An Improved Algorithm to Maintain the Binary Search Tree Dynamically

Inayat-ur-Rehman, Zeeshan Mehta, Manzoor Elahi
Dept. of Computer Science
COMSATS institute of information technology
Islamabad, Pakistan
{inayat, , zeeshanmehta , tamimy}@comsats.edu.pk

Saif ur Rehman Khan
Dept. of Software Engineering
University of Malaya
Malaya, Kuala Lumpur, Malaysia
saif_rehman@siswa.um.edu.my

Abstract— Binary Search Tree (BST) is an acyclic graph that is widely used to arrange the data for optimal search. In order to maintain the binary search tree in optimal shape several algorithms have been proposed. A recently proposed technique [1] applies single and double rotations to balance the binary search tree. However, due to double rotation it takes almost double time as compared to single rotation; ultimately increasing the demand for memory and processor. In this paper we propose a new technique which solves the double rotation problem in almost half of the steps of existing algorithm.

***Index Terms*— Nonlinear Data Structures; BST; Rotation**

I. INTRODUCTION

Binary search tree is a non-cyclic graph with sorting mechanism. First of all a key is chosen as root of the empty tree. The root can have maximum of two nodes which act as right subtree and left subtree. The greater value from root is inserted as right subtree and smaller value from root as left subtree. Next, these left and right nodes may act as roots of right and left subtrees [2].

Binary search technique is mostly used to search the data from static data structure where the data is already sorted. BST is maintained for efficient searching from dynamic data structure. However, when the randomly chosen values are inserted in BST then it may lose its shape. If value to be inserted is greater than the root value then the BST may be right skewed and vise versa. Ω(4n / n3/2) number of different binary trees are possible for n number of nodes [5].

The height of the BST plays an important role for searching, insertion and deletion. If the BST is right skewed or left skewed then the height of the BST may reach up to n-1 [7]. In such cases it is same as linked list. When we search a specific value from linked list its running time is O(n), because we have to visit each and every node of linked list [3]. BST is maintained just like dictionary where the words are sorted alphabetically. To search the key from BST first of all we compare the key value with root node and if it is greater then we move to right else go to left. In each level we compare the key with only one node. Thus the maximum number of comparisons to search a key from BST is equal to its height. The average number of comparisons for random binary search tree is [4]:

$$\alpha \log n + O(\log \log n), where\, \alpha = 4.31107...$$

When the height of BST increases the number of comparisons to search the key also increases which can be shown by the Equation 1.

$$Height_{BST} \propto Comparison_{Node} \quad (1)$$

To search a key from random BST the average number of comparisons are approximately *log(n)* but if the BST is properly balance then it can be reduced by 15 to 20% [8]. Height plays the same role in insertion and deletion because to insert or delete a key we have to reach in perfect location. It can be shown in mathematical form as depicted in Equation 2.

$$Height_{BST} \propto Comparison_{Node} \in [Insertion_{Data} \vee Deletion_{Data}] \quad (2)$$

Several BST balancing techniques have been proposed [6]. This paper presents an improvement in algorithm proposed by Prasad *et al.* [1], which is an efficient algorithm to maintain the BST dynamically. This technique applies the right and left rotations to balance only that portion of the tree which loses its shape. There is an equal chance to apply the double rotation, which makes the tree maintenance costly. In this paper, we propose an algorithm that handles the single or double rotation by reducing the number of calculation steps of Prasad *et al.* [1] technique. Ultimately, it decreases the memory and processor requirements. This paper is organized as follows:

Section-II provides the related literature review of several BST balancing technique. Section-III presents the modified insertion algorithm, while section-IV discusses the experimental study results. Finally section–V concludes the paper and discusses the future work.

