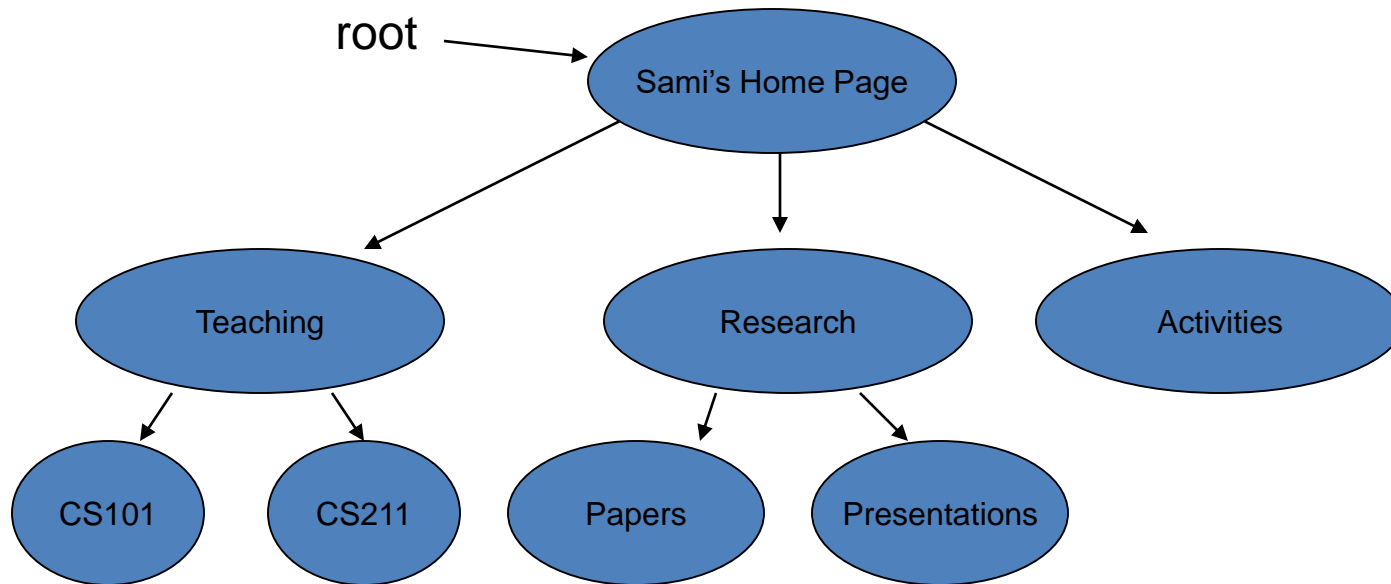


Trees

Example Tree



Definitions

- ✓ **root** - starting point (top) of the tree
- ✓ **parent** (ancestor) - the vertex “above” this vertex
- ✓ **child** (descendent) - the vertices “below” this vertex

Definitions

- any two vertices must have one and only one path between them else its not a tree
- a tree with **N nodes** has **N-1 edges**

Definitions

- ✓ **leaves** (terminal nodes) - have no children
- ✓ **level** - the number of edges between this node and the root
- ✓ **ordered tree** - where children's order is significant

Definitions

- ✓ **Depth of a node** - the length of the path from the root to that node
 - root: depth 0
- ✓ **Height of a node** - the length of the longest path from that node to a leaf
 - any leaf: height 0
- ✓ **Height of a tree**: The length of the longest path from the root to a leaf

Balanced Trees

- the **difference** between the height of the left sub-tree and the height of the right sub-tree is **not more than 1**.

