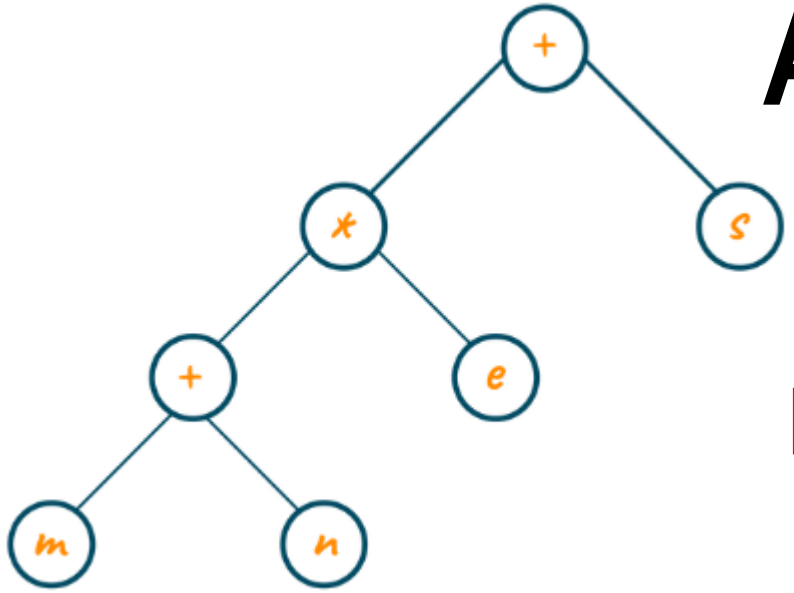
Data Structures and Algorithm



Moazzam Ali Sahi

Lecture # 21

Expression tree

Last Lecture

- Tree Data Structure
- Implementation of Tree DS
- Types of trees

This Lecture

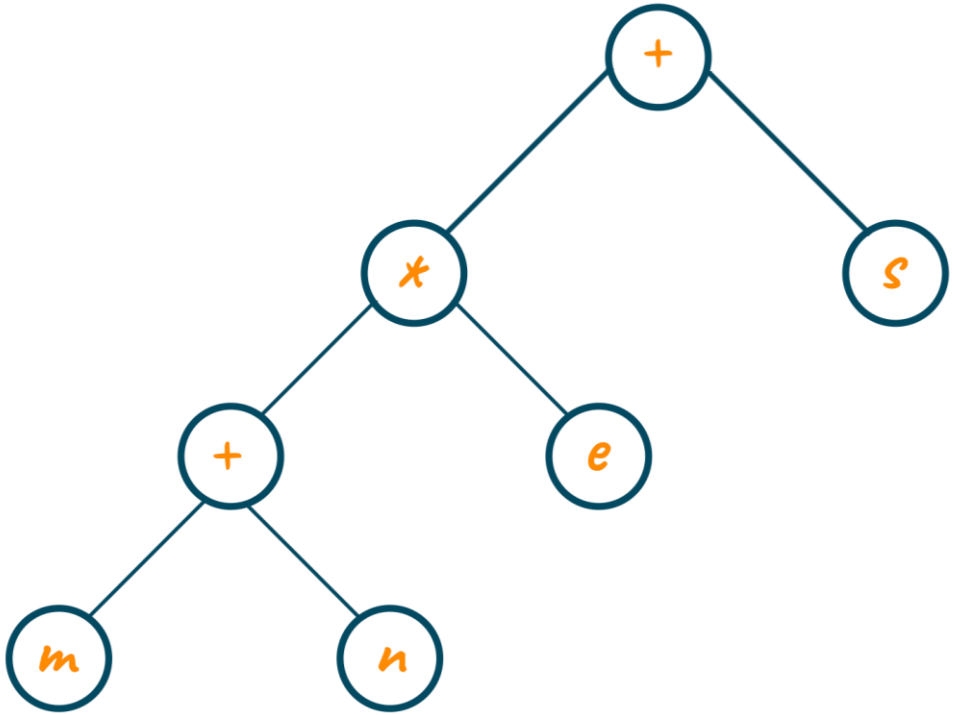
- Expression Trees
- Traversal in Trees

What is Expression Tree?

- An expression tree is one form of binary tree that is used to represent the expressions.
- A binary expression tree can represent two types of expressions i.e., algebraic expressions and Boolean expressions.
- It can represent the expressions containing both, unary and binary operators.
- Just like a binary tree, an expression tree has zero, one, or two nodes for each parent node.
- The leaves of the expression tree are operands such as variable name or constants and the other nodes represents the operator of the expression.
- You can evaluate the expression tree by applying the operator at the root node with the values obtained by recursively evaluating the left and right subtree

Expression tree

- The expression tree for the equation: (m+n)*e+s



How to Construct an Expression Tree?

- There are 2 ways to construct an Expression tree
  - Manual method
  - Using Stack
  - *Can be implemented using postfix expression as well (Best way)*

Manual method

(m+n)*e+s

- Steps

1) Look for the highest preference operator in the expression
2) Here it is ???
3) Repeat until all the expression is solved (one operation at a time)

Using Stack

(m+n)*e+s

- We loop through input expression and do the following for every character.
  1) If a character is an operand push that into the stack
  2) If a character is an operator pop two values from the stack, make them its child and push the current node again.
- In the end, the only element of the stack will be the root of an expression tree.

Using Postfix Expression

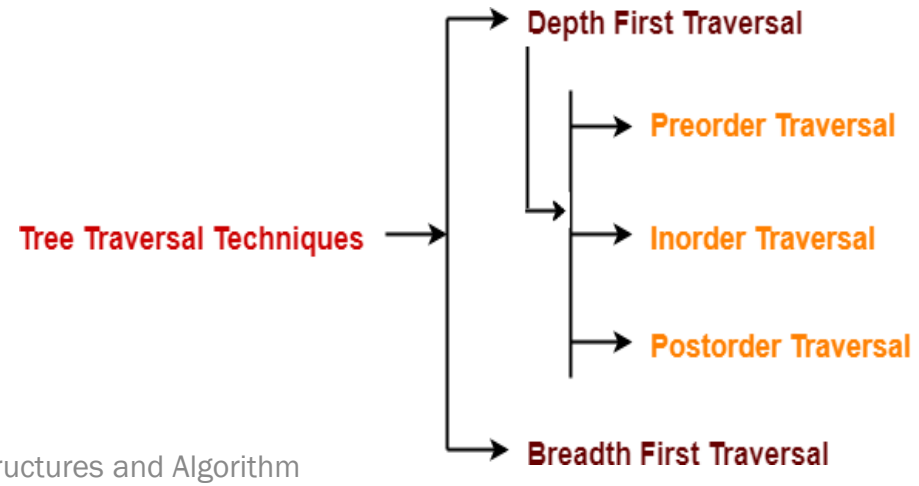
(m+n)*e+s

mn+e*s+

- While traversing through postfix expression, if the symbol encountered is an operand, then its pointer is pushed into the stack.
- At the same time, if the symbol encountered is an operator, then the pointers of two one-node trees T1 and T2 storing the operands of the expressions are popped from the stack.
- Later, the new tree is formed with the root as the operator and left and right subtree as childrens pointers to T2 and T1 respectively.
- At last, the pointer to this new tree is pushed in the stack.

