Prüfung

Prof. Dr.-Ing. K. D. Müller-Glaser

Informationstechnik

SS 2008

Institut für Technik der Informationsverarbeitung, Universität Karlsruhe

Musterlösung zur Klausur SS 2008

Hinweise zur Klausur

Hilfsmittel

Zur Prüfung sind keine Hilfsmittel zugelassen. Nicht erlaubt sind insbesondere die Verwendung eines Mobiltelefons und jegliche Kommunikation mit anderen Personen.

Prüfungsdauer

Die Prüfungsdauer beträgt 120 Minuten.

Prüfungsunterlagen

Die Prüfungsunterlagen bestehen aus insgesamt 23 Seiten Aufgabenblättern (einschließlich diesem Titelblatt und zusätzlicher Lösungsblätter).

Bitte vermerken Sie vor der Bearbeitung der Aufgaben auf jeder Seite oben Ihren Namen, auf der ersten Seite zusätzlich die Matrikelnummer!

Falls Sie zusätzliche Blätter zur Lösung der Aufgaben benötigen, verwenden Sie die zusätzlichen Seiten am Ende der Klausur bzw. fragen Sie zusätzlichem Papier. Auf jedes zusätzliche Lösungsblatt ist neben dem Namen auch die Aufgaben- und die Seitennummer mit einzutragen. Vermeiden Sie das Beschreiben der Rückseiten. Die Verwendung von eigenen Blättern ist nicht erlaubt. Am Ende der Prüfung sind die 23 Seiten Aufgaben- und Lösungsblätter und alle verwendeten zusätzlichen Lösungsblätter abzugeben.
Verwenden Sie zum Bearbeiten der Aufgaben lediglich dokumentenechte Schreibgeräte – keinen Bleistift sowie Rotstifte!

Wie verabredet enthält die Klausur mehr Aufgaben als, bei einer richtigen Lösung, zum Erreichen einer sehr guten Note (1,0) notwendig sind. (Nutzen Sie die Auswahlmöglichkeit)

Aufgabe	1	2	3	4	5	6	7	8	Σ
≈ Gewichtung ca. [%]	10	13	10	15	15	14	14	9	100
Ergebnis [P]									

Aufgabe 1 Allgemeine Fragen

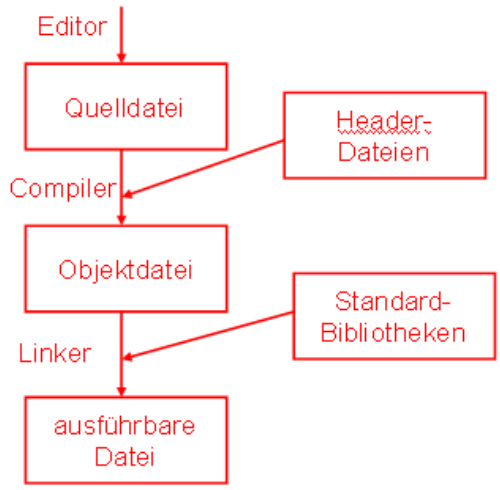
Beantworten Sie folgende Fragen:

A) Nennen Sie einen Unterschied zwischen einer imperativen und einer funktionalen Programmiersprache.

- Im Gegensatz zur imperativen Programmierung sind Funktionen in funktionalen Programmiersprachen keine zusammengefassten Codeblöcke, Funktion entspricht eher der mathematischen Definition einer Funktion
- Ein funktionales Programm besteht ausschließlich aus Funktionsdefinitionen und Funktionsaufrufen und besitzt keine Kontrollstrukturen wie z.B. Schleifen

B) Bringen Sie die folgenden Begriffe in eine logische Reihenfolge und stellen Sie die Beziehungen grafisch dar:

Linker - Headerdateien - Standardbibliotheken - ausführbare Datei - Compiler - Quelldatei - Editor - Objektdatei



C) Nennen Sie vier Phasen eines Softwareentwicklungsprozesses.

Analyse → Design/Entwurf → Konstruktion/Programmierung/Implementierung → Evaluation/Test

D) Nennen Sie zwei Unterschiede zwischen Zeigern und Referenzen.

- Zeiger belegt Speicher, Referenz belegt keinen Speicherplatz
- Zeiger ist veränderbar, Referenz muss initialisiert werden und ist nicht veränderbar
- Zeiger ist ein Ausdruck, der Adresse und Typ eines anderen Objektes repräsentiert, Referenz ist ein anderer Name („Aliasname“) für ein bereits existierendes Objekt

E) Welches Prinzip wird in einer Warteschlange (Queue) verwendet? Beschreiben Sie dieses kurz.

FIFO Prinzip (First-In First-Out), in einer Warteschlange kann eine beliebige Anzahl von Objekten gespeichert werden, die gespeicherten Objekte können nur in der gleichen Reihenfolge wieder gelesen werden, wie sie gespeichert wurden.

F) Escape-Sequenzen werden unter anderem zur Darstellung nicht druckbarer Zeichen eingesetzt.

☒ Richtig ☐ Falsch

G) Nennen Sie vier verschiedene Suchalgorithmen.

Lineare Suche, Binäre Suche, Interpolationssuche
Breitensuche, Tiefensuche

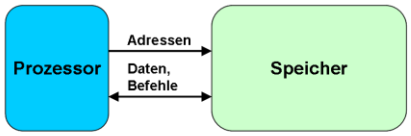
H) Erläutern Sie das Prinzip, auf dem die Datenstruktur Stapelspeicher (Stack) basiert, und nennen Sie die zugehörigen Operationen. Erläutern Sie zusätzlich die Aufgabe der jeweiligen Operation.

LIFO-Prinzip (Last-In First-Out): In einem Stack kann eine beliebige Anzahl von Objekten gespeichert werden, jedoch können die gespeicherten Objekte nur in umgekehrter Reihenfolge wieder gelesen werden, wie sie gespeichert wurden.

