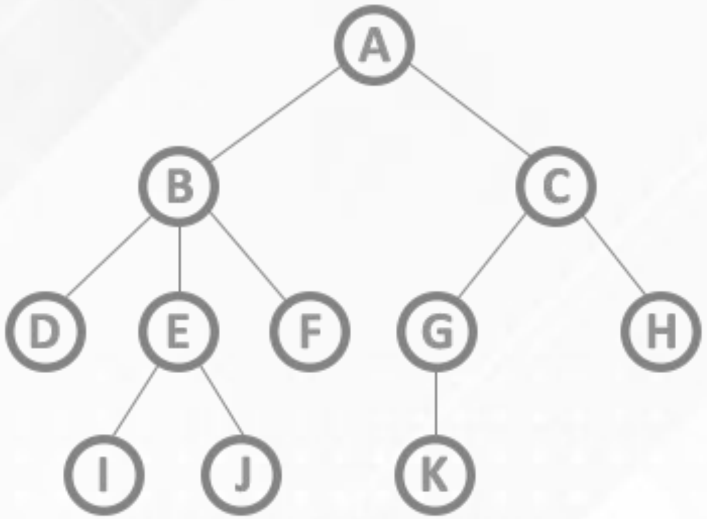
Unit-3

Non-Linear Data Structure (Tree)

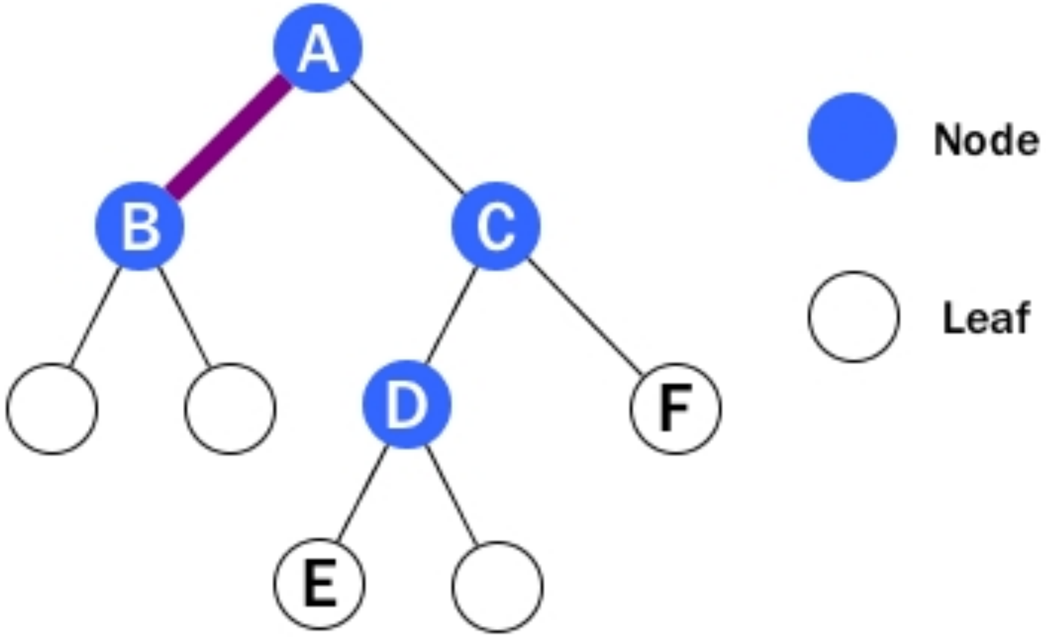
Part 3

- Height/Weight – Balanced Tree
- Multiway Search Tree (B-Tree)



Binary Tree

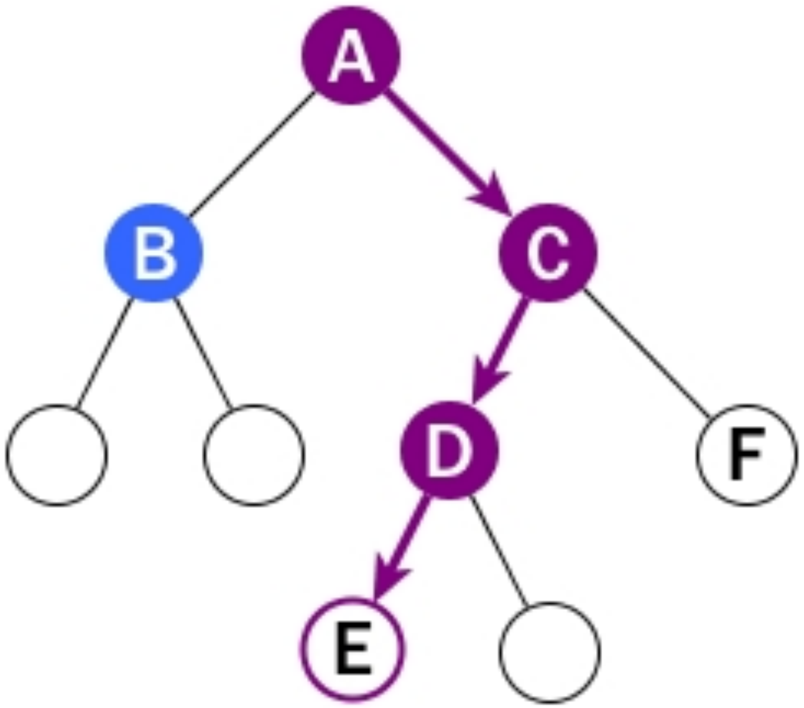
- **Edge** – connection between one node to another.



About Edge

Binary Tree

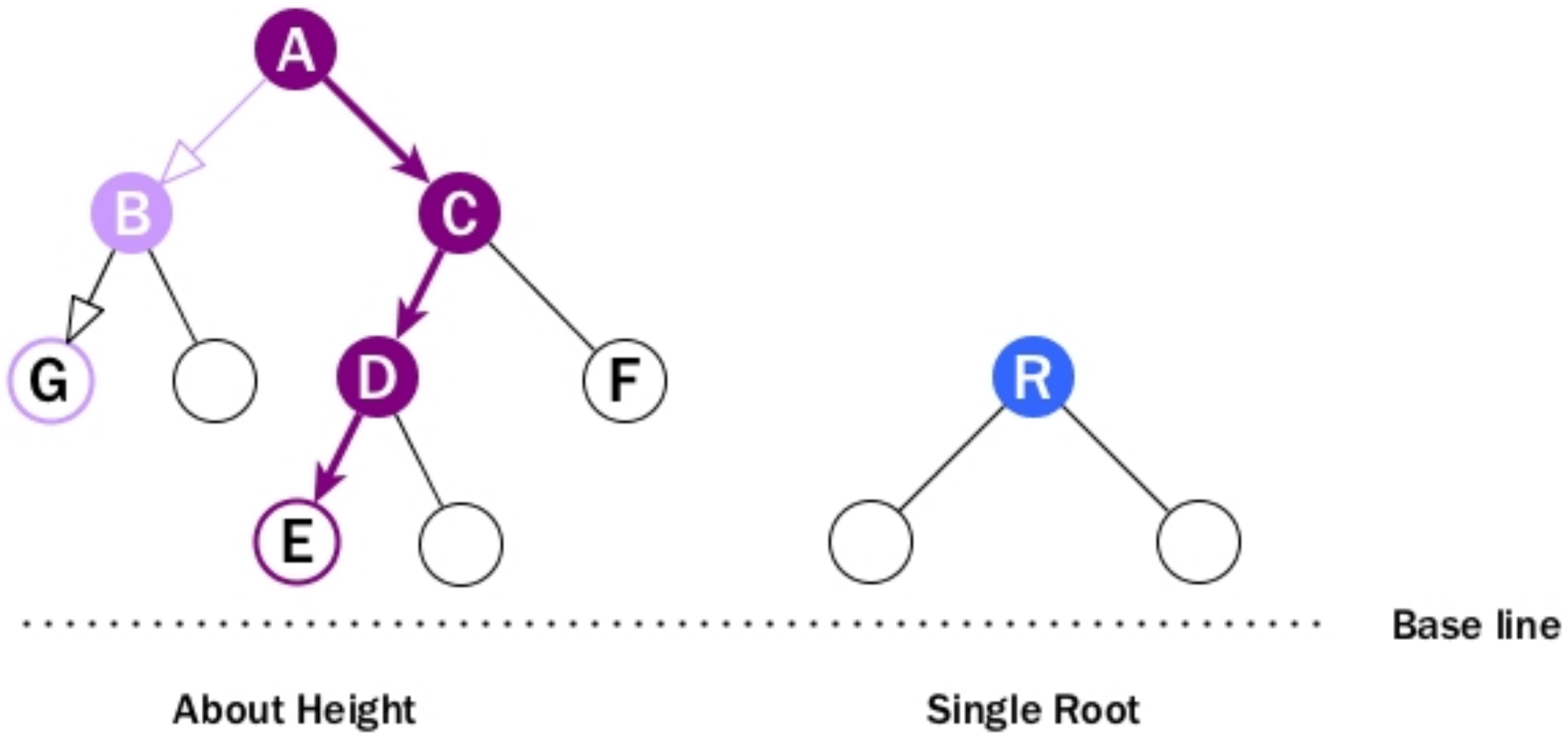
- **Path** – *a sequence of nodes and edges connecting a node with a descendant.*



About Path

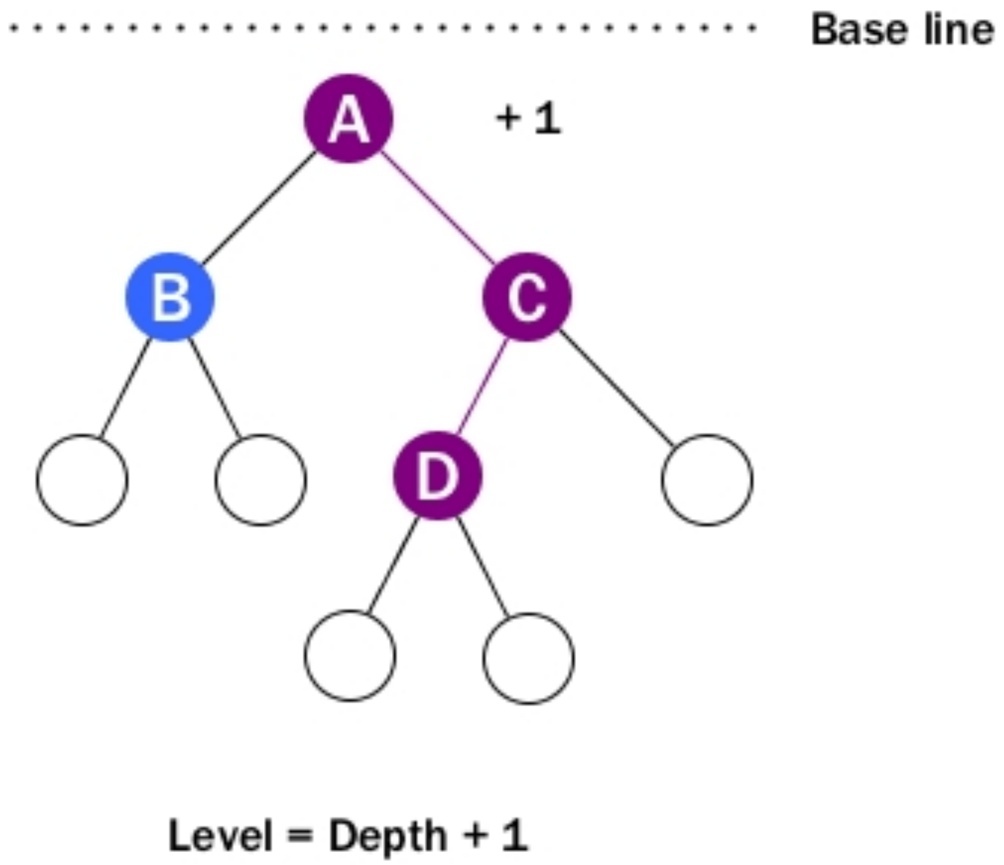
Binary Tree

- **Height of node** – *The height of a node is the number of edges on the longest downward path between that node and a leaf.*
- Note that **the depth** / **height of the root is 0**.



Binary Tree

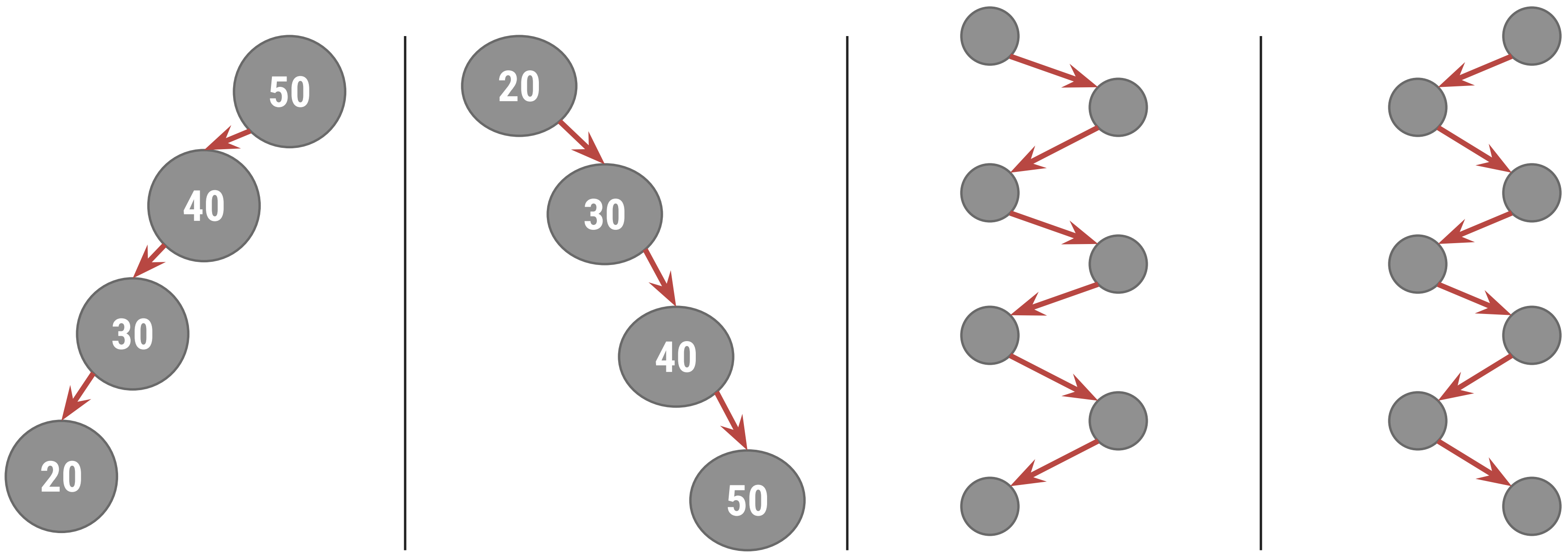
- **Level** – *The level of a node is defined by 1 + the number of connections between the node and the root.*
- The important thing to remember is when talking about level, it **starts from 1** and **the level of the root is 1**. We need to be careful about this when solving problems related to level.



