13

BS TREES

13.1 Binary Trees

A binary Tree T is defined as a finite set of element, called nodes such that :

(a) T is empty (called the empty tree or null tree), or

(b) T contains distinguished node R called the root of T and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2

If T_1 is non empty then its roots is called left successor, if T_2 is non empty then its root is called right successor.

If a node has no successor then it is called terminal node. We can also view R as parent

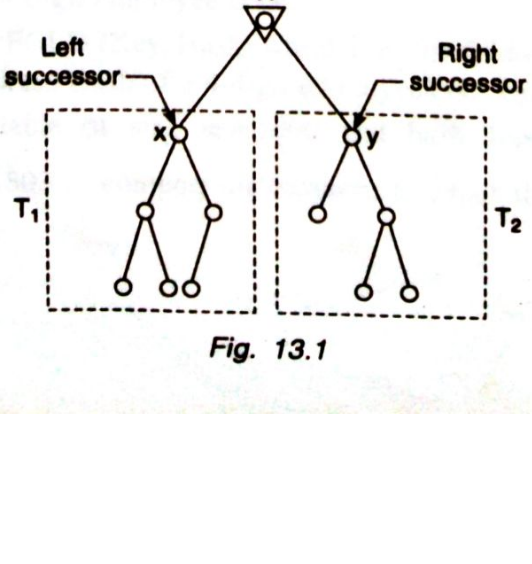


Fig. 13.1

and x and y as its Left and right child. R is also predecessor to x and y or x and y are successors to R.

Complete Binary Tree

A binary tree T is said to be complete binary tree if all its levels, except possibly the last level have maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

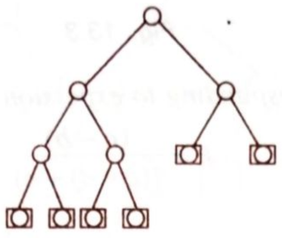


Fig. 13.2

EXAMPLE 13.1. *Show that there can be at most* $\lfloor \log_2 (n+1) \rfloor$ *nodes in a complete binary tree.*

SOLUTION : We can have $2^0, 2^1, 2^2, 2^3, ..., 2^k$ nodes, in a k-level binary tree of n nodes, then

$$2^0 + 2^1 + 2^2 + 2^3 + 2^k \geq n$$

$$\frac{1 . (2^k - 1)}{2 - 1} \geq n.$$

$$2^k - 1 \geq n$$

$$2^k \geq n + 1$$

Taking the log to the base 2, on both sides

$$k \geq \log_2 (n + 1)$$

or $$k_{\max} = \lfloor \log_2 (n+1) \rfloor.$$

and, $$D_n = k + 1 \text{ (Depth of tree)}$$

$$\boxed{D_n = \log_2 (n + 1) + 1}$$

13.1.1 Extended Binary Trees (2 Trees)

A binary tree T is said to be 2-tree or an extended binary tree if each node N has either 0 or 2 children. In such a case the nodes with 2 children are called internal nodes, and the nodes with 0 children are called external nodes. Example of 2-tree are expressions with binary operations, where operators represent internal node and identifiers are external node.

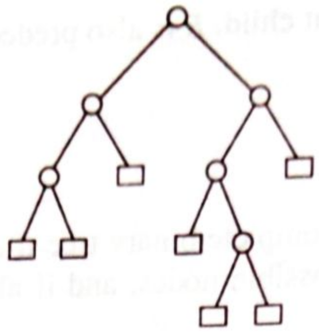
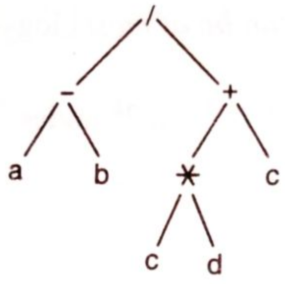


Fig. 13.3

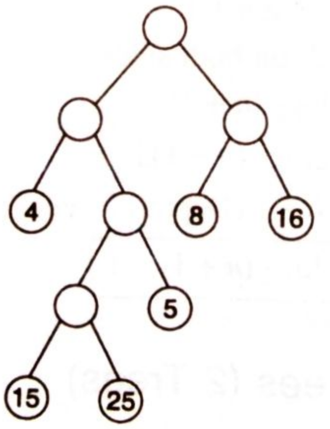
EXAMPLE 13.2. *Find 2-tree corresponding to expession*

$$E = \frac{(a-b)}{((c*d)+e)}$$

SOLUTION : $E = \frac{(a-b)}{((c*d)+e)}$



EXAMPLE 13.3. *Find the weighted length P of the given 2-tree.*



SOLUTION : Multiply each weight ω by the path length from root of T, to the node containing weight and then sum all these

$$P = 2.4 + 4.15 + 4.25 + 3.5 + 2.8 + 2.16$$
$$= 8 + 60 + 100 + 15 + 16 + 32$$
$$= 231.$$

Path length in extended binary tree :

