MALLA REDDY UNIVERSITY

MR22-1CS0104

ADVANCED DATA STRUCTURES

II YEAR B.TECH. (CSE) / II – SEM

Unit-4

Efficient Binary Search Trees:

Binary Search Tree -

AVL Trees - Insertion and deletion operations,

Splay Trees - Insertion and deletion operations,

Red - Black Trees - Insertion and deletion operations,

B-Trees – Basic operations of B-Trees,

B+ Trees.

Tree Traversal

- A traversal of a tree T is a systematic way of accessing, or “visiting,” all the positions of T.
- Three ways of tree traversals,
  - Preorder Traversals
  - Inorder Traversals and
  - Postorder Traversals

Tree Traversal Algorithm

Preorder Traversals:

void preorder(Node root) {

Initially check if root is equal to null or not, if root is not equal to null then recur the tree towards left subtree and right subtree

Recurrence steps if root != null are:

Step 1: Print the data before recursive calls with root.data as argument and every element should be printed with spaces.

Step 2: Call function preorder() with root.left as argument to traverse tree towards left in recursive way.

Step 3: Call function preorder() with root.right as argument to traverse tree towards right in recursive way. **}**

Tree Traversal Algorithm

Inorder Traversals:

void inorder(Node root) {

Initially check if root is equal to null or not, if root is not equal to null then recur the tree towards left subtree and right subtree

Recurrence steps if root != null are:

Step 1: Call function inorder() with root.left as argument to traverse tree towards left in recursive way.

Step 2: Print the data in the middle with root.data as argument and every element should be printed with spaces.

Step 3: Call function inorder() with root.right as argument to traverse tree towards right in recursive way. }

Tree Traversal Algorithm

Postorder Traversals:

void postorder(Node root) {

Initially check if root is equal to null or not, if root is not equal to null then recur the tree towards left subtree and right subtree

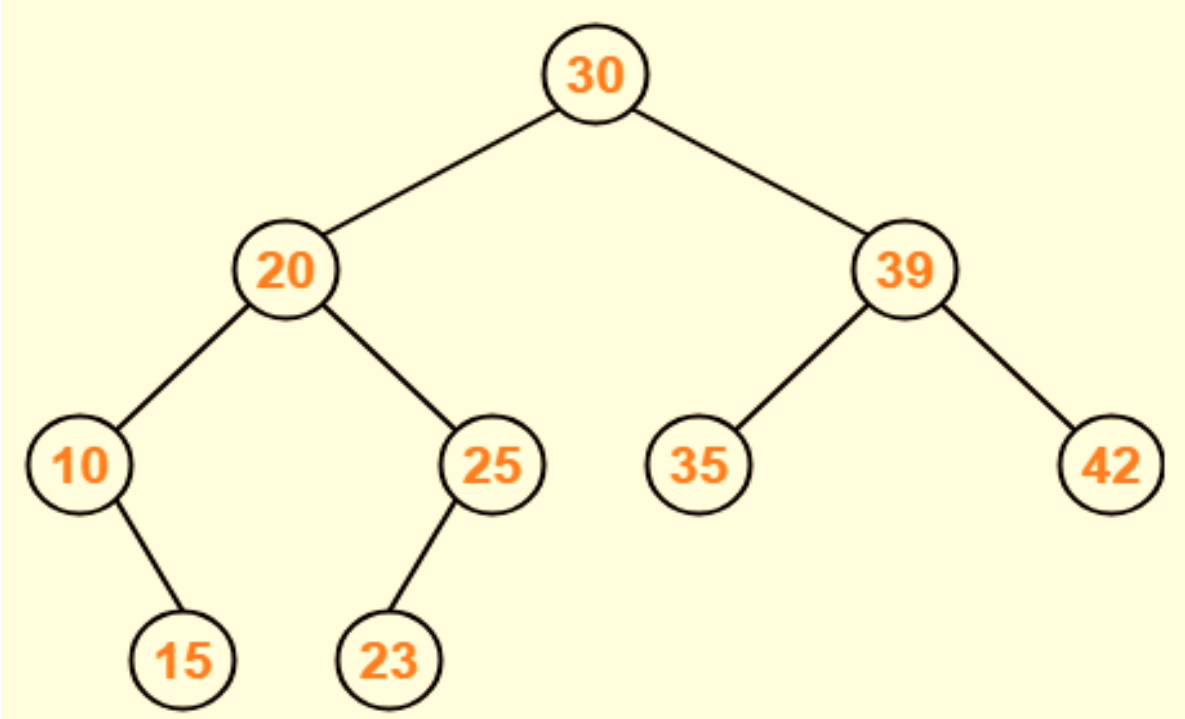
Recurrence steps if root != null are:

Step 1: Call function postorder() with root.left as argument to traverse tree towards left in recursive way.

Step 2: Call function postorder() with root.right as argument to traverse tree towards right in recursive way.

Step 3: Print the data after both recurrence calls with root.data as argument and every element should be printed with spaces. **}**

Tree Traversal Algorithm



Inorder: 10, 15, 20, 23, 25, 30, 35, 39, 42

Preorder: 30, 20, 10, 15, 25, 23, 39, 35, 42

Postorder: 15, 10, 23, 25, 20, 35, 42, 39, 30

Tree Traversal

```
void inorderInBST(Node root) {
        if (root == null)          return;
        inorderInBST(root.left);
        System.out.print(root.data + " ");
        inorderInBST(root.right);    }

void preorderInBST(Node root) {
        if (root == null)          return;
        System.out.print(root.data + " ");
        preorderInBST(root.left);
        preorderInBST(root.right);     }
```

Tree Traversal

```
void postorderInBST(Node root) {
        if (root == null)          return;
        postorderInBST(root.left);
        postorderInBST(root.right);
        System.out.print(root.data + " ");       }
```

Binary Search Tree

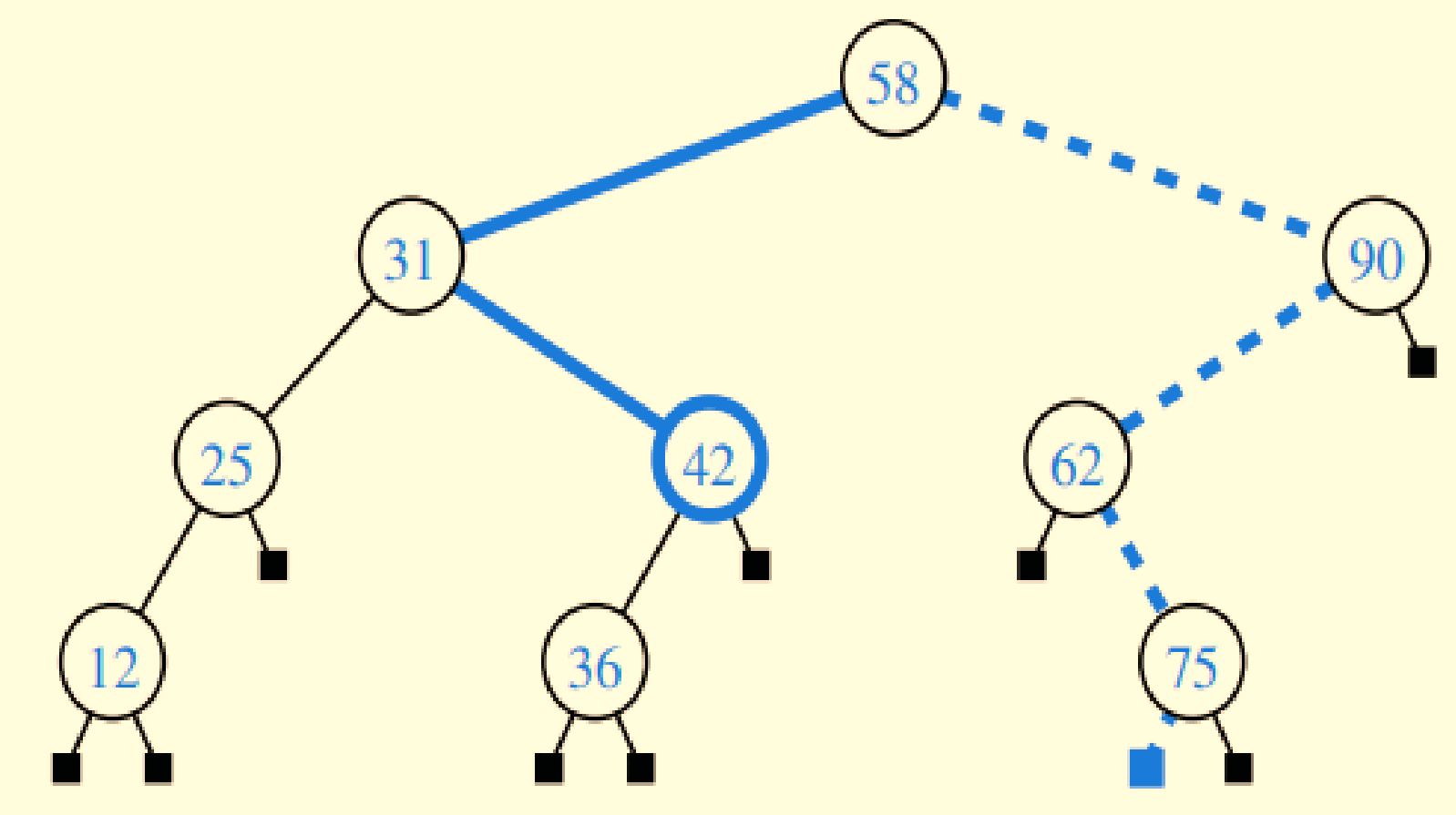
- An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree, defining a structure we call a binary search tree.

Property of Binary Search Tree:

- Position p stores an element of S, denoted as e(p).
- Elements stored in the left subtree of p (if any) are less than e(p).
- Elements stored in the right subtree of p (if any) are greater than e(p).

Binary Search Tree

The solid path is traversed when searching (successfully) for 42.
The dashed path is traversed when searching (unsuccessfully) for 70.

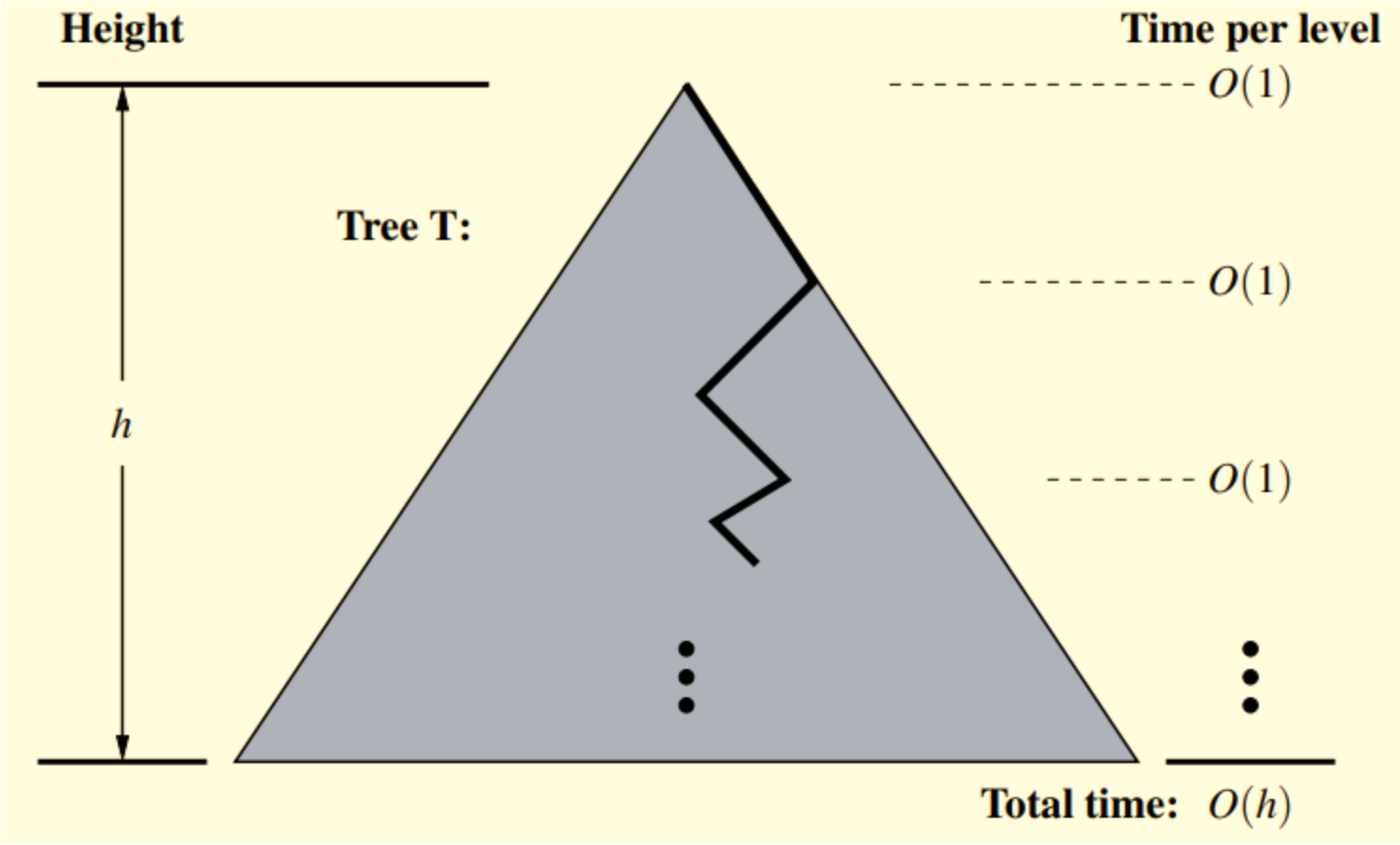


Binary Search Tree

```
private boolean searchRec(TreeNode root, int val) {
        if (root == null)
            return false;
        if (root.val == val)
            return true;
        if (val < root.val)
            return searchRec(root.left, val);
       return searchRec(root.right, val);
    }
```

Analysis of Binary Tree Search

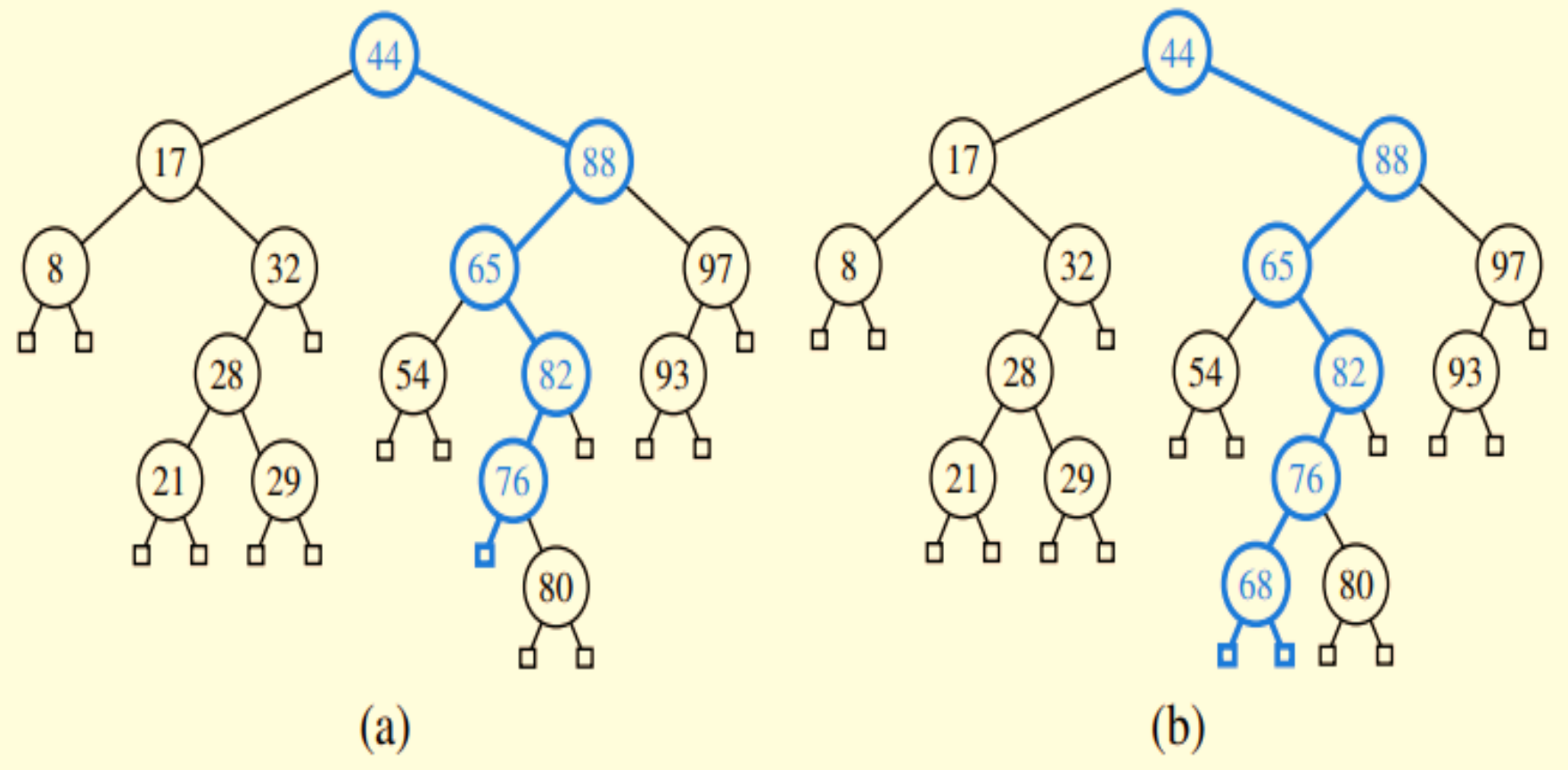
The overall search runs in **O(h) time**, where h is the height of the binary search tree T.



Binary Search Tree

Insert Operation:

Insertion of an entry with key 68 into the search tree.



Binary Search Tree

```
Node insertBinarySearchTree(Node root, int item) {
        if (root == null) {
                root = newNode(item);
                return root;          }
        if (item < root.data)
                root.left = insertBinarySearchTree(root.left, item);
        else if (item > root.data)
                root.right = insertBinarySearchTree(root.right, item);
        else
                System.out.println("Element already exists in BST.");
        return root;    }
```

Binary Search Tree

Delete Operation:

- Deletion of an entry from a binary search tree is a bit more complex than inserting a new entry.
- To delete an entry with key k, we begin to find the position p.
- If the search returns an external node, then there is no entry to remove.
- Otherwise, we distinguish between two cases,
    - If position p has one child
    - If position p has two children

Binary Search Tree

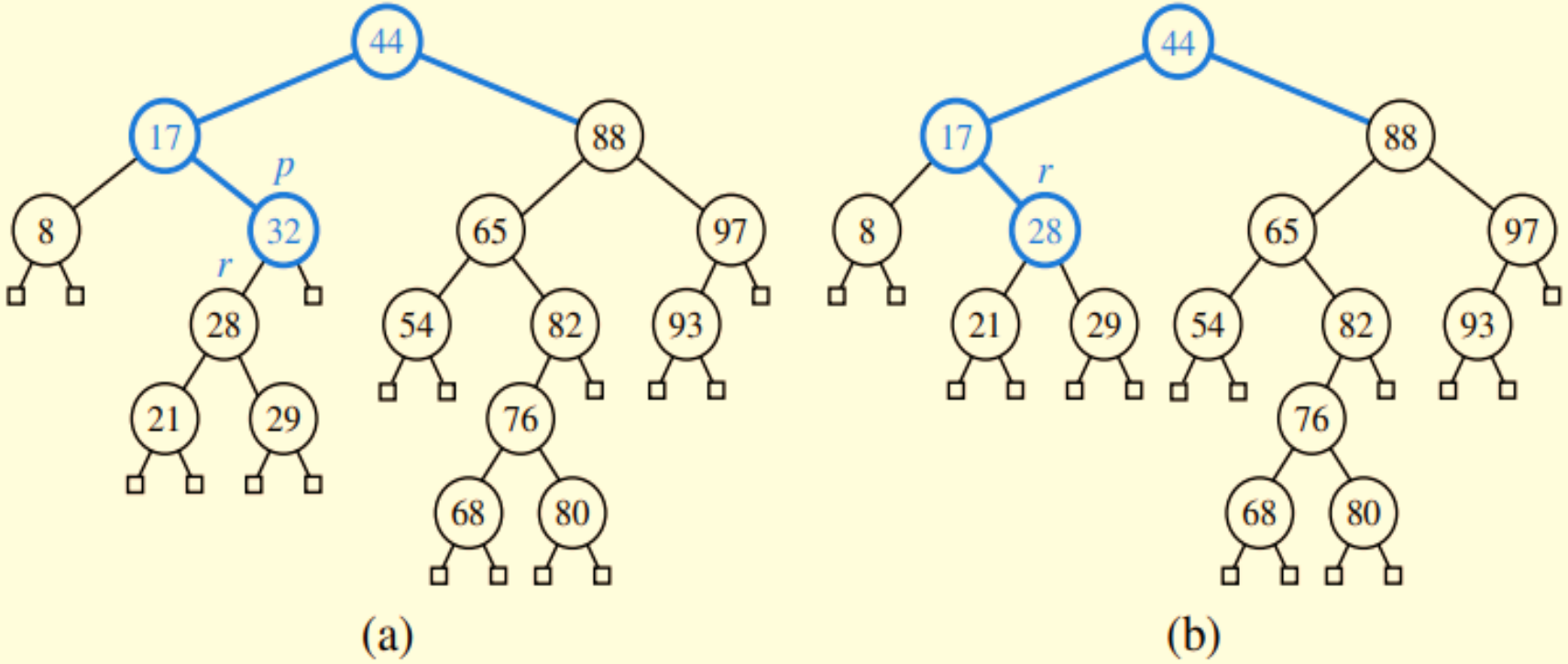


Figure 11.5: Deletion from the binary search tree of Figure 11.4b, where the entry to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

Binary Search Tree

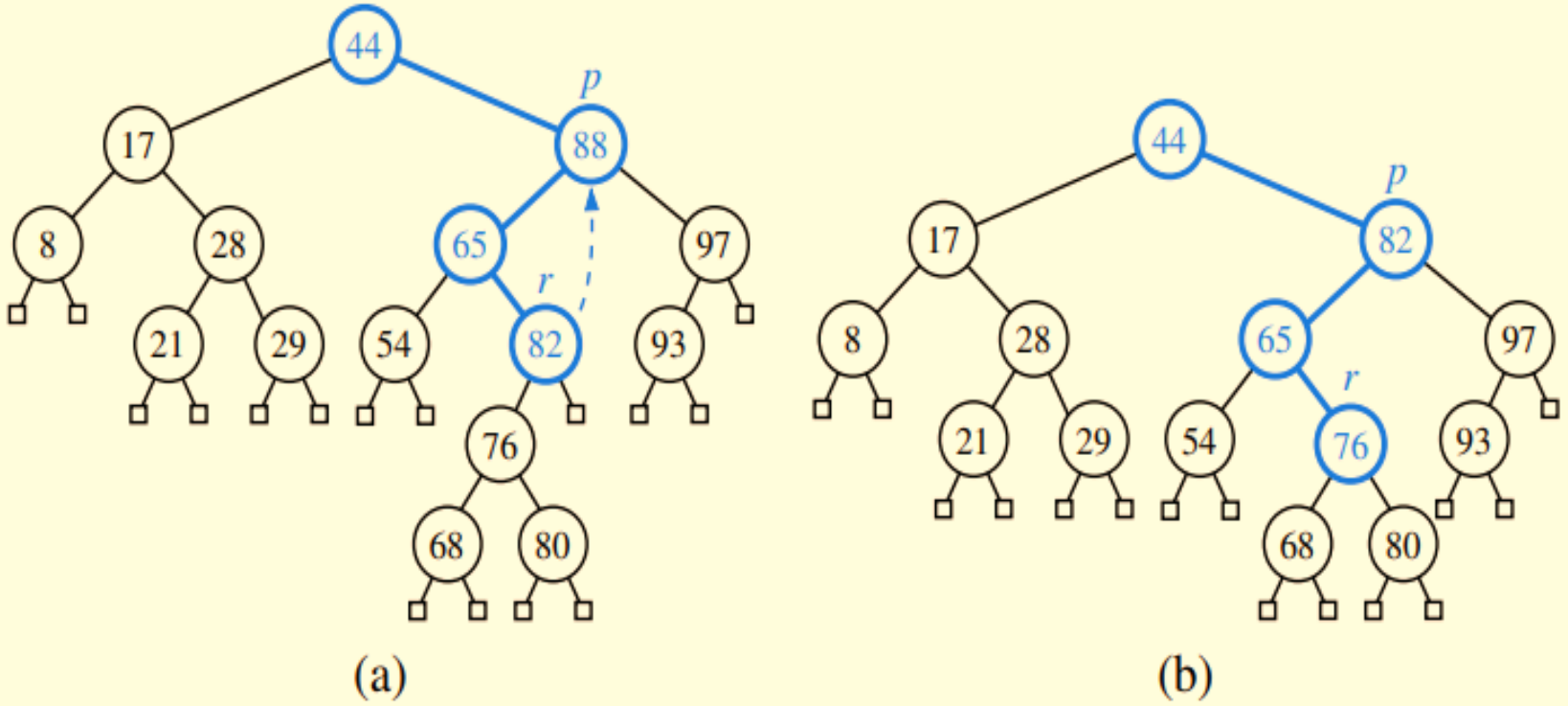


Figure 11.6: Deletion from the binary search tree of Figure 11.5b, where the entry to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

Binary Search Tree

```
private TreeNode deleteRec(TreeNode root, int key) {
        if (root == null)           return root;
        if (key < root.val)
            root.left = deleteRec(root.left, key);
        else if (key > root.val)
            root.right = deleteRec(root.right, key);
else {  if(root.left == null && root.right == null)
                    root = null;
        else if (root.left == null)          return root.right;
            else if (root.right == null)       return root.left;
             Node temp = minValue(root.right);
          root.val=temp.val;
            root.right = deleteRec(root.right, root.val);        }
        return root;    }
```

Binary Search Tree

```
private int minValue(TreeNode root) {
        int minVal = root.val;
        while (root.left != null) {
            minVal = root.left.val;
            root = root.left;
        }
        return minVal;
    }
```

Balanced Binary Search Tree

The standard binary search tree supports **O(n)** worst-case time, because some sequences of operations may lead to an unbalanced tree with height proportional to n.

A balanced binary tree is also known as height balanced tree.

Examples:

- AVL Tree
- Red - Black Tree
- B - Tree
- B+ Tree

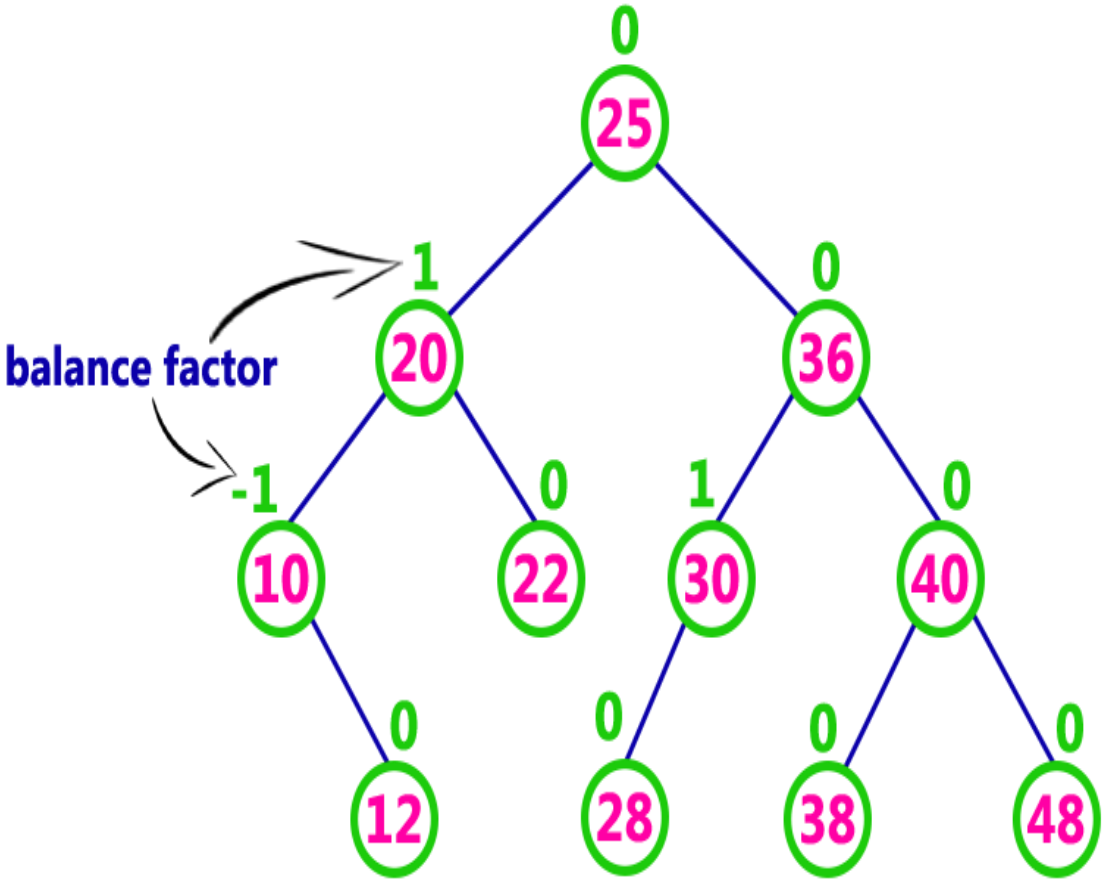
AVL Tree

- Adelson-Velsky and Landis.
- AVL Trees prevent the tree from becoming lopsided or skewed.
- The balance factor of every node in the tree is either -1, 0 or +1.
- Balance factor: The difference between the heights of left subtree and right subtree.

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

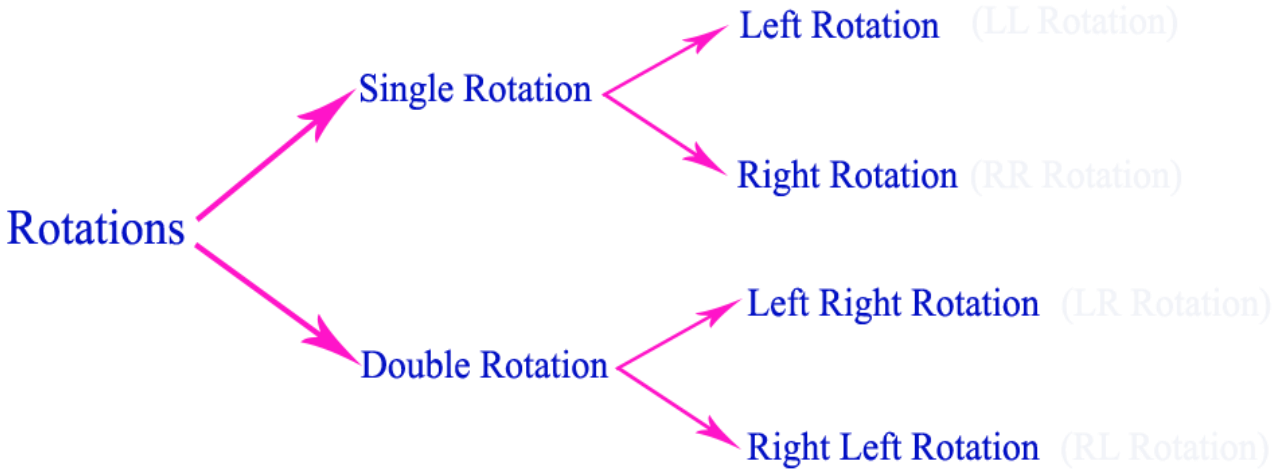
AVL tree

Example:



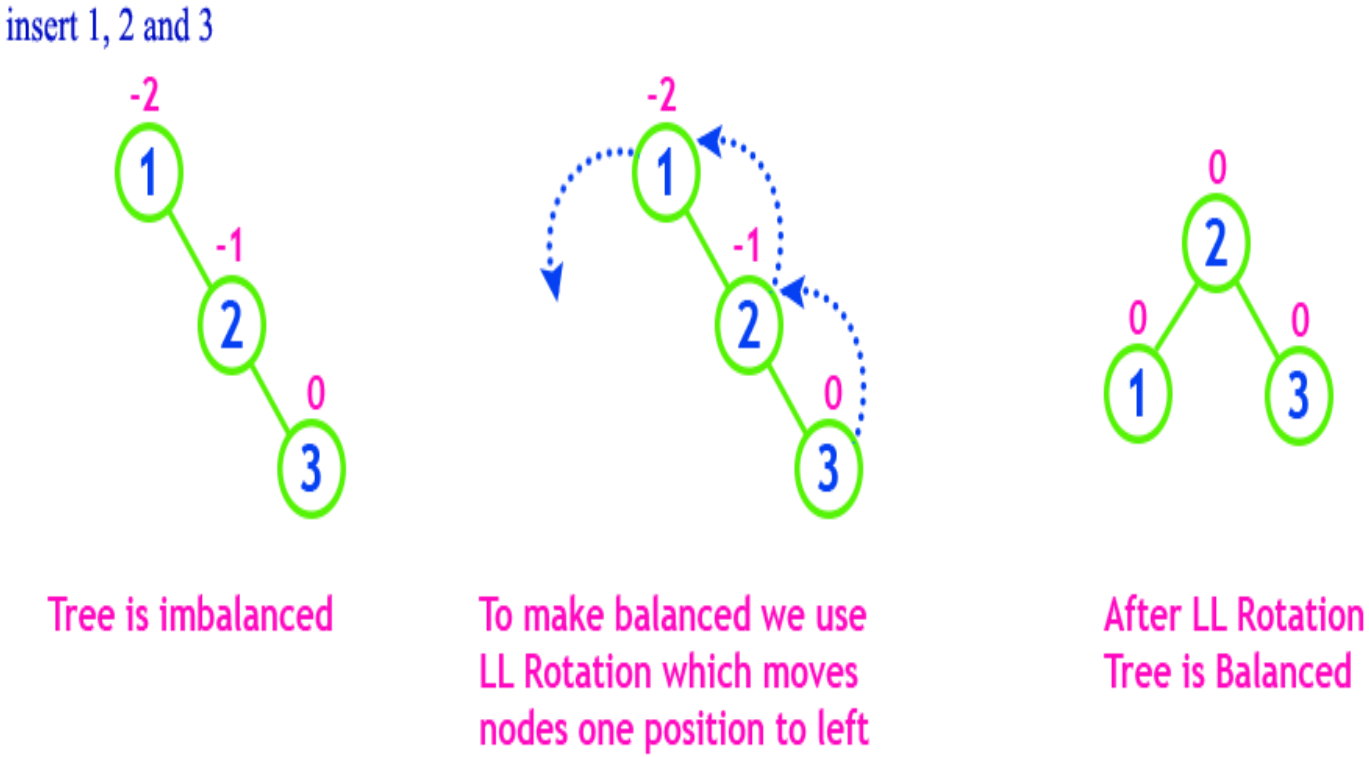
AVL Tree Rotations

- After performing operations like insertion and deletion, if the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.
- There are four rotations and they are classified into two types.



