Building optimal binary search trees from sorted values in O(N) time.

Jean G. Vaucher, professeur titulaire
Departement d'informatique et de recherche opérationnelle, Université de Montréal,
C.P. 6128, Succursale Centre-Ville, Montréal, Canada, H3C 3J7
Email: vaucher@iro.umontreal.ca

Summary

First, we present a simple algorithm which, given a sorted sequence of node values, can build a binary search tree of minimum height in O(N) time. The algorithm works with sequences whose length is, a priori, unknown. Previous algorithms [1-3] required the number of elements to be known in advance. Although the produced trees are of minimum height, they are generally unbalanced. We then show how to convert them into *optimal* trees with a minimum internal path length in O(log N) time. The trees produced, both minimum height and optimal, have characteristic shapes which can easily be predicted from the binary representation of tree size.

Key Words: binary search tree, balanced trees, data structures

Introduction

The binary search tree (BST) is a well known data structure: it is a binary tree with the property that the value of any given node is larger than the node values in its left sub-tree and smaller than the values in its right sub-tree. Figure 1 shows two such trees. In this figure and in what follows, we assume integer values for the nodes.

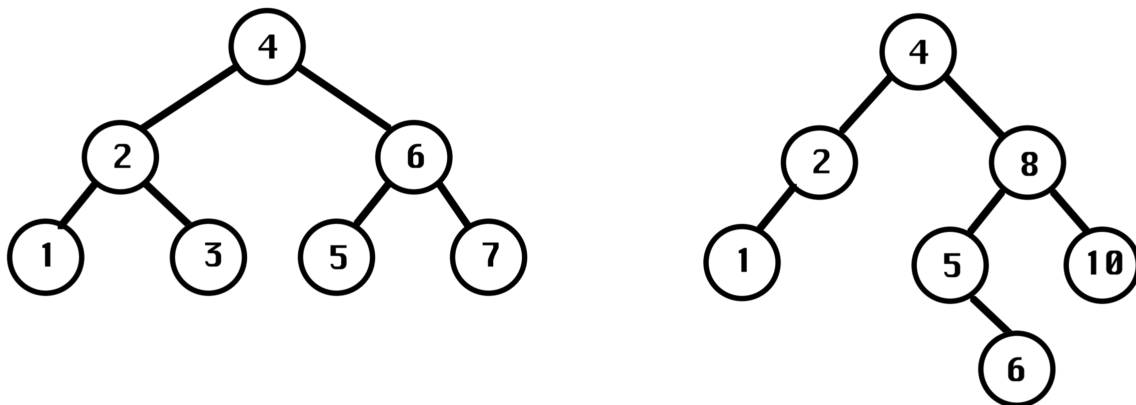

Figure 1 - Binary search trees

The *height* of the tree is an important factor in the analysis of tree algorithms. In this paper the *height*, h, is defined to be the number of nodes on the longest path from the root

to a leaf. The height of the tree on the left is 3 and the height of the tree on the right is 4. The tree on the left with all the leaves on the bottom level is said to be *perfect*. It contains exactly 2^h-1 nodes. More generally, the minimum height for a tree containing N nodes is $\lceil \log_2(N+1) \rceil$. Conversely, the maximum height is N if the tree has degenerated into a list.

Any modern text on data structures describes the properties of BSTs and gives the basic algorithms to find, add or remove an element. Given random values, the time complexity of these basic operations is O(log n); furthermore, the values from a binary search tree can be output in sorted order in O(n) time by a simple recursive algorithm shown below.

```
class Node {
   vType value;
   Node  left, right;
}

void printTree ( Node p)
{  if p ≠ null then
   {  printTree (p.left) ;
      print (p.value) ;
      printTree (p.right) ;
   }
}
```

Now consider the inverse operation, namely: building a tree given a sequence of sorted values such as produced by printTree. Of course, this can be accomplished by successive INSERT operations but the complexity of this approach is O(N log N) at best. Actually, if the basic insertion algorithm is used with a sorted set of values, the tree degenerates into a list and the complexity is $O(N^2)$.

