

# Searching Approaches For BST-based Leftist Tree Using Breadth-First-Traversal

Ghanshyam I Prajapati<sup>#1</sup>, Chetan K Solanki<sup>#2</sup>, Jayesh G Chaudhary<sup>#3</sup>

<sup>1</sup>Lecturer, Shri S'ad Vidya Mandal Institute of Technology, Bharuch (Gujarat), India

<sup>2</sup>Lecturer, C.K.Pithawalla College of Engineering, Surat (Gujarat), India

<sup>3</sup>PG-Student (Computer Engineering),

Birla Vishvakarma Mahavidyalaya, Vallabh Vidyanagar-Anand (Gujarat), India

## Abstract

A BST-based Leftist Tree is a simple modification of normal Leftist Binary Tree except that  $DATA[i] \geq DATA[FATHER[i]]$  and is based on the concepts of BST (Binary Search Tree) to form. Two different searching approaches (Static Approach and Hybrid Approach) based on Breadth-First-Traversal have been defined and implemented to find whether the given Binary Search Tree is BST-based leftist or not, if yes, then these approaches also give its shortest distance to traverse from root node to its minimal leaf node in left-subtree of given BST. The Static Approach uses an array (static list) which is easy to implement but requires more space and also causes the problem of memory wastage or memory un-usage. While Hybrid Approach is based on traditional queue data structure and augmented data structure which gives less time complexity and less space complexity as compared to Static Approach.

**Key Words:** Left-Shortest-Distance(LDIST), Right-Shortest-Distance(RDIST), Time and Space Complexity

## 1. Introduction

Advanced Data Structure and Augmented Data Structure are rapidly emerging as latest fields of research under the mixed or hybrid area of *Computer Algorithms* and *Data Structure Methodology*. In this area, we need to create an entirely new type of data structure or modified an existing one to add additional information to support desired applications. Augmenting a data structure is not always straightforward, since the added information must be updated and maintained by the ordinary operations on the data structure [1].

### 1.1. BST-based leftist tree

A (normal or usual) leftist tree [2], [3] is a part of advanced data structure and is special kind of a binary tree which *strictly* follows the following properties for all node-K including root node:

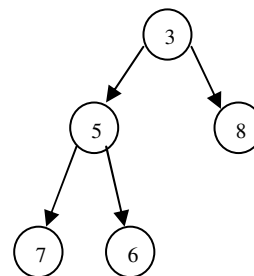
- (a)  $DATA[LEFT[K]] \geq DATA[K]$  and  $DATA[RIGHT[K]] \geq DATA[K]$
- (b)  $LDIST[K] \geq RDIST[K]$

A Binary Search Tree or BST-based leftist tree is a combination of advanced data structure and augmented data structure and is a special kind of a binary tree which has to follow the following properties for all node-K including root node:

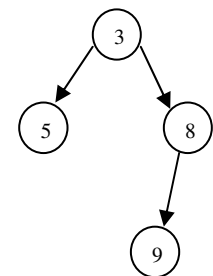
- (c)  $DATA[LEFT[K]] \leq DATA[K] \leq DATA[RIGHT[K]]$
- (d)  $LDIST[K] \geq RDIST[K]$

Here, property (d) of BST-based leftist tree remains same as property (b) in normal leftist tree but there is a minor change in property (c) as in normal leftist tree. In normal leftist tree [4], [5], data part of parent node is always less than or equal to its children data parts, while BST-based leftist tree assures to be a kind of binary search tree. This clear difference between normal leftist tree and BST-based leftist tree can be shown in figure 1.

