Lehrstuhl für Informatik III
Prof. Dr.-Ing. M. Nagl

Klausur
Grundgebiete der Informatik I,II

(DPO98/04)

Datum: 22. 02. 2005

Teil II: Algorithmen und Programmiertechniken

Name: _____
(in Druckschrift)

Matr. Nr.: _____

Unterschrift _____

Aufgabe	Max. Punkte	Korrektur		Einsichtnahme	
		Punkte	Kürzel	Punkte	Kürzel
II.1	10				
II.2	10				
II.3	20				
II.4	20				
Σ	60				

Aufgabe II.1 Syntax *(10 Punkte)*

(a) In der Vorlesung wurde erläutert, dass bei der Syntax einer Programmiersprache zwischen lexikalischer, kontextfreier und kontextsensitiver Syntax unterschieden wird. Geben Sie aus dem unten angegebenen C++-Programm jeweils ein Beispiel für die drei Syntaxarten an. *(3 Punkte)*

```
float f(float b, unsigned int e) {
   float r = 1.0;
   while (e > 0) {
      if ((e % 2) != 0) {
         r = r * b;
      }
      b = b * b;
      e = e / 2;
   }
   return r;
}

int main()
{
   float s = f(2,4);
   return 0;
}
```

- lexikalische Syntax z.B.: 1.0 (1 Punkt)
- kontextfreie Syntax z.B.:

  ```
  if ((e % 2) != 0) {
    r = r * b;
  }
  ```

  (1 Punkt)
- kontextsensitive Syntax z.B.: `f(2,4)` ist eine Anwendungsstelle der Deklaration `float f(float b, unsigned int e)`. (1 Punkt)

(b) Geben Sie für jede der drei Syntaxarten eine Regel an, die der jeweiligen Syntaxart zugeordnet ist. Wo möglich benutzen Sie die EBNF und kennzeichnen Sie dabei Terminale durch Unterstreichen. *(3 Punkte)*

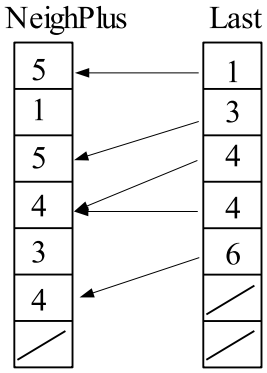
- lexikalische Syntax z.B.: digit ::= 0|1|2|3|4|5|6|7|8|9 (1 Punkt)
- kontextfreie Syntax
  z.B.: do_while_statement ::= do { statement_list } while (expression) (1 Punkt)
- kontextsensitive Syntax z.B.: Die Typen der Parameter bei einem Prozeduraufruf müssen kompatibel zu denen in der Prozedurdeklaration sein und die Reihenfolge muss übereinstimmen. (1 Punkt)

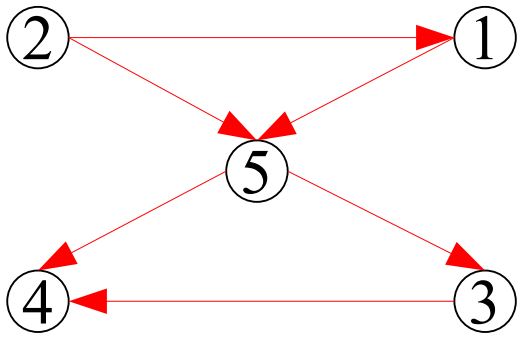
(c) In der Vorlesung wurden zwei Parameterübergabe-Mechanismen vorgestellt. Nennen und erläutern Sie diese. Geben Sie umgangssprachlich für jeden Mechanismus kurz an, wie dieser sinnvoll angewendet wird. *(4 Punkte)* Call-by-value kopiert den betreffenden Aktualparameter (1 Punkt). Änderung an der Kopie sind nur im aktuellen Unterprogrammaufruf sichtbar. Call-by-value dient dem Aufrufer als Garantie, dass der Wert des Aktualparametes während des Unterprogrammaufrufs nicht verändert wird (1 Punkt für ≥ 1 Beispiel). Call-by-reference kopiert nicht, sondern führt lediglich einen neuen Bezeichner für den Aktualparameter ein (1 Punkt). Änderung wirken sich daher auch auf den Aktualparameter des Aufrufers aus. Sinnvoll ist der Einsatz von call-by-reference für Ausgabeparameter und bei der Übergabe großer Datenstrukturen, z.B. große Felder (1 Punkt für ≥ 1 Beispiel).

