

# Efficient Dynamic IPv4/IPv6 Lookup Scheme Based on the Most Specific Prefixes

Y.-K. Chang, D.-C. Lin, Y.-C. Lin, and C.-C. Chen

*Abstract* -- IP lookup schemes in Internet routers can be broadly classified into two categories: static and dynamic. While static schemes improve lookup speed and reduce memory requirement by using pre-computations, their drawback is that when prefixes are added or deleted, the entire data structure may need to be rebuilt and thus very slow. Dynamic schemes can perform fast prefix insertions and deletions in real time. In this paper, we develop a dynamic scheme called Most Specific Prefix Tree (MSPT). MSPT is a balanced binary search tree based on the most specific prefixes in the routing table. The prefixes other than the most specific ones are allocated to the enclosure sets of some nodes in MSPT. The worst-case time complexities of MSPT search, insertion, and deletion algorithms are  $O(\log M)$ ,  $O(\log M + W)$ , and  $O(\log M \times \log W)$ , respectively, where  $M$  is the number of nodes in MSPT. However, in practice, the time complexities of the MSPT search, insertion, and deletion are all  $O(\log M)$  because  $M$  is very close to the routing table size. Our IPv4/IPv6 experiments show that MSPT not only performs better than the existing dynamic and static schemes in update speed, but also better than the existing dynamic schemes in lookup speed.

*Keywords* : IP address lookup, dynamic routing table, fast update, precomputation

## 1 Introduction

The Internet traffic growth has been increased in an unprecedented rate recently. The exponential growth rate of the traffic on the Internet is mostly due to the advent of the World Wide Web (WWW). Backbone routers with link speed at several 10-gigabits per second (Gbps), such as OC-192, 10 Gigabits and OC-768, 40 Gigabits, are thus commonly deployed. To handle gigabit-per-second traffic rates, these backbone routers must be able to forward millions of packets per second at each port. Moreover, Internet host count also rapidly increases. The scarcity of IP addresses makes us use the Classless Inter-Domain Routing (CIDR) [9] scheme. With CIDR, routers aggregate forwarding information by storing address prefixes that represent a group of addresses reachable through the same interface. Each route entry (prefix) in the routing table can have an arbitrary length ranging from 1 to 32 bits, instead of 8, 16, 24 bits in Classful Address scheme. As a result, the IP lookups in routers become the most critical tasks for packet forwarding. The IP lookups work as follows. When a router receives a packet, the destination address in the packet's header is used to lookup the

routing table for finding the best output port to send the packet. There may be more than one route entries in the routing table that match the destination address. Therefore, it may require some techniques to determine the best match. Since the longest prefix from all the matched entries is the best match for a destination address, the IP lookup problem becomes the longest prefix match (LPM) problem.

To design a good IP address lookup scheme, we need to consider the following key requirements: lookup speed, memory usage, scalability, and update speed.

■ **Lookup speed:** To handle the increased traffic, the IP address lookup scheme should quickly decide where to send the packets. This is clearly important for routers not to be the bottlenecks in the Internet.

■ **Memory usage:** Schemes that are memory-efficient can also lead to good search time because compact data structures can fit in fast but small cache memory.

■ **Scalability:** Due to the fast growth of Internet traffic and increasing demand for the IP addresses, it is expected that routing table size is growing and the address prefix length will significantly increase when switching to IPv6 [6]. Today, IPv6 has been gaining wider acceptance to replace its predecessor, IPv4, and has early deployed in Europe, Asia, and North America [14]. Therefore, an IP address lookup scheme must have the capability of handling large routing tables and longer addresses.

■ **Update:** Route update reflects the changes of the network topology or routing policies. Currently, the Internet has a peak of hundreds or even thousands of BGP updates per second, seen in real routers or in the experimental environment [12], [13], [20], [28]. Thus, the address lookup schemes with fast update time are desirable to avoid routing instabilities. These updates should interfere little with normal IP address lookups.

Various algorithms for high-performance IP address lookup have been proposed in the literature. In the survey paper [24], a large variety of routing lookup algorithms are classified and their complexities of the worst case lookup, update, and memory references are compared, including a category of algorithms based on trie structure. Based on trie structure, a set of prefix compression and transformation techniques are used to either make the whole data structure small enough to fit in a cache, or speedup the tree traversal procedure.

Although the intensive research has been conducted in recent years for IP lookup problem, schemes that address the balance between lookup speed, memory requirement, update, and scalability are needed. The static schemes [2], [3], [5], [7], [10], [15], [24], [26], usually use pre-computations to simplify the data structure of the routing tables and thus improve the lookup speed and reduce memory requirement. However, pre-computations may cause the entire data structure to be rebuilt when a single prefix is added or deleted. Rebuilding the routing tables seriously affects the lookup and update performance of a backbone router. Thus, the schemes based on pre-computations are not suitable for dynamic routing tables. On the other hand, the schemes based on the trie data structure like binary trie, multi-bit trie and Patricia trie [26] do not use pre-computations and thus are good for dynamic routing tables.

The basic idea of the proposed data structure in this paper is to divide the prefixes in a routing table into two groups, the most-specific prefixes (MSP's) and the non most-specific prefixes (non-MSP's). The problem of finding the longest prefix match (LPM) becomes how to organize the MSP's and non-MSP's such that searching for the LPM and updating the routing table are fast. We can use any data structure to organize all MSP's since they are disjoint. The most challenging problem is how to store non-MSP's in the balanced tree constructed from the MSP's such that we only check a limited number of non-MSP's for finding the longest matched prefix when the data structure of MSP's is traversed. We demonstrate this idea by using an AVL tree.

We develop a new data structure suitable for dynamic routing tables called Most Specific Prefix Tree (MSPT). MSPT is an augmented balanced binary search tree (AVL tree). The basic structure of MSPT is constructed by using the most specific prefixes as keys. The most specific prefixes are the ones that do not cover any other prefixes in the routing table. The other prefixes called *non-most specific prefixes* are put in the enclosure sets associated with the nodes in MSPT. The proposed search, insertion, and deletion algorithms for MSPT take  $O(\log M)$  time for real routing tables, where  $M$  is the number of nodes in MSPT. We conduct performance experiments to compare MSPT with the existing dynamic routing table schemes such as the prefix binary tree on binary tree (PBOB) [18] and the multiway range

tree (MRT) [31], and several precomputation-based schemes. In terms of search and memory usage, MSPT performs better than PBOB. In terms of update, MSPT and PBOB perform equally well. However, when 16-bit segmentation table is used, the update speed for MSPT is much better than PBOB. As expected, MRT performs the best in terms of lookup speed because it uses a multiway data structure instead of binary one. The update speed for MRT is worse than MSPT and PBOB. Moreover, lookup speed and memory requirement for MSPT is very close to the static schemes.

The rest of the paper is organized as follows. Section 2 describes the existing IP lookup schemes. Section 3 illustrates the proposed MSPT algorithms and the data structure. The results of performance comparisons using real routing tables are given in Section 4 and Section 5 is the conclusion.

## 2 Related Work

In this section, we review the existing schemes that are close to the designs for dynamic routing tables. We'll not discuss the static schemes [24] designed for improving lookup speed and minimizing the memory consumption nor the ones that need hardware support.

Kim and Sahni [25] developed the first known data structure called a collection of red-black tree (CRBT) that supports search, insert, and delete operations for dynamic routing tables in  $O(\log N)$  time each for a routing table of  $N$  entries. In [17], Lu and Sahni employed the priority search tree (PST) to develop a dynamic routing table data structure with a time complexity of  $O(\log N)$  for a search, an insertion or a deletion. The experiment results showed that PST performs a little worse than CRBT in search time. However, PST performs much better than CRBT in terms of insert, delete, and memory usage. In [18], Lu and Sahni developed a data structure called BOB (binary tree on binary tree) for dynamic routing tables. Based on the BOB, data structures PBOB (prefix BOB) and LMPBOB (longest matching prefix BOB) are also proposed for highest-priority prefix matching and longest-matching prefix. With real routing tables, LMPBOB and PBOB complete the longest prefix match operations in  $O(W)$  and  $O(\log N)$ , respectively. Insert and delete operations both take  $O(\log N)$  time. Although the theoretical complexities of PBOB and LMPBOB are not better than other

dynamic schemes, the experiment results using real routing tables showed that PBOB and LMPBOB perform much better than PST in terms of search, insert, delete, and memory usage. The experiment results measured in [16] showed that all search, insert, and delete operations for PST are even worse than the binary trie when large routing tables are used. One reason that PST does not perform well is because it takes time to transform a route prefix to satisfy the constraints of priority search trees. This transformation complicates the lookup process and increases the memory requirement of data structure. Also, the memory requirement for PST may be too high for IPv6.

Besides the binary balanced tree based schemes proposed for dynamic routing tables, two schemes based on multiway search trees were also proposed in [19] and [31]. Warkhede and others [31] proposed a B-tree data structure called multiway range tree (MRT). MRT finds the longest matching prefix in  $O(\log N)$  time and takes  $O(m \log_m N)$  time for inserting or deleting a prefix, where  $m$  is the order of the B-tree and  $N$  is the number of prefixes in the routing table. MRT is not only suitable for dynamic routing tables because of its fast lookup and update speed, but also can be used for solving the range match problem. However, there are many duplicate endpoints stored in the internal nodes, and a prefix may be stored in at most  $m - 1$  nodes per B-tree level, where  $m$  is the order of the B-tree. This drawback increases the update time and requires a larger memory. Lu and Sahni [19] proposed another B-tree data structure called prefix in B-tree (PIBT) for solving this drawback and hence used memory more efficiently. A crucial difference between PIBT and MRT is that each prefix is stored in  $O(1)$  B-tree nodes per B-tree level in PIBT, while each prefix is stored in  $O(m)$  nodes per level in MRT. The asymptotic complexity to find the longest matching prefix is the same and the measured time for this operation is also nearly the same for PIBT and MRT. Both schemes used in prefix routing tables take  $O(n)$  memory. However, PIBT is more memory efficient than MRT by a constant factor.

The multi-bit tries were usually thought of static IP lookup data structures. They also can be designed to support dynamic updates although their update speeds are usually slower than the dynamic schemes stated above. Variable stride tries based the controlled prefix expansion technique and dynamic programming are proposed in [27]. One of the algorithms

proposed in [27] called Expanded Tries, provides fast lookup times as well as fast update times. The Expanded trie scheme can also be tuned to trade increased memory for reduced search times. The tree bitmap scheme [8] which is based on the 4-bit fixed stride trie can also provide faster search and update speeds.

Other than the dynamic schemes, one alternative to solving update problem is to maintain two routing tables, one active and the other rebuilt for a given number of updates and then hot-swap between them. Although the hot-swap schemes retain the faster query times, they perform updates off-line and thus incur the following three drawbacks. (1) The route information stored in routing tables reflects the topology of the network. Route update messages are generated when the network topology changes due to changes in connectivity or routing policies. Packets are not routed on an optimal path until the routing table is updated. Thus, longer update latency can result in a transient increase in packet round-trip times and packet loss. (2) If the route update algorithm in a router cannot keep up with the updates, a condition known as route flap may be triggered. When a route flap takes place, a router processing a backlog of route updates is incorrectly marked as being unreachable by other routers. This state change creates a domino effect of route updates that can cripple a network. (3) If the routing table is unavailable due to an update, incoming packets must either be buffered or dropped. Dropping TCP packets may trigger the TCP congestion avoidance mechanism, thereby reduce the steady state throughput and the overall network performance.