II. MAINTAINING BINARY SEARCH TREE DYNAMICALLY

Prasad et al. [1] proposes an algorithm that dynamically balances the BST. The BST becomes right skewed or left

978-1-4673-2252-2/12/$31.00 ©2012 IEEE

skewed if randomly insertion or deletion is applied. There can be four cases when the BST loses its shape as depicted in Figures 1-4:

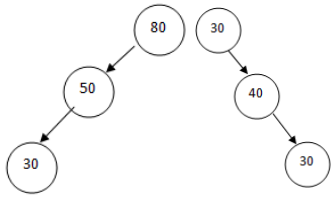

FIGURE-1 FIGURE-2

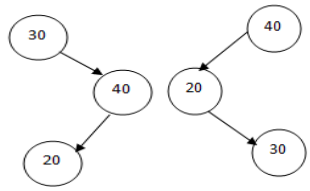

FIGURE-3 FIGURE-4

Different proposed techniques take the entire tree as input and apply rotation to balance the BST. For most of the techniques extra memory is needed to store the tree and time for balancing is O(n). This technique works in two phases. First of all random insertion or deletion is applied then in second phase BST's shape is checked and rotation is applied (if needed). Two extra pointers grandfather and fgrandfather are needed to check the tree shape. If the grandfather of the node being inserted has only one child then the rotation is applied to balance the tree. Same is the case while employing the deletion algorithm. After deleting a node, if the grandfather of the node deleted has only one child then again apply the rotation to balance the tree. This technique needs no extra memory except two pointers and running time is also *O (log n)* [1].

III. EXISTING INSERTION ALGORITHMS

This section is divided into three main categories. First of all we discuss the conventional BST insertion algorithm. Next, we discuss the modified insertion algorithm and finally we discuss the proposed insertion algorithm.

A. Conventional Insertion Algorithm

To insert a new node in BST, first of all we check the root node. If root is NULL (means root is not yet created) then we assign the newnode address to root node. If there is already an existing root node, we compare the value of newnode with root's value. If the value in the newnode is greater and root of right is NULL then we link the newnode to the right pointer else to the left pointer. But if the newnode value is greater and right pointer is already pointing to existing node then we move to the right child, considering that node as root node. This process is continued until we get the NULL pointer.

B. Modified Insertion Algorithm

Prasad et al. [1] proposes an algorithm that doesn't allow any grandfather with one child. If such a case occurs then apply the rotation to reduce the height of the BST. They applied single or double rotation as per requirement. The single rotation needs 4 steps and the number of steps is multiplied in double rotation.

Considering example shown in Figure 5, the insertion of new node named Newnode needs double rotation. In first step, the value is inserted as per rules of BST insertion. As Newnodes value (40) is greater then father's value (30) and grandfathers value (50) is greater than the father's value. So first rotation it moves on left side of father as shown in Figure1. In step two (Figure-5(b) fathers (30) is rotated to left which is first rotation. In Figure-5(c) right rotation is applied to grandfather. So to balance the tree we applied double rotation, which takes the half steps.

C. Proposed Insertion Algorithm

The modified algorithm as depicted by Figure 6 avoids the double rotation and performs the worst case task (double rotation) in 3 steps. In step 6(a) new value is inserted as per BST defined rules. In step 6(b) we point the Newnode's right pointer the father of Newnode and point the left pointer to Grandfather of Newnode. In step 6(c) we simply point the fgrandfather to Newnode and make it grandfather.

The insertion algorithm is implemented in C language by using the structures. Two pointers named *right and *left are used to move to a right and left. The third one element in the structure is an integer value named input. The other pointer is called the *root which points the root node of the BST.

```
struct BST
{
   int input;
    BST *left, *right;
                              }*root=NULL;
```

Initially the root is assigned the null value so that it could not act as dangling pointer. We use two extra pointers named Grandfather and fgrandfather to monitor the newnode position. First of all the data is inserted in BST as per data insertion algorithm and we change the position of these pointers as we move. The insert algorithm is passed a newnode address pointed by *t.

```
void insert(struct BST *t)
{
BST *ptr,*fgrandfather,*grandfather,*father;
ptr=fgrandfather=grandfather=father=root;
  if(root==NULL){
root = t; }
```

If the first time insertion algorithm is called then the pointer root will have NULL. In such case the new node (pointed by *t) will be the root node.

