Institut 186 für Computergraphik	**Algorithmen und Datenstrukturen 1**				VO-Prüfung 5. Dezember 1997 Gruppe C	
Kennz.†	**Matrikelnr.**	**Beil.**‡	1 (25)	2 (20)	3 (25)	4 (30)
Nachname, Vorname						Σ (100)

† Geben Sie jene Kennzahl an, auf die das Zeugnis ausgestellt werden soll.
‡ Geben Sie an, wieviele Zusatzbl. Sie abgeben (**jedes mit Name & Matr.-Nr. beschriftet!**).

(25) Aufgabe 1 – Graphen

(15) **a)** Eine Telefongesellschaft verwendet folgenden Algorithmus zum Durchschalten von Verbindungen in einem Netzwerk von Telefonzentralen. Die Verbindungen zwischen den einzelnen Zentralen sind in einer modifizierten Adjazenzliste gespeichert. Die Zentralen sind von 0 bis N-1 durchnummeriert. In der für jede Zentrale assoziierten Liste sind aber nicht nur direkte Verbindungen gespeichert sondern zusätzliche Routinginformation. Diese Information gibt an, über welche Verbindung ein Datenpaket weitergeleitet werden soll, das eine bestimmte Zieladresse hat. Es wird angenommen, daß Zentralen mit ähnlichen Nummern nahe beisammen liegen, und somit Gespräche über die gleiche Verbindung vermittelt werden können. Daher ist mit jedem Eintrag in der Adjazenzliste ein Intervall von Kontennummern gespeichert (gegeben durch die Intervallgrenzen `first` und `last`). Die Liste ist nach aufsteigenden Intervallen sortiert. Die Untergrenze des ersten Intervalls ist 0. Die Nachbarzentrale ist durch `neighbour` bezeichnet.

```
TYPE  Pointer = ^Node;
      Node = RECORD
                 neigbour: INTEGER;    (* Nachbarzentrale *)
                 first,last: INTEGER;  (* Intervallgrenzen *)
                 next: Pointer;
             END;
VAR   routingInfo : ARRAY[0..N-1] of Pointer;
```

Schreiben Sie eine *rekursive* Prozedur

```
PROCEDURE printRoute(from,to:INTEGER);
```

die den durch die Routinginformation definierten Weg von der Zentrale `from` zur Zentrale `to` ausgibt. Alle Zentralen die auf dem Weg eines durchgeschalteten Telefonats sind sollen mit `write` ausgegeben werden. Die Intervalle decken `0..N-1` ab. Es ist *keine* Fehlerbehandlung erforderlich.

(10) **b)** Zeichen Sie ein Gerüst des folgenden Graphen:

(20) Aufgabe 2 – Familienstammbaum

Gegeben ist folgende Datenstruktur für einen Familienstammbaum:

```
CONST MaxChildren =...;
TYPE  PersonPtr = ^Person;
      Person = RECORD
                 name: STRING;
                 sibling: PersonenPtr;
 firstChild: PersonenPtr;
               END;
VAR   forefather : PersonPtr;
```

Durch das Feld `sibling` sind Geschwister jeweils in einer `nil`-terminierten Liste verkettet. Das Feld `firstChild` zeigt auf das erste Kind der Person, die weiteren Kinder sind über den `sibling`-Pointer des ersten Kindes erreichbar.

Schreiben Sie eine Prozedur

```
FUNCTION printDescendents(p : PersonPtr; num :INTEGER) : INTEGER;
```

die all diejeneigen Nachkommen der Person `p` mit `writeln` ausgibt, die genau `num` Kinder haben. *Hinweis*: Sie können die Anzahl der Geschwister (einschließlich der Person selbst), die eine Person hat, als Funktionswert zurückliefern.

(25) Aufgabe 3 – Kundendatenbank

In einer Hashtabelle mit äusserer Verkettung werden Kundendaten gespeichert. Zu jedem Kunden werden Kundennummer und Name gespeichert. Der Schlüssel des Datensatzes ist die Kundennummer. Die Hasthabelle enthält `NumSlots` Elemente `0..NumSlots-1`, wobei `NumSlots` eine Primzahl ist. Die Hashfunktion f ist gegeben und berechnet den Hashwert als **Ziffernsumme der letzten drei Ziffern der Kundennummer** modulo `NumSlots`.

(20) **a)** Tragen Sie die Datensätze der folgenden Kunden **in der angegebenen Reihenfolge** in eine Hashtabelle, wie sie oben beschrieben wurde, ein (`NumSlots = 7`). Verwenden Sie eine analoge Version der Prozedur `insert` aus dem Skriptum. **Zeichenen Sie die entstehende Hashtabelle!**

Einzutragende Datensätze:

55 669	Hugo Schneider
64 254	Hans Müller
42 090	Horst Winzer
83 242	Hubert Becker
76 747	Hannes Metzger
34 314	Hartmuth Mahler
22 594	Helwig Zimmermann
24 848	Heinrich Schlosser

(5) **b)** Welche Datenstruktur würden Sie verwenden, wenn Sie die Kunden nicht nur suchen müßten, sondern außerdem noch nach Knundennummer sortiert ausgeben?

(30) Aufgabe 4 – Objektorientierte Matrizen

In objektorientiertem Turbo Pascal sollen quadratische Matrizen als Objekte verwirklicht werden. Abhängig von Dimension und Besetung der Matrix soll der Benutzer eine effiziente Implementierung auswählen können. Bei jeder Matrix ist ihre Größe `size` als `INTEGER` Komponente eingetragen. Für Matrizen der Größe 10x10 und kleiner wird ein konstantes Array zur Implementierung verwendet. Für größere Matrizen wird angenommen, daß sie sehr dünn (*sparse*) besetzt sind. Daher werden alle von 0 verschiedenen Elemente expliziet in einer linearen Liste gespeichert. Dies führt zu folgendem objektorientiertem Entwurf:

Die Basisklasse `Matrix` gibt das Klassen-Interface vor: Mit der Methode `getValue` können die einzelnen Matrizenelemente gelesen werden.

Die Subklasse `SmallMatrix` verwaltet die Matrizenelemente in einem Array. Dabei kann angenommen werden, daß auf leere Elemente in diesem Array nicht zugegriffen wird (z.B. bei einer 3x3 Matrix wird `getValue(5,5);` nicht verwendet).

Die Subklasse `SparseMatrix` verwaltet Knoten der Hilfsdatenstruktur `Node` in einer einfachen, unsortierten linearen Liste, die bei `first` beginnt. Jeder Knoten dieser Liste enthält die beiden Indizes und den Wert des Elements. Für die Implementierung von `getValue`: ist ein Element nicht in der Liste, so ist es gleich 0 (siehe oben).

Implementieren Sie die Klassen `Matrix, SmallMatrix` und `SparseMatrix`, sowie deren Methoden `getValue`. Konstruktoren und Destruktoren müssen Sie nicht berücksichtigen.

Schreiben Sie außerdem eine Methode `addMatrix`, die zu einer Matrix eine andere Matrix der selben Größe elementweise addiert. Nehmen Sie dazu an, daß sie den Elementen einer Matrix mit der Methode `setValue` Werte zuweisen können.