Beispiel A

Sort List

Zu Ihrer Angabe erhalten Sie drei Dateien:
list_maxsort_inc.h: stellt verwendete Strukturen und Deklarationen zur Verfügung.
list_maxsort_iue.c: hält bereits entwickelte Funktionen bereit.
list_maxsort.c: hier kommt Ihr Code hinein. Aus main() werden Ihre Funktionen und auch Funktionen aus list_maxsort_iue.c aufgerufen.
Erstellen Sie ein Projekt bzw. ein ausführbares Programm mit beiden .c Dateien. Die .h Datei wird in beiden .c Dateien inkludiert.

Die Datenstruktur list_t stellt die Elemente einer doppelt verketteten Liste dar. Gespeichert werden Worte im Datentyp data_t.
In der Datei list_maxsort_iue.c stellen wir Ihnen einige Funktionen zu Initialisierung bereit, welche dann auch in list_maxsort.c aufgerufen werden.

Mit der Funktion list_push(list_t **plist, data_t *pdata) wird ein neuer Datensatz *pdata vorne an die bestehende Liste *plist angehängt.
list_print gibt die Liste aus und list_free gibt sie wieder frei.
init_data legt einen Testdatensatz zum Eintragen in die Liste an und list_free gibt diesen wieder frei.

Implementieren Sie die Funktion

```
void list_sort(list_t **plist)
```

welche die übergebene Liste sortiert. *plist zeigt auf den Anfang der bereits bestehenden Liste.
Sortieren Sie diese alphabetisch nach den enthaltenen Wörtern und geben Sie die neue Liste wieder in *plist zurück.

Hierbei soll ausschließlich mit den schon bestehenden Listenelementen effizient gearbeitet werden, da jede Anforderung und Freigabe von Speicher Laufzeit kostet.
Das originale Bubblesort Verfahren kommt aufgrund der benötigten, bei eine Liste nicht vorhandenen, Zugriffe über einen Index (z.B. feld[i]) nicht in Frage.
Das Iterieren nach vorne und hinten und Entfernen von Elementen aus der Liste ist allerdings sehr effizient.

Das Sortierverfahren soll folgendermaßen aussehen:
Starten Sie mit der unsortierten Originalliste und einer zweiten leeren Liste.
* Suchen Sie in der unsortierten Liste das größte Element.
* Enfernen Sie dieses Element aus der Verkettung in der Liste.
* Hängen Sie das Element in der zweiten Liste am Anfang ein.
Wiederholen Sie diese Vorgänge, bis die unsortierte Liste leer ist. Danach sollten Sie die zweite Liste, ganz ohne Malloc nur mit den Elementen der ersten Liste, sortiert vorliegen haben.

Achten Sie bei der Implementierung auf die korrekte Behandlung von Sonderfällen, z.B. wenn das größte Element ganz vorne oder ganz hinten in der Liste gefunden wird.
Testen Sie Ihre Funktion ausführlich. Achten Sie dabei darauf, dass Fehler korrekt behandelt werden, keine Daten verloren gehen oder überschrieben werden und geben Sie alle angeforderten Speicherbereiche wieder ordnungsgemäß frei.

- list_maxsort_inc.h
- list_maxsort_iue.c
- list_maxsort.c

Verzeichnis herunterladen

Beispiel B

Ring Read

Zu Ihrer Angabe erhalten Sie drei Dateien.
ring_read_inc.h: stellt verwendete Strukturen und Deklarationen zur Verfügung.
ring_read_iue.c: hält bereits entwickelte Funktionen bereit.
ring_read.c hier kommt Ihr Code hinein. Aus main() werden Ihre Funktionen und auch Funktionen aus xxx_iue.c aufgerufen.
Erstellen Sie ein Projekt bzw. ein ausführbares Programm mit beiden .c Dateien. Die .h Datei wird in beiden .c Dateien inkludiert.

Der Zirkular- oder Ringbuffer stellt einen Speicherplatz für eine gewisse Anzahl von Datensätzen bereit (siehe Vorlesung 2.1 - Stapel und Schlange der Jupyter Notebooks zu Programmieren 2).
Mit einer Schreibefunktion wird, solange noch Platz ist, von vorne nach hinten in den Ring geschrieben.
Kommt die Schreibefunktion am Ende an, so wird vom Anfang des Feldes weitergeschrieben.
Mit einer Lesefunktion wird jeweils ein Element vom Anfang des Ringes abgenommen und der Anfang wandert um eins weiter. Es wird somit vorne wieder ein Element frei, welches wiederum von der Schreibfunktion verwendet werden darf.

Mit der Struktur ring_t wird ein einfacher Ringbuffer realisiert.

```
typedef struct
{
   char word[255];
} data_t;

typedef struct
{
   data_t *buffer;
   long size, fill, rd, wr;
} ring_t;
```

In buffer wird ein Feld für einzelne Wörter der Struktur data_t angelegt,
size gibt die Größe des angelegten Feldes an, fill gibt die aktuelle Anzahl der Einträge im Buffer an,
rd gibt den Index an, von dem als nächstes gelesen werden soll, und
wr gibt den Index an, an dem als nächstes geschrieben wird.