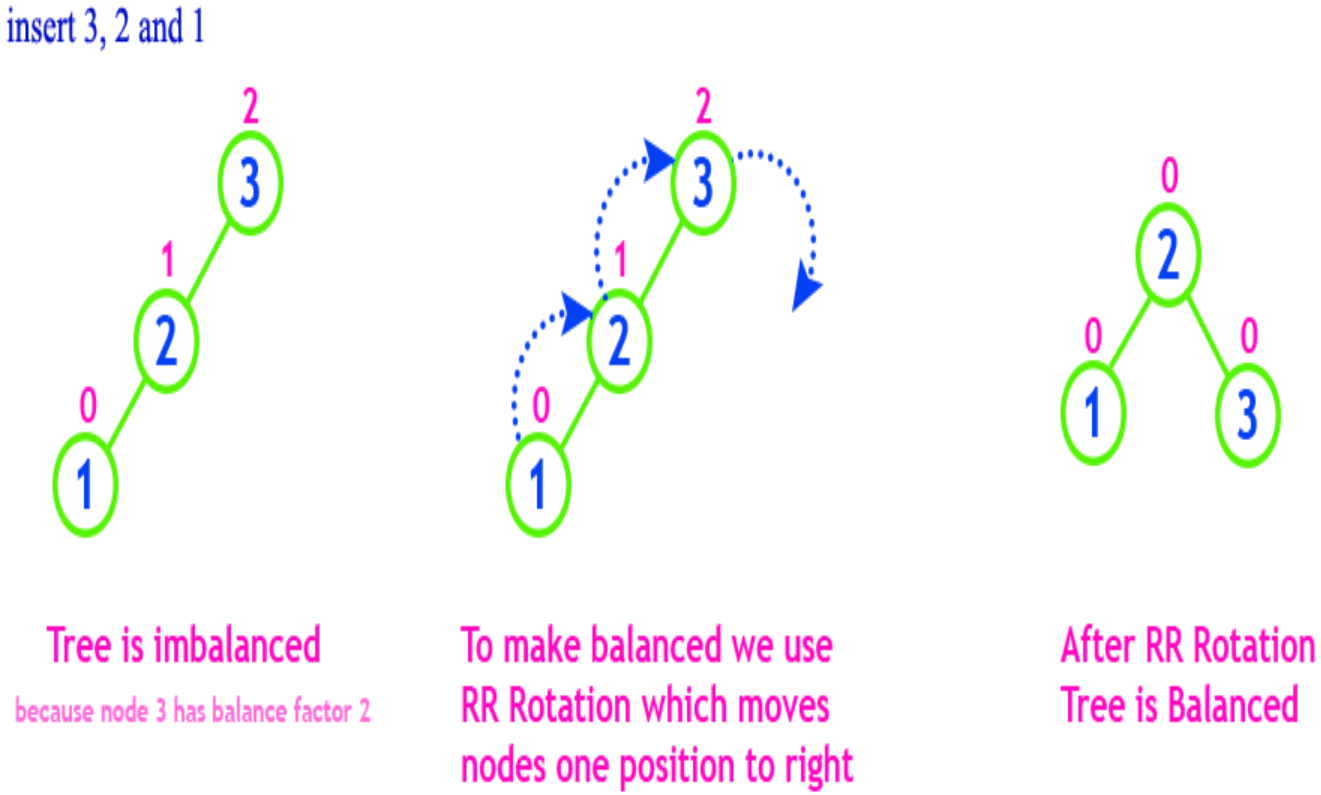
Single Left Rotation (LL Rotation)

- Every node moves one position to left from the current position.



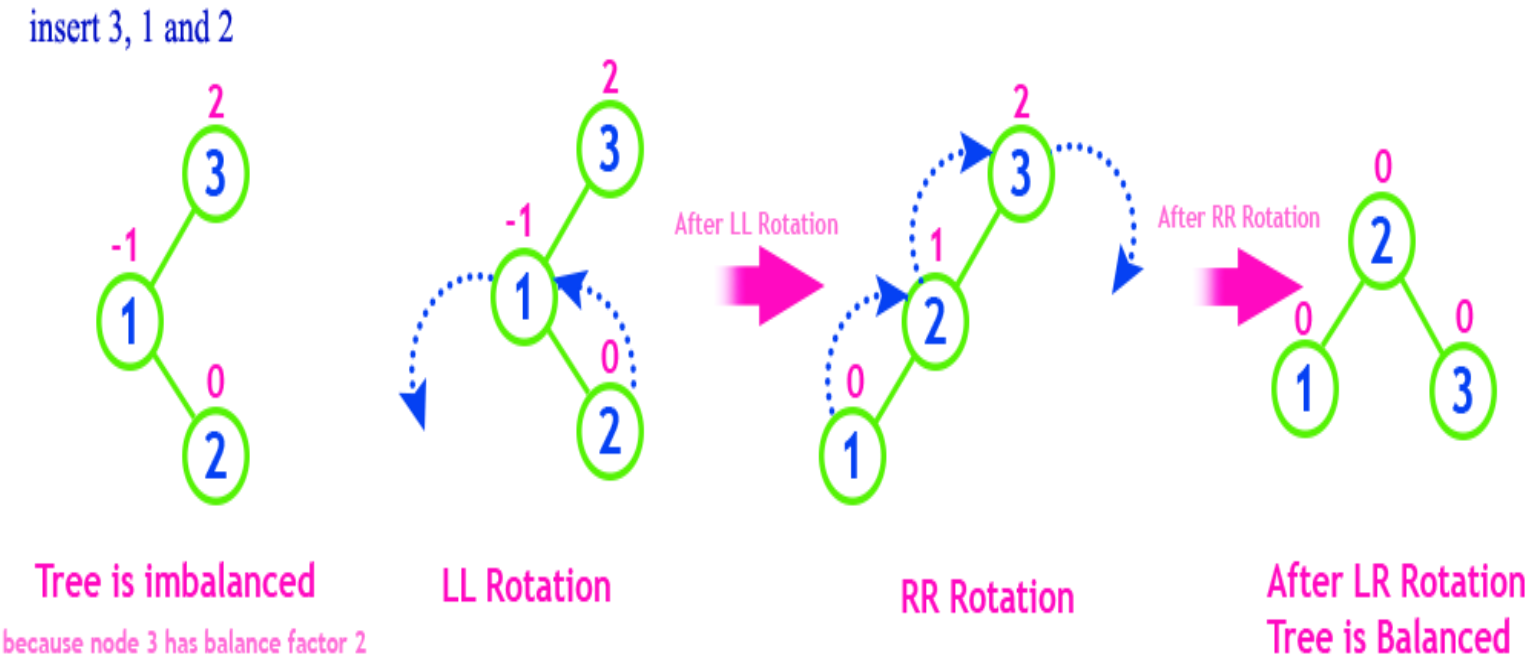
Single Right Rotation (RR Rotation)

- Every node moves one position to right from the current position.



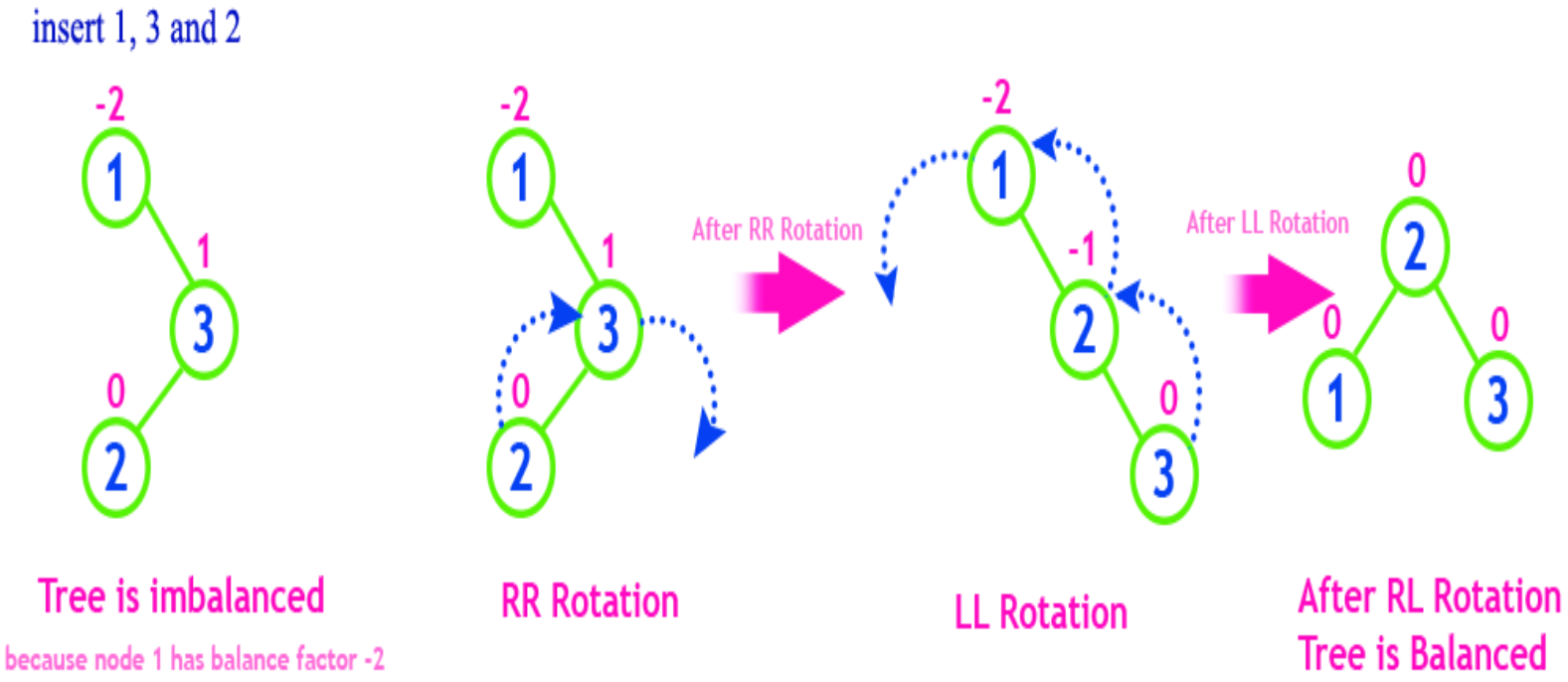
Left Right Rotation (LR Rotation)

- sequence of single left rotation followed by a single right rotation



Right Left Rotation (RL Rotation)

- sequence of single right rotation followed by a single left rotation



Operations on an AVL Tree

- The following operations are performed on AVL tree...
  - Search
  - Insertion
  - Deletion

Search Operation in AVL Tree

- The search operation in the AVL tree is similar to the search operation in a Binary search tree.
- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left subtree.
- Step 6 - If search element is larger, then continue the search process in right subtree.

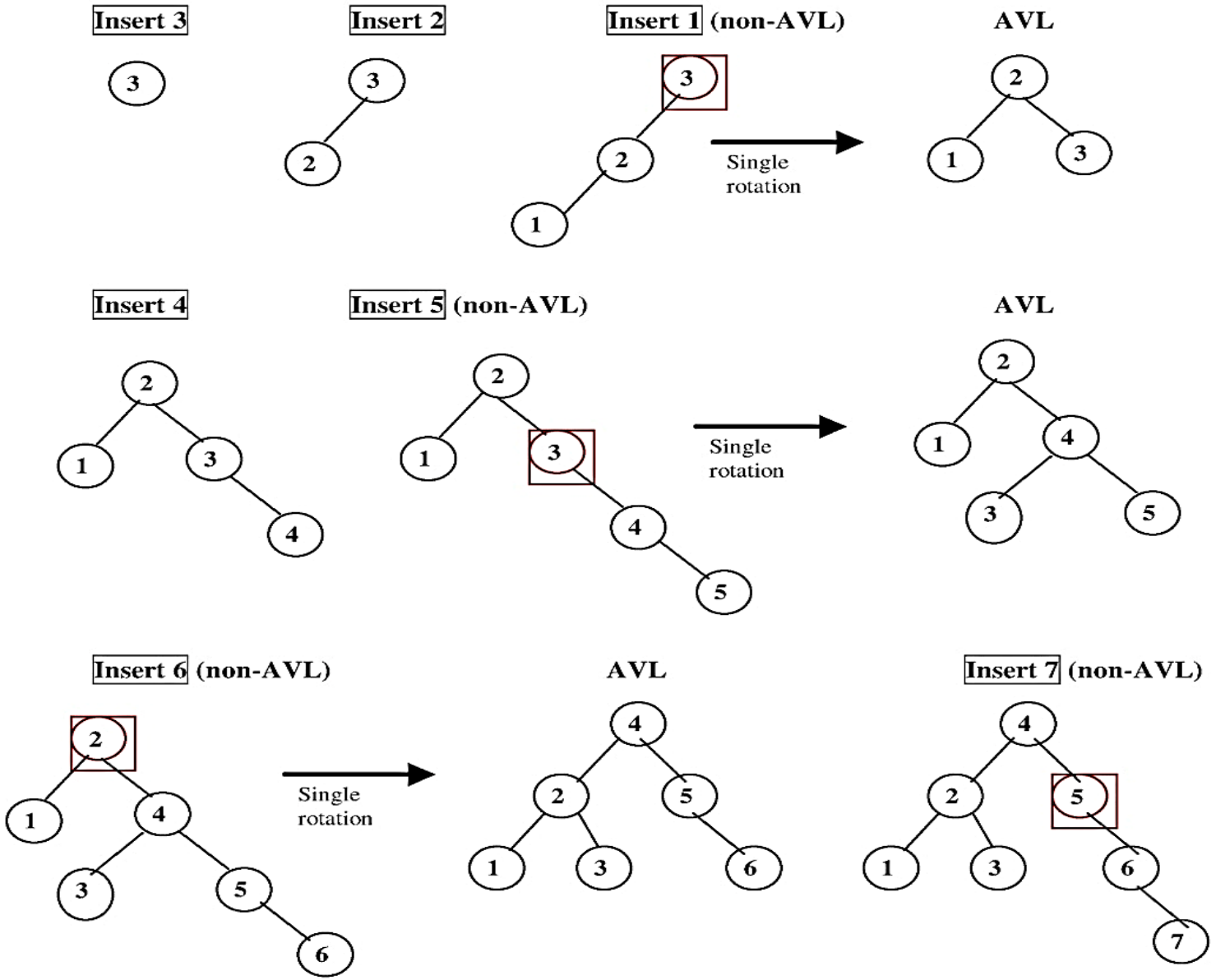
Search Operation in AVL Tree

- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insert Operation in AVL Tree

- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the Balance Factor of every node.
- Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
- Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

Insert Operation in AVL Tree



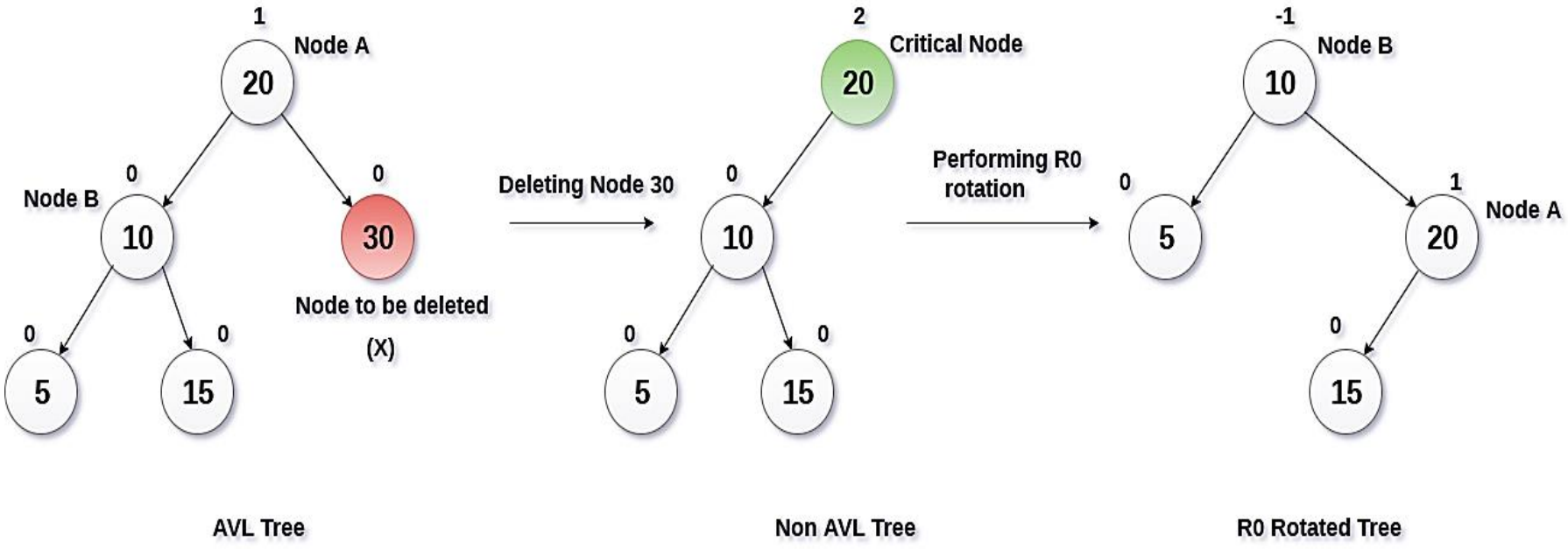
Delete Operation in AVL Tree

- The deletion operation in AVL Tree is similar to deletion operation in BST.
- But after every deletion operation, we need to check with the Balance Factor condition.
- If the tree is balanced after deletion go for next operation
- otherwise perform suitable rotation to make the tree Balanced.

Example:

Delete values 5, 8 and 2 in the above tree

Delete Operation in AVL Tree



Program for AVL Tree

```
class AVLTree {
    Node root;

int height(Node N) {
        if (N == null)
            return 0;
        return N.height;    }

int max(int a, int b) {
        return (a > b) ? a : b;    }

Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;
        x.right = y;
        y.left = T2;
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;
        return x;    }
```

Program for AVL Tree

```
Node leftRotate(Node x) {
      Node y = x.right;
      Node T2 = y.left;
      y.left = x;
      x.right = T2;
      x.height = max(height(x.left), height(x.right)) + 1;
      y.height = max(height(y.left), height(y.right)) + 1;
      return y;    }

  int getBalance(Node N) {
      if (N == null)
          return 0;
      return height(N.left) - height(N.right);
  }
```

Program for AVL Tree

```
Node insert(Node node, int key) {
    if (node == null)
        return (new Node(key));
    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else          return node;
    node.height = 1 + max(height(node.left), height(node.right));
    int balance = getBalance(node);
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);       }
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);       }
    return node;    }
```

Program for AVL Tree

```
Node minValueNode(Node node) {
        Node current = node;
        while (current.left != null)
            current = current.left;
        return current;     }


void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }     }
```

Program for AVL Tree

```
Node deleteNode(Node root, int key) {
    if (root == null)
        return root;
    if (key < root.key)
        root.left = deleteNode(root.left, key);
    else if (key > root.key)
        root.right = deleteNode(root.right, key);
    else {
        if ((root.left == null) || (root.right == null)) {
            Node temp = null;
            if (temp == root.left)
                temp = root.right;
            else            temp = root.left;
            if (temp == null) {
                temp = root;
                root = null;            }
else                root = temp;        }
else {
            Node temp = minValueNode(root.right);
            root.key = temp.key;
            root.right = deleteNode(root.right, temp.key);        }    }
```

Program for AVL Tree

```
if (root == null)
        return root;
    root.height = max(height(root.left), height(root.right)) + 1;
    int balance = getBalance(root);
    if (balance > 1 && getBalance(root.left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root.left) < 0) {
        root.left = leftRotate(root.left);
        return rightRotate(root);       }

    if (balance < -1 && getBalance(root.right) <= 0)
        return leftRotate(root);

    if (balance < -1 && getBalance(root.right) > 0) {
        root.right = rightRotate(root.right);
        return leftRotate(root);       }

  return root;    }
```

Program for AVL Tree

```
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        AVLTree tree = new AVLTree();
        System.out.println("AVL Tree Operations");
        char ch;
        do {
            System.out.println("\n1. Insert");
            System.out.println("2. Delete");
            System.out.println("3. Display");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    System.out.println("Enter integer element to insert:");
                    tree.root = tree.insert(tree.root, scanner.nextInt());
                    break;
                case 2:
                    System.out.println("Enter integer element to delete:");
                    tree.root = tree.deleteNode(tree.root, scanner.nextInt());
                    break;
```

Program for AVL Tree

```
case 3:
                System.out.println("Preorder traversal of constructed AVL tree is :");
                tree.preOrder(tree.root);
                break;
            default:
                System.out.println("Wrong Entry \n ");
                break;
        }
        System.out.println("\nDo you want to continue (Type y or n) \n");
        ch = scanner.next().charAt(0);
    } while (ch == 'Y' || ch == 'y');
  }
}
```

Splay Tree

- A self-balancing binary search tree.
- Brings the recently accessed item to root of the tree, so that if we access the same item again then it takes only O(1) time.
- Every operation (insertion, deletion or search) on a splay tree performs the splaying operation.

Splay Tree

Insertion:

- Inserts the new element as it inserted into the binary search tree.
- the newly inserted element is splayed so that it is placed at root of the tree.

Deletion:

- First splays the element to be deleted there by bringing it to the root and then perform the deletion operation.

Splay Tree

Search:

- The search operation in a splay tree is search the element using binary search process then splay the searched element so that it placed at the root of the tree.

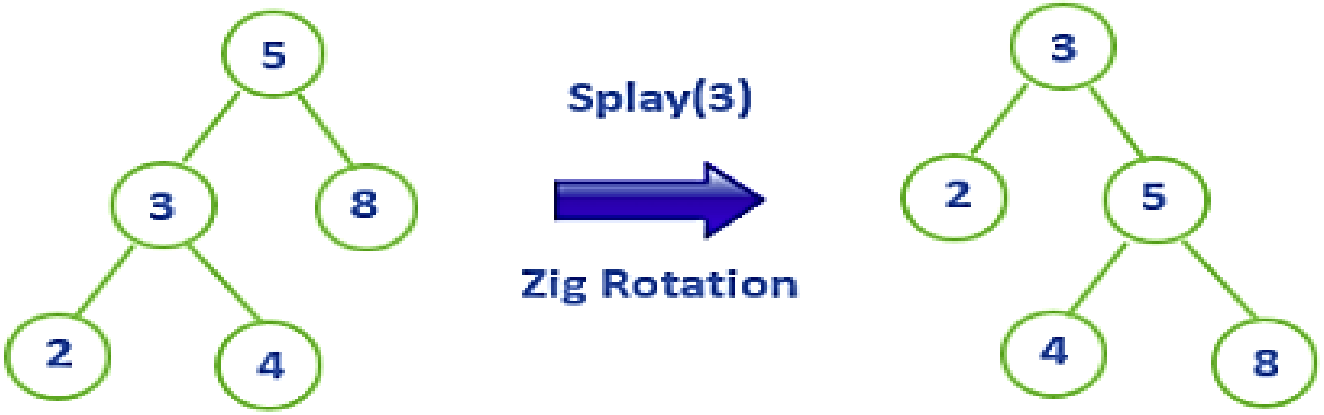
Splay Tree

- To splay an element to the root, rotations that are similar to that of an AVL tree.

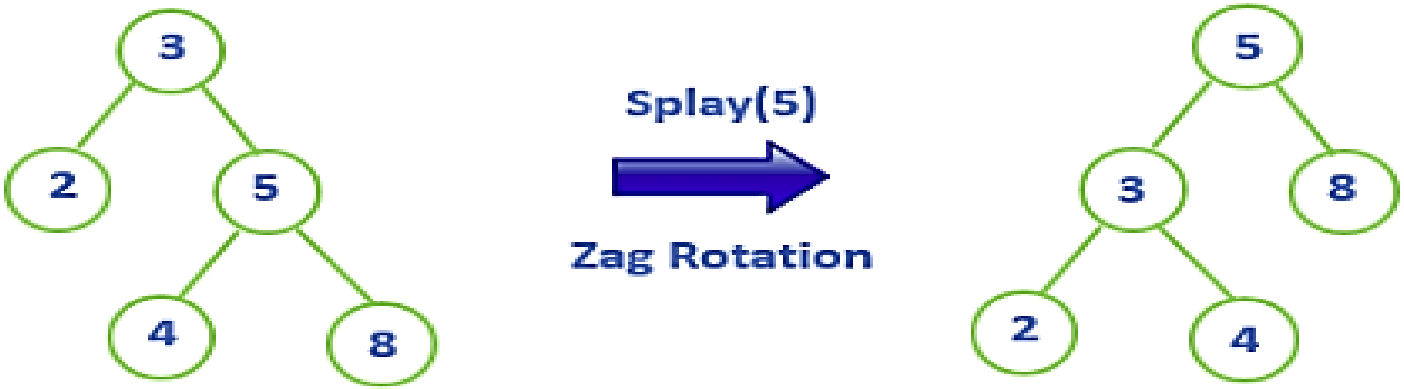
Rotations of splay tree

- Zig rotation
- Zag rotation
- Zig-Zig rotation
- Zag-Zag rotation
- Zig-Zag rotation
- Zag-Zig rotation

