J. King Saud Univ., Vol. 4, *Eng. Sci.* (1), pp. 105-125 (1412 A.H./1992)

COMPUTER SCIENCE

Arabic Dictionary Compression Using An Invertible Integer Transformation and a Bitmap Representation

M.A. El-Affendi
Department of Computer Science, College of Computer & Information Sciences, King Saud University, P.O. Box 78115, Riyadh 11543, Saudi Arabia
(Received on 7/3/1990; Accepted for publication 18/11/1990)

Abstract. Due to advances in computer technology, language dictionaries are gradually gaining additional importance. Great achievements have been reported in areas such as natural language processing, speech recognition and other AI applications. Most of these applications require the availability of a dictionary that can be maintained and accessed in a very efficient way. Dictionaries are also required in conventional applications such as spelling correction and data base management.

In this paper a flexible, efficient bitmap compression method for Arabic dictionaries is described. The method is based on an invertible integer transformation and may either be used to store root-based dictionaries or stem-based dictionaries. In the first case each root may cost up to 5 bits at most (less than two bits per character) if all Arabic roots are stored. Practically, only one bit is needed to store the whole root, but the additional bits are due to the sparsity of the resulting structure. It is quite feasible that the number of bits per root may be reduced to less than 2. For the second type of dictionary 17 bits at most are needed to represent a stem up to 7 characters in length (less than 3 bits per character). Again this figure may be improved if the sparsity problem is solved. The operations required to maintain the dictionary are very simple and very cheap in terms of processor time. The flexibility of the resulting structure is reflected in the fact that an unlimited number of attributes may be added as parallel arrays or bitmaps.

Introduction

Dictionaries are now commonly used in many computer applications such as spelling correction [1], data base management systems, natural language processing [2, pp. 295-344] and several other areas. The structure of the dictionary and the amount of information kept in it varies from one application to another. For some applications the dictionary is just a glossary of proper words in the language that may be used to check if a given word is a member of the language or not, and if the word is wrong provides some alternatives (*e.g.* Spelling correction programs). In some other cases

the dictionary is loaded with some additional information such as syntactic and semantic attributes. This is generally the case in most natural language processing and knowledge acquisition systems.

As for the Arabic language, dictionaries are of special nature. The organization of most classical Arabic dictionaries such as Lissan El-Arab [3] are based on word roots rather than on the word itself. This is mainly due to the strong derivational properties that characterise the Arabic language. It is well known that, apart from few exceptions, any Arabic word may easily be reduceed to a simple root using a well defined invertible procedure. This means that instead of storing the whole word in the dictionary one can just store the corresponding root and rely on the established measures and methods to generate the word or analyse it [4]. This convention has also been adopted in many computerised systems today [5, pp. 247-332]. The root orgnisation method greatly reduces the size of the dictionary, but clearly the morphological analysis (synthesis) process is a big burden on computerised systems, especially those which just need to check interactively if a word is proper or not.

In this paper, a flexible, efficient bitmap compression method is described. The method is based on an inveretible integer transformation and may either be used to compliment the root compression technique or independently to store all possible stems in the language. The second option may be taken in the case of interactive applications where morphological analysis may greatly affect the response time. The basic terminology is introduced in section 2. Observations on Arabic word structure and dictionary organisation are given in section 3. The bitmap compression method is described in section 4. Conclusions and recommendations are given in section 5.

Basic Definitions and Terminology

According to Arabic morphology, most Arabic words has the following general structure:

$$\text{word} = \text{prefixes} + \text{stem} + \text{suffixes} \qquad (1)$$

where the prefixes and suffixes are picked from a finite, well-defined set of affixes in the language (the plus sign is used here to indicate concatenation). Generally, a word may take up to three prefixes and up to three suffixes. Examples of prefixes are (الـ، سـ، اسـت). Examples of suffixes are (ون، ين، أن، هم).

The stem (جـذع) of the word is usually derived from the root (جـذر) using a standard measure (ميزان صرفي). The root is the atom from which words are composed using standard measures and affixes. The root consists of the minimum number of characters beyond which the word will cease to have an independent meaning. Dropping any character from the root makes it meaningless. Most Arabic roots are triliteral or tetraliteral [5, p 132].

Observations on Arabic Word Structure and Dictionary Organization

The Arabic language is characterised by some interesting morphological properties that makes it unique among its current counterparts. The following are some of these properties:

i) Apart from few exceptions, any proper word in the language may be reduced to a corresponding root in the language.
ii) The total number of roots in classical Arabic [5,p 132] is slightly less than ten thousands.
iii) 70% of the above roots are triliteral, 27% are tetraliteral and only 2% are pentaliteral [5,p 132].
iv) The number of words that may be generated from a single root are 120 on average.
v) Words are generated from roots using a regular, invertible process that may be programmed. Given a root, one can easily generate any of the corresponding words using appropriate measures and affixes. On the other hand, given any word the corresponding root may easily be traced by simply reversing the process.
vi) The number of characters in a standard stem may rarely exceed seven [6,p 67]. In the majority of cases it is below six.

