1 Graphen in C++

1.1 Lösungsidee

Folgend ist die Lösungsidee für die Implementierung eines Graphen mit einer Adjazanzmatrix in C++ über Klassen angeführt.
In diesem Fall sollten zwei Klassen (Vertex, Graph) ausreichen um die Aufgabenstellung zu erfüllen.

Vertex:
Da Ein Vertex einmal hinzugefügt wird und daher seine Daten sich auch nicht ändern dürfen, soll diese Klasse so implementiert werden , sodass die Member nicht mehr überschrieben werden können. (const). Dadurch kann auch auf Getter und Setter Methoden verzichtet werden und die Member können mittels Public Keyword nach außen sichtbar gemacht werden. Damit soll eine final Verhalten der Instanz einer Klasse erreicht werden.
Da es nicht abzusehen ist ob mit diesen Klassen weitergearbeitet wird, also keine Vererbungen zu erwarten sind, werden diese Klassen nicht mit virtuellen Methoden versehen, da dies unnötig erscheint.

Graph:
Eine Klasse Graph soll einen Graphen mittels einer Adjazanzmatrix abbilden. Daher soll wie folgt vorgegangen werden um dies zu erreichen:

1. Ein zweidimensionales Array des Datentyps double soll die Kanten und dessen Gewichtung abbilden. (0 = keine Kante vorhanden)
2. Ein Vektor von Vertex soll alle eingefügten Knoten in der Ordnung, in der Sie eingefügt wurden, enthalten.
3. Da der Index eines Vertex im Vector gleich dem Index i und j in den beiden Dimensionen des zweidimensionalen Arrays entspricht, also ein Vertex in der Matrix, kann über diesen Index ein Vertex in der Matrix referenziert und so die Kanten abgebildet werden. (Einfügen, Löschen usw)

Des Weiteren sollen alle verlangten Methoden Implementiert und soweit abstrahiert werden sodass wiederholende Logik in Private Methoden abgebildet werden. Da in einem Vektor die Pointer der Vertex Instanzen gehalten werden kann man sich nicht auf einen eindeutigen Wert des Vertex als eindeutigen Identifier nicht verlassen. Daher soll darauf geachtet werden das geprüft wird, dass die Vertex Instanzen die einer Methode übergeben werden gültig sind und insbesondere Instanzen entsprechen, die in diesem Graphen verwaltet werden.

Bei der Breiten- und Tiefensuche soll rekursiv vorgegangen werden wobei hierbei besonders auf Zyklen in dem Graphen geachtet werden muss, da diese zu StackOverflowExceptions führen können, da man hier in eine Endlosrekursion gerät.

Breitensuche:

Bei der Breitensuche sollen alle Vertex die über die in der Matrix enthaltenen Knoten verknüpft sind auf einer Ebene ausgeben werden bevor man sich auf die nächste Ebene begibt und diese ausgibt.

Tiefensuche:

Bei der Tiefensuche soll man rekursiv sich bis zum letzten Knoten runterhanteln und gibt zuerst die Kindknoten aus bevor man beim Auflösen der Rekursion ihre Elternknoten ausgibt.

Diese Übung wird wie mit Professor Gegenleitner vereinbart, in Form eines Codeblocks Projekts abgegeben werden.

1.2 Implementierung

Folgend ist die Implementierung diese Aufgabe angeführt

1.2.1 graph.h

Folgend ist die Spezifikation der Klassen für Vertex und Graph angeführt.

