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

☒ Gruppe 1 (J. Heinzelreiter)

☐ Gruppe 2 (M. Hava) Name: Papesh Konstantin Aufwand [h]: 15

☐ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: _____ Punkte: _____

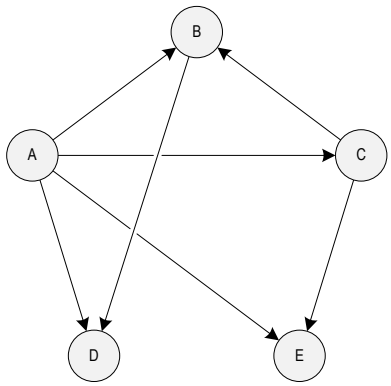
Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (25P + 40 P)	100	90	90
2 (35 P)	100	100	100

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/Adjazenzmatrix). 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.

SWO31 & SWB31 Softwareentwicklung mit klassischen Sprachen und Bibliotheken – WS 2018/19
Übungsabgabe 5

Konstantin Papesh

11. November 2018

5.1 Abstrakter Datentyp „Gerichteter Graph“

5.1.1 Lösungsidee

Grundsätzlich sollen beide Implementierungen die gleichen Schnittstellen bieten, um das Testen zu vereinfachen. Dafür wird mithilfe von cmake *test.c* jeweils für jeden Testfall konfiguriert. Mögliche Konfigurationen sind:

1. matrix
2. list
3. top

Wobei *top* sich auf 5.2 bezieht und automatisch als Liste kompiliert.
Beim Programm handelt es sich um die Verwaltung eines gerichteten Graphen, wobei uns die Implementierung die Möglichkeit bieten soll, dynamisch Knoten hinzuzufügen und zu entfernen. Weiters sollen Verbindungen zwischen Knoten hergestellt werden können und gleichzeitig entfernt werden können. Beide Implementierungen sollen so verfasst werden, sodass der Nutzer keinen Unterschied bei der Benützung merkt. Da das Programm dynamisch erweiterbar sein soll, muss mithilfe der *malloc*-Funktion gearbeitet werden. Für beide Implementierungen sind folgende Funktionen verfügbar:

1. init
2. destroy
3. createNode
4. createEdge
5. deleteNode
6. deleteEdge
7. print

Dabei werden *init* und *destroy* jeweils für die Erstellung bzw. Zerstörung des Structs verwendet. Die *print*-Funktion wird zur ungeordneten Ausgabe der Implementierung verwendet. Dabei ist die Ausgabe nicht ident. Die restlichen Funktionen sind selbsterklärend.

Adjazenzmatrix

Grundsätzlich funktioniert die Adjazenzmatrix mithilfe eines einzigen Structs, welcher die maximale Anzahl der Knoten, die momentane Anzahl der Knoten und zwei Arrays enthält; ein Array mit den Payloads und eines mit den Verbindungen. Dabei entspricht der Index in dem Payload-Array dem Index der Verbindung. Das Payload-Array ist als Char*-Array ausgeführt, das Verbindungs-Array als Bool-Array.
Beim Hinzufügen von Knoten muss darauf geachtet werden, dass die allokierte Matrixgröße nicht überschritten wird. Bevor dies geschied, wird mithilfe einer Helferfunktion zwei neue Matrizen allokiert[1], die alten Matrizen hineinkopiert und diese dann auch freigegeben. Eine weitere "versteckte"Helferfunktion nennt sich *mapPayloadToIndex*, welche einen, vom Nutzer gegebenen Knotennamen, auf einen Index der Payload und damit der Matrix mappt. Liefert bei keinem gefundenen Knoten -1 zurück.
Um eine ähnliche Funktion handelt es sich bei *mapIndexToAddr*, welche als Helferfunktion für das Matrix-Array gedacht ist. Dabei wird jeweils *from* und *to* gegeben und die Funktion liefert die Addresse des Bool-Bits zurück. Dies spart Zeit bei Iterationen, da nicht immer manuell die Adresse ausgerechnet werden muss, da aufgrund der dynamischen Allokierung des Arrays das Array nur 1-dimensional ausgeführt ist.

