Lösungen zur Vorlesung
GRUNDLAGEN DER INFORMATIK I
Studiengang Elektrotechnik
WS 02/03

AG Betriebssysteme FB3
Kirsten Berkenkötter

Lösung Übungszettel 5

1 Aufgabe 1: Mehrdimensionale Arrays

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

#define MAX 5

void matrixadd(int a[MAX][MAX], int b[MAX][MAX], int result[MAX][MAX])
{
   int i,j;

   for(i = 0; i < MAX; i++)
      for (j = 0; j < MAX; j++)
         result[i][j] = a[i][j] + b[i][j];
}

int main()
{
   int matrixa[MAX][MAX], matrixb[MAX][MAX], result[MAX][MAX];
   int i,j;

   srand(time(NULL));

   for(i = 0; i < MAX; i++)
      for (j = 0; j < MAX; j++)
      {
         matrixa[i][j] = rand()%10;
         matrixb[i][j] = rand()%10;
      }

   for(i = 0; i < MAX; i++)
   {
      for (j= 0; j <MAX; j++)
      {
         printf("%3d ", matrixa[i][j]);
      }
      printf("\n");
   }
   printf("\n");
```

```
   for(i = 0; i < MAX; i++)
   {
      for(j= 0; j < MAX; j++)
         printf("%3d ", matrixb[i][j]);
      printf("\n");
   }
   printf("\n");

   matrixadd(matrixa,matrixb, result);

   for(i = 0; i < MAX; i++)
   {
      for(j= 0; j < MAX; j++)
         printf("%3d ", result[i][j]);
      printf("\n");
   }
}
```

2 Aufgabe 2: Doppelt verkettete Listen

```
/* Doppelt verkettete Liste */

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

#include <stdio.h>

// definiert einen Datentyp, mit dem die Daten
// der Liste gespeichert werden koennen
// key gibt den Schluesselwert des Elements
// (damit die Elemente auch sortiert werden koennen)
// name sind die eigentlichen Daten
typedef struct {
   int key;
   char name[10];
} userdata_t;

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

// definiert einen Struktur, welche die Elemente
// der Liste bilden
// zunaechst werden die eigentlichen Daten benoetigt
// (zuvor definierter Datentyp)
// ausserdem muss das naechste und das vorherige Element
// der Liste gemerkt werden:
// Rekursive Datentypen: Die Zeiger next und
// previous verweisen auf einen Eintragen desselben Typs
// list_element_t
struct list_element_t {
   //eigentliche Daten
   userdata_t data;
```

```
   // Zeiger auf naechstes Listenelement (oder NULL)
   struct list_element_t *next;
   // Zeiger auf vorhergehendes Listenelement (oder NULL)
   struct list_element_t *previous;
};

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

// ein Datentyp fuer die Verwaltung von Listen
// genannt Listhandler
typedef struct {
   // Zeiger auf Listenanfang
   struct list_element_t *head;
   // Zeiger auf DEN VORGAENGER des aktuell zu
   //bearbeitenden Listenelements
   struct list_element_t *thisElement;
   // Zeiger auf das letzte Listenelement
   struct list_element_t *lastElement;
} list_handle_t;

/*-------------------------------------------------------*/
/* Es folgen die Funktionen zur Verwaltung von Listen */
/*-------------------------------------------------------*/

// Funktion zum Erzeugen einer Liste
// Rueckgabewert: Zeiger auf Listhandler
// keine Parameter
list_handle_t *createList(void) {
   // einen Zeiger auf einen Listhandler erzeugen
   list_handle_t *h;

   // malloc() erzeugt dynamisch Speicher
   // und gibt einen Pointer auf den Speicheranfang zurueck
   h =(list_handle_t *)malloc(sizeof(list_handle_t));

   // jetzt ist Platz im Speicher reserviert
   // markiere, dass Liste leer ist
   // Anfang, Ende und Vorgaenger des aktuelles Element sind NULL,
   // d.h. vorerst nicht belegt
   h->head =NULL;
   h->thisElement = NULL;
   h->lastElement = NULL;

