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

Probeklausur
Grundgebiete der Informatik II

Datum: 18. 06. 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				
Σ	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 B.1 nicht unterschritten werden.

- Welcher der Sachverhalte ist ein Beispiel für kontextfreie Syntax? *(1 Punkt)*
  - ☒ Anweisungen in einer Anweisungsliste müssen durch “;” voneinander getrennt werden.
  - ☐ Die Typverträglichkeit von Operand und Operator muss gegeben sein.
  - ☐ Bezeichner müssen vor ihrer Verwendung deklariert werden.
  - ☐ Die Parameter bei Deklaration und Aufruf einer Prozedur müssen übereinstimmen.
- Gegeben seien folgende Deklarationen: `T var; T *zgr;`. Welche der folgenden Deklarationen mit Initialisierung ist korrekt? *(2 Punkte)*
  - ☐ `T ref = &var;`
  - ☒ `T &ref = var;`
  - ☐ `T ref = &zgr;`
  - ☐ `T ref = *var;`
- “Call by Reference” bezeichnet den Parameterübergabe-Mechanismus, bei dem ... *(1 Punkt)*
  - ☐ der Wert des Aktualparameters in den Formalparameter kopiert wird.
  - ☒ Formalparameteränderungen den Aktualparameter verändern.
- Welchen Aufwand hat die binäre Suche in einem sortierten Feld mit n Elementen im schlechtesten Fall? *(1 Punkt)*
  - ☐ $O(n * n)$
  - ☐ $O(n * \log(n))$
  - ☒ $O(\log(n))$
  - ☐ $O(\log(n) * \log(n))$
- Wie ist der Gültigkeitsbereich einer Variablen definiert? *(1 Punkt)*

- ☐ Eine Variable ist nur gültig innerhalb des Blocks, in dem sie deklariert ist, und in den umgebenden Blöcken, ggfs. nicht sichtbar aufgrund weiterer Deklarationen mit gleichem Bezeichner.
- ☐ Variable nur im entsprechenden Block gültig, weder in den enthaltenen noch in umgebenden Blöcken.
- ☒ Variable ist nur im entsprechenden Block und den darin enthaltenen Blöcken gültig, ggfs. nicht sichtbar aufgrund weiterer Deklarationen mit gleichem Bezeichner.

- Definieren Sie kurz die Sichtbarkeit von Variablen. *(1 Punkt)*

  Variablen sind innerhalb des Blocks und allen enthaltenen Blöcken sichtbar, es sei denn dort sind Deklarationen mit gleichem Namen, diese verdecken dann die andere Deklaration.

- Welche Besonderheit kennzeichnet ein stabiles Sortierverfahren? *(1 Punkt)*

  - ☐ Das Feld ist anschließend absteigend sortiert.
  - ☒ Die Reihenfolge von Elementen mit gleichem Schlüssel wird nicht verändert.
  - ☐ Stabile Sortierverfahren sind für die Sortierung auf stabilen Speichermedien (Bandlaufwerken) besonders gut geeignet.

- Das Sortierverfahren “Sortieren durch Einfügen” hat den Aufwand $O(n^2)$. Für welchen Fall gilt dies? *(2 Punkte)*

  - ☐ nur für den besten Fall
  - ☐ nur für den durchschnittlichen Fall
  - ☐ für alle Fälle
  - ☒ für alle Fälle ausser dem besten

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

(a) Im folgenden Programm sind insgesamt 3 Syntaxfehler versteckt. Finden Sie diese und ordnen Sie sie in die Kategorien kontextfreie und kontextsensitive Fehler ein. Tragen Sie die Fehler in die nachfolgende Tabelle ein. Geben Sie für jeden Fehler die Zeilennummer, eine kurze Beschreibung des Fehlers und seine Kategorie an. *(3 Punkte)*

```
1   void StraightSelection(int feld[], int start, int ende) {
2       int elem;
3       for (int i = start; i <= ende - 1; i++) {
4         int k = i;
5         elem = feld[i];
6         for (int j = i + 1; j <= ende; j++) {
7             // ...
8         }
9         feld[k] = feld[j];
10        feld[i] = elem;
11      }
12      return 0;
13  }
14  int main()  {
15    int feld[] ={ 'g', 'a', 'i', 'h', 'n', 'l', 'm', 'x' };
16    StraightSelection(feld, 0, 7);
17    return 0;
18
```

Zeile	Beschreibung	ks/kf
9	j hier nicht mehr gültig	ks
12	return in void Funktion verboten	ks
18	Abschließende } fehlt	kf

(b) Erstellen Sie, basierend auf der folgenden Beschreibung, ein Codefragment. Sie sollen also kein vollständiges Programm erstellen! *(3 Punkte)*

Zu deklarieren sind ein int-Feld namens feld, das maximal 100 Elemente speichern kann, und zusätzlich weitere Hilfsvariablen. Das Ende des Feldes wird durch den Wert -1 markiert. Summieren Sie alle Elemente des Felds auf und speichern Sie das arithmetische Mittel in der Variablen mittel ab.