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

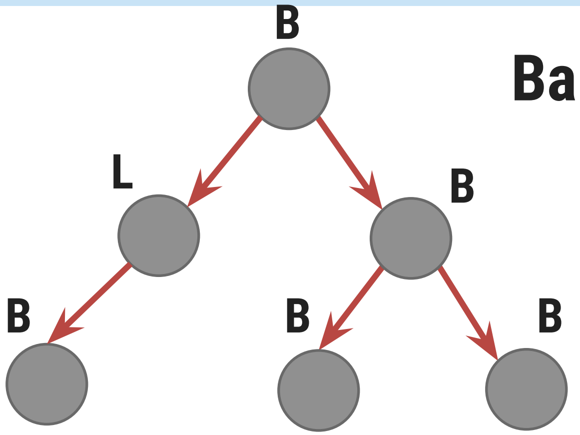


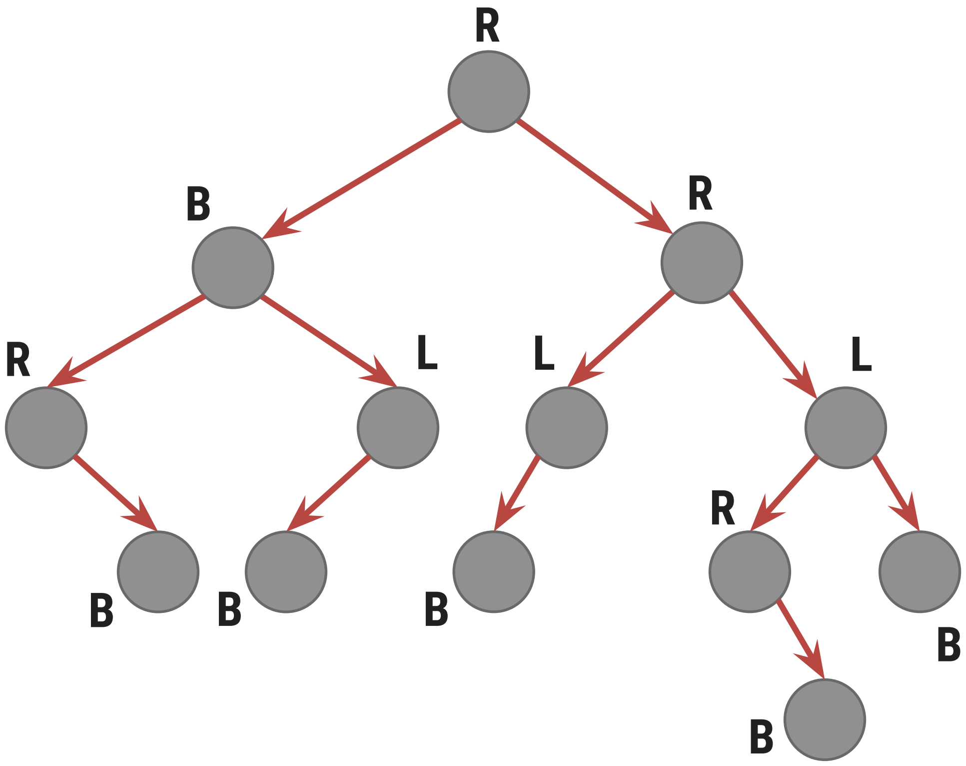
Height Balanced Tree (AVL Tree)

- A tree is called **AVL tree (Height Balanced Tree)**, if each node possessed one of the following properties
  - A **node** is called **left heavy (L)**, 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 (R)**, 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 (B)**, 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**
- **If all the node have balance factor = 1 (left heavy, L) or 0 (balanced, B) or -1 (right heavy, R), then tree is height balanced tree**
- **Balance factor = (Height of Left SubTree – Height of Right SubTree) i.e. [BF = H_{Lt} – H_{Rt}]**

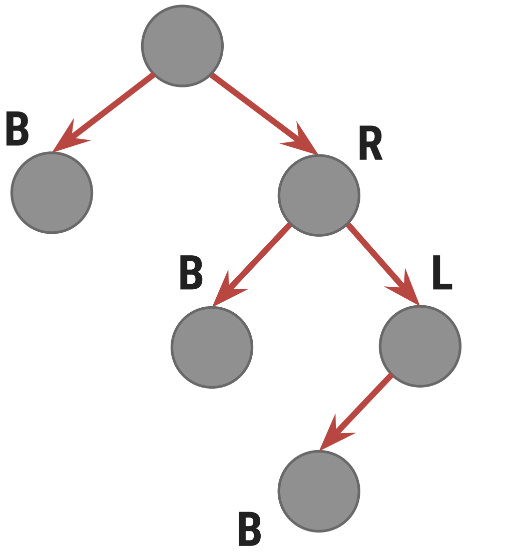
AVL Tree

Balanced Trees

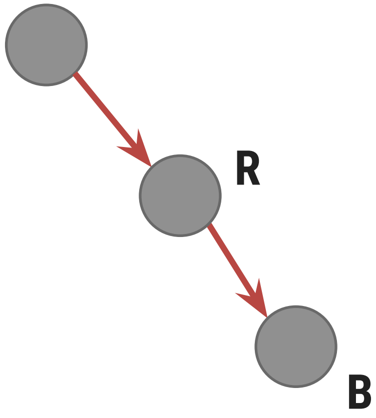




Critical Node
Unbalanced Node



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**

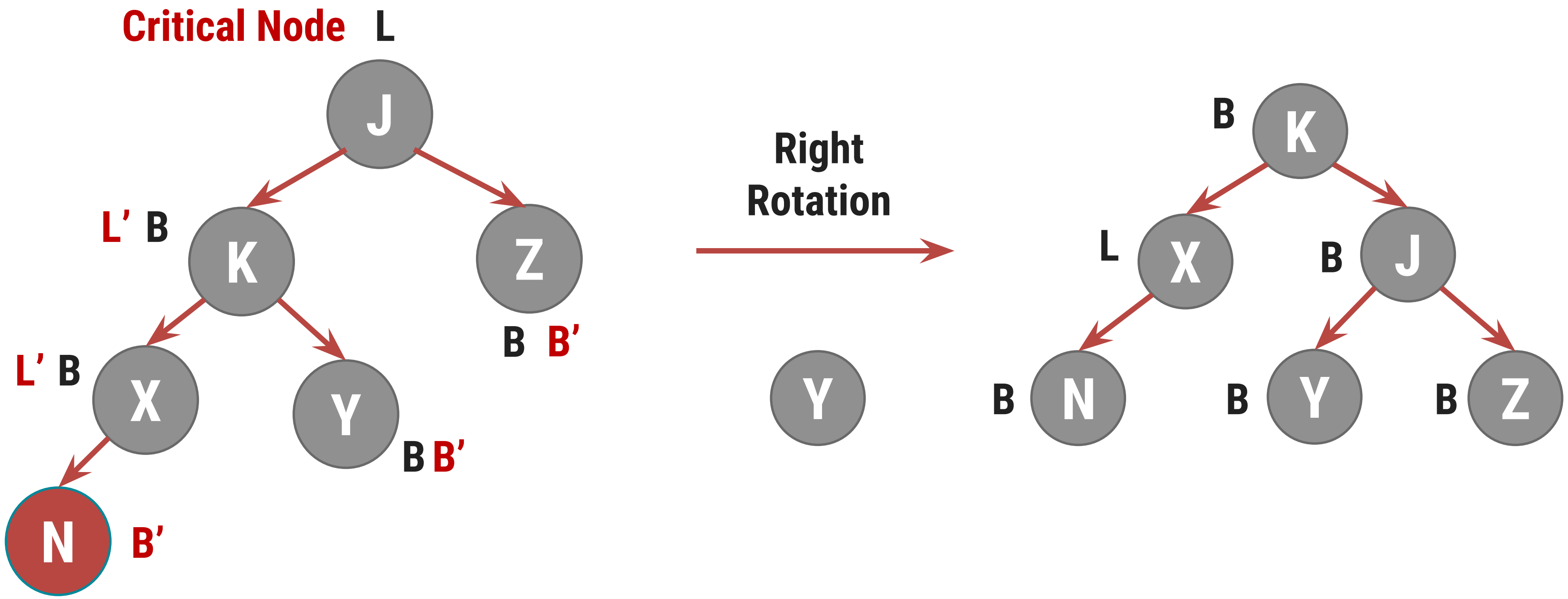
Right Rotation

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**



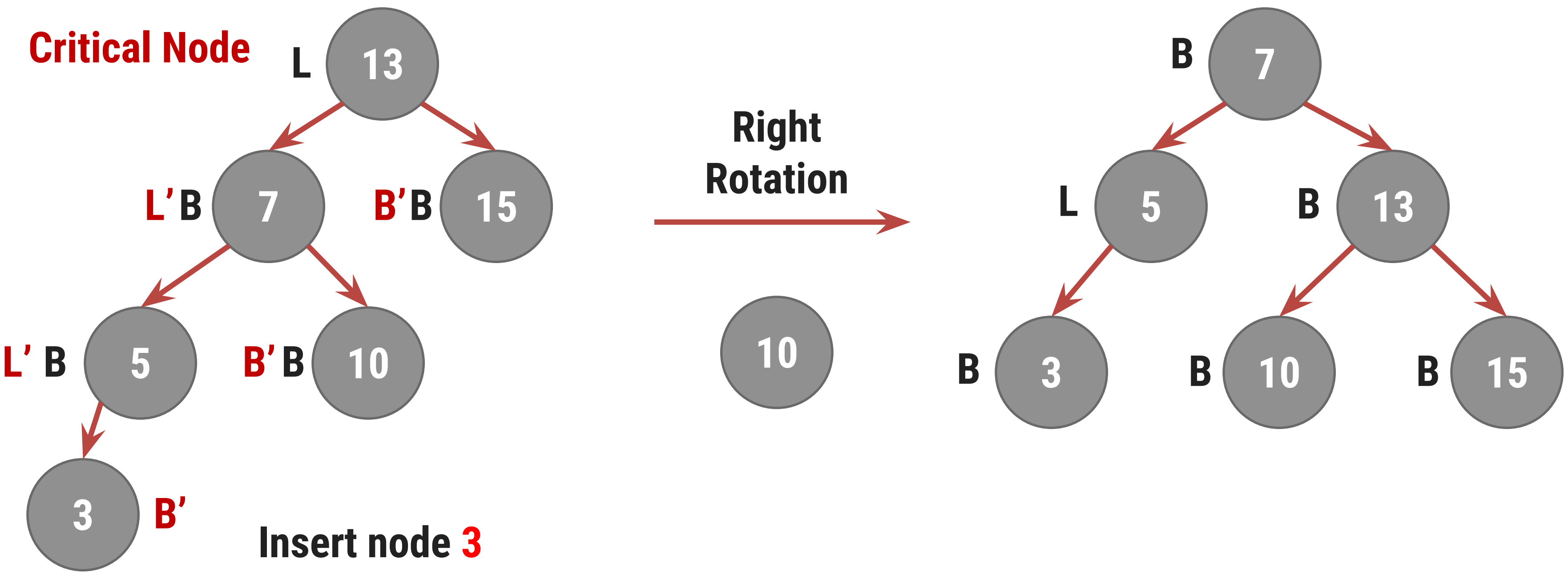
Right Rotation

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**



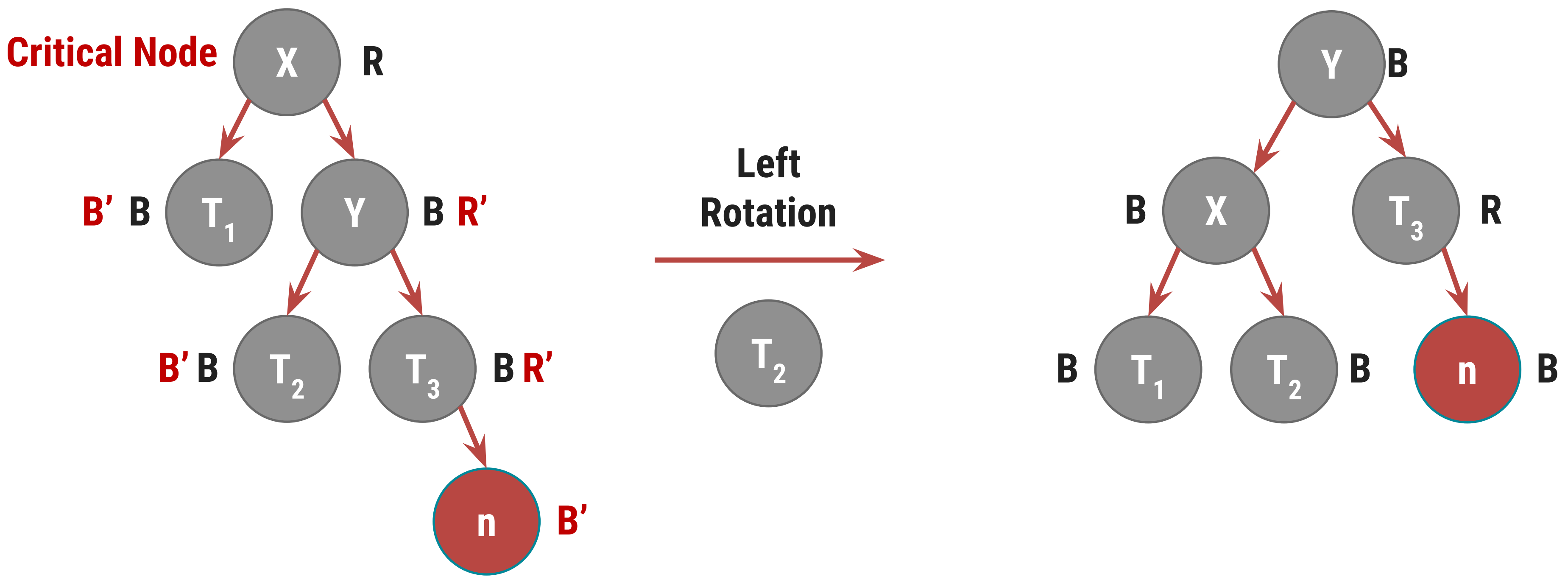
Left Rotation

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

Remember : (Insertion) in (SubTree)

