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

Klausur
Grundgebiete der Informatik I,II

(DPO98)

Datum: 12. 08. 2003

Teil B: Algorithmen und Programmiertechniken

Name: _____
(in Druckschrift)

Matr. Nr.: _____

Unterschrift _____

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

Aufgabe B.1 Wissensfragen *(10 Punkte)*

Beantworten Sie die folgenden Wissensfragen kurz und bündig:

- Die Syntax von Programmiersprachen wird auf verschiedenen Ebenen definiert. Welche Ebenen für die Syntaxbeschreibung kennen Sie? *(1 Punkt)*

  Antwort:

- In der Vorlesung haben Sie gelernt, dass Zeiger ein unsicheres Konstrukt sind und viele Gefahren bergen. Geben Sie mindestens vier dieser Gefahren an. *(2 Punkte)*

  Antwort:

- Beschreiben Sie den Unterschied zwischen den Parameterübergabemechanismen Call-by-Value und Call-by-Reference. *(1 Punkt)*

  Antwort:

- Erläutern Sie, in welchen Blöcken eine Variable sichtbar ist. *(1 Punkt)*

  Antwort:

- Geben Sie zwei Gründe für die Verwendung von Modulen mit sauberer Schnittstellendefinition an. *(2 Punkte)*

  Antwort:

- Welcher grundlegende Unterschied besteht zwischen einem abstrakten Datentypmodul und einem abstrakten Datenobjektmodul? *(1 Punkt)*

  Antwort:

- Erläutern Sie, was man unter dem Datenabstraktionsprinzip versteht. *(1 Punkt)*

  Antwort:

- Ein dynamisches Feld soll mittels einer Liste realisiert werden. Es gibt nur einen Zeiger auf das erste Element der Liste. Wie aufwändig ist im Allgemeinen das Finden eines Elementes in einer Liste mit n Elementen (Antwort in O-Notation)? *(1 Punkt)*

  Antwort:

Aufgabe B.2 C++-Syntax und -Semantik *(10 Punkte)*

(a) Die kontextfreie Syntax von Programmiersprachen wird in der Regel durch eine EBNF dargestellt. Im folgenden wird ein Ausschnitt der EBNF von C++ betrachtet (vereinfacht). Geben Sie zu der folgenden umgangssprachlichen Beschreibung die passende EBNF an. *(4 Punkte)*

> Mehrere Anweisungen (statement) werden zu einer Liste zusammengefasst (stmt_list), die mindestens eine Anweisung enthält. Mehrere Anweisungen werden durch “;” voneinander getrennt. Alle Anweisungen der Liste sind durch “{” “}” eingeschlossen.
>
> Eine Anweisung kann unter anderem eine Zuweisung (assignment), eine zweiseitig bedingte Anweisung (if_statement) oder eine for-Schleife (for_loop) sein.
>
> Die zweiseitig bedingte Anweisung wird durch das Schlüsselwort if eingeleitet und verzweigt abhängig von einer, in Klammern stehenden, Bedingung (condition) in zwei Teile, die jeweils aus einer Liste von Anweisungen bestehen und durch das Schlüsselwort else voneinander getrennt sind.

Hinweis: Die EBNF für die Zuweisung, die for-Schleife und die Bedingungen können Sie als gegeben ansehen. Sie müssen also nur die EBNF für statement, stmt_list und if_statement angeben! Kennzeichnen Sie Terminal-Symbole!

(b) Untersuchen Sie die folgenden Quelltextfragmente auf Fehler; geben Sie eine kurze Erläuterung zu den Fehlern an und ordnen Sie den Fehler in die Kategorien lexikalische, kontextfreie und kontextsensitive Syntax ein.

