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

Klausur
Grundgebiete der Informatik 2

(DPO98/04)

Datum: 05. 09. 2006

Algorithmen und Programmiertechniken

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

Aufgabe	Max. Punkte	Korrektur		Einsichtnahme	
		Punkte	Kürzel	Punkte	Kürzel
1	10				
2	13				
3	22				
4	15				
Σ	60				

Aufgabe 1 Objektorientierung *(10 Punkte)*

(a) In folgendem Programm werden die Variablen `x` und `y` als Aktualparameter an die Prozeduren `addiere` und `subtrahiere` übergeben. Geben Sie rechts in der Tabelle die Werte von `x` und `y` nach jeder Prozedurausführung an. *(6 Punkte)*

```
class Basisklasse {
public:
   virtual void addiere(int &x, int &y) {
      x = x + y;
      y = y + 1;
   }
   void subtrahiere(int &x, int y){
      x = x - y;
      y = y - 1;
   }
};

class Kindklasse : public Basisklasse {
public:
   void addiere(int &x, int &y) {
      x = x + y +y;
      y = y + 2;
   }
   void subtrahiere(int &x, int y){
      x = x - y - y;
      y = y - 2;
   }
};

int main() {
   Basisklasse *b = new Basisklasse();
   Kindklasse *k = new Kindklasse();
   int x = 10, y = 2;
   b->addiere(x, y);   // 1. Prozeduraufruf
   b = k;
   b->addiere(x, y);   // 2. Prozeduraufruf
   x = 10;
   y = 2;
   b->subtrahiere(x, y);   // 3. Prozeduraufruf
   return 0;
}
```

	x	y
1. Prozeduraufruf		
2. Prozeduraufruf		
3. Prozeduraufruf		

(b) Tragen Sie in der Tabelle ein, welche Zuweisung in dem nachfolgenden Programm erlaubt ist oder aufgrund von Einschränkungen der Sichtbarkeiten verboten ist. *(4 Punkte)*

```
class Basisklasse {
public:
  int a;
protected:
  int b;
private:
  int c;
};

class Kindklasse : public Basisklasse {
public:
  void zuweisen(){
   this->b = 4;  // 1. Zuweisung
   this->c = 3;  // 2. Zuweisung
  }
};

int main() {
   Kindklasse *ko = new Kindklasse();
   ko->zuweisen();
   ko->a = 2;  // 3. Zuweisung
   ko->b = 1;  // 4. Zuweisung
   return 0;
}
```

	erlaubt/verboten
1. Zuweisung	
2. Zuweisung	
3. Zuweisung	
4. Zuweisung	

Aufgabe 2 Speicherung von Graphen *(13 Punkte)*

(a) In dieser Aufgabe wird ein Graph betrachtet, der das vereinfachte Busliniennetz an der *Langen Nacht der Museen in Aachen* repräsentiert. Insgesamt gibt es zwei Buslinien, die am Bushof starten und Sehenswürdigkeiten in Aachen anfahren und wieder zum Bushof zurückkehren. Dies umfasst eine Linie für den Neuen Aachener Kunstverein (NAK), das Ludwig Forum (Ludwig F.) und das Suermondt-Ludwig-Museum (S.-L.-M.). Die andere Linie befördert Besucher zu Gut Rosenberg (Gut R.) und zum Zollmuseum Friedrichs.

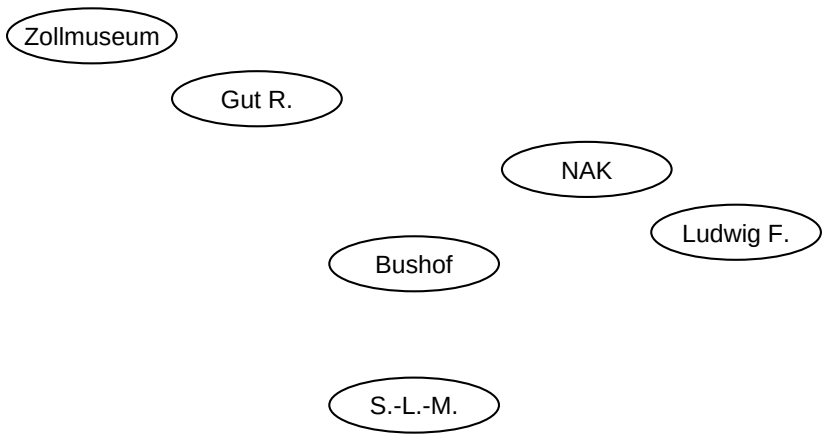
In dem Graph sind die Knoten Haltestellen und die Kanten direkte Busverbindungen. Der Graph ist gerichtet und jede Kante besitzt einen ganzzahligen Wert für die Fahrtdauer in Minuten.

