Performance Analysis of BSTs in System Software*

Ben Pfaff
Stanford University
Department of Computer Science
blp@cs.stanford.edu

Abstract

Binary search tree (BST) based data structures, such as AVL trees, red-black trees, and splay trees, are often used in system software, such as operating system kernels. Choosing the right kind of tree can impact performance significantly, but the literature offers few empirical studies for guidance. We compare 20 BST variants using three experiments in real-world scenarios with real and artificial workloads. The results indicate that when input is expected to be randomly ordered with occasional runs of sorted order, red-black trees are preferred; when insertions often occur in sorted order, AVL trees excel for later random access, whereas splay trees perform best for later sequential or clustered access. For node representations, use of parent pointers is shown to be the fastest choice, with threaded nodes a close second choice that saves memory; nodes without parent pointers or threads suffer when traversal and modification are combined; maintaining a in-order doubly linked list is advantageous when traversal is very common; and right-threaded nodes perform poorly.

1 Introduction

OS kernels and other system software often use binary search tree (BST) based data structures such as AVL trees, red-black trees, or splay trees. Choosing the right tree and node representation can impact the performance of code that uses these data structures. Surprisingly, there has been little empirical study of the relationship between the algorithms used for managing BST-based data structures and performance characteristics in real systems [1, 2, 3]. This paper attempts to fill this gap by thoroughly analyzing the performance of 20 different variants of binary search trees under real and artificial workloads.

We present three experiments. The first two examine data structures used in kernels in memory management and networking, and the third analyzes a part of a source code cross-referencing tool. In each case, some test workloads are drawn from real-world situations, and some reflect worst- and best-case input order for BSTs.

We compare four variants on the BST data structure: unbalanced BSTs, AVL trees, red-black trees, and splay trees. The results show that each should be preferred in a different situation. Unbalanced BSTs are best when randomly ordered input can be relied upon; if random ordering is the norm but occasional runs of sorted order are expected, then red-black trees should be chosen. On the other hand, if insertions often occur in a sorted order, AVL trees excel when later accesses tend to be random, and splay trees perform best when later accesses are sequential or clustered.

For each BST data structure variant, we compare five different node representations: plain, with parent pointers, threaded, right-threaded, and with an in-order linked list. Combining traversal and modification in a BST with plain nodes requires extra searches, although there are fewer fields to update. Parent pointers are generally fastest, as long as the cost of an additional pointer field per node is not important. Threaded representations are almost as fast as parent pointers, but require less space. Right-threaded representations fare poorly in all of our experiments, as do in-order linked lists.

The remainder of this paper is organized as follows. Section 2 describes the data structures and node representations being compared and section 3 states our hypotheses for their performance. Section 4 describes the experimental platform used for comparisons. Section 5 presents the experiments and discusses their results individually, while section 6 discusses the results as a whole. Section 7 covers related work, and section 8 draws overall conclusions.

*An extended abstract of this paper appeared in the proceedings of SIGMETRICS/Performance 2004.

2 BST Variants

This paper focuses on two dimensions within the BST design space. The first dimension under consideration is the choice of data structure; the second, choice of BST node representation.

2.1 Data Structure

In an ordinary, unbalanced BST, insertion of data items in a pathological order, such as sorted order, causes BST performance to drop from $O(\lg n)$ to $O(n)$ per operation in a n-node tree [4, 5, 6].

One solution is a balanced tree, which uses balancing rules to bound tree height to $O(\lg n)$. The most popular balanced trees are *AVL trees* and *red-black trees*, which limit the height of an n-node tree to no more than $1.4405 \lg(n+2) - 0.3277$ and $2 \lg(n+1)$, respectively [7, 8, 9, 10].

Another solution is a self-adjusting tree, that is, one that dynamically moves frequently accessed nodes near the tree's root. The most popular self-adjusting tree structure is the *splay tree*, which rotates or "splays" each node to the root at time of access [11]. Splay tree amortized performance has been shown to be optimal in several important respects [11, 12].

2.2 Node Representation

At a minimum, a BST node contains a data item and left and right child pointers. Efficient traversal of such a *ordinary BST* requires maintenance of a stack of nodes to revisit. Most kinds of tree modification during traversal also require that the stack be regenerated with an $O(\lg n)$-cost search operation.

One obvious way to solve these problems is to add a parent pointer to the node, yielding a *tree with parent pointers*. Another simple technique is to add a predecessor and a successor pointer to each node, maintaining a doubly linked list of in-order nodes; we will call this a *linked list tree* for short.

Alternatively, each right child pointer that would otherwise be null can be used to point to the node's in-order successor, and similarly for left child pointers and node predecessors. These successor and predecessor pointers are called *threads*, and a tree with threads is a *threaded tree* [13]. Figure 1(a) shows a threaded BST, with threads drawn as dotted lines.

When only successors are of interest, not predecessors, it is possible to thread right child pointers only, producing a *right-threaded tree* in which finding the successor of a node is fast, but finding a predecessor requires a search starting from the root. Figure 1(b) shows a right-threaded BST.

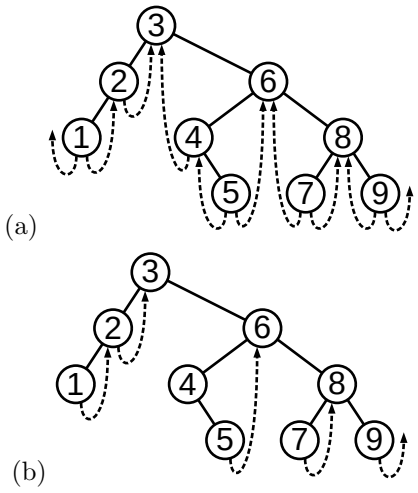

Figure 1: (a) A threaded BST, with threads drawn as dotted lines. (b) A right-threaded BST.