Case 1: Insertion into **Left sub-tree** of **nodes Left child**

- **Single - Right Rotation** **[L L -> R]**

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

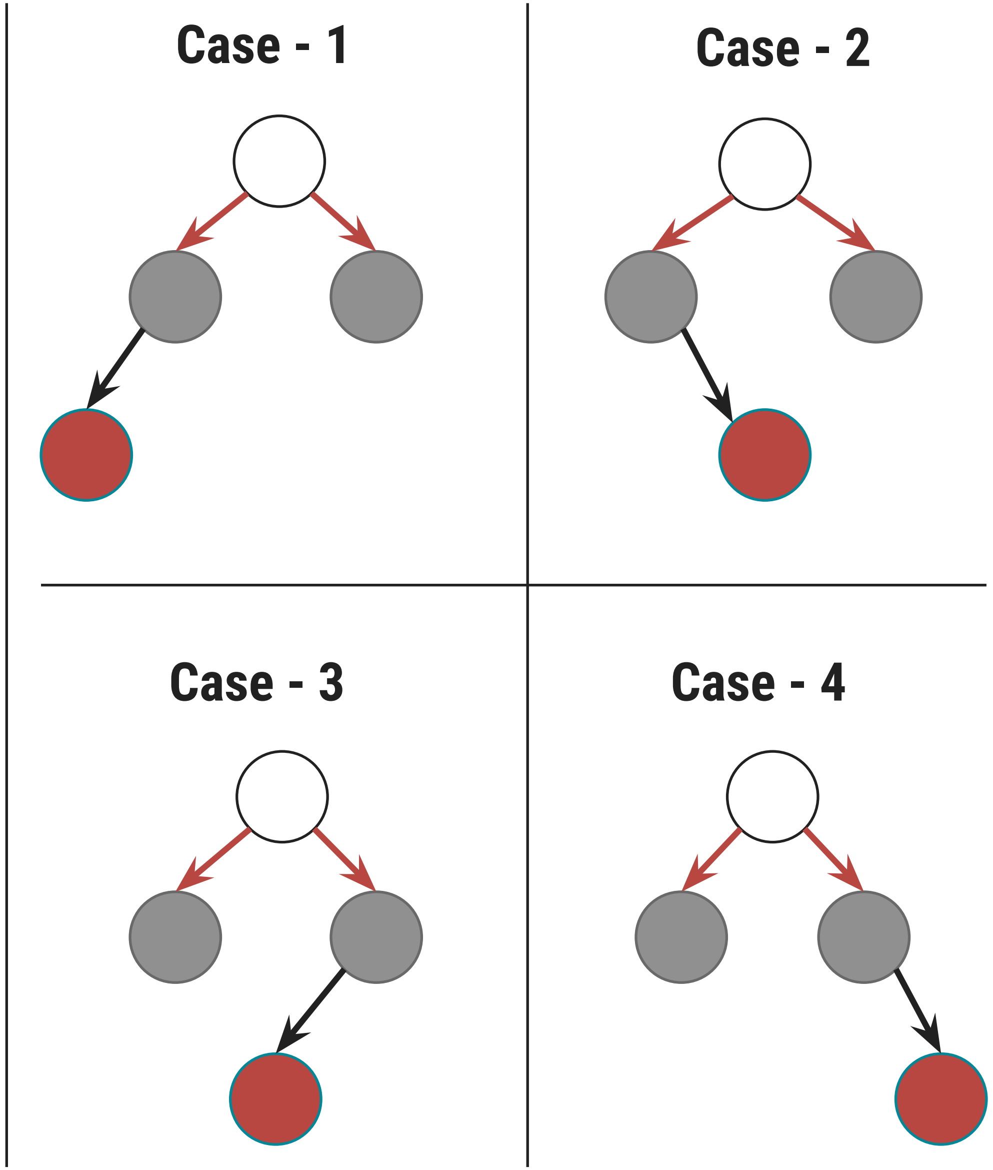
- **Double - Left Right Rotation** **[R L -> L R]**

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

- **Double - Right Left Rotation** **[L R -> R L]**

Case 4: Insertion into **Right sub-tree** of **node's Right child**

- **Single - Left Rotation** **[R R -> L]**



Insertion into Left sub-tree of nodes Left child : [L L -> R]

- **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**

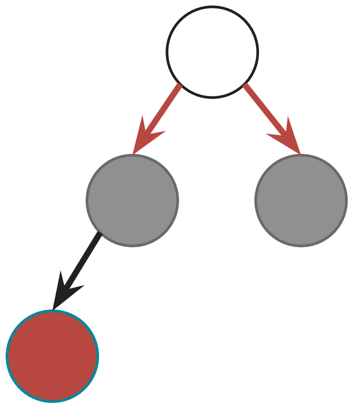
  a. Detach leaf child's right sub-tree

  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

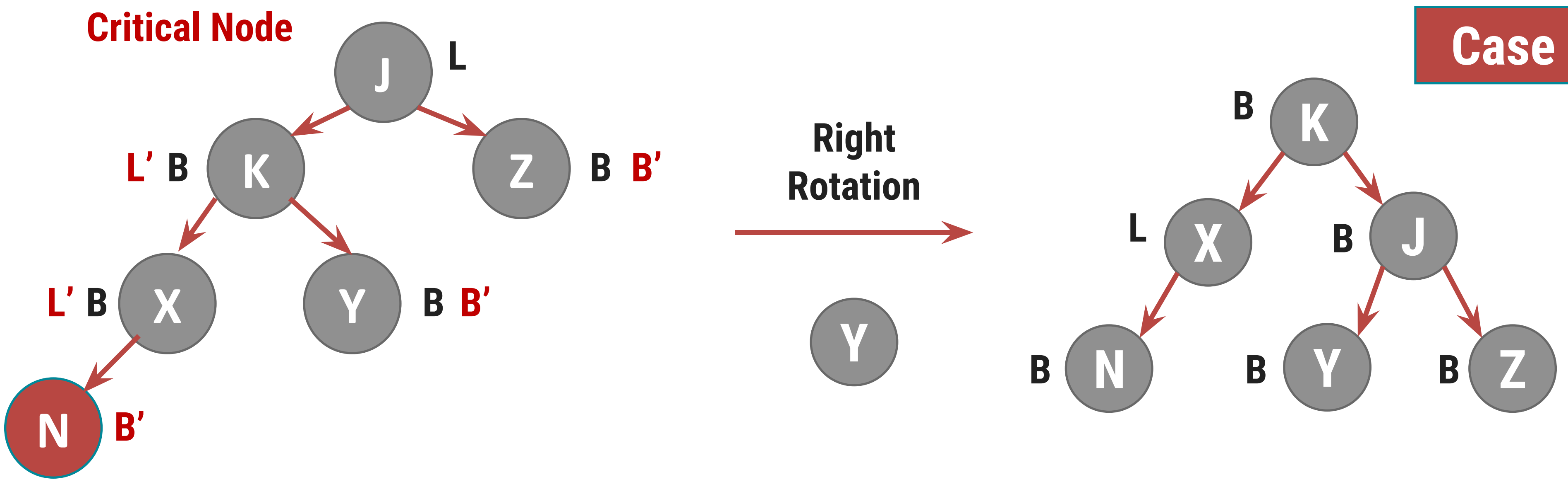
Case - 1

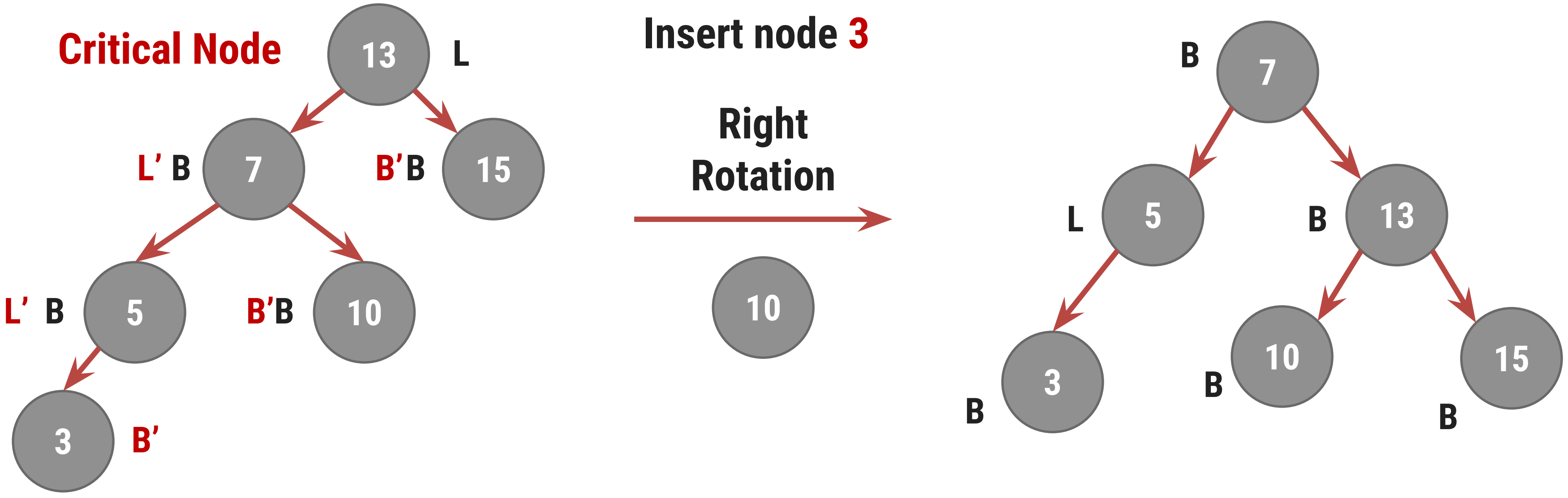


Single Right Rotation
of
unbalanced node

Insertion into Left sub-tree of nodes Left child : [L L -> R]

Case - 1





Insertion into Right sub-tree of node's Right child : [R R -> L]

- **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**

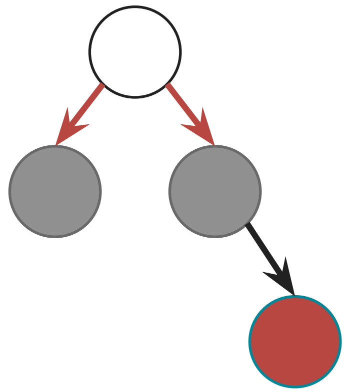
  A. Detach right child's leaf sub-tree

  B. Consider right child to be new parent

  C. Attach old parent onto left of new parent

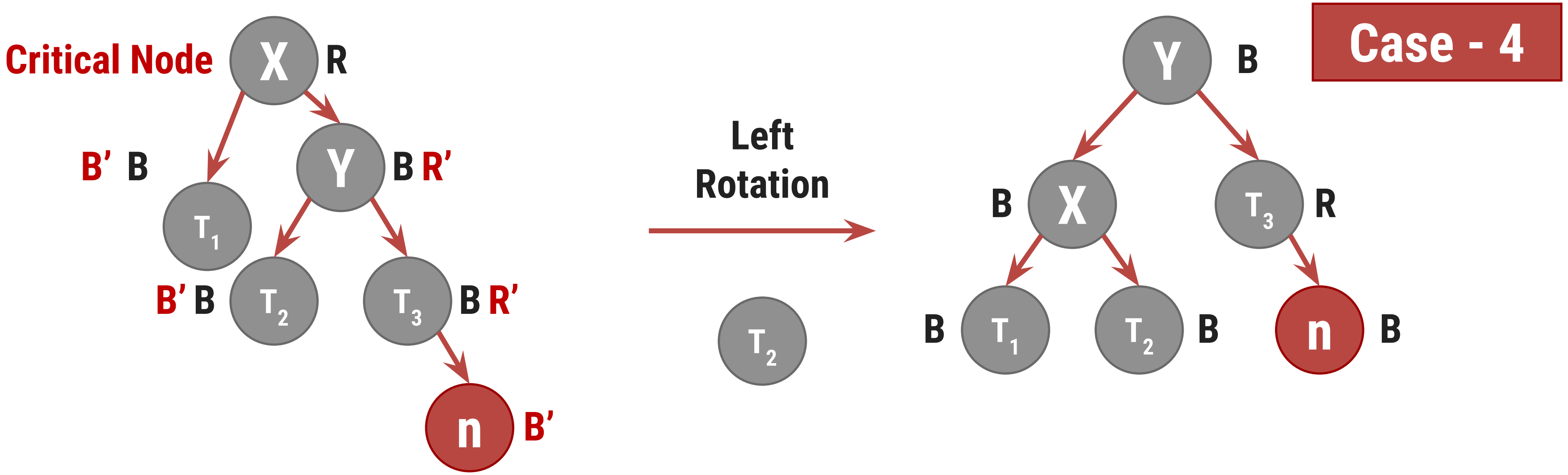
  D. Attach old right child's old left sub-tree as right sub-tree of new left child

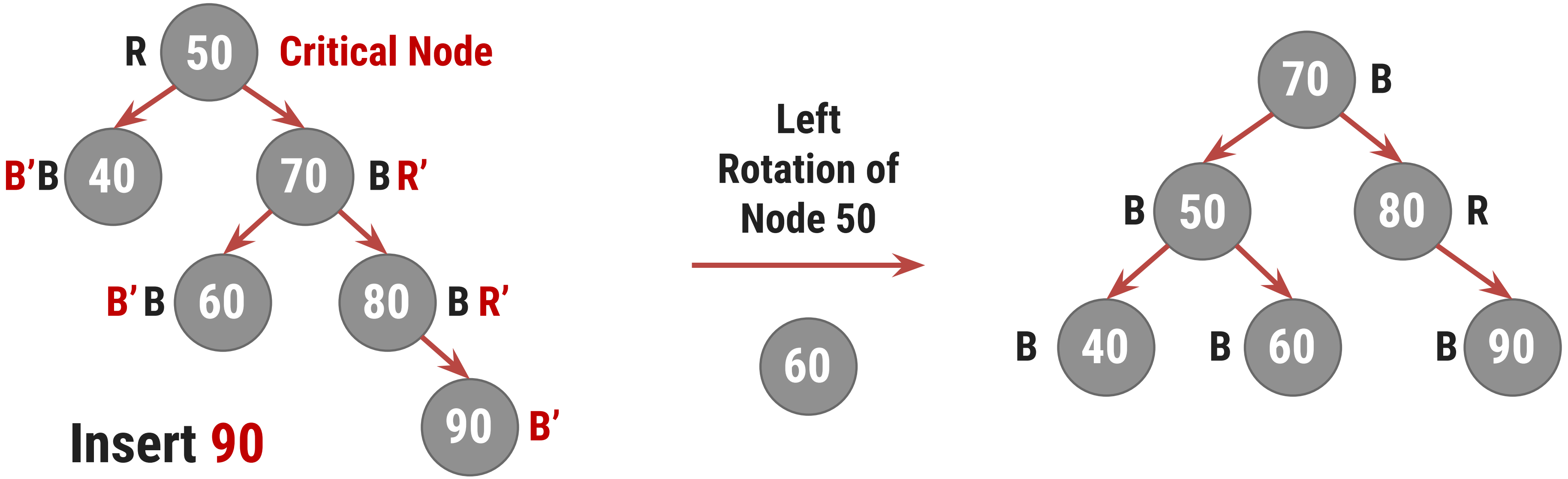
Case - 4



Single Left Rotation
of
unbalanced node

Insertion into Right sub-tree of node's Right child : [R R -> L]



