Inhaltsverzeichnis

1. Abstrakter Datentyp Gerichteter Graph

1.1. Lösungsidee

1.1.1. Liste

Um einen gerichteten Graphen in einer Listenstruktur darstellen zu können, benötigt es eine Liste, in der alle Nodes mit ihrer Payload gespeichert werden. Zusätzlich wird für jede „node“ eine Liste benötigt, in der alle „edges“ zu einer anderen „node" gespeichert werden. In den „edge-nodes“ wird allerdings nicht die Payload der „target-node“ gespeichert, sondern nur eine Referenz auf die „target-node“. Mit dieser Methode erspart man sich bei diversen Operationen am Graphen einige „strcmp“ - Vergleiche.

Beim Einfügen einer „edge“ wird geprüft, ob diese bereits existiert. Wenn nicht, dann wird sie an die „edge-list“ der entsprechenden „node“ angehängt.

Beim Entfernen einer „node“ werden zuvor alle „edges“ entfernt, die auf diese „node“ verweisen. Im Anschluss kann die „edge-list“ der „node“ sowie die „node“ selbst entfernt bzw. freigegeben werden.

Beim Einfügen und Entfernen von „nodes“ und „edges“ werden Fehler (wie z.b.: Fehlen der angegeben „target-node“) durch eine Fehlermeldung abgefangen.

→ Topologische Sortierung ist Teil der Graph-List-Implementierung (Näheres siehe unter Punkt 2)

1.1.2. Matrix

Bei dieser Implementierung wird eine Struktur mit (BOOL**, CHAR**, INT) angelegt. Mit dieser Struktur ist es nun möglich einen beliebig großen (dynamisch allokierten) Graphen anzulegen.

Grundsätzlich wird mit dem BOOL** eine dyn. Feld mit dem Datentyp BOOL* angelegt. Jeder dieser BOOL* Zeigt wiederum auf ein dyn. allokiertes Feld. Dieses Vorgehen ist ähnlich zu jenem des dynamisch allokierten String Arrays.

Immer wenn eine neue „node“ bzw. eine neue „edge“ hinzugefügt wird, wird neuer Speicher allokiert (malloc) bzw. reallokiert (realloc). Mit dieser Implementierung wird wirklich nur jener Speicher reserviert, der auch verwendet wird.

Beim Einfügen einer neuen „node“ muss jedes dyn. BOOL Array um einen Eintrag erweitert werden, der auf False gesetzt wird. Beim letzten Eintrag muss ein komplettes Bool Array Initialisiert werden, da dieses gerade durch den neuen „node“ entstanden ist.

Beim Löschen eines Nodes werden die Einträge im dynamische allokierten Bool Feld sowie die BOOL Pointer jeweils geschiftet, so dass das zu löschende Element überschrieben wird. Im Anschluss wird das Feld reallocated (das letzte Element im dyn. Feld wird freigegeben).

1.2. Code

Graph_list/Types.h

```c
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_TYPES_H
#define GRAPHS_TYPES_H

/* ------------------------------------------------------------*/

typedef struct graph_list_node {
    char *payload;
    struct edge_node *edges;
    struct graph_list_node *next;
} graph_list_node;

typedef graph_list_node *graph_node_ptr;
typedef graph_node_ptr graph_list;

typedef struct edge_node {
    struct edge_node *next;
    struct graph_list_node *target;
} edge_node;

typedef edge_node *edge_node_ptr;

/* ------------------------------------------------------------*/

#endif //GRAPHS_TYPES_H
```