Trees - Example

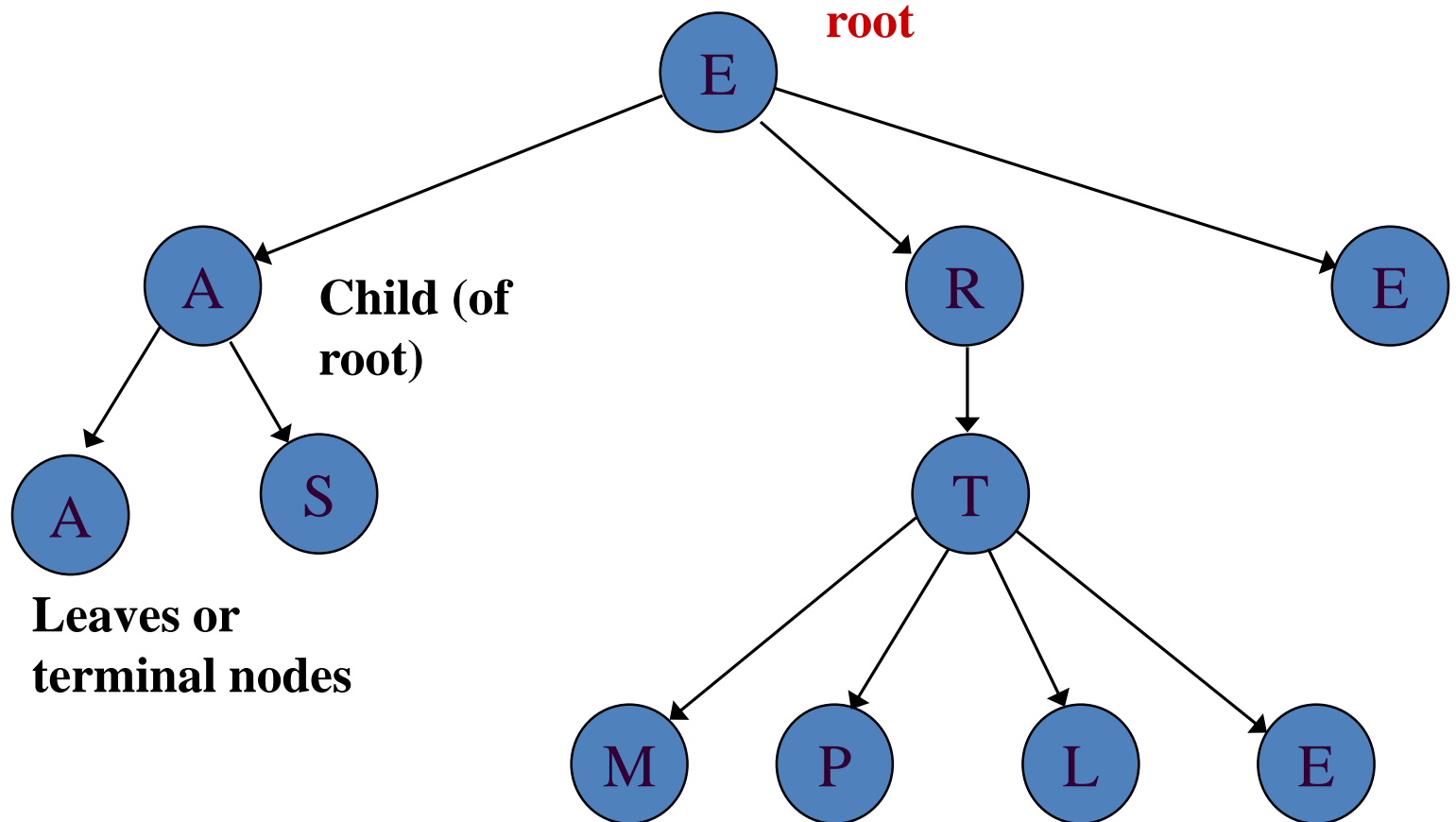
Level

0

1

2

3



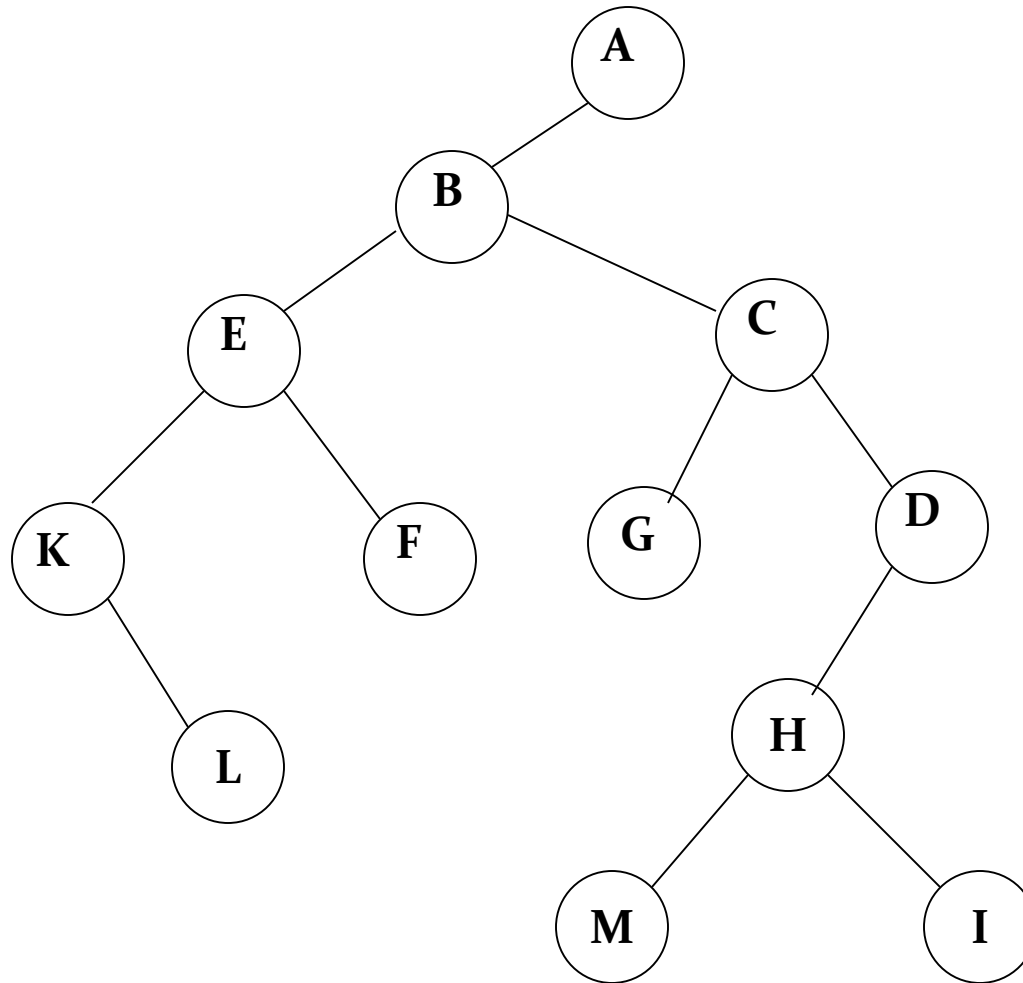
Depth of T: 2

Height of T: 1

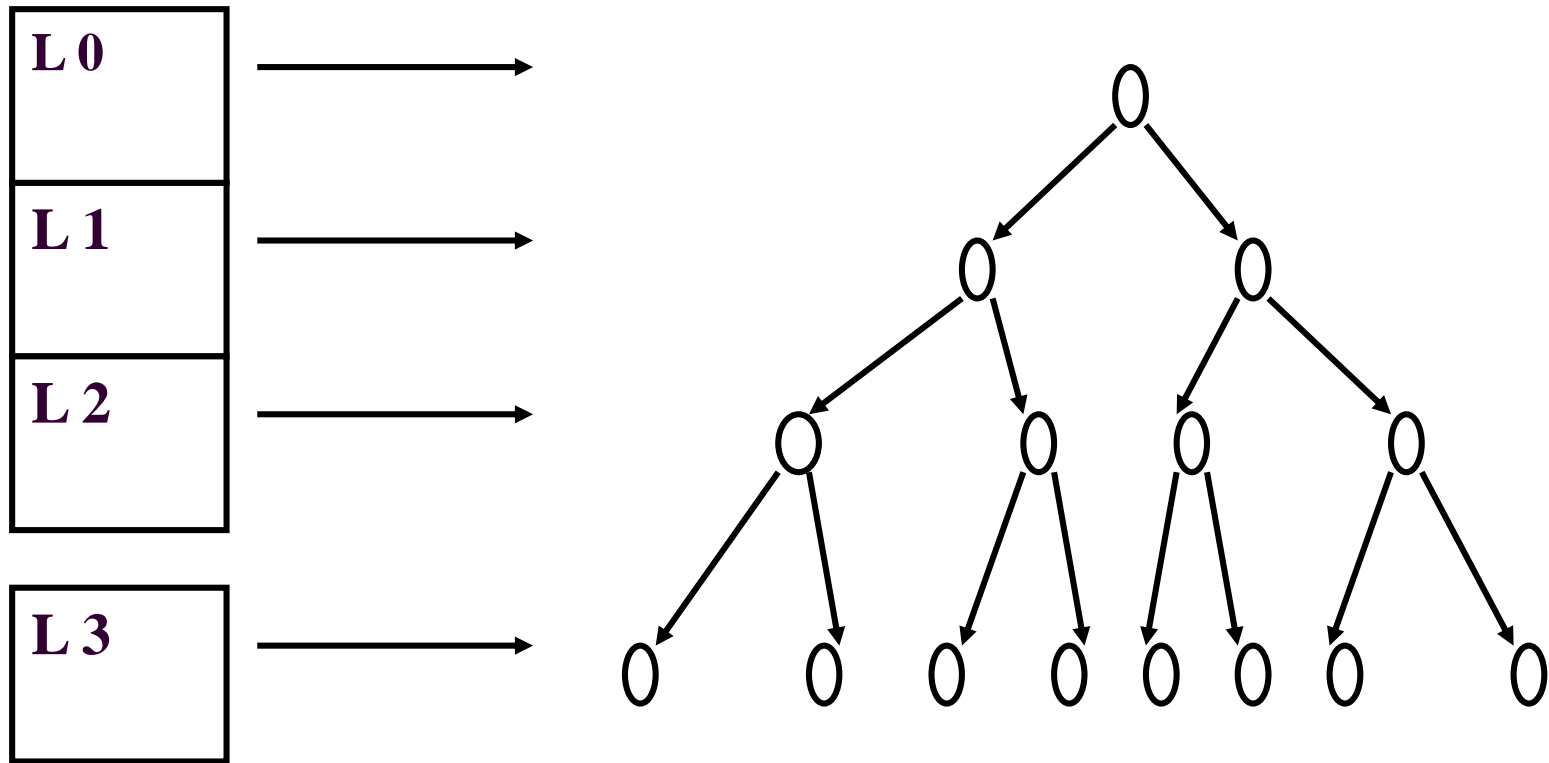
Binary Trees

- ✚ A special class of trees: max degree for each node is 2
- *Recursive definition:* A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.

Example



Height of a Complete Binary Tree



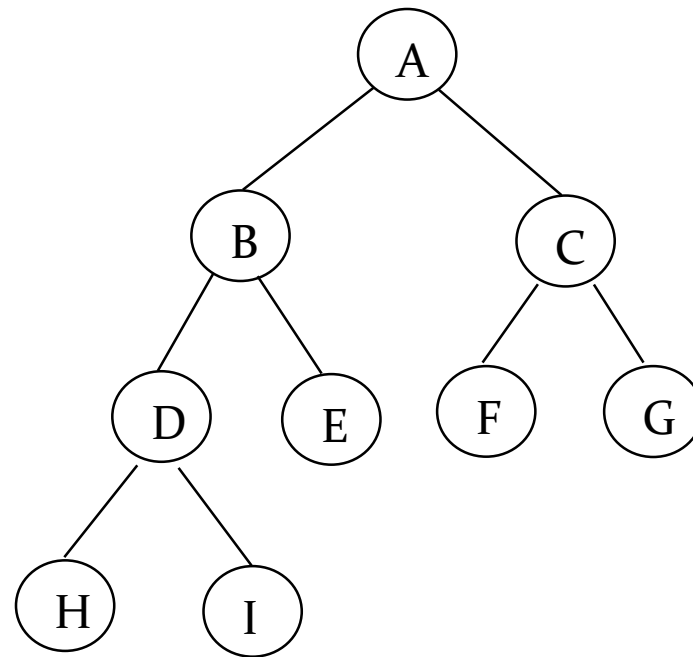
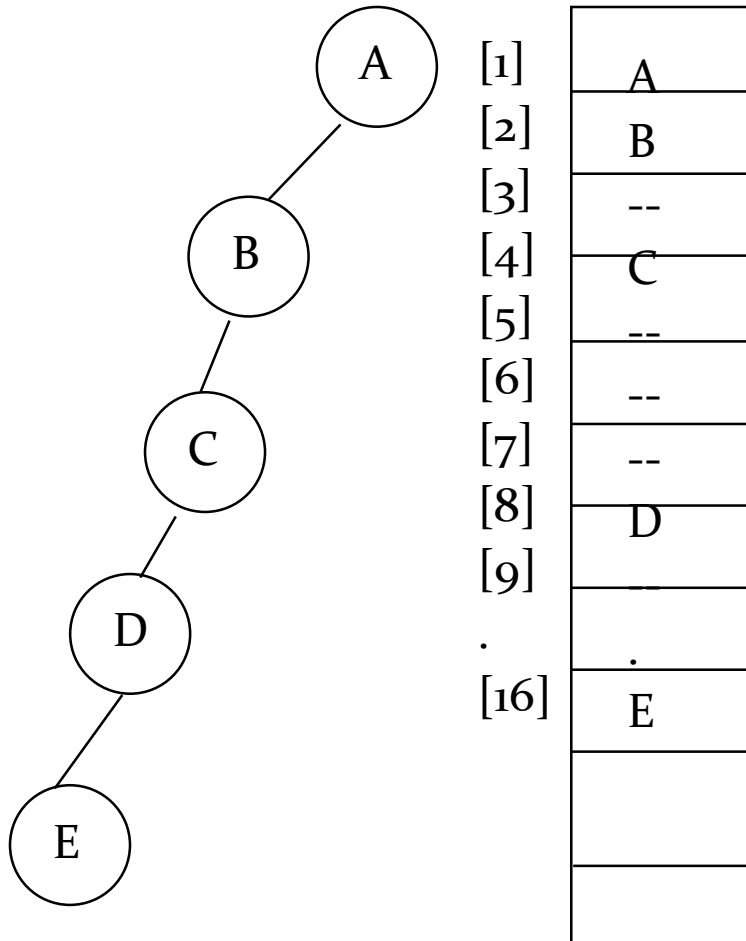
At each level the number of the nodes is **doubled**.

total number of nodes: $1 + 2 + 2^2 + 2^3 = 2^4 - 1 = 15$

Binary Tree Representations

- ✿ If a complete binary tree with n nodes is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have:
 - ✚ $parent(i)$ is at $i/2$ if $i \neq 1$. If $i=1$, i is at the root and has no parent.
 - ✿ $leftChild(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
 - ✿ $rightChild(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child.

Sequential / Implicit Array Representation

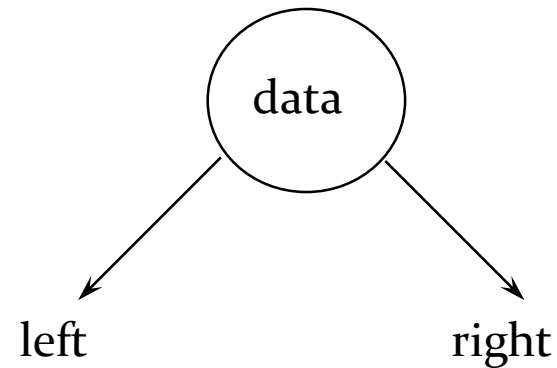
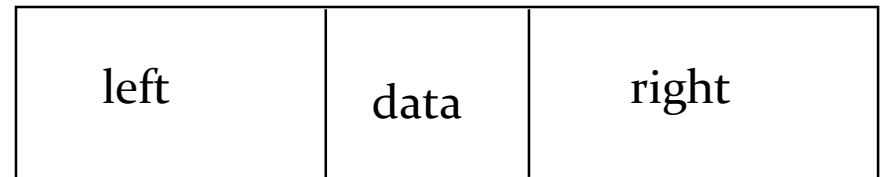


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

(1) waste space
(2) insertion/deletion problem

Dynamic/ Linked Representation

```
typedef struct node
{
  Int data;
  Struct node *left;
  Struct node *right;
} bin;
```



Binary Tree Operation

- Creation of binary tree
- Insertion of a node in the tree
- Searching
- Deleting a node

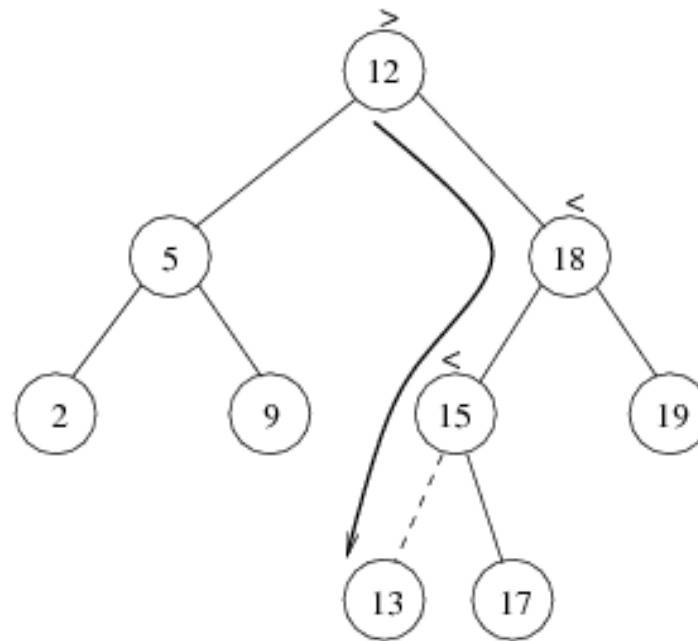
Binary Tree Operations (create)

- The basis of our binary tree node is the following struct declaration:

```
struct TreeNode
{
    int value;
    TreeNode *left;
    TreeNode *right;
};
```


Binary Tree Operations (Insert)

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



Binary Search Tree (BST)

- A binary tree is useful data structure when two way decisions must be made at each point in a process.
- Suppose, we want to find duplicates in a list of numbers.
 - One way is to go on comparing each number with all those that precede it. Number of comparisons increases.
- Comparison can be reduced by using BST.

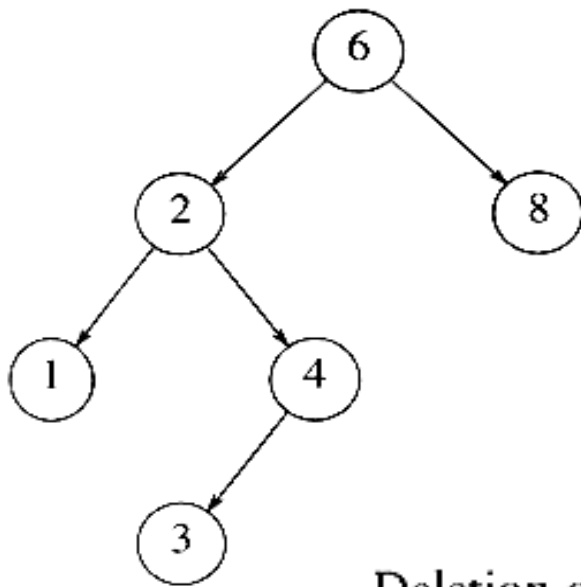
Binary Tree Operations

(Delete a node)

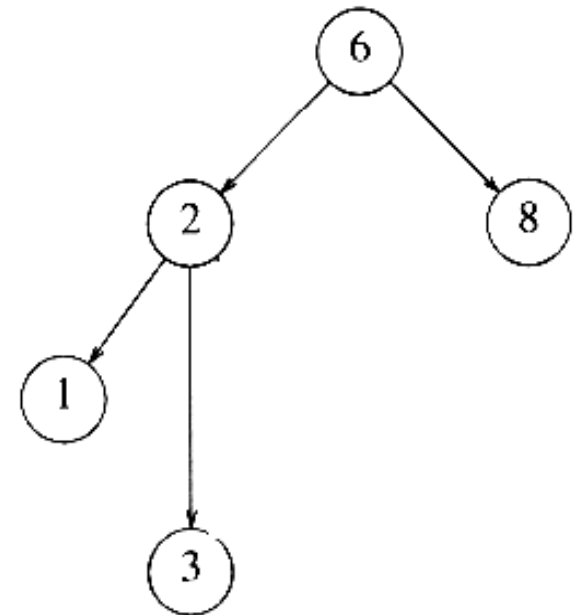
- When we delete a node, we need to consider how we take care of the children of the deleted node.
- This has to be done such that the property of the **binary search tree** is maintained.
- Three cases:
 - (1) the node is a leaf
 - » Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node