3 Performance Hypotheses

Before describing the experiments, we will take a brief look at the qualitative performance we expect from operations on each tree variant.

3.1 Data Structure

When data items are inserted in random order, any BST-based data structure should provide acceptable behavior. There is no need for complex rebalancing algorithms, because ordinary BSTs will produce acceptably balanced trees with high likelihood. Thus, for random insertions, data structures with the least extra overhead (above ordinary BSTs) should yield the best performance. The red-black balancing rule is more permissive than the AVL balancing rule, resulting in less superfluous rebalancing, so red-black trees should perform better for random insertions. We suspect that splay trees will be slower than either AVL or red-black trees on random insertions because they splay the inserted node to the root on every insertion, whereas the other balanced trees only perform balancing as necessary.

On the other hand, when data items are inserted in sorted order, ordinary BST behavior should be the slowest by far. In such a case, the stricter the balancing rule, the shorter in height the tree that should be produced. We therefore expect to see that for pathological input, AVL trees should perform better than

red-black trees. Known linear performance for sequential access patterns in splay trees [12] suggests that splay trees should perform well, but it is difficult to predict how they will do relative to AVL and red-black trees.

Some data structures require extra memory on top of that required by ordinary BSTs: AVL trees require 2 extra bits per node and red-black trees require 1 extra bit. Due to padding, these bits actually add 4 bytes to each node in libavl. They could be absorbed into other fields, but not in portable C code and at a performance cost. Splay trees do not require extra memory.

3.2 Node Representation

To distinguish threads from child pointers, threaded nodes require an extra check for each link followed, costing extra time. This penalty could be significant because following links is such a common operation in examining and manipulating binary trees. Following a link from a right-threaded node is the same as plain nodes on the left side, and the same as threaded nodes on the right side.

Ideally, traversal operations, for finding in-order successors and predecessors, should achieve $O(1)$ performance. The actual implementation and performance of these functions is a key difference between node representations. Based on the complexity of operation in the normal case, the table below lists them in suspected order of fastest to slowest:

linked list $O(1)$ traversal by a single branchless pointer dereference.

threads $O(1)$ traversal by following a single pointer upward or a series downward.

right threads $O(1)$ successor operation, as for threaded nodes, but finding a node's predecessor requires an $O(\lg n)$ search from the tree root when it starts from a node with no left child.

parent pointers $O(1)$ traversal by following $O(1)$ pointers upward or downward in the tree.

plain $O(1)$ traversal by maintaining an explicit stack. Performance degrades to $O(\lg n)$ if the tree is modified.

Rotation operations, used to maintain balance in balanced trees, suffer overhead under many node representations. With parent pointers, each rotation requires 3 extra pointer assignments, doubling the number of assignments. In threaded trees, each rotation requires 1 or 2 extra assignments and a two-way branch, similar to following links. In right-threaded trees, each rotation requires a two-way branch and sometimes 1 extra assignment.

Node representations also have different memory requirements. Parent pointers add a single pointer field to each node; linked list nodes add two. Threaded nodes add 2 "tag" bits to each node used to distinguish threads from child pointers, and right-threaded similarly nodes add 1 tag bit. In AVL and red-black trees, tag bits can be absorbed into otherwise unused padding at no extra cost.

4 Experimental Platform

If we consider ordinary BSTs, AVL trees, red-black trees, and splay trees, for each of the five kinds of tree nodes described above, there are 20 different ways to implement binary search tree-based data structures. To test 12 of the 20 variations, we used GNU libavl 2.0.1, a free software library in portable ANSI/ISO C [14]. GNU libavl does not currently include splay trees or linked list node representations, so the 8 remaining variants were implemented for this paper.

All of the libavl tree implementations implement the same interface, summarized in Table 1. The interface divides into two categories of functionality: "set" functions and "traverser" functions. The set interface manipulates trees in terms of a set abstraction, that is, as an unordered collection of unique items.[1]. The `probe` function combines search and insertion and the `insert` function inserts a new item into a tree when it is known not to contain an item with an equal key. Function `delete` removes an item. The `find` function searches for an item whose key equals a given target.

The traverser interface allows a program to iterate through the items in a tree in sorted order. Each traverser tracks the position of a node in the tree, with the guarantee that a traverser remains valid across any series of tree modifications as long as its own node is not deleted. A traverser may also have a null current position. Functions that create traversers include `t_first`, to create a traverser with the minimum value in the tree as its current position; `t_last`, for the maximum value; `t_find`, for a value equal to a target; and `t_insert`, to insert a new item and create a traverser at its position. Operations on traversers include `t_next` and `t_prev`, which move a traverser to the next or previous node, respectively, in sorted order and return its data.

[1] libavl does not directly support trees that contain duplicates

`find`(x): searches for an item that equals x

`insert`(x): inserts x (for use when x is known not to be in the tree)

`probe`(x): combines search and insertion: searches for x and inserts it if not found

`delete`(x): deletes an item equal to x if one exists in the tree

`t_first`(t): positions traverser t at the minimum item in the tree

`t_last`(t): positions traverser t at the maximum item in the tree

`t_find`(t, x): positions traverser t at an item equal to x or at the null position if none exists

`t_insert`(t, x): positions traverser t at an item equal to x, inserting x if none exists

`t_next`(t): advances traverser t to the next item in in-order

`t_prev`(t): advances traverser t to the previous item in in-order

Table 1: Summary of important libavl functions.

5 Experiments

This section discusses three experiments carried out to test data structure performance. Each experiment takes advantage of a different property of BSTs. Two of the three experiments are simulations of actual uses of balanced binary trees within the Linux kernel, version 2.4.20. All three are intended to represent substantial real-world uses for BST-based structures.

