252-0027-00: Einführung in die Programmierung

Übungsblatt 12

Abgabe: 09. Dezember 2025, 23:59

Checken Sie mit IntelliJ wie bisher die neue Übungsvorlage aus. Importieren Sie beide IntelliJ-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben).

Ab Woche 5 werden die TAs Ihnen nur dann Feedback geben, wenn Sie eine entsprechende Datei namens "requestfeedback.txt" in dem Projekt-Ordner hochladen. In der Text-Datei geben Sie an, für welche Aufgaben Sie Feedback erhalten möchten. Um eine neue Text-Datei in IntelliJ zu erstellen, rechts-klicken Sie den uXX Projekt-Ordner und wählen *New* → *File*. Im erscheinenden Fenster, benennen Sie die Datei mit "requestfeedback.txt" und drücken die Enter-Taste. Danach wählen Sie *Add*, falls IntelliJ Sie fragt, ob Sie die Datei zu git hinzufügen möchten.

Aufgabe 1: Cyclic List

Bisher haben Sie einfach verkettete Listen gesehen. Zusätzlich wurde ein `IntList`-Interface eingeführt (siehe Anhang), welches die Methoden der Liste abstrahiert.

a) In dieser Aufgabe üben Sie den Umgang mit Interfaces. Die Klasse `LinkedIntList` hat alle Methoden, welche vom Interface `IntList` gefordert werden. Implementieren Sie dann eine Methode `ListUtil.addMin(IntList x)`, welche der Liste `x` die kleinste Zahl anhängt, welche in `x` enthalten ist. Implementieren Sie zuletzt die Methode `ListUtil.addMinImpl(LinkedIntList x)`, welche ebenfalls der Liste `x` die kleinste Zahl anhängt, welche in `x` enthalten ist, aber dafür die Methode `ListUtil.addMin` verwenden soll. Sie dürfen für beide Methoden annehmen, dass die übergebene Liste mindestens ein Element enthält.

b) In dieser Aufgabe implementieren Sie eine neue Variante einer Liste, die zyklische Liste, welche ebenfalls das `IntList`-Interface implementiert. Zyklische Listen sind ähnlich zu einfach verketteten Listen mit dem Unterschied, dass das `next`-Feld der letzten `Node` der Liste, falls es einen letzten Knoten gibt, auf den ersten `Node` der Liste zeigt. Die Knoten der Liste bilden also einen Zyklus. Zusätzlich hat die Liste kein Feld für den ersten Knoten der Liste, da dies unnötig ist. Das Feld `last`, das auf den letzten Knoten zeigt, ist nach wie vor vorhanden. Abbildung 1 zeigt eine solche zyklische Listen mit den Elementen 1, 3, 3, 7. Implementieren Sie die zyklische Liste in der Datei "CircularLinkedIntList.java". Einige Tests für die Liste finden Sie in `IntListTest`.

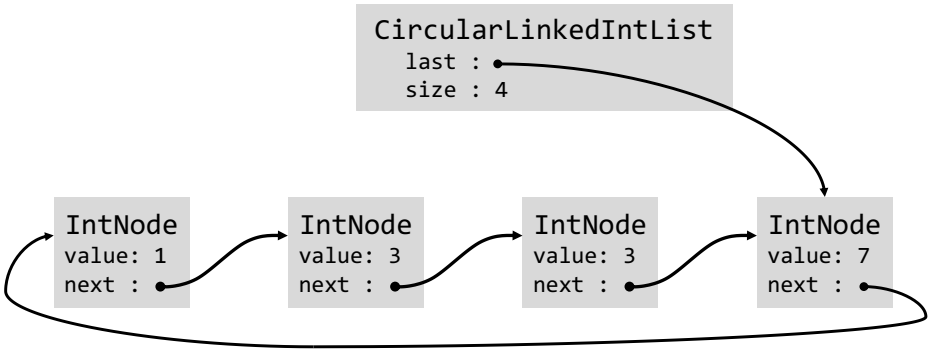

Abbildung 1: Zyklische Liste mit Werten 1, 3, 3, 7.

