Programming Techniques and Data Structures

Ellis Horowitz Editor

Efficient Algorithms to Globally Balance a Binary Search Tree

HSI CHANG and S. SITHARAMA IYENGAR

ABSTRACT: A binary search tree can be globally balanced by readjustment of pointers or with a sorting process in $O(n)$ time, n being the total number of nodes. This paper presents three global balancing algorithms, one of which uses folding with the other two adopting parallel procedures. These algorithms show improvement in time efficiency over some sequential algorithms [1, 2, 7] when applied to large binary search trees. A comparison of various algorithms is presented.

1. INTRODUCTION

A binary search tree is organized such that for any node, all keys in the left subtree are smaller and those in the right subtree are greater than the key value of that node. It provides a method of data organization which is both flexible and efficient.

The formation of binary trees grown in a random manner may have some branches longer than others. We know a binary tree search will, on the average, require $(2(n+1)/n)H_n - 3$ ($\approx 1.386 \log_2 n$) comparisons for a tree with n nodes if the keys are inserted into the tree in random order. Thus, by making extra efforts to maintain a perfectly balanced tree instead of a random tree, we could—provided all keys are looked up with equal probability—expect an average improvement of approximately 27.85 percent in the search path length [3, 7].

The task of balancing a binary search tree is then to adjust the left and right pointers of all the nodes in the tree so that the search path is optimized; we can attain the same effect as a binary search without requiring a sorted list. Algorithms that dynamically restore tree structure to balance during insertion or deletion of nodes have been described by Adelson-Velskii, Landis, and Knuth [3, 6]. However, in this paper, we deal with global algorithms that balance the entire tree at one time. A global tree balancing algorithm generally runs in linear time and consists of two parts: first, a traversal to determine the order of all nodes, then restructuring pointers based on that order.

Martin and Ness [7] developed an algorithm that reorganizes a tree with n nodes by repetitively subdividing n by 2 and uses the results as guidance to step through the framework of a perfectly balanced tree, i.e., for each node, the number of nodes in its left subtree and right subtree differ by 1 at most. An in-order traversal is carried out concurrently to provide relative node positions for the pointer-restructuring procedure which fits them into proper places in the balanced tree structure. In this algorithm, a stack is required to save pointers during traversal.

Bentley [1] discusses an algorithm to produce perfectly balanced trees. In this algorithm, nodes are passed as a set or a linked list to the balancing procedure which finds the median element of the set of nodes as root and splits the set into two subsets, each forming a balanced subtree at the next level. This process continues until there are no more elements to split.

Another algorithm that needs no extra storage was given by Day [2]. The input tree is required to be right-threaded with negative backtrack pointers to allow

```
Algorithm S:
{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(1: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-1);  {form left subtree}
              LSON(TL) ← ANSL;
              LSON(TR) ← ANSR;
              GROW(MID+1,HIGH); {form right subtree}
              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 + 1)/ 2⌋;  {folding value}
    ROOT ←LINK(M);  {new root}
    if N = 2 * M
       then {N is even}
            begin
              M ← M + 1;  {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;
```

FIGURE 1. Algorithm S.

stackless traversal. A right-skewed tree is formed after traversal and this list-like structure serves as a backbone for the pointer-restructuring procedure which shifts nodes to the left side of the backbone until the subtree path lengths balance, i.e., no path (from root to leaf) differs in length from any other path by more than 1. The produced tree, on the other hand, is not threaded although it can be made a threaded tree with some modification. Martin and Ness's algorithm and Day's algorithm are given as tutorial matter on pages 700–701 for interested readers. For a broader treatment on balancing binary search trees, see [4–6, 8]. These include the present investigation of the authors.

In the next section, we present a new sequential balancing algorithm which also uses partitioning but reduces the number of times of partitioning through folding. Section 3 presents two parallel algorithms as variations of the previous sequential algorithm; and in Section 4, we analyze and compare these algorithms.

2. SEQUENTIAL BALANCING THROUGH FOLDING

The node of a tree is assumed to contain the following fields: left subtree pointer LSON, right subtree pointer RSON, and KEY, while ROOT points to the root node of the tree. We first traverse the original tree to determine the order of all n nodes. Array LINK is used to store the ordering information so that the ith element of LINK contains a pointer to the ith node in ascending key order, i.e., KEY(LINK($i-1$)) < KEY(LINK(i)) < KEY(LINK($i+1$)).