### 3 The Most Specific Prefix Tree

In this section, we introduce a new data structure called *Most Specific Prefix Tree* (MSPT) that is suitable for dynamic routing tables. We first outline the idea of MSPT and provide notations and definitions to make our presentation clear.

A  $W$ -bit *range*,  $R$ , is denoted as  $[L, U]$  such that  $0 \leq L, U \leq 2^W - 1$  and  $L \leq U$ , where  $W$  is number of bits in the address space of the range. A  $W$ -bit *prefix*  $P$  is denoted as  $p/len$  in the length format or denoted as  $p/netmask$  in the mask format, where  $p$  is a  $W$ -bit number,  $len$  is the prefix length, and  $netmask$  is a  $W$ -bit bitmap.  $P$  can also be treated as a range  $[L, U]$  such that  $L = p - p\%2^{W-len}$  and  $U = L + 2^{W-len} - 1$ , where  $\%$  is the integer modulus operator. A

prefix of length  $W - len$  can also be represented in the *ternary format* as  $b_{W-1}...b_{len}^*...^*$ , where  $b_j = 0$  or  $1$  for  $W - 1 \geq j \geq len$  and  $*$  is the “don’t care” bit. For simplicity, a single don’t care bit is used to denote a series of don’t care bits. Thus, the prefix  $1^*$  denotes  $1^*...^*$  in a 5-bit address space. For example,  $192.168.10.0/24$  is a prefix in its length format which can also be represented in the mask format as  $192.168.10.0/255.255.255.0$  or in the ternary format as  $11000000\ 10101000\ 00001010^*$ .

Let  $R_1 = [L_1, U_1]$  and  $R_2 = [L_2, U_2]$  be two ranges.  $R_1$  and  $R_2$  are said to be *disjoint* if none of them is covered by the other, i.e.,  $U_1 < L_2$  or  $U_2 < L_1$ .  $R_1$  is said to be *smaller than*  $R_2$  (denoted as  $R_1 < R_2$ ) if  $U_1 < L_2$ . We use ‘ $\langle \rangle$ ’ as the *disjoint* operator. Thus,  $R_1 \langle \rangle R_2$  means  $R_1$  and  $R_2$  are disjoint.  $R_1$  and  $R_2$  are *nested* (or *enclosed*) if and only if the address space covered by one range is a subset of that covered by the other, denoted as  $R_2 \supseteq R_1$  or  $R_2 \subseteq R_1$ , where ‘ $\supseteq$ ’ or ‘ $\subseteq$ ’ is called the *nest* or *enclosure* operator.  $R_1$  and  $R_2$  are *intersecting* if and only if  $R_1$  and  $R_2$  are partially overlapped, i.e., either  $L_1 < L_2 \leq U_1 < U_2$  or  $L_2 < L_1 \leq U_2 < U_1$ .

The *longest common ancestor* of two prefixes  $A$  and  $B$  denoted as  $LCA(A, B)$  is the longest prefix that covers both prefixes. Given two prefixes,  $A = a_{W-1}...a_i^*$  and  $B = b_{W-1}...b_j^*$ ,  $LCA(A, B) = c_{W-1}...c_{m+1}^*$  where  $c_k = a_k = b_k$  for  $k = m+1$  to  $W-1$  if  $a_k = b_k$  and  $c_k = *$  for  $k = 0$  to  $m$  if  $a_m \neq b_m$  (i.e.,  $m$  is the most significant bit position such that  $a_m \neq b_m$ ). It is easy to show that for three disjoint prefixes  $A < B < C$ ,  $LCA(A, C)$  also encloses  $B$ .

A prefix is called the *most specific prefix* if it does not enclose any other prefixes in the routing table. Otherwise, it is called the *non-most specific prefix*. *Most Specific Prefix Tree (MSPT)* is a balanced binary search tree constructed by using the most specific prefixes in the routing table as the keys. The prefix key of a node  $x$  is denoted as  $x.prefix$ . The non-most specific prefixes are placed in the *enclosure sets (Eset)* of nodes in MSPT by following the *MSPT enclosure constraint* defined below.

**MSPT Enclosure Constraint:** *Each non-most specific prefix  $p$  is placed in the enclosure set of a node  $x$  (denoted as  $x.Eset$ ) if  $p$  encloses  $x.prefix$ , and  $p$  does not enclose the prefix of any ancestor of node  $x$  in MSPT.*

Notice that the MSPT enclosure constraint is similar to the range allocation rule in [18]. The data structure for the enclosure set of a node in MSPT has a very important impact on the



$P_{10}$ , and  $P_{11}$  in  $P_9.Eset$  are disjoint from  $P_4$ . Let node  $y$  be one of the descendants of a node  $x$  in MSPT and  $EP_x$  and  $EP_y$  be any enclosure prefix in  $x.Eset$  and  $y.Eset$ , respectively. Obviously, it is possible that  $EP_x$  and  $EP_y$  are disjoint. We prove  $EP_x$  must enclose  $EP_y$  by contradiction. Assume  $EP_y$  encloses  $EP_x$ . This assumption results in that  $EP_y$  also encloses  $x.prefix$ . By the definition of MSPT,  $EP_y$  should have been put in  $x.Eset$ . Therefore, if  $EP_x$  and  $EP_y$  are not disjoint, then  $EP_x$  must enclose  $EP_y$  and thus Property (2b) is true. For example, in Figure 1,  $P_1$  in  $P_9.Eset$  encloses  $P_8$  in  $P_2.Eset$ . For two nodes  $x$  and  $y$  in MSPT, we assume none of them is the ancestor of the other. Let  $P_x$  be any enclosure prefix in  $x.Eset$ , and  $P_y$  be  $y.prefix$  or any enclosure prefix in  $y.Eset$ . Also, let node  $z$  be the lowest common ancestor of nodes  $x$  and  $y$  in MSPT. Thus, we have  $x.prefix < z < y.prefix$  or  $y.prefix < z < x.prefix$ . Assume  $P_x$  covers  $P_y$ . Prefix  $P_x$  must also cover  $y.prefix$ . Since  $P_x$  covers both  $x.prefix$  and  $P_y$ ,  $P_x$  must also cover  $z.prefix$ . As a result,  $P_x$  should have been put in  $z.Eset$ . This is a contradiction. Therefore,  $P_x$  and  $P_y$  are disjoint and Property (3) is true. For example, in Figure 1,  $P_1$ ,  $P_{10}$ , and  $P_{11}$  in  $P_9.Eset$  are disjoint from  $P_5.prefix$  and  $P_{12}$  in  $P_5.Eset$ .

### 3.1 Finding the Longest Prefix Match

The longest prefix that matches the destination address  $d$  may be found along a path from the root toward a leaf node in a top-down manner. When a node is traversed, we check if the associated prefix key matches  $d$ . Note that all prefix keys of the nodes in MSPT are disjoint. If a prefix key that encloses  $d$  is found, it must be the longest prefix match (LPM) for  $d$  and the search completes. Otherwise, the enclosure sets associated with the nodes traversed are checked from the leaf to the root in a bottom-up manner. If there are more than one prefixes that enclose  $d$  in a node's enclosure set, the longest one is the LPM for  $d$  and the search stops. If no LPM is found after the root is reached, the default port is returned as the LPM.

Figure 2 shows the algorithm  $MSPT\_Search(d, root)$  that finds the LPM for address  $d$ . In  $MSPT\_Search()$ , the first while loop looks for a node whose prefix encloses  $d$ . In addition, the pointers to the traversed nodes with non-empty  $Eset$  are recorded in array  $node[]$ . In the second for loop, we search the enclosure sets recoded in array  $node[]$  in a bottom-up manner. As long as we find the LPM in an enclosure set, the search process completes. If no match is

```

Algorithm MSPT_Search(d, root) // d is the destination address
{
01 x = root; s = 0;
02 while (x ≠ NULL) {
03   if (x.prefix ⊇ d) return x.port;
04   else
05     if (x.Eset ≠ NULL) {s = s + 1; node[s] = x; }
06     if (d < x) x = x.LeftChild;
07     else x = x.RightChild;
08 }
09 for (i = s; i >= 1; i --) {
10   Find the longest prefix EP in node[i].Eset such that EP encloses d;
11   if (EP ≠ NULL) return EP.port;
12 }
13 return default_port;
}

```

Figure 2. Algorithm to find longest prefix match.

found in the second loop, the default port is returned. To speed up the search in an enclosure set, we can use a 32-bit bitmap to record the lengths of the prefixes in the enclosure set (denoted as  $node[i].bitmap$ ). The set  $i^{th}$  bit in the bitmap indicates that there is an enclosure prefix of length  $i$  in the enclosure set. When we said the most (least) significant set bit in a 32-bit bitmap is  $k$ , we mean all the bits from 31 to  $k + 1$  (from 0 to  $k - 1$ ) are 0 and the bit  $k$  is 1. We also assume that finding the most (or the least) significant set bit in a 32-bit bitmap can be computed by a single CPU instruction (e.g., BSF instruction and FFS instruction supported in Intel Pentium processor and IXP 24xx network processor, respectively), similar to the other bit manipulation instructions such as AND, OR, and XOR. If the least significant set bit of the bitmap is  $k$ , all other enclosure prefixes in the enclosure set must be longer than  $k$ . Also, if any enclosure prefix  $P$  in  $node[i].Eset$  covers  $d$ , the prefix length of  $P$  must be shorter than or equal to  $LCA(d, node[i].prefix)$  because  $P$  also covers  $node[i].prefix$ . If  $k$  is shorter or equal to  $LCA(d, node[i].prefix)$ ,  $node[i].Eset$  must contain an enclosure prefix that covers  $d$ . As a result, finding the longest prefix in a node's enclosure set takes a constant time by using the bitmap because only a constant number of instructions are needed. Therefore, the complexity of function  $MSPT\_Search()$  is  $O(\log M)$ , where  $M$  is the number of nodes in MSPT. Note that  $M$  is the number of the most specific prefixes and thus is less than the total number of prefixes in the routing table. As we will show in the routing table analysis that  $M$  is about 91-93% of  $N$  for the routing tables we tested.

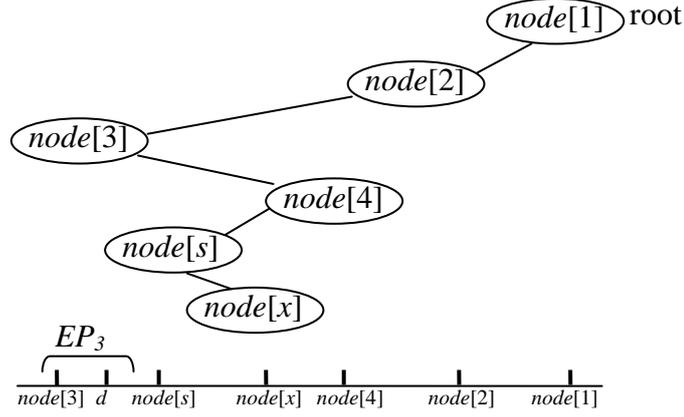


Figure 3. An MSPT example, where  $EP_3 \in node[3].Eset$  and  $EP_3$  encloses both  $d$  and  $node[3].prefix$ .

