An $O(\log n)$ Dynamic Router-Table Design [*]

Sartaj Sahni & Kun Suk Kim
{sahni, kskim}@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

Internet (IP) packet forwarding is typically done by finding the longest prefix in a router table that matches the packet's destination address. For W-bit destination addresses, the use of binary tries enables us to determine the longest matching prefix in $O(W)$ time, independent of the number n of prefixes in the router table. New prefixes may be inserted and old ones deleted in $O(W)$ time also. Since $n << 2^W$ in real router tables, it is desirable to develop a data structure that permits longest prefix matching as well as the insertion and deletion of prefixes in $O(\log n)$. These three operations can be done with $O(\log n)$ cache misses using a B-tree data structure [19]. However, the run-time (including operation cost and cost of cache misses) is not $O(\log n)$. In this paper we develop a data structure in which prefix matching, prefix insertion, and deletion can each be done in $O(\log n)$ time. Experiments using real IPv4 routing databases indicate that although the proposed data structure is slower than optimized variable-stride tries for longest prefix matching, the proposed data structure is considerably faster for the insert and delete operations.

Keywords: Packet routing, longest matching prefix, red-black trees.

1 Introduction

An Internet router table is a set of tuples of the form (p, a), where p is a binary string whose length is at most W ($W = 32$ for IPv4 destination addresses and $W = 128$ for IPv6), and a is an output link (or next hop). When a packet with destination address A arrives at a router, we are to find the pair (p, a) in the router table for which p is a longest matching prefix of A (i.e., p is a prefix of A and there is no longer prefix q of A such that (q, b) is in the table). Once this pair is determined, the packet is sent to ouput link a. The speed at which the router can route packets is limited by the time it takes to perform this table lookup for each packet.

Longest prefix routing is used because this results in smaller and more manageable router tables. It is impractical for a router table to contain an entry for each of the possible destination addresses. Two of the reasons this is so are (1) the number of such entries would be almost one hundred million and would triple every three years, and (2) every time a new host comes online, all router tables will need to

[*]This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

incorporate the new host's address. By using longest prefix routing, the size of router tables is contained to a reasonable quantity and information about host/router changes made in one part of the Internet need not be propagated throughout the Internet.

Several solutions for the IP lookup problem (i.e., finding the longest matching prefix) have been proposed. IP lookup in the BSD kernel is done using the Patricia data structure [17], which is a variant of a compressed binary trie [8]. This scheme requires $O(W)$ memory accesses per lookup. We note that the lookup complexity of longest prefix matching algorithms is generally measured by the number of accesses made to main memory (equivalently, the number of cache misses). Dynamic prefix tries, which are an extension of the Patricia data structure, and which also take $O(W)$ memory accesses for lookup, have been proposed by Doeringer et al. [5]. LC tries for longest prefix matching are developed in [13]. Degermark et al. [4] have proposed a three-level tree structure for the routing table. Using this structure, IPv4 lookups require at most 12 memory accesses. The data structure of [4], called the Lulea scheme, is essentially a three-level fixed-stride trie in which trie nodes are compressed using a bitmap. The multibit trie data structures of Srinivasan and Varghese [18] are, perhaps, the most flexible and effective trie structure for IP lookup. Using a technique called controlled prefix expansion, which is very similar to the technique used in [4], tries of a predetermined height (and hence with a predetermined number of memory accesses per lookup) may be constructed for any prefix set. Srinivasan and Vargese [18] have developed dynamic programming algorithms to obtain space optimal fixed-stride and variable-stride tries of a given height. Improved algorithms to construct optimal multibit tries appear in [14, 15].

Waldvogel et al. [20] have proposed a scheme that performs a binary search on hash tables organized by prefix length. Using this binary search scheme, we can perform longest prefix matching in $O(\log W)$ expected time. The expected insert and delete time is $O(n \log^2 W)$. The basic hash-table structure of [20] may be modified so that the expected time for longest-prefix matching is $O(\alpha + \log W)$ and the expected insert/delete time is $O(\alpha \sqrt[\alpha]{n} W \log W)$, for any $\alpha > 1$ [20]. An alternative adaptation of binary search to longest prefix matching is developed in [9]. Using this adaptation, a lookup in a table that has n prefixes takes $O(W + \log n)$ time.

Cheung and McCanne [3] have developed "a model for table-driven route lookup and cast the table design problem as an optimization problem within this model." Their model accounts for the memory hierarchy of modern computers and they optimize average performance rather than worst-case performance.

Hardware solutions that involve the use of content addressable memory [10] as well as solutions that involve modifications to the Internet Protocol (i.e., the addition of information to each packet) have also been proposed [2, 12, 1].

Gupta and McKeown [7] examine the asymptotic complexity of a related problem, packet classification. They develop two data structures, heap-on-trie (HoT) and binary-search-tree-on-trie (BoT), for the dynamic packet classification problem. The complexity of these data structures (for packet classification and the insertion and deletion of rules) also is dependent on W. For d-dimensional rules, a search in a HoT takes $O(W^d)$ and an update (insert or delete) takes $O(W^d \log n)$ time. The corresponding times for a BoT are $O(W^d \log n)$ and $O(W^{d-1} \log n)$, respectively.

Lampson et al. [9] have proposed a binary search scheme in which prefixes are encoded as ranges. Even though this scheme permits one to determine the longest matching prefix in $O(\log n)$ time, inserts and deletes take $O(n)$ time. In fact, for their scheme, they state that "there does not appear to be any update technique that is faster than just building the table from scratch." Ergun et al. [6] use ranges to develop a biased skip list structure that performs longest prefix matching in $O(\log n)$ time. Their scheme is designed to give good expected performance for "bursty access patterns". The biased skip list scheme of Ergun et al. [6] permits inserts and deletes in $O(\log n)$ time only in the severely restricted and impractical situation when all prefixes in the router table are of the same length. For the more general, and practical, case when the router table comprises prefixes of different length, their scheme takes $O(n)$ expected time for each insert and delete. In this paper, we show how to use the range encoding idea of [9] so that longest prefix matching as well as prefix insertion and deletion can be done in $O(\log n)$ time.

The simplest efficient data structure for a dynamic router-table is a a compressed binary trie [8]. Using a compressed binary trie, longest-prefix matching and prefix insertion and deletion take $O(W)$ worst-case time each.

Suri et al. [19] have proposed a B-tree data structure for dynamic router tables. Using their structure, we may find $LMP(d)$ in $O(\log n)$ time. However, inserts/deletes take $O(W \log n)$ time. The number of cache misses is $O(\log n)$ for each operation. When W bits fit in $O(1)$ words (as is the case for IPv4 and IPv6 prefixes) logical operations on W-bit vectors can be done in $O(1)$ time each. In this case, the scheme of [19] takes $O(\log W * \log n)$ time for an insert and $O(W + \log n) = O(W)$ time for an update.

Despite the intense research that has been conducted in recent years, there is no known way to peform longest prefix matches as well as insertion and deletion of prefixes in $O(\log n)$ time.

In Section 2, we describe the range encoding technique of [9]. We establish a few properties of ranges that represent prefixes in Section 3. Our $O(\log n)$ method is described in Section 4. In Section 5, we

Prefix Name	Prefix	Range Start	Range Finish
P1	*	0	31
P2	0101*	10	11
P3	100*	16	19
P4	1001*	18	19
P5	10111	23	23

Figure 1: Prefixes and their ranges

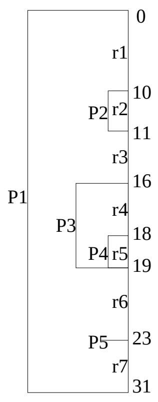
present our experimental results. These results, obtained using real IPv4 prefix databases, indicate that the $O(\log n)$ method proposed in this paper represents a good alternative to existing methods in environments where there is a significant number of insert and/or delete opeations. For example, our method takes more time to find the longest matching prefix than do the variable-stride tries of [18]. However, although these tries are optimized for longest matching prefix searches, they perform very poorly when it comes to insertion and deletion of prefixes. Our proposed method handily outperforms variable-stride tries on these latter operations.

2 Prefixes And Ranges

Lampson, Srinivasan, and Varghese [9] have proposed a binary search scheme for longest prefix matching. In this scheme, each prefix is represented as a range $[s, f]$, where s is the start of the range for the prefix and f is the finish of the range for that prefix. For example, when $W = 5$, the prefix $P = 1*$ matches all destination addresses in the range [10000, 11111] = [16, 31]. So, for prefix P, $s = 16$ and $f = 31$. Figure 1 shows a set of five prefixes together with the start and finish of the range for each. This figure assumes that $W = 5$. The prefix P1 = *, which matches all legal destination addresses, is called the *default* prefix. Although a real router database may not include the default prefix, *we assume throughout this paper that this prefix is always present.* This assumption does not, in any way, affect the validity of our work as we may simply augment router databases that do not include the default prefix with a default prefix whose next hop field is null.

Prefixes and their ranges may be drawn as nested rectangles as in Figure 2(a), which gives the pictorial representation of the five prefixes of Figure 1.

