Klausur Allgemeine Informatik II

SS 1999

Beispiel-Lösung

Name ______________________________

Vorname ______________________________

Matrikel-Nr. ______________________________

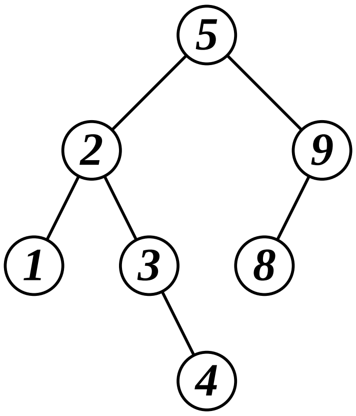
Studiengang ______________________________

Semester ______________________________

Aufgabe		Punkte	Bewertung
1	(binäre Bäume)	18	
2	(Ring)	18	
3	(EBNF)	12	
4	(Rekursion)	18	
5	(Hash-Verfahren)	16	
6	(Typ-Erweiterungen)	18	
Summe		**100**	

Aufgabe 1 (18 Punkte)

Gegeben sei folgender binärer Baum, bei dem ganze Zahlen als Schlüssel dienen:



(a) Bitte kreuzen Sie das jeweils zutreffende an:

Aussage	Korrekt	Falsch
Der Baum ist bezüglich der Höhe ausgeglichen.		
Der Baum ist nach dem Gewicht ausgeglichen.		
Der Baum ist sortiert.		

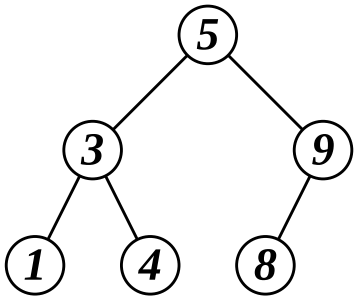
(b) Wie würde der Baum aussehen, wenn der Knoten mit der Nummer 2 gelöscht werden würde entsprechend der üblichen Löschprozedur für sortierte binäre Bäume?

(c) Zeichnen Sie auf, wie der vorgegebene Baum aussehen würde, wenn ein Element mit dem Schlüssel 7 hinzukommen würde.

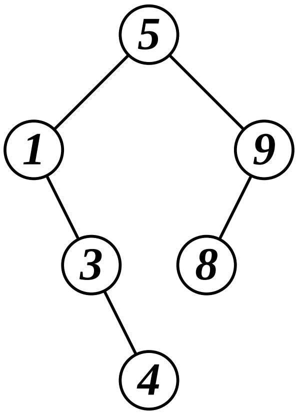
(d) Geben Sie die Schlüssel des Baumes in Postorder an.

(*a*) Der Baum ist bezüglich der Höhe ausgeglichen und sortiert, jedoch nicht nach dem Gewicht ausgeglichen.

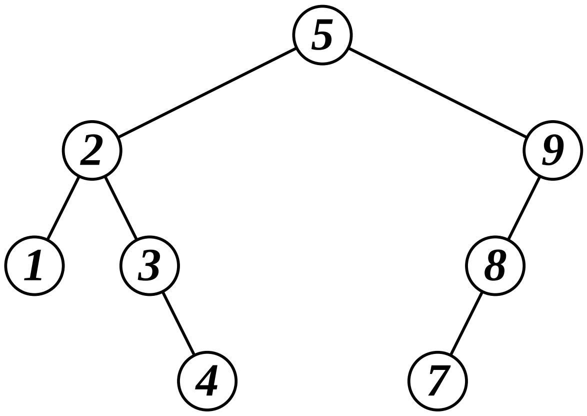
(*b*) Typischerweise wird das nächstgrößere Element hochgezogen (so wurde es in der Vorlesung vorgestellt):



Korrekt wäre es jedoch auch, das nächstniedrigere Element zu nehmen:



(*c*)



(*d*)

