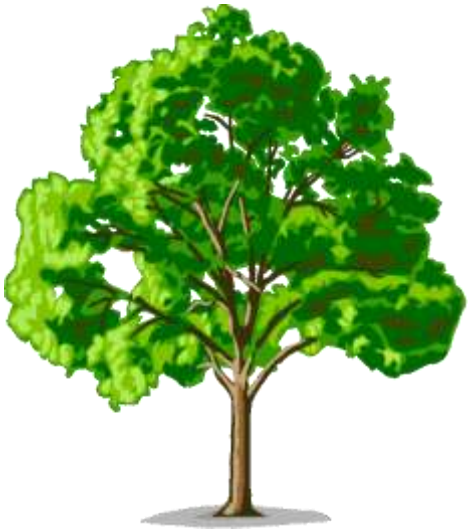


# TREE
# DATA STRUCTURE

Dr.Pradeep Kumar Mallick

# Outline

- Definition
- Tree Terminology
- Binary Trees
- Types of Binary Trees
- Some Q & A
- Expression Trees
- Tree Traversal
- Construction of Binary Tree
- Construction of Expression Trees and its Evaluation
- MCQs and Assignments
- Binary Search Trees (BST)
- Creation, Insertion & Deletion Operations on BSTs
- Height Balanced Trees (AVL Trees) (Insertion & Deletion)
- B-Trees and B+ Trees (Insertion & Deletion)
- Heap Trees (Max Heap & Min Heap)
- Assignments and Doubt Clearance

# Why Study Trees???

They give the hierarchical relationship between different elements.

Good example of recursive data structures
Methods are recursive algorithms

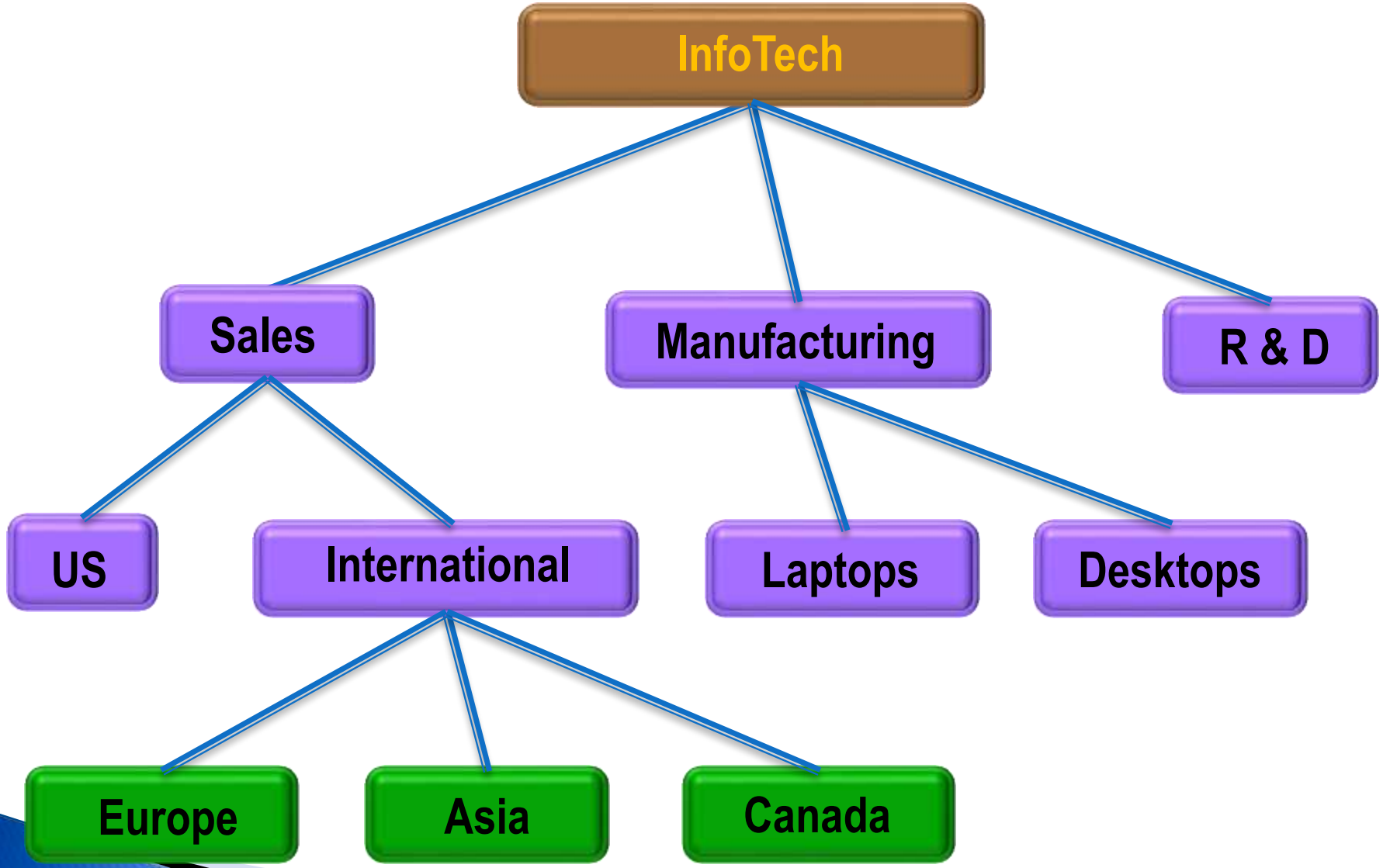
For language processing (human or computer).
Eg: Parse trees

For knowledge representation and modeling of the "real world" relations.
E.g. family trees; the Linnaean taxonomy (kingdom, phylum, …, species); etc.

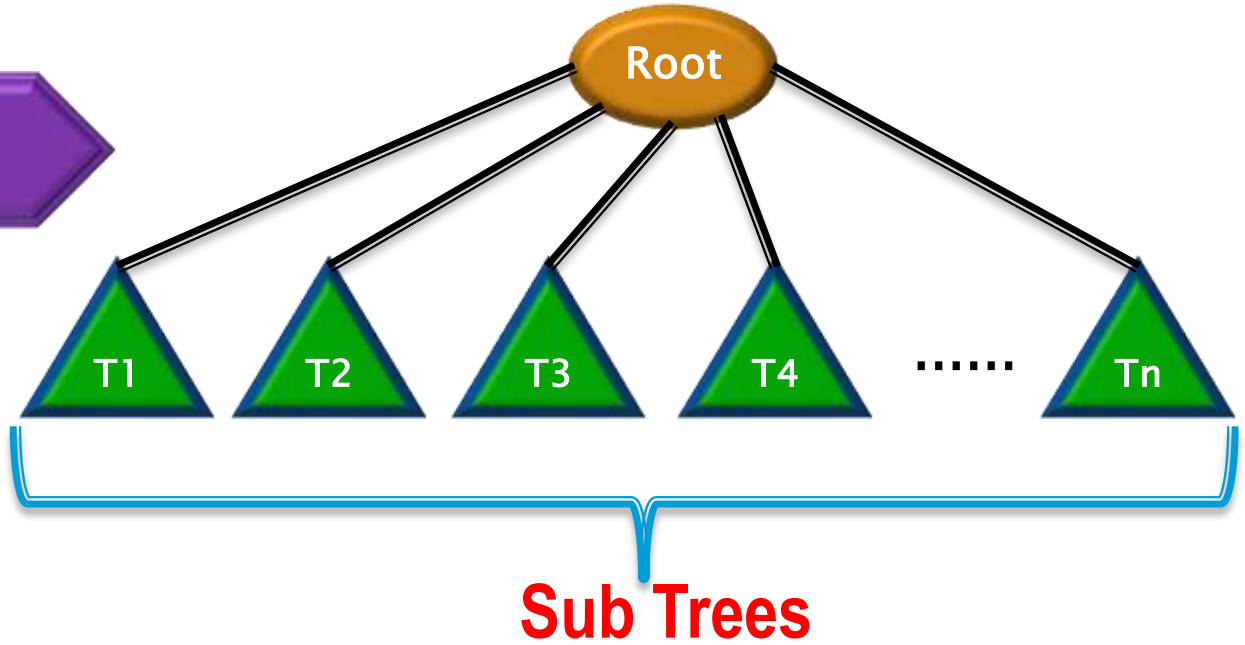
A Real World Tree

A Tree Data Structure

# Data Organized in a Tree Structure



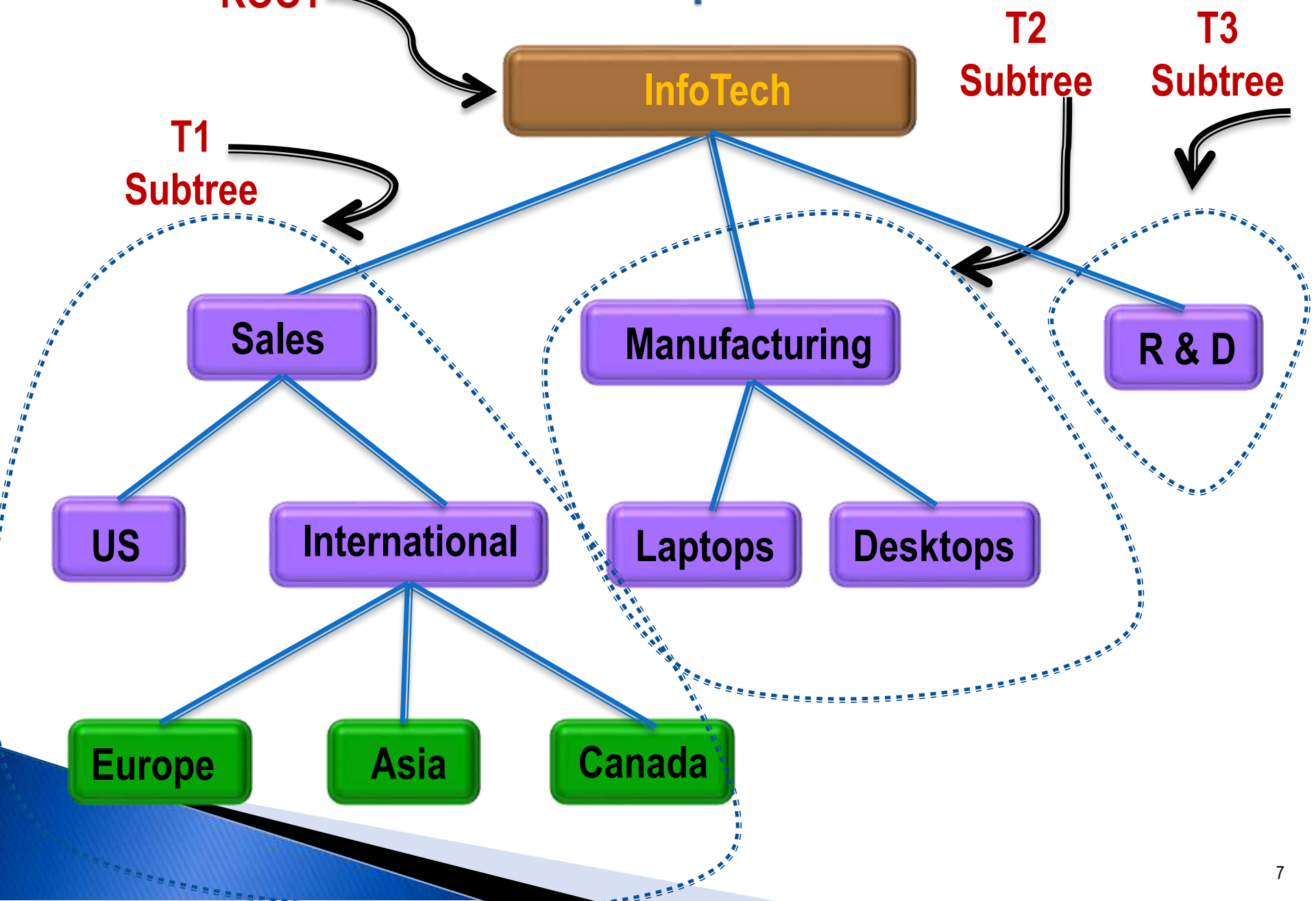
## A General Tree




❖ A *tree* structure means that the data are organized so that items of information are related by branches.

❖ It is used to represent hierarchical relationship existing among several data items.

# Example


ROOT
InfoTech
T1
Subtree
T2
Subtree
T3
Subtree
Sales
Manufacturing
R & D
US
International
Laptops
Desktops
Europe
Asia
Canada

# What is a Tree???

A tree is a non-linear data structure made up of nodes or vertices and edges without having any cycle.

The tree with no nodes is called the null or empty tree.

A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

## Definition:

A **tree** is a finite set of one or more data items (nodes) such that

- There is a specially designated node called the ***root*** node,
- The remaining nodes are partitioned into n>=0 disjoint set

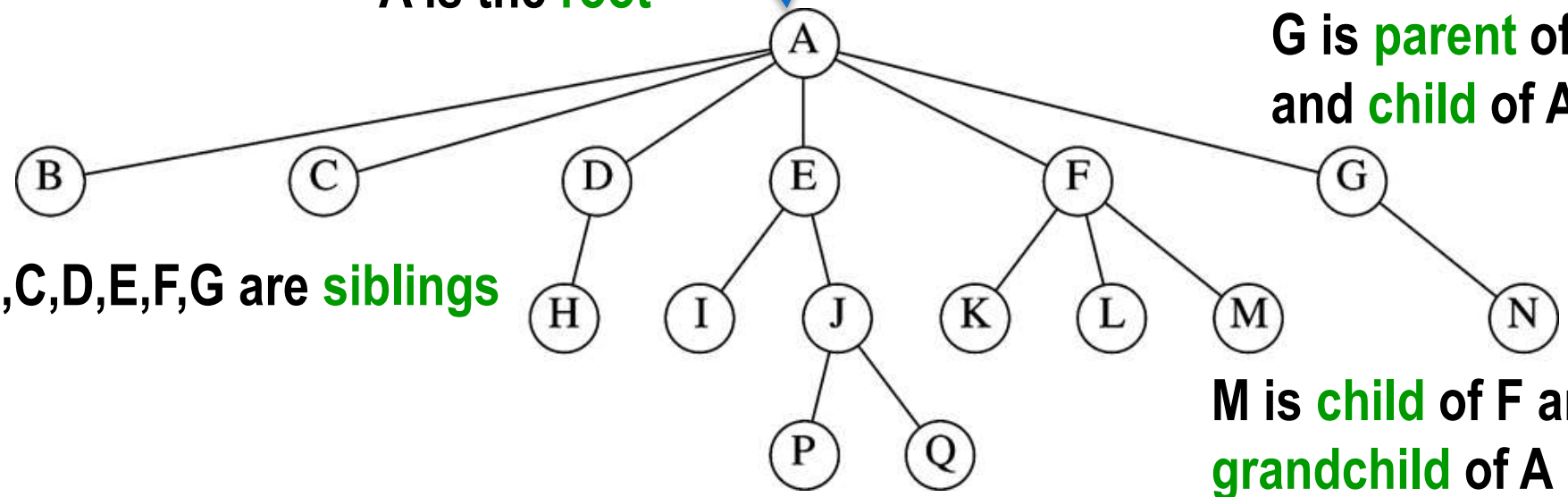
T1,…,Tn, where each of these sets is a tree. T1,…,Tn are called the **subtrees** of the root.

# Familiarity with Tree Terms

B,C,D,E,F and G are children of A

A is the root

G is parent of N and child of A




B,C,D,E,F,G are siblings

M is child of F and grandchild of A

A is an ancestor of P

P is a descendant of A

B,C,H,I,P,Q,K,L,M,N are leaves

K,L,M are siblings

I and J are siblings

P and Q are siblings

# TREE TERMINOLOGY

- **Node:** Each data item in a tree is a called a **node.** It specifies the data information and links to other data items.
- **Root:** It is a specially designated data item in a tree. It is the first in the hierarchical arrangement of data items.
- **Subtree:** Tree consisting of a node and its descendants.
- **Degree:** The number of immediate nodes (subtrees) attached to a node.
- **Degree of a tree:** The maximum degree of node in a tree.
- **Terminal/ Leaf node(s):** A node with degree zero.
- **Non-terminal/ Interior node(s):** Nodes with at least degree one.
- **Child node:** The immediate successor of a node.
- **Parent node:** The immediate predecessor of a node.
- **Ancestors:** Parent, grandparent, great-grandparent, etc.
- **Descendants:** Child, grandchild, great-grandchild, etc.

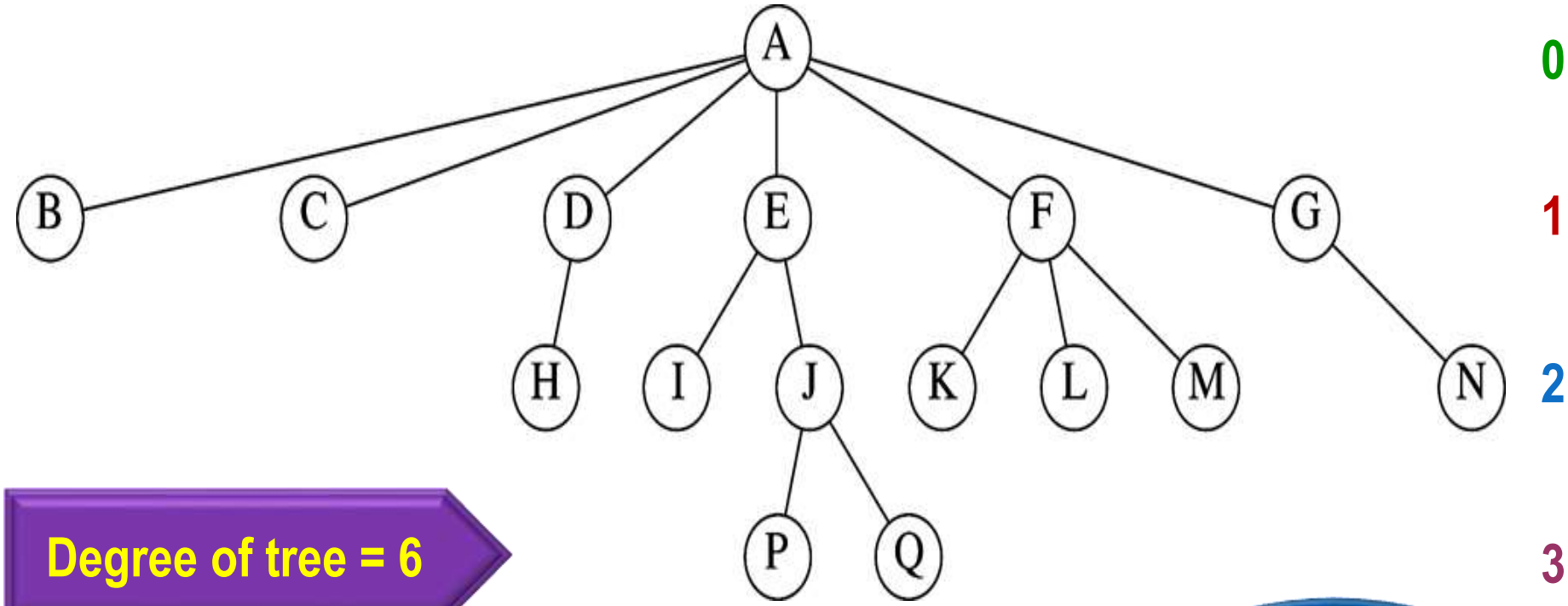
# Contd…

- **Siblings:** Nodes having the same parent node.
- **Edge:** A line connecting two nodes (parent and child nodes).
- **Path:** A sequence of consecutive edges from the source node to the destination node.
- **Height of a node:** The height of a node $n_i$ is the length of the longest path from $n_i$ to a leaf node.
- **Depth of a node:** The depth of a node $n_i$ is the length of the longest path from root to $n_i$.
- **Forest**: A forest is a set of n ≥ 0 disjoint trees.

**Height of root = Height of tree = Depth of tree**

**Height of leaf node = 0**

**Depth of root node = 0**




Degree of tree = 6

Degree(A)= 6, Degree(F)= 3, Degree(L)= 0

Height(J)= 1, Height(E)= 2, Height(K)= 0

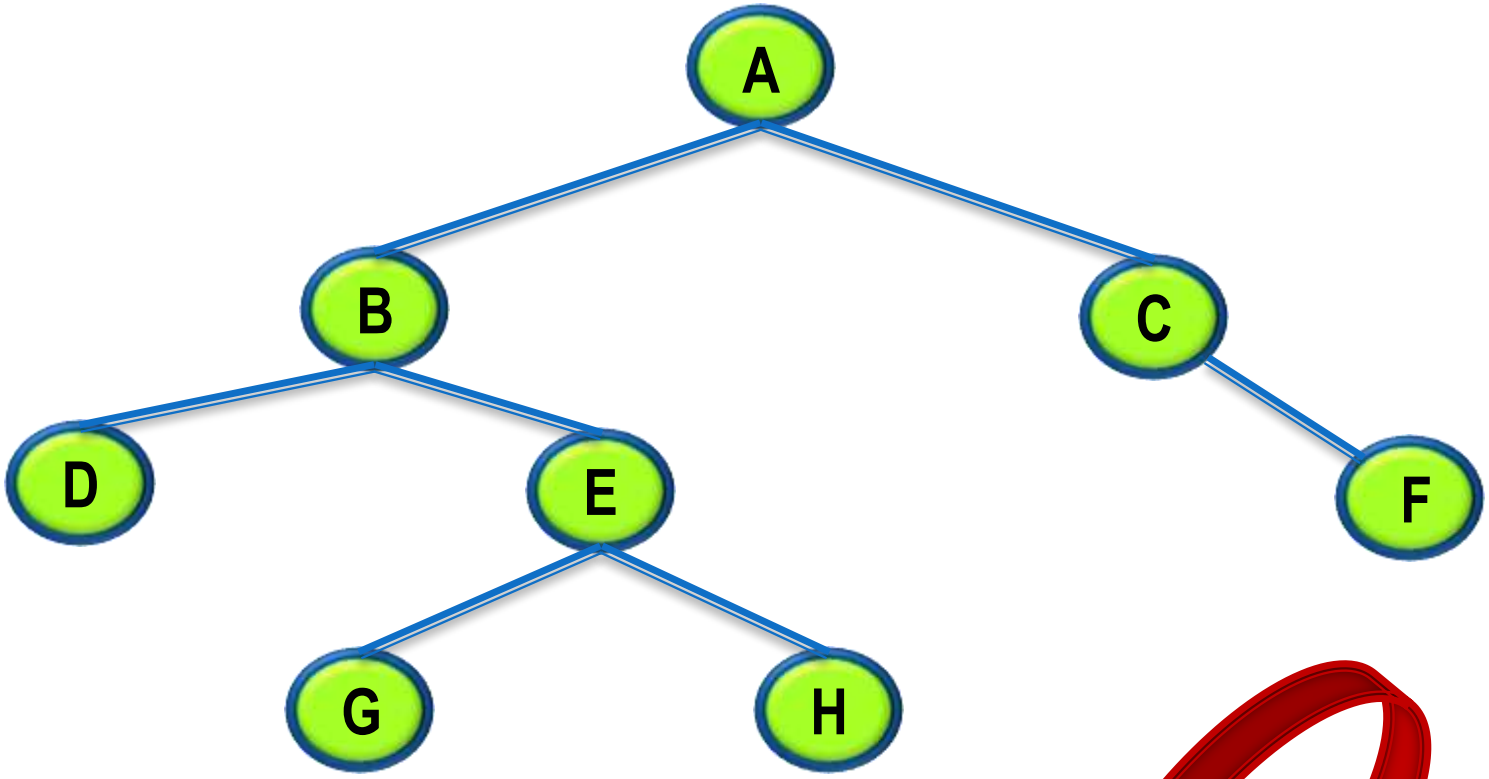
Depth(M)= 2, Depth(G)= 1, Depth(P)= 3

Height of tree = Depth of tree = 3

Root is always at Level 0

Siblings are placed in the same level

# Lets Answer These???




What is the depth of the tree???

3

# Applications of trees

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

# Types of Tree data structure

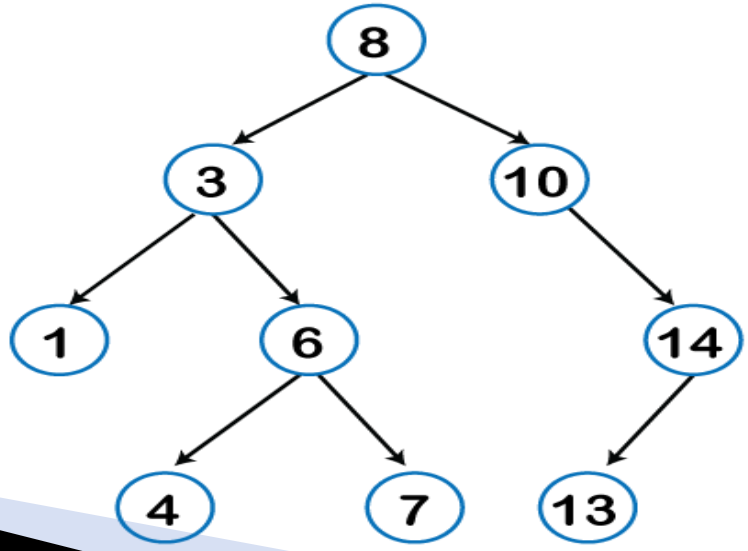
1. **General tree:**

- The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain).
- The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.
- There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
- Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

# Types of Tree data structure

Binary tree:

- A binary tree is a finite set of data items which is either empty or consists of a single item called the root and two disjoint binary trees called the left subtree and right subtree.
- In a binary tree the maximum degree of any node is at most 2.
- That means ,there may be a zero degree node (empty tree) or a one degree node and two degree node.



# Types of Tree data structure

Binary Search tree:

- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

**Properties**:

The properties that separate a binary search tree from a regular binary tree is

- All nodes of left subtree are less than the root node
- All nodes of right subtree are more than the root node
- Both subtrees of each node are also BSTs i.e. they have the above two properties

# Types of Tree data structure

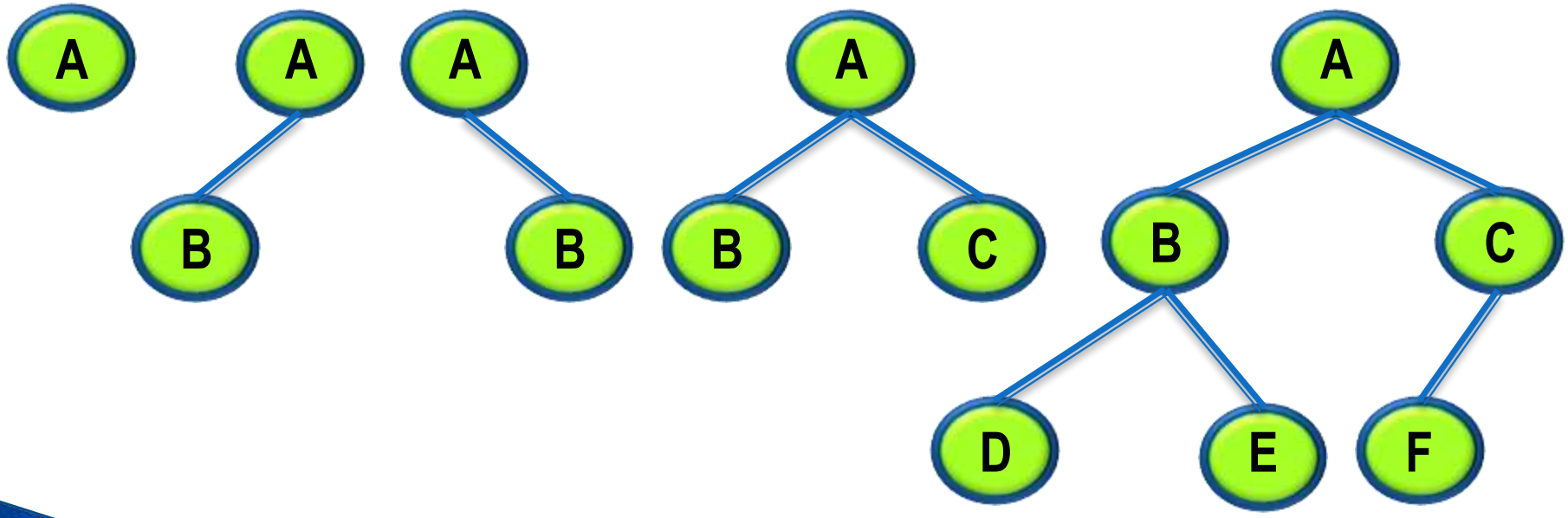
AVL tree:

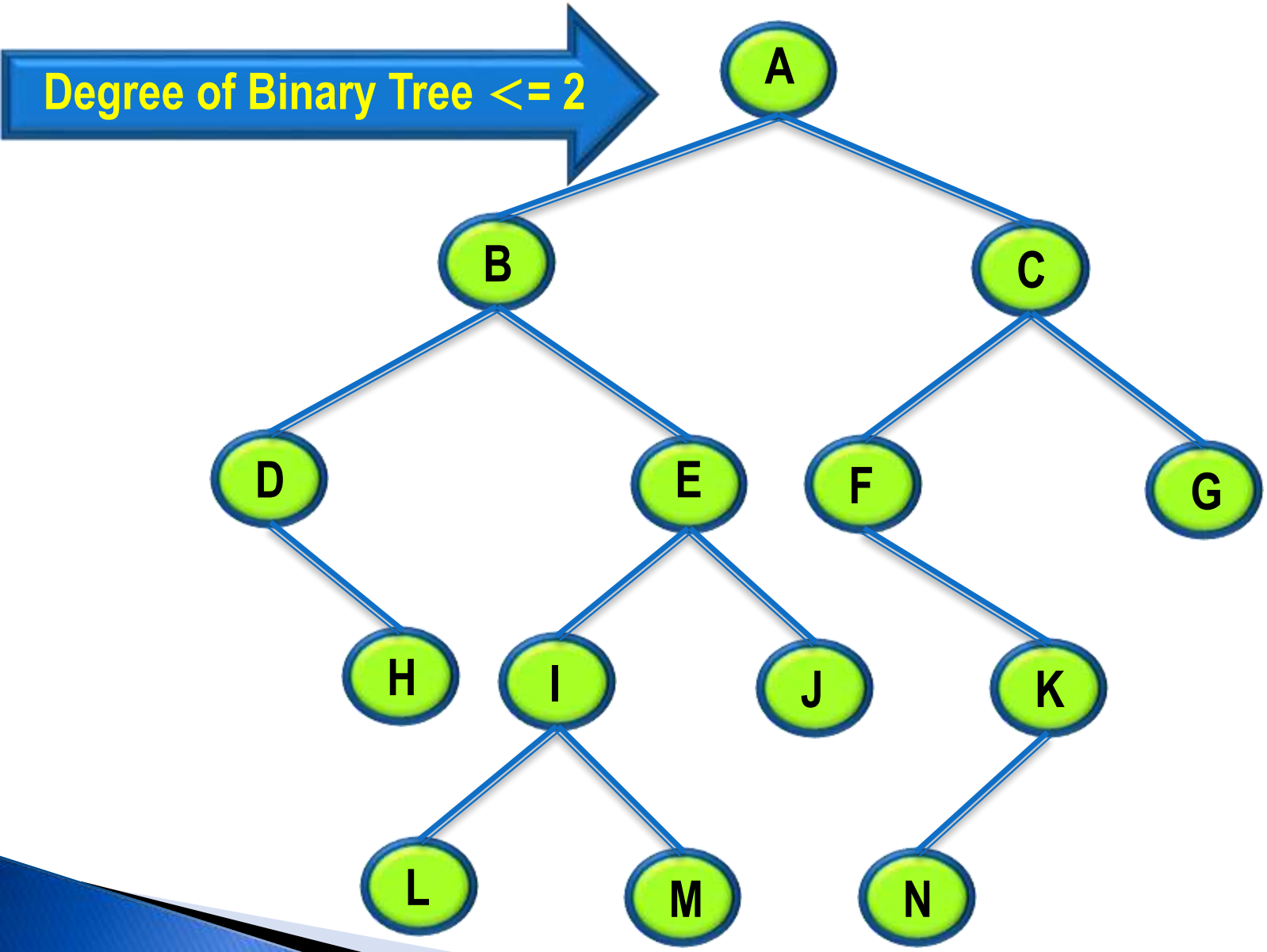
- It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree.
- AVL tree satisfies the property of the *binary tree* as well as of the *binary search tree*.
- It is a self-balancing binary search tree that was invented by *Adelson Velsky Lindas*. Here, self-balancing means that balancing the heights of left subtree and right subtree.
- This balancing is measured in terms of the *balancing factor*.

