DATA STRUCTURES FOR DYNAMIC ROUTER TABLE

By

HAIBIN LU

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2003

To my family.

ACKNOWLEDGMENTS

I would like to give my sincere thankfulness to my advisor, Dr. Sartaj Sahni, for his mentoring and support throughout my Ph.D. study. It would be impossible to have my research career without his guidance.

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

I am very grateful to Dr. Sanjay Ranka, Dr. Randy Chow, Dr. Richard Newman, Dr. Michael Fang for serving on my Ph.D. supervisory committee and providing helpful suggestions.

I want to dedicate this dissertation to my parents. Without their encouragement and hard work, I could not think of getting a doctoral degree. Finally, I would like to give my special thanks to my wife, Lan, whose caring and love enabled me to complete this work.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DATA STRUCTURES FOR DYNAMIC ROUTER TABLE

By

Haibin Lu

August 2003

Chair: Sartaj Sahni
Major Department: Computer Information and Science and Engineering

Internet routers use router tables to classify incoming packets based on the information carried in the packet headers. Packet classification is one of the network bottlenecks, especially when a high update rate becomes necessary. Much of the research in the router-table area has focused on static prefix tables, where updates usually require the rebuilding of the whole router table. Some router-table designs rely on the relatively short IPv4 addresses to achieve desired efficiency. However, these designs have bad scalability in terms of the prefix length.

We propose several schemes to represent one-dimensional dynamic range tables, that is, tables into/from which rules are inserted/deleted concurrent with packet classification, and filters are specified as ranges. Our schemes allow real-time update and at the same time provide efficient lookup. The lookup and update complexities of our schemes are logarithmic functions of the number of the filters. The first scheme PST, which is based on priority search trees, uses the most specific rule tie breaker. The second scheme is called BOB (Binary search tree On Binary search tree). This scheme uses the highest priority tie breaker. In order to utilize the wide cache line size and reduce the tree height, a third scheme is developed in which the top level

tree is a B-Tree. This scheme also uses the highest priority tie breaker. All three schemes are suitable for prefix filters as well as for range filters in which no two filters have intersecting ranges. In addition, the PST also can handle a conflict-free range set.

CHAPTER 1
INTRODUCTION AND RELATED WORK

1.1 Introduction

Today's Internet consists of thousands of packet networks interconnected by routers. When a host sends a packet into the Internet, the routers relay the packet towards its final destination. The routers exchange routing information with each other, and use the information gathered to calculate the paths to all reachable destinations. Each packet is treated independently and forwarded to a next router based on its destination address.

The data structure a router uses to query next hop is called the *router table*. Each entry in the router table is a rule of the form (address prefix, next hop). Table 1–1 shows a set of five rules. We use W to denote the maximum possible length of a prefix. In IPv4, $W = 32$ and in IPv6, $W = 128$. In Table 1–1 W is 5. The prefix P1, which matches all the destination addresses, is called the *default prefix*. The prefix P3 matches the destination addresses between 16 and 19. If the address prefix of a rule matches the destination address the incoming packet carries, the next hop of this rule is used to forward packet.

Address prefix was introduced by CIDR (Classless Interdomain Routing) to deal with address depletion and router table explosion. The result of CIDR's address aggregation is that there may have several rules whose prefixes match the destination address. For example, the rules P1, P3 and P4 in Table 1–1 match the destination address 19. In this case, a tie breaker is needed to select one of the matching rules. The *most specific matching* is usually used, namely, the longest prefix matching the

destination address is the winner. For our example router table, P4 is the winner for destination address 19.

The other two popular tie breakers are *first matching* and *highest priority matching*. For first matching tie breaker, the rule table is assumed to be a linear list of rules with the rules indexed 1 through n for an n-rule table. The first rule that matches the incoming package is used. Notice that the rule R1 is selected for every incoming packet since it matches all the destination addresses. In order to give a chance to other rules to become the winner, we must index the rules carefully, and the default prefix should be the last rule.

In the highest priority matching, each rule is assigned a priority, and the rule with the highest priority is selected from those matching the incoming packet.[1] Notice that the first matching tie breaker is a special case of the highest priority matching tie breaker(simply assign each rule a priority equal to the negative of its index in the linear linear).

Table 1–1: A router table with five rules ($W = 5$)

Rule Name	Prefix Name	Prefix	Next Hop	Range Start	Range Finish
R1	P1	*	N1	0	31
R2	P2	0101*	N2	10	11
R3	P3	100*	N3	16	19
R4	P4	1001*	N4	18	19
R5	P5	10111	N5	23	23

The query based on the destination address is usually called *address lookup* or *packet forwarding*. In general other fields such as source address and port numbers may also be used, and the router table consists of the rules of the form (F, A), where F is a filter and A is an action. The action component of a rule specifies what is

[1] 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

to be done when a packet that satisfies the rule filter is received. Sample actions are drop the packet, forward the packet along a certain output link, and reserve a specified amount of bandwidth. Tie breakers similar to those mentioned earlier are used to select a rule from the set of rules that match the incoming packet. We call this problem *packet classification*.

1.1.1 Static Router Table

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?

To handle update, static schemes usually use two copies - working and shadow - of the router tables. Lookups are done using the working table. Updates are performed, in the background (either in real time on the shadow table or by batching updates and reconstructing an updated shadow at suitable intervals); periodically, the shadow replaces the working table, and the caches of the working table are flushed. In this mode of update operation, many packets may be misclassified, because the working copy isn't immediately updated. The number of misclassified packets depends on the periodicity with which the working table can be replaced by an updated shadow. Further, additional memory is required for the shadow table and for periodic reconstruction of the working table. It is important to have shorter preprocessing time in order to reduce the number of misclassified packets.

1.1.2 Dynamic Router Table

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 router-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.

We believe that dynamic structures for router tables is becoming a necessity. First, update occurs frequently in the backbone area. Labovitz et al. [1] found update rate could reach as high as 1000 per second. These updates stem from the route failure, route repair and route fail-over. With the number of autonomous systems continuously increasing, it is reasonable to expect the raising update rate. The router table needs to be updated in order to reflect the route change. Second, fast processing of update is preferred because during the batch and reconstruction, end-to-end delay increases, packet loss raises dramatically, and the part of network may experience connectivity loss. Labovitz et al. [2] observed dramatically increased packet loss and end-to-end latency during the BGP routing change. Batch and expensive reconstruction make things worse. While BGP takes time to converge, route-repair events usually do not cause multiple announcements, and the latency for router table to become stable due to these events should only depend on the network delay and router processing delays along the path [2]. In addition, when the BGP coverage time gets reduced, the processing delay may dominate. Pei et al. [3] reduce the convergence time from 30.3 seconds to 0.3 seconds for a failure withdraw in the testbed by applying two consistency assertions to BGP. Macian et al. [4] emphasize the importance of supporting high update rate. Dynamic router tables that permit high-speed inserts and deletes are essential in QoS and VAS applications [4]. For example, edge routers that do stateful filtering require high-speed updates [5].

For dynamic router 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.

For dynamic router table, the following metrics are measured to compare the performance:

1. *Lookup Time.*
2. *Insertion Time.* This is the time required to insert a new rule into the rule table.
3. *Deletion Time.* This is the time required to delete a rule from the rule table.
4. *Storage requirement.*

Note that there is only a working table for dynamic schemes and updates are made directly to the working table in real time. In this mode of update, no packet is improperly classified. However, packet classification/forwarding may be delayed until a preceding update completes. To minimize this delay, it is necessary that update be done as fast as possible.

Another important metric we concern for both static and dynamic router table is the scalability to IPv6. IPv6, the next generation of IP, uses 128-bit addresses ($W = 128$). Although some of the schemes in section 1.2 work well for IPv4 ($W = 32$), they have bad scalability in terms of the prefix length.

1.2 Related Work

Data structures for rule tables in which each filter is a address prefix and the rule priority is the length of this prefix[2] have been intensely researched in recent years. We refer to rule tables of this type as longest-matching prefix-tables (LMPT). 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). Although every LMPT is also an NHPRT, an NHPRT may not be an LMPT.

Ruiz-Sanchez et al. [6] review data structures for static LMPTs and Sahni et al. [7] review data structures for both static and dynamic LMPTs.

1.2.1 Trie

Several trie-based data structures for LMPTs have been proposed [8, 9, 10, 11, 12, 13, 14]. Structures such as that of Doeringer et al. [10] use the path-compression technique. Thus the memory requirement is $O(n)$. The search is guided by the input key and only inspects the bit position stored at the internal node due to a successful search bias. When the search reaches the leaf node and the search does not succeed, the downward path may be backtracked to find the longest matching prefix. Hence the search can be carried out in $O(W)$ time. The update operation, insert or delete, is natural in trie structure, and can also be performed in $O(W)$ time. The memory accesses during these operations are $O(W)$. For IPv6, $O(W = 128)$ memory accesses are quite expensive. Moreover, path compression reduces the height of trie only if the prefixes scatter inside the trie sparsely. When the number of prefixes increases, lots of branch nodes are needed and path compression does not have many nodes to

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

compress. Ruiz-Sanchez et al. [6] observe that the height of BSD version of path-compressed trie is 26 for a IPv4 router table with 47,113 prefixes, and the height of a simple binary trie is only 30.

In order to reduce the trie length, Gupta et al. [15] uses DIR-24-8 scheme which fully expands the binary trie at depth 24, i.e., all prefixes with length less than or equal to 24 are expanded to 24-bit prefixes as many as needed, and a table with 2^{24} entries is used to store these expanded prefixes. For those prefixes longer than 24 bits, a second table is used to store them. The correspondence is established by storing pointers in the first table which point to the proper entries in the second table. The first table has 2^{24} entries, and each entry is 16 bits (32M bytes in total). The first bit of each entry indicates whether the next 15 bits store the next hop or a pointer into 2nd table. With more than 32M bytes memory usage, the scheme can perform search in at most two memory accesses. But it is not scalable to IPv6 because expanding to 24 bits already takes too much memory. Gupta et al. [15] also propose alternatives that use less memory but require more memory accesses.

Degermark et al. [9] use a similar prefix expansion technique at multiple depths. Bitmap compression is deployed to reduced the memory requirement greatly. A router table with 40,000 rules can fit into 160K bytes. In the worst case, the number of memory accesses is nine. Huang et al. [16] fully expand the binary trie at depth 16 and also expand the sbutries rooted at the nodes in depth 16 to their own depths. The bitmap compression is also applied to reduce the memory requirement. The router tables used in the experiment can be compacted into less than 500K bytes. The number of worst case memory accesses is three. Both schemes [9, 16] heavily depend on the prefix distribution. It is hard to decide a proper memory size for the scheme ahead of time. For example, in extreme case, if n prefixes in the router table all have length 32, and their first 16-bits are distinct (assume $n <= 2^{16}$), the scheme [16] needs at least $2^{14}n$ bytes.

Nilsson et al. [11] apply the level compression as well as path compression to the binary trie. A binary trie is path-compressed first, then level compression is used to reduce the height of the trie further by substituting k highest levels of the binary trie with a single degree-2^k node. Although the search complexity of LC (level compressed) trie is still $O(W)$, the height of LC-trie is around 8 for the router tables used in author's analysis.

These data structures [9, 11, 15] as well as Srinivasan et al. [12] attempt to optimize lookup time through an expensive preprocessing step. They, while providing very fast lookup capability, have a prohibitive insert/delete time, so they are suitable only for static router-tables (i.e., tables into/from which no inserts and deletes take place).

Sahni et al. [13, 14] provide efficient constructions for fixed-stride and variable-stride multibit tries. The lookup time and memory requirement are optimized through expensive preprocessing.

Aiming at improving update speed for fixed-stride multibit trie at pipelined ASIC architecture, Basu et al. [17] describe an algorithm to optimize and balance the memory requirement across the pipeline stages.

1.2.2 Sets of Equal-Length Prefixes

Waldvogel et al. [18] have proposed a scheme that performs a binary search on hash tables organized by prefix length. In order to support binary search, $O(\log W)$ markers are generated for each prefix, and the longest matching prefix is precomputed for each marker. This binary search scheme has an expected complexity of $O(\log W)$ for lookup. The memory requirement is bounded by $O(n \log W)$. By introducing a technique called marker partitioning in the full version of Waldvogel et al. [18], the scheme has $O(\alpha \sqrt[\alpha]{n} W \log W)$ insert/delete time and an increased search time $O(\alpha + \log W)$, for $\alpha > 1$.

1.2.3 End-Point Array

An alternative adaptation of binary search to longest-prefix matching is developed in [19]. The distinct end points (start points and finish points) of the ranges defined by the prefixes are stored in ascending order in an array. The end points divide the universe into $O(n)$ basic intervals. The $LMP(d)$ is precomputed for each interval as well as for each end point. $LMP(d)$ is found by performing a binary search on this ordered array. A lookup in a table that has n prefixes takes $O(\log n)$ time. Because the schemes [19] use expensive precomputation, they are not suited for a dynamic router-tables.

1.2.4 Multiway Range Tree

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 $O(\log_m n)$ time. However, inserts/deletes take $O(W \log_m n)$ 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 Suri et al. [20] takes $O(m \log_2 W \log_m n)$ time for an insertion and $O(m \log_m n + W)$ for a deletion. Assume one node can fit into $O(1)$ cache line, the number of memory accesses that occur when the data structure of Suri et al. [20] is used is $O(\log_m n)$ per search, and $O(m \log_m n)$ per update.

1.2.5 $O(\log n)$ Dynamic Solutions

Sahni et al. [21, 22] 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 $O(\log n)$. Sahni et al. [22] show that their ACRBT structure is easily modified to extend the biased-skip-list structure of Ergun et al. [23] 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

Ergun et al. [23], the modified structure of Sahni et al. [22] 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 [22], which performs the three dynamic LMPT operations in $O(\log n)$ amortized time and which adapts to provide faster lookups for bursty traffic.

1.2.6 Highest-Priority Prefix Table

When an HPPT (highest-priority prefix-table) is represented as a binary trie [24], each of the three dynamic HPPT operations takes $O(W)$ time and cache misses.

Gupta et al. [25] 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.

1.2.7 TCAM

Ternary content-addressible memories, TCAMs, use parallelism to achieve $O(1)$ lookup [26]. Each memory cell of a TCAM may be set to one of three states 0, 1, and don't care. The prefixes of a router table are stored in a TCAM in descending order of prefix length. Assume that each work of the TCAM has 32 cells. The prefix 10* is stored in a TCAM work as 10??...?, where ? denotes a don't care and there are 30 ?s in the given sequence. To do a longest-prefix match, the destination address is matched, in parallel, against every TCAM entry and a sorted-by-length linear list, the longest matching-prefix can be determined in $O(1)$ time. A prefix may be inserted or deleted in $O(W)$ time, where W is the length of the longest prefix [27]. Although TCAMs provide a simple and efficient solution for static and dynamic router tables, this solution requires special hardware, costs more, and uses more power and board space than solutions that employ SDRAMs. TCAMs have longer latency than SDRAMs.

Since TCAM requires an arbitration module to choose the longest matching prefix and a more complex arbitration module are needed for a bigger router table, the latency of TCAM increases with router table size. EZchip Technologies, for example, claim that classifiers can forgo TCAMs in favor of comoodity memory solutions [5, 28]. Algorithmic approaches that have lower power consumption and are conservative on board space at the price of slightly increased search latency are sought. "System vendors are willing to accept some latency in their searches if it means lowering the power of a line card" [28].

1.2.8 Others

Cheung et al. [29] developed a model for table-driven route lookup and cast the table design problem as an optimization problem within this model. Their model accounts for the memory hierarchy of modern computers, and they optimize average performance rather than worst-case performance.

Solutions that involve modifications to the Internet Protocol (i.e., the addition of information to each packet) have also been proposed [30, 31, 32].

1.3 Contribution

We have developed data structures for dynamic router tables. The data structures use $O(n)$ space except that RIBT uses $O(n \log_m n)$ space. Our first data structure, PST [33, 34], uses the most specific matching tie breaker. It permits one to search, insert, and delete in $O(\log n)$ time each. Although $O(\log n)$ time data structures for prefix tables were known prior to our work [21, 22], the PST is more memory efficient than the data structures of [21, 22]. Further, PST is significantly superior on the insert and delete operations, while being competitive on the search operation. For nonintersecting ranges and conflict-free ranges PSTs are the first to permit $O(\log n)$ search, insert, and delete.

The second data structure, BOB [35], works for highest-priority matching with nonintersecting ranges. 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. For the case when all rule filters are prefixes, the data structure PBOB (prefix BOB) permits highest-priority matching as well as rule insertion and deletion in $O(W)$ time each. 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. PBOB can also support the dynamic-table operations in $O(\log n)$ time and with $O(\log n)$ cache misses for nonintersecting ranges when the number of nesting levels is a constant.

To utilize the wide cache line size, e.g., 64-byte cache line, 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 Suri et al. [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 Suri et al. [20] is used. Further, an update operation using the B-tree structure of Suri et al. [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.

With the $O(\log n)$ operation time, our data structures scale well to the large router tables. Since the complexity is independent of the prefix length, our data structures are also scalable to IPv6.

Another important feature of our data structures is that nonintersecting ranges are supported naturally, whereas most existing data structures support ranges (necessary when the filters are defined for port numbers) by breaking one range into $O(W)$ prefixes which results in $O(W \log n)$ memory requirement. Supporting ranges is also a nice feature for network layer addresses. The range that a prefix covers must be a power of two, and it must start at a number which is a multiple of the range size. But the end points and the size of a normal range can be any number. Supporting ranges means one can allocate a range with arbitrary size to a network (AppleTalk supports this feature) and the range aggregation is potentially better than that of prefix. For example, two disjoint prefixes can aggregate into one prefix only if their ranges are adjacent to each other and they have the same length, whereas the two disjoint ranges can aggregate into one range as long as they are next to each other. So, range aggregation is expected to result in router tables that have fewer rules.

CHAPTER 2
$O(\log n)$ DYNAMIC ROUTER TABLE FOR PREFIXES AND RANGES

In this chapter, we show in Section 2.2 how priority-search trees may be used to represent dynamic prefix-router-tables. The resulting structure, which is conceptually simpler than the CRBT structure of Sahni et al. [21], permits lookup, insert, and delete in $O(\log n)$ time each. For range router-tables, we consider the case when the best matching-prefix is the most-specific matching prefix (this is the range analog of longest-matching prefix). In Section 2.3, we show that dynamic range-router-tables that employ most-specific range matching and in which no two ranges overlap may be efficiently represented using two priority-search trees. Using this two-priority-search-tree representation, lookup, insert, and delete can be done in $O(\log n)$ time each. The general case of non-conflicting ranges is considered in Section 2.4. In this section, we augment the data structure of Section 2.3 with several red-black trees to obtain a range-router-table representation for non-conflicting ranges that permits lookup, insert, and delete in $O(\log n)$ time each. Section 2.1 introduces the terminology we use. In this section, we also develop the mathematical foundation that forms the basis of our data structures. Experimental results are reported in Section 2.5.

2.1 Preliminaries

2.1.1 Prefixes and Longest-Prefix Matching

The prefix 1101* matches all destination addresses that begin with 1101 and 10010* matches all destination addresses that begin with 10010. For example, when $W = 5$, 1101* matches the addresses $\{11010, 11011\} = \{26, 27\}$, and when $W = 6$, 1101* matches $\{110100, 110101, 110110, 110111\} = \{52, 53, 54, 55\}$. Suppose that a router table includes the prefixes $P1 = 101*$, $P2 = 10010*$, $P3 = 01*$, $P4 = 1*$, and

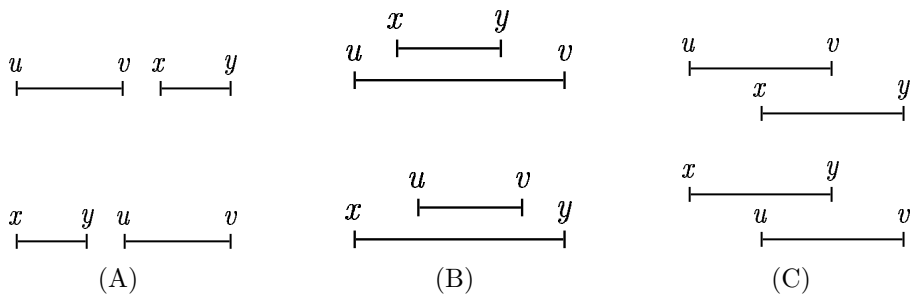


Figure 2–1: Relationships between pairs of ranges. A)Two ranges are disjoint. B)Two ranges are nested. C)Two ranges intersect.

$P5 = 1010*$. The destination address $d = 1010100$ is matched by the prefixes $P1$, $P4$, and $P5$. Since $|P1| = 3$ (the length of a prefix is number of bits in the prefix), $|P4| = 1$, and $|P5| = 4$, $P5$ is the longest prefix that matches d. In **longest-prefix routing**, the next hop for a packet destined for d is given by the longest prefix that matches d.

2.1.2 Ranges and Projections

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 **covers** *or* **matches** *all addresses* d *such that* $u \leq d \leq v$. $range(q)$ *is a predicate that is true iff* q *is a range.*

The start point of the range $r = [3, 9]$ is 3 and its finish point is 9. This range covers or matches the addresses {3, 4, 5, 6, 7, 8, 9}. In IPv4, s and f are up to 32 bits long, and in IPv6, s and f may be up to 128 bits long. The IPv4 prefix $P = 0*$ corresponds to the range $[0, 2^{31} - 1]$. The range [3,9] does not correspond to any single IPv4 prefix. We may draw the range $r = [u, v] = \{u, u+1, ..., v\}$ as a horizontal line that begins at u and ends at v. Figure 2–1 shows ranges drawn in this fashion.

Notice that every prefix of a prefix router-table may be represented as a range. For example, when $W = 6$, the prefix $P = 1101*$ matches addresses in the range $[52, 55]$. So, we say $P = 1101* = [52, 55]$, $start(P) = 52$, and $finish(P) = 55$.

Since a range represents a set of (contiguous) points, we may use standard set operations and relations such as $\cap$ and $\subset$ when dealing with ranges. So, for example, $[2,6] \cap [4,8] = [4,6]$. Note that some operations between ranges my not yield a range. For example, $[2,6] \cup [8,10] = \{2,3,4,5,6,8,9,10\}$ is not a range.

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.

$$disjoint(r,s) \iff overlap(r,s) = \emptyset \iff v < x \vee y < u$$

Figure 2–1(A) shows the two cases for disjoint sets.

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

$$\begin{aligned} nested(r,s) &\iff overlap(r,s) = r \vee overlap(r,s) = s \\ &\iff r \subseteq s \vee s \subseteq r \\ &\iff x \leq u \leq v \leq y \vee u \leq x \leq y \leq v \end{aligned}$$

Figure 2–1(B) shows the two cases for nested sets.

(c) The predicate $intersect(r,s)$ is true iff r and s have a nonempty intersection that is different from both r and s.

$$\begin{aligned} intersect(r,s) &\iff r \cap s \neq \emptyset \wedge r \cap s \neq r \wedge r \cap s \neq s \\ &\iff \neg disjoint(r,s) \wedge \neg nested(r,s) \\ &\iff u < x \leq v < y \vee x < u \leq y < v \end{aligned}$$

Figure 2–1(C) shows the two cases for ranges that intersect.

Notice that $overlap(r,s) = [x,v]$ when $u < x \leq v < y$ and $overlap(r,s) = [u,y]$ when $x < u \leq y < v$.

[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. $[4, 4]$ is the overlap of $[2, 4]$ and $[4, 6]$; and $overlap([3, 8], [2, 4]) = [3, 4]$.

Lemma 1 *Let r and s be two ranges. Exactly one of the following is true.*

1. $disjoint(r, s)$
2. $nested(r, s)$
3. $intersect(r, s)$

Proof Straightforward. ■

Definition 3 *Let $R = \{r_1, ..., r_n\}$ be a set of n ranges. The* **projection**, $\Pi(R)$, *of R is*

$$\Pi(R) = \cup_{1 \leq i \leq n} r_i$$

That is, $\Pi(R)$ comprises all addresses that are covered by at least one range of R.

For $A = \{[2, 5], [3, 6], [8, 9]\}$, $\Pi(A) = \{2, 3, 4, 5, 6, 8, 9\}$, and for $B = \{[4, 8], [7, 9]\}$, $\Pi(B) = \{4, 5, 6, 7, 8, 9\}$. $\Pi(A)$ is not a range. However, $\Pi(B)$ is the range $[4, 9]$. Note that $\Pi(R)$ is a range iff $d \in \Pi(R)$ for every d, $u \leq d \leq v$, where $u = \min\{d | d \in \Pi(R)\}$ and $v = \max\{d | d \in \Pi(R)\}$.

Lemma 2 *Let $R = \{r_1, r_2, ..., r_n\}$ be a set of n ranges such that $\Pi(R) = [u, v]$.*

(a) $u = minStart(R) = \min\{start(r_i)\}$ and $v = maxFinish(R) = \max\{finish(r_i)\}$.

(b) Let s be a range. $\Pi(R \cup \{s\})$ is a range iff $start(s) \leq v+1$ and $finish(s) \geq u-1$.

(c) When $\Pi(R \cup \{s\}) = [x, y]$, $x = \min\{u, start(s)\}$ and $y = \max\{v, finish(s)\}$.

Proof (a) is straightforward. Figure 2–2 shows all possible cases for which $\Pi(R \cup \{s\})$ is a range, s is shown as a solid line. (b) and (c) are readily verified for each case of Figure 2–2. ■

2.1.3 Most-Specific-Range Routing and Conflict-Free Ranges

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

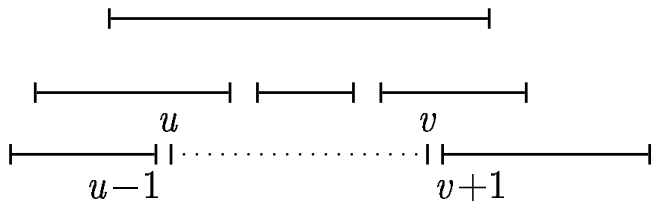


Figure 2–2: Cases for Lemma 2

$[2,4]$ is more specific than $[1,6]$, and $[5,9]$ is more spcific than $[5,12]$. Since $[2,4]$ and $[8,14]$ are disjoint, neither is more specific than the other. Also, since $[4,14]$ and $[6,20]$ intersect, neither is more specific than the other.

Definition 5 *Let R be a range set. $ranges(d,R)$ (or simply $ranges(d)$ when R is implicit) is the subset of ranges of R that match/cover 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 in $ranges(d)$. $msr([u,v],R) = msr(u,v,R) = r$ iff $msr(d,R) = r$, $u \leq d \leq v$. When R is implicit, we write $msr(u,v)$ and $msr([u,v])$ in place of $msr(u,v,R)$ and $msr([u,v],R)$. In* **most-specific-range routing***, the next hop for packets destined for d is given by the next-hop information associated with $msr(d)$.*

When $R = \{[2,4],[1,6]\}$, $ranges(3) = \{[2,4],[1,6]\}$, $msr(3) = [2,4]$, $msr(1) = [1,6]$, $msr(7) = \emptyset$, and $msr(5,6) = [1,6]$. When $R = \{[4,14],[6,20],[6,14],[8,12]\}$, $msr(4,5) = [4,14]$, $msr(6,7) = [6,14]$, $msr(8,12) = [8,12]$, $msr(13,14) = [6,14]$, and $msr(15,20) = [6,20]$.

Definition 6 *The range set R has a* **conflict** *iff there exists a destination address d for which $ranges(d) \neq \emptyset \wedge msr(d) = \emptyset$. R is* **conflict free** *iff it has no conflict. The predicate $conflictFree(R)$ is true iff R is a conflict-free range set.*

$conflictFRee(\{[2,8],[4,12],[4,8])\}$ is true while $conflictFree(\{[2,8],[4,12])\}$ is false.

We note that our definition of conflict free is a natural extension to ranges of the definition of conflict free given by Hari et al. [36] for the case of two-dimensional prefix rules.

Definition 7 *Let r and s be two intersecting ranges of the range set R. The subset $Q \subset R$ is a* **resolving subset** *for these two ranges iff Q is conflict free and $\Pi(Q) = overlap(r, s)$. Two ranges of a range set are in* **conflict** *iff they intersect and have no resolving subset. Two ranges are* **conflict free** *iff they are not in conflict.*

Lemma 3 *A range set is conflict free iff it has no pair of ranges that are in conflict.*

Proof Follows from the definition of a conflict-free range set. ■

Lemma 4 *Let R be a conflict-free range set. Let r be an arbitrary range. Let A be the subset of R that comprises all ranges of R that are contained in r. A is conflict free.*

Proof Since R is conflict free, every pair (s, t) of intersecting ranges in A has a resolving subset B in R. From Definition 7, it follows that every range in B is contained in $overlap(s, t)$. Hence, $B \subset A$. Therefore, every pair of intersecting ranges of A has a resolving subset in A. So, A is conflict free. ■

Lemma 5 *Let R be a conflict-free range set. Let A, $A \subseteq R$ be such that $\Pi(A) = r = [u, v]$.*

1. $\exists B \subseteq R[conflictFree(A \cup B) \wedge \Pi(A) = \Pi(A \cup B)]$
2. *Let $s \in R$ be such that $intersect(r, s)$. $\exists B \subseteq R[\Pi(B) = overlap(r, s)]$*
3. *$R \cup \{r\}$ is conflict free.*

Proof

1. Follows from Lemma 4.
2. When $r \in R$, (2) follows from the definition of a conflict-free range set. So, assume $r \notin R$. Let C comprise all ranges of A contained in s. If s intersects no range of A, $\Pi(C) = overlap(r, s)$. If s intersects at least one range of A, then

let $t \in A$ be an intersecting range with maximum overlap. Since R is conflict free, $\exists D \subseteq R[\Pi(D) = overlap(t, s)]$. We see that $\Pi(C \cup D) = overlap(r, s)$.

3. From parts (1) and (2) of this lemma, it follows that there is a resolving subset in $R \cup \{r\}$ for every $s \in R$ that intersects with r. Hence, $R \cup \{r\}$ is conflict free.

■

Definition 8

$maxP(u, v, R) = \max\{finish(\Pi(A)) | A \subseteq R \wedge range(\Pi(A)) \wedge start(\Pi(A)) = u \wedge finish(\Pi(A)) \leq v\}$ is the maximum possible projection that is a range that starts at u and finishes by v.

$minP(u, v, R) = \min\{start(\Pi(A)) | A \subseteq R \wedge range(\Pi(A)) \wedge finish(\Pi(A)) = v \wedge start(\Pi(A)) \geq u\}$ is the minimum possible projection that is a range that finishes at v and starts by u.

When $\nexists A \subseteq R[range(\Pi(A)) \wedge start(\Pi(A)) = u \wedge finish(\Pi(A)) \leq v]$, we say that $maxP(u, v, R)$ does not exist. Similarly, $minP(u, v, R)$ may not exist. At times, we use $maxP$ and $minP$ as abbreviations for $maxP(u, v, R)$ and $minP(u, v, R)$, respectively.

$maxY(u, v, R) = \max\{y | [x, y] \in R \wedge x < u \leq y < v\}$ and $minX(u, v, R) = \min\{x | [x, y] \in R \wedge u < x \leq v < y\}$. Note that $maxY$ and $minX$ may not exist.

Lemma 6 *Let R be a conflict-free range set. Let $A = R \cup \{r\}$, where $r = [u, v] \notin R$.*

$$conflictFree(A) \iff maxY(u, v, R) \leq maxP(u, v, R) \wedge minX(u, v, R) \geq minP(u, v, R)$$

where $maxY \leq maxP$ ($minX \geq minP$) is true whenever $maxY$ ($minX$) does not exist and is false when $maxY$ ($minX$) exists but $maxP$ ($minP$) does not.

Proof ($\Longrightarrow$) Assume that A is conflict free. When neither $maxY$ nor $minX$ exist (this happens iff no range of R intersects $r = [u, v]$), $maxY \leq maxP \wedge minX \geq minP$. When, $maxY$ exists, $s = [x, maxY] \in A \wedge x < u \leq maxY < v$. (Note that $intersect(r, s)$.) Since A is conflict free, A has a (resolving) subset B for which

$\Pi(B) = overlap(r, s) = [u, maxY]$. Therefore, $maxY \leq maxP$. Similarly, when $minX$ exists, $minX \geq minP$.

($\Longleftarrow$) Assume $maxY(u, v, R) \leq maxP(u, v, R) \wedge minX(u, v, R) \geq minP(u, v, R)$. When neither $maxY$ nor $minX$ exist, no range of R intersects r. When, $maxY$ exists, $\exists s = [x, y] \in A[x < u \leq y < v]$. Consider any such $s = [x, y]$. Since $maxY \leq maxP$ and $maxY$ exists, $maxP$ exists. Hence, $\exists B \subseteq R[conflictFree(B) \wedge \Pi(B) = [u, maxP]]$. When $y = maxP$, B is a resolving subset for s and r in A. When $y < maxP$, $intersect(s, [u, maxP])$. Since $R \cup \{[u, maxP]\}$ is conflict free (lemma 5(3)), $R \cup \{[u, maxP]\}$ (and so also R and A) has a resolving subset for s and $[u, maxP]$. This resolving subset is also a resolving subset for s and r. When $minX$ exists, $\exists s = [x, y] \in A[u < x \leq v < y]$. In a manner analogous to the proof for the case $maxY$ exists, we may show that A has a resolving subset for r and each such s. Hence, in all cases, intersecting ranges of A have a resolving subset. So, A is conflict free. ■

Lemma 7 *Let R be a conflict-free range set. Let $A = R - \{r\}$ for some $r \in R$.*

$$\nexists B \subseteq A[\Pi(B) = r] \wedge \nexists s \in A[r \subseteq s] \Longrightarrow \nexists C \subseteq A[r \subseteq \Pi(C)]$$

Proof Assume

$$\nexists B \subseteq A[\Pi(B) = r] \tag{2.1}$$

and

$$\nexists s \in A[r \subseteq s] \tag{2.2}$$

We need to show that $\nexists C \subseteq A[r \subseteq \Pi(C)]$.

Suppose that there is a C such that $C \subseteq A \wedge r \subseteq \Pi(C)$. From $C \subseteq A$ and Equation 2.2, it follows that

$$\forall t \in C[disjoint(r, t) \vee intersect(r, t) \vee t \subset r] \tag{2.3}$$

If $\nexists t \in C[intersect(r,t)]$, then from Equation 2.3, we get $\forall t \in C[disjoint(r,t) \vee t \subset r]$. From this and $r \subseteq \Pi(C)$, it follows that all destination addresses d, $d \in r$, are covered by ranges of C that are contained in r. Therefore, $\exists B \subseteq C \subseteq A(\Pi(B) = r)$. This contradicts Equation 2.1.

Next, suppose $\exists t \in C[intersect(r,t)]$. Let D be the union of the resolving subsets for all of these t and r in R. Clearly, all ranges in D are contained in r. Further, let E be the subset of all ranges in C that are contained in r. It is easy to see that $D \cup E \subseteq A \wedge \Pi(D \cup E) = r$. This contradicts Equation 2.1. ■

Lemma 8 *Let R be a conflict-free range set. Let $A = R - \{r\}$, for some $r \in R$.*

1. $\exists B \subseteq A[\Pi(B) = r] \Longrightarrow conflictFree(A)$.
2. $\nexists B \subseteq A[\Pi(B) = r] \Longrightarrow [conflictFree(A) \Longleftrightarrow \nexists s \in A[r \subset s] \vee [m,n] \in A]$, *where* $m = \max\{start(s) | s \in A \wedge r \subseteq s\}$, *and* $n = \min\{finish(s) | s \in A \wedge r \subseteq s\}$.

Proof For (1), we note that by replacing r by B in every resolving subset for intersecting ranges in R, we get resolving subsets that do not include r. Hence all of these resolving subsets are present in A. So, A is conflict free.

For (2), assume that $\nexists B \subseteq A[\Pi(B) = r]$.

($\Longrightarrow$) Assume that A is conflict free. We need to prove

$$\nexists s \in A[r \subset s] \vee [m,n] \in A \tag{2.4}$$

We do this by contradiction. So, assume

$$\exists s \in A[r \subset s] \wedge [m,n] \notin A \tag{2.5}$$

Since $\exists s \in A[r \subset s]$, m and n are well defined. Equation 2.5 implies that A has a range $[m,y]$, $y > n$ as well as a range $[x,n]$, $x < m$. Further, $intersect([m,y],[x,n])$ and $r \subseteq overlap([m,y],[x,n]) = [m,n]$. Let B be the subset of R comprised of all ranges contained in $[m,n]$. From Lemma 4, it follows that B is conflict free. However, r is the projection of no subset of $C = B - \{r\}$. Further, no range of C contains

r. From Lemma 7, it follows that no subset of C has a projection that contains r. In particular, C has no subset whose projection is $[m, n]$. Therefore, A, has no subset whose projection is $[m, n]$. So, A has no resolving subset for $[m, y]$ and $[x, n]$. Therefore, A is not conflict free, a contradiction.

($\Longleftarrow$) If no range of A contains r, then r is not part of the resolving subset for any pair of intersecting ranges of R. This, together with the fact that R is conflict free, implies that A is conflict free. If $[m, n] \in A$, we can use $[m, n]$ in place of r in any resolving subset for intersecting ranges of R. Therefore, A has a resolving subset for every pair of intersecting ranges. So, A is conflict free. ■

Lemma 9 *Let R be a conflict-free range set and let d be a destination address. If $ranges(d) \neq \emptyset$, then $start(msr(d)) = a = maxStart(ranges(d)) = \max\{start(r) | r \in ranges(d)\}$ and $finish(msr(d)) = b = minFinish(ranges(d)) = \min\{finish(r) | r \in ranges(d)\}$.*

Proof Since R is conflict free and $ranges(d) \neq \emptyset$, $msr(d) \neq \emptyset$. Assume that $msr(d) = s$. If $s \neq [a, b]$, then $start(s) < a$ or $finish(s) > b$. Assume that $start(s) < a$ (the case $finish(s) > b$ is similar). Let $t \in ranges(d)$ be such that $start(t) = a$. Now, $intersect(s, t) \vee t \subset s$. Hence, $s \neq msr(d)$. ■

2.1.4 Normalized Ranges

Definition 9 [Normalized Ranges] *The range set R is* **normalized** *iff one of the following is true.*

1. $|R| \leq 1$.
2. $|R| > 1$ *and for every $r \in R$ and every $s \in R$, $r \neq s$, one of the following is true.*
   (a) $disjoint(r, s)$.
   (b) $nested(r, s) \wedge start(r) \neq start(s) \wedge finish(r) \neq finish(s)$. *That is, r and s are nested and do not have a common end-point.*

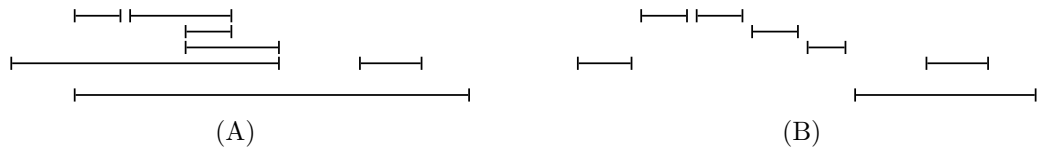


Figure 2–3: Unnormalized and normalized range sets

Figure 2–3(A) shows a range set that is not normalized (it contains ranges that intersect as well as nested ranges that have common end-points). Figure 2–3(B) shows a normalized range set. Regardless of which of these two range sets is used, every destination d has the same most-specific range.

Definition 10 *An ordered sequence of ranges* $(r_1, ..., r_n)$ *is a* **chain** *iff* $\forall i < n$ $[start(r_{i+1}) = finish(r_i)]$. *A range set* R *is a* **chain** *iff its ranges can be ordered so as to form a chain.* $chain(R)$ *is a predicate that is true iff* R *is a chain.*

The range sequence $([2,4],[5,7],[8,12])$ is a chain while $([5,8],[12,14])$ and $([5,8],[2,4])$ are not. The range sets $\{[5,8],[2,4]\}$ and $\{[2,4],[8,12],[5,7]\}$ are chains while $\{[2,4],[8,12]\}$ and $\{[2,4],[5,7],[8,12],[9,10]\}$ are not. Note that when R is a chain, $\Pi(R) = [minStart(R), maxFinish(R)]$.

Lemma 10 *Let* N *be a normalized range set.*

$$A \subseteq N \wedge \Pi(A) = [u,v] \Longrightarrow \exists B \subseteq N[chain(B) \wedge \Pi(B) = [u,v]]$$

Proof Let B be the subset of A obtained by removing from A all ranges that are nested within at least one other range of A. Clearly, $\Pi(B) = \Pi(A) = [u,v]$. Since N is normalized and $B \subseteq N$, B is also normalized. From Definition 9 and the fact that B has no pair of nested ranges, it follows that all ranges of B are disjoint. For disjoint ranges to have a projection that is a range, the disjoint ranges must form a chain. ■

Lemma 11 *Let* N *be a normalized range set.*

1. N *may be uniquely partitioned into a set of longest chains* $CP(N) = \{C_1, ..., C_k\}$, $N = \cup_{1 \le i \le k} C_i$. *By longest chains, we mean that no two chains*

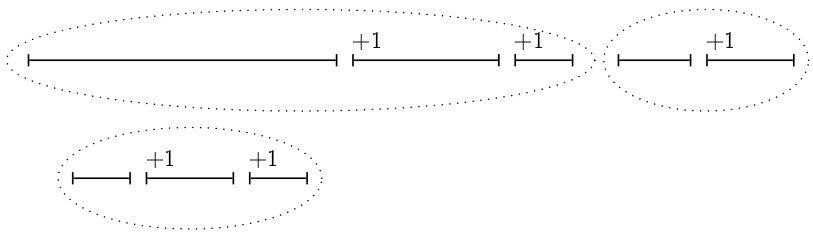


Figure 2–4: Partitioning a normalized range set into chains

of CP may be combined into a single chain. $CP(N)$ is called a **canonical partitioning**.

2. *For all i and j, $1 \leq i < j \leq k$, C_i and C_j are either disjoint,or C_i is properly contained within a range of C_j or C_j is properly contained within a range of C_i. A chain C_i is properly contained within the range r iff $\Pi(C_i) \subset r$ and C_i and r share no end point.*

Proof Direct consequence of the definition of a normalized set and that of a chain.

■

Figure 2–4 shows a normalized range set and its canonical partitioning into three chains.

Next we state a chopping rule that we use to transform every conflict-free range set R into an equivalent normalized range set $norm(R)$. By equivalent, we mean that for every destination d, the most-specific matching-range is the same in R as it is in $norm(R)$.

Definition 11 [Chopping Rule] *Let $r = [u, v] \in R$, where R is a range set. $chop(r, R)$ (or more simply $chop(r)$ when R is implicit), is as defined below.*

1. *If neither $maxP(u, v-1, R)$ nor $minP(u+1, v, R)$ exists, $chop(r) = r$.*
2. *If only $maxP(u, v-1, R)$ exists, $chop(r) = [maxP(u, v-1, R)+1, finish(r)]$.*
3. *If only $minP(u+1, v, R)$ exists, $chop(r) = [start(r), minP(u+1, v, R)-1]$.*
4. *If both $maxP(u, v-1, R)$ and $minP(u+1, v, R)$ exist and $maxP(u, v-1, R)+1 \leq minP(u+1, v, R)-1$, $chop(r) = [maxP(u, v-1, R)+1, minP(u+1, v, R)-1]$.*

5. *If both* $maxP(u, v-1, R)$ *and* $minP(u+1, v, R)$ *exist and* $maxP(u, v-1, R)+1 > minP(u+1, v, R) - 1$, $chop(r) = \emptyset$, *where* $\emptyset$ *denotes the null range. The null range neither intersects nor is contained in any other range.*

. *Define* $norm(R) = \{chop(r) | r \in R \wedge chop(r) \neq \emptyset\}$.

Lemma 12 *Let* R *be a conflict-free range set.*

$$\begin{aligned} \forall r \in R\ \forall s \in R[s \subset r \implies & [s \subset chop(r) \wedge start(s) \neq start(chop(r)) \\ & \wedge finish(s) \neq finish(chop(r))] \\ & \vee disjoint(s, chop(r))] \end{aligned}$$

Proof The lemma is trivially true when $chop(r) = \emptyset$ ($disjoint(s, \emptyset)$ is true). So, assume that $chop(r) = r'$. For the lemma to be false, either $intersect(s, r')$ or ($r' \subseteq s$ or s and r' have a common end point).

If $intersect(s, r')$, then either $start(r') < start(s) \leq finish(r') < finish(s)$ or $start(s) < start(r') \leq finish(s) < finish(r')$. Assume the former (the latter case is similar). From the chopping rule, it follows that $\exists A \subseteq R[\Pi(A) = [finish(r') + 1, finish(r)]$. Therefore, $A \cup \{s\} \subseteq R \wedge \Pi(A \cup \{s\}) = [start(s), finish(r)]$. From this, $start(r) \leq start(r') < start(s)$, and the chopping rule, we get $finish(chop(r)) < start(s)$. But, $start(s) \leq finish(r')$, a contradiction.

So, $r' \subseteq s$ or s and r' have a common end point. First consider the case $r' \subseteq s \subset r$. Suppose that $start(s) \neq start(r)$ (the case $finish(s) \neq finish(r)$ is similar). Since $r' = chop(r)$, $\exists A \subseteq R[\Pi(A) = [finish(r') + 1, finish(r)]$. Therefore, $\Pi(A \cup \{s\}) = [start(s), finish(r)]$ and $start(r) < start(s) \leq start(r')$. From the chopping rule, it follows that $finish(chop(r)) < start(s) \leq start(r') \leq finish(r')$, a contradiction. Therefore, $s \subset r'$. If $start(s) = start(r')$, $maxP(start(r), finish(r)-1) \geq finish(s)$. So, $start(r') > finish(s)$, which contradicts $s \subset r'$. The case $finish(s) = finish(r')$ is similar. ■

Lemma 13 *Let r and s be two intersecting ranges of a conflict-free range set R.*
$disjoint(chop(r), overlap(r,s)) \wedge disjoint(chop(s), overlap(r,s))$
$\wedge\ disjoint(chop(r), chop(s))$

Proof Without loss of generality, we may assume that $start(r) < start(s) <= finish(r) < finish(s)$. Since R is conflict free, $\exists A[A \subset R \wedge \Pi(A) = overlap(r,s)]$. Therefore, $finish(chop(r)) < start(s)$ and $start(chop(s)) > finish(r)$. This proves the lemma. ■

Lemma 14 *Let R be a conflict-free range set. For every $r' \in norm(R)$ there is a unique $r \in R$ such that $chop(r) = r'$.*

Proof Let r' be any range in $norm(R)$. Clearly, for every $r' \in norm(R)$, there is at least one $r \in R$ such that $chop(r) = r'$. Suppose two different ranges r and s of R have $r' = chop(r) = chop(s)$.

If $intersect(r,s)$, then from Lemma 13 we get $disjoint(chop(r), chop(s))$. So, $chop(r) \neq chop(s)$.

If $nested(r,s)$, then from Lemma 12 it follows that $s \subset chop(r) \vee disjoint(s, chop(r))$ when $s \subset r$ and $r \subset chop(s) \vee disjoint(r, chop(s))$ when $r \subset s$. Consider the former case (the latter case is similar). $s \subset chop(r)$ implies $chop(s) \neq chop(r)$. $disjoint(s, chop(r))$ also implies $chop(s) \neq chop(r)$.

The final case is $disjoint(r,s)$. In this case, clearly, $chop(s) \neq chop(r)$. ■

For $r' \in norm(R)$, define $full(r') = chop^{-1}(r') = r$, where r is the unique range in R for which $chop(r) = r'$. Notice that $full(chop(r)) = r$ except when $chop(r) = \emptyset$.

Lemma 15 *For every conflict-free range set R, $norm(R)$ is a normalized conflict-free range set.*

Proof We shall show that $norm(R)$ is normalized. Since a normalized range set has no intersecting ranges, every normalized range set is conflict free.

If $|norm(R)| \leq 1$, $norm(R)$ is normalized. So, assume that $|norm(R)| > 1$. Let r' and s' be two different ranges in $norm(R)$. We need to show that r' and s' satsify

property 2(a) or 2(b) of Definition 9. Let $r = [u, v] = full(r')$ and $s = full(s')$. There are three possible cases for r and s, they either intersect, are nested, or are disjoint (Lemma 1).

Case 1: $intersect(r, s)$. From Lemma 13, it follows that r' and s' are disjoint.

Case 2: $nested(r, s)$. Either $s \subset r$ or $r \subset s$. Assume the former (the latter case is similar). From Lemma 12, we get $[s \subset chop(r) \wedge start(s) \neq start(chop(r)) \wedge finish(s) \neq finish(chop(r))] \vee disjoint(s, chop(r))$

$s \subset chop(r) \wedge start(s) \neq start(chop(r)) \wedge finish(s) \neq finish(chop(r))$ implies that s' and r' are nested and do not have a common end-point. $disjoint(s, chop(r))$ implies that s' and r' are disjoint.

Case 3. $disjoint(r, s)$. Clearly, $disjoint(r', s')$. ■

Lemma 16 *Let $r' \in norm(R)$, where R is a conflict-free range set.*

$$\nexists s' \in norm(R)[s' \subset r'] \Longrightarrow r = full(r') = msr(r', R)$$

Proof Assume that $\nexists s' \in norm(R)[s' \subset r']$. If $\exists d \in r'[r \neq msr(d, R)]$, then $\exists s \subset r[d \in s]$. From Lemma 12, it follows that $s \subset r' \vee s \cap r' = \emptyset$. Since $d \in s \wedge d \in r'$, $s \cap r' \neq \emptyset$. Hence, $s \subset r'$. From Lemma 4, it follows that $A = \{t | t \in R \wedge t \subseteq s\} \neq \emptyset$ is conflict free. From the chopping rule it follows that $norm(A) \neq \emptyset$. So, $\exists t' \in norm(A) \subset norm(R)[t' \subseteq t = full(t') \subset r']$. This violates the assumption of this lemma. Therefore, $\nexists d \in r'[r \neq msr(d, R)]$. So, $r = msr(r', R)$. ■

Lemma 17 *Let R be a conflict-free range set, let x be the start point of some range in R, and let y be the finish point of some range in R.*

1. *Let $s \in R$ be such that $start(s) = x$ and $finish(s) = \min\{finish(t) | t \in R \wedge start(t) = x\}$*
   (a) *$chop(s) \neq \emptyset$.*
   (b) *$start(chop(s)) = x$.*
   (c) *$chop(s)$ is the only range in $norm(R)$ that starts at x.*

2. *Let* $s \in R$ *be such that* $finish(s) = y$ *and* $start(s) = \max\{start(t) | t \in R \wedge finish(t) = y\}$
   (a) $chop(s) \neq \emptyset$.
   (b) $finish(chop(s)) = y$.
   (c) $chop(s)$ *is the only range in* $norm(R)$ *that finishes at* y.

Proof We prove 1(a) - (c). 2(a) - (c) are similar. Since $maxP(start(s), finish(s) - 1, R)$ does not exist, case 5 of the chopping rule does not apply and $chop(s) \neq \emptyset$. One of the cases 1 and 3 applies. In both of these cases, $start(chop(s)) = x$. For 1(c), we note that the definition of a normalized set (Definition 9) implies that no two ranges of $norm(R)$ share an end point. In particular, $norm(R)$ can have only one range that has x as an end point. ■

Lemma 18 *Let* $r' \in norm(R)$*, where* R *is a conflict-free range set.*

$$start(full(r')) \neq start(r') \Longrightarrow \nexists s \in R[start(s) = start(r')]$$

Proof Suppose that $start(full(r')) \neq start(r')$ and $\exists s \in R[start(s) = start(r')]$. From Lemma 17(1a and 1b), it follows that $\exists t \in R[start(t) = start(r') \wedge chop(t) \neq \emptyset \wedge start(chop(t)) = start(r')]$. Therefore, $norm(R)$ has at least two ranges (r' and $chop(t)$) that start at $start(r')$. This contradicts Lemma 17(1c). ■

Lemma 19 *Let* R *be a conflict-free range set. Let* $r \in R$ *be such that* $r = msr(u, v, R)$ *for some range* $[u, v]$. $r' = chop(r) = msr(u, v, norm(R))$.

Proof From the definition of msr, it follows that there is no $s \in R$ such that $s \subset r \wedge s \cap [u, v] \neq \emptyset$. Therefore, $[u, v] \subseteq chop(r)$. Further, from Lemmas 12 and 13, it follows that $norm(R)$ contains no $s' \subset chop(r)$. So, $r' = msr(u, v, norm(R))$. ■

Lemma 20 *Let* R *be a conflict-free range set that has a subset whose projection equals* $[x, y]$. *Let* $A \subseteq R$ *comprise all* $r \in R$ *such that* $r \subseteq [x, y]$.

1. $\exists B \subseteq norm(R)[\Pi(B) = [x, y]]$
2. $C = \{full(r') | r' \in norm(R) \wedge r' \subseteq [x, y]\} \subseteq A$

Proof

1. From Lemma 4, it follows that A is conflict free. Further, since R has a subset whose projection equals $[x, y]$, $\Pi(A) = [x, y]$. From Lemma 19, it follows that every $d \in [x, y]$ has a most-specific range in $norm(A)$. Therefore, $\Pi(norm(A)) = [x, y]$. From the definition of the chopping rule and that of A, we see that $\forall r \in A[chop(r, A) = chop(r, R)]$. So, $norm(A) \subseteq norm(R)$.
2. First, assume that $[x, y] \in R$. Suppose there is a range $r' \in norm(R)$ such that $r' \subseteq [x, y]$ and $r = full(r') \notin A$. There are three cases for r.

   Case 1: $disjoint(r, [x, y])$. In this case, $disjoint(r', [x, y])$ and so $r' \not\subseteq [x, y]$.

   Case 2: $intersect(r, [x, y])$. From Lemma 13, we get $disjoint(chop(r), [x, y])$. So $r' \not\subseteq [x, y]$.

   Case 3: $[x, y] \subset r$. From Lemma 12 and $r' \subseteq [x, y]$, we get $disjoint([x, y], chop(r))$. So $r' \not\subseteq [x, y]$.

   When $[x, y] \notin R$, let $R' = R \cup \{[x, y]\}$, $C' = \{full(r') | r' \in norm(R') \wedge r' \subseteq [x, y]\}$ and $A' = A \cup \{[x, y]\}$. Using the lemma case we have already proved, we get $C' \subseteq A'$. Since $chop([x, y], R') = \emptyset$ and $chop(s, R) = chop(s, R')$ for every $s \in R$, $norm(R') = norm(R)$. Therefore, $C = C'$. So, $C \subseteq A'$. Finally, since $[x, y] \notin C$, $C \subseteq A$.

■

Lemma 21 *Let R be a conflict-free range set. Let $r \notin R$ be such that $R \cup \{r\}$ is conflict free.*

1. $chop(r, R \cup \{r\}) = \emptyset \Longrightarrow \forall t \in R[chop(t, R) = chop(t, R \cup \{r\})]$.
2. *Let s be the smallest range of R that contains r. Assume that s exists and that $chop(r, R \cup \{r\}) \neq \emptyset$.*
   (a) $\forall t \in R - \{s\}[chop(t, R) = chop(t, R \cup \{r\})]$.

(b) $chop(s, R) \neq chop(s, R \cup \{r\}) \Longrightarrow (x' = u' \wedge y' = v') \vee (x' = u' \wedge y' > v) \vee (x' < u \wedge y' = v')$, *where* $r = [u, v]$, $chop(r, R \cup \{r\}) = chop(r, R) = [u', v']$, *and* $chop(s, R) = [x', y']$.

Proof For (1), note that $chop(r, R \cup \{r\}) = \emptyset \Longrightarrow \exists A \subseteq R[\Pi(A) = r]$. Therefore, the addition of r to R does not affect any of the $maxP$ and $minP$ values.

For (2a), suppose there are two different ranges g and h in R such that $chop(g, R) \neq chop(g, R \cup \{r\})$ and $chop(h, R) \neq chop(h, R \cup \{r\})$ From the chopping rule, it follows that

$$r \subset g \wedge r \subset h \tag{2.6}$$

Therefore, $\neg disjoint(g, h)$. From this and Lemma 1, we get $intersect(g, h) \vee nested(g, h)$. Equation 2.6 and $intersect(g, h)$ imply $r \subseteq overlap(g, h)$. From this and Lemma 13, we get $disjoint(r, chop(g, R)) \wedge disjoint(r, chop(h, R))$. Therefore, $chop(g, R) = chop(g, R \cup \{r\})$ and $chop(h, R) = chop(h, R \cup \{r\})$, a contradiction. So, $\neg intersect(g, h)$.

If $nested(g, h)$, we may assume, without loss of generality, that $g \subset h$. This and Equation 2.6 yield $r \subset g \subset h$. Therefore, $maxP(x, y-1, R) = maxP(x, y-1, R \cup \{r\})$ and $minP(x+1, y, R) = minP(x+1, y, R \cup \{r\})$, where $h = [x, y]$. So, $chop(h, R) = chop(h, R \cup \{r\})$, a contradiction.

Hence, there can be at most one range of R whose $chop()$ value changes as a result of the addition of r. The preceding proof for the case $nested(g, h)$ also establishes that the $chop()$ value may change only for the range s, that is for the smallest enclosing range of r (i.e., smallest $s \in R[r \subset s]$).

For (2b), assume that $chop(s, R) \neq chop(s, R \cup \{r\})$. This implies that $chop(s, R) \neq \emptyset$ and so x' and y' are well defined. (Note that from part (1), we get $chop(r, R) \neq \emptyset$.) We consider each of the three cases for the relationship between r and $chop(s, R)$ (Lemma 1).

Case 1: $disjoint(r, chop(s, R))$. This case cannot arise, because then $chop(s, R) = chop(s, R \cup \{r\})$.

Case 2: $intersect(r, chop(s, R))$. Now, either $x' < u \leq y' < v$ or $u < x' \leq v < y'$. Consider the former case. Since $r \subset s$, $v \leq y$. When $v = y$, $minP(u+1, v, R \cup \{r\}) = minP(x+1, y, R) = y'+1$. So, $v' = y'$. Therefore, $x' < u \wedge y' = v'$. Consider the case $v < y$. From the chopping rule, it follows that $\exists A \subseteq R \subset R \cup \{r\}[\Pi(A) = [y'+1, y]]$. From this, Lemma 5(2), and the fact that $R \cup \{r\}$ is conflict free, we conclude $\exists B \in R \cup \{r\}[\Pi(B) = overlap(r, [y'+1, y]) = [y'+1, v]]$. From this and $minP(x+1, y, R) = y'+1$, we get $minP(u+1, v, R \cup \{r\}) = y'+1$. So, $v' = y'$. Once again, $x' < u \wedge y' = v'$. Using a similar argument, we may show that when $u < x' \leq v < y'$, $x' = u' \wedge y' > v$.

Case 3: $nested(r, chop(s, R))$. So, either $r \subseteq chop(s, R)$ or $chop(s, R) \subset r$. First, consider all possibilities for $r \subseteq chop(s, R)$. The case $x' < u \leq v < y'$ cannot arise, because this implies $chop(s, R) = chop(s, R \cup \{r\})$. When $x' = u \leq v < y'$, $u' = x'$. So, $x' = u' \wedge y' > v$. When $x' < u \leq v = y'$, $v' = y'$. So, $x' < u \wedge y' = v'$. The final case is when $x' = u \leq v = y'$. Now, $u' = x' \wedge y' = v'$.

Using an argument similar to that used in part (2a), we may show that when $chop(s, R) \subset r$, $x' = u' \wedge y' = v'$. ■

Lemma 22 *Let R, $r = [u, v]$, $s = [x, y]$, x', y', u' and v' be as in Lemma 21. Assume that s exists and $chop(s) \neq \emptyset$.*

1. $disjoint(r, chop(s, R)) \vee x' < u \leq v < y' \Longrightarrow chop(s, R \cup \{r\}) = chop(s, R)$.
2. $x' = u' \wedge y' = v' \Longrightarrow chop(s, R \cup \{r\}) = \emptyset$.
3. *Suppose $x' = u' \wedge y' > v$. If $maxP(v'+1, y', R)$ doesn't exist, then $chop(s, R \cup \{r\}) = [v+1, y']$. If it exists, $chop(s, R \cup \{r\}) = [maxP(v'+1, y', R)+1, y']$.*
4. *Suppose $x' < u' \wedge y' = v'$. If $minP(x', u'-1, R)$ doesn't exist, then $chop(s, R \cup \{r\}) = [x', u-1]$. If it exists, $chop(s, R \cup \{r\}) = [x', minP(x', u'-1, R)-1]$.*

Proof (1) follows from the proof of Lemma 21(2b). For (2), from the proof cases of Lemma 21(2b) that have $x' = u' \wedge y' = v'$, it follows that case 5 of the chopping rule applies for s in $R \cup \{r\}$. So, $chop(s, R \cup \{r\}) = \emptyset$.

For (3), $finish(chop(s, R \cup \{r\})) = y'$ follows from the proof of Lemma 21(2b). Also, we observe that $maxP(x, y-1, R \cup \{r\}) \geq v$. So, (3b) can be false only when $maxP(x, y-1, R \cup \{r\}) > v$ and either (a) $maxP(v'+1, y', R)$ doesn't exist or (b) $maxP(v'+1, y', R) < maxP(x, y-1, R \cup \{r\})$. For (a), $\exists [c, d] \in R[x \leq c \leq v' \wedge v < d < y']$. For (b), $\exists [c, d] \in R[x \leq c \leq v' \wedge v < maxP(v'+1, y', R) < d < y']$. In both cases, $c \leq u$ implies that $r = [u, v] \subset [c, d] \subset s$. This contradicts the assumption that s is the smallest enclosing range of r. Also, in both cases, $c > u$ implies $intersect(r, [c, d])$. So, $R \cup \{r\}$ has a subset whose projection is $[c, v]$. Therefore, $finish(chop(u, v, R \cup \{r\})) < c \leq v'$, a contradiction.

The proof for (4) is similar to that for (3). ■

Lemma 23 *Let R be a conflict-free range set. Let $r = [u, v] \in R$ be such that $R - \{r\}$ is conflict free.*

1. $chop(r, R) = \emptyset \Longrightarrow \forall t \in R - \{r\}[chop(t, R) = chop(t, R - \{r\})]$.
2. *Let $s = [x, y]$ be the smallest range of $R - \{r\}$ that contains r. Assume that s exists and that $chop(r, R) = [u', v']$.*
   (a) $\forall t \in R - \{r, s\}[chop(t, R) = chop(t, R - \{r\})]$.
   (b) $chop(s, R) = \emptyset \Longrightarrow chop(s, R - \{r\}) = [u', v']$.
   (c) $chop(s, R) = [x', y'] \Longrightarrow chop(s, R - \{r\}) = [\min\{x', u'\}, \max\{y', v'\}]$.

Proof For (1), note that $chop(r, R) = \emptyset \Longrightarrow \exists A \subseteq R - \{r\}[\Pi(A) = r]$. Therefore, the removal of r from R does not affect any of the $maxP$ and $minP$ values.

For (2a) note that by substituting $R - \{r\}$ for R in Lemma 21(2a), we get $\forall t \in R - \{r, s\}[chop(t, R - \{r\}) = chop(t, R)]$. (2b) and (2c) follow from Lemma 22.

■

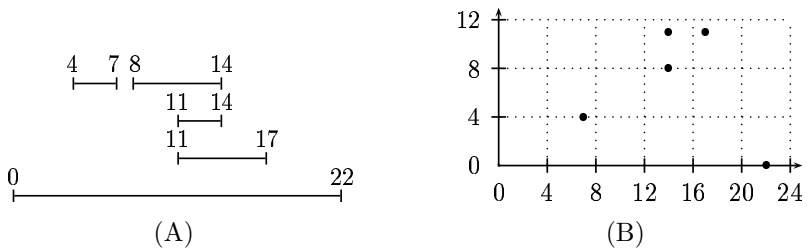


Figure 2–5: An example range set R and its mapping $map1(R)$ into points in 2D

2.1.5 Priority Search Trees And Ranges

A priority-search tree (PST) [37] is a data structure that is used to represent a set of tuples of the form $(key1, key2, data)$, where $key1 \geq 0$, $key2 \geq 0$, and no two tuples have the same $key1$ value. The data structure is simultaneously a min-tree on $key2$ (i.e., the $key2$ value in each node of the tree is $\leq$ the $key2$ value in each descendent node) and a search tree on $key1$. There are two common PST representations [37]:

1. In a **radix priority-search tree** (RPST), the underlying tree is a binary radix tree on $key1$.
2. In a **red-black priority-search tree** (RBPST), the underlying tree is a red-black tree.

McCreight [37] has suggested a PST representation of a collection of ranges with distinct finish points. This representation uses the following mapping of a range r into a PST tuple:

$$(key1, key2, data) = (finish(r), start(r), data) \tag{2.7}$$

where $data$ is any information (e.g., next hop) associated with the range. Each range r is, therefore mapped to a point $map1(r) = (x, y) = (key1, key2) = (finish(r), start(r))$ in 2-dimensional space. Figure 2–5 shows a set of ranges and the equivalent set of 2-dimensional points (x, y).

McCreight [37] has observed the when the mapping of Equation 2.7 is used to obtain a point set $P = map1(R)$ from a range set R, then $ranges(d)$ is given by

the points that lie in the rectangle (including points on the boundary) defined by $x_{left} = d$, $x_{right} = \infty$, $y_{top} = d$, and $y_{bottom} = 0$. These points are obtained using the method $enumerateRectangle(x_{left}, x_{right}, y_{top}) = enumerateRectangle(d, \infty, d)$ of a PST (y_{bottom} is implicit and is always 0).

When an RPST is used to represent the point set P, the complexity of

$$enumerateRectangle(x_{left}, x_{right}, y_{top})$$

is $O(\log maxX + s)$, where $maxX$ is the largest x value in P and s is the number of points in the query rectangle. When the point set is represented as an RBPST, this complexity becomes $O(\log n + s)$, where $n = |P|$. A point (x, y) (and hence a range $[y, x]$) may be inseted into or deleted from an RPST (RBPST) in $O(\log maxX)$ ($O(\log n)$) time [37].

2.2 Prefixes

Let R be a set of ranges such that each range represents a prefix. It is well known (see Sahni et al. [21], for example) that no two ranges of R intersect. Therefore, R is conflict free. For simplicity, assume that R includes the range that corresponds to the prefix *. With this assumption, $msr(d)$ is defined for every d. From Lemma 9, it follows that $msr(d)$ is the range $[maxStart(ranges(d)), minFinish(ranges(d))]$. To find this range easily, we first transform $P = map1(R)$ into a point set $transform1(P)$ so that no two points of $transform1(P)$ have the same x-value. Then, we represent $transform1(P)$ as a PST.

Definition 12 *Let W be the (maximum) number of bits in a destination address ($W = 32$ in IPv4). Let $(x, y) \in P$. $transform1(x, y) = (x', y') = (2^W x - y + 2^W - 1, y)$ and $transform1(P) = \{transform1(x, y) | (x, y) \in P\}$.*

We see that $0 \leq x' < 2^{2W}$ for every $(x', y') \in transform1(P)$ and that no two points in $transform1(P)$ have the same x'-value. Let $PST1(P)$ be the PST for

$transform1(P)$. The operation

$$enumerateRectangle(2^W d - d + 2^W - 1, \infty, d)$$

performed on $PST1$ yields $ranges(d)$. To find $msr(d)$, we employ the

$$minXinRectangle(x_{left}, x_{right}, y_{top})$$

operation, which determines the point in the defined rectangle that has the least x-value. It is easy to see that

$$minXinRectangle(2^W d - d + 2^W - 1, \infty, d)$$

performed on $PST1$ yields $msr(d)$.

To insert the prefix whose range in $[u, v]$, we insert $transform1(map1([u, v]))$ into $PST1$. In case this prefix is already in $PST1$, we simply update the next-hop information for this prefix. To delete the prefix whose range is $[u, v]$, we delete $transform1(map1([u, v]))$ from $PST1$. When deleting a prefix, we must take care not to delete the prefix *. Requests to delete this prefix should simply result in setting the next-hop associated with this prefix to $\emptyset$.

Since, $minXinRectangle$, insert, and delete each take $O(W)$ ($O(\log n)$) time when $PST1$ is an RPST (RBPST), $PST1$ provides a router-table representation in which longest-prefix matching, prefix insertion, and prefix deletion can be done in $O(W)$ time each when an RPST is used and in $O(\log n)$ time each when an RBPST is used.

2.3 Nonintersecting Ranges

Let R be a set of nonintersecting ranges. Clearly, R is conflict free. For simplicity, assume that R includes the range z that matches all destination addresses ($z =$

$[0, 2^{32} - 1]$ in the case of IPv4). With this assumption, $msr(d)$ is defined for every d. We may use $PST1(transform1(map1(R)))$ to find $msr(d)$ as described in Section 2.2.

Insertion of a range r is to be permitted only if r does not intersect any of the ranges of R. Once we have verified this, we can insert r into $PST1$ as described in Section 2.2. Range intersection may be verified by noting that there are two cases for range intersection (Definition 2(c)). When inserting $r = [u, v]$, we need to determine if $\exists s = [x, y] \in R[u < x \leq v < y \vee x < u \leq y < v]$. We see that $\exists s \in R[x < u \leq y < v]$ iff $map1(R)$ has at least one point in the rectangle defined by $x_{left} = u$, $x_{right} = v - 1$, and $y_{top} = u - 1$ (recall that $y_{bottom} = 0$ by default). Hence, $\exists s \in R[x < u \leq y < v]$ iff $minXinRectangle(2^W u - (u - 1) + 2^W - 1, 2^W(v - 1) + 2^W - 1, u - 1)$ exists in PST1.

To verify $\exists s \in R[u < x \leq v < y]$, map the ranges of R into 2-dimensional points using the mapping, $map2(r) = (start(r), 2^W - 1 - finish(r))$. Call the resulting set of mapped points $map2(R)$. We see that $\exists s \in R[u < x \leq v < y]$ iff $map2(R)$ has at least one point in the rectangle defined by $x_{left} = u + 1$, $x_{right} = v$, and $y_{top} = (2^W - 1) - v - 1$. To verify this, we maintain a second PST, $PST2$ of points in $transform2(map2(R))$, where $transform2(x, y) = (2^W x + y, y)$ Hence, $\exists s \in R[u < x \leq v < y]$ iff $minXinRectangle(2^W(u+1), 2^W v + (2^W - 1) - v - 1, (2^W - 1) - v - 1)$ exists.

To delete a range r, we must delete r from both $PST1$ and $PST2$. Deletion of a range from a PST is similar to deletion of a prefix as discussed in Section 2.2.

The complexity of the operations to find $msr(d)$, insert a range, and delete a range are the same as that for these opertions for the case when R is a set of ranges that correspond to prefixes.

Step 1: If $r = [u, v] \in R$, update the next-hop information associated with $r \in R$ and terminate.

Step 2: Compute $maxP(u, v, R)$, $minP(u, v, R)$, $maxY(u, v, R)$ and $minX(u, v, R)$.

Step 3: If $maxY(u, v, R) \leq maxP(u, v, R) \wedge minX(u, v, R) \geq minP(u, v, R)$, $R \cup \{r\}$ is conflict free; otherwise, it is not. In the former case, insert $transform1(map1(r))$ into $PST1$ and $transform2(map2(r))$ into $PST2$. In the latter case, the insert operation fails.

Figure 2–6: Insert $r = [u, v]$ into the conflict-free range set R

2.4 Conflict-Free Ranges

In this section, we extend the two-PST data structure of Section 2.3 to the general case when R is an arbitrary conflict-free range set. Once again, we assume that R includes the range z that matches all destination addresses. $PST1$ and $PST2$ are defined for the range set R as in Sections 2.2 and 2.3.

2.4.1 Determine $msr(d)$

Since R is conflict free, $msr(d)$ is determined by Lemma 9. Hence, $msrd(d)$ may be obtained by performing the operation

$$minXinRectangle(2^W d - d + 2^W - 1, \infty, d)$$

on PST1.

2.4.2 Insert A Range

When inserting a range $r = [u, v] \notin R$, we must insert $transform(map1(r))$ into $PST1$ and $transform2(map2(r))$ into $PST2$. Additionally, we must verify that $R \cup \{r\}$ is conflict free. This verification is done using Lemma 6. Figure 2–6 gives a high-level description of our algorithm to insert a range into R.

Step 1 is done by searching for $transform1(map1(r))$ in $PST1$. For Step 2, we note that

$$maxY(u, v, R) = maxXinRectangle(2^W u - (u-1) + 2^W - 1, 2^W(v-1) + 2^W - 1, u-1)$$

$$minX(u, v, R) = minXinRectangle(2^W(u+1), 2^W v + (2^W - 1) - v - 1, (2^W - 1) - v - 1)$$

Step 1: If $r = z$, change the next-hop information for z to $\emptyset$ and terminate.
Step 2: Delete $transform1(map1(r))$ from $PST1$ and $transform2(map2(r))$ from $PST2$ to get the PSTs for $A = R - \{r\}$. If $PST1$ did not have $transform1(map1(r))$, $r \notin R$; terminate.
Step 3: Determine whether or not A has a subset whose projection equals $r = [u, v]$.
Step 4: If A has such a subset, conclude $conflictFree(A)$ and terminate.
Step 5: Determine whether A has a range that contains $r = [u, v]$. If not, conclude $conflictFree(A)$ and terminate.
Step 6: Determine m and n as defined in Lemma 8 as follows.
$m = start(maxXinRectangle(0, 2^W u + (2^W - 1) - v, (2^W - 1) - v)$ (use PST2)
$n = finish(minXinRectangle(2^W v - u + 2^W - 1, \infty, u)$ (use PST1)

Step 7: Determine whether $[m, n] \in A$. If so, conclude $conflictFree(A)$. Otherwise, conclude $\neg conflictFree(A)$. In the latter case reinsert $transform1(map1(r))$ into $PST1$ and $transform2(map2(r))$ into $PST2$ and disallow the deletion of r.

Figure 2–7: Delete the range $r = [u, v]$ from the conflict-free range set R

where for $maxY$ we use $PST1$ and for $minX$ we use $PST2$. Section 2.4.4 describes the computation of $maxP$ and $minP$. The point insertions of Step 3 are done using the standard insert algorithm for a PST [37].

2.4.3 Delete A Range

Suppose we are to delete the range $r = [u, v]$. This deletion is to be permitted iff $r \neq z$ and $A = R - \{r\}$ is conflict free. Figure 2–7 gives a high-level description of our algorithm to delete r. Its correctness follows from Lemma 8.

Step 2 employs the standard PST algorithm to delete a point [37]. For Step 3, we note that A has a subset whose projection equals $r = [u, v]$ iff $maxP(u, v, A) = v$. In Section 2.4.4, we show how $maxP(u, v, A)$ may be computed efficiently. For Step 5, we note that $r = [u, v] \subseteq s = [x, y]$ iff $x \leq u \wedge y \geq v$. So, A has such a range iff

$$minXinRectangle(2^W v - u + 2^W - 1, \infty, u)$$

exists in $PST1$.

In Step 6, we assume that $maxXinRectangle$ and $minXinRectangle$ return the range of R that corresponds to the desired point in the rectangle. To determine

Step 1: Find $r' \in norm(R)$ such that $start(r') = u$.
If no such r' or $start(full(r')) \neq u \vee finish(full(r')) > v$, $maxP$ does not exist; terminate.
Step 2: $maxP = finish(r')$;
while $(s' \in norm(R) \wedge start(s') = maxP + 1 \wedge (full(s') \subseteq [u, v])$
$maxP = finish(s')$;

Figure 2–8: Simple algorithm to compute $maxP(u, v, R)$, where $[u, v]$ is a range and $conflictFree(R)$

whether $[m, n] \in A$ (Step 7), we search for the point $(2^W n - m + 2^W - 1, m)$ in $PST1$ using the standard PST search algorithm [37]. The reinsertion into $PST1$ and $PST2$, if necessary, is done using the standard PST insert algorithm [37].

2.4.4 Computing $maxP$ and $minP$

Although $maxP$ and $minP$ are relatively difficult to compute using data structures such as $PST1$ and $PST2$ that directly represent R, they may be computed efficiently using data structures for $norm(R)$. In this section, we show how to compute $maxP$ from $norm(R)$. The computation of $minP$ is similar.

2.4.5 A Simple Algorithm to Compute $maxP$

Figure 2–8 is a high-level description of a simple, though not efficient, algorithm to compute $maxP(u, v, R)$.

Theorem 1 *Figure 2–8 correctly computes $maxP(u, v, R)$.*

Proof First consider Step 1. From Lemma 17(a), it follows that

$$\nexists r' \in norm(R)[start(r') = u] \Longrightarrow \nexists r \in R[start(r) = u]$$

. Therefore, $\nexists r' \in norm(R)[start(r') = u] \Longrightarrow \nexists maxP$. From Lemma 18, it follows that $start(full(r')) \neq start(r') = u \Longrightarrow \nexists s \in R[start(s) = start(r') = u]$. So, $start(full(r')) \neq u \Longrightarrow \nexists maxP$. Finally, $u = start(r') = start(full(r))$ implies $finish(full(r')) = \min\{finish(t) | t \in R \wedge start(t) = u\}$ (Lemma 17(1)). So, $finish(full(r')) > v$ implies $\nexists s \in R[start(s) = u \wedge finish(s) \leq v]$. Hence, $start(r') = u \wedge finish(full(r')) > v \Longrightarrow \nexists maxP$. Further, when

$\exists r' \in norm(R)[start(r') = u \wedge finish(full(r')) \leq v]$, $maxP$ exists and $maxP \geq finish(full(r')) \geq finish(r')$. Therefore, Step 1 correctly identifies the case when $maxP$ doesn't exist.

We get to Step 2 only when $maxP$ exists. From the definition of $maxP$, $\exists A \subseteq R[\Pi(A) = [u, maxP]]$. From this and Lemma 20(1), we get $\exists B \subseteq norm(R)[\Pi(B) = [u, maxP]]$. Now, from Lemma 10, we get $\exists D \subseteq norm(R)[chain(D) \wedge \Pi(D) = [u, maxP]]$. From Lemma 11, it follows that D is a sub-chain of the unique chain $C_i \in CP(norm(R))$ that includes r'. Let r', s'_1, s'_2, ..., s'_q be the tail of C_i. It follows that $maxP$ is either $finish(r')$ or $finish(s'_j)$ for some j in the range $[1, q]$. Let j be the least integer such that $full(s'_j) \not\subseteq [u, v]$. If such a j does not exist, then $maxP = finish(s'_q)$ as $norm(R)$ has no subset whose projection equals $[u, x]$ for any $x > finish(s'_q)$. So, assume that j exists. From Lemma 20(2), it follows that $maxP < finish(s'_j)$. Hence, Step 2 correctly determines $maxP$. ■

2.4.6 An Efficient Algorithm to Compute $maxP$

The algorithm of Figure 2–8 takes time $O(length(C_i))$, where $length(C_i)$ is the number of ranges in the chain $C_i \in CP(norm(R))$ that contains r'. We can reduce this time to $O(\log length(C_i))$ by representing each chain of $CP(norm(R))$ as a red-black tree (actually any balanced search tree structure that permits efficient join and split operations may be used). The number of red-black trees we use equals the number of chains in $CP(norm(R))$.

Let $D = (t'_1, ..., t'_q)$ be a chain in $CP(norm(R))$. The red-black tree, $RBT(D)$, for D has one node for each of the ranges t'_i. The key value for the node for t'_i is $start(t'_i)$ (equivalently, $finish(t'_i)$ may be used as the search tree key). Each node of $RBT(D)$ has the following four values (in addition to having a t'_i and other values necessary for efficient implementation): $minStartLeft$, $minStartRight$, $maxFinishLeft$, and $maxFinishRight$. For a node p that has an empty left subtree, $minStartLeft = 2^W - 1$ and $maxFinishLeft = 0$. Similarly, when p has an empty right subtree,

$minStartRight = 2^W - 1$ and $maxFinishRight = 0$. Otherwise,

$$\begin{aligned} minStartLeft &= \min\{start(full(r'))|r' \in leftSubtree(p)\} \\ minStartRight &= \min\{start(full(r'))|r' \in rightSubtree(p)\} \\ maxFinishLeft &= \max\{finish(full(r'))|r' \in leftSubtree(p)\} \\ maxFinishRight &= \max\{finish(full(r'))|r' \in rightSubtree(p)\} \end{aligned}$$

The collection of red-black trees representing $norm(R)$ is augmented by an additional red-black tree $endPointsTree(norm(R))$ that represents the end points of the ranges in $norm(R)$. With each point x in $endPointsTree$, we store a pointer to the node in $RBT(D)$ that represents s'. Alternatively, we may use a PST, $PST3$, for the range set $chains = \{[start(C_i), finish(C_i)]|C_i \in CP(norm(R))\}$. The points in $PST3$ are $map1(chains)$; with each point in $PST3$, we keep a pointer to the root of the RBT for that chain. Note that since range end-points are distinct in $chains$, we do not need to use $transform1$ as used in $PST1$. To find an end point d, we first find the smallest chain that covers d by performing the operation $minXinRectangle(d, \infty, d)$ in $PST3$. Next, we follow the pointer associated with this chain to get to the corresponding RBT. Finally, a search of this RBT gets us to the RBT node for the s' with the given end point. In the sequel, we assume that $endPointsTree$, rather than $PST3$, is used. A parallel discussion is possible for the case when $PST3$ is used.

To implement Step 1 of Figure 2–8, we search $endPointsTree$ for the point u. If $u \notin endPointsTree$, then $\nexists r' \in norm(R)[start(r') = u]$. If $u \in endPointsTree$, then we use the pointer in the node for u to get to the root of the RBT that has r'. A search in this RBT for u locates r'. We may now perform the remaining checks of Step 1 using the data associated with r'.

Suppose that $maxP$ exists. At the start of Step 2, we are positioned at the RBT node that represents r'. This is node 0 of Figure 2–9. We need to find $s' \in norm(R)$

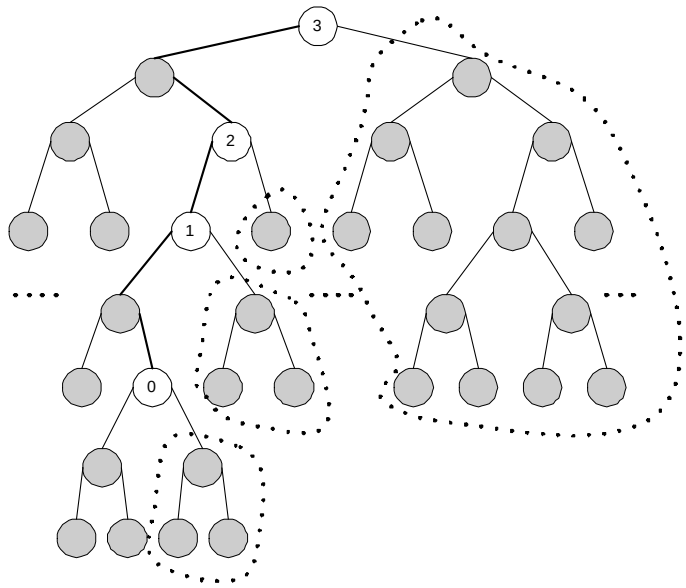


Figure 2–9: An example RBT

with least s' such that $start(s') > finish(r') \wedge full(s') \not\subseteq [u, v]$. If there is no such s', then $maxP = \max\{finish(root.range), root.maxFinishRight\}$. If such an s' exists, $maxP = start(s') - 1$.

s' may be found in $O(height(RBT))$ time using a simple search process. We illustrate this process using the tree of Figure 2–9. We begin at node 0. If $[minStartRight, maxFinishRight] \subseteq [u, v]$, then s' is not in the right subtree of node 0. Since node 0 is a right child, s' is not in its parent. So, we back up to node 1 (in general, we back up to the nearest ancestor whose left subtree contains the current node). Let t'_1 be the range in node 1. $s' = t'$ iff $t' \not\subseteq [u, v]$. If $s' \neq t'$, we perform the test $[minStartRight, maxFinishRight] \subseteq [u, v]$ at node 1 to determine whether or not s' is in the right subtree of node 1. If the test is true, we back up to node 2. Otherwise, s' is in the right subtree of node 1. When the right subtree (if any) that contains s' is identified, we make a downward pass in this subtree to locate s'. Figure 2–10 describes this downward pass.

```
downwardPass(currentNode)
  // currentNode is the root of a subtree all of whose ranges start at the right of u
  // This subtree contains s'. Return maxP.
  while (true) {
    if ([currentNode.minStartLeft, currentNode.maxFinishRight] ⊆ [u, v])
      // s' not in left subtree
      if (currentNode.range ⊆ [u, v])
        // s' ∉ currentNode. s' must be in right subtree.
        currentNode = currentNode.rightChild;
      else return (start(currentNode.range) − 1);
    else // s' is in left subtree
      currentNode = currentNode.leftChild;
  }
```

Figure 2–10: Find s' (and hence $maxP$) in a subtree known to contain s'

2.4.7 Wrapping Up Insertion of a Range

Now that we have augmented $PST1$ and $PST2$ with a collection of RBTs and an $endPointsTree$, whenever we insert a range $r = [u, v]$ into R, we mut update not only $PST1$ and $PST2$ as described in Section 2.4.2, but also the RBT collection and $endPointsTree$. To do this, we first compute $chop(r, R \cup \{r\}) = chop(r, R) = [u', v']$ by first computing $minP(u + 1, v)$ and $maxP(u, v - 1)$ as described in Section 2.4.4. $[u', v']$ is now easily obtained from the chopping rule. Lemma 21 tells us that the only $s \in R$ whose $chop()$ value may change as a result of the insertion of r is the smallest enclosing range of r. Since $z \in R$ and $r \neq z$, such an s must exist. Rather than search for this s explicitly, we use the cases (2)–(4) conditions of Lemma 22 to find $s' = chop(s, R)$ in $endPointsTree$. Note that if $chop(s, R) = \emptyset$, the search in $endPointsTree$ will not find s; but when $chop(s, R) = \emptyset$, $chop(s, R \cup \{r\}) = \emptyset$. So, no change in $chop(s, R)$ is called for.

Note that the insertion of r may combine two chains of $CP(norm(R))$. In this case, we use the *join* operation of red-black trees to combine the RBTs corresponding to these two chains.

2.4.8 Wrapping Up Deletion of a Range

When $chop(r, R) = \emptyset$, no chnges are to be made to the RBTs and $endPointsTree$ (Lemma 23(1)). So, assume that $chop(r, R) \neq \emptyset$. We first find s, the smallest range that contains r (see Lemma 23(2)). Note that since $z \in R$ and $r \neq z$, s exists. One may verify that s is one of the ranges given by the following two operations.

$$minXinRectangle(2^W v - u + 2^W - 1, \infty, u)$$

$$maxXinRectangle(0, 2^W u + 2^W - 1 - v, 2^W - 1 - v)$$

where the first operation is done in $PST1$ and the second in $PST2$ (both operations are done after $transform1(map1([u, v]))$ has been deleted from $PST1$ and $transform2(map2([u, v]))$ has been deleted from $PST2$). The ranges returned by these two operations may be compared to determine which is s.

Once we have identified s, Lemma 23(2) is used to determine $chop(s, R-\{r\})$. Assume that $chop(s, R) \neq \emptyset$. Let $chop(r, R) = r' = [u', v']$ and $chop(s, R) = s' = [x', y']$. When s' and r' are in different RBTs (this is the case when $r' \subset s'$, $chop(s, R) = chop(s, R - \{r\})$ and the RBT that contains s' may need to be split into two RBTs. When s' and r' are in the same RBT, they are in the same chain of $CP(norm(R))$. If s' are r' are adjacent ranges of this chain, we may simply remove the RBT node for r' and update that for s' to reflect its new start or finish point (only one may change). When r' and s' are not adjacent ranges, the nodes for these two ranges are removed from the RBT (this may split the RBT into up to two RBTs) and $chop(s, R - \{r\})$ inserted. Figure 2–11 shows the different cases.

2.4.9 Complexity

The portions of the search, insert, and delete algorithms that deal only with $PST1$ and $PST2$ have the same asymptotic complexity as their counterparts for the case of nonintersecting ranges (Section 2.3). The portions that deal with the RBTs and $endPointsTree$ require a constant number of search, insert, delete, join, and split

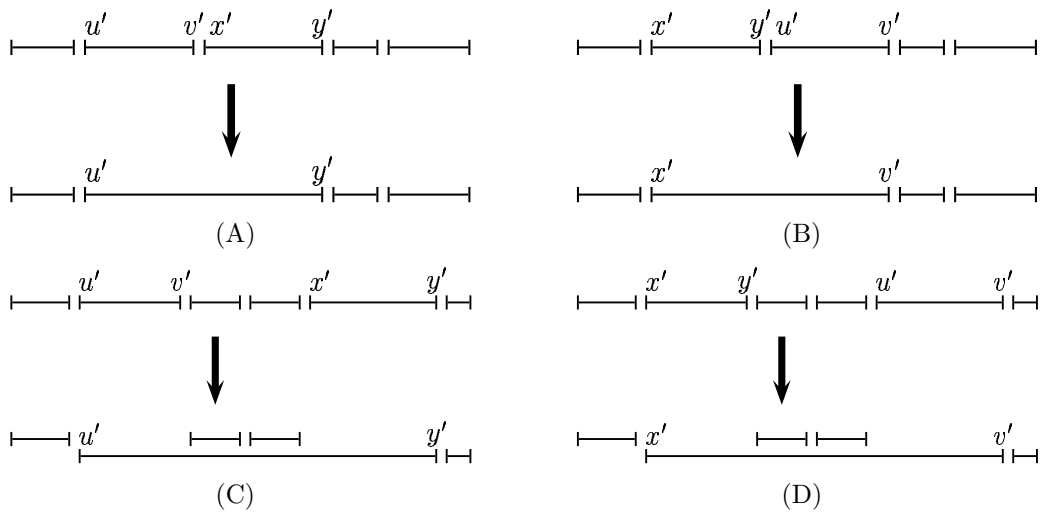


Figure 2–11: Cases when s' and r' are in the same chain of $CP(norm(R))$

operations on these structures. Since each of these operations takes $O(\log n)$ time on a red-black tree and since we can update the values $minStartLeft$, $minStartRight$, and so on, that are stored in the RBT nodes in the same asymptotic time as taken by an insert/delete/join/split, the overall complexity of our proposed data structure is $O(\log n)$ for each operation when $RBPST$s are used for $PST1$ and $PST2$. When $RPST$s are used, the search complexity is $O(W)$ and the insert and delete complexity is $O(W + \log n) = O(W)$.

2.5 Experimental Results

2.5.1 Prefixes

We programmed our red-black priority-search tree algorithm for prefixes (Section 2.2) in C++ and compared its performance to that of the ACBRT of Sahni et al. [22]. Recall that the ACBRT is the best performing $O(\log n)$ data structure reported in [22] for dynamic prefix-tables. For test data, we used six IPv4 prefix databases obtained from [38]. The number of prefixes in each of these databases as well as the memory requirements for each database of prefixes using our data structure (PST) of Section 2.2 as well as the ACBRT structure of Sahni et al. [22] are

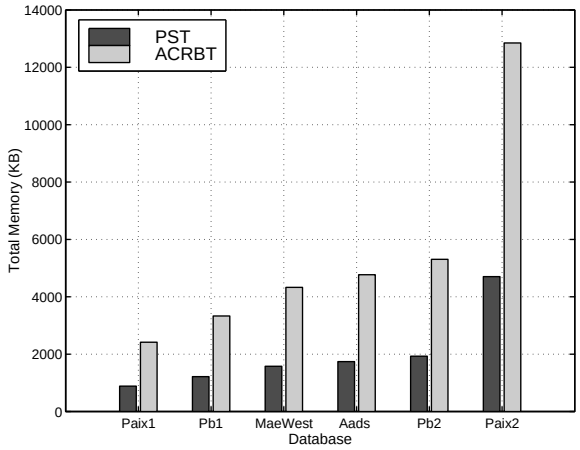


Figure 2–12: Memory usage

shown in Table 2–1. The datbases Paix1, Pb1, MaeWest and Aads were obtained on Nov 22, 2001, while Pb2 and Paix2 were obtained Sept. 13, 2000. Figure 2–12 is a plot of the data of Table 2–1. As can be seen, the ACBRT structure takes almost three times as much memory as is taken by the PST structure. Further, the memory requirement of the PST structure can be reduced to about 50% that of our current implementation. This reduction requires an n-node implementation of a priority-search tree as described in [37] rather than our current implementation, which uses $2n - 1$ nodes as in [39].

Table 2–1: Memory usage

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
	ACRBT	2417	3331	4327	4769	5305	12851

To obtain the mean time to find the longest matching-prefix (i.e., to perform a search), we started with a PST or ACRBT that contained all prefixes of a prefix database. Next, 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 each of these start points. The time required to determine all of these longest-matching prefixes was measured

and averaged over the number of start points (equal to the number of prefixes). The experiment was repeated 20 times and the mean and standard deviation of the 20 mean times computed. Table 2–2 gives the mean time required to find the longest matching-prefix on a Sun Blade 100 workstation that has a 500MHz UltraSPARC-Iie processor and has a 256KB L2 cache. The standard deviation in the mean time is also given in this table. On our Sun workstation, finding the longest matching-prefix takes about 10% to 14% less time using an ACRBT than a PST.

Table 2–2: Prefix times on a 500MHz Sun Blade 100 workstation

Database			Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Search (μsec)	PST	Mean	2.88	3.06	3.25	3.31	3.43	4.06
		Std	0.36	0.18	0.17	0.16	0.09	0.05
	ACRBT	Mean	2.60	2.77	2.87	2.87	3.09	3.51
		Std	0.25	0.16	0.16	0.12	0.13	0.04
Insert (μsec)	PST	Mean	3.90	4.45	4.83	5.18	5.14	6.04
		Std	0.57	0.63	0.51	0.48	0.19	0.20
	ACRBT	Mean	21.15	23.42	24.77	25.36	25.54	28.07
		Std	1.11	0.66	0.38	0.29	0.19	0.18
Delete (μsec)	PST	Mean	4.36	4.45	4.73	4.71	5.06	5.48
		Std	0.91	0.63	0.53	0.00	0.19	0.16
	ACRBT	Mean	21.24	22.68	23.16	23.71	24.56	25.64
		Std	0.95	0.55	0.49	0.35	0.26	0.21

To obtain the mean time to insert 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. This experiment was repeated 20 times and the mean of the mean as well as the standard deviation in the mean computed. These latter two quantities are given in Table 2–2 for our Sun workstation. As can be seen, insertions into a PST take between 18% and 22% the time to insert into an ACRBT!

The mean and standard deviation data reported in Table 2–2 for the delete operation were obtained in a similar fashion by starting with a data structure that

had 100% of the prefixes in the database and measuring the time to delete a randomly selected 33% of these prefixes. Deletion from a PST takes about 20% the time required to delete from an ACRBT.

Tables 2–3 and 2–4 give the corresponding times on a 700MHz Pentium III PC and a 1.4GHz Pentium 4 PC, respectively. Both computers have a 256KB L2 cache. The run times on our 700MHz Pentium III are about one-half the times on our Sun workstation. Surprisingly, when going from the 700MHz Pentium III to the 1.4GHz Pentium 4, the measured time to find the longest matching-prefix decreased by only about 5% for PST. More surprisingly, the corresponding times for ACRBT actually increased. The net result of the slight decrease in time for PST and the increase for ACRBT is that, on our Pentium 4 PC, the PST is faster than the ACRBT on all three operations–find longest matching-prefix, insert, and delete. This somewhat surprising behavior is due to architectural differences (e.g., differences in width and size of L1 cache lines) between the Pentium III and 4 processors.

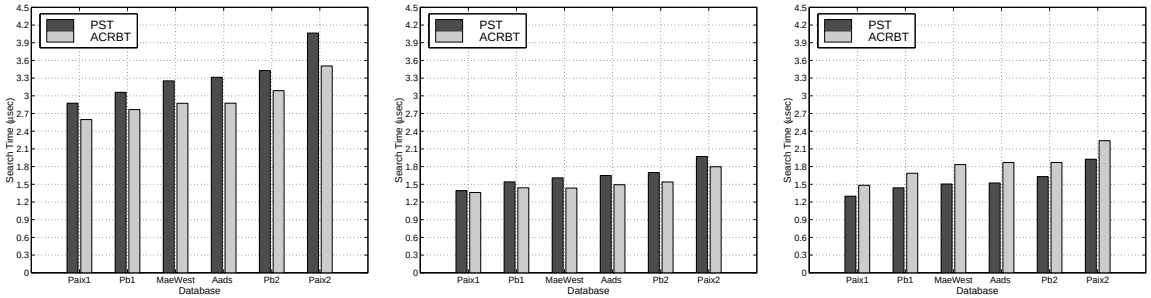
Table 2–3: Prefix times on a 700MHz Pentium III PC

Database			Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Search (μsec)	PST	Mean	1.39	1.54	1.61	1.65	1.70	1.97
		Std	0.27	0.22	0.17	0.14	0.00	0.04
	ACRBT	Mean	1.36	1.44	1.44	1.49	1.54	1.80
		Std	0.25	0.18	0.13	0.14	0.14	0.06
Insert (μsec)	PST	Mean	2.41	2.63	2.60	2.83	2.80	3.07
		Std	0.87	0.30	0.53	0.43	0.40	0.14
	ACRBT	Mean	11.97	12.63	13.48	13.62	13.77	14.93
		Std	0.95	0.67	0.24	0.48	0.35	0.18
Delete (μsec)	PST	Mean	2.32	2.38	2.49	2.45	2.55	2.91
		Std	0.82	0.61	0.52	0.47	0.00	0.17
	ACRBT	Mean	11.69	12.55	12.95	13.01	13.40	14.10
		Std	0.87	0.63	0.54	0.44	0.48	0.16

Figures 2–13, 2–14, and 2–15 histogram the search, insert, and delete time data of the preceding tables.

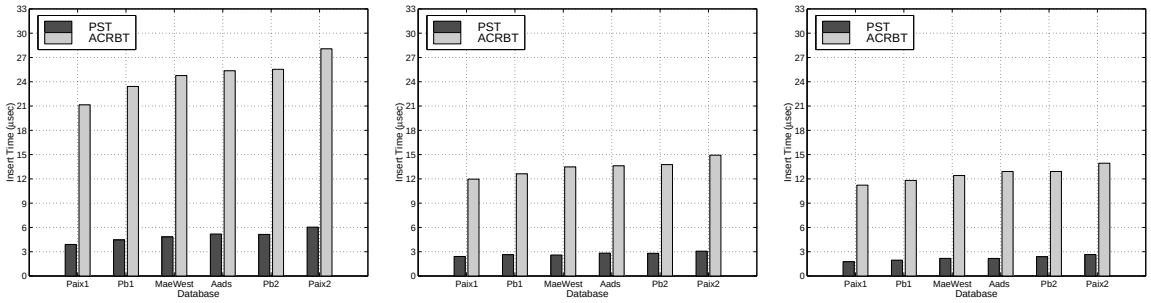
Table 2–4: Prefix times on a 1.4GHz Pentium 4 PC

Database			Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Search (μsec)	PST	Mean	1.30	1.44	1.51	1.52	1.63	1.92
		Std	0.19	0.18	0.17	0.13	0.13	0.06
	ACRBT	Mean	1.48	1.69	1.83	1.87	1.87	2.24
		Std	0.31	0.20	0.16	0.07	0.14	0.05
Insert (μsec)	PST	Mean	1.76	1.96	2.18	2.17	2.38	2.65
		Std	0.41	0.69	0.00	0.44	0.35	0.18
	ACRBT	Mean	11.22	11.81	12.41	12.91	12.92	13.94
		Std	0.41	0.60	0.41	0.44	0.26	0.18
Delete (μsec)	PST	Mean	1.76	1.69	1.92	1.93	2.00	2.22
		Std	0.41	0.60	0.38	0.21	0.42	0.17
	ACRBT	Mean	9.46	10.39	10.54	10.42	10.92	11.64
		Std	0.57	0.63	0.38	0.21	0.42	0.16



(A) (B) (C)

Figure 2–13: Time for searching longest matching prefix. A)Sun. B)Pentium 700MHz. C)Pentium 1.4GHz



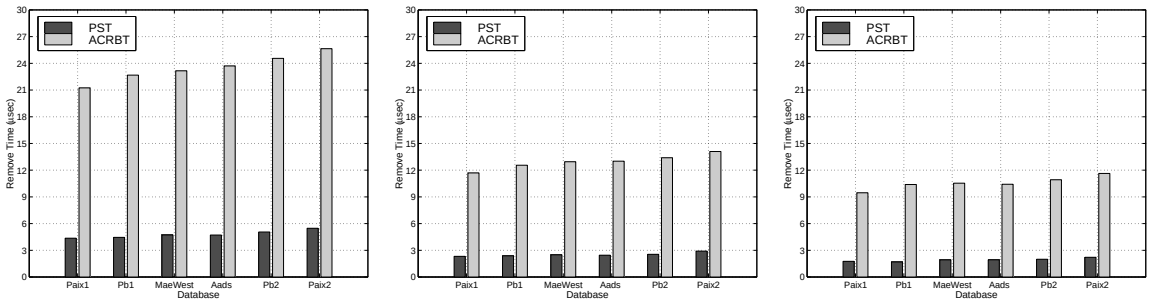
(A) (B) (C)

Figure 2–14: Time for inserting a prefix. A)Sun. B)Pentium 700MHz. C)Pentium 1.4GHz

2.5.2 Nonintersecting Ranges

To benchmark our algorithm for nonintersecting ranges (Section 2.3), we generated three different sets of random[1] nonintersecting ranges. These, respectively, had

[1] We resorted to randomly generated data sets because no benchmark data for nonintersecting ranges was available.



(A) (B) (C)

Figure 2–15: Time for deleting a prefix. A)Sun. B)Pentium 700MHz. C)Pentium 1.4GHz

30000, 50000, and 80000 ranges. Table 2–5 gives the memory requirement as well as the mean times and standard deviations for search, insert, and delete. The run times are for our 700MHz Pentium III PC. The search, insert, and delete experiments were modeled after those conducted for the case of prefix databases.

Table 2–5: Nonintersecting Ranges. 700 MHz PIII

Num of Ranges		30000	50000	80000
Memory Usage (KB)		3360	5600	8960
Search	Mean	1.92	2.19	2.51
(μsec)	Std	0.15	0.04	0.06
Insert	Mean	8.65	9.27	9.88
(μsec)	Std	0.49	0.29	0.17
Remove	Mean	5.75	6.42	6.81
(μsec)	Std	0.44	0.28	0.14

2.5.3 Conflict-free Ranges

Table 2–6 gives the memory required as well as the mean times and standard deviations for the case of conflict-free ranges. The range sequence used is generated so that when the ranges are inserted in sequence order, there are no conflicts. For deletion, 33% of the ranges are removed in the reverse of the insert order.

2.6 Conclusion

We have developed data structures for dynamic router tables. Our data structures permit one to search, insert, and delete in $O(\log n)$ time each. Although $O(\log n)$

Table 2–6: Conflict-free Ranges. PIII 700MHz with 256K L2 cache

Num of Ranges in R		30000	50000	80000
Num of Ranges in norm(R)	Mean	29688	48868	76472
	Std	18.03	42.90	60.05
Memory Usage (KB)	Mean	6240	9979	15219
	Std	7.06	10.91	11.19
Search (μsec)	Mean	1.98	2.34	2.69
	Std	0.07	0.09	0.06
Insert (μsec)	Mean	18.45	19.65	20.76
	Std	0.51	0.27	0.27
Remove (μsec)	Mean	19.3	20.49	21.60
	Std	0.41	0.13	0.29

time data structures for prefix tables were known prior to our work [21, 22], our data structure is more memory efficient than the data structures of Sahni et al. [21, 22]. Further, our data structure is significantly superior on the insert and delete operations, while being competitive on the search operation.

For nonintersecting ranges and conflict-free ranges our data structures are the first to permit $O(\log n)$ search, insert, and delete.

CHAPTER 3
DYNAMIC IP ROUTER TABLES USING HIGHEST-PRIORITY MATCHING

In this chapter, we focus on data structures for dynamic NHPRTs, HPPTs and LMPTs. In Section 3.2, we develop the data structure binary tree on binary tree (BOB). This data structure is proposed for the representation of dynamic NHPRTs. Using BOB, a lookup takes $O(\log^2 n)$ time and cache misses; a new rule may be inserted and an old one deleted in $O(\log n)$ time and cache misses. For HPPTs, we propose a modified version of BOB–PBOB (prefix BOB)–in Section 3.3. Using PBOB, a lookup, rule insertion and deletion each take $O(W)$ time and cache misses. In Section 3.4, we develop the data structures LMPBOB (longest matching-prefix BOB) for LMPTs. Using LMPBOB, the longest matching-prefix may be found in $O(W)$ time and $O(\log n)$ cache misses; rule insertion and deletion each take $O(\log n)$ time and cache misses. 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. Section 3.1 introduces some terminology and Experimental results are presented in Section 3.6.

3.1 Preliminaries

Definition 13 *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* $u \leq d \leq v$*.*

The start point of the range $r = [3, 9]$ is 3 and its finish point is 9. This range matches the addresses $\{3, 4, 5, 6, 7, 8, 9\}$. In IPv4, s and f are up to 32 bits long, and in IPv6, s and f may be up to 128 bits long. The IPv4 prefix $P = 0*$ corresponds to

the range $[0, 2^{31} - 1]$. The range [3,9] does not correspond to any single IPv4 prefix. We may draw the range $r = [u, v] = \{u, u + 1, ..., v\}$ as a horizontal line that begins at u and ends at v. Figure 2–1 shows ranges drawn in this fashion.

Notice that every prefix of a prefix router-table may be represented as a range. For example, when $W = 6$, the prefix $P = 1101*$ matches addresses in the range $[52, 55]$. So, we say $P = 1101* = [52, 55]$, $start(P) = 52$, and $finish(P) = 55$.

Since a range represents a set of (contiguous) points, we may use standard set operations and relations such as $\cap$ and $\subset$ when dealing with ranges. So, for example, $[2, 6] \cap [4, 8] = [4, 6]$. Note that some operations between ranges my not yield a range. For example, $[2, 6] \cup [8, 10] = \{2, 3, 4, 5, 6, 8, 9, 10\}$, which is not a range.

Definition 14 *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.

$$disjoint(r, s) \iff overlap(r, s) = \emptyset \iff v < x \vee y < u$$

Figure 2–1(A) shows the two cases for disjoint sets.

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

$$\begin{aligned} nested(r, s) &\iff overlap(r, s) = r \vee overlap(r, s) = s \\ &\iff r \subseteq s \vee s \subseteq r \\ &\iff x \leq u \leq v \leq y \vee u \leq x \leq y \leq v \end{aligned}$$

Figure 2–1(B) shows the two cases for nested sets.

(c) *The predicate* $intersect(r, s)$ *is true iff* r *and* s *have a nonempty intersection that is different from both* r *and* s.

$$\begin{aligned} intersect(r, s) &\iff r \cap s \neq \emptyset \wedge r \cap s \neq r \wedge r \cap s \neq s \\ &\iff \neg disjoint(r, s) \wedge \neg nested(r, s) \\ &\iff u < x \leq v < y \vee x < u \leq y < v \end{aligned}$$

Figure 2–1(C) shows the two cases for ranges that intersect. Notice that $overlap(r, s) = [x, v]$ *when* $u < x \leq v < y$ *and* $overlap(r, s) = [u, y]$ *when* $x < u \leq y < v$.

[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. $[4, 4]$ is the overlap of $[2, 4]$ and $[4, 6]$; and $overlap([3, 8], [2, 4]) = [3, 4]$.

Lemma 24 *Let* r *and* s *be two ranges. Exactly one of the following is true.*

1. $disjoint(r, s)$
2. $nested(r, s)$
3. $intersect(r, s)$

Proof Straightforward. ■

Definition 15 *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 16 *The range* r *is* **more specific** *than the range* s *iff* $r \subset s$.

$[2, 4]$ is more specific than $[1, 6]$, and $[5, 9]$ is more spcific than $[5, 12]$. Since $[2, 4]$ and $[8, 14]$ are disjoint, neither is more specific than the other. Also, since $[4, 14]$ and $[6, 20]$ intersect, neither is more specific than the other.

Definition 17 *Let* R *be a range set.* $ranges(d, R)$ *(or simply* $ranges(d)$ *when* R *is implicit) is the subset of ranges 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 in* $ranges(d)$. $msr([u, v], R) = msr(u, v, R) = r$ *iff* $msr(d, R) = r$,

$u \leq d \leq v$. When R is implicit, we write $msr(u, v)$ and $msr([u, v])$ in place of $msr(u, v, R)$ and $msr([u, v], R)$. $hpr(d)$ is the highest-priority range in $ranges(d)$. We assume that ranges are assigned priorities in such a way that $hpr(d)$ is uniquely defined for every d.

When $R = \{[2, 4], [1, 6]\}$, $ranges(3) = \{[2, 4], [1, 6]\}$, $msr(3) = [2, 4]$, $msr(1) = [1, 6]$, $msr(7) = \emptyset$, and $msr(5, 6) = [1, 6]$. When $R = \{[4, 14], [6, 20], [6, 14], [8, 12]\}$, $msr(4, 5) = [4, 14]$, $msr(6, 7) = [6, 14]$, $msr(8, 12) = [8, 12]$, $msr(13, 14) = [6, 14]$, and $msr(15, 20) = [6, 20]$.

Definition 18 *Let r and s be two ranges. $r < s \Leftrightarrow start(r) < start(s) \vee (start(r) = start(s) \wedge finish(r) > finish(s))$.*

Note that for every pair, r and s, of different ranges, either $r < s$ or $s < r$.

Lemma 25 *Let R be a nonintersecting range set. If $r \cap s \neq \emptyset$ for r , $s \in R$, then the following are 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. ■

3.2 Nonintersecting Highest-Priority Rule-Tables (NHRTs)—BOB

3.2.1 The Data Structure

The data structure binary tree on binary tree (BOB) that is being proposed here for NHRTs comprises a single balanced binary search tree at the top level. This top-level balanced binary search tree is called the point search tree (PTST). For an n-rule NHRT, the PTST has at most $2n$ nodes (we call this the PTST **size constraint**). The size constraint is necessary to enable $O(\log n)$ update. With each node z of the PTST, we associate a point, $point(z)$. The PTST is a standard red-black binary search tree (actually, any binary search tree structure that supports efficient search, insert, and delete may be used) on the $point(z)$ values of its node set [24]. That

is, for every node z of the PTST, nodes in the left subtree of z have smaller point values than $point(z)$, and nodes in the right subtree of z have larger point values than $point(z)$.

Let R be the set of nonintersecting ranges of the NHRT. Each range of R is stored in exactly one of the nodes of the PTST. More specifically, the root of the PTST stores all ranges $r \in R$ such that $start(r) \leq point(root) \leq finish(r)$; all ranges $r \in R$ such that $finish(r) < point(root)$ are stored in the left subtree of the root; all ranges $r \in R$ such that $point(root) < start(r)$ (i.e., the remaining ranges of R) are stored in the right subtree of the root. The ranges allocated to the left and right subtrees of the root are allocated to nodes in these subtrees using the just stated **range allocation rule** recursively. Note that the range allocation rule is quite similar to that used for interval trees [40].

For the range allocation rule to successfuly allocate all $r \in R$ to exactly one node of the PTST, the PTST must have at least one node z for which $start(r) \leq point(z) \leq finish(r)$. Table 3–1 gives an example set of nonintersecting ranges, and Figure 3–1 shows a possible PTST for this set of ranges (we say possible, because we haven't specified how to select the $point(z)$ values and even with specified $point(z)$ values, the corresponding red-black tree isn't unique). The number inside each node is $point(z)$, and outside each node, we give $ranges(z)$.

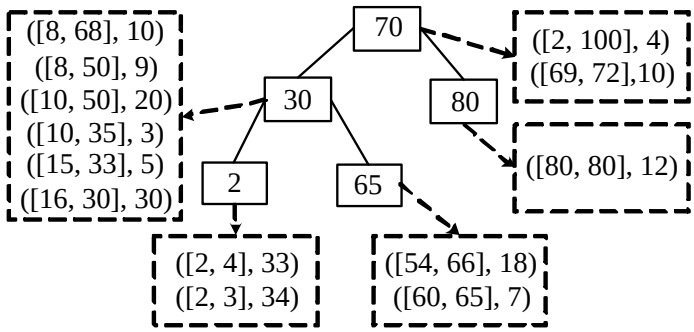


Figure 3–1: A possible PTST

Table 3–1: A nonintersecting range set

range	**priority**
[2, 100]	4
[2, 4]	33
[2, 3]	34
[8, 68]	10
[8, 50]	9
[10, 50]	20
[10, 35]	3
[15, 33]	5
[16, 30]	30
[54, 66]	18
[60, 65]	7
[69, 72]	10
[80, 80]	12

Let $ranges(z)$ be the subset of ranges of R allocated to node z of the PTST.[1] Since the PTST may have as many as $2n$ nodes and since each range of R is in exactly one of the sets $ranges(z)$, some of the $ranges(z)$ sets may be empty.

The ranges in $ranges(z)$ may be ordered using the $<$ relation of Definition 18. Using this $<$ relation, we put the ranges of $ranges(z)$ into a red-black tree (any balanced binary search tree structure that supports efficient search, insert, delete, join, and split may be used) called the range search-tree or $RST(z)$. Each node x of $RST(z)$ stores exactly one range of $ranges(z)$. We refer to this range as $range(x)$. Every node y in the left (right) subtree of node x of $RST(z)$ has $range(y) < range(x)$ ($range(y) > range(x)$). In addition, each node x stores the quantity $mp(x)$, which is the maximum of the priorities of the ranges associated with the nodes in the subtree

[1] We have overloaded the function $ranges$. When u is a node, $ranges(u)$ refers to the ranges stored in node u of a PTST; when u is a destination address, $ranges(u)$ refers to the ranges that match u

rooted at x. $mp(x)$ may be defined recursively as below.

$$mp(x) = \begin{cases} p(x) & \text{if } x \text{ is leaf} \\ \max\{mp(leftChild(x)), mp(rightChild(x)), p(x)\} & \text{otherwise} \end{cases}$$

where $p(x) = priority(range(x))$. Figure 3–2 gives a possible RST structure for $ranges(30)$ of Figure 3–1. Each node shows $(range(x), p(x), mp(x))$.

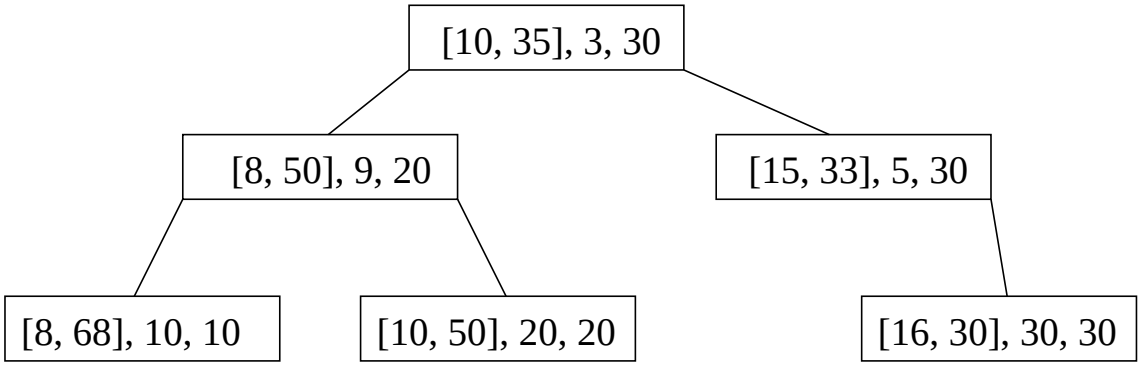


Figure 3–2: An example RST for $ranges(30)$ of Figure 3–1

Lemma 26 *Let z be a node in a PTST and let x be a node in $RST(z)$. Let $st(x) = start(range(x))$ and $fn(x) = finish(range(x))$.*

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 For 1, we see that when y is in the right subtree of x, $range(y) > range(x)$. From Definition 18, it follows that $st(y) \geq st(x)$. Further, since $range(y) \cap range(x) \neq \emptyset$, if $st(y) > st(x)$, then $fn(y) \leq fn(x)$ (Lemma 25); if $st(y) = st(x)$, $fn(y) < fn(x)$ (Definition 18). The proof for 2 is similar. ■

3.2.2 Search for $hpr(d)$

The highest-priority range that matches the destination address d may be found by following a path from the root of the PTST toward a leaf of the PTST. Figure 3–3 gives the algorithm. For simplicity, this algorithm finds $hp = priority(hpr(d))$ rather than $hpr(d)$. The algorithm is easily modified to return $hpr(d)$ instead.

```
Algorithm hp(d) {
   // return the priority of hpr(d)
   // easily extended to return hpr(d)
    hp = -1; // assuming 0 is the smallest priority value
    z  = root; // root of PTST
    while (z != null) {
        if (d > point(z)) {
            RST(z)->hpRight(d, hp);
            z = rightChild(z);
        }
        else if (d < point(z)) {
            RST(z)->hpLeft(d, hp);
            z = leftChild(z);
        }
        else // d == point(z)
             return max{hp, mp(RST(z)->root)};
    }
    return hp;
}
```

Figure 3–3: Algorithm to find $priority(hpr(d))$

We begin by initializing $hp = -1$ and z is set to the root of the PTST. This initialization assumes that all priorities are ≥ 0. The variable z is used to follow a path from the root toward a leaf. When $d > point(z)$, d may be matched only by ranges in $RST(z)$ and those in the right subtree of z. The method `RST(z)->hpRight(d,hp)` (Figure 3–4) updates hp to reflect any matching ranges in $RST(z)$. This method makes use of the fact that $d > point(z)$. Consider a node x of $RST(z)$. If $d > fn(x)$, then d is to the right (i.e., $d > finish(range(x))$) of $range(x)$ and also to the right of all ranges in the right subtree of x. Hence, we may proceed to examine the ranges in the left subtree of x. When $d \leq fn(x)$, $range(x)$ as well as all ranges in the left subtree of x match d. Additional matching ranges may be present in the right subtree of x. $hpLeft(d, hp)$ is the analogous method for the case when $d < point(z)$.

