Dynamic Routing Tables Using Simple Balanced Search Trees

Y.-K. Chang and Y.-C. Lin
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan R.O.C.
ykchang@mail.ncku.edu.tw

Abstract. Various schemes for high-performance IP address lookups have been proposed recently. Pre-computations are usually used by the special designed IP address lookup algorithms for better performance in terms of lookup speed and memory requirement. However, the disadvantage of the pre-computation based schemes is that when a single prefix is added or deleted, the entire data structure may need to be rebuilt. Rebuilding the entire data structure seriously affects the lookup performance of a backbone router and thus not suitable for dynamic routing tables.

In this paper, we develop a new dynamic routing table algorithm. The proposed data structure consists of a collection of balanced binary search trees. The search, insertion, and deletion operations can be finished in $O(\log N)$ time, where N is the number of prefixes in a routing table. Comparing with the best existing dynamic routing table algorithm which is PBOB (Prefix Binary tree On Binary tree), our experiment results using the realistic routing tables show that the proposed scheme performs better than PBOB in terms of lookup speed, insertion time, deletion time, and memory requirement.

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

1. Introduction

To handle gigabit-per-second traffic rates, the current backbone routers must be able to forward millions of packets per second at each port. The IP address lookup is the most critical task in the router. When a router receives a packet, the destination address in the packet's header is used to lookup the routing table. There may be more than one route entries in the routing table that match the destination address. Therefore, it may require some comparisons with every route entries to determine which one is the longest matching. The longest route from all the matched entries is called the longest prefix match (LPM). The IP address lookup problem becomes a longest prefix matching problem.

To design a good IP address lookup scheme, we should consider four key aspects: lookup speed, storage requirement, update time and scalability. For any scheme, it is hard to perform well in all aspects. The update process is the concern

of this paper. Currently, the Internet has a peak of a few hundred BGP updates per second. Thus, the address lookup schemes with fast update time are desirable to avoid routing instabilities. These updates should interfere little with normal address lookup operation.

Various algorithms for high-performance IP address lookup have been proposed. In the survey paper [10], a large variety of routing lookup algorithms are classified and their complexities of worst case lookup, update, and memory references are compared. Despite the intense research that has been conducted in recent years, there should be a balance between lookup speed, memory requirement, update, and scalability for a good IP address lookup scheme. The pre-computation [2], [3], [5], [8], [11] perform a lot of pre-computation and thus improve the performance of the lookup speed and memory requirement. However, a disadvantage of the pre-computation is that when a single prefix is added or deleted, the entire data structure may need to be rebuilt. Rebuilding the routing tables seriously affects the update performance of a backbone router. Thus, the schemes based on pre-computation are not suitable for dynamic routing tables. On the other hand, schemes based on the trie data structure like binary trie, multi-bit trie and Patricia trie [9] do not use pre-computation; however, their performances grow linearly with the address length, and thus the scalability of these schemes is not good when switching to IPv6 or large routing table.

Although schemes like [4], [6], [12] develop a search tree data structure that is suitable for the representation of dynamic routing tables, the complex data structure leads to the memory requirement expanded and reduce the performance of lookup. Sahni and Kim [4] developed a data structure, called a collection of red-black tree (CRBT), that supports three operations for dynamic routing table of N prefixes (longest prefix match, prefix insert, prefix delete) in $O(\log N)$ time each. In [6], 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. On practical routing tables, LMPBOB and PBOB permit longest prefix matching in $O(W)$ and $O(\log N)$, where W is 32 for IPv4 or 128 for IPv6. For the insertion and delete operations, they both take $O(\log N)$ time. Suri et al. [12] have proposed a B-tree data structure called multiway range tree. This scheme achieves the optimal lookup time of binary search, but also can be updated in logarithmic time when a prefix is inserted or deleted.

