**Data Structures Lab 09**

**Course:** Data Structures (CL2001) **Semester:** Fall 2024

**Instructor:** Mr. Sameer Faisal

**Note:**

* + - * Lab manual cover following below Stack and Queue topics

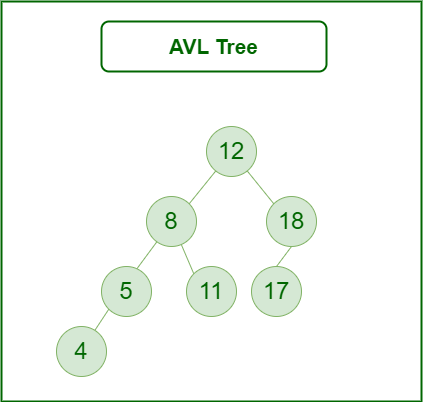
**{Self-balancing Binary Trees, AVL Trees, 2-3 Trees}**

* Maintain discipline during the lab.
* Just raise hand if you have any problem.
* Completing all tasks of each lab is compulsory.
* Get your lab checked at the end of the session.
* Don’t just blatantly copy the same code make changes to it accordingly

# **AVL Tree Data Structure**

An **AVL tree** defined as a self-balancing [**Binary Search Tree**](https://www.geeksforgeeks.org/binary-search-tree-data-structure/) (BST) where the difference between heights of left and right subtrees for any node cannot be more than one. The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node. The AVL tree is named after its inventors, Georgy **A**delson-**V**elsky and Evgenii **L**andis, who published it in their 1962 paper “An algorithm for the organization of information”.

### Example of AVL Trees:



**AVL tree**

The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1. Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.

**BALANCE FACTOR = HEIGHT(LEFT SUBTREE) – HEIGHT(RIGHT SUBTREE)**

### Operations on an AVL Tree:

* Insertion
* Deletion
* Searching [It is similar to performing a search in BST]

### Rotating the subtrees in an AVL Tree while inserting:

The AVL Tree may rotate in one of the following four ways to keep itself balanced:

**Left Rotation**:

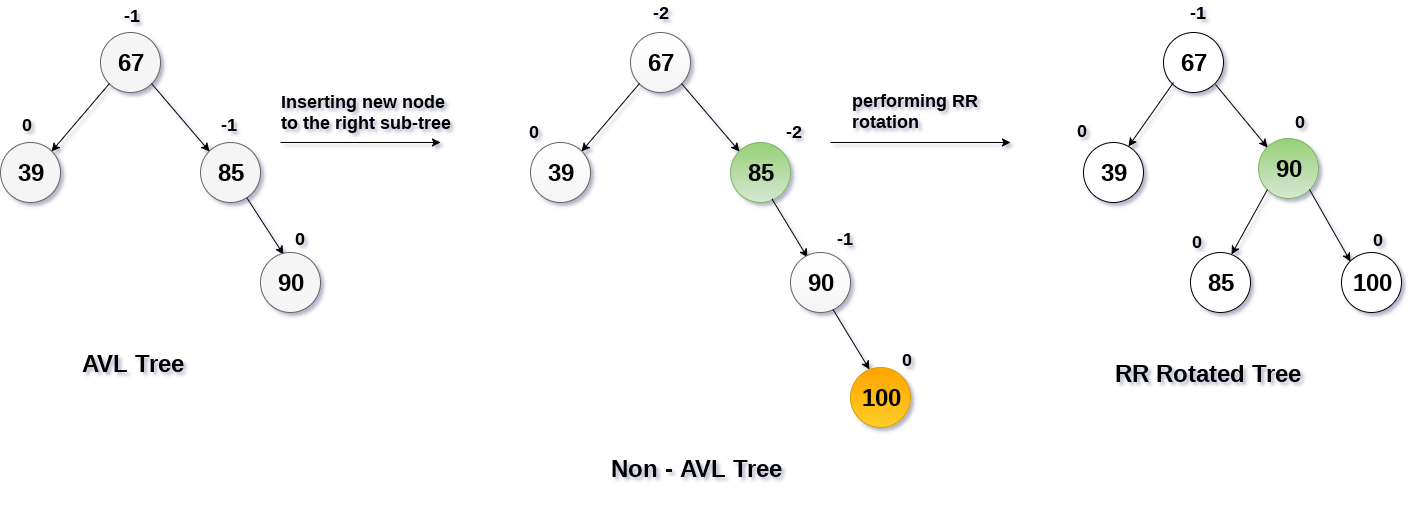
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.