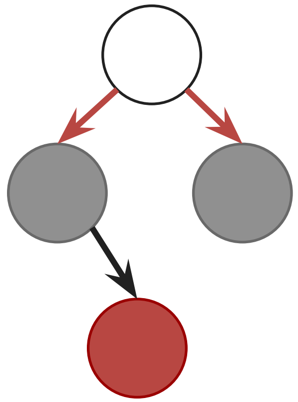


Insertion into Right sub-tree of node's Left child : [R L -> L R]

- **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**

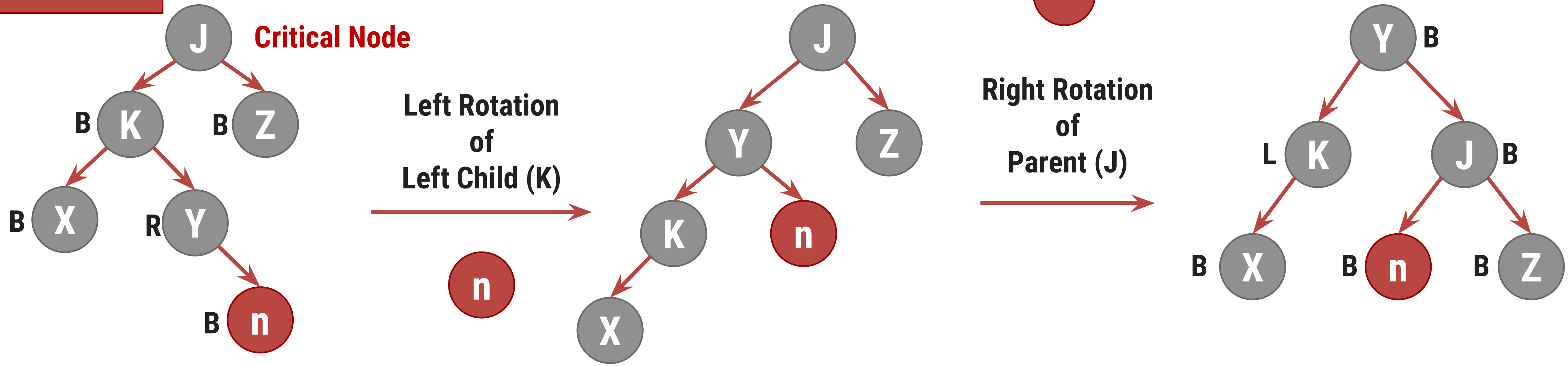
Case - 2



Left Right Rotation

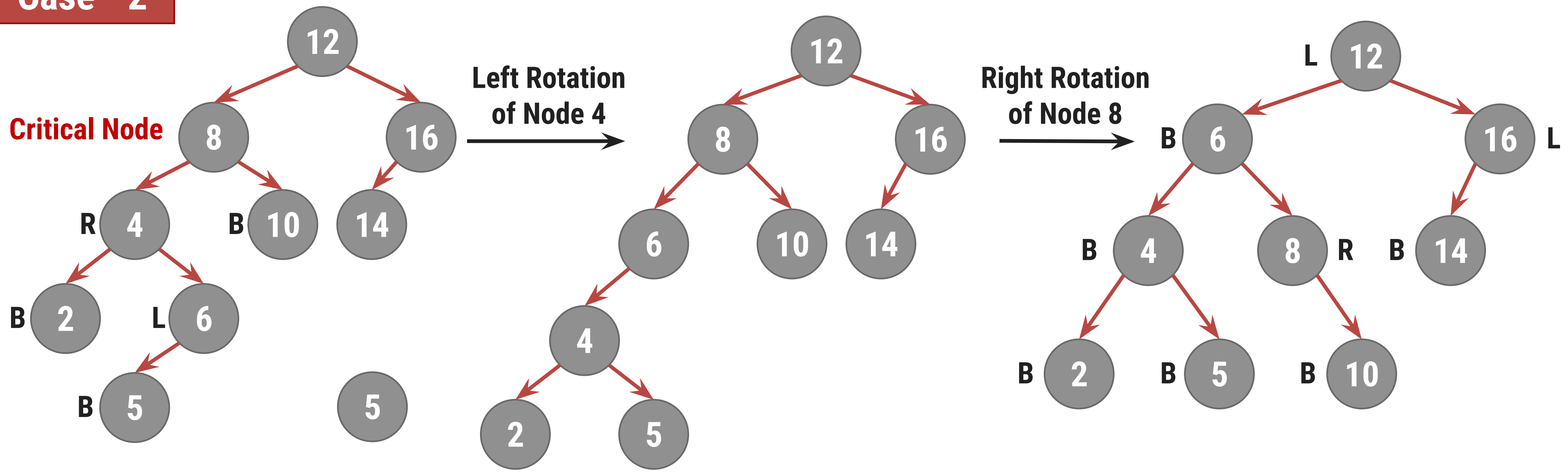
Left Rotation of Left Child
followed by
Right Rotation of Parent

Case - 2



Insertion into Right sub-tree of node's Left child : [R L -> L R]

Case - 2

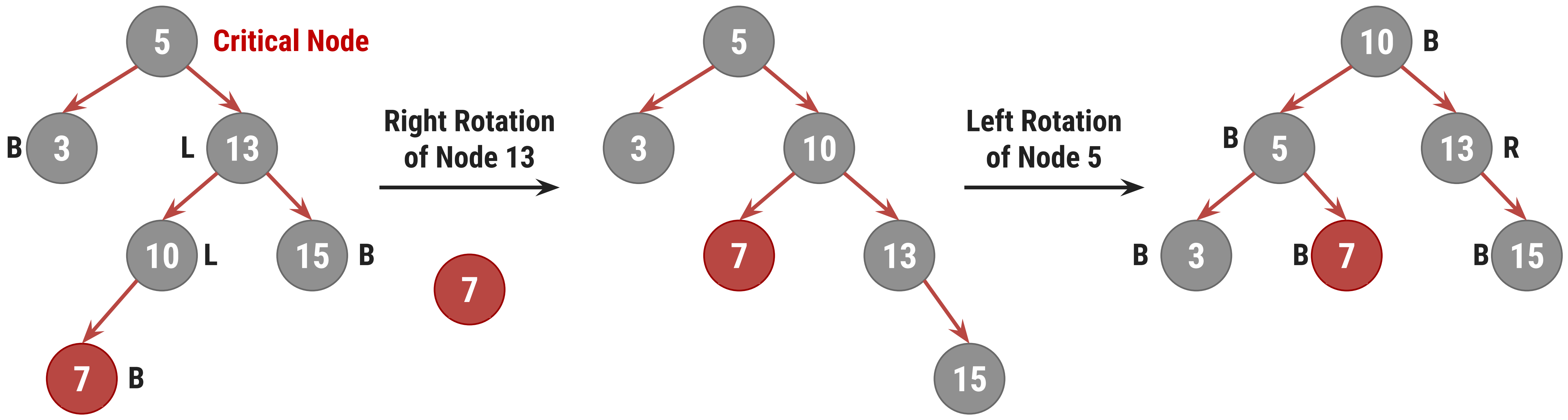


Left Right Rotation

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

Insertion into Left sub-tree of node's Right child : [L R -> R L]

Case - 3

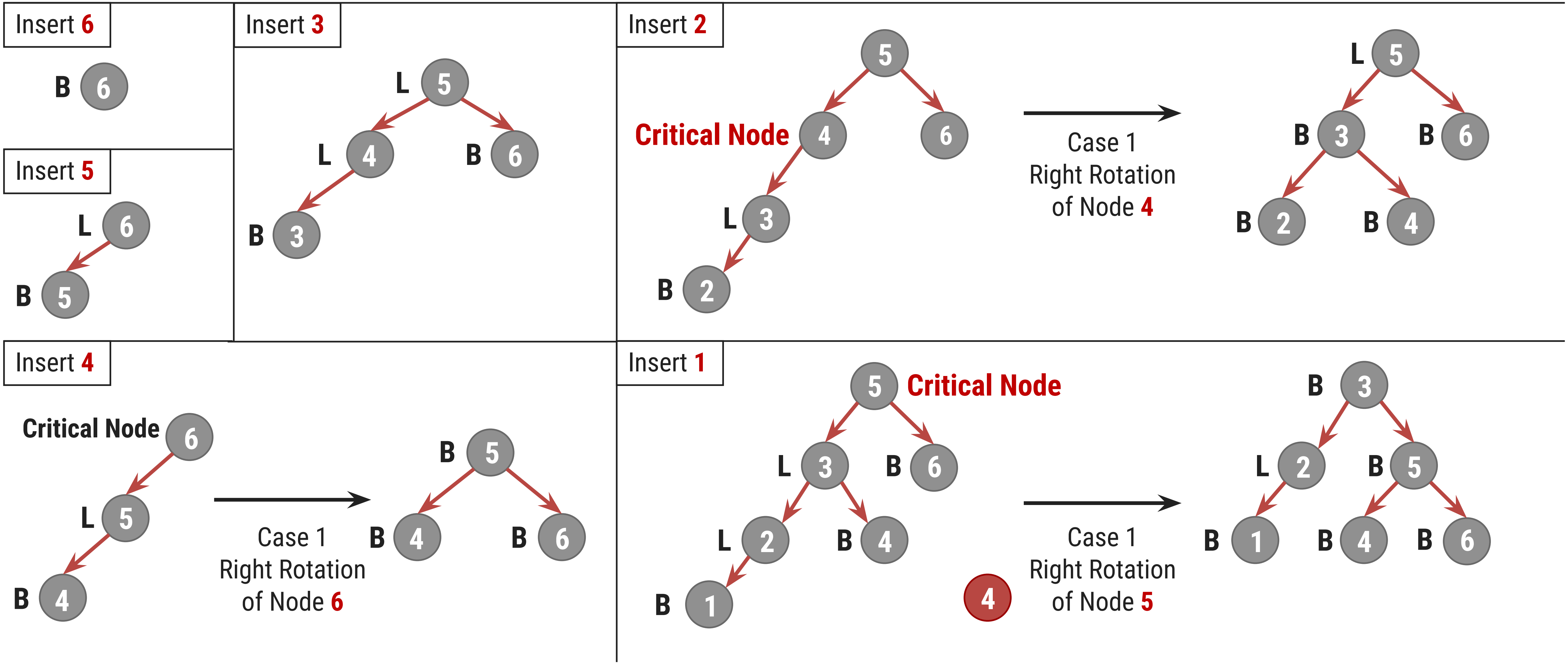


Right Left Rotation

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

Construct AVL Search Tree

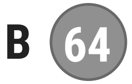
Construct AVL Search tree by inserting following elements in order of their occurrence **6, 5, 4, 3, 2, 1**



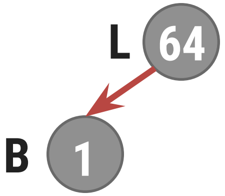
Construct AVL Search Tree

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

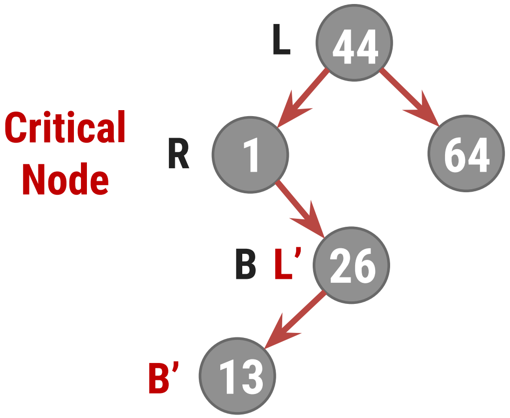
Insert **64**



Insert **1**

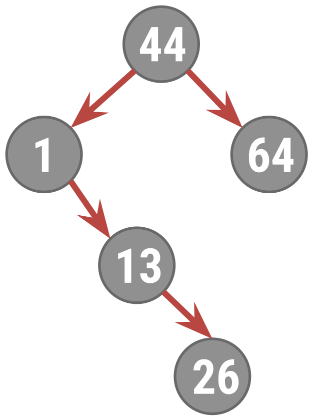


Insert **26 , 13**

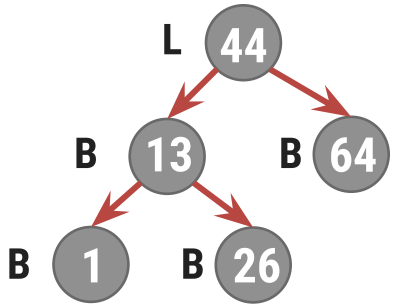


Case 3: Right Left Rotation
Right Rotation of Right Child **26**
Followed By
Left Rotation of Parent **1**

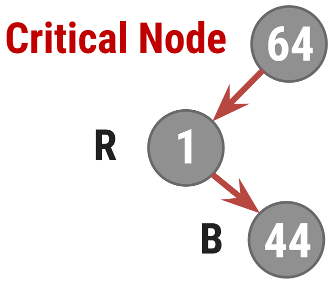
Right Rotation of Right Child **26**



Left Rotation of Parent **1**

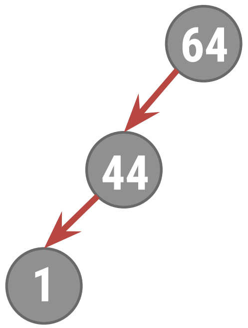


Insert **44**

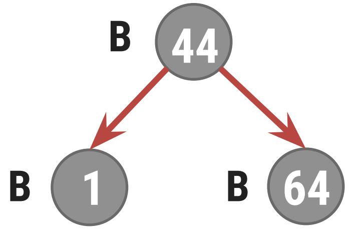


Case 2: Left Right Rotation
Left Rotation of Left Child **1**
Followed By
Right Rotation of Parent **64**

Left Rotation of Left Child **1**



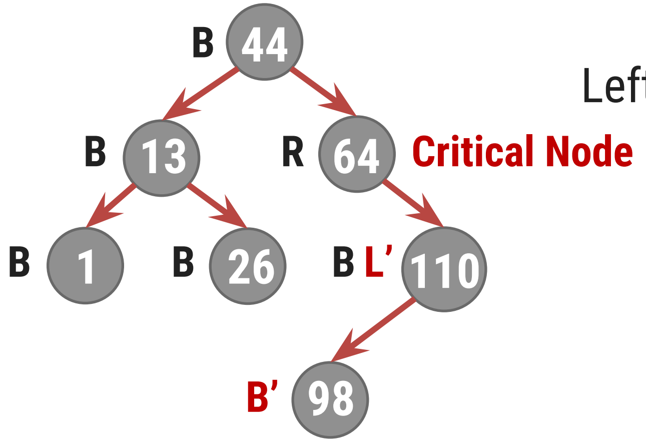
Right Rotation of Parent **64**



Construct AVL Search Tree

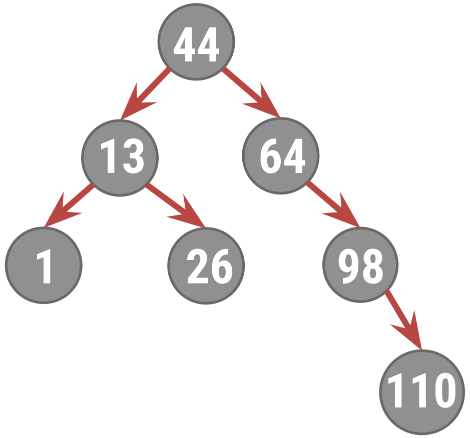
Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

