Java Collection: Extensive repetition s. Code

Graphs

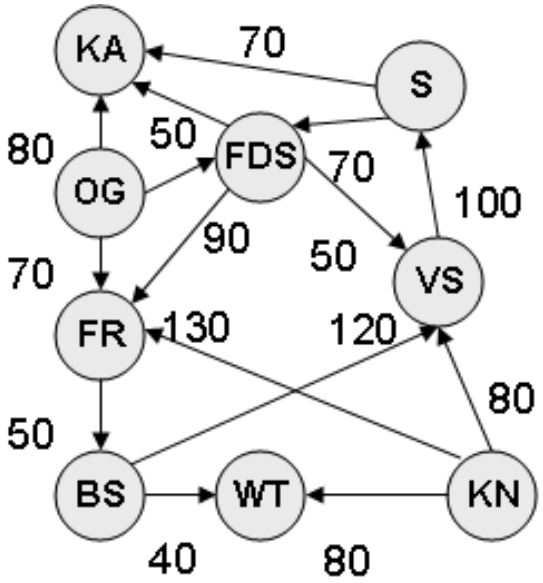
Tree:

- 0..1 ingoing branches
- 0...n outgoing branches

Graph

- 0..n ingoing branches
- 0...n outgoing branches

Recap

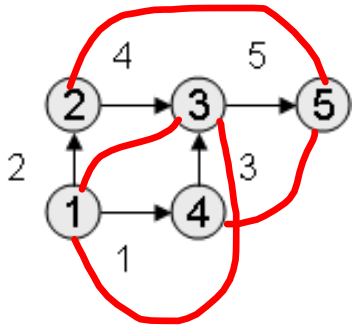


Uses Cases

- Route planning
  - > shortests distance between nodes
  - > traveling salesman
- Flows (strom, good, sewage)
  - > maximum throughput
- Project management
  - > critical path
  - > shortest time
- Google page rank

