Praktikum 1: Modulare Programmierung

1. Lernziele

Anwenden der modularen Programmierung sowie Vertiefen der Kenntnisse über die Gültigkeit und Sichtbarkeit von Variablen.

2. Aufgabe

Modulare Programmierung beschreibt die Aufteilung eines Programms in Module, die einzeln geplant, programmiert und getestet werden können. In der ersten Praktikumsaufgabe soll dazu eine Vorrangwarteschlange und ein Testtreiber entwickelt werden und diese mit einem vorgegebenen Hauptprogramm kombiniert werden.

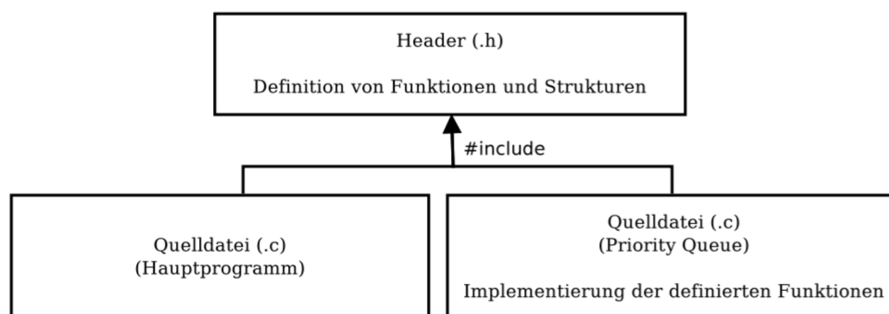

Abbildung 1: Modulare Programmierung

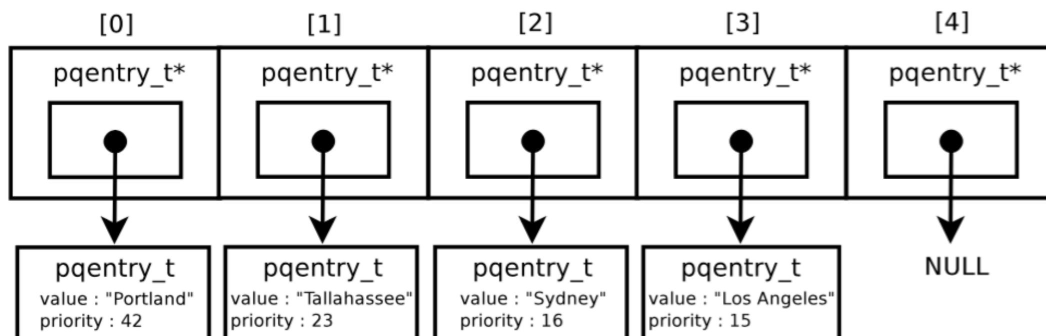

Abbildung 2: Datenstruktur der Vorrangwarteschlange

Eine Vorrangwarteschlange ist eine erweiterte Form der Warteschlange. Elemente, die in die Warteschlange gelegt werden, wird ein Prioritätswert mitgegeben, der die Reihenfolge der Abarbeitung der Elemente bestimmt. Dabei steht der niedrigste Prioritätswert für die höchste Priorität.

Teil 1: Erstellen Sie eine Datenstruktur `priorityqueue_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 `priorityqueue_t` die folgenden Operationen:

- `priorityqueue_t* pqueue_create()`
  erstellt eine neue, leere Vorrangwarteschlange.
- `void pqueue_insert(priorityqueue_t *pq, char* value, float p)`
  fügt den Wert `value` mit der Priorität `p` in die Vorrangwarteschlange `pq` ein.
- `char* pqueue_extractMin(priorityqueue_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 pqueue_decreaseKey(priorityqueue_t *pq, char* value, float p)`
  ändert die Priorität des Wertes `value` in der Vorrangwarteschlange `pq` auf den neuen Wert `p`.
- `void pqueue_remove(priorityqueue_t *pq, char* value)`
  löscht den Wert `value` aus der Vorrangwarteschlange `pq`.
- `void pqueue_destroy(priorityqueue_t *pq)`
  zerstört die Vorrangwarteschlange `pq` und gibt den belegten Speicher wieder frei.