Insert **110 , 98**

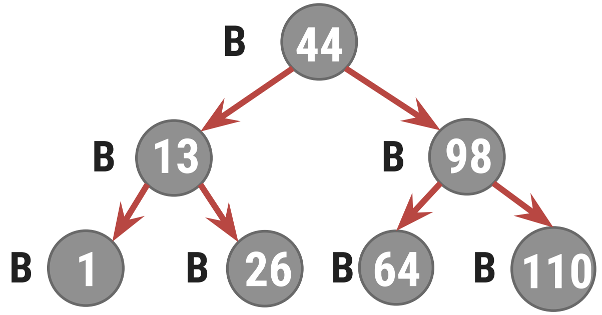


Case 3: Right Left Rotation
Right Rotation of Right Child 110
Followed By
Left Rotation of Parent 64

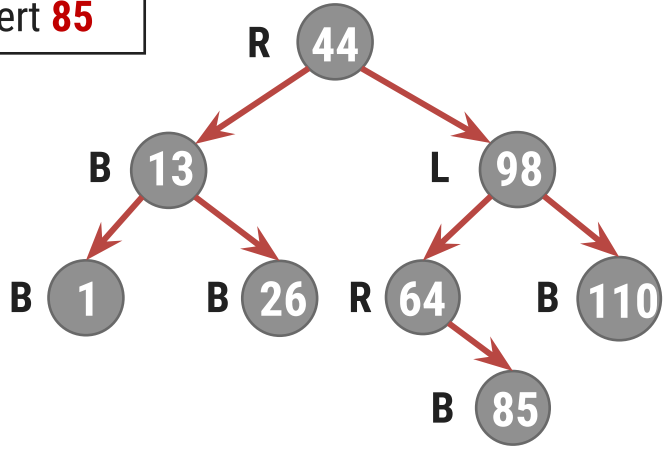
Right Rotation of Right Child **110**



Left Rotation of Parent **64**

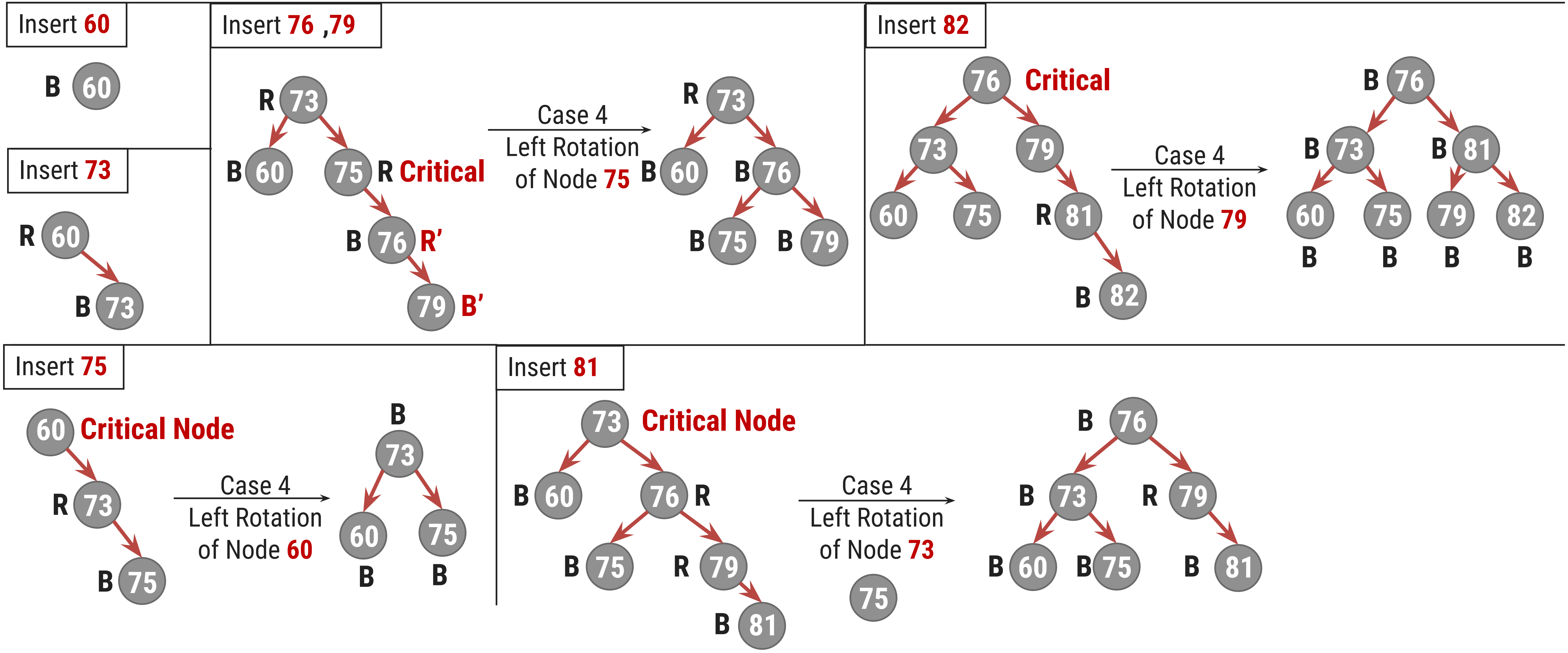


Insert **85**



Construct AVL Search Tree

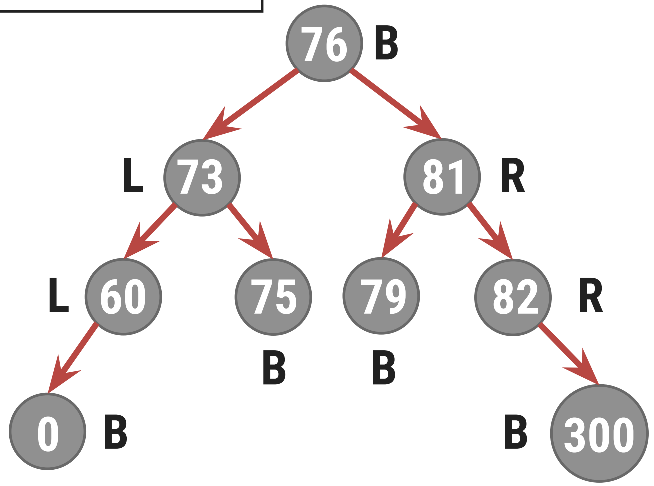
Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,82,300,0,5,73**



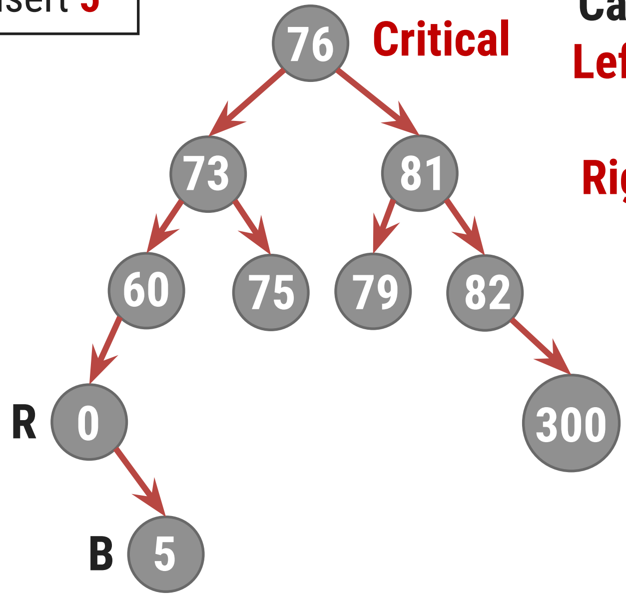
Construct AVL Search Tree

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

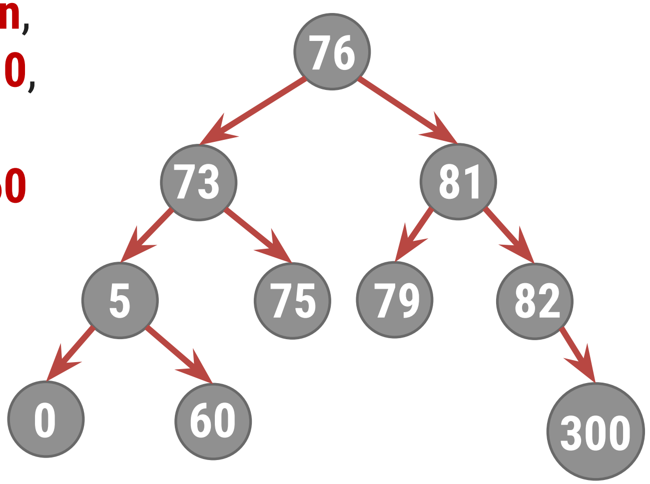
Insert **300 , 0**



Insert **5**



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

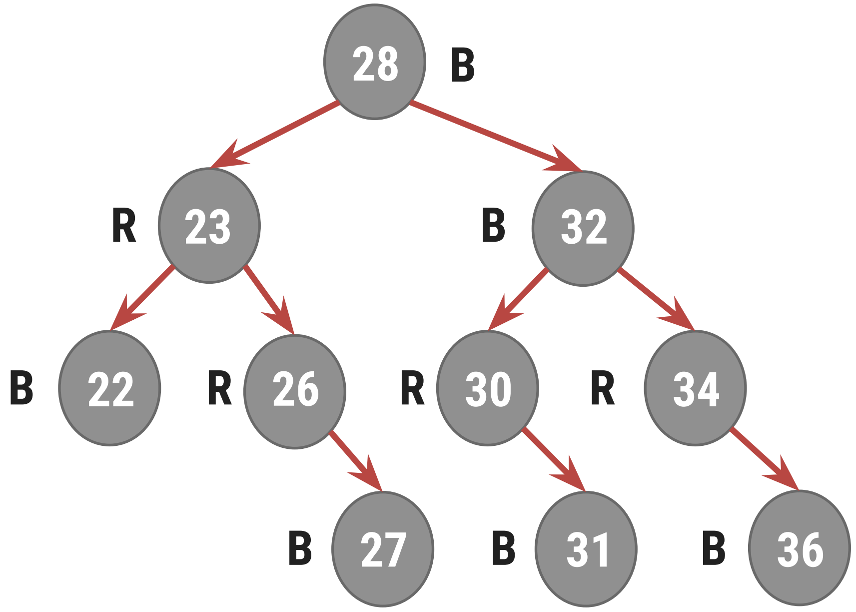


Insert **73**

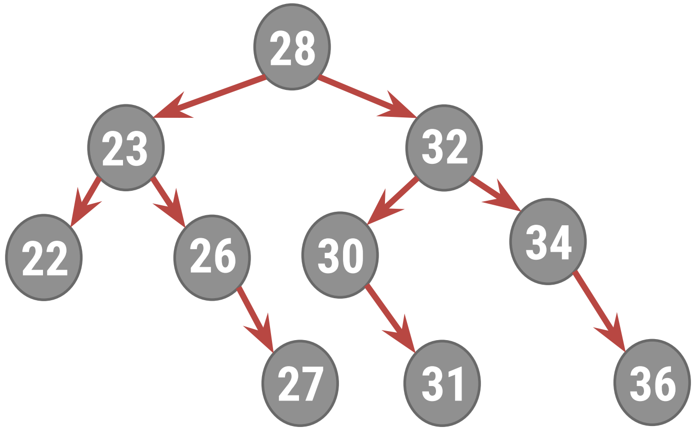
Can not Insert **73** as duplicate key found