Splay Tree

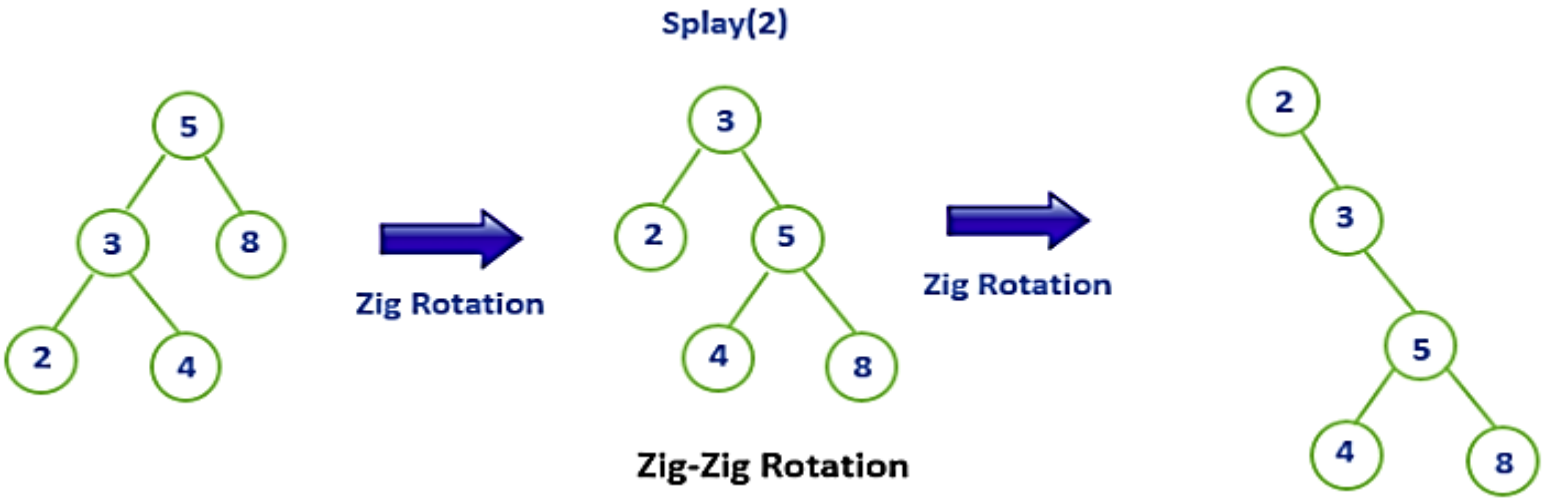


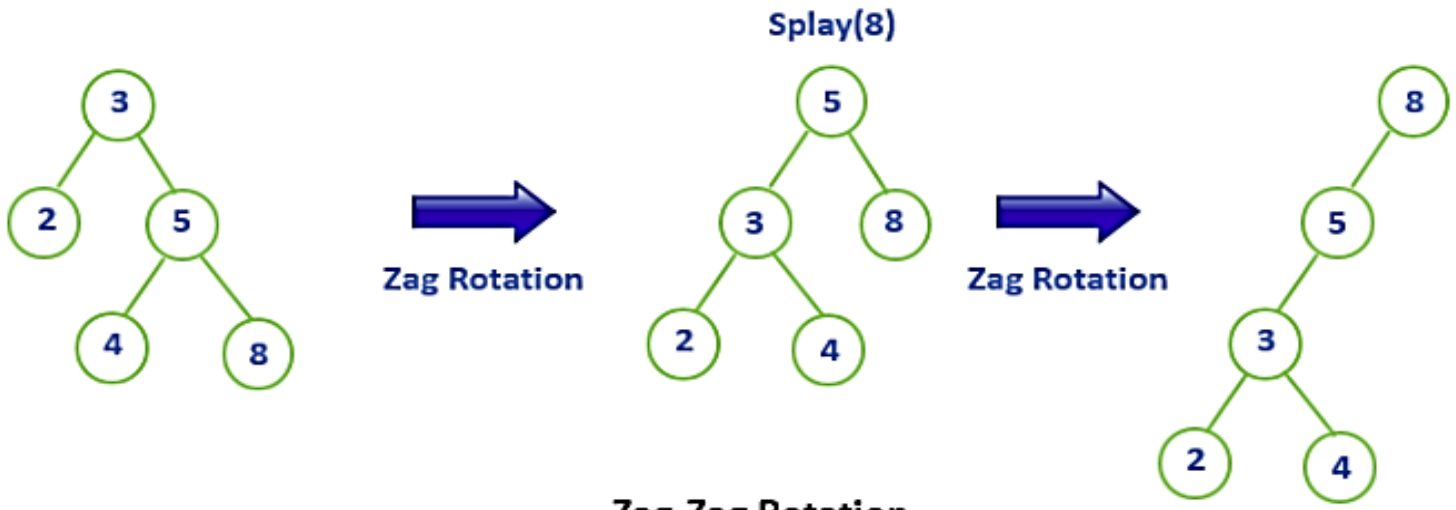
Zig Rotation



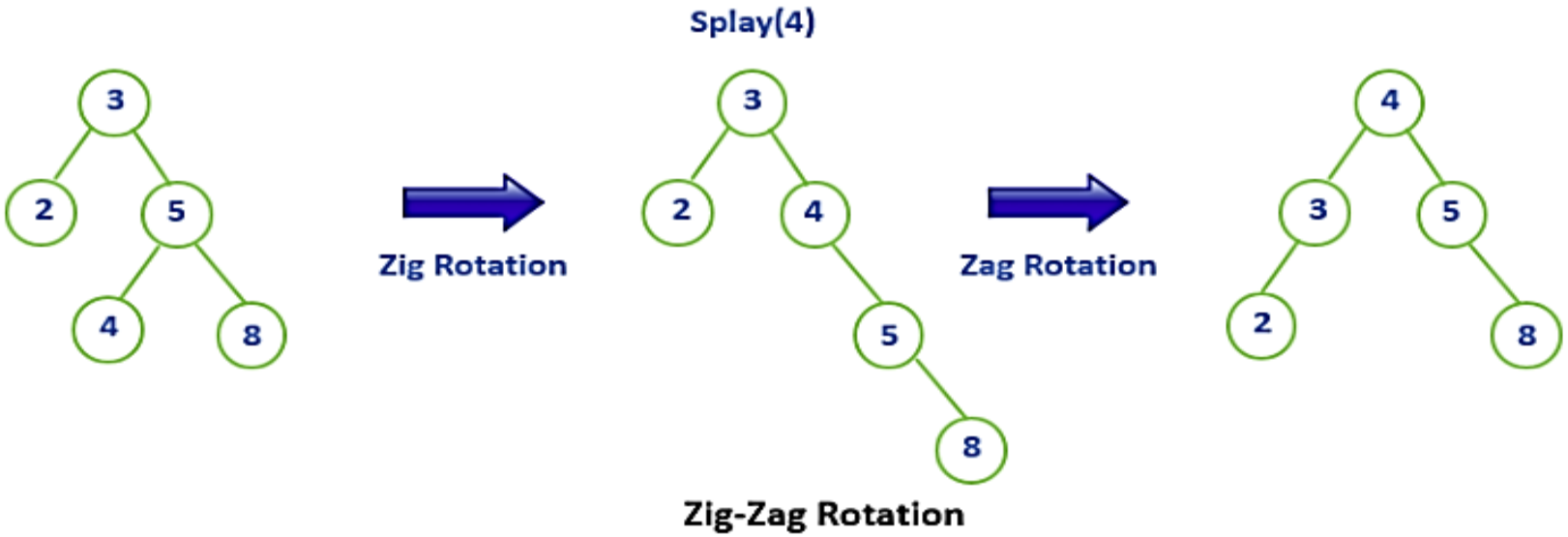
Zag Rotation

Splay Tree

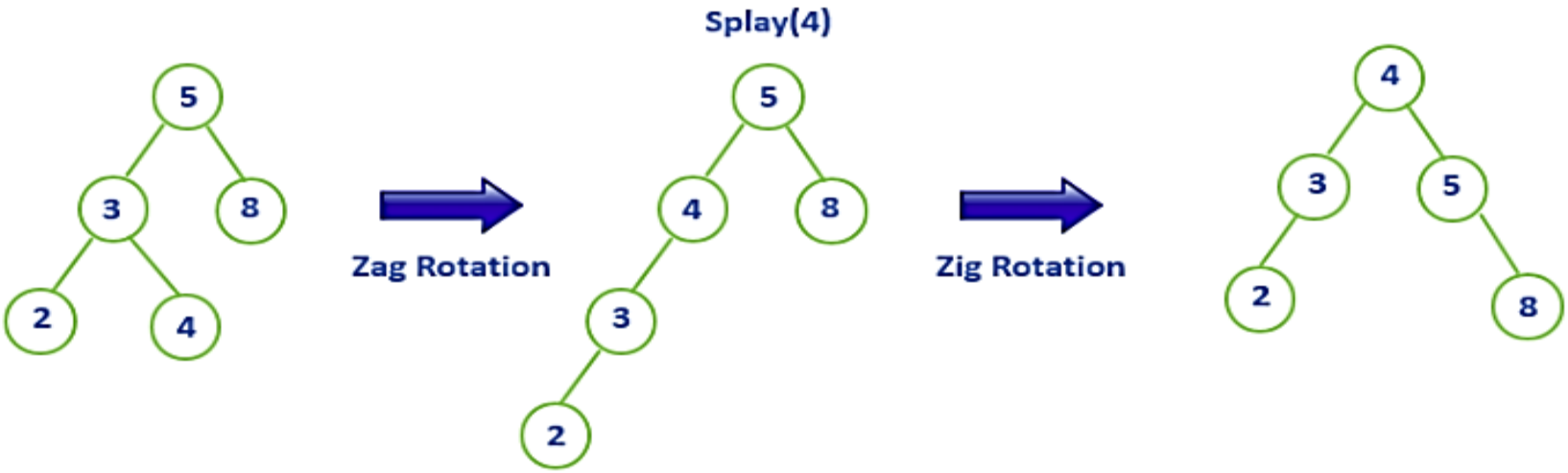

Splay(2)
5
3
8
2
4
Zig Rotation
3
2
5
4
8
Zig Rotation
2
3
5
4
8
Zig-Zig Rotation


Splay(8)
3
2
5
4
8
Zag Rotation
5
3
8
2
4
Zag Rotation
8
5
3
2
4
Zag-Zag Rotation

Splay Tree



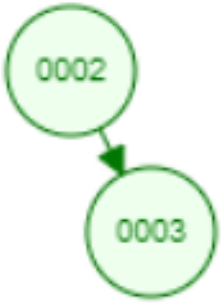
Zig-Zag Rotation

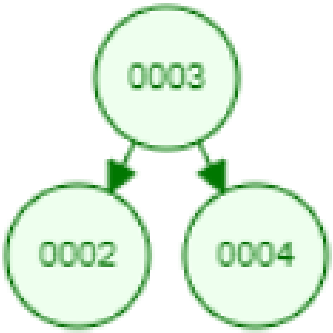


Zag-Zig Rotation

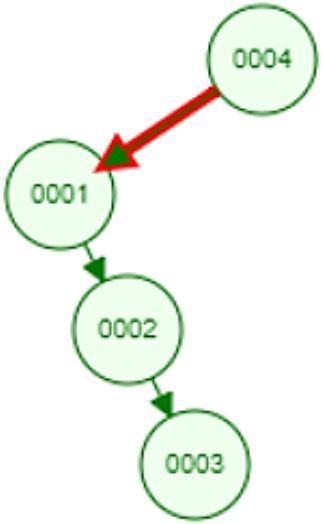
Splay Tree Insertion

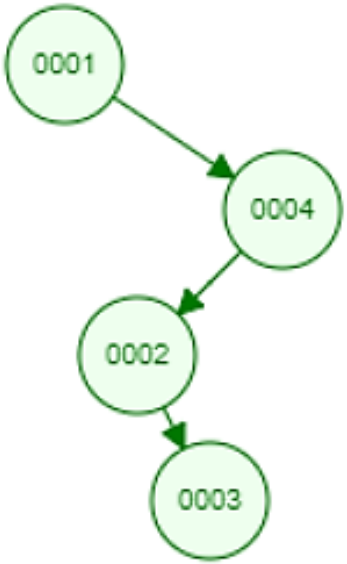
Insertion: 2, 3, 4, 1, 6, 5.

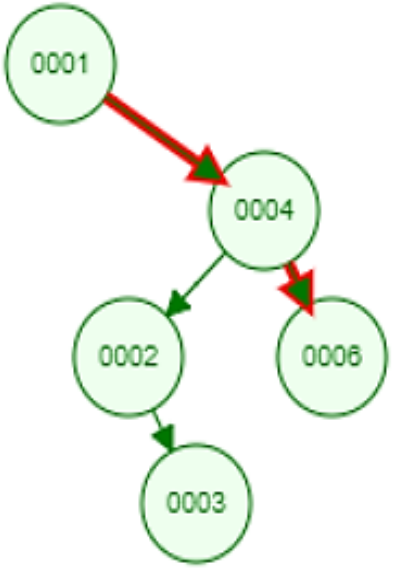


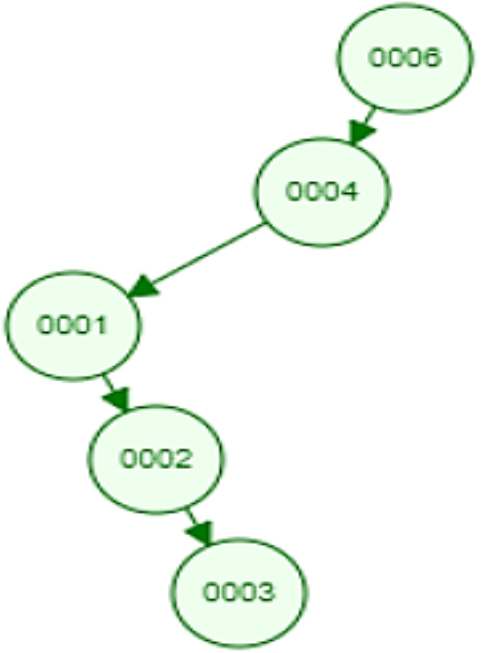


Splay Tree

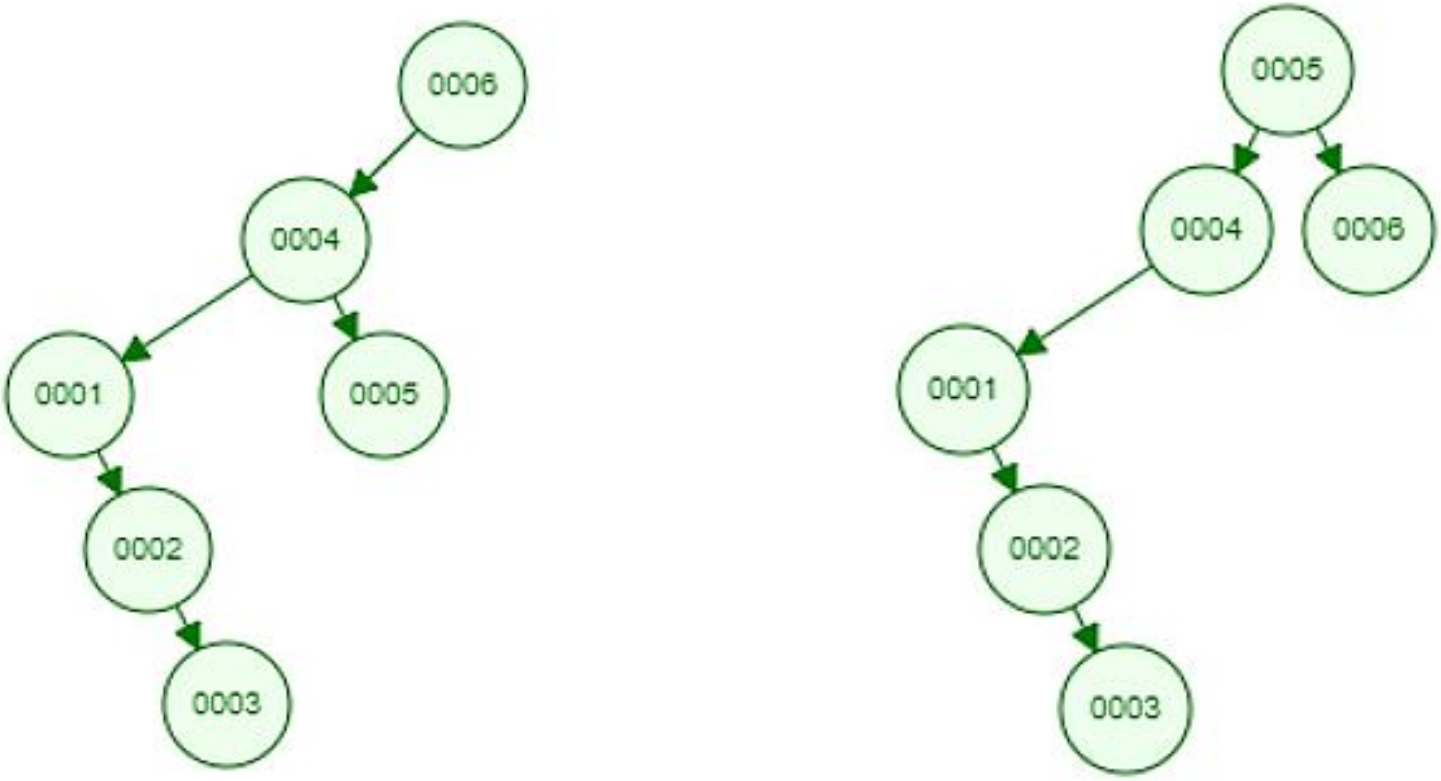

0004
0001
0002
0003


0001
0004
0002
0003


0001
0004
0002
0006
0003


0006
0004
0001
0002
0003

Splay Tree



Splay Tree Deletion

Types of Deletions:

There are two types of deletions in the splay trees,

- Bottom-up splaying
- Top-down splaying

Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the parent of the deleted node.

Splay Tree Deletion

Types of Deletion:

There are two types of deletion in the splay trees,

- Bottom-up splaying
- Top-down splaying

Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the parent of the deleted node.

Splay Tree Deletion

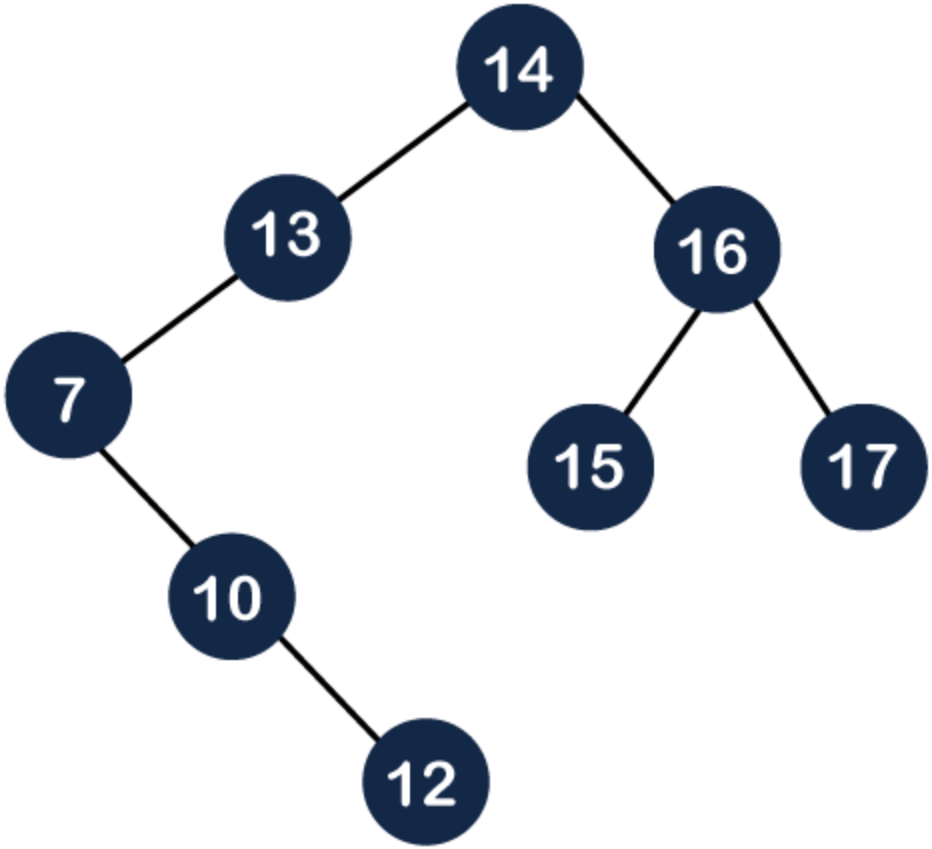
Top-down splaying

In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

For joining, Consider the maximum node value from the left subtree and make it as root node.

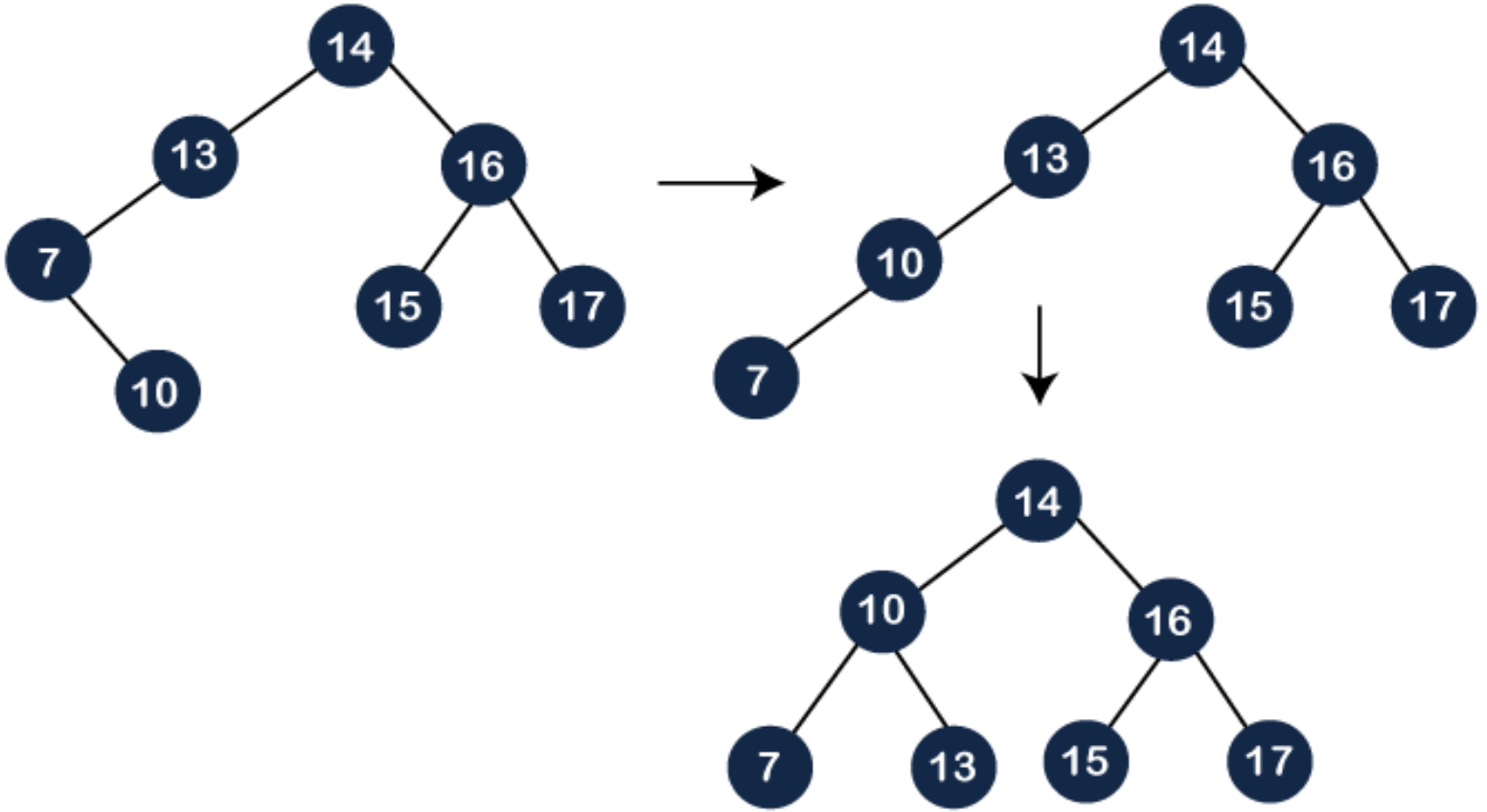
Splay Tree Bottom-up Splaying

Example: Delete 12, 14



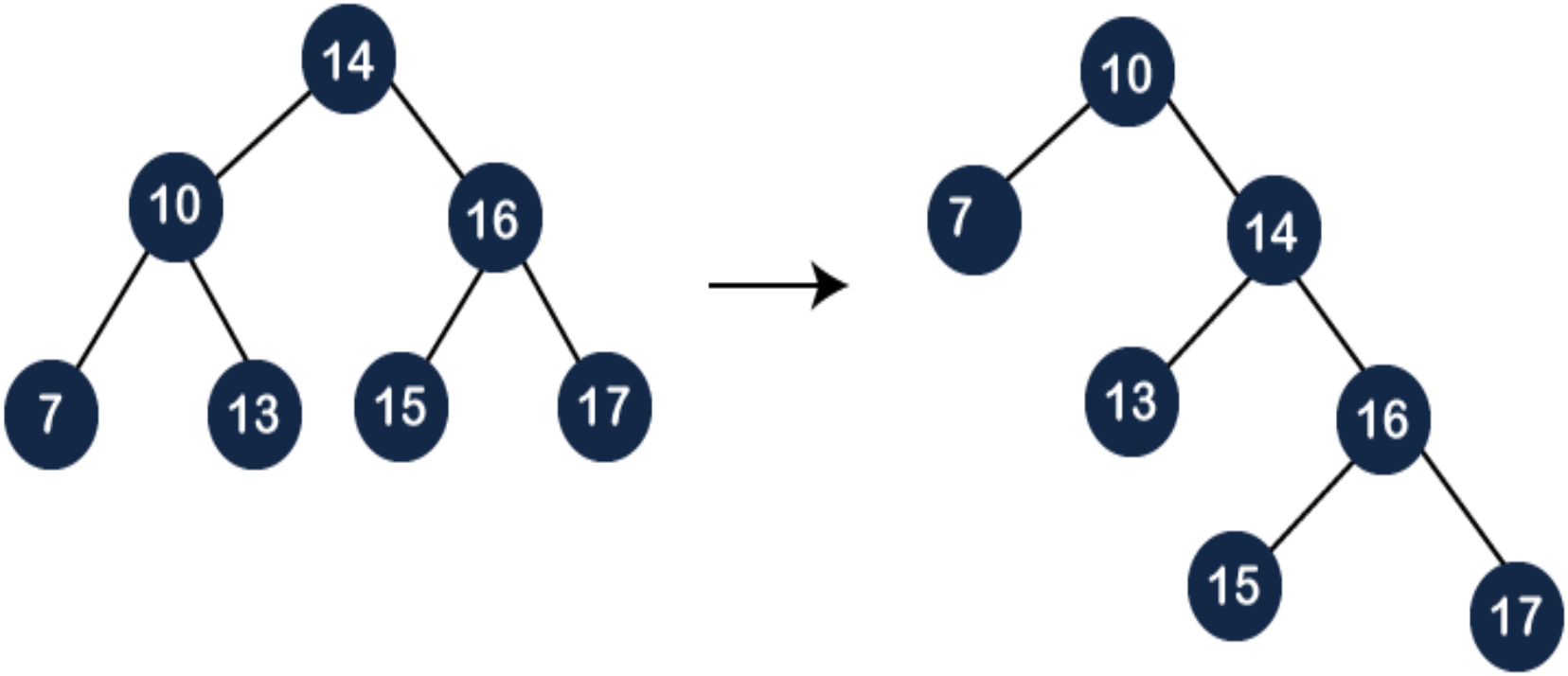
Splay Tree Bottom-up Splaying

Delete 12 : As 12 is a leaf node, simply delete the node and splay the parent of the deleted node, i.e., 10



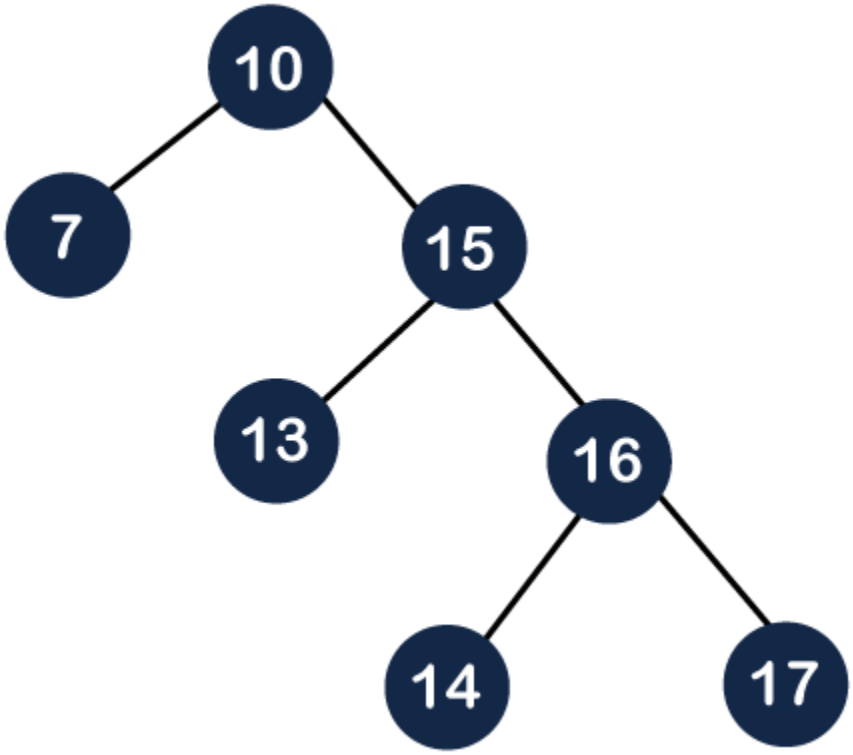
Splay Tree Bottom-up Splaying

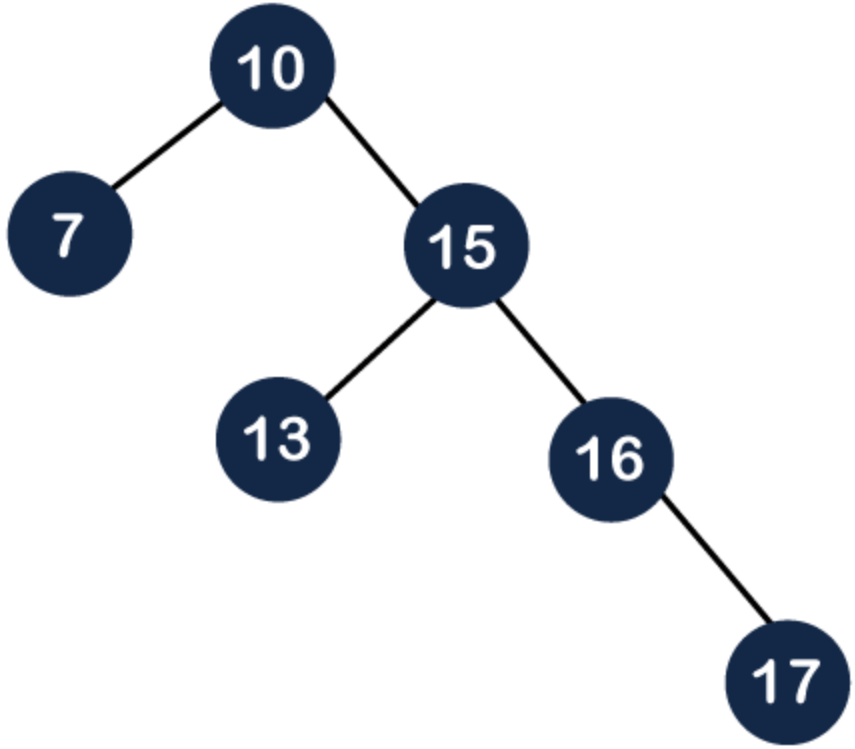
10 becomes the root node now.



Splay Tree Bottom-up Splaying

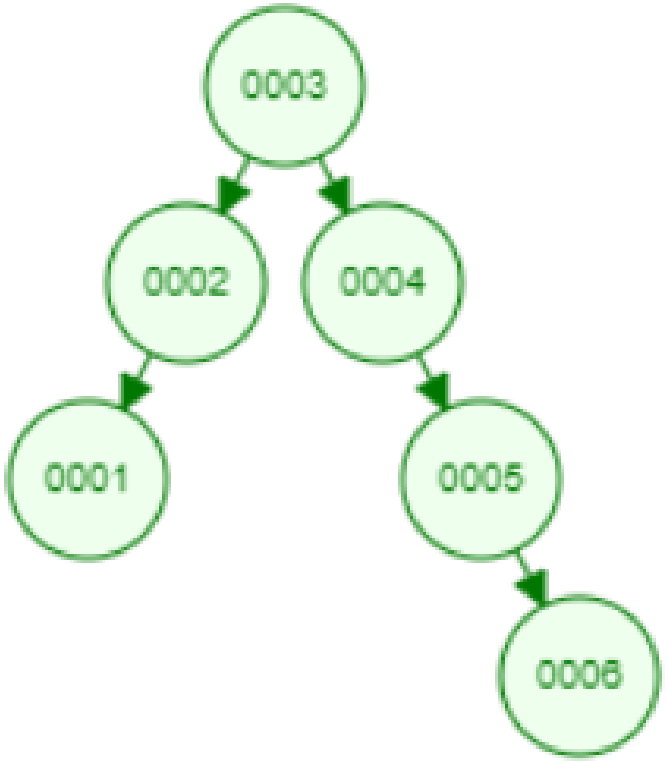
Delete 14: Replace the value of the node either using inorder predecessor or inorder successor as it is internal node.