To reorganize a tree by partitioning the set, following Bentley, we first find the left median,[1] $\lfloor (n+1)/2 \rfloor$, of the whole set; the node corresponding to that element is to be the root of the new tree. The remaining $\lfloor (n-1)/2 \rfloor$ elements with values less than the left median in the left subset and the other $\lceil (n-1)/2 \rceil$ elements in the right subset form two partitions. At the next level, we find for each of these two subsets their medians and use the two associated nodes as roots of the two corresponding subtrees as well as LSON and RSON, respectively, of the previous root. This process continues to find the medians at each level, partitions around them, and restructures the tree from top to bottom and from left to right.

Further considering the use of LINK, we observe that for trees containing an odd number of nodes, the median of an ordered set also serves as a special value, denoted by M, to let us construct the right half-tree concurrently with the left half-tree because the counterpart elements in the balanced two halves differ by M. If we know the position of an element K in the left half-tree, we can also determine the position of its counterpart element $K + M$ in the right half-tree. So, we only have to partition half of the entire set in order to reorganize the tree structure.

We call this procedure *folding* because it is like something bending over upon itself to form a symmetrical structure. The value M used in folding is called the *folding value.* When the total number of nodes is even, we let the right median, $n/2 + 1$, be the folding value M. The node associated with the left median remains as the root of the tree and the node corresponding to M is placed as the leftmost node in the right half-tree.

Algorithm S (Figure 1) is a global balancing algorithm based on this idea. Procedure TRAVBIND traverse the original tree and sets up the auxiliary array LINK. After traversal, procedure GROW recursively partitions and forms the balanced left half-tree with its right-half counterpart built in the same step through folding. Two parameters are used in GROW: LOW, the lower bound, and HIGH, the upper bound for an ordered subset to be relinked in balanced form. The root of a balanced subtree is the left median MID of the subset. Three conditions direct the course of GROW:

1. If LOW > HIGH, we have a null subtree; return a null pointer.
2. If LOW = HIGH, we have a leaf; set up a terminal node and return the node pointer.
3. If LOW < HIGH, we have a divisible subset; do the following:
   a) find the root T through MID.
   b) GROW left subtree and return root as LSON of T.
   c) GROW right subtree and return root as RSON of T.

A tree is balanced when the ROOT of the tree, and the LSON and RSON of each node are readjusted.

3. PARALLEL BALANCING

Balancing a tree through folding reveals parallelism in the pointer-restructuring procedure. It is possible to divide the set of nodes into subsets and perform reorganizing operations on these parts simultaneously. One way to balance the left half- and right half-tree at the same time is given in Algorithm P1 (Figure 2). In this algorithm, folding is not used and the pointer-restructuring procedure GROW directly resembles Bentley's scheme [1].

Another degree of parallelism exists between traversal and pointer restructuring. If we physically rearrange all nodes in sorted order during traversal, the pointer-restructuring procedure becomes a pointer generator which needs only to know the size of the tree to compute all the LSONs and RSONs for corresponding nodes in the balanced tree structure. A copy of the original tree has to be made for traversal and sorting so that nodes could be regenerated without destroying useful information. Algorithm P2 (Figure 3) describes the way to exploit this parallelism.

4. A COMPARISON OF PERFORMANCE MEASURES

Three conditions, when there are 0, 1, or more than 1 element in a subset, dictate the course of the pointer-restructuring procedure in our algorithms. Take Algo-

[1] If we have n sorted elements $X_1, \ldots, X_n$, the median is $(X_{\lfloor (n+1)/2 \rfloor} + X_{\lfloor n/2+1 \rfloor})/2$, the left median is $X_{\lfloor (n+1)/2 \rfloor}$, and the right median is $X_{\lfloor n/2+1 \rfloor}$.

```
Algorithm P1:
{build balanced left half-tree and right half-tree in parallel}
procedure BALANCE(ROOT,LSON,RSON,n):
  integer ROOT, n; integer array LSON, RSON;
  begin
    integer N; integer array LINK(1: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}
    procedure GROW(LOW,HIGH):
      integer LOW, HIGH;
      begin
        {MID is the median of a subset bound by LOW and HIGH,
         T is the root of the balanced subtree reflected by
         the subset}
         integer MID, T;
         case
           LOW > HIGH  {null branch}: return(null);
           LOW = HIGH  {leaf} :
             begin
               T ← LINK(LOW);
               LSON(T), RSON(T) ← null;
               return(T);
             end;
           LOW < HIGH  {divisible subset} :
             begin
               MID ← ⌊(LOW + HIGH)/ 2⌋;
               T ← LINK(MID);
               cobegin  {form left and right subtrees in parallel}
                 LSON(T) ← GROW(LOW,MID-1);
                 RSON(T) ← GROW(MID+1,HIGH);
               coend;
               return(T);
             end;
         end;
      end;
      if n ≤ 2 then return;
      N ← 0;  {initialize counter}
      TRAVBIND(ROOT);
      M ← ⌊(N + 1)/ 2⌋;
    ROOT ← LINK(M);  {new root}
      {balance the left and right half-trees in parallel}
      cobegin
        LSON(ROOT) ← GROW(1,M-1);
        RSON(ROOT) ← GROW(M+1,N);
      coend;
end;
```