   // gib den neuen Handler zurueck
   return h;
}

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

//Funktion, die Elemente an die Liste anhaengt
//kein Rueckgabewert
```

```
//Parameter: Listhandler (fuer die Liste, an die angehaengt werden soll
//Datensatz, der im neuen Element der Liste stehen soll
void appendList(list_handle_t *h, userdata_t d) {

   //1. Schritt: Speicher fuer neues Listenelement
   // allokieren
   struct list_element_t *l;
   l =(struct list_element_t *)malloc(sizeof(struct list_element_t));

   // 2. Schritt: Nutzdaten d in neues Listenelement kopieren
   l->data = d;

   // 3. Schritt: Verkettung - bisher letztes
   // Listenelement muss neuen Eintrag l als
   // Nachfolger bekommen. l hat keinen Nachfolger
   // bisher letztes Element- falls vorhanden - wird
   // Vorgaenger von l
   l->next = NULL;

   // Unterscheide zwischen leerer und gefuellter Liste
   // 1. Fall: leere Liste
   if ( h->head == NULL ) {
      //es existiert kein Vorgaenger, also wird l head
      //und der Vorgaenger ist NULL
      l->previous = NULL;
      h->head = l;
      h->thisElement = NULL; // zu head existiert
                        // kein Vorgaenger
   }
   // 2. Fall: es existiert ein Element
   else {
      // l ist naechstes Element des bisher letzten
      h->lastElement->next = l;
      // bisher letztes Element wird Vorgaenger von l
      l->previous = h->lastElement;
   }

   // nach dem Anhaengen ist das neue Element auch das letze
   h->lastElement = l;
}

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

// Funktion zum Einfuegen in die Liste
// kein Rueckgabewert
// Parameter: Listhandler fuer die Liste, in die eingefuegt werden soll
// Datensatz fuer das neue Element der Liste
void insertList(list_handle_t *h, userdata_t d) {

   //1. Schritt: Speicher fuer neues Listenelement allokieren
   struct list_element_t *l;
```

```
struct list_element_t *temp;
l = (struct list_element_t *)malloc(sizeof(struct list_element_t));

// 2. Schritt: Nutzdaten d in neues Listenelement kopieren
l->data = d;

// 3. Schritt: Verketten des neuen Elements

// 1. Fall: die Liste ist leer, das neue Element wird das erste
// jetzt ist das neue Element sowohl das letzte als auch das erste
// thisElement zeigt auf den Vorgaenger des aktuellen Elements,
// beim ersten Element gibt es aber keinen Vorgaenger, deshalb NULL
// Dementsprechend ist auch das Vorgaengerelement NULL
if ( h->head == NULL ) {
   h->head = l;
   h->thisElement = NULL;
   h->lastElement = l;
   l->next = NULL;
   l->previous = NULL;
}
// 2. Fall: es gibt genau ein Element in der Liste
// in diesem Fall sind head und lastElement gleich diesem Element
// und thisElement ist NULL (siehe auch 1. Fall)
// das neue Element wird also vor dem Element in der Liste eingehaengt
// es ist damit der neue Kopf der Liste
// ausserdem ist der Nachfolger des neuen Elements das alte Element in
// der Liste
// Da das neue Element neuer Kopf wird, gibt es auch keinen Vorgaenger
// Das bisherige Kopfelement bekommt aber den neuen Kopf als Vorgaenger
// zugewiesen
else if ( h->thisElement == NULL ) {
   l->next = h->head;
   l->previous = NULL;
   h->head->previous = l;
   h->head = l;
}
// 3. Fall: es gibt mehr als ein Element in der Liste
// in diesem Fall wird der Nachfolger des neuen Elements das bisher
// aktuelle (thisElement zeigt auf dessen Vorgaenger!)
// der Nachfolger des Vorgaengers des aktuellen Elements muss jetzt auf
// das neu eingefuegte Element zeigen
// der Vorgaenger des neueinzufuegenden Elements wird das bisher
// aktuelle Element, der Vorgaenger des letzten Elements wird das
// neu einzufuegende
else {
   l->next = h->thisElement->next;
   l->previous = h->thisElement;
   if(h->thisElement->next)
      h->thisElement->next->previous = l;
   h->thisElement->next = l;
}
```

