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

# Klausur
# Grundgebiete der Informatik II

**(DPO04)**

Datum: 05. 08. 2004

# 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 | | | | |
| $\Sigma$ | 60 | | | | |

## Aufgabe II.1 Wissensfragen *(10 Punkte)*

Kreuzen Sie bei den folgenden Aufgaben die jeweils richtige Aussage an, bzw. beantworten Sie die Fragen in einem Satz.

Das Ankreuzen falscher Aussagen führt zu Abzügen entsprechend der Punktezahl der betreffenden Frage. Wenn Sie keine der Aussagen ankreuzen, wird die entsprechende Frage mit 0 Punkten gewertet. Insgesamt können 0 Punkte für die Aufgabe II.1 nicht unterschritten werden.

- Zählen Sie die Gestaltungshilfsmittel bei der Syntaxfestlegung mittels ENBF auf. *(1 Punkt)*

  Sequenz, Option, Wiederholung, Gruppierung, Fallunterscheidung, Rekursion

- Welche Kombination von Referenz und Zeiger ist zulässig? *(1 Punkt)*

  - ☐ Zeiger auf Referenz
  - ☐ Referenz auf Referenz
  - ☒ Referenz auf Zeiger

- Wie ist die Lebensdauer eines lokalen Objektes definiert? *(1 Punkt)*

  - ☒ Nur innerhalb des definierenden Blocks
  - ☐ Innerhalb und außerhalb des definierenden Blocks
  - ☐ Nur außerhalb des definierenden Blocks

- Beschreiben Sie kurz den Unterschied zwischen White-Box und Black-Box Tests. *(1 Punkt)*

  Black-Box prüft nur vorher definiertes Außenverhalten. White-Box kann einzelne Ausführungspfade testen.

- Welchen Aufwand besitzt der Algorithmus Quicksort im schlechtesten Fall? *(1 Punkt)*

  - ☐ $O(n^2 \log n)$
  - ☒ $O(n^2)$
  - ☐ $O(n \log n)$
  - ☐ $O(n^3)$

- Welcher Unterschied besteht zwischen einem ADO und einem ADT? *(1 Punkt)*

  ADO: Nur eine Instanz; ADT: Beliebig viele Instanzen

- Was unterscheidet einen Stack von einer allgemeinen Liste? *(1 Punkt)*

  Bei einem Stack wird nur an einer Seite eingefügt und entfernt. LiFo-Prinzip.

- Wieso ist es sinnvoll n-äre Bäume mit Hilfe der Left-Most-Child Right-Sibling Methode auf binäre Bäume abzustützen? *(1 Punkt)*

  Diese Realisierung benötigt weniger Speicherplatz für die NULL-Zeiger.

- Welchen Aufwand haben die üblichen Operationen (Suchen, Einfügen, Löschen) auf einem ausgeglichenen binären Suchbaum im Allgemeinen? *(1 Punkt)*

  - ☒ $O(\log n)$
  - ☐ $O(n)$
  - ☐ $O(2n)$
  - ☐ $O(n \log n)$

- Welche Gefahren bringt die unreflektierte Nutzung der objektorientierten Programmierung mit sich? *(1 Punkt)*

  Klassen für alles; Wilde Beziehungen zwischen Klassen; unklare Ausführungsstrukturen durch dynamisches Binden

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

(a) Im folgenden Programm sind insgesamt 4 Syntaxfehler versteckt. Finden Sie diese und ordnen Sie sie in die Kategorien kontextfreie und kontextsensitive Fehler ein. Tragen Sie Ihre Lösung in die nachfolgende Tabelle ein. Geben Sie für jeden Fehler die Zeilennummer, eine kurze Beschreibung des Fehlers und seine Kategorie an. *(4 Punkte)*

```
1   #include <iostream>
2
3   typedef char FeldTyp[100];
4
5   void SortStraightSelection(FeldTyp &feld, int start, int ende) {
6     char elem;
7
8     for (int i = start; i <= ende - 1; i++) {
9       k = i;
10      elem = feld[i];
11      for (int j = i + 1; j <= ende, j++) {
12        if (feld[j] == elem) {
13          k = j;
14          elem = feld[j];
15        }
16      }
17      feld[k] = feld[i];
18      feld[i] = elem
19    }
20  }
21
22  int main()  {
23    FeldTyp feld ={ 'g', 'a', 'i', 'h', 'n', 'l', 'm', 'x' };
24    SortStraightSelection(feld, 0);
25    for (int i = 0; i < 8; i++) {
26      std::cout << feld[i];
27    }
28    return 0;
29  }
```

| Zeile | Beschreibung | ks/kf |
|---|---|---|
| 9 | k nicht deklariert. | ks |
| 11 | Syntax der for-Anweisung inkorrekt. ; fehlt. | kf |
| 18 | ; am Ende vergessen. | kf |
| 24 | Parameter fehlt. | ks |

