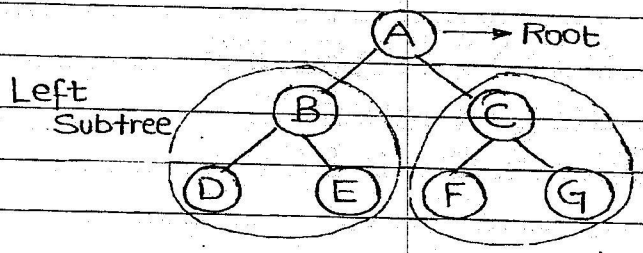
Page No. 1
Date : 20/03/16

Chapter 6 : Trees & Graphs Algo-

≡ Tree Traversals -

1. Pre-Order 2. Post-order 3. In-order

Ex-1



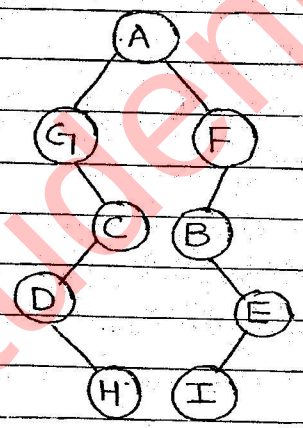
Recursion is the best idea for tree problems as all nodes have some similar quality

Pre-order ~ Root LST RST
A B D E C F G

Post-order ~ D E B F G G A (LST RST Root)

In-order ~ D B E A F C G (LST Root RST)

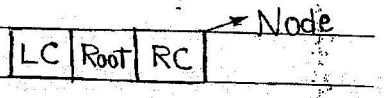
Ex-2



Pre-order → A G C D H F B E I

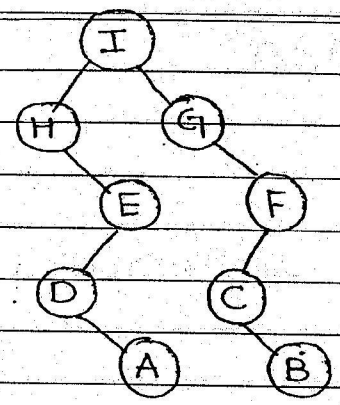
Post-order → H D C G I E B F A

In-order → G D H C A B I E F



Structure array

X-3



Post-order →

ADEHBCFGI

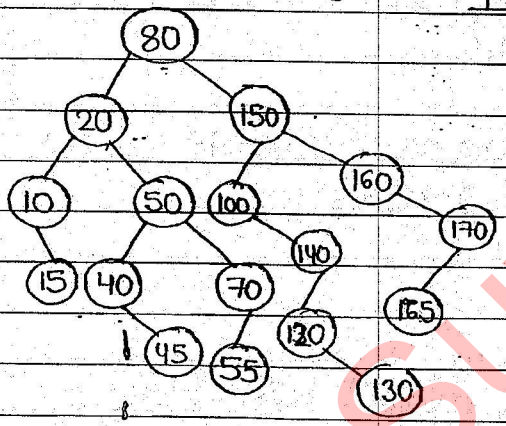
In-order →

HDAEIGBCBF

Pre-order →

IHEDAGFCB

X-4



Pre-order - 80, 20, 10, 15, 50, 40, 45, 70, 55, 150, 100, 140, 120, 130, 160, 170, 165

Post-order - 15, 10, 45, 40, 55, 70, 50, 20, 130, 120, 140, 100, 165, 170, 160, 150, 80

In-order - 10, 15, 20, 40, 45, 50, 55, 70, 80, 100, 120, 130, 140, 150, 160, 165, 170

BST → In-order → ascending order

Meaning of tree given → root address given.

With the help of links, accessible to every node.

```
Pre-order (root) ⇒T(n)
{
printf (root → data); ⇒c
pre-order (root → left); ⇒T(n/2)
pre-order (root → right); ⇒T(n/2)
}
```

Recursive program for pre-order.

$T(n) = 2T(n/2) + c$ // equal partitions
$= O(n)$

$T(n) = T(n-1) + c$ // worst partitions
$= O(n)$

Every case (BC, WC, AC) = O(n) [In, Pre, Post]
Pre-order, In-order and Post-order will take O(n) [Every case] on n nodes binary tree. because every node have to be visited or traversed.