Die Verwendung von cout ohne Import ist kein Fehler! *(4 Punkte)*

1.
```
for (int i = 0; i < 100; i++) {
  // ....
}
cout << "i = " << i;
```

2.
```
bool test = true && false;
if test {
  // ....
} else {
  // ....
}
```

3.
```
void sort(int feld[], int start, int ende) {
  // ...
}

int main() {
  char feld[] = { 'a', 'c' };
  sort(feld, 0, 2);
}
```

4.
```
  char feld[] = { 'a', 'c', 'a'a, '0', 'b' };
```

(c) Spielen Sie den unten angegebenen Algorithmus durch und notieren Sie in der danebenstehenden Tabelle die Werte der einzelnen Variablen für vier Schleifendurchläufe an der Stelle (*). *(2 Punkte)*

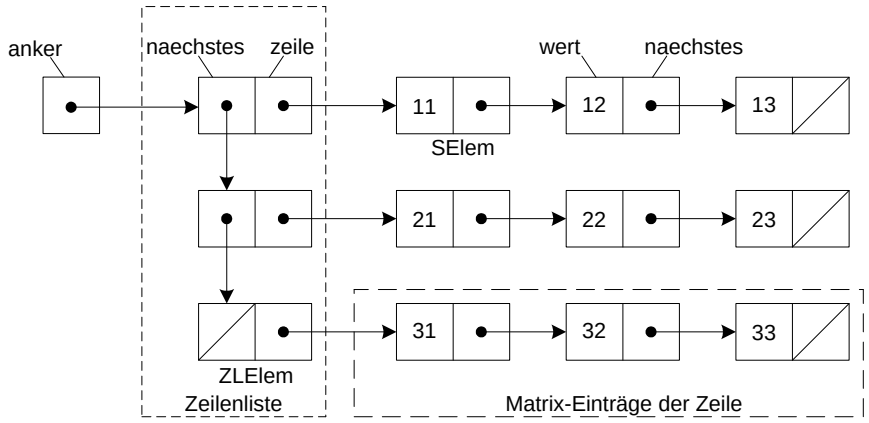
```
int x = 1, y = 4;
bool ende = true;

do {
   x = y*2;
   y = x;
   x++;
   ende = (x > 100) || (y > 200);
   // (*)
} while (!ende);
```

Durchlauf	x	y	ende
1			
2			
3			
4			

Aufgabe B.3 Implementierung des ADT Matrix *(20 Punkte)*

In dieser Aufgabe soll ein abstraktes Datentypmodul zur Speicherung von Matrizen als Klasse implementiert werden. Im Gegensatz zu den gängigen Feldimplementierungen soll es möglich sein, einer Matrix zur Laufzeit dynamisch Zeilen und Spalten hinzuzufügen. Daher soll die Matrix mit Hilfe der in der Abbildung dargestellten Zeigerstruktur realisiert werden. In einer verketteten Liste (Zeilenliste) werden die Zeilen der Matrix abgelegt, und für jede Zeile wird eine verkettete Liste der Werte in dieser Zeile verwaltet. Die Werte sollen vom Typ Integer (`int`) sein. Die Abbildung zeigt die resultierende Struktur für die angegebene Beispielmatrix.



$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}$$

Schnittstelle der Klasse:

```
class Matrix {
public:
   // Konstruktoren
   CMatrix();
   CMatrix(int zeilen, int spalten);

   // Dynamische Erweiterung
   void NeueSpalteHintenAnfuegen();
   void NeueZeileUntenAnfuegen();

   // Ausgabe auf cout (Standardausgabe)
   void ZeilenweiseAusgeben();

   // Einträge setzen/lesen
   bool SetzeWertInZelle(int zeile, int spalte, int neuerWert);
   bool GibWertInZelle(int zeile, int spalte, int& wert);
private:
   ...
}
```

(a) Geben Sie die benötigten Datenstrukturen im privaten Teil der Schnittstelle an. Verwenden Sie die in der Abbildung angegebenen Bezeichnungen (**ZLElem** für die Einträge der Zeilenliste, **SElem** für die Einträge in der Liste der Matrixwerte). *(4 Punkte)*

```
class Matrix {
...
private:
   struct SElem {









   };

   struct ZLElem {









   };

   ZLElem *anker;
};
```

