PRG1x & ADE1x	Üb. zu Einf. in die Programmierung und Element. Alg. u. Datenstrukt.	WS 20/21, Übung 8

Abgabetermin: Mi, 09. 12. 2020

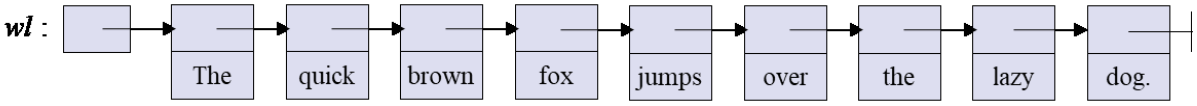
☐ **Gr. 1,** Dr. D. Auer
☒ **Gr. 2,** Dr. G. Kronberger
☐ **Gr. 3,** Dr. S. Wagner

Name Angelos Angelis **Aufwand in h** 10

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

1. Dynamische Datenstruktur für Texte (12 Punkte)

Für eine Textverarbeitungssoftware wird eine dynamische Datenstruktur benötigt, in der eine Sequenz von beliebig vielen Wörtern (vom Datentyp *STRING*) in Form einer *einfach-verketteten Liste* gespeichert werden kann. *Beispiel*:



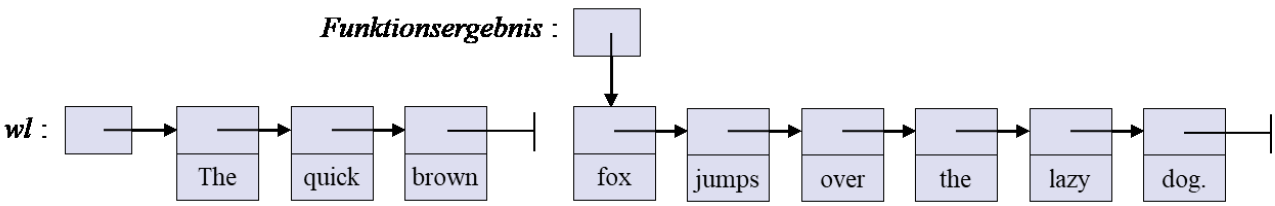
Realisieren Sie die geforderte Datenstruktur auf Basis der unten angegebenen Deklarationen indem Sie die angeführten Operationen implementieren. Gehen Sie bei dieser Aufgabe davon aus, dass nur Wörter mit einer maximalen Länge *maxWordLength* zu speichern sind.

```
CONST
  maxWordLength = 20;
TYPE
  WordNodePtr = ^WordNode;
  WordNode = RECORD
    next: WordNodePtr;
    word: STRING[maxWordLength];
  END; (* WordNode *)
  WordListPtr = WordNodePtr;

FUNCTION NewWordList: WordListPtr;
(* returns empty list *)
PROCEDURE DisposeWordList(VAR wl: WordListPtr);
(* disposes all nodes and sets wl to empty list *)
PROCEDURE AppendWord(VAR wl: WordListPtr; word: STRING);
(* appends word at the end of list wl *)
FUNCTION WordListLength(wl: WordListPtr): INTEGER;
(* returns number of characters in wl (i.e. all characters of
   all words and one space between two words) *)
PROCEDURE PrintWordList(wl: WordListPtr);
(* prints all words separated by a single space character *)
FUNCTION CopyWordList(wl: WordListPtr): WordListPtr;
(* copies all nodes of wl and returns head of new list *)
FUNCTION SplitWordList(VAR wl: WordListPtr; pos: INTEGER): WordListPtr;
(* splits wl at character position pos and returns list starting at node with
   character at pos *)
```

Hinweis zu Funktion *SplitWordList*: Die Liste *wl* wird bei dem Knoten geteilt, dessen Datenkomponente *word* das Zeichen an der Position *pos* enthält. Dabei muss zwischen zwei Wörtern jeweils ein Leerzeichen mitgezählt werden. Bei einem Wert *pos* außerhalb des gültigen Bereichs einer Liste *wl* ($1 \le pos \le WordListLength(wl)$) bleibt die Liste *wl* unverändert und das Funktionsergebnis liefert *NIL*.

Beispiel für Aufruf *SplitWordList* für obige Liste *wl* und *pos* = 18 (Zeichen an Pos. 18 = "o" in fox):



Testen Sie Ihre Implementierung ausführlich.