Lampson et al. [9] propose the construction of a table of distinct range end-points such as the one shown in Figure 2(b). The distinct end points (range start and finish points) for the prefixes of Figure 1 are [0, 10, 11, 16, 18, 19, 23, 31]. Let r_i, $1 \leq i \leq q \leq 2n$ be the q distinct range-end-points for a set of n prefixes. Let $r_{q+1} = \infty$. Let $LMP(d)$ be the longest matching prefix for the destination address



(a)

End Point	$>$	$=$
0	P1	P1
10	P2	P2
11	P1	P2
16	P3	P3
18	P4	P4
19	P1	P4
23	P1	P5
31	-	P1

(b)

Figure 2: (a) Pictorial representation of prefixes and ranges (b) Table for binary search

d. With each distinct range end-point, r_i, $1 \leq i \leq q$, the table stores the longest matching prefix for destination addresses d such that (a) $r_i < d < r_{i+1}$ (this is the column labeled "$>$" in Figure 2(b)) and (b) $r_i = d$ (column labeled "$=$"). Now, $LMP(d)$, $r_1 \leq d \leq r_q$ can be determined in $O(\log n)$ time by performing a binary search to find the unique i such that $r_i \leq d < r_{i+1}$. If $r_i = d$, $LMP(d)$ is given by the "$=$" entry; otherwise, it is given by the "$>$" entry. For example, since $d = 20$ satisfies $19 \leq d < 23$ and since $d \neq 19$, the "$>$" entry of the end point 19 is used to determine that $LMP(20)$ is P1. As noted by Lampson et al. [9], the range end-point table can be built in $O(n)$ time (this assumes that the end points are available in ascending order). Unfortunately, as stated in [9], updating the range end-point table following the insertion or deletion of a prefix also takes $O(n)$ time because $O(n)$ "$>$" and/or "$=$" entries may change. Although Lampson et al. [9] provide ways to reduce the complexity of the search for the LMP by a constant factor, these methods do not result in schemes that permit prefix insertion and deletion in $O(\log n)$ time.

3 Properties Of Prefix Ranges

The *length*, $length(P)$, of a prefix P is the number of zeroes and ones in the binary representation of the prefix. For example, P1 of Figure 1 has a length of 0 and $length(\text{P4}) = 4$. W is the number of bits in a destination address. Hence, the number of bits in the start and finish points of a prefix also is W. $P = [s, f]$ is a *trivial prefix* iff $length(P) = W$ (equivalently, iff $s = f$). P is a *nontrivial prefix* iff $length(P) < W$ (equivalently, iff $s \neq f$). Prefixes P1–P4 of Figure 1 are nontrivial while P5 is a trivial prefix. Let $lsb(x)$ be the least significant bit in the binary representation of x. For example, $lsb(32) = 0$ and $\text{lsb}(3) = 1$.

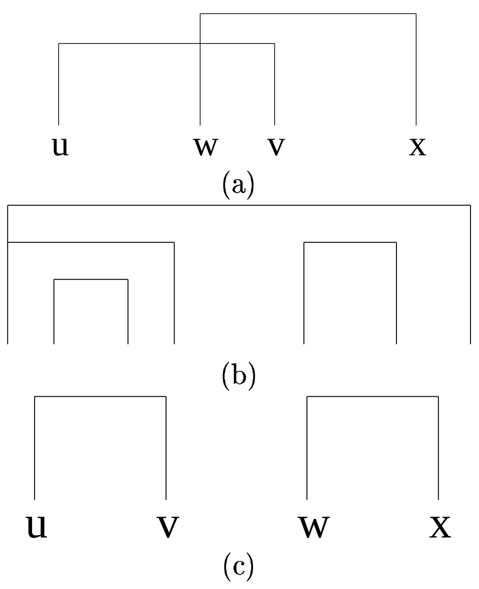


Figure 3: (a) Intersecting ranges (b) Nested ranges (c) Disjoint ranges

Lemma 1 *If $P = [s, f]$ is a nontrivial prefix, then $lsb(s) = 0$ and $lsb(f) = 1$.*

Proof Since P is nontrivial, $length(P) < W$. Therefore, s is the bits of P followed by $W - length(P) > 0$ zeroes and f is the bits of P followed by $W - length(P) > 0$ ones. Consequently, $lsb(s) = 0$ and $lsb(f) = 1$. ■

Two ranges $[u, v]$ and $[w, x]$, $u \le v$, $w \le x$, $u \le w$, *intersect* iff $u < w < v < x$ (see Figure 3(a)). The ranges are *nested* iff $u \le w \le x \le v$ (see Figure 3(b)). The ranges are *disjoint* iff $v < w$ (see Figure 3(c)). Two prefixes intersect, are nested, or are disjoint iff the corresponding property holds with respect to their ranges.

The following lemma is implicit in [9] and other papers on prefix matching.

Lemma 2 *Let $P_i = [s_i, f_i]$ and $P_j = [s_j, f_j]$ be two different prefixes. P_i and P_j are either nested or disjoint (i.e., they cannot intersect).*

Proof When $length(P_i) = length(P_j)$, the destination addresses matched by P_i and P_j are different. So, the ranges of P_i and P_j (and hence the prefixes) are disjoint. When $length(P_i) \neq length(P_j)$, we may, without loss of generality, assume that $length(P_i) < length(P_j)$. If P_i is not a prefix of P_j (i.e., P_i and P_j differ in one of the specified bits), then again, the ranges of P_i and P_j (and hence the prefixes) are disjoint. If P_i is a prefix of P_j, $s_i \le s_j \le f_j \le f_i$. Consequently, P_j is nested within P_i. ■

Lemma 3 *Let $P = [s, f]$, $s \neq f$, be a prefix and let $a = \lfloor (s+f)/2 \rfloor$. P is the longest length prefix that includes*[1] $[a, a+1]$.

Proof First observe that $f = s + 2^{W-g} - 1$, where $g = \mathit{length}(P)$. Since prefixes do not intersect, any longer (or equal) length prefix $P' = [s', f']$ that includes $[a, a+1]$ must have $s \leq s' \leq a < a+1 \leq f' \leq f$. Further, s, s', f, and f' all have the same first g bits and s' and f' have the same first $g+1$ (or more) bits. Since a and $a+1$ differ in bit $g+1$, P' cannot include $[a, a+1]$. Therefore, no prefix whose length is longer than that of P can include $[a, a+1]$. If $\mathit{length}(P') = \mathit{length}(P)$, $P' = P$. So, P is the longest length prefix that includes $[a, a+1]$. ■

4 Representation Using Binary Search Trees

4.1 The Representation

Let r_i, $1 \leq i \leq q \leq 2n$ be the distinct end points of the given set of n prefixes. Assume that these end points are ordered so that $r_i < r_{i+1}$, $1 \leq i < q$. Each of the intervals $[r_i, r_{i+1}]$, $1 \leq i < q$ is called a *basic interval*. The basic intervals of the five-prefix example of Figure 1 are [0, 10], [10, 11], [11, 16], [16, 18], [18, 19], [19, 23], and [23, 31]. These basic intervals are labeled $r1$ through $r7$ in Figure 2(a).

To perform longest prefix matches, inserts and deletes in $O(\log n)$ time per operation, we use a collection of $n+1$ binary search trees (CBST). Although the $O(\log n)$ performance results only when each of the $n+1$ binary search trees in the CBST is a balanced binary search tree, we introduce the CBST in terms of binary search trees that are not necessarily balanced.

4.1.1 The Basic Interval Tree (BIT)

Of the $n+1$ binary search trees in the CBST, one is called the *basic interval tree (BIT)*. The BIT comprises internal and external nodes and there is one internal node for each r_i. Since the BIT has q internal nodes, it has $q+1$ external nodes. The first and last of these, in inorder, have no significance. The remaining $q-1$ external nodes, in inorder, represent the $q-1$ basic intervals of the given prefix set. Figure 4(a) gives a possible (we say possible because, at this time, any binary search tree organization for the internal nodes will suffice) BIT for our five-prefix example of Figure 2(a). Internal nodes are shown as rectangles while circles denote external nodes.

The fields of the BIT internal nodes are called *key*, *leftChild*, and *rightChild*. We describe the structure of the BIT external nodes later.

[1]The prefix $P_i = [s_i, f_i]$ includes the interval $[a, b]$ iff $s_i \leq a \leq b \leq f_i$.