(b) Implementieren Sie die Prozedur NeueSpalteHintenAnfuegen. Diese Prozedur erweitert die Matrix um eine weitere Spalte, die rechts hinten angefügt wird. Das heißt, es muss an jede Zeile ein weiteres Element angehängt werden. Beachten Sie, dass die angefügten Elemente initialisiert werden müssen. Als Ausgangswert können sie 0 annehmen. *(6 Punkte)*

```
void Matrix::NeueSpalteHintenAnfuegen() {
   ZLElem *aktZeile=anker;
   // fuer jede Zeile
   while (aktZeile!=NULL) {
      SElem *aktElem=aktZeile->zeile;
      if (aktElem==NULL) {
         // falls noch keine Spalte existiert
         // ... (braucht nicht implementiert zu werden)
     } else {
         // laufe bis zum Ende









         // neues Element anhaengen









         // neues Element initialisieren









      }
      // weiter in der naechsten Zeile




   } // end while
}
```

(c) Geben Sie eine Prozedur an, die zeilenweise alle Matrixelemente auf der Standardausgabe ausgibt. Nutzen Sie für die Ausgabe cout. Die einzelnen Werte der Matrix sollen durch Leerzeichen getrennt werden und die Matrix-Zeilen jeweils in einer separaten Zeile ausgegeben werden. (Implementieren Sie einen Durchlauf durch die Matrix, benutzen Sie dabei *nicht* die Funktion GibWertInZelle.) *(5 Punkte)*

```
void Matrix::ZeilenweiseAusgeben() {
   ZLElem *aktZeile=anker;
   // fuer jede Zeile
   while (aktZeile!=NULL) {




      // alle Elemente ausgeben









      // weiter in der naechsten Zeile




   }
}
```

(d) (1.) Warum ist es aufwändiger, alle Zellen spaltenweise auszugeben? (2.) Wie kann die Datenstruktur erweitert werden, um das spaltenweise Durchlaufen zu vereinfachen (ohne das zeilenweise Durchlaufen zu erschweren)? (3.) Welcher zusätzliche Aufwand an anderer Stelle entsteht dadurch? *(3 Punkte)*

1.

2.

3.

(e) Bei Matrizen, die nicht in allen Zellen einen Wert enthalten bzw. den selben Wert enthalten, kann Speicherplatz eingespart werden, indem nur die belegten Zellen gespeichert werden. In dieser Aufgabe soll die bisherige Implementierung so umgestellt werden, dass zwar nach wie vor alle Zeilen gespeichert werden, aber innerhalb der Zeilen nur die belegten Zellen.

Dazu muss bei jedem Matrixelement die Spaltenummer mit abgespeichert werden (spalte). Zusätzlich wird ein Wert benötigt, der die Spaltenanzahl der Matrix angibt (maxSpalte).

Ändern Sie die Datenstruktur aus (a) entsprechend ab. (Sie brauchen nur die zusätzlichen Elemente anzugeben.) *(2 Punkte)*

```
class Matrix {
...
private:
   struct SElem {









   };

   struct ZLElem {









   };

   ZLElem *anker;

};
```

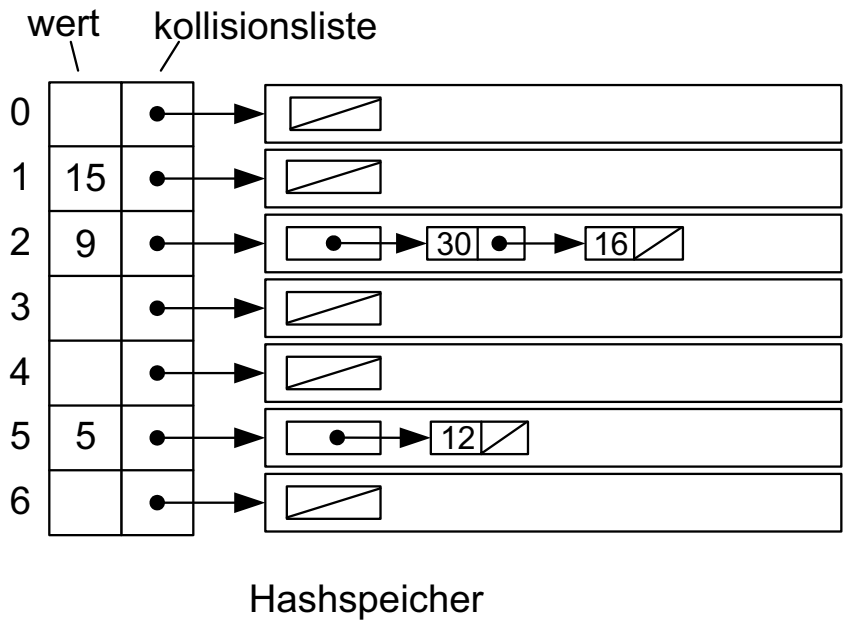
Aufgabe B.4 Mengenrealisierung mit Hashing *(20 Punkte)*