Graph_list/Graph_list.h

```c
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_GRAPH_LIST_H
#define GRAPHS_GRAPH_LIST_H


#if !defined GRAPH_LIST_H
#define      GRAPH_LIST_H

/* ------------------------------------------------------------*/

#include <stdbool.h>
#include "./types.h"

/* ------------------------------------------------------------*/

void init_graph_l (graph_list *list);
void add_graph_node_l (graph_list *list, char *str);
void print_graph_nodes_l (graph_node_ptr list);
void remove_node_l (graph_node_ptr *list, char *str);
void add_edge_l (graph_list list, char  *origin_str, char  *target_str);
void remove_all_edges_of_l (graph_list list, char *str);
void remove_edge_l (graph_list  list,char *origin_str, char *target_str);
void topological_sort_l(graph_list list);
void free_list(graph_list *list);

/* ------------------------------------------------------------*/

graph_node_ptr new_node(char *str);
void prepend (graph_list *list, graph_node_ptr node);
```

```
bool node_exists(graph_list list, char *str);
bool edge_exists(graph_node_ptr origin, graph_node_ptr target);

/* ------------------------------------------------------------*/

#endif   // GRAPH_LIST_H


#endif //GRAPHS_GRAPH_LIST_H
```

Graph_list/Graph_list.c

```
//
// Created by Michael Neuhold on 09.11.19.
//


#include "./graph_list.h"
#include "./edge_list.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <limits.h>

/* ------------------------------------------------------------*/

graph_node_ptr new_node (char *str) {
    char *payload = malloc(sizeof(char) * strlen(str) + 1);
    strcpy(payload,str);
    graph_node_ptr n = malloc(sizeof(graph_list_node));
    n -> payload = payload;
    n -> edges = NULL;
    n -> next = NULL;
    return n;
}

bool node_exists(graph_list list, char *str) {
    graph_list l = list;
    while(l != NULL && strcmp(l -> payload,str) != 0) {
        l = l -> next;
    }
    return (l != NULL) ? true : false;
}

/* ------------------------------------------------------------*/

void init_graph_l (graph_list *list) {
    *list = NULL;
}

/* ------------------------------------------------------------*/

void prepend (graph_list *list, graph_node_ptr node) {
    node -> next = *list;
    *list = node;
}

/* ------------------------------------------------------------*/

void add_graph_node_l (graph_list *list, char *str) {
    // check if node already exists
    if(node_exists(*list,str)) {
        printf("node exists already: %s\n", str);
        return;
    }
```

```
    // prepend node:
    prepend(list, new_node(str));
}

/* ------------------------------------------------------------*/

void print_graph_nodes_l (graph_node_ptr list) {
    graph_node_ptr l = list;
    printf("\n--------------------\n");
    while(l != NULL) {
        printf("%s", l -> payload);
        printf(" | --> ");
        print_edge_list(l -> edges);
        l = l -> next;
        printf("\n--------------------\n");
    }
}

/* ------------------------------------------------------------*/

void remove_edges_to_node(graph_node_ptr list, graph_node_ptr node) {
    edge_node_ptr edges_list;
    graph_node_ptr l = list;
    while(l != NULL) {
        edges_list = l -> edges;
        while(edges_list != NULL) {
            if(edges_list -> target == node) {
                remove_edge_l(list,l -> payload, edges_list -> target ->
payload);
            }
            edges_list = edges_list -> next;
        }
        l = l -> next;
    }
}

void remove_node_l (graph_node_ptr *list, char *str) {
    graph_node_ptr prev = NULL;
    graph_node_ptr l = *list;
    graph_node_ptr n = NULL;
    while(l != NULL && strcmp(l -> payload,str) != 0) {
        prev = l;
        l = l -> next;
    }
    if(l != NULL && prev != NULL) {
        remove_edges_to_node(*list, l);     // remove all edges on this node
        remove_edge_list(&(l -> edges));    // remove edge list
        prev -> next = l -> next;
        l -> next = NULL;
        free(n);
    } else if(l != NULL) {
        remove_edges_to_node(*list, l);     // remove all edges on this node
        remove_edge_list(&(l -> edges));    // remove edge list
        *list = (*list) -> next;
        free(l);
    }
}

/* ------------------------------------------------------------*/

graph_node_ptr getNode(graph_list list, char *str) {
    graph_list l = list;
    while(l != NULL && strcmp(l -> payload,str) != 0) {
        l = l -> next;
    }
    return (l != NULL) ? l : NULL;
}
```

```c
/* ------------------------------------------------------------*/

bool edge_exists(graph_node_ptr origin, graph_node_ptr target) {
    edge_node_ptr enp = origin -> edges;
    graph_node_ptr n;
    while(enp != NULL) {
        n = enp -> target;
        if(strcmp(target -> payload, n -> payload) == 0) {
            printf("edge already exists!\n");
            return true;
        }
        enp = enp -> next;
    }
    return false;
}

/* ------------------------------------------------------------*/

void add_edge_l (graph_list list, char  *origin_str, char  *target_str) {
    graph_node_ptr origin = getNode(list, origin_str);
    graph_node_ptr target = getNode(list, target_str);

    if( origin == NULL || target == NULL) {
        printf("origin or target node does not exist!");
        return;
    }

    // to prevent edge duplication
    if(!edge_exists(origin,target)){
        prepend_edge_node(&(origin -> edges), new_edge_node(target));
    }
}

/* ------------------------------------------------------------*/

void remove_all_edges_of_l (graph_list list, char *str) {
    graph_node_ptr node = getNode(list,str);
    if(node == NULL) {
        printf("node does not exist!\n");
        return;
    }
    remove_edge_list(&(node -> edges));
}

/* ------------------------------------------------------------*/

void remove_edge_l (graph_list  list, char *origin_str, char *target_str) {
    graph_node_ptr origin = getNode(list, origin_str);
    graph_node_ptr target = getNode(list, target_str);

    if(origin == NULL || target == NULL) {
        printf("origin or target node not exist");
        return;
    }

    edge_node_ptr el = origin -> edges; // el = edge list of origin node
    edge_node_ptr prev = NULL; // prev pointer - necessary to delete edge
    while(el != NULL && el -> target != target) {
        prev = el;
        el = el -> next;
    }

    if(el != NULL && prev != NULL) {
        prev -> next = el -> next;
        el -> next = NULL;
        free(el);
    } else if(el != NULL) {
        origin -> edges = origin -> edges -> next;
```

```
    } else {
        printf("entered edge does not exist!\n");
    }
}

/* ------------------------------------------------------------*/
```

