SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2018/19, Angabe 5

 Gruppe 1 (J. Heinzelreiter)

 Gruppe 2 (M. Hava)   Name: ___Niklas Vest_____________________   Aufwand [h]: ___20______

 Gruppe 3 (P. Kulczycki)   Übungsleiter/Tutor: ______________________   Punkte: ___________

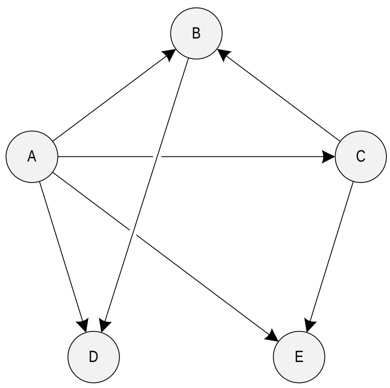
Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (25P + 40 P)	50%	100%	100%
2 (35 P)	30%	90%	40%

Beispiel 1: Abstrakter Datentyp „Gerichteter Graph" (`src/adt/`)

Implementieren Sie den abstrakten Datentyp „Gerichteter Graph" in zwei Varianten. Die eine Variante verwendet für den Graphen eine Nachbarschaftsmatrix (siehe dazu auch de.wikipedia.org/wiki/Adjazenz-matrix). Die andere Variante verwendet für den Graphen eine Nachbarschaftsliste (siehe dazu auch de.wikipedia.org/wiki/Adjazenzliste).

Beachten Sie die folgenden Anforderungen und Hinweise:

1. Beide Implementierungen haben identische Schnittstellen.
2. Beide Implementierungen verhalten sich – aus der Sicht eines Anwenders – absolut gleich.
3. Die „Payload" eines Knotens ist eine beliebig lange, dynamisch allokierte Zeichenkette.
4. Kanten haben keine „Payload" (also kein Gewicht etc.)
5. Die Anzahl der Knoten und Kanten, die der ADT aufnehmen kann, ist beliebig.
6. Knoten müssen dynamisch hinzugefügt und auch wieder gelöscht werden können. Wird ein Knoten gelöscht, so werden auch alle seine inzidenten Kanten gelöscht.
7. Kanten müssen hinzugefügt und auch wieder gelöscht werden können. Wird eine Kante hinzugefügt, so müssen seine inzidenten Knoten bereits Teil des ADTs sein.
8. Der ADT muss auf der Konsole entsprechend ausgegeben werden können.
9. Die Nachbarschaftsmatrix ist als dynamisch allokiertes (und damit eindimensionales) Feld aufzubauen.
10. Die Nachbarschaftsliste ist als eine Liste von Listen aufzubauen.
11. Führen Sie beide Implementierungen als separate C-Module (`.c`- und `.h`-Dateien) aus.
12. Implementieren Sie *ein* Testmodul (für beide Varianten) und testen Sie ausführlich.



Beispiel 2: Topologisches Sortieren (`src/top/`)

Implementieren Sie eine C-Funktion `topological_sort`, die die Knoten eines ADTs von Beispiel 1 topologisch sortiert auf der Konsole ausgibt. Diese Funktion können Sie als Teil des ADTs (in einer Variante Ihrer Wahl) ausführen.

Eine mögliche Ausgabe für den oben dargestellten Graphen wäre `A,C,B,D,E`. Siehe dazu auch en.wikipedia.org/wiki/Topological_sorting.

Ausarbeitung 05

Niklas Vest

November 12, 2018

1 Abstrakter Datentyp "Gerichteter Graph"

1.1 Lösungsidee

1.1.1 ADT

In einem Header *directed_graph.h* befinden sich die Schnittstellen für den abstrakten Datentyp "Gerichteter Graph". In einem separaten Header *graph_types.h* sind dann Typen und Typ-Aliases beschrieben. In *CmakeLists.txt* werden dann zwei Build-Targets angelegt. Eines von beiden wird mit einer extra Preprozessor Definition $UE05\_USE\_LIST$ gebaut, anhand deren dann zwischen Matrix- und Listen-Implementierung ausgewählt wird. Im code befinden sich Prüfungen auf die Existenz dieser Definition: Ist die Definition vorhanden, wird der Text aus *directed_graph*$_m$*at.c* entfernt und der Linked verbindet Aufrufe an die Schnittstellen des Graphentyps mit den Implementierungen in *directed_list.c*. Wenn $UE05\_USE\_LIST$ nicht gesetzt ist, wird die Matrix Implementierung verwendet.

1.1.2 Matrix

Bei der Matrix Implementierung besteht werden für die Informationsverwaltung bzgl. ein- und ausgehender Kanten eine Ajazenzmatrix verwendet. Diese ist als eindimensionales Array von *bool*s realisiert. Die Knoten des Graphen werden in einer einfach verketteten Liste gespeichert.

1.1.3 Liste

Die zweite Implementierung des ADT verwendet Listen, welche Aufschluss über ausgehende Kanten geben. Eingehende Kanten können dadurch nur mühselig berechnet werden, in dem man jede Adiazenzliste jedes Knotens betrachtet und nach einem bestimmten Zielknoten sucht.

Um eine zweite Implementierung einer einfach verketteten Liste zu vermeiden, verwende ich zur Speicherung von Knoten keine weitere Liste sondern eine minimale Vektor Implementierung. Ein Feld an Knoten - deren Index im Feld zugleich ihre ID ist - wird bei Überfüllung mit doppelter Kapazität neu angelegt.

Anmerkung: Man hätte mit *void∗* eine gewisse ”Generizität” erlangen können, man wäre dann aber gezwungen bei jedem Zugriff auf die Daten des knotens einen type cast durchzuführen und das ist nicht nur unsauber sondern auch unsicher.