Die Speicherung des Graphen erfolgt in einer Adjazenz-Matrix. Jeder ganzzahlige Wert in der Matrix steht für eine Kante. Die Zeile, in der der Wert steht, ist mit dem Quellknoten beschriftet (links), die Spalte mit dem Zielknoten (oben). Der Wert ist die Fahrtdauer. Existiert zwischen zwei Knoten keine Kante, lautet der Wert “-“.

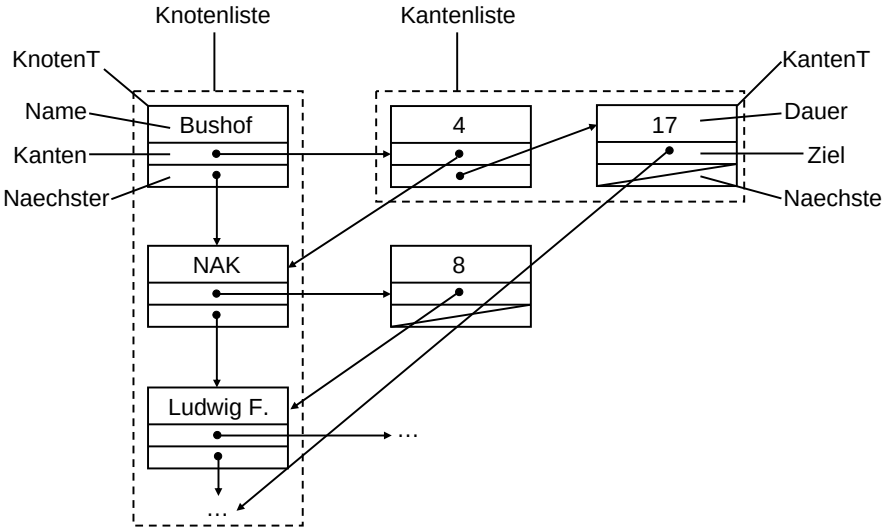
Zeichnen Sie in den unvollständigen Graphen die gerichteten Kanten mit den Werten entsprechend der Matrix ein.

(4 Punkte)

	Bushof	NAK	Ludwig F.	S.-L.-M.	Gut R.	Zollmuseum
Bushof	-	4	-	-	17	-
NAK	-	-	8	-	-	-
Ludwig F.	-	-	-	14	-	-
S.-L.-M.	10	-	-	-	-	-
Gut R.	-	-	-	-	-	8
Zollmuseum	25	-	-	-	-	-



(b) Im folgenden soll der Graph mit einer einfach verketteten Knotenliste und entsprechenden Kantenlisten gespeichert werden. Jede Kantenliste ist ebenfalls einfach verkettet und enthält alle auslaufenden Kanten zu einem Knoten des Graphen. Die Reihenfolge der Kantenelemente innerhalb der Liste spielt keine Rolle.

