Path between nodes in a full Binary Tree

Mudit Rathore[1], Ishani Mishra[2], and Gugloth Lavanya[3]

[1]Indian Institute of Information Technology Allahabad, ICM2015502
[2]Indian Institute of Information Technology Allahabad, IWM2015008
[3]Indian Institute of Information Technology Allahabad, ISM2015003

Abstract— **Trees are non linear data structures in which data is organized in a hierarchical manner. A tree represents hierarchical relationships among its elements. It is very useful for information retrieval. Searching in it is very fast. Trees can be seen as a work flow for compositing digital images for visual effects or as a form of a multi-stage decision-making. They also come in handy for solving network path issues and in router algorithms.**
This paper deals with an algorithm to find out the path between any two nodes of a full binary tree.

I. INTRODUCTION

Tree [1] is a non-linear data structure which organizes data in hierarchical structure. In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order.

In tree data structure, every individual element is called as **Node**. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

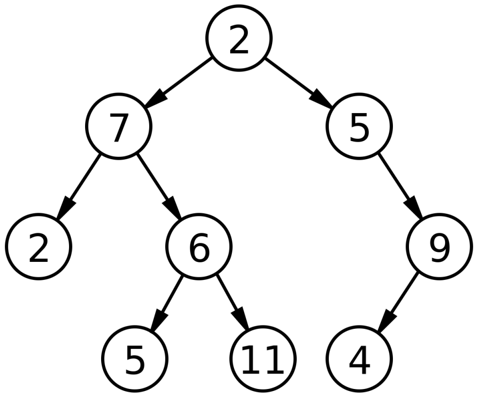

Fig. 1. A Tree

The node at the top of the tree is called **root**. There is only one root per tree and one path from the root node to any node. Every tree must have a root node. We can say that root node is the origin of tree. In any tree, there must be only one root node. In the above depicted Fig. 1., 2 is the root node.

Any node except the root node has one edge upward to a node called **parent**. The node below a given node connected by its edge downward is called its child node.

The node which does not have any child node is called the **leaf node**. All the other nodes except the child nodes are called the non leaf nodes or non terminal nodes. In the above depicted Fig. 1., 2, 5, 11, 4 are the leaf nodes or the terminal nodes, while, the remaining are non terminal or non leaf nodes.

Level of any node is defined as the distance of that node from the root. All nodes at a same distance from the root will belong to the same level. In Fig. 1., 2 is at level 0; 7 and 5 are at level 1; 2, 6, 9 are at level 2; and so on.

Path of a node is defined as the sequence of nodes $N_1, N_2, ...N_m$ such that each node N_i is parent of N_{i+1} for 1 < i < m. In a tree there is only one path between any two nodes. The path length is defined as the number of edges on the path (m-1).

Trees can be recursively defined [1] as- A tree is a finite set of nodes such that:

1) There is a distinguished node called the root and
2) The remaining nodes are partitioned into n $>=$ 0 disjoint sets $T_1, T_2, ...T_n$ where each of these sets is a tree. Sets $T_1, T_2, ...T_n$ are sub trees of the root.

A tree whose elements have at most 2 children is called a **binary tree** [2]. Since each element in a binary tree can have only 2 children, we typically name them

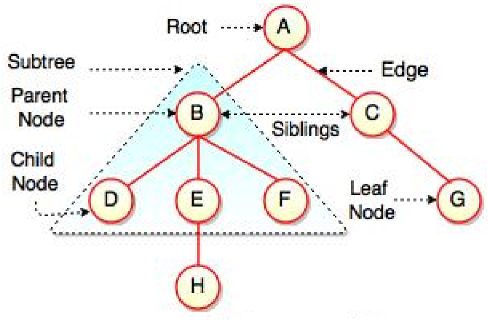

Fig. 2. Structure of Tree

the left and right child. A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Tree node contains following parts.

1) Data
2) Pointer to left child
3) Pointer to right child

The structure of a tree in code can be seen as the following:

```
struct Node
{
    int key;
    struct Node *left, *right;
};
```

A **full binary tree** (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

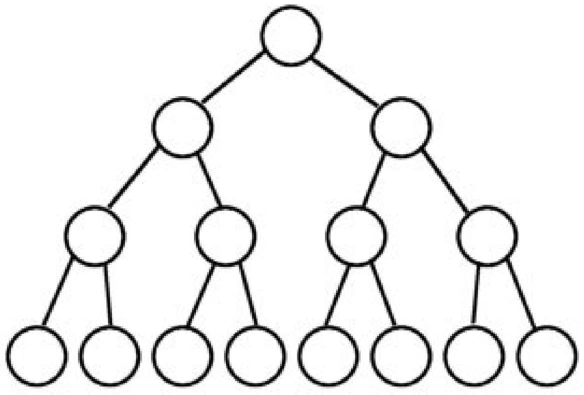

Fig. 3. A full binary tree

Level order traversal of a tree is breadth first traversal for the tree. The level order traversal of the graph in Fig. 1., is **2 7 5 2 6 9 5 11 4**.