# Binary Tree

They are also known as 2-way trees

- A ***binary tree*** is a tree where each node has **no more than two** children.
- That is, a node in a binary tree can have a maximum of two child nodes.




Degree of Binary Tree <= 2
LEVEL
A
B
C
D
E
F
G
H
I
J
K
L
M
N
0
1
2
3
4

# Properties of A Binary Tree

The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$, $i \geq 1$. (if root level is 1)

If Root level is Zero: $2^{\ell}$

The maximum number of nodes in a binary tree of $k$ levels is $2^k-1$, $k \geq 1$.

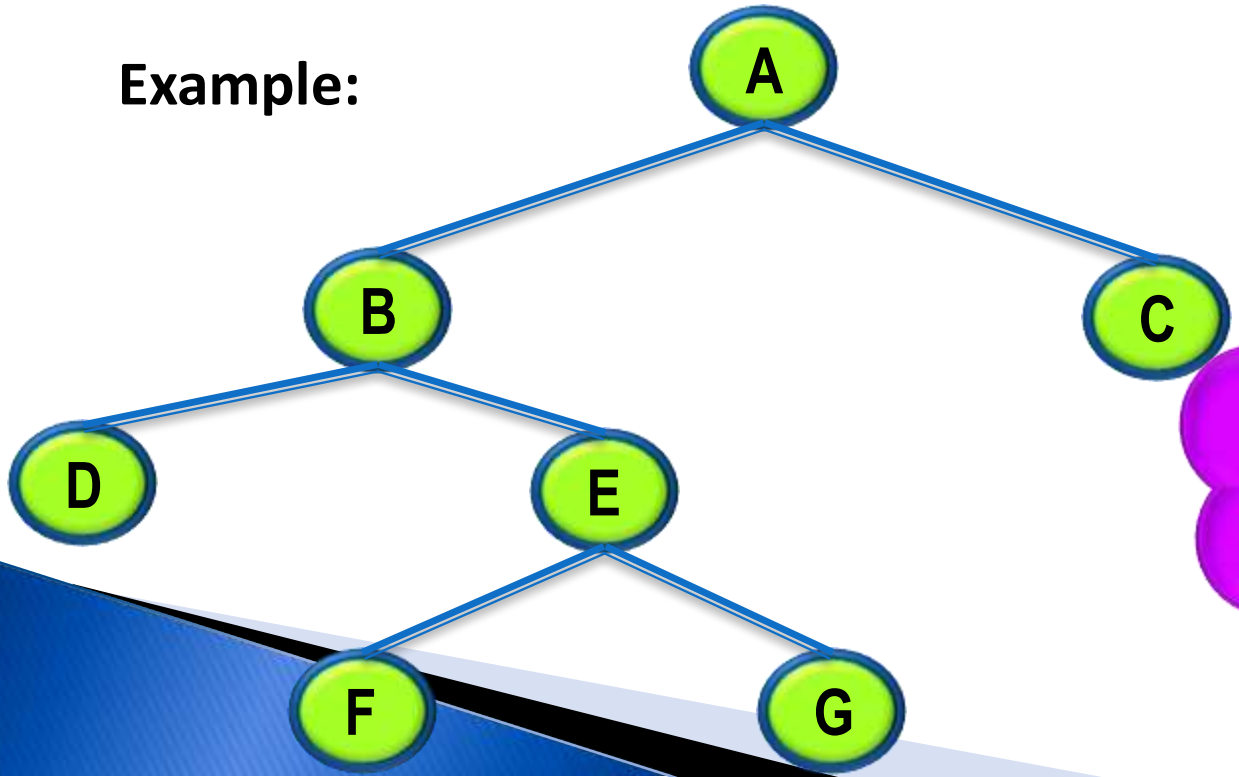
For any nonempty binary tree, T, if $n_0$ is the number of leaf nodes and $n_2$ is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

# Strictly Binary Trees

- If every non-terminal node in a binary tree consists of non-empty left- and right subtrees, then such a tree is called a ***strictly binary tree***.

**Each node in the tree has either no children or exactly two child nodes.**

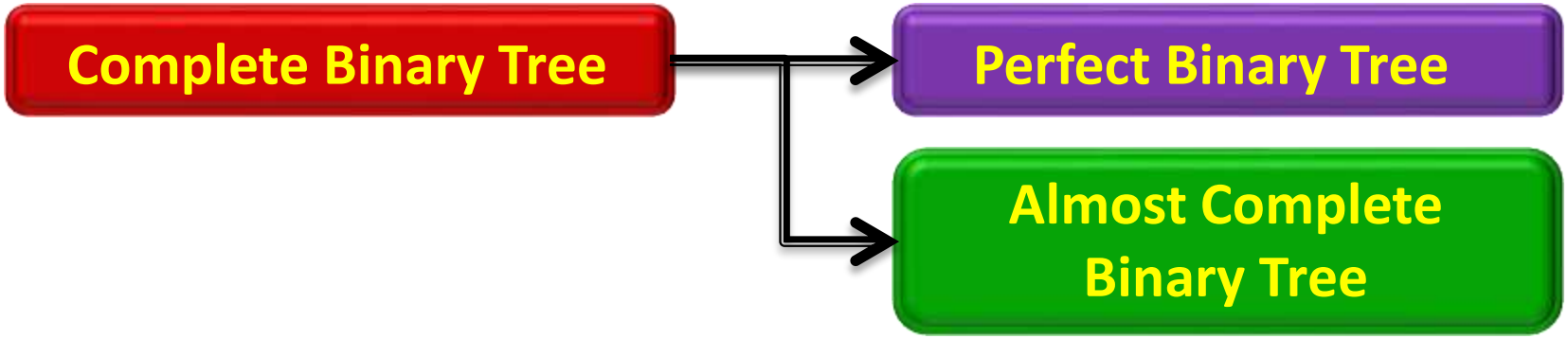
**Example:**




**Also known as Full Binary Tree**

# Complete Binary Trees

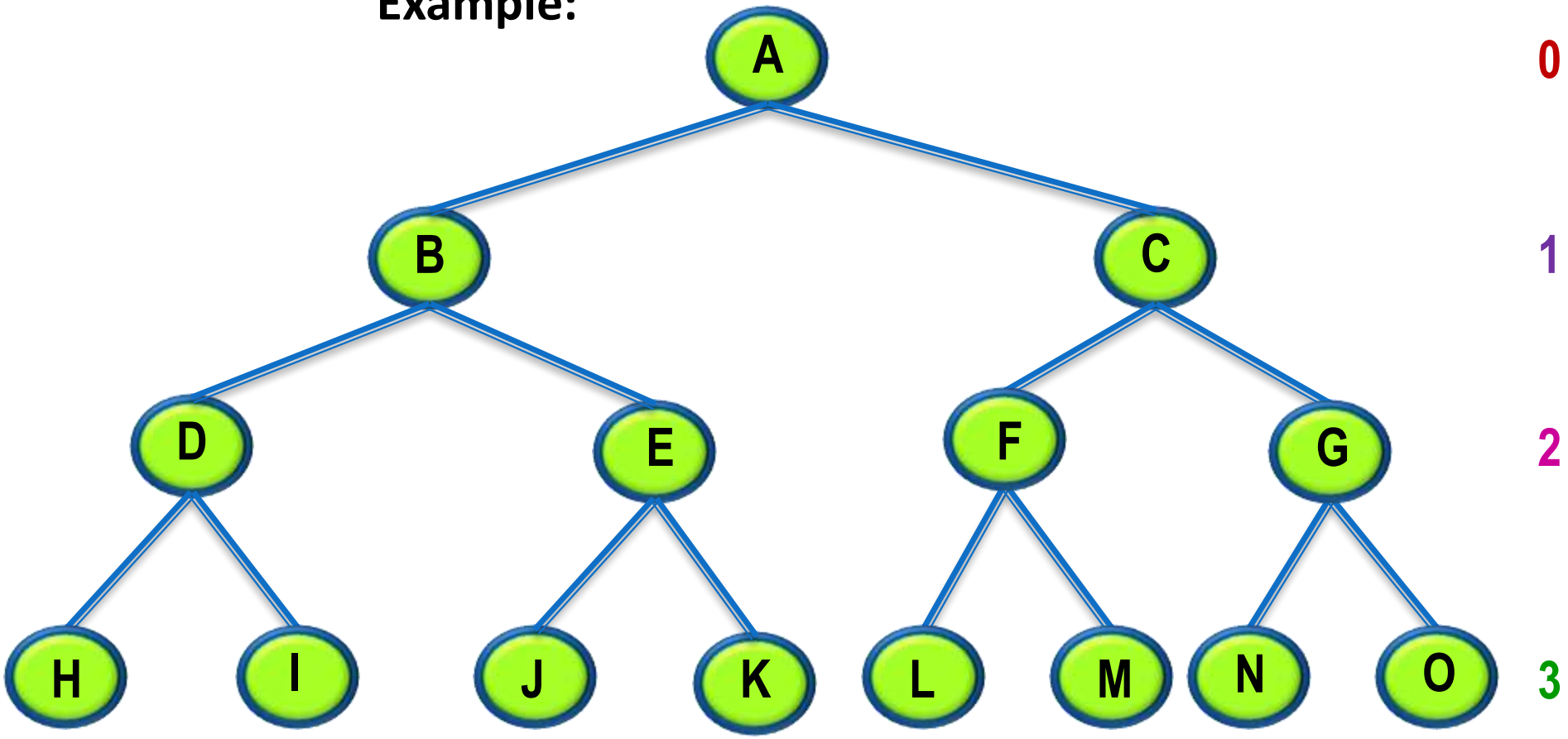
- A **Complete Binary Tree** is a binary tree in which every level, ***except possibly the last***, is completely filled, and all nodes are as far left as possible.




**A Perfect Binary Tree is a binary tree where every level is completely filled.**

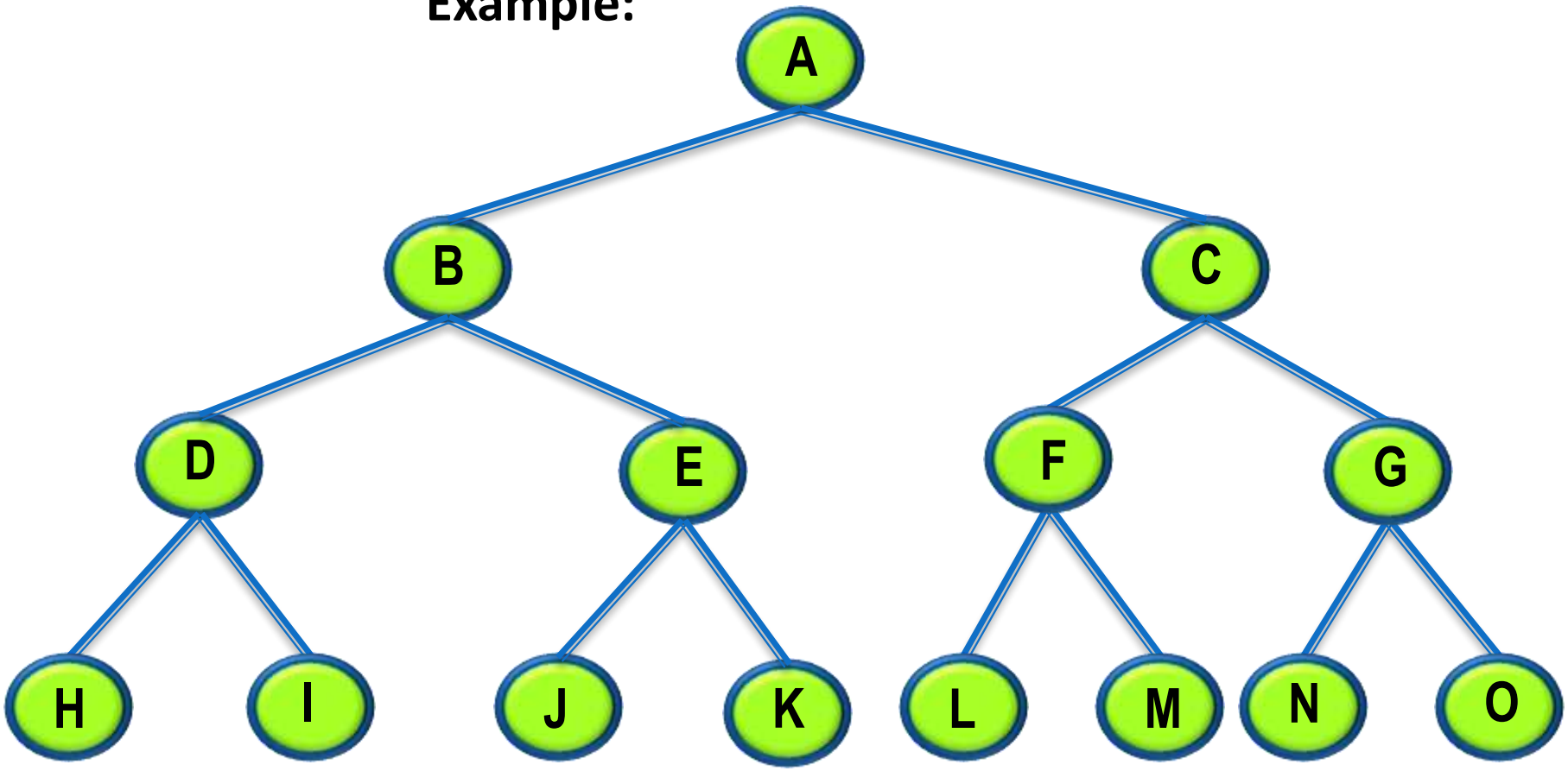
**An Almost Complete Binary Tree is a binary tree where all levels are filled expect the last level and the nodes in the last level are as far left as possible. It is also known as nearly complete binary tree.**

Example:




A perfect binary tree is both full and complete but not vice-versa

**Example:**

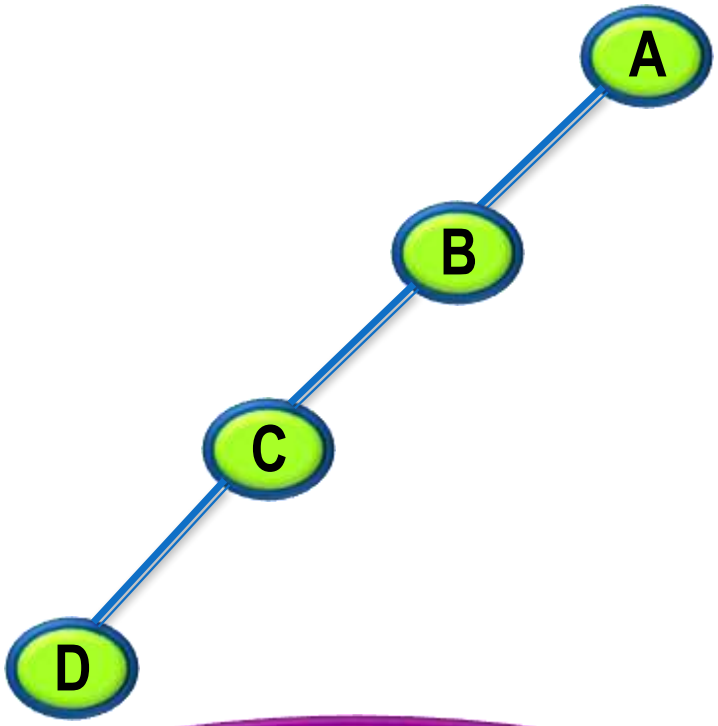


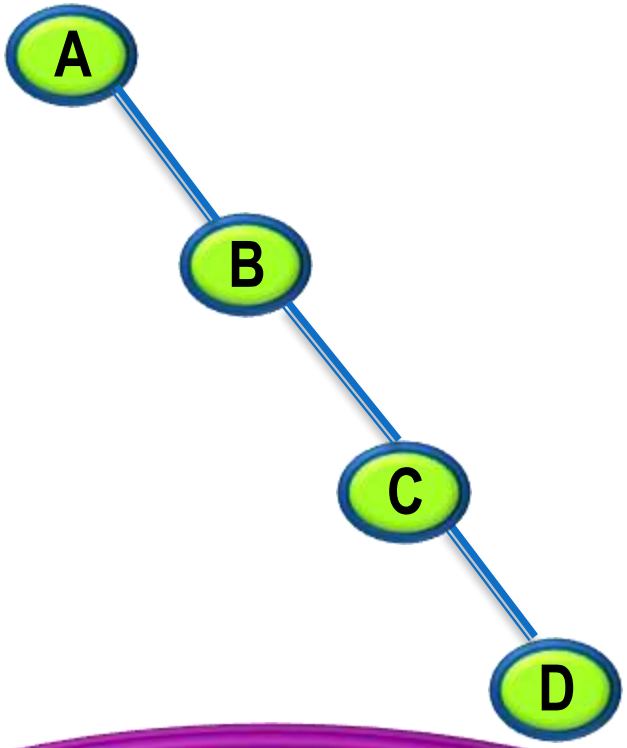

**An almost complete binary tree is not always a full binary tree**

**A complete binary tree with *n* vertices and height *H* satisfies:**

- $2^H \leq n < 2^{H+1}$
- $H \leq \log n < H + 1$
- $H = \text{floor}(\log n)$

# Skewed Binary Trees




A Binary Tree in which each node has only a *left* subtree

A Binary Tree in which each node has only a *right* subtree

# Test Your Brain!!!

What is the max number nodes at some level $\ell$ in a binary tree?

→ $2^{\ell}$

What is the total number of nodes N of a full tree with height h?

→ $2^{h} - 1$

What is the height h of a full tree with N nodes?

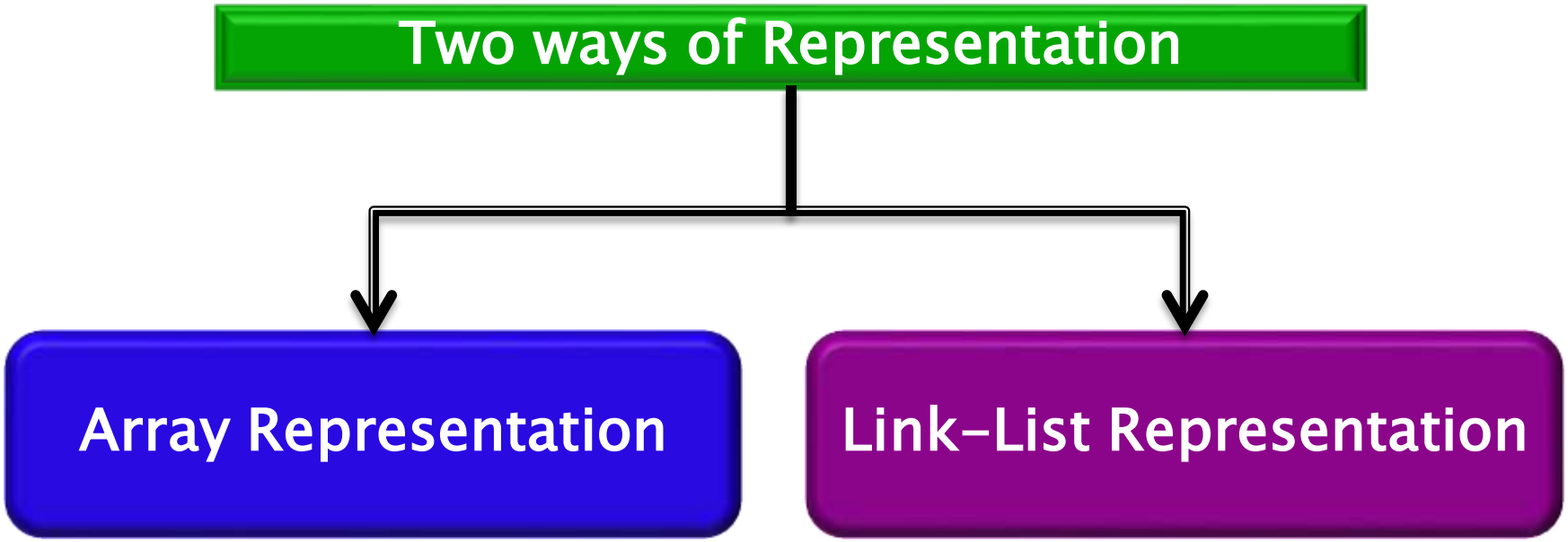
$2^{h} - 1 = N$

$\rightarrow 2^{h} = N + 1$

$\rightarrow h = \log(N + 1)$

$\rightarrow O(\log N)$

# Memory Representations of Binary Tree




**Array Representation is static in nature**

**Link-List Representation is dynamic in nature**

# Array Representation of Binary Tree

**If a complete binary with *n* nodes is represented sequentially, then for any node with index *i*, $0 \le i \le n-1$, we have,**

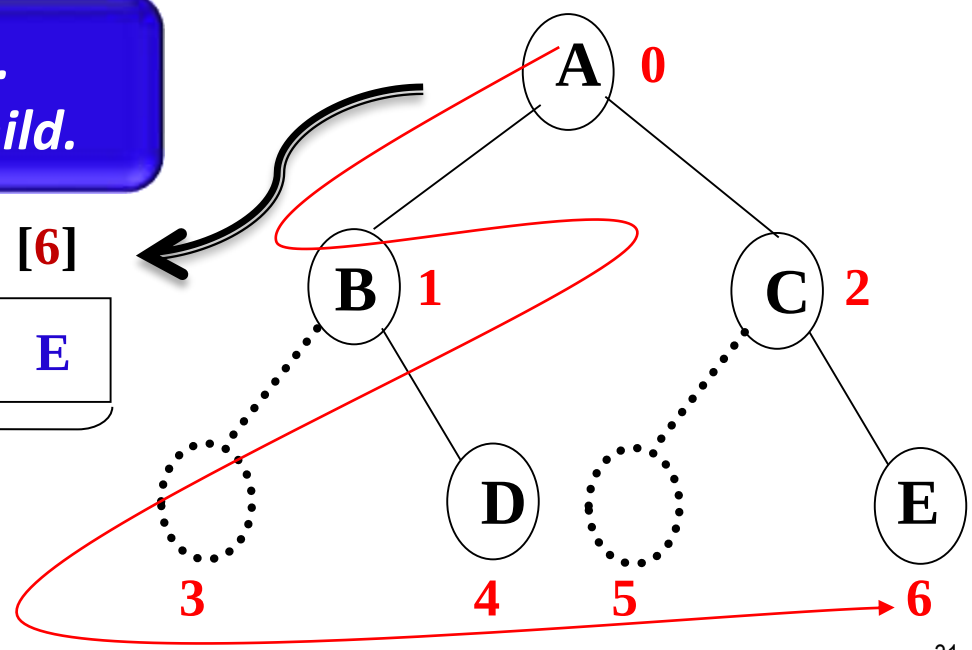
***parent(i)* is at $\lfloor (i-1)/2 \rfloor$ if $i \neq 0$.**
**If *i* = 0, then *i* is at the root and has no parent.**

***LeftChild(i) is at 2i+1 if $2i+1 \le n$.***
***If $2i+1 > n$, then i has no left child.***

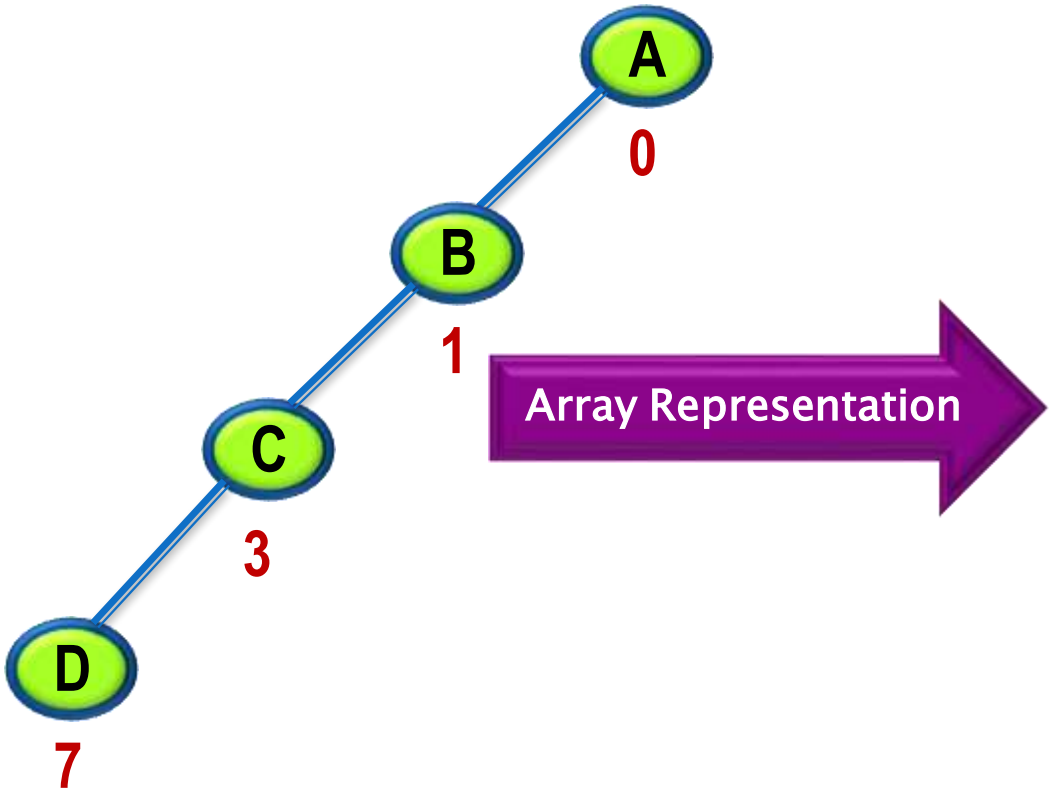
***RightChild(i) is at 2i+2 if $2i+2 \le n$.***
***If $2i+2 > n$, then i has no right child.***

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
|---|---|---|---|---|---|---|
| A | B | C | — | D | — | E |

Level 0 Level 1 Level 2



# Example




| | |
|---|---|
| [0] | A |
| [1] | B |
| [2] | --- |
| [3] | C |
| [4] | --- |
| [5] | --- |
| [6] | --- |
| [7] | D |
| [8] | --- |

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

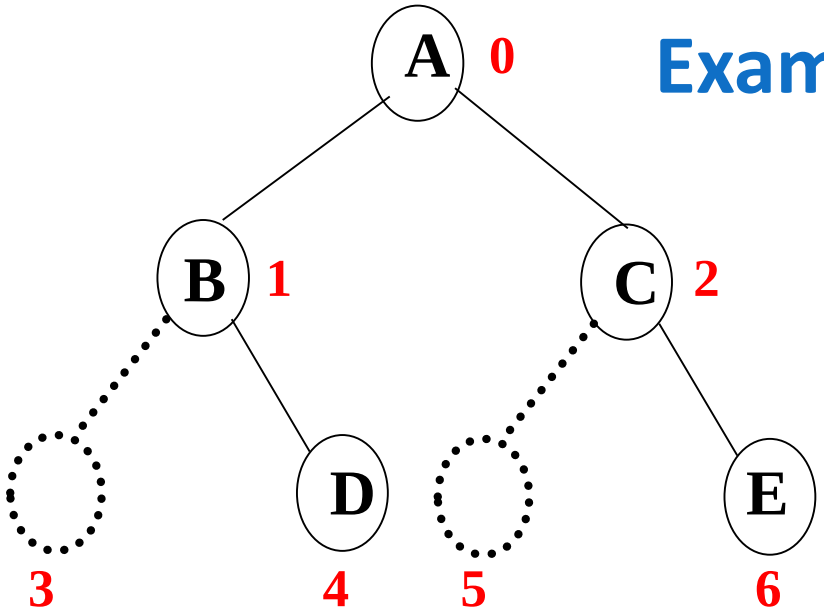
# Link List Representation of Binary Tree

- Here a single tree node consist of three fields such as,
  - Data: It holds the value of the node
  - Left child: It is a link field which contains the address of its left child
  - Right child: It contains the address of its right child