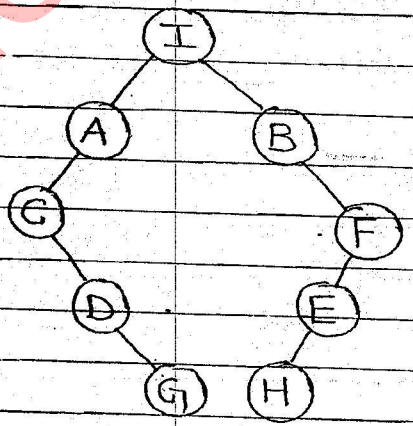
Stack-size — logn — Best case
n — Worst case.

i/P — 2 Bytes.

Space-complexity :- i/P + extra (stack)
O(n) WC O(logn) BC.

☞ In-order traversal will give the ascending order of elements.

Ex-5 Consider the following binary tree data.
Pre-order : I, A, C, D, G, B, F, E, H
In-order : C, D, G, A, I, B, H, E, F
find Post-order : G, D, C, A, H, E, F, B, I

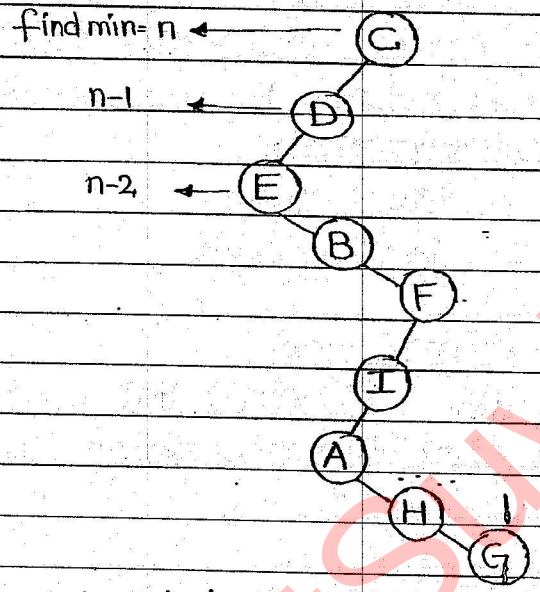


Ex-6 Consider the following binary tree data-

Post-order- G, H, A, I, F, B, E, D, C

In-order - E, B, A, H, G, I, F, D, C.

Pre-order : C, D, E, B, F, I, A, H, G



Finding root O(1)
Traversing O(n)
(Linear Search)
→ for 1- element
O(n²)
→ for n-elements (n-level)
n times linear search.

✓ To construct binary tree for given (pre-order)(in-order) (post-order) will take $O(n^2)$

Ex-7 Consider the following BST data-

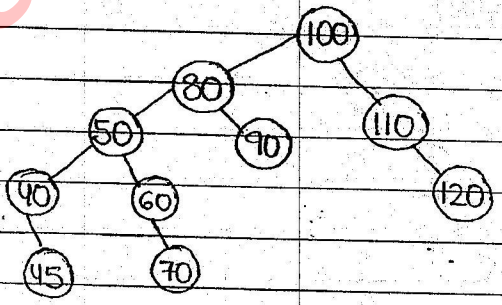
Pre-order : 100, 80, 50, 40, 45, 60, 70, 90, 110, 120

→ In-order : 40 45 50 60 70 80 90 100 110 120

What is post-order ?

Ascending order
(Bcoz of BST)

O(nlogn)
[n times binary search]

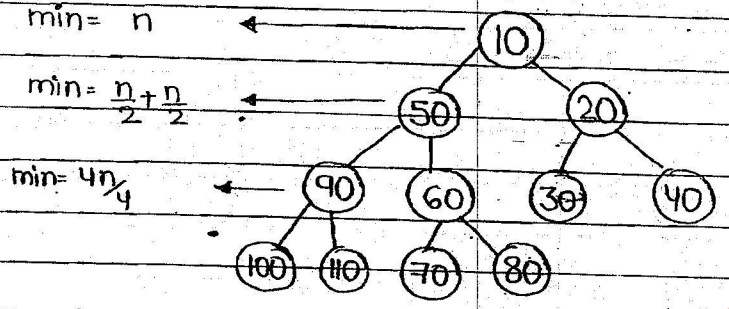


45, 40, 70, 60, 50, 90, 80, 120, 110, 100

Ex-8 Consider the following min-heap tree data.