The primary platform for the experiment, for which times are reported, was a Compaq Armada M700 with 576 MB RAM and a Mobile Pentium III processor at 500 MHz with 256 kB cache, running Debian GNU/Linux "unstable" with the Linux 2.4.22 kernel. The compiler used was GCC 2.95.4 with flags `-O3 -DNDEBUG`.

The experiments were also run on other $x86$-based machines, including Pentium III and Pentium IV desktop machines, and under other versions of GCC, including 3.2.2 and 3.3. The differences between processors and compiler versions were minor. The one significant difference that did appear is described as part of its experimental results, in section 5.2.1.

5.1 Virtual Memory Areas

Each process in a Unix-like kernel has a number of virtual memory areas (VMAs). At a minimum, a statically linked binary has one VMA for each of its code, data, and stack segments. Dynamically linked binaries also have code and data VMAs for each dynamic library. Processes can create an arbitrary number of VMAs, up to the limit imposed by the operating system or machine architecture, by mapping disk files into memory with the `mmap` system call. A complementary system call, `munmap`, can partially or entirely eliminate existing VMAs [15].

Since VMAs vary in size over several orders of magnitude, from 4 kB to over 1 GB, conventional hash tables cannot keep track of them efficiently. Hash tables also do not efficiently support range queries needed to determine which existing VMAs overlap the range requested by a `mmap` or `munmap` call.

BST-based data structures, on the other hand, have both these properties, so many kernels use BSTs for keeping track of VMAs: Linux before 2.4.10 used AVL trees, OpenBSD and later versions of Linux use red-black trees, FreeBSD uses splay trees, and so does Windows NT for its VMA equivalents [16].

When BSTs store intervals that can overlap, efficient performance requires the tree structure to be augmented with an extra field, as in interval trees [17]. VMAs do not overlap, so this is unnecessary as long as care is taken during insertion operations. Interval trees are not used here, nor are they used by any of the kernels mentioned above.

The first experiment simulates VMA activity during program execution. Modifications to VMA tables by `mmap` and `munmap` calls are simulated by tree modifications. Test sequences were drawn from four sources, primarily by instrumenting the behavior of real programs:

- Mozilla 1.0 [18] over the course of a brief browsing session. In total, `mmap` is called 1,459 times and `munmap` 1,264 times. At the peak there are 108 VMAs, with an average of 98.
- VMware GSX Server 2.0.1 $x86$ virtual machine monitor [19] over the boot sequence and brief usage of a Debian GNU/Linux virtual machine. VMware GSX Server divides itself into several processes; behavior of only the most VMA-intensive of these is simulated. In total, `mmap` and `munmap` are each called 2,233 times. At peak there are 2,233 VMAs, with an average of 1,117.
- Squid web cache 2.4.STABLE4 running under User-Mode Linux 2.4.18.48, which relies on `mmap` for simulating virtual memory. In total, `mmap` is called 1,278 times and `munmap` 1,403 times. There are at most 735 VMAs at any time, 400 on average.
- A synthetic test set consisting of 1,024 `mmap` calls followed by 1,024 `munmap` calls, all of which begin

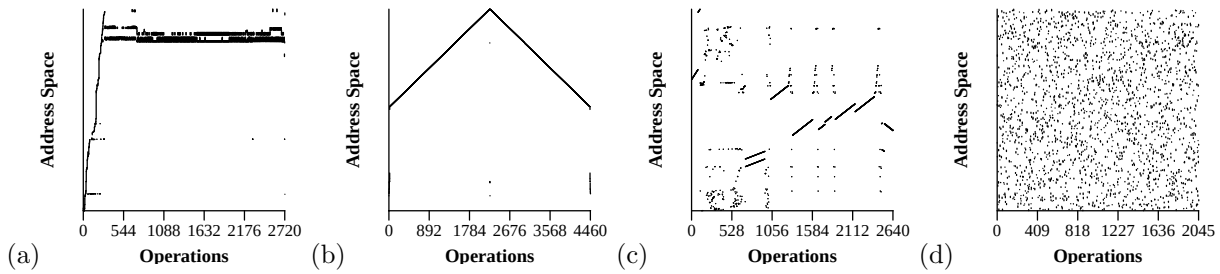

Figure 2: Call sequences in (a) Mozilla 1.0, (b) VMware GSX Server 2.0.1, (c) squid running under User-Mode Linux 2.4.18.48, and (d) random test sets. Part (b) omits one `mmap-munmap` pair for memory region 0x20000000 to 0x30000000 and (c) omits address space gaps; the others are complete.

at random page-aligned locations and continue for a random number of pages. This is unrealistic, provided as an example of the best case for an ordinary BST.

Figure 2 depicts the sequence of calls in each test set. Notably, the VMware GSX Server test set is close to being a worst case for unbalanced BSTs because of its sequence of one-page `mmap`s at sequential virtual addresses. The Linux kernel (and the simulation) never merges VMAs that refer to files, even when they map sequential blocks within the same file, so each of these calls creates a new VMA. On the other hand, the random test set is the best case for unbalanced BSTs.

The function called most often in the VMA simulation, 20,110 times among the four test sets, is `t_equal_range`, a function implemented for this simulation that finds the range of tree nodes that overlap a given range as a pair of traversers. Other functions called often are `t_next`, 11,051 times; `insert`, 5,994 times; `delete`, 5,044 times; and `t_prev`, 4,460 times. Use of other tree functions is insignificant.

Table 2 shows the time, in seconds, to run 1,000 iterations of each test set using each of the 20 tree types, and the number of comparisons performed in a single run of each.

5.1.1 Real-World Data Sets

The Mozilla, VMware, and Squid data sets are considered "real world" data sets because they are drawn directly from real programs.

