Programmierung 2

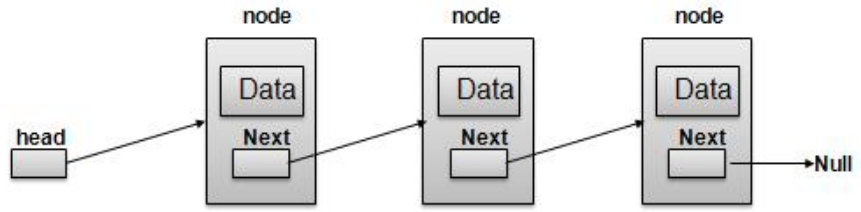
Kapitel 9 – Generics und Collections

Vielen Dank an Prof. Erik Buchmann
für die beigesteuerten Folien

Motivation

- Arrays haben Nachteile
  - Feste Länge
  - Können nur Elemente genau eines Datentyps enthalten
  - Keine Unterstützung für Operationen auf der Datenstruktur
  - Nicht alle Probleme lassen sich mit einem Array oder einer Matrix lösen

- Verkettet Listen lösen einige dieser Probleme

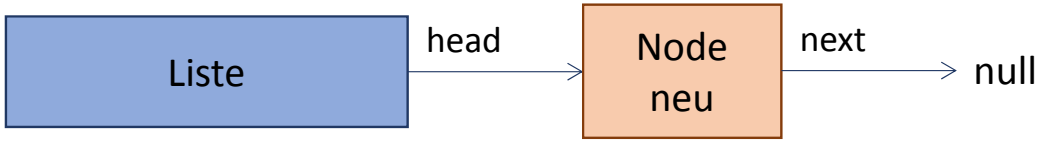


Implementierung einer verketteten Liste

- Anfügen eines neuen Elements zur leeren Liste
  - "Verbiegen des head Zeigers" auf das neue Element

- Vorher

- Nachher

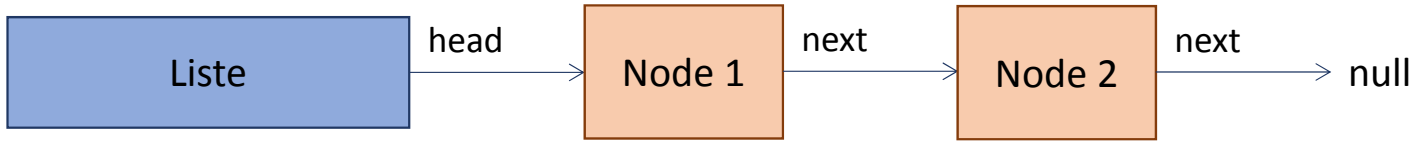


Falls head == null, dann setze head auf das neue Elemente

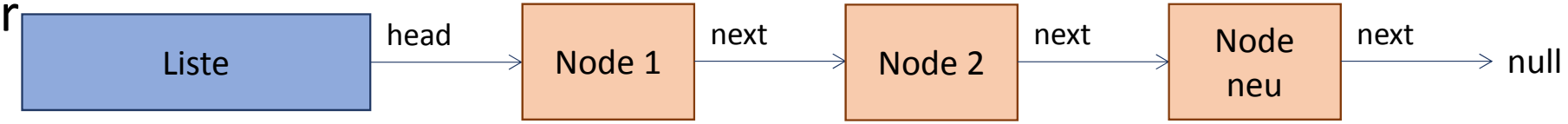
Implementierung einer verketteten Liste

- Anfügen eines neuen Elements zu einer Liste mit Elementen
  - "Verbiegen des Zeigers" des letzten Nodes der Liste auf das neue Element

- Vorher



- Nachher



Iteriere über alle Elemente der Liste bis Node.next == null, dann füge Elemente hinzu

Implementierung einer verketteten Liste (Version 1)

- Verkette Liste für das Speichern von int Werten (siehe Liste1.java)

- Vorteile gegenüber einem Array
  - Dynamische Länge die zur Laufzeit verändert werden kann
  - Unterstützung für verschiedene Operationen zur Manipulation von Listenelementen
    - Anhängen eines Elements
    - Einfügen eines Elements an einem Index
    - Zugriff auf ein Element an einem Index
    - Überschreiben eines Elements an einem Index
    - Löschen eines Elements an einem Index
    - Rückgabe der Anzahl der Elemente
    - …

- Nachteile
  - Aktuell muss eine Liste pro Datentyp implementiert werden
  - Verschiedene Datentypen können nicht in einer Liste gespeichert werden

Wie könnte man diese Probleme lösen?

