

☒ Gruppe 1 (J. Heinzelreiter)☐ Gruppe 2 (M. Hava)Name: Papesh KonstantinAufwand [h]: 16☐ Gruppe 3 (P. Kulczycki)

Übungsleiter/Tutor: _____

Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (30 P)	90	100	100
2 (50 P + 20 P)	100	100	100

Beispiel 1: Autobau (src/wmb/)

Sie werden von einem großen Münchner Autobauer beauftragt, die Verwaltung von Autobestandteilen softwareseitig zu unterstützen. Für ein Auto werden die folgenden Daten verwaltet: Typ, Farbe, Seriennummer, Produktionsdatum, Produktionsort, Getriebeart, Antriebsart, Höchstgeschwindigkeit und Gewicht. Für einen Motor werden folgende Daten benötigt: Motornummer, Treibstoffart, Leistung, Normverbrauch und Produktionsdatum. Für die Räder eines Autos werden Felgendurchmesser, Produktionsjahr, Geschwindigkeitsindex und Hersteller verwaltet.

Modellieren Sie mindestens die Klassen `auto`, `motor` und `rad` und implementieren Sie diese als C++-Klassen. „Bauen“ Sie Autos mit unterschiedlichen Konfigurationen zu Testzwecken zusammen. Es muss auch eine einfache Ausgabe von Fahrzeugdaten mittels `operator<<` möglich sein. Testen Sie ihre Implementierung ausführlich, lesen Sie alle benötigten Testdaten mittels `operator>>` ein.

Beispiel 2: ADT „Graph“ (src/adt/)

(a) Der von Ihnen in Übung 5 implementierte ADT für die Darstellung von ungewichteten gerichteten Graphen durch eine Adjazenzmatrix soll als Ausgangspunkt für eine objektorientierte Implementierung dienen, die dann allerdings gewichtete gerichtete Graphen repräsentieren kann. Gute Kandidaten für die zu implementierenden Klassen sind `vertex_t` und `graph_t`. Instanzen der Klasse `vertex_t` sind benannte Knoten, wobei die Namen Zeichenketten (`std::string`) sind. Die Klasse `graph_t` muss mindestens die folgende Funktionalität aufweisen:

```
handle_t      add_vertex (vertex_t vertex);    // moves 'vertex' into graph
void          add_edge   (handle_t const from, handle_t const to, int const weight);
std::ostream & print      (std::ostream & out) const;
```

Fügen Sie Ihren Klassen weitere notwendige Methoden und Datenkomponenten hinzu und testen Sie Ihre Implementierung ausführlich. Verwenden Sie dafür Streams zusammen mit `operator<<` und `operator>>`.

Anmerkung: Die Klasse `handle_t` identifiziert einen Vertex in einem Graphen eindeutig.

(b) Es gibt einen interessanten gierigen (*greedy*) Algorithmus zur Berechnung des kürzesten Wegs zwischen zwei Knoten in einem Graphen. Recherchieren Sie in der Algorithmenliteratur, suchen Sie nach dem Dijkstra-Algorithmus (*single-source shortest path*). Implementieren Sie in Ihrer Klasse `graph_t` eine Methode

```
int shortest_path (handle_t const from, handle_t const to) const;
```

die die Länge des kürzesten Wegs zwischen den beiden Knoten `from` und `to` nach dem Verfahren von Dijkstra ermittelt.

SWO31 & SWB31 Softwareentwicklung mit klassischen Sprachen und Bibliotheken – WS 2018/19

Übungsabgabe 7

Konstantin Papesh

3. Dezember 2018

7.1 Autobau

7.1.1 Lösungsidee

Grundsätzlich sollen drei Klassen erstellt werden *Auto*, *Motor* und *Rad*. Da das Programm auf Englisch ausgeführt wird heißen die Klassen repektiv *car*, *engine* und *wheel*.

main.cpp

In der Main wird zuerst ein Auto, ein Motor und ein Reifen erstellt und diese dann über die Konsole eingelesen. Darauffolgend wird das gesamte konstruierte Auto ausgegeben.

car

Grundsätzlich besteht *car* nur aus den verschiedenen Eigenschaften des Autos und zwei Hilfsfunktionen. Diese dienen dazu, jeweils einen Motor oder einen Reifen zum Auto hinzuzufügen. Weiters wird der *operator«* überladen, um die Ausgabe des Autos mithilfe von « zu ermöglichen. Wie alle anderen Klassen auch unterstützt diese Klasse das Einlesen der Parameter über *operator«*.