Complexity The complexity of the invocation `RST(z)->hpRight(d,hp)` is readily seen to be $O(height(RST(z)) = O(\log n)$. Consequently, the complexity of $hp(d)$ is $O(\log^2 n)$. To determine $hpr(d)$ we need only add code to the methods $hp(d)$,

```
Algorithm hpRight(d, hp) {
   // update hp to account for any ranges in RST(z) that match d
   // d > point(z)
    x  = root; // root of RST(z)
    while (x != null)
        if (d > fn(x))
            x = leftChild(x);
        else {
              hp = max{hp, p(x), mp(leftChild(x))};
              x = rightChild(x);
        }
}
```

Figure 3–4: Algorithm $hpRight(d, hp)$

$hpRight(d, hp)$, and $hpLeft(d, hp)$ so as to keep track of the range whose priority is the current value of hp. So, $hpr(d)$ may be found in $O(\log^2 n)$ time also.

3.2.3 Insert a Range

A range r that is known to have no intersection with any of the existing ranges in the router table, may be inserted using the algorithm of Figure 3–5. In the `while` loop, we find the node z nearest to the root such that r matches $point(z)$ (i.e., $start(r) \leq point(z) \leq finish(r)$). If such a z exists, the range r is inserted into $RST(z)$ using the standard red-black insertion algorithm [24]. During this insertion, it is necessary to update some of the mp values on the insert path. This update is done easily. In case the PTST has no z such that r matches $point(z)$, we insert a new node into the PTST. This insertion is done using the method $insertNewNode$.

To insert a new node into the PTST, we first create a new PTST node y and define $point(y)$ and $RST(y)$. $point(y)$ may be set to be any destination address matched by r (i.e., any address such that $start(r) \leq point(y) \leq finish(r)$) may be used. In our implementation, we use $point(y) = start(r)$. $RST(y)$ has only a root node and this root contains r; its mp value is $priority(r)$. If the PTST is currently empty, y becomes the new root and we are done. Otherwise, the new node y may be inserted where the search conducted in the `while` loop of Figure 3–5 terminated. That

```
Algorithm insert(r) {
   // insert the nonintersecting range r
    z  = root; // root of PTST
    while (z != null)
        if (finish(r) < point(z))
            z = leftChild(z);
        else if (start(r) > point(z))
            z = rightChild(z);
        else {// r matches point(z)
              RST(z)->insert(r);
              return;
        }

    // there is no node z such that r matches point(z)
    // insert a new node into PTST
    insertNewNode(r);
}
```

Figure 3–5: Algorithm to insert a nonintersecting range

is, as a child of the last non-null value of z. Following this insertion, the traditional bottom-up red-black rebalancing pass is made [24]. This rebalancing pass may require color changes and at most one rotation. Color changes do not affect the tree structure. However, a rebalancing rotation, if performed, affects the tree structure and may lead to a violation of the range allocation rule. Rebalancing rotations are investigated in the next section.

We note that if the number of nodes in the PTST was at most $2|R|$, where $|R|$ is the number of ranges prior to the insertion of a new range r, then following the insertion, $|PTST| \leq 2|R| + 1 < 2(|R| + 1)$, where $|PTST|$ is the number of nodes in the PTST and $|R| + 1$ is the number of ranges following the inertion of r. Hence an insert does not violate the PTST size constraint.

Complexity Exclusive of the time required to perform the tasks associated with a rebalancing rotation, the time required to insert a range is $O(height(PTST)) = O(\log n)$. As we shall see in the next section, a rebalancing rotation can be done in $O(\log n)$ time. Since at most one rebalancing rotation is needed following an insert,

the time to insert a range is $O(\log n)$. In case it is necessary for us to verify that the range to be inserted does not intersect an existing range, we can augment the PTST with priority search trees as in [34] and use these trees for intersection detection. The overall complexity of an insert remains $O(\log n)$.

3.2.4 Red-Black-Tree Rotations

Figures 3–6 and 3–7, respectively, show the red-black LL and RR rotations used to rebalance a red-black tree following an insert or delete (see [24]). In these figures, $pt()$ is an abbreviation for $point()$. Since the remaining rotation types, LR and RL, may, respectively, be viewed as an RR rotation followed by an LL rotation and an LL rotation followed by an RR rotation, it suffices to examine LL and RR rotations alone.

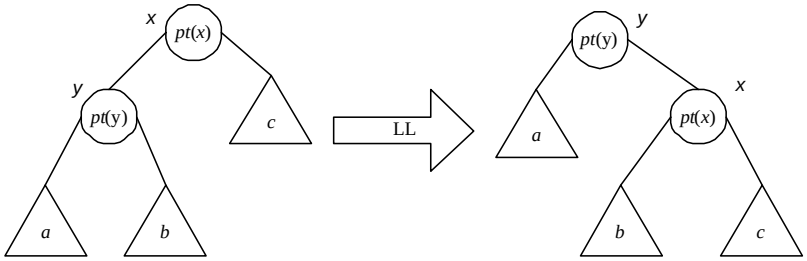


Figure 3–6: LL rotation

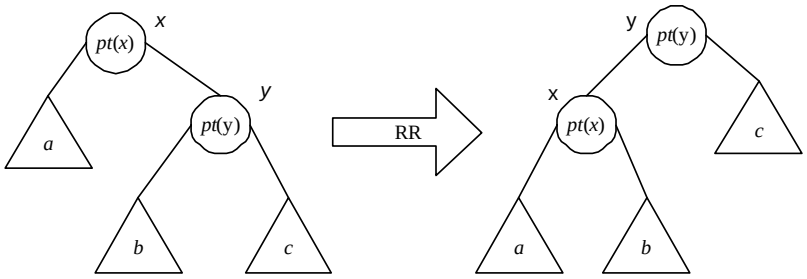


Figure 3–7: RR rotation

Lemma 27 *Let R be a set of nonintersecting ranges. Let $ranges(z) \subseteq R$ be the ranges allocated by the range allocation rule to node z of the PTST prior to an LL or RR rotation. Let $ranges'(z)$ be this subset for the PTST node z following the rotation.*

$ranges(z) = ranges'(z)$ for all nodes z in the subtrees a, b, and c of Figures 3–6 and 3–7.

Proof Consider an LL rotation. Let $ranges(subtree(x))$ be the union of the ranges allocated to the nodes in the subtree whose root is x. Since the range allocation rule allocates each range r to the node z nearest the root such that r matches $point(z)$, $ranges(subtree(x)) = ranges'(subtree(y))$. Further, $r \in ranges(a)$ iff $r \in ranges(subtree(x))$ and $finish(r) < point(y)$. Consequently, $r \in ranges'(a)$. From this and the fact that the LL rotation doesn't change the positioning of nodes in a, it follows that for every node z in the subtree a, $ranges(a) = ranges'(a)$. The proof for the nodes in b and c as well as for the RR rotation is similar. ■

Let x and y be as in Figures 3–6 and 3–7. From Lemma 27, it follows that $ranges(z) = ranges'(z)$ for all z in the PTST except possibly for $z \in \{x, y\}$. It is not too difficult to see that $ranges'(y) = ranges(y) \cup S$ and $ranges'(x) = ranges(x) - S$, where

$$S = \{r | r \in ranges(x) \wedge start(r) \leq point(y) \leq finish(r)\}$$

Since we are dealing with a set of nonintersecting ranges, all ranges in $ranges(y)$ are nested within the ranges of S. Figure 3–8 shows the ranges of $ranges(x)$ using solid lines and those of $ranges(y)$ using broken lines. S is the set of ranges drawn above $ranges(y)$ (i.e., the solid lines above the broken lines).

The range $rMax$ of S with largest $start()$ value may be found by searching $RST(x)$ for the range with largest $start()$ value that matches $point(y)$. (Note that $rMax = msr(point(y), ranges(x))$.) Since $RST(x)$ is a binary search tree of an ordered set of ranges (Definition 18), $rMax$ may be found in $O(height(RST(x))$ time by following a path from the root downward. If $rMax$ doesn't exist, $S = \emptyset$, $ranges'(x) = ranges(x)$ and $ranges'(y) = ranges(y)$.

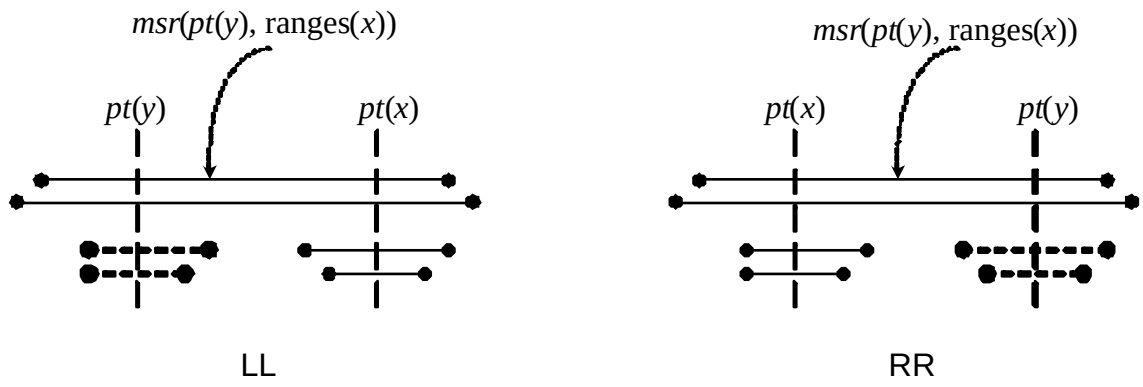


Figure 3–8: $ranges(x)$ and $ranges(y)$ for LL and RR rotations. Nodes x and y are as in Figures 3–6 and 3–7

Assume that $rMax$ exists. We may use the *split* operation [24] to extract from $RST(x)$ the ranges that belong to S. The operation

$$RST(x) \rightarrow split(small, rMax, big)$$

separates $RST(x)$ into an RST *small* of ranges $<$ (Definition 18) than $rMax$ and an RST *big* of ranges $>$ than $rMax$. We see that $RST'(x) = big$ and $RST'(y) = join(small, rMax, RST(y))$, where *join* [24] combines the red-black tree *small* with ranges $< rMax$, the range $rMax$, and the red-black tree $RST(y)$ with ranges $> rMax$ into a single red-black tree.

The standard *split* and *join* operations of Horowitz et al. [24] need to be modified slightly so as to update the *mp* values of affected nodes. This modification doesn't affect the asymptotic complexity, which is logarithmic in the number of nodes in the tree being split or logarithmic in the sum of the number of nodes in the two trees being joined, of the *split* and *join* operations. So, the complexity of performing an LL or RR rotation (and hence of performing an LR or RL rotation) in the PTST is $O(\log n)$.

3.2.5 Delete a Range

Figure 3–9 gives our algorithm to delete a range r. Note that if r is one of the ranges in the PTST, then r must be in the RST of the node z that is closest to the root and such that r matches $point(z)$. The `while` loop of Figure 3–9 finds this z and deletes r from $RST(z)$.

```
Algorithm delete(r) {
   // delete the range r
    z  = root; // root of PTST
    while (z != null)
        if (finish(r) < point(z))
            z = leftChild(z);
        else if (start(r) > point(z))
            z = rightChild(z);
        else {// r matches point(z)
              RST(z)->delete(r);
              cleanup(z);
              return;
        }
}
```

Figure 3–9: Algorithm to delete a range

Assume that r is, in fact, one of the ranges in our PTST. To delete r from $RST(z)$, we use the standard red-black deletion algorithm [24] modified to update mp values as necessary. Following the deletion of r from $RST(z)$ we perform a cleanup operation that is necessary to maintain the size constraint of the PTST. Figure 3–10 gives the steps in the method `cleanup`.

```
Algorithm cleanup(z) {
   // maintain size constraint
   if (RST(z) is empty and the degree of z is 0 or 1)
      delete node z from the PTST and rebalance;

   while (|PTST| > 2|R|)
      delete a degree 0 or degree 1 node z with empty
      RST(z) from the PTST and rebalance;
}
```

Figure 3–10: Algorithm to maintain size constraint following a delete

Notice that following the deletion of r from $RST(z)$, $RST(z)$ may or may not be empty. If $RST(z)$ becomes empty and the degree of node z is either 0 or 1, node z is deleted from the PTST using the standard red-black node deletion algorithm [24]. If this deletion requires a rotation (at most one rotation may be required) the rotation is done as described in Section 3.2.4. Since the number of ranges and nodes has each decreased by 1, the size constraint may be violated (this happens if $|PTST| = 2|R|$ prior to the delete). Hence, it may be necessary to remove a node from the PTST to restore the size constraint.

If $RST(z)$ becomes empty and the degree of z is 2 or if $RST(z)$ does not become empty, z is not deleted from the PTST. Now, $|PTST|$ is unchanged by the deletion of r and $|R|$ reduces by 1. Again, it is possible that we have a size constraint violation. If so, up to two nodes may have to be removed from the PTST to restore the size constraint.

The size constraint, if violated, is restored in the `while` loop of Figure 3–10. This restoration is done by removing one or two (as needed) degree 0 or degree 1 nodes that have an empty RST. Lemma 28 shows that whenever the size constraint is violated, the PTST has at least one degree 0 or degree 1 node with an empty RST. So, the node z needed for deletion in each iteration of the `while` loop always exists.

Lemma 28 *When the PTST has $> 2n$ nodes, where $n = |R|$, the PTST has at least one degree 0 or degree 1 node that has an empty PTST.*

Proof Suppose not. Then the degree of every node that has an empty RST is 2. Let n_2 be the total number of degree 2 nodes, n_1 the total number of degree 1 nodes, n_0 the total number of degree 0 nodes, n_e the total number of nodes that have an empty RST, and n_n the total number of nodes that have a nonempty RST. Since all PTST nodes that have an empty RST are degree 2 nodes, $n_2 \geq n_e$. Further, since there are only n ranges and each range is stored in exactly one RST, there are at most n nodes that have a nonempty RST, i.e., $n \geq n_n$. Thus $n_2 + n \geq n_e + n_n =$

$|PTST|$, i.e., $n_2 \geq |PTST| - n$. From [24], we know that $n_0 = n_2 + 1$. Hence, $n_0 + n_1 + n_2 = n_2 + 1 + n_1 + n_2 > n_2 + n_2 \geq 2|PTST| - 2n > |PTST|$. This contradicts $n_0 + n_1 + n_2 = |PTST|$. ■

To find the degree 0 and degree 1 nodes that have an empty RST efficiently, we maintain a doubly-linked list of these nodes. Also, a doubly-linked list of degree 2 nodes that have an empty RST is maintained. When a range is inserted or deleted, PTST nodes may be added/removed from these doubly-linked lists and nodes may move from one list to another. The required operations can be done in $O(1)$ time each.

Complexity It takes $O(\log n)$ time to find the PTST node z that contains the range r that is to be deleted. Another $O(\log n)$ time is needed to delete r from $RST(z)$. The cleanup step removes up to 2 nodes from the PTST. This takes another $O(\log n)$ time. So, the overall delete time is $O(\log n)$.

3.2.6 Expected Complexity of BOB

Let $maxR$ be the maximum number of ranges that match any destination address. So, $|ranges(z)| = |RST(z)| \leq maxR$ for every node z of the PTST. We may, therefore, restate the complexity of the BOB operations–lookup, insert, delete–as $O(\log n \log maxR)$, $O(\log n)$, and $O(\log n)$, respectively.

Sahni et al. [21] have analyzed the prefixes in several real IPv4 prefix router-tables. They report that a destination address is matched by about 1 prefix on average; the maximum number of prefixes that match a destination address is at most 6. Making the assumption that this analysis holds true even for real range router-tables (no data is available for us to perform such an analysis), we conclude that $maxR \leq 6$. So, the expected complexity of BOB on real router-tables is $O(\log n)$ per operation.

3.3 Highest-Priority Prefix-Tables (HPPTs)—PBOB

3.3.1 The Data Structure

When all rule filters are prefixes, $maxR \leq \min\{n, W\}$. Hence, if BOB is used to represent an HPPT, the search complexity is $O(\log n * \min\{\log n, \log W\})$; the insert and delete complexities are $O(\log n)$ each.

Since $maxR \leq 6$ for real prefix router-tables, we may expect to see better performance using a simpler structure (i.e., a structure with smaller overhead and possibly worse asymptotic complexity) for $ranges(z)$ than the RST structure described in Section 3.2. In PBOB, we replace the RST in each node, z, of the BOB PTST with an array linear list [41], $ALL(z)$, of pairs of the form $(pLength, priority)$, where $pLength$ is a prefix length (i.e., number of bits) and $priority$ is the prefix priority. $ALL(z)$ has one pair for each range $r \in ranges(z)$. The $pLength$ value of this pair is the length of the prefix that corresponds to the range r and the $priority$ value is the priority of the range r. The pairs in $ALL(z)$ are in ascending order of $pLength$. Note that since the ranges in $ranges(z)$ are nested and match $point(z)$, the corresponding prefixes have different length.

3.3.2 Lookup

Figure 3–11 gives the algoritm to find the priority of the highest-priority prefix that matches the destination address d. The method `maxp()` returns the highest priority of any prefix in $ALL(z)$ (note that all prefxes in $ALL(z)$ match $point(z)$). The method `searchALL(d,hp)` examines the prefixes in $ALL(z)$ and updates hp taking into account the priorities of those prefixes in $ALL(z)$ that match d.

The method `searchALL(d,hp)` utilizes the following lemma. Consequently, it examines prefixes of $ALL(z)$ in increasing order of length until either all prefixes have been examined or until the first (i.e., shortest) prefix that doesn't match d is examined.

```
Algorithm hp(d) {
   // return the priority of hpp(d)
   // easily extended to return hpp(d)
    hp = -1; // assuming 0 is the smallest priority value
    z  = root; // root of PTST
    while (z != null) {
        if (d == point(z))
            return max{hp, ALL(z)->maxp()};
        ALL(z)->searchALL(d,hp);
        if (d < point(z))
            z = leftChild(z);
        else
            z = rightChild(z);
    }
    return hp;
}
```

Figure 3–11: Algorithm to find $priority(hpp(d))$

Lemma 29 *If a prefix in $ALL(z)$ doesn't match a destination address d, then no longer-length prefix in $ALL(z)$ matches d.*

Proof Let p_1 and p_2 be prefixes in $ALL(z)$. Let l_i be the length of p_i. Assume that $l_1 < l_2$ and that p_1 doesn't match d. Since both p_1 and p_2 match $point(z)$, p_2 is nested within p_1. Therefore, all destination addresses that are matched by p_2 are also matched by p_1. So, p_2 doesn't match d. ■

One way to determine whether a length l prefix of $ALL(z)$ matches d is to use the following lemma. The check of this lemma may be implemented using a mask to extract the most-signifcant bits of $point(z)$ and d.

Lemma 30 *A length l prefix p of $ALL(z)$ matches d iff the most-significant l bits of $point(z)$ and d are the same.*

Proof Straightforward. ■

Complexity We assume that the masking operations can be done in $O(1)$ time each. (In IPv4, for example, each mask is 32 bits long and we may extract any subset of bits from a 32-bit integer by taking the logical and of the appropriate mask and

the integer.) The number of PTST nodes reached in the `while` loop of Figure 3–11 is $O(\log n)$ and the time spent at each node z that is reached is linear in the number of prefixes in $ALL(z)$ that match d. Since the PTST has at most $maxR$ prefixes that match d, the complexity of our lookup algorithm is $O(\log n + maxR) = O(W)$ (note that $\log_2 n \leq W$ and $maxR \leq W$).

3.3.3 Insertion and Deletion

The PBOB algorithms to insert/delete a prefix are simple adaptations of the corresponding algorithms for BOB. $rMax$ is found by examining the prefixes in $ALL(x)$ in increasing order of length. $ALL'(y)$ is obtained by prepending the prefixes in $ALL(x)$ whose length is $\leq$ the length of $rMax$ to $ALL(y)$, and $ALL'(x)$ is obtained from $ALL(x)$ by removeing the prefixes whose lenth is $\leq$ the length of $rMax$. The time require to find $rMax$ is $O(maxR)$. This is also the time required to compute $ALL'(y)$ and $ALL'(x)$. The overall complexity of an insert/delete operation is $O(\log n + maxR) = O(W)$.

As noted earlier, $maxR \leq 6$ in practice. So, in practice, PBOB takes $O(\log n)$ time and makes $O(\log n)$ cache misses per operation.

3.4 Longest-Matching Prefix-Tables (LMPTs)—LMPBOB

3.4.1 The Data Structure

Using $priority = pLength$, a PBOB may be used to represent an LMPT obtaining the same performance as for an HPPT. However, we may achieve some reduction in the memory required by the data structure if we replace the array linear list that is stored in each node of the PTST by a W-bit vector, bit. $bit(z)[i]$ denotes the ith bit of the bit vector stored in node z of the PTST, $bit(z)[i] = 1$ iff $ALL(z)$ has a prefix whose length is i. We note that Suri et al. [20] use W-bit vectors to keep track of prefix lengths in their data structure also.

3.4.2 Lookup

Figure 3–12 gives the algorithm to find the length of the longest matching-prefix, $lmp(d)$, for destination d. The method `longest()` returns the largest i such that $bit(z)[i] = 1$ (i.e., it returns the length of the longest prefix stored in node z). The method `searchBitVector(d,hp,k)` examines $bit(z)$ and updates hp taking into account the lengths of those prefixes in this bit vector that match d. The method `same(k+1, point(z), d)` returns `true` iff $point(z)$ and d agree on their $k+1$ most significant bits.

```
Algorithm lmp(d) {
   // return the length of lmp(d)
   // easily extended to return lmp(d)
    hp = 0; // length of lmp
    k = 0;  // next bit position to examine is k+1
    z  = root; // root of PTST
    while (z != null) {
        if (d == point(z))
            return max{k, z->longest()};
        bit(z)->searchBitVector(d,hp,k);
        if (d < point(z))
            z = leftChild(z);
        else
            z = rightChild(z);
    }
    return hp;
}
```

Figure 3–12: Algorithm to find $length(lmp(d))$

```
Algorithm searchBitVector(d,hp,k) {
   // update hp and k
    while (k < W && same(k+1, point(z), d) {
        if (bit(z)[k+1] == 1)
           hp = k+1;
        k++;
    }
}
```

Figure 3–13: Algorithm to search a bit vector for prefixes that match d

The method `searchBitVector(d,hp,k)` (Figure 3–13) utilizes the next two lemmas.

Lemma 31 *If $bit(z)[i]$ corresponds to a prefix that doesn't match the destination address d, then $bit(z)[j]$, $j > i$ corresponds to a prefix that doesn't match d.*

Proof $bit(z)[q]$ corresponds to the prefix p_q whose length is q and which equals the q most significant bits of $point(z)$. So, p_i matches all points that are matched by p_j. Hence, if p_i doesn't match d, p_j doesn't match d either. ■

Lemma 32 *Let w and z be two nodes in a PTST such that w is a descendent of z. Suppose that $z->bit(q)$ corresponds to a prefix p_q that matches d. $w->bit(j)$, $j \leq q$ cannot correspond to a prefix that matches d.*

Proof Suppose that $w->bit(j)$ corresponds to the prefix p_j, p_j matches d, and $j \leq q$. So, p_j equals the j most significant bits of d. Since p_q matches d and also $point(z)$, d and $point(z)$ have the same q most significant bits. Therefore, p_j matches $point(z)$. So, by the range allocation rule, p_j should be stored in node z and not in node w, a contradiction. ■

Complexity We assume that the method `same` can be implemented using masks and Boolean operations so as to have complexity $O(1)$. Sine a bit vector has the same number of bits as does a destination address, this assumption is consistent with the implicit assumption that arithmetic on destination addresses takes $O(1)$ time. The total time spent in all invocations of `searchBitVector` is $O(W + \log n)$. The time spent in the remaining steps of `lmp(d)` is $O(\log n)$. So, the overall complexity of `lmp(d)` is $O(W + \log n) = O(W)$. Even though the time complexity is $O(W)$, the number of cache misses is $O(\log n)$ (note that each bit vector takes the same amount of space as needed to store a destination address).