A diagram of a tree and a tree

Description automatically generated

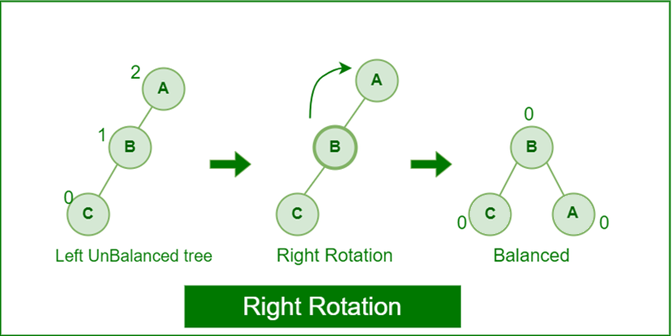
**Left-Rotation in AVL tree**

**Example:**



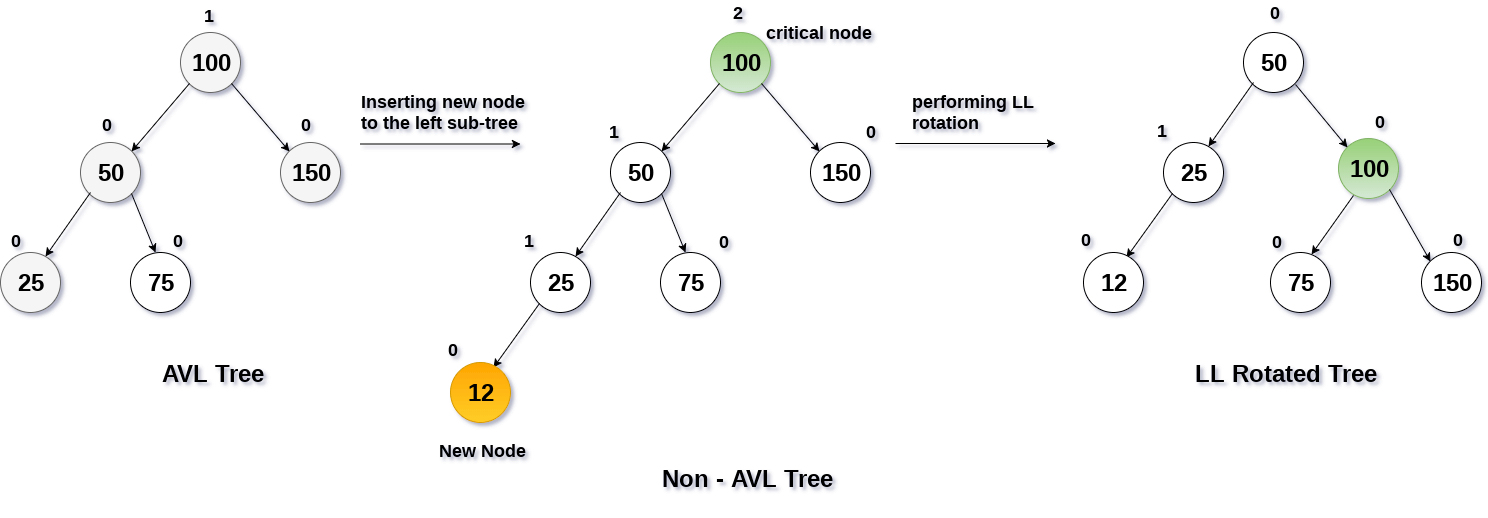
**Right Rotation:**

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



**Right Rotation in AVL tree**

**Example:**



**Left-Right Rotation**:

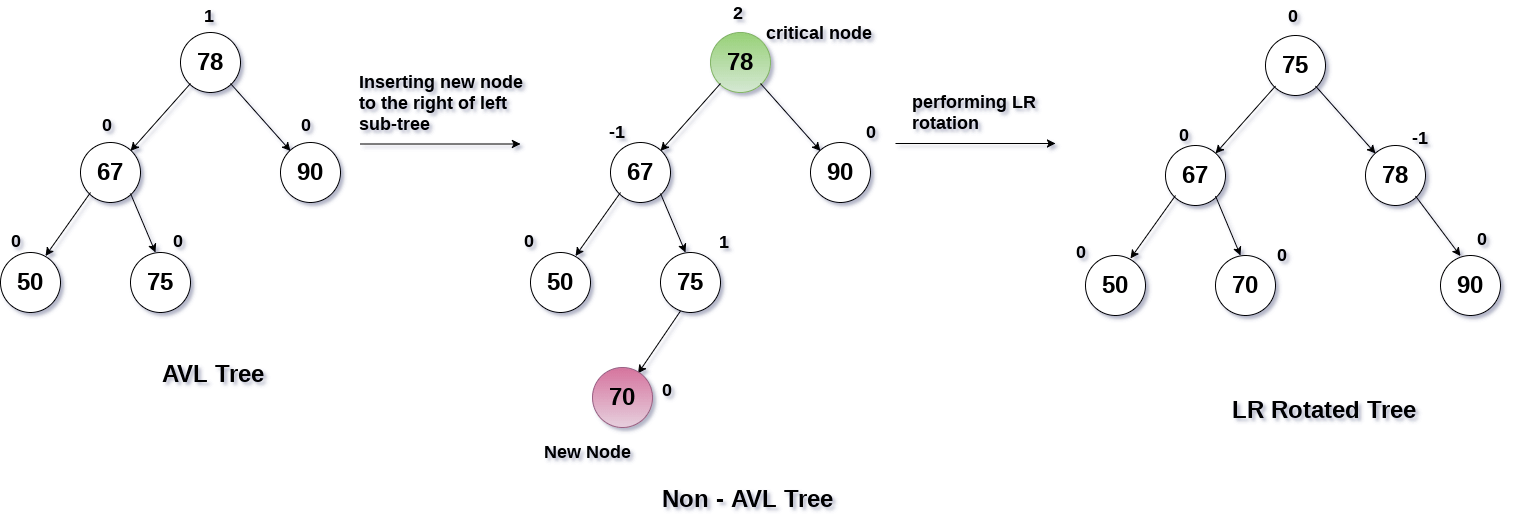
