Binary Search Tree Balancing Methods: A Critical Study

Suri Pushpa[1], **Prasad Vinod**[2]

[1]Dept. of Computer Science and Applications, Kurukshetra University, Haryana, India
[2]Dept. of Technology, Majan University, Sultanate of Oman.

Summary
Binary search tree is a best-suited data structure for data storage and retrieval when entire tree could be accommodated in the primary memory. However, this is true only when the tree is height-balanced. Lesser the height faster the search will be. Despite of the wide popularity of Binary search trees there has been a major concern to maintain the tree in proper shape. In worst case, a binary search tree may reduce to a linear link list, thereby reducing search to be sequential. Unfortunately, structure of the tree depends on nature of input. If input keys are not in random order the tree will become higher and higher on one side. In addition to that, the tree may become unbalanced after a series of operations like insertions and deletions. To maintain the tree in optimal shape many algorithms have been presented over the years. Most of the algorithms are static in nature as they take a whole binary search tree as input to create a balanced version of the tree. In this paper, few techniques have been discussed and analyzed in terms of time and space requirement.

Key words:
Binary Search Tree; Tree Balancing; Dynamic Balancing; Static Balancing

1. Introduction

Binary search tree is most basic, nonlinear data structure in computer science that can be defined as “a finite set of nodes that is either empty or consists of a root and two disjoint subsets called left and right sub-trees. Binary trees are most widely used to implement binary search algorithm for the faster data access. When memory allocation is static and data size is reasonably small, an array may be used instead to accomplish the same task. However, for large data set array is not a good option since it requires contiguous memory that system may not provide sometimes. In ideal situation, we would expect the tree to be of minimal height that is possible only when the tree is height balanced. With a *n* node random binary search tree search time grows only logarithmically *O(lg(n))* as size of input grows. A binary search tree requires approximately 1.36*(lg (n))* comparisons if keys are inserted in random order. It is also a well-known fact that total path length of a random tree can be further reduced by 27.85 percent by applying some rebalancing mechanism. There are two methods to rebalance a binary tree, dynamic rebalancing, and global (static) rebalancing. Both methods having advantages and disadvantages. Dynamic methods to create a balance binary search tree have been around since Adel’son-Velskii & Landis [1] proposed the AVL Tree. Dynamic rebalancing methods maintain a tree in optimal shape by adjusting the tree whenever a node is inserted or deleted. Examples of this approach are height-balance tree, weight-balance tree, and B-trees. Rather then readjusting the tree every now and then global or static rebalancing methods allows the tree to grow unconstrained, and readjustment is done only when such a need is arises. To achieve this task many computer scientist have proposed various solutions. Following is a comparison on various proposed solutions. For the interested readers algorithms are given in the appendix.

2. AVL Algorithm

Soviet Mathematicians G. M. Adel’son-Vel’skii & E. M. Landis [1] proposed an algorithm to create a balanced binary search tree dynamically. Every node in the tree has to maintain additional information (apart from data and pointers) called “balance factor” that stores the effective balance of the tree rooted at that node. Tree is said to be balanced if the difference between the heights of two sub-trees of any node (balance factor) is between –1 and 1. Mathematically, –1 <= balance factor <=1. After each operation tree has to be examined to ensure that it is balanced. If the tree has become unbalanced appropriate rotation is performed in the appropriate direction.
AVL algorithm [1] balances a tree dynamically that means prior knowledge of the size of input is not required. In addition to that, it does not require much space during execution. On the downside, the tree has to be examined and adjusted accordingly after each insertion and deletion making it slow during execution. Furthermore, extra memory is required in every node to

maintain additional information called “balance factor”. In addition to that, sometimes, double rotation may be required making whole process a complex task.

3. Martin and Ness’s Algorithm

Martin & Ness [5] developed an algorithm to balance a binary tree globally. Algorithm takes an arbitrary binary search tree as input and creates a perfect balanced tree. They used a stack to traverse the tree to determine the order of nodes and then repeatedly dividing N (number of nodes) by two to get the median. In this way, number of nodes in the both sub-trees would differ by one making it optimal in shape. Rather than readjusting tree after each operation they adopted static method that was totally different from AVL [1]. In the process, they used extra space in form of stack and array making algorithm to be space inefficient. Time complexity of the algorithm is linear *O(n),* as each node of the tree has to be visited. Major problem with this approach is to determine “when maintenance work has to applied” as there is no deterministic method to predict when tree will become unbalanced and when there will be a need of readjustments.

