

<input type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name <u>Andreas Roither</u>	Aufwand in h <u>5 h</u>
<input type="checkbox"/> Gr. 2, Dr. H. Gruber		
<input checked="" type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte _____	Kurzzeichen Tutor / Übungsleiter _____ / _____

1. Index-Generator

(18 Punkte)

Gesucht ist ein Pascal-Programm *IndexGen*, das für einen gegebenen Text (in einer Textdatei) einen *Index* erzeugt. Ein Index ist die lexikographisch sortierte Liste aller Wörter des Texts, wobei für jedes Wort in aufsteigend sortierter Reihenfolge die Nummern all jener Zeilen angegeben ist, in denen das Wort im Text vorkommt. Dabei ist zwischen Groß- und Kleinschreibung nicht zu unterscheiden, alle Wörter können deshalb z. B. in Kleinbuchstaben umgesetzt werden.

Beispiel:

Text:	Ach wie gut, dass niemand weiß, dass ich Rumpelstilzchen heiß.
-------	---

Index (nur auszugsweise dargestellt):	ach	1
	...	
	dass	1, 2
	...	
	wie	1

Ihr Programm muss mit

```
IndexGen InputFileName.txt
```

aufgerufen werden können (der Name der Textdatei wird also in Form eines Kommandozeilen-Parameters übergeben) und muss den Index auf die Standardausgabe schreiben. Der Index kann dann bei Bedarf mit Hilfe von Ausgabeumleitung auch in eine Datei umgeleitet werden, z. B. mit

```
IndexGen InputFileName.txt > IndexFileName.txt
```

Verwenden Sie eine Hashtabelle zur Verwaltung der Einträge (= Wort mit seinen Zeilennummern). Vor Ausgabe des Ergebnisses sind die Wörter im Index mit Quicksort zu sortieren. Testen Sie Ihre Lösungen ausführlich, indem Sie für das Fachhochschul-Studiengesetz (in der Datei *FHSStG2011.txt* im moodle-Kurs) einen Index generieren und vergleichen Sie Ihre Lösung auch mit jener von KollegInnen um festzustellen, welche effizienter ist. Dazu können Sie das Modul *Timer* (im moodle-Kurs) verwenden. Um längere Laufzeiten zu erhalten, sollten Sie sich auch (wieder einmal?) mit Franz Kafka beschäftigen (siehe *Kafka.txt* moodle-Kurs).

Bemerkungen: Da das Thema Dateibearbeitung noch nicht besprochen wurde, finden Sie im moodle-Kurs in *IndexGen.pas* eine Vorlage für das zu erstellende Programm.

2. Güte von Hash-Funktionen

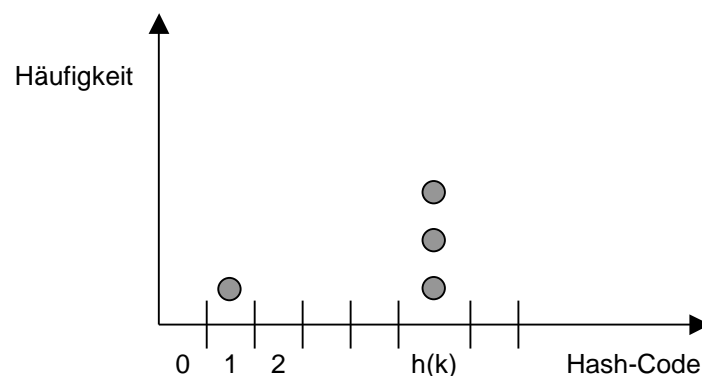
(6 Punkte)

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. Dabei ist eine Gleichverteilung anzustreben.

