A Novel Data Structure for String Matching Applicable in Network Processing

Nasser Yazdani and Hossein Mohammadi

Router Laboratory, School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran

Summary
We address prefix matching problems which constitute the building block of some applications in the computer realm and related area. It is assumed there are strings of an alphabet Σ which are ordered. The data strings can have different lengths and some of them can be prefixes of others. A well known application of prefix matching is layer 3 IP switching in which routers forward an IP packet by checking its destination address and finding the longest matching prefix from a database. In layer 4 switching, the source and destination addresses are used to classify packets for differentiated service and Quality of Services (QoS). We believe the fundamental issue preventing applying the usual tree structures such as B-tree to the prefix matching problem is the lack of a systematic method to compare and sort strings of different lengths. We introduce a simple scheme for comparing and sorting strings of different lengths first. Then, since the usual data structures can not be applied directly to the sorted strings, we manipulate data and tune the tree structures. We propose twp tree structures and devise all related procedures to build trees and process queries. A binary prefix tree is introduced and which can be extended to static and dynamic m_way prefix trees.

Keywords
Prefix tree, IP Lookup, Packet Classification

Introduction

Rapid growth of the Internet has faced researcher with some new challenging problems. This growth has affected our work, communication and social life very deeply. Every one wants to join this new and exciting world by creating his/her own website. Then, the number of hosts on the Internet is growing everyday and consequently, the data traffic is exploding. On the other hand, some new applications such as multimedia, hypertext data, video conferencing, remote imaging, etc., which are very data intensive contribute to this traffic explosion. All these demand for higher bandwidth on the communication line and fast and efficient methods for the traditional computer network problems.

To keep up with these waves of demands and increased traffic, the speed of the communication lines has been increased from 10 Mbps (Megabit per second) to gigabit per second. A new connection based technology, ATM, has been emerged. However, since the huge investment on the relatively old Ethernet technology is already on place, and certainly it is not going to leave the scene very soon, the gigabit Ethernet technology has been developed. Even though the gigabit Ethernet has borrowed some new ideas and technology from ATM, it is still faithful to its core idea and remains connectionless. This implies the routers which forward IP datagrams must determine the next destination for each data packets. To do this, the routers search the IP routing tables to find the address of the next hop to which the packet is going to be forwarded on the path towards the final destination. With current trend in the network technology finding the next hop for each datagram becomes harder and harder. Increasing number of hosts on the Internet expands the global network and number of hops in the Internet. Then, the size of the routing table grows everyday and requires faster access methods. Unfortunately, increasing the speed of data link worsen the situation since the time to send a datagram decreases with the links speed. Then, we cut in the middle of two factors which together require smaller search time in a bigger set. Advances in the semiconductor technology which improves the processing capability of new CPU chips can pay off in some degree. However, since the links speed grows faster than the processing speed and the size of data is growing on the other hand, it sounds the IP lookup problem can be a serious bottleneck.

In this paper, we prose a new indexing and searching scheme for the IP lookup problem. Unlike most of the previous proposed methods, our method is independent from IP address size or length and can be scaled to *IPv*6 protocols with 128 bit address without any extra memory space, except for the data elements, and search time. The proposed method is based on the binary search tree with the $Log_2 N$ search time and N memory space where N the number of data elements. Interestingly, our method does not consider any assumption about the distribution of length of data. Implementing the proposed method in software or hardware is easy and, indeed, straight forward. The rest of the paper has been organized as the following.

The rest of the paper organized as follows. In section two, we formally define the problem and explain the IP lookup problem which motivates us to accomplish this work. In section three, a formal method is given for comparing strings of different lengths and sorting them

Manuscript received July 25, 2006.
Manuscript revised July 30 , 2006

based on this scheme. We apply the binary search tree structure to the prefix matching problem in section 4. The most basic procedures to build the index tree and process queries are devised in this section. Section 5 reviews related works and tries to explain the base of the previous methods and concludes the paper.

2. Motivation and Problem Definition

Increasing the speed of communication lines from 10 Mbs to several gigabits per second has brought the IP lookup problem to the attention of researchers as a bottleneck in the Internet communication in the last years. Figure 1 illustrates the problem. Each IP packet contains its destination address. Routers must determine for each packet the address of the next hop to which the packet must be forwarded. Routers do this by checking the destination address and finding the longest matching *prefix* in its database. Table 1 shows an example of this database. The database consists of IP address prefixes and their corresponding hops. For example, assuming the destination address of the packet is 1011000110 00 , the packet is sent to hop 10 since the prefix 10110001* is the longest matching prefix with the packet destination address. This problem is more crucial now due to the rapid growth of the Internet traffics. The number of hosts on the Internet is growing and the data traffic is exploding. Routers must find the longest matching prefix in a larger data set, due to increase in number of LANs, in a smaller amount of time. For instance, assuming the IP routing database has one hundred thousand prefixes and the link speed is 2.5 Gbp, routers must be able to find the longest matching prefix in 200 nanoseconds.

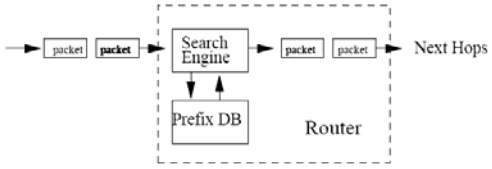

Figure 1: IP Lookup in Routers

This problem motivates us in proposing efficient Prefix Trees data structures. Prefix Trees can be used in applications which involve matching strings of different lengths. In devising Prefix Trees we assume there are strings of an alphabet Σ which are ordered. The strings are not necessarily of the same lengths. In particular, besides the exact match queries; we are also interested in the following queries.