Da die Klasse Vertex nur konstante Member hat auf welche nur lesend zugegriffen werden kann und es nur eine Friend operater Funktion enthält, wurde diese Klasse vollständig inline implementiert.

```
/*
 * graph.h
 *
 *  Created on: Dec 5, 2014
 *      Author: Thomas Herzog
 *
 *      This header file specifies the graph related classes.
 */

#ifndef GRAPH_H_
#define GRAPH_H_

#include <string>
#include <vector>
#include <iostream>
#include <sstream>
#include <ostream>

using namespace std;

namespace graph {
    /**
     * This class represents the graph nodes.
     *
     * Therefore that this class is considered to produce final instances,
     * the implementation of this class is provided inline.
     */
    class Vertex {

        public:
            ////////////////////////////////////////////////////////////
            // Public members                                         //
            ////////////////////////////////////////////////////////////
            const string name;

            ////////////////////////////////////////////////////////////
            // Constructor and Destructor                            //
            ////////////////////////////////////////////////////////////
            inline Vertex(const string name) :
                    name(name) {
            }

            ////////////////////////////////////////////////////////////
            // friend methods                                        //
            ////////////////////////////////////////////////////////////
            inline friend ostream& operator<<(ostream& os,
                    const Vertex& vertex) {
                os << vertex.name;
                return os;
            }
    };

    /**
     * This class represents the graph.
     *
     * This class specifies the graph which shall be implemented with an
     * adjazanzmatrix.
     */
    class Graph {

        private:
            ////////////////////////////////////////////////////////////
            // Private members                                       //
            ////////////////////////////////////////////////////////////
            /**
             * The matrix containing the edge weight information
             */
            double** matrix;

            /**
             * The vector containing all placed graph vertex nodes
             */
            vector<Vertex*> nodes;

            ////////////////////////////////////////////////////////////
            // Private methods                                       //
            ////////////////////////////////////////////////////////////
            /**
             * Gets the index of the vertex in the vector which is the same index as in the
matrix (i = j).
             *
             * @param
             *      node the node the get its index
             * @return
             *      the index of the node in the vector or -1 if the vertex instance is not
hold by this graph.
             */
            int getIdxForVertex(const Vertex* node) const;

            /**
             * Recursively walks through the graph from the given start vertex node until it
reaches the end o f
             * the tree or it detects an recursion
             *
             * @param
             *      node the start node to traverse from
             * @param
             *      visited the vector used as a stack for detecting already visited vertex
nodes.
             *
             * @return
             *      true if the graph has cycles false otherwise
             */
            bool hasCycles(Vertex* node, vector<Vertex*>* visited) const;

            /**
             * Recursively walks through the graph by crawling to the last elements of the
graph.
```

```
             *
             * @param
             *      node the node to start from
             *
             * @param
             *      visited the vector used as a stack to detect cycles in the graph
             */
            void deepTraversePrint(Vertex* node,
                    vector<Vertex*>* visited) const;

            /**
             * Recursively walks through the graph by visiting the all elements on the smae
level before
             * crawling to the next level.
             *
             * @param
             *      node the node to start from
             *
             * @param
             *      visited the vector used as a stack to detect cycles in the graph
             */
            void breathTraversePrint(Vertex* node,
                    vector<Vertex*>* visited) const;

            /**
             * Answers the question if the current visited vertex node is a valid vertex node
and hold by this graph.
             *
             * @param
             *      node the node to be checked
             *
             * @return
             *      true if the vertex node is valid, false otherwise
             */
            bool isManagedVertex(const Vertex* node) const;

        public:
            ////////////////////////////////////////////////////////////
            // Public members                                         //
            ////////////////////////////////////////////////////////////
            /**
             * The graph node max size. More nodes cannot be handled.
             */
            int maxSize;

            /**
             * Flag which indicates that loggin is intended.
             */
            bool log = false;

            ////////////////////////////////////////////////////////////
            // Constructor and Destructor                             //
            ////////////////////////////////////////////////////////////
            /**
             * Constructor for productive usage.
             *
             * @param
             *      maxSize the max size of graph nodes which this graph can handle.
             */
            Graph(const int maxSize);

            /**
             * Constructor for test usage if logging is intended
             *
             * @param
             *      log indicates if logging is intended
             * @param
             *      maxSize the max size of graph nodes which this graph can handle.
             */
            Graph(const int maxSize, const bool log);

            /**
             * Default destructor which cleans up all held resources.
             */
            ~Graph();
            ////////////////////////////////////////////////////////////
            // Public methods                                         //
            ////////////////////////////////////////////////////////////
            /**
             * Adds a vertex node to the graph.
             * This node will be added to the end of the matrix.
             *
             * @param
             *      node the node to be added
             */
            void addVertex(Vertex* node);

            /**
             * Adds a edge between two graph vertex nodes.
             *
             * @param
             *      startNode the vertex node which references a other node
             * @param
             *      endNode the vertex node which is referenced by the start node
             * @param
             *      weight the weight of the node
             */
            void addEdge(const Vertex* startNode, const Vertex* endNode,
                    double weight);

            /**
             * Clears all edges of the graph which leaves teh vertex nodes unconnected to each
other
             */
            void clearEdges();

            /**
             * Clears the graph by removing all edges nodes from the graph
             */
            void clear();

            /**
             * Prints the graph vertex nodes from the last level up
             *
             * @param
             *      start the vertex node to start from
             */
            void printDepthFirst(const Vertex* start) const;

            /**
             * Prints the graph vertex nodes from the top level down and prints all nodes on
the same level before
             * crawling to the next level
             *
             * @param
             *      start the vertex node to start from
             */
            void printBreadthFirst(const Vertex* start) const;

            /**
             * Answers the question if this graph has cycles or not.
             *
             * @return
             *      true if the graph has cycles false otherwise
             */
            bool hasCycles() const;

            ////////////////////////////////////////////////////////////
            // friend methods                                         //
            ////////////////////////////////////////////////////////////
            friend ostream& operator<<(ostream& os, const Graph& graph);
    };
}

#endif /* GRAPH_H_ */
```

