UNIT - III

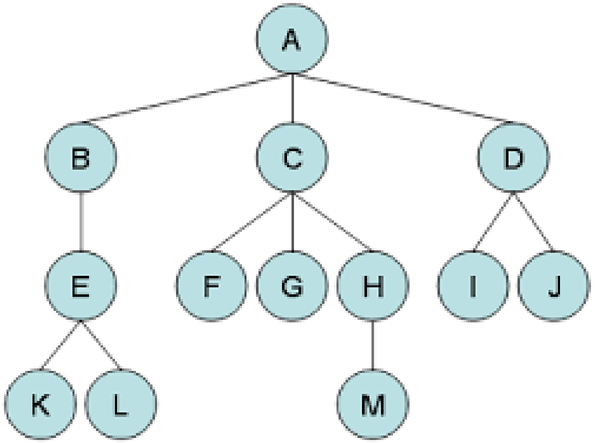
Trees – Terminology, Representation of Trees, Binary tree ADT, Properties of Binary Trees, Binary Tree Representations-array and linked representations, Binary Tree traversals, Threaded binary trees, Binary Heap-Properties,Max and Min Heap, Operations-Insertion and Deletion.

Search Trees-Binary Search tree, Tree traversals, AVL tree – operations, B-tree – operations, B+ trees, Red Black tree.

Tree:

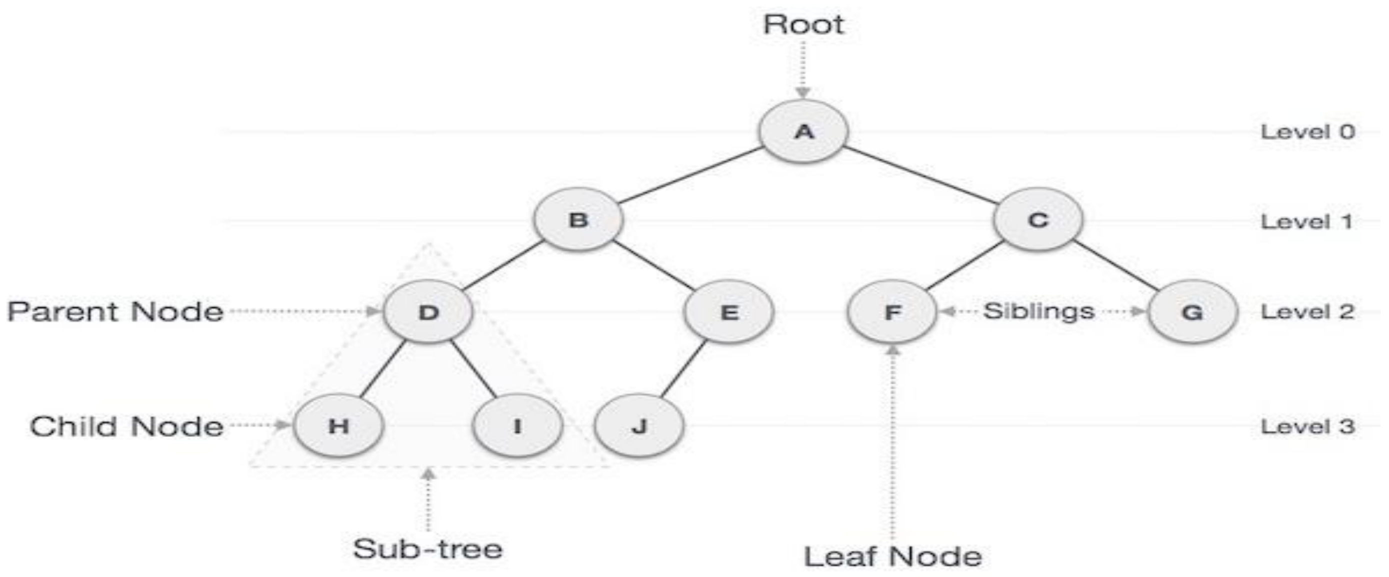
- A tree is **a non linear hierarchical** data structure,consists of **nodes** connected by **edges**.

Why Tree Data Structure?



- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

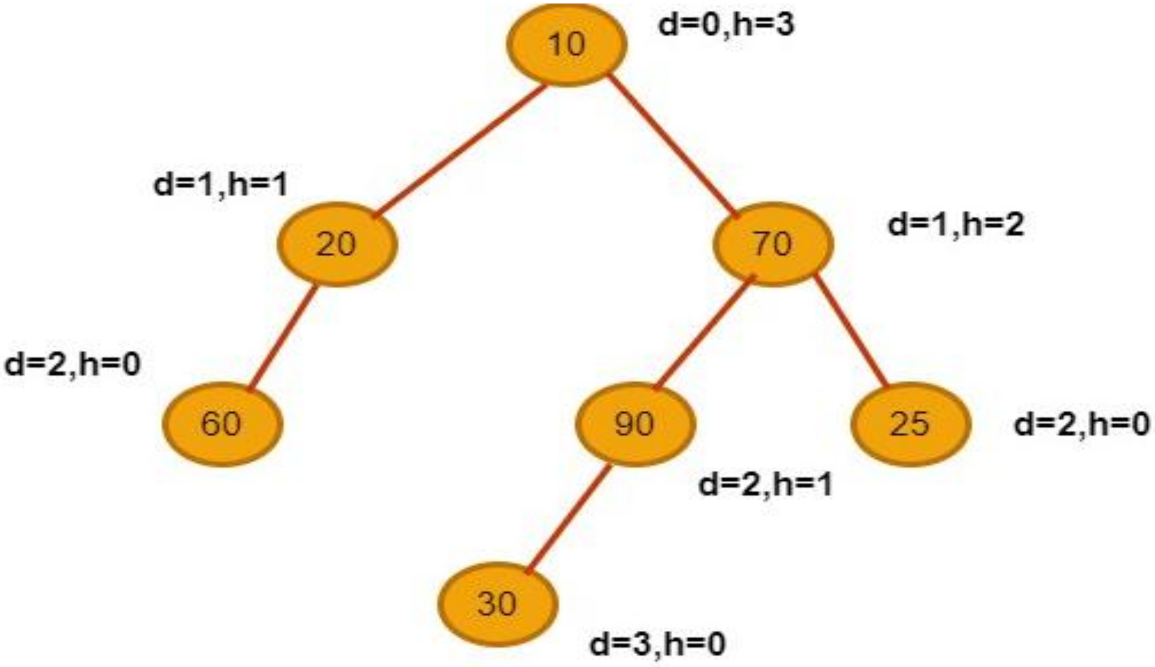
Tree Terminology



- **Node-**A node is an entity that contains a key or value and pointers/link to its child nodes.
- **Edge-**It is the link between any two nodes.
- **Root -**The node at the top/start of the tree is called root. There is only one root per tree.
- **Parent -** Any node except the root node has one edge upward to a node called parent.
- **Child -** The node below a given node connected by its edge downward is called its child node.
- **Leaf -** The node which does not have any child node is called the leaf node.
- **Subtree -**Subtree represents the descendants of a node.
- **Levels -**Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Siblings:** if two are more nodes are having same parent in a tree, then they are called siblings.

Tree Terminology...

- . **Height of a Node-**The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).
- **Depth of a Node-**The depth of a node is the number of edges from the root to the node.
- **Height of a Tree-**The height of a Tree is the height of the root node or the depth of the deepest node or maximum of all nodes height or maximum of all nodes depth.
- In a tree, height of a tree is equal to depth of a tree.



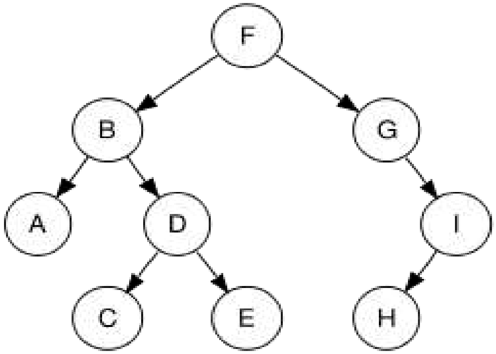
Types of Trees

- Different types of tree are:

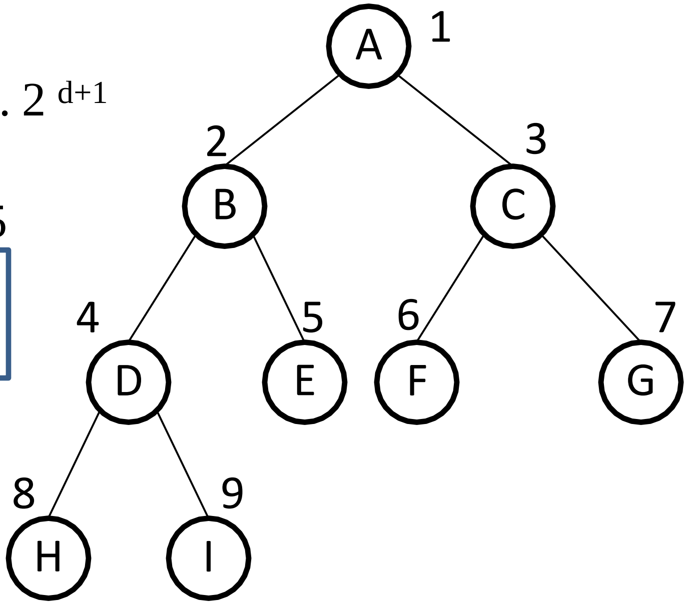
1. **Binary Tree**
   1. Strict Binary Tree
   2. Complete Binary Tree
   3. Perfect Binary Tree
   4. Extended Binary Tree
2. **Expression Tree**
3. **Threaded Binary Tree**
4. **Binary Heap**
5. **Search Trees**
   1. Binary Search Tree(BST)
   2. AVL Tree
   3. Red Black Tree
   4. B-Tree
   5. B+-Tree

Binary Tree

- A **Binary tree** is a tree which is either empty or each node can have maximum of two child nodes, i.e. each node can have 0, or 1 or 2 child nodes.



- **Binary Tree Representations:**
  1. **Array Representation**
  2. **Linked List Representation**
- **Array Representation of Binary Tree**
  - Binary Tree is stored in a single array.
  - Number are assigned to each node from leftmost to rightmost node.
  - Root node always assign no. 1
  - Left Child is placed at position [2 * K] (k is position of root/parent)
  - Right Child is placed at position [2 * K + 1]
  - Size of array is Depends on Depth of tree i.e. 2^{d+1}



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I						

Note: If root is placed at 0th position then

L Child = [2 * k + 1]

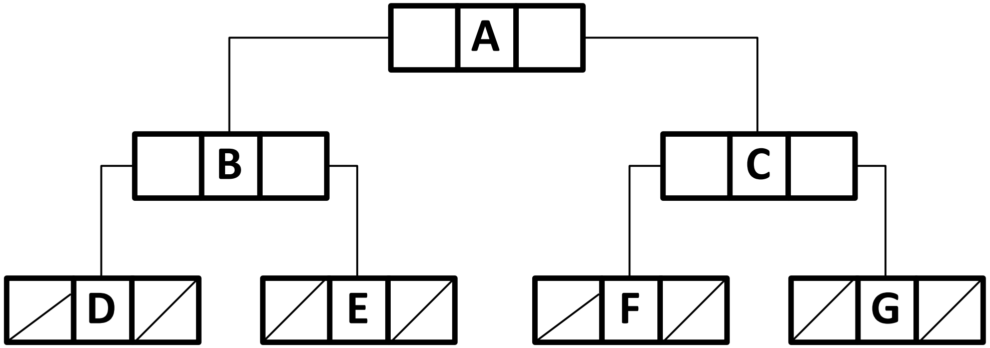
R child = [2 * K + 2]

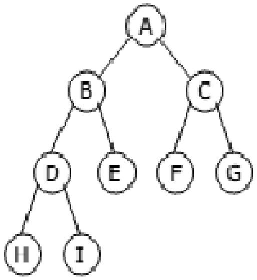
Binary Tree Representations...

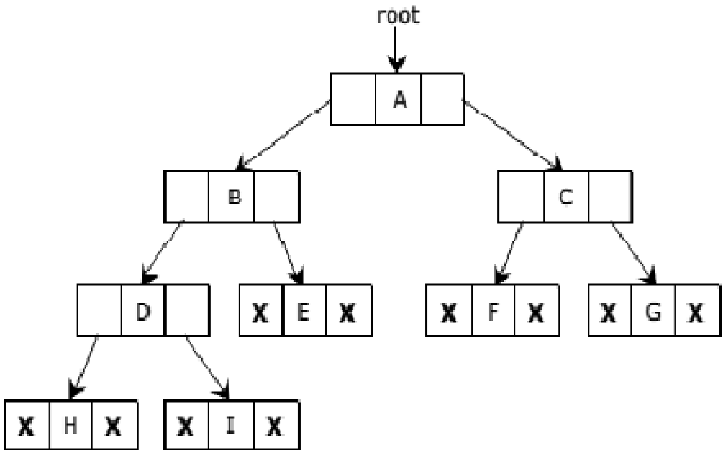
- **Linked List Representation of Binary Tree:**
  - Each node contains three fields –Left child address , Data and Right child address.
  - If the node does not contain left /right child hen address field contains NULL.
  - **Node of a binary tree is declared as:**

```
struct node
{
  struct node*left;
  int data;
  struct node *right;
};
```

E.g.



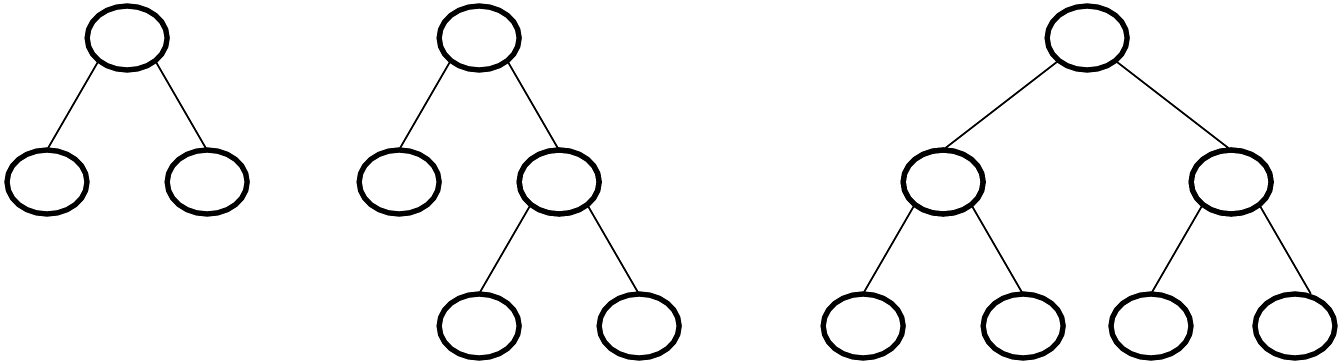




Types of Binary Tree

1. Strict Binary Tree:

- ➢ Each node in a tree is either leaf or has exactly two children.



Strict Binary Tree

- ➢ Strictly binary tree with n Leaves always contains 2n-1 nodes

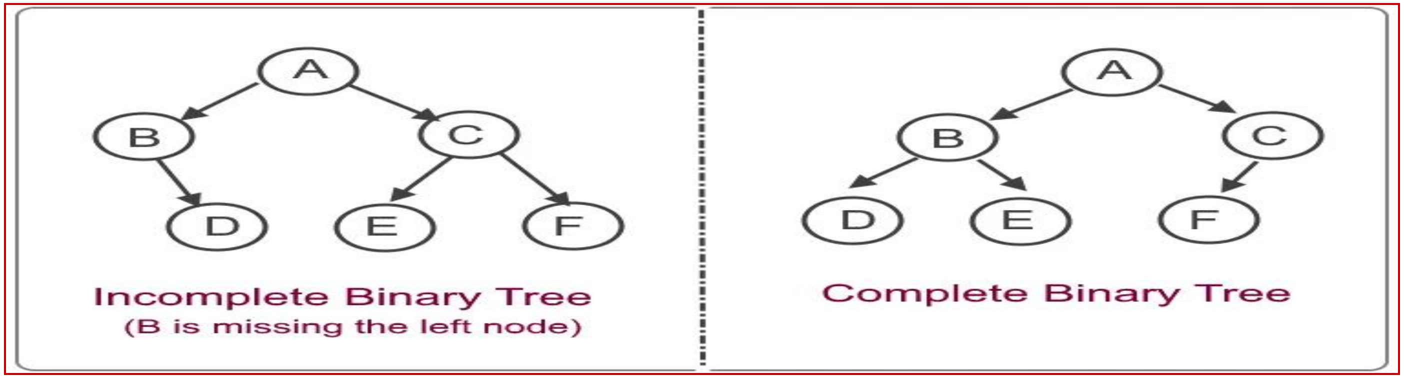
E.g.

- 2 Leaves in First Example having 2*2-1=3 Nodes.
- 3 Leaves in Second Example having 3*2-1=5 Nodes.
- 4 Leaves in Third Example having 4*2-1=7 Nodes

Types of Binary Tree...

2.Complete Binary Tree:

- ➢A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

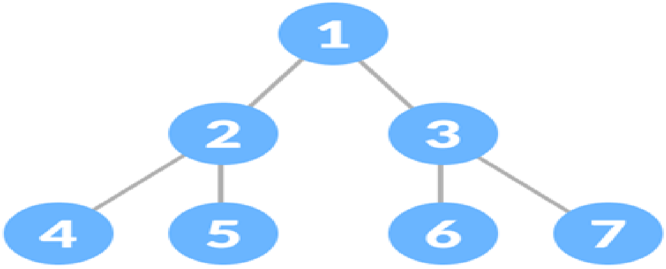


Incomplete Binary Tree
(B is missing the left node)

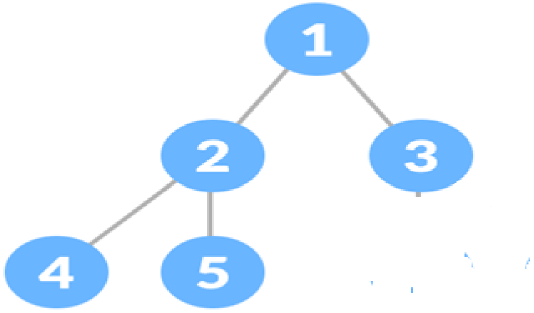
Complete Binary Tree

3.Perfect Binary Tree:

- ➢A **perfect binary tree** is a **binary tree** in which all interior nodes have two children and all leaves are at same level.



Perfect Binary Tree



Not Perfect Binary Tree

Types of Binary Tree...

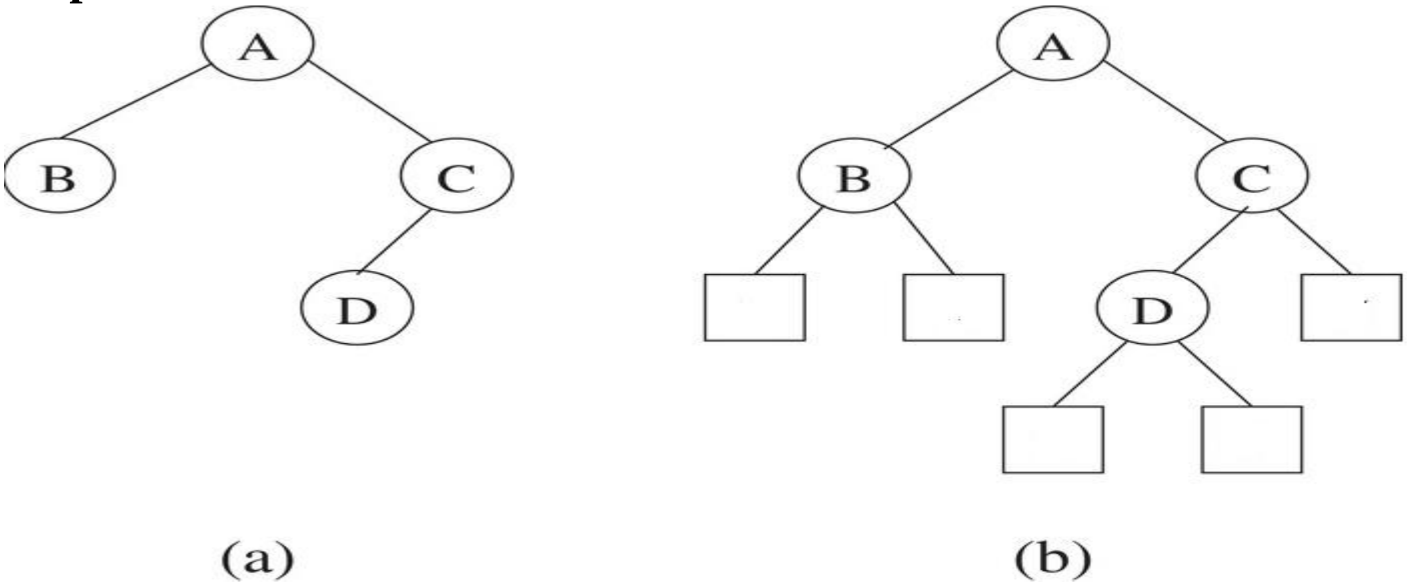
4.Extended Binary Tree:

- Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called external nodes whereas other nodes are called internal nodes.

- **Properties of External binary tree**

  1. The nodes from the original tree are internal nodes and the special nodes are external nodes.
  2. All external nodes are leaf nodes and the internal nodes are non-leaf nodes.
  3. Every internal node has exactly two children and every external node is a leaf.

- **Example:**



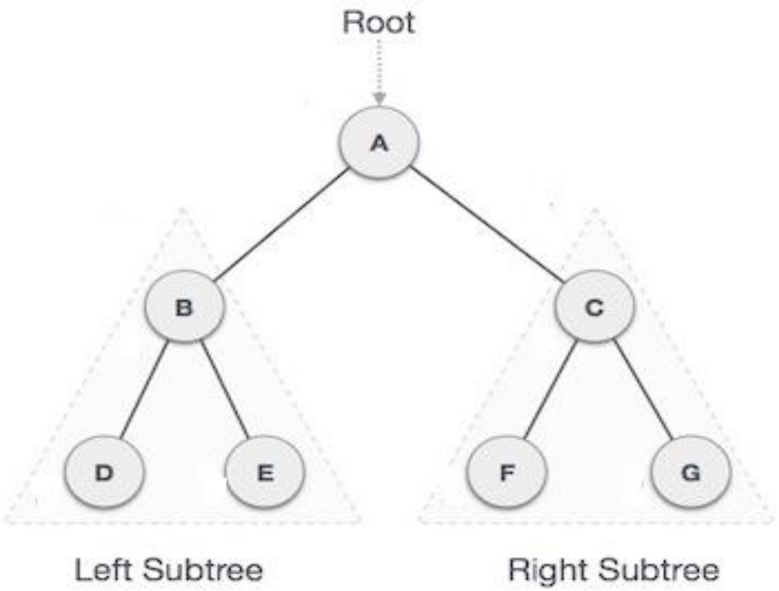
Binary Tree Traversals

Binary Tree Traversal

- ➢Traversal is a process to visit all the nodes of a tree and print their values . Because, all nodes are connected via edges (links) we always start from the root node.
- ➢Different types of tree traversals are:
    1. In-order Traversal
    2. Pre-order Traversal
    3. Post-order Traversal

1. In-order Traversal (LDR)

1. First, visit all the nodes in the left sub tree
2. Then the root node.
3. Visit all the nodes in the right sub tree.

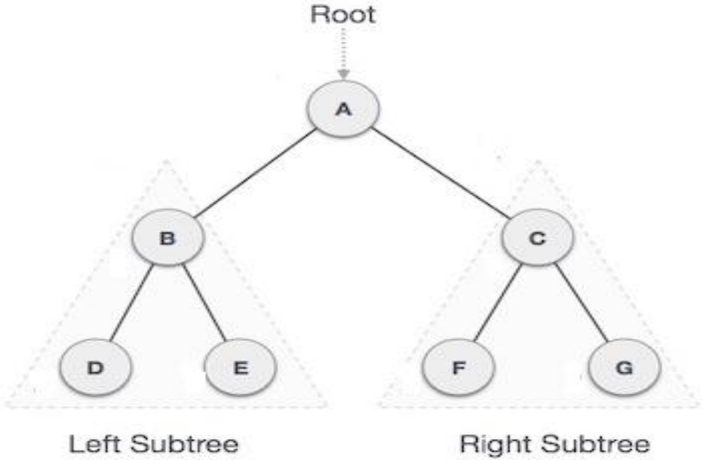


In-order: D → B → E → A → F → C → G

Binary Tree Traversals

2. Pre-order Traversal (DLR)

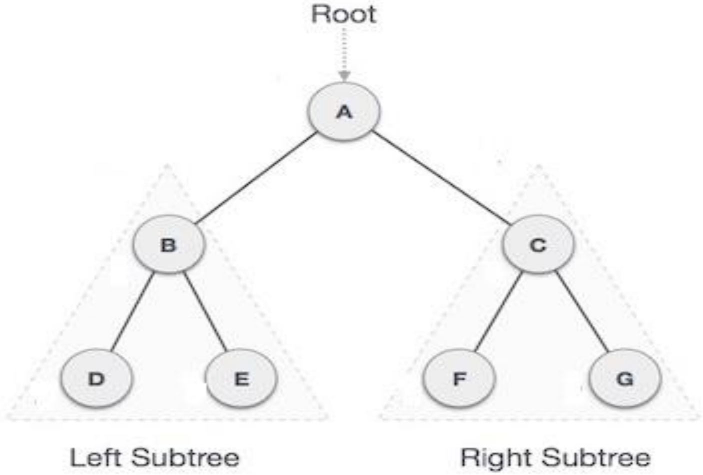
1. Then the root node.
2. First, visit all the nodes in the left sub tree.
3. Visit all the nodes in the right sub tree.



Pre -order: A → B → D → E → C → F → G

3.Postorder traversal

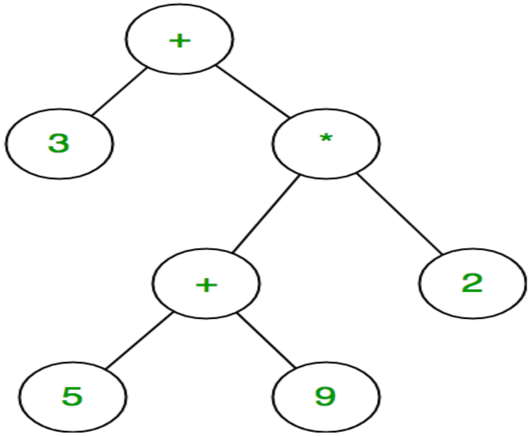
1. Visit all the nodes in the left subtree.
2. Visit all the nodes in the right subtree.
3. Visit the root node.



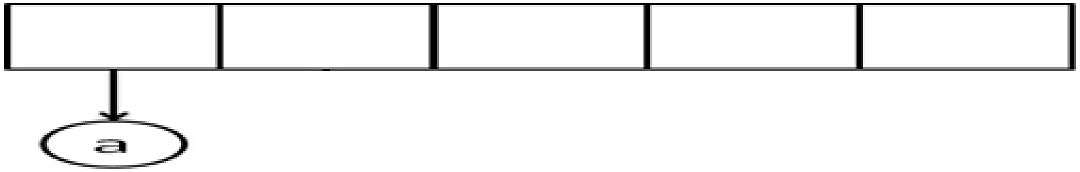
Post -order: ***D → E → B → F → G → C → A***

Expression Tree

- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- **Example**: Expression tree for 3 + ((5+9)*2)

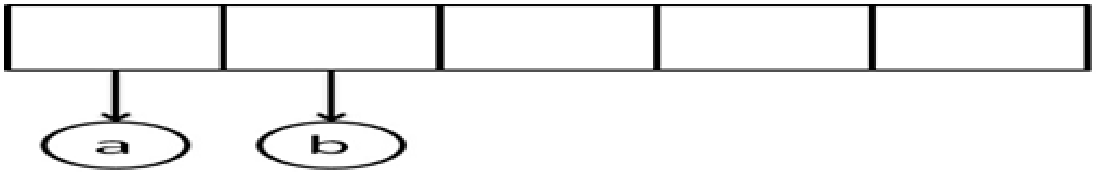


- **Steps for construction of Expression tree for a given expression:**
  1. Convert the given expression to post fix expression.
  2. Scan the postfix expression from left to right one character at a time.
     1. If character is operand push that into stack.
     2. If character is operator pop two values from stack make them its child and push current node again.
  3. At the end only element of stack will be root of expression tree.
- **Problem: Expression is (a+b)*c**
  1. The postfix is: a b + c *
  2. The first symbol 'a' is operand, it is pushed onto the stack

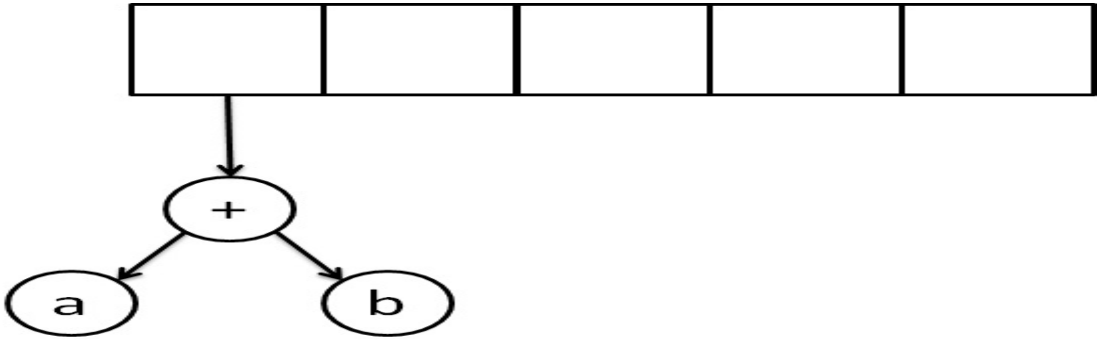


Expression Tree...

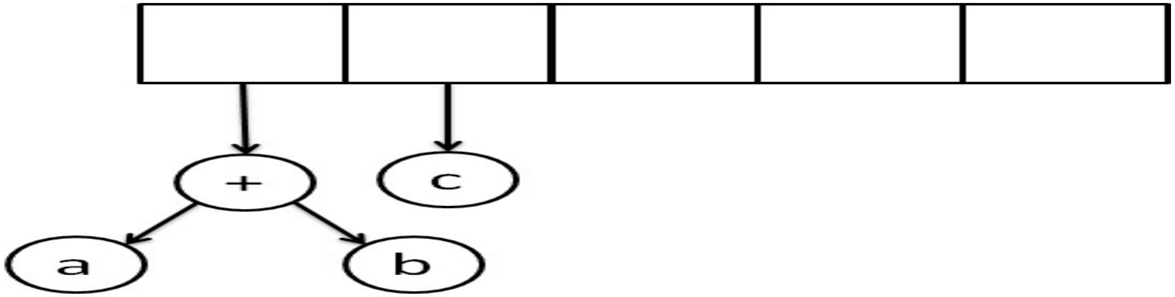
Next symbol ‘b’ is an operand , insert ‘b’ onto stack



Next, **'+'** symbol, so topmost two operands are popped , a new tree is formed and push a pointer to it onto the stack.

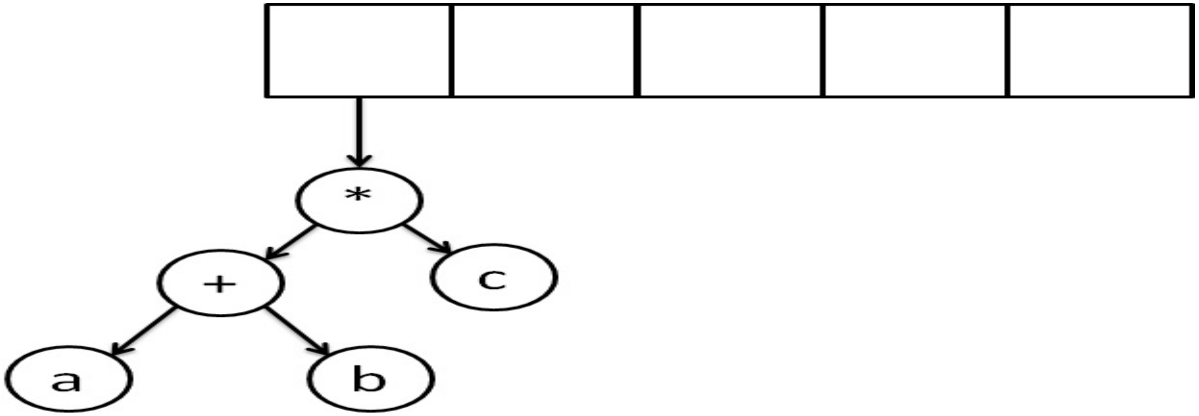


Next, **'c'** is read, we create one node tree and push a pointer to it onto the stack.

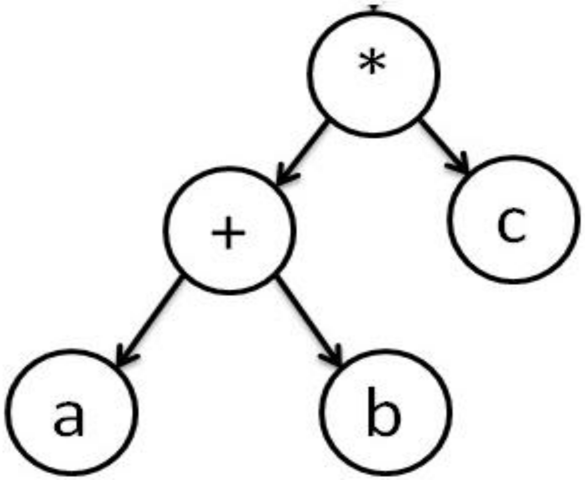


Expression Tree...

Finally, the last symbol is read **' * '**, we pop topmost two tree pointers and form a new tree with **' * '** as root, and a pointer to the final tree is pushed on the stack.



Pop the only element from the stack , we get an expression tree



Threaded Binary Tree

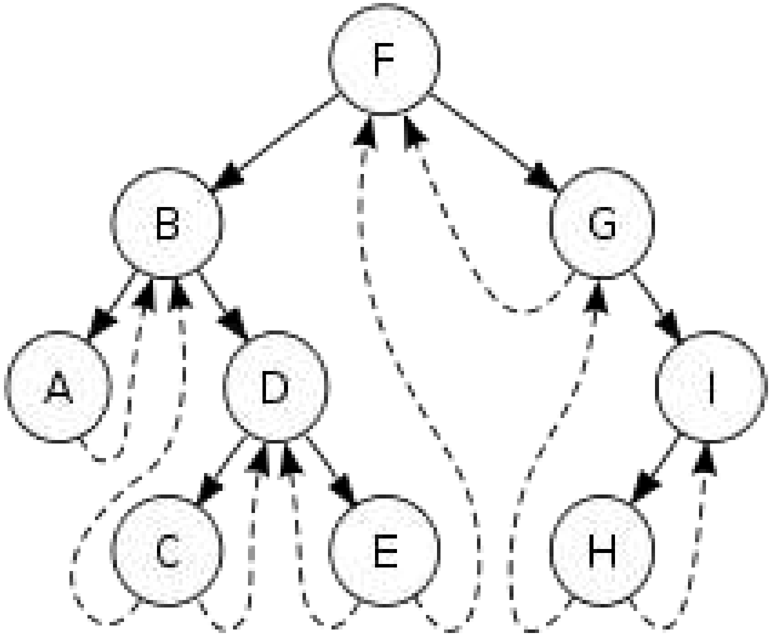
- **Threaded Binary Tree:**

  Binary tree is said to be threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node .

- **Why do we need Threaded Binary Tree?**

  1. Binary trees have a lot of wasted space, the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
  2. Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.

- **Example:**



Threaded Binary Tree...