Operationen: push (Objekt im Stapelspeicher ablegen)
pop (zuletzt gespeichertes Objekt lesen und vom Stapel entfernen)
top oder peek (oberstes Element lesen ohne es zu löschen)

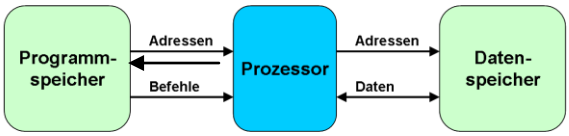
I) Was ist der Hauptunterschied zwischen der von-Neumann- und der Harvard-Architektur? Hinweis: Sie können Ihre Lösung auch graphisch darstellen.

von Neumann-Architektur



ein Speicher sowohl für Programm-Daten als auch Daten-Daten

Harvard-Architektur



getrennte Speicher für Programm-Daten und Daten-Daten / gleichzeitiger Zugriff möglich

J) Nennen Sie einen Vorteil und einen Nachteil der RISC-Rechnerarchitektur gegenüber der CISC-Rechnerarchitektur.

RISC: einfachere Befehle, schneller in der Ausführung

CISC: komplexere Befehle, weniger Programmcodezeilen

Aufgabe 2 C++ Verständnisfragen

Beantworten Sie folgende Fragen:

A) Gegeben ist der folgende kurze Programmausschnitt. Nehmen Sie an, dass das Programm korrekt und fehlerfrei kompiliert worden ist. Welches sind die Ausgaben auf dem Bildschirm aufgrund der dargestellten Zeilen C++ Programmcode?

```
int main()
{
    float zahl1 = 9/4;
    int zahl2 = 5.7;

    cout << zahl1 << endl;      2
    cout << zahl2 << endl;      5
}
```

B) Nennen Sie drei elementaren Datentypen für Ganzzahlen.

short, int, long, char

C) Nach Ausführung der folgenden Schleife, welchen Wert speichert die Variable *x*?

```
int x = 0, y = 0;
while( ++y < 3)
    x += y;
```

x = 3

D) Nach Ausführung der folgenden Anweisungen, welchen Wert speichert der String *s*?

```
string s("WOW!");
int pos = s.find("!");
s.insert(pos, ", " + s, 0, 5);
s.erase(2,4);
```

s = "WOOW!"

E) Definieren Sie in einer einzigen Zeile Programmcode ein Array mit Namen *VW*, in dem das Monatsgehalt von 20 Angestellten gespeichert werden kann. Die ersten beiden Arrayelemente werden mit jeweils 3000.0 initialisiert.

```
float/double VW[20] = {3000.0, 3000.0};
```

F) Einer Funktion wird ein Arrayname als Argument übergeben. Was erhält daraus resultierend die Funktion?

Die Funktion erhält nur die Adresse des ersten Arrayelements. / Pointer auf Array

G) Hier soll zwischen richtigem und fehlerhaftem C++ Code unterschieden werden.

a) `int n, int i;`

☐ Richtig ☒ Falsch

Trennung der Befehle durch Semikolon

b) `char c('a');`

☒ Richtig ☐ Falsch

c) `Short min(0);`

☐ Richtig ☒ Falsch

Short muss klein geschrieben werden

d) `double 2_slow = 1E-4;`

☐ Richtig ☒ Falsch

Variablenname darf nicht mit einer Zahl beginnen

H) ~~Angenommen die *int*-Variable *i* speichert die Zahl 3, dann wird durch den Befehl~~

~~`cout << --i << "," << i++ <<endl;`~~

~~die folgende Ausgabe generiert:~~

~~a) 2,2~~ b) 3,4 c) 3,3

Antwort: ☒

I) Angenommen *Z* ist eine *bool*-Variable, *X* und *Y* zwei *int*-Variablen, die 1 bzw. 2 speichern. Finden Sie heraus, ob *Z* nach Ausführung der jeweiligen Zeile den Wert *true* oder *false* bekommt.

A) `Z = X-- || ++Y % 3;` ☒ *true* ☐ *false*

B) `Z = (Y | 1 && --X) % 3;` ☐ *true* ☒ *false*

J) Sind die folgenden Funktionsaufrufe richtig oder falsch?

a)
```
double mittel(double, double, double); //Prototyp
double result = mittel (7.0, 12.3); //Aufruf
```
☐ Richtig ☒ Falsch

mittel( ) besitzt drei Argumente nicht zwei

b)
```
void put(char c); //Prototyp
char c = put('A'); //Aufruf
```
☐ Richtig ☒ Falsch

put( ) hat keinen Rückgabewert

c)
```
double kubik(double); //Prototyp
double x = 2.4; //Variablendeklaration
cout << kubik(x); //Aufruf
```
☒ Richtig ☐ Falsch

K) Nach Ausführung der *switch*-Anweisung

```
char c = 'S';
switch (c)
{
   default : c = 'X';
   case 'S': c = 'T';
   case 'A': c = 'B';
}
```

Beachten: die break Anweisung fehlt

enthält die Variable *c* das folgende Zeichen:

a) `'B'` b) `'X'` c) `'T'`

Antwort: a)

L) Finden Sie jeweils einen Fehler in jedem der folgenden Quellcodeabschnitte?

a)
```
class A
{
  private:
       string typ;
  public:
       void A(const string&);
       string get(void) const;
};
```

Antwort: Der Konstruktor darf keinen Rückgabewert aufweisen.

b)
```
class B
{
  private:
       double Arr[5];
  public:
       double* feld(int i){ double erg; erg = Arr[i];
                            return &erg; }
}
```

Antwort: (1) Ein Semikolon fehlt nach der abschließenden geschweiften Klammer der Definition der Klasse *B*. (2) Die Funktion *feld* gibt einen Zeiger auf ein Objekt (Lokale Variable *erg*), das innerhalb der Funktion definiert wurde, zurück.

c)
```
class C
{
  private:
       long x;
  public:
       C(long n);
       void setX(long n)const {x = n;}
};
```

(2) wird vom Compiler nicht als Fehler erkannt, daher (1) oder (2)

Antwort: Die *setX* –Methode verändert das Objekt und kann daher nicht als read-only deklariert werden.