1.2.2 graph.cpp

Folgend ist die Implementierung der Spezifikation der Klassen in der Header Datei graph.h angeführt. Es ist hier nur die Implementierung der Klasse Graph zu sehen, da die Klasse Vertex inline implementiert wurde und daher in der graph.h zu finden ist.

```
/*
 * edge.cpp
 *
 *  Created on: Dec 8, 2014
 *      Author: Thomas Herzog
 *
 *      This class contains the graph implementation of the graph.h specification.
 */
#include "graph.h"
#include <iostream>
#include <ostream>
#include <algorithm>
#include <cassert>

using namespace std;

namespace graph {
    ////////////////////////////////////////////////////////////
    // Constructor and Destructor                             //
    ////////////////////////////////////////////////////////////
    Graph::Graph(const int maxSize) :
            Graph(maxSize, false) {
    }

    Graph::Graph(const int maxSize, bool log) :
            maxSize(maxSize), log(log) {

        /* log constructor call */
        if (log) {
            cout << "constructor (" << maxSize << ", " << log << ") called"
                    << endl << flush;
        }
        /* validate maxSize */
        if (maxSize <= 0) {
            this->maxSize = 10;
            cout << "invalid maxSize !!! default set to '10'" << endl << flush;

        }

        /* Init first dimension */
        matrix = (double**) malloc(this->maxSize * sizeof(double**));
        if (matrix == nullptr) {
            exit(EXIT_FAILURE);
        } /* if */

        /* Init second dimension */
        for (int i = 0; i < this->maxSize; ++i) {
            matrix[i] = (double*) malloc(this->maxSize * sizeof(double*));
            if (matrix == nullptr) {
                exit(EXIT_FAILURE);
            } /* if */
        } /* for */

        /* set all edge values to 0 */
        clearEdges();

        /* Init the nodes vector */
        nodes.reserve(this->maxSize);
    }

    Graph::~Graph() {
        /* log destructor call */
        if (log) {
            cout << "destructor () called." << endl << flush;
        }

        /* delete vertex nodes */
        clear();
        /* delete matrix */
        for (int i = 0; i < maxSize; i++) {
            free(matrix[i]);
            matrix[i] = nullptr;
        } /* for */
        free(matrix);
        matrix = nullptr;
    }

    ////////////////////////////////////////////////////////////
    // Private methods                                        //
    ////////////////////////////////////////////////////////////
    int Graph::getIdxForVertex(const Vertex* node) const {
        size_t idx = -1;

        if (!nodes.empty()) {
            idx = 0;
            while ((idx < nodes.size()) && (nodes[idx] != node)) {
                idx++;
            } /* while */
            idx = (idx == nodes.size()) ? -1 : idx;
        } /* if */

        return idx;
    }

    bool Graph::hasCycles(Vertex* node, vector<Vertex*>* visited) const {
        bool invalid = false;

        /* add to call stack */
        visited->push_back(node);

        if ((find(visited->begin(), visited->end(), node) == visited->end())) {
            /* get first connected vertex index */
            size_t colIdx = 0;
            int rowIdx = getIdxForVertex(node);
            while ((!invalid) && (colIdx < nodes.size())) {
                /* visit next child */
                if (matrix[rowIdx][colIdx] != 0) {
                    invalid = hasCycles(nodes[colIdx], visited);
                } /* if */
                colIdx++;
            } /* while */
        } else {
            invalid = true;
        } /* if */

        /* remove from call stack */
        visited->pop_back();

        return invalid;
    }

    void Graph::deepTraversePrint(Vertex* node,
            vector<Vertex*>* visited) const {

        /* anchor for already visited */
        if ((find(visited->begin(), visited->end(), node) == visited->end())) {
            /* add to call stack */
            visited->push_back(node);

            /* print node */
            cout << "Visited: " << *node << endl << flush;

            /* get first connected vertex index */
            int rowIdx = getIdxForVertex(node);
            for (size_t i = 0; i < nodes.size(); ++i) {
                /* visit next child */
                if (matrix[rowIdx][i] != 0) {
                    deepTraversePrint(nodes[i], visited);
                } /* if */
            } /* while */

            /* remove from call stack */
            visited->pop_back();
        } /* if */
    }

    void Graph::breathTraversePrint(Vertex* node,
            vector<Vertex*>* visited) const {

        /* anchor for already visited */
        if ((find(visited->begin(), visited->end(), node) == visited->end())) {

            /* add to call stack */
            visited->push_back(node);

            /* print visited */
            cout << "Parent: " << *node << endl << flush;

            /* get first connected vertex index */
            int rowIdx = getIdxForVertex(node);
            vector<Vertex*> toVisitVector;
            for (size_t i = 0; i < nodes.size(); ++i) {
                /* visit next child */
                if (matrix[rowIdx][i] != 0) {
                    Vertex* curNode = nodes[i];
                    toVisitVector.push_back(curNode);
                    cout << "     Child: " << *curNode << endl << flush;
                } /* if */
            } /* for */

            /* visit the next level */
            for (auto it = toVisitVector.begin(); it != toVisitVector.end();
                    it++) {