### Proof of correctness of the algorithm `MSPT_Search()`.

Let  $node[i]$  for  $i = 1$  to  $s$  be the nodes traversed when searching MSPT with destination address  $d$ , where  $node[1]$  is the root of MSPT. Figure 3 shows an example for  $s = 5$ . Node  $node[s]$  may have a child denoted as  $node[x]$  which is not traversed as shown in Figure 3. To prove the correctness of the algorithm `MSPT_Search()`, we have to consider three cases. Case 1 - if a most specific prefix that matches  $d$  is found in the first while loop, it must be the longest matching prefix in the routing table. Case 2 - if a prefix in  $node[i].Eset$  that matches  $d$  is first found in the second for loop, it must be the longest match prefix in the routing table and thus no other enclosure set in MSPT needs to be checked. Case 3 -  $node[x].prefix$  and prefixes in  $node[x].Eset$  do not need to be checked. Case 4 - no enclosure set other than the ones traversed needs to be checked.

In the first case, the first while loop examines  $node[i]$  from  $i = 1$  to  $s$  in a top-down manner. If  $node[i].prefix$  covers  $d$ , it must be the LPM because all the prefixes in nodes are the most specific and disjoint. In the second case, all prefixes  $node[i].prefix$  for  $1 \leq i \leq s$  do not cover  $d$ . We check  $node[i].Eset$  from  $i = s$  to 1 in a bottom-up manner. If we find the longest prefix match of  $d$  in  $node[i].Eset$ , we don't need to check the enclosure sets  $node[k].Eset$  for  $k = i - 1$  to 1 because of Property (2b).

Since node  $x$  is not traversed when searching MSPT with destination address  $d$ , we must have either  $d < node[s].prefix < node[x].prefix$  or  $node[x].prefix < node[s].prefix < d$ . From Property (2a), we know that  $node[x].prefix$  and any enclosure prefix in  $node[x].Eset$  are

disjoint from  $node[s].prefix$  and thus they must also disjoint from  $d$ . Therefore,  $node[x].prefix$  and  $node[x].Eset$  don't need to be checked and the case 3 is true. From Property (2a) and Property (3), we know all the prefixes in the enclosure sets other than the ones traversed are disjoint from the prefixes traversed. As a result, case 4 is true.

Consider the example in Figure 3 and the address  $d$  that does not match any most specific prefix. Therefore, the second for loop must be executed. Assume  $node[3].prefix < d < node[s].prefix$  and there exists an enclosure prefix  $EP_3$  in  $node[3].Eset$  that covers  $d$ . Assume  $node[2]$  also contains an enclosure prefix  $EP_2$  in its enclosure set that covers  $d$ . Since both  $EP_2$  and  $EP_3$  cover  $d$ ,  $EP_2$  must enclose  $EP_3$  because of the MSPT constraint. Thus,  $EP_2$  cannot be the LPM of  $d$ . Assume  $node[x]$  contains an enclosure prefix  $EP_x$  that covers  $d$ . Since  $d < node[s].prefix < node[x].prefix$ ,  $EP_x$  must enclose  $node[s].prefix$  and thus the MSPT enclosure constraint is violated. Therefore,  $node[x].Eset$  does not need to be checked. In summary, starting from  $node[s]$  to  $node[1]$ , if the enclosure set of the first node traversed contains an enclosure prefix  $EP$  that covers address  $d$ ,  $EP$  must be the LPM of  $d$ .

### 3.2 Insertion

A prefix  $P$  is inserted by performing a binary search tree traversal from root to a leaf node or to a node with only one child. Similar to function `MSPT_Search`, the pointers to the nodes traversed are recorded in array  $node[]$ . If we find a node  $x$  whose prefix key ( $x.prefix$ ) encloses  $P$ , then  $x.prefix$  is put in  $x.Eset$  and  $x.prefix$  is replaced with  $P$ . Otherwise, if  $x.prefix$  is enclosed by  $P$ , then  $P$  is put in  $x.Eset$  directly. However, if  $P$  is disjoint from  $x.prefix$ , then the same process repeats on  $x$ 's left child if  $P$  is smaller than  $x.prefix$  or on  $x$ 's right child otherwise. Finally, when all the prefixes associated with the nodes traversed are disjoint from  $P$ , a new node associated with  $P$  is created and inserted in MSPT as the child of the last traversed node. It is possible that MSPT becomes unbalanced after a new node is inserted. We use the pointers stored in array  $node[]$  in a bottom-up manner to check if any sub-tree rooted at  $node[i]$  is unbalanced. If so, the tree balancing operation at  $node[i]$  is performed.

Figure 4 shows the MSPT insertion algorithm `MSPT_insert( $P$ , root)`. In the while loop, we find a node  $x$  such that  $P$  encloses  $x.prefix$  or  $P$  is enclosed by  $x.prefix$ . If such a node  $x$

```

Algorithm MSPT_insert(P, root)
{
01 x = root; s = 0;
02 while (x ≠ NULL) {
03   s = s + 1;
04   node[s] = x;
05   if (P = x.prefix) return;
06   if (P ⊆ x.prefix) { // x.prefix encloses prefix P
07     x.Eset = x.Eset + {x.prefix};
08     x.prefix = P; return; }
09   if (x.prefix ⊆ P) { // prefix P encloses x.prefix
10     x.Eset = x.Eset + {P}; return; }
11   if (P > x.prefix) x = x.RightChild;
12   else x = x.LeftChild;
13 }
14 new_node = Create_A_Node(P); // create a new node and insert it in MSPT
15 if (P > node[s].prefix) node[s].RightChild = new_node;
16 else node[s].LeftChild = new_node;
17 BST_Balancing(node, s); // Perform balancing operation in a bottom-up manner
}

```

Figure 4. Algorithm to insert a prefix.

exists, we insert  $P$  into  $x.Eset$  if  $P$  encloses  $x.prefix$  or insert  $x.prefix$  into  $x.Eset$  and replace  $x.prefix$  with  $P$  if prefix  $P$  is enclosed by  $x.prefix$ . We use array  $node[]$  to store the pointers pointing to the nodes traversed in the while loop. Notice that the time complexity of putting a prefix in a node's enclosure set is  $O(\log W)$  when using ESBT or  $O(1)$  when using ESLA.

The codes following the while loop must be executed when  $P$  is disjoint from all the prefix keys in MSPT.  $P$  is inserted as the child of the last traversed node. Finally, the tree balancing operation (function  $BST\_Balancing()$ ) must be performed because MSPT may not be balanced after a new node is inserted. The balancing operation searches the unbalanced node by following the pointers in array  $node[]$  in a bottom-up manner. As we know that if the balanced search tree is implemented with AVL tree, the tree balancing needs at most one rotation for insertion. However, a tree rotation may result in the violation of MSPT enclosure constraint. We will discuss the rotation and tree balancing problems in section 3.4 and show that a rotation requires  $O(\log W)$  time when using ESBT or  $O(W)$  time when using ESLA.

As a result, *the time complexity for insertion is  $O(\log M + \log W)$  when using ESBT or  $O(\log M + W)$  when using ESLA, where  $M$  is the number of nodes in MSPT.*

### 3.3 Deletion

```

Algorithm MSPT_delete(P, root)
{
01 x = root; s = 0;
02 while (x ≠ NULL) {
03   if (P = x.prefix) {
04     if (x is a leaf node or x has only one child) // Case 2
05       Delete_leaf_prefix_node(x, node, s);
06     else Delete_internal_prefix_node(x, node, s); // Case 3
07     return;
08   }
09   if (x.prefix ⊆ P) { x.Eset = x.Eset − {P}; return; } // Case 1
10   if (P ⊆ x.prefix) return; // This case is not possible
11   s = s + 1; node[s] = x;
12   if (P > x.prefix) x = x.RightChild;
13   else x = x.LeftChild;
14 }
}

```

Figure 5. Algorithm to delete a prefix.

Similar to the other existing dynamic schemes, deleting a prefix is always more complicated than lookup or inserting a prefix. Therefore, we break down the deletion procedure into three cases. Case 1 - Delete an enclosure prefix in the enclosure set of a node. Case 2 - Delete the prefix key of a leaf node or a node with only one child. Case 3 - Delete the prefix key of a node with two children. Figure 5 shows the algorithm *MSPT\_delete*.

Case 1 is trivial. We only need to remove the prefix from the enclosure set. No node in MSPT is really deleted. Thus, tree rotation and enclosure set adjustment are not needed.

In case 2, the following sub-cases are considered. Assume the prefix to be deleted is in node *x*. (1) If *x* is a leaf and *x.Eset* is empty, then node *x* is deleted from MSPT and tree rotation may be needed by calling *BST\_Balancing()*. (2) If *x* is a leaf and *x.Eset* is not empty, then the longest prefix is removed from *x.Eset* and replaces *x.prefix*. Tree rotation is not needed because the MSPT properties described earlier are still satisfied. (3) If node *x* has a child node *y*, *x.Eset* is not empty, and the longest prefix (*EP*) in *x.Eset* does not enclose *y.prefix*, then prefix *EP* is removed from *x.Eset* and replaces *x.prefix*. Tree rotation is also not needed. (4) If node *x* has a child node *y* and *x.Eset* is empty or all the prefixes in *x.Eset* enclose *y.prefix*, then *x.prefix* is replaced with *y.prefix* and the prefixes in *y.Eset* is added in *x.Eset*. Finally, the leaf node *y* is freed and the tree balancing is performed by using *BST\_Balancing()*. Figure 6 shows the detailed algorithm for this case. Excluding the time

```

Algorithm Delete_leaf_prefix_node( $x, node, s$ )
{
01 if ( $x$  is a leaf node){
02   if ( $x.Eset \neq \text{NULL}$ ){
03      $EP =$  the longest prefix in  $x.Eset$ ;
04      $x.prefix = EP$ ;  $x.Eset = x.Eset - \{EP\}$ ;
05   } else {  $\text{free\_node}(x)$ ;  $\text{BST\_balancing}(node, s)$ ; }
06 } else { // node  $x$  has only one child
07    $y =$  the only child of  $x$ ;
08   if ( $x.Eset = \text{NULL}$ ) {
09      $x.prefix = y.prefix$ ;  $x.Eset = y.Eset$ ;
10      $\text{free\_node}(y)$ ;  $\text{BST\_balancing}(node, s)$ ;
11   } else {
12      $EP =$  the longest prefix in  $x.Eset$ ; // '<>' is the disjoint operator
13     if ( $EP \langle \rangle y.prefix$ ) {  $x.prefix = EP$ ;  $x.Eset = x.Eset - \{EP\}$ ;
14     } else {
15        $x.prefix = y.prefix$ ;  $x.Eset = x.Eset \cup y.Eset$ ;
16        $\text{free\_node}(y)$ ;  $\text{BST\_balancing}(node, s)$ ; }
17   }
18 }
}

```

Figure 6. Algorithm to delete a leaf node or a node with only one child.

taken for procedure *BST\_Balancing()*, the time complexity for the tasks done in algorithm *Delete\_leaf\_prefix\_node()* is  $O(\log W)$  for ESBT or  $O(W)$  for ESLA. We shall discuss the procedure *BST\_Balancing()* in subsection 3.4.