Aufgabe 2: Loop-Invariante

Gegeben die Pre- und Postcondition formulieren Sie *nützliche Schleifeninvarianten* für folgende Programme in der Datei "LoopInvariante.txt".

1. Um die Loop-Invariante einfacher schreiben zu können, dürfen Sie `contains(arr, c)` benutzen. Hier sagt uns `contains(arr, c)`, ob der `Char c` im Array `arr` enthalten ist. Ebenfalls können Sie `subarray(arr, i)` benutzen, welches eine Kopie vom Array `arr` von Index `0` bis und mit `i` darstellt. Alternativ könnte man auch formale Notation benutzen, indem man mit Quantoren arbeitet.

```
void erase(char[] arr, char c) {
  // Precondition: arr != null && c != 'x'
  int i = 0;

  // Loop-Invariante:
  while (i != arr.length) {
    if (arr[i] == c) {
      arr[i] = 'x';
    }

    i++;
  }

  // Postcondition: !contains(arr, c)
}
```

2.
```
public int compute(String s, char c) {
  // Precondition: s != null
  int x;
  int i;

  x = 0;
  i = -1;

  // Loop-Invariante:
  while (x < s.length() && i < 0) {
    if (s.charAt(x) == c) {
      i = x;
    }
    x = x + 1;
  }

  // Postcondition:
  //   (0 <= i && i < s.length() && s.charAt(i) == c) || count(s, c) == 0
  return i;
}
```

Die Methode `count(String s, char c)` gibt zurück, wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop-Invariante in die Datei "LoopInvariante.txt".

Achtung: Die Aufgabe ist schwerer, als es zuerst scheint. Überprüfen Sie Ihre Lösung sorgfältig.

Aufgabe 3: Expression Evaluator

In dieser und in folgenden Übungen werden Sie eine Reihe von Programmen schreiben, welche andere Programme interpretieren, kompilieren oder (in kompilierter Form) ausführen. Die Programmiersprachen definieren wir selber.

Als Einstieg schreiben Sie ein Programm, welches mathematische Ausdrücke (*expressions*) auswertet. Die Ausdrücke bestehen aus Zahlen, Variablen, Operatoren wie $+$ oder $-$ und einfachen Funktionen wie $\sin()$ oder $\cos()$. Die genaue Syntax für diese Ausdrücke finden Sie als EBNF-Beschreibung in Abbildung 2.

Ein Programm, das Ausdrücke auswertet, muss natürlich entscheiden, ob eine gegebene Zeichenkette überhaupt ein gültiger Ausdruck ist[1]. Das nennt man *parsen* und ein solches Programm heisst *Parser*. Aus einer EBNF-Beschreibung wie dieser kann man einfach einen Parser erstellen[2]:

- Regeln werden zu Methoden.
- Alternativen werden zu `if`-Anweisungen.
- Regeln auf der RHS werden zu Methodenaufrufen.

[1] ähnlich wie Sie, wenn Sie z.B. mit Hilfe einer Tabelle überprüfen, ob ein Symbol einer EBNF-Beschreibung entspricht

[2] Im Allgemeinen, d.h. für gewisse andere EBNF-Beschreibungen, ist das leider nicht möglich.

digit ← 0 | 1 | ... | 9
char ← A | B | ... | Z | a | b | ... | z
num ← *digit* { *digit* } [. *digit* { *digit* }]
var ← *char* { *char* }
func ← *char* { *char* } (
op ← + | - | * | / | ^
open ← (
close ←)

atom ← *num* | *var*
term ← *open expr close* | *func expr close* | *atom*
expr ← *term* [*op term*]

Abbildung 2: **EBNF-Beschreibung** von *expr*

Man unterscheidet dabei zwischen zwei Arten von Regeln: *Parser-Regeln* und *Tokenizer-Regeln*. Zuerst teilt ein *Tokenizer* die Zeichenkette aufgrund der Tokenizer-Regeln in eine Reihe von Tokens auf. In unserer EBNF-Beschreibung sind die Tokenizer-Regeln rot dargestellt. Die grauen Regeln werden zwar intern vom Tokenizer verwendet, aber erzeugen keine eigenen Tokens. Zum Beispiel erzeugt die Zeichenkette "`sin(1 + x) * 3.14`" die folgende Reihe von Tokens:

func : `sin(` | *num* : `1` | *op* : `+` | *var* : `x` | *close* : `)` | *op* : `*` | *num* : `3.14`