Neben den Deklarationen ist lediglich der Algorithmus zum Bilden des arithmetischen Mittels zu realisieren, nicht jedoch das Füllen des Feldes und Abspeichern des Endes des Feldes durch ein Element mit dem Wert -1. Sie können davon ausgehen, dass die Endemarkierung innerhalb des Feldes auftritt.

```
int feld[100];
int mittel;
int summe = 0;
int i = 0;
while (feld[i] != -1) {
  summe += feld[i];
  i++;
}
mittel = summe / i;
```

(c) In dem folgenden Programm werden die Variablen `a`, `b` und `c` an die Prozedur `f` übergeben. Geben Sie in der Tabelle den Wert von `a`, `b` und `c` nach der jeweiligen Ausführung der Prozedur `f` an. *(4 Punkte)*

```
void f(int &x, int y, int *z) {
  x=*z+x;
  y++;
  *z= y - *z;
}

int main() {
  int a=1, b=3, c=5;
  f(a, b, &c); //1. Aufruf
  f(a, b, &c); //2. Aufruf
  return 0;
}
```

Antwort:

	a	b	c
vor 1. Aufruf	1	3	5
nach 1. Aufruf	6	3	-1
nach 2. Aufruf	5	3	5

Aufgabe II.3 Implementierung einer Telefonliste *(20 Punkte)*

In dieser Aufgabe soll eine Telefonliste implementiert werden, mit der für eine Menge von Mitarbeitern die Verwaltung der von ihnen geführten Telefongespräche organisiert werden kann.

Für jeden Mitarbeiter wird auf der Halde ein Datensatz vom Typ Data verwaltet, der den Namen und die Gesamtlänge der von diesem Mitarbeiter geführten Telefongespräche speichert. Der Typ Data ist wie folgt deklariert:

```
struct Data {
    char* name;     // Name des Mitarbeiters
    int minutes;    // Gesamtdauer aller Gespräche des Mitarbeiters
}
```

Das Programm Phonelist stellt Funktionen für das Einfügen und Löschen von Datensätzen für Mitarbeiter, die Zugriff auf diese Datensätze über einen Cursor, das Ändern der Gesamtdauer der Telefongespräche eines Mitarbeiters, das Sortieren der gesamten Datenstruktur. Phonelist hat folgende Schnittstelle:

```
PhonelistInit();           // Initialisiert die Liste

void PhonelistAdd (int newPersonalNo, char* newName, int newMinutes);
// Erstellt neuen Datensatz mit Datensätzen auf der Halde

void PhonelistRemove (int personalNo);
// Löscht Datensatz für angebenene Personalnummer

void PhonelistChangeMinutes(int personalNo, int newMinutes);
// Ändert im Datensatz für die Personalnummer die Gesprächsdauer

void PhonelistSort();
// Sortierfunktion nach Personalnummer

// Navigation über alle Einträge der Liste:
void PhonelistFirst();      // Cursor auf ersten Platz setzen
void PhonelistNext();       // Cursor auf nächsten Platz setzen
bool PhonelistHasValue();   // Test, ob der Cursor auf einem belegten
                            // Platz steht

int PhonelistGetPersonalNoAtCursor(); // Personalnummer bei Cursor
Data* PhonelistGetDataAtCursor();     // Datensatz bei Cursor
```

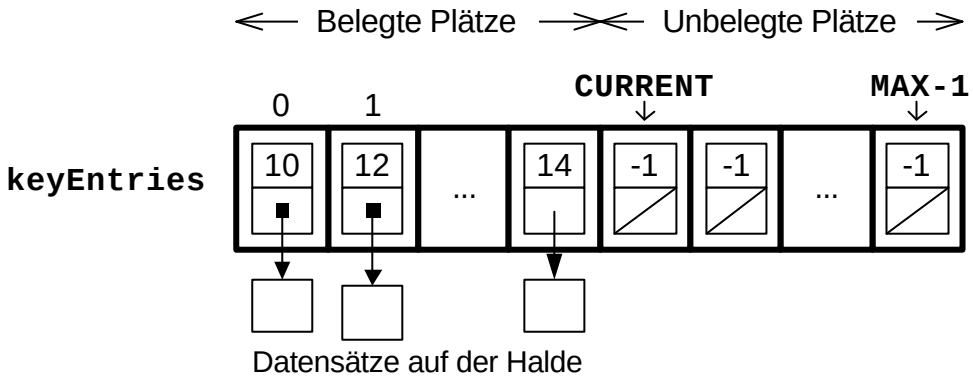
Intern soll Phonelist mit Hilfe eines Arrays keyEntries mit fest vorgegebener Größe MAX=100 realisiert werden (siehe Bild). Jeder Platz im Array keyEntries ist ein Verbund mit Namen Entry, der die Personalnummer personalNo vom Typ int und einen Zeiger data auf den Mitarbeiter-Datensatz (struct Data) mit den weiteren Angaben Name und Dauer der Telefongespräche speichert. Die Datensätze der Mitarbeiter werden auf der Halde abgelegt.