Sequential representation of the trees can be done using arrays. To represent tree using array, numbering of nodes can start either from 0–(n-1) or 1–n or in other words array indexing can either start from 0 or from 1.
For the first case, when indexing starts from 0:
if (say)father=p;
then left-child=(2*p)+1;
and right-child=(2*p)+2;

For second case, when indexing starts from 1:
if (say)father=p;
then left-child=(2*p);
and right-child=(2*p)+1;
where father,left-child and right-child are the values of indices of the array.

The **Lowest Common Ancestor or the LCA** [3] of n_1 and n_2 in T is the shared ancestor of n_1 and n_2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure [4] for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n_1, plus the distance from the root to n_2, minus twice the distance from the root to their lowest common ancestor.

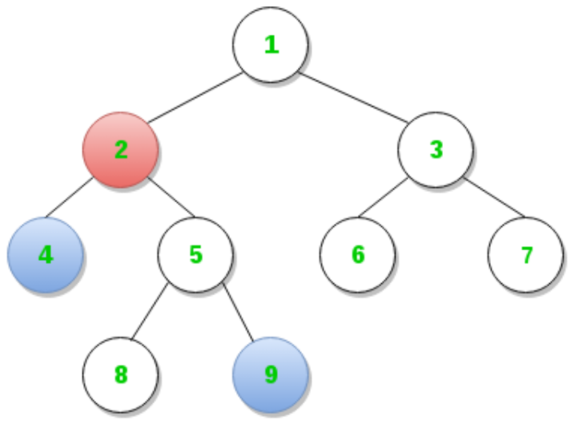

Fig. 4. A full binary tree

For example, in the above figure, LCA of nodes 4 and 9 is 2.

II. METHODOLOGY

The paper deals with finding the path between two nodes in a full binary tree.

A. Input

The user is made to input the number of nodes and the level order traversal array of the full binary tree. The user is then asked to enter the two nodes between which path has to be found.

B. Inserting nodes into the tree

The level order array given by the user is used to build the tree using a recursive approach. Given an array of elements, we have to construct a full binary tree from this array in level order fashion. As we already know that if parent node is at index i in the array then the left child of that node is at index (2*i + 1) and right child is at index (2*i + 2) in the array, given indexing of the array starts from 0. Using this concept, we can easily insert the left and right nodes by choosing its parent node. We will insert the first element present in the array as the root node at level 0 in the tree and start traversing the array. For every node i we will insert both left and right children in the tree. Once the full binary tree is built the proposed algorithm is run on it.

```
Node* insertLevelOrder(int ar[],
        Node* root, int i, int n)
{
  if(i < n)
  {
    Node* temp = newNode(ar[i]);
    root = temp;
    root->left = insertLevelOrder(ar,
    root->left, 2 * i + 1, n);
    root->right = insertLevelOrder(ar,
    root->right, 2 * i + 2, n);
  }
  return root;
}
```

1) Time Complexity: Each element of the level order array is traversed once linearly hence the complexity of the *insertLevelOrder()* function is $O(n)$, where n is the total number of nodes.

C. Proposed Algorithm

The proposed algorithm has two functions namely *findPath()* and *printPath()*.

The function *findPath()* tells if there is a path between the root and the nodes separately. The function returns false if there exists no path between the node and the root. The function returns true if there exists a path and also adds the path from the root to the given node to the path vector. *findPath()* is called from the *printPath()* function.

The *printPath()* function, if gets true from the *findPath()* function computes the path between the two nodes and prints it. This is done using the concept of least common ansesstor (LCA). The fact that the path from root till LCA of the two nodes will be the same is used here.