A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.

A diagram of a left-right rotation

Description automatically generated

**Left-Right Rotation in AVL tree**

**Example:**



**Right-Left Rotation:**

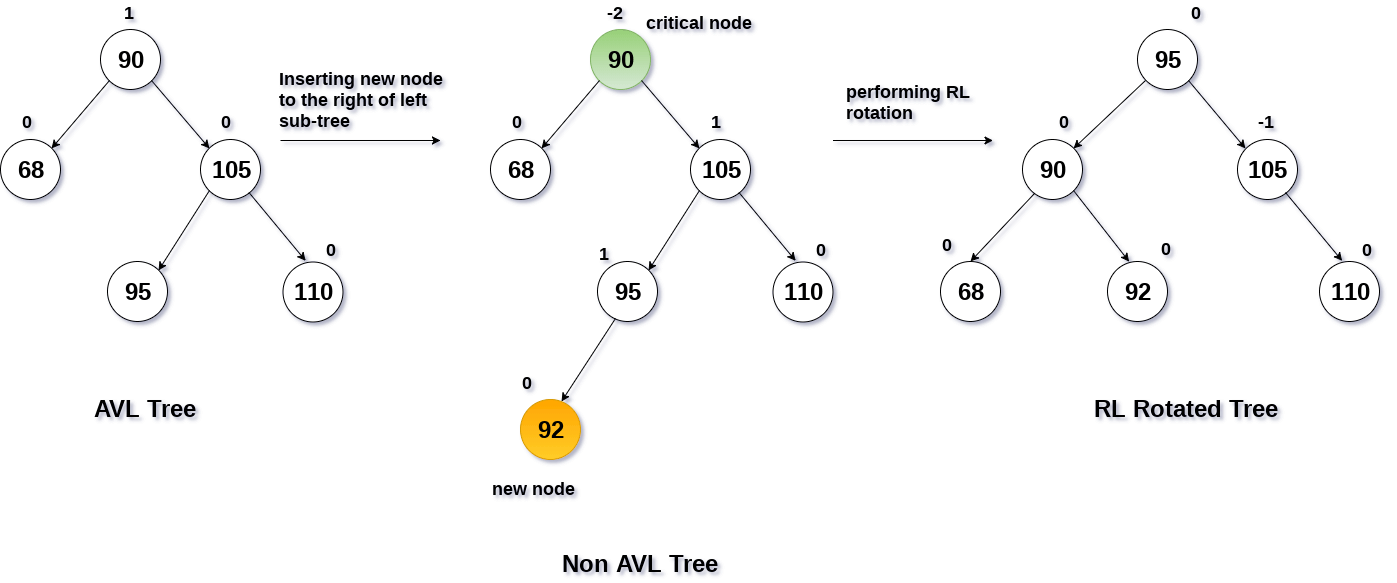
A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.