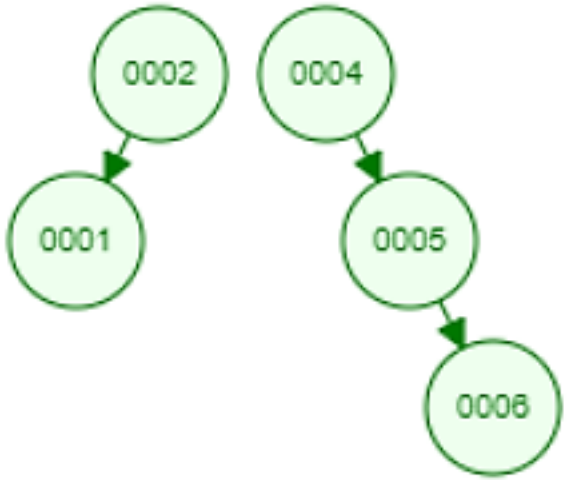
Splay Tree Top - Down Splaying

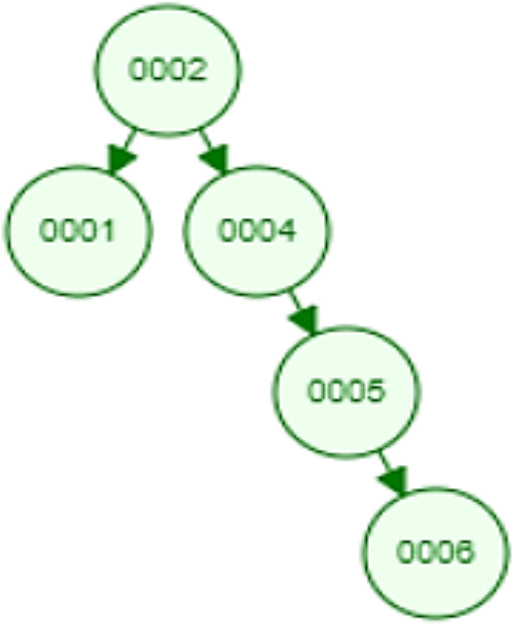
Delete 3



Splay Tree Top - Down Splaying

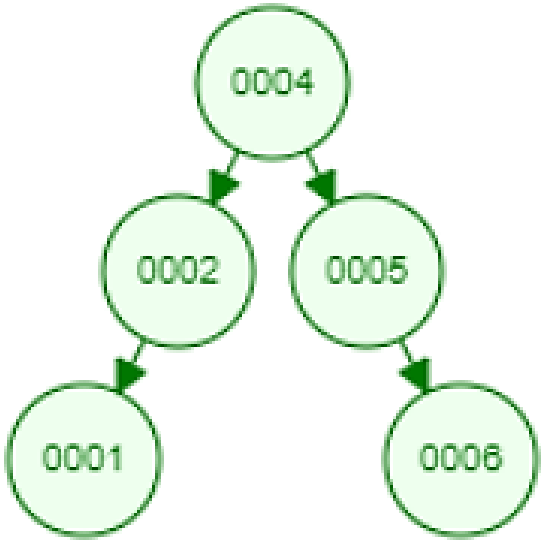
Deletion: 3,

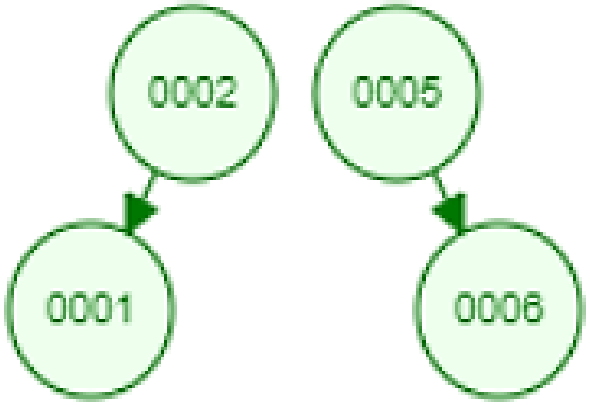


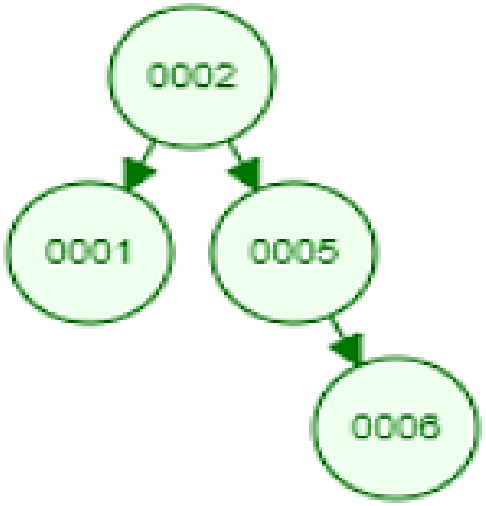


Splay Tree Top - Down Splaying

Deletion: 4

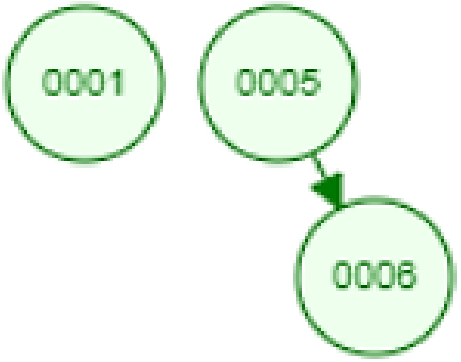


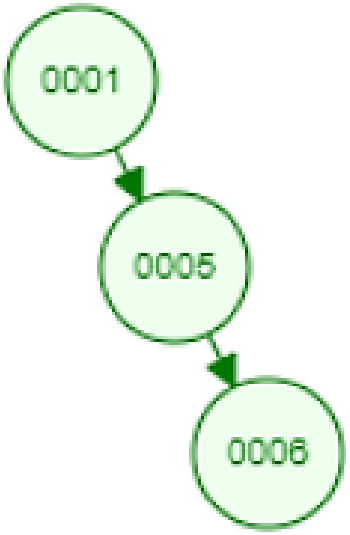




Splay Tree Top - Down Splaying

Deletion: 2

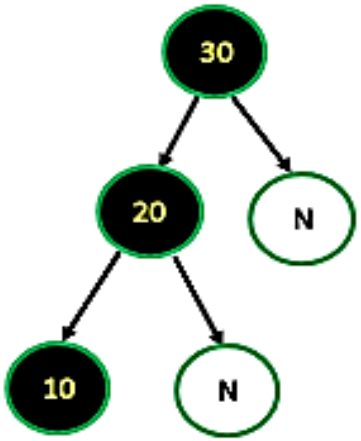




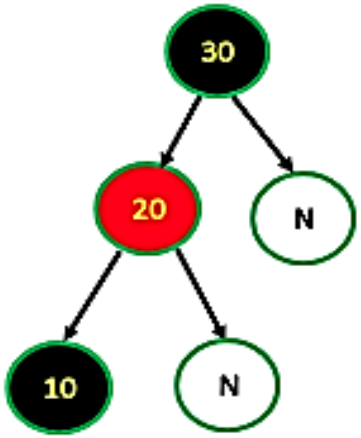
Red - Black Tree

- A self-balancing binary search tree.
- All the nodes follows the following properties.
- **Property 1:** Every node has a color either red or black.
- **Property 2:** Root node of the tree is always black.
- **Property 3:** There are no two adjacent red nodes. In other words, a red node cannot have a red parent or red child.
- **Property 4:** All paths from the root node to a leaf node has equal number of black nodes.

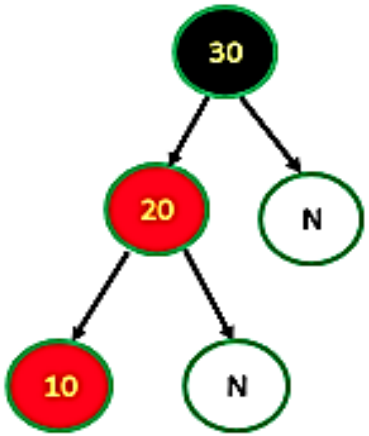
Red - Black Tree



Violates property 4

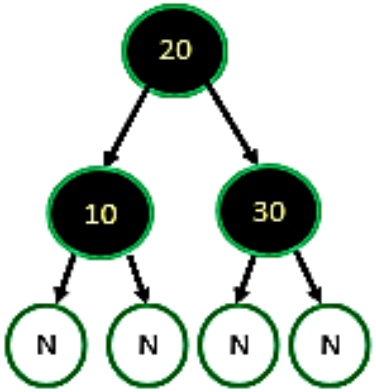


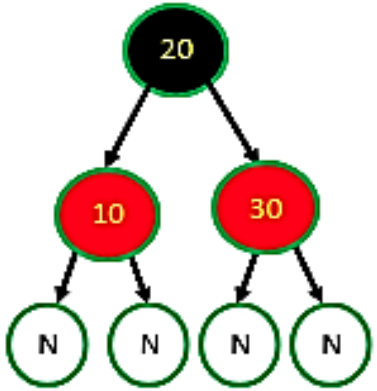
Violates property 4



Violates property 3

Only possible Red-Black trees :





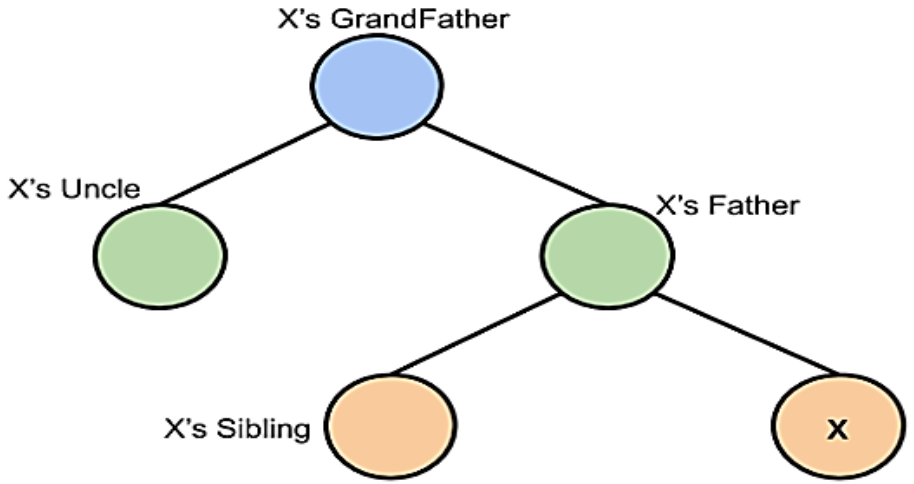
N-Represents NULL

Red - Black Tree Insertion

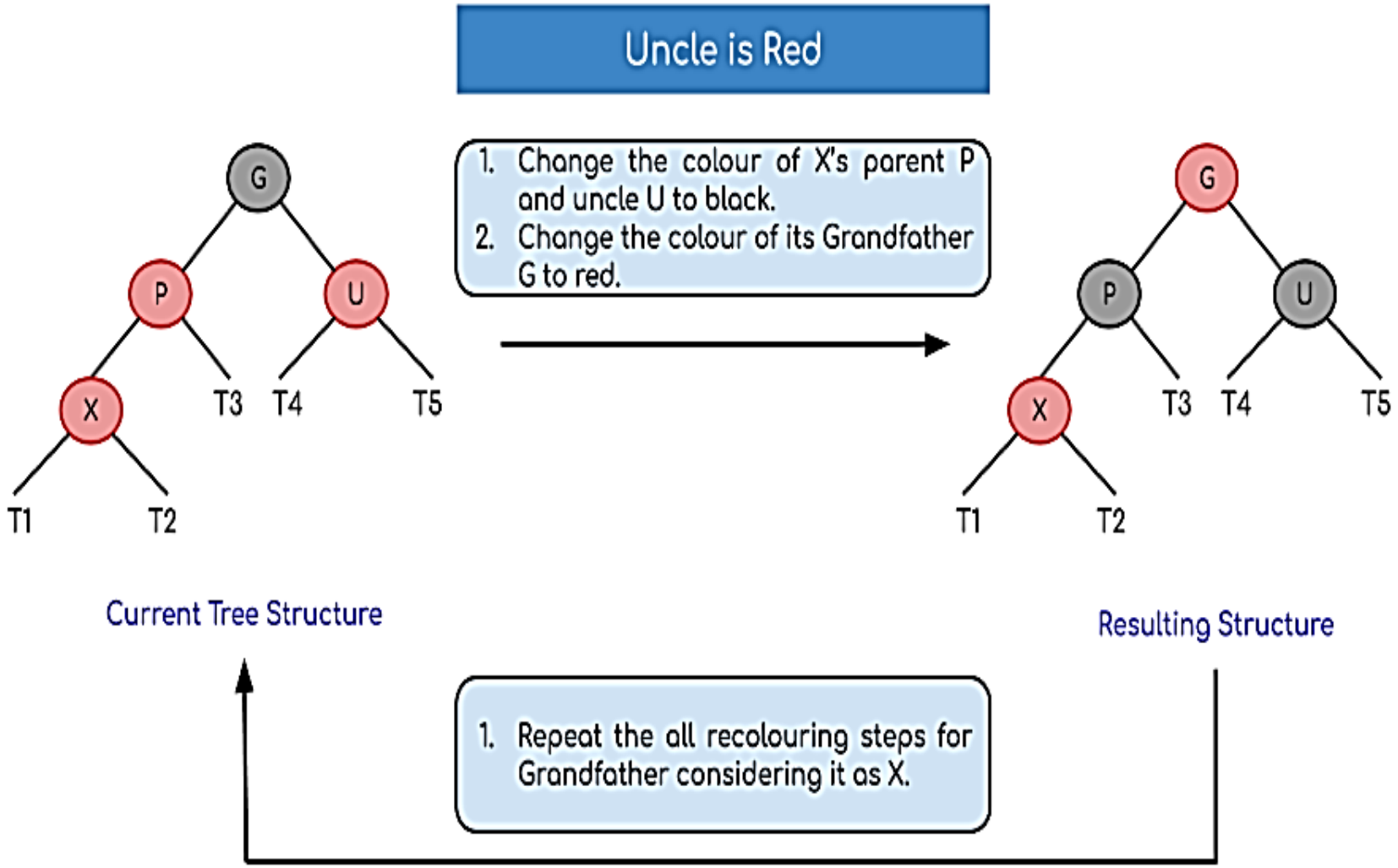
- In the Red-Black tree Insertion, we use two tools to do the balancing.
    - Recoloring
    - Rotation
- Recolouring is the change in colour of the node i.e. if it is red then change it to black and vice versa
- always try recolouring first, if recolouring doesn’t work, then we go for rotation.

Red - Black Tree Insertion

- The algorithms have mainly two cases depending upon the colour of the uncle.
  - If the uncle node is red, we do recolour (Parent, GrandParent, Sibling).
  - If the uncle is black, we do rotations and/or recolouring.

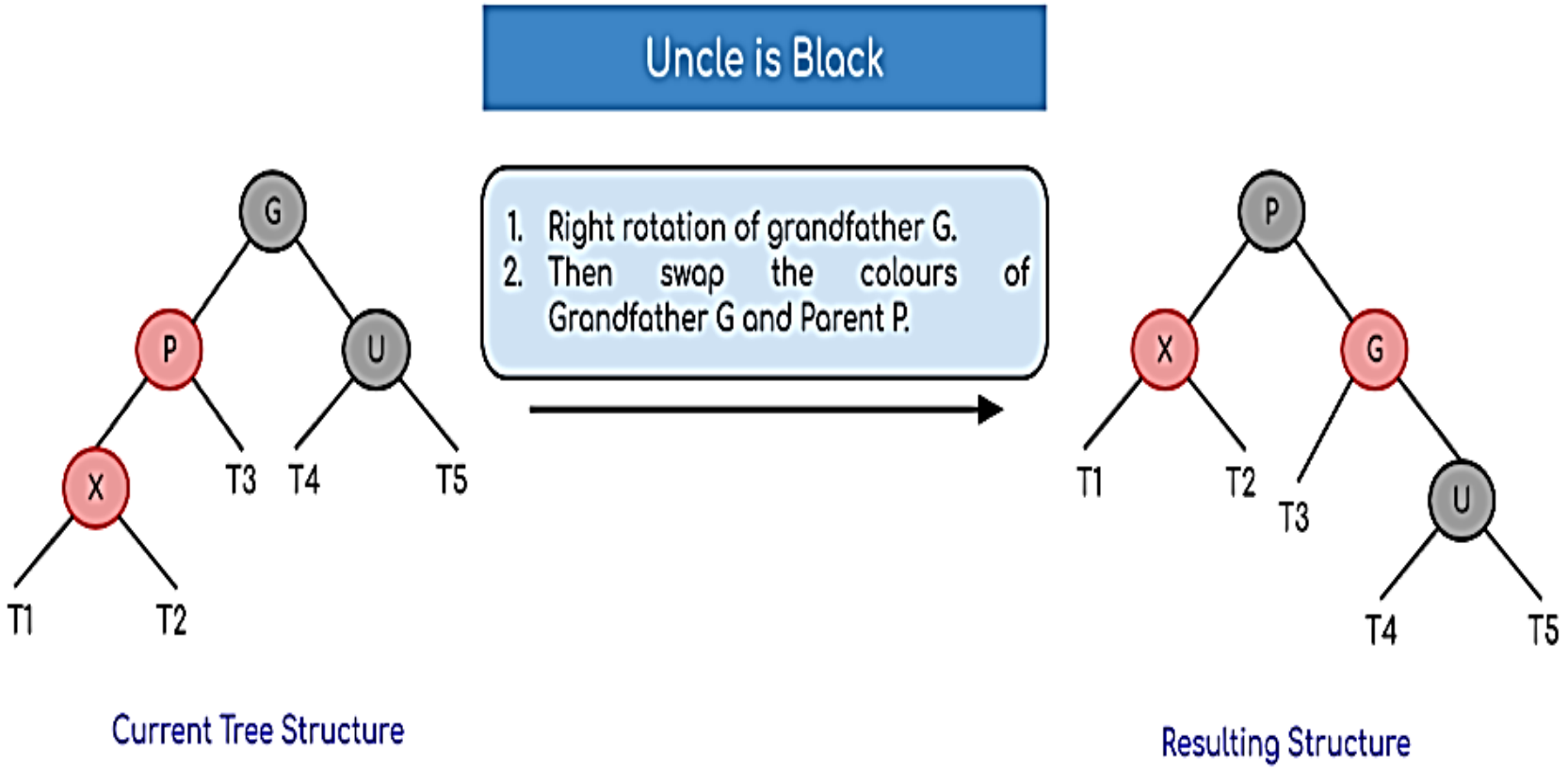


Red - Black Tree Insertion



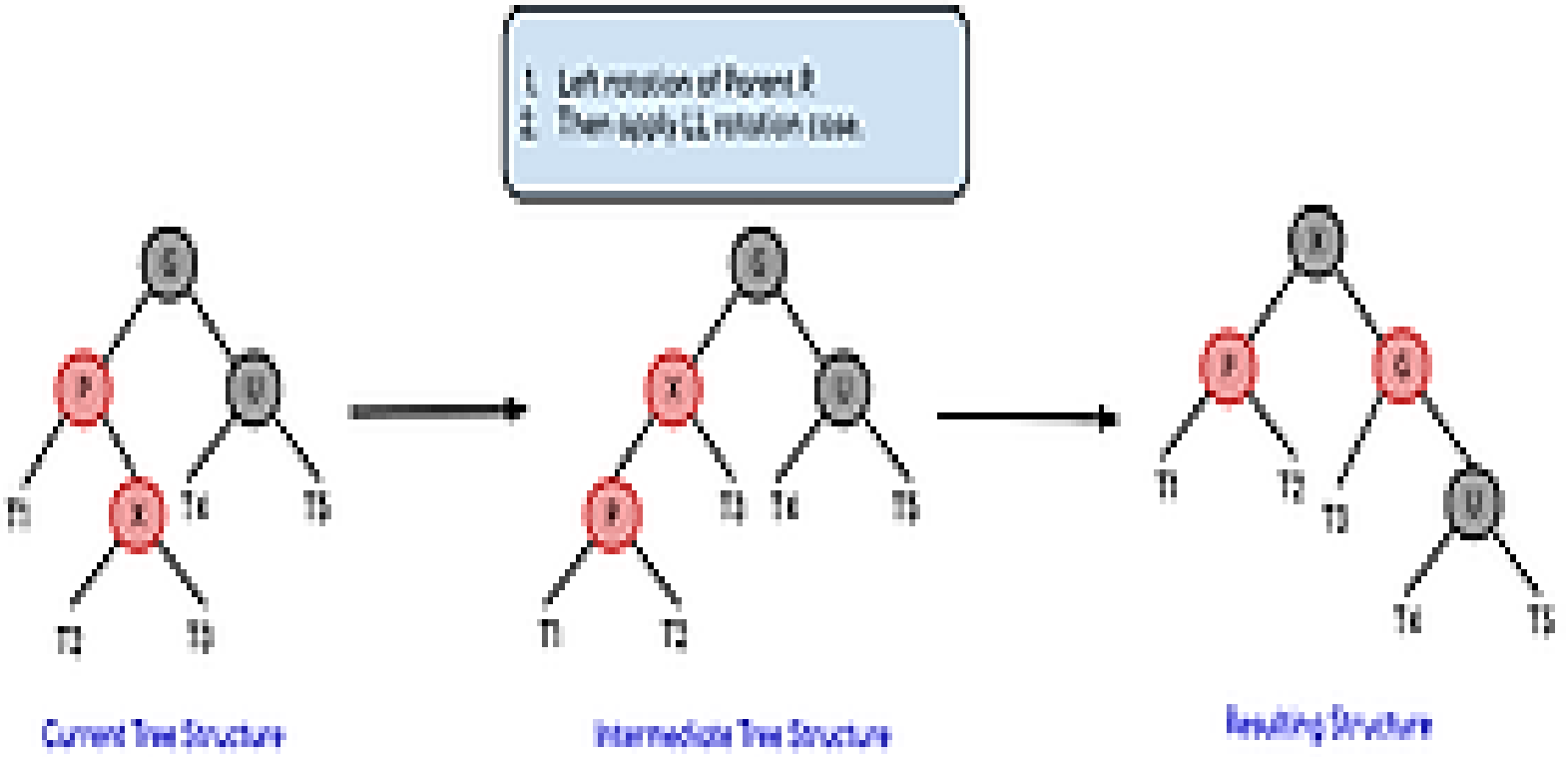
Red - Black Tree Insertion

- If the node's uncle has black colour then there are 4 possible cases:
    - Left Left Case (LL rotation):



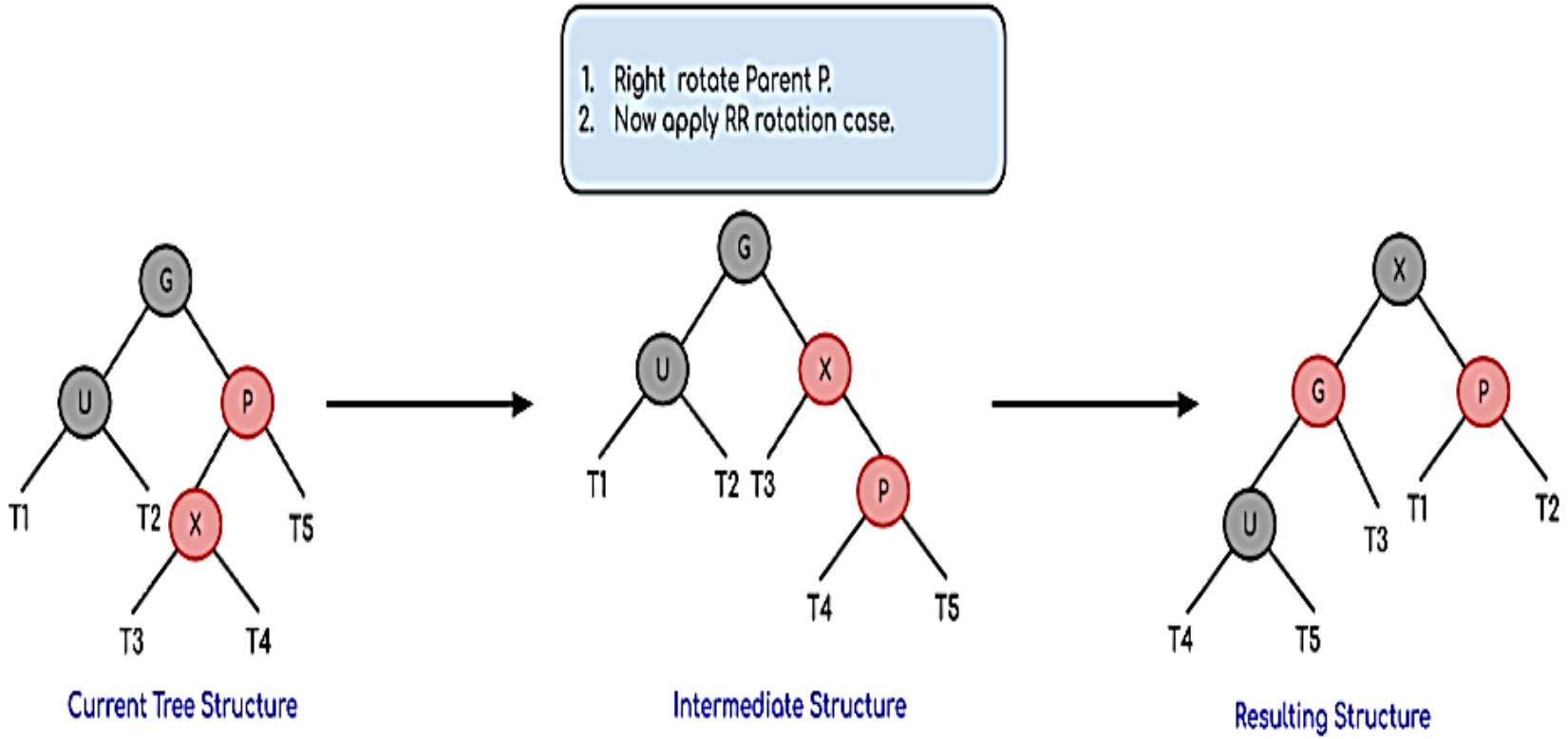
Red - Black Tree Insertion

- If the node's uncle has black colour then there are 4 possible cases:
    - Left Right Case (LR rotation):



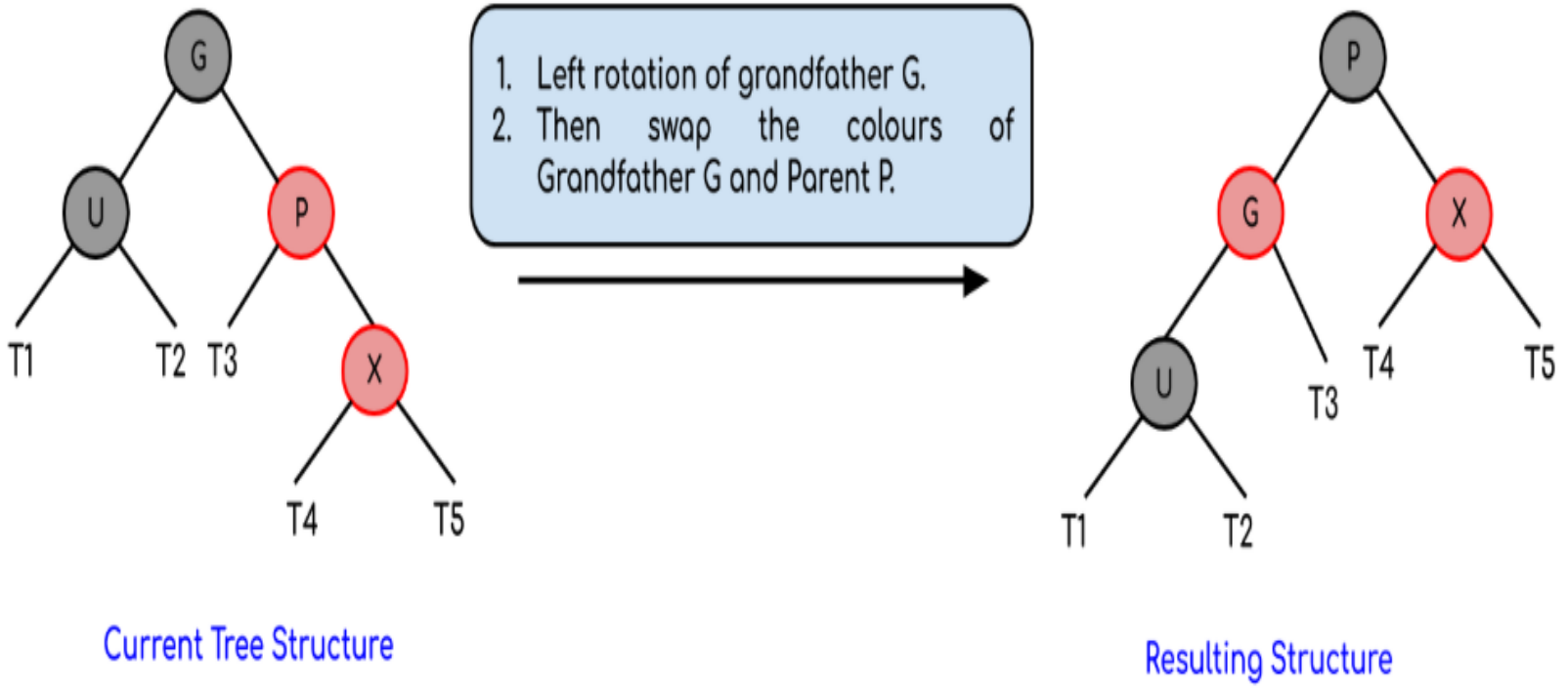
Red - Black Tree Insertion

- If the node's uncle has black colour then there are 4 possible cases:
    - Right Left Case (RL rotation):



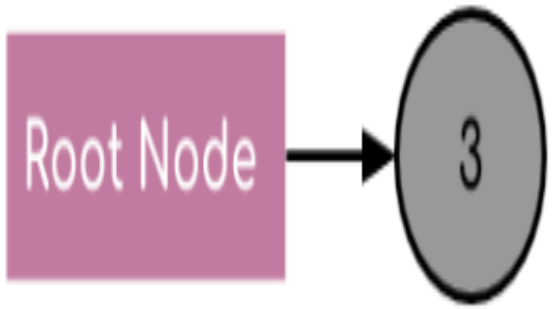
Red - Black Tree Insertion

- If the node's uncle has black colour then there are 4 possible cases:
    - Right Right Case (RR rotation):



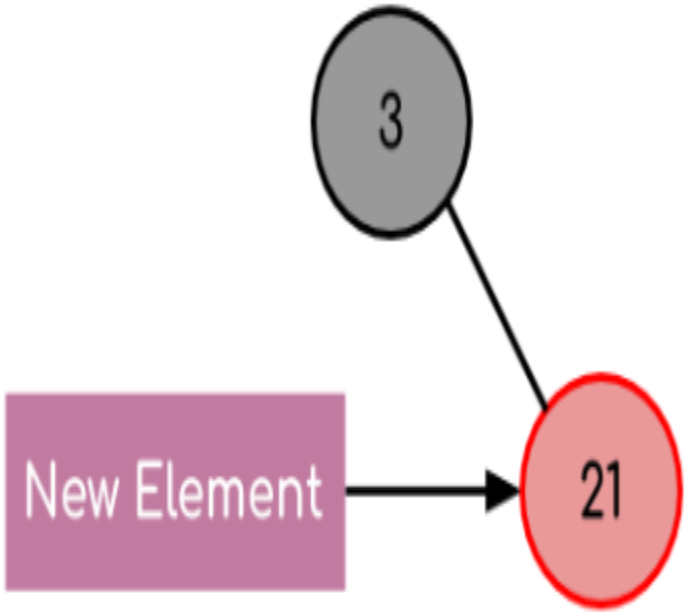
Red - Black Tree Insertion Example

Step 1: Inserting element 3 inside the tree.



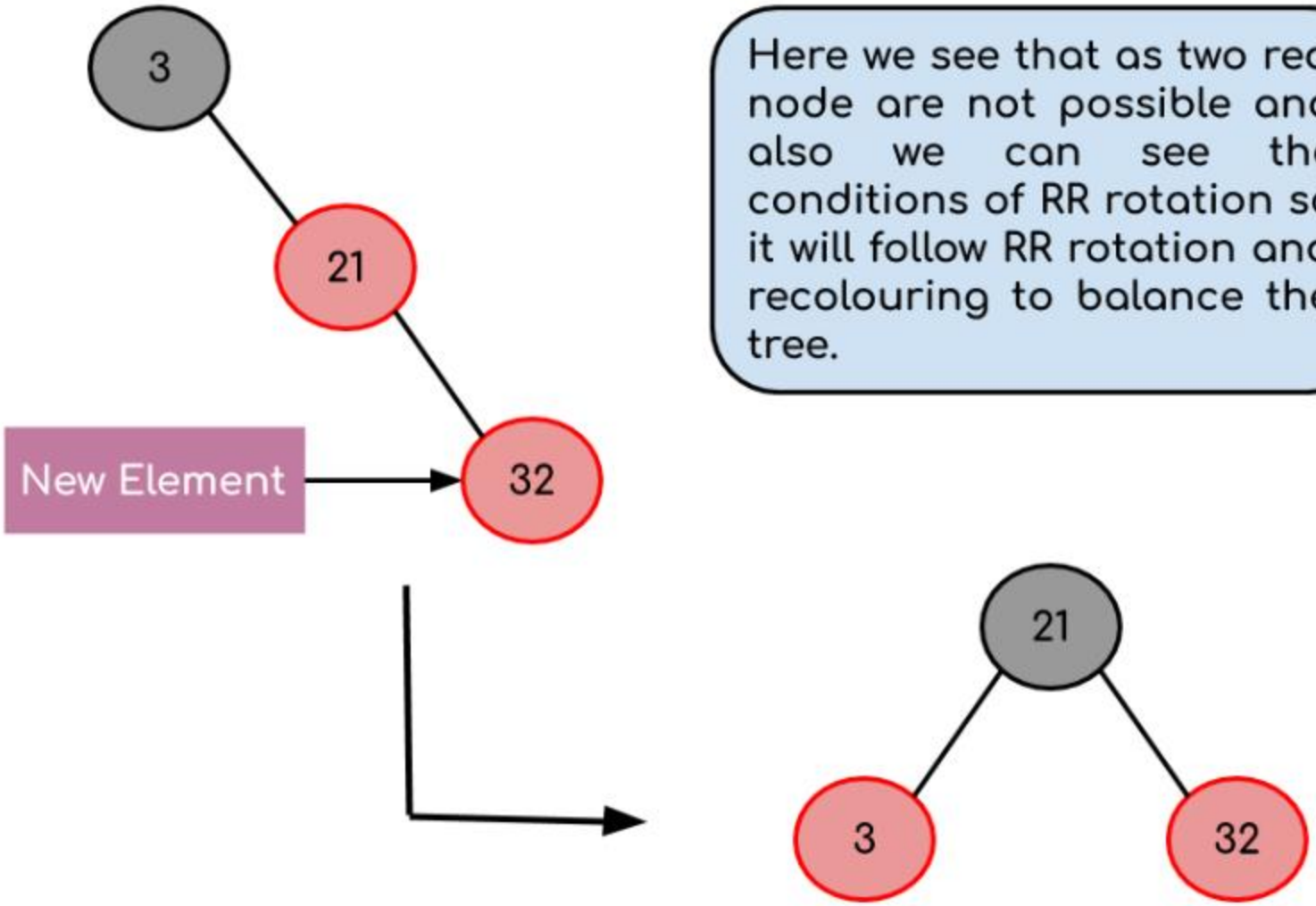
Red - Black Tree Insertion Example

Step 2: Inserting element 21 inside the tree.



Red - Black Tree Insertion Example

Step 3: Inserting element 32 inside the tree.



