennung des Listenendes
    - while (currentElement != head)

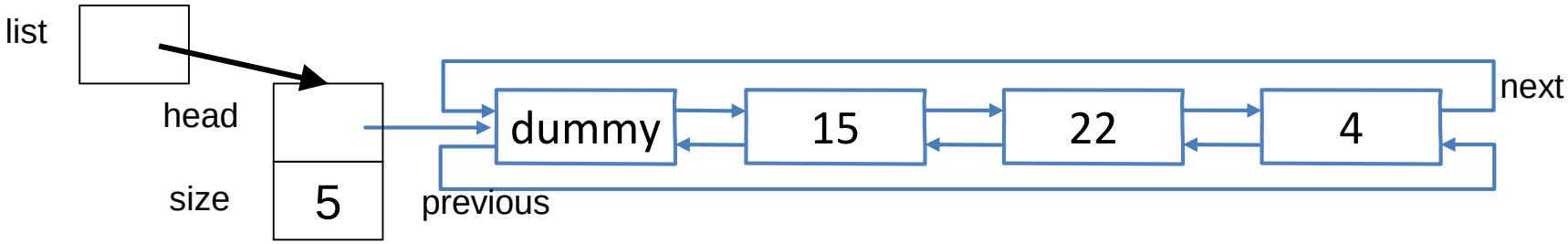
Doppelt verkettete Liste

- Einfügen(i, wert)
  - Starte beim Listenkopf oder Ende, je nachdem was näher ist
  - Sonst wie einfügen bei einfach verketteten Listen, nur dass 4 Referenzen angepasst werden müssen statt 2
  - Komplexität:
  - Beispiel: list.add(0, -5)



Doppelt verkettete Liste

- Entfernen(int i)
  - Gehe zum i. Element wie oben
  - Hänge Referenzen entsprechend um
  - Komplexität:
  - Beispiel: list.remove(0)



- Entfernen(Entry element)
  - Komplexität:

Doppelt verkettete Liste

- Zugriff(i)
  - Gehe wie oben zum i. Element
  - Komplexität?:
- Suche(wert)
  - Sequentielle Suche beginnend beim Listenkopf oder Ende

- Vorteile gegenüber einfach verketteter Liste
  - Einfügen und Entfernen am Ende ist O(1)
  - Entfernen eines bestimmten Elements ist O(1)
  - Element kann in n/2 Schritten erreicht werden
- Nachteile
  - Höherer Speicherverbrauch durch zusätzlichen Zeiger

Zusammenfassung

	Arraybasiert	Einfach verkettet	Doppelt verkettet
Einfügen am Listenanfang	O(n)	O(1)	O(1)
Einfügen an Position (int)	O(n)	O(n)	O(n)
Einfügen an Position (Referenz auf Voriges)	O(n)	O(1)	O(1)
Einfügen am Listenende	O(1)	O(n)	O(1)
Element entfernen (Referenz auf Element)	O(n)	O(n)	O(1)
Zugriff Position (int)	O(1)	O(n)	O(n)

Stack (Stapel)

- Viele Anwendungen arbeiten mit einer LIFO (last in first out) Liste, für die ich nur Operationen am Listenende brauche

Methode	Erläuterung	Komplexität	
		ArrayList	LinkedList
E push(E item)	Einfügen am Listenende		
E peek()	Lesen am Listenende		
E pop()	Lesen und Löschen am Ende		
boolean empty()	Prüfen, ob Stack leer ist		

- Implementierung basierend auf welcher Liste?
- Anwendungen
  - Unterprogrammaufruf
  - Auflösung rekursiver Funktionen in iterative
  - …

Queue (Schlange)