Tree Traversals

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- Because, all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree.
  a) Pre-order Traversal
  b) In-order Traversal
  c) Post-order Traversal



Pre-order Traversal

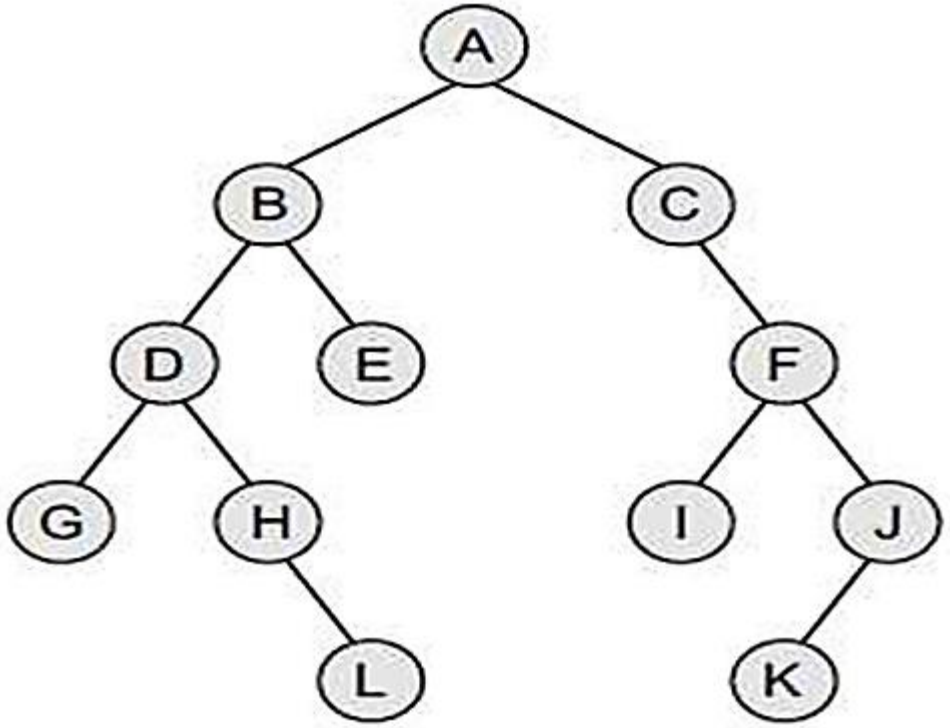
NLR

- To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.
    a) Visiting the root node,
    b) Traversing the left sub-tree, and finally
    c) Traversing the right sub-tree.

Algorithm for Pre-order

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER(TREE -> LEFT)
Step 4: PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

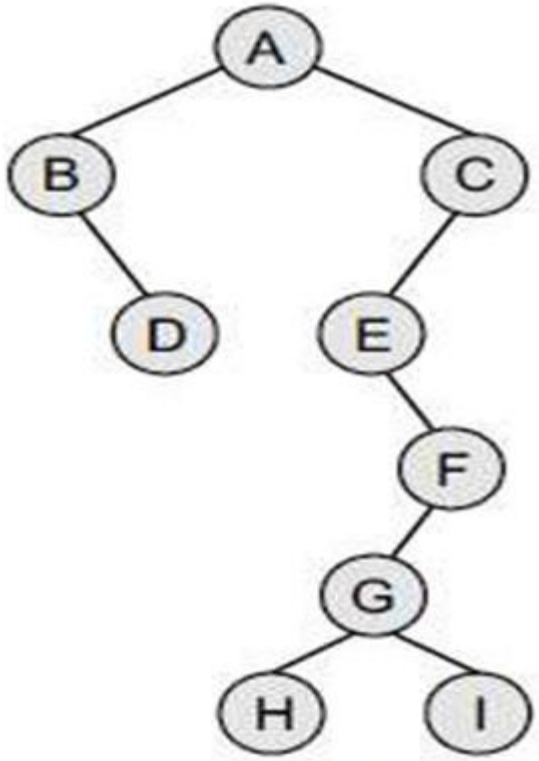
EXAMPLE PRE-ORDER



a) Visiting the root node
b) Traversing the left sub-tree, and finally
c) Traversing the right sub-tree

Traversal Order: A,B,D,G,H,L,E,C,F,I,J,K

EXAMPLE PRE-ORDER



a) Visiting the root node
b) Traversing the left sub-tree, and finally
c) Traversing the right sub-tree

Traversal Order: A,B,D,C,E,F,G,H,I

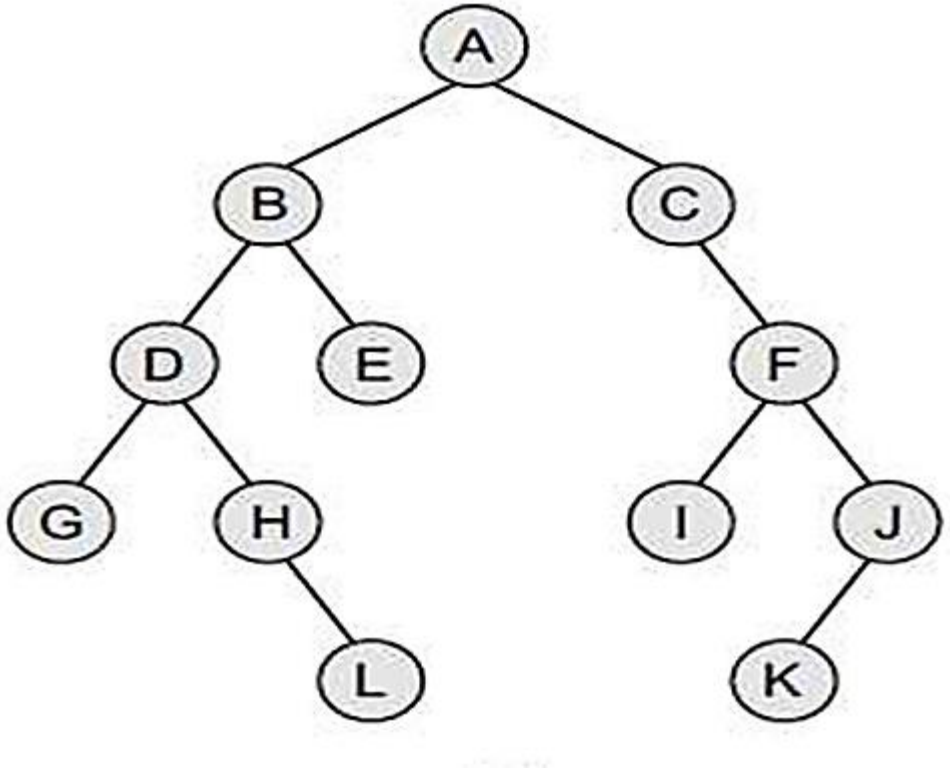
In-order Traversal

- To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.
    a) Traversing the left sub-tree,
    b) Visiting the root node, and finally
    c) Traversing the right sub-tree