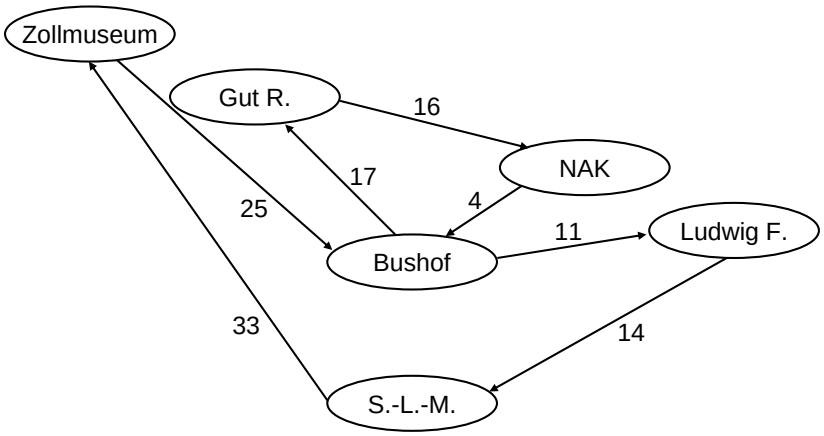


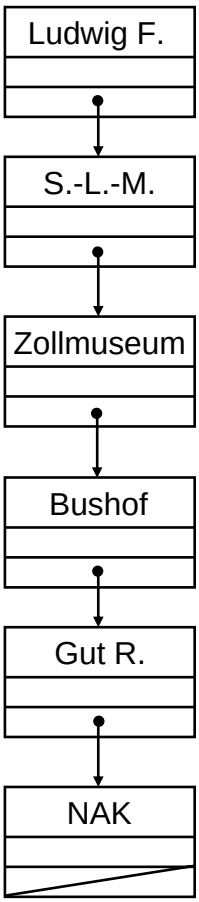
Die Listenstruktur sowie die Typen für Knoten (`KnotenT`) und Kanten (`KantenT`) entnehmen Sie bitte der obigen Zeichnung. Geben Sie für die Datenstrukturen `KnotenT` und `KantenT` die entsprechenden Deklarationen an.

(4 Punkte)

(c) Geben Sie für den folgenden Graphen die entsprechenden Knoten- und Kantenlisten gemäß der Zeichnung aus Teilaufgabe (b) an.

(5 Punkte)