Danach entscheidet der Parser aufgrund der Parser-Regeln (oben in Schwarz dargestellt), ob eine solche Reihe von Tokens einen gültigen Ausdruck darstellt. Abbildung 3 zeigt, wie die Parser-Methode für *term* aussehen könnte.

a) In der Übungsvorlage finden Sie eine `Tokenizer`-Implementation, eine Vorlage für den `ExprParser` und eine `EvaluatorApp` mit einer `main()`-Methode. Diese parst die vom Benutzer eingegebenen Zeichenketten und gibt an, ob sie gültige Ausdrücke sind. Wenn der Benutzer "exit" eingibt, terminiert das Programm. Ihre Aufgabe ist es, den `ExprParser` zu schreiben.

   Erstellen Sie in der schon vorgegebenen `parse(String)`-Methode eine `Tokenizer`-Instanz. Die Methoden des `Tokenizers` sind denen der `Scanner`-Klasse nachempfunden. Sie können also die `hasNext*()`-Methoden verwenden, um zu prüfen, welche Art von Token als nächstes kommt, und die `next*()`-Methoden, um Tokens zu "konsumieren". Schreiben Sie die nötigen `parse*(...)`-Methoden, eine für jede Parser-Regel. Die erste Ihrer `parse*(...)`-Methoden rufen Sie von `parse(String)` aus auf. Diese Methoden sollen eine `EvaluationException` mit einer sinnvollen Fehlermeldung werfen, falls die Zeichenkette kein gültiger Ausdruck ist. Falls z.B. nach "`(`" und einer *expr* das Token "`10`" statt "`)`" folgt, könnte die Fehlermeldung lauten:

   ```
   Syntax error: unexpected token '10', expected ')'
   ```

b) Um aus dem `ExprParser` einen `ExprEvaluator` zu machen, kann man die Methoden so ändern, dass sie im selben Zug das Resultat berechnen. Jede Methode überprüft dann nicht

```
/* checks if the next tokens form a valid term */
void parseTerm(...) {
    if(next token is a "open") {
        consume "open" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else if(next token is a "func") {
        consume "func" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else {
        // check if the tokens are a valid atom:
        parseAtom(...);
    }
}
```

Abbildung 3: Parser-Methode für *term*

```
/* evaluate the next tokens as a term */
double evalTerm(...) {
    if(next token is a "open") {
        consume "open" token
        double val = evalExpr(...);
        check whether next token is a "close" & consume
        return val;
    }
    else if(next token is a "func") {
        consume "func" token
        double arg = evalExpr(...);
        check whether next token is a "close" & consume
        double result = apply function to arg
        return result;
    }
    else {
        return evalAtom(...);
    }
}
```

Abbildung 4: Evaluator-Methode für *term*

nur, ob die nächsten Tokens der Regel entsprechen, sondern gibt auch gleich den Wert des entsprechenden Ausdruck-Teils zurück. Dies sehen Sie in Abbildung 4.

Benennen Sie die Klasse und die Methoden um[3], so dass sie die neue Funktionalität widerspiegeln. Nun können Sie entscheiden: Erstens, welche Funktionen sind erlaubt? Sie sollten sin(), cos() und tan() unterstützen, aber auch andere Funktionen wie abs() oder log() könnten später Spass machen[4]. Zweitens können Sie entscheiden, wie Sie mit Variablen umgehen. Sie sollten mindestens eine "x"-Variable unterstützen, und wir empfehlen, dass Sie den Wert dafür dem `ExprEvaluator`-Konstruktor übergeben. Sie sollten eine Exception werfen, falls unbekannte Funktionen oder Variablen in einem Ausdruck vorkommen.

Am Schluss sollte die `EvaluatorApp` das Resultat der eingegebenen Ausdrücke ausgegeben, statt nur zu sagen, ob sie gültig sind. Wenn Sie wollen, können Sie dem Benutzer auch die Möglichkeit geben, Werte für Variablen zu definieren.

xkcd: (by Randall Munroe (CC BY-NC 2.5)

[3]Verwenden Sie dafür die *Rename*-Funktion von IntelliJ, welche Sie in der Übungsstunde gesehen haben.
[4]Schauen Sie sich die `Math`-Klasse für weitere Kandidaten an (oder implementieren Sie selber welche).

