Implementation of Trie Structure for Storing and Searching of English Spelled Homophone Words

Dr. Vimal P.Parmar [*], Dr. CK Kumbharana [**]

[*] Research Scholar, Department of Computer Science, Saurashtra University, Rajkot, Gujarat, INDIA
[**] Head, Guide, Department of Computer Science, Saurashtra University Rajkot, Gujarat, INDIA

Abstract- Searching is the fundamental computing task in computer applications. Storage and data structure has greater impact on retrieval of data. Numbers of implementation are available to efficiently store and manipulate data. The selection of such appropriate structure depends on the nature of the application and the way in which data are manipulated. Key plays a vital role for searching data. In most of the implementation search is performed based on entire key value. Trie structure is a data structure that performs storing and searching based on individual character or digit that composite a key. Trie structure is suitable for many applications where efficient searching is the prime requirement and one such an application is to storage of homophone. This paper implements a trie structure for storage and retrieval of English homophones. Homophone is the term used to describe words which have similar type of pronunciation having different spelling and meaning. Many computer applications require manipulation of homophones. In this research paper an effort has been made to implement trie structure to store and search homophones. The paper describes two implementation algorithms for constructing a trie structure and then searching from the constructed structure. Phonetic algorithms are used to determine phonetic identity of the words but once identified it is required to store them efficiently as well as search them efficiently. The paper is concluded with resultant sample data set.

Index Terms- Trie structure, m-ary Tree, Storage Structure, Data Structure, Searching, Phonetic Matching, Pronunciation, Homophone

I. INTRODUCTION

Two basic operations of major computer applications are storing data and searching them efficiently. Various data structures are used to make searching data more efficient and effective. If an attempt is being made for efficient storage and manipulation of data then searching specific data requires more efforts. Similarly, if searching of data is being made efficient then storage and manipulation if data requires more efforts. For example widely used searching technique is binary search which performs according order function of logarithm but when inserting an element it must be inserted at proper position to maintain the list ordered either ascending or descending. In contrast to binary search, linear search performs insertion of an element very easily without concerning of maintaining order but searching a linear list is an order of total number of elements. Thus trade off is required between inserting and searching element. Further most of the searching is applied on entire key value comparison. This paper models and implements a trie structure which performs insertion as well as searching based on individual character rather than entire search key value. Homophones are the words having similar pronunciation but different spelling and different meaning. Many algorithms are available to identify homophones as per the application requirement. In this paper several homophone are stored and searched. Homophones have some similarity in their spellings structure, trie structure is used to store and search homophones because trie structure stores and search according to individual character positions. Homophones can be stored and manipulated in number of different ways including neural network, grouping them or using special indexing schemes. Trie structure is chosen due to its individual character manipulation to determine the specific position in a structure. A model is proposed to insert and search homophone and then it is implemented. The model is tested using set of several homophones.

II. TRIE STRUCTURE INTRODUCTION

In computer science field data structure determines the organization of data and how these data are manipulated. Trie structure is a special kind of data structure in which data are organized as per the individual characters of key element. In any other implementation of tree branching occurs based on entire key value where as in trie structure branching occurs according to a part of an entire key value. Trie structure is also referred to as an m-ary tree due to number of branches according to individual characters. Branching at each level depends on a particular character position. Here for the sake of simplicity key consists of only alphabetic characters.

An three dimensional character array is used to store the key values. Te specific position is calculated based on the first character. Row is identified by alphabetic positions starting from A to Z and column is identified by a branching level number. First character of key value determines the row number and initial column number 1 where it will be stored. Whenever second key with the same character encountered same position is calculated which is occupied by the previous key term. So now branching occurs. It is not possible to store two keys at a single position so next level is computed and which replaced the previous key value. Now to store the first key value, second character is selected to determine the row and column number is the level number here 2 for example where the first key will be allocated. To store second key again second character position is determined to select the row in column level 2. This process is

repeated for any number of keys with limited to the storage defines for the trie structure. Here the structure is flat and looks like a matrix form but the values stored along with the level number in matrix creates branching for the storage and hence the name m-ary tree. Implementation requires calculating of rows and columns to determine the key position. Once the trie structure is constructed it can be searched in better way with the similar kind of processing. Homophone words are stored and searched using trie structure according to designed model algorithm. This processing will not identify whether the given words re homophones or not but merely stores and searches identified homophones using any of the available algorithms.