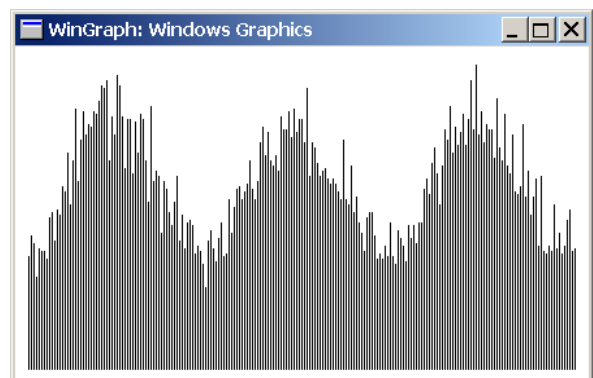
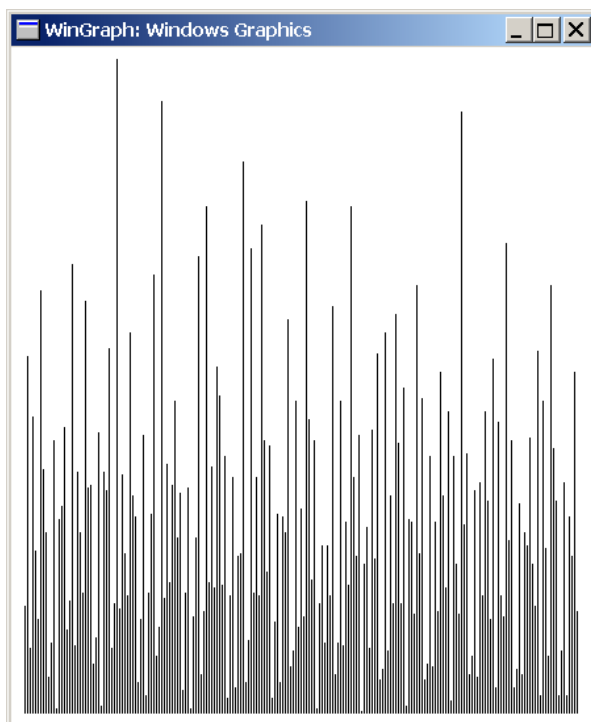
Ende des Wintersemesters wurde ein Pascal-Programm zur einfachen Erstellung von Graphiken unter Windows (*WinGraph.pas* mit dem Testprogramm *WG_Test.pas*) vorgestellt und dazu verwendet, um die Güte von Zufallszahlengeneratoren zu visualisieren (z. B. in Form des Himmels-tests). Benutzen Sie dieses System (im moodle-Kurs) 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 „Das Schloss“ von Franz Kafka). Ändern Sie die Prozedur *Redraw*-Prozedur so ab, 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 und 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*:



Übung 1

Aufgabe 1

Lösungsidee

Die erstellte Hash Tabelle enthält eine Doppelt verkettete Liste, in dieser werden die Zeilennummern der Wörter gespeichert. Da die Hash Tabelle selbst nicht durch eine Quicksort Funktion sortiert werden darf, wird nebenbei beim einfügen eines Wortes in die Hash Tabelle, dasselbe Wort in ein String Array eingefügt. Dieses Array wird dann sortiert und der Reihe nach durch gegangen. Dabei wird der Hashcode des Wortes ausgerechnet und an der entsprechenden Stelle im Array wird nach dem Wort gesucht. Zusätzlich werden dann auch die Zeilennummern ausgegeben.

```

1  (* IndexGen:                                     HDO, 2002-02-28 *)
   (* -----                                     *)
3  (* Generation of a sorted index of all words in a text file. *)
   (* Command line:                               *)
5  (*      IndexGen [ fileName ]                  *)
   (=====*)
7  PROGRAM IndexGen;

9  USES
   WinCrt, Timer;

11
12  CONST
13  EF = CHR(0);          (*end of file character*)
   maxWordLen = 30;      (*max. number of characters per word*)
15  chars = [ 'a' .. 'z', 'ä', 'ö', 'ü', 'ß',
              'A' .. 'Z', 'Ä', 'Ö', 'Ü' ];
17  size = 30000;

19  TYPE
   Word = STRING[maxWordLen];

21
   doubleListPtr = ^listElement;
23  listElement = record
       val : Integer;
25     Prev : doubleListPtr;
       Next : doubleListPtr;
27  end; (*Record*)

29  dList = ^List;
   List = RECORD
31     first: doubleListPtr;
       last: doubleListPtr;
33  END; (*Record*)

35  NodePtr = ^Node;
   Node = RECORD
37     key: STRING;
       data : dList;
39     next: NodePtr;
       END; (*Record*)
41  ListPtr = NodePtr;
   HashTable = ARRAY[0..size-1] OF ListPtr;

```

```

43  VAR
44      txt: TEXT;           (*text file*)
45      curLine: STRING;     (*current line from file txt*)
46      curCh: CHAR;        (*current character*)
47      curLineNr: INTEGER;  (*current line number*)
48      curColNr: INTEGER;   (*current column number*)
49      ht : HashTable;
50      wordArray : ARRAY of STRING;
51      wordCount : Integer;
52
53      (* New has table node *)
54      function NewHashNode(key: String; next: NodePtr; data : dList) : NodePtr;
55      var
56          n: NodePtr;
57      begin
58          New(n);
59          n^.key := key;
60          n^.next := next;
61          n^.data := data;
62          NewHashNode := n;
63      end; (*NewNode*)
64
65      (* New double linked list node *)
66      function NewDLLListNode(val : Integer) : doubleListPtr;
67      var temp : doubleListPtr;
68      begin
69          New(temp);
70          temp^.val := val;
71          temp^.prev := Nil;
72          temp^.next := Nil;
73          NewDLLListNode := temp;
74      end;
75
76      (* init double linked list *)
77      procedure InitDLList(var l : dList);
78      begin
79          l^.first := Nil;
80          l^.last := Nil;
81      end;
82
83      (* append to double linked list *)
84      procedure AppendDLList(var l : dList; val : Integer);
85      var n : doubleListPtr;
86      begin
87
88          if (l^.first = Nil) then
89              begin
90                  n := NewDLLListNode(val);
91                  l^.first := n;
92                  l^.last := n;
93              end
94          else
95              begin
96                  if l^.last^.val <> val then begin
97                      n := NewDLLListNode(val);
98                      n^.prev := l^.last;
99                      l^.last^.next := n;