Aufgabe 4: Contact Tracing

In dieser Aufgabe implementieren Sie eine Contact-Tracing-Applikation, welche es ermöglichen soll, Kontakte während eines Virus-Ausbruches nachzuverfolgen. Ihre Implementierung soll zunächst Begegnungen zwischen verschiedenen `Person`-Instanzen anonym protokollieren, so dass bei einem positiven Test die Benachrichtigung aller Personen möglich ist, die direkt oder indirekt mit einer positiv getesteten Person in Kontakt standen.

Anonyme Begegnungen. Um Anonymität zu gewährleisten, dürfen zwei Personen A und B bei einer Begegnung lediglich anonyme Integer-IDs austauschen, ohne dabei die Identität der jeweils anderen Person aufzudecken. Beide Personen speichern hierbei sowohl die eigene ID als auch die ID der anderen Person. Bei der positiven Testung von A kann dann mithilfe der anonymen IDs, die A genutzt hat, festgestellt werden, ob B einer dieser IDs begegnet ist. Um zu vermeiden, dass wiederkehrende IDs die Identifikation einer Person über mehrere Begegnungen hinweg ermöglichen, benutzt jede Person für jede Begegnung frische IDs, welche über eine zentrale Klasse `ContactTracer` vergeben werden. Frisch bedeutet hierbei, dass eine ID zuvor noch nie bei einer Begegnung verwendet wurde.

Direkte und indirekte Kontakte. Nachdem eine Reihe an Begegnungen protokolliert wurden, wird eine oder mehrere Personen positiv getestet. Mit dem erfassten Netzwerk aus Begegnungen soll Ihre Applikation dann zwei verschiedene Arten an Kontaktpersonen bestimmen:

- Als *direkte Kontakte* gelten alle Personen, die eine Begegnung mit einer positiv getesteten Person hatten.
- Als *indirekte Kontakte* hingegen gelten alle Personen, die zwar selbst keine Begegnung mit einer positiv getesteten Person hatten, jedoch Kontakt mit mindestens einer anderen Person, welche als direkter Kontakt gilt, hatten. Indirekte Kontakte mit mehr als einer Zwischenperson müssen Sie dabei nicht berücksichtigen.

Sie dürfen dabei annehmen, dass zunächst alle Begegnungen erfasst werden und erst dann Personen positiv getestet werden. Nach der ersten positiven Testung finden keine weiteren Begegnungen mehr statt.

Benachrichtigungen. Da nicht alle Personen gleichermassen gefährdet sind, soll Ihre Applikation die Benachrichtigung der Kontaktpersonen vom Alter, der Art des Kontaktes, sowie dem Testergebnis der jeweiligen Kontaktperson abhängig machen. Dabei soll eine der drei Warnstufen *Keine Benachrichtigung*, *Low-Risk-Benachrichtigung* oder *High-Risk-Benachrichtigung* ausgesprochen werden. Zu Beginn haben alle Personen die Standard-Warnstufe *Keine Benachrichtigung* und gelten als negativ getestet. Davon ausgehend sollen nach jedem registrierten positiven Test die zugehörigen Kontaktpersonen wie folgt benachrichtigt werden:

Testergebnis der Kontaktperson	Alter der Kontaktperson	Direkter Kontakt	Indirekter Kontakt
Positiv	-	Keine Benachr.	Keine Benachr.
Negativ	$\leq$ 60 Jahre alt	High-Risk	Keine Benachr.
Negativ	$>$ 60 Jahre alt	High-Risk	Low-Risk

Eine negativ getestete Person, die höchstens 60 Jahre alt ist und die nur in indirektem Kontakt zu einer positiven Person stand, soll beispielsweise keine Benachrichtigung erhalten (Reihe 2). Eine negativ getestete Person über 60 Jahre hingegen soll als indirekter Kontakt eine Low-Risk-Benachrichtigung erhalten (Reihe 3).

