Name: _____

Abiturprüfung 2020

Informatik, Leistungskurs

Aufgabenstellung:

Informatikstudentin Charlotte Jabelonski arbeitet regelmäßig als „DJ Charlski“. Für DJs ist es eine Herausforderung, die einzelnen Musikstücke aus ihrer Sammlung jeden Abend neu so anzuordnen, dass sie passend aufeinander folgend abgespielt werden.

DJ Charlski möchte eine Software für DJs entwickeln, die dabei hilft, schnell passende Nachfolgestücke zu finden. Dazu werden alle Musikstücke in einem Graphen gespeichert. Eine Kante des Graphen entspricht einem direkten *Übergang* von einem zu einem anderen Musikstück und bestimmt durch ihr Kantengewicht die *Güte* dieses Übergangs.

- Güte 3: optimaler Übergang
- Güte 2: guter Übergang
- Güte 1: schlechter Übergang (d. h. nur in Ausnahmefällen zu benutzen)

Wenn Musikstücke gar nicht durch eine Kante verbunden sind, dürfen sie nie direkt hintereinander gespielt werden. Zusätzlich gilt, dass jedes Musikstück an einem Abend nur einmal gespielt werden darf.

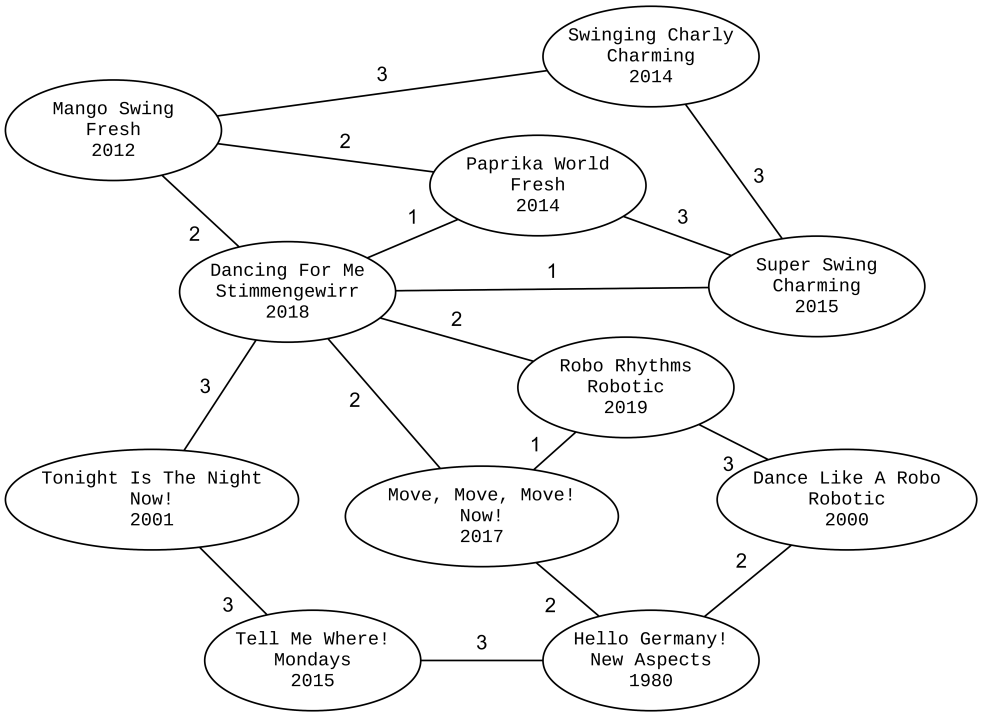


Abbildung 1: Musiksammlung als Graph

Name: _____

Abbildung 1 zeigt die Musiksammlung von DJ Charlski als Graph. Die Knoten des Graphen entsprechen jeweils einem Musikstück der Musiksammlung und beinhalten von oben nach unten den Titel, den Interpreten und das Erscheinungsjahr des Musikstücks. Aus Gründen der Vereinfachung kann davon ausgegangen werden, dass jedes Musikstück im Graphen durch seinen Titel eindeutig identifizierbar ist.

Mithilfe des Graphen kann DJ Charlski nun auch *Übergangspfade* zwischen zwei beliebigen Musikstücken herausfinden. Ein Übergangspfad zwischen zwei Musikstücken entspricht dabei einem Pfad innerhalb des Graphen, der die beiden Musikstücke miteinander verbindet. Dabei bestimmt das kleinste Kantengewicht des Pfades die Güte des Übergangspfades. Besteht ein Pfad z. B. aus drei Kanten mit den Gewichten 2, 3 und 3, so handelt es sich um einen guten Übergangspfad mit der Güte 2.

a) DJ Charlski sucht nach einem optimalen Übergangspfad (Güte 3) vom Titel `"Mango Swing"` zu `"Paprika World"`, wobei jeder Titel an einem Abend höchstens einmal abgespielt werden darf.

Geben Sie einen optimalen Übergangspfad (Güte 3) zwischen den Musikstücken mit den Titeln `"Mango Swing"` *und* `"Paprika World"` *an.*

Erläutern Sie im Sachzusammenhang, welche Auswirkungen es hat, wenn der Titel `"Dancing For Me"` *an einem Abend als erstes abgespielt wird.*

(7 Punkte)

Zur Modellierung eines Prototyps ihrer DJ-Software hat DJ Charlski ein Implementationsdiagramm erstellt, von dem ein Ausschnitt in Abbildung 2 dargestellt ist.