engine

Ähnlich aufgebaut wie *car.cpp*, nur ohne Hilfsfunktionen da zu *engine* keine weiteren Klassen hinzugefügt werden können.

wheel.cpp

Siehe *engine*

7.1.2 Implementierung

Listing 7.1: main.cpp

```

1 #include <iostream>
2 #include "car.h"
3 #include "engine.h"
4 #include "wheel.h"
5
6 int main() {
7     car car1("Auto", "Blue", 12, 1543280209, "Munich", "Automatic", "4x4", 212,
8             2200);
9     engine engine1(1, "Diesel", 120, 5.6, 1543750209);
10    wheel wheel;
11    std::cout << "Enter car type, colour, serial number, production date, production
12    location,"
13    " transmission type, driving mode, maximum speed and weight:" <<
14    std::endl;
15    std::cin >> car1;
16    std::cout << "Enter serial number, fuel type, horse power, consumption and
17    production date of engine:" << std::endl;
18    std::cin >> engine1;
19    std::cout << "Enter diameter, production year, speed index and manufacturer of
20    wheel:" << std::endl;
21    std::cin >> wheel;
22    car1.addEngine(engine1);
23    car1.addWheel(wheel);
24
25    std::cout << car1;
26 }
```

Listing 7.2: car.h

```

1 //
2 // Created by khp on 30.11.18.
3 //
4
5 #ifndef SRC_CAR_H
6 #define SRC_CAR_H
7
8 #include <string>
9 #include <vector>
10 #include "wheel.h"
11 #include "engine.h"
12
13 class car {
14     friend std::ostream &operator<<(std::ostream &, const car &);
15     friend std::istream &operator>>(std::istream&, car&);
16     friend void operator>>(wheel&, car&);
17     friend void operator>>(engine&, car&);
18
19 public:
20     car() = default;
21     car(std::string type, std::string colour, int serialNr, time_t prodDate, std::
22     string prodLoc,
23         std::string transmission, std::string drivingMode, int maxVelocity, int
24         weight);
```

```

23
24     void addEngine(engine);
25
26     void addWheel(wheel);
27
28 private:
29     std::string _type{""};
30     std::string _colour{""};
31     int _serialNr{0};
32     time_t _prodDate{0};
33     std::string _prodLoc{""};
34     std::string _transmission{""};
35     std::string _drivingMode{""};
36     int _maxVelocity{0};
37     int _weight{0};
38
39     engine _engine;
40     wheel _wheel;
41 };
42
43 std::ostream &operator<<(std::ostream &, const car &);
44 std::istream &operator>>(std::istream&, car&);
45 void operator>>(wheel&, car&);
46 void operator>>(engine&, car&);
47
48
49 #endif //SRC_CAR_H

```

Listing 7.3: car.cpp

```

1 //
2 // Created by khp on 30.11.18.
3 //
4
5 #include <iostream>
6 #include <iomanip>
7 #include "car.h"
8 #include "wheel.h"
9
10 car::car(std::string type, std::string colour, int serialNr, time_t prodDate, std::
    string prodLoc,
11         std::string transmission, std::string drivingMode, int maxVelocity, int
    weight) {
12     _type = type;
13     _colour = colour;
14     _serialNr = serialNr;
15     _prodDate = prodDate;
16     _prodLoc = prodLoc;
17     _transmission = transmission;
18     _drivingMode = drivingMode;
19     _maxVelocity = maxVelocity;
20     _weight = weight;
21 }
22
23 void car::addEngine(engine engine) {
24     _engine = engine;
25 }

```

```

26
27 void car::addWheel(wheel wheel) {
28     _wheel = wheel;
29 }
30
31 std::ostream &operator<<(std::ostream &os, const car &car) {
32     os << "Car type = " << car._type << std::endl
33     << "Colour = " << car._colour << std::endl
34     << "Serial number = " << car._serialNr << std::endl
35     << "Production date = " << std::put_time(std::localtime(&car._prodDate), "%c
    %Z") << std::endl
36     << "Production location = " << car._prodLoc << std::endl
37     << "Transmission type = " << car._transmission << std::endl
38     << "Driving mode = " << car._drivingMode << std::endl
39     << "Maximum speed = " << car._maxVelocity << " kph" << std::endl
40     << "Weight = " << car._weight << " kg" << std::endl;
41     os << car._engine << std::endl;
42     os << car._wheel << std::endl;
43     return os;
44 }
45
46 std::istream &operator>>(std::istream &is, car &car) {
47     is >> car._type >> car._colour >> car._serialNr >> car._prodDate >> car._prodLoc
    >> car._transmission >> car._drivingMode >> car._maxVelocity >> car._weight;
48     return is;
49 }
50
51 void operator>>(wheel &wheel, car &car) {
52     car.addWheel(wheel);
53 }
54
55 void operator>>(engine &engine, car &car) {
56     car.addEngine(engine);
57 }

