Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Shuffle Tree: A Randomized Self-Adjusting Binary Search Tree

Mayur Patel (drone115b@gmail.com)

Research Article

Keywords: binary search trees, self-adjusting data structures, randomized algorithms, randomized data structures

Posted Date: July 11th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3136752/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Shuffle Tree: A Randomized Self-Adjusting Binary Search Tree

Mayur Patel

Toronto, Ontario, Canada.

Contributing authors: patelm@acm.org;

Abstract

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 applied randomly during each traversal, which keeps the cost of balancing activity low.

Keywords: binary search trees, self-adjusting data structures, randomized algorithms, 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 ways of maintaining them. Many binary search trees require balancing information to be stored in each node, but some trees do not [1–3]. These have a slight advantage for the software engineer, because they consume less memory.

Scapegoat trees and general balanced trees monitor traversal depth to detect poor balance [3, 4]. A node is identified under which balance is particularly bad; the node is called the scapegoat. The subtree 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 [1]. Allen and Munro presented a similar move-to-root scheme [5], 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 [6–8]. 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 [1, 9]. Others have suggested that steps can be probabilistically skipped to reduce costs [10–14]. Multi-splaying has been developed as a way of performing several, smaller splays as an alternative to a single, comprehensive operation [15–17].

Here, we introduce a new binary search tree which we call a shuffle tree. The name comes from how it applies rotations to optimize traversal. 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 numerous times results in a randomized order of playing cards, numerous shuffle tree operations result in a good node configuration.

Shuffle tree nodes do not store any balancing information and do not require parent pointers if rotations are done on-line during traversal. 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 [18]. 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. One possible implementation is given in the next section.

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[9]. 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 [19, 20]. As a result, the data in the tree behave as if they were inserted randomly, which has the property of being probabilistically balanced. 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 during each iteration. Furthermore, the tree potentially does more restructuring

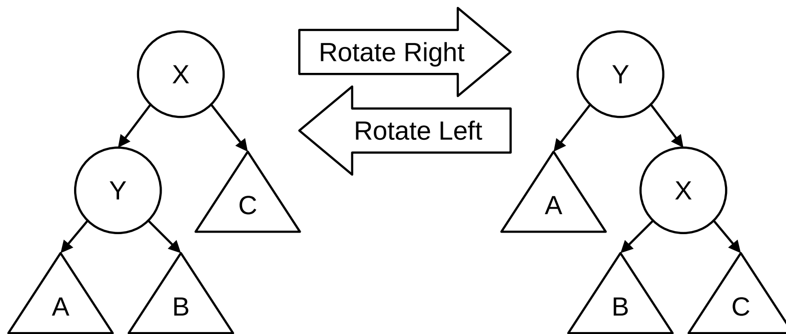

Fig. 1 The shuffle tree uses the common single rotation, depicted here, during its balancing operations.

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 [2]. 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. Bitner did something similar, tracking the number of accesses to a node and moving it up the tree above less frequently accessed data [21]. Since priorities are updated monotonically, these schemes will not react well if access patterns change. The self-adjusting scheme cannot be applied if hashing is used to produce the priority value. Different distributions for assigning the priorities have also been explored [22].

2 Algorithms

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.

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 $[0, N]$, where N is the size of the tree. As we iterate from a parent node to its child, we decrease the counter with $I \leftarrow \frac{1}{2} \cdot (I - 1)$.

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. Shuffle trees use single rotations as depicted in Figure 1.

Algorithm 1 Shuffle Tree Traveral Function.

```
Require: key k
Require: node* root
Require: int treesize
Ensure: returns node* with key==k or the leaf with the closest key to k
  signed int iCounter ← rand() modulus (treesize + 1)
  node* pRet ← 0
  node* p ← root
  while p do
    pRet ← p
    if k ¡ value(p) then
      p ← left(p)
      if (0 == iCounter) and p then
        RotateRight(pRet)
        pRet ← parent(p)
      end if
    else
      if k ¿ value(p) then
        p ← right(p)
        if (0 == iCounter) and p then
          RotateLeft(pRet)
          pRet ← parent(p)
        end if
      else
        break while
      end if
    end if
    iCounter ← (iCounter − 1) >> 1
  end while
  return pRet
```

As search depth increases, the likelihood of a rotation increases. No rotations will occur beyond depth $log_2(N)$. The counter requires $log_2(N)$ 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.

In effect, the balancing technique is a kind of random sampling. Nodes are selected randomly from traversal paths and their balance is manipulated to shorten those paths. Frequently used data attract more attention and, therefore, benefit more from balancing activity than infrequently used data. The idea of shortening traversal paths as a means of improving tree state has been used before. [23–26].