III. Phonetic Algorithms

Many phonetic algorithms are developed to determine phonetic similarity. Few of them are listed below.

Soundex

The soundex algorithm was developed by Robert C. Russell and Margaret K. Odell in 1918[2]. The soundex algorithm determines phonetic similarity using four character coded string for the given words. The first character among them represents the first alphabet of the given English word and remaining three characters are digits according to the given word.

Daitch-mokotoff soundex

A modified algorithm of original soundex is D-M soundex developed in 1985 by Gary mokotoff and then improved for efficiency by Randy Daitch to compare surnames of Slavic and German languages using the six digit numeric code for the given word[1,5].

Kolner phonetic

This algorithm is similar to that of soundex algorithm but was designed for German words rather than English words[1].

Metaphone, Double metaphone and Metaphone 3

Metaphone algorithm was originally developed by Lawrence Phillips in 1990 which determine phonetic similarity using a string of three characters derived from the given word. Then enhanced metaphone algorithm was developed by him to support other languages and was known as the double metaphone. Again next variation of algorithm was designed in 2009 to achieve greater accuracy in identifying homophones.

NYSIIS

New York state Identification and Intelligence System in short NYSIIS phonetic algorithms, which was developed in 1970 with higher accuracy than soundex algorithm.

Match Rating Approach

The match rating Approach MRA is a phonetic identification algorithm concludes using the distance among characters of words and was developed by Western Airlines in 1977 for preparing indices and matching homophonous names[1].

Caverphone

The Caverphone phonetic algorithm was developed in 2002 by David Hood at the University of Otago in New Zealand and then was revised in 2004 which was created for the purpose of data matching between late 19th century and early 20th century electoral rolls to commonly recognize the names[1].

Any of the algorithms can be used or combining them to determine the phonetic similarity between given words. In this paper homophone words are directly taken which are common English words. These words are used to store as key in a trie structure. Trie structure prepares and organizes data using individual character positions. Homophone have more similarity in their spelling so trie structure is selected to store such homophone.

Construction of model to prepare Trie Structure For Homophones

A model is prepared to construct trie structure that will hold homophones. Number of data structures exist to store and search efficiently but due to phonetic similarity and hence having some spelling similarity trie structure is selected to store and retrieve homophones.

Trie structure instead of entire key value search it uses index based on the individual character position to store and retrieve element. Following model represents construction and insertion of homophone in a trie structure.

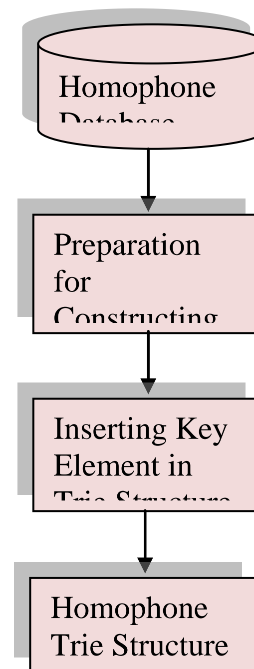

Figure 1 : Construction of Homophone Trie Structure

First it is required to have a set of homophones available which is ready to be inserted in a trie structure. To determine the homophones either of the available common phonetic algorithms can be useful. Then data structure is selected to construct and store these homophones. To determine the character position, alphabet character string is required to locate the position of homophone. Once the empty location is found it can be stored otherwise next level is created to store. After the trie structure is

created it can be searched using the similar manner. Detailed process is implemented in a following algorithm.

IV. Implementation of Homophone Trie Structure

Following algorithm implement the construction of a trie structure.

