Adjazenzlisten implementieren

Mit Hilfe der folgenden Anleitung sollen Sie erfahren, wie sich eine Java-Klasse zum Speichern von Graphen implementieren lässt.

Graphen als Abstrakter Datentyp

Folgendes Interface zeigt einen Abstrakten Datentypen `Graph` mit den notwendigen Operationen:

```
public interface Graph<K> {
 public boolean addVertex(K vertex);
 public boolean addEdge(K from, K to);
 public boolean removeEdge(K from, K to);
 public boolean isDirected();

 public int getNofVertices();
 public int getNofEdges();

 public Set<K> getVertices();
 public Set<K> getAdjacentVertices(K vertex);
 public Object clone();
}
```

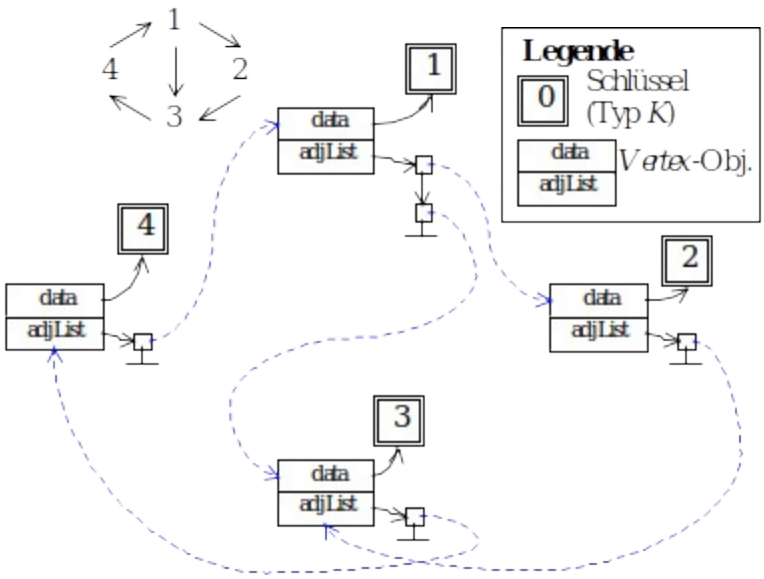
Konzept für Implementierung mit Adjazenzlisten

Für die meisten Algorithmen, die in diesem Kurs betrachtet werden, sind Adjazenzlisten eine gute Wahl. Deswegen soll dieses Konzept in der Implementierung umgesetzt werden. Zur internen Repräsentation der Graphen-Knoten werden Objekte einer Klasse `Vertex` verwendet:

```
private static class Vertex<K> {
 K data;
 List<Vertex<K>> adjList = new LinkedList<Vertex<K>>();
 ...
}
```

Die `Vertex`-Objekte enthalten also jeweils eine Referenz auf das Schlüsselobjekt zur Identifikation des Knotens, den sie repräsentieren, sowie eine Liste mit Referenzen auf die `Vertex`-Objekte aller Knoten, zu denen eine Kante führt.

Nebenstehende Abbildung zeigt einen Graphen mit 4 Knoten und 5 Kanten sowie dessen Repräsentation im Speicher.

