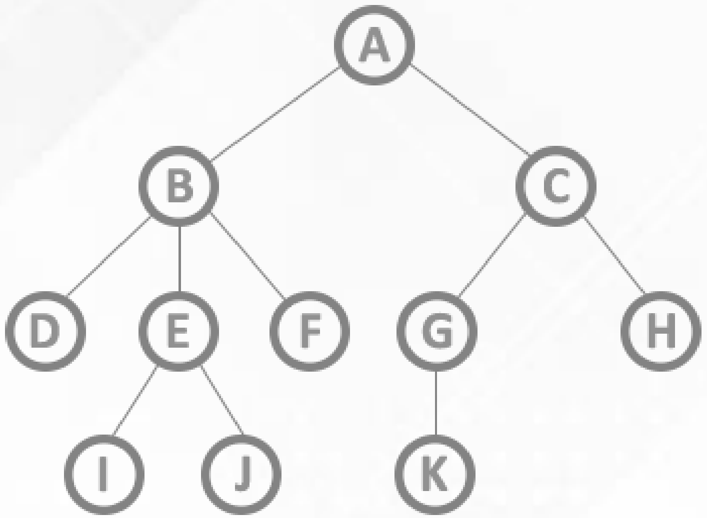
# Unit-6

# Non-Linear Data Structure

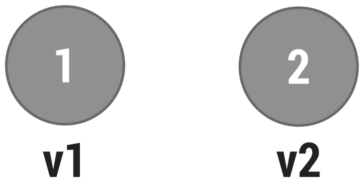
# Tree Part-1



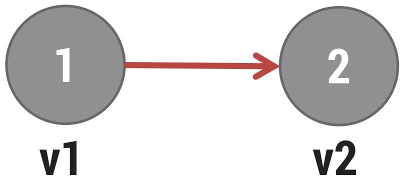

**Asst. Prof. Kumar Prasun**
Computer Application Department
Padmakanya Multiple Campus, Baghbazar

Kumar.prasun@pkmc.tu.edu.np
+9779851149487

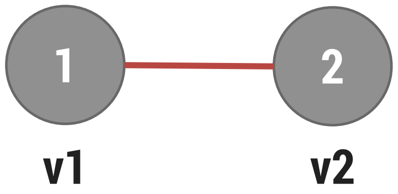
# Basic Notations of Graph Theory



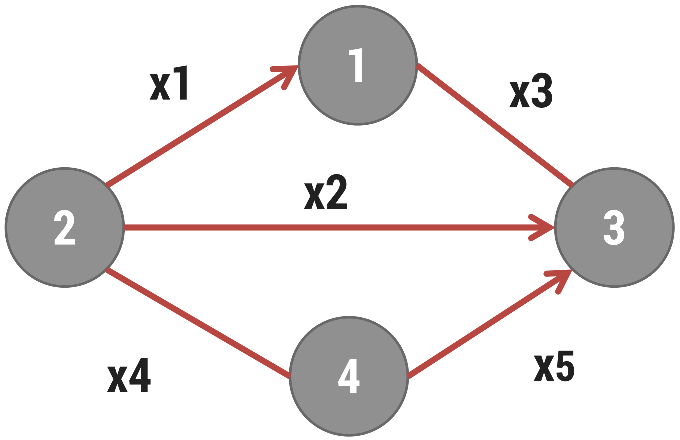

(a)



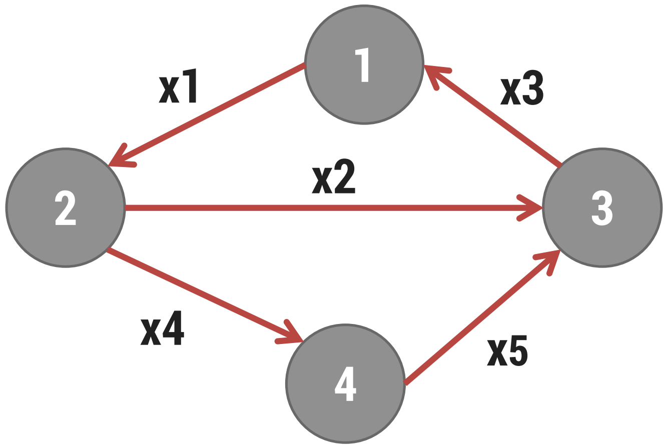

(b)



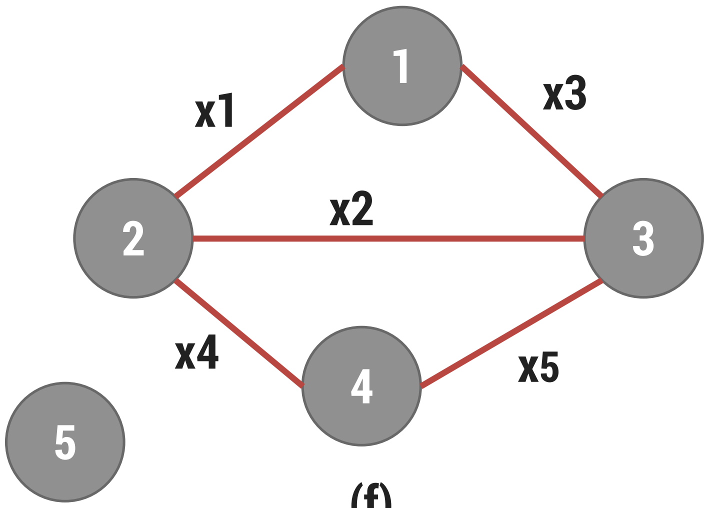

(c)



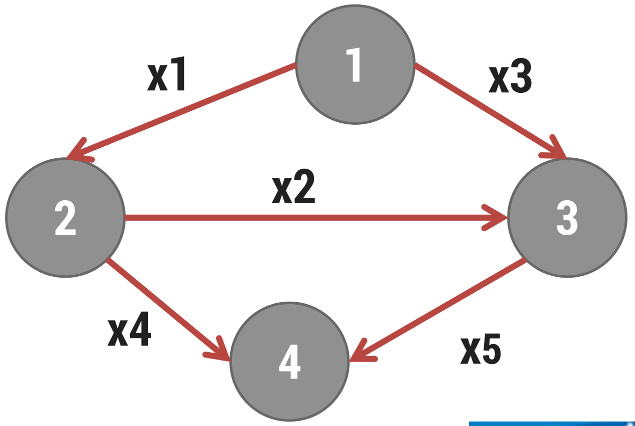

(d)




(e)




(f)




(g)

# Basic Notations of Graph Theory

- Consider diagrams shown in above figure
- Every diagrams represent Graphs
- Every diagram consists of a **set of points** which are shown by **dots** or **circles** and are sometimes labelled $V_1$, $V_2$, $V_3$... OR 1,2,3...
- In every diagrams, certain pairs of such points are connected by lines or arcs
- Note that every arc start at one point and ends at another point

# Basic Notations of Graph Theory

## Graph

- **A graph G** consist of a **non-empty set V** called the **set of nodes** (points, vertices) of the graph, a **set E** which is the **set of edges** and a **mapping** from the set of edges **E to** a set of **pairs of elements of V**
- It is also convenient to write a graph as **G=(V,E)**
- Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X Є E is thus associated with a pair of nodes (u,v) where u, v Є V then we says that edge x connect u and v

## Adjacent Nodes

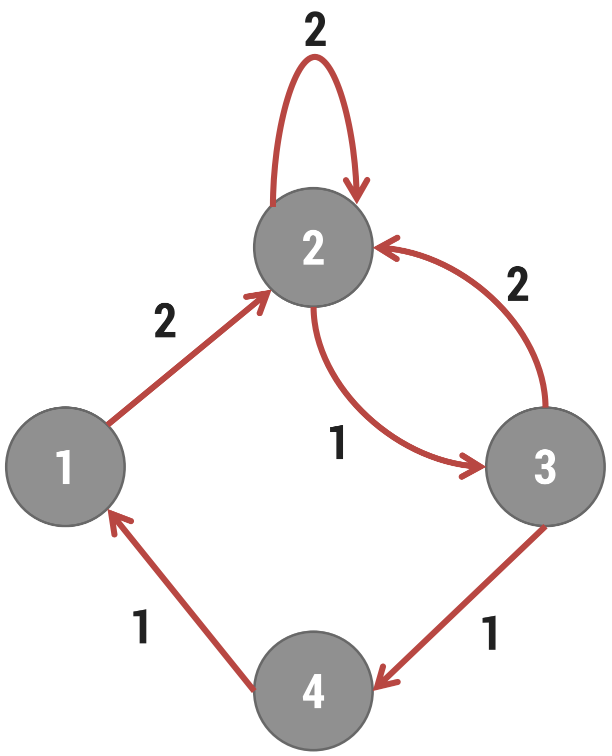
- Any two nodes which are connected by an edge in a graph are called adjacent nodes

# Graph – Concepts & Definitions

- **Directed & Undirected Edge**
  - In a graph **G=(V,E)** an **edge** which is **directed** from one end to another end is called a **directed edge**, while the edge which has no specific direction is called **undirected edge**
- **Directed graph (Digraph)**
  - A graph in which **every edge is directed** is called directed graph or digraph e.g. **b,e & g** are directed graphs
- **Undirected graph**
  - A graph in which **every edge is undirected** is called undirected graph e.g. **c & f** are undirected graphs
- **Mixed Graph**
  - If **some** of the **edges** are **directed** and **some are undirected** in graph then the graph is called mixed graph e.g. **d** is mixed graph

# Graph – Concepts & Definitions

- **Loop (Sling)**
  - An **edge** of a graph **which joins a node to itself** is called a loop (sling).
  - The ***direction of a loop is of no significance*** so it can be considered either a directed or an undirected.
- **Distinct Edges**
  - In case of directed edges, **two possible edges** between any pair of nodes which **are opposite in direction** are considered **Distinct**.
- **Parallel Edges**
  - In some directed as well as undirected graphs, we may have **certain pairs of nodes joined by more than one edges**, such edges are called **Parallel** edges.

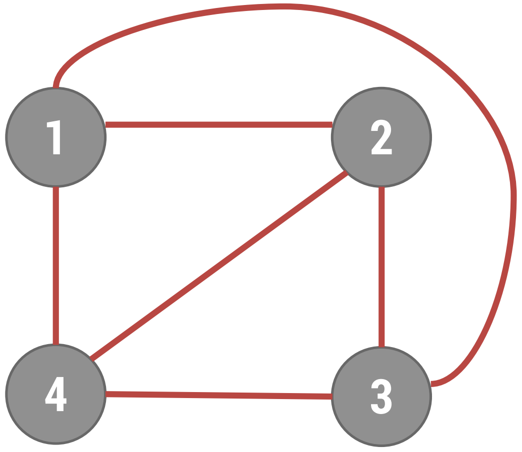


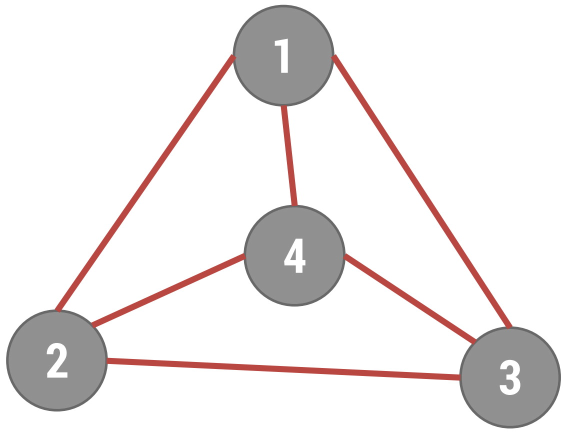
# Graph – Concepts & Definitions

- **Multigraph**
  - Any **graph** which **contains** some **parallel edges** is called **multigraph**
  - If there is no more then one edge between a pair of nodes then such a graph is called **Simple graph**
- **Weighted Graph**
  - A graph in which **weights are assigned to every edge** is called weighted graph
- **Isolated Node**
  - In a graph a **node** which is **not adjacent to any other node** is called isolated node
- **Null Graph**
  - A graph **containing only isolated nodes** are called null graph. In other words set of edges in null graph is empty

# Graph – Concepts & Definitions

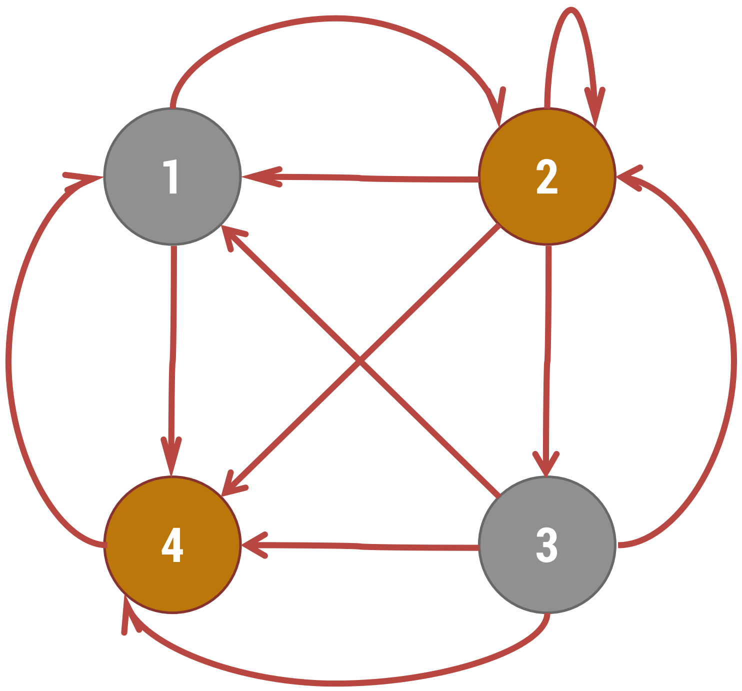
- For a given **graph** there is **no unique diagram** which represents the graph.
- We can obtain a variety of diagrams by locating the nodes in an arbitrary numbers.
- Following both diagrams represents same Graph.




- **Indegree of Node**
  - The **no of edges** which have **V as their terminal node** is call as indegree of node V.
- **Outdegree of Node**
  - The **no of edges** which have **V as their initial node** is call as outdegree of node V.
- **Total degree of Node**
  - Sum of indegree and outdegree of node V is called its Total Degree or Degree of vertex.

# Path of the Graph




Some of the path from 2 to 4

**P1** = ((2,4))
**P2** = ((2,3), (3,4))
**P3** = ((2,1), (1,4))
**P4** = ((2,3), (3,1), (1,4))
**P5** = ((2,3), (3,2), (2,4))
**P6** = ((2,2), (2,4))

- Let G=(V, E) be a simple digraph such that the terminal node of any edge in the sequence is the initial node of the edge, if any appearing next in the sequence defined as **path of the graph**.
- **Length of Path**
  - The number of edges appearing in the sequence of the path is called length of path.

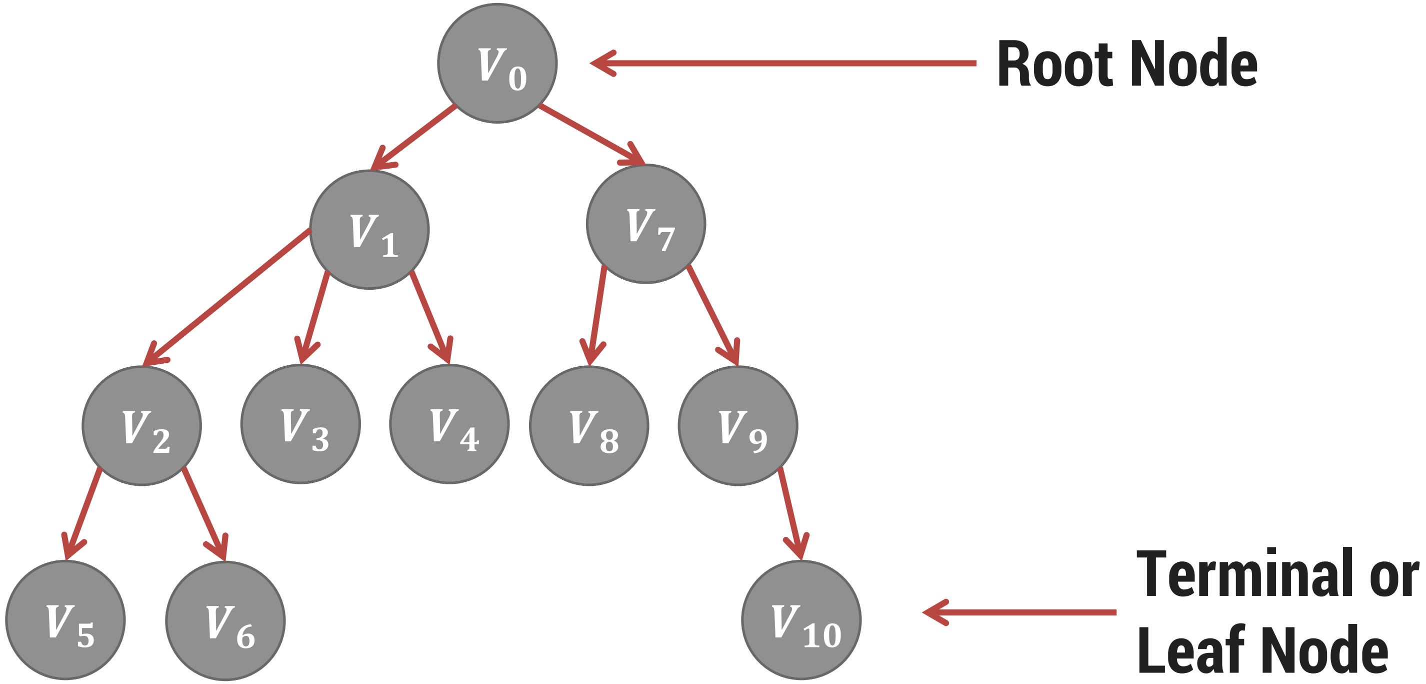
# Graph – Concepts & Definitions

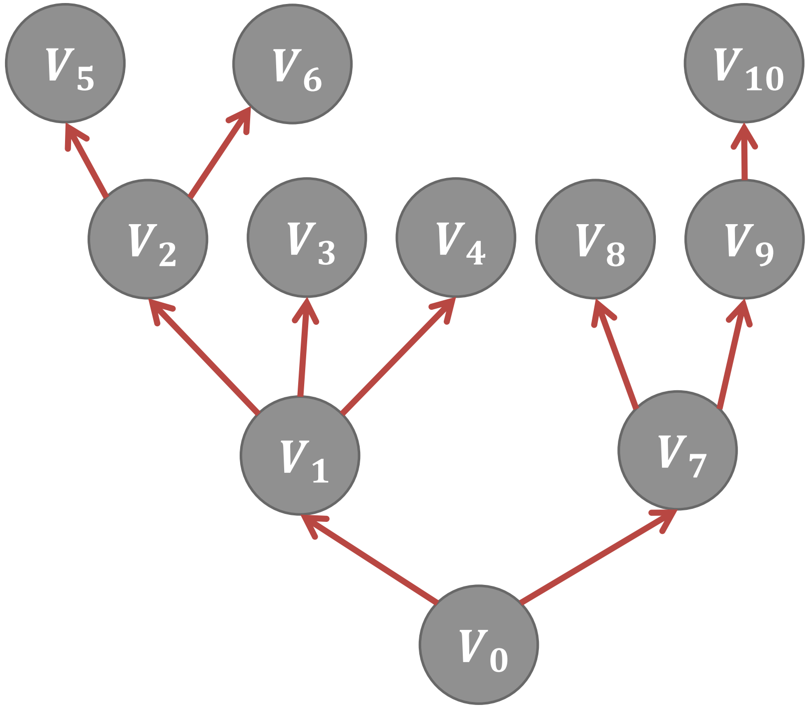
- **Simple Path (Edge Simple)**
  - A **path** in a diagraph in which **the edges are distinct** is called simple path or edge simple
  - Path P5, P6 are Simple Paths
- **Elementary Path (Node Simple)**
  - A **path** in which **all the nodes through which it traverses** are **distinct** is called elementary path
  - Path P1, P2, P3 & P4 are elementary Path
  - Path P5, P6 are Simple but not Elementary
- **Cycle (Circuit)**
  - A **path** which **originates and ends in the same node** is called cycle (circuit)
  - E.g. C1 = ((2,2)), C2 = ((1,2),(2,1)), C3 = ((2,3), (3,1), (1,2)
- **Acyclic Diagraph**
  - A simple **diagraph which does not have any cycle** is called Acyclic Diagraph.

# Tree– Concepts & Definitions

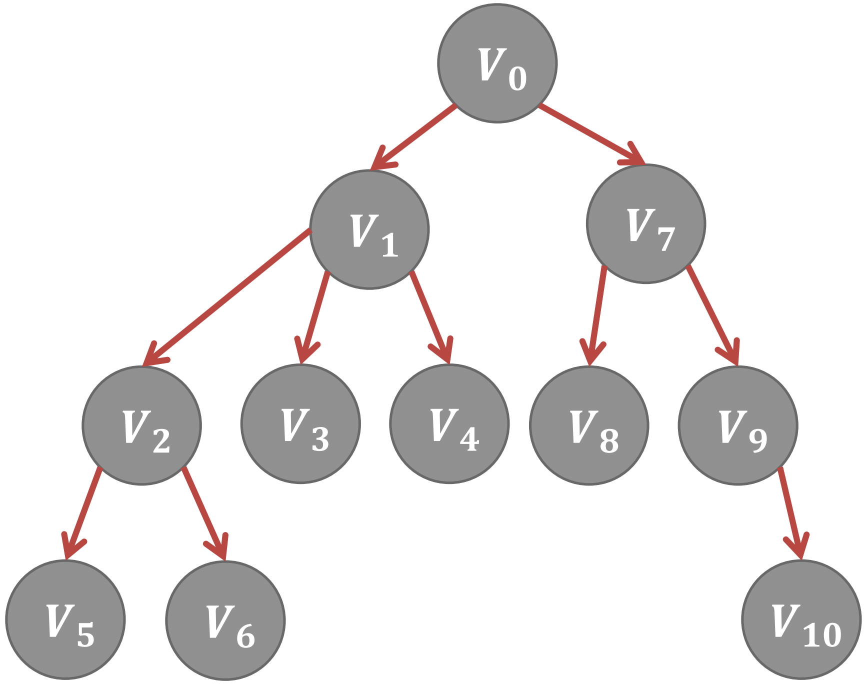
## Directed Tree

- A directed tree is an acyclic digraph which has one node called its root with in degree 0, while all other nodes have in degree 1.
- Every directed tree must have at least one node.
- An isolated node is also a directed tree.

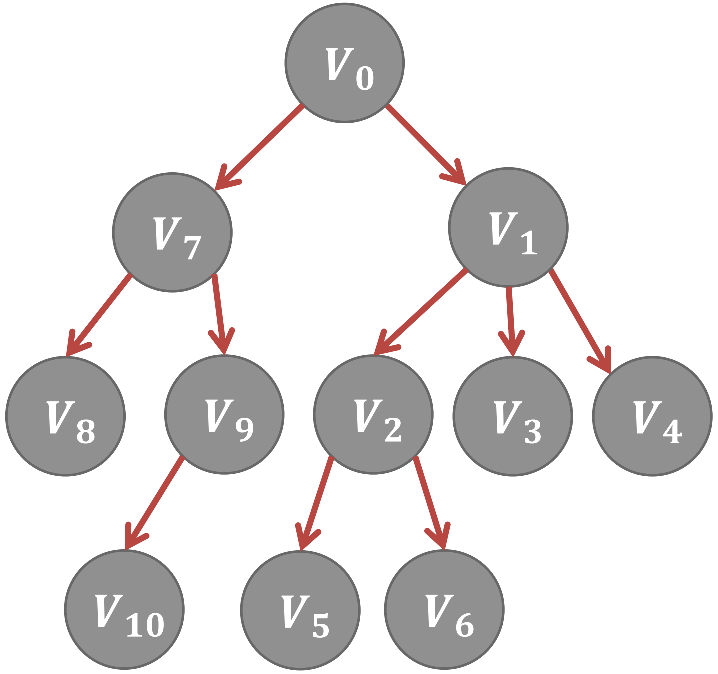



$V_5$
$V_6$
$V_{10}$
$V_2$
$V_3$
$V_4$
$V_8$
$V_9$
$V_1$
$V_7$
$V_0$

**(a)**


$V_0$
$V_1$
$V_7$
$V_2$
$V_3$
$V_4$
$V_8$
$V_9$
$V_5$
$V_6$
$V_{10}$

**(b)**


$V_0$
$V_7$
$V_1$
$V_8$
$V_9$
$V_2$
$V_3$
$V_4$
$V_{10}$
$V_5$
$V_6$

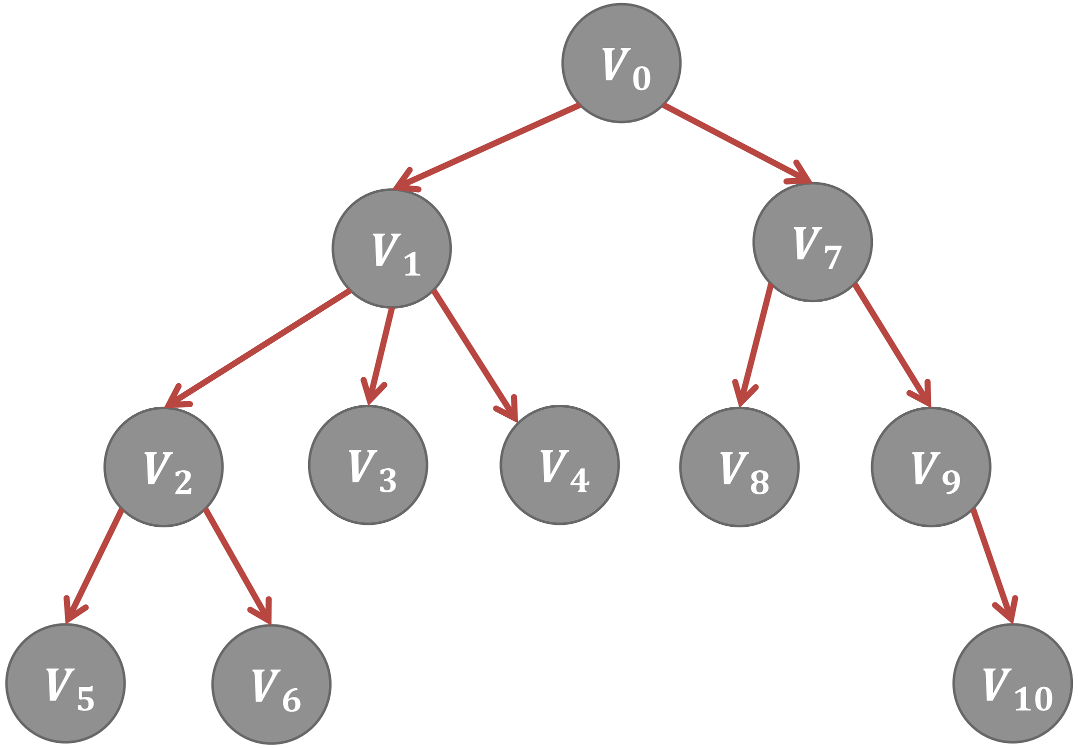
**(c)**

# Tree– Concepts & Definitions

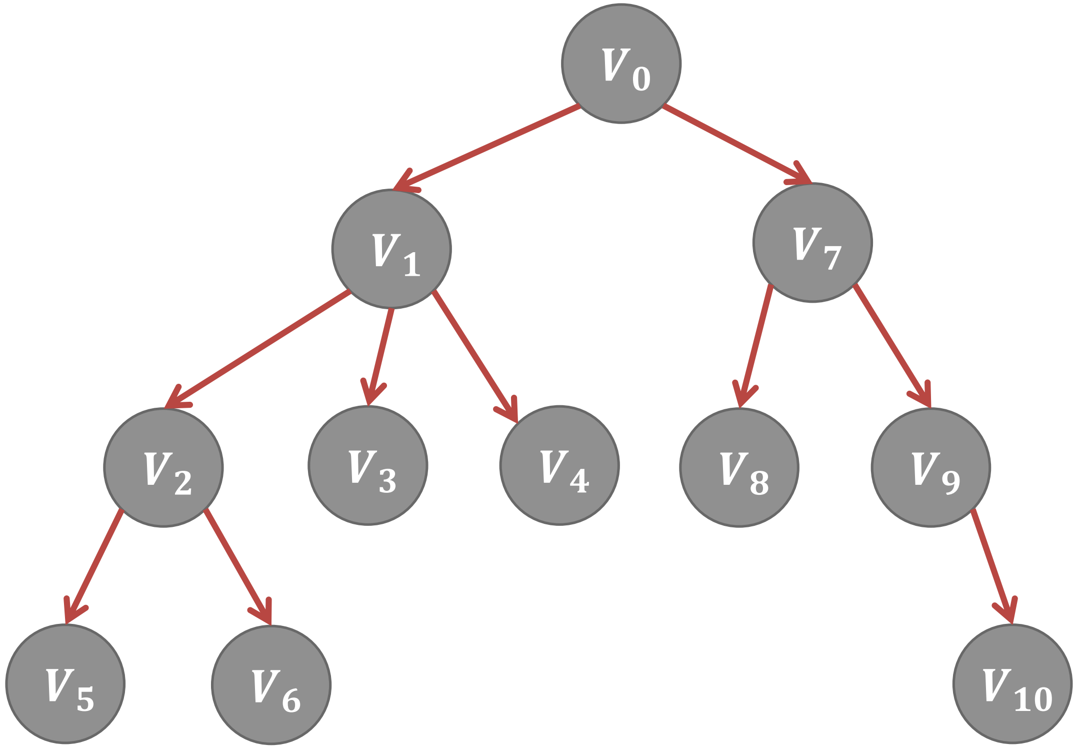
- **Terminal Node (Leaf Node)**
  - In a directed tree, any **node** which **has out degree 0** is called terminal node or leaf node.
- **Level of Node**
  - The level of any node is the length of its path from the root.
- **Ordered Tree**
  - In a directed tree an ordering of the nodes at each level is prescribed then such a tree is called ordered tree.
  - The diagrams (b) and (c) represents same directed tree but different ordered tree.
- **Forest**
  - If we delete the root and its edges connecting the nodes at level 1, we obtain a set of disjoint tree. A set of disjoint tree is a forest.