- **Types of threaded binary tree**

1. **Right threaded binary tree**

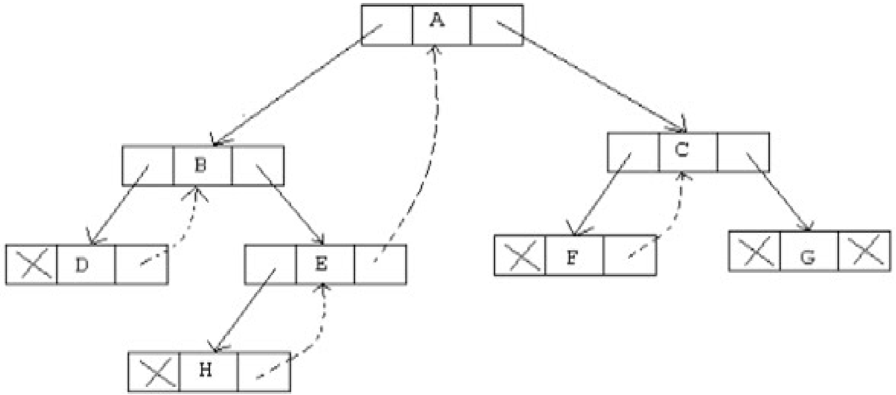
   The right NULL pointer of the node is replaced by a thread to the inorder successor of that node is called as right threaded binary tree.

2. **Left threaded binary tree**

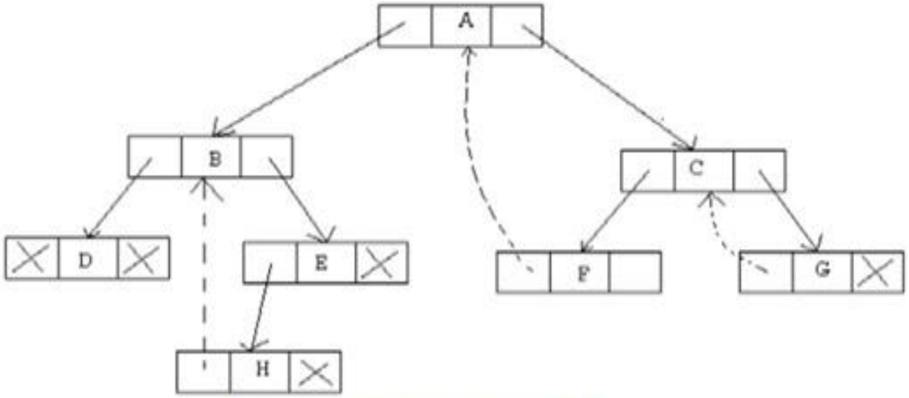
   The left NULL pointer of the node is replaced by a thread to the inorder predecessor of that node is called as right threaded binary tree.

3. **Fully threaded binary tree**

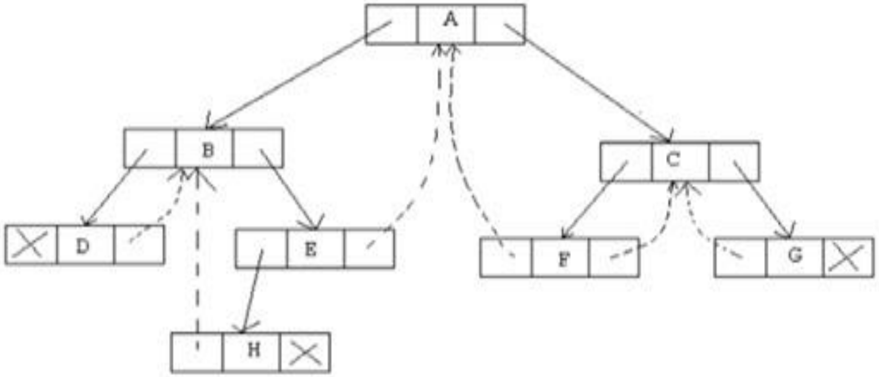
   Both left and right NULL pointers can be used to point to Inorder predecessor an successor of that node respectively, is called a fully threaded binary tree.



RIGHT THREADED BINARY TREE



LEFT THREADED BINARY TREE



Fully Threaded Binary Tree

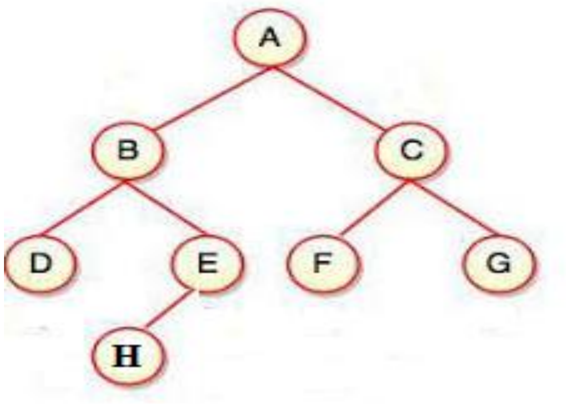
Threaded Binary Tree...

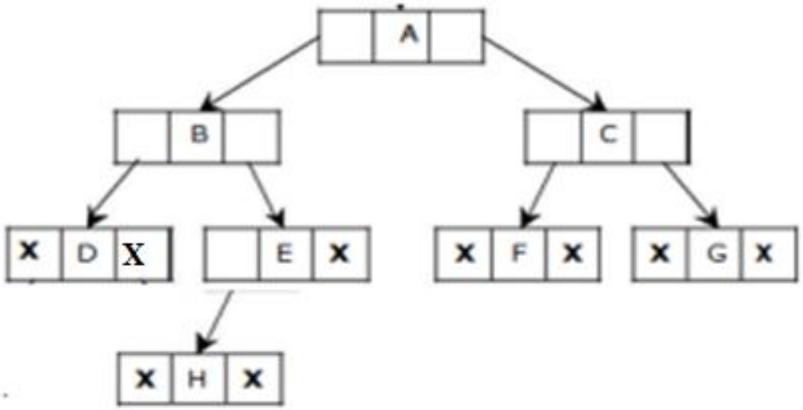
- **Construct threaded binary tree for following binary tree**

1. Write the inorder traversal for the given binary tree

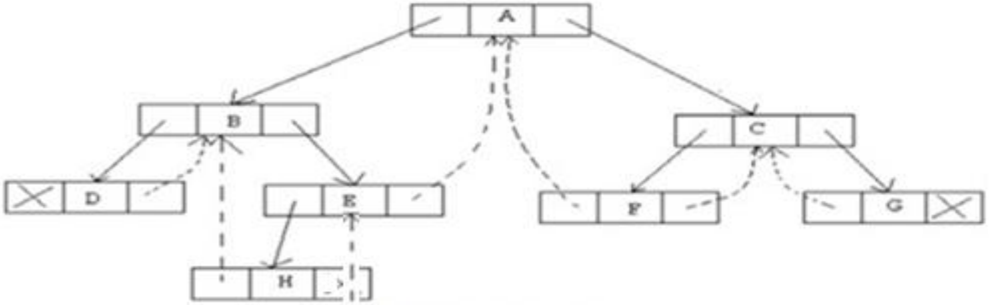
   Inorder is **D B H E A F C G**

2. Represent the binary tree using linked list representation





3. Link the left and right NULL pointers to inorder Successor or predecessor respectively as shown below



Binary Heap

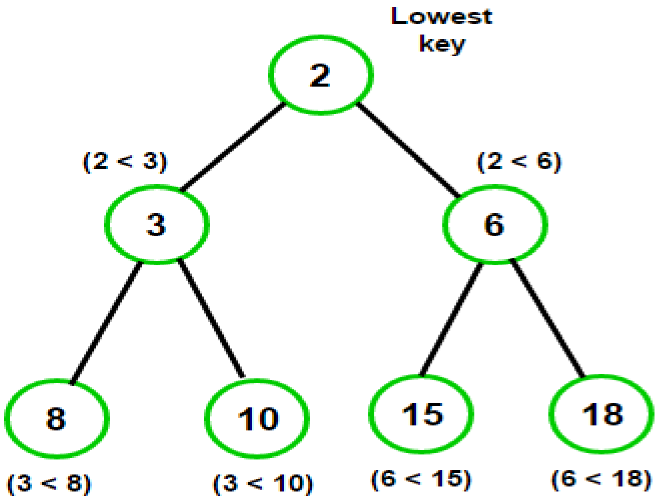
- A binary tree is said to be binary heap if it follows the following two properties:

1. **Heap order property**

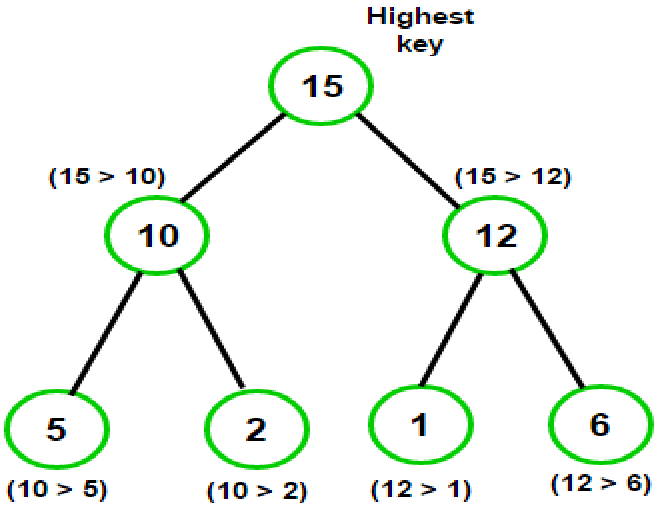
   Every node is less than or equal to its children (Min Heap) or every node is greater than or equal to its children(Max Heap).

2. **Structure property**

   The tree is completely filled except possibly the bottom level, which is filled from left to right.



Min Heap
(Parent key is less than or equal to (≤) the child key)



Max Heap
(Parent key is greater than or equal to (≥) the child key)

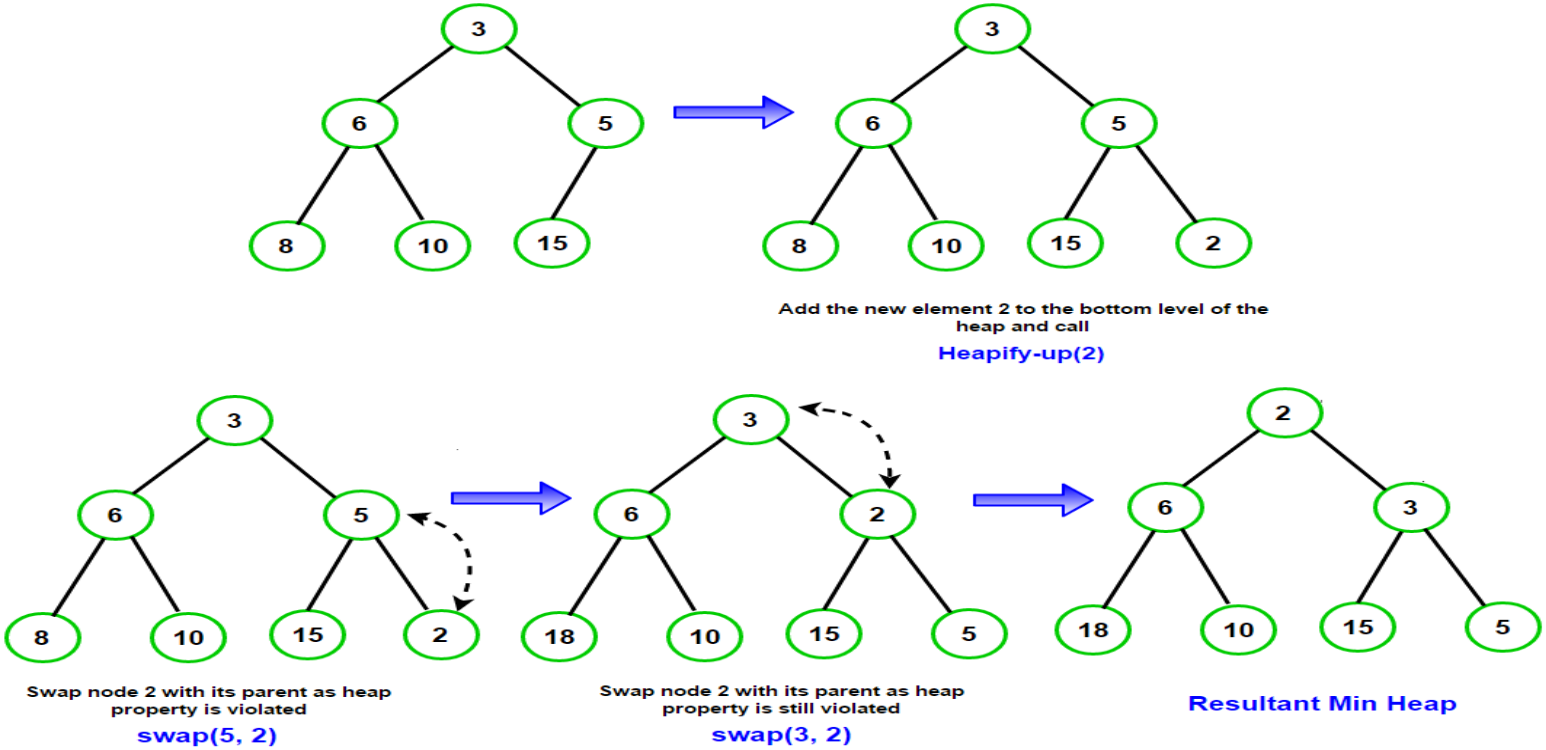
Binary Heap...

- **Operations:**

  1. **Insertion:**

     Steps:

     1. First increase the heap size by 1, so that it can store the new element.
     2. Insert the new element at the end of the Heap.
     3. This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, **heapify** this newly inserted element following a bottom-up approach.



Add the new element 2 to the bottom level of the heap and call

Heapify-up(2)

Swap node 2 with its parent as heap property is violated

swap(5, 2)

Swap node 2 with its parent as heap property is still violated

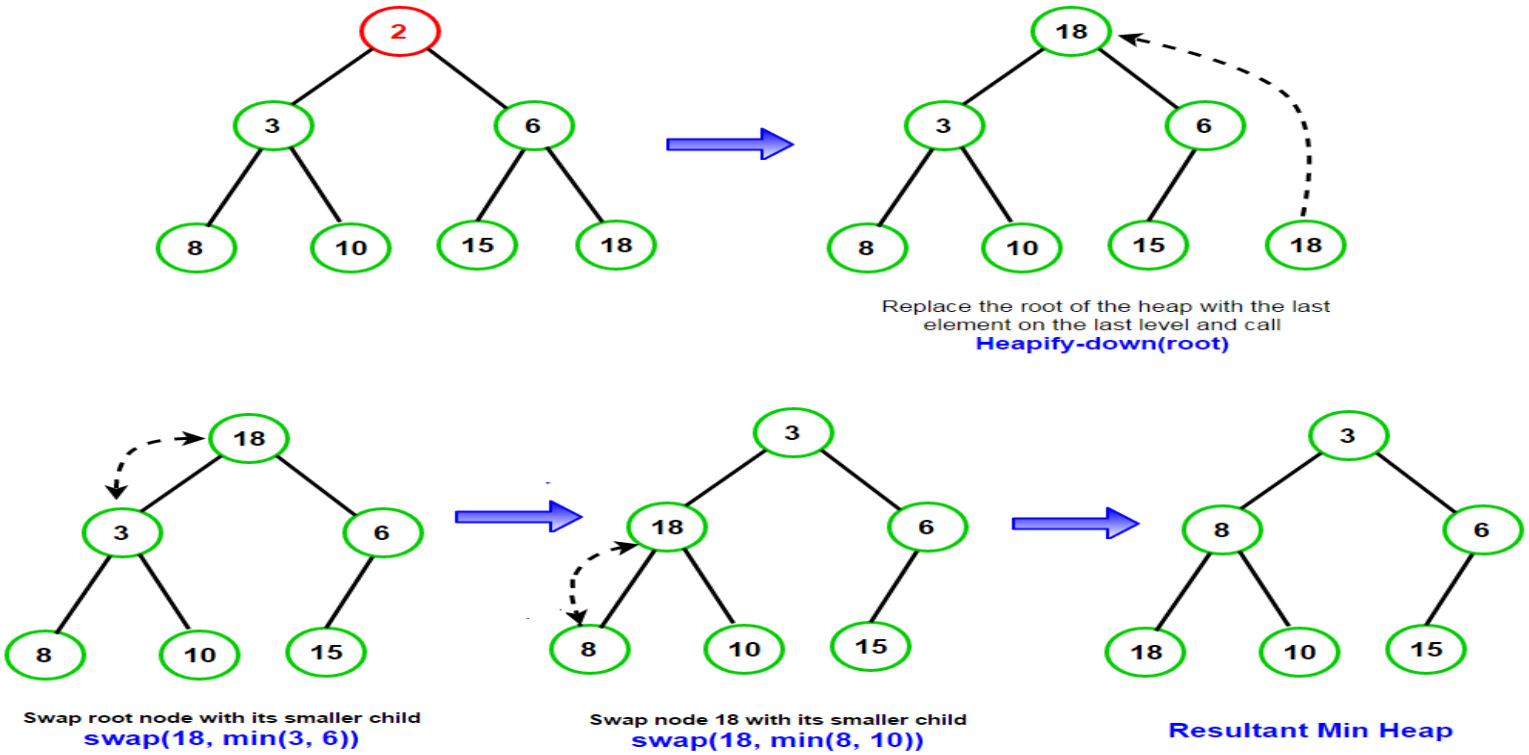
swap(3, 2)

Resultant Min Heap

Binary Heap...

2. Deletion:

1. Delete the root element
2. Replace the root by the last element.
3. Delete the last element from the Heap.
4. Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.



Replace the root of the heap with the last element on the last level and call
Heapify-down(root)

Swap root node with its smaller child
swap(18, min(3, 6))

Swap node 18 with its smaller child
swap(18, min(8, 10))

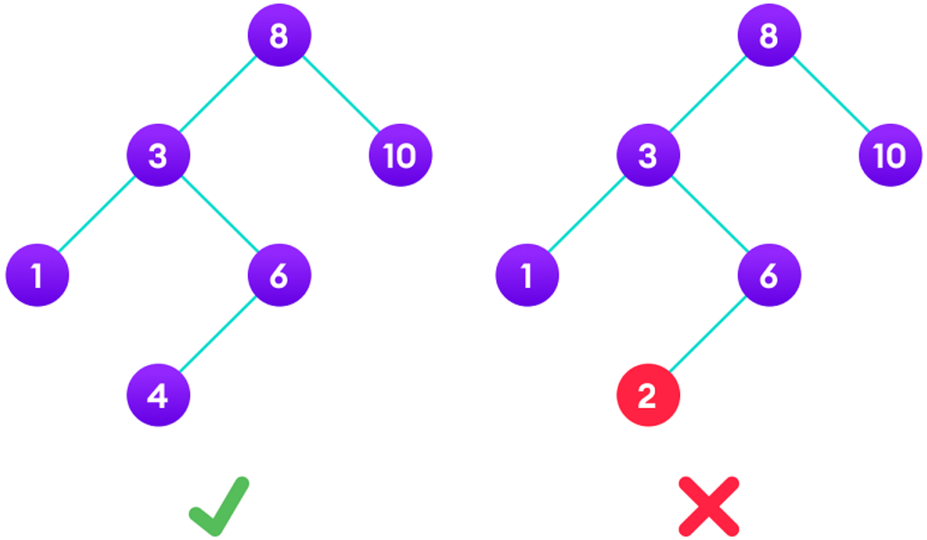
Resultant Min Heap

Binary Search Tree(BST)

- **Binary Search Tree:**

  A binary search tree, also known as an ordered binary tree, in which all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.

- **Example :**

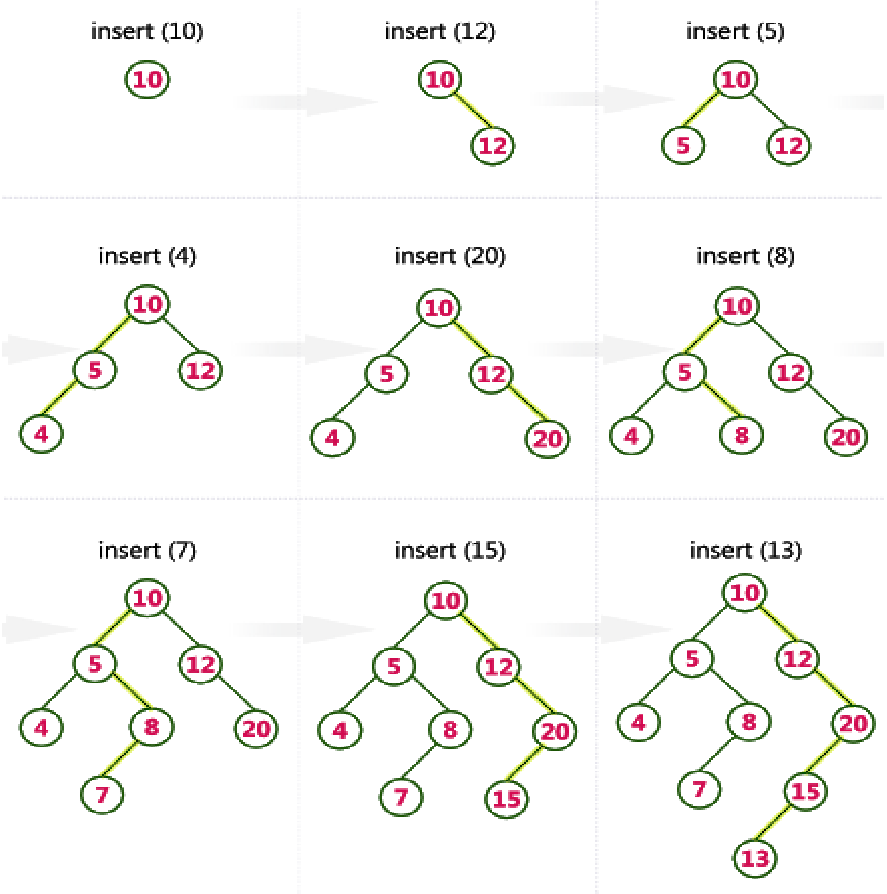


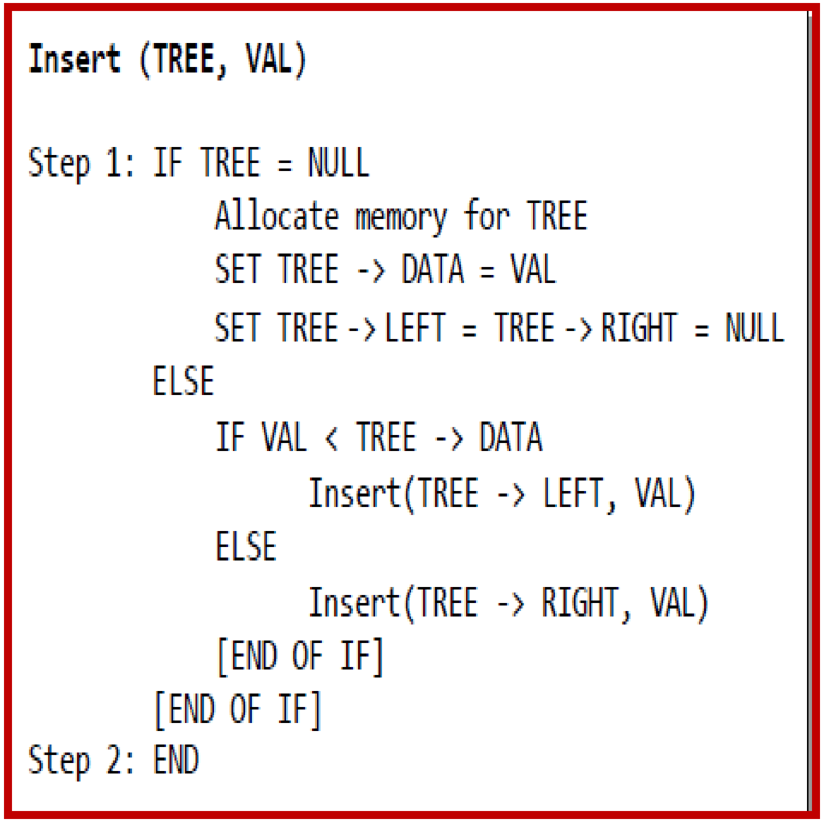
- **Operations :**
  1. Insert: insert a node in the BST.
  2. Search: Searches for a node in the BST.
  3. Delete: deletes a node from the BST.

Binary Search Tree(BST)...

➢ **Insert:**

Inorder to insert an element, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left sub tree and insert the data. Otherwise, search for the empty location in the right sub tree and insert the data.





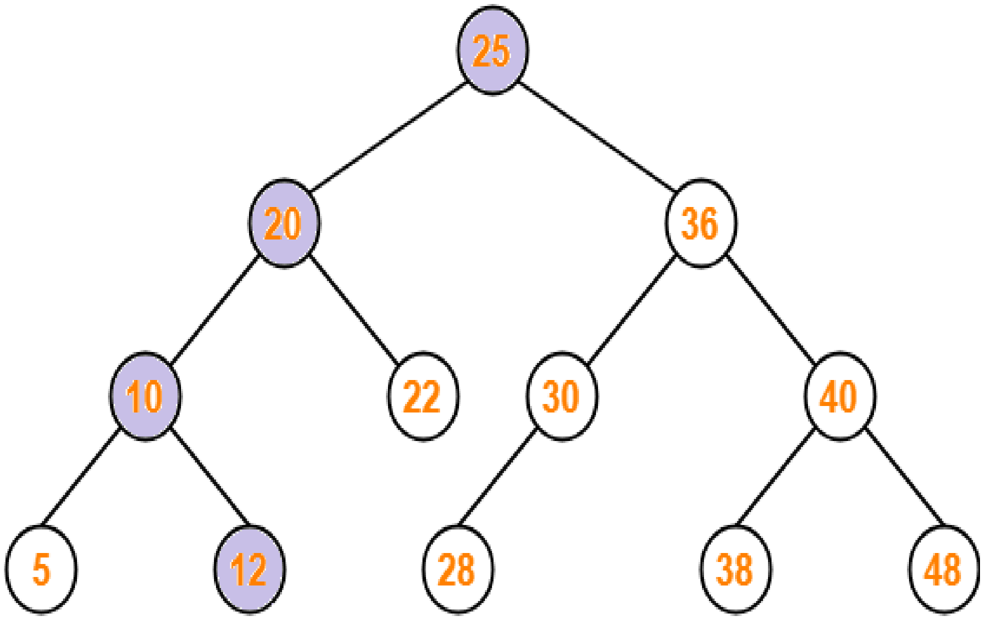
```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                Insert(TREE -> LEFT, VAL)
            ELSE
                Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

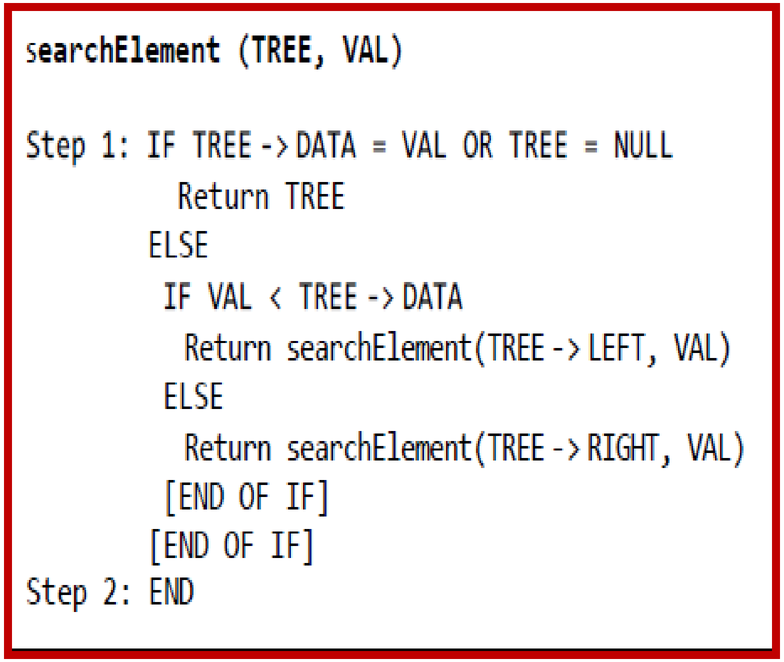
Binary Search Tree(BST)...

- **Search:**

  Inorder to search an element in BST, start from the root node. if the data is less than the key value, search for the element in the left sub tree. Otherwise, search for the element in the right sub tree. Follow the same procedure for each node.

- **Example:** Search for element 12





```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
            Return TREE
        ELSE
          IF VAL < TREE -> DATA
            Return searchElement(TREE -> LEFT, VAL)
          ELSE
            Return searchElement(TREE -> RIGHT, VAL)
          [END OF IF]
        [END OF IF]
Step 2: END
```

Binary Search Tree(BST)...

- **Delete:**

  Deleting a node from Binary search tree includes following three cases:

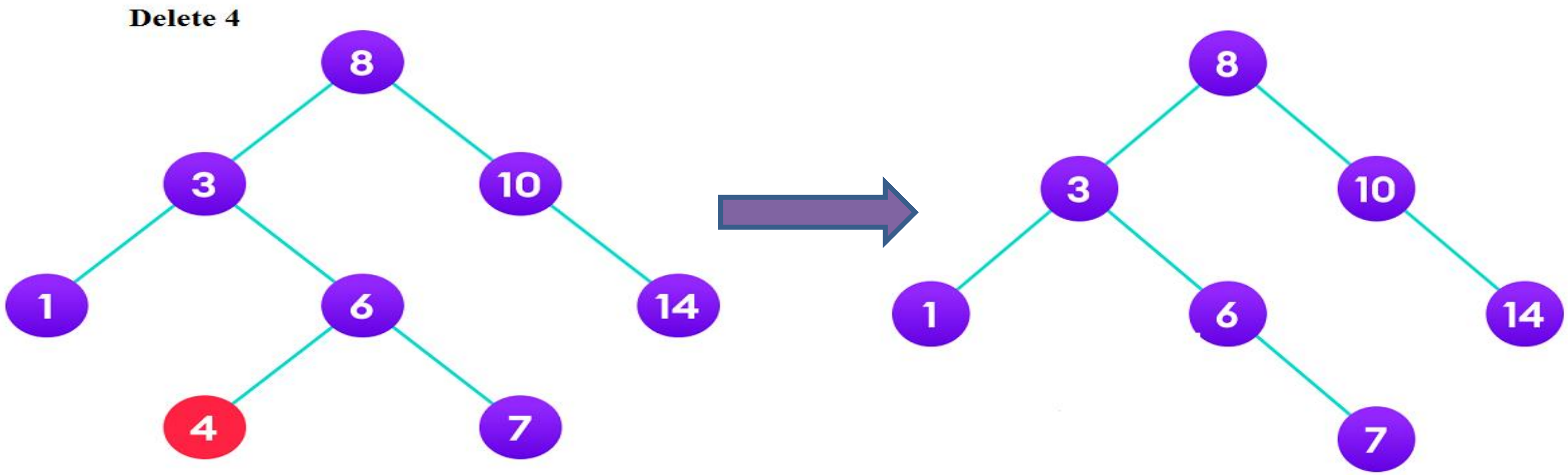
  Case 1: Deleting a Leaf node (A node with no children)

  Case 2: Deleting a node with one child

  Case 3: Deleting a node with two children

- **Case 1: Deleting a Leaf node (A node with no children)**

  Search for that node in BST, If the node to be deleted is the leaf node then simply delete the node from the tree .

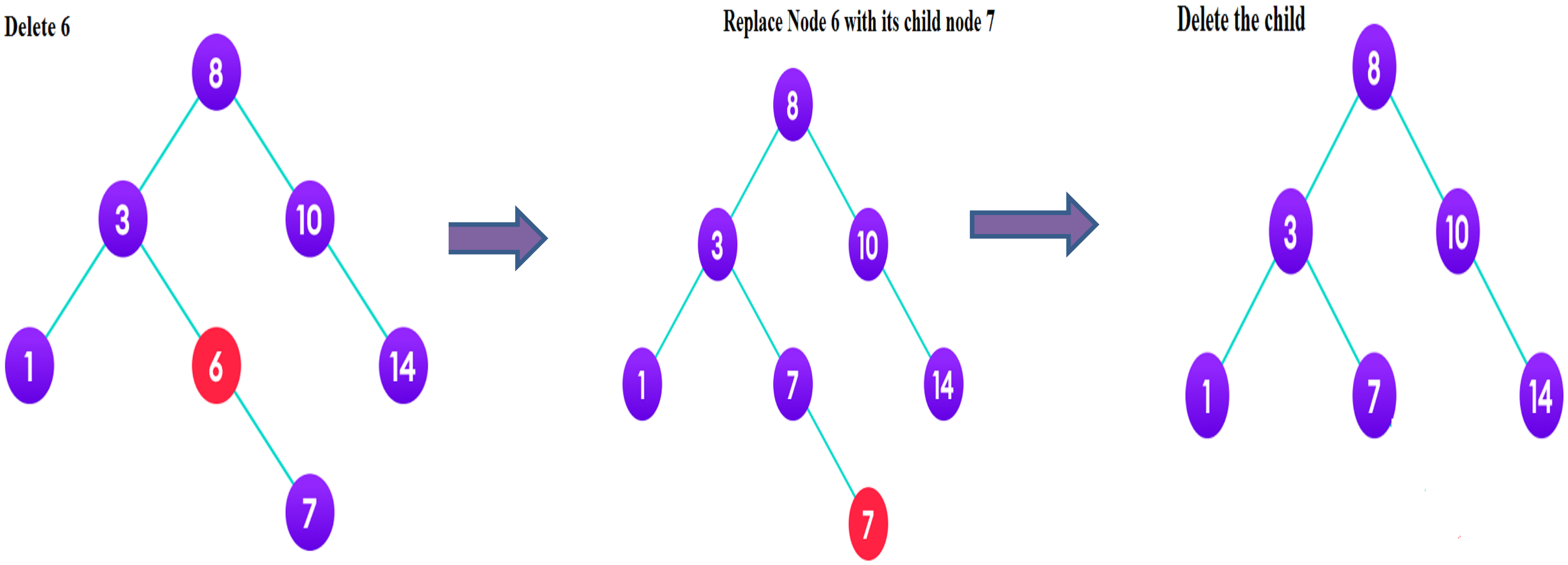


Binary Search Tree(BST)...

➢Case 2: Deleting a node with one child

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.



Delete 6

Replace Node 6 with its child node 7

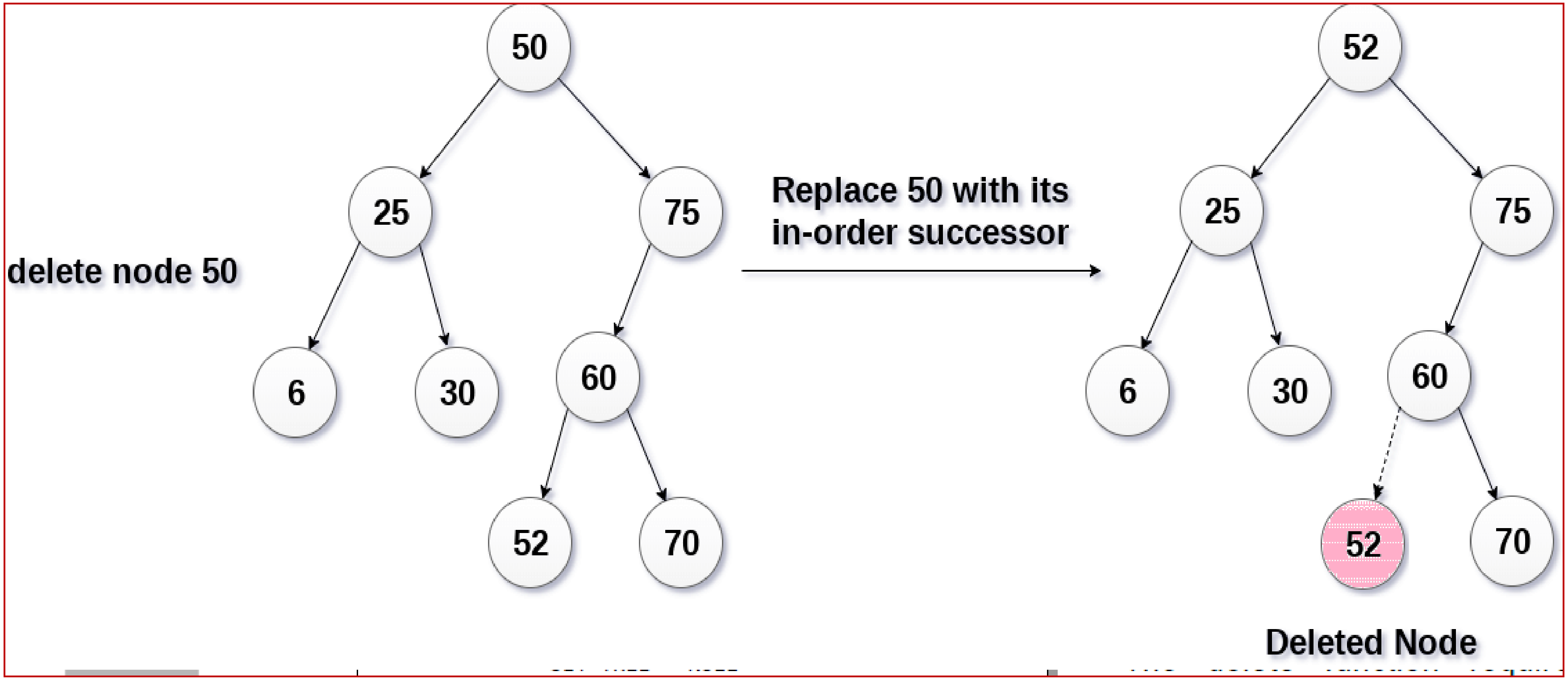
Delete the child

Binary Search Tree(BST)...

➢Case 3: Deleting a node with two children

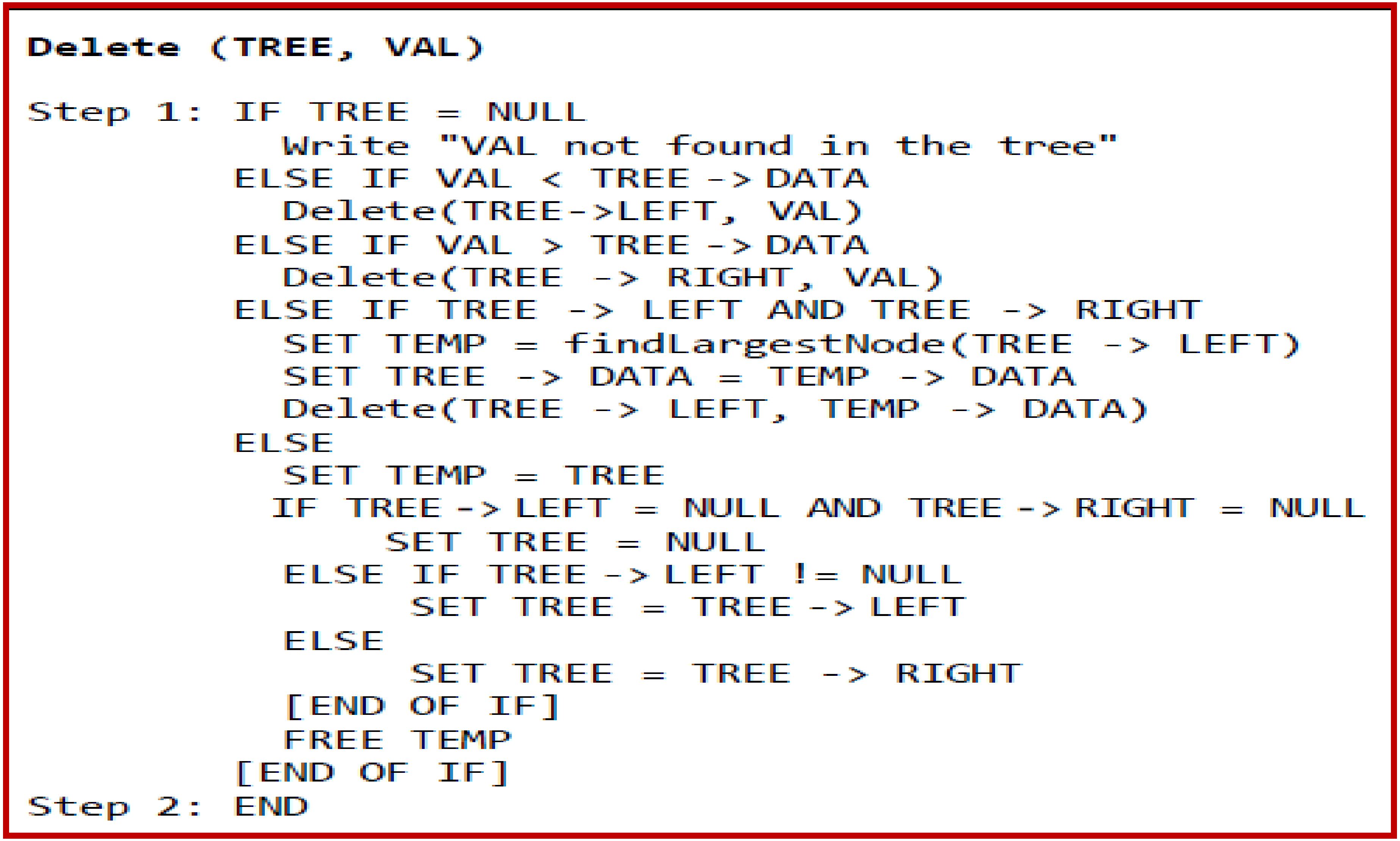
In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
2. The in-order predecessor or the successor can then be deleted using case1 or case2.



Binary Search Tree(BST)...

Deletion:



```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
           Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
           Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
           Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
           SET TEMP = findLargestNode(TREE -> LEFT)
           SET TREE -> DATA = TEMP -> DATA
           Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
           SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
                 SET TREE = NULL
           ELSE IF TREE -> LEFT != NULL
                 SET TREE = TREE -> LEFT
           ELSE
                 SET TREE = TREE -> RIGHT
           [END OF IF]
           FREE TEMP
        [END OF IF]
Step 2: END
```

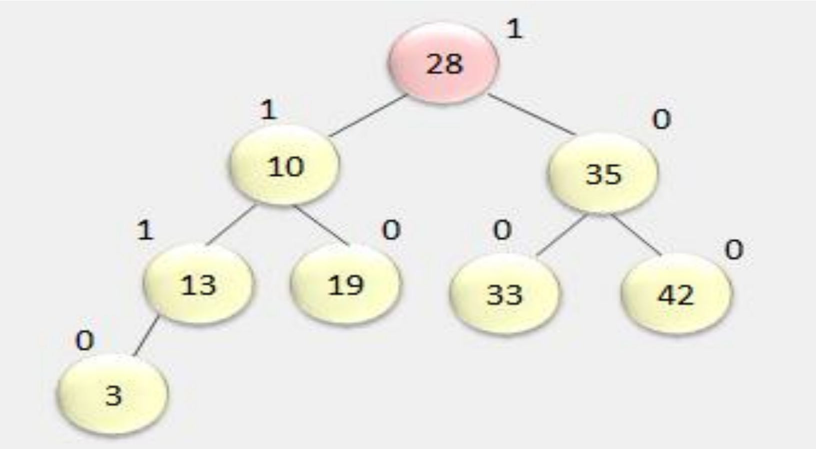
AVL Tree

AVL Tree:

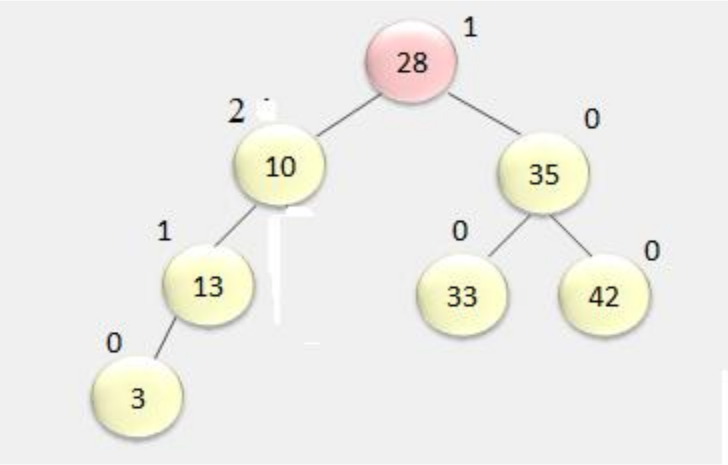
- It is a self-balancing binary search tree invented by G.M. **A**delson-**V**elsky and E.M. **L**andis in 1962.
- The structure of an AVL tree is the same as that of a binary search tree, in addition to this every node in this tree is associated with balance factor.
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- A binary search tree in which every node has a balance factor of **–1, 0, or 1** is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.



AVL Tree

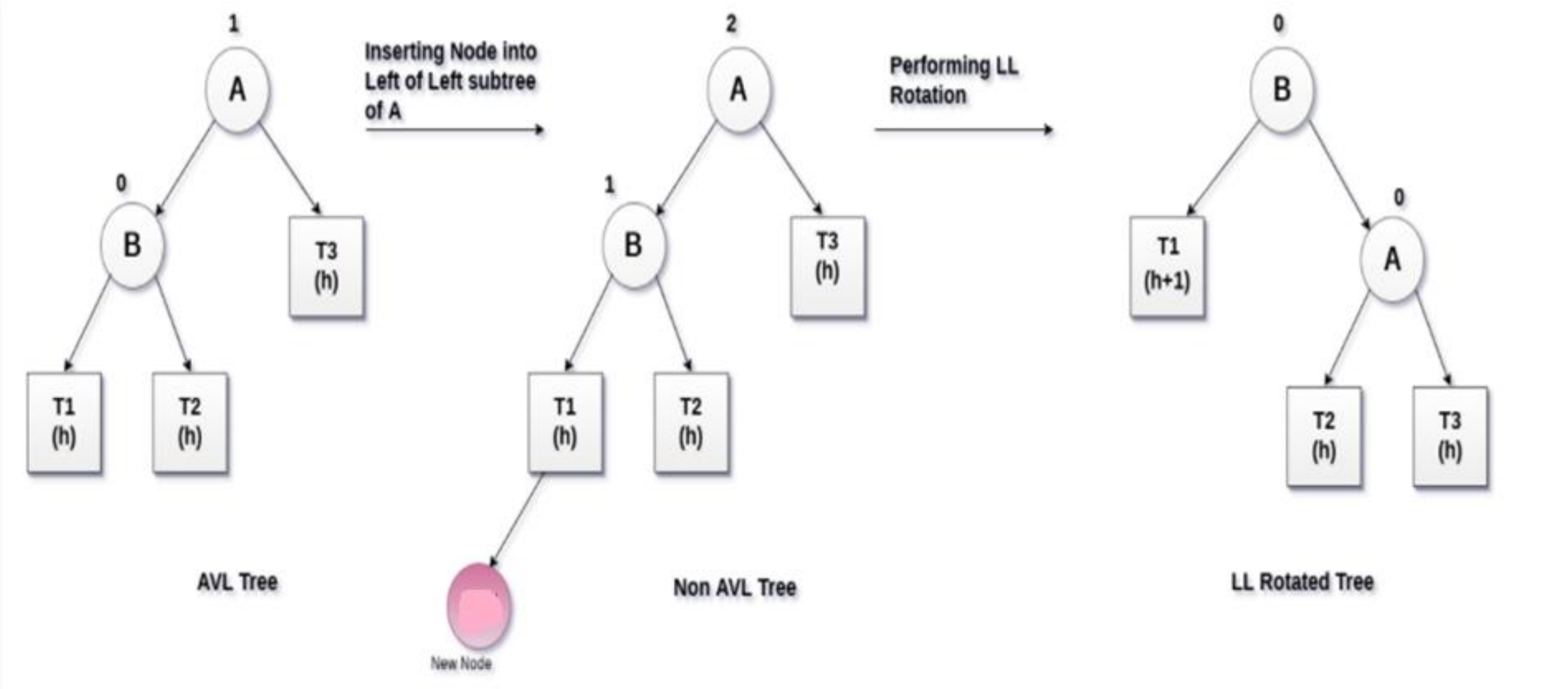


Not AVL Tree

AVL Tree...

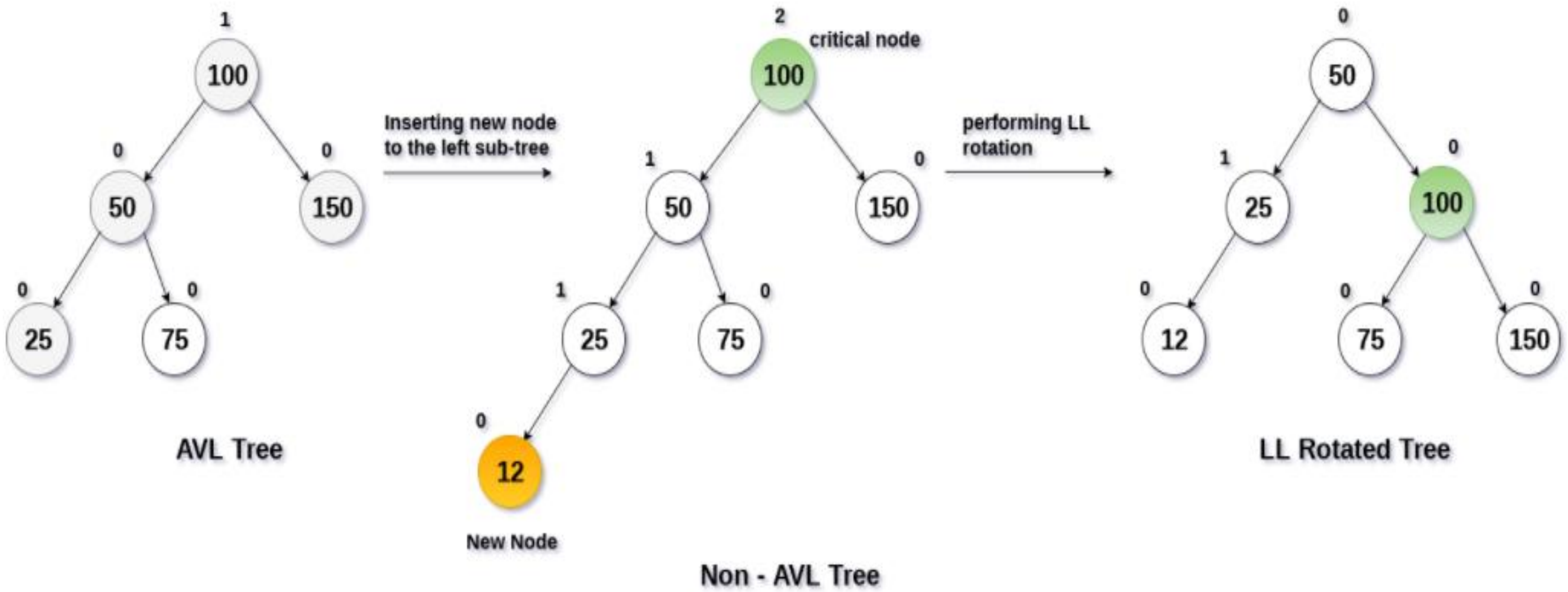
- The Unbalanced node in the AVL tree is called critical node.
- In order to balance the unbalanced tree/node we have to make rotations .Four types of rotations are:

1. **LL rotation:** The new node is inserted in the left sub-tree of the left sub-tree of the critical node.



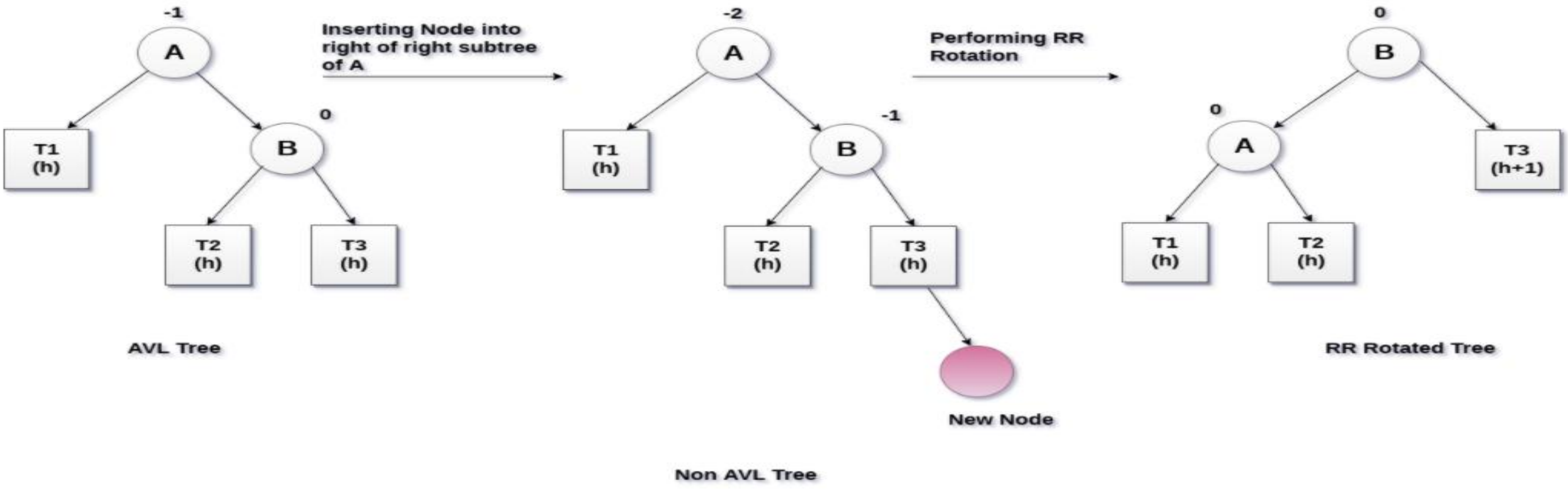
AVL Tree...

LL Rotation Example:



2.RR Rotation: The new node is inserted in the right sub-tree of the right sub-tree of the critical node.

AVL Tree...



Inserting Node into right of right subtree of A

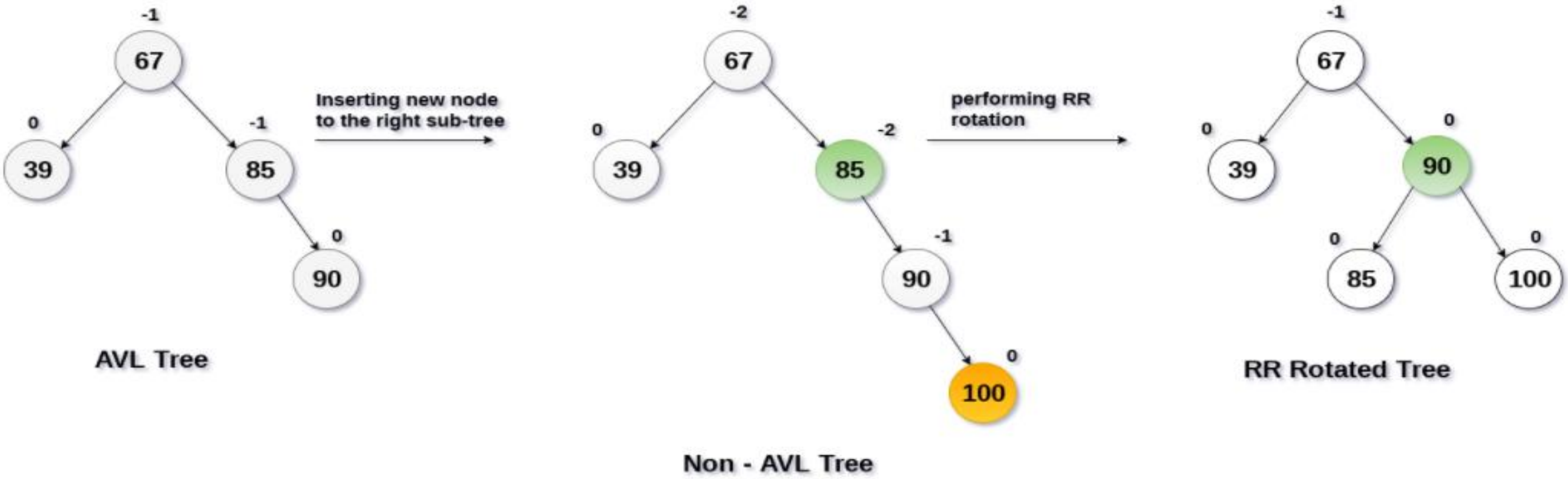
Performing RR Rotation

AVL Tree

Non AVL Tree

RR Rotated Tree

RR Rotation Example



Inserting new node to the right sub-tree

performing RR rotation

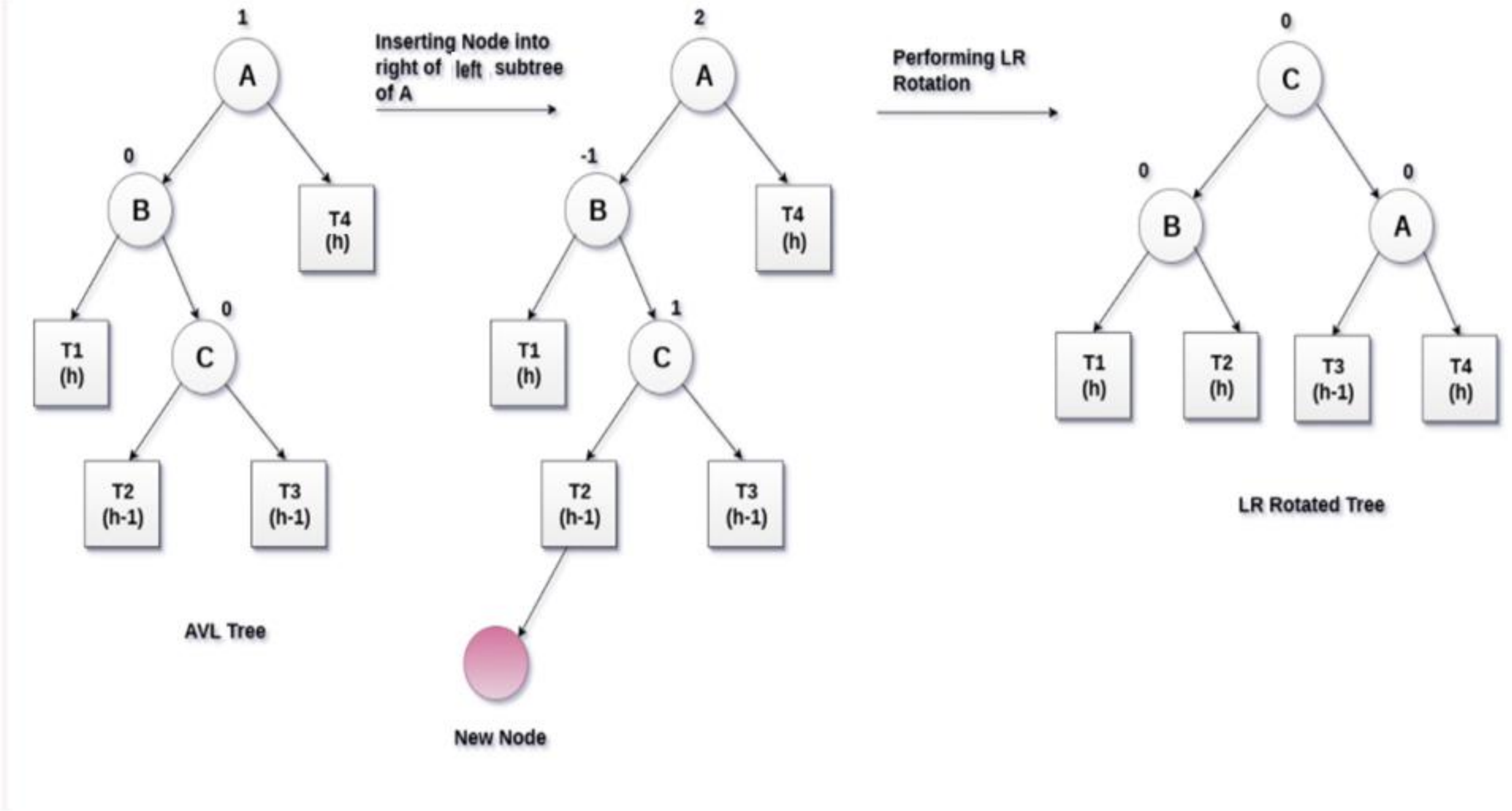
AVL Tree

Non - AVL Tree

RR Rotated Tree

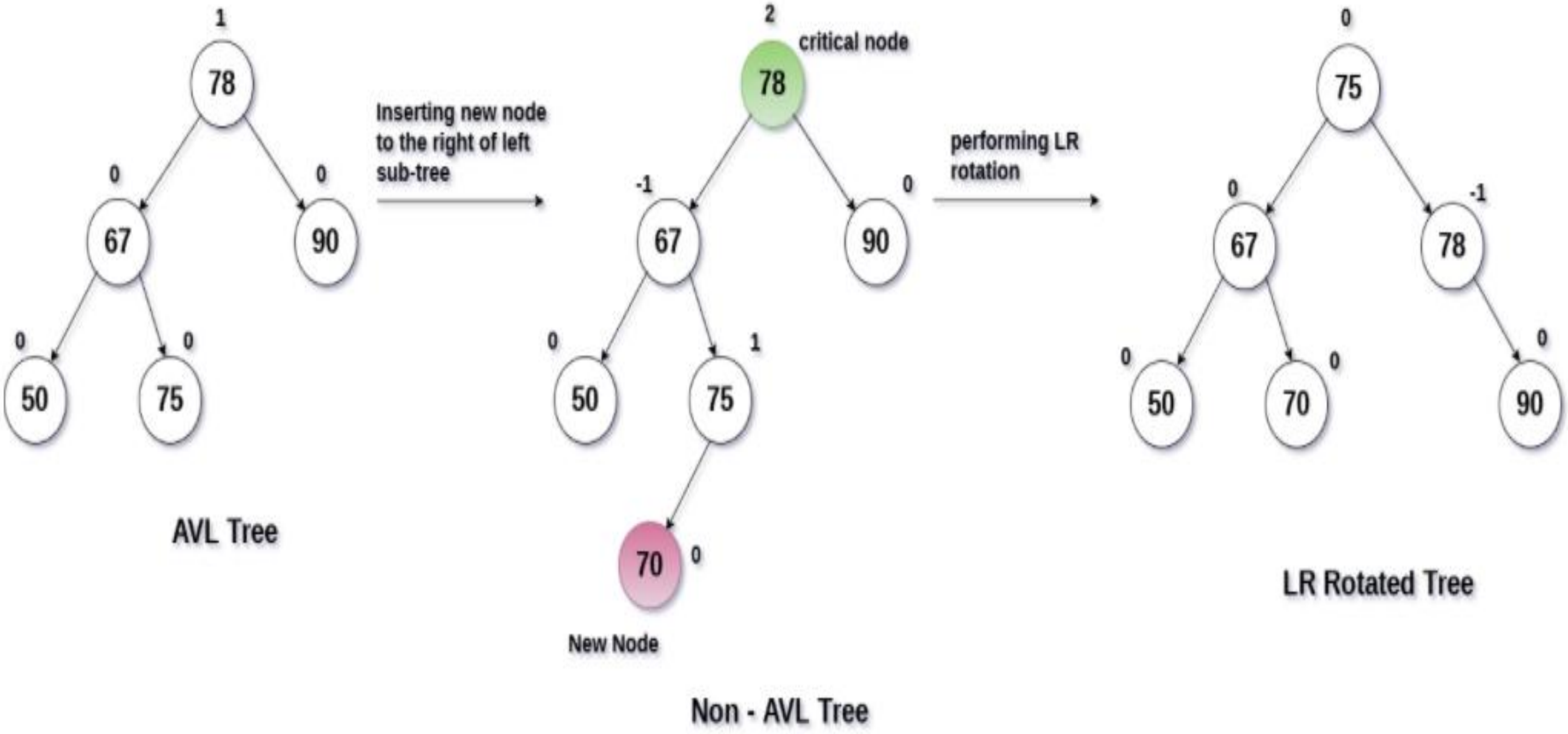
AVL Tree...

3.LR Rotation : The new node is inserted in the right sub-tree of the left sub-tree of the critical node.



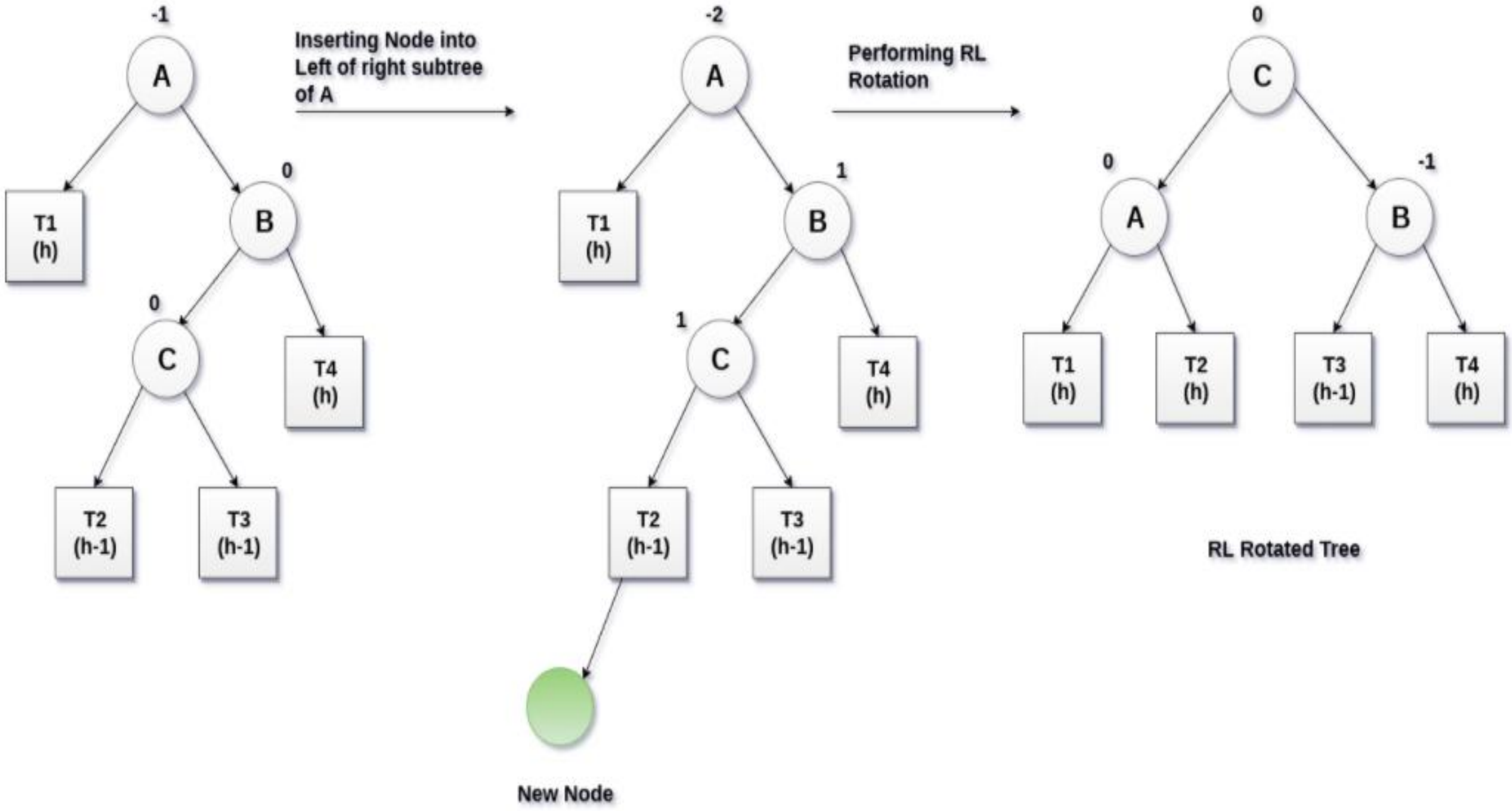
AVL Tree...

LR Rotation Example



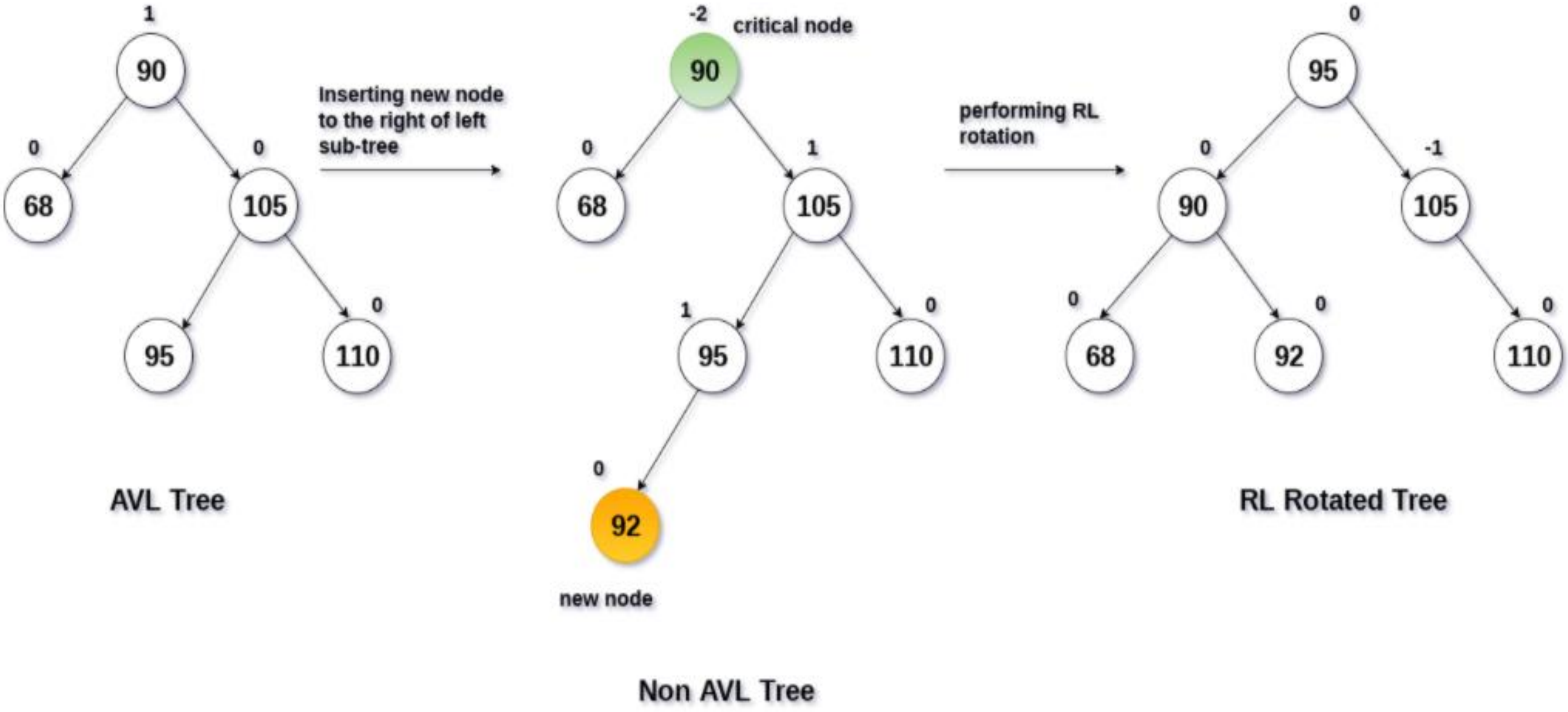
AVL Tree...

4.RL Rotation: The new node is inserted in the left sub-tree of the right sub-tree of the critical node.



AVL Tree...

RL Rotation Example



AVL Tree...

Operations:

1. **Insertion**
2. **Deletion**
3. **Searching** (Search operation in AVL is same as BST)

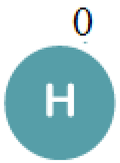
1. Insertion:

- Insertion in an AVL tree is done in the same way as it is done in a binary search tree.
- In the AVL tree, the new node is always inserted as the leaf node.
- After insertion the balance factor of each node has to be updated. If any unbalance node is found then rotation has to be performed to balance the tree.

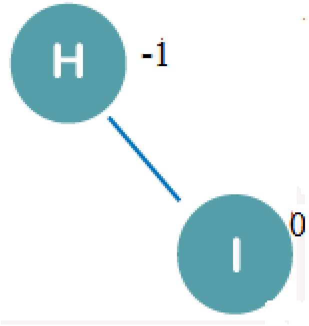
Example:

Construct an AVL tree for the elements: H, I, J, B, A, E, C, F, D, G, K, L

Insert H



Insert I

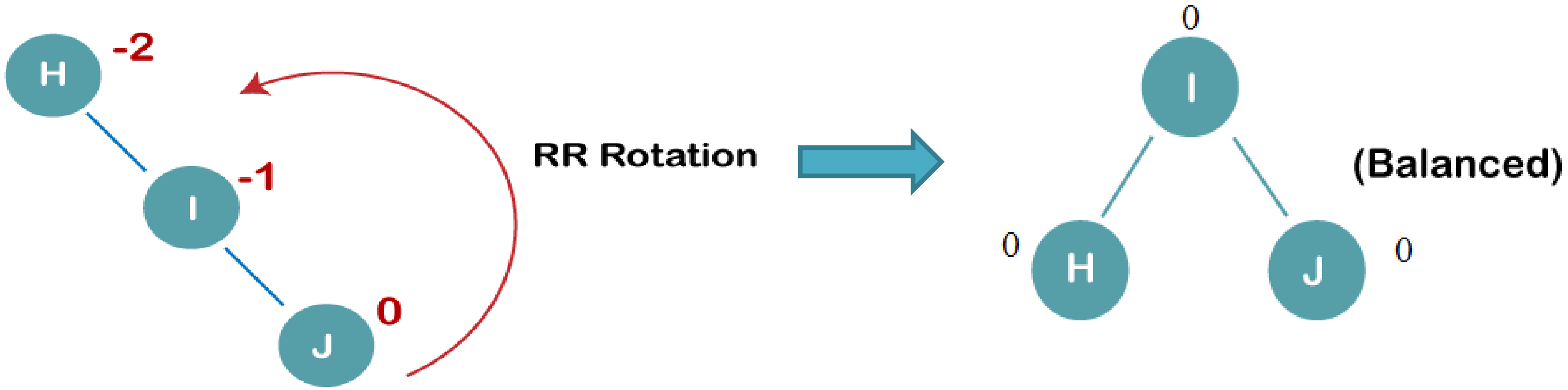


AVL Tree...

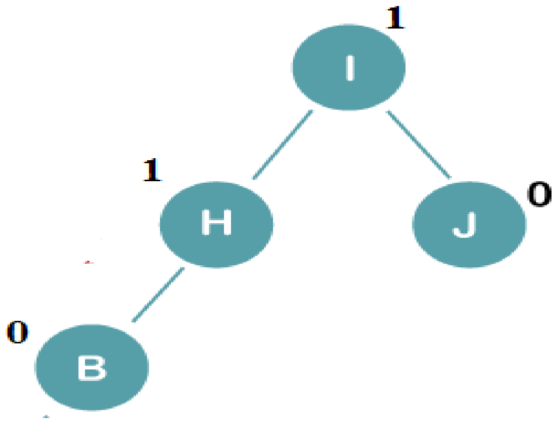
Example...

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

Insert J