Algorithm for Pre-order

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: INORDER(TREE -> LEFT)
Step 3: Write TREE -> DATA
Step 4: INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

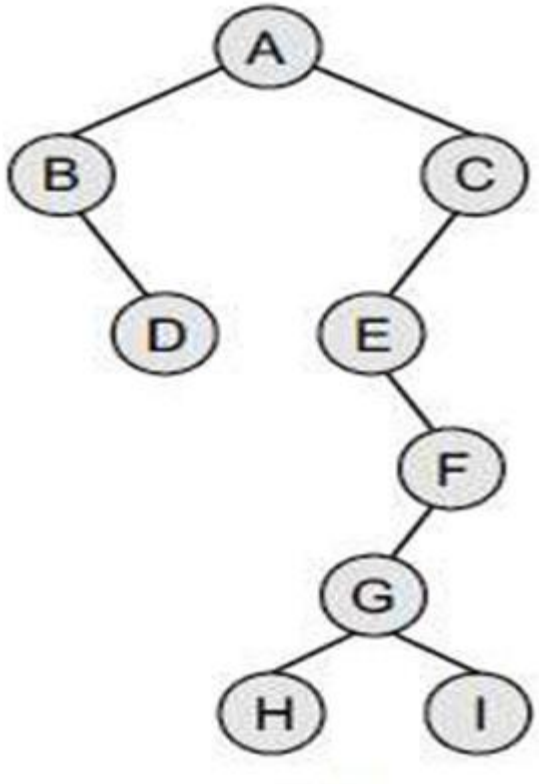
EXAMPLE In-ORDER



a) Traversing the left sub-tree,
b) Visiting the root node, and finally
c) Traversing the right sub-tree

Traversal Order: G,D,H,L,B,E,A,C,I,F,K,J

EXAMPLE In-ORDER



a) Traversing the left sub-tree,
b) Visiting the root node, and finally
c) Traversing the right sub-tree

Traversal Order: B,D,A,E,H,G,I,F,C

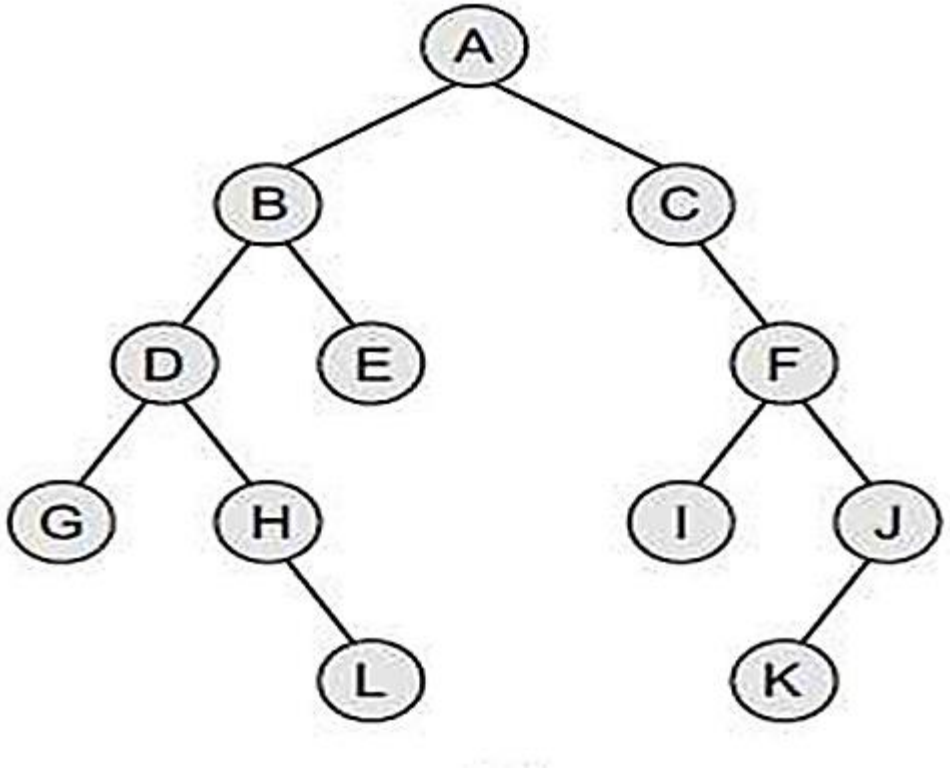
Post-order Traversal

- To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.
    a) Traversing the left sub-tree,
    b) Traversing the right sub-tree, and finally
    c) Visiting the root node.

Algorithm for Pre-order

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER(TREE -> LEFT)
Step 3: POSTORDER(TREE -> RIGHT)
Step 4: Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

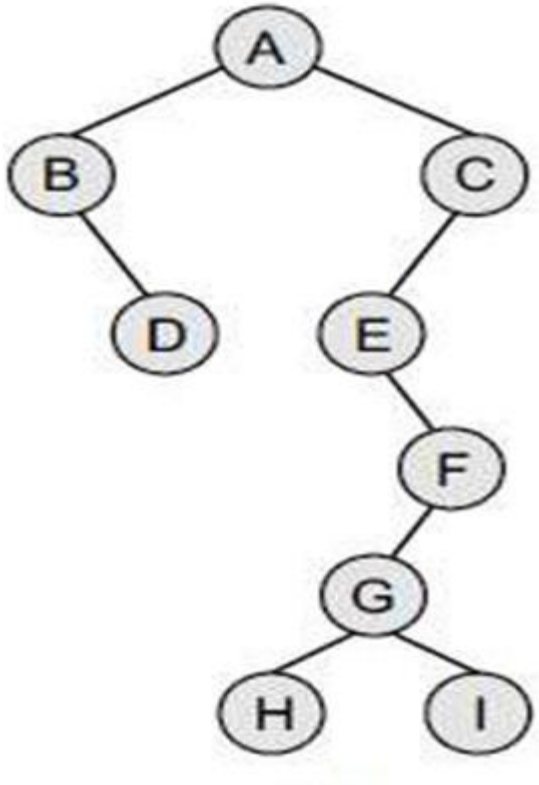
EXAMPLE Post-ORDER



a) Traversing the left sub-tree,
b) Traversing the right sub-tree, and finally
c) Visiting the root node.

Traversal Order: G,L,H,D,E,B,I,K,J,F,C,A

EXAMPLE Post-ORDER



a) Traversing the left sub-tree,
b) Traversing the right sub-tree, and finally
c) Visiting the root node.