```
   // zu guter Letzt wird der Vorgaenger des aktuellen Elements
   // auf das neu eingefuegte Element gesetzt
   h->thisElement = l;

   // falls hinten an die Liste angehaengt wurde:
   if ( h->lastElement->next )
      h->lastElement = l;
}

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

// Funktion, um Elemente aus der Liste zu lesen
// Rueckgabewert: gefundenes Nutzdaten
// Parameter: Listhandler der zu bearbeitenden Liste
userdata_t *readElementList(list_handle_t *h) {
   // wenn der Listhandler nicht gueltig ist
   // oder die Liste leer ist
   // oder der Vorgaenger des aktuellen Elements keinen Nachfolger hat
   // gib NULL zurueck, da ungueltige Werte
   if ( !h || !(h->head) || !(h->thisElement->next) ) return NULL;

   // wenn es keinen Vorgaenger vom aktuellen Element gibt, existiert nur ein
   // Element, das Head, also gib dessen Daten zurueck
   if ( !(h->thisElement) ) return &(h->head->data);

   // in allen anderen Faellen
   // gib die Daten des aktuellen Elements zurueck
   return &(h->thisElement->next->data);
}

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

// Funktion zum Zuruecksetzen des Vorgaengers des aktuellen Elements
// kein Rueckgabewert
// Parameter: Listhandler der zu bearbeitenden Liste
void rewindList(list_handle_t *h) {
   // wenn der Handler gueltig ist, setze den Vorgaenger des aktuellen
   // Elements auf NULL
   if ( h ) h->thisElement = NULL;
}

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

// Funktion, um schrittweise durch die Liste zu gehen (vorwaerts)
// Rueckgabewert: Zeiger auf die Daten des aktuellen Listenelements
// Parameter: Listhandeler der zu bearbeitenden Liste
userdata_t *traverseListForward(list_handle_t *h) {
   // Zeiger fuer die Daten des Listenelements
   userdata_t *dPtr;
```

```
   // wenn der Listenhandler ungueltig ist
   // oder der Vorgaenger des aktuellen Elements gleich dem letzten
   // Element ist, ist die Liste ungueltig, Rueckgabewert NULL
   if ( (!h) || h->thisElement == h->lastElement ) {
      return NULL;
   }

   // wenn die Liste nur ein Element hat, ist der Rueckgabewert
   // der Zeiger auf die Daten des Heads
   // thisElement wird fuer den naechsten Schritt hochgesetzt
   if ( h->thisElement == NULL ) {
      h->thisElement = h->head;
      return &(h->head->data);
   }

   // in allen anderen Faellen ist der Rueckgabewert ein Zeiger
   // auf die Daten des letzten Elements
   dPtr = &(h->thisElement->next->data);

   // das aktuelle Element wird auf das naechste Element gesetzt
   // (fuer den naechsten Schritt)
   h->thisElement = h->thisElement->next;

   // Rueckgabe der Daten
   return dPtr;

}

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

// Funktion, um schrittweise durch die Liste zu gehen (rückwärts)
// Rueckgabewert: Zeiger auf die Daten des aktuellen Listenelements
// Parameter: Listhandler der zu bearbeitenden Liste
userdata_t *traverseListBackward(list_handle_t *h) {
   // Zeiger fuer die Daten des Listenelements
   userdata_t *dPtr;

   // wenn der Listenhandler ungueltig ist
   // oder der Vorgaenger des aktuellen Elements gleich dem letzten
   // Element ist, ist die Liste ungueltig, Rueckgabewert NULL
   if ( !h ) {
      return NULL;
   }

   // wenn die Liste nur ein Element hat, ist der Rueckgabewert
   // der Zeiger auf die Daten des Heads
   if ( h->thisElement == NULL ) {
      h->thisElement = h->head;
      return &(h->head->data);
   }
```

```
    // in allen anderen Faellen ist der Rueckgabewert ein Zeiger
    // auf die Daten des letzten Elements
    dPtr = &(h->thisElement->next->data);