```

Listing 7.4: engine.h

```

1 //
2 // Created by khp on 30.11.18.
3 //
4
5 #ifndef SRC_ENGINE_H
6 #define SRC_ENGINE_H
7
8 #include <string>
9
10 class engine {
11     friend std::ostream &operator<<(std::ostream&, const engine&);
12     friend std::istream &operator>>(std::istream&, engine&);
13 public:
14     engine() = default;
15     engine(unsigned int engineNum, std::string fuelType, unsigned int power, float
    consumption, time_t prodDate);
16
17 private:
18     unsigned int _engineNum{0};
19     std::string _fuelType{""};

```

```

20     unsigned int _power{0};
21     float _consumption{0.0};
22     time_t _prodDate{0};
23 };
24 std::ostream &operator<<(std::ostream&, const engine&);
25 std::istream &operator>>(std::istream&, engine&);
26
27 #endif //SRC_ENGINE_H

```

Listing 7.5: engine.cpp

```

1 //
2 // Created by khp on 30.11.18.
3 //
4
5 #include <iostream>
6 #include <ctime>
7 #include <iomanip>
8 #include "engine.h"
9
10 engine::engine(unsigned int engineNum, std::string fuelType, unsigned int power,
11               float consumption, time_t prodDate) {
12     _engineNum = engineNum;
13     _fuelType = fuelType;
14     _power = power;
15     _consumption = consumption;
16     _prodDate = prodDate;
17 }
18
19 std::ostream &operator<<(std::ostream &os, const engine &engine) {
20     if(engine._power == 0) { // NO POWER = NO ENGINE
21         os << "No engine selected" << std::endl;
22         return os;
23     }
24     os << "Engine number = " << engine._engineNum << std::endl
25        << "Fuel type = " << engine._fuelType << std::endl
26        << "Power = " << engine._power << " hp" << std::endl
27        << "Consumption = " << engine._consumption << " l/100km" << std::endl
28        << "Production date = " << std::put_time(std::localtime(&engine._prodDate),
29        "%c %Z") << std::endl << std::endl;
30     return os;
31 }
32
33 std::istream &operator>>(std::istream &is, engine &engine) {
34     is >> engine._engineNum >> engine._fuelType >> engine._power >> engine._consumption >> engine._prodDate;
35     return is;
36 }

```

Listing 7.6: wheel.h

```

1 //
2 // Created by khp on 30.11.18.
3 //
4
5 #ifndef SRC_WHEEL_H
6 #define SRC_WHEEL_H

```

```

7
8
9 #include <string>
10
11 class wheel {
12     friend std::ostream &operator<<(std::ostream&, const wheel&);
13     friend std::istream &operator>>(std::istream&, wheel&);
14 public:
15     wheel() = default;
16     wheel(unsigned int diameter, unsigned short int prodYear, char speedIndex, std::
        string manufacturer);
17
18 private:
19     unsigned int _diameter{0};
20     unsigned short int _prodYear{0};
21     char _speedIndex{};
22     std::string _manufacturer{""};
23 };
24 std::ostream &operator<<(std::ostream&, const wheel&);
25 std::istream &operator>>(std::istream&, wheel&);
26
27
28 #endif //SRC_WHEEL_H

