Dr. Annette Bieniusa
Mathias Weber, M. Sc.
Peter Zeller, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lösungshinweise/-vorschläge zum Übungsblatt 13: Software-Entwicklung 1 (WS 2017/18)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 String-Suche (10 Punkte)

a) Schreiben Sie eine Methode `bool contains(char *search, int searchSize, char *input, int inputSize)`. Die Methode nimmt einen Such-Text `search` der Länge `searchSize` und einen Eingabe-Text `input` der Länge `inputSize`. Die Funktion soll `true` zurückgeben, wenn der gesuchte Text im Eingabe-Text vorkommt und ansonsten `false`.

Verwenden Sie keine Funktionen aus der C-Bibliothek für diese Aufgabe.

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

bool contains(char *search, int searchSize, char *input, int inputSize)
{
    for (int i=0; i<inputSize-searchSize; i++) {
        bool found = true;
        for (int j=0; j<searchSize; j++)
        {
            if (input[i+j] != search[j])
            {
                found = false;
                break;
            }
        }
        if (found)
        {
            return true;
        }
    }
    return false;

}
```

b) Schreiben Sie eine `main`-Funktion für ihr Programm, welches einen Such-Text als Programm-Parameter nimmt und die Standard-Eingabe nach Vorkommen des Such-Strings durchsucht. Die Zeilen der Eingabe, welche den Such-Text enthalten, sollen zusammen mit ihrer Zeilennummer ausgegeben werden. Dabei soll zuerst die Zeilennummer ausgegeben werden, dann ein Leerzeichen und und dann der Text der Zeile.

```
int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("1 program parameter expected but %d given.\n", argc);
        return 1;
    }
    char *searchString = argv[1];
    int searchStringLen = strlen(searchString);
    int bufferSize = 1024;
    char *buffer = malloc(bufferSize);
    int lineNr = 1;
    while (true)
    {
        char *s = fgets(buffer, bufferSize, stdin);
        if (s == NULL)
        {
            break;
        }
        if (contains(searchString, searchStringLen, s, strlen(s))) {
            printf("%d %s", lineNr, buffer);
        }
        lineNr++;
    }
    free(buffer);
}
```

Aufgabe 2 Sortierte binäre Bäume in C (14 Punkte)

In dieser Aufgabe sollen Sie ein Wörterbuch (engl.: dictionary) implementieren, welches intern einen sortierten binären Baum verwendet. Das Wörterbuch ist eine Map mit Strings als Schlüssel und als Wert. Wir verwenden die folgenden Typdefinitionen um das Wörterbuch selbst und die Knoten des Baums darzustellen:

```
typedef struct treenode treenode_t;
struct treenode
{
    char *key;
    char *value;
    treenode_t *left;
    treenode_t *right;
};

typedef struct dictionary dictionary_t;
struct dictionary
{
    treenode_t *root;
};
```

Ein Knoten speichert Schlüssel `key` und Wert `value` als Zeiger zu einem C-String ab. Außerdem hat jeder Knoten einen Zeiger auf seinen linken und rechten Kind-Knoten. Das Wörterbuch selbst enthält nur einen Zeiger auf den Wurzelknoten.

Die Sortierung des Baums ergibt sich über das Vergleichen der Schlüssel mit `strcmp`. Für einen Knoten im Baum mit Schlüssel `k` gilt, dass alle Schlüssel `kl` im linken Teilbaum kleiner sind (`strcmp(kl,k) < 0`) und alle Schlüssel `kr` im rechten Teilbaum größer (`strcmp(kr, k) > 0`). Der Baum enthält also auch keinen Schlüssel mehrmals.

Laden Sie sich zur Bearbeitung der Aufgabe die Vorlage `dict.c` herunter. Diese enthält die Typen, Vorlagen für die zu implementierenden Funktionen und eine `main`-Funktion mit der sich das Wörterbuch testen lässt. Dazu können Sie dem Programm die folgenden Befehle geben, um mit einem Wörterbuch zu arbeiten:

```
put k v
get k
del k
```

Die ersten 3 Zeichen geben den Befehl an. Danach kommt getrennt durch ein Leerzeichen der Schlüssel und für `put` nach einem weiteren Leerzeichen der Wert. Bei einem Aufruf von `get` wird der aktuelle Wert im Wörterbuch ausgegeben.

a) Schreiben Sie eine Funktion `dictionary_t *dict_new()`, welche ein neues leeres Wörterbuch erstellt und einen Zeiger auf die Wörterbuch-Struktur zurückgibt.