Aufgabe 3 Dateiverarbeitung

In dieser Aufgabe soll eine Datei eingelesen und der Inhalt der Datei mit Hilfe von Methoden der Klasse / Bibliothek `string` verarbeitet werden. Gegeben sei die Datei, welche in Abbildung 3.1 zu sehen ist (entspricht einem Auszug aus einer Schaltwerksdatei des Projektpraktikums).
Die Form einer Zeile ist wie folgt vorgegeben: Die Zeile beginnt mit *g*, gefolgt von einer dreistelligen Zahl. Anschließend folgt nach einem Doppelpunkt der Typ des jeweiligen Gatters. In den folgenden Klammern sind die Signale getrennt durch Kommata benannt. Das letzt genannte Signal bildet den Ausgang, die anderen sind die Eingänge (beim Gatterelement inv1a gibt es nur einen Eingang). Abgeschlossen wird die Zeile mit einem Semikolon.
Die Aufgabenstellung ist zu überprüfen, ob das Signal *clk* (Taktsignal) korrekt angeschlossen ist. **Das Signal *clk* muss an allen Flipflops am zweiten Eingang angeschlossen sein und darf an keinem anderen Eingang oder Ausgang (Flipflop oder anderes Gatterelement) angeschlossen sein.** Implementieren Sie die folgenden Programmteile, die im Rahmen der Aufgabenstellung benötigt werden (die Aufgabenstellung wird hierdurch nicht vollständig gelöst).

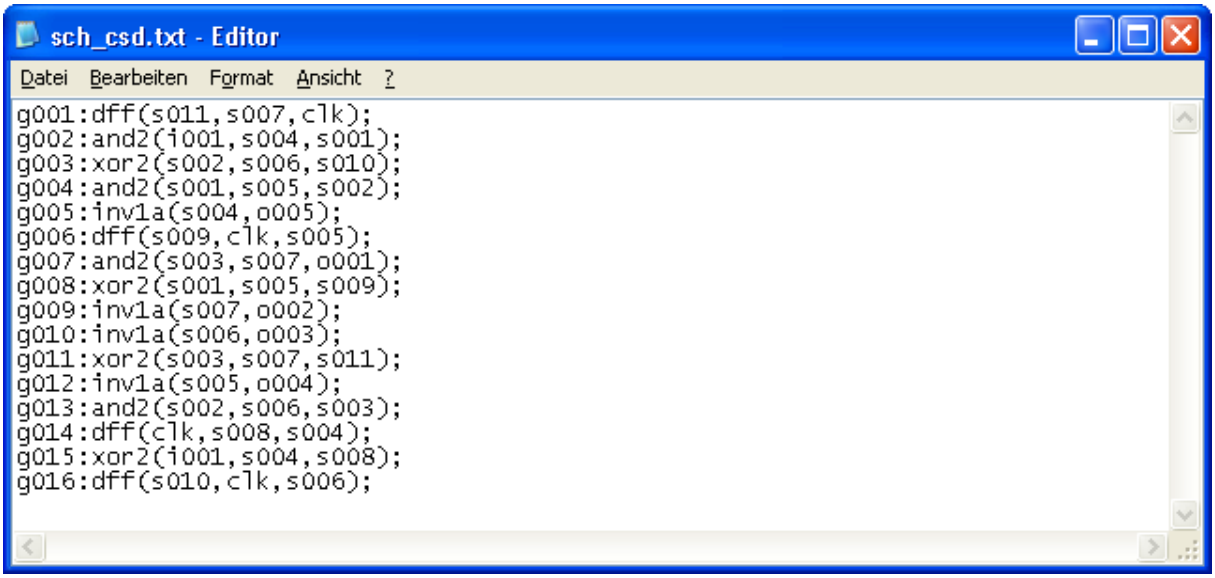


Abbildung 3.1: Schaltwerksdatei

A) Schreiben Sie eine Zeile Programmcode, welche die Datei "sch_csd.txt" zum lesen öffnet.

```
ifstream myfile("sch_csd.txt");              //oder
fstream myfile("sch_csd.txt",ios::in);
```

B) Schreiben Sie eine Schleife, welche eine Datei komplett Zeile für Zeile bis zum Ende ausliest und auf dem Bildschirm darstellt. Die Datei ist bereits unter dem Filestream "myfile" geöffnet. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
string zeile;
while(getline(myfile, zeile))
{
     cout << zeile << endl;
}
```

C) Schreiben Sie die Funktion *check_dff( )*, welche als Übergabewert eine Zeile aus der Datei erhält. Die Funktion soll ausschließlich überprüfen, ob die Zeile ein Flipflop darstellt. Wenn dies der Fall ist, soll der Rückgabewert *true* sein, andernfalls *false*. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
bool check_dff(string zeile)
{
    if ((zeile.find("dff") != -1)
        return true;

    return false;
}
```

D) Schreiben Sie die Funktion *check_clk_dff( )*, welche überprüft, ob das Signal *clk* am zweiten Eingang angeschlossen ist. Zu beachten ist, dass das Signal *clk* nur am zweiten Eingang vom Flipflop und an keinem anderen Eingang oder Ausgang vom Flipflop angeschlossen sein darf. Es kann dabei davon ausgegangen werden, dass die Funktion nur aufgerufen wird, wenn es sich bei der Zeile um ein Flipflop handelt. Die Funktion soll *true* zurückgeben, wenn das Signal *clk* nur am zweiten Eingang angeschlossen ist, andernfalls *false*. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
bool check_clk_dff(string zeile)
{
    int k1 = zeile.find("(");
    int beg = zeile.find(",");
    int end = zeile.find(",", beg+1);
    int k2 = zeile.find(")");

    if (zeile.substr(k1+1, beg-k1-1) == "clk")
        return false;

    if (zeile.substr(end+1, k2-end-1) == "clk")
        return false;

    if (zeile.substr(beg+1, end-beg-1) == "clk")
        return true;
    else
        return false;
}
```

Aufgabe 4 Kontrollstrukturen und Sortieralgorithmus

Gegeben ist eine einfach verkettete Liste *VKL* aus mehreren Elementen vom Typ "Graphelement". Zur verketteten Liste gibt es einen Zeiger auf das erste Element (*VKL_start*) und einen Zeiger auf das letzte Element (*VKL_ende*). Zusätzlich zeigt die Variable *next* von jedem Graphelement der verketteten Liste auf das nächste Element in der Liste. Beim letzten Element zeigt die Variable *next* auf NULL. Außerdem gibt es eine weitere globale Variable (*anzahl_Elemente*), die die Anzahl der Graphelemente der verketten Liste speichert.