In this paper, we develop a data structure based on a collection of independent balanced search trees. Unlike the augmented data structures proposed in the literature, the proposed scheme can be implemented with any balanced tree algorithm without any modification. As a result, the proposed data structure is simple and has a better performance than PBOB we compared.

The rest of the paper is organized as follows. Section 2 presents a simple analysis for the routing tables. Section 3 illustrates proposed scheme based on the analysis in section 2 and the detailed algorithms. The results of performance comparisons using real routing tables are presented in section 4. Finally, a concluding remark is given in the last section.

Table 1: Prefix enclosure analysis for three realistic routing tables.

Database (year-mouth)	AS6447 (2000-4)	AS6447 (2002-4)	AS6447 (2005-4)
number of prefixes	79530	124798	163535
Level-1 prefixes	73891(92.9%)	114745 (91.9%)	150245 (91.9%)
Level-2 prefixes	4874 (6.1%)	8496 (6.8%)	11135 (6.8%)
Level-3 prefixes	642 (0.8%)	1290 (1%)	1775 (1.1%)
Level-4 prefixes	104 (0.1%)	235 (0.2%)	329 (0.2%)
Level-5 prefixes	17	29	45
Level-6 prefixes	2	3	6

2. Analysis of Covering and Covered Prefixes

The Border Gateway Protocol (BGP) is the de facto standard inter-domain routing protocol in the Internet. BGP provides loop-free inter-domain routing between autonomous systems, each consisting of a set of routers that operate under the same administration. The address space represented by an advertised BGP prefix may be a sub-block of another existing prefix. The former is called a *covered* prefix and the latter a *covering* prefix. For example, the address block 140.116.82.0/24 is covered by another address block 140.116.0.0/16.

We analyzed three BGP routing tables obtained from [1], and obtained the detailed statistics for the enclosure relationship between the covered and covering prefixes. Theoretically, one prefix may be covered by at most 31 prefixes for IPv4. The prefix and the ones that cover it form a prefix enclosure chain. Therefore, the theoretical worse-case enclosure chain size is 32 for IPv4. Contrary to the definition in [7], we number the prefixes in a bottom-up manner. For example, if a prefix enclosure chain consists of five prefixes Pi for i = 5 to 1, where P5 is the shortest prefix that covers the other four prefixes and P1 is the longest one that is covered by the other four prefixes. The prefixes like P1 that do not cover any other prefix in the routing table are called the *level-1* prefixes. The prefixes that only contain level-1 prefixes are called level-2 prefixes, and so on. Figure 1 shows the enclosure relationship between covering and covered prefixes marked with their levels for an example routing table that has the enclosure chain size of 5. Our analysis shows that the chain size is 6 for all the tables we examined. We further show the number of prefixes in each level for all the three routing tables in Table 1. The level-1 prefixes account for about 92% ~ 93% of the prefixes in a routing table. The level-2 prefixes account for about 6% ~ 7% of the prefixes. The prefixes in other levels only account for less than 1% of the total prefixes. Since the prefixes in each level are disjoint, it is straightforward to design dynamic routing lookup algorithms with search and update complexity of $O(\log N)$ for a routing table consisting of N prefixes.

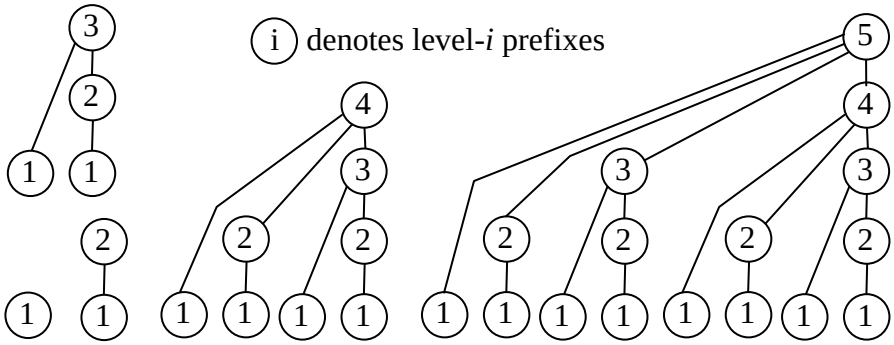

Figure 1: Enclosure relationship between covering and covered prefixes, assuming the maximum size of a prefix enclosure chain is 5.