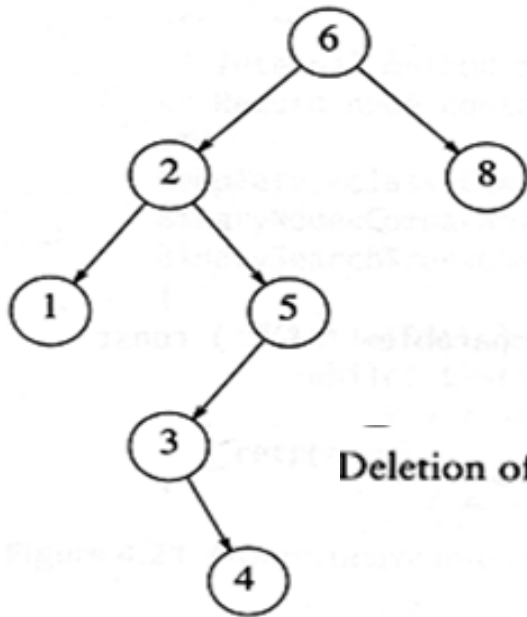


Deletion of a node (4) with one child,

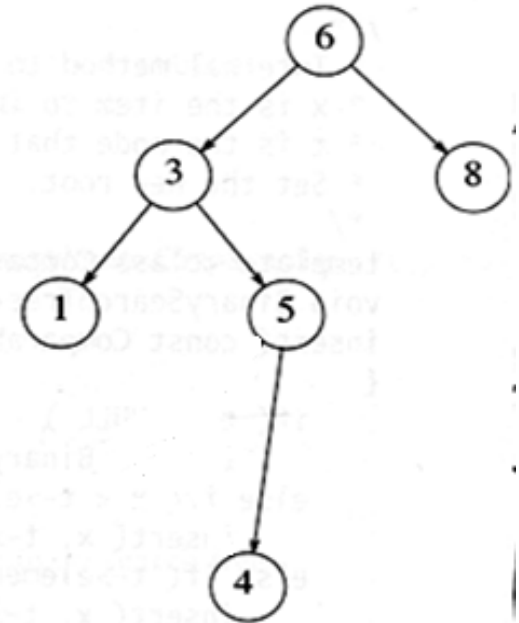


(3) the node has 2 children

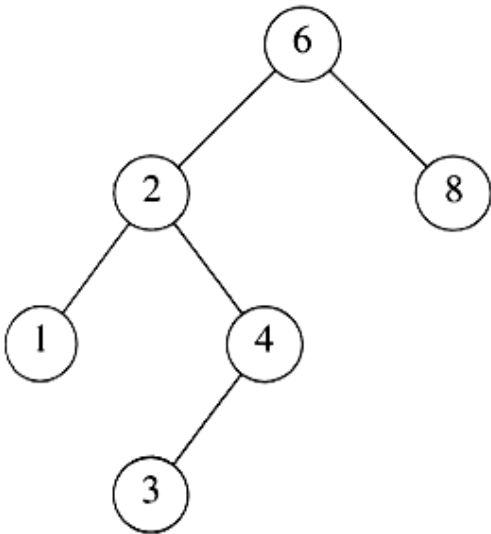
- replace the key of that node with the minimum element at the right subtree
- delete the minimum element
 - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.



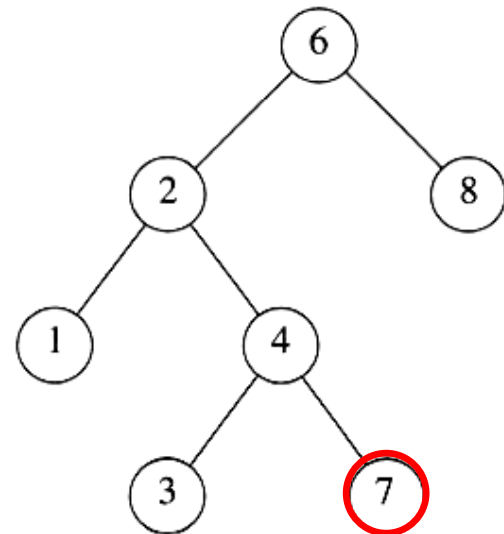
Deletion of a node (2) with two children



Binary Search Trees



A binary search tree



Not a binary search tree

Binary Tree Traversals

- ✚ Let L, V, and R stand for moving left, visiting the node, and moving right.
- ✚ There are six possible combinations of traversal
 - ✚ lVr, lrV, Vlr, Vrl, rVl, rlV
- ✚ Adopt convention that we traverse left before right, only 3 traversals remain
 - ✚ lVr, lrV, Vlr
 - ✚ inorder, postorder, preorder

Binary Tree - Traversal

There are six possible orderings of traversal.

By convention, we always traverse the left subtree before the right one.

That reduces the choices to three:

- **Preorder** - Visit the root node first
- **Inorder** - Visit the left subtree, then the root
- **Postorder** - Visit the root node last

Preorder Traversal

- Visit the root of the tree first,
then visit the nodes in the left subtree,
then visit the nodes in the right subtree

Preorder(tree)

If tree is not NULL

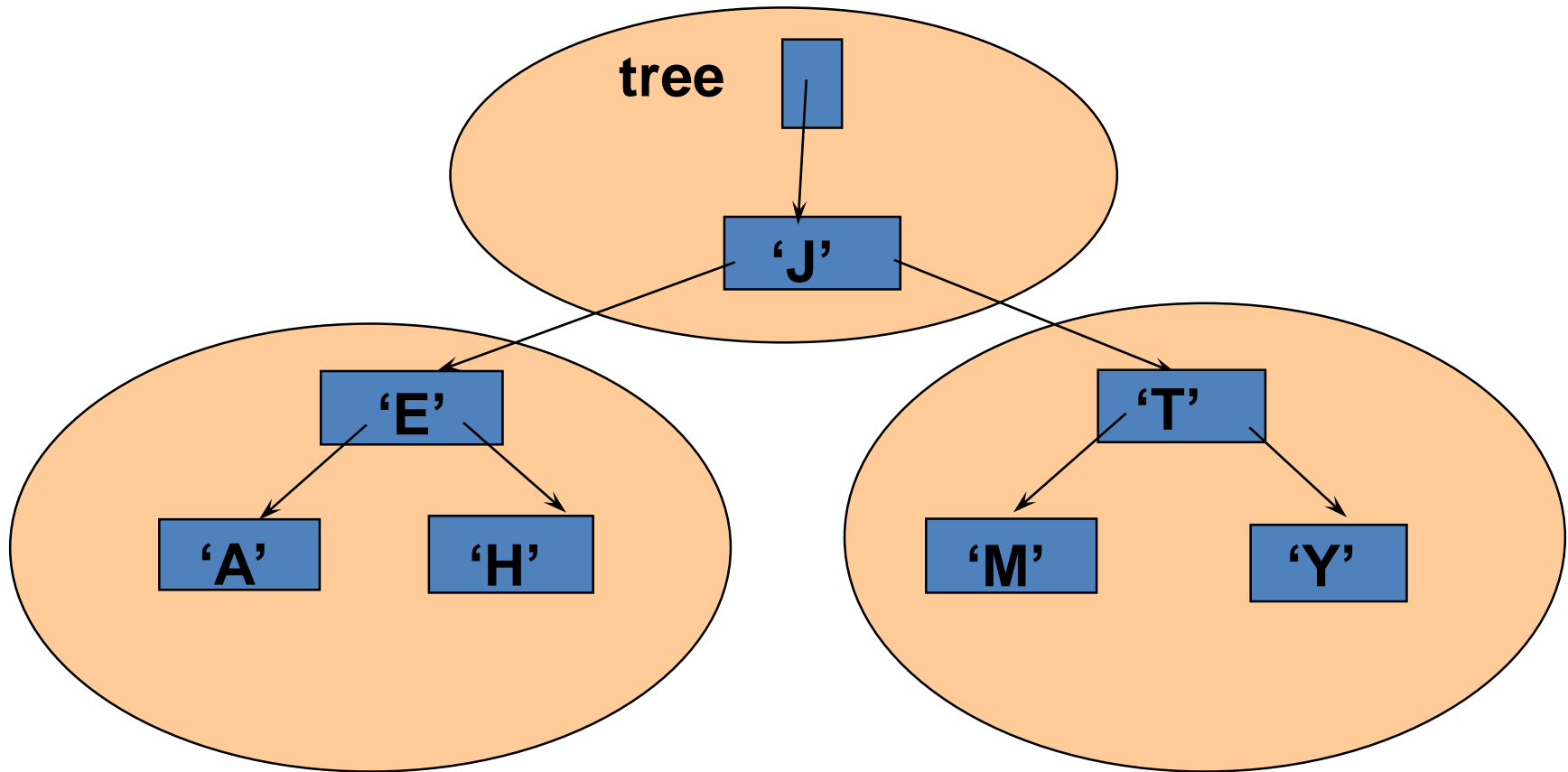
Visit Info(tree)

Preorder(Left(tree))

Preorder(Right(tree))

(Warning: "visit" means that the algorithm does something with the values in the node, e.g., print the value)

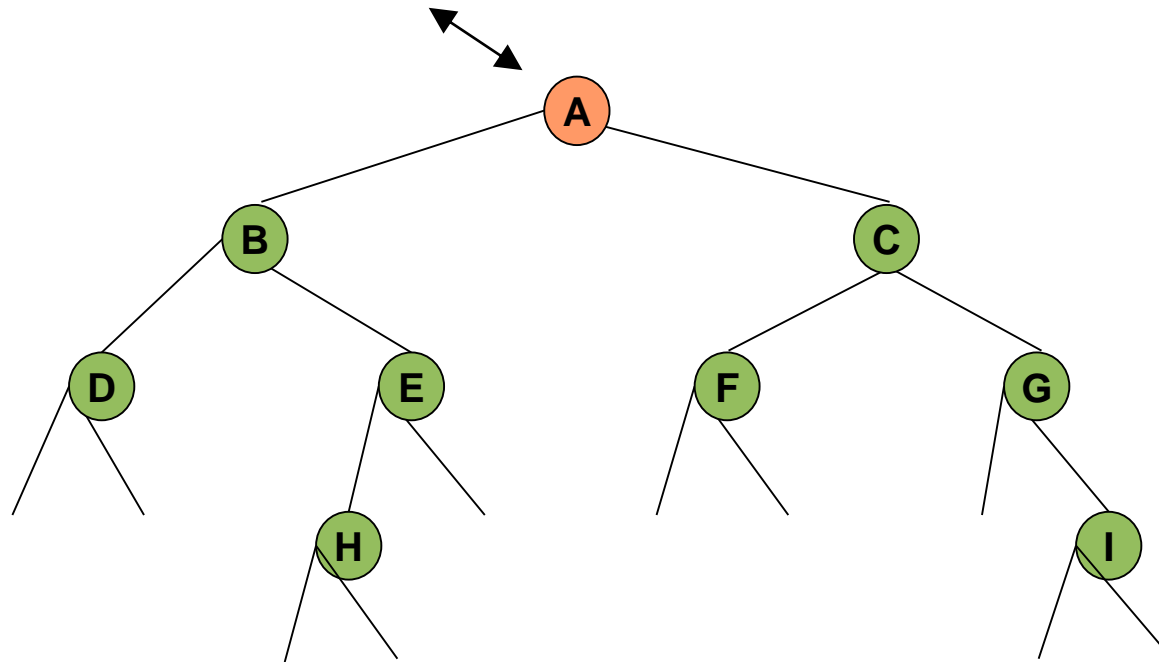
Preorder Traversal:



Visit left subtree second

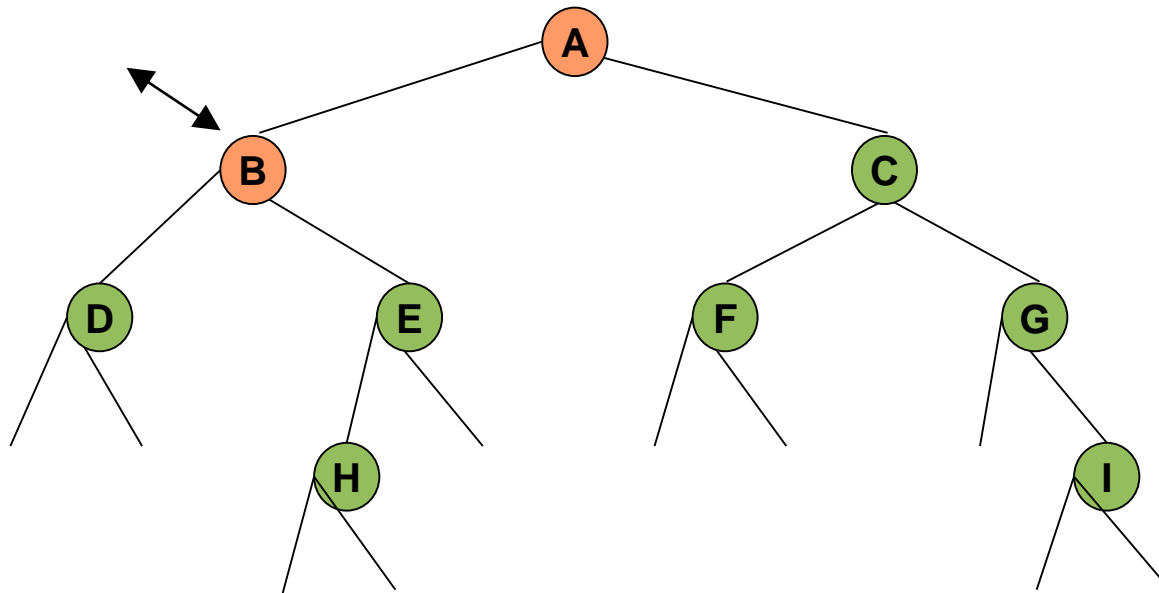
Visit right subtree last

Traversing a Tree Preorder



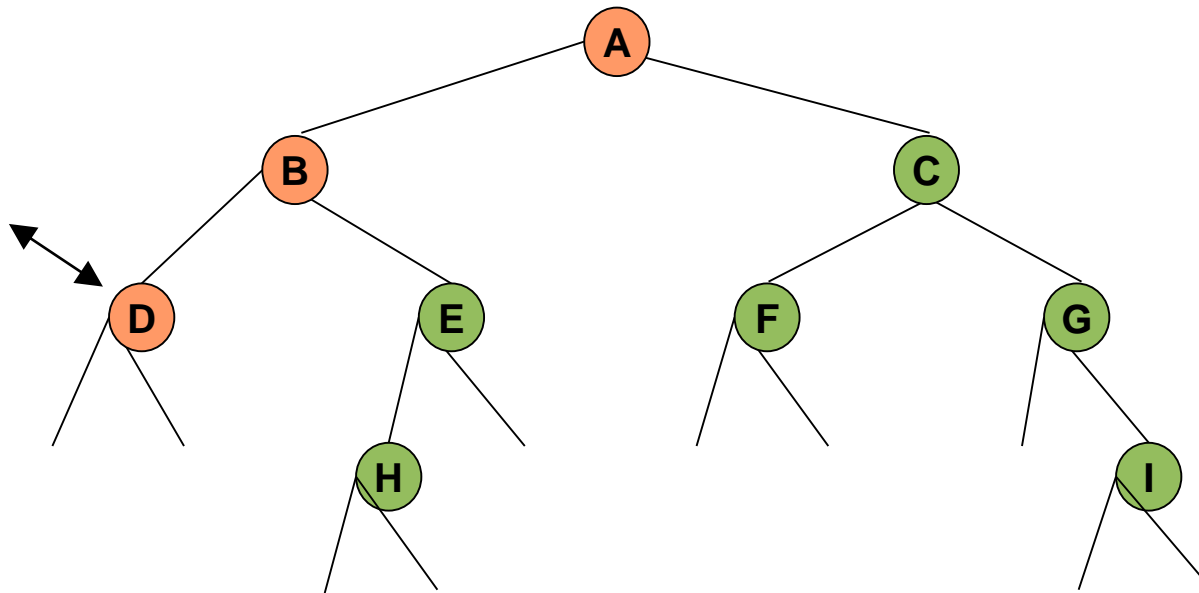
Result: A

Traversing a Tree Preorder



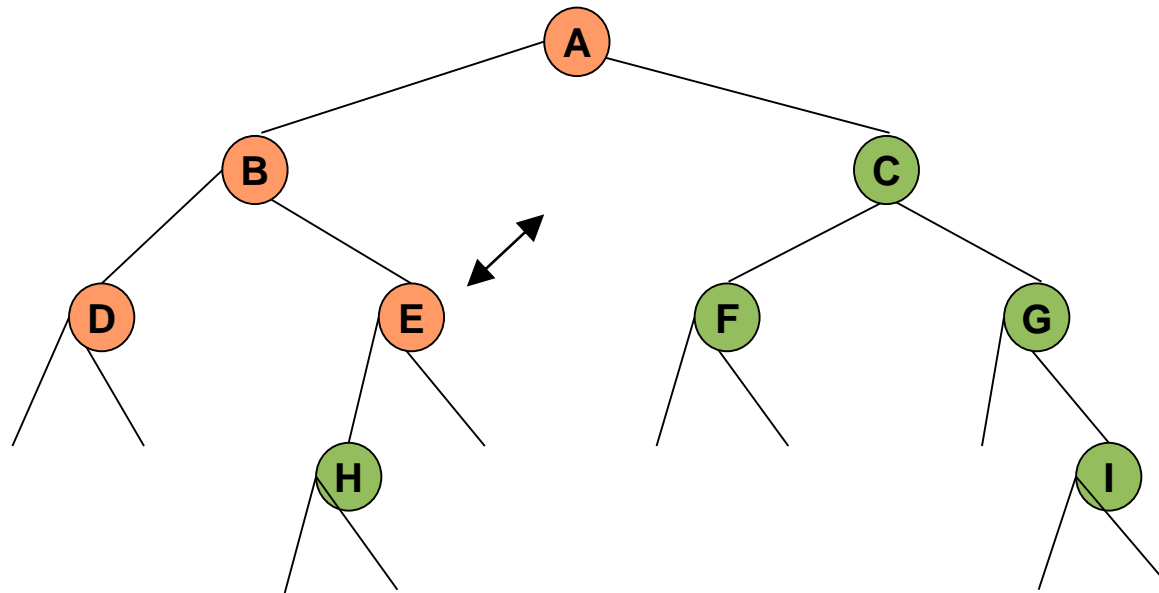
Result: AB

Traversing a Tree Preorder



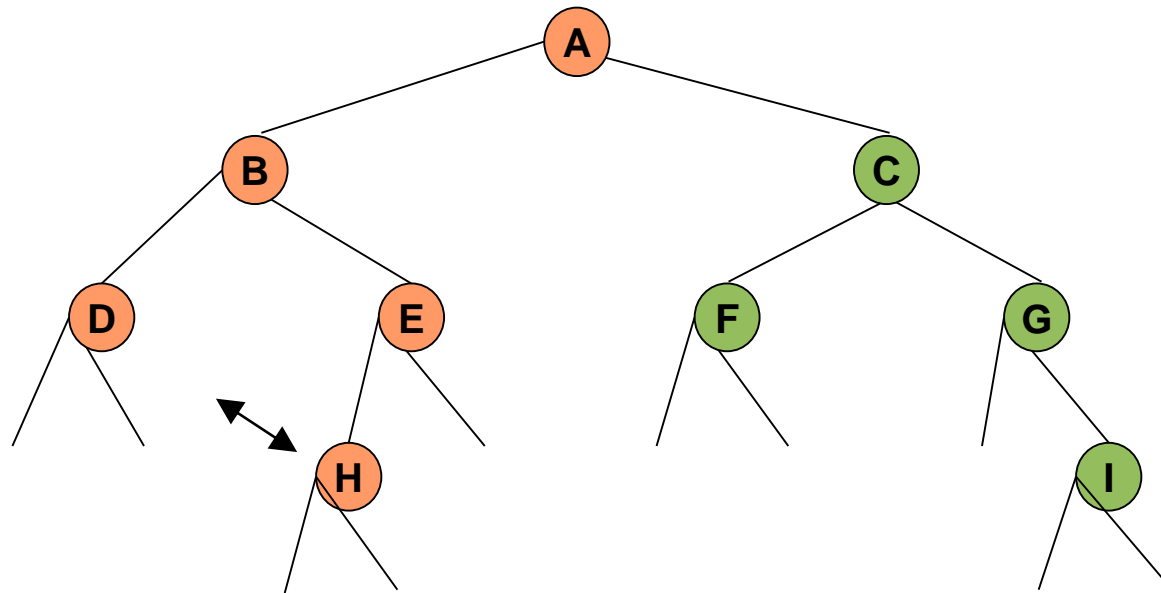
Result: ABD

Traversing a Tree Preorder



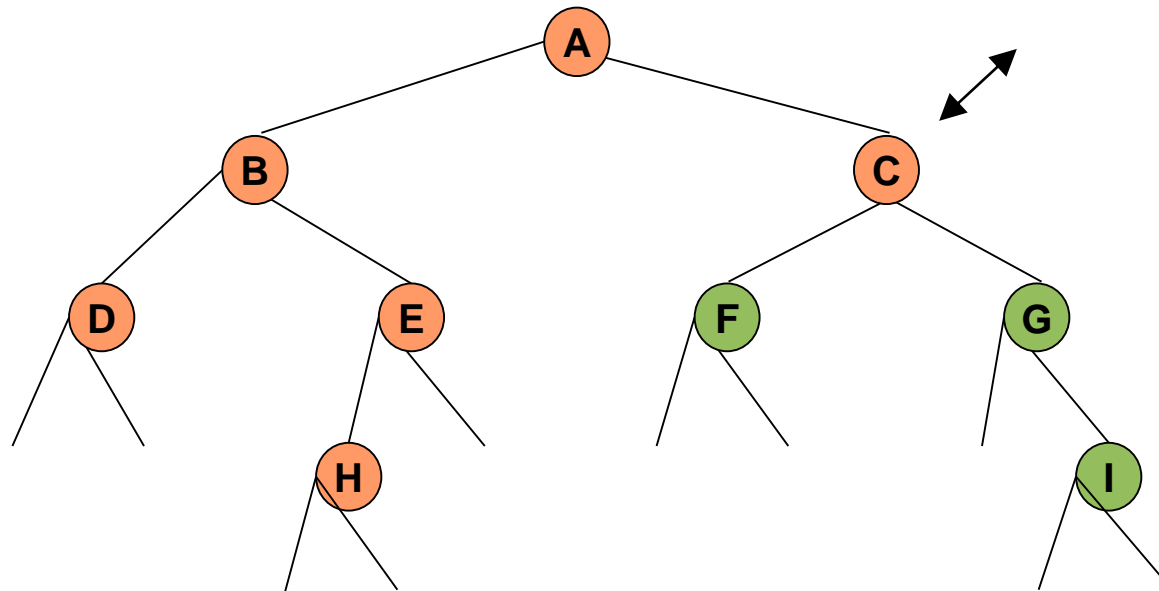
Result: ABDE

Traversing a Tree Preorder



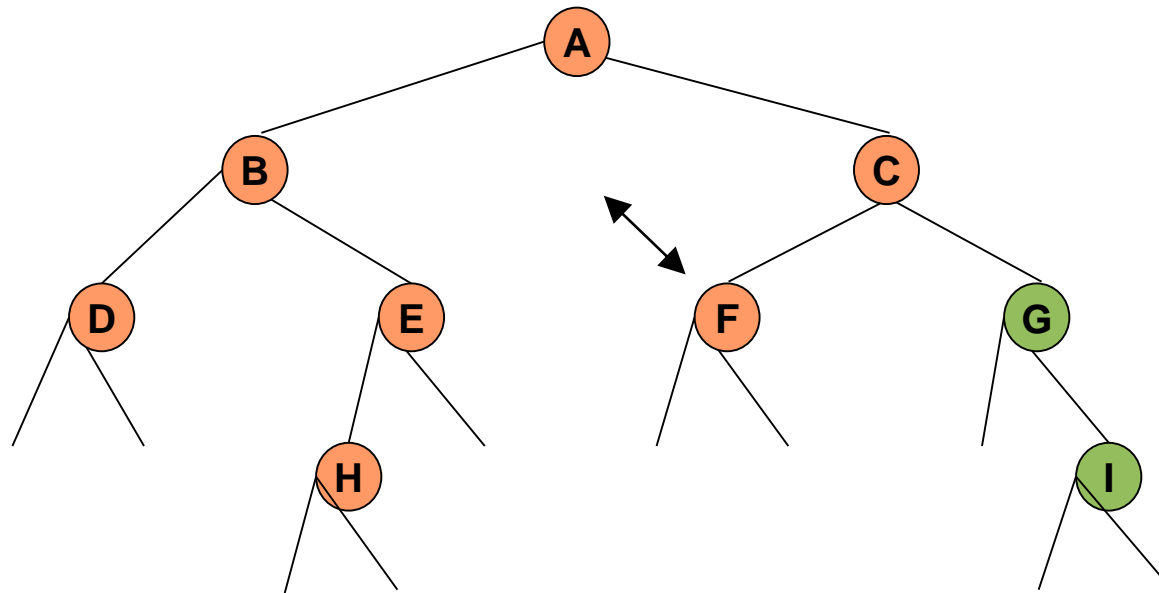
Result: ABDEH

Traversing a Tree Preorder



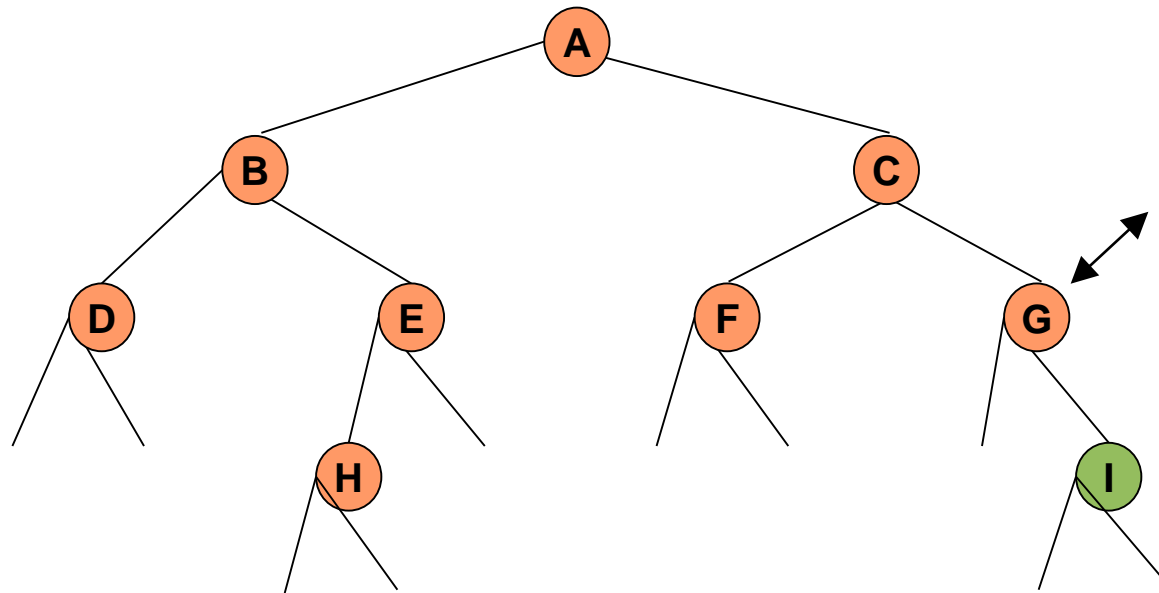
Result: ABDEHC

Traversing a Tree Preorder



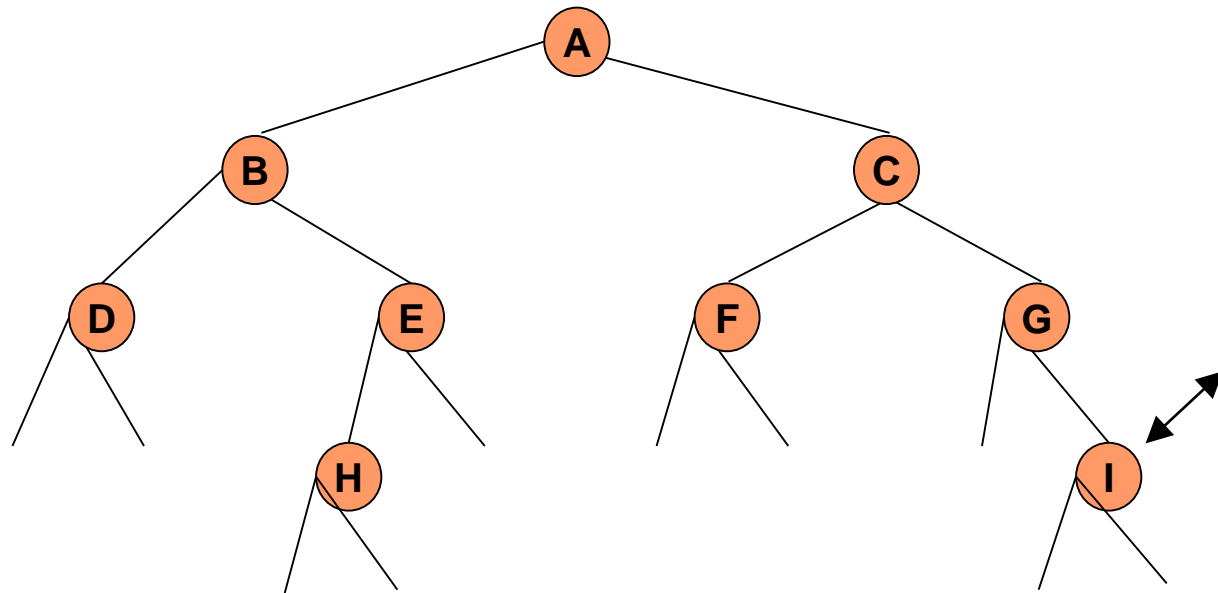
Result: ABDEHCF

