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

Klausur
Grundgebiete der Informatik 2

(DPO98/04)

Datum: 20. 07. 2006

Algorithmen und Programmiertechniken

Name: *(in Druckschrift)*	_____
Matr. Nr.:	_____
Unterschrift	_____

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

Aufgabe 1 Wissensfragen *(10 Punkte)*

(a) Die Syntax von Programmiersprachen wird auf verschiedenen Ebenen definiert. Welche drei Ebenen für die Syntaxbeschreibung haben Sie in der Vorlesung kennengelernt? *(1 Punkt)*

(b) Die kontextfreie Syntax von Programmiersprachen wird durch EBNF-Regeln festgelegt. Erläutern Sie kurz den Unterschied zwischen *Terminalsymbolen* und *Nichtterminalsymbolen* in EBNF-Regeln. *(1 Punkt)*

(c) Nennen Sie zwei Gründe für die Verwendung von Modulen mit Schnittstellen. *(1 Punkt)*

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

(e) Definieren Sie kurz die Sichtbarkeit von Variablen in Blöcken. *(1 Punkt)*

(f) Worin besteht der Unterschied zwischen einem Zeiger und einer Referenz? *(1 Punkt)*

(g) Was versteht man unter dem *Überladen einer Funktion*? *(1 Punkt)*

(h) Erläutern Sie kurz, welche Idee hinter der *Vererbung* steckt. *(1 Punkt)*

(i) Skizzieren Sie kurz die Idee von *Sortieren duch Auswählen.* *(1 Punkt)*

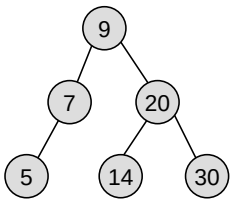
(j) Geben Sie die Komplexität von Quicksort im schlechtesten Fall an. *(1 Punkt)*

Aufgabe 2 Binäre Suchbäume *(10 Punkte)*

In der Vorlesung wurde das Konzept des *binären Suchbaums* vorgestellt. In einem binären Suchbaum werden Elemente geordnet gespeichert. Dies ermöglicht eine effiziente Suche der gespeicherten Elemente.

(a) In der folgenden Teilaufgabe sind die zu speichernden Elemente ganzzahlige Werte. Die folgenden Werte sollen der Reihe nach in einen leeren binären Suchbaum eingefügt werden: 19, 10, 25, 27, 3, 21, 2. Zeichnen Sie den resultierenden binären Suchbaum. *(3 Punkte)*

(b) Geben Sie die Werte des folgenden binären Suchbaums in *Postorder*-Reihenfolge an. *(2 Punkte)*



(c) Die folgende Prozedur soll die Ausgabe eines binären Suchbaums in Postorder-Reihenfolge in C++ *rekursiv* implementieren. Hierzu wird zunächst ein geeigneter Datentyp NodeT für die Knoten des Baums definiert, der aus drei Elementen besteht: Einem ganzzahligen Wert value, einem Zeiger left auf den linken und einem Zeiger right auf den rechten Nachfolger im Baum. Der Prozedur wird ein Zeiger root übergeben, der auf die Wurzel des auszugebenden Teilbaums verweist. Die Kommentare im Programm geben an, welche Anweisungen an die entsprechende Stelle eingefügt werden müssen. Bitte ergänzen Sie das Programm an den gekennzeichneten Stellen.

(5 Punkte)

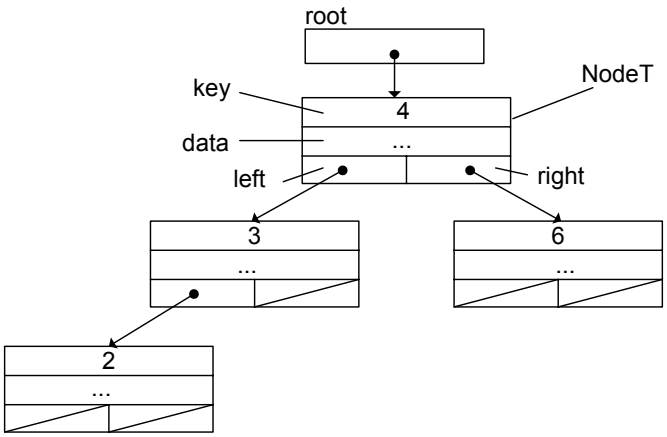
```
#include <iostream.h>
#include <stdlib.h>


//Definition des Datentyps 'NodeT'
struct NodeT{
     int value;
     NodeT *left;
     NodeT *right;
};


//Prozedur 'Postorder' zum Ausgeben des Baums
void Postorder (NodeT *root){

   //Überprüfen, ob linker Teilbaum leer ist;
   //Falls nicht, Prozedur auf linkem Teilbaum aufrufen;




   //Überprüfen, ob rechter Teilbaum leer ist;
   //Falls nicht, Prozedur auf rechtem Teilbaum aufrufen;




   //Wert des Knotens ausgeben



}
```