1.2 Implementierung

Listing 1: *main.c*

```c
#include "directed_graph.h"
#include "unic.h"

int main()
{
    unic_init();

    // create graph
    graph_ptr pgraph = create_graph();
    unic_ass_true(pgraph != NULL, "Graph creation");
    unic_ass_eq_i((int) nr_of_nodes(pgraph), 0, "No nodes on new graph");
    unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges on new graph");

    //add nodes

    node_id a = add_graph_node(pgraph, "A");
    unic_ass_eq_i((int) nr_of_nodes(pgraph), 1, "New node nr after node insert");
    unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges after node insert");

    node_id b = add_graph_node(pgraph, "B");
    unic_ass_eq_i((int) nr_of_nodes(pgraph), 2, "New node nr after two node inserts");
    unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges after two node inserts");

    node_id c = add_graph_node(pgraph, "C");

    node_id d = add_graph_node(pgraph, "D");

    node_id e = add_graph_node(pgraph, "E");
    unic_ass_eq_i((int) nr_of_nodes(pgraph), 5, "New node nr after all node insert");
    unic_ass_eq_i((int) nr_of_edges(pgraph), 0, "No edges after all node inserts");

    // edges from A

    add_graph_edge(pgraph, a, b);
    unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after edge insert");
    unic_ass_eq_i(1, (int) nr_of_edges(pgraph), "New edge nr after edge insert");

    add_graph_edge(pgraph, a, c);
```

```
    unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after two edge inserts");
    unic_ass_eq_i(2, (int) nr_of_edges(pgraph), "New edge nr after two edge inserts");

    add_graph_edge(pgraph, a, d);

    add_graph_edge(pgraph, a, e);
    unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after all edge inserts");
    unic_ass_eq_i(4, (int) nr_of_edges(pgraph), "New edge nr after all edge inserts");

    unic_ass_eq_i(4, (int) node_out_degree(pgraph, a), "Correct outgoing node degree");

    // edges from B
    add_graph_edge(pgraph, b, d);
    unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "No new node nr after new edge insert from different node");
    unic_ass_eq_i(5, (int) nr_of_edges(pgraph), "New edge nr after new edge insert from different node");

    unic_ass_eq_i(2, (int) node_in_degree(pgraph, d), "Correct incoming node degree");

    // edges from C
    add_graph_edge(pgraph, c, b);
    add_graph_edge(pgraph, c, e);

    // print dis boi once
    print_graph(pgraph);

    // remove an existing edge
    remove_graph_edge(pgraph, b, d);
    unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "Unchanged number of node after edge removal");
    unic_ass_eq_i(6, (int) nr_of_edges(pgraph), "Decreased number of edges after edge removal");

    // remove none existend edge
    remove_graph_edge(pgraph, b, a);
    unic_ass_eq_i(5, (int) nr_of_nodes(pgraph), "Unchanged number of nodes after nonexistent edge removal");
    unic_ass_eq_i(6, (int) nr_of_edges(pgraph), "Unchanged number of edges after nonexistent edge removal");

    // remove existing node
    remove_graph_node(pgraph, c);
    unic_ass_eq_i(4, (int) nr_of_nodes(pgraph), "Decreased number of nodes after node removal");
    unic_ass_eq_i(3, (int) nr_of_edges(pgraph), "Decreased number of edges after node removal");

    unic_ass_eq_i(3, (int) node_out_degree(pgraph, a), "Correct outgoing node degree after manipulation");
    unic_ass_eq_i(1, (int) node_in_degree(pgraph, e), "Correct incoming node
```

```
      degree after manipulation");

    // remove nonexistent node
    remove_graph_node(pgraph, 8);
    unic_ass_eq_i(4, (int) nr_of_nodes(pgraph), "Unchanged number of nodes
      after nonexistent node removal");
    unic_ass_eq_i(3, (int) nr_of_edges(pgraph), "Unchanged number of edges
      after nonexistent node removal");

    // a second print
    // note: print_graph prints only nodes
    // connected by an edge
    print_graph(pgraph);

    delete_graph(&pgraph);
    unic_ass_true(pgraph == NULL, "Graph is null pointer after delete");
    return unic_get_results();
}
```

Listing 2: *directed_graph.h*

```
#ifndef UE05_DIRECTED_GRAPH_H
#define UE05_DIRECTED_GRAPH_H

#include <stdlib.h>
#include "graph_types.h"

/**
 * @return A pointer to a newly created graph.
 */
graph_ptr create_graph();

/**
 * Deletes the supplied graph.
 * @param ppgraph The graph to delete.
 */
void delete_graph(graph_ptr *ppgraph);

/**
 * @param pgraph The graph to add a node to.
 * @param payload The payload of the graph node.
 * @return The id of the newly created node.
 * @Note If the supplied graph is NULL, the function
 *         will still return a theoretically valid id
 *         but any usage results in undefined behaviour.
 */
node_id add_graph_node(graph_ptr pgraph, node_payload payload);

/**
 * Removes the node with the specified id and all edges
 * associated with it from the graph.
 * If the node does not know about the node with said id,
 * this function does nothing.
 * @param pgraph The graph to remove the node from.
 * @param id The id of the node to delete.
 */
void remove_graph_node(graph_ptr pgraph, node_id id);

```

```c
/**
 * Adds a directed edge to the graph, starting from the
 * supplied source node and ending at the destination node.
 * @param pgraph The graph to add the edge to.
 * @param from The source node for the edge.
 * @param to The destination node for the edge.
 */
void add_graph_edge(graph_ptr pgraph, node_id from, node_id to);

/**
 * Removes the edge going from <i>from</i> to <i>to</i>.
 * @param pgraph The graph to remove the edge from.
 * @param from The source node of the edge.
 * @param to The destination node of the edge.
 */
void remove_graph_edge(graph_ptr pgraph, node_id from, node_id to);

/**
 * @param cpgraph The graph to query.
 * @return The number of nodes in the graph.
 */
size_t nr_of_nodes(cgraph_ptr cpgraph);

/**
 * @param pgraph The graph to query.
 * @return The number of edges in the graph.
 */
size_t nr_of_edges(cgraph_ptr pgraph);

/**
 * Prints all relevant nodes. (i.e. nodes
 * that have outgoing or incoming directed
 * edges.)
 * @param cpgraph The graph to print.
 */
void print_graph(cgraph_ptr cpgraph);

/**
 * A simple iterator function which executes the
 * supplied consumer function for each edge in
 * the graph (with the respective edge as
 * argument).
 * @param cpgraph The graph to iterate.
 * @param consume_edge_f The consumer function to use.
 */
void for_each_edge(cgraph_ptr cpgraph, graph_edge_consumer consume_edge_f);

/**
 * @param cpgraph The graph to query.
 * @param id The id of the node to fetch information from.
 * @return The number of outgoing edges.
 */
size_t node_out_degree(cgraph_ptr cpgraph, node_id id);

/**
 * @param cpgraph The graph to query.
 * @param id The id of the node to fetch information from.
```

```c
 * @return The number of incoming edges.
 */
size_t node_in_degree(cgraph_ptr cpgraph, node_id id);

/**
 * @param cpgraph The graph from which to fetch 'em nodes.
 * @return An array containing all nodes of the graphs
 *         including their respective direct dependencies.
 */
conscious_node *get_conscious_nodes(cgraph_ptr cpgraph);

/**
 * @param cpgraph The graph to query.
 * @param id The id of the node to fetch.
 * @return The node within the supplied graph with the
 *          specified id. If the graph does not contain
 *         such a node, NULL is returned.
 */
graph_node_ptr get_graph_node_by_id(cgraph_ptr cpgraph, node_id id);

#endif //!UE05_DIRECTED_GRAPH_H
```