```

```

101     l^.last := n;
102     end;
103 end;
104
105 (* returns the hashcode of a key *)
106 function HashCode4(key: String): Integer;
107     var
108         hc, i : Integer;
109     begin
110         hc := 0;
111
112         for i := 1 to Length(key) do begin
113             {Q-}
114             {R-}
115             hc := 31 * hc + Ord(key[i]);
116             {R+}
117             {Q+}
118         end; (* for *)
119
120         HashCode4 := Abs(hc) MOD size;
121     end; (* HashCode4 *)
122
123 (* Lookup combines search and prepend *)
124 function Lookup(key: String; val : Integer) : NodePtr;
125     var
126         i: Integer;
127         n: NodePtr;
128         l: dList;
129     begin
130         i := HashCode4(key);
131         //WriteLn('Hashwert= ', i);
132         n := ht[i];
133
134         while (n <> Nil) do begin
135             if (n^.key = key) THEN BEGIN
136                 AppendDlList(n^.data, val);
137                 exit;
138             end;
139             n := n^.next;
140         end;
141
142         if n = nil then begin
143             New(l);
144             InitDLList(l);
145             AppendDlList(l, val);
146
147             n := NewHashNode(key, ht[i], l);
148             ht[i] := n;
149
150             if wordCount >= High(wordArray) then
151                 SetLength(wordArray, (wordCount + 500));
152
153             wordArray[wordcount] := key;
154             wordCount := wordCount + 1;
155         end; (* if *)
156
157         Lookup := n;

```

```

159   end; (* Lookup *)
161
162 FUNCTION LowerCase(ch: CHAR): STRING;
163 BEGIN
164     CASE ch OF
165         'A'..'Z': LowerCase := CHR(ORD(ch) + (ORD('a') - ORD('A')));
166         'Ä', 'ä': LowerCase := 'ae';
167         'Ö', 'ö': LowerCase := 'oe';
168         'Ü', 'ü': LowerCase := 'ue';
169         'ß': LowerCase := 'ss';
170     ELSE (*all the others*)
171         LowerCase := ch;
172     END; (*CASE*)
173 END; (*LowerCase*)
174
175 PROCEDURE GetNextChar; (*updates curChar, ...*)
176 BEGIN
177     IF curColNr < Length(curLine) THEN BEGIN
178         curColNr := curColNr + 1;
179         curCh := curLine[curColNr]
180     END (*THEN*)
181 ELSE BEGIN (*curColNr >= Length(curLine)*)
182     IF NOT Eof(txt) THEN BEGIN
183         ReadLn(txt, curLine);
184         curLineNr := curLineNr + 1;
185         curColNr := 0;
186         curCh := ' '; (*separate lines by ' '*)
187     END (*THEN*)
188     ELSE (*Eof(txt)*)
189         curCh := EF;
190     END; (*ELSE*)
191 END; (*GetNextChar*)
192
193 PROCEDURE GetNextWord(VAR w: Word; VAR lnr: INTEGER);
194 BEGIN
195     WHILE (curCh <> EF) AND NOT (curCh IN chars) DO BEGIN
196         GetNextChar;
197     END; (*WHILE*)
198     lnr := curLineNr;
199     IF curCh <> EF THEN BEGIN
200         w := LowerCase(curCh);
201         GetNextChar;
202         WHILE (curCh <> EF) AND (curCh IN chars) DO BEGIN
203             w := Concat(w, LowerCase(curCh));
204             GetNextChar;
205         END; (*WHILE*)
206         END (*THEN*)
207     ELSE (*curCh = EF*)
208         w := ' ';
209 END; (*GetNextWord*)
210
211 procedure WriteLineNumbers(d : dList);
212 var
213     n : doubleListPtr;
214 begin
215     if d^.first <> NIL then begin
216         n := d^.first;