Splay trees shine in the three real-world tests. They beat all of the other tree types by a wide margin, bettering the best of the competition by 23% to 40% each time. The reason lies in the test sets' high locality of reference, as shown in Figure 2, along with the splay tree's ability to keep frequently used nodes near the top of the tree. This is demonstrated by the comparison count numbers: in each case, the next larger comparison count was 2.0 to 3.4 times that of the splay tree. The tendency of the VMware and Squid test sets toward sequential access is also a factor, since such access patterns have been proved to take linear time in splay trees [12].

As expected, ordinary BSTs are slower than any other data structure for the real-world tests, by more than an order of magnitude in the pathological VMware test set. The Mozilla test with plain node representation, where ordinary BSTs are the fastest of the four, is exceptional because of a detail of the libavl implementation, in that it uses a stack of fixed maximum size for in-order traversal in plain BSTs.[2] When traversal encounters a tree branch deep enough to overflow the stack, it rebalances the entire tree, reducing the tree's height to the minimum possible for its number of nodes. Insertion, deletion, and search operations on the resulting tree are then, initially, guaranteed to run in $O(\lg n)$ time. In each run of the Mozilla and Squid test sets such forced rebalancing occurs three times for plain BSTs, and once in the VMware test set. (Plain splay trees have similar fixed-sized stacks for traversal, but the test does not trigger it.)

In these data sets, the AVL tree implementations were consistently faster than red-black trees, by up to 20%. Table 2 shows that there is a corresponding 12% to 32% increase in the number of comparisons from AVL to red-black, which suggests that the stricter AVL balancing rule may be responsible. Further investigation shows that although the average internal path length for the red-black trees is only about 3% longer than for the AVL trees, the path of maximum length is 16% to 29% longer in both test sets, as shown in table 3. The conclusion is that although both AVL and red-black trees globally balance the trees about as well, the AVL balancing rule

[2]The maximum size is 32 entries by default.

test set	representation	time (seconds)				comparison count			
		BST	AVL	RB	splay	BST	AVL	RB	splay
Mozilla	plain	4.49	4.81	5.32	2.71	71,864	62,523	79,481	23,398
	parents	15.67	3.65	3.78	2.63	635,957	54,842	68,942	22,284
	threads	16.77	3.93	3.95	2.67	635,957	54,842	68,942	22,284
	right threads	16.91	4.07	4.20	2.68	634,844	53,962	69,357	22,241
	linked list	16.31	3.64	4.35	2.74	559,062	46,904	69,232	22,240
VMware	plain	208.00*	8.72	10.59	3.77	7,503,740	137,081	193,984	32,190
	parents	447.40*	6.31	7.32	3.62	14,865,389	122,087	175,861	35,733
	threads	445.80*	6.91	8.51	3.64	14,865,389	122,087	175,861	29,758
	right threads	446.40*	6.88	8.59	3.51	14,872,042	122,076	175,886	22,725
	linked list	472.00*	7.35	8.60	3.45	14,865,389	122,087	175,861	29,831
Squid	plain	7.34	4.41	4.67	2.84	250,883	67,079	77,836	30,086
	parents	12.52	3.69	3.80	2.64	487,818	62,467	72,867	29,338
	threads	13.44	3.92	4.18	2.70	487,818	62,467	72,867	28,156
	right threads	14.46	4.17	4.27	2.86	511,903	63,189	71,629	30,199
	linked list	13.13	4.02	4.19	2.65	487,818	62,467	72,867	28,173
random	plain	2.83	2.81	2.86	3.43	37,396	34,090	34,176	44,438
	parents	1.63	1.67	1.64	1.94	28,123	25,958	25,983	33,036
	threads	1.64	1.74	1.68	2.02	28,123	25,958	25,983	33,036
	right threads	1.92	1.96	1.93	2.22	30,598	28,438	28,393	34,672
	linked list	1.46	1.54	1.51	1.74	27,845	25,808	25,605	31,965

Table 2: Time, in seconds, for 1,000 runs of each VMA simulation test set, and number of comparisons during each run. *Estimated based on 10 runs.

leads to better local balancing in the important places for test sets like these with strong locality. The splay tree results for maximum path length are deceptive because a splay operation on a node roughly halves the path length to every node along its access path, so that only nodes accessed infrequently will lie along the path of maximum length.

Considering AVL and red-black tree implementations for the real-world test sets, in all cases plain representations were the slowest and parent pointer representations were the fastest. Given the experimental operation mix, there are two causes for this behavior. First, the plain representation is slow because insertions and deletions invalidate the stacks required for traversal with plain nodes, forcing `t_next` to perform an extra search; the right-threaded representation is slow because of the use of `t_prev`. Second, parent pointers are faster than threaded representations because distinguishing threads from ordinary child pointers requires an extra step.

5.1.2 Random Data Set

For the random data set, all of the implementations were clustered within a relatively small range of about 2.3 times variation. Within each node representation, speed tended to be ordered from fastest to slowest as ordinary BST, red-black tree, AVL tree, splay tree. Intuitively, the fastest implementations should be the ones that perform the least extra work, because random order is ideal for an ordinary BST. To verify this, we must have some means to measure "extra work"; because the tested balanced trees all rebalance in terms of rotations, counting rotations is one reasonable way. For the random data set, ordinary BSTs make no rotations, red-black trees make 209, AVL trees make 267, and splay trees make 2,678.[3] This ordering neatly coincides with observed performance tendencies, further explaining why splay trees are consistently separated from the other tree types by a relatively wide gap.

Along the other dimension, speed tended to be arranged from fastest to slowest in the order of linked list nodes, nodes with parent pointers, threaded nodes, right-threaded nodes, and plain nodes, indicating that performance favors representations that make in-order traversal speedy.