In-order : 100, 90, 110, 50, 70, 60, 80, 10, 30, 20, 40

? Post-order : 100, 110, 90, 70, 80, 60, 50, 30, 40, 20, 10



finding root = O(n)
searching = O(1)
→ for 1-element

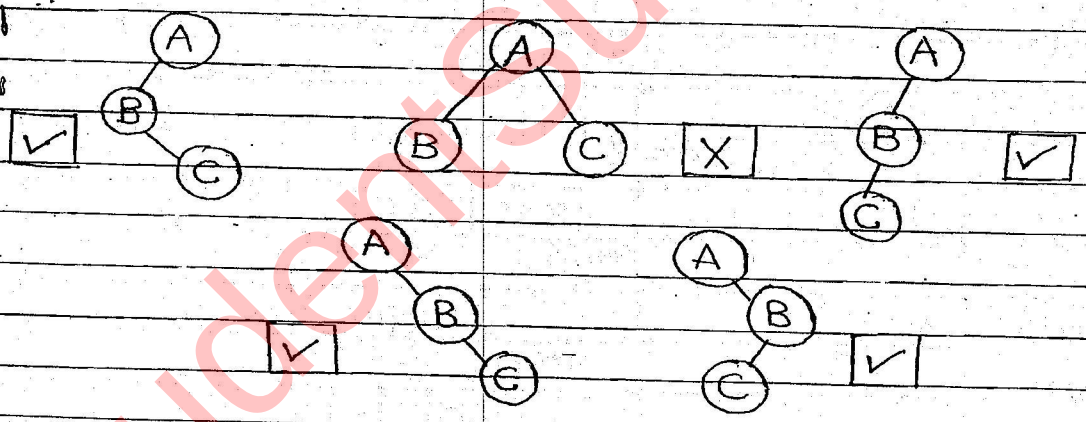
$\log_2 n$

$O(n \log_2 n)$

Ex-9 Consider the following binary tree data.

Pre-order : A B C

Post-order : C B A ? In-order : Not unique



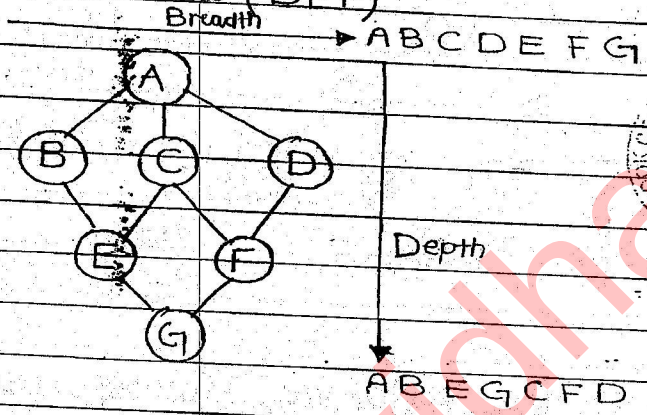
✍ To construct unique binary tree, in-order is compulsory.

Pre + Post = unique binary tree not possible

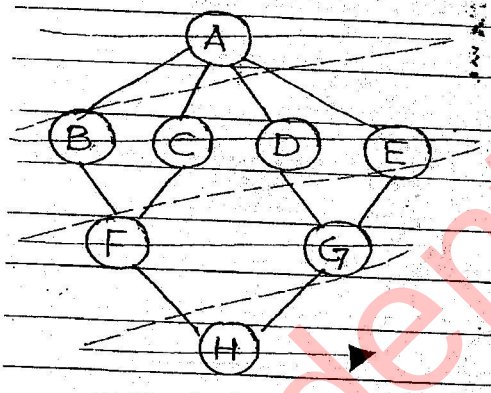
✍ Time complexity = $O(2^n)$ [Manual checking]

