BOB

Sartaj Sahni Haibin Lu

April 22, 2002

Abstract

In packet classification problem a best matching rule is found for incoming packet using one of the following rule, first matching rule, most specific rule, or highest priority rule. While packet classification problem on prefix router table with most specific rule was substantially studied, the one with highest priority rule hasn't. We propose a dynamic data structure BOB for packet classification with highest priority rule that matches the given destination address. BOB can be used for general nonintersecting range router table (including prefix router table). When constraining to prefix router table, further improvements for BOB are presented, BOBIP for highest priority rule and BOBIPV for most specific rule. As in experiment results, the proposed data structure achieve the good balance between lookup time, update time and memory requirement.

1 Introduction

For packet classification problem, a router table consisting of pairs of (rule, action) is used to classify incoming packets. For each incoming packet, a best rule matching the packet-header data is chosen from router table, and the corresponding action is carried out on the incoming packet, e.g., forwarding packet to next hop, rejecting packet. In one-dimensional packet classification problem, each rule is either a prefix or a range and the packet-head data used for the classification is the destination address of the packet. The best rule may be most specific rule (longest-prefix or shortest-range matching) or the highest priority. The router table can be static or **dynamic**. A dynamic router table is one into/from which rules are inserted/deleted concurrent with packet classification. In this paper, we focus on the dynamic router tables with the highest priority rule. While the described data structure is for one-dimensional case, it's easy to be extended to multi-dimensional dynamic packet classification problem by building tries on top of it.

Substantial research has been done on one-dimensional classifiers with longest-prefix matching [?]. Sahni et al [?] review such data structures that have been proposed. Most of them using expensive precomputation to fasten the lookup, which inhibits concurrent insert or delete rule. The scheme at [?] performs a binary search on prefix length, resulting in $O(\log W)$ expected complexity for lookup but are not suited for dynamic router-tables. Sahni and Kim [?] developed ACRBT to support the three operations of dynamic prefix routing table in $O(\log n)$ time each. The algorithm Sahni and Lu [?] proposed permits $O(\log n)$ lookup, insert and delete using priority search tree. While having the comparative lookup speed as [?] , it consumes less memory and has much faster insert and delete speed. Additional data structures is also developed to support shortest range routing rule for nonintersecting ranges and conflict-free ranges.

For highest priority matching rule, Gupta and Mckeown [?] have developed two data structures, heap on trie (HOT) and binary search tree on trie (BOT). Both of these are suitable for router table with arbitrary ranges. The HOT takes $O(W)$ time for a lookup and $O(W \log n)$ time for an insert or delete. The BOT takes $O(W \log n)$ time for a lookup and $O(W)$ time an insert or delete. When applied to prefix router table, HOT degrades to trie which permits each of three operations - lookup, insert and delete at $O(W)$ time, and BOT costs $O(W \log W)$ for a lookup and $O(W)$ for an insert or delete. Notice that the worst case complexities of HOT and BOT are also worst case cache misses.

In this paper, we propose binary search tree on binary search tree (BOB) to support lookup in $O(\log^2 n)$ time and insert/delete in $O(\log n)$ time. BOB can be used for prefix router table or nonintersecting ranges (prefixes are nonintersecting ranges) with highest priority matching rule. While applied to prefix router table, it costs $O(\log n \log W)$ for a lookup and $O(\log n)$ for an update. Further improvement, BOBIP, is proposed. In BOBIP, each of three operation can be done in $O(W)$ time but with actually $O(\log n)$ cache misses. When limiting to longest-prefix matching, BOBIPV is developed to support $O(W)$ for lookup with $O(\log n)$ cache misses, and $O(\log n)$ complexity for update.

2 Preliminaries

Definition 1 *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 $u \leq d \leq v$.*

Definition 2 *Let $r = [u, v]$ and $s = [x, y]$ be two ranges. Let $overlap(r, s) = r \cap s$.*
(a). The predicate $disjoint(r, s)$ is true iff r and s are disjoint.

$$disjoint(r, s) \Leftrightarrow overlap(r, s) = \emptyset$$

(b). The predicate $nested(r, s)$ is true iff one of the ranges is contained within the other.

$$nested(r, s) \Leftrightarrow overlap(r, s) = r \vee overlap(r, s) = s$$

Definition 3 *The range set R is nonintersecting iff $disjoint(r, s) \vee nested(r, s)$ for every pair of ranges r and $s \in R$.*

Definition 4 *The range r is more specific than the range s iff $r \subset s$.*

Definition 5 *Let R be range set. $ranges(d, R)$ (or simply $ranges(d)$ when R is implicit) is the subset of R that match the destination address d. $msr(d, R)$ (or $msr(d)$) is the most specific range of R that matches d. That is, $msr(d)$ is the most specific range is $ranges(d)$. In most-specific-range routing, the next hop for packets destined for d is given by the next-hop information associated with $msr(d)$.*

Definition 6 *Let R be range set. Each range $r \in R$ is assigned a priority p. $hpr(d)$ is the range that has the highest priority among $ranges(d, R)$. To avoid the possibility of tie, ranges are assigned different priorities. In highest-priority-range routing, the next hop for packets destined for d is given by the next-hop information associated with $hpr(d)$.*

Definition 7 *Let r and s be two ranges.* $r < s \Leftrightarrow start(r) < start(s) \vee (start(r) = start(s) \wedge finish(r) > finish(s))$

Lemma 1 *Let R be nonintersecting range set. If $r \cap s \neq \emptyset$ for r , $s \in R$, then the following is true:*
(1). $start(r) < start(s) \Rightarrow finish(r) \geq finish(s)$.
(2). $finish(r) > finish(s) \Rightarrow start(r) \leq start(s)$.