Adjazenzliste

Die Adjazenzliste implementiert im Gegensatz zu 5.1.1 zwei Datentypen:

1. node
2. connection

Sobald ein Knoten hinzugefügt wird, wird dieser einfach an den letzten Knoten hinzugefügt. So entsteht eine einseitig verknüpfte Liste, was jedoch für unsere Zwecke völlig ausreicht. Weiters enthält jeder Knoten eine weitere Liste, welche die Verbindungen zu den anderen Knoten enthält. Somit gibt es keine ßentrale"Verwaltung mehr, sondern es muss nur noch darauf geachtet werden, dass jeder Knoten und jede Liste fehlerfrei allokiert wird. Für die SZerstörung"wird rekursiv über die Knotenliste iteriert, bis man am letzten Element angekommen ist. Dort wird dann wiederrum über die Verbindungsliste iteriert bis dort das Ende erreicht ist. Dann wird rekursiv der Speicher der einzelnen Knoten freigegeben. Durch die Rekursion wird sichergestellt, dass jedes Element freigegeben wird.
Als Hilfsfunktionen sind *deleteNodeByPtr* und *deleteConnectionList* vorhanden, welche die Löschung eines Knotens oder einer Verbindungsliste übernehmen. Im Falle von *deleteConnectionList* geschied dies rekursiv.

[1] *payload* und *matrix*

5.1.2 Implementierung

Listing 5.1: am_adt.h

```
1  //
2  // Created by khp on 08.11.18.
3  //
4
5  #ifndef PROJECT_AM_ADT_H
6  #define PROJECT_AM_ADT_H
7
8  #include <stdbool.h>
9
10 typedef struct amStruct_t *adtPtr_t;
11 struct amStruct_t {
12     int maxNodes;
13     int nodesCount;
14     char **payload;
15     bool *matrix;
16 };
17
18 adtPtr_t init(void);
19
20 void destroy(adtPtr_t);
21
22 void createNode(adtPtr_t, char *);
23
24 bool createEdge(adtPtr_t, char *, char *);
25
26 void deleteNode(adtPtr_t, char *);
27
28 bool deleteEdge(adtPtr_t, char *, char *);
29
30 void print(adtPtr_t);
31
32 #endif //PROJECT_AM_ADT_H
```