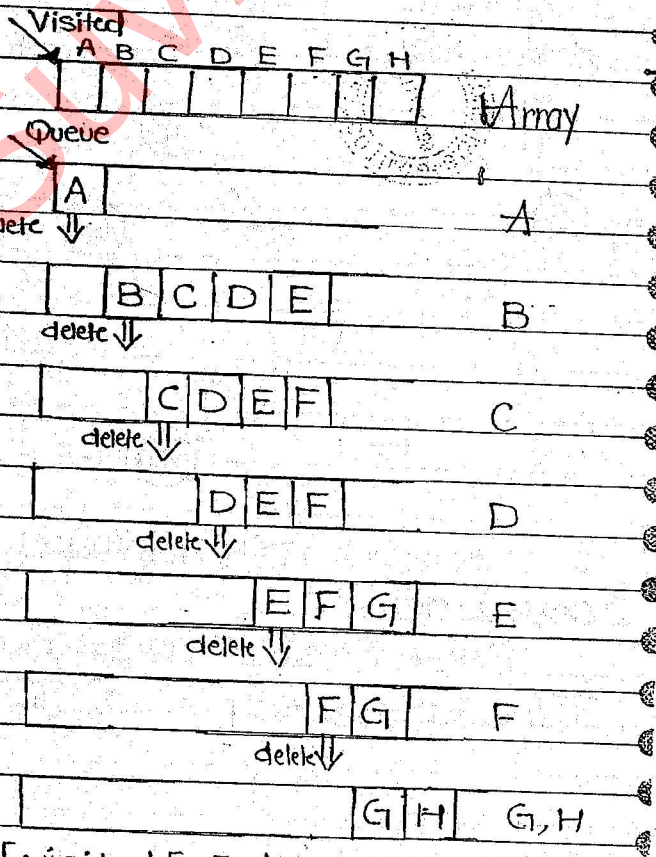
Graph Traversals-

1. Breadth first traversal (BFT)
2. Depth first traversal (DFT)



I Breadth - first :-





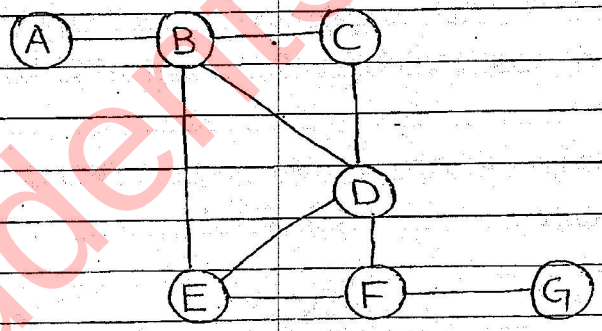

```
BFT(v)
{
1. visited(v)=1
2. add(v,Q)
3. while(Q is not empty){
     x= delete(Q);
     printf(x);
     for all w adj to x {
     if(w is not visited){visited[w]=1; add(w,Q);}}}
}
```

- while loop covering nodes & for loop edges.
- size of the queue is V
- To implement BFT, queue data structure is used.
- Time complexity = O(V+E) [Every case]
- Space complexity = i/p + extra
  - adjacency list O(V+E)
  - queue O(V)
  - visited array O(V)
  - ⇒ O(V+E)
- Level order traversal → another name for BFT.

Ex-1
Consider the following graph as previous one. Find correct BFT from the following -

(i) ABCDEFGH ✓
(ii) CAFBDEHG ✓
(iii) ADCBEGFH ✓
(iv) AFGBCDEA ✓
(v) EAGCDFBH ✗

Ex-2 Consider following graph -



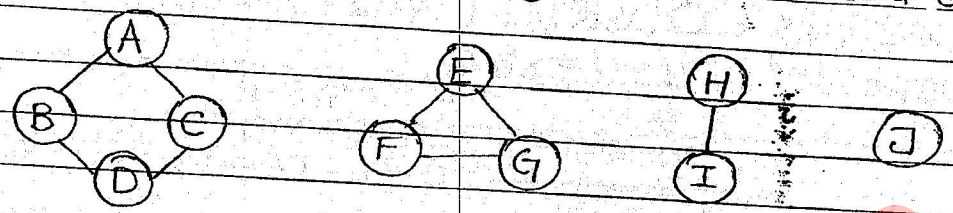
Check out?