3.4.3 Insertion and Deletion

The insert and delete algorithms are similar to the corresponding algorithms for HPPTs. The essential difference are as below.

1. Rather than insert or delete a prefix from an $ALL(z)$, we set $bit(z)[l]$, where l is the length of the prefix being inserted or deleted, to 1 or 0, respectively.
2. For a rotation, we do not look for $rMax$ in $bit(x)$. Instead, we find the largest integer $iMax$ such that the prefix that corresponds to $bit(x)[iMax]$ matches $point(y)$. The first (bit 0 comes before bit 1) $iMax$ bits of $bit'(y)$ are the first $iMax$ bits of $bit(x)$ and the remaining bits of $bit'(y)$ are the same as the corresponding bits of $bit(y)$. $bit'(x)$ is obtained from $bit(x)$ by setting its first $iMax$ bits to 0.

Complexity $iMax$ may be determined in $O(\log W)$ time using binary search; $bit'(x)$ and $bit'(y)$ may be computed in $O(1)$ time using masks and boolean operations. The remaining tasks performed during an insert or delete take $O(\log n)$ time. So, the overall complexity of an insert or delete operation is $O(\log n + \log W) = O(\log(Wn))$. The number of cache misses is $O(\log n)$.

3.5 Implementation Details and Memory Requirement

3.5.1 Memory Management

We implemented our data structures in C++. Since dynamic memory allocation and deallocation using C++'s methods `new` and `delete` are very time consuming, we implemented our own methods to manage memory. We maintained our own list of free memory. Whenever this list was exhausted, we used the `new` method to get a large chunk of memory to add to our free list. Memory was then allocated from this large chunk as needed by our data structures. Whenever memory was to be deallocated, it was put back on to our free list.

3.5.2 BOB

As described in Section 3.2, each node z of the PTST of BOB has the following fields: $color$, $point(z)$, RST, $leftChild$, and $rightChild$. To improve the lookup performance of BOB, we added the following fields: $maxPriority$ (maximum priority of

the ranges in $ranges(z)$), $minSt$ (smallest starting point of the ranges in $ranges(z)$), and $maxFn$ (largest finish point of the ranges in $ranges(z)$). Correspondingly, the statements `RST->hpRight(d,h)` and `RST(z)->hpLeft(d,h)` of Figure 3–3 are excuted only when $maxPriority > hp\&\&d <= maxFn$ and $maxPriority > hp\&\&minSt \leq d$, respectively.

With the added fields, each node of the PTST has 8 fields. For the *color* and $maxPriority$ fields, we allocate 1 byte each. Assuming 4 bytes for each of the remaining fields, we get a node size of 26 bytes. For improved cache performance, it is desirable to align node to 4-byte memory-boundaries. This alignment is simplified if node size is an integral multiple of 4 bytes. Therefore, for practical purposes, the PTST node-size becomes 28 bytes.

In our implementation of `hpRight` (Figure 3–4), the `while` loop conditonal was changed from `x != null` to `x != null && mp > hp`. A corresponding change was made to `hpLeft`.

The nodes of an RST have the following fields: *color*, *mp*, *st*, fn, *p*, $leftChild$, and $rightChild$. Using 1 byte for the *color*, *p*, and *mp* fields each, and 4 bytes for each of the remaining fields, the size of an RST node becomes 19 bytes. Again, for ease of alignment to 4-byte boundaries, we make the RST-node size 20 bytes. In addition to nodes, every nonempty RST has the fields *root* (pointer to root of RST) and *rank* (rank of red-black tree) field. Each of these fields is a 4-byte field.

For the doubly-linked lists of PTST nodes with an empty RST, we used the $minSt$ and $maxFn$ fields to, respectively, represent left and right pointers. So, there is no space overhead (other than the space needed to keep track of the first node) associated with maintaining the two doubly-linked lists of PTST nodes that have an empty RST.

Since an instance of BOB may have up to $2n$ PTST nodes, n nonempty RSTs, and n RST nodes, the maximum space/memory required by BOB is $28*2n+8*n+20*n = 84n$ bytes.

3.5.3 PBOB

The required fields in each node z of the PTST of PBOB are: $color$, $point(z)$, ALL, $size$, $length$, $leftChild$, and $rightChild$, where ALL is a one-dimensional array, each entry of which has the subfields $pLength$ and $priority$; $size$ is the dimension of the array, and $length$ is the number of pairs currently in the array linear list. The array ALL initially has enough space to accomodate 4 pairs $(pLength, priority)$. When the capacity of an ALL is exceeded, the size of the ALL is increased by 4 pairs (since at most 6 pairs are expected in an ALL, the size of an ALL needs to be increased at most once; in theory, an ALL may get as many as W pairs and, in theory, using array doubling as in [41] may work better than increasing the array size by 4 each time array capacity is exceeded).

To improve the lookup performance of PBOB, the field $maxPriority$ (maximum priority of the prefixes in $ALL(z)$), may be added. Note that $minSt$ (smallest starting point of the prefixes in $ALL(z)$), and $maxFn$ (largest finish point of the prefixes in $ALL(z)$) are easily computed from $point(z)$ and the $pLength$ of the shortest (i.e., first) prefix in $ALL(z)$. When the nodes of the PTST are augmented with a $maxPriority$ field, the expression `ALL(z)->maxp()` in Figure 3–11 may be changed to `maxPriority(z)`, and the statement `ALL(z)->searchALL(d,hp)` executed only when

$$maxPriority > hp \ \&\& \ minSt \leq d \ \&\& \ d \leq maxFn$$

Since $searchALL$ does its first check against the shortest prefix in the array linear-list and this check tests $minSt \leq d \&\& d \leq maxFn$, it is sufficient to execute the statement `ALL(z)->searchALL(d,hp)` only when $maxPriority > hp$.

Using 1 byte for each of the fields: *color*, *size*, *length*, *maxPriority*, *pLength*, and *priority*; and 4 bytes for each of the remaining fields, the initial size of a PTST node of PBOB is 24 bytes.

For the doubly-linked lists of PTST nodes with an empty *ALL*, we used the 8 bytes of memory allocated to the empty array *ALL* to, respectively, represent left and right pointers. So, there is no space overhead (other than the space needed to keep track of the first node) associated with maintaining the two doubly-linked lists of PTST nodes that have an empty *ALL*.

Since an instance of PBOB may have up to $2n$ PTST nodes, the minimum space/memory required by these $2n$ PTST nodes is $24 * 2n = 48n$ bytes. However, some PTST nodes may have more than 4 pairs in their *ALL*. There can be at most $n/5$ such nodes. So, the maximum space-requirement of PBOB is $48n + 8n/5 = 49.6n$ bytes.

3.5.4 LMPBOB

In the case of LMPBOB, each node z of the PTST has the following fields: *color*, $point(z)$, *bit*, $leftChild$, and $rightChild$.

To improve the lookup performance of PBOB, the fields *minLength* (minimum of lengths of prefixes in $bit(z)$) and *maxLength* may be added. When the nodes of the PTST are augmented with a *minLength* and a *maxLength* field, we replace the statement `bit(z)->searchBitVector(d,hp,k)` of Figure 3–12 by

```
if (same(minLength, point(z), d)) {
   hp = k = minLength;
   bit(z)->searchBitVector(d,hp,k);
}
```

Observe that *maxLength* of LMPBOB is equivalent to *maxPriority* of BOB and PBOB.

Using 1 byte for each of the fields: *color*, *minLength*, and *maxLength*; 8 bytes for *bit* (this analysis is for IPv4); and 4 bytes for each of the remaining fields, the size of a PTST node of LMPBOB is 23 bytes. Again, to easily align PTST nodes along 4-byte boundaries, we pad an LMP PTST node so that its size is 24 bytes.

For the doubly-linked lists of PTST nodes with an empty *bit* vector, we used the 8 bytes of memory allocated to the empty bit vector *bit* to represent left and right pointers. So, there is no space overhead (other than the space needed to keep track of the first node) associated with maintaining the two doubly-linked lists of PTST nodes that have an empty *bit*.

Since an instance of LMPBOB may have up to $2n$ PTST nodes, the space/memory required by these $2n$ PTST nodes is $24 * 2n = 48n$ bytes.

3.6 Experimental Results

3.6.1 Test Data and Memory Requirement

We implemented the BOB, PBOB, and and LMPBOB data structures and associated algorithms in C++ as described in Section 3.5 and measured their performance on a 1.4GHz PC. To assess the performance of these data structures, we used six IPv4 prefix databases obtained from [38][2] . We assigned each prefix a priority equal to its length. Hence, BOB, PBOB, and LMPBOB were all used in a longest matching-prefix mode. For dynamic router-tables that use the longest matching-prefix tie breaker, the PST structure of Lu et al. [33, 34] provides $O(\log n)$ lookup, insert, and delete. So, we included the PST in our experimental evaluation of BOB, PBOB, and LMPBOB.

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 3–2. For the memory

[2] Our experiments are limited to prefix databases because range databases are not available for benchmarking

requirement, we performed two measurements. Measure1 gives the memory used by a data structure that is the result of a series of insertions made into an initially empty instance of the data structure. For Measure1, less than 1% of the PTST-nodes in the constructed BOB, PBOB, and LMPBOB instances are empty. So, these data structures use close to the minimum amount of memory they could use. Measure2 gives the memory used after 75% of the prefixes in the data structure constructed for Measure1 are deleted. In the resulting BOB, PBOB, and LMPBOB instances, almost half the PTST nodes are empty. The datbases Paix1, Pb1, MaeWest and Aads were obtained on Nov 22, 2001, while Pb2 and Paix2 were obtained Sep 13, 2000. Figures 3–14 and 3–15 histogram the data of Table 3–2. The memory required by PBOB and LMPBOB is the same when rounded to the nearest KB. This is so because in each of these structures, the number of PTST nodes is the same; the minimum size of a PTST node in PBOB is 24 bytes, very few PTST nodes of PBOB are bigger than 24 bytes because the average value of $|ranges(z)|$ is about 1 for our data sets and the maximum value is at most 6; and the size of PTST node in LMPBOB is 24 bytes. In Measure1, the memory required by BOB is about 2.38 times that required by PBOB and LMPBOB. However, in Measure2, this ratio is about 1.75. Also, note that, in Measure1, PST takes slightly more memory than does BOB, whereas, in Measure2, BOB takes about 50% more memory than does PST. We note also that the memory requirement of PST may be reduced by about 50% using a priority-search-tree implementation different from that used in [33]. Of course, using this more memory efficient implementation would increase the run-time of PST.

3.6.2 Preliminary Timing Experiments

We performed preliminary experiments to determine the effectiveness of the changes suggested in Section 3.5. Since these changes are only to the lookup algorithm, our preliminary timing experiments masured only the lookup times for the BOB, PBOB, and LMPBOB data structures. To obtain the mean lookup-time, we

Table 3–2: Memory usage

Database		Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Num of Prefixes		16172	22225	28889	31827	35303	85988
Measure1 (KB)	PST	884	1215	1579	1740	1930	4702
	BOB	851	1176	1526	1682	1876	4527
	PBOB	357	495	642	708	790	1901
	LMPBOB	357	495	642	708	790	1901
Measure2 (KB)	PST	221	303	395	435	482	1175
	BOB	331	455	592	652	723	1760
	PBOB	189	260	338	372	413	1007
	LMPBOB	189	260	338	372	413	1007

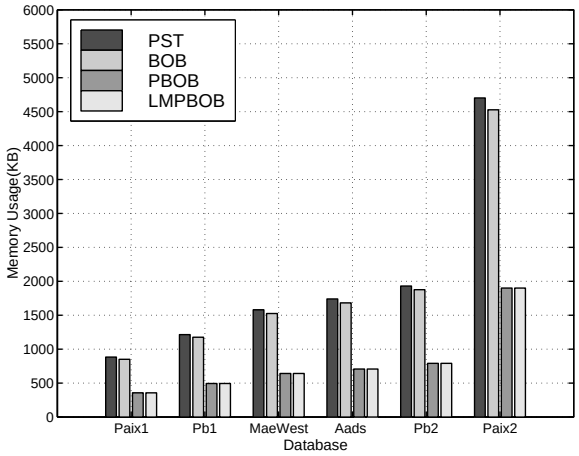


Figure 3–14: Memory usage–measure1

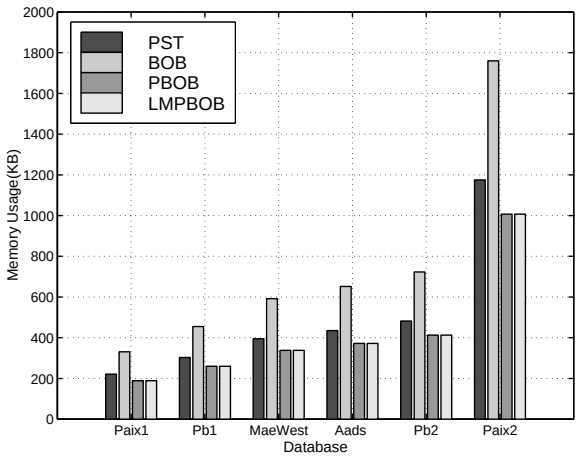


Figure 3–15: Memory usage–measure2

started with a BOB, PBOB, or LMPBOB that contained all prefixes of a prefix database. Next, we created a list of the start points of the ranges corresponding to

the prefixes in a database and then added 1 to each of these start points. Call this list L. A random permutation of L was generated and this permutation determined the order in which we searched for the longest matching-prefix for each of addresses in L. The time required to determine all of these longest-matching prefixes was measured and averaged over the number of addresses in L (actually, since the time to perform all these lookups was too small to measure accurately, we repeated the lookup for all addresses in L several times and then averaged). The experiment was repeated 10 times, each time using different random permutation of L, and the mean of these average times computed. The mean times for the implementation described in Section 3.5 is the base lookup-time.

For BOB, we found that omitting the predicates $d \leq maxFn$ and $minSt \leq d$ resulted in a mean lookup time that is approximately twice the base lookup time. On the other hand, elimination of the predicate $maxPriority > hp$ reduces the mean lookup time by about 2%. Even though the use of the predicate $maxPriority > hp$ increased the lookup time slightly on our test data, we believe this is a good heuristic for data sets in which the priorities are not highly correlated with the lengths of the prefixes or ranges. So, our remaining experiments retained this predicate. Eliminating the predicate $mp > hp$ had no noticeable effect on mean lookup time. This is to be expected on our data sets, because for these data sets, the maximum value of $|ranges(z)|$ is $\leq maxR = 6$. The predicate $mp > hp$ is expected to be effective on data sets with a larger value of $maxR$. So, we retained this predicate for our remaining tests.

For PBOB, elimination of the predicate $hp < maxPriority$ results in a very slight decrease in the mean lookup time relative to the base case. Hwoever, we expect that for data sets in which the priority isn't highly correlated with the prefix length, this predicate will actually reduce lookup time. Therefore, for further experiments, we retain this predicate in our lookup code.

In the case of LMPBOB, the introduction of the statement `hp = k = minLength` into the base code, results in a lookup time that is 15% less than when this statement is removed.

3.6.3 Run-Time Experiments

We measured the mean lookup-time as described in Section 3.6.2. The standard deviation in the average times across the 10 repetitions described in Section 3.6.2 was also computed. These mean times and standard deviations are reported in Table 3–3. The mean times are also histogrammed in Figure 3–16. It is interesting to note that PBOB, which can handle prefix tables with arbitrary priority assignements is actually 20% to 30% faster than PST, which is limited to prefix tables that employ the longest matching-prefix tie breaker. Further, lookups in BOB, which can handle range tables with arbitrary priorities are slightly slower than in PST. LMPBOB, which, like PST, is designed specifically for longest-matching-prefix lookups is slightly inferior to the more general PBOB.

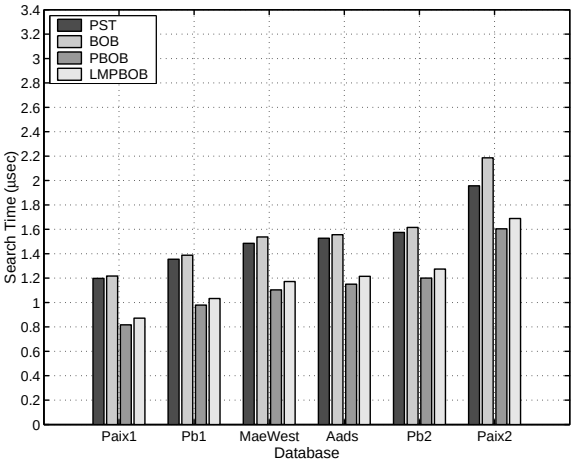


Figure 3–16: Search time

To obtain the mean insert-time, 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. (Once again,

Table 3–3: Prefix times on a 1.4GHz Pentium 4 PC with an 8K L1 data cache and a 256K L2 cache

Database			Paix1	Pb1	MaeWest	Aads	Pb2	Paix2
Search (μsec)	PST	Mean	1.20	1.35	1.49	1.53	1.57	1.96
		Std	0.01	0.01	0.04	0.01	0.00	0.01
	BOB	Mean	1.22	1.39	1.54	1.56	1.62	2.19
		Std	0.01	0.02	0.02	0.02	0.02	0.01
	PBOB	Mean	0.82	0.98	1.10	1.15	1.20	1.60
		Std	0.01	0.01	0.01	0.01	0.01	0.01
	LMPBOB	Mean	0.87	1.03	1.17	1.21	1.27	1.69
		Std	0.01	0.01	0.01	0.01	0.01	0.01
Insert (μsec)	PST	Mean	2.17	2.35	2.53	2.60	2.64	3.03
		Std	0.07	0.04	0.03	0.01	0.05	0.01
	BOB	Mean	1.70	1.89	2.06	2.10	2.16	2.55
		Std	0.06	0.06	0.05	0.05	0.05	0.03
	PBOB	Mean	1.04	1.25	1.39	1.44	1.51	1.93
		Std	0.06	0.05	0.00	0.05	0.05	0.06
	LMPBOB	Mean	1.06	1.29	1.47	1.50	1.57	1.98
		Std	0.07	0.07	0.06	0.06	0.04	0.01
Delete (μsec)	PST	Mean	1.72	1.87	2.06	2.09	2.11	2.48
		Std	0.04	0.05	0.05	0.06	0.04	0.06
	BOB	Mean	1.04	1.13	1.26	1.27	1.32	1.69
		Std	0.06	0.05	0.04	0.05	0.06	0.06
	PBOB	Mean	0.68	0.82	0.90	0.91	0.97	1.30
		Std	0.07	0.06	0.05	0.06	0.03	0.05
	LMPBOB	Mean	0.67	0.82	0.89	0.92	0.95	1.26
		Std	0.06	0.06	0.05	0.05	0.03	0.05
Num of Copies			15	11	9	8	8	3

since the time to insert the remaining 33% of the prefixes was too small to measure accurately, we started with several copies of the data structure and inserted the 33% prefixes into each copy; measured the time to insert in all copies; and divided by the number of copies and number of prefixes inserted). This experiment was repeated 10 times, each time starting with a different permutation of the database prefixes, and the mean of the mean as well as the standard deviation in the mean computed. These latter two quantities as well as the number of copies of each data structure we used for the inserts are given in Table 3–3. Figure 3–17 histograms the mean insert-time. As can be seen, insertions into PBOB take between 40% and 60% less

time than do insertions into PST; insertions into LMPBOB take slightly more time than do insertions into PBOB; and insertions into PST take 20% to 25% more time than do insertions into BOB.

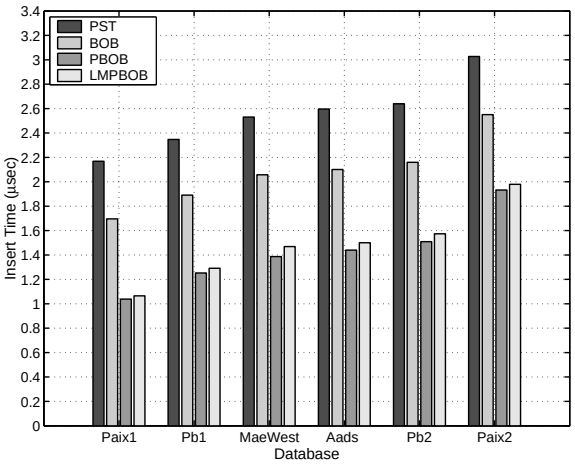


Figure 3–17: Insert time

The mean and standard deviation data reported for the delete operation in Table 3–3 and Figure 3–18 was obtained in a similar fashion by starting with a data structure that had 100% of the prefixes in the database and measuring the time to delete a randomly selected 33% of these prefixes. Deletion from PBOB takes less than 50% the time required to delete from an PST. For the delete operation, however, LMPBOB is slightly faster than PBOB. Deletions from BOB take about 40% less time than do deletions from PST.

3.7 Conclusion

Table 3.7 gives the worst-case memory required by each of the data structures. The data of this table are for IPv4. When comparing these memory requirement data, we should keep in mind that BOB, PBOB, and LMPBOB have different capabilities. BOB works for highest-priority matching with nonintersecting ranges; PBOB is limited to highest-priority matching with prefixes; and LMPBOB is limited

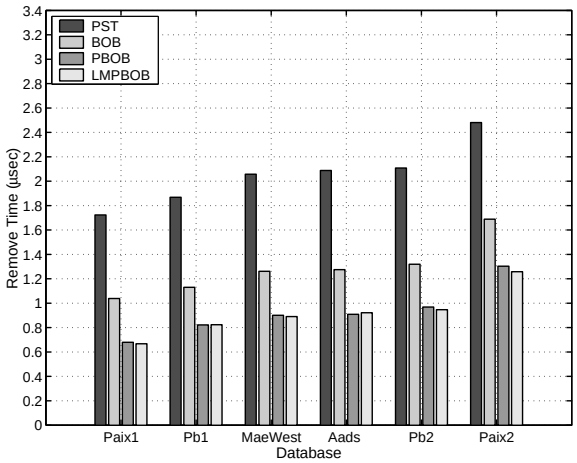


Figure 3–18: Delete time

to longest-length matching with prefixes. The PST structure of Lu et al. [33] has the same restrictions as does LMPBOB.

Table 3–4: Node sizes and worst-case memory requirement in bytes for IPv4 router tables.

	BOB	PBOB	LMPBOB	PST
Node Size	PTST(28) RST(20)	$\geq$24	24	28
Memory Required	$84n$	$49.6n$	$48n$	$56n$

Table 3–5 gives the asymptotic time complexity and Table 3–6 gives the asymptotic cache misses for our data structures. In these tables, $maxR$ is the maximum number of ranges or prefixes that match any destination address and $maxL$ is the maximum number of cache-lines needed by any of the array linear-lists stored in a PTST node. For LMPBOB, it is assumed that mask operations on W-bit vectors take $O(1)$ time and that an entire W-bit vector can be accessed with $O(1)$ cache misses.

Table 3–5: Time complexity

	BOB	PBOB	LMPBOB	PST
Search	$O(\log n \log maxR)$	$O(\log n + maxR)$	$O(\log n + W)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n + maxR)$	$O(\log n + \log W)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n + maxR)$	$O(\log n + \log W)$	$O(\log n)$

Table 3–6: Cache misses

	BOB	PBOB	LMPBOB	PST
Search	$O(\log n \log maxR)$	$O(\log n + maxL)$	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n + maxL)$	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n + maxL)$	$O(\log n)$	$O(\log n)$

Our experiments show that PBOB is to be preferred over PST and LMPBOB for the representation of dynamic longest-matching prefix-router-tables. This is somewhat surprising because PBOB may be used for highest-priority prefix-router-tables, not just longest-matching prefix-router-tables. A possible reason why PBOB is faster than LMPBOB is that in LMPBOB one has to check $O(W)$ prefix lengths, whereas in PBOB $O(maxR)$ lengths are checked (note that in our test databases, $W = 32$ and $maxR \leq 6$). BOB is slower than and requires more memory than PBOB when tested with longest-matching prefix-router tables. The same relative performance between BOB and PBOB is expected when filters are prefixes with arbitrary priority. Of the data structures considered in this chapter, BOB, of course, remains the only choice when the filters are ranges that have an associated priority.