```
struct Graphelement
{
   string name;
   Gatter* typ;
   Graphelement* next;
   Graphelement* graph[5];
   int anzahl_graph_ziele;
   double laufzeit;
};

Graphelement* VKL_start = NULL;
Graphelement* VKL_ende = NULL;

short anzahl_Elemente;
```

Hinweis: Es soll angenommen werden, dass die verkettete Liste schon aufgebaut ist, die Zeiger *VKL_start* und *VKL_ende* auf das jeweilige Graphelement zeigen und die Datenelemente der einzelnen Graphelemente komplett gefüllt sind.

A) Schreiben Sie eine for-Schleife, die die Anzahl der Elemente der verketteten Liste ermittelt und in der Variablen *anzahl_Elemente* speichert. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
anzahl_Elemente = 0;

for(Graphelement* ptr_Graph = VKL_start; ptr_Graph != NULL;
    ptr_Graph = ptr_Graph->next)
   anzahl_Elemente += 1;

 // oder

for(Graphelement* ptr_Graph = VKL_start; ptr_Graph != NULL;
    anzahl_Elemente += 1)
   ptr_Graph = ptr_Graph->next;
```

B) Erzeugen Sie nun ein statisches Array mit dem Namen *Arr_ptr*, das aus 20 Elementen (Annahme: *anzahl_Elemente* ist kleiner gleich 20) vom Typ „Zeiger auf konstantes Graphelement“ besteht. Im nächsten Schritt soll jedes Feld an der i-ten Stelle ($0 \leq i \leq$ *anzahl_Elemente*-1) des Arrays auf das i-te Graphelement in der verketteten Liste zeigen. Die restlichen Zeiger des Arrays sollen, wenn noch welche nicht benutzt werden, auf NULL zeigen. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
const Graphelement* Arr_ptr[20];
int j = 0;

for(Graphelement* ptr_Graph = VKL_start; ptr_Graph != NULL;
    ptr_Graph = ptr_Graph->next)
{
   Arr_ptr[j] = ptr_Graph;
   j++;
}
for (j; j < 20; j++) Arr_ptr[j] = NULL;

// oder

Graphelement* temp = VKL_start;

for (unsigned int i = 0; i < 20; i++)
    if (i < anzahl_Elemente) {
       Arr_ptr[i] = temp;
       temp = temp->next;
   } else
       Arr_ptr[i] = NULL;
```

C) Nun soll der Quicksort-Sortieralgorithmus angewendet werden, um die Graphelemente nach deren Laufzeit zu sortieren. Hierzu soll das Sortieren anhand der „Zeiger auf konstantes Graphelement“ des Arrays *Arr_ptr* durchgeführt werden.

a) Nennen Sie einen Vorteil des Sortierens mit Zeigern auf konstante Datenelemente.

Das Sortieren mit Zeigern auf konstante Datenelemente schützt die zu sortierenden Daten, in dem es deren Veränderung verhindert (nur read-only Operationen).
oder
Schnelleres Sortieren (direkter Zugriff auf die Elemente)

b) Im Folgenden ist der Pseudocode vom Quicksort-Algorithmus angegeben.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 vertausche A[i] ↔ A[j]
7  vertausche A[i + 1] ↔ A[r]
8  return i + 1
```

Hinweis: Um ein ganzes Feld A zu sortieren, muss ***Quicksort(A,0,länge[A]-1)*** aufgerufen werden.
Setzen Sie die zwei Funktionen *Quicksort* und *Partition* in C++ Code um. Passen Sie dabei den Pseudocode der Problemstellung an (in Bezug auf den Vergleich der Laufzeiten). Nachstehend sind die Prototypen dieser Funktionen vorgegeben. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
short Partition (const Graphelement* A[], short p, short r)
{
   const Graphelement* x = NULL = A[r];
   const Graphelement* tmp;
   signed int i = p-1;


   for(int j=p; j <= (r-1); j++)
      if (A[j]->laufzeit <= x->laufzeit)
      {
         i++;
         tmp = A[i];
         A[i] = A[j];
         A[j] = tmp;
      }
   tmp = A[i+1];
   A[i+1] = A[r];
   A[r] = tmp;

   return i+1;









}
```

```
void Quicksort (const Graphelement* A[], short p, short r)
{
   short q;

   if (p < r)
   {
      q = Partition(A,p,r);
      Quicksort(A,p,q-1);
      Quicksort(A,q+1,r);
   }




}
```

c) Schreiben Sie nun eine C++ Zeile, die das Sortieren des Arrays *Arr_ptr* nach dem Quicksort-Algorithmus durchführt.

```
Quicksort(Arr_ptr, 0, anzahl_Elemente-1);
```

Aufgabe 5 Straßenbahnanzeigetafel

Zur Speicherung und Steuerung der Anzeige an einer Straßenbahnhaltestelle (Tramhaltestelle) soll eine doppelt verkettete Liste verwendet werden. Gegeben ist dabei der folgende C++ Programmcode, dieser ist an zwei Stellen nicht vollständig. Außerdem sind die Funktionen noch nicht implementiert. Sie sollen in den jeweiligen Unteraufgaben den Programmcode ergänzen und sich mit zwei bestimmten Funktionen beschäftigen. Eine Beschreibung der Funktionen und des Aufbaus einer doppelt verketteten Liste finden Sie auf der nächsten Seite.

```
#include <iostream>
#include <string>
using namespace std;

const short max_anzahl = 5;

//Hier steht der Programmcode aus Aufgabenteil A!

Tram* start_liste = NULL;
Tram* ende_liste = NULL;

void daten_holen(short* linie, string* Ziel, short* minuten);
void tram_hinzufuegen(short linie, string Ziel, short minuten);
void tram_anzeigen(Tram *ptram);

bool tram_ausgefallen(Tram* ptr_tram);
void tram_loeschen(Tram* ptr_tram);

int main()
{
   int anzahl = 0;
   short linie; string Ziel; short minuten;

   while (true)
   {
      Tram* ptram = start_liste;
      while (ptram != NULL)
      {
         ptram->minuten--;
         if ((ptram->minuten == 0) || (tram_ausgefallen(ptram) == true))
         {
             tram_loeschen(ptram);
             ptram = start_liste;
             anzahl--;
         }
         else
            ptram = ptram->next;
      }

      while (anzahl < max_anzahl)
      {
         daten_holen(&linie, &Ziel, &minuten);
         tram_hinzufuegen(linie, Ziel, minuten);
         anzahl++;
      }