```

Listing 7.7: wheel.cpp

```

1 //
2 // Created by khp on 30.11.18.
3 //
4
5 #include <iostream>
6 #include <sstream>
7 #include <vector>
8 #include <iterator>
9 #include "wheel.h"
10
11 #define DELIMITER ;
12
13 wheel::wheel(unsigned int diameter, unsigned short int prodYear, char speedIndex,
        std::string manufacturer) {
14     _diameter = diameter;
15     _prodYear = prodYear;
16     _speedIndex = speedIndex;
17     _manufacturer = manufacturer;
18 }
19
20 std::ostream &operator<<(std::ostream &os, const wheel &wheel) {
21     os << "Wheel diameter = " << wheel._diameter << std::endl
22     << "Production year = " << wheel._prodYear << std::endl
23     << "Speed index = " << wheel._speedIndex << std::endl
24     << "Manufacturer = " << wheel._manufacturer << std::endl;
25     return os;
26 }
27
28 std::istream &operator>>(std::istream &is, wheel &wheel) {
29     is >> wheel._diameter >> wheel._manufacturer >> wheel._speedIndex >> wheel.
        _prodYear;

```

```
30     return is;  
31 }
```


7.1.3 Testen

```

Enter car type, colour, serial number, production date, production location, transmission type, driving mode, maximum speed and weight:
auto blue 324243 1943/08/21 munich automatic 4x4 220 2200
Enter serial number, fuel type, horse power, consumption and production date of engine:
231232 diesel 250 5.6 1943/08/21
Enter diameter, production year, speed index and manufacturer of wheel:
18 2018 L Badmonth
Enter car type, colour, serial number, production date, production location, transmission type, driving mode, maximum speed and weight:
lkw red 4325345 1943/05/31 stuttgart manual 2 150 4300
Enter serial number, fuel type, horse power, consumption and production date of engine:
355436 diesel 450 15.6 1943/05/31
Enter diameter, production year, speed index and manufacturer of wheel:
22 1995 A Badmonth
Car 1:
Car type = auto
Colour = blue
Serial number = 324243
Production date = Sun Dec 2 22:22:11 2018 CET
Production location = munich
Transmission type = automatic
Driving mode = 4x4
Maximum speed = 220 kph
Weight = 2200 kg
Engine number = 231232
Fuel type = diesel
Power = 250 hp
Consumption = 5.6L/100km
Production date = Fri Nov 30 14:48:51 2018 CET

Wheel diameter = 18
Production year = 2018
Speed index = L
Manufacturer = Badmonth

Car 2:
Car type = lkw
Colour = red
Serial number = 4325345
Production date = Sun May 31 18:02:11 2009 CEST
Production location = stuttgart
Transmission type = manual
Driving mode = 2
Maximum speed = 150 kph
Weight = 4300 kg
Engine number = 355436
Fuel type = diesel
Power = 450 hp
Consumption = 15.6L/100km
Production date = Sun May 31 09:08:51 2009 CEST

Wheel diameter = 22
Production year = 1995
Speed index = A
Manufacturer = Badmonth

```

Abbildung 7.1: Beispielergabe für 2 Fahrzeuge
Grüner Text bedeutet Tastatureingabe

7.2 ADT

7.2.1 Lösungsidee

Die Implementierung der Matrix entspricht der Empfehlung der Angabe, es gibt daher drei Klassen:

1. `graph_t`
2. `handle_t`
3. `vertex_t`

`graph_t`

ist für die Verwaltung des Graphen, der Matrix und den darin enthaltenen Vertex's zuständig. Grundsätzlich besteht die Klasse aus öffentlichen Methoden und privaten Variablen bzw. Containern.

`addVertex` fügt einen Vertex dem Graphen hinzu und liefert einen `handle_t` zurück, dieser ist einmalig und ist für das erneute Bearbeiten des Vertex notwendig. In der Implementierung wird zuerst überprüft, ob der Handle einzigartig ist. Ist dies der Fall wird der Handle mit dem Vertex in eine `std::map` eingefügt. Dies ist eine Standardimplementierung und ermöglicht es, den Vertex mit dem Handle wieder schnell und einfach zu lokalisieren. Danach wird jeweils eine weitere Spalte an alle existierenden Zeilen gehängt und eine weitere Zeile am Ende hinzugefügt. Alle hinzugefügten Werte haben den Wert `0`. Letztendlich wird noch der Handle zu einem internen Vector hinzugefügt, die Position in diesem Vector entspricht der Spalte in der Matrix. Dies vereinfacht die Adressierung und das Verbinden der Vertexe.

`removeVertex` ist ähnlich implementiert wie `addVertex`, wobei natürlich nicht darauf geachtet werden muss, ein `handle_t` zu finden. Es wird lediglich in der Map gesucht, ob der gesuchte Handle sich überhaupt im Graphen befindet. Ist dies der Fall wird der Index dieses Handles ermittelt und daraufhin alle Spalten mit diesem Index gelöscht. Am Ende wird dann noch die Zeile mit diesem Index gelöscht und der Vertex selbst aus `_nodes`-Array und der Map gelöscht.