3. The Proposed Scheme

From Figure 1, two important properties of the prefix enclosure relationship can be obtained. The first property is that all the prefixes in one level are disjoint. The second property is that the prefix containing any one of the level-*i* prefixes must be stored in level-(*i*+1) or in higher level. Therefore, if we can find a prefix match in level-*i* prefixes, no search is needed in *i*+1 or higher level. Assume there are at most *s* levels in the routing table. Based on the above two properties, we can build *s* independent data structures for the lookup problem by obeying the *tree level constraint* as follows.

Tree Level Constraint: Based on the enclosure relationship between prefixes, the level-*i* prefixes are stored in the level-*i* data structure.

Therefore, for a destination address *d*, if a prefix in the level-1 data structure is found to match *d*, it must be the only matching prefix in the level-1 data structure. Moreover, this matching prefix must be the longest prefix match. Other prefixes that also match *d* must be in the higher level data structures. As a result, the higher level data structures do not need to be searched. Furthermore, if the level-1 data structure does not contain a prefix that matches *d*, we perform the same search process in the level-2 data structure. If a matching prefix is found, it must be the longest prefix match. No other higher level data structure needs to be searched. This search process continues until the level-*s* data structure is searched, where *s* is the maximum number of levels.

```
Algorithm Search(d, root[], s)
{ // d is the destination address, s is the number of trees
01    for ( i = 1 ; i <= s ; i++ ) {
02          x = root[i];
03          while ( x ≠ NULL ) {
04                if ( x.prefix ⊇ d ) return x.prefix; //A ⊇ B denotes A covers B
05                else
06                      if ( d < x ) x = x.LeftChild;
07                      else x = x.RightChild;
08          } //end while
09    } //end for
10    return default_prefix
}
```

Figure 2: Algorithm to find the longest prefix match.

```
Algorithm Insert (P, root[], s)
{ // P is the newly added prefix, s is the number of trees
01    for ( i = 1 ; i <= s ; i++ ) {
02          x = root[i];
03          while ( x ≠ NULL ) {
04                if ( P = x.prefix ) return;
05                if ( P ⊆ x.prefix ) {      /* x.prefix encloses prefix P */
06                      Q = x.prefix; x.prefix = P; P = Q; break; }
07                if ( x.prefix ⊆ P ) break;
08                if ( P > x.prefix )
09                      if ( x.RightChild = NULL ) {
10                            x.RightChild = Create_A_Node(P);
11                            BST_Balancing(root[i], x.RightChild); return;
12                      } else x = x.RightChild;
13                else
14                      if ( x.LeftChild = NULL ) {
15                            x.LeftChild = Create_A_Node(P);
16                            BST_Balancing(root[i], x.LeftChild); return;
17                      } else x = x.LeftChild;
18          } //end while
19    } //end for
20    root[++s] = Create_A_Node(P); // The level is increased by one
}
```

Figure 3: Algorithm to insert a prefix.

If the enclosure relationship between prefixes is changed because of insertion or deletion, the locations of some of the prefixes must also be adjusted in order to follow the tree level constraint. In this paper, we decide to use a balanced binary search tree to implement each level of prefixes. Other data structures will be considered in the future. Since the number of levels is a constant and each level is implemented as a balanced binary search tree, the search time complexity must be $O(\log N)$ for a routing table of N prefixes. Figure 2 shows the search algorithm *Search*(*d*, *root*[], *s*), where parameter *d* is the destination address and there are *s* balanced binary search trees.

