A B-Tree Dynamic Router-Table Design [*]

Haibin Lu & Sartaj Sahni
{halu, sahni}@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

We propose B-tree data structures for dynamic router-tables for the cases when the filters are prefixes as well as when they are non-intersecting ranges. A crucial difference between our data structure for prefix filters and the B-tree router-table data structure of Suri et al. [20] is that in our data structure, each prefix is stored in $O(1)$ B-tree nodes per B-tree level, whereas in the structure of Suri et al. [20], each prefix is stored in $O(m)$ nodes per level (m is the order of the B-tree). As a result of this difference, a prefix may be inserted or deleted from an n-filter router table accessing only $O(\log_m n)$ nodes of our data structure; these operations access $O(m \log_m n)$ nodes using the structure of [20]. Even though the asysmptotic complexity of prefix insertion and deletion is the same in both B-tree structures, experiments conducted by us show that because of the reduced cache misses for our structure, the measured average insert and delete times using our structure are about 30% less than when the B-tree structure of [20] is used. Further, an update operation using the B-tree structure of [20] will, in the worst case, make 2.5 times as many cache misses as made when our structure is used. The asymptotic complexity to find the longest matching prefix is the same, $O(m \log_m n)$ in both B-tree structures, and in both structures, this operation accesses $O(\log_m n)$ nodes. The measured time for this operation also is nearly the same for both data structures. Both B-tree structures for prefix router-tables take $O(n)$ memory. However, our structure is more memory efficient by a constant factor. For the case of non-intersecting ranges, the highest-priority range that matches a given destination address may be found in $O(m \log_m n)$ time using our proposed B-tree data structure. The time to insert and delete a range is $O((m + D) \log_m n)$, where D is the maximum nesting depth of the ranges. Our data structure for non-intersecting ranges requires $O(n \log_m n)$ memory and $O(\log_m n)$ nodes are accessed during an operation.

Keywords: Packet routing, dynamic router-tables, longest-prefix matching, highest-priority-range matching, non-intersecting ranges, B-trees.

1 Introduction

An Internet router classifies incoming packets into flows[1] utilizing information contained in packet headers and a table of (classification) rules. This table is called the *rule table* (equivalently, *router table*). In this paper, we assume that packet classification is done using only the destination address of a packet. Each rule-table rule is a pair of the form (F, A), where F is a filter and A is an action. The action component of a rule specifies what is to be done when a packet that satisfies the rule filter is received. Sample actions

[*]This work was supported, in part, by the National Science Foundation under grant CCR-9912395.
[1]A *flow* is a set of packets that are to be treated similarly for routing purposes.

are drop the packet, forward the packet along a certain output link, and reserve a specified amount of bandwidth. In this paper, we assume that each rule filter is either a range $[u, v]$ of destination addresses or a destination address prefix. A range filter $[u, v]$ *matches* the destination address d iff $u \leq d \leq v$, while a prefix filter p matches all destination addresses that begin with p[2].

Since an Internet rule-table may contain several rules that match a given destination address d, a tie breaker is used to select a rule from the set of rules that match d. For purposes of this tie breaker, each rule is assigned a priority, and the highest-priority rule that matches d determines the action for all packets whose destination address is d[3]. We refer to rule tables in which the filters are ranges and in which the highest-priority matching filter is used as highest-priority range-tables (HPRT). When the filters of no two rules of an HPRT intersect, the HPRT is a nonintersecting HPRT (NHPRT).

In a *static* rule-table, the rule set does not vary in time. For these tables, we are concerned primarily with the following metrics:

1. *Time required to process an incoming packet.* This is the time required to search the rule table for the rule to use. We refer to this operation as a **lookup**.

2. *Preprocessing time.* This is the time to create the rule-table data structure.

3. *Storage requirement.* That is, how much memory is required by the rule-table data structure?

In practice, rule tables are seldom truly static. At best, rules may be added to or deleted from the rule table infrequently. Typically, in a “static” rule table, inserts/deletes are batched and the rule-table data structure reconstructed as needed.

In a *dynamic* rule-table, rules are added/deleted with some frequency. For such tables, inserts/deletes are not batched. Rather, they are performed in real time. For such tables, we are concerned additionally with the time required to insert/delete a rule. For a dynamic rule-table, the initial rule-table data structure is constructed by starting with an empty data structure and then inserting the initial set of rules into the data structure one by one. So, typically, in the case of dynamic tables, the preprocessing metric, mentioned above, is very closely related to the insert time.

Data structures for rule tables in which each filter is a destination address prefix and the rule priority is the length of this prefix have been intensely researched in recent years. We refer to rule tables of this

[2]For example, the filter 10* matches all destination adresses that begin with the bit sequence 10; the length of this prefix is 2.

[3]We may assume either that all priorities are distinct or that selection among rules that have the same priority may be done in an arbitrary fashion

type as longest-matching prefix-tables (LMPT). Although every LMPT is also an NHPRT, an NHPRT may not be an LMPT. We use W to denote the maximum possible length of a prefix. In IPv4, $W = 32$ and in IPv6, $W = 128$.

Ruiz-Sanchez, Biersack, and Dabbous [11] review data structures for static LMPTs and Sahni, Kim, and Lu [17] review data structures for both static and dynamic LMPTs. Several trie-based data structures for LMPTs have been proposed [18, 1, 2, 10, 19, 12, 13]. Structures such as that of [18] perform each of the dynamic router-table operations (lookup, insert, delete) in $O(W)$ time. Others (e.g., [1, 2, 10, 19, 12, 13]) attempt to optimize lookup time and memory requirement through an expensive preprocessing step. These structures, while providing very fast lookup capability, have a prohibitive insert/delete time and so, they are suitable only for static router-tables (i.e., tables into/from which no inserts and deletes take place).

Waldvogel et al. [21] have proposed a scheme that performs a binary search on hash tables organized by prefix length. This binary-search scheme has an expected complexity of $O(\log W)$ for lookup. An alternative adaptation of binary search to longest-prefix matching is developed in [7]. Using this adaptation, a lookup in a table that has n prefixes takes $O(W + \log n) = O(W)$ time. Because the schemes of [21] and [7] use expensive precomputation, they are not suited for a dynamic router-tables.

Suri et al. [20] have proposed a B-tree data structure for dynamic LMPTs. Using their structure, we may find the longest matching-prefix, $lmp(d)$, in $O(\log n)$ time. However, inserts/deletes take $O(W \log n)$ time. 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 [20] takes $O(\log W \log n)$ time for an insert and $O(W + \log n) = O(W)$ time for a delete. The number of cache misses that occur when the data structure of [20] is used is $O(\log n)$ per opertion.

Sahni and Kim [14, 15] develop data structures, called a collection of red-black trees (CRBT) and alternative collection of red-black trees (ACRBT), that support the three operations of a dynamic LMPT in $O(\log n)$ time each. The number of cache misses is also $O(\log n)$. In [15], Sahni and Kim show that their ACRBT structure is easily modified to extend the biased-skip-list structure of Ergun et al. [3] so as to obtain a biased-skip-list structure for dynamic LMPTs. Using this modified biased skip-list structure, lookup, insert, and delete can each be done in $O(\log n)$ expected time and $O(\log n)$ expected cache misses. Like the original biased-skip list structure of [3], the modified structure of [15] adapts so as to perform lookups faster for bursty access patterns than for non-bursty patterns. The ACRBT structure may also be adapted to obtain a collection of splay trees structure [15], which performs the three dynamic LMPT operations in $O(\log n)$ amortized time and which adapts to provide faster lookups for bursty traffic.

Lu and Sahni [8] use priority search trees to arrive at an $O(\log n)$ data structure for dynamic prefix-tables. This structure is faster than the CRBT structure of [14, 15]. Lu and Sahni [8] also propose a data structure that employs priority search trees and red-black trees for the representation of rule tables in which the filters are a conflict-free set of ranges. This data structure permits most-specific-range matching as well as range insertion and deletion to be done in $O(\log n)$ time each.