Although the range allocation rule used by our data structures is similar to that used in an interval tree [40], the unique feature of our structures is the $2n$ size constraint. The size constraint is essential for $O(\log n)$ update.

CHAPTER 4
A B-TREE DYNAMIC ROUTER-TABLE DESIGN

In this chapter, 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 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 4.1, 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 4.2. Experimental results comparing the performance of our PIBT structure, the multiway range tree (MRT) structure of Suri

Table 4–1: An example prefix set R ($W = 5$)

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

et al. [20], and the best binary tree structure for dynamic prefix-tables, PBOB [35], are presented in Section 4.3.

4.1 Longest-Matching Prefix-Tables—LMPT

4.1.1 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 4–1 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 Suri et al. [20] (MRT) store the endpoints $E(P)$ together with additional information in a B-tree[1] [41], each structure uses a different variety of B-tree. Our PIBT structure uses a B-tree in which each key (endpoint) is stored

[1] 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.

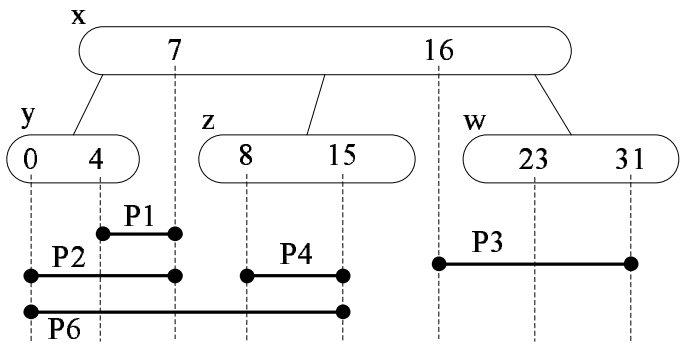


Figure 4–1: B-tree for the endpoints of the prefixes of Figure 4–1

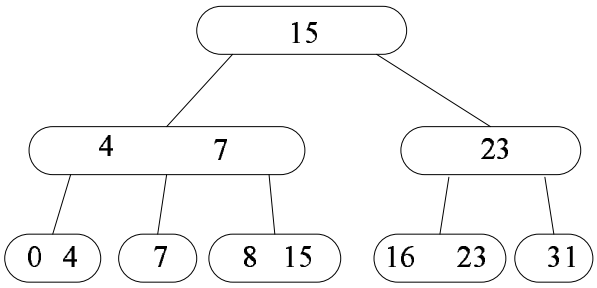


Figure 4–2: Alternative B-tree for Figure 4–1

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 4–1 shows a possible order-3 B-tree for the endpoints of the prefix set of Figure 4–1. 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 4–2 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:

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

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,

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

. For the example B-tree of Figure 4–1, $int(x) = [0, 31]$, $int_0(x) = int(y) = [0, 7]$, $int_1(x) = int(z) = [7, 16]$, $int_2(x) = int(w) = [16, 31]$, $int_0(y) = [0, 0]$, $int_1(y) = [0, 4]$, $int_2(y) = [4, 7]$, and $int_0(z) = [7, 8]$.

Node x of a PIBT has $t + 1$ W-bit vectors $x.interval_i$, $0 \leq i \leq t$ and t W-bit vectors $x.equal_i$, $1 \leq i \leq t$. 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 4–1, $y.interval_2[3] = 1$ 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.

4.1.2 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 4–3) employs the following lemma.

Lemma 33 *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 $lmp(d)$.

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

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

Figure 4–3: Algorithm to find the length of $lmp(d)$

$O(\log_2 n + \log_2 W) = O(\log_2(nW))$. The number of nodes accessed by the algorithm is $O(\log_m n)$.

4.1.3 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.

4.1.4 Inserting an endpoint

The algorithm to insert an endpoint u into the PIBT is an adaptation of the standard B-tree insertion algorithm [41]. 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 [41] 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 4–4).

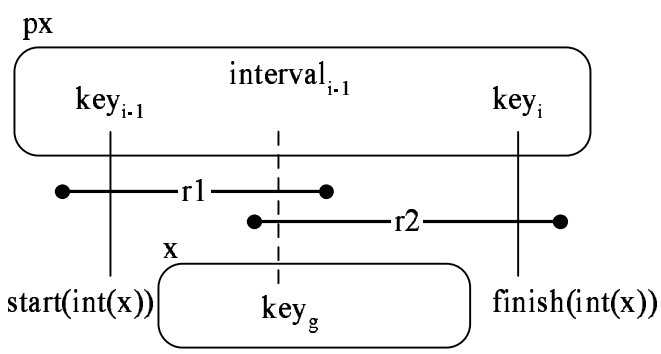


Figure 4–4: Node splitting

Algorithm $insertEndPoint(u, x, i)$ (Figure 4–5) 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

```
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 = ⌈m/2⌉;
        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 ≤ 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 4–5: Insert a new endpoint into the PIBT

```
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) ⊆ 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 ≥ start(p);
    Let j be the largest integer such that key_j ≤ finish(p);
    x.interval_q[length(p)] = 1, i ≤ q < j;
    if (i > 0) updateIntervals(p, child_{i-1});
    if (j <= t && j ≠ i − 1) updateIntervals(p, child_j);
}
```

Figure 4–6: Update intervals to account for prefix p

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 $O(\log_m n)$.

4.1.5 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 of Figure 4–6. 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 4–7 shows a possible set of nodes visited by x.

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.

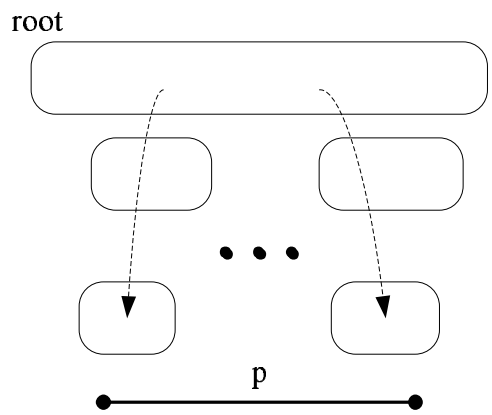


Figure 4–7: Nodes visited when updating *intervals*

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 $O(\log_m n)$.

4.1.6 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 4–6) 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.

The B-tree key deletion algorithm of Sahni et al. [41] 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.

4.1.7 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.

4.1.8 Borrow from a Sibling

Suppose that x's nearest left sibling y has more than $\lceil m/2 \rceil$ keys (Figure 4–8). 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).

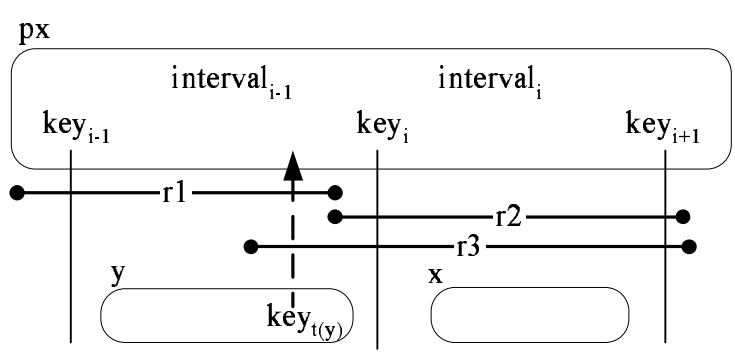


Figure 4–8: Borrow from right sibling

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.

4.1.9 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 4–9 shows the situation when y is the nearest right sibling of x.

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 *intervals* 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 *intervals* of y. $px.interval_i$ is removed from px.

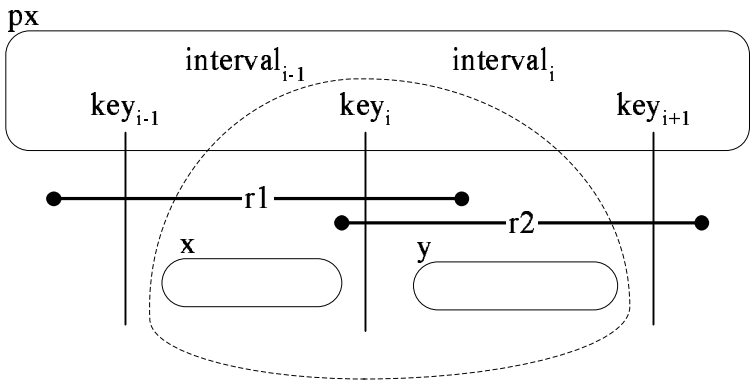


Figure 4–9: Merge two nodes

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.

4.1.10 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 $x.child_i$ [41]. Let $y.key_{t(y)}$ be the largest key in the subtree $x.child_{i-1}$ (Figure 4–10).

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

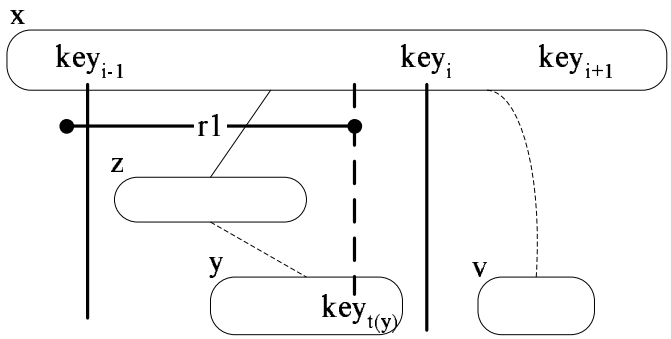


Figure 4–10: Deleting $x.key_i$

$[start(int(z)), key_{t(y)}]$ but not the range $[start(int(z)), u]$ are to be removed from the *intervals* 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 *intervals* 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 4–11. 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'.

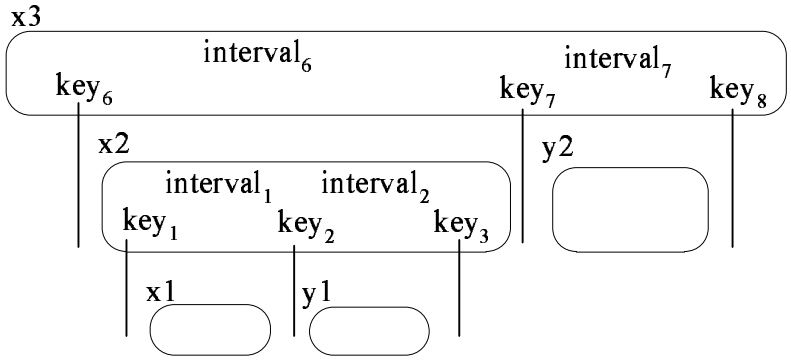


Figure 4–11: Merging adjacent siblings

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.

4.1.11 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 Lu et al. [35], 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 Lu et al. [35] 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.

4.2 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.

4.2.1 Preliminaries

Definition 19 *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 20 *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 4–1).

Lemma 34 *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. ■

4.2.2 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 35 *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 34(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 $O(n \log_m n)$.

4.2.3 RIBT Operations

Figure 4–12 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 4–3) primarily in the absence of the **break** statement in the **while** loop. Since Lemma 33 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

```
Algorithm hp(d) {
    // return the priority of the highest-priority range that matches d
    hp = −1; // assume that all priorities are ≥ 0
    x = root of RIBT;
    while (x ≠ null) {
        Let t = number of keys in x;
        Let i be the smallest integer such that x.key_{i−1} ≤ d ≤ x.key_i.
        if (i ≤ 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 4–12: Algorithm to find priority of $hpr(d)$

$D \leq W$ for prefixes, in practice, we expect D to be quite small (in practical IPv4 prefix databases, for example, $D \leq 6$ [21]). Each $intervalH$ max-heap of the RIBT has at 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 $O(\log_m n)$.

4.3 Experimental Results

We implemented the B-tree router-table data structures PIBT (Section 4.1) 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 Lu et al. [35] in our performance measurements. Our experiments were conducted using six IPv4 prefix databases obtained from [38]. 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 4–2. Although our PIBT structure uses about 12% less memory than is used by the MRT structure of Suri et al. [20], the PBOB structure of Lu et al. [35] uses about one-half the memory used by PIBT.

Table 4–2: Memory Usage. $m = 32$ for PIBT and MRT

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

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 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 4–3 gives the measured average lookup times. These average times are histogrammed in Figure 4–13. 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.

Table 4–3: Lookup time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT. Variance is < 0.02

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

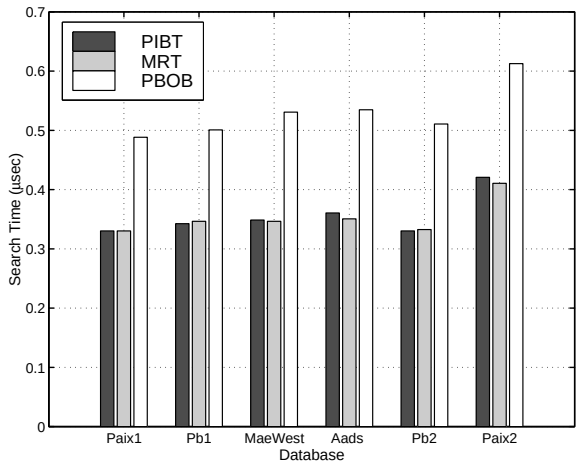


Figure 4–13: Lookup time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT

For the average update (insert/delete) time, we start by selecting 1000 prefixes from the database. 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 4–4 gives the computed mean times and Figure 4–14 histograms these average times.

Table 4–4: Update time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT

		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

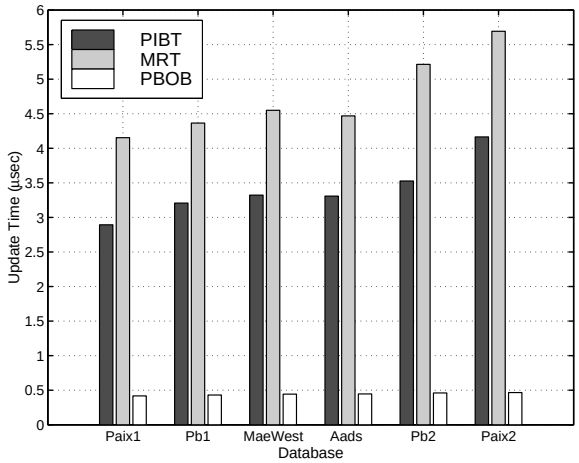


Figure 4–14: Update time on a Pentium III 700MHz PC. $m = 32$ for PIBT and MRT

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.

4.4 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 Suri et al. [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.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

We have developed several data structures for dynamic router tables. The novelty of our data structures is supporting real-time update, supporting range filters, and supporting the highest-priority matching tie breaker (BOB, $PBOB$, $RIBT$).

The first data structure, PST [33, 34], permits one to search, insert, and delete in $O(\log n)$ time each using the most specific matching tie breaker. Although $O(\log n)$ time data structures for prefix tables were known prior to our work [21, 22], PST is more memory efficient than the data structures of Sahni et al. [21, 22]. Further, PST is significantly superior on the insert and delete operations, while being competitive on the search operation. For nonintersecting ranges and conflict-free ranges PSTs are the first to permit $O(\log n)$ search, insert, and delete. All of our data structures based on priority search tree uses $O(n)$ memory.

The second data structure, BOB [35], works for highest-priority matching with nonintersecting ranges. Its variant, $PBOB$, works for prefix set or nonintersecting range set with a limited number of nesting levels (6 for IPv4 backbone router table [38]). In order to support $O(\log n)$ time update, BOB transforms the problem of removing an empty degree-2 node from the top-level tree to the problem of removing an empty degree-1 or degree-0 node from the top-level tree. Our experiments show that $PBOB$ is to be preferred over PST for the representation of dynamic longest-matching prefix-router-tables. But PST remains the only choice for detecting intersect and conflict between ranges in $O(\log n)$ 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. Both of them use $O(n)$ memory.

The third data structure is based on B-tree in order to utilize the wide cache line size. It is designed for prefix filters as well as non-intersecting range filters. Suri et al. [20] proposed a multi-way range tree that is also based on B-tree. A crucial difference between our data structure for prefix filters and that 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, ours is faster for the update operation. For the search operation, both B-tree structures take about the same time. When compared to $PBOB$, we find 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.

5.2 Future Work

It is more challenging to design data structures for multidimensional router tables. The problem of point location in a set of n non-overlapping d-dimensional hyper-rectangles requires $O(\log n)$ time with $O(n^d)$ memory requirement or $O((\log n)^{d-1})$ time with $O(n)$ memory requirement [43]. Multidimensional classification is no easier than point location problem since the filters can overlap. The above complexity bounds do not consider update. Thus they are for static data structures.

Almost all the existing schemes for multidimensional router tables focus on static data structures for prefix filters. Gupta et al. [43] review static data structures for multidimensional router tables. Hierarchical tries requires $O(W^d)$ time for search with $O(dWn)$ memory requirement. Hierarchical tries can support update in $O(dW)$ time. Set pruning trees [44] support search in $O(dW)$ time with $O(n^d)$ memory requirement. Cross-producting [44] decomposes the search into d one-dimensional

classifications and thus supports search in $O(dT)$ time, where T is the search time for one-dimensional classification. Cross-producting requires $O(n^d)$ memory, and is also suitable for range filters if the data structure used for one-dimensional classification also supports range filters. HiCuts [47] supports search in $O(W)$ time with $O(n^d)$ memory requirement. Other data structures, like Grid-of-tries [44], FIS tree [45], and tuple space search [46] are also designed for static router tables.

Designing schemes for dynamic multidimensional router tables is even more challenging. While the Internet community is trying to reduce the workload of packet classification inside core using technology like MPLS, classification still has to be done somewhere to aggregate and deaggregate the traffic inside a network as long as the packets still carry IP addresses, port numbers and other related fields.

REFERENCES

[1] C. Labovitz, G.Malan, and F.Jahanian, Internet routing instability, *ACM SIGCOMM*, Cannes, French Riviera, France, September 1997.

[2] C. Labovitz, A.Ahuja, A.Bose, and F.Jahanian, Delayed Internet routing convergence, *ACM SIGCOMM*, Stockholm, Sweden, August-September 2000.

[3] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S.Wu and L. Zhang, Improving BGP convergence through consistency assertions, *IEEE INFOCOM*, New York City, New York, USA, June 2002.

[4] C. Macian, R. Finthammer, An evaluation of the key design criteria to achieve high update rates in packet classifiers, *IEEE Network*, 15(6):24-29, November/December 2001.

[5] F.Baboescu, S.Singh, and G.Varghese, Packet classification for core routers: is there an alternative to CAMs? *IEEE INFOCOM*, San Francisco, California, USA, April 2003.

[6] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 15(2):8-23, March/April 2001.

[7] 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)*, Makati City, Metro Manila, Philippines, May 2002.

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

[9] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *ACM SIGCOMM*, Cannes, French Riviera, France, September 1997.

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

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

[12] V. Srinivasan , George Varghese, Faster IP lookups using controlled prefix expansion, *ACM SIGMETRICS Performance Evaluation Review*, 26(1):1-10, 1998

[13] 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 (FTDCS)*, Bologna, Italy, October-November 2001.

[14] S. Sahni and K. Kim, Efficient construction of variable-stride multibit tries for IP lookup, *Proceedings IEEE Symposium on Applications and the Internet (SAINT)*, Nara city, Nara, Japan, January-February 2002.

[15] P.Gupta, S.Lin, and N. McKeown, Routing lookups in hardware at memory access speeds, *IEEE INFOCOM*, San Francisco, USA, March-April 1998.

[16] N. Huang, S. Zhao, A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers, *IEEE Jounal on Selected Areas in Communications*, 17(6):1093-1104, June 1999.

[17] A.Basu, G.Narlika, Fast incremental updates for pipelined forwarding engines, *IEEE INFOCOM*, San Francisco, California, USA, April 2003.

[18] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *ACM SIGCOMM*, Cannes, French Riviera, France, September 1997.

[19] B. Lampson, V. Srinivasan, and G. Varghese, IP lookup using multi-way and multicolumn search, *IEEE INFOCOM*, San Francisco, USA, March-April 1998.

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

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

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

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

[24] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of data structures in C++, W.H. Freeman, New York, 1995.

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

[26] A. McAuley and P. Francis, Fast routing table lookups using CAMs, *IEEE INFOCOM*, San Francisco, CA, USA, March-April 1993.

[27] D.Shah, and P.Gupta, Fast updating algorithms for TCAMs, *IEEE MICRO*, 21(1):36-47, 2001.

[28] C.Matsumoto, CAM vendors consider algorithmic alternatives, http://www.eetimes.com/story/OEG20020520S0014, *EETIMES*, May 20, 2002.

[29] G. Cheung and S. McCanne, Optimal routing table design for IP address lookups under memory constraints, *IEEE INFOCOM*, New York City, New York, USA, March 1999.

[30] G. Chandranmenon and G. Varghese, Trading packet headers for packet processing, *IEEE Transactions on Networking*, 4(2): 141-152, 1996.

[31] P. Newman, G. Minshall, and L. Huston, IP switching and gigabit routers, *IEEE Communications Magazine*, 64-69, January 1997.

[32] A. Bremler-Barr, Y. Afek, and S. Har-Peled, Routing with a clue, *ACM SIGCOMM*, Cambridge, MA, USA, September 1999.

[33] H. Lu and S. Sahni, Priority search trees and dynamic router-tables. Submitted.

[34] H. Lu and S. Sahni, O(log n) dynamic router-tables for ranges, *IEEE Symposium on Computers and Communications*, Kiris-Kemer, Turkey, June-July 2003.

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

[36] A. Hari, S. Suri, and G. Parulkar, Detecting and resolving packet filter conflicts, *IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.

[37] E. McCreight, Priority search trees, *SIAM Jr. on Computing*, 14(2):257-276, 1985.

[38] Merit, Ipma statistics, http://nic.merit.edu/ipma, November 25, 2001.

[39] K. Melhorn, Data structures and algorithms 3: Multi-dimensional searching and computational geometry, Springer Verlag, New York, 1984.

[40] T.H.Cormen, C.E.Leiserson, R.L. Rivest, C.Stein, Introduction to Algorithms, 2nd edition, McGraw Hill, New York, 2001.

[41] S. Sahni, Data structures, algorithms, and applications in Java, McGraw Hill, New York, 2000.

[42] P. Warkhede, S. Suri, and G. Varghese, Fast packet classification for two-Dimensional conflict-Free filters, *IEEE INFOCOM*, Anchorage, Alaska, USA, April 2001.

[43] P. Gupta and N. Mckeown, Algorithms for packet classification, *IEEE Network*, 15(2):24-32, 2001.

[44] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, *ACM SIGCOMM*, Vancouver, BC, Canada, August-September 1998.

[45] A. Feldmann, S. Muthukrishnan, Tradeoffs for packet classification, *IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.

[46] V. Srinivasan, S. Suri, G. Varghese, Packet classification using tuple space search, *ACM SIGCOMM*, Cambridge, Massachusetts, USA, September 1999.

[47] P. Gupta and N. Mckeown, Classification using hierarchical intelligent cuttings, *IEEE Micro*, 20(1):34-41, 2000.

BIOGRAPHICAL SKETCH

Haibin Lu received the Bachelor of Engineering and Master of Engineering degrees from the Electronic Engineering Department at Tsinghua University, China, in 1997 and 1999 respectively. He did research on multimedia processing and communication for the master's and bachelor's degrees. He received the Ph.D. degree from the Computer and Information Science and Engineering Department at the University of Florida in 2003. His doctoral research focuses on Internet routing, especially on designing efficient data structures for packet classification. His current research interests include Internet routing, multimedia communication, wireless networks, and network security.