Insert B

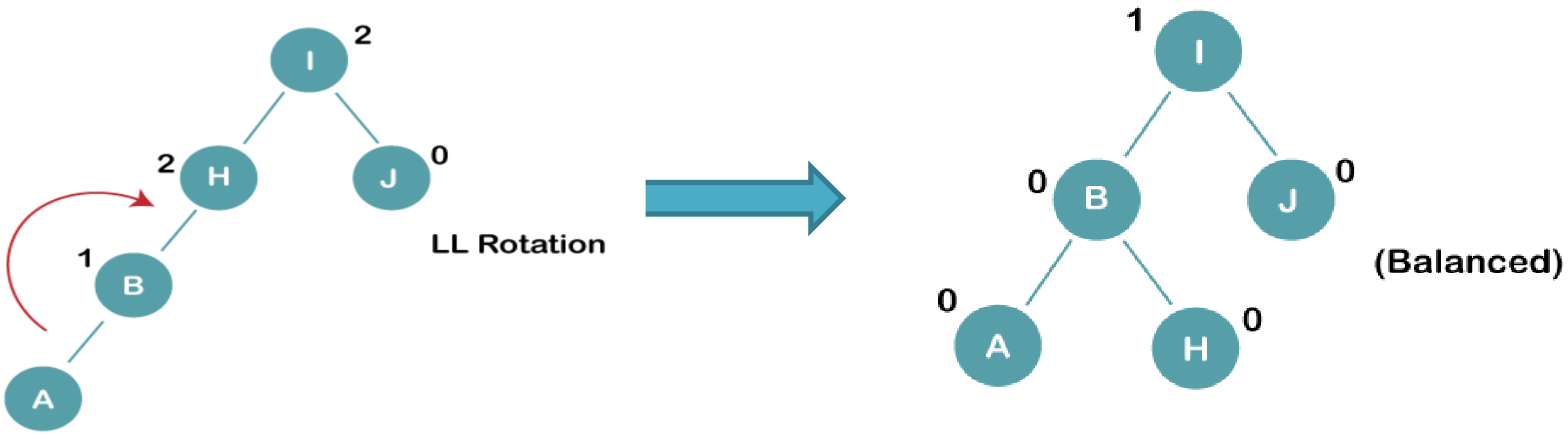


AVL Tree...

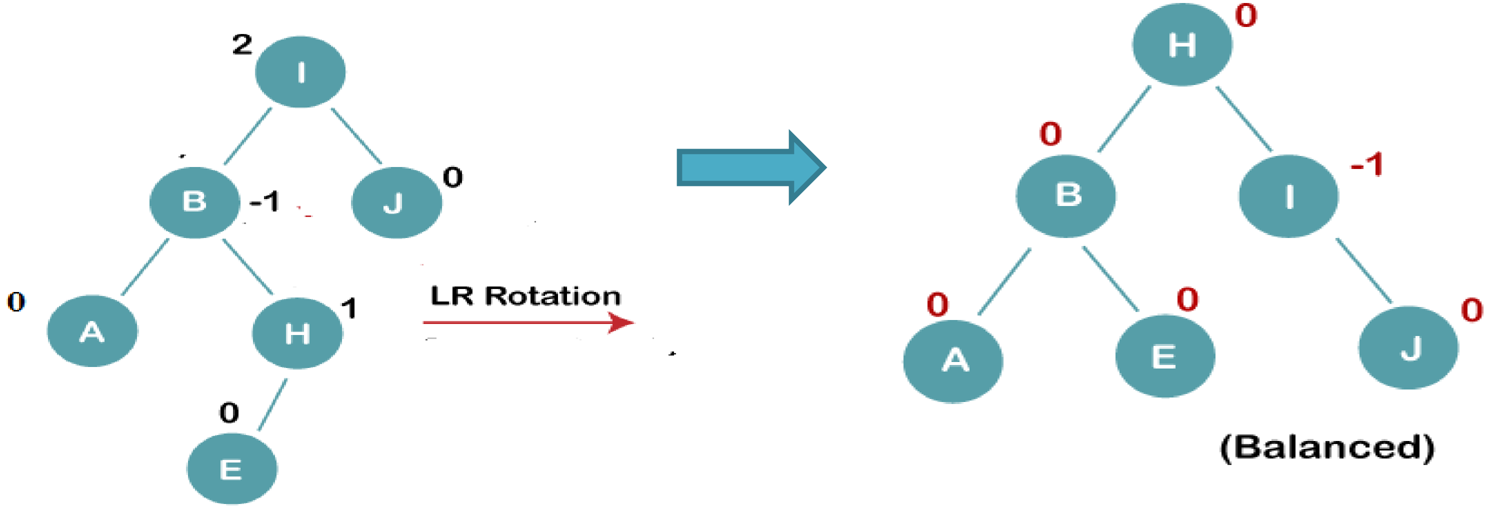
Example...

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

Insert A



Insert E

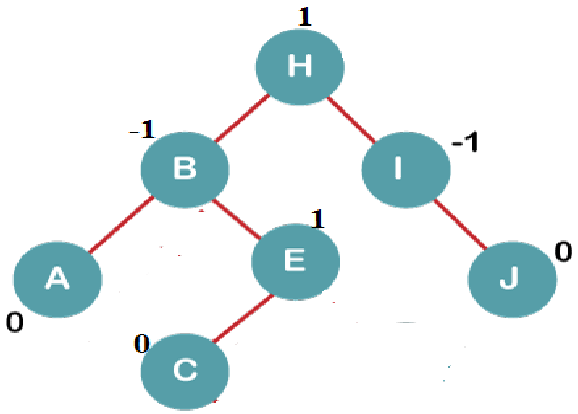


AVL Tree...

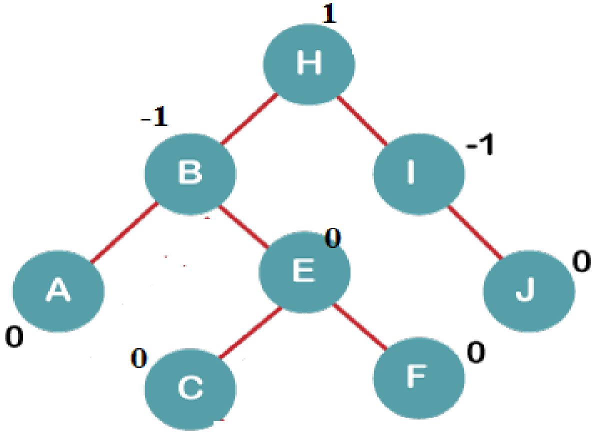
Example...

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

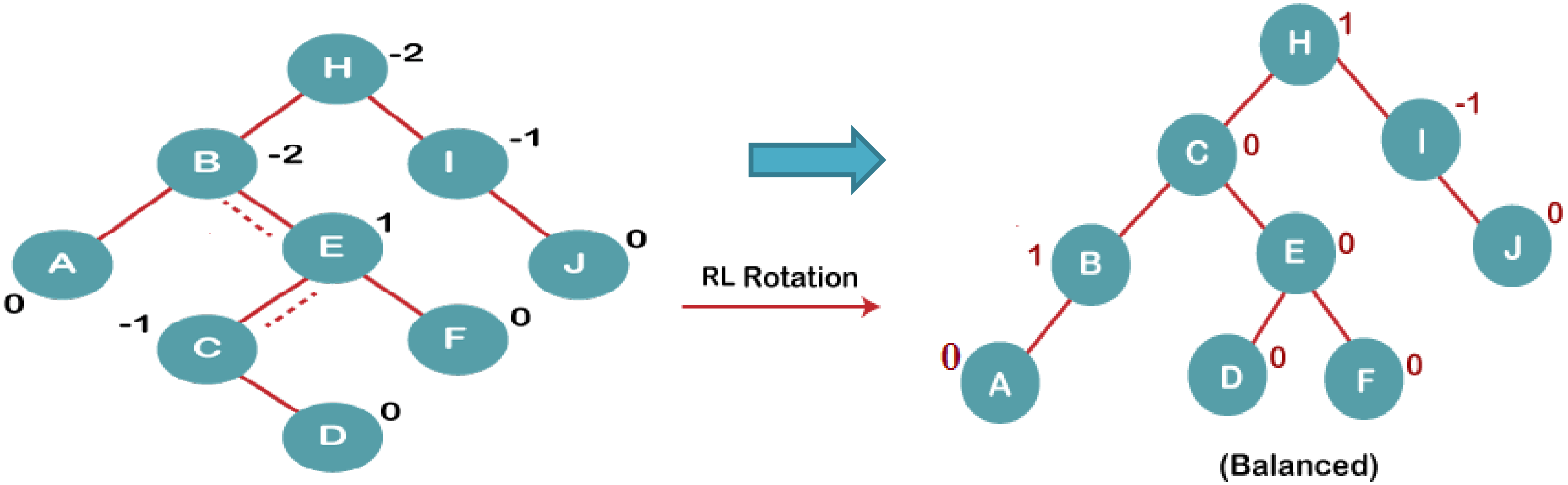
Insert C



Insert F



Insert D

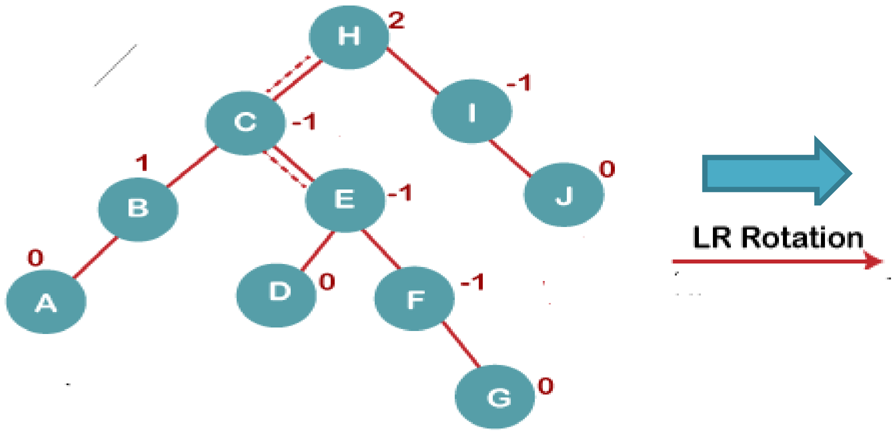


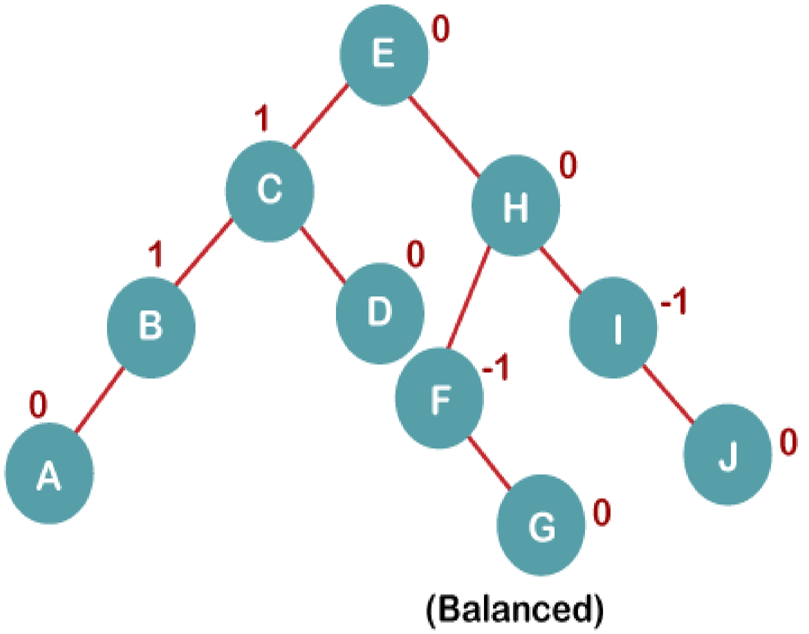
AVL Tree...

Example...

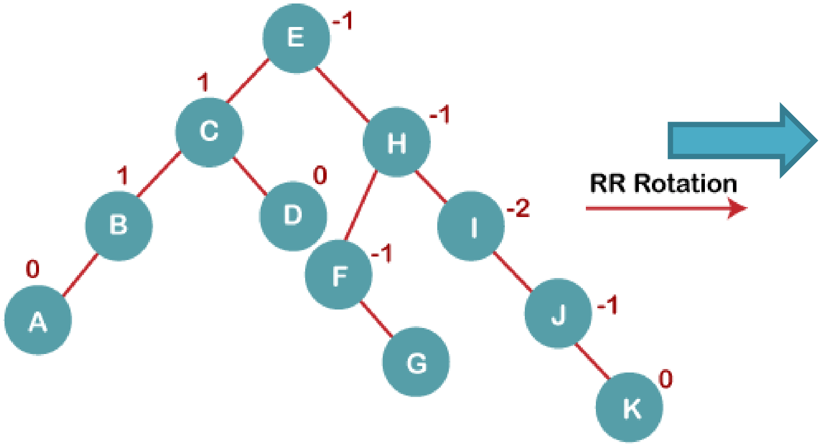
Elements: H, I, J, B, A, E, C, F, D, G, K

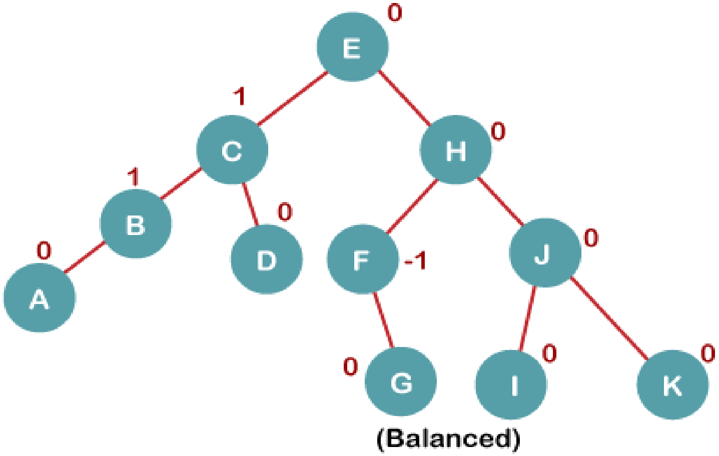
Insert G





Insert K



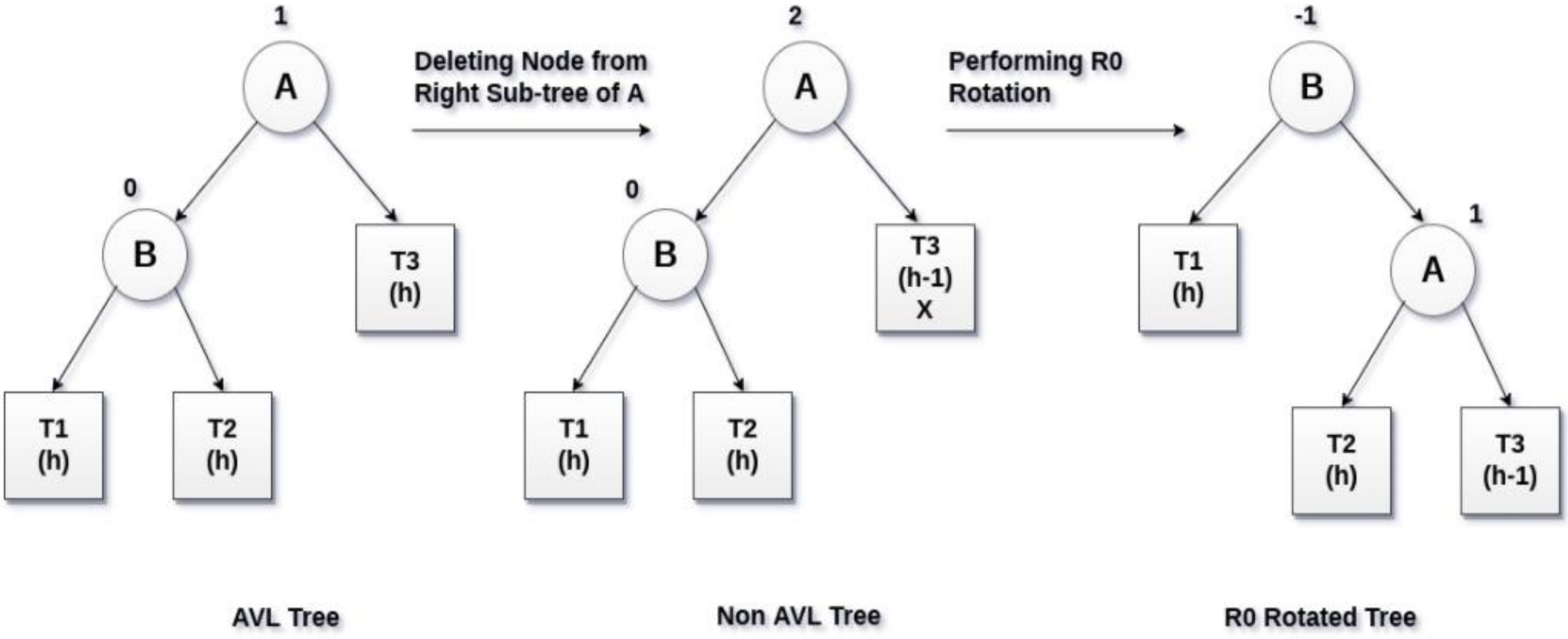


AVL Tree...

2. Deletion:

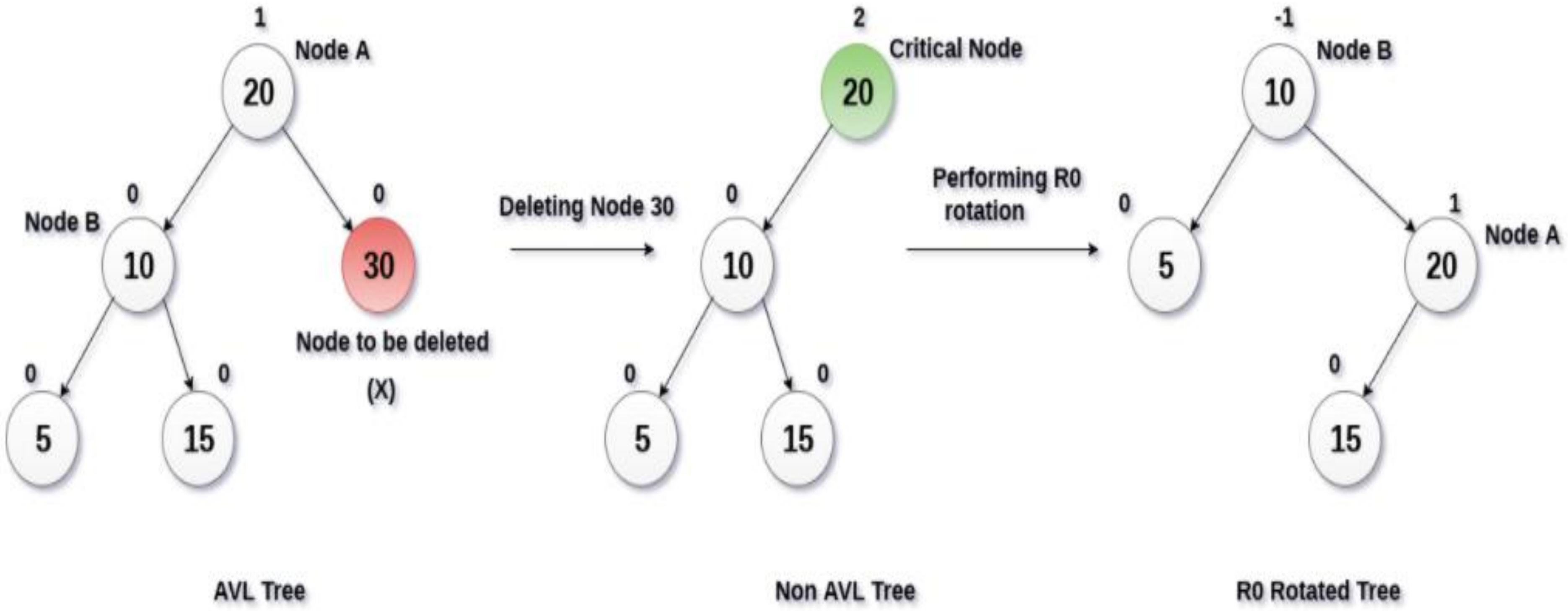
- Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation.(L rotations are mirror images of R Rotations)
- **R rotations are**

  1. R0 rotation (Node B has balance factor 0)



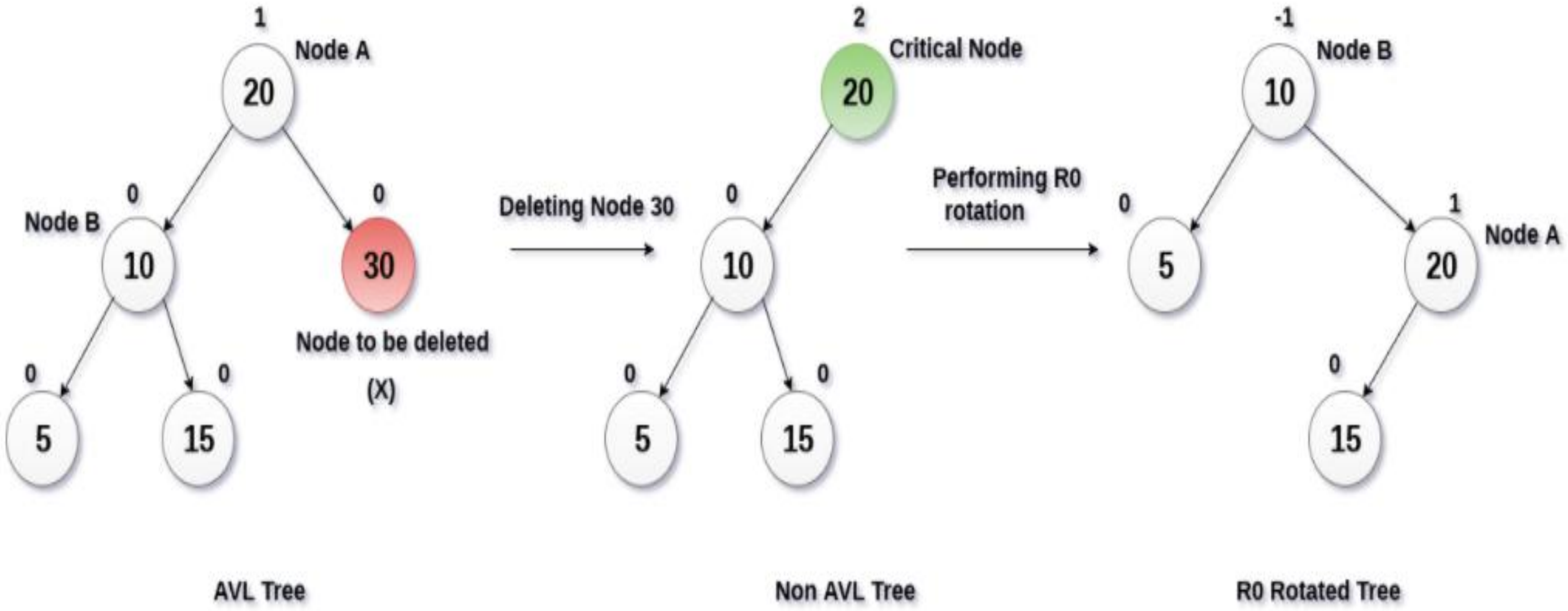
AVL Tree...

R0 Rotation Example



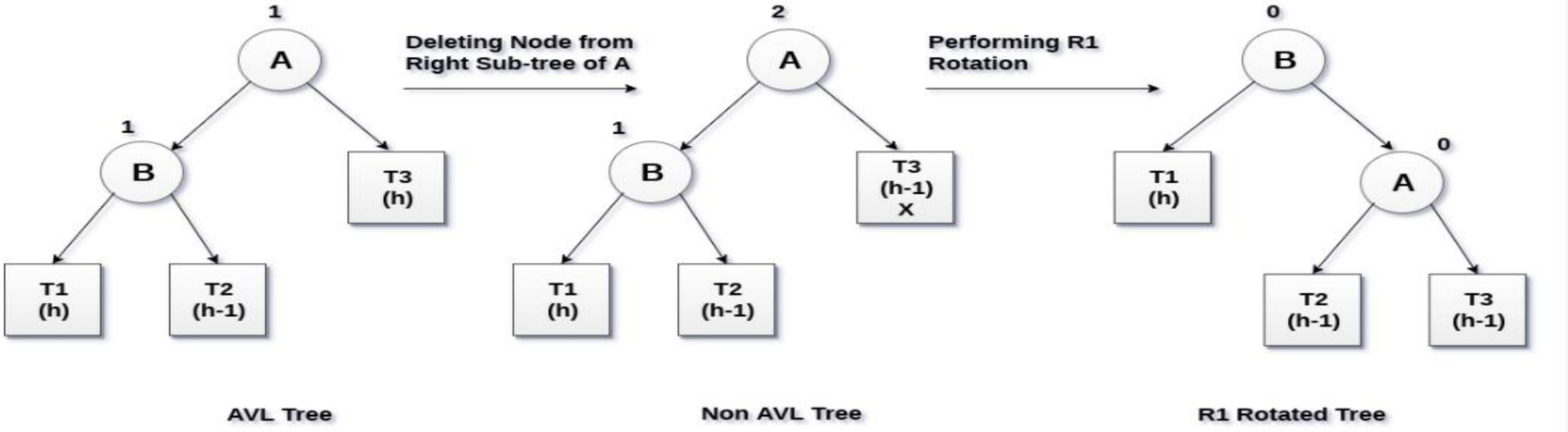
AVL Tree...

R0 Rotation Example

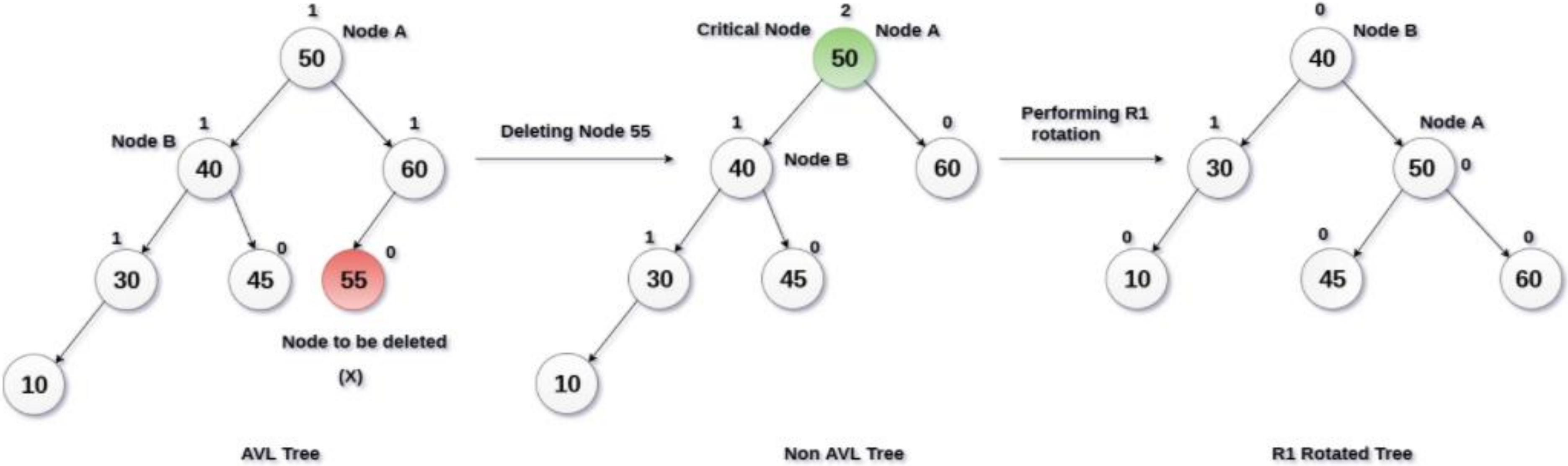


AVL Tree...

2. R1 Rotation (Node B has balance factor 1)

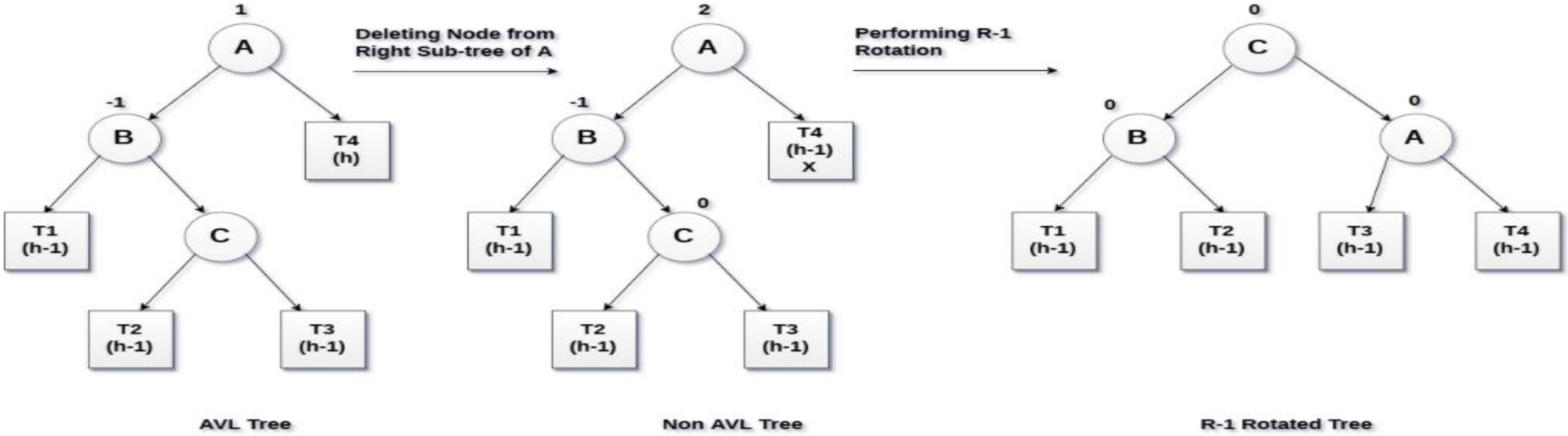


Example

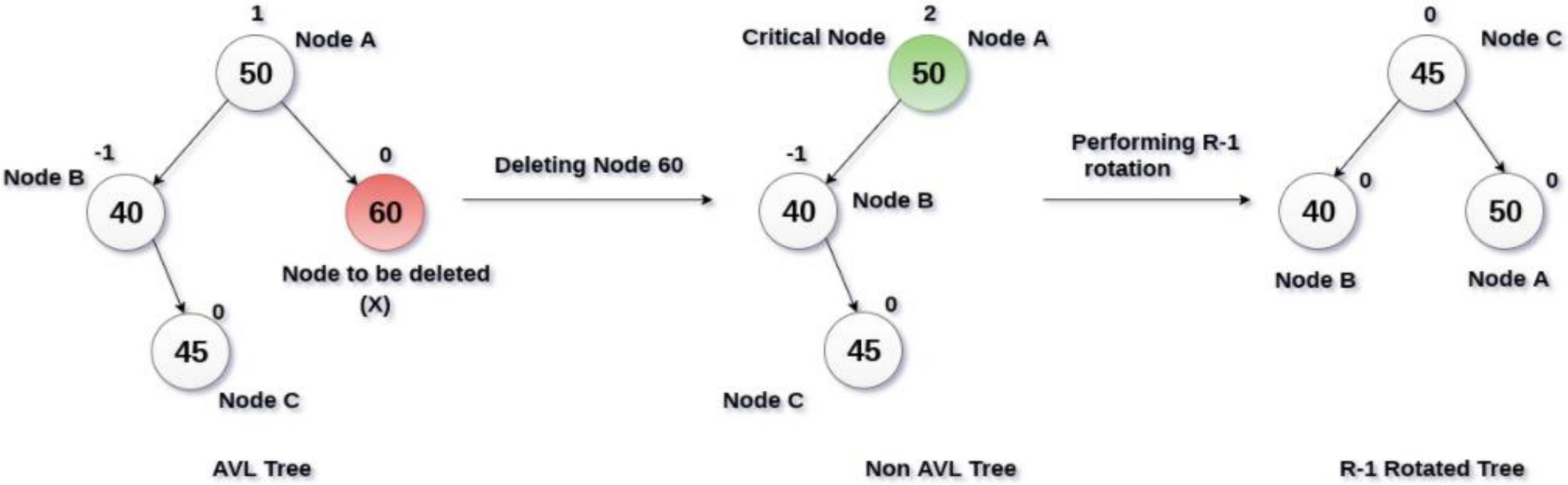


AVL Tree...

2. R-1 Rotation (Node B has balance factor -1)



Example:



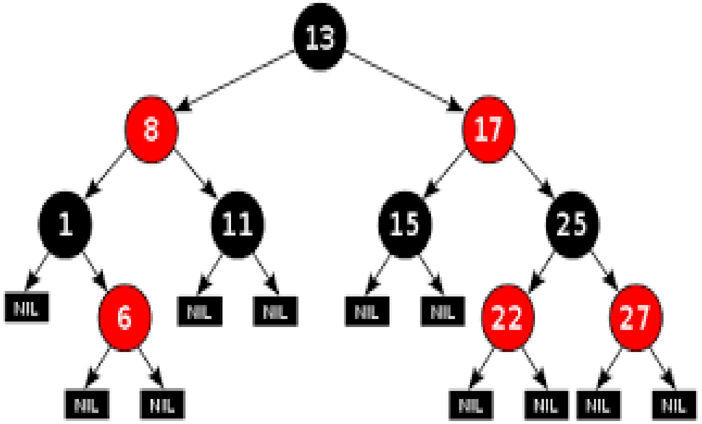
Red Black Tree

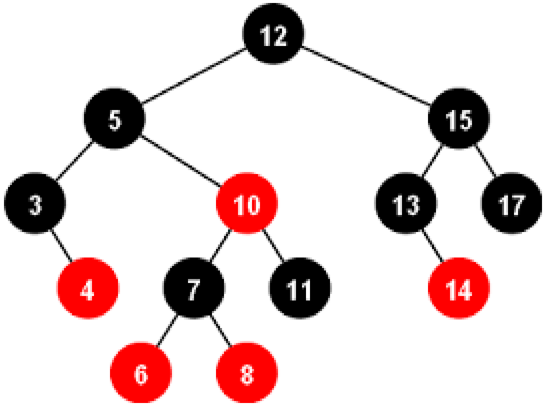
Red Black Tree:

- It is a self balanced Binary Search Tree in which every node is colored either RED or BLACK .
- In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

1. Red - Black Tree must be a Binary Search Tree.
2. The ROOT node must be colored BLACK.
3. The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
4. In all the paths of the tree, there should be same number of BLACK colored nodes.
5. Every new node must be inserted with RED color.
6. Every leaf (e.i. NULL node) must be colored BLACK.



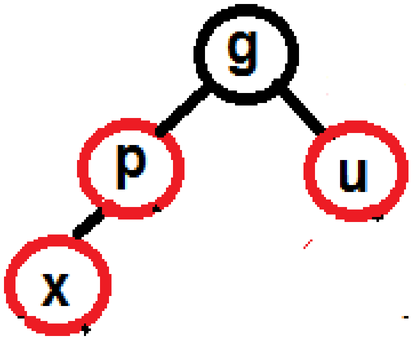


Red Black Tree...

Operations:

1.Insertion:

- Insertion in red black tree is same as BST.
- The color of inserted node is Red.



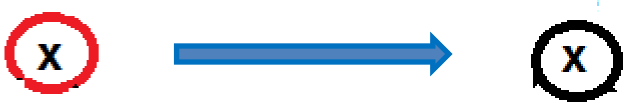
- In above Red Black tree g- Grand parent of node x, p is parent of x and u is uncle of x, where x is newly inserted node.
- In Red-Black tree, we use two tools to do balancing.
  1) Recoloring
  2) Rotation
- If uncle is red, we do recoloring.
- If uncle is black, we do rotations and/or recoloring.
- Color of a NULL node is considered as BLACK.

Red Black Tree...

Insertion Algorithm:

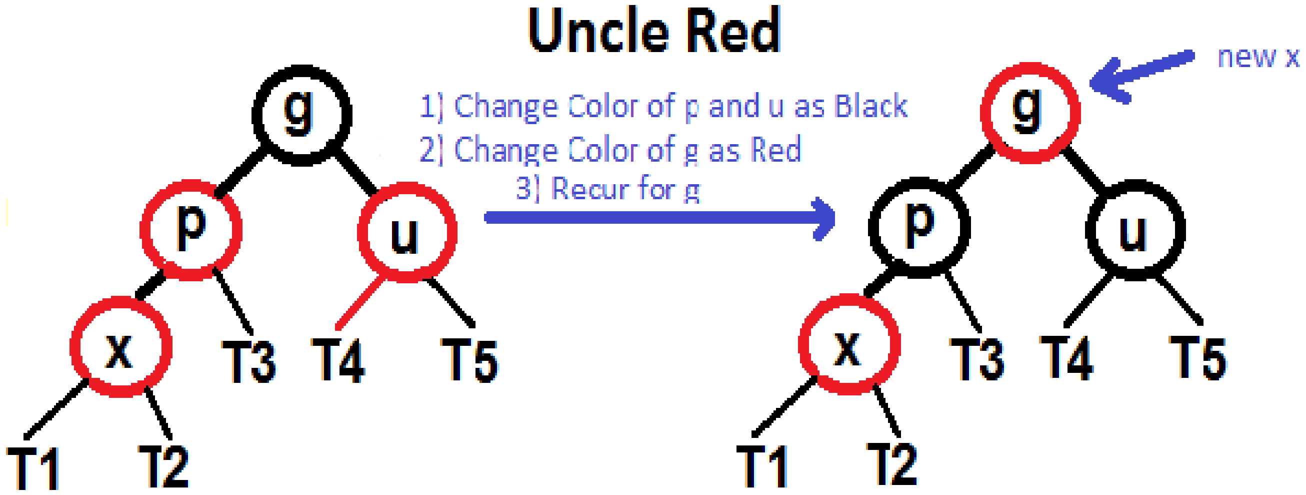
Let x be the newly inserted node, the color of newly inserted nodes as **RED .**

Case 1: If x is root, change color of x as **BLACK**



Case 2: If x’s uncle is **RED** (**Red Uncle Condition**)

1. Change color of parent and uncle as **BLACK**.
2. color of grand parent as **RED**.
3. Change x = x’s grandparent, repeat steps 2 and 3 for new x.



1] Change Color of p and u as Black
2] Change Color of g as Red
3] Recur for g

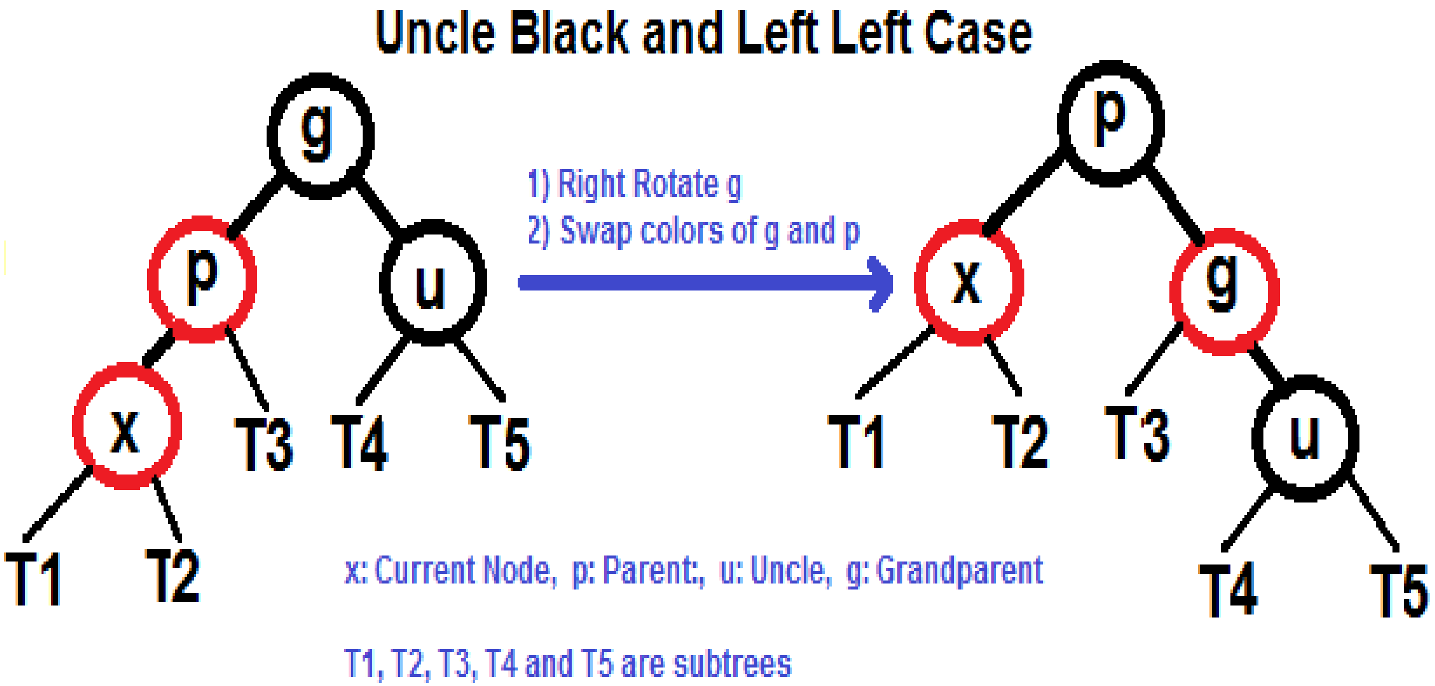
x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

Red Black Tree...

Case 3: If x’s uncle is BLACK or no uncle, The possible four configurations are

1. Left Left Case (p is left child of g and x is left child of p).
   1. Right Rotate Grand parent g
   2. Swap colors of g and p



Uncle Black and Left Left Case

