"PERFORMANCE OF DATA STRUCTURES ON STRING SEARCH"

MS. URVASHI KUMARI
RESEARCH SCHOLAR
PUNE

DR. SARIKA SHARMA
PROFESSOR & DIRECTOR
JSPM'S ENIAC INSTITUTE OF
COMPUTER APPLICATIONS
WAGHOLI, PUNE

INTRODUCTION

A text is a string or set of strings. Let $\sum$ be an finite set of alphabet and both the query string and the searched text are array of elements of $\sum$. The $\sum$ may be a usual human alphabet (for example A-Z of English alphabet) or binary alphabet ($\sum = \{0,1\}$) or DNA alphabet ($\sum = \{A,C,G,T\}$) in bioinformatics. It is considered that the text is an array T[1...n] of length n and the query string is distinct array S[1...m] of length m and that m<=n. The problem of string matching is to find if the string S is found in text T as fig 1.1

The Text T

T	H	I	S		I	S		A		S	T	R	I	N	G
										S	T	R	I	N	G

The string S

DIFFERENT APPROACH FOR STRING MATCHING

There are two different approach of string matching or string searching

1. In one approach, given a query string S & text T, the text T is scanned to search the query string S and try to find a place where the query string is found within a larger text. There are large no. of algorithms existing to solve string matching problem and can be applied on the basis of requirement of exact or approximate string matching or single pattern and multiple pattern matching.

1.1 EXACT STRING MATCHING

In case of exact string matching Brute Force Algorithm, Searching with automation, Rabin Karp algorithm, Knuth-Morris Pratt Algorithm etc., search the position in T from where the query string S is exactly found.

1.2 APPROXIMATE STRING MATCHING

In this approach, the algorithms are designed to find all the occurrences of query string in the text whose edit distance to the query string is at most K. The edit distance between two strings is defined as minimum no of character modification need to make them equal. Dynamic programming is a method to solve approximate string matching problem.

In this approach when the string is searched through the whole text for solution, it is fast and efficient only in case of short text. There is very little to improve upon the query time as loading up the whole text into memory takes up the maximum chunk of processing time. With the increasing size of textual electronic data, brute force scanning is not viable approach for string searching. There is hence a requirement to build some kind of indexes over these massive textual data to effectively process string queries of arbitrary lengths.

2. The second approach of string matching or string searching, the attention is given on creating an index for effective searching the string T. For this purpose the text document is first preprocessed. The preprocessor normalizes the document. It removes all the delimiters (spaces, commas), replaces non-alphanumeric symbols and unifies upper and lower-case characters. It then returns the list of single terms that occur in the document. The set of all distinct words in the index is termed as vocabulary. These terms in vocabulary are represented in data structure in memory. This data structure will then provide indexing into the text so that the string search and comparison can be performed more efficiently.

According to The American Heritage Dictionary of English Language , 4th edition index is defined as follows: 1. Something that serve to guide, point out, or otherwise facilitate reference especially : a. An alphabetized list of names, places and subjects treated in a printed work, giving the page or pages on which each item is mentioned b. A thumb index. c. Any table, file or catalog. d. Computer science A list of keywords associated with a record or document used especially as an aid in searching for information.

The index is of course one of the tools needed to fully solve the user queries as the retrieval of query string location is the very first step in query processing. Other steps involved in query processing are information retrieval (IR), ranking algorithms, user feedback model etc and are beyond the scope of this survey. Here after we will concentrate on challenging problem related with the design of efficient and effective index data structure which a basic block for many textual data based applications.

There are many criteria for a good index design which includes over all speed, disk and memory space requirements, CPU time, ease of index construction , index maintenance in case of dynamic environment, scalability etc. In reality no indexing scheme is best at all the above mentioned criteria but different class of queries can be solved by different indexing schemes, managing different type of data and hence suitable for different type of applications. As a result no single data structure can be considered the best indexing scheme as each one is having its own advantages and disadvantages and we must know the details to make a right choice while implementing an IR system or search engine.

Fundamental structures such as trees and hash tables are used for managing distinct strings or words in vast range of applications but making a right choice of structures is crucial for the efficiency of searching the string, memory requirement, time required to insert and search the string keys and a choice of maintaining the strings sorted order.

In this paper we propose to evaluate the existing data structures for string matching and searching problem.