The Compression Method

The compression method adopted in this paper is a two-stage process:
a) first, each string in the dictionary is transformed into a unique finite integer using a simple, invertible integer transformation.
b) secondly, the integer representation obtained in (a) is then transformed into a single bit on a bitmap using set operations.

An invertible integer transformation for character strings

In order for the method to work properly and for the dictionary to be useful, the transformation function has been chosen such that the following conditions are satisfied:

i) uniqueness

The integer representation obtained for each string should be unique. Two strings may have the same integer representation only if they are equal.

ii) optimality

The integer representation obtained for a string should be as small as possible. This is important since the size of integers on most computers is very restricted. Note that here we mean constrained optimality. The major constraint here is the invertability condition below.

iii) invertability

It should be possible to obtain the string back from its equivalent integer representation by simply inverting the process.

iv) operation preservence

The transformation should preserve most of the known string operations such as insertion, deletion, concatenation ... etc. This is important because in many applications users may need to generate new strings by simple manipulation of existing strings.

To satisfy the uniqueness condition (i), it has been decided that each alpha character is a unique digit in some number system whose radix is equal to the total number of alpha characters needed to represent strings in the dictionary. Neglecting vowels and assuming that various shapes of an Arabic letter reduce to the same internal code and including the white space character, the total number of characters needed to represent words in the dictionary will be around 31 (here we differentiate between internal representation and external shape of characters. *e.g.* The hamza has four shapes which logically reduce to one. However, the vowels may be needed to obtain the external shape). This means that the radix of the required number system should at least be 31. However, to satisfy the invertability condition (iii), the radix should be a power of 2. This will guarantee compatibility with the internal binary representation of integers and greatly simplify the extraction of strings from the equivalent integer representation. Consequently it has been decided to use 32 instead of 31 as a radix for the needed number system. This implies that 5 bits are needed to represent each character instead of 7 or 8 (as in most character codes). It seems that this is the most we can do to satisfy the optimality condition (ii) at this stage. Note that in the second phase of this process this figure will be reduced to less than two bits per character for triliteral roots. According to this scheme 8 bytes will be enough to store the integer representation of a 12-character string.

Having identified the required number system, the transformation is simply achieved by viewing each word as a pentadecimal integer whose most significant numeral is the first character and the least significant numeral is the last character. Each character is then converted into an equivalent penta numeral using a table like the table shown in appendix A. Before stating the transformation algorithm, let us introduce some data structures. Note that all data structures and algorithms below will be introduced directly in turbo Pascal to facilitate easy communication and understanding. If the reader has any problems in understanding one of the algorithms he is advised to refer to any standard reference on turbo Pascal *e.g.* [7]. At this stage two data structures are needed:

```
ARAB_CH: ARRAY [0..31] OF CHAR; {THIS ARRAY HOLDS THE
         ARABIC CHARS.}
VALUE : ARRAY[128..165] OF INTEGER;
```

{THE VALUE ARRAY HOLDS THE PENTA VALUES OF ARABIC CHARACTERS INDEXED BY THEIR POSITIONS IN THE CHARACTER CODE TABLE (128 - 165)

The above arrays may be created using the following piece of code:

```
ARAB_CH[0]:=''; {THE FIRST CHARACTER IS THE NULL CHARACTER}
FOR I:=1 TO 30 DO
BEGIN
    READLN(ARAB_CH[I]);
    VALUE[ORD(ARAB_CH[I]):=I; {PENTA VALUE BECOMES I}
END;
```

Here ORD is the Pascal ordinal function that returns the position of the character in the character code table.

In view of this the pentadecimal transformation may be expressed as follows:

Algorithm (1)

```
    FUNCTION H(WORD: STRING): LONGINT;
VAR  K: INTEGER; TEMP: LONGINT; {LONGINT REQUIRES 4 BYTES}
BEGIN
    TEMP:=0;
    FOR I:=1 TO LENGTH(WORD) DO
    TEMP:=TEMP*32+VALUE[ORD(WORD[K])];
    H:=TEMP;
END;
```

Here WORD is the dictionary entry to be tranformed and H is the equivalent pentadecimal representation. To illustrate how the above algorithm works consider the word (كتب). The penta values corresponding to letters ب، ت، ك are 24,5 and 3 respectively (from right to left). The above algorithm will compute H as:

$$H:= (1*32*32)+(2*32)+3 \longrightarrow 24739 \qquad (2)$$

A string whose length is less or equal to 6 may be extracted from its equivalent integer representation using the following simple function:

Algorithm (2)

```
    FUNCTION RET-STRING (H_VALUE: LONGINT): STRING:
VAR TEMP_H: LONGINT; RST: STRING;
N,PENTA_NUMERAL: INTEGER;
BEGIN
```

```
    RST:=''; {RST is the string to be obtained from H_VALUE}
    N:=LOG 32(H_VALUE); { N is the expected length of RST}
    {LOG 32 is a function that gives the rounded log to base 32}
    WHILE N>=0 DO
    BEGIN
    {Obtain the penta numeral for the current character}
    PENTA_NUMERAL:=H_VALUE DIV POWER (32,N);
    {The POWER function raises 32 to the power N -Concatenate character to
    string}
    RST:=RST+ARAB_CH[PENTA_NUMERAL];
    {Get H_VALUE for remaining substring}
    H_VALUE:=H_VALUE MOD POWER(32,N);
    N:=N-1; {Consider next character}
END;
RET_STRING:=RST;
END;
```

For an implementation of the function LOG32 and POWER see appendix B. The main reason behind the restriction on the number of characters is that the DIV and MOD operators are restricted to integers which occupy less than 4 bytes on most computers. However, longer strings may easily by handled by simulating the DIV and MOD operations for double integers and other types of numerical data using the INT function and related arithmetic operations. Alternatively, a long string may be decomposed into several segments. For Arabic, six characters are more than enough in many cases.

For an implementation based on shift operations for the above algorithm see appendix C.

Operations preserved by the transformation

In this section it is shown that most string operations are preserved under the tranformation H described in the above section. These include:

a) Character insertion

As explained earlier, under this transformation each character is represented by a numeral in the pentadecimal system. Each penta numeral is equivalent to 5 bits on the binary scale. To insert a new character we simply insert the equivalent penta numeral in the proper position. *e.g.* To insert a character at position P from the beginning one may use the following algorithm:

Algroithm (3)

```
PROCEDURE INSERT(X: CHAR; P: INTEGER; WORD: STRING);
                {Insert a character X at position P in WORD}
```

```
VAR
H_VALUE, HIGH, LOW: LONGINT;
N: INTEGER;
BEGIN
    H_VALUE:=H(WORD);
    N:=LOG32(H_VALUE);
        {Split word into HIGH and LOW}
    HIGH:=H_VALUE DIV POWER(32,N-P+1); {Strip off the low N-P+1
        characters and leave P-1 characters to form HIGH}
    LOW:=H_VALUE MOD POWER(32,N-P+1);
    HIGH:=HIGH*32+PENTA-NUMERAL[ORD(X)]; {Insert character}
    H_VALUE:=HIGH*POWER(32,N-P+1)+LOW; {Merge HIGH and LOW}
    WORD:=RET_STRING(H-VALUE); {Obtain new word}
END;
```

This scheme may easily be generalised to insert k characters at any position in the string. For an implementation using shift operations see appendix C.

b) Character deletion

A similar algorithm may be used to delete a character from any position in a given string. To delete a character at position P from the beginning the following algorithm may be used:

Algorithm (4)

```
    PROCEDURE DELETE (X: CHAR; P: INTEGER; WORD: STRING);
VAR
H_VALUE, HIGH, LOW: LONGINT;
N: INTEGER;
BEGIN
    H_VALUE:=H(WORD);
    N:=LOG32(H_VALUE);
    HIGH:=H_VALUE DIV POWER (32,N-P);
    {High consists of the first P characters}
    LOW:=H_VALUE MOD POWER (32, N-P);
        {Low consists of the remaining N-P characters}
        {Now delete the character at position P}
    HIGH:-HIGH DIV 32;
    H_VALUE:-HIGH*POWER(32,N-P)-LOW; {Merge HIGH and LOW}
    WORD:=RET_STRING(H_VALUE);
END;
```

Another implementation for the delete operation based on shift operations is given in appendix C.

c) String padding

According to this scheme padding may be effected by adding zeros at the end of the H-value for the string. As shown in Appendix A, the zero penta value corresponds to the NULL character. To add the equivalent of K NULL characters at the end of a given H value, one simply performs the following operation:

$$H:=H\ SHL\ (5*K); \tag{3}$$

d) Pattern matching and string ordering

It is interesting to note that under this transformation pattern matching is reduced to a single integer operation. However, it is important to note that if strings are to be ordered in some way then comparisons under H may lead to proper results only if the involved strings are of the same length. Therefore, the value of H for shorter strings should be adjusted before comparison using the padding operation described in c above.

e) String concatenation

Examples of string concatenation have already been encountered in (a) and (b) above. These are the last steps of algorithms (3) and (4). Generally, to concatenate a string S2 of length N to another string S1 one performs the following:

$$H(NEW\_STRING):=H(S1)\ SHL\ (5*N) + H(S2); \tag{4}$$

A bitmap representation for the dictionary

This section describes the second phase of the dictionary implementation where the H value of each root is transformed into a single bit on a bitmap scale. The implementation is strongly based on Pascal set structures, although some other alternative structures may be employed. To understand the algorithms and the discussion below a deep understanding of the set structure and its implementation is required. For this purpose the reader is referred to reference [8, pp. 23-34]. The reader is also referred to the manual of turbo Pascal version 5 in order to understand the basic concepts of type casting.