In 1976, Wirth [1, p. 195] gave an efficient algorithm to construct a tree of N nodes. The algorithm is recursive: a tree of N nodes is built by reading a node value and doing recursive calls to build two sub-trees of (N-1)/2 nodes. The resulting tree is perfectly balanced and the running time is O(N), but the value of N must be known before hand. Wirth was not concerned with node order and the tree was arbitrarily read in *pre-order*. More recently, Carrano [2, p. 480, 3, p. 545], gives a similar algorithm which works with sorted values but again N must be known beforehand. The method is shown below.

```
Node buildTree ( int N )
{   if N = 0 then
        return null ;
    else
    {   Node  left  := readTree ( N / 2 ) ;
        vType value := nextValue () ;
        Node  right := readTree ( (N-1) / 2 ) ;
        return new Node( value, left, right ) ;
    }
}
```

In what follows, we develop an algorithm which does the same thing as Carrano's but does not require prior knowledge of N. This algorithm is presented in four steps. First, we start with a simple function which works for *perfect* trees such as shown at the left of Figure 1. This function needs a parameter: *h*, the height of the tree. Second we note, that with a test for end of file, the simple function still builds a minimum height tree for any value *h* greater or equal to the correct value. Thirdly, we add a driver routine which builds successively bigger perfect trees until the end of data is reached. This driver function requires no height parameter and works in O(N) time. The tree that is built is of minimum height and supports the usual operations in logarithmic time but it may not have an optimal shape. Finally, we show how to modify the tree to achieve minimum internal path length with O(log N) *rotation* operations.

The pseudo-code used for the programming examples is based on Java with some Simula (Algol) notation for clarity. The Simula influence can be seen in the use of "=" for equality and ":=" for assignment and the **if...then... else...** syntax**.** As in C and Java, we assume that parameters are passed by value and that **return** exits immediately. Finally, in order to make the algorithms match the text and easy to follow, we have used more variables and code than strictly necessary.

Reading a perfect tree

Given 2^h-1 ordered nodes values, the function readTree1 -shown below- builds a perfect tree of height h. It assumes that the input contains exactly the right number of values and does not check for premature end of data.

```
Node readTree1 ( int h )
{   if h < 1 then
        return null ;
    else
    {   Node  left  := readTree1 ( h-1 ) ;
        vType value := nextValue () ;
        Node  right := readTree1 ( h-1 ) ;
        return new Node( value, left, right ) ;
    }
}
```

Handling premature end of data

The next version of the input function, readTree2, is identical to the first except that we add a test to stop construction when there is no more data to read. We assume that the `end_of_data` test comes before reading and that the test can be repeated without error even after it has returned `true`. The algorithm still needs the expected tree height, h, as a parameter but it stops creating nodes as soon as the input values are all read.

```
Node readTree2 ( int h )
```

```
{  if h < 1 then
     return null ;
   else
   {   Node left  := readTree2 ( h-1 ) ;
     if end_of_data() then return left ;
       vType value := nextValue () ;
       Node  right := readTree2 ( h-1 ) ;
       return new Node( value, left, right ) ;
   }
}
```

As long as the data is not exhausted, all trees returned by `readTree2` will be perfect. Since the algorithm proceeds in order building the left sub-tree before the node, the fact that we are able to read a node value implies that *end_of_data* was not encountered during the construction of the left sub-tree/ it is therefore perfect; but the same cannot be said for right sub-trees.

Figure 2 shows what happens, when the initial value of h is correct or larger than strictly necessary but there are fewer than 2^h-1 nodes and *end_of_data* is encountered while building the tree. In the example, there are only 5 node values but the function is called with an expected height h = 4. The shaded part shows the virtual perfect tree of height 4 (with 15 nodes) which could have been returned by the read function. When there are fewer nodes, the algorithm traverses this virtual tree in the usual order building nodes with successive input value until the *end_of_data* is reached. Essentially, it fills in the bottom left-hand corner of the virtual tree. In terms of execution time, overestimating the tree height means that we visit the extra virtual nodes between the root and the actual nodes built.

