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

Klausur
Grundgebiete der Informatik I,II

(DPO98/04)

Datum: 21. 09. 2004

Teil II: Algorithmen und Programmiertechniken

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

Aufgabe	Max. Punkte	Korrektur		Einsichtnahme	
		Punkte	Kürzel	Punkte	Kürzel
II.1	8				
II.2	12				
II.3	20				
II.4	20				
Σ	60				

Aufgabe II.1 Wissensfragen *(8 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 II.1 nicht unterschritten werden.

- Welche der folgenden Aussagen für den Zusammenhang zwischen EBNF und Syntaxdiagrammen ist richtig? *(1 Punkt)*

  - ☐ Syntaxdiagrammen beschreiben die kontextsensitive Syntax und nicht die kontextfreie wie die EBNF, da sich ihre Graphstruktur hierfür besser eignet.
  - ☒ Es gibt eine 1:1 Entsprechung zwischen Syntaxdiagrammen und der EBNF.
  - ☐ Es gibt keinen Zusammenhang zwischen Syntaxdiagrammen und der ENBF.

- Welchen Aspekt einer Programmiersprache beschreibt die kontextsensitive Syntax? *(1 Punkt)*

  - ☐ Sie beschreibt den Aufbau eines Programms.
  - ☐ Sie beschreibt den Aufbau von lexikalischen Einheiten.
  - ☒ Sie beschreibt unter Anderem den Zusammenhang von Variablen und ihrer Verwendung.

- Warum wird bei großen Feldern und Eingangsparametern Call-by-Reference verwendet? Worauf muss dann selbst geachtet werden? *(2 Punkte)*

  Effizienz. Das Feld müsste sonst kopiert werden. Das Feld darf nicht verändert werden.

- Nennen sie zwei Gefahren, die die Verwendung von Zeigern mit sich bringt? *(1 Punkt)*

  unübersichtliche Strukturen, Aliasing, nicht mehr ansprechbare Objekte, hängende Zeiger

- Geben Sie die Komplexität der binären Suche in einem sortierten Feld mit n Elementen an. *(1 Punkt)*

  - ☐ $O(\log n^2)$
  - ☒ $O(\log n)$
  - ☐ $O(n^2)$
  - ☐ $O(n \log n)$

- In welcher Reihenfolge wird ein Baum bei dem Preorder-Durchlauf durchlaufen? *(1 Punkt)*

- ☐ Linker Teilbaum, Wurzel, Rechter Teilbaum
- ☐ Linker Teilbaum, Rechter Teilbaum, Wurzel
- ☒ Wurzel, Linker Teilbaum, Rechter Teilbaum

- Wie aufwändig ist das Einfügen in einen ausgeglichenen binären Suchbaum, wenn dieser n Elemente enthält? *(1 Punkt)*

  - ☒ $O(\log n)$
  - ☐ $O(n)$
  - ☐ $O(2n)$
  - ☐ $O(n \log n)$

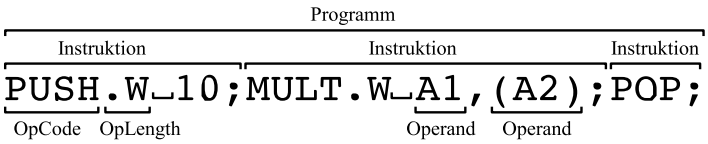
Aufgabe II.2 Syntaxbeschreibung *(12 Punkte)*

Für die Programmierung von Mikrocontrollern wird im Allgemeinen eine Maschinensprache verwendet. Im Gegensatz zu herkömmlichen Programmiersprachen wie C++ haben Maschinensprachen eine viel eingeschränktere Struktur.