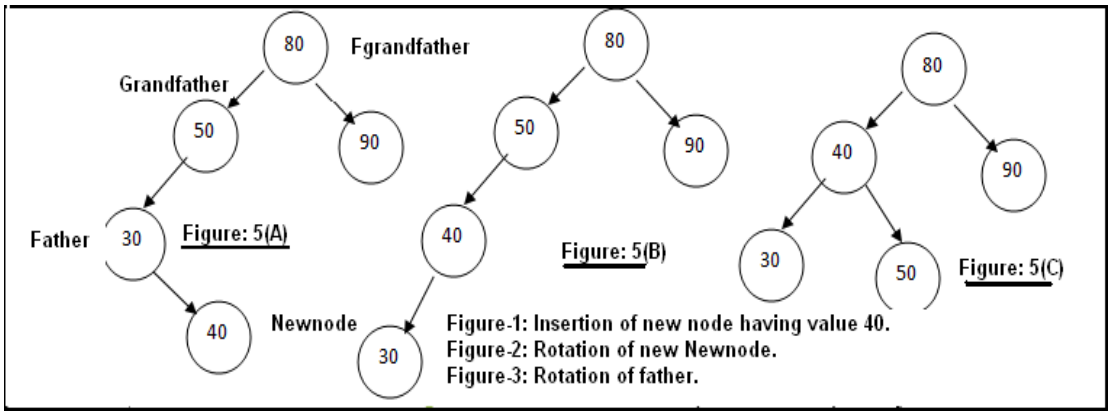

FIGURE-5: PRASAD ET AL. [1] ALGORITHM'S WORKING

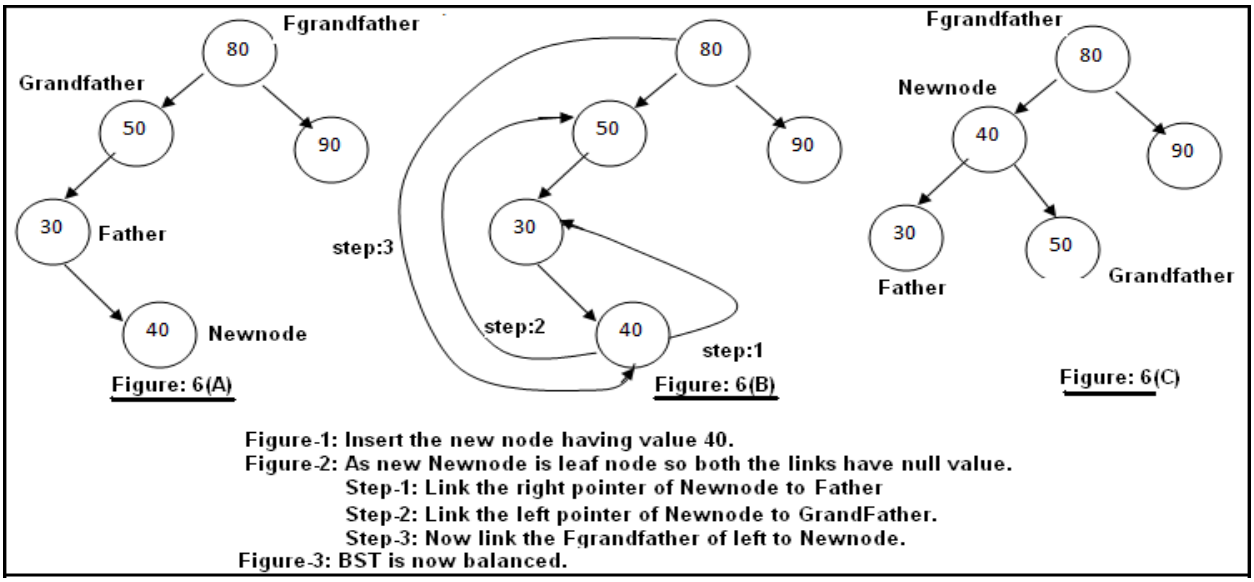

FIGURE-6: PROPOSED ALGORITHM'S WORKING

```
else {
        BST* ptr;
        ptr = root;
        while(ptr)
        {
          fgrandfather=grandfather;
          grandfather=father;
          father = ptr;
          if(t->val > ptr->val)
             ptr = ptr->right;
                 else ptr = ptr->left;
        }
```

Now, we take a pointer *ptr which points the root node. We move the pointer *ptr until we reach the location where we insert the value. The rule of movement is that if the new node (*t) value is greater than the root node (*ptr) value then we move to the right subtree else we go to left subtree. This process is continued and the pointers (father, grandfather, fgrandfather) are moved accordingly until we reach the desired location (*ptr gets the Null value).

```
if(t->val < father->reg)
                father->left = t;
        else
                father->right = t;
```

When the *ptr gets the NULL value then we check that the new node (*t) will be connected to right of father or left.

Now at this stage the conventional insertion of BST is complete. The next stage is to check and adjust the balance of the BST. There can be four different shapes of the unbalanced tree and the solution for each case is discussed in detail.

1) Solution for Single Left Rotation:

When father's value is greater than grandfather's value and newnode value (t->val) is also greater than father's value.

```
if (fgrandfather!=grandfather && father-> right==t && grandfather->right==father && grandfather->left==NULL)
{
  if(fgrandfather->right==grandfather) {
       fgrandfather->right=father;
  }else
  {
       fgrandfather->left=father;
    }
  father->left=grandfather;
  grandfather->right=NULL;  }
```

2) Solution for Single Right Rotation

When father's value is less than grandfather's value and newnode's value is also less than father's value.