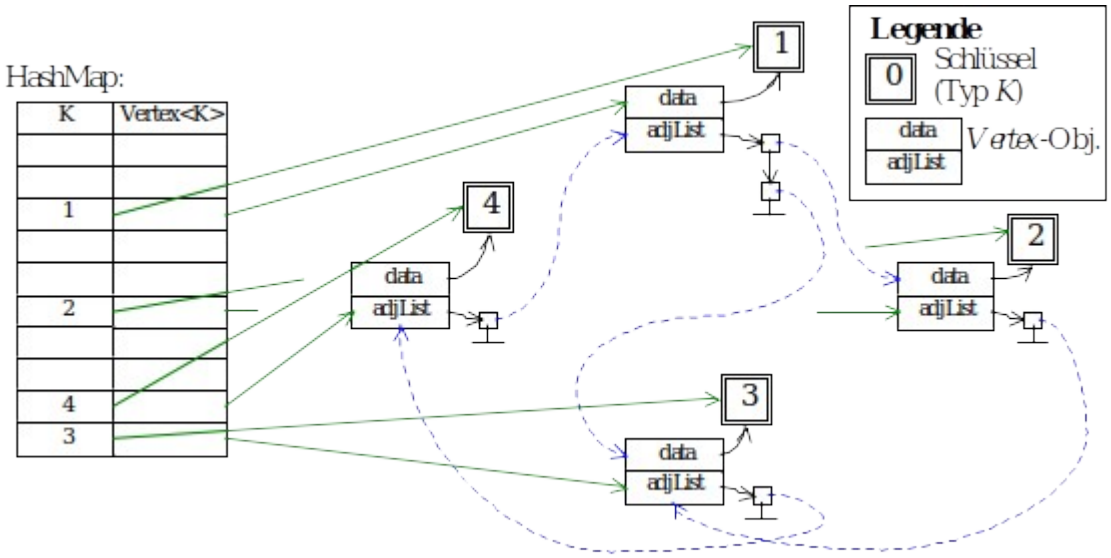


Bemerkung: In den Schnittstellen werden nicht die Objekte vom Typ `Vertex` übergeben, sondern die von diesen referenzierten Schlüsselobjekte vom Typ K. Referenzen auf `Vertex`-Objekte dürfen ausserhalb der Graphen-Klasse selbst nicht sichtbar werden.

Das bedeutet aber auch, dass es eine effiziente Möglichkeit geben muss, zu einem beispielsweise an `addEdge` übergebenen Schlüsselobjekt das zugehörige `Vertex`-Objekt im Graphen zu finden. Zu diesem Zweck empfiehlt sich der Einsatz einer `HashMap<K, Vertex<K>>`[1]. Wie die Typenparametrisierung bereits andeutet, werden dabei die Objekte vom Typ K als Schlüssel eingesetzt, um dann mittels der `get`-Methode aus dem `Map`-Interface[2] jeweils das entsprechende `Vertex`-Objekt zu erhalten.

[1] siehe Java-Dokumentation zu `java.util.HashMap`

[2] siehe Java-Dokumentation zu `java.util.Map`



Vorbereitung zur Implementation

Auf dem File-Server finden Sie ein Programmier-Projekt `ch.fhnw.algd2.graphedit` mit einem Graphen-Editor. Diesen können Sie starten, aber Sie können nichts sinnvolles damit tun, da die darin verwendete Klasse `AdjListGraph` noch nicht vollständig programmiert ist.

Die Klasse implementiert das Interface `Graph` und erweitert die Klasse `AbstractGraph`. Diese Basisklasse implementiert folgende Methoden:

- `toString`: Ausgabe des Graphen in einen String
- `getNofVertices`: Anzahl der Knoten
- `getNofEdges`: Anzahl der Kanten
- `isDirected`: Abfrage, ob der Graph gerichtet oder ungerichtet ist. Diese Eigenschaft wird mit dem Konstruktor gesetzt.

Die beiden Methoden `getNofVertices` und `getNofEdges` sind aber ziemlich ineffizient implementiert. Sie sollten sie zu gegebener Zeit in Ihrer Implementierung der Klasse `AdjListGraph` mit effizienteren Implementationen überschreiben.

Zum Testen Ihrer Implementation steht ein Graphen-Editor zur Verfügung. Damit dieser Editor überhaupt funktioniert muss er wissen, wie der Graph beschaffen ist. Dazu verwendet er Ihre (zu erstellende) Klasse `AdjListGraph`. Den Editor benötigen Sie auch später noch zur Implementierung von Graphen-Algorithmen.

Eine Benutzungsanleitung finden Sie im Anschluss an die Aufgaben.

Aufgabe 1: Graphenklasse implementieren

Vervollständigen Sie die Klasse `AdjListGraph`, die Graphen mit Hilfe von Verbindungslisten verwaltet.

- Beginnen Sie mit der Methode `addVertex`. Wenn diese richtig funktioniert, dann sollten Sie die Knoten im Editor sehen können.
- Implementieren Sie danach auch noch `addEdge` und `getAdjacentVertices`. Jetzt sollten auch die Kanten sichtbar werden. Zeichnen Sie einige Graphen und vergewissern Sie sich, dass Ihre Implementationen korrekt sind.
- Implementieren Sie nun `removeEdge` und optimieren Sie `getNofVertices` und `getNofEdges`.
- (*herausfordernd*) Programmieren Sie folgenden Konstruktor:

```
public AdjListGraph(AdjListGraph<K> orig) // copy constructor
```

  Dieser Konstruktor erhält einen bestehenden Graphen der gleichen Klasse und baut eine Kopie davon auf. Solche Kontruktoren werden auch *copy constructor* genannt. Testen Sie diesen Konstruktor indem Sie auf den *Copy*-Knopf im Graphen-Editor drücken. Dabei wird die Methode `AdjListGraph.clone()` aufgerufen und so eine Kopie des Graphen erzeugt und dann dargestellt. Da der Graph-Editor einfach den bisherigen Graphen durch den neuen ersetzt, sollten Sie bei korrekter Implementierung keinen Unterschied zwischen “vorher” und “nachher” sehen!