The insertion of a prefix *P* is done by performing tree traversals from the level-1 tree to the level-*s* tree. The main task when traversing the trees is to check if there exists a prefix that covers *P* or is covered by *P*. If no such prefix is found, then *P* is disjoint from all the prefixes in the level-1 tree. And thus, *P* is inserted as a leaf node in the level-1 tree. A possible rotation of balanced binary search trees is needed after *P* is inserted. However, if a prefix *Q* in the level-1 tree is found to cover *P*, then *Q* is replaced by *P* and the process of inserting *Q* in the level-2 tree is performed. If a prefix *Q* is found to be covered by *P*, the process of inserting *P* in the level-2 tree is performed. In other words, the same insertion process repeats for trees of level-2 to level-*s*, where *s* is the number of trees before inserting a prefix. If *P* covers all the prefixes in the routing table, a new tree at level *s*+1 will be generated.

Figure 3 shows the insertion algorithm *Insert*(*P*, *root*[], *s*) that inserts a prefix *P* in the balanced binary search trees rooted at *root*[*1..s*]. After *P* is inserted in one of the balanced binary search trees, a possible balancing operation (function *BST_Balancing*()) must be performed.

The deletion process of deleting a prefix *D* first finds out which tree contains *D* among *s* balanced binary search trees, assume *D* is in the level-*i* tree. There may be a prefix *Q* that covers *D* in the level-(*i*+1) tree. If prefix *D* is the only prefix that is covered by *Q* in the level-*i* tree, then the tree level constraint will be violated after deleting *D* from the level-*i* tree. Therefore, in this case, prefix *Q* must be moved from the level-(*i*+1) tree and inserted into the level-*i* tree (i.e., use *Q* to replace *D*). The violation of tree level constraint may cause a chain effect to higher level trees. On the other hand, if prefix *D* is not the only prefix covered by *Q* in the level-*i* tree, then anything other than deleting prefix *D* is not required.

The process of checking whether or not prefix *D* is the only prefix covered by prefix *Q* may have a strong impact on the overall process time for deletion. Therefore, we propose an efficient scheme to minimize the time taken for this process. This scheme only checks if prefix *Q* covers the prefixes *Y* and *Z* that are the smallest prefix in *D*'s right subtree and the largest prefix in *D*'s left subtree, respectively. To explain why we don't need to examine other prefix, it is sufficient to consider *Y* only as follows. If there is another prefix *U* that is also covered by prefix *Q*, then *Q* must also cover *Y* because *Y* locates between *D* and *U*. One may argue that the faster way to know if there exists another prefix that is also covered by *Q* in the level-*i* tree is to examine the prefixes on the path from the node associated *D* to the node associated *Y* one-by-one while traversing the tree and stop as soon as we find another prefix is covered by *Q*. But we should know that even we find a prefix *T* covered by *Q* earlier than reaching the node associated *Y*, the node associated with prefix *Y* still needs to be visited because *Y* can be used to replace *D*. Therefore, the best way is directly go to

the node associated with Y from the node associated with D and checking if Y is covered by Q. Also notice that it is possible that Q is at level-k for $k \geq i+2$ (i.e., no prefixes in the level-m tree cover D, $m = i+1$ to $k-1$). In this case, there must be a prefix enclosure chain for D consisting of a prefix in each level-j for $j = k$ to i. Thus, prefix Q must remain in the level-k tree because of the prefix enclosure chain for D.