      //Hier steht der Programmcode aus Aufgabenteil B!
          tram_anzeigen(ptram);
      delay(60);
   }
   return 0;
}
```

- Die Funktion *daten_holen( )* importiert Daten von der nächsten ankommenden Straßenbahn aus der Datenbank und speichert diese in den entsprechenden Variablen.

- Die Funktion *tram_hinzufuegen( )* erzeugt ein neues Element am Ende der doppelt verketteten Liste und speichert die übergebenen Daten in diesem Element.

- Die Funktion *tram_anzeigen( )* stellt die Daten des übergebenen Elements (Typ: *Tram*) auf der Anzeigetafel dar.

- Die Funktion *tram_ausgefallen( )* hat als Rückgabewert *true*, falls die übergebene Straßenbahn ausgefallen ist, andernfalls *false*.

- Die Funktion *tram_loeschen( )* löscht das übergebene Element (Adresse des Elements wird übergeben) aus der verketteten Liste.

- Die Funktion *delay( )* wartet die übergebene Zeit in Sekunden.

Die folgende Abbildung zeigt den prinzipiellen Aufbau einer doppelt verketteten Liste.

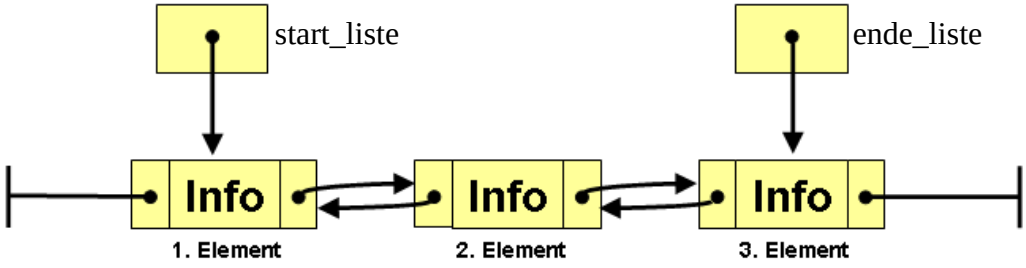


Abbildung 4.1 prinzipieller Aufbau einer doppelt verketteten Liste

A) Gegeben sind die folgenden Informationen (entscheiden Sie dabei selbstständig, welche Informationen Sie benötigen und an welcher Stelle):
   - Die Anzeige kann maximal die fünf folgenden Straßenbahnen anzeigen.
   - Die Anzeige zeigt die einstellige Nummer der Straßenbahn.
   - Die Anzeige zeigt das Ziel der Straßenbahn. Dabei ist das Ziel ein Ort oder eine Straße.
   - Die Anzeige zeigt, in wie viel Minuten die Straßenbahn voraussichtlich ankommt. Es werden nur volle Minuten angezeigt (maximal 99 Minuten).

   Definieren Sie eine **minimale** (bezüglich der Anzahl der Elemente) Struktur (struct) *Tram*, welche alle Informationen bezüglich **einer** Straßenbahn speichert. Diese Struktur soll auch direkt für die doppelt verkettete Liste verwendet werden können. Verwenden Sie für die einzelnen Werte geeignete Variablentypen. Beachten Sie bezüglich der Variablennamen auch den oben vorgegebenen Programmcode.
   → Lösen Sie die Aufgabe auf der nächsten Seite

Lösung zur Aufgabe 5A):

```
struct Tram
{
    short linie;
    string ziel;
    short minuten;
    Tram* next;
    Tram* prev;
};
```

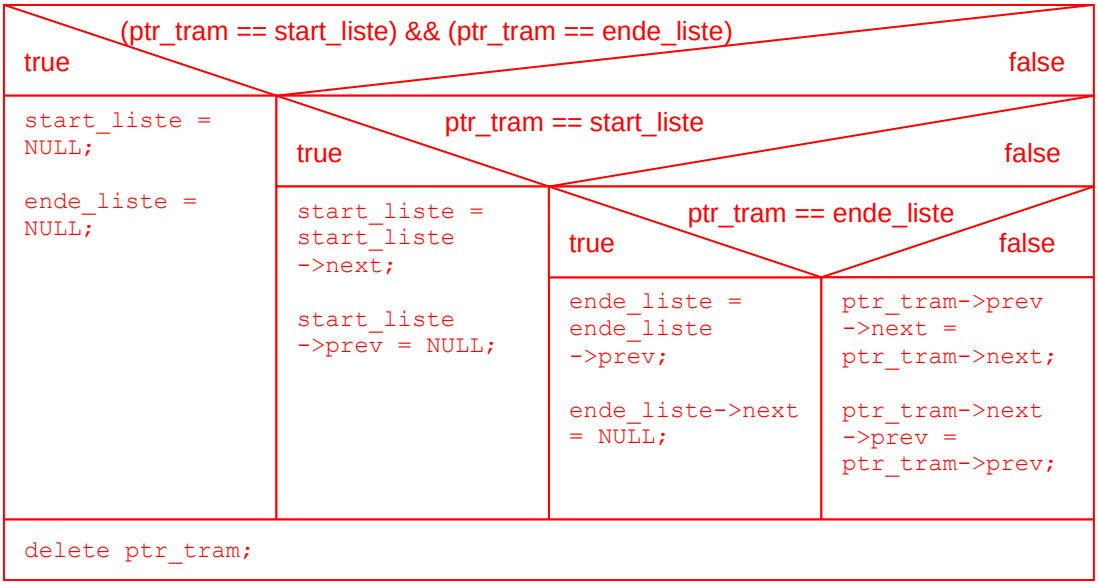
B) Schreiben Sie eine *for*-Scheife, welche alle Elemente in der doppelt verketten Liste (Typ: *Tram*) nacheinander vom ersten bis zum letzten Element durchgeht. Die Laufvariable soll dabei den Namen *ptram* haben. Dabei wird in der for-Schleife die Funktion *tram-anzeigen( )* aufgerufen-siehe Hauptprogramm.