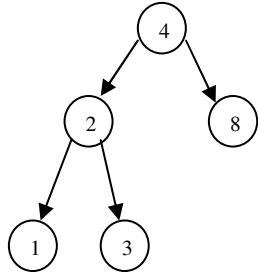
$LDIST[K]$  is the minimum distance or minimum number of nodes from root node to its leaf node in left subtree of a given node-K and same for  $RDIST[K]$  in right subtree. Figure 1.1 shows a perfect example of normal leftist tree because each node having lesser data part then its both children (left and right) and at root node,  $LDIST$  is 2 which is greater than  $RDIST$  that is 1 from root node, and also each node including root node of figure 1.1 follows property (b) of normal leftist tree. But in figure 1.2, at root node  $LDIST$  is 1 which is not greater than or equal to  $RDIST$  which is 2 in this case, so figure 1.2 does not



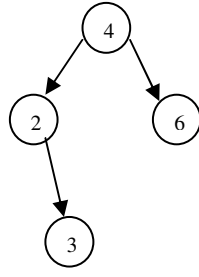
( Fig 1.1)  
(Normal Leftist Tree)



( Fig 1.2)  
(Not a Normal Leftist Tree)



( Fig 1.3 )  
(BST-based Leftist Tree)



( Fig 1.4 )  
(Not a BST-based Leftist Tree)

(Fig 1 : Difference between Normal and BST-based Leftist Tree)

Show an example of normal leftist tree. Figure 1.3 and figure 1.4, both follow the properties of binary search tree. From figure 1.3, LDIST (2 from root node in this case) is greater than RDIST (1 from root node in this case) and each node of this tree of figure 1.3 follows property (d) of BST-based leftist tree therefore figure 1.3 is an example of BST-based leftist tree, but in figure 1.4, property (d) does not follow at node '2' so this is not an example of BST-based leftist tree.

## 1.2. Breadth-first-search in binary tree

A breadth first search (BFS) method [6] searches or processes each node of binary tree in *level by level* in increasing fashion and *left to right* in each level. In a binary tree, searching or traversal starts from root node because it is present at level - 0 (level - zero). And then it will jump to next level that is level-1, level-2 and so on, and each level it processes in left to right manner. BFS for figure 1.1 is 3, 5, 8, 7, 6. Normally, this method does not require recursion. In depth first search (DFS) method [6], searching or processing of nodes of binary tree is in *deeper-in-level* manner. DFS (in preorder [7] fashion) for figure 1.1 is 3, 5, 7, 6, 8. This DFS method usually requires recursion and ultimately uses stack data structure to implement.

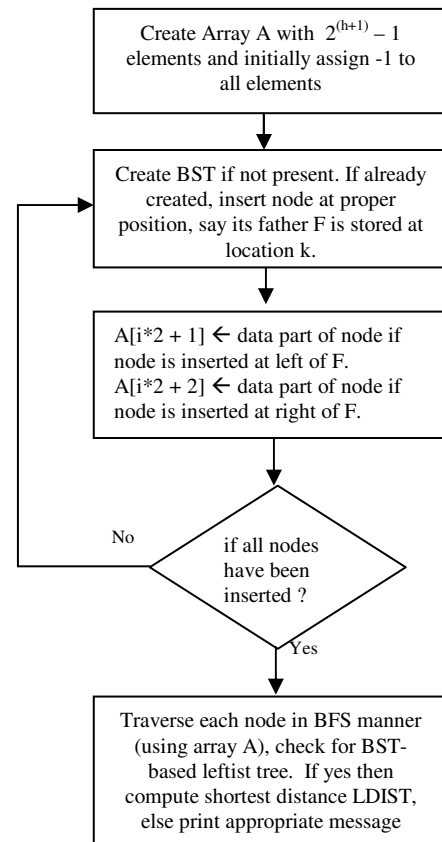
In this paper, we have presented and implemented (in C++) two different approaches, **Static Approach** and **Hybrid Approach**, for finding whether the given binary search tree is BST-based leftist tree or not, and if it is, then these both the approaches give shortest distance in left subtree from root node.

## 2. Static approach

This approach uses an array or a static list to store data part of a node or binary tree information in breadth-first-manner. Here the size of array is taken as  $2^{(h+1)} - 1$  where h is height of the given BST with n nodes, this array also stores the children (left and right both) information for

each node including leaf nodes. For example, the height of BST shown in figure 1.3 is 2 so array size for this case is 7.