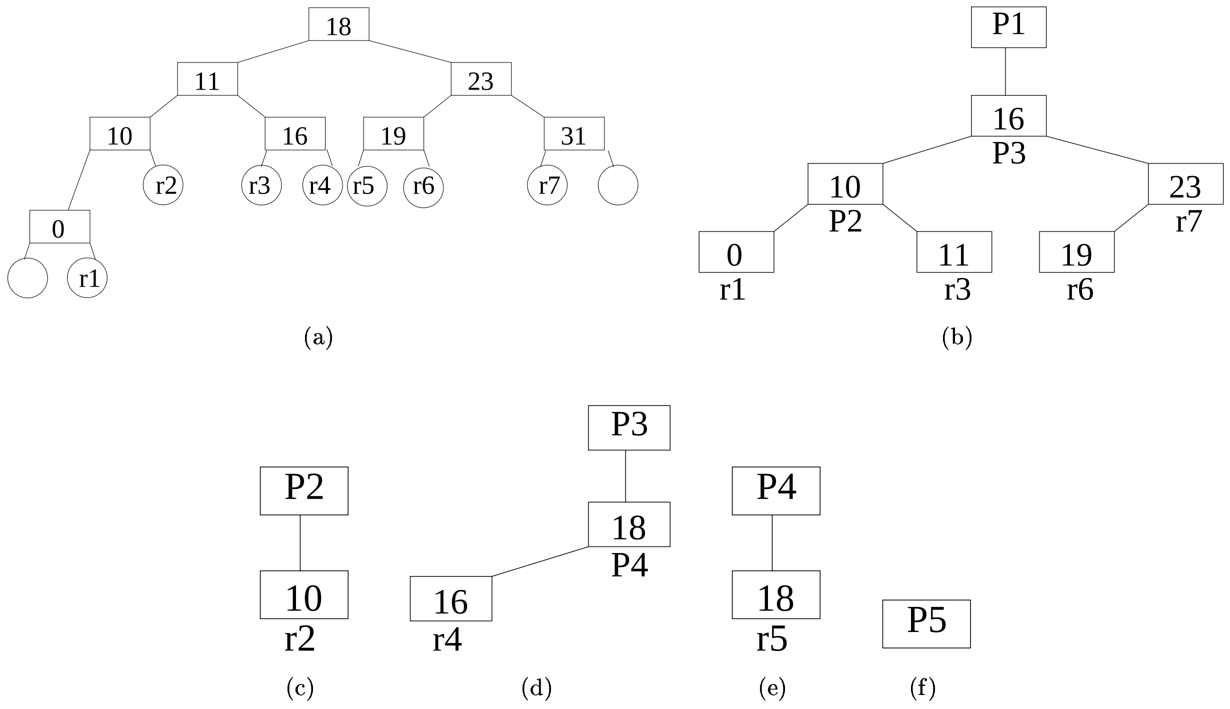


Figure 4: CBST for Figure 2(a). (a) base interval tree (b) prefix tree for $P1$ (c) prefix tree for $P2$ (d) prefix tree for $P3$ (e) prefix tree for $P4$ (f) prefix tree for $P5$

4.1.2 The Prefix Trees

The remaining n binary search trees in the CBST are *prefix trees*. For each of the n prefixes in the router table, there is exactly one prefix tree. For each prefix and basic interval, x, define $next(x)$ to be the smallest range prefix (i.e., the longest prefix) whose range includes the range of x. For the example of Figure 2(a), the $next()$ values for the basic intervals $r1$ through $r7$ are, respectively, $P1$, $P2$, $P1$, $P3$, $P4$, $P1$, and $P1$. Notice that the next value for the range $[r_i, r_{i+1}]$ is the same as the "$>$" value for r_i in Figure 2(b), $1 \leq i < q$. The $next()$ values for the nontrivial prefixes $P1$ through $P4$ of Figure 2(a) are, respectively, "-", $P1$, $P1$, and $P3$. The $next()$ values for the basic intervals and the nontrivial prefixes of Figure 2(a) are shown in Figure 5 as left arrows.

The prefix tree for prefix P comprises a header node plus one node, called a *prefix node*, for every nontrivial prefix or basic interval x such that $next(x) = P$. The prefix trees for each of the five prefixes of Figure 2(a) are shown in Figures 4(b)-(f). Notice that prefix trees do not have external nodes and that the prefix nodes of a prefix tree store the start point of the range or prefix represented by that prefix node. In the figures, the start points of the basic intervals and prefixes are shown inside the prefix nodes

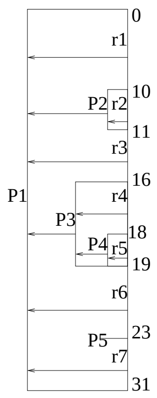


Figure 5: *next*() values are shown as left arrows.

while the basic interval or prefix name is shown outside the node. *Notice also that nontrivial prefixes and basic intervals do not store the value of next() explicitly. The value of next() is stored only in the header of a prefix tree.*[2]

4.1.3 BIT External Nodes

Each of the $q-1$ external nodes of the BIT that represents a basic interval x points to the prefix node that represents this basic interval in the prefix tree for $next(x)$. We call this pointer *basicIntervalPointer*. In addition, an external node that represents the basic interval $x = [r_i, r_{i+1}]$ has a pointer *startPointer* (*finishPointer*) which points to the header node of the prefix tree for the trivial prefix (if any) whose range start and finish points are r_i (r_{i+1}). For example, *startPointer* for $r7$ = [23,31] in Figure 2(a) points to the header node for the prefix tree of the trivial prefix $P5$; *finishPointer* for $r6$ = [19, 23] also points to the header node for the prefix tree of $P5$; the remaining start and finish pointers are null.

4.2 Longest Prefix Matching

Notice that, because of our assumption that the default prefix is always present, there is always a prefix in our database that matches any W-bit destination address d. The search for the longest prefix that matches d is done in two steps:

Step 1 First we start at the root of the BIT and move down to an appropriate external node. An external node x that represents the basic interval $[r_i, r_{i+1}]$ is *appropriate* for d iff (a) $d = r_i$ and $x.startPointer \neq null$, or (b) $d = r_{i+1}$ and $x.finishPointer \neq null$, or (c) $LMP(d) = next(x)$.

[2]If $next()$ values were explicitly stored with basic intervals and trivial prefixes, an update would take $O(n)$ time, because $O(n)$ $next()$ values change following an insert/delete.

Notice that the appropriate node for a given d may not be unique. For instance, for our example BIT, the external nodes for both $r6$ and $r7$ are appropriate when $d = 23$. When $d = 18$, only the external node for $r5$ is appropriate.

Step 2 If cases (a) or (b) of Step 1 apply, then $LMP(d)$ is obtained by following the non-null start or finish pointer. When case (c) applies, the basic interval pointer is followed into the prefix tree corresponding to $next(x)$. The header node of this prefix tree contains the longest matching prefix for d. This header node is located by following parent pointers.

In step 1, we search for an appropriate external node by performing a series of comparisons beginning at the root of the BIT. The search process differs from that employed to search a normal binary search tree (see, for example, [8]) only in how we handle equality between the address d and the key in the current search tree node y. Whenever d equals the key in an internal node y (i.e., $d = y.key$) of the BIT, we know that the basic interval $[r_i, r_{i+1}]$ represented by the rightmost (leftmost) external node in the left (right) subtree of y is such that $r_{i+1} = d$ ($r_i = d$). It is not too difficult to see that one (or both) of these two external nodes is an appropriate external node for d. To determine which, we examine the least significant bit ($lsb(key.y)$) of $key.y$ (equivalently, examine $lsb(d)$). If $lsb(key.y) = 0$, then it follows from Lemma 1 that $y.key = d$ is the start point of some prefix (note that the start and finish points of a trivial prefix are the same). Therefore, the leftmost external node in the right subtree of y is an appropriate node for d (recall that the basic interval for this external node is $[r_i, r_{i+1}]$, where $r_i = d$). When $lsb(y.key) = 1$, $y.key = d$ is the finish point of some prefix and so the rightmost external node in the left subtree of y is an appropriate node for d. This external node has $r_{i+1} = d$.

As an example, suppose we wish to determine $LMP(11)$. We start at the root of the BIT of Figure 4(a). Since $d = 11 < root.key = 18$, the current node y become the left child of the root. Now, since $d = y.key$ and $lsb(y.key) = 1$, the appropriate external node for d is the rightmost external node in the left subtree of y. This external node represents the basic interval $r2$. Notice that $next(r2) = P2$. As another example, consider determining $LMP(18)$. Since $d = 18 = root.key$ and $lsb(root.key) = 0$, the appropriate external node is the leftmost external node in the right subtree of the root. This external node represents the basic interval $r5 = [r_i, r_{i+1}] = [18, 19]$. Once again, notice that $LMP(18) = next(r5) = P4$. For $d = 23$, we reach the external node for $r6 = [r_i, r_{i+1}] = [19,23]$. Since $d = r_{i+1}$ and the finish pointer of this external node is non-null, the finish pointer (this points to the header node of the prefix tree for the trivial prefix $P5$) is used to determine $LMP(23) = P5$. Notice that when the router table has a trivial prefix that matches the destination address d, this trivial prefix is $LMP(d)$.

```
algorithm $longestMatchingPrefix(d)$
{// return header node for $LMP(d)$
    // find appropriate external node
    $y$ = root of BIT;
    while ($y$ is an internal node)
        if ($d < y.key$) $y = y.leftChild$;
        else if ($d > y.key$) $y = y.rightChild$;
        else // $d$ equals $y.key$
            if ($lsb(y.key)$ is 0)
                {
                    $eNode$ = leftmost external node in right subtree of $y$;
                    if ($eNode.startPointer$ is null)
                        return ($prefix(eNode.basicIntervalPointer)$);
                    else return ($eNode.startPointer$);
                }
            else //$lsb(y.key)$ is 1
                {
                    $eNode$ = right most external node in left subtree of $y$;
                    if ($eNode.finishPointer$ is null)
                        return ($prefix(eNode.basicIntervalPointer)$);
                    else return ($eNode.finishPointer$);
                }
    return ($prefix(y.basicIntervalPointer)$);
}

algorithm $prefix(pNode)$
{// return prefix in header node of prefix tree that contains node $pNode$
    $y = pNode$;
    while ($y$ is not a header node)
        $y = y.parent$;
    return ($y$);
}
```

Figure 6: Algorithm to find $LMP(d)$

Figure 6 gives a high-level statement of the algorithm to determine $LMP(d)$.

