

<input checked="" type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name <u>PAPESH Konstantin</u>	Aufwand in h <u>8</u>
<input type="checkbox"/> Gr. 2, Dr. H. Gruber		
<input type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte <u>16</u>	Kurzzeichen Tutor / Übungsleiter <u>L.S.</u> / <u></u>

**1. Worthäufigkeiten mit unterschiedlichen Datenstrukturen (6 + 6 + 4 + 2 Punkte)**

Entwickeln Sie ein Pascal-Programm *WordCounter*, das für eine Textdatei die Häufigkeiten der darin vorkommenden Wörter ermittelt – zwischen Groß- und Kleinschreibung ist dabei nicht zu unterscheiden – und das am häufigsten vorkommende Wort mit seiner Häufigkeit ausgibt.

Zur Verwaltung der Wörter und ihrer Häufigkeiten verwenden Sie (in dieser Reihenfolge):

- einen *binären Suchbaum*,
- eine *Hashtabelle* mit Kollisionsbehandlungs-Strategie *Verkettung* (engl. *chaining*) und
- eine *Hashtabelle* mit einer anderen Kollisionsbehandlungs-Strategie ("offene Adressierung"), wie *lineare* oder *quadratische* Kollisionsbehandlung bzw. *doppeltes Hashing*.

Untersuchen Sie für alle drei Varianten die Speicherplatz- und Laufzeiteffizienz (mittels *Timer.pas*) und diskutieren Sie die Vor- und Nachteile (ev. sogar Probleme) der drei 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) bis c).

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.

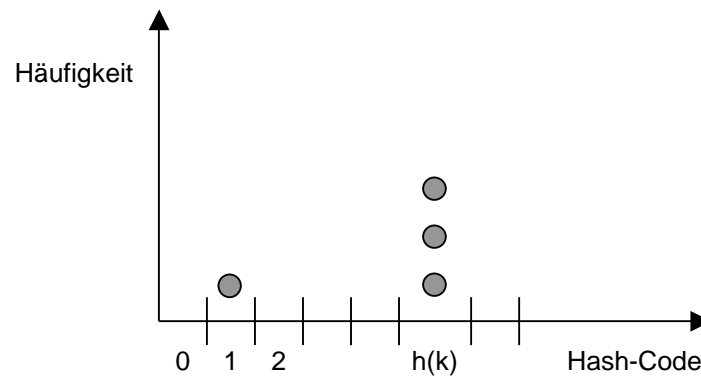
**2. Güte von Hash-Funktionen****(6 Punkte)**

Betrachtet werden Hash-Funktionen  $h$ , die Wörter (Schlüssel, engl. *keys*)  $k$  auf  $n$  positive ganze Zahlen (engl. *hash codes*)  $hc = h(k)$  im Bereich von 0 bis  $n - 1$  abbilden. Diese Hash-Codes können zum Indizieren von Hash-Tabellen verwendet werden. Die Güte einer Hash-Funktion wird neben ihrer Effizienz (geringer Aufwand zur Berechnung) vor allem dadurch bestimmt, wie gut sie die Schlüsselmenge (den Wertebereich) auf den Bereich der Hash-Codes (den Bildbereich) abbildet: eine Gleichverteilung stellt das Optimum dar.