Structure of the Node

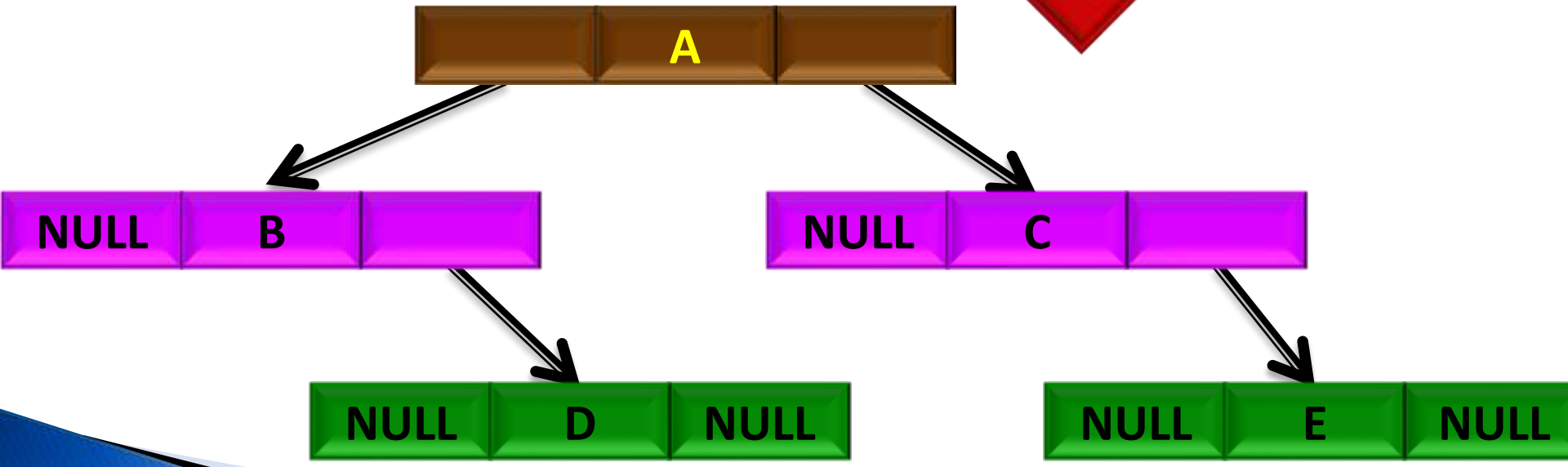
```
struct  treenode
{
    char data ;
    struct  treenode *lchild ;
    struct  treenode *rchild ;
} ;
```

# Example




The field value is NULL when child node is absent

Link List Representation



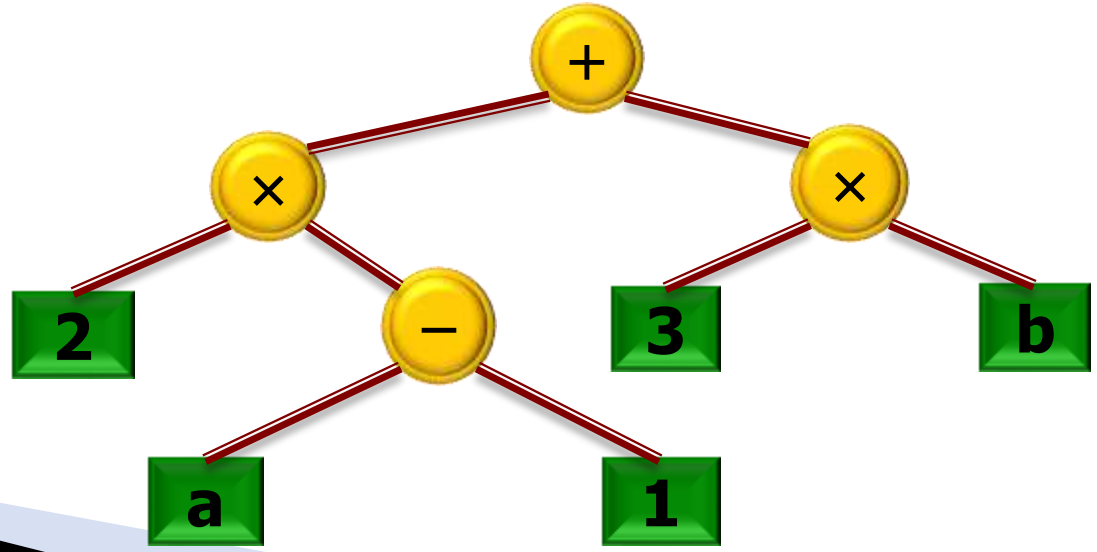
# Expression Trees

**Expression Tree is a binary tree associated with an arithmetic expression, here**

- **internal nodes are operators**
- **external nodes are operands**

**Example:**

**Arithmetic expression tree for the expression**
**$(2 \times (a - 1) + (3 \times b))$**



# Traversing Binary Trees

**Traversing a binary tree is a way of visiting each node exactly ones in a systematic manner**

**There are three standard ways of traversing a binary tree T with a root R. They are as follows:**

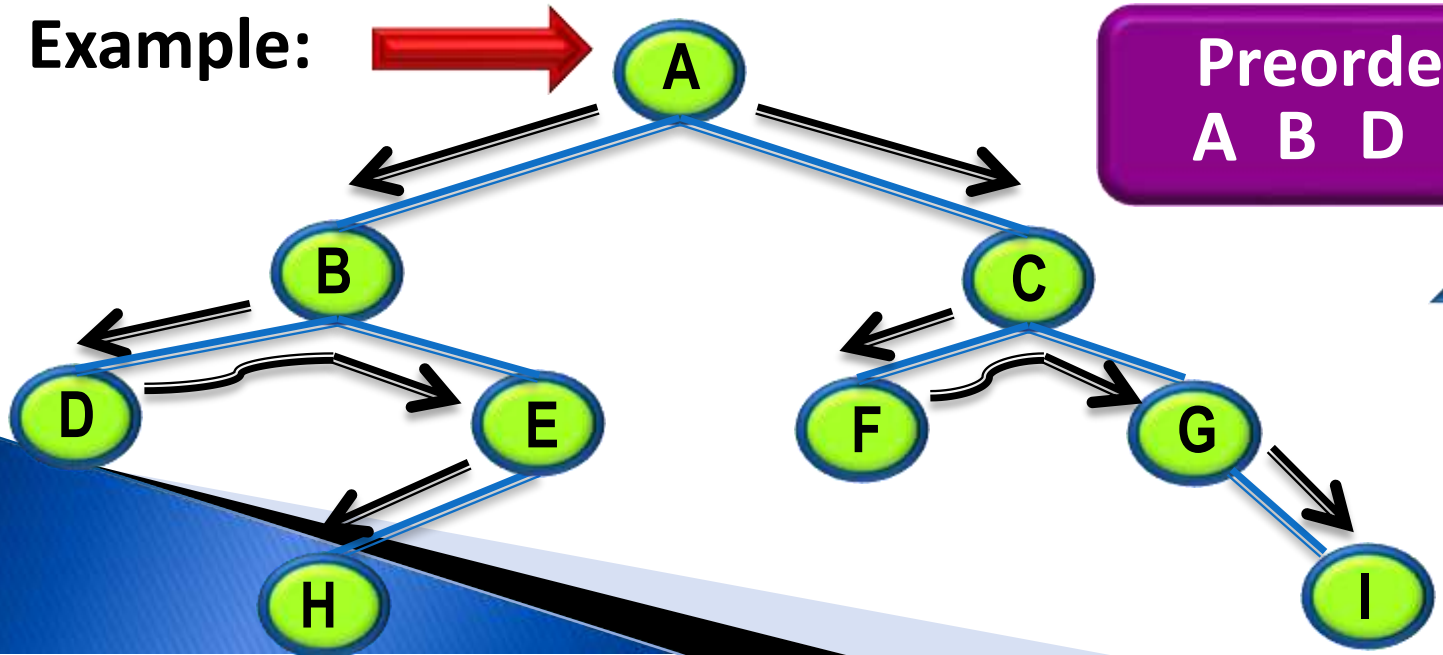
- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

# Recursive Traversal Algorithms

## Preorder Traversal

**Preorder(T)**
**Step 1: Process the root node (say R) → Visit**
**Step 2: Traverse the left subtree in Preorder**
**Step 3: Traverse the right subtree in Preorder**

**Example:**



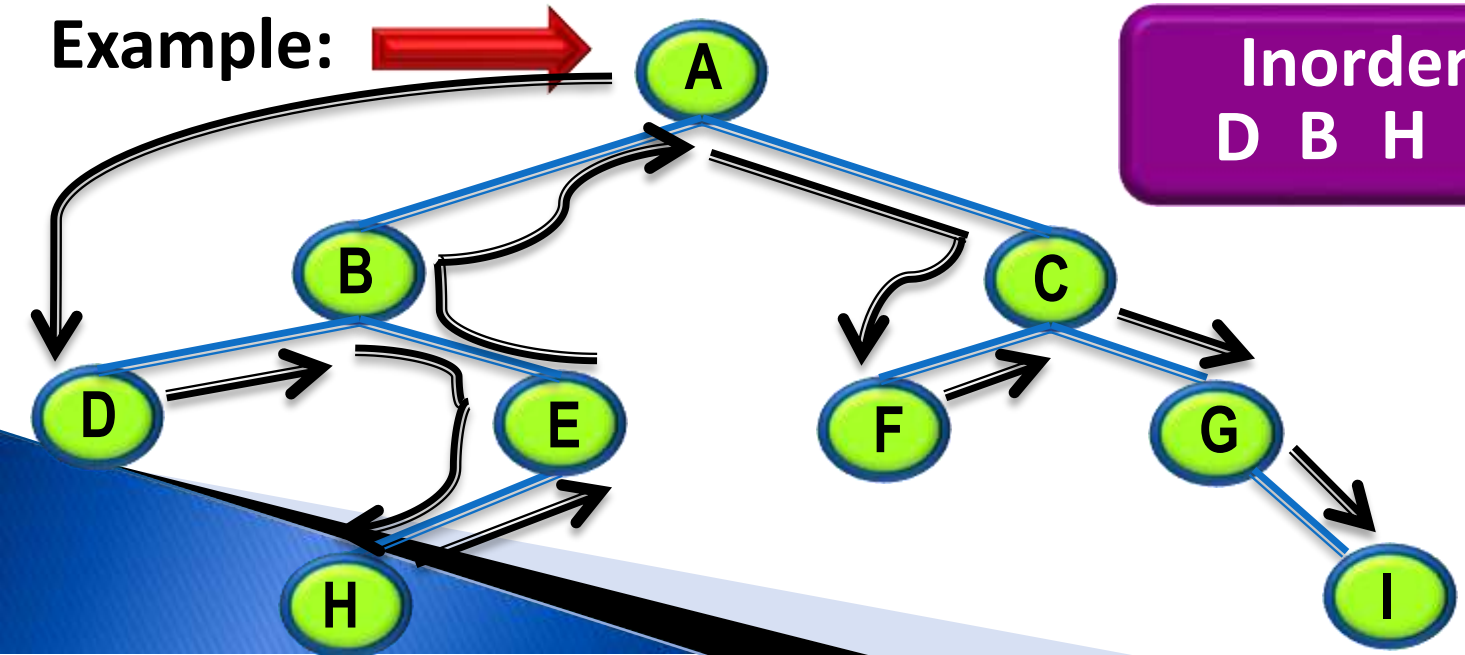

**Preorder Sequence:**
**A B D E H C F G I**

# Recursive Traversal Algorithms

## Inorder Traversal

**Inorder(T)**
**Step 1: Traverse the left subtree in Inorder**
**Step 2: Process the root node (say R) → Visit**
**Step 3: Traverse the right subtree in Inorder**

**Example:**



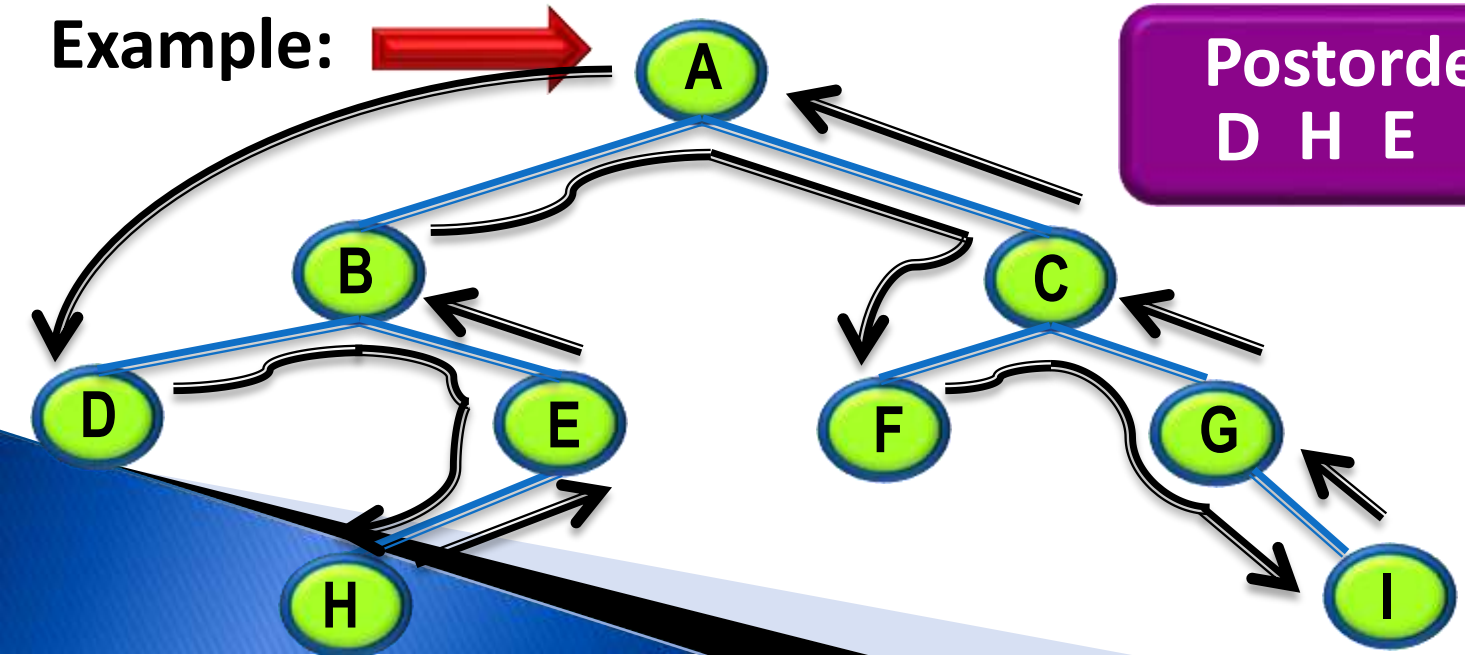

**Inorder Sequence:**
**D B H E A F C G I**

# Recursive Traversal Algorithms

## Postorder Traversal

**Postorder(T)**
**Step 1: Traverse the left subtree in Postorder**
**Step 2: Traverse the right subtree in Postorder**
**Step 3: Process the root node (say R) → Visit**

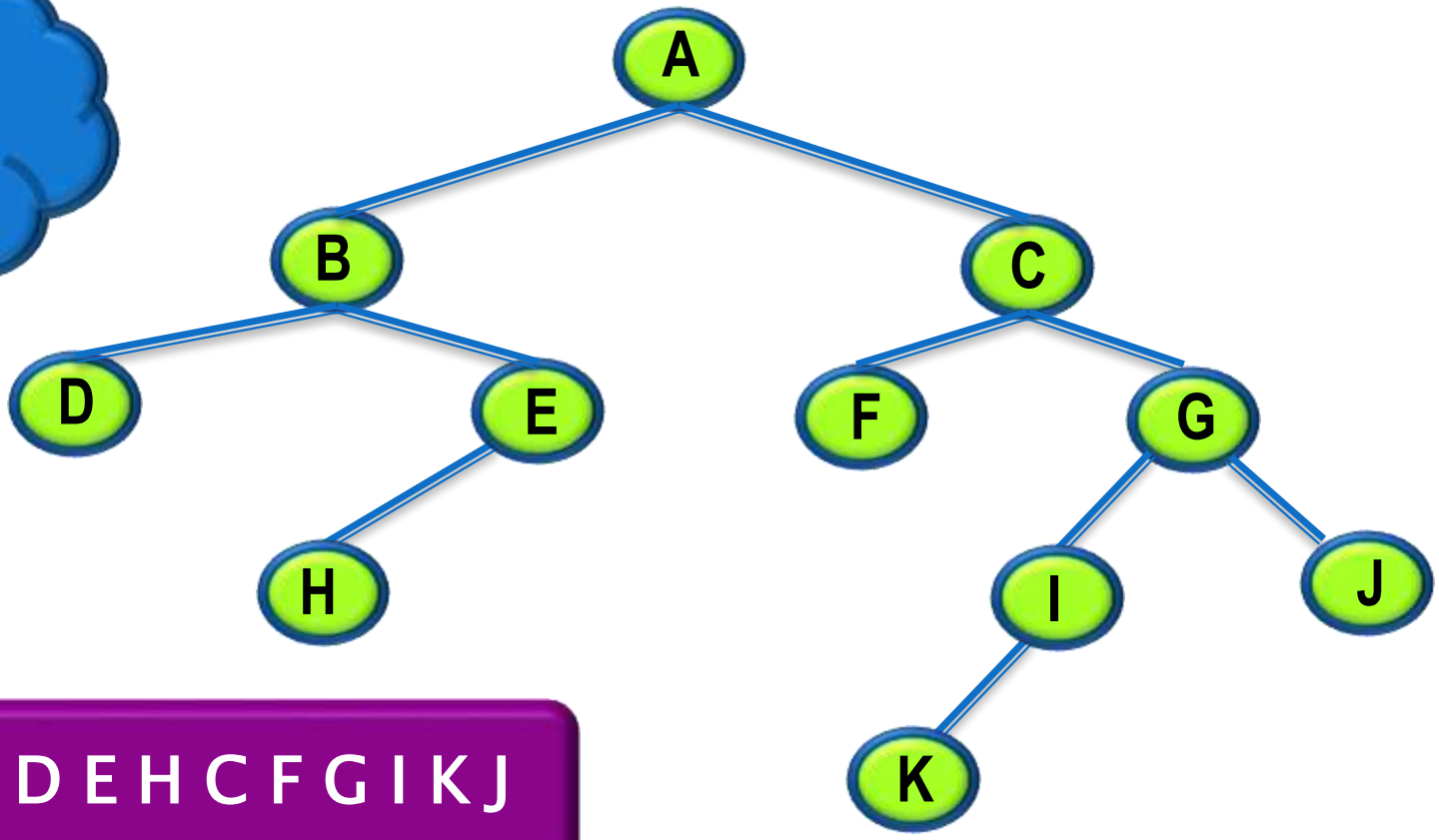
**Example:**




**Postorder Sequence:**
**D H E B F I G C A**

# Traverse the Binary Tree

Postorder
???




Preorder: A B D E H C F G I K J

Inorder: D B H E A F C K I G J

Postorder: D H E B F K I J G C A

**In Preorder Traversal Sequence:**
**The root is always present at the beginning (followed by the left & then the right subtree nodes)**

Preorder:
A B D E H C F G I K J

Root

Left Subtree

Right Subtree

**In Inorder Traversal Sequence:**
**The root is always present at the middle (following the left & followed by the right subtree nodes)**

Inorder:

D B H E A F C K I G J

Left Subtree | Root | Right Subtree

**In Postorder Traversal Sequence:**
**The root is always present at the end (starting with the left & then the right subtree nodes)**

Postorder:

D H E B F K I J G C A

Left Subtree

Right Subtree

Root

# To create a new node

```
/*To create a new node*/
struct node* createNode(int val)
{
    struct node* Node = (struct node*)malloc(sizeof(st
   ruct node));
    Node->element = val;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}
```

# Binary tree in preorder

```
/*function to traverse the nodes of binary tree in pre
  order*/
void traversePreorder(struct node* root)
{
    if (root == NULL)
        return;
    printf(" %d ", root->element);
    traversePreorder(root->left);
    traversePreorder(root->right);
}
```

Or

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

# Binary tree in Inorder

```
/*function to traverse the nodes of binary tree in Inor
  der*/
void traverseInorder(struct node* root)
{
    if (root == NULL)
        return;
    traverseInorder(root->left);
    printf(" %d ", root->element);
    traverseInorder(root->right);
}
```

or

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

# Binary tree in postorder*

```
/*function to traverse the nodes of binary tree in pos
  torder*/
void traversePostorder(struct node* root)
{
    if (root == NULL)
        return;
    traversePostorder(root->left);
    traversePostorder(root->right);
    printf(" %d ", root->element);
}
```

**Or**

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

# Binary Tree Traversal...

```
#include<stdio.h>
struct node{
    int data;
    struct node *left;
    struct node *right;
};
typedef struct node NODE;
NODE *root=NULL;
NODE *insert(NODE *tree, int num);
void preorder(NODE *tree);
void inorder(NODE *tree);
void postorder(NODE *tree);
int count=1;
void main(){
    int choice, num;
    do{
        printf("\n 1. Insert \n 2. Preorder \n 3. Inorder \n 4. Postorder \n 5. Exit");
        printf("\n Enter choice:");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                printf("\n Enter item:");
                scanf("%d",&num);
                root=insert(root, num);
                break;
            case 2:
                printf("\n Preorder Traversing \n");
                preorder(root);
                break;
            case 3:
                printf("\n Inorder Traversing \n");
                inorder(root);
                break;
            case 4:
                printf("\n Postorder Traversing \n");
                postorder(root);
                break;
            case 5:
                exit(0);
            default:
                printf("\n Wrong Choice!!!");
        }
    }while(choice!=5);
}
```

# Binary Tree Traversal...

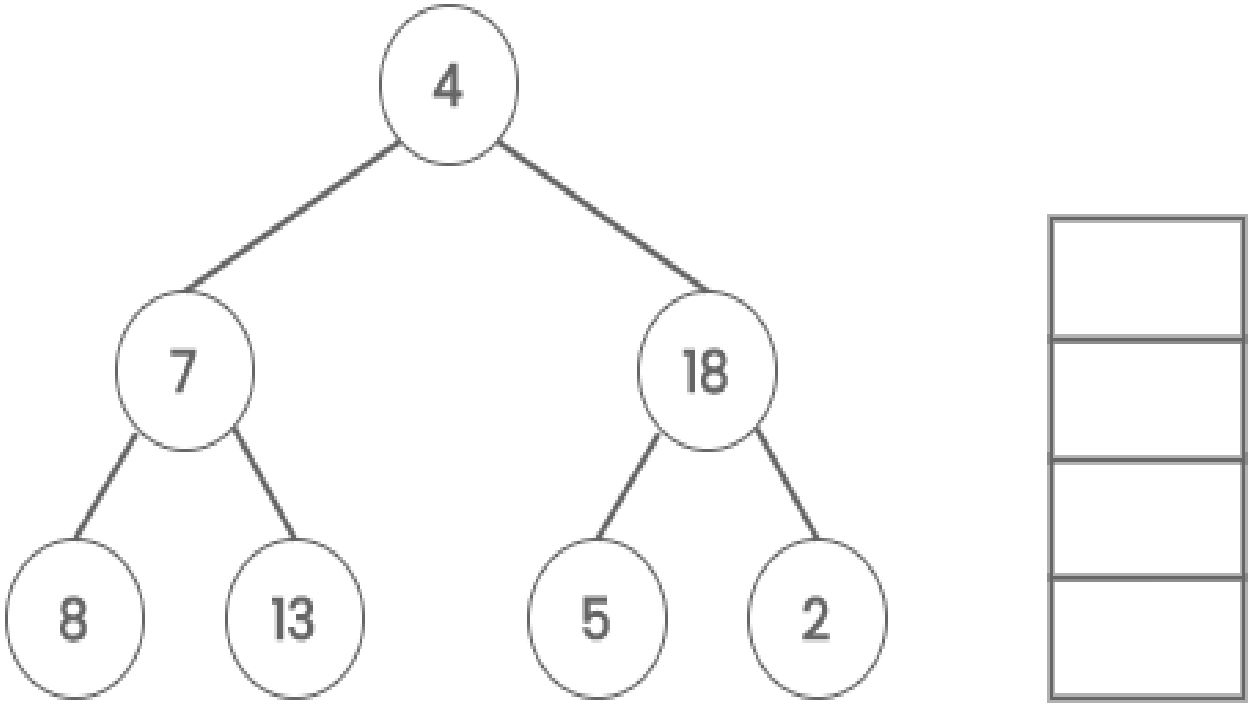
```
NODE *insert(NODE *P, int num){
    if(P==NULL){
        P=(NODE *)malloc(sizeof(NODE));
        P->left=NULL;
        P->right=NULL;
        P->data=num;
        count++;
    }
    else if(count%2==0)
        P->left=insert(P->left,num);
    else
        P->right=insert(P->right, num);
    return (P);
}

void preorder(NODE *P){
    if(P!=NULL){
        printf("%d \n", P->data);
        preorder(P->left);
        preorder(P->right);
    }
}

void inorder(NODE *P){
    if(P!=NULL){
        inorder(P->left);
        printf("%d \n", P->data);
        inorder(P->right);
    }
}
void postorder(NODE *P){
    if(P!=NULL){
        postorder(P->left);
        postorder(P->right);
        printf("%d \n", P->data);
    }
}
```

# Inorder Tree Traversal without Recursion

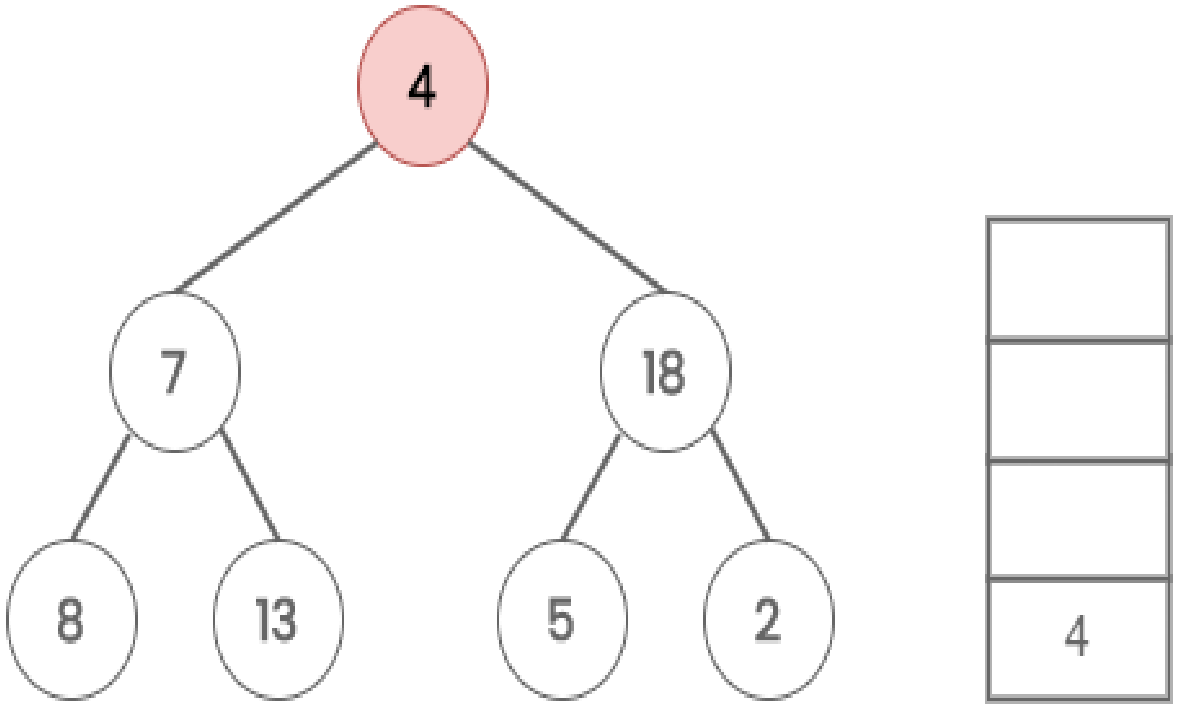
```
inorder (root)
BEGIN
1.   curr = root;
2.   while(curr != NULL or Stack is nonempty) DO
3.        if(curr  != NULL) Then
4.             push(curr)
5.             curr = curr->left
6.        else
7.             curr = pop()
8.             print(curr->data)
9.             curr = curr->right
END
```

# Inorder Tree Traversal without Recursion–Example

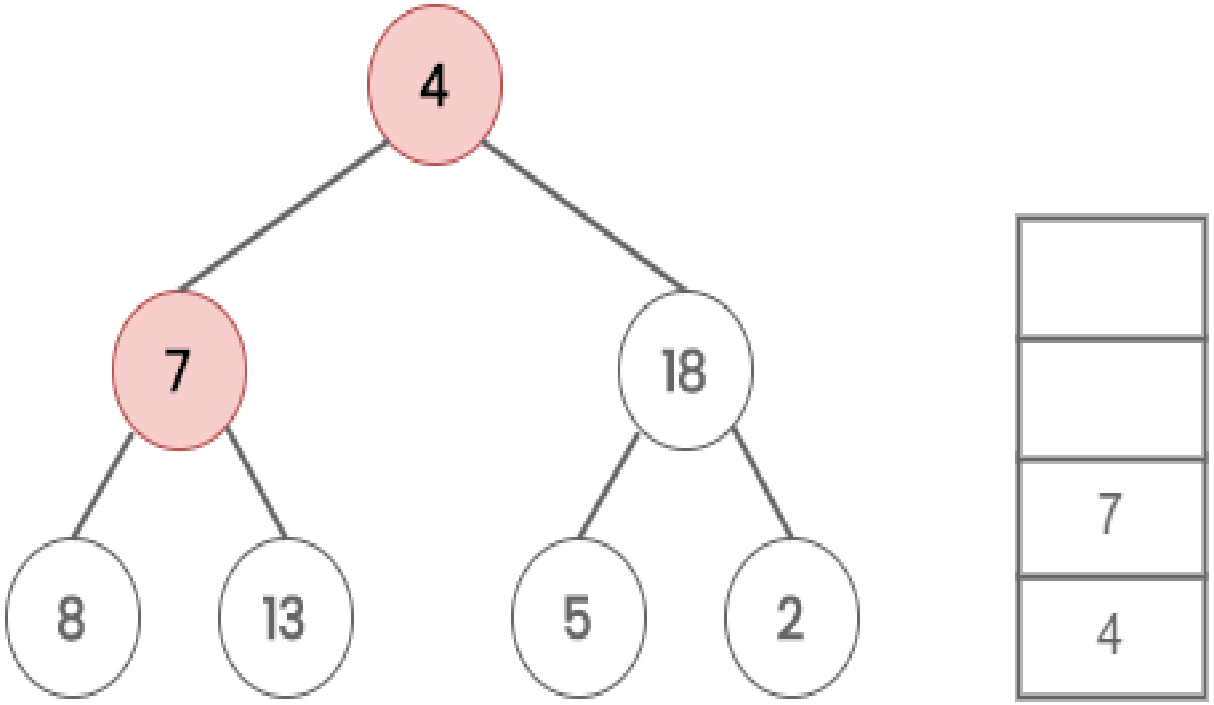
Consider the following tree.



