"Informatik II"

Studiengänge B_ET, B_IK, B_Ma, B_FM, M_IG

2017

Hinweis: In der vorliegenden Musterklausur werden beispielhaft mögliche Aufgentypen gezeigt. In der Klausur können ausdrücklich auch andere Konstellationen (z.B. dynamische Datenstrukturen werden in einer Klasse verwaltet) auftreten.

1. Dynamische Datenstrukturen und objektorientierte Programmierung (25 Punkte)

   Gegeben sei eine Datenstruktur mit folgendem Aufbau

```
struct lelem {
   float w;
   lelem * pNext;
};
```

   Basierend auf dieser Datenstruktur soll eine Klasse **liste** mit der folgenden Klassendefinition aufgebaut werden:

```
class liste {
     private:
        lelem * anchor;           // Zeiger auf den Beginn einer von
                                  // liste verwalteten einfach verketteten
                                  // Liste
     public:
        liste();                  // Konstruktor  zum Erzeugen
                                  // einer leeren Liste
        ~liste();                 // Destruktor, gibt den ges. dynamisch
                                  // angeforderten Speicherplatz frei
        void einfuegen (float x);  // einfuegen eines neuen Listenelementes
        void ausgabe_liste();     // Ausgabe aller Datenelemente der Liste
        int ausgabe_liste_datei(char * filename);
                // Ausgabe aller Datenelemente der Liste in Datei
};
```

Die Funktionalität der Methoden der Klasse **liste** ist wie folgt definiert:

- einfuegen:
  Die Methode **einfuegen** unterscheidet, ob die von einem Objekt der Klasse **liste** adressierte einfach verkettete Liste leer ist oder nicht. Wenn die Liste leer ist, soll ein neues erstes Listenelement angelegt werden, sonst soll ein neues letztes Element erzeugt werden. Der Datenwert des neuen Elements soll mit dem übergebenen Wert **x** geschrieben werden, das neue Datenelement soll in der Nachfolgeinformation **next** den Wert *NULL* enthalten.
  5 Punkte
- ausgabe_liste:
  Linksbündige Ausgabe aller Datenelemente der Liste mit einer Breite von 15 Zeichen für jede Zahl. Es sollen , sofern möglich, 5 Elemente auf einer Bildschirmzeile stehen.
  4 Punkte
- ausgabe_liste_datei:
  Die Datenelemente der einfach verketteten Liste sollen durch Leerzeichen getrennt in eine Datei geschrieben werden, deren Name vom Nutzer anzugeben ist. Die Methode gibt einen Integer-Wert mit folgender Bedeutung zurück:
  - 0 - Ausgeben in Datei mißglückt
  - 1 - Ausgeben in Datei gelungen 4 Punkte

Implementieren Sie den Konstruktor (1 Punkt), den Destruktor (2 Punkte) sowie die angegebenen Methoden der Klasse **liste**.
Schreiben Sie ein Rahmenprogramm, das die oben beschriebenen Methoden benutzt und den folgenden Aufbau hat:

** Konstruieren Sie zunächst ein Objekt, welches eine leere Liste verwaltet. Fügen Sie nun in das ursprünglich leere Objekt 10 Elemente ein, deren Datenwerte vom Nutzer abzufragen sind.

** Geben Sie dann die Datenwerte **w** der Elemente der einfach verketteten Liste, die durch das Objekt verwaltet wird, auf dem Bildschirm aus. Schreiben Sie dann die Datenwerte **w** der Elemente der einfach verketteten Liste, die durch das Objekt verwaltet wird, in eine Datei, deren Namen vom Nutzer zu erfragen ist.