Aufgabe 2: Topologisches Sortieren implementieren

Implementieren Sie in Ihrer Klasse `AdjListGraph` zusätzlich das Interface `Graph-Algorithms.TopSort`. Die in diesem Interface definierte Methode `void sort()` hat weder Parameter noch einen Rückgabewert. Das Resultat soll auf der Konsole ausgegeben werden.

Dieses Resultat soll entweder eine topologische Reihenfolge des Graphen sein oder eine Fehlermeldung (z.B. ob Zyklen vorkommen oder ob es sich nicht um einen gerichteten Graphen handelt). Programmieren Sie diese Methode mit Aufwand in $O(|V| + |E|)$.

Welche zusätzlichen Attribute in welcher/welchen Klasse/n sind notwendig? Erweitern Sie die Datenstrukturen möglichst sparsam. Wenn Sie neue Methoden einführen, deklarieren Sie diese als `private`.

Zur Berechnung des Eingangsgrades können Sie entweder die `addEdge`- und `removeEdge`-Methoden ergänzen, oder zu Beginn von `sort` über alle Knoten den Eingangsgrad der Nachbarn nachführen. Für welche Variante haben Sie sich entschieden? Warum? Gibt es Unterschiede in der Effizienz?

Benutzung des Editors

Starten Sie den Graphen-Editor, wahlweise mit dem Gradle Task `run` oder indem Sie die direkt die Klasse `ch.fhnw.algd2.graphedit.GraphEditor` starten.

An der rechten Seite sind die wichtigsten Bedienelemente angeordnet. Es gibt 4 verschiedene Modi in denen sich der Editor befinden kann.

1. *Selection*
   Wenn Sie einige Knoten (und Kanten) gezeichnet haben, können Sie in diesem Modus die Knoten selektieren und verschieben. Die Selektion von Knoten ist erst für spätere Aufgaben relevant.
2. *Insert Node*
   Sie können in diesem Modus Knoten durch einfachen Mausklick hinzufügen. Mit jedem Mausklick wird ein neuer Knoten erzeugt. Ist zugleich die Funktion “Auto Name” eingeschaltet, dann wird jeder Knoten automatisch einen Namen erhalten (eine fortlaufende Nummer). Schalten Sie diese Option aus, dann werden Sie nach jedem Mausklick nach einem Namen gefragt.
3. *Insert Edge*
   In diesem Modus können Sie nun Kanten einfügen, indem Sie zuerst auf den Ausgangsknoten und dann auf den Zielknoten klicken. Der Ausgangsknoten wird markiert solange der Zielknoten noch nicht angewählt wurde.
4. *Remove Edge*
   Funktioniert analog zum Insert Edge Modus, nur wird hier die Verbindung zwischen den Knoten gelöscht. Es wird dabei nicht geprüft, ob eine Verbindung bestanden hat oder nicht.

Mit *Print* können Sie den Graphen auf der Konsole ausgeben lassen (dient hauptsächlich zur Überprüfung des angezeigten Graphen)

Im *File*-Menu können Sie unter *New Graph* zwischen vier verschiedenen neuen Graphenarten wählen. Um welche Art Graph es sich gerade handelt, wird auch unten rechts in der Statuszeile angezeigt.

Mit *Save...* und *Open...* können Sie einen Graphen speichern, bzw. wieder laden.

Warnung!

Beim Graphen-Editor handelt es sich um eine einfache Implementation. Viele Funktionen sind nicht “wasserdicht” implementiert, d.h. es werden nicht alle Eventualitäten abgefangen und behandelt. Besondere Vorsicht ist geboten im Umgang mit dem “Auto Name”. Es wird ein simpler Zähler hochgezählt. Nach dem Selberbenennen kann es zu Duplikaten kommen (die von Ihrer Graphklasse ignoriert werden sollten). Bei der Eingabe von Kantengewichten wird nicht überprüft ob es sich tatsächlich um Zahlen handelt.