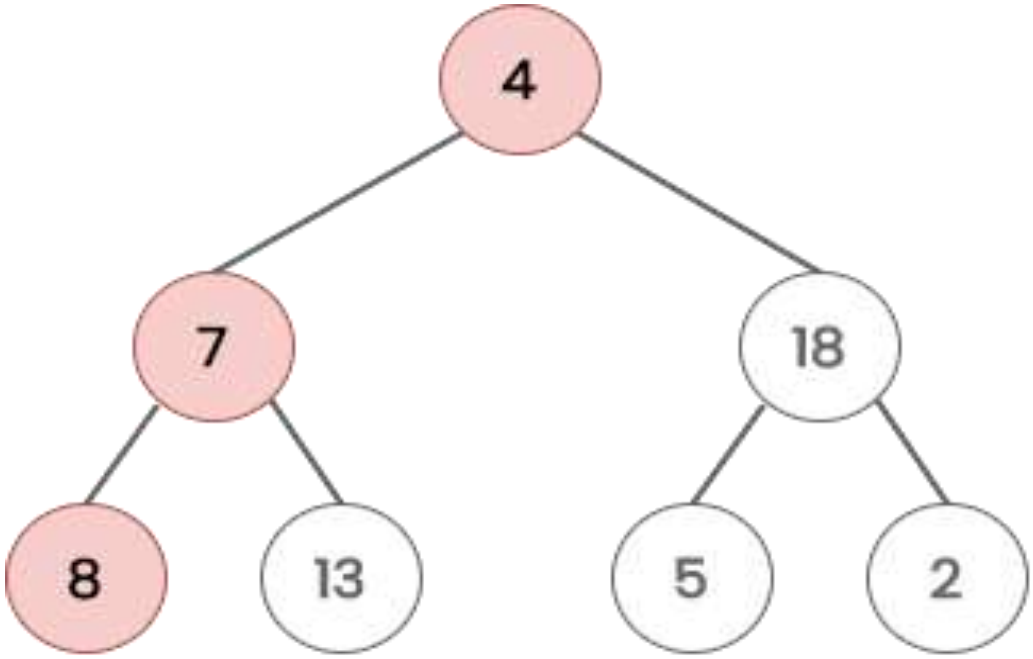
# Inorder Tree Traversal without Recursion–Example

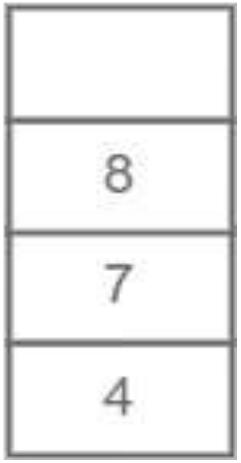


# Inorder Tree Traversal without Recursion-Example

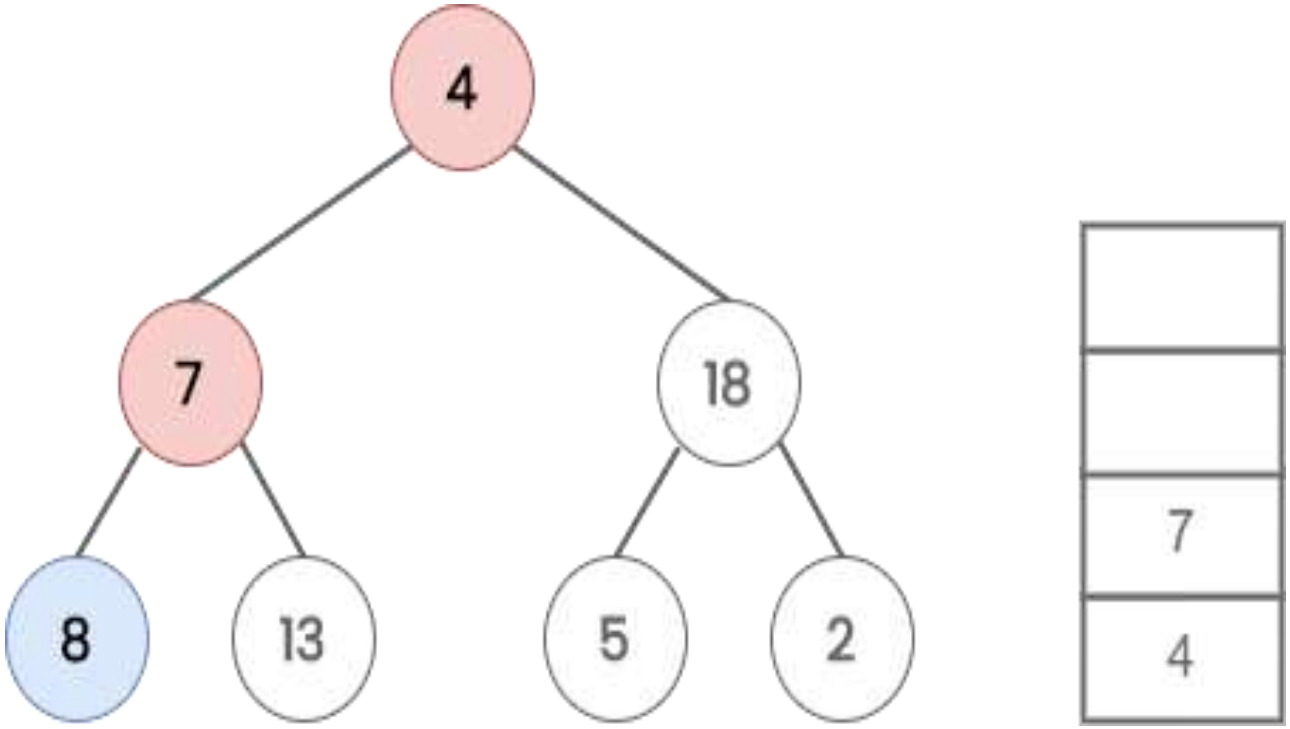


# Inorder Tree Traversal without Recursion-Example

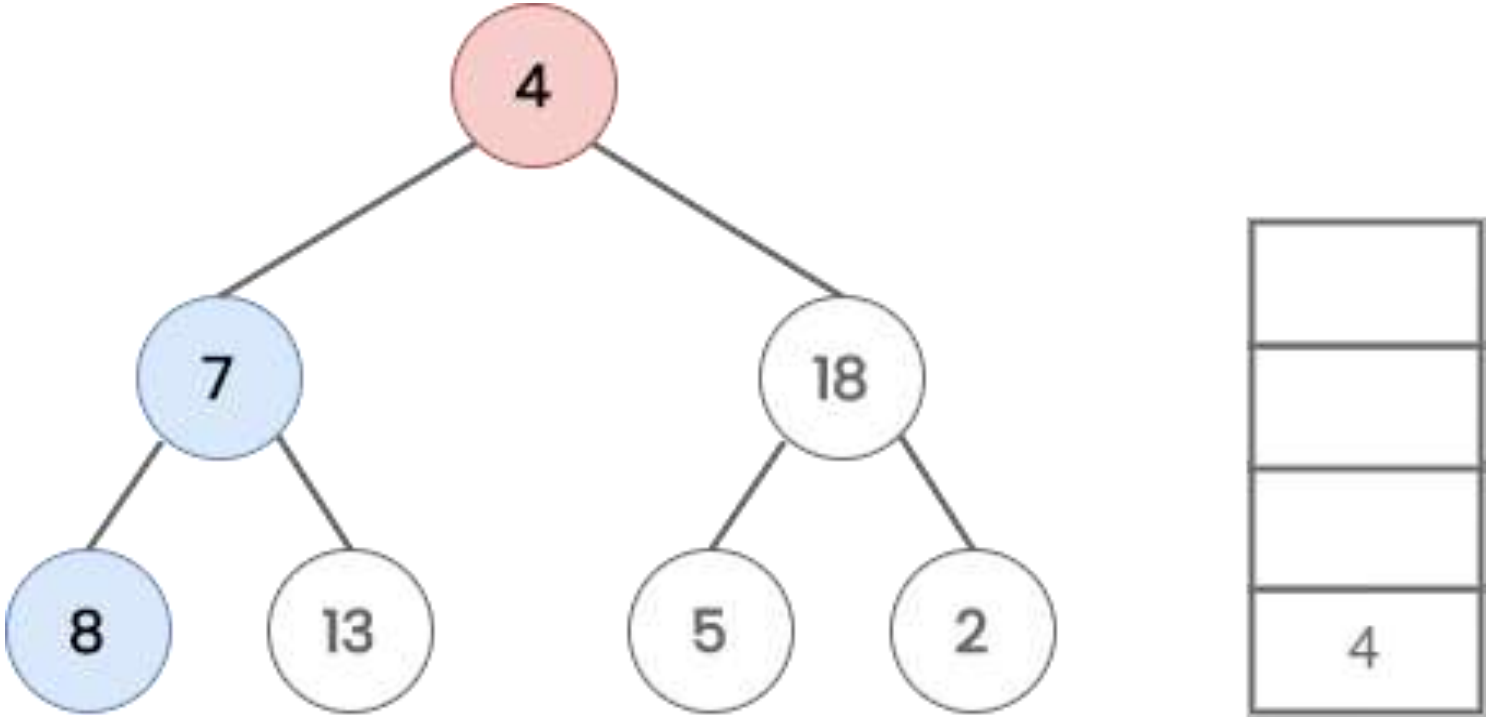


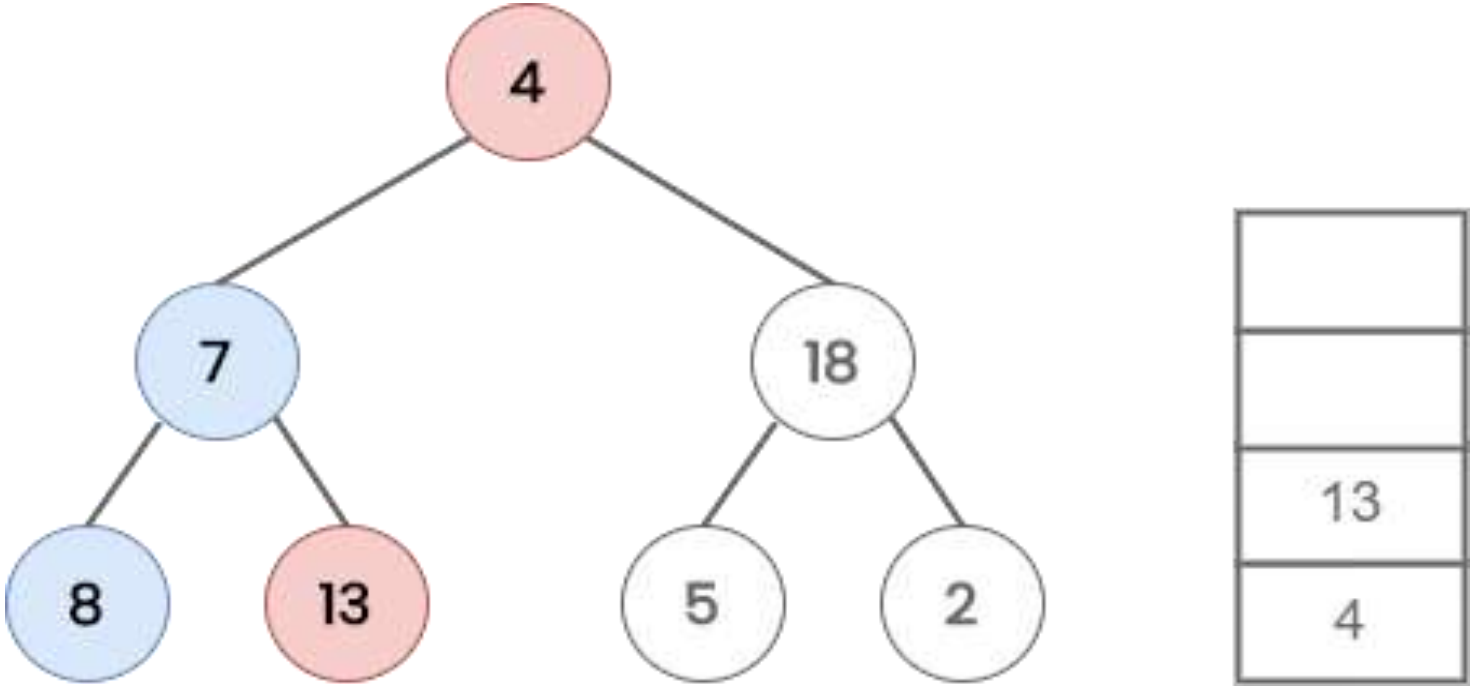
# Inorder Tree Traversal without Recursion-Example



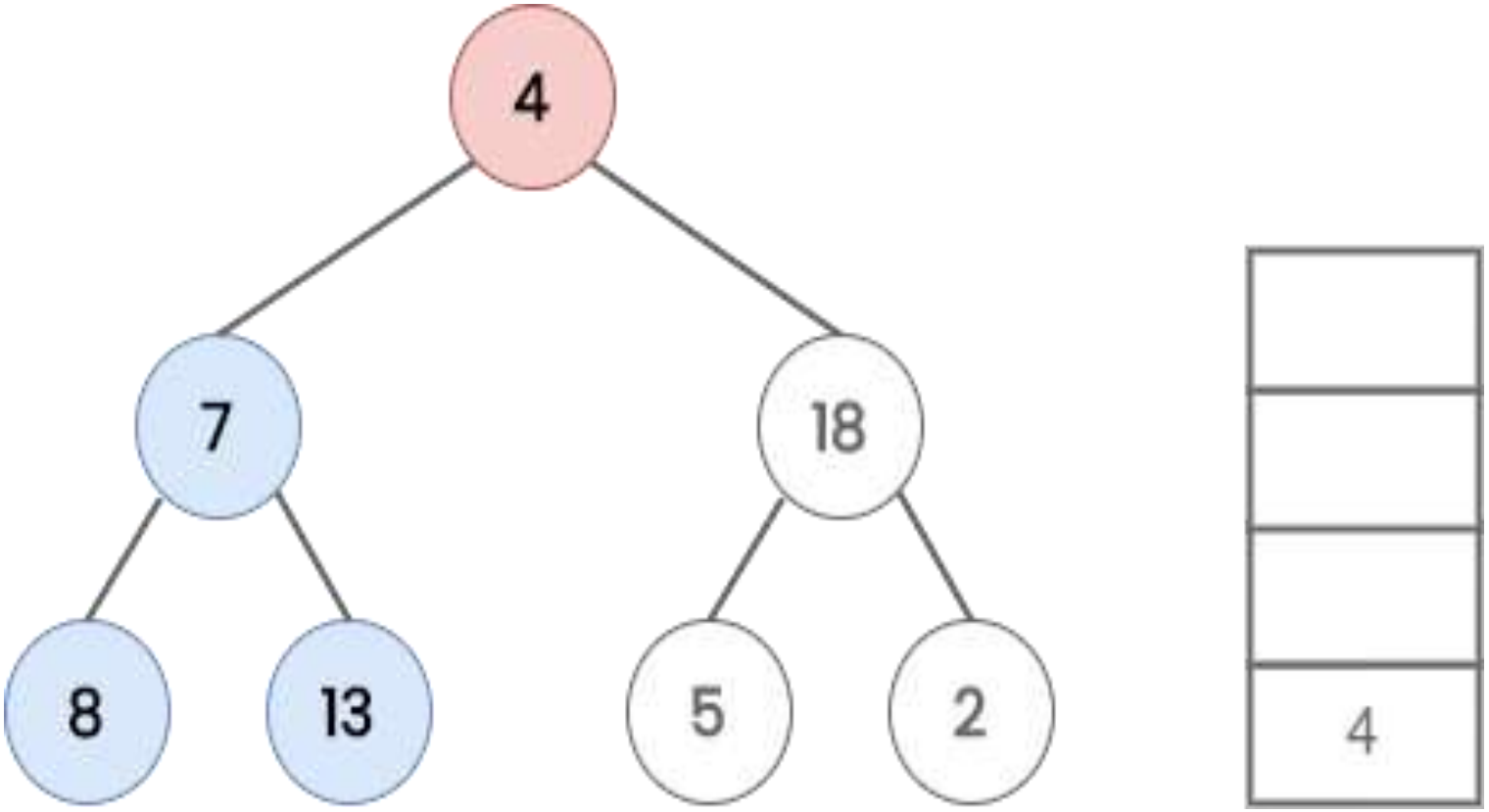
# Inorder Tree Traversal without Recursion-Example



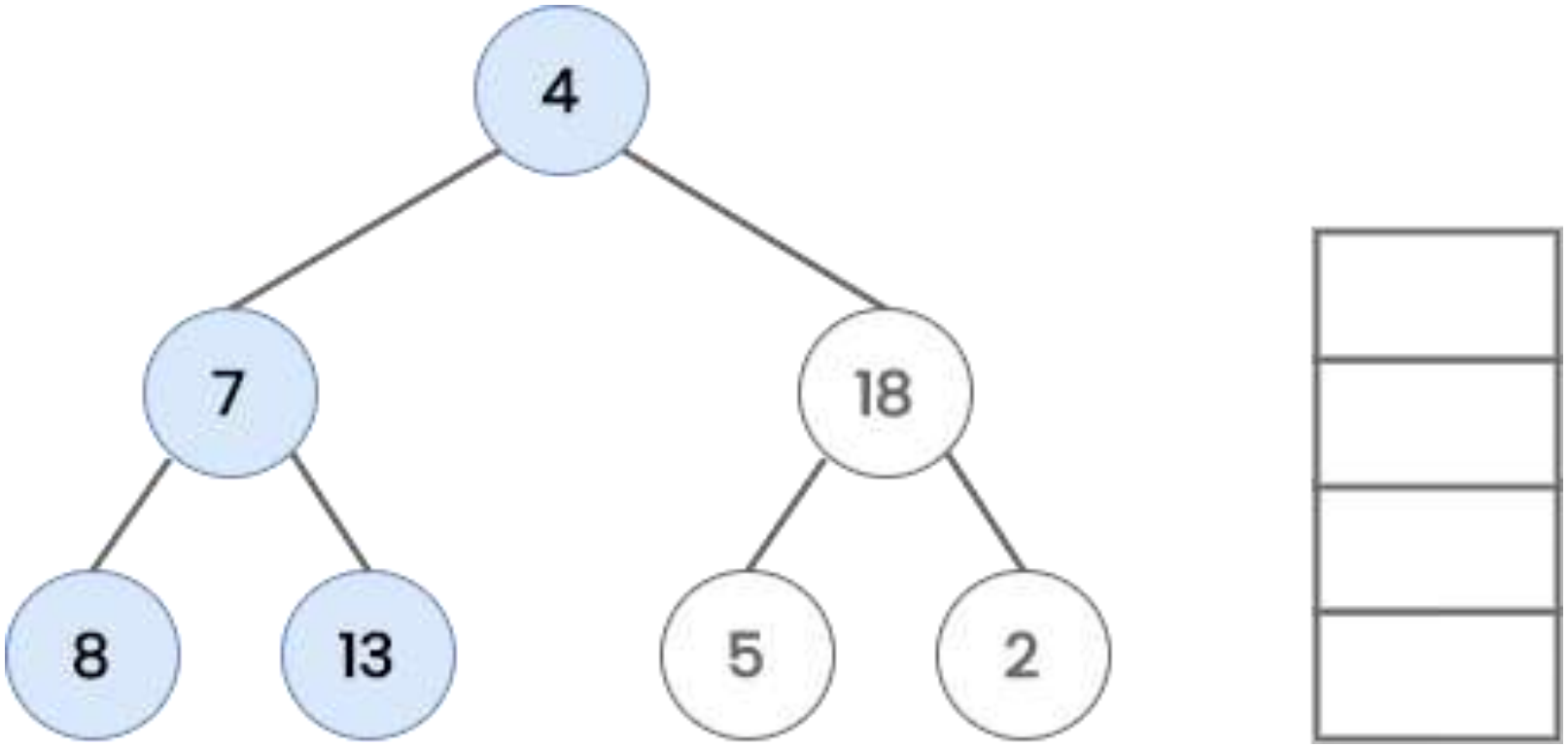
# Inorder Tree Traversal without Recursion-Example



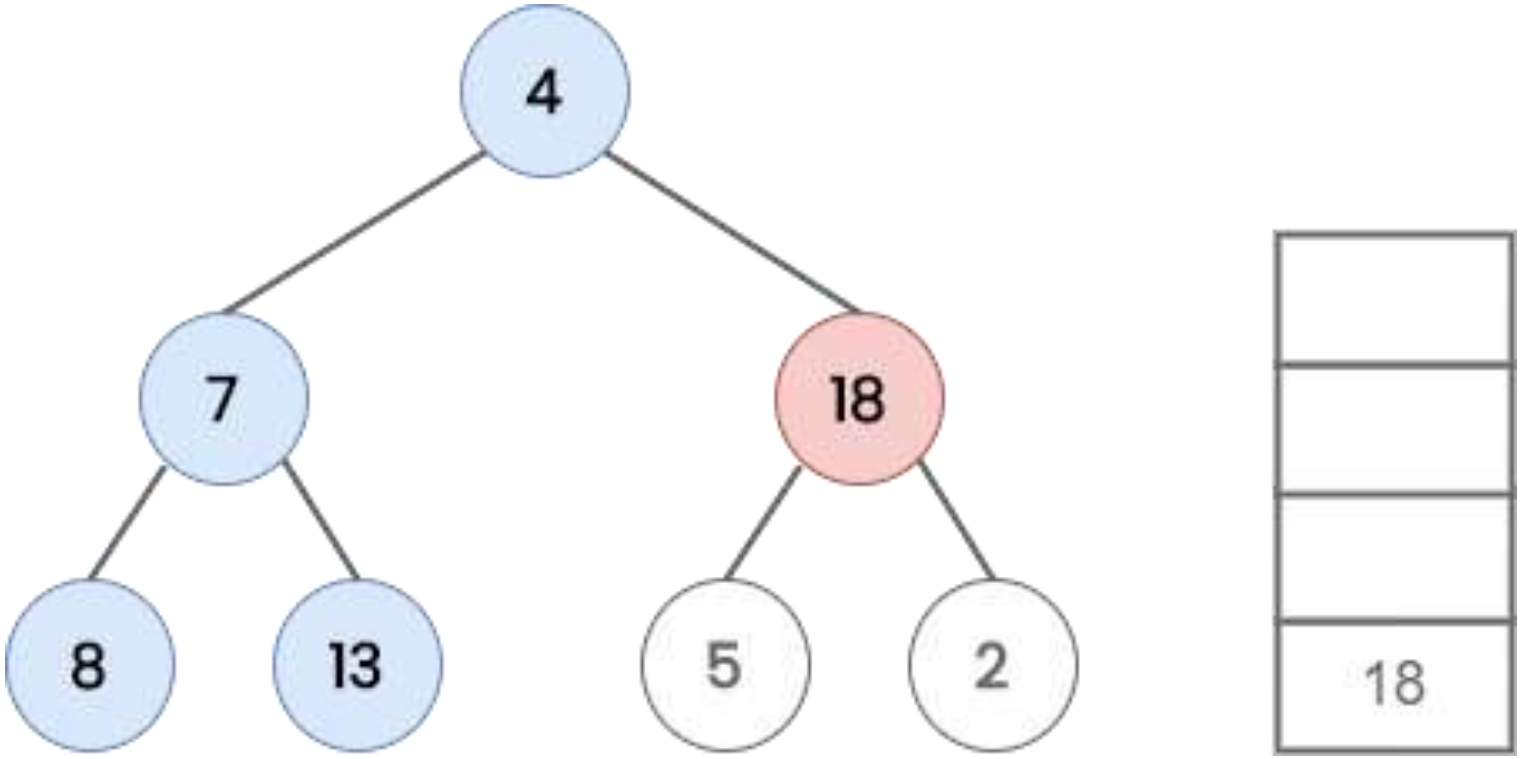
# Inorder Tree Traversal without Recursion-Example



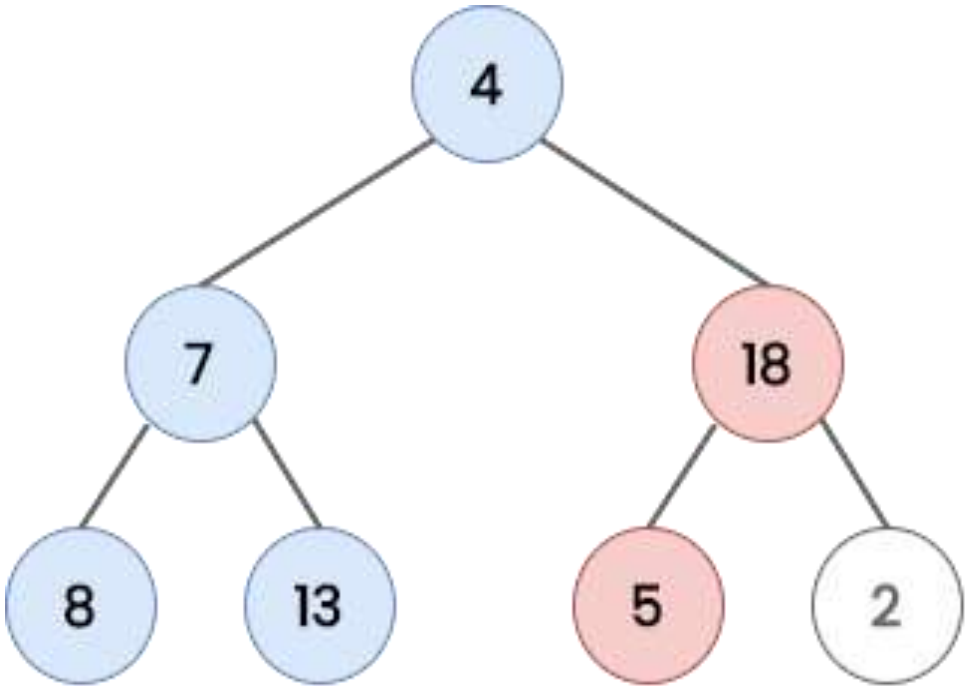
# Inorder Tree Traversal without Recursion-Example



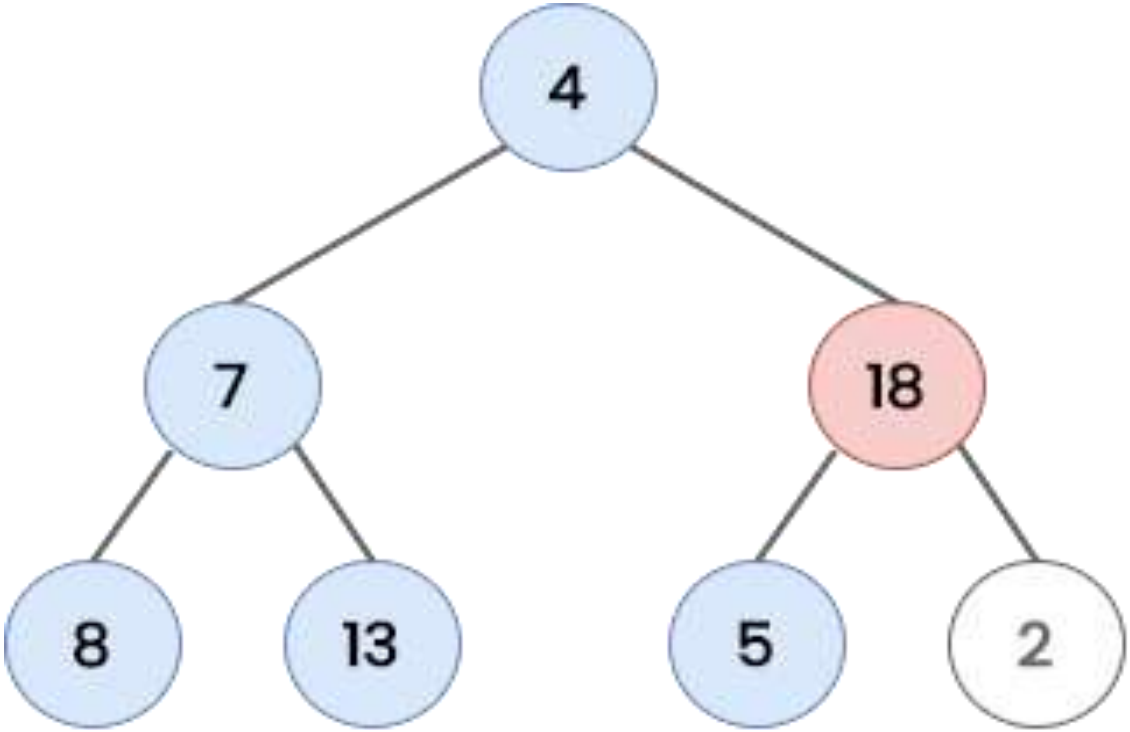
# Inorder Tree Traversal without Recursion-Example