It is assumed that the dictionary is just a list of proper words in the language without vowels. This type of dictionary is useful in many applications such as spelling correction, natural language processing, data base management... etc. This assumption greatly simplifies the technique presented here without much loss in generality. The bitmap representation discussed here may later be extended to include other attribute bits and bytes whenever needed. It is also assumed that all written shapes of an Arabic character reduce to the same character code. In this way one ends up with 31 characters, including the NULL character. Given these facts an Arabic dictionary may be implemented using one of the following options or a combination of both:

A bitmap for triliteral Arabic roots

The triliteral roots have the property that the corresponding value of H is always in the range (1000,30000). This is because the first Arabic character corresponds to a penta value of 1, and even if the first triliteral root consists exclusively of the first character then according to algorithm (1) one gets:

$$H:=1+(1*32)+(1*32*32) \text{ which is } > 1000 \qquad (5)$$

A similar argument may be used to show that the H value for the last root in the Arabic dictionary is less than 3000. It is also known that the total number of triliteral roots in the language is in the range (6000,7000). This means that if we use a bitmap consisting of 29000 bits to represent about 6500 roots then each root requires less than 5 bits in the worst case. Practically the number of bits per root may be reduced to less than 4 bits per root.

The bitmap is implemented using the Pascal set structure. The bitmap is viewed as an array structure BITMAP[1..1000] consisting of a finite number of SET_UNITS, where each SET_UNIT is defined as a set structure consisting of 32 bits. Initally all bits in each SET_UNIT are set to zero. To add a root to the bitmap one simply uses the root H value to compute the index of the appropriate SET_UNIT and the offset inside that unit. The index of the appropriate SET_UNIT is computed using the operation H DIV 32 while the offset inside the SET_UNIT is computed using the operation H MOD 32. The position corresponding to the offset inside the SET_UNIT is then set to 1 indicating the presence of a root. As an example the K^{th} SET_UNIT may look as follows:

SET_UNIT NO. K:

00001000011100110001010101000011

Note that the above SET_UNIT indicates the presence of 12 roots whose H values start at (32*K)+4 and finish at (32*K)+31.

It is important to note here that this representation for the 12 roots may be stored in the computer as a 32-bit single integer using type casting operations. Therefore, the whole bitmap may be saved as a sequence consisting of about only 1000 32-bit integers, and this is the main attraction of the process (See algorithm (6) below).

Algorithm (5)

Data structures: It is assumed that the following data structures are defined at the top of the enclosing program:

```
TYPE
    S = ↑ SET_UNIT;
    SET_UNIT= SET OF 0..31; {members of the bitmap}
    Q = ↑ LONGWORD;
VAR
    BITMAP = ARRAY [1..1000] OF SET_UNIT;
    {It has been felt from the example dictionary that at most 1000 SET_UNITS are
    required to store the whole dictionary}
    I,K,R: INTEGER;
    P: POINTER;
    M: Q;
```

Algorithm (6)

Adding a root to the bitmap

```
PROCEDURE ADD_ROOT (ROOT: STRING);
VAR
    H_VALUE: LONGINT; K,R: INTEGER;
BEGIN
    H_VALUE:=H(ROOT); {compute H using algorithm 4.1}
    H_VALUE:=H_VALUE – MIN; { numbers below MIN do not correspond to
    proper roots}
    K:=H_VALUE DIV 32; {compute index of appropriate SET_UNIT}
    R:=H_VALUE MOD 32; {compute the offset inside the SET_UNIT}
    BITMAP[K]:= BITMAP[K]+[R]; {set position R in the SET_UNIT to 1}
END;
```

Algorithm (7)

Saving the bitmap:

In the algorithm below type casting operations are used to store the bitmap. Each SET_UNIT in the bitmap is tranformed into an integer pointed to by M (see the declarations above)

To write the bitmap to a file one performs the following:

```
FOR K:=1 TO 1000 DO
BEGIN

    P:=@(BITMAP[K]); {take the address of a bitmap Kth element and store it in
    P}
    M:=P; {convert the bitmap unit into an equivalent longword pointed to by the
    pointer M}
    WRITE (FILENAME,' ',M ↑ ); {save the bitmap element}
END;
```

Note here that the symbol '@' is the address operator that takes the address of an object, and the symbol '↑' is a qualify operator indicating the object pointed to by the preceding pointer *e.g.* M ↑ is used to indicate the object pointed to by the pointer M.