Die Syntax unserer Maschinensprache sei wie folgt charakterisiert:

- Ein Maschinencode-Programm besteht aus einer beliebigen Anzahl von Instruktionen (Instruktion).
- Jede Instruktion ist wie folgt aufgebaut:
  1. Die Instruktion selber wird durch einen OpCode beschrieben. Ein OpCode ist eine Folge von Buchstaben mit der Mindestlänge 1.
  2. Hinter dem OpCode kann optional mit “.” getrennt die Wort-Länge (OpLength) für die Instruktion angegeben werden. Zulässige Wort-Längen sind: Byte “B”, Word “W” und Long “L”.
  3. Jede Instruktion null, einen oder zwei Operanden besitzen. Vor dem ersten Operanden muss ein Leerzeichen stehen. Zwei Operanden werden durch “,” getrennt.
- Ein Operand kann wie folgt definiert sein:
  - Bezeichner: identifier (wie ein C++-Bezeichner)
  - Integer-Wert: integer (wie ein ganzzahliges Literal)
  - Indirekter Zugriff auf eine Register durch Klammerung des Registerbezeichners: "(" identifier ")"
- Instruktionen werden mit “;” abgeschlossen.

Unten stehende Grafik zeigt ein Beispiel für ein Maschinencode-Programm. In dieser Grafik wurden die einzelnen Elemente der Sprache zur Verdeutlichung hervorgehoben.



(a) Geben Sie die EBNF für die oben beschriebene Maschinensprache an. Die Nichtterminalsymbole **letter**, **identifier** und **integer** sind bereits definiert und somit gegeben. *(6 Punkte)*

Programm ::= { Instruktion }
Instruktion ::= OpCode ["." OpLength] [" " Operand ["," Operand]] ";"
OpCode ::= letter { letter }
OpLength ::= "B" | "W" | "L"
Operand ::= identifier | integer | "(" identifier ")"

(b) Welche der folgenden Maschinensprachen-Instruktionen sind korrekt? Begründen Sie Ihre Antwort kurz, wenn eine Instruktion nicht korrekt ist. *(4 Punkte)*

1. MOVE.W (A0),D1;
   Korrekt.
2. ADD D0,D1;MOVE.D0;
   Falsch. Nach MOVE. muss Länge kommen.
3. PUSH.W (1);
   Falsch. Anstelle der 1 wird ein **identifier** erwartet.
4. PUSH.W 1,(AO);
   Korrekt.

(c) Eine nach obiger Beschreibung erstellte EBNF hat den Nachteil, dass ungültige Maschinencode-Programme erstellt werden können. Beispielsweise können OpCodes oder Registerbezeichner benutzt werden, die nicht existieren. Beschreiben Sie kurz umgangssprachlich, wie sich dieses Problem bereits in der EBNF vermeiden lässt. *(2 Punkte)*

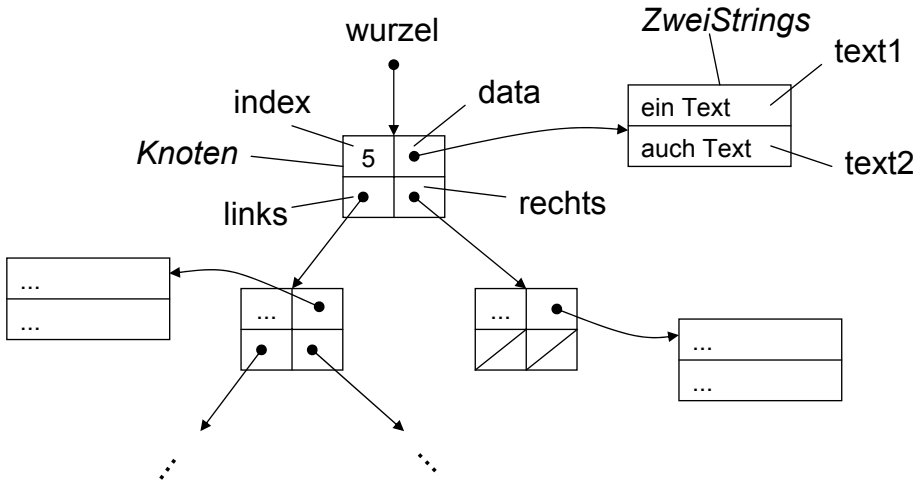
Für OpCodes und für Registerbezeichner jeweils eine EBNF angeben, die die zulässigen Möglichkeiten aufzählt.