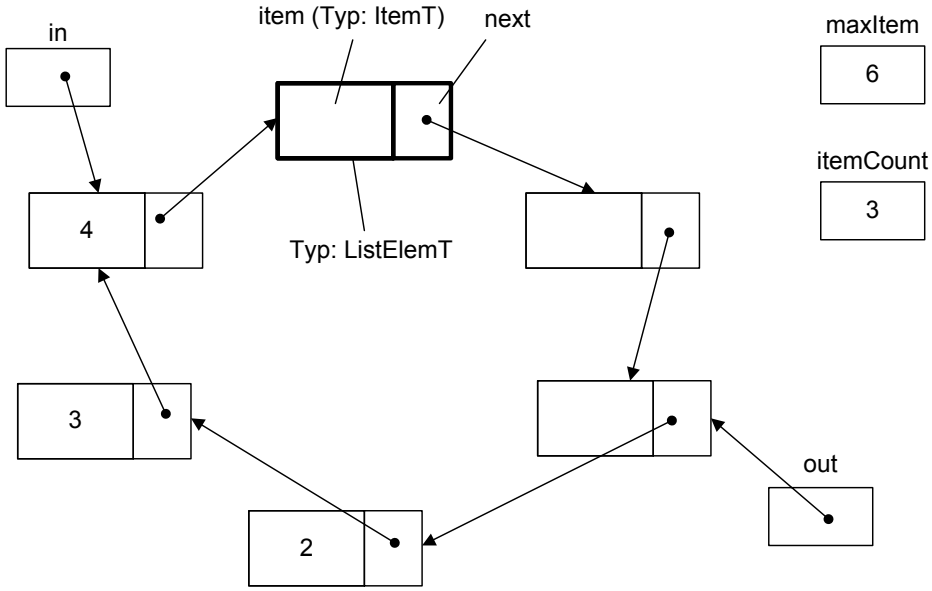




Aufgabe 3 ADT GenDynBuffer als Ringliste *(22 Punkte)*

In dieser Aufgabe soll ein ADT für einen generischen dynamischen Puffer realisiert werden. Der Puffer hat FIFO-Verhalten (first in first out), d.h. Einträge werden in der Reihenfolge ihrer Einfügung auch wieder entnommen. Generisch ist der Puffer insoweit, als dass der konkrete Eintragstyp ein generischer formaler Parameter ItemT ist. Dynamisch ist der Puffer, da auch nach Erzeugung eines Puffer-Datenobjekts die Anzahl der gleichzeitig speicherbaren Einträge noch nachträglich erhöht werden kann. Im Folgenden ist die Schnittstelle des ADT dargestellt:

```
template<class ItemT>
class GenDynBuffer {
public:
    // Erzeugt einen Puffer mit der anfänglichen Größe 'initSize'.
    GenDynBuffer(int initSize);

    // Fügt einen Eintrag in den Puffer ein. Der Verwender muss vorher
    // prüfen, ob noch Platz im Puffer ist.
    void Enqueue(ItemT newItem);

    // Entfernt den ältesten Eintrag aus dem Puffer und gibt ihn zurück.
    // Der Verwender mus vorher prüfen, ob noch mind. ein Element im Puffer ist.
    ItemT Dequeue();

    // Gibt true zurück gdw. der Puffer voll ist.
    bool IsFull();

    // Gibt true zurück gdw. der Puffer leer ist.
    bool IsEmpty();

    // Vergrößert die Zahl der gleichzeitig speicherbaren Einträge um 'addSize'.
    // 'addSize' muss mindestens '1' sein.
    void Grow(int addSize);

private:
    struct ListElemT;
    // ...
    // s. Aufgabenteil (a)

    // Erzeugt eine Liste der Größe 'size'. Zeiger 'head' zeigt nach Aufruf
    // auf den Anfang, Zeiger 'tail' auf das Ende der Liste
    // (beide Zeiger per Referenz übergeben).
    void CreateList(ListElemT* & head, ListElemT* & tail, int size);


};
```



Der Puffer ist als einfach verkettete Ringliste auf der Halde realisiert (s. Abbildung, dort werden als Daten ganze Zahlen angenommen). Die Listenelemente (Typ **ListElemT**) bestehen aus dem zu speichernden Eintrag und einem Zeiger auf das nächste Listenelement. Der Zeiger **in** verweist auf das Listenelement mit dem zuletzt dem Puffer per **Enqueue** hinzugefügten Eintrag, hier die Zahl **4**. Der nächste Wert ist im nachfolgenden Listenelement (fett) einzufügen. Der Zeiger **out** verweist auf das Listenelement mit dem Eintrag, der dem Puffer per **Dequeue** zuletzt entnommen wurde. Der als nächstes zu entnehmende Wert steht also im Nachfolger dieses Listenelements, hier die Zahl **2**. Des Weiteren wird in **maxItem** die aktuelle Größe des Puffers gespeichert und in **itemCount** die aktuelle Anzahl der im Puffer befindlichen Einträge (jeweils vom Typ **int**). Direkt nach Erzeugung eines neuen, leeren Puffers verweisen **in** und **out** auf das selbe, beliebige Listenelement.

(a) Vervollständigen Sie den privaten Teil der Schnittstelle: Geben Sie die Definition von **ListElemT** sowie von **in**, **out**, **maxItem** und **itemCount** an. *(4 Punkte)*

```
struct ListElemT {




};
```

(b) Implementieren Sie die Funktion **IsFull**. Hinweise: Die Lösung ist sehr kurz und basiert *nicht* auf Zeigervergleichen. *(2 Punkte)*

```
template <class ItemT>
bool GenDynBuffer<ItemT>::IsFull() {




}
```

(c) Implementieren Sie die Funktion **Dequeue**. Sie können sich dabei darauf verlassen, dass der Puffer vorher nicht leer ist. Hinweise: Vergessen Sie nicht, **itemCount** zu aktualisieren. Lesen Sie in der Erläuterung der Datenstruktur genau nach, wo der **out**-Zeiger vor dem Aufruf von **Dequeue** steht. *(4 Punkte)*

```
template <class ItemT>
ItemT GenDynBuffer<ItemT>::Dequeue() {









}
```

(d) Implementieren Sie die Prozedur **Grow**, die die Größe des Puffers um **intSize** erhöht. Verwenden Sie dabei die Prozedur **CreateList**. Dabei muss beachtet werden, dass der normale Ablauf von Einfügen und Auslesen nicht gestört wird. Orientieren Sie sich daher an den Kommentaren im Programmtext. *(7 Punkte)*