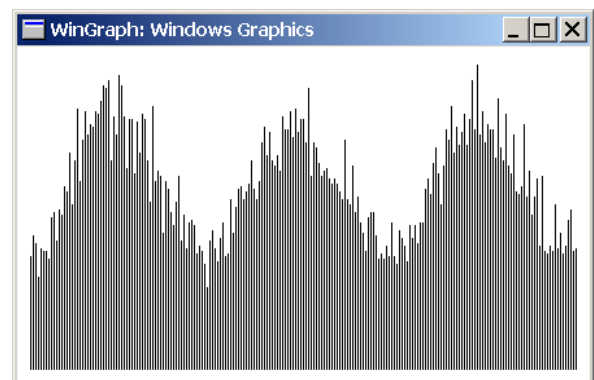
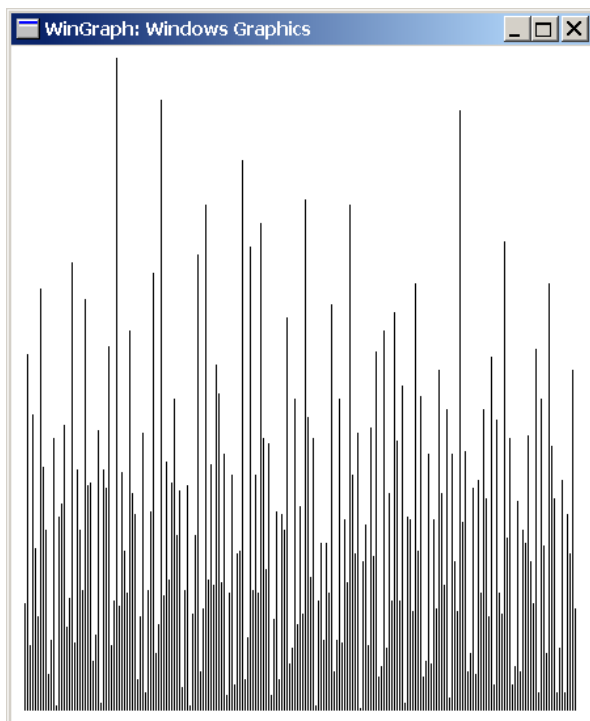
Zu Beginn dieses Semesters wurde schon ein Pascal-Programm zur einfachen Erstellung von Graphiken unter Windows (in der ZIP-Datei unter *WinGraph + Testprogramm*) vorgestellt und dazu verwendet, um die Güte von Zufallszahlengeneratoren zu visualisieren (z. B. in Form des Himmels-tests). Benutzen Sie dieses System nun, um die Güte von *mindestens drei unterschiedlichen* Hash-Funktionen zu visualisieren.

Zu Testzwecken finden Sie in der Datei *KafkaWords.txt* über 10.000 unterschiedliche Wörter (aus *Kafka.txt*). Entwickeln Sie eine *Redraw*-Prozedur, so dass die Wörter aus der Wortdatei gelesen werden, für jedes Wort der Hash-Code mittels einer Hash-Funktion berechnet wird und die Häufigkeit der einzelnen Hash-Codes ermittelt sowie visualisiert wird.

Dazu werden in einem zweidimensionalen Koordinatensystem horizontal die Hash-Codes von 0 bis  $n - 1$  aufgetragen und vertikal jeweils ein Punkt dargestellt, wenn der entsprechende Hash-Code ermittelt wurde. Folgende Darstellung zeigt einen Zustand, bei dem z. B. der Hash-Code  $h(k)$  bereits dreimal ermittelt wurde (es also schon zu zwei Kollisionen gekommen ist):



Die beiden Abbildungen unten zeigen zwei mögliche Ergebnisse für zwei unterschiedliche Hash-Funktionen mit jeweils  $n = 211$  und den Wörtern aus der Datei *KafkaWords.txt*:



# ADF2x & PRO2X Algorithmen & Datenstrukturen und OOP – SS 2018 Übungsabgabe 2

Konstantin Papesh

4. April 2018

## Zusammenfassung

In dieser Übung werden zuerst drei Systeme entworfen, mit denen Daten verarbeitet und deren Häufigkeit gezählt werden können. Als zweite Aufgabe wird die Güte von drei Hashalgorithmen verglichen.

## 2.1 Worthäufigkeit mit unterschiedlichen Datenstrukturen

Es soll die Speicher- und Laufzeiteffizienz mehrerer Datenstrukturen verglichen werden. Dabei sind drei Datenstrukturen zu konstruieren:

1. Ein binärer Suchbaum
2. Hashtabelle mit Verkettung
3. Hashtabelle mit linearer oder quadratischer Kollisionsbehandlung <sup>1</sup>

Die entwickelten Algorithmen sind dann mithilfe des Romans *Das Schloss* von Franz Kafka zu vergleichen.