```
dictionary_t *dict_new()
{
    dictionary_t *result = malloc(sizeof(dictionary));
    if (result == NULL)
    {
        printf("not enough memory for dict_new");
        exit(1);
    }
    result->root = NULL;
    return result;
}
```

b) Schreiben Sie eine Funktion **`void`** `dict_put(dictionary_t *dict,` **`char`** `*key,` **`char`** `*value)`, welche einen Eintrag mit Schlüssel `key` und Wert `value` im Wörterbuch speichert. Falls bereits ein Eintrag mit dem Schlüssel existiert, wird der Wert ersetzt.

Die C-Strings `key` und `value` sind nur für den Aufruf der Funktion gültig. Erstellen Sie deshalb wenn nötig Kopien der Strings.

```
void treenode_insert(treenode_t **nodePtr, char *key, char *value)
{
    treenode_t *node = *nodePtr;
    if (node == NULL)
    {
        treenode_t *new = malloc(sizeof(treenode_t));
        if (new == NULL)
        {
            printf("not enough memory for treenode_insert\n");
            exit(1);
        }
        new->key = copy_string(key);
        new->value = copy_string(value);
        new->left = NULL;
        new->right = NULL;
        *nodePtr = new;
        return;
    }
    int cmp = strcmp(key, node->key);
    if (cmp < 0)
    {
        treenode_insert(&node->left, key, value);
    }
    else if (cmp > 0)
    {
        treenode_insert(&node->right, key, value);
    }
    else
    {
        // Vorhandenen Wert ersetzen
        char *new = realloc(node->value, strlen(value) + 1);
        if (new == NULL)
```

```
        {
            printf("not enough memory for realloc\n");
            exit(1);
        }
        node->value = new;
        strcpy(node->value, value);
    }
}

void dict_put(dictionary_t *dict, char *key, char *value)
{
    treenode_insert(&dict->root, key, value);
}
```

c) Schreiben Sie eine Funktion `char *dict_get(dictionary_t *dict, char *key)`, welche den Wert zum gegeben Schlüssel `key` zurückgibt. Falls `key` nicht im Wörterbuch gespeichert ist, soll `NULL` zurückgegeben werden.

```
char *treenode_get(treenode_t *node, char *key)
{
    if (node == NULL)
    {
        // nicht gefunden
        return NULL;
    }
    int cmp = strcmp(key, node->key);
    if (cmp < 0)
    {
        return treenode_get(node->left, key);
    }
    else if (cmp > 0)
    {
        return treenode_get(node->right, key);
    }
    else
    {
        // Eintrag gefunden
        return node->value;
    }
}

char *dict_get(dictionary_t *dict, char *key)
{
    return treenode_get(dict->root, key);
}
```

d) Schreiben Sie eine Funktion `void dict_free(dictionary_t *dict)`, welcher sämtlichen vom Wörterbuch benutzten Speicher freigibt.

```
void treenode_free(treenode_t *node)
{
    if (node == NULL)
    {
        return;
    }
    treenode_free(node->left);
    treenode_free(node->right);
    free(node->key);
    free(node->value);
    free(node);
}
```

```c
void dict_free(dictionary_t *dict)
{
    treenode_free(dict->root);
    free(dict);
}
```

e) *Freiwillige Zusatzaufgabe*: Schreiben Sie eine Funktion `void dict_delete(dictionary_t *dict, char * key)`, welche den Eintrag mit Schlüssel `key` aus dem Wörterbuch löscht.

```c
treenode_t **treenode_max(treenode_t **nodePtr)
{
    treenode_t *node = *nodePtr;
    if (node->right == NULL)
    {
        return nodePtr;
    }
    return treenode_max(&node->right);
}

void treenode_delete(treenode_t **nodePtr, char *key)
{
    treenode_t *node = *nodePtr;
    if (node == NULL)
    {
        // nicht gefunden
        return;
    }
    int cmp = strcmp(key, node->key);
    if (cmp < 0)
    {
        // suche im linken Teilbaum
        return treenode_delete(&node->left, key);
    }
    else if (cmp > 0)
    {
        // suche im rechten Teilbaum
        return treenode_delete(&node->right, key);
    }
    else
    {
        // zu loeschendes Element gefunden
        if (node->left == NULL)
        {
            // wenn der linke Teilbaum leer ist, dann nur den rechten nehmen
            *nodePtr = node->right;
            node->right = NULL;
            treenode_free(node);
        }
        else if (node->right == NULL)
        {
            // analog zum Fall node->left == NULL
            *nodePtr = node->left;
            node->left = NULL;
            treenode_free(node);
        }
        else
        {
            // wenn der zu löschende Knoten zwei Kinder hat, den Knoten mit dem
            // groessten Element aus dem linken Teilbaum suchen:
            treenode_t **maxNodeLeftPtr = treenode_max(&node->left);
            treenode_t *maxNodeLeft = *maxNodeLeftPtr;
            // Die Markierung von diesem Knoten nehmen wir fuer den aktuellen:
            free(node->key);
```