(b) Spielen Sie den unten angegebenen Algorithmus durch und notieren Sie in der daneben stehenden Tabelle die Werte der Variablen $x$, $y$ und $z$ nach dem Funktionsaufruf (1) bzw. (2). Überlegen Sie dabei, welches func nach der Wertzuweisung a = b an der Stelle (*) zuständig ist. *(6 Punkte)*

```
class A {
public:
  virtual void func(int &x, int &y, int z) {
      x = x + y;
      y = y - 2;
      z++;
   }
};
class B : public A {
public:
  virtual void func(int &x, int &y, int z) {
      x = x - y;
      y = y * 2;
      z++;
   }
};
int main() {
   A *a = new A();
   B *b = new B();
   int x = 6, y = 3, z = 2;
   a->func(x, y, z);   // (1)
   a = b;              // (*)
   a->func(x, y, 4);   // (2)
}
```

| x | y | z | |
|---|---|---|---|
| 6 | 3 | 2 | Anfangswerte |
| 9 | 1 | 2 | Wert nach Aufruf (1) |
| 8 | 2 | 2 | Wert nach Aufruf (2) |

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

In dieser Aufgabe soll ein ADT implementiert werden, der gerichtete Graphen kapselt. Die einzelnen Knoten werden über Bezeichnungen vom Typ **int** identifiziert. Im Folgenden ist ein Auszug aus der Schnittstelle des ADT dargestellt:

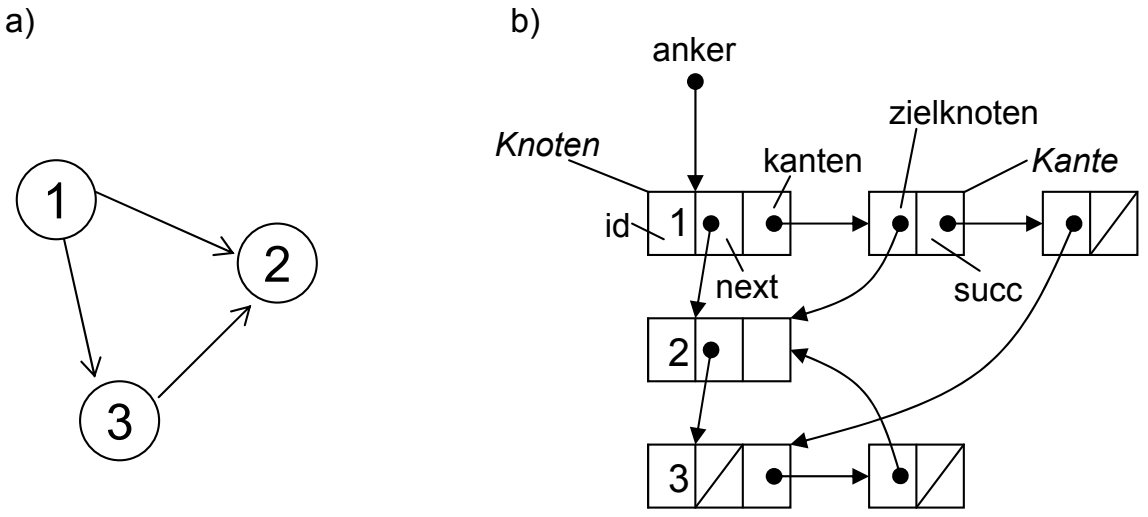
```
class ADTGraph
{
public:
   // Konstruktor
   ADTGraph();

   // Fügt einen Knoten mit der Id id hinzu.
   void addNode(int id);

   // Trägt eine Kante vom Knoten from
   // zum Knoten to ein. Rückgabewert ist
   // true, falls die Kante erzeugt wurde, sonst false.
   bool addEdge(int from, int to);

   //...
}
```