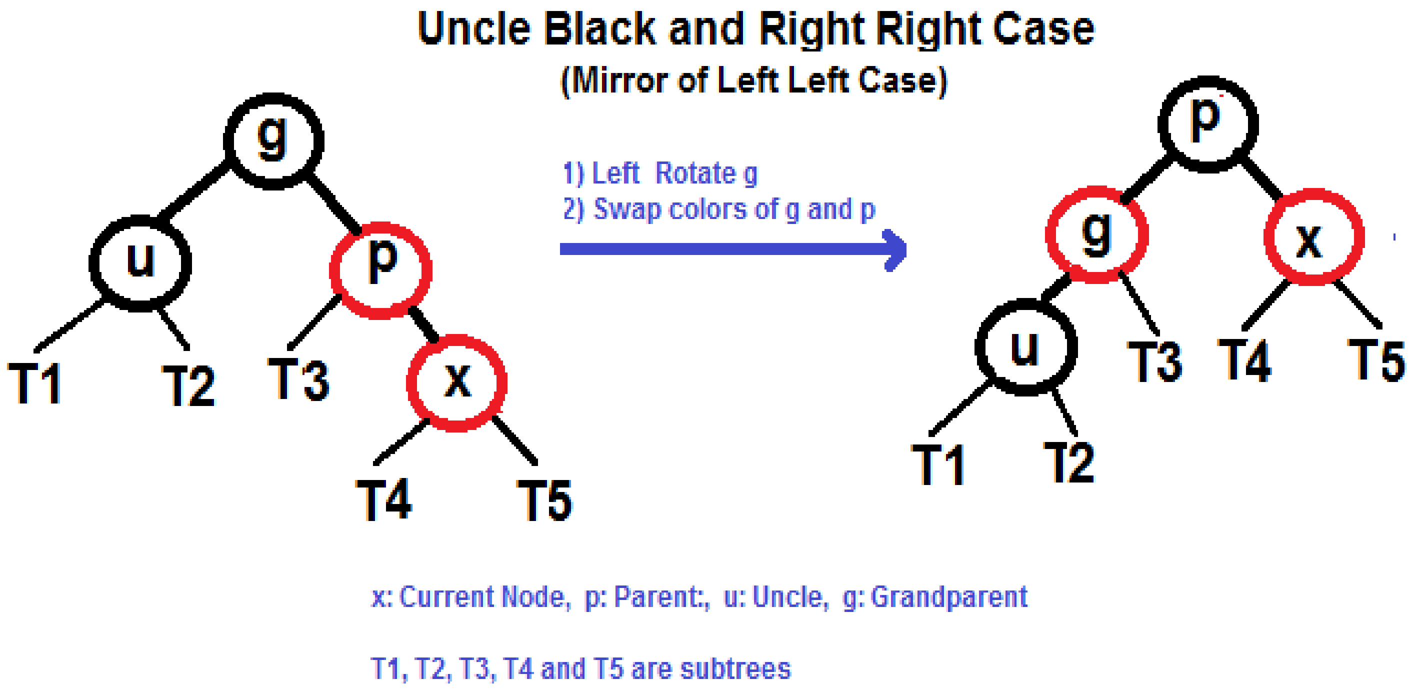
1) Right Rotate g
2) Swap colors of g and p

x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

Red Black Tree...

2. Right Right Case (p is right child of g and x is right child of p).
   i. Left Rotate Grand parent g
   ii. Swap colors of g and p



Uncle Black and Right Right Case

(Mirror of Left Left Case)

1) Left Rotate g

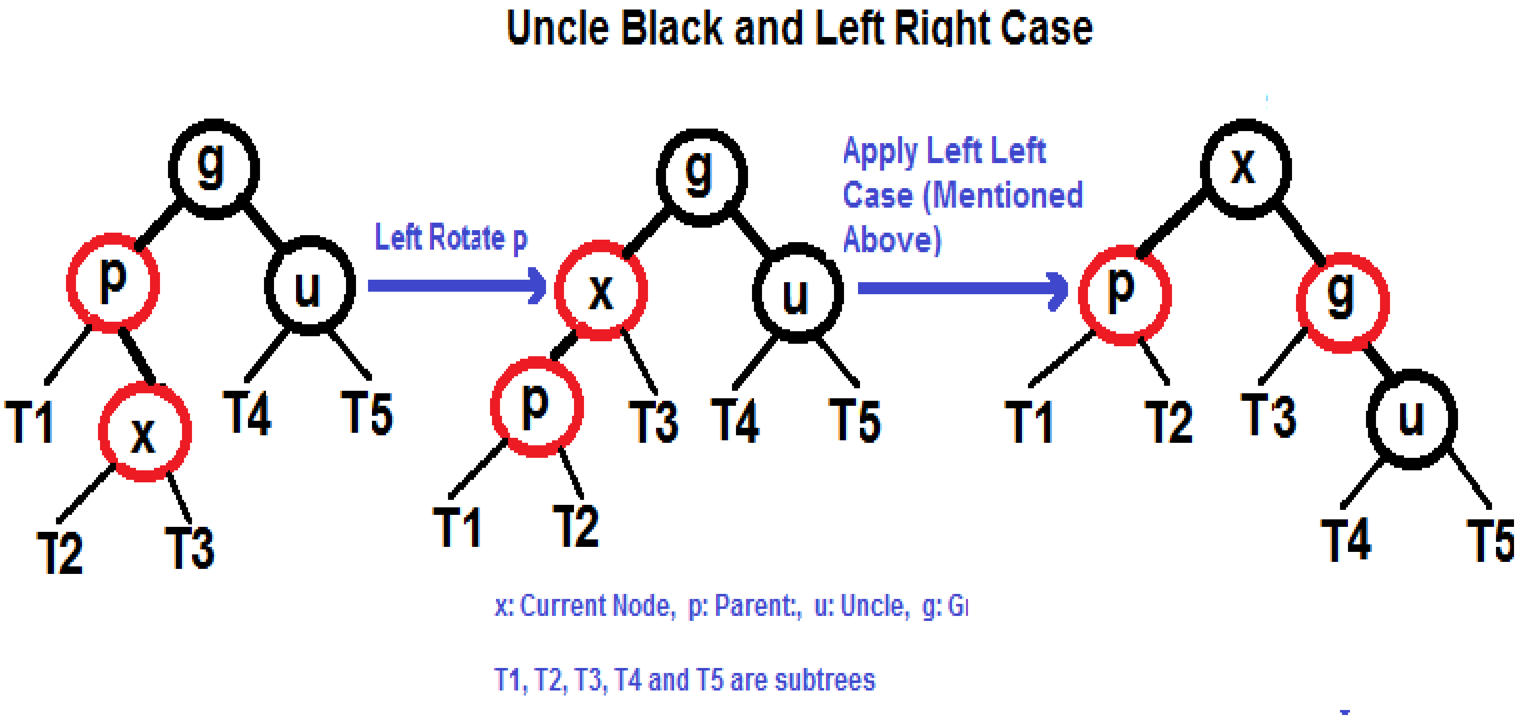
2) Swap colors of g and p

x: Current Node, p: Parent:, u: Uncle, g: Grandparent

T1, T2, T3, T4 and T5 are subtrees

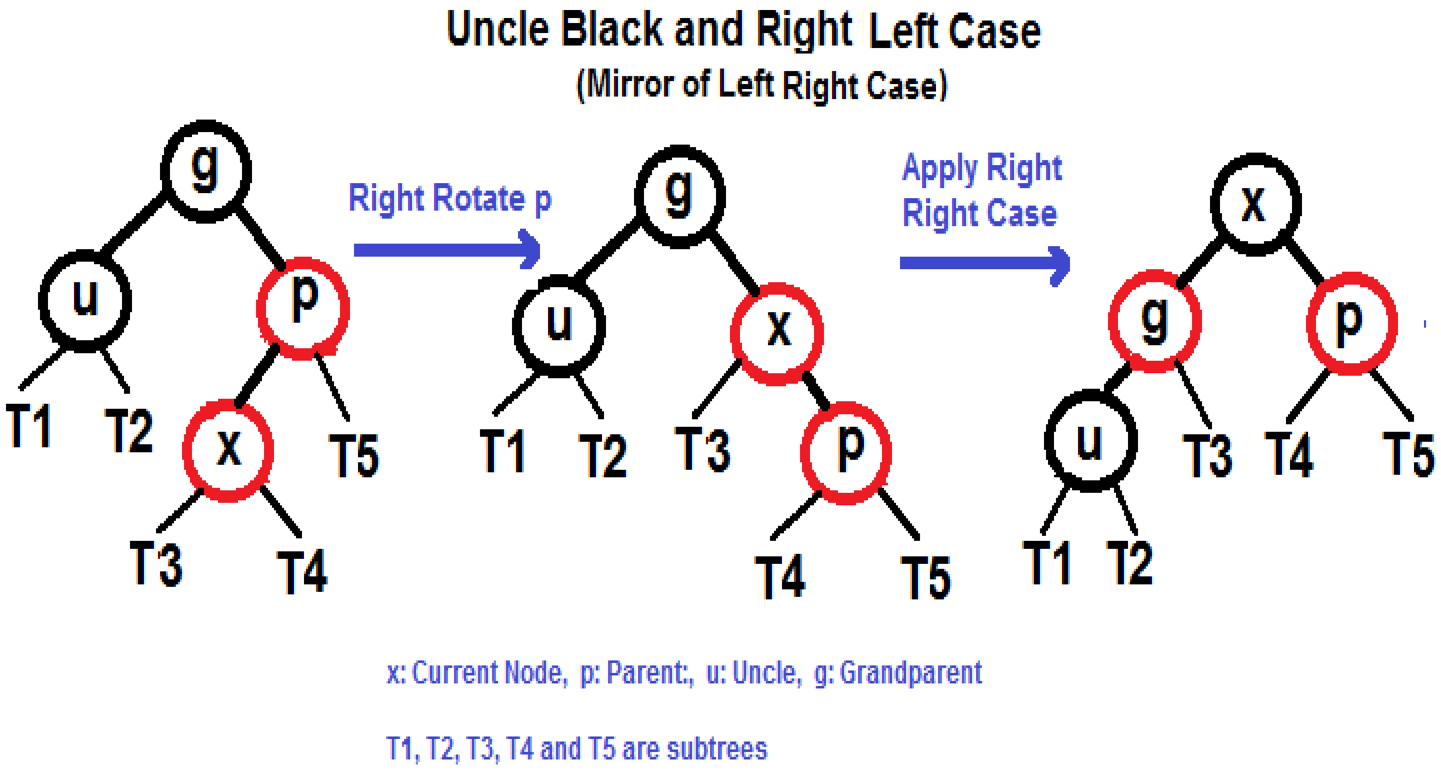
Red Black Tree...

3. Left Right Case (p is left child of g and x is right child of p).
   - i. Left Rotate p
   - ii. Apply Left-Left Case


Uncle Black and Left Right Case
g
p
u
T1
x
T4
T5
T2
T3
Left Rotate p
g
x
u
p
T3
T4
T5
T1
T2
Apply Left Left Case (Mentioned Above)
x
p
g
T1
T2
T3
u
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Gr
T1, T2, T3, T4 and T5 are subtrees

Red Black Tree...

4. Right Left Case (p is right child of g and x is left child of p).
    i. Right Rotate p
    ii. Apply Right-Right Case

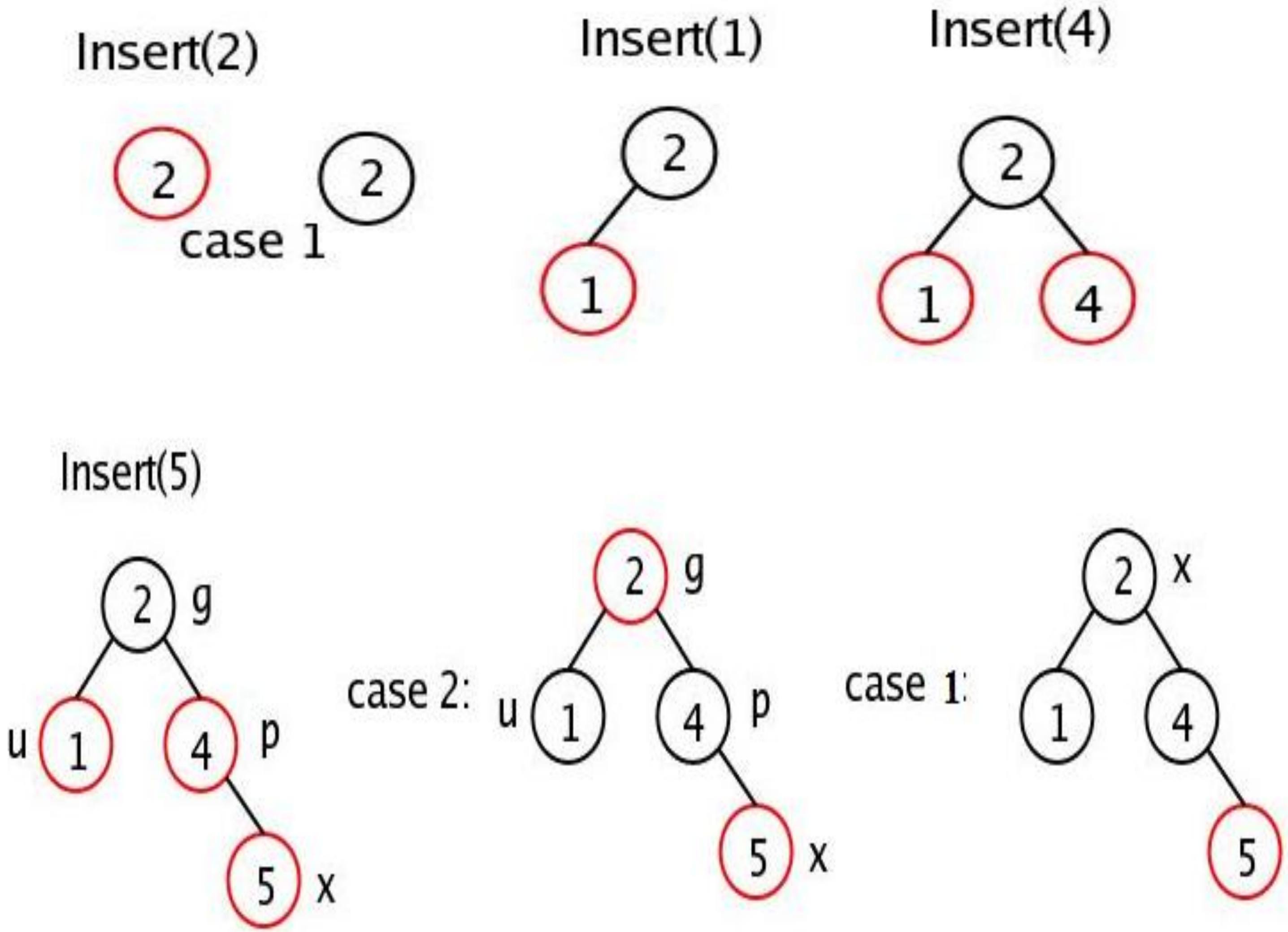


Uncle Black and Right Left Case
(Mirror of Left Right Case)
g
u
p
T1
T2
x
T5
T3
T4
Right Rotate p
g
u
x
T1
T2
T3
p
T4
T5
Apply Right Right Case
x
g
p
u
T3
T4
T5
T1
T2
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

Red Black Tree...

Problem: Insert the following elements into Red black tree

2, 1, 4, 5, 9, 3, 6, 7

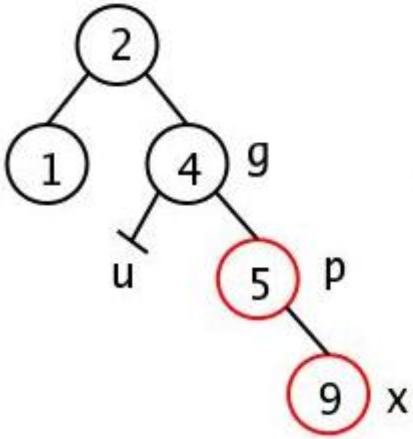


Red Black Tree...

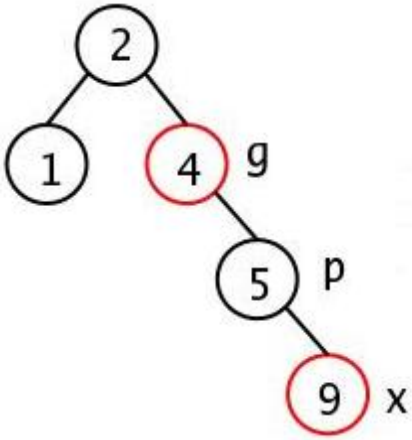
Problem: Insert the following elements into Red black tree

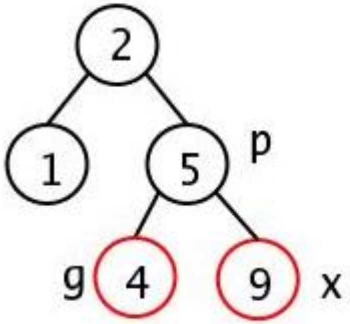
2, 1, 4, 5, 9, 3, 6, 7

Insert(9)

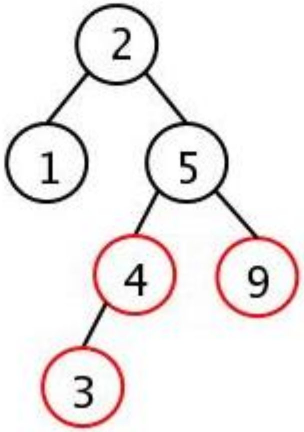


case 3(ii)

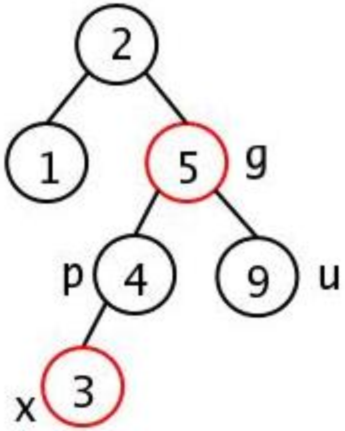




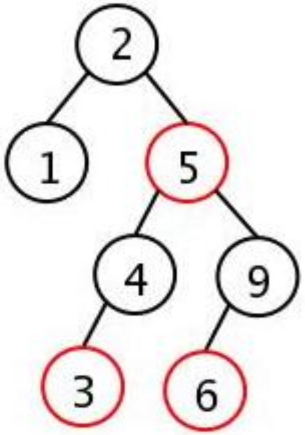
Insert(3)



case 2:



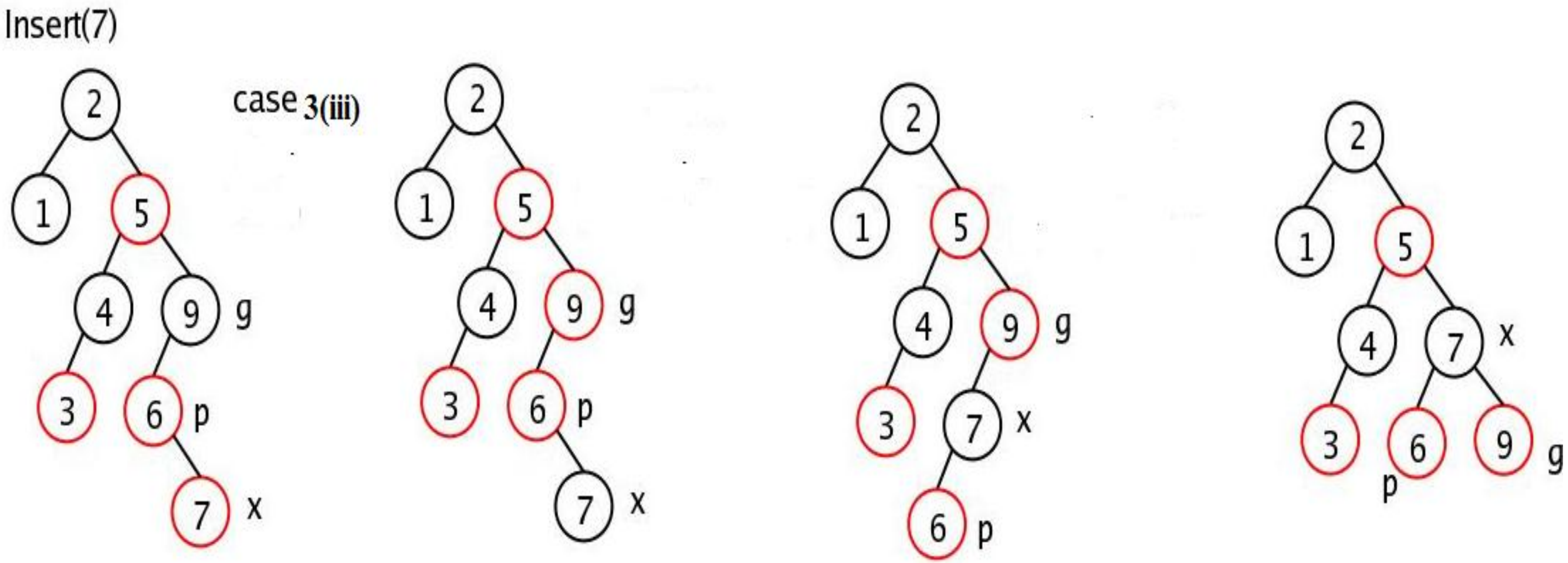
Insert(6)



Red Black Tree...

Problem: Insert the following elements into Red black tree

2, 1, 4, 5, 9, 3, 6, 7



Red Black Tree...

Deletion:

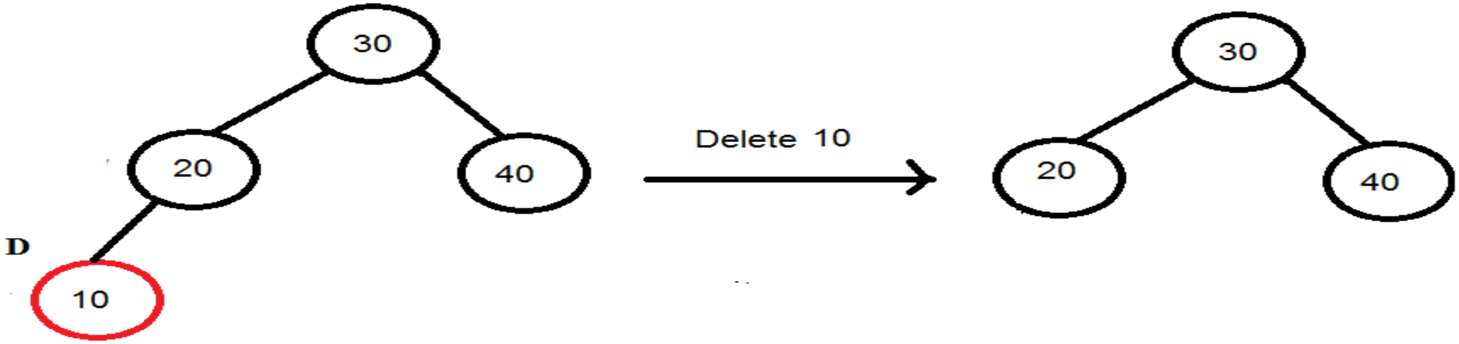
Important points to be remembered during deletion:

1. Red Black tree follows BST Deletion
2. In this only last nodes(leave nodes) are deleted ,Nodes with children are replaced.
3. In case node to be replaced has two children then replace it with closest predecessor from left sub tree.
4. In case node to be replaced has one child then replace it with its child.
5. In case node to be replaced has no child then replace it with NULL node.
6. If black node replaces black node then it will become double black.
7. If red node replaces black node then it will become single black.

Deletion Cases:

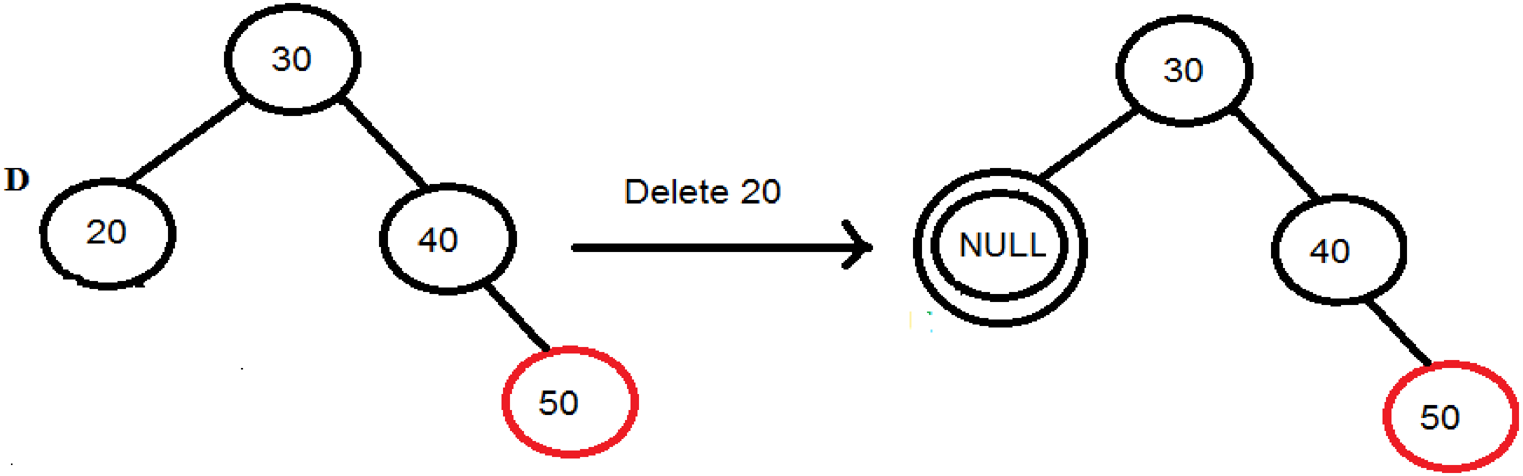
Cases 1:

i) if deleted node D is RED-Delete it directly and do nothing.



Red Black Tree...

ii) if deleted node D is not RED-then Double Black DB Exists.



Case 2: If DB is a root node –then remove DB directly and make it single black.

Case 3:

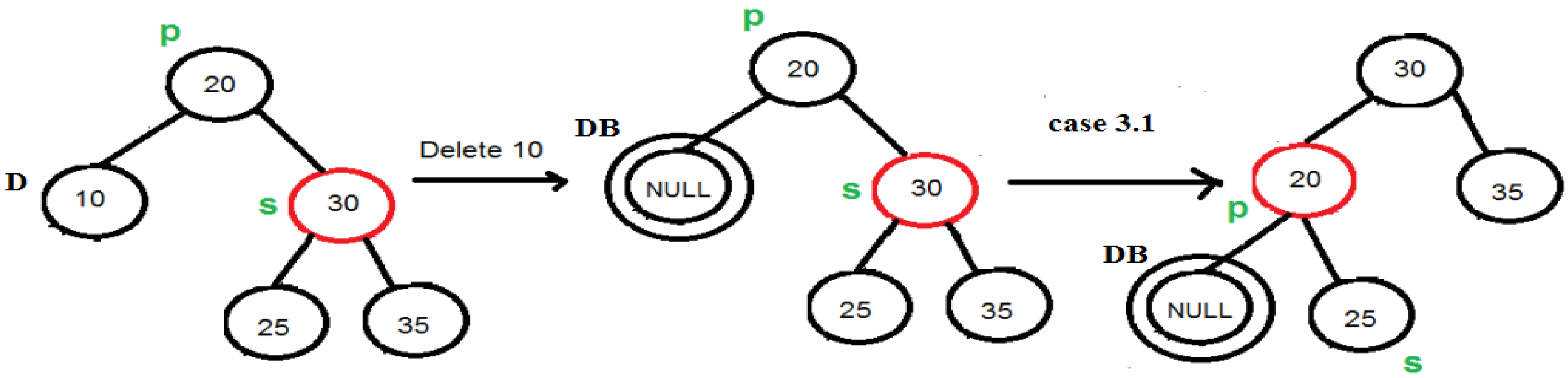
3.1 if DB's sibling is red

i. Rotate P in DB direction

ii. Swap colors of P and S

iii. Reapply cases if needed.

Red Black Tree...



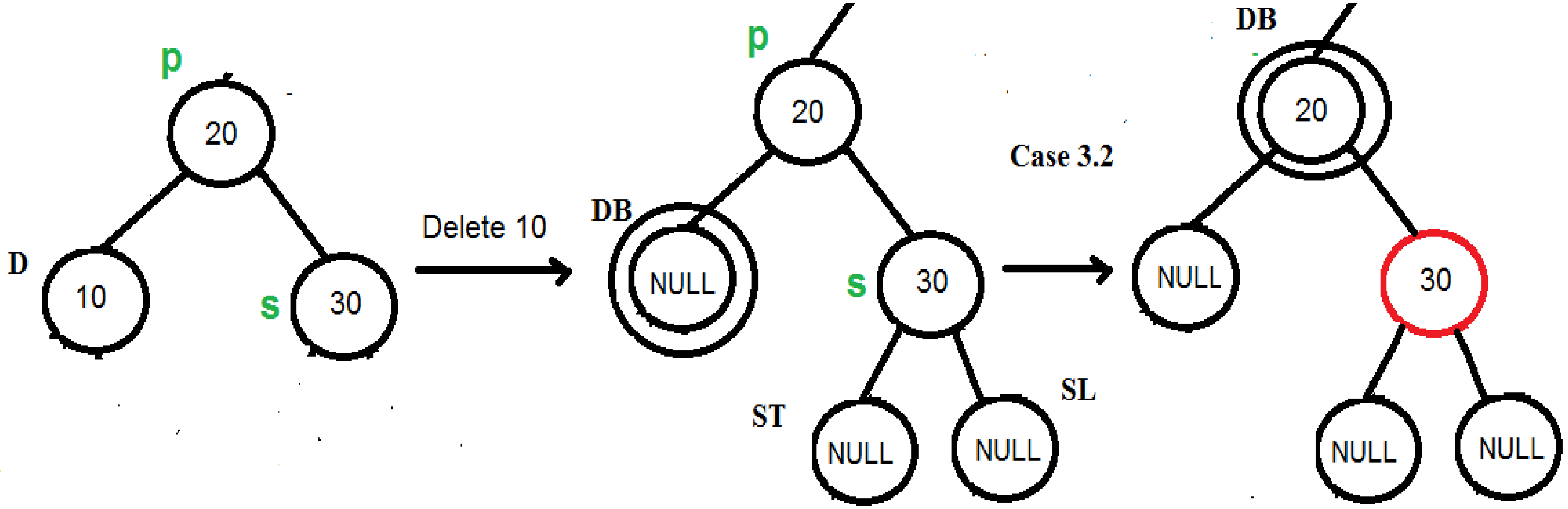
3.2 if DB's sibling is black and both children are black(SL and ST are black)

i. Remove DB

ii. Add black to P:if P is black make DB ,if P is red make single black.

iii. Make S as RED

iv. DB still exist ,follow other cases.



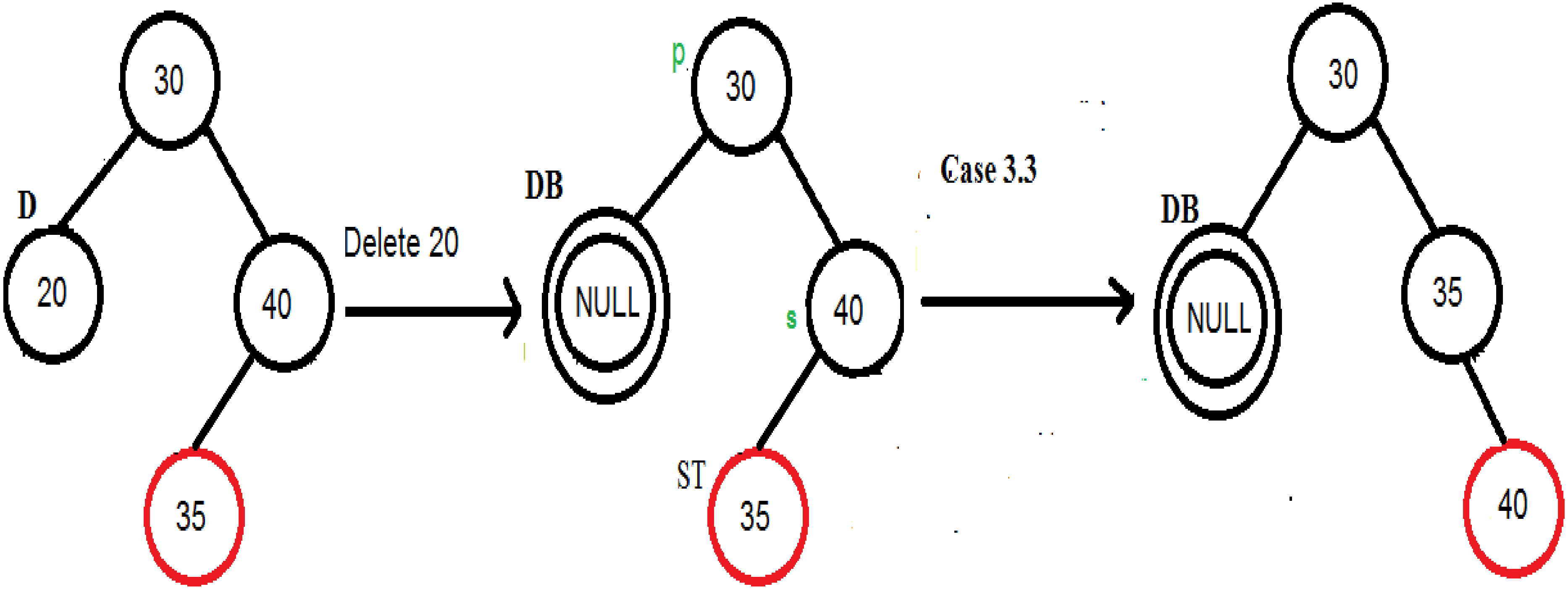