EVALUATION OF DIFFERENT DATA STRUCTURES FOR STRING MANAGEMENT

There is a range of data structure which can be used for managing string for task like vocabulary accumulation for the text documents. The performance of different data structures depends on various features of input text documents out of which the most important is the skew which is defined the phenomenon of common words dominating as a proportion of word occurrences. The other characteristics on which the performance depends is locality: the degree to which a recently -observed word is likely to occur again, independent of its overall frequency. There are characteristics like order of string, no. of distinct string in a document also important to consider while evaluating the candidate data structure for string management. Hence it is important to understand the structural properties of various data structures along with their performance in respect to time and space and their suitability for many common string processing application like vocabulary accumulation or index creation.

LINKED LIST AND ARRAYS

The simplest dynamic data structures is the linked list, developed by Newel, Shaw and Simon in 1955 and they used this data structures as a primary data structures in their Information processing language [Newell and Tonge , 1960] . Linked list consists of a chain of nodes where each node stores data item (ex-string) along with a pointer to the next node in the chain. To store a sting we traverse the list by comparing strings of each node to see if it matches and if match is not found , the string is stored in a node and attached to the head or tail or in sorted order in the linked list. Linked list requires no average and at worst O(N) comparison per search where N is the total number of strings in the list. The linked list became very popular in computing environment and was used to develop several programming languages such as COMIT and LISP as well as File structures for operating system. Linked list is however not suitable for applications like vocabulary accumulation or index creation because of lack in scalability. A scalable data structure is one that remains efficient to access as the number of data item stored increases [Nikolas Askitis 2005] Strings can also be stored using a fixed size or dynamic array , with the cost of O(N) per operation of access, insertion or deletion. The cost of resizing the dynamic array also need to be taken consideration into , which involve accessing every string which can be expensive for large N. Even for maintaining array in sorted order high cost is involved. For example for inserting a new string into an array , the existing strings must be moved to maintain sort order. As array is also not scalable like linked list , it is thus inefficient for computing applications like vocabulary accumulation.

SEARCH TREES

A standard binary search tree (BST) has nodes having three information, out of which one is the key and two pointers each for left child and right child. The key feature for BST is that all keys less than or equal to the node key are stored at left sub-tree and otherwise at the right sub-tree. A string key search starts at the root, if the string key is not found in a node, a comparison is done to find the relation between string key and node key and is determining the branch downwards the tree to follow. An unsuccessful search terminates at the leaf node and then depending on the requirement of the task the key string can be inserted in the BST.

The performance of BST depends on the order in which the sting keys are inserted. In worst case BST acts like a linear linked list and its time complexity is O(N). This happens when the input strings are in sorted order. If the input is not ordered or skewed, the performance of BST is very good but we cannot use BST as even if first few strings are in sorted order the performance gets down dramatically.

AVL trees [Cormen et.el] is a variant of BST which removes the drawback of BST by reorganizing the tree at the time of insertion or deletion to maintain a balanced tree. Red and Black tree [Cormen et,el. Knuth] is another variant of BST which maintains the balance tree with some additional information on every node . Both these variants are proven to have a logarithmic worst case performance by removing the chance of tree formation similar to linked list but due to the reorganization frequently used keys are not necessarily clustered near the root nodes. Additionally a cost of balancing the tree incur in case of insertion of skewed data.

Splay Trees are another adaptive variation of BST have an amortized cost of O(M log N) for a sequence of M sequence of insertion , deletion and searches [Mehta & Sahani, Bell & Gupta, Sleator & Tarjan]. A tree is modified at each access by rotating the access node through rotating with respect to parent node and grandparent node which replaces the access node with its grandparent [Mehta & Sahani]. Splaying leads to clustering of commonly accessed node near the root , an advantage for skew input. Splaying can be done recursively either bottom up or top down. Bottom up splaying is found to be more efficient however there is high cost involved in rotation which adds up for an expected low cost search [Heinz & Zobel].

HASH TABLES

Hash tables are faster than any tree structure, buts it's performance comes with a price. The search can become sequential if the data set is large and the hash table is comparatively small. This leads to longer search time. In other way if the hash table is proportionately larger then there is a problem of wastage of memory space. Standard methods of hashing strings and maintaining a chained hash table are not fast , however , in other work it is been shown that with the use of bitwise hash function [Ramakrishna &

Zobel], chaining and move-to-front in chains hash tables are much faster . The disadvantage of hashing is that the strings are randomly distributed among slots, while this problem is not significant for bulk insertion or creation of a static index, it means that hashing is not a candidate structure for the application where the strings are required to be in sorted order. Based on the success of copying strings to array based buckets in string sorting [Sinha et,el., Akshit & Zobel] replaced the linked list of the chaining hash table for forming the cache conscious hash.