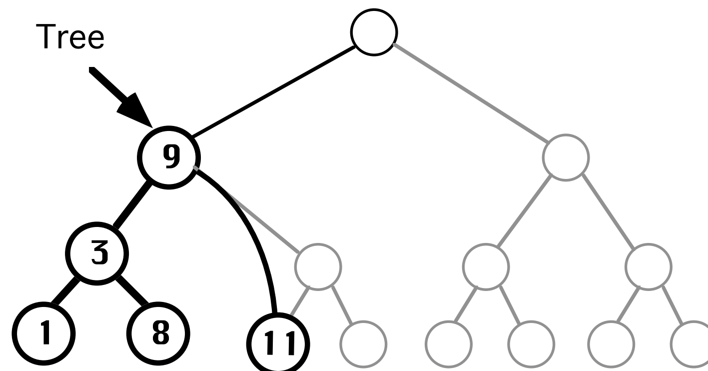

Figure 2 - Tree returned by readTree2(4) from values 1, 3, 8, 9 and 11

Essentially, the algorithm tries to build successively taller perfect trees until all node values have been read. As long as the initial value of $h \geq \log_2(N+1)$ all nodes will be read and the returned tree will be of minimum height but it may be unbalanced. The left sub-tree of the root - being perfect - contains exactly $2^{h"-1}-1$ nodes (where h" is the actual height of the tree) but the right sub-tree, which contains the remaining nodes, may contain anywhere from 0 to $2^{h"-1}-1$ nodes.

Dispensing with the estimated height parameter

To avoid specifying an initial value for the tree height, *h*, we use a loop which calls readTree2 with successively larger values of *h* to build trees of increasing size until the end of data is reached.

```
Node readMinTree ()
{
    int  h    := 0;
    Node tree := null;
    while not end_of_data() do
    {
        Node  left := tree;
        vType root_value := nextValue () ;
        Node  right := readTree2 ( h ) ;
        tree  := new Node(root_value, left, right ) ;
        h ++ ;
    }
    return tree ;
}
```

In the function: *tree* is the last tree that was built and its height is *h*. While there are nodes to read, *tree* is a perfect tree containing exactly 2^h-1 nodes. The next larger tree (of height h+1) uses the old tree as its left sub-tree. The next node value is read for the root and we call readTree2 to build a new right sub-tree of the same height as the left. If data is still not exhausted, the resulting tree is again perfect and we repeat the process until end of data is reached.

Initially, we start with h=0 and an empty tree. When the loop terminates, the tree built by this function has the same shape as described for readTree2: minimum height and a full left sub-tree but a right sub-tree containing anywhere from zero to 2^{h-1}-1 nodes.

Essentially the algorithm does a recursive traversal of the tree that it builds and its time complexity is O(N).

The shape of returned trees

The trees returned by readMinTree have a definite characteristic shape. While the left sub-tree of the root is perfect, the right sub-tree contains at most the same number of nodes as the left and generally fewer so the right half of the tree is generally not as deep as the left. The same reasoning applies to right sub-trees, which will either be balanced or skewed to the left. Overall, the tree shape, as shown, below could be characterized as a *staircase*.

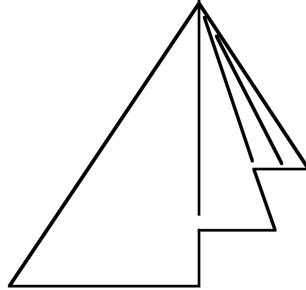

Figure 3 - Characteristic shape of the trees constructed by readMinTree

We can get a more precise idea of the tree topology by noting that our tree is composed of a succession of perfect trees that decrease in size as they are built from left to right and we can think of the nodes on the right edge of the tree as a list connecting these perfect trees together. This view will make it easy to determine the shape of the tree from the binary coding of N, the tree size. Consider the perfect tree of height h shown below. It contains $2^h–1$ nodes but, if we add the extra *link* node at the top, the structure accounts for 2^h nodes.