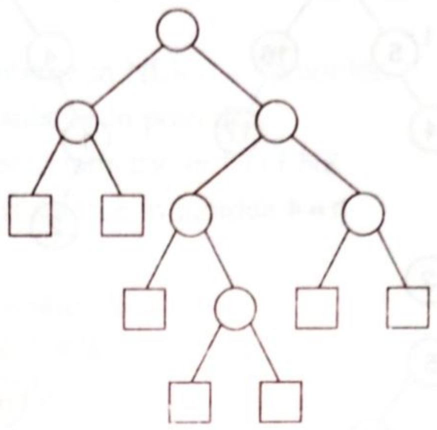
External path length : It is path length from root to node having 0 children.

Internal path length : It is path length from root to a node having 2 children.

Note : In a 2-tree/extended tree node has either 0 or 2 children.

EXAMPLE 13.4. ***If n(E) represents number of external nodes and n(I) represents number of internal nodes in a 2-tree prove that n(E) = n(I) + 1, if E and I and external and internal path lengths and n is number of internal nodes, then also prove that E = I + 2n.***

SOLUTION :



We can easily see that

$$n(E) = 7 \quad \text{and} \quad n(I) = 6$$

$\therefore$ $\qquad n(E) = n(I) + 1 \qquad \ldots(1)$

Also, $\qquad E = 2 + 2 + 3 + 4 + 4 + 3 + 3 = 21$

$\qquad I = 0 + 1 + 1 + 2 + 2 + 3 = 9$

and $\qquad n = 6$

therefore $\qquad E = I + 2n$

13.2 Binary Sequence Tree

It is a binary trees as shown below :

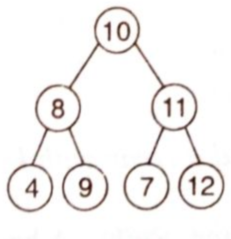


Fig. 13.4

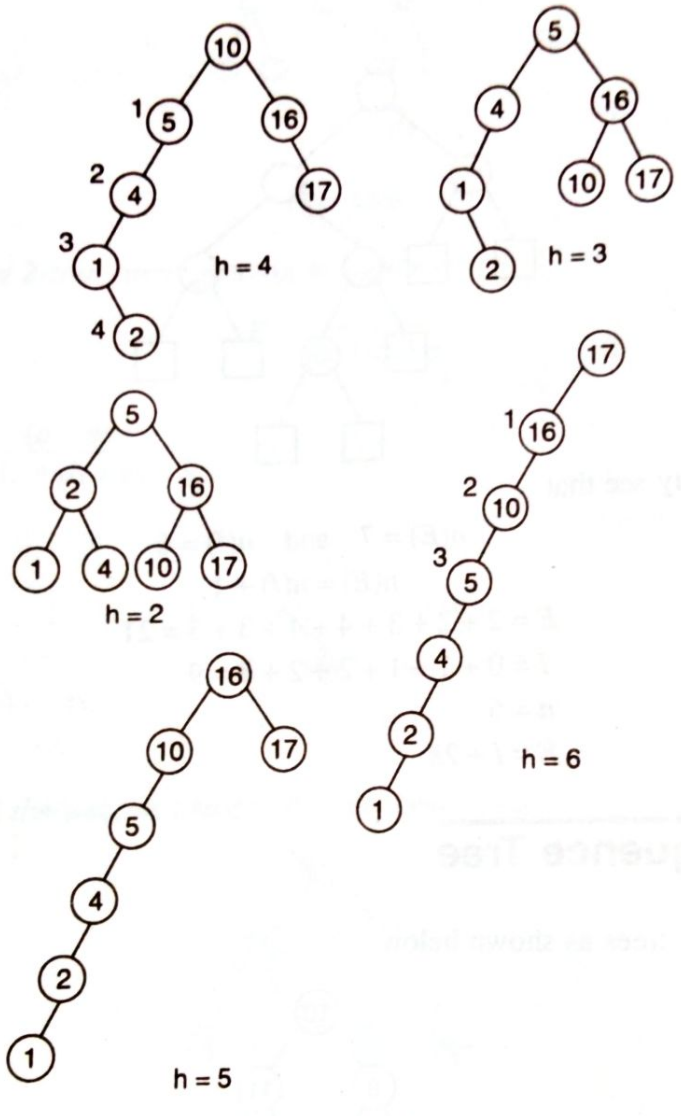
Such a tree can be represented by a linked data structure in which each node is an object. In addition to the key field, each node contains field left, right, and P that most points to the nodes corresponding to its left child, right child and its parent respectively. If any of these is missing field contains the value NIL. Root node is the only node whose parent field is NIL.