Traversing a Tree Preorder



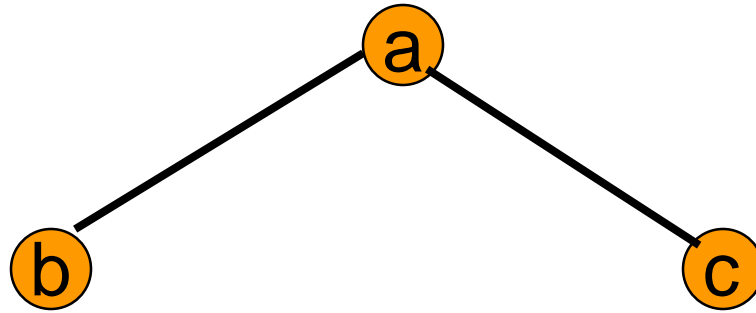
Result: ABDEHCFG

Traversing a Tree Preorder



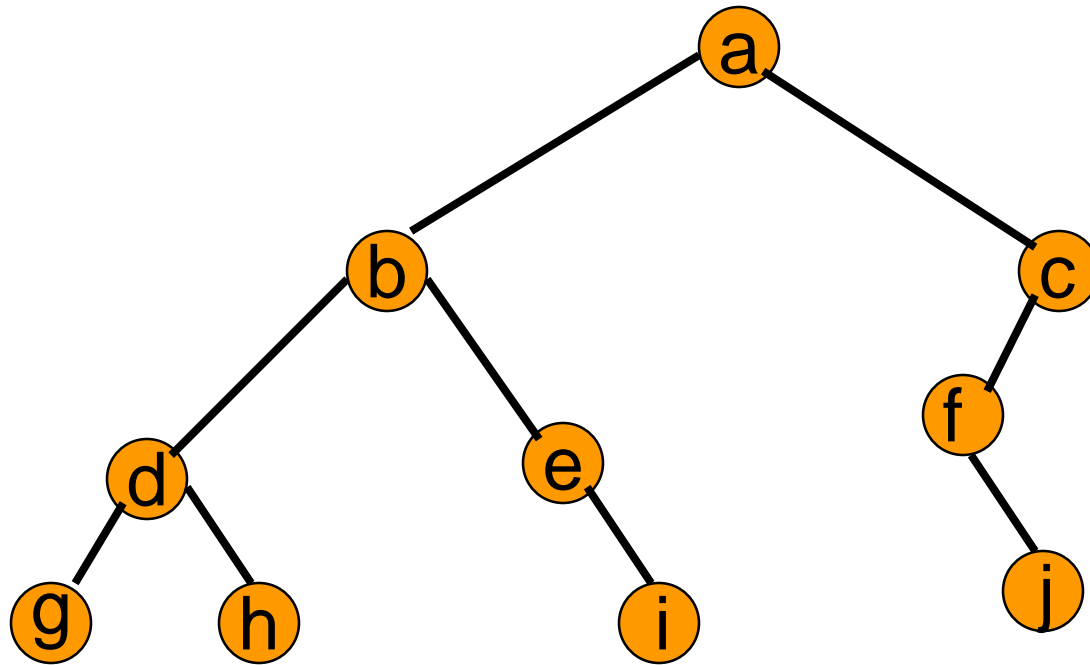
Result: ABDEHCFG I

Preorder Example (Visit = print)



a b c

Preorder Example (Visit = print)



a b d g h e i c f j

Preorder Traversal

```
Void preOrder(BinaryTreeNode
    t)
{
    if (t != null)
    {
        visit(t);
        preOrder(t.leftChild);

        preOrder(t.rightChild);
    }
}
```

Inorder Traversal

- Visit the nodes in the left subtree,
then visit the root of the tree,
then visit the nodes in the right subtree

Inorder(tree)

If tree is not NULL

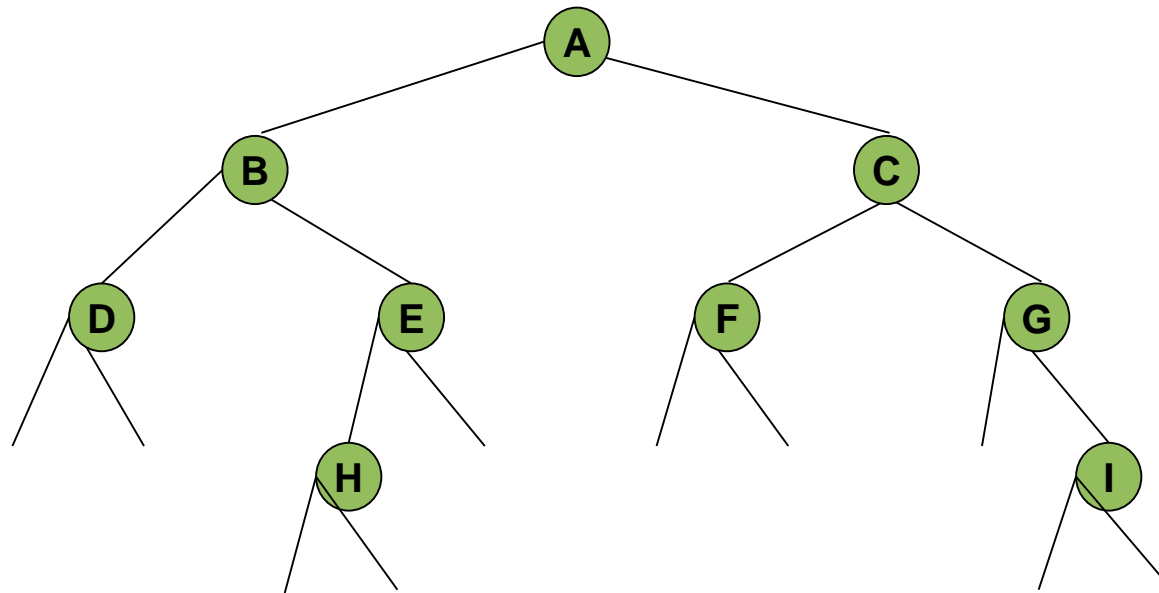
Inorder(Left(tree))