4. Day’s Algorithm

A. Colin Day [4] proposed an algorithm in Fortran to rebalance a binary search tree statically. Since Fortran do not support recursion a threaded tree had to be used to traverse the tree without using stack. “Threaded” binary trees, are supported by back-pointers to perform the traversal and thus eliminating the need of recursion. Right pointer of every node whose right sub-tree is empty used as a thread pointer that points to it’s in-order successor, making it possible to move upward in the tree without using stack.
Day’s algorithm consists of two phases. In first phase, tree is traversed in in-order to visit every node, creating a backbone (linked list). The final linked list was a sorted list of items as in-order traversal results in sorted output. In second phase, degenerated tree is transformed into a balanced tree by performing a series of left rotations. Day used a threaded tree to traverse the tree therefore some additional information has to be maintained in each node to distinguish a thread pointer from actual pointer. Day’s algorithm is not very demanding as far as space requirement is concern during execution time. However, time complexity of algorithm is *O(n),* still it is very efficient algorithm if perfect balance is not required.

5. Chang and Iyengar’s Algorithm

Chang & Iyengar [3] proposed an algorithm to balance a binary search tree globally. Tree is recursively traversed to collect all the pointers in the array. When the tree is traversed in in-order output is sorted collection of keys in ascending order. After traversal, the array is recursively partitioned into two parts left and right with each part differing in one key at most. Left partition creating left sub-tree and right partition right sub-tree. During each partition median has to be determined that becomes the root of left and right partition. Algorithm is not space efficient since it requires all the pointers to be copied into an array doubling the space requirement. Array needs contiguous memory that system may not provide sometimes particularly when data size is quite large. Since each node of the tree has to be visited, Algorithm runs in *O(n)* time.

6. Stout and Warren’s Modification

Stout & Warren [7] proposed improvement over-Day’s algorithm. Stout and Warren made some changes in existing Day’s algorithm [4].
The first phase of the algorithm converts a tree into a vine (each parent node has only a right child and the nodes are in sorted order) proceeding top down through the tree, creating an initial portion, that has been transformed, and a remaining portion of nodes with larger keys, which may require further transformation. In second phase, a balanced tree is created by applying a series of compressions to the degenerated tree obtained from the first phase. They used a simple binary search tree rather then a threaded tree that is a major advantage over the Day’s Algorithm. In addition to that, algorithm needs only constant space to store temporary variables and not much demanding in terms of run-time space requirement. The beauty of their algorithm is neither they used stack or recursion to convert the tree into some intermediate form (“vine” in their terminology). However, time complexity remains same as before that is *O(n).*

7. Sleater and Tarjan’s Splay Trees

Sleator & Trajan [6] invented splay tree a self-adjusting tree, where it is guaranteed that amortize cost of a sequence of operations like insertion and deletion will be *O(lg(n)).* Amortized time; the time per operation averaged over a worst-case sequence of operations. Thus, splay trees are as efficient as balanced trees when total running time is the major concern rather than the cost of individual operation. To reduce total access time frequently accessed items should always be near to the root. They invented a simple method of restructuring a tree so that after each access accessed item moves closer

to the root assuming, this item will be accessed again in near future (Principle of locality). According to Principle of locality any item that is accessed now, it is highly likely that same item will be accessed in near future. The restructuring technique used in splay trees is called splaying. Splaying moves a node that is recently accessed towards the root of the tree by performing a sequence of rotations along the original path from the node to the root. Their work has shown that in amortized sense splay trees are as efficient as both dynamically and statically balanced trees.
Following table figure 1 summarizes the performance of various algorithms in terms of time and space requirement.

	Time	Space
Martin & Ness	*O(N)*	Required stack to carry out the traversal
A. Colin Day	*O(N)*	Little run time space is required to hold temporary variables but tree has to be threaded
Chang & Ayengar	*O(N)*	Additional workspace required equal to the size of the tree.
Stout & Warren	*O(N)*	Only fixed amount of space is required.

Table 1

8. Conclusion

Although it is known that if input is random, we will be closer to a balanced tree. Still some balancing technique is required to prevent the tree from becoming higher on one side resulting after a series of insertions and deletions. Ultimate goal is to maintain the tree in such way that its height is always *O(lg(n))* so that all basic tree operations could be performed in *O(lg(n))* time. Many techniques for tree balancing have been developed over the years and some of them have been discussed and analyzed in this paper. Most of the algorithms are static in nature and taking time linear to the size of input, and in addition to that, in few cases significant amount of space is required. Run time overhead for static algorithms is certainly less in compared to the dynamic algorithms but there has to be some predefined time interval to rebalance the tree. All the balancing algorithms presented here (apart from AVL algorithm [1]) are made to run in two phases. First phase converts arbitrary binary tree into some intermediate tree and in the next phase, intermediate tree is converted into a balanced tree. In order to create intermediate tree, each node of the tree has to visited resulting run time complexity to be *O(N)*. No algorithm has been developed so far that could balance a tree in lesser time, consequently, there is a huge scope of improvement over existing methods as they are lacking in one aspect or other.