```
for (Tram* ptram = start_liste; ptr_tram != NULL; ptram = ptram->next)
```

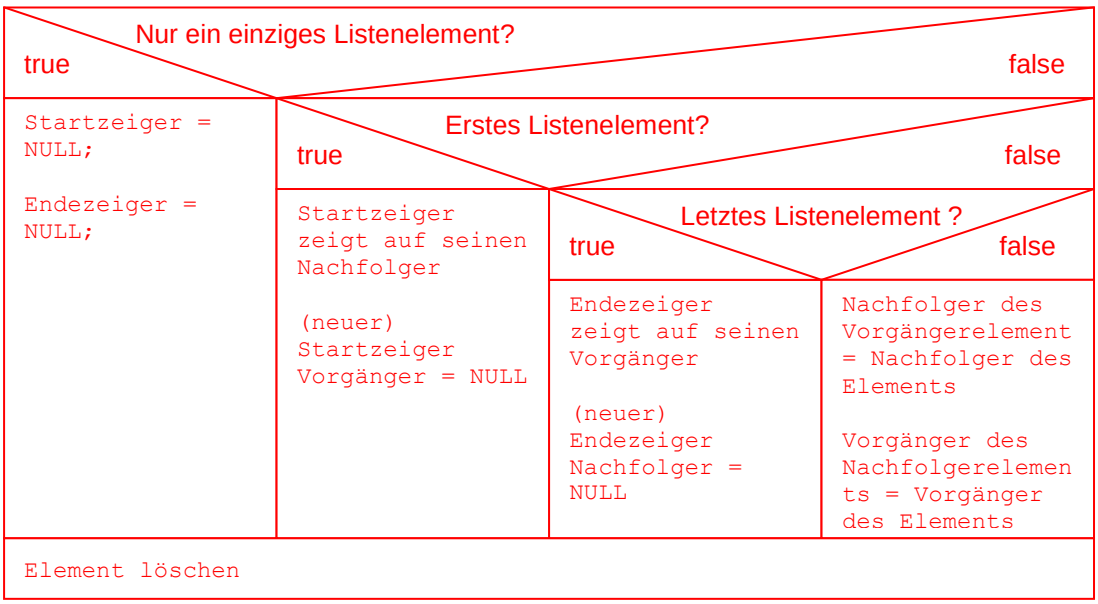
C) Implementieren Sie die Funktion *tram_hinzufuegen( )*. Diese soll ein neues Element vom Typ *Tram* erzeugen und dies am Ende der doppelt verketten Liste anfügen (Dabei soll nach dem Hinzufügen eine doppelt verkettete Liste vorhanden sein – Abbildung 4.1). Weiterhin sollen die übergebenen Informationen in das neu erzeugte Element gespeichert werden.

```
void tram_hinzufuegen(short linie, string Ziel, short minuten)
{
   Tram* neu_tram = new Tram;

   if (start_liste == NULL)
   {
      start_liste = neu_tram;
      neu_tram->prev = NULL;
   }
   else
   {
      neu_tram->prev = ende_liste;
      ende_liste->next = neu_tram;
   }

   ende_liste = neu_tram;
   neu_tram->next = NULL;

   neu_tram->linie = linie;
   neu_tram->minuten = minuten;
   neu_tram->ziel = Ziel;