Unbenutzte Array-Plätze sind durch die Verbundwerte {personalNo= -1, data=NULL} gekennzeichnet.

Im Array sollen stets nur die Plätze mit Index i < CURRENT belegt sein. Eine Variable CURRENT vom Typ int soll den Index des ersten unbelegten Platzes speichern.

Die Funktion sort sortiert das Array keyEntries aufsteigend nach der Personalnummer.

Mit PhonelistFirst und PhonelistNext kann eine Markierung (CURSOR) auf einen beliebigen Platz im Array positioniert werden. Für diesen Platz kann mit PhonelistHasValue überprüft werden, ob er mit einem gültigen Datensatz belegt ist. Danach können Personalnummer und der zugehörige Datensatz mit PhonelistGetPersonalNoAtCursor und PhonelistGetDataAtCursor abgefragt werden.



(a) Ergänzen Sie die nötigen Deklarationen der beschriebenen internen Datenstrukturen im privaten Teil der Schnittstelle von Phonelist. *(3 Punkte)*

```
typedef struct EntryT {
  int personalNo;
  Data* data;
};
EntryT keyEntries[MAX];
int CURRENT;
```

(b) Implementieren Sie die Funktion PhonelistInit. Darin werden die in Teilaufgabe (a) deklarierten Datenstrukturen geeignet initialisiert. *(5 Punkte)*

```
PhonelistInit(){
  EntryT entry;              //Lx Dekl.+Belegung
  entry.personalNo= -1;
  entry.data=NULL;
  for (int i=0; i <MAX; i++){   //Lx Zuweisungsschleife
    keyEntries[i]= entry;
  }
  CURRENT=0;
  CURSOR=0;
}
```

(c) Implementieren Sie die Funktion PhonelistAdd für das Einfügen eines Eintrags für einen Mitarbeiter in die Liste. Es darf zur Vereinfachung davon ausgegangen werden, daß Eintragungen mit derselben Personalnummer nicht vorkommen. Einfügen eines Eintrags soll nur möglich sein, solange noch ein unbelegter Platz vorhanden ist. Es wird keine Sortierung beim Einfügen verlangt. Denken Sie daran, dass der Eintrag erzeugt werden muss. *(6 Punkte)*

```
void PhonelistAdd(int persNo, char* name, int minutes){
  if (CURRENT < MAX) {                //Lx Indexabfrage
    EntryT entry;                     //Lx Dekl+Belegung
    entry.personalNo= persNo;
    entry.data= new Data();
    entry.data->name=name;
    entry.data->minutes=minutes;
    keyEntries[CURRENT]= entry; //Lx Einfügen bei CURRENT
    CURRENT=CURRENT+1;              //Lx Weitersetzen CURRENT
  }
}
```

(d) Implementieren Sie die Funktion `PhonelistSort` mit Hilfe von Bubblesort. Ergänzen Sie den fehlenden Quellcode (Vergleich und Vertauschung zweier Array-Einträge). Der Tausch findet statt, falls die Personalnummer von `e2` kleiner ist als die Personalnummer vom Eintrag `e1`. *(4 Punkte)*

```
void PhonelistSort(){
 bool getauscht;
  do{
    getauscht= false;
    // Schleife bis CURRENT vermeidet Sortieren unbelegter Plätze
    for (int i=0; i<CURRENT-1; i++){
      Entry e1= keyEntries[i];
      Entry e2= keyEntries[i+1];
      // [Hier fehlenden Quellcode]
      if (e2.personalNo < e1.personalNo){ //Lx Vergleich
        keyEntries[i]  = e1;
        keyEntries[i+1]= e2;              //Lx Tausch
        getauscht=true;                   //Lx Variable gesetzt
      }
    }
  } while (getauscht);
}
```

(e) Was kann bei der Bubblesort-Formulierung von (d) verbessert werden? *(2 Punkte)*
Man könnte sich die Stelle der letzten Vertauschung merken. Im nächste Durchlauf muss die Schleife dann nur noch bis dort laufen.

