Fakultät IV

Klausur MPGI 2
02. Oktober 2012

Brock
Eppner / Höfer / Putz / Rouiller

Name: .....

Vorname: .....

Matr.-Nr.: .....

Bearbeitungszeit: 75 Minuten

- Es ist ein auf beiden Seiten beschriebenes DIN-A4 Blatt als Hilfsmittel zugelassen, wenn es Deinen Namen und Deine Matrikelnummer enthält. Nicht zugelassen sind elektronische Hilfsmittel, wie z. B. Taschenrechner, Handys oder Laptops.
- Benutze für die Lösung der Aufgaben nur das mit diesem Deckblatt ausgeteilte Papier. **Lösungen, die auf anderem Papier geschrieben werden, können nicht gewertet werden!**
- Schreibe Deine Lösungen auf das Aufgabenblatt der jeweiligen Aufgabe. Verwende auch die Rückseiten. **Schreibe keine Lösungen einer Aufgabe auf ein Blatt, das nicht zu dieser Aufgabe gehört!** Wenn Du zusätzliche, von uns ausgegebene Blätter verwendest, gib unbedingt an, zu welcher Aufgabe die Lösung gehört!
- Schreib deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen werden nicht gewertet! Streiche gegebenenfalls eine Lösung durch!
- Schreibe nur in **blau** oder **schwarz**. Lösungen, die mit Bleistift geschrieben sind, können nicht gewertet werden!
- Erscheint Dir eine Aufgabe mehrdeutig, wende dich an die Betreuer.
- Du kannst die Aufgaben in einer beliebigen Reihenfolge bearbeiten.
- Gib Nebenrechnungen an, um Deinen Lösungsweg zu verdeutlichen.
- Trage *jetzt* (vor Beginn der Bearbeitungszeit) auf *allen* Blättern Deinen Namen und Deine Matrikelnummer ein.

	Punkte	Erreichte Punkte
1	12	
2	8	
3	5	
4	11	
5	6	
6	12	
7	6	
Σ	60.0	

1. Aufgabe (12 Punkte): Bäume

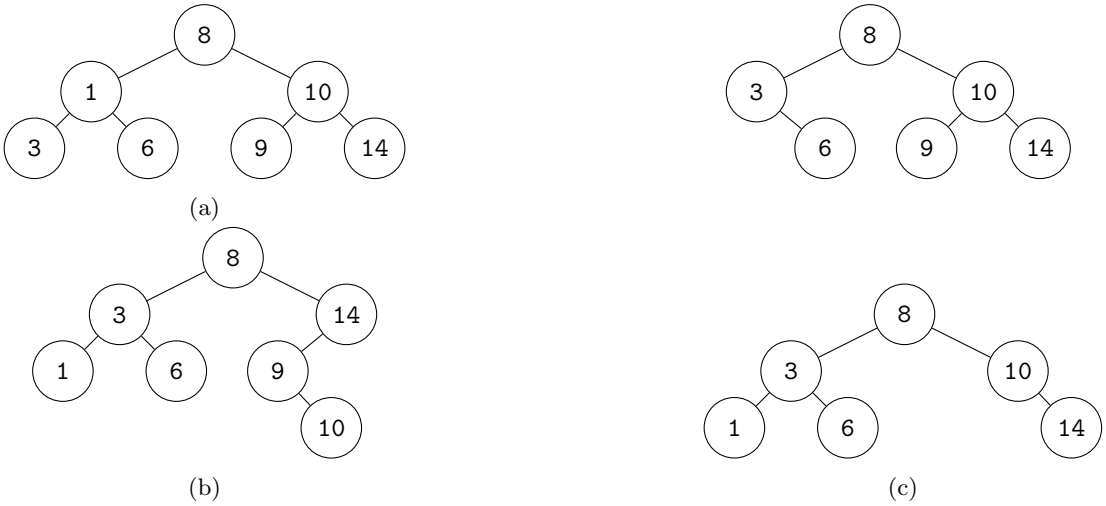


Abbildung 1

1.1. Definition binärer Bäume (4 Punkte)

Welche Bäume sind binäre Suchbäume?