```

```

217     WHILE (n <> NIL) DO BEGIN
218         Write(n^.val, ' ');
219         n := n^.next;
220     END;
221 end;
222 end;
223
224 procedure WriteHashTable;
225 var
226     h : Integer;
227     n : NodePtr;
228     d : dList;
229 begin
230     for h:= 0 to size-1 do begin
231         if ht[h] <> nil then begin
232             Write(h, ': ');
233             n := ht[h];
234
235             while n <> nil do begin
236                 Write(n^.key, ' ');
237                 WriteLineNumbers(n^.data);
238
239                 n := n^.next;
240             end; (* while *)
241             WriteLn;
242         end; (* if *)
243     end; (* for *)
244 end; (* WriteHashTable *)
245
246 PROCEDURE Swap(VAR a, b : String);
247 VAR
248     temp : String;
249 BEGIN
250     temp := b;
251     b := a;
252     a := temp;
253 END;
254
255 FUNCTION LT(a, b : String) : BOOLEAN;
256 BEGIN
257     LT := a < b;
258 END;
259
260 (* quicksort rec for string arrays *)
261 PROCEDURE QuickSort(VAR arr : ARRAY OF String; n : INTEGER);
262 PROCEDURE QuickSortRec(VAR arr : ARRAY OF String; l, u : INTEGER);
263 VAR
264     p : String;
265     i, j : INTEGER;
266 BEGIN
267     IF l < u THEN BEGIN
268         (* at least 2 elements *)
269         p := arr[l + (u - l) DIV 2]; (* use first element as pivot *)
270         i := l;
271         j := u;
272         REPEAT
273             WHILE LT(arr[i], p) DO Inc(i);
274             WHILE LT(p, arr[j]) DO Dec(j);

```

```

275         IF i <= j THEN BEGIN
277         IF i <> j THEN BEGIN
            Swap(arr[i], arr[j]);
279         END;
            Inc(i);
281         Dec(j);
            END;
283         UNTIL i > j;
            IF j > l THEN QuickSortRec(arr, l, j);
285         IF i < u THEN QuickSortRec(arr, i, u);
            END;
287     END;

289     BEGIN
        QuickSortRec(arr, Low(arr), n);
291     END;

293     procedure PrintStringArray();
    var
295         i : Integer;
        n : NodePtr;
297     begin

299         for i := 0 to wordCount do begin
            WriteLn;
301             Write(wordArray[i], ': ', #9);
            n := ht[HashCode4(wordArray[i])];
303
            while (n <> NIL) AND (n^.key <> wordArray[i]) do begin
305                 n := n^.next;
            end;
307
            if n <> nil then
309                 WriteLineNumbers(n^.data);
            end;
311        end;

313     procedure Init();
    var i : Integer;
315     begin
        SetLength(wordArray, 1000);
317
        for i := 0 to size-1 do begin
319             ht[i] := NIL;
        end;
321    end;

323     VAR
        txtName: STRING;
325         w: Word;           (*current word*)
        lnr: INTEGER;        (*line number of current word*)
327         n: LONGINT;        (*number of words*)

329 BEGIN (*IndexGen*)
    Init;
331     Write('IndexGen: index generation for text file ');

```



```
333 IF ParamCount = 0 THEN BEGIN
    WriteLn;
335 WriteLn;
    Write( 'name of text file > ' );
337 ReadLn(txtName);
END (*THEN*)
339 ELSE BEGIN
    txtName := ParamStr(1);
341 WriteLn(txtName);
END; (*ELSE*)
343 WriteLn;

345 (*—— read text from text file ——*)
Assign(txt, txtName);
347 Reset(txt);
curLine := '';
349 curLineNr := 0;
curColNr := 1; (*curColNr > Length(curLine) forces reading of first line*)
351 GetNextChar; (*curCh now holds first character*)
n := 0;
353 wordCount := 0;

355 StartTimer;
GetNextWord(w, lnr);
357 WHILE Length(w) > 0 DO BEGIN
    //WriteLn(w, ' ', lnr);
359 Lookup(w, lnr);
    n := n + 1;
361 GetNextWord(w, lnr);
END; (*WHILE*)
363 QuickSort(wordArray, wordCount);
PrintStringArray();
365 StopTimer;

367 WriteLn;
WriteLn('number of words: ', n, ' ', wordCount);
369 WriteLn('elapsed time: ', ElapsedTime);
Close(txt);
371 ReadLn;
373 END. (*IndexGen*)
```