Listing 3: *graph_types.h*

```c
#ifndef UE05_GRAPH_TYPES_H
#define UE05_GRAPH_TYPES_H

#include <stdlib.h>

/**
 * The data associated with graph nodes.
 */
typedef const char* node_payload;

/**
 * The identifying descriptor of graph nodes.
 */
typedef unsigned node_id;

/**
 * A node within a {@link graph}.
 */
typedef struct {
    /**
     * The unique node descriptor.
     */
    node_id id;

    /**
     * The data associated with a node.
     */
    node_payload payload;
} graph_node;
typedef graph_node *graph_node_ptr;

/**
 * Represents a graph nodes dependencies.
 * (i. e. all nodes that have an outgoing
```

```c
 * edge landing at said node.)
 */
typedef node_id *dependencies;

/**
 * A node which is aware of its dependencies.
 */
typedef struct {
    /**
     * All information about the node itself .
     */
    graph_node data;

    /**
     * The dependencies of the node.
     */
    dependencies dependency_arr;

    /**
     * Have a guess what this describes.
     */
    size_t nr_of_dependencies;
} conscious_node;

/**
 * A directed graph.
 */
typedef struct graph graph;
typedef struct graph *graph_ptr;
typedef const struct graph *cgraph_ptr;

/**
 * Functions of this type can be used with the graphs
 * for_each_edge function to achieve some sort of
 * iterative behaviour without the need of a separate
 * iterator class for each graph implementation.
 *  Objects of this function type are supposed
 *  to be supplied with the source and destination node
 *  of a directed edge.
 */
typedef void(*graph_edge_consumer)(graph_node_ptr from, graph_node_ptr to);

#endif //!UE05_GRAPH_TYPES_H
```