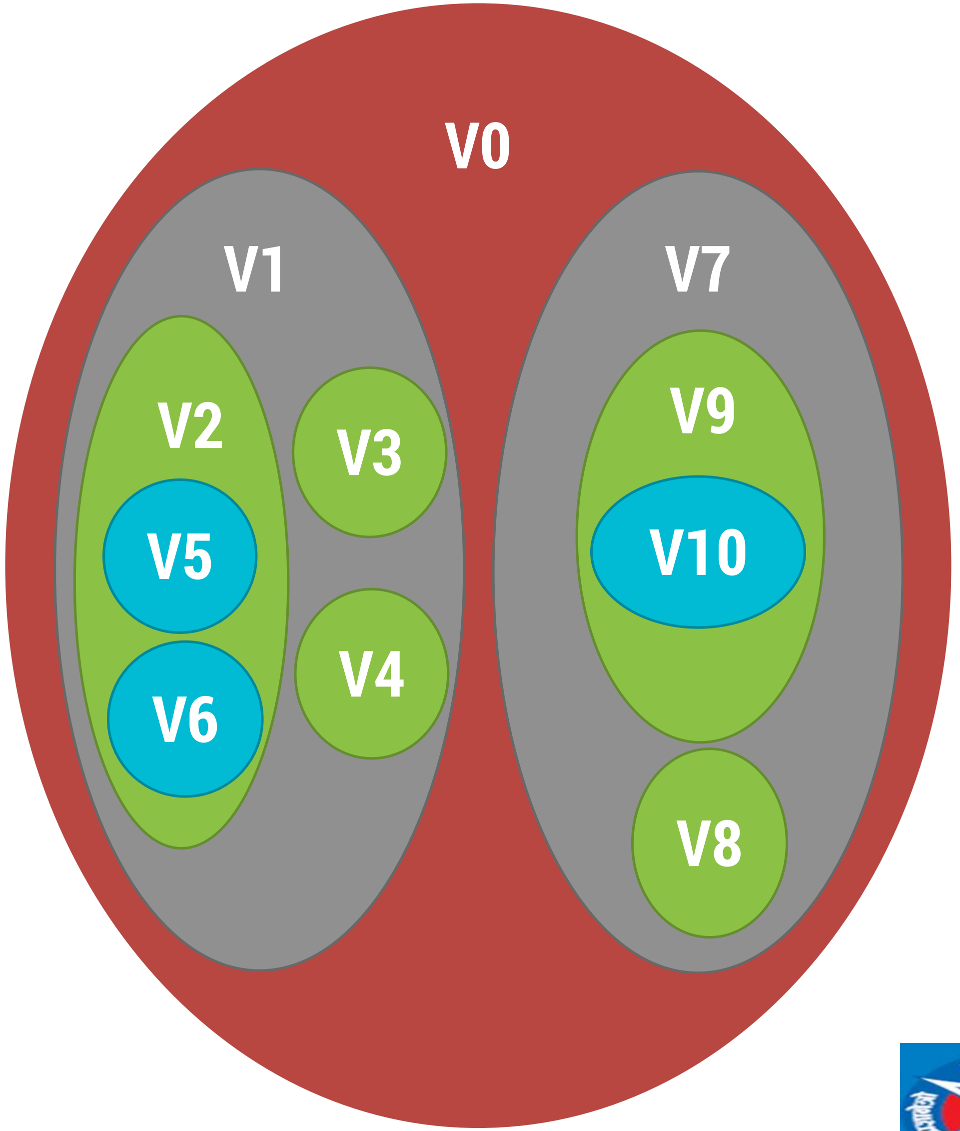
# Representation of Directed Tree

- Other way to represent directed tree are
  - Venn Diagram
  - Nesting of Parenthesis
  - Like table content of Book
  - Level Format

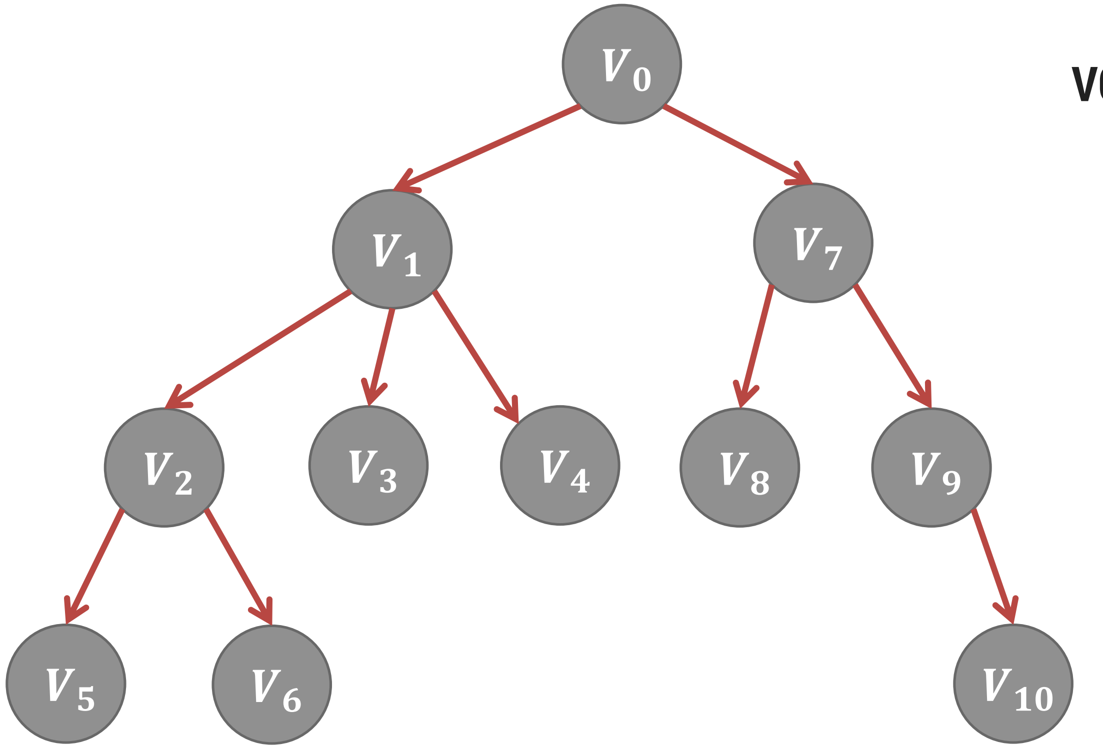


# Venn Diagram

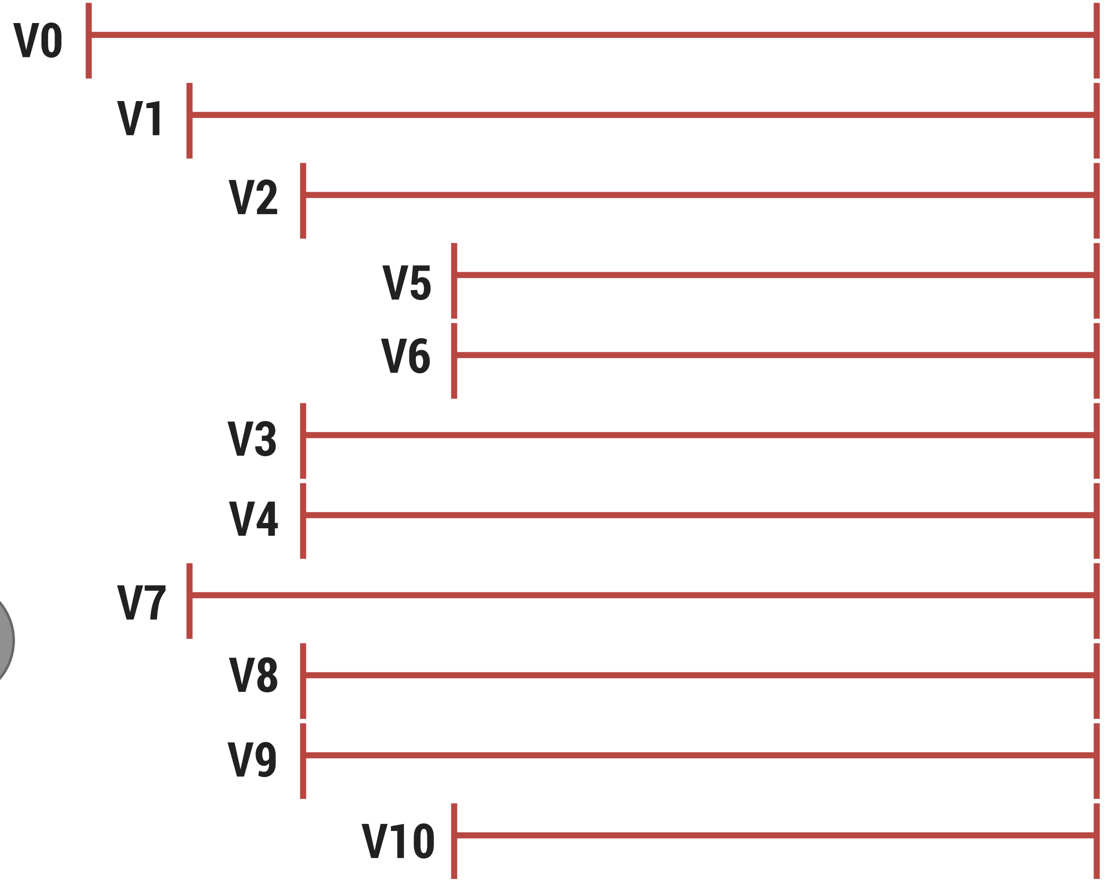


# Nesting of Parenthesis



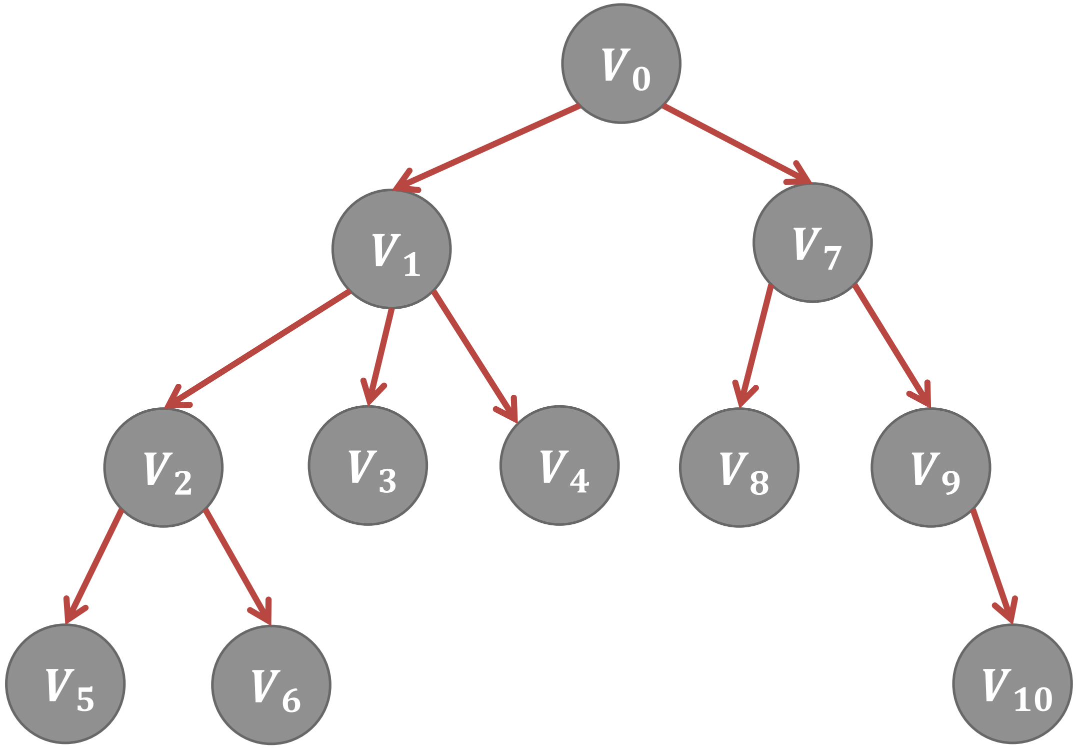

Like a table Content of Book




$$(V_0 \ (V_1 \ (V_2 \ (V_5) \ (V_6) \ ) \ (V_3) \ (V_4) \ ) \ (V_7 \ (V_8) \ (V_9 \ (V_{10}) \ ) \ ) \ )$$

Nesting of Parenthesis

# Level Format




1 $V_0$
2 $V_1$
3 $V_2$
4 $V_5$
4 $V_6$
3 $V_3$
3 $V_4$
2 $V_7$
3 $V_8$
3 $V_9$
4 $V_{10}$

# Tree– Concepts & Definitions

- The node that is reachable from a node is called **descendant** of a node.
- The nodes which **are reachable from a node through a single edge** are called the **children of node**.
- **M-ary Tree**
  - If in a directed tree the **out degree of every node** is **less than or equal to m** then tree is called an m-ary tree.
- **Full or Complete M-ary Tree**
  - If **the out degree of each and every node** is **exactly equal to m or 0** and their **number of nodes at level i is m(i-1)** then the tree is called a full or complete m-ary tree.
- **Positional M-ary Tree**
  - If we consider m-ary trees in which the m children of any node are assumed to have m distinct positions, if such positions are taken into account, then tree is called positional m-ary tree.

# Tree– Concepts & Definitions

## Height of the tree

- The height of a tree is the length of the path from the root to the deepest node in the tree.

## Binary Tree

- If in a directed tree the **out degree of every node** is **less than or equal to 2** then tree is called binary tree.

## Strictly Binary Tree

- A strictly binary tree (sometimes proper binary tree or 2-tree or full binary tree) is a tree in **which every node other than the leaves has two children.**

## Complete Binary Tree

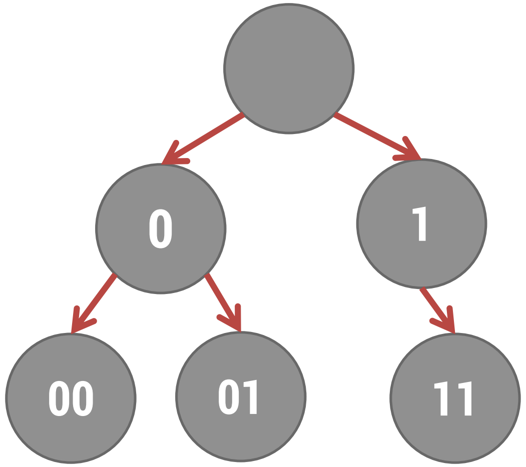
- If the **out degree of each and every node is exactly equal to 2 or 0** and **their number of nodes at level i is 2(i-1)** then the tree is called a full or complete binary tree.

## Sibling

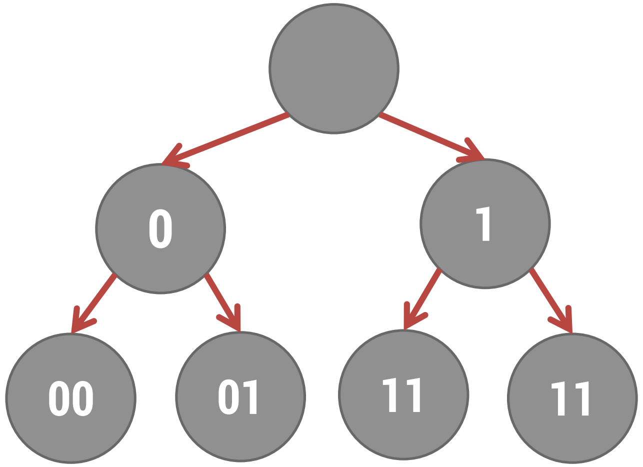
- Siblings are nodes that share the same parent node

## Positional m-ary Tree

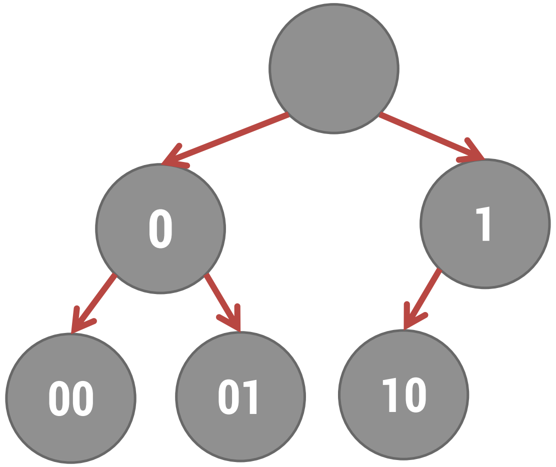
- If we consider m-ary trees in which the **m children of any node** are assumed **to have m distinct positions**, if such positions are taken into account, then tree is called positional m-ary tree




(a) Binary tree



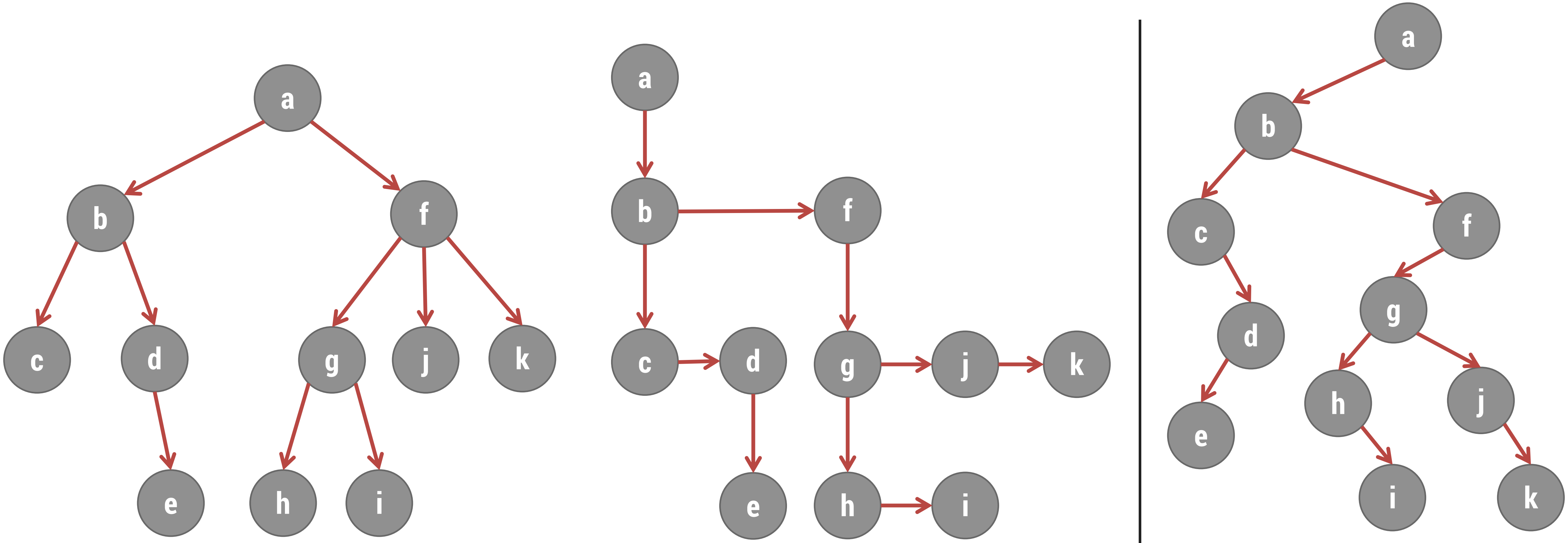

(b) Complete binary tree



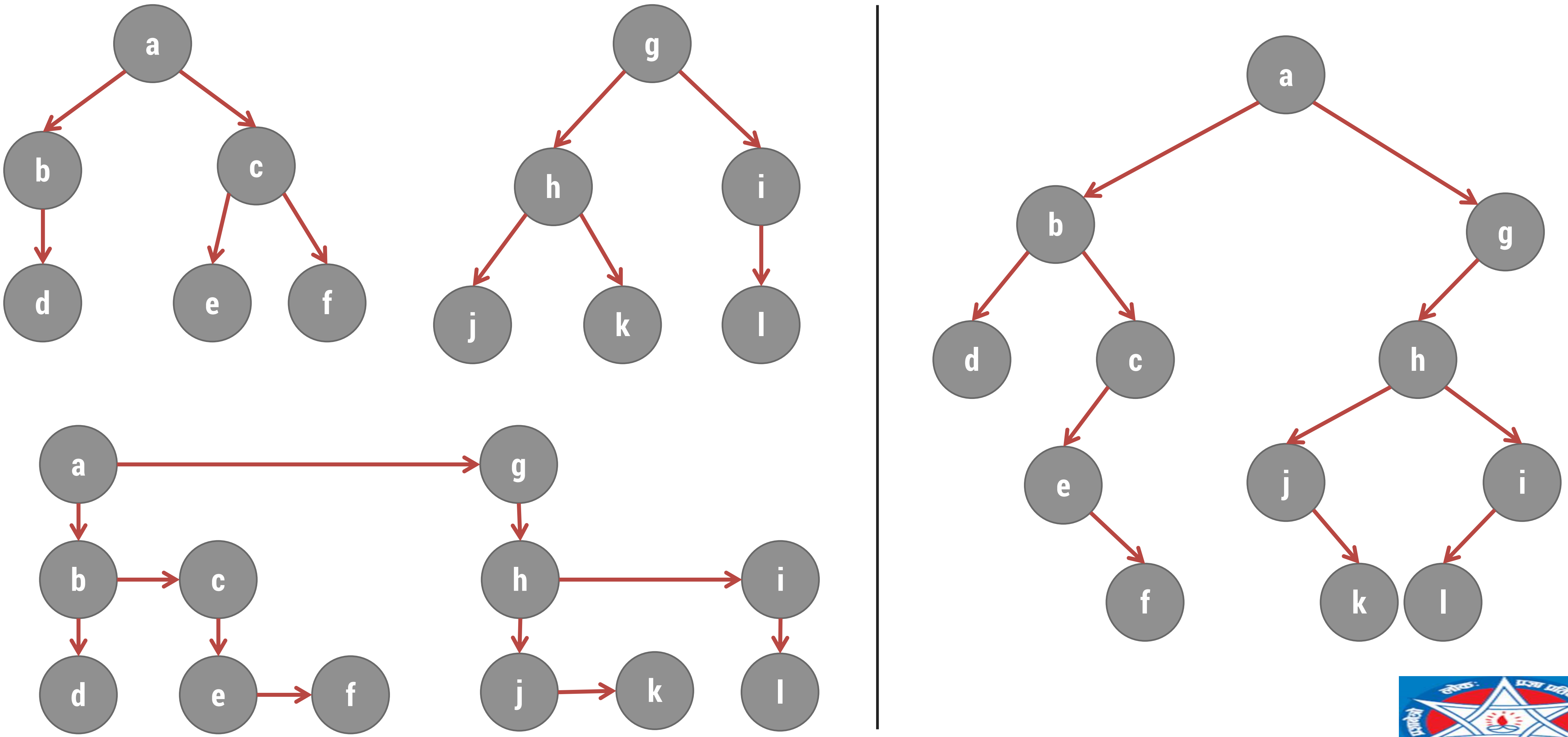

(c)

# Convert any tree to Binary Tree

- Every Tree can be Uniquely represented by binary tree
- Let’s have an example to convert given tree into binary tree

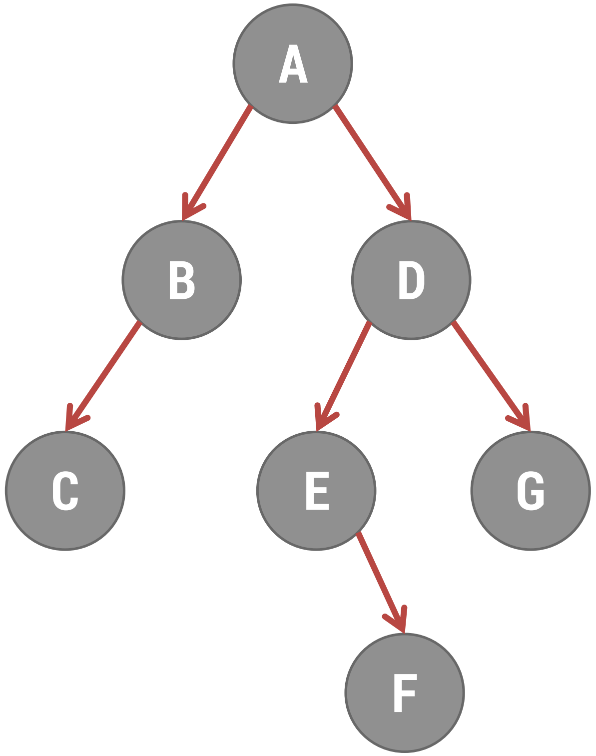


# Convert Forest to Binary Tree



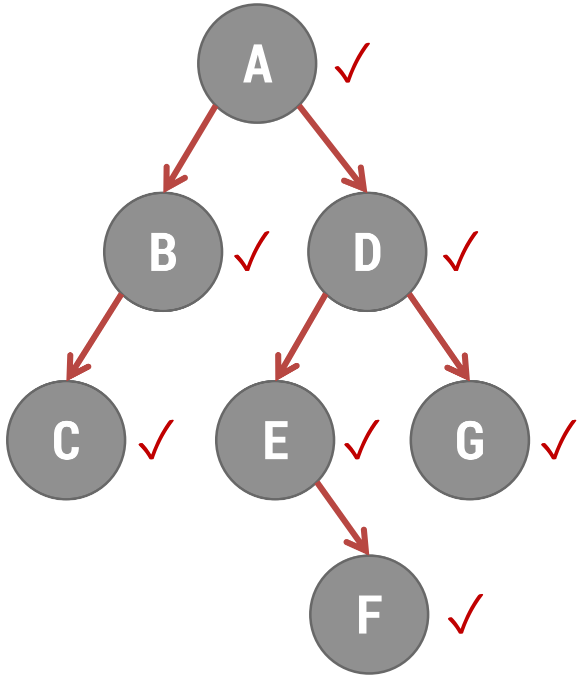
# Tree Traversal

- The most common operations performed on tree structure is that of traversal.
- This is a **procedure by which each node in the tree is processed exactly once** in a systematic manner.
- There are three ways of traversing a binary tree.
    1. Preorder Traversal
    2. Inorder Traversal
    3. Postorder Traversal



# Preorder Traversal

- Preorder traversal of a binary tree is defined as follow
  1. **Process** the **root node**
  2. **Traverse** the **left subtree** in preorder
  3. **Traverse** the **right subtree** in preorder
- If particular **subtree is empty** (i.e., node has no left or right descendant) the traversal is performed by **doing nothing.**
- In other words, a **null subtree** is **considered to be fully traversed** when it is encountered.

