

A Randomized Self-Adjusting Binary Search Tree

Mayur Patel
VFX Department Supervisor
Animal Logic Film
patelm@acm.org

We present algorithms for a new self-adjusting binary search tree, which we call a shuffle tree. The tree is easy to implement and does not require parent pointers or balancing information to be stored in tree nodes. A maximum of one rotation is probabilistically applied during each traversal, which keeps the cost of balancing activity low. We present tests to compare the performance of shuffle trees with splay trees and randomized binary search trees.

Categories and Subject Descriptors: E.1 [Data]: Data Structures-*trees*

General Terms: Algorithms

Additional Key Words and Phrases: binary search trees, self-adjusting data structures, randomized data structures

1. Introduction

Binary search trees are fundamental in computer science and are frequently used to implement associative data structures. There are many different ways of maintaining them. Many binary search trees require balancing information to be stored in each node, but some trees do not [5] [10] [11]. These have a slight advantage for the software engineer, because they consume less memory.

Scapegoat trees monitor traversal depth to detect poor balance [5]. A node is identified under which balance is particularly bad; the node is called the scapegoat. The sub-tree under the scapegoat is rebuilt using an operation that is linear with its size. These trees perform well under random access, but sequential insertion performs poorly because of repeated, expensive rebalancing operations. Scapegoat trees do not optimize access to the most active data.

Splay trees are perhaps the most interesting binary search tree whose nodes do not store balancing data. Nodes found during traversal are relocated to the root of the tree by applying a series of rotations, called splaying [11]. Allen and Munro presented a similar move-to-root scheme [3], but splay trees perform better by applying a more sophisticated combination of rotations depending upon the configuration of the traversal path. Because splaying occurs during searches, the tree restructures itself to optimize access to the most active data. Splay trees are the most popular and successful self-adjusting binary search tree.

Many have recognized that splaying can sometimes be expensive. The authors suggested that work could be reduced by splaying only deep traversal paths or by using a different set of rotations, known as semi-splaying [2][11]. Others have suggested that steps can be probabilistically skipped to reduce costs [1] [4].

Here, we introduce a new binary search tree which we call a shuffle tree. The name comes from how it converges to its balanced state. A rotation in a binary tree relocates a range of keys within the tree in much the same way that cutting a deck relocates a range of playing cards. During each access to the shuffle tree, we perform at most one probabilistic rotation. Just as cutting a deck a large number of times results in a randomized order of playing cards, a large number of shuffle tree operations results in a good node configuration.

Like scapegoat trees, shuffle trees sample tree depth to identify regions which may be unbalanced. A randomized counter is decremented during traversal until it reaches zero, at which time a rotation is performed. This balancing operation occurs during all operations, including searches, so the tree reconfigures itself for the distribution of accesses. Shuffle trees do not require balancing data to be stored in tree nodes because counters are reset for each access.

Allen and Munro presented a rotate-parent scheme for self-adjusting binary search trees [3]. Shuffle trees differ because it is possible for any ancestor to be selected for rotation. Also, if the working set is clustered at the root of the tree, then traversal paths may not be long enough to trigger balancing activity very often. This tends to preserve favorable shuffle tree configurations.

Martinez and Roura discussed a randomized binary search tree that ensures an equal probability of any node becoming the root during insertion [8]. As a result, the data in the tree behave as if they were inserted randomly, which has the property of being probabilistically balanced [9] [10]. At each iteration, a new random number is calculated which determines whether the current node will be swapped with the new node during insertion. They discussed a self-adjusting strategy that involves probabilistically choosing an ancestor of the target node with which to swap positions. The tree uses more random bits than shuffle trees, because a new random number is calculated for each iteration. Furthermore, the tree potentially does more restructuring per-operation than the shuffle tree. Swapping node positions in the tree requires specialized operations.

Seidel and Aragon presented a different randomized binary search tree that uses randomized priorities in a treap data structure [10]. The balancing activity is based on single rotations and the expected number of rotations per operation is two. Since the random priorities are usually immutable, the authors described a variation of the tree which uses hashing. Hashing eliminates the need to store balancing information in the tree. Their proposed self-adjusting strategy changes the priorities of frequently accessed items over time; and so it does not apply to the hashing-based implementation. Furthermore, since priorities are updated monotonically, it will not react well if access patterns change.

In section 2, we present the algorithms that govern the behavior of shuffle trees. In section 3, we present variations of shuffle tree algorithms and we discuss implementation issues. In section 4, we compare test results of shuffle tree operations with those of splay trees and randomized binary search trees. We summarize our conclusions in section 5.