☐ Baum (a)
☐ Baum (b)
☐ Baum (c)
☐ Baum (d)

Welche Bäume sind balanciert?

☐ Baum (a)
☐ Baum (b)
☐ Baum (c)
☐ Baum (d)

Welche Bäume sind linksvoll?

☐ Baum (a)
☐ Baum (b)
☐ Baum (c)
☐ Baum (d)

Welche Bäume sind rechtsvoll?

☐ Baum (a)
☐ Baum (b)
☐ Baum (c)
☐ Baum (d)

1.2. AVL-Bäume (2 Punkte) Angenommen, (c) ist ein AVL-Baum. Zeichne die Bäume, die als Zwischenergebnisse entstehen, wenn wir den Baum balancieren, sowie das Endresultat.

1.3. Binärer Suchbaum (4 Punkte) Was ist der mittlere Zeitaufwand einer Suche in einem binären Suchbaum mit n Knoten?

☐ $O(n)$
☐ $O(log(n))$
☐ $O(n\ log(n))$
☐ $O(n^2)$

Wie hoch ist der Aufwand im worst-case?

☐ $O(n)$
☐ $O(log(n))$
☐ $O(n\ log(n))$
☐ $O(n^2)$

Im Folgenden ist eine Java-Implementierungen für einen binären Suchbaum angegeben, bei dem die Knoten Instanzen der Klasse `BSTNode` sind.

```
public class BinarySearchTree {

        private BSTNode root;

        /*
         * More declarations ...
         */
}

class BSTNode{

    private BSTNode left;
    private BSTNode right;
    private int value;

    BinarySearchTreeNode getLeft() {
        return left;
    }

    BinarySearchTreeNode getRight() {
        return right;
    }

    int getValue() {
        return value;
    }
}
```

Implementiere die Methode `BSTNode find(int value)` der Klasse `BinarySearchTree` **ohne Rekursion zu verwenden**. Die Methode soll den Knoten mit gegebenen Wert zurückgeben oder `null`, falls ein solcher Knoten nicht existiert.

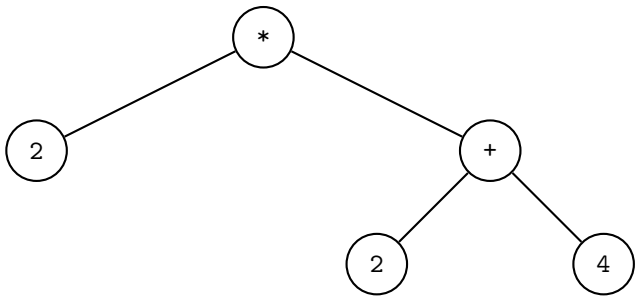
```
public BSTNode find(int value) {




}
```

1.4. Expression Trees (2 Punkte) Bei der Implementierung von Compilern oder Taschenrechnern müssen algebraische Ausdrücke wie z. B. "$2 * (2 + 4)$" gespeichert und ausgewertet werden. Dafür werden sog. Ausdrucksbäume (*expression trees*) verwendet, bei denen Blätter numerische Werte und innere Knoten Operatoren repräsentieren.

Der Ausdruck "$2 * (2 + 4)$" würde beispielsweise durch folgenden Baum dargestellt:



Welcher Baumtraversierung würde die Elemente in der Reihenfolge "$2 * 2 + 4$" wiedergeben?

Welcher Baumtraversierung würde die Elemente in der Reihenfolge "$* \; 2 + 2 \;\; 4$" wiedergeben?

