

<input type="checkbox"/> Gr. 1, DI Franz Gruber-Leitner	Name <u>Roman Lumetsberger</u>	Aufwand in h <u>7</u>
<input type="checkbox"/> Gr. 2, Dr. Erik Pitzer	Punkte _____	Kurzzeichen Tutor / Übungsleiter _____ / _____

Graphen, die Zweite**(14 + 6 + 4 Punkte)**

- a) Die von Ihnen in der Übung 3 realisierte Implementierung einer abstrakten Datenstruktur (ADS) für die Darstellung von Graphen durch eine Adjazenzmatrix soll als Ausgangspunkt für eine objektorientierte Implementierung dienen, die gewichtete Graphen repräsentieren kann.

Gute Kandidaten für die zu implementierenden Klassen sind *Vertex* (für die Knoten) und *Graph* (für die Darstellung von gewichteten gerichteten Graphen in Form der Adjazenzmatrix).

Objekte der Klasse *Vertex* sind benannte Knoten, wobei deren Namen beliebige Zeichenketten sein können.

Objekte der Klasse *Graph* verwalten alle relevanten Informationen für einen Graphen (mit einer maximalen Anzahl *max* von Knoten). Die Klasse *Graph* sollte daher mindestens folgende Funktionalität (in Form von Methoden) aufweisen:

```
void addVertex(Vertex *v); // reports an error if more than max vertices
void addEdge(const Vertex *start, const Vertex *end, double weight);
// reports an error if vertices not previously added or weight <= 0
```

Darüber hinaus, soll noch der Ausgabeoperator überladen werden um Graphen auszugeben, sowie Methoden um entweder nur die Kanten, oder sowohl alle Knoten als auch alle Kanten zu löschen.

Fügen Sie Ihren Klassen je nach Bedarf weitere notwendige Methoden und/oder Datenkomponenten hinzu und testen Sie Ihre Implementierung ausführlich.

- b) So wie es auf Bäumen verschiedene Durchlaufstrategien gibt (*in*, *pre* oder *post order*) gibt es auch auf Graphen zwei Strategien, alle Knoten in einer definierten Reihenfolge zu besuchen, die Tiefen- und die Breiten-„Suche“. Machen Sie sich dazu in der Algorithmenliteratur schlau und implementieren Sie beide Strategien in Form von Methoden:

```
void printDepthFirst(const Vertex *start) const;
void printBreadthFirst(const Vertex *start) const;
```

Fügen Sie Ihren Klassen je nach Bedarf weitere notwendige Methoden und/oder Datenkomponenten hinzu und testen Sie Ihre Implementierung ausführlich.

- c) Abschließen entwickeln Sie noch eine Methode um für einen beliebigen Graphen zu prüfen, ob er Zyklen enthält. Ob es also mindestens einen Konten gibt, von dem aus es möglich ist, über eine Folge von Kanten, wieder zum Ausgangspunkt zurück zu kehren:

```
bool hasCycles() const;
```

1 Aufgabe 1 - Graph - Objektorientiert

1.1 Lösungsidee

1.1.1 Klasse Vertex

Es wird eine Klasse *Vertex* benötigt, die als Datenkapsel für den Kontennamen (*string*) dient. Weiters wird der Gleichheitsoperator und der Ausgabeoperator benötigt.

1.1.2 Klasse Graph

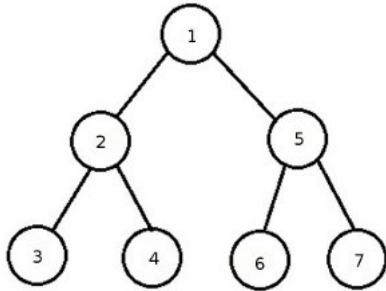
Diese Klasse enthält die Adjazenzmatrix und ein Array mit den Konten (*Vertex*) des Graphen.

- Beim Konstruktor wird die Matrix und das Array der Knoten allokiert.
- Beim Hinzufügen von Knoten wird in das Array der Konten eingefügt.
- Beim Hinzufügen von Kanten wird das Gewicht in die Matrix eingetragen.
- Beim Löschen der Kanten wird die Matrix wieder auf 0 gesetzt.
- Beim Löschen der Knoten wird die Matrix auf 0 gesetzt und alle Array-Elemente der Knoten auf *nullptr* gesetzt.
- Im Destruktor wird die Matrix und das Array wieder gelöscht.