Splay trees consistently performed worst, and required the most comparisons, within each node representation category in the random test. This reflects

[3]Number of rotations is generally independent of node representation. Numbers for right-threaded trees do differ slightly due to a specialized deletion algorithm.

test set	AVL	red-black	splay
Mozilla	468 : 6.8	485 : 8.1	1,227 : 25.6
VMware	9,471 : 9.2	9,723 : 13.1	426,378 : 572.3
Squid	2,903 : 8.5	3,000 : 10.7	20,196 : 100.2

Table 3: Average internal path length and average maximum path length, respectively, for each kind of tree in the real-world test sets.

the work required to splay each node accessed to the root of the tree, in the expectation that it would soon be accessed again. This effort is wasted for the random set because of its lack of locality.

5.1.3 Optimization

One feature missing from libavl is the ability to directly delete the node referenced by a traverser. Such an operation can avoid a search from the root to the node to delete and thereby reduces deletion time to $O(1)$. For the purposes of this paper such a `t_delete` function was written for implementations with threads and parent pointers. Plain representations are unlikely to benefit because most changes invalidate all traverser stacks, and implementation for right-threaded representations appears to always require a search, so `t_delete` was not implemented for those representations.

Table 4 shows the timings when `t_delete` is substituted for `delete` where appropriate. Up to 16% improvement was observed for the balanced tree implementations, even larger for the unbalanced trees. Based on these results, we suggest that future versions of libavl should include a `t_delete` function.

A related function `insert_before`, for $O(1)$ insertion with a known insertion point, was also implemented and found to give this experiment no noticeable performance benefit, so detailed analysis is omitted.

5.2 Internet Peer Cache

RFC 791 [20] requires that each IP packet sent by an internet host be stamped with a 16-bit identification field that "must be unique for that source-destination pair and protocol for the time the datagram will be active in the internet system." Most TCP/IP implementations ensure this by maintaining a counter for each host they contact, incrementing a host's counter once for every packet transmitted to it.

The Linux kernel keeps track of identification data using an AVL tree indexed by peer IP address.[4] As

test set	representation	BST	AVL	RB	splay
Mozilla	parents	11.12	3.25	3.17	2.53
	threads	11.91	3.52	3.30	2.67
VMware	parents	331.00*	5.42	6.35	3.22
	threads	325.00*	5.96	7.42	3.55
Squid	parents	11.22	3.54	3.62	2.53
	threads	12.10	3.78	3.99	2.70
random	parents	1.60	1.64	1.62	1.90
	threads	1.61	1.71	1.65	2.02

Table 4: Time, in seconds, for 1,000 runs of VMA simulation test sets optimized with `t_delete`. *Estimated based on 10 runs.

a comment in the source notes: [21]

> Such an implementation has been chosen not just for fun. It's a way to prevent easy and efficient DoS attacks by creating hash collisions. A huge amount of long living nodes in a single hash slot would significantly delay lookups...

Techniques such as universal hashing [22] can make hash collision attacks significantly more difficult. On the other hand, balanced tree techniques make them impossible, so they are a way to dodge an entire class of potential problems.

The peer AVL tree is used as a second-level cache only. The first time a packet is sent to a given host, a node with its IP address is inserted into the tree. A direct pointer to this node is also put into the route cache, which is the first-level cache. As long as the host's route remains cached, the AVL tree is not consulted again for that host. If the route is dropped from the cache (generally due to inactivity), and later added back in, the AVL tree is searched a second time. If, on the other hand, the route is dropped and remains inactive, sometime later the host's peer cache AVL tree entry will be dropped. Peer cache entries not recently used can also be dropped if the cache grows too much, but entries are never dropped while in use in the route cache.

The simulation attempts to reproduce the most important features of the peer cache. The route cache is modeled heuristically, not directly. All other features described above are directly reproduced. Two scenarios are modeled:

- Web server under normal load. A new host with a randomly selected IP address makes its first request every 10 ms to 1.5 s, with a mean of

[4] Although RFC 791 allows it, Linux does not maintain separate identification counters for TCP and UDP protocols.

test set	representation	BST	AVL	RB	splay
normal	plain	4.74	5.10	5.06	7.70
	parents	3.94	4.07	3.78	7.19
	threads	3.99	4.45	4.17	13.25
	right threads	5.52	5.71	5.64	8.29
	linked list	4.93	5.40	5.25	8.41
attack	plain	*	3.76	4.32	4.14
	parents	*	2.65	2.92	3.31
	threads	*	2.97	3.37	7.77
	right threads	*	4.04	4.77	4.62
	linked list	*	4.10	4.46	4.78

Table 5: Times, in seconds, for 500 and 5 runs, respectively, of the normal and attack scenario simulations. *Greater than 60 seconds for a single run.

38 ms. Each currently connected host makes a request every 10 ms to 10 min, with a mean of 30 s, and hosts make a mean of 9 requests each.

- Attack. A new "host" with a IP address consecutively greater than the previous makes a request every 10 ms for 1,000 s. "Hosts" otherwise behave as above, in order to ensure that they stay cached for some time.

In each scenario, new peers continue to arrive for 1,000 s simulated time, and then the simulation continues with no new peers until the cache becomes empty. As a result, insertion (`insert` or `t_insert`) and deletion (`delete` or `t_delete`) functions are called the same number of times for each test set, 2,691 times each for the normal web server test set and 137,146 times for the attack test set. Due to the read-write locks used by the kernel implementation, a `find` operation is performed before every insertion. Along with the searches performed to look up peer cache entries expired from the route cache, this makes `find` the most-used function at 3,884 calls in the normal test set and 147,942 in the attack test set. Use of other libavl functions is unimportant.