Now consider case 3 for deleting the prefix that is equal to the prefix key of an internal node  $P$  with two children. It is the most complicated case for deletion because three issues described below must be considered after the prefix is removed from MSPT. The first issue is how to select a node such that the prefix key of the selected node can replace  $P$ . The second issue is how to adjust the enclosure sets in order to not violate the MSPT enclosure constraint. The third issue is how to rebalance MSPT if a node is really deleted from MSPT, which is the procedure *BST\_Balancing()* described later. In order to clearly describe the proposed algorithm, the third case of deleting the prefix in node  $x$  is further decomposed into two sub-cases: when  $x.Eset$  is empty or not. The time complexity in the third case is  $O(\log M + \log M \times \log W)$  for ESBT or  $O(\log M + W)$  for ESLA, where  $M$  is the number of nodes in MSPT.

(1)  $x.Eset$  is empty.

We need to determine how to select a node to replace node  $x$  and find the enclosure set of the replacing node. Tree balancing will be described later in subsection 3.4.

(a) Selecting a node to replace node  $x$ :

The traditional method to delete an internal node  $x$  with two children in the binary search tree is to select a node to replace  $x$  and then delete the selected node. The selected node is the one with the smallest (or largest) key among all the keys that are larger (or smaller) than  $x$ . In a balanced binary search tree, the selected node must be a leaf node or a node having only one child. After the replacement, the deletion problem becomes deleting a leaf node or a node with only one child. This is the second case for deletion solved earlier. Assume the two nodes that can be selected to replace node  $x$  are the node  $y$  and the node  $z$  as shown in Figure 7. Selecting a node to replace  $x$  improperly may result in tree rotations in all the levels of the balanced tree from the selected node to the root. We use AVL tree as the example to illustrate our idea to achieve the goal of reducing the number of required rotations. Assume each node maintains a balance factor (bf) defined as the height of node's left subtree minus that of the node's right subtree. Therefore, based on the definition of AVL tree, bf can only be  $-1$ ,  $0$ , or  $+1$ .

Consider if node  $x$  is replaced by  $y$ . The heights of the subtrees rooted at node  $R_i$  ( $1 \leq i \leq s$ ) may remain the same or be reduced by one when node  $y$  is deleted. We say that the height reduction (denoted by  $HR$ ) of the subtree rooted at  $R_i$  is  $0$  or  $1$  after node  $y$  is deleted. Two factors that determine if a rotation is needed for node  $R_i$  are the balance factor of  $R_i$  ( $R_i.bf$ ) and the height reduction of the  $R_i$ 's left subtree ( $R_i.leftHR$ ). When a rotation is needed for  $R_i$  and what is the height reduction of subtree rooted as  $R_i$  will be analyzed as follows.

We know that  $R_i.leftHR$  could be  $0$  or  $1$ . If  $R_i.leftHR = 0$ , no rotation is needed for  $R_i$  because the balance condition of  $R_i$  is unchanged. It is easy to see that no rotation is needed for  $R_k$  ( $1 \leq k < i$ ) too. Now we consider when  $R_i.leftHR = 1$ . If  $R_i.bf$  is  $1$  before  $y$  is deleted,  $R_i.bf$  becomes  $0$  and the height reduction of  $R_i$  is  $1$  after  $y$  is deleted. Since  $R_i.bf = 0$ , no rotation is needed for  $R_i$ . If  $R_i.bf$  is  $0$  before  $y$  is deleted,  $R_i.bf$  becomes  $-1$  and the height reduction of  $R_i$  is  $0$  after  $y$  is deleted. As a result, no rotation is needed for  $R_i$ . If  $R_i.bf$  is  $-1$

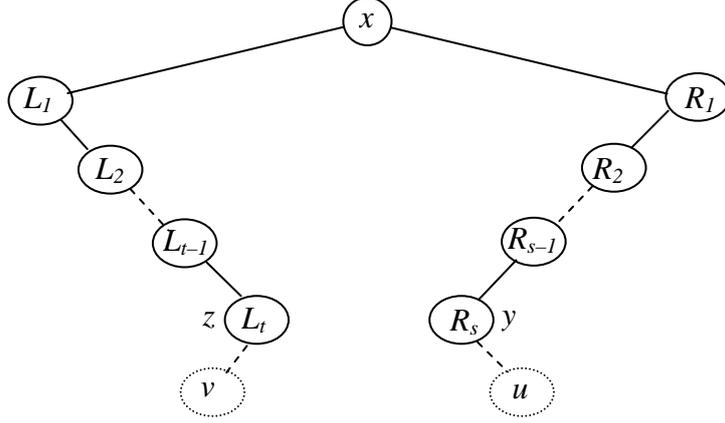


Figure 7. Node  $x$  is to be replaced. Nodes  $y$  and  $z$  are the smallest in  $x$ 's right subtree and the largest in  $x$ 's left subtree, respectively. Node  $y$  (or  $z$ ) may have a child node labeled as  $u$  (or  $v$ ).

before  $y$  is deleted,  $R_i.bf$  becomes  $-2$  after  $y$  is deleted and thus a rotation is needed for  $R_i$ . After rotation,  $R_i.bf$  becomes  $0$  and the height reduction of  $R_i$  is  $1$ .

Based on the above analysis, the best choice is to select  $y$  to replace  $x$  when the balance factor of  $R_{s-1}$  ( $y$ 's parent) is  $0$ . The other best choice is to select node  $z$  to replace  $x$  when the balance factor of  $L_{t-1}$  ( $z$ 's parent) is  $0$ . We can break the tie between  $y$  and  $z$  arbitrarily. In general, we compute the total number of rotations needed when  $y$  or  $z$  deleted. The computation process is done from  $R_{s-1}$  to  $R_1$  and from  $L_{t-1}$  to  $L_1$  in a bottom-up manner. The process can be performed in  $O(\log M)$  time. We select the node  $y$  to replace  $x$  if deleting  $y$  generates less rotations than deleting node  $z$ . Otherwise we select  $z$  to replace  $x$ .

(b) Adjust the enclosure set of the replacing node:

Assume the replacing node is  $y$  and thus  $x.prefix$  is replaced with  $y.prefix$ . This step looks for the prefixes that enclose  $y.prefix$  in the enclosure sets of nodes  $R_i$  for  $i = s$  to  $1$ . Let  $EP_i$  be an enclosure prefix in  $R_i.Eset$  that encloses  $y.prefix$  for  $i = 1$  to  $s$ . The possible length of  $EP_i$  must meet the following restrictions. (1) Since  $EP_i$  encloses both  $y.prefix$  and  $R_i.prefix$ , we must have  $\text{length}(EP_i) \leq \text{length}(LCA(y.prefix, R_i.prefix))$ . (2) Since  $x.prefix$  is not enclosed by  $EP_i$ , we must have  $\text{length}(EP_i) > \text{length}(LCA(x.prefix, R_i.prefix))$ . (3) Since  $R_j$  is in the left subtree of  $R_i$  for  $j > i$ , we must have  $R_j.prefix < R_i.prefix$  and also  $\text{length}(EP_j) > \text{length}(EP_i)$ . These three restrictions narrow down the lengths of enclosure prefixes that cover  $y.prefix$  when this step is executed from the  $R_s$  up to  $R_1$ . Therefore, if the enclosure sets are organized as linear arrays, we can find the enclosure set of  $y.prefix$  from  $R_s$  to  $R_1$  and move them to

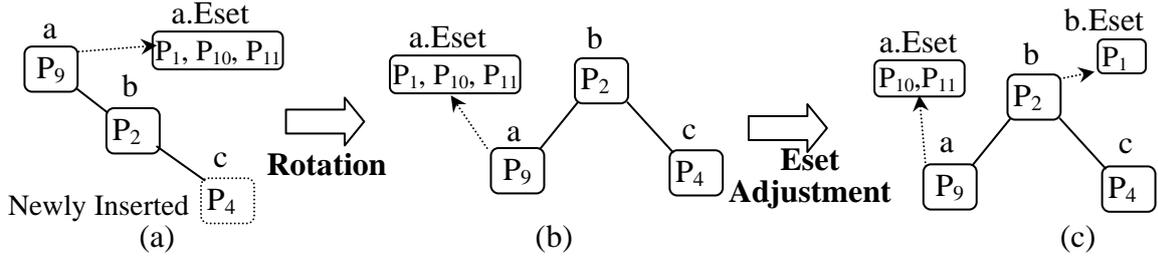


Figure 8. Rotation example where the prefixes are defined in Figure 1.

node  $x$  with the worst case time complexity of  $O(W)$ . However, if the enclosure sets are implemented as balanced binary search trees, the worst case time complexity of this step becomes  $O(\log M \times \log W)$ .

**Optimizations.** Two optimizations can be used to reduce or avoid the overhead in adjusting the enclosure set of the replacing node and the tree balancing operation. The first optimization that was proposed in [18] does not really free the node  $x$  that is supposed to be freed. The node  $x$  is marked as invalid and  $x.prefix$  is only used for the search operation. Some leaf nodes or nodes with one child are deleted when the number of nodes in MSPT is double of the number of the most specific prefixes in the routing table. Based on this optimization, the enclosure set adjustments and the tree balancing operations can be avoided completely. It may be possible that an invalid node can be selected to store a newly inserted prefix. However, since the newly inserted prefixes are normally inserted as the leaf nodes, the invalid leaf nodes have a better chance to be used as the placeholders for the newly inserted prefixes than the invalid internal node. We use the following example to describe the second optimization. If node  $x$  in Figure 7 is an invalid node and the newly inserted prefix happens to be inserted as the left child of node  $y$ , then the second optimization replaces  $x.prefix$  with  $y.prefix$ , performs the enclosure set adjustments for  $y.prefix$ , and put the newly inserted prefix in node  $y$ . As a result, the tree balancing operations are avoided completely. Notice that the tree balancing operations are more time-consuming than the enclosure set adjustments.

(2)  $x.Eset$  is not empty.

Let  $EP_x$  be the longest prefix in  $x.Eset$ . Four cases are considered as follows.

Case 1:  $EP_x$  is disjoint from both  $y.prefix$  and  $z.prefix$  ( $y.prefix \diamond EP_x$  and  $z.prefix \diamond EP_x$ ).

Obviously,  $EP_x$  is disjoint from all the most specific prefixes in MSPT. Thus, we remove  $EP_x$  from  $x.Eset$  and replace  $x.prefix$  with  $EP_x$ .

Case 2:  $EP_x$  encloses  $y.prefix$  and,  $EP_x$  and  $z.prefix$  are disjoint ( $y.prefix \subseteq EP_x$  and  $z.prefix \diamond EP_x$ ). Assume  $R_iEset$  contains an enclosure prefix  $EP_i$  where  $1 \leq i \leq s - 1$ . Based on the MSPT properties,  $x.prefix$  and  $EP_i$  must be disjoint. If  $EP_i$  encloses  $y$ ,  $EP_x$  must also enclose  $EP_i$ . Therefore, if such prefix  $EP_i$  exists,  $EP_i$  has to be moved to node along with  $y.Eset$ . In other words, the step (b) of adjusting the enclosure set of the replacing node is not needed. As a result, the following steps are performed: (1)  $x.prefix$  is replaced with  $y.prefix$ . (2)  $y.Eset$  is appended to  $x.Eset$ . (3) Node  $y$  is removed. (4) The enclosure set of  $y.prefix$  is adjusted.

