Inhaltsverzeichnis

[1. Abstrakter Datentyp Gerichteter Graph 2](#_Toc24358568)

[1.1. Lösungsidee 2](#_Toc24358569)

[1.1.1. Liste 2](#_Toc24358570)

[1.1.2. Matrix 2](#_Toc24358571)

[1.2. Code 3](#_Toc24358572)

[1.3. Testfälle 15](#_Toc24358573)

[1.3.1. Test Cases Liste 15](#_Toc24358574)

[1.3.2. Test Cases Matrix 17](#_Toc24358575)

[2. Topologisches sortieren 19](#_Toc24358576)

[2.1. Lösungsidee 19](#_Toc24358577)

[2.2. Code 19](#_Toc24358578)

[2.3. Testfälle 20](#_Toc24358579)

# Abstrakter Datentyp Gerichteter Graph

## Lösungsidee

### 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)

### 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).

## Code

|  |
| --- |
| Graph\_list/Types.h |
| *// // 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 |
| *// // 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**; }  */\* ---------------------------------------------------------\*/* **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 |
| *// // 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**; }  */\* ---------------------------------------------------------\*/* **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 |
| *// // 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 |
| *// // 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**; } |

## Testfälle

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

### 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 |

### 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 |

# Topologisches sortieren

## 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.

## 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"**); } |

## Testfälle

*Siehe Test Cases Liste 1.3.1. ;)*