Aufgabe 1. Implementiere doppelt verkettete Listen, die beliebige Daten speichern können (als `void *`). Die Liste soll zusätzlich einen Stack enthalten, auf dem gelöschte Elemente abgelegt werden, damit man sie wiederherstellen kann.

```
1  /* Leere Liste erstellen */
2  LIST *list_create();
3
4  /* Element hinter E einfügen, NULL heißt am Anfang */
5  LISTNODE *list_insert(LIST *L, LISTNODE *E, void *p);
6  /* Element hinter E einfügen, dessen datenpointer
7     auf n Zellen allokierten Speicher zeigt. Sollte bei
8     Speichermangel einen Nullpointer liefern. */
9  LISTNODE *list_insert_alloc(LIST *L, LISTNODE *E,
10         unsigned int n);
11
12 /* Element am Anfang bzw. Ende einfügen */
13 LISTNODE *list_unshift(LIST *L, void *p);
14 LISTNODE *list_push(LIST *L, void *p);
15
16 /* Element am Anfang bzw. Ende entfernen und
17    die Daten zurück geben */
18 void *list_shift(LIST *L);
19 void *list_pop(LIST *L);
20
21 /* eine Element aus der Liste entfernen */
22 void list_delete(LIST *L, LISTNODE *E);
23
24 /* zwei Listen zusammenfügen */
25 LIST *list_merge(LIST *L, LIST *M);
26
27 /* Liste inklusive allen Elementen frei geben */
28 void list_free(LIST *L);
29
30 /* Das eben gelöschte Element wiederherstellen */
31 void list_undelete(LIST *L);
32 /* Alle gelöschten Elemente wiederherstellen */
33 void list_dance(LIST *L);
```

Aufgabe 2. Erstelle eine Liste, füge einige Strings in sie ein und sortiere die Liste lexikographisch.

Aufgabe 3. a) Implementiere die Addition, Multiplikation, Potenzen und Division komplexer Zahlen. Verwende dazu folgende Header-Datei:

```
1 typedef struct _COMPLEX {
2         double real;
3         double imag;
4 } COMPLEX;
5
6 COMPLEX cplx_add(COMPLEX a, COMPLEX b);
7 COMPLEX cplx_mult(COMPLEX a, COMPLEX b);
8 COMPLEX cplx_pot(COMPLEX a, unsigned long n);
9 COMPLEX cplx_div(COMPLEX a, COMPLEX b);
```

b) Implementiere die Addition, Multiplikation und Division sowie das Kürzen rationaler Zahlen. Schreibe dazu erst die Header-Datei.