- To find the longest string which is a prefix of a given query string.
- To find the smallest prefix of a given query string.
- To list all the strings which are prefixes of a given query string.
- To find all the strings such that a given query string is a prefix of them.

The items 1 and 3 are important in IP routing lookup and packet classification in the TCP/IP protocols, and indeed, they are the core of layers 3 and 4 switching.

In the following, we propose two tree structures for the prefix matching problem. First, a binary search tree is devised which uses $O(N)$ memory. This data structure is efficient in memory usage if the search time is not a bottleneck. We extend the binary search data structure to *m_way* tree and propose a scheme for static data sets.

Table 1: A sample Database of Prefixes for IP lookup problem

Prefix	***Next hop***
10*	7
01*	5
110*	3
1011*	5
0001*	0
01011*	7
00010*	1
001100*	2
1011001*	3
1011010*	5
0100110*	6
01001100*	4
10110011*	8
10110001*	10
01011001*	9

3. Background and basic issues

The most common data structure devised for the string matching problem is *trie* which is based on the "thumb-index" scheme on a large dictionary [15]. A trie is essentially an *m_way tree*. Each internal node of a trie has m branches, each branch corresponds to a character in the alphabet. Each data string in a trie is represented by a leaf and its value corresponds to the path from the root to the leaf. Figure 2 shows an example for the strings of table 1 where m is 2 and the alphabet is only $\{0,1\}$. The blank internal nodes are considered as place holders since they do not represent any data element. We have relaxed the condition of representing of each data element by a leaf node since some data elements are prefixes of others. A nice property of this data structure is that it is so easy to list all prefixes of a given string. We can start from the root and follow the branches corresponding to the characters in the query string to leaf at each internal node. black nodes in Figure 2, is a prefix in the path from the root to the end leaf. While giving a good search time to find prefixes of a query

string, the following shortcomings can be identified in the trie structure of Figure 2.

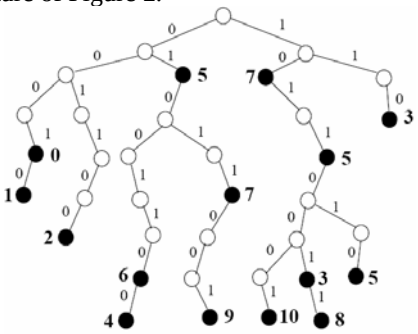

Figure 2: A trie structure for the prefixes in table 1. Bold numbers represent next hops.

- The blank nodes, i.e., the place holders, do not correspond to any data element in the data set. They consume memory and add to the hight of the trie and, consequently, prolong the search process.
- The search time corresponds to the length of data elements. For instance, for the IP lookup problem, in the worst case, this can be $O(32)$ in IPv4 and $O(128)$ in IPv6. Therefore, even a small set of data may require a long search time.
- The number of branches corresponds to the number of characters in the alphabet. This makes the data structure inflexible.

Different solutions have been proposed to overcome these shortcomings. Knuth in [15] proposed to continue branching at the first characters as long as each data element can be uniquely identified. This compresses the last part of the trie and can save time and space with little extra work. He also shows the average search time for large N is only $O(log_M N)$ for random data where N is the number of data elements and M is the number of characters in the alphabet. Random data here means the data elements are uniformly distributed. The total space required for building the trie is proportional to MN/ln M. Patricia tree [13] compresses the total path by eliminating each internal node with only one child and, consequently, skipping some characters and increasing the node utilization. Some recently proposed methods, [21], [20] and [17], try to check several characters instead of one in order to reduce the hight of the tree while minimizes the memory usage. Nevertheless, none of the proposed methods completely eliminates the redundant space. This is the problem we are going to tackle. As previously explained, we are going to apply regular tree structures to the prefix matching problem. Before discussing any of these data structures, we need to address the sorting strings of different lengths" which is fundamental to the proposed methods. We have employed the idea of m_way prefix tree for developing software-based IP lookup methods in [4],[5] and Hardware-Assisted methods in [6], [7] and network processor architectures in [8],[9].

4. Sorting strings of different lengths

Why can't we apply the well known tree structures like the binary search tree to the prefix matching problem? Why are there blank nodes in the trie of Figure 2? The answer is that there is no well known method to sort strings of different lengths, specially, when the strings are prefixes of each others. We can apply the binary search tree to the numbers and texts since they can be sorted. Indeed, sorting acts like a function which gives the relative position of each data element in the sorted space. Then, the sorted space can be divided so that in each search a limited number of data elements are compared with the query datum. Therefore, we must find a sort function for strings of different length which takes any string and find its position with respect to others. The position of each string must be unique and the sort function must not map two different data elements to the same position. Before defining the sorting function, it is worth noting that the characters in the alphabet are assumed to be ordered. This is not a limitation to our method since any alphabet can be sorted at the machine level. With this assumption in mind and assuming the fact that strings can be prefixes of others, in the following, we define a simple method for comparing two strings of different lengths. Regarding this definition, we define a sort mechanism.