Aufgabe 3 ADT Map als binärer Suchbaum *(20 Punkte)*

In dieser Aufgabe soll ein ADT Map als Klasse realisiert werden, die Paare aus Schlüsseln und Werten speichert. Dabei können Werte unter einem bestimmten Schlüssel gespeichert werden und später über die Angabe des Schlüssels wieder ausgelesen werden. Die Map bietet die im public-Bereich der Klassendeklaration angegebene Schnittstelle an:

```
typedef int KeyT;
typedef int DataT;

class Map
{
public:
   Map(void);
   // initialisiert die Map

   bool Contains(KeyT nKey);
   // prüft ob das Element mit dem Schlüssel
   // 'nKey' in der Map enthalten ist

   bool Insert(KeyT nKey, DataT nData);
   // fügt den Wert 'nData' unter dem
   // Schlüssel 'nKey' in die Map ein;
   // Rückgabewert ist 'false', wenn der
   // Schlüssel schon belegt ist

   bool Get(KeyT nKey, DataT &nData);
   // Liefert den Wert zu Schlüssel
   // 'nKey' im Referenzparameter 'nData';
   // Rückgabewert ist 'false', wenn der
   // Schlüssel nicht existiert


private:
   // private Datenstruktur siehe Aufgabenteil a)
   ...
   bool findNode(KeyT nKey, NodeT* & foundNode);
   // Lokale Hilfsfunktion: Sucht den Knoten mit dem
   // Schlüssel 'nKey' im Baum. Siehe Aufgabenteil b)
};
```

Im Rumpf der Klasse soll die Map als binärer Suchbaum realisiert werden (vgl. Abbildung). Das Sortierkriterium ist der Schlüssel. Die einzelnen Knoten des Baumes bestehen aus einem Eintrag für den Schlüssel (`key`, vom Typ `keyT`), einem für den Wert (`data`, vom Typ `DataT`) sowie den Zeigern auf den linken und rechten Nachfolger. Ein Zeiger auf die Wurzel des Baumes wird in der privaten Variable `root` gespeichert. Neue Datensätze werden in den Baum entsprechend der Suchbaumeigenschaft eingefügt.



(a) Geben Sie die Deklaration der benötigten Knoten-Datenstruktur (`NodeT`) im privaten Teil der Klassendeklaration an. Verwenden Sie die Bezeichner aus der Abbildung. Deklarieren Sie auch die Wurzel des Baumes. *(3 Punkte)*

```
struct NodeT {




};
```

(b) In der Klassendeklaration ist die private Hilfsfunktion `findNode` zum Finden eines Knotens angegeben. Vervollständigen Sie die Implementierung dieser Funktion. Der Rückgabewert der Funktion soll angeben, ob ein Knoten mit dem übergebenen Schlüssel gefunden wurde (`true`) oder nicht (`false`). Wurde er gefunden, wird über den per Referenz übergebenen Zeiger `foundNode` seine Adresse zurückgegeben. Ist noch kein Knoten mit dem entsprechenden Schlüssel im Baum enthalten, erhält `node` die Adresse des Knotens, unter dem der Knoten mit dem

übergebenen Schlüssel einzufügen wäre. Ist der Baum leer, erhält foundNode den Wert NULL. Die Navigation durch den Baum ist iterativ zu implementieren.
(8 Punkte)

```
bool Map::findNode(KeyT nKey, NodeT* & foundNode) {
   foundNode=root;
   if (foundNode==NULL) {









   } else {
      while (true) {
         if (foundNode->key==nKey) {









         } else if (foundNode->key < nKey) {

















         } else {

















         }
      }
   }
}
```