In [9], Lu and Sahni develop a data structure called BOB (binary tree on binary tree) for dynamic router-tables in which the rule filters are nonintersecting ranges and in which ties are broken by selecting the highest-priority rule that matches a destination address. Using BOB, the highest-priority rule that matches a destination address may be found in $O(\log^2 n)$ time; a new rule may be inserted and an old one deleted in $O(\log n)$ time. Related structures PBOB (prefix BOB) and LMPBOB (longest matching-prefix BOB) are proposed for highest-priority prefix matching and longest-matching prefixes. These structures apply when all filters are prefixes. The data structure LMPBOB permits longest-prefix matching in $O(W)$ time; rule insertion and deletion each take $O(\log n)$ time. On practical rule tables, BOB and PBOB perform each of the three dynamic-table operations in $O(\log n)$ time and with $O(\log n)$ cache misses. The number of cache misses incurred by LMPBOB is also $O(\log n)$.

When an HPPT (highest-priority prefix-table) is represented as a binary trie [5], each of the three dynamic HBRT operations takes $O(W)$ time and cache misses.

Gupta and McKeown [4] have developed two data structures for dynamic HPRTs—heap on trie (HOT) and binary search tree on trie (BOT). The HOT structure takes $O(W)$ time for a lookup and $O(W \log n)$ time for an insert or delete. The BOT structure takes $O(W \log n)$ time for a lookup and $O(W)$ time for an insert/delete. The number of cache misses in a HOT and BOT is asymptotically the same as the time complexity of the corresponding operation.

In this paper, we focus on B-tree data structures for dynamic NHPRTs and LMPTs. We are interested in the B-tree, because by varying the order of the B-tree, we can control the height of the tree and hence control the number of cache misses incurred when performing a rule-table operation. Although Suri et al. [20] have proposed a B-tree data structure for dynamic prefix-tables, their structure has the following shortcomings:

1. A prefix may be stored in $O(m)$ nodes at each level of the order m B-tree. This results in excessive cache misses during the insert and delete operations.

2. Some of prefix end-points are stored twice in the B-tree. This is because every endpoint is stored

Preifx Name	Prefix	Range Start	Range Finish
P1	001*	4	7
P2	00*	0	7
P3	1*	16	31
P4	01*	8	15
P5	10111	23	23
P6	0*	0	15

Figure 1: An example prefix set R ($W = 5$)

in a leaf node and some of the endpoints are additionally stored in interior nodes. This duplicity in end-point storage increases memory requirement.

Our proposed B-tree structure doesn't suffer from these shortcomings. In our structure, each prefix is stored in $O(1)$ nodes at each level, and each prefix end-point is stored once. Consequently, even though the asymptotic complexity of performing dynamic prefix-table operations is the same in both structures and the asymptotic memory requirements of both are the same, our structure is faster for the insert and delete operations and also takes less memory.

In Section 2, we develop our B-tree data structure, PIBT (prefix in B-tree), for dynamic prefix-tables. Our B-tree structure for non-intersecting ranges, RIBT (range in B-tree), is developed in Section 3. Experimental results comparing the performance of our PIBT structure, the multiway range tree (MRT) structure of Suri et al. [20], and the best binary tree structure for dynamic prefix-tables, PBOB [9], are presented in Section 4.

2 Longest-Matching Prefix-Tables—LMPT

2.1 The Prefix In B-Tree Structure—PIBT

A **range** $r = [u, v]$ is a pair of addresses u and v, $u \leq v$. The range r represents the addresses $\{u, u+1, ..., v\}$. $start(r) = u$ is the start point of the range and $finish(r) = v$ is the finish point of the range. The range r **matches** all addresses d such that $u \leq d \leq v$. Every prefix of a prefix router-table may be represented as a range. For example, when $W = 5$, the prefix $p = 100*$ matches addresses in the range $[16, 19]$. So, we say $p = 100* = [16, 19]$, $start(p) = 16$, and $finish(p) = 19$. The length of p is 3. Figure 1 shows a prefix set and the ranges of the prefixes.

The set of start and finish points of a collection P of prefixes is the set of **endpoints**, $E(P)$, of P. When $|P| = n$, $|E(P)| \leq 2n$. Although our PIBT structure and the MRT structure of [20] (MRT) store

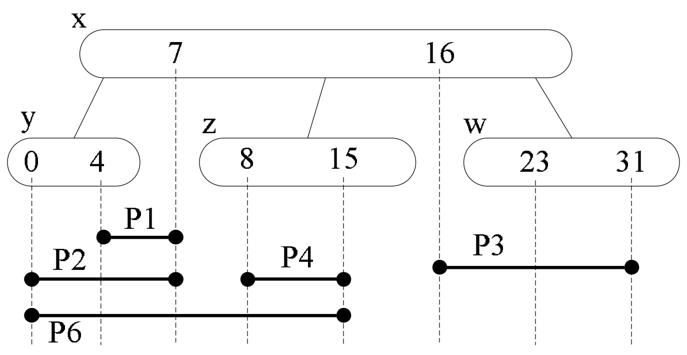

Figure 2: B-tree for the endpoints of the prefixes of Figure 1

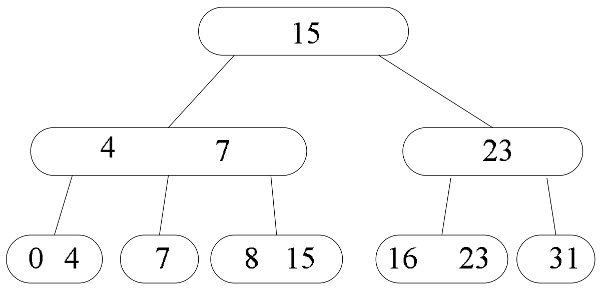

Figure 3: Alternative B-tree for Figure 1

the endpoints $E(P)$ together with additional information in a B-tree[4] [16], each structure uses a different variety of B-tree. Our PIBT structure uses a B-tree in which each key (endpoint) is stored exactly once, while the MRT uses a B-tree in which each key is stored once in a leaf node and some of the keys are additionally stored in interior nodes. Figure 2 shows a possible order-3 B-tree for the endpoints of the prefix set of Figure 1. In this example, each endpoint is stored in exactly one node. This example B-tree is a possible B-tree for PIBT but not for MRT.

Figure 3 shows a possible order 3 B-tree in which each endpoint is stored in exactly one leaf node and some endpoints are also stored in interior nodes. This example B-tree is a possible B-tree for MRT but not for PIBT.

With each node x of a PIBT B-tree, we associate an interval $int(x)$ of the destination address space $[0, 2^W - 1]$. The interval $int(root)$ associated with the root of the B-tree is $[0, 2^W - 1]$. Let x be a B-tree node that has t keys. The format of this node is:

$$t, child_0, (key_1, child_1), \cdots, (key_t, child_t)$$

[4]A **B-tree of order** m is an m-way search tree. If the B-tree is not empty, the root has at least two children and other internal nodes have at least $\lceil m/2 \rceil$ children. All external nodes are at the same level.

where key_i is the ith key in the node ($key_1 < key_2 < \cdots < key_t$) and $child_i$ is a pointer to the ith subtree. In case of ambiguity, we use the notation $x.key_i$ and $x.child_i$ to refer to the ith key and child, respectively, of node x. Let $key_0 = start(int(x))$ and $key_{t+1} = finish(int(x))$. By definition,

$$int_i(x) = int(child_i) = [key_i, key_{i+1}], 0 \le i \le t$$

. For the example B-tree of Figure 2, $int(x) = [0, 31]$, $int_0(x) = int(y) = [0, 7]$, $int_1(x) = int(z) = [7, 16]$, $int_2(x) = int(w) = [16, 31]$, $int_0(y) = [0, 0]$, $int_1(y) = [0, 4]$, $int_2(y) = [4, 7]$, and $int_0(z) = [7, 8]$.