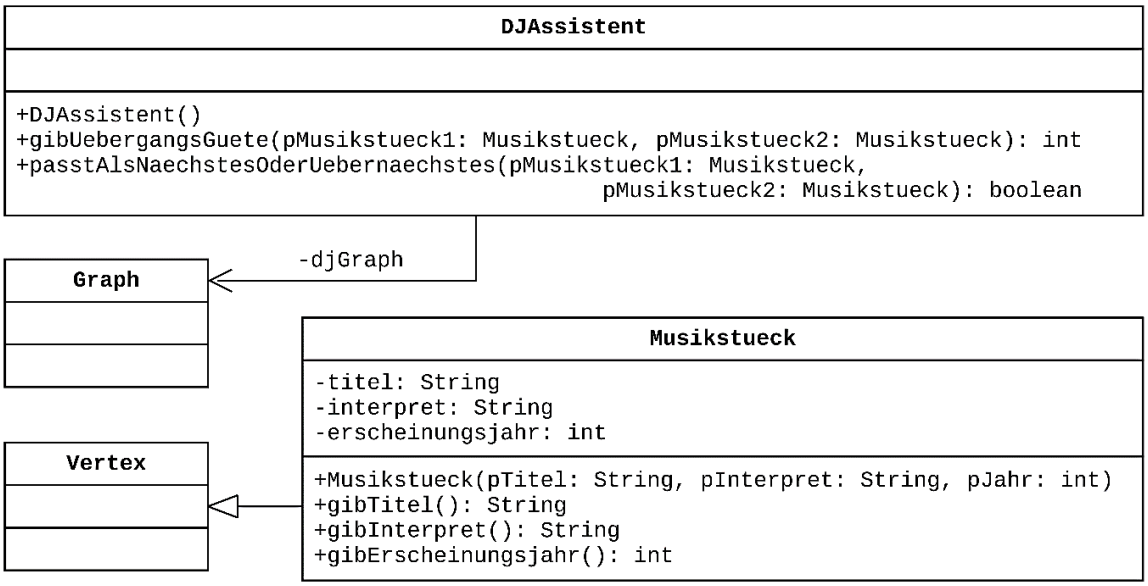


Abbildung 2: Teilmodellierung der DJ-Software in Form eines Implementationsdiagramms

Name: _____

b) *Erläutern Sie die Beziehungen zwischen den Klassen* `DJAssistent`, `Musikstueck`, `Graph` *und* `Vertex`.

Begründen Sie, dass es nicht sinnvoll ist, die Klasse `DJAssistent` *als Unterklasse der Klasse* `Graph` *zu modellieren.*

(8 Punkte)

c) DJ Charlski wird bei ihrer Arbeit häufig gefragt, ob sie als nächstes oder übernächstes ein bestimmtes Musikstück spielen kann. Wenn es einen mindestens guten Übergangspfad (Güte 2 oder 3) vom aktuellen zum gewünschten Musikstück gibt, kommt sie diesen Wünschen gerne nach.

Um solche Fragen schneller beantworten zu können, hat DJ Charlski bereits eine Methode `gibUebergangsGuete` implementiert, welche die Güte des direkten Übergangs zwischen zwei `Musikstueck`-Objekten als Wert zwischen 1 und 3 zurückgibt. Außerdem plant sie eine weitere Methode `passtAlsNaechstesOderUebernaechstes` zu implementieren. Die Dokumentationen beider Methoden befinden sich im Anhang.

Im Folgenden referenziere die Variable `musikstueck1` das `Musikstueck`-Objekt zum Titel `"Dancing For Me"` und die Variable `musikstueck2` das Objekt zum Titel `"Paprika World"` aus dem in Abbildung 1 dargestellten Beispielgraphen.

Geben Sie an, welche Werte folgende Methodenaufrufe für den in Abbildung 1 gegebenen Beispielgraphen zurückgeben:

```
gibUebergangsGuete(musikstueck1, musikstueck2)
```

```
passtAlsNaechstesOderUebernaechstes(musikstueck1, musikstueck2)
```

Implementieren Sie nur die Methode `passtAlsNaechstesOderUebernaechstes`.

Hinweis: Die Methode `gibUebergangsGuete` darf genutzt werden, sie muss aber nicht implementiert werden.

(11 Punkte)

Name: _____

d) DJ Charlski hat die Methode `wasLiefereIch` der Klasse `DJAssistent` implementiert:

```
 1 public boolean wasLiefereIch(Musikstueck pMusikstueck1,
 2                               Musikstueck pMusikstueck2) {
 3   Queue<Vertex> schlange = new Queue<Vertex>();
 4   djGraph.setAllVertexMarks(false);
 5   pMusikstueck1.setMark(true);
 6   schlange.enqueue(pMusikstueck1);
 7   while (!schlange.isEmpty()
 8                     && schlange.front() != pMusikstueck2) {
 9     Vertex a = schlange.front();
10     List<Vertex> liste = djGraph.getNeighbours(a);
11     liste.toFirst();
12     while (liste.hasAccess()) {
13       Vertex b = liste.getContent();
14       Edge e = djGraph.getEdge(a, b);
15       if (e.getWeight() == 3 && !b.isMarked()) {
16         b.setMark(true);
17         schlange.enqueue(b);
18       }
19       liste.next();
20     }
21     schlange.dequeue();
22   }
23   return (!schlange.isEmpty()
24                      && schlange.front() == pMusikstueck2);
25 }
```

Die Methode `wasLiefereIch` werde mit dem in Abbildung 1 dargestellten Graphen und mit den Parametern `musikstueck1` und `musikstueck2` aufgerufen. Dabei referenziere `musikstueck1` das `Musikstueck`-Objekt zum Titel `"Dancing For Me"` und `musikstueck2` das Objekt zum Titel `"Hello Germany!"` aus dem in Abbildung 1 dargestellten Beispielgraphen.