Theorem 1 *(a) Algorithm* longestMatchingPrefix *correctly finds $LMP(d)$.*
(b) The complexity of algorithm longestMatchingPrefix *is $O(height(BIT) + height(prefixTree(d)))$, where $prefixTree(d)$ is the prefix tree for $LMP(d)$.*

Proof Correctness follows from the definition of the BIT and prefix tree data structures. For the complexity, we note that it takes $O(height(BIT))$ time to find the appropriate external node and an

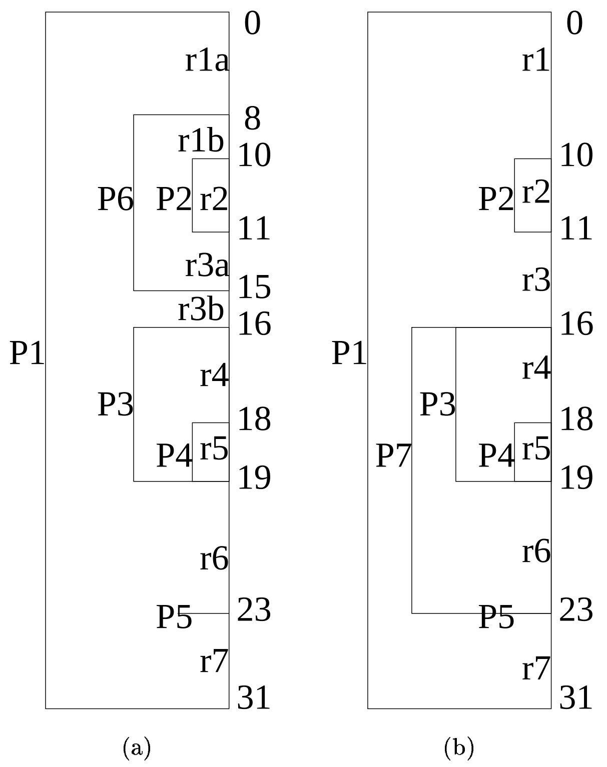


Figure 7: (a) Figure 2(a) after inserting $P6 = 01*$ (b) Figure 2(a) after inserting $P7 = 10*$

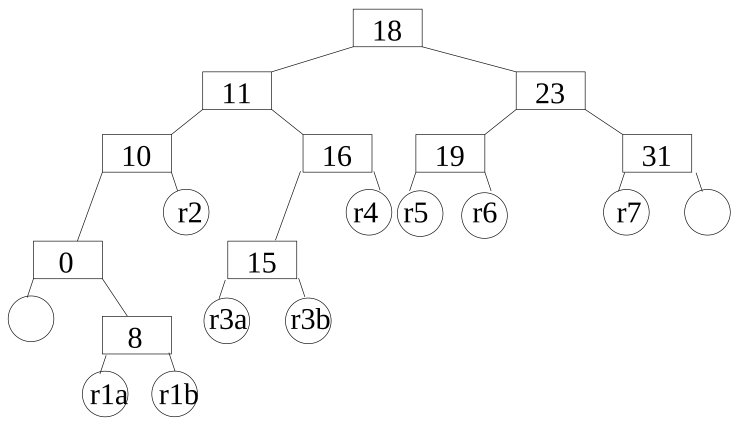
additional $O(height(prefixTree(d)))$ time to find $LMP(d)$ in case the function $prefix$ is invoked. ■

4.3 Inserting A Prefix

Suppose we wish to add the prefix $P6$ = 01* = [8, 15] to the prefix set $P1$-$P5$. Figure 7(a) gives the pictorial representation for the prefixes $P1$-$P6$. Relative to the pictorial representation of $P1$-$P5$ (Figure 2(a)), we see that the insertion of $P6$ has created two new end points (8 and 15), the basic interval $r1$ has been split into the basic intervals $r1a$ and $r1b$ as a result of the new end point 8, and the basic interval $r3$ has been split into the basic intervals $r3a$ and $r3b$ as a result of the new end point 15.

Figure 7(b) shows the pictorial representation for the case when $P1$-$P5$ are augmented by the prefix $P7$ = 10* = [16, 23]. In this case no new end points are created and none of the basic intervals of $P1$-$P5$ split.

In addition to possibly increasing the number of distinct end points, the insertion of a new prefix changes the $next()$ value of certain prefixes and basic intervals. The insertion of $P6$ into $P1$-$P5$ changes $next(P2)$ from $P1$ to $P6$ ($next(r1b)$ and $next(r3a)$ become $P6$). The insertion of $P7$ into $R1$-$R5$ changes



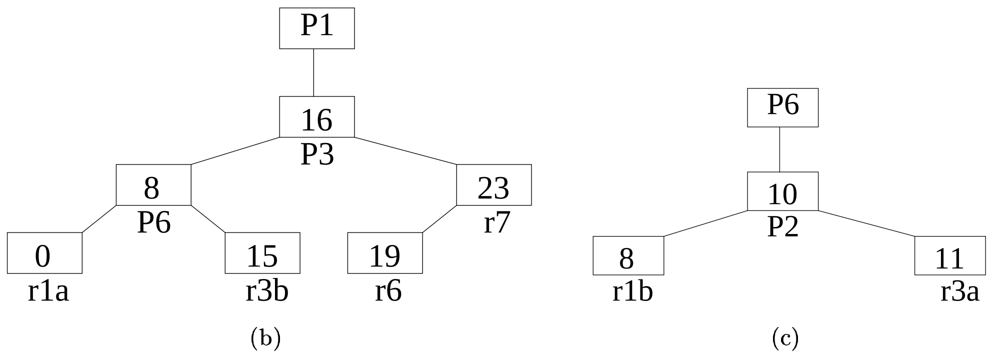


Figure 8: Basic interval tree and prefix trees after inserting $P6 = 01*$ into Figure 4: (a) BIT for $P1$-$P5$ and $P6$ (b) prefix tree for $P1$ (c) prefix tree for $P6$

$next(P3)$ and $next(r6)$ from $P1$ to $P7$.

4.3.1 Updating The BIT

Since the default prefix $*$ is always present, we need not be concerned with insertion into an empty BIT.

It is easy to verify that the insertion of a new prefix will increase the number of distinct end points by 0, 1, or 2. Correspondingly, the number of basic intervals will increase by 0, 1, or 2. Because the number of internal (external) nodes in a BIT equals (is one more than) the number of distinct end points, the number of internal and external nodes in the BIT increases by the same amount as does the number of distinct end points. Figure 8(a) shows the BIT for $P1$-$P6$. Since the insertion of $P7$ into the prefix set $P1$-$P5$ does not change the set of distinct end points, the BIT for $P1$-$P5$ and $P7$ has the same structure as does that for $P1$-$P5$.

```
algorithm $insertEndPoint(u)$
{// insert the end point $u$ into the BIT
  $y$ = root of BIT;
  while ($y$ is an internal node)
    if ($u < y.key$) $y = y.leftChild$;
    else if ($u > y.key$) $y = y.rightChild$;
    else // $u$ equals $y.key$
    {// $u$ is not a new end point
      if (length of new prefix is $W$)
        {
          $eNode$ = leftmost external node in right subtree of $y$;
          update $eNode.startPointer$ to point to header node for new prefix;
          $eNode$ = right most external node in left subtree of $y$;
          update $eNode.finishPointer$ to point to header node for new prefix;
        }
        return;
    }
  // $u$ is a new end point
  insert a new internal node $z$ with $z.key = u$ between $y$ and its
   parent and create a new external node for the remaining child of $z$;
  return;
}
```

Figure 9: Algorithm to insert an end point

Lemma 4 *Let $P = [s, f]$ be a new prefix that is inserted into a router database. Assume that the insertion of P creates no new end points.*

(a) If length(P) $< W$, the BIT is unchanged. (Even though the next value may change for several basic intervals and prefixes, these changes do not affect the BIT.)

(b) If length(P) $= W$, the structure of the BIT is unchanged. However, the start pointer in the external node for the basic interval $[r_i, r_{i+1}]$, where $r_i = s = f$ and the finish pointer in the external node for the basic interval $[r_i, r_{i+1}]$, where $r_{i+1} = s = f$ change (both now point to the header node for the prefix tree of P).