In this approach, while creating a BST or inserting a node into a BST, each time it updates array. Assume a father of inserted node is stored at location i in array then if inserted node is left child then it will be stored at  $(2*i + 1)$  in array while right child will be at  $(2*i + 2)$ .



( Fig 2 : A flowchart for static approach )

This approach uses traditional or old-fashioned data structure for searching of BST-based leftist tree, it is written as:

```
struct BST_TREE
{
    int DATA;
    struct BST_TREE *LEFT;
    struct BST_TREE *RIGHT;
};
```

```

void STATIC_BFS( )
{
    int PATH1=0,PATH2=0;
    for(int i=0;i<2047;i++) //for BST with max height of 10
    {
        if(A[i]!=-1)
        {
            if(A[i*2+1]==-1 && A[i*2+2]!=-1)
            {
                printf("No Leftist Tree");
                return;
            }
            if(A[i*2+1]==-1 && A[i*2+2]==-1)
            {
                if ( A[i]<=START->DATA) //START is a root node
                {
                    int a=0;
                    while(1) //while loop-1
                    {
                        if(i>=pow(2,a)-1 && i<=pow(2,a+1)-2)
                        {
                            if(PATH1==0 || PATH1>a)
                                PATH1=a;
                            break;
                        }
                        else
                            a++;
                    } //end of while loop-1
                }
                else //A[i] >START->DATA
                {
                    int b=0;
                    while(1) //while loop-2
                    {
                        if(i>=pow(2,b)-1 && i<=pow(2,b+1)-2)
                        {
                            if(PATH2==0 || PATH2>b)
                                PATH2=b;
                            break;
                        }
                        else
                            b++;
                    } //end of while loop-2
                }
            }
        } //end of for loop

        if(PATH1 >= PATH2)
            printf("BST-Leftist with shortest path -- %d ",PATH1);
        else
            printf("No BST-Leftist Tree");
    } //end of BFS function
}

```

( Fig 3: Pseudo-code(partial) for static approach )

in which, DATA is used to store data part or information part of node while LEFT and RIGHT are the pointers which are used to point to left child or left subtree and right child or right subtree respectively.

A flowchart of this approach is given in figure 2 and pseudocode (which is in C++) is shown in figure 3. Once the BST (with all given nodes) has been formed then STATIC\_BFS( ) function will be invoked to fulfil our objective. The working mechanism of this STATIC\_BFS function can be written as :

- [1] Initially declare two integer variables PATH1 and PATH2 which will store the values of left-shortest-distance (LDIST) and right-shortest-distance (RDIST) respectively for BST-based leftist tree.
- [2] Check for each node for the properties satisfaction of BST-based leftist for  $2^{(h+1)} - 1$  times. If at any node of BST, condition (d) of BST-based leftist tree satisfies then LDIST and RIST will be calculated and stored in PATH1 and PATH2 respectively using while loop.
- [3] At the end, comparison will be done between PATH1 and PATH2. As per the property (d) of BST-based leftist, if PATH1 is greater than or equal to PATH2 then our BST is called as BST-based Leftist Tree and the value of PATH1 (left-shortest-distance) will be our output otherwise it will print appropriate message.