2. Worthäufigkeit (12 Punkte)

Entwickeln Sie ein Pascal-Programm, mit dem die Häufigkeit von Wörtern ermittelt werden kann. Das Programm soll die einzelnen Wörter einlesen und mit ihren Häufigkeiten in einer *doppelt-verketteten* Liste auf Basis folgender Deklarationen speichern.

```
TYPE
  WordNodePtr = ^WordNode;
  WordNode = RECORD
    prev, next: WordNodePtr;
    word: STRING;
    n: INTEGER;   (* frequency of word *)
  END; (* WordNode *)
```

a) In einer ersten Version hängen Sie neue Knoten für ein eingelesenes Wort einfach hinten an die Liste an. Vorne einzufügen brächte keinen Vorteil, denn man muss ohnedies für jedes Wort feststellen, ob es nicht bereits in der Liste enthalten ist (in diesem Fall ist nur der entsprechende Zähler n zu erhöhen). Anschließend sollen alle Knoten mit $n = 1$ aus der Liste entfernt werden und die in der Liste verbleibenden Wörter mit ihren Häufigkeiten ausgeben werden.

b) Implementieren Sie nun eine zweite Version in Form einer „selbstorganisierenden" Liste: Die Wortknoten sind in der Liste unsortiert gespeichert, jedes Mal, wenn ein (altes oder neues) Wort registriert wird, kommt der entsprechende Wortknoten aber ganz an den Anfang der Liste, wodurch häufig auftretende Wörter am Beginn der Liste stehen. Anschließend sollen wieder alle Knoten mit $n = 1$ aus der Liste entfernt werden und die in der Liste verbleibenden Wörter mit ihren Häufigkeiten ausgeben werden.

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee" an.
2. Dokumentieren und kommentieren Sie Ihre Algorithmen.
3. Bei Programmen: Geben Sie immer auch Testfälle ab, an denen man erkennen kann, dass Ihr Programm funktioniert, und dass es auch in Fehlersituation entsprechend reagiert.

1) ***Lösungsidee:***

Als erstes was mir aufgefallen ist, dass fast jede Prozedur/Funktion die eine Lister bearbeiten soll meistens Überprüfen soll ob die Liste Leer ist. Da das zu viel Schreibarbeit wäre habe ich eine extra Prozedur programmiert die überprüft ob die Liste leer ist und falls dies der Fall wird das Programm angehalten und dem User gesagt, dass die Liste Leer ist.

Ebenfalls gehe ich davon aus dass die Wörter mit einem Leerzeichen weitergegeben werden.

`FUNCTION NewWordList: WordListPtr:`
An sich selbstverständlich man muss nur NewWordList gleich NIL setzen.

`PROCEDURE DisposeWordList(VAR wl: WordListPtr)`
Die Liste wird durchgegangen mit dem wl und einem temporären Pointer wobei wl immer eine Node weiter ist als der temporäre Point, und jedes mal wird der temporäre Pointe Disposed

`PROCEDURE AppendWord(VAR wl: WordListPtr; word:STRING`
Die Liste muss bis zur letzten Node durchgelaufen werden. Anschließend muss der neu Erstellte Node mit dem übergebenem wort am letzten Platz eingehängt werden(nach letztem Node und vor NIL. Hier muss man ebenfalls beachten ob die Liste schon Leer ist, dann muss man nämlich der wl pointer einfach auf den neu erstellten Node zeigen.

`FUNCTION WordListLength(wl: WordListPtr): INTEGER`
Hier durchläuft man ebenfalls die wl Liste wobei jedes Mal wo der Node gewächselt wird die Länge des Wortes im Node in ein counter gespeichert addiert wird.

`PROCEDURE PrintWordList(wl: WordListPtr)`
Hier durchläuft man ebenfalls die wl Liste wobei jedes Mal wo der Node gewächselt wird das Wort des Nodes ausgegeben wird.

`FUNCTION CopyWordList(wl: WordListPtr): WordListPtr`
Hier durchläuft man ebenfalls die wl Liste wobei jedes Mal wo der Node gewächselt wird das Wort des Nodes eingelesen wird und in der neuen Liste nach hinten eingereiht wird.

`FUNCTION SplitWordList(VAR wl: WordListPtr; pos: INTEGER): WordListPtr`
Die Liste wird durchgegangen mit dem ausgabepointer und einem temporären Pointer wobei der ausgabepointer immer eine Node weiter ist als der temporäre Pointer. Bei jedem durchlauf addiert man die Wortlänge auf einem counter. Falls der counter, pos überschreitet geht man aus der schleife raus. Dann muss der temp. Pointer auf Nil zeigen und dann wird der ausgabePointer ausgegeben.

