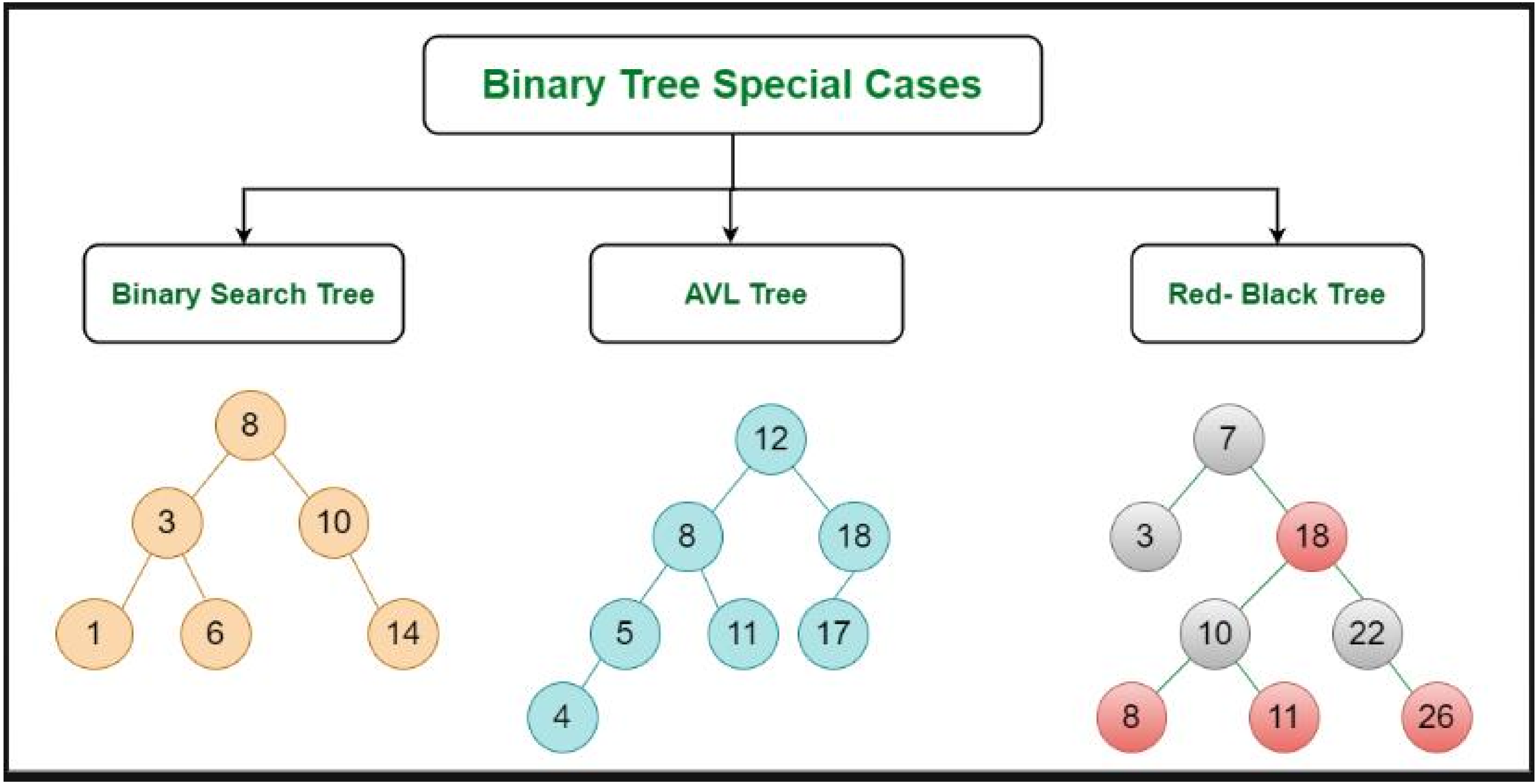
TREES

CODETANTRA


Binary Tree Special Cases
Binary Search Tree
AVL Tree
Red- Black Tree
8
3
10
1
6
14
12
8
18
5
11
17
4
7
3
18
10
22
8
11
26

Binary Search Trees

Binary search trees (BSTs) are a type of binary tree where every node follows the property that values less than the node's value are located in the left subtree, and values greater than the node's value are located in the right subtree. Traversing a BST in different orders yields different sequences of the nodes' values:

Binary search trees help us speed up our binary search as we are able to find items faster.