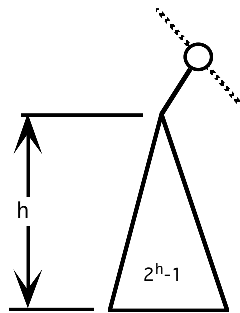

Figure 4 - Perfect tree component

Thus there is a direct correspondence between the perfect trees in our structure and the *ones* in the binary representation of the tree size. For example, consider a tree of 37 nodes: "100101_2" in binary. This corresponds to a tree shown below with 3 linked perfect trees whose sizes correspond to the powers of 2 that add up to 37: 32 + 4 + 1. The shape is shown in Figure 5.

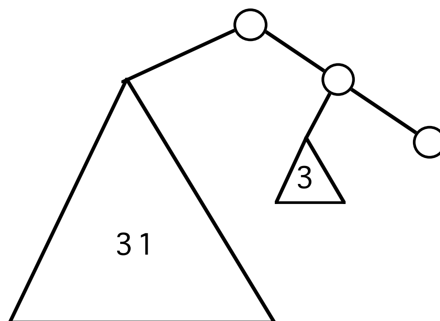

Figure 5 - Staircase of size 37

Improving the tree

The time complexity of traditional search tree algorithms is strongly dependent on the tree height. In the case of our staircase trees, the height is a minimum and equal to $\lceil \log_2(N+1) \rceil$ so that standard operations can be done in O(log N) time. This height is indicative of worst case operation and compares favorably to balanced AVL trees where the height in the worst case is 1.44 log(N+2) [5, p.118].

However, for complexity in the average case, the staircase shape is not optimal. Consider the extreme case when the number of nodes is a power of 2. The tree that we return is shown on the left in Figure 6. In such a case, the right sub-tree is empty and all nodes are in a perfect tree on the left. A better disposition of the nodes is shown on the right where all nodes in the left sub-tree have been moved up by one level and the root has been moved to the bottom.

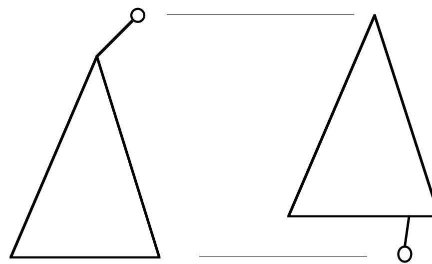

Figure 6 - Optimizing a staircase tree

The tree on the right is an example of an *optimal* search tree: one in which all levels, except possibly for the bottom one, are completely filled. In such a tree, there is no empty slot closer to the root into which a node from the bottom could be moved up and the *total path length*, the sum of the distances between each node and the root, is minimized. Note that the improvement in average path length to be gained from modifying our trees will be marginal - at most one level - as shown in figure 6. This is because staircase trees are already of minimum height and represent intermediate stages between perfect trees whose heights differ by one.

In what follows, we will show how to modify our staircase tree into an optimal shape in *h* steps, where *h* is the tree height. To do this we will use rotations - operations commonly used in AVL trees [4-6] - which move one sub-tree up closer to the root while moving another sub-tree down; all the while keeping the tree height constant. If the number of nodes going up is greater than the number going down, the rotation improves the average path length.

In Figure 7 below, the tree on the left is typical of the staircase trees that we produce. Here X is the root with a perfect sub-tree to its left and a smaller tree (R) to its right. The root Y of the left sub-tree has two equal size (perfect) sub-trees labeled L (left) and M (middle). As a result of a rotation, the old root X and its imperfect right sub-tree R are moved down one level. Y and its sub-tree L move up one level. Y becomes the new root. The middle tree, M, is now tied to X instead of Y, but in terms of distances of its nodes to

the (new) root, nothing has changed and it can be thought of as a fixed pivot upon which the other sub-trees balance. Note that the rotation does not change the tree height and it maintains the order between nodes and sub-trees: L < Y < M < X < R. The important effect is that L moves up whereas the smaller R moves down.