Aufgabe II.4 Doppelt verkettete Liste, Suchen und Sortieren *(20 Punkte)*

In dieser Aufgabe sollen einige Funktionen für eine doppelt verkettete Liste implementiert werden. Das folgende Listing zeigt die Header-Datei der Liste:

```
// LinList.h

struct ListElem {
   int value;
   ListElem* next;
   ListElem* prev;
};

ListElem *head;

void init();
void append(int value);
ListElem* find(int value);
void sort();
```

Die Struktur `ListElem` dient als Typ für die einzelnen Listenelemente. In der Liste werden Integer-Werte (`value`) gespeichert. `prev` zeigt auf das vorherige Element oder beim ersten Element auf `NULL`, `next` auf das nächste Element bzw. auf `NULL` beim letzten Element. Der Zeiger `head` verweist auf das erste Element der Liste oder auf `NULL`, falls die Liste leer ist.

(a) Implementieren Sie die Prozedur `append`, die ein neues Listenelement mit dem übergebenen Wert erzeugt und hinten an die Liste anhängt. Beachten Sie den Sonderfall, dass die Liste leer ist. *(7 Punkte)*

```
void append(int wert) {
   ListElem *newElem; //Lx
   // neues Element initialisieren //Lx
   newElem=new ListElem; //Lx
   newElem->next=NULL;   //Lx
   newElem->value=wert; //Lx

   // neues Element hinten anhängen //Lx
   if (head==NULL) { //Lx
      //erster Fall: Liste ist leer //Lx
      head=newElem;  //Lx
      newElem->prev=NULL; //Lx
   } else { //Lx
      // zweiter Fall: Liste enthält mind. ein Element //Lx
      ListElem* cursor; //Lx
      cursor=head; //Lx
      // laufe zu letztem Element //Lx
      while (cursor->next!=NULL) { //Lx
         cursor=cursor->next; //Lx
      } //Lx
      cursor->next=newElem; //Lx
      newElem->prev=cursor; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(b) Implementieren Sie die Funktion `find`, die das erste Auftreten des übergebenen Werts in der Liste findet und einen Zeiger auf das entsprechende Listenelement zurückliefert. *(3 Punkte)*

```
ListElem* find(int value) {
   ListElem* cursor=head; //Lx
   while (cursor!=NULL) { //Lx
      if (cursor->value==value) break; //Lx
      cursor=cursor->next; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   return cursor;
}
```

(c) Ergänzen Sie die Implementierung der Prozedur `sort`, die die Liste mittels Bubblesort aufsteigend sortiert. Die zu ergänzende Stelle ist im Kommentar angegeben, es ist die Vertauschung der Elemente zu implementieren. Ändern Sie die Verzeigerung der Elemente, tauschen Sie also nicht die Werte aus! Beachten Sie die Spezialfälle, dass eins der zu tauschenden Elemente am Anfang bzw. am Ende der Liste steht. *(5 Punkte)*

```
void sort() {
   bool changed;
   do {
      ListElem* cursor=head;
      changed=false;
      while(cursor->next!=NULL) {
         if (cursor->value > cursor->next->value) {
            // Verkettung aendern
            ListElem *el0,*el1,*el2,*el3;
            el0=cursor->prev;
            el1=cursor;
            el2=cursor->next;
            el3=el2->next;
            // el1 und el2 sind zu vertauschen, also neue Reihenfolge:
            // el0, el2, el1, el3
            // hier die Vertauschung ergänzen:

            if (el0!=NULL) { //Lx
               el0->next=el2; //Lx
            } else { //Lx
               head=el2; //Lx
            }//Lx
            el2->prev=el0; //Lx
            el2->next=el1; //Lx
            el1->prev=el2; //Lx
            el1->next=el3; //Lx
            if (el3!=NULL) el3->prev=el1; //Lx
            //Lx
            //Lx
            //Lx
            //Lx
            //Lx
            //Lx
            cursor=el2;
            changed=true;
         }
```

```
            cursor=cursor->next;
         }
      } while(changed);
   }
```

(d) Implementieren Sie die Prozedur `dump`, die die Liste auf der Standardausgabe ausgibt. Um zu verdeutlichen, dass es sich um eine Liste handelt, sollen die einzelnen Werte durch "->" getrennt werden. Am Ende der Liste soll kein Pfeil stehen, sondern in eine neue Zeile gesprungen werden. *(5 Punkte)*

```
void dump() {
   ListElem * cursor=head; //Lx
   while(cursor!=NULL) { //Lx
      cout << cursor->value; //Lx
      if (cursor->next!=NULL) { //Lx
         cout << " -> "; //Lx
      } else {//Lx
         cout << "\n";  //Lx
      } //Lx
      cursor=cursor->next; //Lx
   } //Lx
}
```