Data Structures

IT 205

Dr. Manish Khare

Lecture – 18
19-Feb-2018

Binary Tree Representations

- A binary tree data structure is represented using two methods. Those methods are as follows...
  - **Array Representation**
  - **Linked List Representation**

Binary Tree Representation using Array

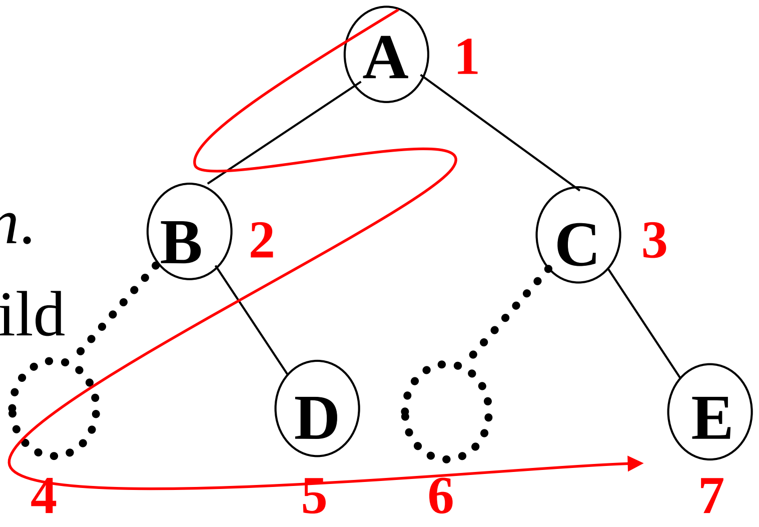
- In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

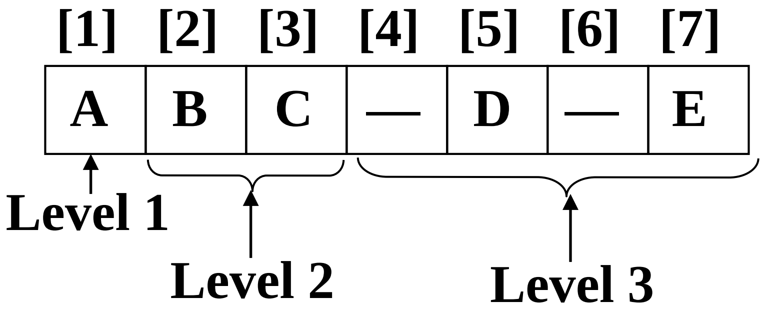
- To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of $\mathbf{2^{n+1} - 1}$.

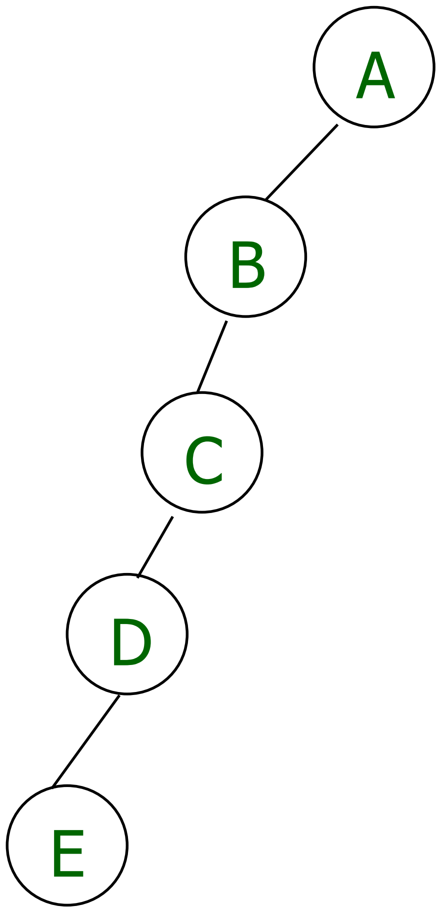
Binary Tree Representation using Array

- Binary tree representations (using array)
  - **Lemma 5.3:** If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

1. *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
   If $i = 1$, i is at the root and has no parent.
2. *LeftChild(i)* is at $2i$ if $2i \le n$.
   If $2i > n$, then i has no left child.
3. *RightChild(i)* is at $2i+1$ if $2i+1 \le n$.
   If $2i + 1 > n$, then i has no right child