Graph_list/Edge_list.h

```
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_EDGE_LIST_H
#define GRAPHS_EDGE_LIST_H

/* ------------------------------------------------------------*/

#include "./types.h"

/* ------------------------------------------------------------*/

edge_node_ptr new_edge_node(graph_node_ptr target);
void prepend_edge_node(edge_node_ptr *edge_list, edge_node_ptr n);
void print_edge_list(edge_node_ptr edge_list);
void remove_edge_list(edge_node_ptr *edge_list);

/* ------------------------------------------------------------*/

#endif //GRAPHS_EDGE_LIST_H
```

Graph_list/Edge_list.c

```
//
// Created by Michael Neuhold on 09.11.19.
//


/* ------------------------------------------------------------*/

#include <stdlib.h>
#include <stdio.h>
#include "./edge_list.h"

/* ------------------------------------------------------------*/

edge_node_ptr new_edge_node(graph_node_ptr target) {
    edge_node_ptr n = malloc(sizeof(edge_node));
    n -> target = target;
    n -> next = NULL;
    return n;
}

/* ------------------------------------------------------------*/

void prepend_edge_node(edge_node_ptr *edge_list, edge_node_ptr n) {
    n -> next = *edge_list;
    *edge_list = n;
}

/* ------------------------------------------------------------*/

void print_edge_list(edge_node_ptr edge_list) {
    edge_node_ptr l = edge_list;
    while(l != NULL) {
        graph_node_ptr n = l -> target;
```

```
        printf("# %s ", n -> payload );
        l = l -> next;
    }
}

/* ------------------------------------------------------------*/

void remove_edge_list(edge_node_ptr *edge_list) {
    edge_node_ptr n;
    while(*edge_list != NULL) {
        n = *edge_list;
        (*edge_list) = (*edge_list) -> next;
        free(n);
    }
}
```