2 Aufgabe 2 - Graph - Durchlaufstrategien

2.1 Tiefensuche

Die Tiefensuche arbeitet sich zuerst bis in die tiefste Ebene vor und besucht danach die Knoten von abzweigenden Pfaden.



2.1.1 Algorithmus

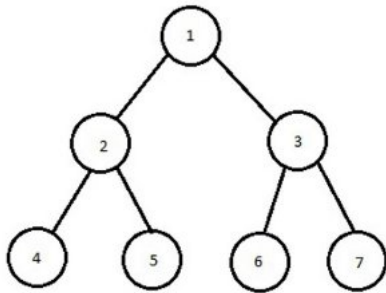
Dazu wird ein Stack benötigt, dieser wird als eigene Hilfsklasse *VertexStack* implementiert.

1. Das Startelement wird auf den Stack gelegt und als **besucht** markiert.

2. Solange der Stack nicht leer ist, wird das oberste Element des Stacks betrachtet (*peek*) und die nächste Kante gesucht.
3.
 - a) Wird eine Kante zu einem Knoten gefunden, der noch nicht besucht wurde, wird dieser als **besucht** markiert und auf den Stack gelegt (*push*).
 - b) Wird keine weitere Kante gefunden, wird das Element vom Stack entfernt (*pop*).

2.2 Breitensuche

Die Breitensuche besucht zuerst alle vom Startpunkt ausgehenden Knoten und arbeitet sich danach zu den tieferen Ebenen durch.



2.2.1 Algorithmus

Dazu wird eine Queue benötigt, diese wird als eigene Hilfsklasse *VertexQueue* implementiert.

1. Das Startelement wird als **besucht** markiert und alle erreichbaren Knoten in die Queue eingefügt (*enqueue*) und ebenfalls als **besucht** markiert.
2. Solange die Queue nicht leer ist, wird das nächste Element aus der Queue entfernt (*dequeue*) und die nächste Kante gesucht.
3. Wird eine Kante zu einem Knoten, der noch nicht besucht wurde, gefunden, wird dieser als **besucht** markiert und in die Queue eingefügt (*enqueue*).

3 Aufgabe 3 - Graph - Zyklen

Um Zyklen in einem Graphen zu erkennen, kann der Algorithmus der Tiefensuche verwendet werden und falls es eine Kante zu einem Knoten gibt, der sich bereits im Stack befindet, dann wurde ein Zyklus gefunden. Dieses Verfahren muss für jeden Knoten als Startknoten wiederholt werden, falls noch kein Zyklus gefunden wurde.

3.1 Sourcecode

Vertex.h

```
1  /*****
2   Vertex.h
3   Roman Lumetsberger
4
5   Header for class Vertex
6   *****/
7  #ifndef VERTEX_H
8  #define VERTEX_H
9
10 #include <string>
11
12 class Vertex
13 {
14     private:
15         std::string name;
16     public:
17         Vertex(std::string name);
18         virtual ~Vertex();
19
20         bool operator== (const Vertex& right) const;
21         friend std::ostream& operator << (std::ostream& os, const Vertex &v);
22     private:
23 };
24
25 #endif // VERTEX_H
```

VertexList.h

```
1  /*****
2   VertexList.h
3   Roman Lumetsberger
4
5   Defines the vertex list struct
6   Used for VertexStack and VertexQueue
7   *****/
8  #ifndef VERTEX_LIST_H
9  #define VERTEX_LIST_H
10
11 #include "Vertex.h"
12
13 //declares the list element data structure
14 struct VertexList {
15     const Vertex *data;
16     VertexList *next;
```

```
17 };
18
19 #endif
```

VertexQueue.h

```
1  /*****
2   VertexQueue.h
3   Roman Lumetsberger
4
5   Header for class VertexQueue
6   Defines a Queue of vertices
7   Only used internally
8   *****/
9  #ifndef VERTEXQUEUE_H
10 #define VERTEXQUEUE_H
11
12 #include "Vertex.h"
13 #include "VertexList.h"
14
15 /* minimal queue implementation */
16 class VertexQueue
17 {
18     private:
19         VertexList *start;
20         //used for adding items at the end
21         VertexList *end;
22     public:
23         VertexQueue();
24         VertexQueue(const VertexQueue &queue);
25         virtual ~VertexQueue();
26
27         void enqueue(const Vertex* item);
28         const Vertex* dequeue();
29
30         bool isEmpty() const {return start == nullptr;}
31 };
32
33 #endif // VERTEXQUEUE_H
```