gerichtet / ungerichtet | direct / non-directed graphs

gewichtet / ungewichtet | weighted / unweighted graphs

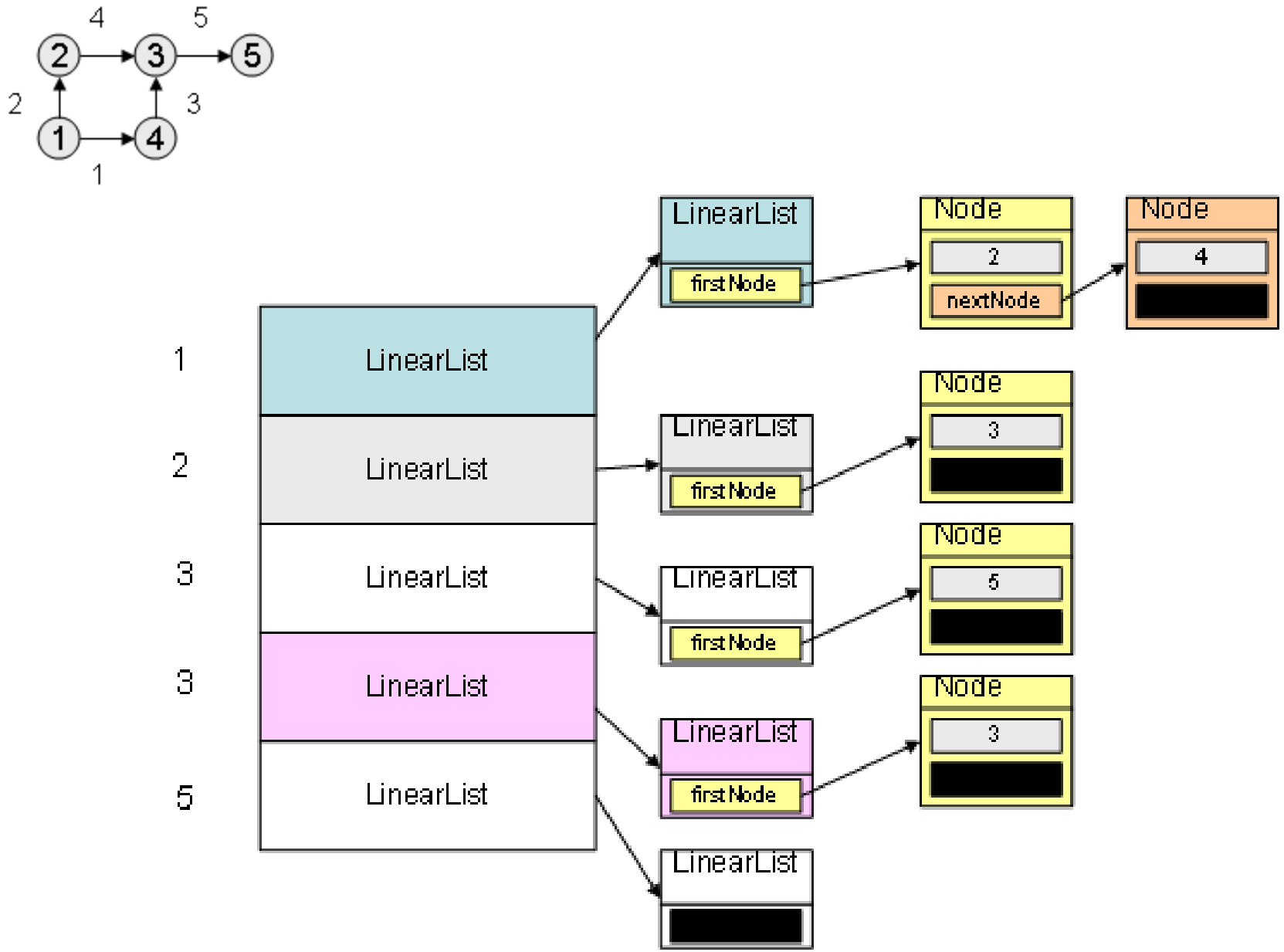


Recap

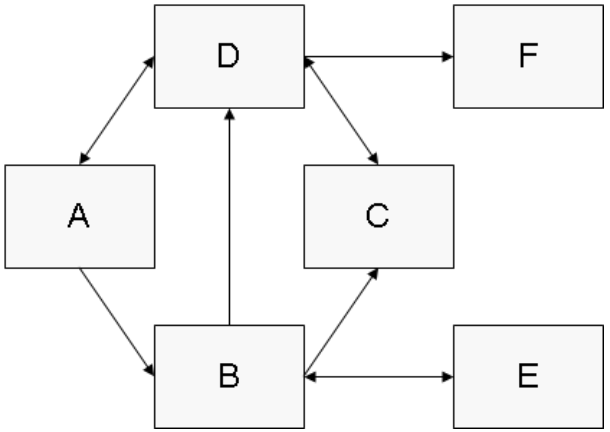
Adjazenzmatrix / Adjacency Matrix

Transitive Hülle / Transitive Envelope

	1	2	3	4	5
1		2	4	1	9
2			4		9
3					5
4			3		8
5					


4
5
2
3
5
2
3
1
4
1
LinearList
Node
2
4
firstNode
nextNode
1
LinearList
2
LinearList
3
LinearList
3
LinearList
5
LinearList
LinearList
Node
3
Node
5
Node
3
firstNode

Google Page Rank Algorithm



Recap

A = 4/5 * (1/ 3* D) + 1/5

B = 4/5 * (1/2 * A + E) + 1/5

C = 4/5 * (1/3 * B + 1/3* D) + 1/5

D = 4/5 * (1/2 * A + 1/3 * B + C) + 1/5

E = 4/5 * (1/3 * B) + 1/5

F = 4/5 * (1/3 * D) + 1/5

4/5 = Damping Factor (Dämpfungsfaktor). It expresses the percentageof vistors coming via links, the rest (here 20% = 1/5) directly navigates the web page.

A = 4/5 * (1/ 3* 1) + 1/5 = 7/15

B = 4/5 * (1/2 * 1 + 1) + 1/5 = 14/10

C = 4/5 * (1/3 * 1 + 1/3* 1) + 1/5 = ...

D = 4/5 * (1/2 * 1 + 1/3 * 1 + 1) + 1/5

E = 4/5 * (1/3 * 1) + 1/5

F = 4/5 * (1/3 * 1) + 1/5

A = 4/5 * (1/ 3* D) + 1/5

B = 4/5 * (1/2 * 7/15 + E) + 1/5 =

C = 4/5 * (1/3 * 14/10 + 1/3* D) + 1/5

D = 4/5 * (1/2 * 7/15 + 1/3 * 14/10 + C) + 1/5

E = 4/5 * (1/3 * 14/10) + 1/5

F = 4/5 * (1/3 * D) + 1/5

Java 8: Working with functional interfaces and lambda expressions

It is not possible in Java prior to version 8 to pass functions as arguments.