The keys in a binary search tree is always stored statisfying the **binary-search-tree property.**

If x is any node in a tree and y is its left subtrees
then key $[x] \geq$ key $[y]$ and if z is right subtree
then key $[x] \leq$ key $[z]$

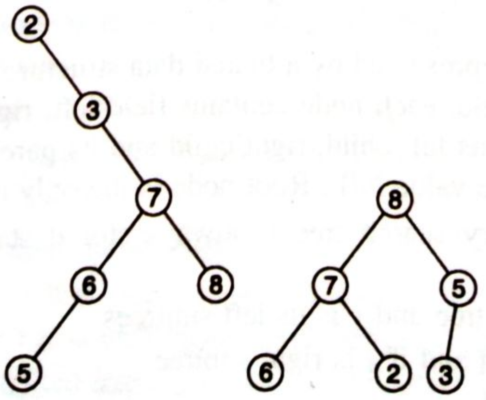
EXAMPLE 13.5. *Draw binary search trees of height 2, 3, 4, 5 and 6 on the set of keys 1, 4, 5, 10, 16, 17, 2.*

SOLUTION :



EXAMPLE 13.6. *Construct a BST and a Heap with keys 2, 3, 5, 6, 7, 8.*

SOLUTION : *Left* : A binary search tree. *Right* : A heap but not a binary search tree.



13.3 Tree Traversal

Traversing a binary tree means that passing through the tree, enumerating each of its node once *i.e.*, each node is visited exactly once.

Preorder (NLR or depth first order)

1. Visit the root
2. Traverse the left subtree in NLR, *i.e.*, preorder.
3. Traverse the right subtree in preorders.

Inorder Traversal (or symmetric order) LNR

1. Traverse the the left subtree in inorder.
2. Visit the root
3. Traverse the right subtree in inorder.

Post order Traversal (LRN)

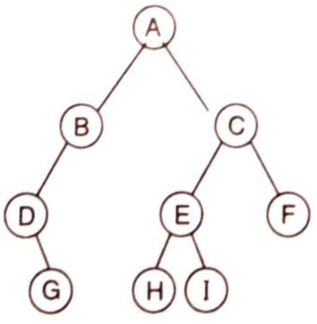
1. Traverse the left tree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

Preorder : This can be implemented using stack, Initially push NULL onto stack then set $P \leftarrow$ Root then repeat the folowing steps until $P \leftarrow$ NULL or while $P \neq$ NULL

1. Process the root and proceed down the left path, pushing right child Right $[x]$ onto stack and processing down each left child, Left $[x]$ Traversing stops when left $[P] \leftarrow$ NULL
2. Pop and assign to P the top element of stack. If $P \neq$ NULL, then goto step 1, otherwise exit.

EXAMPLE 13.7. *Give the preorder traversal of the following Binary tree.*

SOLUTION :



1. Push NULL to stack, S, ϕ
2. Process A, push C to S, $S : \phi, C$
3. Process B, D; push G

   $S : \phi, C, G$
4. Pop G and process
5. Pop C and process
6. Push F, *i.e.*, ϕ, F
7. Process E, Push I, $S : \phi\ F, I$
8. Process H.

9. pop *I* and process

10. Pop *F* and process

This	**ABDGCEHIF**	**⟵ Preorder**
Inorder	:	**DGBAHEICF**
Postorder	:	**GDBHIEFCA**

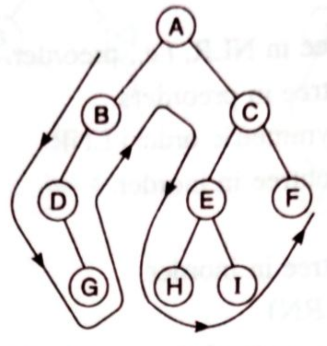


Fig. 13.5. Shows preorder traversal.

Preorder-Tree-Walk (x)

```
1. Push (S, ϕ)
2.   P ← x
3. while (Left [P] ≠ NULL)
4.   print (key (P))
5.      P ← Left [P]
6.      Push (S, Right [P])
7.  P ← pop (S)
8.  If P ≠ NULL goto step 3.
```

There are exactly n calls, so the running time is $\theta(n)$.

Inorder Traversal

```
INORDER-TREE-WALK (x)
If x ≠ NIL then
      INORDER-TREE-WALK (Left [x])
            Print key [x]
      INORDER-TREE-WALK (Right [x])
```

Inorder Algo

Initially push NULL onto stack and then set $P \leftarrow$ root then repeat the following steps.