Definition 1: Assume there are two strings $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$ where a_i and b_j are characters of alphabet Σ Also assume there is a character $\perp$ which belongs to Σ. Then,

- If $n = m$, two strings have the same length, the values of A and B are compared to each other based on the order of characters in Σ.
- if $n \neq m$ (assume $n < m$), then, the two substrings $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_n$ are compared to each other. The substring with bigger (smaller) value is considered bigger (smaller) if two substring are not equal. If $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_n$ are equal, then, the $(n+1)$ th character of string B is checked. We consider $B \leq A$ if b_{n+1} th is equal or before $\perp$ in the ordering of characters in Σ, and $B > A$ otherwise.

The $\perp$ character should be chosen in such a way that the probability of any character(s) in the lower order or upper order of $\perp$ be is roughly equal. For instance, in the English alphabet, assuming the probability of a character to be in the range $A - M$ or $N - Z$ in a text to be roughly %50 , M can be considered as $\perp$. Then, BOAT is

smaller than GOAT and SAD is bigger BALLOON. CAT is considered bigger than CATEGORY since the fourth character in CATEGORY, E, is smaller than M. Or in the binary alphabet, $\{0,1\}$, assuming $\perp$ is 0, clearly, 1101 is greater than 1011 and smaller than 11101, and 1011 is greater than 101101. Therefore, definition **1** gives us the necessary tool to compare and determine the relative standing of all strings respect to each others. This may not be mathematically or in the usual text matching context correct, but it is a convention to determine the relative position of any query string. It is straight forward to show that based on definition **1**, the position of each string can be determined uniquely and it is impossible for two strings to be mapped to the same position unless they are identical. By applying definition **1**, the prefixes in table 1 can be sorted in the ascending order as the following:

$00010*, 0001*, 001100*, 01001100*, 0100110*, 01011001*, 01011*, 01*,$

$10*, 10110001*, 1011001*, 10110011*, 1011010*, 1011*, 110*$

This is the key idea in proposing our methods. As long as we know how to sort strings, all well known index structures can be applied to the string matching problem. Before explaining any specific method or data structure, it is better to define our interpretation of *matching* two strings of different lengths precisely.

Definition 2 Assume there are two strings $A = a_1a_2\ldots a_n$ and $B = b_1b_2\ldots b_m$ where a_i and b_j are character of alphabet Σ. Then, A and B are matching if $n = m$ and two strings are identical, or (assuming $m > n$), two $a_1a_2\ldots a_n$ and $b_1b_2\ldots b_n$ substrings are the same. Otherwise, A and B are not matching.

4. Binary Search Tree

Regarding definition **1**, applying the binary search tree data structure to the string prefix matching problem seems straight forward. However, this is not the case and it requires tackling more subtle issues. Figure 3 shows the result of applying the binary search tree to the strings (prefixes) of table 1. This data structure works well to find the longest matching prefix of string $1011000110\ 00$. The thick lines there show the search path for this string. The search is the same as in any binary search tree, starting from the root, comparing the query string with its value and following a subtree based on the comparison result. The search must be followed to the leaves since we are looking for longest matching prefix. This data structure is superior to the trie structure of figure 2 if it can be proved to work correctly.

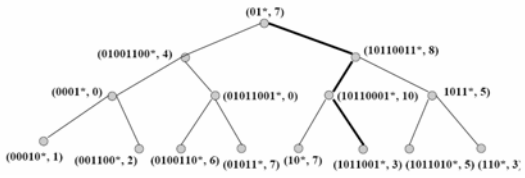

Figure 3: A binary balance tree for the prefixes in table 1.

The search fails to find the longest matching prefix of string $1011000010\ 00$ even though there are two matching prefixes 10 and 1011. Why? The reason is that the prefixes are *ranges* and not just a data point in the search space. For instance, string 10 includes all of the strings starting with 10 in table 1. However, they have been treated as the points in sorting and building the index tree. We prove in the following that this method works if none of the data element is a prefix from another. Before proceeding further, it is important to define two additional concepts which are frequently encountered in the rest of this paper.

Definition 3 Two strings A and B are disjoint if they are not a prefix (or substring) of the other.

Definition 4 A string S is called an enclosure if there exists at least one data string such that S is a prefix of that string.

As an example, *BAT* and *PHONE* are disjoint, but *DATE* is an enclosure of *DATED* and *DATELINE*. As another example, 1011 is an enclosure in the data set of table 1. We call these elements enclosures since they include other data strings in their spaces. An enclosure represents its data space as a point in the data set. For instance, all data strings in table 1 which are included in the range of 1011, such as 1011001, 1011010, and 1011 itself are considered as a point represented by 1011. A data element may be included in an enclosure or be disjoint with all other elements.

Lemma 5 Assume there is a set of strings which are disjoint. Then, a binary tree built based on definition **1** can correctly find the matching prefix(es) of any query string.

Proof: First, we claim if there is a matching prefix with a given query string, it must be unique. Assume there are more than one matching prefix. Then, since the prefixes of a string must match each other the shorter string will be enclosure of others and this contradicts our assumption that all data elements are disjoint. Next, we claim that the matching prefix and the query string will map to the same place in the index tree and will follow the same path. However, this is clear from the fact that the query and the prefix strings are disjoint with the rest of data set. Therefore, the query string will follow exactly the same path as the matching prefix when it is added (or searched) into the index tree.