Traversal Order: D,B,H,I,G,F,E,C,A

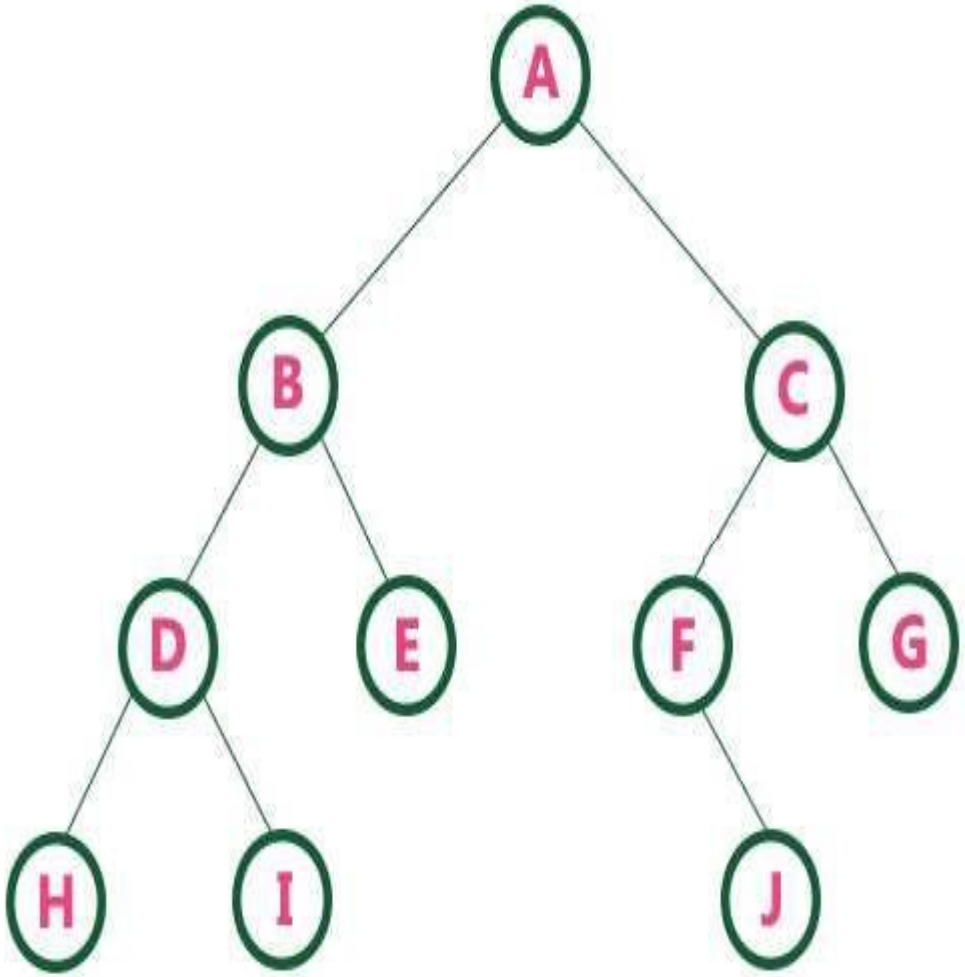
Construction of Binary tree with In-order & pre-order

In - Order :- LEFT NODE RIGHT

Pre - Order :- NODE LEFT RIGHT

Pre Order for the Tree :- A B D H I E C F J G

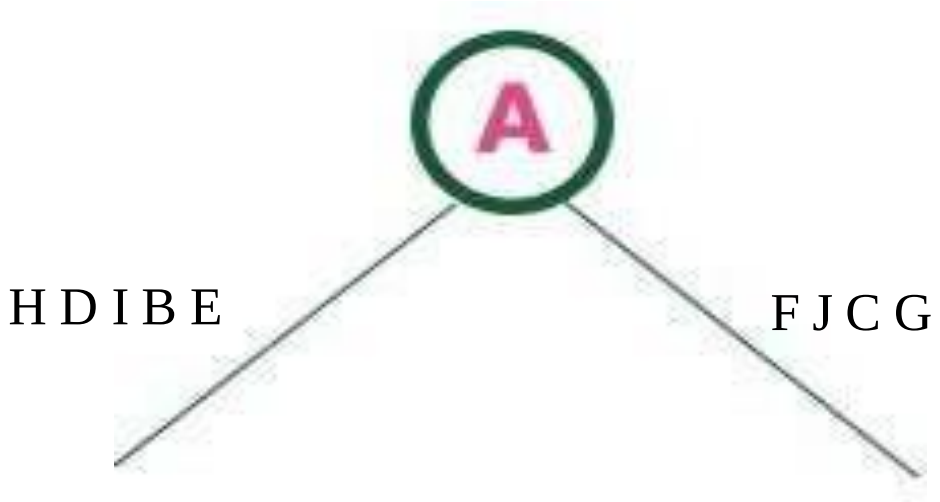
In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from PRE ORDER and LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- **A** B D H I E C F J G

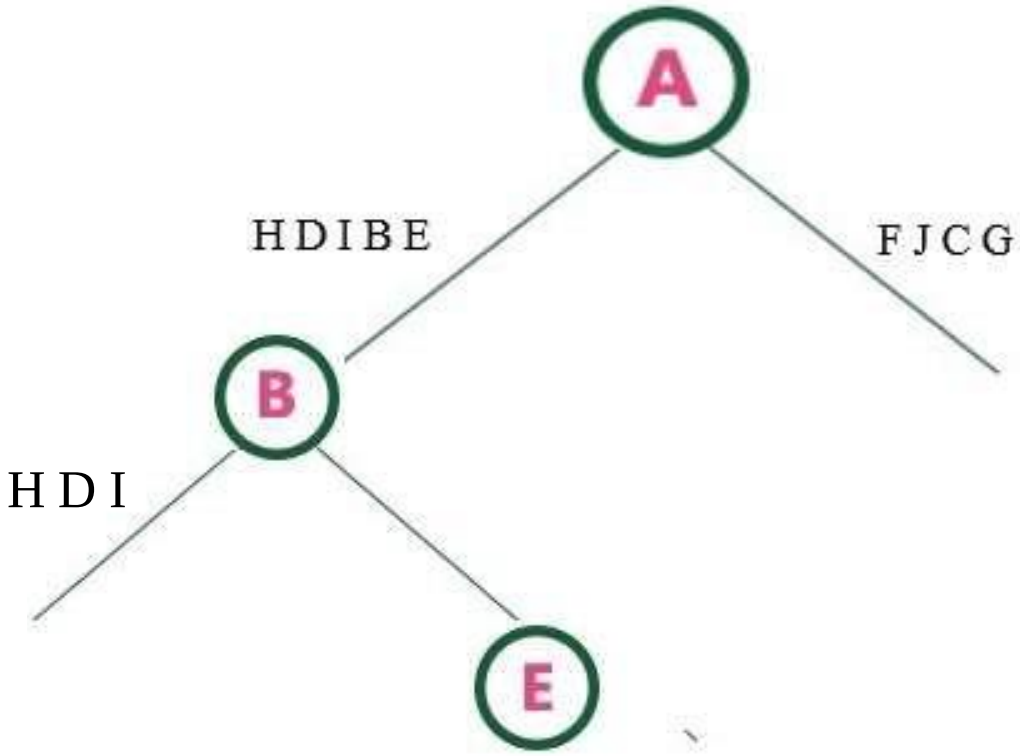
In Order for the Tree :- H D I B E **A** F J C G