1. **Proceed down the left-most path rooted at P, pushing each node onto stack and stopping when no left child is left to be pushed.**
2. **pop and process the nodes on stack. If NULL is popped then exit. If a node with right child is popped and processed set $P \leftarrow$ Right [P] and goto step 1.**

EXAMPLE 13.8. *Give the stack implementation of Inorder traversal.*

SOLUTION :

```
INORDER (Key, Left, Right, x)
1.   top ← 1, push (S, NULL)
          p ← x
2.   While (P ≠ NULL)
          top ← top + 1
          S[top] ← key [P]
               P ← Left [P]
     End while
3.   Key [P] ← S [top]
     top ← top - 1
4.   While (P ≠ NULL)
     print key [P]
          If Right [P] ≠ NULL
               P ← Right [P]
               goto step 2
          end if
     key [P] ← S [top]
          top ← top - 1
     end while
     return.
```

EXAMPLE 13.9. *Give the recursive postorder tree-walk LRN.*

SOLUTION :

```
POST-ORDER-TREE-WALK (x)
     if x ≠ NIL
          POST-ORDER-TREE-WALK (Left [x])
          POST-ORDER-TREE-WALK (Right [x])
          Print Key [x]
```

Running time is $\theta(n)$.

Post-Order Algo

1. **Proceed down the left most path. At each node *N* push *N* onto stack and if *N* has right child, push – Right (*N*) onto stack.**
2. **Pop and process positive nodes. If –ve node *N* is popped set ptr = – *N* and repeat step 1.**

EXAMPLE 13.10. *Let T be an ordered tree with more than one node. Is it possible that the preorder traversal of T visits the nodes in the same order as the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur. Likewise, is it possible that the preorder traversal of T visits the nodes in the reverse order of the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur.*

SOLUTION : It is not possible for the postorder and preorder travesal of a tree with more than one node to visit the nodes in the same order. A preorder traversal will always visit the root node first, while a postorder traversal node will always visit an external node first.

It is possible for a preorder and a postorder traversal to visit the nodes in the reverse order. Consider the case of a tree with only two nodes.

EXAMPLE 13.11. *The inorder and preorder traversals of tree T is given as below :*

Inorder : E A C K F H D B G

Preorder : F A E K C D H G B

Construct a binary tree.

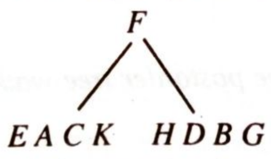
SOLUTION :

(a) Root of tree is obtained by choosing first node in preorder. Here *F*.

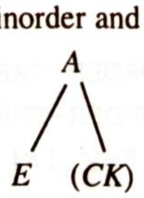
(b) Left child of *F* is all nodes prior to *F* in inorder and all nodes to right of *F* forms right child.

Looking at inorder

(*E A C K*) *F* (*H D B G*)

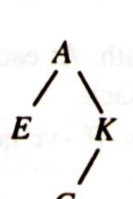


In *EACK* *A* occurs as first node in preorder therefore *A* forms root.
and *E* occurs to left of *A* in inorder and *CK* to right of *A* in inorder.

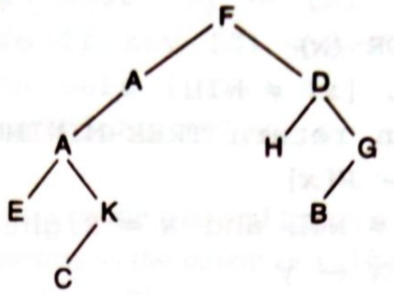


in *CK*, *K* occurs first in preorder,
C occurs to left of *K* in inorder

Thus



Thus we construct the tree as belows :



13.4 Querying a Binary Search Tree

Searching : finding a node with key value k in a binary tree

```
Tree-Search [x, k]
        If x = NIL or k = key [x]
                then return x
        If k < key [x]
                then return tree-search (Left [x], k)
                else return (Right [x], k)
```

running time $O(h)$ in a *ht* . h of a tree.

```
ITERATIVE-TREE-SEARCH (x, k)
while (x ≠ NIL and k ≠ key [x])
        do if k < key [x]
                then x ← Left [x]
                else x ← Right [x]
return x.
```

Minimum and Max

Using binary tree property we can write.

```
TREE-MINIMUM (x)
        While Left [x] ≠ NIL
                do x ← Left [x]
        return x
TREE-MAXIMUM (x)
        while Right [x] ≠ NIL
                do x ← Right [x]
        return x.
```

Both of these procedure run in $O(h)$ time.