Aufgabe 1: Strings in C

Ihre Aufgabe ist es verschiedene Funktionen zur Behandlung von Strings in C zu implementieren und diese auch hinreichend zu testen. Dazu steht ihnen ein Code Template zur Verfügung.

Es folgt eine Beschreibung der zu implementierenden Funktionen.

normalisiere

Implementiere

```
char* normalisiere(char* s)
```

Eingabe ist ein String welcher nur aus Klein-/Grossbuchstaben und Leerzeichen besteht.

Rückgabe ist ein (neuer) String in welchem

- alle Leerzeichen eliminiert wurden
- alle Grossbuchstaben in Kleinbuchstaben umgewandelt wurden.

Z.B.

```
normalisiere("Ha Ll o o") ==> "halloo"
```

Hinweise.

- Speicher für den Rückgabe String muss dynamisch allokiert werden
- Berechne zuerst die Größe des Strings
- Allokiere entsprechend Speicher für den neuen String
- Übertrage alle Zeichen (ausser Leerzeichen) in den neuen String, wobei Grossbuchstaben in Kleinbuchstaben umgewandelt werden

copyStr

Implementiere

```
char* copyStr(char* s)
```

Eingabe ist ein beliebiger String.

Rückgabe ist eine Kopie des Strings.

Hinweise:

- Speicher für den Rückgabe String muss dynamisch allokiert werden
- Verwende `copy`

putBack

Implementiere

```
char* putBack(char c, char* s)
```

Eingabe ist ein beliebiger String.

Rückgabe ist ein (neuer) String in welchem Zeichen c am Schluss angehängt ist

Z.B.

```
putBack('!', "abcd")  ==> "abcd!"
```

Hinweise.

- Speicher für den Rückgabe String muss dynamisch allokiert werden
- Verwende wenn möglich `putFront` und `reverse` unter Aussnutzung folgender Invariante

```
reverse(putFront(c,reverse(s))) = putBack(c, s)
```

rev

Implementiere

```
char* rev(char* s)
```

Eingabe ist ein beliebiger String.

Rückgabe ist ein neuer String welcher die Umkehrung des Eingabestrings ist.

Z.B.

```
rev("abcd!") ==> "!dcba"
```

Hinweise.

- Speicher für den Rückgabe String muss dynamisch allokiert werden
- Die Implementierung soll rekursiv sein und die Hilfsroutine putBack verwenden

replace

Implementiere

```
void replace(char* s, struct OldNew* m, int n)
```

wobei

```
struct OldNew {
  char old;
  char new;
};
```

Eingaben sind:

1. Ein beliebiger String s

2. Eine Referenz auf ein Array vom Typ OldNew

3. Die Größe des Arrays

Rückgabe ist ein (neuer) String in welchem alle Zeichen entsprechend dem “OldNew” Array ersetzt wurden.

Z.B. Falls

```
struct OldNew m[] = { {'B', 'b'}, {'s', '!'}};
```

dann

```
replace("Aa dss fBB", m, 2) ==> "Aa d!! fbb"
```

Hinweise.

- Speicher für den Rückgabe String muss dynamisch allokiert werden
- Wir gehen davon auss, die Domäne der “old” Zeichen verschieden ist von der Domäne der “new” Zeichen

D.h. Fälle wie

```
struct OldNew m2[] = { {'B', 'b'}, {'b', '!'}};
```

sind ausgeschlossen.

Allgemeine Hinweise

Achte auf die Speicherallokation und Speicherfreigabe.

Bestehende Funktionalitäten aus der C Standard Bibliothek dürfen NICHT verwendet werden.

Verwende und erweitere die zur Verfügung gestellten Testroutinen.

Code Template für Aufgabe 1

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


enum Bool;
struct OldNew;

int length(char *s);
char* normalisiere(char* s);
void copy(char* s, int n, char* t);
char* copyStr(char* s);
char* putFront(char c, char* s);
char* reverse(char* s);
char* putBack(char c, char* s);
char* rev(char* s);
void replace(char* s, struct OldNew* m, int n);
char* show(enum Bool b);
enum Bool strCmp(char* s1, char* s2);



// Anzahl aller Zeicher (ohne Null-terminator).
int length(char *s) {
  int n = 0;
  while(*s != '\0') {
    n++;
    s++;
  }

  return n;
```

```
}


// Normalisiere C String.
// 1. Eliminiere Leerzeichen.
// 2. Alle Grossbuchstaben werden in Kleinbuchstaben umgewandelt.
// 3. Kleinbuchstaben bleiben unveraendert.
// Annahme: C String besteht nur aus Klein-/Grossbuchstaben und
         Leerzeichen.
char* normalisiere(char* s) {
  // TODO
}


// Kopiere n Zeichen von s nach t.
// Annahme: n ist > 0
void copy(char* s, int n, char* t) {
  int i = 0;
  while(i < n) {
    t[i] = s[i];
    i++;
  }
}