Listing 5.2: am_adt.c

```
1  //
2  // Created by khp on 08.11.18.
3  //
4  #include <stdbool.h>
5  #include <string.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include "am_adt.h"
9
10 #define INITIAL_SIZE 2
11 #define SIZE_MULTIPLIER 2
12
13 int mapPayloadToIndex(char **payload, char *searched, int nodesCount) {
14     for (int i = 0; i < nodesCount; ++i) {
15         if (strcmp(*(payload + i), searched) == 0) {
16             return i;
17         }
18     }
```

```
19      return -1;
20  }
21
22  void copyPayload(char **from, char **to, int size) {
23      for (int i = 0; i < size; ++i) {
24          *(to + i) = *(from + i);
25      }
26  }
27
28  bool *mapIndexToAddr(bool *matrix, int from, int to, int nodesCount, int maxNodes) {
29      if (from > nodesCount || to > nodesCount) {
30          printf("Invalid index!");
31          return NULL;
32      }
33      bool *memAddr = matrix + (from * maxNodes) + to;
34      return memAddr;
35  }
36
37  void copyMatrix(bool *from, int fromMaxNodes, bool *to, int toMaxNodes, int
        nodesCount) {
38      for (int i = 0; i < (nodesCount); ++i) {
39          for (int j = 0; j < nodesCount; ++j) {
40              *mapIndexToAddr(to, i, j, nodesCount, toMaxNodes) = *mapIndexToAddr(from
        , i, j, nodesCount, fromMaxNodes);
41          }
42      }
43  }
44
45  void resizeAM(adtPtr_t amPtr, int size) { //extra ned im Header
46      char **oldPayload = amPtr->payload;
47      bool *oldMatrix = amPtr->matrix;
48      int oldMaxNodes = amPtr->maxNodes;
49      amPtr->maxNodes = size;
50      amPtr->payload = (char **) malloc((unsigned int)amPtr->maxNodes * sizeof(char *)
        );
51      amPtr->matrix = (bool *) malloc((unsigned int)(amPtr->maxNodes * amPtr->maxNodes
        ) * sizeof(bool));
52      copyPayload(oldPayload, amPtr->payload, amPtr->nodesCount);
53      copyMatrix(oldMatrix, oldMaxNodes, amPtr->matrix, amPtr->maxNodes, amPtr->
        nodesCount);
54      free(oldMatrix);
55      free(oldPayload);
56  }
57
58  adtPtr_t init(void) {
59      adtPtr_t amPtr = malloc(sizeof(struct amStruct_t));
60      amPtr->nodesCount = 0;
61      resizeAM(amPtr, INITIAL_SIZE);
62      return amPtr;
63  }
64
65  void destroy(adtPtr_t amPtr) {
66      for (int i = 0; i < amPtr->nodesCount; ++i) {
67          free(*(amPtr->payload + i));
68      }
69      free(amPtr->payload);
70      free(amPtr->matrix);
```

```
71      free(amPtr);
72  }
73
74  void createNode(adtPtr_t amPtr, char *name) {
75      char *tString = malloc(strlen(name) * sizeof(char));
76      strcpy(tString, name);
77      if (amPtr->nodesCount == amPtr->maxNodes) {
78          resizeAM(amPtr, amPtr->maxNodes * SIZE_MULTIPLIER);
79      }
80      *(amPtr->payload + amPtr->nodesCount) = tString;
81      amPtr->nodesCount++;
82
83      for (int i = 0; i < amPtr->nodesCount; ++i) {
84          *(mapIndexToAddr(amPtr->matrix, amPtr->nodesCount, i, amPtr->nodesCount,
        amPtr->maxNodes)) = false;
85      }
86      for (int i = 0; i < amPtr->nodesCount; ++i) {
87          *(mapIndexToAddr(amPtr->matrix, i, amPtr->nodesCount, amPtr->nodesCount,
        amPtr->maxNodes)) = false;
88      }
89  }
90
91  bool createEdge(adtPtr_t amPtr, char *from, char *to) {
92      int fromIndex = mapPayloadToIndex(amPtr->payload, from, amPtr->nodesCount);
93      int toIndex = mapPayloadToIndex(amPtr->payload, to, amPtr->nodesCount);
94      if (fromIndex == -1) {
95          printf("%s does not exist!", from);
96          return false;
97      } else if (toIndex == -1) {
98          printf("%s does not exist!", to);
99          return false;
100     }
101
102     *mapIndexToAddr(amPtr->matrix, fromIndex, toIndex, amPtr->nodesCount, amPtr->
        maxNodes) = true;
103     return true;
104 }
105
106 void deleteNode(adtPtr_t amPtr, char *name) {
107     //swap with last index and reduce nodesCount
108     int removeIndex = mapPayloadToIndex(amPtr->payload, name, amPtr->nodesCount);
109     if (removeIndex == -1) {
110         printf("%s does not exist!", name);
111     }
112     for (int i = 0; i < amPtr->nodesCount; ++i) {
113         *(mapIndexToAddr(amPtr->matrix, removeIndex, i, amPtr->nodesCount, amPtr->
        maxNodes)) = *(mapIndexToAddr(
114                 amPtr->matrix, amPtr->nodesCount - 1, i, amPtr->nodesCount, amPtr->
        maxNodes));
115     }
116     for (int i = 0; i < amPtr->nodesCount; ++i) {
117         *(mapIndexToAddr(amPtr->matrix, i, removeIndex, amPtr->nodesCount, amPtr->
        maxNodes)) = *(mapIndexToAddr(
118                 amPtr->matrix, i, amPtr->nodesCount - 1, amPtr->nodesCount, amPtr->
        maxNodes));
119     }
120     *(amPtr->payload + removeIndex) = *(amPtr->payload + (amPtr->nodesCount - 1));
```

```
121     free(*(amPtr->payload + (amPtr->nodesCount)));
122     amPtr->nodesCount--;
123 }
124 
125 bool deleteEdge(adtPtr_t amPtr, char *from, char *to) {
126     int fromIndex = mapPayloadToIndex(amPtr->payload, from, amPtr->nodesCount);
127     int toIndex = mapPayloadToIndex(amPtr->payload, to, amPtr->nodesCount);
128     if (fromIndex == -1) {
129         printf("%s does not exist!", from);
130         return false;
131     } else if (toIndex == -1) {
132         printf("%s does not exist!", to);
133         return false;
134     }
135 
136     *mapIndexToAddr(amPtr->matrix, fromIndex, toIndex, amPtr->nodesCount, amPtr->
        maxNodes) = false;
137     return true;
138 }
139 
140 void print(adtPtr_t amPtr) {
141     printf("    ");
142     for (int k = 0; k < amPtr->nodesCount; ++k) {
143         printf("%s", *(amPtr->payload + k));
144     }
145     printf("\n");
146     for (int i = 0; i < amPtr->nodesCount; ++i) {
147         printf("%s | ", *(amPtr->payload + i));
148         for (int j = 0; j < amPtr->nodesCount; ++j) {
149             printf("%i", *(mapIndexToAddr(amPtr->matrix, i, j, amPtr->nodesCount,
        amPtr->maxNodes)));
150         }
151         printf("\n");
152     }
153 }
```

