**Unit 6: Tree**

1. **Concept and Definition**

A tree is a finite set of elements that is either empty or partitioned into number of disjoint subset. The first subset contains the single element called the root of the tree.The other subset are called childs of the tree.

* **Formal Definition**: This definition is "recursive" in that it defines tree in terms of itself. The definition is also "constructive" in that it describes how to construct a tree.
  1. A single node is a tree. It is "root."
  2. Suppose N is a node and T1, T2, ..., Tk are trees with roots n1, n2, ...,nk, respectively. We can construct a new tree T by making N the parent of the nodes n1, n2, ..., nk. Then, N is the root of T and T1, T2, ..., Tk are subtrees.

eg:

A Tree and graph are hierarchical data structure rather than linear whereas stack,queue, linked list,array are linear data structure.

Application of tree:

The common application of binary tree are as follows:

* **Directory Structure of a file system/storage**: tree is used to manage file structure by operating system.
* **Structure of arithmetic expression:** the tree is also used in parsing in the theory of compiler design.
* **Hierarchy of an organization:** The tree diagram is used to model the hierarchical diagram.

A common example of tree is Binary Tree

1. **Binary Tree**

A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are are themselves binary trees, called the left and right subtrees of the original tree. A left or right subtree can be empty. That is, Binary tree have at most two childs. Tree does not contain any cycle. Each element of a binary tree is called node of the tree. The following figure is an example of binary tree which consist seven nodes with A as its root. Its left subtree is rooted at B and its right subtree is rooted at C.



The absence of a branch indicate an empty subtree. Following figure are not a binary tree.







* If A is the root of the binary tree and B is the root of its left or right subtree, then A is said to be the ***Father*** of B And B is said to be left or right son of A.
* A node that has no sons is called a ***leaf***.
* Node n1 is an ***Ancestor*** of node n2 ( and n2 is ***descendant*** of n1) if n1 is either the father of n2 or the father of some ancestor of n2. For example in figure 6.2, A is an Ancestor of G, and G is an descendant of C. But D is neither an ancestor nor a descendant of C
* Two nodes are **brother** if they are left and right sons of the same father.
* A node n2 is a ***left descendant*** of node n1 if n2 is either the left son of n1 or a descendant of the left son of n1. A **right descendant** may be similarly defined.
* Going from the leaves to the root is called “***climbing”***  the tree, and going from the root to the leaves is called “descending ” the tree.

1. **Operation:**

There are a number of primitive operations that can be performed on a binary tree. Following are the operation that can be performed on node nd pointed by a pointer p.

1. info(p) : Returns the information/content of node nd.
2. left(p): Returns Pointer to the left son of node nd, return null if left son is empty
3. right(p): Returns Pointer to the right son of node nd, return null if right son is empty
4. father(p): Returns Pointer to the father of node nd, return null if father is empty.
5. brother(p): Returns Pointer to the brother of node nd, return null if brother is not present
6. isleft(p): retuns the value true if node nd is left son of some other node in a tree.
7. isright(p): returns the value true if node nd is right son of some other node in a tree.
8. maketree(x): creates a new binary tree consisting of a single node with information field x and returns a pointer to that node.
9. setleft(p, x): accepts a pointer p to a binary tree node with no left son and creates a left son of node(x) with information field x.
10. setrightt(p, x): accepts a pointer p to a binary tree node with no right son and creates a right son of node(x) with information field x.

maketree(x),setleft(p,x),setright(p,x) are used when constructing tree.

The function isleft(p), isright(p), and brother(p) can be implemented using the functions left(p), right(p), and father(p). For example isleft may be implemented as:  
q = father(p);

if (q==null)

return(false);

if(left(q)==p) // p points to the root

return(true);

return false;

or simply father(p) && p==left(father(p)). isright is also implemented in similar manner or by callingisleft.brother(p).

The function brother (p) is implementd as

if(father(p) == null)

return(null);

if(isleft(p))

return(right(father(p));

return(left(father(p));

1. **Types of Binary Tree**
   1. A binary tree is called ***Strictly binary tree*** if every non leaf node in a binary tree has nonempty left and right subtrees . For eg.



A strictly binary tree with n leaves always contains 2 n - 1 nodes.

The **level** of a node in a binary tree is defined as follows: The root of tree has level 0, and the level of any other node in the tree is one more than the level of its father.



The Depth of a binary tree is the maximum level of any leaf in the tree. This equals the length of longest path from the root to any in the binary tree.The depth of above (fig 6.6) binary tree is 3.

* 1. **A complete binary tree** of depth d is the strictly binary tree all of whose leaves are at level d. Figure 6.7 illustrates the complete binary tree of depth 3.



If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l+1. The root in a binary tree is at level 0 and it can contain 2l nodes at level l.