TRIES

A Tries is a multi -way tree data structure that stores sets of strings by successively partitioning them across letters of an alphabet [De la Briandais . E. Fredkin ,Jacquet & Szpankowski]. It was originally proposed by de la Briandais and was later called a trie by Fredkin. Tries have two properties which cannot be implemented on BST.

1. The strings are grouped by shared prefix.
2. There is an absence of string comparison which is an important feature for efficiently searching and matching strings.

Tries can be traversed speedily and offer a good worst time performance without an overhead cost of rotation or balancing [De la Briandais, Williams J. et,el]. Due to its appropriateness trie data structure is being used for a range of applications like dictionary management, pattern matching , natural language processing , IP routing etc,.

Although Trie is fast for string operations but it is space-intensive, which become a serious problem in practice [Knuth] and they are restricted to applications not involving large set of strings. There are various ways to reduce the space requirement of trie and hence make it applicable for larger volume of string sets also. Space requirement can be minimized by either changing the node structure in such a way that it requires less memory or no of nodes required can be reduced.

VARIANTS OF TRIES

The array trie is implemented in an array of pointers, one for each letter of alphabet [De la Briandais]. A string of length k is stored as a chain of k nodes. Storing strings in this manner form array trie that branches from a single root as shown in fig 1. The trie nodes are accessed on the basis of the lead character which is the index in the pointer array. The lead character is then removed and the next trie node is accessed or created in case of not found.

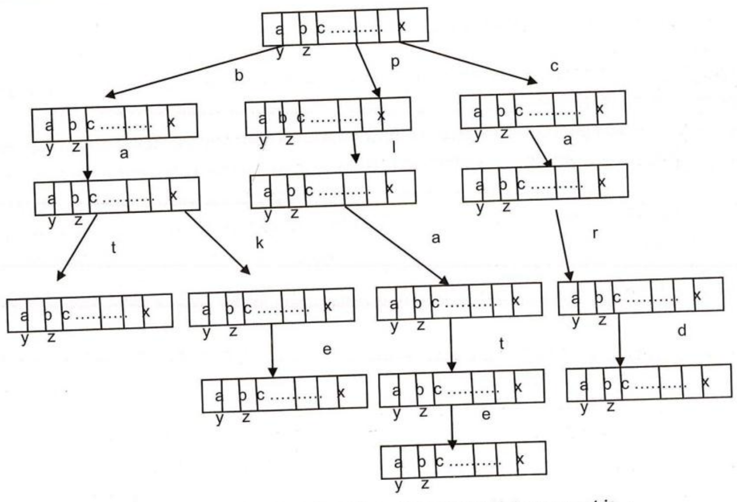

Figure 1. The Strings “bat”,”bake”, “plate”, and “card” are inserted into an array trie .

The search or insertion procedure terminates when the characters of string get exhausted, in this case the last node created will have its end-of-string flag set on searched or inserted. The access to these pointer array is fast but most of the pointers are likely to be unused which waste the space.

THE LIST TRIES

The tries proposed by [Mehta & Sahani] also called the list trie [Williams J. et,el] is the simplest approach for reducing the space requirement of the array trie, by changing the representation of nodes to the linked list that only stores the non null pointers. In contrast to the array trie where the space is allocated for n pointers per node where n is the size of alphabet set, the list trie creates as many nodes as needed. The list trie is similar to array trie except the representation of node comprising of three fields lead character, child pointer and sibling pointer. The sibling pointer points to the next character in the trie node and child pointer points to the next trie node. In figure 2 the horizontal line indicates the sibling pointer. Last node of the child list have end of string flag set to indicate that all characters of the strings are consumed.

