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

Klausur
Grundgebiete der Informatik I,II

(DPO98/04)

Datum: 28. 07. 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 **Fragenteil** *(10 Punkte)*

(a) Welche Aspekte einer Programmiersprache werden durch die kontextsensitive Syntax beschrieben? Welche durch die kontextfreie? *(1 Punkt)*

(b) Skizzieren Sie kurz die Idee des Sortieralgorithmus “Sortieren durch Einfügen”. *(1 Punkt)*

(c) Welcher Unterschied besteht zwischen internen und externen Sortierverfahren? *(1 Punkt)*

(d) Erläutern Sie den Unterschied zwischen der Gültigkeit und Sichtbarkeit einer Variablen. *(1 Punkt)*

(e) Erläutern Sie den Unterschied zwischen einem abstrakten Datentypen (ADT) und einer Klasse in C++. *(1 Punkt)*

(f) Was unterscheidet ein abstraktes Datenobjektmodul (ADO) von einem abstrakten Datentypmodul (ADT)? *(1 Punkt)*

(g) Warum ist es sinnvoll, n-äre Bäume mit Hilfe der Leftmost-Child-Right-Sibling-Darstellung auf binäre Bäume abzubilden? *(1 Punkt)*

(h) Charakterisieren Sie das Datenabstraktionsprinzip. *(1 Punkt)*

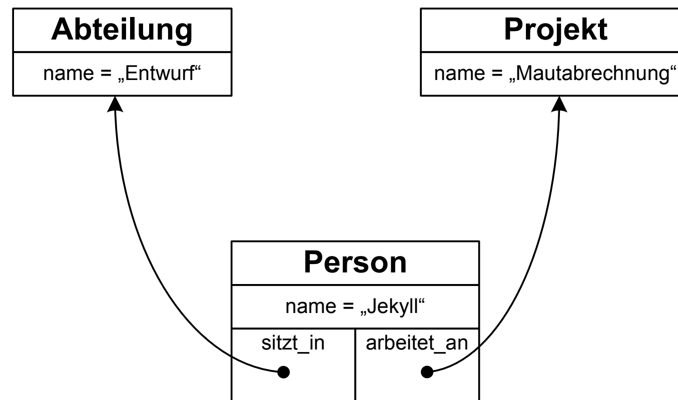
(i) Geben Sie die Komplexität von Quicksort im schlechtesten Fall an. Wodurch entsteht diese? *(1 Punkt)*

(j) Unter welchen Umständen degeneriert ein binärer Suchbaum zu einer Liste? Welche Eigenschaft von binären Suchbäumen geht hierdurch verloren? *(1 Punkt)*

Aufgabe II.2 Verbundobjekt auf der Halde *(10 Punkte)*

In dieser Aufgabe sollen Verbundtypen deklariert und Manipulationen von Verbundobjekten auf der Halde graphisch dargestellt werden. Nach Ausführung der letzten Anweisung vor den Kommentarzeilen innerhalb der main-Funktion sieht die Halde aus wie in der Abbildung dargestellt (s. nächste Seite).

```
int main()
{
   Abteilung* entwurf = new Abteilung;
   entwurf->name = "Entwurf";

   Projekt* maut = new Projekt;
   maut->name = "Mautabrechnung";

   Person* jekyll = new Person;
   jekyll->name = "Jekyll";
   jekyll->sitzt_in = entwurf;
   jekyll->arbeitet_an = maut;

   // An dieser Stelle sieht die Struktur auf ======================
   // der Halde wie in der Abbildung aus. ==========================

   Abteilung* impl = new Abteilung;
   impl->name = "Implementierung";

   Projekt* hartz = new Projekt;
   hartz->name = "Hartz IV";

   Person* hyde = jekyll;

   Person* meier = new Person;
   meier->name = "Meier";
   meier->sitzt_in = entwurf;
   meier->arbeitet_an = maut;

   hyde->arbeitet_an = hartz;

   Person* schmidt = new Person;
   schmidt->name = "Schmidt";
   schmidt->sitzt_in = impl;
   schmidt->arbeitet_an = jekyll->arbeitet_an;

   return 0;
}
```



