Hochschule Niederrhein	Programmentwicklung II	Prof. Dr. Rethmann
Fachbereich 03	Übung 1	Prof. Dr. Stockmanns
Bachelor Informatik	SoSe 2020	Prof. Dr. Brandt

Modulare Programmierung in C

Lernziele

Anwenden der modularen Programmierung in C sowie Vertiefen der Kenntnisse über die Gültigkeit und Sichtbarkeit von Variablen. Modulare Programmierung beschreibt die Aufteilung eines Programms in Module, die einzeln geplant, programmiert und getestet werden können.

Aufgabe 1:

Implementieren Sie eine Datenstruktur `stack_t` mittels modularer Programmierung. Die Struktur soll folgende Operationen unterstützen:

- `stack_t *createStack();`
  erzeugt einen leeren Stack
- `char isEmpty(stack_t *s);`
  prüft, ob der Stack `s` leer ist
- `void push(stack_t *s, float value);`
  legt den Wert `value` auf den Stack `s`
- `float top(stack_t *s);`
  liefert das zuletzt eingefügte Element des Stacks `s`
- `void pop(stack_t *s);`
  entfernt das zuletzt eingefügte Element vom Stack `s`
- `char getError(stack_t *s);`
  liefert den Inhalt der Fehlervariablen
- `void destroyStack(stack_t *s);`
  zerstört den Stack und gibt belegten Speicherplatz frei

Der Stack soll bei Bedarf automatisch vergrößert werden, sodass er beliebig viele Float-Werte speichern kann. Wenn auf einem leeren Stack ein `top` oder `pop` ausgeführt wird, soll eine Fehlervariable gesetzt werden.

Schreiben Sie ein Programm in C, dass einen in umgekehrter polnischer Notation gegebenen arithmetischen Ausdruck einliest, mit Hilfe des Stacks auswertet und das Ergebnis anzeigt. Unter `https://de.wikipedia.org/wiki/Umgekehrte_polnische_Notation` finden Sie Informationen zu der umgekehrten polnischen Notation und wie man mit deren Hilfe den Wert arithmetischer Ausdrücke berechnen kann.

Aufgabe 2: (muss bearbeitet und abgegeben werden)

Abgabe der Programme bis Mittwoch, 06.05.2020 um 12 Uhr Abgabe der Reviews bis Mittwoch, 13.05.2020 um 12 Uhr

Implementieren Sie eine Datenstruktur `heap_t` mittels modularer Programmierung. Die Struktur soll folgende Operationen unterstützen:

- `heap_t *createHeap();`
  erzeugt einen leeren Heap
- `char insert(heap_t *h, int val);`
  fügt den Wert `val` in den Heap `h` ein
- `int minimum(heap_t *h);`
  liefert das minimale Element des Heaps `h`
- `char extractMin(heap_t *h);`
  entfernt das minimale Element aus dem Heap `h`
- `char getError(heap_t *h);`
  liefert den Inhalt der Fehlervariablen des Heaps `h`
- `char* toString(heap_t *h);`
  liefert den Inhalt der Heaps `h` als Zeichenkette
- `void destroyHeap(heap_t *h);`
  zerstört den Heap `h` und gibt belegten Speicherplatz frei

Der Heap soll bei Bedarf automatisch vergrößert werden, sodass er beliebig viele `int`–Werte speichern kann. Wenn `minimum` auf einem leeren Heap ausgeführt wird, soll eine Fehlervariable gesetzt werden.

Erstellen Sie einen Testtreiber und Testfälle, um die Korrektheit Ihrer Implementierung nachzuweisen. Test sind grundsätzlich so zu gestalten, dass sie automatisch ablaufen und die Ergebnisse selbst überprüfen.

Informationen darüber, was ein Heap ist, wie man einen Heap mit Hilfe eines Arrays implementiert und wie die einzelnen Operationen implementiert werden können, finden Sie unter `https://de.wikipedia.org/wiki/Bin%C3%A4rer_Heap`.

Aufgabe 3: *Routenplanung*

Erstellen Sie eine Datenstruktur `pqueue_t`, die beliebig viele Zeichenketten speichern kann. Zur internen Kapselung der Zeichenketten und Prioritätswerte, erstellen Sie eine Datenstruktur `pqentry_t`. Verwenden Sie für die Priorität den Datentyp `float`.

Die Datenstruktur soll beliebig viele Datensätze aufnehmen können, d.h. sie muss bei Bedarf automatisch vergrößert werden. Zudem soll die Implementierung einen Zugriff auf die interne Repräsentation verhindern (Datenkapselung).

Implementieren Sie für die Datenstruktur `pqueue_t` die folgenden Operationen:

- `pqueue_t* pq_create();`
  erstellt eine neue, leere Vorrangwarteschlange.
- `void pq_insert(pqueue_t *pq, char *value, float p);`
  fügt den Wert `value` mit der Priorität `p` in die Vorrangwarteschlange `pq` ein.
- `char* pq_extractMin(pqueue_t *pq);`
  liefert den Wert aus der Vorrangwarteschlange `pq` mit höchster Priorität, also mit kleinstem nummerischen Wert der Komponente `priority` und entfernt den Eintrag aus `pq`.
- `void pq_decreaseKey(pqueue_t *pq, char* value, float p);`
  ändert die Priorität des Wertes `value` in der Vorrangwarteschlange `pq` auf den neuen Wert `p`.
- `void pq_remove(pqueue_t *pq, char* value);`
  löscht den Wert `value` aus der Vorrangwarteschlange `pq`.
- `void pq_destroy(pqueue_t *pq);`
  zerstört die Vorrangwarteschlange `pq` und gibt den belegten Speicher wieder frei.

Diese Datenstruktur wird für die Berechnung kürzester Wege mittels des Algorithmus von Dijkstra benötigt. Oft wird ein binärer Heap verwendet, um eine Vorrangwarteschlange zu realisieren.