Node x of a PIBT has $t + 1$ W-bit vectors $x.interval_i$, $0 \le i \le t$ and t W-bit vectors $x.equal_i$, $1 \le i \le t$. The lth bit of $x.interval_i$, denoted $x.interval_i[l]$ is 1 iff there is a length l prefix whose range includes $int_i(x)$ but not $int(x)$. This rule for the *interval* vectors is called the **prefix allocation rule**. For our example of Figure 2, $y.interval_2[3] = 1$ because prefix P1 has length 3 and range [4,7]; [4,7] includes $int_2(y) = [4, 7]$ but not $int(y) = [0, 7]$. We say that P1 is stored in $y.interval_2$ and in node y. It is easy to see that a prefix may be stored in up to $m - 1$ intervals of an order m B-tree node and in up to 2 nodes at each level of the B-tree.

The bit $x.equal_i[l]$ is 1 iff there is a length l prefix that has a start or finish endpoint equal to key_i of x. For our example, prefixes P2 and P6 have 0 as their start endpoint. Since the length of P2 is 2 and that of P6 is 1, $y.equal_1[1] = y.equal_1[2] = 1$; all other bits of $y.equal_1$ are 0.

To conserve space, leaf nodes do not have child pointers. Further, to reduce memory accesses, child pointers and interval bit-vectors are interleaved so that $child_i$ and $interval_i$ can be accessed with a single cache miss provided cache lines are long enough. In the sequel, we assume that W is sufficiently small so that this is the case. Further, we assume that bit-vector operations on W-bit vectors take $O(1)$ time. This assumption is certainly valid for IPv4 where $W = 32$ and a W-bit vector may be represented as a 4-byte integer.

2.2 Finding The Longest Matching-Prefix

As in [20], we determine only the length of the longest prefix that matches a given destination address d. From this length and d, the longest matching-prefix, $lmp(d)$, is easily computed. The PIBT search algorithm (Figure 4) employs the following lemma.

Lemma 1 *Let P be a set of prefixes. If P contains a prefix whose start or finish endpoint equals d, then the longest prefix, $lmp(d)$, that matches d has its start or finish point equal to d.*

Proof Let $p \in P$ be a prefix that matches d and whose start or finish endpoint equals d. Let $q \in P$ be a prefix that matches d but whose start and finish endpoints are different from d. It is easy to see that the range of p is properly contained in the range of q. Therefore, p is a longer prefix than q. So, $lmp(d) \neq q$. The lemma follows. ■

The PIBT search algorithm first constructs a W-bit vector *matchVector*. When the router table has no prefix whose start or finish endpoint equals the destination address d, the constructed bit vector satisfies $matchVector[l] = 1$ iff there is a length l prefix that matches d. Otherwise, $matchVector[l] = 1$ iff there is a length l prefix whose start or finish endpoint equals d. The maximum l such that $matchVector[l] = 1$ is the length of $lmp(d)$.

```
Algorithm $lengthOflmp(d)$ {
    // return the length of the longest prefix that matches $d$
    $matchVector = 0$;
    $x$ = root of PIBT;
    while ($x \neq$ null) {
        Let $t$ = number of keys in $x$;
        Let $i$ be the smallest integer such that $x.key_{i-1} \leq d \leq x.key_i$, $1 \leq i \leq t+1$;
        if ($i \leq t$ && $d == x.key_i$) {
            $matchVector = x.equal_i$;
            break;}
        $matchVector \mid= x.interval_{i-1}$;
        $x = x.child_{i-1}$;
    }
    return Largest $l$ such that $matchVector[l] = 1$;
}
```

Figure 4: Algorithm to find the length of $lmp(d)$

Complexity Analysis

Each iteration of the **while** loop takes $O(\log_2 m)$ time (we assume throughout this paper that, for sufficiently large m, a B-tree node is searched using a binary search) and the number of iterations is $O(\log_m n)$. The largest l such that $matchVector[l] = 1$ may be found in $O(\log_2 W)$ time by performing $O(\log_2 W)$ operations on the W-bit vector *matchVector*. So, the overall complexity is $O(\log_2 n + \log_2 W)$ $= O(\log_2(nW))$. The number of nodes accessed by the algorithm is $O(\log_m n)$.

2.3 Inserting A Prefix

To insert a prefix p, we must do the following:

1. Insert $start(p)$ into the PIBT and update the corresponding *equal* vector. If $start(p)$ is already in the PIBT, only the corresponding *equal* vector is to be updated.

2. Insert $finish(p)$ (provided, of course, that $finish(p) \neq start(p)$) into the PIBT and update the corresponding *equal* vector. If $finish(p)$ is already in the PIBT, only the corresponding *equal* vector is to be updated.

3. Update the *interval* vectors so as to satisfy the prefix allocation rule.

2.3.1 Inserting an Endpoint

The algorithm to insert an endpoint u into the PIBT is an adaptation of the standard B-tree insertion algorithm [16]. We search the PIBT for a key equal to u. In case case u is already in the PIBT, the associated *equal* vector is updated to account for the new prefix p that begins or ends at u and whose length equals $length(p)$ (note that if the $length(p)$ bit of the equal vector is already 1, the prefix p is a duplicate prefix).

If u is not in the PIBT, the search for u terminates at a leaf x of the PIBT. Let t be the number of keys in x. The endpoint u is inserted into node x between key_{i-1} and key_i, where $key_{i-1} < u < key_i$. The key sequence in the node becomes

$$key_1, \cdots, key_{i-1}, u, key_i, \cdots, key_t$$

the *interval* vector sequence becomes

$$interval_0, \cdots, interval_{i-1}, interval_{i-1}, interval_i, \cdots, interval_t$$

and the *equal* vector associated with u has a 1 only for the position $length(p)$. Notice that the insertion of u splits the old int_{i-1} into the two intervals $[start(int_{i-1}), u]$ and $[u, finish(int_{i-1})]$. Further, the original bit vector $interval_{i-1}$ is the interval vector for each of these two intervals (we don't account for the new prefix p at this time). The orignal W-bit interval vector $interval_{i-1}$ may be replicated in $O(1)$ time so that we have separate copies of the interval vector for each of the two new intervals.

When $t < m-1$, the described insertion of u, the creation of the *equal* vector for u, and the replication of $interval_{i-1}$ together with an incrementing of the key count t for x completes the insertion of u. When, $t = m-1$, the described operations on x yield a node that has 1 key more than its capacity $m-1$. The format of node x at this time is

$$m, key_1, \cdots, key_m, interval_0, \cdots interval_m$$

(the $child_i$ pointers and $equal_i$ vectors are not shown). Node x is split into two [16] around key_g, where $g = \lceil m/2 \rceil$. Keys to the left of key_g (along with the associated *equal* and *interval* vectors remain in node x), those to the right are placed into a new node y, and the tuple $(key_g, equal_g, y)$ inserted into the parent of x. Let x' denote the new x. x' has $g-1$ keys while y has $m-g$ keys. The formats of x' and y are

$$x': \quad g-1, key_1, \cdots, key_{g-1}, interval_0, \cdots interval_{g-1}$$

$$y: \quad m-g, key_{g+1}, \cdots, key_m, interval_g, \cdots interval_m$$

Before proceeding to insert the tuple $(key_g, equal_g, y)$ into the parent of x, we need to adjust the *interval* vectors in x and y to account for the fact that $int(x')$ and $int(y)$ are not the same as $int(x)$. Prefixes like $r1$ that include the range $int(x') = [start(int(x)), key_g]$ need to be removed from the *intervals* of x' and inserted into an *interval* vector in the parent node; those like $r2$ that include the range $int(y) = [key_g, finish(int(x))]$ need to be removed from the *intervals* of y and inserted into a parent *interval* (see Figure 5).