Case 3:  $EP_x$  encloses  $z.prefix$  and  $EP_x$  and  $y.prefix$  are disjoint ( $y.prefix \diamond EP_x$  and  $z.prefix \subseteq EP_x$ ). We replace  $x.prefix$  with  $z.prefix$  and other operations are similar to case 2.

Case 4:  $EP_x$  encloses both  $y.prefix$  and  $z.prefix$  ( $y.prefix \subseteq EP_x$  and  $z.prefix \subseteq EP_x$ ). It is similar to case 2 or 3. We can either perform the same steps in case 2 or in case 3. The one that needs fewer rotations is better.

One remaining task in cases 2, 3, and 4 is the tree balancing after a node is freed. As a result, *the time complexity for deletion is  $O(\log W \times \log M)$  when using ESBT or  $O(W \times \log M)$  when using ESLA, where  $M$  is the number of nodes in MSPT.* The same optimizations for the sub-case where  $x.Eset$  is empty can also be used here. We will not show the algorithm *Delete\_internal\_prefix\_node( $x, node, s$ )* in details because of the space limit.

### 3.4 Rotation and Tree Balancing

When inserting/deleting a node into/from the balanced MSPT, one or more rotations may be needed to rebalance MSPT. Those rotations may result in a violation of enclosure constraint and the search operation may fail. Based on the MSPT properties, all non-most specific prefixes allocated to the left subtree of a node  $x$  in MSPT must be disjoint from  $x.prefix$  and smaller than  $x.prefix$ . Similarly, the prefixes allocated to the right subtree of node  $x$  in MSPT must be disjoint from  $x.prefix$  and larger than  $x.prefix$ . For example, Figure 8(a)

shows that a new node  $c$  associated with prefix  $P_4$  is inserted as the right child of node  $b$ . After balancing MSPT as shown in Figure 8(b), the enclosure constraint is violated because  $P_1$  also encloses  $P_2$ . Figure 8(c) shows the correct MSPT after adjusting the enclosure sets.

Balanced binary search tree can be implemented by a Red-Black tree or an AVL tree. Figure 9(a) shows the LL rotation used to balance the balanced binary search tree following an insert or a delete operation. Figure 9(b) shows the LR rotation if the AVL tree is used. RR and RL rotations are not shown because they are similar to the LL and LR rotations. We may respectively view the LR and RL rotations as a RR rotation followed by an LL rotation and an LL rotation followed by an RR rotation.

In Figure 9(a), we observe that the relative positions of nodes  $a$  and  $b$  change after performing an LL or RR rotation. Node  $a$  becomes the child of node  $b$ . To avoid violating the MSPT enclosure constraint, we have to find a set  $S$ , such that after performing a rebalancing rotation,  $b.Eset = b.Eset \cup S$  and  $a.Eset = a.Eset - S$ , where  $S = \{ p \mid p \in a.Eset \text{ and } p \text{ encloses } b.prefix \}$ . The time required to perform an LL and RR rotation depends on how to determine the set  $S$ , remove  $S$  from  $a.Eset$ , and then insert  $S$  into  $b.Eset$ . The time taken for an LR or RL rotation is roughly twice as long as that for an LL or RR rotation.

To find the set  $S$ , we can just find the prefix  $pMax$  with the longest prefix length that encloses  $b.prefix$  in the  $a.Eset$ . As a matter of fact,  $pMax$  encloses both  $a.prefix$  and  $b.prefix$  and thus  $pMax$  must be shorter than or equal to  $LCA(a.prefix, b.prefix)$ . If the data structure for the enclosure set is a linear array (ESLA), set  $S$  can be easily determined to be the prefixes that are shorter than or equal to  $LCA(a.prefix, b.prefix)$ . Thus, the complexity of performing a rotation is  $O(W)$ . Moreover, if the data structure for the enclosure set is a balanced binary search tree of an ordered set of prefix lengths (ESBT), the prefix  $pMax$  can be found in  $O(\text{height}(a.Eset))$  time by following a path from the root to a leaf node. If  $pMax$  exists, we can use the *split* [11] operation to extract the prefixes that belong to  $S$  from  $a.Eset$ . We first separate  $a.Eset$  into a balanced binary tree  $TSmall$  in which all prefixes are shorter than  $pMax$  and a balanced binary tree  $TBig$  in which all prefixes are not longer than  $pMax$ . Then we use the *join* [11] operation to combine the tree  $TSmall$ , prefix  $pMax$ , and the tree  $b.Eset$  into a single balanced binary tree. Finally, we have  $a.Eset = TBig$ , and  $b.Eset =$

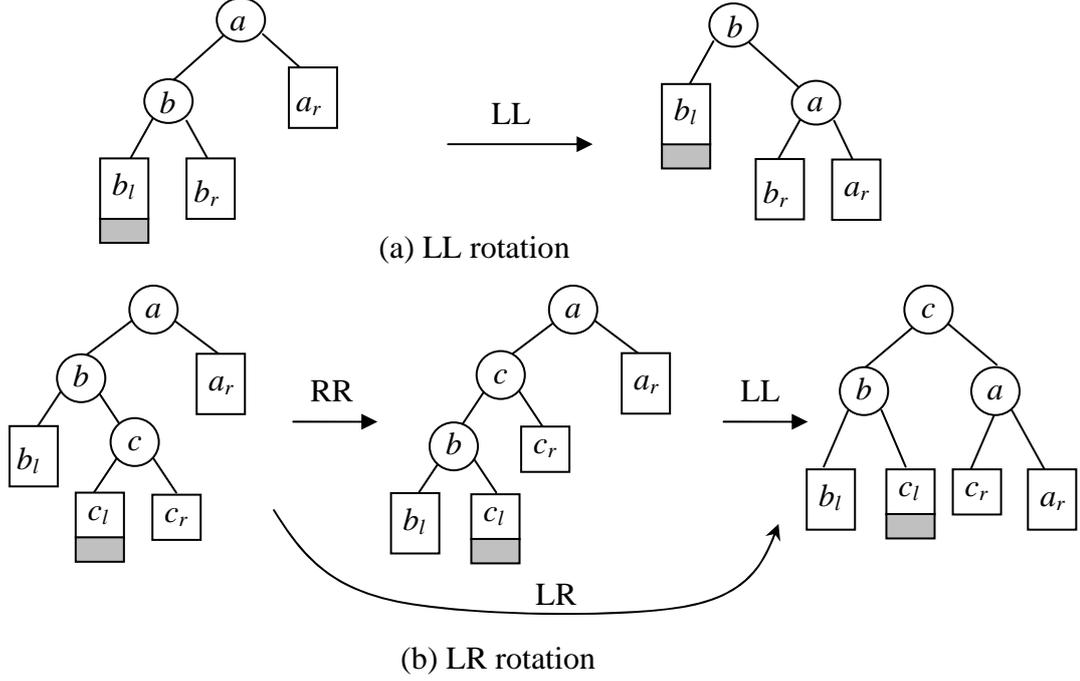


Figure 9. LL and LR rotations.

$join(T_{small}, p_{Max}, b.Eset)$  after performing a rebalancing rotation. Although, the *split* and *join* operations of [11] need to be modified slightly, this modification does not affect the complexity. So, the complexity of performing an LL or RR (LR or RL rotation) rotation in the enclosure set is  $O(\log W)$ .

*Tree Balancing.* Tree balancing operation is performed in a bottom-up manner after a node is really freed. No matter which balanced binary search tree (AVL or Red-Black tree) is used, rotation may be required for every node traversed from node  $R_{s-1}$  up to  $R_l$  or from  $L_{t-1}$  up to  $L_l$  (see Figure 7). Theoretically, there are  $O(\log M)$  nodes that need rotations. Based on the rotation result stated above, the tree balancing requires  $O(W \times \log M)$  time for ESLA  $O(\log W \times \log M)$  time for ESBT.

### 3.5 Analysis

As stated in previous sections, the most time-consuming operation in MSPT is the tree rotation which involves moving the non-most specific prefixes between nodes. The number of the non-most specific prefixes [21] and their locations in MSPT are two very important factors that will influence the performance of MSPT. Therefore, we analyze the six BGP

Table 1. Data structure analysis for MSPT.

Routing tables (year-month)	AS6447a (2000-4)	AS6447b (2002-4)	AS2493 (2005-4)	Amsterdam (2006-5)	London (2006-5)	Frankfurt (2006-5)
# of prefixes	79,535	124,803	157,027	187,444	188,979	191,810
# of the most specific prefixes	73,900	114,745	143,684	171,494	172,938	175,549
# of the non-most specific prefixes	5,635	10,058	13,343	15,950	16,041	16,261
# of empty enclosure sets	68,639	105,561	131,520	156,979	158,314	160,753
# of nonempty enclosure sets	5,261	9,184	12,164	14,515	14,624	14,796
Max. size of nonempty enclosure sets	4	5	4	5	5	5
Avg. size of nonempty enclosure sets	1.07	1.10	1.10	1.10	1.10	1.10

routing tables we use in the paper, and show the detailed statistics in Table 1. These tables are obtained from [1], [22], and [23].

In Table 1, we observe that about 91% ~ 93% prefixes in a routing table are the most specific prefixes. Almost 91% ~ 93% enclosure sets are empty. The average size of all enclosure sets is very small (0.08 to 0.09). Excluding the empty enclosure sets, the average size of nonempty enclosure sets is also small (1.07 to 1.10) and the maximum size of nonempty enclosure sets is 4 or 5. Therefore, the performance overhead of tree rotations that redistribute the prefixes of the enclosure sets in MSPT will not be significant.

As described earlier, we have used a balanced tree to store the prefixes in each enclosure set. However, the balanced tree is only suitable for a large number of keys instead of a few keys. Therefore, better performance can be obtained by using a simple linear list to store each enclosure set. The linear list is in ascending order of prefix length. As explained before, a 32-bit bitmap that only records the prefix lengths of the prefixes in the enclosure set can speed up the match decision.

Based on the above analysis for the average sizes of enclosure sets (0.08 to 0.09) and non-empty enclosure sets (1.07 to 1.10), it is reasonable to assume that the number of prefixes in any enclosure set is a constant instead of  $O(W)$ . Also, since most of the enclosure sets are empty as shown in Table 1, no time is needed for searching enclosure prefix when adjusting the enclosure set. Thus a tree rotation can be done in a constant time and tree balancing can be done in  $O(\log M)$  time. Therefore, in practice, the MSPT takes  $O(\log M)$  time to perform a search, an insertion, or a deletion.

