ADF 2*x* & PRO 2*x* **Übungen zu Fortgeschrittenen Algorithmen & Datenstrukturen und OOP** **SS 21, Übung 1**

Abgabetermin: Mi, 17. 03. 2021

☐ **Gr. 1,** Dr. S. Wagner **Name** Angelos Angelis **Aufwand in h** 9

☒ **Gr. 2,** Dr. D. Auer

☐ **Gr. 3,** Dr. G. Kronberger **Punkte** _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

1. Worthäufigkeiten mit Hash-Tabellen (11 + 11 + 2 Punkte)

Entwickeln Sie ein Pascal-Programm *WordCounter*, das für eine Textdatei die Häufigkeiten der darin vorkommenden Wörter ermittelt und folgende Ergebnisse ausgibt: (1) die Anzahl aller vorkommenden Wörter, (2) die Anzahl der Wörter, die öfter als einmal vorkommen und (3) das am häufigsten vorkommende Wort mit seiner Häufigkeit. Zwischen Groß- und Kleinschreibung ist bei dieser Aufgabe nicht zu unterscheiden.

Zur Verwaltung der Wörter und ihrer Häufigkeiten verwenden Sie:

a) eine *Hash-Tabelle* mit Kollisionsbehandlungs-Strategie *Verkettung* (engl. *chaining*) und

b) eine *Hash-Tabelle* mit Kollisionsbehandlungs-Strategie *offene Adressierung*, z.B. *lineare* oder *quadratische* Kollisionsbehandlung bzw. *doppeltes Hashing*.

Hinweis für Ergebnis (2): Ermitteln Sie die Anzahl der Wörter, die öfter als einmal vorkommen, indem sie alle Wörter mit Häufigkeit 1 aus der Hash-Tabelle entfernen und danach alle verbleibenden Wörter zählen.

Untersuchen Sie für beide Varianten die Laufzeiteffizienz (mittels *Timer.pas*) und diskutieren Sie die Vor- und Nachteile der beiden Varianten.

Da das Thema Dateibearbeitung noch nicht behandelt wurde, finden Sie im Moodle-Kurs in der Datei *WordStuff.zip* mit *WordReader.pas* ein Modul, das eine einfache Schnittstelle zum Lesen von Wörtern aus Textdateien zur Verfügung stellt und mit *WordCounter.pas* eine Vorlage für die von Ihnen zu erstellenden Programmversionen, je eine für a) und b).

Testen Sie Ihre Programme mit kleineren Textdateien ausführlich bevor Sie im Roman "Das Schloß" von Franz Kafka (in der Datei *Kafka.txt*) das häufigste Wort ermitteln.

Lösungsidee:

1) Hierbei habe ich jedes Wort dass in der Haschtable einzuspeichern ist noch bevor es in de Hashtable gespeichert wurde gezählt. Zu beachten ist hier, dass man auch Duplikate die vorkommen zählen soll.
2) Hierbei muss man erstmal alle Wörter die Doppelt vorkommen löschen. Bei der Hashtable mit Chaining ist dies einbischen komplizierter aber es ist das gleiche Prinzip wie bei der Hashtable mit linearer Kollisionsbehandlung. Man muss alle Stellen der Hashtable also von 0 bis size-1 durchlaufen und überprüfen wie oft sie vorkommen. Falls sie <2 mal vorkommen löscht man sie. Letzten Endes muss man dann nachdem die Wörter gelöscht wurden die Hashtable nochmal von 0 bis size-1 durchlaufen und jeden Eintrag zählen und diese Zahl dann ausgeben. So hat man dann die Anzahl der nicht „unique“ Wörter.
3) Es soll das Wort gefunden werden, das am häufigsten vorkommt. Hierbei wird wieder die Hashtable von 0 bis size – 1 durchgelaufen und immer bei jedem Eintrag überprüft wie oft ein Wort vorkommt. Falls es mehr ist als der erstellte temporäre Node-Pointer wird die Eintrag in diesen temporären Node-Pointer gespeichert.
4) Chain-Collision-Handling vs Linear-Collision-Handling: Beim Chain-Collision-Handling dauert zwar das einfügen in die Hashtable nicht so lang (~3x weniger) wie beim Linear-Collision-Handling dafür sind aber die Prozeduren/Funktionen von der Linear-Collision-Handling Hashtable viel unkomplizierter bzw. einfacher und kürzer als die der Chain-Collision-Handling Hashtable.