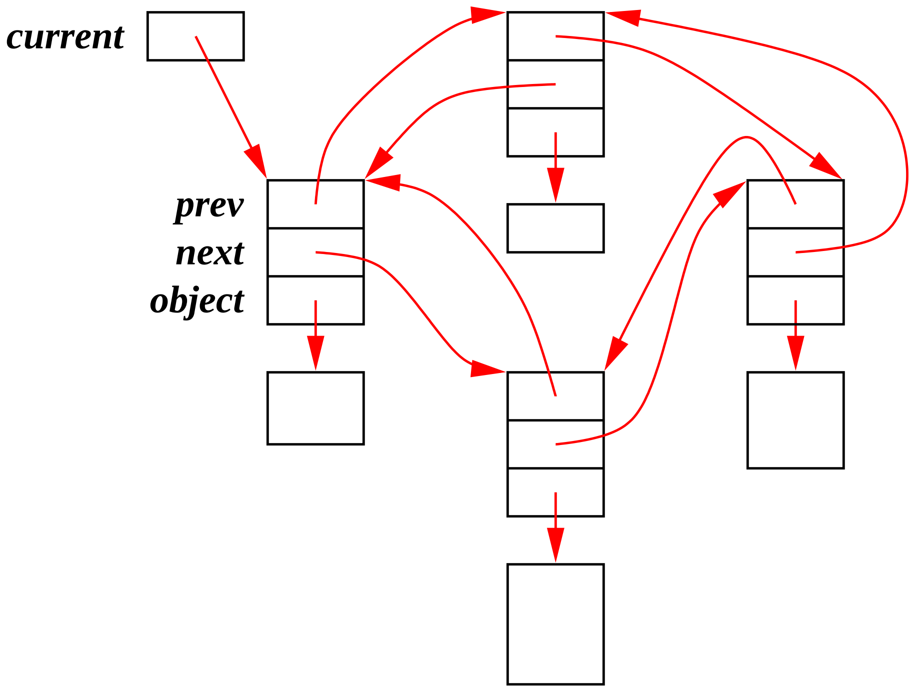
```
1 4 3 2 8 9 5
```

Aufgabe 2 (18 Punkte)

Gegeben sei die Datenstruktur eines Ringes:

```
TYPE
   Element = POINTER TO ElementRec;
   ElementRec =
      RECORD
         prev, next: Element;
         object: Objects.Object;
      END;
   Ring =
      RECORD
         current: Element;
      END;
```

Bei einem Ring sind alle Elemente ringförmig angeordnet. Im Unterschied zur doppelt verketteten Liste gilt somit für alle Elemente, daß sowohl *prev* und *next* jeweils ungleich **NIL** sind:



Bei einem Ring ohne Elemente wäre *current* gleich **NIL**.

Schreiben Sie eine Prozedur **Insert**, die ein neues Element an irgendeiner Stelle in den Ring einfügt:

```
PROCEDURE Insert(VAR ring: Ring; object: Objects.Object);
```

Hinweise: Sie können das neue Element vor *ring.current* oder hinter *ring.current* einfügen. Achten Sie aber darauf, daß *ring.current* auch **NIL** sein könnte.

Folgende Variante hängt das neue Element **vor** *ring.current* ein:

```
PROCEDURE Insert(VAR ring: Ring; object: Objects.Object);
   VAR
      element: Element;
BEGIN
   NEW(element); element.object := object;
   IF ring.current = NIL THEN
      element.next := element;
      element.prev := element;
      ring.current := element;
   ELSE
      element.next := ring.current;
      element.prev := ring.current.prev;
      element.prev.next := element;
      element.next.prev := element;
   END;
END Insert;
```

Alternativ wäre es natürlich auch denkbar, das neue Element **hinter** *ring.current* einzufügen. Dann wären die beiden ersten Zeilen im **ELSE**-Zweig durch folgende Zeilen zu ersetzen:

```
element.next := ring.current.next;
element.prev := ring.current;
```

Aufgabe 3 (12 Punkte)

Gegeben sei folgende Grammatik in EBNF mit *Expression* als Startsymbol:

```
Expression = List | Atom .
Atom = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
List = "(" [ ExpressionList ] ")" .
ExpressionList = Expression { "," Expression } .
```

Stellen Sie bei folgenden Teilaufgaben jeweils fest, ob es sich um einen Satz dieser Grammatik handelt. Wenn ja, sollten Sie eine mögliche Ableitung angeben. Leerzeichen sind dabei zu ignorieren.