2. Aufgabe (8 Punkte): Sortieren

Gegeben seien folgende zwei Algorithmen:

```
public class SortingAlgorithm1 {
  public int[] sort(int[] array) {
    int n = array.length;

    for(int j = 0; j < n-1; j++) {
      int index = j;

      for(int i = j+1; i < n; i++) {
        if (array[i] < array[index]) {
          index = i;
        }
      }

      if( index != j ) {
        int tmp = array[index];
        array[index] = array[j];
        array[j] = tmp;
      }
    }
    return array;
  }
}
```

```
public class SortingAlgorithm2 {
  public List<int> sort(List<int> list) {
    List<int> newList = new ArrayList<int>(list);
    if (newList.size() <= 1)
      return newList;

    int split = newList.size() / 2;
    List<int> left = sort(newList.subList(0, split));
    List<int> right = sort(newList.subList(split, newList.size()));

    return combine(left, right);
  }

  public List<int> combine(List<int> left, List<int> right) {
    List<int> retval = new ArrayList<T>(left.size() + right.size());
    while (!left.isEmpty() && !right.isEmpty()) {
      if (left.get(0) <= right.get(0)) {
        retval.add(left.remove(0));
      } else {
        retval.add(right.remove(0));
      }
    }
    if (!left.isEmpty())
      retval.addAll(left);
    if (!right.isEmpty())
      retval.addAll(right);

    return retval;
  }
}
```

2.1. Handsimulation (2 Punkte) Wir rufen SortingAlgorithm1 mit folgendem Array auf: $\{3, 7, 2, 1, 8, 0\}$.

Fülle die nachstehende Tabelle aus, indem Du den Zustand der Variablen `array` am Ende der äußeren for-Schleife in jeder Iteration angibst!

Schritt j	Inhalt von `array`
initial	{3, 7, 2, 1, 8, 0}
0	
1	
2	
3	
4	

2.2. Algorithmen identifizieren (2 Punkte) Um welche Sortieralgorithmen handelt es sich?

- SortingAlgorithm1:

- SortingAlgorithm2:

2.3. Aufwand (2 Punkte) Sei n die Anzahl der Elemente in `array` bzw. `list`. Wie lautet die kleinste obere Schranke für die Laufzeit für gegebenen Algorithmen in Abhängigkeit von n? (O-Notation)

- SortingAlgorithm1:

- SortingAlgorithm2:

2.4. In-place Sortieralgorithmen (2 Punkte) Wann ist ein Sortieralgorithmus *in-place*? Antworte in einem Satz.

Handelt es sich bei den gegebenen Algorithmen um *in-place* Sortieralgorithmen?

- SortingAlgorithm1:
  ☐ Ja
  ☐ Nein

- SortingAlgorithm2:
  ☐ Ja
  ☐ Nein

3. Aufgabe (5 Punkte): Graphen I - Scheduling

Fluglotsen weisen einfliegenden Flugzeugen ein Zeitfenster von wenigen Minuten zur Landung zu. Die Landereihenfolge der Flugzeuge hängt dabei von vielen Faktoren ab. Kriterien für eine Priorisierung sind beispielsweise wartende Anschlussflüge bei Verspätung.

Stell Dir vor, Du bist ein angehender Fluglotse und bekommst in einer Prüfung die folgenden beiden Landeanflugsituationen in den Graphen A und B vorgelegt. Die Abhängigkeiten der Flugzeuge (Knoten) untereinander sind durch die gerichteten Pfeile dargestellt.

Hinweis: Der Pfeil von $EJ \rightarrow BA$ bedeutet hier, daß Flugzeug EJ **vor** BA landen muß.