Graph_matrix/types.h

```
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_TYPES_H
#define GRAPHS_TYPES_H

/* ------------------------------------------------------------*/

#include <stdbool.h>

typedef struct matrix {
    char **nodes;
    bool **edges;
    int elm;
} matrix;

typedef matrix graph_matrix;

/* ------------------------------------------------------------*/

#endif //GRAPHS_TYPES_H
```

Graph_matrix/graph_matrix.h

```
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_GRAPH_MATRIX_H
#define GRAPHS_GRAPH_MATRIX_H

#include "./types.h"

void init_graph_m (graph_matrix *matrix);
void add_graph_node_m (graph_matrix *matrix, char *str);
void print_graph_nodes_m(graph_matrix matrix);
void add_edge_m (graph_matrix matrix, char  *origin_str, char  *target_str);
void remove_edge_m (graph_matrix  matrix, char *origin_str, char *target_str);
void remove_node_m (graph_matrix *matrix, char *str);

#endif //GRAPHS_GRAPH_MATRIX_H
```

Graph_matrix/graph_matrix.c

```c
//
// Created by Michael Neuhold on 09.11.19.
//

#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "./graph_matrix.h"

/* ------------------------------------------------------------*/

void init_graph_m (graph_matrix *matrix) {
    matrix -> elm = 0;
    matrix -> nodes = NULL;
    matrix -> edges = NULL;
}

/* ------------------------------------------------------------*/

bool node_exists_m(graph_matrix matrix, char *str) {
    int i = 0;
    while(i < matrix.elm) {
        if(strcmp((matrix.nodes)[i], str) == 0) {
            return true;
        }
        i++;
    }
    return false;
}

/* ------------------------------------------------------------*/

void add_graph_node_m (graph_matrix *matrix, char *str) {

    if(node_exists_m(*matrix, str)) {
        printf("node is already part of graph");
        return;
    }

    // allocate payload
    char *payload = (char*)malloc(sizeof(char) * (strlen(str) + 1));
    strcpy(payload,str);

    if(matrix -> elm == 0) {
        // allocate first element
        matrix -> nodes = (char**)malloc(sizeof(char*));
        matrix -> nodes[0] = payload;

        matrix -> edges = (bool**)malloc(sizeof(bool*));
        matrix -> edges[0] = (bool*)malloc(sizeof(bool));
        matrix -> edges[0][0] = false;

    } else {
        // allocate n element
        matrix -> nodes = (char**)realloc(matrix -> nodes, sizeof(char*) *
(matrix -> elm + 1));
        matrix -> nodes[matrix -> elm] = payload;

        matrix -> edges = (bool**)realloc(matrix -> edges, sizeof(bool*) *
(matrix -> elm + 1));
        for(int i = 0; i < matrix -> elm; i++) {
            matrix -> edges[i] = (bool*)realloc(matrix -> edges[i],
sizeof(bool) * (matrix -> elm + 1));
            matrix -> edges[i][matrix -> elm] = false;
        }
```

```
        matrix -> edges[matrix -> elm] = (bool*)malloc(sizeof(bool) * (matrix
-> elm + 1));
        for(int i = 0; i <= matrix -> elm; i++) {
            matrix -> edges[matrix -> elm][i] = false;
        }
    }
    matrix -> elm++;
}

/* ----------------------------------------------------------------*/

void print_graph_nodes_m (graph_matrix matrix) {
    printf("---> print\n");
    printf("\t\t");
    for(int i = 0; i < matrix.elm; i++) { printf("%s\t", matrix.nodes[i]); }
    printf("\n");
    for(int i = 0; i < matrix.elm; i++) {
        printf("%s\t", matrix.nodes[i]);
        for(int j = 0; j < matrix.elm; j++) {
            printf("%d\t\t", (matrix.edges[i])[j]);
        }
        printf("\n");
    }
}

/* ----------------------------------------------------------------*/

int get_node_index (graph_matrix matrix, char *str) {
    int i = 0;
    while(i < matrix.elm) {
        if(strcmp(matrix.nodes[i], str) == 0) {
            return i;
        }
        i++;
    }
    return -1;
}

/* ----------------------------------------------------------------*/

void add_edge_m (graph_matrix matrix, char  *origin_str, char  *target_str) {
    int origin_index = get_node_index(matrix, origin_str);
    int target_index = get_node_index(matrix, target_str);

    if(target_index == -1 || origin_index == -1) {
        printf("origin or target node does not exist!\n");
        return;
    }

    (matrix.edges[origin_index])[target_index] = true;
}

/* ----------------------------------------------------------------*/

void remove_edge_m (graph_matrix  matrix,char *origin_str, char *target_str) {
    int origin_index = get_node_index(matrix, origin_str);
    int target_index = get_node_index(matrix, target_str);

    if(target_index == -1 || origin_index == -1) {
        printf("origin or target node does not exist!\n");
        return;
    }

    (matrix.edges[origin_index])[target_index] = false;
}

/* ----------------------------------------------------------------*/
```

```c
void remove_node_m (graph_matrix *matrix, char *str) {
    int index = get_node_index(*matrix, str);

    free((matrix -> edges)[index]);
    (matrix -> edges)[index] = NULL;

    free((matrix -> nodes)[index]);
    (matrix -> edges)[index] = NULL;

    if(matrix -> elm == 1) {
        matrix -> edges = NULL;
        matrix -> nodes = NULL;
    } else {
        for(int i = index + 1; i < matrix -> elm; i++) {
            (matrix -> edges)[i-1] = (matrix -> edges)[i];
            (matrix -> nodes)[i-1] = (matrix -> nodes)[i];
        }
        matrix -> edges = (bool**)realloc(matrix -> edges, sizeof(bool*) *
(matrix -> elm-1));
        matrix -> nodes = (char**)realloc(matrix -> nodes, sizeof(char*) *
(matrix -> elm-1));

        for(int i = 0; i < matrix -> elm-1; i++) {
            for(int j = index + 1; j < matrix -> elm; j++) {
                (matrix -> edges[i])[j-1] = (matrix -> edges[i])[j];
            }
            (matrix -> edges)[i] = (bool*)realloc((matrix -> edges)[i],
sizeof(bool) * matrix -> elm-1);
        }
    }

    matrix -> elm--;
}

/* ------------------------------------------------------------*/
```