**Angabe, keine Lösungsidee -1**

### 2.1.1 Implementierung

**Listing 2.1:** WordCounter.pas

```
1 (* WordCounter:                                HD0, 2003-02-28 *)
2 (* -----                                     *)
3 (* Template for programs that count words in text files.      *)
4 (*=====*)
5 PROGRAM WordCounter;
6
7   USES
8     Timer, WordReader, ModBinarySearchTree;
9
10  VAR
11    c, maxC : integer;
12    mostFreqWord : string;
```

---

<sup>1</sup>In dieser ausgearbeiteten Übung mithilfe linearer Kollisionsbehandlung

```

13     w: Word;
14     n: LONGINT;
15
16 BEGIN (*WordCounter*)
17     init();
18     WriteLn('WordCounter:');
19     OpenFile('Kafka.txt', toLower);
20     StartTimer;
21     n := 0;
22     maxC := 0;
23     ReadWord(w);
24     WHILE w <> '' DO BEGIN
25         n := n + 1;
26         c := addOrInc(w);
27         if c > maxC then begin
28             maxC := c;
29             mostFreqWord := w;
30         end;
31         ReadWord(w);
32     END; (*WHILE*)
33     StopTimer;
34     CloseFile;
35     WriteLn('number of words: ', n);
36     WriteLn('elapsed time: ', ElapsedTime);
37     writeLn('Most frequent word ', mostFreqWord, ' ', maxC);
38
39 END. (*WordCounter*)

```

Listing 2.2: ModStringHash.pas

```

1 unit ModStringHash;
2
3 interface
4 function stringHash1(w : string) : integer;
5     Zahl am Ende des Funktionsnamens weglassen!
6 implementation
7 function stringHash1(w : string) : integer;
8 var i : integer;
9     sum : integer;
10 begin
11     sum := 0;
12     for i := 1 to length(w) do begin
13         sum := (sum + Ord(w[i])) mod 32768;
14     end;
15     stringHash1 := sum;
16 end;
17
18 begin
19 end.

```

**Magic-Number -0,5**

Listing 2.3: ModBinarySearchTree.pas

```

1 unit ModBinarySearchTree;
2
3 interface
4 function addOrInc(w : string) : integer;
5 function find(w : string) : integer;

```

```

6 procedure init;
7
8 implementation
9 uses ModStringHash;
10 const
11     M = 53;
12     LEFT = 1;
13     RIGHT = 2;
14 type
15     NodePtr = ^NodeRec;
16     NodeRec = record
17         key : string;
18         hash : integer;
19         val : integer;
20         left : NodePtr;
21         right : NodePtr;
22     end;
23
24 var
25     root : NodePtr;
26
27 function addOrInc(w : string) : integer;
28 var
29     h : integer;
30     n : NodePtr;
31     nOld : NodePtr;
32     direction : integer;
33 begin
34     h := stringHash1(w) mod M;
35     n := root;
36     nOld := n;
37     direction := 0; //1 == LEFT, 2 == RIGHT
38     while (n <> NIL) and (n^.key <> w) do
39     begin
40         nOld := n;
41         if h < n^.hash then begin
42             n := n^.left;
43             direction := LEFT;
44         end else begin
45             n := n^.right;
46             direction := RIGHT;
47         end;
48     end;
49     if n = NIL then begin
50         New(n);
51         n^.key := w;
52         n^.hash := h;
53         n^.val := 1;
54         n^.left := NIL;
55         n^.right := NIL;
56         if direction = LEFT then
57             nOld^.left := n
58         else if direction = RIGHT then
59             nOld^.right := n
60         else
61             root := n;
62     end else

```

**schlechte Namen -0,5**

**eigene Funktion zum Erstellen von Knoten -0,5**

```

63     inc(n^.val);
64     addOrInc := n^.val;
65 end;
66
67 function find(w : string) : integer;
68 var h : integer;
69     n : NodePtr;
70 begin
71     h := stringHash1(w) mod M;
72     n := root;
73     while (n <> NIL) and (n^.key <> w) do begin
74         if h < n^.hash then
75             n := n^.left
76         else
77             n := n^.right;
78     end;
79     if n = NIL then
80         find := 0
81     else
82         find := n^.val;
83
84 end;
85
86 procedure init;
87 begin
88     root := NIL;
89 end;
90
91
92 begin
93 end.