VertexStack.h

```
1  /*****
2   VertexStack.h
3   Roman Lumetsberger
4
5   Header for class VertexStack
```

```
6   Defines a Stack of vertices
7   Only used internally
8   *****/
9   #ifndef VERTEXSTACK_H
10  #define VERTEXSTACK_H
11
12  #include "Vertex.h"
13  #include "VertexList.h"
14
15  /* minimal stack implementation */
16  class VertexStack
17  {
18  private:
19      VertexList *start;
20  public:
21      VertexStack();
22      VertexStack(const VertexStack &stack);
23      virtual ~VertexStack();
24
25      void push(const Vertex *vertex);
26      const Vertex *pop();
27      const Vertex *peek();
28      bool contains(const Vertex *vertex) const;
29      bool isEmpty() const {return start == nullptr;}
30
31  };
32
33  #endif // VERTEXSTACK_H
```

Graph.h

```
1  /*****
2   graph.h
3   Roman Lumetsberger
4
5   Header for class Graph
6   *****/
7  #ifndef GRAPH_H
8  #define GRAPH_H
9
10 #include <iostream>
11 #include "Vertex.h"
12
13 class Graph
14 {
15 private:
16     int maxVertices;
17     int vertexCount;
```

```
18
19     double *matrix;
20     Vertex **vertexArray;
21
22     int getVertexId(const Vertex& v) const;
23     bool hasEdge(const Vertex& first, const Vertex& second) const;
24     double getWeight(const Vertex& first, const Vertex& second) const;
25
26 public:
27     Graph(int maxVertices);
28     Graph(const Graph& graph); //copy constructor needed because of dynamic memory
29     virtual ~Graph();
30
31     void addVertex(Vertex *v);
32     void addEdge(const Vertex *start, const Vertex *end, double weight);
33     void removeEdges();
34     void removeVertices(); //does not delete the vertices
35
36     void printDepthFirst(const Vertex *start) const;
37     void printBreadthFirst(const Vertex *start) const;
38     bool hasCycles() const;
39
40     Graph& operator= (const Graph &g); //assignment operator needed because of dynamic memory
41     friend std::ostream& operator << (std::ostream &os, const Graph &g);
42
43 };
44
45 #endif // GRAPH_H
```

Vertex.cpp

```
1 /******
2 Vertex.cpp
3 Roman Lumetsberger
4
5 Implementation of the Vertex class
6 *****
7 #include <string>
8 #include <iostream>
9
10 #include "Vertex.h"
11 using namespace std;
12
13 Vertex::Vertex(string name) : name(name) {}
14
15 Vertex::~Vertex() {} // nothing todo here
16
17 bool Vertex::operator== (const Vertex& right) const {
```

```
18     return name == right.name;
19 }
20
21 ostream& operator <<(ostream& os, const Vertex& v) {
22     os << v.name;
23     return os;
24 }
```

VertexQueue.cpp

```
1  /*****
2   VertexQueue.cpp
3   Roman Lumetsberger
4
5   Implementation of VertexQueue class
6   *****/
7  #include "VertexQueue.h"
8  #include "Vertex.h"
9  #include <cassert>
10
11 VertexQueue::VertexQueue() :start(nullptr), end(nullptr){}
12
13 //copy constructor
14 VertexQueue::VertexQueue(const VertexQueue &queue) :VertexQueue() {
15     if(queue.isEmpty()) return; //Nothing to do
16
17     VertexList *item = queue.start;
18     //loop through all items an enqueue them
19     while(item != nullptr) {
20         enqueue(item->data);
21         item = item->next;
22     }
23 }
24
25 VertexQueue::~VertexQueue() {
26     //remove all items
27     while(!isEmpty())
28         dequeue();
29 }
30
31
32 void VertexQueue::enqueue(const Vertex* item) {
33     VertexList *element=new VertexList();
34     element->data = item;
35     element->next = nullptr;
36
37     //first element
38     if(start == nullptr) {
39         start = element;
```



```
40 }
41 else {
42     end->next = element;
43 }
44 end = element;
45 }
46
47 const Vertex *VertexQueue::dequeue() {
48     assert(start != nullptr);
49     VertexList *listItem = start;
50     start = start->next;
51     const Vertex* item = listItem->data;
52     delete listItem;
53
54     /* if the queue is empty set end to the nullptr */
55     if(start == nullptr)
56         end = nullptr;
57
58     return item;
59 }
```