Analysieren Sie die Methode `wasLiefereIch` *und erläutern Sie den Quelltext unter Bezugnahme auf die Veränderungen der Schlange bei Anwendung der Methode auf die Beispieldaten.*

Ermitteln Sie den Rückgabewert der Methode bei Anwendung auf die Beispieldaten.

Erläutern Sie die Bedeutung der Knotenmarkierungen (Quelltextzeilen 5, 15 und 16) im Hinblick auf die Funktionsweise des Algorithmus.

Erläutern Sie die Funktionalität der Methode `wasLiefereIch` *im Sachzusammenhang.*

(12 Punkte)

Name: _____

e) DJ Charlski stellt die Software einem Kollegen vor. Dieser ist begeistert, allerdings merkt er an, dass die Verwaltung der Daten bei einem sehr großen Datenbestand eventuell umständlich wäre. Er hat gehört, dass sich dafür die Verwendung einer Datenbank besser eignet.

Entwickeln Sie eine Datenbankmodellierung in Form eines Entity-Relationship-Diagramms, welche die Speicherung aller Musikstücke und deren Beziehungen zueinander ermöglicht.

Erläutern Sie, wie Ihre Modellierung die Speicherung aller Musikstücke und deren Beziehungen zueinander ermöglicht.

Entwickeln Sie eine Idee, wie eine Datenbankabfrage aussehen könnte, die auf Grundlage Ihrer Datenbankmodellierung dieselbe Information wie die Methode `passtAlsNaechstesOderUebernaechstes` *ermittelt.*

Hinweis: Die Datenbankabfrage selbst muss nicht angegeben werden.

(12 Punkte)

Zugelassene Hilfsmittel:

- GTR (grafikfähiger Taschenrechner) oder CAS (Computer-Algebra-System)
- Wörterbuch zur deutschen Rechtschreibung

Name: _____

Anhang

Die Klasse `Musikstueck`

Diese Klasse ist eine Unterklasse der Klasse `Vertex` (siehe weitere Dokumentationen) und stellt zusätzlich zu den geerbten Methoden die folgenden Methoden zur Verfügung:

Auszug aus der Dokumentation der Klasse `Musikstueck`

Konstruktor **`Musikstueck(String pTitel, String pInterpret, int pJahr)`**
Ein Musikstück mit dem Titel `pTitel`, dem Interpreten `pInterpret` und dem Erscheinungsjahr `pJahr` wird initialisiert. Da der Titel eines Musikstücks dieses eindeutig identifiziert, wird der ID-Eintrag der Oberklasse `Vertex` auf `pTitel` gesetzt.

Anfrage **`String gibTitel()`**
Die Methode gibt den eindeutigen Titel des Musikstücks zurück.

Anfrage **`String gibInterpret()`**
Die Methode gibt den Interpreten des Musikstücks zurück.

Anfrage **`int gibErscheinungsjahr()`**
Die Methode gibt das Erscheinungsjahr des Musikstücks zurück.

Name: _____

Die Klasse `DJAssistent`

Objekte dieser Klasse verwalten einen Graphen mit Objekten der Klasse `Musikstueck` und ihren Beziehungen untereinander. Die Klasse stellt unter anderem folgende Methoden zur Verfügung:

Auszug aus der Dokumentation der Klasse `DJAssistent`

Konstruktor **`DJAssistent()`**
Ein Objekt der Klasse `DJAssistent` wird initialisiert.

Anfrage **`int gibUebergangsGuete(Musikstueck pMusikstueck1, Musikstueck pMusikstueck2)`**
Die Methode gibt die Güte des direkten Übergangs der beiden `Musikstueck`-Objekte `pMusikstueck1` und `pMusikstueck2` als Wert zwischen 1 und 3 zurück (vgl. Aufgabenstellung). Gibt es zwischen beiden Musikstücken gar keinen Übergang, wird 0 geliefert.

Anfrage **`boolean passtAlsNaechstesOderUebernaechstes(Musikstueck pMusikstueck1, Musikstueck pMusikstueck2)`**
Die Methode überprüft, ob es gut ist, ein Musikstück direkt nach einem anderen oder als übernächstes abzuspielen.
Dazu überprüft sie, ob es einen mindestens guten Übergangspfad, d. h. Güte 2 oder 3, gibt, welcher die beiden `Musikstueck`-Objekte `pMusikstueck1` und `pMusikstueck2` mit einer oder zwei Kanten verbindet (vgl. Aufgabenstellung).
Die Methode gibt `true` zurück, falls es einen solchen Übergangspfad gibt, andernfalls gibt sie `false` zurück.

Name: _____

Die Klasse Graph

Die Klasse `Graph` stellt einen ungerichteten, kantengewichteten Graphen dar. Es können Knoten- und Kantenobjekte hinzugefügt und entfernt, flache Kopien der Knoten- und Kantenlisten des Graphen angefragt und Markierungen von Knoten und Kanten gesetzt und überprüft werden. Des Weiteren kann eine Liste der Nachbarn eines bestimmten Knoten, eine Liste der inzidenten Kanten eines bestimmten Knoten und die Kante von einem bestimmten Knoten zu einem anderen Knoten angefragt werden. Abgesehen davon kann abgefragt werden, welches Knotenobjekt zu einer bestimmten ID gehört und ob der Graph leer ist.