Now finding the ROOT from PRE ORDER and LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- **A** **B** D H I E C F J G

In Order for the Tree :- H D I **B** E **A** F J C G

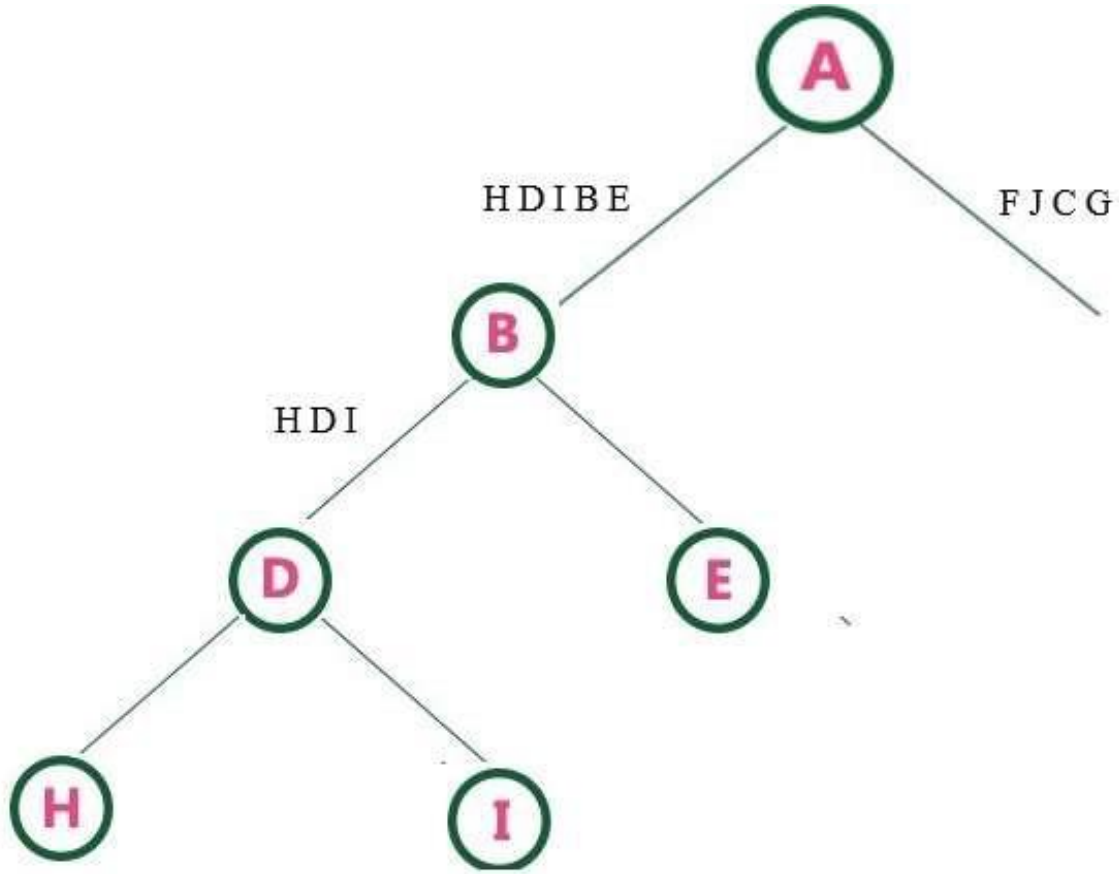


Now finding the ROOT from PRE ORDER and LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- **A B D** H I E C F J G