## 4 Performance Evaluations

In this section, we first present the performance results for IPv4 routing tables. The six BGP tables [1], [22], [23] analyzed in previous section are used in the experiments. In addition to the proposed MSPT algorithm, we also experiment on five existing dynamic schemes, the dynamic segment tree [4], the prefix binary tree on the binary tree structure (PBOB) [18], the multiway range tree (MRT) [31], the prefix in B-tree (PIBT) [19], and the tree bitmap scheme [8]. In the experiments, both MRT and PIBT use 32-way B-tree data structures. The dynamic schemes with names prefixed with "OLDP-" (for example, OLDP-PBOB) are based on the one-level dynamic partitioning (OLDP) method proposed in [16]. OLDP method is a variant of the 16-bit segmentation table. Moreover, our performance results also include the static schemes such as binary search on prefix lengths (BLS) [29], the small forwarding table (Lulea) [7], a compressed table structure (Pisa) [5], and binary range search (BRS-16) [15] enhanced with a 16-bit segmentation table.

All tested schemes are implemented in C. GNU gcc-3.3.5 compiler enabled with optimization level O4 is used. The experiments are conducted on a Debian GNU/Linux 3.1 platform with a Pentium 4 2.4GHz processor containing 8KB L1, 512KB L2 caches and 512MB main memory.

**Total Memory Requirement.** Table 2 shows the amount of memory consumed by each of the tested schemes. Figure 10 shows the same results in bar chart form. The pre-computation-based schemes except Pisa scheme have the smaller memory requirement. Except Tree-Bitmap scheme, MSPT or OLDP-MSPT has a better performance than all dynamic schemes. Compared with PBOB, MSPT uses about 30% less memory than PBOB. This result can be attributed to the small number of nodes and the small number of non-empty enclosure sets in MSPT. Moreover, in PBOB, less than 1 % of the range sets (similar to the enclosure sets in MSPT) are empty. Therefore, additional memory is needed to store those nonempty range sets. On the contrary, almost all enclosure sets in MSPT are empty. Hence, MSPT always requires less memory space than PBOB. DST needs more memory than MSPT and PBOB because some prefixes are duplicated in DST. As for PIBT and MRT, they both need larger memory space because their node size is large and not all the pointers and keys in the nodes are used.

Table 2. The statistics of memory requirement (in KB) for IPv4.

Routing tables	AS6447a	AS6447b	AS2493	Amsterdam	London	Frankfurt
Dynamic Schemes						
MSPT	1,433	2,237	2,809	3,684	3,714	3,771
DST	3,010	4,120	5,008	5,820	5,880	5,990
PBOB	2,122	3,299	4,129	4,940	4,980	5,055
MRT	3,665	5,689	7,099	8,542	8,591	8,754
PIBT	3,596	5,636	7,017	8,452	8,536	8,661
Tree bitmap	1,323	2,000	2,443	2,867	2,894	2,935
OLDP-MSPT	1,828	2,533	3,027	3,855	3,882	3,933
OLDP-DST	3,441	4,468	5,341	6,190	6,235	6,316
OLDP-PBOB	2,465	3,550	4,298	5,213	5,253	5,323
Static Schemes						
BRS-16	1,474	1,972	2,308	2,511	2,527	2,558
BLS	1,459	2,315	3,292	3,817	3,940	4,102
Lulea	521	802	895	1,026	1,036	1,046
Pisa	2,089	2,749	4,703	10,125	10,165	10,170

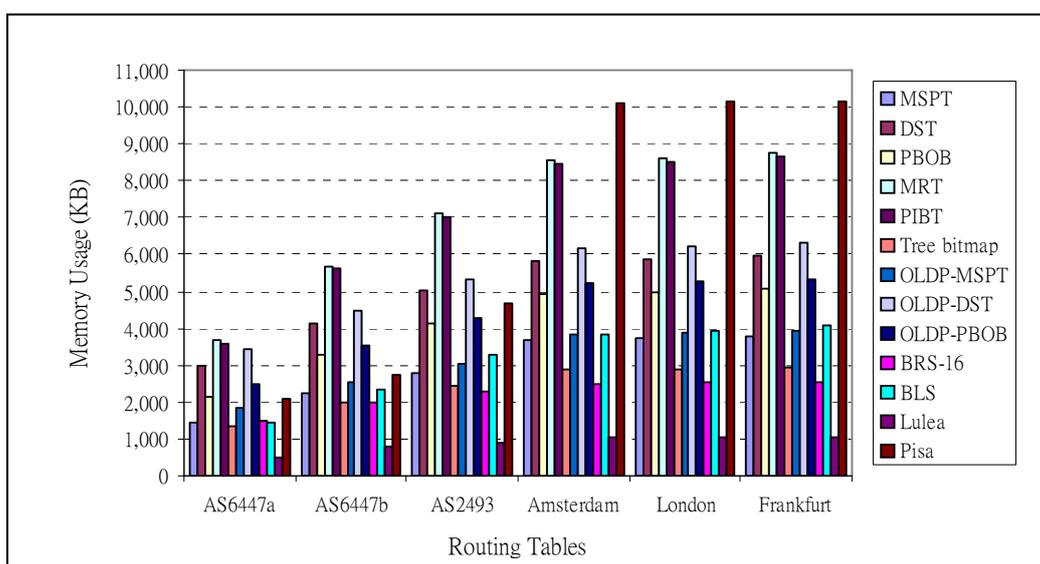


Figure 10. Total memory requirement (in KB) for IPv4.

**Search Time.** To measure the lookup times, we need the simulated IP traffic which is obtained as follows. We first use an array  $A$  to store the address parts of all prefixes in a routing table and then increment them by one. A random permutation of  $A$  is generated and this permutation determines the search order. The time required to determine all the longest prefix matches is measured and averaged over the number of addresses in  $A$ . The experiment is repeated 100 times, and the mean value of these average times is computed. These mean times are reported in Table 3 and in Figure 11.

First, only consider the dynamic schemes. The MRT and PIBT perform the best among all schemes because they have lower tree heights than MSPT, DST, PBOP, and Tree Bitmap.

Table 3. The statistics of search time (in Microsecond) for IPv4.

Routing tables	AS6447a	AS6447b	AS2493	Amsterdam	London	Frankfurt
Dynamic Schemes						
MSPT	0.58	0.72	0.80	0.60	0.58	0.58
DST	0.67	0.78	0.83	0.63	0.63	0.64
PBOB	0.97	1.17	1.28	0.98	1.00	0.98
MRT	0.46	0.53	0.56	0.44	0.45	0.44
PIBT	0.39	0.45	0.48	0.39	0.38	0.38
Tree bitmap	0.57	0.62	0.60	0.54	0.53	0.53
OLDP-MSPT	0.25	0.32	0.34	0.26	0.26	0.27
OLDP-DST	0.28	0.35	0.36	0.28	0.27	0.28
OLDP-PBOB	0.41	0.46	0.48	0.36	0.37	0.36
Static Schemes						
BRS-16	0.25	0.33	0.36	0.35	0.36	0.36
BLS	0.26	0.38	0.44	0.41	0.41	0.43
Lulea	0.22	0.24	0.25	0.24	0.25	0.25
Pisa	0.04	0.04	0.04	0.04	0.04	0.04

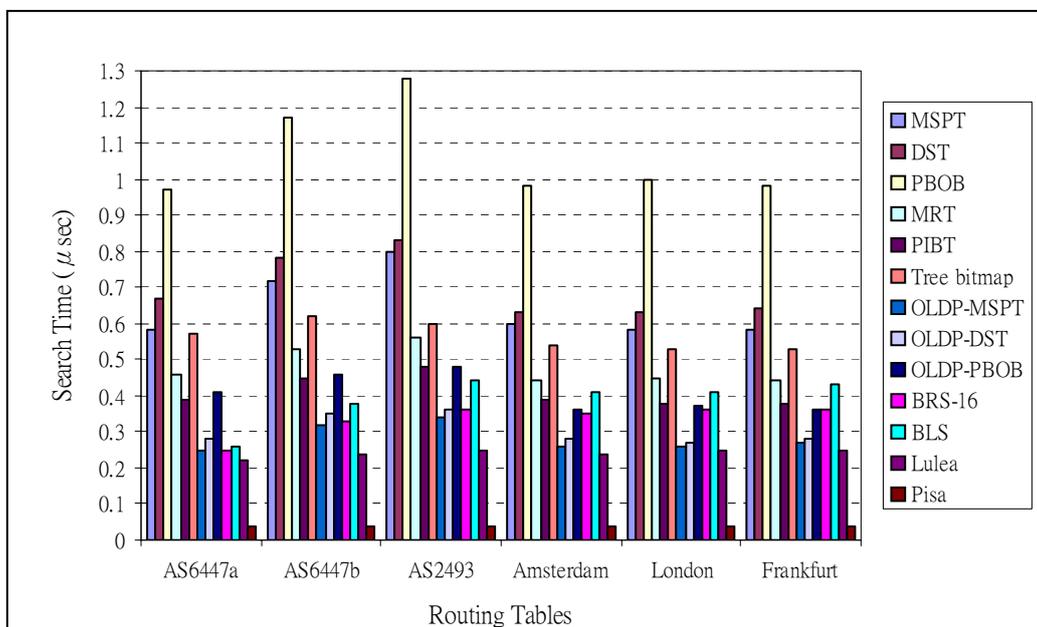


Figure 11. Search time (in Microsecond) for IPv4.

PIBT performs a little better than MRT. The lookup time for MSPT is about 90% of PBOB. This is because each node in MSPT represents a prefix in a routing table and almost all enclosure sets in MSPT are empty. For PBOB, however, the key in a node represents a singleton address. All prefixes in a routing table must be stored in all the range sets of PBOB. As a result, less than 1% range sets are empty. Therefore, the overhead in MSPT to check whether there exist other prefixes also matching the destination address is much less than that of checking the range sets in PBOB. Thus, the MSPT indeed has the better search performance than PBOB. Moreover, MSPT performs a little worse than Tree Bitmap.

Table 4. The statistics of update traces.

Traces	Amsterdam	London	Frankfurt
Period	2006/5/1, 0:00~00:20	2006/5/5, 16:00~16:05	2006/5/6, 08:10~08:15
# of insertion	11,966	15,028	13,780
# of deletion	1,252	672	981
Total updates	13,218	15,700	14,761

However, all the three schemes with OLDP perform much better than Tree Bitmap and OLDP-MSPT performs the best among them.

After adopting OLDP, the number of prefixes in each segment is much smaller than the number of prefixes in original prefix database. The small amount of prefixes in each segment reduces the difference of search performance among all OLDP schemes and also improves the update performance of all the schemes. Therefore, the search performance of OLDP-MSPT is even better than BRS-16 and BLS when the router table sizes are larger than 120,000 prefixes. Also, OLDP-MSPT is a little worse than Lulea and much worse than Pisa.

**Update Time.** For tables Amsterdam, London, and Frankfurt, we use the update traces available in [23]. The statistics of these traces are shown in Table 4. Since the other three tables (AS6447a, AS6447b, and AS2493) have no corresponding BGP insertion and deletion traces, we start by randomly selecting 2000 prefixes from the routing tables. The remaining prefixes are used to build the initial MSPT. After MSPT is constructed, the selected 2000 prefixes are inserted in a random order. Once the 2000 insertions are done, the selected 2000 prefixes are removed from MSPT, also in a random order. The total elapsed time of inserting 2000 prefixes and removing 2000 prefixes is divided by 4000 to get the average update time. This experiment is repeated 100 times and the mean of the average update times is computed.