(i) EBDFCAG ✓
(ii) GFCEDBA ✗
(iii) CBDAEFG ✓
(iv) DBCEFAG ✓

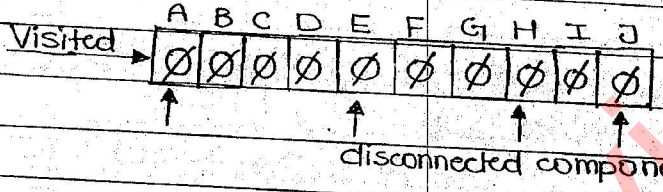
Page No. 8
Date :20/03/16

Applications of BFT →

1. Verify whether a given graph is connected or not.

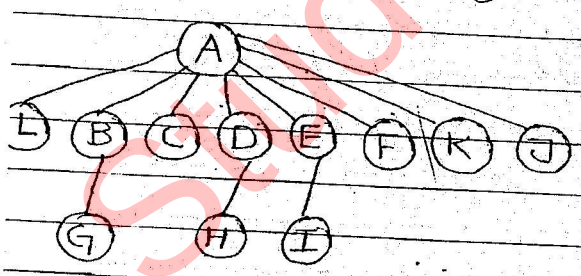


ABCD EFG HI J

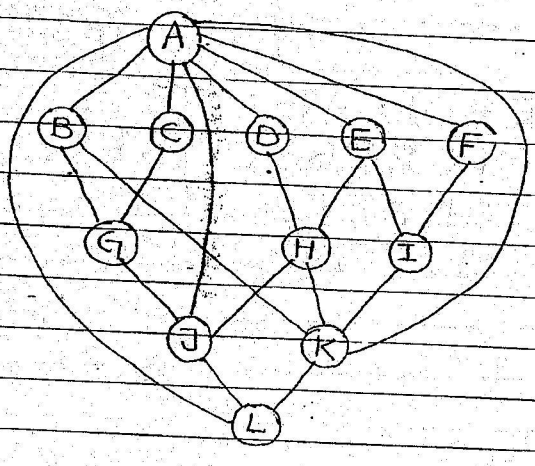


4 times BFT
4 connected components.

2. find out the connected components in given graph.
3. Helps in checking whether given graph contain cycle or not.
4. Linked list is a directed graph. (an apply BFT on it to check whether cycle is present or not.
5. finding shortest path from a single sourc in an unweighted graph. O(E+V)



BFT-tree

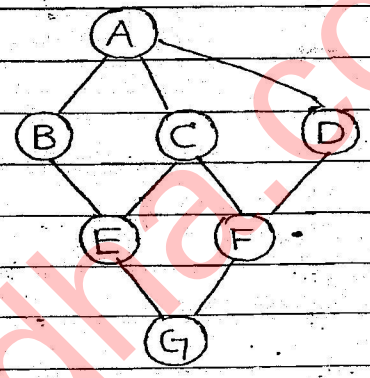


Page No.9
Date :20/03/16

5. Verify whether a graph is bipartite is not.

II Depth-First:

```
DFT(v) {
1   visited(v)=1;
2   printf(v);
3   for all w adj to v
    {
    i  if(w is not visited)
    ii   DFT(w); }}
```



	A	B	C	D	E	F	G
	∅	∅	∅	∅	∅	∅	∅

Visited

DFT(A)
- v 2
- 3 w = B,C,D
- v B DFT(B)
  - 2. v 2
  - 3. w = A,E
  - 1. ✗ E DFT(E)
    - 2. v 2
    - 3. w = B,C,G
    - 1. ✗ G DFT(G)
      - 2. v 2
      - 3. w = E,F
      - 1. ✗ F DFT(F)
        - 2. v 2
        - 3. w = C,D
        - 1. ✗,C DFT(C)
          - 2. v 2
          - 3. w = A,E,F
          - i. ✗,✗,✗
      - 1. ✗ F DFT(D)
        - v 2
        - 3. w = A,F
        - 1. ✗ ✗

Back-track

Output: A B E G F C D

Stack-size = 6

No. of nodes = 7

1. To implement DFT, we are using stack.
2. Time complexity - O(V+E)
   Space complexity - s/p + stack + extra
   (list) O(V+E) O(V) O(V) ⇒ O(V+E)
   (matrix) TC = O(V²) SC = O(V²)

Ex-1 Consider the graph as previous one. find correct DFT's.

i) ABEGFCD [✓] (adjacent, adjacent, Back-track)
ii) ABECFDG [✓]
iii) DFCEBAG [✓]
iv) CFDABEG [✓] S.size=7

Ex-2 Consider the following graph - (same as given in ex-2 of BFT). Check out?

(i) edbcafg [✓]
(ii) fdebagc [✗]
(iii) abdfgce [✗]
(iv) dcbefgaa [✓] 1/14 2/13 3/12 4/9 5/8 6/A 10/11
(v) befgdca [✓] 1/14 2/11 3/10 4/5 5/9 7/8 12/13 (start/finish)

Applications of DFT →

1. Verify whether given graph is connected or not.
2. Check for the number of connected components.
3. finding a cycle in graph.
4. DFT cannot work for finding single source shortest path in the given un-weighed graph.
5. Using DFT, we can verify given directed graph is strongly connected or not.
6. Using DFT, we can verify given vertex is articulation point or not.
   (apply DFT two times)