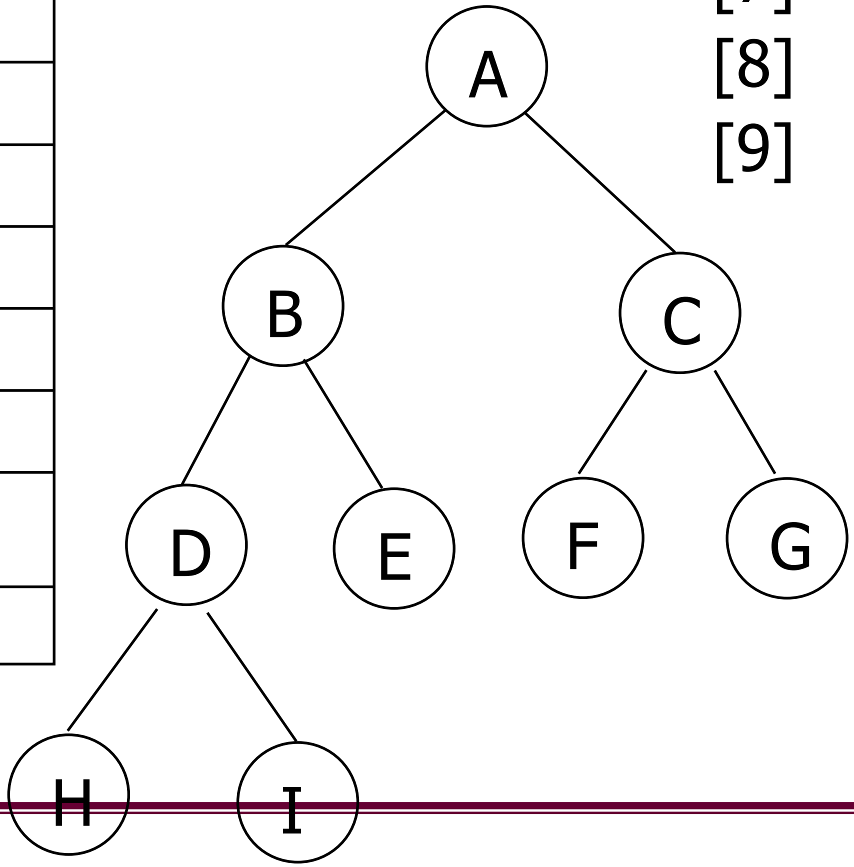

[1]
[2]
[3]
[4]
[5]
[6]
[7]
A
B
C
—
D
—
E
Level 1
Level 2
Level 3



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

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

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



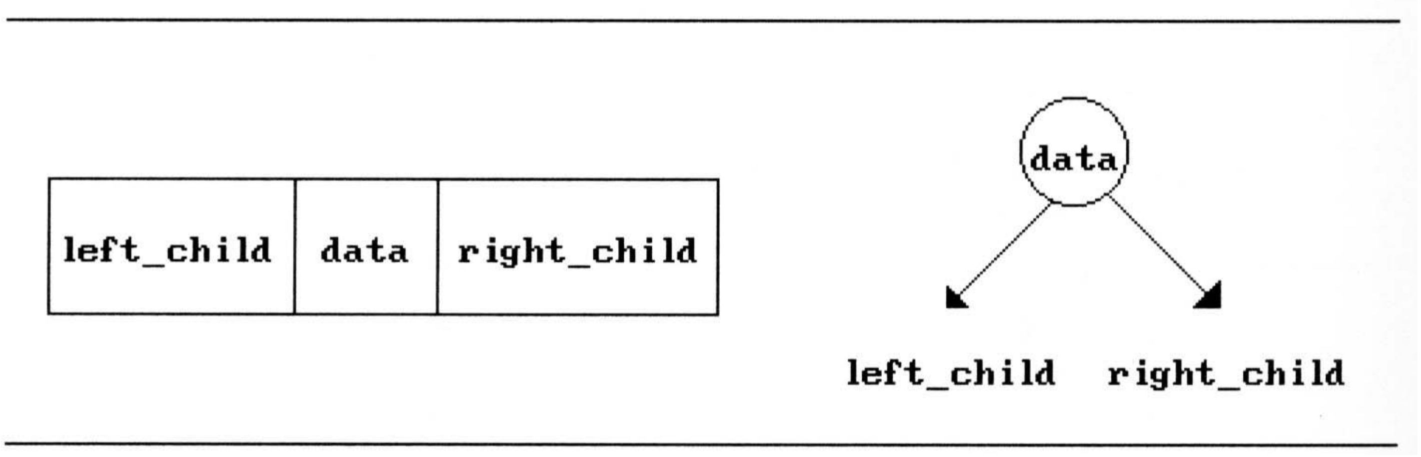
- Waste spaces: in the worst case, a skewed tree of depth *k* requires 2^k-1 spaces. Of these, only *k* spaces will be occupied

