Width of a Binary Tree

Nishant Doshi
Pursuing Ph.D.
SVNIT, Surat
Gujarat, India

Tarun Sureja
Assistant Professor
R.K. College of Engineering & Technology Rajkot, Gujarat, India

Bhavesh Akbari
Lecturer
R.K. College of Engineering & Technology Rajkot, Gujarat, India

Hiren Savaliya
Lecturer
V.V.P. Engineering College
Rajkot, Gujarat, India

Viraj Daxini
Lecturer
V.V.P. Engineering College
Rajkot, Gujarat, India

ABSTRACT

Till current date in majority books on algorithm and research papers, they talk about height of a binary tree in terms like height balanced binary tree. In this paper the notion of width of a binary tree has been introduced and later the recursive algorithm based on the traversal techniques of the binary tree is given. Later the iterative version of algorithm using the notion of stack is introduced. The width of a binary tree is defined based on the number of nodes at every level. The highest of all is the width of a binary tree. The same concept can be applied to the general tree.

General Terms
Algorithms

Keywords
Algorithm, Binary tree, Stack, Width.

1. INTRODUCTION

Binary tree is a tree in which each internal node including root has at most two children. Leaf node has no children. The root node is at level 0 and so on. The width of a binary tree is defined as the maximum number of nodes in a given tree at some level. For example in figure 2 the width is 2 as level 1 contains two nodes, and all other level contains only one node. If two levels had same number of nodes than we can consider either of them as shown in figure 1. As a literature survey till now the material which was published and application found is height balanced binary tree or we can say AVL tree [1-9]. This paper outlines the concept of width.

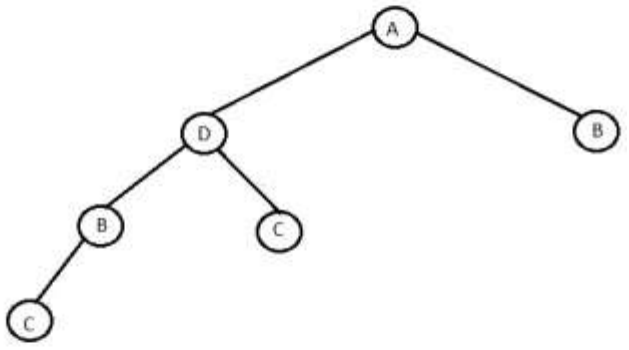

Fig. 1

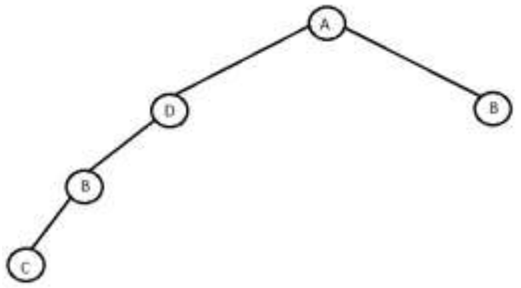

Fig 2

2. RECURSIVE APPROACH

Here we can use the same concept of recursive traversal of binary tree with little modification by introducing the notion of the level. We assume that we have sufficient amount of array to store the number of nodes of each level. For simplicity we start the index of array from zero because tree start from level 0. Let's develop the algorithm step by step. The arguments are the tree pointer and the level number. We can call by passing ROOT and 0 from calling function. The first step will be to check if the tree from that node will be existing or not. Assume *t* is the tree pointer which passed as an argument then the code will be.

```
If (t = NULL)
Then
    Return
```

Now if there is some nodes like left or right than we have to recursively call that part but before it we increment the number of nodes in that level by 1 as follows, where *l* is the level number.

Level[*l*] =Level[*l*] +1

Here we can create the array elements at runtime by different memory operations to save the space. So the final algorithm is as follows.

```
Algorithm Findwidth (TREEPTR *t, l)
{
1.   If(t = NULL)
2.   Then
3.       Return
4.   Level[l]=Level[l] +1
5.   Findwidth(left(t),l+1)
6.   Findwidth(right(t),l+1)
7.   Return
}
```

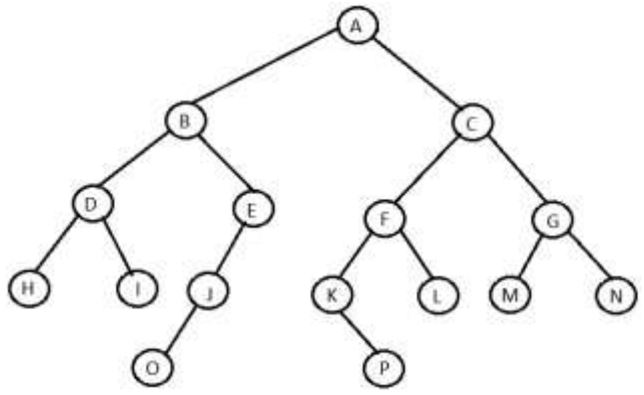

Fig 3

Level	No. of Nodes
0	1
1	2
2	4
3	7
4	2

Table 1

As shown in figure 3 the example of binary tree is given and table 1 summarize the number of nodes at each level. From table 1 and by the definition given in the introduction of this paper, the width of binary tree is 7.

3. ITERATIVE APPROACH

In the binary tree for iterative traversal we use the concept of stack. The same concept is used in this one. Here we assume the linked representation of the node in binary tree and in the stack as given in figure 4 and figure 5 respectively.