Aufgabe II.2 Graphenspeicherung *(10 Punkte)*

(a) Für die knotenorientierte Speicherung von gerichteten Graphen kann beispielsweise die “verdichtete sequentielle Speicherung” verwendet werden, siehe Vorlesung. Hierbei werden zwei Felder NeighPlus und Last verwendet. Im Feld NeighPlus werden für jeden Knoten die ausgehenden Kanten des Graphen dadurch gespeichert, dass seine Nachfolgerknoten aufgeführt werden. Das Feld Last speichert an der Stelle k den letzten Index des Feldes NeighPlus, an dem ein Nachfolgerknoten des Knotens k gespeichert wird. An dem folgenden Index des Feldes NeighPlus steht dann der erste Nachfolger des Knotens $k+1$. Ist $Last[k] == Last[k-1]$ so hat der Knoten k keine Nachfolger.

Rekonstruieren Sie aus der verdichteten sequentiellen Speicherung den Graphen. *(3 Punkte)*

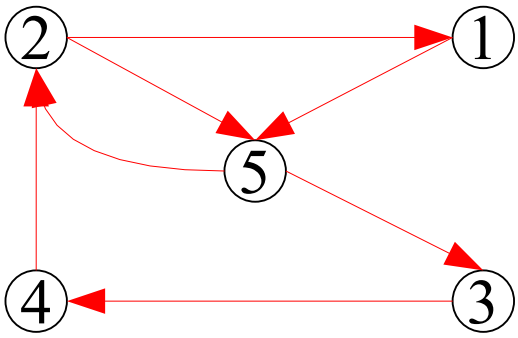


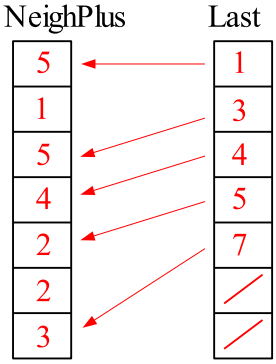


(b) An dem Graphen aus Aufgabenteil (a) sollen nun folgende Änderungen vorgenommen werden:

1. Lösche Kanten von 5 nach 4
2. Erzeuge Kante von 5 nach 2
3. Erzeuge Kante von 4 nach 2

Geben Sie erst den geänderten Graphen und dann die geänderte Datenstruktur zur Speicherung dieses Graphen an. *(4 Punkte)*





(c) Welchen **Vorteil** hat diese Darstellung verglichen mit sequentiellen Adjazenzlisten? Begründen Sie ihre Antwort! *(1 Punkt)*

Wesentliche bessere Ausnutzung des Speicherplatzes, wenn Graph nicht vollständig gefüllt.

(d) Welchen **Nachteil** hat diese Darstellung verglichen mit sequentiellen Adjazenzlisten? Begründen Sie ihre Antwort! *(1 Punkt)*

Hinzunahme neuer Kanten sehr schwer, da beide Felder umorganisiert werden müssen.

(e) Sowohl die sequentiellen Adjazenzlisten als auch ihre verdichtete Darstellung eignen sich nicht für häufig zu ändernde Graphen. Welche Graphdarstellung löst dieses Problem? *(1 Punkt)*