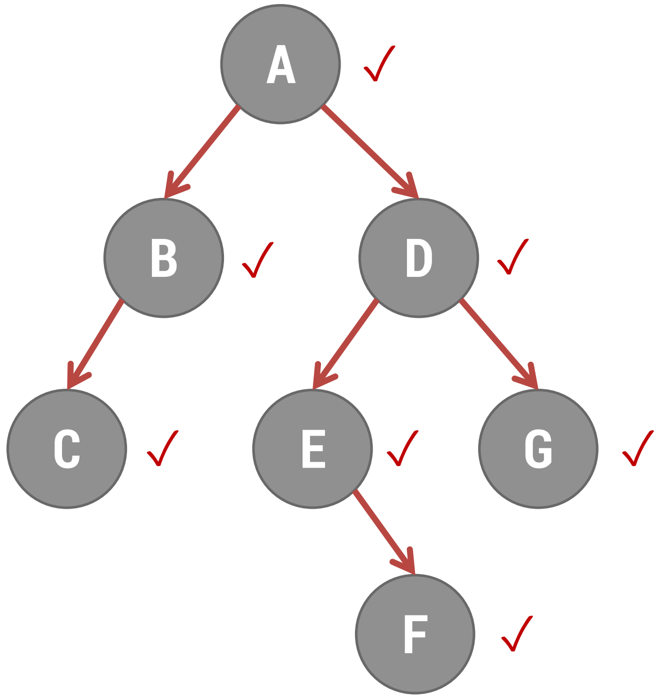



**Preorder traversal of a given tree as**

**A B C D E F G**

# Inorder Traversal

- Inorder traversal of a binary tree is defined as follow
  1. **Traverse** the **left subtree** in Inorder
  2. **Process** the **root node**
  3. **Traverse** the **right subtree** in Inorder



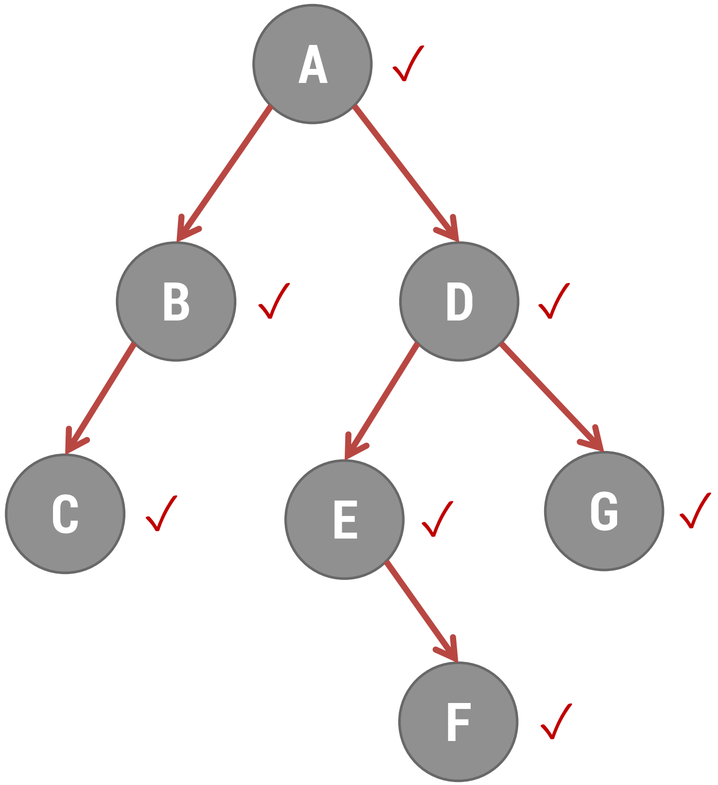

**Inorder traversal of a given tree as**

**C B A E F D G**

# Postorder Traversal

- Postorder traversal of a binary tree is defined as follow

1. **Traverse** the **left subtree** in Postorder
2. **Traverse** the **right subtree** in Postorder
3. **Process** the **root node**



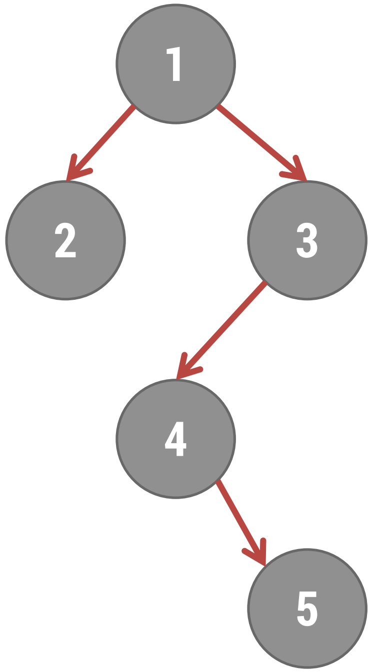

**Postorder traversal of a given tree as**

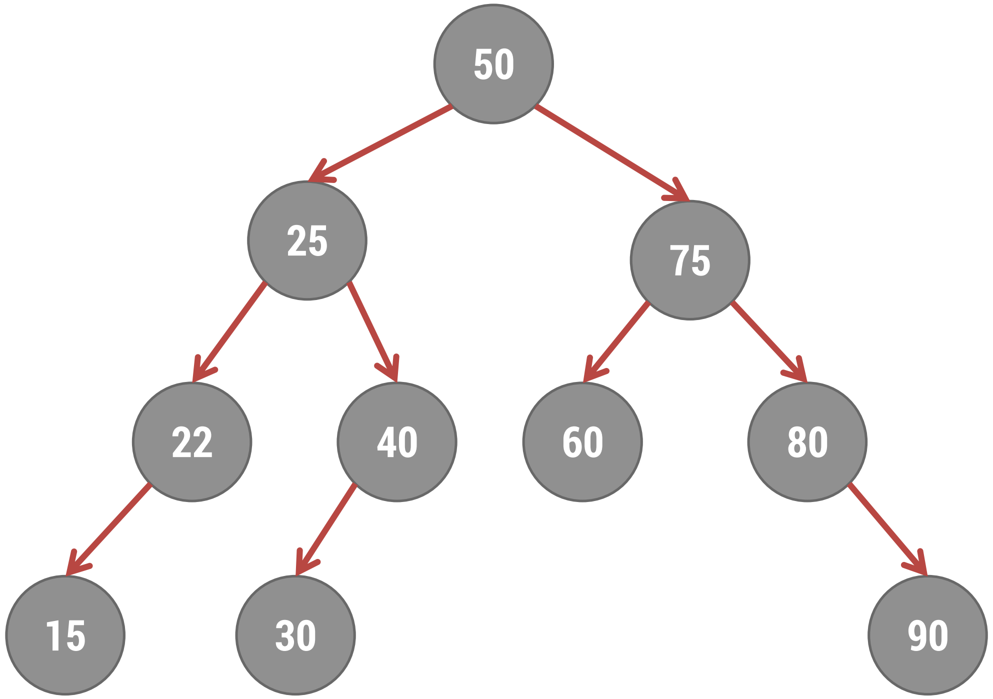
**C B F E G D A**

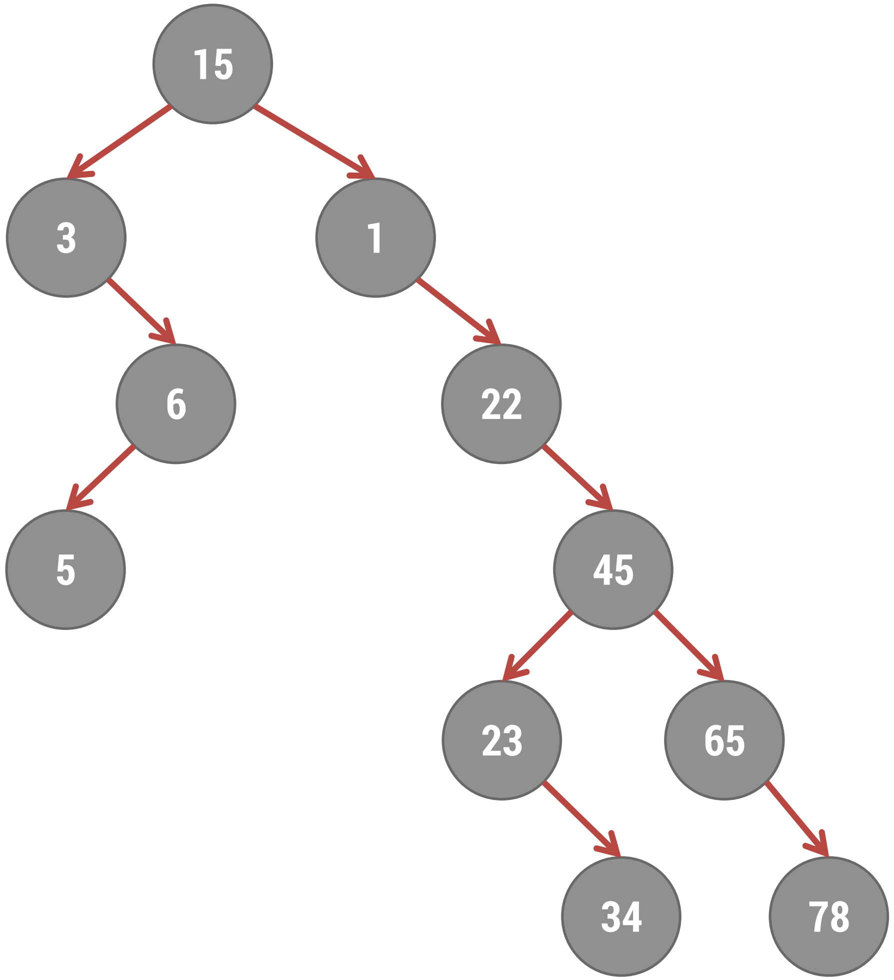
# Converse Traversal

- If we ***interchange left and right words*** ***in the preceding definitions***, we obtain three new traversal orders which are called
  - **Converse Preorder** Traversal: A D G E F B C
  - **Converse Inorder** Traversal: G D F E A B C
  - **Converse Postorder** Traversal: G F E D C B A

# Write Pre/In/Post Order Traversal



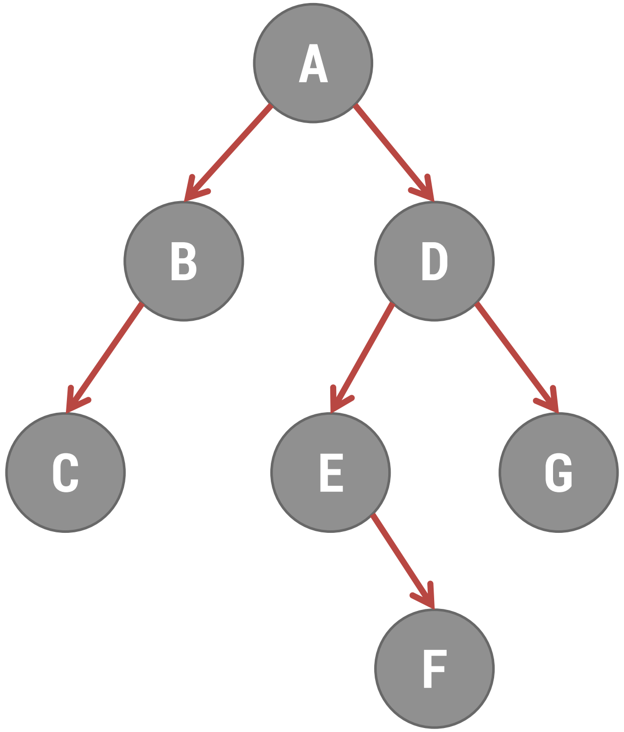



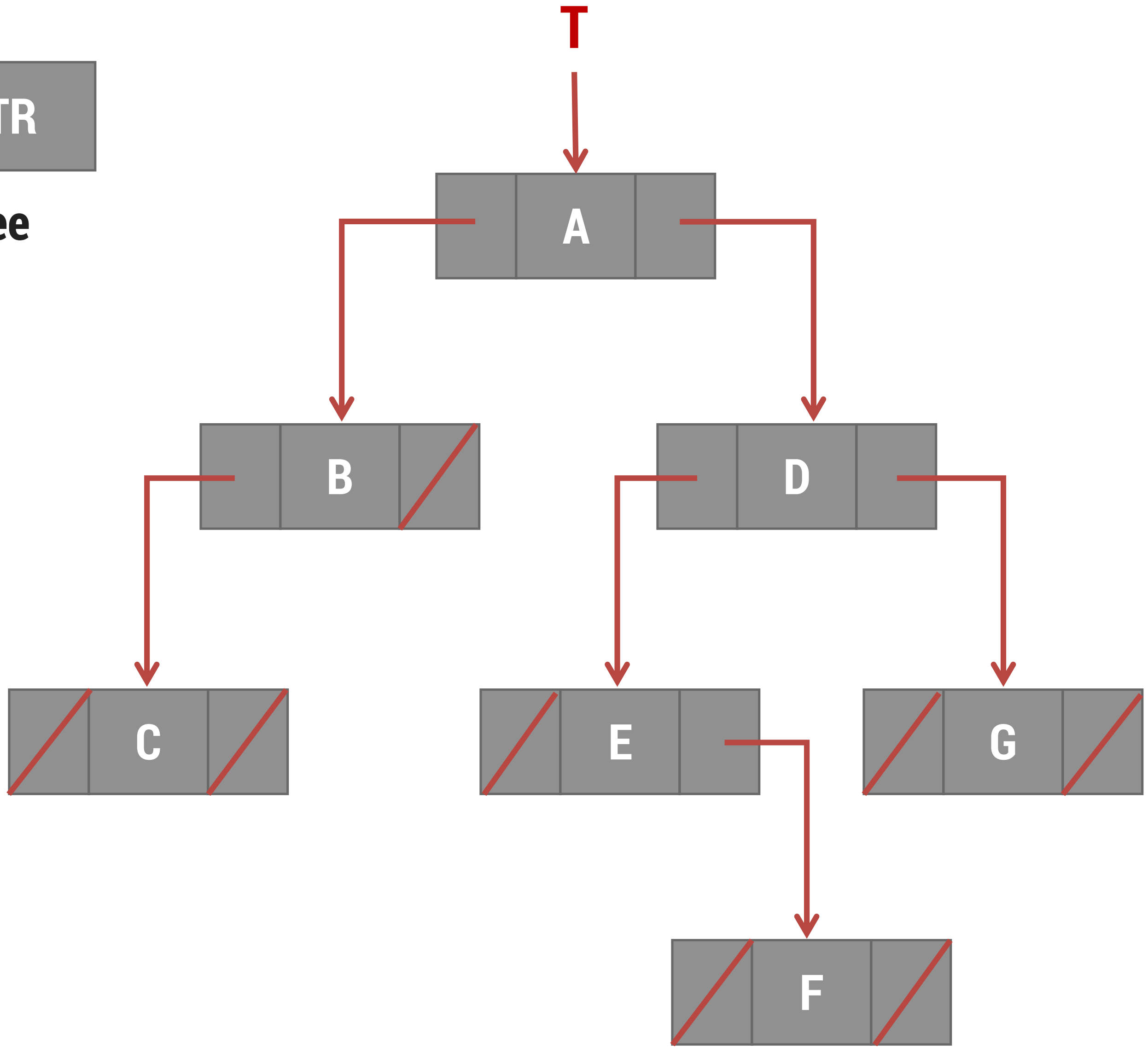



# Linked Representation of Binary Tree

| LPTR | DATA | RPTR |
|---|---|---|

Typical node of Binary Tree



# Algorithm of Binary Tree Traversal

- Preorder Traversal - Procedure: RPREORDER(T)
- Inorder Traversal - Procedure: RINORDER(T)
- Postorder Traversal - Procedure: RPOSTORDER(T)

# Procedure: RPREORDER(T)

- This procedure **traverses the tree** in **preorder**, in a recursive manner.
- **T is root node address** of given binary tree
- Node structure of binary tree is described as below

| LPTR | DATA | RPTR |
|---|---|---|

**Typical node of Binary Tree**

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write (‘Empty Tree’)
          return
   ELSE   write (DATA(T))
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RPREORDER (LPTR (T))
3. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN   RPREORDER (RPTR (T))
4. [Finished]
   Return
```

# Procedure: RINORDER(T)

- This procedure **traverses the tree** in **InOrder**, in a recursive manner.
- **T is root node address** of given binary tree.
- Node structure of binary tree is described as below.

| LPTR | DATA | RPTR |
|---|---|---|

**Typical node of Binary Tree**

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write ('Empty Tree')
          return
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RINORDER (LPTR (T))
3. [Process the Root Node]
   write (DATA(T))
4. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN   RINORDER (RPTR (T))
5. [Finished]
   Return
```

# Procedure: RPOSTORDER(T)

- This procedure **traverses the tree** in **PostOrder**, in a recursive manner.
- **T is root node address** of given binary tree.
- Node structure of binary tree is described as below.

| LPTR | DATA | RPTR |
|---|---|---|

**Typical node of Binary Tree**

```
1. [Check for Empty Tree]
   IF     T = NULL
   THEN   write (‘Empty Tree’)
          return
2. [Process the Left Sub Tree]
   IF     LPTR (T) ≠ NULL
   THEN   RPOSTORDER (LPTR (T))
3. [Process the Right Sub Tree]
   IF     RPTR (T) ≠ NULL
   THEN   RPOSTORDER (RPTR (T))
4. [Process the Root Node]
   write (DATA(T))
5. [Finished]
   Return
```

# Construct Binary Tree from Traversal

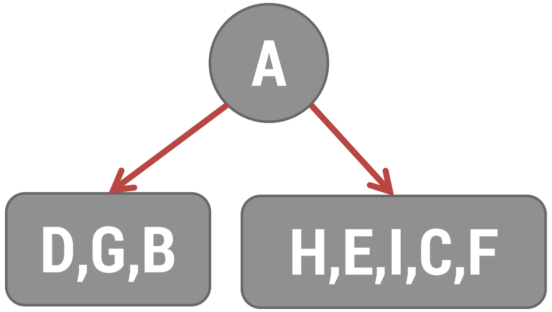
**Construct a Binary tree** from the given **Inorder** and **Postorder** traversals

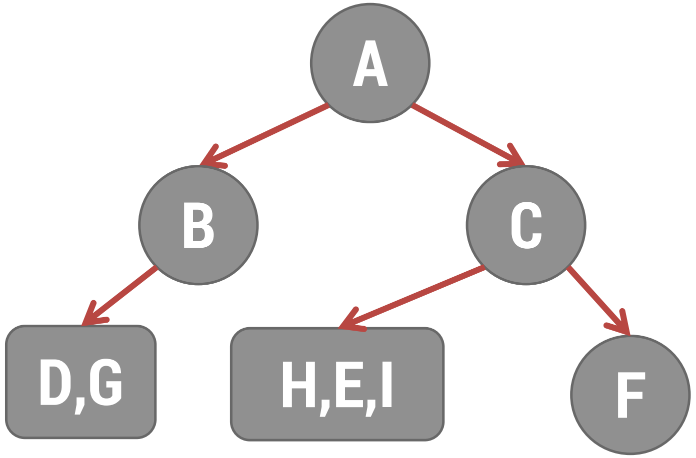
**Inorder** : D G B **A** H E I C F
**Postorder** : G D B H I E F C **A**

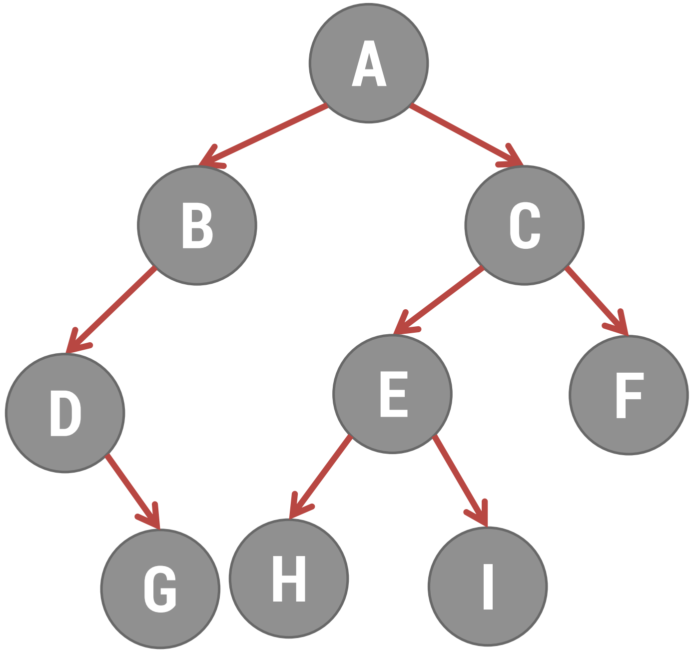
- Step 1: Find the root node
  - Preoder Traversal – first node is root node
  - Postoder Traversal last node is root node
- Step 2: Find Left & Right Sub Tree
  - Inorder traversal gives Left and right sub tree

