Aufgabe 1. Schreibe eine Funktion, die mithilfe von `qsort` ein Array von Strings lexikographisch (wie im Telefonbuch) sortiert. Das heißt, dass erst nach der ersten Stelle sortiert wird, dann nach der Zweiten usw. In der `<string.h>` liegt eine Vergleichsfunktion für diese Situation vor: `strcmp`. Als Beispiel hier eine lexikographisch sortierte Liste:

```
1
10
133
2
2344
Hallo
Thor
Tor
```

Aufgabe 2. Lese noch einmal im Skript die Sektion über doppelt verkettete Listen. Implementiere nun Doppelt-verkettete Listen, die `double`-Variablen speichern, d.h. implementiere die `.c`-Datei zu folgender Headerdatei `list.h`:

```
#ifndef LIST_H
#define LIST_H

typedef struct NODE {
  struct NODE  *next;
  struct NODE  *prev;
  double        data;
} NODE;

typedef struct {
  NODE *first;
  NODE *last;
} LIST;

LIST *list_create();        /* Liste erstellen */
void  list_free(LIST *L);   /* Speicher freigeben */

/* Fuege hinter cursor einen neuen Knoten in die Liste
   ein. Falls cursor gleich NULL ist, fuege am Anfang
```

```
20    der Liste ein. Gibt einen Pointer auf  das neu
21    eingefuegte Element zurueck, oder NULL im
22    Fehlerfall. */
23 NODE *list_insert(LIST *L, NODE *cursor, double data);
24
25 /* Loesche einen Knoten aus der Liste. */
26 void list_delete(LIST *L, NODE *del);
27
28 #endif
```

Aufgabe 3. Lese noch einmal im Skript die Sektion über Hashtabellen. Implementiere nun Hashtabellen, die `double`-Variablen speichern, d.h. implementiere die `.c`-Datei zu der unten gegebenen Headerdatei `hash.h`.

Tip: Hierfür benötigst du verkettete Listen, die beliebige Daten (heißt `void *`) speichern können. Du kannst die Lösung von der Listen-Aufgabe entsprechend anpassen.

```
1  #ifndef HASHTABLE_H
2  #define HASHTABLE_H
3
4  #include "list.h" /* Wir benutzen die verketteten
5                       Listen von weiter oben. */
6
7  struct HASH_TABLE;  /* Ist weiter unten definiert. */
8
9  /* Ein Knoten der Hashtabelle. */
10 typedef struct {
11   int key;     /* Der Schluessel */
12   double data; /* Die Daten */
13 } HASH_NODE;
14
15 /* Die vom User definierte Hashfunktion, neben einem
16  * Schluessel ist noch die Hashtabelle selbst
17  * Teil des Input. */
18 typedef int (*hashfunc) (int key,
19                          struct HASH_TABLE * H);
20
21 typedef struct HASH_TABLE
```

```
22    /* Das Array von Listen H[i] */{
23    LIST     **buckets;
24    /* Die Laenge des Arrays */
25    int        num_buckets;
26    /* Die Hashfunktion f */
27    hashfunc   hashfunktion;
28  } HASH_TABLE;
29
30  /* Erstelle eine Hashtabelle mit vorgegebener Anzahl
31   * an Buckets und Hashfunktion. */
32  HASH_TABLE * hashtable_create (int num_buckets,
33                                 hashfunc f);
34
35  /* Gebe den Speicher einer Hashtabelle wieder frei. */
36  void         hashtable_free (HASH_TABLE *H);
37
38  /* Erstelle einen neuen Knoten und fuege ihn in eine
39   * Hashtabelle ein. Gibt einen Pointer auf den neu
40   * eingefuegten Knoten zurueck,
41   * oder NULL bei einem Fehler. */
42  HASH_NODE * hashtable_insert (HASH_TABLE *H,
43                                int Schluessel,
44                                double data);
45
46  /* Suche den Knoten mit vorgegebenem Schluessel in
47   * einer Hashtabelle. Gibt einen Pointer auf den
48   * Knoten zurueck, wenn dieser in der Tabelle
49   * gespeichert war, NULL sonst. */
50  HASH_NODE * hashtable_get (HASH_TABLE *H,
51                             int Schluessel);
52
53  /* Loesche den Knoten mit vorgegebenem Schluessel
54   * aus der Tabelle. */
55  void hashtable_delete (HASH_TABLE *H, int Schluessel);
56
57  #endif
```