    // das aktuelle Element wird auf das vorige Element gesetzt
    // (fuer den naechsten Schritt)
    // wenn es keinen Vorgaenger mehr gibt, ist der Anfang der
    // Liste erreicht, deshalb wird thisElement auf NULL gesetzt
    if (h->thisElement->previous)
      h->thisElement = h->thisElement->previous;
    else
      h->thisElement = NULL;

    // Rueckgabe der Daten
    return dPtr;

  }

  /*————————————————————————————————*/

  void deleteList(list_handle_t *h) {

    struct list_element_t *tmp;

    /*— Wir koennen nur etwas loeschen, wenn der Handle definiert
     ist und die Liste nicht leer ist. Es sind 3 F=E4lle zu unterscheiden:
     (1) Wenn h->thisElement == NULL, ist der Listenkopf zu loeschen.
      (2) Wenn h->thisElement != NULL und h->thisElement->next == NULL,
wurde
       die Liste bereits vollstaendig traversiert. Es gibt kein zu
       löschendesElement.
     (3) Wenn h->thisElement != NULL und h->thisElement->next != NULL, ist
       ein Element HINTER dem Listenkopf zu l=F6schen. Falls dies das
       letzte der Liste ist, muessen wir auch den lastElement-Zeiger
       anpassen. —*/
    if ( h // illegaler List Handle
       && h->head // Liste != leer
       && ( !(h->thisElement) // Listenkopf ist zu loeschen
          || (h->thisElement->next) // Die Liste ist NICHT komplett
                          // durchgelesen worden
       ) ) {

       if ( !(h->thisElement) ) {
         /*— Falls das erste Element zu loeschen ist —*/
         tmp = h->head;
         h->head = h->head->next; // Der neue Kopf  ist der Nachfolger
                          // des alten Kopfes.
         //wenn der Kopf geloescht wird, muss der Vorgaenger des
         //neuen Kopfes NULL sein
         h->head->previous = NULL;
         if ( !(h->head) ) h->lastElement = NULL;
```

```
      }
      else {
         /*— Falls ein Element HINTER dem Listenkopf zu loeschen ist:
             das zu loeschende Element ist durch
             h->thisElement->next identifiziert. Der Sonderfall
             h->thisElement->next == NULL wurde bereits ausgeschlossen. —*/
         tmp = h->thisElement->next;
         /*— falls wir das letzte Listenelement geloescht haben —*/
         /*previous braucht nicht geaendert werden, wenn das letzte
           Element geloescht wurde!*/
         if (tmp == h->lastElement) h->lastElement = h->thisElement;
         /*— hier wird das aktuelle Element aus der Liste "ausgekettet" —*/
         /*der Nachfolger des auszukettenden Elements wird der neue
           Nachfolger von thisElement, deshalb wird this Element dessen
           Vorgaenger*/
         if(h->thisElement->next->next)
            h->thisElement->next->next->previous = h->thisElement;
         h->thisElement->next = h->thisElement->next->next;
      }

      /*— Freigabe des zu loeschenden Listeneintrags: Der Speicherplatz
          "wird an das Betriebssystem zurueckgegeben" —*/
      free(tmp);

   }

}

/*————————————————————————————————————————-*/

// Funktion zum Ausgeben der Liste
// kein Rueckgabewert
// Parameter: Listhandler der zu bearbeitenden Liste
void printList(list_handle_t *h) {
   // Zeiger auf die Daten
   userdata_t *dPtr;

   // Liste zuruecksetzen
   rewindList(h);

   // Daten holen
   dPtr = traverseListForward(h);
   // wenn die Daten gueltig sind, ausgeben bis Ende
   while(dPtr != NULL) {
      printf("key = %d name = %s\n",
      dPtr->key, dPtr->name);
      dPtr = traverseListForward(h);
   }
}