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

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



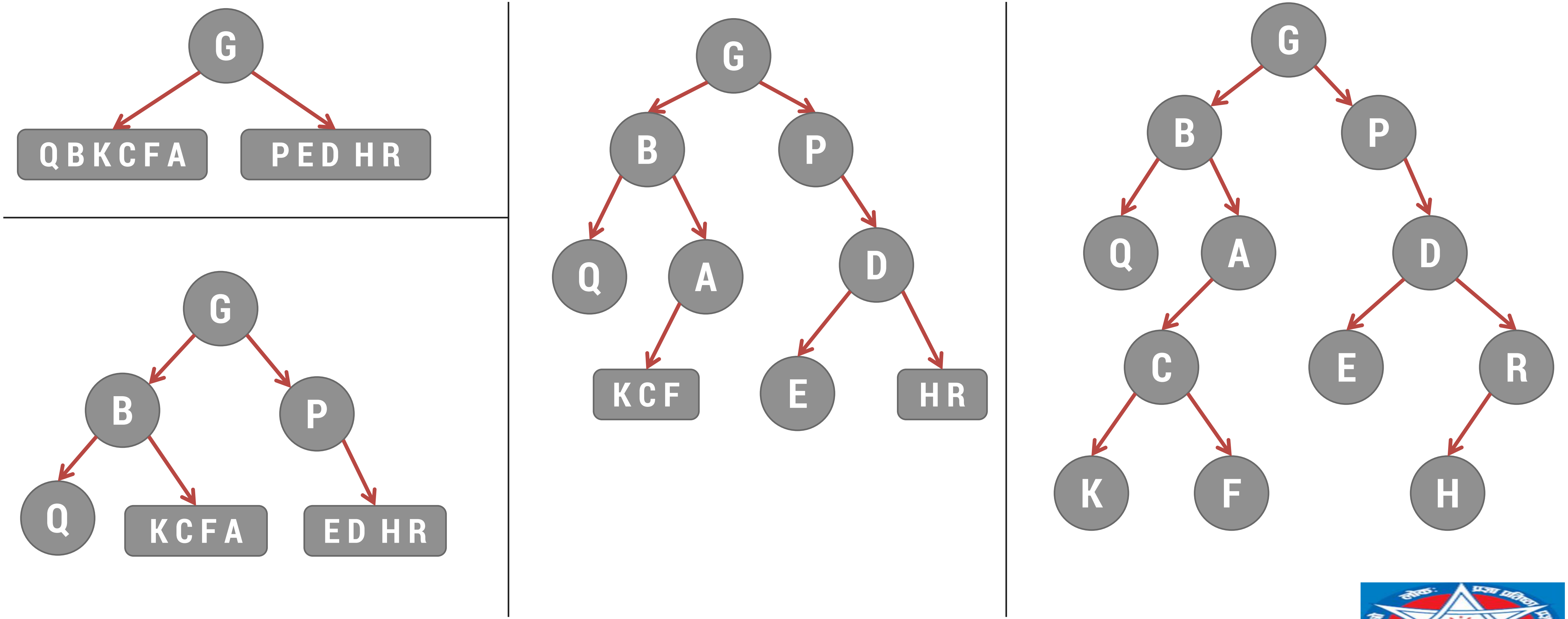



# Construct Binary Tree from Traversal

**Preorder :** G B Q A C K F P D E R H

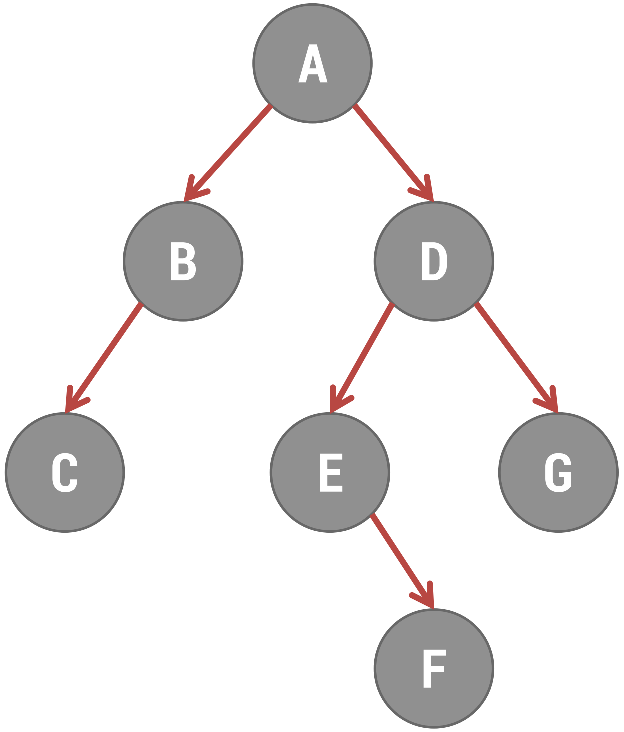
**Inorder :** Q B K C F A G P E D H R

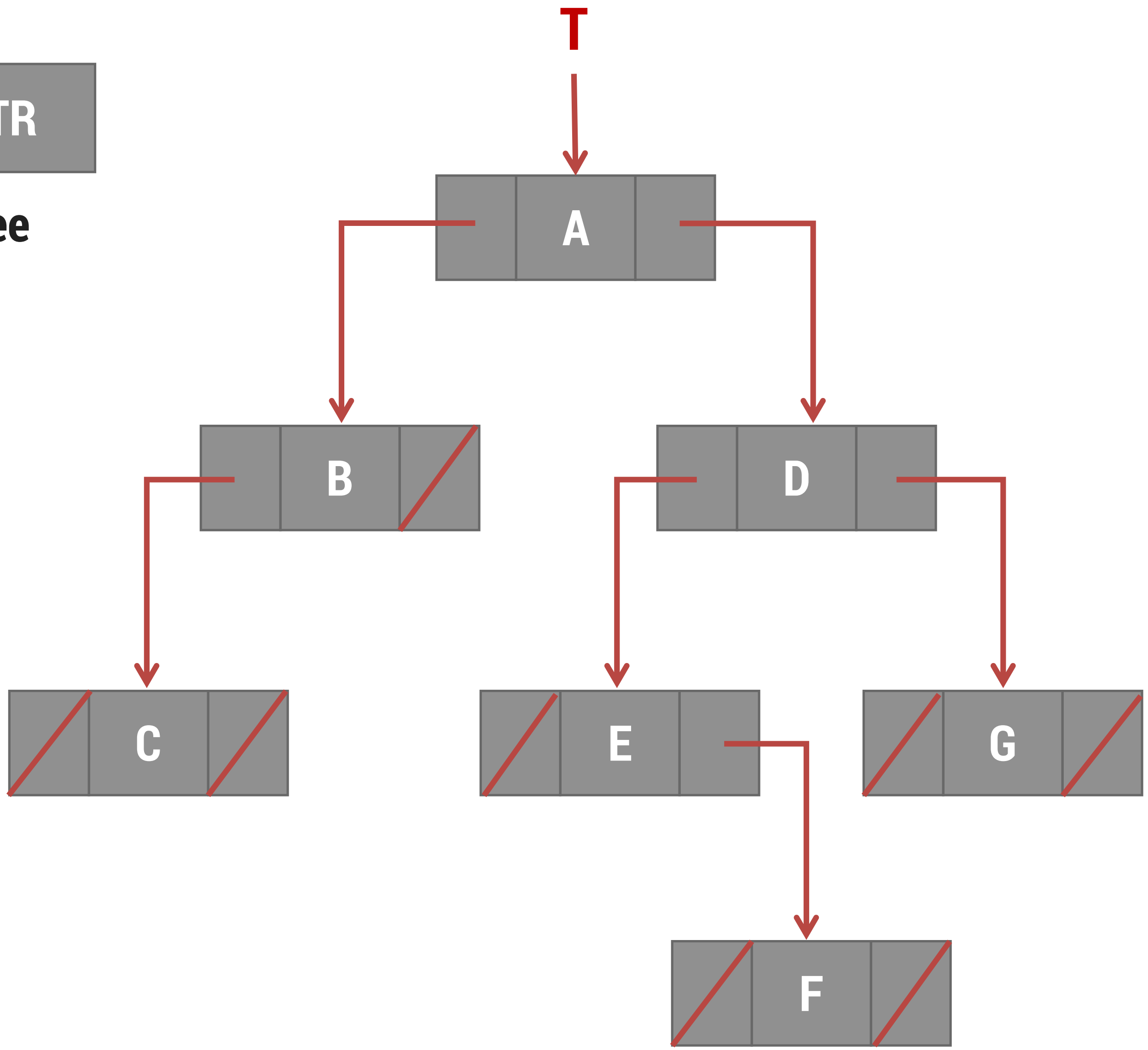


# Linked Representation of Binary Tree

| LPTR | DATA | RPTR |
|---|---|---|

**Typical node of Binary Tree**



# Threaded Binary Tree

- The **wasted NULL** links in the binary tree storage representation can be **replaced by threads**
- A binary **tree** is **threaded according** to particular **traversal order**. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order
- **In-Threaded Binary Tree**
  - **If left link of node P is null**, then this link is **replaced by** the **address of its predecessor**
  - **If right link of node P is null**, then this link is **replaced by** the **address of its successor**
- Because the left or right **link** of **a node** can denote **either structural link** or **a thread**, we must somehow be able to distinguish them

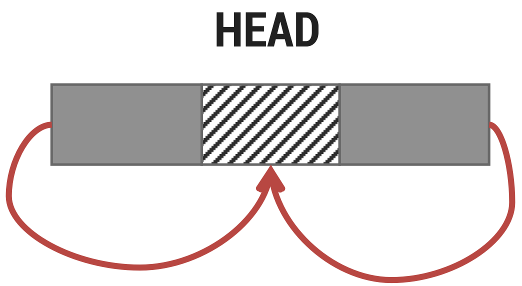
# Threaded Binary Tree

- **Method 1:-** Represent **thread a Negative address**
- **Method 2:-** To have a **separate Boolean flag** for each of left and right pointers, node structure for this is given below

| LPTR | LTHREAD | DATA | RTHREAD | RPTR |
|---|---|---|---|---|

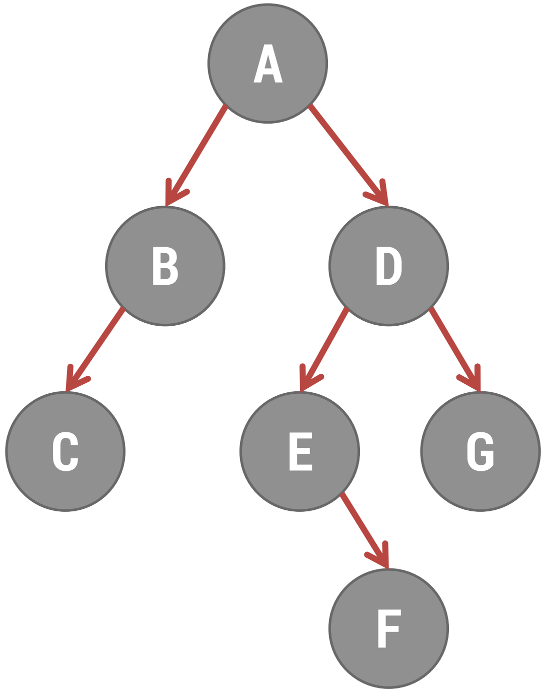
**Typical node of Threaded Binary Tree**

- **LTHREAD = true =** Denotes leaf thread link
- **LTHREAD = false =** Denotes leaf structural link
- **RTHREAD = true =** Denotes right threaded link
- **RTHREAD = false =** Denotes right structural link



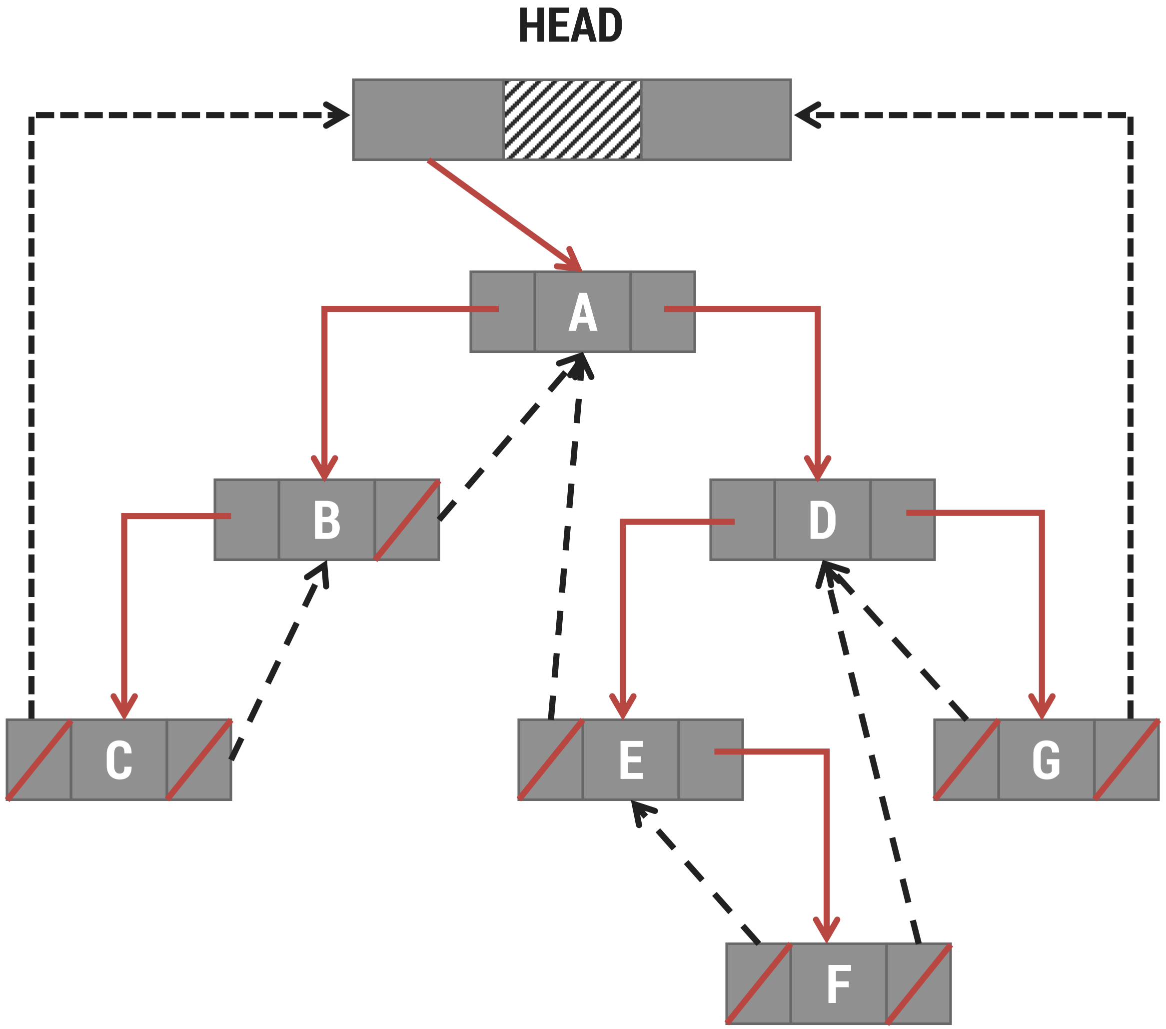

**Head node is simply another node which serves as the predecessor and successor of first and last tree nodes.**
**Tree is attached to the left branch of the head node.**

# Threaded Binary Tree



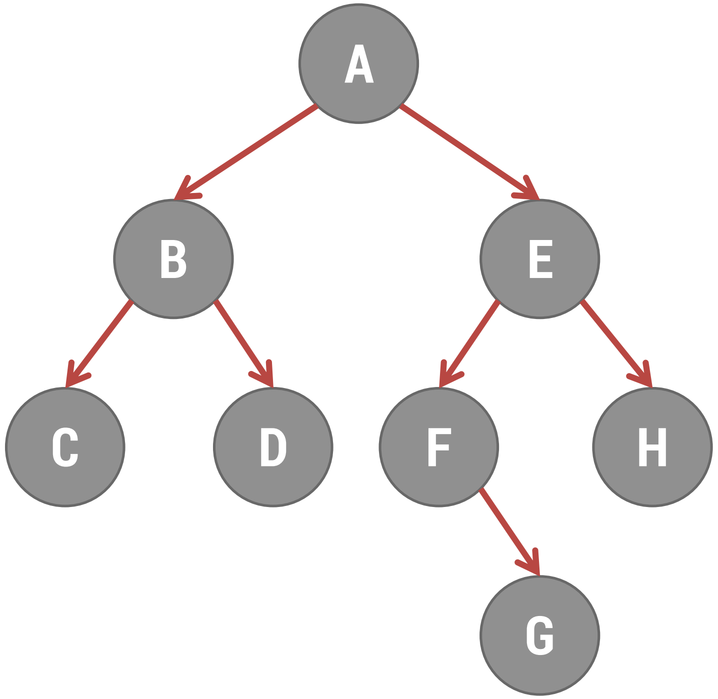

**Inorder Traversal**

**C B A E F D G**




**Fully In-Threaded Binary Tree**

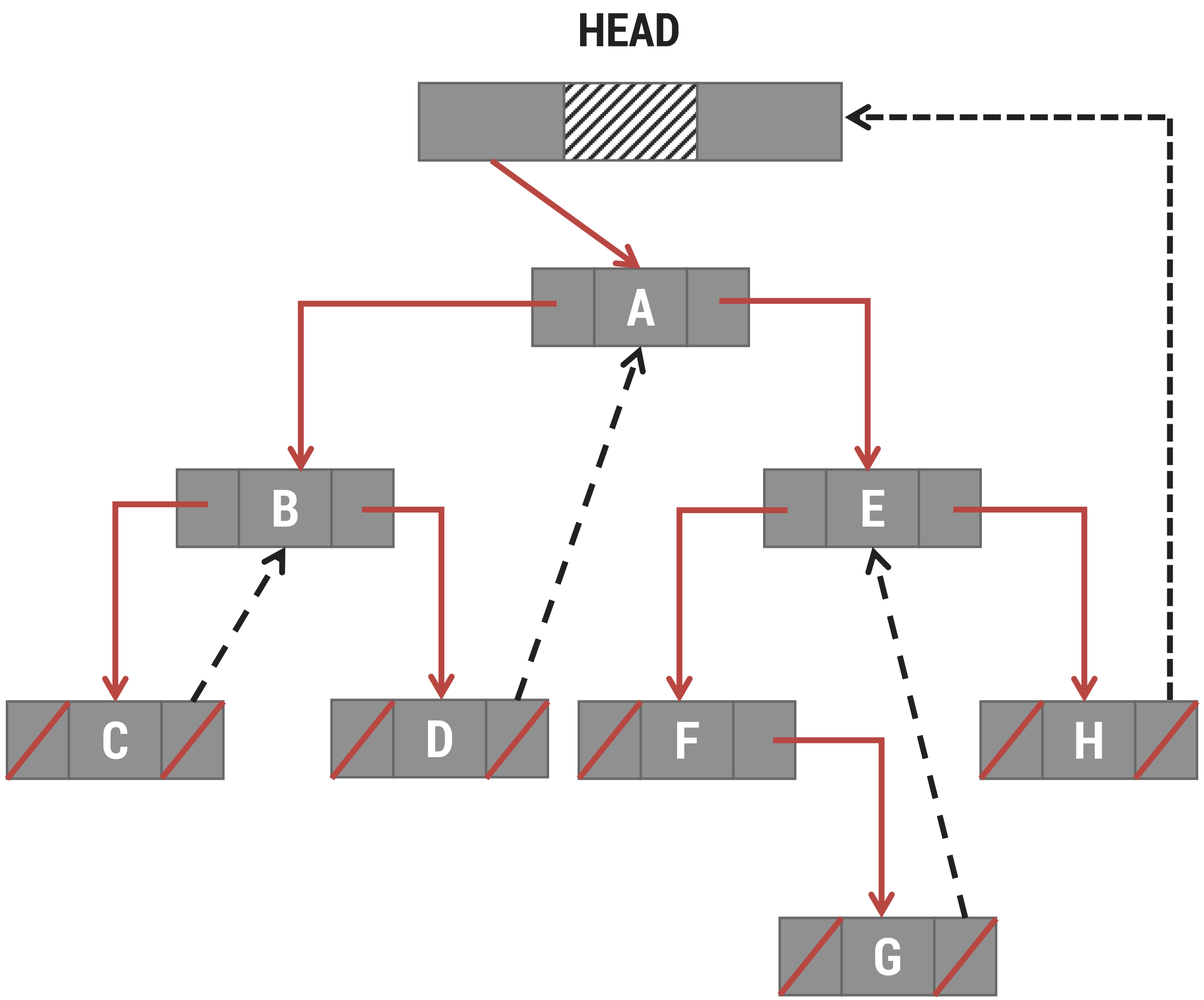
# Threaded Binary Tree




Inorder Traversal

C B D A F G E H

## Construct Right In-Threaded Binary Tree of given Tree



# Advantages of Threaded Binary Tree

- **Inorder traversal is faster** than unthreaded version as stack is not required.
- **Effectively determines** the **predecessor and successor** for inorder traversal, for unthreaded tree this task is more difficult.
- **A stack is required** to provide upward pointing information **in binary tree** which **threading provides without stack**.
- It is possible to **generate successor or predecessor** of any node **without** having over head of **stack** with the help of threading.

# Disadvantages of Threaded Binary Tree

- Threaded trees are **unable to share common sub trees.**
- If **Negative addressing is not permitted** in programming language, **two additional fields are required.**
- **Insertion** into and **deletion** from threaded binary tree are **more time consuming** because both thread and structural link must be maintained.

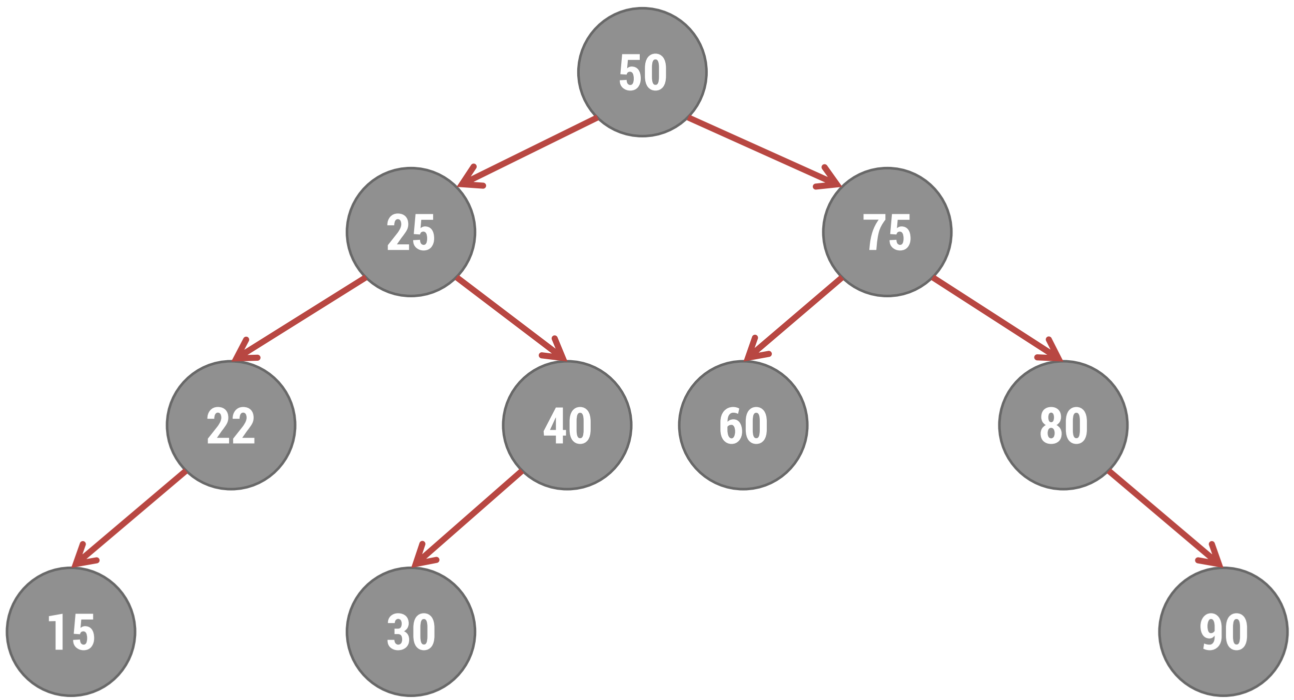
# Binary Search Tree (BST)

- A **binary search tree** is a **binary tree** in which **each node** possessed a key that **satisfy** the **following conditions**

  1. All **key** (if any) in **the left sub tree** of the root **precedes the key** in the **root**
  2. The **key in the root precedes** all **key** (if any) in the **right sub tree**
  3. The **left and right sub trees** of the root are again **search trees**

# Construct Binary Search Tree (BST)

Construct binary search tree for the following data
50 , 25 , 75 , 22 , 40 , 60 , 80 , 90 , 15 , 30

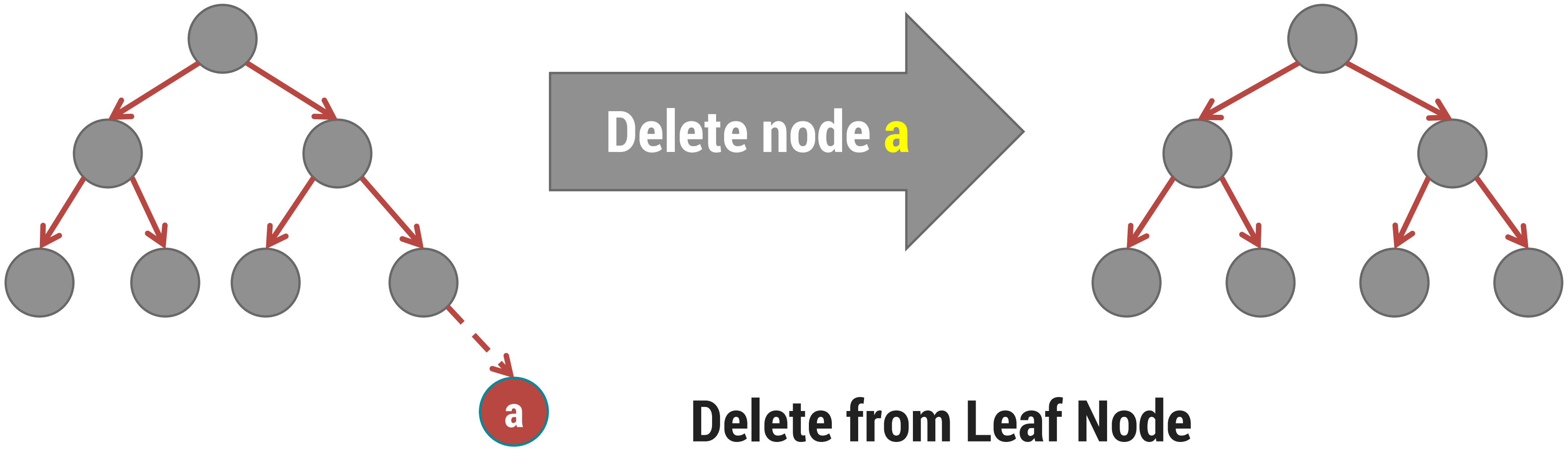



Construct binary search tree for the following data
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5

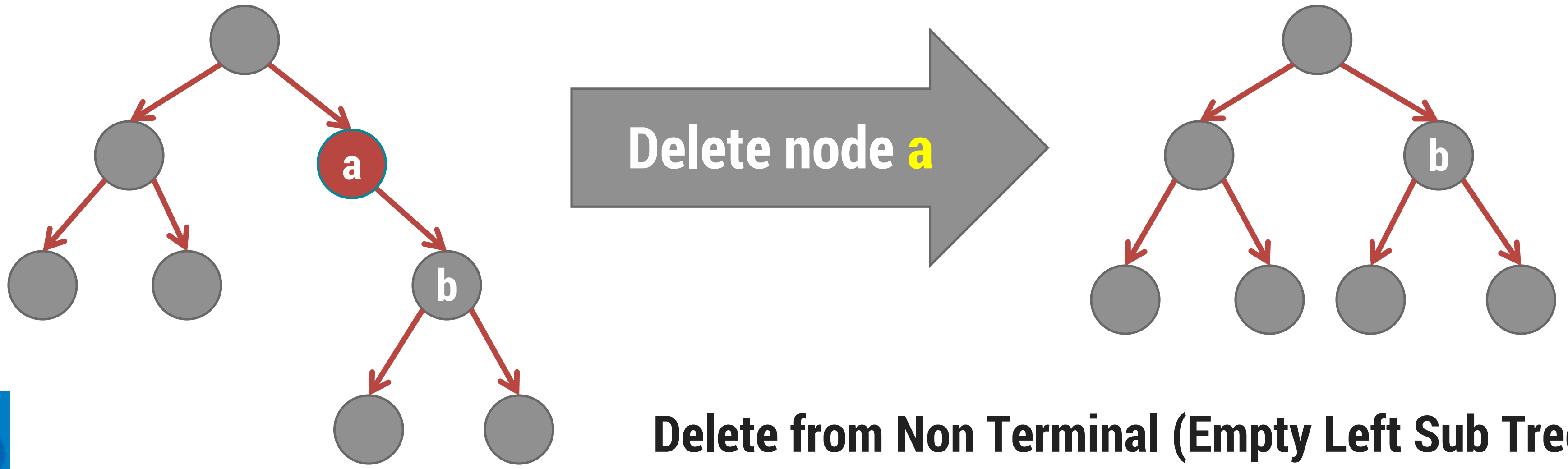
# Search a node in Binary Search Tree

- To search for target value.
- We first compare it with the key at root of the tree.
- If it is not same, we go to either Left sub tree or Right sub tree as appropriate and repeat the search in sub tree.
- If we have **In-Order List** & we want to search for specific node it requires **O(n) time.**
- In case of **Binary tree** it requires **$O(Log_2n)$** time to search a node.