Quellcode: Ich habe zwei verschieden Dateien einmal die Chain-Collision-Handling und einmal die Linear-Collision-Handling(V2)

Chain-Collision:

```
(* WordCounter:                                    HDO, 2003-02-28 *)
(* -----------                                                     *)
(*=================================================================*)
PROGRAM WordCounter;

  USES
    Crt, Timer, WordReader;

  CONST
    size = 3000;

  TYPE
    NodePtr = ^NodeRec;
    NodeRec = RECORD
      key : WORD;
      freq : INTEGER;
      next :NodePtr;
    END;
    ListPtr = NodePtr;
    Hashtable = ARRAY [0..size-1] OF ListPtr;
```

```
  VAR
    ht: HashTable;
    hcount : INTEGER;

  (* --- hash functions --- *)
  FUNCTION HashCode1(key: STRING): INTEGER;
  BEGIN (* HashCode1 *)
    HashCode1 := Ord(key[1]) MOD size;
  END; (* HashCode1 *)

  FUNCTION HashCode2(key: STRING): INTEGER;
  BEGIN (* HashCode2 *)
    HashCode2 := (Ord(key[1]) + Length(key)) MOD size;
  END; (* HashCode2 *)

  FUNCTION HashCode3(key: STRING): INTEGER;
  BEGIN (* HashCode3 *)
    IF Length(key) = 1 THEN
      HashCode3 := ((Ord(key[1]) * 7 + 1) * 17) MOD size
    ELSE
      HashCode3 := ((Ord(key[1]) * 7 + Ord(key[2]) + Length(key)) * 17) MOD size;
  END; (* HashCode3 *)

  FUNCTION HashCode4(key: STRING): INTEGER;
    VAR
      hc, i: INTEGER;
  BEGIN (* HashCode4 *)
    hc := 0;
    FOR i:=1 TO Length(key) DO BEGIN
      hc := hc + Ord(key[i]);
    END; (*FOR*)
    HashCode4 := hc MOD size;
  END; (* HashCode4 *)

  FUNCTION HashCode5(key: STRING): INTEGER;
    VAR
      hc, i: INTEGER;
  BEGIN (* HashCode5 *)
    hc := 0;
    FOR i:=1 TO Length(key) DO BEGIN
(*$Q-*)
(*$R-*)
      hc := hc * 31 + Ord(key[i]);
(*$R+*)
```

```
(*$Q+*)
    END; (*FOR*)
    HashCode5 := Abs(hc) MOD size;
  END; (* HashCode5 *)

  FUNCTION HashCode(key: STRING): INTEGER;
  BEGIN (* HashCode *)
    HashCode := HashCode4(key);
  END; (* HashCode *)

(* --- hashtable handling ---*)
  FUNCTION NewNode(key: STRING; next: NodePtr): NodePtr;
    VAR
      n: NodePtr;
  BEGIN (* NewNode *)
    New(n);
    n^.key := key;
    n^.next := next;
    n^.freq := 1;
    NewNode := n;
  END; (* NewNode *)

  PROCEDURE InitHashTable;
    VAR
      h: INTEGER;
  BEGIN
    FOR h:= 0 TO size-1 DO BEGIN
      ht[h] := NIL;
    END; (*FOR*)
  END; (*InitHashTable*)

  PROCEDURE WriteHashTable;
    VAR
      h: INTEGER;
      n: NodePtr;
  BEGIN (* WriteHashTable *)
    FOR h := 0 TO size-1 DO BEGIN
      IF ht[h] <> NIL THEN BEGIN
        Write(h, ': ');
        n := ht[h];
        WHILE n <> NIL DO BEGIN
          Write(n^.key, ' ' ,n^.freq);
          n := n^.next;
        END; (*WHILE*)
        WriteLn;
```

```
      END; (*IF*)
    END; (*FOR*)
  END; (* WriteHashTable *)