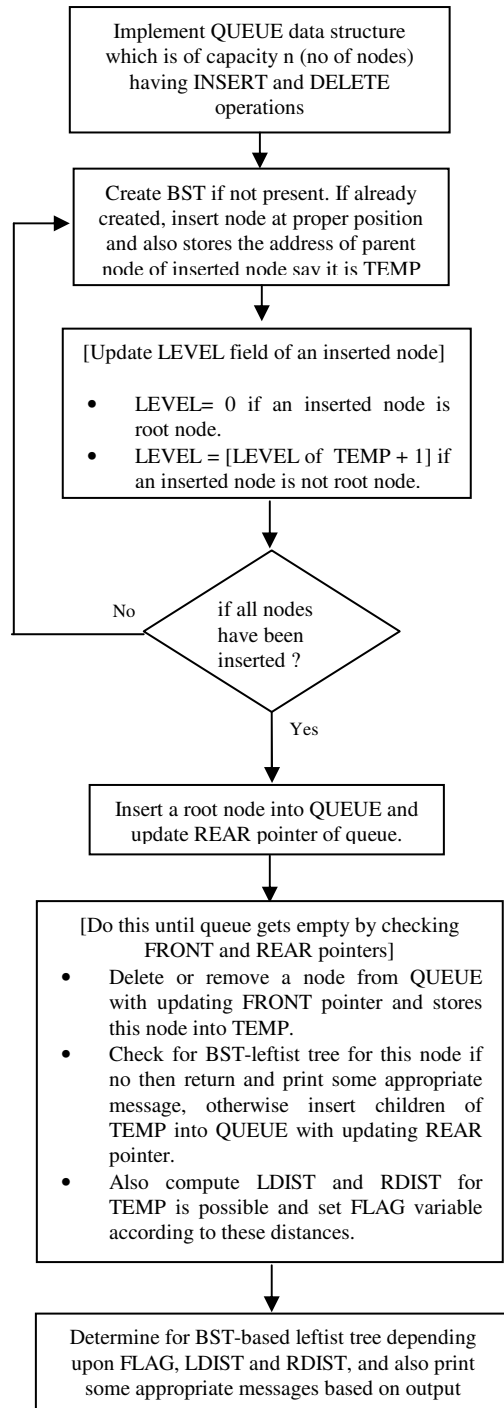
### 3. Hybrid approach

This approach is presented and implemented using traditional queue data structure and augmented data structure, therefore this approach is called as **Hybrid Approach**. This combines features of both the things, queue data structure plus augmented data structure. This approach does not use any kind of array to store data or information part of each node but only uses an extra field 'LEVEL' for each node. This is nothing but augmenting our traditional data structure of binary tree. So the new or modified or augmented data structure of binary tree or binary search tree is given as:

```

struct BST_TREE
{
    int DATA;
    int LEVEL;
    struct BST_TREE *LEFT;
    struct BST_TREE *RIGHT;
};

```



( Fig 4 : A Flowchart for hybrid approach )

Here DATA, LEFT and RIGHT possess their usual meanings as in traditional data structure of BST, but a new field is added named 'LEVEL' to augment this tree data structure. This field of a node keeps track of level of a node. For example, level of root node is zero, so LEVEL field of root node stores value 0. An immediate left child or an immediate right of root node is at level

```

void HYBRID_BFS(struct BST_TREE *TEMP)
{
    int FLAG=1,PATH1=0,PATH2=0;
    struct BST_TREE *NEW;
    if(TEMP==NULL)
        return;
    else
    {
        Q_INSERT(TEMP); //insert root node into queue
        while(FRONT <= REAR)
        {
            NEW=Q_DELETE();
            if ((NEW->LEFT==NULL &&
                NEW->RIGHT == NULL))
            {
                if(NEW->DATA<=START->DATA)
                {
                    //START is root node
                    if(PATH1!=0)
                    {
                        if(PATH1 > NEW->LEVEL)
                            PATH1=NEW->LEVEL;
                    }
                    else
                        PATH1=NEW->LEVEL;
                }
                else
                {
                    if(PATH2!=0)
                    {
                        if(PATH2 > NEW->LEVEL)
                            PATH2=NEW->LEVEL;
                    }
                    else
                        PATH2=NEW->LEVEL;
                }
            }
            if(NEW->LEFT != NULL)
                Q_INSERT(NEW->LEFT);
            else if(NEW->LEFT == NULL &&
                NEW->RIGHT != NULL)
            {
                FLAG=0; //no leftist tree
                break; //immediately terminates if left
                //child is null and right child is present
            }
            if (NEW->RIGHT != NULL)
                Q_INSERT(NEW->RIGHT);
        }
        //end of while loop
    } //end of else statement
    if(FLAG==0)
        printf("No BST-Leftist Tree \n");
    else
    {
        if(PATH1 >=PATH2)
            printf("Leftist with PATH1 -- %d",PATH1)
        else
            printf("No Leftist Tree\n");
    }
} //end of HYBRID_BFS() function
  