PROCEDURE TRIE_HOMOHONE (KEY, ROW, COL, TRIE) : This procedure insert KEY in a trie structure TRIE represented using three dimensional character array. ROW and COL represents row and column without any initial value but after successful insertion ROW and COL contains the position where the KEY is inserted and on unsuccessful both of them contain value zero. The detailed algorithm is listed given below which contains some of the housekeeping variables.

```
PROCEDURE TRIE_HOMOPHONE (KEY, ROW, COL, TRIE)
BEGIN
S = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
COL = 1
LEN = LENGTH(KEY)
REPEAT FOR K = 1 TO LEN
BEGIN
     ROW = INDEX(S, KEY[K])
     IF TRIE[ROW][COL] IS EMPTY THEN
    TRIE[ROW][COL] = KEY
    RETURN
IF TRIE[ROW][COL] = KEY THEN
    PRINT "DUPLICATE KEY VALUE"
    ROW = COL = 0
    RETURN
IF INDEX("0123456789", TRIE[R][C][1])  <> 0 THEN
     COL = TRIE[ROW][COL]
ELSE
     LEVEL = LEVEL + 1
     TMP = TRIE[ROW][COL]
     TRIE[ROW][COL] = LEVEL
     COL = LEVEL
     ROW = INDEX(S, TMP[K+1])
     TRIE[ROW][COL] = TMP
END REPEAT
RETURN
END PROCEDURE
```

FUNCTION INDEX (S, C) : This function searches the first occurrence of character C in string S consisting of alphabets A to Z and returns the index number if found otherwise zero. For simplicity all the KEY entries are considered as upper case entries.

```
FUNCTION INDEX (S, C)
BEGIN
LEN = LENGTH (S)
REPEAT FOR I = 1 TO LEN
BEGIN
    IF S[I] = C THEN
        RETURN I
END REPEAT
RETURN 0
END FUNCTION
```

First a variable S is initialized with string of ordered alphabets from A to Z. Initial value of COL is set to 1 means level no 1. Length is calculated and stored in LEN for the given KEY which is to be inserted in a structure. Process begins by using a looping structure from K = 1 to LEN. Initially TRIE is empty.

For example if KEY is “CHECK” then length LEN = 5. Now K = 1 at first iteration. INDEX is a function returns the index of the specified character if found otherwise zero. If character is ‘A’ then index is 1, if ‘B’ then index is ‘2’ and so on if character is ‘Z’ then index is 26.Index is used to determine the ROW number of the KEY to be placed in a trie structure. So INDEX(S, ‘C’) will return 3 as index which is assigned to ROW. So now COL = 1 and ROW = 3. First if tests whether TRIE[ROW][COL] is empty so in this case it will return true and stores KEY = “CHECK at position TRIE[3][1].

Now second homophone “CHEQUE” is inserted KEY = “CHEQUE”. Same process is repeated. At first iteration again it will calculate ROW = 3 and COL = 1 because INDEX (S, ‘C’) will return 3. But now in this case TRIE[3][1] is not empty because it is occupied by previous KEY “CHECK” . So condition will be failed and it will test second if statement to test whether occupied KEY entry is same as the given current KEY. If so it will print the message of duplicate key and return. Here both the entries are different as stored key is “CHECK” and current KEY to be inserted is “CHEQUE” so condition will become false. Now third if statement will test whether occupied entry contains any digit or not using the function call to INDEX (“0123456789”, TRIE[ROW][COL][1]). If digit is found then move to that column using variable COL. But in this case it contains KEY value “CHECK” so now else part will be executed. Initially LEVEL is 0 so due to collision it is increased to next level so now LEVEL = 1. TMP will hold previous KEY “CHECK” stored at TRIE[ROW][COL] which will be updated with level number LEVEL = 1. Now COL = 1 and ROW is calculated as ROW = 8 using function INDEX(S, ‘H’) where position of character ‘H’ is 8. Now TRIE[8][1] is assigned a previous key “CHECK”. This process is repeated for K = 2 so now ROW = 8 due to INDEX(S, ‘H’) for new KEY so again COL = 1 and ROW = 8 which is occupied by KEY “CHECK”. So again LEVEL is increased to LEVEL = 2 and COL = 2 with ROW = 5 due to the third character ‘E’ has an index 5. So “CHECK" is moved to TRIE[5][2] and TRIE[8][1] = 2. In next iteration K = 3 due to third character matching tie occurs. So now ROW = 5 and COL = 3 so TRIE[ROW][COL] contains key value “CHECK”. Same process is repeated and level is increased by 1 so now LEVEL = 3 so COL = 3 and ROW = 3 due to INDEX (S, ‘C’) the key “CHECK” is moved too TRIE[3][3] and TRIE[5][2] = 3. Now K = 4 in next iteration so now ROW = 17 due to INDEX(S, ‘Q’) and COL = 3 which is an empty entry so “CHEQUE” will be stored at location TRIE[17][3]. This process is repeated for all the entries of all the homophones.