`addEdge` überprüft zuerst, ob sich sowohl *from* als auch *to* im Graph befinden. Ist dies der Fall werden die beiden Indexe der beiden Handles gesucht und dann der mitgegebene Weight an der geeigneten Position gesetzt.

`removeEdge` entspricht `addEdge`, nur dass statt eines Weights der Wert `0` eingetragen wird.

`print` printet den gesamten Graphen in einen gegebenen *ostream*. Dabei wird zuerst der `_nodes`-Vector ausgegeben da dieser die Vertexes in richtiger Reihenfolge enthält. Danach wird über die Adjazenzmatrix iteriert und das jeweilige Gewicht ausgegeben.

`getIndex` ist eine private Helferfunktion welcher den Index zu einem gegebenen *handle_t* zurückliefert. Wird kein Index gefunden, wird *INVALID_HANDLE* zurückgeliefert.

`getHandlesIndex` ist eine private Helferfunktion für den *getShortestPath*-Algorithmus. Dabei wird in einem Vector nach einem Keypair gesucht, welches als Key den mitgelieferten Handle hat. Wird das Keypair gefunden, wird der Index dieses Keypairs zurückgeliefert.

`getShortestPath` ist die Implementierung der *b*-Aufgabe. Dabei wird zuerst ein Vector erstellt mit allen Vertexen, die im Graphen enthalten sind. Diese gelten dann als *Noch-Nicht-Besucht*. Danach wird kontrolliert ob sowohl *from* als auch *to* im Graph enthalten sind. Ist dies der Fall wird über den Vector *unvisitedHandles* iteriert, und zwar so lange bis das letzte Element im Vector der gesuchte Handle ist. Dies bedeutet dann, dass die kürzeste Route gefunden wurde. Ist das letzte Element im Vector nicht das gesuchte *to*, wird der Index des aktuellen Elements gesucht. Dieser wird für die Routenlänge gebraucht. Danach wird in einer for-Schleife jeweils die momentane kürzeste Strecke zu einem Knoten abgefragt und die Strecke zu diesem Knoten über den momentanen Knoten berechnet. Ist die Route über den momentanen Knoten kleiner als der eingetragene Wert bei einem Knoten wird der Wert überschrieben.

`handle_t`

Dient zur Verwaltung der einzigartigen ID des Vertex in einem Graphen. Dabei besteht die Klasse selbst nur aus dieser ID, welche aber gegen Änderungen abgesichert ist.

`handle_t` ist der Konstruktor der Klasse. Diesem wird eine ID mitgegeben, welche er speichert. Diese ist nicht mehr änderbar sondern bleibt über die Existenz des Handle konstant.

`getID` liefert die ID zurück, da diese *private* ist um Veränderungen zu vermeiden.

`vertex_t`

Speichert nur die Payload des Knoten, in unserem Fall ein *std::string*. Weiters existieren *operator«* und *operator»*, um die Ein- und Ausgabe dieses Operators zu ermöglichen.

7.2.2 Implementierung

Listing 7.8: main.cpp

```

1 //
2 // Created by khp on 01.12.18.
3 //
4
5 #include "vertexT.h"
6 #include "graphT.h"
7

```

```

8 int main() {
9     graph_t graph;
10    graph.print(std::cout) << std::endl;
11    vertex_t vertex1;
12    vertex_t vertex2;
13    vertex_t vertex3;
14
15    vertex1.name = "A";
16    vertex2.name = "B";
17    vertex3.name = "C";
18
19    // Hinzufügen eines Vertexes zu einem Graphen.
20    handle_t handleV1 = graph.addVertex(vertex1);
21    handle_t handleV2 = vertex2 >> graph;
22    std::cout << graph << std::endl;
23
24    // Hinzufügen eines Edges zu einem Graphen
25    graph.addEdge(handleV1, handleV2, 200);
26    std::cout << graph << std::endl;
27
28    // Hinzufügen eines Vertexes und eines Edges zu einem Graphen.
29    handle_t handleV3 = graph.addVertex(vertex3);
30    graph.addEdge(handleV3, handleV1, 29);
31    std::cout << graph << std::endl;
32
33    // Entfernen eines Vertexes über den Handle
34    graph.removeVertex(handleV2);
35    std::cout << graph << std::endl;
36
37    // Es ist auch möglich einen Vertex öfter in einen Graphen zu geben.
38    handleV2 = graph.addVertex(vertex2);
39    handle_t handleV4 = graph.addVertex(vertex2);
40    graph.addEdge(handleV2, handleV4, 40);
41    graph.addEdge(handleV4, handleV2, 40);
42    std::cout << graph << std::endl;
43
44    // Berechnung des kürzesten Weges
45    std::cout << "Shortest connection C-A (29):" << graph.getShortestPath(handleV3,
46    handleV1) << std::endl;
47    std::cout << "Shortest connection B1-B2 (40):" << graph.getShortestPath(handleV2
48    , handleV4) << std::endl;
49    std::cout << "Shortest connection A-B (-1, NO PATH):" << graph.getShortestPath(
50    handleV1, handleV4) << std::endl;
51 }