```

( Fig 5 : Pseudocode(partial) for hybrid approach )

one therefore LEVEL field of this node (either left or right child of root node) is 1. The LEVEL field of a node-k will be updated or set according to its level while insertion of node-k in BST. Once the BST with all given nodes has been formed, a hybrid approach uses *queue* data structure to traverse a formed BST in BFS (Breadth First Search) manner to achieve a goal for finding BST-based leftist with shortest distance in such BST.

A simple flowchart of hybrid approach is given in figure 4 and pseudocode can be written and shown in figure 5 for the same. This approach uses a queue data structure with capacity of n (n is number of nodes in BST) and having INSERT and DELETE operations (say Q\_INSERT and Q\_DELETE in pseudocode of this approach). While inserting a node into BST, each time LEVEL of inserted node is updated or set. After creation of binary search tree (BST), HYBRID\_BFS( ) function will be invoked with passing an address of root node as an argument (in this case root node is stored at START) to reach to our destination.

The working of HYBRID\_BFS( ) can be given as:

- [1] In order to traverse a given BST in BFS manner, we start with root node so first of all , root node will be inserted in queue if tree is not empty or null.
- [2] While loop is kept to keep track of all the nodes of BST have been traversed in BFS manner (level by level) or not. In this loop, each time check the property (d) of BST-based leftist tree for deleted node. If satisfies then children of this node is inserted in queue (first left child and then right child). Also at this node, compute PATH1 and PATH2 (used to store LDIST and RDIST) if possible, and depending upon these values FLAG will be set to either 0 or 1. If PATH1 is greater than or equal to PATH2 then FLAG is will be set to 1 otherwise 0.
- [3] At the last, if all the nodes have been traversed (it means queue is empty) then the time to show correct outputs based the values of FLAG, PATH1 and PATH2. If the value of FLAG is zero then we can say property (d) of BST-based leftist tree is not satisfied so the given BST is not an example of BST-based leftist. If FLAG is one and the value of PATH1 is greater than or equal to PATH2 then and then we can say that our BST is a BST-based leftist tree and prints its left shortest distance (PATH1).

#### 4. Performance analysis

Analysis of algorithm depends upon various factors such as memory, communication bandwidth, or computer hardware. But the most often used is the computational time that an algorithm requires for completing the given

task. Also the performance of an algorithm focuses on time and space complexity. The space complexity refers to the amount of memory required by an algorithm to run to completion while time complexity is referred to as the amount of time required by an algorithm to run to completion [8]. There are three various asymptotic notations which are currently being used to express time and space complexity, ‘O’ (Oh or Big Oh) notation, ‘Ω’ (Omega) notation and ‘θ’ (Theta) notation [9]. Usually O-notation is used to express time complexity for an algorithm because it gives upper bound or upper limit of time to complete the given task

A time-complexity analysis of Static approach and Hybrid approach is shown in table 1. As we know that to create a binary search tree with n nodes, time complexity is  $O(n \lg n)$  so in both the cases (static as well as hybrid), time complexity for creation of BST with n nodes is  $O(n \lg n)$ . But both the approaches differ in BFS with n nodes.

