One-Time Binary Search Tree Balancing: The Day/Stout/Warren (DSW) Algorithm

Timothy J. Rolfe
Computer Science Department
Eastern Washington University
Cheney, Washington 99004-2412 USA
<Timothy.Rolfe@ewu.edu>

Abstract

A. Colin Day proposed, and Quentin F. Stout and Bette L. Warren modified, an algorithm (the Day/Stout/Warren or DSW algorithm) that, in O(N) time and O(1) space, transforms an arbitrary binary search tree into a degenerate tree, and from that generates the most balanced possible binary search tree.

1. Introduction

I would like to present an algorithm for consideration based on its elegance. Some data structures texts give a method for transforming a binary search tree into the most compact possible form through writing the tree's contents out to a file and then reading the file so as to generate a balanced binary tree [1]. This obviously requires additional space (within the file) proportional to the number of elements in the tree. It is possible, though, to accomplish the same thing without that overhead.

Dynamic methods of balancing of binary search trees have been around at least since Adel'son-Velskii and Landis proposed the AVL tree [2]. In 1976, A. Colin Day proposed an algorithm to accomplish the balancing of a binary search tree *without* all the overhead of the AVL tree [3]. He used an existing method to transform a binary search tree into a degenerate tree (a linked list through the right child pointers with null pointers for all the left child pointers), and followed that with a controlled application of one of the rotations used in the maintenance of an AVL tree. The result was a balanced tree — one in which all paths from the root to leaf nodes differ in length by at most one. Alternatively, it is a tree in which every level, except possibly the bottommost level, is fully occupied. (Thus, we are using the term in a more stringent sense here than is found when it is used for dynamically balanced trees such as the AVL tree. In the AVL tree, the balance condition is that the heights of every node's subtrees differ at most by one.)

Ten years later, Quentin F. Stout and Bette L. Warren [4] proposed an improvement on the over-all algorithm, noting that we could replace the first phase of Day's original algorithm with an alternative method very similar to the second phase of Day's algorithm.

2. Day's Original Algorithm

Day developed his algorithm in Fortran, which at the time did not support recursion, records, and pointers. Consequently pointers were implemented by means of arrays of subscripts: ILB(K) would be the subscript for the left child to node K, and IRB(K) would be the subscript for the right child. In addition, to eliminate the need for recursion, he generated what we call a "threaded" binary tree, with back-pointers to control the traversal, eliminating the need for a recursive implementation in the traversal. These back-pointers used the same pointer array as the right-child pointers: A positive number gave the subscript for the right child node, while a negative number indicated the subscript of the next node to backtrack to in the traversal.

The first phase of Day's algorithm was to transform the binary search tree into a linked list—which he referred to as the backbone—using what he called "a well-known and described" method. This was accomplished by doing an in-order traversal of the tree, stripping nodes out of the tree as the nodes are visit and generating a sorted list of nodes as a queue, adding the nodes stripped from the tree to the back of the list.

Consistent with the programming style of the time, Day developed his explicit code using a number of conditional and unconditional branches. We represent the spirit of this first phase in the following C++ code segment. However, it does take advantage of recursion, records (classes), and pointers. Since a stack discipline is easier to implement, it actually traverses the binary search tree in reverse, and pushes nodes onto the list being generated—thus generating the list as forward-sorted. (Note that this requires passing a pointer by reference for the `List` generated in stack fashion.)

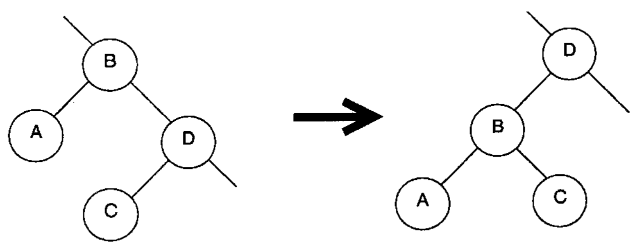
```
// Reverse in-order traversal, pushing nodes
// onto a linked list (through the ->Right link)
// during the traversal.  Tree passed through "node"
// is destroyed.
// NOTE earlier:  "typedef struct BaseCell* BasePtr;"

void DSW::Strip( BasePtr node,
                 BasePtr &List, int &size )
{
   if ( node != NULL ) {
```

Reviewed Papers

```
      //Retain pointer to left subtree.
    BasePtr Left = node->Left;
      //For degenerate tree, all left pointers = NULL
    node->Left = NULL;
      //Reversed; hence right, middle, then left.
    Strip ( node->Right, List, size );
      //Stack discipline in building the list
      node->Right = List;
      List = node;
      size++;           // Count the number of nodes
      Strip ( Left, List, size );
   }
}
```