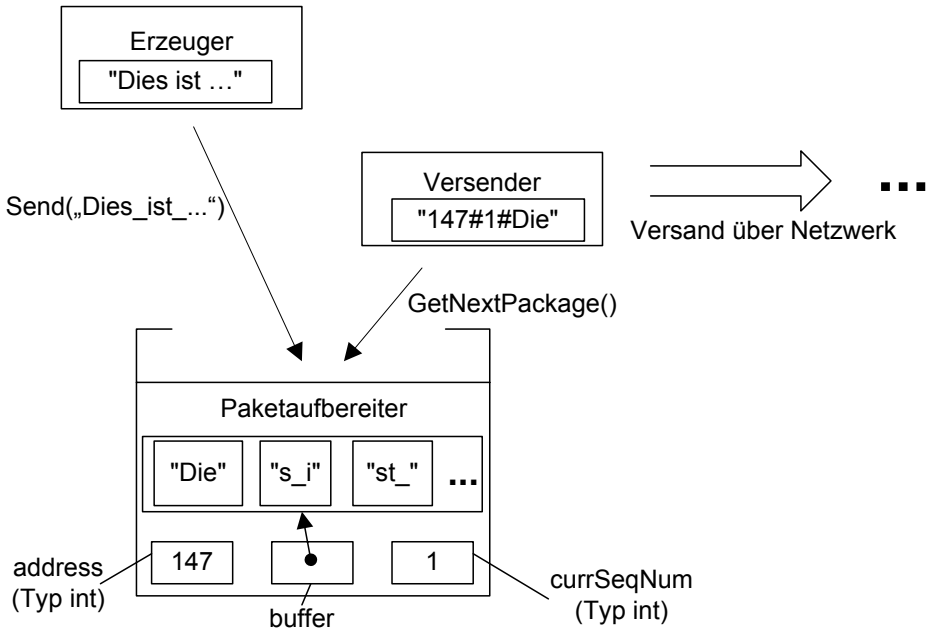
```
template <class ItemT>
void GenDynBuffer<ItemT>::Grow(int addSize) {
    // Liste richtiger Laenge erzeugen




    // Nachfolger des letzten Elements der
    // neuen Liste auf den aktuellen Nachfolger von
    // 'out' setzen



    // Erstes Element der neuen Liste als
    // Nachfolger von 'out' verzeigern



    // 'out'-Zeiger auf letztes Element der
    // neuen Liste setzen



    // Bei leerem Puffer 'in'-Zeiger
    // gleich 'out'-Zeiger setzen







    // Maximale Größe anpassen


}
```

(e) Implementieren Sie die private Funktion CreateList. Diese Funktion erzeugte eine einfach verkettete Liste der Größe size mit Elementen vom Typ ListElemT. Der Anfang der Liste wird in head zurückgegeben, das Ende in tail. Gehen Sie davon aus, dass in size immer ein Wert von mindestens eins übergeben wird. *(5 Punkte)*

```
template <class ItemT>
void GenDynBuffer<ItemT>::CreateList(ListElemT* & head,
                                     ListElemT* & tail, int size) {









}
```

Aufgabe 4 ADT Paketaufbereiter *(15 Punkte)*

Diese Aufgabe beschäftigt sich mit einem Teilproblem bei der Realisierung von geschichteten Netzwerkprotokollen. Das Gesamtsystem (siehe Abbildung) besteht aus einem Erzeuger, der Nachrichten versenden möchte, die aus beliebig langen Zeichenketten bestehen, sowie einem Versender, der das Verschicken der Nachrichten über das Netzwerk durchführt. Diese beiden kommunizieren untereinander über eine Instanz des in dieser Aufgabe zu implementierenden ADT Paketaufbereiter. Dieser erhält die Nachricht über einen Aufruf der Schnittstellenfunktion (siehe Listing auf der nächsten Seite) Send, bricht die Nachricht in einzelne (kürzere) Fragmente auf und legt sie in einem Puffer ab. Immer wenn das Netzwerk frei ist, kann der Versender mittels der Funktion GetNextPacket eins der Fragmente entnehmen und dann über das Netzwerk verschicken.