Visit Info(tree)

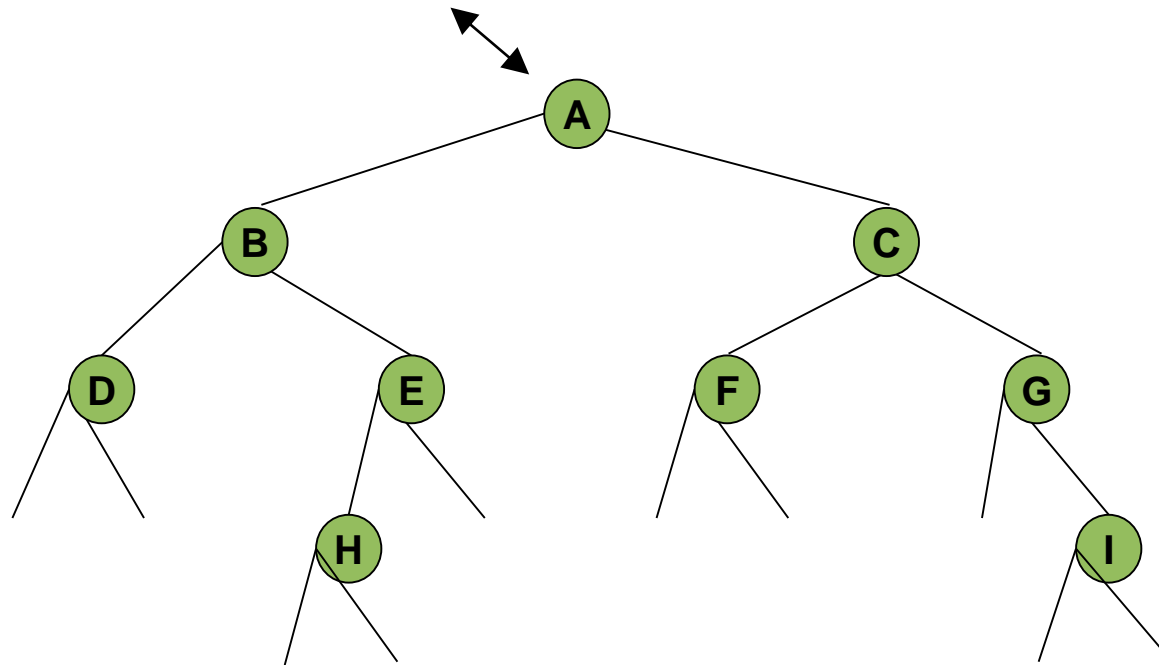
Inorder(Right(tree))

(Warning: "visit" means that the algorithm does something with the values in the node, e.g., print the value)