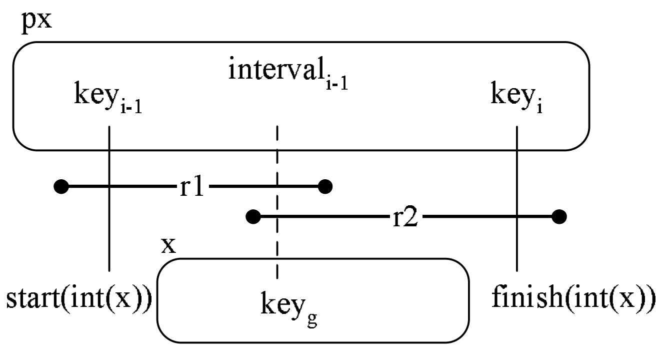

Figure 5: Node splitting

Algorithm *insertEndPoint*(u, x, i) (Figure 6) inserts the endpoint u into the leaf node x of the PIBT and performs node splits as needed. It is assumed that $x.key_{i-1} < u < x.key_i$. The method

$$matchingPrefixVector(s, f, intervalVector)$$

returns a bit vector that contains the prefixes of *intervalVector* that include/match the range $[s, f]$.

Complexity Analysis

The time to find x and i prior to invoking *insertEndPoint* is the time, $O(\log_2 n)$, to search unsuccessfully for u in the PIBT. Each time a node splits, $O(\log_2 W)$ time is spent computing *leftBits* and *rightBits* (we assume that each operation on a W-bit vector takes $O(1)$ time) and an additional $O(m)$ time is spent updating x and creating y. So, each iteration of the **do** loop takes $O(m + \log_2 W)$ time. Since the height of the PIBT tree is $O(\log_m n)$, the total time needed to insert an endpoint is $O((m + \log_2 W) \log_m n)$; the number of nodes accessed is $O(\log_m n)$.

```
$AlgorithminsertEndPoint(u, x, i)$ {
    //insert $u$, $u = start(p)$ or $u = finish(p)$ into
    //the leaf node $x$, $x.key_{i-1} < u < x.key_i$
    $leftBits = rightBits = 0$; // carry over prefixes from children
    $child = null$; // right child of $u$
    $eq = equal$ bit-vector with 1 at position $length(p)$; // $p$ is insert prefix
    do {
        right shift the keys, child pointers, equal vectors, and interval
            vectors of $x$ by 1 beginning with those at position $i$;
        insert $u$ as the new $key_i$;
        $equal_i = eq$;
        $child_i = child$;
        $interval_i = interval_{i-1}|rightBits$;
        $interval_{i-1}| = leftBits$;
        if ($x$ has less than $m$ keys) return;
        // node overflow, split $x$
        $g = \lceil m/2 \rceil$;
        $keyg = key_g$;
        Split $x$ into the new $x$ (i.e., $x'$) and $y$ as described above;
        // adjust interval vectors in $x$ and $y$
        $leftBits = matchingPrefixVector(start(int(x)), u, x.interval_0)$;
        for ($j = 0; j \leq g; j++$)
            $x.interval_j$ &=~$leftBits$; // remove from $x.interval_j$
        $rightBits = matchingPrefixVector(u, finish(int(y)), y.interval_0)$;
        for ($j = 0; j <= m - g; j++$)
            $y.interval_j$ &=~$rightBits$; // remove from $y.interval_j$
        $u = keyg$; $child = y$;
        $eq = equal$ vector of $key_g$;
        $x = parent(x)$;
        Set $i$ so that $x.key_{i-1} < u < x.key_i$;
    } while ($x != root$);
    // create a new root
    New root $r$ has a single key $u$ with $equal$ vector $eq$;
    $r.child_0$ = old root;
    $r.child_1 = child$;
    $r.interval_0 = leftBits$;
    $r.interval_1 = rightBits$;
}
```

Figure 6: Insert a new endpoint into the PIBT

2.3.2 Update Interval Vectors

Following the insertion of the endpoints of the new prefix p, the interval vectors in the nodes of the PIBT need to be updated to account for the new prefix p. The prefix allocation rule leads to the interval update

```
**Algorithm** $updateIntervals(p, x)$ {
    // use prefix allocation rule to place $p$ in proper nodes
    **if** ($x == null$ or ($int(x)$ and $p$ have at most one common address)
        or ($int(x) \subseteq p$))**return**;
    Let $t$ be the number of keys in $x$;
    $key_0 = start(int(x))$; $key_{t+1} = finish(int(x))$;
    Let $i$ be the smallest integer such that $key_i \geq start(p)$;
    Let $j$ be the largest integer such that $key_j \leq finish(p)$;
    $x.interval_q[length(p)] = 1, i \leq q < j$;
    **if** ($i > 0$) $updateIntervals(p, child_{i-1})$;
    **if** ($j <= t$ && $j \neq i - 1$) $updateIntervals(p, child_j)$;
}
```

Figure 7: Update intervals to account for prefix p

algorithm of Figure 7. On initial invocation, x is the root of the PIBT. The interval update algorithm assumes that p is not the default prefix $*$ that matches all destination addresses (this prefix, if present, may be stored outside the PIBT and handled as a special case).

Figure 8 shows a possible set of nodes visited by x.

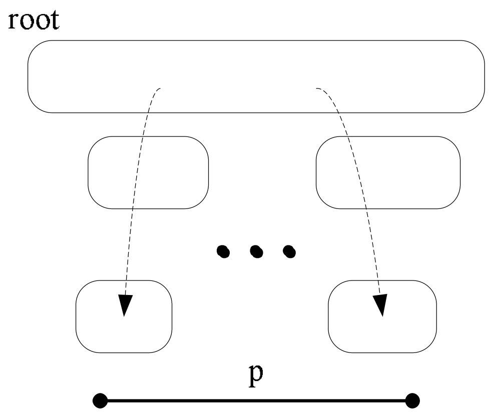

Figure 8: Nodes visited when updating *intervals*

Complexity Analysis

Algorithm *updateIntervals* visits at most 2 nodes on each level of the PIBT and at each node, $O(m)$ time is spent. So, the complexity of *updateIntervals* is $O(m \log_m n)$. This algorithm accesses $O(\log_m n)$ nodes of the PIBT.

Combining the complexities of all parts of the algorithm to insert a prefix, we see that the time to insert is $O((m + \log_2 W) \log_m n)$ and that the number of nodes accessed during a prefix insertion is

$O(\log_m n)$.

2.4 Deleting A Prefix

To delete a prefix p, we do the following.

1. Remove p from all *interval* vectors that contain p.
2. Update the *equal* vector for $start(p)$ and remove $start(p)$ from the PIBT if its *equal* vector is now zero.
3. If $start(p) \neq finish(p)$, update the *equal* vector for $finish(p)$ and remove $finish(p)$ from the PIBT if its *equal* vector is now zero.

The first of these steps (i.e., removing p from *interval* vectors) is almost identical to the corresponding step (Figure 7) for prefix insertion. The only difference is that instead of setting $x.interval_q[length(p)]$ to 1, we now set it to 0. The time complexity of this step remains $O(m \log_m n)$ and this step accesses $O(\log_m n)$ nodes of the PIBT.

2.4.1 Deleting an Endpoint

The B-tree key deletion algorithm of [16] considers two cases:

1. The key to be deleted is in a leaf node.
2. The key to be deleted is an interior (i.e., non-leaf) node.

Deleting from a Leaf Node

To delete the endpoint u, we first search the PIBT for the node x that contains this endpoint. Suppose that x is a leaf of the PIBT and that $u = x.key_i$. Since u is an endpoint of no prefix, $x.interval_{i-1} = x.interval_i$ and $x.equal_i = 0$. key_i, $x.interval_i$, $x.equal_i$, and $x.child_i$ are removed from node x and the keys to the right of key_i together with the associated *interval*, *equal*, and *child* values are shifted one position left. If the number of keys that remain in x is at least $\lceil m/2 \rceil$ (2 in case x is the root), we are done. Otherwise, node x is deficient and we do the following

1. If a nearest sibling of x has more than $\lceil m/2 \rceil$ keys, x gains/borrows a key via this nearest sibling and so is no longer deficient.
2. Otherwise, x merges with its nearest sibling. The merge may cause $px = parent(x)$ to become deficient in which case, this deficiency resolution process is repeated at px.