The algorithm listing (1) shows one possible implementation of a traversal function for shuffle trees. Either it finds the node associated with a given key, or the leaf node having the closest key to it. Strictly speaking, parent pointers do not need to be stored in each node, but we use them to simplify our presentation.

Methods for insertion, deletion and search can all use this traversal function to navigate the tree and to conduct balancing activity. A reference implementation of a shuffle tree in C++ is available from the author online.

3 Analysis

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.

Before determining the cost of a shuffle tree traversal, we'll clarify the terminology we will be using.

Definition 1 (Node Depth)**.** *We use $L(X)$ for node X, to refer to the number of ancestors it has in the tree. We use the term, depth, for this value. The node at the root has a depth of zero; each of its immediate children has a depth of one, and so forth. The number of comparisons required to access a node with depth d is $(d+1)$, so we use this as the cost of accessing a node.*

Definition 2 (Weight Balance)**.** *A weight-balanced tree is one where the probability of accessing the subtrees of any node is equal. The expected depth when accessing a node, X, in the tree is $L(X) \leq log_2 N$, where N is the number of nodes in the tree.*

Definition 3 (Node Weight)**.** *In a weight-balanced tree, a node X has a weight $W(X)$ defined by $W(X) = N \cdot \sum_{k \in X} P(k)$ where N is the number of nodes in the tree, and $P(k)$ is the probability of accessing node k. The summation includes node X and each of its descendants. In a weight-balanced tree, a node X having weight $W(X)$ has an expected depth of $L(X) = log_2 N - log_2 W(X)$.*

Definition 4 (Node Balance)**.** *A node, X, has a value, $\alpha(X)$, which describes its balance. Neither child of the node has a weight less than $\alpha(X) \cdot W(X)$. Node balance has a range $0 \leq \alpha(X) \leq \frac{1}{2}$.*

Definition 5 (Node Size)**.** *The size of a node, X, is given by $S(X)$. This gives the number of nodes rooted under X, including X itself.*

Lemma 1. *The expected amortized cost of a shuffle tree traversal in a weight-balanced tree has an upper bound of $2log_2 N + 2 = O(log_2 N)$.*

Proof of Lemma-1. We begin with a weight-balanced tree. A tree containing zero or one node is trivially weight-balanced.

We define a potential function, $\Phi_i = \sum_k L(k)$ for the state of the tree at time i. The summation includes all the nodes contained in the tree. Changes in its value reflect the effect of shuffle tree balancing activity on the nodes of the tree.

Due to the shuffle tree counter, the probability of choosing the root as a pivot during a shuffle tree traversal is $\frac{1}{N+1}$, a pivot at depth one is $\frac{2}{N+1}$, a pivot at depth two is $\frac{4}{N+1}$, and so forth. We can provide an upper bound with a simplification to the denominator that will be convenient later.

$$P(L(X) = d) < \frac{2^d}{N} \tag{1}$$

where X is a rotation pivot with depth d.

When a rotation occurs, some nodes in the tree become deeper and some shallower. During a right rotation as illustrated in the Figure 1, the change to the potential function can be quantified with $[W(X) - W(Y)] - [W(Y) - W(B)]$. We will only explore the right rotation in our analysis with the understanding that a left rotation is completely analogous. Given that the tree is weight-balanced, we can have bounds on the values of these node weights. For a rotation pivot X at depth d:

$$W(X) \leq (\frac{1}{2})^d N \tag{2}$$

$$W(Y) \leq \frac{1}{2} W(X) \tag{3}$$

$$W(B) \leq \frac{1}{2} W(Y) \tag{4}$$

The absolutely worst case impact for the tree would be when $W(X) = (\frac{1}{2})^d N$ and $W(Y) = 0$ and $W(B) = 0$.

Now we can assemble the individual terms to determine the expected change in potential due to a shuffle tree traversal.

$$\Delta\Phi_i = \Phi_{i+1} - \Phi_i \tag{5}$$

$$\mathbb{E}[\Delta\Phi_i] = \sum_{d=0}^{log_2 N} P(L(X) = d) \cdot \big([W(X) - W(Y)] - [W(Y) - W(B)]\big) \tag{6}$$

$$\mathbb{E}[\Delta\Phi_i] < \sum_{d=0}^{log_2 N} \frac{2^d}{N} \cdot \big((\frac{1}{2})^d N\big) \tag{7}$$

$$\mathbb{E}[\Delta\Phi_i] < \sum_{d=0}^{log_2 N} 1 = log_2(N) + 1 \tag{8}$$

Now we can combine terms to estimate the amortized cost of a shuffle tree traversal.

$\mathbb{E}[C_i] = c_i + (\Phi_{i+1} - \Phi_i)$ where C_i is the amortized cost of a single traversal and c_i is the cost of accessing a node in a weight-balanced tree, the number of comparisons executed to find the target node.