```
else if (fgrandfather!=grandfather && father->left==t && grandfather->left==father && grandfather->right==NULL){
    if(fgrandfather->right==grandfather)
  {  fgrandfather->right=father;
  }else {
       fgrandfather->left=father;
    }
  father->right=grandfather;
  grandfather->left=NULL;
}
```

3) Solution for Left Right Rotation (Double Rotation)

When father's value is less than grandfather's value and newnode's value is greater than father's value the we need first left rotation to make the BST skewed then we apply the right rotation.

```
else if (fgrandfather!=grandfather && father->right==t && grandfather->left==father && grandfather->right==NULL)
          {
              if(fgrandfather->right==grandfather)
            {  fgrandfather->right=t;
            }else
            {
fgrandfather->left=t;
      }
       t->left=father;
       t->right=grandfather;
grandfather->right=grandfather->left=father->left=father->right=NULL;
```

4) Solution for Right-Left Rotation (Double Rotation)

If the father's value is greater than grandfather's value and newnode value (t->val) is less than father's value then we first apply the right rotation and then left rotation.

```
else if (fgrandfather!=grandfather && father-> left==t && grandfather->right==father && grandfather->left==NULL)
{
    if(fgrandfather->right==grandfather)
     {
          fgrandfather->right=t;
      }else{
       fgrandfather->left=t;      }
  t->right=father;
  t->left=grandfather;
grandfather->right=grandfather->left=father->left=father->right=NULL;
    }else   return;   }
```

IV. Experimental Results

The graph (Figure 7) shows the performance of proposed rotation algorithm with compared to existing algorithm. Ten sample runs (Table I) are provided with number of rotations and the steps taken by existing (EA) and proposed algorithm (PA). We have tested the proposed algorithm using Dev compiler for 10 different inputs starting from 50 to 5,000.