Beim Schreiben wird, solange noch Platz ist, an Stelle wr in buffer geschrieben und danach wr und fill um eins erhöht.
Beim nächsten Schreiben wird somit auf die nächste Stelle geschrieben.
Zeigt wr über das Ende des Feldes hinaus, so wird wr wieder auf 0 gesetzt, um wieder am Anfang weiterzuschreiben.

Beim Lesen wird, soferne Elemente im Buffer stehen, die Stelle rd aus buffer ausgelesen und danach rd ebenfalls um eins weitergerückt. Hingegen fill (die Anzahl der Elemente im Buffer) muss um eins erniedrigt werden.
Ebenfalls wird wieder beim Anfang zu lesen begonnen, falls rd über das Ende des Buffers hinauszeigt.

ring_init reserviert in ring den Buffer buffer mit sz Elementen und setzt rd, wr und fill für einen leeren Buffer,
ring_print gibt den Buffer aus, ring_free gibt ihn frei.
Init_data legt ein Feld von Testdaten in data an und free_data gibt diese Daten dann auch wieder frei.
Diese Daten werden in weiterer Folge in den Buffer geschrieben und auch wieder gelesen.

Ist beim Schreiben kein Platz im Buffer, so wird -1, ansonsten 0 zurückgegeben.
Sind beim Lesen keine Daten vorhanden, so wird -1, ansonsten 0 zurückgegeben.

Implementieren Sie die Funktion

```
long ring_read(ring_t *ring, data_t *data)
```

die ein Element aus dem Ringbuffer liest und in *data abspeichert (kopiert).
Wird versucht aus einem leeren Buffer zu lesen, so wird -1, ansonsten 0 zurückgegeben.
Setzen Sie rd und fill richtig, überschreiben Sie keine Daten, lassen Sie keine Daten verlorengehen.

Testen Sie Ihre Funktion ausführlich. Achten Sie dabei darauf, dass Fehler korrekt behandelt werden, keine Daten verloren gehen oder überschrieben werden und geben Sie alle angeforderten Speicherbereiche wieder ordnungsgemäß frei.

Beispiel C

Neuaufbau (rehashing) einer Hashtabelle

Zu Ihrer Angabe erhalten Sie drei Dateien.
rehash.h: stellt verwendete Strukturen und Deklarationen zur Verfügung.
rehash_iue.c: hält bereits entwickelte Funktionen bereit.
rehash.c hier kommt Ihr Code hinein. Aus main() werden Ihre Funktionen und auch Funktionen aus xxx_iue.c aufgerufen.
Erstellen Sie ein Projekt bzw. ein ausführbares Programm mit beiden .c Dateien. Die .h Datei wird in beiden .c Dateien inkludiert.

In main() wird eine Hashtabelle von Worten aus data_t aufgebaut. Hierbei wird mitgezählt, wie oft die Worte in einem Text vorkommen.
Dazu stellen wir Ihnen die Funktionen hash_func, hash_init, hash_find, hash_insert, hash_print und hash_free zur Verfügung.

Die Hashtabelle wird mit der Datenstruktur hash_t abgebildet. Die Größe des Arrays ist hashsize.

```
struct collision_list_s
{
   data_t data;
   struct collision_list_s *next;
};
typedef struct collision_list_s collision_list_t;

typedef struct
{
   collision_list_t **hasharray;
   long hashsize;
} hash_t;
```

Neue Elemente werden (nachdem der Hash mittels der Hashfunktion ermittelt wurde) an der entsprechenden Stelle in hasharray eingetragen.
Dort wird zur Kollisionsbehandlung eine Liste aus Elementen vom Typ collision_list_t geführt (siehe Vorlesung 4.2 - Hash der Jupyter Notebooks zu Programmieren 2)
Ist diese Liste leer, so wird ein neues Listenelement collision_list_t angefordert, mit den neuen Daten initialisiert und an dieser Stelle eingetragen.
Befindet sich dort schon eine Liste mit Elementen, so wird das neue Listenelement vorne in dieser Liste eingefügt.
Ist das neue Element bereits in der Liste vorhanden, so wird es überschrieben.

Wird ein Element gesucht, so wird entsprechend der Hashfunktion zuerst die zugeordnete Stelle idx der Hashtabelle ermittelt.
Danach wird die an Stelle hasharray[idx] hinterlegte Liste durchsucht und das gesuchte Element zurückgegeben.
Wird das Element nicht gefunden, so wird 0 zurückgegeben.

Geht man nun nacheinander alle Listen der Hashtabelle elementweise durch, so bekommt man alle abgespeicherten Elemente der Hashtabelle.

Implementieren Sie nun die Funktion

```
void hash_resize(hash_t *hash, long newsize)
```

- die eine neue Hashtabelle mit der Größe newsize mit den Wörtern aus der übergebenen "alten" Hashtabelle hash richtig anlegt,
- die alte Hashtabelle freigibt und
- die neue Hashtabelle dann wiederum mittels hash zurückgibt.

Testen Sie Ihre Funktion ausführlich. Achten Sie dabei darauf, dass Fehler korrekt behandelt werden, keine Daten verloren gehen oder überschrieben werden und geben Sie alle angeforderten Speicherbereiche wieder ordnungsgemäß frei.

- rehash_inc.h
- rehash_iue.c
- rehash.c

Verzeichnis herunterladen