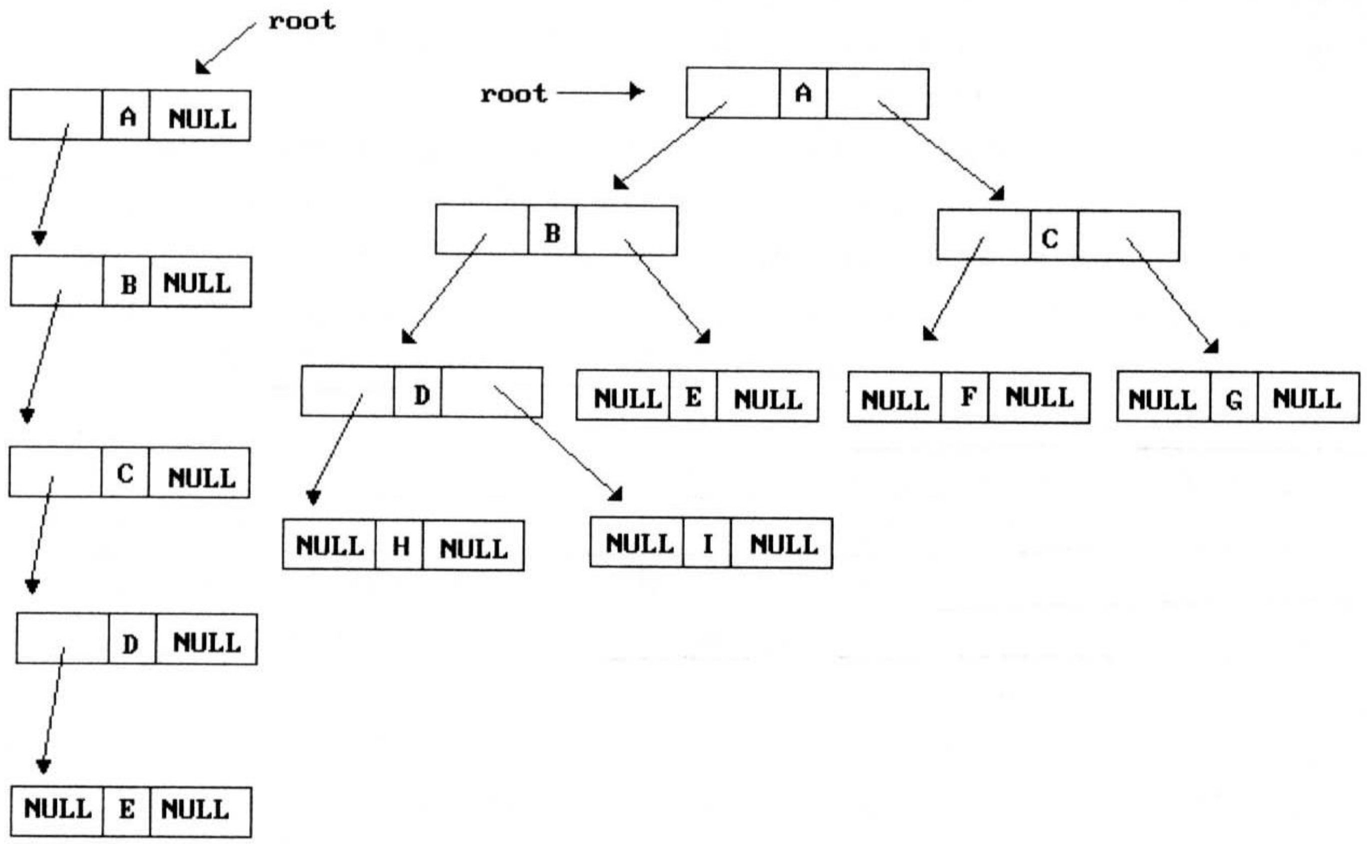
- Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes

Binary Tree Representation using Linked List

- Although the array representation is good for complete binary trees, it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations.
- Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes.
- These problems can be overcome easily through the use of linked representation.
- Each node has three fields: left child, data, and right child.

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```





root
A
NULL
B
NULL
C
NULL
D
NULL
NULL
E
NULL
root
A
B
C
D
NULL
E
NULL
NULL
F
NULL
NULL
G
NULL
NULL
H
NULL
NULL
I
NULL

Binary Tree Traversal

- When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.
- **Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.**
- 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.