A diagram of a diagram of a right-left rotation

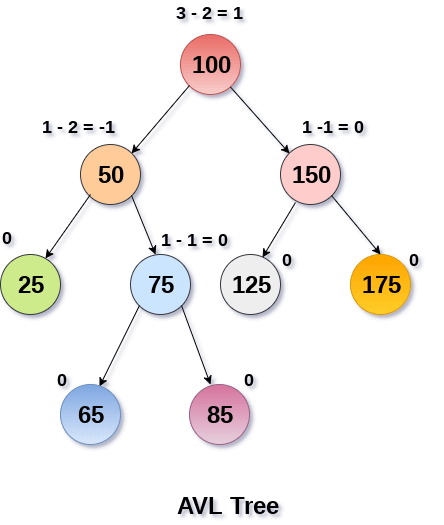
Description automatically generated

**Right-Left Rotation in AVL tree**

**Example:**



An **AVL tree** is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



### Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. Better searching time complexity compared to other trees like binary tree.
4. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

### Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
4. Take more processing for balancing.

**Case for Insertion:**

START

if node == null then:

return new node

if key < node.key then:

node.left = insert (node.left, key)

else if (key > node.key) then:

node.right = insert (node.right, key)

else

return node

node.height = 1 + max (height (node.left), height (node.right))

balance = getBalance (node)

if balance > 1 and key < node.left.key then:

rightRotate

if balance < -1 and key > node.right.key then:

leftRotate

if balance > 1 and key > node.left.key then:

node.left = leftRotate (node.left)

rightRotate

if balance < -1 and key < node.right.key then:

node.right = rightRotate (node.right)

leftRotate (node)

return node

END

### ****Steps to follow for insertion:****

Let the newly inserted node be **w**

* Perform standard**BST** insert for **w**.
* Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y**be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild**of **z** that comes on the path from **w**to **z**.
* Re-balance the tree by performing appropriate rotations on the subtree rooted with**z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
* Following are the possible 4 arrangements:
  + y is the left child of z and x is the left child of y (Left Left Case)
  + y is the left child of z and x is the right child of y (Left Right Case)
  + y is the right child of z and x is the right child of y (Right Right Case)
  + y is the right child of z and x is the left child of y (Right Left Case)

**Case for Deletion**

START

if root == null: return root

if key < root.key:

root.left = delete Node

else if key > root.key:

root.right = delete Node

else:

if root.left == null or root.right == null then:

Node temp = null

if (temp == root.left)

temp = root.right

else

temp = root.left

if temp == null then:

temp = root

root = null

else

root = temp

else:

temp = minimum valued node

root.key = temp.key

root.right = delete Node

if (root == null) then:

return root

root.height = max (height (root.left), height (root.right)) + 1

balance = getBalance

if balance > 1 and getBalance (root.left) >= 0:

rightRotate

if balance > 1 and getBalance (root.left) < 0:

root.left = leftRotate (root.left);

rightRotate

if balance < -1 and getBalance (root.right) <= 0:

leftRotate

if balance < -1 and getBalance (root.right) > 0:

root.right = rightRotate (root.right);

leftRotate

return root

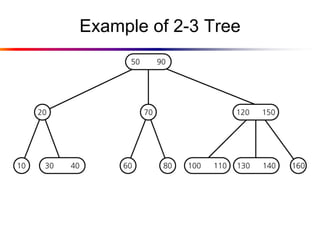
END

### ****Steps to follow for Deletion:****

1. Perform the normal BST deletion.
2. The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3. Get the balance factor (left subtree height – right subtree height) of the current node.
4. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

# **2-3 Trees Data Structure**

A 2-3 Tree is a type of self-balancing search tree used in computer science to maintain data in a sorted order, enabling efficient search, insertion, and deletion operations. Each node in a 2-3 Tree can have either one key (forming a 2-node) or two keys (forming a 3-node), with each node having either two or three child nodes, respectively. This means that nodes can store multiple keys and adjust their structure dynamically to maintain balance, distributing keys in a way that ensures all leaf nodes are at the same depth.