In Order for the Tree :- H **D** I **B** E **A** F J C G

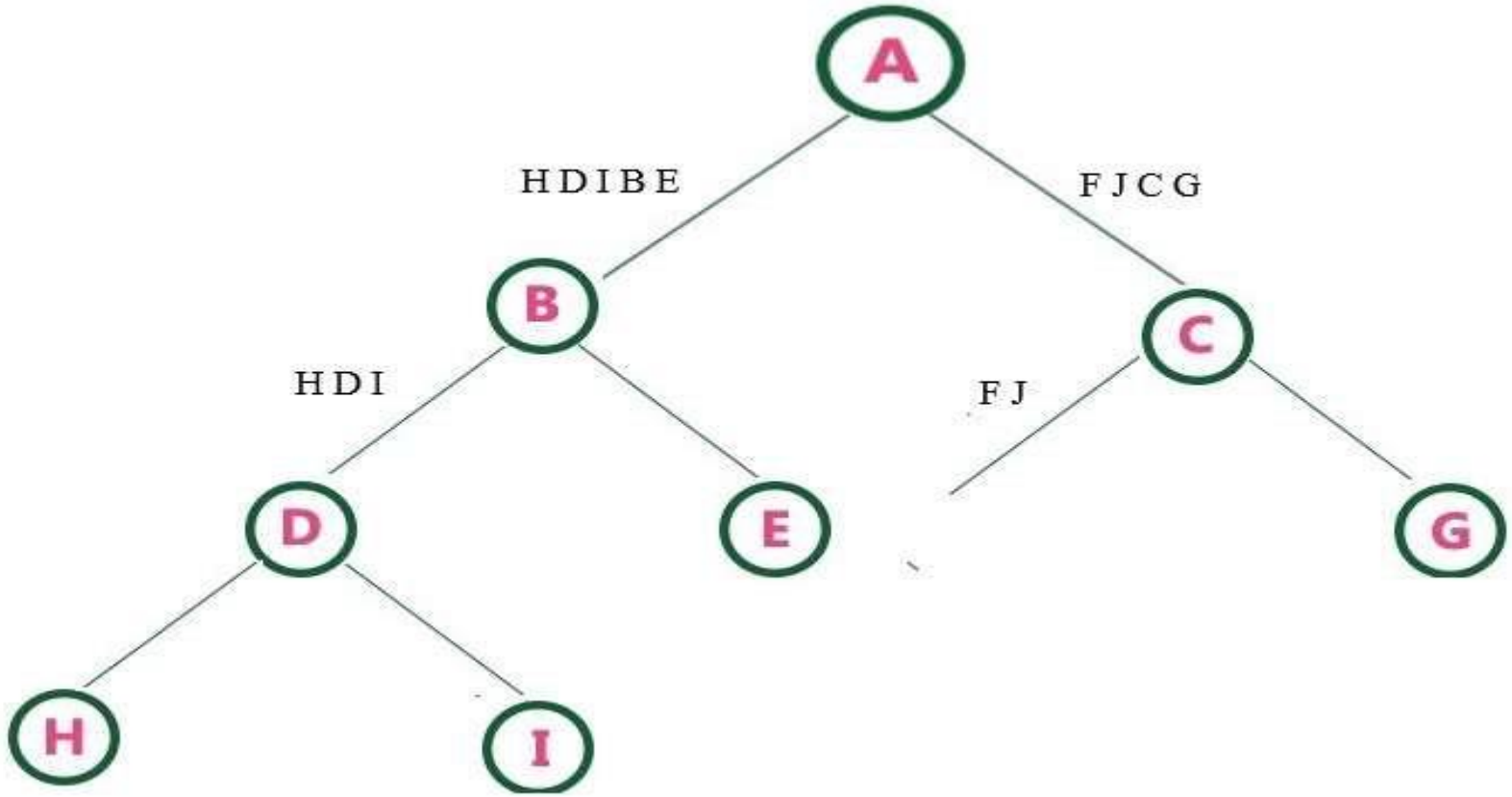
Now LEFT SUB TREE IS COMPLETED



Now finding the ROOT from PRE ORDER and LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- **A B D** H I E **C** F J G

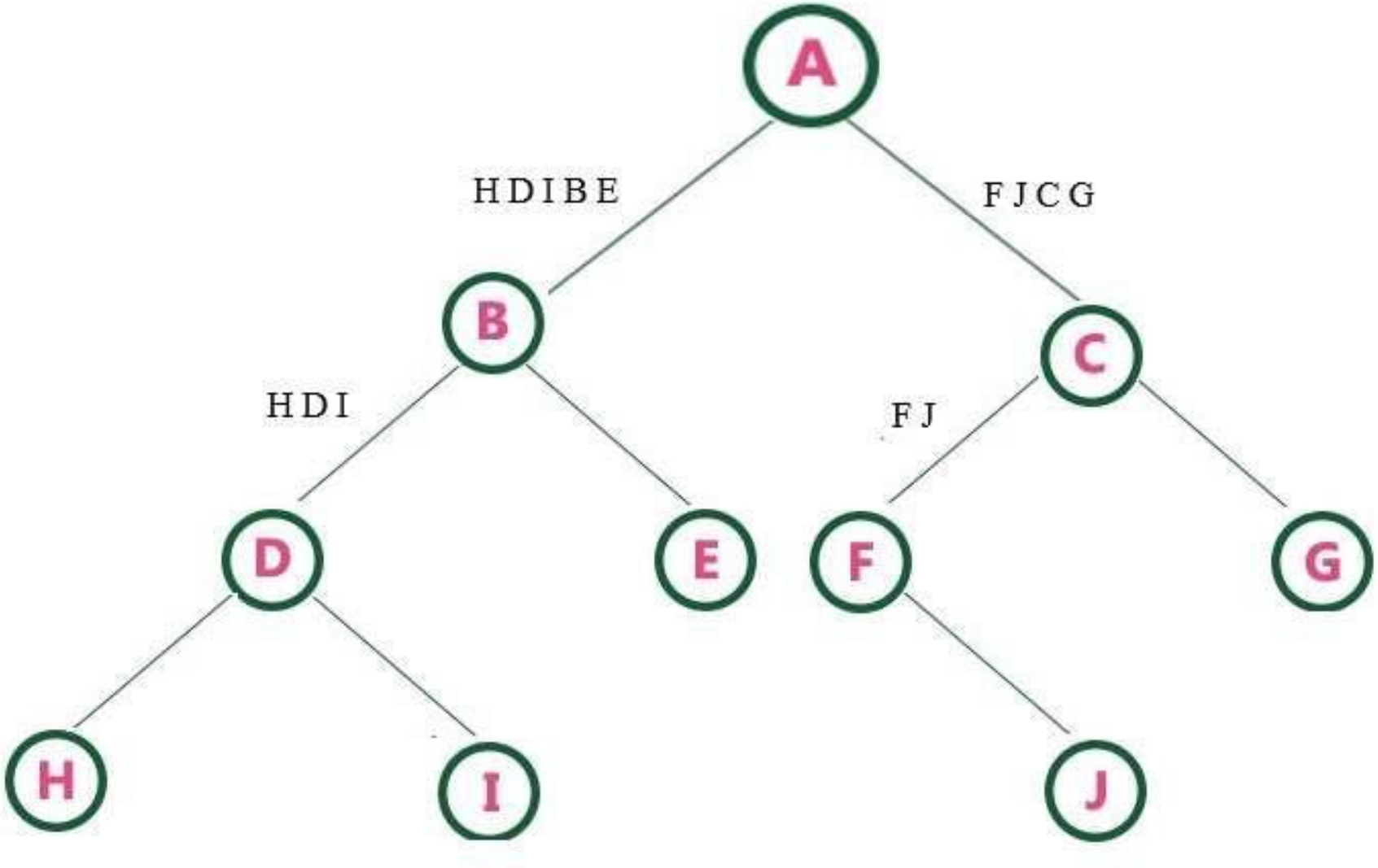
In Order for the Tree :- H **D** I **B** E **A** F J **C** G