```
            free(node->value);
            node->key = maxNodeLeft->key;
            node->value = maxNodeLeft->value;
            // Der maximale Knoten im linken Teilbaum kann höchstens einen linken
            // Teilbaum haben. Daher kann er gelöscht und durch seinen Linken
            // Teilbaum ersetzt werden.
            *maxNodeLeftPtr = maxNodeLeft->left;
            free(maxNodeLeft);
        }
    }
}

void dict_delete(dictionary_t *dict, char *key)
{
    treenode_delete(&dict->root, key);
}
```

Aufgabe 3 Verkette Liste (15 Punkte)

Laden Sie sich die Datei `linkedlist.c` herunter, welche die Implementierung von einfach verketteten Listen aus der Vorlesung enthält. In dieser Aufgabe sollen Sie diese Liste um weitere Funktionen erweitern.

a) Schreiben Sie eine Funktion **`bool`** `list_is_sorted(linked_list_t *ll)`, welche prüft ob die in der Liste `ll` gespeicherten Werte aufsteigend sortiert sind.

```
bool list_is_sorted(linked_list_t *ll)
{
  node_t *a = ll->first;
  if (a == NULL)
  {
    return true;
  }
  node_t *b = a->next;
  while (b)
  {
    if (a->value > b->value)
    {
      return false;
    }
    a = b;
    b = b->next;
  }
  return true;
}
```

b) Schreiben Sie eine Funktion **`bool`** `list_has_duplicates(linked_list_t *ll)`, welche prüft, ob in der Liste `ll` Werte existieren, die mehrmals vorkommen.

```
bool contains(node_t *node, int value)
{
  while (node)
  {
    if (node->value == value)
    {
      return true;
    }
    node = node->next;
  }
  return false;
}
```

```c
bool list_has_duplicates(linked_list_t *ll)
{
  for (node_t *node = ll->first; node; node = node->next)
  {
    if (contains(node->next, node->value))
    {
      return true;
    }
  }
  return false;
}
```

c) Schreiben Sie eine Funktion `void list_add_before(linked_list_t *ll, int x, int y)`, welche einen neuen Eintrag mit Wert `x` direkt vor dem ersten Eintrag mit Wert `y` in die Liste einfügt. Wenn `y` nicht in der Liste enthalten ist soll `x` am Ende der Liste eingefügt werden.

```c
void list_add_before(linked_list_t *ll, int x, int y)
{
  node_t **nodePtr = &ll->first;
  while (*nodePtr && (*nodePtr)->value != y)
  {
    nodePtr = &(*nodePtr)->next;
  }
  node_t *new_node = malloc(sizeof(node_t));
  if (!new_node)
  {
    printf("Couldn't allocate new node");
    exit(-1);
  }
  // Initalisiere den neuen Knoten
  new_node->value = x;
  new_node->next = *nodePtr;
  *nodePtr = new_node;
}
```

d) Schreiben Sie eine Funktion `int list_remove(linked_list_t *ll, int value)`, welche alle Vorkommen von `value` aus der Liste `ll` entfernt und zurückgibt, wie viele Elemente entfernt wurden.

Die Liste soll von der Funktion nur einmal durchlaufen werden.

```c
int list_remove(linked_list_t *ll, int value)
{
  int count = 0;
  node_t **nodePtr = &ll->first;
  while (*nodePtr)
  {
    node_t *node = *nodePtr;
    if (node->value == value)
    {
      *nodePtr = node->next;
      free(node);
      count++;
    }
    else
    {
      nodePtr = &node->next;
    }
  }
  return count;
}
```

Aufgabe 4 Mergesort (9 Punkte)

```c
/** Zusammenfuehren von zwei sortierten Listen */
node_t * merge(node_t * a, node_t * b)
{
  if (a == NULL)
  {
    return b;
  }
  else if (b == NULL)
  {
    return a;
  }
  if (a->value < b->value)
  {
    a->next = merge(a->next, b);
    return a;
  }
  else
  {
    b->next = merge(a, b->next);
    return b;
  }
}

/**
  * Sortieren mit Angabe der Laenge, damit diese nicht immer neu berechnet
  * werden muss.
  */
node_t * mergesort(node_t *n, int length)
{
  if (n == NULL || n->next == NULL)
  {
    return n;
  }
  // Liste in 2 Teile aufteilen:
  node_t *m = n;
  int mid = length / 2;
  for (int i = 0; i < mid - 1; i++)
  {
    m = m->next;
  }
  node_t * partOne = n;
  node_t * partTwo = m->next;
  // Liste trennen:
  m->next = NULL;

  // Teillisten sortieren:
  partOne = mergesort(partOne, mid);
  partTwo = mergesort(partTwo, length - mid);

  // sortierte Teillisten zusammenfuehren:
  return merge(partOne, partTwo);
}


void sort(linked_list_t *ll)
{
  ll->first = mergesort(ll->first, list_size(ll));
}
```