Once we generate the degenerate tree, we transform it into a balanced tree by doing repeated leftward rotations such as we use in maintaining an AVL tree. In such a rotation, the tree retains its structure as a binary search tree, but links are altered so that the focus node moves down to the left by one position, and its right child moves up into its location. The subtree that lies between these two nodes shifts from leftward of the right node to rightward of the left node. Figure 1, taken from Day's paper [5], the tree is rotated left at node B, so that B moves down by one position, node D



The transformation

moves up, and node C changes sides.
Figure 1: Leftward Rotation

The crucial question for the algorithm is: How many times do we apply the transformation within each pass? We control this by adjusting the length of the backbone. We may state the algorithm in words as follows:

1. Reduce the length of the backbone by 1
2. Divide the length of the backbone by 2 [rounding down if the length is not even] to find the number of transformations, *m*.
3. If *m* is zero, exit; otherwise perform *m* transformations on the backbone.
4. Return to 1. [6]

Stout and Warren did the translation of Day from Fortran into Pascal, which we translated into C++. Rather than "backbone", Stout and Warren refer to the degenerate tree as a "vine", hence the naming "vine_to_tree". (Their implementation also uses a "pseudo-root", comparable with a dummy header cell for a linked list.)

```
void DSW::compression ( BasePtr root, int count )
{
   BasePtr scanner = root;

   for ( int j = 0; j < count; j++ )
   {
      //Leftward rotation
      BasePtr child = scanner->Right;
      scanner->Right = child->Right;
      scanner = scanner->Right;
      child->Right = scanner->Left;
      scanner->Left = child;
   }  // end for
}     // end compression

// Loop structure taken directly from Day's code
void DSW::vine_to_tree ( BasePtr root, int size )
{
   int NBack = size - 1;
   for ( int M = NBack / 2; M > 0; M = NBack / 2)
   {
      compression ( root, M );
      NBack = NBack - M - 1;
   }
}
```

3. Algorithm as Revised by Stout and Warren

Stout and Warren made a major change in the first phase of the algorithm (generation of the degenerate tree), and introduced a small change in the second phase. They noted that we could also perform the generation of the degenerate tree by means of rotations, simply using the rightward rotation during the first phase. Their implementation avoids special handling for the tree root by having a pseudo-root, so that all transformations occur farther down in the tree. Starting from the right child of the pseudo-root (that is, the actual tree root itself), rightward rotations are performed until the left child link is null. Then the focus for these rotations moves one step to the right. Figure 2 comes from their paper [7].

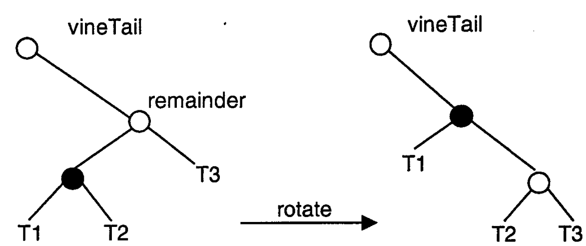


Figure 2: Rightward Rotation

Here is the translation of their Pascal into C++.

```
// Tree to Vine algorithm: "pseudo-root" is passed
// comparable with a dummy header for a linked list.
void DSW::tree_to_vine ( BasePtr root, int &size )
{
   BasePtr vineTail, remainder, tempPtr;

   vineTail = root;
   remainder = vineTail->Right;
   size = 0;
```

```
    while ( remainder != NULL )
    {
       //If no leftward subtree, move rightward
       if ( remainder->Left == NULL ) {
          vineTail = remainder;
          remainder = remainder->Right;
          size++;
       }
       //else eliminate leftward subtree by rotations
       else {              // Rightward rotation
          tempPtr = remainder->Left;
          remainder->Left = tempPtr->Right;
          tempPtr->Right = remainder;
          remainder = tempPtr;
          vineTail->Right = tempPtr;
       }
    }
}
```

In addition to this change, providing symmetry to the two phases, they also made a minor change to Day's second phase. They recognized that one can generate not just a balanced tree, but also a complete tree, one in which the bottommost tree level is filled from left to right. The only change is to do a partial pass down the backbone/vine such that the remaining structure has a vine length given (for some integer *k*) by 2^k–1. In the following code segment, the discovery of the size of the full binary tree portion in the complete tree has been explicitly coded — Stout and Warren represent it in one-line pseudo-code fashion. The `compression` function is the same as in the discussion of the second phase in Day's original algorithm, and so is not repeated here.

```
int FullSize ( int size )
{                // Full portion of a complete tree
   int Rtn = 1;
   while ( Rtn <= size )      // One step PAST FULL
      Rtn = Rtn + Rtn + 1;    // next pow(2,k)-1
   return Rtn/2;
}

void DSW::vine_to_tree ( BasePtr root, int size )
{
   int full_count = FullSize (size);
   compression(root, size - full_count);
   for ( size = full_count ; size > 1 ; size /= 2 )
      compression ( root, size / 2 );
}
```