Proof Straightforward. ■

To update the BIT as required by the insertion of the prefix $P = [s, f]$, we insert the end points s and f into the BIT using algorithm *insertEndPoint* of Figure 9. Of course, when $s = f$, we invoke *insertEndPoint* just once.

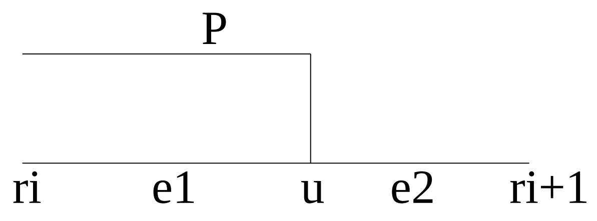


Figure 10: Splitting a basic interval when $lsb(u) = 1$

The fields of the two external node children of the newly created internal node z are easily changed/set to their correct values. When a new internal node z is created, a basic interval $[r_i, r_{i+1}]$ is split into the two basic intervals $[r_i, u]$ and $[u, r_{i+1}]$. Let $e1$ and $e2$, respectively, be the external nodes that represent these basic intervals. Let e be the external node that represents the original interval $[r_i, r_{i+1}]$ (note that e is either $e1$ or $e2$). The start pointer of $e1$ is the start pointer of e and the finish pointer of $e2$ is the finish pointer of e. When the length of the new prefix P is W, the basic interval pointers of $e1$ and $e2$ are the same as that of e and the finish pointer of $e1$ and the start pointer of $e2$ point to the header node of the prefix tree of the new prefix. When $length(P) \neq W$, the finish pointer of $e1$ and the start pointer of $e2$ are null. Further, when $length(P) \neq W$ and $lsb(u) = 1$ (see Figure 10), the basic interval pointer of $e2$ is the same as that of e and the basic interval pointer of $e1$ points to a new node that is to go into the prefix tree of the new prefix P. The case when $length(P) \neq W$ and $lsb(u) = 0$ is similar.

Theorem 2 *(a) Algorithm* insertEndPoint *correctly inserts an end point into the BIT.*
(b) The complexity of the algorithm is $O(height(BIT))$.

Proof Correctness follows from the definition of a BIT. For the complexity, we see that it takes $O(height(BIT))$ time to exit the **while** loop. The ensuing insert (if any) of a new internal and external node takes $O(1)$ time if the BIT is not to be balanced and $O(height(BIT))$ time if the BIT is to be balanced. ■

4.3.2 Updating Prefix Trees

When the prefix $P = [s, f]$ is inserted, we must create a new prefix tree for P. Additionally, when $length(P) < W$ or when $length(P) = W$ and s is a new end point, we must update the prefix tree for the longest prefix $Q = [a, b]$ such that $a \leq s \leq f \leq b$ (i.e., the prefix Q such that $next(P) = Q$). Note that because of our assumption that the default prefix $*$ is always present, Q exists whenever P is not the default prefix. We assume that whenever a request is made to insert a prefix that is already in the database, we need only update the next-hop information associated with this prefix. Therefore, the only time that Q does not exist, we are to simply locate the header node for the default prefix and update the next-hop information. For the remainder of this subsection, we assume that Q exists. Additional work that is to be done includes the insertion of up to two new basic interval nodes. These nodes go into the prefix trees for P and/or Q.

Consider the insertion of $P6 = [8, 15]$ into $P1$-$P5$ (Figures 2(a) and 7(a)). When $P6$ is inserted, $Q = P1$. Let Z be the set of prefixes and basic intervals x for which $next(x) = Q = P1$ and the range of x is contained within that of $P6$ (i.e., $P2$). The $next()$ value for the prefixes and basic intervals in Z changes from $Q = P1$ to $P6$. The basic intervals ($r1$ and $r3$) that intersect the range of $P6$ (recall from Lemma 2 that no prefix can intersect $P6$) get split into four basic intervals with two of these having $next$ value Q and the other two having $next$ value $P6$. The prefix trees for prefixes other than Q and $P6$ are unaffected by the insertion of $P6$.

To make the above changes, we use the split and join operations [8] of a binary search tree. For binary search trees T, $small$, and big, these operations are defined below.

1. $T.split(u)$ Split T into two binary search trees $small$ and big such that $small$ has all keys/elements of T that are less than u and big has those that are greater than or equal to u.

2. $join(small, big)$ This operation starts with two binary search trees $small$ and big with the property that all keys in $small$ are less than every key in big and creates a binary search tree that includes all keys in $small$ and big.

To determine, the basic intervals and prefixes in the prefix tree of $Q = P1$ whose $next$ value changes to $P6$, we first split the prefix tree of $P1$ by invoking $split(8)$ (8 is the start point of the new prefix $P6$). The resulting binary search trees $small1$ and $big1$ have the keys $\{0\}$ and $\{10, 11, 16, 19, 23\}$, respectively. Next, we split the binary search tree $big1$ by invoking $split(15)$ (15 is the finish point of $P6$) to get the binary search trees $small2$ and $big2$, which have the keys $\{10, 11\}$ and $\{16, 19, 23\}$, respectively. We now have three binary search trees $small1$ with key $\{0\}$ representing the basic interval $\{r1a\}$, $small2$ with keys $\{10, 11\}$ representing $\{r1b, P2\}$, and $big2$ with keys $\{16, 19, 23\}$ representing $\{P3, r6, r7\}$. To construct the new prefix tree for $P1$, we join $small1$ and $big2$ and then insert the basic interval $r3b$ as well as the new prefix $P6$. To get the prefix tree for $P6$, we insert the basic interval $r1b$ into $small2$. The resulting $P1$ and $P6$ prefix trees are shown in Figures 8(b) and (c).

Now, consider the insertion of $P7 = [16, 23]$ into $P1 - P5$. Once again, $Q = P1$. Following $split(16)$, $small1$ has the keys $\{0, 10, 11\}$ and $big1$ has the keys $\{16, 19, 23\}$. When $big1$ is split using $split(23)$, we get $small2$ with keys $\{16, 19\}$ and $big2$ with the key $\{23\}$. To get the new prefix tree for $P1$, we join

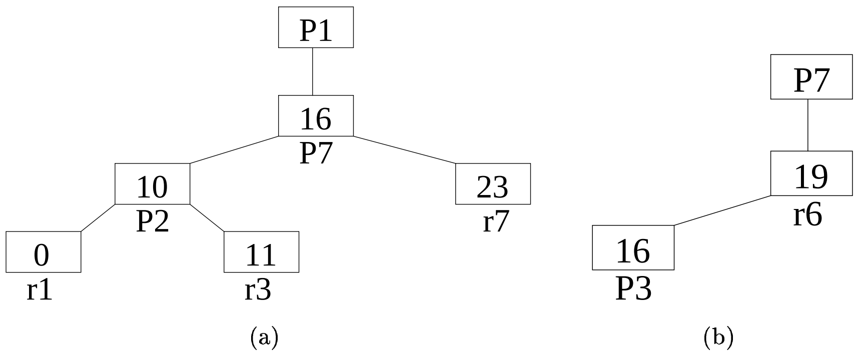


Figure 11: Prefix trees after inserting $P7 = 10*$ into $P1$-$P5$ (a) prefix tree for $P1$ after the insertion of $P7$ (b) prefix tree for $P7$

*small*1 and *big*2 and then insert the new prefix $P7$. The resulting tree has the keys $\{0, 10, 11, 16, 23\}$ (the key 16 represents $P7$). *small*2 is the tree for $P7$. These prefix trees for $P1$ and $P7$ are shown in Figures 11(a) and (b).

To complete the discussion of the insertion operation, we need to describe how the prefix Q is determined. When $length(P) < W$, Q may be determined using Lemma 5. When $length(P) = W$ and s is a new end point, Q is $LMP(s)$.