```

Listing 7.9: graphT.h

```

1 //
2 // Created by khp on 01.12.18.
3 //
4
5 #ifndef SWO_GRAPHT_H
6 #define SWO_GRAPHT_H
7
8 #include <iostream>
9 #include <vector>
10 #include <map>

```

```

11 #include "handleT.h"
12 #include "vertexT.h"
13
14 class graph_t {
15     friend std::ostream &operator<<(std::ostream &os, const graph_t &graph);
16     friend handle_t operator>>(vertex_t &vertex, graph_t &graph);
17 public:
18     handle_t addVertex(vertex_t vertex);
19     void removeVertex(handle_t handle);
20     void addEdge(handle_t const from, handle_t const to, int const weight);
21     void removeEdge(handle_t const from, handle_t const to);
22     int getShortestPath(handle_t const from, handle_t const to) const;
23     std::ostream & print (std::ostream &out) const;
24
25 private:
26     long unsigned int getIndex(handle_t) const;
27     long unsigned int getHandlesIndex(handle_t handle, std::vector<std::pair<
        handle_t, int>> handles) const;
28     static bool sortByDist(const std::pair<handle_t, int> &a, const std::pair<
        handle_t, int> &b);
29     std::vector<std::vector<int>> _adjacencies{};
30     std::vector<handle_t> _nodes{};
31     std::map<unsigned long int, vertex_t> _handleMap{};
32 };
33 std::ostream &operator<<(std::ostream &os, const graph_t &graph);
34 handle_t operator>>(vertex_t &vertex, graph_t &graph);
35
36 #endif //SWO_GRAPH_T_H

```

Listing 7.10: graphT.cpp

```

1 //
2 // Created by khp on 01.12.18.
3 //
4
5 #include <algorithm>
6 #include "graphT.h"
7
8 #define INVALID_NODE -1
9 #define INVALID_HANDLE -2
10 #define NO_PATH INT16_MAX
11
12 handle_t graph_t::addVertex(vertex_t vertex) {
13     handle_t tHandle(rand());
14     while (_handleMap.find(tHandle.getID()) != _handleMap.end()) { // ID does not exist yet
15         handle_t tHandle2(rand());
16         tHandle = tHandle2;
17     }
18     _handleMap.insert({tHandle.getID(), vertex});
19     for (long unsigned int i = 0; i < _adjacencies.size(); ++i) {
20         _adjacencies[i].push_back(0);
21     }
22     std::vector<int> tVec(_adjacencies.size() + 1, 0);
23     _adjacencies.push_back(tVec);
24     _nodes.push_back(tHandle);
25     return tHandle;