Proof Straightforward. ■

3 Data Structure

3.1 Point Search Tree (PTST)

We use balanced binary search tree as outermost data structure, and $pt(z)$, a distinguished point associated with each node z, as the search key. The range set R is partitioned into subsets according to the following range allocation rule.

Range Allocation Rule For the ranges containing $pt(root)$, we allocate them to the root. The right subtree of the root stores the subset of the ranges lying completely to the right of $pt(root)$, and the left subtree stores those lying completely to the left of $pt(root)$. Then, the left and right subtrees of the root are constructed recursively in the same way.

Let $G(z)$ denotes the subset allocated to node z, and $G(subtree(z))$ the union of the subsets allocated to the nodes at the subtree rooting at z (including z). We call node z **empty node** if $G(z)$ is empty, otherwise nonempty node.

Updating the balanced binary tree may cause tree out of balance. Rebalancing is usually achieved through rotations (figures 1 or 2). The subsets associated the nodes participating rotation may be changed.

Lemma 2 *In PTST, left/right rotation doesn't change $G(z)$, for the node $z \in a \cup b \cup c$ (a, b, c are subtrees at the figure 1 or 2).*

Proof For right rotation, $G(subtree(x))$ before rotation is equal to $G(subtree(y))$ after rotation since no change is made to the structure above. For subtree a,

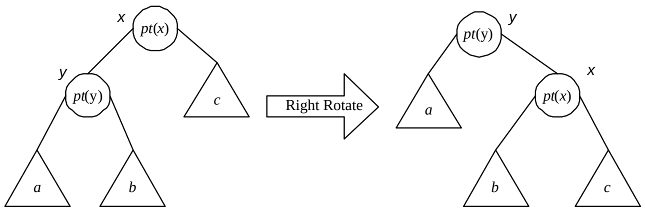

Figure 1: Right Rotation

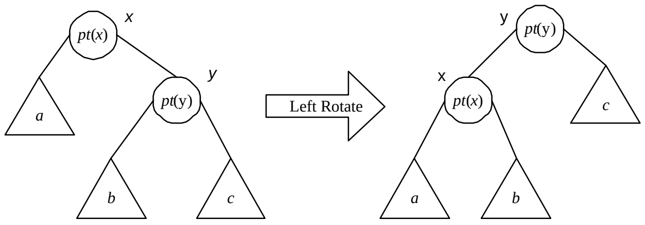

Figure 2: Left Rotation

range $r \in G(a)$ if and only if r lies completely to the left of $pt(y)$. Therefore, no change is made for $G(a)$ through the rotation. Since the structure of the subtree a remains the same, $G(z)$ is intact for all $z \in a$. Similar for $z \in b$ and $z \in c$. Left rotation is symmetric. ■

Let the subset allocated to node y / x after rotation $G'(y)$ / $G'(x)$. Notice that some of the ranges in $G(x)$ may have to be moved to $G'(y)$ due to allocation rule. Thus $G'(x)$ may be empty. If empty node is not allowed inside PTST, node x has to be deleted some time during the rebalancing procedure. But deletion a node may trigger bottom-up balance again, and rotation may create new empty node, and so on. Although the process will eventually terminate, we can not guarantee $O(\log n)$ complexity for update. Instead, we allow empty node inside PTST, but control the total number of empty nodes. The invariant below is introduced to achieve $O(\log n)$ time update while the asymptotic lookup time is maintained.

Invariant of PTST The total number of nodes is less than or equal to $2\,|R|$.

Lemma 3 *Left or right rotation won't violate the invariant of PTST.*

Proof True since left or right rotation doesn't create or eliminate nodes, and doesn't change $|R|$. ■

Lemma 4 *If PTST has $2n$ nodes, where $n = |R|$, then at least one degree-0 or degree-1 node is empty.*

Proof Suppose not. Then all empty nodes are degree-2. Let n_2 be the total number of degree-2 nodes, n_1 be total number of degree-1 nodes, n_0 the total number of degree-0 nodes, n_e the total number of empty nodes, and n_n the total number of nonempty nodes in PTST. Since all empty nodes are degree-2, $n_2 \geq n_e$. There are at most n nonempty nodes, i.e., $n \geq n_n$. Thus $n_2 + n \geq n_e + n_n = 2n$, i.e., $n_2 \geq n$. We know that $n_0 = n_2 + 1$. Hence $n_0 + n_1 + n_2 = n_2 + 1 + n_1 + n_2 > n_2 + n_2 \geq 2n$. Contradict since $n_0 + n_1 + n_2 = 2n$. ■

The empty nodes in PTST are kept in two linked lists, one for degree-0 or degree-1 nodes, the other for degree-2 nodes. Inserting an empty node into lists costs $O(1)$ time. Given the pointer the empty node, removing the node also takes $O(1)$ time.

3.2 Range Search Tree (RST)

For every node z in PTST, we store $G(z)$ in a balanced binary search tree called $RST(z)$. The search key is the range. Since the range is unique, the less-than relation defined at definition 7 is used. The balanced binary tree used has to support the operations split or join in $O(\log n)$ time as needed for updating BOB (section ?). There are three fields in a node x in $RST(z)$.