Implementierung einer verketteten Liste (Version 2)

- Implementierung des Datenanteils mit der Klasse Object (siehe Liste2.java)

- Vorteile
  - Implementierung unterstützt alle Referenzdatentypen
  - Verschiedene Datentypen können nun in einer Liste gespeichert werden

- Nachteile
  - Wrapper-Objekte für primitive Datentypen notwendig (aber Autoboxing/Unboxing)
  - Typumwandlungen während der Interaktion mit den Elementen einer Liste notwendig

Wie könnte man dieses Problem lösen?

Implementierung einer verketteten Liste (Version 3)

- Implementierung als generische Klasse (siehe Liste3.java)
- Vorteile
  - Implementierung unterstützt alle Referenzdatentypen
  - Keine Typumwandlungen während der Interaktion mit den Elementen einer Liste notwendig
- Nachteile
  - Verschiedene Datentypen können nicht in einer Liste gespeichert werden
  - Wrapper-Objekte für primitive Datentypen notwendig (aber Autoboxing/Unboxing)

Generische Klassen: Klassendefinition

- Definition einer generischen Klasse durch Angabe von Typ-Parameter(n) hinter dem Klassennamen

- Beispiel: class AGenericClass<T> {…}
  - T muss der Bezeichner einer Klasse sein
  - T kann im weiteren Klassenkopf und im Klassenrumpf (fast) überall da verwendet werden, wo Klassennamen stehen können

```
class D<T> extends T {}
class E<T> {T obj;}

interface I <T1, T2> {
public void setze(T1 arg0, T2 arg1)
}

class F<T4, T5, T6> implements I<T4, T6> {
public T5 liefere() {…}
public void setze(T4 arg0, T6 arg1) {…}
}
```

Beispiel: Typ-Parameter als Einschränkungen

- Motivation: Modellierung einer Getränkeflasche
  - Implementierung einer Klasse Bottle pro Getränk ist zu aufwendig
  - Bottle<Object> ist zu ungenau und lässt auch gefährliche Flüssigkeiten zu

- Lösung: Einschränkung des Typ Parameters

```
abstract class Drink{}      // Oberklasse für alle Getränke
class Beer extends Drink {} // Bier ist ein Getränk
class Petrol {}             // Benzin ist kein Getränk

class Bottle<T extends Drink> {} // Zielparameter T muss ein Getränk sein

Bottle<Drink> a = new Bottle<Drink>();
Bottle<Beer> b = new Bottle<Beer>();
Bottle<Object> c = new Bottle<Object>(); // Fehler
Bottle<Petrol> d = new Bottle<Petrol>(); // Fehler
```

Generische Klassen

- Generisch = parametrisiert
- Vorteile
  - Sicherstellung Semantische Korrektheit
  - Vermeidung von Typumwandlungen
  - Vermeidung von ClassCastExceptions