Dokumentation der Klasse `Graph`

Konstruktor **`Graph()`**

Ein Objekt vom Typ `Graph` wird erstellt. Der von diesem Objekt repräsentierte Graph ist leer.

Auftrag **`void addVertex(Vertex pVertex)`**

Der Auftrag fügt den Knoten `pVertex` vom Typ `Vertex` in den Graphen ein, sofern es noch keinen Knoten mit demselben ID-Eintrag wie `pVertex` im Graphen gibt und `pVertex` eine ID ungleich `null` hat. Ansonsten passiert nichts.

Auftrag **`void addEdge(Edge pEdge)`**

Der Auftrag fügt die Kante `pEdge` in den Graphen ein, sofern beide durch die Kante verbundenen Knoten im Graphen enthalten sind, nicht identisch sind und noch keine Kante zwischen den beiden Knoten existiert. Ansonsten passiert nichts.

Auftrag **`void removeVertex(Vertex pVertex)`**

Der Auftrag entfernt den Knoten `pVertex` aus dem Graphen und löscht alle Kanten, die mit ihm inzident sind. Ist der Knoten `pVertex` nicht im Graphen enthalten, passiert nichts.

Auftrag **`void removeEdge(Edge pEdge)`**

Der Auftrag entfernt die Kante `pEdge` aus dem Graphen. Ist die Kante `pEdge` nicht im Graphen enthalten, passiert nichts.

Anfrage **`Vertex getVertex(String pID)`**

Die Anfrage liefert das Knotenobjekt mit `pID` als ID. Ist ein solches Knotenobjekt nicht im Graphen enthalten, wird `null` zurückgeliefert.

Name: _____

Anfrage **`List<Vertex> getVertices()`**

Die Anfrage liefert eine neue Liste aller Knotenobjekte vom Typ `List<Vertex>`. Enthält der Graph keine Knotenobjekte, so wird eine leere Liste zurückgeliefert.

Anfrage **`List<Vertex> getNeighbours(Vertex pVertex)`**

Die Anfrage liefert alle Nachbarn des Knotens `pVertex` als neue Liste vom Typ `List<Vertex>`. Hat der Knoten `pVertex` keine Nachbarn in diesem Graphen oder ist gar nicht in diesem Graphen enthalten, so wird eine leere Liste zurückgeliefert.

Anfrage **`List<Edge> getEdges()`**

Die Anfrage liefert eine neue Liste aller Kantenobjekte vom Typ `List<Edge>`. Enthält der Graph keine Kantenobjekte, so wird eine leere Liste zurückgeliefert.

Anfrage **`List<Edge> getEdges(Vertex pVertex)`**

Die Anfrage liefert eine neue Liste aller inzidenten Kanten zum Knoten `pVertex`. Hat der Knoten `pVertex` keine inzidenten Kanten in diesem Graphen oder ist gar nicht in diesem Graphen enthalten, so wird eine leere Liste zurückgeliefert.

Anfrage **`Edge getEdge(Vertex pVertex, Vertex pAnotherVertex)`**

Die Anfrage liefert die Kante, welche die Knoten `pVertex` und `pAnotherVertex` verbindet, als Objekt vom Typ `Edge`. Ist der Knoten `pVertex` oder der Knoten `pAnotherVertex` nicht im Graphen enthalten oder gibt es keine Kante, die beide Knoten verbindet, so wird `null` zurückgeliefert.

Auftrag **`void setAllVertexMarks(boolean pMark)`**

Der Auftrag setzt die Markierungen aller Knoten des Graphen auf den Wert `pMark`.

Anfrage **`boolean allVerticesMarked()`**

Die Anfrage liefert `true`, wenn die Markierungen aller Knoten des Graphen den Wert `true` haben, ansonsten `false`.

Auftrag **`void setAllEdgeMarks(boolean pMark)`**

Der Auftrag setzt die Markierungen aller Kanten des Graphen auf den Wert `pMark`.

Name: _____

Anfrage `boolean allEdgesMarked()`

Die Anfrage liefert `true`, wenn die Markierungen aller Kanten des Graphen den Wert `true` haben, ansonsten `false`.

Anfrage `boolean isEmpty()`

Die Anfrage liefert `true`, wenn der Graph keine Knoten enthält, ansonsten `false`.

Die Klasse `Vertex`

Die Klasse `Vertex` stellt einen einzelnen Knoten eines Graphen dar. Jedes Objekt dieser Klasse verfügt über eine im Graphen eindeutige ID als String und kann diese ID zurückliefern. Darüber hinaus kann eine Markierung gesetzt und abgefragt werden.

Dokumentation der Klasse `Vertex`

Konstruktor `Vertex(String pID)`

Ein neues Objekt vom Typ `Vertex` ohne Inhaltsobjekt wird erstellt. Der von diesem Objekt repräsentierte Knoten ist noch in keinen Graphen eingefügt. Seine Markierung hat den Wert `false`.

Anfrage `String getID()`

Die Anfrage liefert die `ID` des Knotens als `String`.

Auftrag `void setMark(boolean pMark)`

Der Auftrag setzt die Markierung des Knotens auf den Wert `pMark`.

Anfrage `boolean isMarked()`

Die Anfrage liefert `true`, wenn die Markierung des Knotens den Wert `true` hat, ansonsten `false`.

Name: _____

Die Klasse Edge

Die Klasse Edge stellt eine einzelne, ungerichtete Kante eines Graphen dar. Beim Erstellen werden die beiden durch sie zu verbindenden Knotenobjekte und eine Gewichtung als `double` übergeben. Beide Knotenobjekte können abgefragt werden. Des Weiteren können die Gewichtung und eine Markierung gesetzt und abgefragt werden.