(a) In der main-Funktion werden Verbundobjekte der Typen Abteilung, Projekt und Person angelegt und verwendet. Deklarieren Sie die Verbundtypen auf geeignete Weise. *(4 Punkte)*

(b) Zeichnen Sie, wie die Haldenstruktur direkt vor dem Verlassen der `main`-Funktion aussieht. Benutzen Sie dieselbe Notation wie in der Abbildung auf der vorhergehenden Seite und kürzen Sie dabei Namen *nicht* ab. *(6 Punkte)*

Aufgabe II.3 — ADT Liste — *(20 Punkte)*

In dieser Aufgabe soll eine geordnete Liste (ADT OrdList) mit Elementen vom Typ int mit Hilfe sequentieller Speicherung der Elemente in einem Array realisiert werden.

Der ADT hat die folgende Schnittstelle: Boolesche Rückgabewerte dienen der Erfoglskontrolle; d.h. bei Rückgabewert true war die Operation erfolgreich, sonst false.

```
class OrdList {

public:
   //Konstruktor: Initialisiert die Liste.
   OrdList(void);

   //Liefert true zurück, wenn die Liste voll ist.
   bool isFull(void);

   //Hängt den übergebenen Wert 'value' hinten an die Liste an.
   bool append(int value);

   //Navigation durch die Liste:

   //Geht zum ersten Element.
   void moveFirst(void);

   //Geht ein Element weiter.
   bool moveNext(void);

   //Gibt den Wert des aktuellen Elements, falls existent, im
   //Referenzparameter 'value' zurück, der Rückgabewert ist dann 'true'.
   bool getCurrent(int& value);

   //Fügt ein Element mit dem Wert 'value' vor dem aktuellen Element ein.
   bool insertBeforeCurrent(int value);

private:
   //... wird später ergänzt.
};
```

Intern soll die Speicherung wie in folgendem Beispiel erfolgen:

	0	1	2	3	4	
elements	1	23	42	55		...

firstFree	4	current	0

Das Array `elements` enthält die Listenelemente: `1` am Anfang der Liste, dann `23`, `42` und `55`. Die Integervariable `firstFree` gibt den Index des ersten noch nicht belegten Listenplatzes an, hier `4`. Zum Durchlauf durch die Liste muss eine aktuelle Position in Form eines Index gespeichert werden. Hier ist das erste Element das aktuelle, daher enthält die Integervariable `current` den Wert 0.

(a) Geben Sie die Definition der Datenstruktur an. Deklarieren Sie zunächst den Typ des Element-Arrays (`ElemArray`), so dass `MAX_ELEM` Werte in der Liste gespeichert werden können. Danach geben Sie die Deklaration des Array-Objekts und die übrigen Teile der Datenstruktur an. Verwenden Sie für die einzelnen Deklarationen die hier und in der Abbildung angegebenen Bezeichner. *(4 Punkte)*

```
private:

   static const int MAX_ELEM=100;
```

(b) Realisieren Sie den Konstruktor von ArrayList. Dieser soll die Datenstruktur so initialisieren, dass die Liste leer ist. Für eine leere Liste soll das erste (noch nicht vorhandene) Element das aktuelle sein. *(2 Punkte)*

```
OrdList::OrdList(void)
{

}
```

(c) Implementieren Sie die Funktion append, die ein Element mit dem Wert value am Ende der Liste anfügt, wenn diese noch nicht voll ist. Der Rückgabewert ist dann true, andernfalls false. Beachten Sie auch das Setzen von firstFree. *(4 Punkte)*

```
bool OrdList::append(int value)
{

}
```