# Delete node from Binary Search Tree

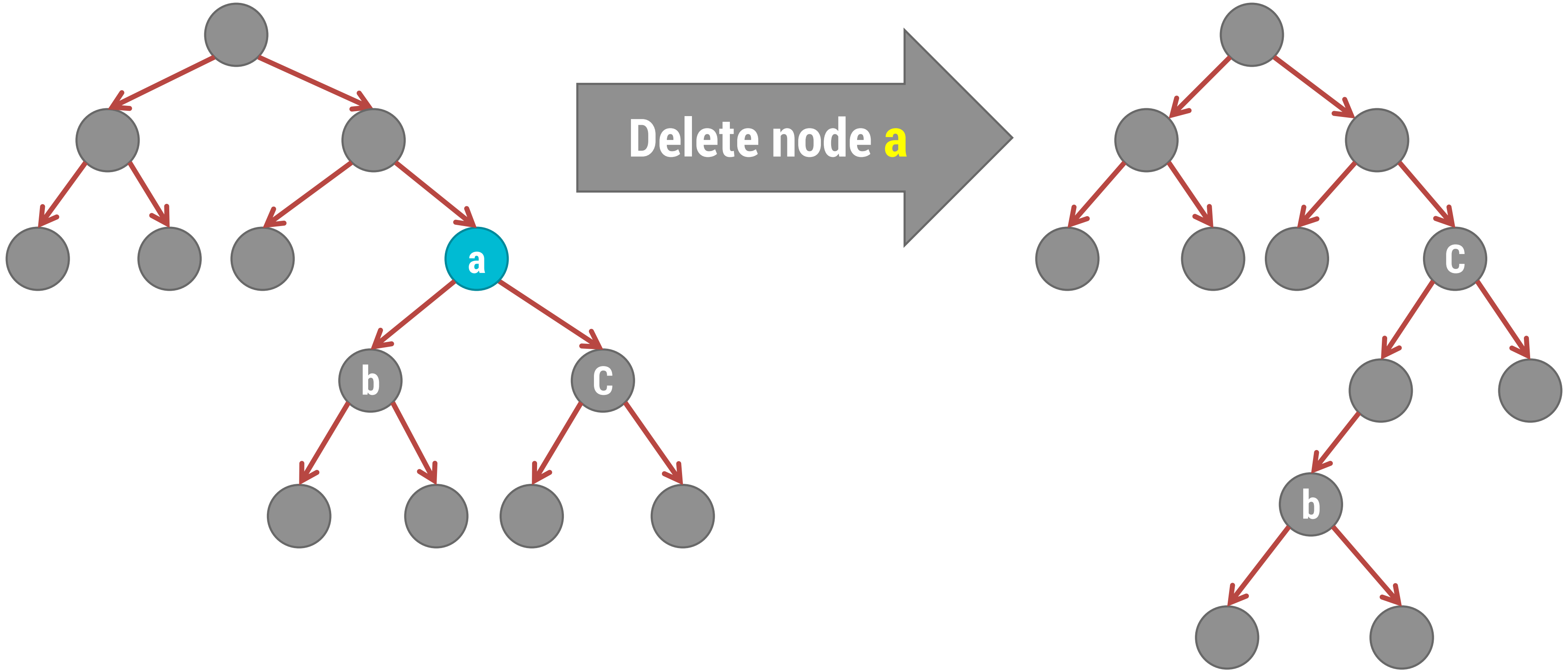



**Delete from Leaf Node**




**Delete from Non Terminal (Empty Left Sub Tree)**

# Delete node from BST

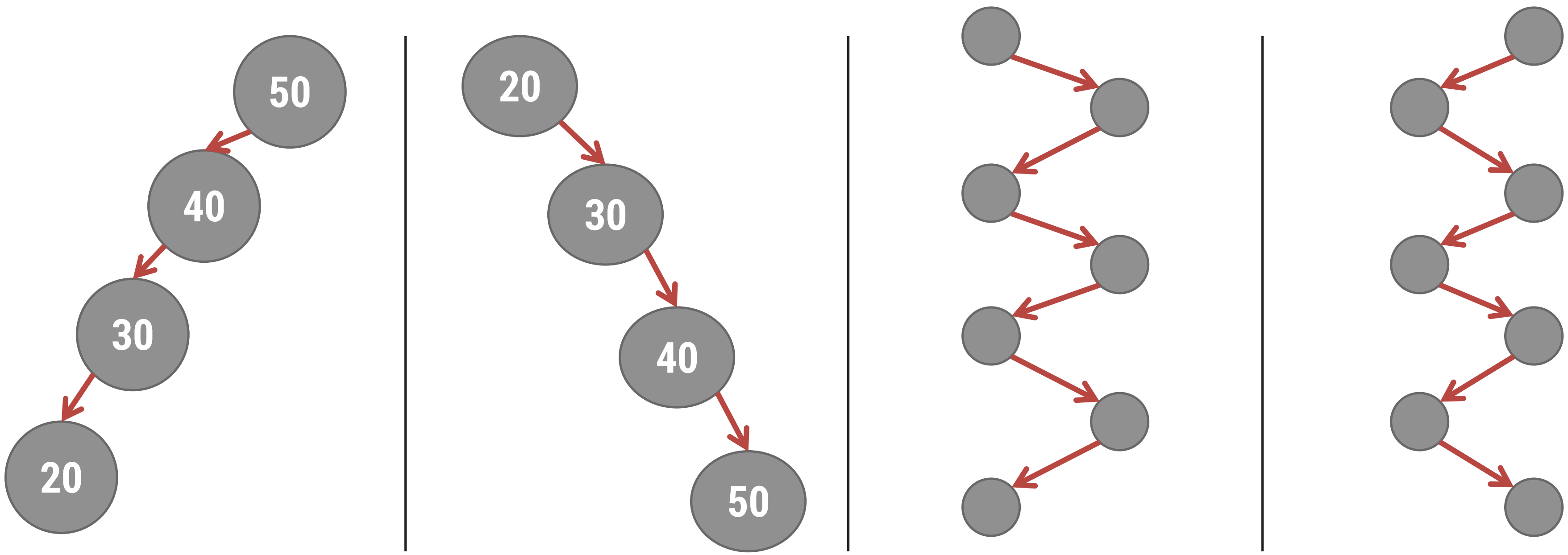



**Delete from Non Terminal (Neither Sub Tree is Empty)**

# Balanced Tree

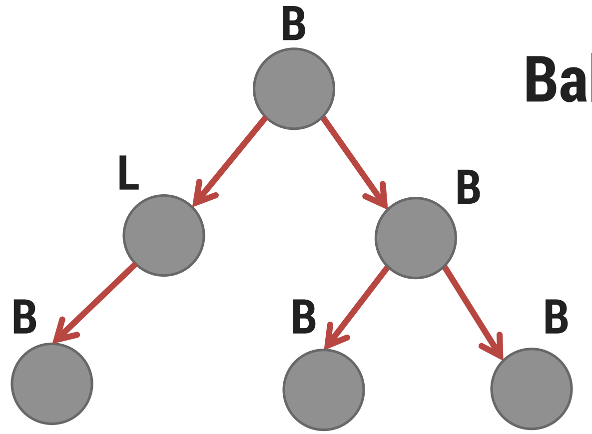
- Binary Search Tree gives advantage of Fast Search, but sometimes in few cases we are not able to get this advantage. E.g. look into worst case BST
- Balanced binary trees are classified into two categories
  - Height Balanced Tree (AVL Tree)
  - Weight Balanced Tree

**Worst search time cases for Binary Search Tree**

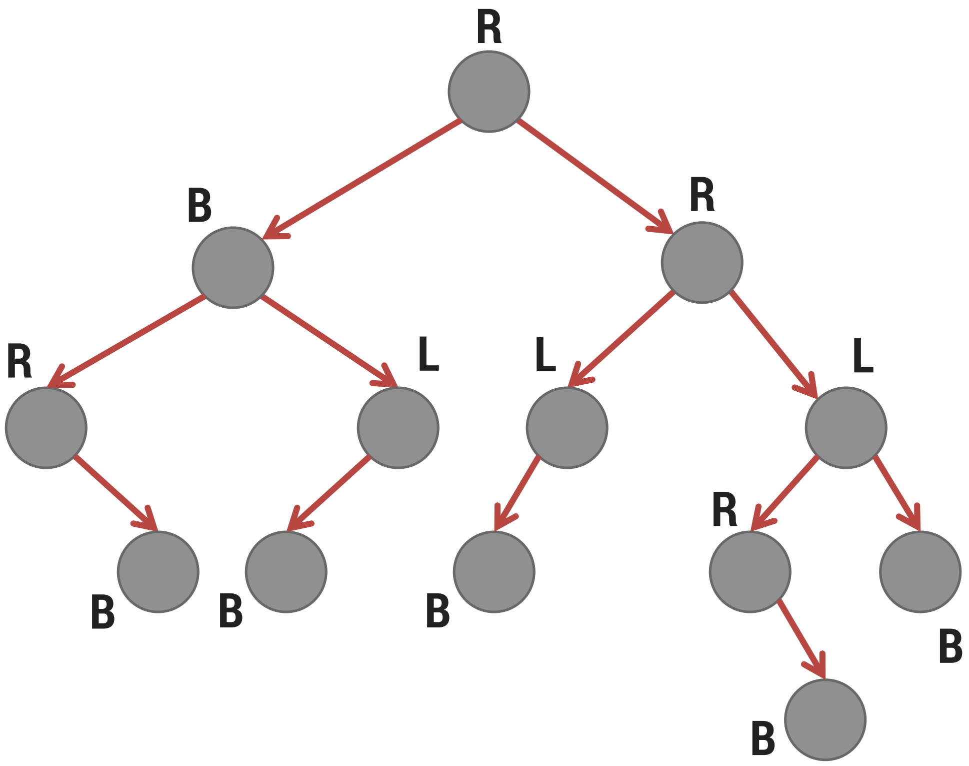


# Height Balanced Tree (AVL Tree)

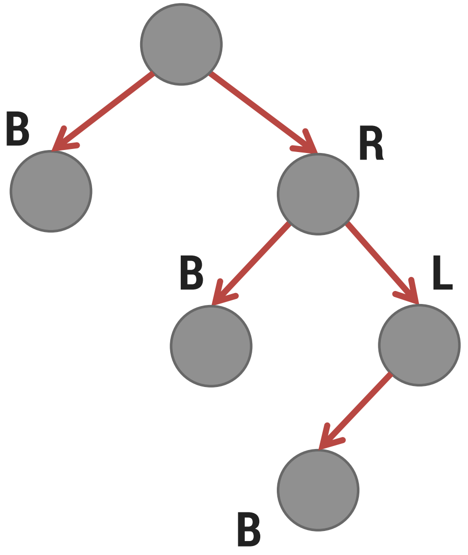
- A tree is called **AVL tree (Height Balanced Tree)**, if each node possessed one of the following properties
  - A **node** is called **left heavy**, if the **longest path in its left sub tree** is **one** longer than the **longest path of its right sub tree**
  - A **node** is called **right heavy**, if the **longest path in its right subtree** is **one** longer than **the longest path of its left sub tree**
  - A **node** is called **balanced**, if the longest path in **both the right and left sub-trees** are equal
- In height balanced tree, each node must be in one of these states
- If there exists a node in a tree where this is not true, then such a tree is called **Unbalanced**



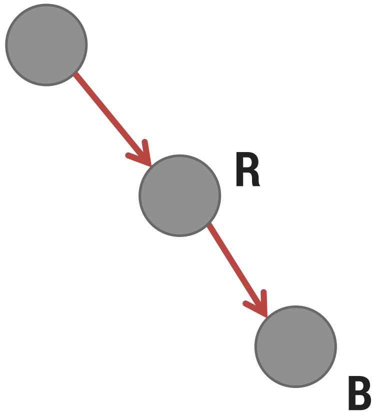

**Balanced Trees**




**Critical Node**
**Unbalanced Node**




**Critical Node**
**Unbalanced Node**




- Sometimes tree becomes unbalanced by inserting or deleting any node
- Then based on position of insertion, we need to rotate the unbalanced node
- **Rotation** is the **process** to **make tree balanced**

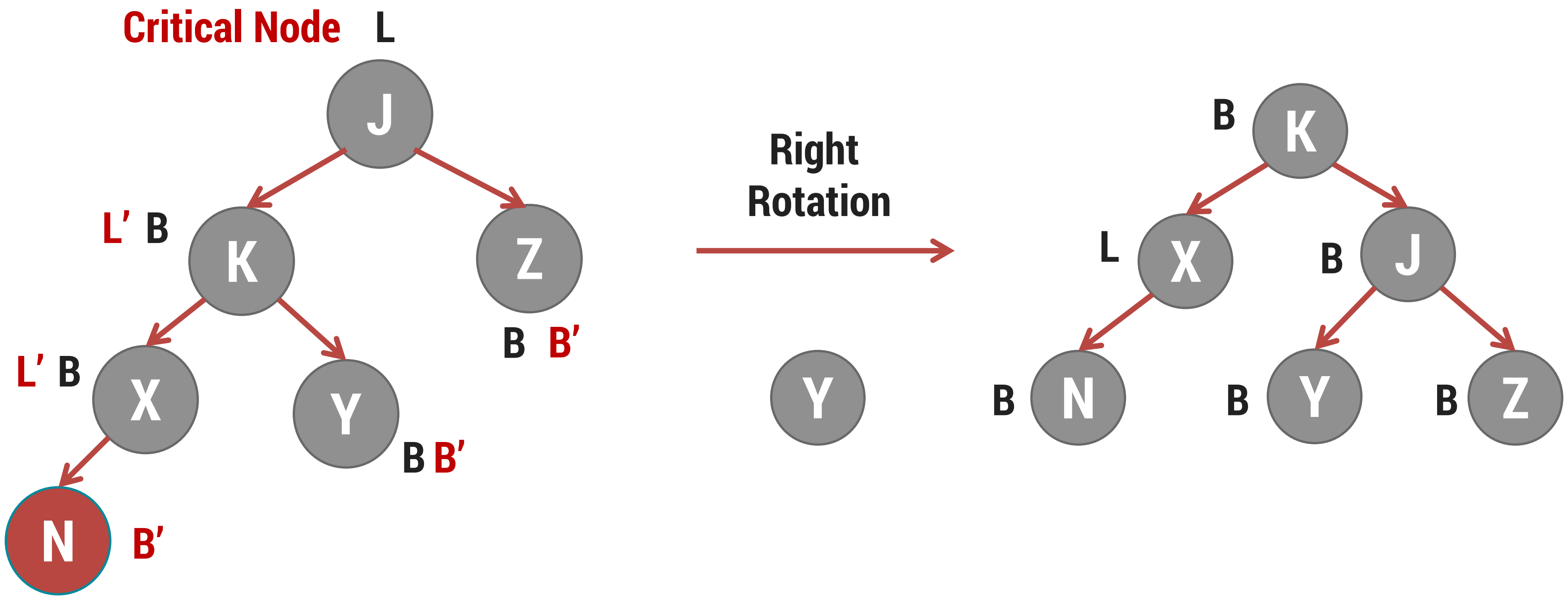
# Right Rotation

a. **Detach** left child's right sub-tree

b. Consider **left child** to be the **new parent**

c. **Attach old parent** onto **right of new parent**

d. **Attach old left child's old right sub-tree** as **left sub-tree of new right child**



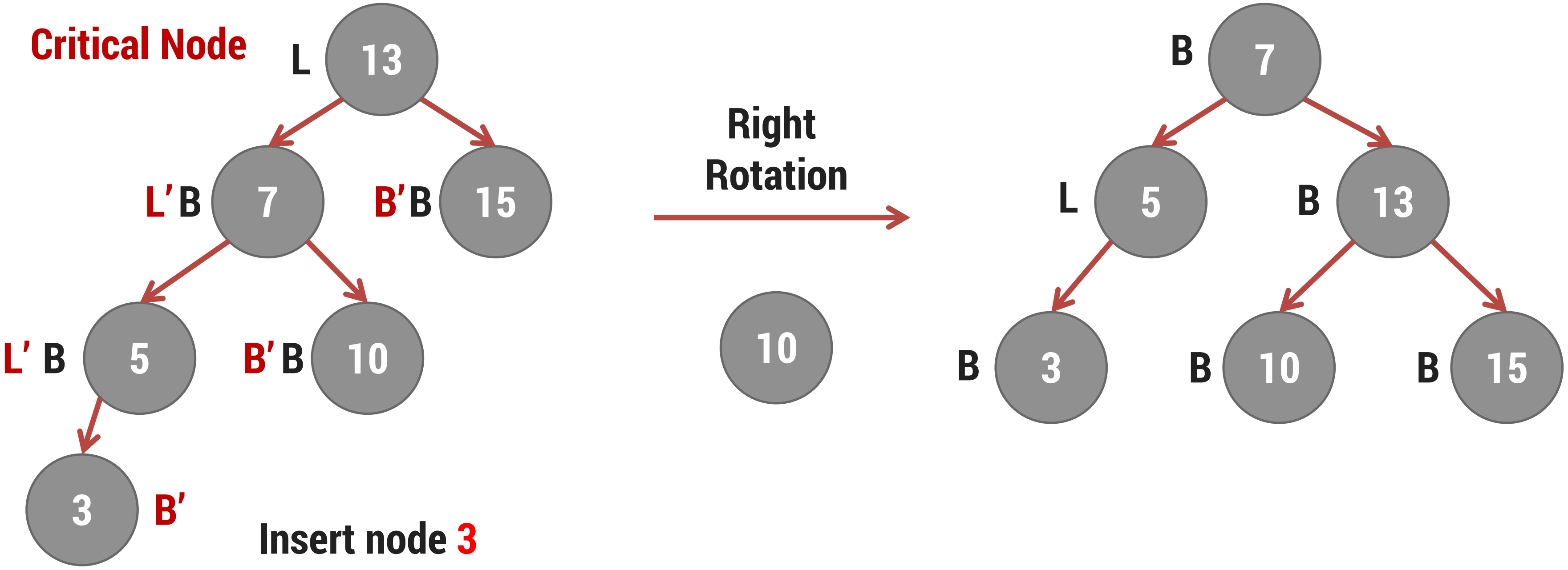
# Right Rotation

a. **Detach** left child's right sub-tree

b. Consider **left child** to be the **new parent**

c. **Attach old parent** onto **right of new parent**

d. **Attach old left child's old right sub-tree** as **left sub-tree of new right child**



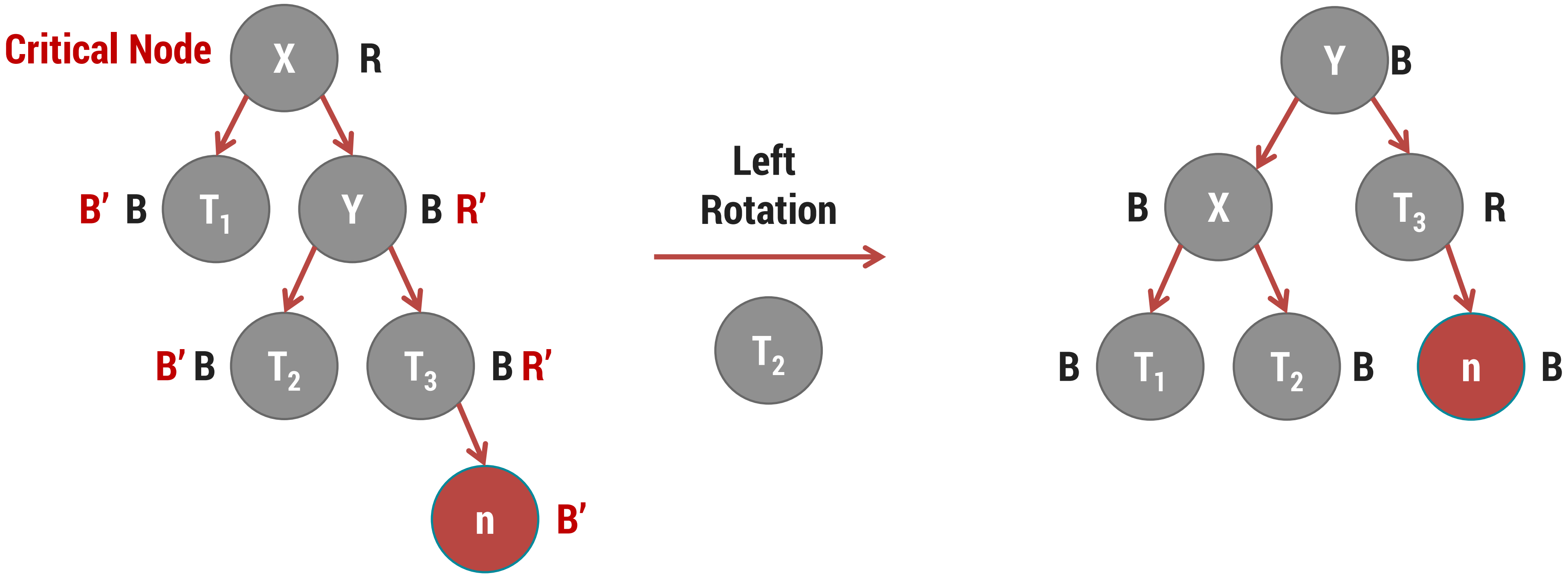
# Left Rotation

a. **Detach** right child's leaf sub-tree

b. Consider **right child** to be **new parent**

c. **Attach old parent** onto **left of new parent**

d. **Attach old right child's old left sub-tree** as **right sub-tree of new left child**



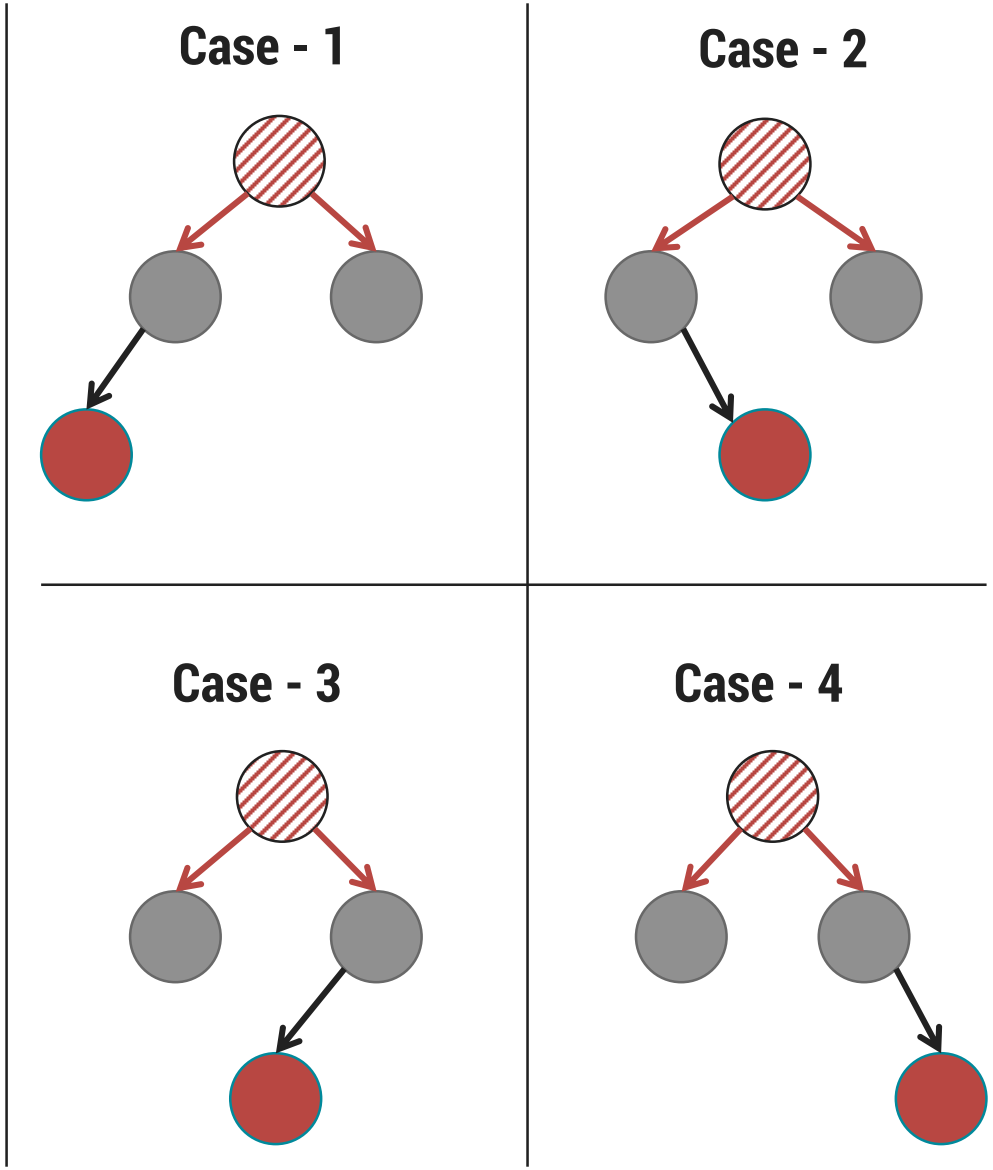
# Select Rotation based on Insertion Position

**Case 1:** Insertion into **Left sub-tree** of **nodes Left child**
- ➥ **Single Right Rotation**

**Case 2:** Insertion into **Right sub-tree** of **node's Left child**
- ➥ **Left Right Rotation**

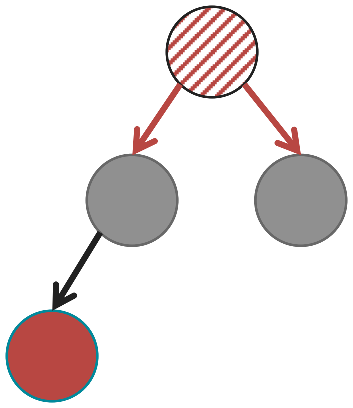
**Case 3:** Insertion into **Left sub-tree** of **node's Right child**
- ➥ **Right Left Rotation**

**Case 4:** Insertion into **Right sub-tree** of **node's Right child**
- ➥ **Single Left Rotation**



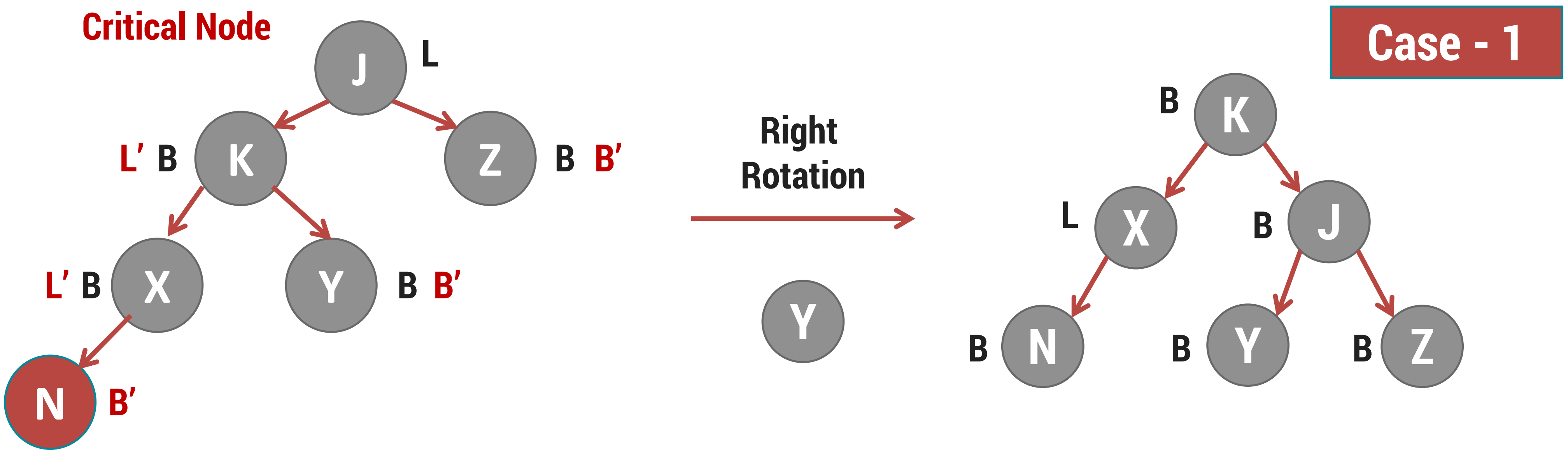
- **Case 1:** If node becomes **unbalanced** after **insertion** of new node at **Left sub-tree** of nodes **Left child**, then we need to perform **Single Right Rotation** of **unbalanced node** to balance the node
- **Right Rotation**
  a. Detach leaf child's right sub-tree
  b. Consider leaf child to be the new parent
  c. Attach old parent onto right of new parent
  d. Attach old leaf child's old right sub-tree as leaf sub-tree of new right child