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



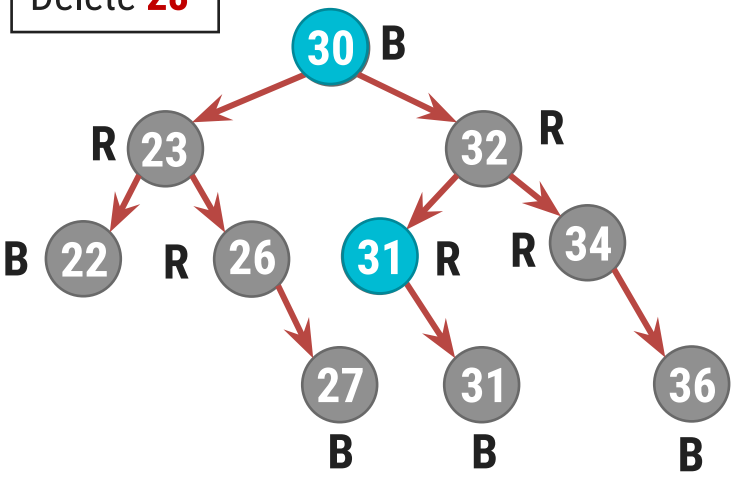
Deleting node from AVL Tree



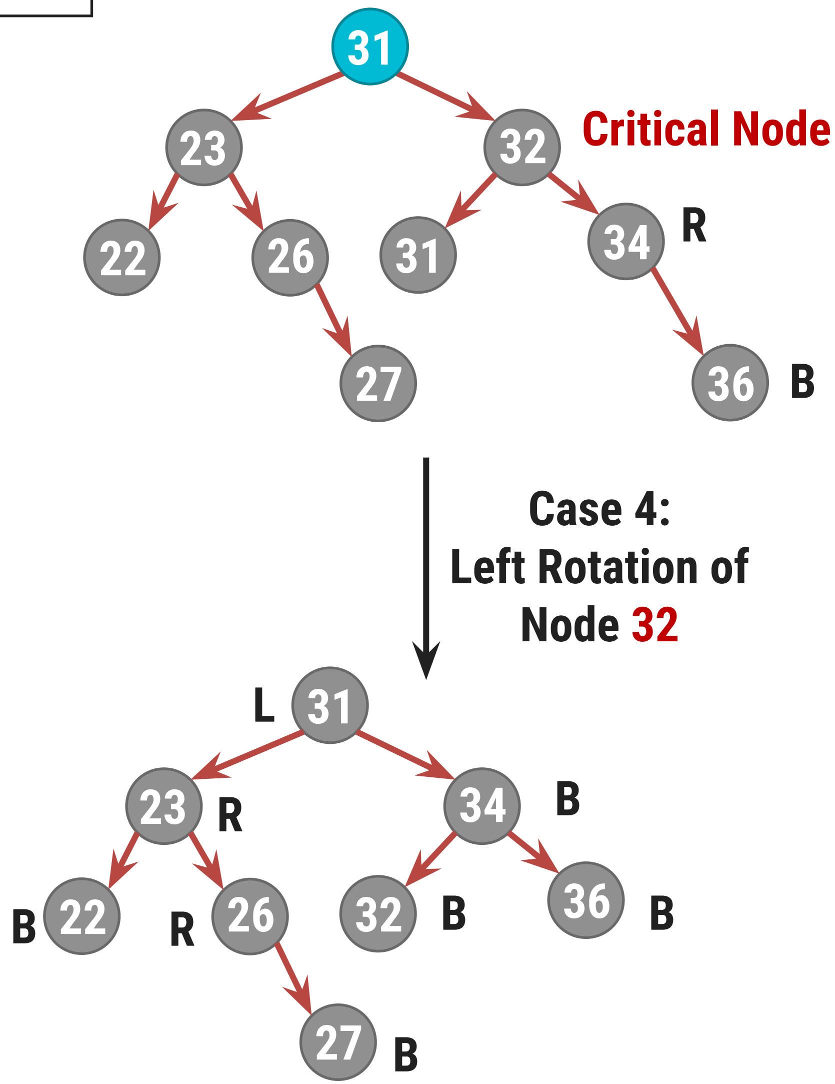
In-Order Traversal

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

Delete **28**

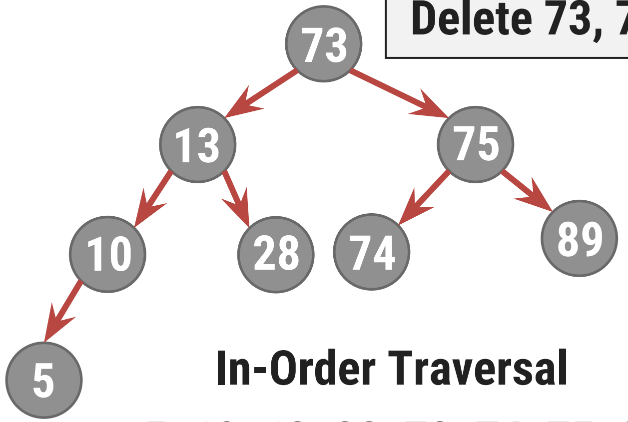


Delete **30**



Deleting node from AVL Tree

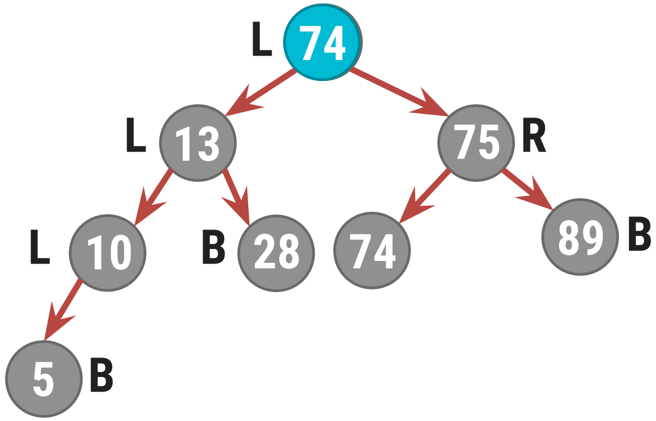
Delete 73, 74



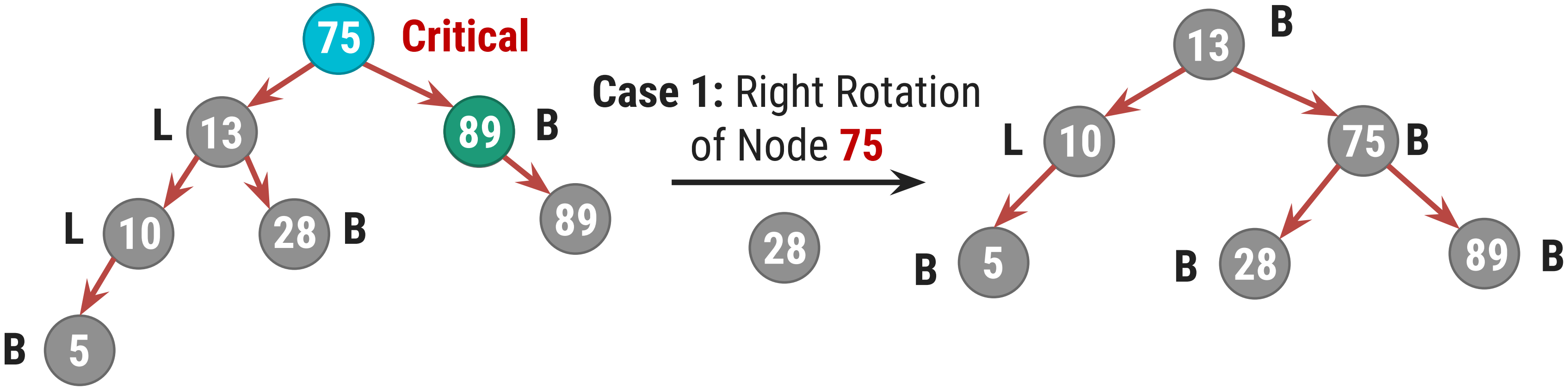
In-Order Traversal

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

Delete **73**



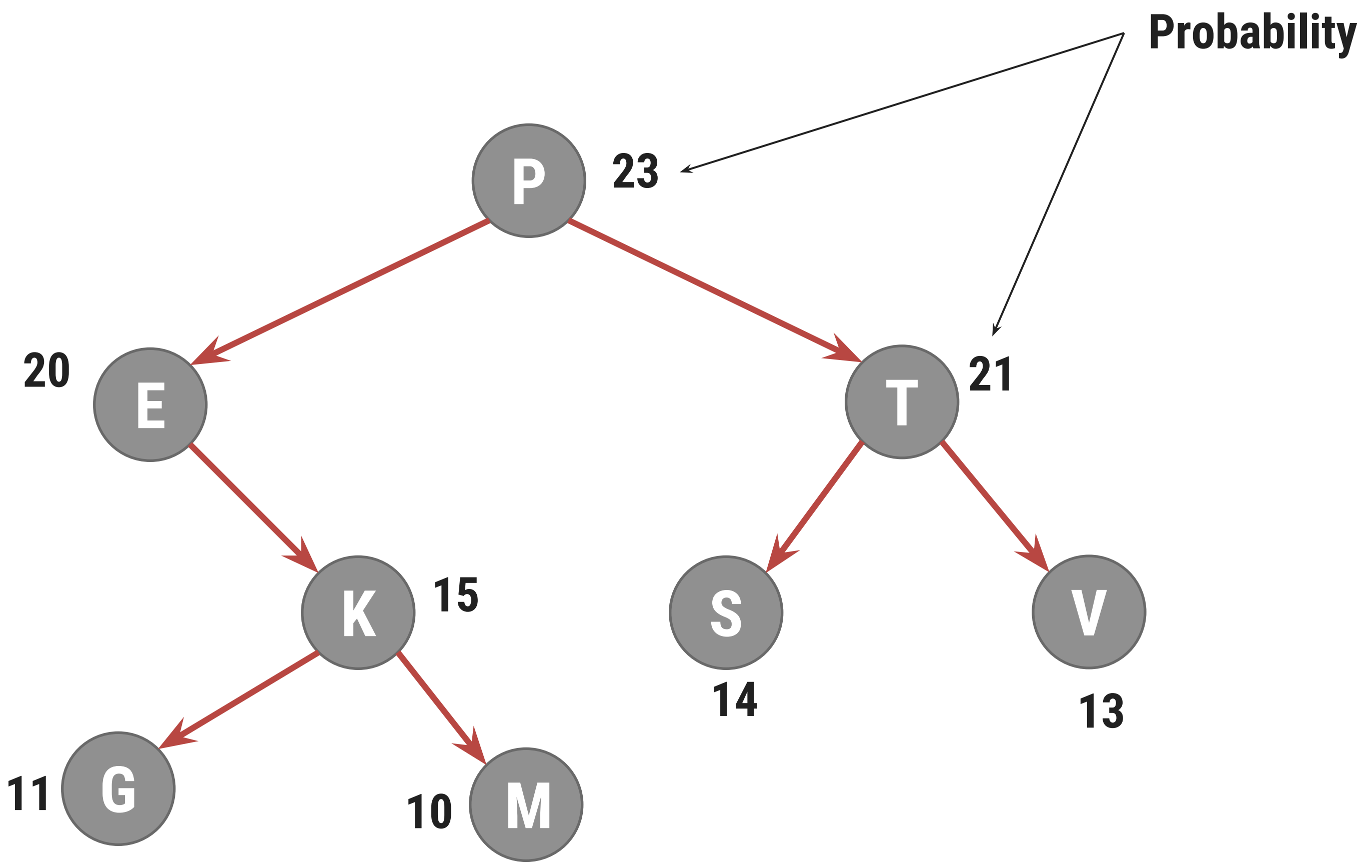
Delete **74**



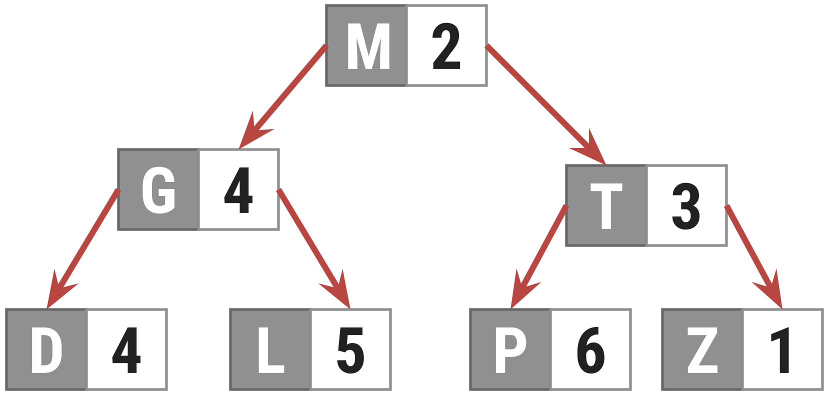
Weight Balanced Tree

- 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

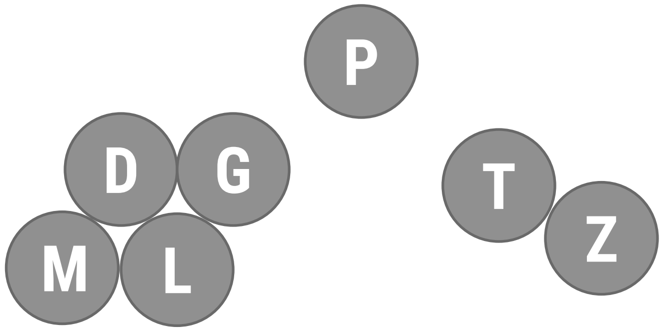
Weight Balanced Tree

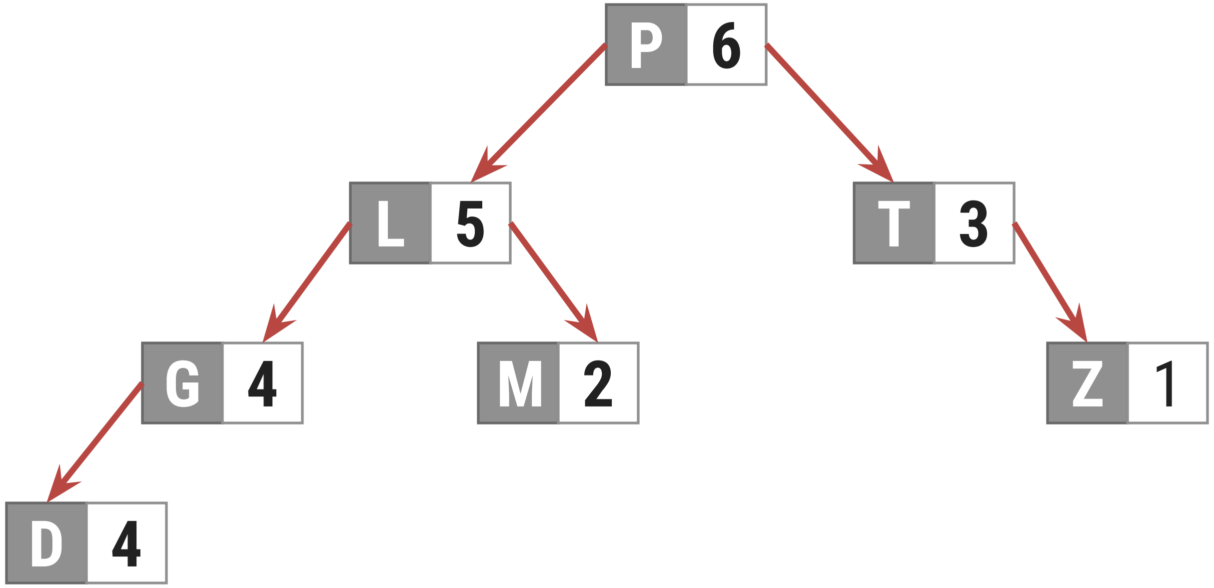


Weight Balanced Tree



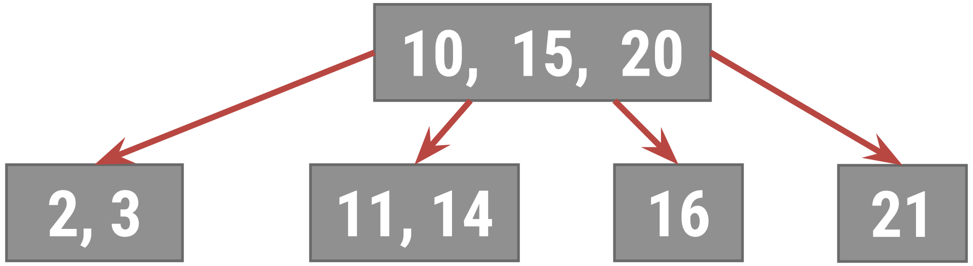
Ordered Tree





Multiway Search Tree (B - Tree)