Lemma 5 *Let R be a prefix set that includes the default prefix $*$. Let $P = [s, f]$, $s \neq f$ (i.e., $length(P) < W$), $P \notin R$, be a prefix that is to be inserted into R. Let $a = \lfloor (s+f)/2 \rfloor$.*

(a) *There is a unique basic interval x of R that contains $[a, a+1]$.*

(b) *The longest prefix $Q \in R$ that includes the interval $[s, f]$ is $next(x)$.*

Proof (a) Since the default prefix $*$ is in R, the distinct end points of R are $0 = r_1 < r_2 < ... < r_q = 2^W - 1$. Therefore, there is a unique i such that $r_i \leq a < a+1 \leq r_{i+1}$. So, $x = [r_i, r_{i+1}]$ is the unique basic interval of R that contains $[a, a_{i+1}]$.
(b) By definition, $next(x)$ is the smallest range prefix (i.e., longest prefix) $P' = [s', f']$ of R that includes the basic interval $[r_i, r_{i+1}]$. Therefore, P' is the longest prefix of R that includes $[a, a+1]$. From Lemma 3 and $P \notin R$ (so $P \neq P'$), it follows that $length(P') < length(P)$. Since prefixes do not intersect and since both P and P' include $[a, a+1]$, $s' \leq s \leq a < a+1 \leq f \leq f'$. Further, since P' is the longest prefix of R with this property, $Q = P' = next(x)$. ■

Figure 12 gives a high-level description of our algorithm to update the prefix trees.

```
algorithm $updatePrefixTrees(s, f)$
{// update the prefix trees when the prefix $P = [s, f]$ is inserted
    if ($s == 0 \&\& f == 2^W - 1$)
    {// $P$ is the default prefix
        Update next-hop field of default prefix;
        return;
    }
    if ($s == f$)
    {// $length(P) = W$
        if ($P$ is not a new prefix) Update next-hop field for $P$;
        else
        {
            Create a header node for $P$'s prefix tree;
            if ($s$ is a new point)
            {
                $Q = LMP(s)$;
                Insert the basic interval that begins at $s$ into $Q$;
            }
        }
        return;
    }
    // $P$ is a nontrivial prefix
    Determine $Q$ using Lemma 5;
    if ($P == Q$) Update next hop of $Q$ and return;
    $(small1, big1) = Q.split(s)$;
    $(small2, big2) = big1.split(f)$;
    $Q = join(small1, big2)$;
    Insert $s$ (i.e., prefix $P$) into $Q$;
    if ($f <$ finish point of prefix represented by $Q$)
        Insert $f$ into $Q$;
    Insert basic intervals into $Q$ as needed;
    Insert basic intervals into $small2$ as needed;
    $small2$ is the prefix tree for $P$;
}
```

Figure 12: Algorithm to update prefix trees

Theorem 3 *(a) Algorithm* updatePrefixTrees *correctly updates a prefix tree.*
(b) The complexity of the algorithm is $O(height(BIT) + split(pt) + join(pt) + insert(pt))$, where $split(pt)$, $join(pt)$, and $insert(pt)$ are, respectively, the times to split a prefix tree, join two prefix trees, and insert into a prefix tree.

Proof Correctness follows from the definition of a prefix tree. For the complexity, we see that it takes

$O(height(BIT))$ time to determine Q. In addition to determining Q, at most 2 splits, 1 join, and 3 insertions into prefix trees are done. ■

Theorem 4 *The complexity of the insert-prefix operation is* $O(height(BIT) + split(pt) + join(pt) + insert(pt))$.

Proof Follows from the complexities of *insertEndPoint* and *updatePrefixTrees* and the observation that when a prefix is inserted, we make at most 2 invocations of *insertEndPoint* and 1 of $updatePrefixTrees$. ■

4.4 Deleting A Prefix

To delete $P6 = [8, 15]$ from the database $P1 - P6$ of Figure 7(a), we must do the following:

1. Delete 8 and 15 from the BIT and merge the basic intervals $r1a$ and $r1b$ as well as $r3a$ and $r3b$.
2. Move the prefix-tree node for $P2$, which is presently in the in prefix tree for $P6$ to the prefix tree for $P1$ and discard the remainder of the prefix tree for $P6$.

To delete $P7 = [16, 23]$ from the database $P - P5$ and $P7$ of Figure 7(b), we must move the prefix-tree nodes for $P3$ and $r6$ from the prefix tree for $P7$ to the prefix tree for $P1$ and discard the header node of the prefix tree for $P7$. To delete $P5 = [23, 23]$ from $P1 - P5$, we must remove 23 from the BIT, merge the basic intervals $r6$ and $r7$, and discard the prefix tree for $P5$. The deletion of the default prefix $*$ requires us to simply change the next-hop field for this prefix to null (recall that the default prefix must be retained in the database at all times).

In the remainder of our discussion, we assume that the prefix to be deleted is not the default prefix. We see that the deletion of a prefix $P = [s, f]$, $P \neq *$, requires us to perform some or all of the following tasks:

1. Locate the prefix tree for P.
2. Determine the longest prefix L whose range includes $[s, f]$ ($L = P1$ in our preceding examples).
3. Determine whether s and/or f are to be deleted from the BIT. If so, delete them.
4. Move a portion of the prefix tree for P into that of L and discard the remainder.
5. Merge pairs of external nodes in the BIT.

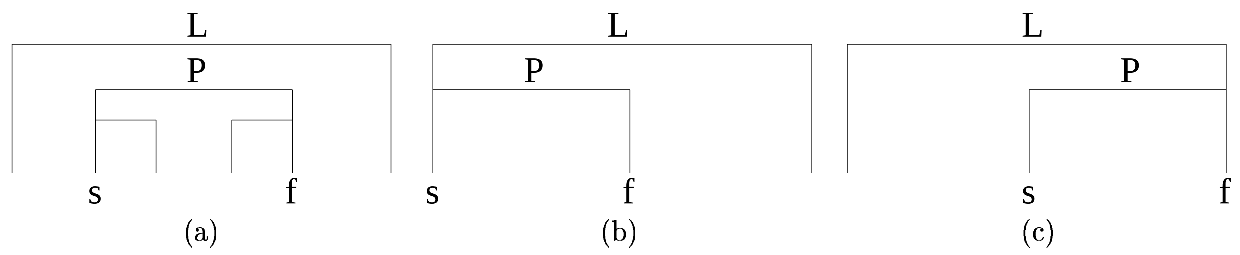


Figure 13: (a) P is shortest (b) P is not shortest (c) P is not shortest

To perform task 1, we observe that when $s = f$, the prefix tree for P may be located by first determining an external node e of the BIT that represents a basic interval $[r_i, r_{i+1}]$ with either $r_i = s$ or $r_{i+1} = f$. In the former case, *e.startPointer* gives us the desired prefix tree and in the latter case, $e.finishPointer$ does this. In case the pointer is null, P is not a prefix of the database. When $s \neq f$, task 1 may be performed using Lemma 5 to determine prefix Q using s and f. If $Q \neq P$, then P is not in the prefix database. In case the prefix to be deleted is not in the database, the deletion algorithm terminates.

A simple strategy for task 2 is to add a prefix-node pointer $prefixNode$ to the header node of every prefix tree. The prefix-node pointer for the prefix S points to the unique node N that is in one of the prefix trees and represents prefix S. By following parent pointers from N, we reach the header node for the prefix L. The prefix-pointer in the header node of the prefix tree for S is set when S is inserted into the database. Once set, this pointer does not change. A slightly more involved strategy is described now. This strategy does not require us to make any changes to the BIT or prefix-trees structures. First note that since the prefix database contains the default prefix $*$ and since $P \neq *$, the database contains a unique prefix L of longest length whose range includes $[s, f]$. To determine L, let U denote the subset of database prefixes that either start at s or finish at f (or both). Since $P \in U$, U is not empty. Let S be the shortest prefix in U. We consider the following three cases, which are exhaustive: (1) $P = S$, (2) $P \neq S$ and S starts at s, and (3) $P \neq S$ and S finishes at f. These three cases are shown pictorially in Figure 13. Let x be the basic interval (if any) that includes $[s-1, s]$ (note that when $s = 0$, there is no such x) and let y be the basic interval (if any) that includes $[f, f+1]$. We see that, in all cases, L is the shorter of the prefixes $next(x)$ and $next(y)$. We may determine $next(x)$ ($next(y)$) by following the basic interval pointer in the BIT external node for x (y) to the prefix tree for $next(x)$ ($next(y)$) and then following parent pointers to the header node for $next(x)$ ($next(y)$).

The easiest way to perform task 3 is to augment the BIT structure so that with each distinct end point we maintain a count of the number of prefixes in the database that start (finish) at that end point. When this count is 1, the deletion of $P = [s, f]$ requires us to remove s (f) from the BIT. The insert algorithm is easily modified to update the count fields whenever a prefix is inserted. An alternative strategy, which doesn't require us to augment either the BIT or prefix-trees structure, is described now. When $s = f$, s is to be deleted from the BIT iff there is no other prefix for which s is an end point. To determine this, compute $next(x)$, where x is the basic interval that includes $[s, s+1]$ in case $lsb(s) = 0$ and includes $[f-1, f]$, otherwise. When $lsb(s) = 0$, s is to be deleted iff the start point of $next(x) \neq s$. When $lsb(s) = 1$, s is to be deleted iff the finish point of $next(x) \neq s$. When $s \neq f$, s (f) is to be deleted iff none of the following is true (1) there is a prefix in the database whose start and finish points are s (f), (2) $next(x) \neq P$, where x is the basic interval (if any) that includes $[s, s+1]$ $([f-1, f])$, or (3) start (finish) point of L (task 2) equals s (f).

For task 4, we first delete the header node of the prefix tree for P as well as the basic interval nodes for the up to two basic intervals in the prefix tree of P that are to be merged with adjacent basic intervals. Call the resulting binary search tree $PT'(P)$. Next, we split the prefix tree $PT(L)$ for P as in $(small, big) = PT(L).split(s)$. The new prefix tree for L is $join(join(small, PT'(P)), big)$.

Task 5 is to be done only when either s or f or both are to be deleted. This task is easily integrated into the delete s (f) task (task 3).