Searching model of key element from homophone trie structure

Once the trie structure is constructed it can be searched for a given key homophone. Following model depicts the process of searching from trie structure.

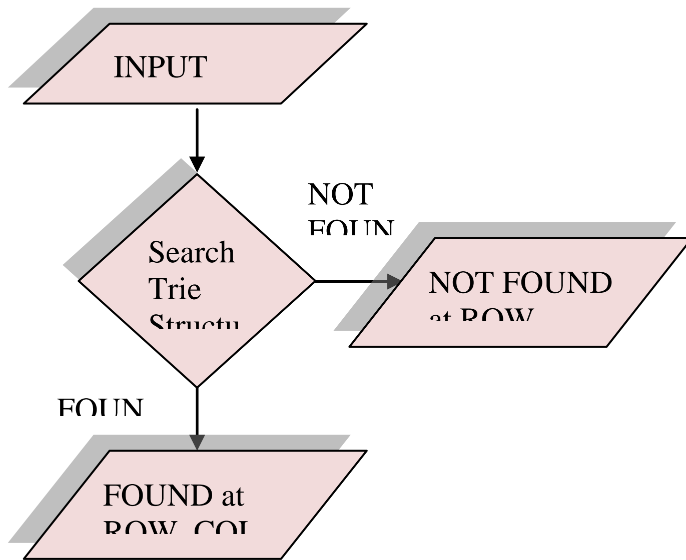

Figure 2: Searching key from Homophone Trie Structure

To search key element from a constructed trie structure it follows the similar process as of insertion procedure. It tries to search an element from level 1 that is COL = 1 and ROW = Index of first character and if an occupied entry is KEY element then search process is over with success and if number found search moves to next level and continues the process until an element is found or failed due to not found. The detailed searching process is listed in the following algorithm.

V. Implementation of SEARCHING KEY Homophone FROM Trie Structure

Following algorithm implements the searching key element from homophone trie structure.

PROCEDURE TRIE_SEARCH (KEY, ROW, COL, TRIE)

This procedure searches given KEY from trie structure TRIE and if KEY is found then sets ROW and COL accordingly otherwise it assigns 0 to both meaning element is not found. This procedure also uses the alphabet array from A to Z to determine the ROW and uses the described INDEX function.

```
PROCEDURE TRIE (KEY, ROW, COL, TRIE)
BEGIN
S = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LEN = LENGTH (KEY)
COL = 1
REPEAT FOR K = 1 TO LEN
BEGIN
ROW = INDEX (S, KEY[K])
      WHILE TRIE[ROW][COL] IS NOT EMPTY
      BEGIN
           IF TRIE[ROW][COL] = KEY THEN
                  RETURN
           IF INDEX ("0123456789",
TRIE[ROW][COL][1] ) = 0 THEN
                  ROW = COL = 0
                  PRINT "UNEXPECTED KEY"
                  RETURN
              ELSE
                  COL = TRIE[ROW][COL]
      END WHILE
END REPEAT
ROW = COL = 0
RETURN
END PROCEDURE
```

This procedure searches given KEY element starting with first column COL = 1 and first character's index ROW. Then it repeats until TRIE[ROW][COL] is not empty. First it tests whether it contains KEY and if it is then returns indicating KEY is found at ROW and COL. If it is not found at that position it will test whether entry at TRIE[ROW][COL] contains any digit means move to next level that is next column which is stored at TRIE[ROW][COL] and which is the ELSE part of the second IF. If it does not contain digit means it is occupied by some other entry and returns failure by printing message and setting ROW = COL = 0 as in IF statement. This process is repeated until conclusion is found.

For example consider the same example as given in insertion. If KEY = "CHECK" then searching starts from ROW = 3 and COL = 1 where level number 1 is stored. It is digit so COL = 1 and ROW = 8 due to second character 'H'. Now TRIE[8][1] contains next level number 2 so now COL = 2 and ROW = 5 as third character is 'E'. Now TRIE[5][2] will be tested which contains level 3 so COL = 3 and ROW = 3 as 4th character in KEY is 'C'. So now TRIE[3][3] is tested with KEY. Now match is found so search process is terminated with ROW = 3 and COL = 3. In each of the above steps while loop terminates when an empty entry is found and search begins from next character by selecting ROW as index number of next character. If all the characters are scanned and no element is found then at last it display the message KEY not found and terminates the search procedure by assigning zero to ROW and COL.