Java Collections Framework

Java Collections Framework

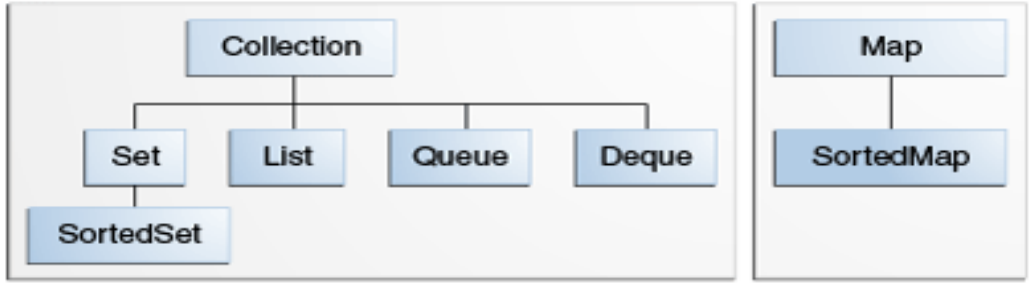
- Interfaces
  - Definieren abstrakte Datentypen die es ermöglichen Kollektionen unabhängig von ihrer internen Implementierung zu manipulieren
- Implementationen
  - Wiederverwendbare Datenstrukturen
  - Konkrete Implementierungen der Interfaces
- Algorithmen
  - Operationen für die Manipulation von Datenstrukturen, z.B. Suche, Sortieren, Reihenfolgeumkehr, Einfügen oder Löschen

Java Collections Framework – Vorteile

- Code Wiederverwendung
- Vereinfachte Programmierung, schneller
- Interoperabilität zwischen unabhängigen APIs
- Einfacher neue APIs zu entwickeln, weil man Framework verwenden kann
- Gesteigerte Effizienz und Robustheit

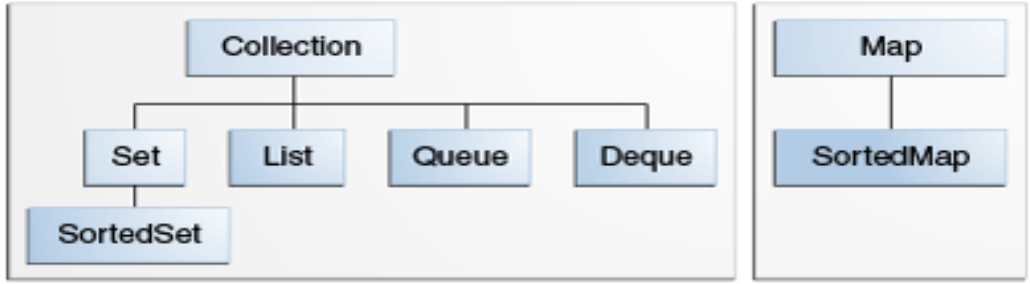
Interfaces

- Set (Menge)
  - Keine Duplikate, maximal ein Null Element
  - Achtung bei Manipulation einzelner Elemente mit Bezug auf Unterscheidbarkeit von Elementen
  - Keine Reihenfolge festgelegt
    - SortedSet: sortiert eingefügte Elemente in aufsteigender Reihenfolge
- List (Liste)
  - Eine sortierte Menge von Elementen
  - Duplikate und mehrere Null Elemente möglich
  - Indizierung startet bei 0, Reihenfolge wird über Indizes kontrolliert
  - Random Access auf Elemente



Interfaces

- Queue (Warteschlange)
  - FIFO, LIFO, Prioritätsgeordnet
- Deque (Doppelschlange)
  - FIFO, LIFO, lineare Ordnung
  - Elemente können auf beiden Seiten des Deque eingefügt und entfernt werden
- Map
  - Verwaltet Key-Value Paare
  - Keine Duplikate erlaubt
  - Keine Reihenfolge festgelegt
    - SortedMap: sortiert eingefügte Elemente aufsteigend nach Key Wert



Implementierungen

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- Alle Implementierungen
  - Erlauben Null Elemente
  - Sind nicht synchronisiert

- Im Gegensatz: Legacy Implementierungen
  - Hashtable und Vector sind synchronisiert
  - Hashtable erlaubt keine Null Elemente

Unterschiedliche List Implementierungen

- ArrayList
  - Schneller Zugriff auf indiziertes Elemente (konstante Zeit)

- LinkedList
  - Schneller wenn über die Elemente der Datenstruktur iteriert wird und neue Elemente hinzugefügt werden
  - Java-Version der verketteten Liste die wir beispielhaft definiert hatten (siehe Liste3.java)