For static schemes, inserting or deleting a prefix may affects entire data structure. For BRS-16, the update times are obtained by calculating the time of maintaining the 16-bit segmentation table plus the rebuilding time of the corresponding data structure to that segment. As for BLS, the time of creating all makers when inserting a prefix and the time of finding the LPM of all makers are counted. For Lulea, we calculate the time of rebuilding the entire data structure.

Table 5 and Figure 12 show the average update times we measured. The bar charts in Figure 12 do not show the performances of static schemes because their update speeds are too

Table 5. The statistics of update time (in Microsecond) for IPv4.

Routing tables	AS6447a	AS6447b	AS2493	Amsterdam	London	Frankfurt
Dynamic Schemes						
MSPT	0.79	0.81	0.80	1.18	1.21	1.20
DST	0.81	0.82	0.81	1.21	1.24	1.22
PBOB	0.86	0.87	0.86	1.24	1.27	1.26
MRT	1.31	1.34	1.30	1.70	1.75	1.72
PIBT	1.51	1.52	1.52	1.91	1.95	1.93
Tree bitmap	1.68	1.69	1.69	1.94	2.02	1.96
OLDP-MSPT	0.55	0.56	0.55	0.78	0.82	0.78
OLDP-DST	0.60	0.62	0.60	0.82	0.88	0.85
OLDP-PBOB	0.62	0.64	0.64	0.88	0.92	0.88
Static Schemes						
BRS-16	9.66	9.81	9.96	12.00	13.56	14.20
BLS	133	210	254	287	302	321
Lulea	1,455	1,634	1,885	1,998	2,012	2,088

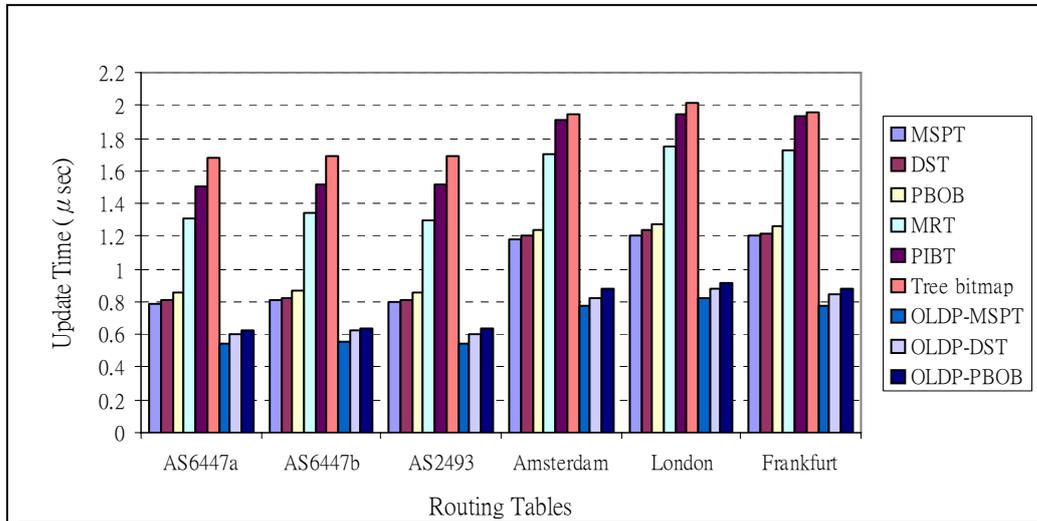


Figure 12. Update time (in Microsecond) for IPv4.

slow compared with the dynamic schemes. The update performance of dynamic schemes is much better than that of static schemes. DST and MSPT have almost the same update performance, and their performances are the best among all schemes. But, when using OLDAP, OLDAP-MSPT has better performance than OLDAP-DST and OLDAP-PBOB. MRT and PIPT perform worse than, MSPT, DST, and PBOB because the insertions or deletions involving rotations in the B-tree structure are slow.

In addition to the average performance, we also show the worst case performance for table AS2493 in terms of 99th percentiles of all measured search and update times in Table 6. The worst case results for other tables are not shown because they have similar trends as table AS2493. We can see that OLDAP-MSPT is the best among all the dynamic schemes.

Table 6. The worse case performance (in Microsecond) for AS2493.

Schemes	search	update
MSPT	1.05	1.13
DST	1.07	1.21
PBOB	1.42	1.24
MRT	0.70	1.82
PIBT	0.65	2.02
Tree bitmap	0.77	2.04
OLDP-MSPT	0.51	0.82
OLDP-DST	0.54	0.84
OLDP-PBOB	0.62	0.84
BRS-16	0.53	13
BLS	0.68	257
Lulea	0.47	1890

**Integrated Performance Analysis.** Since different lookup schemes have their merits in terms of search speed, update speed, or memory consumption, we propose a simple performance model by considering all these three factors together. We analyze the maximum number of lookups ( $N_s$ ) that a lookup scheme can sustain in one second when there are  $N_u$  update messages to be processed in the same one-second period.  $N_u$  can be up to a few hundreds to a few thousands in the real routing environment [12], [13], [20], [28]. By assuming that the search and update operations take  $T_s$  and  $T_u$  microseconds, respectively, we have  $T_s \times N_s + T_u \times N_u = 1,000,000$ . To simplify our analysis, we assume  $N_u = \alpha \times N_s$ . Thus, we have  $N_s = 1,000,000 / (T_s + \alpha \times T_u)$ . To include the memory consumption in the analysis, we treat the maximum number of sustained lookups and the memory size as the performance and cost of a lookup scheme. Thus, we use the performance-cost ratio ( $N_s/Mem$ ) as the metric to compare all the schemes.

Table 7 and Figures 13 and 14 show the comparison results for AS2493 table. Other tables of different sizes show similar results which are not given in this paper. The ideal scheme is assumed to have the best search and update speeds we measured among all the experimented schemes and have the same size of memory as the linear list of routing entries in length format (i.e., 32-bit address/5-bit length/8-bit port). Therefore, the ideal scheme has the performance of  $T_s = 0.04 \mu s$  and  $T_u = 0.55 \mu s$ , and needs 863 KB of memory. The parameter  $\alpha$  is set as 0.05 to 0.001. For the ideal case, it amounts to the update rate of 12k ~ 632k updates per second. Because the performance results of the ideal case are much better

Table 7: Integrated performance comparisons in terms of the maximum number of sustained lookups and the performance-cost ratio for AS2493 table.

$\alpha$	$N_s$				Performance-Cost ratio			
	0.05	0.01	0.005	0.001	0.05	0.01	0.005	0.001
Ideal	12,500,000	20,833,333	22,727,273	24,509,804	14,484	24,141	26,335	28,401
MSPT	1,190,476	1,237,624	1,243,781	1,248,751	424	441	443	445
DST	1,148,765	1,193,175	1,198,969	1,203,645	229	238	239	240
PBOB	755,858	776,036	778,634	780,725	183	188	189	189
MRT	1,600,000	1,745,201	1,765,225	1,781,578	225	246	249	251
PIBT	1,798,561	2,019,386	2,050,861	2,076,757	256	288	292	296
Tree bitmap	1,460,920	1,621,008	1,643,520	1,661,985	598	664	673	680
OLDP-MSPT	2,721,088	2,894,356	2,917,578	2,936,426	899	956	964	970
OLDP-DST	2,564,103	2,732,240	2,754,821	2,773,156	480	512	516	519
OLDP-PBOB	1,953,125	2,055,921	2,069,536	2,080,559	454	478	482	484
BRS-16	1,165,501	2,175,805	2,440,215	2,702,995	505	943	1,057	1,171
BLS	76,104	335,570	584,795	1,440,922	23	102	178	438
Lulea	10,582	52,356	103,359	468,384	12	58	115	523

than the existing schemes, they are truncated at 3,500,000 and 1,500 in Figure 13 and 14, respectively.

OLDP-MSPT performs best among all the dynamic schemes in terms of the maximum number of sustained lookups and the performance-cost ratio. As expected, OLDP-MSPT becomes worse than BRS-16 when  $\alpha$  becomes smaller ( $\alpha < 0.01$ ). However, BRS-16 and all the static schemes enhanced with 16-bit segmentation tables are not suitable for IPv6. Although BRS can be used for IPv6, it does not support dynamic updates because BRS uses a linear array structure.

**IPv6 performance.** To measure the performance for IPv6, we need IPv6 routing tables. Since at present, IPv6 is not as popular as IPv4, current IPv6 table sizes are small and unlikely to reflect future IPv6 network growth. In our experiments, we use two small real IPv6 routing tables (V6table-1 and V6table-2) obtained from IPv6 backbone routers [1]. In addition, we use two large IPv6 routing tables (GV6table-1 and GV6table-2) that are generated based on the generation model proposed in [30].

Although many IPv4 schemes adopt 16-bit segmentation table to speed up the lookup speeds, but for IPv6, the 16-bit segmentation table is no longer suitable because the addresses of IPv6 are 128 bits long. Therefore, we only experiment the dynamic schemes such as

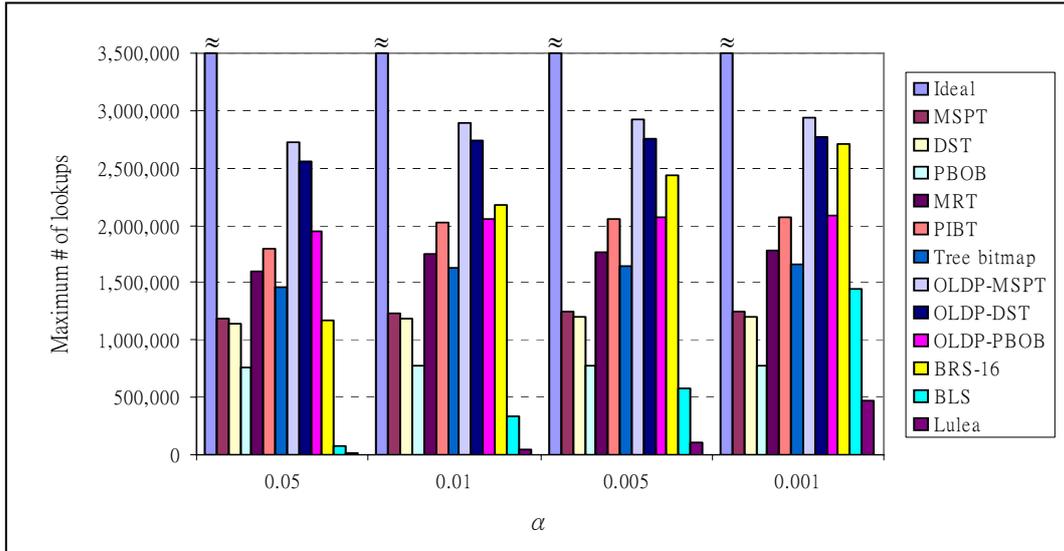


Figure 13. Maximum numbers of sustained lookups.