4. Discussion

A program [8] tested implementations of Day's original algorithm, Stout and Warren's modification, and Robert Sedgewick's [9] alternative algorithm (see endnote 1) for one-time binary search tree balancing, as well as the AVL tree. Explicit average times were captured for building the tree with random data and emptying it without any balancing (to provide a baseline to subtract, allowing capture of just the balancing time), as well as building the tree, balancing it, and then emptying it. They simply built and emptied the AVL tree, of course. The following code segment shows the code to capture the baseline, flagging where the tree balancing would go.

```
srand(Seed);
                      // Get the same start
                      // to the rand() sequence
Start = clock();
for (Pass = 0; Pass < Npasses; Pass++)
{
   for ( k = 0; k < N; k++ )
      Tree.Insert ( rand() );
      // NO tree balancing to get base line;
      // else “Tree.balanceXX();”
   Tree.Empty();
}

T0 = 1000 * double(clock()-Start) /
     CLOCKS_PER_SEC / Npasses;
```

The results are as expected. Figure 3, over-all processing time, as expected for a binary search tree with random data, shows some upward curvature (given the expected N log N behavior).

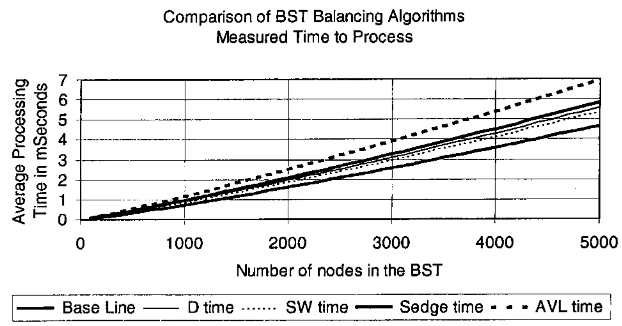


Figure 3: Over-All Processing Time

Figure 4 shows the data once the base line time (tree creation and emptying) is subtracted, leaving just the tree balancing time (and comparable information for the AVL tree). The balancing time, as expected, appears to be linear. There is more noise, of course, since the data are showing differences.

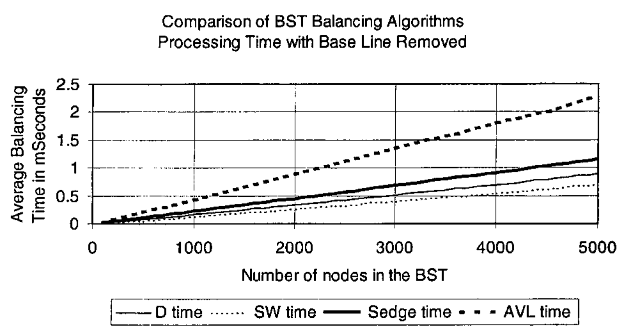


Figure 4: Time Required to Balance

These data show that Stout and Warren's change in the first phase of the algorithm actually generated a speed-up in processing. On the other hand, Robert Sedgewick's algorithm (based on rotating the k^{th} nodes to root positions in the tree and its subtrees) did require more time. As expected, building the full-blown AVL tree required significantly more time.

As noted in the introduction, the term "balanced" as used for the result of the DSW algorithm is more stringent than the balance condition for the AVL tree. One might ask how closely the AVL tree approaches the compactness of the DSW-generated binary search tree. It has been proven that the worst-case root height for an AVL tree is bounded above by approximately $1.44 \log_2(N+2)$, [10] while the root height of a DSW-generated tree is $[\log_2(N+1)]$. One might ask, though, both how the root height of the *average* AVL tree compares with the DSW-generated tree — which identifies the worst-case node depth and thus the worst case number of comparisons to find or fail to find an item — and how the node depths differ on average. Figure 5 shows the results of capturing both average root height and average node depth for a large range of binary search trees. The average AVL root height has some interesting oscillations tied to the powers of 2 [11].

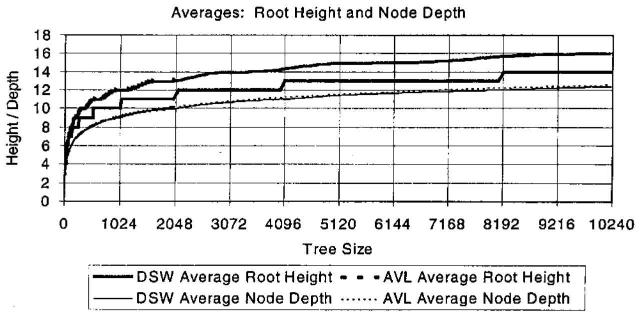


Figure 5: Average Root Height and Node Depth

While the average root height of the AVL tree is noticeably larger than that for the DSW-generated tree, the average node depth is nearly the same. Figure 6 shows how the AVL tree differs from the DSW-generated tree in terms of the *percentage* difference of the average root height and the average node depth. (As can be seen, we collected more data points for trees of size 10 through 2000 than for trees from 2100 through 10200.)