Red Black Tree...

3.3 if DB's sibling is black ,inline child SL is black and ST is RED

i. Swap colors of S and ST

ii. Rotate S in opposite direction to DB

iii. Follow case 3.4



Red Black Tree...

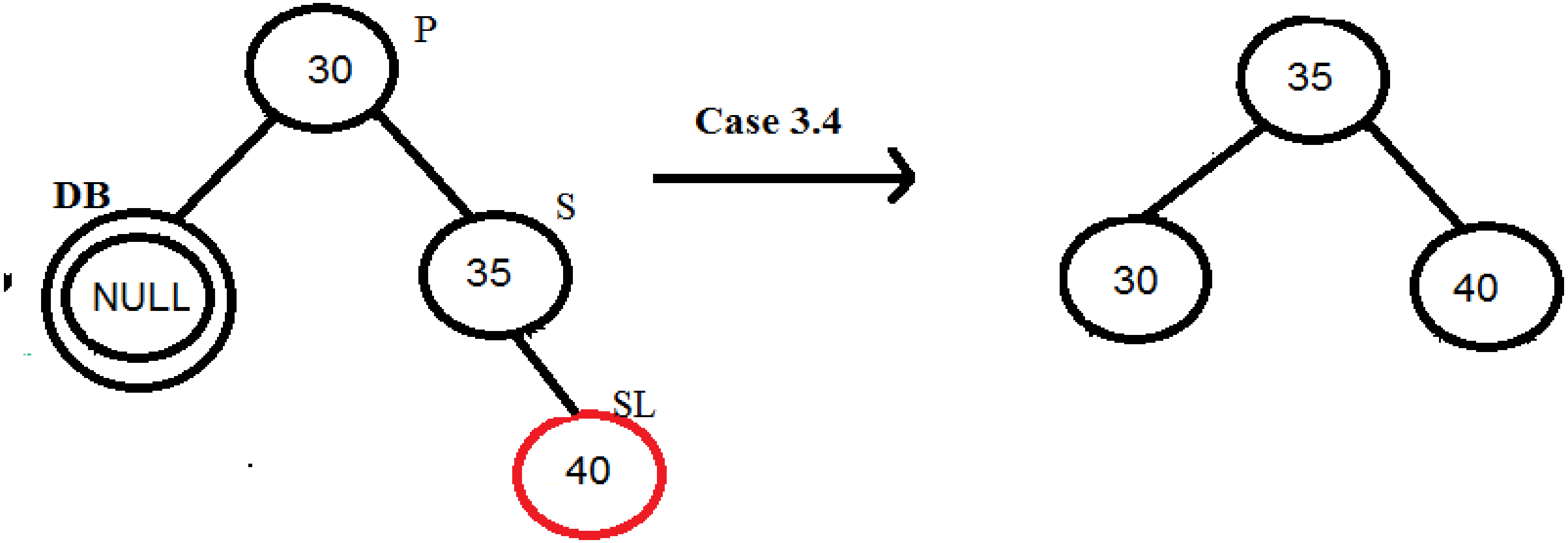
3.4 if DB’s sibling is black ,inline child SL is RED

i. Swap colors of P and S

ii. Rotate P in same direction to DB

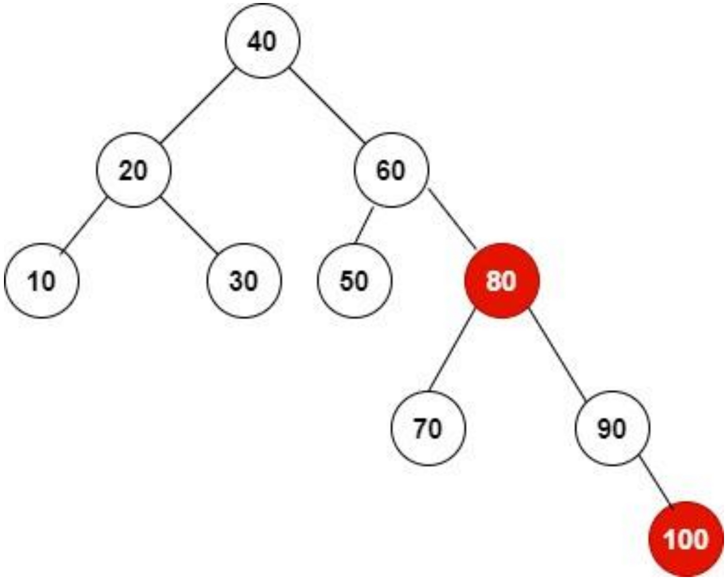
iii. Remove BD

iv. Change color of SL to Black

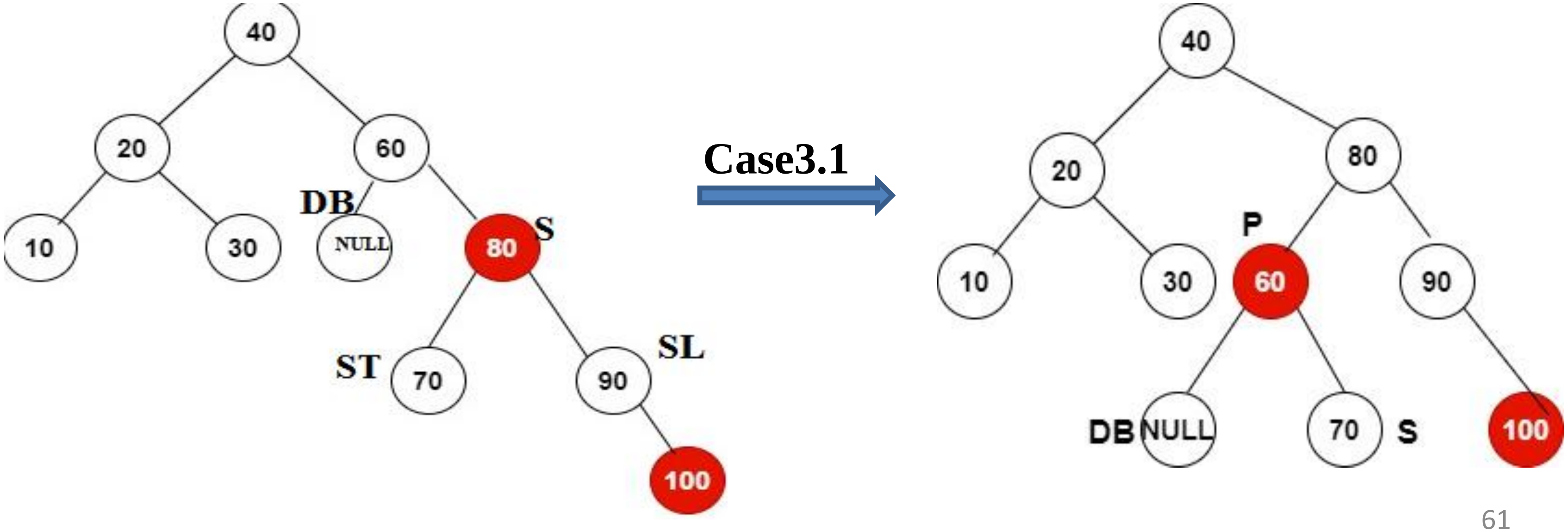


Red Black Tree...

Deletion Example: Delete 50,20,100,90,40,60,10,30,70,80 from the following RB Tree

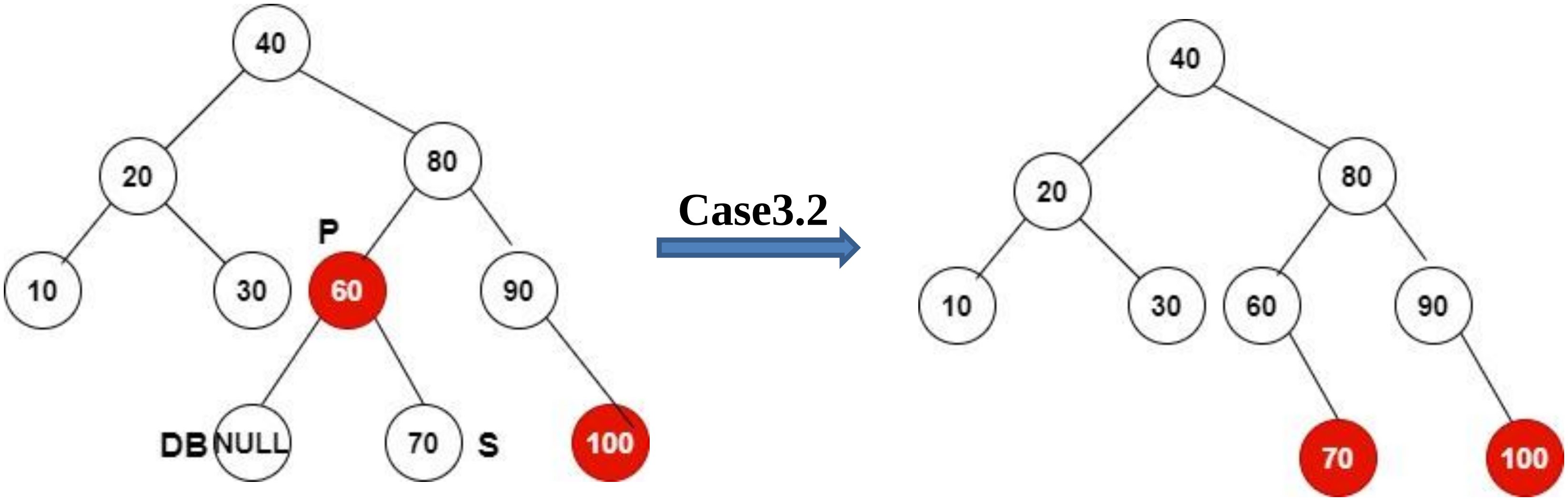


Delete 50



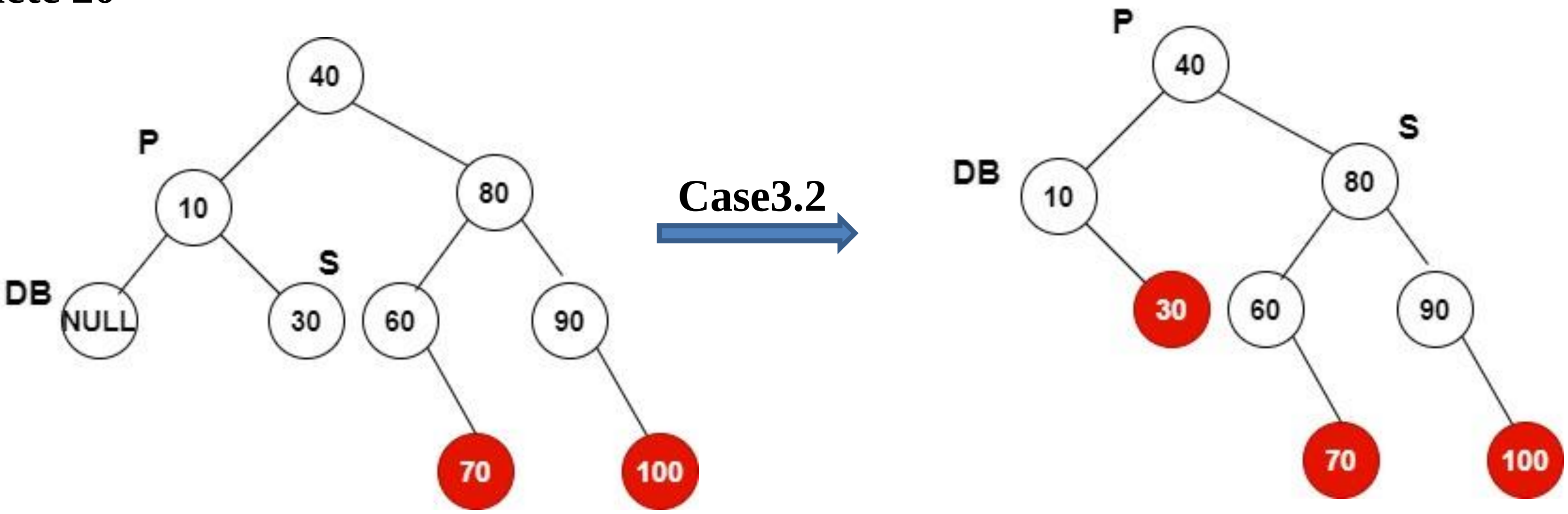
Red Black Tree...

Deletion Example: Delete 50,20,100,90 from the following RB Tree



Case3.2

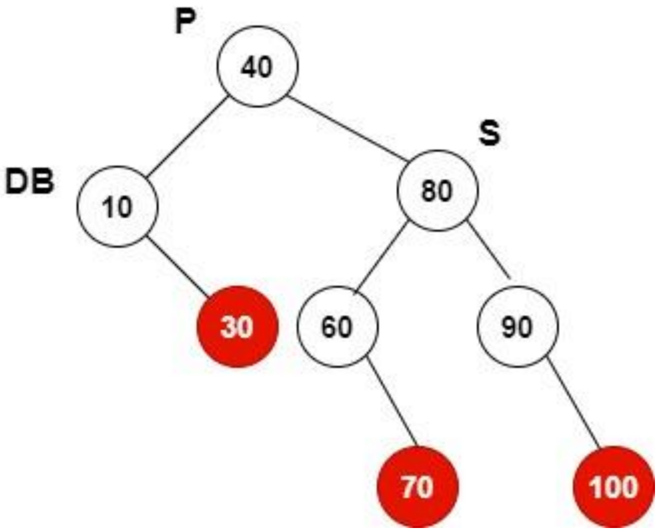
Delete 20



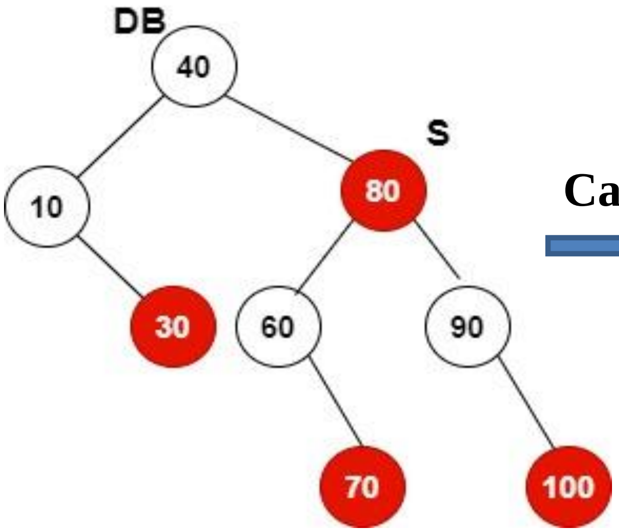
Case3.2

Red Black Tree...

Deletion Example: Delete 50,20,100,90 from the following RB Tree

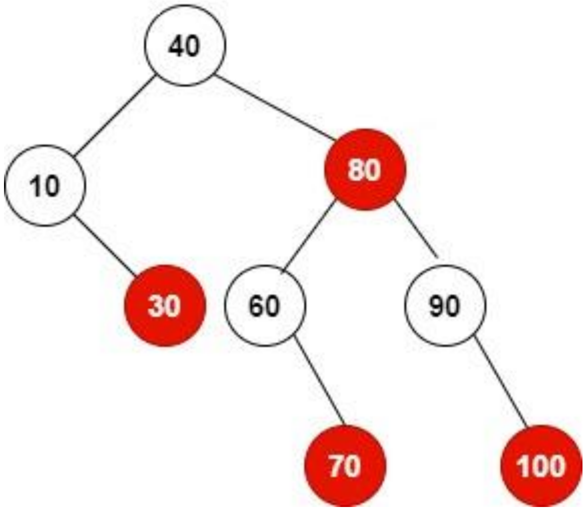


Case3.2

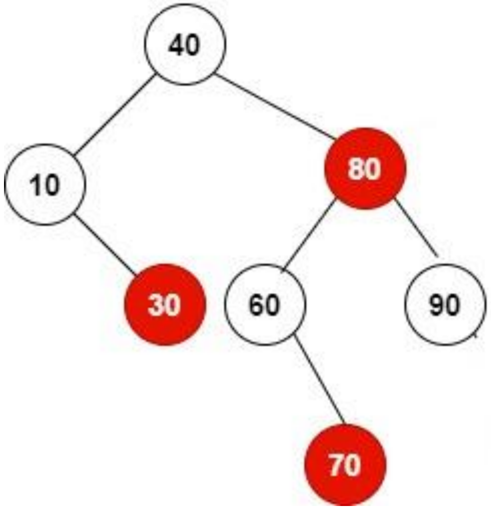


Case 2

Delete 100



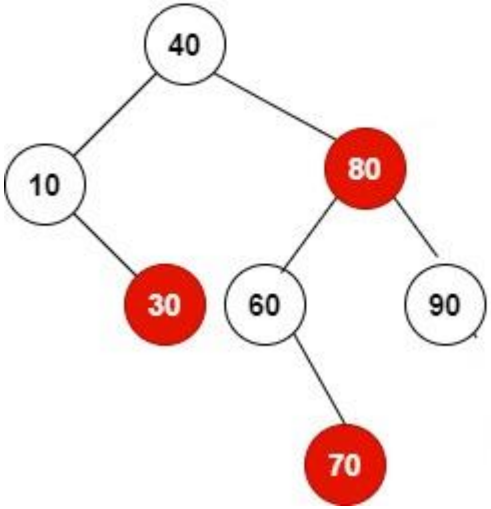
Case 1

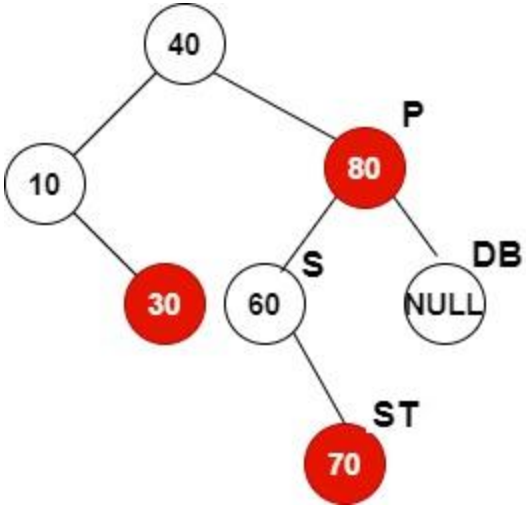


Red Black Tree...

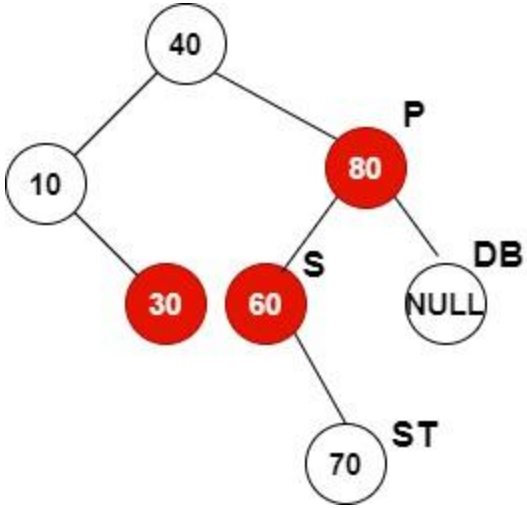
Deletion Example: Delete 50,20,100,90 from the following RB Tree

Delete 90

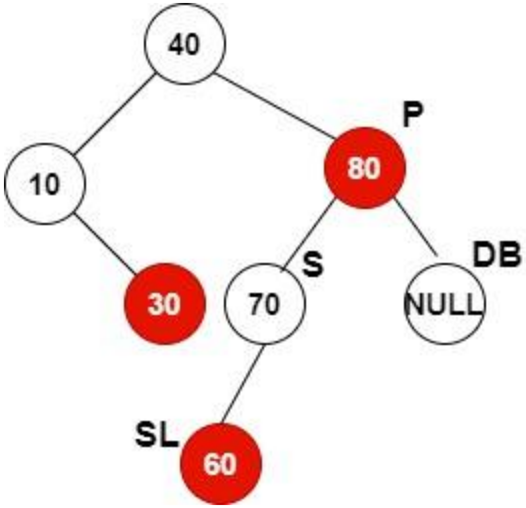




Case 3.3 -swap color of S and ST

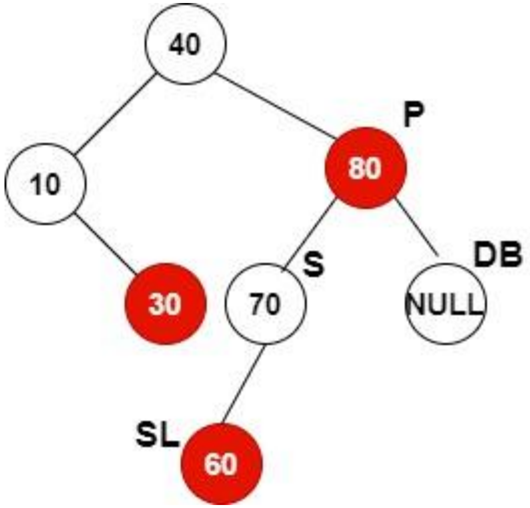


Case 3.3 –Rotate S in Opp. to DB

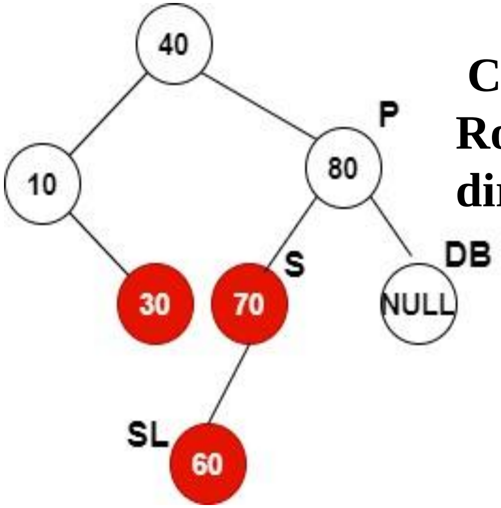


Red Black Tree...

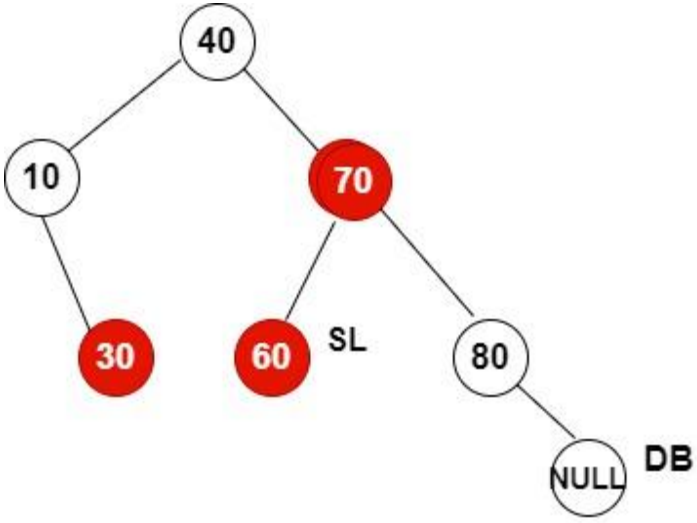
Deletion Example: Delete 50,20,100,90 from the following RB Tree



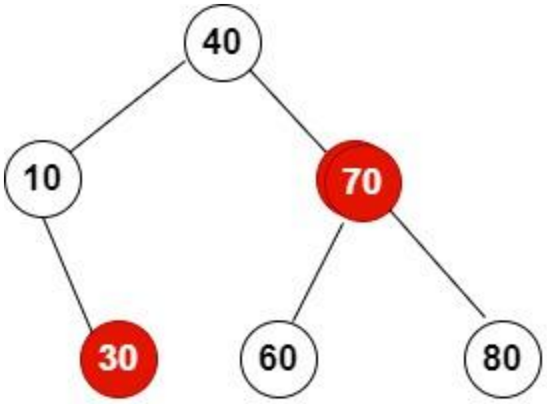
Case 3.4 -swap color of S and P



Case 3.4 – Rotate P in same direction of DB



Case 3.4 –Remove DB and change color of SL to black

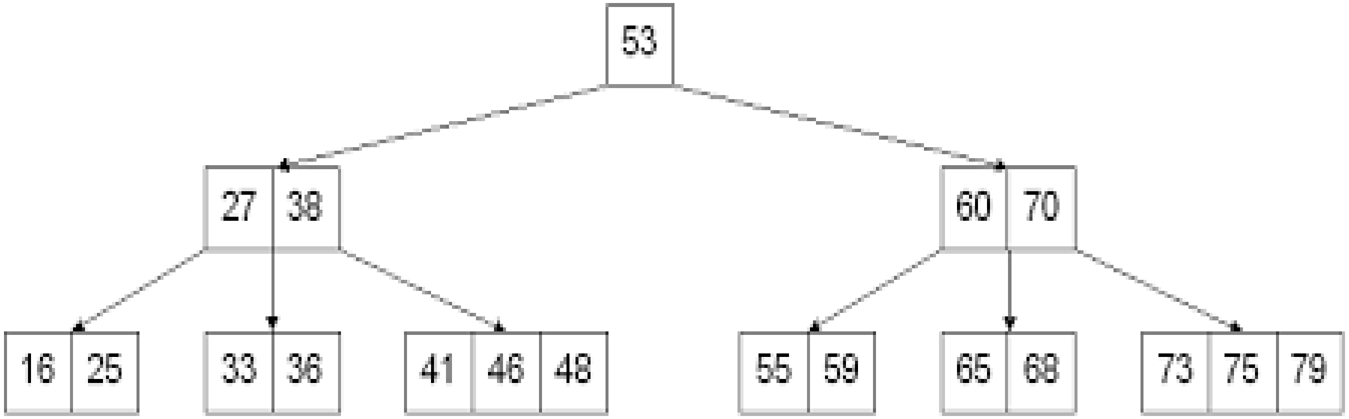


B Tree

- B-Tree is a M-way search tree that can be widely used for disk access.
- One of the main reason of using B tree is its capability to store large number of keys in a single node by keeping the height of the tree relatively small.

Properties of B-Tree:

1. B-Tree of order M will have atmost M children.
2. Every node in a B-tree except root and the leaf nodes has at least ⌈M/2⌉ children.
3. All leaves are at the same level.
4. All nodes may contain maximum M – 1 keys and minimum of ⌈M/2⌉ -1 keys.
5. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
6. The root has at least two children if it is not a leaf node.



B Tree...

Operations :

- **Searching:** Searching in B-tree is similar to BST

- **Insertion:**

  In a B-Tree, a new element must be added only at the leaf node. That means, the new key value is always attached to the leaf node only. The insertion operation is performed as follows...

  Step 1 - Check whether tree is Empty.

  Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

  Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

  Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

  Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

  Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

B Tree...

- Insert the following elements into a B-tree of order 3

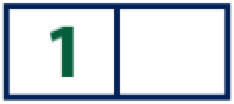
  1,2,3,4,5,6,7,8,9,10

Max no. of keys =M-1= 3-1=2

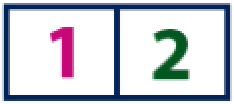
Min no. of keys = [M/2] -1 =3/2-1=2-1=1

Min no. of children = [M/2] =3/2=2

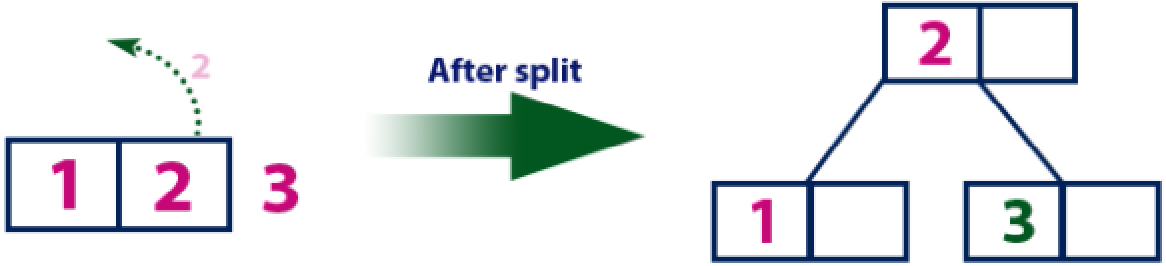
Insert 1



Insert 2



Insert 3