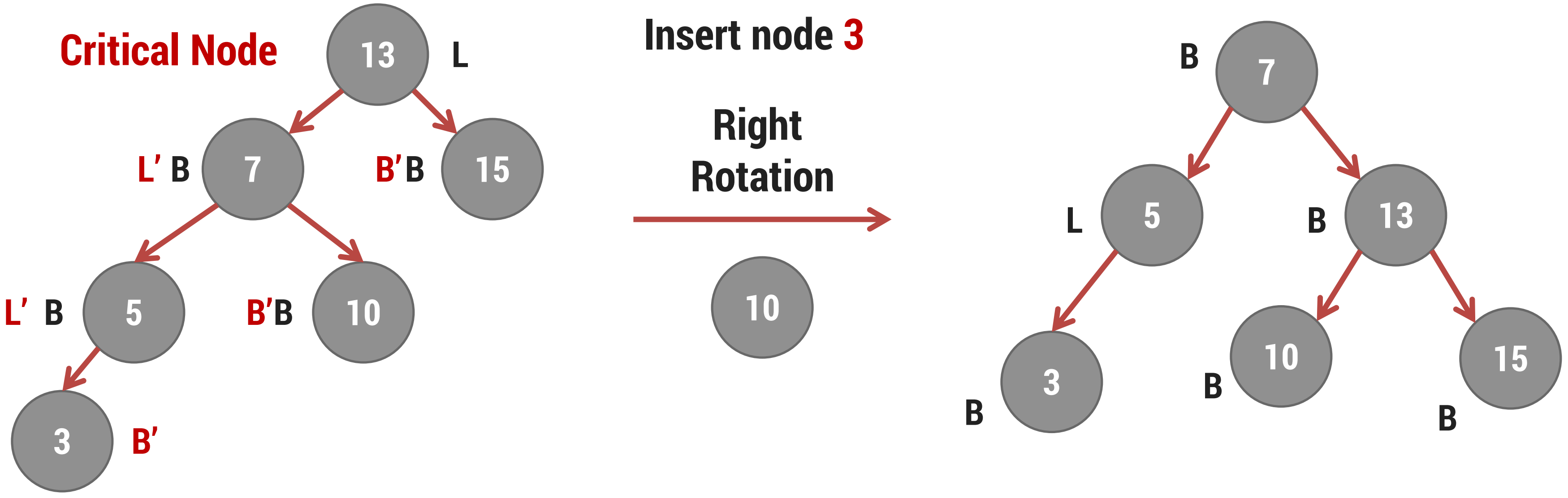
**Case - 1**



**Single Right Rotation**
of
**unbalanced node**

# Insertion into Left sub-tree of nodes Left child

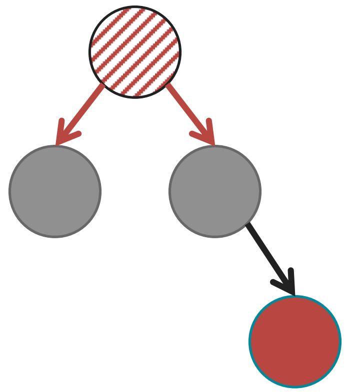


# Insertion into Right sub-tree of node's Right child

- **Case 4:** If node becomes unbalanced after **insertion of new node** at **Right sub-tree** of **nodes Right child**, then we need to perform **Single Left Rotation** of **unbalance node** to balance the node
- **Left Rotation**
  A. Detach right child's leaf sub-tree
  B. Consider right child to be new parent
  C. Attach old parent onto left of new parent
  D. Attach old right child's old left sub-tree as right sub-tree of new left child

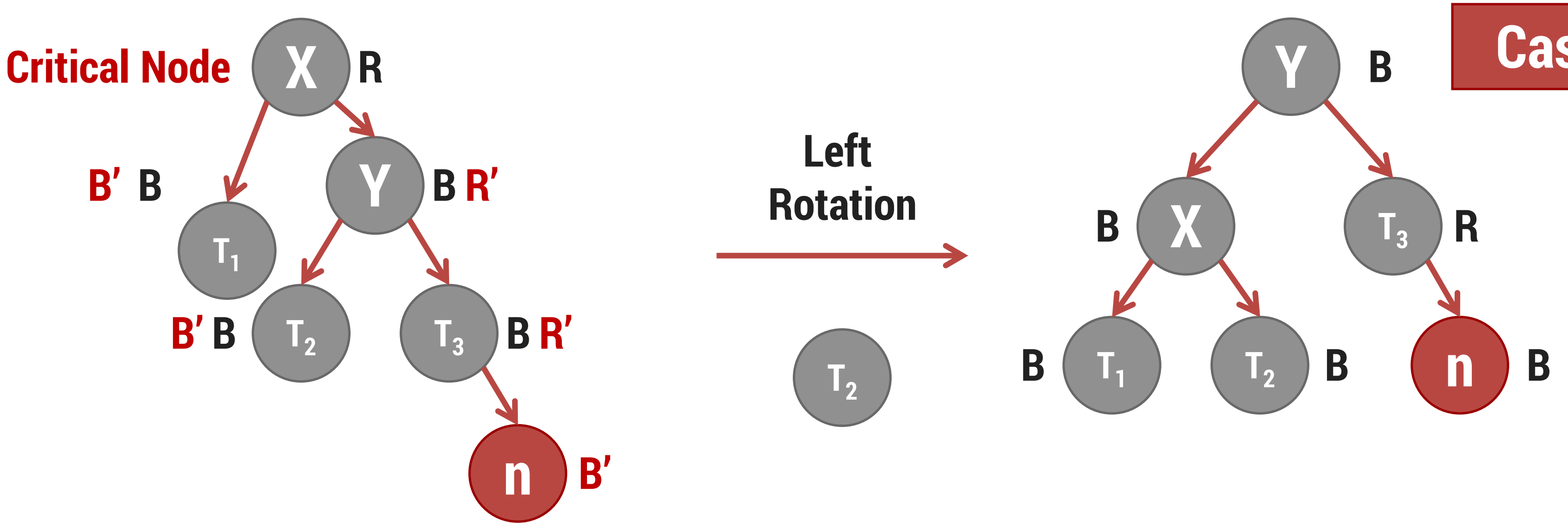
**Case - 4**

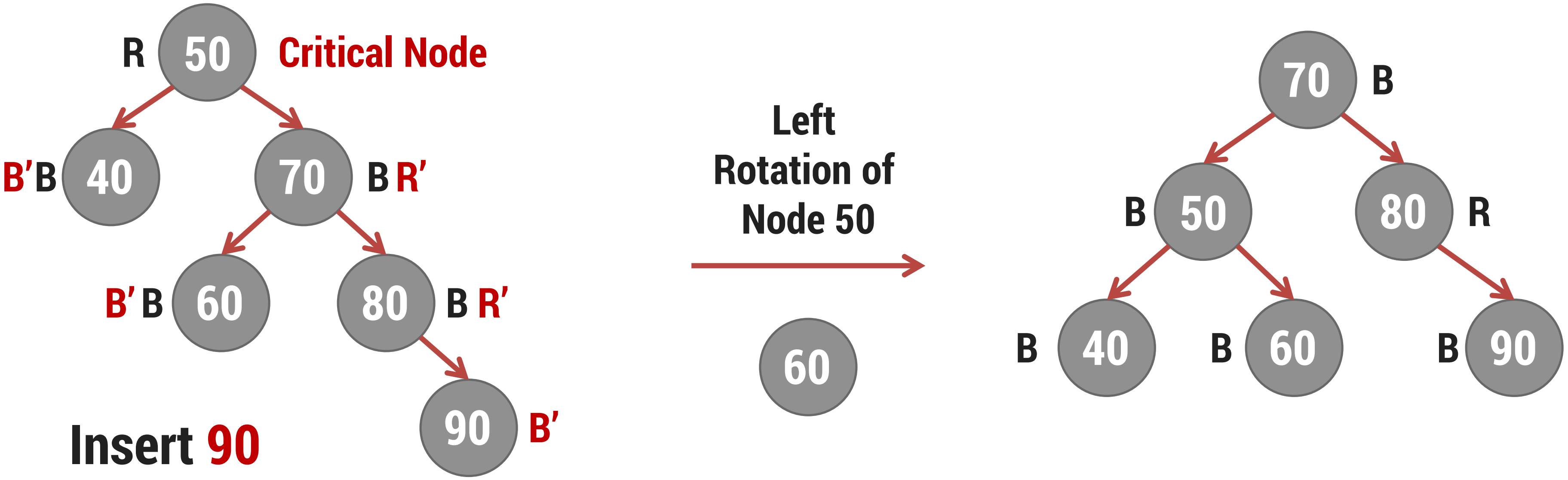


**Single Left Rotation**
of
**unbalanced node**

# Insertion into Right sub-tree of node's Right child

**Case - 4**



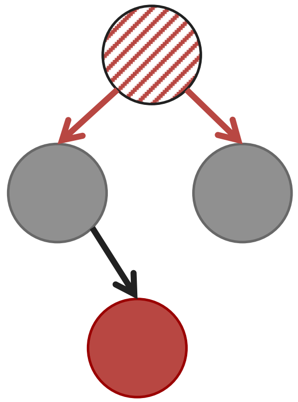




Insert 90

- **Case 2:** If node becomes unbalanced **after insertion** of **new node** at **Right sub-tree** of **node's Left child**, then we need to perform **Left Right Rotation** for **unbalanced node**.

- **Left Right Rotation**
  - **Left Rotation** of **Left Child** followed by
  - **Right Rotation** of **Parent**

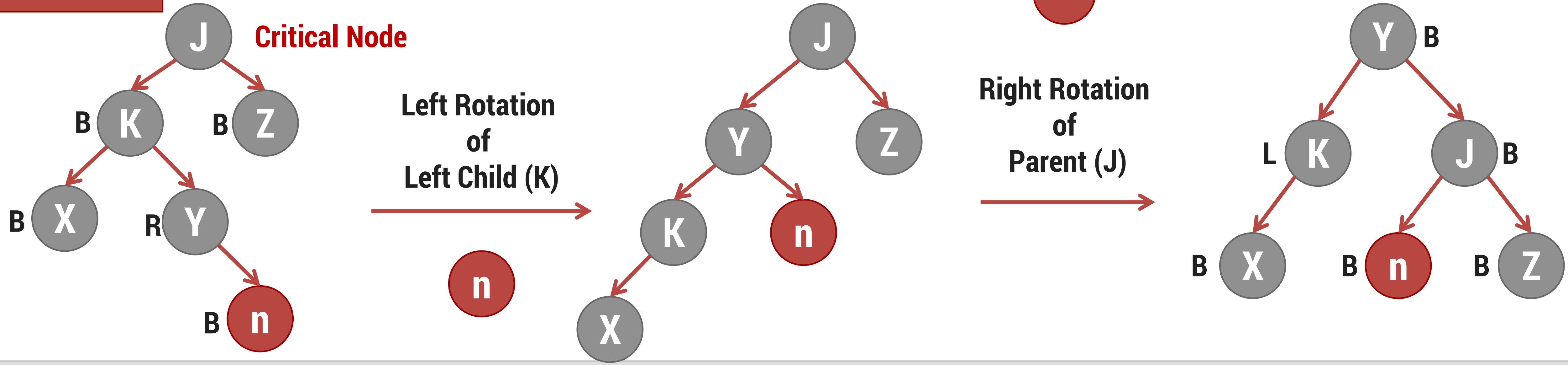
Case - 2



## Left Right Rotation

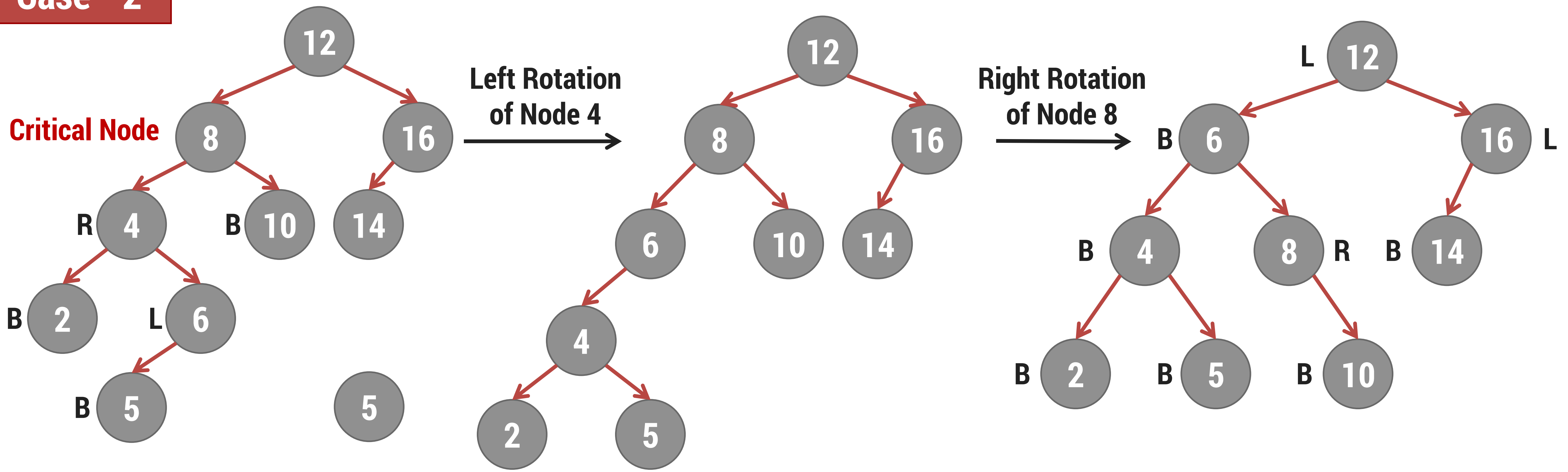
Left Rotation of Left Child
followed by
Right Rotation of Parent

Case - 2



# Insertion into Right sub-tree of node's Left child

## Case - 2

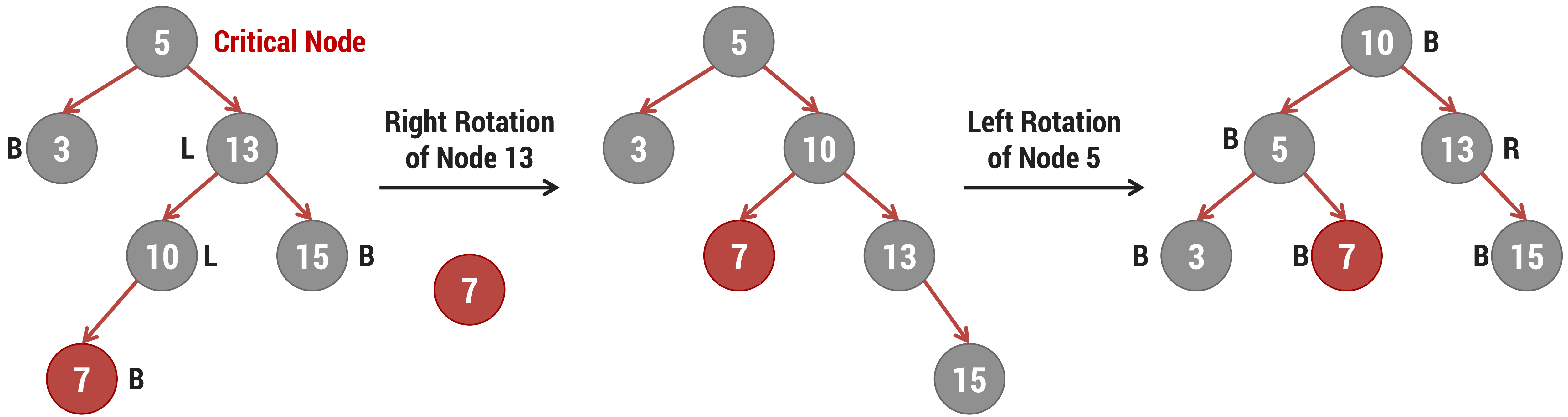



**Left Right Rotation**

Left Rotation of Left Child (4)
followed by
Right Rotation of Parent (8)

# Insertion into Left sub-tree of node's Right child

**Case - 3**

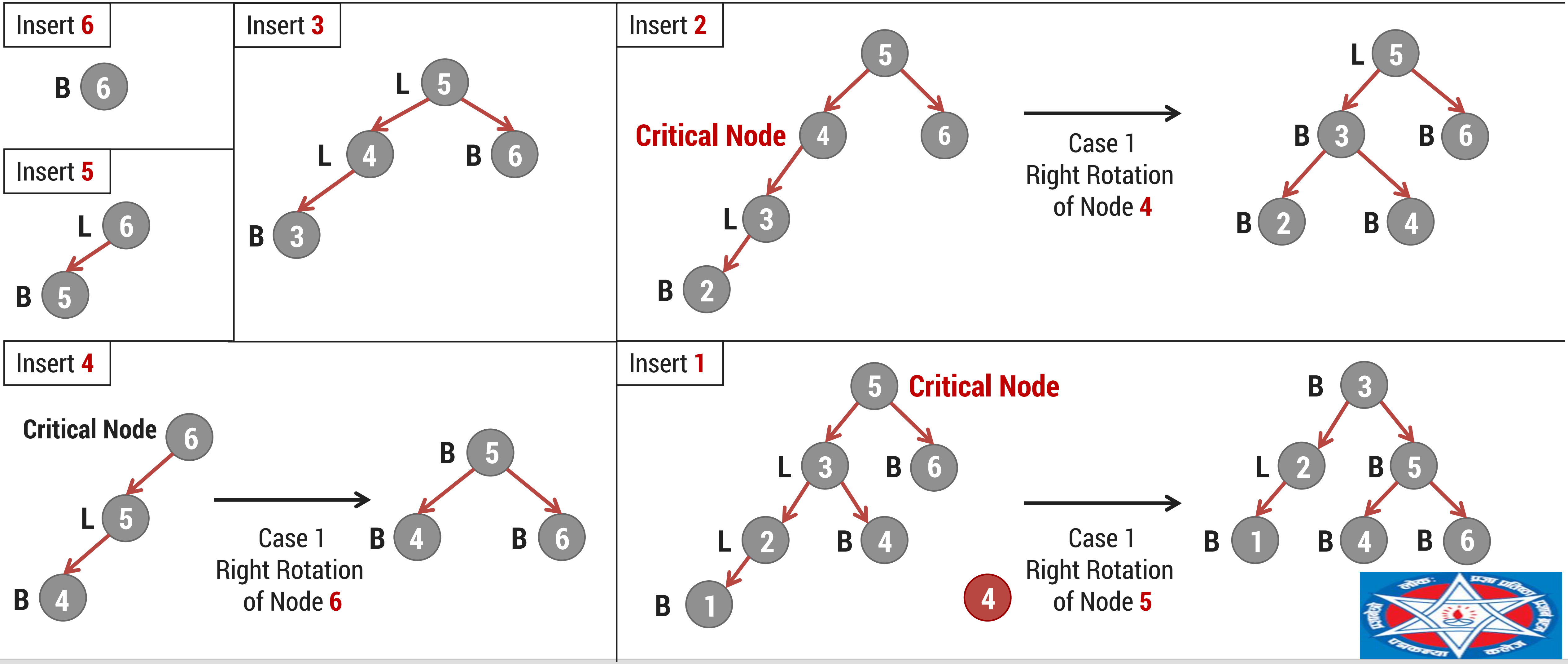



## Right Left Rotation

**Right Rotation of Right Child (13)**
**followed by**
**Left Rotation of Parent (5)**

# Construct AVL Search Tree

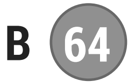
Construct AVL Search tree by inserting following elements in order of their occurrence **6, 5, 4, 3, 2, 1**



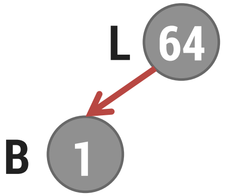
# Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

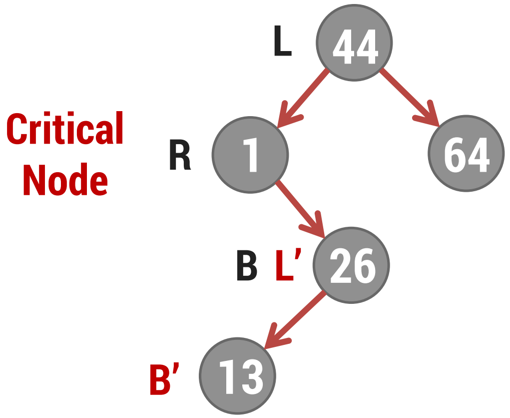
Insert **64**




Insert **1**

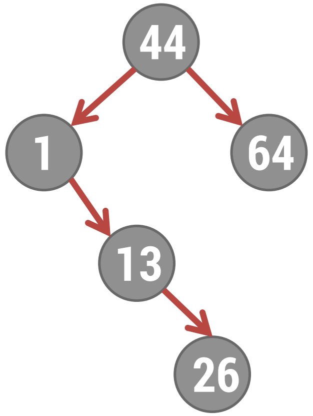



Insert **26 , 13**

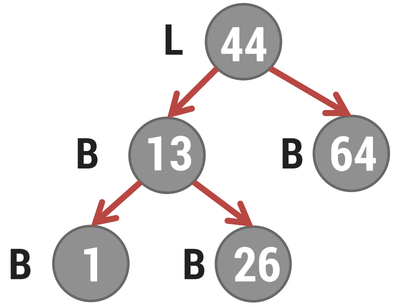



**Case 3**: Right Left Rotation
Right Rotation of Right Child **26**
Followed By
Left Rotation of Parent **1**

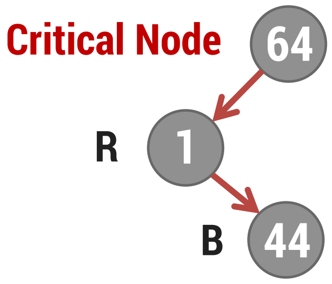
Right Rotation of Right Child **26**




Left Rotation of Parent **1**

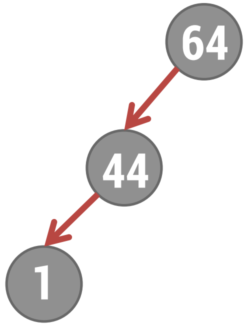



Insert **44**

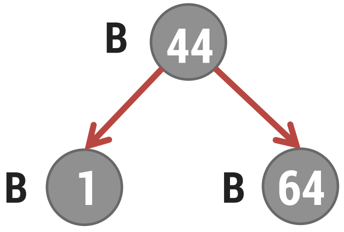



**Case 2**: Left Right Rotation
Left Rotation of Left Child **1**
Followed By
Right Rotation of Parent **64**

Left Rotation of Left Child **1**




Right Rotation of Parent **64**

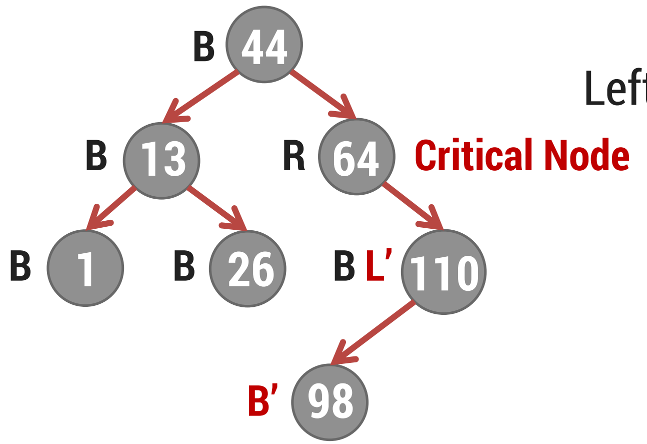


# Construct AVL Search Tree

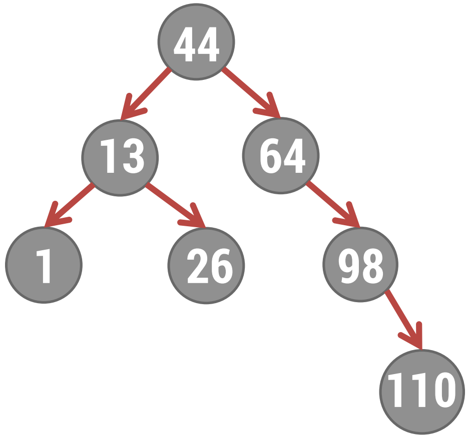
Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

**Insert 110 , 98**

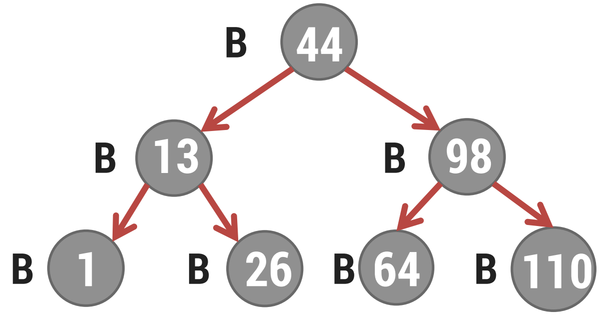
**Case 3**: Right Left Rotation
Right Rotation of Right Child 110
Followed By
Left Rotation of Parent 64



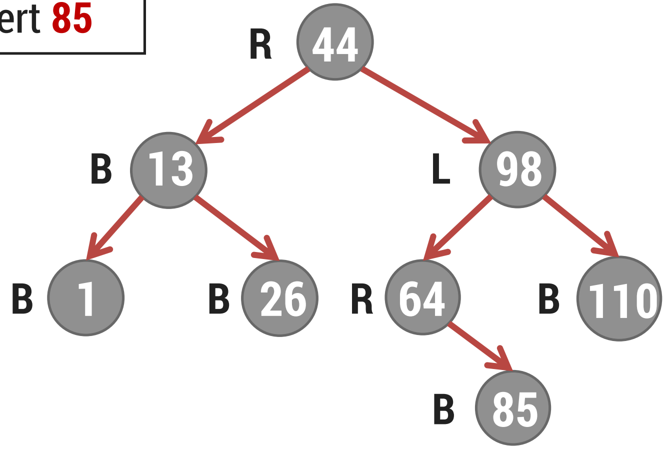

Right Rotation of Right Child **110**




Left Rotation of Parent **64**



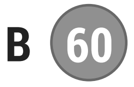

**Insert 85**



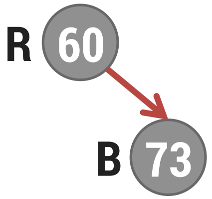
# Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,82,300,0,5,73**