```

**Listing 2.4:** ModHashTableChaining.pas

```

1 unit ModHashTableChaining;
2
3 interface
4 function addOrInc(w : string) : integer;
5 function find(w : string) : integer;
6 procedure writeTable;
7 procedure init;
8
9 implementation
10 uses ModStringHash;
11 const
12     M = 53;
13 type
14     NodePtr = ^NodeRec;
15     NodeRec = record
16         key : string;
17         val : integer;
18         next : NodePtr;
19     end;
20     ListPtr = NodePtr;
21
22 var
23     table : array[0..M-1] of ListPtr;

```

```

24
25 function addOrInc(w : string) : integer;
26 var h : integer;
27     n : NodePtr;
28 begin
29     h := stringHash1(w) mod M;
30     n := table[h];
31     while (n <> NIL) and (n^.key <> w) do
32         n := n^.next;
33     if n = NIL then begin
34         New(n);
35         n^.key := w;
36         n^.val := 1;
37         n^.next := table[h];
38         table[h] := n;
39     end else
40         inc(n^.val);
41     addOrInc := n^.val;
42 end;
43
44 function find(w : string) : integer;
45 var h : integer;
46     n : NodePtr;
47 begin
48     h := stringHash1(w) mod M;
49     n := table[h];
50     while (n <> NIL) and (n^.key <> w) do
51         n := n^.next;
52     if n = NIL then
53         find := 0
54     else
55         find := n^.val;
56 end;
57
58 procedure init;
59 var
60     i : integer;
61 begin
62     (* TODO: if entries exist we need to dispose them*)
63     for i := 0 to M-1 do begin
64         table[i] := NIL;
65     end;
66 end;
67
68 procedure writeList(n : ListPtr);
69 Begin
70     if n = NIL then
71         Write(' -| ')
72     else begin
73         write('-> ', n^.key, ', ', n^.val);
74         writeList(n^.next);
75     end;
76 end;
77
78
79 procedure writeTable;
80 var i : integer;

```

**Erstellen von Knoten in eigene Funktion**

**ungünstiges Kommentar -0,5**

```

81 begin
82     for i := 1 to M -1 do begin
83         WriteList(table[i]);
84         WriteLn;
85     end;
86 end;
87
88
89 begin
90 end.

```

**Listing 2.5:** ModHashTableLinearProbing.pas

```

1 unit ModHashTableLinearProbing;
2 (*$R-*)
3 interface
4 function addOrInc(w : string) : integer;
5 function find(w : string) : integer;
6 procedure writeTable;
7 procedure init;
8
9 implementation
10 uses ModStringHash;
11
12 type
13     keyValuePair = record
14         key : string;
15         value : integer;
16     end;
17     keyValuePairArray = array[0..0] of keyValuePair;
18
19 var
20     tablePtr : ^keyValuePairArray;
21     m : integer;
22
23 procedure init;
24 var i : integer;
25 begin
26     m := 3;
27     getMem(tablePtr, m * (SIZEOF(keyValuePair)));
28     for i := 0 to m-1 do begin
29         tablePtr^[i].key := '';
30         tablePtr^[i].value := 0;
31     end;
32 end;
33
34 procedure writeTable;
35 var i : integer;
36 begin
37     for i := 0 to m-1 do begin
38         writeLn(i, ': ', tablePtr^[i].key, ', ', tablePtr^[i].value);
39     end;
40 end;
41
42 function find(w : string) : integer;
43 var h : integer;
44     col : integer;