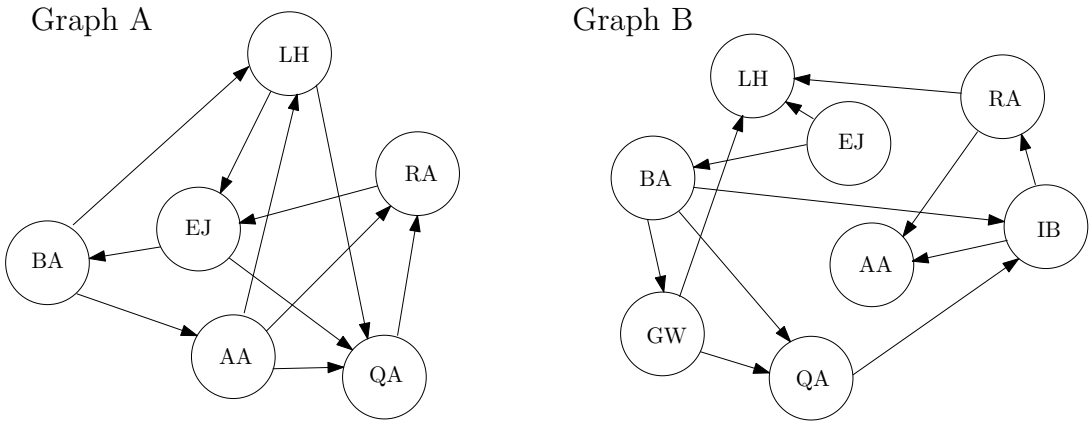


Abbildung 2: Zwei verschiedene Landeanflugssituationen

3.1. Scheduling (2 Punkte) Du musst nun in kürzester Zeit entscheiden, ob es für die beiden Situationen eine Landereihenfolge gibt. Falls ja, gib diese für den jeweiligen Graphen an. Falls nein, begründe kurz, warum es nicht möglich ist.

Gibt es eine mögliche Landereihefolge für **Graph A**?
☐ ja, nämlich:
☐ nein, da:

Gibt es eine mögliche Landereihefolge für **Graph B**?
☐ ja, nämlich:
☐ nein, da:

3.2. Algorithmus (1 Punkt) Welcher Algorithmus kommt hier zum Einsatz?

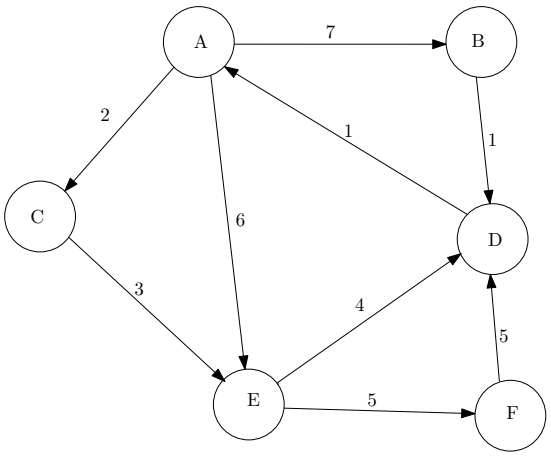
3.3. Abhängigkeiten auflösen (2 Punkte) Sollte es keine Landereihenfolge geben, welches Flugzeug aus welchem Graphen könntest Du auf einen anderen Flughafen umleiten, um doch noch eine Landereihenfolge zu ermöglichen? Durch die Umleitung werden dieses Flugzeug und seine Abhängigkeiten aus dem Graphen entfernt.

Flugzeugname:
Graph:
Landereihenfolge:

Gäbe es noch ein anderes Flugzeug, das in Frage käme?
☐ ja, nämlich
☐ nein, da

4. Aufgabe (11 Punkte): Graphen II - Kürzeste Wege

Gegeben ist folgender gerichteter Graph. Die Gewichte auf den Kanten entsprechen den Entfernungen zwischen zwei Knoten.



4.1. Handsimulation (5 Punkte) Führe eine Handsimulation des **Dijkstra**-Algorithmus aus, um die kürzesten Wege von Knoten A zu allen anderen Knoten zu bestimmen. Nutze dazu die beiden gegebenen Tabellen.

Hinweis: A^0 bedeutet: A hat kürzeste Distanz 0 zu sich selbst.
Trage in der **zweiten** Tabelle für jeden Schritt **nur diejenigen** aktuell kürzesten Distanzen von Knoten ein, die sich **ändern**. Gib für diese Knoten auch jeweils den zugehörigen Vorgängerknoten (nach dem Komma) an.

Schritt	aktueller Knoten	Priority-Queue (sortiert nach aktuell kürzester Distanz zum Startknoten)
0		A^0
1		
2		
3		
4		
5		
6		

	Knoten					
Schritt	A	B	C	D	E	F
0	0, null	∞, null	∞, null	∞, null	∞, null	∞, null
1						
2						
3						
4						
5						
6						

4.2. Kürzeste Wege Effizienz (4 Punkte) Berechnet **Dijkstra** diesen kürzesten Weg am effizientesten? Entscheide, ob die folgenden Algorithmen das Problem effizienter lösen. Begründe kurz Deine Entscheidung.