IndexGen.pas

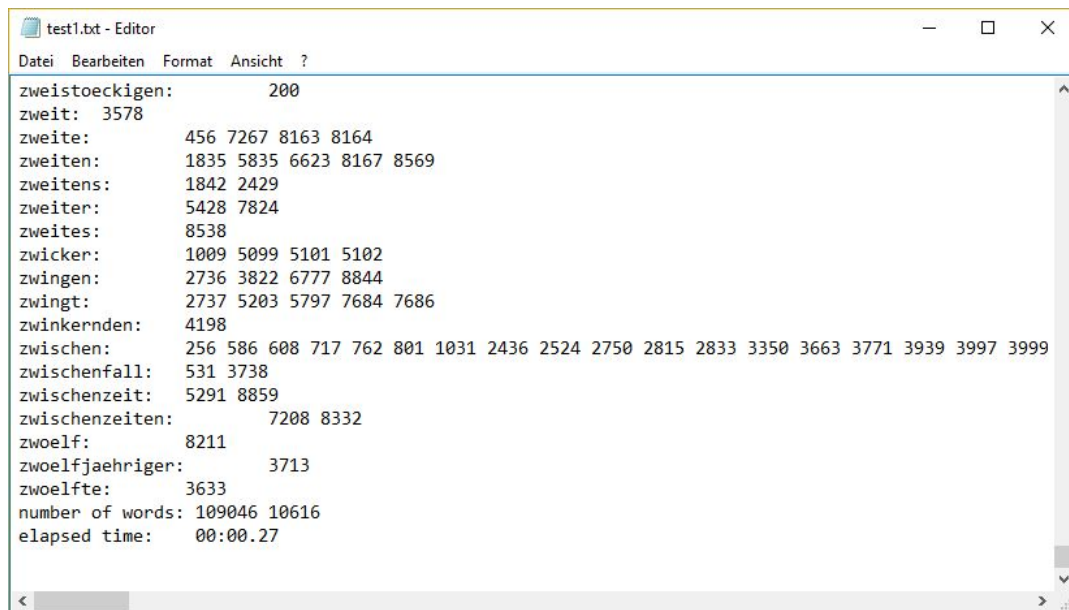


Abbildung 1: Ausgabe

Aufgabe 2

Lösungsidee

Ähnlich wie bei Aufgabe 1 wird eine Hash Tabelle erstellt. Anschließend wird beim zeichnen für jede Position in der Tabelle die Knoten gezählt und gezeichnet. Es werden drei verschiedene Hash Funktionen getestet. Die ungewöhnliche Größe des String Arrays ergibt sich aus dem Testen. Ist die Größe des Arrays durch 3 teilbar, ist die Verteilung in der Tabelle lückenhaft und ähnlich schlecht wie bei der ersten oder zweiten Hash Funktion.

```

2 PROGRAM WG.Hash;
4  USES
   { $IFDEF FPC }
6   Windows,
   { $ELSE }
8   WinTypes, WinProcs,
   { $ENDIF }
10  Strings, WinCrt, WinGraph;

12 CONST
   EF = CHR(0);           (*end of file character*)
14  maxWordLen = 30;       (*max. number of characters per word*)
   chars = [ 'a' .. 'z', 'ä', 'ö', 'ü', 'ß',
16            'A' .. 'Z', 'Ä', 'Ö', 'Ü' ];
   size = 421;

18
20 TYPE
   Word = STRING[ maxWordLen ];

22   NodePtr = ^Node;
   Node = RECORD
24     key: STRING;

```

```

    next: NodePtr;
26 END; (*Record*)
    ListPtr = NodePtr;
28 HashTable = ARRAY[0..size-1] OF ListPtr;

30 VAR
    txt: TEXT;           (*text file*)
32    curLine: STRING;    (*current line from file txt*)
    curCh: CHAR;         (*current character*)
34    curLineNr: INTEGER; (*current line number*)
    curColNr: INTEGER;   (*current column number*)
36    ht : HashTable;
    option : Integer;

38
function NewHashNode(key: String; next: NodePtr) : NodePtr;
40 var
    n: NodePtr;
42 begin
    New(n);
44    n^.key := key;
    n^.next := next;
46    NewHashNode := n;
end; (*NewNode*)

48
(* returns the hashcode of a key *)
50 function HashCode1(key: String): Integer;
begin
52    HashCode1 := Ord(key[1]) MOD size;
end; (*HashCode1*)

54
(* compiler hashcode.. *)
56 function HashCode2(key: String): Integer;
begin
58    if Length(key) = 1 then
        HashCode2 := (Ord(key[1]) * 7 + 1) * 17 MOD size
60    else
        HashCode2 := (Ord(key[1]) * 7 + Ord(key[2]) + Length(key)) * 17 MOD size
62    end; (*HashCode2*)

64
(* returns the hashcode of a key *)
function HashCode3(key: String): Integer;
66    var
        hc, i : Integer;
68    begin
        hc := 0;

70
        for i := 1 to Length(key) do begin
72            {Q-}
            {R-}
74            hc := 31 * hc + Ord(key[i]);
            {R+}
            {Q+}
76            end; (* for *)

78
        HashCode3 := Abs(hc) MOD size;
80    end; (*HashCode3*)

82
(* Lookup combines search and prepend *)