Intern verwendet Paketaufbereiter eine Instanz des generischen Puffers aus der vorherigen Aufgabe, um die Fragmente der Nachricht zu speichern. Diese wird auf der Halde erzeugt und die Adresse im Zeiger buffer gespeichert. Außerdem enthält die private Variable address die Netzwerkadresse des Empfängers der Nachricht, der Einfachheit halber hier als Integer. Die beim Aufruf von GetNextPacket erzeugten Pakete enthalten eine fortlaufende Nummer, dafür wird in currSeqNum die als nächstes zu verwendende Nummer verwaltet.

```
class Paketaufbereiter {
public:
    // Konstruktor, 'addr' enthaelt Netzwerkadresse
    // des Empfaengers
    Paketaufbereiter(int addr);

    // Nachricht 'data' aufbereiten
    void Send(string data);

    // Naechstes Paket zum Verschicken
    // in Referenzparameter 'aPacket' abholen
    bool GetNextPacket(string & aPacket);

private:
    // Netzwerkadresse, wird im Konstruktor initialisiert
    int address;

    // Fortlaufende Paketnummer, wird im Konstruktor auf '0' initialisiert
    int currSeqNum;

    // Konvertiert eine Zahl in deren Zeichenkettendarstellung
    string IntToStr(int integer);

    // Teilt den String 'data' in Fragmente fester Groesse auf,
    // speichert diese im per Rerenz übergebenen Feld 'frgs'.
    // Die Anzahl von dessen Elementen wird in 'frgsNum' zurückgegeben.
    void Fragment(string & data, string* & frgs, int & frgsNum);

    // Zeiger auf Instanz des generischen Puffers
    // siehe Teilaufgabe a)
    // ...

};
```

(a) Im privaten Teil der Schnittstelle fehlt eine Deklaration für einen *Zeiger* auf einen Puffer. Der Puffer ist eine Instanz des generischen Puffers aus der vorherigen Aufgabe. In dem Puffer sollen Strings gespeichert werden. Deklarieren Sie diesen Zeiger. Die Initialisierung des Zeigers findet im Konstruktor statt (nicht zu implementieren). *(3 Punkte)*

```
// Zeiger auf Instanz des generischen Puffers
```

(b) Implementieren Sie die Funktion **Send**. Ein Teil des Programmtexts ist bereits vorgegeben. Ergänzen Sie das Einfügen der zurückgegebenen Fragmente in den Puffer. Prüfen Sie dabei vor jedem Fragment, ob noch mindestens ein Platz im Puffer frei ist, falls nicht, vergrößern Sie ihn um **frgsNum** (Schnittstelle aus Aufgabe 3 verwenden!) *(5 Punkte)*

```
void Paketaufbereiter::Send(string data) {
    string* frgs = NULL;
    int frgsNum;
    // Aufteilen der Zeichenketten in Fragmente
    Fragment(data, frgs, frgsNum);
    // ab hier kann 'frgs' wie ein Feld der
    // Groesse 'frgsNum' verwendet werden

    // Alle Fragmente in 'buffer' ablegen









}
```

(c) Die tatsächlich versendbaren Daten können vom Versender per GetNextPacket wieder aus dem Puffer entnommen werden. Der Ausgabeparameter aPacket besteht nach dem Aufruf aus einem String, der sich zusammensetzt aus der Empfänger-Adresse, der fortlaufenden Nummer und dem eigentlichen Textfragment. Alle Teile des Pakets werden durch ein # voneinander abgetrennt. Ist der Puffer leer, so gibt GetNextPacket false zurück, sonst true. Implementieren Sie GetNextPacket. Hinweise: Der +-Operator kann auf zwei Strings angewendet werden und ergibt einen String, der aus der Verkettung beider Strings besteht. Die Funktion IntToStr konvertiert eine ganze Zahl in einen String. *(7 Punkte)*

```
bool Paketaufbereiter::GetNextPacket(string & aPacket) {
    // Sicherheitsabfrage: Puffer leer? Ggf. Abbruch und
    // geeigneter Rückgabewert









    // Adresse und Paketnummer mit Nachrichtenfragment aus
    // Puffer zusammensetzen









    // Fortlaufende Paketnummer erhöhen




    // Rückgabewert der Funktion


}
```