1) Find path from root to n1 and store it in a vector or array.
2) Find path from root to n2 and store it in another vector or array.
3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

```
bool findPath(Node *root, vector<int>
                          &path, int k)
{
  if (root == NULL)
       return false;
  path.push_back(root->key);
  if (root->key == k)
       return true;
  if ((root->left && findPath(root->
  left, path, k)) || (root->right &&
  findPath(root->right, path, k)) )
       return true;
  path.pop_back();
  return false;
}

void printPath(Node *root, int n1, int
n2)
{
    vector<int> path1, path2;

    if (!findPath(root, path1, n1) ||
          !findPath(root, path2, n2))
        cout<<"Path does not exist!";
        cout<<endl;
    int i;
    for (i = 0; i < path1.size() &&
          i < path2.size() ; i++) {
        if (path1[i] != path2[i]) {
              break;
              }
        }
         cout<<"The path between "<<n1<<"
         and "<<n2<<" is: ";
         for(int j = path1.size() - 1;
         j >= i - 1; j--) {
              cout<<path1[j]<<" ";
         }
        for(int j = i; j < path2.size();
        j++) {
              cout<<path2[j]<<" ";
              }
}
```

1) Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, paths are found between the nodes and the root respectively and then path arrays are compared.

D. Space Complexity

Space complexity of the program is $O(n)$ because we have used 3 arrays, one of size n, n being the total number of nodes in the tree. One array of size n is used to store the level order traversal of the tree during the input while the remaining two arrays store the path of the node from the root. The path arrays (vectors) are less than size n. Hence the overall space complexity of the program is $O(n)$.

III. RESULTS

The proposed algorithm is run on the below shown full binary tree and the following results were obtained:

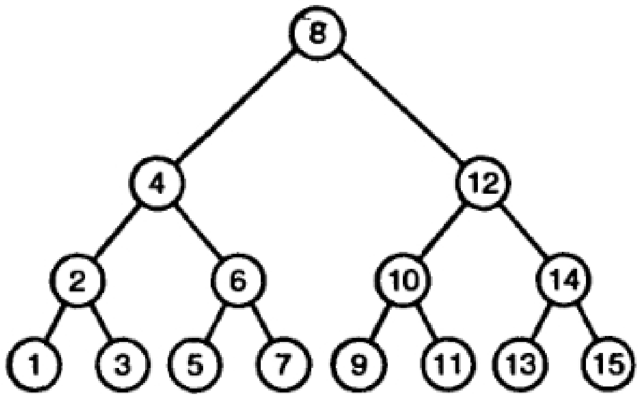

Fig. 5. Input: Full binary tree with 15 nodes

```
Enter total number of nodes: 15
Enter the level order traversal of the tree:
8 4 12 2 6 10 14 1 3 5 7 9 11 13 15
Enter the nodes between which you want to find a
path: 3 13
The path between 3 and 13 is: 3 2 4 8 12 14 13
```

```
Enter total number of nodes: 15
Enter the level order traversal of the tree:
8 4 12 2 6 10 14 1 3 5 7 9 11 13 15
Enter the nodes between which you want to find a
path: 7 9
The path between 3 and 13 is: 7 6 4 8 12 10 9
```

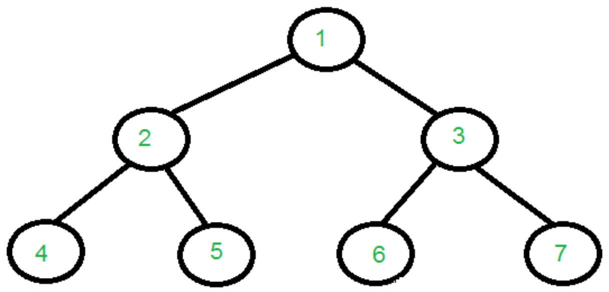

Fig. 6. Input: Full binary tree with 7 nodes

```
Enter total number of nodes: 7
Enter the level order traversal of the tree:
1 2 3 4 5 6 7
Enter the nodes between which you want to find a
path: 2 3
The path between 3 and 13 is: 2 1 3
```

```
Enter total number of nodes: 7
Enter the level order traversal of the tree:
1 2 3 4 5 6 7
Enter the nodes between which you want to find a
path: 5 3
The path between 3 and 13 is: 5 2 1 3
```

IV. DISCUSSION

The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n_1 and n_2 are present in binary tree, we can find LCA using single traversal of binary tree and without extra storage for path arrays. The idea is to traverse the tree starting from root. If any of the given keys (n_1 and n_2) matche with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

References

[1] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Nick Parlante *Binary Trees.* [On-line]. Available: http://cslibrary.stanford.edu/110/BinaryTrees.pdf.

[3] Fayssal El Moufatich *Lowest Common Ancestor(LCA)*. Technische University at Munchen St. Petersburg JASS [2008].

[4] WANG Xingbo *"Analytic Formulas for Computing LCA and Path in Complete Binary Trees"*. International Journal of Scientific and Innovative Mathematical Research (IJSIMR) Volume 3, Issue 3, March 2015, PP 81-88