```

```

84  procedure Lookup(key: String);
      var
86      i: Integer;
      n: NodePtr;
begin
88      IF option = 1 THEN i := HashCode1(key)
      ELSE IF option = 2 THEN i := HashCode2(key)
90      ELSE IF option = 3 THEN i := HashCode3(key)
      ELSE BEGIN
92          WriteLn('Invalid option');
          Halt;
94      END;
      n := ht[i];

96      while (n <> Nil) do begin
98          if (n^.key = key) THEN BEGIN
              exit;
100          end;
              n := n^.next;
102      end;

104      n := NewHashNode(key, ht[i]);
      ht[i] := n;
106  end; (* Lookup *)

108  FUNCTION LowerCase(ch: CHAR): STRING;
  BEGIN
110      CASE ch OF
          'A'..'Z': LowerCase := CHR(ORD(ch) + (ORD('a') - ORD('A')));
112          'Ä', 'ä': LowerCase := 'ae';
          'Ö', 'ö': LowerCase := 'oe';
114          'Ü', 'ü': LowerCase := 'ue';
          'ß': LowerCase := 'ss';
116          ELSE (*all the others*)
              LowerCase := ch;
118          END; (*CASE*)
  END; (*LowerCase*)

120  PROCEDURE GetNextChar; (*updates curChar, ...*)
  BEGIN
122      IF curColNr < Length(curLine) THEN BEGIN
124          curColNr := curColNr + 1;
          curCh := curLine[curColNr]
126      END (*THEN*)
      ELSE BEGIN (*curColNr >= Length(curLine)*)
128          IF NOT Eof(txt) THEN BEGIN
              ReadLn(txt, curLine);
130              curLineNr := curLineNr + 1;
              curColNr := 0;
132              curCh := ' '; (*separate lines by ' '*)
              END (*THEN*)
              ELSE (*Eof(txt)*)
134                  curCh := EF;
136          END; (*ELSE*)
  END; (*GetNextChar*)

138  PROCEDURE GetNextWord(VAR w: Word; VAR lnr: INTEGER);
  BEGIN
140

```

```

142   WHILE (curCh <> EF) AND NOT (curCh IN chars) DO BEGIN
      GetNextChar;
143   END; (*WHILE*)
144   lnr := curLineNr;
145   IF curCh <> EF THEN BEGIN
146     w := LowerCase(curCh);
      GetNextChar;
147     WHILE (curCh <> EF) AND (curCh IN chars) DO BEGIN
148       w := Concat(w , LowerCase(curCh));
149       GetNextChar;
150     END; (*WHILE*)
151     END (*THEN*)
152   ELSE (*curCh = EF*)
153     w := '';
154   END; (*GetNextWord*)
155
156   (* Counts nodes *)
157   FUNCTION CountNodes(n : ListPtr) : INTEGER;
158   VAR
159     count : INTEGER;
160   BEGIN
161     count := 0;
162     WHILE n <> NIL DO BEGIN
163       Inc(count);
164       n := n^.next;
165     END;
166     CountNodes := count;
167   END;
168
169   (* Draw function with eclipse *)
170   PROCEDURE Draw(table : HashTable; dc : HDC; r : TRect);
171   VAR i, j : INTEGER;
172     stepX : REAL;
173     w, h : INTEGER;
174     maxVal, count : INTEGER;
175     x, y : REAL;
176     hFactor : REAL;
177
178   BEGIN
179     w := r.right - r.left;
180     h := r.bottom - r.top;
181     count := 1;
182     maxVal := count;
183
184     FOR i := Low(table) TO High(table) DO BEGIN
185       count := CountNodes(table[i]);
186       IF maxVal < count THEN BEGIN
187         maxVal := count;
188       END;
189     END;
190     IF maxVal = 0 THEN BEGIN
191       maxVal := 1;
192     END;
193
194     stepX := w / (High(table) - Low(table) + 1);
195     hFactor := (h / stepX) / maxVal;
196     x := r.left;
197     count := 0;

```

```

200     FOR i := Low(table) TO High(table) DO BEGIN
        count := CountNodes(table[i]);
202     y := r.bottom;
        FOR j := 1 TO Round(hFactor * count) DO BEGIN
204         Ellipse(dc, Round(x), Round(y + stepX), Round(x + stepX), Round(y));
            y := y - stepX;
206         END;
            x := x + stepX;
208         END;
        END;
210
        (* Function to call the actual drawing function *)
212     PROCEDURE DrawHash(dc: HDC; wnd: HWND; r: TRect);
        BEGIN
214         Draw(ht, dc, r);
        END;
216
        PROCEDURE Init;
218     VAR
        i : INTEGER;
220     BEGIN
        FOR i := 0 TO size - 1 DO BEGIN
222         ht[i] := NIL;
        END;
224     END;
    VAR
226     txtName: STRING;
        w: Word;      (*current word*)
228     lnr: INTEGER;   (*line number of current word*)
        n: LONGINT;   (*number of words*)
230
    BEGIN
232     option := 1;
        n := 0;
234     Init();

236     IF ParamCount = 0 THEN BEGIN
        WriteLn;
238         WriteLn;
        Write('name of text file > ');
240         ReadLn(txtName);
    END ELSE BEGIN
242         txtName := ParamStr(1);
        WriteLn(txtName);
244     END;
        WriteLn;

246
        WriteLn('Choose HashFunction: ');
248         WriteLn('1. func (bad)');
        WriteLn('2. func (better)');
250         WriteLn('3. func (best)');
        Write('> ');
252         ReadLn(option);

254     Assign(txt, txtName);
        Reset(txt);
256     curLine := '';

