WS 95/96

Algorithmen und Programmierung 3

Lösungen zur Klausur

Aufgabe 1

a)

Repräsentation: s. c)

b) *Modell:* Menge von Folgen der Länge $0 < n < maxLength$ über dem Alphabet ALPH={’A’...’Z’}

Abstraktionsfunktion :

abstr: KeysetT -> POT $\{<a_1...a_n> \mid a_j \in$ ALPH $\}$ (POT = Potenzmenge)

abstr(k) = {abstr’(b^.key) | b Blattknoten der Repräsentation}

und abstr’ : KeyT -> $\{<a_1...a_n> \mid a_j \in$ ALPH $\}$

ist die Funktion, die einem Feld von Zeichen des Alphabets auf kanonische Weise eine Folge zuordnet.

Repräsentationsinvariante

Für alle k : keyT, Länge (k) = n , gilt: T:trie enthält k gdw.

(es gibt einen Wurzelpfad T^.k[1] -> ... -> T^.k[m] -> b mit b ist Blattknoten und enthält k, m <= n)

oder

(es gibt einen Wurzelpfad T^.k[1] -> ... -> T^.k[n] -> c mit c ist kein Blattknoten und das letzte Element des Zeigerfelds von c verweist auf ein Blatt, das k enthält)

c)

```
DEFINITION MODULE
... s. Aufgabenstellung

IMPLEMENTATION MODULE Trie;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

CONST lastCh = ’[’, ; (* ’[’ folgt ’Z’ in der ASCII-Folge *)
      maxPos = 26; (* number of chars in alphabet +1 *)

(************** Repräsentation ************************)

TYPE   KeysetT = POINTER TO node;
        leafT   = KeysetT;
        ptrArrayT = ARRAY[’A‘..lastCh] OF KeysetT;
        node    = RECORD
                     CASE leaf: BOOLEAN OF
                          FALSE: ptrs: ptrArrayT |
                          TRUE: key: KeyT
                     END
                 END;

(*************** Hilfsprozeduren ********************)
(* fuer lauffaehiges Programm notwendig;             *)
(* konnten selbstverständlich vorausgesetzt werden,  *)
(* Ausnahme: makeLeaf (Blatt erzeugen)               *)

PROCEDURE length(k: KeyT): CARDINAL;
....
END length;
PROCEDURE stringCmp(k1,k2: KeyT): BOOLEAN;
...
END stringCmp;
```

```
PROCEDURE ch2alpha(x:CHAR):alphabet;  (* Typ Character in Typ alphabet *)
...
END ch2alpha;

PROCEDURE makeLeaf( k:KeyT): leafT;
(*pre: ks <> NIL  and ks^.leaf = FALSE *)
(*post: ks‘ = Zeiger auf neu erzeugte Seite s mit s^.leaf=TRUE *)
VAR tmp: KeysetT;
       i:   CARDINAL;
BEGIN
       NEW(tmp);
       tmp^.leaf:=TRUE;
       FOR i:=1 TO length(k) DO
              tmp^.key[i] := k[i];
       END;
       tmp^.key[i+1] := ch2alpha(CHR(0));
       RETURN tmp
END makeLeaf;

(******************************************************)

PROCEDURE create(): KeysetT;
VAR ks: KeysetT;
    c : CHAR;
BEGIN
       NEW(ks);
       ks^.leaf:=FALSE;
       FOR c:=‘A‘ TO lastCh DO
         ks^.ptrs[c] := NIL
       END;
       RETURN ks
END create;

PROCEDURE search(ks: KeysetT; k: KeyT) : BOOLEAN;
(* Pre: der Schluessel k genügt der Definition *)
(* Effekt: keiner;  val: true, wenn k gefunden, false sonst *)
VAR len: CARDINAL;

PROCEDURE searchSuffix(ks:KeysetT; first, last:CARDINAL): BOOLEAN;
       BEGIN
        IF ks <> NIL THEN
         CASE ks^.leaf OF
            TRUE: RETURN ( stringCmp(k,ks^.key)  )
            FALSE:
             IF first > last THEN
             RETURN searchSuffix(ks^.ptrs[lastCh], first,last)
            (* es gibt k‘ mit k‘[i] = k[i], i=1..length(k)     *)
            (* d.h. k echtes Prfix eines anderen Schluessels k‘*)
             ELSE RETURN searchSuffix(ks^.ptrs[k[first]], first+1,last)
            (* Weitersuchen mit Restsuffixk[first+1]..k[last]  *)
         END
        END  (*CASE *)
       ELSE RETURN FALSE END
END searchSuffix;
BEGIN (* search *)
   len:=length(k);
   IF len > 0 THEN RETURN searchSuffix(ks,1,len)
      ELSE RETURN FALSE  END
END search;
```

```
PROCEDURE insert(VAR entry: KeysetT; k: KeyT);
(* Pre: der Schlüssel k genügt der Definition, ks enthält weniger als   *)
(* die maximale Anzahl darstellbarer Schlüssel                           *)
(* Effekt: ks‘ = ks ∪ {k}                                         *)
(* die nachstehende Loesung verwendet fuer einen Schlüssel nur  *)
(* den minimal nötigen Wurzelpfad. Das ist der, der erlaubt,    *)
(* den Schlüssel von allen anderen vorhandenen  unterscheiden   *)
(* zu können. Der Schlüssel selbst befindet sich immer auch in  *)
(* einem Blatt, s. letzter Fall (ELSIF) in der Fallunterscheidung*)