Quellcode:

```
UNIT UnitWordNodeList;

INTERFACE
CONST
  maxWordLength = 20;

TYPE
  WordNodePtr = ^WordNode;
  WordNode = RECORD
    next: WordNodePtr;
    word: STRING[maxWordLength];
    END;(* WordNode*)
  WordListPtr=WordNodePtr;

FUNCTION NewWordList: WordListPtr;
PROCEDURE DisposeWordList(VAR wl: WordListPtr);
PROCEDURE AppendWord(VAR wl: WordListPtr; word:STRING);
FUNCTION WordListLength(wl: WordListPtr): INTEGER;
PROCEDURE PrintWordList(wl: WordListPtr);
FUNCTION CopyWordList(wl: WordListPtr): WordListPtr;
FUNCTION SplitWordList(VAR wl: WordListPtr; pos: INTEGER): WordListPtr;



IMPLEMENTATION

FUNCTION NewWordList: WordListPtr;
BEGIN (* NewWordList *)
  NewWordList := NIL;
END; (* NewWordList *)

PROCEDURE CheckEmptyness(wl : WordListPtr);
BEGIN
  IF (wl = NIL) THEN BEGIN
    WriteLn('Your List is empty');
    HALT;
  END; (* IF *)
END;

PROCEDURE DisposeWordList(VAR wl : WordListPtr);
VAR tmp : WordListPtr;
```

```
BEGIN (* DisposeWordList *)
  CheckEmptyness(wl);
  WHILE (wl <> NIL) DO BEGIN
    tmp := wl;
    wl := wl^.next;
    Dispose(tmp);
  END; (* WHILE *)
END; (* DisposeWordList *)

PROCEDURE AppendWord(VAR wl : WordListPtr; word : STRING);
VAR
tmp : WordListPtr;
newWord : WordListPtr;
BEGIN (* AppendWord *)
  New(newWord);
  newWord^.word := word;
  newWord^.next := NIL;
  IF wl = NIL THEN BEGIN
    wl := newWord;
  END ELSE BEGIN
    tmp := wl;
    WHILE (tmp^.next <> NIL) DO BEGIN
      tmp := tmp^.next;
    END; (* WHILE *)
    tmp^.next := newWord;
  END;
END; (* AppendWord *)

FUNCTION WordListLength(wl: WordListPtr): INTEGER;
VAR
tmp : WordListPtr;
count : INTEGER;
BEGIN
  tmp := wl;
  count := Length(wl^.word);
  CheckEmptyness(wl);
  WHILE (tmp^.next <> NIL) DO BEGIN
    tmp := tmp^.next;
    count := count + Length(tmp^.word);
  END; (* WHILE *)
  WordListLength := count;
END;
```

```
PROCEDURE PrintWordList(wl : WordListPtr);
VAR
tmp : WordListPtr;
BEGIN (* PrintWordList *)
  tmp := wl;
  CheckEmptyness(wl);
  WHILE (tmp <> NIL) DO BEGIN
    Write(tmp^.word);
    tmp := tmp^.next;
  END; (* WHILE *)
END; (* PrintWordList *)

FUNCTION CopyWordList(wl : WordListPtr): WordListPtr;
VAR
tmp : WordListPtr;
copiedList : WordListPtr;
BEGIN (* CopyWordList *)
  tmp := wl;
  copiedList := NewWordList;
  CheckEmptyness(wl);
  WHILE(tmp <> NIL) DO BEGIN
    AppendWord(copiedList, tmp^.word);
    tmp := tmp^.next;
  END;
  CopyWordList := copiedList;
END; (* CopyWordList *)

FUNCTION SplitWordList(VAR wl : WordListPtr; pos : INTEGER): WordListPtr;
VAR
  tmp: WordListPtr;
  splittedWordList: WordListPtr;
  count: INTEGER;
BEGIN (* SplitWordList *)
  tmp := wl;
  splittedWordList := wl;
  count:= Length(tmp^.word);
  CheckEmptyness(wl);
  IF (count >= pos) THEN BEGIN
    wl := NIL;
  END ELSE BEGIN
    WHILE (count < pos) DO BEGIN
      splittedWordList := splittedWordList^.next;
      count := count + Length(splittedWordList^.word);
```

```
      IF (count < pos) THEN BEGIN
        tmp := tmp^.next;
      END; (* IF *)
    END; (* WHILE *)
    tmp^.next := NIL;
  END; (* IF *)
  SplitWordList := splittedWordList;
END;


BEGIN (* UnitWordNodeList *)

END. (* UnitWordNodeList *)
```