PROCEDURE DeleteUniq;
VAR
  e,n,prev: NodePtr;
  i : INTEGER;
BEGIN
  prev := ht[0];
  IF prev <> NIL THEN n := prev^.next ELSE n := NIL;
  FOR i := 0 TO size-1 DO BEGIN
    prev := ht[i];
    IF prev <> NIL THEN n := prev^.next ELSE n := NIL;
    WHILE (ht[i] <> NIL) AND (ht[i]^.freq = 1) DO BEGIN
      e := ht[i];
      ht[i] := n;
      prev^.next := NIL;
      prev := ht[i];
      Dispose(e);
      IF prev <> NIL THEN n := prev^.next ELSE n := NIL;
    END; (* WHILE *)
    WHILE (n <> NIL) DO BEGIN
      IF (n^.freq = 1) THEN BEGIN
        e := n;
        n := n^.next;
        prev^.next^.next := NIL;
        prev^.next := n;
        Dispose(e);
      END ELSE BEGIN (* IF *)
        n := n^.next;
        prev := prev^.next;
      END
    END; (* WHILE *)
  END; (* FOR *)
END;

FUNCTION CountNonUniq: INTEGER;
VAR
  n: NodePtr;
  i,count : LONGINT;
BEGIN
  DeleteUniq;
  count := 0;
  FOR i := 0 TO size-1 DO BEGIN
```

```
    n := ht[i];
    WHILE (n <> NIL) DO BEGIN
      Inc(count);
      n := n^.next;
    END; (* WHILE *)
  END; (* FOR *)
  CountNonUniq := count;
END;

FUNCTION Lookup(key: STRING): NodePtr;
VAR
  h: INTEGER;
  n: NodePtr;
  collPossible: BOOLEAN;
BEGIN (* Lookup *)
  h := HashCode(key);
  n := ht[h];
  collPossible := FALSE;
  WHILE (n <> NIL) AND (n^.key <> key) DO BEGIN
    n := n^.next;
    collPossible := TRUE;
  END; (*WHILE*)
  IF n = NIL THEN BEGIN
    n := NewNode(key, ht[h]);  (*prepend*)
    ht[h] := n;
  END ELSE IF (n^.key = key) THEN BEGIN
    n^.freq := n^.freq +1;
    collPossible := FALSE;
  END; (*IF*)
  Lookup := n;
END; (* Lookup *)

  FUNCTION MostUsed : NodePtr;
  VAR
    n,count: NodePtr;
    i : INTEGER;
  BEGIN (* MostUsed *)
    count := NewNode('count',NIL);
    n := ht[0];
    FOR i := 0 TO size-1 DO BEGIN
      WHILE (n <> NIL) DO BEGIN
        IF (n^.freq >= count^.freq) THEN BEGIN
          count := n;
          n := n^.next;
        END ELSE n := n^.next;
```

```
      END; (* WHILE *)
      n := ht[i];
    END; (* FOR *)
    MostUsed := count;
  END; (* MostUsed *)

VAR
  w: Word;
  n: LONGINT;
  test : NodePtr;
BEGIN (*WordCounter*)
  WriteLn('WordCounter:');
  OpenFile('Semester_2/Uebung_1/WordStuff/Kafka.txt', toLower);
  StartTimer;
  InitHashTable;
  n := 0;
  hcount := 0;
  ReadWord(w);
  WHILE w <> '' DO BEGIN
    n := n + 1;
    (*insert word in data structure and count its occurence*)
    ReadWord(w);
    Lookup(w);
  END; (*WHILE*)
  StopTimer;
  CloseFile;
  WriteLn('number of words: ', n);
  WriteLn('elapsed time:    ', ElapsedTime);
  (*search in data structure for word with max. occurrence*)
  WriteLn('Most Used Word: ',MostUsed^.key, ' ', MostUsed^.freq);
  WriteLn('Non Unique Words: ',CountNonUniq);

END. (*WordCounter*)
```