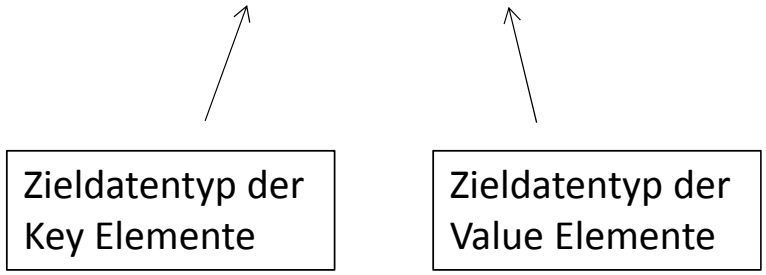
Beispiel: HashMap

Key		Value
Lisa	⟶	Hauptstraße 15, 04277 Leipzig
Steve	⟶	Fischerstrasse 20, 04105 Leipzig
Pete	⟶	Neuer Weg 140, 04103 Leipzig

- Was ist eine HashMap?
  - Menge von Key-Value Paaren
  - Schlüssel wirkt als Index auf genau ein Value Element
  - Schlüssel Duplikate sind nicht möglich
  - Es besteht keine Ordnung (definierte Reihenfolge) für Key Elemente

- Generischer Datentyp, parametrisierbar

```
HashMap<String, Adresse> map = new HashMap<String, Adresse>();
```



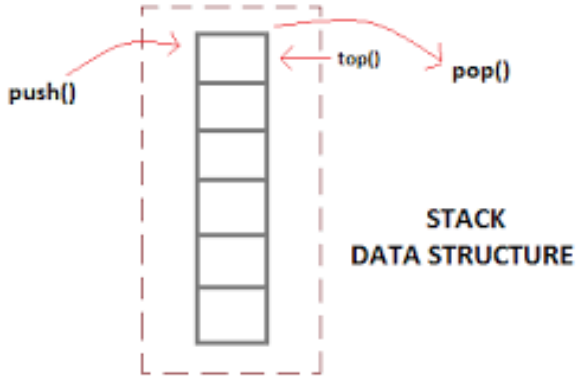
Anwendung: HashMap

- Erstellung einer Adress-Datenbank zur Verwaltung von Clubmitgliedern
- Zuordnung von Personen zu Ihren Adressen
- HashMap Key-Value Paare
  - Key = Personenname, z.B. Lisa
  - Value = Addresse, z.B. Hauptstrasse 15, 04277 Leipzig
- Siehe HashMapExample.java

Weitere Datenstrukturen

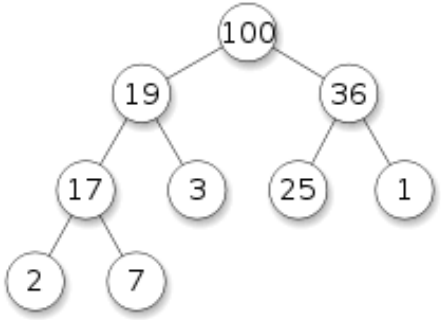
Weitere Datenstrukturen – Stapel

- Stapel (Stack)
  - Elemente werden nur oben auf dem Stapel abgelegt und entfernt (LIFO)
  - Operationen: push, pop, peek



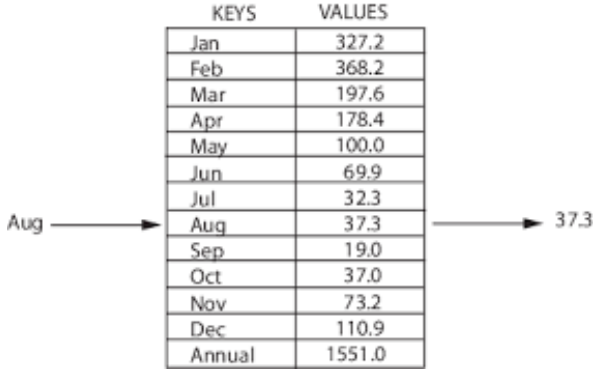
Weitere Datenstrukturen – Menge

- Menge (Set)
  - Verwaltung einer Menge unterscheidbarer Elemente (keine Duplikate)
  - Es besteht normalerweise eine Ordnung auf den Elementen (oftmals aber keine ermittelte Reihen/Rangfolge)
  - Unterscheidbarkeit muss bei der Manipulation von Elementen bedacht werden