1. $r(x)$ which is the range stored at node x. Let $st(x) = start(r(x))$ and $fn(x) = finish(r(x))$.

2. $p(x)$ which is $priority(r(x))$.

3. $mp(x)$ which stores the maximum priority among the priorities of the ranges at the subtree rooted at x (including x). It can be computed as follows.
$$mp(x) = \begin{cases} p(x) & \text{if x is leaf.} \\ \max\{mp(lchild(x)), mp(rchild(x)), p(x)\} & \text{otherwise.} \end{cases}$$

Lemma 5 *The following is true in RST.*
(1).For every node y in the right subtree of x, $st(y) \geq st(x)$ and $fn(y) \leq fn(x)$.
(2).For every node y in the left subtree of x, $st(y) \leq st(x)$ and $fn(y) \geq fn(x)$.

Proof If y is in the right subtree of x, then $r(y) > r(x)$. From definition 7, $st(y) \geq st(x)$. Since $r(y) \cap r(x) \neq \emptyset$, if $st(y) > st(x)$, then $fn(y) \leq fn(x)$ due to lemma 1; if $st(y)$ equals to $st(x)$, $fn(y)$ must be less than $fn(x)$ in order to have $r(y) > r(x)$. Similar reason for (2). ■

4 Lookup

Let Q be query point. The lookup procedure returns the highest priority (hp) among the priorities of those ranges containing Q. We walk through the PTST using standard binary search. At the node z of PTST, if Q equals to the $pt(z)$, then all the ranges in $G(z)$ contain Q and no ranges in the subtrees of z contain Q, we simply update hp if current hp is less than maximum priority in $G(z)$, and terminate. If Q is less than $pt(z)$, we call the procedure $hpLeft$ on $RST(z)$ then descend to left child of z. Otherwise, Q is larger than $pt(z)$, call the $hpRight$ on $RST(z)$ then go to right child of z.

The methods $hpLeft$ and $hpRight$ are defined as the following. They update hp if current hp is less than the highest priority (hp) among the priorities of those ranges containing Q in $G(z)$.

$hpLeft$. The precondition of this procedure is $Q < pt(z)$. Starting from the root of $RST(z)$, walk down $RST(z)$. When current node is x, either $Q < st(x)$ or $Q \geq st(x)$. If $Q < st(x)$, then $st(y) \geq st(x) > Q$ for any node y in the right subtree of x. This mean no range in the right subtree of x contains Q. We simply descend to $lchild(x)$. In case of $Q \geq st(x)$, we have $st(y) \leq st(x) \leq Q < pt(z) \leq fn(x) \leq fn(y)$ for any node y in the left subtree of x, therefore $Q \in r(x)$ and $Q \in r(y)$. Update hp if current hp is less than $\max\{mp(lchild(x)), p(x)\}$ and descend to $rchild(x)$.

$hpRight$. The precondition is $Q > pt(z)$. Starting from the root of $RST(z)$, walk down $RST(z)$. Let x be current node, either $Q > fn(x)$ or $Q \leq fn(x)$. If $Q > fn(x)$, then for any node y in the right subtree of x, $fn(y) \leq fn(x) < Q$, so no range in the right subtree contains Q, descend to $lchild(x)$. If $Q \leq fn(x)$, we have $fn(y) \geq fn(x) \geq Q > pt(z) \geq st(x) \geq st(y)$ for any node y in the left subtree of x, therefore $Q \in r(x)$ and $Q \in r(y)$. Update hp if current hp is less than $\max\{mp(lchild(x)), p(x)\}$ and descend to $rchild(x)$.

Early termination of $hpLeft$ and $hpRight$. When trying to descend to $lchild(x)$, return if current hp is larger than or equal to $mp(lchild(x))$. When try to descend to $rchild(x)$, return if current hp is larger than or equal to $mp(rchild(x))$.

Complexity Since the cost of $hpLeft(hpRight)$ is $O(\log n)$ for nonintersecting ranges, $O(\log W)$ for prefixes (at most W prefixes can contain a given point), and the height of PTST is $O(\log n)$, the complexity of search is $O(\log^2 n)$ for nonintersecting ranges and $O(\log n \log W)$ for prefixes.

5 Insert

We insert range r with priority p into BOB as follows.

1. Find node z in PTST such that $pt(z) \in r$.

2. If no node z with $pt(z) \in r$, create a new node y in PTST with $pt(y) = start(r)$ (Any value between $st(r)$ and $fn(r)$ will serve). Create a new RST, associate it with node y, and insert (r, p) into $RST(y)$. Rebalance PTST if necessary. Done!

3. If there is node z with $pt(z) \in r$, then insert (r, p) into $RST(z)$. Done!

Complexity Step 1 costs $O(\log 2n)$ since at most $2n$ nodes at PTST. Step 2 can be done at $O(\log n)$ time as rebalancing PTST takes $O(\log n)$. Inserting into RST (step 3) takes $O(\log n)$ for ranges and $O(\min\{\log W, \log n\})$ for prefixes

```
hp = -1;//assuming 0 is the smallest priority value
z  = root(PTST);
while(z != null){
    if(Q > pt(z)){
        RST(z)->hpRight(Q, hp);
        z = rchild(z);
    }
    else if (Q < pt(z)){
        RST(z)->hpLeft(Q, hp);
        z = lchild(z);
    }
    else //Q == pt(z){
        hp = max{hp, mp(RST(z)->root)};
        return;
    }
}
```

Figure 3: Lookup Algorithm

because one RST has at most W prefixes. Totally, $O(\log n)$ for both ranges and prefixes.