                breathTraversePrint((*it), visited);
            } /* for */

            /* clear stack */
            toVisitVector.clear();

            /* remove from call stack */
            visited->pop_back();
        }
    }

    bool Graph::isManagedVertex(const Vertex* node) const {
        /* start is null */
        if (node == nullptr) {
            cout << "start vertex is null !!! Cannot print graph on null vertex"
                    << endl << flush;
            return false;
        }
        /* start not part of graph */
        else if (getIdxForVertex(node) == -1) {
            cout << "given start is not part of this graph !!!" << endl
                    << flush;
            return false;
        } /* if */

        return true;
    }

    ////////////////////////////////////////////////////////////
    // Public methods                                         //
    ////////////////////////////////////////////////////////////
    void Graph::addVertex(Vertex* node) {
        /* null node*/
        if (node == nullptr) {
            cout << "cannot add null vertex node !!!" << endl << flush;
            return;
        }
        /* node already  present in vector (pointer address) */
        if (find(nodes.begin(), nodes.end(), node) != nodes.end()) {
            cout << "vertex already present !!! Adding aborted" << endl
                    << flush;
            return;
        }
        /* container already full */
        else if (nodes.size() == size_t(maxSize)) {
            cout << "maxSize exceeded !!!" << endl << flush;
            return;
        }
        /* add node */
        else {
            nodes.push_back(node);
        } /* if */
    }

    void Graph::addEdge(const Vertex* startNode, const Vertex* endNode,
            double weight) {
        int rowIdx, colIdx;

        /* null vertex */
        if ((startNode == nullptr) || ((endNode == nullptr))) {
            cout << "start and end node are not allowed to be null !!!" << endl
                    << flush;
            return;
        }
        /* no vertex node present */
        else if (nodes.size() == 0) {
            cout << "no vertex nodes present !!!" << endl << flush;

            return;
        }
        /* invalid weight */
        else if (weight <= 0) {
            cout << "invalid weight !!! 0 < weight < DOUBLE_MAX";

            return;
        } /* if */

        /* get intended indices */
        rowIdx = getIdxForVertex(startNode);
        colIdx = getIdxForVertex(endNode);

        /* vertex instance not managed */
        if ((rowIdx < 0) || (colIdx < 0)) {
            cout << "Vertex instance not hold by backed container !!! start="
                    << startNode->name << " | end=" << endNode->name << endl
                    << flush;

            return;
        }
        /* add edge */
        else {
            matrix[rowIdx][colIdx] = weight;
        } /* if */
    }

    void Graph::clearEdges() {
        /* delete all edge weights */
        for (int i = 0; i < maxSize; i++) {
            for (int j = 0; j < maxSize; j++) {
                matrix[i][j] = 0;
            } /* for */
        } /* for */
    }

    void Graph::clear() {
        /* remove all edges */
        clearEdges();

        /* remove all vertex nodes */
        for (auto it = nodes.begin(); it != nodes.end(); it++) {
            delete (*it);
        } /* for */

        /* empty vector */
        nodes.clear();
    }

    void Graph::printDepthFirst(const Vertex* start) const {
        /* is vertex instance managed */
        if (isManagedVertex(start)) {
            /* prepare call stack */
            vector<Vertex*> visited;

            /* start recursion */
            deepTraversePrint(nodes[getIdxForVertex(start)], &visited);
        } /* if */

    }

    void Graph::printBreadthFirst(const Vertex* start) const {
        /* is vertex instance managed */
        if (isManagedVertex(start)) {

            /* prepare call stack */
            vector<Vertex*> visited;

            /* start recursion */
            breathTraversePrint(nodes[getIdxForVertex(start)], &visited);
        } /* if */

    }

    bool Graph::hasCycles() const {
        /* vertex node presents */
        if (nodes.size() == 0) {
            cout << "cannot determine cycles on empty graph" << endl << flush;
            return false;
        }

        size_t i = 0;
        bool cyclic = false;

        /* Walk through all nodes  */
        while ((i < nodes.size()) && (!cyclic)) {
            /* prepare call stack */
            vector<Vertex*> visited;

            cyclic = hasCycles(nodes[i], &visited);

            i++;
        } /* while */

        return cyclic;
    }

////////////////////////////////////////////////////////////
// friend methods                                         //
////////////////////////////////////////////////////////////
    ostream& operator<<(ostream& os, const Graph& graph) {
        int i = 0;
        /* introduce graph printing */
        os << "print graph:" << endl;
        /* iterate via present vertex nodes in the graph */
        for (auto it = graph.nodes.begin(); it != graph.nodes.end(); it++) {
            os << i << ". node: " << (*it)->name << endl;
            for (size_t j = 0; j < graph.nodes.size(); ++j) {
                if (graph.matrix[i][j] != 0) {
                    os << "     related node: " << (**it) << "("
                            << graph.matrix[i][j] << ")" << endl;
                } /* if */
            } /* for */
            i++;
        } /* for */

        return os;
    }
}
```