(d) Implementieren Sie die Funktion getCurrent, die im Referenzparameter value den Wert des aktuellen Listenelements zurückgibt, wenn current den Index eines gültigen Wertes enthält. Der Rückgabewert ist dann true, andernfalls false. *(3 Punkte)*

```
bool OrdList::getCurrent(int& value)
{

}
```

(e) Hier soll die Funktion insertBeforeCurrent implementiert werden. Diese fügt, wenn noch Platz in der Liste ist, den übergebenen Wert *vor* dem aktuellen Element ein. Dazu müssen alle Elemente im Array, beginnend beim aktuellen, um eins nach hinten verschoben werden. Vergessen Sie nicht, firstFree zu setzen. Im Erfolgsfall ist der Rückgabewert der Funktion true, sonst false. *(5 Punkte)*

```
bool OrdList::insertBeforeCurrent(int value)
{

}
```

(f) Geben Sie zwei Nachteile der hier gewählten Realisierung der Datenstruktur an. *(1 Punkt)*

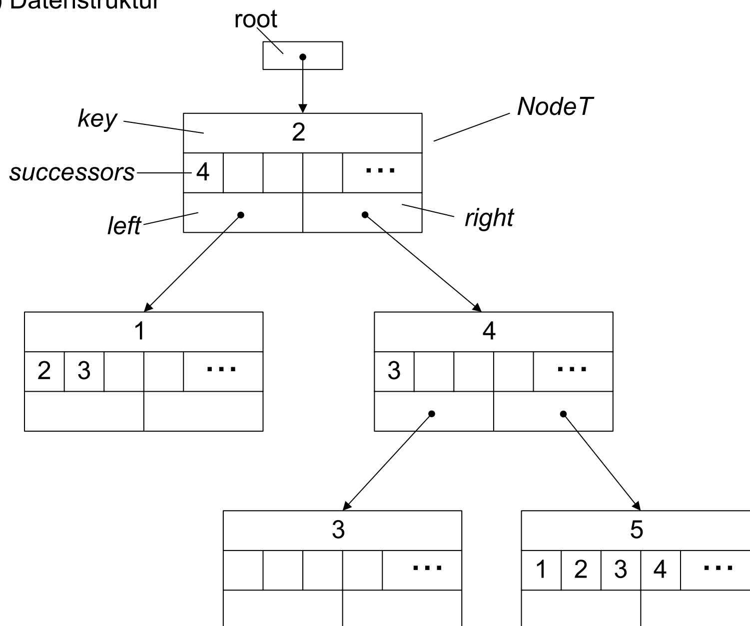
(g) Nennen Sie eine alternative Realisierung, die diese Nachteile vermeidet. *(1 Punkt)*

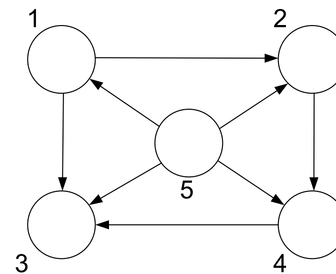
Aufgabe II.4 ADT Graph *(20 Punkte)*

Nun soll die geordnete Liste aus der vorigen Aufgabe genutzt werden, um die Nachfolger eines Knotens in einem Graphen abzulegen (adjazenzorientierte Speicherung). Um die Knoten des Graphen selbst schnell auffinden zu können, werden diese anhand ihrer Nummer in einem binären Suchbaum abgelegt. Im Folgenden wird ein Ausschnitt aus der Schnittstelle des ADT angegeben:

```
class Graph
{
public:

   // Konstruktor: Initialisiert die Datenstruktur.
   // Zur Darstellung des leeren Graphen wird die Wurzel
   // des Knoten-Baums auf NULL gesetzt.
   Graph();

   // Fügt dem Graphen einen Knoten mit der Nummer
   // 'nodeKey' hinzu.
   void addNode(int nodeKey);

   // Fügt dem Graphen eine Kante vom Knoten mit der
   // Nummer 'from' zu dem mit der Nummer 'to' hinzu.
   // Ist nur erfolgreich (Rückgabe 'true'), wenn die
   // Nachfolgerliste des Knoten nicht voll ist (sonst
   // Rückgabe 'false').
   bool addEdge(int from, int to);


   // Gibt true zurück, falls eine Kante zwischen dem
   // Knoten mit der Nummer 'from' und dem mit der
   // Nummer 'to' im Graphen existiert.
   bool hasEdge(int from, int to);

   // ... (weitere Operationen zum Zugriff auf Knoten und
   // Kanten)
private:
   //wird später ergänzt
};
```