Floyd-Warshall löst das hier gegebene kürzeste Wege Problem

☐ effizienter als Dijkstra, da

☐ *genauso* effizient wie Dijkstra, da

☐ *nicht* effizienter als Dijkstra, da

☐ gar nicht, da

Bellmann-Ford löst das hier gegebene kürzeste Wege Problem

☐ effizienter als Dijkstra, da

☐ *genauso* effizient wie Dijkstra, da

☐ *nicht* effizienter als Dijkstra, da

☐ gar nicht, da

Kruskal löst das hier gegebene kürzeste Wege Problem

☐ effizienter als Dijkstra, da

☐ *genauso* effizient wie Dijkstra, da

☐ *nicht* effizienter als Dijkstra, da

☐ gar nicht, da

Ford-Fulkerson löst das hier gegebene kürzeste Wege Problem

☐ effizienter als Dijkstra, da

☐ *genauso* effizient wie Dijkstra, da

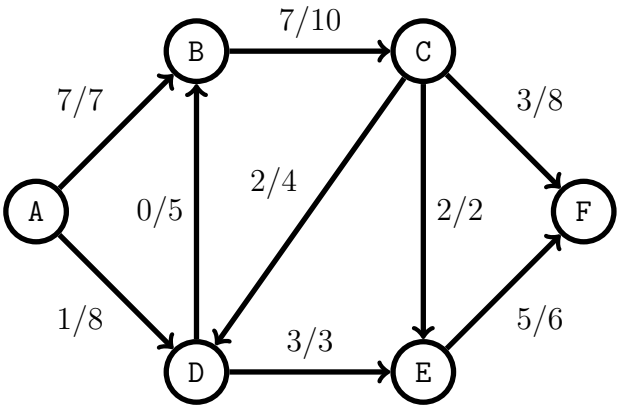
☐ *nicht* effizienter als Dijkstra, da

☐ gar nicht, da

4.3. Negative Kanten (2 Punkte) Wir setzen nun das Gewicht der Kante von Knoten F nach D auf den Wert -5. Warum funktioniert Dijkstra hier und auch im Allgemeinen nicht, wenn ein Graph negative Kantengewichte enthält? Begründe Deine Antwort kurz.

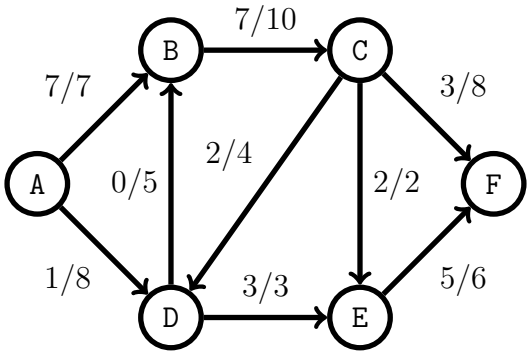
5. Aufgabe (6 Punkte): Flüsse in Netzwerken

Gegeben sei das folgende Flussnetzwerk mit initialem Fluss:

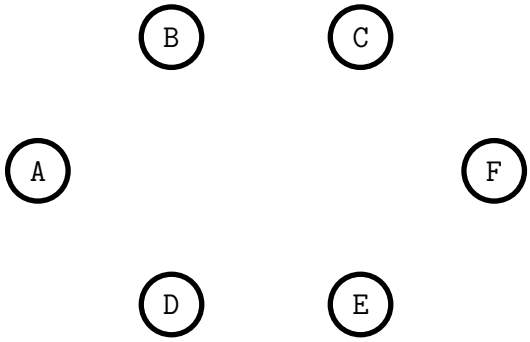


5.1. Maximaler Fluss (5 Punkte) Bestimme jeweils den Restflussgraph, einen Erweiterungspfad und den neuen Fluss, bis ein maximaler Fluss erreicht ist! (Je nach Wahl der Erweiterungspfade benötigst Du weniger Iterationen, als vorgegebene Diagramme vorhanden sind.)