Dokumentation der Klasse Edge

Konstruktor **`Edge(Vertex pVertex, Vertex pAnotherVertex, double pWeight)`**

Ein neues Objekt vom Typ `Edge` wird erstellt. Die von diesem Objekt repräsentierte Kante verbindet die Knoten `pVertex` und `pAnotherVertex` mit der Gewichtung `pWeight` und ist noch in keinen Graphen eingefügt. Ihre Markierung hat den Wert `false`.

Auftrag **`void setWeight(double pWeight)`**

Der Auftrag setzt das Gewicht der Kante auf den Wert `pWeight`.

Anfrage **`double getWeight()`**

Die Anfrage liefert das Gewicht der Kante als `double`.

Anfrage **`Vertex[] getVertices()`**

Die Anfrage gibt die beiden Knoten, die durch die Kante verbunden werden, als neues Feld vom Typ `Vertex` zurück. Das Feld hat genau zwei Einträge mit den Indexwerten `0` und `1`.

Auftrag **`void setMark(boolean pMark)`**

Der Auftrag setzt die Markierung der Kante auf den Wert `pMark`.

Auftrag **`void setWeight(double pWeight)`**

Der Auftrag setzt das Gewicht der Kante auf den Wert `pWeight`.

Anfrage **`boolean isMarked()`**

Die Anfrage liefert `true`, wenn die Markierung der Kante den Wert `true` hat, ansonsten `false`.

Name: _____

Die generische Klasse List<ContentType>

Objekte der generischen Klasse **`List`** verwalten beliebig viele, linear angeordnete Objekte vom Typ **`ContentType`**. Auf höchstens ein Listenobjekt, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.

Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt werden.

Dokumentation der Klasse `List<ContentType>`

Konstruktor **`List()`**
Eine leere Liste wird erzeugt. Objekte, die in dieser Liste verwaltet werden, müssen vom Typ `ContentType` sein.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert den Wert `true`, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert `false`.

Anfrage **`boolean hasAccess()`**
Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.

Auftrag **`void next()`**
Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d. h., `hasAccess()` liefert den Wert `false`.

Auftrag **`void toFirst()`**
Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Auftrag **`void toLast()`**
Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Name: _____

Anfrage **`ContentType getContent()`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt zurückgegeben. Andernfalls (`hasAccess() == false`) gibt die Anfrage den Wert `null` zurück.

Auftrag **`void setContent(ContentType pContent)`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`) und `pContent` ungleich `null` ist, wird das aktuelle Objekt durch `pContent` ersetzt. Sonst bleibt die Liste unverändert.

Auftrag **`void append(ContentType pContent)`**

Ein neues Objekt `pContent` wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt `pContent` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (`hasAccess() == false`).
Falls `pContent` gleich `null` ist, bleibt die Liste unverändert.

Auftrag **`void insert(ContentType pContent)`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird ein neues Objekt `pContent` vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert.
Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (`hasAccess() == false`), wird `pContent` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt.
Falls es kein aktuelles Objekt gibt (`hasAccess() == false`) und die Liste nicht leer ist oder `pContent == null` ist, bleibt die Liste unverändert.

Auftrag **`void concat(List<ContentType> pList)`**

Die Liste `pList` wird an die Liste angehängt. Anschließend wird `pList` eine leere Liste. Das aktuelle Objekt bleibt unverändert. Falls es sich bei der Liste und `pList` um dasselbe Objekt handelt, `pList == null` oder eine leere Liste ist, bleibt die Liste unverändert.

Auftrag **`void remove()`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (`hasAccess() == false`).
Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (`hasAccess() == false`), bleibt die Liste unverändert.

Name: _____

Die generische Klasse Queue<ContentType>

Objekte der generischen Klasse **`Queue`** (Warteschlange) verwalten beliebige Objekte vom Typ **`ContentType`** nach dem First-In-First-Out-Prinzip, d. h., das zuerst abgelegte Objekt wird als erstes wieder entnommen. Alle Methoden haben eine konstante Laufzeit, unabhängig von der Anzahl der verwalteten Objekte.

Dokumentation der generischen Klasse `Queue<ContentType>`

Konstruktor **`Queue()`**

Eine leere Schlange wird erzeugt. Objekte, die in dieser Schlange verwaltet werden, müssen vom Typ `ContentType` sein.

Anfrage **`boolean isEmpty()`**

Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag **`void enqueue(ContentType pContent)`**

Das Objekt `pContent` wird an die Schlange angehängt. Falls `pContent` gleich `null` ist, bleibt die Schlange unverändert.

Auftrag **`void dequeue()`**

Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.

Anfrage **`ContentType front()`**

Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird `null` zurückgegeben.

Unterlagen für die Lehrkraft

Abiturprüfung 2020

Informatik, Leistungskurs

1. Aufgabenart

Analyse, Modellierung und Implementation von kontextbezogenen Problemstellungen mit Schwerpunkt auf den Inhaltsfeldern Daten und ihre Strukturierung und Algorithmen

2. Aufgabenstellung[1]

siehe Prüfungsaufgabe

3. Materialgrundlage

entfällt

4. Bezüge zum Kernlehrplan und zu den Vorgaben 2020

Die Aufgaben weisen vielfältige Bezüge zu den Kompetenzerwartungen und Inhaltsfeldern des Kernlehrplans bzw. zu den in den Vorgaben ausgewiesenen Fokussierungen auf.
Im Folgenden wird auf Bezüge von zentraler Bedeutung hingewiesen.