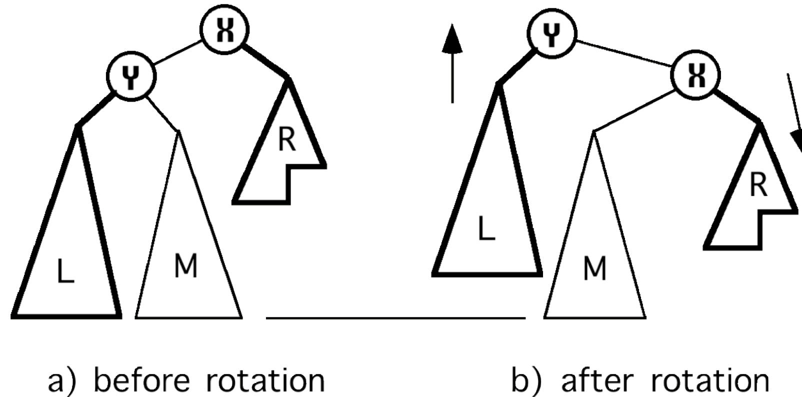

Figure 7 - Effect of a rotation

Another way to consider the effect of the rotations is to note that they modify the tree towards the optimal shape by bringing down the smaller imperfect sub-tree (R) on the right until its bottom layer lines up with the bottom of the tree. One rotation may not be enough. In the example of Figure 7, after the first rotation, R could still move down and we would do further rotations on X. With each rotation, R is attached to a lower point in the main (left) tree. When R reaches the bottom, we can start applying rotations to R itself with a view of bringing its right half in line with its left. Now for every tree on the right that goes down, a tree on the left must go up but you will note in what follows that the trees that move up (like L) start on the bottom and go up by only one level. Thus these promotions do not destroy the optimal shape.

Knowing N, the number of nodes in the tree[1], we can compute the number of nodes in the various sub-trees and decide if a rotation is warranted:

- nodes in the whole tree:	N
- height of the tree:	$h = \lceil \log_2(N+1) \rceil$
- height of tree rooted at Y :	$hL = h-1$
- size of tree rooted at Y :	$2^{h-1} - 1$
- size of R :	$nR = N-(2^{h-1}- 1)-1 = N - 2^{h-1}$
- height of R:	$hR = \lceil \log_2(nR+1) \rceil$

An example will clarify the situation. In figure 8, we consider the optimization of the staircase tree shown previously. Here, in the initial situation (a), N=37 and h=6, the heights of the trees on the left and right are 5 and 3 respectively. A rotation is warranted and the original root along with the right sub-tree moves down one level as shown in (b)

---

[1] These should be counted by the input function nextValue and made available in a global variable.

[the arrow always the shows the tree under consideration]. Now, hR remains at 3 but hL = 4 and a further rotation brings us to (c). At this point, the right sub-tree is level with the bottom of the tree and further rotation of the original right sub-tree is no longer beneficial.