Ziel dieser Aufgabe ist die Diskussion der Realisierung eines abstrakten Datentyps `IntMenge` zur Verwaltung einer Menge von ganzen, nicht-negativen Zahlen.

Die zu speichernden Mengenelemente werden in einer Tabelle mit sieben Plätzen verwaltet. Für eine einzufügende Zahl x ermittelt die Hashfunktion `h(x)=x mod 7` (ganzzahliger Rest bei Division durch 7) den Index des Tabellenplatzes, an dem x abgelegt wird. Falls dieser Tabellenplatz bereits belegt ist, wird das einzufügende Element am Ende einer verketteten *Kollisionsliste* gespeichert, die für jeden Tabellenplatz separat verwaltet wird.

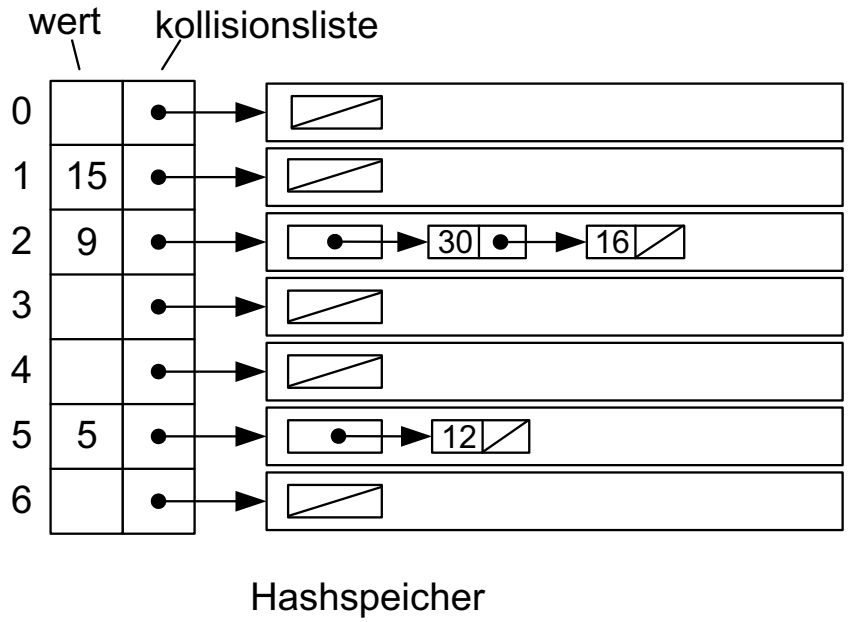
Bei der Suche nach einer Zahl x in der Menge wird zunächst mit der Hashfunktion `h` der Tabellenplatz bestimmt, an dem x abgelegt sein kann. Falls die gesuchte Zahl nicht auf diesem Tabellenplatz gespeichert ist, wird die Suche in der zugehörigen Kollisionsliste fortgesetzt.

Die folgende Abbildung zeigt die Belegung des Hashspeichers mit sechs Werten. Einige Stellen in der Tabelle sind frei, andere mit genau einem Element belegt, weitere schließlich auf Tabelle und entsprechende Kollisionsliste verteilt.