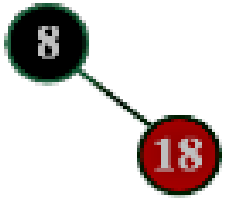
Red - Black Tree Insertion Example

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

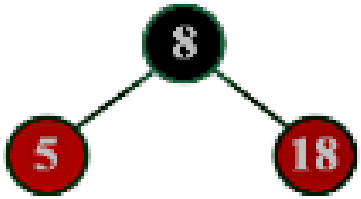
insert (18)

Tree is not Empty. So insert newNode with red color.



insert (5)

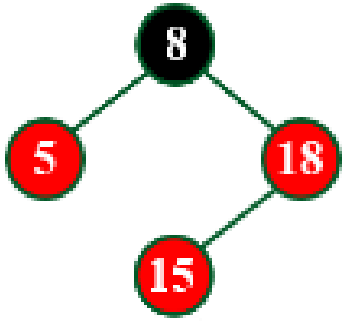
Tree is not Empty. So insert newNode with red color.



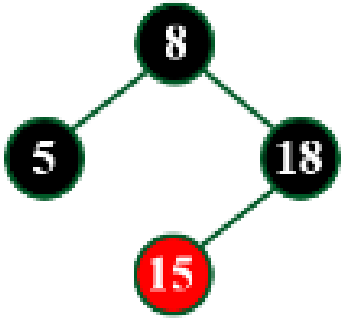
Red - Black Tree Insertion Example

insert (15)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

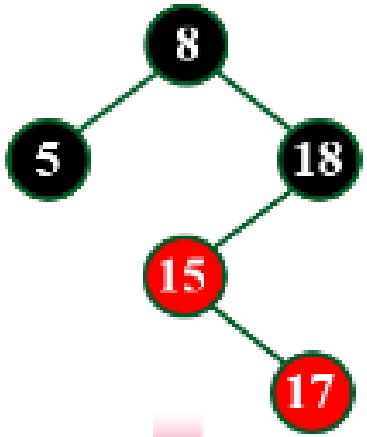


After Recolor operation, the tree is satisfying all Red Black Tree properties.

Red - Black Tree Insertion Example

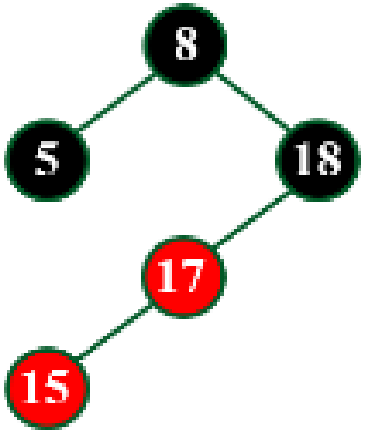
insert (17)

Tree is not Empty. So insert newNode with red color.

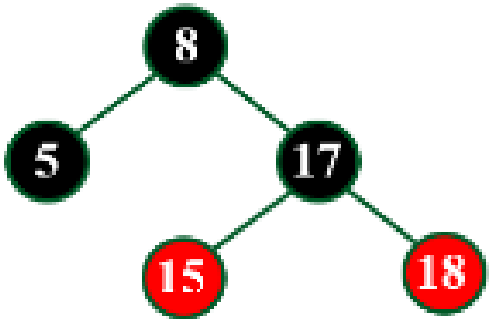


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation



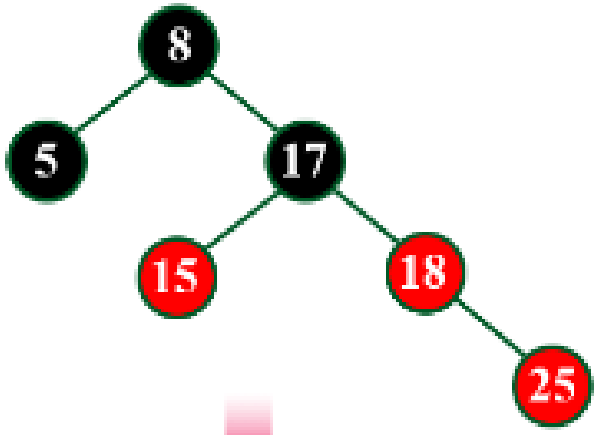
After Right Rotation & Recolor



Red - Black Tree Insertion Example

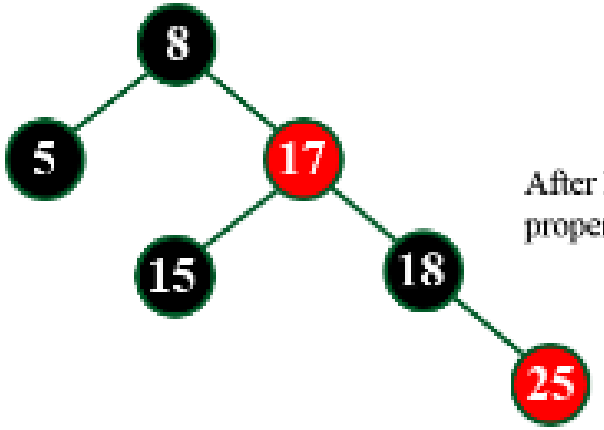
Insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

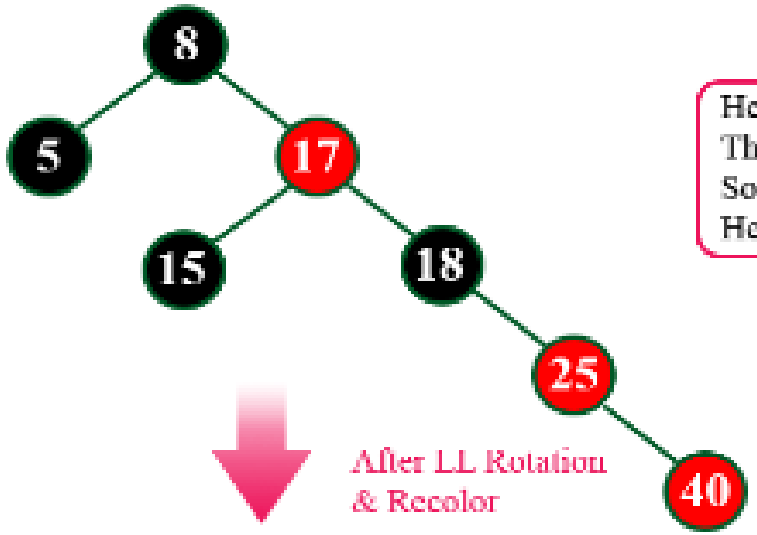


After Recolor operation, the tree is satisfying all Red Black Tree properties.

Red - Black Tree Insertion Example

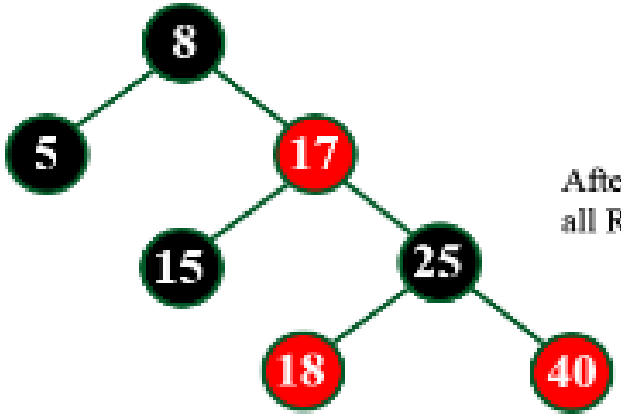
insert (40)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL.
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation & Recolor

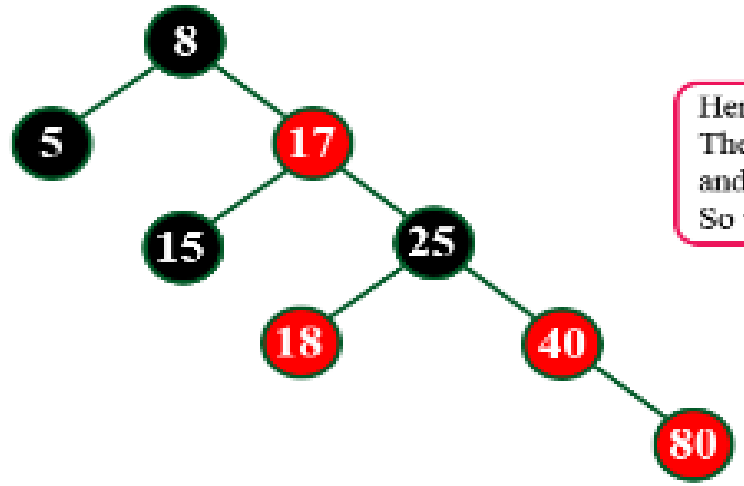


After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

Red - Black Tree Insertion Example

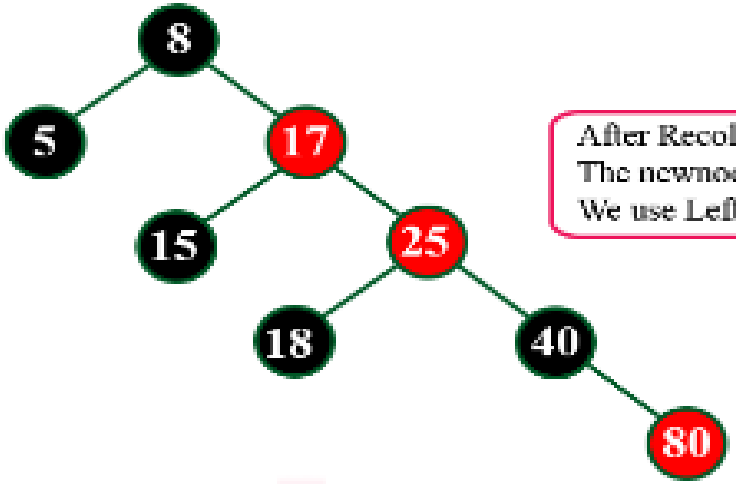
insert (80)

Tree is not Empty. So insert newNode with red color.



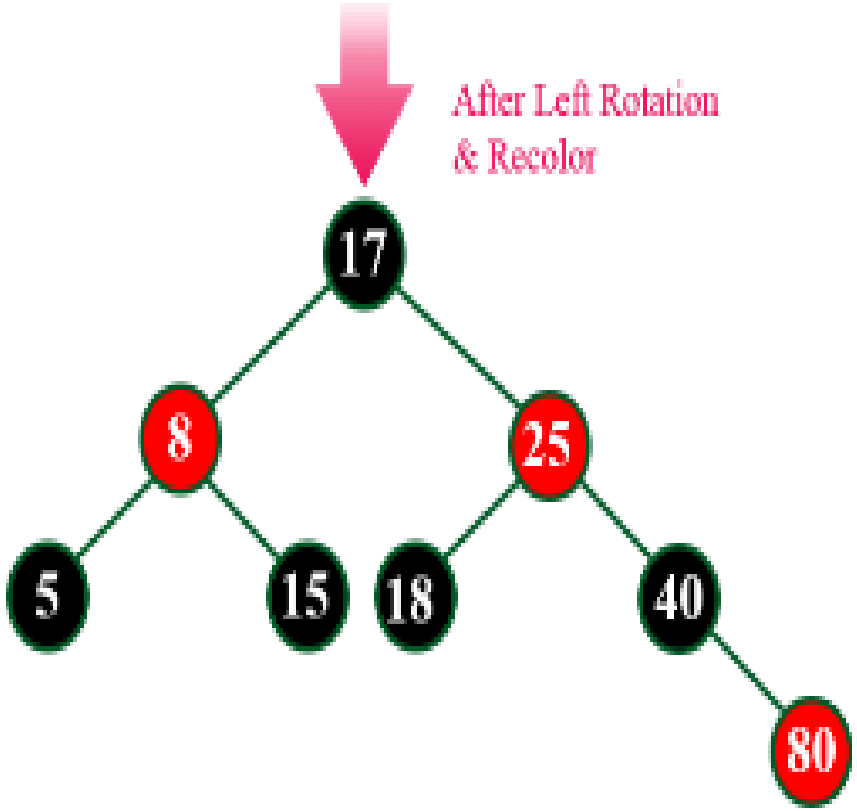
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

Red - Black Tree Insertion Example



Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

Red - Black Tree Deletion

Deletion Steps:

- If the node to be deleted has no children, simply remove it and update the parent node.
- If the node to be deleted has only one child, replace the node with its child.
- If the node to be deleted has two children, then replace the node with its in-order predecessor or successor and delete it.

Red - Black Tree Deletion

Deletion Steps:

- After the node is deleted, the red-black properties might be violated. To restore these properties, some color changes and rotations are performed.
- The deletion operation in a red-black tree takes O(log n) time on average.

Red - Black Tree Deletion

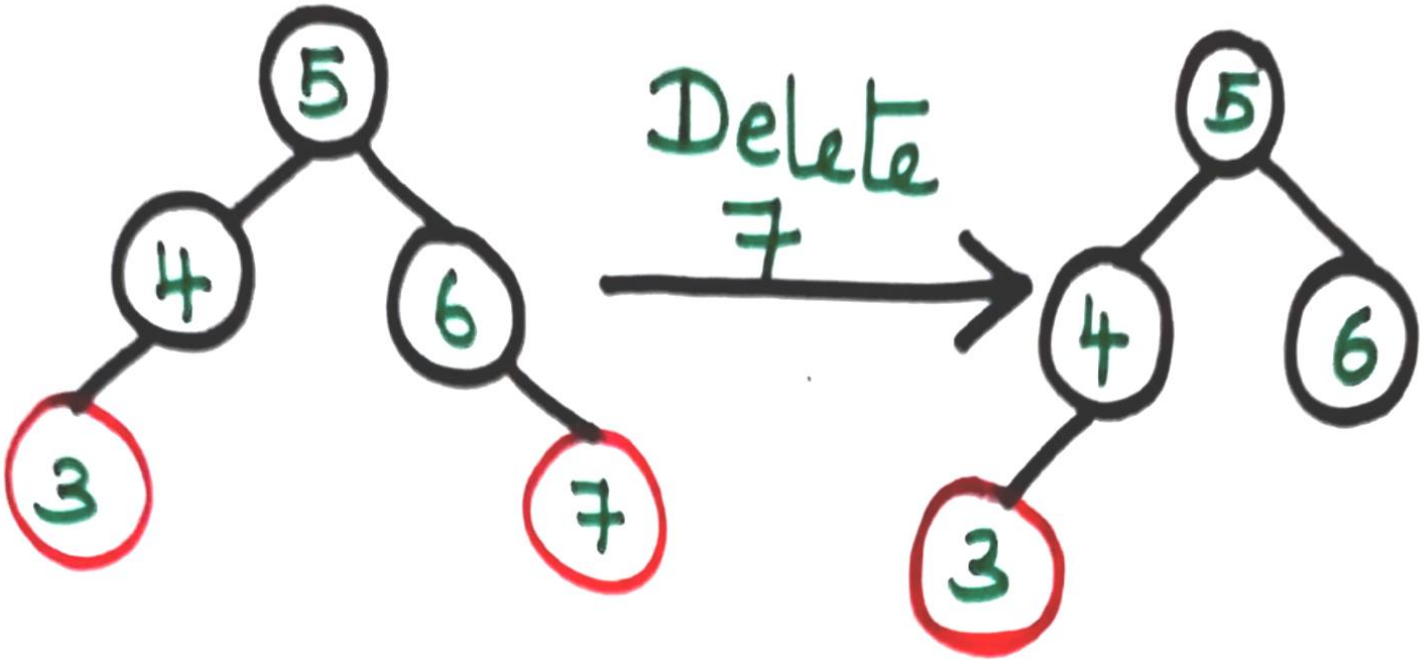
- Deletion is a fairly complex process.
- The notion of double black is used.
- When a black node is deleted and replaced by a black child, the child is marked as **double black**.
- The main task now becomes to convert this double black to single black.

Red - Black Tree Deletion

Case 1: Deleting a red node

Case 1.1: Deleting a leaf node

- directly delete the node.

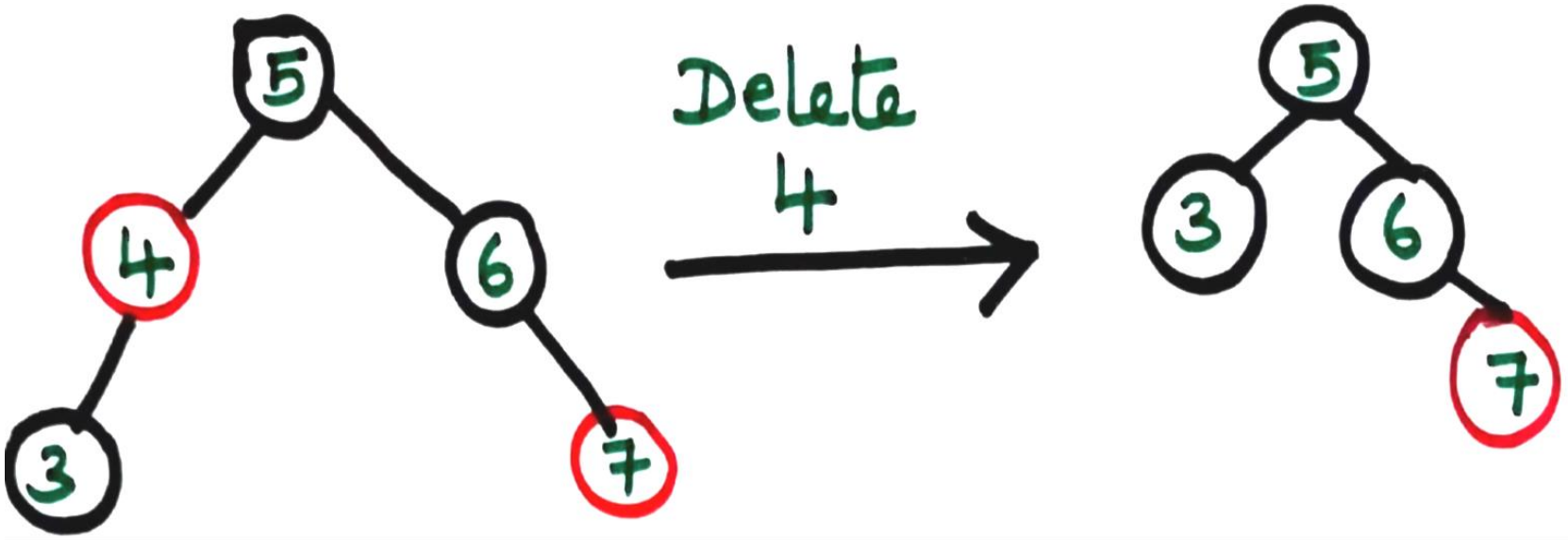


Red - Black Tree Deletion

Case 1: Deleting a red node

Case 1.2: Deleting a Non-leaf / Internal node

- delete the value of a node by replacing it with its child and recolor the node to black.

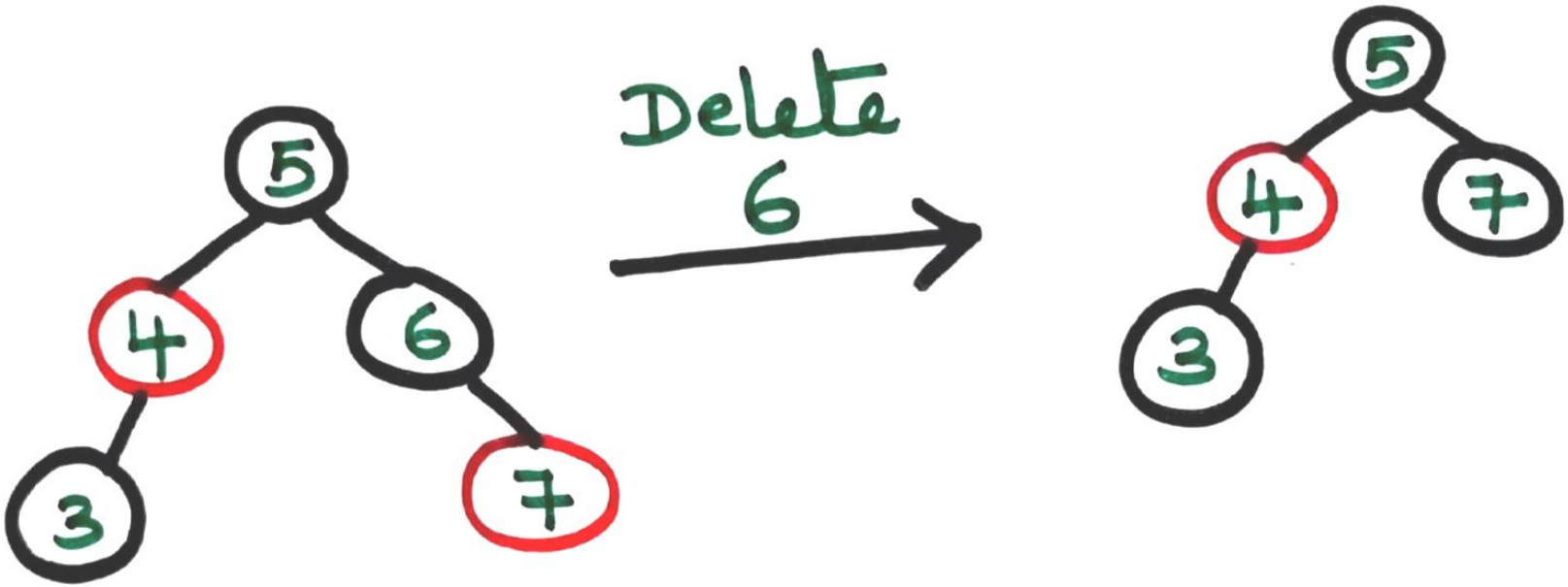


Red - Black Tree Deletion

Case 2: Deleting a black node

Case 2.1: Deleting a Non-leaf / Internal node with red child.

- delete the value of a node by replacing it with its child and recolor the node to black.

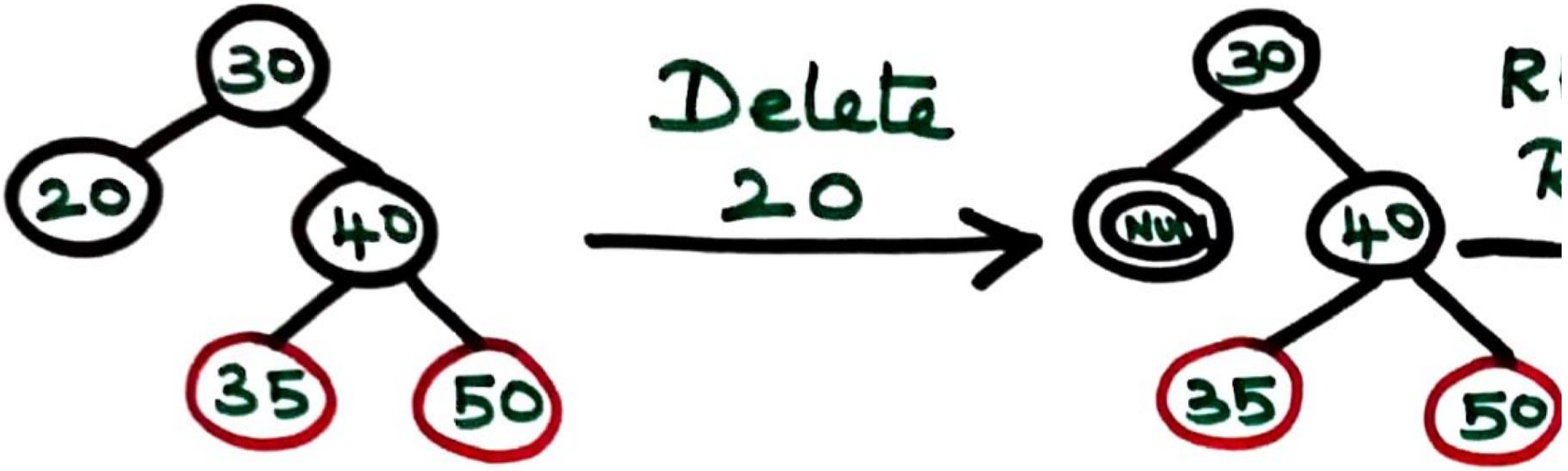


Red - Black Tree Deletion

Case 2: Deleting a black node

Case 2.2: Deleting a Non-leaf / Internal node with black child or no child.

- delete the value of a node by replacing it with its child and mark the node with double black.

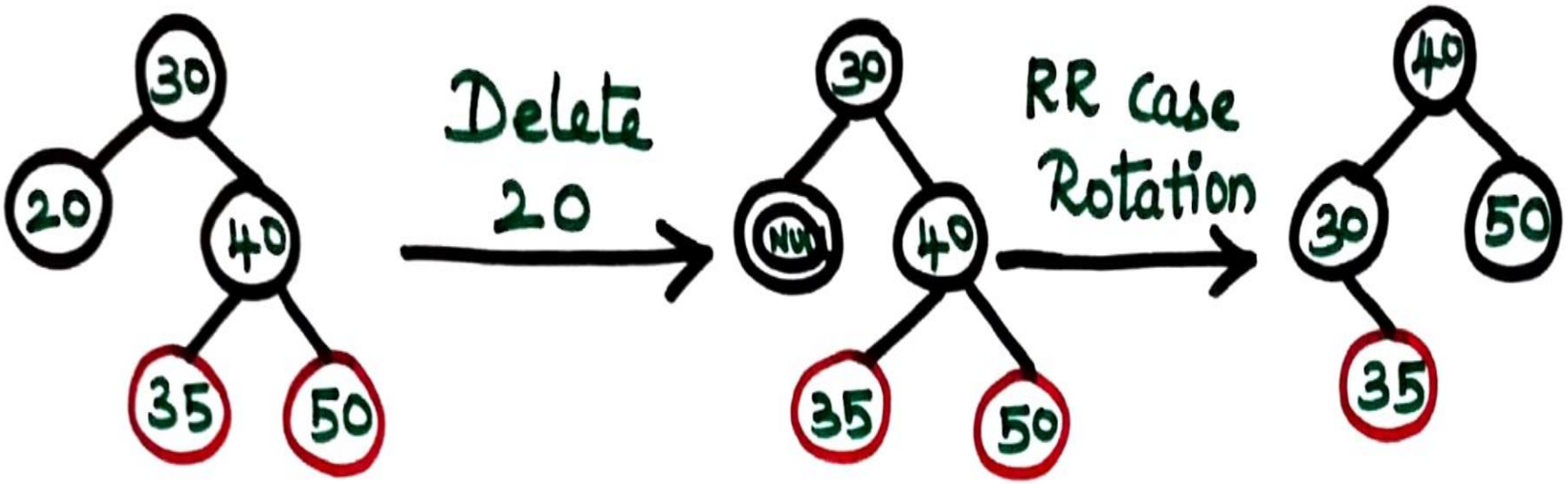


Red - Black Tree Deletion

- If double black exists, then eliminate it with the following cases,

Case 2.2.1: Black Sibling with red(atleast one) child

- Perform rotation (LL case or RR case or LR case or RL case)



Red - Black Tree Deletion

Case 2.2.2: Black Sibling with black / no child

- Perform recoloring on Sibling and Parent.

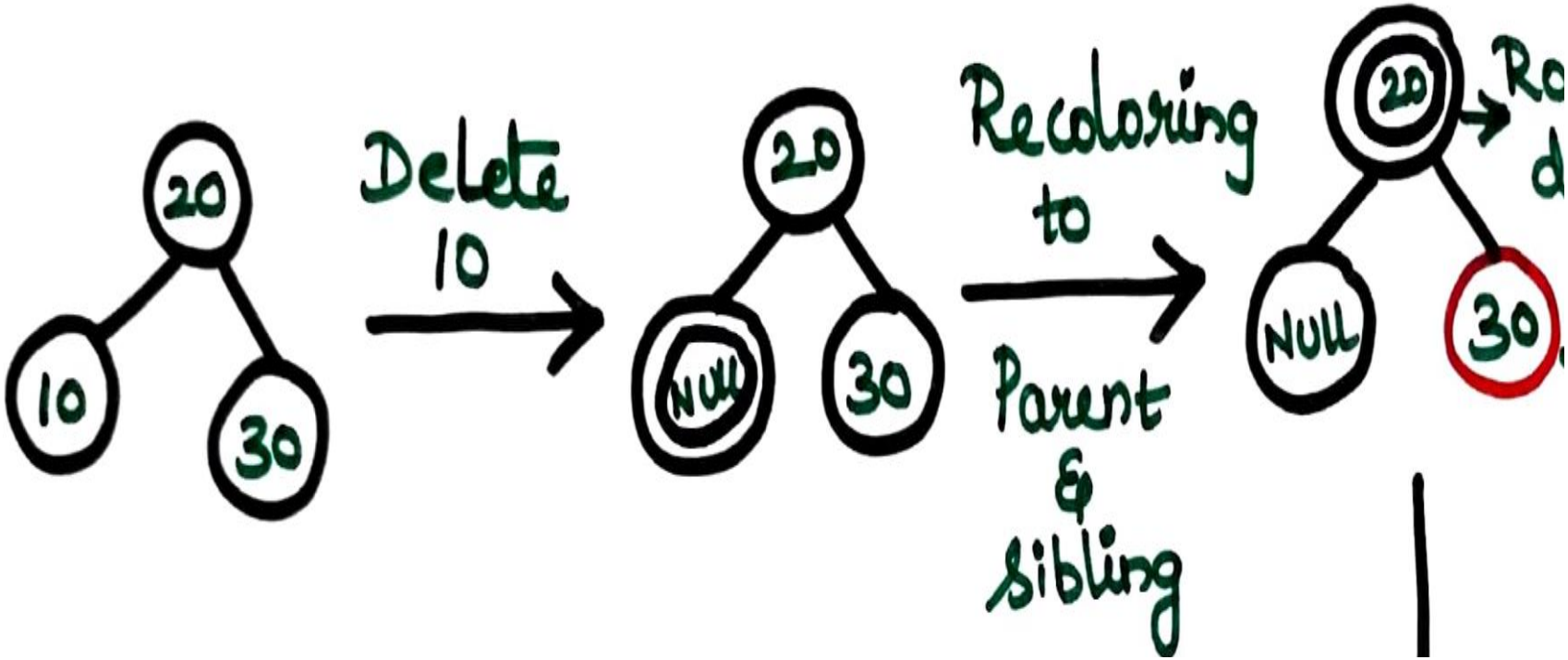
- If parent is already in black then mark it as double black.

- Continue the process upwards until the double black is eliminated.

- If the parent is root node and it is double black then mark the root as Single black directly.