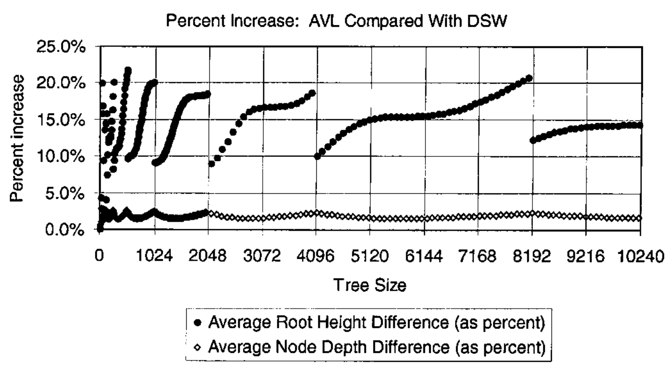


Figure 6: Percent Increase: AVL Compared with DSW

The DSW algorithm for tree balancing would be useful in circumstances in which one generates an entire binary search tree at the beginning of processing, followed by item look-up access for the rest of processing. Regardless of the order of data as we build the tree, the resulting tree converts into the most efficient possible look-up structure as a balanced tree.

The DSW algorithm is also useful pedagogically within a course on data structures where one progresses from the binary search tree into self-adjusting trees, since it gives a first exposure to doing rotations within a binary search tree. We find these rotations in AVL trees and various other methods for maintaining binary search trees for efficient access (such as red-black trees).

Acknowledgements

The computations shown in the discussion section were performed on equipment purchased under the State of Washington Higher Education Coordinating Board grant for the Eastern Washington University Center for Distributed Computing Studies.

I am grateful for discussions of this material when I presented it to the January meeting of the Inland Northwest Chapter of the Association for Computing Machinery (ACM). In particular, I would like to thank Professor Steve Simmons for raising the issue of how well an AVL tree satisfies the more stringent balancing seen in a tree balanced by the DSW algorithm.

References

[1] Frank M. Carrano and Janet J. Prichard, *Data Abstraction and Problem Solving: Walls and Mirrors* (third edition; Addison-Wesley, 2002), pp. 552-554. They present here a variant of the binary search algorithm to read sorted data into a balanced binary search tree.The DSW algorithm is covered in Adam Drozdek's text (which is where I first encountered it) immediately before his treatment of AVL trees. Adam Drozdek, *Data Structures and Algorithms in C++* (second edition; Brooks/Cole, 2001), pp. 255-258. Robert Sedgewick, on the other hand, has a very interesting rotation-based balancing algorithm derived from an algorithm that partitions a binary search tree, moving the k^{th} entry to the root. Robert Sedgewick, *Algorithms in C*, Vol. I (third edition; Addison-Wesley, 1998), p. 530-531.

[2] G. M. Adel'son-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Dockl.*, 1962.

[3] A. Colin Day, "Balancing a Binary Tree", *Computer Journal*, XIX (1976), pp. 360-361. This makes reference to A. Colin Day, *Fortran Techniques, With Special Reference to Non-Numerical Applications* (Cambridge University Press, 1972), pp. 76-81.

[4] Quentin F. Stout and Bette L. Warren, "Tree Rebalancing in Optimal Time and Space", *Communications of the ACM*, Vol. 29, No. 9 (September 1986), p. 902-908. The full text is available on-line for ACM members through <http://www.acm.org/pubs/contents/journals/cacm/> — navigate to Vol. 29, No. 9.

[5] Day (1976), p. 360.

[6] Day (1976), p. 360.

[7] Stout and Warren, p. 904.

[8] The code for this program is available through the following URL:<http://penguin.ewu.edu/~trolfe/DSWpaper/Code.html>

[9] Sedgewick, p. 530.

[10] Mark Allen Weiss, *Algorithms, Data Structures, and Problem Solving with C++* (Addison-Wesley Publishing Company, 1996), p. 565. See also Drozdek, p. 259, which also quotes a result from Knuth to the effect that the average is $\log_2(N)$ + 0.25, significantly less than the worst case.

[11] For a more extended discussion of the AVL tree, see Timothy J. Rolfe, "Algorithm Alley: AVL Trees", *Dr. Dobb's Journal*, Vol. 25, No. 12 (December 2000), pp. 149-152.