# Datenstrukturen
# Übung – SS2020

**Abgabe:** 19.07. bis 24Uhr.
**Abgabeort:** Bitte laden Sie Ihre Klasse im OPAL-Bereich "Abgabe_Pflichtaufgabe_11" hoch. Ergänzen Sie bei Ihrer hochgeladenen Klasse die Namen (Vor- und Nachname) aller beteiligten Personen. Sie können Ihre Abgabe bis zur Abgabefrist beliebig oft löschen und erneut einreichen.

**Wichtig:** Da es sich um eine Prüfungsvorleistung handelt, ist es wichtig, dass jeder Teilnehmer Zugang zu allen notwendigen Informationen hat. Deswegen bitten wir Sie Fragen direkt ins Forum, in den Thread "Fragen zur Pflichtaufgabe 11" zu stellen.

Packen Sie Ihre Klasse in das Package "pvl11_$groupX". Sollten Sie nicht in einer Gruppe arbeiten, wählen Sie den Namen statt "pvl11_$Nachname_$Vorname".
Ihre Klasse soll den Namen "PVL11_$GroupX" bzw. "PVL11_$Nachname_$Vorname" besitzen.

Für die Abgabe: Packen Sie all Ihre Klassen in ein .zip Archiv mit dem Namen "PVL11_GroupX.zip"
[Anmerkung: .rar ist kein .zip]

## Prüfungsvorleistung 11 – Graphen V

Implementieren Sie einen gewichteten, gerichteten Graph. Ein Graph sei dabei ein Tupel von Knoten V und Kanten E : G=(V,E). Der Graph soll dabei N Knoten besitzen, indiziert von 0 bis N-1. Eine Kante soll eine nutzbare Verbindung zwischen paarweisen disjunkten Knoten sein.

Die folgenden Methoden finden Sie im Interface "**Graph_V**", hier finden Sie eine Spezifikation für das Verhalten der Methoden, Beispiele folgen weiter unten. Weiterhin sei eine minimale Implementierung "**Edge**" gegeben, diese wird für die Schnittstelle der Methoden genutzt. Verändern Sie diese Implementierung nicht. Es steht Ihnen jedoch frei intern eine andere Implementierung zu nutzen.

**Java**: public PVL9_$groupX(int n)
**Python**: __init__(self, n: int)

Kreieren Sie einen Graphen, welcher "n" Knoten besitzt. Es soll keine Kante vorhanden sein.

**Java**: Boolean setEdge(int source, int destin, int weight)
**Python**: def setEdge(self, source: int, destin : int, weight : int) -> bool

Erstellen Sie eine neue Kante von "source" nach "destin". Geben Sie "true" zurück falls diese Kante neu erstellt wurde. Existiere diese Kante bereits überschreiben Sie das Gewicht dieser Kante und geben Sie "true" zurück. Solle ("from","to") kein valides Tuple sein, geben Sie "false" zurück. (Achtung **gerichteter** Graph)

**Java**: List<List<Edge>> getEdges()
**Python**: def getEdges(self) -> List[List[Edge]]

Geben Sie eine Liste von Listen zurück. Diese i-te Liste soll dabei angeben, welche Kanten vom i-ten Knoten ausgehen.

**Java**: List<List<Edge>> getAllPaths(int source, int destin)
**Python**: def getAllPaths(self, source: int, destin : int) -> List[List[Edge]]

Geben Sie eine Liste von Listen zurück. Jede Liste soll einen **Weg** von “source“ nach “destin“ repräsentieren. Es gibt keine weiteren Einschränkungen für diese Wege.

**Java**: Boolean hasEulerianCircle()
**Python**: def hasEulerianCircle(self) -> bool

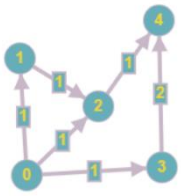
Geben Sie eine Boolschen Wert zurück. Geben Sie “True“ zurück falls es einen Eulerkreis gibt, geben Sie “False“ zurück, wenn es keinen Eulerkreis gibt. Ein Kreis ist ein Eulerkreis, wenn er alle Kanten des Graphen genau einmal enthält.

**Java**: List<List<Integer>> allPairShortestPath()
**Python**: def allPairShortestPath(self) -> List[List[Integer]]

Geben Sie eine 2D Matrix zurück, welche an Stelle [i][j] eine Zahl enthält, die die Länge des kürzesten Weges von i nach j beschreibt. Benutzen Sie den Dijkstra **nicht**. Diese Methode wird nur auf Graphen mit nicht negativen Kantengewichten aufgerufen. Sollte es einen Weg nicht geben, setzten Sie den Eintrag auf -1. Der Eintrag zu sich selbst, soll stets 0 sein.

graph_I

graph_II



```
Graph_V graph_I = PVL11_$GroupX(6);
graph_I.setEdge(0,1,1);
graph_I.setEdge(1,2,1);
graph_I.setEdge(2,3,1);
graph_I.setEdge(3,4,1);
graph_I.setEdge(4,1,1);
graph_I.setEdge(1,3,1);
graph_I.setEdge(3,0,1);
graph_I.setEdge(0,5,1);
graph_I.setEdge(5,2,1);
graph_I.setEdge(2,0,1);

graph_I.hasEulerianCircle(); //true

Graph_V graph_II = PVL11_$GroupX(5);
graph_II.setEdge(0,1,1);
graph_II.setEdge(0,2,1);
graph_II.setEdge(0,3,1);
graph_II.setEdge(1,2,1);
graph_II.setEdge(2,4,1);
graph_II.setEdge(3,4,2);

graph_II.getAllPaths(0,4);
//{
// {Edge(1,1), Edge(2,1), Edge(4,1)},
// {Edge(2,1), Edge(4,1)},
// {Edge(3,1), Edge(4,2)},
// }

graph_II.allPairShortestPath();
//{
// {0,1,1,1,2},
// {-1,0,1,-1,2},
// {-1,-1,0,-1,1},
// {-1,-1,-1,0,2},
// {-1,-1,-1,-1,0},
// }
```