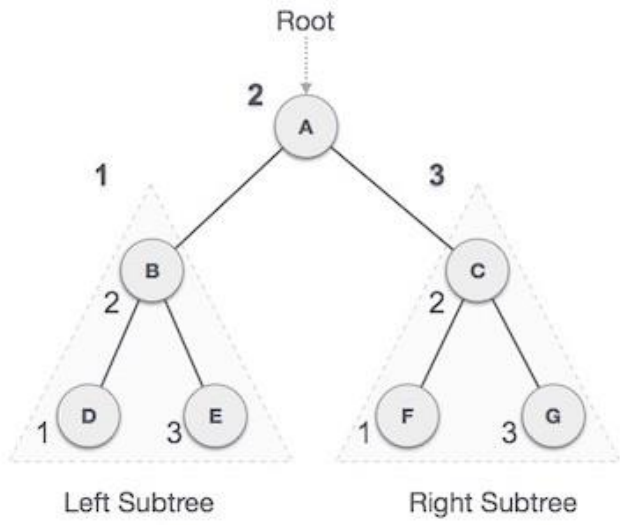
- There are three types of binary tree traversals.
  - **In - Order Traversal (LDR)**
  - **Pre - Order Traversal (DLR)**
  - **Post - Order Traversal (LRD)**

- Three other types of binary tree traversals
  - Reverse In-Order Traversal (RDL)
  - Reverse Pre-Order Traversal (DRL)
  - Reverse Post-Order Traversal (RLD)

In-order Traversal

- In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.
- **Algorithm**
  - Until all nodes are traversed –
  - **Step 1** – Recursively traverse left subtree.
  - **Step 2** – Visit root node.
  - **Step 3** – Recursively traverse right subtree.

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
   if (ptr) {
      inorder(ptr->left_child);
      printf("%d",ptr->data);
      inorder(ptr->right_child);
   }
}
```

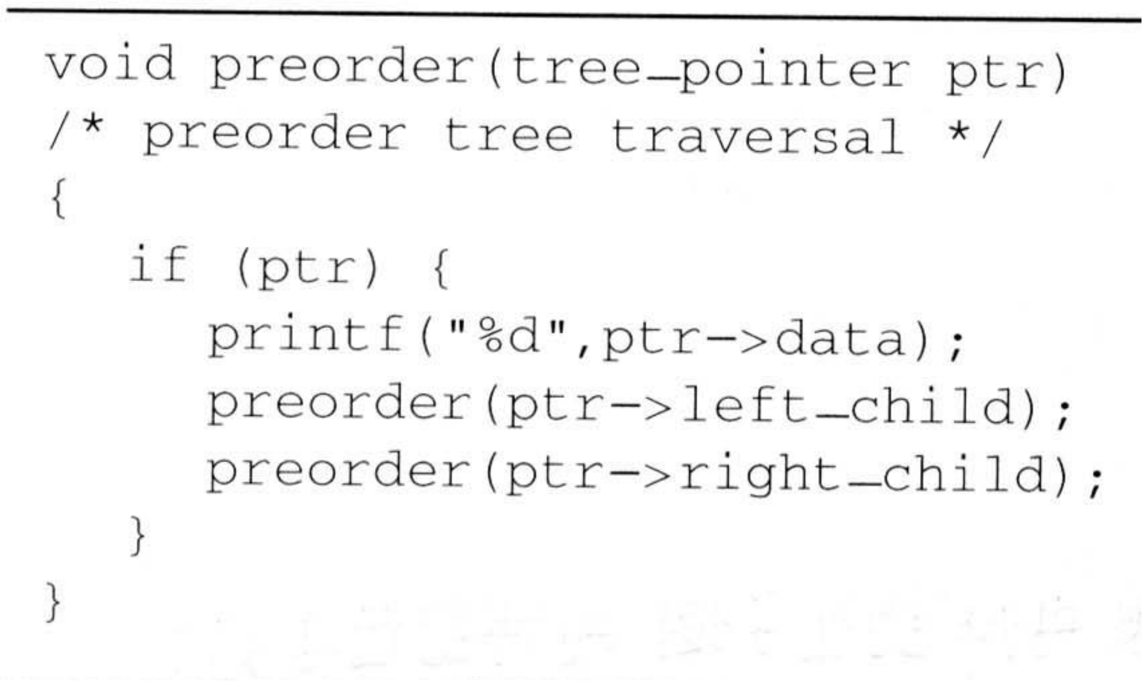


- We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

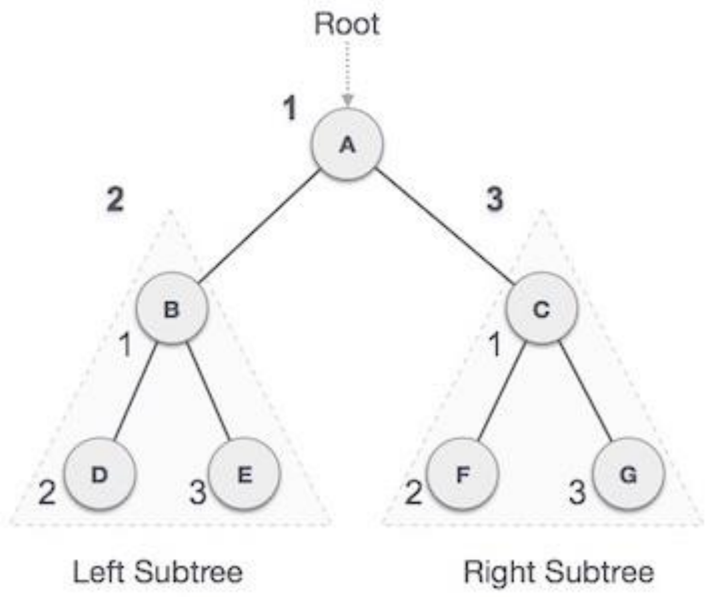
$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Pre-Order Traversal

- In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.
- Algorithm
  - Until all nodes are traversed –
  - **Step 1** – Visit root node.
  - **Step 2** – Recursively traverse left subtree.
  - **Step 3** – Recursively traverse right subtree.