We can use the binary search tree for the addition and deletion of items in a tree.

We can also represent data in a ranked order using a binary tree. And in some cases, it can be used as a chart to represent a collection of information.

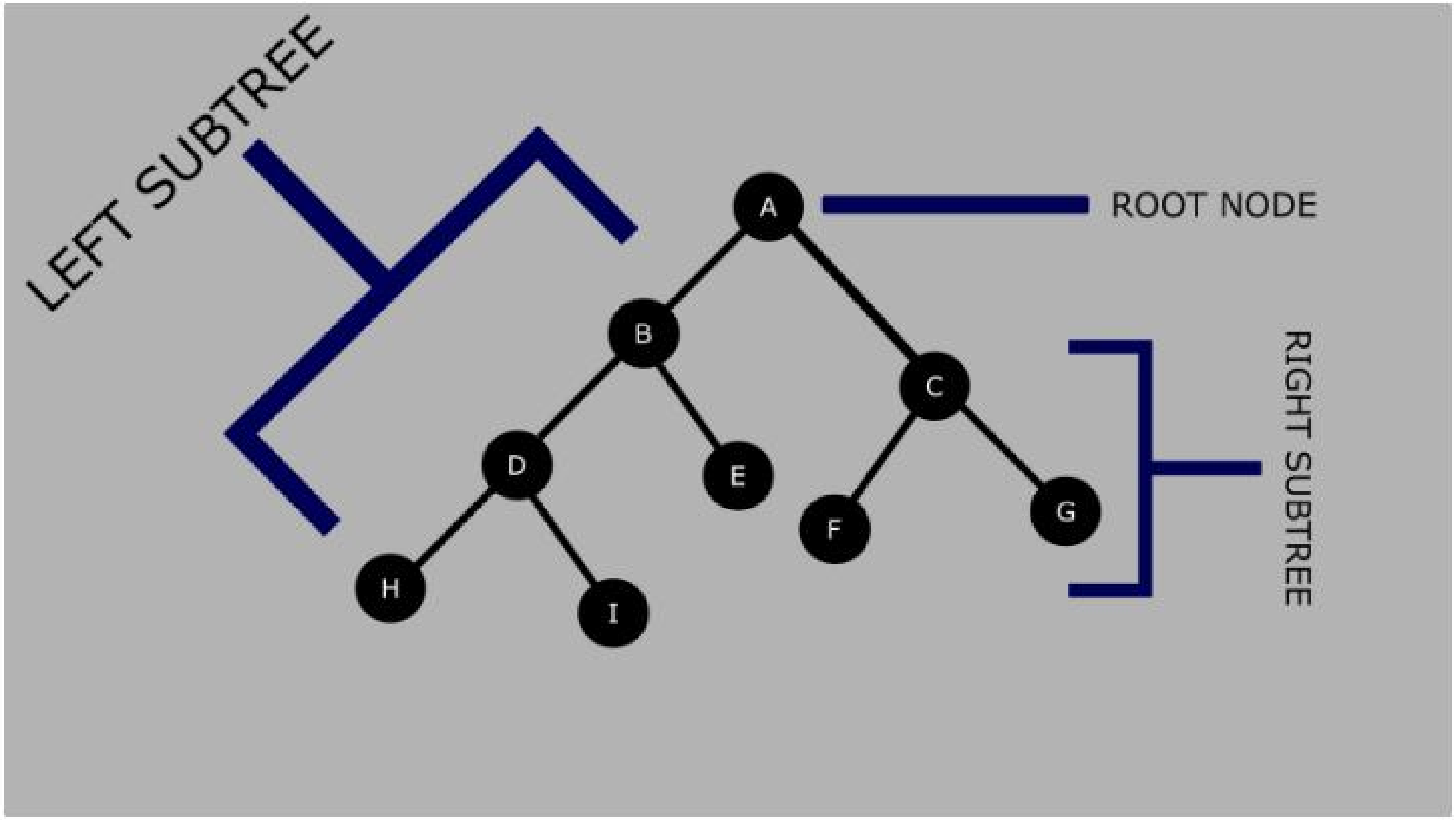


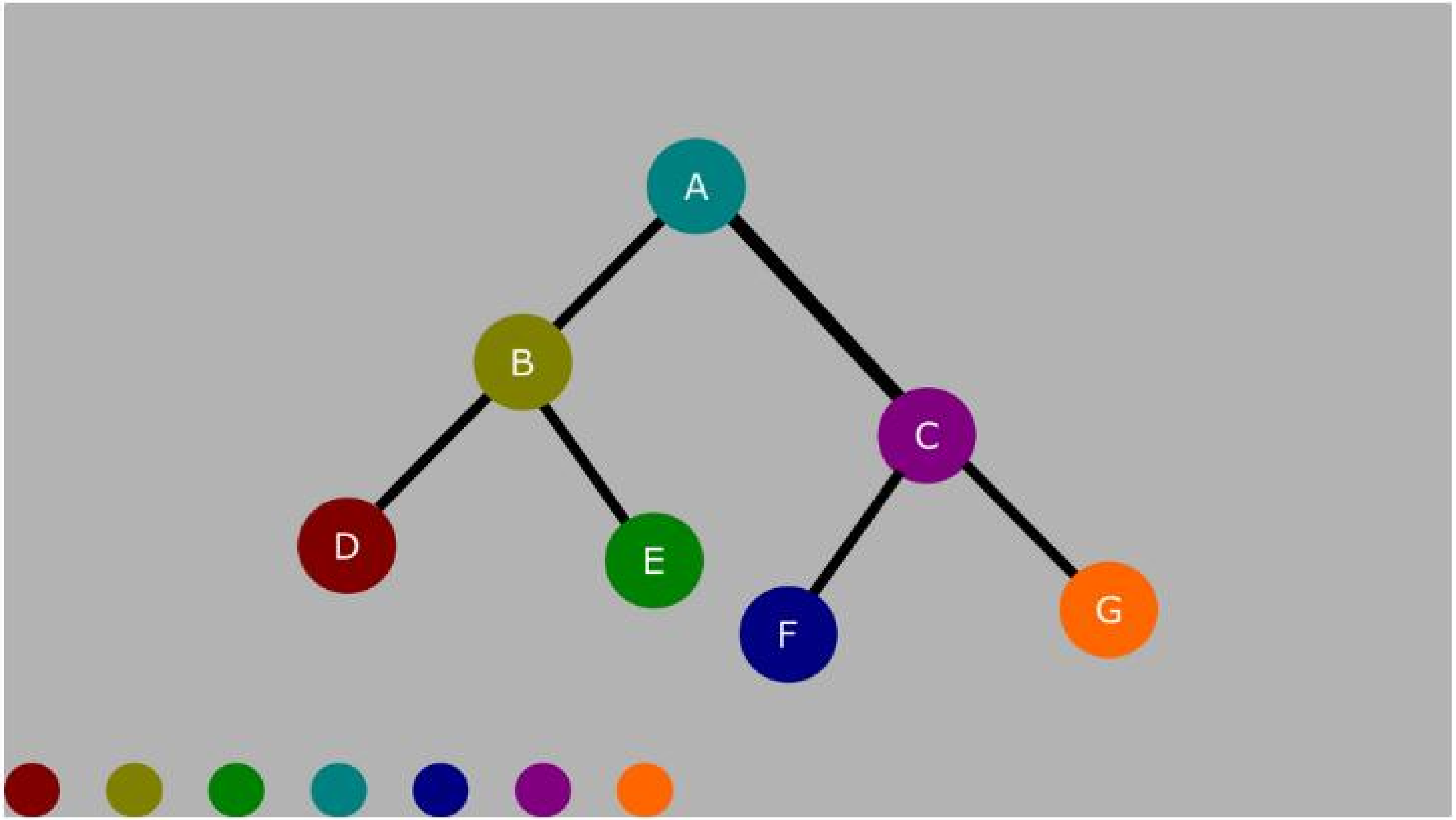
Diagram of a binary search tree

Inorder Traversal: In this traversal, nodes are visited in the order left subtree - node itself - right subtree.
In inorder traversal, the nodes of the binary tree are visited in the following order:

1. Visit the left subtree.
2. Visit the current node.
3. Visit the right subtree.

In other words, for each node, the left subtree is traversed first, then the current node is visited, and finally, the right subtree is traversed.