B Tree...

- Insert the following elements into a B-tree of order 3

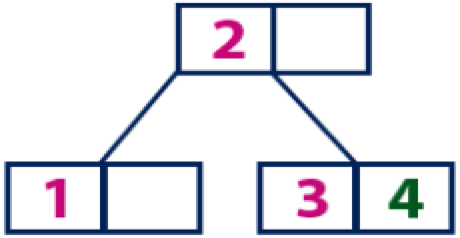
  1,2,3,4,5,6,7,8,9,10

Max no. of keys =M-1= 3-1=2

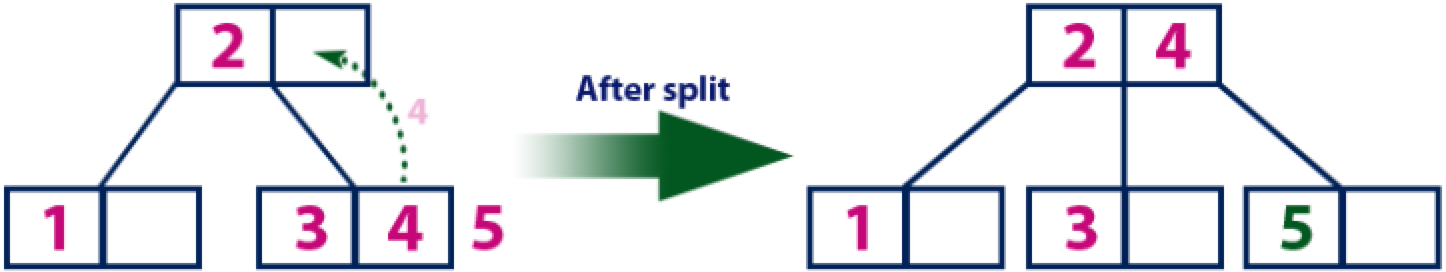
Min no. of keys = [M/2] -1 =3/2-1=2-1=1

Min no. of children = [M/2] =3/2=2

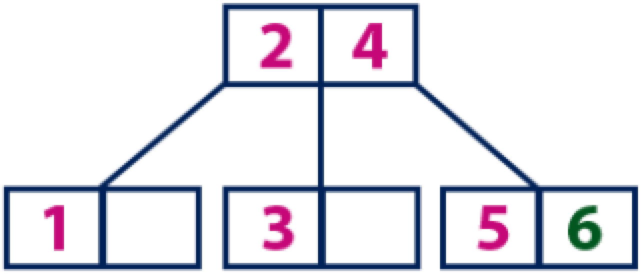
Insert 4



Insert 5



Insert 6



B- Tree...

- Insert the following elements into a B-tree of order 3

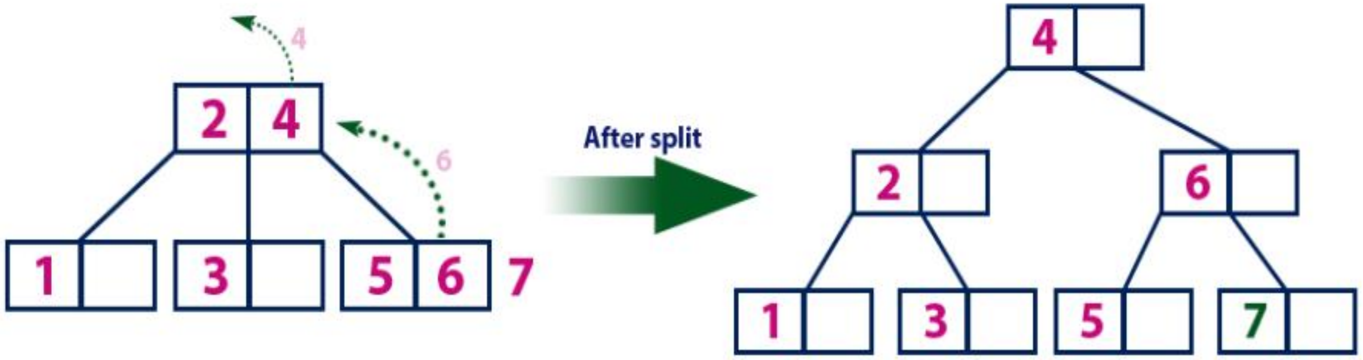
  1,2,3,4,5,6,7,8,9,10

Max no. of keys =M-1= 3-1=2

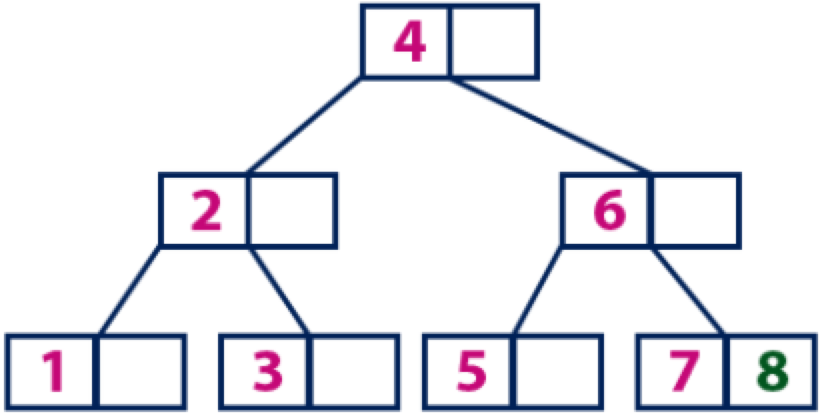
Min no. of keys = [M/2] -1 =3/2-1=2-1=1

Min no. of children = [M/2] =3/2=2

Insert 7



Insert 8



B Tree...

- Insert the following elements into a B-tree of order 3

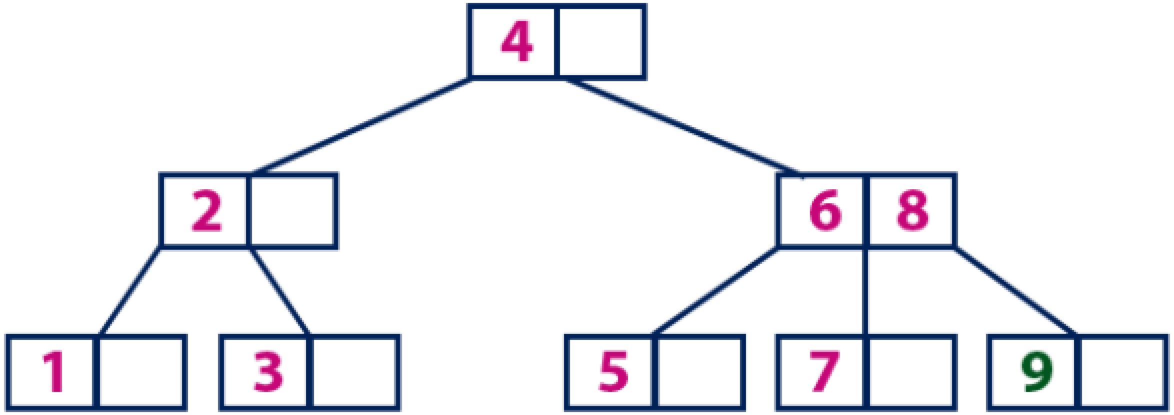
  1,2,3,4,5,6,7,8,9,10

Max no. of keys =M-1= 3-1=2

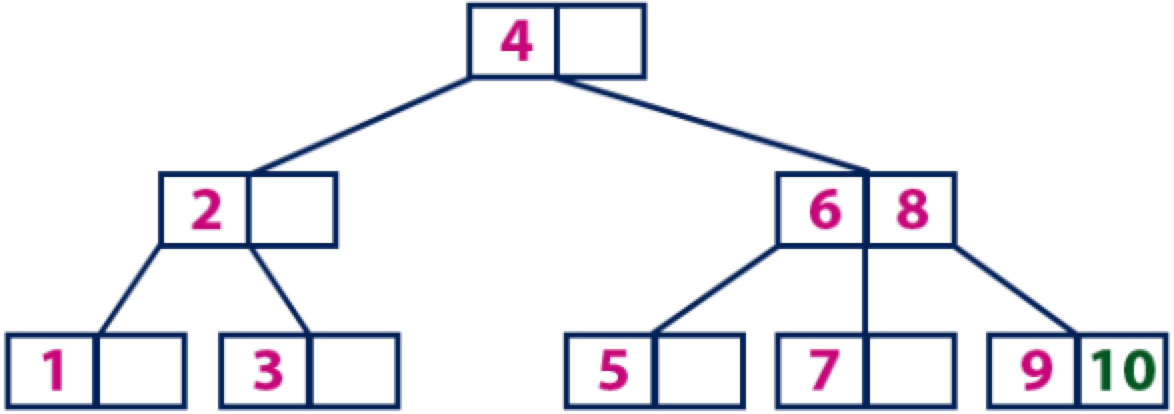
Min no. of keys = [M/2] -1 =3/2-1=2-1=1

Min no. of children = [M/2] =3/2=2

Insert 9



Insert 10



B Tree...

- ➢ Insert the following elements into a B-tree of order 3

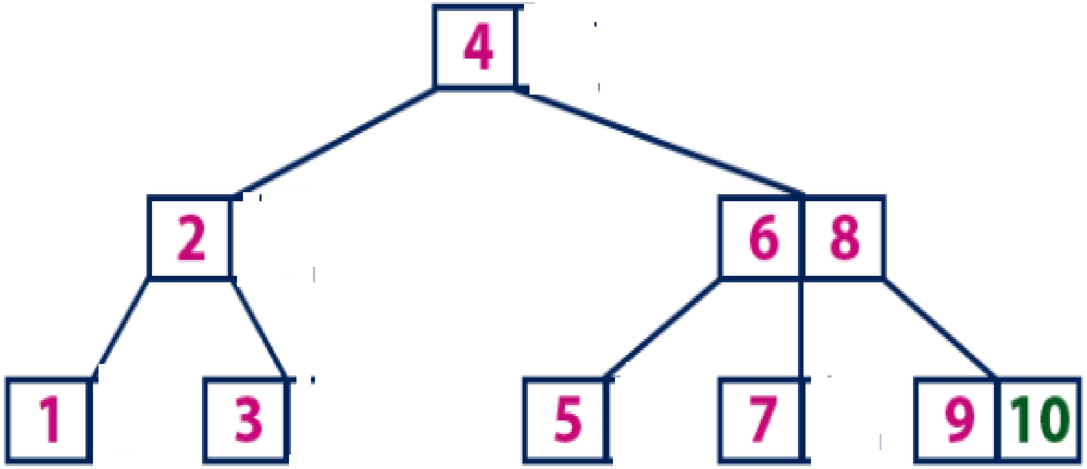
1,2,3,4,5,6,7,8,9,10

Max no. of keys =M-1= 3-1=2

Min no. of keys = ⌈M/2⌉ -1 =3/2-1=2-1=1

Min no. of children = ⌈M/2⌉ =3/2=2

Finally, B-tree is

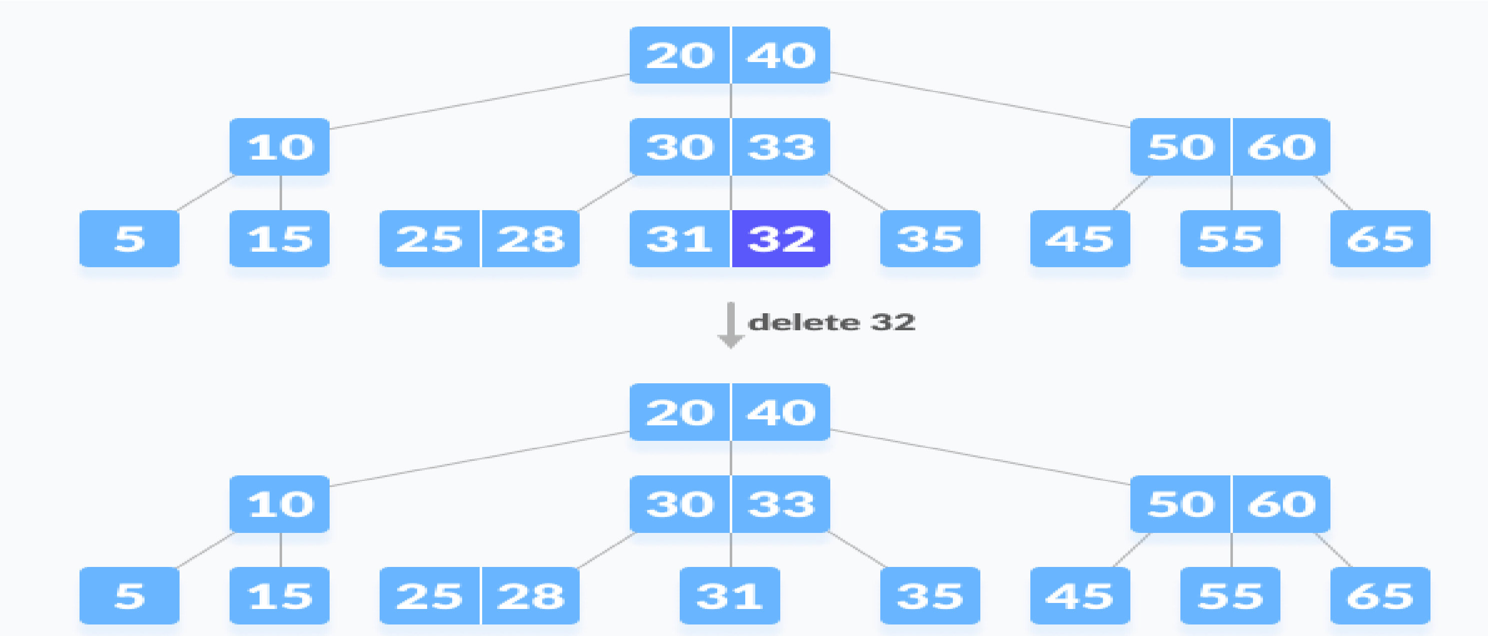


B Tree...

- **Deletion:**
  - Deleting an element on a B-tree consists of three main events: **searching** the node where the key to be deleted exists, **deleting** the key and **balancing** the tree if required.
  - While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.
  - There are three main cases for deletion operation in a B tree.

Case I:The key to be deleted lies in the leaf. There are two cases for it.

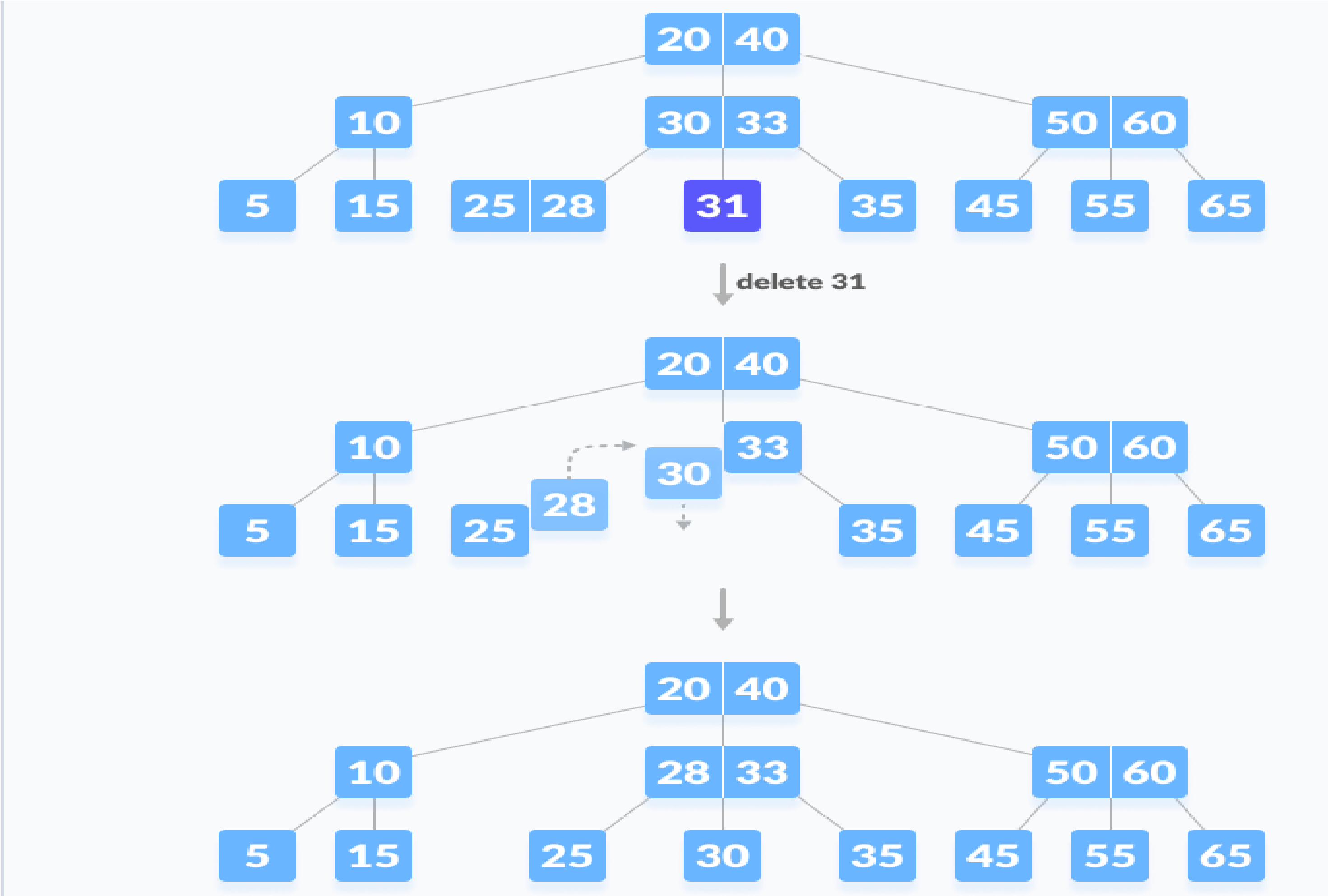
i) The deletion of the key does not violate the property of the minimum number of keys.



B Tree...

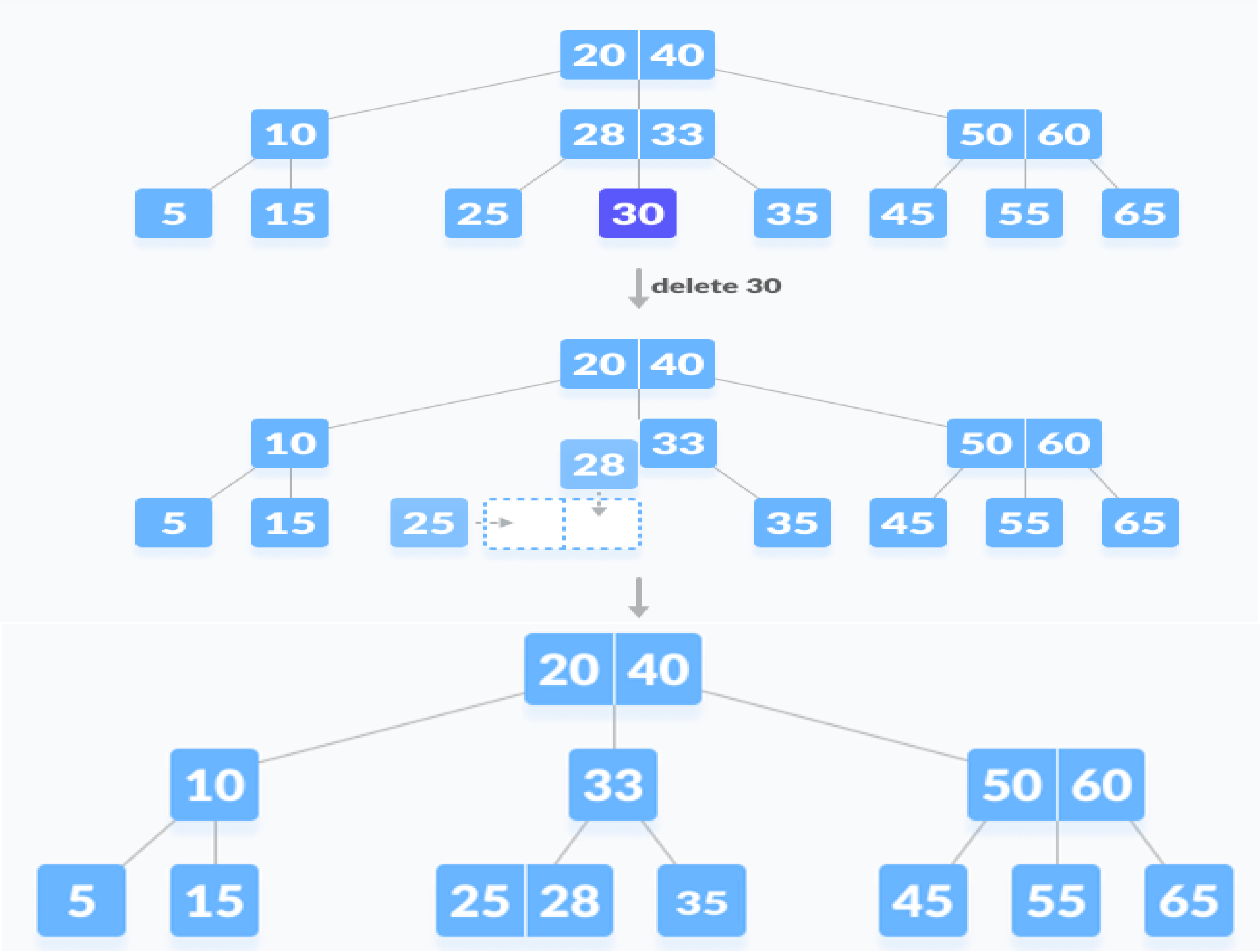
ii) The deletion of the key violates the property of the minimum number of keys.

In this case, we borrow a key from its immediate neighbouring sibling node in the order of left to right. First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node. Else, check to borrow from the immediate right sibling node



B Tree...

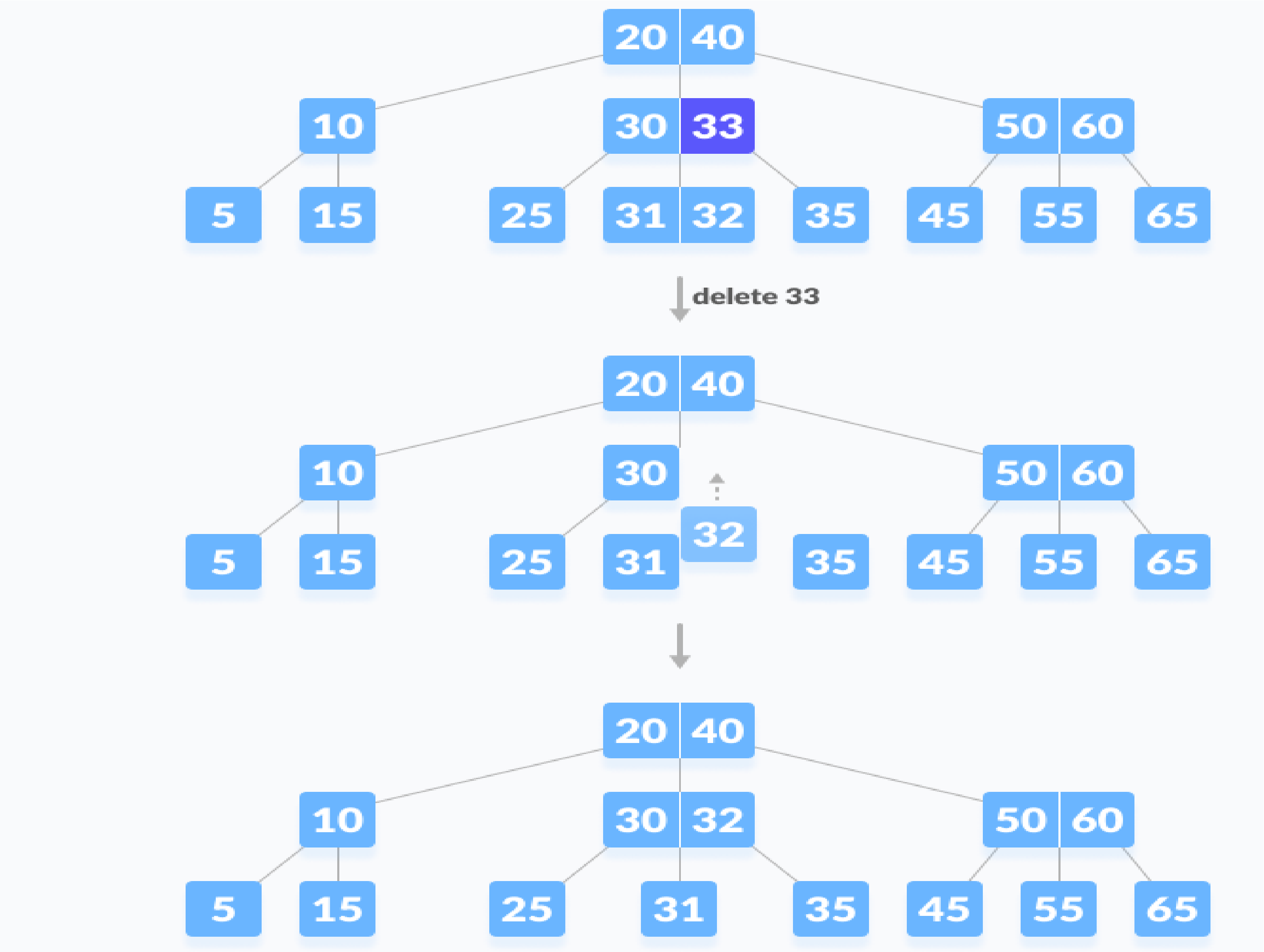
iii) If both the immediate sibling nodes already have a minimum number of keys: then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.



B Tree...

Case II :If the key to be deleted lies in the internal node, the following cases occur.

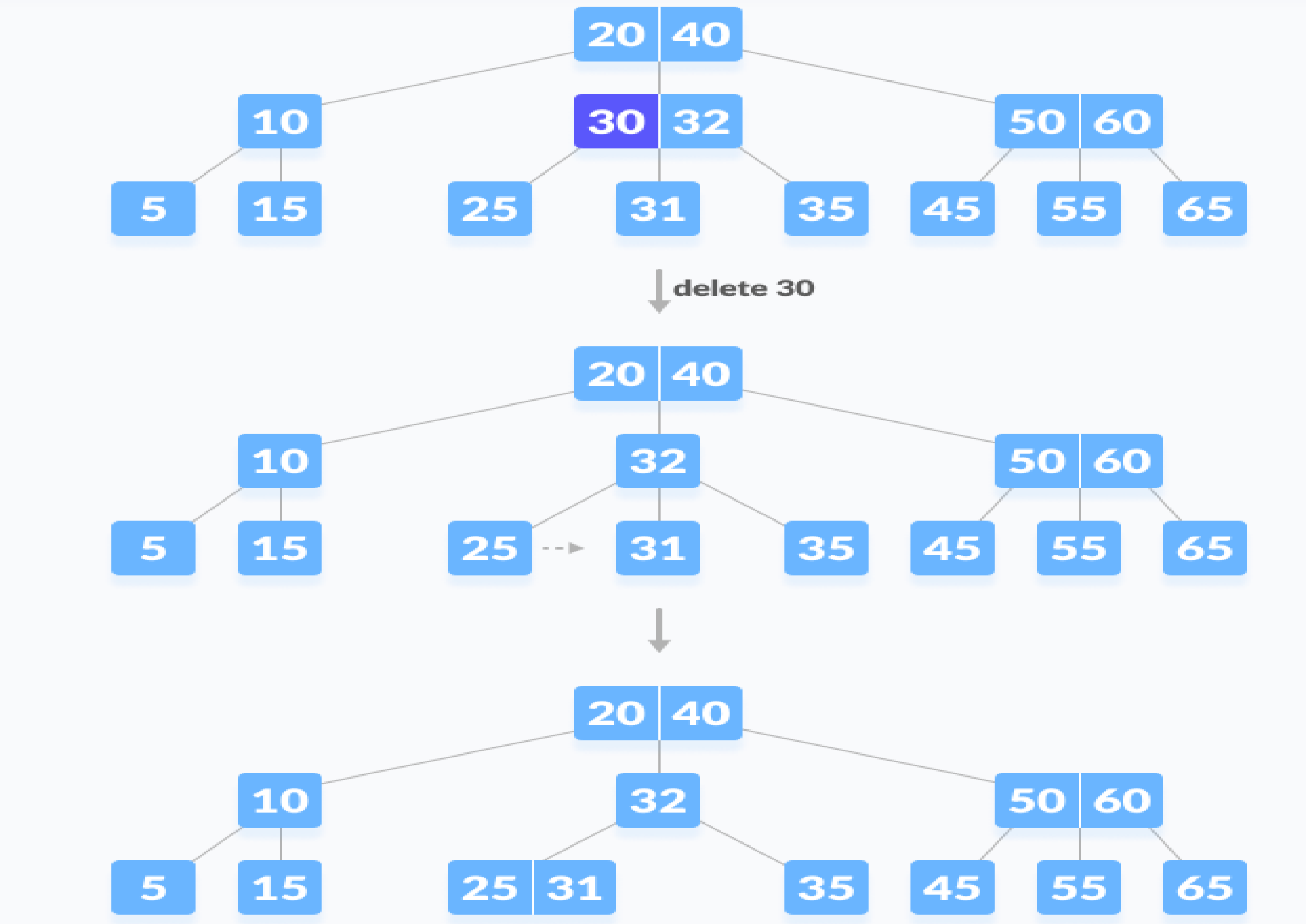
i) The internal node, which is deleted, is replaced by an Inorder predecessor if the left child has more than the minimum number of keys.



B Tree...

Case II :If the key to be deleted lies in the internal node, the following cases occur.

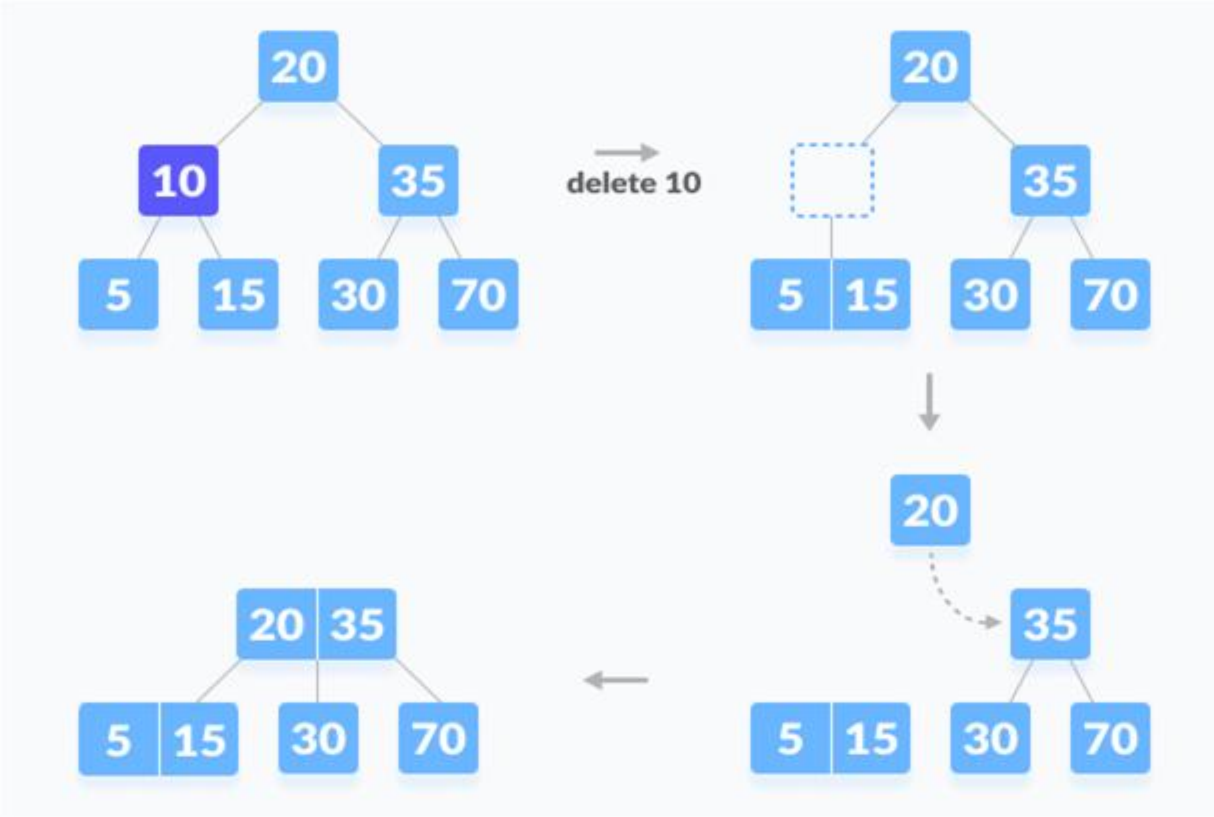
ii) If either child has exactly a minimum number of keys then, merge the left and the right children. After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.



B Tree...

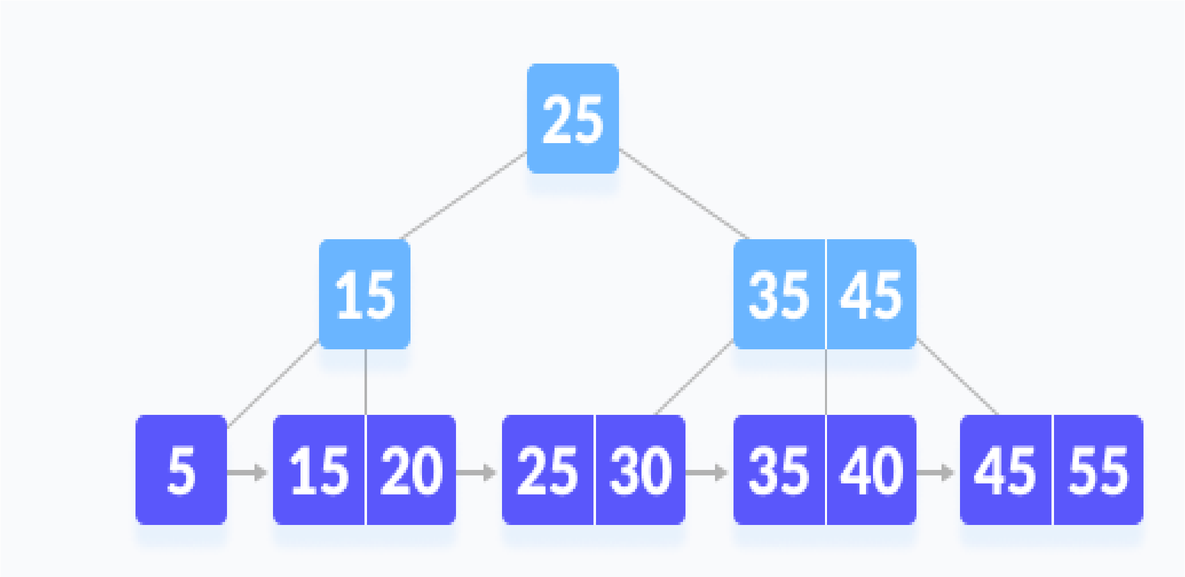
Case III : In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(ii) i.e. merging the children.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).



B+ Tree...

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values. The internal nodes of B+ tree are often called index nodes.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.



B+ Tree...

Operations:

- **Insertion:**

Case I

If the leaf is not full, insert the key into the leaf node in increasing order.