Algorithm (8)

loading the bitmap: To load the bitmap from a file one simply reverses the above procedure:

```
FOR K:=1 TO 1000 DO
BEGIN
    READ(FILENAME,X); {X is a longword}
    P:=@(X); {store the address of X in pointer P}
    S:=P; {convert the longword back into SET_UNIT}
    BITMAP[K]:=S ↑ ;
END;
```

Note that the above algorithms assume that all the bitmap units include some elements. However, if the structure is sparse then the algorithms may be modified to compress the parts that include consecutive empty bitmap units. The value of the longword corresponding to an empty bitmap unit will be zero. To compress the empty parts, one may simply count the number of consecutive zeros and write the count as a negative integer in the list.

If the reader has any problems in understanding the above algorithms he should consult the references sited at the beginning of this section.

A bitmap for other roots and stems

If a morphological analyser is used, then the above bitmap is enough to verify if a given word is a proper member of the language or not. Similarly, the above bitmaps may be used with a morphological synthesizer to generate proper words in the language. However, current morphological analysers and synthesizers may be quite slow for interactive applications such as spelling correction. In such cases it may be more appropriate to generate a bitmap for all actively used stems corresponding to each of the above mentioned roots. According to observation (vi) of section 3 above, these stems are six characters long on average. The problem here is that H is a bit large [in the range $(32^5,32^6)$] and the resulting structure may be very sparse. This problem also arises in the cases of tetraliteral and pentaliteral roots. To avoid this the following partial bitmap may be used:

Algorithm (9)

i) if word length > 3 then divide the word into two segments: a HEAD and a TAIL. The TAIL is composed of the last three letters while the HEAD is used to store the first part of the word (one to four characters may be accommodated in the HEAD).

ii) Compute H for each segment using algorithm (1) above.

iii) use a bitmap similar to the ROOT bitmap Eq. (5) to represent the TAIL part of all words. In many cases the TAIL is a root or close to a root.

iv) use a word array to store H for the HEAD part corresponding to each TAIL. The size of the array should be equal to the number of bits in the TAIL bitmap. If a TAIL has more than one HEAD then a negative integer may be used to indicate the place of all corresponding HEADs in some additional structure. *i.e.* an array entry may be positive or negative. If it is positive then it represents the value of H for a unique HEAD. If it is negative then it indicates the presence of more than one HEAD for the corresponding TAIL. The place of these HEADs in some additional structure is obtained by simply negating the entry.

v) The additional structure may be a simple table consisting of several entries for each TAIL. Empty areas may be compressed by just storing the number of zero entries.

Assuming that there may be up to 100000 proper Arabic stems, the above scheme may be organised to accommodate all these stems in a very efficient way. More statistics are needed to produce a very efficient representation.

Adding vowels

Vowels are very important in the study of Arabic morphology and they should be included with the word. The bitmap representation method described above allows the inclusion of vowels as a separate compressed attribute of any element in the dictionary. This is achieved by adopting a tetradecimal number system where:

0 stands for sukoon
1 stands for fatha
2 stands for dhamma
3 stands for kassra

The radix of this number system is 4, which is a power of 2. In this way, we can use a transformation similar to algrorithm (1) to convert the vowels of a given word into a single small integer. Later the transformation may be inverted to obtain the vowels back using an algorithm (2). The vowels are an important aid in restricting the total number of characters to 31.

Note that the above vowel representation system may be extended to include tanween and shadda. In that case the octal number system will be more appropriate. However, for morphological purposes, the above vowels are enough. Tanween is relevant only in case of syntax analysis and text formatting. It is also well known that the presence of a shadda indicates that the associated character is doubled and that

the first occurrence of this character is associated with a sukoon. This fact may be used to implement the shadda implicitly.

Implementation Issues

The above ideas have been used to implement an actual Arabic dictionary consisting of about 10000 roots. 70% of these are triliteral roots. It is interesting to note that only 4000 bytes were needed to store around 7000 triliteral roots *i.e.* less than two bits per character. As for the remaining roots the structure is less efficient but still better than conventioanl methods. The roots were taken from the famous Arabic dictionary "EL QAMOOS EL-MUHEET". Sections of the implementation program are given in appendices B and C. The whole program is available on request.

Conclusions

A bitmap compression method has been proposed for Arabic dictionaries. The method is based on an invertible integer transformation and may be used to store triliteral roots or full stems. There is no reason why the method may not be used to store other attributes for roots and words. The method does not only provide an efficient storage scheme, but also simplifies the search process for words and roots and offers a strong basis for morphological analysis. It is also believed that the method will have wide implications in the area of text editing and encryption. Perhaps the major gain here is that the word is now an integer entity that may easily be manipulated and transformed.

The main problem encountered is that, due to the invertability constraint, the size of the transformation H increases dramatically as the number of characters in the world increases. This leads to very sparse bitmaps with large empty spaces. More work is needed to solve this problem.