(* der selbst aus drei Faellen besteht.                          *)
(* Beispiel: der Baum besteht nach Einfügen des ersten Schlüssels s *)
(* aus der Wurzel und einem Blatt, das s enthält und auf das der *)
(* s[1] entsprechende Zeiger des Wurzelknotens verweist.         *)
(* Eine Loesung von Aufgabe 1 ist auch dann korrekt, wenn fuer   *)
(* jeden Schluessel ein vollstaendiger Wurzelpfad angelegt       *)
(* wurde. Die 4. Fallunterscheidung (ELSIF) vereinfacht sich     *)
(* dann drastisch.                                                *)

PROCEDURE insertSuffix(VAR ksOld, ks:KeysetT; pos,last:CARDINAL);
(* ksOld: Zeiger auf Knoten des Baums, ks: Zeiger auf Nachfolger  *)
(* entsprechend Wurzelpfad des einzufügenden Schluessels k, also  *)
(* ks = NIL möglich! pos: pos des naechsten zu bearbeitenden      *)
(* Zeichens, last = Länge von k, 2<=pos<=last+1                   *)
VAR tmp: leafT;
       tmpPtr: KeysetT;
       kold: KeyT;
       l1,l2,i : CARDINAL;
BEGIN
        IF ks = NIL THEN
          (* kein existierender Schluessel hat gleiches Praefix *)
          (* -> Blatt erzeugen                                  *)
         ks := makeLeaf(k)
        ELSIF (NOT ks^.leaf) AND (pos <= last)
          (* es gibt k‘ mit k‘[i] = k[i] fuer i=1..pos-1        *)
          (* und pos <= last  -> Wurzelpfad weiterverfolgen     *)
        THEN insertSuffix(ks,ks^.ptrs [k[pos]],pos+1,last)
        ELSIF (NOT ks^.leaf) AND (pos > last) THEN
          (* es gibt k‘ mit k‘[i] = k[i], i=1..length(k)       *)
          (* und es gibt Verweisknoten v = ks^.ptrs[k‘[pos-1]  *)
          (* mit v^.ptrs[k‘[length(k)+1]] <> NIL               *)
          (* ->   k echtes Praefix eines anderen Schluessels   *)
         ks^.ptrs[lastCh] := makeLeaf (k)
        ELSIF
          (* es gibt k‘ mit k‘[i] = k[i], i=1..pos-1            *)
          (* und ks^.leaf  -> k‘=a(1)...a(pos-1)..a‘(pos-1+j)   *)
          (*                  k =a(1)...a(pos-1)..a (pos-1+m)   *)
          (* Fallunterscheidung: 1) j>m und a‘(i)=a(i), i>pos-1 *)
          (*                     2) m>j und ...                 *)
          (*   3) es gibt i<=j, i<=m und a‘(pos-1+l)=a(pos-1+l) *)
          (*                       und a‘(pos-1+i) <> a(pos-1+i) *)
          (* Entfaellt bis auf Duplikatsprüfung, wenn keine     *)
          (* minimalen sondern vollständige Schlüsselpräfixe als*)
          (* Wurzelpfadeverwendet werden                        *)
        ks^.leaf AND (pos-1 <= last) THEN
        IF NOT stringCmp(k,ks^.key) THEN  (* kein Duplikat    *)
         tmp:=ks;
            (* rette altes Blatt                        *)
```

```
        ks:=create();
        l1:= length(k);
        kold := tmp^.key;
        l2:= length(kold);
        tmpPtr := ks;
          (* Wurzelpfad verlängern, bis einer der Fälle zutrifft *)
        WHILE (pos <= l1) AND (pos <= l2)
              AND (k[pos] = kold[pos])   DO
                tmpPtr^.ptrs[k[pos]] := create();
                tmpPtr := tmpPtr^.ptrs[k[pos]];
              INC(pos);
         END;
         IF   (* Fall 1)  *)
           pos > l1 THEN
                (* k echtes Prfix des schon vorhandener Schluessels *)
                tmpPtr^.ptrs[kold[pos]] := tmp;
                tmpPtr^.ptrs[lastCh] := makeLeaf(k);
            ELSIF  (* Fall 2) *)
               pos > l2 THEN
                 (* vorhandener Schluessel ist echtes      *)
                 (*  Praefix  von k                        *)
                tmpPtr^.ptrs[k[pos]] :=makeLeaf(k);
                tmpPtr^.ptrs[lastCh] :=tmp
             ELSIF    (* Fall 3) *)
                k[pos] <> kold[pos] THEN
                tmpPtr^.ptrs[kold[pos]] := tmp;
                (* altes Blatt anfuegen *)
                tmpPtr^.ptrs[k[pos]] := makeLeaf(k)
                (* Blatt von k          *)
              END (* IF : Blattbehandlung  *)
         END (* Duplikate werden ignoriert *)

      END (* Fallunterscheidung *);
  END insertSuffix;

 BEGIN (* insert *)
       (* pre: entry <> NIL AND  NOT entry^leaf       *)
      insertSuffix (entry, entry^.ptrs[k[1]],2,length(k))
END insert;


BEGIN
END Trie.
```