From the properties of the weight-balanced tree and the relationship between the number of comparisons and node depth:

$$\begin{aligned} \mathbb{E}[c_i] &\le log_2(N)+1 && (9)\\ \mathbb{E}[C_i] &= \mathbb{E}[c_i]+\mathbb{E}[\Delta\Phi_i] && (10)\\ \mathbb{E}[C_i] &< \big(log_2(N)+1\big)+\big(log_2(N)+1\big) = 2log_2(N)+2 && (11) \end{aligned}$$

□

Lemma 2. *In a shuffle tree, performing a rotation with node X as the pivot will change the potential of the tree by an expected value of $W(X)\big(1-\frac{3}{2}(\alpha(X)^2+(1-\alpha(X))^2)\big)$.*

Proof of Lemma-2. Let's review the change of potential in the tree as first discussed in Lemma-1. Again, we will only explore right rotations, with the understanding that left rotations are completely analogous.

The change in potential from a right rotation is $[W(X)-W(Y)]-[W(Y)-W(B)]$. Previously, we used pessimistic estimates for these values, but in practice, a shuffle tree traversal can reduce the potential if $2W(Y) > W(X)+W(B)$.

During a shuffle tree traversal, both X and Y are on the traversal path based on how pivots are selected; but whether B is on the traversal path is neither known nor relevant. With probability $\alpha(X)$, Y has a weight of $\alpha(X)\cdot W(X)$, and with probability $(1-\alpha(X))$, it has a weight of $(1-\alpha(X))\cdot W(X)$. Node B has a weight that is either $\alpha(Y)\cdot W(Y)$ or $(1-\alpha(Y))\cdot W(Y)$.

$$\begin{aligned} \mathbb{E}[W(Y)] &= W(X)\cdot\big(\alpha(X)^2+(1-\alpha(X))^2\big) && (12)\\ \mathbb{E}[W(B)] &= \frac{1}{2}\cdot W(Y) && (13) \end{aligned}$$

The expected change in potential is given by $[W(X)-W(Y)]-[W(Y)-W(B)] = W(X)\big(1-\frac{3}{2}\big(\alpha(X)^2+(1-\alpha(X))^2\big)\big)$.

□

Theorem 1. *When a shuffle tree traversal is used for searches, insertions and deletions; we expect the tree to be approximately weight-balanced.*

Proof of Theorem-1. This follows from Lemma-1 and Lemma-2.

The potential has an expected maximum increase of $(\frac{1}{4})W(X)$ when $\alpha(X)=\frac{1}{2}$ and an expected reduction of $(-\frac{1}{2})W(X)$ when $\alpha(X)=0$. The expected change in potential is zero when $\alpha(X)=(\frac{1}{2}-\frac{1}{2\sqrt{3}})\approx 0.211$. At a node that is well-balanced, a rotation will increase the potential, but shallow traversals are less likely to result in a rotation. If the working set is clustered near the root, then rotations that increase the potential will be less frequent. As $\alpha(X)$ approaches zero, rotations that decrease the potential are more frequent and more aggressive.

The definite integral tells us that, over the possible values of node balance, the total energy that would increase the potential equals the total energy that would decrease the potential. The maximum value of the potential is bounded.

$$\int_{\alpha=0}^{\frac{1}{2}} \left(1 - \frac{3}{2}(\alpha^2 + (1-\alpha)^2\right) d\alpha = 0 \tag{14}$$

□

Theorem 2. *If we use the shuffle tree traversal only for insertions and deletions, using the normal binary tree traversal for searches, then we expect the tree to be approximately height-balanced.*

Proof of Theorem-2. This follows from Lemma-1 and Lemma-2.

Each node is inserted once. Each is deleted once. The probability of accessing each node through a shuffle tree traversal is equal to $\frac{1}{N}$. In this case, the size of a node is also its weight, $W(X) = S(X)$, resulting in an approximately height-balanced tree.

□

4 Conclusions

We have introduced a new self-adjusting binary search tree, the shuffle tree, which uses random sampling to approximate a weight-balanced configuration. It is easy to implement, has a small memory footprint and has a small per-operation balancing cost. A static version of the tree, where the balancing operation only occurs during insertions and deletions, approximates a height-balanced tree.

A topic for future research would be to determine the shuffle tree's behavior under more complex access patterns; for example, when the probability of accessing a node depends upon previous operations performed on the tree.