References

[1] Peterson, J. *Design of a Spelling Program. An Experiment in Program Design.* Berlin, Heidelberg, New York: Springer Verlag, 1980.

[2] Rich, E. *Artificial Intellegence.* London, Singapore: McGraw Hill, 1983.

[3] Ibn Manzour, Jamal El Din. *Lissan El-Arab.* Beirut: Dar Sadir, 1300H.

[4] Al-Fedaghi, S. and Al-Anzi, F. "A New Algorithm to Generate Root-Pattern Forms." *The 11th National computer conference,* Dhahran, Saudi Arabia, (1988), 391-400.

[5] Ali, Nabil. *The Arabic Language and the Computer.* Riyadh: Ta'reeb, 1988.

[6] El-Hamlawi, Ahmed. *Shetha El-Arf Fi Fan El-Sarf.* Beirut: Educational Books Library, 1988.

[7] Hergert, D. *Mastering Turbo Pascal 5.* Singapore: Tech Publications for Sybex Inc., 1988.

[8] Wirth, N. *Algorithms + Data Structrures = Programs,* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1976.

Appendix (A)

Arabic Character Table

Penta value	Ord [ch]	Arab-ch
1	128	ء
2	134	ا
3	135	ب
4	136	ة
5	137	ت
6	139	ث
7	140	ج
8	142	ح
9	143	خ
10	144	د
11	145	ذ
12	146	ر
13	147	ز
14	148	س
15	149	ش
16	150	ص
17	151	ض
18	152	ط
19	153	ظ
20	154	ع
21	155	غ
22	156	ف
23	157	ق
24	158	ك
25	159	ل
26	160	م
27	161	ن
28	162	ه
29	163	و
30	165	ي

Appendix (B)

Implementation of Major Algorithms (1) – (4) Using MOD, DIV and Power Operations

To Run the Program you Need Turbo Pascal Ver. 5 and Musaid El–Arabi Ver. 2 (2 مساعد العربي)

```
PROGRAM TEST 1 (INPUT, OUTPUT);
  USES TP_API;
  VAR
    A_CHAR, CH: CHAR;
    MR, ITEM, RT, ITEM 2: STRING;
    Y, Y2: LONGINT;
    P, I: INTEGER;
    VALUE: ARRAY [128 .. 165] OF INTEGER;
    ARAB_CH: ARRAY [0 .. 31] OF STRING [1];
    EP, GP, FP1, GP1, FP2: TEXT;

    FUNCTION POWER (M: LONGINT; N: INTEGER): LONGINT;

      POWER FUNCTION IMPLEMENTATION

    VAR
    P: LONGINT;
    BEGIN
      P: = ROUND (EXP (N*LN(INT(M))));
      POWER: = P;
    END;

    FUNCTION LOG32 (X: LONGINT): INTEGER;

      LOG32 FUNCTION IMPLEMENTATION

    BEGIN
      LOG32: = ROUND (LN(X)/LN(32));
    END;

    FUNCTION H(WORD: STRING): LONGINT;

      ALGORITHM (1) : THE TRANSFORMATION H
```

```
    VAR
    K: INTEGER;
    TEMP: LONGINT;
    BEGIN
        TEMP: = 0;
        FOR K: = 1 TO LENGTH (WORD) DO
        TEMP: = TEMP* 32 + VALUE [ORD (WORD[K])];
        H: = TEMP;
    END;
```

```
    FUNCTION RET_STRING (X: LONGINT): STRING;
```

ALGORITHM (2) : INVERTING THE TRANSFORMATION H

```
  VAR
    H_VALUE: LONGINT; RST: STRING;
    N,PENTA_NUMERAL: INTEGER;
    BEGIN
        RST: = '';
        H_VALUE: = X;
        N: = LOG 32 (H_VALUE);
        WHILE N > = 0 DO
        BEGIN
            PENTA_NUMERAL: = H_VALUE DIV POWER (32, N);
            RST: = RST + ARAB–CH[PENTA–NUMERAL];
            H_VALUE: = H_VALUE MOD POWER (32, N);
            N: = N–1;
        END;
        RET_STRING: = RST;
  END;
```

```
PROCEDURE INSERT (X: CHAR; P: INTEGER; VAR WORD: STRING);
```

ALGORITHM (3) : THE INSERTION PROCEDURE

```
  VAR
    H_VALUE, H_VALUE2, HIGH, LOW: LONGINT;
    N: INTEGER;
    BEGIN
        H_VALUE: = H (WORD);
        N: = LOG 32 (H_VALUE);
        HIGH: = H_VALUE DIV POWER (32, N–P + 1);
```

```
        LOW: = H_VALUE MOD POWER (32, N–P +1);
        HIGH: = HIGH*32 + VALUE[ORD(X)]; { INSERT CHARACTER}
        H_VALUE: = HIGH*POWER (32, N–P + 1) + LOW;
        WORD: = RET_STRING (H_VALUE);
        WRITELN (WORD);
    END;
```

```
PROCEDURE DELETE (P: INTEGER; VAR WORD : STRING);
```