VertexStack.cpp

```
1  /*****
2   VertexStack.cpp
3   Roman Lumetsberger
4
5   Implementation of VertexStack class
6   *****/
7  #include <cassert>
8
9  #include "VertexStack.h"
10 #include "Vertex.h"
11 #include "VertexList.h"
12
13 VertexStack::VertexStack() : start(nullptr) {}
14
15 //Copy constructor
16 VertexStack::VertexStack(const VertexStack& stack):VertexStack() {
17     if(stack.isEmpty()) return; //nothing to do
18
19     start = new VertexList();
20     start->data = stack.start->data;
21     start->next = nullptr;
22
23     VertexList *currentItem = start;
24     VertexList *item = stack.start->next;
25
26     //loop through all items and copy them
```

```
27 while(item != nullptr) {
28     VertexList* newItem = new VertexList();
29     newItem->data = item->data;
30     newItem->next = nullptr;
31     currentItem->next = newItem;
32     currentItem = newItem;
33
34     item = item->next;
35 }
36 }
37
38 VertexStack::~VertexStack() {
39     // remove all items
40     while(!isEmpty())
41         pop();
42 }
43
44 void VertexStack::push(const Vertex* vertex) {
45     assert(vertex != nullptr);
46
47     VertexList *element = new VertexList();
48     element->data = vertex;
49     element->next = start;
50
51     start = element;
52 }
53
54 const Vertex* VertexStack::peek() {
55     assert(!isEmpty());
56     return start->data;
57 }
58
59
60 const Vertex *VertexStack::pop() {
61     assert(!isEmpty());
62     VertexList *firstListItem = start;
63     const Vertex *item = start->data;
64     start = start->next;
65     delete firstListItem;
66     return item;
67 }
68
69 bool VertexStack::contains(const Vertex* vertex) const {
70     VertexList *current = start;
71     while(current != nullptr) {
72         if( *(current->data) == *vertex) {
73             return true;
74         }
75         current = current->next;
```

```
76 }
77 return false;
78 }
```