(c) Implementieren Sie die Funktion **Insert** zum Einfügen eines neuen Schlüssel/Wert-Paares in den Baum. Verwenden Sie dabei die Hilfsfunktion aus Teilaufgabe b). Die bereits vorgegebenen Kommentare strukturieren die Implementierung. Bitte tragen Sie Ihre Lösung an den vorgegebenen Stellen unter dem jeweiligen Kommentar ein. *(9 Punkte)*

```
bool Map::Insert(KeyT nKey, DataT nData)
{
   // mittels 'findNode(...)' prüfen, ob der Knoten schon existiert,
   // bzw. die Einfügestelle bestimmen. Ggf. Abbruch. (Rückgabewert!)(2 P)









   // neuen Knoten erzeugen und initialisieren (3 P)









   // Spezialfall: Baum ist leer, neuen
   // Knoten entsprechend einfügen (2 P)
   if ( // unter dieser Zeile die Bedingung einsetzen!



      ) { // Knoten richtig einfügen



   } else {
      // auf der richtigen Seite als Nachkomme des
      // von 'findNode(...)' gelieferten Knotens einfügen (2 P)









   }
   return true;
}
```

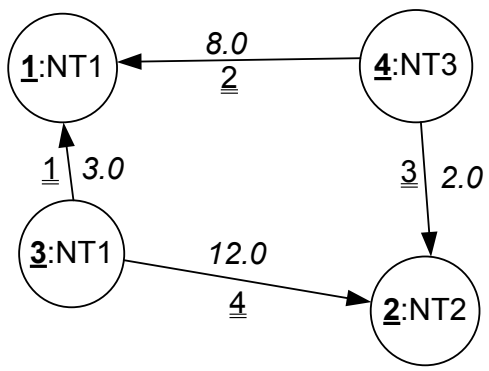
Aufgabe 4 ADT Graph *(20 Punkte)*

Ziel dieser Aufgabe ist die Realisierung eines ADT Graph zum Speichern von Graphen mit getypten Knoten und gerichteten Kanten zwischen jeweils zwei Knoten. Knoten und Kanten werden jeweils über eindeutige Bezeichner vom Typ int identifiziert. Die Bezeichner werden vom Verwender des ADT Graph vergeben. Der ADT Graph überwacht, ob die Bezeichner eindeutig gewählt wurden. Kanten haben ein Kantengewicht vom Typ double. Die Schnittstelle des ADT bietet Funktionen zum Einfügen sowie zur Existenzprüfung von Knoten und Kanten an:

```
class Graph {
public:
   Graph(void);
   // Initialisiert den Graphen.

   bool InsertNode(int nodeId, string nodeType);
   // Fügt einen neuen Knoten vom Typ 'nodeType'
   // mit dem Bezeichner 'nodeId' in den Graphen ein.
   // Rückgabewert ist 'true' bei Erfolg und 'false',
   // falls 'nodeId' schon vergeben ist.

   bool InsertEdge(int fromNode, int toNode, int edgeId, double edgeWeight);
   // Fügt eine Kante mit dem Bezeichner 'edgeId' vom Knoten 'fromNode'
   // zum Knoten 'toNode' mit dem Gewicht 'edgeWeight' ein. Rückgabewert
   // 'true' bei Erfolg und 'false' falls (mind.) einer
   // der Knoten nicht existiert oder die Kante schon existiert.

   bool ExistsNode(int nodeId);
   // Rückgabewert ist 'true', falls Knoten 'nodeId' im Graph existiert

   bool GetEdgeWeight(int edgeId, double &edgeWeight);
   // Gibt im Referenzparameter 'edgeWeight' das Gewicht
   // der Kante mit dem Bezeichner 'edgeId' zurück
   // Der Rückgabewert ist 'false', wenn die Kante nicht
   // existiert, sonst 'true'.

private:
   ...
};
```

Abbildung a) zeigt einen Beispielgraphen. Innerhalb der Knoten ist vor dem Doppelpunkt der Knotenbezeichner (einfach unterstrichen) und hinter dem Doppelpunkt der Knotentyp angegeben. Die Kantenbezeichner (doppelt unterstrichen) und die Kantengewichte (kursiv) sind an den Kanten notiert.