Table 5 shows the time required for multiple simulated runs of each test set using each tree type. Unbalanced binary trees were not simulated in the attack scenario because of pathological behavior of the algorithm in that case. Ordinary BSTs and splay trees are included in the table, but would not be suitable choices in this scenario because they do not provide guaranteed performance.

5.2.1 Normal Data Set

Within the normal data set, if we sort the representations by speed within each type of tree, in three of four cases the order is the same. From fastest to slowest, this order is: parent pointers, threads, plain, linked list, and right threads. The speed of parent pointer and thread representations reflects the ability to use `t_delete` for $O(1)$ deletion. Among the other representations, plain is fastest because it is not hampered by manipulation of a linked list or right threads, which do not provide any benefits for this experiment.

Splay trees are the exception, for which the threaded representation is by far the slowest. After some investigation, the problem turned out to be an unexpected artifact of the libavl implementation of threaded splay trees. In particular, the routine that implements splaying moves up the splay tree to the root using a well-known algorithm for finding the parent of a threaded node [13, exercise 19]. This algorithm is $O(1)$ on average for randomly selected nodes in a threaded tree, and its wide use in libavl's threaded tree implementations shows that it has reasonable performance in most cases. Unfortunately, its pattern of use for splaying coupled with the typical structure of a splay tree leads to poor performance. (The same effect is visible, to a lesser degree, in the threaded splay tree performance shown earlier in Table 2.)

To fix this performance anomaly, the implementation of threaded splay trees was modified to maintain a stack of parent pointers for splaying purposes, instead of depending on the algorithm for finding a node's parent. The revised code brought the threaded splay tree runtime down to 8.11 s for the normal case and 4.47 s for the attack case, which are much more competitive times. The revised performance is still not entirely consonant with the other threaded trees, probably due to the need to maintain the stack. (Similar improvement was found when the revised code was substituted in the first experiment.)

When this experiment was run on a Pentium IV system, instead of the Pentium III-based primary experimental platform, the results changed significantly for splay trees. In particular, the ordering from fastest to slowest changed to parent pointers, linked list, threads, right threads, and plain. We speculate that the Pentium IV has extra-large penalties on branches, favoring parent pointer and linked list representations which require fewer branches than threaded or right-threaded nodes.

In the normal case, within each node representation, speed is consistently, from fastest to slowest, in the order of unbalanced trees, red-black trees, AVL trees, and splay trees. This matches our hypotheses for performance of insertion in random order (see 3). Red-black trees with parent pointers are an unex-

plained anomaly.

5.2.2 Attack Data Set

In the attack case, AVL trees perform better than red-black trees because the AVL balancing rule does a better job of keeping the tree height to a minimum at the important point, the point of the next insertion: the maximum path length in the AVL tree, averaged over all operations, is 14.6, whereas in the red-black tree it is 21.5. On the other hand, the average internal path length across all the operations varies by less than 1% between AVL and red-black trees.

Splay trees fare poorly for the attack data set, which is somewhat surprising given that sequential accesses in splay trees perform in linear time. Examination of the data set revealed that although peer IP addresses appear in sorted order, the subsequent randomness in request spacing is sufficient to create a great deal of disorder: over the 285,088 requested handled by the data set, there is an average distance of 22,652 operations between requests with sequential IP addresses.

5.3 Cross-Reference Collator

The two previous experiments focused on problems that were solved especially well with the use of BST-based data structures. The third experiment is somewhat different in that its problem can be better solved with other techniques; in particular, by use of a hash table during accumulation of data followed by a final sorting pass. It remains interesting for at least one reason: the C++98 standard library [23] contains "set" and "map" template classes based on balanced trees,[5] but none based on hashes. For that reason, C++ programmers interested in convenience and portability may select one of these standard templates where a hash-based implementation would be more appropriate.

Cross-references of identifier usage can be useful in software development, especially when source code used as a reference is available as hard copy only. One way to construct such a tool is to divide it into parts connected through a Unix pipeline, like so:

```
find . | extract-ids | sort | merge-ids
```

where `extract-ids` reads a list of files from its input, extracts their interesting identifiers, and writes them along with filename and line number to its output, `merge-ids` collects adjacent lines for identical identifiers into a more readable format, and `find` and `sort` are the standard Unix utilities. This experiment examines the performance of a utility to take the place of the final two programs, implemented using libavl.

This program, here called a "cross-reference collator," inserts a set of identifiers into a single tree as they are read. In addition, each identifier has its own tree that contains names of the files in which the identifier appears. Furthermore, each file has a set of line numbers attached to it corresponding to the lines in the file on which the identifier appears. After all of the input has been read, the collator traverses the data structure to produce a fully sorted cross-reference listing.

The input used for testing the collator was contents of directory `include/linux` within the source code for Linux 2.4.18. When `extract-id` is used, this directory contains 118,639 interesting identifiers, or 47,952 dropping duplicates, in 841 files. With this test set, the collator utility creates a total of 47,953 trees (one for each unique identifier plus one to organize the identifiers) and makes 237,278 calls to `probe` (two for each identifier). It also calls `t_next` 159,815 times while composing output.

Table 6 shows the times, in seconds, for 5 runs of the cross-reference collator over the test set for each of libavl's table implementations. The test set labeled "normal" is the normal case where the collator's input is presented in the order it is produced by `extract-ids`. Sets "sorted" and "shuffled" present the artificial cases where its input is arranged in order by identifier or in random order, respectively, beforehand. Splay trees are tested, but their amortized performance guarantees do not meet the per-operation requirements of the C++98 standard. Unbalanced BSTs also do not meet these requirements.