2. Shuffle Trees

The motivation behind shuffle trees is to reduce the cost of self-adjustment, especially when the tree already displays good balance. The solution has the added benefit of having a small memory footprint. Here, we describe the algorithms that determine the behavior of the shuffle tree.

Before we begin, we require some definitions:

- A self-adjusting binary search tree is one that reconfigures itself to optimize performance for the distribution of accesses, even during searches. It differs from self-balancing trees because it does not necessarily guarantee $O(\lg N)$ worst-case access time. Self-adjusting trees require no prior information about the pattern of accesses.
- A height-balanced tree is one where the descendents of each node are equally split between its left and right sub-trees.
- If each node in a binary tree has an associated weight, then it is weight-balanced if the sum of the weights in each node's left sub-tree is equal to the sum of the weights in its right sub-tree. We will concern ourselves only with a specific kind of weight-balanced tree; where the weight of each node, n , is the probability of it being accessed, p_n .

- The size of a height-balanced tree is the number of nodes in the tree. Assume that we have a weight-balanced tree with probabilities for weights as described above. The tree has N nodes. If a sub-tree contains a set of nodes, U ; then we use the term “size” to describe the sub-tree’s metric: $\left(\sum_{i \in U} Np_i\right)$. Using this definition, we can say that, in a weight-balanced tree, the size of the left sub-tree under each node is equal to the size of the right sub-tree.
- We will use α -balance as a metric of balance quality [5]. If U is the set of nodes rooted at node n , and $S(U)$ is the size of U , then the size of n ’s left and right sub-trees cannot exceed $\alpha S(U)$. The value of α is bounded by $\frac{1}{2} \leq \alpha < 1$. We use the definition of size which takes weights into account as described above. This will allow us to apply the α -balance criterion to weight-balanced trees. If the α -balance for a tree, T , is given by α_T , then the α -balance for each node in the tree does not exceed α_T . Notice that α -balance values are relative to a specific access distribution.
- We define the working set as the set of nodes whose probability of being accessed is above an arbitrarily low threshold. While the size of the working set may be close to N , the number of nodes may be significantly less.
- The expression $\lg x$ is the base-two logarithm of x . Logarithms of other bases will be written explicitly, such as $\log_B x$.
- The shuffle tree balancing operation uses single rotations. Figure 1 demonstrates the effects of a single rotation. In section 3, we will briefly describe how double rotations can also be used in some applications; but otherwise, rotations discussed should be assumed to be single rotations.

Single Rotation

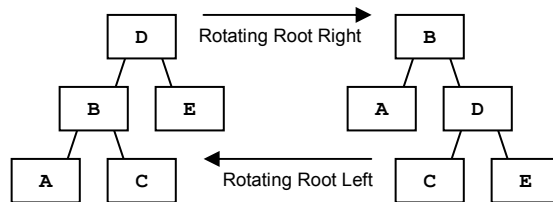


Figure 1: This figure demonstrates how a single rotation changes the configuration of a binary tree.

When traversing the shuffle tree, we conduct a probabilistic balancing operation. At the beginning of traversal, we set an integer count-down value, I , to a random number in the range $[0, N-1]$, where N is the size of the tree. As we iterate from a parent node to its child, we decrease the counter with $I := \frac{1}{2}(I - 1)$. When the counter equals zero, then the current node becomes a candidate rotation pivot. If we need to iterate past the candidate pivot, then we will commit to the rotation. We rotate the pivot away from the direction of traversal.

As search depth increases, the likelihood of a rotation increases. No rotations will occur beyond depth $\lg N$. The counter requires $\lg N$ random bits per operation. Shuffle trees also record their size, so that the counter can be set. No balancing information needs to be recorded in tree nodes. The quality of the random number generator is relevant, but this is ignored for our purposes.

Algorithms for search, insertion and deletion from unbalanced binary search trees can be easily modified to support this balancing operation. Insertion requires a traversal to identify a leaf upon which to append the new node. Deletion requires a traversal to identify the target node, as well as

to identify a leaf which can easily be swapped with the target. These paths extend from root to leaf, so they provide good opportunities to sample depth and to conduct balancing activity. Searches may not navigate to a leaf; if the working set is clustered near the root, then deep searches will not be required. As a result, rotations can occur less frequently in a well-configured tree.