A complete binary tree has leaf at same depth d that contains exactly 2d nodes at each level l between 0 and d.

* 1. A binary tree of depth d is an **almost complete binary tree** if:
     1. Any node nd at level less than d-1 has two sons.
     2. for any node nd in the tree with a right descendant at level d, nd must have a left son and every left descendants of nd is either a leaf at level d or has two sons. fig(from book pg 253 c)

1. **Huffman algorithm**

Huffman codes are used to compress data by representing each alphabet by unique binary codes in an optimal way. As an example consider the file of 100,000 characters with the following frequency distribution assuming that there are only 7 characters

f(a) = 40,000 , f(b) = 20,000 , f(c) = 15,000 , f(d) = 12,000 , f(e) = 8,000 , f(f) = 3,000 , f(g) = 2,000.

Here fixed length code for 7 characters we need 3 bits to represent all characters like a = 000 , b = 001 , c = 010 , d = 011 , e = 100 , f = 101 , g = 110.

Total number of bits required due to fixed length code is 300,000.

Now consider variable length character so that character with highest frequency is given smaller codes like

a =0,b=10,c=110,d=1110,e=11111,f=111101,g=111100

Total number of bits required due to variable length code is

40,000\*1 + 20,000\*2 + 15,000\*3 + 12,000\*4 + 8,000\*5 + 3,000\*6 + 2,000\*6. i.e. 243,000 bits

Here we saved approximately 19% of the space.

1. **Application of Binary Tree**
   1. A binary tree is a useful data structure when two - way decisions must be made at each point in a process. For example , to find duplicate in the given list. First technique is to compare each number with other number in the list. However this involves large number of comparison.The number of comparison is reduced by using a binary tree. The first number in the list is assumed as a root of a binary tree with empty left and right subtrees. Each successive number in the list is then compared to the number in the root. If it matches, we have a duplicate, If it is smaller, we examine the left subtree; if it is larger, we examine right subtree. If the subtree is not empty, the number is not a duplicate and is placed into a new node at that position in the tree. If the subtree is nonempty, we compare the number to the contents of the root of the subtree and the entire process is repeated with the subtree. An algorithm for doing this follows:pg255;

Following figure illustrate the tree constructed from 20……

* 1. Another common operation is traverse a binary tree; ie. to pass through the tree enumerating each of its node once. Traversing a binary tree involves visiting the roots and traversing its left and right subtrees.The only difference among the methods is the order in which these three operations are performed. There are three different technique which are as follows:

1. **Pre-Order:** to traverse a nonempty binary tree in preorder(also known as depth-first-order), we perform following three operation.
   1. visit the root
   2. traverse the left sub tree in preorder
   3. traverse the right sub tree in preorder
2. **In-order:** to traverse a nonempty binary tree in inorder( or symmetric order), we perform following three operation.
   1. traverse the left sub tree in inorder
   2. visit the root
   3. traverse the right sub tree in inorder
3. **Postorder:** to traverse a nonempty binary tree in postorder:
   1. traverse the left sub tree in postorder
   2. traverse the right sub tree in postorder
   3. visit root.

Following figure illustrate two binary trees and their traversal in preorder, inorder, and postorder.

pg257

* 1. Binary search tree: (eg studied later)
  2. Expression are also represented in binary tree

1. **Binary Search Tree**

A binary search tree (BST) is an ordered collection of items organized into nodes which have two pointers: left and right.The values in a BST must satisfy the following rule.The BST Property: If node x is to the left of node y, then x≤y. If x is right of y, then x>y. A BST can hold any type of item, as long as the item can be ordered (defined <, >, ==).

A BINARY SEARCH TREE is a binary tree in symmetric order.Symmetric order means that:

* every node has a key
* every node’s key is

larger than all keys in its left subtree

smaller than all keys in its right subtree

Let us construct a binary tree from following input as:

There are a number of operations on BST’s that are important to understand. We will discuss some of the basic operations such as how to insert a node into a BST, how to delete a node from a BST and how to search for a node in a BST.

* 1. Insertion (N,T) : insert a node N to tree T.
  2. Deletion (N,T) : delete a node N from tree T.
  3. Searching (N,T) : find a node N from tree T.

1. **Inserting a node**

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows.

Assume we insert a node N to tree T. if the tree is empty, then we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Insert(N, T) = N if T is empty

= insert(N, T.left) if N < T

= insert(N, T.right) if N > T

1. **Deleting a node**

A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9. Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

**Case 1 : The node to delete is a leaf node**

This is a very easy case. Just delete the node. We are done

**Case 2 : The node to delete is a node with one child.**

This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.

**Case 3: The node to delete is a node with two children**

This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.

Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.

1. **Searching for a node:**

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows.

If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not.

Search(N, T) = false if T is empty

= true if T = N

= search(N, T.left) if N < T

= search(N, T.right) if N > T