Wichtig: Mehrfaches Einfügen desselben Elements in eine Menge ist verboten! Ein unbesetzter Platz im Hashspeicher wird durch eine negative Zahl markiert (deshalb sind nur nicht-negative Zahlen als Mengenelemente zulässig).

(a) Fügen Sie in der folgenden Abbildung nacheinander folgende Zahlen in den Hashspeicher: 13, 26, 8, 17, 5, 24 *(3 Punkte)*



(b) Eine weitere Frage vorab: welche alternative Möglichkeit kennen Sie aus der Vorlesung, um Mengen zu realisieren? Erläutern Sie kurz ihre Antwort. *(1 Punkt)*

Die Schnittstelle des zu realisierenden abstrakten Datentyps IntMenge lautet:

```
class IntMenge {

public:
   // Konstruktor, erzeugt eine leere Menge
   IntMenge();

   // Fuegt eine Zahl x in die Menge ein
   // Eine bereits vorhandene Zahl wird ignoriert.
   void Einfuegen (int x);

   // TRUE, wenn Zahl x in der Menge enthalten ist
   bool IstVorhanden (int x);

private:
   const static int BEHAELTER_ANZAHL= 7;
   struct ElementTyp {
      int wert;
      IntListe* kollisionsliste;
   };
   ElementTyp behaelterFeld[BEHAELTER_ANZAHL-1];

   // Abbildungsfunktion h(x) = x mod BEHAELTER_ANZAHL
   int h(int x);
}
```

Die Realisierung des ADT IntMenge benutzt für die Kollisionslisten den unten angegebenen abstrakten Datentyp IntListe. Der ADT IntListe realisiert unsortierte Listen für natürliche Zahlen und benutzt dazu intern eine verkettete Liste. Dies ist jedoch an der Schnittstelle nicht zu erkennen:

```
class IntListe {

public:
  // Konstruktor
  IntListe();

  // Haengt Zahl x an das Ende der Liste an
  // Mehrfaches Anhaengen einer Zahl ist erlaubt
  void HintenAnhaengen (int x);

  // TRUE, wenn Zahl x in der Liste enthalten ist
  bool IstVorhanden(int x);
private:
  ...
}
```

(c) Implementieren Sie den Konstruktor IntMenge::IntMenge(). Für jeden Platz der Tabelle wird zum einen der Tabellenplatz mit der negativen Zahl -1 belegt, und zum anderen wird die zugehörige Kollisionsliste als leere Liste initialisiert. *(3 Punkte)*

```
IntMenge::IntMenge() {




}
```

(d) Implementieren Sie die Operation IntMenge::IstVorhanden(int x). Dabei wird zunächst mittels der Hashfunktion h der richtige Tabellenplatz für die Zahl x bestimmt und dieser Platz bzw. die zugehörige Kollisionsliste überprüft, ob x dort abgelegt ist. *(3 Punkte)*

```
bool IntMenge::IstVorhanden(int x) {




}
```

(e) Implementieren Sie die Operation IntMenge::Einfuegen(int x). Falls x nicht negativ ist und x noch nicht in der Menge vorhanden ist (benutzen Sie dafür IntMenge::IstVorhanden(int x)), wird mittels h der richtige Tabellenplatz bestimmt und an diesem Platz abgelegt *oder* bei einer Kollision am Ende der zugehörigen Kollisionsliste abgelegt (IntListe::HintenAnhaengen(int x)). *(7 Punkte)*

```
void IntMenge::Einfuegen(int x) {




}
```

(f) Bisher stützt sich die Realisierung der Menge auf unsortierte Listen ab. Beschreiben Sie kurz, welche Änderungen sich beim Einfügen, Suchen und Löschen ergeben, wenn stattdessen sortierte Listen eingesetzt werden? (grobe Erläuterung genügt, keinen Quellcode angeben) *(3 Punkte)*

- Änderung beim Einfügen:

- Änderung beim Suchen:

- Änderung beim Löschen: