Dynamic Segment Trees for Ranges and Prefixes

Yeim-Kuan Chang and Yung-Chieh Lin

Abstract—In this paper, we develop a segment tree data structure for solving dynamic table lookup problems. The proposed dynamic segment tree (DST) uses all of the distinct end points of ranges as the keys based on a new range end point scheme. The new end point scheme generates fewer end points than the traditional end point scheme. DST is implemented as a balanced binary search tree augmented with a range set in each node. The performance of accessing and updating the ranges stored in each node is improved by an efficient range set data structure that combines the priority queue and the interval tree. Based on the proposed data structures, the time complexities of search, insertion, and deletion in a set of N arbitrary ranges are $O(\log N)$, $O(\log N \times \log Max)$, and $O(Max \times \log N \times \log Max)$, respectively, where Max is the maximum number of ranges covering any address. In practical routing tables, Max is a small constant (six for the routing tables we tested). The memory requirement for DST is $O(N \log N)$. The experimental results using real Internet Protocol version 4 (IPv4) routing tables show that both the DST and prefix binary tree on binary tree (PBOB) by Lu et al. (2004) perform much better than the multiway range tree (MRT) by Warkhede et al. (2004) and prefix in B-tree (PIBT) by Lu et al. (2005) in terms of update speed and memory consumption, but DST performs much better than PBOB and a little slower than MRT and PIBT in terms of search speed.

Index Terms—Segment tree, elementary intervals, B-trees, dynamic routing tables.

1 Introduction

The World Wide Web (WWW) and many emerging Internet applications, such as video conferencing, remote distance learning, and digital libraries, have created heavy traffic on the Internet. Thus, to continue providing good quality of service on the Internet, four key issues must be addressed in designing next-generation Internet routers [3]. These are

1. higher link speeds,
2. better router data throughput,
3. a faster packet forwarding rate, and
4. quick adaptation to route changes [9].

The solutions to the first two issues are now readily available from fiberoptic cables and new Internet Protocol (IP)-switching technology [20]. This paper deals with the third and fourth issues.

Since the number of end users and the routing information on the Internet has grown enormously, prefix matching was introduced in the early 1990s to reduce the size of a routing table. Further reduction in the size of router tables can be achieved by means of the Classless Interdomain Routing (CIDR) [7] scheme, which was introduced in 1993 and which aggregates network addresses in arbitrary powers of two. With CIDR, a network has a prefix length of about 0 to 32 bits. When an IP packet is received, the IP router computes the longest matching prefix in its routing table among multiple prefixes that match the destination address of the packet. The packet is then forwarded from the input interface to the output interface associated with the longest matching prefix.

To forward packets at high speeds and at fast IP lookup, algorithms are implemented in routers. These IP lookup algorithms can be broadly classified into two categories: static and dynamic. The static lookup schemes usually need precomputation to improve the lookup speed and reduce the memory requirement. The drawback of static schemes is that, when a single prefix is added or deleted, the entire table lookup data structure may need to be rebuilt. Routing table rebuilding has a negative impact on lookup performance and is thus not suitable for dynamic routing tables. The dynamic schemes can add or delete prefixes in real time.

In this paper, we propose the dynamic segment tree (DST) that is suitable for dynamic routing tables. We solve the IP lookup problem by treating the prefixes as ranges. In other words, our data structure not only solves the prefix match problem but also the general range match problem. DST is a segment tree in which the keys are the distinct end points of the ranges using the proposed range end point scheme. The segment tree in DST is constructed based on the concept of *elementary intervals* [1], [4], which will be defined later in this paper. All of the elementary intervals are disjoint and cover the entire address space. Each leaf node in DST corresponds to an elementary interval. Thus, searching DST for the ranges that match a given address d can be performed by finding the leaf node that corresponds to the elementary interval containing address d. Although many existing dynamic schemes also use segment trees or other balanced search trees, such as priority search trees

• *The authors are with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, Ta-Hsueh Road, 701 Tainan, Taiwan, ROC.*
E-mail: ykchang@mail.ncku.edu.tw, p7894110@ccmail.ncku.edu.tw.

Manuscript received 28 Jan. 2006; revised 30 July 2006; accepted 2 Nov. 2006; published online 27 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0030-0106.
Digital Object Identifier no. 10.1109/TC.2007.1037.

(PSTs) and interval trees, the proposed DST has the following advantages:

1. Both the segment tree and interval tree are simpler than a PST and, thus, they are easy to implement. For the segment tree, the matching ranges of a given address d are in the nodes on the searching path from the root to the leaf node. No nonmatching range is involved in the search. However, for the interval tree, the matching or nonmatching ranges of address d may be in the nodes on the searching path. The search in the interval tree needs to sort out the matching ranges of d from all of the ranges encountered on the searching path of the interval tree. Thus, the search in the interval tree is slower than that in the segment tree.
2. The keys in existing segment-tree-based data structures like the multiway range tree (MRT) [26] and prefix in B-tree (PIBT) [15] are according to the traditional end point definition in which, for a range $[e, f]$, keys e and f are used. However, in the proposed DST, $e-1$ and f are used as the keys. As shown in our experiments for real routing tables, the new end point scheme generates fewer keys than the traditional end point scheme. Therefore, the height of DST is smaller than that if the traditional end points are used.
3. Existing segment-tree-based data structures MRT and PIBT need an extra equal list or heap associated with each key to record which range starts or finishes at the key. Equal lists or heaps are used to determine if a key has to be removed from a node after a range is deleted. Each node in DST is augmented with a range set called the *canonical set*, which is a direct correspondence to the span list in MRT or the interval heap in the range in B-tree (RIBT). Based on the concept of elementary intervals, the proposed DST node deletion rule can determine if a key needs to be removed without any extra information like equal lists or heaps. Thus, the node structure in DST is smaller. DST is also improved by the proposed range set data structure that combines the priority queue and the interval tree [4].

Of course, the segment-tree-based schemes are not without a disadvantage. DST stores two keys for one range and each range is stored in $O(\log N)$ nodes in a tree of N ranges, whereas the interval tree uses only one key for each range and each range is stored in only one node.

The proposed DST search, insertion, and deletion operations for N arbitrary ranges take $O(\log N)$, $O(\log N \times \log Max)$, and $O(Max \times \log N \times \log Max)$ time, respectively, where Max is the maximum number of ranges covering any address. We conduct performance experiments with real routing tables to compare DST with existing dynamic schemes such as the prefix binary tree on binary tree (PBOB) [14], PIBT [15], and MRT [26]. In terms of search speed, DST performs much better than PBOB, but it is a little worse than MRT and PIBT. In terms of update speed, DST and PBOB perform equally well. In addition, DST needs more memory than PBOB.

The rest of the paper is organized as follows: Section 2 reviews the existing dynamic schemes. In Section 3, we present a list of the terminologies used in this paper. Section 4 gives a detailed design of the proposed data structure and, finally, Section 5 presents the performance comparisons of existing dynamic schemes followed by the concluding remarks.

2 Discussion on Existing Schemes

Various algorithms for high-performance table lookups have been proposed in the literature. In [21], a large variety of routing lookup algorithms is surveyed and their complexities of the worst-case lookup, update, and memory references are compared. Despite the intensive research conducted in recent years regarding the IP lookup problem, further studies on the balance between lookup speed, memory requirement, update, and scalability for an efficient table lookup scheme are needed. The schemes proposed in the literature [5], [8], [11], [18], [23] are mostly static and thus cannot afford frequent insertions and deletions. Some schemes based on the trie data structure [21], like binary trie, multibit trie [24], Patricia trie [23], heap on trie (HOT) [27], and binary search tree on trie (BOT) [27], do not use precomputation and are thus suitable for dynamic routing tables. However, their lookup speeds degrade linearly with the address length when switching to Internet Protocol version 6 (IPv6). In this section, we review the existing dynamic schemes that not only allow dynamic insertions and deletions but also scale well to IPv6. We classify these dynamic schemes based on their basic data structures as follows. We assume that the number of ranges or prefixes considered is N, and the order of the B-tree is m.

Priority Search Tree (PST). PST [16] represents a dynamic set $\mathcal{G}$ of ordered tuples [x, y], where $\mathrm{x} \geq 0$, $\mathrm{y} \geq 0$, and no two tuples have the same x value. PST is a combination of a priority queue and a search tree that operates on x and y values, respectively. With PST, we can efficiently find tuples lying in a 2D range $(-\infty : xright] \times [yleft, yright]$. In [13], Lu and Sahni proposed a dynamic lookup scheme based on an enhanced PST that maps a range $\mathrm{R} = [s, f]$ into a PST tuple $[f, s]$. All ranges that match an address d can be obtained by finding all PST tuples lying in a 2D range $[\mathrm{d}, \infty] \times [\mathrm{d}, 0]$. To eliminate the possibility that two tuples have the same x value, they proposed two tuple transformations,

$$transform1([x, y]) = [2^W x + 2^W - 1 - y, y]$$

and

$$transform1(\mathcal{G}) = \{transform1([x, y]) | [x, y] \in \mathcal{G}\}.$$

Based on the proposed data structure, inserting/deleting a range into/from a set of conflict-free ranges and searching for the most specific range can be completed in $O(\log N)$ time each in the worst case.

Interval Tree. This is a binary search tree that can also be used to store a dynamic set of tuples [x, y]. Each node in an interval tree is associated with a key that must be covered by at least one range. Depending on whether a node can store more than one range, two different interval trees were

proposed in the literature [1], [4]. In [1], each node is allowed to store more than one range. Thus, we call it a *fat interval tree*. The number of nodes in the interval tree is $O(N)$. To insert a range $\mathrm{R} = [e, f]$, if R covers the key of the root, R is stored in the root. Otherwise, R is inserted in the left (right) subtree of the root when f is smaller (e is larger) than the key of the root. When R does not cover the key of any node that is traversed, a new node with the key selected from addresses e to f is created and inserted as the left or right child of the node that was last visited. Using the interval tree organization for a prefix routing table, the longest matching prefix, as well as the highest priority matching prefix, can be found in $O(\log N + k)$ time, where k is the number of prefixes that match the given address. Prefix insertion and deletion are very expensive because ranges in some nodes may need relocations after tree rotations. In [4], each node of the interval tree stores exactly one range. This is called the *thin interval tree*. Since ranges may overlap, two comparison rules are used to compare if a range is smaller or larger than another range. For two ranges $\mathrm{R1} = [e1, f1]$ and $\mathrm{R2} = [e2, f2]$, the first rule defines $\mathrm{R1} < \mathrm{R2}$ if $e1 < e2$. If there is a tie in the first rule, then the second rule applies. The second rule defines $\mathrm{R1} < \mathrm{R2}$ if R2 is a subrange of R1 (that is, $e1 = e2$ and $f1 > f2$). Based on the comparison rules, ranges can be stored in a balanced binary search tree such as red-black trees. In addition, each node stores a max value, which is the maximum of the finish end points of all ranges stored in the subtree rooted at that node. In contrast to the fat interval tree [1], prefix insertion and deletion take $O(\log N)$ time. However, $O(min\{N, k \log N\})$ time is needed to find the longest matching prefix, as well as the highest priority matching prefix, where k is the number of matched prefixes for a given address.

In [14], Lu and Sahni developed an enhanced fat interval tree [1] for dynamic routing tables. Two changes are reinforced on the interval tree. First, the rule that a key in the interval tree must be covered by at least one range is relaxed. Second, this relaxation is again restricted by a *size constraint* that limits the total number of nodes in the interval tree to be not more than 2N. Lu and Sahni also discussed the data structures that are suitable for organizing the ranges stored in each node of the interval tree based on nonintersecting ranges (binary tree on binary tree (BOB)), arbitrary ranges (Compact BOB (CBOB)), or prefixes (PBOB and longest matching prefix BOB (LMPBOB)). With real routing tables, BOB and PBOB complete a search, insertion, and deletion in $O(\log N)$ time.

Segment Tree. This is another search tree in which the keys are the distinct end points of all ranges. Each leaf node is associated with an *interval*, called the *elementary interval*, which is defined as a range of addresses. The interval of an internal node is the union of the intervals of its children. The range R is stored in a node x if it covers the interval associated with node x and not the interval associated with x's parent. Thus, each range may be stored in $O(N \log N)$ nodes in a binary segment tree or in $O(N \log_m N)$ nodes in a multiway segment tree of order m. However, the definitions of intervals for different segment-tree-based schemes are fundamentally different, which, in turn, make their performance varied.

Feldmann and Muthukrishnan [6] used a static data structure called fat inverted multiway segment (FIS) tree to solve the packet classification problem. To allow dynamic insertions and deletions of ranges, they proposed a *dynamic FIS tree* of order m in which the internal nodes to which the leaves are connected may have a degree of between m/c_1 and $c_2 \times m$ for suitable constants $\mathrm{c}_1, \mathrm{c}_2 \geq 1$. Other internal nodes have a degree of m. Thus, the search time of $O(\log_m N)$ can be achieved. Sahni and Kim [22] developed a data structure for dynamic routing tables that is called a collection of red-black trees (CRBT). The basic interval tree of CRBT is constructed from the distinct end points of all prefixes, which is similar to the segment tree. CRBT supports a search, an insertion, or a deletion in $O(\log N)$ time. In addition to CRBT, two multiway segment trees were proposed in [15] and [26]. Moreover, a B-tree data structure called MRT was proposed in [26] to find the longest matching prefix in $O(\log_m N)$ time and to insert or delete a prefix in $O(m \log_m N)$ time. MRT is suitable for both prefixes and ranges. However, many end points are duplicated in the internal nodes and a prefix may be stored in at most $m - 1$ nodes per B-tree level. This drawback increases the update time and memory requirement. The B-tree data structures called PIBT and RIBT in [15] were proposed for solving this drawback and, hence, use memory more efficiently. An important advantage of RIBT over MRT is that each prefix is stored in $O(1)$ B-tree nodes per B-tree level in RIBT. The asymptotic complexity to find the longest matching prefix is the same and the measured time for this operation is also nearly the same for PIBT and MRT. However, PIBT is more memory efficient by a constant factor than MRT.

Discussion. We first summarize the complexities of all of the dynamic schemes in Table 1 in which $\mathrm{DST_a}$ and $\mathrm{DST_p}$ are the proposed DST for arbitrary ranges and prefixes, respectively. We can see two obvious differences among all of these schemes. First, all interval-tree-based schemes and CRBT require the least memory, which is $O(N)$. Second, multiway segment-tree-based schemes take the least search time, which is $O(\log_m N)$. However, other performance subtleties among all schemes cannot be obtained without extensive performance experiments.

The experimental results in [13] showed that PST performs a little worse than CRBT in terms of search time. However, PST performs much better than CRBT in terms of insertion, deletion, and memory usage. The later experimental results in [12] showed that all search, insertion, and deletion operations for PST are even worse than those of the binary trie when large routing tables are used. One reason why PST does not scale well is because it takes time to transform a route prefix to satisfy the constraints of PSTs. This transformation complicates the lookup process and increases the memory requirement. Also, the memory requirement for PST may be too high for IPv6. Although the theoretical complexities of PBOB and LMPBOB are not better than other schemes, the experimental results in [14] using real routing tables showed that PBOB and LMPBOB

TABLE 1
Complexities of All Dynamic Schemes

Name	Search	Insert	Delete	Memory	Range Type
BTrie	W	W	W	NW	prefix
HOT	W	$W \log N$	$W \log N$	NW	arbitrary
BOT	$W \log N$	W	W	NW	arbitrary
FIS	$\log_m N$	$N\log N(\log_m N)^2$	$N\log N(\log_m N)^2$	$N \times N^{\log_{m'} N}$	arbitrary
BOB	$\log N \times \log Max$	$\log N$	$\log N$	N	nonintersecting
CBOB	$\log N + Max$	$\log N + Max$	$\log N + Max$		arbitrary
PBOB	W	W	W		prefix
LMPBOB	W	$\log N + \log W$	$\log N + \log W$		prefix
CRBT	$\log N$	$\log N$	$\log N$	N	prefix
MRT	$\log_m N$	$m\log_m N$	$m\log_m N$	N	prefix
PIBT	$\log_m N$	$m\log_m N$	$m\log_m N$	$mN\log_m N$	prefix
RIBT	$\log_m N$	$m\log_m N$	$m\log_m N$	$N\log_m N$	nonintersecting
DST_a	$\log N$	$\log N \times \log Max$	$\log N \times Max \times \log Max$	$N\log N$	arbitrary
DST_p	$\log N$	W	W		prefix

Assume that the maximum number of ranges which match any address is Max. For FIS, MRT, and PIBT, m is the order of the B-tree.

perform much better than PST in terms of search, insertion, and deletion times and memory usage.

3 PRELIMINARIES

The W-bit address space covers the addresses 0 to $2^W - 1$. A W-bit *prefix* is denoted by p/len in the length format, where p is a W-bit number and len is the prefix length. Usually, for a prefix $\mathrm{P} = p/len$, $p\%2^{\mathrm{W}-len}$ is zero, and P covers addresses p to $p + 2^{\mathrm{W}-len} - 1$, where % is the modulus operator. An example set of 6-bit prefixes is shown in Table 2 and will be used throughout this paper. A W-bit *range* $R = [e, f]$ is a series of consecutive W-bit addresses from e to f, where $0 \leq e \leq f \leq 2^W - 1$. $\mathrm{R1} = [e1, f1]$ is said to be *more specific* than $\mathrm{R2} = [e2, f2]$ if $e2 \leq e1$ and $f1 \leq f2$. By checking if a range is more specific than another, the conflicting range set is defined as follows:

Definition 1. *The range set $\mathcal{R}$ has a* conflict *if and only if there exists an address d for which, among all the ranges that cover d, no one range is more specific than the others.*

If a range set does not have a conflict, we call it a *conflict-free* range set. A set of prefixes is a conflict-free range set. For a set of N W-bit ranges, the address space of 0 to $2^W - 1$ can be seen as being partitioned into a number of address intervals. These intervals are called *elementary intervals* if they satisfy the following definition:

TABLE 2
An Example Prefix Set Consisting of Nine 6-Bit Prefixes

ID	Prefix	Range	Minus-1 scheme		Traditional scheme	
			start	finish	start	finish
P1	000000/2	[0, 15]	-	15	0	15
P2	010000/2	[16, 31]	15	31	16	31
P3	000100/4	[4, 7]	3	7	4	7
P4	100000/1	[32, 63]	31	-	32	63
P5	010110/5	[22, 23]	21	23	22	23
P6	110000/2	[48, 63]	47	-	48	63
P7	110000/4	[48, 51]	47	51	48	51
P8	110111/6	[55, 55]	54	55	55	55
P9	100000/3	[32, 39]	31	39	32	39

Definition 2. *Let the set of k elementary intervals constructed from a set $\mathcal{R}$ of W-bit ranges be $\mathcal{X} = \{X_i | X_i = [e_i, f_i], \text{for } i = 1 \text{ to } k\}$. $\mathcal{X}$ must satisfy the following:*

1. $e_1 = 0$ *and* $f_k = 2^W - 1$,
2. $f_i = \mathrm{e}_{i+1} - 1$ *for* $i = 1$ *to* $k - 1$,
3. *all addresses in X_i are covered by the same subset of $\mathcal{R}$ (called the range matching set of X_i), denoted by EI_i, and*
4. $\mathrm{EI}_i \neq \mathrm{EI}_{i+1}$, *for* $i = 1$ *to* $k - 1$.

From Definition 2, every two elementary intervals must be disjoint and the whole address space is the union of all elementary intervals. Fig. 1a shows the elementary intervals X_1 to X_{12} that are portioned by the distinct end points of nine prefixes in Table 2. The range matching sets of two consecutive elementary intervals must be different; otherwise, the end point delimiting these two elementary intervals should not exist. For example, $\mathrm{EI}_1 = \{\mathrm{P1}\} \neq \mathrm{EI}_2 = \{\mathrm{P1}, \mathrm{P3}\}$.

How should the end points of ranges be defined in order to precisely represent the elementary intervals? The traditional definition uses e and f of a range $[e, f]$ as end points. Partitioning the entire address space into disjoint segments by these end points must consider two types of addresses separately: the addresses that are equal to any end point and those that are not. The matching ranges for an address that is equal to one of the end points or that is between two end points must be stored separately. For example, the *binary range search (BRS)* proposed in [11] uses "$>$" and "$=$" ports to deal with the addresses that are equal to or not equal to any end point. Fig. 1b shows the list of end points and their associated "$>$" and "$=$" ports based on BRS. As a result, the data structure of BRS is not compact because the information of some "$>$" and "$=$" ports are redundant. Obviously, the definition of elementary intervals is not satisfied by the traditional end point scheme.

Subsequently, we propose a new end point scheme, called the *minus-1-end point* scheme, and show that the definition of elementary intervals can be precisely satisfied. No additional information like "$=$" ports in BRS is needed and the end point list is more compact than the traditional one.

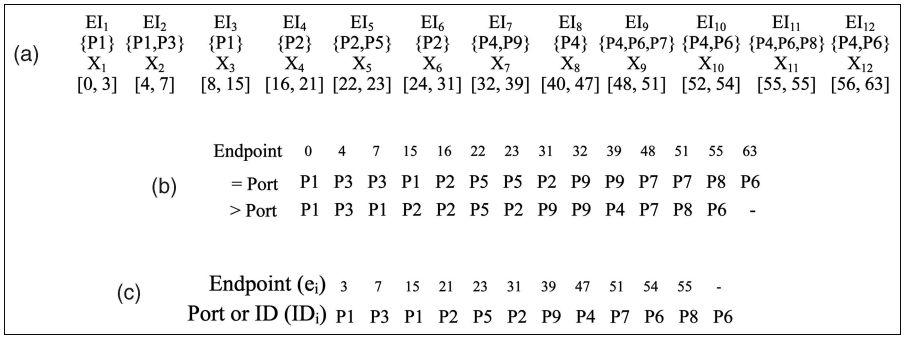

Fig. 1. Elementary intervals and end point schemes for ranges. (a) Elementary intervals. (b) Binary range search [11] that uses "$>$" and "$=$" ports. (c) Minus-1-end point scheme.

Definition 3 (minus-1-end point scheme). *The two end points of a range* $[\mathrm{e}, \mathrm{f}]$ *are* $e-1$ *and* f.

Since the entire W-bit address space is $[0, 2^W - 1]$, end points -1 and $2^W - 1$ are not physically stored in the list of end points. Let e_i for i $= 1$ to k be the distinct end points of a range set based on the minus-1-end point scheme. We can show that intervals $\mathrm{X}_1 = [0, e_1]$, $\mathrm{X_i} = [e_{i-1} + 1, e_i]$ for $i = 2$ to k, and $\mathrm{X_{k+1}} = [e_k + 1, 2^W - 1]$ are the elementary intervals defined in Definition 2. It is sufficient to show that the range matching set $\mathrm{EI_i}$ of $\mathrm{X_i} = [e_{i-1} + 1, e_i]$ is not the same as $\mathrm{EI_{i-1}}$ of $\mathrm{X_{i-1}} = [e_{i-2} + 1, e_{i-1}]$ or $\mathrm{EI_{i+1}}$ of $\mathrm{X_{i+1}} = [e_i + 1, e_{i+1}]$ as follows: We know that an end point is either the finish end point of a range or the start end point of a range minus one. Thus, for the interval $\mathrm{X_i} = [e_{i-1} + 1, e_i]$, we need to consider four possible cases, as shown in Table 3. In this study, we only describe the first case because the other three are similar. In the first case, e_{i-1} is the start end point of range $\mathrm{R1} = [s1, f1]$ minus one (that is, $e_{i-1} = s1 - 1$) and e_i is the start end point of range $\mathrm{R2} = [s2, f2]$ minus one (that is, $e_i = s2 - 1$). $\mathrm{R1} \notin \mathrm{EI_{i-1}}$ and $\mathrm{R1} \in \mathrm{EI_i}$ because $\mathrm{X_i} = [s1, s2 - 1]$ and $\mathrm{X_{i-1}} = [e_{i-2} + 1, s1 - 1]$. Similarly, $\mathrm{R2} \notin \mathrm{EI_i}$ and $\mathrm{R2} \in \mathrm{EI_{i+1}}$ because $\mathrm{X_i} = [s1, s2 - 1]$ and $\mathrm{X_{i+1}} = [s2, e_{i+1}]$. As a result, $\mathrm{EI_i}$ is not the same as $\mathrm{EI_{i-1}}$ or $\mathrm{EI_{i+1}}$.

Notice that e_1 (the smallest end point) may be zero, but e_k (the largest end point) cannot be $2^W - 1$. For example, the list of 11 end points constructed from the prefixes in Table 2 based on the minus-1-end point scheme is shown in Fig. 1c. The list of port IDs ($\mathrm{ID_1}$ to $\mathrm{ID_{11}}$) representing the routing information is also shown. A binary search can be used to find the matching elementary interval for a given address. If we precompute the range matching set for each elementary interval, a binary search scheme similar to the BRS that returns the highest priority matching range can be developed. It is worth saying that, for contiguous ranges (the finish end point of a range is equal to the start end point of another range minus one), the minus-1-end point scheme generates fewer distinct end points than the traditional end point scheme. For example, given two 4-bit ranges [0, 3] and [4, 7], the minus-1-end point scheme only needs two end points (3 and 7), whereas the traditional scheme needs four end points (0, 3, 4, and 7). We will show that this advantage of the minus-1-end point scheme is true for the prefixes in the routing tables in the performance evaluation section.

TABLE 3
Elementary Interval of $\mathrm{X_i}$ with $\mathrm{R1} = [\mathrm{s1}, \mathrm{f1}]$ and $\mathrm{R2} = [\mathrm{s2}, \mathrm{f2}]$

	Relations \ Interval	$\mathrm{X_{i-1}}$	$\mathrm{X_i}$		$\mathrm{X_{i+1}}$
1	$\mathrm{e_{i-1}} = \mathrm{s1} - 1$, $\mathrm{e_i} = \mathrm{s2} - 1$	$\mathrm{R1} \notin \mathrm{EI_{i-1}}$	$\mathrm{R1} \in \mathrm{EI_i}$	$\mathrm{R2} \notin \mathrm{EI_i}$	$\mathrm{R2} \in \mathrm{EI_{i+1}}$
2	$\mathrm{e_{i-1}} = \mathrm{s1} - 1$, $\mathrm{e_i} = \mathrm{f2}$	$\mathrm{R1} \notin \mathrm{EI_{i-1}}$	$\mathrm{R1} \in \mathrm{EI_i}$	$\mathrm{R2} \in \mathrm{EI_i}$	$\mathrm{R2} \notin \mathrm{EI_{i+1}}$
3	$\mathrm{e_{i-1}} = \mathrm{f1}$, $\mathrm{e_i} = \mathrm{s2} - 1$	$\mathrm{R1} \in \mathrm{EI_{i-1}}$	$\mathrm{R1} \notin \mathrm{EI_i}$	$\mathrm{R2} \notin \mathrm{EI_i}$	$\mathrm{R2} \in \mathrm{EI_{i+1}}$
4	$\mathrm{e_{i-1}} = \mathrm{f1}$, $\mathrm{e_i} = \mathrm{f2}$	$\mathrm{R1} \in \mathrm{EI_{i-1}}$	$\mathrm{R1} \notin \mathrm{EI_i}$	$\mathrm{R2} \in \mathrm{EI_i}$	$\mathrm{R2} \notin \mathrm{EI_{i+1}}$

4 Proposed Dynamic Segment Tree

Although the binary search on the linear list of end points based on the elementary intervals described in the previous section solves the range table lookup problem, inserting or deleting an end point requires shifting the entire list in the worst case. Thus, it is not suitable for frequent range updates. Alternatively, a segment tree [1] is a data structure designed for storing ranges based on the elementary intervals. The skeleton of the segment tree is a balanced binary search tree in which the keys are the distinct end points of a set $\mathcal{R}$ of N original ranges. The elementary intervals constructed from the end points of the ranges in $\mathcal{R}$ correspond to the leaf nodes of the segment tree. The interval covered by an internal node v is the union of elementary intervals corresponding to the leaf nodes in the subtree rooted at v. Each node v is associated with a subset of $\mathcal{R}$ (called the *canonical subset of* $\mathcal{R}$ or simply the *canonical set*) based on the *range allocation rule*, which will be defined later. The segment tree can efficiently determine the elementary interval that contains a given address. The set of matching ranges for the given address d can be obtained by traversing the segment tree from the root to the leaf node that corresponds to the elementary interval containing d.

Traditionally, the segment tree is constructed by precomputing the elementary intervals first and then using a bottom-up approach to finish the remaining parts of the tree structure [1]. This approach makes the segment tree have a static data structure. Hence, the segment tree does not support fast updates for dynamic routing tables. In this paper, we propose a data structure, called the DST, based on the segment tree. DST can dynamically insert/delete prefixes or ranges into/from the segment tree. The proposed DST completely avoids rebuilding the entire data structure while performing updates.

The AVL tree and the red-black tree [4] are the two most popular balanced binary search trees that have the same complexity for all tree operations. Red-black trees are

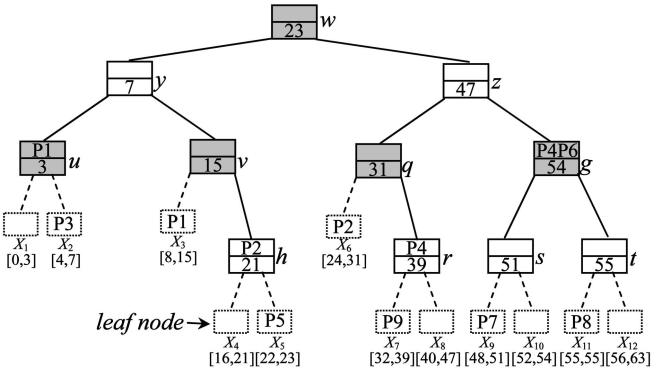

Fig. 2. A possible DST built according to the prefixes in Table 2.

```
Algorithm DST_Search(root, d)
{
01   z = root; k = 1;
02   while (z ≠ null){
03        if (z.Cset ≠ ϕ) Cset[k++] = z.Cset;
04        if (z is a leaf node) break;
05        if (d ≤ z.key) z = z.left;
06        else z = z.right;
07   }
08   if (the range set is conflict-free)
09        return the highest priority (most specific) range in Cset[k];
10   else
11        return the highest priority range among Cset[1], …, Cset[k];
}
```

Fig. 3. The DST search algorithm.

generally believed to be faster than AVL trees by a constant factor. For example, the AVL tree of N nodes may require $O(\log N)$ rotations to delete a node, whereas the red-black tree only needs three rotations at most. Therefore, we implement and describe the proposed DST with the red-black tree.

Similarly to that in the red-black tree, each node in DST contains the following fields: *color* (red or black), *key* (end points of ranges), and *left* and *right* (pointers to the left and right children). The field fld of node x is denoted by $x.fld$. We use $successor(x)$ and $predecessor(x)$ to denote the successor and predecessor of node x in the sorted order, respectively. If $successor(x)$ does not exist, its key is set to $2^W - 1$. Similarly, if $predecessor(x)$ does not exist, its key is set to -1. For the two nodes x and y, we simply write $x < y$ to indicate that $x.key < y.key$. Since we do not physically store the parent pointer in each node, we denote the parent of node x by $parent(x)$ instead of $x.parent$. Each node x in DST is associated with an interval denoted by $intvl(x)$. The interval and the canonical set of node x, denoted by $intvl(x)$ and $x.Cset$, are similarly defined as the traditional segment tree as described above.

Definition 4 (DST range allocation rule). *Range R is stored in the canonical set of node v ($v.Cset$) if and only if the interval of v ($intvl(v)$) is contained in R, but the interval of parent(v) ($intvl(parent(v))$) is not contained in R.*

The keys in DST are the end points of ranges derived from the minus-1-end point scheme. Fig. 2 shows a possible DST built from the prefixes in Table 2. For node y, $successor(y)$ and $predecessor(y)$ are v and u, respectively, and $intvl(y)$ is [0, 23]. Also, the two elementary intervals delimited by y are $[u.key + 1, y.key] = [4, 7]$ and $[y.key + 1, v.key] = [8, 15]$. For example, in Fig. 2, P5 is stored in the right leaf node of h because interval [22, 23] is contained in P5, but the interval [16, 23] associated with node v is not. Based on Definition 4, the following Cset properties of DST can be obtained. This property guarantees that any range is stored in at most two nodes per tree level in DST.

Cset Property of DST. *For each node v in DST, we have 1) $v.Cset \cap v.left.Cset = \phi$, 2) $v.Cset \cap v.right.Cset = \phi$, and 3) $v.left.Cset \cap v.right.Cset = \phi$.*

Since ranges may be added in or removed from the canonical set of a node, when we write "add a range R in node x," we mean adding R in $x.Cset$. The Cset field of leaf nodes is used in the same way as the regular nodes. Other fields in the leaf nodes are not used. Leaf nodes are not considered as regular nodes. Thus, when we perform rotations to balance DST (this will be described later), leaf nodes are not rotated.

4.1 Search in DST

The most specific range or the highest priority range that matches a query address can be found by traversing the tree from the root toward a leaf. Fig. 3 shows the DST search algorithm. The *while loop* in lines 2-7 implements the tree traversal according to the query key d. While traversing DST, all nonempty canonical sets of the traversed nodes are recorded in array Cset[1..k], where k is the number of nonempty canonical sets traversed. If the ranges in DST are conflict-free, the final matched range is the most specific range in Cset[k], which is the last nonempty canonical set traversed. However, if the ranges in DST are arbitrary, the final matched range is the highest priority range among all ranges in Cset[1] to Cset[k].

Consider the example DST in Fig. 2. If destination address d is 51, the query path of address d is w-z-g-s. Since P7 is the most specific range among all ranges (P4, P6, and P7) that match d, P7 is the final matched result.

Complexity. The DST search algorithm visits one node per level. Therefore, $O(\log N)$ nodes are traversed. Assume that the maximum size of a canonical set is $O(CSize)$ and that accessing the highest priority range in a canonical set takes $O(g(CSize))$ time. Thus, if the range set is conflict-free, the time complexity of a DST search is $O(\log N + g(CSize))$. Otherwise, if the range set is arbitrary, the time complexity of a DST search is $O(\log N + \log N \times g(CSize))$.

4.2 Insertion in DST

Inserting a range $R = [e, f]$ in DST requires the following steps:

1. If e is not zero, insert $e - 1$ as a new key in DST.
2. If f is not $2^W - 1$, insert f as a new key in DST.
3. Insert R in DST according to the DST range *allocation rule*.

```
Algorithm DST_Insert_Endpoint(root, ep)
{
01  y = null; x = root;
02  while (x ≠ leaf node){
03        if (x.key = ep) return; // key ep already exists
04        y = x;
05        if (ep < x.key) x = x.left;
06        else x = x.right;
07  }
08  x.parent = y;
09  x.key = ep; x.color = red;
10  attach leaf nodes to x.left and x.right;
11  if (y = null) root = x;
12  Insertion_Fixup(root, x);
}
```

Fig. 4. The DST end point insertion algorithm.

4.2.1 Inserting a New Key in DST

In Step 1 or 2, if the new key exists in DST, no new node is created. Otherwise, a new node associated with the new key will be created and inserted in DST as a leaf node.

We use a slightly modified version of the standard red-black tree insertion algorithm [4] to insert a new key into DST. Fig. 4 shows the insertion algorithm *DST_Insert_Endpoint*. Just like in DST search, the insertion algorithm traverses the tree from the root to a leaf node x according to the new key ep. If we find that the new key exists in DST when traversing the tree, the search stops. After a leaf node x is reached, it is assigned the new key and the color red. Node x becomes the regular node in DST. Finally, node x is attached with two leaf nodes as its left and right children.

Inserting a new key may cause a violation of one of the red-black properties [4] (that is, $x.parent.color = \text{red}$). Hence, line 12 in Fig. 4 invokes the algorithm $\text{Insertion\_Fixup}(root, x)$ that restores the red-black properties. Insertion_Fixup is similar to the algorithm *RB-Insert* in Section 14.3 of [4] and is thus omitted in this paper. Since the proposed DST is augmented with the canonical sets, the left and right rotations may change the intervals covered by the nodes involved in rotations. Hence, we have to adjust the canonical sets of said nodes. We will show how to do this adjustment in Section 4.3.

4.2.2 Inserting a Range into DST

After the two end points of range R are inserted in DST, R can then be inserted in the canonical sets of some proper nodes in DST according to the DST range allocation rule. Inserting a range $\text{R} = [e, f]$ takes the following steps:

1. Find the lowest common ancestor (LCA), *node* y, of two nodes with keys $e-1$ and f.
2. If R contains $intvl(y)$, add R in $y.Cset$.
3. If e is zero, insert R in $y.left.Cset$. Otherwise, insert R in $u_i.right.Cset$ for $i = 1$ to m if $u_{\text{i}}.right \neq u_{\text{i+1}}$, where u_1 to u_{m} are the descendants of node y in the path from y to u_{m}, that is, the node with key $e-1$, and insert R in $u_m.right.Cset$.
4. If f is $2^W - 1$, insert R in $y.right.Cset$. Otherwise, insert R in $v_i.left.Cset$ for $i = 1$ to n if $v_{\text{i}}.\text{left} \neq v_{\text{i+1}}$, where v_1 to v_{n} are the descendants of node y in the path from y to v_{n}, that, is the node with key f, and insert R in $v_n.left.Cset$.

Fig. 5 shows the algorithm $\text{DST\_Insert\_Range}(root, R)$ that inserts range R in DST. Since the two end points $e-1$ and f of range $\text{R} = [e, f]$ have been inserted in DST, we can locate the LCA (node y) of the two nodes with keys $e-1$ and f in the tree. Notice that y could be the node with key $e-1$ or f. While traversing the tree for finding the LCA node y, the lower and upper bounds lb and ub of the interval $intvl(y)$ can also be obtained. In Step 2, if range R contains the interval $[lb, ub]$, it must be stored in y. Steps 3 and 4 try to store range R in the subtrees that are rooted at node y's left and right children, respectively. Since keys -1 and $2^W - 1$ are not stored in DST, it is possible that range R completely contains the interval $intvl(y.left)$ or $intvl(y.right)$. Thus, the

```
Algorithm DST_Insert_Range(root, R) // assume R=[e, f]
{
    lb = 0; ub = 2^W – 1;
    ////////// step 1 //////////////////////////////////////////////////////////////////////////////////////////
01  Find LCA node y and the lower and upper bounds lb and ub of the subtree rooted at y;
    ////////// step 2 //////////////////////////////////////////////////////////////////////////////////////////
02  if ( [lb, ub] is contained in R ) { Add R in y.Cset; return;}
    ////////// step 3 //////////////////////////////////////////////////////////////////////////////////////////
03  lby = lb; uby = y.key;
04  if (e = 0) Add R in y.left.Cset; // i.e., [lby, uby] is contained in R
05  if (e – 1 < y.key) {
06        x = y.left;
07        while (x ≠ leaf node){
08              if (x.key = e – 1) { Add R in x.right.Cset; break; }
09              if (e – 1 < x.key) {
10                    Add R in x.right.Cset;
11                    x = x.left;
12              }else x = x.right;
13        }
14  }
    ////////// step 4 //////////////////////////////////////////////////////////////////////////////////////////
15  x = y.right; lby = y.key; uby = ub;
16  if (f = 2^W – 1) Add R in x.Cset; // i.e., [lby, uby] is contained in R
17  if clause is the same as lines 5-14 with 'e – 1' and 'f 'exchanged, '>' and '<'
    exchanged, and 'left' and 'right' exchanged
}
```

Fig. 5. The DST range insertion algorithm according to the range allocation rule.

```
Algorithm Optimized_DST_Insert(root, R) // Assume R=[e, f]
{
01  x = root;
02  if (x = dummy node) { Insert the first two nodes with keys e – 1 and f; return ;}
03  if (LCA node y for R is not found){
04        Find the leaf node x such that x.key < e – 1 or f < x.key
05        Insert two nodes with keys e – 1 and f in the tree;
06        perform necessary rotations;
07        return;
08  } else {
09        Find the lower and upper bounds lb and ub of subtree rooted at y;
10        if ([lb, ub] ⊆ R) {Add R in y.Cset; return; }
11  }
12  if (e = 0) Add R in y.left.Cset;
13  else if (e – 1 < y.key) {
14        x = y.left;
15        while (x ≠ dummy node){
16              if (x.key = e – 1) {Add R in x.right.Cset; break;}
17              if (e – 1 < x.key) { Add R in x.right.Cset; x = left(x); }
18              else x = x.right;
19        }
20  }
21  if (x = dummy node)
22        x.key = e – 1; x.color = red;
23        attach dummy nodes to x.left and x.right;
24        Add R in x.right.Cset;
25        Insertion_Fixup(T, x);
26  }
27  if (f = 2^W – 1) Add R in y.right.Cset;
28  else if clause is the same as lines 13-27 with 'e – 1' and 'f' exchanged, '>' and '<'
    exchanged, and 'left' and 'right' exchanged
}
```

Fig. 6. The optimized DST insertion algorithm.

first thing to do in Step 3 is to add R in $y.left.Cset$ if R contains $intvl(y.left)$. This is only possible when e is zero. Otherwise, we add R in the canonical sets of proper nodes in the left subtree of node y, as shown in Lines 5-14. We traverse the nodes ($u_{\rm i}$ for $i = 1$ to m) in the path from y to the node with key $e - 1$, where u_1 is $y.left$, and $u_{\rm m}$ is the node with key $e - 1$. We insert R in $u_{\rm i}$.right.Cset for $i = 1$ to m if $u_{\rm i}.right \neq u_{\rm i+1}$. Step 4 is similar to Step 3.

Assume that the tree in Fig. 2 is a red-black tree right after end point 15 of range $\mathrm{P1} = [0, 15]$ is inserted. P1 will be inserted as follows: The LCA node of end points -1 and 15 is y. The interval $intvl(u)$ is [0, 7] contained in P1 because the start point of P1 is zero. Thus, P1 is stored in $u.Cset$. Since P1 contains $intvl(y.left)$ but not $intvl(v)$, P1 is also stored in $v.left.Cset$.

4.2.3 Complexity

Assume that the size of the canonical set is $O(CSize)$ and inserting a range in a canonical set takes $O(f(CSize))$ time. Inserting an end point in DST takes $O(\log N)$ time plus $O(CSize \times f(CSize))$ time for a constant number of red-black tree rotations, as will be described later. As shown in lines 7-13 in Fig. 5, each iteration of the while loop inserts a range into a canonical set. Thus, the worst-case time complexity is $O(\log N \times f(CSize))$. Furthermore, inserting a range and the two associated end points takes $O((\log N + CSize) \times f(CSize))$ time.

4.3 Optimized Insertion in DST

The insertion algorithm proposed above takes three separate steps sequentially to complete the insertion process. This is its major disadvantage in terms of update speed. In this section, we develop an optimized insertion algorithm that combines these three separate steps into one. In other words, the steps for inserting the two end points of a range R are performed while inserting R. The worst-case execution time of the optimized insertion algorithm is thus only two-thirds that in the unoptimized insertion algorithm. Fig. 6 shows the pseudocode of the optimized insertion algorithm. Since keys $e - 1$ and f may not exist in the tree, we give a different definition for the LCA node of R as follows: Let $\mathcal{G}$ be the set of existing keys contained in range $\mathrm{R} = [e - 1, f]$. If $\mathcal{G}$ is not empty, the LCA node of R is defined to be the LCA of all nodes that contain any key in $\mathcal{G}$. If $\mathcal{G}$ is empty, $e - 1$ and f must be between two successive keys, assuming that -1 and $2^W - 1$ are the smallest and largest keys in the tree, respectively. One of these two successive keys must be in a leaf node. Thus, as shown in lines 1-6 (Step 1) in Fig. 6, two newly created nodes with keys $e - 1$ and f are inserted under this leaf node and rotations to balance the tree may be needed. For example, if range $\mathrm{R} = [1, 2]$ is to be inserted into the segment tree in Fig. 2, two new nodes with keys $e - 1$ and f are inserted under node u.

If the LCA node y for range $\mathrm{R} = [e, f]$ is found, it is possible that R contains the interval of node y. Thus, Step 2 is needed. Step 3 in lines 9-23 inserts end point $e - 1$ and range R in some nodes of the left subtree of node y. In line 10, if e is zero, we immediately add R in $y.left.Cset$. Otherwise, similar operations to that of algorithm DST_Insert_Range in Fig. 5 are performed. The only difference is that we need to make sure whether key $e - 1$ exists in the tree or not. Step 4, which involves the insertion of end point f and range R, is similar to Step 3. The complexity of the optimized insertion algorithm is the same as the unoptimized one.

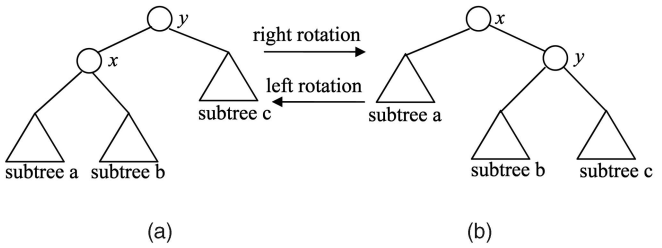

Fig. 7. The DST right and left rotations.

4.4 DST Rotations

Rotations are used to rebalance the search tree after inserting or deleting a key. For DST, changing colors and pointers is the same as that in the original red-black tree and is thus straightforward. However, changing the pointer structure also alters the relative positions of the canonical sets involved in rotations. As a result, the DST range allocation rule may be violated. Therefore, we must adjust the canonical sets involved in rotations to maintain the range allocation rule.

Changing the pointer structure through rotations is a local operation in the tree. Fig. 7 shows the left and right rotations for DST (the same as the red-black tree). The interval covered by any node in subtrees a, b, or c does not change after a left or right rotation. For example, the interval covered by subtree b remains $[x.key + 1, y.key]$ after a rotation. However, the intervals covered by nodes x and y are changed. Before the left rotation around node x is carried out as shown in Fig. 7b, $intvl(x)$ is the union of all intervals covered by subtrees a, b, and c and $intvl(y)$ is the union of all intervals covered by subtrees b and c. After the left rotation, $intvl(x)$ becomes the union of all intervals covered by subtrees b and c and $intvl(y)$ becomes the union of all intervals covered by subtrees a, b, and c. Therefore, we need to reallocate the ranges in the canonical sets of nodes x and y.

We only show the range relocation for left rotations because the range relocation for right rotations is similar. To avoid confusion, we use $Cset$ and $\hat{C}set$ to denote the canonical sets before and after left rotations, respectively. We assume that subtrees a, b, and c are not empty. Similar operations can be taken if either subtree a, b, or c is empty. We perform three steps to reallocate the ranges in the canonical sets of nodes x and y as follows:

1. All ranges stored in $x.Cset$ must be moved to $y.\hat{C}set$ because node y covers the union of subtrees a, b, and c after rotation. Therefore, we perform $y.\hat{C}set = x.Cset$.
2. There might be a range R that simultaneously covers the intervals of subtree a and subtree b. Therefore, after the left rotation, R must be removed from $x.left.Cset$ and $y.left.Cset$ and must be inserted in $x.\hat{C}set$. Therefore, we perform the following:

$$Cset1 = x.left.Cset \cap y.left.Cset,$$
$$x.\hat{C}set = Cset1,$$
$$x.left.\hat{C}set = x.left.Cset - Cset1,$$

and $x.right.\hat{C}set = y.left.Cset - Cset1$.

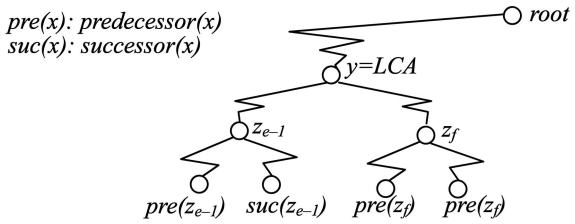

Fig. 8. Illustration for deleting a range $\mathrm{R} = [\mathrm{e}, \mathrm{f}]$.

3. Similarly to Step 2 above, all ranges stored in $y.Cset$ need to be removed and inserted into the canonical sets of the root nodes of subtrees b and c. Therefore, we perform $x.right.\hat{C}set = x.right.\hat{C}set \cup y.Cset$ and $y.right.\hat{C}set = y.right.Cset \cup y.Cset$.

Complexity. Assume that the size of the canonical set is $O(CSize)$ and inserting/deleting a range into/from a canonical set takes $O(f(CSize))$ time. Since there may be $O(CSize)$ ranges that are inserted into or deleted from a canonical set, the time complexity of a rotation is $O(CSize \times f(CSize))$.

4.5 Deletion in DST

Deleting a range $R = [e, f]$ from DST needs the following steps:

1. Remove R from all canonical sets that contain R based on the *range allocation rule*.
2. If e is not zero, determine if the node z with key $e - 1$ needs to be deleted from DST. If so, the following operations must also be performed:
   a. Adjust the canonical sets of some nodes in the subtree rooted at node z.
   b. Rebalance the tree.
3. If f is not $2^W - 1$, this step is the same as Step 2 with keys $e - 1$ and f exchanged.

The process of deleting a range $\mathrm{R} = [e, f]$ is illustrated in Fig. 8. We first find the LCA node y of the two nodes $\mathrm{z_{e-1}}$ and $\mathrm{z_f}$ with keys $e - 1$ and f. Step 1 deletes the range R by traversing the two paths from y to $\mathrm{z_{e-1}}$ and from y to z_{f}, according to the DST range allocation rule. This step is the same as Step 3 for the insertion of a range as shown in Fig. 5. The time complexity of this step is $O(\log N \times f(CSize))$, assuming that the size of the canonical sets is $O(CSize)$ and the time to delete a range from a canonical set is $O(f(CSize))$. Steps 2 and 3 determine if nodes $\mathrm{z_{e-1}}$ and $\mathrm{z_f}$ really need to be removed from DST based on the DST *node deletion rule*, which will be described later.

To delete an end point, we must make sure that this end point is not also the start or finish point of another range. For example, in Fig. 2, when we delete $\mathrm{P6} = [48, 63]$, node z with key 47 cannot be deleted because end point 48 is also the start point of P7. In general, the *DST node deletion rule* is based on Definition 2, which states that the range matching sets EI_i and EI_{i+1} of two consecutive elementary intervals X_{i} and $X_{\mathrm{i+1}}$ are different. Assume that intervals X_{i} and $X_{\mathrm{i+1}}$ are delimited by the key in node z. After a range is removed from DST, if EI_i and EI_{i+1} become the same, node z should be removed from DST based on the *node deletion rule*.

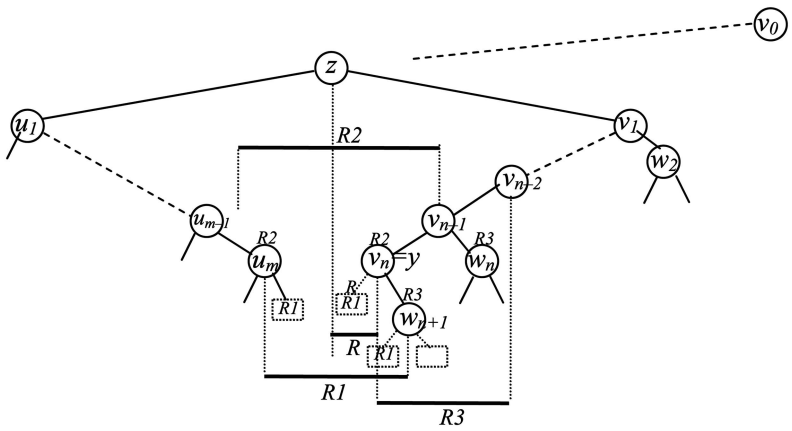

Fig. 9. Replace node z to be deleted by z's successor, $v_{\rm n}$. Node $v_0 = {\rm successor}(z_{max})$, where z_{max} is the node with the largest key in the subtree rooted at z.

To determine if $\mathrm{EI}_i = \mathrm{EI}_{i+1}$, we have to compute the canonical sets whose ranges contain intervals $X_{\rm i}$ and $X_{\rm i+1}$. Since the ranges in the canonical sets of node z's ancestors contain both $X_{\rm i}$ and $X_{\rm i+1}$ and thus belong to both EI_i and EI_{i+1}, they can be ignored when determining if $\mathrm{EI}_i = \mathrm{EI}_{i+1}$. As a result, EI_i and EI_{i+1} can be computed as follows (also see Fig. 9):

$$\mathrm{EI}_i = z.left.Cset \cup u_1.right.Cset \cup \ldots \cup u_{\rm m}.right.Cset, \quad (1)$$

$$\mathrm{EI}_{i+1} = z.right.Cset \cup v_1.left.Cset \cup \ldots \cup v_{\rm n}.left.Cset, \quad (2)$$

where u_j for $\mathrm{j} = 1$ to m and v_k for $\mathrm{k} = 1$ to n are the descendant nodes of z in the paths from z to $u_m = predecessor(z)$ and from z to $v_n = successor(z)$, respectively.

Definition 5 (DST node deletion rule). *Node z must be deleted if and only if $\mathrm{EI}_i = \mathrm{EI}_{i+1}$, where EI_i and EI_{i+1} are computed based on (1) and (2).*

For example, in Fig. 2, interval X_6 is contained in intervals $intvl(w)$, $intvl(z)$, and $intvl(q)$. Thus, the range matching set EI_6 of X_6 is {P2}, which is the union of $w.Cset$, $z.Cset$, $q.Cset$, and $q.left.Cset$. Similarly, EI_5 is {P2, P5}, which is the union of $w.Cset$, $y.Cset$, $v.Cset$, and $v.right.Cset$. If P5 is deleted, node w also has to be deleted.

Now, we describe how to remove a node z from DST. As in the deletion algorithm for the binary search tree described in [4], three cases are considered when deleting node z from DST: 1) z is a leaf node in DST, 2) z is a node of degree one, and 3) z is a node of degree two.

Case 1: Delete node z that is a leaf node. Deleting a leaf node z needs the following steps:

1.1. Reset node z as a leaf node.
1.2. Remove the two leaf nodes of node z.
1.3. Perform a red-black tree fix-up operation [4] from node $parent(z)$ up to the root.

Notice that node z is a regular node and not a leaf node. The first thing that should be done is to reset node z as a leaf node. Without loss of generality, we assume that z is the right child of its parent, that is, $z = \mathrm{z}.parent.right$. Node r with key 39 in Fig. 2 is an example. According to the DST *node deletion rule*, node z must be deleted if $z.left.Cset = z.right.Cset$ after a range R is removed from the tree. Range R must be stored either in $z.left.Cset$ or $z.right.Cset$, but not in both before R is removed. Also, according to the DST *range allocation rule*, it is impossible if there exists another range R1 that is stored in $z.left.Cset$ or $z.right.Cset$. As a result, we have $z.left.Cset = \phi$ and $z.right.Cset = \{\mathrm{R}\}$ or $z.left.Cset = \{\mathrm{R}\}$ and $z.right.Cset = \phi$ before range R is deleted. However, it is possible that some ranges may be stored in $z.Cset$ (that is, $z.Cset \neq \phi$). The ranges stored in $z.Cset$ contain the interval $intvl(z)$. Thus, after deleting node z, the ranges stored in $z.Cset$ are not moved by resetting node z as a leaf node (Step 1.1). Then, the two leaf children of node z are deleted (Step 1.2).

The last thing that should be done is to rebalance the tree (Step 1.3). Regardless of the use of either the AVL or the red-black tree, standard balancing operations can be used. As stated earlier, only the canonical sets of some nodes may need to be adjusted.

Case 2: Delete a node z of degree 1. We further break down this case into four subcases. Let z_p and z_c be the parent and child of node z, respectively. The first subcase deals with the condition when z is the right child of z_p and z_c is z's right child, that is, $z = \mathrm{z}.parent.right$ and $z.left$ is the leaf. The other three subcases, $z = \mathrm{z}.parent.left$ and $z.right$ is leaf, $z = z.parent.right$ and $z.right$ is the leaf, and $z = z.parent.left$ and $z.left$ is the leaf, can be considered similarly.