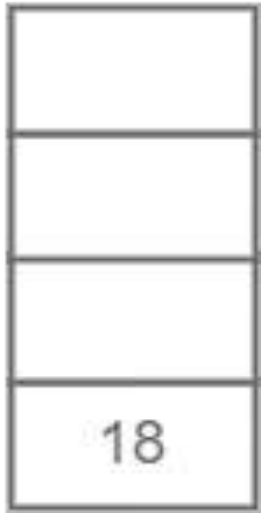


# Inorder Tree Traversal without Recursion-Example

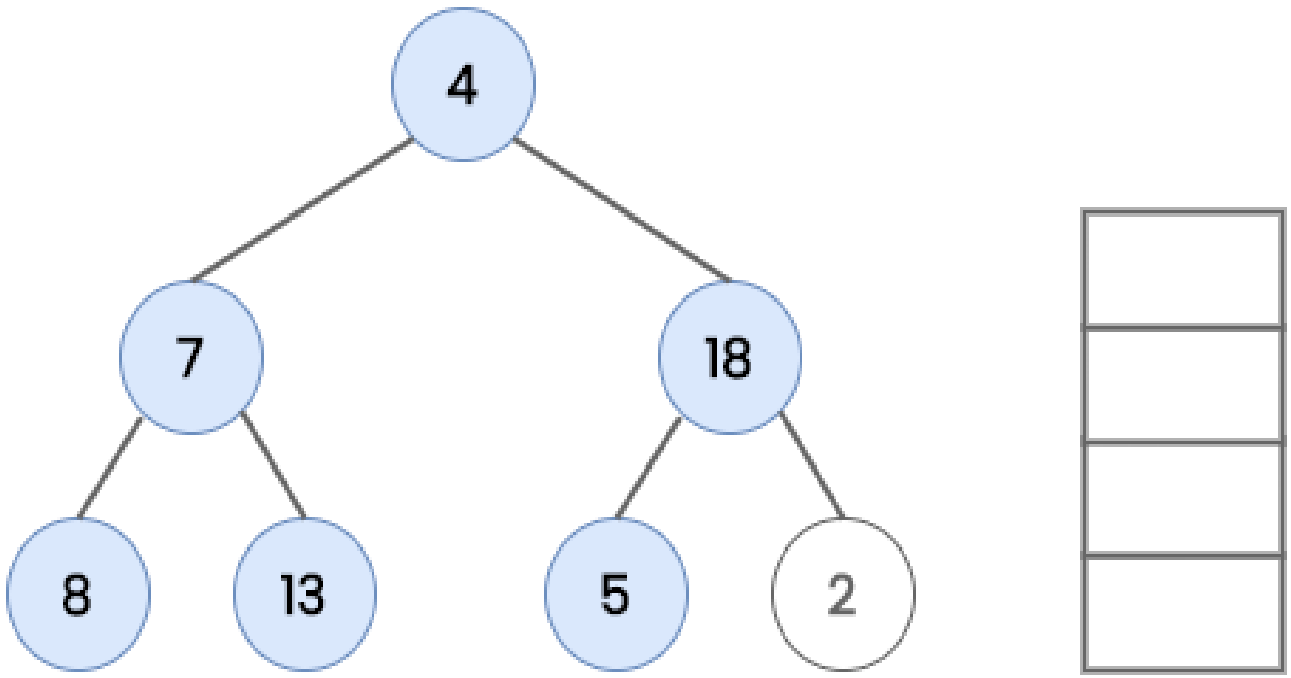


# Inorder Tree Traversal without Recursion-Example

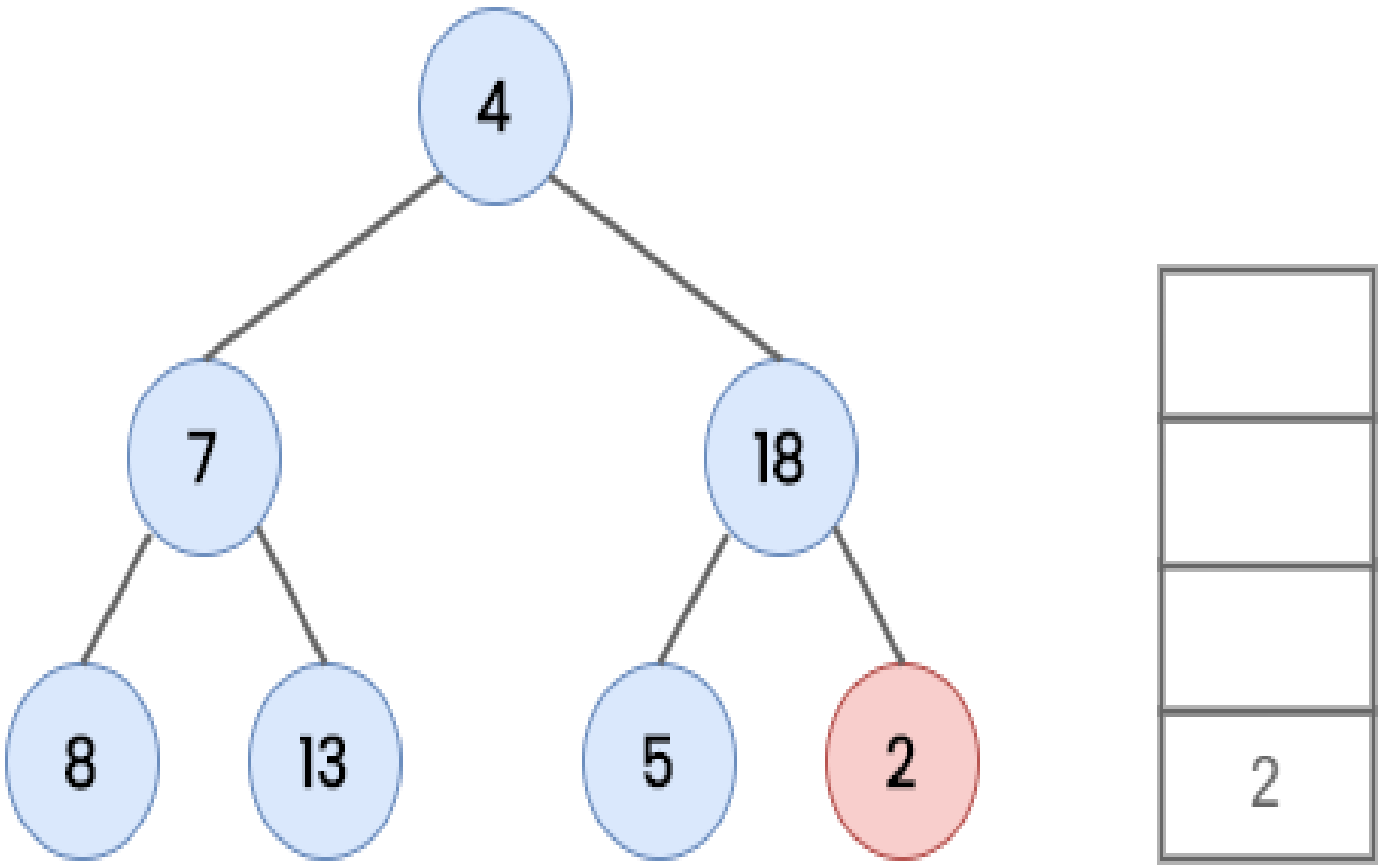


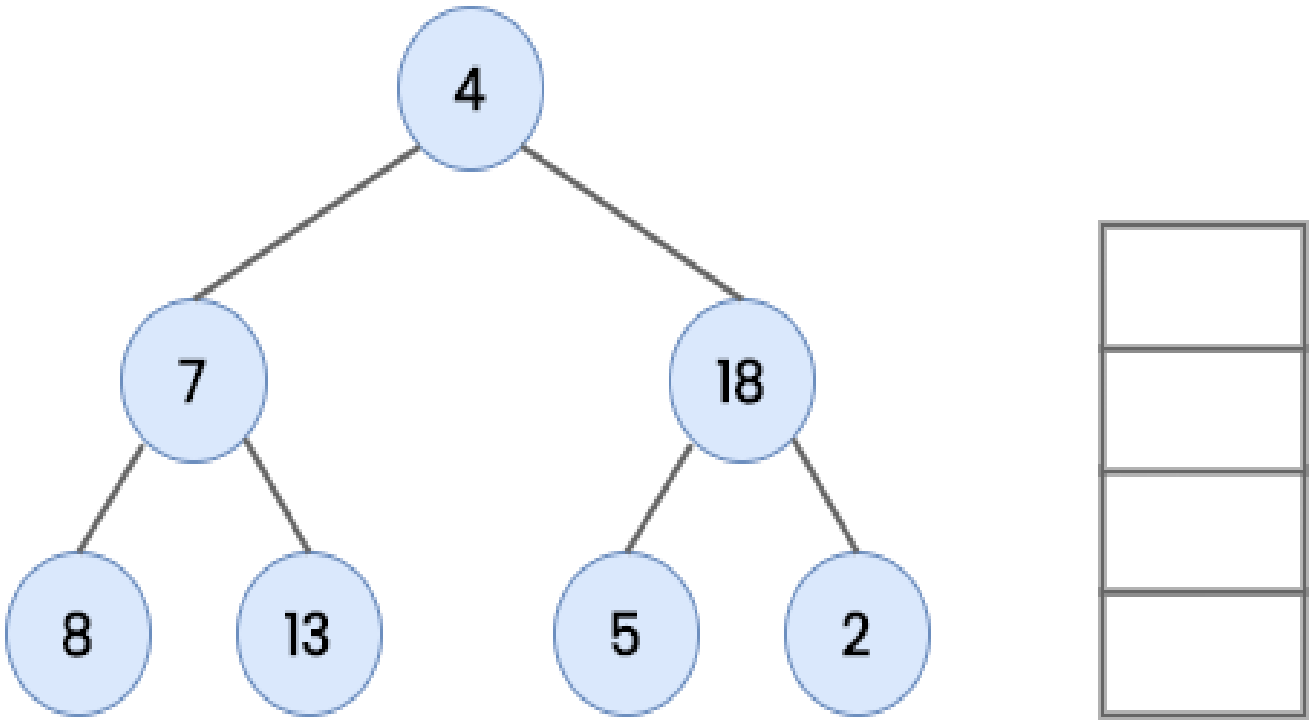
# Inorder Tree Traversal without Recursion–Example



# Inorder Tree Traversal without Recursion–Example



# Inorder Tree Traversal without Recursion-Example




The nodes are processed in the order [ 8, 7, 13, 4, 5, 18, 2 ] . This is the required in order traversal of the given tree.

# Pre-order Traversal Without Recursion

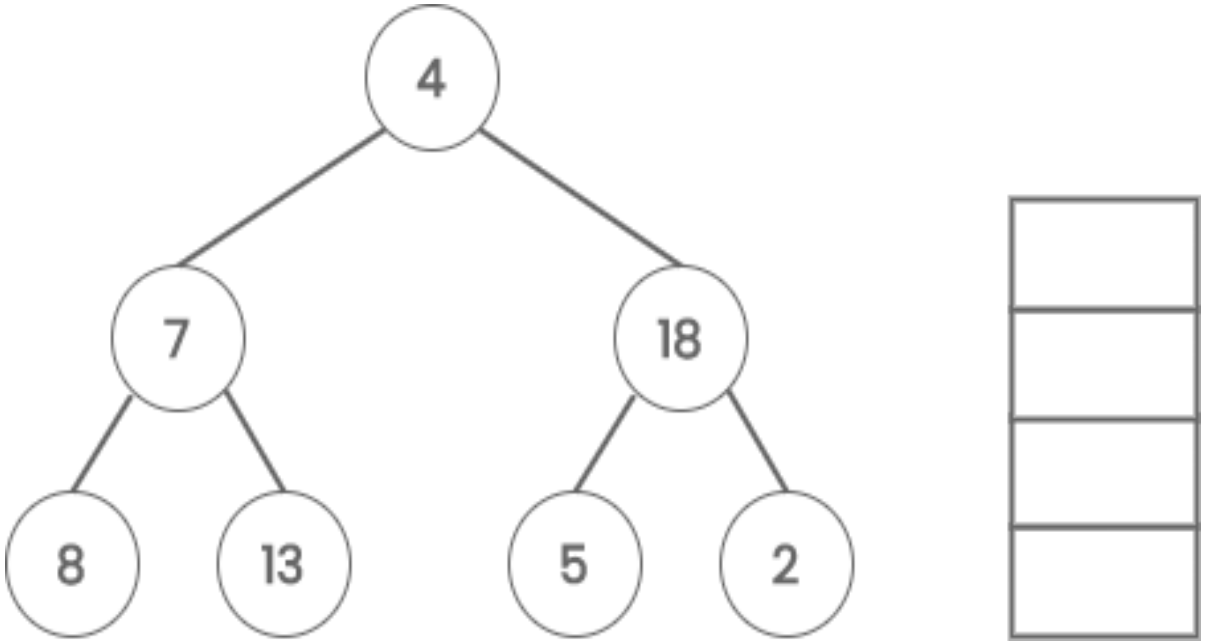
**The following operations are performed to traverse a binary tree in pre-order using a stack:**

1. Start with root node and push onto stack.
2. Repeat while the stack is not empty
   - POP the top element (PTR) from the stack and process the node.
   - PUSH the right child of PTR onto to stack.
   - PUSH the left child of PTR onto to stack.

# Preorder Traversal using a Stack

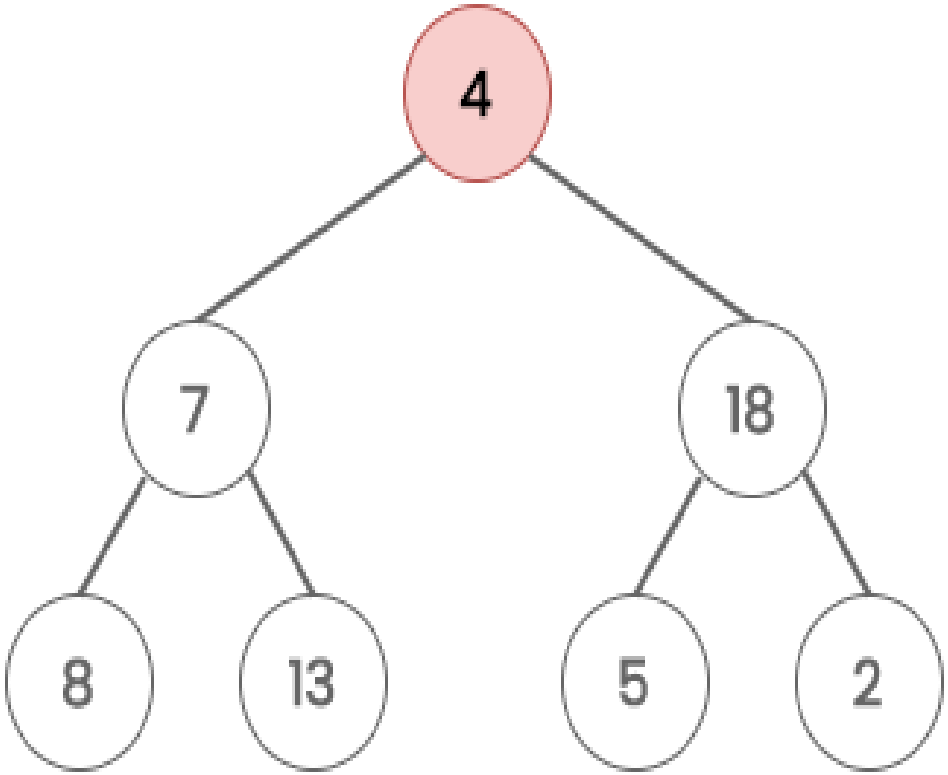
```
Preorder(root)
Begin
1.   curr =root
2.   Push(curr)
3.   While (Stack is non empty) Do
          curr=pop()
          printf(curr->data)
          if(curr->right != NULL) then
                    push(curr->right)
          if(curr-> left != NULL) then
                    curr=curr->left
                    push(curr)
4. exit
```

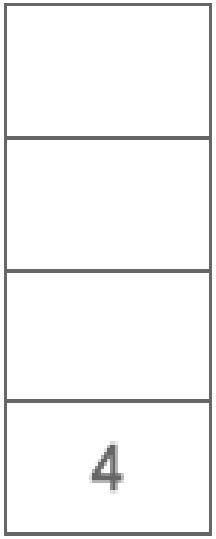
# Preorder Traversal using a Stack-Example



# Preorder Traversal using a Stack-Example

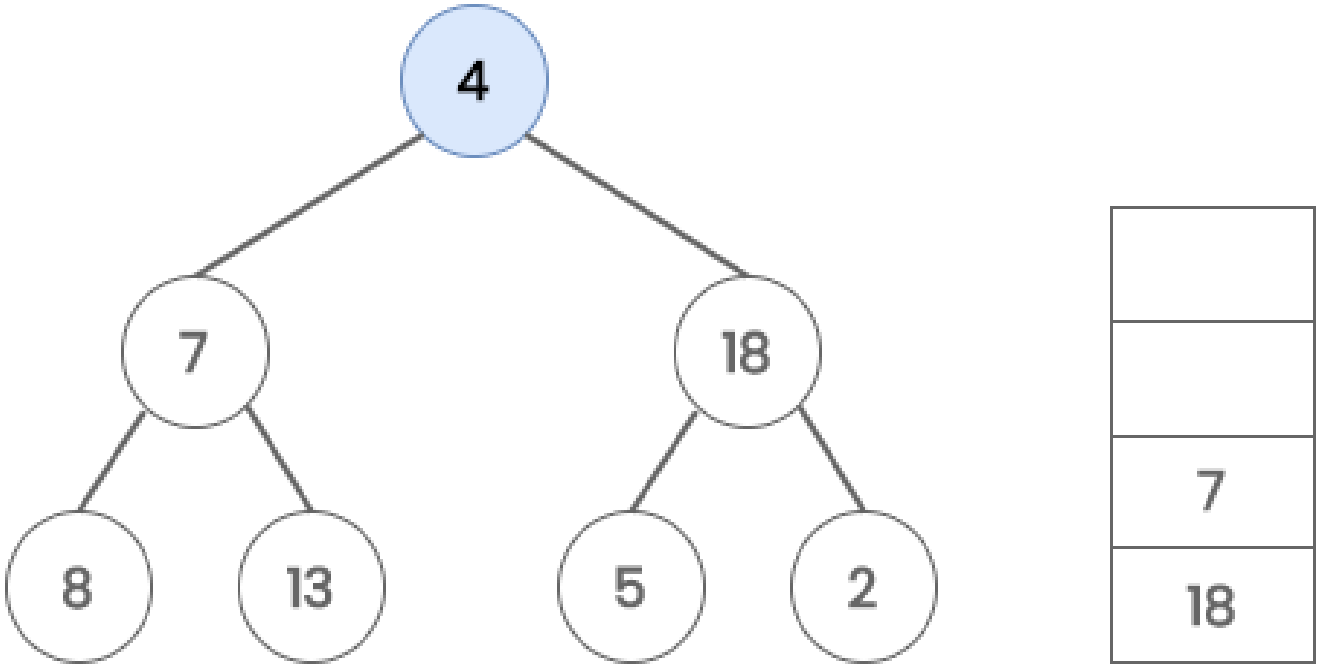
Start with node 4 and push it onto the stack.



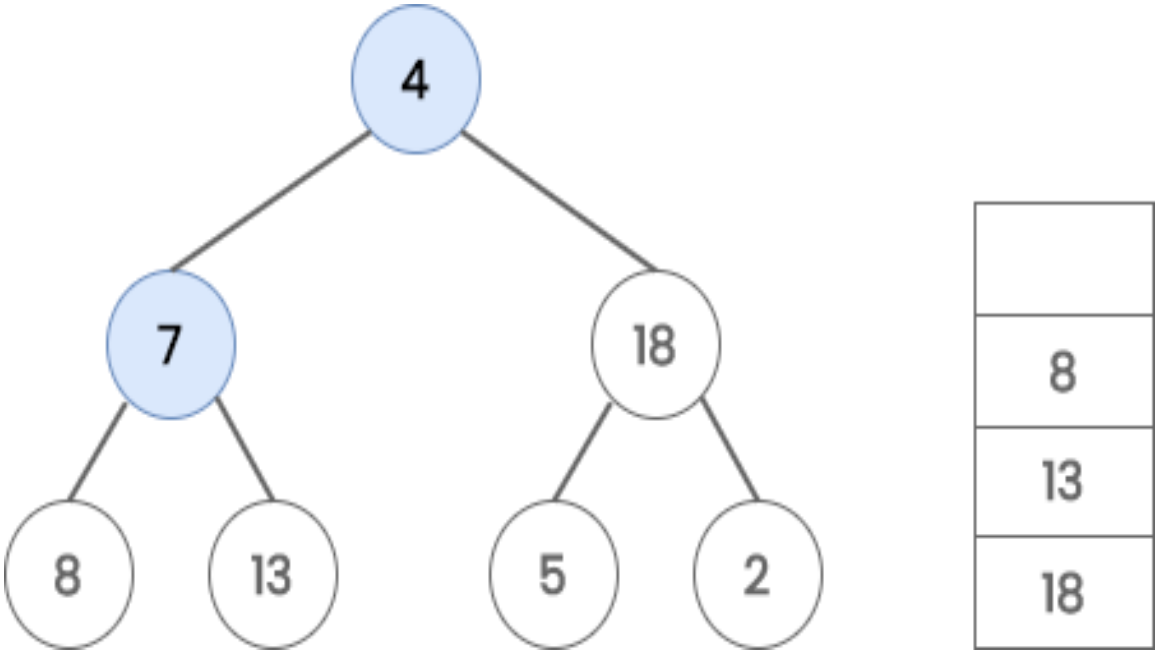
# Preorder Traversal using a Stack-Example

Since the stack is not empty, POP 4 from the stack, process it and PUSH its left(7) and right(18) child onto the stack.



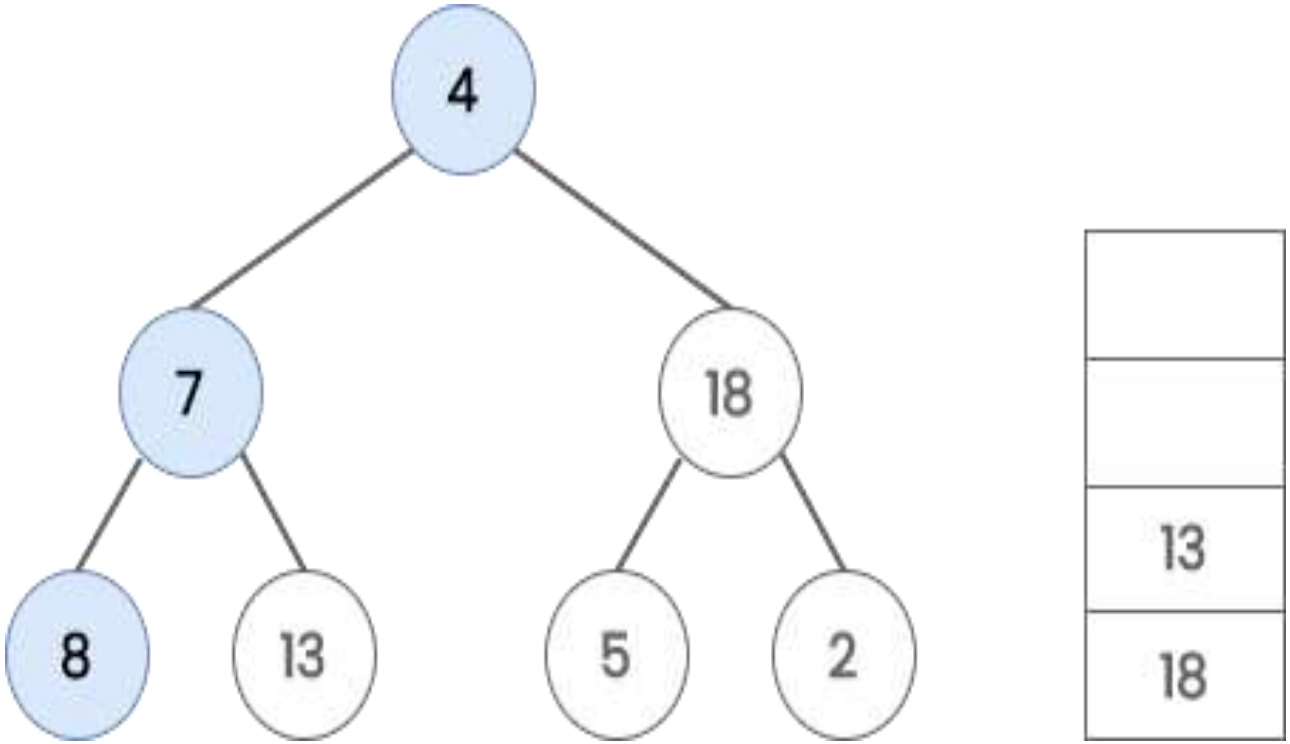
# Preorder Traversal using a Stack-Example

Repeat the same process since the stack is not empty. POP 7 from the stack, process it and PUSH its left(8) and right (13) child onto the stack.



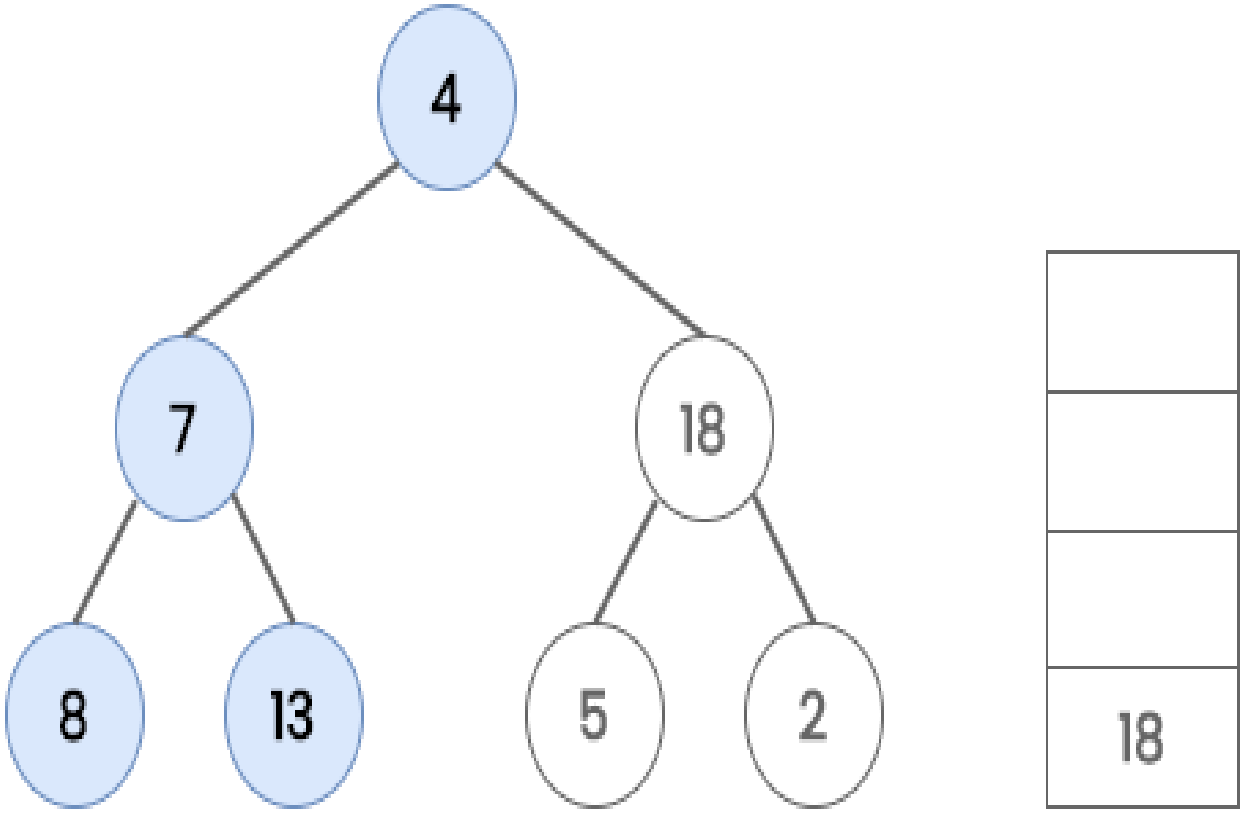
# Preorder Traversal using a Stack-Example

Again, POP 8 from the stack and process it. Since it has no right or left child we don't have to PUSH anything to the stack.



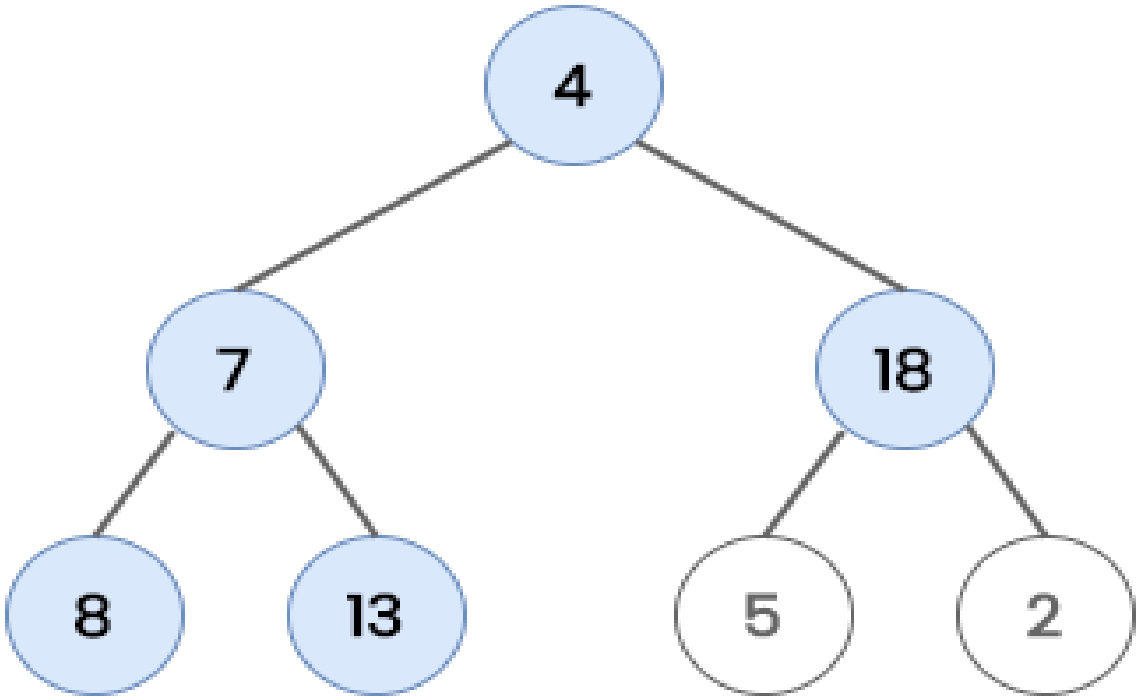
# Preorder Traversal using a Stack-Example

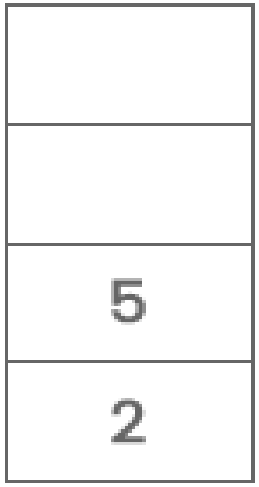
Now POP 13 from the stack and process it. We don't have to PUSH anything to the stack because it also doesn't have any subtrees.