}
```

D) Zeichnen Sie zur Funktion *tram_loeschen( )* ein Nassi-Shneiderman-Diagramm, welches den Ablauf der Funktion beschreibt. Die Funktion erhält als Übergabewert die Adresse des zu löschenden Elements und soll dieses aus der doppelt verketteten Liste entfernen. (Dabei soll nach dem Löschen eine doppelt verkettete Liste vorhanden sein - Abbildung 4.1). Sie können dabei im Nassi-Shneiderman-Diagramm Pseudocode oder C++-Programmcode verwenden.
Hinweis: Die Zeiger *start_liste* und *end_liste* sind globale Varibalen.



oder:

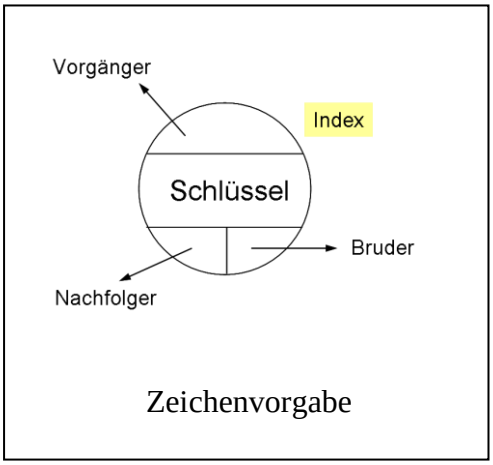


Aufgabe 6 Datenstrukturen, Baumdarstellung

A) Nennen Sie mindestens vier Operationen, die auf dynamische Datenstrukturen ausgeführt werden können.

Search(S,k), Insert(S,x), Delete(S,x), Minimum(S), Maximum(S), Successor(S,x), Predecessor(S,x)

B) Zeichnen Sie die verkettete Datenstruktur zur Darstellung des Baums, der durch die in der folgenden Tabelle gelisteten Attribute gegeben ist. Tragen Sie bei jedem Knoten ein: Index, Schlüssel und die notwendigen Zeiger.
Der Index des Wurzelelements ist 5.

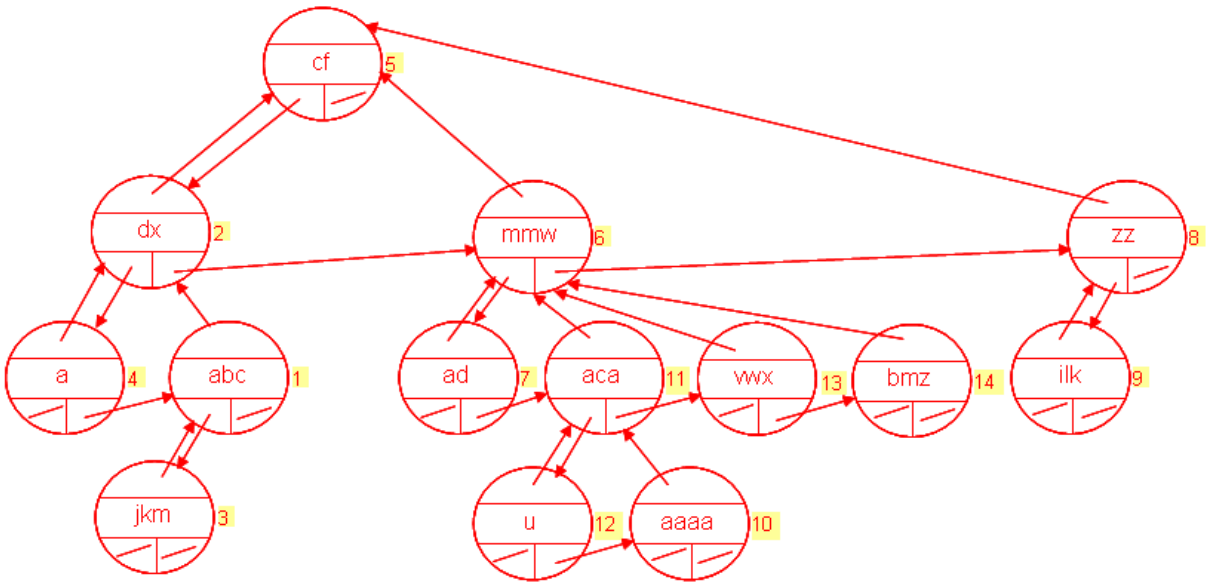


Zeichenvorgabe

Index	Schlüssel	Nachfolger	Bruder
1	abc	3	NIL
2	dx	4	6
3	jkm	NIL	NIL
4	a	NIL	1
5	cf	2	NIL
6	mmw	7	8
7	ad	NIL	11
8	zz	9	NIL
9	ilk	NIL	NIL
10	aaaa	NIL	NIL
11	aca	12	13
12	u	NIL	10
13	vwx	NIL	14
14	bmz	NIL	NIL

→ Lösen Sie die Aufgabe auf der nächsten Seite

Lösung zu Aufgabe 6B):



C) Welche maximale Tiefe hat der Baum (Abstand von der Wurzel zu den Blättern)?

maximale Tiefe: 3

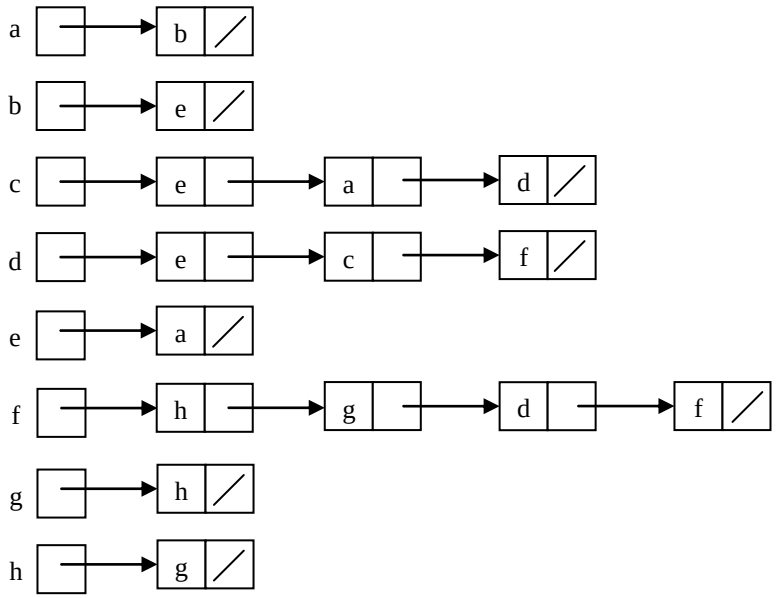
D) Geben Sie den Pseudocode für einen Algorithmus an, mit dem Sie die maximale Tiefe des Baums ermitteln können.

Hinweis: Das Problem lässt sich sehr gut rekursiv lösen.

```
max_tiefe = 0;
tiefe = 0;
tiefe_bestimmen (startknoten, tiefe);

tiefe_bestimmen (knoten, tiefe)
{
   if (max_tiefe < tiefe)
      max_tiefe = tiefe;
   if (knoten.bruder != NULL)
      tiefe_bestimmen (knoten.bruder, tiefe);
   if (knoten.nachfolger != NULL)
   {
      tiefe++;
      tiefe_bestimmen (knoten.nachfolger, tiefe);
   }
}
```

Aufgabe 7 Graphen und Tiefensuchalgorithmus

Gegeben ist die folgende Adjazenzliste:



Gegeben ist der Pseudocode des Tiefensuchalgorithmus:

```
DFS(G)
for alle Knoten u ε V[G]
  do farbe[u]=weiss;
     Π[u]=NIL;
zeit=0;
for alle Knoten u ε V[G]
  do if farbe[u]==weiss
    then DFS-VISIT(u);