Listing 4: $directed\_graph_{l}ist.c$

```c
// warning: ISO C forbids an empty translation unit
typedef int i_am_a_number_making_the_compiler_happy_because_the_pedantic\
_compile_option_warns_about_empty_translation_units;

#ifdef UE05_USE_LIST

#include <stdbool.h>
#include <memory.h>
#include "directed_graph.h"
#include "node_list.h"

#define INITIAL_VECTOR_SIZE 5
```

```c
/**
 * A dynamically growing array of list node pointers.
 */
typedef list_node_ptr *list_node_ptr_vec;

struct graph {
    /**
     * A vector containing all nodes of the graph.
     */
    list_node_ptr_vec nodes;

    /**
     * The current size of the vector.
     */
    size_t vec_size;
};

/**
 * @param payload The payload of the node.
 * @param id The id of the node.
 * @return A pointer to the newly created graph node.
 */
static graph_node_ptr create_graph_node(node_payload payload, node_id id);

/**
 * @param pgraph The graph for which to fetch the first
 *               free node it.
 * @return The first available node id for the
 *         supplied graph. If the graphs adjacency matrix
 *         is full, it is resized.
 */
static node_id generate_id(graph_ptr pgraph);

/**
 * Resizes the supplied graphs node vector
 * by doubling its capacity.
 * @param pgraph The graph of which the node vector
 *                   shall be resized.
 */
static void grow_node_vector(graph_ptr pgraph);

/**
 * @param pgraph The graph to query.
 * @param id The id of the node to search for.
 * @return True if the node exists, false
 *         otherwise.
 */
static bool node_exists(cgraph_ptr pgraph, node_id id);

/**
 * Deletes the node with the specified id.
 * @param pgraph The graph from which to delete the node.
 * @param id The id of the node to delete.
 */
static void delete_graph_node(graph_ptr pgraph, node_id id);
```

```c
graph_ptr create_graph()
{
    graph_ptr pgraph = (graph_ptr) malloc(sizeof(graph));
    pgraph->nodes = (list_node_ptr_vec) malloc(sizeof(list_node_ptr) *
     INITIAL_VECTOR_SIZE);
    for (size_t i = 0; i < INITIAL_VECTOR_SIZE; ++i) {
        pgraph->nodes[i] = NULL;
    }
    pgraph->vec_size = INITIAL_VECTOR_SIZE;
    return pgraph;
}

void delete_graph(graph_ptr *ppgraph)
{
    if (ppgraph != NULL && *ppgraph != NULL) {
        graph_ptr pgraph = *ppgraph;
        // if pgraph is not null, pgraph->nodes
        // is guaranteed to not be null because it's
        // a vector which is first allocated during
        // graph creation!
        for (node_id id = 0; id < pgraph->vec_size; ++id) {
            if (node_exists(pgraph, id)) {
                delete_graph_node(pgraph, id);
            }
        }
        free(pgraph->nodes);
        free(pgraph);
        *ppgraph = NULL;
    }
}

void delete_graph_node(graph_ptr pgraph, node_id id)
{
    if (node_exists(pgraph, id)) {
        free((char *) pgraph->nodes[id]->data->payload);
        free(pgraph->nodes[id]->data);
        delete_list(&(pgraph->nodes[id]));
    }
}

node_id add_graph_node(graph_ptr pgraph, node_payload payload)
{
    node_id id = 0;
    if (pgraph != NULL) {
        id = generate_id(pgraph);
        graph_node_ptr pnode = create_graph_node(payload, id);
        // if the next free id is not within the vectors bounds
        if (pgraph->vec_size <= id) {
            grow_node_vector(pgraph);
        }
        // add the graph node to the vector
            // the vector contains lists with their head
            // being the source graph node of an edge
            // and all ancestors representing the destination
            // nodes for edges going out from said source node.
        append_to_list_sorted(&(pgraph->nodes[id]), pnode);
    }
```

```c
    return id;
}

void remove_graph_node(graph_ptr pgraph, node_id id)
{
    if (node_exists(pgraph, id)) {
        // iterate all nodes and delete edges to the node with the specified id
        for (size_t current_id = 0; current_id < pgraph->vec_size; ++
     current_id) {
            if (node_exists(pgraph, (node_id) current_id)) {
                // since the node_id is the index into the vector
                remove_graph_edge(pgraph, (node_id) current_id, id);
            }
        }
        delete_graph_node(pgraph, id);
    }
}

void add_graph_edge(graph_ptr pgraph, node_id from, node_id to)
{
    // if both from and to exist
    if (node_exists(pgraph, from) && node_exists(pgraph, to)) {
        // add an entry with TO to the edge-list of FROM
        graph_node_ptr to_ptr = pgraph->nodes[to]->data;
        append_to_list_sorted(&(pgraph->nodes[from]), to_ptr);
    }
}

void remove_graph_edge(graph_ptr pgraph, node_id from, node_id to)
{
    if (node_exists(pgraph, from) && node_exists(pgraph, to)) {
        // traverse list of edges until the node with the desired ID is reached
        list_node_ptr prev_edge = pgraph->nodes[from];
        list_node_ptr curr_edge =prev_edge->next;
        while (curr_edge != NULL && curr_edge->data->id != to) {
            prev_edge = curr_edge;
            curr_edge = curr_edge->next;
        }
        if (curr_edge != NULL) {
            // and delete dat boi
            delete_list_node_after(prev_edge);
        }
    }
}

size_t nr_of_nodes(const cgraph_ptr cpgraph)
{
    size_t nr = 0;
    if (cpgraph != NULL) {
        for (size_t i = 0; i < cpgraph->vec_size; ++i) {
            if (node_exists(cpgraph, (node_id) i)) {
                nr += 1;
            }
        }
    }
    return nr;
}
```

```c

size_t nr_of_edges(const cgraph_ptr pgraph)
{
    size_t nr = 0;
    if (pgraph != NULL) {
        for (size_t i = 0; i < pgraph->vec_size; ++i) {
            if (node_exists(pgraph, (node_id) i)) {
                // for each node in the vector
                list_node_ptr curr_edge = pgraph->nodes[i]->next;
                while (curr_edge != NULL) {
                    // add +1 to the total sum for each
                    // list node after the head
                    nr += 1;
                    curr_edge = curr_edge->next;
                }
            }
        }
    }
    return nr;
}

void for_each_edge(const cgraph_ptr cpgraph, graph_edge_consumer
     consume_edge_f)
{
    if (cpgraph != NULL) {
        // for each source node
        for (size_t i = 0; i < cpgraph->vec_size; ++i) {
            list_node_ptr curr_src_node = cpgraph->nodes[i];
            if (curr_src_node != NULL) {
                graph_node_ptr src_graph_node = curr_src_node->data;
                list_node_ptr curr_dest_node = curr_src_node->next;
                // combine it with each of its destination nodes
                while (curr_dest_node != NULL) {
                    // and invoke the callback
                    consume_edge_f(src_graph_node, curr_dest_node->data);
                    curr_dest_node = curr_dest_node->next;
                }
            }
        }
    }
}

size_t node_out_degree(const cgraph_ptr cpgraph, node_id id)
{
    size_t degree = 0;
    if (node_exists(cpgraph, id)) {
        // first node is source node, ancestors are edge destinations
        list_node_ptr edge_list = cpgraph->nodes[id]->next;
        degree = get_list_size(edge_list);
    }
    return degree;
}

size_t node_in_degree(const cgraph_ptr cpgraph, node_id id)
{
    size_t degree = 0;
    // if the destination is valid
```

```c
    if (node_exists(cpgraph, id)) {
        // for each node in the vector
        for (size_t i = 0; i < cpgraph->vec_size; ++i) {
            if (node_exists(cpgraph, (node_id) i)) {
                // check if its edge list contains
                // the node with the specified id
                list_node_ptr curr_dest_node = cpgraph->nodes[i]->next;
                bool has_edge_to_id = false;
                while (curr_dest_node != NULL && !has_edge_to_id) {
                    has_edge_to_id = curr_dest_node->data->id == id;
                    curr_dest_node = curr_dest_node->next;
                }
                if (has_edge_to_id) {
                    degree += 1;
                }
            }
        }
    }
    return degree;
}

// -------------------------------------- TRANSLATION-UNIT-LOCAL FUNCTIONS -------------------------------------- //

graph_node_ptr create_graph_node(node_payload payload, node_id id)
{
    graph_node_ptr pnode = (graph_node_ptr) malloc(sizeof(graph_node));
    char *str_copy = (char *) malloc(sizeof(char) * strlen(payload) + 1);
    strcpy(str_copy, payload);
    pnode->payload = str_copy;
    pnode->id = id;
    return pnode;
}

node_id generate_id(graph_ptr pgraph)
{
    // if the graph is null, return 0
    node_id id = 0;
    if (pgraph != NULL) {
        // search the vector for the first entry
        // that is not NULL. if the vector is full,
        // return an id outside the bounds of the vector
        while (id < pgraph->vec_size && pgraph->nodes[id] != NULL) {
            id += 1;
        }
    }
    return id;
}

void grow_node_vector(graph_ptr pgraph)
{
    if (pgraph != NULL) {
        size_t new_length = pgraph->vec_size * 2;
        pgraph->nodes = (list_node_ptr_vec) realloc(pgraph->nodes, sizeof(list_node_ptr) * new_length);
        for (size_t i = 0; i < pgraph->vec_size; ++i) {
```

```
            // also initialize new items to NULL
            pgraph->nodes[pgraph->vec_size + i] = NULL;
        }
        pgraph->vec_size = new_length;
    }
}

bool node_exists(const cgraph_ptr pgraph, node_id id)
{
    return pgraph != NULL && id < pgraph->vec_size && pgraph->nodes[id] != NULL;
}

#endif
```