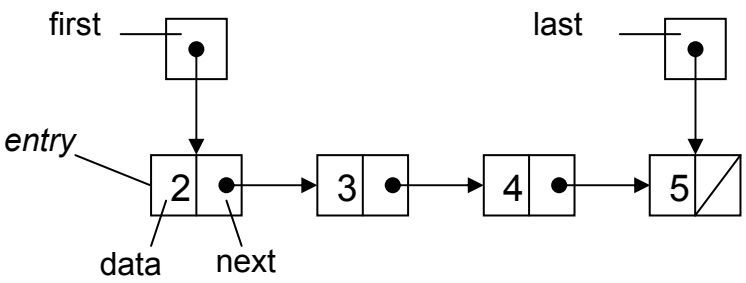
Verkettete Adjazenzlisten

Aufgabe II.3 Realisierung des ADT Warteschlange *(20 Punkte)*

In dieser Aufgabe soll eine Warteschlange als ADT in C++ implementiert werden. Die Schlange speichert Werte des Typs integer. Der ADT soll folgende Schnittstelle realisieren:

```
class Queue {
public:
   // Konstruktor, initialisiert die Datenstruktur;
   Queue();

   // Reiht den Wert data hinten in die Schlange ein
   void enqueue(int data);

   // liest den vordersten Wert aus der Schlange und entfernt ihn
   int dequeue();

   // liefert true, wenn keine weiteren Elemente mehr aufgenommen
   // werden können
   bool isFull();

   // liefert true, wenn die Schlange leer ist
   bool isEmpty();

private:
   ...
};
```

Die folgende Abbildung verdeutlicht die Datenstruktur zur internen Realisierung des ADT: Die Schlange ist als einfach verkettete Liste realisiert. Die Elemente der Liste vom Typ **entry** bestehen aus dem integer-Wert **data** und einem Zeiger auf das nächste Element (**next**). Der Zeiger **first** zeigt auf den Anfang der Liste. Hier werden Elemente mit **dequeue** gelesen und entfernt. Der Zeiger **last** zeigt auf das letzte Element der Liste, neue Einträge werden hinter dem letzten Element angehängt.



(a) Definieren Sie den Datentyp **entry** und die Zeiger **first** und **last**. *(5 Punkte)*

```
struct entry {
      int data; //Lx
      entry* next; //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
   }; //Lx

entry* first; //Lx
entry* last; //Lx
//Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
      //Lx
```

(b) Implementieren Sie den Konstruktor, der die interne Datenstruktur geeignet initialisiert. Verwenden Sie keine „Dummy“-Elemente am Anfang und am Ende der Liste. *(2 Punkte)*

```
Queue::Queue() {
   first=last=NULL; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(c) Implementieren Sie die Funktione **dequeue**, die den Wert des ersten Elements zurückgibt und dieses dann aus der Schlange entfernt. Ist die Schlange leer, so gibt die Funktion den Wert -1 zurück. Beachten Sie auch den Spezialfall, dass das letzte Element aus der Liste entfernt wird. Vergessen Sie nicht, den Speicher des entfernten Elements freizugeben. *(8 Punkte)*

```
int Queue::dequeue() {
   int data; //Lx
   if (first!=NULL) { //Lx
      data=first->data; //Lx
      entry* old=first; //Lx
      first=first->next; //Lx
      if (first==NULL) last=NULL; //Lx
      delete old; //Lx
      return data; //Lx
   } else { //Lx
      return -1; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

Jetzt soll die Schlange verwendet werden, um ein Produzenten/Verbraucher-System zu simulieren. Dabei gibt es einen Produzenten, der „Produkte“ erzeugt und einen Konsumenten, der diese „Produkte“ verbraucht. In unserer Simulation werden beide durch Prozeduren dargestellt, die auf einer Schlange arbeiten. Die Produkte werden durch Integer-Werte dargestellt, die in der Schlange abgelegt werden. Der Produzent stellt Produkte bzw. Integerwerte in die Schlange ein, der Konsument liest sie aus. Die Schlange dient dabei der Entkopplung von Produzent und Konsument, so müssen Produzent und Konsument nicht immer genau abwechselnd gestartet werden.

