PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Binary Tree

Dr. Shylaja S S

Binary Tree

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called as the left and right subtree of the root. The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique. The binary tree is a Divide – And – Conquer ready structure.

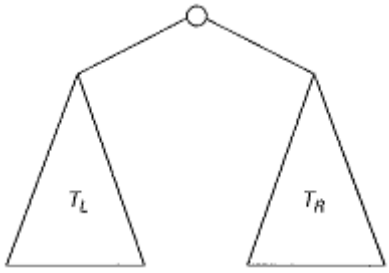


Fig. 1: Standard representation of a Binary Tree

Height of a Binary Tree: Length of the longest path from root to leaf

ALGORITHM Height(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if T = Ø return -1

else return max(Height(T_L), Height(T_R))+ 1

Height of a Binary Tree Analysis:

We measure the problem's instance size by the number of nodes n(T) in a given binary tree T. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions A(n(T)) made by the algorithm are the same.

We have the following recurrence relation for A(n(T)):

$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1$ for $n(T) > 0$,

$A(0) = 0$.

Before we solve this recurrence (can you tell what its solution is?), let us note that addition is not the most frequently executed operation of this algorithm. What is? Checking-and this is very typical for binary tree algorithms-that the

tree is not empty. For example, for the empty tree, the comparison T = Ø is executed once but there are no additions, and for a single-node tree, the comparison and addition numbers are three and one, respectively.

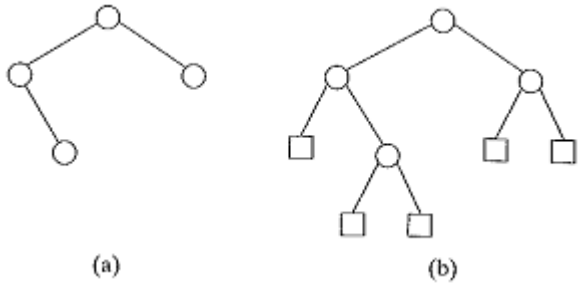


Fig. 2: (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes. The extra nodes (shown by little squares in Fig. 2) are called external; the original nodes (shown by little circles) are called internal. By definition, the extension of the empty binary tree is a single external node.

It is easy to see that the height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node. Thus, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have. Checking Fig. 2 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always one more than the number of internal nodes n:

$$x=n+1 \qquad \text{---------- (1)}$$

To prove this formula, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equation (1).

Note that equation (1) also applies to any nonempty full binary tree, in

which, by definition, every node has either zero or two children: for a **full binary tree**, n and x denote the numbers of parental nodes and leaves, respectively.

Returning to algorithm Height, the number of comparisons to check whether the tree is empty is

$$C(n)=n+x=2n+1,$$

while the number of additions is

$$A(n) = n$$

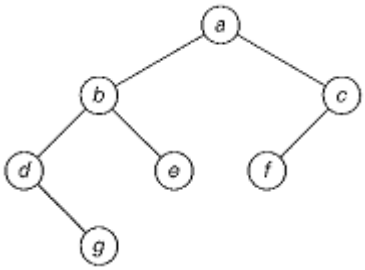
Binary Tree Traversal

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ just by the timing of the root's visit:

In the **preorder traversal**, the root is visited before the left and right subtrees are visited (in that order).

In the **inorder traversal**, the root is visited after visiting its left subtree but before visiting the right subtree.

In the **postorder traversal**, the root is visited after visiting the left and right subtrees (in that order).



Preorder: a, b, d, g, e, c, f
Inorder: d, g, b, e, a, f, c
Postorder: g, d, e, b, f, c, a

Fig. 3: Binary tree and its traversals

Algorithm Inorder(T)
if T ≠ Ø
    Inorder(T_{left})
    print(root of T)
    Inorder(T_{right})

Algorithm Preorder(T)
if T ≠ Ø
    print(root of T)
    Preorder(T_{left})
    Preorder(T_{right})

Algorithm Postorder(T)
if T ≠ Ø
    Postorder(T_{left})
    Postorder(T_{right})
    print(root of T)