**Table 1 : Comparison between time complexity of static and hybrid approach**

	Static Approach	Hybrid Approach
Creation of Binary Search Tree with n nodes	$O(n \lg n)$	$O(n \lg n)$
BFS for n nodes	$O(2^d)$ where $d = \text{height of BST} + 1$ $\text{height of BST } (h) = \log_2 n$ so we can write $O(2^d) = O(2^{\lg n + 1})$ (where $\lg n = \log_2 n$ )	$O(n)$

In static approach, we have to check each and every element in array for the satisfaction of property (d) of BST-based leftist tree and the same array is in size of  $2^{h+1}$ , where h is height of BST with n nodes, so each step in for loop (refer figure 3) will be performed  $2^{h+1}$  times (refer figure 3) so the time complexity of static approach is  $O(2^{h+1})$ . While in the hybrid approach, queue data structure with size of n (where n is number of nodes) is used for breadth first search to check whether the given BST is BST-based leftist or not. Therefore only  $n + 1$  comparisons will be taken for BFS to find BST-based leftist tree with left shortest distance (refer figure 5), so the time complexity for BFS with n nodes in hybrid approach is  $O(n)$  which is quite better than static approach. If we talk about space complexity then hybrid approach is better than static approach, because static approach requires maximum  $(2^{h+1}-1)$  memory locations (where h is height of BST with n nodes) to store data part of each node of BST. But in case of hybrid approach,

queue size is of only  $n$  (where  $n$  is number of nodes in BST) and in fact, an extra memory field (for LEVEL) is required to store the level of each node of BST, but this is quite negligible and can be ignored. Hence, this proves that the hybrid approach is faster and requires less memory as compared to the static approach. Here we ignore time overhead of queue operations (Q\_INSERT and Q\_DELETE) in hybrid approach because whenever we insert or remove single node into or from queue, it always gives time complexity in constant manner and can be written as  $O(1)$ .

## 5. Conclusion

In this paper, we have presented and implemented (in C++) two different searching approaches, static and hybrid, for finding BST-based leftist tree with left shortest distance using Breadth-First-Search methodology. From the flowcharts and pseudocode of both these approaches, it is seen that the static approach is quite easy to understand and easy to implement because it does not require any queue operations for BFS, but this is little bit time consuming and spacious as compared to hybrid approach. In other words, a hybrid approach gives better results with respect to time and space than static approach; this also shows an overhead of queue maintenance and LEVEL field of modified tree data structure. But this can be overlooked to make faster algorithm and to save memory.

## 6. References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Augmented Data Structure, "Introduction to Algorithms", 2<sup>nd</sup> Edition, PHI Publications, New Delhi, India, 2002, page no : 302
- [2] Concepts of Leftist tree : webpage visited on 11<sup>th</sup> Nov 2009 <http://www.brpreiss.com/books/opus5/html/page361.htm>
- [3] Fundamentals of Leftist tree : webpage visited on 11<sup>th</sup> Nov 2009 <http://www.dgp.toronto.edu/people/JamesStewart/378notes/10leftist/>
- [4] Definition of Leftist tree : webpage visited on 15<sup>th</sup> Nov 2009 <http://www.itl.nist.gov/div897/sqg/dads/HTML/leftisttree.html>
- [5] More on Leftist Tree : webpage visited on 15<sup>th</sup> Nov 2009 [http://en.allexperts.com/e/l/le/leftist\\_tree.htm](http://en.allexperts.com/e/l/le/leftist_tree.htm)
- [6] G. Brassard and P. Bratley, BFS, "Fundamentals of Algorithms", 1<sup>st</sup> Edition, PHI Publications, New Delhi, India, 2008, page no : 302 - 305
- [7] E. Horowitz, S. Sahni and S. Rajasekaran, Preorder Traversal, "Fundamentals of Computer Algorithms", 2<sup>nd</sup> Edition, Universities Press, Hyderabad, India, 2008, page no : 333-336
- [8] N. Upadhyay, Analysing Algorithms, "The design and analysis of algorithms", 2<sup>nd</sup> Edition, S.K. Kataria & Sons

- Publications, New-Delhi, India, 2006, page no : 20 - 21
- [9] E. Horowitz, S. Sahni and S. Rajasekaran, Asymptotic Notation, "Fundamentals of Computer Algorithms", 2<sup>nd</sup> Edition, Universities Press, Hyderabad, India, 2008, page no : 39 - 47