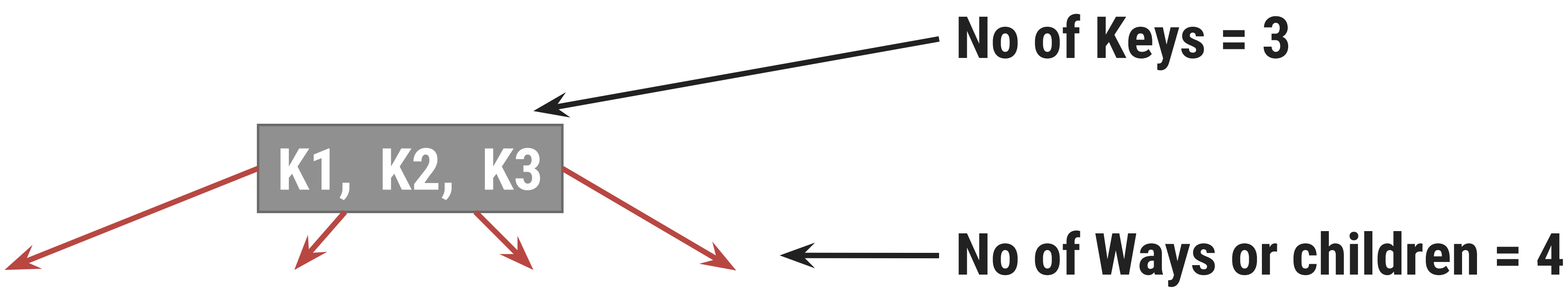
- The **nodes** in a **binary tree** like AVL tree **contains only one record**
- **AVL tree is** commonly **stored in primary memory**
- In **database applications** where **huge volume** of **data** is handled, the **search tree** can **not be accommodated** in **primary memory**
- **B-Trees** are 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)

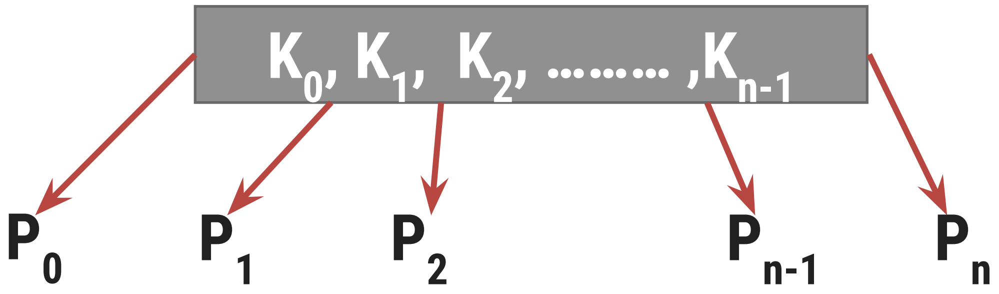
- An **M- way** tree contains **multiple keys** in a node
- This leads to **reduction** in overall **height** of the tree
- If a **node** of M-way tree **holds K keys** then it will have **k+1 children**



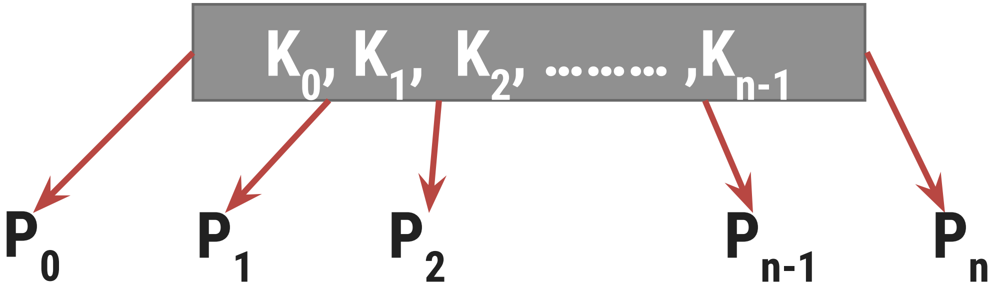
Multiway Search Tree (B - Tree)

- 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**

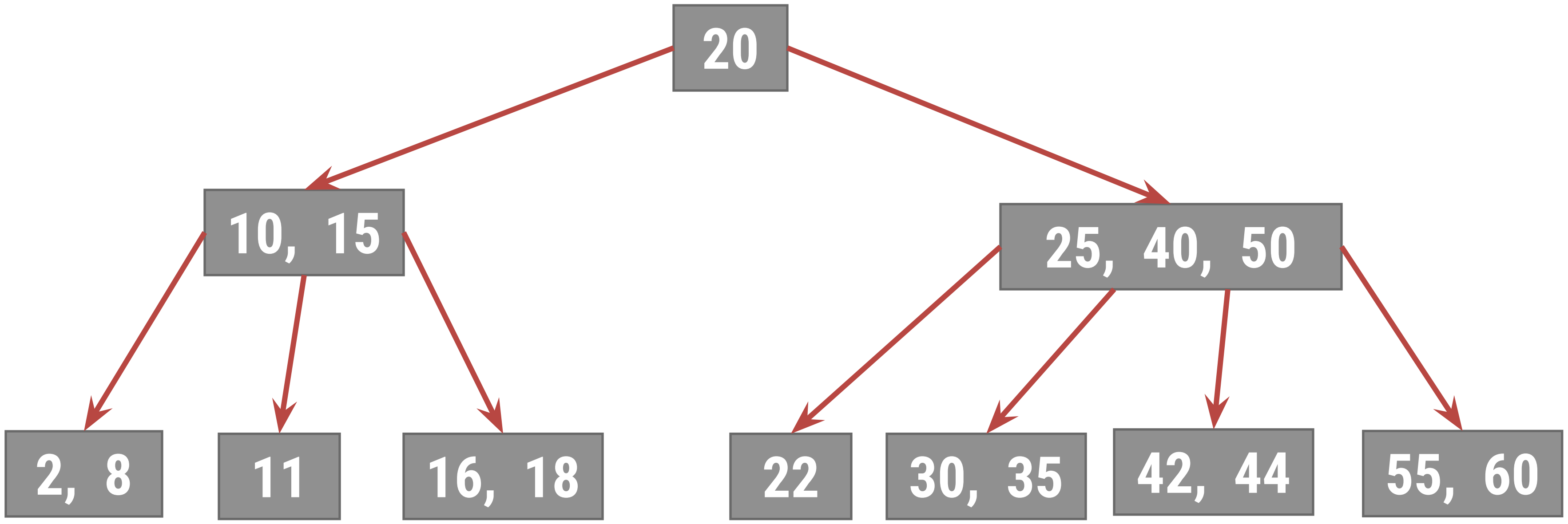


Multiway Search Tree (B - Tree)



- $K_0, K_1, K_2, \ldots\ldots\ldots, K_{n-1}$ 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)

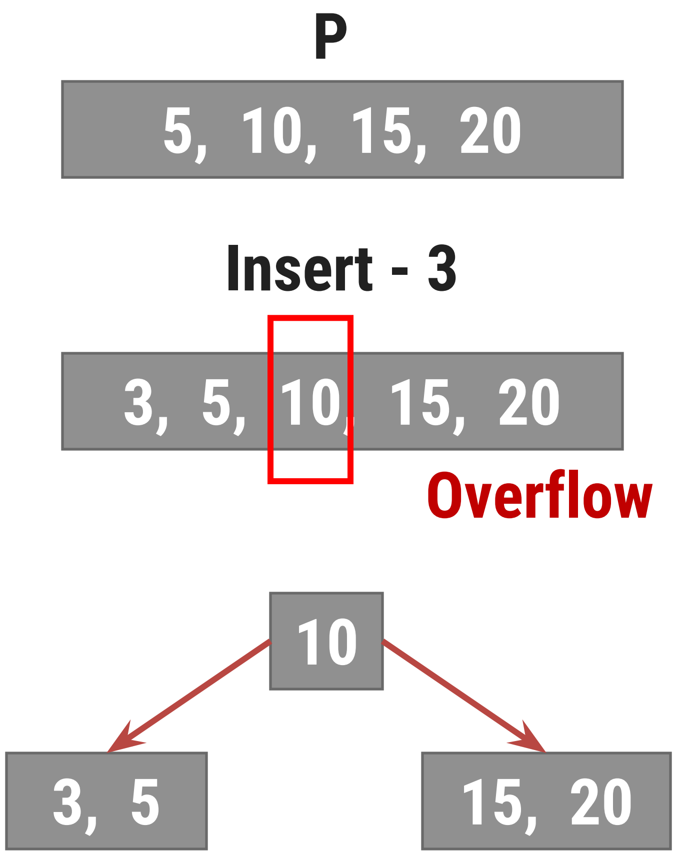


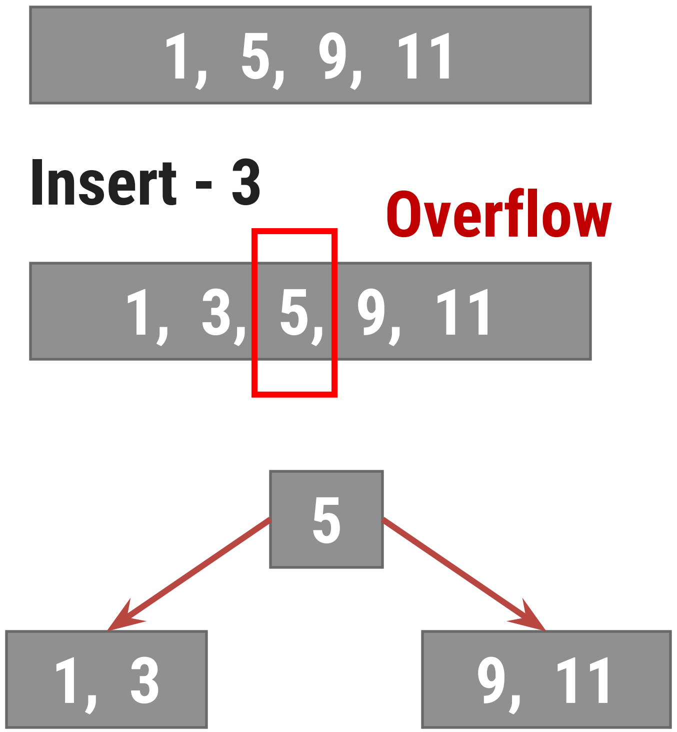
B-Tree of Order 4 (4 way Tree)

Insertion of Key in B-Tree

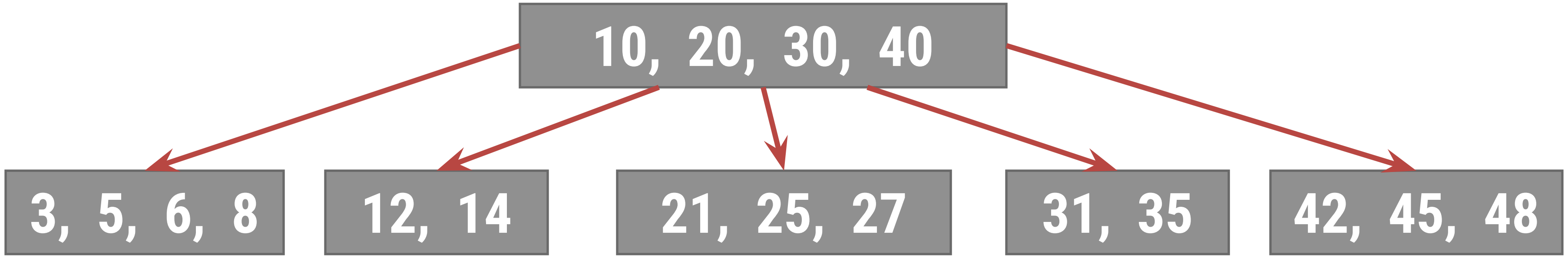
1. If **Root** is **NULL**, **construct** a node and **insert key**
2. If **Root** is **NOT NULL**
   I. Find the **correct leaf** node to which key should be added
   II. If **leaf node has space** to accommodate key, it is **inserted** and **sorted**
   III. If **leaf node does not have space** to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

Split Node (5 way Tree, max 4 Keys)

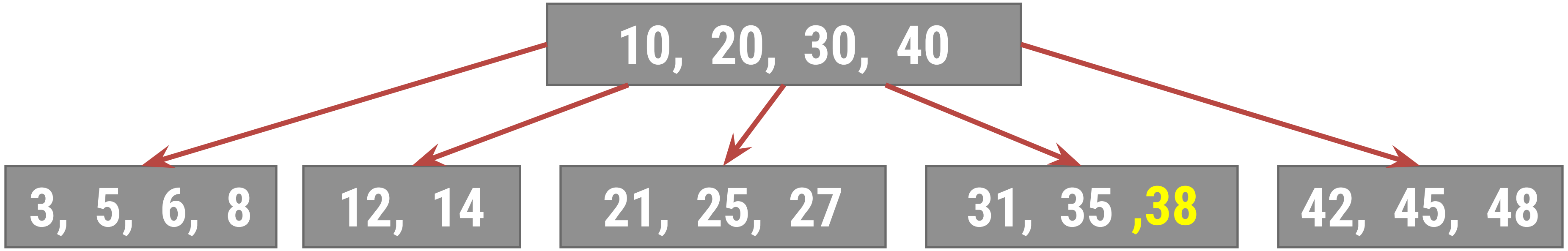




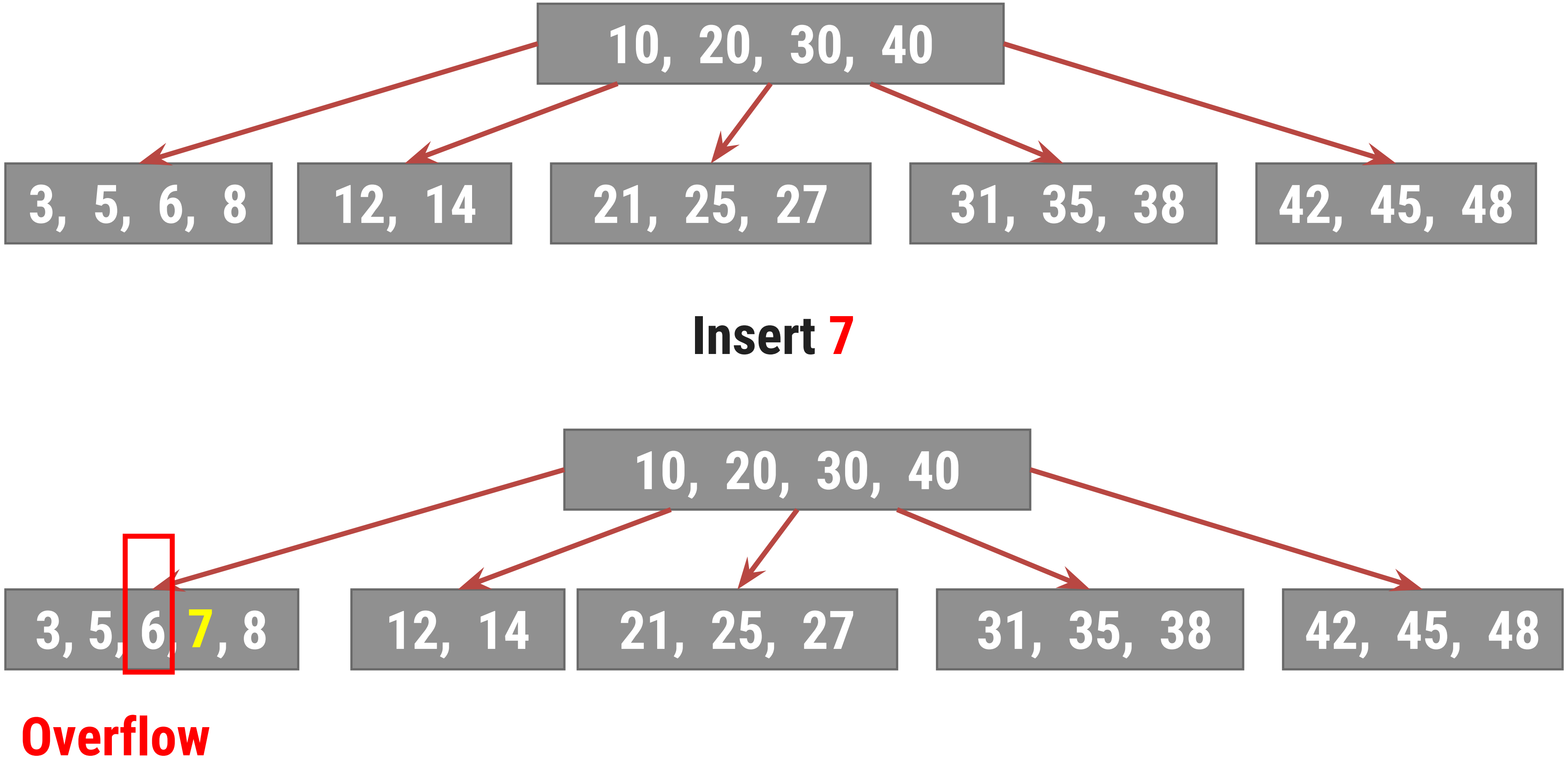
Split Node (5 way Tree, max 4 Keys)