IO_lib.h

```c
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_IO_LIB_H
#define GRAPHS_IO_LIB_H

#include <stdio.h>

void print_line();

#endif //GRAPHS_IO_LIB_H
```

IO_lib.c

```c
//
// Created by Michael Neuhold on 09.11.19.
//

#include "io_lib.h"

void print_line() {
    for (int i = 0; i < 50; i++) {
        printf("-");
    }
    printf("\n");
}
```

Test.h

```
//
// Created by Michael Neuhold on 09.11.19.
//

#ifndef GRAPHS_TEST_H
#define GRAPHS_TEST_H

#define GRAPH_LIST

#ifdef GRAPH_LIST

/* ------------------------------------------------------------*/

#include "./graph_list/graph_list.h"
typedef graph_list graph;
#define graph_init init_graph_l
#define graph_add_node add_graph_node_l
#define graph_print print_graph_nodes_l
#define graph_add_edge add_edge_l
#define graph_remove_all_edges_from_node remove_all_edges_of_l
#define graph_remove_edge remove_edge_l
#define graph_remove_node remove_node_l
#define graph_topological_sort topological_sort_l
#define graph_free free_list
/* ------------------------------------------------------------*/

#else

/* ------------------------------------------------------------*/
#include "./graph_matrix/graph_matrix.h"
typedef graph_matrix graph;
#define graph_init init_graph_m
#define graph_add_node add_graph_node_m
#define graph_print print_graph_nodes_m
#define graph_add_edge add_edge_m
#define graph_remove_all_edges_from_node remove_all_edges_of_m
#define graph_remove_edge remove_edge_m
#define graph_remove_node remove_node_m
/* ------------------------------------------------------------*/

#endif

#endif //GRAPHS_TEST_H
```