FIGURE 2. Algorithm P1.

rithm P1 for example; the number of times each case is executed can be calculated as follows:

Let n be the total number of nodes in a set ($n \geq 3$), and let

$$a = \lfloor \log_2 n \rfloor, \quad \text{and}$$

$$b = \lfloor (2^a + 2^{a-1} - 1)/n \rfloor$$

Let K_i be the number of times case i is executed in Algorithm P1, $i = 1, 2, 3$.

Then,

$$K_1 = |2^a(\lfloor (n - 2^{a-1})/2^{a-1} \rfloor) - n - 1| \leq 2^{a-1}$$

$$K_2 = n - K_3$$

$$K_3 = 2^{a-b} - 1 + K_1 b$$

Let us suppose that C_0 is the average time to visit a node during traversal and C_i is the unit time taken to execute case i once. The total balancing time $T(n)$ can

be expressed as

$$T(n) = C_0n + C_1K_1 + C_2K_2 + C_3K_3 = O(n).$$

This expression also applies to Algorithms S and P2.

Based upon the above expression, we will do the analyses of Algorithms S, P1, and P2. When balanced through folding as described in Algorithm S, only $\lfloor n/2 \rfloor$ elements have to be partitioned. This tends to decrease K_i's by half and slightly increases C_i's. If parallel execution is implemented efficiently, Algorithm P1 should have similar effects as decreasing K_i's, in terms of turnaround time, yet without increasing C_i's.

The savings in time becomes obvious when n is large. Algorithm P2 overlaps traversal and pointer-restructuring, so that the turnaround time should be the greater of the two. Also, C_i's tend to be smaller, but more space is required and physical record movement tends to increase C_0.

The sequential Martin–Ness and Bentley algorithms that recursively partition the set of nodes to form a balanced, binary search tree follow the same partitioning scheme as that in Algorithm S.

A higher degree of parallelism and time efficiency is achieved in our algorithms at the expense of increased working space size. By using an array instead of a stack to store the ordering information of the set of nodes, we are able to build halves of the balanced tree simultaneously as in Algorithm S, or construct left subtrees and right subtrees in parallel as in Algorithm P1. By making a copy of the original tree, as done in Algorithm P2, we

```
Algorithm P2:
{build a balanced tree with a parallel sorting process}
procedure BALANCE(ROOT,KEY,LSON,RSON,n) :
  integer ROOT, n; array KEY; integer array LSON, RSON;
  begin
    integer N;
    integer array KEYCOPY(1:n), LSONCOPY(1:n), RSONCOPY(1:n);
    {traverse the copy of the original tree and sort keys}
    procedure TRAVSORT(T) :
      integer T;  {pointer to the next node to be visited}
      begin
        if T = null then return;
        TRAVSORT(LSONCOPY(T));
        N ← N + 1;  {count the sequence of visit}
        KEY(N) ← KEYCOPY(T);  {put Nth key in Nth position}
        TRAVSORT(RSONCOPY(T));
      end;
    {generate pointers for a balanced tree structure}
    procedure GROW(LOW,HIGH):
      integer LOW, HIGH;
      begin
        integer MID;  {median of a subset bound by LOW and HIGH}
        case
          LOW > HIGH {null branch}: return (null);
          LOW = HIGH {leaf}:
            begin
              LSON(LOW), RSON(LOW) ← null;
              return(LOW);
            end;
          LOW < HIGH {divisible subset}:
            begin
              MID ← ⌊(LOW + HIGH)/ 2⌋;
              cobegin  {form left and right subtrees in parallel}
                LSON(MID) ← GROW(LOW,MID-1);
                RSON(MID) ← GROW(MID+1,HIGH);
              coend;
              return(MID);
            end;
        end;
      end;
    if n ≤ 2 then return;
    {make a copy of the original tree}
    KEYCOPY ← KEY; LSONCOPY ← LSON; RSONCOPY ← RSON;
    N ← 0; {initialize counter}
    {sort keys and regenerate pointers in parallel}
    cobegin
      TRAVSORT(ROOT);
      ROOT ← GROW(1,n);  {new root}
    coend;
  end;
```