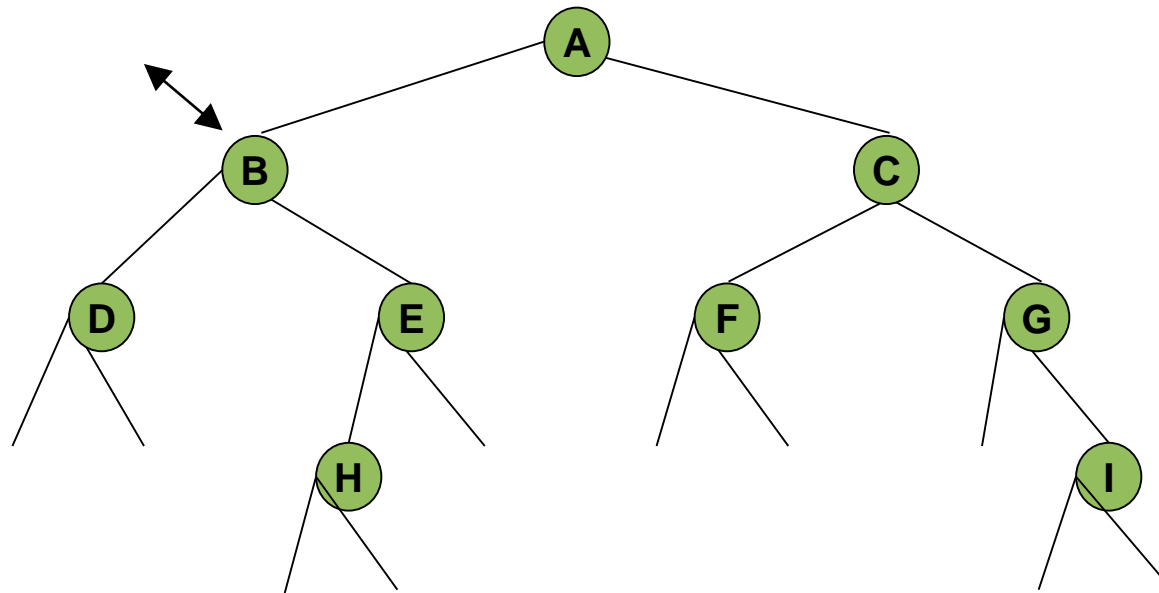
Traversing a Tree Inorder



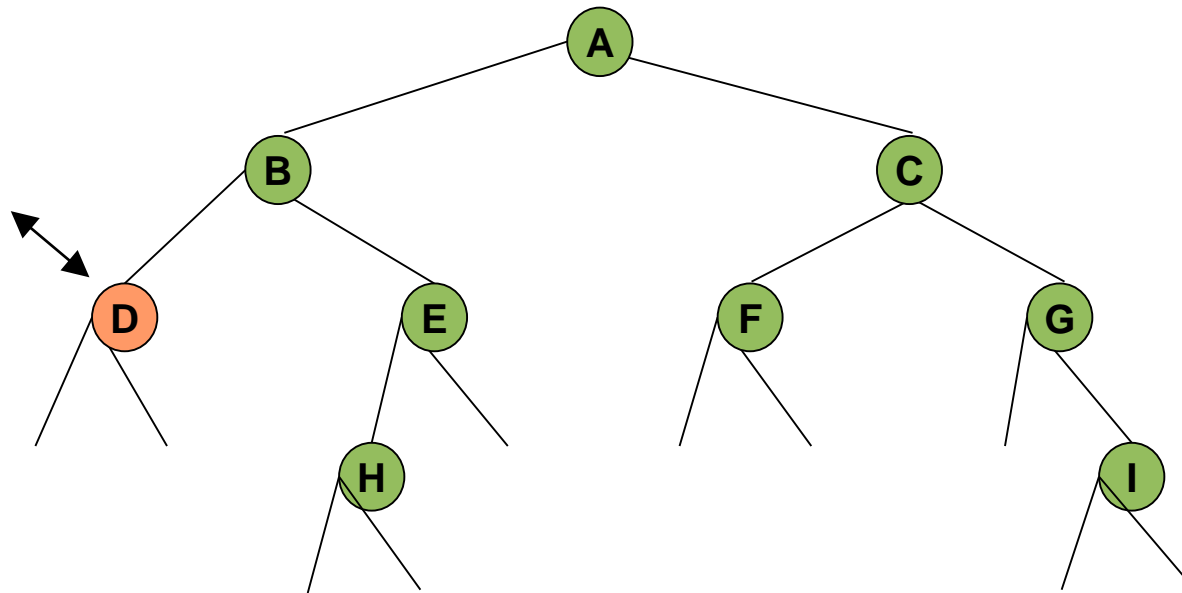
Traversing a Tree Inorder



Traversing a Tree Inorder

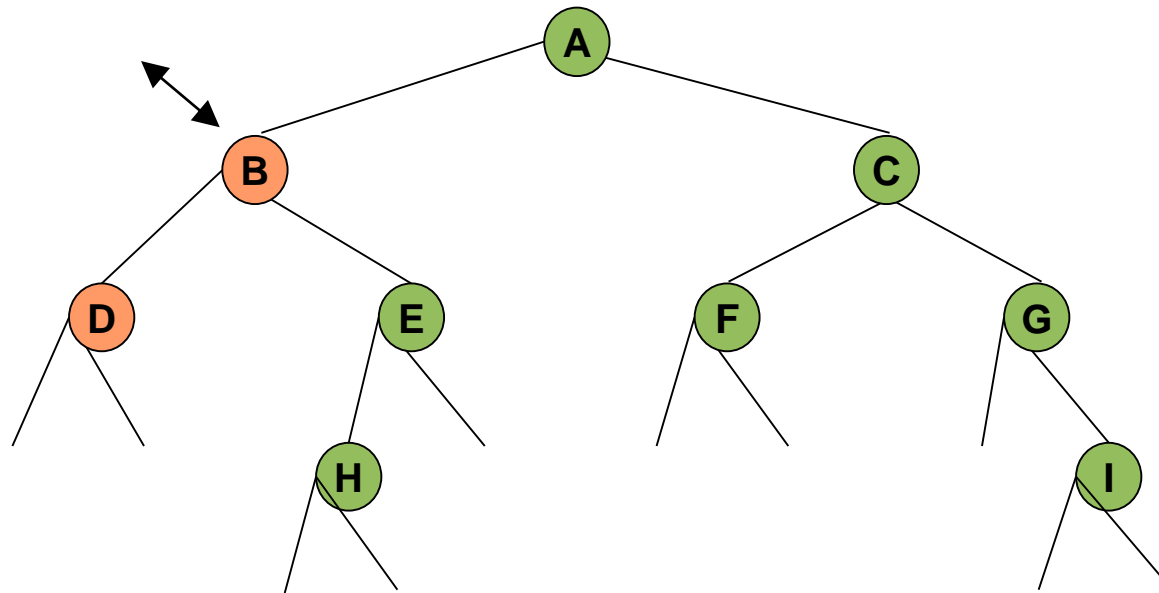


Traversing a Tree Inorder



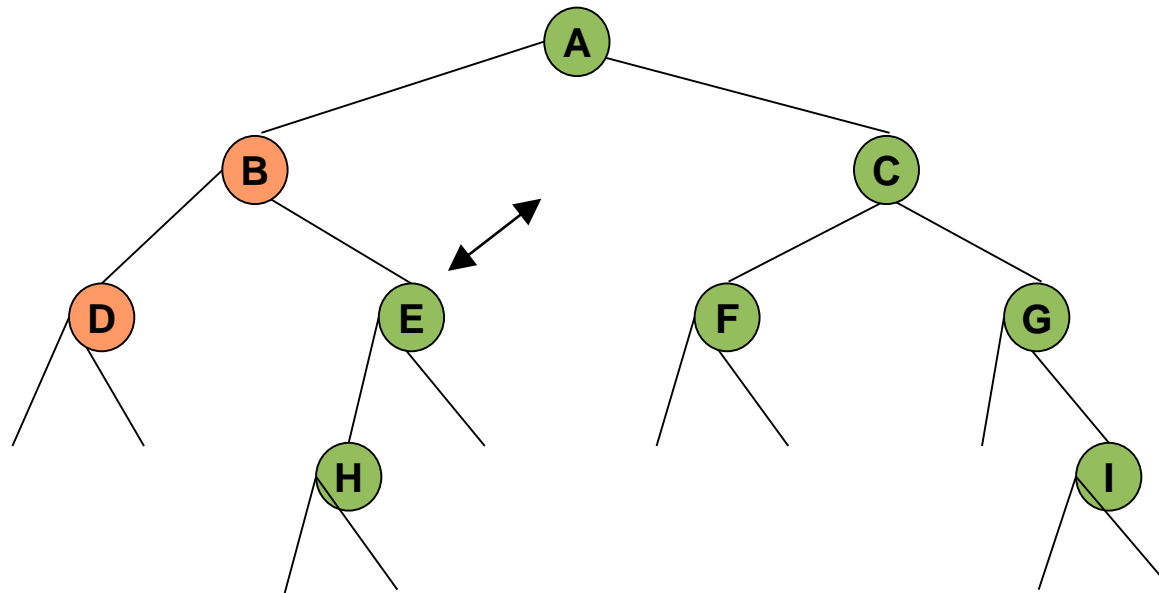
Result: D

Traversing a Tree Inorder



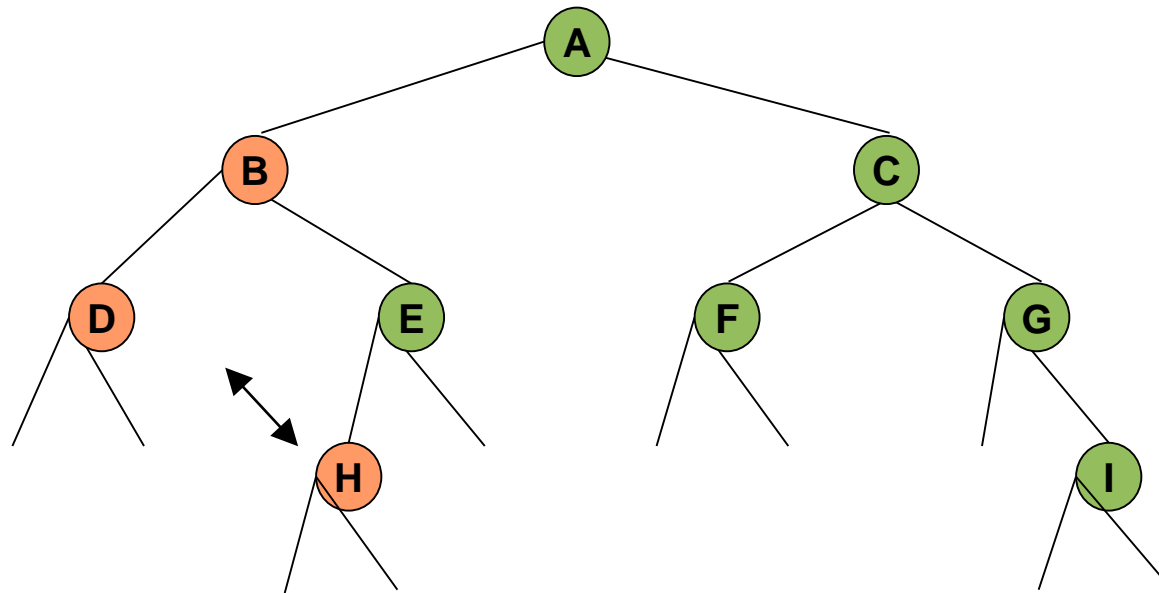
Result: DB

Traversing a Tree Inorder



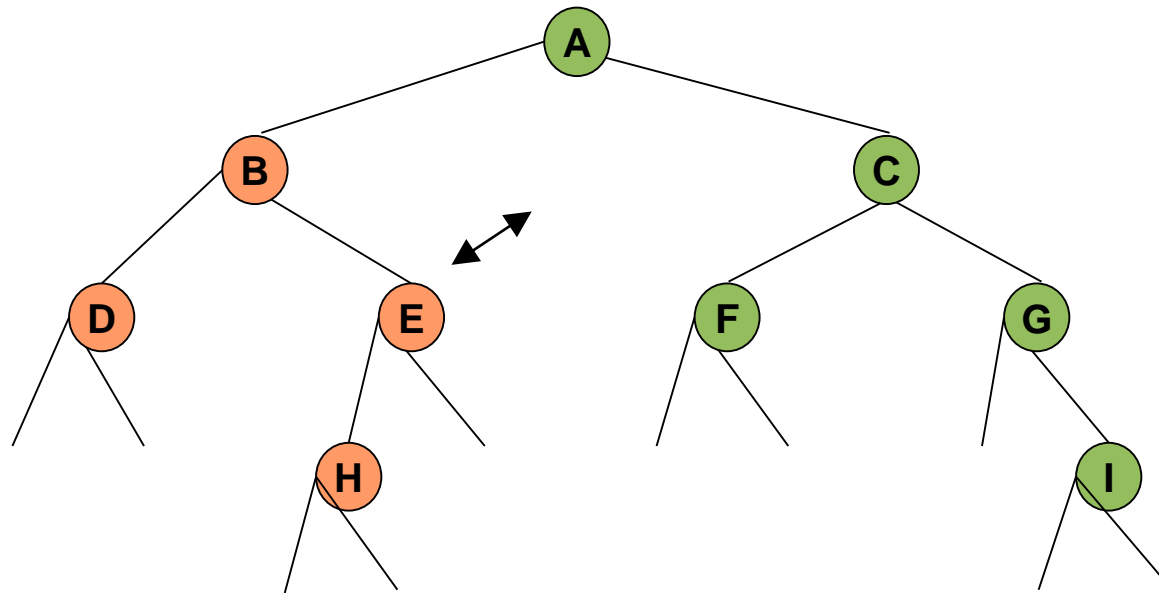
Result: DB

Traversing a Tree Inorder



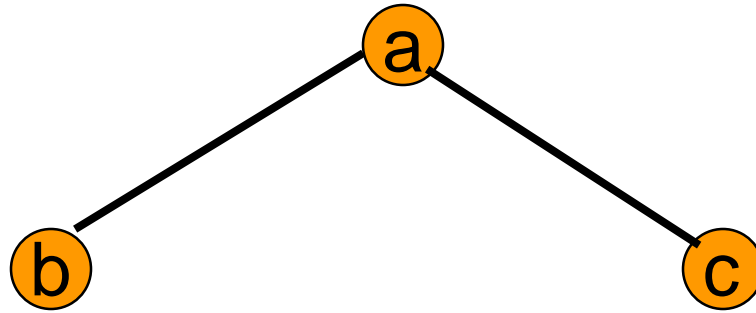
Result: DBH

Traversing a Tree Inorder



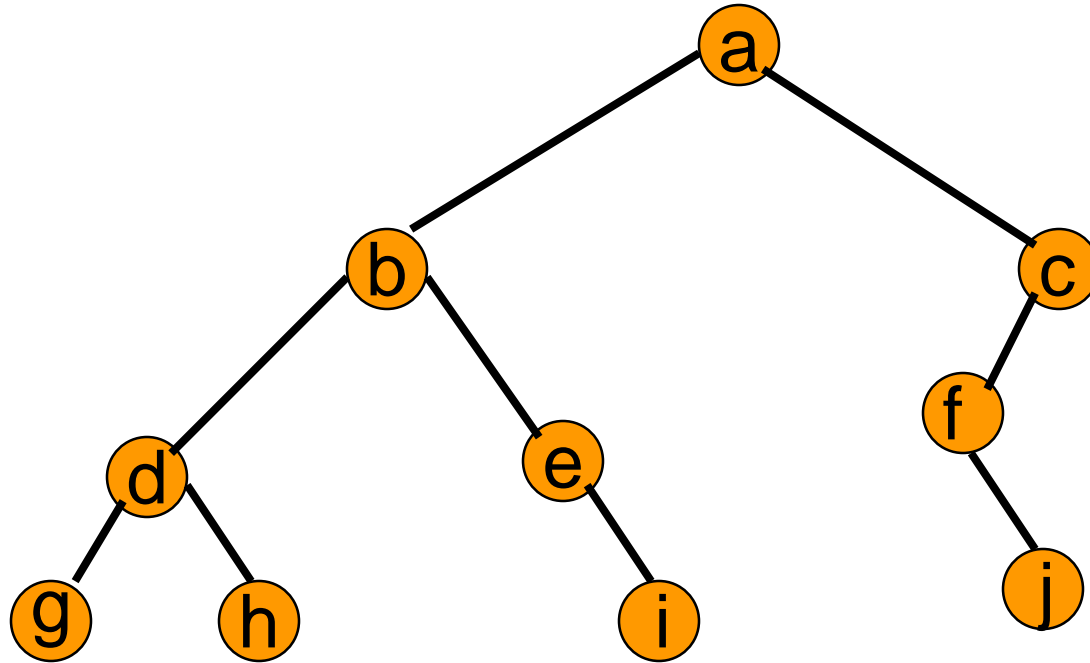
Result: DBHE

Inorder Example (Visit = print)



b a c

Inorder Example (Visit = print)

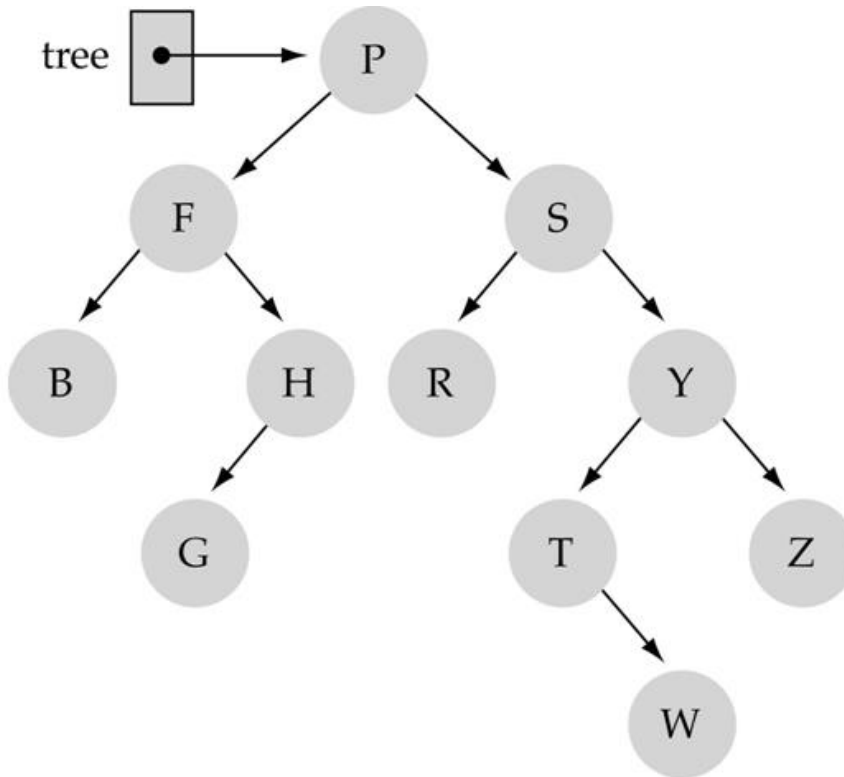


g d h b e i a f j c

Inorder Traversal

```
Void inOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        inOrder(t.leftChild);
        visit(t);
        inOrder(t.rightChild);
    }
}
```

Tree Traversals Example



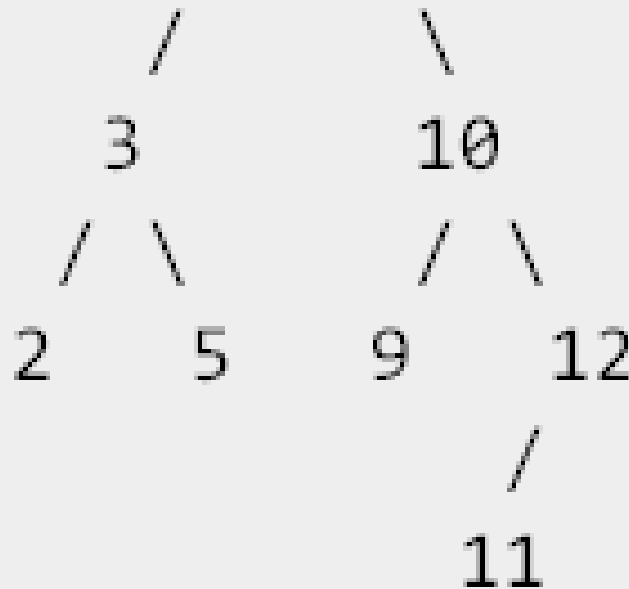
Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