References

[1] Adel'son-Vel'skii, G.M., and Landis, E.M., 1962, An Algorithm for the Organization of information. *Soviet Mathematics Doklady*, 3, pp.1259–1263.

[2] Bayer, R., 1972, Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica,* 1, pp. 290–306.

[3] Chang, H., and Iyengar S.S., July 1984, Efficient Algorithms To Globally Balance a Binary Search Tree, *Communication of the ACM* 27, 8, pp. 695-702.

[4] Day, A. C., 1976, Balancing a Binary Tree, *Computer Journal*, XIX, pp. 360-361.

[5] Martin, W.A., and Ness, D.N., Feb 1972, Optimal Binary Trees Grown with a Sorting Algorithm*. Communication of the ACM* 15, 2, pp. 88-93

[6] Sleator, D.D., and Tarjon R. E., July 1985, Self-Adjusting Binary Search Trees. *Journal of The ACM*, 32(3), pp. 652-686.

[7] Stout, F., and Bette, L. W., September 1986, Tree Rebalancing in Optimal Time and Space, *Communication of the ACM*, Vol. 29, No. 9, pp. 902-908.

Vinod Prasad has Master's degrees in Computer Science and Mathematics. At present, he is pursuing his PhD in Computer Science. His area of research is Algorithms and Data Structure where he is working on Binary search tree data structures. Vinod has published and presented a number of papers in national and international journals, and conference proceedings.

Appendix

Martin and Ness's Algorithm:

```
procedure BALANCE(ROOT, LSON, RSON, n):
    integer ROOT, n; integer array LSON, RSON;
    begin
    {T holds pointer to node to be visited during traversal,
    STACK is used to store T, TOP points to the top element of STACK}
    integer T, TOP, ANS; integer array STACK(I:n);
    {Traverse the input tree and return a node pointer
    in ascending key order via ANS}
    procedure TRAVNEXT:
        begin
            if T ≠ null
                then begin
                    TOP← TOP + I;
                    STACK(TOP)← T;
                    T← LSON(T);
                    TRAVNEXT;
                    end;
            else begin
                    ANS ← STACK(TOP);
                    TOP← TOP- 1;
                    T← RSON(ANS);
            end;
        end;
        {Restructure pointers by partitioning}
        procedure GROW(N):
            integer N; {number of elements in a subset }
            begin
            {T is root of a balanced subtree reflected by a subset,
            LPTR is used to temporarily store the LSON of T}
            integer T, LPTR;
            case
                N = 0 {null branch}:
                    begin
                        ANS← null;
                    end;
                N = 1 {leaf}:
                    begin
                        TRAVNEXT;
                        LSON(ANS), RSON(ANS) ← null;
                    end;
                N > 1 {divisible subset}:
                    begin
                        GROW(L(N-1)/2); {form left subtree}
                        LPTR ← ANS;
                        TRAVNEXT;
                        T ← ANS;
                        GROW((N-1)/2); {form right subtree}
                        RSON(T)  ←ANS;
                        LSON(T) ←  LPTR
```

```
                                                    ANS ← T;
                                            end;
                            end;
        end;
        if n ≤ 2 then return;
        T← ROOT;
        TOP← 0; {initialization}
        GROW(n);
        ROOT ←ANS; {new root}
end;
```