Figure 4 shows the details of the deletion algorithm. The while loop search for the prefix D in each tree rooted at $root[i]$ for $i = 1$ to s. When a node x associated with D is found in level-i tree, the function *Search_a_Tree_for_Enclosure*($root[i+1]$, D) as shown in line 5 is performed to find a prefix Q that contains D in the level-(i+1) tree. If such Q does not exist, we delete node x from the level-i tree directly by using the standard balanced binary search tree deletion algorithm as shown in line 6. As explained above, we don't worry about if a prefix containing prefix D exists in the higher level tree than i. Lines 7-17 take care when a prefix Q that contains D exists in the level-(i+1) tree. If the right subtree of node x is not empty, it must exist a node y which is the x's successor. Otherwise, x's successor is the node already visited and recorded in line 21. If y exists and $y.prefix$ is contained in Q, we replace x with y and delete node y directly, as shown in lines 10-11. Similar operations are done for the largest prefix in the left subtree of node x.

```
Algorithm Delete(D, root[], s)
{ // y and z are the successor and predecessor of node x containing prefix D
01  for ( i = 1 ; i ≤ s ; i++ ) {
02      x = root[i]; y = z = NULL;
03      while ( x ≠ NULL ) {
04          if ( x.prefix = D ) {
05              q = Search_a_Tree_for_Enclosure(root[i+1], D);
06              if ( q = NULL ) { BST_Delete(root[i], x); return; }
07              else {
08                  if ( x.RightChild ≠ NULL ) y = Smallest_Prefix(x.RightChild);
09                  if ( y ≠ NULL and q.prefix ⊇ y.prefix ) {
10                      x.prefix = y.prefix;
11                      BST_Delete(root[i], y); return; }
12                  if ( x.LeftChild ≠ NULL ) z = Largest_Prefix(x.LeftChild);
13                  if ( z ≠ NULL and q.prefix ⊇ z.prefix ) {
14                      x.prefix = z.prefix;
15                      BST_Delete(root[i], z); return; }
16                  x.prefix = q.prefix; D = q.prefix; break;
17              }
18          }
19          if ( D ⊃ x.prefix ) break; // D contains x.prefix and break inner loop
20          if ( D ⊂ x.prefix ) return; // D does not exist
21          if ( D < x ) { y = x; x = x.LeftChild; }
22          else { z = x; x = x.RightChild; }
23      } //end while
24  } //end for
}
```

Figure 4: Algorithm to delete a prefix.

4. Performance Evaluations

In this section, we present the performance results for IPv4 routing tables. Three BGP tables of different sizes obtained from [1] are used in our experiments. These BGP routing tables reflect the realistic sizes of the routing tables in the backbone routers currently deployed on the Internet. We compare the proposed algorithm with the prefix binary tree on the binary tree structure (PBOB) [6]. We only choose PBOB for comparisons because other schemes [4], [12] do not perform better than PBOB. The performance experiments are implemented in C language on a Linux Redhat platform with a 2.4G Pentium IV processor containing 8KB L1, 256KB L2 caches and 768MB main memory. Moreover, GNU gcc-3.2.2 compiler with optimization level –O4 is used.

Table 2 (a) shows the amount of memory used by each of the tested schemes. We can see that the proposed scheme uses about 15% less memory than the PBOB structure. This result can be attributed to that the node structure of our scheme is much simpler than that of PBOB. Besides, each node of the PBOB structure is associated a prefix set, and less than 1% of these prefix sets are empty. For every PBOB nodes that associate the non-empty prefix sets, it needs additional memory to store these non-empty prefix sets (each non-empty prefix set is constructed by an array structure with six entries). To measure the lookup times, we first use an array *A* to store the address parts of all prefixes in a routing table and then randomize them to obtain the input query address sequence. The time required to determine all the LPMs is measured and averaged over the number of addresses in *A*. The experiment is repeated 100 times, and the mean of these average times is computed. These mean times are reported in Table 2 (b). Although the worst case search time may be worse than that in PBOB because all the balanced binary trees must be searched, the average time is better than PBOB. This is because most of the search result can be determined in the level-1 tree. For the average update (insert/delete) time, we start by randomly selecting 5% of the prefixes from the routing tables. The remaining prefixes are used to build the desired data structures (PBOB and the proposed balanced binary search trees). After the desired data structure is constructed, the 5% selected prefixes are inserted into the structure one by one. Once the selected prefixes are all already inserted, we proceed to remove them from the constructed structure one by one. The total elapsed insertion and deletion times are averaged to get the average insertion and deletion times. This experiment is also repeated 100 times and the mean of the average times is reported in Table 2 (c) and (d). The deletion times for PBOB are obtained by the implementation with the optimized version of the deletion algorithm proposed in [6]. In other words, the empty nodes in PBOB are not removed if they have two children nodes. However, in the proposed scheme, we implement the complete deletion procedure such that as long as a prefix is deleted, the corresponding node in one of the balanced trees is removed and the required rotations are also performed. Even with this implementation difference, the deletion time of the proposed scheme still performs better than PBOB.