```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
   if (ptr) {
      printf("%d",ptr->data);
      preorder(ptr->left_child);
      preorder(ptr->right_child);
   }
}
```



- We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

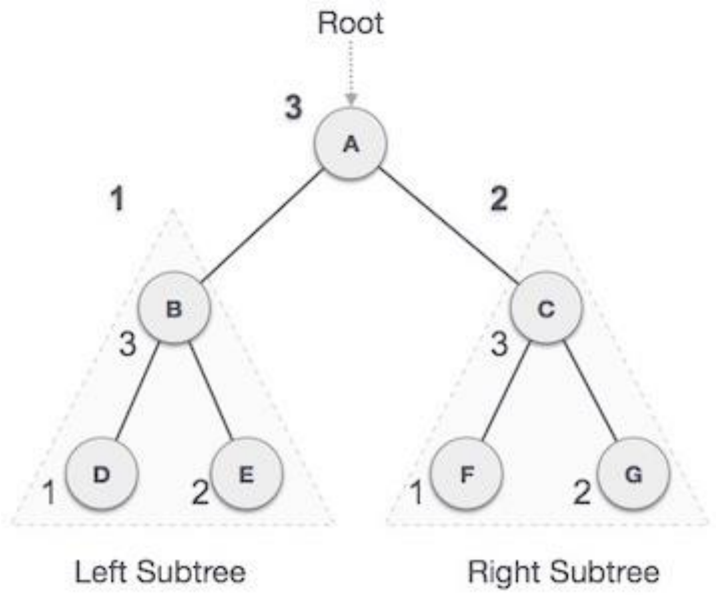
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post-Order Traversal

- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

- Algorithm
  - Until all nodes are traversed –
  - **Step 1** – Recursively traverse left subtree.
  - **Step 2** – Recursively traverse right subtree.
  - **Step 3** – Visit root node.