Listing 5.3: al_adt.h

```
1  //
2  // Created by khp on 08.11.18.
3  //
4  
5  #ifndef PROJECT_AL_ADT_H
6  #define PROJECT_AL_ADT_H
7  
8  #include <stdbool.h>
9  
10 typedef struct node *adtPtr_t;
11 typedef struct connection *connectionPtr_t;
12 struct node {
13     char *payload;
14     connectionPtr_t nextConnection;
15     adtPtr_t nextNode;
16 };
17 struct connection {
18     adtPtr_t node;
19     connectionPtr_t nextConnection;
```

```
20 };
21
22 adtPtr_t init(void);
23
24 void destroy(adtPtr_t);
25
26 void createNode(adtPtr_t, char *);
27
28 bool createEdge(adtPtr_t, char *, char *);
29
30 void deleteNode(adtPtr_t, char *);
31
32 bool deleteEdge(adtPtr_t, char *, char *);
33
34 void print(adtPtr_t);
35
36 #endif //PROJECT_AL_ADT_H
```

Listing 5.4: al_adt.c

```
1  //
2  // Created by khp on 08.11.18.
3  //
4
5  #include <stdbool.h>
6  #include <string.h>
7  #include <stdlib.h>
8  #include <stdio.h>
9  #include "al_adt.h"
10
11 void initNode(adtPtr_t nodePtr) {
12     nodePtr->payload = "";
13     nodePtr->nextConnection = NULL;
14     nodePtr->nextNode = NULL;
15 }
16
17 adtPtr_t init(void) {
18     adtPtr_t newNode = malloc(sizeof(struct node));
19     initNode(newNode);
20     return newNode;
21 }
22
23 void createNode(adtPtr_t nodePtr, char *name) {
24     if (strcmp(nodePtr->payload, "") == 0) { //first element
25         nodePtr->payload = name;
26         return;
27     }
28     adtPtr_t newNode = malloc(sizeof(struct node));
29     initNode(newNode);
30     newNode->payload = name;
31     while (nodePtr->nextNode != NULL)
32         nodePtr = nodePtr->nextNode;
33     nodePtr->nextNode = newNode;
34 }
35
36 adtPtr_t getNodePtr(adtPtr_t nodePtr, char *name) {
37     while (nodePtr != NULL) {
```

```
38          if (strcmp(nodePtr->payload, name) == 0) {
39              return nodePtr;
40          }
41          nodePtr = nodePtr->nextNode;
42      }
43      return NULL;
44  }
45
46  bool createEdge(adtPtr_t nodePtr, char *from, char *to) {
47      adtPtr_t fromPtr;
48      adtPtr_t toPtr;
49      connectionPtr_t newConnection = malloc(sizeof(struct connection));
50      newConnection->nextConnection = NULL;
51      fromPtr = getNodePtr(nodePtr, from);
52      toPtr = getNodePtr(nodePtr, to);
53
54      if (fromPtr == NULL) {
55          printf("%s does not exist!\n", from);
56          return false;
57      }
58      if (toPtr == NULL) {
59          printf("%s does not exist!\n", to);
60          return false;
61      }
62
63      newConnection->node = toPtr;
64
65      if (fromPtr->nextConnection == NULL) {
66          fromPtr->nextConnection = newConnection;
67      } else {
68          connectionPtr_t fromConnection = fromPtr->nextConnection;
69          while (fromConnection->nextConnection != NULL)
70              fromConnection = fromConnection->nextConnection;
71
72          fromConnection->nextConnection = newConnection;
73      }
74      return true;
75  }
76
77  void deleteConnectionList(connectionPtr_t connectionPtr) {
78      if (connectionPtr != NULL) {
79          deleteConnectionList(connectionPtr->nextConnection);
80          free(connectionPtr);
81      }
82  }
83
84  void deleteNodeByPtr(adtPtr_t originPtr, adtPtr_t delPtr) {
85      adtPtr_t tPtr = originPtr;
86      while (tPtr->nextNode != delPtr)
87          tPtr = tPtr->nextNode;
88
89      tPtr->nextNode = tPtr->nextNode->nextNode;
90
91      tPtr = originPtr;
92      while (tPtr->nextNode != NULL) {
93          connectionPtr_t nextConnection = tPtr->nextConnection;
94          connectionPtr_t curConnection = tPtr;
```

```
95          bool finished = false;
96          while (nextConnection != NULL) {
97              if (nextConnection->node == delPtr) {
98                  connectionPtr_t tConnection = nextConnection;
99                  curConnection->nextConnection = nextConnection->nextConnection;
100                 free(tConnection);
101                 finished = true;
102             }
103             if (finished == true)
104                 break;
105             curConnection = nextConnection;
106             nextConnection = nextConnection->nextConnection;
107         }
108         tPtr = tPtr->nextNode;
109     }
110
111     deleteConnectionList(delPtr->nextConnection);
112     free(delPtr);
113 }
114
115 void deleteNode(adtPtr_t nodePtr, char *name) {
116     adtPtr_t namePtr;
117     adtPtr_t originPtr = nodePtr;
118
119     namePtr = getNodePtr(nodePtr, name);
120
121     if (namePtr == NULL) {
122         printf("Name does not exist!\n");
123         return;
124     }
125
126     deleteNodeByPtr(originPtr, namePtr);
127 }
128
129 bool deleteEdge(adtPtr_t nodePtr, char *from, char *to) {
130     adtPtr_t fromPtr;
131     adtPtr_t toPtr;
132
133     fromPtr = getNodePtr(nodePtr, from);
134     toPtr = getNodePtr(nodePtr, to);
135
136     if (fromPtr == NULL) {
137         printf("%s does not exist!\n", from);
138         return false;
139     }
140     if (toPtr == NULL) {
141         printf("%s does not exist!\n", to);
142         return false;
143     }
144
145     connectionPtr_t curConnection = fromPtr;
146     while (curConnection != NULL) {
147         connectionPtr_t nextConnection = curConnection->nextConnection;
148         bool finished = false;
149         if (nextConnection != NULL) {
150             if (nextConnection->node == toPtr) {
151                 curConnection->nextConnection = nextConnection->nextConnection;
```

```
152                 free(nextConnection);
153                 finished = true;
154             }
155         }
156         if (finished == true)
157             return true;
158         curConnection = curConnection->nextConnection;
159     }
160     return false;
161 }
162
163 void print(adtPtr_t node) {
164     while (node != NULL) {
165         printf("%s : ", node->payload);
166         connectionPtr_t nextConnection = node->nextConnection;
167         while (nextConnection != NULL) {
168             printf("%s ", nextConnection->node->payload);
169             nextConnection = nextConnection->nextConnection;
170         }
171         printf("\n");
172         node = node->nextNode;
173     }
174 }
175
176 void destroy(adtPtr_t node){
177     if(node != NULL) {
178         destroy(node->nextNode);
179         deleteConnectionList(node->nextConnection);
180         free(node);
181     }
182 }
```