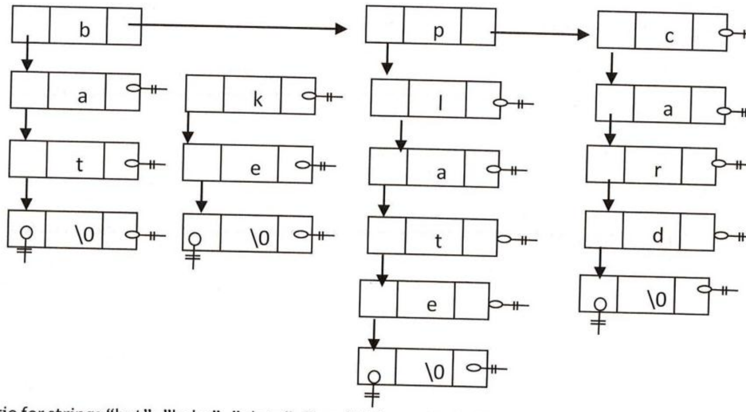

Figure 2. The list trie for strings "bat", "bake", "plate", "card". The nodes indicating the end of strings have both child and sibling pointer having null values.

METHODS FOR IMPROVING EFFICIENCY OF DATA STRUCTURES

1. There are different works done by researcher to improve the drawback of trie data structure and make it more efficient for string management. Many researcher have tried to conserve the space by reducing the number of required nodes and by changing their structure. The Compact trie [Bell et,el.] reduces the no of nodes by combining chain of all node leading to a single leaf by a single node. Patricia trie [Aoe et,el. Andersson & Nilsson] in which the nodes that have no branching are collapsed to form a single node for saving storage space. Ternary search tree [Bently & Sedgewick, Clement,] are fast but space intensive. Burst tries [Steffen Heinzet,el.] is a BST or other data structures that act as a container. Burst tries is currently one of the leading string management technique.

Following Table 1.0 shows the latest work done by different researchers in the area of improving the efficiency of in-memory string operations :

Table 1.0

S. No.	Author	Year of Publication	Implementation	Remarks
1	Steffen HeinzS, Justin Zobel & hugh E. Williams	ACM Transaction on Information System 2002	Burst Trie with three components : Records , & Access trie Array Mapped Trie is used to implement Access Trie and BST is used to implement Container.	Burst Trie is highly efficient for managing in-memory large string sets . The performance depends on the distribution of the strings.
2	Nikolas Askitis & Ranjan Sinha	AUSTRALASIAN COMPUTER SCIENCE CONFERENCE 2007	HAT Trie: where the container of Burst trie is replaced with cache concious hash tables	HAT-trie was 80% faster than Burst trie with a space reduction of 70%. Chaining hash table is proved better only in case of given extreme space.
3	Sebastian Kniesburges and Christian Scheideler	5th International Workshop, WALCOM 2011.	Hashed Patricia Trie: to find longest prefix matching applied over distributed hashing table	Combined the advantages of patricia trie and Hash table and proved as a efficient algorithm for the application.
4	Simon M. C. Yuen, Fu-Lai Chung, Robert Wing Pong Luk.	Proceedings of the 2006 International Conference on Bioinformatics & Computational Biology, 2006.	Hashed Trie : for indexing genomic data bases where the speed of indexing and retrieval can be increased by utilizing the overlapping characteristic in	The proposed data structure is about 4-5 times faster than the Bitwise and Perfect Hashing.
5.	Ferenc Bondon	Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'2004). Volume 126 of CEUR Workshop Proceedings	Trie Based Frequent Item Set Mining: Used Trie data structure for mining pattern of two or more event happening together (association).	The author has explored various representation of tree Trie data structure and concluded that a carefully chosen combination of techniques can lead to 2 to 1000 fold decrease in run time without significantly increasing memory consumption.
6.	Aridj Mohamed, Zegour D. Edine	IJCSI , Vol 7, Issue 6, Nov. 2010	TH*:Scalable Distributed Trie Hashing . The hashing is done on the basis of the character and its position in the string in a client-server architecture.	Trie hashing is used on the file which is sorted on some key value. The whole file is divided into different buckets of records on different servers and clients can search or update the keys .
7	Packer Ali, Dhamin	ACM SIGCOMM 1997	Scalable High speed IP Routing Lookups: Trie implemented for IP routing based on hashing on length of string and buckets of different lengths.	Mapping bit length in patricia trie to separate hash tables and using binary search on hash tables to cut down the cost of searching.