Aufgabe 2

a)

Signatur etc. in Aufgabe gegeben.

Modell: Menge der mxm - Matrizen über |N .

Semantik:

```
        erzeuge() = ( 0 )i,j    (Nullmatrix)
        lege (kw, i,j) = kw' mit kw'[i,j] = kw[i,j]+1 und
                         kw'[x,y] = kw[x,y] für x <> i, y <> j
        entferne(kw,i,j)  = kw wenn kw[i,j]  = 0,
                          = kw' mit kw'[i,j] = kw[i,j] - 1 und
                                    kw'[x,y] = kw[x,y] für x <> i, y <> j
```

```
DEFINITION MODULE Klotz;
(* Blockwelt,, objektorientiert implementiert, um Spezialfall von         *)
(* Aufgabenteil c) leicht durch Redefinition einer Operation zu erledigen *)
(* modul-basierte Lösung selbstverständlich auch ok                       *)
(* Repräsentationsinvariante:                                             *)
(* für jedes Objekt bw:blockwelt, 0<= i,j <= max gilt:                    *)
(* 0 <= bw.welt[i,j] <= maxHoehe                                          *)

FROM StckCl1 IMPORT StckArr, maxElements;
(* Ein Stapel mit geeignetem Basistyp kann vorausgesetzt werden *)
CONST max = 6; (* 6x6-Welt  *)
      maxHoehe = maxElements;
TYPE blockTyp = (pyramid, cube);

CLASS  blockwelt;
  (* privat ! *)
  welt: ARRAY[1..max] OF ARRAY [1..max] OF StckArr;
  (* öffentlich *)
  PROCEDURE erzeuge() : blockwelt;
  (* erzeugt ein neues Objekt , nur in modulorientierter *)
  (* Implementierung wichtig                             *)
  PROCEDURE lege(x,y: CARDINAL): BOOLEAN;
  (* pre: b := create(), 0 <= x,y <= max  *)
  (*  AND welt[x,y].Hoehe < maxHoehe      *)
  (* post: block an pos x,y gelegt        *)
  (* val: TRUE wenn erfolgreich           *)

  PROCEDURE entferne (x,y: CARDINAL):BOOLEAN;
  (* pre: b := create(), 0 <= x,y <= max  *)
  (* post: block an pos x,y gelegt        *)
  (* val: TRUE wenn mind. ein Block an Pos x,y gelegen hat   *)
  (*      und entfernt wurde *)
  END blockwelt;

(**** Aufgabe 2,  Teil c  ******************************    *)
(* es muß lediglich die Operation ’lege’ redefiniert werden *)

  CLASS neueWelt (blockwelt);

  PROCEDURE lege(x,y: CARDINAL; t:blockTyp):BOOLEAN;
  (* wie oben, zusätzlicher Parameter fr Blocktyp   *)
  (* post: if welt[x,y].top() = würfel then         *)
  (*        block an Pos. x,y gelegt, return true  *)
  (*       else kein Effekt, return false          *)
  (* val: wie in post beschrieben                  *)
  END neueWelt;

END Klotz.


IMPLEMENTATION MODULE Klotz;

FROM TModule IMPORT ElemT, makeElem, Compare;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

VAR i,j: CARDINAL;
```

```
CLASS IMPLEMENTATION blockwelt;

PROCEDURE erzeuge() : blockwelt;
 VAR bnew: POINTER TO blockwelt;
 BEGIN
  NEW(bnew);
  (* Initialisierung im  Block der Klasse *)
  RETURN bnew^
END erzeuge;


PROCEDURE lege(x,y: CARDINAL): BOOLEAN;
(* pre: b := create(), 0 <= x,y <= max  *)
(* post: block an pos x,y gelegt        *)
(* val: TRUE wenn erfolgreich           *)
BEGIN
      IF welt[x,y].IsFull() THEN RETURN FALSE
       ELSE welt[x,y].Push(makeElem(0)); RETURN TRUE
      END;
END lege;

PROCEDURE entferne  (x,y: CARDINAL):BOOLEAN;
(* pre: b := create(), 0 <= x,y <= max  *)
(* post: block an pos x,y gelegt        *)
(* val: TRUE wenn mind. ein Block an Pos x,y gelegen hat   *)
```