We first prove that node z must be black. Assume that node z is red based on the property of the red-black tree that dictates that a red node cannot have a red child. Therefore, node $z.right$ must be black and the number of regular black nodes from node z to z's left leaf child is zero. However, the number of regular black nodes from node z to z.right is at least one. This violates the red-black tree property that every simple path from a node to a descendant leaf contains the same number of black nodes. For the same reason, we can prove that $z.right$ is red and $z.right$ is a regular leaf node. We first summarize the process of deleting node z as follows and give the details afterward:

2.1. $z_c.Cset = z.Cset$.
2.2. $z.parent.right = z.right$.
2.3. Change the color of node $\mathrm{z}.right$ to black.
2.4. Delete node z and leaf node $z.left$ from DST.

TABLE 4
Four Cases for Deleting a Node with One Child

	Before deleting any range			After deleting range R1 or R2		
Case	z.left.Cset	z_c.Cset	z_c.left.Cset	z.left.Cset	z_c.Cset	z_c.left.Cset
1	{R2}	ϕ	{R1,R2}	{R2}	ϕ	{R2}
2	{R2}	ϕ	ϕ	ϕ	ϕ	ϕ
3	ϕ	{R2}	ϕ	ϕ	ϕ	ϕ
4	ϕ	ϕ	{R1}	ϕ	ϕ	ϕ

Based on the DST Cset properties that were described earlier in this section, the following conditions must be true: $z.left.Cset \cap z_c.Cset = \phi$ and $z_c.Cset \cap z_c.left.Cset = \phi$. According to the DST node deletion rule, the third condition, $z.left.Cset \neq z_c.Cset \cup z_c.left.Cset$ before any range is deleted or $z.left.Cset = z_c.Cset \cup z_c.left.Cset$ after a range is deleted, must also be true. Based on these three conditions, four cases in Table 4 are identified as follows: Assume that the union of $z.left.Cset$ and $z_c.Cset$ consists of two or more ranges before any range is deleted, like $z.left.Cset \cup z_c.Cset = \{\mathrm{Ra}, \mathrm{Rb}\}$, for instance. This assumption results in the key of node z being the common end point of two or more ranges. Thus, no matter which range (Ra or Rb) is deleted, it is impossible that

$$z.left.Cset = z_c.Cset \cup z_c.left.Cset.$$

Therefore, if we want to have a node deleted after a range is removed from DST, we must have $z.left.Cset = \{\mathrm{R2}\}$, $z_c.Cset = \phi$, $z.left.Cset = \phi$, and $z_c.Cset = \{\mathrm{R2}\}$ or $z.left.Cset = \phi$ and $z_c.Cset = \phi$. The canonical set $z_c.left.Cset$ can be determined accordingly. As a result, if any one of the four cases in Table 4 happens, node z must be deleted when range R1 or R2 is removed from DST.

Consider Case 1 in Table 4 in which $z.left.Cset = \{\mathrm{R2}\}$, $x.Cset = \phi$, and $x.left.Cset = \{\mathrm{R1}, \mathrm{R2}\}$. Deleting range R1 results in $z.left.Cset = z_c.Cset \cup z_c.left.Cset$ and, thus, node z can be removed from DST. If node z is deleted, the interval associated with node z_c is expanded from $[z.key + 1, z_c.key]$ to $[z_p.key + 1, z_c.key]$. Thus, the remaining range R2 in both $z.left.Cset$ and $z_c.left.Cset$ is kept in $z_c.left.Cset$ and node z. No matter which case we are considering, $x.\hat{C}set$ is always empty after the deletion of range R1 or R2. Thus, Step 2.1 performing $z_c.Cset = z.Cset$ reinforces the range allocation rule because the start end point of the ranges in z.$Cset$ remains $z_p.key + 1$. Since changing the color of the node to black maintains all of the properties of the red-black tree, no other operation is needed. Finally, node z and the leaf node $z.left$ can be deleted immediately.

Case 3: Delete node z of degree 2. To delete a node z of degree two, we first replace $z.key$ with $y.key$ and then delete y, where y is either z's successor or z's predecessor in the tree. If $y = successor(z)$, y must have no left child. Similarly, if $y = predecessor(z)$, y must have no right child. Node y must be of degree one or zero. We shall describe how to select z's successor or z's predecessor in order to reduce the total overhead of deleting node y later. Currently, we assume that $successor(z)$ is selected to replace z. To facilitate the analysis, we further break down this case into two subcases: y has one child and y is a leaf node. We only give the detailed analysis for the former because the analysis for both is similar. Fig. 9 illustrates the general case for which node y (v_n) has a right child $x = w_{n+1}$ and range R is deleted. We first summarize the deletion process as follows and give the details afterward:

3.1. $z.key = \mathrm{y}.key$.
3.2. v_{n-1}.left $= w_{n+1}$ (z.$right = w_1$ if $n = 1$).
3.3. $w_{n+1}.Cset = w_{n+1}.Cset \cup v_n.Cset$.
3.4. For each range $R = [e, f]$ in $w_{n+1}.Cset$, $m = k$ if v_k.key $\leq f < v_{k-1}$.key for $0 < k \leq n-2$ or $m = 0$ if v_0.key $\leq f$, we perform the following operations:
   a. Delete R from w_i.Cset, for i = n + 1 to m + 2.
   b. Add R in v_{m+1}.Cset.
3.5. Remove node y and y's left leaf child.
3.6. Perform the red-black tree fix-up operation [4] from node w_{n+1} up to the root.

One special case is when n = 1, that is, $y = \mathrm{z}.right$. In this case, only Step 3.2 needs to be changed. The change is shown in the parentheses in Step 3.2. Therefore, we assume that n > 1 and, thus, $y \neq \mathrm{z}.right$ hereafter.

The first two steps are straightforward. According to the DST *node deletion rule* in Definition 5, if node y must be deleted from DST, any range in v_n.left.Cset or v_i.Cset for i = 1 to n must also be stored in u_m.right.Cset or u_i.Cset for i = 1 to m after a range is deleted. For example, R1 in y.left.Cset is also in u_m.right.Cset. Therefore, the canonical set $y.left.Cset$ can be deleted directly. Also, the ranges in v_n.Cset (for example, R2 in Fig. 9) must be moved to $\mathrm{w}_{\mathrm{n+1}}$.Cset, as shown in Step 3.3.

There may exist a range that belongs to both w_{n+1}.Cset and w_n.Cset. $R3 = [v_n.\mathrm{key} + 1, v_{n-2}.\mathrm{key}]$ in Fig. 9 is an example. Thus, R3 should be deleted from both w_{n+1}.Cset and w_n.Cset and inserted into v_{n-1}.Cset after node y is deleted. The general case is shown in Step 3.4. Finally, node y and y's left leaf node are deleted and a red-black tree fix-up operation is needed.

Consider the DST shown in Fig. 2. Assume range P5 is deleted. Node w with key 23 must be deleted because the range matching set of elementary interval X_5 is the same as that of X_6, which is {P2}. To delete node w, Steps 3.1 and 3.2 perform the operations of $w.key = q.key$ and $z.left = r$. Since $q.Cset$ is empty, Step 3.3 does nothing. In step S.4, we have to remove range P4 from $r.Cset$ and $g.Cset$ and put it in $z.Cset$ because $\mathrm{P4} = [32, 63]$. Finally, the deletion of nodes q and $q.left$ and the red-black tree fix-up operation that will change the color of node r to black will complete the process of deleting node w.

Complexity. We use the same assumptions in DST insertion. The overall complexity of the deletion process is equal to that of deleting a node of degree two in Case 3 because, in Step 3.4, the ranges may be removed from the canonical sets of some nodes and be added in other nodes. Thus, the overall complexity of DST deletion is $O(\log N \times Max \times f(Max))$ because there are $O(Max)$ ranges in $w_{n+1}.Cset$ and each range in $w_{n+1}.Cset$ may be inserted/deleted into/from w_i.Cset for i = n − 1 to 1, where n is $O(\log N)$ and inserting/deleting a range into/from a canonical set takes $O(f(Max))$ time.

4.5.1 Cost of Selecting Node y to Replace Node z

Based on the proposed DST, changing the colors and pointers of the nodes is a simple operation since it does not involve any complicated data structure. Two complicated operations include the rotations and the canonical set adjustment in Step 3.3. These two operations involve searching, inserting, and deleting ranges in and from the canonical sets. The data structures that can be used for canonical sets will be discussed later.

In Section 4.3, we discussed the three steps needed for left rotations. In the second step of left rotations, we have to delete the common ranges from $x.left.Cset$ and $y.left.Cset$ and insert them in $x.\hat{C}set$, which is the canonical set of node x after a rotation. This will translate into the following *canonical set operations*: The first operation is to compute $I_{intersect} = x.left.Cset \cap y.left.Cset$, which is the intersection of the two canonical sets $x.left.Cset$ and $y.left.Cset$. The next two operations are $x.left.Cset = x.left.Cset - I_{intersect}$ and $y.left.Cset = y.left.Cset - I_{intersect}$. The last operation computes $y.\hat{C}set = x.Cset + I_{intersect}$. In the third step of left rotations, we move all of the ranges in $y.Cset$ to both $x.right.\hat{C}set$ and $y.right.\hat{C}set$. This will translate into the following canonical set operations:

$$x.right.\hat{C}set = x.right.Cset + y.Cset$$

and

$$y.right.\hat{C}set = y.right.Cset - y.Cset.$$

To simplify the process of analyzing the cost of replacing z with y, we assume that one rotation needs six canonical set operations (four for Step 2 and two for Step 3). Based on the red-black tree, a deletion requires at most three rotations if the deleted node is black. Therefore, we assume that 18 canonical set operations are needed for a red-black tree deletion.

For the canonical set adjustment in Step 3.3 as described above, the cost is calculated as follows: Assume that range $\mathrm{R_j} = [e_j, f_j]$ in w_{n+1}.Cset and $v_{kj}.key \le f_j < v_{kj-1}.key$ for $0 < k_j \le n-2$ or $v_{kj}.key \le f_j$ if $j = 0$. Therefore, we need $n - k_j - 1$ canonical set deletions from $w_i.Cset$ for $i = n$ to $k_j + 2$ and one canonical set addition in $v_{kj+1}.Cset$. Therefore, $n - k_j$ canonical set operations are needed for $\mathrm{R_j}$. Assume that there are l such ranges ($\mathrm{R_1}$ to R_l) in $w_{n+1}.Cset$. In total, we need $l \times n - \Sigma^l k_j$ canonical set operations. The operation of determining a range $\mathrm{R_j} \in w_{n+1}$.Cset such that v_{kj}.key $\le f_j < v_{kj-1}.key$ for $0 < k_j \le n-2$ or $v_{kj}.key \le f_j$ if $j = 0$ can be computed in a very efficient way as follows: Please refer to Fig. 9. When we perform the node deletion rule by traversing the tree from the root to $predecessor(z)$ and $successor(z)$, we can record the keys and pointers of nodes v_0 to v_n in two separate lists, namely, *klist* and *plist*. Then, a simple binary search on the list *klist* with value f_j can determine whether a range $\mathrm{R_j} = [e_j, f_j]$ in $w_{s+1}.Cset$ satisfies $v_{kj}.key \le f_j < v_{kj-1}$.key for $0 < k_j \le n-2$ or $v_{kj}.key \le f_j$ for $k_j = 0$. With this efficient method, the execution time for determining whether or not the range $\mathrm{R_j} = [e_j, f_j]$ in $w_{s+1}.Cset$ needs to be moved or can be ignored.