1.2.3 graph_test.cpp

Folgend ist der Source der Tests angeführt.

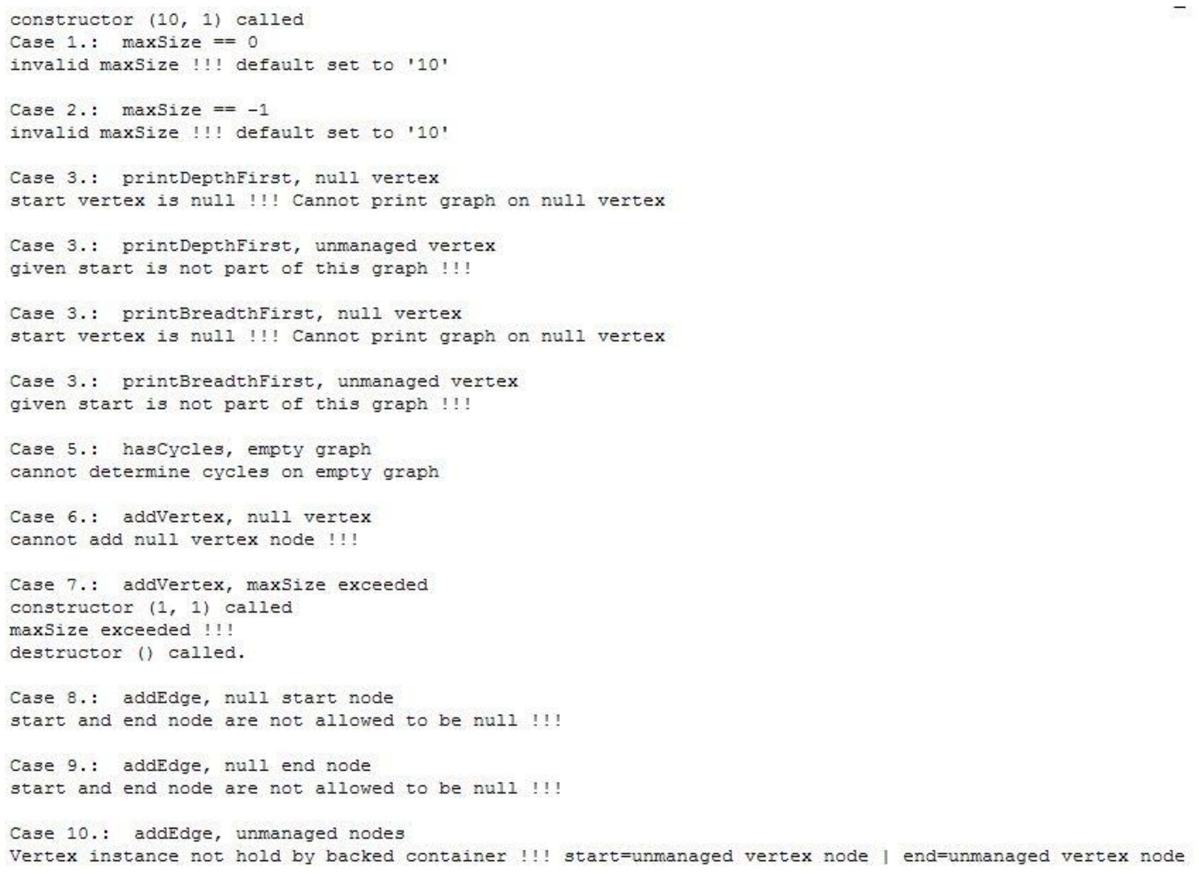
```
/*
 * graph_tests.cpp
 *
 *  Created on: Dec 9, 2014
 *      Author: cchet
 */
#include <string>
#include <iostream>
#include <cassert>
#include "graph.h"

using namespace std;
using namespace graph;

int main(int argc, char** argv) {
    Graph* graph = new Graph(10, true);
    Vertex* nullVertex = nullptr;
    Vertex* unmanagedVertex = new Vertex("unmanaged vertex node");

    ////////////////////////////////////////////////////
    // invalid argument tests                         //
    ////////////////////////////////////////////////////
    /*case 1: */
    cout << "Case 1.:  maxSize == 0" << endl << flush;
    Graph* case1_graph = new Graph(0);
    delete case1_graph;

    /* case 2: */
    cout << endl << "Case 2.:  maxSize == -1" << endl << flush;
    Graph* case2_graph = new Graph(-1);
    delete case2_graph;

    /* case 3: */
    cout << endl << "Case 3.:  printDepthFirst, null vertex" << endl << flush;
    graph->printDepthFirst(nullVertex);

    /* case 3: */
    cout << endl << "Case 3.:  printDepthFirst, unmanaged vertex" << endl
            << flush;
    graph->printDepthFirst(unmanagedVertex);

    /* case 4: */
    cout << endl << "Case 3.:  printBreadthFirst, null vertex" << endl << flush;
    graph->printBreadthFirst(nullVertex);

    /* case 4: */
    cout << endl << "Case 3.:  printBreadthFirst, unmanaged vertex" << endl
            << flush;
    graph->printBreadthFirst(unmanagedVertex);

    /* case 5: */
    cout << endl << "Case 5.:  hasCycles, empty graph" << endl << flush;
    graph->hasCycles();

    /* case 6: */
    cout << endl << "Case 6.:  addVertex, null vertex" << endl << flush;
    graph->addVertex(nullVertex);

    /* case 7: */
    cout << endl << "Case 7.:  addVertex, maxSize exceeded" << endl << flush;
    Graph* case7_graph = new Graph(1, true);
    Vertex* case7_1_vertex = new Vertex("case7_1_vertex");
    Vertex* case7_2_vertex = new Vertex("case7_2_vertex");
    case7_graph->addVertex(case7_1_vertex);
    case7_graph->addVertex(case7_2_vertex);
    // cleanup
    case7_graph->clear();
    case7_1_vertex = nullptr;
    case7_2_vertex = nullptr;
    delete case7_graph;

    /* case 8: */
    cout << endl << "Case 8.:  addEdge, null start node" << endl << flush;
    graph->addEdge(nullVertex, unmanagedVertex, 10);

    /* case 9: */
    cout << endl << "Case 9.:  addEdge, null end node" << endl << flush;
    graph->addEdge(unmanagedVertex, nullVertex, 10);

    /* case 10: */
    cout << endl << "Case 10.:  addEdge, unmanaged nodes" << endl << flush;
    Vertex* case10_1_vertex = new Vertex("case10_1_vertex");
    graph->addVertex(case10_1_vertex);
    graph->addEdge(unmanagedVertex, unmanagedVertex, 10);
    // cleanup
    graph->clear();
    case10_1_vertex = nullptr;

    ////////////////////////////////////////////////////