LPTR	info	RPTR

Fig 4

Level no.	Addr. of node	Next

Fig 5

Here LPTR is the pointer to the left subtree of a node. Rptr is the pointer to the right subtree of a node. Info is the data of a node. *Level no.* is the level of a node and *addr* is the address of that node. Next is pointer to the next in the list. Now let's develop step by step the algorithm. The first step is to create an empty stack. First push the root and level no. as 0 and address of root as the top of the stack so the step are

```
    Stack S
    PUSH(S, r, 0)
```

Next is loop till we not have an empty stack. There is function called GetTopLevel which return the level no. of the top most node in a stack. We store in some variable called Curr_level and make change in the Level array which was passed as argument. Now pop the top node and if it has left and/or right child than push all of them and continue with the loop. So the code will be as follow.

```
    While (NotEmpty(S))
    Do
        Curr_level=GetTopLevel(S)
        Level [Curr_level] = Level [Curr_level] + 1
        Node = POP(S)
        If (LPTR (Node))
        Then
                PUSH(S, LPTR (Node), Curr_level+1)
        If (RPTR (Node))
        Then
                PUSH(S, RPTR (Node), Curr_level+1)
    Done
```

After above steps we have Level array contains number of nodes in each level so next is to find the maximum in the array. So if we assume that there are N total nodes in a given binary tree than the running time for recursive as well as iterative is O (N).

```
Algorithm Finfwidth (Root r, Level [])
{
1.  Stack S
2.  PUSH(S,r,0)
3.  While (NotEmpty(S))
4.  Do
5.      Curr_level=GetTopLevel(S)
6.      Level[Curr_level]= Level[Curr_level] + 1
7.      Node = POP(S)
8.      If (LPTR(Node))
9.      Then
10.             PUSH(S, LPTR(Node), Curr_level+1)
11.     If (RPTR(Node))
12.     Then
13.             PUSH(S, RPTR(Node), Curr_level+1)
14. Done
15. Return  FindMax(Level)
```

Now let's take an example as given in figure 3 and see the execution of stack as step by step.

Iteration	Stack content			Level array				
0	0	A	Null	0	0	0	0	0
1	1	B		1	0	0	0	0
	1	C	Null					
2	2	D		1	1	0	0	0
	2	E						
	1	C	Null					
3	3	H		1	1	1	0	0
	3	I						
	2	E						
	1	C	Null					
4	3	I		1	1	1	1	0
	2	E						
	1	C	Null					
5	2	E		1	1	1	2	0
	1	C	Null					
6	3	J		1	1	2	2	0

	1	C	Null					
7	4	O		1	1	2	3	0
	1	C	Null					
8	1	C	Null	1	1	2	3	1
9	2	F		1	2	2	3	1
	2	G	Null					
10	3	K		1	2	3	3	1
	3	L						
	2	G	Null					
11	4	P		1	2	3	4	1
	3	L						
	2	G	Null					
12	3	L		1	2	3	4	2
	2	G	Null					
13	2	G	Null	1	2	3	5	2
14	3	M		1	2	4	5	2
	3	N	Null					
15	3	N	Null	1	2	4	6	2
16	EMPTY			1	2	4	7	2

Table 2

Here we can see the numbers of iterations are 16 which are equal to the number of nodes in a binary tree. So this completes the proof that running time of this algorithm is O (N) because to find maximum in a array of size ≤ N requires O (N) so total is O (N). It is required constant time to push and pop the node in the stack. For simplicity the algorithm for PUSH, POP and FindMax is omitted.

4. CONCLUSION

This paper gives the idea of width of a binary tree. This concept can be applied to general tree also. By summing all the Level array element we can get total number of nodes in a given binary tree and based on that we can predict that to make a given tree as height balanced binary tree how many level should be required. In this paper we had only introduce the concept of width of a binary tree, there will be no any applications given which can be useful for this approach. That can be done as future work where width can be useful parameter.

5. REFERENCES

[1] T.H.Cormen, C.E. Leiserson, R. L. Rivest, C. Stein, “Introduction to algorithms”, second edition, McGraw-Hill publication, 2002.

[2] C. C. Foster, Information retrieval: information storage and retrieval using AVL trees, Proceedings of the 1965 20th national conference, p.192-205, August 24-26, 1965, Cleveland, Ohio, United States.

[3] Foster, C.C. A study of A VL trees. GER-12158, Goodyear Aerospace Corp., Akron, Ohio, Apr. 1965.

[4] Harold S. Stone, Introduction to Computer Organization and Data Structures, McGraw-Hill, Inc., New York, NY, 1971.

[5] P. L. Karlton , S. H. Fuller , R. E. Scroggs , E. B. Kaehler, Performance of height-balanced trees, Communications of the ACM, v.19 n.1, p.23-28, Jan. 1976

[6] J.-L. Baer , B. Schwab, A comparison of tree-balancing algorithms, Communications of the ACM, v.20 n.5, p.322-330, May 1977.

[7] J. L. Baer, Weight-balanced trees, Proceedings of the May 19-22, 1975, national computer conference and exposition, May 19-22, 1975, Anaheim, California.

[8] Ralston, R. 2009. ACL2-certified AVL trees. In Proceedings of the Eighth international Workshop on the Acl2 theorem Prover and Its Applications (Boston, Massachusetts, May 11 - 12, 2009). ACL2 '09. ACM, New York, NY, 71-74.

[9] Yi-Ying Zhang, Wen-Cheng Yang, Kee-Bum Kim, Myong-Soon Park, "An AVL Tree-Based Dynamic Key Management in Hierarchical Wireless Sensor Network," iih-msp, pp.298-303, 2008 International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2008.