Day’s Algorithm

```
procedure BALANCE(ROOT LSON, RSON, n):
        integer ROOT, n; integer array LSON, RSON;
        begin
                integer T, LAST_VISITED, L, R, M, BACKBONE_LENGTH;
                {stripe nodes off right-threaded tree to form backbone. T points
                to the node to be visited, LAST_VISITED holds pointer to the last
                visited node}
                if n ≤ 2 then return;
                T← ROOT;
                while LSON(T) ≠ null do {find the first node}
                        begin
                                T ← LSON(T);
                        end;
                ROOT ← T; {root of backbone}
                LSON(ROOT) ← null;
                LAST_VISITED ← T;
                T ← RSON(T);
                while T≠ null do {traverse}
                        begin
                                if T > 0
                                        then begin
                                                while LSON(T) ≠ null do
                                                        begin;
                                                                T← LSON(T);
                                                        end;
                                                end;

                                else T ← -(T); {backtrack}
                                RSON(LAST_VISITED) ← T; {chain together}
                                LASTVISITED ← T;
                                LSON(T) ← null;
                                T ← RSON(T);
                        end;
                        {restructure backbone to a balanced tree. M is the number of
                         transformations needed in a pass, T now points to node to be
                         shifted out of the backbone, L is the left ancestor of T, R is the
                         right son of T, BACKBONE_LENGTH is the number of nodes
                         in the backbone}
```

```
                              BACKBONE_LENGTH ← n - 1;
                              M ← LBACKBONE_LENGTH/2;
                              while M > 0 do {transform}
                                     begin
                                            T ← ROOT; {move on ROOT in anticipation}
                                            ROOT ← RSON(ROOT);
                                            RSON(T) ← LSON(ROOT);
                                            LSON(ROOT)  ←T;
                                            T← RSON(ROOT);
                                            L← ROOT;
                                            for I ←2 to M by 1 do {shift}
                                                   begin
                                                          R ← RSON(T);
                                                          RSON(L) ← R;
                                                          RSON(T) ← LSON(R);
                                                          LSON(R) ← T;
                                                          T ← RSON(R);
                                                          L ← R;
                                                   end;
                                     BACKBONE_LENGTH ← BACKBONE_LENGTH - M - I;
                                     M ← LBACKBONE_LENGTH/2;
                                     end;
       end;
```

Chang and Iyengar's Algorithm

```
{globally balance a binary search tree through folding}
procedure BALANCE(ROOT, LSON, RSON, n):
       integer ROOT, n; integer array LSON, RSON;
       begin
              integer N, M, ANSL, ANSR; integer array LINK(I : n);
              {traverse the original tree and set up LINK}
              procedure TRAVBIND(T):
                     integer T; {pointer to the next node to be visited}
                     begin
                            if T = null then return;
                            TRAVBIND(LSON(T));
                            N ← N + 1; {count the sequence of visit}
                            LINK(N) ← T; {store the pointer to the Nth node
                                          in the Nth element of LINK }
                            TRAVBIND(RSON(T));
                     end;
                     {reorganize a tree by partitioning and folding}
                     procedure GROW(LOW, HIGH):
                            integer LOW, HIGH;
                            begin
                                   {MID is the median of a subset bound by LOW and HIGH,
                                   TL is the subtree root in the balanced left half-tree,
                                   TR is the counterpart of TL in the right half-tree.
                                   TL, TR are returned via ANSL, ANSR, respectively}
```

```
                                **integer** MID, TL, TR;
                                **case**
                                    LOW > HIGH {null branch  }:
                                        **begin**
                                            ANSL, ANSR ← null;
                                        **end;**
                                    LOW = HIGH {leaf } :
                                        **begin**
                                        ANSL← LINK(LOW);
                                        ANSR ← LINK(LOW+M);
                                        LSON(ANSL), RSON(ANSL) ←  null;
                                        LSON(ANSR), RSON(ANSR) ← null;
                                        **end;**
                                    LOW < HIGH {divisible subset}:
                                        begin
                                        MID ← (LOW + HIGH)/ 2;
                                        TL ← LINK(MID);
                                        TR ← LINK(MID+M);

                                    GROW(LOW, MID-I);{form left subtree }
                                    LSON(TL)  ←ANSL;
                                    LSON(TR) ← ANSR;
                                    GROW(MID+1,HIGH);form rightsubtree}
                                    RSON(TL) ← ANSL;
                                    RSON(TR) ←ANSR;
                                    ANSL ← TL;
                                    ANSR ← TR;
                                **end;**
                            **end;**
                        **end;**
                    **if** n ≤ 2 then return;
                    N  ← 0; {initialize counter}
                    TRAVBIND(ROOT);
                    M ← (N + I)/ 2; {folding value}
                    ROOT ←LINK(M); {new root}
                    **if** N = 2 * M
                        **then** {N is even }
                            **begin**
                                M ← M + I; {adjust folding value}
                                GROW(1,M-2);
                                {put the node associated with M as a terminal node
                                left to its immediate successor}
                                LSON(LINK(M)), RSON(LINK(M)) ← null;
                                LSON( LINK( M+ 1) ) ← LINK(M) ;
                            **end;**
                    **else** {N is odd} GROW(1,M-1);
                LSON(ROOT) ← ANSL;
                RSON(ROOT) ← ANSR;
**end;**
```