If a node in the tree is not weight-balanced, then an access is more likely to traverse into its larger sub-tree. A rotation at the node probably moves some of the descendants from the larger sub-tree to the smaller one. Since these operations are probabilistic, rotations will occur which can deteriorate balance; but as the imbalance increases, the likelihood of a rotation that improves balance increases.

In effect, the balancing technique is a kind of random sampling. Nodes are selected randomly from traversal paths and their balance is manipulated. Frequently used data attract more attention and, therefore, benefit more from balancing activity than infrequently used data. The tree eventually approximates a weight-balanced configuration for the data set, where the probability of access is the weight for each node.

The cost of shuffle tree operations can be determined by measuring the cost of a shuffle tree traversal. Searches, insertions and deletions all execute a constant number of traversals, with constant-time overheads.

[Proofs omitted.]

3. Implementations

Shuffle tree nodes do not store any balancing information and do not require parent pointers if rotations are done within the traversal loop. An integer records the total number of elements in the tree. Shuffle trees also require a random number generator. These resources are sufficient for calculating the random counter which is used to identify a rotation pivot.

Sleator demonstrated that top-down splay trees can be abstracted such that all other routines defer to a central function for balancing activity and traversal [12]. Writing search, deletion and insertion functions becomes very simple as a result of this centralization. Shuffle trees can likewise be implemented around a traversal function.

We provide one possible traversal function for shuffle trees. We assume parent pointers are available to simplify the presentation:

```
// returns node with key k,  
// or returns the leaf containing  
// the closest key to k.  
node *  
Traverse(  
    key        k,  
    node       *root,  
    int        treesize  
) {  
    signed int iCounter = rand() % treesize;  
    node       *pRet = 0;  
    node       *p = root;  
  
    while( p )  
    {  
        pRet = p;  
        if( k < value(p) )  
        {  
            p = left(p);  
  
            if(( ! iCounter ) && p )  
            {  
                RotateRight( pRet );  
            }  
        }  
    }  
}
```

```

        pRet = parent(p);
    } // end if

} else if ( value(p) < k )
{
    p = right(p);

    if(( ! iCounter ) && p )
    {
        RotateLeft( pRet );
        pRet = parent(p);
    } // end if

} else
    break; // break while

--iCounter;
iCounter >>= 1;

} // end while

return( pRet );
}

```

We hope this is sufficient to demonstrate the simplicity of implementing shuffle trees. Next we discuss some variations to the basic shuffle tree and discuss when such variations might be useful.

A simple way of accelerating the response of the shuffle tree is to use a smaller bound to calculate the random counter. Martinez and Roura suggested a similar strategy to accelerate the self-adjusting version of their randomized binary search tree [8]: $I = \text{rand}(0, f(N))$, where $f(N) < N$.

Unfortunately, balance beyond depth $\lg(f(N))$ will suffer. If the number of nodes in the working set is known, this would be a good value to use for calculating the random counter.

The shuffle tree could also be made more responsive by allowing for more than one rotation per access. When a rotation occurs, the count-down value can be reset. This would make a subsequent rotation possible. As the tree converges to a balanced state, the tendency to conduct multiple rotations will be suppressed and the tree will act more like a normal shuffle tree. In our testing, we did not observe significantly improved performance from this modification; presumably this is because our tests did not show particularly bad static balance.

In some cases, it might be desirable to suppress the optimizing behavior. One could do this by introducing a new counter to measure the actual depth of traversal and by comparing it against a depth bound. If the depth of the traversal exceeds the given bound, and if the random counter has identified a rotation pivot; then the rotation is performed. If the target node is found to be shallower than the depth bound, then no rotation is performed even if a rotation pivot has been identified.

As the depth bound is increased, fewer rotations occur. Activity to optimize access to the working set is suppressed, but rotations can still occur to improve balance if a reasonable depth bound is used.

For some applications, it may be necessary to prevent leaf nodes from being rotated such that they become non-leaf nodes. Data compression is one such application [2][7]. The code for data in the leaf node is easily derived from the unique traversal path from the root to the leaf. The normal balancing operation, which uses single rotations, is not optimal for this situation, because it relies on leaf nodes becoming non-leaves in order to propagate data quickly toward the root of the tree. Double rotations can be introduced to the shuffle tree balancing operation to improve

performance under these circumstances. The following figure illustrates how a double rotation reconfigures nodes in a tree.