# Preorder Traversal using a Stack-Example

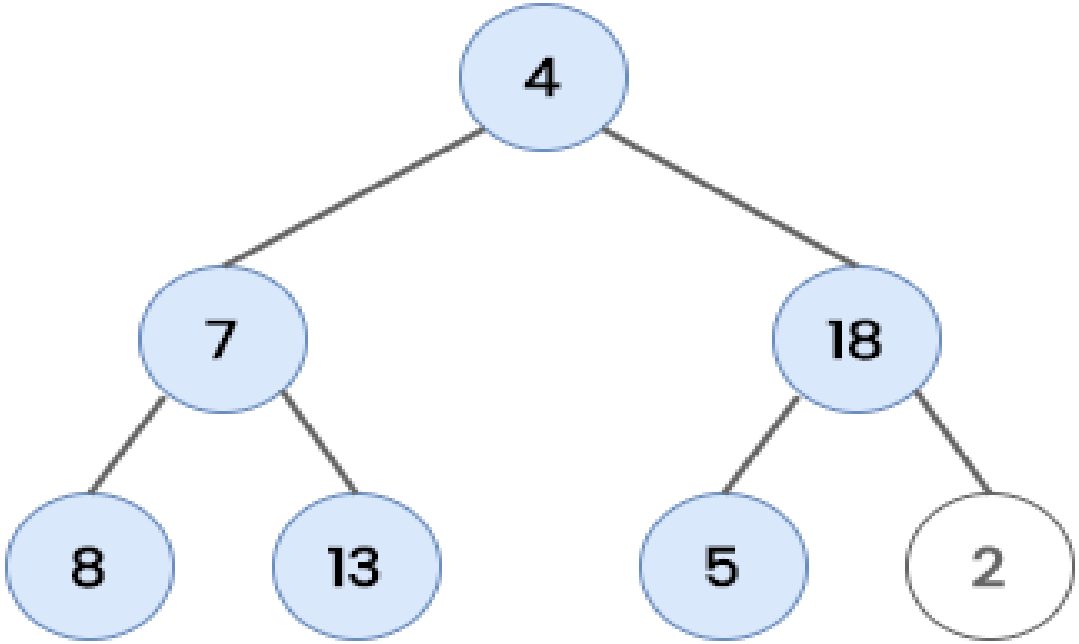
POP 18 from the stack, process it and PUSH its left(5) and right(2) child to the stack.

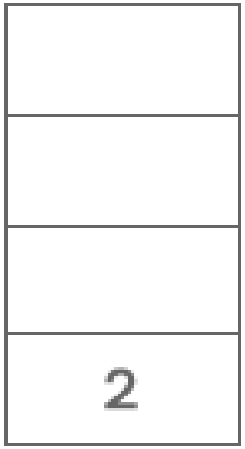


# Preorder Traversal using a Stack-Example

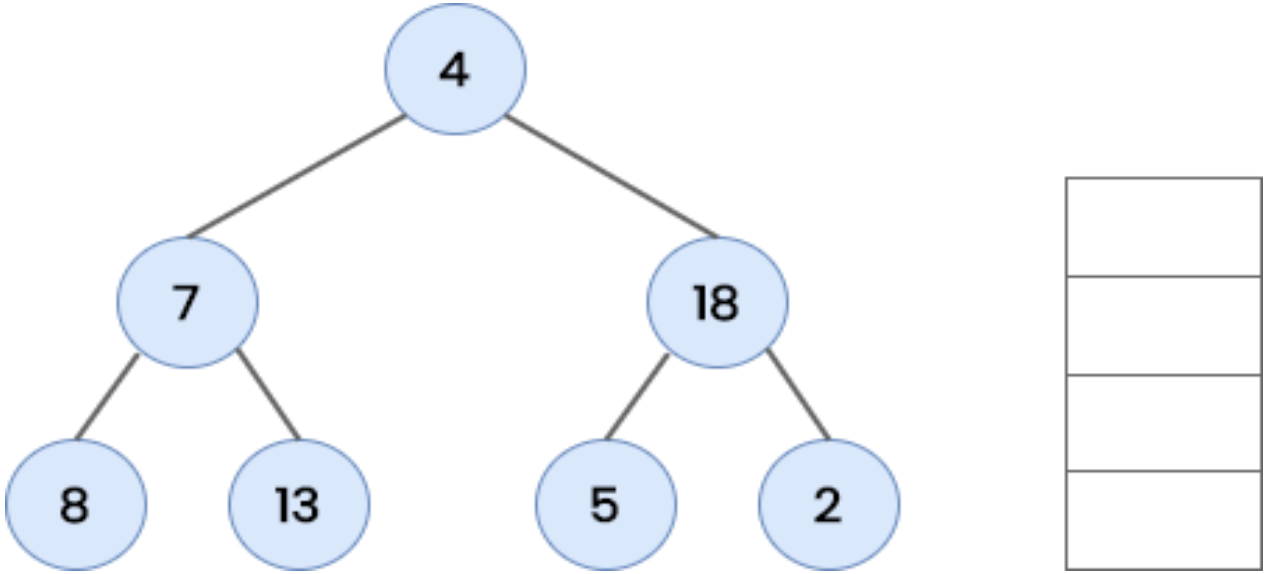
Similarly POP 5 and 2 from the stack one after another. Since both these nodes don't have any child, we don't have to PUSH anything onto the stack.



# Preorder Traversal using a Stack-Example

Similarly POP 5 and 2 from the stack one after another. Since both these nodes don't have any child, we don't have to PUSH anything onto the stack.




The nodes are processed in the order [ 4, 7, 8, 3, 18, 5, 2 ] . This is the required preorder traversal of the given tree.

# Post-order Traversal using a Stack

The post order traversal algorithm is more difficult to implement than the previous two algorithms. The following operations are performed to traverse a binary tree in post-order using a stack:

1. Start from the root, call it PTR.
2. Push PTR onto stack if PTR is not NULL.
3. Move to left of PTR and repeat step 2.
4. If PTR has a right child R, then PUSH -R onto the stack.
5. Pop and process positive element from stack and set as PTR.
6. If a negative node is popped, (PTR = -N), then set PTR = N and go to step 2.

# Postorder Traversal using a Stack– Formal Algorithm

```
postorder (root)
BEGIN
1.  curr = root;
2.  while(curr != NULL or Stack is nonempty) DO
3.      while(curr !=NULL) DO
4.          if(curr->right !=NULL) Then
5.              push(curr->right)
6.          push(curr)
7.          curr=curr->left
8.      curr=pop()
9.      if(curr->right !=NULL and top()==curr->right) Then
10.         pop()
11.         push(curr)
12.         curr=curr->right
13.     else
14.         print(curr->data)
15.         curr=NULL
END
```

# Binary Tree Construction from Preorder and Inorder

BEGIN

1. First element in the ***Preorder*** is the root
2. Search root element in ***Inorder***, say it is found at position *i*. Once found
   1. Construct left subtree for the elements which are left to ***i***
   2. Construct right subtree for the elements which are right to ***i***
3. Recursively construct left subtree and link it root left and recursively construct right subtree and link it root right

END

Preorder: A B D G H E C F I J

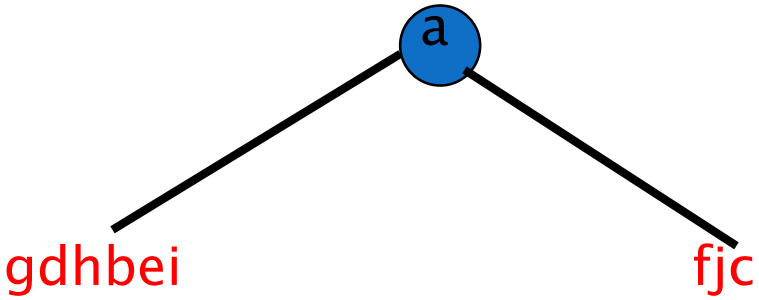
Inorder: G D H B E A I F J C

**In-order:** DBEAFC

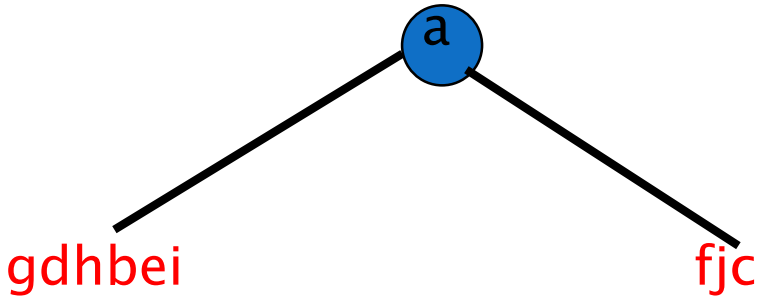
**Pre- Order:** ABDECF

# Inorder And Preorder

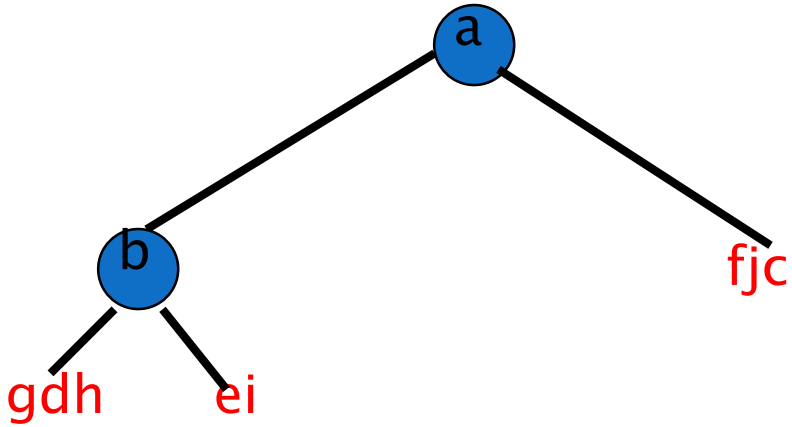
- inorder: g d h b e i a f j c
- preorder: a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree: gdhbei are nodes in the left subtree: fjc are nodes in the right subtree.



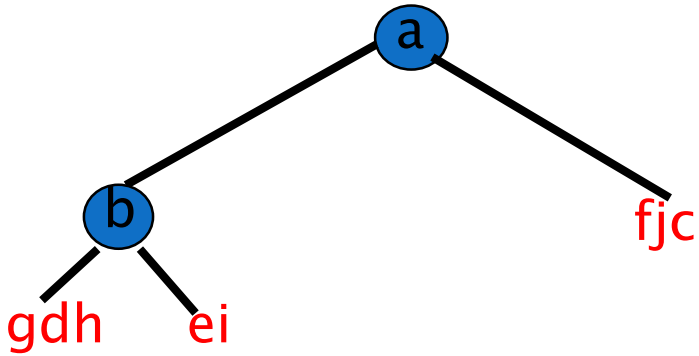
# Inorder And Preorder



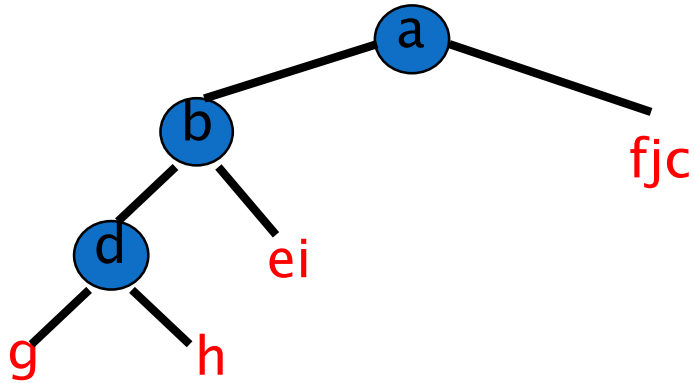

- preorder: b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



# Inorder And Preorder




- preorder: d g h e i c f j
- d is the next root; g is in the left subtree; h is in the right subtree.



# Binary Tree Construction from Postorder and Inorder

BEGIN

1. Last element in the ***Postorder*** is the root
2. Search root element in ***Inorder***, say it is found at position *i*. Once found
    1. Construct left subtree for the elements which are left to ***i***
    2. Construct right subtree for the elements which are right to ***i***
3. Recursively construct left subtree and link it root left and recursively construct right subtree and link it root right

END

Postorder: G H D E B I J F C A

Inorder: G D H B E A I F J C

# Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder: g d h b e i a f j c
- postorder: g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

# Expression Tree from Postfix Expression

```
Expression_Tree(PostfixExp a[])
BEGIN
1.  FOR i= 1 to n DO
2.      IF a[i] is Operand Then
3.          PUSH(a[i]
4.      Else
5.          Right=POP()
6.          Left=POP()
7.          New_Node=Node(left, right, a[i])
8.          PUSH(New_Node)
9.  Root=POP()
10. Return Root
END
```

PostfixExp: A B + C D E + * /

# Expression Tree from Prefix Expression

```
Expression_Tree(PrefixExp a[])
BEGIN
1.   FOR i= n to 1 STEP -1 DO
2.       IF a[i] is Operand Then
3.           PUSH(a[i]
4.       Else
5.           Left=POP()
6.           Right=POP()
7.           New_Node=Node(left, right, a[i])
8.           PUSH(New_Node)
9.   Root=POP()
10. Return Root
END
```

Prefix Exp: / + A B * C + D E

# Expression Tree Evaluation

```
ExpEval(ExpTree T)
BEGIN
1.  WHILE T != NULL DO
2.      IF (T->Left==NULL && T->Right==NULL) Then
3.          return T->Data
4.      ELSE
5.          op=T->Data
6.          opnd1=ExpEval(T->Left)
7.          opnd2=ExpEval(T->Right)
8.          return(ExpEval(opnd1 op opnd2))
END
```

# Evaluating an Expression tree

Assuming that *t* is a valid expression tree, a pseudo code algorithm for evaluating the expression tree is:

```
evalExpr(ExpressionTree t){
   if(t->left == NULL && t->right == NULL)    // node t is a leaf node
      return (t->info)    // which is an operand
   else {
      op =  t->info
      opnd1 = evalExpr(t->left)
      opnd2 = evalExpr(t->right) ;
      return(eval(opnd1, op, opnd2))
   }
}
```

# Binary Search Tree

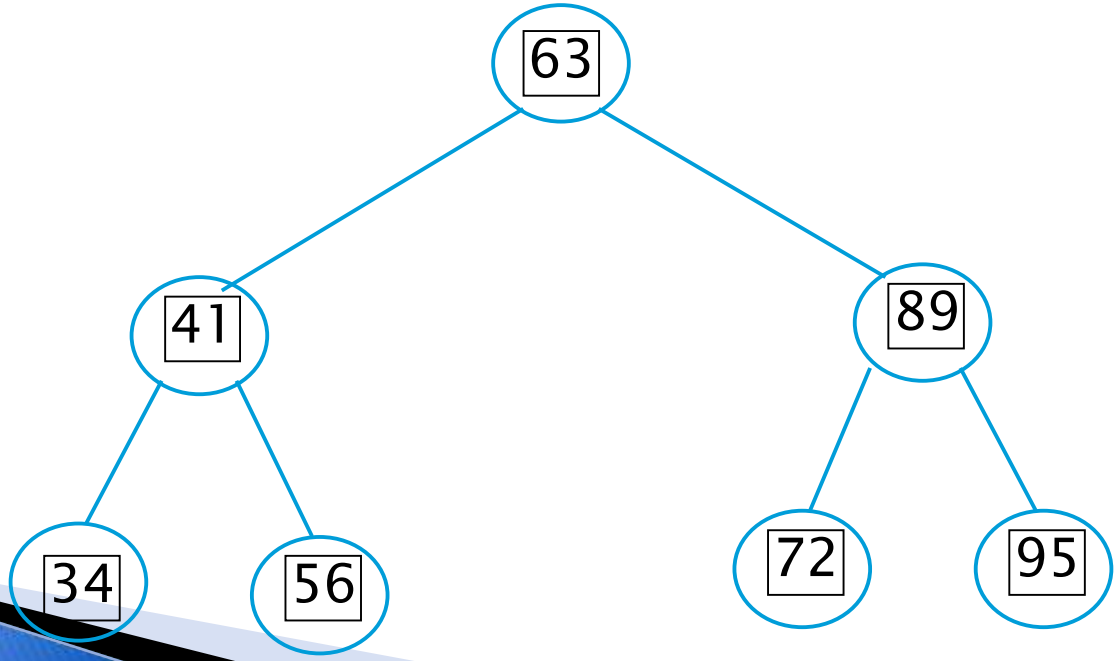
- Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.
- This data organization leads to O(log n) complexity for searching, insertion and deletion in certain types of the BST (balanced trees).
  - O(h) in general

# Advantages of Binary Search Tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

# Organization Rule for BST

- values at all nodes in the left subtree of a node are less than the node value
- values in all nodes at the right subtree of a node are greater than the node values



# Binary Tree

```
typedef struct node {
            int  info;
            struct node *left;
           struct node *right
   } *treeNode;
```

# BST Creation

Suppose the data elements are – **45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

# BST Operations: Search

Searching in the BST

method search(key)

- implements the binary search based on comparison of the items in the tree
- the items in the BST must be comparable (e.g integers, strings, etc.)

- The search starts from the root node.
- It probes down, comparing the values in each node with the target, till it finds the node value equal to the target.
  - Returns this node address or null if there is none (i.e. search is unsuccessful).

# Search in a BST: Non-Recursive code

```
treeNode iterativeSearch(treeNode root, int key) {
    // Traverse untill root reaches to terminal node
    while (root != NULL) {
        if (key > root->info)   // pass right subtree as new tree
            root = root->right;
        else if (key < root->info)  // pass left subtree as new
    tree
            root = root->left;
        else
            return root; // if the key is found return 1
    }
    return NULL;
}
```

# Search in a BST: Recursive code

```
treeNode search(treeNode root, int key)  {
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->info == key)
        return root;

    // Key is greater than root's key
    if (root->info < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

# BST Operations: Insertion

method insert(key)

- find the place for the new item near the frontier of the BST while retaining its organization of data:
  - starting at the root it probes down the tree till it finds a node whose left or right pointer is NULL and that is a logical place for the new value to be inserted

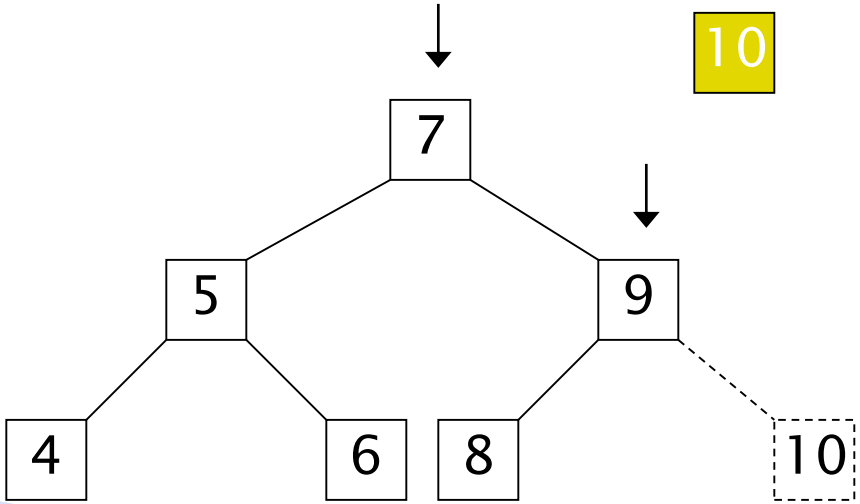
# Operations: Insertion

Case 1: The Tree is Empty

- Set the root to a new node containing the item

Case 2: The Tree is Not Empty

- Call a recursive function to insert the item




10 > 7

10 > 9

# BST Insertion: Recursive

```
Node*insert( node*root, int key)
{
    if(root==NULL)
    {
        root=(node*)malloc(sizeof(node));
        root->left=root->right=NULL;
        root->info=key;
    }
    else if  (key<root->info)
        root->left=insert(root->left,key);
     else if  (key> root->info)
         root-> right =insert(root->right,
key);
```

```
else if  (Key== root->info)
{
    printf("Duplicate Node");
    exit (0);
}
return (root)

}//end of function
```

# BST Insertion: Nonrecursive Code

```
truct node *insert_nrec(struct node *root, int ikey)
{
        struct node *tmp,*par,*ptr;

        ptr = root;
        par = NULL;

        while( ptr!=NULL)
        {
                par = ptr;
                if(ikey < ptr->info)
                        ptr = ptr->lchild;
                else if( ikey > ptr->info )
                        ptr = ptr->rchild;
                else
                {
                        printf("\nDuplicate key");
                        return root;
                }
        }
```

```
tmp=(struct node
    *)malloc(sizeof(struct node));
        tmp->info=ikey;
        tmp->lchild=NULL;
        tmp->rchild=NULL;

        if(par==NULL)
                root=tmp;
        else if( ikey < par->info )
                par->lchild=tmp;
        else
                par->rchild=tmp;

        return root;
}/*End of insert_nrec( )*/
```

# BST Insertion: Recursive

```
treeNode insert(treeNode root, int key)  {
    if(root==NULL)  {
        root=(treeNode) malloc(sizeof(struct node));
        root->left=root->right = NULL;
        root->info=key;
     }
    else {
         if(key<root->info)
             root->left=insert(root->left, key);
         else if(key>root->info)
            root->right=insert(root->right, key);
    }
    return root;
  }
```

# BST: Smallest & Largest (Recursive)

```
findmin(node  root)
{
   if(root== NULL)
        return NULL
   else if(root->left== Null)
        return root;
else
     return(findmin(root->left));
}
```

```
findmax(node  root)
{
      if(root== NULL)
             return NULL
      else if(root->right== Null)
              return root;
       else
       return(findmax(root->right);
}
```

# BST: Smallest & Largest (Non-Recursive)

```
findmin(node  root)
{
      if(root!=NULL)
      {
          while(root->left != NULL)
           {
              root=root->left
           }
      }
    return root;
}
```

```
findmax(node  root)
{
      if(root!=NULL)
      {
          while(root->right != NULL)
           {
              root=root->right;
           }
      }
    return root;
}
```

# BST Operations: Removal

- removes a specified item from the BST and adjusts the tree
- uses a binary search to locate the target item:
  - starting at the root it probes down the tree till it finds the target or reaches a leaf node (i.e. target not in the tree)
- removal of a node must not leave a 'gap' in the tree,

# Removal in BST – Pseudocode

method remove (key)

I if the tree is empty return false

II Attempt to locate the node containing the target using the binary search algorithm

if the target is not found return false

else the target is found, so remove the node:

Case 1: if the node has 2 empty subtrees

replace the link in the parent with NULL

Case 2: if the node has a left and a right subtree

- replace the node's value with the max(min) value in the left(right) subtree
- delete the max(min) node in the left(right) subtree

# Removal in BST – Pseudocode

Case 3: if the node has no left child

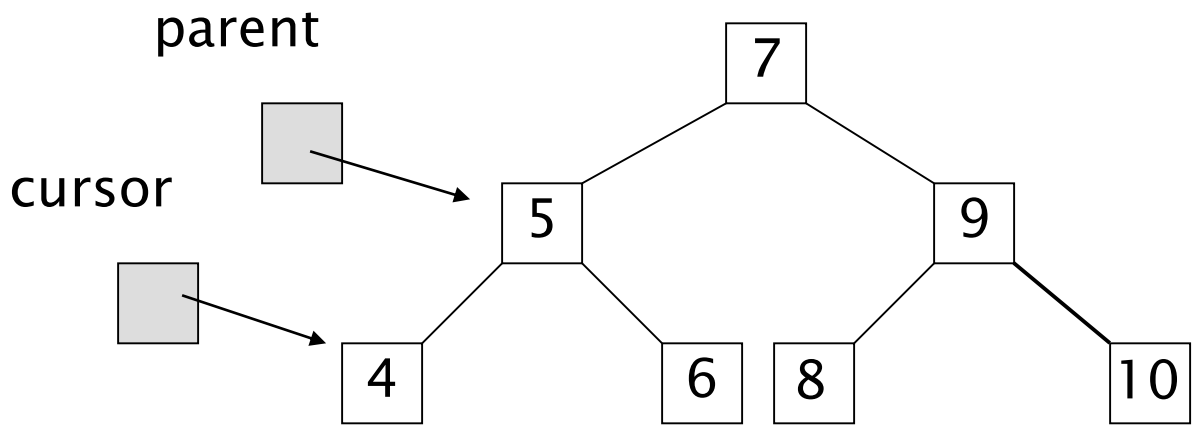
- link the parent of the node to the right (non-empty) subtree

Case 4: if the node has no right child

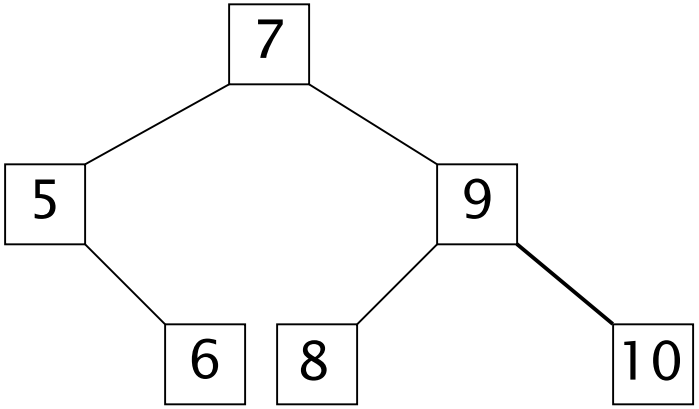
- link the parent of the target to the left (non-empty) subtree

# Removal in BST: Example

Case 1: removing a node with 2 empty subtrees




Removing 4
replace the link in the parent with NULL



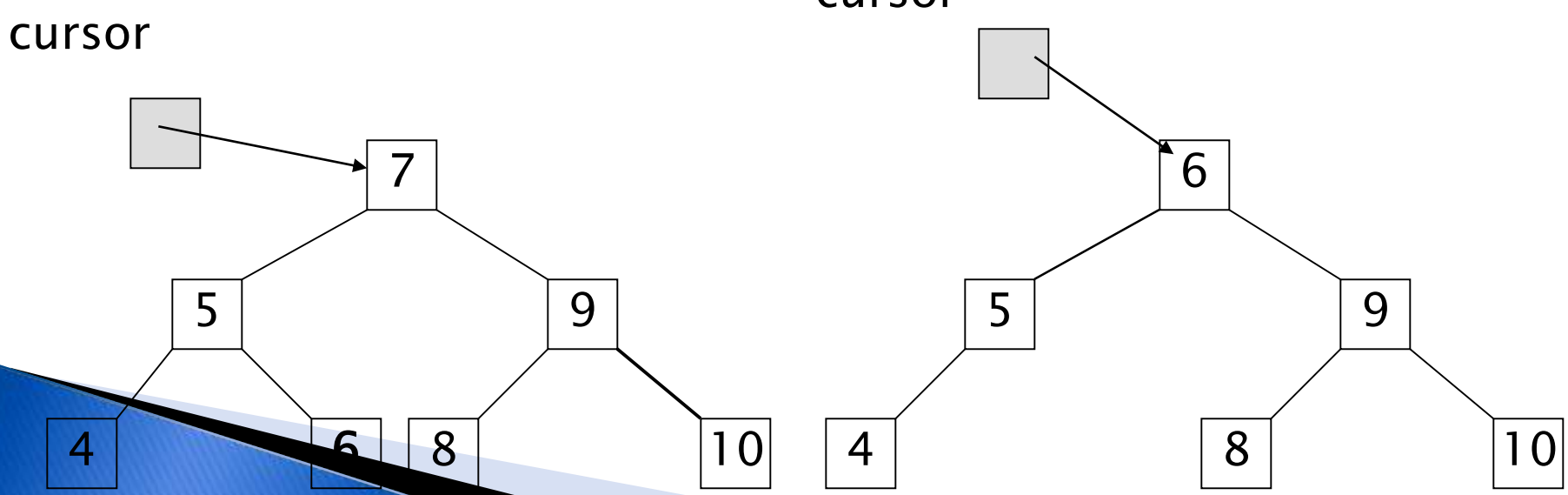
# Removal in BST: Example

Case 2: removing a node with 2 subtrees

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

What other element can be used as replacement?

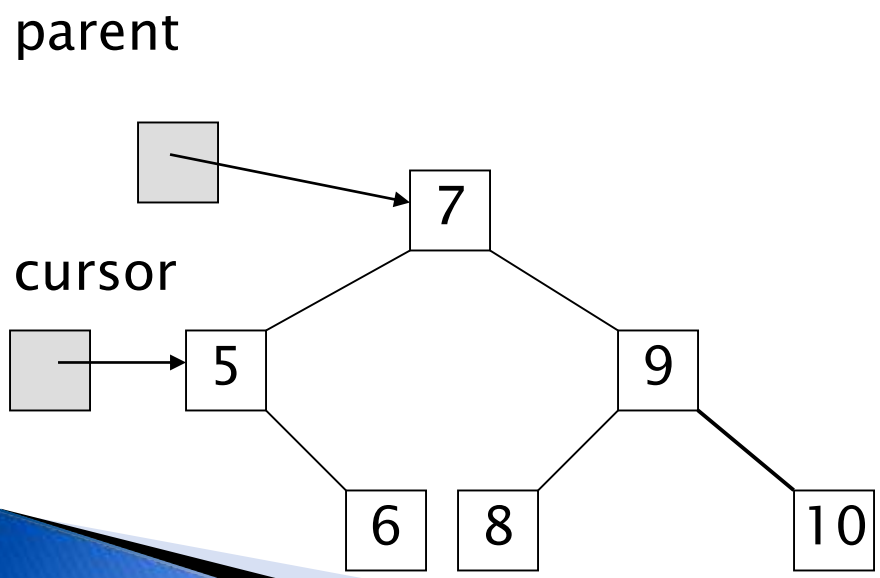
Removing 7

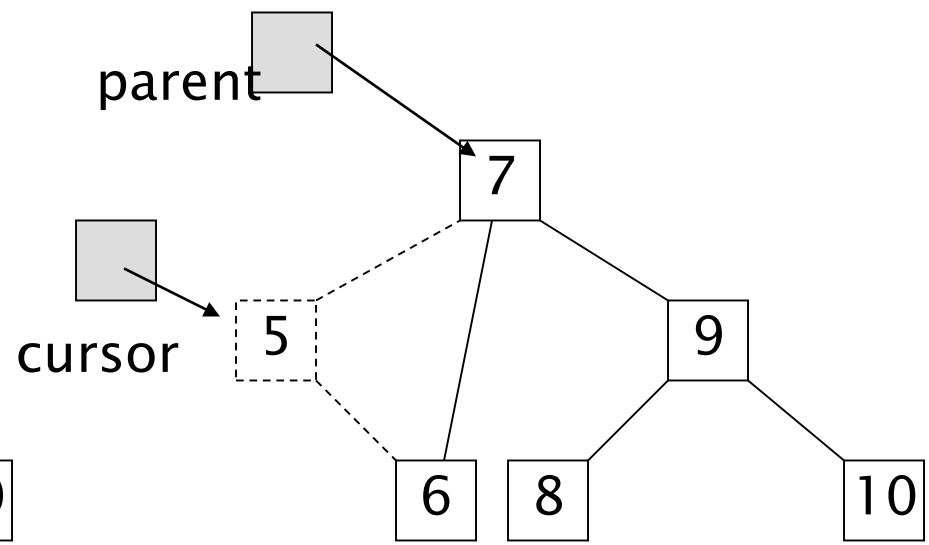


# Removal in BST: Example

Case 3: removing a node with 1 empty subtree

the node has no left child:
link the parent of the node to the right (non-empty) subtree



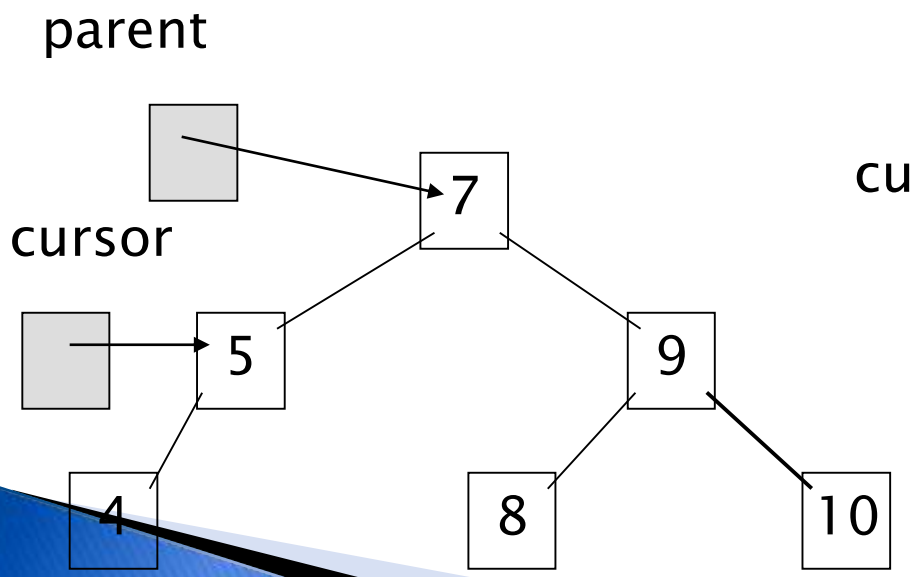



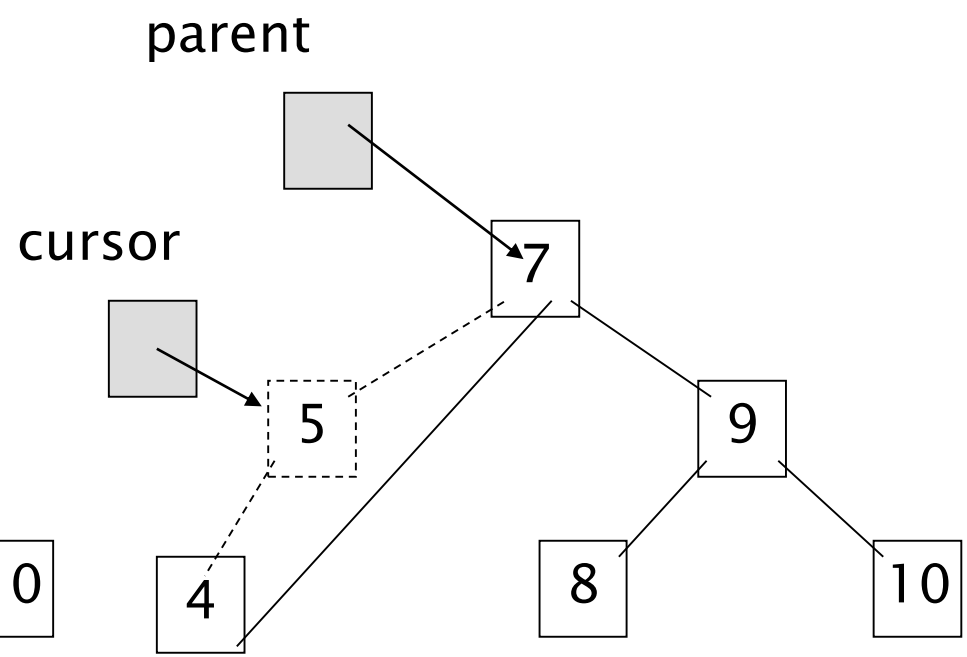
# Removal in BST: Example

Case 4: removing a node with 1 empty subtree

the node has no right child:
link the parent of the node to the left (non-empty) subtree

Removing 5



# Removal in BST: C code