```

```

26 }
27
28 void graph_t::removeVertex(handle_t handle) {
29     auto idx = _handleMap.find(handle.getID());
30     if (idx == _handleMap.end()) {
31         std::cout << "Vertex not found!" << std::endl;
32         return;
33     }
34
35     long unsigned int index = getIndex(handle);
36     for (long unsigned int i = 0; i < _adjacencies.size(); ++i) {
37         _adjacencies[i].erase(_adjacencies[i].begin() + index);
38     }
39     _adjacencies.erase(_adjacencies.begin() + index);
40     _nodes.erase(_nodes.begin() + index);
41     _handleMap.erase(handle.getID());
42 }
43
44 void graph_t::addEdge(handle_t const from, handle_t const to, int const weight) {
45     auto idxFrom = _handleMap.find(from.getID());
46     if (idxFrom == _handleMap.end()) {
47         std::cout << "Vertex from not found!" << std::endl;
48         return;
49     }
50
51     auto idxTo = _handleMap.find(to.getID());
52     if (idxTo == _handleMap.end()) {
53         std::cout << "Vertex to not found!" << std::endl;
54         return;
55     }
56
57     long unsigned int indexFrom = getIndex(from);
58     long unsigned int indexTo = getIndex(to);
59     _adjacencies[indexFrom][indexTo] = weight;
60 }
61
62 void graph_t::removeEdge(handle_t const from, handle_t const to) {
63     auto idxFrom = _handleMap.find(from.getID());
64     if (idxFrom == _handleMap.end()) {
65         std::cout << "Vertex from not found!" << std::endl;
66         return;
67     }
68
69     auto idxTo = _handleMap.find(to.getID());
70     if (idxTo == _handleMap.end()) {
71         std::cout << "Vertex to not found!" << std::endl;
72         return;
73     }
74     long unsigned int indexFrom = getIndex(from);
75     long unsigned int indexTo = getIndex(to);
76     _adjacencies[indexFrom][indexTo] = 0;
77 }
78
79 std::ostream &graph_t::print(std::ostream &out) const {
80     for (long unsigned int k = 0; k < _nodes.size(); ++k) {
81         auto idx = _handleMap.find(_nodes[k].getID());
82         out << idx->second.name;

```

```

83     }
84     out << std::endl;
85     for (long unsigned int i = 0; i < _adjacencies.size(); ++i) {
86         out << "|";
87         for (long unsigned int j = 0; j < _adjacencies.size(); ++j) {
88             out << _adjacencies[i][j] << "|";
89         }
90         out << std::endl;
91     }
92     return out;
93 }
94
95 std::ostream &operator<<(std::ostream &os, const graph_t &graph) {
96     graph.print(os);
97     return os;
98 }
99
100 long unsigned int graph_t::getIndex(handle_t handle) const {
101     for (long unsigned int i = 0; i < _nodes.size(); ++i) {
102         if (_nodes[i].getID() == handle.getID()) return i;
103     }
104     return INVALID_HANDLE;
105 }
106
107 long unsigned int graph_t::getHandlesIndex(handle_t handle, std::vector<std::pair<
    handle_t, int>> handles) const {
108     for (long unsigned int i = 0; i < handles.size(); ++i) {
109         if (handles[i].first.getID() == handle.getID())
110             return i;
111     }
112     return 0;
113 }
114
115 handle_t operator>>(vertex_t &vertex, graph_t &graph) {
116     handle_t tHandle = graph.addVertex(vertex);
117     return tHandle;
118 }
119
120 int graph_t::getShortestPath(handle_t const from, handle_t const to) const {
121     std::vector<std::pair<handle_t, int>> unvisitedHandles;
122
123     auto idxFrom = _handleMap.find(from.getID());
124     if (idxFrom == _handleMap.end()) {
125         std::cout << "Vertex from not found!" << std::endl;
126         return INVALID_NODE;
127     }
128
129     auto idxTo = _handleMap.find(to.getID());
130     if (idxTo == _handleMap.end()) {
131         std::cout << "Vertex to not found!" << std::endl;
132         return INVALID_NODE;
133     }
134
135     for (long unsigned int i = 0; i < _nodes.size(); ++i) {
136         if (_nodes[i].getID() != from.getID())
137             unvisitedHandles.push_back(std::make_pair(_nodes[i], NO_PATH));
138     }

```

```

139     unvisitedHandles.push_back(std::make_pair(_nodes[getIndex(from)], 0));
140
141     while (!unvisitedHandles.empty()) {
142         auto curItem = unvisitedHandles.back();
143         unvisitedHandles.pop_back();
144
145         if (curItem.first.getID() == to.getID())
146             return ((curItem.second == NO_PATH) ? -1 : curItem.second);
147
148         long unsigned int curIdx = getIndex(curItem.first);
149         for (long unsigned int i = 0; i < _adjacencies.size(); ++i) {
150             long unsigned int nextIdx = getHandlesIndex(_nodes[i], unvisitedHandles)
151             ;
152             if (nextIdx != INVALID_HANDLE) {
153                 int alt = curItem.second + _adjacencies[curIdx][i];
154                 if (alt != 0 && unvisitedHandles[nextIdx].second > alt) {
155                     unvisitedHandles[nextIdx].second = alt;
156                 }
157             }
158         }
159         return NO_PATH;
160     }