Borrow from a Sibling

Suppose that x's nearest left sibling y has more than $\lceil m/2 \rceil$ keys (Figure 9). Let $key_{t(y)}$ be the largest (i.e., rightmost) key in y and let $px.key_i$ be the key in px such that $px.child_{i-1} = y$ and $px.child_i = x$ (i.e., $px.key_i$ is the key in px that is between y and x).

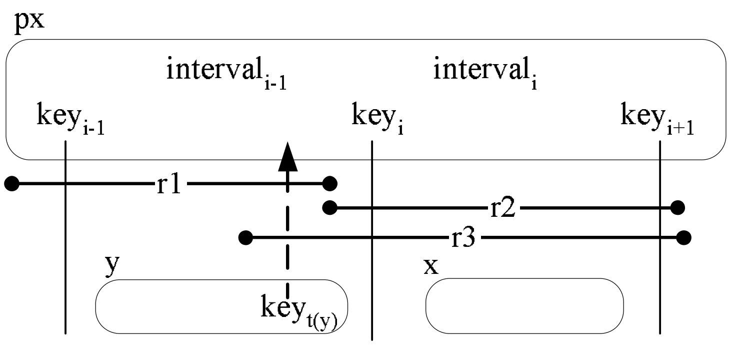

Figure 9: Borrow from right sibling

The borrow operation does the following:

1. In px, key_i and $equal_i$ are replaced by $key_{t(y)}$ and its associated $equal$ vector.

2. In x, all keys and associated vectors and child pointers are shifted one right; $y.child_{t(y)}$, $y.interval_{t(y)}$, $px.key_i$, and $px.equal_i$, respectively, become $x.child_0$, $x.interval_0$, $x.key_0$, $x.equal_0$.

3. From the *intervals* of y, remove the prefixes that include the range $[px.key_{i-1}, key_{t(y)}]$ and add these removed prefixes to $px.interval_{i-1}$.

4. From $px.interval_i$, remove those prefixes that do not include the range $[key_{t(y)}, px.key_{i+1}]$ and add these removed prefixes to the *intervals* of x other than $x.interval_0$.

5. To $x.interval_0$ (formerly $y.interval_{t(y)}$) add all pefixes originally in $px.interval_{i-1}$. Next, remove from $x.interval_0$, those prefixes that contain the range $[key_{t(y)}, px.key_{i+1}]$. Since these removed prefixes are already included in $px.interval_i$, they are not to be added again.

One may verify that following the borrow operation, we have a properly structured PIBT. Further, since the prefixes of $interval_i$ that do not include a given range may be found in $O(\log_2 W)$ time using a binary search on prefix length, the time complexity of the borrow operation is $O(m + \log_2 W)$ and the borrow operation accesses 3 nodes.

Merging Two Adjacent Siblings

When node x is deficient and its nearest sibling y has exactly $\lceil m/2 \rceil - 1$ keys, nodes x, y and the in-between key, $px.key_i$, in the parent px of x are combined into a single node. The resulting single node has $2\lceil m/2 \rceil - 2$ keys. Figure 10 shows the situation when y is the nearest right sibling of x.

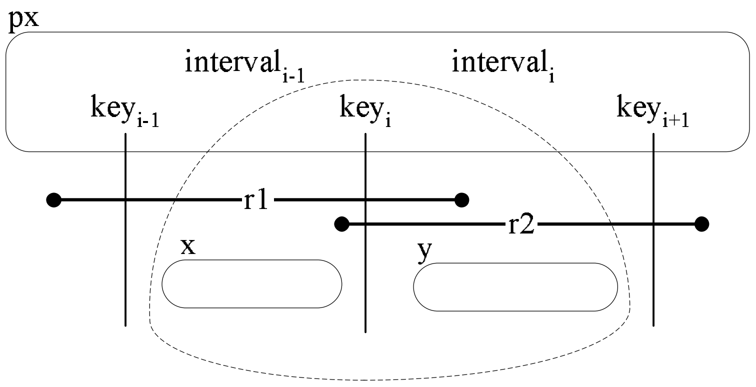

Figure 10: Merge two nodes

The steps in the merge of x and y are:

1. The prefixes in $px.interval_{i-1}$ that do not include the range $[px.key_{i-1}, px.key_{i+1}]$ are removed from $px.interval_{i-1}$ and added to the *interval*s of x.

2. The prefixes in $px.interval_i$ that do not include the range $[px.key_{i-1}, px.key_{i+1}]$ are added to the *interval*s of y. $px.interval_i$ is removed from px.

3. Remove $px.key_i$ and its associated *equal* vector from px and append to the right of x. Next, append the contents of y to the right of the new x.

Since the merging of x and y reduces the number of keys in px by 1, px may become deficient. If so, the borrow/merge process is repeated at px. In this way, the deficiency may be propogated all the way up to the root. In case the root becomes deficient it has no keys and so is discarded.

Complexity Analysis

Since the prefixes of $interval_i$ that do not include a given range may be found in $O(\log_2 W)$ time using a binary search on prefix length, two nodes may be merged in $O(m + \log_2 W)$ time; the number of nodes accessed during the merge is 3.

Deleting from a Non-leaf Node

To delete the endpoint $u = x.key_i$ from the non-leaf node x, u is replaced by either the largest key in the subtree $x.child_{i-1}$ or the smallest key in the subtree $x.child_i$ [16]. Let $y.key_{t(y)}$ be the largest key in the subtree $x.child_{i-1}$ (Figure 11).

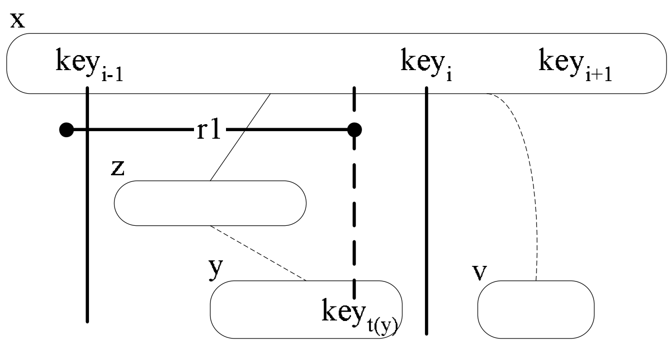

Figure 11: Deleting $x.key_i$

When u is replaced by $y.key_{t(y)}$, it is necessary also to replace $x.equal_i$ by $y.equal_{t(y)}$. Before proceeding to remove $y.key_{t(y)}$ from the leaf node y, we need to adjust the *interval* values of nodes on the path from x to y. Let z, $z \neq x$, be a node on the path from x to y. As a result of the relocation of $y.key_{t(y)}$, $int(z)$ shrinks from $[start(int(z)), u]$ to $[start(int(z)), y.key_{t(y)}]$. So, prefixes that include the range $[start(int(z)), key_{t(y)}]$ but not the range $[start(int(z)), u]$ are to be removed from the *interval*s of z and added to the parent of z. Since, there are no endpoints between $y.key_{t(y)}$ and $u = x.key_i$, these prefixes that are to be removed from the *interval*s of z must have $y.keyt(y)$ as an endpoint (in particular, these prefixes finish at $y.key_{t(y)}$). Hence, these prefixes are in $y.equal_{t(y)}$, and so, the number of these prefixes is $O(W)$. As we retrace the path from y to x, the bit vectors for the set of prefixes to be removed from each node may be constructed in total time $O(\log_m n + W)$. As it takes $O(m)$ time to remove the desired prefixes from the intervals of each node z and to add to the parent of z, the total time needed to update the *interval* values for all nodes on the path from x to y (including nodes x and y) is $O(m \log_m n + W)$.