Surprisingly in scientific literature, no better worst-case bound have been obtained for algorithms and data structures manipulating arbitrarily long strings in -memory. As far as string-matching data structures are concerned, suffix array, patricia trie and suffix tree are particularly effective in handling unbounded length string which are small enough to fit in memory. However they are no longer efficient when the text collection becomes large, changes over time and make considerable use of external memory. Due to the limitation of the speed of external memory and with latest development of main memory capacity and speed, most of the applications are using main memory to store all its required data. Most of the researchers have used this advantage of main memory over external memory and developed alternate solutions for better in-memory string management. The table 1.1 below shows the computational criteria of recent development in string operations:

S.No.	Author	Computational Criteria					Performance Comparison				
		RAM & Dynamic Data	RAM & Static Data	Cache Oblivious	Cache Concious	Disk Based	BST	HASH	ARRAY	Linked List	Other
1	HeinzS, Zobel & Williams (2002)	Y	Y	-	-	-	Y	X	Y	Y	Y
2	Askitis & Sinha (2007)			-	Y	-	Y	Y	Y	Y	Y
3	Sebastian & Christian Scheideler (2011)	Y	Y	-		-	-	-	-	-	-
4	Yuen, Chung, Robert (2006)	Y	Y	-	Y	-	Y	Y	Y	Y	Y
5	Ferenc Bondon	Y	Y								Y
6	Aridj Mohamed, Zegour D. Edine	Y				Y	-	-	-	-	
7	Packer Ali, Dhamin	Y	Y	-	-	-	-	-	-	-	-

Table 1.1

Different area of Application

Table 1-2 shows the different areas where the efficiency of improved data structures are tested

S.No.	Author	Information Retrieval	Dictionary Structure	Search Engine	String Key	Others (dist. env. / Freq. itemset mining,)	Common Prefix	Common Suffix	key based hashing	string length	Other (overlapping Char)	Binary	English Alphabet	other (eg. Geonomic data)
1	Steffen HeinzS, Justin Zobel & hugh E. Williams	Y	Y		Y			Y					Y	
2	Nikolas Askitis & Ranjan Sinha	Y	Y		Y			Y	Y				Y	
3	Sebastian Kniesburges and Christian Scheideler	Y			Y	Y	Y		Y			Y		
4	Simon M. C. Yuen, Fu-Lai Chung, Robert Wing Pong Luk.	Y			Y				Y		Y			Y
5	Ferenc Bondon					Y	Y						Y	
6	Aridj Mohamed, Zegour D. Edine	Y			Y	Y			Y				Y	
7	Packer Ali, Dhamin	-	-	-	Y					Y		Y		

CONCLUSION

As in the current information age string search is one of the most important operation in many applications, we need to improve on the existing indexing techniques for speeding up the string searching time . Based on the literature review the researcher found that many text based indexes were developed and various techniques like hash table and BST have given adequately good results for the small set of strings but have their own limitations when more strings are added. Besides the large no of strings in textual data , there is a problem caused by uneven distribution of words based on the first character of the string. For example in English language there are many more words starting from letter 'S' than words starting from 'X'. Existing indexing techniques build their search tree on the basis of first character or first two character of the word which require more time to search common words.

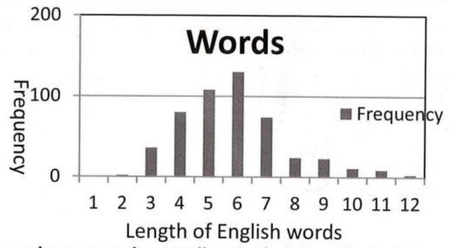

Fig 3 : The words arranged according to their length depicts parabolic shape

Along with this we observed that the total number of words are also related with the word length as depicted in the fig 3 which is drawn after taking 500 English words on the basis of simple random sampling.

Based on the above review the researcher concluded that the length of the string can be a criteria to segregate and minimize the search space for searching a string and hence propose to develop a hybrid data structure model for managing strings in main memory on the basis of first character of the string and the length of the string by utilizing the good features of different existing data structures which will be used to design efficient algorithms for insertion, deletion and searching of a strings.