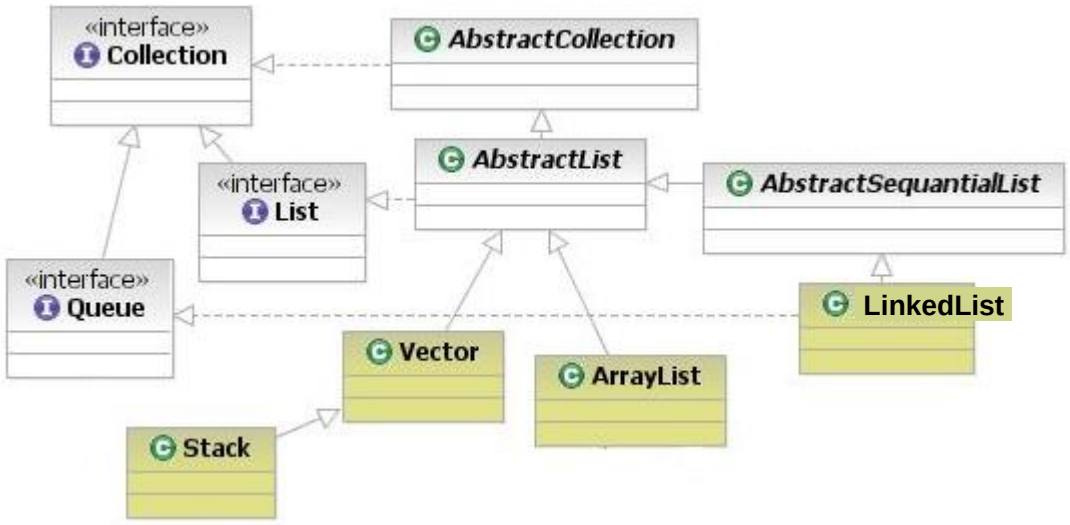
- FIFO (first in first out) Listen erlauben das Hinzufügen am Ende und Lesen am Anfang der Liste

Methode	Erläuterung	Komplexität	
		ArrayList	LinkedList
boolean offer(E item)	Einfügen am Listenende		
E peek()	Lesen am Listenanfang		
E poll()	Lesen und Löschen am Anfang		
boolean empty()	Prüfen, ob Queue leer ist		

- Implementierung basierend auf welcher Liste?
- Anwendungen
  - Unterprogrammaufruf
  - Auflösung rekursiver Funktionen in iterative
  - …

Listen in Java

- Java bietet bereits Klassen für Listen
- Interface List
  - Deklariert alle Methoden, die jede Liste können muss
- Vector
  - Arraybasierte Liste
  - Synchronisiert
- ArrayList
  - Arraybasierte Liste
  - Schneller
- LinkedList
  - Doppelt verkettete Liste



Listen in Java

- Interface List
  - **`add(E o)`**
    - fügt eine Element o vom Typ E am Ende der Liste an
  - **`add(int i, E o)`**
    - fügt eine Element o vom Typ E an Position i ein
  - **`addAll(Collection<? extends E> c)`**
    - fügt alle Elemente der Collection c am Ende der Liste hinzu
  - **`get(int i)`**
    - liefert das i-te Element der Liste zurück (0 basiert)
  - **`remove(Object o)`**
    - entfernt den ersten gefundenen Eintrag von o aus der Liste
  - **`remove(int i)`**
    - entfernt das Element an der Position i
  - **`size()`**
    - liefert die Anzahl der Elemente zurück

Listen in Java

- Interface List
  - **`isEmpty()`**
    - wahr, wenn die Liste leer ist
  - **`clear()`**
    - entfernt alle Elemente aus der Liste
  - **`contains(Object o)`**
    - wahr, wenn o in der Liste enthalten ist
  - **`indexOf (Object o)`**
    - liefert die Position von o oder -1 zurück
  - **`equals(Object o)`**
    - vergleicht das übergegene Objekt mit der Liste auf Gleichheit
  - **`iterator()`**
    - liefert den Iterator über alle Elemente zurück

Listen in Java

- ArrayList
  - realisiert eine lineare Liste als dynamisches Array
- Konstruktoren
  - **`ArrayList()`**
    - erzeugt eine leere Liste mit einem initialen Array der Größe 10
  - **`ArrayList(int i)`**
    - erzeugt eine leere Liste mit einem initialen Array der Größe i
  - **`ArrayList(Collection<? extends E> c)`**
    - erzeugt eine flache Kopie der Collection c
- Deklaration und Definition
  - **`List<E> myList = new ArrayList<E>();`** `// E generisch`

```
List<Person> myList = new ArrayList<Person>();
```

Listen in Java

- LinkedList
  - realisiert eine doppelt verkettete lineare Liste
- Konstruktoren
  - **`LinkedList()`**
    - erzeugt eine leere Liste
  - **`LinkedList(Collection<? extends E> c)`**
    - erzeugt eine flache Kopie der Collection c
- Deklaration und Definition
  - **`List<E> myList = new LinkedList<E>();`**
    // Elemente vom Typ E
  - Beispiel: eine Liste von Personen

```
List<Person> myList = new LinkedList<Person>();
```

Listen in Java: Übung

- Erstellen sie eine Arrayliste, und fügen sie zwei Person Objekte hinzu
  - Person(String name, int age)
- Geben sie die erste Person der Liste aus
- Löschen sie den letzten Eintrag der Liste (ohne zu wissen, dass es zwei sind)