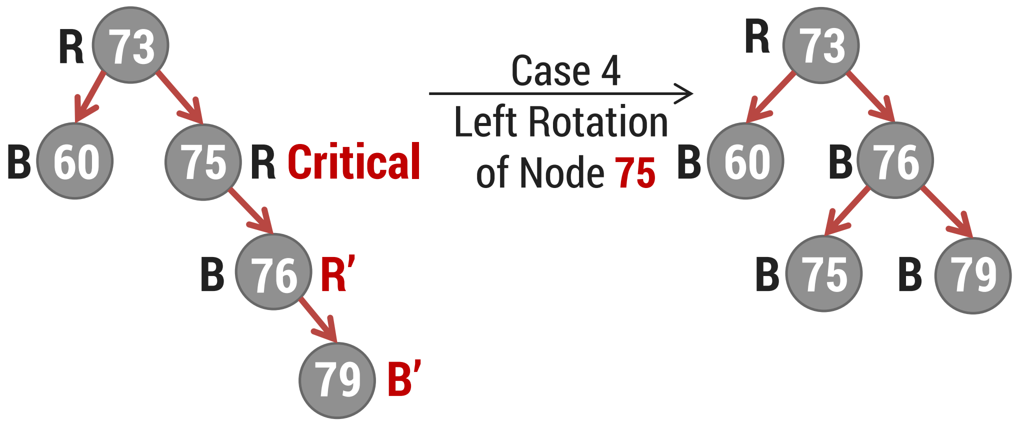
Insert **60**



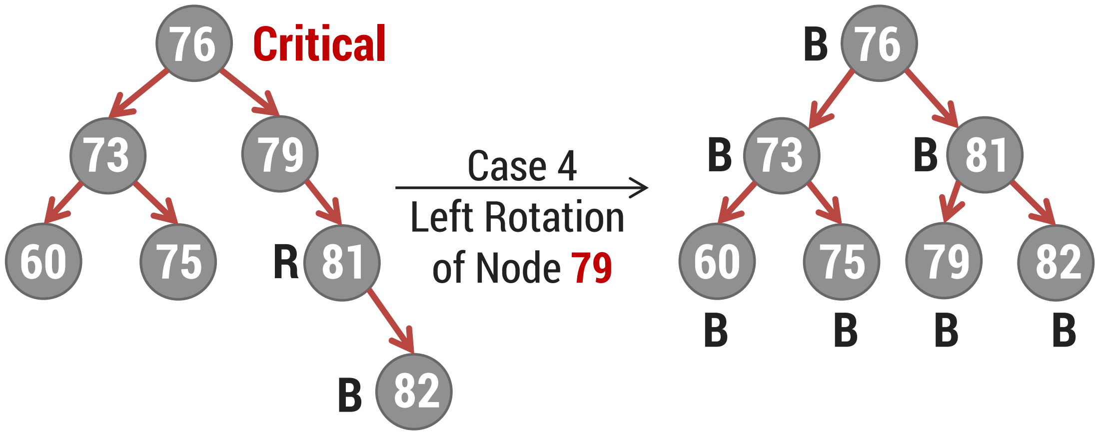

Insert **73**



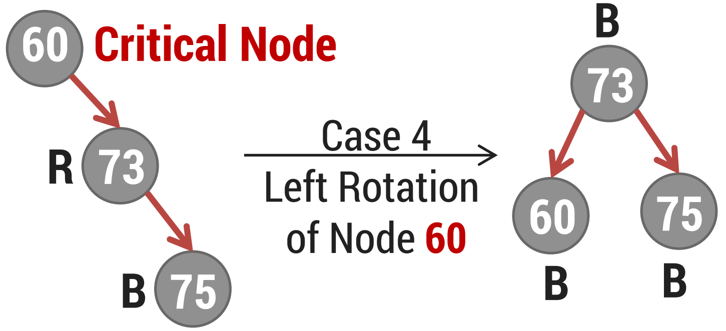

Insert **76 ,79**



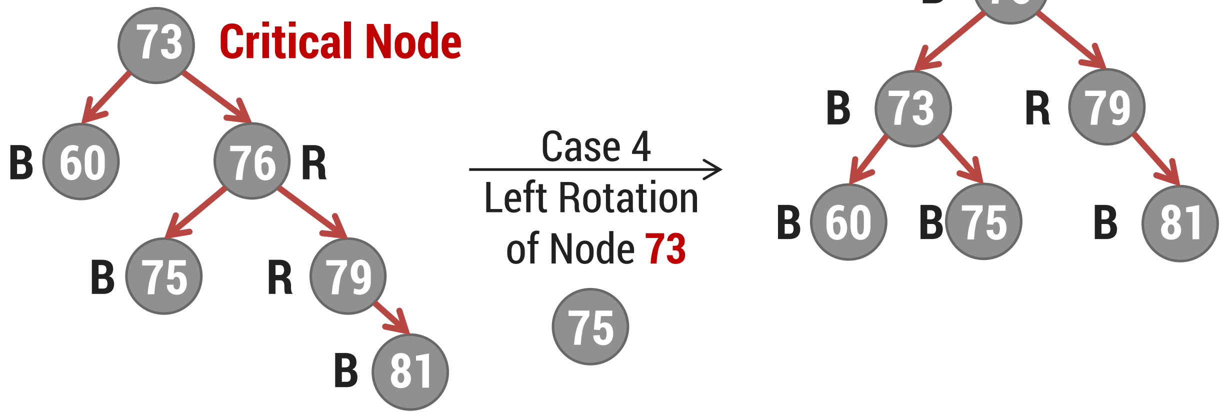

Insert **82**




Insert **75**



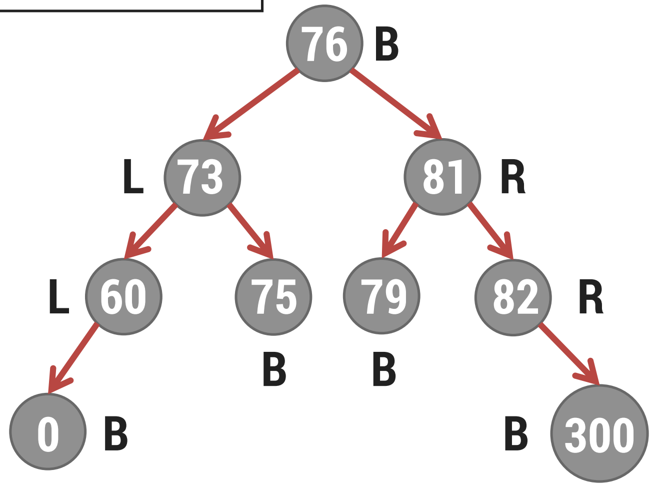

Insert **81**



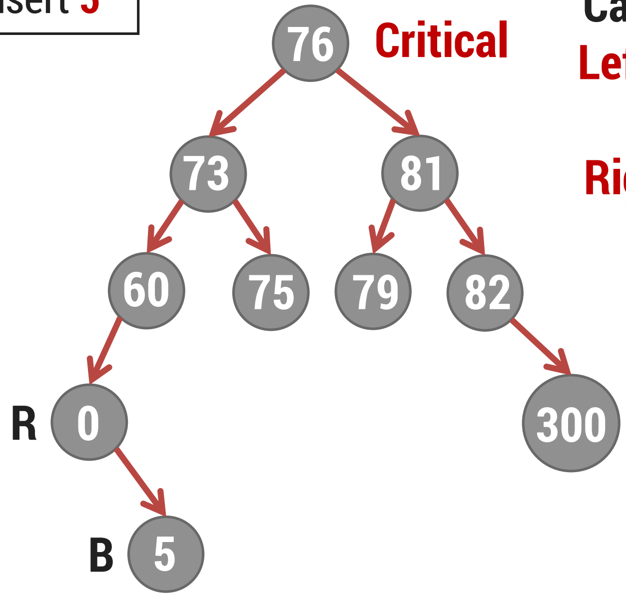
# Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,82,300,0,5,73**

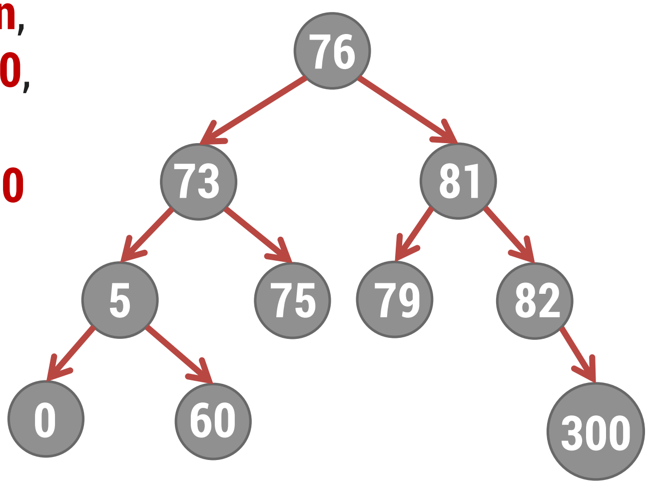
Insert **300 , 0**




Insert **5**




**Case 2: Left Right Rotation**,
**Left Rotation** of Left Child **0**,
Followed By
**Right Rotation** of Parent **60**

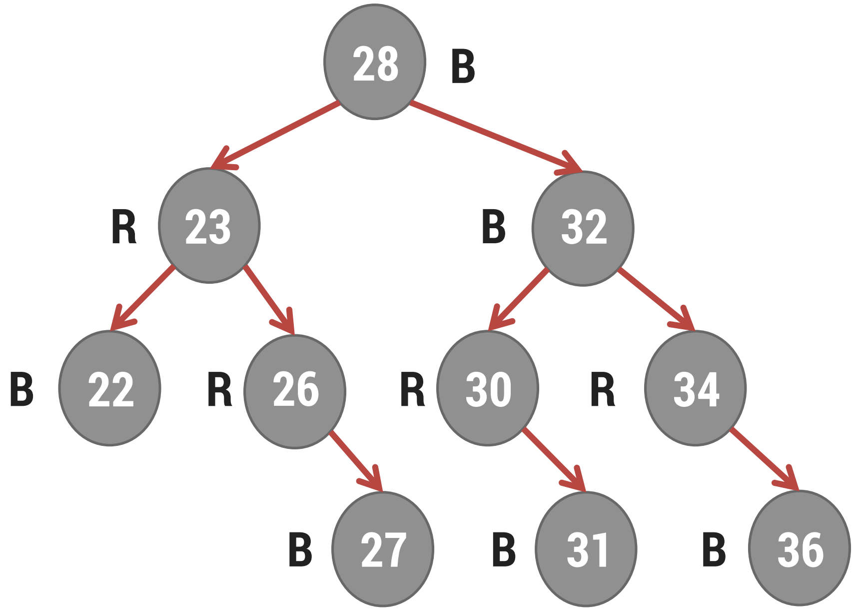



Insert **73**

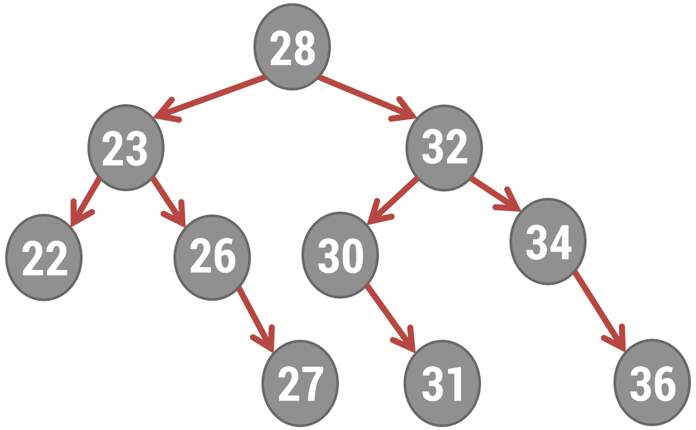
Can not Insert **73** as duplicate key found

# Deleting node from AVL Tree

- If element to be deleted **does not have empty right sub-tree**, then **element** is **replaced** with its **In-Order successor** and its **In-Order successor** is **deleted** instead
- During **winding up phase**, we need to **revisit every node** on the **path** from the **point of deletion** up to the **root**, rebalance the tree if require



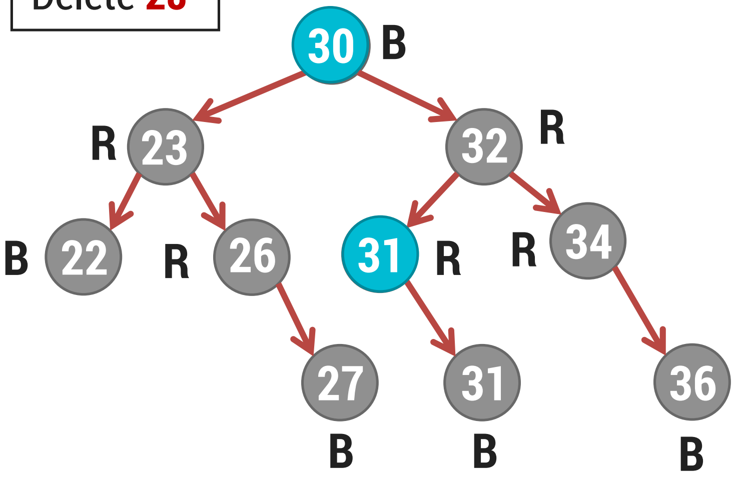
# Deleting node from AVL Tree



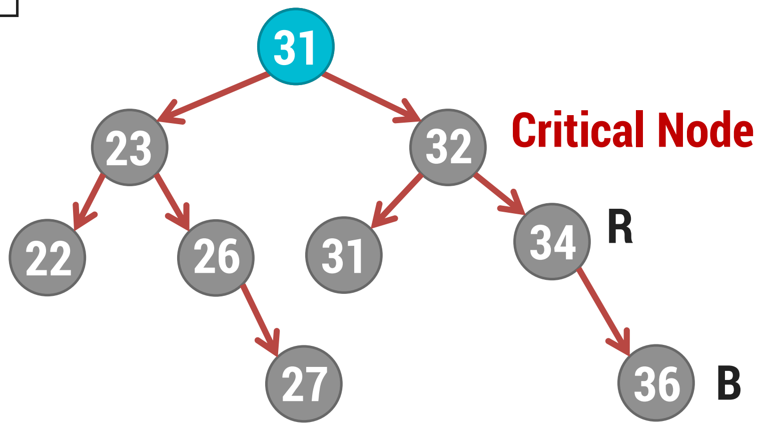

**In-Order Traversal**

**22, 23, 26, 27, 28, 30, 31, 32, 34, 36**

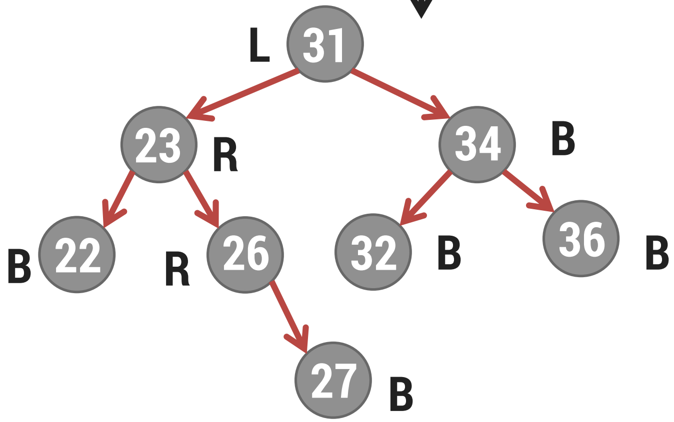
Delete **28**




Delete **30**

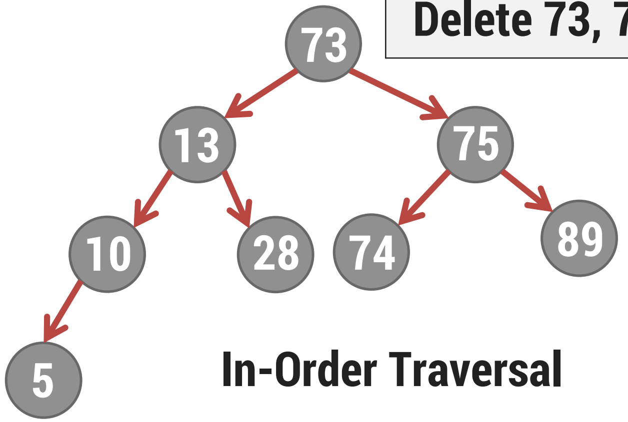



**Case 4:**
**Left Rotation of**
**Node 32**



# Deleting node from AVL Tree

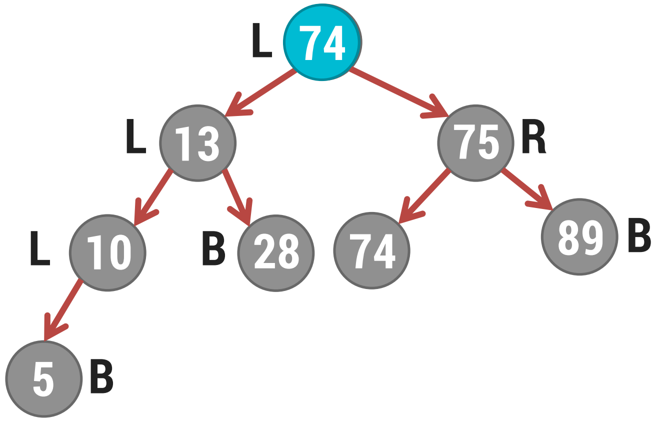
**Delete 73, 74**



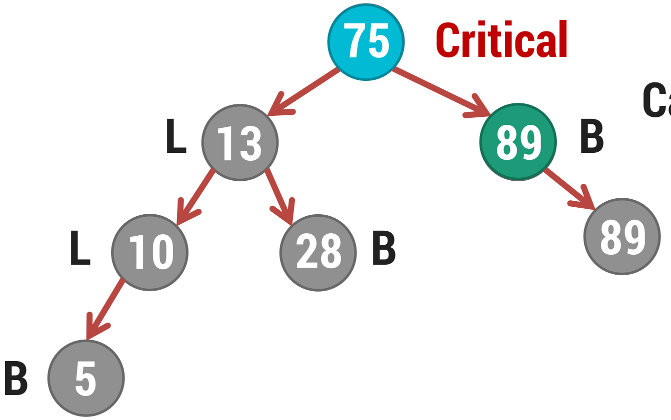

**In-Order Traversal**

**5, 10, 13, 28, 73, 74, 75, 89**

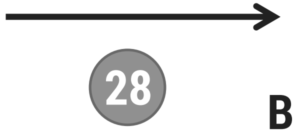
Delete **73**

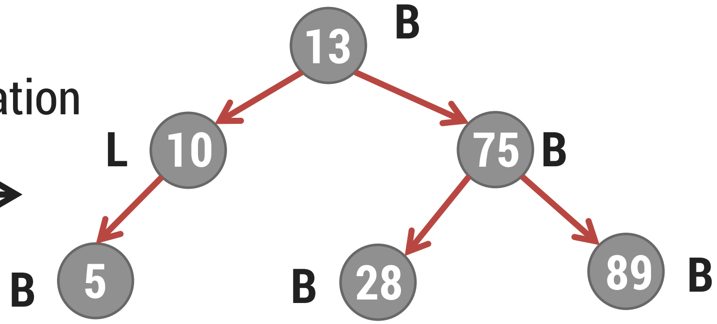



Delete **74**




**Case 1:** Right Rotation of Node **75**

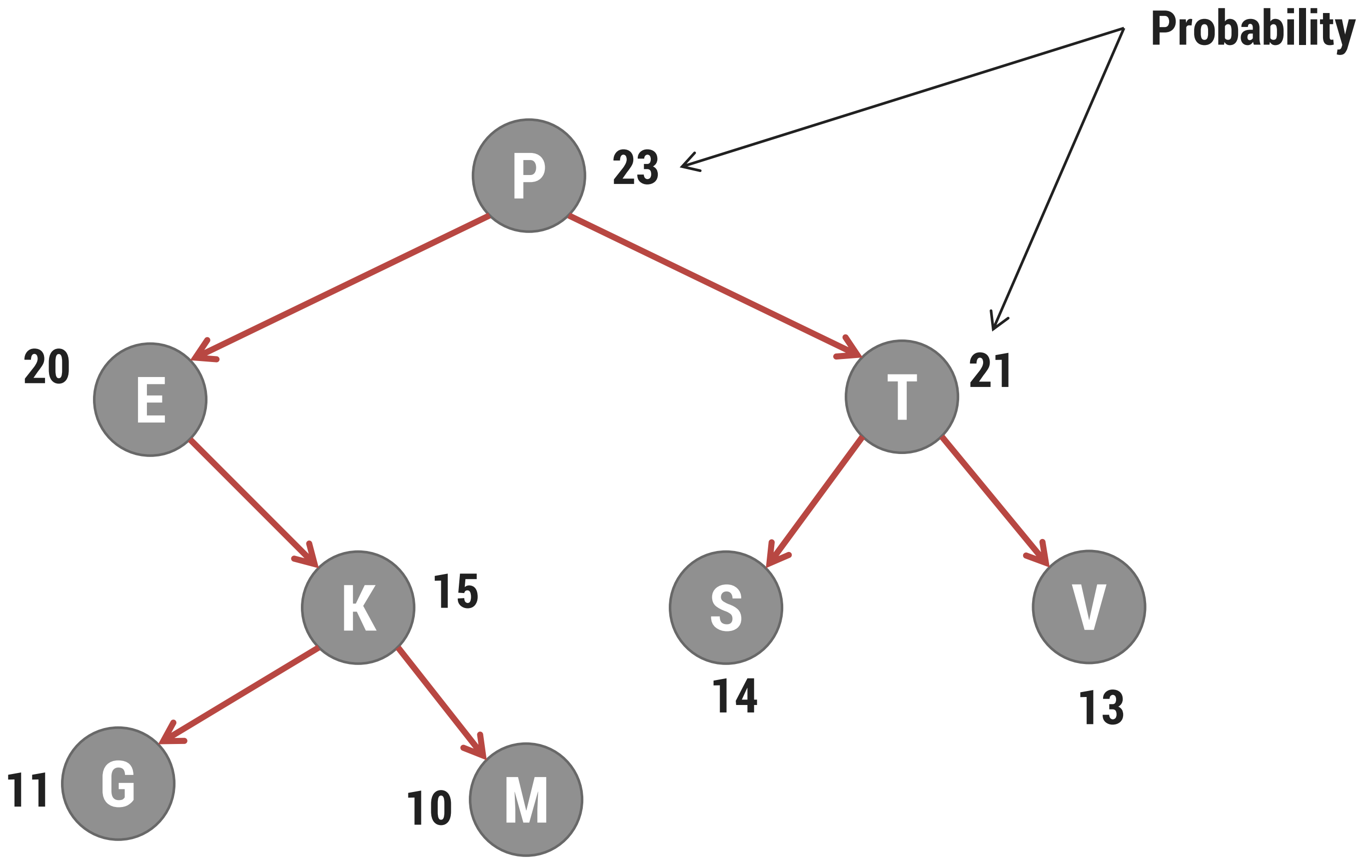


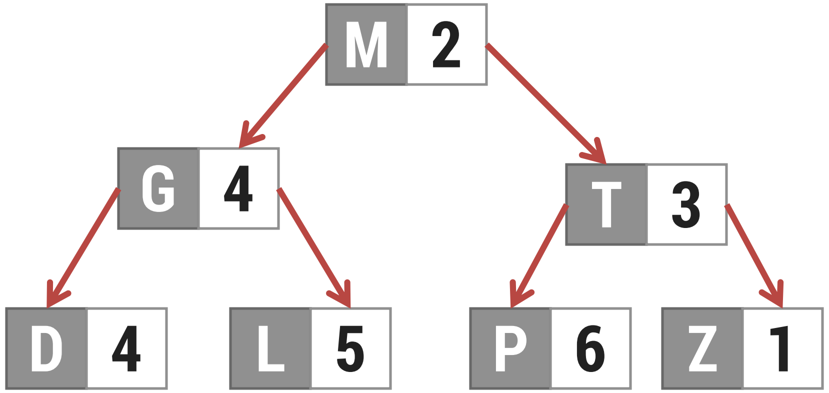
# Weight Balanced Tree

- In a **weight balanced tree**, the **nodes are arranged** on the **basis of** the knowledge available on the **probability for searching** each node
- The node with **highest probability** is placed at the **root** of the tree
- The **nodes** in the **left sub-tree** are **less in ranking** as well as **less in probability** then the root node
- The **nodes** in the **right sub-tree** are **higher in ranking** but **less in probability** then the root node
- Each node of such a Tree has an information field contains the value of the node and count number of times node has been visited

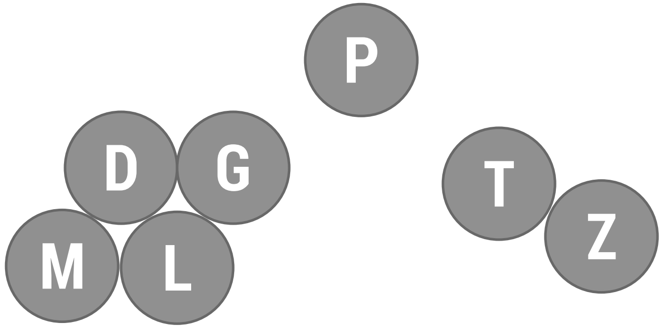
# Weight Balanced Tree

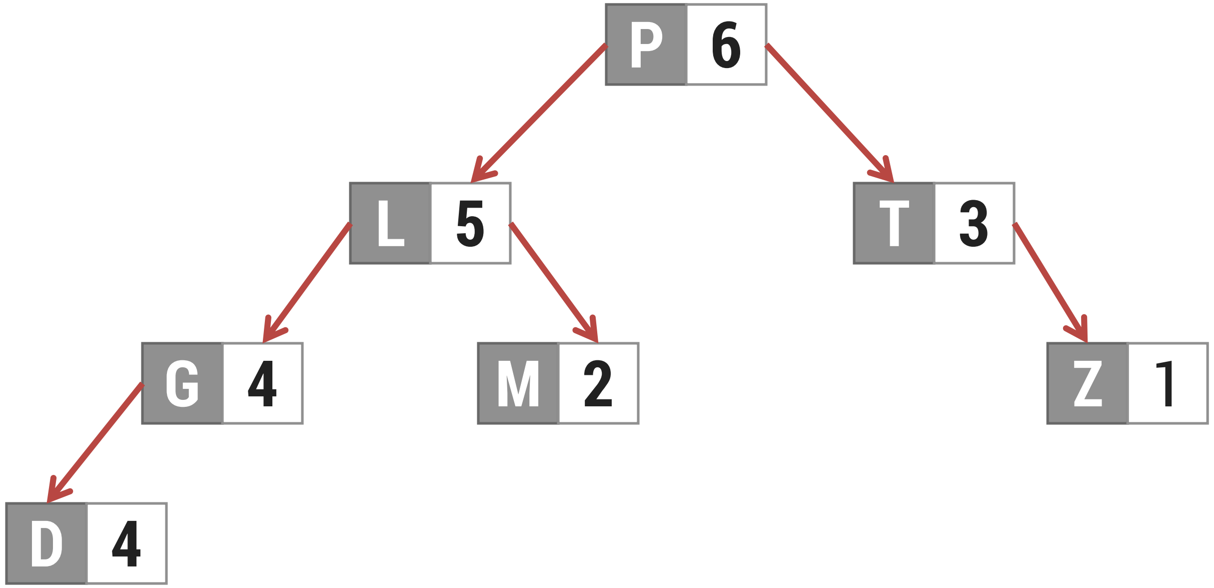


# Weight Balanced Tree



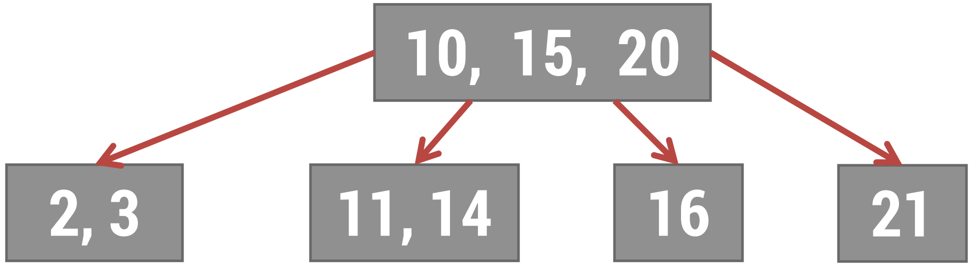

Ordered Tree



# Multiway Search Tree (B - Tree)

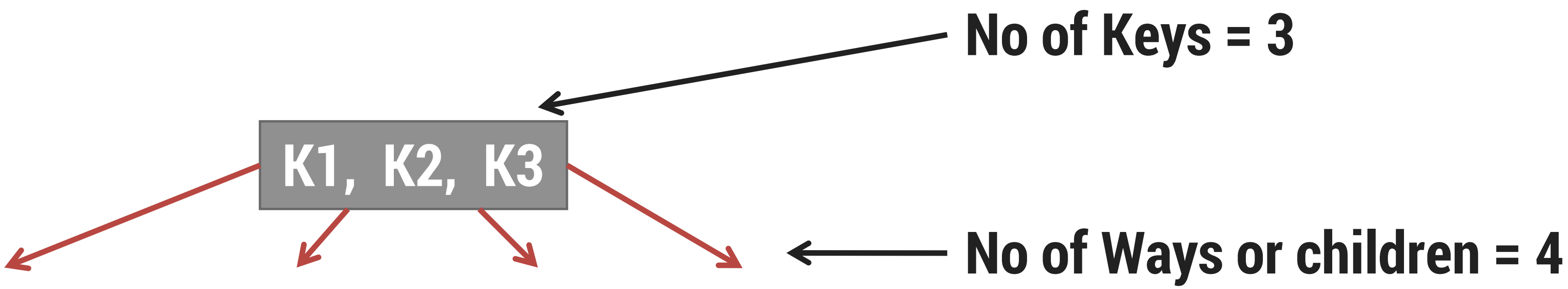
- The **nodes** in a **binary tree** like AVL tree **contains only one record**
- **AVL tree is** commonly **stored in primary memory**
- In **database applications** where **huge volume** of **data** is handled, the **search tree** can **not be accommodated** in **primary memory**
- **B-Trees** are primarily meant for **secondary storage**
- **B-Tree** is a **M-way tree** which can have **maximum of M Children**