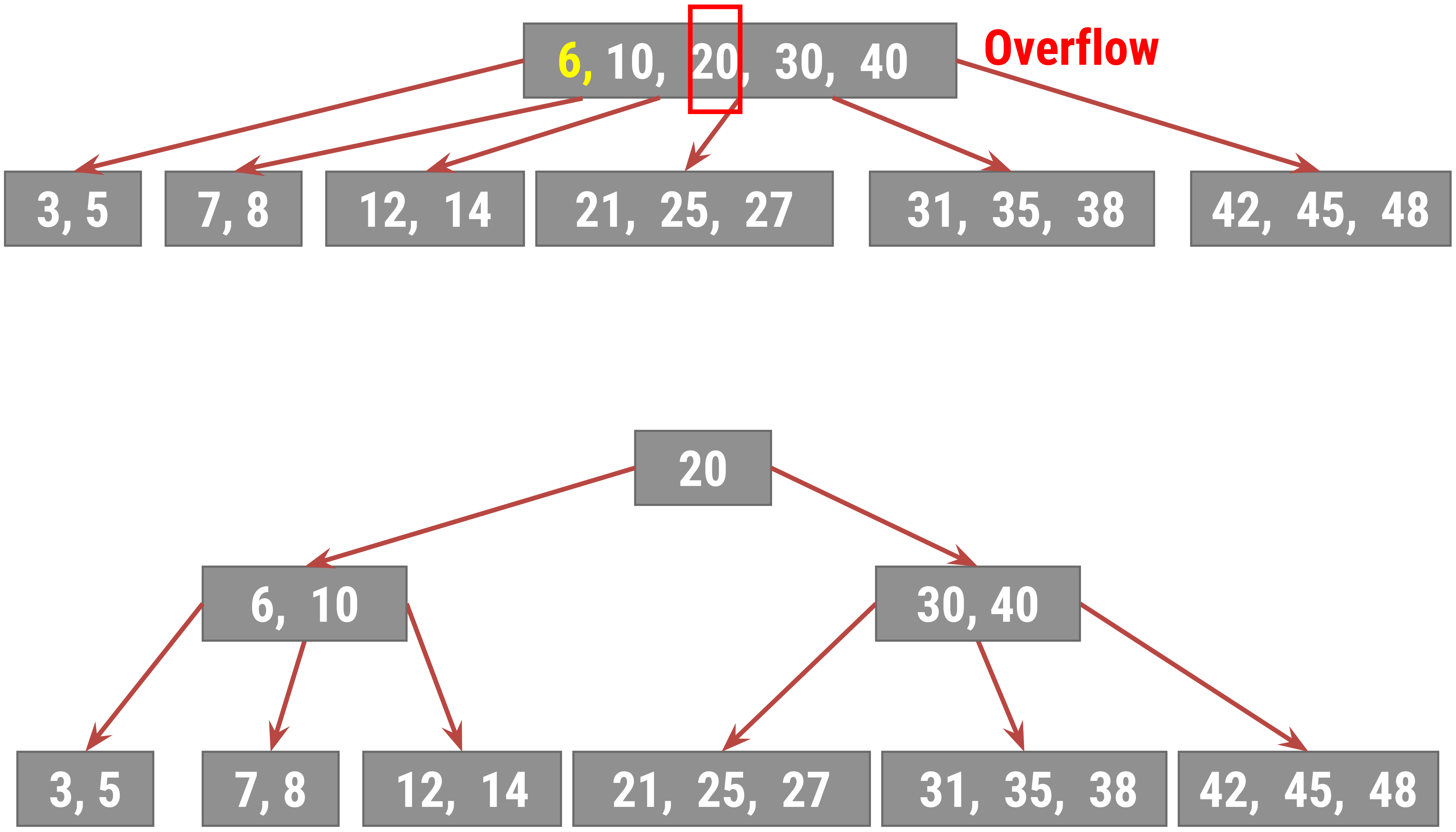
Insert - 38



Split Node (5 way Tree, max 4 Keys)



Split Node (5 way Tree, max 4 Keys)

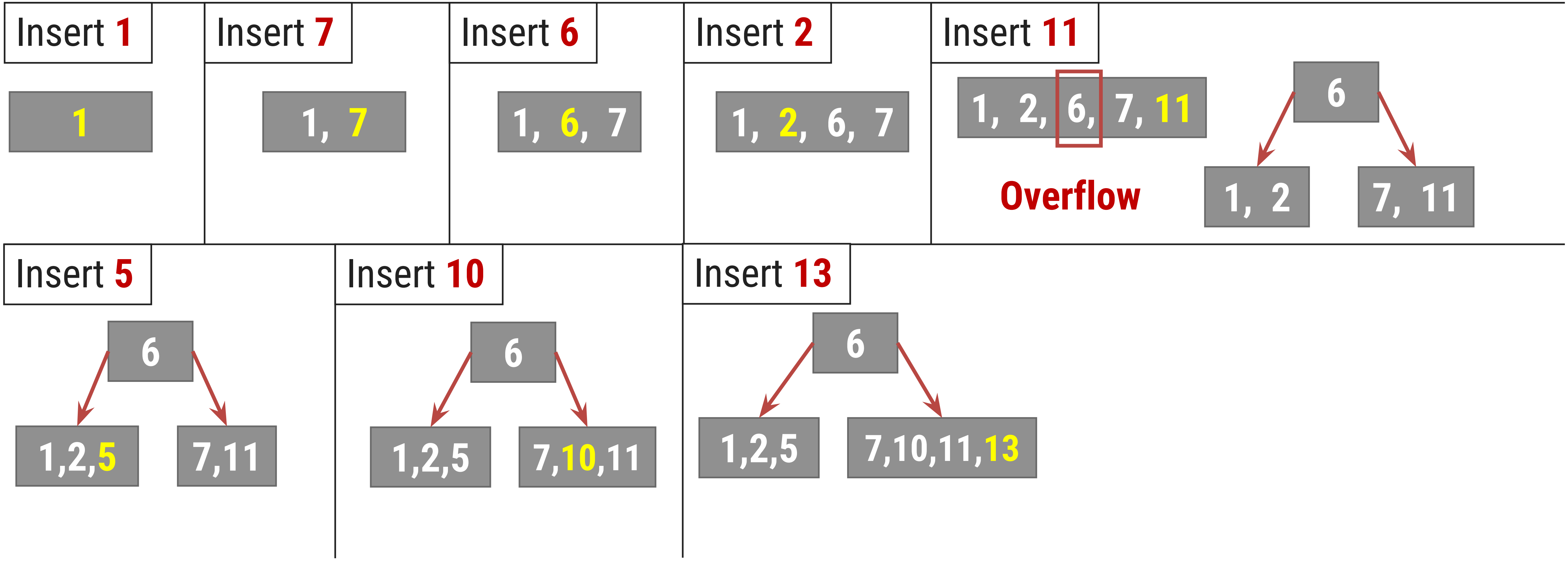


Construct M-Way Tree

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

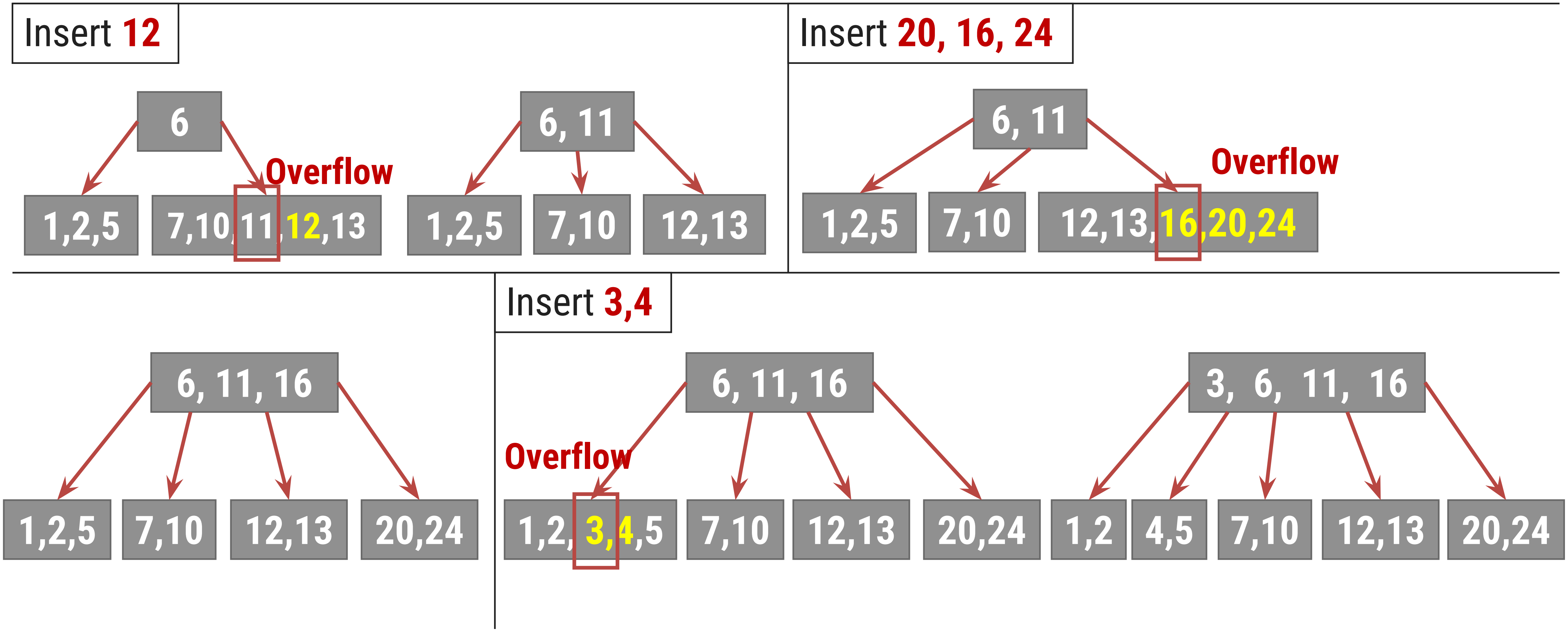
We are asked to **create 5 Order Tree (5 Way Tree) maximum 4 records** can be accommodated in a node



Construct M-Way Tree

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

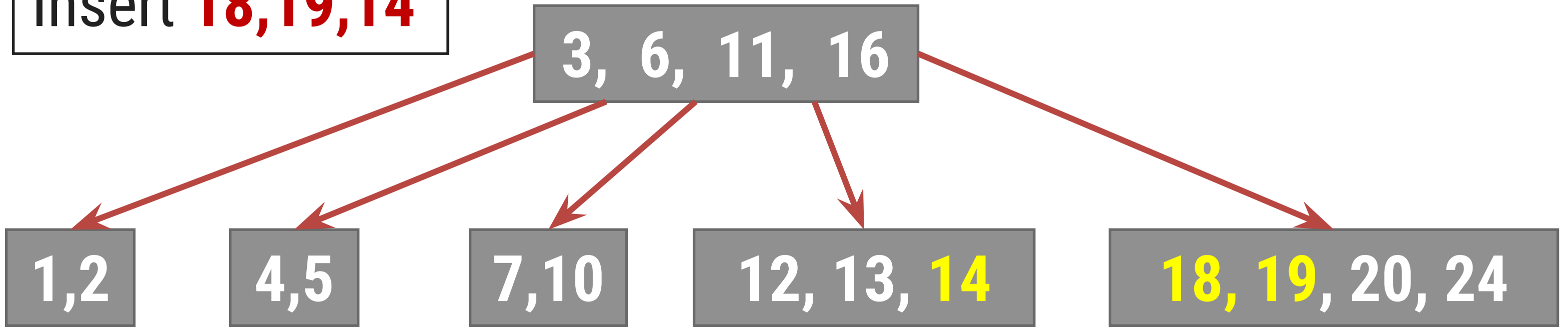
1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25



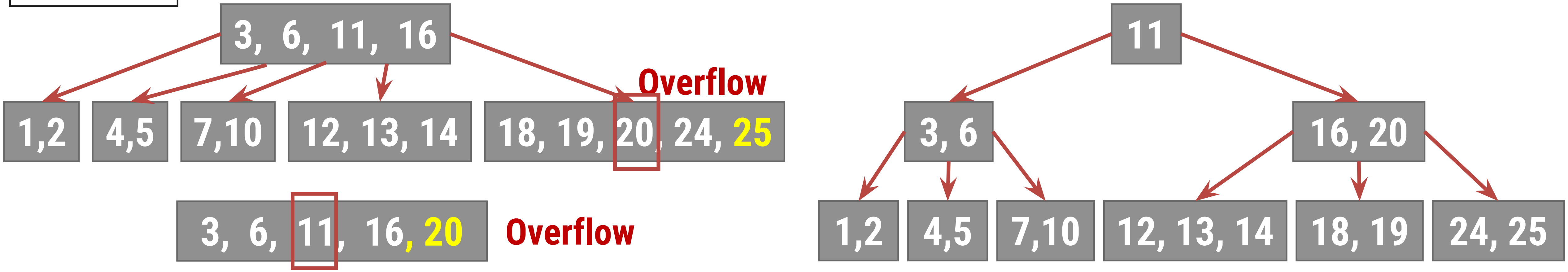
Construct M-Way Tree

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**



Thank You