We modify the binary search tree in order to handle a data set of strings with enclosures. What is really done in building a binary tree is dividing the data space into half recursively. We assume each subtree in the binary tree as a data space represented by the element in the root of the subtree. Each enclosure is also considered as a point which includes all matching strings in its data space. Then, the data strings are sorted and the binary tree is built as usual by recursively splitting the data space into two at each step. If the split point are just a single string the building process is followed as usual. However, when the split point is an enclosure, all included data strings are distributed in the right and left subtree regarding definition **1** and the property of the binary search tree. It is important to emphasize that an enclosure may be chosen as split point, or root, even though there are some other disjoint data elements. However, this does not change anything since the included data elements are distributed in subtrees with other disjoint data. In other words, the subtree rooted in an enclosure is not *exclusive*. Nevertheless, the data elements included in an enclosure space are always guaranteed to be in the subtree rooted by the enclosure, but not all data in the subtree have to be in the enclosure space. The building process is applied to the subspaces recursively.

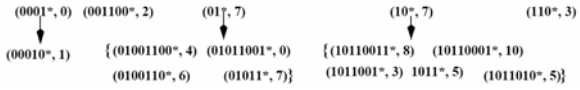

Figure 4: The first step sorting for the prefixes in table 1

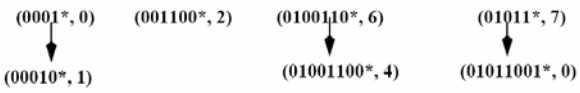

Figure 5: Sorting elements in the left subspace of figure 4.

Before going to the formal definition of procedures to build the binary search tree for the string prefix matching problem, we first apply it to the data set of Table 1. Figure **5** shows the result of sorting in the first step. There are only five disjoint elements, 0001,0001100,01,10 and 110 , and 01 is the median and taken as the split point. Since 01 is an enclosure, all elements contained in it are distributed in the two subspace and put on the left subspace since all of them are smaller. Figure 5 illustrates applying procedure to the left subspace and Figure 6. shows the final binary tree. As the reader may notice, the tree is not balance and search, in the worst case, takes one step more compare to the binary tree of Figure 3. However, this guarantees to not miss any matching prefix as we will prove it formally.

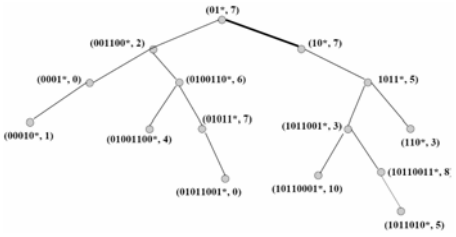

Figure 6: Our proposed binary tree for the elements in table 1.

In the following, we give formal procedures for building the index structure. First, we start with sorting strings based on our definition of the string comparison and enclosures.

4.1. Sorting strings with enclosures

We propose two procedures for sorting a set of strings with different lengths which potentially can contain enclosures. Both procedures assume each enclosure has a *bag* in which all elements contained in its space are put. Clearly, the sorting procedure is based on definition **1**.

The worst case running time of this procedure is $O(N^2)$ where N is the the number of elements in the list. This is when all data elements are disjoint and $\sum_{i=1}^{N-1} i$ comparisons are needed. The best case running time is $O(N)$ when the first string(s) is an enclsoure of others and consequently, the rest of data are put in its bag. This procedure does two things. It sorts the data strings, while filling enclosures with thier data elements. We call this process *enclosurizing* and formally defining all related procedures to build the *Perfix Binary Tree*. It is important to remind that the enclosurizing process works in just one level, while the string set can have different levels like shells of an onion. Therefore, we need to apply the enclosurizing process recursively in order to build the prefix tree as it will be discussed later.

Even though we do not have a precise mathematical analysis of average runnning time of the above sort procedure, it is expected to be slow since it uses the idea of bubble sort. Therefore, it is better to develop another sort algorithm which uses the idea of quick sort [10]. Even though the worst and best case running time of this procedure, Sort procedure 2, is the same as sort procedure 1, the average case seems to be faster. The main idea in Sort procedure 2 is to divide the data space into three, instead of two, if the split point is an enclosure. In this way, smaller data strings are put in the left side of the partition point, the larger data strings in the right side. Finally, the matching elements, the strings which are in the space of the split string, are put on its bag. If the split point is disjoint with the rest of data, the regular quick sort method is used. One qustion still remains. How do we identify the split element? We suggest to use the element with the minimum length at

each step . It will be proven later that this idea works. The intuition behind using the element with minimum length is to force enclsoures to be split points. Therefore, at each step more elements can be excluded from the sort process which makes the sort process faster. Furthermore, this can guarantee the final result is *enclosurized*; all data elements in the space of enclosures are put in their bags and excluded from the data list. The *MinLength* function in the following sort procedure gets a list of strings and returns the one with the minimum length.

The last line in the algorithm concatenates the results of sorts from the left to the right subspaces and put it in the $List$. It is obvious the worst case running time of this algorithm is $O(N^2)$ since its worst case is the worst case of quick sort [10]. However, it best case is different and is $O(N)$. This is due to the fact that the first element can be a enclosure of the rest and the list can be processed once. The average running time is the average running time of quick sort implying $O(N \lg N)$. The algorithm spends more time to process at each step, $O(2N)$, to find the element with minimum length and compare elements with the split point. However, the number of recurisve calls are smaller since some data element are eliminated at each step. The worst case of recursive calls is the quick sort recursive calls when all elements are disjiont. In this case, the average running time is longer, i.e., $O(N \lg N)$.