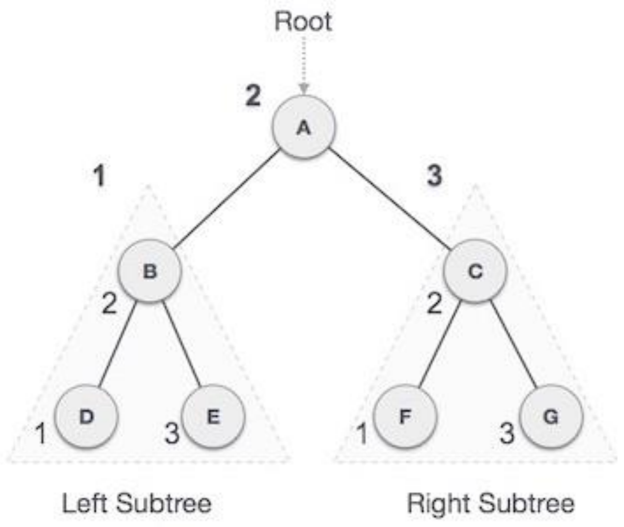
```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
   if (ptr) {
      postorder(ptr->left_child);
      postorder(ptr->right_child);
      printf("%d",ptr->data);
   }
}
```



- We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

- Reverse Tree Traversal
  - Reverse In-Order Traversal (RDL)
  - Reverse Pre-Order Traversal (DRL)
  - Reverse Post-Order Traversal (RLD)



- Reverse In-Order Traversal

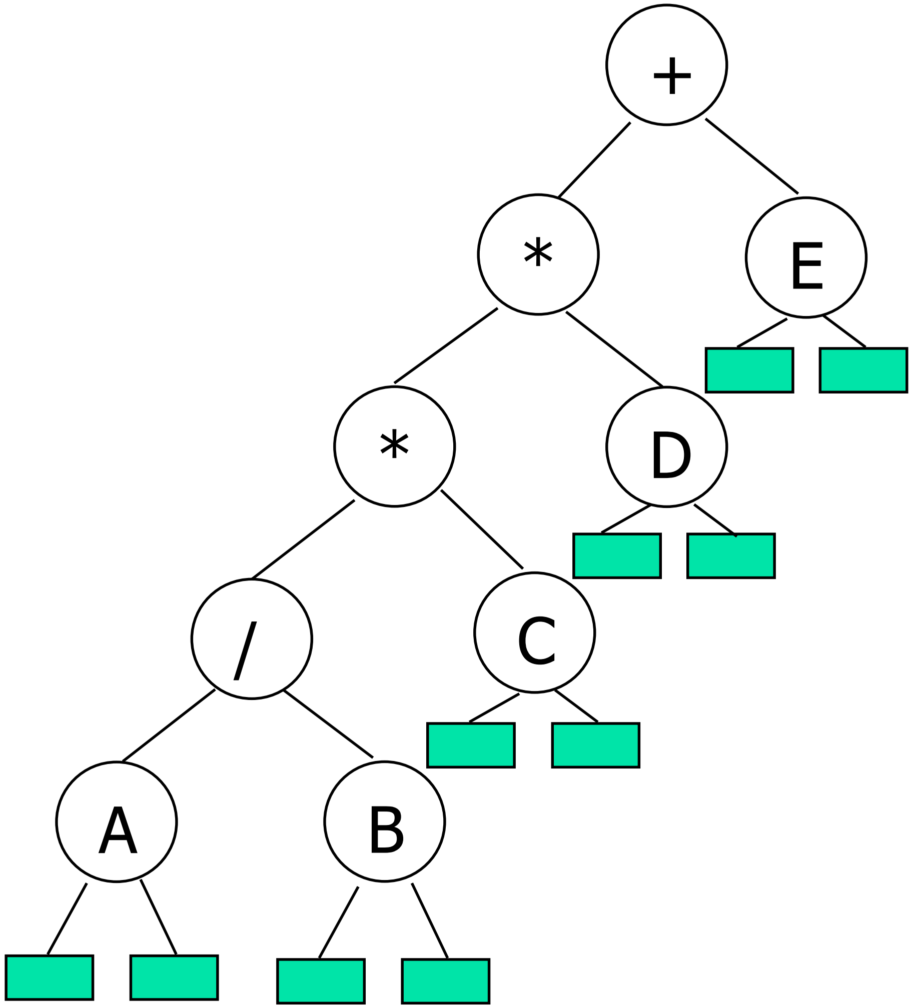
G-C-F-A-E-B-D

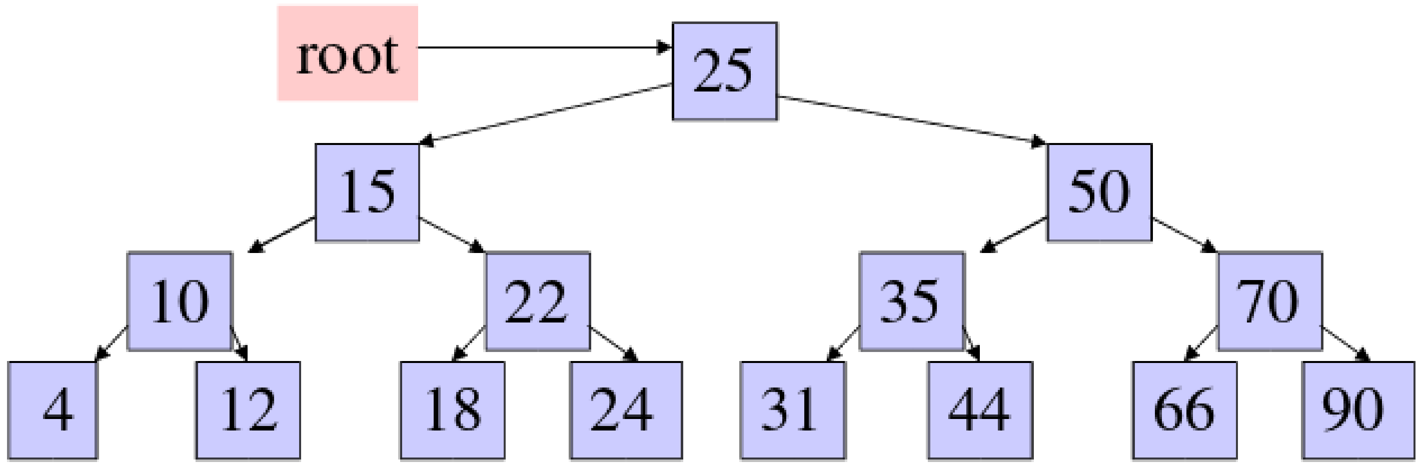
- Reverse Pre-Order Traversal

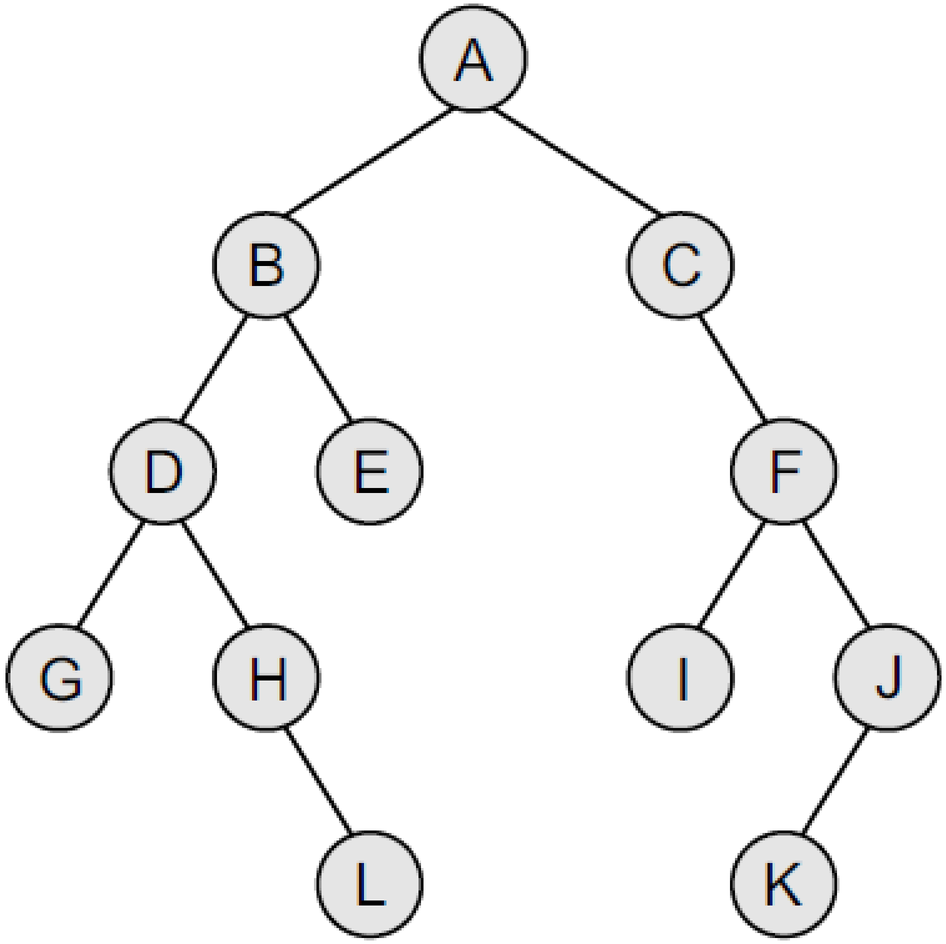