(*a*) `1`

(*b*) `()`

(*c*) `(1)`

(*d*) `(1 2)`

(*e*) `(1, 2)`

(*f*) `(1, 2), 3`

(*a*) Ist ein Satz. Ableitung:

```
Expression
Atom
"1"
```

(*b*) Ist ein Satz. Ableitung:

```
Expression
List
"(" ")"
```

(*c*) Ist ein Satz. Ableitung:

```
Expression
List
"(" ExpressionList ")"
"(" Expression ")"
"(" Atom ")"
"(" "1" ")"
```

(*d*) Ist kein Satz.

(*e*) Ist ein Satz. Ableitung:

```
Expression
List
"(" ExpressionList ")"
"(" Expression "," Expression ")"
"(" Atom "," Expression ")"
"(" "1" "," Expression ")"
"(" "1" "," Atom ")"
"(" "1" "," "2" ")"
```

(*f*) Ist kein Satz.

Aufgabe 4 (18 Punkte)

Für den größten gemeinsamen Teiler gilt folgende rekursive Beziehung für $m, n \geq 0$:

- $ggt(0, n) = n$
- $ggt(m, n) = ggt(n \bmod m, m)$, falls $m > 0$

(*a*) Schreiben Sie eine rekursive Prozedur *GGT*, die direkt auf der angeführten Beziehung beruht.

(*b*) Ersetzen Sie die *tail recursion* in diesem Beispiel durch eine Schleife und formulieren Sie entsprechend eine iterative Version von *GGT*.

(*a*)

```
PROCEDURE GGT(m, n: INTEGER) : INTEGER;
BEGIN
   ASSERT((m >= 0) & (n >= 0));
   IF m = 0 THEN
      RETURN n
   ELSE
      RETURN GGT(n MOD m, m)
   END;
END GGT;
```

(*b*)

```
PROCEDURE GGT(m, n: INTEGER) : INTEGER;
   VAR
      nextm: INTEGER;
BEGIN
   ASSERT((m >= 0) & (n >= 0));
   WHILE m # 0 DO
      nextm := n MOD m;
      n := m;
      m := nextm;
   END;
   RETURN n
END GGT;
```

Aufgabe 5 (16 Punkte)

(*a*) Bitte kreuzen Sie die Eigenschaften an, die für eine gute Hash-Funktion wichtig sind und realistischerweise auch erwartet werden können:

Stetigkeit	
Surjektivität	
Injektivität	
Effiziente Berechenbarkeit	
Nicht-Determinismus	

(*b*) Für einen Geburtstagskalender haben Sie folgenden Datentyp definiert:

```
TYPE
   BirthDate = POINTER TO BirthDateRec;
   BirthDateRec =
      RECORD
         (Objects.ObjectRec)
         day: INTEGER; (* 1..31 *)
         month: INTEGER; (* 1..12 *)
         year: INTEGER; (* e.g. 1999 *)
         name: ARRAY 80 OF CHAR;
      END;
```

Geben Sie eine (gute!) Hash-Funktion für diesen Datentyp, die nur das Datum berücksichtigt (also Tag, Monat und Jahr) und eine (von der Tabellengröße unabhängige) ganze Zahl zurückliefert.

(*a*) Wichtig sind Surjektivität und effiziente Berechenbarkeit. Injektivität wäre erfreulich, kann jedoch realistischerweise nicht erwartet werden. Stetigkeit und Nicht-Determinismus wären fatal für eine Hash-Funktion.