REFERENCES

1. A. Andersson and S. Nilsson. *Improved behavior of tries by adaptive branching*. Information processing letters, 46(6), 295-300, 1993
2. Aridj Mohamed , Z.D. Edinne , *TH*: Scalable Distributed Trie Hashing* , IJCSI , Vol 7, Issue 6, Nov. 109-115 ,2010
3. Askitis , N. &Zobel, J.(2005), Cache Conscious Collision resolution for string hash table, in Proc. SPIRE string Processing and information retrieval Symp, Springer-Verlag,pp. 92-104
4. Askitis, Nikolas, and Ranjan Sinha.(2007) "HAT-trie: a cache-conscious trie-based data structure for strings." Proceedings of the thirtieth Australasian conference on Computer science-Volume 62. Australian Computer Society, Inc.,
5. Bell , T.C.,Cleary , J.G.& Witten, I.H.(1990),Text Compression , Prentice Hall
6. Bentley, J., & Sedgewick, R. (1998). Ternary tree, Dr. Dobb's.
7. D.D. Sleator and R.E.Tarjan . Self Adjusting binary Search tree . Jour of the ACM ,32,652-686, 1985
8. D.E.Knuth. The art of computer Programming vol. 3 : Sorting and Searching, Second Edition, Addison Wesley , Massachusetts, 1973
9. De la Briandais . File searching using variable length keys. In proc. Western Joint Computer Conference , pages 295-298, New york,
10. Dinesh P. Mehta and S. Sahani ,A hand book of Data structures and Applications , Chapman& Hall Computer and Information Science Series,Washington ,D.C. 221-236
11. E. Fredkin . Trie Memory , Communications of the ACM , 3(9):490-499,1960
12. Ferenc Bondon , A Trie Based Frequent Item Set Mining, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'2004). Volume 126
13. H.E.Williams J. Zobel and S. Heinz , Self -adjusting trees in practice for large text collections. Software-Practice and Experience , 31(10):925-939,2001
14. Jacquet and Szpankowski. Analysis of digital trie with markovian dependency. IEEE Transactions on information theory, 37(5):1470-1475, 1991

15. J.I. Aoe, K. Morimoto, M.Shishibori & K.H.Park. A trie Compaction algorithm for a large set of keys. IEEE transactins on knowledge and data Engineering , 8(3):476-491, 1996

16. J.Bell and G.K.Gupta , An evaluation of Self -adjusting binary search tree techniques, Software - Practice and Experience , 23(4):369-382,1993

17. J. Bently and R. Sedgewick. Fast algorithm for sorting and searching strings. In Proc. Annual ACM-SIAM Symp. On Discrete Algorithm, pages 360-369, ACM 1998 .

18. J. Clement, P.Flajolet and B. Vallee. Dynamic source of information theory: Ageneral analysis of trie structures. Algorithmica, 29(1/2): 307-369,2001

19. M.V.Ramakrishna and J. Zobel. Performance in practice of String hashing functions. In R. Topor and K.Tanka , editors , Proc. Int. Conf. on Database Systems for Advanced Applications, pages 215-223, Australlia , 1997

20. Newell and F.M. Tonge , An introduction to information processing language V. Communications of the ACM , 3(4); 205-211, 1960

21. Packer Ali, Dhamin, Scalable High speed IP Routing Lookups, ACM SIGCOMM 1997

22. S.Heinz and J.Zobel . Practical data structures for managing small set of strings. In M.J.Oudshoorn, editor, proc. Of the Australasion Computer Science Conf. , p. 75-84, Melbourne Victoria, Australia, Jan 2002 Sebastian Kniesburges.

23. Christian Scheideler Hashed Predecessor Patricia Trie - A Data Structure for Efficient Predecessor Queries in Peer-to-Peer Systems , Distributed Computing, Vol. 7611 (2012), pp. 435-436, Sebastian Kniesburges.

24. Christian Scheideler Hashed Patricia Trie: Efficient Longest Prefix Matching in Peer-to-Peer Systems , WALCOM: Algorithms and Computation , 5th International Workshop, WALCOM 2011, 2011. pp 170-181.

25. Sinha, R., Ring D. & Zobel, J. , Cache efficient string sorting using copying, ACM Jour. Of Exp. Algorithmics 11 (1.2), 2006.

26. Simon M.C. Yuen, Fu-Lai Chung, Robert Wing Pong Luk: Fast Dictionary lookup in Genomic information retrieval ,BIOCOMP 2006 pp. 251-257.

27. Steffen Heinz, Justin Zobel and Hugh E. Williams, Burst Tries: A fast, Efficient Data Structure for String Keys. ACM Transactions on Information Systems, 20(2):192-223,2002.

28. T.H. Cormen, C.E.Leirerson and R.L. Rivest, Introduction to algorithm . The MIT press, Massachusetts,1990.