ALGORITHM (4): THE DELETE PROCEDURE

```
  VAR
    H_VLAUE, HIGH, LOW: LONGINT;
    N: INTEGER;
    BEGIN
        H_VALUE: = H (WORD);
        N: = LOG32 (H_VALUE); WRITELN (N);
        HIGH: = H_VALUE DIV POWER (32, N–P);
        LOW: = H_VALUE MOD POWER (32, N–P);
        HIGH: = HIGH DIVE 32;
        H_VALUE: = HIGH*POWER (32, N–P) + LOW;
        WORD: = RET_STRING (H_VALUE);
    END;
  BEGIN
    ASSIGN(FP,'"ARB.LET"'); RESET (FP);
    ASSIGN(GP, "ARB. RES"); REWRITE (GP);
    SSIGN(FP1, "MSR. TXT"); RESET (FP1);
    ASSIGN(GP1, "MSR. RES"); REWRITE (GP1);
    ARAB_CH[0]: = '';
    FOR I: = 1 TO 30 DO
    BEGIN
        READLN(FP,CH); ARAB-CH[I]: = CH;
        {ARABIC CHARACTERS ARE SUPPOSED TO BE IN THE FILE
         ARB.LET}
        VALUE [ORD (CH)]: = I;
        WRITELN(GP, ARAB_CH[I],'          ',ORD(CH),'          ',I);
   END;
    READLN(ITEM); {TEST H AND RET_STRING}
    Y: = H(ITEM); RT: = RET_STRING (Y); WRITELN(Y,'          ',RT);
    READLN(ITEM); READLN(A_CHAR); READLN(P);
    {TEST INSERT}
    INSERT (A_CHAR, P, ITEM); WRITELN(' ITEM AFTER
    INSERTION=',ITEM);
```

```
    READLN(ITEM); READLN(P);
    {TEST THE DELETION PROCEDURE}
    DELETE(P, ITEM); WRITELN(' ITEM AFTER DELETION =', ITEM);
  END.
```

Appendix (C)

Implementation of Major Algorithms Using Shift Operations

To Run the Program You Need Turbo Pascal Version and Musaid El-Arabi Ver. 2 (2 مساعد العربي)

```
PROGRAM TEST2 (INPUT, OUTPUT);
  USES TP_API;
  VAR
    CH,A_CHAR: CHAR;
    MR, ITEM, RT, ITEM2, A-WORD: STRING;
    Y, Y2: LONGINT;
    I,P: INTEGER;
    VALUE: ARRAY [128 .. 165] OF INTEGER;
    ARAB_CH: ARRAY [0 .. 31] OF STRING [1];
    FP, GP, FP1, GP1, FP2: TEXT;
  ---------------------------------------------------
    FUNCTION LOG32 (X: LONGINT): INTEGER;
  ---------------------------------------------------
      LOG32 FUNCTION IMPLEMENTATION
  ---------------------------------------------------
      BEGIN
        LOG32:= ROUND (LN(X)/LN(32));
      END;
  ---------------------------------------------------
    FUNCTION H(WOED: STRING): LONGINT;
  ---------------------------------------------------
      ALGORITHM (1): THE TRANSFORMATION H
  ---------------------------------------------------
      VAR
      K: INTEGER;
      TEMP: LONGINT;
      BEGIN
        TEMP:=0;
        FOR K:=1 TO LENGTH (WORD) DO
        TEMP:=TEMP*32+VALUE [ORD(WORD [K])];
        H:=TEMP;
      END;
```

```
FUNCTION RET_STRING (X: LONGINT): STRING;
```

ALGORITHM (2): INVERTING THE TRANSFORMATION H

```
  VAR
    H_VALUE: LONGINT; RST: STRING;
    N, PENTA_NUMERAL: INTEGER;
    BEGIN
      RST:='';
      N:=LOG32 (X); WRITELN ('N=',N);
      H_VALUE:=X SHL (5*(6-N)+2); {STRIP LEADING ZEROS}
      WHILE ABS (H_VALUE)>0 DO
      BEGIN
        PENTA_NUMERAL:=H_VALUE SHR 27;
        RST:=RST+ARAB_CH [PENTA_NUMERAL];


    H_VALUE:=H_VALUE SHL 5;
  END;
  RET_STRING:=RST;
END;
```

```
PROCEDURE INSERT (X: CHAR; P: INTEGER; VAR WORD: STRING);
```