Now finding the ROOT from PRE ORDER
and
LEFT & RIGHT CHILD'S from IN

Pre Order for the Tree :- **A B D** H I E **C F** J G

In Order for the Tree :- H **D** I **B** E **A F** J **C** G



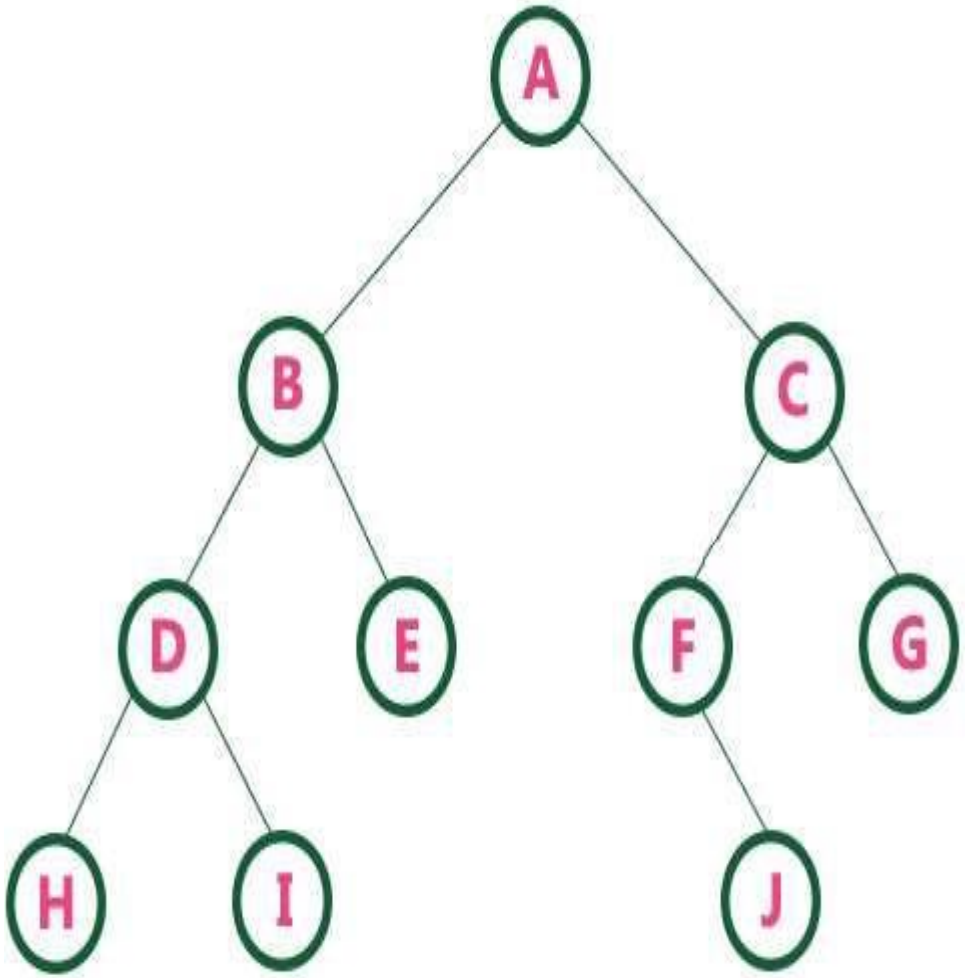
Construction of Binary tree with In-order & Post-order

In - Order :- LEFT NODE RIGHT

Post - Order :- LEFT RIGHT NODE

Post Order for the Tree :- H I D E B J F G C A

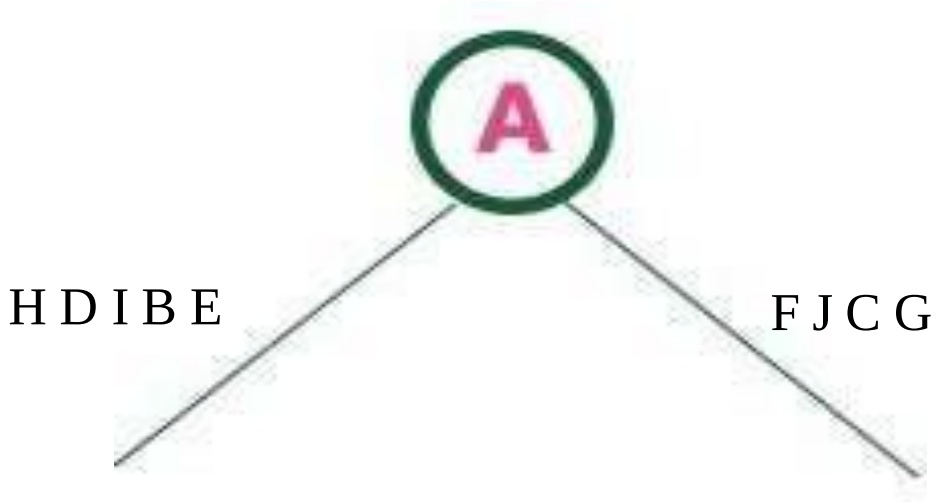
In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from POST ORDER and LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E B J F G C **A**

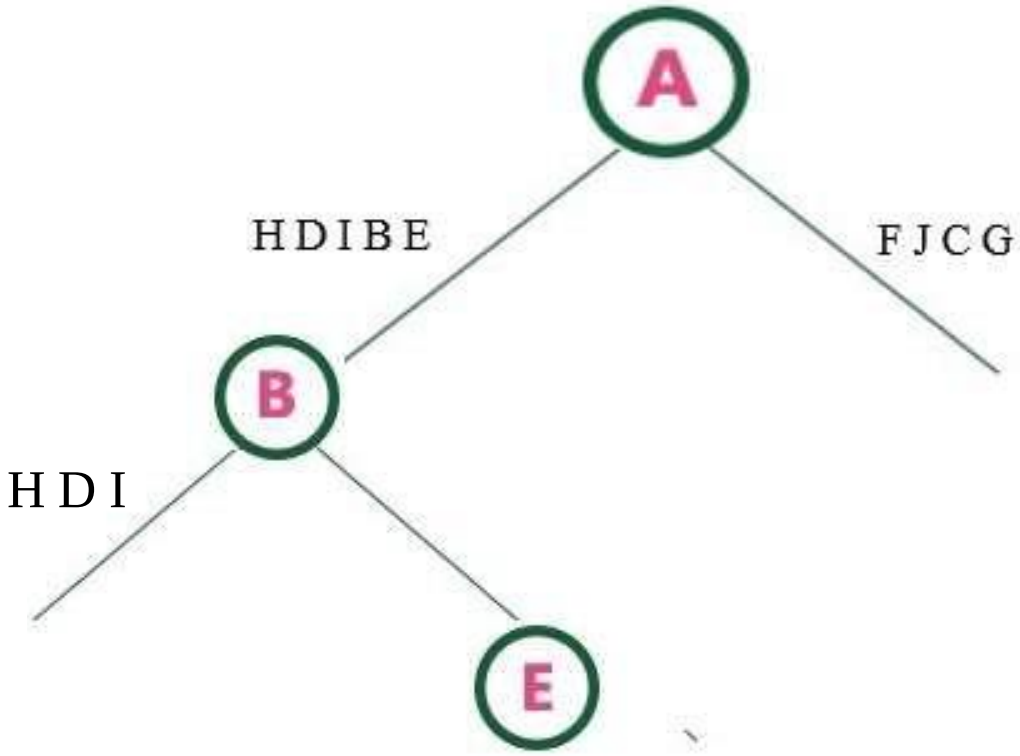
In Order for the Tree :- H D I B E **A** F J C G



Now finding the ROOT from POST ORDER and LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E **B** J F G C A