Test.c

```
#include <stdio.h>
#include <stdlib.h>

#include "./test.h"
#include "./io_lib.h"

// What do you want to test?
#define DELETE_EDGE
#define DELETE_NODE
#define TOPSORT

int main () {

    graph g1;
    graph g2;

    // init graph
    graph_init(&g1);
```

```
    graph_init(&g2);

    // add some nodes
    graph_add_node(&g1, "Node 1");
    graph_add_node(&g1, "Node 2");
    graph_add_node(&g1, "Node 3");
    graph_add_node(&g1, "Node 4");
    graph_add_node(&g1, "Node 5");
    graph_add_node(&g1, "Node 6");

    graph_add_node(&g2, "Unterhose");
    graph_add_node(&g2, "Pullover");
    graph_add_node(&g2, "Mantel");
    graph_add_node(&g2, "Hose");
    graph_add_node(&g2, "Schuhe");
    graph_add_node(&g2, "Socken");
    graph_add_node(&g2, "Unterhemd");

    // add some edges
    graph_add_edge(g1, "Node 1", "Node 2");
    graph_add_edge(g1, "Node 1", "Node 3");
    graph_add_edge(g1, "Node 1", "Node 4");
    graph_add_edge(g1, "Node 4", "Node 1");
    graph_add_edge(g1, "Node 2", "Node 1");
    graph_add_edge(g1, "Node 3", "Node 4");

    graph_add_edge(g2, "Unterhose", "Hose");
    graph_add_edge(g2, "Pullover", "Mantel");
    graph_add_edge(g2, "Hose", "Mantel");
    graph_add_edge(g2, "Hose", "Schuhe");
    graph_add_edge(g2, "Socken", "Schuhe");
    graph_add_edge(g2, "Unterhemd", "Pullover");

    /* ------------------------------------------------------------*/

    print_line();
    printf("graph g1: \n");
    graph_print(g1);

    print_line();
    printf("graph g2: \n");
    graph_print(g2);

    /* ------------------------------------------------------------*/

    #ifdef DELETE_EDGE
    print_line();
    printf("delete edge (Node 1 -> Node 3) from g1\n");
    graph_remove_edge(g1,"Node 1", "Node 3");

    printf("delete edge (Hose -> Mantel) from g2\n");
    graph_remove_edge(g2,"Hose", "Mantel");

    print_line();
    printf("graph g1: \n");
    graph_print(g1);

    print_line();
    printf("graph g2: \n");
    graph_print(g2);
    #endif

    /* ------------------------------------------------------------*/

    #ifdef DELETE_NODE
    print_line();
    printf("delete node (Node 1) from g1\n");
    graph_remove_node(&g1, "Node 1");
```

```
    printf("delete node (Schuhe) from g2\n");
    graph_remove_node(&g2, "Schuhe");

    print_line();
    printf("graph g1: \n");
    graph_print(g1);

    print_line();
    printf("graph g2: \n");
    graph_print(g2);

    #endif

    /* ------------------------------------------------------------*/

    #ifdef TOPSORT
    print_line();
    printf("graph g1 top-sorted: \n");
    graph_topological_sort(g1);

    print_line();
    printf("graph g2 top-sorted: \n");
    graph_topological_sort(g2);

    #endif
    /* ------------------------------------------------------------*/

    return EXIT_SUCCESS;
}
```

1.3. Testfälle

Im Test.h File kann die Implementierung (Liste/Matrix) mit einem Define geändert werden.