Listing 5: $directed\_graph_{m}at.c$

```
// warning: ISO C forbids an empty translation unit
typedef int i_am_a_number_making_the_compiler_happy_because_the_pedantic\
_compile_option_warns_about_empty_translation_units;

#ifndef UE05_USE_LIST

#include <stdlib.h>
#include <stdbool.h>
#include <memory.h>
#include "../adt/graph_types.h"
#include "../adt/directed_graph.h"
#include "../adt/node_list.h"

/**
 * The type of the elements in the
 * adjacency matrix.
 */
typedef bool adjacenty_mat_type;

/**
 * A matrix representing adjacency relations.
 */
typedef adjacenty_mat_type *adjacency_mat;

struct graph {
    /**
     * A matrix representing relationships
     * between all nodes of the graph.
     */
    adjacency_mat edges;

    /**
     * A single linked list of graph nodes.
     */
    list_node_ptr nodes;

    /**
     * The current dimension of the edge matrix.
     */
    size_t mat_side_len;
};
```

```

/**
 * @param pgraph The graph for which to fetch the first
 *               free node it.
 * @return The first available node id for the
 *         supplied graph. If the graphs adjacency matrix
 *         is full, it is resized.
 */
static node_id generate_id(graph_ptr pgraph);

/**
 * A function <i>f : R<sup>2</sup> --> R</i>
 * which maps the 2D indices <i>from</i> and
 * <i>to</i> to a 1D array index.
 * @param side_length The side length of the virtual matrix.
 * @param from The row to target.
 * @param to The column to target.
 * @return The projected 1D array index.
 */
static size_t calc_mat_index(size_t side_length, node_id from, node_id to);

/**
 * Sets the specified edge value for the edge from node
 * <i>from</i> to node <i>to</i>.
 * @param pgraph The graph to set the edge value for.
 * @param from The source node of the edge.
 * @param to The destination node of the edge.
 * @param edge_flag The value to set. <i>true</i> adds the
 *                  edge, <i>false</i> removes it.
 */
static void set_edge_value(graph_ptr pgraph, node_id from, node_id to, bool edge_flag);

/**
 * @param payload The payload of the node.
 * @param id The id of the node.
 * @return A pointer to the newly created graph node.
 */
static graph_node_ptr create_graph_node(node_payload payload, node_id id);

/**
 * Increases the supplied graphs edge matrix by one
 * row and one column.
 * @param pgraph The graph to manipulate.
 */
static void enlarge_matrix(graph_ptr pgraph);

/**
 * @param cpgraph The graph to query.
 * @param id The id to search for.
 * @return True if there is a node with the
 *         specified ID in the supplied graph.
 */
static bool node_exists(cgraph_ptr cpgraph, node_id id);

/**
 * @param cpgraph The graph to query
```

```
 * @param source The source node.
 * @param destination The destination node.
 * @return True if there is an edge going from the
 *         supplied source node to the destination node,
 *         False otherwise.
 */
bool node_has_edge_to(cgraph_ptr cpgraph, node_id source, node_id destination
    );

graph_ptr create_graph()
{
    graph_ptr pgraph = (graph_ptr) malloc(sizeof(graph));
    pgraph->mat_side_len = 0;
    pgraph->edges = NULL;
    pgraph->nodes = NULL;
    return pgraph;
}

void delete_graph(graph_ptr *ppgraph)
{
    if (ppgraph != NULL && *ppgraph != NULL) {
        graph_ptr pgraph = *ppgraph;

        // delete_list only deletes list nodes, not
        // associated data! We have to do that manually
        list_node_ptr curr = pgraph->nodes;
        while (curr != NULL) {
            free((char *) curr->data->payload);
            free(curr->data);
            curr = curr->next;
        }

        // free remaining memory associated with the graph
        free(pgraph->edges);
        delete_list(&(pgraph->nodes));
        free(pgraph);

        *ppgraph = NULL;
    }
}

node_id add_graph_node(graph_ptr pgraph, node_payload payload)
{
    node_id id = 0;
    if (pgraph != NULL) {
        id = generate_id(pgraph);
        graph_node_ptr pnode = create_graph_node(payload, id);
        append_to_list_sorted(&(pgraph->nodes), pnode);
    }
    return id;
}

void remove_graph_node(graph_ptr pgraph, node_id id)
{
    if (pgraph != NULL) {
        list_node_ptr node_to_delete = get_node_by_id(pgraph->nodes, id);
        if (node_to_delete != NULL) { // if it exists
```

```c
            // remove all edges to the node
            for (unsigned i = 0; i < nr_of_nodes(pgraph); ++i) {
                remove_graph_edge(pgraph, id, i);
                remove_graph_edge(pgraph, i, id);
            }

            list_node_ptr curr = pgraph->nodes;

            // if the node to delete is not the first one
            if (node_to_delete != curr) {
                // go on an epic adventure to find the
                // predecessor to dat boi
                while (curr->next != node_to_delete) {
                    curr = curr->next;
                }

                // unchain node_to_delete
                curr->next = node_to_delete->next;
            } else {
                // uncahin node_to_delete
                pgraph->nodes = node_to_delete->next;
            }

            // "encapsulate" the node so it
            // represents an individual list
            node_to_delete->next = NULL;
            // and delete it
            free((char *) node_to_delete->data->payload);
            free(node_to_delete->data);
            delete_list(&node_to_delete);
        }
    }
}

void add_graph_edge(graph_ptr pgraph, node_id from, node_id to)
{
    set_edge_value(pgraph, from, to, true);
}

void remove_graph_edge(graph_ptr pgraph, node_id from, node_id to)
{
    set_edge_value(pgraph, from, to, false);
}

size_t nr_of_nodes(const cgraph_ptr pgraph)
{
    size_t nr = 0;
    if (pgraph != NULL) {
        nr = get_list_size(pgraph->nodes);
    }
    return nr;
}

size_t nr_of_edges(const cgraph_ptr pgraph)
{
    size_t nr = 0;
    if (pgraph != NULL) {
```

```c
        size_t mat_arr_length = pgraph->mat_side_len * pgraph->mat_side_len;
        for (size_t i = 0; i < mat_arr_length; ++i) {
            if (pgraph->edges[i]) {
                nr += 1;
            }
        }
    }
    return nr;
}

void for_each_edge(const cgraph_ptr cpgraph, graph_edge_consumer consume_edge_f)
{
    if (cpgraph != NULL && consume_edge_f != NULL) {
        // for each row (source node)
        for (node_id from = 0; from < cpgraph->mat_side_len; ++from) {
            list_node_ptr from_ptr = get_node_by_id(cpgraph->nodes, from);
            // for each column (destination node) of the current row
            for (node_id to = 0; to < cpgraph->mat_side_len; ++to) {
                // if they are connected via an edge
                if (cpgraph->edges[calc_mat_index(cpgraph->mat_side_len, from, to)]) {
                    list_node_ptr to_ptr = get_node_by_id(cpgraph->nodes, to);
                    // consume it
                    consume_edge_f(from_ptr->data, to_ptr->data);
                }
            }
        }
    }
}