```

```
258  curLineNr := 0;  
    curColNr := 1;  
    GetNextChar;  
260  
    GetNextWord(w, lnr);  
262  WHILE Length(w) > 0 DO BEGIN  
        LookUp(w);  
264    n := n + 1;  
        GetNextWord(w, lnr);  
266  END;  
  
268  redrawProc := DrawHash;  
    WGMain;  
270  
    WriteLn;  
272    WriteLn('number of words: ', n);  
  
274  Close(txt);  
END.
```

WG_Hash.pas

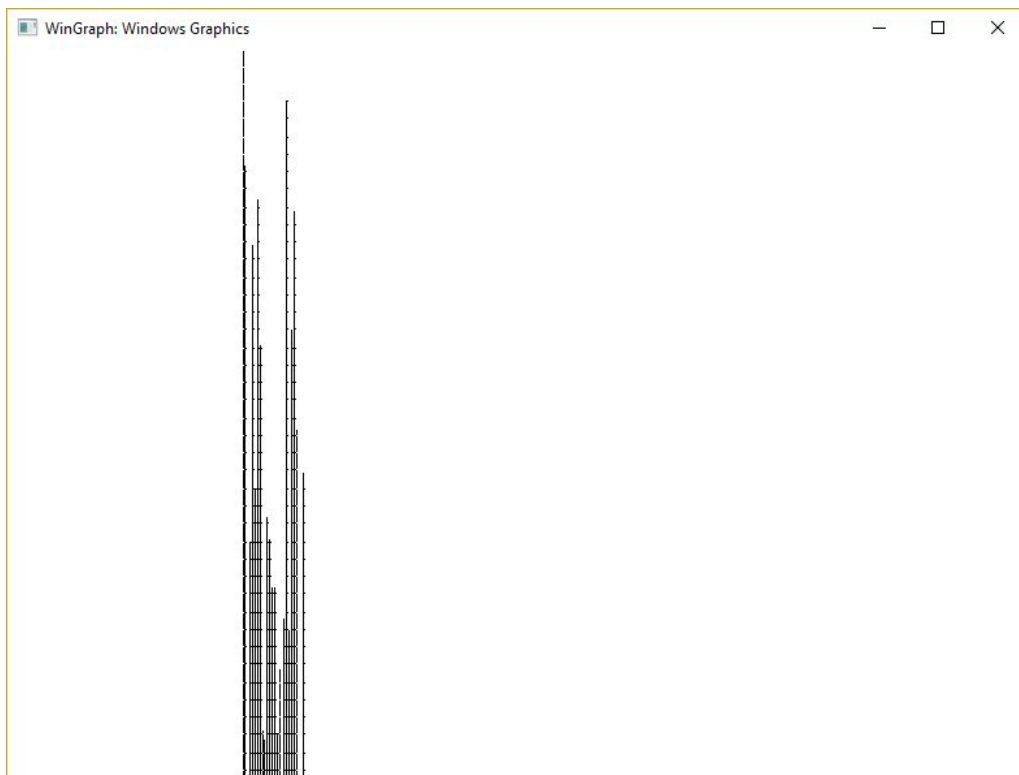


Abbildung 2: Ausgabe

Hier sieht man deutlich das die Verteilung der Elemente in der Tabelle sehr schlecht ist, die Verteilung der Elemente ist sehr nahe beieinander.