Die folgende Abbildung verdeutlicht den internen Aufbau der Datenstruktur (a):





Die Knoten sind in einem binären Suchbaum gespeichert. Dieser ist mit Hilfe von Zeigern auf der Halde realisiert. Die Variable root zeigt auf den Wurzelknoten des Baums oder ist NULL, falls der Baum leer ist und der Graph somit keine Knoten enthält. Jeder Baumknoten hat eine Komponente key vom Typ int, die zur Speicherung der Bezeichnung des Graphknotens benutzt wird. Die Zeiger left und right verweisen auf das linke bzw. rechte Kind des Baumknotens. Jeder Baumknoten enthält eine Instanz (successors) der Liste aus der vorherigen Aufgabe, um dort die Knotennummern der Nachfolger des entsprechenden Graphknotens abzulegen. Die Knotennummern sind dort aufsteigend sortiert abgelegt.

Die im Teil a) der Abbildung dargestellte Situation entspricht dem Graphen im Teil b) der Abbildung. So hat bspw. der Knoten 2 im Graphen nur den Nachfolger 4, daher ist 4 das einzige Element in der successors-Liste des entsprechenden Baumknotens. Der Knoten 5 hingegen hat im Graphen die Nachfolger 1, 2, 3 und 4, daher sind in seinen successors die Knotennummern 1, 2, 3, 4 enthalten.

(a) Geben Sie die Definition der Datenstruktur im private-Teil der Klassendeklaration an. Definieren Sie den Typ für Baumknoten `NodeT`, sowie die Wurzel des Baumes. *(5 Punkte)*

```
//...
private:
```

(b) Implementieren Sie die lokale Hilfsfunktion findNode, die beginnend am Baumknoten current den Baumknoten mit dem key nodeKey sucht. Dabei ist die übliche Eigenschaft eines binären Suchbaums zu nutzen, dass kleinere Werte links und größere Werte rechts unterhalb eines Knotens abgespeichert werden.

Die Implementierung soll iterativ sein. Die Funktion soll NULL zurückgeben, wenn der gesuchte Knoten nicht im Baum enthalten ist, und einen Zeiger auf den Knoten, wenn er gefunden wurde. *(7 Punkte)*

```
TreeGraph::NodeT* TreeGraph::findNode(int nodeKey)
{

}
```

(c) Implementieren Sie die Funktion addEdge, die eine Kante vom Knoten mit der Nummer from zu dem Knoten mit der Nummer to in die Datenstruktur einträgt. Die Funktion ist nur erfolgreich, wenn in der Nachfolgerliste des from-Knotens noch Platz für ein weiteres Element ist. Dann wird nach dem Einfügen true zurückgegeben, sonst false.

Gehen Sie davon aus, dass beide Knoten existieren und dass noch keine Kante zwischen ihnen existiert. Beides muss also nicht geprüft werden.

Nutzen Sie zum Auffinden des Knotens mit der Nummer from die Funktion findNode aus der vorherigen Teilaufgabe. Beachten Sie, dass die Nachfolger eines Knotens aufsteigend sortiert in der Nachfolgerliste abgelegt werden müssen. Sie müssen also zunächst die richtige Einfügestelle suchen. *(8 Punkte)*

```
bool Graph::addEdge(int from, int to)
{

}
```