Finally, we have to consider the rotations needed when $z.key$ is replaced by the key of $\mathrm{predecessor}(z)$ or $\mathrm{successor}(z)$, as in Steps 1.3 and 3.6, required in the red-black tree deletion fix-up operations. Assume that node y is replaced by $successor(z)$. If $successor(z)$ has a child w_{n+1}, $successor(z)$ must be black, and w_{n+1} must be red before deleting node z, as based on the definition of red-black trees. When node $successor(z)$ is removed from DST, node w_{n+1} can be set to black. As a result, all red-black tree properties are maintained and, thus, no further rotation is needed. Now, consider that $successor(z)$ has no child. If $successor(z)$ is red, no rotation is needed when $\mathrm{successor}(z)$ is deleted. However, if $successor(z)$ is black, we have to follow the red-black tree deletion fix-up operation to rebalance the tree. This tree deletion fix-up operation needs three rotations at most, which account for 18 canonical set operations.

Now, we summarize the above analysis and come up with the *replacing node selection rule* as follows: *If the number of canonical set operations needed for replacing node z with $successor(z)$ is less than or equal to that needed for replacing z with $predecessor(z)$, we select $successor(z)$ to replace z. Otherwise, we select $predecessor(z)$ to replace z.*

4.6 Data Structure for Canonical Sets

As stated earlier, the most complicated operations in DST are the ones that act on canonical sets. In this section, we develop efficient data structures for organizing ranges in canonical sets based on different types of ranges: arbitrary and prefix ranges. Before describing the details, we give the following assumptions first: Each range is assigned a priority. In general, a range R1 is assigned a higher priority than R2 if R1 is more specific than R2. Otherwise, the priorities of ranges R1 and R2 can be assigned randomly.

As shown in Fig. 3, the proposed search algorithm finds matches for address d by performing a simple DST traversal from the root to the leaf node that corresponds to the elementary interval that contains d. All ranges in the canonical sets of the traversed nodes match d. If the application using the search requires all of the matched ranges, then the process is completed. Now, we assume that the application needs the highest priority range. If the range set consists of arbitrary ranges, the proposed search algorithm needs to search all of the canonical sets traversed. If the range set is conflict-free, only the last visited nonempty canonical set needs to be searched. In BOB [14], regardless of the range sets being arbitrary, conflict-free, or prefix ranges, not all ranges encountered in the searching path match d. Therefore, extra operations are needed to determine the matching ranges of d. This is why the proposed search algorithm is faster than BOB, as we shall see in the performance evaluation.

4.6.1 A Combination of the Priority Queue and the Thin Interval Tree for Arbitrary Ranges

No matter whether the range set is arbitrary or conflict-free, it is important to have an efficient data structure to find the highest priority range in a range set, as well as to insert/delete a range into/from a range set. The priority queue provides a partial solution because it is only good for search and insertion. To achieve a fast deletion performance, we

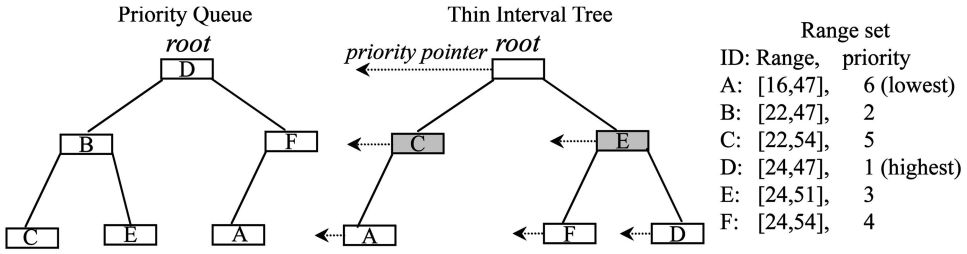

Fig. 10. Canonical set data structure for arbitrary ranges.

propose a data structure that combines the priority queue and the thin interval tree [4]. Fig. 10 shows an example for a canonical set consisting of six additional ranges that are added in the segment tree in Fig. 2. These six ranges are stored in the canonical set of node q that is associated with interval [24, 47]. Obviously, finding the highest priority range from the root of the priority queue takes a constant time. The thin interval tree facilitates the fast implementation of a range's deletion from the canonical set. Each node in the thin interval tree is augmented with a *priority pointer* pointing to the corresponding node in the priority queue. The process of deleting a range works as follows: Let R be the range to be deleted. We first delete R from the thin interval tree. Then, by following the priority pointer in R, the corresponding node in the priority queue can be deleted accordingly. Inserting a new range takes similar actions. As a result, inserting/deleting a range into/from the thin interval tree and the priority tree takes $O(\log CSize)$ time, where $CSize$ is the size of the canonical set.

4.6.2 A Combination of the Bit Vector and the Linear List for Prefix Ranges

If prefixes are considered, our experiments show that the best data structure for the range set having the fastest search performance is the combination of a bit vector and a linear list. The bit vector stores the lengths of the prefixes in the canonical set and the linear list stores their next hop ports. Since the maximum number of prefixes in a real routing table that cover any address is six, it is practical to store prefixes in a canonical set in a linear list. Searching the longest prefix in the bit vector of a canonical set takes a constant time, but inserting/deleting a prefix into/from the linear list of a canonical set takes $O(W)$ time.

5 Experimental Results

In this section, we present the experimental results of the proposed DST and other prominent dynamic schemes in terms of memory requirement, search, insertion, and deletion times. We implemented all tested schemes in C and measured their performance based on five IPv4 routing tables obtained from [2] and [17]. Table 5 shows the detailed statistics of these five routing tables. The first three routing tables having the same AS number are obtained from the same Border Gateway Protocol (BGP) backbone router [17] at different times and the other two tables are from [2]. The experiments were run on a 2.4 GHz Pentium IV PC with 8 Kbyte L1, 256 Kbyte L2 caches, and 512 Mbyte main memory. The gcc-3.2.2 compiler of Redhat 9.0 with an optimization level of -O4 was used. The instruction called ReaD Time Stamp Counter (RDTSC) is also used to keep an accurate count of every clock cycle that occurs in the processor.

As we can see in Table 5, the number of distinct end points $|\mathrm{E}|_{\mathrm{e,f}}$ using the traditional end point scheme is almost twice the number of prefixes. The number of distinct end points $|\mathrm{E}|_{\mathrm{e-1,f}}$ using the minus-1-end point scheme is only about 1.36 times the number of prefixes in average. In our experiments, both DST and PBOB are implemented with red-black trees and PIBT and MRT are 32-way B-tree data structures. Table 6 and Fig. 11a show the memory requirements for all tested schemes. As expected, PBOB consumes the least memory because each prefix is stored only once in the tree, whereas DST, MRT, and PIBT may store each prefix in many nodes. MRT and PIBT consume more memory than PBOB and DST because their node sizes are larger than that in PBOB and DST and the keys and child pointers of the 32-way B-trees used in MRT and PIBT may be underutilized.

To measure the search times, we randomized the start addresses of all prefixes in the original routing table as the input IP traffic in the simulation. The average search times from the input IP traffic are illustrated in Table 7 and Fig. 11b. As we can see, PBOB performs much worse than the other three schemes. DST is only a little slower than MRT and PIBT. The search time of DST is about 50-60 percent of that of PBOB. MRT and PIBT have the best search times because the heights of B-trees in MRT and PIBT are smaller than that of the binary search trees in PBOB and DST. Since the number of distinct end points $|\mathrm{E}|_{\mathrm{e-1,f}}$ using the minus-1-end point scheme is only

TABLE 5
Analyses of Five BGP Routing Tables Obtained from [2] and [17]

Table	AS6447a	AS6447b	AS6447c	AS7660	AS2493
Year-Month	2000-4	2002-4	2005-4	2005-4	2005-4
# of prefixes	79,560	124,824	163,574	159,816	157,118
$\|E\|_{e-1,f}$	112,975	172,279	219,038	210,395	207,008
$\|E\|_{e,f}$	154,743	241,329	314,637	306,679	301,914

TABLE 6
The Memory Requirement (Kbytes)

Scheme	AS6447a	AS6447b	AS6447c	AS7660	AS2493
DST	2220	3385	4304	4134	4068
PBOB	1525	2374	3101	3013	2966
PIBT	4957	7732	10081	9826	9673
MRT	4815	7511	9792	9545	9396

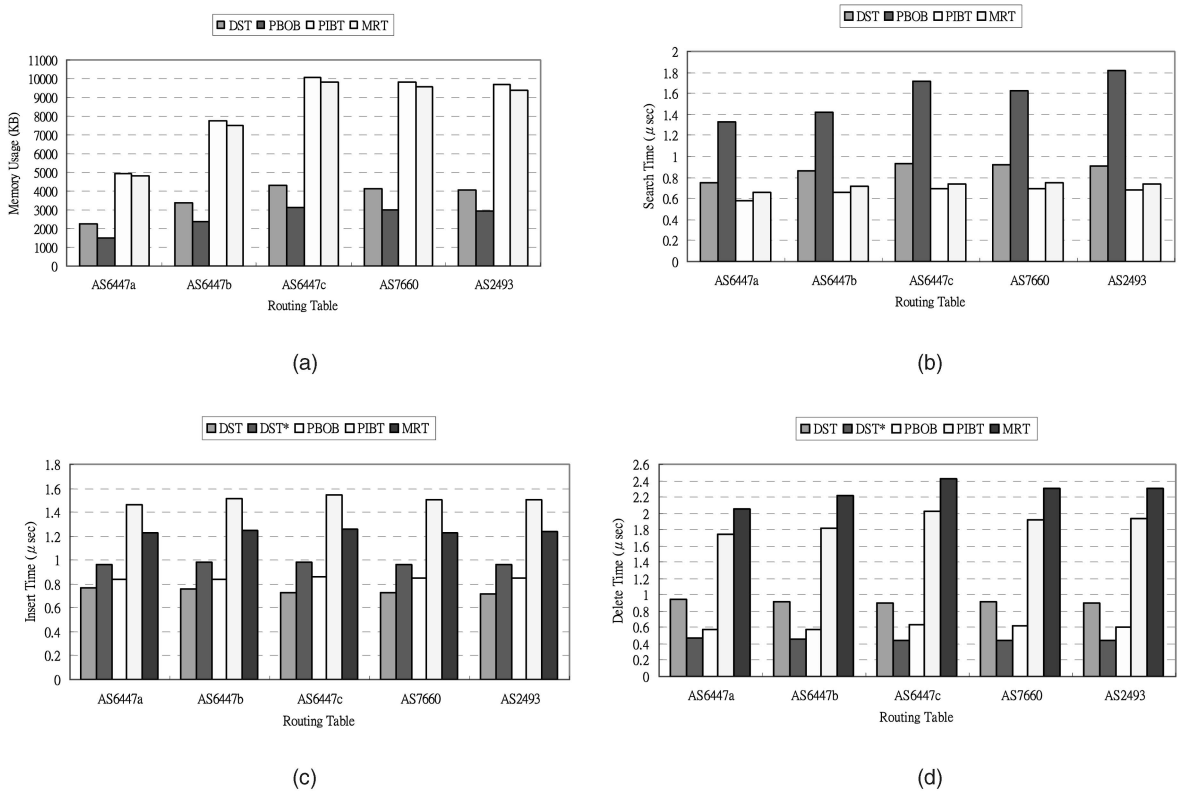

Fig. 11. Performance comparisons.

about 1.36 times the number of prefixes, the heights of DST ($\log(|E|_{e-1,f})$) and PBOB (log(number of prefixes)) are approximately the same. Thus, the reduction of the number of end points in the tree is only a secondary factor that improves the DST search speed. The primary factor is as follows: Both DST and PBOB augment the tree nodes with the canonical sets and range sets (that is, $range()$ in [14]), respectively. For simplicity, we refer to the canonical sets and range sets as the *cover sets* when no confusion is incurred. DST and PBOB are similar in that the cover sets of the nodes along the search path need to be examined to find the longest prefix match. However, the search speeds of DST and PBOB are different in how the cover sets are examined. Since PBOB does not guarantee that all of the prefixes in the cover sets of the nodes along the search path match the target IP address, it has to examine every prefix in the cover sets encountered one by one to determine if the target IP address is matched. For DST, two properties of the cover sets are guaranteed. First, all prefixes in the cover sets of the nodes encountered along the search path must match the target IP address. Second, the prefixes in the cover set of the node in the lower tree level must be more specific than that in the cover set of the node in the higher level. Therefore, for DST, the longest prefix match is immediately available in the last nonempty cover set in the search path. The time-consuming operations required for PBOB to examine the prefixes in the cover sets are not needed. Thus, DST has a faster speed than PBOB.