Testfälle:

1)

```
The quick brown fox jumps over the lazy dog.
Copied List: The quick brown fox jumps over the lazy dog.
WordListLength: 44
Spliting at pos 18: fox jumps over the lazy dog.
```

2)

```
Testing empty List
Your List is empty
```

3)

```
List with only 1 Node
OneWordOnly
Spliting at pos 18: OneWordOnly
Other Part of the List: Your List is empty
```

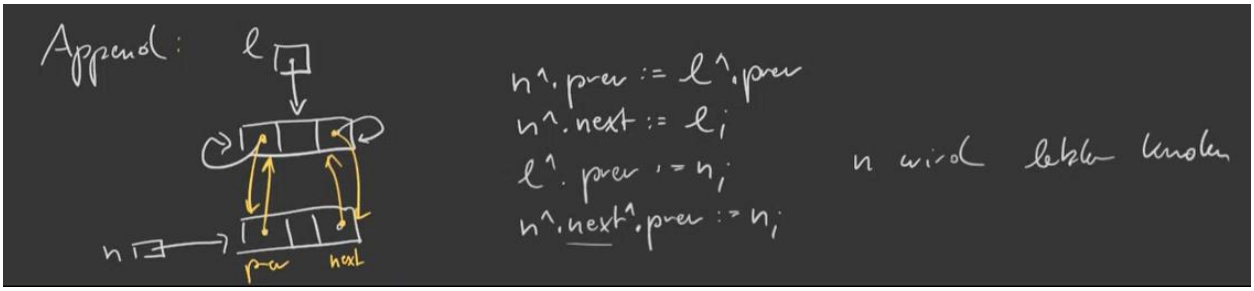
4)

```
The quick brown fox jumps over the lazy dog.
Spliting at a blank pos 10: quick brown fox jumps over the lazy dog.
The
```

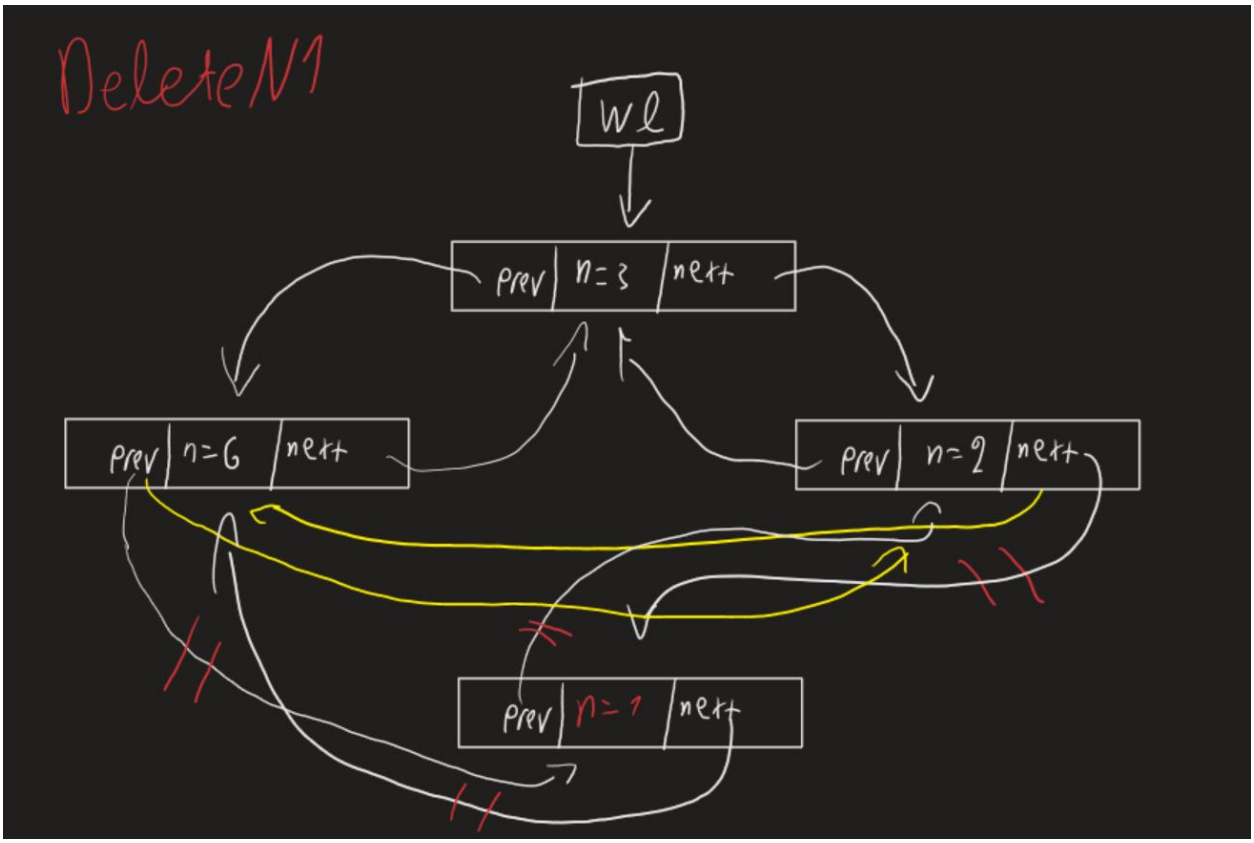
2)