5.1 Rebalance PTST

Rebalancing uses single left/right rotation (figures 1 and 2). We should make sure that

1. Range allocation rule is still maintained after rotation.
2. The linked lists for empty nodes are adjusted if some node becomes empty, nonempty, or some empty node change its degree. We handle this by removing x or y from empty list if it's empty before rotation, and insert x or y into empty list if it's empty after rotation. This can be at $O(1)$ time since at most two removing from linked lists and two insertion into linked lists and each can be done in $O(1)$ time.

Preserve range allocation rule

Single rotation won't affect subtrees a, b and c in figures 1 and 2 as stated in lemma 2. It does affect $G(x)$ and $G(y)$. The only case is some of ranges in $G(x)$ contain $pt(y)$, these ranges have to be removed from $G(x)$ and inserted into $G(y)$. Call the subsets allocated to x and y after rotation $G'(x)$ and $G'(y)$.

Observe that, for both left and right rotation, the ranges in $G(x)$ containing $pt(y)$ are less than those in $G(x)$ not containing $pt(y)$, and also less than all the ranges in $G(y)$, i.e., let $r \in G(x)$, $pt(y) \in r$, $s \in G(x)$, $pt(y) \notin s$, $t \in G(y)$, then $r < t < s$ for right rotation, and $r < s < t$ for left rotation as shown at figure 4 (Ranges with solid belong to $G(x)$, those with dotted line are in $G(y)$). Thus, we can use split and join to adjust $G(x)$ and $G(y)$ as following for both left and right rotation. Recall than $msr(pt(y), G(x))$ is the most specific range in $G(x)$ containing $pt(y)$. Hence $r \leq msr(pt(y), G(x)) < t < s$ for right rotation, and $r \leq msr(pt(y), G(x)) < s < t$ for left rotation. So $msr(pt(y), G(x))$ can be used as split element to conduct three-way split. Split $G(x)$ into G_{small}, $msr(pt(y), G(x))$ and G_{big}, where G_{small} includes all the ranges less than $msr(pt(y), G(x))$, and G_{big} has all the ranges larger than $msr(pt(y), G(x))$. Consistent with the range allocation rule, G_{big} is $G'(x)$, and $G'(y)$ is the join of G_{small}, $msr(pt(y), G(x))$ and $G(y)$ (figure 5).

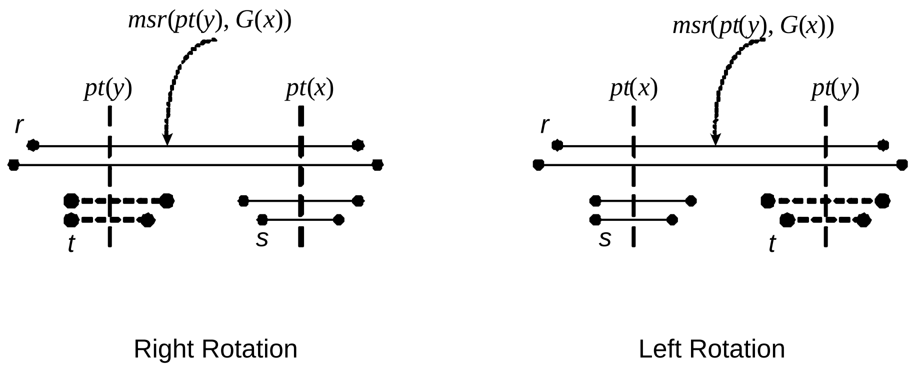

Figure 4: $G(x)$ and $G(y)$ for single rotation. Nodes x and y are listed at figures 1 and 2

$$\begin{tikzcd}
 & G_{small} \arrow[r] & G^1(y) \\
G(x) \arrow[ur] \arrow[r] \arrow[ddr] & msr(pt(y), G(x)) \arrow[ur] & \\
 & & G(y) \arrow[uu] \\
 & G_{big} = G^1(x) &
\end{tikzcd}$$

Figure 5: Split and Join for $G(x)$ and $G(y)$

During the split/join, proper updates of augmented field of RST along the split or join path, i.e., mp, have to be carried out.

Find $msr(pt(y), G(x))$. We know that the intersection of all ranges in $G(x)$ is not empty and $G(x)$ is nonintersecting. These properties are useful to support finding $msr(d, G(x))$ for a given point d in $O(h)$ time where h is the height of RST. We start from the root of $RST(x)$. Let w be current node in $RST(x)$. Either $d < st(w)$, or $st(w) \leq d \leq fn(w)$, or $fn(w) < d$. If $d < st(w)$, we descend to $lchild(w)$ because all ranges at the right subtree of w have starting point larger than or equal to $st(w)$, hence don't contain d. If $st(w) \leq d \leq fn(w)$, we update msr to $r(w)$ and go to $rchild(w)$ since all ranges at the left subtree of w have starting point less than or equal to $st(w)$, they are less specific than $[st(w), fn(w)]$. The last case is $fn(w) < d$. We descend to $lchild(w)$ because all ranges at the right subtree of w have finishing point less than or equal to $fn(w)$, they don't contain d. Apparently, finding $msr(d, G(x))$ takes $O(h)$.

In conclusion, single rotation for rebalancing PTST can be done in $O(\log n)$ time for ranges and $O(min\{\log W, \log n\})$ for prefixes. Because O(1) rotations are needed, total costs of rebalancing PTST is $O(\log n)$ time for both ranges and prefixes.

6 Delete

Delete range r from BOB as follows.

1. Find node z in PTST such that $pt(z) \in r$.
2. If no node z with $pt(z) \in r$, Fail.
3. Delete r from $RST(z)$ if any.
4. If $RST(z)$ becomes empty and z is degree-0 or degree-1 node, then delete z from PTST, rebalance PTST if necessary.
5. If $RST(z)$ becomes empty and z is degree-2 node, insert $(z, 2)$ into linked list for degree-2 empty nodes.
6. while current total number of nodes in PTST is larger than $2\,|R|$, get a degree-0 or degree-1 empty node from linked list for degree-0/1 empty nodes, and delete it and rebalance PTST if necessary.

Notice than the total number of nodes in PTST is less than or equal to $2\,|R|$ before deletion. If deletion is successful, i.e., range r is deleted, then $|R|$ becomes one less, the invariant may not hold. In worst case, the total number of nodes in PTST equals to $2\,|R|$ before deleting r, and becomes 2 more than $2\,|R|$ after rotation. Thus at most two empty nodes of PTST may have to be deleted in order to maintain the invariant. Since the deletion of empty node does not affect $|R|$, step 6 may be executed at most twice.

Similar to the complexity analysis of insertion, the deletion costs cost $O(\log n)$ for both ranges and prefixes.

7 BOB for IP

In current computer architecture, cache miss is quite expensive. For RST with height h, the worst case cache miss is h. Since at most W prefixes can contain the same point for the router table with prefixes, we may want to store $G(z)$ in an array rather than using binary search tree in order to reduce cache misses. While this may increase the computational cost, we may still benefit from less cache misses. Two modifications based on BOB are described in this section, one for prefix router table with the highest priority matching rule (BOBIP), the other for prefix router table with the longest matching prefix rule (BOBIPV).

7.1 BOBIP

The pairs of (prefix length, prefix priority) sorted by prefix length at increasing order are stored at an array. Since we know the prefixes in $G(z)$ contain $pt(z)$, the end points of prefix pf with length len can be computed at O(1) time as following.

$start(pf) = pt(z)$ bitwise-and $mask1(len)$

$finish(pf) = pt(z)$ bitwise-or $mask2(len)$

where $mask1(len)$ has the first len bits $'1'$ starting from the most significant bit and the rest $'0'$. $mask2(len)$ is the reverse of $mask1(len)$.

Lookup We walk through the PTST as in BOB lookup algorithm. Instead of doing binary search at $RST(z)$ at node z, we linear search the array starting at the first entry of the array, and stopping at the first entry which does not contain query point Q. During the linear search, hp is updated if the prefix contains Q and the priority of this prefix is higher. If Q equals to $pt(z)$, terminate since no prefixes at left or right subtree of node z contains Q. The computational complexity is $O(W)$ since at most W prefixes can contain Q. But the cache misses are limited to $O(\log 2n)$, which is the height of PTST. Recall that the computational cost and cache misses for BOB on prefix router tabel are both $O(\log W \log 2n)$.

Insert Instead of inserting (prefix, prefix priority) into $RST(z)$, we insert (prefix length, prefix priority) into array. In worst case, all the existing entries have to be shifted to accommodate the new pair. In order to save space, we only keep several empty entries in the array. When the entries are all occupied, the array has to be expanded. Expanding array is achieved by allocating a new node with more space. Consequently, the worst case cost of inserting a pair into array cost $O(W)$. Rebalancing PTST is performed as before except that the split and join of RSTs is replaced by moving pairs from $G(x)$ to $G(y)$. Moving pairs may cause node y to expand, node x to shrink if x has more empty array entries than allowed. All these can be done $O(W)$ time. Accordingly, the overall insertion complexity is $O(W)$. But the cache misses is $O(\log 2n)$.

Delete Deleting a pair from $G(z)$ is achieved by removing the corresponding entry from array. The node is shrunk if the node has more empty entries than allowed. Rebalaning PTST as needed. The overall complexity is $O(W)$, and the cache misses is $O(\log 2n)$.

7.2 BOBIPV

For longest-prefix matching rule, the prefix length equals to the prefix priority in the pair of (prefix length, prefix priority). Thus save prefix length only is enough. To avoid dynamically change the node size, a W-bits vector is used to represent $G(z)$. If a prefix with length len is presented at $G(z)$, the corresponding bit of the vector is set to $'1'$. Otherwise, $'0'$. Another potential advantage of using vector is reducing memory requirement.

Lookup We walk through the PTST as in BOB lookup algorithm. Instead of doing binary search at $RST(z)$ at node z, a linear search is performed on the vector starting at the kth bit. If the ith bit is $'1'$ and the corresponding prefix (length i) contains query point Q, update hp to i. The linear search stops at the first bit not containing query point Q, say mth entry. If Q equals to $pt(z)$, terminate. Otherwise we descend to the left child of z if $Q < pt(z)$, the right child of z if $Q > pt(z)$. Then assign m to k (since in $child(z)$ no prefix with length less than m contains Q) and start linear search again. Initially, $k = 0$ and $hp = 0$ (since the default prefix * always presents). The final value of hp is the length of the longest prefix matching Q. The computational complexity is $O(W)$ and the cache misses are limited to $O(\log 2n)$.

Insert Inserting a prefix into $G(z)$ is done by setting the corresponding bit to $'1'$ at $O(1)$ time using bitwise-and if the bitwise operation on vector is considered $O(1)$. When the left or right rotation during rebalancing PTST requires moving prefixes from $G(x)$ to $G(y)$, a binary search is performed on $vector(x)$ to determine a bit position n such that the prefixes from bit_0 to bit_n contain $pt(y)$ if the prefixes present in $G(x)$. Those prefixes can be removed from $G(x)$ by setting bit_0 to bit_n to $'0'$ using bitwise-and with a predefined mask, and be inserted into $G(y)$ by bitwise-or. So, the overall computational complexity is $O(\log W + \log 2n)$ and the cache misses $O(\log 2n)$.

Delete Deleting a prefix from $G(z)$ is done by setting the corresponding bit to $'0'$ at $O(1)$ time. Rebalancing PTST as needed in the same way of insertion. So, the overall computational complexity is $O(\log W + \log 2n)$ and the cache misses $O(\log 2n)$.

8 Implementation

Memory Management The customized memory management is used. A list of free objects is maintained, initially having zero object inside. Whenever a memory allocation is required, e.g., new operator, one object is returned if the free list is not empty. Otherwise, a big block of memory, say enough to hold 1,000 objects, is allocated and sliced into many objects, then linked together into free list. The object is put back into free list if it's released by delete operator. The customized memory manager is such simple that it does not take responsibility of ensure proper alignment for class data members, i.e. aligning data members at addresses that are natural for the data type and the processor involved. For example, a 4-byte data member should have an address that is a multiple of four. Since the memory manager calls the system memory manager to allocate the big block which is guaranteed to be suitably aligned for any type, the first object in that block is always aligned. Hence, if the maximum size of the data member is four bytes (this is the case for BOB, BOBIP and BOBIPV), we can ensure that all object in the free list are proper aligned by making the size of class exactly dividable by four (padding bytes may be used if the size of the class is not dividable by 4).

In the implementation of BOB, we augment the PTST node z in favor of search with $maxPri(z)$ (maximum priority value in $RST(z)$), $minSt(z)$ (smallest starting point of ranges in $RST(z)$), $maxFn(z)$ (largest finishing point of ranges in $RST(z)$). With these three fields, we don't call $hpLeft$ or $hpRight$ of $RST(z)$ if current hp is no less than $maxPri(z)$ or $Q \notin [minSt(z), maxFn(z)]$. With these three additional fields in PTST, we can also use $minSt(z)$ and $maxFn(z)$ as next and previous pointers for doubly linked list, thus keep the empty lists inside PTST.

For BOBIP, each node at PTST has four array entries initially. Since each prefix length takes a byte to store and one byte is used for priority value, four entries count to 8 bytes, thus next and previous pointers for doubly linked list can share the memory with the array. When the node expands, four more entries are allocated. The maximum number of empty array entries allowed is three except that the node can has four empty entries if the node is empty. For BOBIPV, the bit vector (32 bits for IPv4) and next pointer for doubly linked is put into union.

In the implementation of red black tree, parent pointer (using it favors update) is not used in order to reduce the size of nodes. The smaller the node size, the more nodes cache can hold, the faster the lookup. Node sizes and worst-case memory usages are listed at table 1. At worst case, PTST has $2n$ nodes. Thus the memory usage of BOB is $O(28 \times 2n + 20n)$. Since the nodes at BOBIP can have three empty array entries (6 bytes), it may waste at most $6n$ bytes (If the node is empty, the array is used by two pointers for doubly linked list). Thus the memory requirement for BOBIP is $O(24 \times 2n + 6n)$.

	BOB	BOBIP	BOBIPV	PST
Node Size	PTST(28) RST(20)	24(Minimum)	24	28
Memory Usage	$O(76n)$	$O(54n)$	$O(48n)$	$O(56n)$

Table 1: Node Sizes and Worst-case Memory Usages in Bytes(For IPv4 Router Table).

9 Experiment Results

We programmed BOB, BOBIP and BOBIPV based on red black trees. Because BOB is for nonintersecting ranges with the highest priority matching rule, BOBIP for prefixes with the highest priority matching rule, and BOBIPV for

		BOB	BOBIP	BOBIPV	PST
Lookup	Cost	$O(\log n \log W)$	$O(W)$	$O(W)$	$O(\log n)$
	Cache Misses	$O(\log n \log W)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Insert	Cost	$O(\log n)$	$O(W)$	$O(\log n + \log W)$	$O(\log n)$
	Cache Misses	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Delete	Cost	$O(\log n)$	$O(W)$	$O(\log n + \log W)$	$O(\log n)$
	Cache Misses	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 2: Operation Costs and Cache Misses (For IPv4 Router Table).

longest prefix matching, they are generally incomparable. Here we apply them to longest prefix matching. The results should give us some idea about their relative performances on this specific application. Recall that priority search tree [?] is one of the best performing $O(\log n)$ data structures reported for dynamic prefix router table. We also compare with it. Six IPv4 prefix databases obtained from [?] are used. The database 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 these databases as well as the memory requirments for each database are shown in table 3 and figure 6. The memory usages for BOB, BOBIP and BOBIPV are measured right after insert all the prefixes in each databases. The ratio of the number of empty nodes to the number of prefixes is below one percent. Since BOB, BOBIP and BOBIPV allow the ratio to reach fifty percent. The memory usages for them in table 3 are really best case memory usages.

Database		Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Num of Prefixes		16172	22225	28889	31827	35303	85988
Memory (KB)	PST	884	1215	1579	1740	1930	4702
	BOB	977	1350	1752	1932	2150	5200
	BOBIP	357	495	642	708	790	1901
	BOBIPV	357	495	642	708	790	1901

Table 3: Memory Usage

To obtain the lookup speed, we started with each data structure containing all prefixes of a prefix database. A random permutation of the set of start points of the ranges corresponding to the prefixes was obtained. This permutation determined the order in which we searched for the longest matching prefix for these start points. The procedure is repeated until the cumulate time was larger than one second. Then the elapsed time is divided by the product of the number of loops and the number of start points to get the average time per lookup. Using different permutation, the above is repeated for ten time and the mean and standard deviation of the ten mean times were computed. Notice we use the start points as $pt(z)$ for node z in PTST for BOB, BOBIP and BOBIPV, and searching determines if the query point equals to $pt(z)$. To be

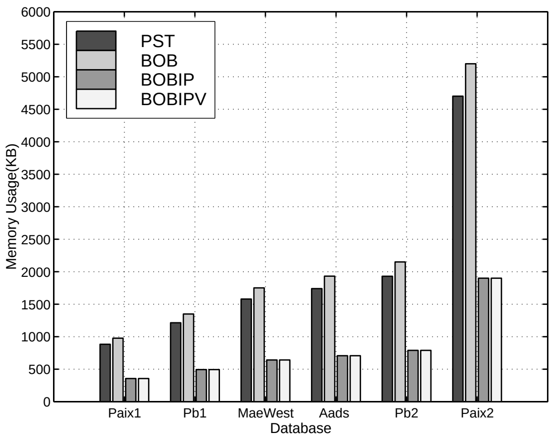

Figure 6: Memory Usage

fair for PST, we simple use start points plus one as query point. Table 4 gives the mean time required to find the longest prefix on a 1.4GHz Pentium 4 PC. As we can see, BOBIP has the best lookup performance. Though BOBIPV has the same asymptotic complexity as BOBIP for lookup, its performance is worst. By checking the number of prefixes in $G(z)$ for node z in PTST, we found that the majority of them have one or two prefix, only a few of them have more than three prefixes. The number of prefixes that BOBIP checks along the path may less than that of BOBIPV since BOBIPV checks the prefixes which may not even present. The average computational cost of BOBIP may be lower. BOB has worst performance on lookup since both its complexity and its cache misses are $O(\log n \log W)$, highest among these four data structures. Recall that the lookup was carried on right after insertion of all prefixes, when the number of empty nodes in PTST is blow 1% of the total prefixes. But having as many as the empty nodes inside BOB, BOBIP or BOBIPV won't degrade the lookup performance much since the height of PTST is logarithmic to the number of nodes.

To obtain the mean time of inserting a prefix, we started with a random permutation of the prefixes in a database, inserted the first 67% of the prefixes into an initially empty data structure, measured the time to insert the remaining 33%, and computed the mean insert time by dividing by the number of prefixes in 33% of the database. We noticed that the elapsed time to insert the rest 33% was actually very small, e.g., several ticks which can not be accurate. To get accurate measurement, we duplicated each data structure as in table 4. For example, we made 15 copies of each data structure for Paix1. With these duplication, we first inserted the first 67% of the prefixes into these 15 initially empty copies, measured the time to insert the remaining 33% into these 15 copies, and computed the mean insert time by dividing by the product of

Database			Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Search (μsec)	PST	Mean	1.20	1.35	1.48	1.53	1.58	1.96
		Std	0.01	0.01	0.01	0.01	0.01	0.01
	BOB	Mean	1.28	1.44	1.61	1.64	1.70	2.27
		Std	0.01	0.01	0.01	0.01	0.02	0.01
	BOBIP	Mean	0.81	0.97	1.11	1.17	1.20	1.60
		Std	0.01	0.01	0.01	0.01	0.01	0.01
	BOBIPV	Mean	1.20	1.37	1.51	1.57	1.62	2.05
		Std	0.01	0.01	0.01	0.01	0.01	0.01
Insert (μsec)	PST	Mean	2.21	2.38	2.58	2.61	2.66	3.01
		Std	0.04	0.06	0.06	0.04	0.05	0.05
	BOB	Mean	1.68	1.89	2.07	2.09	2.16	2.59
		Std	0.06	0.06	0.07	0.06	0.05	0.05
	BOBIP	Mean	1.02	1.24	1.40	1.45	1.51	1.96
		Std	0.06	0.04	0.04	0.06	0.05	0.05
	BOBIPV	Mean	1.09	1.30	1.42	1.51	1.56	1.93
		Std	0.05	0.06	0.05	0.05	0.05	0.06
Delete (μsec)	PST	Mean	1.75	1.89	2.04	2.11	2.15	2.50
		Std	0.04	0.06	0.06	0.04	0.07	0.06
	BOB	Mean	1.04	1.19	1.27	1.27	1.33	1.64
		Std	0.06	0.06	0.00	0.05	0.06	0.04
	BOBIP	Mean	0.70	0.81	0.89	0.94	0.98	1.28
		Std	0.06	0.06	0.06	0.00	0.04	0.01
	BOBIPV	Mean	0.66	0.79	0.87	0.91	0.94	1.25
		Std	0.06	0.06	0.06	0.06	0.05	0.06
Num of Copies			15	11	9	8	8	3

Table 4: Prefix times on a 1.4GHz Pentium 4 PC

15 and the number of prefixes in 33% of the database. The experiment was repeated 10 times and the mean of the mean as well as the standard deviation are calculated. As can be seen at table 4, though BOBIPV should have better performance than BOBIP since its node size is fixes, BOBIP and BOBIPV have the similar performance because only a few nodes in PTST have more than 3 prefixes as we just mentioned, so just a few operations of expansion or shrinking node size are required. PST has worst performance because the constant factor in its asymptotic complexity is higher.

To get the mean time to deleting a prefix, we still kept several copies, say 15 for PAIX1, for each data structure. Inserted all the prefixes in the database, then recorded the time to remove 33% of the prefixes from these 15 copies. The elapsed time was divided by 15 and the number of prefixes in 33% of the database. The experiment was repeated 10 times. Then the mean of the mean and the standard deviation are calculated. The relative performances are similar to that of insertion. We should expect better performance for both insertion and deletion if only single copy was used due to the utilization of cache. Having

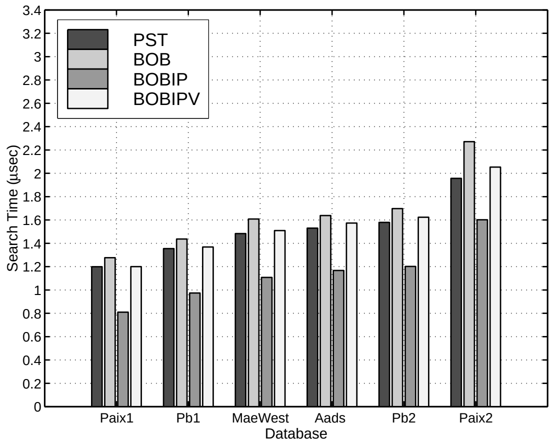

Figure 7: Search Time

more empty nodes inside PTST certainly helps insertion since we have more chance to find node z such that the prefix waiting for insert contains $pt(z)$, so no need to rebalance PTST. But the performance of deletion may be hurt by having more empty nodes because when the ration of the number of empty nodes to the number of prefixes reach 50%, every deletion triggers 2 removing of empty nodes.

10 Conclusion

We have developed the data structures for dynamic router table with highest priority matching rule. BOB permits lookup in $O(\log^2 n)$ for nonintersecting ranges and in $O(\log n \log W)$ for prefixes, insertion and deletion are done in $O(\log n)$ time each. A specialized version of BOB, BIBIP, is presented for prefix router table with highest priority matching rule. While the computational cost of BOB for each of three operations is $O(W)$, the cache misses is limited to $O(\log n)$. Further simplification of BOBIP results in BOBIPV with W-bits vector to support longest matching prefix routing. The lookup complexity of BOBIPV is $O(W)$ but the caches misses is $O(\log n)$. BOBIPV allows the insertion or deletion a prefix finish in $O(\log n)$ time.

11 Reference

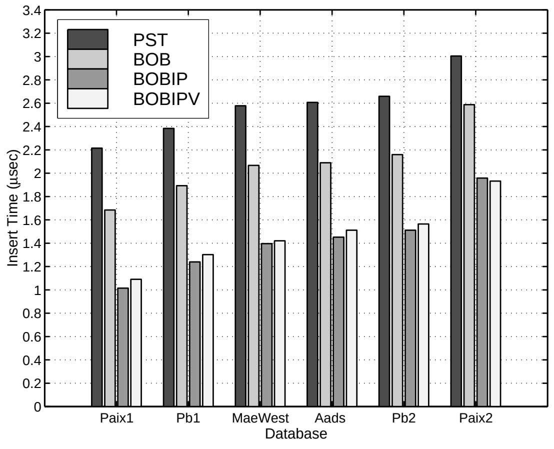

Figure 8: Insert Time

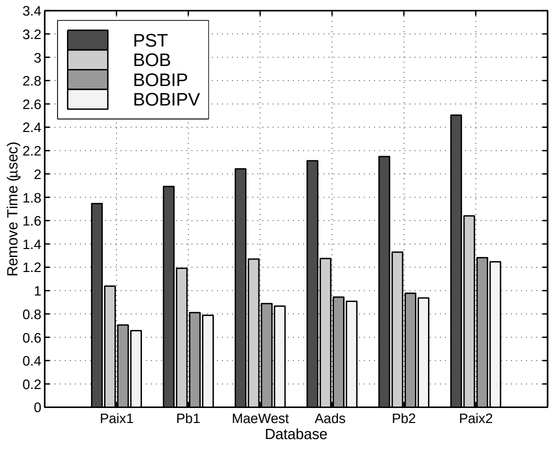

Figure 9: Delete Time