(d) Implementieren Sie die Funktion produce, die ein Produkt erzeugt. Zur Kennzeichnung der Produkte wird der globale Zähler counter genutzt. Die Schlange ist über die Variable q ansprechbar. Ist noch Platz in der Schlange, soll der globale Produktzähler erhöht, sein Wert auf der Standard-Ausgabe ausgegeben und danach in die Schlange eingefügt werden. Ist die Schlange voll, soll stattdessen eine Warnung ausgegeben werden. *(5 Punkte)*

```
Queue q; //Instanz des ADT Schlange
int counter=0; //Produktzähler

void produce() {
   if (!q.isFull()) { //Lx
      counter++; //Lx
      cout << "Produkt erzeugt: "<< counter << "\n"; //Lx
      q.enqueue(counter); //Lx
   } else { //Lx
      cout << "Puffer voll, bitte erst Produkte verbrauchen" <<endl; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

Aufgabe II.4 Realisierung einer Schlange mit Prioritäten *(20 Punkte)*

Aufbauend auf der in der vorherigen Aufgabe realisierten Schlange soll nun eine Schlange mit Prioritäten als ADT in C++ implementiert werden. Diese Prioritätswarteschlange besteht aus mehreren Schlangen, die jeweils einer bestimmten Priorität entsprechen. Beim Einfügen wird über die Priorität die Schlange bestimmt, in die der Eintrag eingefügt wird. Beim Auslesen wird immer das erste Element der Schlange mit der dann höchsten Priorität zurückgegeben, die mindestens ein Element enthält. Die folgende Tabelle zeigt ein Beispiel mit drei Schlangen der Prioritäten 5, 6 und 7:

Priorität	*vorne – Inhalt der Schlange – hinten*
7	1, 4, 6
6	
5	8, 3, 4, 3

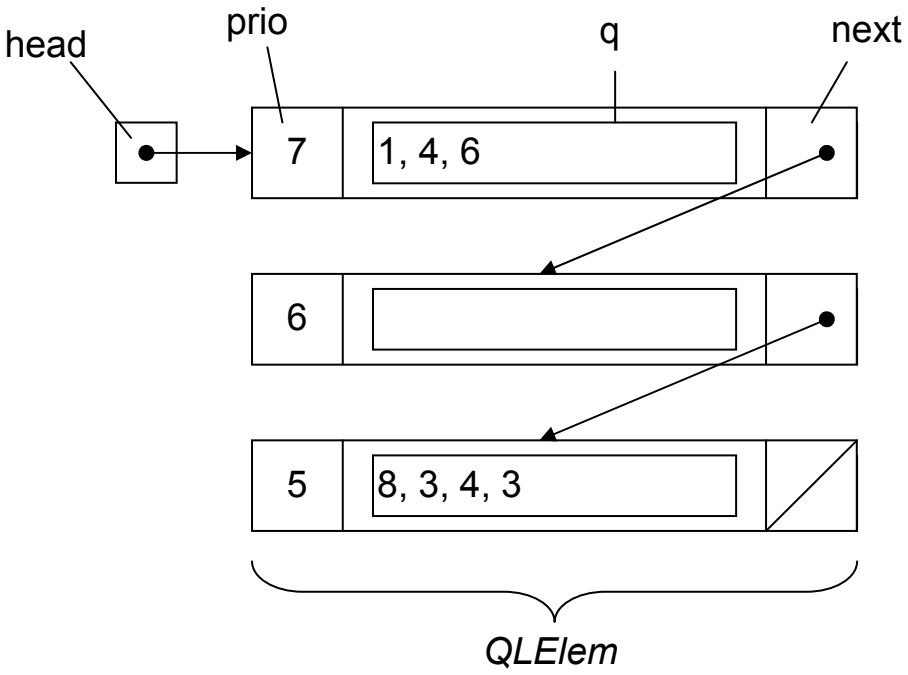
Beim Auslesen von Werten aus der Schlange werden in diesem Fall zunächst die Elemente der Schlange mit der Priorität 7 zurückgegeben, also beim ersten Aufruf 1, dann 4, schließlich 6. Da die Schlange mit der Priorität 6 leer ist, würden die folgenden Aufrufe die Werte aus der Schlange mit der Priorität 5 lieferen, also zuerst 8, usw.

Die Schnittstelle der Warteschlange mit Prioritäten soll die folgenden Methoden enthalten:

```
class PriorityQueue {
public:
   // Konstruktor, initialisiert die Schlange so,
   // dass genau die ganzzahligen Prioritäten von min
   // bis max unterstützt werden. Die unterstützten
   // Prioritäten können später nicht mehr geändert werden.
   PriorityQueue(int min, int max);

   // Fügt ein Element mit dem Wert val und der Priorität prio ein,
   // falls prio eine gültige Priorität ist und die Schlange der
   // Priorität nicht voll ist.
   // Bei Erfolg ist der Rückgabewert true, sonst false.
   bool pqEnqueue(int prio, int val);