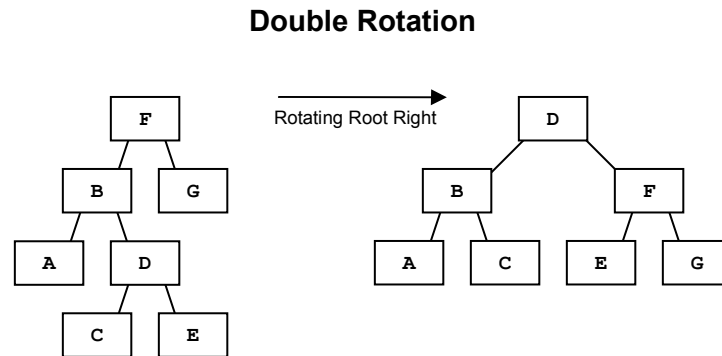


Figure 2: This figure demonstrates how a double right rotation changes the configuration of a binary tree. Double left rotations occur in a symmetric way.

As we have described the normal balancing operation, when a rotation occurs, the rotation pivot is the parent of the current iteration node. To incorporate double rotations, we test to determine whether the traversal would continue to the left or right of the current iteration node at the time when a rotation is requested. If a right rotation is being requested by the balancing operation, and if the current iteration node's value is less than the value of the search key, then a double right rotation should be performed at the pivot. If the current node's value is greater than the search key, then a normal single right rotation should be performed at the pivot. This is similar to a single step in a semi-splay operation [11]. Left rotations occur under symmetric conditions. Also, rotations are suppressed in all circumstances when a leaf would become a non-leaf. This extends the basic shuffle tree balancing technique to applications whose leafs and non-leafs must remain distinct from one another.

As more binary search trees become available whose nodes do not store balancing information, it may become feasible for the engineer to concatenate different tree algorithms together into a common code. A hybrid tree may be able to detect conditions that favor one family of algorithms over another, allowing it to dynamically change its behavior. The specific properties of such a tree require investigation.

4. Performance Tests

In this section, we discuss tests we ran to compare the performance of shuffle trees to that of splay trees and randomized binary search trees. Splay trees are the most popular self-adjusting tree. Randomized binary search trees are fast and probabilistically balanced, but not self-adjusting.

Our splay tree implementation was based on Sleator's code for top-down splaying [12]. We modified it only to insure a consistent coding style between the test data structures. Seidel and Aragon's randomized binary search tree was implemented with a hashing function to provide the necessary randomized priority value [10]. The shuffle tree was implemented without parent pointers. It used a fast implementation of the Mersenne Twister algorithm to calculate random numbers. It also used a bit-mask, rather than a modulus operation, to produce the random counter. As a result, the divisor was effectively the largest power of two, less than or equal to the tree size.

We wrote a test program which includes insertions, deletions and searches. The inner loop of the program closely resembles the following code:

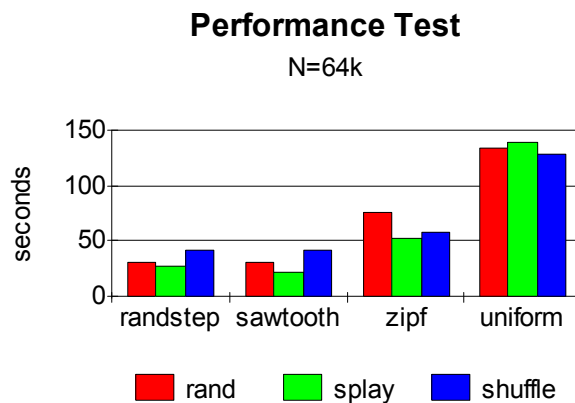
```
req = generate_requests(N);
nErasures = 0;
start_timer();
for(i=0; i<(N*8); i+=1)
{
    pFind = tree.Find( req[i] );
    if( pFind == NOT_FOUND )
        tree.Insert( req[i] );
    else
        if( nErasures < N )
        {
            tree.Erase( req[i] );
            nErasures += 1;
        }
}
stop_timer();
```

The test program executes approximately $8N$ searches, $2N$ insertions and N deletions on integer keys, where N is the size of the tree being tested. It should be noted that insertions are actually insert-on-miss and the deletions are delete-on-hit; this means that insertions and deletions immediately follow searches to the same key. This pattern favors splay trees; for example, the splay tree insertion will execute in constant time.

