

Beispiel A

List Count

Zu Ihrer Angabe erhalten Sie drei Dateien:

list_count_inc.h: stellt verwendete Strukturen und Deklarationen zur Verfügung.

list_count_iue.c: hält bereits entwickelte Funktionen bereit.

list_count.c hier kommt Ihr Code hinein. Aus main() werden Ihre Funktionen und auch Funktionen aus list_count_iue.c aufgerufen.

Erstellen Sie ein Projekt bzw. ein ausführbares Programm mit beiden .c Dateien. Die .h Datei wird in beiden .c Dateien inkludiert.

list_t stellen die Elemente einer doppelt verketteten Liste dar. Gespeichert werden Worte im Datentyp data_t. Die Struktur data_t enthält die Zeichenkette word und eine Zähler count.

In der Datei list_count_iue.c stellen wir Ihnen einige Funktionen zu Initialisierung bereit, welche dann auch in list_count.c aufgerufen werden.

Mit der Funktion list_push() 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() extrahiert die Wörter einer Zeichenkette und erzeugt ein dynamisch alloziertes Hilfsfeld von data_c, welches dann einfach in eine Liste umgewandelt werden kann. free_data gibt dieses Hilfsfeld wieder frei.

Die Funktion init_data() legt Testdaten in einem dynamisch alloziertes Hilfsfeld an. Free_data(data_t **data) gibt dieses auch wieder frei.

In der Funktion main() werden Worte mittels init_data() in einem dynamisch angelegten Feld data_t *data abgespeichert und danach in eine Liste list_t *list eingefügt.

Der Zähler count in den Listenelementen wird mit 1 initialisiert.

Implementieren Sie die Funktion




```
void list_count(list_t **plist)
```

welche mehrfach vorkommende Einträge aus der Liste entfernt und dabei in count der verbliebenen Einträge mitzählt, wie oft das Wort vorgekommen ist.

Die Liste wird vor und nach dem Aufruf von list_count ausgegeben.

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_count_inc.h
-  list_count_iue.c
-  list_count.c

Verzeichnis herunterladen

Beispiel B

Ring Double

Zu Ihrer Angabe erhalten Sie drei Dateien.

ring_double_inc.h: stellt verwendete Strukturen und Deklarationen zur Verfügung.

ring_double_jue.c: hält bereits entwickelte Funktionen bereit.

ring_double.c hier kommt Ihr Code hinein. Aus main() werden Ihre Funktionen und auch Funktionen aus ring_double_jue.c aufgerufen.

Erstellen Sie ein Projekt bzw. ein ausführbares Programm mit beiden .c Dateien. Die .h Datei wird in beiden .c Dateien inkludiert.

Der Ringbuffer stellt einen Speicherplatz für eine festgelegte Anzahl von Datensätzen bereit.

Mit einer Schreibfunktion wird, solange noch Platz ist, von vorne nach hinten in den Ring geschrieben. Kommt die Schreibfunktion 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:

buffer ist ein Feld für einzelne Wörter der Struktur data_t,

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.

```
|----- ring_t -----|
|                         |
| buffer: |w4 |w5 |   |   |w1 |w2 |w3 | |
|                         |
|                         |
|                         |
| wr = 2 -----         |
| rd = 4 -----         |
| size = 7               |
| fill = 5               |
|-----|
```

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, sofern Elemente im Buffer stehen, die Stelle rd aus buffer ausgelesen und danach rd ebenfalls um eins weitergerückt. Hingegen muss fill (die Anzahl der Elemente im Buffer) um eins erniedrigt werden. Falls rd über das Ende des Buffers hinauszeigt, wird rd wieder auf 0 gesetzt, somit beim nächsten Lesevorgang wieder beim Anfang von buffer zu lesen begonnen.