Graph.cpp

```
1  /*****
2   Graph.cpp
3   Roman Lumetsberger
4
5   Implementation of the Graph class
6   *****/
7  #include <cassert>
8  #include <cstring>
9  #include <iostream>
10
11 #include "Graph.h"
12 #include "Vertex.h"
13 #include "VertexStack.h"
14 #include "VertexQueue.h"
15
16 using namespace std;
17
18 Graph::Graph(int maxVertices) :maxVertices(maxVertices), vertexCount(0) {
19     matrix = new double[maxVertices*maxVertices]();
20     vertexArray = new Vertex*[maxVertices]();
21 }
22
23 /* copy constructor needed because of dynamic memory */
24 Graph::Graph(const Graph& graph) : Graph(graph.maxVertices) {
25     //copy matrix data
26     memcpy(matrix, graph.matrix, sizeof(double) * maxVertices * maxVertices);
27     //copy vertex pointers
28     memcpy(vertexArray, graph.vertexArray, sizeof(Vertex*) * maxVertices);
29     //copy vertexCount
30     vertexCount = graph.vertexCount;
31 }
32
33 Graph::~Graph() {
34     delete[] matrix;
35     delete[] vertexArray;
36 }
37
38 /* assignment operator needed because of dynamic memory */
39 Graph& Graph::operator=(const Graph &graph) {
40     if(this == &graph) return *this; //do nothing on x = x
41
42     delete[] matrix;
43     delete[] vertexArray;
```

```
44 maxVertices = graph.maxVertices;
45 matrix = new double[maxVertices*maxVertices];
46 vertexArray = new Vertex*[maxVertices];
47
48 //copy matrix data
49 memcpy(matrix, graph.matrix, sizeof(double) * maxVertices*maxVertices);
50 //copy vertex pointers
51 memcpy(vertexArray, graph.vertexArray, sizeof(Vertex*) * maxVertices);
52 vertexCount = graph.vertexCount;
53 return *this;
54 }
55
56 void Graph::addVertex(Vertex *v) {
57     assert(v != nullptr);
58     if(vertexCount == maxVertices) {
59         cerr << "max vertex count (" << maxVertices << ") already reached" << endl;
60         cerr << "vertex not added to graph" << endl;
61         return;
62     }
63     vertexArray[vertexCount] = v;
64     vertexCount++;
65 }
66
67 void Graph::addEdge(const Vertex *start, const Vertex *end, double weight) {
68     assert(start != nullptr);
69     assert(end != nullptr);
70
71     if(weight <= 0) {
72         cerr << "Weight has to be greater than 0" << endl;
73         return;
74     }
75
76     int startId = getVertexId(*start);
77     int endId = getVertexId(*end);
78
79     if(startId == -1) {
80         cerr << "Start vertex " << (*start) << " is not part of the graph" << endl;
81         return;
82     }
83
84     if(endId == -1) {
85         cerr << "End vertex " << (*end) << " is not part of the graph" << endl;
86         return;
87     }
88     matrix[(startId * maxVertices)+endId] = weight;
89 }
90
91 void Graph::removeEdges() {
92     memset(matrix, 0, sizeof(double) * maxVertices*maxVertices);
```

```
93 }
94
95 void Graph::removeVertices() {
96     removeEdges();
97     for(int i = 0; i < maxVertices; i++) {
98         vertexArray[i] = nullptr;
99     }
100     vertexCount = 0;
101 }
102
103 /* Gets the array index for the given vertex
104     Used for operating with the matrix
105 */
106 int Graph::getVertexId(const Vertex& v) const {
107     for(int i = 0; i < vertexCount; i++) {
108         if(*vertexArray[i] == v) {
109             return i;
110         }
111     }
112     return -1;
113 }
114
115 /* gets the weight out of the matrix
116     no check for vertices here because the method is only used internally
117 */
118 double Graph::getWeight(const Vertex& first, const Vertex& second) const {
119     return matrix[(getVertexId(first) * maxVertices) + getVertexId(second)];
120 }
121
122 bool Graph::hasEdge(const Vertex& first, const Vertex& second) const {
123     return getWeight(first, second) > 0;
124 }
125
126 ostream &operator<<(ostream& os, const Graph& g) {