5.1.3 Testen

Listing 5.5: test.c

```
1  //
2  // Created by khp on 08.11.18.
3  //
4
5  #if USE_LIST
6  #include "al_adt.h"
7  #if SORT
8  #include "../top/top.h"
9  #endif
10 #else
11 #include "am_adt.h"
12 #endif
13
14 #include <stdlib.h>
15 #include <stdio.h>
16
17 int main() {
18     printf("## ORIGINAL ARRAY ##\n");
19     adtPtr_t adtPtr = init();
20     createNode(adtPtr, "A");
21     createNode(adtPtr, "B");
22     createNode(adtPtr, "C");
23     createNode(adtPtr, "D");
24     createNode(adtPtr, "E");
25     createEdge(adtPtr, "A", "B");
26     createEdge(adtPtr, "A", "C");
27     createEdge(adtPtr, "A", "D");
28     createEdge(adtPtr, "A", "E");
29     createEdge(adtPtr, "B", "D");
30     createEdge(adtPtr, "C", "B");
31     createEdge(adtPtr, "C", "E");
32     print(adtPtr);
33
34     printf("\n\n## TEST 1 - REMOVAL OF 1 NODE ##\n");
35     deleteNode(adtPtr, "E");
36     printf("REMOVED 1 NODE (E):\n");
37     print(adtPtr);
38     destroy(adtPtr);
39
40     printf("\n\n## TEST 2 - REMOVAL OF 1 EDGE ##\n");
41     adtPtr = init();
42     createNode(adtPtr, "A");
43     createNode(adtPtr, "B");
44     createNode(adtPtr, "C");
45     createNode(adtPtr, "D");
46     createNode(adtPtr, "E");
47     createEdge(adtPtr, "A", "B");
48     createEdge(adtPtr, "A", "C");
49     createEdge(adtPtr, "A", "D");
50     createEdge(adtPtr, "A", "E");
51     createEdge(adtPtr, "B", "D");
52     createEdge(adtPtr, "C", "B");
53     createEdge(adtPtr, "C", "E");
```

```
54      deleteEdge(adtPtr, "C", "E");
55      printf("REMOVED 1 EDGE (C-E):\n");
56      print(adtPtr);
57      destroy(adtPtr);
58
59      printf("\n\n## TEST 3 - REMOVAL OF 1 NODE (NOT EXISTING) ##\n");
60      adtPtr = init();
61      createNode(adtPtr, "A");
62      createNode(adtPtr, "B");
63      createNode(adtPtr, "C");
64      createNode(adtPtr, "D");
65      createNode(adtPtr, "E");
66      createEdge(adtPtr, "A", "B");
67      createEdge(adtPtr, "A", "C");
68      createEdge(adtPtr, "A", "D");
69      createEdge(adtPtr, "A", "E");
70      createEdge(adtPtr, "B", "D");
71      createEdge(adtPtr, "C", "B");
72      createEdge(adtPtr, "C", "E");
73      deleteNode(adtPtr, "X");
74      printf("THROWS ERROR, DOESN'T ALTER NODES\n");
75      print(adtPtr);
76      destroy(adtPtr);
77
78      printf("\n\n## TEST 4 - REMOVAL OF 1 EDGE (NOT EXISTING) ##\n");
79      adtPtr = init();
80      createNode(adtPtr, "A");
81      createNode(adtPtr, "B");
82      createNode(adtPtr, "C");
83      createNode(adtPtr, "D");
84      createNode(adtPtr, "E");
85      createEdge(adtPtr, "A", "B");
86      createEdge(adtPtr, "A", "C");
87      createEdge(adtPtr, "A", "D");
88      createEdge(adtPtr, "A", "E");