(*b*) Folgende Version ist sehr effizient:

```
PROCEDURE HashByDate(birthdate: BirthDate) : INTEGER;
BEGIN
   RETURN birthdate.day +
          SYSTEM.LSH(birthdate.month - 1, 5) +
          SYSTEM.LSH(birthdate.year, 5 + 4)
END HashByDate;
```

Ohne Lücken im Wertebereich, jedoch etwas weniger effizient, wäre:

```
PROCEDURE HashByDate(birthdate: BirthDate) : INTEGER;
BEGIN
   RETURN birthdate.day +
          (birthdate.month - 1) * 32 +
          birthdate.year * 32 * 12
END HashByDate;
```

Natürlich ist es auch akzeptabel, wenn eine Hash-Funktion angegeben wird, die kompatibel zu *Hashes.HashFunction* aus der Vorlesung ist:

```
PROCEDURE HashByDate(object: Objects.Object) : INTEGER;
BEGIN
   WITH object: BirthDate DO
      RETURN (* ... *)
   END;
END HashByDate;
```

Aufgabe 6 (18 Punkte)

Gegeben seien folgende Typdefinitionen:

```
TYPE
   Partner = POINTER TO PartnerRec;
   PartnerRec = RECORD name, address, phone: ARRAY 80 OF CHAR END;
   Person = POINTER TO PersonRec;
   PersonRec = RECORD (PartnerRec) title: ARRAY 12 OF CHAR END;
   Employee = POINTER TO EmployeeRec;
   EmployeeRec = RECORD (PersonRec) office, department: ARRAY 16 OF CHAR END;
   Company = POINTER TO CompanyRec;
   CompanyRec = RECORD (PartnerRec) head: Employee END;
```

(*a*) Zeichnen Sie ein Diagramm, das die Typ-Hierarchie widerspiegelt.

(*b*) Bitte kreuzen Sie jeweils das Zutreffende an:

Aussage	Korrekt	Falsch
Employee ist eine Erweiterung von *Partner*		
Company ist eine Erweiterung von *Person*		
Person ist eine direkte Erweiterung von *Partner*		
Partner ist eine Erweiterung von *Company*		

Hinzu kommen folgende Deklarationen:

```
VAR partner: Partner; person: Person; employee: Employee; company: Company;

PROCEDURE Assign(VAR ptr: Partner; partner: Partner);
BEGIN
   ptr := partner;
END Assign;
```

Geben Sie bei folgenden Teilaufgaben jeweils an, ob der jeweilige Programmtext vom Compiler akzeptiert wird und ob er zur Laufzeit fehlerfrei durchläuft. Begründen Sie Ihre Angaben jeweils durch die Angaben der statischen und dynamischen Typen von *ptr* und *partner* innerhalb von *Assign*.

(*c*)

```
NEW(partner); NEW(employee); Assign(partner, employee);
```

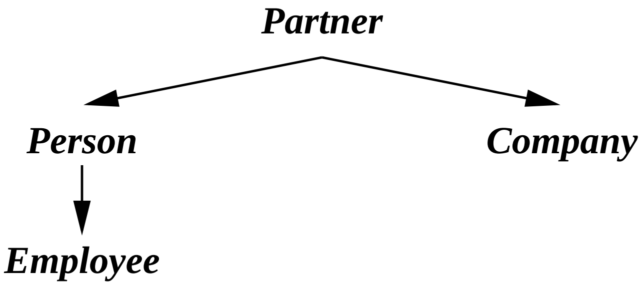
(*d*)

```
NEW(company); NEW(person); Assign(company, person);
```

(*e*)

```
NEW(employee); NEW(person); Assign(employee, person);
```

(*a*)



(*b*)

Aussage	Korrekt	Falsch
Employee ist eine Erweiterung von *Partner*	X	
Company ist eine Erweiterung von *Person*		X
Person ist eine direkte Erweiterung von *Partner*	X	
Partner ist eine Erweiterung von *Company*		X

(*c-e*)

Aufgabe	*ptr*		*partner*		Statisch OK?	Zur Laufzeit OK?
	statisch	dynamisch	statisch	dynamisch		
c	*Partner*	*Partner*	*Partner*	*Employee*	Ja	Ja
d	*Partner*	*Company*	*Partner*	*Person*	Ja	Nein
e	*Partner*	*Employee*	*Partner*	*Person*	Ja	Nein