Theorem 6 Sort procedure 2 correctly sort and enclosurize a data set of strings of different lengths.

Proof: Correctly sorting data strings is obvious from the well known quick sort algorithm. In each step, the process tries to find the most probable enclosure point, the minimal length element. Then, it partitions the data space based on the split point and puts matching elements in the bag. Therefore, at the end, by recusively applying algorithm the final result is sorted. Now, we must show the data set are enclosurized, i.e, all elements in the spaces of enclosures are put in the corresponding bags and are out of the list. Assume this is not the case and there is a string, A, which is in the space of an enclsoure C and has not been placed in C's bag. This can only be possible if A and C fall in different subspaces in the partitioning data set. However, this is impossible because at each step we split the data space with respect to the string with the minimal length. Therefore, A and C both have to be in the same subspace with respect to the split point. There are two cases: 1) A and C match the split element then, they will be placed in the bag of the split point. (2) they are disjoint. In either case, A and C stay in the same partition until the space is partitioned by C and A is put in its bag. It is worth noting that the key point here is spliting based on the minimal length string. Otherwise, it can not be guaranteed that A and C will stay in the same partition.

4.2. Building tree

The prefix binary search tree can be built as usual after sorting the data strings. The building process is the same as any binary search tree. Any one of the discussed sorting procedures can be used to sort the data srings first. There is a subtle difference between the prefix and the usual binary search tree. In the ordinary binary search tree the data elements are sorted once in the begining and they remian sorted until the end. However, in the prefix tree, the strings in the enclosures' bags are not sorted. Furthemore, the sort procedures enclosurize data elements only one level whereas some strings in the bags may be enclsoures of others. Therefore, we need to apply the sort process recursively to the subsapces. In the following, we use a general Sort procedure to formaly define the *BuildTree* algorithm which takes a set of strings as the input and returns a pointer to the root of the index structure. The user can choose any of the sort algorithms discussed above. However, the algorithm based on the quick sort should be faster. The BuildTree procedure is given in a recursive format again, but implementing it in a procedural form is straightforward.

Building Tree Procedure:

```
BuildTree(List)
    if List is empty, return.
    Sort(List);
    let m be the median of List
    root ← m;
    let leftList and rightList contain
     all elements in the left and right of m.
    if m is an enclosure, then,
      distribute elements in m's bag
      into leftList and rightList.
    leftChild(root) ← BuildTree(leftList);
    rightChild(root) ← BuildTree(rightList);
    return address of root.
end BuildTree;
```

The most time cosuming part of the *BuildTree* procedure is *Sort*, specially in the first call. Using the second sort algorithm with average running time proportional to $O(n \lg n)$, the average running time of *BuildTree* can be approximated by the following recurrense.

$$T(n) = 2T(n/2) + n \lg n$$

Solving this recurrence yields the average running time of the procedure as $O(n \lg^2 n)$ [10]. However, if the data strings are sorted ones, in the next calls, the procedure can be made faster. This can be done due to the fact that in the next recursive call of *BuildTree*, the data strings are sorted except when the partition point is an enclosure. The situation can be improved by keeping the smaller and

bigger strings in bags separated. This does not take extra time and it can be done in the partitioning time. The second improvement can be achieved by sorting the smaller and bigger strings in the split point's bag and concatenating them with the rest. Since the number of elements in the bag are small compare to the size of data strings, the time saved in this way can be considerable. The following lemma gives an appropriate base for revising *BuildTree*.

Lemma 7 Assume A is a string and B an enclosure such that A is not in the space of B. Then, all elements contained in B space are in the same sorting rank respect to A as B. In other words, all elements enclosed in B are bigger than A if $B > A$, and smaller otherwise.

Proof: Let $A = a_1a_2 \cdot a_k$ and $B = b_1b_2 \cdot b_l$ and $A < B$. Then, if $k \le l$ implies $a_1a_2 \cdot a_k$ is smaller than $b_1b_2 \cdot b_k$ according to definition **1**. However, the length of any included string C in B is bigger than l and contains $b_1b_2 \cdot b_l$ prefix at the beginning. This directly implies that $C > A$. If $k > l$ then $a_1a_2 \cdot a_l$ is smaller than $b_1b_2 \cdot b_l$ by the assumption. Let C, $c_1c_2 \cdot c_m$, be a string in B, then, the first l characters of C are the same B. Since $b_1b_2 \cdot b_l$ is bigger than $a_1a_2 \cdot a_l$, extending both strings by any character in Σ will keep the same sort order based on definition **1**.

The *BuildTree* procedure can be modified based on this lemma as follows. It is assumed the strings in $List$ are already sorted by a *Sort* procedure.

Building Tree Procedure 2

```
BuildTree(List)
    if List is empty, return.
    let m be the median of List
    root ← m;
    let leftList and rightList contain all
    elements in the left and right of m.
    if m is an enclosure, then,
      leftList ← leftList & Sort(leftBag);
      rightList ← rightList & Sort(rightBag);
    leftChild(root) ← BuildTree(leftList);
    rightChild(root) ← BuildTree(rightList);
    return address of root.
end BuildTree;
```

leftBag and *rightBag* contain strings which are smaller and bigger respectively than the enclsoure in its bag. Symbol & represents concatenation of two strings. The running time of the algorithm depends on the input data. In the best case, strings are disjoint and the algorithm does not need to call *Sort* function. It is easy to show the running time is $O(N)$ and overall running time of *BuildTree* is proportional to $Sort$ or $O(N \lg N)$. Since it is not expected the *leftBag* and *rightBag* lists contain many strings compare to the size of the whole data, the running time of *BuildTree* is not expected to exceed $O(N \lg N)$.