5.3.1 Normal Data Set

For the normal case, splay trees are consistent winners, taking advantage of locality, resulting from the tendency of identifiers to recur within a few lines of first mention. Red-black trees are consistently slightly faster than AVL trees, reflecting the more relaxed red-black balancing rule that results in less rebalancing. Finally, the times for unbalanced trees are only at most 11% greater than for AVL trees, demonstrating that the natural identifier ordering tends toward randomness.

5.3.2 Sorted Data Set

As usual, splay trees are the big winners in the sorted case, beating all other implementations by 28% or

[5] Implementations based on skip lists, etc., are also possible. The GNU/HP/SGI implementation uses red-black trees with parent pointers.

test set	representation	BST	AVL	RB	splay
normal	plain	5.22	4.62	4.49	4.03
	parents	4.97	4.47	4.33	4.00
	threads	5.00	4.63	4.51	4.03
	right threads	5.12	4.66	4.57	4.06
	linked list	5.05	4.79	4.59	4.03
sorted	plain	*	4.21	4.90	2.91
	parents	*	4.04	4.70	2.90
	threads	*	4.25	5.02	2.89
	right threads	*	4.24	5.02	2.87
	linked list	*	4.31	4.98	2.83
shuffled	plain	5.98	5.80	5.69	6.54
	parents	5.80	5.68	5.51	6.61
	threads	5.80	5.77	5.65	6.56
	right threads	5.89	5.80	5.71	6.63
	linked list	5.88	6.06	5.84	6.64

Table 6: Times, in seconds, for 5 runs of the unsorted and sorted versions of the cross-reference collator for each kind of tree. *Pathological case not measured.

more. Comparing AVL and red-black tree results, the AVL tree's stricter balancing rule pays off, producing a tree of identifiers with maximum path length of 14 (averaged over all insertions) instead of the red-black tree's 25, and yielding the 13% to 16% edge of AVL trees in that category.

5.3.3 Shuffled Test Set

The results for insertion in random order again reflect the disadvantage of splay trees for random access and the advantage of the more relaxed red-black tree balancing rule. More curious is that every time in the "shuffled" test set is worse, by 13% to 21%, than the corresponding time in the normal set. Normally one would expect that the shuffled times should be about the same or better than the times for normal insertion order, because random order is ideal for BST insertion.

The processor cache, along with the structure of the input data, turns out to be the culprit. When any given identifier is used once in a source file, it tends to be used again within a few lines of code. This natural clustering yields a good hit rate in the processor cache. Shuffling the input data destroys clustering and reduces the effect of the processor cache for large input data sets. To test this hypothesis, the experiment was rerun using only the first 10,000 identifiers from the test set. The results, shown in Table 7, confirm the idea: the edge of the normal input data over the shuffled input data is reduced to at most 7% for non-splay variants. Further investigation using the P6 performance counters [24] directly confirms that reducing the test set size reduces additional cache misses in the shuffled case from 248% to 33%.

test set	representation	BST	AVL	RB	splay
normal	plain	3.24	3.09	3.01	2.80
	parents	3.10	2.94	2.86	2.81
	threads	3.12	3.04	2.98	2.82
	right threads	3.21	3.06	3.02	2.85
	linked list	3.19	3.21	3.08	2.87
shuffled	plain	3.23	3.28	3.24	3.51
	parents	3.13	3.12	3.05	3.55
	threads	3.15	3.20	3.15	3.55
	right threads	3.22	3.22	3.19	3.60
	linked list	3.24	3.43	3.32	3.64

Table 7: Times, in seconds, for 50 runs of a reduced version of the cross-reference collator for each kind of tree designed to fit within the processor cache.

6 Discussion

The preceding sections examined three experiments with binary search tree-based data structures and discussed the results for each one in isolation. This section attempts to extend these individual results into broader conclusions about the performance of BST-based data structures in a more general context.

Each of the experiments involves three different kinds of test sets: those drawn from real-world situations, and variations that yield best- and worst-case performance in unbalanced BSTs. (For the peer cache experiment, the normal case is close to the best case.) The experimental results allow straightforward guidelines for effective choice of data structure to be drawn up, described in detail in the following sections.

6.1 Choice of Data Structure

In the best case for unbalanced BSTs, that is, insertion of items in random order, unbalanced trees are the best choice because of their low time and space overhead. If items are normally inserted in random order but include occasional runs in sorted order, then red-black trees are preferred to AVL trees because their more relaxed balancing rule does less work attempting to balance already random data. Splay trees are undesirable, because of the cost of splaying on every access.

In the worst case for unbalanced BSTs, that is, insertion of items in sorted order, splay trees are the

best choice as long as subsequent accesses tend to be somewhat sequential (as in the VMA experiment in section 5.1) or clustered (as in the cross-reference experiment in section 5.3). Splay trees should not, however, be used if bounded performance guarantees are required. Otherwise, AVL trees are a better choice because they tend to keep the maximum path length below that for red-black trees, although the internal path length for AVL and red-black trees is comparable. Unbalanced BSTs should not be considered.

Real-world situations fall somewhere between the two extremes: in two of our experiments red-black performance was better than AVL performance and in the other one it was the reverse; in one, splay trees were better than either. The choice of data structure can then be made based on which extreme is thought more likely or for which performance is more important, or based on actual tests.

6.2 Choice of Node Representation

The results also aid the choice of BST representation. BST representations with parent pointers had the best performance in all three of our experiments, followed closely by threaded representations. Threads use less memory than a parent pointer and are therefore preferable when space is at a premium.

A plain BST representation, without parent pointers or threads, saves time taken updating these additional fields, but the advantage is more than made up by costs incurred when traversal is combined with modifications. A plain representation also seems to preclude efficient implementation of `t_delete`.