Intern soll die Speicherung der Graphen kantenorientiert erfolgen. Dies verdeutlicht die folgende Abbildung. Sie zeigt einen Beispiel-Graphen (a) und die entsprechende Darstellung in der Datenstruktur des ADT (b): Der Ankerzeiger **anker** verweist auf das erste Element einer Liste aller Knoten. Die Einträge dieser Liste sind vom Typ **Knoten** und speichern die Bezeichnung des Knotens (**id**) sowie einen Zeiger (**kanten**) auf die Liste seiner auslaufenden Kanten. Über ein Kantenelement (Typ **Kante**) erreicht man den Nachfolgerknoten mittels Zeiger **zielknoten**. Die Elemente der Knoten- und Kantenlisten sind mit den Zeigern **next** bzw. **succ** verkettet.



Die zugehörigen Deklarationen im privaten Teil der Klassendeklaration des ADT Graph lauten wie folgt:

```
struct Knoten;
struct Kante {
   Knoten* zielknoten;
   Kante* succ;
};

struct Knoten {
   int id;
   Kante* kanten;
   Knoten* next;
};

Knoten* anker;
```

(a) Zunächst soll die Anwendung des ADT verdeutlicht werden. Geben Sie ein Codefragment an, das eine Instanz des ADT erzeugt und den Graphen aus der Abbildung a) aufbaut. Verwenden Sie die Schnittstellenoperationen, um die Knoten und Kanten zu erzeugen. *(4 Punkte)*

```
ADTGraph graph; //Lx
graph.addNode(1); //Lx
graph.addNode(2); //Lx
graph.addNode(3); //Lx
graph.addEdge(1,2); //Lx
graph.addEdge(1,3); //Lx
graph.addEdge(3,2); //Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
```

(b) Implementieren Sie den Konstruktor des ADT ADTGraph. Die Datenstruktur muss geeignet initialisiert werden. *(2 Punkte)*

```
ADTGraph::ADTGraph() {
   anker=NULL; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(c) Realisieren Sie die Funktion addNode, die einen neuen Knoten anlegt. Der Knoten erhält den im Parameter id übergebenen Wert als Bezeichnung. Er soll der Einfachheit halber am Anfang der Liste der Knoten eingefügt werden. Achten Sie darauf, den neuen Knoten geeignet zu initialisieren. Gehen Sie zur Vereinfachung davon aus, dass der Knoten noch nicht existiert. Dies muss auch nicht geprüft werden. *(5 Punkte)*

```
void ADTGraph::addNode(int id) {
   Knoten *n=new Knoten; //Lx
   n->id=id; //Lx
   n->kanten=NULL; //Lx
   n->next=anker; //Lx