89      createEdge(adtPtr, "B", "D");
90      createEdge(adtPtr, "C", "B");
91      createEdge(adtPtr, "C", "E");
92      deleteEdge(adtPtr, "X", "Y");
93      printf("THROWS ERROR, DOESN'T ALTER NODES\n");
94      print(adtPtr);
95      destroy(adtPtr);
96
97      printf("\n\n## TEST 5 - PRINTING EMTPY ARRAY ##\n");
98      adtPtr = init();
99      printf("PRINTS EMPTY ARRAY\n");
100     print(adtPtr);
101     destroy(adtPtr);
102
103 #if SORT
104     printf("\n\n## TEST 6 - SORTING ##\n");
105     adtPtr = init();
106     createNode(adtPtr, "A");
107     createNode(adtPtr, "B");
108     createNode(adtPtr, "C");
109     createNode(adtPtr, "D");
110     createNode(adtPtr, "E");
```

```
111     createEdge(adtPtr, "A", "B");
112     createEdge(adtPtr, "A", "C");
113     createEdge(adtPtr, "A", "D");
114     createEdge(adtPtr, "A", "E");
115     createEdge(adtPtr, "B", "D");
116     createEdge(adtPtr, "C", "B");
117     createEdge(adtPtr, "C", "E");
118     printf("SORTING ARRAY:\n");
119     topological_sort(adtPtr);
120     print(adtPtr);
121 
122     printf("\n\n## TEST 7 - SORTING - SAME AMOUNT OF NODES ##\n");
123     adtPtr = init();
124     createNode(adtPtr, "A");
125     createNode(adtPtr, "B");
126     createNode(adtPtr, "C");
127     createNode(adtPtr, "D");
128     createNode(adtPtr, "E");
129     createEdge(adtPtr, "A", "B");
130     createEdge(adtPtr, "A", "C");
131     createEdge(adtPtr, "A", "D");
132     createEdge(adtPtr, "A", "E");
133     createEdge(adtPtr, "B", "D");
134     createEdge(adtPtr, "B", "E");
135     createEdge(adtPtr, "C", "B");
136     createEdge(adtPtr, "C", "E");
137     printf("SORTING ARRAY:\n");
138     topological_sort(adtPtr);
139     print(adtPtr);
140 
141     printf("\n\n## TEST 8 - SORTING - NO CONNECTIONS ##\n");
142     adtPtr = init();
143     createNode(adtPtr, "A");
144     createNode(adtPtr, "B");
145     createNode(adtPtr, "C");
146     createNode(adtPtr, "D");
147     createNode(adtPtr, "E");
148     printf("SORTING ARRAY:\n");
149     topological_sort(adtPtr);
150     print(adtPtr);
151 #endif
152 }
```

Output der Matrix

```
## ORIGINAL ARRAY ##
    ABCDE
A | 01111
B | 00010
C | 01001
D | 00000
E | 00000


## TEST 1 - REMOVAL OF 1 NODE ##
```

```
REMOVED 1 NODE (E):
    ABCD
A | 0111
B | 0001
C | 0100
D | 0000


## TEST 2 - REMOVAL OF 1 EDGE ##
REMOVED 1 EDGE (C-E):
    ABCDE
A | 2401111
B | 083010
C | 01000
D | 00000
E | 00000


## TEST 3 - REMOVAL OF 1 NODE (NOT EXISTING) ##
X does not exist!THROWS ERROR, DOESN'T ALTER NODES
    ABCD
A | 96111
B | 08301
C | 0100
D | 0000
```

Die Memoryallokierung dieser Implementierung ist fehlerhaft, daher schlägt das Programm vorzeitig fehl.