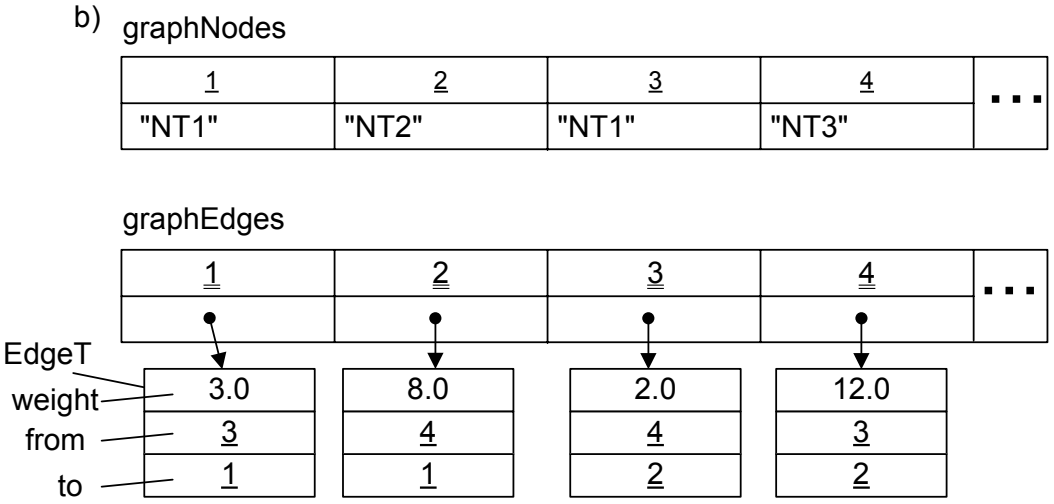
a)



(a) Die Realisierung von Graph stützt sich auf einen generischen Baustein GenMap zur Speicherung von Schlüssel/Wert-Paaren ab. Diese generische Map ist aus der speziellen Map aus der vorigen Aufgabe 3 zu entwickeln.

Die Map in Aufgabe 3 ist spezifisch für Schlüssel vom Datentyp KeyT und Daten vom Datentyp DataT realisiert. Wie muss die Klassendeklaration in der Headerdatei geändert werden, um eine generische Klasse GenMap zu definieren, die als formale generische Parameter den Schlüsseltyp und den Typ für die zu speichernden Daten erhält? Geben sie kurz die Änderung der Klassendeklaration aus Aufgabe 3 an. Verwenden Sie KeyT und DataT auch als Bezeichner für die formalen generischen Parameter. *(3 Punkte)*

Nun zur Realisierung des ADT Graph:



Die grundlegende Realisierungsidee ist die Verwendung zweier Instanzen der generischen Map aus der vorherigen Teilaufgabe (vgl. Abbildung b): Die Map graphNodes dient der Verwaltung der Knoten im Graphen. Als Schlüssel dient der Knotenbezeichner (Typ int), der Wert ist der Knotentyp als Zeichenkette (string). Die Map graphEdges enthält alle Kanten. Hier sind die Kantenbezeichner die Schlüssel (Typ int), Zeiger auf Verbunde mit weiteren Informationen (EdgeT-Zeiger) die Werte. Die Verbunde enthalten das Kantengewicht vom Typ double in der Komponente weight sowie Quellknoten und Zielknoten vom Typ int in den Komponenten from und to.

(b) Definieren Sie die interne Datenstruktur für den ADT Graph wie oben beschrieben und in Abbildung b) dargestellt. Orientieren Sie sich dabei auch an den vorgegebenen Kommentaren. *(4 Punkte)*

```
// Map-Instanz für Knoten (1 P)




// Verbundtyp für Kanten (2 P)




// Map-Instanz für Kanten (1 P)
```

(c) Implementieren Sie die Funktion **ExistsNode**, die für einen übergebenen Knotenbezeichner (**nodeId**) prüft, ob der zugehörige Knoten existiert. Hinweise: Verwenden Sie die Schnittstelle der Map aus Aufgabe 3, genauer deren generische Instanz. Die Lösung basiert auf dem Aufruf einer Schnittstellenfunktion. *(2 Punkte)*

```
bool Graph::ExistsNode(int nodeId)
{




}
```

(d) Implementieren Sie die Funktion **InsertNode**. Verwenden Sie auch hier die Schnittstelle der Map. Vergessen Sie nicht den Rückgabewert: **true** steht für erfolgreiches Einfügen, **false** steht für einen Fehler, insbesondere wenn der übergebene Knotenbezeichner (**nodeId**) bereits in Verwendung ist. *(2 Punkte)*

```
bool Graph::InsertNode(int nodeId, string nodeType)
{




}
```

(e) Implementieren Sie die Funktion InsertEdge. Verwenden Sie die Schnittstelle der Map. Prüfen Sie, ob die zu verbindenden Knoten existieren und ob noch keine Kante mit dem selben Bezeichner existiert. Sonst kann die Kante nicht erzeugt werden und der Rückgabewert der Funktion ist false. Erzeugen Sie erst nach diesen Überprüfungen einen neuen Verbund für die Kante. *(9 Punkte)*

```
bool Graph::InsertEdge(int fromNode, int toNode, int edgeId, double edgeWeight)
{









}
```