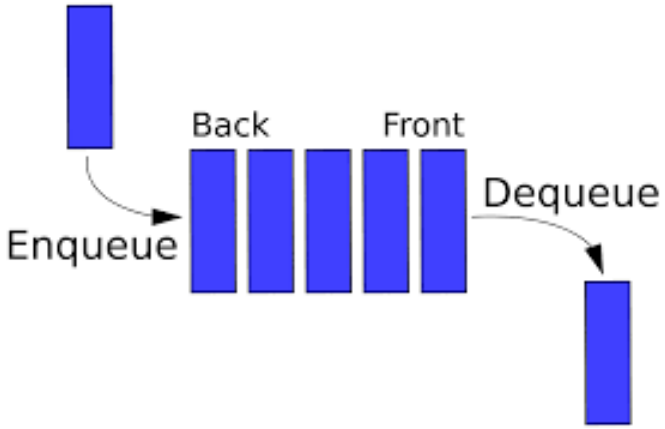
Weitere Datenstrukturen – Wörterbuch

- Wörterbuch (Map)
  - Key-Value Paare: Schlüssel der auf ein Datenelement verweist
  - Keys müssen unterscheidbar sein (keine Duplikate)
  - Ordnung besteht auf den Schlüsselwerten



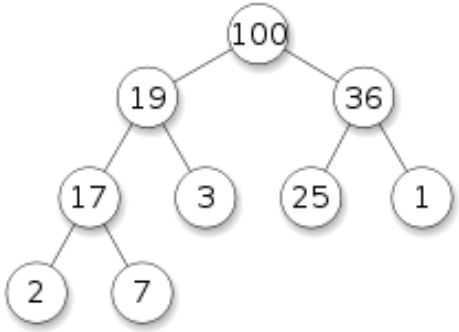
Weitere Datenstrukturen – Warteschlange

- Warteschlange (Queue)
  - Lineare Ordnung: FIFO, LIFO, Prioritätsgeordnet
  - Speichert typischerweise Elemente bevor sie verarbeitet werden
  - Operationen: dequeue, enqueue



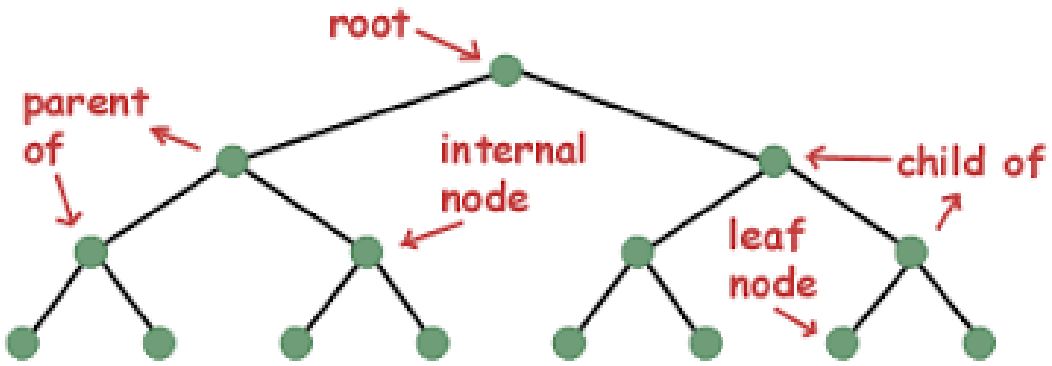
Weitere Datenstrukturen – Halde

- Halde (Heap)
  - Meist als Binärbaum implementiert
  - Es besteht eine Ordnung auf den Elementen
    - Parent enthält Wert der größer-gleich dem der Kinder ist (max heap)
    - Parent enthält Wert der kleiner-gleich dem der Kinder ist (min heap)
  - Operationen: insert, remove, extractMin, extractMax