size_t node_out_degree(const cgraph_ptr cpgraph, node_id source)
{
    size_t degree = 0;
    if (node_exists(cpgraph, source)) {
        for (node_id i = 0; i < cpgraph->mat_side_len; ++i) {
            // +1 for each edge in the <source>th row
            // because the row _i_ represents outgoing
            // edges to the node with id _i_
            if (node_has_edge_to(cpgraph, source, i)) {
                degree += 1;
            }
        }
    }
    return degree;
}

size_t node_in_degree(const cgraph_ptr cpgraph, node_id destination)
{
    size_t degree = 0;
    // if destination is not out of bounds for the matrix
    if (node_exists(cpgraph, destination)) {
        // +1 for each edge in the <destination>th row
        // because the row _i_ represents outgoing
        // edges to the node with id _i_
        for (node_id i = 0; i < cpgraph->mat_side_len; ++i) {
```

```
            if (node_has_edge_to(cpgraph, i, destination)) {
                degree += 1;
            }
        }
    }
    return degree;
}

conscious_node *get_conscious_nodes(const cgraph_ptr cpgraph)
{
    conscious_node *nodes_arr = NULL;
    if (cpgraph != NULL) {
        size_t node_count = nr_of_nodes(cpgraph);
        nodes_arr = (conscious_node *) malloc(sizeof(conscious_node) * node_count);
        for (node_id i = 0; i < node_count; ++i) {
            if (node_exists(cpgraph, i)) {
                conscious_node cn;
                cn.data = *(get_graph_node_by_id(cpgraph, i));
                cn.dependency_arr = (dependencies) malloc(sizeof(node_id) * node_in_degree(cpgraph, i));
                cn.nr_of_dependencies = 0;
                for (node_id dep = 0; dep < cpgraph->mat_side_len; ++dep) {
                    if (node_has_edge_to(cpgraph, dep, i)) {
                        cn.dependency_arr[cn.nr_of_dependencies++] = dep;
                    }
                }
                nodes_arr[i] = cn;
            }
        }
    }
    return nodes_arr;
}

graph_node_ptr get_graph_node_by_id(cgraph_ptr cpgraph, node_id id)
{
    graph_node_ptr pgraph_node = NULL;
    if (cpgraph != NULL) {
        list_node_ptr curr = cpgraph->nodes;
        while (curr != NULL && curr->data->id != id) {
            curr = curr->next;
        }
        if (curr != NULL) {
            pgraph_node = curr->data;
        }
    }
    return pgraph_node;
}

// ---------------------------------------- TRANSLATION-UNIT-LOCAL FUNCTIONS ---------------------------------------- //

graph_node_ptr create_graph_node(node_payload payload, node_id id)
{
    graph_node_ptr pnode = (graph_node_ptr) malloc(sizeof(graph_node));
    char *str_cpy = (char *) malloc(sizeof(char) * strlen(payload) + 1);
```

```c
    strcpy(str_cpy, payload);
    pnode->payload = str_cpy;
    pnode->id = id;
    return pnode;
}

node_id generate_id(graph_ptr pgraph)
{
    node_id id = 0;
    // if the graph is empty or there are no nodes
    // return 0!
    if (pgraph != NULL) {

        if (pgraph->edges == NULL) {
            enlarge_matrix(pgraph);
            // we do not need to do anything else here;
            // 0 is the next available index!
        } else {
            // the node list is sorted so if the first
            // node id is not 0 we can immediately return 0.
            // otherwise we have to search for any "ID gaps"
            if (pgraph->nodes->data->id == 0) {

                list_node_ptr curr = pgraph->nodes->next;
                while (curr != NULL && curr->data->id == id + 1) {
                    id += 1; // same as curr = curr->data->id;
                    curr = curr->next;
                }

                // in both cases (curr == NULL and curr != NULL)
                // we have to increment since _id_ still holds
                // the last unavailable id before the one that's
                // available
                id +=1 ;

                if (curr == NULL) {
                    // id would now index our matrix out
                    // of bounds so lets make it BIGGUR
                    enlarge_matrix(pgraph);
                }
            }
        }
    }
    return id;
}

size_t calc_mat_index(size_t side_length, node_id from, node_id to)
{
    return from * side_length + to;
}

void set_edge_value(graph_ptr pgraph, node_id from, node_id to, bool
     edge_flag)
{
    if (pgraph != NULL) {
        if (node_exists(pgraph, from) && node_exists(pgraph, to)) {
            pgraph->edges[calc_mat_index(pgraph->mat_side_len, from, to)] =
```

```
    edge_flag;
        }
    }
}

void enlarge_matrix(graph_ptr pgraph)
{
    // prevent enlargement if the matrix is not completely
    // STUFFED with IDs
    if (pgraph != NULL && pgraph->mat_side_len == nr_of_nodes(pgraph)) {
        // allocate new matrix
        size_t new_mat_side_len = pgraph->mat_side_len + 1;
        size_t arr_length = new_mat_side_len * new_mat_side_len;
        adjacency_mat new_mat = (adjacency_mat) malloc(sizeof(
    adjacenty_mat_type) * arr_length);

        if (pgraph->mat_side_len == 0) {
            new_mat[0] = false;
        } else {
            // copy values
            for (node_id from = 0; from < pgraph->mat_side_len; ++from) {

                // write old values to the new matrix
                for (node_id to = 0; to < pgraph->mat_side_len; ++to) {
                    size_t old_index = calc_mat_index(pgraph->mat_side_len,
    from, to);
                    size_t new_index = calc_mat_index(new_mat_side_len, from,
     to);
                    new_mat[new_index] = pgraph->edges[old_index];
                }

                // set the edge flags for new nodes to 0

                // graph->mat_side_len is now the last column / row since the
                // new side length is now graph->mat_side_len + 1
                size_t last_row_index = calc_mat_index(new_mat_side_len, (
    node_id) pgraph->mat_side_len, from);
                size_t last_col_index = calc_mat_index(new_mat_side_len, from
    , (node_id) pgraph->mat_side_len);
                size_t corner_index = calc_mat_index(new_mat_side_len, (
    node_id) pgraph->mat_side_len,
                                                     (node_id) pgraph->
    mat_side_len);
                new_mat[last_row_index] = false;
                new_mat[last_col_index] = false;
                new_mat[corner_index] = false;
            }
        }

        // delete old matrix and deploy new one
        free(pgraph->edges);
        pgraph->edges = new_mat;
        pgraph->mat_side_len += 1;
    }
}

bool node_exists(const cgraph_ptr pgraph, node_id id)
```

```
{
    list_node_ptr pgraph_node = NULL;
    if (pgraph != NULL) {
        pgraph_node = get_node_by_id(pgraph->nodes, id);
    }
    return pgraph_node != NULL;
}

bool node_has_edge_to(const cgraph_ptr cpgraph, node_id source, node_id
     destination)
{
    size_t index = calc_mat_index(cpgraph->mat_side_len, source, destination)
     ;
    return cpgraph->edges[index];
}

#endif
```