   anker=n; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(d) Implementieren Sie die Funktion addEdge. Diese soll eine Kante vom Knoten mit der Bezeichnung from zum Knoten mit der Bezeichnung to eintragen. Dazu soll zunächst in einer Schleife die Knotenliste sequentiell durchlaufen werden, wobei die Knoten mit den entsprechenden Bezeichnungen gesucht werden. Wenn nicht beide Knoten gefunden wurden, kann die Kante nicht erzeugt werden. Die Funktion gibt dann false zurück. Sind die Knoten vorhanden, ist für den from-Knoten die Liste der Kanten zu aktualisieren und es soll true zurückgegeben werden. Dabei kann das Kantenelement am Anfang der Kantenliste eingefügt werden. Gehen Sie davon aus, dass die Kante noch nicht im Graphen enthalten ist. Es muss also keine entsprechende Prüfung implementiert werden. *(9 Punkte)*

```
bool ADTGraph::addEdge(int from, int to) {
   Knoten *cursor=anker; //Lx
   Knoten *fromNode=NULL,*toNode=NULL; //Lx
   while (cursor!=NULL) { //Lx
      if (cursor->id==from) fromNode=cursor; //Lx
      if (cursor->id==to) toNode=cursor; //Lx
      cursor=cursor->next; //Lx
   }//Lx
   if (fromNode==NULL || toNode==NULL) return false; //Lx

   Kante *newEdge=new Kante; //Lx
   newEdge->zielknoten=toNode; //Lx
   newEdge->succ=fromNode->kanten; //Lx
   fromNode->kanten=newEdge; //Lx

   return true; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

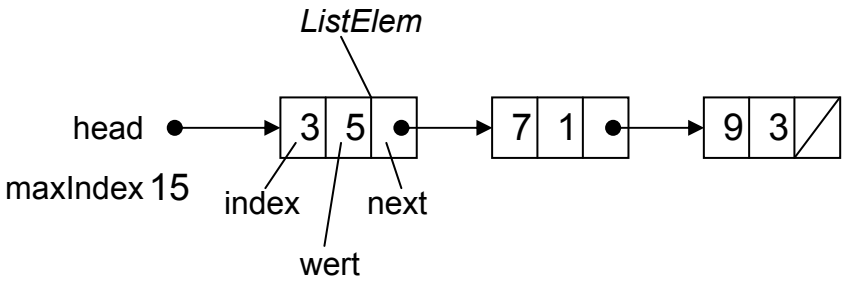
## Aufgabe II.4 Realisierung dünnbesetzter Matrizen *(20 Punkte)*

Für die Realisierung von dünnbesetzten Matrizen — also Matrizen mit vielen 0 Einträgen — sind normale Matrizen (also Felder von Feldern) ungeeignet, da sie zuviel Speicherplatz verschenken. An ihrer Stelle können “dynamische Felder” für die Zeilen als verkettete Listen auf der Halde verwendet werden, in denen nur die Einträge belegt sind, die auch tatsächlich benötigt werden.

---

Die Klasse DynVector realisiert ein solches “dynamisches Feld”. Mit Hilfe dieser Klasse kann zunächst ein Feld beliebiger Größe erzeugt werden, wobei jedoch nur die tatsächlich verwendeten Einträge sortiert nach Indizes in der Liste stehen. Bei Bedarf kann die Größe des Feldes angepasst werden.

Für ein Feld mit oberem Index 15, bei dem nur für die Indizes 3, 7 und 9 Werte existieren, ergibt sich:




```
class DynVector{
public:
    // Initialisiert den Vektor mit einer bestimmten maximalen Größe.
    DynVector(int newSize);

    // Setze Größe des Feldes auf "newSize" und erweitert das Feld.
    void modifySize(int newSize);

    // Liefert "true" zurück, wenn der Index unter dem maximal erlaubten
    // liegt. Setzt den Wert "wert" an der Stelle "index" und überschreibt
    // dabei ggf. einen Wert.
    bool set(int index, int wert);

    // Liest den Wert an der Stelle "index" und gibt den Wert in der Variablen
    // "wert" zurück. Liefert "true" zurück, falls der Wert zuvor gesetzt war,
    // sonst "false".
    bool at(int index, int &wert);
private:
    // ... private Deklarationen ...
};
```

Im folgenden sollen zunächst einige Realisierungsaspekte der Klasse DynVector betrachtet werden.

(a) Deklarieren Sie den Datentyp ListElem für die Listenelemente analog zur obigen Grafik. Deklarieren Sie auch die Komponenten für die maximal erlaubte Elementanzahl und den Verweis auf die Liste. *(4 Punkte)*

```
struct ListElem {                //Lx
    int index;                   //Lx  Feld-Index
    int wert;                    //Lx
    ListElem *next;              //Lx    Verkettung
};                               //Lx
                                 //Lx
int maxIndex;                    //Lx    Index
                                 //Lx
ListElem *head;                  //Lx    Listenkopf
 //Lx
 //Lx
 //Lx
 //Lx
 //Lx
 //Lx
 //Lx
 //Lx
```

---

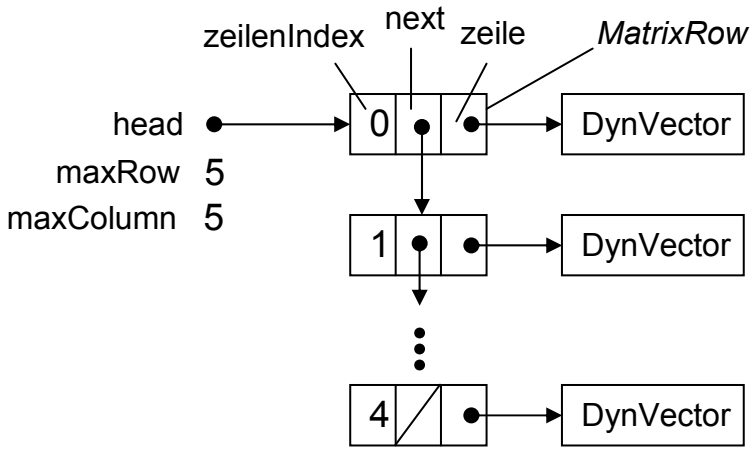
Die Klasse DynVector soll nun für die Realisierung von dünn besetzten Matrizen in der Klasse SparseMatrix verwendet werden. Die Klasse SparseMatrix hat folgende Schnittstelle:

```
class SparseMatrix {
public:
    // Konstruktor: Initialisiert Matrix.
    SparseMatrix(int maxRow, int maxColumn);

    // Setzt den Wert an der Stelle (i, j) auf "wert", falls i und j im
    // jeweils zulässigen Bereich liegen.
    bool set(int i, int j, int wert);

    // ... weitere Methoden z.B. für das Lesen des Elementes an der Stelle
    // (i,j).

private:
    // ... private Deklarationen ...
};
```




(b) Formulieren Sie den privaten Teil der Schnittstelle von SparseMatrix: Deklarieren Sie die Listenelemente MatrixRow für die Zeilen (Zeilen selbst jeweils mittels DynVector realisiert), die maximal erlaubte Zeilen- und Spaltenanzahl und den Anker für die Zeilenliste. *(5 Punkte)*

```
struct MatrixRow {          //Lx
    int zeilenIndex;        //Lx  Zeilen-Index
    MatrixRow *next;        //Lx  Verkettung
    DynVector *zeile;       //Lx  Verwendung DynVector
};                          //Lx
                            //Lx
int maxRow;                 //Lx  für die beide max. Indizes
int maxColumn;              //Lx
                            //Lx
MatrixRow *head;            //Lx  Listenkopf
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
```

(c) Realisieren Sie den Konstruktor der Klasse SparseMatrix. Der Konstruktor soll bereits die vollständige Zeilenliste anlegen, also für jeden Zeilenindex ein Listenelement. Er initialisiert auch die an dem Listenelement hängenden Felder. Beachten Sie, dass dieser Konstruktor anders ist als die üblichen Konstruktoren, da er Elemente auf der Halde anlegt.

Tipps: Achten Sie auf die korrekte Verzeigerung der Listenelemente und das Setzen des Anker-Zeigers. Verwenden Sie den Zeiger prev, den Sie stets so setzen sollten, dass er auf das zuletzt erzeugte Element zeigt. *(6 Punkte)*

```
SparseMatrix::SparseMatrix(int row, int column) {
    MatrixRow *newRow = NULL;
    MatrixRow *prev = NULL;

    maxRow = row;
    maxColumn = column;
//Lx
    for (int i = 0; i < maxRow; i++) {            //Lx
        newRow = new MatrixRow();                 //Lx
        newRow->zeilenIndex = i;                  //Lx
        newRow->zeile = new DynVector(maxColumn); //Lx
        if (prev != NULL) {                       //Lx
            prev->next = newRow;                  //Lx
        } else {                                  //Lx
            head = newRow;                        //Lx
        }                                         //Lx
        prev = newRow;                            //Lx
    }                                             //Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
}
```

(d) Realisieren Sie die Methode set der Klasse SparseMatrix.

Prüfen Sie, ob Zeilenindex i und Spaltenindex j im erlaubten Bereich liegen. Durchlaufen Sie dann die Zeilenliste, um die passende Zeile zu finden. Danach erfolgt das Finden und Setzen des Matrixelements mit Hilfe von DynVector::set.

*(5 Punkte)*

```
bool SparseMatrix::set(int i, int j, int wert) {
    // (1) Teste ob Index erlaubt.       //Lx
    if (i >= maxRow || j >= maxColumn) { //Lx
        return false;                    //Lx
    }                                    //Lx

    MatrixRow *cursor = head;            //Lx
    while (cursor->zeilenIndex != i) {   //Lx
        cursor = cursor->next;           //Lx
    }                                    //Lx
    return cursor->zeile->set(j, wert);  //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
    //Lx
}
```