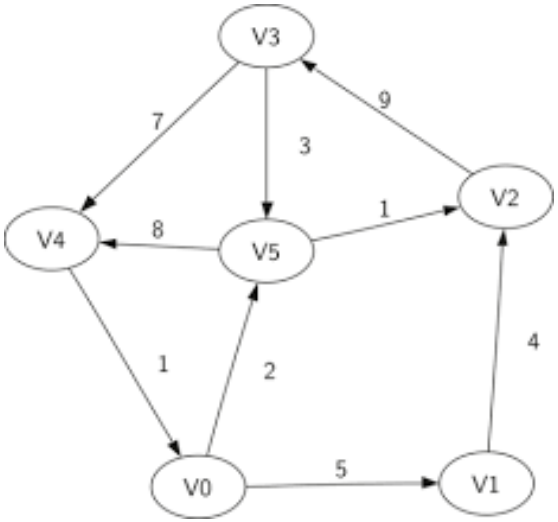
Weitere Datenstrukturen – Baum

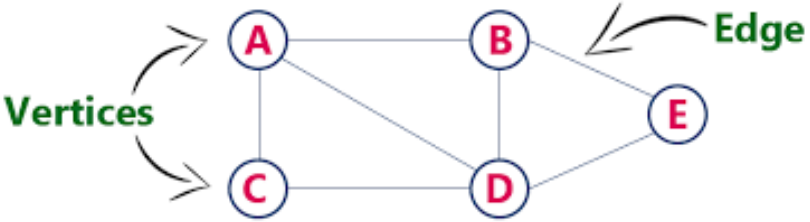
- Baum (Tree)
- Operationen: Tree Traversal, In-order/Pre-order/Post-order Walks
- Weitere Operationen: einfügen, löschen, sortieren, suchen, Teilgraphen entfernen/hinzufügen,



Weitere Datenstrukturen – Graph

- Graph
  - Besteht aus Knoten (Vertices) und Kanten (Edges)
- Operationen
  - Test ob Kanten zwischen zwei Punkten existieren
  - Auflistung aller Nachbarn eines Knoten
  - Einfügen von Knoten und Kanten
  - Lesen und Schreiben von Knoten oder Kantengewichten
  - Bestimmung minimaler Distanzen zwischen Knoten





Übung

- Mit welcher Datenstruktur könnte man den Array in der Bank Terminal Beispielanwendung ersetzen?
- Welche Vorteile ergeben sich?
- Gibt es Nachteile?

```
public class Bank {
    private Konto[] kontos;
    private int zaehler;

    public Bank() {
        kontos = new Konto[MAX_ANZAHL_KONTEN];
        zaehler = -1;
    }

    public Konto[] getKontos() {
        return kontos;
    }
…
```

Antworten

- Mit welcher Datenstruktur könnte man den Array in der Bank Terminal Beispielanwendung ersetzen?
  - Mit einer Liste, z.B. Klasse ArrayList oder Klasse LinkedList

```
LinkedList<Konto> kontos = new LinkedList<Konto>();
```

- Welche Vorteile ergeben sich?
  - Beliebige Anzahl von Konten können nun verwaltet werden
  - Methoden der Datenstruktur implementieren bereits verschiedene Operationen, damit spart man sich den Aufwand der Neuimplementierung (Folgen: effiziente Programmierung und robuster Code)
- Gibt es Nachteile?
  - Mit und ohne Parametrisierung mit Zielklasse Konto sind Typumwandlungen notwendig, da verschiedene Kontotypen in der gleichen Liste gespeichert werden

Wie könnte man diese Probleme lösen?

Zusammenfassung

- Arrays haben Limitierungen die man mit dynamischen Datenstrukturen umgehen kann
- Java Collections Framework stellt verschiedene Interfaces, Implementierungen und Algorithmen bereit um die Implementierung von Datenstrukturen zu erleichtern
- Das Framework bietet verschiedene Vorteile, u.a. Code Wiederverwendung, vereinfachte Programmierung, Gesteigerte Effizienz und Robustheit

Kontrollfragen

- Welche Datenstruktur erweitert einen Array derart, dass die Anzahl der Elemente nicht mehr begrenzt ist?
- Welche Implementierungen hält Java für die Umsetzung der genannten Datenstruktur bereit?
- Nennen Sie ein Anwendungsbeispiel für eine HashMap