Construct Tree from given Inorder and Postorder traversals

Inorder is: **2 3 5 7 9 10 11 12**

Postorder is: **2 5 3 9 11 12 10 7**

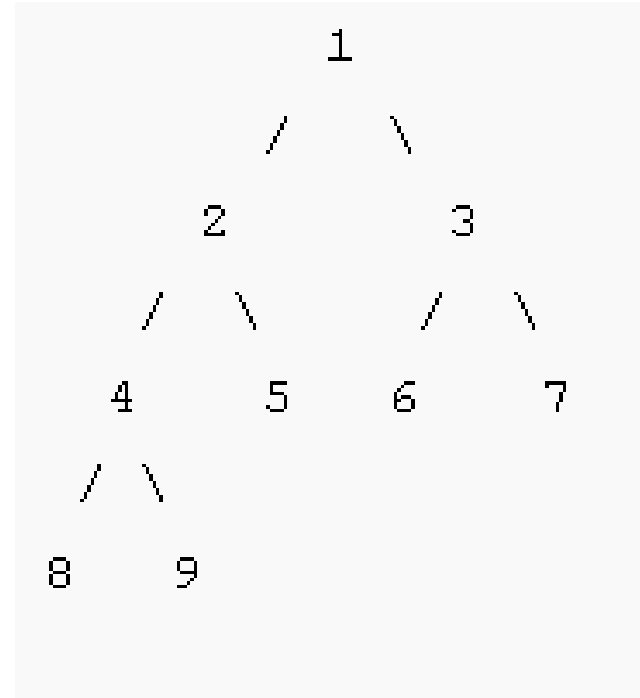
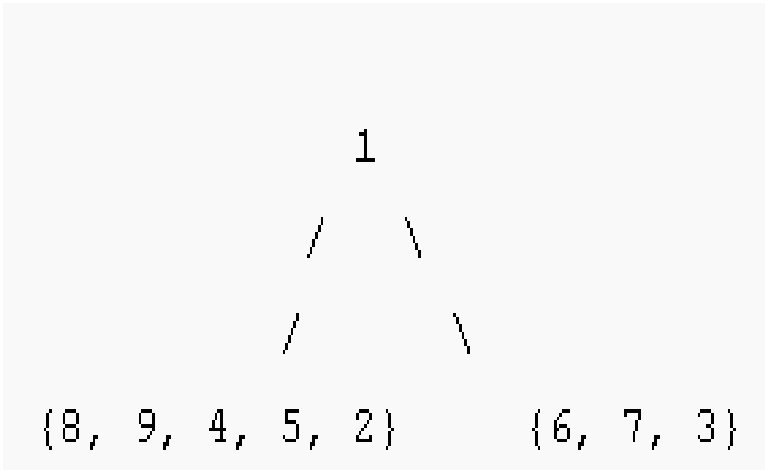
Search for **7** in Inorder sequence. Once we know position of **7** (or index of **7**) in Inorder sequence, we also know that all elements on left side of **7** are in left subtree and elements on right are in right subtree.



Construct Tree from given Preorder and Postorder traversals

```
preorder = {1, 2, 4, 8, 9, 5, 3, 6, 7}
postorder = {8, 9, 4, 5, 2, 6, 7, 3, 1};
```

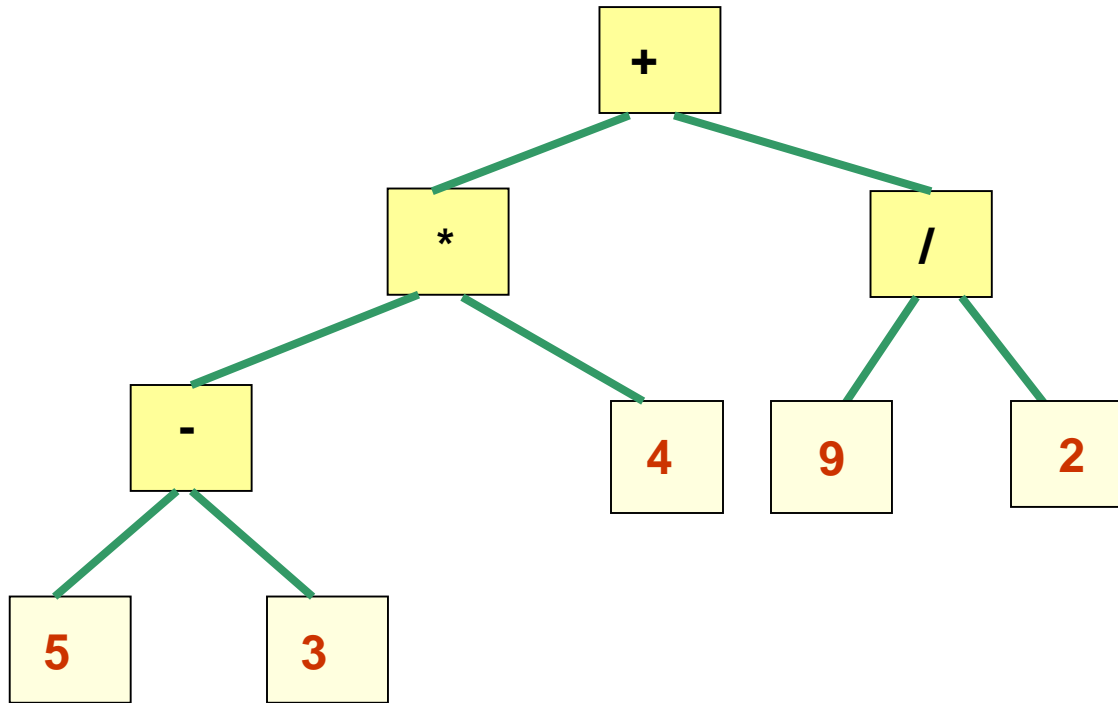
No pre[], know left and right elements, {8,9} of
 5. All nodes before 29 in post[] must be in left
 free. Same as left subtree, and the elements
 subtree.
 more, they are in the value next to 1 in pre[],
 must be left child of root.



Expression tree

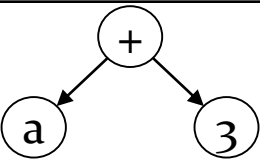
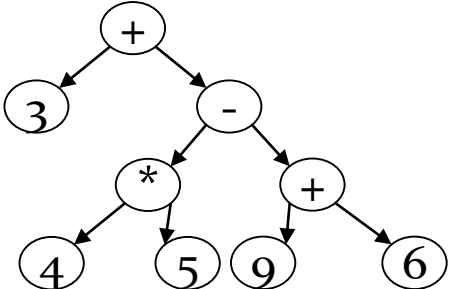
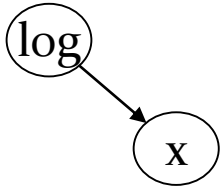
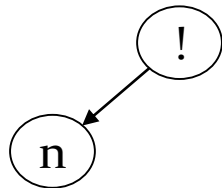
- An expression tree for an arithmetic, relational, or logical expression is a binary tree in which:
 - The parentheses in the expression do not appear.
 - The leaves are the variables or constants in the expression.
 - The non-leaf nodes are the operators in the expression:
 - A node for a binary operator has two non-empty subtrees.
 - A node for a unary operator has one non-empty subtree.
- The operators, constants, and variables are arranged in such a way that an inorder traversal of the tree produces the original expression without parentheses.

Example: An Expression Tree



$(5 - 3) * 4 + 9 / 2$

Expression Tree Examples

Expression	Expression Tree	Inorder Traversal Result
$(a+3)$	 <pre> graph TD A((+)) --> B((a)) A --> C((3)) </pre>	$a + 3$
$3+(4*5-(9+6))$	 <pre> graph TD A((+)) --> B((3)) A --> C((-)) C --> D((*)) C --> E((+)) D --> F((4)) D --> G((5)) E --> H((9)) E --> I((6)) </pre>	$3+4*5-9+6$
$\log(x)$	 <pre> graph TD A((log)) --> B((x)) </pre>	$\log x$
$n!$	 <pre> graph TD A((!)) --> B((n)) </pre>	$n !$

Why Expression trees?

- Expression trees are used to remove ambiguity in expressions.
- Consider the algebraic expression $2 - 3 * 4 + 5$.
- Without the use of precedence rules or parentheses, different orders of evaluation are possible:

$$((2-3)*(4+5)) = -9$$

$$((2-(3*4))+5) = -5$$

$$(2-((3*4)+5)) = -15$$

$$(((2-3)*4)+5) = 1$$

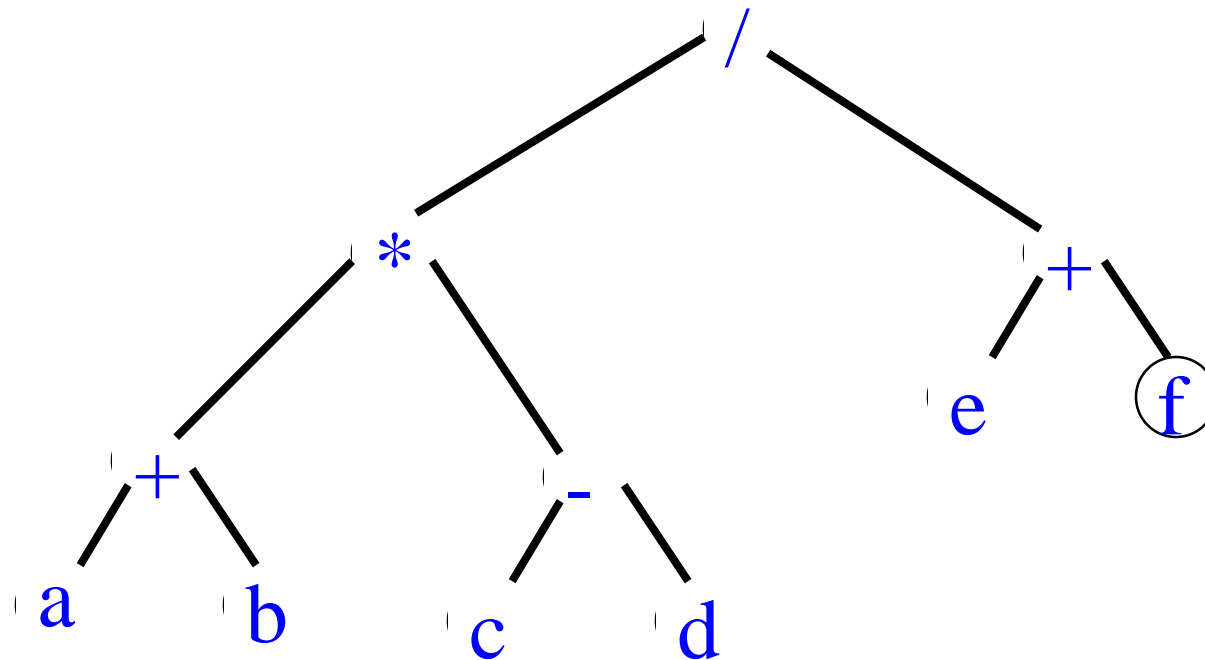
$$(2-(3*(4+5))) = -25$$

- The expression is ambiguous because it uses infix notation: each operator is placed between its operands.

- Expression trees can be very useful for:
 - Evaluation of the expression.
 - Generating correct compiler code to actually compute the expression's value at execution time.
 - Performing symbolic mathematical operations (such as differentiation) on the expression.

Expression	Expression Tree	Infix form
$2 - 3 * 4 + 5$	<pre> graph TD Plus((+)) --> Minus((-)) Plus --> 5((5)) Minus --> 2((2)) Minus --> Star((*)) Star --> 3((3)) Star --> 4((4)) </pre>	$2 - 3 * 4 + 5$
$(2 - 3) * (4 + 5)$	<pre> graph TD Star((*)) --> Minus((-)) Star --> Plus((+)) Minus --> 2((2)) Minus --> 3((3)) Plus --> 4((4)) Plus --> 5((5)) </pre>	$2 - 3 * 4 + 5$
$2 - (3 * 4 + 5)$	<pre> graph TD Minus((-)) --> 2((2)) Minus --> Plus((+)) Plus --> Star((*)) Plus --> 5((5)) Star --> 3((3)) Star --> 4((4)) </pre>	$2 - 3 * 4 + 5$

Constructing Expression Tree Using Postfix Expression



`a b + c d - * e f + /`