Listing 6: *directed_graph.c*

```
// common implementations for both
// the adjacency matrix and list
// version of the directed graph

#include "directed_graph.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

/**
 * Prints a pair of nodes connected by an edge.
 * @param from The source node.
 * @param to The destination node.
 */
static void print_pair(graph_node_ptr from, graph_node_ptr to);

void print_graph(const cgraph_ptr graph)
{
    if (graph != NULL) {
        for_each_edge(graph, print_pair);
    }
}

// ------------------------------------ TRANSLATION
     -UNIT-LOCAL FUNCTIONS
     ------------------------------------ //

void print_pair(graph_node_ptr from, graph_node_ptr to)
{
    if (from != NULL && to != NULL) {
        printf("( %s ) --> ( %s )\n", from->payload, to->payload);
    }
}
```

Listing 7: *node_list.h*

```
#ifndef UE05_NODE_LIST_H
#define UE05_NODE_LIST_H
```

```c

#include "graph_types.h"
#include <stdlib.h>

/**
 * A node of a single linked list.
 */
typedef struct list_node {
    /**
     * The information stored in the list node.
     */
    graph_node_ptr data;

    /**
     * The next node in the list.
     */
    struct list_node *next;
} list_node;
typedef list_node *list_node_ptr;

/**
 * Recursively deletes a list starting from a given node.
 * Note: this does not delete the data within the list!
 * @param plist The list to delete.
 */
void delete_list(list_node_ptr *plist);

/**
 * Appends the supplied node to the back of the list.
 * @param plist The list to append to.
 * @param data The graph data to add to the appended list node.
 */
void append_to_list_sorted(list_node_ptr *plist, graph_node_ptr data);

/**
 * @param list The list to query.
 * @param id The id of the node to fetch.
 * @return The list node with the specified id.
 */
list_node_ptr get_node_by_id(list_node_ptr list, node_id id);

/**
 * Removes the direct ancestor of <i>plist_node</i>.
 * @param plist_node The predecessor of the node to delete.
 */
void delete_list_node_after(list_node_ptr plist_node);

/**
 * @param list The list to use for evaluation.
 * @return The number of elements in the list.
 */
size_t get_list_size(list_node_ptr list);

#endif //!UE05_NODE_LIST_H
```

Listing 8: *node_list.c*

```c
#include "node_list.h"
```

```c
#include <stdlib.h>
#include <assert.h>

/**
 * @param data The data to stuff into the list node.
 * @return A pointer to the newly created list node.
 */
static list_node_ptr create_list_node(graph_node_ptr data);

void delete_list(list_node_ptr *plist)
{
    if (plist != NULL) {
        list_node_ptr node = *plist;
        if (node != NULL) {
            list_node_ptr next = node->next;
            free(node);
            // point plist to the next node in the list;
            *plist = next;
            // delete recursively
            delete_list(plist);
        }
        *plist = NULL;
    }
}

void append_to_list_sorted(list_node_ptr *plist, graph_node_ptr data)
{
    if (plist != NULL && data != NULL) {
        list_node_ptr node = create_list_node(data);
        if (*plist == NULL) {
            *plist = node;
        } else {
            list_node_ptr current = *plist;
            while (current->next != NULL && current->next->data->id < data->id) {
                // this should nevur evur heppen
                assert(data->id != current->next->data->id);
                current = current->next;
            }

            // if there is an "ID gap", insert inbetween
            if (current->next != NULL && current->next->data->id > data->id) {
                node->next = current->next;
            }
            current->next = node;
        }
    }
}

list_node_ptr get_node_by_id(list_node_ptr list, node_id id)
{
    if (list == NULL || list->data->id > id) {
        return NULL;
    } else if (list->data->id == id) {
        return list;
    } else {
```

```c
        return get_node_by_id(list->next, id);
    }
}

void delete_list_node_after(list_node_ptr plist_node)
{
    if (plist_node != NULL && plist_node->next != NULL) {
        list_node_ptr buffer = plist_node->next->next;
        free(plist_node->next);
        plist_node->next = buffer;
    }
}

size_t get_list_size(list_node_ptr list)
{
    size_t size = 0;
    if (list != NULL) {
        size = 1 + get_list_size(list->next);
    }
    return size;
}

// ------------------------------------- TRANSLATION-UNIT-LOCAL FUNCTIONS -------------------------------------- //

list_node_ptr create_list_node(graph_node_ptr data)
{
    list_node_ptr node = (list_node_ptr) malloc(sizeof(list_node));
    node->next = NULL;
    node->data = data;
    return node;
}
```