```

**Key ist der Hashwert, Value der String -0,5**

**Pascal is a language for savages.**



```

45 begin
46   h := stringHash1(w) mod m;
47   col := 0;
48   while(tablePtr^[h].key <> w) and
49     (tablePtr^[h].key <> '') and
50     (col < M) do begin
51     h:= (h + 1) mod m; (* Instead of 1 multiply by 2 every while run -->
        quadratic*)
52     Inc(col);
53   end;
54   if tablePtr^[h].key = w then
55     find := tablePtr^[h].value
56   else
57     find := 0;
58 end;
59
60 procedure reallocTable;
61 var oldTablePtr : ^keyValuePairArray;
62   oldM : integer;
63   kvp : keyValuePair;
64   i, j : integer;
65 begin
66   oldTablePtr := tablePtr;
67   oldM := m;
68   m := 2 * m;
69   getMem(tablePtr, m * SIZEOF(keyValuePair));
70   for i := 0 to m-1 do begin
71     tablePtr^[i].key := '';
72     tablePtr^[i].value := 0;
73   end;
74   FOR i := 0 to oldM do begin
75     kvp := oldTablePtr^[i];
76     if kvp.key <> '' then begin
77       for j := 1 to kvp.value do begin
78         addOrInc(kvp.key);
79       end;
80     end;
81   end;
82   freeMem(oldTablePtr, oldM * SIZEOF(keyValuePair));
83 end;
84
85 function addOrInc(w : string) : integer;
86 var h : integer;
87   col : integer;
88 begin
89   h := stringHash1(w) mod m;
90   col := 0;
91   while(tablePtr^[h].key <> w) and
92     (tablePtr^[h].key <> '') and
93     (col < M) do begin
94     h:= (h + 1) mod m;
95     inc(col);
96   end;
97   if tablePtr^[h].key = w then
98     inc(tablePtr^[h].value)
99   else if tablePtr^[h].key = '' then begin
100     tablePtr^[h].key := w;

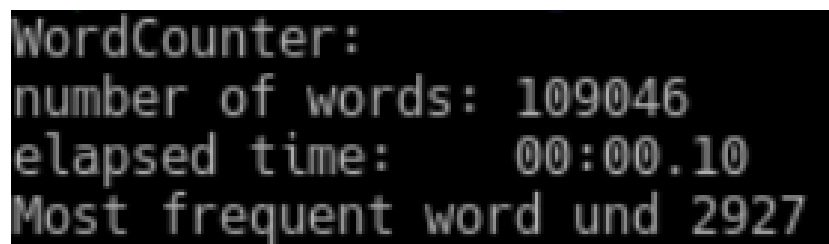
```

```

101     tablePtr^[h].value := 1;
102     addOrInc := tablePtr^[h].value;
103   end else begin
104     reallocTable;
105     addOrInc := addOrInc(w);
106   end;
107 end;
108
109 begin
110 end.
111 (*$R+*)

```

### 2.1.2 Ausgabe

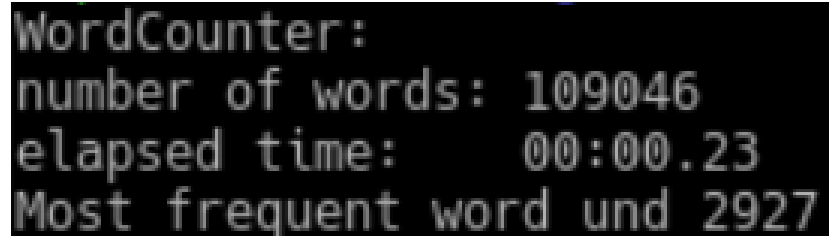


```

WordCounter:
number of words: 109046
elapsed time:    00:00.10
Most frequent word und 2927

```

Abbildung 2.1: Ausgabe des binären Suchbaums

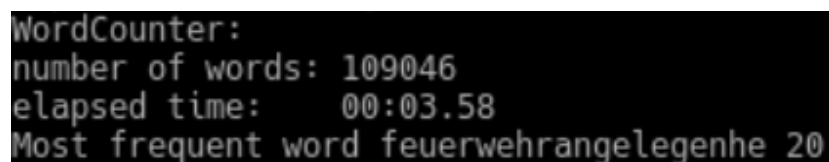


```

WordCounter:
number of words: 109046
elapsed time:    00:00.23
Most frequent word und 2927

```

Abbildung 2.2: Ausgabe der Hashtabelle mit Verkettung



```

WordCounter:
number of words: 109046
elapsed time:    00:03.58
Most frequent word feuerwehrangelegenhe 20

```

Abbildung 2.3: Ausgabe der Hashtabelle mit linearer Kollisionsbehandlung

### 2.1.3 Auswertung

Nach dem Ausführen aller drei Datenstrukturen kommt man auf das Ergebnis, dass der binäre Suchbaum am schnellsten von allen ausgearbeiteten Algorithmen arbeitet<sup>2.1</sup>. So braucht dieser für die Verarbeitung des gegebenen Dokumentes nur *0.1* Sekunden, während die Hashtabelle mit Verkettung<sup>2.2</sup> auf eine Zeit von *0.23* Sekunden kommt. Mit **Mit größerem M ist die Hashtabelle mit einfacher Verkettung am schnellsten.**

## Fehler offene Hashtabelle -2

einer Laufzeit von 3.58 Sekunden braucht die Hashtabelle mit linearer Kollisionsbehandlung<sup>2</sup> am längsten. Aufgrund eines Logikfehlers arbeitet diese zwar alle Wörter des Dokumentes durch, zählt diese aber nicht ordnungsgemäß, wodurch das ermittelte häufigste Wort<sup>2</sup> nicht mit dem wirklichen häufigsten Wort<sup>3</sup> übereinstimmt. Bezüglich Speicherplatz ist die Datei für die Hashtabelle mit Verkettung *ModHashTableChaining.pas* am Kleinsten mit 1.59kB. Die binäre Suchbaum-Datei *ModBinarySearchTree.pas* belegt 1.79kB, die Datei für die Hashtabelle mit linearer Kollisionsbehandlung *ModHashTableLinearProbing.pas* ist mit 2.46kB am Größten.