2. Objektorientierte Programmierung und Datenstrukturen(35 Punkte)

Implementieren Sie eine C++-Klasse *priority_queue*, die eine *priorisierte Warteschlange* realisiert. Die priorisierte Warteschlange soll folgende Eigenschaften haben:

- Die Warteschlange kann Jobs speichern, die der Einfachheit halber aus einem Programmnamen und einer extern vergebenen Priorität bestehen.

```
struct jobs {
   char jobname[50];
   int prio;
   };
```

- es können Elemente an das Ende der Warteschlange abgelegt (Methode *write()*) und wieder entnommen werden (Methode *read()*). Die Reihenfolge der Entnahme soll bei Benutzung der Methode *read* wie folgt geregelt sein:
  - die Warteschlange wird nach dem Job durchsucht, dessen Priorität am größten ist. Dieser wird dann entnommen.
  - stehen mehrere Jobs gleicher größter Priorität in der Warteschlange, so soll die Reihenfolge der Entnahme die Gleiche sein wie die Reihenfolge des Ablegens (first-in-first-out).
- die Datenstruktur, mit der die Warteschlange realisiert wird, ist frei wählbar (z.B. einfach oder doppelt verkettete Liste). Die Warteschlange soll aber in jedem Fall eine maximale Größe besitzen. Diese und die aktuelle Anzahl der gespeicherten Jobs sind in privaten Membervariablen zu speichern.
- Daten können nur aus der Warteschlange entnommen werden, wenn sie nicht leer ist
- Daten können nur in die Warteschlange geschrieben werden, wenn sie nicht voll ist
- Die Warteschlange soll folgende private Methoden enthalten:
  - bool isEmpty() - liefert true, wenn die Warteschlange keine Elemente enthält (2 Punkte)
  - bool isFull() - liefert true, wenn die Warteschlange keine Elemente mehr aufnehmen kann (2 Punkte)
- Die Warteschlange soll folgende öffentliche Methoden enthalten:
  - ein Konstruktor, der die erforderlichen Indizes bzw. Zeiger entsprechend belegt. (1 Punkt)
  - int getSize() - liefert die Anzahl der aktuell gespeicherten Elemente zurück (2 Punkte)
  - void clear() - löscht die gesamte Warteschlange (3 Punkte)

* bool read(jobs & job) - Ergebnis ist true, wenn die Operation read erfolgreich war, sonst false. Der entnommene Job steht im Parameter job (7 Punkte)
* bool write(jobs job) - Ergebnis ist true, wenn die Operation erfolgreich war, sonst false. (4 Punkte)
* *die Operationen* **read** *bzw.* **write** *sollen mit Hilfe der privaten Methoden* **isEmpty** *bzw.* **isFull** *prüfen, ob deren Ausführung möglich ist*
* Friend-Operatorfunktion

  `friend ostream & operator << (ostream & os, const priority_queue & pq)`

  die Jobs, die in einem Objekt der Klasse *priority_queue* abgelegt sind, auf ein Objekt der Klasse ostream (in der Regel cout) ausgibt. Dabei sollen die Jobs in der Reihenfolge Ihres Einfügens in die Warteschlange ausgegeben werden. (5 Punkte)

Testen Sie die Klasse geeignet (alle öffentlichen Methoden sollen mindestens einmal verwendet werden). (9 Punkte)

3. Algorithmierung (9 Punkte)

Eine Primzahl ist eine natürliche Zahl n ($n \geq 2$), die nur durch 1 sowie sich selbst ohne Rest teilbar ist. Entwerfen Sie eine Funktion

```
bool  is_prime(int zahl);
```

die den Wert **true** zurückgibt, wenn der übergebene Parameter **zahl** eine Primzahl ist, den Wert **false**, wenn **zahl** keine Primzahl ist. Verwenden Sie die Funktion **is_prime** dazu, um **alle** Primzahldrillinge im Intervall $1 \ldots 1000$ auf dem Bildschirm auszugeben.

Hinweis: Ein Primzahldrilling oder Primzahl-Triplett ist eine Menge von Primzahlen der Form p, p+2, p+6 oder p, p+4, p+6, d.h. eine Menge von 3 natürlichen Zahlen (z_1, z_2, z_3) ist dann ein Primzahldrilling, wenn gilt z_1 ist eine Primzahl **und** z_2 ist eine Primzahl **und** z_3 ist eine Primzahl, sowie $z_2 = z_1 + 2$ oder $z_2 = z_1 + 4$ sowie $z_3 = z_1 + 6$. Die kleinsten Primzahldrillinge sind (5,7,11), (7,11,13) und (11,13,17).