FIGURE 3. Algorithm P2.

```
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(1: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 + 1;
                  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(⌊(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;
```

Martin and Ness's Algorithm

```
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}
        LAST_VISITED ← 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 ← ⌊BACKBONE_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 - 1;
          M ← ⌊BACKBONE_LENGTH/2⌋;
        end;
    end;
```

Day's Algorithm

TABLE I. Performance of the Algorithms

Algorithm	Martin/Ness [1]	Bentley [2]	Day [3]	Chang/Iyengar		
				S	P1	P2
Right-Threaded Input Tree	No	No	Yes	No	No	No
Additional Work Space	Yes	Yes	No	Yes	Yes	Yes
Run Time*	$O(n)$	$O(n \log_2 n)$**	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Parallel Execution	No	No	No	No	Yes	Yes
Sorting	No	No	No	No	No	Yes
(*P*)-Perfect or (*R*)-Route Balanced	P	P	R	P	P	P

* n is the total number of nodes.
** Build and balance a tree instead of reorganizing an existing one.

are able to sort the nodes and regenerate subtree pointers in the same time.

Day's algorithm uses a stackless traversal to build a linked list and a loop to reorganize the linked list to a route-balanced tree structure without auxiliary storage and recursion. The average time required can be figured as shown below:

$$T(n) \approx C_0 n + C_1 \sum_{i=1}^{a} (\llcorner (n - i)/2^i \lrcorner)$$

$$\leq C_0 n + C_1 (n - 1)(1 - 2^{-a}) = O(n)$$

where C_0 is the average time to visit a node during traversal; C_1 is the time to go through the restructuring loop once; n is the total number of nodes in a tree ($n \geq 3$); and $a = \llcorner \log_2 n \lrcorner$.

Day's algorithm is a more efficient sequential balancing algorithm, if perfect balance is not required. However, in order to use this algorithm, a threaded tree has to be maintained. Properties of previously discussed algorithms are summarized in Table I for comparison.

5. SUMMARY

We have presented an efficient Algorithm S to globally balance binary search trees through folding provided that the order of all nodes is predetermined and the ith item can be found given i. Balancing a tree through folding reveals parallelism in the pointer-restructuring procedure. This is a useful advantage in a parallel processing environment.

Two parallel algorithms to balance binary search trees are proposed. Algorithm P1 partitions a set and balances left subtrees and right subtrees simultaneously. Another parallel algorithm, P2, actually sorts the records and makes pointer regeneration an independent activity. A comparison of our algorithms with other existing algorithms is given in Table I.

Acknowledgments The authors wish to thank an anonymous referee for a number of valuable suggestions and criticisms. Thanks are due to Professor Horowitz for his comments on our paper. We would also like to thank Professor Moitra for her comments on this paper.

REFERENCES

1. Bentley, J.L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517.
2. Day, A.C. Balancing a binary tree. *Comput. J. 19*, 4 (Nov. 1976), 360–361.
3. Horowitz, E., and Sahni, S. *Fundamentals of Data Structures.* Computer Science Press, Inc., Potomac, MD, 1976, pp. 442–456.
4. Iyengar, S.S., and Chang, H. Algorithms to create and maintain balanced and threaded binary search trees. Submitted for publication *Software and Practice Journal* November 1982.
5. Iyengar, S.S., and Chang, H. A fast global algorithm to balance binary search trees. Submitted for publication, 1984.
6. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley Publ. Co., Inc., Reading, MA, 1973, p. 722.
7. Martin, W.A., and Ness, D.N. Optimal binary trees grown with a sorting algorithm. *Commun. ACM 15*, 2 (Feb. 1972), 88–93.
8. Moitra, A., and Iyengar, S.S. Maximally parallel algorithms to balancing binary search trees. Submitted to *IEEE Trans. Comput.* for publication.

CR Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*trees*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation; F.1.3 [**Computation by Abstract Devices**]: Complexity Classes; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity
General Terms: Algorithms, Theory
Additional Key Words and Phrases: binary search tree, computational complexity, parallel algorithms' folding method

Received 4/82; revised 3/83; accepted 11/83

Author's Present Address: Hsi Chang and S. Sitharama Iyengar, Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803.