A-C-G-F-B-E-D

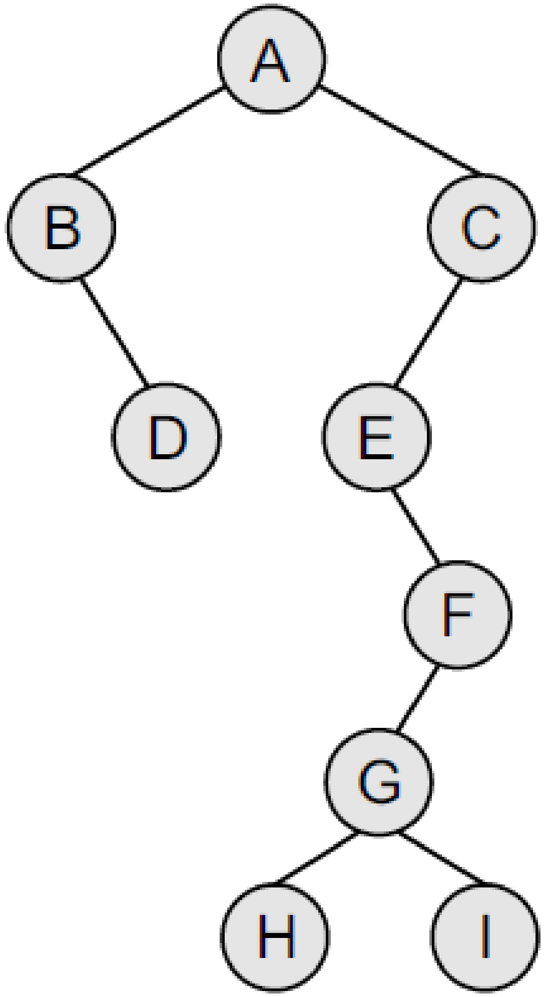
- Reverse Post-Order Traversal

G-F-C-E-D-B-A


+
*
E
*
D
/
C
A
B


root
25
15
50
10
22
35
70
4
12
18
24
31
44
66
90


A
B
C
D
E
F
G
H
I
J
L
K


A
B
C
D
E
F
G
H
I

Level-Order Traversal

- Level-order traversal
  - method:
    - We visit the root first, then the root’s left child, followed by the root’s right child.
    - We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost nodes
  - This traversal requires a queue to implement

➢Level-order traversal (using queue)

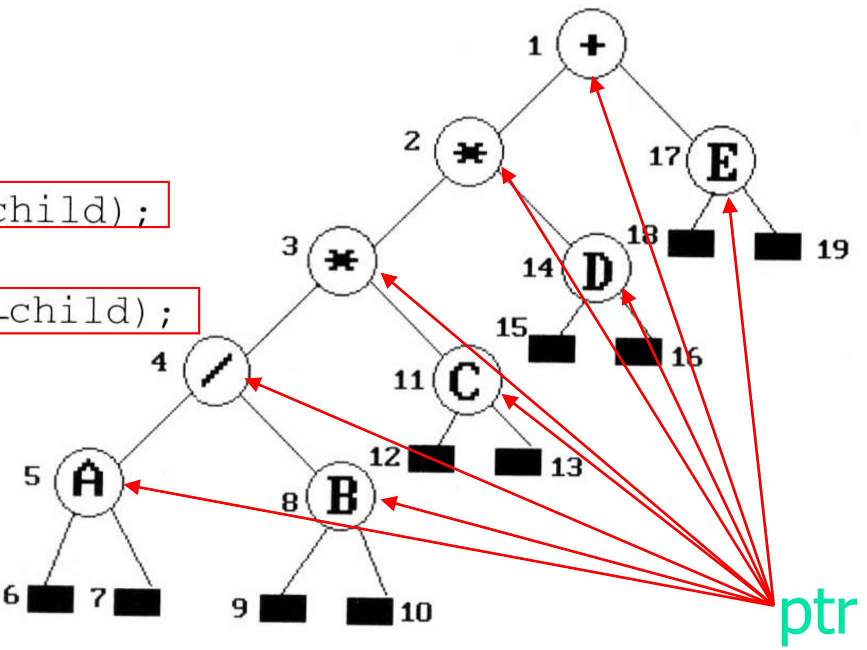
```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
  int front = rear = 0;
  tree_pointer queue[MAX_QUEUE_SIZE];
  if (!ptr) return; /* empty tree */
  addq(front, &rear, ptr);
  for (;;) {
    ptr = deleteq(&front, rear);
    if (ptr) {
      printf("%d",ptr->data);
      if(ptr->left_child)
        addq(front,&rear,ptr->left_child);
      if (ptr->right_child)
        addq(front,&rear,ptr->right_child);
    }
    else break;
  }
}
```

FIFO

Program 5.5: Level order traversal of a binary tree

output: +*E *D /C A B

2	17	3	14	4	11	5	8	
*	E	*	D	/	C	A	B	



Constructing a Binary Tree from Traversal Results