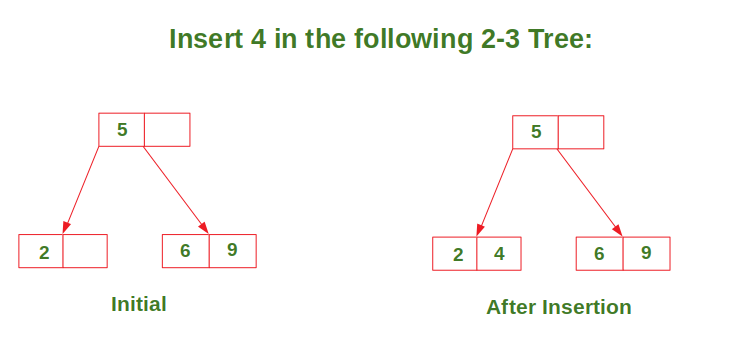


### Operations on an 2-3 Tree:

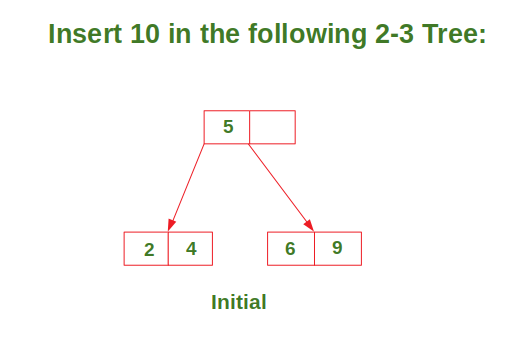
* Insertion
* Deletion
* Searching

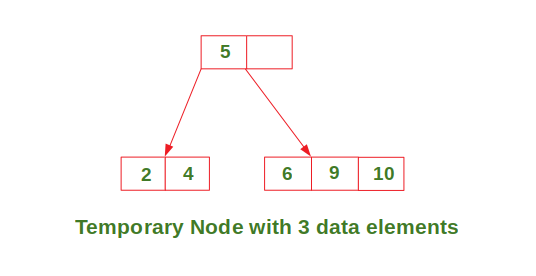
**Insertion:**

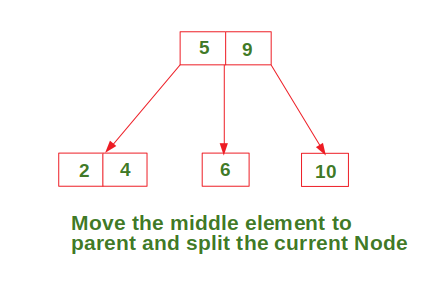
* **Case 1:** Insert in a node with only one data element



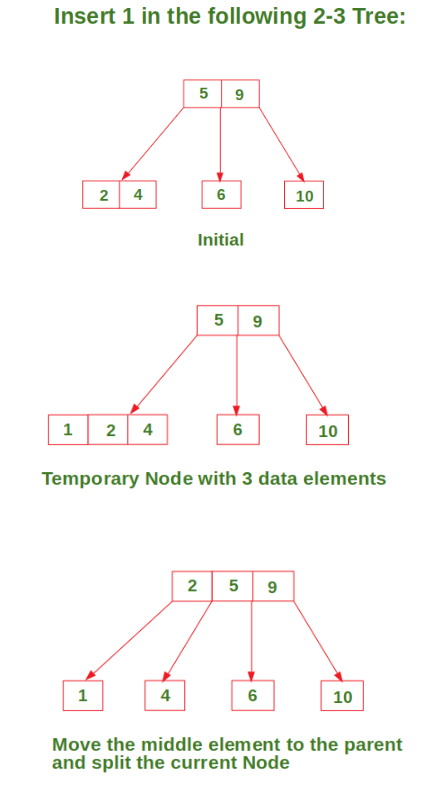
* **Case 2:** Insert in a node with two data elements whose parent contains only one data element.