    // valid tests                                    //
    ////////////////////////////////////////////////////
    /* case 11: */
    cout << endl << "Case 11.:  operator<<" << endl << flush;
    Vertex* case11_1_vertex = new Vertex("case11_1_vertex");
    Vertex* case11_2_vertex = new Vertex("case11_2_vertex");
    Vertex* case11_3_vertex = new Vertex("case11_3_vertex");
    graph->addVertex(case11_1_vertex);
    graph->addVertex(case11_2_vertex);
    graph->addVertex(case11_3_vertex);
    cout << *graph << flush;
    graph->clear();

    /* case 12, 13, 14: */
    cout << endl << "Case 13 - 14:  prepared graph after vertex/aedge adding" << endl <<
flush;
    Vertex* case12_1_vertex = new Vertex("case12_1_vertex");
    Vertex* case12_2_vertex = new Vertex("case12_2_vertex");
    Vertex* case12_3_vertex = new Vertex("case12_3_vertex");
    graph->addVertex(case12_1_vertex);
    graph->addVertex(case12_2_vertex);
    graph->addVertex(case12_3_vertex);
    cout << endl << "After Vertex addition (operator<<)" << endl << *graph << flush;

    graph->addEdge(case12_1_vertex, case12_1_vertex, 1);
    graph->addEdge(case12_2_vertex, case12_3_vertex, 1);
    graph->addEdge(case12_1_vertex, case12_3_vertex, 1);
    graph->addEdge(case12_3_vertex, case12_1_vertex, 1);
    cout << endl << "After Edge addition (operator<<)" << endl << *graph << flush;

    /* case 12: */
    cout << endl << "Case 12.:  printDepthFirst(case12_1_vertex)" << endl << flush;
    graph->printDepthFirst(case12_1_vertex);

    /* case 13: */
    cout << endl << "Case 13.:  printDepthFirst(case12_2_vertex)" << endl << flush;
    graph->printDepthFirst(case12_2_vertex);

    /* case 13: */
    cout << endl << "Case 13.:  printBreadthFirst(case12_1_vertex)" << endl << flush;
    graph->printBreadthFirst(case12_1_vertex);

    /* case 14: */
    cout << endl << "Case 14.:  printBreadthFirst(case12_2_vertex)" << endl << flush;
    graph->printBreadthFirst(case12_2_vertex);

    /* case 15: */
    cout << endl << "Case 15.:  former graph hasCycles == true" << endl << flush;
    assert(graph->hasCycles());
    graph->clear();

    /* case 16: */
    cout << endl << "Case 16.:  hasCycles == false" << endl << flush;
    Vertex* case16_1_vertex = new Vertex("case16_1_vertex");
    Vertex* case16_2_vertex = new Vertex("case16_2_vertex");
    Vertex* case16_3_vertex = new Vertex("case16_3_vertex");
    graph->addVertex(case16_1_vertex);
    graph->addVertex(case16_2_vertex);
    graph->addVertex(case16_3_vertex);
    cout << endl << "Added vertex (operator<<)" << endl << *graph << flush;
    graph->addEdge(case16_1_vertex, case16_2_vertex, 1);
    graph->addEdge(case16_1_vertex, case16_3_vertex, 1);
    graph->addEdge(case16_2_vertex, case16_3_vertex, 1);
    cout << endl << "Added Edge (operator<<)" << endl << *graph << flush;
    assert(!graph->hasCycles());

    // clear
    graph->clear();
    delete graph;

    cout << "" << endl << flush;
}
```