// Baue neuen String welcher eine Kopie des Eingabestrings ist.
char* copyStr(char* s) {
  // TODO
}

// Baue neuen String welcher mit Zeichen c startet gefolgt von allen
         Zeichen in s.
char* putFront(char c, char* s) {
  const int n =  length(s);
  char* r = (char*)malloc(sizeof(char) * (n+2));
  copy(s, n+1, r+1);
  r[0] = c;
  return r;
}


// Umkehrung eines Strings.
char* reverse(char* s) {
  const int n = length(s);
  char* t = (char*)malloc(n + 1);
  int i;

  for(i = 0; i < n; i++) {
    t[i] = s[n-1-i];
```

```
  }
  t[n] = '\0';

  return t;
}

// Baue neuen String welcher aus allen Zeichen in s besteht gefolgt von
        Zeichen c.
char* putBack(char c, char* s) {
  // TODO
}

// Baue einen neuen String welcher die Umkehrung des Eingabestrings ist.
// Hinweis: Die Implementierung soll rekursiv sein und die Hilfsroutine
        putBack verwenden.
char* rev(char* s) {
  // TODO
}


struct OldNew {
  char old;
  char new;
};


// Ersetze in einem String jedes Zeichen 'old' mit dem Zeichen 'new'.
// Die Zeichen 'old' und 'new' sind definiert in einem Array vom Typ struct
        OldNew.
void replace(char* s, struct OldNew* m, int n) {
  // TODO
}

enum Bool {
  True = 1,
  False = 0
};

char* show(enum Bool b) {
  if(b == True) {
    return copyStr("True");
  } else {
    return copyStr("False");
  }
}


// Teste ob zwei Strings identisch sind.
enum Bool strCmp(char* s1, char* s2) {
```

```
  // TODO
}


void userTests() {
  printf("\n\n *** User Tests *** \n\n");

  char s1[] = "Ha Ll o o ";

  printf("\n1. %s", s1);

  printf("\n2. %s", normalisiere(s1));

  char* s2 = (char*)malloc(length("Hello")+1);

  char* s3 = copyStr("Hello");

  printf("\n3. %s", s3);

  char s4[] = "abcd";

  char* s5 = putBack('!',s4);

  printf("\n4. %s", s5);

  char* s6 = rev(s5);

  printf("\n5. %s", s6);

  char s7[] = "Aa dss fBB";

  printf("\n6. %s", s7);

  struct OldNew m[] = { {'B', 'b'}, {'s', '!'}};

  replace(s7, m, 2);

  printf("\n7. %s", s7);

  char s8[] = "HiHi";

  char* s9 = copyStr(s8);

  enum Bool b1 = strCmp(s8,s9);

  char* s10 = show(b1);

  printf("\n8. %s", s10);

  char s11[] = "HiHo";
```

```
  enum Bool b2 = strCmp(s9, s11);

  char* s12 = show(b2);

  printf("\n8. %s", s12);

  free(s2);
  free(s3);
  free(s5);
  free(s6);
  free(s9);
  free(s10);
  free(s12);
}

struct TestCase_ {
  char* input;
  char* expected;
};

typedef struct TestCase_ TC;

void runTests(TC* tc, int n, char* sut(char*)) {
  int i;

  for(i=0; i<n; i++) {
    char* result = sut(tc[i].input);
    if(True == strCmp(tc[i].expected, result)) {
      printf("\n Okay Test (%s,%s) => %s", tc[i].input,tc[i].expected,
         result);
    } else {
      printf("\n Fail Test (%s,%s) => %s", tc[i].input,tc[i].expected,
         result);
    }
    free(result);
  }

}


void unitTests() {
  printf("\n\n *** Unit Tests *** \n\n");

  TC normTests[] = {
    {"hElLo", "hello"},
    {"hEl Lo", "hello"},
    {"h  El Lo", "hello"},
  };
```

```
  runTests(normTests, 3, normalisiere);

}

char* rndString() {
  int i;
  int n = (rand() % 10) + 1;
  char* s = (char*)malloc(n+1);

  for(i=0; i<n; i++) {
    int lowHigh = (rand() % 2) ? 'a' : 'A';
    int c = rand() % 26;
    s[i] = (char)(c + lowHigh);
  }
  s[n] = '\n';

  return s;
}

void invariantenTests() {
  printf("\n\n *** Invarianten Tests *** \n\n");

  int i;
  for(i=0; i<20; i++) {
    char* s = rndString();
    char* r = reverse(s);
    char* n1 = normalisiere(s);
    char* n2 = normalisiere(r);
    char* n3 = reverse(n2);

    if(True == strCmp(n1,n3)) {
      printf("Okay %s", s);
    } else {
      printf("Fail %s", s);
    }

    free(s);
    free(r);
    free(n1);
    free(n2);
    free(n3);
  }



}
```

```
int main() {
  userTests();

  unitTests();

  invariantenTests();
}
```