*StartLambdasNew.java

```
1  package lambda;
2
3  public class StartLambdasNew {
4      public static void main(String[] args) {
5          StartLambdasNew lambdas = new StartLambdasNew();
6
7          lambdas.getLocation(displayMap(String location)));
8
9      }
10
11     public String getLocation(function f) {
12         //returns location e.g. as Latitudes and Longitudes
13         String location = "Konstanz";
14         f(location);
15     }
16
17
18     public void displayMap(String location) {
19         //display the map
20         System.out.println("We are showing on the map " + location);
21     }
22 }
23
24
```

The only mean is to use Interfaces (hier IDisplayMap)

```java
package lambda;

import java.util.function.Consumer;

public class StartLambdasNew {
    public static void main(String[] args) {
        StartLambdasNew lambdas = new StartLambdasNew();

        //Version 1: anonymous inner class
        lambdas.getLocation(new IDisplayMap() {

            @Override
            public void displayMap(String location) {
                System.out.println("We are showing on the map (1) " + location);
            }
        });

        //Version 2
        IDisplayMap displayer = new IDisplayMap(){

            @Override
            public void displayMap(String location) {
                System.out.println("We are showing on the map (2) " + location);
            }
        };
        lambdas.getLocation(displayer);
```

IDisplayMap.java

```java
1  package lambda;
2
3  public interface IDisplayMap {
4      public void displayMap(String location);
5  }
6
```

We still have to write an interface1

Using Java 8 we can pass methods/functions as method parameters using functional interfaces

```
        //Version 3: Using Java 8 & functional interfaces (functional interfaces have exactly one method!)
        IDisplayMap displayer2 = (String location) -> System.out.println("We are showing on the map (3) " + location);
        lambdas.getLocation(displayer2);

        //Version 4: Using Java 8 without parameter types (this can be derived)
        IDisplayMap displayer3 = (location) -> System.out.println("We are showing on the map (4) " + location);
        lambdas.getLocation(displayer3);

        //Version 5: Using Java 8 without parenthesis
        IDisplayMap displayer4 = location -> System.out.println("We are showing on the map (5) " + location);
        lambdas.getLocation(displayer4);

        //Version 6: Using Java 8: for implementations with multiple lines
        IDisplayMap displayer5 = location -> {
            System.out.println("We are showing on the map (6) " + location);
            System.out.println("Just another statment");
        };
        lambdas.getLocation(displayer5);

        //Version 7: Using Java 8 and predefined function interfaces
        Consumer<String> displayer6 = location -> System.out.println("We are showing on the map (7) " + location);
        lambdas.getLocation1(displayer6);

        //Version 8: Using Java 8 and functional interfaces
        lambdas.getLocation1(location -> System.out.println("We are showing on the map (8) " + location));

    }


    public void getLocation1(Consumer<String> function) {
        String location = "Konstanz";
        function.accept(location);
    }


    public void getLocation(IDisplayMap displayer) {
        String location = "Konstanz";
        displayer.displayMap(location);
    }
```

Using our functional interface IDisplayMap

Using predefined functional interfaces

functional interfaces must have exactly one method

There are many predefined functional interfaces

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the oper
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two double-valued operands and producing a double-valued result.
DoubleConsumer	Represents an operation that accepts a single double-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one double-valued argument.
DoubleSupplier	Represents a supplier of double-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single double-valued operand that produces a double-valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation upon two int-valued operands and producing an int-valued result.
IntConsumer	Represents an operation that accepts a single int-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.
IntPredicate	Represents a predicate (boolean-valued function) of one int-valued argument.
IntSupplier	Represents a supplier of int-valued results.
IntToDoubleFunction	Represents a function that accepts an int-valued argument and produces a double-valued result.
IntToLongFunction	Represents a function that accepts an int-valued argument and produces a long-valued result.

2 more examples for the application of lambda expressions

StartLambdasNew.java | StartLambdasOld.java | StartFunctionalInterfaces.java

```java
package lambda;

import java.util.function.DoubleBinaryOperator;
import java.util.function.Predicate;

public class StartFunctionalInterfaces {
    public static void main(String[] args) {
        double result = calculate(2, 3, (a, b) -> a - b);

        System.out.println("The result is " + result);

        Predicate<String> check = b -> b.toUpperCase().equals(b);
        String[] words = new String("SOTE is one class at the HTWG").split(" ");
        evaluate(words, check);

    }


    private static double calculate(double arg1, double arg2, DoubleBinaryOperator operator) {
        return operator.applyAsDouble(arg1, arg2);
    }

    private static void evaluate(String[] words, Predicate<String> predicate) {
        for (int i = 0; i < words.length; i++) {
            if(predicate.test(words[i])) {
                System.out.println("the letter meeting the criteria is " + words[i]);
            }
        }

    }
}
```

checks whether word is already upper case