1.3 Tests

Folgend sind die Tests für die Implementierung der Klasse Graph und Vertex angeführt.
Da bei der Vertex keine besonderen Fehler gemacht werden können konzentrieren sich diese Tests auf die Klasse Graph.

1.3.1 Invalid Arguments

Folgend sind die Tests angeführt, welche die Behandlung von Aufruffehlern testet.

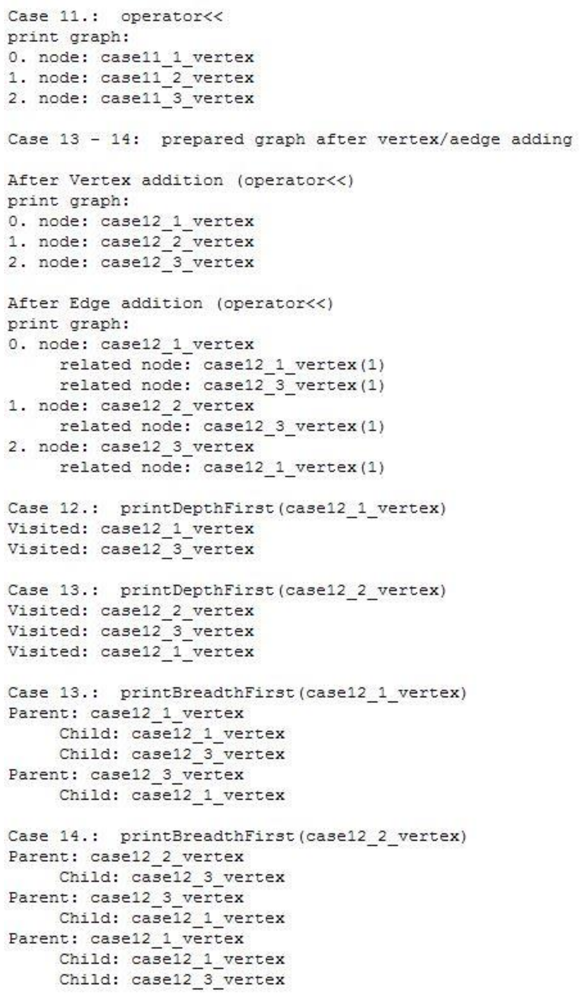


```
constructor (10, 1) called
Case 1.:  maxSize == 0
invalid maxSize !!! default set to '10'

Case 2.:  maxSize == -1
invalid maxSize !!! default set to '10'

Case 3.:  printDepthFirst, null vertex
start vertex is null !!! Cannot print graph on null vertex

Case 3.:  printDepthFirst, unmanaged vertex
given start is not part of this graph !!!

Case 3.:  printBreadthFirst, null vertex
start vertex is null !!! Cannot print graph on null vertex

Case 3.:  printBreadthFirst, unmanaged vertex
given start is not part of this graph !!!

Case 5.:  hasCycles, empty graph
cannot determine cycles on empty graph

Case 6.:  addVertex, null vertex
cannot add null vertex node !!!

Case 7.:  addVertex, maxSize exceeded
constructor (1, 1) called
maxSize exceeded !!!
destructor () called.

Case 8.:  addEdge, null start node
start and end node are not allowed to be null !!!

Case 9.:  addEdge, null end node
start and end node are not allowed to be null !!!

Case 10.:  addEdge, unmanaged nodes
Vertex instance not hold by backed container !!! start=unmanaged vertex node | end=unmanaged vertex node
```