4.3. Query processing

Query processing in the proposed binary search tree is straightforward. The tasks of queries 1, 2 and 3, finding the longest, smallest and all prefixes of a given query strings, from the motivation and problem definition section are almost identical. Therefore, we only give a formal algorithm for the first one, i.e., the longest matching prefix problem. Next, we propose a formal procedure that finds all strings for which a given query string is a prefix. The search process for the longest matching prefix is simple and almost the same as the search in binary search tree. The procedure returns the longest matching prefix if there is any and NULL otherwise.

The longest prefix search procedure

```
/* tree is a pointer to the root of index
tree and str is the query string.*/
Search(tree, str)
    if tree = NIL, return NULL;
    if (str < tree(root)) then;
      prefix ← Search(leftChild(tree), str).
    else
       prefix ← Search(rightChild(tree), str).
    if str matches tree(root) and prefix is NULL, then,
        prefix ← tree(root).
    return prefix;
end Search;
```

In the IP lookup problem str is a packet IP address and the data elements in the tree nodes are network prefixes. It is worth noting that the Search procedure always substitues the matching prefix in the upper level with the matching prefix in the lower level. This is a nice property which prevents from comparing the lengths of the maching prefixes and back tracking mechanism.

Theorem 8 Assume there is a binary search tree of strings built by the *BuildTree* procedure, then, the above search algorithm will find the longest matching prefixes with a given query string if there is any.

Proof: First we must show the search processs will find the answer if there is any. Then, we need to show this is the longest matching prefix. If all elements in the tree are disjoint, then the tree is a normal binary tree and according to lemm 1 the search will find the prefix which is the longest. This is true when there is only one matching prefix which is disjoint with the rest of strings in the index tree. Assume there are more than one matching prefixes in the data set. The smallest one will enclose others in its data space and the second one the rest and so on, like the shells of an onion. According to lemma **7**, the query string will map to the space of the smallest matching prefix first. However, since this prefix is the split point and is the root of subtree, the search process will visit it. This is also true for

all other matching prefixes. Let us see what will happen to the last or the longest prefix. There are two cases. First, the longest prefix is disjoint with the rest at the most interior space or at the most bottom subtree. In this case, the search will find it according to lemma 1. Secondly, the longest matching prefix itself encloses some other prefixes which do not match with the query string. In this case, Search will visit the root and will find the longest matching prefix.

As explained previously, finding the smallest matching prefixes is the same except the algorithm must quit when it find the first matching prefix. The procedure can find all matching prefixes by reporting each of them instead of substituing them with the longer one at each step.

4.4. Insertion

The insertion process is the same as any usual binary search tree when the data set are disjiont or the new string is not an enclosure of any string which is already in the index tree. The process starts from the root and follows the search path and adds it as a child of the last node in the search path. This takes $O(h)$ where h is the height of the tree. The formal routine for this process can be found in many data strucutre and algorithm book, e.g. [10]. When the string to be added is an enclosure of any string in the index tree, we will encounter a problem. The problem raises from the fact that in the proposed tree an enclosure must be in a higher level than the strings contained in its data space. This property is a must in order to guarantee that we will not miss any matching string in the query processing. We propose two solutions for this issue and give our formal procedure for the insertion based on the proposed solution. Then, we formally prove the insertion procedure satisfies the prefix index tree property.

1. The first solution for adding an enclusore is to follow the search path and when it finds the first string which is contained in the new string, insert the new string in this place and make the contained element a child of the inserted node.
2. In the second solution, the search path is followed until we find the first contained node. Then, the contained string is replaced with the prefix. Next, the replaced string is reinserted in the index tree. By replacing a node with a new element, the data in the sbutree may not remain sorted. Therefore, we need to sort the subtree by moving its data elements around the new inserted string.

The first solution is easy and fast, however, it increases the height of the tree and prolongs the overall search time. The second solution takes more time since we need to move some data element, which can be time cosuming, but it may give a smaller height and better search time. We give our formal insertion procedure according to the second solution. The insertion procedure based on the first method is simple and strightforward from the procedure given here.

Insertion Procedure

```
/* tree is a pointer to the root of the index tree
and str is a query string.*/
Insertion(tree, str)
    if tree = NIL, then,
      node ← AllocateNode();
      node ← str;
      make tree parent of node;
      return;
    if str is an enclosure of tree(root) then;
      replace tree(root) with str;
      Insertion(tree, tree(root);
      if str < tree(root), then;
        Move(leftChild(tree), str);
      else
         Move(rightChild(tree), str);
      return;
    if str < tree(root), then;
      Insertion(leftChild(tree), str).
    else
       Insertion(rightChild(tree), str).
end Insertion;
```

The *AllocateNode* function allocates a new node and *Move* recursively moves all data elments in the subtree compare to the given query string. It is important to note that reinsertion of the replaced element does not need to start from the root of the index tree and it can be inserted in the subtree rooted in the replaced node. The following theorem proves the given insertion procedure work correctly.