**4 – way Tree**

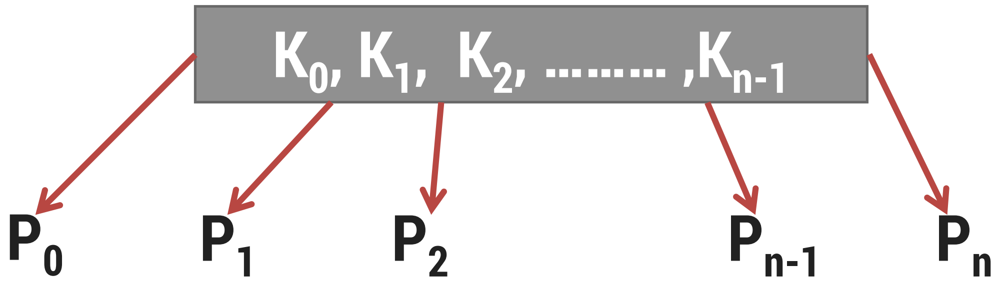
# Multiway Search Tree (B - Tree)

- An **M- way** tree contains **multiple keys** in a node
- This leads to **reduction** in overall **height** of the tree
- If a **node** of M-way tree **holds K keys** then it will have **k+1 children**

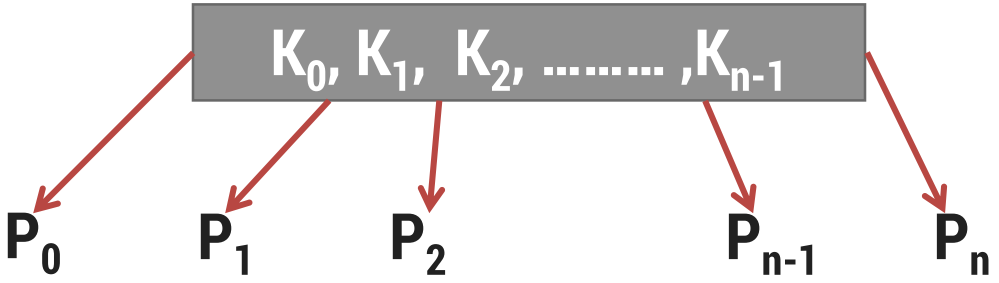


# Multiway Search Tree (B - Tree)

- A **tree** of **order M** is a **M-way** search tree with the following properties
  1. The **Root** can have **1 to M-1 keys**
  2. All **nodes** (**except Root**) have **(M-1)/2** to **(M-1) keys**
  3. All **leaves** are at the **same level**
  4. If a node has **‘t’** number of **children**, then it must have **‘t-1’ keys**
  5. **Keys** of the nodes are stored in **ascending order**

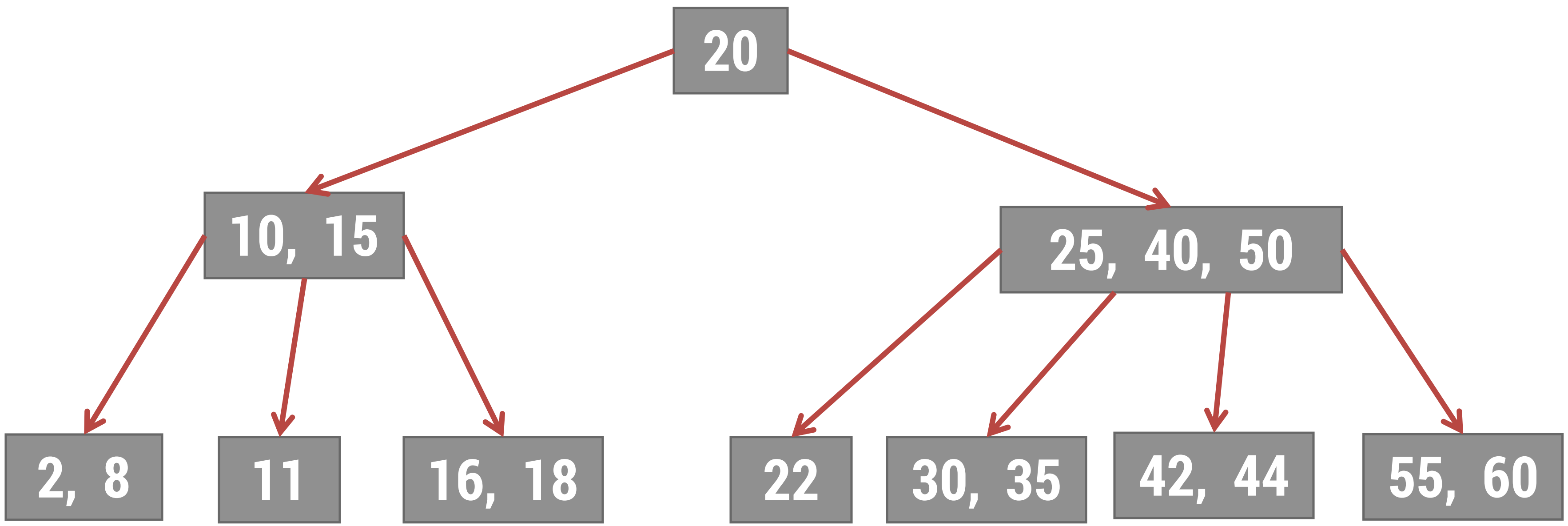


# Multiway Search Tree (B - Tree)




- $K_0, K_1, K_2, \ldots\ldots\ldots, K_{n-1}$ are **keys** stored in the node
- **Sub-Trees** are pointed by $P_0, P_1, P_2, \ldots\ldots\ldots, P_n$ then
  - $K_0$ >= all keys of sub-tree $P_0$
  - $K_1$ >= all keys of sub-tree $P_1$
  - ……….
  - ……….
  - $K_{n-1}$ >= all keys of sub-tree $P_{n-1}$
  - $K_{n-1}$ < all keys of sub-tree $P_n$

# Multiway Search Tree (B - Tree)




**B-Tree of Order 4 (4 way Tree)**

# Insertion of Key in B-Tree

1. If **Root** is **NULL**, **construct** a node and **insert key**
2. If **Root** is **NOT NULL**
   I. Find the **correct leaf** node to which key should be added
   II. If **leaf node has space** to accommodate key, it is **inserted** and **sorted**
   III. If **leaf node does not have space** to accommodate key, we **split node** into two parts

# Split Node (5 way Tree, max 4 Keys)

P

5, 10, 15, 20

Insert - 3

3, 5, 10, 15, 20

Overflow

10

3, 5

15, 20

1, 5, 9, 11

Insert - 3

Overflow

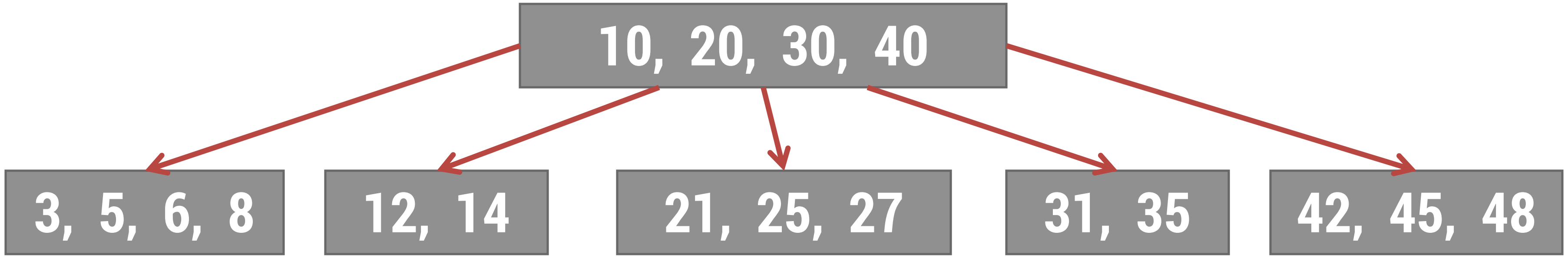
1, 3, 5, 9, 11

5

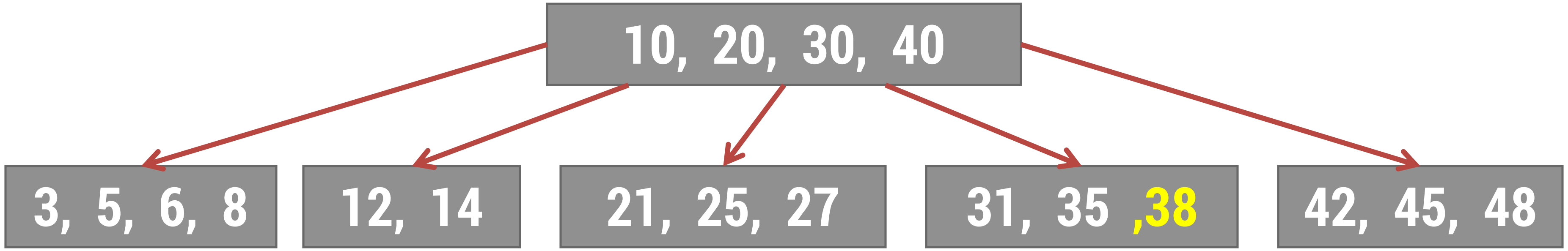
1, 3

9, 11

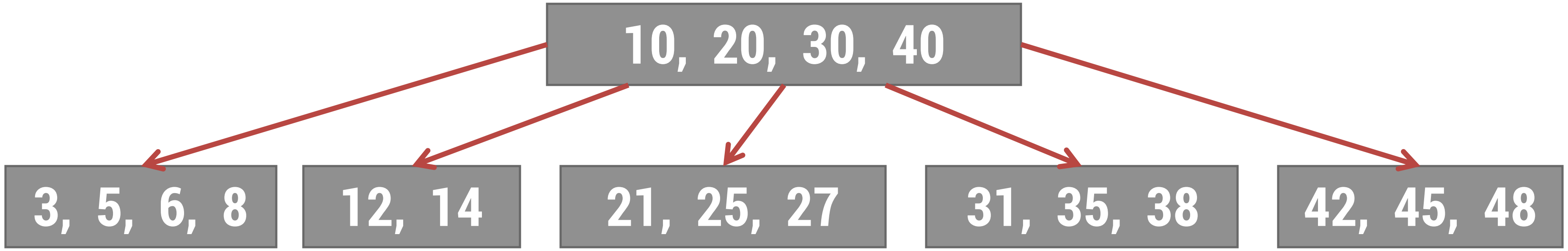
# Split Node (5 way Tree, max 4 Keys)



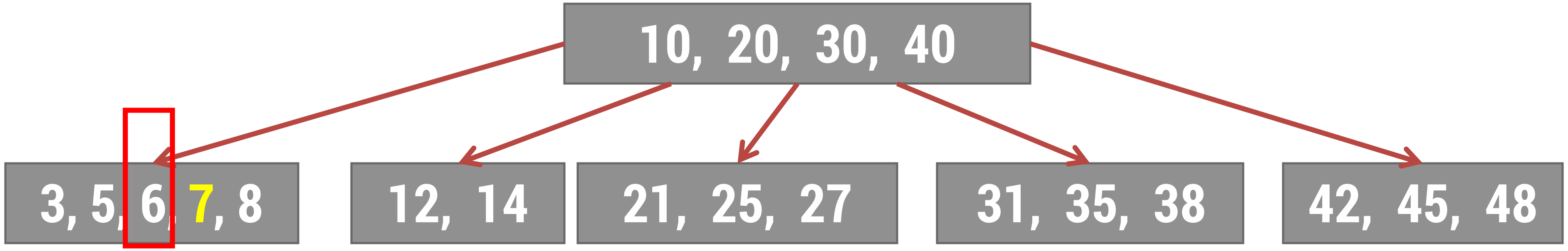

**Insert - 38**



# Split Node (5 way Tree, max 4 Keys)

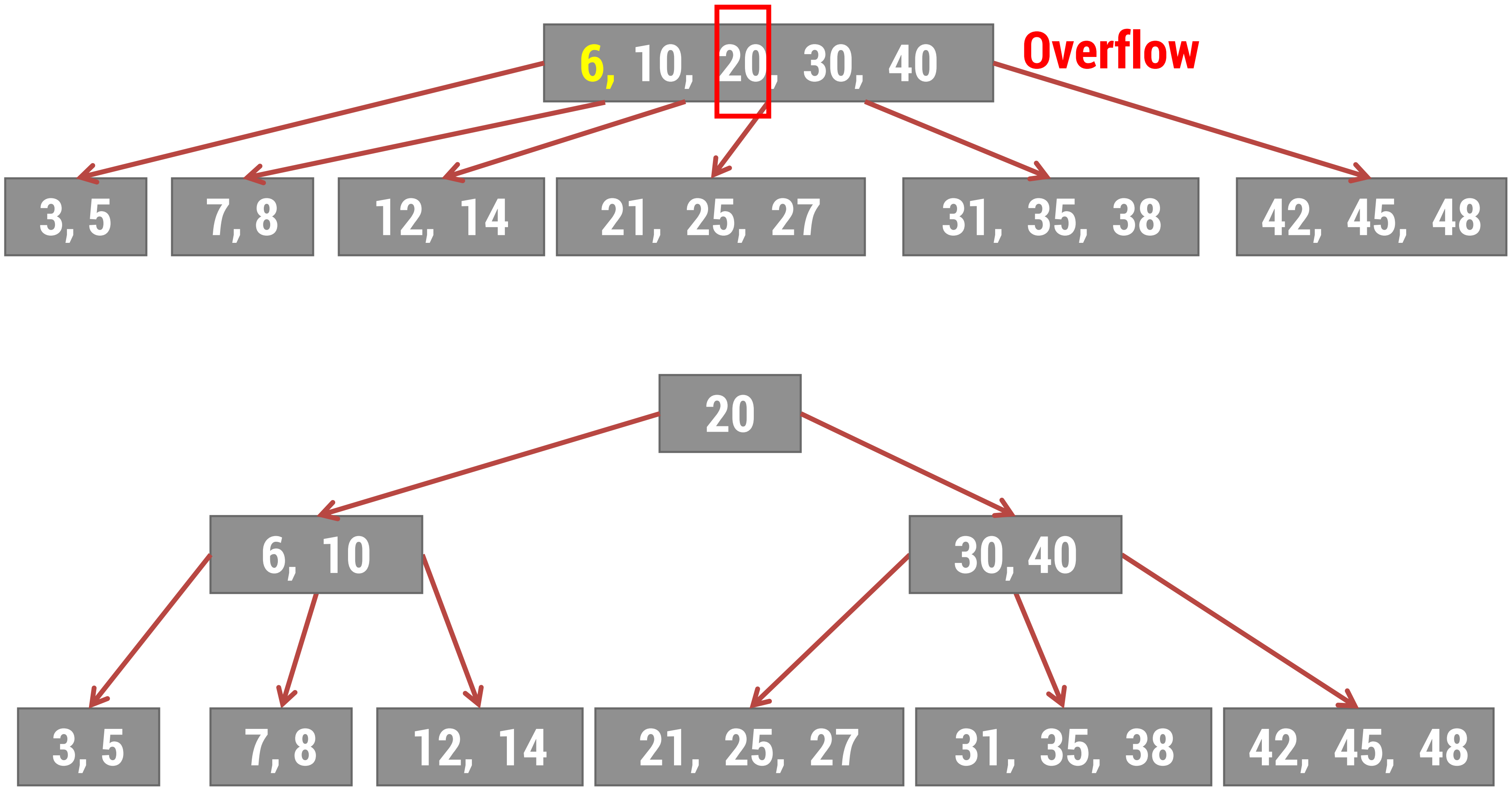



**Insert 7**




**Overflow**

# Split Node (5 way Tree, max 4 Keys)

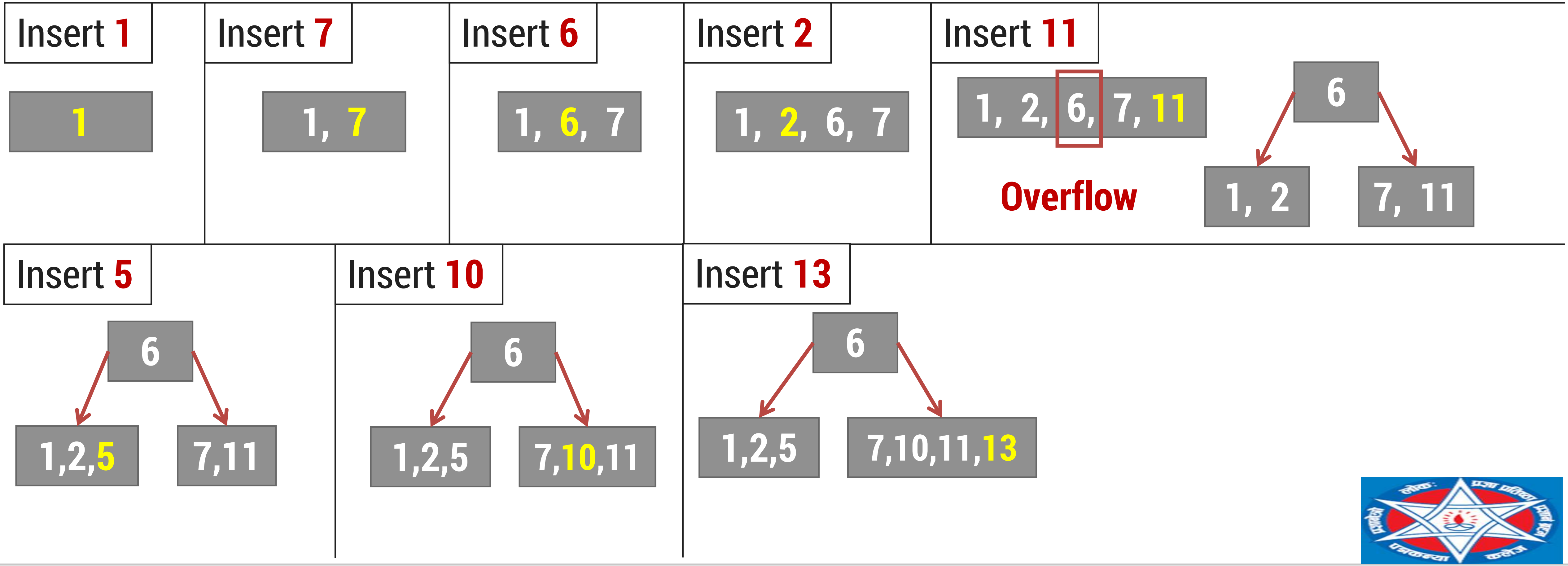


Construct **5 Order (5 Way)** Tree from following data

1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25

---

We are asked to **create 5 Order Tree (5 Way Tree) maximum 4 records** can be accommodated in a node

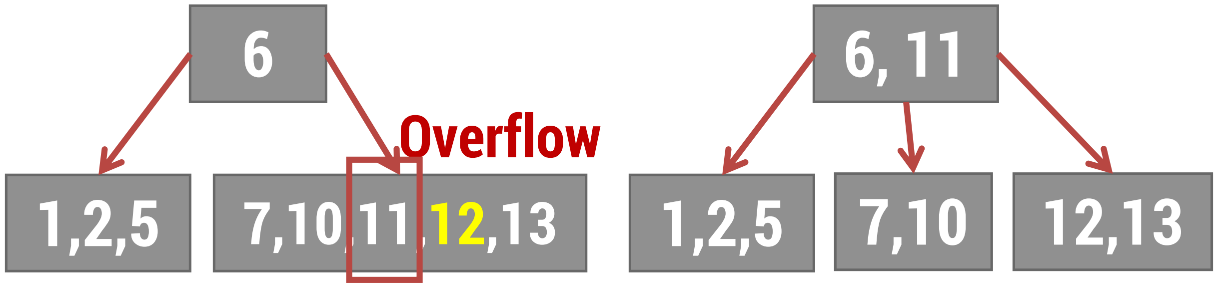


# Construct M-Way Tree

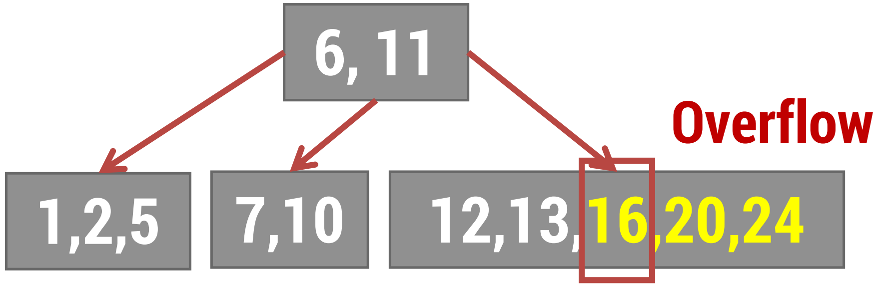
Construct **5 Order (5 Way)** Tree from following data

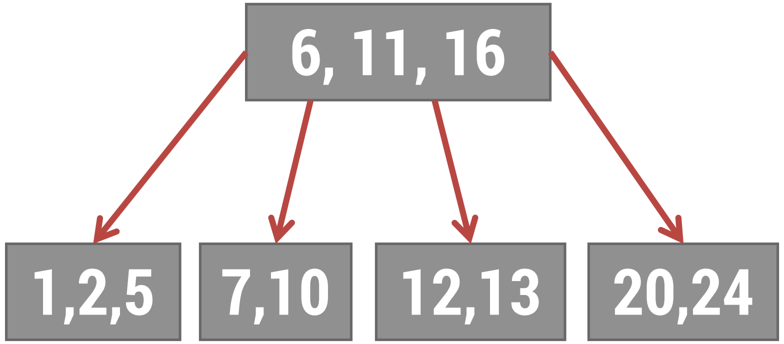
1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25

## Insert 12

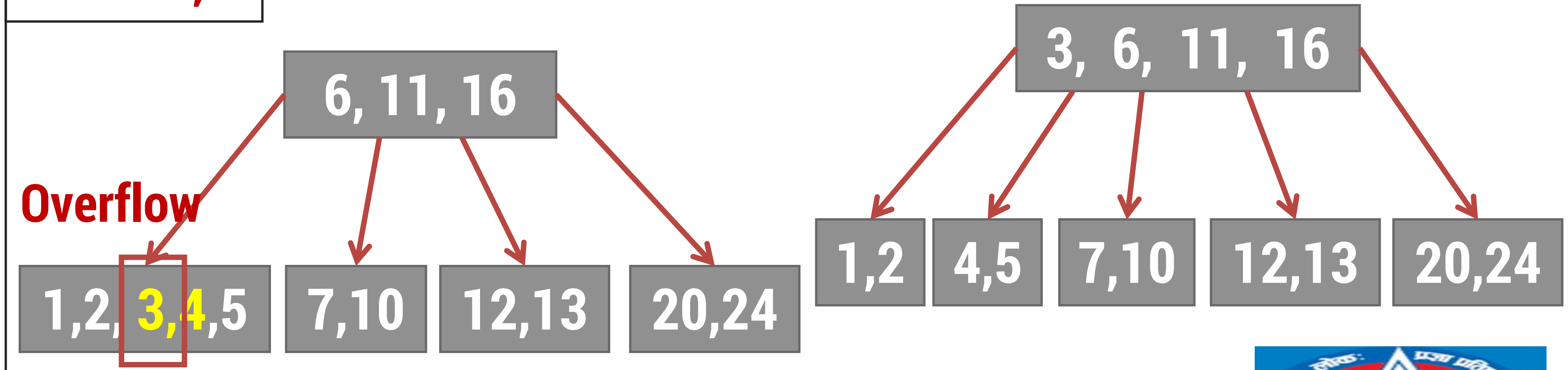



## Insert 20, 16, 24




## Insert 3,4

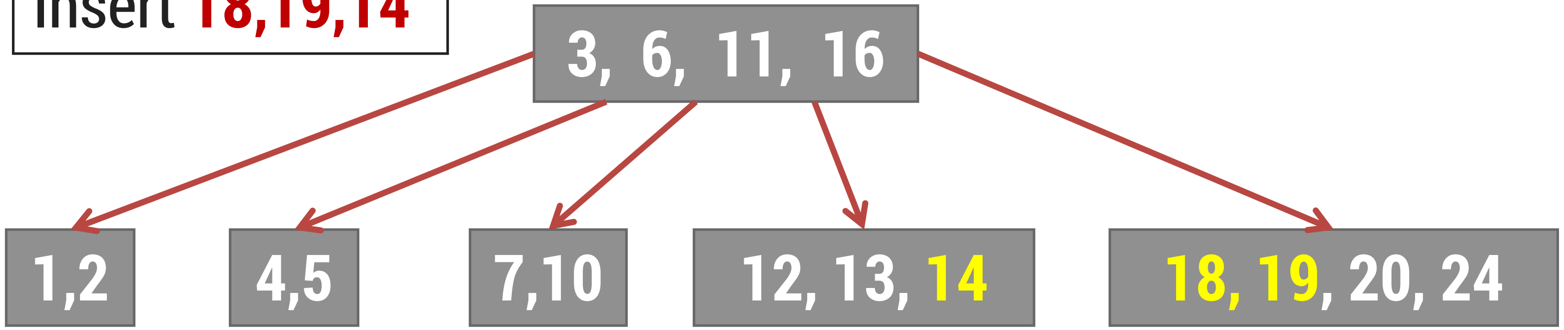


# Construct M-Way Tree

Construct **5 Order (5 Way)** Tree from following data

1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25

Insert **18,19,14**




Insert **25**