TABLE I. RESULT OF DOUBLE ROTATION

Double Rotation	*Current Level*	*Steps for DR to EA*	*Steps in DR by PA*	*Steps Reduced by PA*
12	8	288	144	144
17	10	408	204	204
37	13	888	444	444
69	15	1656	828	828
101	15	2424	1212	1212
126	18	3024	1512	1512
172	17	4128	2064	2064
208	20	4992	2496	2496
323	19	7752	3876	3876
247	21	5928	2964	2964

In Figure 7, green line represents the existing algorithm and blue line represents the proposed rotation algorithm. From the figure, it is clear that

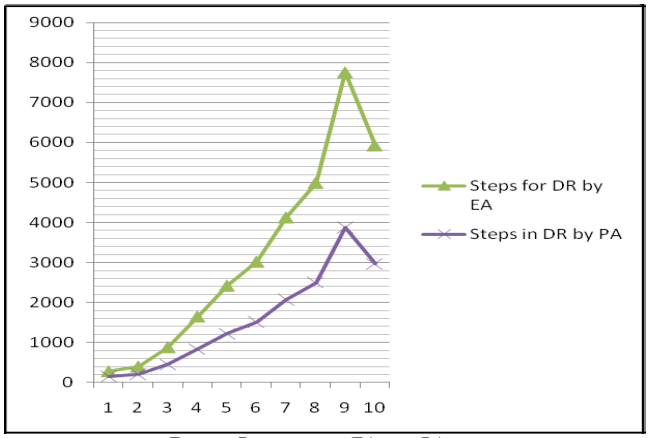

FIGURE-7: RESULT OF EA AND PA

initially both algorithms have shown a similar performance for smaller tree size. But as the size of tree grows the proposed algorithm performs exceptionally well compared to the existing algorithm.

V. CONCLUSION AND FUTURE WORK

BST balancing always remains the hot issue to maintain the BST in optimal shape. Almost all the techniques apply the rotation to balance the BST. To apply the single rotation we perform the four steps and in some cases we need double rotation. Double rotation needs two rotations; first to make the skew and then to balance it. The proposed technique improves the algorithm by discovering the approach other than rotation. The new algorithm improves the average as well as worst case. The efficiency of the propose algorithm can be judged by considering the worst case scenario (double rotation) where it eliminates more than half restructuring steps.

In future, we plan to propose such a rotation technique, which could be applied to such balancing algorithms, where rotation is needed.

REFERENCES

[1]. Vinod, P. Suri, P. and Maple, C., "Maintaining a Binary Search Tree Dynamically". In Proceedings of the 10th International Conference on Information Visualization, pp. 483-488, 2006.

[2]. Heger, D. A., "A Disquisition on the Performance Behavior of Binary Search Tree Data Structures" European Journal for the Informatics Professional, Vol. V, No.5, October 2004.

[3]. Brodal, G. S., Fagerberg, R. and Jacob, R., "Cache Oblivious Search Trees via Binary Trees of Small Height", BRICS Report Series, ISSN 0909-0878, October 2001.

[4]. Li, C-C., "An Immediate Approach to Balancing Nodes of Binary Search Trees" Dept. of Computer Science, Lamar University, Beaumont, Texas, USA.

[5]. Kechid, M., and Myoupo, J. F., "A Coarse Grain Multicomputer Algorithm Solving the Optimal Binary Search Tree Problem", In Proceedings of the 5th International Conference on Information Technology: New Generations, 2008.

[6]. Inayat-ur-Rehman, Saif-ur-Rehman Khan, M. Sikandar Hayat Khayal, "A Survey on Maintaining Binary Search Tree in Optimal Shape," icime, pp.365-369, 2009 International Conference on Information Management and Engineering, 2009.

[7]. Pushpa, S., Vinod, P., and Khader, J. A., "Insertion and Deletion on Binary Search Tree using Modified Insert Delete Pair: An Empirical Study" International Journal of Computer Science and Network Security, Vol.7 No.12, 2007.

[8]. Kruse, R. L. and Ryba, A. J., "Data Structures and Program Design in C++", Prentice Hall, ISBN: 0-13-768995-0, 1998.