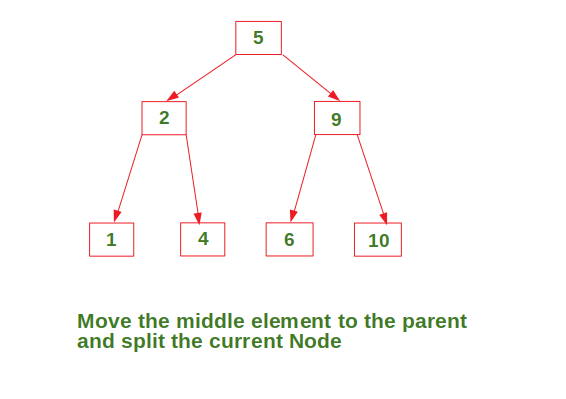






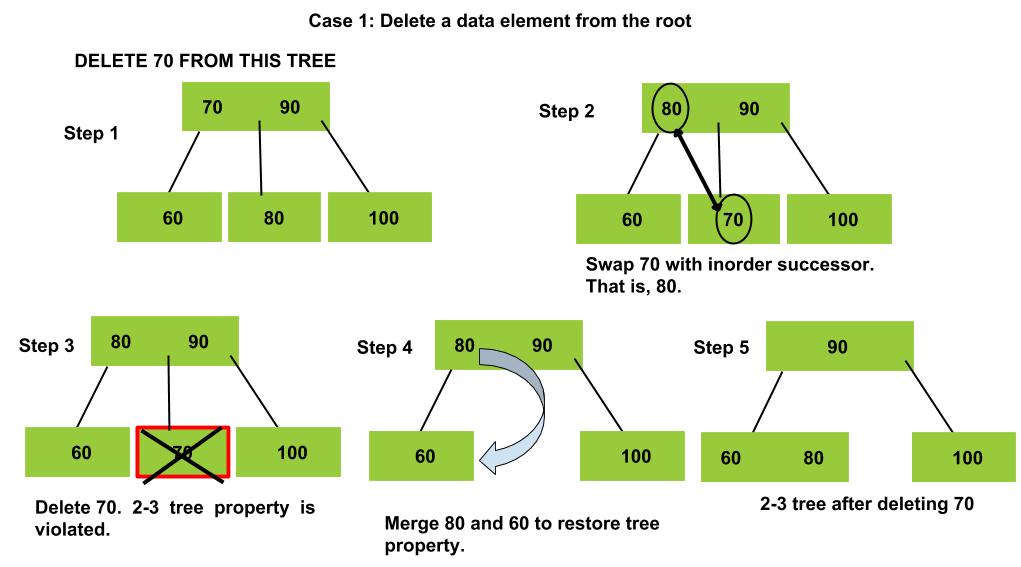
* **Case 3:** Insert in a node with two data elements whose parent also contains two data elements.





**Deletion:**

* To delete a value, it is replaced by its in-order successor and then removed.
* If a node is left with less than one data value then two nodes must be merged together.
* If a node becomes empty after deleting a value, it is then merged with another node.



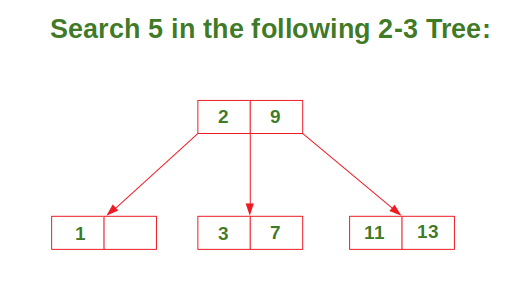
**Searching:**

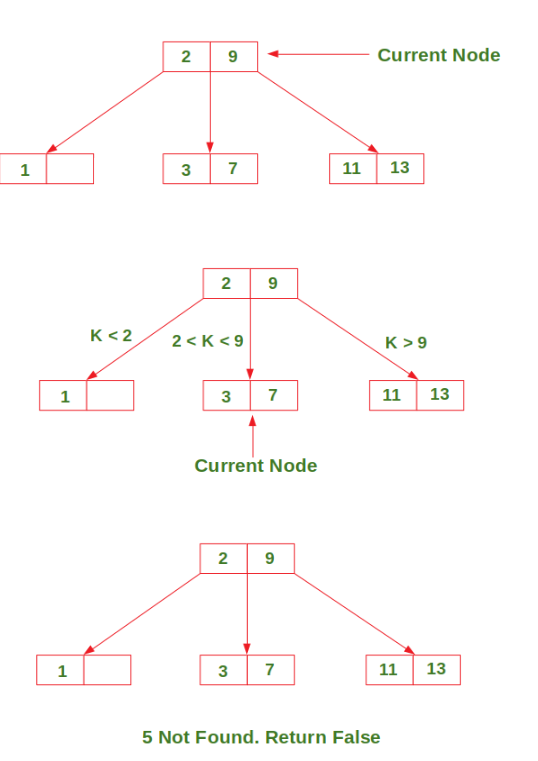
Base cases:

1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn’t contain the required key value **K**, return False.