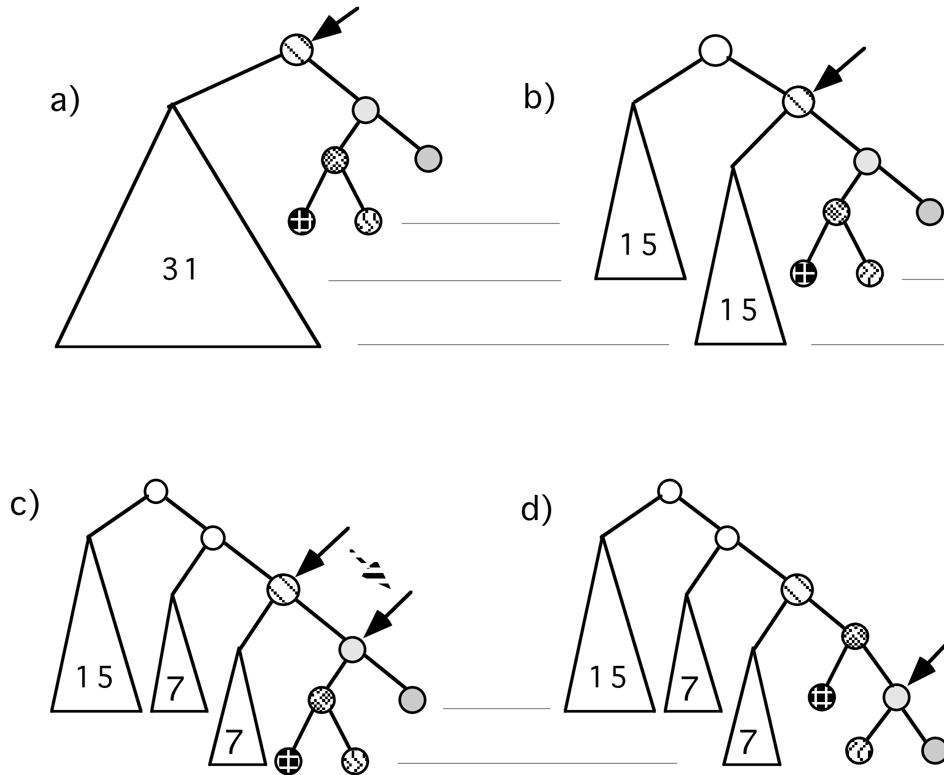

Figure 8 – Optimizing a staircase tree

At this point, we skip the rotation but still go down a level to the right to see if rotations *within* the original right sub-tree could be beneficial. Now we have N=5, hL=2, nR=hR=1. A final rotation brings us to (d) where the rightmost leaf being on the bottom, the work is finished. At each step in the optimization process we go down one level, thus the complexity is O(h) = O(log N).

Going a step further, we can understand the shape of the final tree obtained in (d) above by considering that a perfect tree of the same height would have contained 26 more nodes than our 37-node tree. In an optimal tree, the missing nodes must come from the bottom layer: they are the nodes that would have been leaves below our *promoted* trees. The way we promoted trees was to start with the largest on the left, reducing the size by a factor of two at each step. Therefore, the gaps from right to left in the bottom layer correspond to the binary representation of the missing nodes. In our case: 16 + 8 + 2 = 26 as shown in Figure 9. When comparing figures 8 and 9, remember that the bottom layer of a perfect tree with 2^h-1 nodes contains 2^{h-1} nodes and that the (missing) layer below that would contain 2^h nodes.

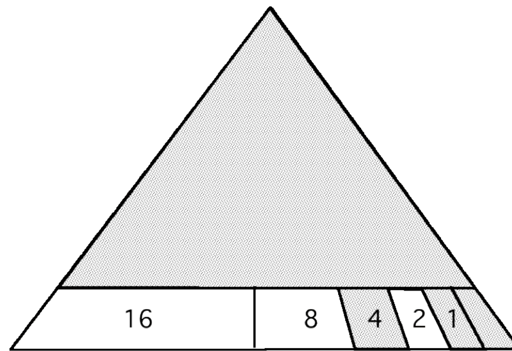

Figure 9 – Layout of an optimized tree

The optimizing algorithm

The code below implements the technique that we have outlined with a slight improvement. In the example of Figure 8, we showed how a sub-tree several levels above the bottom could be moved down with a sequence of rotations; however, it is more efficient to find the node on the right edge below which the sub-tree will eventually be placed and do a single rotation at that point.

```
1     Node optimize ( Node root, int N, int h )
2     {
3       if N <= 1 then return root;
4
5         int  hL = h-1 ;
6         int  nR = N - 2**(h-1);
7         int  hR = ceiling( log(nR+1));
8
9         Node newRoot = root;
10        if hL > hR then
11        {
12            Node leftTree = newRoot = root.left;
13            hL = hL-1;
14            while hL > hR do
15            {
16               leftTree = leftTree.right;
17               hL = hL-1;
18            }
19            root.left = leftTree.right;
20            leftTree.right = root;
21        }
22        root.right = optimize ( root.right, nR, hR );
23        return newRoot;
24     }
```

The method takes 3 parameters: `root`, a pointer to the root of the staircase tree to be optimized; `N`, the tree size and `h` the tree height. It works recursively: on each call, it optimizes a tree by deciding if the right sub-tree should be moved down and if so does the demotion; it then proceeds to optimize the right sub-tree. The algorithm terminates when the tree has shrunk down to a single node (line 3). In the code, we use more significant names, `root` and `leftTree`, to denote the X and Y nodes of Figure 7. Initially (lines 5-6), we compute the size of the right sub-tree, nR, as well as the heights of both sub-trees, `hL` and `hR`. `newRoot` represents the root of the optimized tree; it is initialized to `root`, the value that will be returned if no rotation is done. If the right tree is shorter than the left (line 10), then the right tree will be moved down and lines 12-18 determine where it will be inserted. At the same time the value of `newRoot` is changed. The *rotation* to insert the right sub-tree lower in the left Tree is done in lines 19-20. Whether, the right sub-tree has been moved or not, we optimize it (line 22) and return the optimized tree (line 23).