Let v be the leftmost leaf node in the subtree $x.child_i$. For each node z, $z \neq x$, on the path from x to v, $z.int$ expands from $[u, finish(int(z))]$ to $[y.key_{t(y)}, finish(int(z))]$. Since there is no prefix that has u as its endpoint and since there are no endpoints between u and $y.key_{t(y)}$, no *interval* vectors on the path from x to v are to be changed.

Complexity Analysis

Adding together the complexity of each step of the deletion algorithm, we get $O((m+\log_2 W)\log_m n+W)$

as the overall time complexity of the delete operation. The number of node accesses is $O(\log_m n)$.

The time complexity of the delete operation becomes $O(m \log_m n + W)$ when the search for prefixes that do not match a given range (this is done when two adjacent siblings are merged) is done by a sequential rather than a binary search on prefix length. For example, consider the situation shown in Figure 12. The nodes $x1$ and $y1$ are merged causing $x2$ to become deficient. This merge is followed by the merging of nodes $x2$ and $y2$. To find the prefixes, in a parent interval $x2.interval_j$, $j = 1$ or 2, that do not match $[key_1, key_3]$, we search serially for the least q (in the order $q = W-1, W-2, \cdots, 1$ and stopping as soon as all lower index bits of $interval_j$ are determined to be 0) such that $x2.interval_j[q] = 1$ and the corresponding length q prefix does not match $[x2.key_1, x2.key_3]$. Let this least q be q'. When repeating this search for non-matching prefixes at the parent $x3$ of $x2$, we may start with $q = q' - 1$, because all prefixes in the $x3.interval_j$s, $j = 6$ and 7, that are to be searched are shorter than q'.

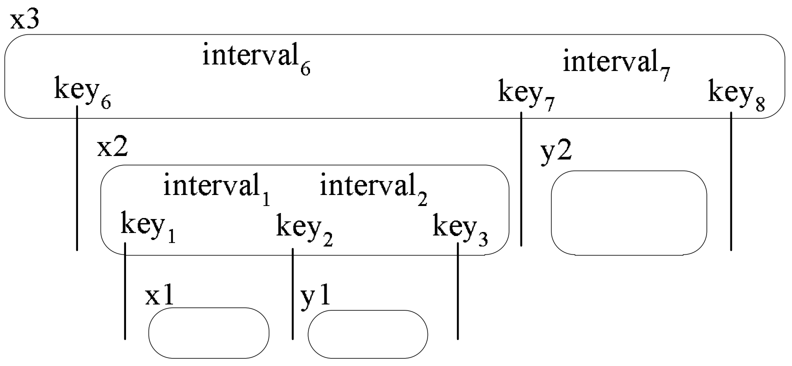

Figure 12: Merging adjacent siblings

Unfortunately, this serial search strategy for the non-matching prefixes cannot be adapted to find, in $O(W)$ time, all the matching prefixes required during the insert operation.

2.5 Cache-Miss Analysis

The number of cache misses that occur during the lookup operation is approximately the same for PIBTs and MRTs. A worst-case lookup will examine $\log_{m/2} n$ nodes. If a binary search is used in each examined node to determine which subtree to move to, the examination of each node will cause about $\log_2(mW/(8b))$ cache misses (b is the size, in bytes, of a cache line). So, the worst-case number of cache misses for a lookup is about $\log_2(mW/(8b)) \log_{m/2} n$. When $W = 32$ (IPv4), $b = 32$ bytes (as it is on a PC) and $m = 32$, this works out to $0.5 \log_2 n$ cache misses. In the case of the PBOB structure of [9], each node fits into a cache line. So, the worst-case number of cache misses equals the worst-case height,

$2\log_2 n$, of the underlying red-black tree.

For the insert operation, we count only the number of read misses (since write misses are non-blocking, these do not affect performance as much as the read misses do). Let s be the size, in number of cache lines, of an MRT node. To split a node, we must read at least the right half of that node. For simplicity, we assume that the entire node is read. With this assumption, our cache-miss count will also account for cache misses that occur on the downward search pass of an insert operation. The total number of nodes that get split during an insert may be as high as h, where h is the height of the MRT. So, the worst-case number of cache misses exclusive of those needed to update information in nodes not accessed by the search and split steps is hs. Besides maintaining the B-tree properties of the MRT, an insert must update the span vector (defined in [20]) stored in each of the children of a node that gets split. This requires $(m-1)h \approx mh$ span vectors to be updated at an additional cost of mh cache misses (each span vector is assumed to fit in a cache line). So, the worst-case number of cache misses is approximately, $h(s+m)$.

The nodes of the PIBT structure are approximately twice as large as those of the MRT. Since the worst-case heights of the PIBT and MRT are almost the same, the number of cache misses during the downward search pass and the upward node-split pass is at most $2hs$. No new nodes are accessed to update *interval* vectors. So, $2hs$ is a bound for the entire insert operation. Since $s \approx 8m/b$, the ratio of the worst-case misses for MRT and PIBT is approximately $(b+8)/16$. When $b = 32$ (as it is for a PC), this ratio is 2.5. That is, the MRT will make 2.5 times as many cache misses, in the worst-case, as will the PIBT during an insert operation.

The PBOB of [9] makes $2\log_2 n$ cache misses during a worst-case insert. Since, $2hs \approx 16m/b \log_{m/2} n = 4\log_2 n$, when $m = b = 32$, an order 32 PIBT makes twice as many cache misses during a worst-case insert as does the PBOB.

The analysis for the delete operation is almost identical, and the cache-miss counts are the same as for the insert operation.

3 Highest-Priority Range-Tables

In this section, we extend the PIBT structure to obtain a B-tree-based data structure called RIBT (range in B-tree). The RIBT structure is for dynamic router-tables whose filters are non-intersecting ranges.

3.1 Preliminaries

Definition 1 *Let r and s be two ranges. Let $overlap(r, s) = r \cap s$.*

(a) r and s are **disjoint** *iff $overlap(r, s) = \emptyset$.*

(b) r and s are **nested** *iff one range is contained within the other. That is, iff $overlap(r, s) = r$ or $overlap(r, s) = s$.*

(c) r and s **intersect** *iff $r \cap s \neq \emptyset \wedge r \cap s \neq r \wedge r \cap s \neq s$.*

Notice that two ranges intersect iff they are neither disjoint nor nested.

$[4, 4]$ is the overlap of $[2, 4]$ and $[4, 6]$; and $overlap([3, 8], [2, 4]) = [3, 4]$. [2, 4] and [6, 9] are disjoint; [2,4] and [3,4] are nested; [2,4] and [2,2] are nested; [2,8] and [4,6] are nested; [2,4] and [4,6] intersect; and [3,8] and [2,4] intersect.

Definition 2 *The range set R is nonintersecting iff $disjoint(r, s) \vee nested(r, s)$ for every pair of ranges r and $s \in R$. Equivalently, iff no two ranges of R intersect.*

Notice that every prefix set defines a nonintersecting range set (Figure 1).

Lemma 2 *Let R be a nonintersecting range set. Assume that $start(r) < finish(r)$ for every range $r \in R$. Let $s = min\{start(r)|r \in R\}$, $f = max\{finish(r)|r \in R\}$.*

(a) R has at least $|R| + 1$ distinct endpoints.

(b) If R has $|R| + 1$ distinct endpoints, then $[s, f] \in R$.

Proof We prove part (a) by induction. If $|R| = 1$, the claim is true since R has two endpoints. Assume that the claim is true whenever $|R| \leq n - 1$. Consider the case $|R| = n$. Let s be the smallest start point of the ranges in R and f the largest finish point. We consider two cases.