127     for(int i = 0; i < g.vertexCount; i++) {
128         Vertex source = *g.vertexArray[i];
129         os << source << " ==> ";
130         for(int j = 0; j < g.vertexCount; j++) {
131             Vertex dest = *g.vertexArray[j];
132             if(g.hasEdge(source, dest)) {
133                 os << dest << "(" << g.getWeight(source, dest) << "),"";
134             }
135         }
136         os << endl;
137     }
138     return os;
139 }
140
141 /* Helper function used for depth and breath search */
```

```
142 void visit(Vertex* vertex) {
143     cout << " ==> " << (*vertex);
144 }
145
146 void Graph::printDepthFirst(const Vertex* start) const {
147     assert(start != nullptr);
148
149     bool *visitedArray = new bool[vertexCount]();
150     VertexStack stack;
151
152     stack.push(start);
153     cout << (*start);
154     int id = getVertexId(*start);
155     visitedArray[id] = true;
156
157     bool edgeFound;
158     while(!stack.isEmpty()) {
159         const Vertex *item = stack.peak();
160         edgeFound = false;
161         for(int i = 0; i < vertexCount; i++) {
162             if(hasEdge(*item, *vertexArray[i]) && !visitedArray[i]) {
163                 stack.push(vertexArray[i]);
164                 visit(vertexArray[i]);
165                 visitedArray[i] = true;
166                 edgeFound = true;
167                 break;
168             }
169         }
170         if (!edgeFound)
171             stack.pop();
172     }
173     cout << endl;
174     delete[] visitedArray;
175 }
176
177 void Graph::printBreadthFirst(const Vertex* start) const {
178     assert(start != nullptr);
179     bool *visitedArray = new bool[vertexCount]();
180     VertexQueue queue;
181
182     cout << (*start);
183     int id = getVertexId(*start);
184     visitedArray[id] = true;
185
186     for(int i = 0; i < vertexCount; i++) {
187         if(hasEdge(*start, *vertexArray[i])) {
188             queue.enqueue(vertexArray[i]);
189             visit(vertexArray[i]);
190             visitedArray[i] = true;
```

```
191     }
192 }
193
194 while(!queue.isEmpty()) {
195     const Vertex *item = queue.dequeue();
196
197     for(int i = 0; i < vertexCount; i++) {
198         if(hasEdge(*item, *vertexArray[i]) && !visitedArray[i]) {
199             queue.enqueue(vertexArray[i]);
200             visit(vertexArray[i]);
201             visitedArray[i] = true;
202         }
203     }
204 }
205 cout << endl;
206 delete[] visitedArray;
207 }
208
209 bool Graph::hasCycles() const {
210     bool cycleFound = false;
211
212     for(int i = 0; i < vertexCount; i++)
213     {
214         bool *visitedArray = new bool[vertexCount]();
215         VertexStack stack;
216         stack.push(vertexArray[i]);
217         visitedArray[i] = true;
218
219         bool edgeFound;
220         while(!stack.isEmpty()) {
221             const Vertex *item = stack.peek();
222             edgeFound = false;
223             for(int i = 0; i < vertexCount; i++) {
224                 if(hasEdge(*item, *vertexArray[i])) {
225                     if(!visitedArray[i]) {
226                         stack.push(vertexArray[i]);
227                         visitedArray[i] = true;
228                         edgeFound = true;
229                         break;
230                     }
231                     else if(stack.contains(vertexArray[i])) {
232                         cycleFound = true;
233                     }
234                 }
235             }
236             if (!edgeFound)
237                 stack.pop();
238         }
239         delete[] visitedArray;
```

```
240
241     //if we found a cycle we can stop the loop here
242     if(cycleFound)
243         break;
244 }
245 return cycleFound;
246 }
```