Lösungsidee:

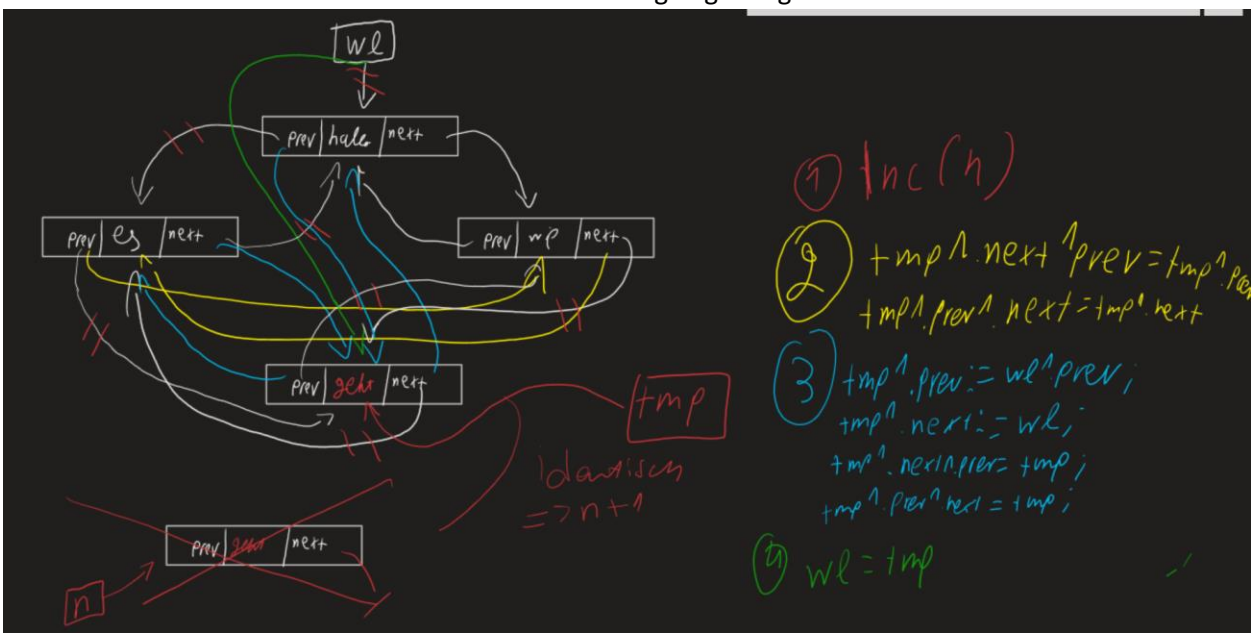
Die Aufgabe habe ich mit einer zyklischen doppelt verketteten Liste gemacht, und ich werde meine Lösungsidee anhand von Skizzen beschreiben. **Ebenfalls wie bei der Aufgabe 1 habe ich hier auch eine Prozedur die Überprüft ob die Liste leer ist.**



Wobei jedes Mal wo ein gleiches Wort eingelesen wird der count im jeweiligen Node erhöht wird.