The linked list node representation was a clear winner for three out of four data sets for the VMA experiment, which made heavy use of traversal operations `t_next` and `t_prev`, but its use exacted a significant penalty for the peer cache experiment. Its high memory cost, two pointer fields per nodes, means that it should only be used when traversal is common.

Right threads seem rarely preferable, especially since there seems no efficient way to implement `t_delete`. (Of course, `t_prev` is also slow in right-threaded trees.) Right threads performed poorly in all three experiments.

7 Related Work

Few comparisons of BST-based data structures exist in the literature, and none make use of real-world test sets or include red-black or splay trees. In 1976, Karlton et al. [1] experimentally examined the performance of HB[k] trees, a class of height-balanced binary trees for which AVL trees form the special case $k = 1$, but did not compare HB[k] trees to other forms of balanced tree. Baer and Schwab [2] empirically compared height-balanced, weight-balanced, and periodically totally restructured trees, using synthetic test sets, and concluded that AVL trees offer the best performance. Cannady [25] made theoretical comparisons height-balanced, bounded-balance, and weight-balanced trees. In 1980, Wright [3] analyzed the performance of several types of trees using a variety of synthetic test sets.

One FreeBSD developer has reported that switching from doubly linked lists to splay trees increased the overall speed of a web server execution by 23% [26].

8 Conclusion

In this paper, we empirically compared the performance of 20 variants on BSTs in three different real-world scenarios, and demonstrated that choice of data structure can significantly impact performance. Our results show that BST data structures and node representations should be chosen based on expected patterns in the input and the mix of operations to be performed.

We found that in selecting data structures, unbalanced BSTs are best when randomly ordered input can be relied upon; if random ordering is the norm but occasional runs of sorted order are expected, then red-black trees should be chosen. On the other hand, if insertions often occur in a sorted order, AVL trees excel when later accesses tend to be random, and splay trees perform best when later accesses are sequential or clustered.

For node representation, we found that parent pointers are generally fastest, so they should be preferred as long as the cost of an additional pointer field per node is not important. If space is at a premium, threaded representations conserve memory and lag only slightly behind parent pointers in speed. A plain BST has fewer fields to update, but combining traversal and modification requires extra searches. Maintain a linked list of in-order nodes is a good choice when traversal is very common, but exacts a high memory cost. Finally, right-threaded representations fared poorly in all of our experiments.

We also showed that implementing an routine for deleting a node referenced by a traverser can yield significant performance improvement for some workloads. Based on our results future versions of libavl are likely to add support for such an operation, as well as splay trees.

Acknowledgments

The author thanks advisor Mendel Rosenblum and colleague Tal Garfinkel. The author was supported by a Stanford Graduate Fellowship during the course of this work.

References

[1] P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler, "Performance of height-balanced trees," *Communications of the ACM*, vol. 19, no. 1, pp. 23–28, 1976.

[2] J.-L. Baer and B. Schwab, "A comparison of tree-balancing algorithms," *Communications of the ACM*, vol. 20, no. 5, pp. 322–330, 1977.

[3] W. E. Wright, "An empirical evaluation of algorithms for dynamically maintaining binary search trees," in *Proceedings of the ACM 1980 annual conference*, pp. 505–515, 1980.

[4] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*, section 6.2.2, pp. 430–31. Reading, Massachusetts: Addison-Wesley, second ed., 1997.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, section 13, p. 244. MIT Press, 1990.

[6] R. Sedgewick, *Algorithms in C, Parts 1–4*, section 12.6, pp. 508–511. Addison-Wesley, 3rd ed., 1998.

[7] G. M. Adel'son-Vel'skiĭ and E. M. Landis, "An algorithm for the organization of information," *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1262, 1962.

[8] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*, section 6.2.3, p. 460. Reading, Massachusetts: Addison-Wesley, second ed., 1997.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, section 14.1, p. 264. MIT Press, 1990.

[10] R. Sedgewick, *Algorithms in C, Parts 1–4*, section 13.4, p. 556. Addison-Wesley, 3rd ed., 1998.

[11] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, vol. 32, pp. 652–686, July 1985.

[12] R. E. Tarjan, "Sequential access in splay trees takes linear time," *Combinatorica*, vol. 5, no. 4, pp. 367–378, 1985.

[13] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*, section 2.3.1, pp. 322–327. Reading, Massachusetts: Addison-Wesley, third ed., 1997.

[14] B. Pfaff, *An Introduction to Binary Search Trees and Balanced Trees*, vol. 1 of *libavl Binary Search Tree Library*. Free Software Foundation, 2.0.1 ed., 2002.

[15] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4BSD operating system*, ch. 5, pp. 117–190. Addison Wesley Longman Publishing Co., Inc., 1996.

[16] H. Custer, ed., *Inside Windows NT*, p. 200. Microsoft Press, 1993.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, section 15.3, pp. 290–295. MIT Press, 1990.

[18] "Mozilla 1.0." Software, June 2002.

[19] VMware, Inc., "VMware GSX Server 2.0.1." Software, 2002.

[20] J. Postel, "RFC 791: Internet Protocol," Sept. 1981. Status: STANDARD.

[21] A. V. Savochkin, "net/ipv4/inetpeer.c." Linux 2.4.18 kernel source, 2001.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, section 12.3.3, pp. 229–232. MIT Press, 1990.

[23] International Organization for Standardization, *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Sept. 1998.

[24] Intel Corporation, *Intel Architecture Software Developer?s Manual Volume 3: System Programming*, 2002.

[25] J. M. Cannady, "Balancing methods for binary search trees," in *Proceedings of the 16th annual conference on Southeast regional conference*, pp. 181–186, ACM Press, 1978.

[26] A. Cox, "fa.m7rv3cv.klm8jr@ifi.uio.no." Usenet, May 2002.