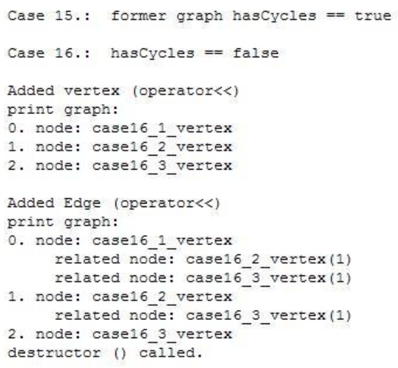
Es ist zu sehen das nullptr sowie nicht gemanagte Instanzen korrekt abgewiesen und eine dementsprechende Meldung ausgegeben wird.
Diese Tests wurden für alle nach außen sichtbaren Methoden angewandt, da die privaten dadurch implizit mit getestet werden.

1.4 Valid arguments

Folgend sind die Tests für die gültigen Argumente angeführt.



```
Case 11.:  operator<<
print graph:
0. node: case11_1_vertex
1. node: case11_2_vertex
2. node: case11_3_vertex

Case 13 - 14:  prepared graph after vertex/aedge adding

After Vertex addition (operator<<)
print graph:
0. node: case12_1_vertex
1. node: case12_2_vertex
2. node: case12_3_vertex

After Edge addition (operator<<)
print graph:
0. node: case12_1_vertex
     related node: case12_1_vertex(1)
     related node: case12_3_vertex(1)
1. node: case12_2_vertex
     related node: case12_3_vertex(1)
2. node: case12_3_vertex
     related node: case12_1_vertex(1)

Case 12.:  printDepthFirst(case12_1_vertex)
Visited: case12_1_vertex
Visited: case12_3_vertex

Case 13.:  printDepthFirst(case12_2_vertex)
Visited: case12_2_vertex
Visited: case12_3_vertex
Visited: case12_1_vertex

Case 13.:  printBreadthFirst(case12_1_vertex)
Parent: case12_1_vertex
     Child: case12_1_vertex
     Child: case12_3_vertex
Parent: case12_3_vertex
     Child: case12_1_vertex

Case 14.:  printBreadthFirst(case12_2_vertex)
Parent: case12_2_vertex
     Child: case12_3_vertex
Parent: case12_3_vertex
     Child: case12_1_vertex
Parent: case12_1_vertex
     Child: case12_1_vertex
     Child: case12_3_vertex
```



```
Case 15.:  former graph hasCycles == true

Case 16.:  hasCycles == false

Added vertex (operator<<)
print graph:
0. node: case16_1_vertex
1. node: case16_2_vertex
2. node: case16_3_vertex

Added Edge (operator<<)
print graph:
0. node: case16_1_vertex
     related node: case16_2_vertex(1)
     related node: case16_3_vertex(1)
1. node: case16_2_vertex
     related node: case16_3_vertex(1)
2. node: case16_3_vertex
destructor () called.
```

Etwaige Zyklen im Graphen werden korrekt behandelt und führen zu keinen Problemen. Die Test zeigen auch das egal von welchen Knoten man ausgeht der darunterliegende Graph immer korrekt angezeigt wird.