Wenn mehrere Personen positiv getestet werden, soll Ihre Applikation immer die höchste geltende Warnstufe für die anderen negativ getesteten Personen berechnen. Dabei ist die Ordnung der Warnstufen wie folgt definiert: *Keine Benachrichtigung* < *Low-Risk Benachrichtigung* < *High-Risk Benachrichtigung*. Positiv getestete Personen hingegen sollen immer die Warnstufe *Keine Benachrichtigung* erhalten. Im Allgemeinen dürfen Sie zudem annehmen, dass eine Person, die einmal positiv getestet wurde, für den Rest der Laufzeit Ihrer Applikation als positiv getestet gilt.

Implementierung. Erweitern Sie den vorgegebenen Code für die Klasse `ContactTracer` und das Interface `Person` wie folgt, um die Contact-Tracing-Applikation umzusetzen:

Implementieren Sie das Interface `Person` mit den folgenden public Methoden:

- `Person.getUsedIds()`. Diese Methode gibt die Liste aller IDs zurück (`List<Integer>`), die für diese `Person` als frische ID verwendet wurden, um eine Begegnung zu protokollieren. Nach Hinzufügen einer ID in diese Liste muss dieselbe ID in die jeweilige `Person.getSeenIds()`-Liste des Gegenübers eingetragen sein.
- `Person.getSeenIds()`. Diese Methode gibt die Liste aller IDs zurück (`List<Integer>`), die diese Person als die frische ID des jeweiligen Gegenübers bei einer Begegnung protokolliert hat. Nach Hinzufügen einer ID in diese Liste muss dieselbe ID in die jeweilige `Person.getUsedIds()`-Liste des Gegenübers eingetragen sein.
- `Person.getNotification()`. Diese Methode gibt den aktuellen Benachrichtigungsstatus der `Person` zurück. Der Rückgabewert soll vom Enum-Typ `NotificationType` sein, welcher vorgegeben ist und die drei möglichen Warnstufen modelliert. `NotificationType` ist im Interface `Person` definiert und enthält die drei Werte `NoNotification` (keine Benachrichtigung), `LowRiskNotification` (Low-Risk-Benachrichtigung) und `HighRiskNotification` (High-Risk-Benachrichtigung).
- `Person.setTestsPositively()`. Diese Methode wird aufgerufen, um eine `Person` als positiv getestet zu markieren. Nach dem Aufrufen dieser Methode sollen automatisch alle Kontakte von A benachrichtigt worden sein und die entsprechende Warnstufe per `Person.getNotification()` zurückgeben.

Implementieren Sie zusätzlich die Klasse `ContactTracer`, welche die folgenden public Methoden besitzt:

- `ContactTracer.registerEncounter(Person p1, Person p2)`. Mit dieser Methode wird eine (beidseitige) Begegnung zwischen `Person`-Objekten `p1` und `p2` protokolliert, indem die beiden Personen anonyme IDs austauschen. Die ausgetauschten IDs müssen dabei unterschiedlich sein. Eine Begegnung zwischen `p1` und `p2` ist beidseitig und muss somit auch als Begegnung zwischen `p2` und `p1` gewertet werden.
- `ContactTracer.createPerson(int age)`. Diese Methode gibt ein `Person`-Objekt zurück. Das Alter der Person ist durch den `age`-Parameter bestimmt.

Alle `Person`-Objekte werden von der Methode `ContactTracer.createPerson(int age)` erstellt. Der `ContactTracer` wird über den parameterfreien Konstruktor `ContactTracer()` instanziiert. Sie dürfen annehmen, dass nie mehr als 1024 Begegnungen zwischen Personen protokolliert werden.

Implementieren Sie auf Basis dieser Vorlage eine Lösung für das Contact-Tracing-Problem. Tests finden Sie in der Datei "ContactTracerTest.java". Die Datei "ContactTracerGradingTest.java" enthält die Tests, welche wir bei der Prüfung für die Korrektur verwendet haben. Wir empfehlen, diese Tests erst zu verwenden, wenn Sie denken, dass Ihre Lösung korrekt ist, damit Sie sehen können, wie Sie bei einer Prüfung abgeschnitten hätten.