Case II

1. If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way.
2. Break the node at m/2th position.
3. Add m/2th key to the parent node as well.
4. If the parent node is already full, follow steps 2 to 3.

Example:

Construct B+ Tree of order 3 for the elements 5,15, 25, 35, 45.

Max no. of keys =M-1= 3-1=2

Min no. of keys = ⌈M/2⌉ -1 =3/2-1=2-1=1

Min no. of children = ⌈M/2⌉ =3/2=2

B+ Tree...

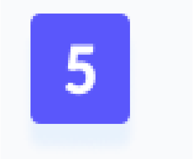
Construct B+ Tree of order 3 for the elements 5,15, 25, 35, 45.

Max no. of keys =M-1= 3-1=2

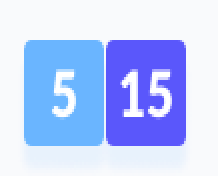
Min no. of keys = [M/2] -1 =3/2-1=2-1=1

Min no. of children = [M/2] =3/2=2

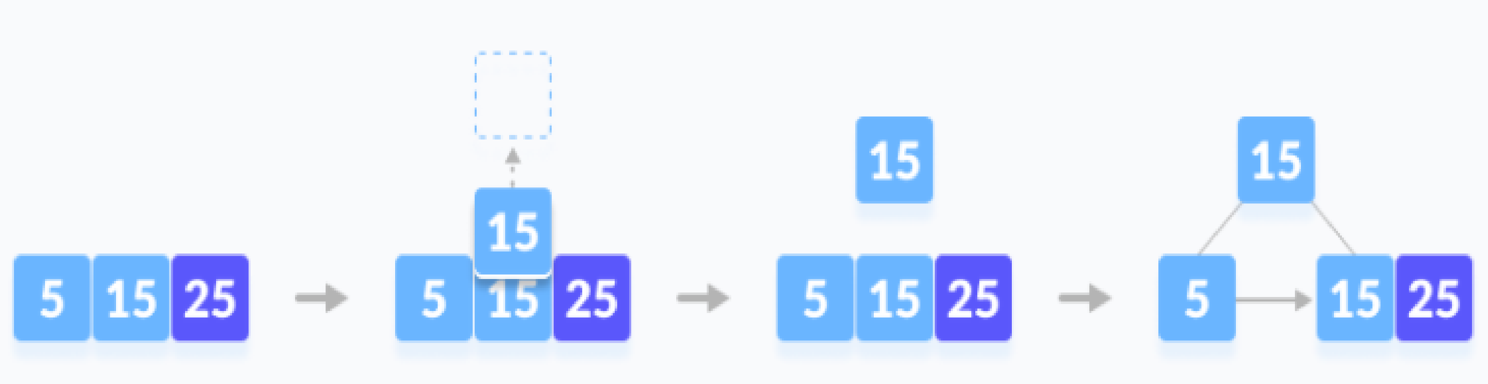
Insert 5



Insert 15



Insert 25



B+ Tree...

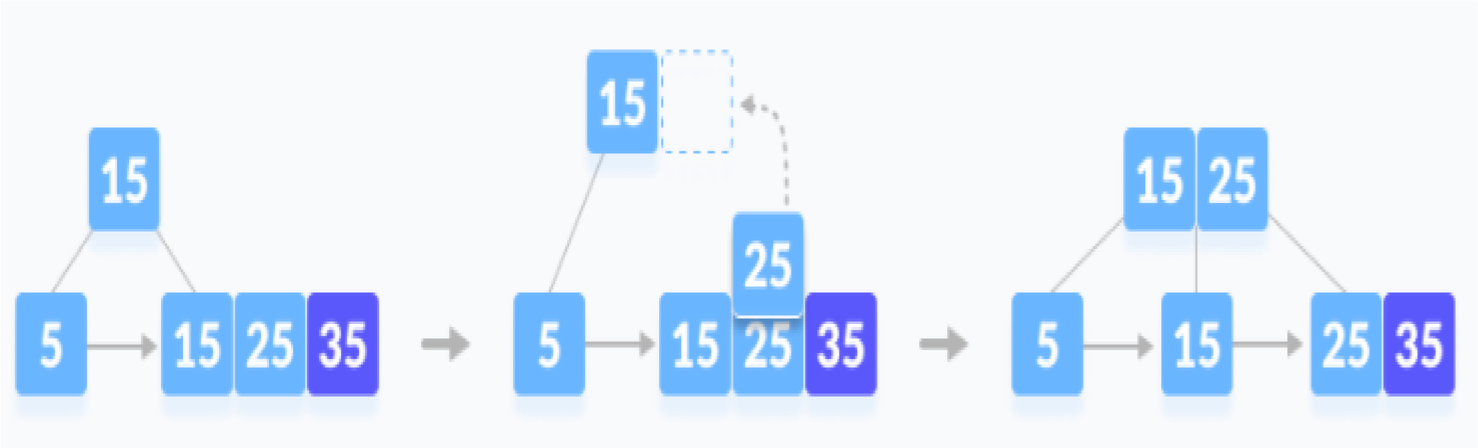
Construct B+ Tree of order 3 for the elements 5,15, 25, 35, 45.

Max no. of keys =M-1= 3-1=2

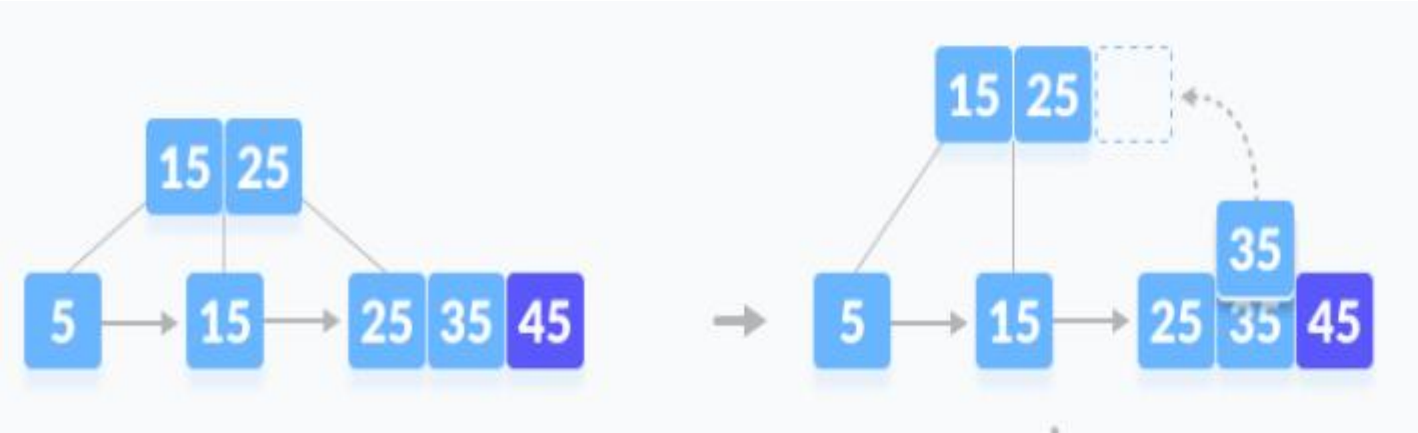
Min no. of keys = [M/2] -1 =3/2-1=2-1=1

Min no. of children = [M/2] =3/2=2

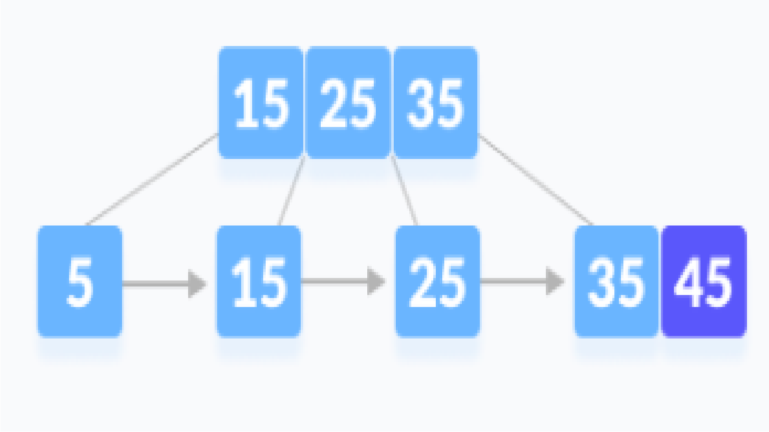
Insert 35

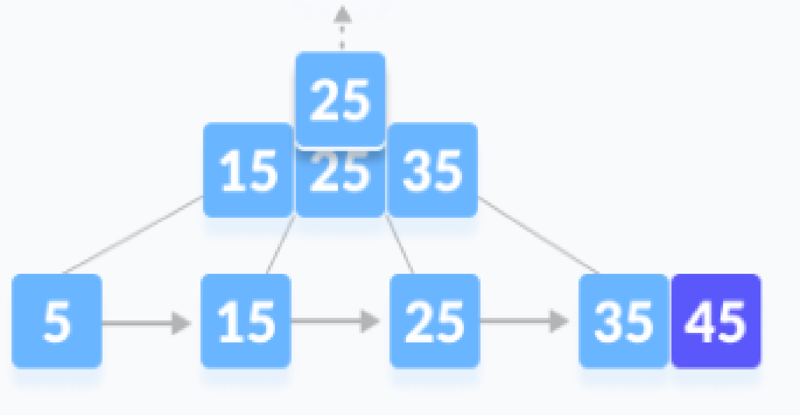


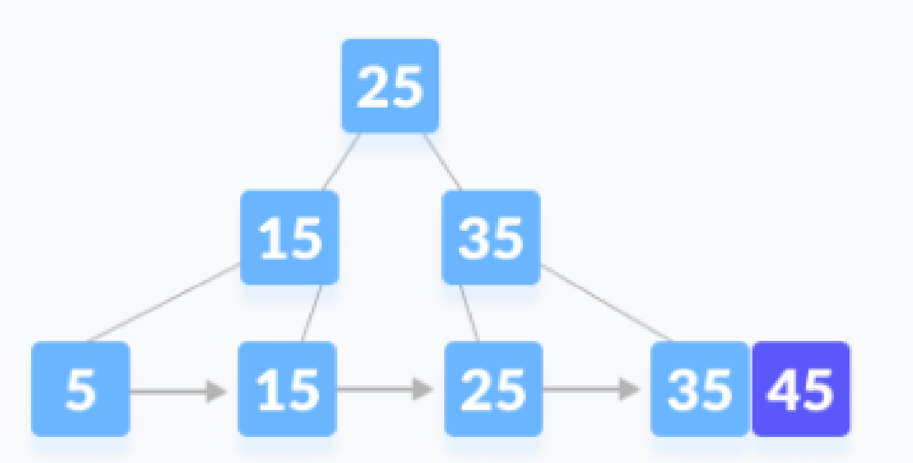
Insert 45



B+ Tree...







B+ Tree...

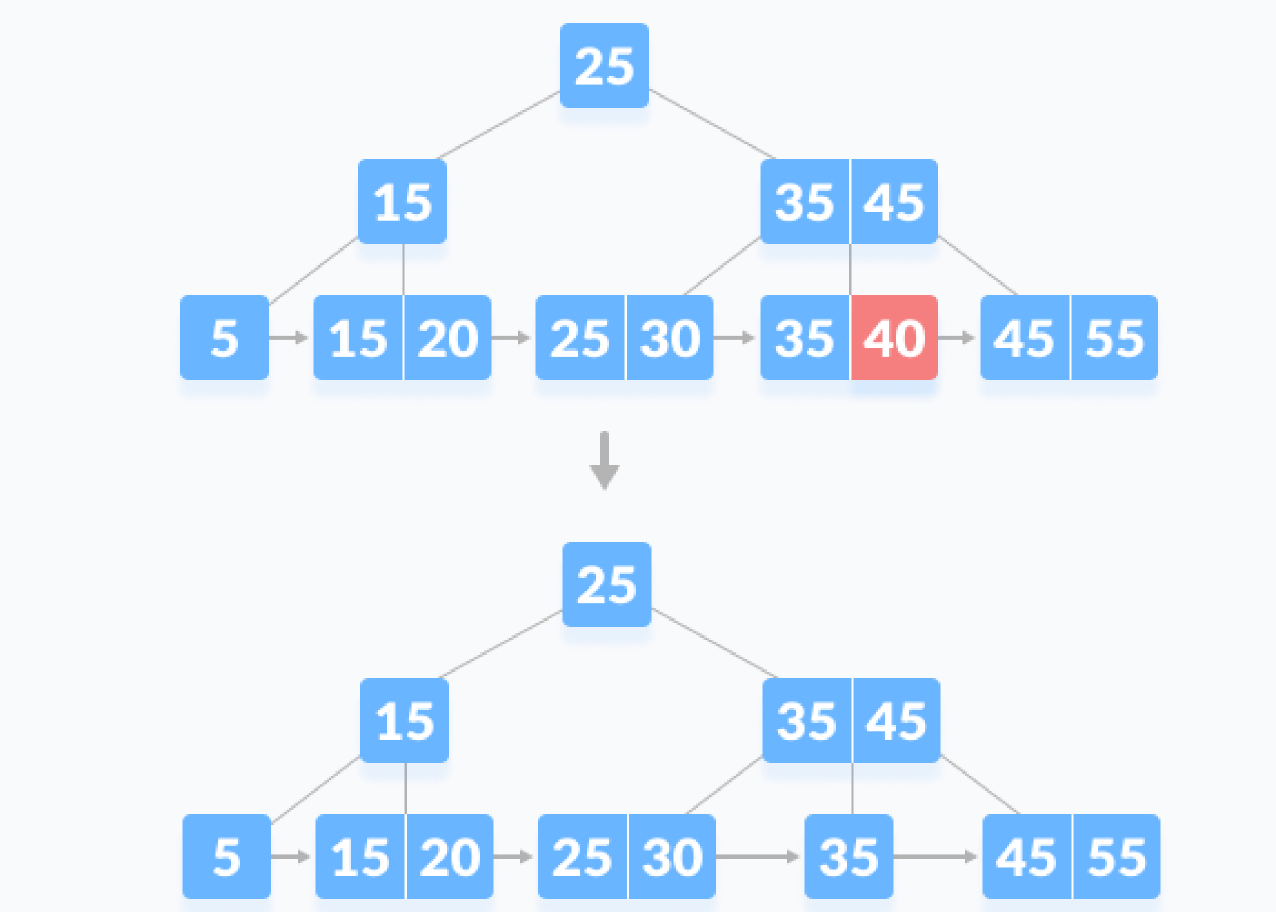
- **Deletion: (B+ Tree of order 3)**

Case I:The key to be deleted is present only at the leaf node not in the indexes (or internal nodes).

There are two cases for it:

i) There is more than the minimum number of keys in the node. Simply delete the key.

Delete 40:



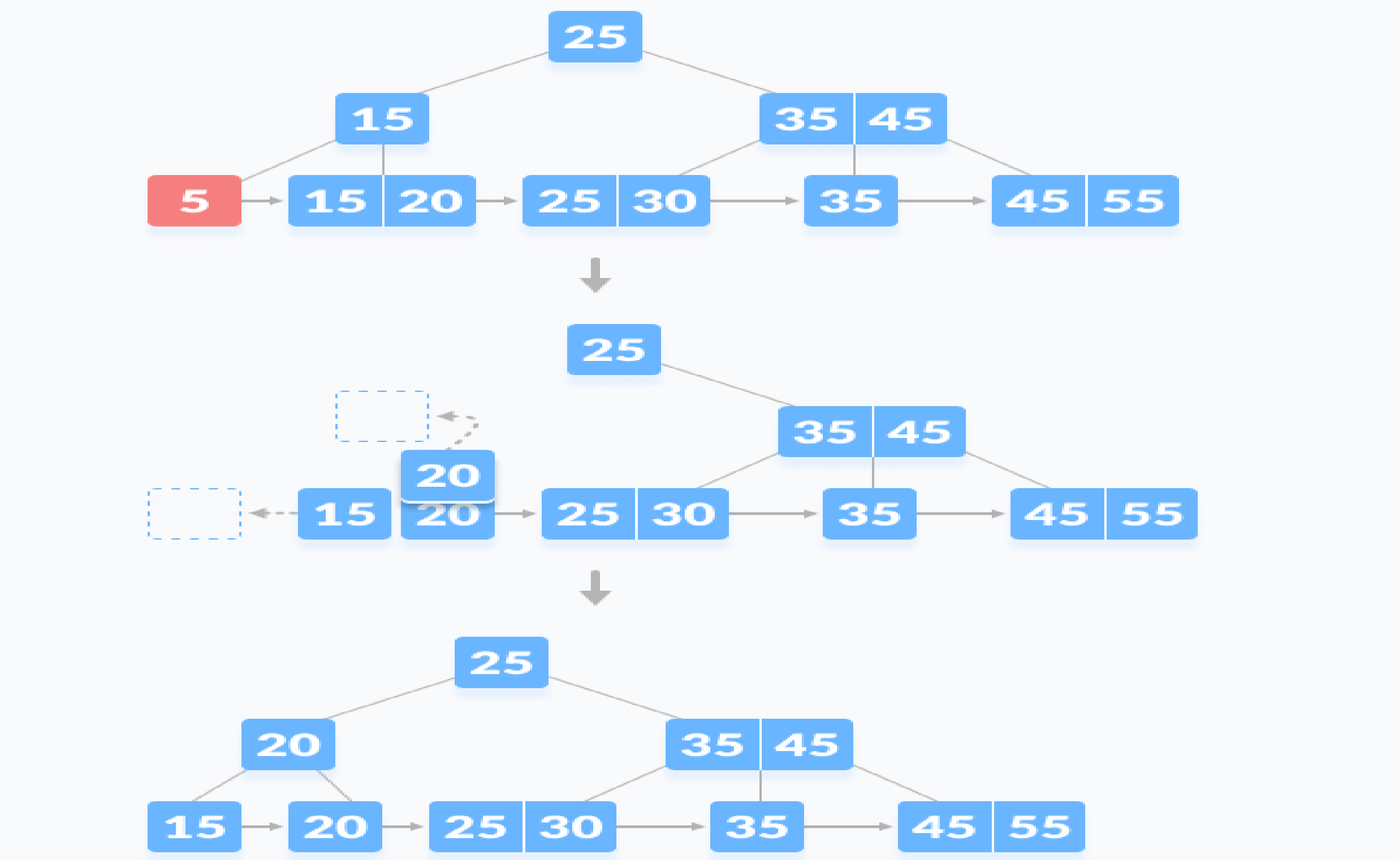
B+ Tree...

➢**Deletion: (B+ Tree of order 3)**

Case I:The key to be deleted is present only at the leaf node not in the indexes (or internal nodes).

ii) There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Minimum from right child through parent.

Delete 5:



B+ Tree...

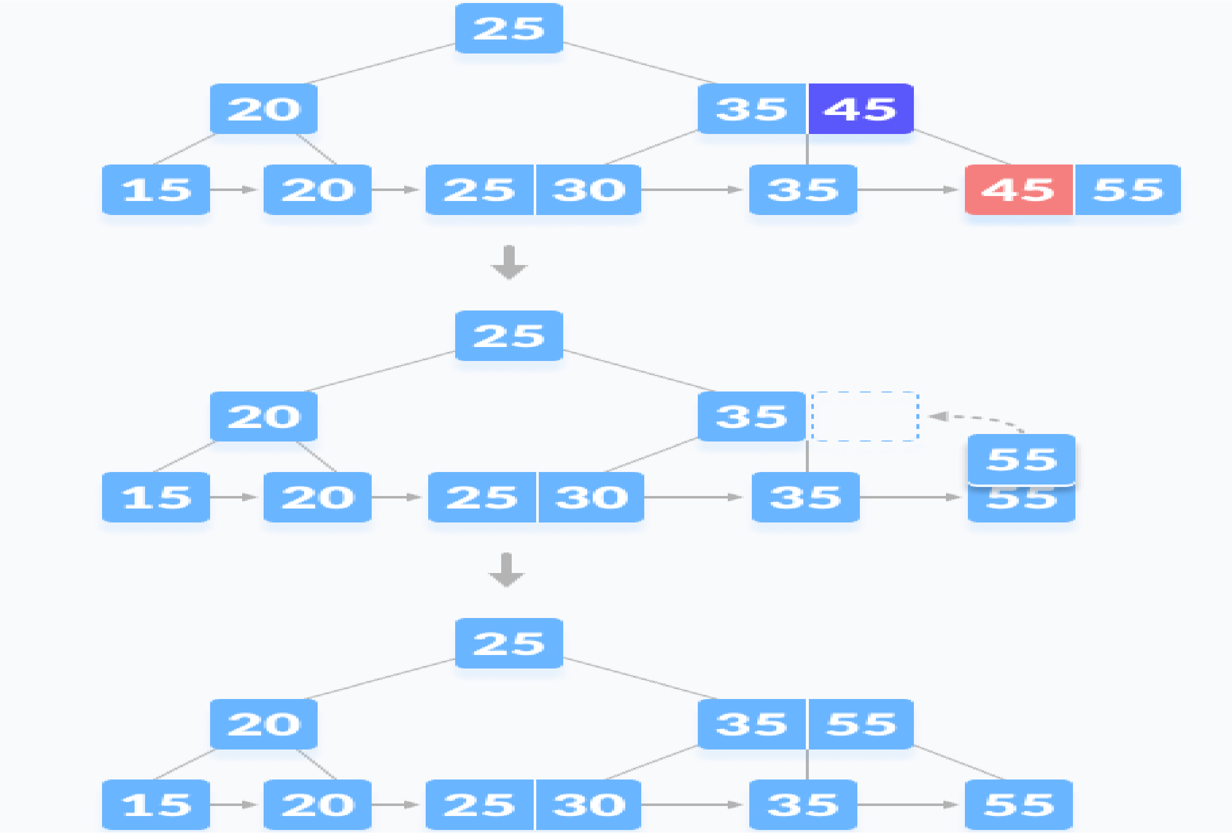
➢**Deletion: (B+ Tree of order 3)**

Case 2:The key to be deleted is present in the internal nodes and leaf node. Then we have to remove them from the internal nodes and leaf node.

There are the following cases for this situation:

i) If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.Fill the empty space in the internal node with the Inorder successor.

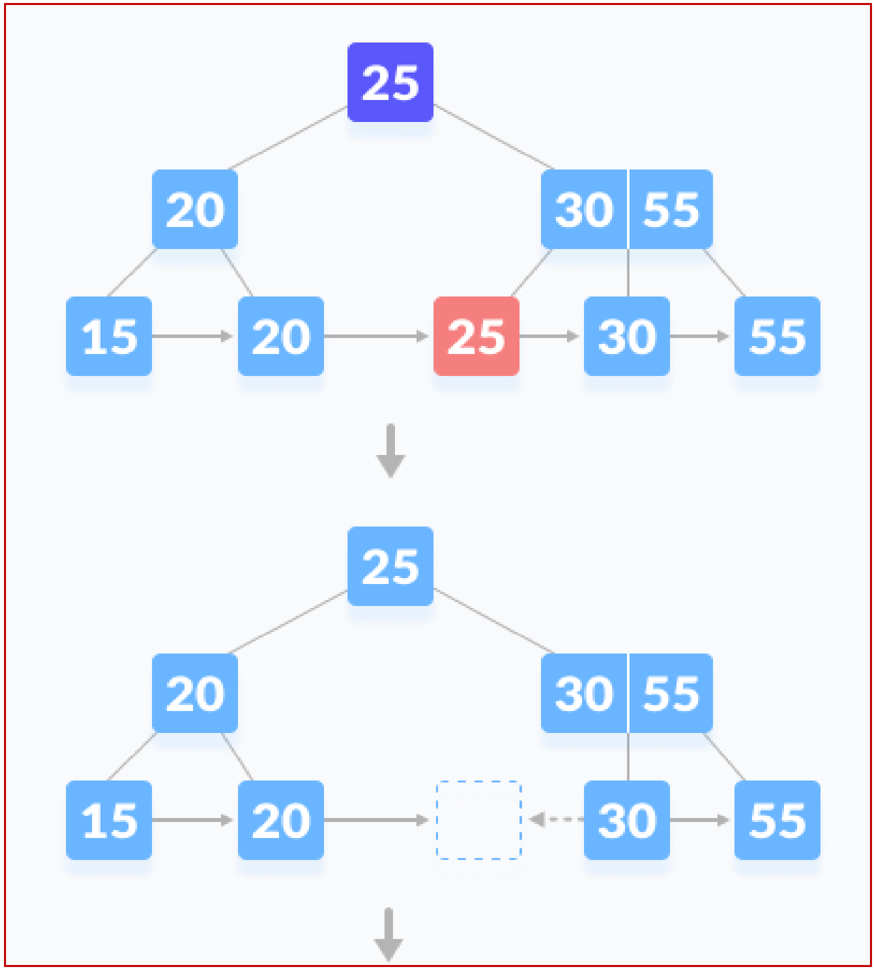
Delete 45:

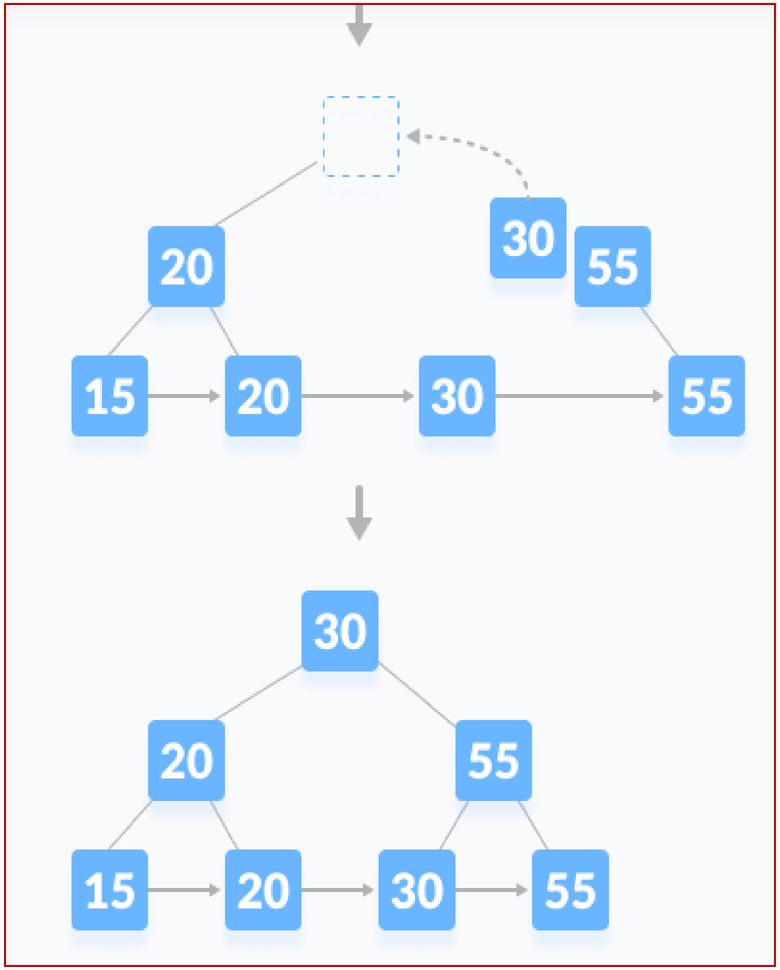


B+ Tree...

iii) This case is similar to Case 2(i) but here, empty space is generated above the immediate parent node.After deleting the key, merge the empty space with its sibling.
Fill the empty space in the grandparent node with the inorder successor.

Delete 25:

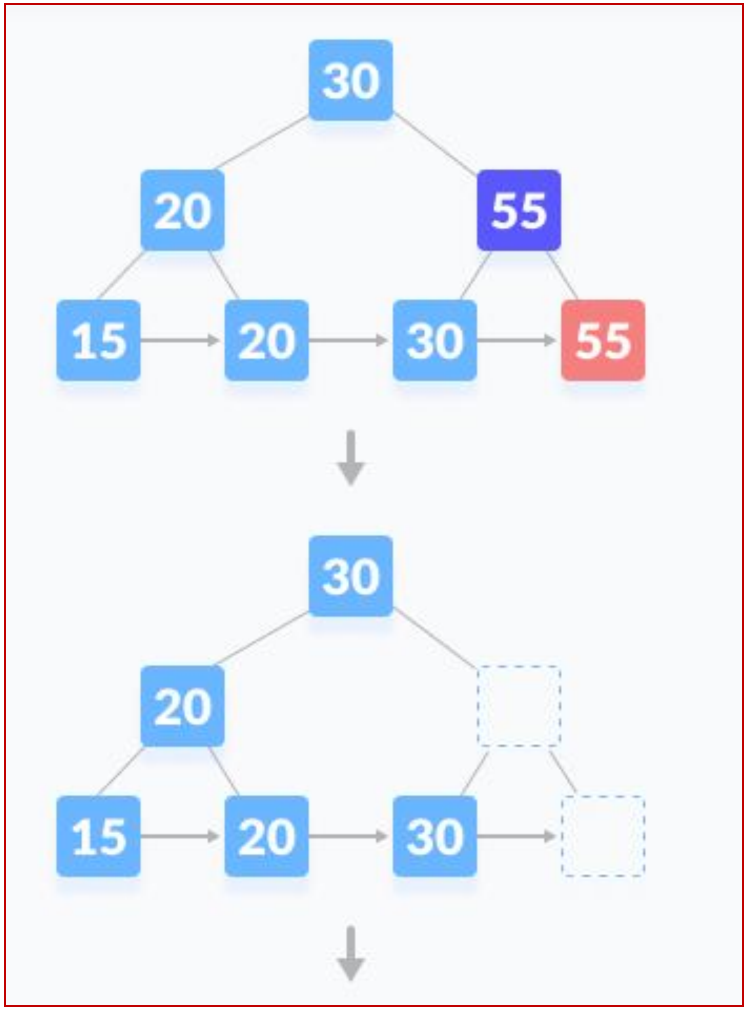


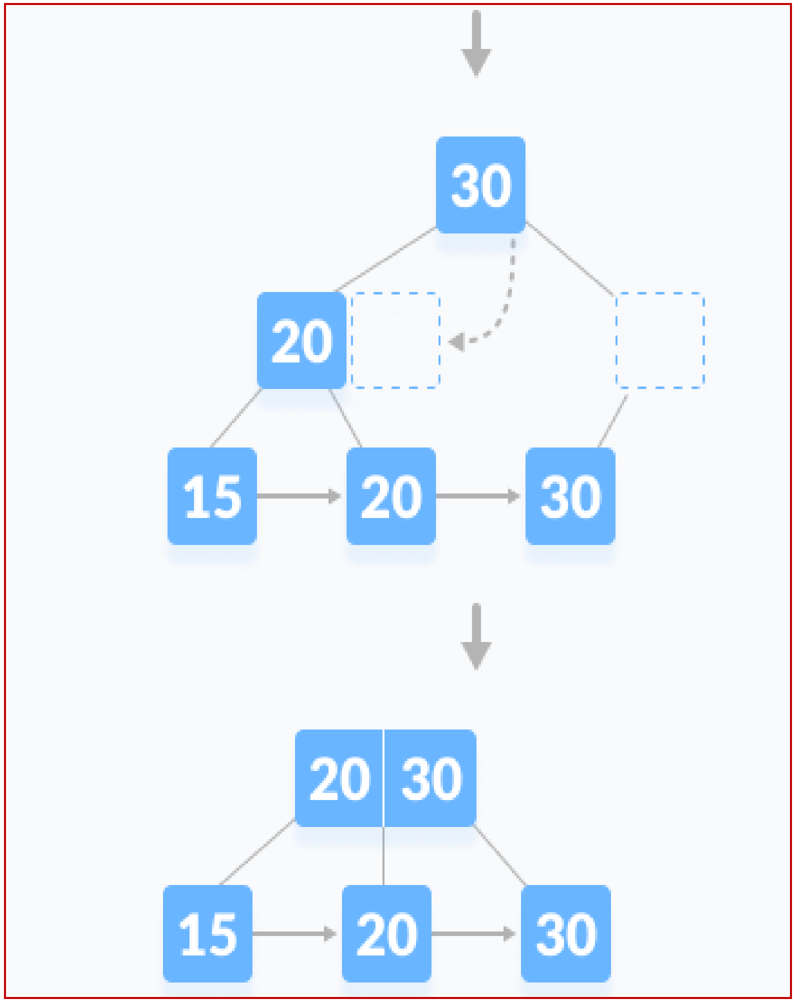


B+ Tree...

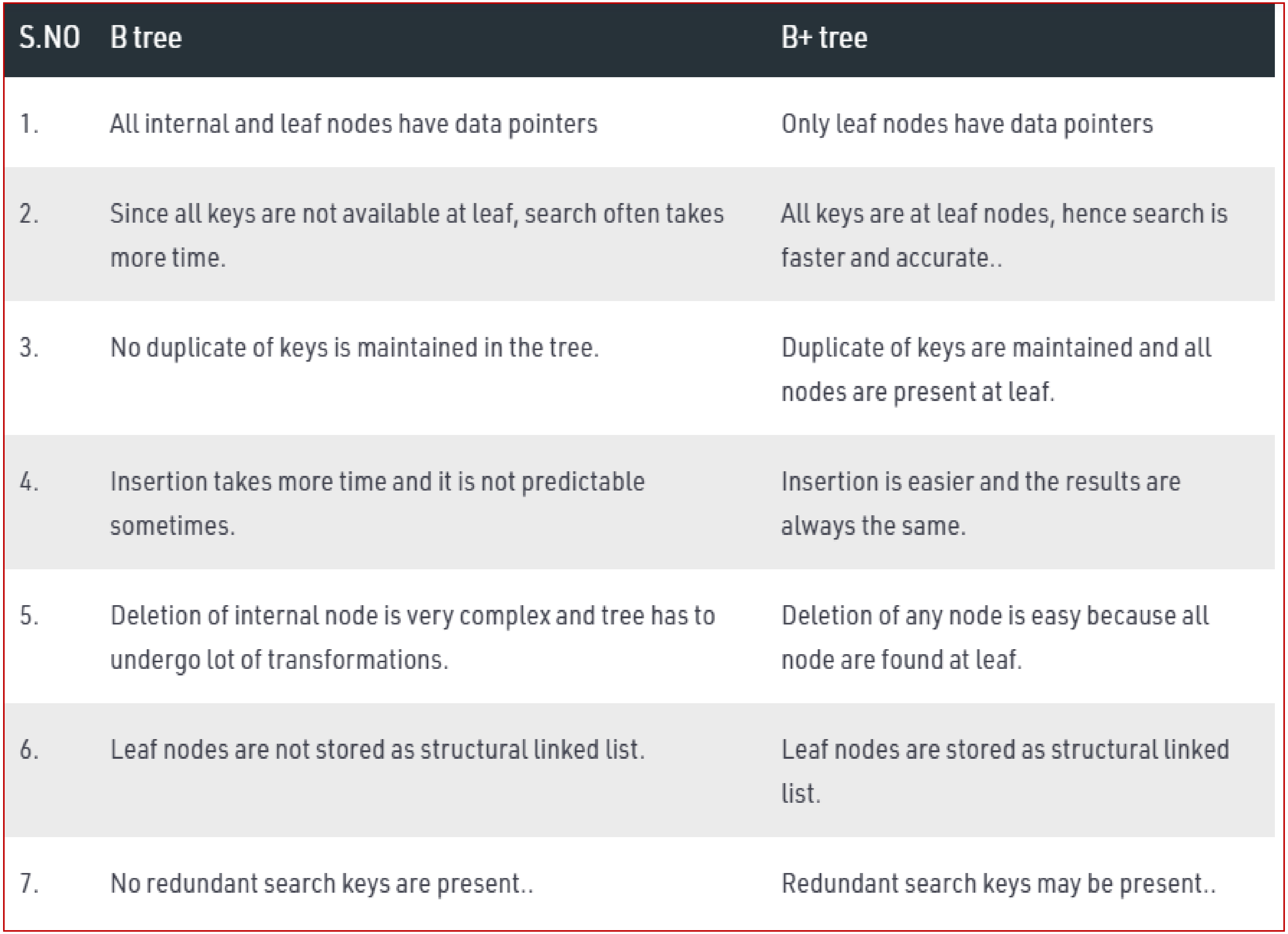
Case 3: In this case, the height of the tree gets shrinked. It is a little complicated.Deleting 55 from the tree below leads to this condition. It can be understood in the illustrations below.

Delete 55:





Comparison between B-Tree and B+ Tree



S.NO	B tree	B+ tree
1.	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
2.	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and accurate..
3.	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at leaf.
4.	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
5.	Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.
6.	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
7.	No redundant search keys are present..	Redundant search keys may be present..