Recursive Calls:

1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** < currentNode.rightVal, we explore the middle subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node.





**Pseudosodes**

* **Insertion:**

FUNCTION insert(tree, key):

IF tree is empty:

CREATE a new root with key

RETURN

currentNode = tree.root

# Step 1: Traverse to find the correct insertion point

WHILE currentNode is not a leaf:

IF currentNode is a 2-node:

IF key < currentNode.key1:

currentNode = currentNode.leftChild

ELSE:

currentNode = currentNode.rightChild

ELSE IF currentNode is a 3-node:

IF key < currentNode.key1:

currentNode = currentNode.leftChild

ELSE IF key > currentNode.key2:

currentNode = currentNode.rightChild

ELSE:

currentNode = currentNode.middleChild

# Step 2: Insert key in the leaf node

INSERT key into currentNode in sorted order

# Step 3: Handle splitting if the node becomes overfull

WHILE currentNode has 3 keys:

middleKey = currentNode.key2

leftChild = Node(currentNode.key1)

rightChild = Node(currentNode.key3)

IF currentNode is the root:

# Create a new root if splitting the root

tree.root = Node(middleKey)

tree.root.leftChild = leftChild

tree.root.rightChild = rightChild

RETURN

parentNode = currentNode.parent

INSERT middleKey into parentNode in sorted order

REPLACE currentNode in parentNode’s children with leftChild and rightChild

currentNode = parentNode # Move up to handle further splits if needed

* **Deletion:**

FUNCTION delete(tree, key):

# Step 1: Locate the node containing the key

currentNode = tree.root

WHILE currentNode is not NULL:

IF key is found in currentNode:

BREAK

ELSE IF currentNode is a 2-node:

IF key < currentNode.KEY1:

currentNode = currentNode.LEFT

ELSE:

currentNode = currentNode.RIGHT

ELSE IF currentNode is a 3-node:

IF key < currentNode.KEY1:

currentNode = currentNode.LEFT

ELSE IF key > currentNode.KEY2:

currentNode = currentNode.RIGHT

ELSE:

currentNode = currentNode.MIDDLE

# Step 2: Delete the key

IF key is in a leaf node:

REMOVE key from the leaf node

ELSE:

# Key is in an internal node, find and replace with predecessor or successor

IF currentNode has LEFT child:

predecessor = findMax(currentNode.LEFT)

REPLACE key in currentNode with predecessor

currentNode = predecessor # Move to predecessor node to delete key

ELSE:

successor = findMin(currentNode.RIGHT)

REPLACE key in currentNode with successor

currentNode = successor # Move to successor node to delete key

# Step 3: Handle underfull nodes after deletion

WHILE currentNode is underfull:

sibling = findSiblingWithExtraKey(currentNode)

IF sibling is not NULL:

# Borrow from sibling if possible

borrowFromSibling(currentNode, sibling)

ELSE:

# Merge with sibling if borrowing is not possible

parentNode = currentNode.PARENT

mergeWithSibling(currentNode, parentNode)

IF parentNode becomes underfull:

currentNode = parentNode

ELSE:

BREAK

# Step 4: Adjust root if it becomes underfull and empty

IF tree.root has no keys:

tree.root = tree.root.LEFT or tree.root.RIGHT # Make child the new root if empty

* **Searching:**

FUNCTION search(tree, key):

# Step 1: Start at the root

currentNode = tree.root

# Step 2: Traverse the tree until key is found or leaf is reached

WHILE currentNode is not NULL:

IF currentNode is a 2-node:

IF key == currentNode.KEY1:

RETURN currentNode # Key found

ELSE IF key < currentNode.KEY1:

currentNode = currentNode.LEFT # Move left

ELSE:

currentNode = currentNode.RIGHT # Move right

ELSE IF currentNode is a 3-node:

IF key == currentNode.KEY1 OR key == currentNode.KEY2:

RETURN currentNode # Key found

ELSE IF key < currentNode.KEY1:

currentNode = currentNode.LEFT # Move left

ELSE IF key > currentNode.KEY2:

currentNode = currentNode.RIGHT # Move right

ELSE:

currentNode = currentNode.MIDDLE # Move middle

# Step 3: If we reached a leaf without finding the key

RETURN NOT\_FOUND # Key not found