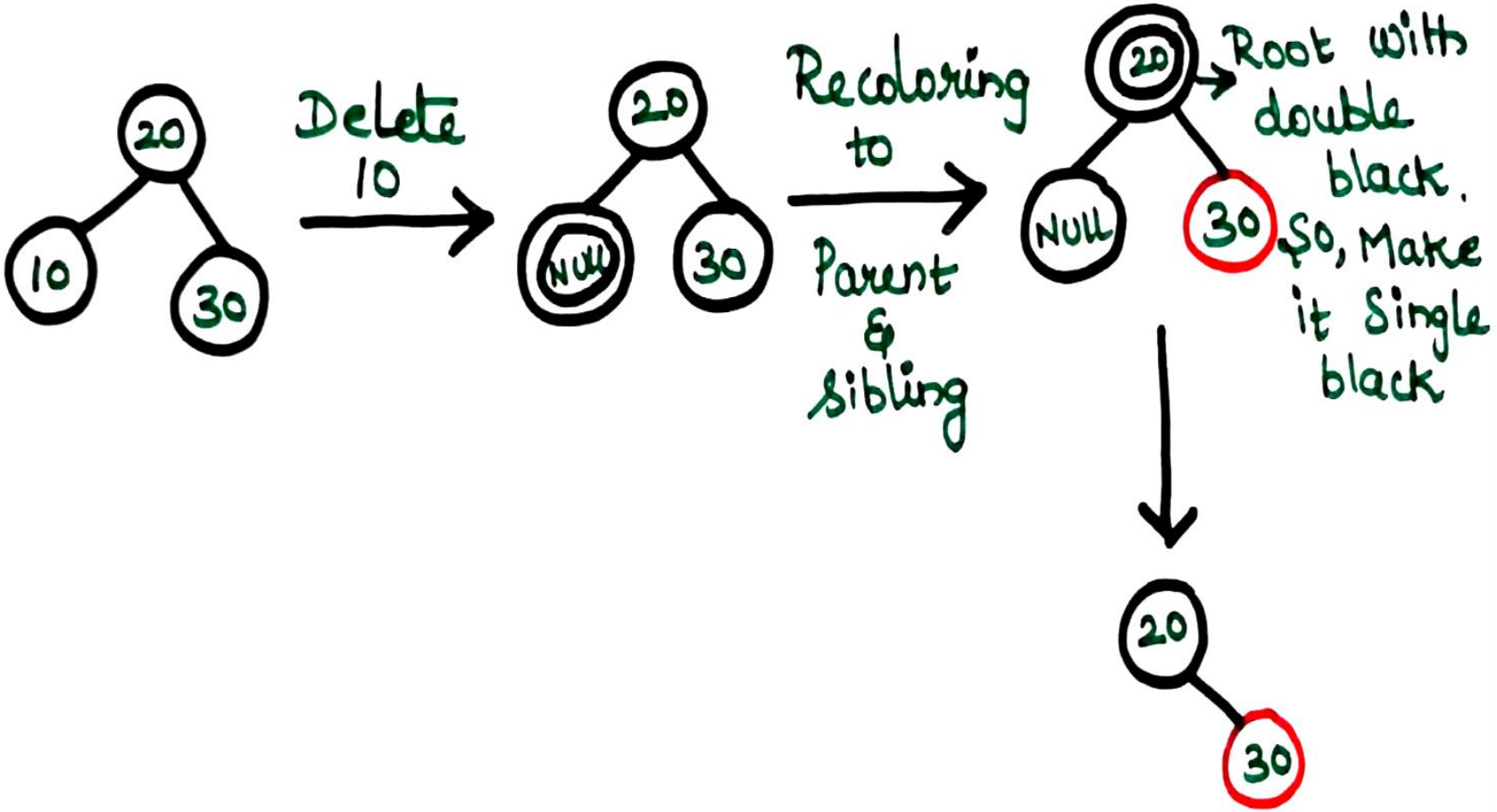
Red - Black Tree Deletion

Case 2.2.2: Black Sibling with black / no child



Red - Black Tree Deletion

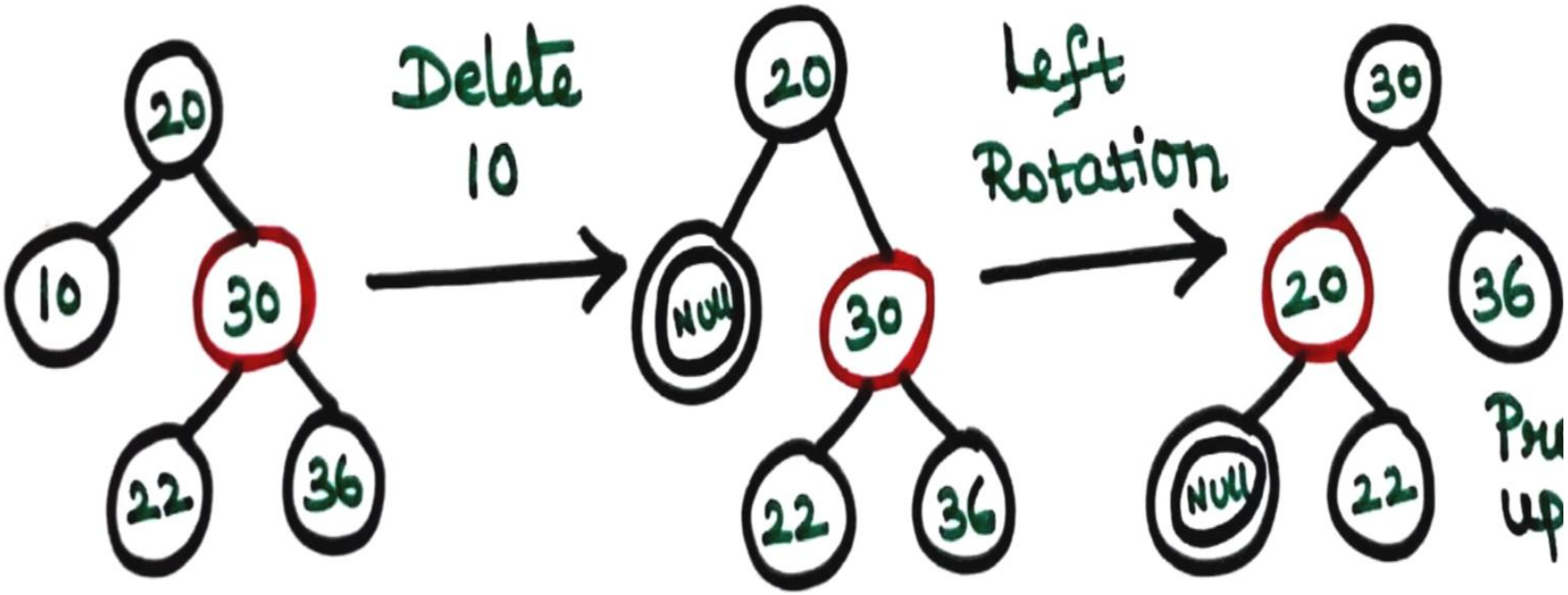
Case 2.2.2: Black Sibling with black / no child



Red - Black Tree Deletion

Case 2.2.3 : Red Sibling

- Perform rotation (LL case or RR case) along with recoloring the parent and Sibling node.

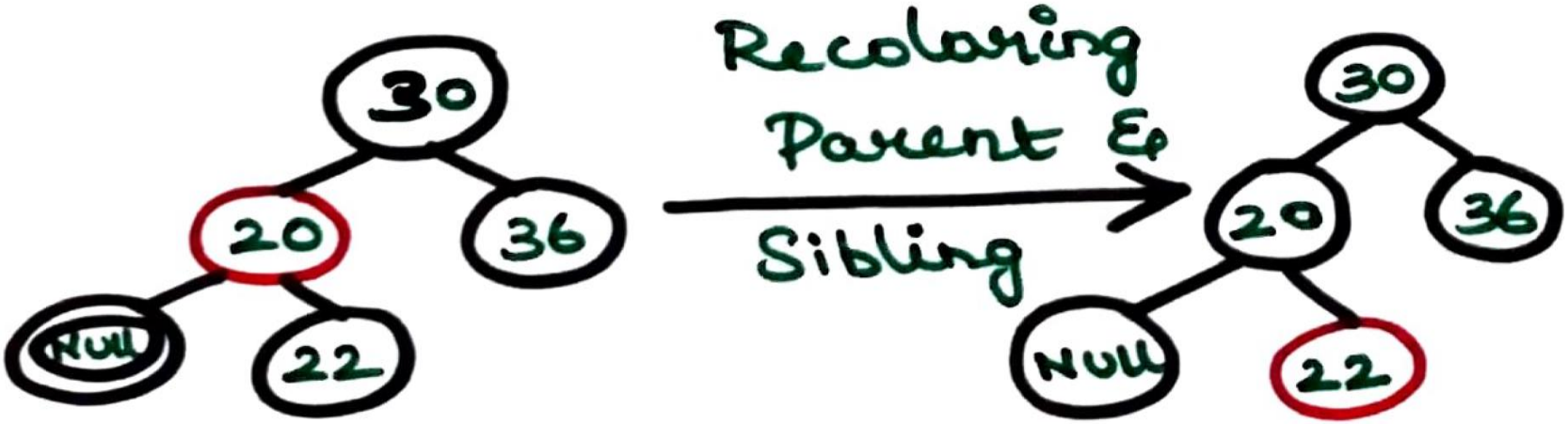


Red - Black Tree Deletion

- Still double black exists in the above resultant tree. So repeat the process until double black is removed.

- Check the sibling again for the resultant tree.

- Black sibling with no child **(Case 2.2.2)**
- Recolor the sibling and parent.

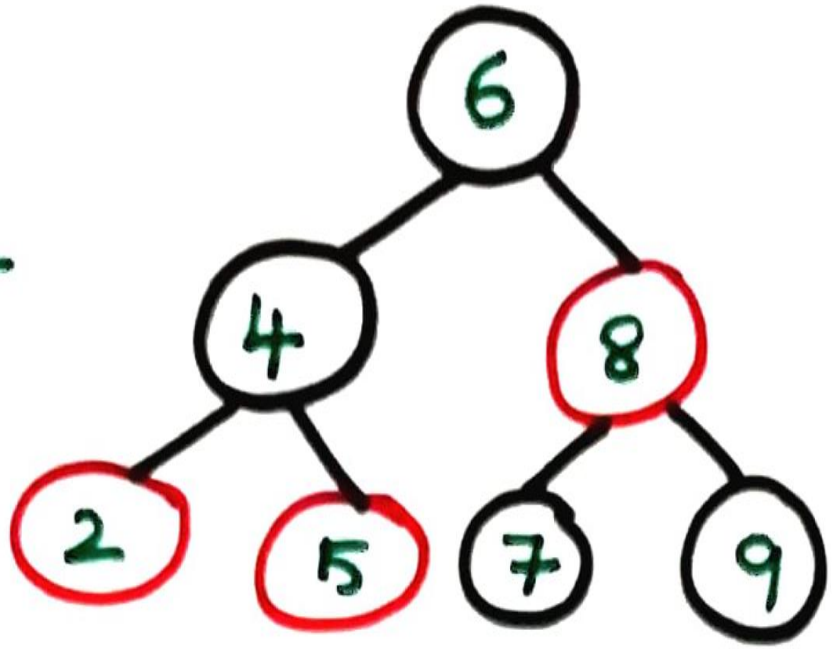


Red - Black Tree Deletion Example

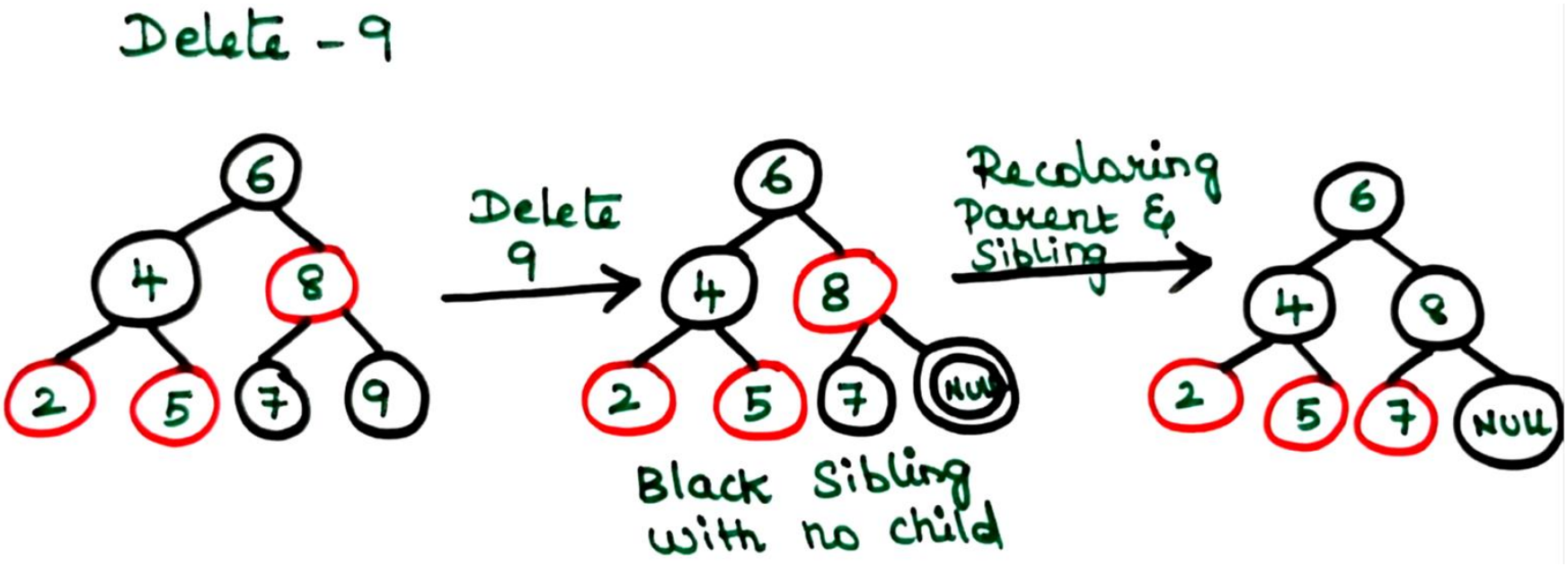
Example :

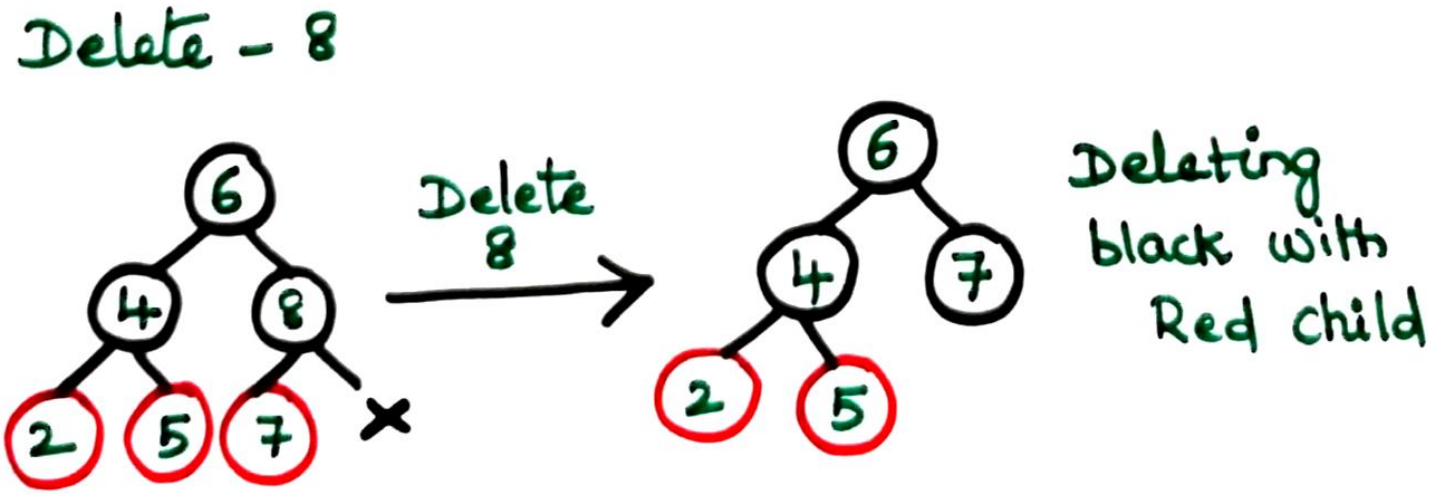
Delete

9, 8, 7

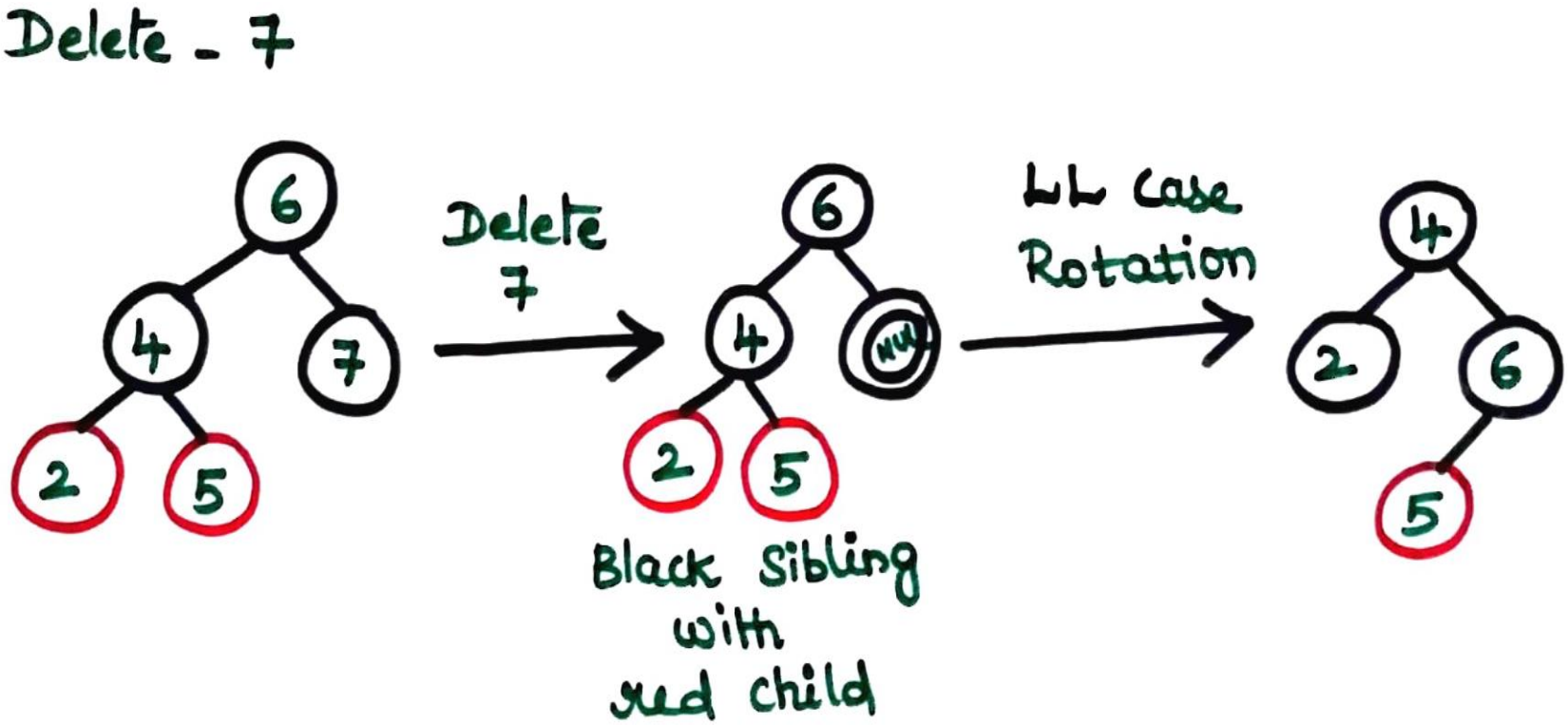


Red - Black Tree Deletion Example





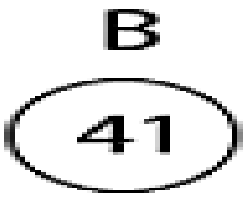
Red - Black Tree Deletion Example



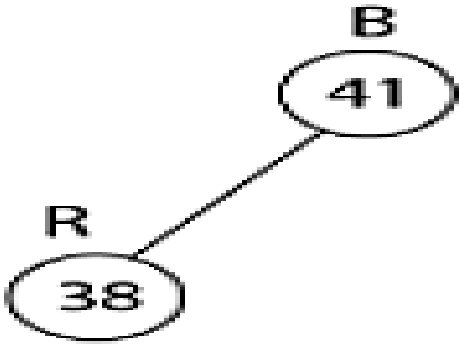
Red - Black Tree Example

Insert the keys 41,38,31,12,19,8 into an initially empty red-black tree.

Insert 41



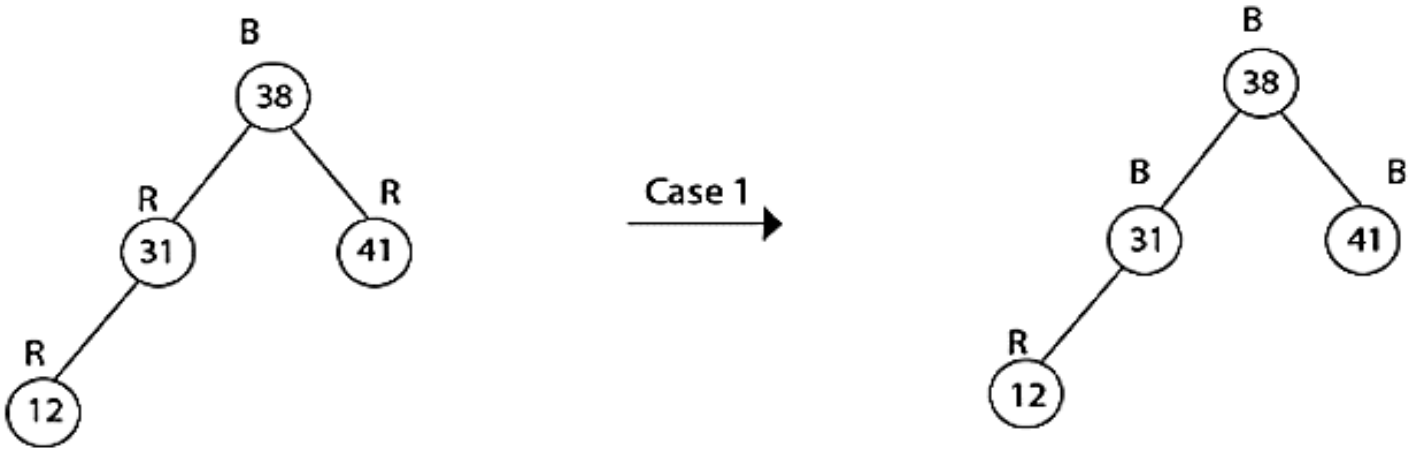
Insert 38



Red - Black Tree Example

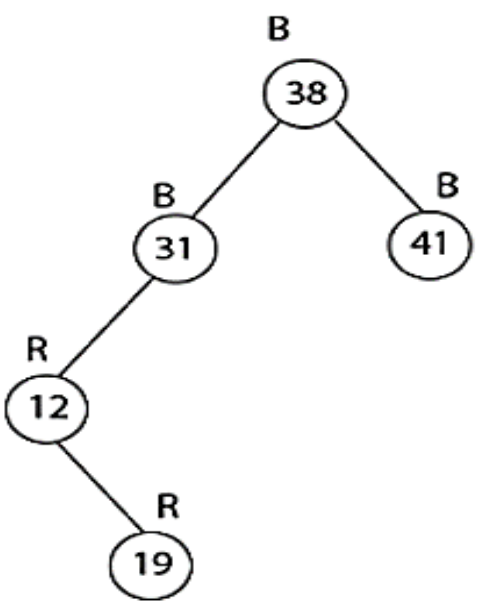
Insert 31

Insert 12

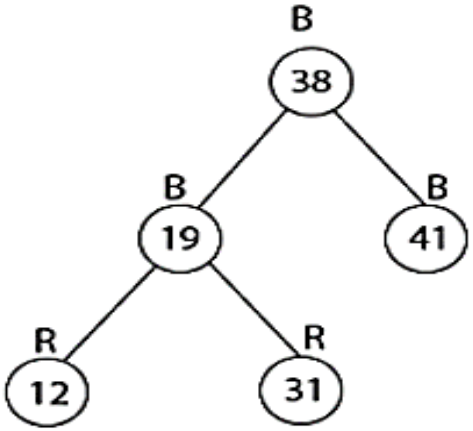


Red - Black Tree Example

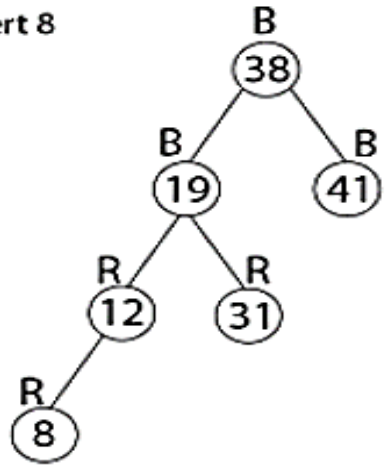
Insert 19



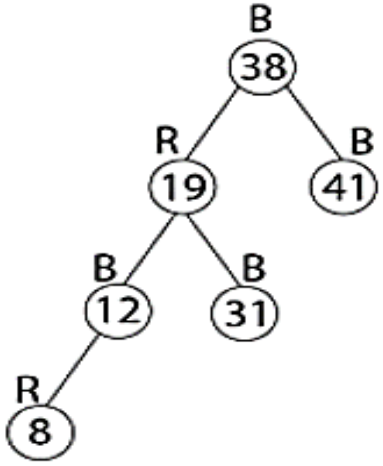
Case 2,3



Insert 8

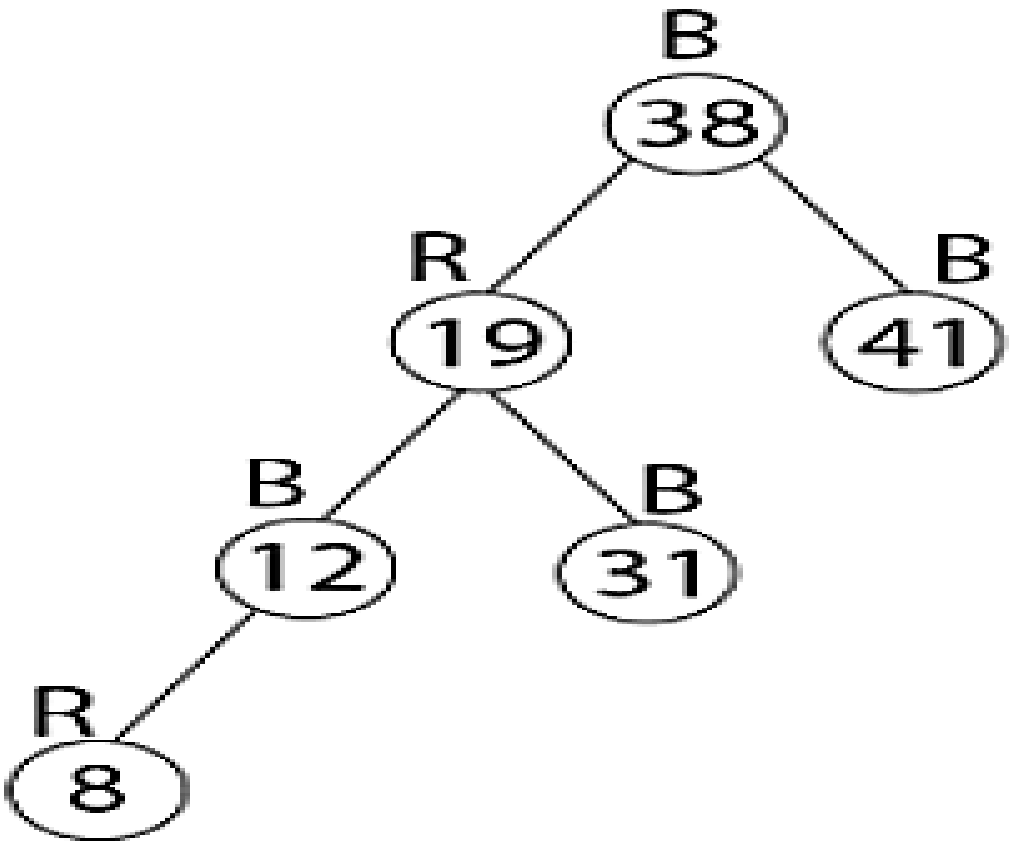


Case-1



Red - Black Tree Example

Thus the final tree is

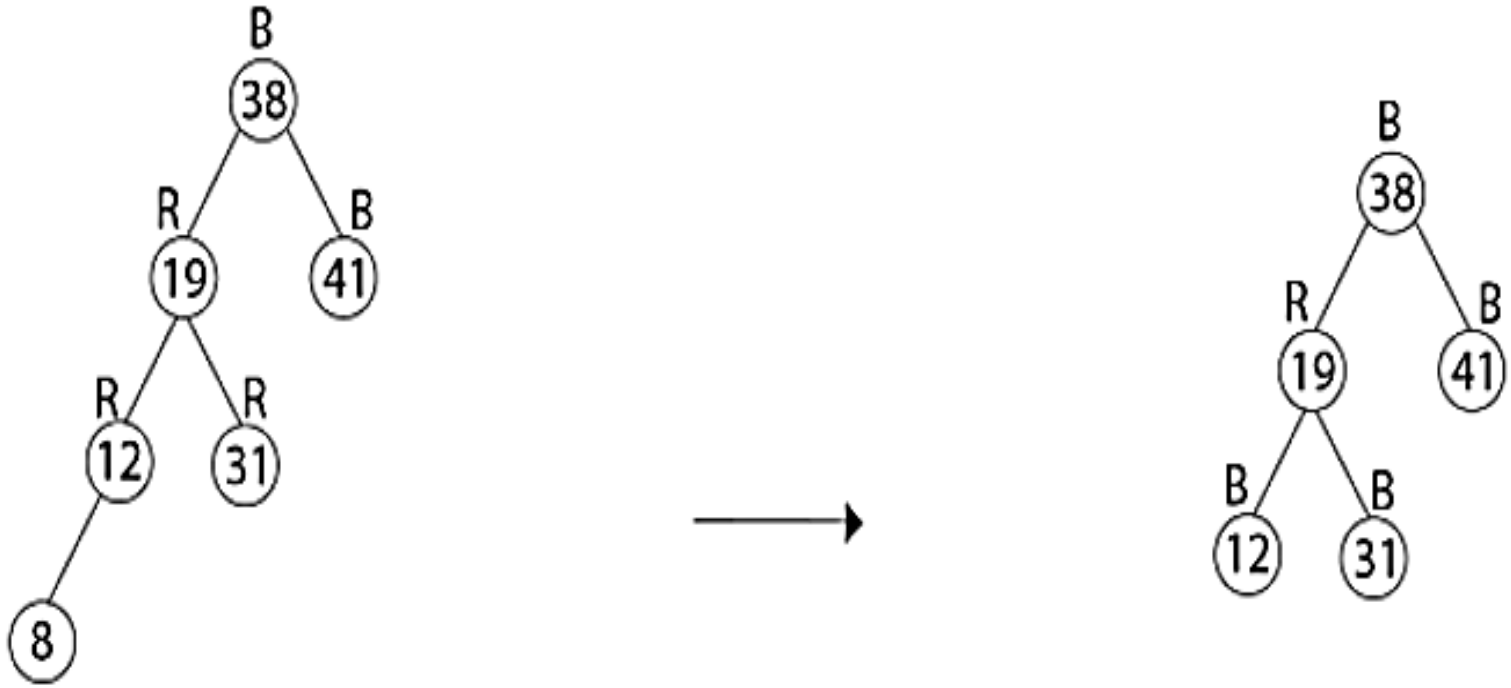


Red - Black Tree Example

Delete the keys in the order 8, 12, 19,31,38,41.

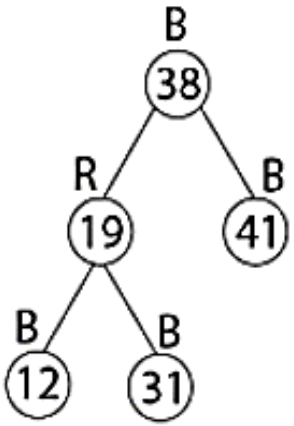
Solution:

Delete 8

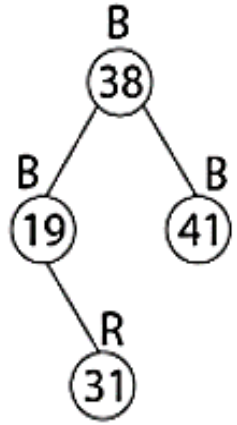


Red - Black Tree Example

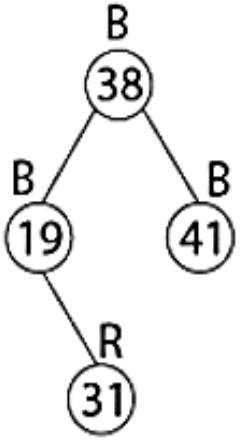
Delete 12

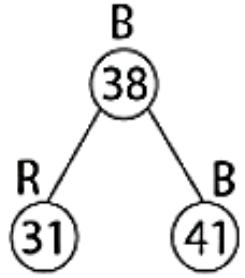


Case-2



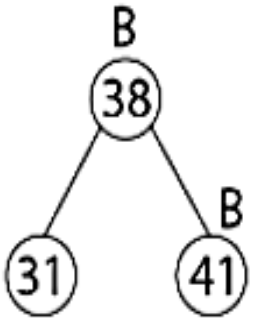
Delete 19



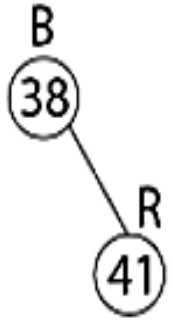


Red - Black Tree Example

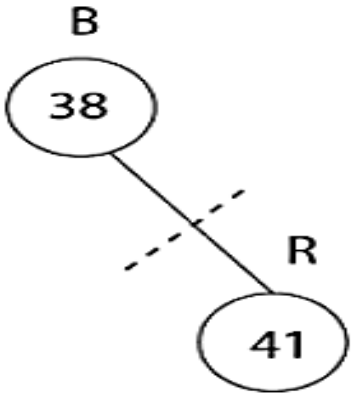
Delete 31

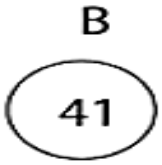


Case-2



Delete 38





Delete 41

No Tree.