1.3.1. Test Cases Liste

```
/Users/michaelneuhold/Documents/FH/Semester/03_Semester/02_SWO_UE/Uebungen/UE05/graphs/cmake-build-debug/graphs
-----------------------------------------------------
graph g1:

---------------------
Node 6 | -->
---------------------
Node 5 | -->
---------------------
Node 4 | --> # Node 1
---------------------
Node 3 | --> # Node 4
---------------------
Node 2 | --> # Node 1
---------------------
Node 1 | --> # Node 4 # Node 3 # Node 2
---------------------
-----------------------------------------------------
graph g2:

---------------------
Unterhemd | --> # Pullover
---------------------
Socken | --> # Schuhe
---------------------
Schuhe | -->
---------------------
Hose | --> # Schuhe # Mantel
---------------------
Mantel | -->
---------------------
Pullover | --> # Mantel
---------------------
Unterhose | --> # Hose
---------------------
-----------------------------------------------------
delete edge (Node 1 -> Node 3) from g1
delete edge (Hose -> Mantel) from g2
-----------------------------------------------------
graph g1:

---------------------
Node 6 | -->
---------------------
Node 5 | -->
---------------------
Node 4 | --> # Node 1
---------------------
Node 3 | --> # Node 4
---------------------
Node 2 | --> # Node 1
---------------------
Node 1 | --> # Node 4 # Node 2
---------------------
```

```
-----------------------------------------------------
graph g2:

---------------------
Unterhemd | --> # Pullover
---------------------
Socken | --> # Schuhe
---------------------
Schuhe | -->
---------------------
Hose | --> # Schuhe
---------------------
Mantel | -->
---------------------
Pullover | --> # Mantel
---------------------
Unterhose | --> # Hose
---------------------
-----------------------------------------------------
delete node (Node 1) from g1
delete node (Schuhe) from g2
-----------------------------------------------------
graph g1:

---------------------
Node 6 | -->
---------------------
Node 5 | -->
---------------------
Node 4 | -->
---------------------
Node 3 | --> # Node 4
---------------------
Node 2 | -->
---------------------
-----------------------------------------------------
graph g2:

---------------------
Unterhemd | --> # Pullover
---------------------
Socken | -->
---------------------
Hose | -->
---------------------
Mantel | -->
---------------------
Pullover | --> # Mantel
---------------------
Unterhose | --> # Hose
---------------------
-----------------------------------------------------
graph g1 top-sorted:
| Node 6 | | Node 5 | | Node 3 | | Node 4 | | Node 2 |
-----------------------------------------------------
graph g2 top-sorted:
| Unterhemd | | Socken | | Pullover | | Mantel | | Unterhose | | Hose |

Process finished with exit code 0
```