Theorem 9 The Insertion procedure given above statisfies the requirements of the proposed prefix binary search tree.

Proof: If the inserted string is not an enclosure of any node, it will be added to the bottom of the tree and the procedure will terminate at the first "if" statement. We must show if there is any enclosure in the index tree for the new added string, the inserted string will be added to the subtree of that enclosure. This is easy to concieve since based on lemma **7** the new string will map to the enclosure space in the sort process (or the search process). If the enclosure already contains some other prefixes in its data space, it is a split point and search path will go thourgh it. If the enclosure does not contain any data element, then, the search path will find it based on the sort definition. In any case, the new string will be inserted in the subtree rooted at the enclosure. Let us assume that a new string A which is going to be inserted to the index structure is an enclosure of some data elements in the index tree. We must show A will be added in a level higher than its enclosed strings and in the meanwhile, all enclosed data elements are in a subtree rooted in A. We start reasoning by looking at the insertion process which starts from the root by comparing

A with the data string in the root. There are three cases. First, two strings are disjiont. Then, A is either smaller or bigger. If it is bigger, the data elements in the left subtree are disjiont with A and can not be enclosed in it. Therefore, by following search in the right subtree, we do not violate any prefix tree property. The second case is when two strings match but A does not enclose the string in root. Again, the new string is either smaller or bigger. Let assume it is smaller. This means the character after the matching characters is smaller than $\perp$ according to definition 1. However, the character in the position after the matching characters of two strings, A and root, from all data strings in the right subtree are bigger than $\perp$ if they are enclosed in the root. Therefore, they are disjoint with A. If the strings in the right subtree are not enclosed in the root string they are disjoint with A and root. In any way, they cannot be enclosed in A. Therefore, the insertion process does not violate any property. The third case is when A is an enclosure of the string in the root. In this case, the root is replace by A, and the property of being enclosers in higher levels than their enclosed strings are kept in this way. Since this is applied recursively from the top to the bottom, the insertion process keeps the prefix binary tree proporty satisfied. The insertion process can stop at the first enclosed element since the shorter prefixes are in the upper level and if the new element encloses a shorter prefixe, it will enlcose the longer ones in its data space as well.

5. Related Work and Conclusion Remarks

[1] is a rich source for the pattern and string macthing problems and related algorithms. The general prefix string matching problem in which one is interested in finding the longest prefix of a pattern that starts at each position of a text string is addressed in the pattern matching litreture. This problem is a general case of the problems we address here and can be solved in a linear time by adopting the string matching algorithm of Knuth, Morris and Pratt [14]. [3] studies the exact complexity and tight comparison bounds of this problem. For matching data of different lengths, the reader can refer to [2] which is dealing with the problem of matching sequences of different lengths.

The main scheme for the prefix string matching data structure which is the base of other methods and intesively discussed in the litreture is *trie* [15]. A trie structure is based on the "thumb-index" on a large dictionary in which a word can be located by checking consecutive letters of a string from the begining to the end. From the tree point of view, a trie is essentially an *m_way* tree whereas a branch in each node corresponds to a letter or character of alphabet Σ. Any string is represented by a path from the root to the leaf corresponding to letters in the pattern. Figure 2 shows a trie in $\{0,1\}$ alphabet set. As discussed previously, the time and space complexity of the trie structure is well known and has been discussed in[15].

The IP lookup problem has been a hot research topic in the last few years and contributed to some new methods for the prefix matching problem in the $\{0,1\}$ alphabet. As expected, the base of most of these methods is the binary trie [18] or the *radix* tree [10]. The main problem with trie approach, in general, and the binary trie in particular is keeping some nodes which do not correspond to any data element. This wastes the space and prolong the search process. The worst case search time of this approach is $O(W)$ where W is the length of the longest string or IP address.

Patricia Trie modifies the binary trie by eliminating most of the unnecessary nodes [13]. The scheme has been implemented in the BSD kernel [19]. Patricia Trie is the base of several new methods which have been proposed in the last years. These approaches try to check several characters, or several bits, at each step, instead of checking only one character. Since checking several characters may deteriorate memory usage and leave many memory space unused [21], all these approaches try to minimize the memory waste. V. Srinivasan and G. Varghese in [20] proposed to expand the original prefixes (strings) into an equivalent set of prefixes with fewer lengths and then, apply a dynamic programming technique to the overall index structure in order to optimize memory usage. [17] proposed a specific case of [20] by locally optimizing memory usage in each step. Finally, a new scheme from Lulea University of Technology, [11], endeavors to reduce the size of data set (routing table) so that it fits in the cache. All these multibit trie schemes is designed for the IP lookup problem and work well with the existing size of data, number of prefixes in the lookup table, and IP address length which is 32 bit currently. Nevertheless, we believe they will not scale well for the larger size of data or longer string, for instance, the next generation of IP (Ipv6) with 128 bit address.

Works in the longest matching prefix string in the IP lookup context go on. For instance, the DP-Trie, Dynamic Prefix Tries, [12], proposed by researcher from IBM, is another version of the binary tree data structures. The data structure tries to compact the Patricia Trie by keeping prefixes themselves and the index of bit position differing in the subtries of each node. [16] exploited almost the same idea by applying binary search tree scheme and extending it to a multiway tree by treating each prefix as a range and identifying each range with L (low) and H (high). Considering the fairly uniform distribution, the average search time on both of these methods should be $O(log_2 N)$ in binary tree and $O(log_m N)$ in case of m_way tree. The Prefix Trees data structures proposed in this paper is similar to these schemes in concept.