B - Tree

- B-Trees, also known as B-Tree or Balanced Tree.
- It is a type of self-balancing tree.
- Each node in a B-Tree can contain multiple keys.
- The tree has a larger branching factor and a shallower height.
- The shallow height leads to less disk I/O, which results in faster search and insertion operations.
- B-Trees maintains balance by ensuring that each node has a minimum number of keys.

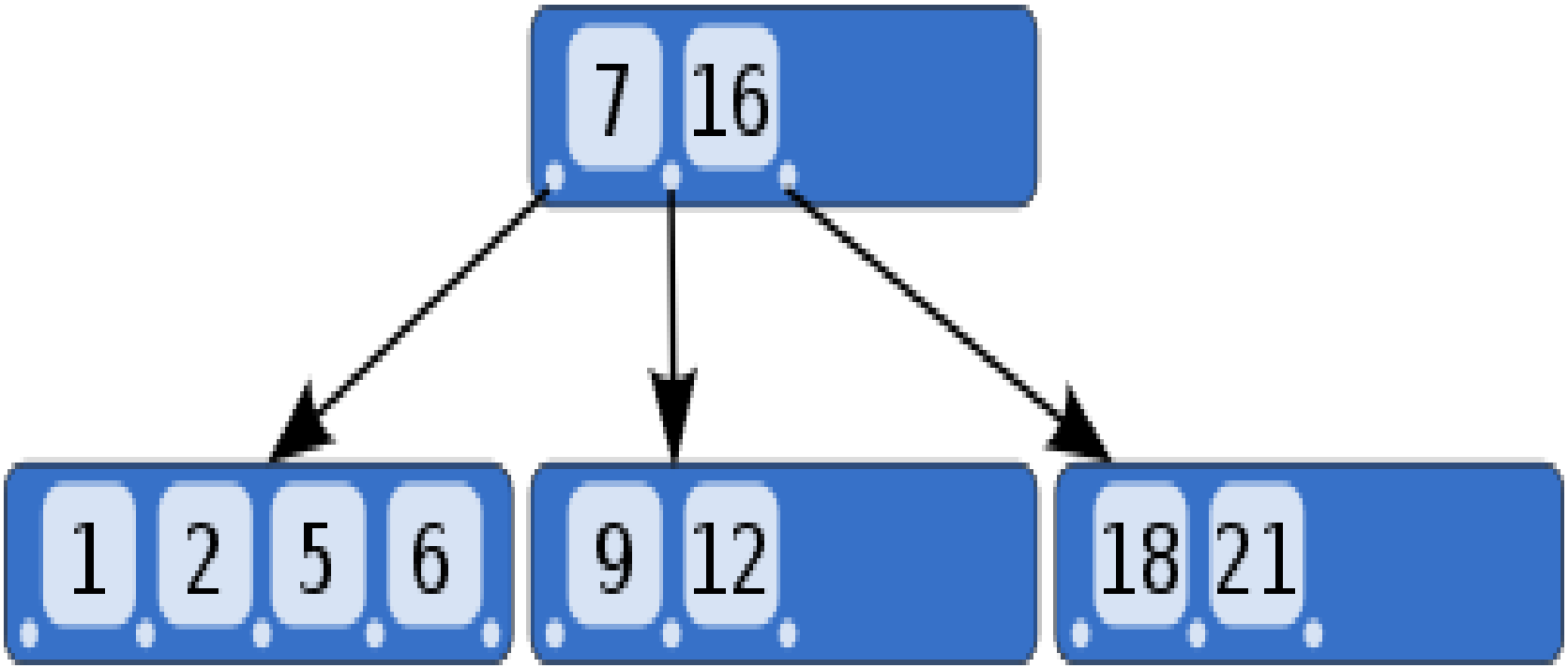
B – Tree properties

- A B-tree of order m satisfies the following properties:
  - Every node has at most m children.
  - The root node has at least two children unless it is a leaf.
  - Every node, except for the root and the leaves, has at least ⌈m/2⌉ children.
  - All leaves appear on the same level.
  - An internal node with m order should contains atmost $m-1$ keys and atleast $(m/2-1)$ keys.

B - Tree

S. No.	Operations	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

B – Tree Example



B – Tree Operations

Traversal:

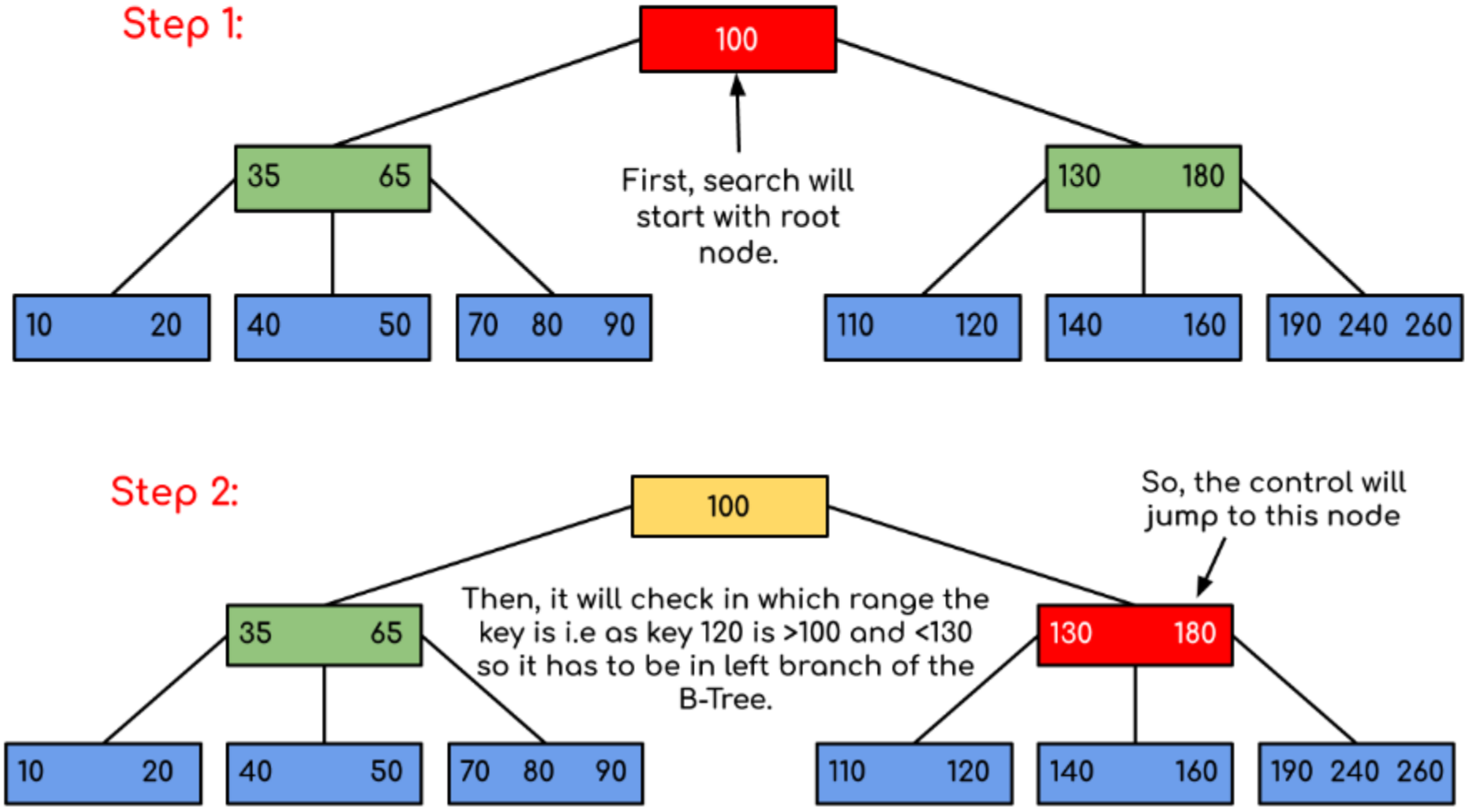
- Inorder traversal (LeftSubTree - Root - RightSubTree)

Search Operation:

- Similar to the search in Binary Search Tree.
- Start from the root and recursively traverse down.
- If the node has the key, we simply return the node.
- Otherwise, we recur down to the appropriate child.
- If we reach a leaf node and don’t find k in the leaf node, then return NULL.

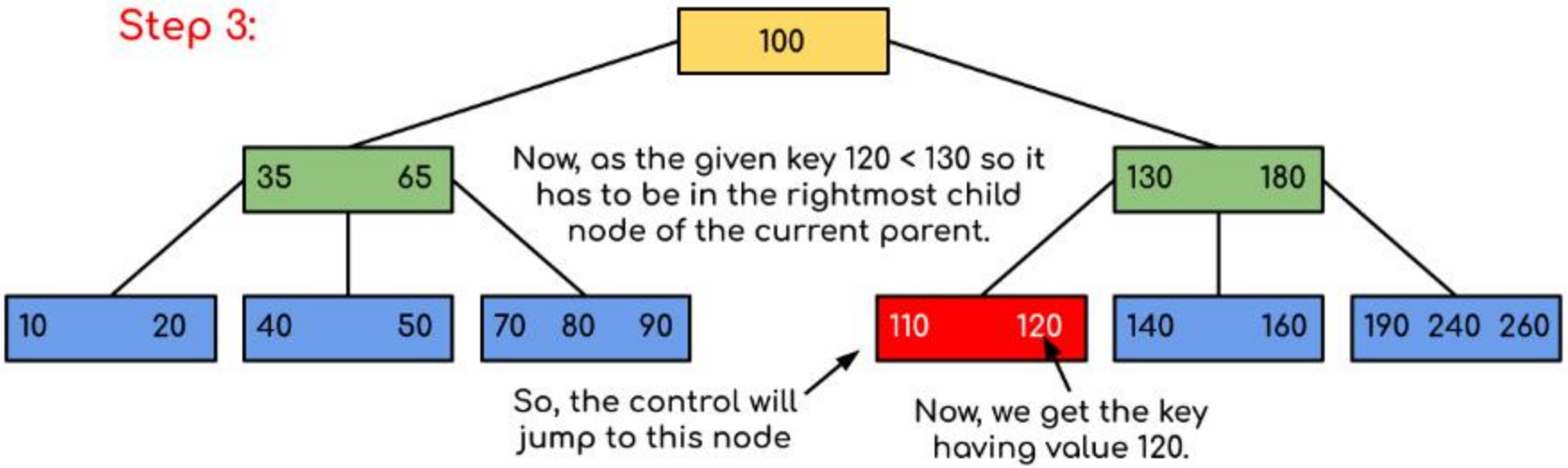
B – Tree Search Operation Example

Input: Search 120 in the given B-Tree.



B – Tree Search Operation Example

Input: Search 120 in the given B-Tree.



B – Tree Operations

Insert Operation:

- A new key is always inserted at the leaf node

- Start from the root and traverse down till we reach a leaf node.

- There is a predefined range on the number of keys that a node can contain.

  - Before inserting a key to the node, make sure that the node has extra space.
  - If a node is full, then use an operation called splitChild() to split a child of a node and create space.

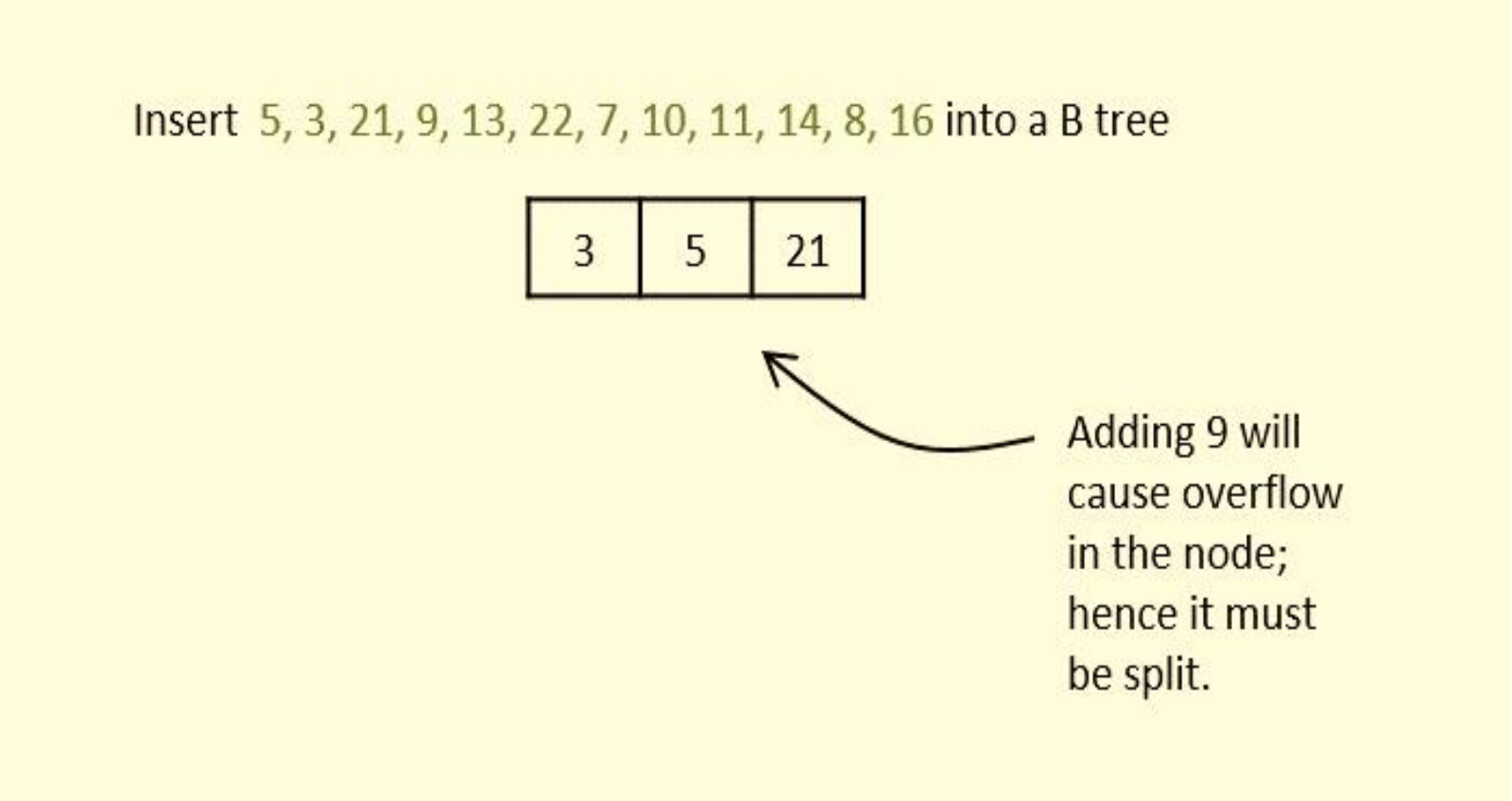
B – Tree Insert Operation Example

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys (m – 1) = 3
- Minimum Keys ($\left\lceil \frac{m}{2} \right\rceil$) – 1 = 1
- Maximum Children = 4
- Minimum Children ($\left\lceil \frac{m}{2} \right\rceil$) = 2

B – Tree Insert Operation Example

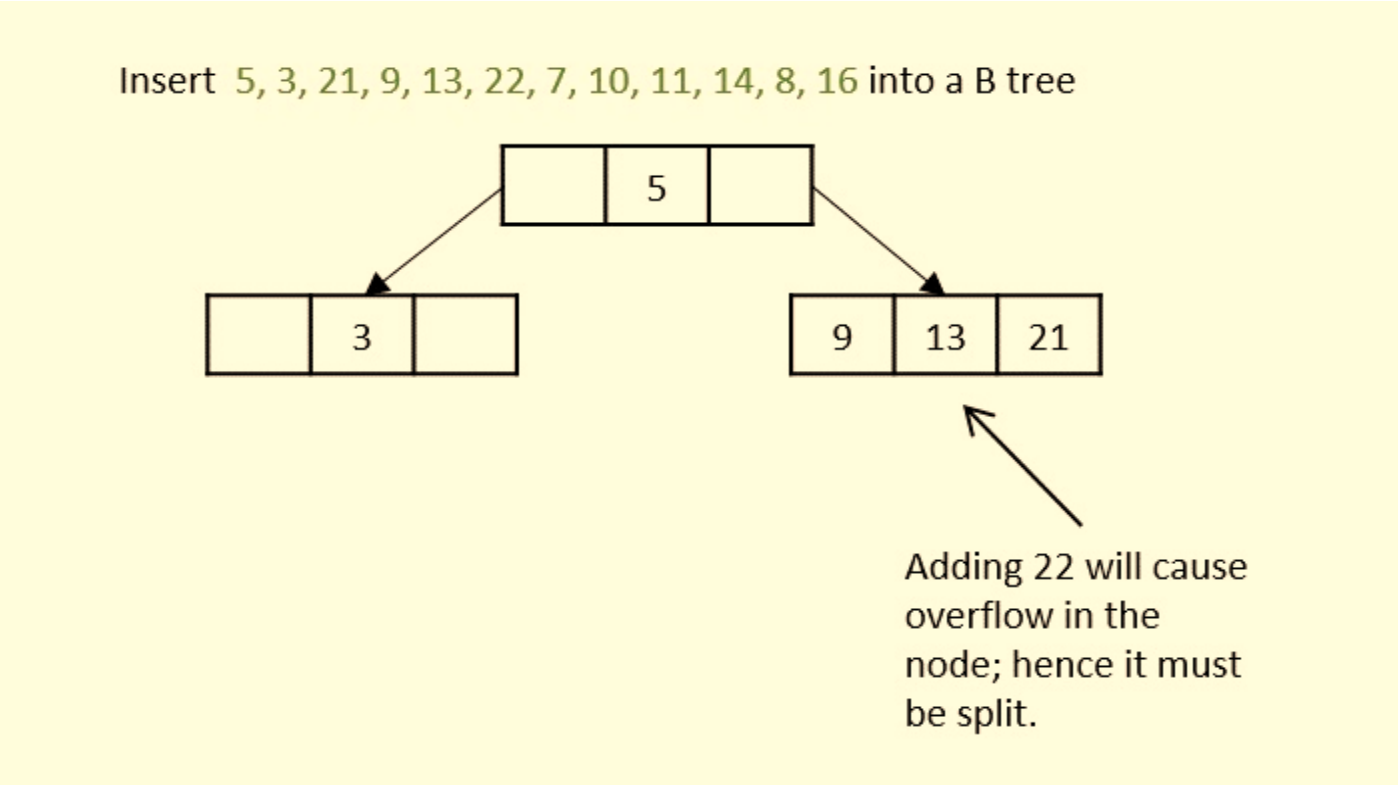
Step 1:



Adding 9 will cause overflow. So, Find the median and split the tree.

B – Tree Insert Operation Example

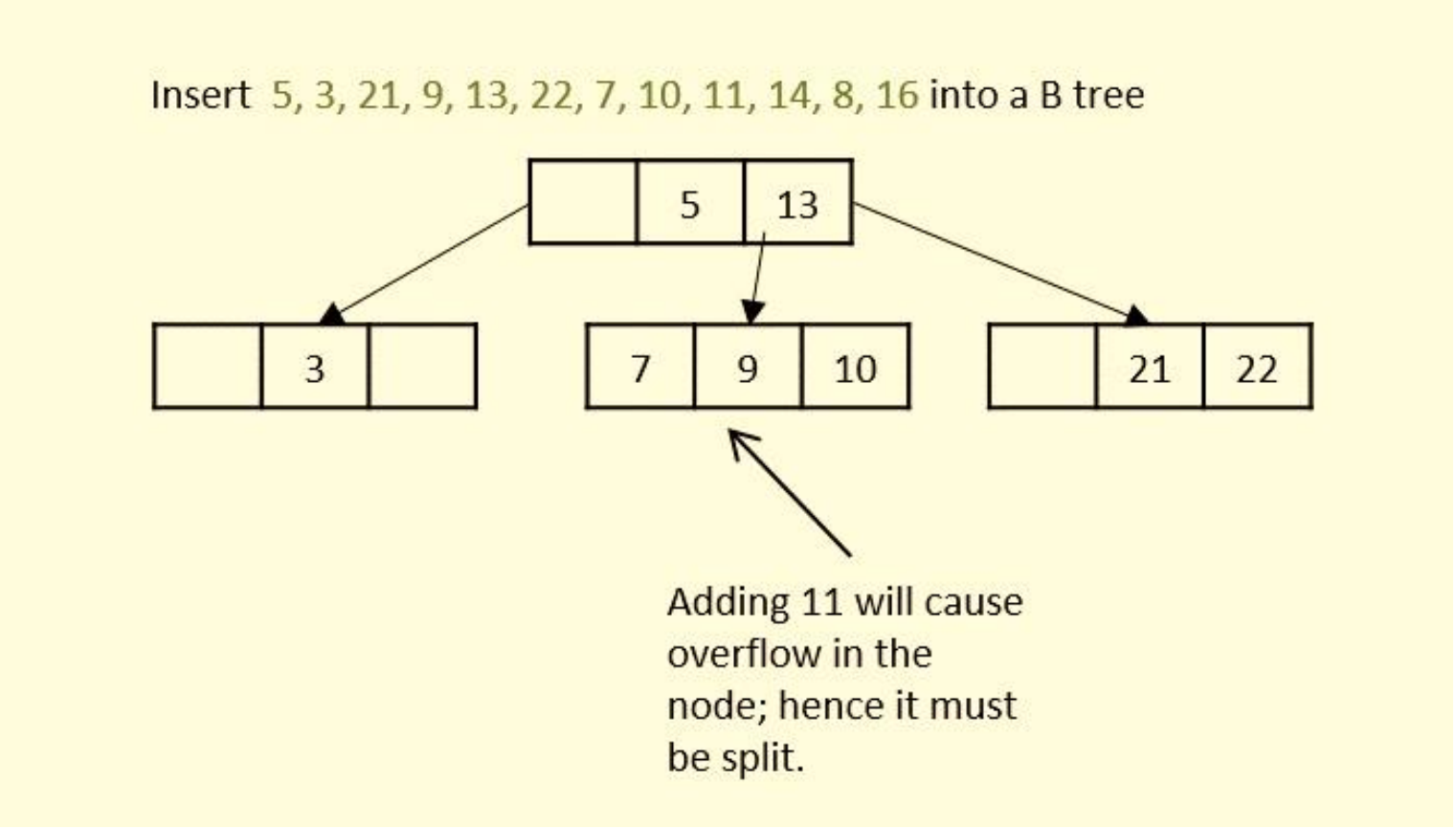
Step 2: The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property.



But if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.

B – Tree Insert Operation Example

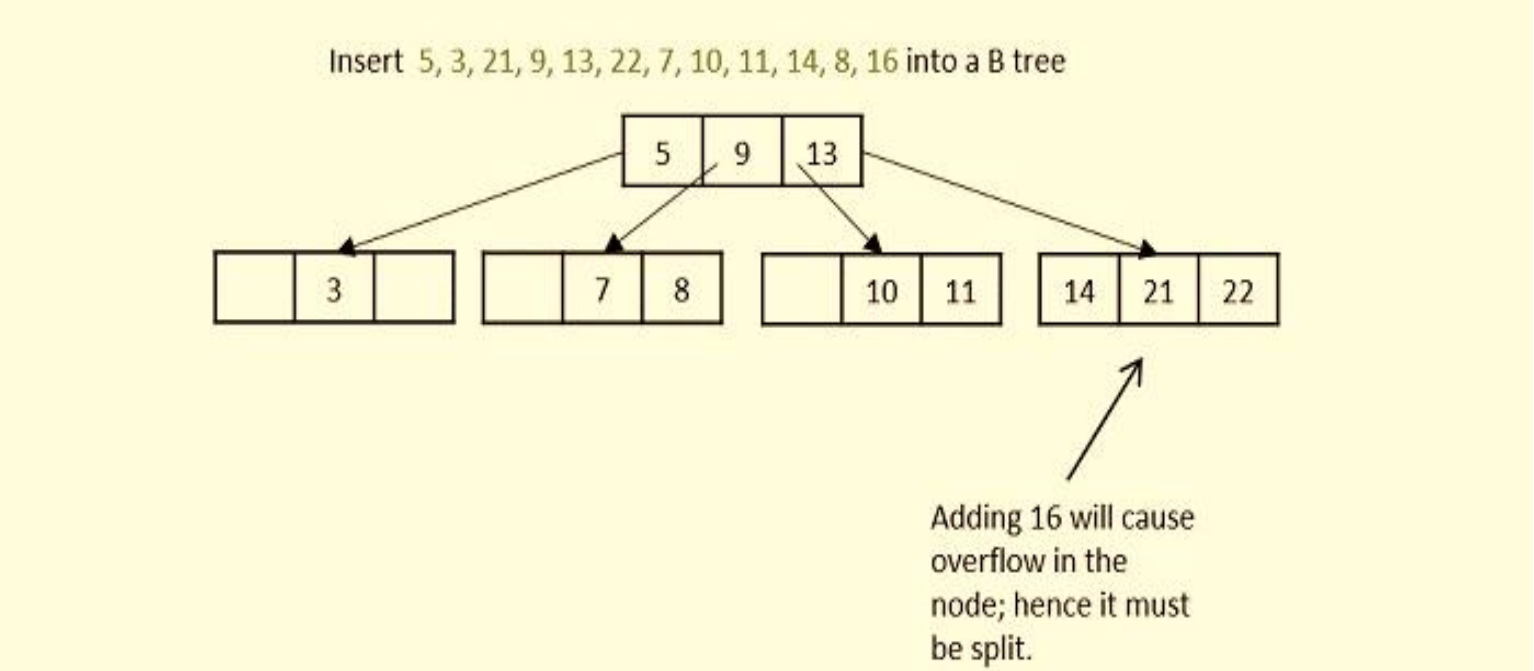
Step 3:



Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.

B – Tree Insert Operation Example

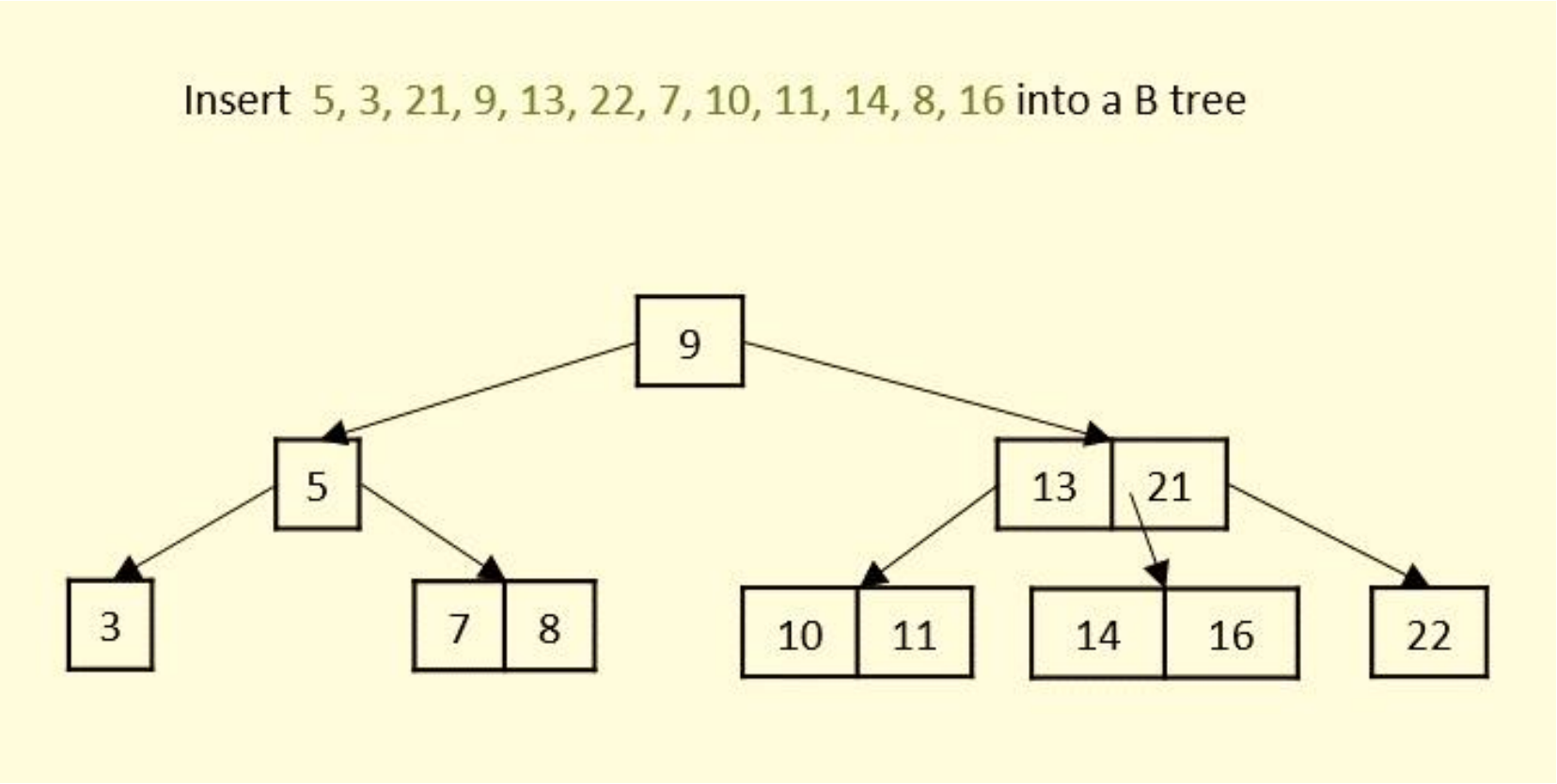
Step 4:



While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.

B – Tree Insert Operation Example

Step 5: The final B tree after inserting all the elements is achieved.



B – Tree Deletion Operation

Delete Operation:

- more complicated than insertion.

- we can delete a key from any node.

- when we delete a key from an internal node, we will have to rearrange the node’s children.

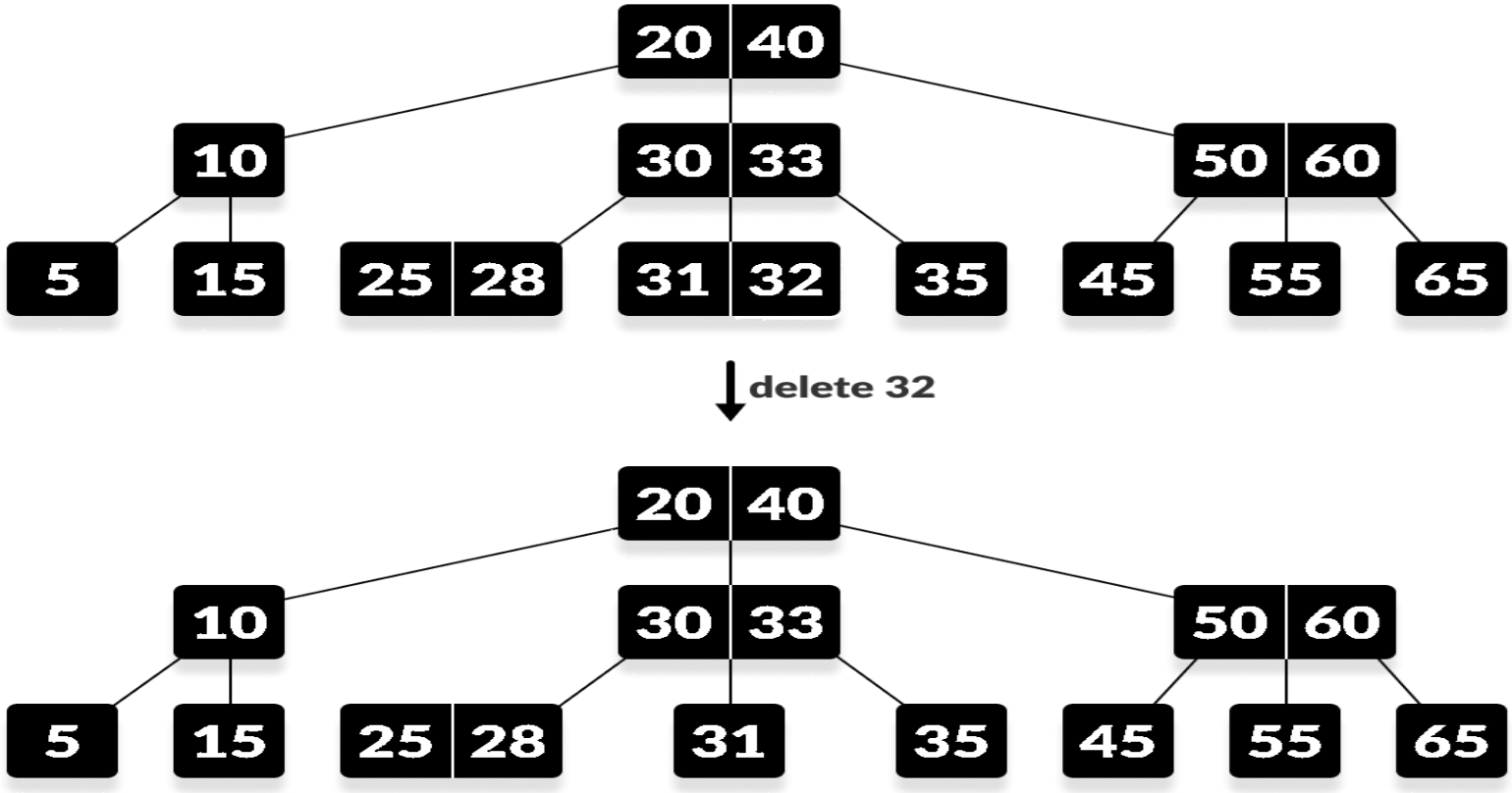
- we must ensure that a node doesn’t get too small during deletion.