In einem Ringbuffer mit der Feldgröße size und mit abgespeicherten Daten der Anzahl fill, liegen somit die Daten - von buffer[rd] bis buffer[wr-1], falls sich alle Elemente bis zum Ende des Buffers ausgehen, beziehungsweise - von buffer[rd] bis buffer[size-1] und von buffer[0] bis buffer[wr-1], falls sich alle Elemente bis zum Ende des Buffers nicht mehr ausgehen.

ring_init() reserviert in ring den Buffer buffer mit sz Elementen und setzt size, rd, wr und fill für einen leeren Buffer, ring_print() gibt den Buffer aus und ring_free() gibt ihn frei.

Init_data() legt ein Feld von Testdaten im dynamisch allozierten Feld data an und free_data() gibt dieses Feld 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_double(ring_t *ring);
```

welche einen bestehenden Ringbuffer ring auf die doppelte Größe vergrößert. Die Daten sollen danach im neuen Buffer ebenfalls konsistent vorliegen. Rückgabewert ist die neue Größe des Ringbuffers. Kann der Buffer nicht vergrößert werden, so soll er nicht verändert und -1 zurückgegeben werden. Setzen Sie size, rd, wr 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.



Verzeichnis herunterladen

Beispiel C

Tree Balance

Zu Ihrer Angabe erhalten Sie drei Dateien:

tree_balance_inc.h: stellt verwendete Strukturen und Deklarationen zur Verfügung.

tree_balance_iue.c: hält bereits entwickelte Funktionen bereit.

tree_balance.c hier kommt Ihr Code hinein. Aus main() werden Ihre Funktionen und auch Funktionen aus tree_balance_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 Datentyp tree_c stellt die Elemente eines binären Suchbaums dar. Gespeichert werden Worte im Datentyp data_c und die Sortierung im Baum erfolgt alphabetisch nach den Worten.

In der Datei tree_balance_iue.c stellen wir Ihnen einige Funktionen zur Initialisierung bereit, welche dann auch in tree_balance.c aufgerufen werden.

tree_insert() fügt ein neues Element mit den Daten data_c im Baum ein, tree_search() sucht ein Element, tree_print() gibt den Baum aus und tree_free() gibt ihn wieder frei.

Mit init_data() werden Testdaten in einem dynamisch allokierendes Feld data aufgebaut und free_data() gibt dieses Feld auch wieder frei.

Danach werden diese Testdaten in einem Baum eingetragen, mit welchem dann weitergearbeitet werden soll.

Implementieren Sie nun die Funktion

```
tree_balance(tree_c **ptree)
```

die den übergebenen Baum *ptree ausbalanciert.

Wir stellen Ihnen hierzu die Hilfsfunktionen

long tree_size(tree_c *tree), welche die Anzahl der Elemente im Baum tree liefert, und

tree_c **tree2feld(tree_c *tree, long *size), welche ein mit malloc neu angelegtes Feld mit Zeigern auf die sortierten Baumelemente aus tree zurückgibt. In *size wird die Anzahl der Baumelemente zurückgegeben.

Selbstverständlich können Sie diese Funktionen beim Ausbalancieren verwenden, beispielsweise um ein Hilfsfeld mit Zeigern auf die Baumelemente anzulegen, welches sie dann zum eigentlichen ausbalancieren heranziehen.

Sie können auch auf die anderen bestehenden Funktionen zurückgreifen und sich beliebige Hilfsfunktionen schreiben, wie beispielsweise eine Funktion balance_insert(), die rekursiv Teile des Feldes in einen Baum einfügt und von tree_balance() aufgerufen wird.

Hierbei stellen wir Ihnen frei, ob Sie den neuen Baum Element für Element wieder aufbauen wollen und danach den alten Baum freigeben oder die bestehenden Baumelemente wiederverwenden und nur die Zeiger left und right anpassen wollen.

Stellen Sie sicher, dass danach der gültige ausbalancierte Baum in *ptree verfügbar ist.

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.



- tree_balance_inc.h
- tree_balance_iue.c
- tree_balance.c