Teil 2: Um die Korrektheit Ihrer Implementierung nachzuweisen, schreiben Sie einen Testtreiber (Programm), der Ihre Datenstruktur testet. Die Tests müssen vollständig automatisiert ablaufen.

Teil 3: Bestimmen Sie die Laufzeiten der Funktion `pqueue_extractMin` sowie der Funktion `pqueue_insert`, indem Sie den unten stehenden Quellcode verwenden. Welchen Unterschied bei den Laufzeiten stellen Sie fest und wie können Sie diesen begründen?

Falls bei Ihrer Implementation die `pqueue_extractMin` Operationen deutlich länger brauchen als die `pqueue_insert` Operationen, dann ändern Sie Ihren Algorithmus so ab, dass die `pqueue_extractMin` Operation effizienter wird. Dafür können Sie beispielsweise die Werte in der `pqueue_insert` Funktion, gemäß der Priorität, sortiert einfügen. Analysieren Sie anschließend erneut das Laufzeitverhalten.

Welchen Unterschied stellen Sie fest, wenn Sie die Werte einmal mit aufsteigenden, absteigenden und zufällig verteilten Prioritätswerten einfügen?

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pqueue.h"

#define MAX 100000

char* randomString(int size) {
    int i;
    char* str = (char*) malloc((size+1)*sizeof(char));
    for(i = 0 ; i < size ; ++i) {
        str[i] = (rand() % 26) + 'A';
    }
    str[size] = '\0';
    return str;
}

int main(int argc, char **argv) {
    int i;
    char* strings[MAX];
    clock_t tic, toc;

    srand(time(NULL));

    for(i = 0 ; i < MAX ; ++i) {
        strings[i] = randomString(8);
    }

    priorityqueue_t *pq = pqueue_create();

    tic = clock();
    for(i = 0 ; i < MAX ; ++i) {
        pqueue_insert(pq, strings[i], rand() % 100);
    }
    toc = clock();

    printf("insertion time: %.2f\n", (float)(toc-tic) / CLOCKS_PER_SEC);

    tic = clock();
    for(i = 0 ; i < MAX ; ++i) {
        pqueue_extractMin(pq);
    }
    toc = clock();

    printf("extract time: %.2f\n", (float)(toc-tic) / CLOCKS_PER_SEC);

    for(i = 0 ; i < MAX ; ++i) {
        free(strings[i]);
    }
    pqueue_destroy(pq);

    return 0;
}
```

3. Hinweise zum Quellcode

Im gegebenen Quellcode werden Variablen des Datentyps `clock_t` für einen Zeitstempel und die Funktion `clock_t clock(void)` für dessen Erzeugung aus der Bibliothek `time.h` verwendet. Diese können in Anwendungen benutzt werden, um die Zeiten zu messen, die Ihre Algorithmen brauchen.

Die Funktion `clock_t clock(void)` gibt die verstrichene Prozessorzeit seit Start des Programms in sogenannten Clocks zurück. Um diesen Wert in eine Zeit aus Sekunden umzurechnen, stellt die Bibliothek `time.h` die Konstante `CLOCKS_PER_SEC` zur Verfügung.

Der Datentyp `clock_t` ist ein Alias für einen arithmetischen Wert, der in verschiedenen C-Bibliotheken zum Teil unterschiedlich definiert ist. In der GNU C-Bibliothek `glibc` ist `clock_t` zum Beispiel ein Alias für den Datentyp `long`.

4. Testat

Voraussetzung ist jeweils ein fehlerfreies, korrekt formatiertes Programm. Der korrekte Programmlauf muss nachgewiesen werden. Sie müssen in der Lage sein, Ihr Programm im Detail zu erklären und ggf. auf Anweisung hin zu modifizieren.

Das Praktikum muss spätestens zu Beginn des nächsten Praktikumtermins vollständig bearbeitet und abtestiert sein.