Case 1. $[s, f] \notin R$. In this case, R can be divided into two nonempty disjoint sets, $R1$ and $R2$, of nonintersecting ranges. Since $start(r) < finish(r)$ for every range in R, this property holds also for the ranges in $R1$ and $R2$. From the induction hypothesis, it follows that $R1$ has at least $|R1| + 1$ distinct endpoints and $R2$ has at least $|R2| + 1$ distinct endpoints. Since $R1$ and $R2$ are disjoint, R has at least $(|R1| + 1) + (|R2| + 1) = |R| + 2$ distinct endpoints.

Case 2. $[s, f] \in R$. This case breaks into two sub-cases.

Case 2.1. In $R - \{[s, f]\}$, there is a range that starts at s and another range that finishes at f. Now, $R - \{[s, f]\}$ can be divided into two non-empty disjoint sets as in case 1. So, $R - \{[s, f]\}$ has at least $|R| + 1$ distinct endpoints.

Case 2.2. In $R - \{[s, f]\}$, there is no range that starts at s or there is no range that finishes at f. Consider the former case. From the induction hypothesis, it follows that $R - \{[s, f]\}$ has at least $|R|$ distinct endpoints. Since R has all the distinct endpoints in $R - \{[s, f]\}$ plus the endpoint s, R has at least $|R| + 1$ distinct endpoints. The proof for the latter case is similar.

Part (b) follows from the proof of Case 1. If $[s, f] \notin R$, R has at least $|R| + 2$ distinct endpoints. ■

3.2 The Range In B-Tree Structure—RIBT

The RIBT is an extension of the PIBT structure to the case of an NHPRT. As in the PIBT structure, we maintain a B-tree of distinct range-endpoints. Let x be a node of the RIBT B-tree. $x.int$ and $x.int_i$ are defined as for the case of the PIBT B-tree. With each endpoint $x.key_i$ in node x, we keep a max-heap, $equalH_i$, of ranges that have $x.key_i$ as an endpoint. As in the case of the PIBT, the default range $[0, 2^W - 1]$ isn't stored in the RIBT. Ranges r with $start(r) = finish(r)$ are stored only in the appropriate $equalH$ heap.

Other ranges are stored in $equalH$ heaps as well as in $intervalH$ max-heaps, which are the counterpart of the $interval$ vectors used in PIBT. An RIBT node that has t keys has t $intervalH$ max-heaps. The ranges stored in these max heaps are determined by a range allocation rule that is similar to the prefix allocation rule employed by a PIBT—a range r is stored in an $intervalH$ max-heap of node x iff r includes $x.int_i$ for some i but does not include $x.int$. As in the case of the PIBT, each range is stored in the $intervalH$ max-heaps of at most 2 nodes at each level of the RIBT B-tree. Let $set(x)$ be the set of ranges to be stored in node x. Unlike the PIBT, where a prefix may be stored in several $interval$ vectors of a node, in an RIBT, each range $r \in set(x)$ is stored in eactly one $intervalH$ max-heap of x. To each range $r \in set(x)$, we assign an index (i, j) such that $x.key_{i-1} < start(r) \leq x.key_i$ and $x.key_j \leq finish(r) < x.key_{j+1}$, where $x.key_{-1} = -\infty$, $x.key_{t+2} = \infty$, and t is the number of keys in node x. Ranges of $set(x)$ that have the same index are stored in the same $intervalH$ max-heap. Thus, with each $intervalH$ max-heap, we associate an index (i, j), which is the index of the ranges in that max heap.

Lemma 3 *The number of intervalH max-heaps in an RIBT node x that has x.t keys is at most x.t.*

Proof The index (i, j) of an $intervalH$ max-heap corresponds to the range $[i, j]$. Let S be the set of ranges that correspond to the *intervalH* max-heaps of x. From the RIBT range allocation rule, it follows that $set(x)$ has no range r such that $start(r) \leq x.key_1$ and $finish(r) \geq x.key_t$. Therefore, $[x.key_0, x.key_{t+1}] \notin S$. Further, S has no range r such that $start(r) = finish(r)$. From the proof of case 1 of Lemma 2(a), it follows that $|S| \leq t$. ■

The structure of an RIBT node that has t keys and $q \leq t$ *intervalH* max-heaps is:

t

$key_1, key_2, ..., key_t$

$(child_0, hpr_0), (child_1, hpr_1), ..., (child_t, hpr_t)$

$equalHptr_1, equalHptr_2, ..., equalHptr_t$

$(i_1, j_1, intervalHptr_1), ..., (i_q, j_q, intervalHptr_q)$

where hpr_s is the highest-priority range in $set(x)$ that matches $x.int_s$, $equalHptr_s$ is a pointer to $equalH_s$, and $intervalHptr_s$ is a pointer to the *intervalH* max-heap whose index is (i_s, j_s).

Since the total number of endpoints is at most $2n$ and since each B-tree node other than the root has at least $\lceil m/2 \rceil - 1$ keys (keys are range endpoints), the number of B-tree nodes in an RIBT is $O(n/m)$. Each B-tree node takes $O(m)$ memory exclusive of the memory required for the max heaps. So, exclusive of the max-heap memory, we need $O(n)$ memory. Each range may be stored on $O(1)$ max heaps at each level of the B-tree. So, the max-heap memory is $O(n \log_m n)$. Therefore, the total memory requirement of the RIBT is $O(n \log_m n)$.

3.3 RIBT Operations

Figure 13 gives the algorithm to find the priority of the highest-priority range that matches the destination address d. This algorithm is easily modified to find the highest-priority range that matches d. The algorithm differs from algorithm *lengthOflmp* (Figure 4) primarily in the absence of the **break** statement in the **while** loop. Since Lemma 1 does not extend to the case of highest-priority matching in non-intersecting ranges, it isn't possible to stop the search for $hp(d)$ following the examination of the *equalH* max-heap for d.

The complexity of $hp(d)$ is $O(\log_2 m \log_m n) = O(\log_2 n)$ and the number of nodes accessed is $O(\log_m n)$. The algorithms to insert and delete a range are similar to the corresponding PIBT algorithms. So, we do not describe these here. When the maximum depth of nesting of the ranges is D. Although $D \leq n$ for ranges and $D \leq W$ for prefixes, in practice, we expect D to be quite small (in practical IPv4 prefix databases, for example, $D \leq 6$ [14]). Each *intervalH* max-heap of the RIBT has at

```
Algorithm $hp(d)$ {
    // return the priority of the highest-priority range that matches $d$
    $hp = -1$; // assume that all priorities are $\geq 0$
    $x$ = root of RIBT;
    while ($x \neq$ null) {
        Let $t$ = number of keys in $x$;
        Let $i$ be the smallest integer such that $x.key_{i-1} \leq d \leq x.key_i$.
        if ($i \leq t \&\& d == x.key_i$)
            $hp = \max\{hp,$ highest priority in $x.equalH_i\}$;
        $hp = \max\{hp, hpr_{i-1}\}$;
        $x = x.child_{i-1}$;
    }
    return $hp$;
}
```

Figure 13: Algorithm to find priority of $hpr(d)$

most D ranges. So, an insert and delete can be done in $O((\log_2 m + D) \log_m n) = O(\log_2 n + D \log_m n)$ time. The number of nodes accessed is $O(\log_m n)$.

4 Experimental Results

We implemented the B-tree router-table data structures PIBT (Section 2) and MRT [20] in C++ and compared their performance on a 700MHz PC. Initial experimentation with the implementations of the two B-tree structures showed that search time is optimal when the B-tree order is 32 (i.e., $m = 32$). Consequently, all experimental results reported in this section are for the case $m = 32$. To determine what benefits accrue from the use of a B-tree relative to a binary search tree, we included also the PBOB data structure of [9] in our performance measurements. Our experiments were conducted using six IPv4 prefix databases obtained from [6]. The datbases Paix1, Pb1, MaeWest and Aads were obtained on Nov 22, 2001, while Pb2 and Paix2 were obtained Sep 13, 2000. The number of prefixes in each of our 6 databases as well as the memory requirement for each database of prefixes are shown in Table 1. Although our PIBT structure uses about 12% less memory than is used by the MRT structure of [20], the PBOB structure of [9] uses about one-half the memory used by PIBT.