Theorem 5 *The complexity of the delete operation is* $O(height(BIT) + height(pt) + split(pt) + join(pt) + delete(pt))$*, where* $height(pt)$ *is the height of a prefix tree and* $delete(pt)$ *is the time to delete from a prefix tree.*

Proof Task 1 is done by searching down the BIT and then (possibly) going up a prefix tree. This takes $O(height(BIT) + height(pt))$ time. Task 2 requires us to go down the BIT and up a prefix tree once for each of x and y. So, task 2 also takes $O(height(BIT) + height(pt))$ time. For task 3, we must determine whether the end points s and f of the prefix that is to be deleted are also to be deleted and then delete these end points if so determined. For each of s and f, we must find a $next()$ value and then (possibly) delete the point from the BIT. It takes $O(height(BIT) + height(pt))$ time to determine $next()$ and $O(height(BIT))$ to delete a point. For task 4, we must do up to 3 deletions from a prefix tree, perform 1 split, and 2 joins. So, task 4 takes $O(delete(pt) + split(pt) + join(pt))$. Finally, task 5 is integrated into task 3 without any increase in asymptotic complexity. ■

4.5 Complexity

The red-black tree [8] is a good choice of data structure for the binary search trees of the CBST. The following properties [8] of red-black trees are important to us:

1. The height of a red-black tree is logarithmic in the number of nodes in the tree.

2. We may insert into, delete from, and split a red-black tree in $O(\text{height of tree})$ time.

3. Two red-black trees with n_1 and n_2 nodes, respectively, may be joined in $O(\log(n_1 n_2))$ time.

From these properties and the earlier stated complexities of the search, insert, and delete algorithms for our proposed CBST structure, it follows that we can perform longest prefix matches as well as prefix insertion and deletion in $O(\log n)$ time, where n is the number of prefixes in the database. When the trees of the CBST structure are implemented as red-black trees, the resulting structure is called CRBT (collection of red-black trees).

Although the use of AVL trees in place of red-black trees also results in $O(\log n)$ router-table operations, red-black trees are generally believed to be faster than AVL trees by a constant factor. When unbalanced binary search trees are used in place of red-black trees, the complexity of the match/insert/delete algorithms becomes $O(n)$ (though the expected complexity is $O(\log n)$). Using splay trees in place of red-black trees results in router-table operations whose amortized complexity is $O(\log n)$. As for the space complexity, the BIT has at most $2n$ internal and $2n+1$ external nodes. Further, the n prefix trees together have n header nodes, $n-1$ prefix nodes (there is no prefix node for the default prefix), and at most $2n-1$ basic interval nodes. So, the BIT and the prefix trees together have at most $8n$ nodes. Therefore, the space complexity is $O(n)$.

4.6 Comments

Our algorithms assume that prefixes are given by the start and finish points of their ranges. In practical databases, this may not be the case; a prefix may be specified by its start point and length. In this case, the finish point of the prefix may be computed in $O(1)$ time provided we precompute the values $A(i) = 2^i - 1$, $0 \leq i \leq W$. The finish point of a prefix P whose start point is s is $s + A(W - length(P))$.

5 Experimental Results

We programmed the CRBT scheme in C++ and measured its performance using IPv4 prefix databases. The codes were run on a SUN Ultra Enterprise 4000/5000 computer. The g++ compiler with optimization

Database	Paix	Pb	MaeWest	Aads	MaeEast
Num of prefixes	85988	35303	30719	27069	22713
Num of 32-bit prefixes	1	0	1	0	1
Num of end points	167434	69280	60105	53064	44463
Max nesting depth	7	6	6	6	7
Avg nesting depth	2.13	1.90	1.90	1.86	1.82
Max prefix tree	76979	44333	38469	36201	32437
Avg prefix tree	2.95	2.96	2.96	2.96	2.96

Table 1: Prefix databases obtained from IPMA project on Sep 13, 2000 [11].

level -O3 was used. For test data, we used the five IPv4 prefix databases of Table 1. Interestingly, the number of distinct end points is almost twice the number of prefixes in each database. The *depth of nesting* is the number of prefixes that cover a given basic interval. For example, the depth of nesting for the basic interval r1 of Figure 2(a) is 1, because prefix P1 is the only prefix that covers r1. The depth of nesting for r5 is 3, because P1, P3 and P4 cover r5. The maximum depth of nesting is surprisingly almost the same for all five of our databases. Note that the depth of nesting reported in Table 1 includes the default prefix that we have added to the database. The average nesting depth is obtained by summing the nesting depth for all basic intervals and dividing by the number of basic intervals. For our sample data, the average nesting depth is very small. In fact, if we eliminate the default prefix added by us to the original databases, the average depth of nesting becomes about 1. So, most of the basic intervals are covered by at most 1 prefix!

Max prefix tree is the maximum number of nodes in any of the constructucted prefix trees. This number does not include the header node. Avg prefix tree is the average number of nodes in a prefix tree. Although the prefix tree for the default prefix has a very large number of nodes (this prefix tree was always the largest), the majority of the prefix trees are rather small.

Table 2 shows the amount of memory used by our data structure. Figure 14 compares the memory used by our data structure and that used by the the optimal variable-stride tries (VST) of Srinivasan and Varghese [18]. CRBT is our collection of red-black trees data structure, optVST is the optimal variable-stride trie of [18], and optVST-Butler is the optimal variable-stride trie of [18] augmented with Butler nodes. k is the height of the VST, and is a user-specified parameter. The data for VSTs are taken from [15]. Our CRBT structure takes 6.4 times the memory required by an optimal VST whose height is 2.

To measure the search, insert, and delete times for our data structure, we first obtained a random

Database	Paix	Pb	MaeWest	Aads	MaeEast
Num of prefixes	85988	35303	30719	27069	22713
Memory	16139	6664	5786	5106	4280

Table 2: Memory for data structure (in KBytes)

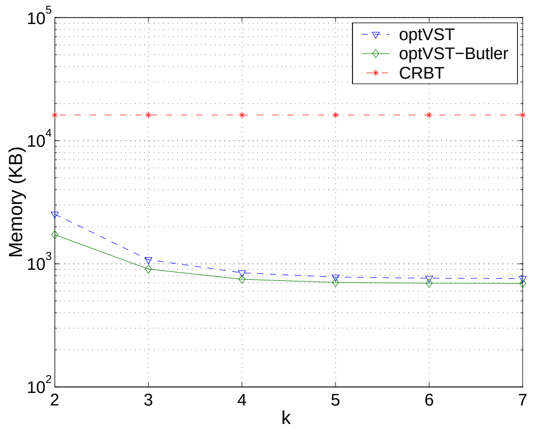


Figure 14: Memory required (in KBytes) by best k-VST and CRBT for Paix

permutation of the prefixes in the databases of [11]. For each database, we started with a CRBT that included the first 75% of the prefixes (order is determined by the random permutation). Then, the remaining 25% were inserted and the time to insert these 25% was measured. The average time for one of these inserts is reported in Table 3. For the delete time, we started with the CRBT for 100% of the prefixes in a database and measured the time to delete the last 25% of these prefixes. The average time for one of these delete operations is reported. Finally, for the search time, we measured the time to perform a search for a destination address in each of the basic intervals, and averaged over the number of basic intervals. The columns labeled Dyn (dynamic) give the times for the case when the insert and delete codes use C++'s `new` and `delete` methods to create and free nodes as required by the insert and delete operations, respectively. The columns labeled Sta (static) is for codes that do not use dynamic memory allocation/deallocation during insert and delete operations. Instead, we begin by allocating the maximum number of prefix trees as well as the maximum number of internal, external, and prefix nodes that may be needed. These allocated nodes are linked into four different chains, one for each node type. During an insert, nodes are taken from these chains, and during a delete, nodes are returned to these chains. As the run times of Table 3 show, dynamic allocation/deallocation accounts for a significant portion of the run time. Although one would, in theory, expect the time for a search to be the same

	Search		Insert		Delete	
Database	Dyn	Sta	Dyn	Sta	Dyn	Sta
Paix	1.97	2.20	47.45	36.29	46.99	36.29
Pb	1.73	1.88	44.19	28.33	44.19	33.99
MaeWest	1.83	2.00	44.28	27.25	42.97	31.25
Aads	1.51	1.88	44.33	28.08	41.38	32.51
MaeEast	1.57	1.80	42.27	28.18	40.51	29.94

Table 3: Execution time (in μsec) for randomized databases

when dynamic and static allocation and deallocation are used, the search times reported in Table 3 differ for three of the five databases. We suspect that this difference is largely due to caching differences resulting from the differences in node addresses in the two schemes. It is interesting to note that even though search, insert, and delete are $O(\log n)$ operations, an insert or delete takes about 25 times as much time as does a search when dynamic allocation and deallocation are used. When static allocation and deallocation are used, this ratio is about 16. In either case, the ratio is far more than the less than two factor between the time to insert/delete from a red-black tree and that to search a red-black tree. This order of magnitude jump in the ratio of insert/delete time and search time is due to the several join and split operations needed to insert/delete into/from a CRBT.