Aufgabe II.3 Realisierung des ADT BinBaumWithData *(20 Punkte)*

In dieser Aufgabe soll ein Binärbaum implementiert werden, der in jedem Knoten neben einer ganzen Zahl als Bezeichnung einen Zeiger auf ein zusammengesetztes Datenfeld auf der Halde enthält. Der Baum soll als ADT realisiert werden. Er bietet an der Schnittstelle folgende Operationen zum Anfragen und Manipulieren der Baumstruktur an:

```
class BinBaumWithData {
private:
   struct Knoten; //Definition erfolgt in Aufgabenteil a)
   //...
public:
   // zum identifizieren einzelner Knoten durch den Verwender
   typedef Knoten* KnotenRef;

   BinBaumWithData(); //Konstruktor

   //legt einen neuen Knoten mit den übergebenen Daten an und liefert
   //diesen als KnotenRef zurück
   KnotenRef erzeugeKnoten(int index, string text1, string text2);

   //liefert den index sowie text1 und text2 des knoten
   void gibKnotendaten(KnotenRef knoten, int& index, string& text1, string& text2);

   //liefert den index des knoten
   int gibIndex(KnotenRef knoten);

   //true, wenn die Wurzel existiert, d.h. der Baum nicht leer ist
   bool hatWurzel();

   //trägt den knoten als Wurzel ein
   void setzeWurzel(KnotenRef knoten);

   //gibt die Wurzel zurück
   KnotenRef gibWurzel();

   //true, wenn das entsprechende Kind vorhanden ist
   bool hatLinkesKind(KnotenRef knoten);
   bool hatRechtesKind(KnotenRef knoten);

   //liefert das entsprechende Kind von vater
   KnotenRef gibLinkesKind(KnotenRef vater);
   KnotenRef gibRechtesKind(KnotenRef vater);

   //trägt neuesKind als das entsprechende Kind von vater ein
   bool einhaengenAlsLinkesKind(KnotenRef vater, KnotenRef neuesKind);
   bool einhaengenAlsRechtesKind(KnotenRef vater, KnotenRef neuesKind);
};
```

Die folgende Abbildung verdeutlicht die interne Realisierung des ADT: Ein Baumknoten (Typ Knoten) enthält ein Element index vom typ int, jeweils einen Zeiger auf das linke und das rechte Kind (links und rechts) und einen Zeiger (data) auf das Datenfeld. Dieses Datenfeld (Typ ZweiStrings) enthält zwei Zeichenketten vom Typ string: text1 und text2. Die private Variable wurzel des ADT zeigt auf die Wurzel des Baumes.



Hinweis zum Datentyp string:

Einer Variablen vom Datentyp string können mit = Zeichenkettenliterale beliebiger Länge zugewiesen werden, aber auch der Wert anderer Variablen vom Typ string. Beispiele:

```
string s="Hallo"; string t=s;
```

(a) Definieren Sie die Datentypen **ZweiStrings** und **Knoten** und den Wurzelzeiger (**wurzel**). Verwenden Sie die Bezeichner aus der Abbildung. *(6 Punkte)*

```
Knoten* wurzel; //Lx
struct ZweiStrings {//Lx
   string text1; //Lx
   string text2; //Lx
}; //Lx

struct Knoten { //Lx
   int index; //Lx
   Knoten* links; //Lx
   Knoten* rechts; //Lx
   ZweiStrings* data; //Lx
}; //Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
//Lx
```

(b) Implementieren Sie den Konstruktor, der die interne Datenstruktur geeignet initialisiert. *(2 Punkte)*

```
BinBaumWithData::BinBaumWithData()
{
   wurzel=NULL; //Lx
   //Lx  für Einfachheit erkannt
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(c) Die Funktion **erzeugeKnoten** erzeugt einen neuen Knoten und initialisiert ihn mit den übergebenen Werten. Beachten Sie, dass ein neuer Knoten zunächst noch nicht in den Baum eingehängt ist. Der erzeugte Knoten soll als **KnotenRef** zurückgegeben werden. *(5 Punkte)*