To measure the average lookup time, for each prefix database, we generated 1000 random addresses, $randAddr[0..999]$, that are matched by one or more of the prefixes in the database. Then, a sequence of 1 million lookups are done by generating 1 million unformly distributed random numbers in the range [0..999]. When a random number i is generated, we find $lmp(randAddr[i])$. From the time required for

Database	Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Num of Prefixes	16172	22225	28889	31827	35303	85988
PIBT (KB)	715	993	1292	1425	1604	3936
MRT (KB)	813	1132	1471	1621	1834	4526
PBOB (KB)	369	509	661	728	811	1961

Table 1: Memory Usage. $m = 32$ for PIBT and MRT

this sequence of 1 million random number generations and lookups, we subtract the time for the random number generation and divide by 1 million to get the average time per search. For each database and router-table data structure, the experiment is done 10 times and the average of the averages as well as the standard deviation of the averages computed. Since, each destination in *randAddr* is searched for approximately 1000 times, the experiment simulates a bursty traffic environment. Since the 1000 addresses in *randAddr* take 4000 bytes, the pollution of L2 cache (256KB) by *randAddr* is less than what it would be if we generated a random sequence of 1 million addresses and saved these in an array.

Table 2 gives the measured average lookup times. These average times are histogrammed in Figure 14. As can be seen, PIBT and MRT have almost the same performance on lookup. This is to be expected as a lookup in either structure results in the same number of cache misses and also does the same amount of work. The lookup time for PIBT and MRT is about 65% that of PBOB. Recall, however, that a lookup in PIBT and MRT can give us $lmp(d)$ but not the next hop associated with $lmp(d)$. Although the next-hop information associated with $lmp(d)$ can be stored in the B-tree, storing this information increases the complexity of the update operations. So, once we have determined $lmp(d)$, we must search another structure that stores the prefixes and associated next-hop information. In the case of PBOB, the next-hop information may be stored within the data structure at no additional cost to the update operations. Hence, it is quite likely that when we account for the added time needed to determine the next hop for $lmp(d)$, the lookup time advantage for the PIBT and MRT structures will diminish significantly.

Database	Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
PIBT	0.33	0.34	0.35	0.36	0.33	0.42
MRT	0.33	0.35	0.35	0.35	0.33	0.41
PBOB	0.49	0.50	0.53	0.53	0.51	0.61

Table 2: Lookup time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT. Variance is < 0.02

For the average update (insert/delete) time, we start by selecting 1000 prefixes from the database.

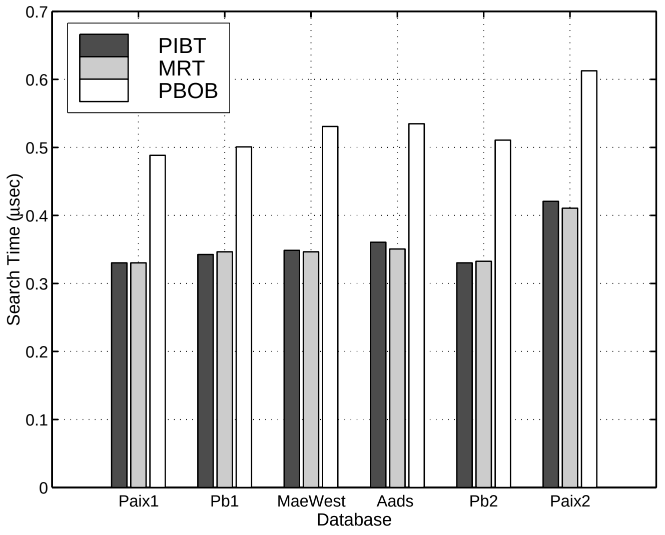

Figure 14: Lookup time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT

Those 1000 prefixes are first removed from the data structure. Once the 1000 removals are done, the removed 1000 prefixes are inserted back into the data structure. This cycle of remove 1000 prefixes and insert 1000 prefixes is repeated a sufficient number of times to make the total elapsed time one second (or more). The elapsed time is divided by 2000 times the number of cycle repetitions to get the average time for a single update. This experiment was repeated 10 times and the mean of the average update times computed. Table 3 gives the computed mean times and Figure 15 histograms these average times.

		Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
PIBT	mean	2.89	3.21	3.32	3.31	3.53	4.16
	std	0.061	0.021	0.040	0.033	0.047	0.016
MRT	mean	4.15	4.37	4.55	4.47	5.21	5.69
	std	0.183	0.034	0.029	0.043	0.239	0.030
PBOB	mean	0.42	0.43	0.44	0.45	0.46	0.47
	std	0.002	0.002	0.001	0.003	0.007	0.004

Table 3: Update time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT

As can be seen, an update in PBOB takes much less time than does an update in either MRT and PIBT. Further, an update in PIBT takes about 30% less time than does an update in MRT.

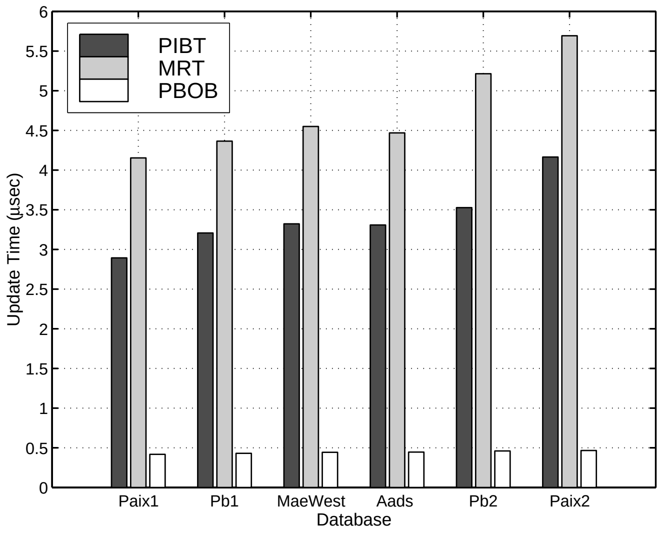

Figure 15: Update time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT

5 Conclusion

We have developed an alternative B-tree representation for dynamic router tables. Although our representation has the same asymptotic complexity as does the B-tree representation of [20], ours is faster for the update operation. This is because our structure performs updates with fewer cache misses. For the search operation, both B-tree structures take about the same time. When compared to the fastest binary tree structure, PBOB, for dynamic router tables, we see that the use of a high-degree tree enables the B-tree structure to perform better on the search operation. However, on the update operation, PBOB is decidedly superior.

References

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

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

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

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

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

[6] Merit, Ipma statistics, http://nic.merit.edu/ipma.

[7] B. Lampson, V. Srinivasan, and G. Varghese, IP lookup using multi-way and multicolumn search, *IEEE INFOCOM 98*, 1998.

[8] H. Lu and S. Sahni, $O(\log n)$ dynamic router-tables for prefixes and ranges. Submitted.

[9] H. Lu and S. Sahni, Dynamic IP router-tables using highest-priority matching. Submitted.

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

[11] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[12] 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*, 2001.

[13] 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, 220-227.

[14] S. Sahni and K. Kim, $O(\log n)$ dynamic packet routing, *IEEE Symposium on Computers and Communications*, 2002.

[15] S. Sahni and K. Kim, Efficient dynamic lookup for bursty access patterns, submitted.

[16] S. Sahni, Data structures, algorithms, and applications in Java, McGraw Hill, NY, 2000, 833 pages.

[17] S. Sahni, K. Kim, H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, 2002, 3-14.

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

[19] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *ACM Transactions on Computer Systems*, Feb:1-40, 1999.

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

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