1.3 Tests

Das Tool für Speicheranalyse ”Valgrind” fand keine invaliden Speicheroperationen oder -verluste.

```
/home/niklas/Documents/Github/fh-hgb-ws1819/swo/ue05/cmake-build-debug/adtm
Test "Graph creation" succeeded.
Test "No nodes on new graph" succeeded.
Test "No edges on new graph" succeeded.
Test "New node nr after node insert" succeeded.
Test "No edges after node insert" succeeded.
Test "New node nr after two node inserts" succeeded.
Test "No edges after two node inserts" succeeded.
Test "New node nr after all node insert" succeeded.
Test "No edges after all node inserts" succeeded.
Test "No new node nr after edge insert" succeeded.
Test "New edge nr after edge insert" succeeded.
Test "No new node nr after two edge inserts" succeeded.
Test "New edge nr after two edge inserts" succeeded.
Test "No new node nr after all edge inserts" succeeded.
Test "New edge nr after all edge inserts" succeeded.
Test "Correct outgoing node degree" succeeded.
Test "No new node nr after new edge insert from different node" succeeded.
Test "New edge nr after new edge insert from different node" succeeded.
Test "Correct incoming node degree" succeeded.
( A ) ⟶ ( B )
( A ) ⟶ ( C )
( A ) ⟶ ( D )
( A ) ⟶ ( E )
( B ) ⟶ ( D )
( C ) ⟶ ( B )
( C ) ⟶ ( E )
Test "Unchanged number of node after edge removal" succeeded.
Test "Decreased number of edges after edge removal" succeeded.
Test "Unchanged number of nodes after nonexistent edge removal" succeeded.
Test "Unchanged number of edges after nonexistent edge removal" succeeded.
Test "Decreased number of nodes after node removal" succeeded.
Test "Decreased number of edges after node removal" succeeded.
Test "Correct outgoing node degree after manipulation" succeeded.
Test "Correct incoming node degree after manipulation" succeeded.
Test "Unchanged number of nodes after nonexistent node removal" succeeded.
Test "Unchanged number of edges after nonexistent node removal" succeeded.
( A ) ⟶ ( B )
( A ) ⟶ ( D )
( A ) ⟶ ( E )
Test "Graph is null pointer after delete" succeeded.
================
UNIC ran 30 tests.
30 tests succeeded.
================

Process finished with exit code 0
```

Figure 1: Resultat von *adt/main.c*

2 Topologisches Sortieren

2.1 Lösungsidee

Zum topologischen Sortieren habe ich den Algorithmus von Kahn herangezogen: Alle Knoten eines graphen, die keine Abhängigkeiten haben (i. e. keine eingehenden Kanten), werden in eine Liste S gepackt. Alle Knoten, welche auf diese Knoten verweisen, werden ”gelöst” von ebendiesen. Danach werden die Knoten aus S in ein weiteres Feld L eingetragen. Dieses Feld wird am schluss die geordnete Menge an Knoten enthalten. Nun werden die zuvor gelösten Knoten in S eingeschrieben. Der ganze Prozess wird dann so lange wiederholt, bis im Graphen keine Kanten mehr verzeichnet sind. Gibt es doch noch welche, ist der Graph zyklisch und kann nicht topologisch sortiert werden. (Quelle: Wikipedia)

2.2 Implementierung

Listing 9: *main.c*

```c
#include <stdio.h>
#include <stdbool.h>
#include "directed_graph.h"
#include "graph_types.h"

#define MAX 100

// Sorting a graph using kahns algorithm
void sort_topologically(graph_ptr pgraph, conscious_node nodes[], bool *error)
{
    /*
     * In order do differentiate between my comments and the
     * outline of kahns algorithm, I put the outline comments
     * at the beginning of a line.
     */

    size_t node_count = nr_of_nodes(pgraph);
    if (error != NULL) {
        *error = false;
    }

//      L  Empty list that will contain the sorted elements
    conscious_node *L = (conscious_node*) malloc(sizeof(conscious_node) * node_count);
    size_t li = 0;

//      S  Set of all nodes with no incoming edge
    conscious_node *S = (conscious_node*) malloc(sizeof(conscious_node) * node_count);
    size_t si = 0;

    conscious_node *All = get_conscious_nodes(pgraph);

    // move all nodes without dependencies to S
```

```c
    for (size_t i = 0; i < node_count; ++i) {
        if (All[i].nr_of_dependencies == 0) {
            S[si++] = All[i];
        }
    }

//    while S is non-empty do
    while (si > 0) {
//          remove a node n from S
        conscious_node n = S[--si];
//          add n to tail of L
        L[li++] = n;
//          for each node m with an edge e from n to m do
        for (size_t potential_m_index = 0; potential_m_index < node_count; ++potential_m_index) {
            conscious_node potential_m = All[potential_m_index];
            size_t m_dep_index = 0;
            while (m_dep_index < potential_m.nr_of_dependencies && potential_m.dependency_arr[m_dep_index] != n.data.id) {
                ++m_dep_index;
            }
            if (m_dep_index < potential_m.nr_of_dependencies) {
                conscious_node m = potential_m;
//                  remove edge e from the graph
                remove_graph_edge(pgraph, n.data.id, m.data.id);
//                  if m has no other incoming edges then
                if (node_in_degree(pgraph, m.data.id) == 0) {
//                  insert m into S
                    S[si++] = m;
                }
            }
        }
    }

    if (nr_of_edges(pgraph) != 0 && error != NULL) {
        *error = true;
    }
    for (size_t cpy = 0; cpy < node_count; ++cpy) {
        nodes[cpy] = L[cpy];
        free(All[cpy].dependency_arr);
    }

    free(All);
    free(L);
    free(S);
}

void print_sort_result(const cgraph_ptr cpgraph, const conscious_node *nodes, bool error)
{
    if (error) {
        printf("Graph was cyclic");
    } else {
        for (size_t i = 0; i < nr_of_nodes(cpgraph); ++i) {
            printf("( %s ), ", nodes[i].data.payload);
        }
        printf("\n");
```

```c
    }
}

int main()
{
    conscious_node nodes[MAX];
    bool error = false;
    graph_ptr pgraph = create_graph();

    // construct graph
    node_id work = add_graph_node(pgraph, "working out a solution");
    node_id finish = add_graph_node(pgraph, "finishing");
    node_id read = add_graph_node(pgraph, "reading the requirements");
    add_graph_edge(pgraph, read, work);
    add_graph_edge(pgraph, work, finish);

    sort_topologically(pgraph, nodes, &error);
    print_sort_result(pgraph, nodes, error);

    // construct cyclic graph
    graph_ptr cyclic_graph = create_graph();
    node_id a = add_graph_node(cyclic_graph, "A");
    add_graph_edge(cyclic_graph, a, a);

    sort_topologically(cyclic_graph, nodes, &error);
    print_sort_result(cyclic_graph, nodes, error);

    delete_graph(&pgraph);
    delete_graph(&cyclic_graph);
    return 0;
}
```

2.3 Tests

```
/home/niklas/Documents/Github/fh-hgb-ws1819/swo/ue05/cmake-build-debug/top
( reading the requirements ), ( working out a solution ), ( finishing ),
Graph was cyclic
Process finished with exit code 0
```

Figure 2: Resultat von *top/main.c*