1. *Inhaltsfelder und inhaltliche Schwerpunkte*
   Daten und ihre Strukturierung
   - Objekte und Klassen
     - Entwurfsdiagramme und Implementationsdiagramme
     - Lineare Strukturen
       Schlange (Klasse `Queue`*)*
       Lineare Liste (Klasse `List`*)*
     - Nicht-lineare Strukturen
       Graphen (Klasse `Graph`, `Vertex`, `Edge`*)*
   - Datenbanken

   Algorithmen
   - Analyse, Entwurf und Implementierung von Algorithmen
   - Algorithmen in ausgewählten informatischen Kontexten

   Formale Sprachen und Automaten
   - Syntax und Semantik einer Programmiersprache
     - Java

2. *Medien/Materialien*
   - entfällt

[1] Die Aufgabenstellung deckt inhaltlich alle drei Anforderungsbereiche ab.

5. Zugelassene Hilfsmittel

- GTR (grafikfähiger Taschenrechner) oder CAS (Computer-Algebra-System)
- Wörterbuch zur deutschen Rechtschreibung

6. Modelllösungen

Die jeweilige Modelllösung stellt eine mögliche Lösung bzw. Lösungsskizze dar. Der gewählte Lösungsansatz und -weg der Schülerinnen und Schüler muss nicht identisch mit dem der Modelllösung sein. Sachlich richtige Alternativen werden mit entsprechender Punktzahl bewertet (Bewertungsbogen: Zeile „Sachlich richtige Lösungsalternative zur Modelllösung“).

Teilaufgabe a)

Zwischen den genannten Stücken sollten die Musikstücke mit den Titeln `"Swinging Charly"` und `"Super Swing"` gespielt werden.

`"Dancing For Me"` bildet sozusagen eine Engstelle in dem Graphen. Wenn man mit diesem Musikstück anfängt, muss man sich entscheiden, ob man ausschließlich Musikstücke aus dem oberen Teilgraphen oder ausschließlich solche aus dem unteren Teilgraphen abspielen möchte. Würde man bei diesem Startstück beide Teilgraphen besuchen wollen, müsste das Startstück doppelt gespielt werden. Dies soll gemäß der Aufgabenstellung jedoch vermieden werden.

Teilaufgabe b)

Die Beziehung zwischen den Klassen `Musikstueck` und `Vertex` ist eine Vererbung, d. h., die Klasse `Musikstueck` besitzt zusätzlich zu ihren spezifischen Attributen und Methoden alle Attribute und Methoden der Klasse `Vertex`. Somit können Objekte der Klasse `Musikstueck` im Graphen als spezielle Knoten verwaltet werden.

Die Klasse `DJAssistent` verwaltet im Attribut `djGraph` ein Objekt der Klasse `Graph`. In diesem Graphen werden die einzelnen Musikstücke, samt ihren Beziehungen untereinander, verwaltet.

Aus Entwurfssicht ist ein Objekt der Klasse `DJAssistent` kein Graph, sondern verwaltet lediglich die Objekte der Klasse `Musikstueck` in einem Objekt der Klasse `Graph`. Würde die Klasse `DJAssistent` als Unterklasse der Klasse `Graph` modelliert, so würden sämtliche Methoden der Klasse `Graph` nach außen sichtbar sein und man könnte von außen die intern gespeicherten Daten kontextwidrig manipulieren. Dies widerspricht dem Grundsatz der Datenkapselung in einer objektorientierten Modellierung.

Teilaufgabe c)

`gibUebergangsGuete(musikstueck1, musikstueck2)` gibt den Wert `1` zurück.

`passtAlsNaechstesOderUebernaechstes(musikstueck1, musikstueck2)` gibt den Wert `true` zurück.

Eine Implementierung der Methode könnte wie folgt aussehen.

```
public boolean passtAlsNaechstesOderUebernaechstes(
            Musikstueck pMusikstueck1, Musikstueck pMusikstueck2) {
  boolean passt = false;
  if (gibUebergangsGuete(pMusikstueck1, pMusikstueck2) >= 2) {
    passt = true;
  } else {
    List<Vertex> nachbarn = djGraph.getNeighbours(pMusikstueck1);
    nachbarn.toFirst();
    while (nachbarn.hasAccess() && !passt) {
      Musikstueck aktuell = (Musikstueck) nachbarn.getContent();
      if (gibUebergangsGuete(pMusikstueck1, aktuell) >= 2
              && gibUebergangsGuete(aktuell, pMusikstueck2) >= 2) {
        passt = true;
      }
      nachbarn.next();
    }
  }
  return passt;
}
```

Teilaufgabe d)

Die Warteschlange wird am Anfang mit dem Objekt `pMusikstueck1 (Musikstueck`-Objekt zum Titel `"Dancing For Me")` initialisiert und der zugehörige Knoten wird markiert (Zeile 5 und 6).

Die Schleife in den Zeilen 7 bis 22 wird nun solange ausgeführt, bis die Warteschlange leer ist (und deshalb der Zielknoten nicht erreicht wurde) oder bis das erste Element der Warteschlange dem Zielknoten (hier also dem `Musikstueck`-Objekt `pMusikstueck2` zu dem Musiktitel `"Hello Germany!"`) entspricht.

Innerhalb der Schleife werden alle unmarkierten 3er-Nachbarn (d. h. über eine Kante mit dem Kantengewicht 3 verbundene Knoten) des ersten Knotens der Warteschlange gesucht, markiert und hinten in die Warteschlange eingereiht (Zeilen 9 bis 20). Dies ist im ersten Durchlauf nur das `Musikstueck`-Objekt zum Titel `"Tonight Is The Night"`.