In Order for the Tree :- H D I **B** E A F J C G

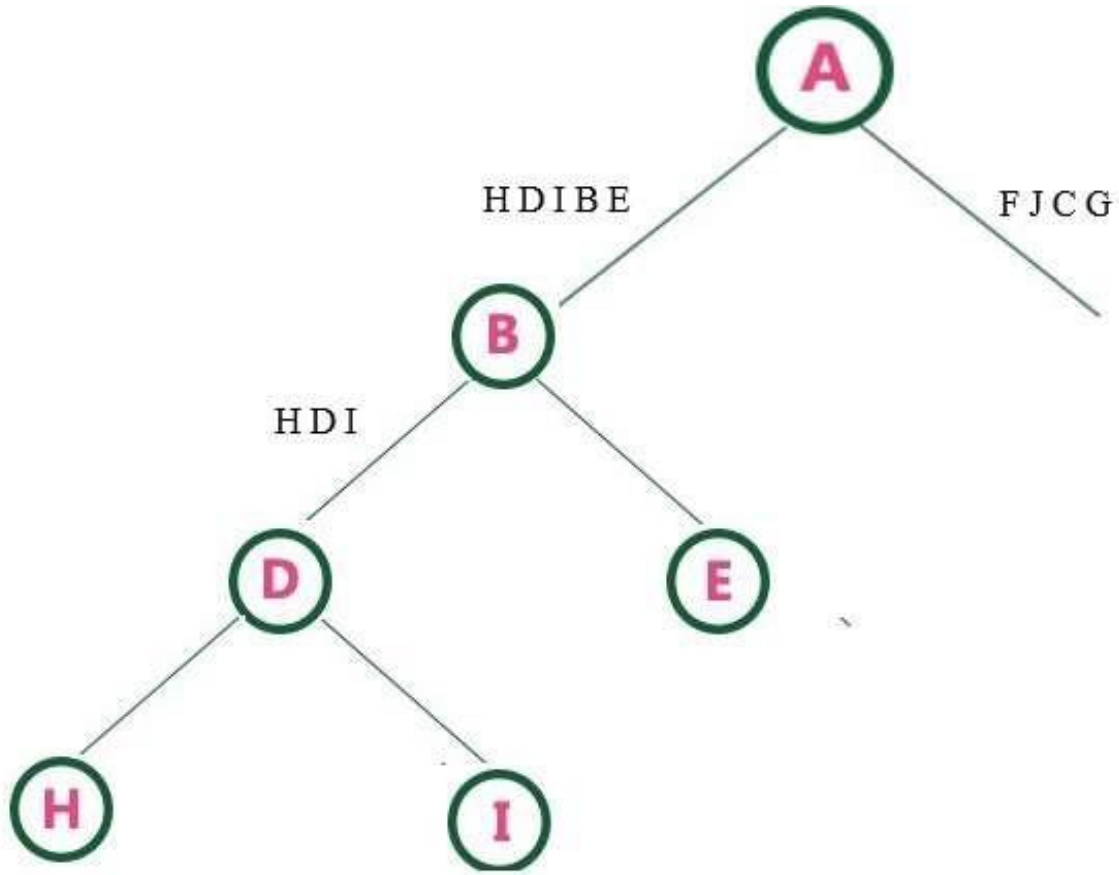


Now finding the ROOT from POST ORDER and LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I **D** E B J F G C A

In Order for the Tree :- H **D** I **B** E **A** F J C G

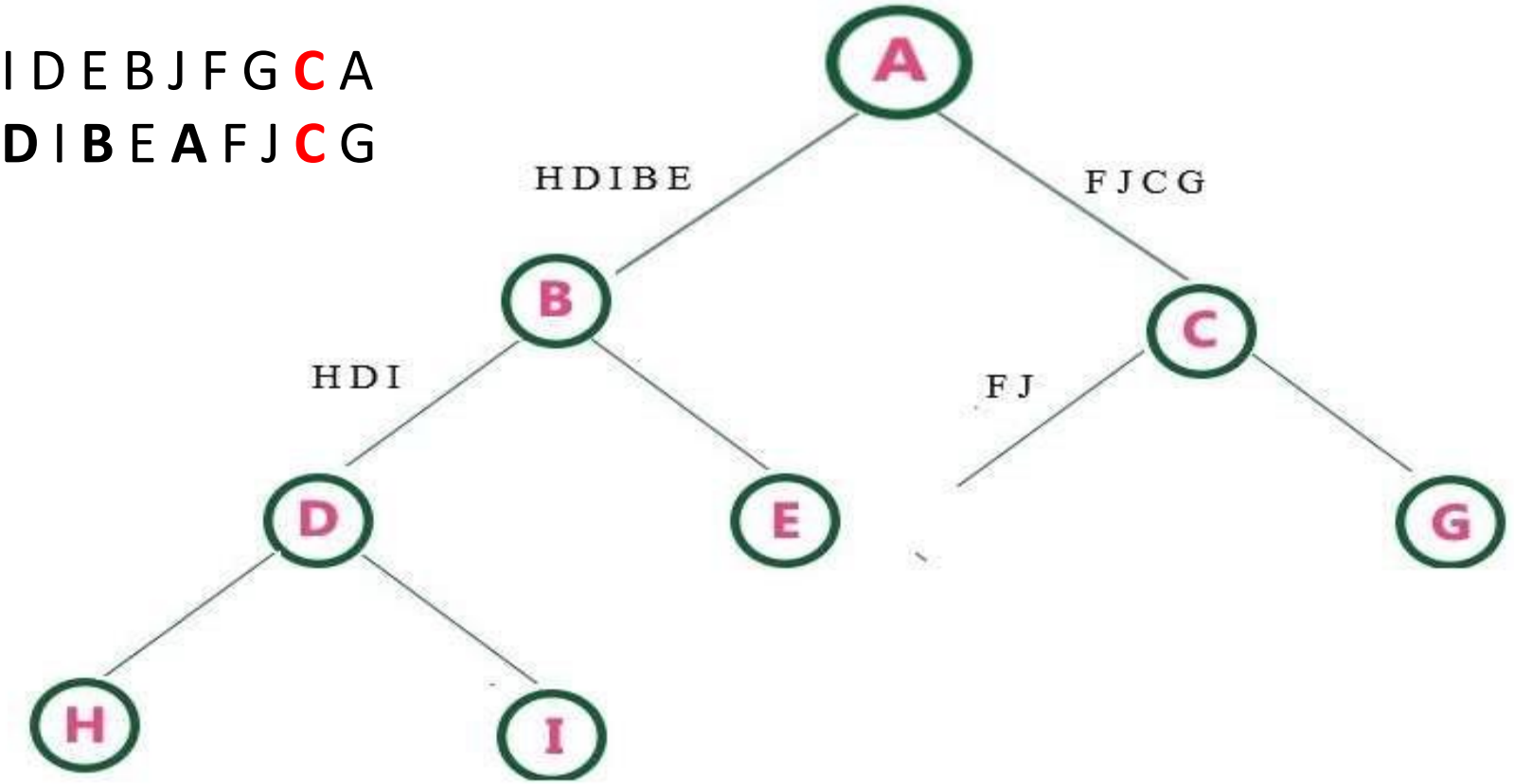
Now LEFT SUB TREE IS COMPLETED



Now finding the ROOT from POST ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E B J F G **C** A

In Order for the Tree :- H **D** I **B** E **A** F J **C** G



Now finding the ROOT from POST ORDER and LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E B J **F** G C A

In Order for the Tree :- H **D** I **B** E **A** **F** J **C** G