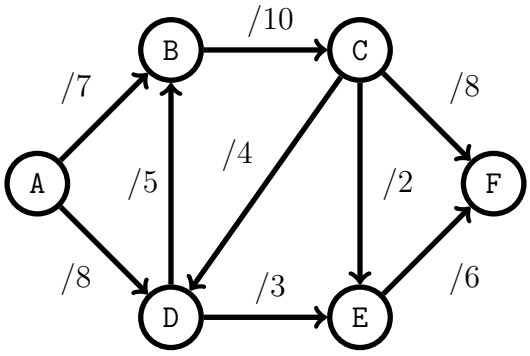
Fluss

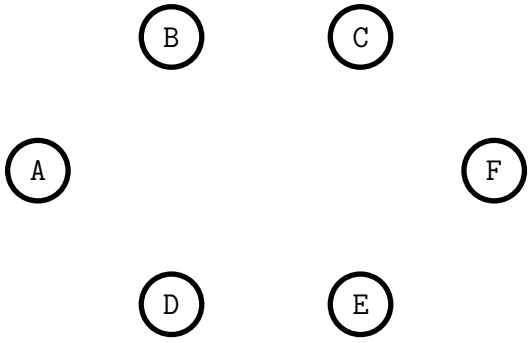


Restflussgraph



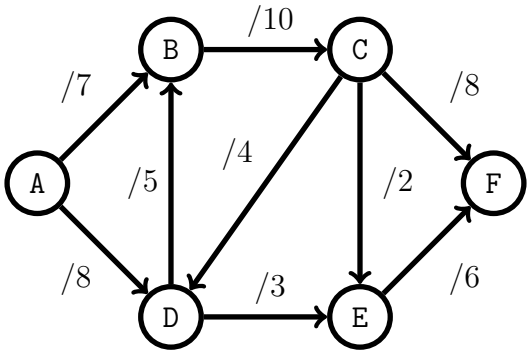
Erweiterungspfad:



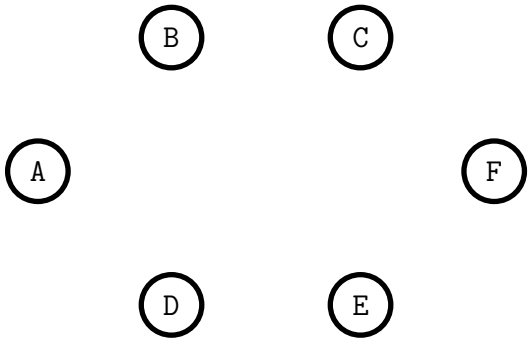


Erweiterungspfad:

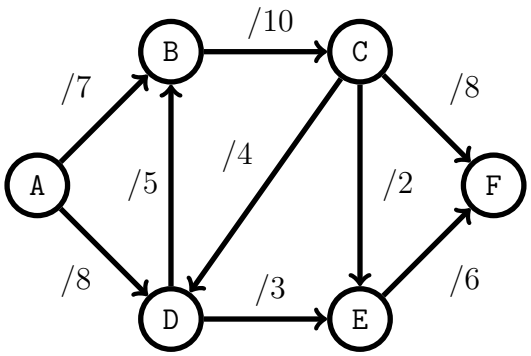
Fluss

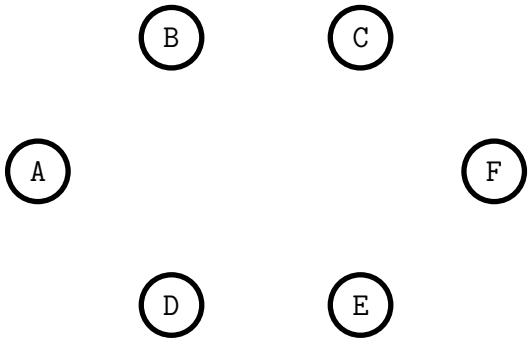


Restflussgraph



Erweiterungspfad:

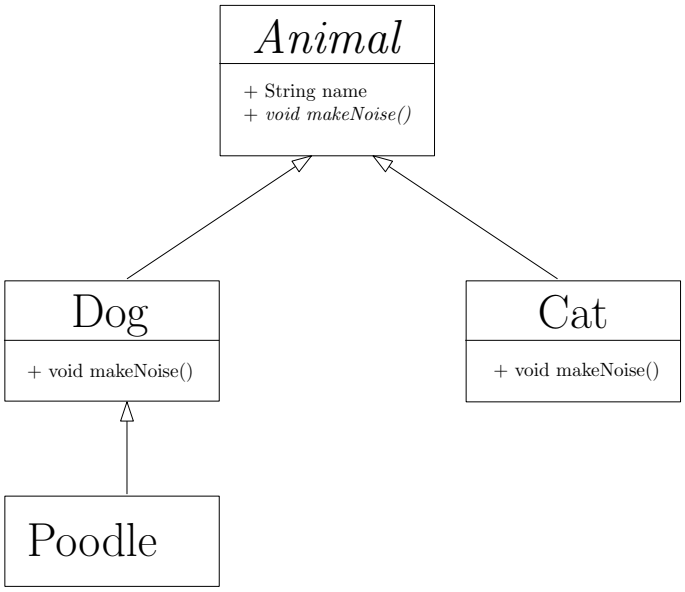




5.2. Minimaler Schnitt (1 Punkt) Die Kapazität(en) welcher Kante(n) muss man mindestens erhöhen, um einen größeren Fluss zu erhalten?

6. Aufgabe (12 Punkte): Objektorientierte Programmierung in Java

Gegeben sei folgende Vererbungshierarchie[1]:



Ein Pfeil von `A` → `B` besagt, dass `A` von `B` erbt. Dabei stellen Elemente, die *kursiv* geschrieben sind, nicht vollständig implementierte Klassen oder Methoden dar. Das + vor einem Element bedeutet, dass das Element `public` ist.

Folgender (unvollständiger) Code ist vorgegeben (Fortsetzung auf der nächsten Seite):

_____ Animal {

```
    public String name;

    _____ void makeNoise();

    public Animal() {
        this.name = "";
    }

    @Override
    public String toString() {
        return name;
    }
}
```

[1] Hinweis: *Poodle* ist das englische Wort für *Pudel*.

Dog {

```
    public Dog(String name) {
        System.out.println("Dog");
        this.name = name;
    }

    @Override
    public void makeNoise() {
        System.out.println("bark");
    }
}
```

Cat {

```
    public Cat(String name) {
        System.out.println("Cat");
        this.name = name;
    }

    @Override
    public void makeNoise() {
        System.out.println("miau");
    }

}
```

Poodle {

```
    Poodle(String name) {
        super(name);
        System.out.println("Poodle");
    }

    @Override
    public void makeNoise() {
        System.out.println("BARK");
    }
}
```

6.1. Code vervollständigen (2 Punkte) Vervollständige den oben angegebenen Quellcode, so dass die gegebene Vererbungshierarchie implementiert wird.

Ergänze dazu die fehlenden Schlüsselwörter und Klassennamen an den gekennzeichneten Stellen.

6.2. Interfaces vs. abstrakte Klassen (3 Punkte) Nenne drei Unterschiede zwischen Interfaces und abstrakten Klassen in Java!

6.3. Objekte und Methoden (2 Punkte) Gib die Ausgabe der folgenden `main`-Methode an:

```
1 public static void main(String[] args) {
2         Dog d = new Poodle("Mephisto");
3         d.makeNoise();
4 }
```

6.4. Fehler finden (3 Punkte) In folgender `main`-Methode befinden sich einige Fehler, über welcher sich der Java-Compiler beschweren würde.

Benenne jede Zeile, in welcher ein Fehler auftritt, und beschreibe in *einem kurzen* Satz, warum der Fehler auftritt.

Hinweis: Es ist nicht nach Compiler-*Warnings*, sondern nur nach Compiler-*Errors* gefragt.

```
1 public static void main(String[] args) {
2         Animal a1 = new Animal();
3         Animal a2 = new Dog("Bello");
4         Dog d2 = a2;
5         Cat c3 = new Cat("Snowball");
6         Animal a3 = c3;
7         Dog d4 = new Dog();
8 }
```

6.5. Call by reference vs. call by value (2 Punkte) Gegeben ist folgendes Code-Fragment:

```
1  public static void method(Dog d1, Dog d2) {
2          Dog tmp = d1;
3          d1 = d2;
4          d2 = tmp;
5  }
6
7  public static void main(String[] args) {
8          Dog d1 = new Dog("Bello");
9          Dog d2 = new Dog("Snowball");
10
11         method(d1, d2);
12
13         // Ausgabe ab hier:
14         System.out.println(d1);
15         System.out.println(d2);
16 }
```

Was ist die Ausgabe des Code-Fragments ab Zeile 13?