DFS-VISIT(u)
farbe[u]=grau; zeit=zeit+1; d[u]=zeit;
for alle v ε Adj[u]
    do if farbe[v]==weiss
      then Π[v]=u;
           DFS-VISIT(v);
farbe[u]=schwarz; zeit=zeit+1; f[u]=zeit;
```

A) Die folgenden Fragen beziehen sich auf die Adjazenzliste (Seite 20)

a) Die Adjazenzliste beschreibt einen (bitte ankreuzen und begründen):

☐ ungerichteten Graphen

☒ gerichteten Graphen

Begründung: bei ungerichteten Graphen muss jede Kante zweimal vorkommen {a, b} und {b, a}

b) Wieviele Knoten und Kanten hat dieser Graph:

Anzahl Knoten: **8** Anzahl Kanten: **15**

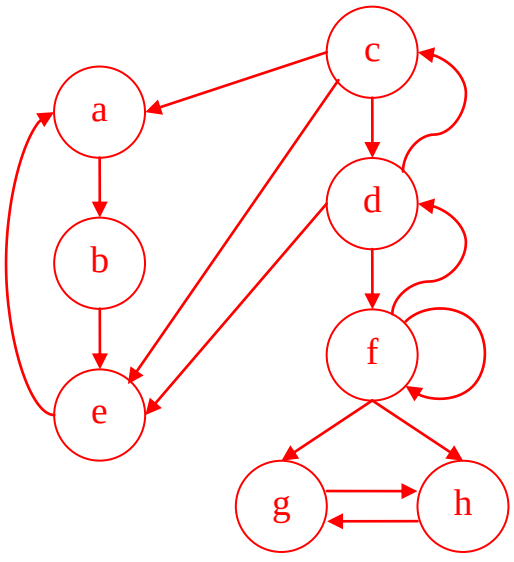
c) Bestimmen Sie den Grad des Knotens f:

d(f) = **6**

d) Bestimmen Sie die Mächtigkeit der Menge der direkt benachbarten Knoten von f

|V'(f)| = **4**

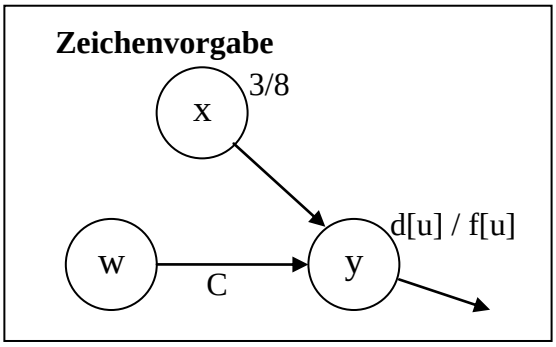
e) Geben Sie eine beliebige einfache Kantenprogression der Länge 4 an:



mögliche Lösungen siehe Graph. z.B.: c-d-f-g-h oder c-d-e-a-b usw.

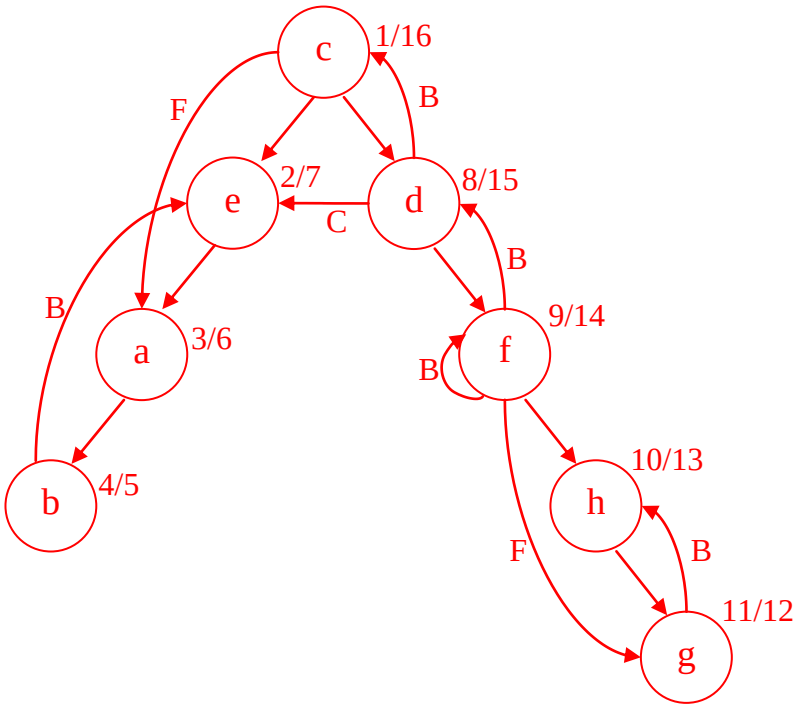
B) Wenden Sie den Tiefensuchalgorithmus (DFS) auf die Adjazenzliste (Seite 20) an, um festzustellen, ob ausgehend vom Knoten c der Knoten g erreichbar ist.

a) Arbeiten Sie den DFS-Algorithmus vollständig ab. Zeichnen Sie den entstehenden Tiefensuchbaum ausgehend von Startknoten c.



- Die Entdeckungsreihenfolge ist dabei bestimmt durch die Adjazenzliste.
- Bestimmen Sie für jeden Knoten die Entdeckungszeit d[u] und die Endzeit f[u]
- Ergänzen Sie alle restlichen Kanten und klassifizieren Sie die Kanten im Tiefsuchbaum und markieren Sie

Baumkanten:	keine Markierung
Vorwärtskanten:	mit einem F (Forward)
Rückwärtskanten:	mit einem B (Backward)
Querkanten:	mit einem C (Cross)



b) Bei welcher Entdeckungszeit könnte der Algorithmus abgebrochen werden, weil der Zielknoten g entdeckt wurde?

möglicher Algorithmus Abbruch bei: 11

Aufgabe 8 O-Notation, Simulated Annealing

Gegeben ist der folgende Algorithmus für Simulated Annealing in Pseudocode:

```
1.  Lies n;                                  //positiv, short integer
2.  T = 2^n;                                 //Anfangswert der Temperatur
3.  X = j0;                                  //Anfangskonfiguration
4.  while (T > 1) do                         //Endtemperatur
5.      Anzahl_Versuche = 0;                 //Anzahl der Versuche: Initialisierung
6.      while (Anzahl_Versuche < 1000) do    //Anzahl der Versuche bei einer Temperatur
7.          j = generate(X);                 //Generiere neue Konfiguration
8.          if (accept(cost(j), cost(X), T)) //spezielle Akzeptanzfunktion
9.              X = j;                       //akzeptierte Lösung abspeichern
10.         Anzahl_Versuche++;               //Anzahl der Versuche: Inkrementierung um 1
11.     T = T/2;                             //Senken der Temperatur
```

In dieser Aufgabe soll die asymptotische Laufzeit vom oben beschriebenen Algorithmus (Simulated Annealing) für den worstcase bestimmt werden. Hierzu soll anhand des Pseudo-Codes die Gesamtlaufzeit T(n) ermittelt und jeweils eine obere asymptotische Schranke O(n) bestimmt werden. Dabei ist zu beachten, dass c_i die Kosten der jeweiligen Codezeile i darstellt und dass die Rechenzeit der Funktionen *generate( )* und *accept( )* bereits in den jeweiligen Kosten (c_7 bzw c_8) beinhaltet sind.

Zeile	Kostenkoeffizient	Zeit(worstcase)
1	c_1	1
2	c_2	1
3	c_3	1
4	c_4	n+1
5	c_5	n
6	c_6	(1000+1)*n
7	c_7	1000*n
8	c_8	1000*n
9	c_9	1000*n
10	c_{10}	1000*n
11	c_{11}	n

T(n) = $n * (c_4 + c_5 + 1001 * c_6 + 1000 * c_7 + 1000 * c_8 + 1000 * c_9 + 1000 * c_{10} + c_{11}) + (c_1 + c_2 + c_3 + c_4)$

O(n) = n