main.cpp

```
1  /*****
2   main.cpp
3   Roman Lumetsberger
4
5   Test program for Graph class
6 *****/
7  #include <iostream>
8  #include "Vertex.h"
9  #include "Graph.h"
10 #include "VertexStack.h"
11 #include "VertexQueue.h"
12
13 using namespace std;
14
15 int main(int argc, char *argv[])
16 {
17     if(argc != 2) {
18         cerr << "Wrong parameter count" << endl;
19         cerr << "Usage: " << argv[0] << " testcase";
20         return 0;
21     }
22
23
24     int testcase = atoi(argv[1]);
25     Graph emptyGraph(5);
26     Graph defaultGraph(6);
27
28     Vertex va("A");
29     Vertex vb("B");
30     Vertex vc("C");
31     Vertex vd("D");
32     Vertex ve("E");
33
34     Vertex vf("F");
35     Vertex vg("G");
36     defaultGraph.addVertex(&va);
37     defaultGraph.addVertex(&vb);
38     defaultGraph.addVertex(&vc);
39     defaultGraph.addVertex(&vd);
```



```
40     defaultGraph.addVertex(&ve);
41
42     switch(testcase) {
43     case 0:
44         cout << "Empty graph:" << endl;
45         cout << emptyGraph;
46         break;
47
48     case 1:
49         cout << "Graph without edges:" << endl;
50         cout << defaultGraph;
51         break;
52
53     case 2:
54         cout << "Vertex limit reached:" << endl;
55         defaultGraph.addVertex(&vf);
56         defaultGraph.addVertex(&vg);
57         break;
58
59     case 3:
60         cout << "Graph with edges:" << endl;
61         defaultGraph.addEdge(&va, &vb, 1);
62         defaultGraph.addEdge(&vb, &vb, 2);
63         defaultGraph.addEdge(&vc, &vd, 3);
64         defaultGraph.addEdge(&vb, &ve, 4);
65         defaultGraph.addEdge(&ve, &vd, 5);
66         cout << defaultGraph;
67
68         break;
69
70     case 4:
71         cout << "Invalid vertex/weight:" << endl;
72         defaultGraph.addEdge(&vf, &vb, 1);
73         defaultGraph.addEdge(&vb, &vf, 1);
74         defaultGraph.addEdge(&va, &vb, 0);
75         defaultGraph.addEdge(&va, &vb, -324);
76         break;
77
78     case 5:
79         cout << "Remove edges:" << endl;
80         defaultGraph.addEdge(&va, &vb, 1);
81         defaultGraph.removeEdges();
82         cout << defaultGraph;
83         break;
84
85     case 6:
86         cout << "Remove vertices/copy constructor:" << endl;
87         defaultGraph.addEdge(&va, &vb, 1);
88         //Use new block here for testing copy constructor
```

```
89     {
90         Graph savedGraph = defaultGraph;
91         defaultGraph.removeVertices();
92         cout << "cleared graph:" << endl;
93         cout << defaultGraph << endl;
94
95         savedGraph.addVertex(&vf);
96         cout << "saved graph:" << endl;
97         cout << savedGraph;
98     }
99     break;
100
101 case 7:
102     cout << "Depth first search:" << endl;
103     defaultGraph.addVertex(&vf);
104     defaultGraph.addEdge(&va, &vb, 1);
105     defaultGraph.addEdge(&va, &vc, 2);
106     defaultGraph.addEdge(&vb, &vd, 3);
107     defaultGraph.addEdge(&vc, &ve, 4);
108     defaultGraph.addEdge(&va, &vf, 5);
109     cout << defaultGraph << endl;
110     cout << "Depth first search start = " << va << endl;
111     defaultGraph.printDepthFirst(&va);
112
113     cout << "Depth first search start = " << vc << endl;
114     defaultGraph.printDepthFirst(&vc);
115     break;
116
117 case 8:
118     cout << "Breadth first search:" << endl;
119     defaultGraph.addVertex(&vf);
120     defaultGraph.addEdge(&va, &vb, 1);
121     defaultGraph.addEdge(&va, &vc, 2);
122     defaultGraph.addEdge(&vb, &vd, 3);
123     defaultGraph.addEdge(&vc, &ve, 4);
124     defaultGraph.addEdge(&va, &vf, 5);
125     cout << defaultGraph << endl;
126     cout << "Breadth first search start = " << va << endl;
127     defaultGraph.printBreadthFirst(&va);
128
129     cout << "Breadth first search start = " << vc << endl;
130     defaultGraph.printBreadthFirst(&vc);
131     break;
132
133 case 9:
134     cout << "HasCycles:" << endl;
135     defaultGraph.addEdge(&va, &vb, 1);
136     defaultGraph.addEdge(&vb, &vc, 2);
137     defaultGraph.addEdge(&vc, &va, 3);
```

```
138     cout << defaultGraph << endl;;
139     cout << "hasCycles = " << boolalpha << defaultGraph.hasCycles() << endl;
140     break;
141
142 case 10:
143     cout << "HasCycles II:" << endl;
144     defaultGraph.addEdge(&vc, &vd, 3);
145     defaultGraph.addEdge(&vd, &va, 3);
146     defaultGraph.addEdge(&vd, &vb, 3);
147     defaultGraph.addEdge(&vb, &ve, 3);
148     defaultGraph.addEdge(&ve, &vb, 3);
149     cout << defaultGraph << endl;;
150     cout << "hasCycles = " << boolalpha << defaultGraph.hasCycles() << endl;
151     break;
152
153 case 11:
154     cout << "No cycles:" << endl;
155     defaultGraph.addEdge(&va, &vb, 1);
156     defaultGraph.addEdge(&vb, &vc, 2);
157     defaultGraph.addEdge(&vc, &vd, 3);
158     cout << defaultGraph << endl;;
159     cout << "hasCycles = " << boolalpha << defaultGraph.hasCycles() << endl;
160     break;
161
162 case 12:
163     cout << "assignment operator" << endl;
164     defaultGraph.addEdge(&va, &vb, 1);
165     defaultGraph.addEdge(&vb, &vc, 2);
166     //new block to define the variable here
167     {
168         Graph newGraph(0);
169         newGraph = defaultGraph;
170         newGraph = newGraph; //self assignment
171         defaultGraph.removeVertices(); //remove vertices from original graph
172
173         newGraph.addVertex(&vf); //add F to the new graph
174         cout << "assigned graph: " << endl;
175         cout << newGraph;
176     }
177     break;
178
179 case 13:
180     cout << "Memory:" << endl;
181     defaultGraph.addEdge(&va, &vb, 1);
182     defaultGraph.addEdge(&vb, &vc, 2);
183     defaultGraph.addEdge(&vc, &vd, 3);
184     cout << defaultGraph << endl;;
185     cout << "hasCycles = " << boolalpha << defaultGraph.hasCycles() << endl;
186     defaultGraph.printBreadthFirst(&va);
```

```
187     defaultGraph.printBreadthFirst(&vc);
188     defaultGraph.removeEdges();
189     defaultGraph.removeVertices();
190     break;
191
192     default:
193         cerr << "invalid testcase";
194         break;
195
196     }
197
198
199     return 0;
200 }
```