7. Aufgabe (6 Punkte): Datenstrukturen

Eine Zeichenkette kann mittels unterschiedlicher Datenstrukturen repräsentiert werden. Im Folgenden sind zwei mögliche Implementierungen vorgegeben, die wir vergleichen möchten:

```
class StringVersion1 {
  char[] data;

  public StringVersion1(String input) {
    data = new char[input.length()];

    for (int i = 0; i < input.length(); i++)
      data[i] = input.charAt(i);
  }

  // more methods...
}
```

```
class StringVersion2 {
  class Node {
    char data;
    Node successor;

    Node(char data, Node successor) {
      this.data = data;
      this.successor = successor;
    }
  }

  Node head, tail;

  public StringVersion2(String input) {
    for (int i = 0; i < input.length(); i++)
      addCharacter(input.charAt(i));
  }

  public void addCharacter(char c) {
    Node character = new Node(c, null);
    if (tail != null)
      tail.successor = character;

    tail = character;

    if (head == null)
      head = character;
  }

  // more methods...
}
```

7.1. Datenstrukturen identifizieren (1 Punkt) Welche uns bekannten Datenstrukturen wurden jeweils verwendet, um StringVersion1 bzw. StringVersion2 zu implementieren?

StringVersion1:

StringVersion2:

7.2. Datenstrukturen vergleichen (5 Punkte) Gib für jede der folgenden Zeichenkettenoperationen an, ob StringVersion1 oder StringVersion2 diese effizienter umsetzen kann.

Wir definieren den Aufwand in Abhängigkeit der Länge einer Zeichenkette. Die Funktionen $f_{\text{StringVersion1}}$ bzw. $f_{\text{StringVersion2}}$ stellen den jeweils benötigte Aufwand der beiden Implementierungen dar.

Zur Erinnerung:

$$\mathcal{O}(f) = \{g \mid \exists\, c > 0\ \exists\, x_0 > 0\ \forall x > x_0 : |g(x)| < c \cdot |f(x)|\}$$

Zwei Zeichenketten aneinanderhängen:

`void concatenate(StringVersion1 str)` bzw. `void concatenate(StringVersion2 str)`

☐ $\mathcal{O}(f_{\text{StringVersion1}}) \subsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) \supsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) = \mathcal{O}(f_{\text{StringVersion2}})$

Den Index des ersten Vorkommens eines Zeichens c innerhalb der Zeichenkette zurückgeben:

`int indexOf(char c)`

☐ $\mathcal{O}(f_{\text{StringVersion1}}) \subsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) \supsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) = \mathcal{O}(f_{\text{StringVersion2}})$

Die Teilsequenz zwischen dem i-ten und dem (i+3)-ten Zeichen ausgeben:

`void printThreeCharacterSubstring(int i)`

☐ $\mathcal{O}(f_{\text{StringVersion1}}) \subsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) \supsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) = \mathcal{O}(f_{\text{StringVersion2}})$

Ein einzelnes Zeichen in der ersten Hälfte der Zeichenkette löschen:

`void eraseAtIndex(int i) mit i < (Länge der Zeichenkette / 2)`

☐ $\mathcal{O}(f_{\text{StringVersion1}}) \subsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) \supsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) = \mathcal{O}(f_{\text{StringVersion2}})$

Ein einzelnes Zeichen in der zweiten Hälfte der Zeichenkette löschen:

`void eraseAtIndex(int i) mit i > (Länge der Zeichenkette / 2)`

☐ $\mathcal{O}(f_{\text{StringVersion1}}) \subsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) \supsetneq \mathcal{O}(f_{\text{StringVersion2}})$
☐ $\mathcal{O}(f_{\text{StringVersion1}}) = \mathcal{O}(f_{\text{StringVersion2}})$