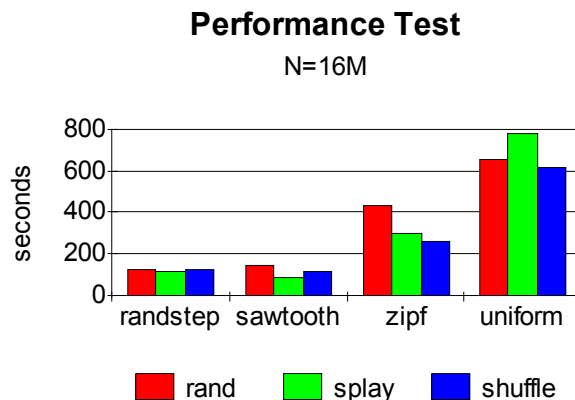
We conducted each test using four different access patterns. The “random step” pattern started at a random key value. Each subsequent key was produced by adding a random number in the range $[-\frac{\sqrt{N}}{2}, \frac{\sqrt{N}}{2}]$ to the key. The “saw-tooth” pattern started at a random key value and conducted a number of sequential accesses before restarting at another random location. The length of each sequential run was a random number in the range $[1, \sqrt{N}]$. The “zipf” pattern created weights using the Zipf probability distribution, and randomly assigned these weights to keys. As a result, the keys were accessed according to the Zipf distribution. The “uniform” pattern generated requests using a uniformly random distribution.

The program was executed using trees of 64k and 16M nodes for each of the tree types. The test was built using the Microsoft Toolkit C++ compiler and run on a 2.66GHz Xeon processor with 2GB of RAM.

The following chart summarizes the performance of the trees with a size of 64k:



The following chart summarizes the performance of the trees with a size of 16M:



The shuffle tree did not perform exceptionally well for small trees, where a large portion of the data structure can fit into the processor memory cache. It is our conjecture that the memory cache improves access for small trees to the degree that the benefit of shuffle tree self-adjustment does not exceed the costs.

With a large tree, however, the shuffle tree did not perform worse than the randomized binary search tree for any access pattern. The shuffle tree did not outperform the splay tree for the highly localized accesses of the random-step pattern and the saw-tooth pattern. This can be expected, because splay trees share the properties of finger search trees that allow them to access localized data very quickly [11].

While the shuffle tree did not always outperform other kinds of binary search trees, its performance was respectable. We believe that shuffle trees will perform well when accesses can be accurately modeled by probability distributions and when accesses to storage are expensive.

5. Conclusions

We have introduced a new binary search tree, the shuffle tree, which uses random sampling to reconfigure itself for the distribution of accesses. It has a small memory footprint and a small per-operation balancing cost. We have presented variations which may be better-suited for specific applications. We have demonstrated, through testing, that the shuffle tree performs respectably against other binary search trees.

Acknowledgements

Thanks to Tasso Lappas and Parag Patel for helpful discussion.

References

- [1] ALBERS, S. AND KARPINSKI, M. 2002. Randomized Splay Trees: Theoretical and Experimental Results. *Information Processing Letters*. Vol 81.
- [2] ALBERS, S. AND WESTBROOK, J. 1998. Self-Organizing Data Structures. In *Online Algorithms: The State of the Art*, FIAT AND WOEGINGER, Eds. Springer LNCS 1442.

- [3] ALLEN, B. AND MUNRO, I. October 1978. Self-Organizing Binary Search Trees. *Journal of the ACM*. Vol 25, No 4.
- [4] FURER, M. January 1999. Randomized Splay Trees. *Proceedings of the tenth annual ACM-SIAM Symposium on Discrete Algorithms*.
- [5] GALPERIN, I. AND RIVEST, R. January 1993. Scapegoat Trees. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete Algorithms*.
- [6] GASTON, G. December 1983. Balancing Binary Trees by Internal Path Reduction. *Communications of the ACM*. Vol 26, No 12.
- [7] JONES, D. W. August 1988. Applications of Splay Trees to Data Compression. *Communications of the ACM*. Vol 31, No 8.
- [8] MARTINEZ C. AND ROURA, S. March 1998. Randomized Binary Search Trees. *Journal of the ACM*. Vol 45, No 2.
- [9] PUGH, W. June 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*. Vol 33, No 6.
- [10] SEIDEL, R. AND ARAGON, C. 1996. Randomized Search Trees. *Algorithmica*. Vol 16, No 4/5.
- [11] SLEATOR, D. AND TARJAN, R. July 1985. Self-Adjusting Binary Search Trees. *Journal of the ACM*. Vol 32, No 3.
- [12] SLEATOR, D. March 1992. An Implementation of Top-Down Splaying.
<ftp://ftp.cs.cmu.edu/user/sleator/splaying/>.