Output der Liste

```
## ORIGINAL ARRAY ##
A : B C D E
B : D
C : B E
D :
E :


## TEST 1 - REMOVAL OF 1 NODE ##
REMOVED 1 NODE (E):
A : B C D
B : D
C : B
D :
```

```
## TEST 2 - REMOVAL OF 1 EDGE ##
REMOVED 1 EDGE (C-E):
A : B C D E
B : D
C : B
D :
E :


## TEST 3 - REMOVAL OF 1 NODE (NOT EXISTING) ##
Name does not exist!
THROWS ERROR, DOESN'T ALTER NODES
A : B C D E
B : D
C : B E
D :
E :


## TEST 4 - REMOVAL OF 1 EDGE (NOT EXISTING) ##
X does not exist!
THROWS ERROR, DOESN'T ALTER NODES
A : B C D E
B : D
C : B E
D :
E :


## TEST 5 - PRINTING EMTPY ARRAY ##
THROWS ERROR, DOESN'T ALTER NODES
 :
```

5.2 Topologisches Sortieren

5.2.1 Lösungsidee

In dieser Nummer ist es die Aufgabe, einen gegebenen *ADT* zu sortieren. In dieser Implementierung geschied dies mit der Listenimplementierung5.1.1. Dabei wird der Ursprungspointer mitgegeben, als Rückgabewert wird ein neuer Ursprungspointer zurückgegeben[2]. Danach wird ein Hilfsarray erstellt, welches auf einem eigens dafür erstellten *struct* basiert. Dieses enthält den Pointer zu der Node und die Anzahl der abgehenden Verbindungen. Danach wird dieses Array gefüllt, vorerst unsortiert, nur mit Pointer und den gezählten Nodes. Danch wird mithilfe des Merge-Sorts[3] dieses Array sortiert, sodass die Nodes mit den meisten abgehenden Verbindungen zuerst kommen. Danach wird über dieses Array drüberiteriert und jedes Element mit dem jeweilig nächsten verknüpft. Danach wird das erste Element des Hilfsarrays zurückgegeben, da dieses die höchsten abgehenden Verbindungen hat.

5.2.2 Implementierung

Listing 5.6: top.h

```
1  //
2  // Created by khp on 08.11.18.
3  //
4
5  #ifndef PROJECT_TOP_H
6  #define PROJECT_TOP_H
7
8  #include "../adt/al_adt.h"
9
10 adtPtr_t topological_sort(adtPtr_t);
11
12 #endif //PROJECT_TOP_H
```

Listing 5.7: top.c

```
1  //
2  // Created by khp on 08.11.18.
3  //
4
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <math.h>
8  #include "../adt/al_adt.h"
9
10 typedef struct nodeConnections* nodeConnectionsPtr_t;
11
12 struct nodeConnections{
13     adtPtr_t node;
14     int connections;
15 };
```