5. Conclusions

We have developed a dynamic routing table data structure based on the prefix enclosure relationship structure. For currently available backbone routing tables, at most six independent balanced binary search trees are needed. Since the level-1 tree

and the level-2 tree account for 97%-99% prefixes in the routing table, the average performance of the lookup, insertion, and deletion times are very well. Since the number of the balanced tree is constant, the search, insertion, and deletion operations can be finished in $O(\log N)$ time, where N is the number of prefixes. Our experiment results show that the proposed scheme performs better than PBOB, the best dynamic routing table algorithm, in terms of lookup speed, insertion time, deletion time, and memory requirement.

Table 2: Performance statistics.

(a) Memory requirements (KB)

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	1,525	2,374	3,101
Proposed scheme	1,330	2,087	2,734

(b) Average search time (microseconds)

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	1.02	1.37	1.57
Proposed scheme	0.65	0.79	0.88

(c) Average insertion time (microseconds)

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	0.90	0.89	1.01
Proposed scheme	0.71	0.75	0.76

(d) Average deletion time (microseconds)

schemes	AS6447 (79,560)	AS6447 (124,824)	AS6447 (163,574)
PBOB	0.57	0.57	0.64
Proposed scheme	0.47	0.48	0.49

References

[1] BGP Routing Table Analysis Reports, http://bgp.potaroo.net/.

[2] A. Brodnik, S. Carlsson, M. Degermark, S. Pink, "Small Forwarding Tables for Fast Routing Lookups," ACM SIGCOMM, pp. 3-14, Sept. 1997.

[3] N. F. Huang, S. M. Zhao, J. Y. Pan, and C. A. Su, "A Fast IP Routing Lookup Scheme for Gigabit Switching Routers," in Proc. INFOCOM, pp. 1429-1436, Mar. 1999.

[4] K. Kim, S Sahni, "An O(logn) Dynamic Router-Table Design," IEEE Transactions on Computers, pp. 351-363, Mar. 2004.

[5] 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, Jun.1999.

[6] H. Lu, S. Sahni, "Enhanced Interval Tree for Dynamic IP Router-Tables," IEEE Transactions on Computers, pp. 1615-1628, Dec. 2004.

[7] X. Meng, Z. Xu, B. Zhang, G.. Huston, S. Lu, L. Zhang, "IPv4 Address Allocation and the BGP Routing Table Evolution," ACM SIGCOMM, pp. 71-80, Jan. 2005.

[8] S. Nilsson and G. Karlsson "IP-Address Lookup Using LC-trie," IEEE Journal on selected Areas in Communications, 17(6):1083-1092, June 1999.

[9] K. Sklower, "A Tree-based Packet Routing Table for Berkeley Unix," Proc. Winter Usenix Conf, pp. 93-99, 1991.

[10] M. A. Ruiz-Sanchez, Ernst W. Biersack, and Walid Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE Network Magazine, 15(2):8--23, March/April 2001.

[11] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable High-Speed IP Routing Lookups," ACM SIGCOMM, pp. 25-36, Sept. 1997.

[12] P. Warkhede, S. Suri, G.. Varghese, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," The International Journal of Computer and Telecommunications Networking, pp. 289-303, Feb. 2004.