   // Entnimmt ein Element aus der Queue und gibt dessen Wert im
   // Referenzparameter value zurück. Dabei wird das Elemente
   // mit der höchsten Priorität zurückgegeben, das zuerst eingefügt wurde.
   // Bei Erfolg ist der Rückgabewert true, ist die Schlange leer dann false.
   bool pqDequeue(int &value);

   // Der Rückgabewert ist true, wenn die Schlange leer ist, sonst false.
   bool pqIsEmpty();
private:
   ...
};
```

Intern wird für jede Priorität eine Instanz des ADT Queue aus der vorherigen Aufgabe genutzt. Im Konstruktor wird dazu einmalig eine verkettete Liste aufgebaut, die die Queue-Instanzen enthält. Beim Einfügen mit enqueue wird der übergebene Wert dann der Queue-Instanz mit der passenden Priorität hinzugefügt. Bei Aufruf von dequeue wird ein Wert aus der Queue-Instanz mit der größten Priorität entnommen, die nicht leer ist.

Die folgende Abbildung verdeutlicht die Realisierung. Die dargestellte Schlange wurde mit dem Konstruktoraufruf Queue(5, 7) erzeugt und dann mit Aufrufen von pqEnqueue gefüllt. Bspw. wurde mit dem Aufruf pqEnqueue(5,8) der Wert 8 in die Schlange der Priorität 5 eingefügt.



Der Zeiger head bildet den Anker der Datenstruktur. Die Elemente der verketteten Liste sind vom Typ QLElem. Ein QLElem besteht aus einer Komponente prio, die die Priorität der Schlange angibt, einer Komponente q, die die Instanz von Queue aus Aufgabe II.3 enthält, und einem Zeiger next für die Verkettung.

Verwenden sie im Folgenden die Schnittstelle der Schlange (Queue) von Aufgabe II.3 zum Zugriff auf die einzelnen Schlangen!

(a) Definieren Sie die Datenstruktur für den ADT. Neben dem Anker **head** und dem Datentyp **QLElem** sind auch zwei Integervariablen **minp** und **maxp** zu definieren, die die kleinste und größte Priorität der Schlange abspeichern. *(5 Punkte)*

```
struct  QLElem { //Lx
   Queue q;  //Lx
   int prio;  //Lx
   QLElem *next;  //Lx
};  //Lx

QLElem *head;  //Lx
int minp;  //Lx
int maxp;  //Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
```

(b) Realisieren Sie den Konstruktor. Er soll die Datenstruktur so initialisieren, dass für alle ganzzaligen Prioritäten zwischen **min** und **max** Einträge (**QLElem**) in der verketteten Liste stehen, die jeweils die Priorität **prio** und eine Instanz von **Queue** enthalten. Dabei sollen die Schlangen nach ihrer Priorität absteigend sortiert sein, die mit höchster Priorität soll also am Anfang der Liste stehen. Beachten Sie das korrekte Setzen des Ankerzeigers (**head**), den sauberen Abschluss der Liste und das Initialisieren von **maxp** und **minp**. *(8 Punkte)*

```
PriorityQueue::PriorityQueue(int min, int max)
{
   if (max<min) return; //Lx
   head=new QLElem; //Lx
   head->prio=min; //Lx
   head->next=NULL; //Lx

   QLElem *curr=head; //Lx
   for (int i=max;i>=min+1;i--) { //Lx
      curr->next=new QLElem; //Lx
      curr=curr->next; //Lx
      curr->prio=i; //Lx
      curr->next=NULL; //Lx
   } //Lx
   minp=min; //Lx
   maxp=max; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
    //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(c) Implementieren Sie die Funktion **pqEnqueue**. Diese fügt den Wert **val** in die Schlange mit der Priorität **prio** ein. Wurde der Wert erfolgreich eingeführt, ist der Rückgabewert **true**. Beachten Sie die Fälle, dass die Priorität ungültig ist und dass die gefundene Schlange voll ist. In beiden Fällen ist der Rückgabewert **false**.

(7 Punkte)

```
bool PriorityQueue::pqEnqueue(int prio, int val) {
   if (prio<minp || prio>maxp) return false; //Lx

   QLElem *curr=head; //Lx
   while (curr->prio!=prio) curr=curr->next; //Lx
   if (!curr->q.isFull()) { //Lx
      curr->q.enqueue(val); //Lx
      return true; //Lx
   } else { //Lx
      return false; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx

}
```