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

```
/* Hauptprogramm: Beispiel zum Arbeiten mit Listen */
/*-------------------------------------------------------*/

int main() {
   // zwei Listhandler
   list_handle_t *list1Handle;
   list_handle_t *list2Handle;
   // Variable fuer Daten
   userdata_t d;
   // Zeiger auf Daten
   userdata_t *dPtr;

   // zwei Listen erzeugen
   list1Handle = createList();
   list2Handle = createList();

   // Daten erzeugen
   d.key = 10;
   // Kopieren von String in char-array:
   // der Ziel char-array muss lang genug sein,
   // um den Quellstring + Null-Character
   // abzuspeichern
   strcpy(d.name,"name1");
   // Daten an die Liste 1 anhaengen
   appendList(list1Handle,d);

   // noch mehr Daten erzeugen und anhaengen
   d.key = 5;
   strcpy(d.name,"name2");
   appendList(list1Handle,d);

   d.key = 7;
   strcpy(d.name,"name3");
   appendList(list1Handle,d);

   // Liste 1 ausgeben
   printf("1. Lesevorgang:\n");
   printList(list1Handle);

   // Liste 1 zuruecksetzen
   rewindList(list1Handle);

   // aktuelles Element ausgeben
   printf("lies aktuelles Element:\n");
   dPtr = traverseListForward(list1Handle);
   printf("key = %d name = %s\n",dPtr->key, dPtr->name);

   // Daten veraendern
   printf("Veraendere die Daten des aktuellen Elementes in der Liste\n");
   dPtr->key = 200;
   strcpy(dPtr->name,"name200");
```

```
// Liste zum zweiten Mal ausgeben
printf("2. Lesevorgang:\n");
printList(list1Handle);

// Liste zuruecksetzen
rewindList(list1Handle);
// zwei Elemente weitergehen
printf("gehe zum dritten Element:\n");
dPtr = traverseListForward(list1Handle);
dPtr = traverseListForward(list1Handle);
// Element lesen
printf("lies drittes Element:\n");
dPtr = readElementList(list1Handle);
printf("key = %d name = %s\n", dPtr->key, dPtr->name);

// neue Daten erzeugen und einfuegen
d.key = 99;
strcpy(d.name,"name99");
insertList(list1Handle,d);
printf("fuege neues Element an der aktuellen Stelle ein:\n");

// Liste zum dritten Mal ausgeben
printf("3. Lesevorgang:\n");
printList(list1Handle);

// Liste zuruecksetzen
rewindList(list1Handle);
printf("zuruecksetzen:\n");

// Daten erzeugen und einfuegen
d.key = 100;
strcpy(d.name,"name100");
insertList(list1Handle,d);
printf("fuege neues Element an der aktuellen Stelle ein:\n");

// Liste zum vierten Mal ausgeben
printf("4. Lesevorgang:\n");
printList(list1Handle);

//gehe zwei Schritte weiter zum dritten Element
rewindList(list1Handle);
printf("zuruecksetzen:\n");
dPtr = traverseListForward(list1Handle);
dPtr = traverseListForward(list1Handle);
printf("zum dritten Element gehen:\n");

// Daten erzeugen und einfuegen
d.key = 101;
strcpy(d.name,"name101");
insertList(list1Handle,d);
```

```
    printf("fuege neues Element an der aktuellen Stelle ein:\n");

    // Liste zum fuenften Mal ausgeben
    printf("5. Lesevorgang:\n");
    printList(list1Handle);

    //2. Element loeschen und ausgeben
    printf("loesche das 2. Element\n");
    rewindList(list1Handle);
    dPtr = traverseListForward(list1Handle);
    deleteList(list1Handle);
    printList(list1Handle);

    //ein Schritt zurueck
    dPtr = traverseListBackward(list1Handle);
    printf("ein Schritt zurueck gehen\n");

    // Element lesen
    printf("lies aktuelles Element:\n");
    dPtr = readElementList(list1Handle);
    printf("key = %d name = %s\n", dPtr->key, dPtr->name);

    //ein Schritt zurueck
    dPtr = traverseListBackward(list1Handle);
    printf("ein Schritt zurueck gehen\n");

    // Element lesen
    printf("lies aktuelles Element:\n");
    dPtr = readElementList(list1Handle);
    printf("key = %d name = %s\n", dPtr->key, dPtr->name);

}
```