Aufgabe 5: Verkehrssystem (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

Ein Reisender plant, ein öffentliches Verkehrssystem zu nutzen, um von einer Startstation zu einer Zielstation zu gelangen. Die Klasse `TransportSystem` enthält eine Liste `List<Station> stations` von Stationen von der Abfahrtsstation bis zur Zielstation. Sie dürfen davon ausgehen, dass `stations` mindestens zwei Stationen enthält. Die Startstation ist durch `stations.getFirst()` gegeben, die Zielstation durch `stations.getLast()`. Jede `Station` enthält den Stationsnamen `name` (vom Typ `char`) und eine Liste von Verbindungen zu anderen Stationen `List<Connection> connections`. Die Klasse `Connection` enthält den Namen `name` der nächsten Station und die Streckenlänge `int distance` dieser Verbindung. Der Konstruktor `Connection(char name, int distance)` initialisiert dabei die Felder auf die entsprechenden Werte. Sie dürfen davon ausgehen, dass jede Station in der Liste `stations` eines Transportsystems einen einzigartigen Namen hat und dass jede Verbindung zu einer Station in dieser Liste führt. Die Verbindungen werden durch drei verschiedene Fahrzeugarten abgedeckt: Bus, Tram und Zug (Train). Die Klassen `Bus`, `Tram` und `Train` erweitern also die Klasse `Connection`. Beachten Sie, dass die Streckenlänge zweier Verbindungen zwischen denselben Stationen auch unterschiedlich sein kann, da zum Beispiel die Zuggleise gerade sind, aber die Strasse für den Bus viele Kurven hat. Die Reisezeit für eine Verbindung sind durch das Transportmittel und die Distanz `distance` wie folgt gegeben:

	Zeit (Minuten)
Bus	10 + max(0, (`distance` - 2) * 7)
Tram	20 + max(0, (`distance` - 5) * 5)
Zug	32 + max(0, (`distance` - 8) * 2)

a) Implementieren Sie die Methode `int getTime()` der Klassen `Bus`, `Tram` und `Train`. Diese Methode soll die Reisezeit der gegebenen Verbindung zurückgeben.

Hier sind einige Beispiele:

- `Bus bus = new Bus('A', 1);`
  `bus.getTime()` gibt 10 zurück.
- `Tram tram = new Tram('B', 6);`
  `tram.getTime()` gibt 25 zurück.
- `Train train = new Train('C', 10);`
  `train.getTime()` gibt 36 zurück.

b) Implementieren Sie die Methode `Route findFastestRoute()` in der Klasse `TransportSystem`, um die schnellste Route von der Abfahrtsstation zur Zielstation zu finden. Die Klasse `Route` enthält die gesamte Reisezeit `int time` in Minuten und die gewählten Verbindungen `List<Connection> connections` in der Reihenfolge, wie ein Reisender sie nehmen soll, um vom Start zum Ziel zu gelangen. Die zurückgegebene Instanz der Klasse `Route` soll also eine valide Reiseroute von Start zu Ziel in `connections` enthalten und `time` soll die dafür notwendige Reisezeit von Start zu Ziel enthalten. Für eine valide Reiseroute gilt, dass die erste Verbindung in `connections` der Startstation enthalten ist, für alle anderen Verbindungen die Verbindung jeweils in `connections` der Destination der zuvor gewählten Verbindung enthalten ist und die Destination der letzten Verbindung die Zielstation ist. Von allen möglichen Reiserouten soll dabei eine beliebige Reiseroute mit minimaler Reisezeit gewählt werden. Falls keine Reiseroute vom Start zum Ziel existiert, soll die Methode `findFastestRoute()` `null` zurückgegeben.

Das Transportnetzwerk kann jeweils als ein gerichteter, gewichteter Graph betrachtet werden. Sie dürfen auch davon ausgehen, dass dieser Graph keine Zyklen enthält. Wenn es also eine Reiseroute von Station S zu Station T gibt, dann existiert keine Reiseroute von Station T zu Station S.