Bei der Aufgabe (b) der einzige Unterschied ist dass wenn ein gleiches Wort erkannt wird, wird n icht nur n um 1 erhöht sondern es wird auch am Anfang angehängt.



Quellcode A:

```
PROGRAM WorthaueufigkeitV1;

TYPE
WordNodePtr = ^WordNode;
WordNode = RECORD
  prev,next: WordNodePtr;
  word: STRING;
  n: INTEGER;   (* frequency of word *)
END; (* WordNode*)

PROCEDURE InitList(VAR wl : WordNodePtr);
BEGIN (* InitList *)
  wl := NIL;
END; (* InitList *)

PROCEDURE CheckEmptyness(wl : WordNodePtr);
BEGIN
  IF (wl = NIL) THEN BEGIN
    WriteLn('Your List is empty');
    HALT;
  END; (* IF *)
END;

PROCEDURE AppendNode(VAR wl : WordNodePtr;word : STRING);
```

```
VAR
tmp,newWord : WordNodePtr;
previousNode : WordNodePtr;
BEGIN (* AppendNode *)
  CheckEmptyness(wl);
  tmp := wl;
  New(newWord);
  newWord^.next := NIL;
  newWord^.prev := NIL;
  newWord^.n := 1;
  newWord^.word := word;
  IF (wl = NIL) THEN BEGIN
    wl := newWord;
    wl^.next := wl;
    wl^.prev := wl;
  END ELSE BEGIN
    WHILE (tmp^.next <> wl) AND (tmp^.word <> newWord^.word) DO BEGIN
      tmp := tmp^.next;
    END; (* WHILE *)
    IF (tmp^.word = newWord^.word) THEN BEGIN
      Inc(tmp^.n);
    END ELSE BEGIN
      newWord^.prev := wl^.prev;
      newWord^.next := wl;
      newWord^.next^.prev := newWord;
      newWord^.prev^.next := newWord;
    END;
  END;
END; (* AppendNode *)

PROCEDURE PrintWordList(wl : WordNodePtr);
VAR
tmp : WordNodePtr;
BEGIN (* PrintWordList *)
  CheckEmptyness(wl);
  tmp := wl^.next;
  WriteLn(wl^.word, ' Haeufigkeit: ', wl^.n);
  WHILE (tmp <> wl) DO BEGIN
    WriteLn(tmp^.word, ' Haeufigkeit: ', tmp^.n);
    tmp := tmp^.next;
  END; (* WHILE *)
END; (* PrintWordList *)

PROCEDURE DeleteN1Words(VAR wl : WordNodePtr);
VAR
```

```
tmp,prev : WordNodePtr;
BEGIN
 tmp := wl;
 prev := wl;
 CheckEmptyness(wl);
  WHILE (tmp <> wl^.prev) DO BEGIN
    IF (tmp^.n <= 1) THEN BEGIN
      IF (tmp = wl) THEN BEGIN
        wl := tmp^.next;
      END; (* IF *)
      prev := tmp^.prev;
      tmp^.prev^.next := tmp^.next;
      tmp^.next^.prev := tmp^.prev;
      tmp^.next := NIL;
      tmp^.prev := NIL;
      Dispose(tmp);
      tmp := prev;
    END; (* IF *)
    tmp := tmp^.next;
  END; (* WHILE *)
  IF (tmp^.n <= 1) THEN BEGIN
    tmp^.prev^.next := tmp^.next;
    tmp^.next^.prev := tmp^.prev;
    tmp^.next := NIL;
    tmp^.prev := NIL;
    Dispose(tmp);
  END; (* IF *)
END;

VAR
wl : WordNodePtr;
BEGIN (* Worthaueufigkeit *)
  InitList(wl);
  AppendNode(wl,'hallo');
  AppendNode(wl,'wie');
  AppendNode(wl,'geht');
  AppendNode(wl,'es');
  AppendNode(wl,'dir');
  AppendNode(wl,'es');
  AppendNode(wl,'es');
  AppendNode(wl,'wie');
  AppendNode(wl,'es');
  AppendNode(wl,'dir');
  AppendNode(wl,'es');
  AppendNode(wl,'es');
```

```
  DeleteN1Words(wl);
  PrintWordList(wl);
END. (* Worthaueufigkeit *)
```