```
treeNode * Delete(treeNode *node, int key)
{
   treeNode *temp;
   if(node==NULL)
       printf("Empty Tree");
   else if(key < node->info)
      node->left = Delete(node->left, key);
   else if(key > node->info)
      node->right = Delete(node->right, key);
   else if(node->right && node->left)
    {
         temp = FindMin(node->right);
         node -> info = temp->info;
         node -> right = Delete(node->right,
                                   temp->info);
   }
```

# Removal in BST: C code

```
else
 {
      temp = node;
      if(node->left == NULL)
            node = node->right;

       else if(node->right == NULL)
            node = node->left;
         free(temp);
      }
   }
   return node;
}
treeNode * FindMin(treeNode *node) {
   if(node==NULL)   return NULL;
   if(node->left != NULL)
      return FindMin(node->left);
   else
      return node;
}
```

# BST: Total Nodes

**totalNodes( NODE *root)**

BEGIN

1. IF (root==NULL) Then
2. return 0;
3. ELSE
4. return totalNodes (root->left) + totalNodes (root->right) + 1

END

---

```
int totalNodes(NODE *tree){
    if(tree==NULL)
        return 0;
    else
        return(totalNodes(tree->left) + totalNodes(tree->right) + 1);
}
```

# BST: Leaf Nodes

**totalLeafNodes( NODE *root)**

BEGIN

1. IF (root==NULL) Then
2. return 0;
3. ELSE IF (root->left==NULL && root->right==NULL) Then
4. return 1
5. ELSE
6. return totalLeafNodes (root->left) + totalLeafNodes (root->right)

END

---

```
int totalLeafNodes(NODE *tree){
    if(tree==NULL)
        return 0;
    else if((tree->left==NULL) && (tree->right==NULL))
        return 1;
    else
        return (totalLeafNodes(tree->left) +totalLeafNodes(tree->right));
}
```

# BST: Internal Nodes

```
totalInternalNodes(NODE *root)
BEGIN
1.  IF ((root==NULL) || (root->left==NULL && root->right==NULL)) Then
2.      return 0;
3.  ELSE
4.      return totalInternalNodes(root->left) + totalInternalNodes(root->right) + 1
END
```

---

```
int totalInternalNodes(NODE *tree){
    if( (tree==NULL) || ((tree->left==NULL) && (tree->right==NULL)))
        return 0;
    else
        return (totalInternalNodes(tree->left)+ totalInternalNodes(tree->right) + 1);
}
```

# BST: Height

```
Height (NODE *root)
BEGIN
1.  IF root==NULL Then
2.      return 0
3.  ELSE
4.      leftHeight=Height(root->left)
5.      rightHeight=Height(root->right)
6.      IF leftHeight > rightHeight Then
7.          return (leftHeight+1)
8.      ELSE
9.          return (rightHeight+1)
END
```

```
int Height(NODE *tree){
    int leftheight, rightheight;
    if(tree==NULL)
        return 0;
    else{
        leftheight = Height(tree->left);
        rightheight = Height(tree->right);
        if(leftheight > rightheight)
            return (leftheight + 1);
        else
            return (rightheight + 1);
    }
}
```

# BST: Delete a Tree

**deleteTree(NODE * root)**

BEGIN

1. IF root!=NULL Then
2. deleteTree(root->left)
3. deleteTree(root->right)
4. free(root)

END

---

```
NODE *deleteTree(NODE *tree){
    if(tree!=NULL){
        deleteTree(tree->left);
        deleteTree(tree->right);
        free(tree);
    }
}
```

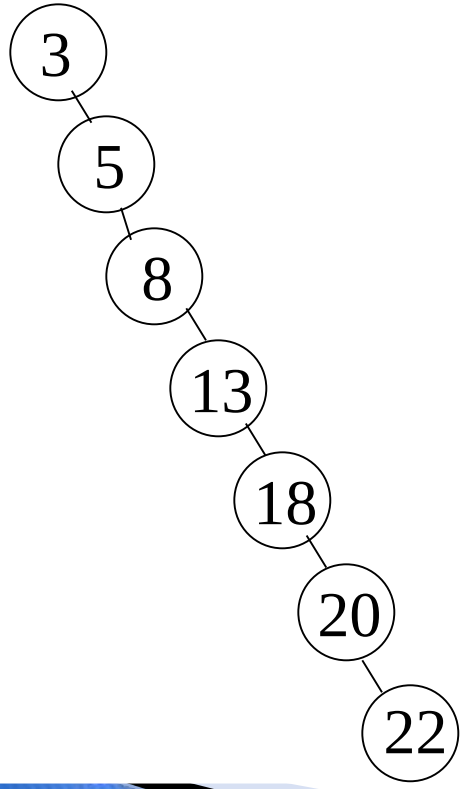
# AVL TREES

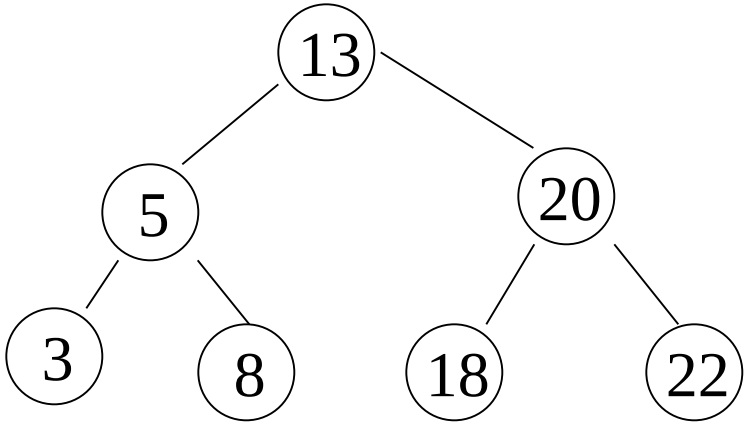
# Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

# Motivation

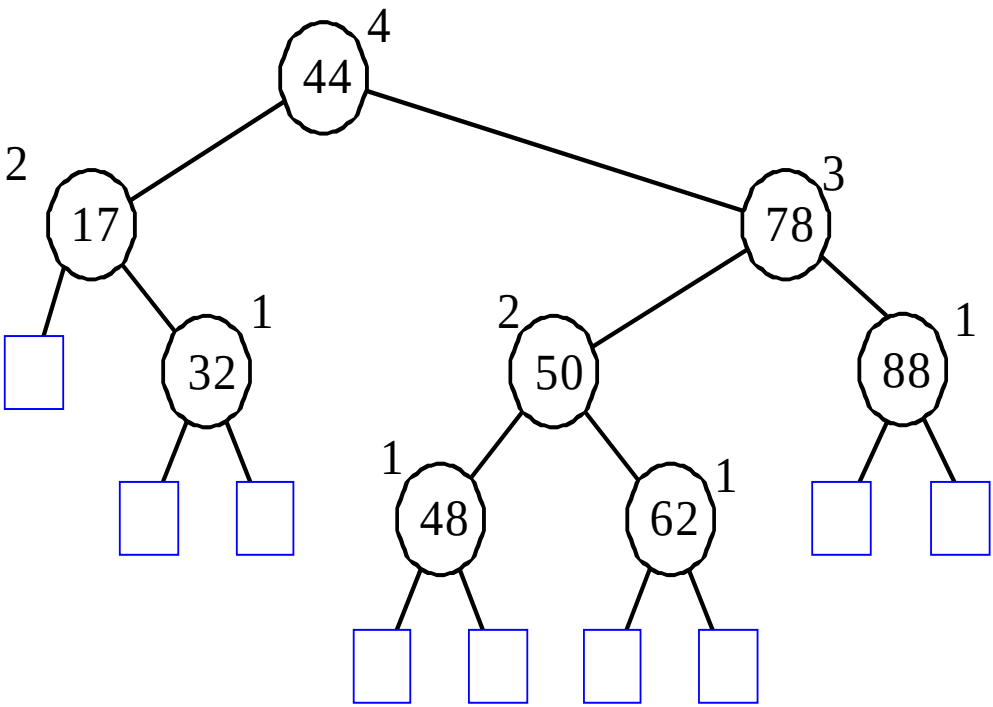
- ✓ **When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22**



# AVL (Adelson-Velskii and Landis) Trees

- **An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1.**




**An example of an AVL tree where the heights are shown next to the nodes**

# Balance Factor

- ✓ The AVL tree is a height-balanced tree where each node's right and left subtree height differences are either -1, 0 or 1.
- ✓ A factor known as the balance factor keeps the subtrees' height differences apart.
- ✓ As a result, we can define AVL as a balanced binary search tree in which each node's balance factor is either -1, 0 or +1. In this case, the formula follows to determine the balancing factor:

**Balance Factor=Height of Left Subtree – Height of Right Subtree.**

- ✓ **Balance condition:**

  For every node in the tree, height of left and right subtree can differ by at most 1

# Balance Factor

- ✓ If a subtree has a **balance factor greater than zero**, it is said to be left-heavy because the height of the left subtree is greater than the height of the right subtree, implying that the left subtree contains more nodes than the right subtree.

- ✓ If a subtree has a **balance factor of less than 0**, it is said to be right-heavy because the height of the left subtree is less than the height of the right subtree, implying that the right subtree has more nodes than the left subtree.

- ✓ If the **balance factor is zero,** the subtree is perfectly balanced, with equal heights in both the left and right subtrees.

# AVL Tree Rotaion

- **LL Rotation:** When a new node is added as the left child of the unbalanced node's left subtree.
- **RR Rotation:** When a new node is added as the right child of the right subtree of an unbalanced node, this process is known as RR Rotation.
- **LR Rotation:** When a new node is added as the left child of an unbalanced node's right subtree.
- **RL Rotation:** When a new node is added as the right child of an unbalanced node's left subtree.

# AVL Tree Rotaion

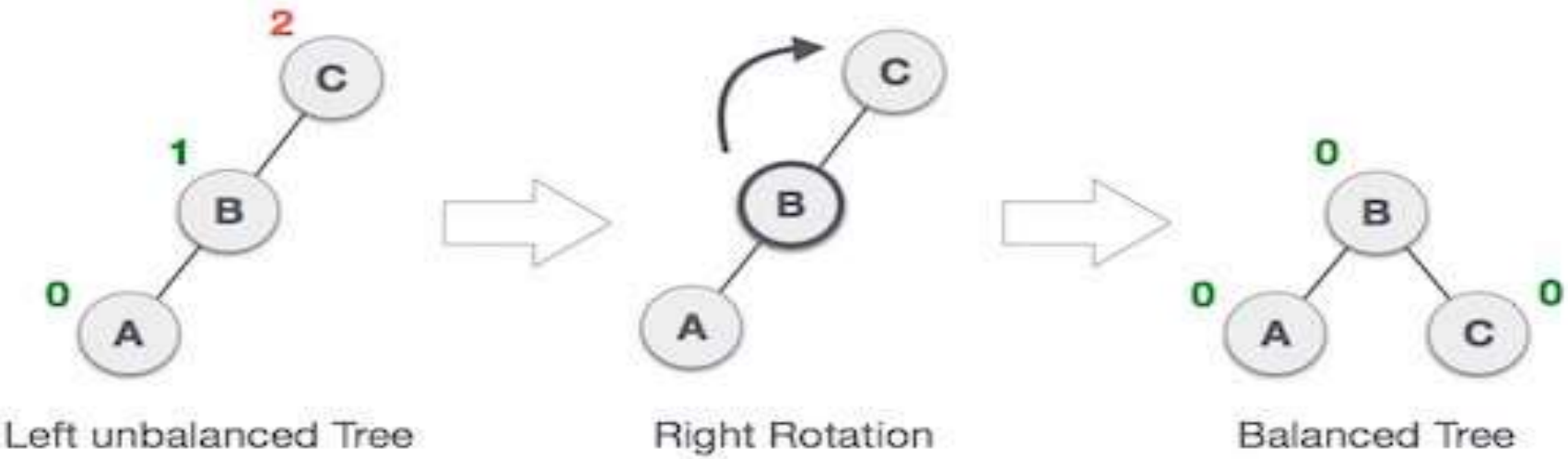
Following are the possible 4 arrangements:

- y is the left child of z and x is the left child of y (**Left Left Case-LL Rotation**)
- y is the left child of z and x is the right child of y (**Left Right Case-LR Rotation**)
- y is the right child of z and x is the right child of y (**Right Right Cas-RR Rotation**)
- y is the right child of z and x is the left child of y (**Right Left Case-RL Rotation**)

**Reference:** *https://www.javatpoint.com/avl-tree-program-in-c*

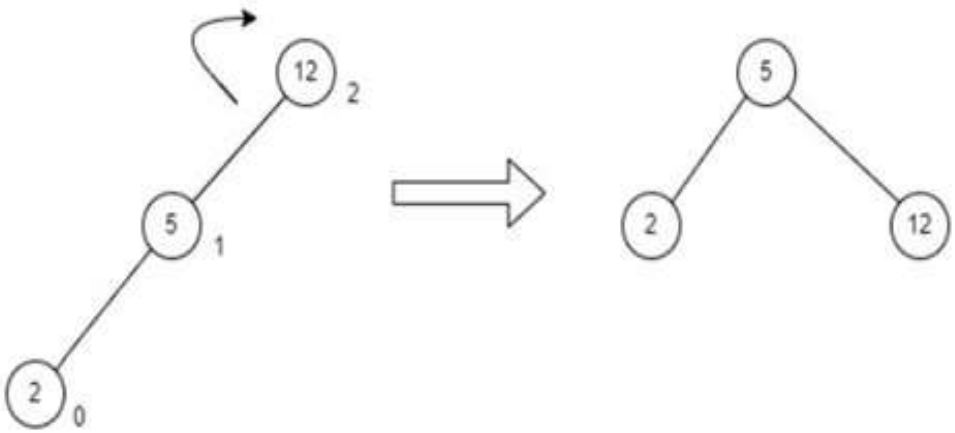
# LL-Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, **LL rotation** is **clockwise rotation**, which is applied on the edge below a node having balance factor 2.




In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

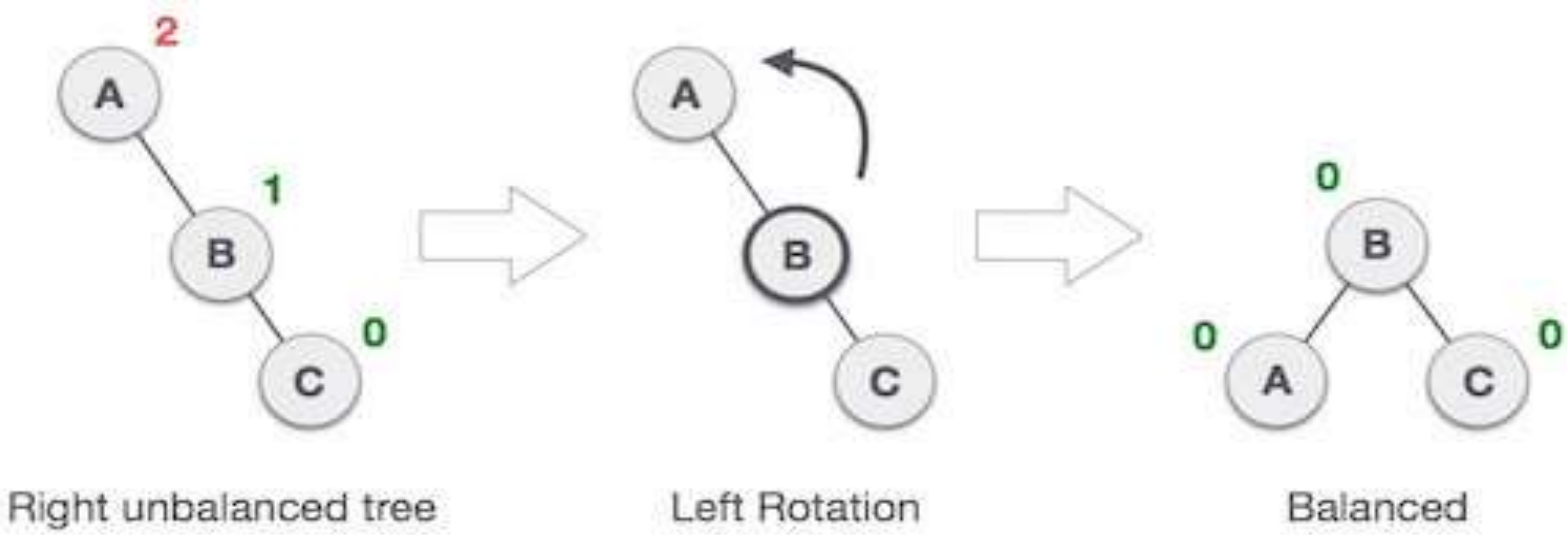
# LL-Rotation




```
NODE *LL(NODE * node){
    NODE *temp;
    temp=node->left;
    node->left=temp->right;
    temp->right=node;
    node->height=findMax(height(node->left),height(node->right))+1;
    temp->height=findMax(height(temp->left), node->height)+1;
    return temp;
}
```

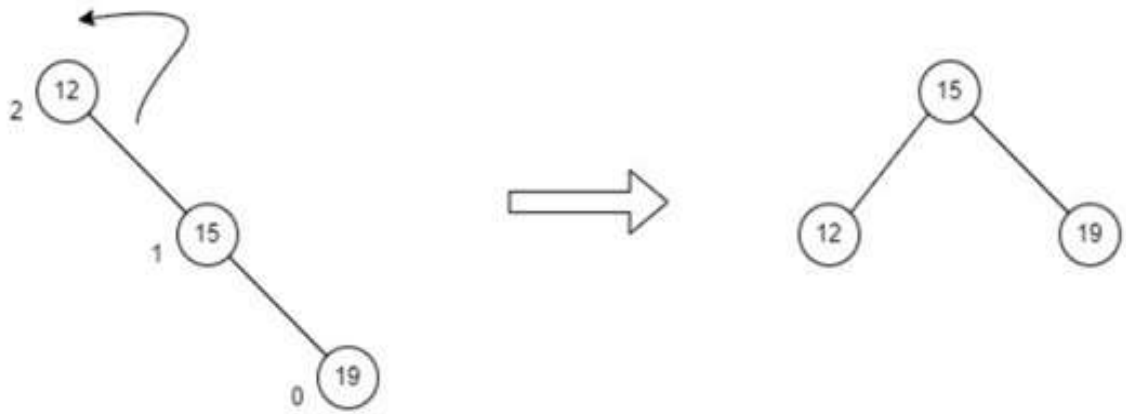
# RR-Rotation

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, **RR rotation is an anticlockwise rotation**, which is applied on the edge below a node having balance factor -2




In above example, node A has balance factor –2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

# RR-Rotation

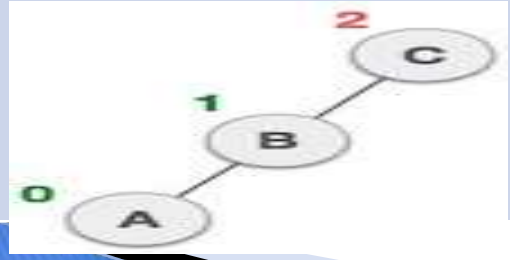



```
NODE *RR(NODE *node){
    NODE *temp;
    temp=node->right;
    node->right=temp->left;
    temp->left=node;
    node->height=findMax(height(node->left), height(node->right))+1;
    temp->height=findMax(node->height,height(temp->right))+1;
    return temp;
}
```

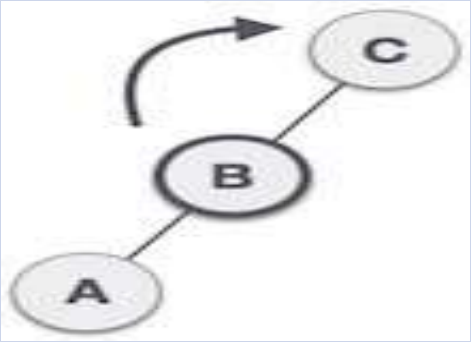
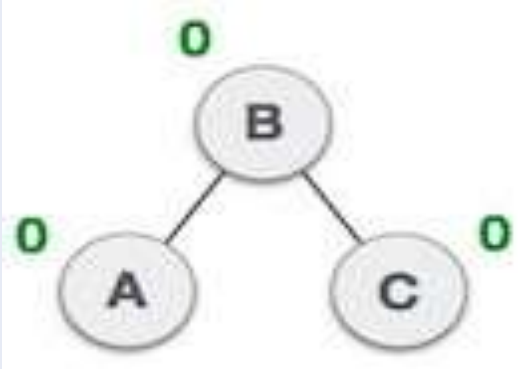
# LR-Rotation

Double rotations are bit tougher than single rotation which has already explained above. **LR rotation = RR rotation + LL rotation,** i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than –1, 0, or 1.

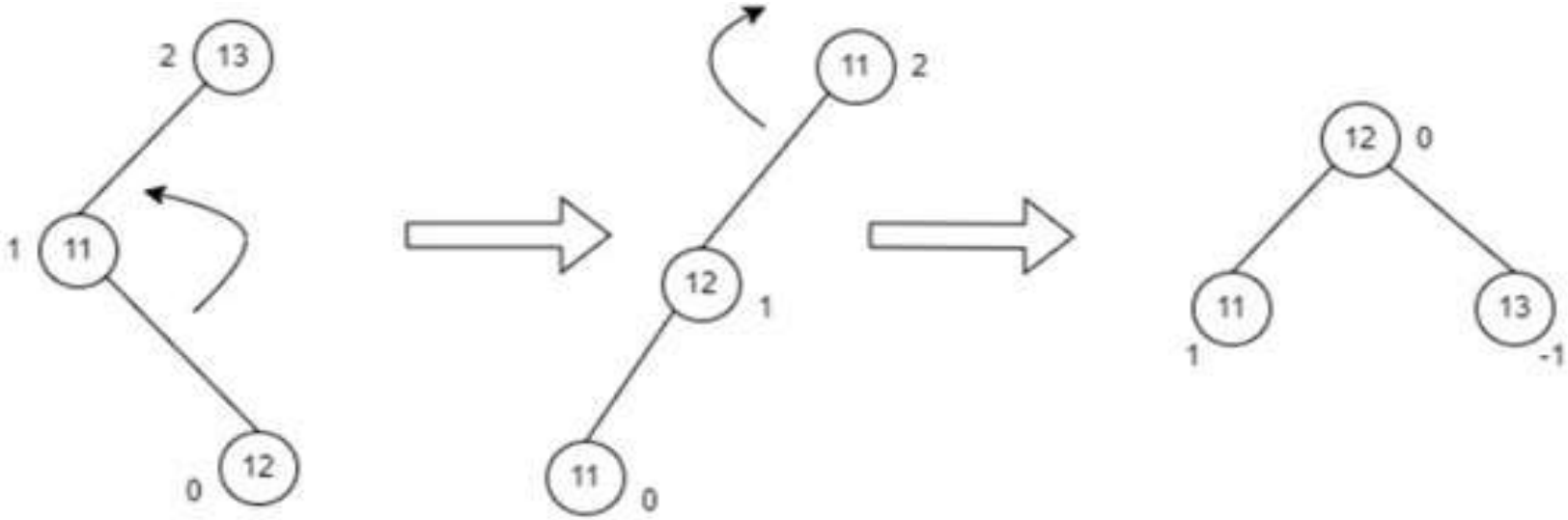
# LR-Rotation

| State | Action |
|---|---|
| | A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C |
| | As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**. |
|  | After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C** |

# LR-Rotation

| State | Action |
|---|---|
|  | Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B |
|  | Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now. |

# LR-Rotation

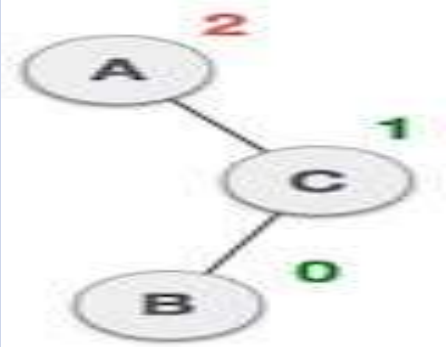
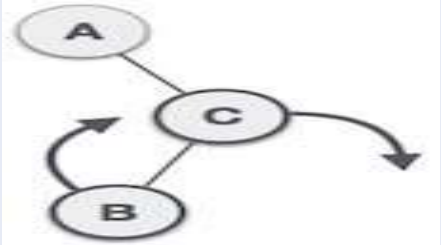
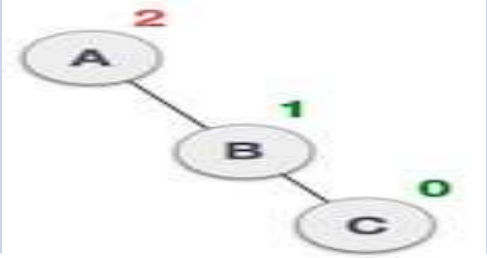