ALGORITHM (3) : THE INSERT OPERATION

```
{ TO INSERT A CHARACTER AT POSITION P}
VAR
  H_VALUE, H_VALUE2, HIGH, LOW: LONGINT;
  N: INTEGER;
BEGIN
  H_VALUE:=H(WORD); N:=LOG32 (H_VALUE);
  H_VALUE2:=H_VALUE SHL (5*(6-N)+2); {STRIP LEADING ZEROS}
  HIGH:=H_VALUE2 SHR (32-5*(P-1));
  LOW:=(H_VALUE2 SHL (5*(P-1))) SHR (32-5*(N-(P-1)));
  HIGH:=HIGH*32+VALUE [ORD(X)]; {INSERT CHARACTER AT
  POSITION P}
  H_VALUE:=(HIGH SHL (5*(N-(P-1))))+LOW;
  WORD:=RET_STRING (H_VALUE);
END;
```

```
PROCEDURE DELETE (P: INTEGER; VAR WORD: STRING);
```

ALGORITHM (4) : THE DELETE OPERATION

```
  VAR
    H_VALUE, HIGH, LOW: LONGINT; N: INTEGER;
    BEGIN
      H_VALUE:=H(WORD); N:=LOG32 (H_VALUE);
      H_VALUE:=H_VALUE SHL (5*(6-N)+2);
      HIGH:=H-VALUE SHR (32-5*P); {HIGH CONSISTS OF CHARS UP
      TO POS P}
      LOW:=(H-VALUE SHL (5*P)) SHR (32-5*(N-P));
      HIGH:=HIGH DIV 32;
      H_VALUE:=HIGH SHL (5*(N-P)) + LOW;
      WORD:=RET_STRING(H_VALUE);
    END;
```

MAIN PROGRAM

```
BEGIN
   ASSIGN(FP, 'ARB. LET'); RESET (FP);
   ASSIGN(GP, 'ARB. RES'); REWRITE (GP);
   ASSIGN(FP1, 'MSR. TXT'); RESET (FP1);
   ASSIGN(GP1, 'MSR. RES'); REWRITE (GP1);
   ARAB_CH [0]:='';
   FOR I:=1 TO 30 DO
   BEGIN
     READLN (FP,CH); ARAB_CH[I]:=CH;
     VALUE [ORD (CH)]:=I;
  END;
  READLN (ITEM);
  Y:=H (ITEM); RT:=RET-STRING (Y); WRITELN(Y,'      ',RT);
  READLN(A_WORD); READLN(A_CHAR); READLN(P);
  INSERT(A_CHAR, P, A_WORD); WRITELN(A_WORD);
  READLN (ITEM); READLN(P);
  DELETE(P, ITEM); WRITELN (ITEM);
  CLOSE(FP); CLOSE(GP);
END.
```

ضغط ألفاظ المعجم العربي باستخدام دالة قابلة للانعكاس وتمثيل ثنائي

محمد أحمد الأفندي

قسم علوم الحاسب، كلية علوم الحاسب، جامعة الملك سعود، ص.ب ٧٨١١٥، الرياض ١١٥٤٣، المملكة العربية السعودية

ملخص البحث. نتيجة للتقدم الكبير في تقنية الإلكترونيات، توسعت تطبيقات الحاسوب في الأعوام الأخيرة لتشمل مجالات جديدة لم تكن سهلة الارتياد من قبل. فمثلاً أصبح الحاسوب اليوم يستخدم في تمييز معاني العبارات اللغوية العادية المقروء منها والمسموع كما صار يستخدم في اكتساب الخبرة والمعرفة وفي التعليم والتدريب وغيرها من تطبيقات الذكاء الاصطناعي. ومن المعروف أن معظم هذه التطبيقات تعتمد في الأساس على توفر معجم لغوي يشمل معظم مفردات اللغة مع كثير من خواصها وملحقاتها. كما أن المعجم أصبح أداة أساسية في بعض التطبيقات التقليدية مثل تصحيح الأخطاء الإملائية وضغط النصوص المُخزّنة. وحتى تحتفظ هذه التطبيقات بفاعليتها وجدواها لابد من تخزين المعجم بصورة فعّالة تضمن سرعة الوصول إلى عناصره وعدم استهلاك حيز كبير من الذاكرة.

في هذا البحث نقدم طريقة مبسّطة لتخزين المعجم العربي بصورة توفر كثيراً من الذاكرة المطلوبة لتخزينه وتضمن سرعة العمليات التي تجري عليه. حسب هذه الطريقة تحوّل كل كلمة في المعجم إلى رقم صحيح باستخدام دالّة قابلة للانعكاس. بعد ذلك يحوّل الرقم الصحيح إلى رقم ثنائي واحد باستخدام عمليات المجموعات. للحصول على الكلمة مرة أخرى ما علينا إلا معرفة موضعها في التركيبة الثنائية الناتجة ثم نعكس الدالّة للحصول على الكلمة كاملة. والبحث ينطوي على شرح وافٍ للطريقة مع توضيح أهميتها المستقبلية.