References

[1] Alberto Apostolico and Zvi Galil, "Pattern Matching Algorithms", Oxfor Univ. Press, 1997.
[2] Tolga Bozkaya, Nasser Yazdani and Z. Meral Ozsoyoglu, "Sequence Matching of different lengths", 6th Int. Conference on Information and Knowledge Management (CIKM'97), 1997.
[3] D. Breslauer, L. Colussi and L. Toniolo, "Tight Comparison Bounds for the String Prefix Matching Problem", Proceeding of Combinatorial Pattern Matching, 4th Symposium, 1993.
[4] Nasser Yazdani, Hossein Mohammadi, "IP Lookup in Software for Large Routing Tables Using DMP-Tree Data Structure" Proceeding of the 9th Asia Pacific Conference on Communications (APCC) 2003
[5] Nasser Yazdani, Hossein Mohammadi, "A Fast and Scalable IP Lookup Scheme: an Effort to Employ Powerful Data Structures", Submitted to the IEEE/ACM Transaction on Networking. 2006.
[6] Hossein Mohammadi, Nasser Yazdani, Behnam Robatmili, Mehrdad Nourani, "Hardware Assisted Software-based IP Lookup for Large Routing Tables", Proceeding of the 9th International Conference on Networks (ICON), 2003, Sydney, Austuralia
[7] Hossein Mohammadi, Nasser Yazdani, "Accelerating Computation Bounded IP Lookup Methods by Adding Simple Instructions", Lecture Notes in Computer Science (LNCS), Vol. 3262, pp. 473 - 482, 2004
[8] Hamid Reza Ghasemi, Hossein Mohammadi, Behnam Robatmili and Nasser Yazdani "Augmenting General Purpose Processors for Network Processing" , In proceeding of the Second IEEE International Conference on Field-Programmable Technology (FPT 2003), Tokyo, Japan.
[9] Hossein Mohammadi, Nasser Yazdani, "A Genetic-Driven Instruction Set for High Speed Network Processors" In proc. of IEEE International Conference on Computer Systems and Applications, pp. 1066- 1073, March 2006.
[10] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Riverst, "Introduction to Algorithms", The MIT press, 1990.
[11] Mikael Degermark, Andrej Brondnik, Svante Carlson and Stephen Pink, "Small Forwarding Tables for Fast Routing Lookups", Proceeding of SIGCOMM 1997.
[12] W.Doeringer, G. Karjoth and M. Nassehi, "Routing on Longest-Matching Prefixes.", IEEE/ACM Trans. Networking, vol. 4, no.1, pp. 86-97, Feb. 1996.
[13] G. H. Gonnet and R. A. Baeza-Yates, "Handbook of Algorithms and Data Structures.", Addison Wesley, 2th Edition, 1991.
[14] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast pattern Matching in Strings", SIAM J. Comput., Vol. 6, p. 322-350, 1977.
[15] Donald E. Knuth, "The Art of Programming", Third Volume, Sorting and Searching, Addison Wesley, 1973.
[16] B. Lampson, V. Srinivasan and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search.", 1998.
[17] S. Nilsson and G. Karlsson, "Fast Address Look-Up for Internet Routers.", Proceedings of IEEE Broadband Communication 98, Apr. 1998.
[18] S. Nilsson and G. Karlsson, "Implementing a Dynamic Compressed Trie.", Proceedings of WAE'98, Saarbrucken, Germany, Aug. 1998.
[19] Sklower, K., "A Tree-Based Routing Table for Berkeley Unix", Proceeding of the Winter Usenix Conference, 1991.
[20] V. Srinivasan and George Varghese, "Fast Address Lookups using Controlled Prefix", Proceedings of ACM Sigmetrics, Sep. 1998.
[21] Johanaton Turner "Design and Analysis of Switching systems", Washington University, St. Louis,Missouri, Jan. 99.
[22] Marcel Waldvogel, George Varghese, Jon Turner, Bernhard Plattner, "Scalable High Speed IP Routing Lookups", Proceedings of ACM Sigcomm, Sep. 1997.

Dr. Nasser Yazdani has a PhD in computer science and engineering from Case Western Reserve Univ, Cleveland, Ohio, USA. Before his Ph.D. he got his bachelor in computer engineering from Sharif University of Technology, Tehran, Iran and worked in Iran Telecommunication Research Center (ITRC) as a researcher and developer. After his Ph.D. he worked in several companies and research institutes in the USA. In 2000, he joined the ECE Department of University of Tehran, Tehran, Iran, as an associate professor. Dr. Yazdani initiated different research projects and labs in high speed networking. His research interest includes: Networking, packet switching, access methods, Operating Systems and Database Systems.

Hossein Mohammadi received his BS in Computer Science and Engineering from University of Theran in 2002, he joined to the Router Laboratories as one of the founder members in 2001 then received his MSC from the same university in 2004. Currently, he is a Ph.D. candidate and he is working as a research assistant in the Router Laboratory as well as a lecturer in Operating Systems in the ECE school. He is also a member of the core team which is developing the first Iranian IP router named as RAHYAB. His research interests include indexing algorithms, mobility modeling and distributed optimization.