B – Tree Deletion Operation Example

Various Cases of Deletion:

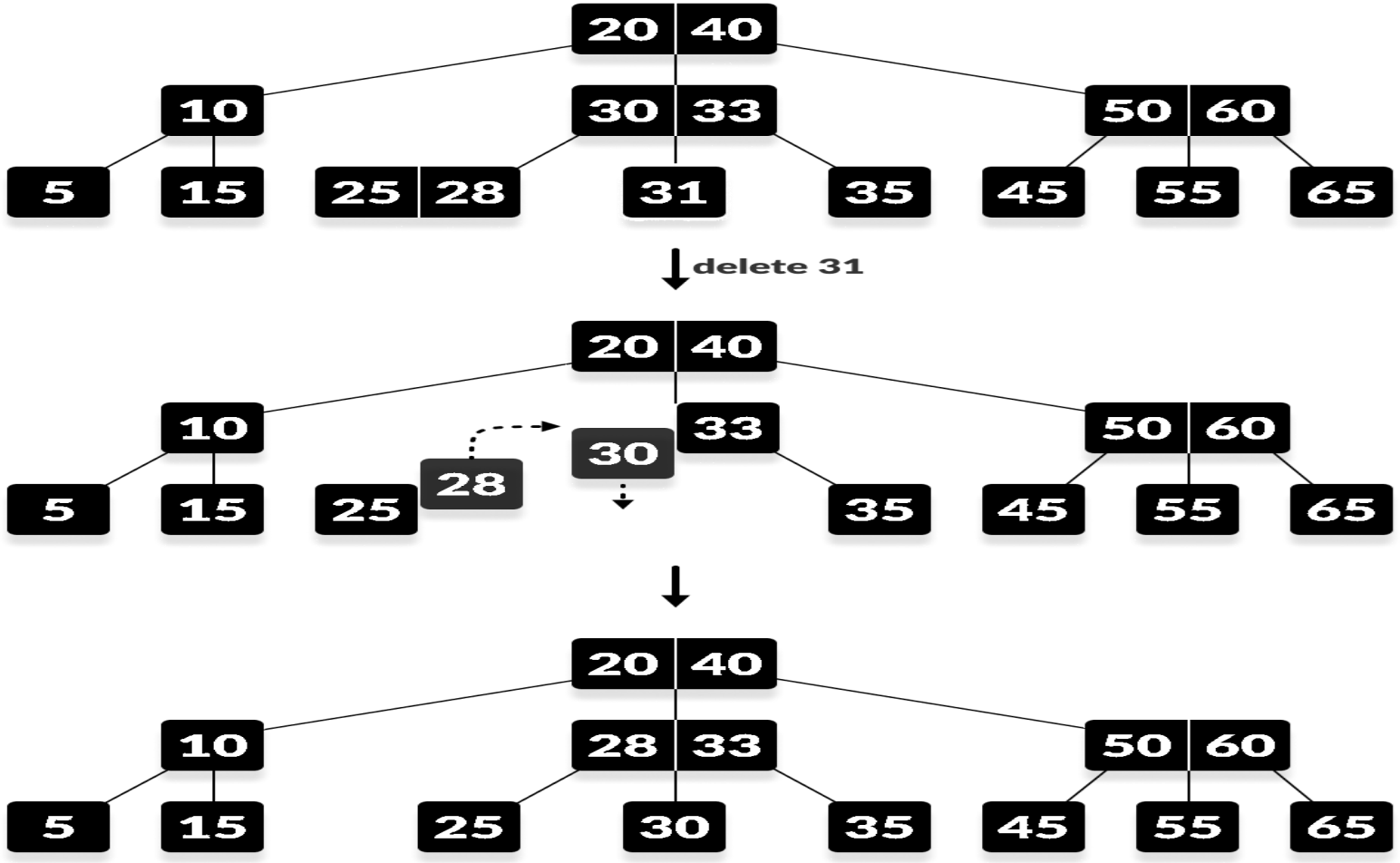
Case 1- If the key k to be deleted is in leaf.

1.1: The deletion does not violate the minimum key property, just delete the node.



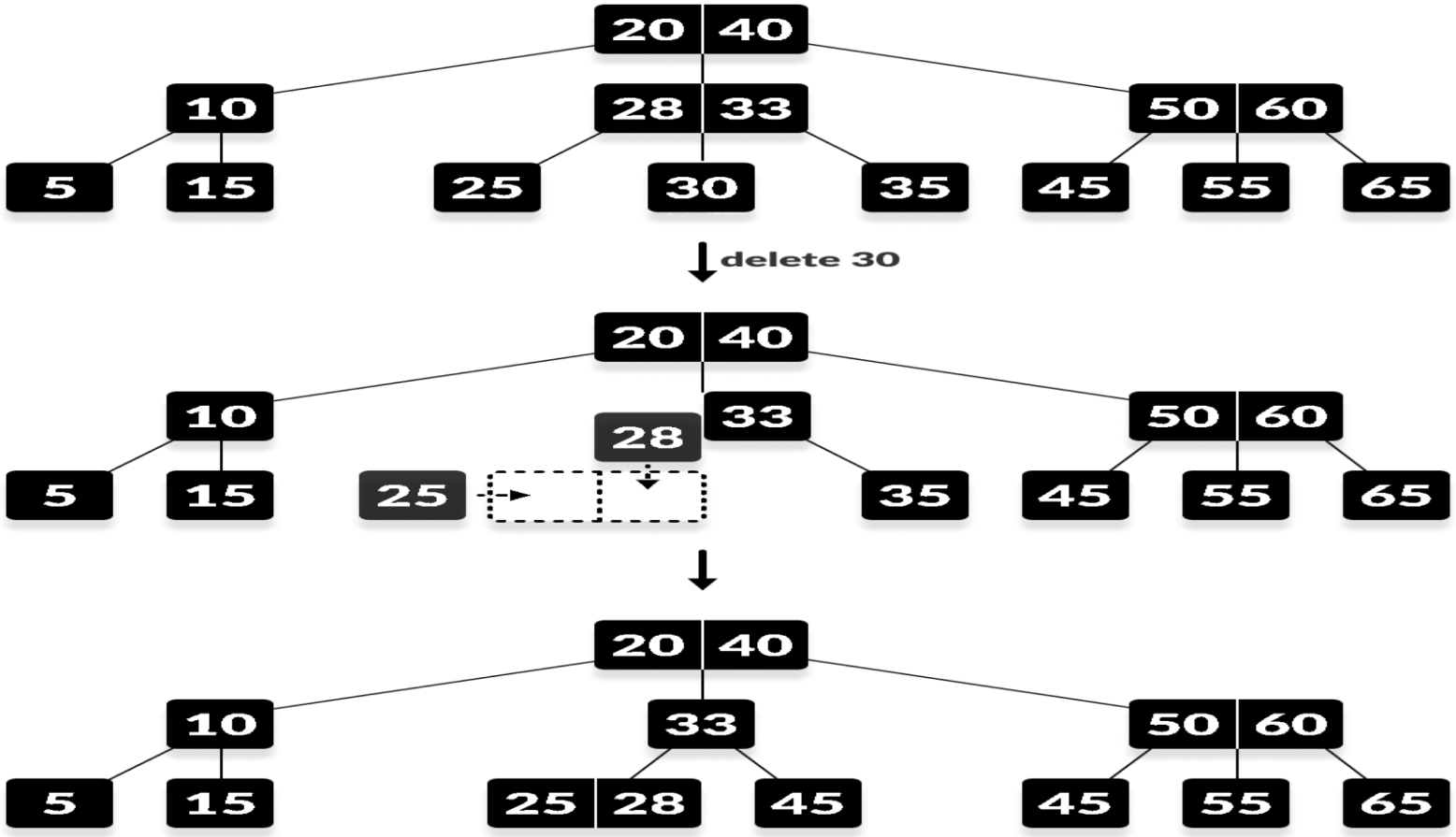
B – Tree Deletion Operation Example

1.2: The deletion violates the minimum key property, borrow a key from either its left sibling or right sibling.



B – Tree Deletion Operation Example

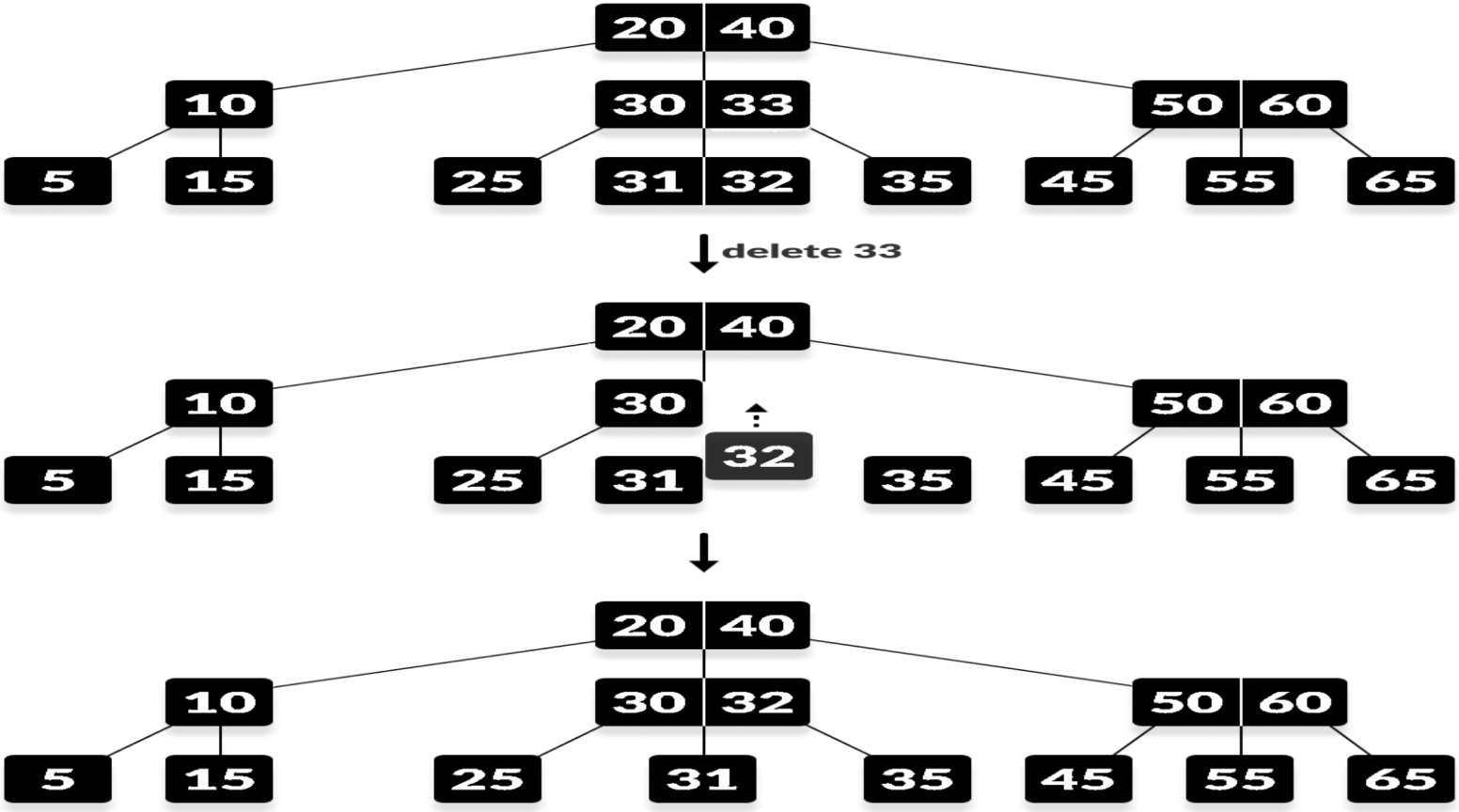
1.3: If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. This merging is done through the parent node.



B – Tree Deletion Operation Example

Case 2: If the key to be deleted lies in the internal node.

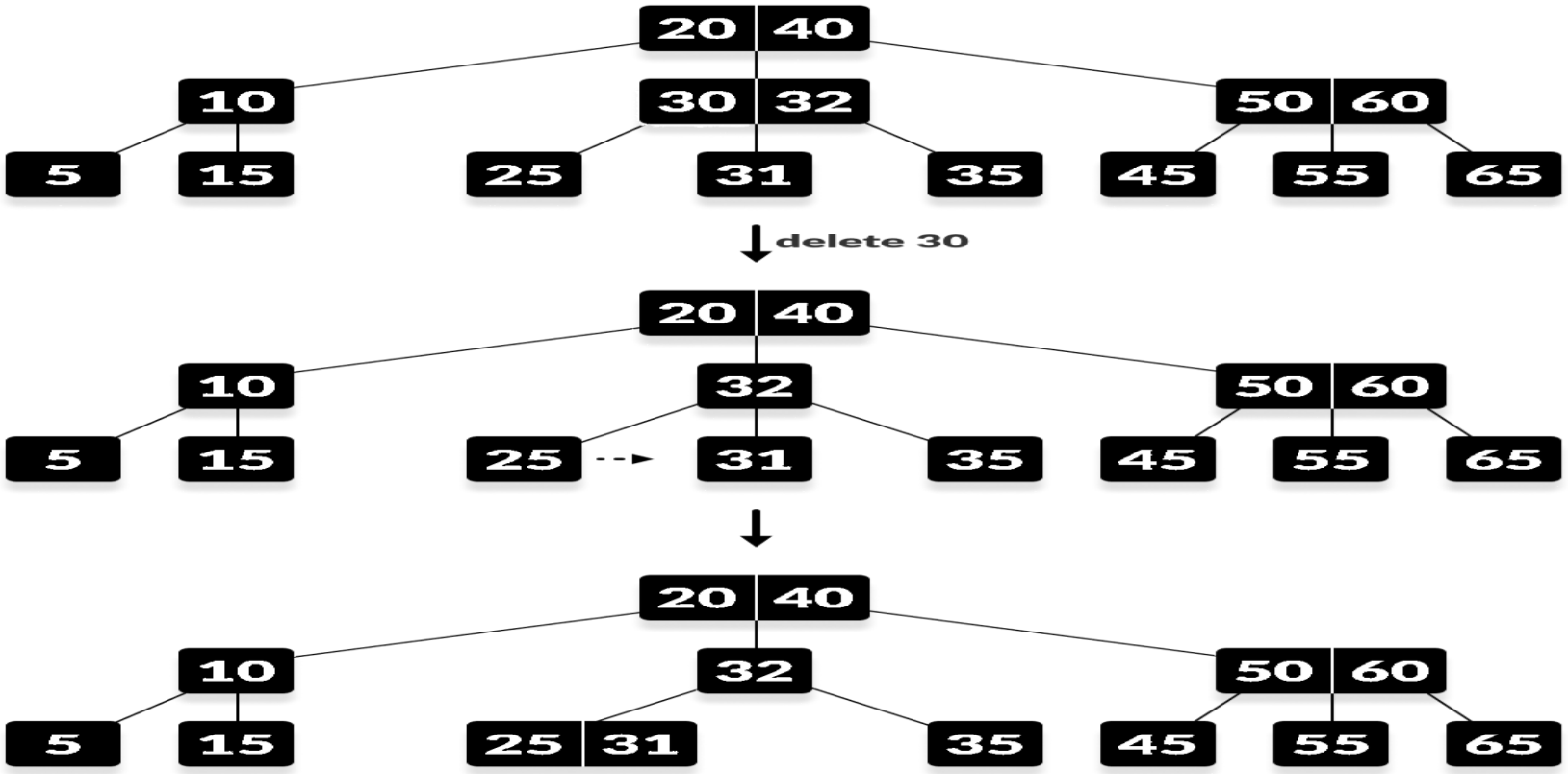
2.1: The internal node is replaced by an inorder predecessor if the left child has more than the minimum number of keys.



B – Tree Deletion Operation Example

2.2: The internal node is replaced by an inorder successor if the right child has more than the minimum number of keys.

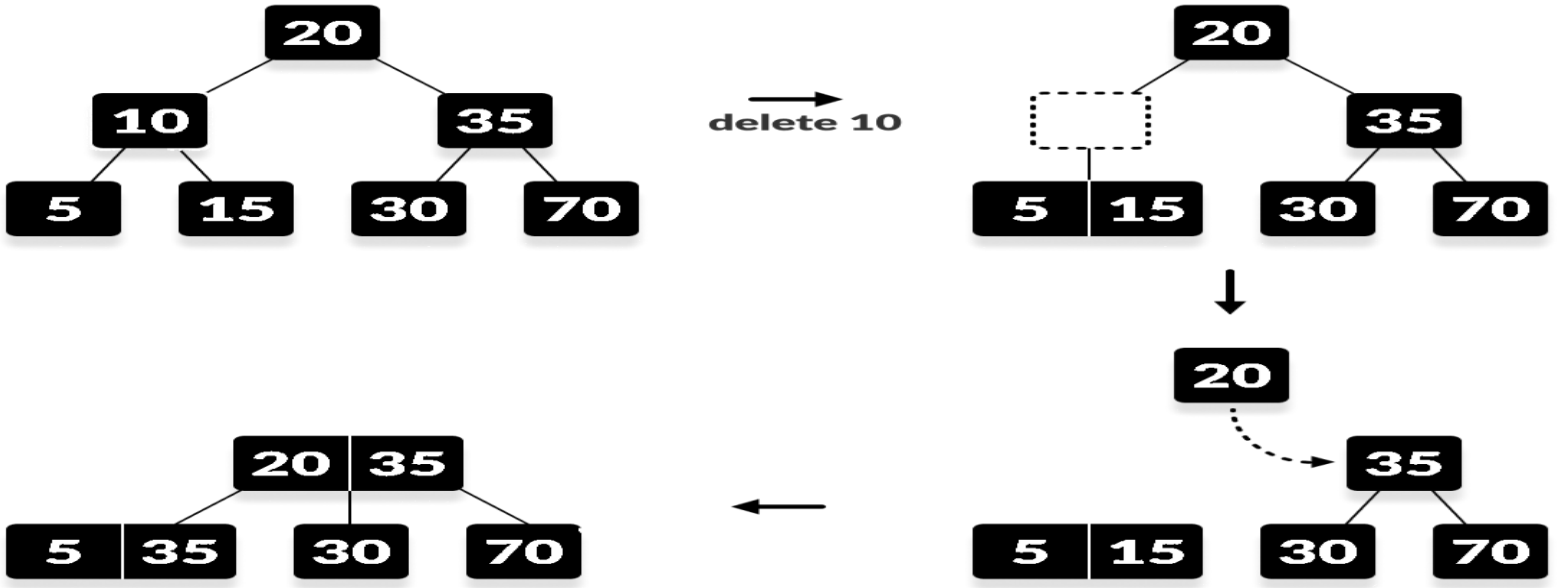
2.3: If either child has exactly a minimum number of keys then, merge the left and the right children.



B – Tree Deletion Operation Example

Case 3: If the key to be deleted lies in the internal node and both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case 2.3 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

- The B+ trees are extensions of B trees designed to make the insertion, deletion and searching operations more efficient
- The properties of B+ trees are similar to the properties of B trees.
- Except that B+ trees store records in leaf nodes and keys in internal nodes.
- All the leaf nodes are connected to each other in a single linked list format and a data pointer is available to point to the data present.

B + Tree Properties

- Every node in a B+ Tree, except root, will hold a maximum of m children and (m-1) keys, and a minimum of (m/2) children and (m/2−1) keys, since the order of the tree is m.
- The root node must have no less than two children and at least one search key.
- All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.
- A B+ tree always maintains sorted data.

B + Tree Insert Operation

- The insertion to a B+ tree starts at a leaf node.

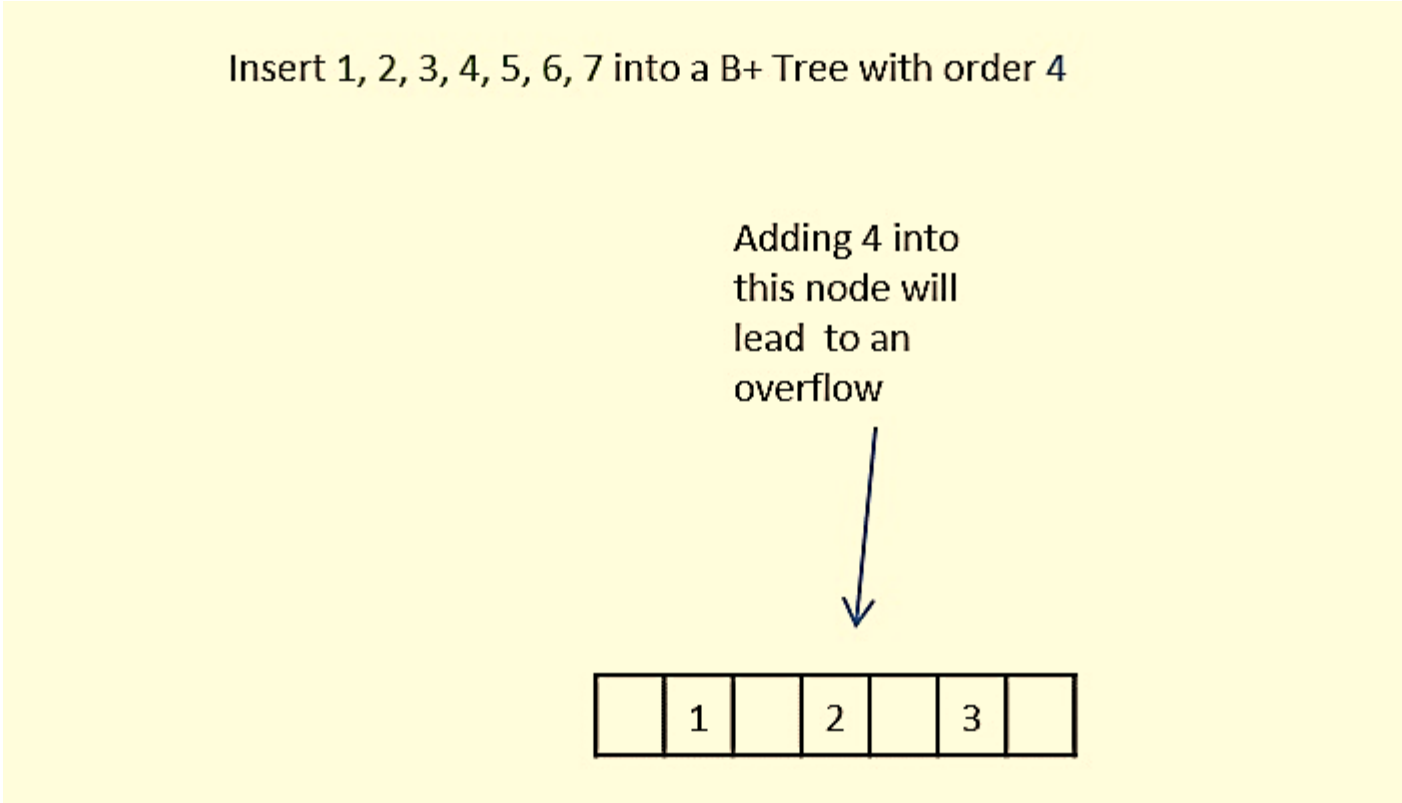
Step 1 − Calculate the maximum and minimum number of keys to be added onto the B+ tree node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Order = 4
Maximum Children (m) = 4
Minimum Children ($\left\lceil \frac{m}{2} \right\rceil$) = 2
Maximum Keys (m - 1) = 3
Minimum Keys ($\left\lceil \frac{m-1}{2} \right\rceil$) = 1

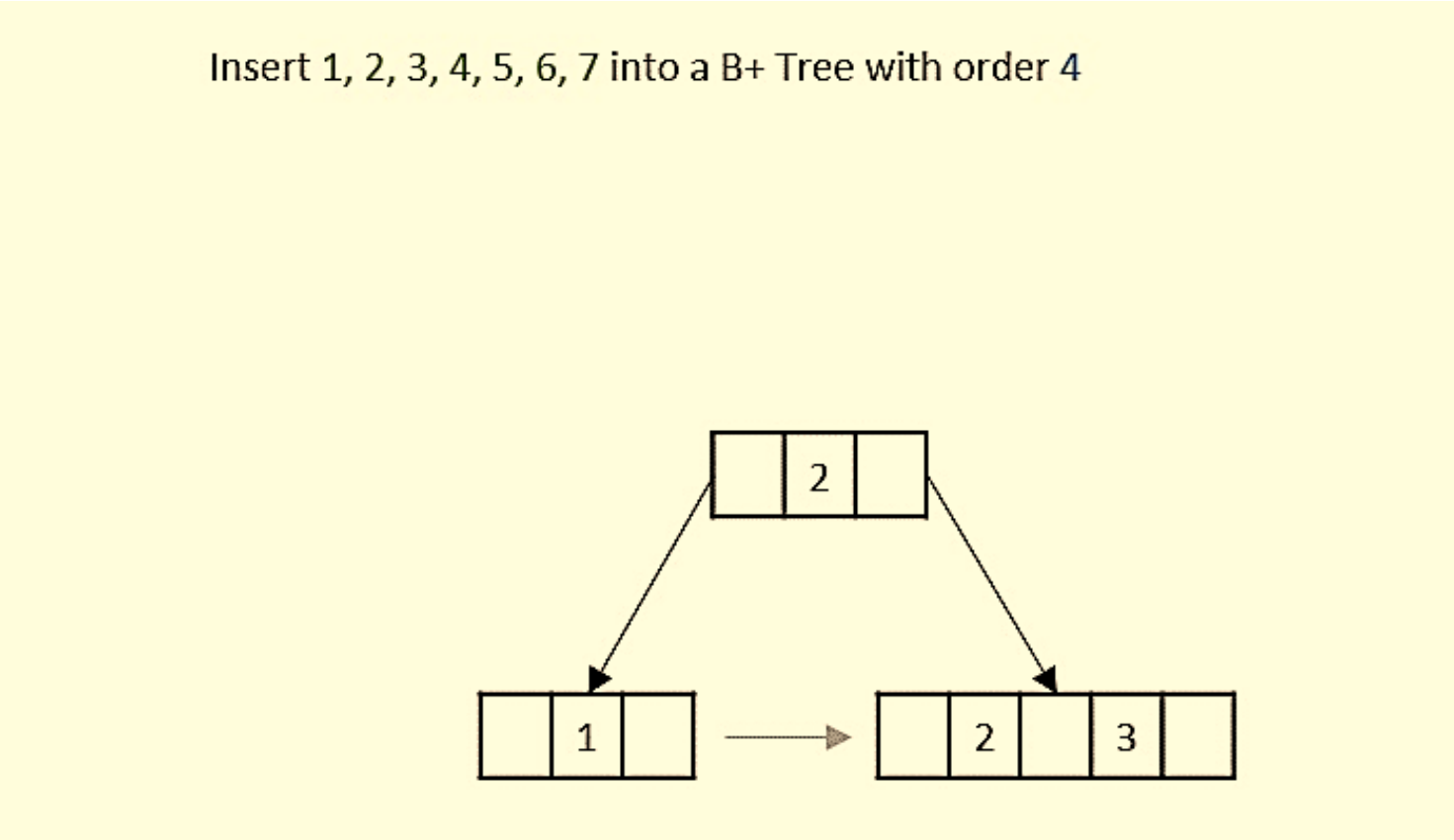
B + Tree Insert Operation

Step 2 - Insert the elements one by one accordingly into a leaf node until it exceeds the maximum key number.



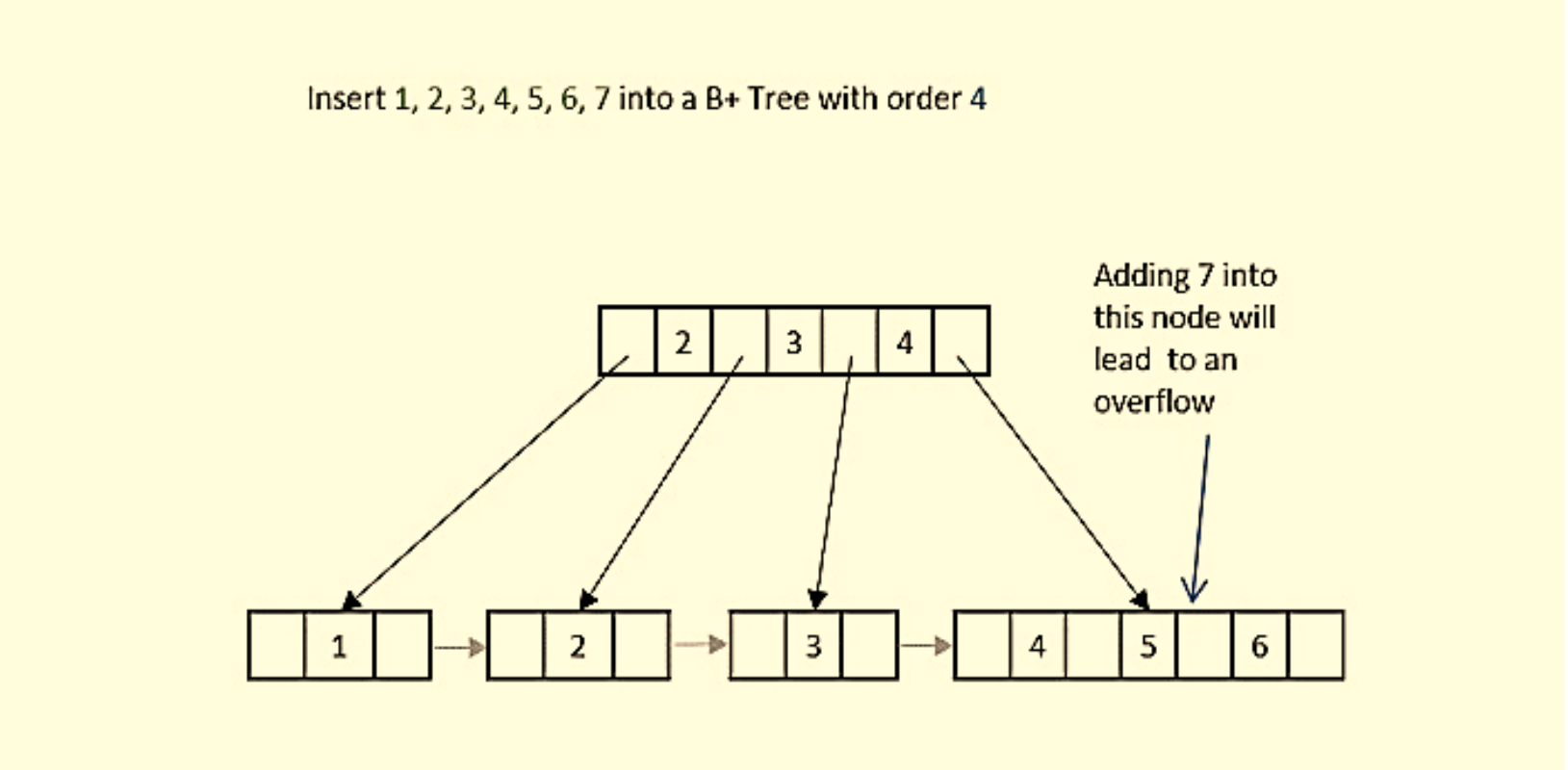
B + Tree Insert Operation

Step 3 - The node is split into half where the left child consists of minimum number of keys and the remaining keys are stored in the right child.



B + Tree Insert Operation

Step 4 - But if the internal node also exceeds the maximum key property, the node is split in half where the left child consists of the minimum keys and remaining keys are stored in the right child. However, the smallest number in the right child is made the parent.



B + Tree Insert Operation

Step 5 - If both the leaf node and internal node have the maximum keys, both of them are split in the similar manner and the smallest key in the right child is added to the parent node.