Am Ende der Schleife wird dann der vorderste Knoten aus der Warteschlange entfernt (Zeile 21), sodass nun vom `Musikstueck`-Objekt zum Titel `"Tonight Is The Night"` weitergesucht wird.

Von diesem Musikstück ausgehend findet der Algorithmus das Nachbarmusikstück zum Titel `"Tell Me Where!"`, welches in die Warteschlange eingereiht und markiert wird. Weitere unmarkierte 3er-Nachbarn gibt es nicht, weshalb das Musikstück zum Titel `"Tonight Is The Night"` wieder aus der Warteschlange entfernt wird.

Ausgehend vom Musikstück zum Titel `"Tell Me Where!"` findet der Algorithmus das Nachbarmusikstück mit dem Titel `"Hello Germany!"`, welches in die Warteschlange eingereiht und markiert wird. Weitere unmarkierte 3er-Nachbarn gibt es nicht, weshalb das Musikstück mit dem Titel `"Tell Me Where!"` wieder aus der Warteschlange entfernt wird.

Die Schleifenbedingung in Zeile 8 ist nun nicht mehr erfüllt, da der erste Knoten der Warteschlange dem Zielknoten `pMusikstueck2` entspricht. Deshalb geht der Algorithmus zur Rückgabe (Zeilen 23 und 24) über, wo festgestellt wird, dass die Schlange nicht leer ist und der erste Knoten der Warteschlange dem Zielknoten `pMusikstueck2` entspricht.

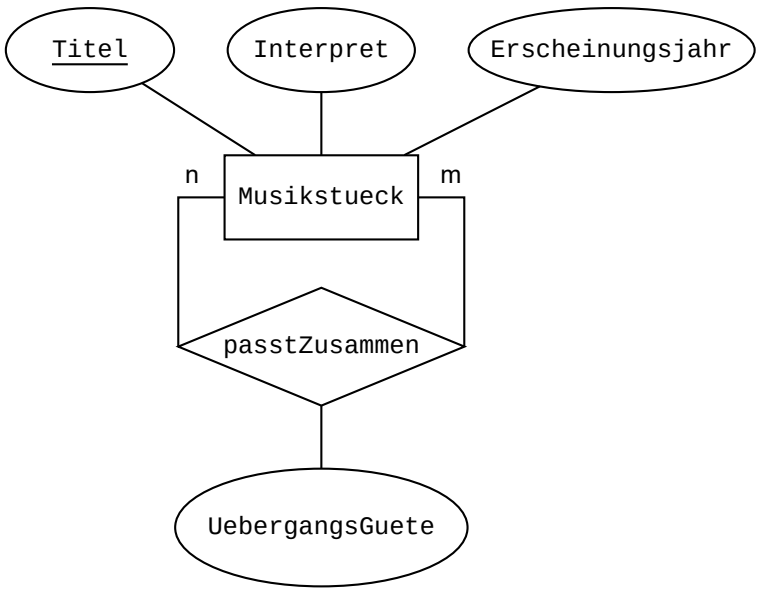
Die Methode gibt den Wert `true` zurück.

Im Hinblick auf die Funktionsweise des Algorithmus bedeutet eine Markierung, dass ein markierter Knoten bei der Suche nach einem Weg von `pMusikstueck1` nach `pMusikstueck2` bereits besucht wurde (Markierung in Zeilen 5 bzw. 16) und deshalb nicht ein weiteres Mal besucht werden darf (Bedingung in Zeile 15).

Im Sachzusammenhang überprüft die Methode, ob es einen optimalen Übergangspfad (Güte 3) zwischen zwei Musikstücken gibt.

Teilaufgabe e)

Eine mögliche Datenbankmodellierung ist die folgende:



Die im Graph gespeicherten Musikstücke entsprechen Entitäten des Entitätstyps `Musikstueck`, dessen Attribute exakt den Attributen der Klasse `Musikstueck` und damit den im Graphen gespeicherten Informationen entsprechen.

Der Beziehungstyp `passtZusammen` modelliert die Kanten, über welche jeweils zwei Objekte der Klasse `Musikstueck` im Graphen verbunden sind, wobei das Kantengewicht als Attribut `UebergangsGuete` der Beziehungsrelation modelliert wurde.

Für die Modellierung des Beziehungstyps `passtZusammen` zwischen zwei Musikstücken wurde ein `m:n`-Beziehungstyp gewählt. So kann ein Musikstück mit beliebig vielen anderen Musikstücken in Beziehung stehen, aber es können auch beliebig viele andere Musikstücke mit einem Musikstück in Beziehung stehen. Dadurch ist gewährleistet, dass jede Kante des Graphen zweimal in der Datenbank gespeichert werden kann, was der Idee des ungerichteten Graphen entspricht.

Insgesamt ermöglicht diese Modellierung somit die Speicherung aller Musikstücke und deren Beziehungen zueinander.

Die Realisierung der Aufgabe zur Methode `passtAlsNaechstesOderUebernaechstes` kann in zwei Schritten erfolgen.

Die Überprüfung, ob zwei Musikstücke durch einen mindestens guten Übergang verbunden sind, erfolgt dadurch, dass die Relation `Musikstueck` über die Relation `passtZusammen` mit sich selbst verbunden wird (`join`). Aus der Ergebnisrelation muss nun noch entsprechend der Bedingung `UebergangsGuete >= 2` selektiert werden. Wenn diese Selektion mindestens einen Datensatz liefert, so ist entschieden, dass die beiden Musikstücke einen mindestens guten Übergang bilden.