We show below the final version of readTree which combines the optimization with the initial tree building to meet the stated objective: constructing an optimal search tree from a sorted set of values.

```
int N;  // number of nodes - incremented by nextValue

Node readTree ()  // final version
{
    int  h    := 0;
    Node tree := null;
    while not end_of_data() do
    {
       Node  left := tree;
       vType root_value := nextValue () ;
       Node  right := readTree2 ( h ) ;
       tree := new Node(root_value, left, right ) ;
       h ++ ;
    }
    return optimize( tree, N, h );
}
```

A Java test version of this algorithm is available on the Internet at the following URL:
`http://www.iro.umontreal.ca/~vaucher/Pubs/BST.java`

Conclusions

We have developed a simple algorithm which, given a sorted sequence of node values, can build a balanced binary search tree in O(N) time, without requiring *a priori* knowledge of the number of elements, N. The novel idea is that a minimum height tree can be constructed by trying to build successively deeper perfect trees, using the tree from the last step as the left sub-tree of the new one. It is then a simple - though tricky - matter to reshape the tree with rotations to minimize internal path length. We also showed that the shape of the trees had a simple one-to-one correspondence to the binary representation of the tree size.

This algorithm could also be used to re-balance an arbitrary tree. Given a language with coroutines (like Simula [7]), we could emulate Ole-Johan Dahl's technique from his classic 1972 paper with Tony Hoare [8]: using one coroutine object to recursively traverse the old tree and provide input for another coroutine using our algorithm to build a better tree. With a current language like Java, rebalancing could still be done but would be less elegant.

I met Ole-Johan Dahl along with his colleague Kristen Nygaard in the early 1970s on a visit to Oslo to learn more about Simula. The concern with rigor and clarity as well as innovation evident in their work over the years has been a continuing source of inspiration and I grieve their passing.

Acknowledgements

I wish to thank the anonymous referee whose comments contributed to a significant improvement in the original paper. In particular, he prodded me into further reflection into the optimization phase of the algorithm leading to the discovery of the simple relations that determine tree shapes. Finally, the referee's suggestion of using tree height instead of size to control the optimization lead to simplification in both the code and the explanation.

References

[1] N. Wirth, *Algorithms + data structures=programs*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

[2] F. M. Carrano and J. J. Prichard, *Data abstraction and problem solving with Java : walls and mirrors*, 1st ed. Boston: Addison-Wesley, 2001.

[3] F. M. Carrano, P. Helman, and R. Veroff, *Data structures and problem solving with Turbo Pascal : walls and mirrors*. Redwood City, Calif. ; Don Mills, Ont.: Benjamin/Cummings Pub. Co., 1993.

[4] G. M. Adelson-Velskii and E. M. Landis, “An Algorithm for the Organization of Information,” *Soviet Mathematics (translated from Doklady Akademii Nauk, SSSR)*, vol. 3, pp. 1259-1263, 1962.

[5] M. A. Weiss, *Data Structures and Algorithm Analysis in Java*: Addison-Wesley, 1999.

[6] D. E. Knuth, *The art of computer programming: Sorting and searching*., vol. 3. Reading, Mass.,: Addison-Wesley Pub. Co., 1973.

[7] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, “SIMULA-67 Common Base Language,” Norwegian Computer Centre, Oslo, Norway, Technical Report 1970.

[8] O.-J. Dahl and C. A. R. Hoare, “Hierarchical Program Structures,” in *Structured Programming*, vol. 8, *A.P.I.C. Studies in Data Processing*. London: Academic Press, 1972, pp. 175-220.