Die Reisezeit ist dabei die Summe der Reisezeiten der einzelnen Verbindungen, wobei an jeder Station ein bisschen Zeit gespart werden kann, falls das Transportmittel nicht gewechselt werden muss. Falls ein Reisender mit einem Bus an eine Station gelangt und dann mit einem Bus weiterreist, so werden 2 Minuten gespart. Das heisst, dass von der Reisezeit 2 Minuten abgezogen werden können. Beim Umsteigen von Tram zu Tram werden dabei 5 Minuten und von Zug zu Zug sogar 10 Minuten gespart.

Hinweis: Es wird nicht von Ihnen erwartet, dass Sie erlangtes Wissen aus anderen Vorlesungen zum Lösen dieser Aufgabe verwenden. Wenn Sie jedoch Erlerntes aus anderen Vorlesungen verwenden wollen, dann dürfen Sie das tun.

In der Datei `TransportSystemTest` finden Sie drei Tests zu dieser Methode. Zusätzlich finden Sie hier Visualisierungen zu diesen Tests in den Abbildungen 5-7, zusammen mit einer kurzen Erklärung.

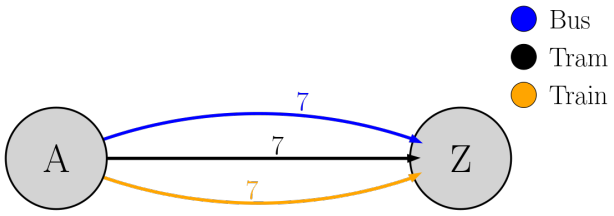

Abbildung 5: Beispielnetzwerk mit Startstation A und Zielstation Z

In Abbildung 5 hat die schnellste Route `route` eine Zeit von 30 Minuten (`route.time` == 30) und die Liste in `route.connections` soll wie folgt aussehen: `[Tram(Z, 7)]`. Das Tram ist also die schnellste Verbindung von Startstation A zu Zielstation Z.

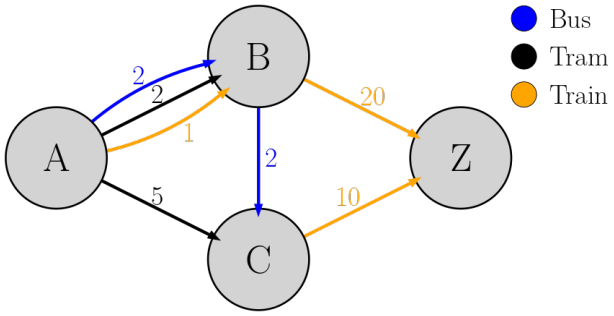

Abbildung 6: Beispielnetzwerk mit Startstation A und Zielstation Z

In Abbildung 6 hat die schnellste Route `route` eine Zeit von 54 Minuten (`route.time` == 54) und die Liste in `route.connections` soll wie folgt aussehen:

```
[Bus(B, 2), Bus(C, 2), Train(Z, 10)]
```

Diese Route beginnt also bei A, führt mit dem Bus zu B (10 Minuten), mit dem Bus weiter zu C (+10 Minuten) und dann mit dem Zug zu Z (+36 Minuten). Auf dem Weg zu Station B und von Station B weg wird beides mal ein Bus verwendet, daher wird die Reisezeit um 2 Minuten verkürzt ($10 - 2 + 10 + 36 = 54$).

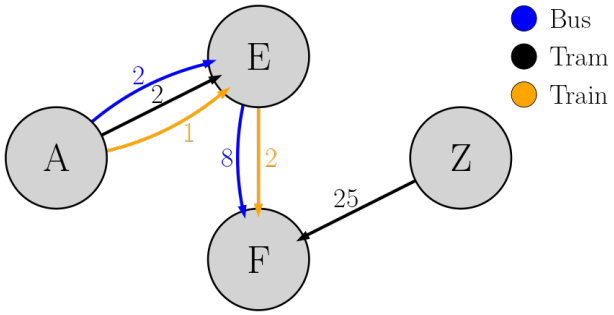

Abbildung 7: Beispielnetzwerk mit Startstation A und Zielstation Z

In Abbildung 7 existiert keine Reiseroute von A nach Z. Von A sind nur die Stationen E und F erreichbar. In diesem Fall soll `findFastestRoute()` `null` zurückgeben.