Die Überprüfung, ob die beiden Musikstücke als übernächstes gut oder sogar optimal hintereinander passen, erfolgt über eine doppelte Verknüpfung `Musikstueck - passtZusammen - Musikstueck - passtZusammen - Musikstueck`. Aus der Ergebnisrelation muss so wie oben entsprechend der Bedingung `UebergangsGuete >= 2` selektiert werden.

7. Teilleistungen – Kriterien / Bewertungsbogen zur Prüfungsarbeit

Name des Prüflings:_____ Kursbezeichnung:_____

Schule: _____

Teilaufgabe a)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**[2]	**ZK**	**DK**
1	gibt einen optimalen Übergangspfad zwischen den genannten Musikstücken an.	3			
2	erläutert im Sachzusammenhang, welche Auswirkungen es hat, wenn der Titel `"Dancing For Me"` an einem Abend als erstes abgespielt wird.	4			
Sachlich richtige Lösungsalternative zur Modelllösung: (7) …… ……					
	Summe Teilaufgabe a)	**7**			

Teilaufgabe b)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	erläutert die Beziehungen zwischen den Klassen `DJAssistent`, `Musikstueck`, `Graph` und `Vertex`.	4			
2	begründet, dass es nicht sinnvoll ist, die Klasse `DJAssistent` als Unterklasse der Klasse `Graph` zu modellieren.	4			
Sachlich richtige Lösungsalternative zur Modelllösung: (8) …… ……					
	Summe Teilaufgabe b)	**8**			

[2] EK = Erstkorrektur; ZK = Zweitkorrektur; DK = Drittkorrektur

Teilaufgabe c)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	gibt an, welche Werte die Methodenaufrufe für den Beispielgraphen zurückgeben.	2			
2	implementiert die Methode `passtAlsNaechstesOderUebernaechstes`.	9			
Sachlich richtige Lösungsalternative zur Modelllösung: (11) …… ……					
	Summe Teilaufgabe c)	**11**			

Teilaufgabe d)

	Anforderungen	**Lösungsqualität**			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	analysiert die Methode und erläutert den Quelltext unter Bezugnahme auf die Veränderungen der Schlange bei Anwendung auf die Beispieldaten.	6			
2	ermittelt den Rückgabewert der Methode bei Anwendung auf die Beispieldaten.	2			
3	erläutert die Bedeutung der Knotenmarkierungen im Hinblick auf die Funktionsweise des Algorithmus.	2			
4	erläutert die Funktionalität der Methode im Sachzusammenhang.	2			
Sachlich richtige Lösungsalternative zur Modelllösung: (12) …… ……					
	Summe Teilaufgabe d)	**12**			

Teilaufgabe e)

	Anforderungen	Lösungsqualität			
	Der Prüfling	**maximal erreichbare Punktzahl**	**EK**	**ZK**	**DK**
1	entwickelt eine Datenbankmodellierung in Form eines Entity-Relationship-Diagramms, welche die Speicherung aller Musikstücke und deren Beziehungen zueinander ermöglicht.	4			
2	erläutert, wie die Modellierung die Speicherung aller Musikstücke und deren Beziehungen zueinander ermöglicht.	4			
3	entwickelt eine Idee, wie eine Datenbankabfrage aussehen könnte, welche auf Grundlage der Datenbankmodellierung dieselbe Information wie die Methode `passtAlsNaechstesOderUebernaechstes` ermittelt.	4			
Sachlich richtige Lösungsalternative zur Modelllösung: (12) …… ……					
	Summe Teilaufgabe e)	**12**			

	Summe insgesamt	**50**			

Festlegung der Gesamtnote (Bitte nur bei der letzten bearbeiteten Aufgabe ausfüllen.)

	Lösungsqualität			
	maximal erreichbare Punktzahl	**EK**	**ZK**	**DK**
Übertrag der Punktsumme aus der ersten bearbeiteten Aufgabe	**50**			
Übertrag der Punktsumme aus der zweiten bearbeiteten Aufgabe	**50**			
Übertrag der Punktsumme aus der dritten bearbeiteten Aufgabe	**50**			
Punktzahl der gesamten Prüfungsleistung	**150**			
aus der Punktsumme resultierende Note gemäß nachfolgender Tabelle				
Note ggf. unter Absenkung um bis zu zwei Notenpunkte gemäß § 13 Abs. 2 APO-GOSt				
Paraphe				

Berechnung der Endnote nach Anlage 4 der Abiturverfügung auf der Grundlage von § 34 APO-GOSt

Die Klausur wird abschließend mit der Note _____ (____ Punkte) bewertet.

Unterschrift, Datum:

Grundsätze für die Bewertung (Notenfindung)

Für die Zuordnung der Notenstufen zu den Punktzahlen ist folgende Tabelle zu verwenden:

Note	**Punkte**	**Erreichte Punktzahl**
sehr gut plus	15	150 – 143
sehr gut	14	142 – 135
sehr gut minus	13	134 – 128
gut plus	12	127 – 120
gut	11	119 – 113
gut minus	10	112 – 105
befriedigend plus	9	104 – 98
befriedigend	8	97 – 90
befriedigend minus	7	89 – 83
ausreichend plus	6	82 – 75
ausreichend	5	74 – 68
ausreichend minus	4	67 – 60
mangelhaft plus	3	59 – 50
mangelhaft	2	49 – 41
mangelhaft minus	1	40 – 30
ungenügend	0	29 – 0