Testfall mit Kafka.txt:

```
WordCounter:
number of words: 109046
elapsed time:    00:00.04
Most Used Word: und 2927
Non Unique Words: 4815
```

Linear-Collision:

```
(* WordCounterV2:                                    HDO, 2003-02-28 *)
(* -----------                                                       *)
(*===================================================================*)
PROGRAM WordCounter;

  USES
    Crt, Timer, WordReader;

  CONST
    size = 11000;

  TYPE
    NodePtr = ^NodeRec;
    NodeRec = RECORD
      key: STRING;
      freq : INTEGER;
      (*data: AnyType*)
    END; (* NodeRec *)
    HashTable = ARRAY[0..size-1] OF NodePtr;

  VAR
    ht: HashTable;

  (* --- hash functions --- *)
  FUNCTION HashCode1(key: STRING): INTEGER;
  BEGIN (* HashCode1 *)
    HashCode1 := Ord(key[1]) MOD size;
  END; (* HashCode1 *)

  FUNCTION HashCode2(key: STRING): INTEGER;
  BEGIN (* HashCode2 *)
    HashCode2 := (Ord(key[1]) + Length(key)) MOD size;
  END; (* HashCode2 *)
```

```
  FUNCTION HashCode3(key: STRING): INTEGER;
  BEGIN (* HashCode3 *)
    IF Length(key) = 1 THEN
      HashCode3 := ((Ord(key[1]) * 7 + 1) * 17) MOD size
    ELSE
      HashCode3 := ((Ord(key[1]) * 7 + Ord(key[2]) + Length(key)) * 17) MOD size;
  END; (* HashCode3 *)

  FUNCTION HashCode4(key: STRING): INTEGER;
    VAR
      hc, i: INTEGER;
  BEGIN (* HashCode4 *)
    hc := 0;
    FOR i:=1 TO Length(key) DO BEGIN
      hc := hc + Ord(key[i]);
    END; (*FOR*)
    HashCode4 := hc MOD size;
  END; (* HashCode4 *)

  FUNCTION HashCode5(key: STRING): INTEGER;
    VAR
      hc, i: INTEGER;
  BEGIN (* HashCode5 *)
    hc := 0;
    FOR i:=1 TO Length(key) DO BEGIN
(*$Q-*)
(*$R-*)
      hc := hc * 31 + Ord(key[i]);
(*$R+*)
(*$Q+*)
    END; (*FOR*)
    HashCode5 := Abs(hc) MOD size;
  END; (* HashCode5 *)

  FUNCTION HashCode(key: STRING): INTEGER;
  BEGIN (* HashCode *)
    HashCode := HashCode1(key);
  END; (* HashCode *)

(* --- hashtable handling --- *)
  FUNCTION NewNode(key: STRING): NodePtr;
    VAR
      n: NodePtr;
  BEGIN (* NewNode *)
    New(n);
```

```
    n^.key := key;
    n^.freq := 1;
    NewNode := n;
  END; (* NewNode *)

  PROCEDURE InitHashTable;
    VAR
      h: INTEGER;
  BEGIN
    FOR h:= 0 TO size-1 DO BEGIN
      ht[h] := NIL;
    END; (*FOR*)
  END; (*InitHashTable*)

  PROCEDURE WriteHashTable;
    VAR
      h: INTEGER;
  BEGIN (* WriteHashTable *)
    FOR h := 0 TO size-1 DO BEGIN
      IF ht[h] <> NIL THEN BEGIN
        WriteLn(h, ': ', ht[h]^.key,' ', ht[h]^.freq);
      END; (*IF*)
    END; (*FOR*)
  END; (* WriteHashTable *)

  FUNCTION Lookup(key: STRING): NodePtr;
    CONST
      c = 7; (* c > 1, no common divisior with size *)
      c2 = 11;
    VAR
      h: INTEGER;
      n: NodePtr;
      nrOfColls: INTEGER;
  BEGIN (* Lookup *)
    h := HashCode(key);
    nrOfColls := 0;
    WHILE (nrOFColls <= size) DO BEGIN
      n := ht[h];
      IF n = NIL THEN BEGIN
        n := NewNode(key);
        ht[h] := n;
        Lookup := n;
        Exit;
      END
      ELSE IF n^.key = key THEN BEGIN
```

```
        n^.freq := n^.freq + 1;
        Lookup := n;
        Exit;
      END;
      (*linear probing*)
      h := (h + 1) MOD size;
      nrOfColls := nrOFColls + 1;
    END; (*WHILE*)
    WriteLn('ERROR: Hashtable overflow.')
  END; (* Lookup *)

  FUNCTION MostUsed : NodePtr;
  VAR
    n,count: NodePtr;
    i : INTEGER;
  BEGIN (* MostUsed *)
    count := NewNode('count');
    n := ht[0];
    FOR i := 0 TO size-1 DO BEGIN
      n := ht[i];
      IF (n <> NIL) AND (n^.freq >= count^.freq) THEN BEGIN
        count := n;
      END;
    END; (* FOR *)
    MostUsed := count;
  END; (* MostUsed *)

PROCEDURE DeleteUniq;
VAR
  n: NodePtr;
  i : INTEGER;
BEGIN
  FOR i := 0 TO size-1 DO BEGIN
    IF (ht[i] <> NIL) AND (ht[i]^.freq = 1) THEN BEGIN
      n := ht[i];
      ht[i] := NIL;
      Dispose(n);
    END; (* IF *)
  END; (* FOR *)
END;

FUNCTION CountNonUniq: INTEGER;
VAR
  count, i : INTEGER;
BEGIN (* CountNonUniq *)
```

```
  count := 0;
  FOR i := 0 TO size-1 DO BEGIN
    IF (ht[i] <> NIL) THEN BEGIN
      Inc(count);
    END; (* IF *)
  END; (* FOR *)
  CountNonUniq := count;
END; (* CountNonUniq *)

VAR
  w: Word;
  n: LONGINT;
  test : NodePtr;
BEGIN (*WordCounter*)
  WriteLn('WordCounter:');
  OpenFile('Semester_2/Uebung_1/WordStuff/Kafka.txt', toLower);
  StartTimer;
  InitHashTable;
  n := 0;
  ReadWord(w);
  WHILE w <> '' DO BEGIN
    n := n + 1;
    (*insert word in data structure and count its occurence*)
    ReadWord(w);
    Lookup(w);
  END; (*WHILE*)
  StopTimer;
  CloseFile;
  WriteLn('number of words: ', n);
  WriteLn('elapsed time:    ', ElapsedTime);
  (*search in data structure for word with max. occurrence*)
  WriteLn('Most Used Word: ',MostUsed^.key, ' ', MostUsed^.freq);
  DeleteUniq;
  WriteLn('Non Unique Words: ',CountNonUniq);
  //WriteHashTable;

END. (*WordCounter*)
```

Testfall mit Kafka.txt:

```
WordCounter:
number of words: 109046
elapsed time:    00:00.99
Most Used Word: und 2927
Non Unique Words: 4815
```