Quellcode B:

```
PROGRAM WorthaueufigkeitV2;

TYPE
WordNodePtr = ^WordNode;
WordNode = RECORD
  prev,next: WordNodePtr;
  word: STRING;
  n: INTEGER;   (* frequency of word *)
END; (* WordNode*)

PROCEDURE InitList(VAR wl : WordNodePtr);
BEGIN (* InitList *)
  wl := NIL;
END; (* InitList *)

PROCEDURE CheckEmptyness(wl : WordNodePtr);
BEGIN
  IF (wl = NIL) THEN BEGIN
    WriteLn('Your List is empty');
    HALT;
  END; (* IF *)
END;

PROCEDURE AppendNode(VAR wl : WordNodePtr;word : STRING);
VAR
tmp,newWord : WordNodePtr;
BEGIN (* AppendNode *)
  tmp := wl;
  New(newWord);
  newWord^.next := NIL;
  newWord^.prev := NIL;
  newWord^.n := 1;
  newWord^.word := word;
  CheckEmptyness(wl);
  IF (wl = NIL) THEN BEGIN
    wl := newWord;
    wl^.next := wl;
    wl^.prev := wl;
```

```
  END ELSE BEGIN
    WHILE (tmp^.next <> wl) AND (tmp^.word <> newWord^.word) DO BEGIN
      tmp := tmp^.next;
    END; (* WHILE *)
    IF (tmp^.word = newWord^.word) THEN BEGIN
      IF (tmp = wl) THEN BEGIN
        Inc(tmp^.n);
      END ELSE BEGIN
        Inc(tmp^.n);
        (*Split duplicate word*)
        tmp^.next^.prev := tmp^.prev;
        tmp^.prev^.next := tmp^.next;
        (*Duplicate word to start*)
        tmp^.prev := wl^.prev;
        tmp^.next := wl;
        tmp^.next^.prev := tmp;
        tmp^.prev^.next := tmp;
        wl := tmp;
      END;
    END ELSE BEGIN
      newWord^.prev := wl^.prev;
      newWord^.next := wl;
      newWord^.next^.prev := newWord;
      newWord^.prev^.next := newWord;
      wl := newWord;
    END;
  END;
END; (* AppendNode *)

PROCEDURE PrintWordList(wl : WordNodePtr);
VAR
tmp : WordNodePtr;
BEGIN (* PrintWordList *)
  CheckEmptyness(wl);
  tmp := wl^.next;
  WriteLn(wl^.word, ' Haeufigkeit: ', wl^.n);
  WHILE (tmp <> wl) DO BEGIN
    WriteLn(tmp^.word, ' Haeufigkeit: ', tmp^.n);
    tmp := tmp^.next;
  END; (* WHILE *)
END; (* PrintWordList *)

PROCEDURE DeleteN1Words(VAR wl : WordNodePtr);
VAR
tmp,prev : WordNodePtr;
```

```
BEGIN
 tmp := wl;
 prev := wl;
 CheckEmptyness(wl);
  WHILE (tmp <> wl^.prev) DO BEGIN
    IF (tmp^.n <= 1) THEN BEGIN
      IF (tmp = wl) THEN BEGIN
        wl := tmp^.next;
      END; (* IF *)
      prev := tmp^.prev;
      tmp^.prev^.next := tmp^.next;
      tmp^.next^.prev := tmp^.prev;
      tmp^.next := NIL;
      tmp^.prev := NIL;
      Dispose(tmp);
      tmp := prev;
    END; (* IF *)
    tmp := tmp^.next;
  END; (* WHILE *)
  IF (tmp^.n <= 1) THEN BEGIN
    tmp^.prev^.next := tmp^.next;
    tmp^.next^.prev := tmp^.prev;
    tmp^.next := NIL;
    tmp^.prev := NIL;
    Dispose(tmp);
  END; (* IF *)
END;

VAR
wl : WordNodePtr;
BEGIN (* Worthaueufigkeit *)
  InitList(wl);
  DeleteN1Words(wl);
  PrintWordList(wl);
END. (* Worthaueufigkeit *)
```

<u>Testfälle:</u>

A)Eingabe : `'hallo','wie','geht','es','dir','es''es','wie','es','dir','es'`

```
wie Haeufigkeit: 2
es Haeufigkeit: 6
dir Haeufigkeit: 2
```

B) Eingabe : `'hallo','wie','geht','es','dir','es''es','wie','es','dir','es'`

```
es Haeufigkeit: 6
dir Haeufigkeit: 2
wie Haeufigkeit: 2
```