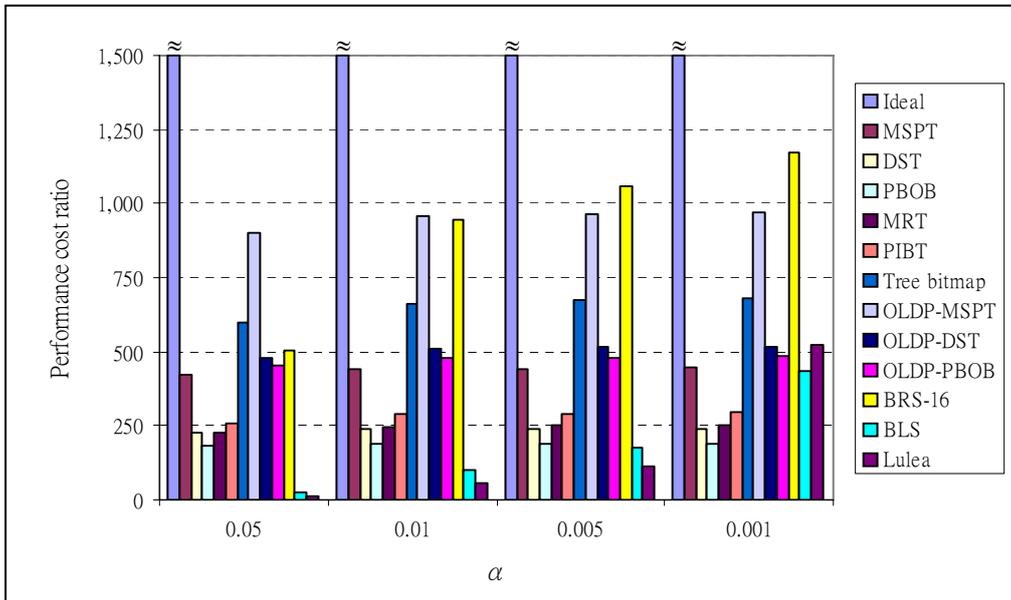


Figure 14. The performance-cost ratios.

MSPT, DST, PBOB and Tree Bitmap. Since all the IPv6 routing tables contain the prefixes of lengths less or equal to 64, the implementation of IPv6 schemes is straightforward by using two 32-bit integers for storing 64-bit keys or prefixes of no more than 64 bits.

Table 8 and Figure 15 show the performance results for all the tested schemes. As expected, Tree Bitmap that is a variant of 4-bit multibit trie performs the worst in search and update speeds because Tree Bitmap grows linearly with the prefix length and thus it does not scale well to longer IP addresses. MSPT consumes much less memory than PBOB for larger tables because, similar to reasons in the IPv4 cases, the number of nodes in MSPT is less than

Table 8. The performance results for IPv6.

IPv6 routing tables		V6table-1	V6table-2	GV6table-1	GV6table-2
# of prefixes		274	593	9,788	20,070
Memory in KB	MSPT	9.9	18.3	312.0	605.0
	DST	15.2	30.2	420.5	840.7
	PBOB	12.5	24.1	386.5	763.1
	Tree bitmap	10.2	15.9	396.0	798.0
Search in $\mu$ s	MSPT	0.25	0.32	0.42	0.53
	DST	0.25	0.33	0.46	0.56
	PBOB	0.26	0.36	0.52	0.75
	Tree bitmap	0.47	0.62	0.73	0.81
Update in $\mu$ s	MSPT	0.56	0.63	0.67	0.73
	DST	0.56	0.66	0.70	0.76
	PBOB	0.59	0.66	0.72	0.79
	Tree bitmap	1.58	1.63	1.67	1.73

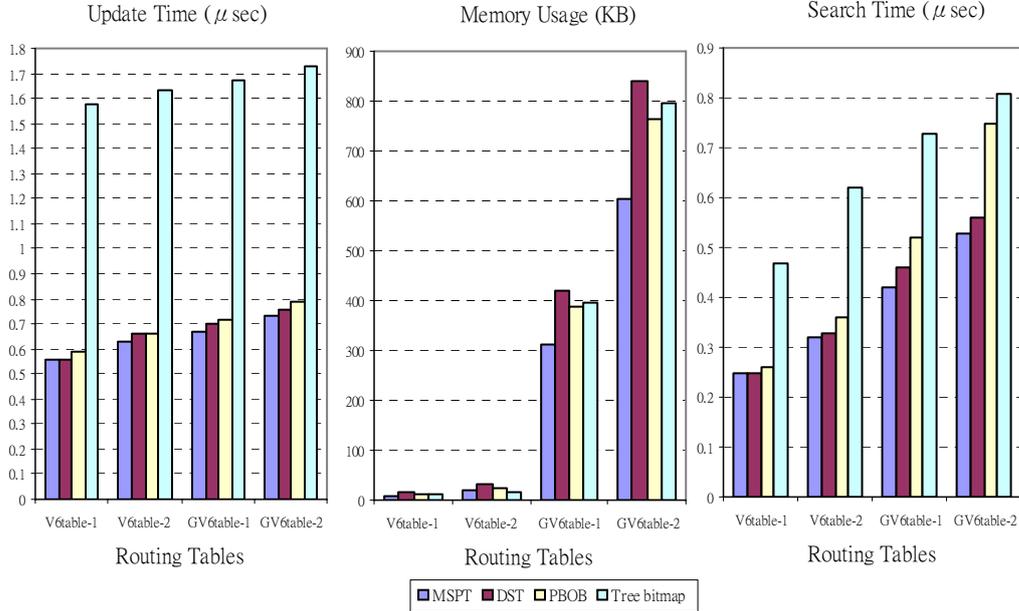


Figure 15. Performance comparisons for IPv6 tables.

that in PBOB and most of the enclosure sets in MSPT are empty. MSPT also performs better than PBOB and DST in search and update speeds. The difference of search performance between MSPT and DST or PBOB increases as routing table size increases. However, the difference of update performance among MSPT, DST, and PBOB is not significant.

## 5 Conclusions

We have developed a new data structure called Most Specific Prefix Tree (MSPT) that is suitable for dynamic routing tables. MSPT is an augmented balanced binary search tree which is constructed by the most specific prefixes in the routing table. The remaining non-

most specific prefixes are placed in the enclosure sets associated with the nodes in MSPT. The proposed search, insertion, and deletion algorithms for MSPT can be finished in  $O(\log M)$  time for real routing tables, where  $M$  is the number of nodes in MSPT. From our experiment results,  $M$  is about 91-93% of the number of prefixes in the routing table.

Comparing with the existing dynamic schemes and several static schemes, our experiments showed that MSPT is superior to all the dynamic schemes in terms of search and update speeds. Our integrated performance analysis shows that MSPT or OLDP-MSPT performs better than all the experimented schemes if the update rate is high. Moreover, MSPT also scales well to IPv6.

## References

- [1] BGP Routing Table Analysis Reports, <http://bgp.potaroo.net/>.
- [2] A. Buchsbaum, G. Fowler, B. Krishnamurthy, K. Vo, and J. Wang, "Fast prefix matching of bounded strings," *ACM Journal of Experimental Algorithmics*, vol. 8, 2003.
- [3] Y. Chang, "Fast binary and multiway prefix searches for packet forwarding," *COMPUTER NETWORKS*, vol. 51, no. 3, pp. 588-605, February 2007.
- [4] Y. Chang and Y. Lin, "Dynamic segment trees for ranges and prefixes," *IEEE Transactions on Computers*, vol. 56, no. 6, pp. 769-784, June 2007.
- [5] P. Crescenzi, L. Dardini, and R. Grossi, "IP address lookup made fast and simple," 7<sup>th</sup> *Annual European Symposium on Algorithms, LNCS*, vol.1643, pp.65-76, 1999.
- [6] S. Deering and R. Hinden, "Internet protocol, version 6 (IPv6) specification," *RFC2460*, December 1998.
- [7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *ACM SIGCOMM*, pp. 3-14, September 1997.
- [8] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: hardware/software IP lookups with incremental updates," *ACM Computer Communications Review*, vol. 34, no. 2, pp. 97-122, April 2004.

- [9] V. Fuller, T. Li, J. Yu and K. Varadhan, "Classless inter-domain routing (CIDR): an address assignment and aggregation strategy," *RFC1519*, September 1993.
- [10] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," *IEEE INFOCOM*, pp. 1240-1247, April 1998.
- [11] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structure in C++*. New York: W.H. Freeman, 1995.
- [12] M. Ichiriu, "High performance layer 3 forwarding - the need for dedicated hardware solutions," White Paper, NetLogic Microsystems, 2000.
- [13] Intel, "IXP2400 Intel Network Processor IPv6 Forwarding Benchmark Full Disclosure Report for Gigabit Ethernet", June 16, 2003.
- [14] IPv6 Forum, <http://www.ipv6forum.com>.
- [15] B. Lampson, V. Srinivasan and G. Varghese, "IP lookups using multiway and multicolumn Search," *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 324-334, June 1999.
- [16] H. Lu, K. Kim, and S. Sahni, "Prefix and interval-partitioned dynamic IP router-tables," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 545-557, May 2005.
- [17] H. Lu, and S. Sahni, " $O(\log n)$  dynamic router-tables for prefixes and ranges," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1217-1230, October 2004.
- [18] H. Lu and S. Sahni, "Enhanced interval tree for dynamic IP router-tables," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1615-1628, December 2004.
- [19] H. Lu and S. Sahni, "A B-tree dynamic router-table design," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 813-824, July 2005.
- [20] Y. Luo, L. Bhuyan, and X. Chen, "Shared memory multiprocessor architectures for software IP routers," *IEEE Transactions on Parallel and Distributed System*, vol. 14, no. 12, pp. 1240-1249, December 2003.
- [21] X. Meng, Z. Xu, B. Zhang, G. Huston, S. Lu, and L. Zhang, "IPv4 address allocation and the BGP routing table evolution," *ACM SIGCOMM*, pp. 71-80, January 2005.

- [22] D. Meyer, University of Oregon Route Views Archive Project, June 2004 (<http://archive.routeviews.org/>).
- [23] RIPE NCC Project at RIPE Coordination Centre: <http://www.ripe.net>.
- [24] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network Magazine*, vol. 15, no. 2, pp. 8-23, March/April 2001.
- [25] S. Sahni and K Kim, "An  $O(\log n)$  dynamic router-table design," *IEEE Transactions on Computers*, Vol. 53, no. 3, pp. 351-363, March 2004.
- [26] K. Sklower, "A Tree-based packet routing table for Berkeley Unix," *Winter Usenix Conference*, pp. 93-99, 1991.
- [27] V. Srinivasan and G. Varghese "Fast Address Lookup Using Controlled Prefix Expansion," *ACM Transactions on Computer Systems*, Vol. 17, No. 1, pp. 1-40, February 1999.
- [28] D. Taylor, J. Turner, W. Lockwood, T. Sproull, and D. Parlour, "Scalable IP lookup for internet routers," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp.522-534, May 2003.
- [29] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable high-speed prefix matching," *ACM Transactions on Computer Systems*, vol. 19, no. 4, pp. 440-482, November 2001.
- [30] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-random generator for IPv6 tables," *12th IEEE Symp. of High Performance Interconnects*, pp. 35-40, August 2004.
- [31] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," *Computer Networks*, vol. 44, no. 3, pp. 289--303, February 2004.