TABLE 7
The Performance of Various Dynamic Schemes

Routing Table		AS6447a	AS6447b	AS6447c	AS7660	AS2493
Search (μsec)	DST	0.75	0.86	0.93	0.92	0.91
	PBOB	1.33	1.42	1.72	1.62	1.82
	PIBT	0.58	0.66	0.69	0.69	0.68
	MRT	0.66	0.72	0.74	0.75	0.74
Insert (μsec)	DST	0.77	0.76	0.73	0.73	0.72
	DST*	0.96	0.98	0.98	0.96	0.96
	PBOB	0.84	0.84	0.86	0.85	0.85
	PIBT	1.46	1.51	1.54	1.50	1.50
	MRT	1.23	1.25	1.26	1.23	1.24
Delete (μsec)	DST	0.94	0.92	0.90	0.91	0.90
	DST*	0.48	0.46	0.45	0.45	0.44
	PBOB	0.57	0.57	0.64	0.62	0.60
	PIBT	1.75	1.81	2.02	1.92	1.94
	MRT	2.05	2.21	2.43	2.31	2.30

To measure the update speeds, we first construct the data structure for each scheme according to the original routing table. Next, we randomly delete 5 percent of the prefixes from the structure we built to obtain the deletion times. Then, we randomly insert these deleted prefixes back into the structure to obtain the insertion times. We illustrate the average insertion and deletion times in Table 7 and Figs. 11c and 11d.

At first glance, we may think that PBOB will perform better than DST in terms of insertion speed because 1) the DST insertion involves three separate steps for inserting two end points and one range into the segment tree and 2) PBOB only needs to insert the range into the node such that the range covers the key of the node or creates a new node in the binary tree. However, based on our experiments, we shall show that DST actually performs a little better than PBOB. First, as stated in the definition of the minus-1-end point scheme, numerous end points of the newly inserted prefixes are shared with the existing prefixes and thus already exist in DST before they are inserted. Therefore, DST generates a lesser number of new nodes

TABLE 8
The Numbers of Newly Created Nodes in PBOB and DST after Inserting the Randomly Selected 5 Percent of the Original Prefixes

Scheme	AS6447a	AS6447b	AS6447c	AS7660	AS2493
# of prefixes	3978	6241	8178	7990	7855
PBOB	3617	5554	7251	7026	6927
DST	3504	4950	6070	5653	5462

(end points) than PBOB, as shown in Table 8, when inserting the randomly selected 5 percent of the original prefixes. Second, the optimized DST insertion algorithm described in Section 4.3 can insert a prefix and the one or two nonexisting end points associated with the prefix simultaneously. Third, PBOB employs a size constraint in the interval tree. Only the nodes of degree 0 or 1 that have an empty range set can be deleted. To find the degree 0 and degree 1 nodes that have an empty range set efficiently, one doubly linked list of these nodes must be maintained. The other doubly linked list of degree 2 nodes that have an empty range set is also needed. PBOB must maintain these two linked lists because it is not guaranteed that a node that can be deleted at one time can be deleted after some nodes are inserted or deleted at another time. For example, when a range is inserted/deleted in/from the binary search tree in PBOB or, when a rotation is performed, nodes may be added in or removed from these doubly linked lists and nodes may move from one list to another. The operations involved in these two doubly linked lists slow down the insertion speed of PBOB. DST does not need to maintain any linked list. Therefore, with these three factors, the insertion speed of DST is better than PBOB. However, the DST deletion is much slower than PBOB because DST has to delete the nodes (called unneeded nodes) containing the end points that do not belong to any prefixes stored in DST. These unneeded nodes can be of any degree. In contrast, PBOB only deletes the nodes of degree 0 and 1 with empty range sets.

To improve the deletion speed of DST, we also conducted experiments by using the idea of the size constraint in PBOB [14]. In other words, the DST deletion process only removes the prefixes stored in the canonical sets and does not delete any unneeded nodes when the number of nodes in DST is not greater than $2N$, where N is the number of prefixes stored in DST. We denote this improved DST scheme by DST*, as shown in Table 7 and Fig. 11. DST* only needs to maintain one doubly linked list of pointers of these unneeded nodes. When the size constraint is violated, we need to delete as many nodes pointed to by the pointers in the doubly linked list to restore the size constraint. As we can see, the deletion speed of DST* is considerably improved. However, the DST insertion is a little slowed down. The reason is as follows: If the newly inserted end point has already been stored in a DST* node P and the pointer of P is also in the doubly linked list, then DST* has to remove the pointer of P from the doubly linked list to complete the insertion process. Notice that, differently from PBOB, tree rotations in DST* incur no operation on the doubly linked list since the canonical set adjustments from rotations will not violate the DST node deletion rule.

6 Conclusion

We have developed a new data structure called DST that is suitable for dynamic range sets. DST is a balanced binary search tree that is built from the distinct end points of ranges based on a novel minus-1-end point scheme. We also proposed a data structure that combines the priority queue and the thin interval tree [4] to efficiently access the ranges stored in the canonical set of a DST node. The experiments employing real IPv4 routing tables showed that the DST performs much better than PBOB and a little worse than the B-tree-based dynamic schemes (MRT and PIBT) in terms of search speed. The memory consumption and update speed of DST are substantially better than those of MRT and PIBT and comparable with PBOB.

Aknowledgments

The authors would like to express their sincere thanks to the editors and the reviewers who gave very insightful and encouraging comments. This work was supported in part by the National Science Council, Republic of China, under Grant NSC-94-2213-E-006-026.

References

[1] M.D. Berg, M.V. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer, 1997.
[2] BGP Routing Table Analysis Reports, http://bgp.potaroo.net/, 2006.
[3] H. Chao, "Next Generation Routers," *Proc. IEEE,* vol. 90, no. 9, pp. 1518-1558, Sept. 2002.
[4] T. Cormen, C. Lieserson, R. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. MIT Press, 2001.
[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '97),* pp. 3-14, Sept. 1997.
[6] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. IEEE INFOCOM '00,* vol. 3, pp. 1193-1202, Mar. 2000.
[7] S. Fuller, T. Li, J. Yu, and K. Varadhan, *Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy,* IETF RFC 1519, Sept. 1993.
[8] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM '98,* pp. 1240-1247, Apr. 1998.
[9] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network,* special issue, vol. 15, no. 2, pp. 24-32, Mar./Apr. 2001.
[10] C. Labovitz, G. Malan, and F. Jahabnian, "Internet Routing Instability," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '97),* pp. 115-126, Sept. 1997.
[11] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking,* pp. 324-334, June 1999.
[12] H. Lu, K. Kim, and S. Sahni, "Prefix and Interval-Partitioned Dynamic IP Router-Tables," *IEEE Trans. Computers,* vol. 54, no. 5, pp. 545-557, May 2005.
[13] H. Lu and S. Sahni, "$O(\log n)$ Dynamic Router-Tables for Prefixes and Ranges," *IEEE Trans. Computers,* vol. 53, no. 10, pp. 1217-1230, Oct. 2004.
[14] H. Lu and S. Sahni, "Enhanced Interval Trees for Dynamic IP Router-Tables," *IEEE Trans. Computers,* vol. 53, no. 12, pp. 1615-1628, Dec. 2004.

[15] H. Lu and S. Sahni, "A B-Tree Dynamic Router-Table Design," *IEEE Trans. Computers*, vol. 54, no. 7, pp. 813-824, July 2005.
[16] E. McCreight, "Priority Search Trees," *SIAM J. Computing*, vol. 14, no. 1, pp. 257-276, 1985.
[17] D. Meyer Univ. of Oregon Route Views Archive Project, http://archive.routeviews.org/, 2006.
[18] D. Morrison, "PATRICIA—Practical Algorithm to Retrieve Information Coded in Alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514-534, Oct. 1968.
[19] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Trie," *IEEE J. Selected Areas in Comm.*, vol. 17, no. 6, pp. 1083-1092, June 1999.
[20] C. Partridge et al., "A 50-Gb/s IP Router," *IEEE/ACM Trans. Networking*, vol. 6, no. 3, pp. 237-248, June 1998.
[21] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.
[22] S. Sahni and K. Kim, "An $O(\log n)$ Dynamic Router-Table Design," *IEEE Trans. Computers*, vol. 53, no. 3, pp. 351-363, Mar. 2004.
[23] K. Sklower, "A Tree-Based Packet Routing Table for Berkeley UNIX," technical report, Univ. of California, Berkeley, 1993.
[24] V. Srinivasan and G. Varghese, "Fast IP Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems*, pp. 1-40, Feb. 1999.
[25] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High-Speed IP Routing Lookups," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '97)*, pp. 25-36, Sept. 1997.
[26] P. Warkhede, S. Suri, and G. Varghese, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," *Computer Networks*, vol. 44, no. 3, pp. 289-303, Feb. 2004.
[27] P. Gupta and N. McKeown, "Dynamic Algorithms with Worst-Case Performance for Packet Classification," *Proc. IFIP-TC6/European Commission Int'l Conf. Broadband Comm., High Performance Networking, and Performance of Comm. Networks (NETWORKING '00)*, 2000.

Yeim-Kuan Chang received the MS degree in computer science from the University of Houston, Clear Lake, in 1990 and the PhD degree in computer science from Texas A&M University, College Station, Texas, in 1995. He is currently an assistant professor in the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan, Republic of China. His research interests include computer architecture, multiprocessor systems, Internet router design, and computer networking.

Yung-Chieh Lin received the MS degree in computer science and information engineering from National Cheng Kung University, Taiwan, Republic of China, in 2005. He is currently working toward the PhD degree in computer science and information engineering at National Cheng Kung University, Taiwan, Republic of China. His current research interests include high-speed networks and high-performance Internet router design.