1.3.2. Test Cases Matrix

```
/Users/michaelneuhold/Documents/FH/Semester/03_Semester/02_SWO_UE/Uebungen/UE05/graphs/cmake-build-debug/graphs
-----------------------------------------------------
graph g1:
---> print
                Node 1  Node 2  Node 3  Node 4  Node 5  Node 6
Node 1  0               1               1               1               0               0

Node 2  1               0               0               0               0               0

Node 3  0               0               0               1               0               0

Node 4  1               0               0               0               0               0

Node 5  0               0               0               0               0               0

Node 6  0               0               0               0               0               0

-----------------------------------------------------
graph g2:
---> print
                Unterhose       Pullover  Mantel  Hose    Schuhe  Socken  Unterhemd
Unterhose       0               0               0               1               0
        0               0
Pullover  0               0               1               0               0               0
        0
Mantel  0               0               0               0               0               0
        0
Hose    0               0               1               0               1               0
        0
Schuhe  0               0               0               0               0               0
        0
Socken  0               0               0               0               1               0
        0
Unterhemd       0               1               0               0               0
        0               0
-----------------------------------------------------
delete edge (Node 1 -> Node 3) from g1
delete edge (Hose -> Mantel) from g2
-----------------------------------------------------
graph g1:
---> print
                Node 1  Node 2  Node 3  Node 4  Node 5  Node 6
Node 1  0               1               0               1               0               0

Node 2  1               0               0               0               0               0

Node 3  0               0               0               1               0               0

Node 4  1               0               0               0               0               0

Node 5  0               0               0               0               0               0

Node 6  0               0               0               0               0               0

-----------------------------------------------------
graph g2:
---> print
                Unterhose       Pullover  Mantel  Hose    Schuhe  Socken  Unterhemd
```

```
Unterhose       0               0               0               1               0
        0               0
Pullover 0              0               1               0               0               0
                0
Mantel  0               0               0               0               0               0
                0
Hose    0               0               0               0               1               0
                0
Schuhe  0               0               0               0               0               0
                0
Socken  0               0               0               0               1               0
                0
Unterhemd       0               1               0               0               0
        0               0
-----------------------------------------------------
delete node (Node 1) from g1
delete node (Schuhe) from g2
-----------------------------------------------------
graph g1:
---> print
                Node 2  Node 3  Node 4  Node 5  Node 6
Node 2  0               0               0               0               0
Node 3  0               0               1               0               0
Node 4  0               0               0               0               0
Node 5  0               0               0               0               0
Node 6  0               0               0               0               0
-----------------------------------------------------
graph g2:
---> print
                Unterhose       Pullover Mantel Hose    Socken  Unterhemd
Unterhose       0               0               0               1               0
        0
Pullover 0              0               1               0               0               0

Mantel  0               0               0               0               0               0

Hose    0               0               0               0               0               0

Socken  0               0               0               0               0               0

Unterhemd       0               1               0               0               0
        0

Process finished with exit code 0
```

2. Topologisches sortieren

2.1. Lösungsidee

Um das Topologische Sortieren umzusetzen, wird die Listen – Implementierung des Graphen verwendet. Grundsätzlich wird immer die „node“ gesucht, die am wenigsten oft in einer Edge referenziert wird. Wurde diese „node“ gefunden, so kann sie ausgegeben werden und aus dem Graphen samt aller „edges“ die von ihr ausgehen, entfernt werden. Dieser Vorgang wird solange wiederholt, bis sich keine „node“ mehr im Graphen befindet (sprich graph_list == NULL). Zu diesem Zeitpunkt wurden alle Nodes in sortierter Reihenfolge am Terminal ausgegeben.

Der Sinn hinter dem topologischen Sortieren ist es, dass die Nodes so angeordnet werden, dass alle Pfeile nach rechts zeigen und es keine rückläufigen Pfeile nach links gibt.

2.2. Code

```
/* ------------------------------------------------------*/

// top sort

int get_as_target_count(graph_list list, char *str) {
    int count = 0;
    edge_node_ptr edges = NULL;
    while(list != NULL) {
        edges = list -> edges;
        while(edges != NULL) {
            if(strcmp(edges -> target -> payload,str) == 0) {
                count++;
            }
            edges = edges -> next;
        }
        list = list -> next;
    }
    return count;
}

bool is_target(graph_list list, graph_node_ptr node) {
    while(list != NULL) {
        if(edge_exists(list, node)) return true;
        list = list -> next;
    }
    return false;
}

void topological_sort_l(graph_list list) {
    graph_list original_list = list;
    graph_node_ptr curr_node = list;
    graph_list sorted_nodes = NULL;

    while(original_list != NULL) {
        int min_cnt = INT_MAX;
        graph_node_ptr min_node = NULL;
        curr_node = original_list;
        while (curr_node != NULL) {
```

```
            // get count node as target
            if(get_as_target_count(original_list,curr_node -> payload)
< min_cnt) {
                min_cnt = get_as_target_count(original_list,curr_node
-> payload);
                min_node = curr_node;
            }
            curr_node = curr_node -> next;
        }
        if(min_node != NULL) {
            printf("| %s | ", min_node->payload);
            // remove min_node
            remove_node_l(&original_list, min_node->payload);
        }
    }
    printf("\n");
}
```

2.3. Testfälle

Siehe Test Cases Liste 1.3.1. ;)