**Speicherverbrauch der Implementierung ist als Metrik wirklich irrelevant**

## 2.2 Güte von Hash-Funktionen

Die Güte mehrerer Hash-Funktionen soll grafisch ermittelt werden. Dabei wird die Datei *KafkaWords.txt* eingelesen und von drei Hash-Funktionen alle Wörter verhasht und in eine Tabelle eingefügt. Dann ist diese Tabelle grafisch auszugeben, durch die jeweilige Häufigkeit eines einzelnen Hashes kann dann die Güte dieses ermittelt werden. Am Idealsten wäre ein konstanter Block, in welchem alle Werte gleich oft vorkommen.

### 2.2.1 Implementierung

In dieser Übung wurden auch die Files *ModStringHash.pas*<sup>2.1.1</sup> und *WordReader.pas*<sup>4</sup> aus der ersten Aufgabe verwendet.

**Listing 2.6:** WordCounter.pas

```

1 (* WordCounter:                                HD0, 2003-02-28 *)
2 (* -----                                     *)
3 (* Template for programs that count words in text files.      *)
4 (*=====*)
5 PROGRAM WordCounter;
6
7   USES WordReader, ModStringHash, ModHash2, ModHash3, graph;
8   const
9     M = 2000;
10  VAR
11    i : integer;
12    w: Word;
13    hash1Table : array[0..M] of integer;
14    hash2Table : array[0..M] of integer;
15    hash3Table : array[0..M] of integer;
16
17 BEGIN (*WordCounter*)
18   OpenFile('Kafka.txt', toLower);
19   ReadWord(w);
20   WHILE w <> '' DO BEGIN
21     ReadWord(w);
22     (* Create 3 hashes *)
23     inc(hash1Table[stringHash1(w)mod M]);
24     inc(hash2Table[stringHash2(w)mod M]);

```

<sup>2</sup>feuerwehrangelegenhe mit 20 Instanzen

<sup>3</sup>und mit 2927 Instanzen

<sup>4</sup>Vorlage aus Moodle

```

25     inc(hash3Table[stringHash3(w)mod M]);
26 END; (*WHILE*)
27 (*INSERT GRAPHICAL REPRESENTATION HERE *)
28
29
30 END. (*WordCounter*)

```

**grafische Repräsentation und  
Ergebnisse fehlen -4**

Listing 2.7: ModHash2.pas

```

1 unit ModHash2;
2
3 interface
4 function stringHash2(w : string) : integer;
5
6 implementation
7 function stringHash2(w:string) : integer;
8   var i : integer;
9       sum : integer;
10 begin
11   sum := 1;
12   for i := 1 to length(w) do begin
13     sum := (sum * Ord(w[i])+Ord(w[i])) mod 32768;
14   end;
15   stringHash2 := sum;
16 end;
17
18 begin
19 end.

```

Listing 2.8: ModHash3.pas

```

1 unit ModHash3;
2
3 interface
4 function stringHash3(w : string) : integer;
5
6 implementation
7 function stringHash3(w : string) : integer;
8   var i : integer;
9       sum : integer;
10 begin
11   sum := 1;
12   for i := 1 to length(w) do begin
13     sum := (sum * Ord(w[i])) mod 32768;
14   end;
15   stringHash3 := sum;
16 end;
17
18 begin
19 end.

```

### 2.2.2 Ausgabe

Da diese Übung auf einem Linux-Rechner ausgearbeitet wurde und die mitgegebenen Module nur auf dem Betriebssystem Windows ausgeführt werden können und nicht

rechtzeitig Ersatz gefunden werden konnte, konnte leider keine grafische Repräsentation der Güte geschaffen werden. **Ich arbeite auch hauptsächlich auf Linux und hab trotzdem immer eine Windows-VM einsatzbereit.**

### 2.2.3 Auswertung

-