Statements and Declarations

Competing Interests

The author did not receive support from any organization for the submitted work. The author certifies that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

[1] Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM **32**(3), 652–686 (1985) https://doi.org/10.1145/3828.3835

[2] Aragon, C.R., Seidel, R.: Randomized search trees. In: IEEE Symposium on Foundations of Computer Science, vol. 30, pp. 540–545 (1989)

[3] Galperin, I., Rivest, R.: Scapegoat trees. In: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (1993)

[4] Andersson, A.: General Balanced Trees. Journal of Algorithms **30**(1), 1–18 (1999) https://doi.org/10.1006/jagm.1998.0967

[5] Allen, B., Munro, I.: Self-organizing binary search trees. In: 17th Annual Symposium on Foundations of Computer Science (sfcs 1976), pp. 166–172 (1976). https://doi.org/10.1109/SFCS.1976.26

[6] Pfaff, B.: Performance analysis of bsts in system software. In: SIGMETRICS '04/Performance '04 (2004)

[7] Huus, E.: Reduced restructuring in splay trees (2014)

[8] Lee, E.K., Martel, C.U.: When to use splay trees. Software: Practice and Experience **37**(15), 1559–1575 (2007)

[9] Albers, S., Westbrook, J.: Self-organizing data structures. In: Fiat, A., Woeginger, G.J. (eds.) Online Algorithms: The State of The Art, pp. 13–51. Springer, Berlin, Heidelberg (1998). https://doi.org/10.1007/BFb0029563 . https://doi.org/10.1007/BFb0029563

[10] Fürer, M.: Randomized splay trees. In: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 903–904 (1999)

[11] Albers, S., Karpinski, M.: Randomized splay trees: theoretical and experimental results. Information Processing Letters **81**(4), 213–221 (2002)

[12] Lai, T.W., Wood, D.: Adaptive heuristics for binary search trees and constant linkage cost. SIAM Journal on Computing **27**(6), 1564–1591 (1998) https://doi.org/10.1137/S0097539793250329 https://doi.org/10.1137/S0097539793250329

[13] Aho, T., Elomaa, T., Kujala, J.: Reducing Splaying by Taking Advantage of Working Sets. In: McGeoch, C.C. (ed.) Experimental Algorithms, pp. 1–13. Springer, Berlin, Heidelberg (2008)

[14] Derryberry, J.C., Sleator, D.D.: Skip-Splay: Toward Achieving the Unified Bound in the BST Model. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) Algorithms and Data Structures, pp. 194–205. Springer, Berlin, Heidelberg (2009)

[15] Wang, C.C., Derryberry, J., Sleator, D.D.: O(log log n)-competitive dynamic binary search trees. In: ACM-SIAM Symposium on Discrete Algorithms (2006)

[16] Derryberry, J., Sleator, D., Wang, C.C.: Properties of multi-splay trees. Technical Report Technical report CMU-CS-09-171, Carnegie Mellon University, Pittsburg, PA, USA (2009)

[17] Wang, C.C.: Multi-splay trees. PhD thesis, Carnegie Mellon University, Pittsburg, PA, USA (2006)

[18] Sleator, D.: An implementation of top-down splaying. https://www.link.cs.cmu.edu/link/ftp-site/splaying/top-down-splay.c (1992)

[19] Roura, S., Martínez, C.: Randomization of search trees by subtree size. In: Díaz, J., Serna, M.J. (eds.) Algorithms - ESA '96, Fourth Annual European Symposium, Barcelona, Spain, September 25-27, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1136, pp. 91–106. Springer, ??? (1996). https://doi.org/10.1007/3-540-61680-2_49 . https://doi.org/10.1007/3-540-61680-2_49

[20] Martínez, C., Roura, S.: Randomized binary search trees. J. ACM **45**(2), 288–323 (1998) https://doi.org/10.1145/274787.274812

[21] Bitner, J.R.: Heuristics that dynamically organize data structures. SIAM Journal on Computing **8**(1), 82–110 (1979) https://doi.org/10.1137/0208007

[22] Tarjan, R.E., Levy, C., Timmel, S.: Zip trees. ACM Trans. Algorithms **17**(4) (2021) https://doi.org/10.1145/3476830

[23] Cho, S., Sahni, S.: A new weight balanced binary search tree. International Journal of Foundations of Computer Science **11** (1970) https://doi.org/10.1142/S0129054100000296

[24] Vinod, P., Pushpa, S., Maple, C.: Maintaining a random binary search tree dynamically. In: Tenth International Conference on Information Visualisation (IV'06), pp. 483–488 (2006). https://doi.org/10.1109/IV.2006.72

[25] Cheetham, R.P., Oommen, B.J., Ng, D.T.H.: Adaptive structuring of binary search trees using conditional rotations. IEEE Transactions on knowledge and data engineering **5**(4), 695–704 (1993)

[26] Cheetham, R.P., Oommen, B.J., Ng, D.T.H.: On using conditional rotation operations to adaptively structure binary search trees. In: Gyssens, M., Paredaens, J., Van Gucht, D. (eds.) ICDT '88, pp. 161–175. Springer, Berlin, Heidelberg (1988)