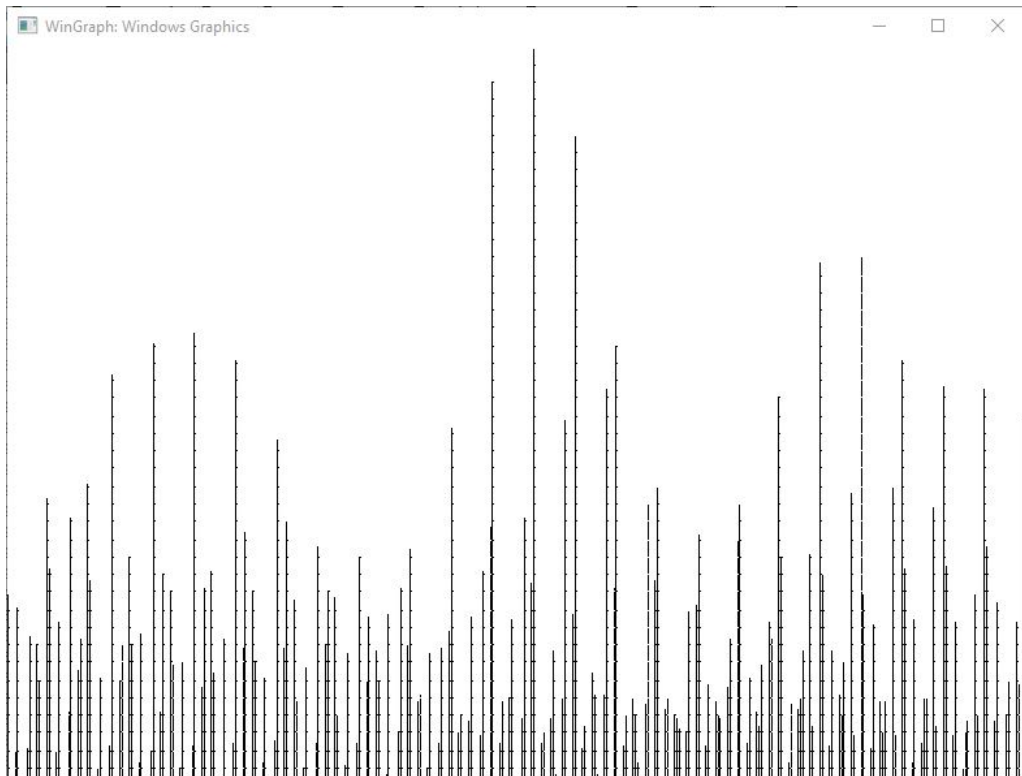


Abbildung 3: Ausgabe

Im Gegensatz zur vorherigen Hash Funktion sieht hier die Verteilung etwas besser aus.

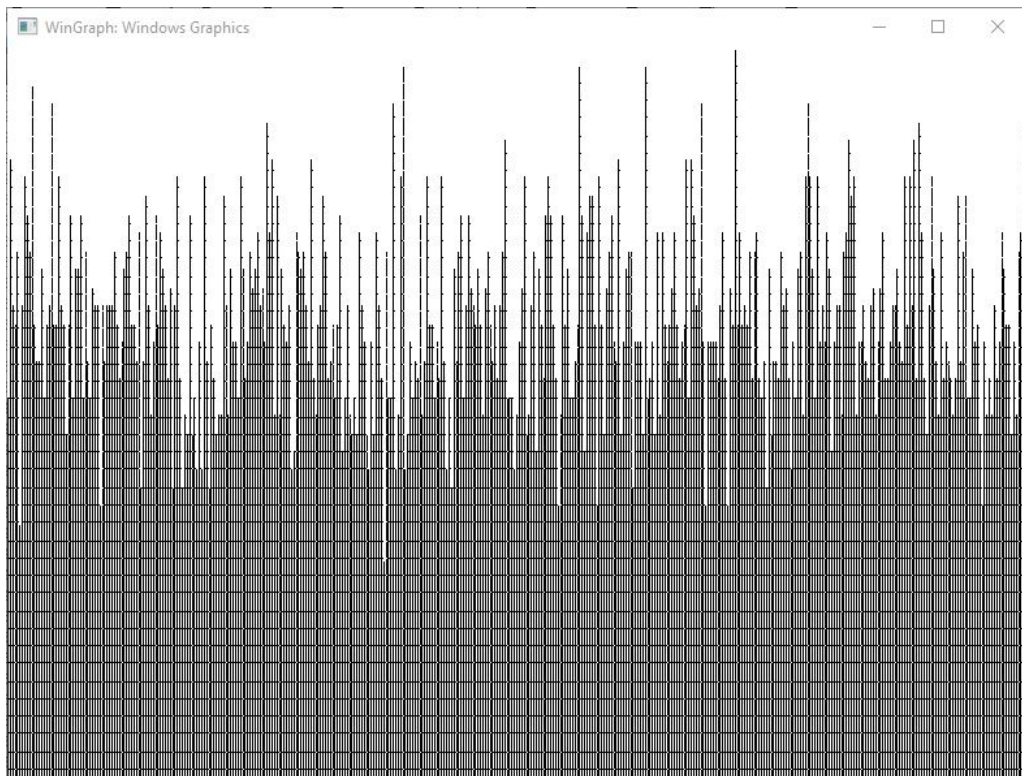


Abbildung 4: Ausgabe

Mit der dritten Hash Funktion sieht man eine bessere Verteilung in der Hash Tabelle.