[2] Dieser kann mit dem ursprünglichen Pointer ident sein.
[3] Dieser wurde aus Übung 2 [2.3] übernommen und leicht abgeändert.

```
16
17 int countConnections(connectionPtr_t connectionPtr) {
18     int i = 0;
19     while(connectionPtr != NULL)  {
20         connectionPtr = connectionPtr->nextConnection;
21         i++;
22     }
23     return i;
24 }
25
26 int countNodes(adtPtr_t nodePtr) {
27     int i = 0;
28     while(nodePtr != NULL) {
29         nodePtr = nodePtr->nextNode;
30         i++;
31     }
32     return i;
33 }
34
35 void fillNodeConnections(adtPtr_t nodePtr, nodeConnectionsPtr_t nodeConnectionsPtr,
       int nodeCount) {
36     for (int i = 0; i < nodeCount; ++i) {
37         (nodeConnectionsPtr + i)->node = nodePtr;
38         (nodeConnectionsPtr + i)->connections = countConnections(nodePtr->
       nextConnection);
39         nodePtr = nodePtr->nextNode;
40     }
41 }
42
43 void sortNodeConnections(nodeConnectionsPtr_t nodeConnectionsPtr, int nodeCount) {
44     if(nodeCount <= 1)
45         return;
46     else {
47         nodeConnectionsPtr_t left = malloc((unsigned int)nodeCount * sizeof(struct
       nodeConnections));
48         nodeConnectionsPtr_t right = malloc((unsigned int)nodeCount * sizeof(struct
       nodeConnections));
49         int leftMax = (int)ceil((double)nodeCount/2);
50         int rightMax = (int)floor((double)nodeCount/2);
51
52         for(int i = 0; i < leftMax; i++)
53             *(left + i) = *(nodeConnectionsPtr + i);
54         for(int i = 0; i < rightMax; i++)
55             *(right + i) = *(nodeConnectionsPtr + leftMax + i);
56
57         sortNodeConnections(left, leftMax);
58         sortNodeConnections(right, rightMax);
59
60         int leftIndex = 0;
61         int rightIndex = 0;
62
63         for(int i = 0; i < nodeCount; i++) {
64             if(leftIndex >= leftMax) { //left array fully in a[]
65                 *(nodeConnectionsPtr + i) = *(right + rightIndex);
66                 rightIndex++;
67             } else if(rightIndex >= rightMax) { //right array fully in a[]
68                 *(nodeConnectionsPtr + i) = *(left + leftIndex);
```

```
69                  leftIndex++;
70              } else {
71                  if((left + leftIndex)->connections > (right + rightIndex)->
        connections) {
72                      *(nodeConnectionsPtr + i) = *(left + leftIndex);
73                      leftIndex++;
74                  } else {
75                      *(nodeConnectionsPtr + i) = *(right + rightIndex);
76                      rightIndex++;
77                  }
78              }
79          }
80          free(left);
81          free(right);
82      }
83  }
84
85  adtPtr_t reconnectNodes(nodeConnectionsPtr_t nodeConnectionsPtr, int nodeCount) {
86      adtPtr_t returnPtr = nodeConnectionsPtr->node;
87      int i;
88      for (i = 0; i < nodeCount-1; ++i) {
89          nodeConnectionsPtr->node->nextNode = (nodeConnectionsPtr+1)->node;
90          nodeConnectionsPtr = (nodeConnectionsPtr+1);
91      }
92      (nodeConnectionsPtr)->node->nextNode = NULL;
93      return returnPtr;
94  }
95
96  adtPtr_t topological_sort(adtPtr_t nodePtr) {
97      int nodeCount = countNodes(nodePtr);
98      nodeConnectionsPtr_t nodeConnectionsPtr = malloc((unsigned int)nodeCount* sizeof
        (struct nodeConnections));
99
100     fillNodeConnections(nodePtr, nodeConnectionsPtr, nodeCount);
101     sortNodeConnections(nodeConnectionsPtr, nodeCount);
102     nodePtr = reconnectNodes(nodeConnectionsPtr, nodeCount);
103     free(nodeConnectionsPtr);
104     return nodePtr;
105 }
```

5.2.3 Testen

Listing 5.8: Sorting-Teil, in test.c integriert

```
printf("\n\n## TEST 6 - SORTING ##\n");
adtPtr = init();
createNode(adtPtr, "A");
createNode(adtPtr, "B");
createNode(adtPtr, "C");
createNode(adtPtr, "D");
createNode(adtPtr, "E");
createEdge(adtPtr, "A", "B");
createEdge(adtPtr, "A", "C");
createEdge(adtPtr, "A", "D");
createEdge(adtPtr, "A", "E");
createEdge(adtPtr, "B", "D");
createEdge(adtPtr, "C", "B");
createEdge(adtPtr, "C", "E");
printf("SORTING ARRAY:\n");
topological_sort(adtPtr);
print(adtPtr);

printf("\n\n## TEST 7 - SORTING - SAME AMOUNT OF NODES ##\n");
adtPtr = init();
createNode(adtPtr, "A");
createNode(adtPtr, "B");
createNode(adtPtr, "C");
createNode(adtPtr, "D");
createNode(adtPtr, "E");
createEdge(adtPtr, "A", "B");
createEdge(adtPtr, "A", "C");
createEdge(adtPtr, "A", "D");
createEdge(adtPtr, "A", "E");
createEdge(adtPtr, "B", "D");
createEdge(adtPtr, "B", "E");
createEdge(adtPtr, "C", "B");
createEdge(adtPtr, "C", "E");
printf("SORTING ARRAY:\n");
topological_sort(adtPtr);
print(adtPtr);

printf("\n\n## TEST 8 - SORTING - NO CONNECTIONS ##\n");
adtPtr = init();
createNode(adtPtr, "A");
createNode(adtPtr, "B");
createNode(adtPtr, "C");
createNode(adtPtr, "D");
createNode(adtPtr, "E");
printf("SORTING ARRAY:\n");
topological_sort(adtPtr);
print(adtPtr);
```

Output des Tests

```
## TEST 6 - SORTING ##
SORTING ARRAY:
A : B C D E
```

```
C : B E
B : D
E :
D :


## TEST 7 - SORTING - SAME AMOUNT OF NODES ##
SORTING ARRAY:
A : B C D E
C : B E
B : D E
E :
D :


## TEST 8 - SORTING - NO CONNECTIONS ##
SORTING ARRAY:
A :
```