```
KnotenRef BinBaumWithData::erzeugeKnoten (int index, string text1,
                                          string text2)
{
   Knoten* pKnoten=new Knoten; //Lx
   pKnoten->index=index; //Lx s.u.
   pKnoten->links=NULL; //Lx
   pKnoten->rechts=NULL; //Lx
   pKnoten->data=new ZweiStrings; //Lx
   pKnoten->data->text1=text1; //Lx
   pKnoten->data->text2=text2; //Lx
   return pKnoten; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(d) Implementieren Sie die Prozedur **gibKnotendaten**, die alle Daten eines Knotens (**index**, **text1**, **text2**) über Referenzparameter zurückgibt. Gehen Sie davon aus, dass **knoten** auf einen gültigen Knoten verweist. Eine Prüfung auf **NULL** ist nicht notwendig. *(3 Punkte)*

```
void BinBaumWithData::gibKnotendaten (KnotenRef knoten, int& index,
                                      string& text1, string& text2)
{
   index=knoten->index; //Lx
   text1=knoten->data->text1; //Lx
   text2=knoten->data->text2; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(e) Die Funktion **hatLinkesKind** gibt **true** zurück, wenn der übergebene Knoten ein linkes Kind hat. Geben Sie die Implementierung dieser Funktion an. Gehen Sie davon aus, dass **knoten** auf einen gültigen Knoten verweist. *(1 Punkt)*

```
bool BinBaumWithData::hatLinkesKind(KnotenRef knoten) {
   return knoten->links!=NULL; //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(f) Implementieren Sie die Prozedur **einhaengenAlsLinkesKind**, die den übergebenen Knoten **knoten** als linkes Kind des Knoten **vater** einträgt, falls dieser noch kein linkes Kind hat. Im Erfolgsfall soll **true** zurückgegeben werden, andernfalls **false**. Verwenden Sie die Funktion **hatLinkesKind**. Gehen Sie davon aus, dass sowohl **vater** als auch **neuesKind** auf einen gültigen Knoten verweisen. *(3 Punkte)*

```
bool BinBaumWithData::einhaengenAlsLinkesKind(KnotenRef vater,
                                              KnotenRef neuesKind) {
   if (hatLinkesKind(vater)) { //Lx
      return false; //Lx
   } else { //Lx
      vater->links=neuesKind; //Lx
      return true; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

Aufgabe II.4 Realisierung eines Lexikons für Personaldaten als ADO *(20 Punkte)*

In dieser Aufgabe soll – basierend auf dem ADT DataBinBaum aus der vorherigen Aufgabe – ein Lexikon für das Personal einer Firma implementiert werden (ADO PersonalLexikon). Ein Lexikon ist eine Datenstruktur mit Einträgen (Schlüssel, Info) wobei Info strukturiert sein kann. Hinweis: Die Aufgabe kann ohne Kenntnis der Lösung der letzten Aufgabe gelöst werden.

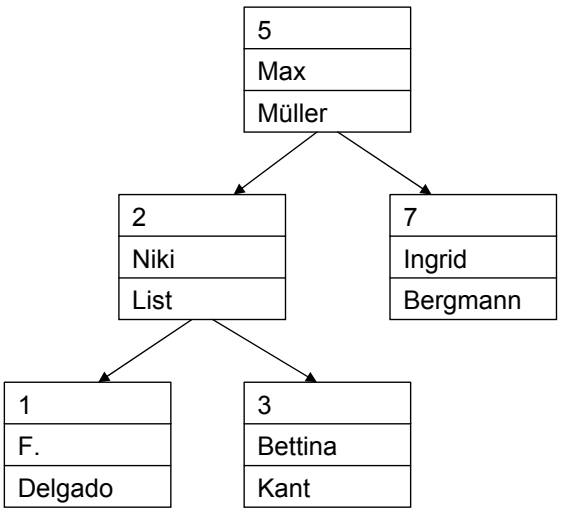
Jede Person wird über eine id vom Typ int identifiziert. Als Info sollen hier der Einfachheit halber nur der Name und der Vorname abgelegt werden.

Der ADO PersonalLexikon hat folgende Schnittstelle:

```
void speichereDatensatz(int id, string name, string vorname);
bool holeDatensatz(int id, string& name, string& vorname);
bool loescheDatensatz(int id);
```

Da der Zugriff immer anhand der Id erfolgt, bietet sich die Speicherung in einem binären Suchbaum mit id als Sortierkriterium an. Zur Realisierung ist der ADT BinBaumWithData aus der vorherigen Aufgabe zu verwenden. Das heisst, dass der ADO im Rumpf eine Instanz von BinBaumWithData enthält, und die Operationen der Schnittstelle des Lexikons mit Hilfe der Operationen von der BinBaumWithData-Instanz realisiert werden (siehe Zusatzblatt). Dabei werden die id als index, der name als text1 und der vorname als text2 im binären Baum gespeichert.

Das folgende Beispiel zeigt den internen Aufbau. Beachten Sie, dass hier nicht eine direkte Implementierung mit Zeigern Im Rumpf erfolgen soll, sondern die Daten in der lokalen Instanz von BinBaumWithData gespeichert werden.



(a) Zunächst soll die Anwendung des Lexikons verdeutlicht werden. Ergänzen Sie das Hauptprogramm so, dass nacheinander die Datensätze mit der id 5 und 2 aus der Abbildung im Lexikon abgespeichert werden. *(2 Punkte)*

```
int main() {
   speichereDatensatz(5, "Müller", "Max"); //Lx
   speichereDatensatz(2, "List", "Niki"); //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   return 0;
}
```

(b) Geben Sie die Deklaration der lokalen Instanz von BinBaumWithData an. *(1 Punkt)*

```
BinBaumWithData baum; //Lx
//Lx
//Lx
//Lx
```

(c) Definieren Sie den Aufzählungstypen ergebnis, dessen Wertebereich aus den Werten gefunden, links und rechts besteht. *(1 Punkt)*

```
typedef enum ergebnis {gefunden, links, rechts}; //Lx
//Lx
//Lx
//Lx
//Lx
```

(d) Implementieren Sie eine lokale Funktion **findeKnoten** für den Rumpf von **PersonalLexikon**, die im binären Suchbaum, beginnend bei der Wurzel, den Knoten mit der **id** sucht. Sie liefert im Referenzparameter **knoten** den Knoten zurück, wenn sie ihn gefunden hat. Der Rückgabewert ist dann **gefunden**. Andernfalls liefert sie den Knoten, unterhalb dessen der Knoten als linkes (Rückgabewert **links**) bzw. rechtes Kind (**rechts**) einzufügen wäre. Sie können davon ausgehen, dass die Wurzel existiert. Die Funktion ist iterativ zu formulieren. *(11 Punkte)*

```
ergebnis findeKnoten(int id, BinBaumWithData::KnotenRef &knoten) {
   BinBaumWithData::KnotenRef aktuell=baum.GibWurzel(); //Lx

   while (true) { //Lx
      if (id < baum.gibIndex(aktuell)) { //Lx
         //linker Teilbaum //Lx
         if (baum.hatLinkesKind(aktuell)) { //Lx
            aktuell= baum.gibLinkesKind(aktuell); //Lx
         } else { //Lx
            knoten=aktuell; //Lx
            return links; //Lx
         } //Lx
      } else if (index==baum.gibIndex(aktuell)){ //Lx
         knoten=aktuell; //Lx
         return gefunden; //Lx
      } else if (index>baum.gibIndex(aktuell)){ //Lx
         //rechter Teilbaum //Lx
         if (baum.hatRechtesKind(aktuell)) { //Lx
            aktuell=baum.gibRechtesKind(aktuell); //Lx
         } else { //Lx
            knoten=aktuell; //Lx
            return rechts; //Lx
         } //Lx
      } //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
```

```
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```

(e) Implementieren Sie die Funktion **holeDatensatz** von **PersonalLexikon**, die den Knoten mit der **id** sucht und ggf. in den Referenzparametern **name** und **vorname** dessen Daten zurückgibt. Wurde der Knoten gefunden, gibt die Funktion **true** zurück, sonst **false**. Verwenden Sie die lokale Funktion **findeKnoten**. Beachten Sie den Sonderfall, dass der Baum leer ist. *(5 Punkte)*

```
bool holeDatensatz(int id, string& name, string& vorname) {
   BinBaumWithData::KnotenRef knoten; //Lx

   if (!baum.hatWurzel()) return false; //Lx

   ergebnis erg=findeKnoten(id, knoten); //Lx
   if (erg==gefunden) { //Lx
      baum.gibKnotendaten(knoten, id, name, vorname); //Lx
      return true; //Lx
   } else { //Lx
      return false; //Lx
   } //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
   //Lx
}
```