```
NODE *LR(NODE *node){
    node->left=RR(node->left);
    return LL(node);
}
```

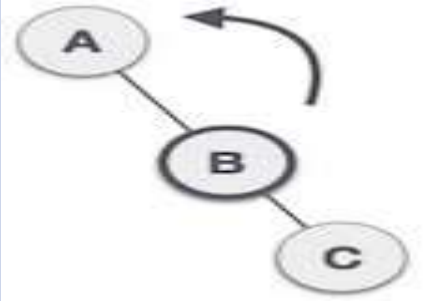
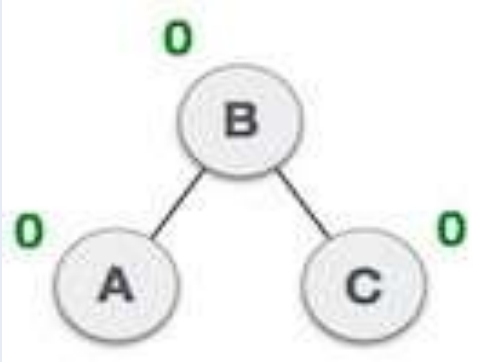
# RL-Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. **R L rotation = LL rotation + RR rotation,** i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

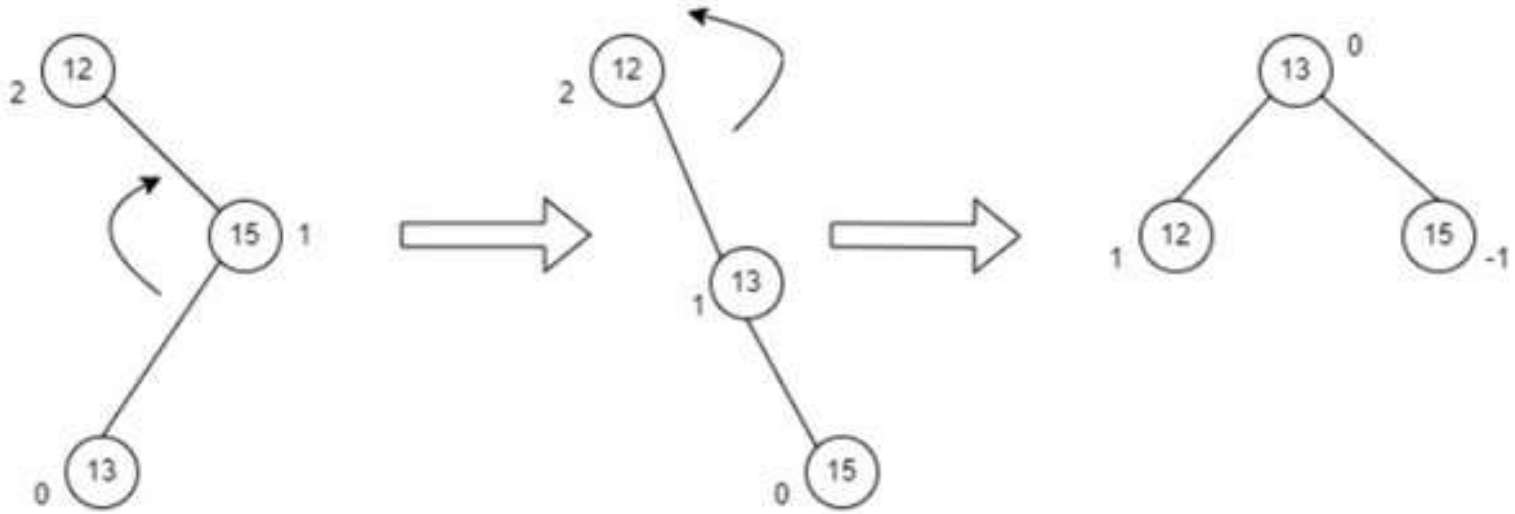
# RL-Rotation

| State | Action |
|---|---|
|  | A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor – 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A |
|  | As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**. |
|  | After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor –2, which is because of the right–subtree of the right–subtree node A. |

# RL-Rotation

| State | Action |
|---|---|
|  | Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B. |
|  | Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now. |

# RL-Rotation




```
NODE *RL(NODE *node){
    node->right=LL(node->right);
    return RR(node);
}
```

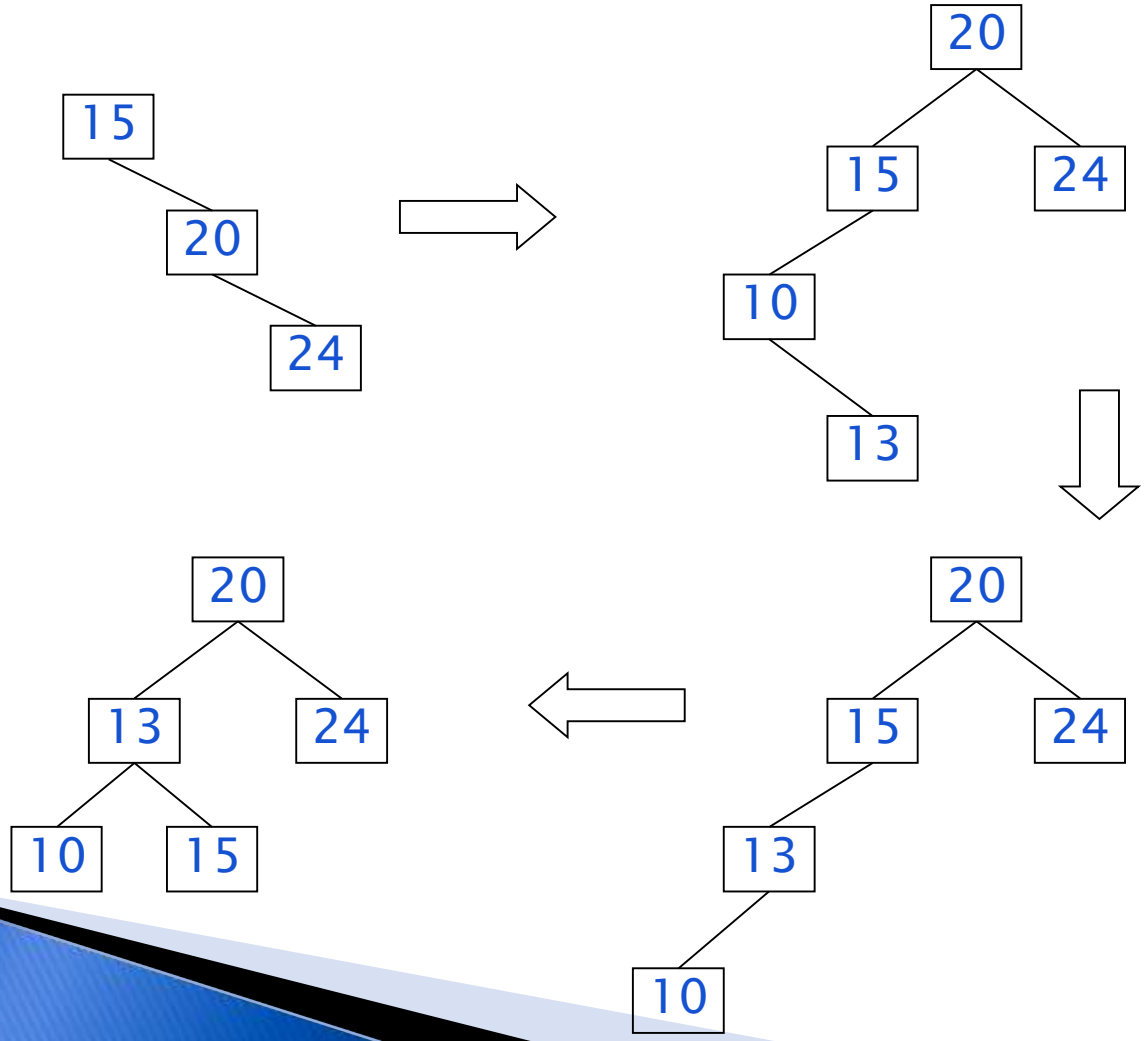
# Questions

| Questions | Final Tree |
| --- | --- |
| Build an AVL tree with the following values:<br>15, 20, 24, 10, 13, 7, 30, 36, 25 | Construct an AVL tree having the following elements:<br><br>40,20,10,25,30,22,50 |
| Construct an AVL tree having the following elements<br>. Insert 14, 17, 11, 7, 53, 4, 13,12,8 into an empty AVL tree | Construct an AVL tree having the following elements<br>55,66,77,15,11,33,22,35,25, 44,88,99 |

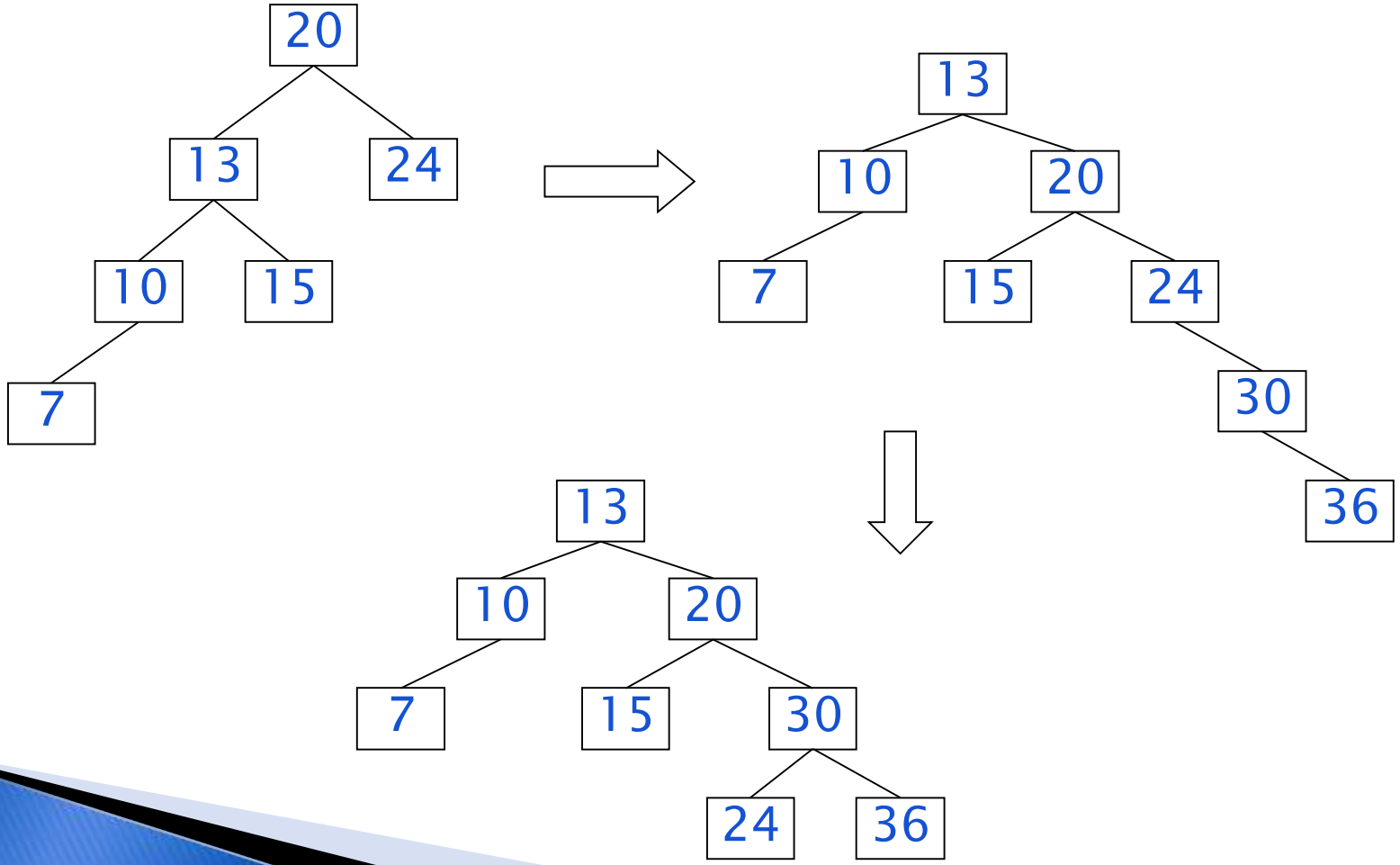
# Exercises

- Build an AVL tree with the following values:
  15, 20, 24, 10, 13, 7, 30, 36, 25

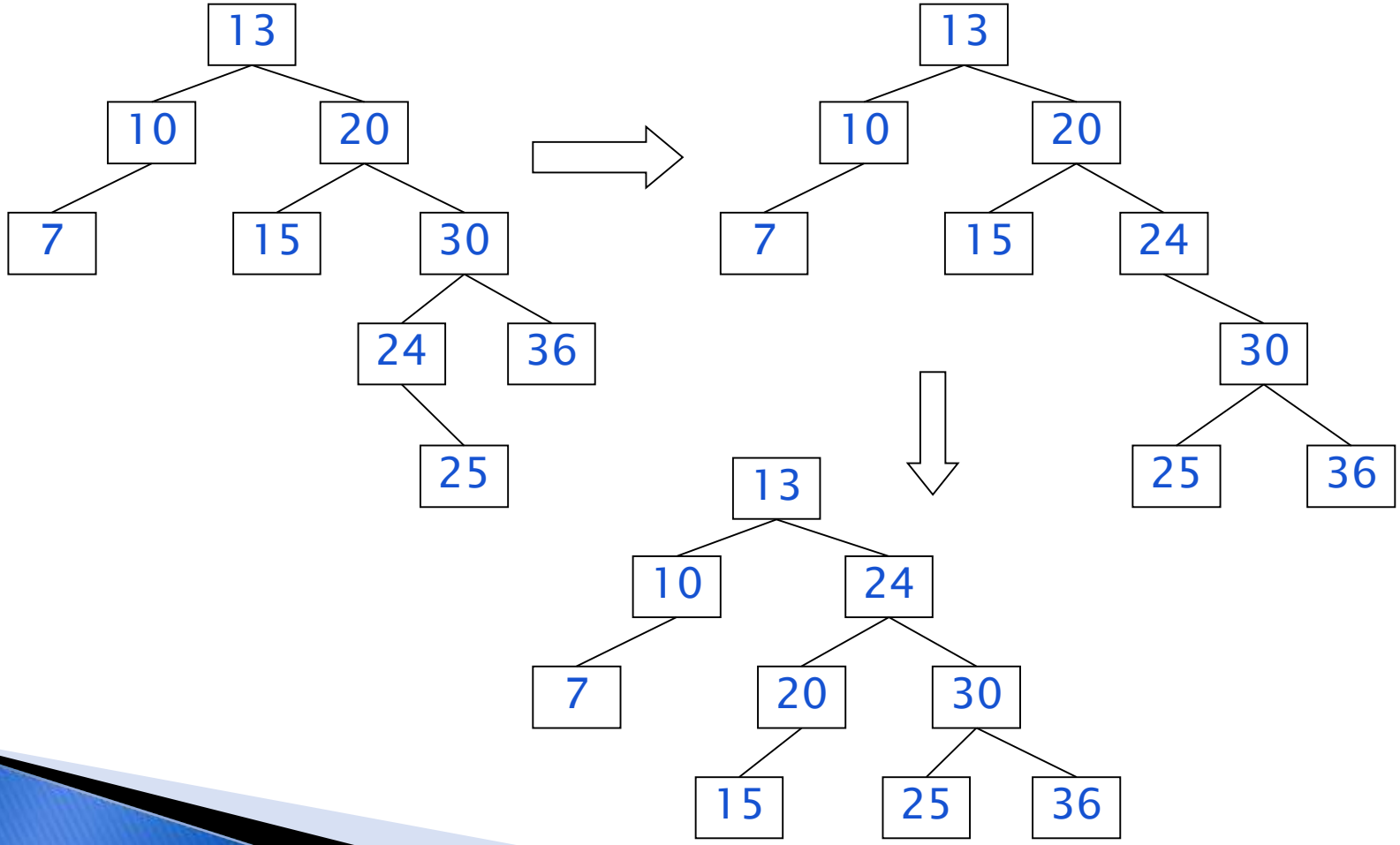
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25



# Insertion of a node into AVL Tree...

```
Insert (NODE *root, int key)
BEGIN
1.  IF root==NULL Then
2.      root=(NODE*) malloc(sizeof(NODE))
3.      root->data=key
4.      root->left=NULL
5.      root->right=NULL
6.      root->height=0
7.  ELSE IF key < root->data Then
8.      root->left=Insert(root->left, key)
9.      IF height(root->left) – height(root->right)==2 Then
10.         IF key<root->left->data Then
11.             root=LL(root)
12.         ELSE
13.             root=LR(root)
14. ELSE IF key > root->data Then
15.     root->right=Insert(root->right, key)
16.     IF height(root->right) – height(root->left)==2 Then
17.         IF key>root->right->data Then
18.             root=RR(root)
19.         ELSE
20.             root=RL(root)
21. root->height=findMax(height(root->left), height(root->right))+1
22. return root
END
```

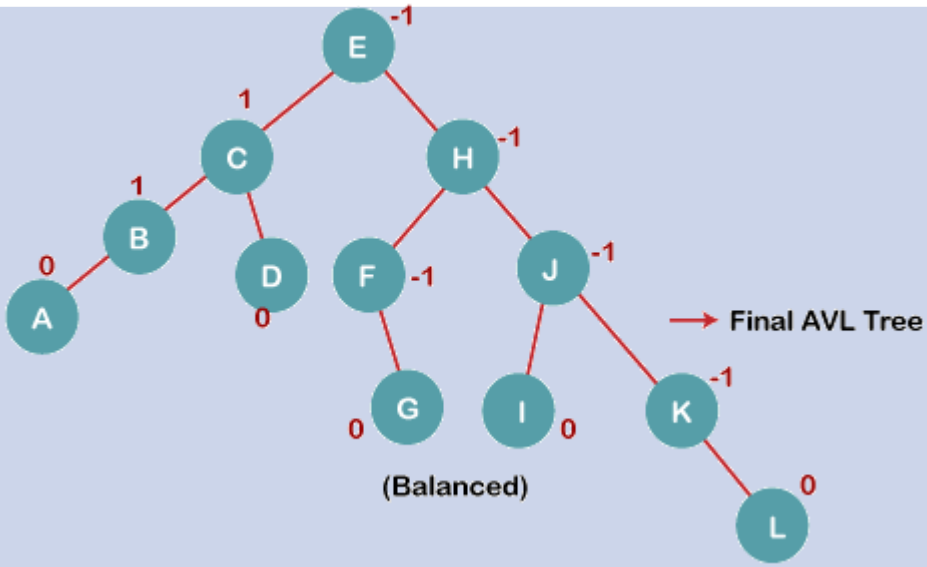
# Insertion of a node into AVL Tree...

```
NODE * insert(NODE * node, int x){
    if(node==NULL){
        node=(NODE *)malloc(sizeof(NODE));
        node->data=x;
        node->left=NULL;
        node->right=NULL;
        node->height=0;
    }
    else{
        if(x<node->data){
            node->left=insert(node->left,x);
            if(height(node->left) - height(node->right)==2)
                if (x<node->left->data)
                    node=LL(node);
                else
                    node=LR(node);
        }
        else if(x>node->data){
            node->right=insert(node->right,x);
            if(height(node->right) - height(node->left)==2)
                if(x>node->right->data)
                    node=RR(node);
                else
                    node=RL(node);
            }
    }
    node->height=findMax(height(node->left), height(node->right))+1;
    return node;
}
```

# Insertion of a node into AVL Tree...

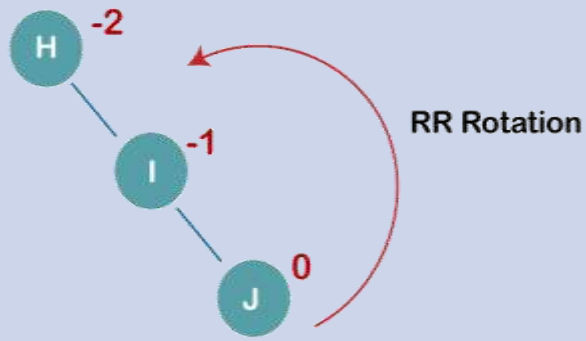
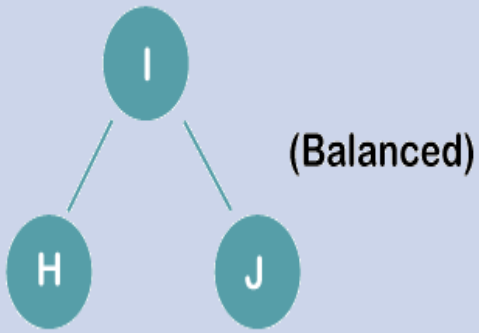
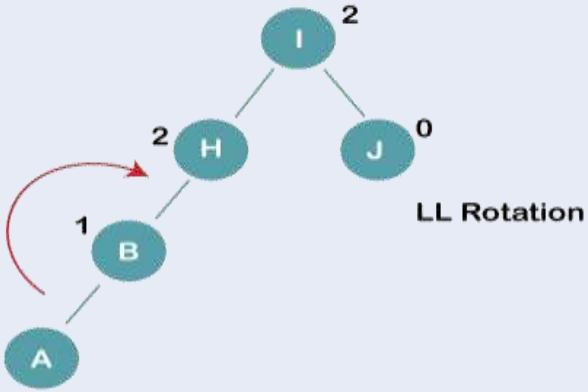
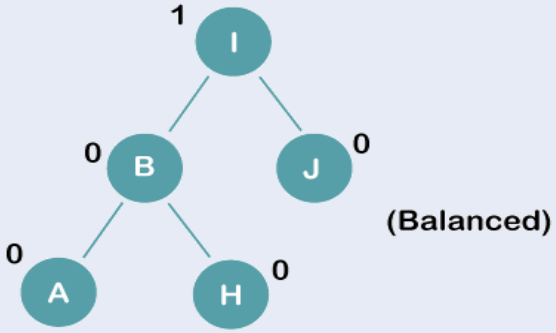
```
NODE *delete(NODE * ptr, int key){
    if(ptr==NULL)
        return NULL;
    if(ptr->left==NULL && ptr->right==NULL){
        if(ptr==root)
            root=NULL;
        free(ptr);
        return NULL;
    }
    if(key<ptr->data)
        ptr->left=delete(ptr->left, key);
    else if(key>ptr->data)
        ptr->right=delete(ptr->right, key);
    else{
        if(height(ptr->left)>height(ptr->right)){
            NODE *inpred=inorderPredecessor(ptr->left);
            ptr->data=inpred->data;
            ptr->left=delete(ptr->left, inpred->data);
        }
        else{
            NODE *insuc=inorderSuccessor(ptr->right);
            ptr->data=insuc->data;
            ptr->right=delete(ptr->right, insuc->data);
        }
    }
    ptr->height=height(ptr);
    if(balanceFactor(ptr)==2 && balanceFactor(ptr->left)>=0)
        return LL(ptr);
    else if(balanceFactor(ptr)==2 && balanceFactor(ptr->left)==-1)
        return LR(ptr);
    else if(balanceFactor(ptr)==2 && balanceFactor(ptr->right)>=0)
        return RR(ptr);
    else if(balanceFactor(ptr)==2 && balanceFactor(ptr->right)==-1)
        return RL(ptr);
    return ptr;
}
```

# Questions

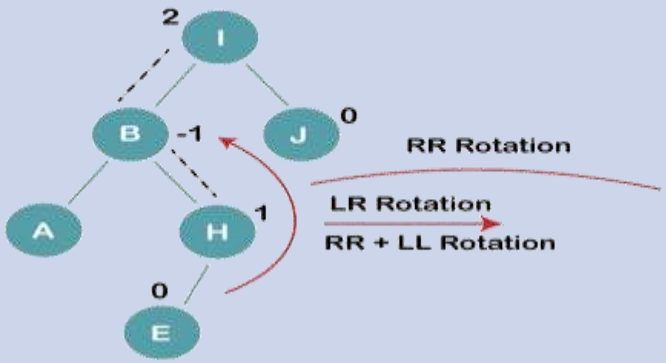
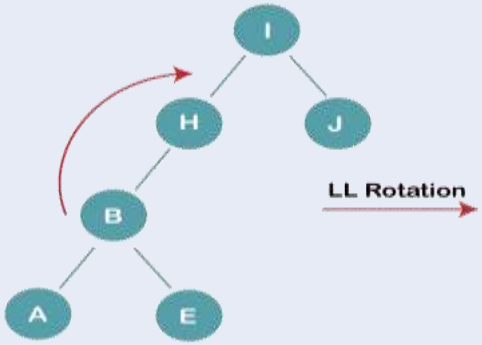
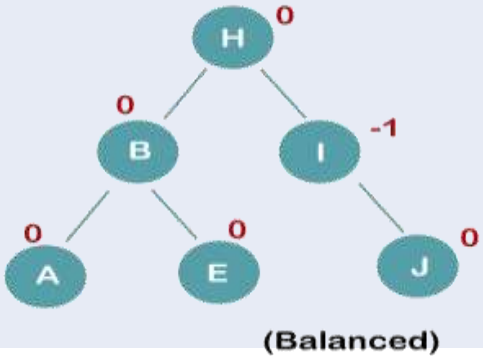
| Questions | Final Tree |
| --- | --- |
| Construct an AVL tree having the following elements<br>**H, I, J, B, A, E, C, F, D, G** |  |

Link: https://www.javatpoint.com/avl-tree

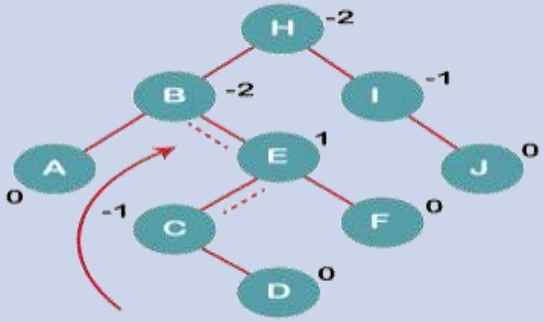
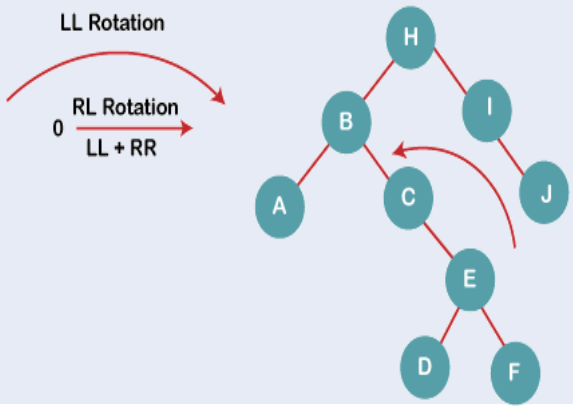
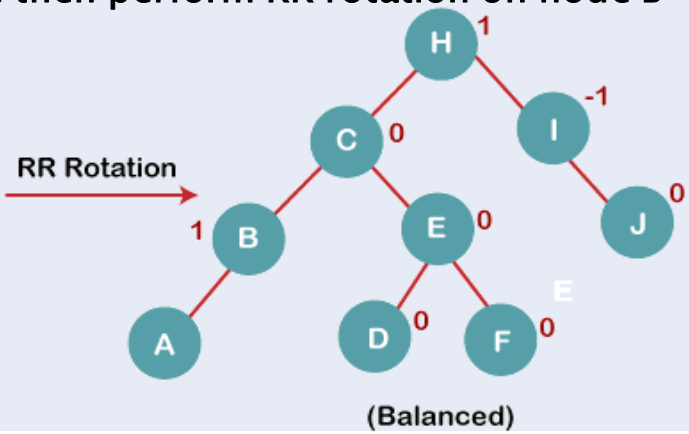
# Steps

| | |
|---|---|
| Insert H,I,J   |   |
| Insert B,A   |   |

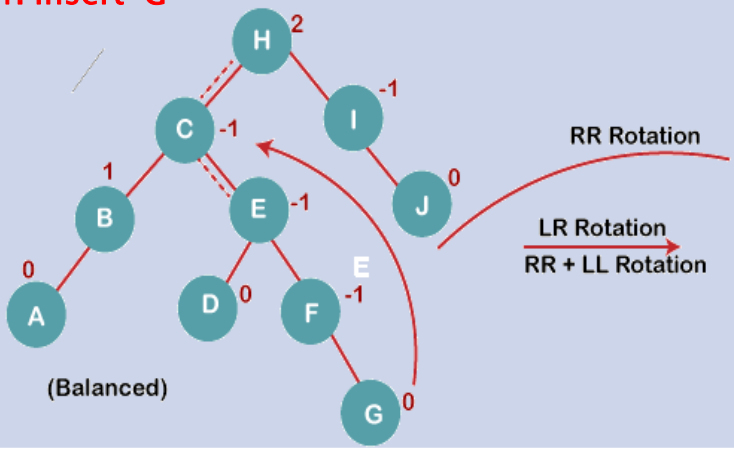
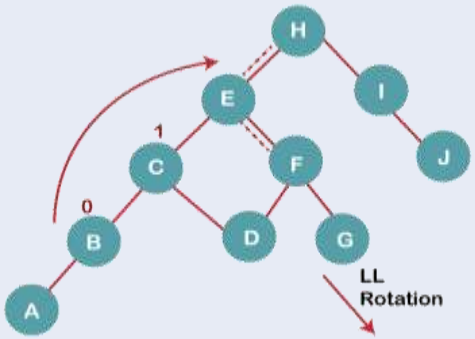
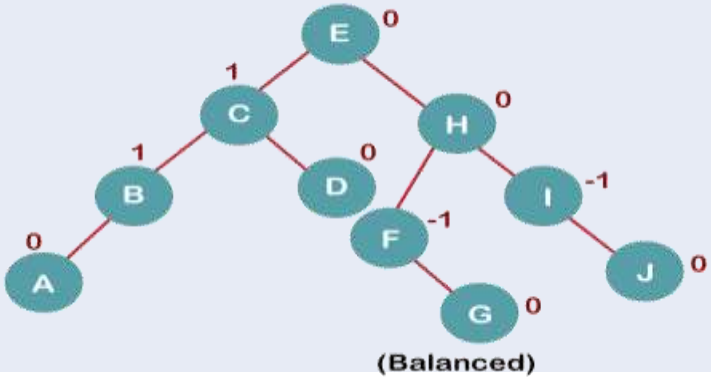
# Steps

| | |
|---|---|
| Insert E   | On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation |
| Step-1:**We first perform RR rotation on node B**   | Step-2:**thenperform LL rotation on the node I**   |

# Steps

| | |
|---|---|
| 4. Insert C, F, D   | On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is –2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation. |
| 4a) We first perform LL rotation on node E   | 4b) We then perform RR rotation on node B   |

# Steps

| | |
|---|---|
| 4. Insert G   | On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation. |
| 5 a) We first perform RR rotation on node C   | 5 b) We then perform LL rotation on node H   |

# *Thank You*

Any Questions???