```

Listing 7.11: handleT.h

```

1  //
2  // Created by khp on 01.12.18.
3  //
4
5  #ifndef SWO_HANDLELET_H
6  #define SWO_HANDLELET_H
7
8
9  class handle_t {
10 public:
11     handle_t(unsigned long int id);
12     unsigned long int getID() const;
13
14 private:
15     unsigned long int id{0};
16 };
17
18
19 #endif //SWO_HANDLELET_H

```

Listing 7.12: handleT.cpp

```

1  //
2  // Created by khp on 01.12.18.
3  //
4
5  #include "handleT.h"
6
7  handle_t::handle_t(unsigned long int id) {
8      this->id = id;
9  }

```



```

10
11 unsigned long int handle_t::getID() const{
12     return this->id;
13 }

```

Listing 7.13: vertexT.h

```

1  //
2  // Created by khp on 01.12.18.
3  //
4
5  #ifndef SWO_VERTEX_H
6  #define SWO_VERTEX_H
7
8
9  #include <string>
10
11 class vertex_t {
12     friend std::ostream &operator<<(std::ostream &os, const vertex_t &vertex);
13     friend std::istream &operator>>(std::istream &is, vertex_t &vertex);
14
15 public:
16     std::string name{""};
17
18 };
19 std::ostream &operator<<(std::ostream &os, const vertex_t &vertex);
20 std::istream &operator>>(std::istream &is, vertex_t &vertex);
21
22 #endif //SWO_VERTEX_H

```

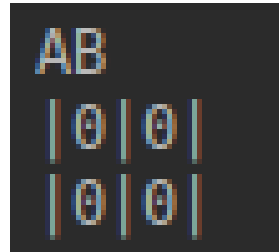
Listing 7.14: vertexT.cpp

```

1  //
2  // Created by khp on 01.12.18.
3  //
4
5  #include "vertexT.h"
6
7 std::ostream &operator<<(std::ostream &os, const vertex_t &vertex) {
8     os << vertex.name;
9     return os;
10 }
11
12 std::istream &operator>>(std::istream &is, vertex_t &vertex) {
13     is >> vertex.name;
14     return is;
15 }

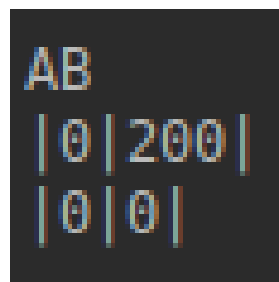
```

7.2.3 Testen



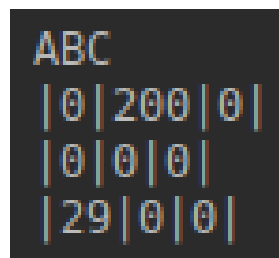
AB	
0 0	
0 0	

Abbildung 7.2: Hinzufügen eines Vertexes zu einem Graphen



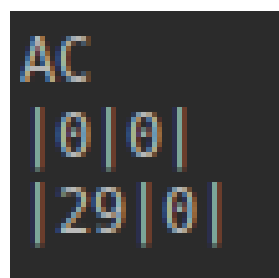
AB	
0 200	
0 0	

Abbildung 7.3: Hinzufügen eines Edges zu einem Graphen



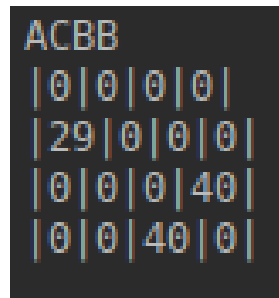
ABC		
0 200 0		
0 0 0		
29 0 0		

Abbildung 7.4: Hinzufügen eines Vertexes und eines Edges zu einem Graphen



AC	
0 0	
29 0	

Abbildung 7.5: Entfernen eines Vertexes über den Handle



ACBB				
0	0	0	0	0
29	0	0	0	0
0	0	0	40	0
0	0	40	0	0

Abbildung 7.6: Es ist auch möglich einen Vertex öfter in einen Graphen zu geben

```
Shortest connection C-A (29):29
Shortest connection B1-B2 (40):40
Shortest connection A-B (-1, NO PATH):-1
```

Abbildung 7.7: Berechnung der kürzesten Distanz zwischen zwei Vertexen