3.2 Testfälle

3.2.1 Testfall 1 - Leerer Graph

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 0
Empty graph:
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.2 Testfall 2 - Graph ohne Kanten

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 1
Graph without edges:
A ==>
B ==>
C ==>
D ==>
E ==>
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.3 Testfall 3 - Max Knotenanzahl überschritten

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 2
Vertex limit reached:
max vertex count (6) already reached
vertex not added to graph
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.4 Testfall 4 - Graph mit Kanten

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 3
Graph with edges:
A ==> B(1),
B ==> B(2),E(4),
C ==> D(3),
D ==>
E ==> D(5),
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.5 Testfall 5 - Ungültiger Knoten / Gewicht

```
romanlum@ubuntu: ~/swo3/UebungModdle5/Beispiel/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ ./graph 4
Invalid vertex/weight:
Start vertex F is not part of the graph
End vertex F is not part of the graph
Weight has to be greater than 0
Weight has to be greater than 0
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ █
```

3.2.6 Testfall 6 - Kanten löschen

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 5
Remove edges:
A ==>
B ==>
C ==>
D ==>
E ==>
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.7 Testfall 7 - Knoten löschen / Kopierkonstruktor

```
romanlum@ubuntu: ~/swo3/UebungModdle5/Beispiel/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ ./graph 6
Remove vertices/copy constructor:
cleared graph:

saved graph:
A ==> B(1),
B ==>
C ==>
D ==>
E ==>
F ==>
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$
```

3.2.8 Testfall 8 - Tiefensuche

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 7
Depth first search:
A ==> B(1),C(2),F(5),
B ==> D(3),
C ==> E(4),
D ==>
E ==>
F ==>

Depth first search start = A
A ==> B ==> D ==> C ==> E ==> F
Depth first search start = C
C ==> E
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.9 Testfall 9 - Breitensuche

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 8
Breadth first search:
A ==> B(1),C(2),F(5),
B ==> D(3),
C ==> E(4),
D ==>
E ==>
F ==>

Breadth first search start = A
A ==> B ==> C ==> F ==> D ==> E
Breadth first search start = C
C ==> E
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.10 Testfall 10 - Zyklen

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 9
HasCycles:
A ==> B(1),
B ==> C(2),
C ==> A(3),
D ==>
E ==>

hasCycles = true
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.11 Testfall 11 - Zyklen II

```
romanlum@ubuntu: ~/swo3/UebungModdle5/Beispiel/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ ./graph 10
HasCycles II:
A ==>
B ==> E(3),
C ==> D(3),
D ==> A(3),B(3),
E ==> B(3),

hasCycles = true
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ █
```

3.2.12 Testfall 12 - Keine Zyklen

```
romanlum@ubuntu: ~/swo3/UebungModdle5/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ ./graph 11
HasCycles:
A ==> B(1),
B ==> C(2),
C ==> D(3),
D ==>
E ==>

hasCycles = false
romanlum@ubuntu:~/swo3/UebungModdle5/bin/Debug$ █
```

3.2.13 Testfall 13 - Zuweisungsoperator

```
romanlum@ubuntu: ~/swo3/UebungModdle5/Beispiel/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ ./graph 12
assignment operator
assigned graph:
A ==> B(1),
B ==> C(2),
C ==>
D ==>
E ==>
F ==>
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ █
```


3.2.14 Testfall 14 - Speichertest

```
romanlum@ubuntu: ~/swo3/UebungModdle5/Beispiel/bin/Debug
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ valgrind ./graph 13
==6627== Memcheck, a memory error detector
==6627== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6627== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6627== Command: ./graph 13
==6627==
Memory:
A ==> B(1),
B ==> C(2),
C ==> D(3),
D ==>
E ==>

hasCycles = false
A ==> B ==> C ==> D
C ==> D
==6627==
==6627== HEAP SUMMARY:
==6627==      in use at exit: 0 bytes in 0 blocks
==6627==    total heap usage: 33 allocs, 33 frees, 785 bytes allocated
==6627==
==6627== All heap blocks were freed -- no leaks are possible
==6627==
==6627== For counts of detected and suppressed errors, rerun with: -v
==6627== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
romanlum@ubuntu:~/swo3/UebungModdle5/Beispiel/bin/Debug$ █
```