The times of Table 3 cannot be compared with the times for corresponding operations on an optimal VST as reported by Sahni and Kim in [15]. This is because in the experiments conducted in [15], the database prefixes were considered in the order they appear in each database rather than in a random order. Further, in the experiments of [15], we started with an optimal VST that contained the first 90% of the database prefixes and then inserted the remaining 10%. The average time for each of these latter inserts is reported in [15]. The delete times are similarly obtained by removing the last 10% of the prefixes from an optimal VST that initially has all 100% of the prefixes. The run times for our CRBT structure for the experiment conducted in [15] are shown in Table 4. Notice that in this experiment, the cost of an insert/delete is only 15 times that of a search when dynamic allocation and deallocation are used. When static allocation and deallocation are used, this ratio drops to about 7.

Figures 15 through 17 compare the run times for the search, insert, and delete operations using the Paix database and the CRBT and optimal VST structures. The search time using the CRBT structure is about 4 times that when an optimal VST of height $k = 2$ is used. However, when $k = 2$, each insert takes about 6 times the time taken by our CRBT structure with dynamic allocation/deallocation and 12 times the time taken by our CRBT structure with static allocation/deallocation! For the delete operation,

	Search		Insert		Delete	
Database	Dyn	Sta	Dyn	Sta	Dyn	Sta
Paix	2.33	2.21	27.91	13.96	30.24	18.61
Pb	1.98	1.98	28.33	14.16	31.16	19.83
MaeWest	2.28	2.28	29.31	13.03	29.31	16.28
Aads	2.22	1.85	29.56	14.78	29.56	18.48
MaeEast	1.76	2.20	26.42	17.61	30.82	17.61

Table 4: Execution time (in μsec) for original databases

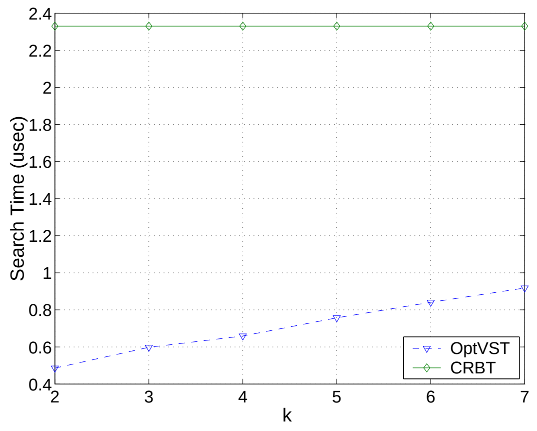


Figure 15: Search time (in μsec) comparison for Paix

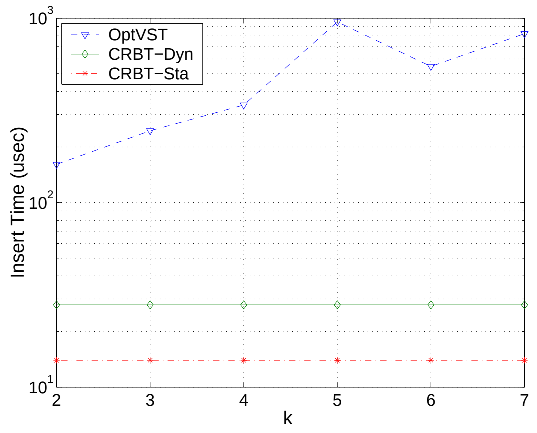


Figure 16: Insert time (in μsec) comparison for Paix

these ratios are 26 and 43, respectively. Note that these ratios increase as we increase k. So, although the CRBT is slower than optimal VSTs for the search operation, it is considerably faster for the insert and delete operations!

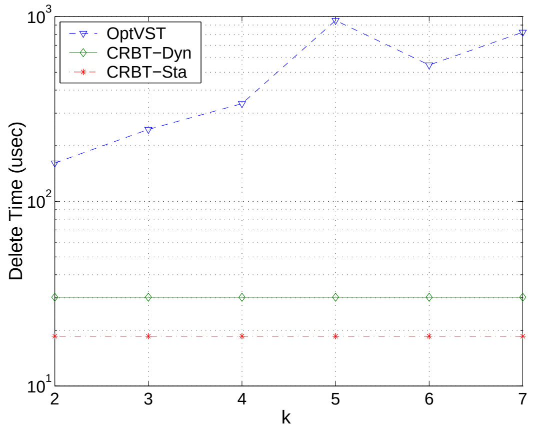


Figure 17: Delete time (in μsec) comparison for Paix

6 Conclusion

The collection of red-black search trees (CRBT) data structure developed by us provides the first known way to perform longest-prefix matches, as well as prefix insert and delete in $O(\log n)$ time. The CRBT is interesting from both the theoretical and practical viewpoints. From the theoretical viewpoint, it represents the first data structure to support dynamic router-table operations in $O(\log n)$ time each. From the practical viewpoint, we note that the CRBT permits updates to be performed in much less time than when structures such as the VST, which are optimized for search, are used. In a security conscious environment, our router would need to operate in a blocking mode (i.e., an insert/delete must complete any inbound/outbound packets are forwarded). In such an environment, the CRBT would block traffic for about 1/10th the time the VST would. On the other hand, when traffic is not blocked due to an insert/delete in progress, the VST would process packets at 4 to 5 times the rate of the CRBT. In another application environment, our concern may be the total time to process a stream of search/insert/delete requests. Suppose that for every pair of insert and delete requests, there are m search requests. Further, suppose that the search/insert/delete times for the optimal VST are 0.5/170/800 micro seconds and that the times for the CRBT are 2.2/14/19 micro seconds (these are approximately the times for Paix). Then, when $m > 551$, the optimal VST would perform better than the CRBT.

It is worth noting that the technique developed here may be used to extend the biased skip list scheme of Ergun et al. [6] so that lookups, inserts, and deletes may all be done in $O(\log n)$ expected time, while providing good expected performance for bursty access patterns (see Sahni and Kim [16]).

Finally, as noted in the introduction, when a compressed binary trie is used to represent a dynamic router table, each of the dynamic router-table operations takes $O(W)$ time. Since the compressed trie algorithms have much smaller constants than do the CRBT algorithms and since $n \leq 2^W$, the CRBT is expected to outperform the compressed binary trie structure for relatively small values of n. The threshold at which the compressed binary trie gives better overall performance is higher for IPv6 than for IPv4.

References

[1] A. Bremler-Barr, Y. Afek, and S. Har-Peled, Routing with a clue, *ACM SIGCOMM* 1999, 203-214.

[2] G. Chandranmenon and G. Varghese, Trading packet headers for packet processing, *IEEE Transactions on Networking*, 1996.

[3] G. Cheung and S. McCanne, Optimal routing table design for IP address lookups under memory constraints, *IEEE INFOCOM*, 1999.

[4] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *ACM SIGCOMM*, 1997, 3-14.

[5] W. Doeringer, G. Karjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking*, 4, 1, 1996, 86-97.

[6] F. Ergun, S. Mittra, S. Sahinalp, J. Sharp, and R. Sinha, A dynamic lookup scheme for bursty access patterns, *IEEE INFOCOM*, 2001.

[7] P. Gupta and N. McKeown, Dynamic algorithms with worst-case performance for packet classification, *IFIP Networking*, 2000.

[8] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of Data Structures in C++, W.H. Freeman, NY, 1995, 653 pages.

[9] B. Lampson, V. Srinivasan, and G. Varghese, IP Lookup using Multi-way and Multicolumn Search, *IEEE INFOCOM 98*, 1998.

[10] A. McAuley and P. Francis, Fast routing table lookups using CAMs, IEEE INFOCOM, 1993, 1382-1391.

[11] Merit, Ipma statistics, http://nic.merit.edu/ipma, (snapshot on Sep. 13, 2000), 2000.

[12] P. Newman, G. Minshall, and L. Huston, IP switching and gigabit routers, *IEEE Communications Magazine*, Jan., 1997.

[13] S. Nilsson and G. Karlsson, Fast address look-up for Internet routers, *IEEE Broadband Communications*, 1998.

[14] S. Sahni and K. Kim, Efficient Construction Of Fixed-Stride Multibit Tries For IP Lookup, *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, 178-184, 2001.

[15] S. Sahni and K. Kim, Efficient Construction Of Variable-Stride Multibit Tries For IP Lookup, *Proceedings IEEE Symposium on Applications and the Internet (SAINT)*, 2002.

[16] S. Sahni and K. Kim, Efficient Dynamic Lookup For Bursty Access Patterns, in preparation.

[17] K. Sklower, A tree-based routing table for Berkeley Unix, Technical Report, University of California, Berkeley, 1993.

[18] V. Srinivasan and G. Varghese, "Faster IP Lookups using Controlled Prefix Expansion", ACM Transactions on Computer Systems, Feb, 1-40, 1999.

[19] S. Suri, G. Vargjese, and P. Warkhede, Multiway range trees: Scalable IP lookup with fast updates, *GLOBECOM 2001.*

[20] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *ACM SIGCOMM*, 1997, 25-36.