VI. Sample DATA Test, Result And Storage

The entire process of insertion and searching homophone is tested with sample data set and accordingly found how the data is stored in a trie structure. Following figure represents how 20 homophones are organized in a trie structure using implantation of entire process described in algorithm. Following list of homophones are tested and where actually they are located is depicted in the following trie structure.

1. CHEQUE
2. CHECK
3. SINE
4. SIGN
5. PIECE
6. PEACE
7. FOUR
8. FOR
9. SHE

10. SEA
11. DONE
12. RIGHT
13. WRITE
14. CLOCK
15. CLOAK
16. LEAVE
17. LIVE
18. TWO
19. TO
20. GOOD

		1	2	3	4	5	6	7	8	9	10	11	12
1	A										CLOAK		
2	B												
3	C	1		CHECK							CLOCK		
4	D	DONE											
5	E		3		SEA		PEACE					LEAVE	
6	F	7											
7	G	GOOD				SIGN							
8	H	2			SHE								
9	I				5		PIECE					LIVE	
10	J												
11	K												
12	L	11	9										
13	M												
14	N					SINE							
15	O						8					10	TO
16	P	6											
17	Q			CHEQUE									
18	R	RIGHT						FOR					
19	S	4											
20	T	12											
21	U							FOUR					
22	V												
23	W	WRITE											TWO
24	X												
25	Y												
26	Z												

Figure 3 : Homophone Trie Structure Storage Representation

Above representation is a storage structure of sample 20 homophones. Each row number is index based on alphabet and column as level number. Level increases as collision found and branches the structure m number of times that's why it is also known as an m-ary tree which may have m branches in general.

Conclusion

May applications require to store and retrieval of homophones and many data structure exists to fulfill the needs of application. The paper is represented to construct the trie structure to store and search homophones. The algorithm for inserting and searching is implemented and tested for sample data and concluded with storage representation. Search becomes efficient due to character by character position finding approach instead of comparing entire key value and hence improvement of searching can be achieved. On drawback of this approach is the storage requirement. Also time and space complexity for any algorithm are in reverse proportional. So to achieve time efficiency it is required to sacrifice space efficiency. The process described here assumes key value only uppercase alphabetic but the algorithms can be improved to store and retrieve keys having digits as well any other special characters.

REFERENCES

[1] Phonetic Comparison Algorithms By BRETT KESSLER Washington University in St. Louis Volume 103:2 (2005) 243–260

[2] Analysis and Comparative Study on Phonetic Matching Techniques, Rima Shah, Dheeraj Kumar Singh, IJCA Volume 87 – No.9, February 2014

[3] Name and address matching strategy – White Paper December 2010

[4] Vimal P Parmar and C K Kumbharana. Article: Study Existing Various Phonetic Algorithms and Designing and Development of a working model for the New Developed Algorithm and Comparison by implementing it with Existing Algorithm(s). International Journal of Computer Applications 98(19):45-49, July 2014

[5] Vimal P Parmar , C K Kumbharana and Apurva K Pandya - Determining the Character Replacement Rules and Implementing Them for Phonetic Identification of Given Words to Identify Similar Pronunciation Words. Futuristic Trends on Computation Analysis and Knowledge Management (ABLAZE) 2015 International Conference at Greater Noida, India Pages : 272-277 Print ISBN : 978-1-4799-8432-9 DOI : 10.1109/ABLAZE.2015.7155010

[6] An Introduction to data structures with applications - by Jean-Paul Tremblay and Paul Sorenson

[7] Programming in ANSI C – E Balagurusamy

[8] Let Us C – Yashawant kanitkar

AUTHORS

First Author – Dr. Vimal P.Parmar, Research Scholar, Department of Computer Science, Saurashtra University, Rajkot Gujarat, INDIA , parmarvimal1976@yahoo.co.in

Second Author – Dr. CK Kumbharana, Head, Guide, Department of Computer Science, Saurashtra University Rajkot Gujarat, INDIA, ckkumbharana@yahoo.com