In a BST, inorder traversal results in visiting nodes in ascending order.



Inorder traversal

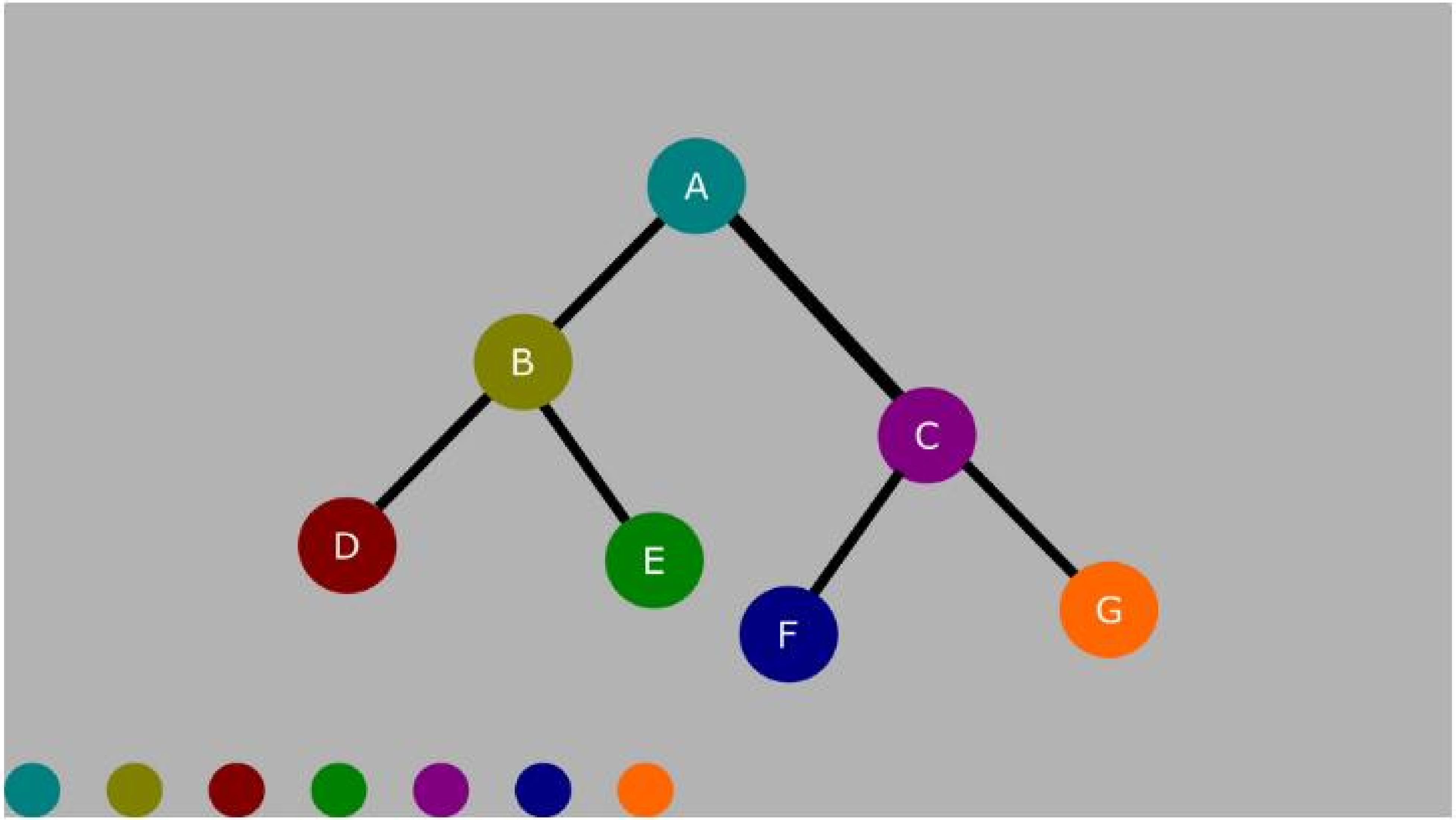
Preorder Traversal: In this traversal, nodes are visited in the order node itself - left subtree - right subtree.

In preorder traversal, the nodes of the binary tree are visited in the following order:

1. Visit the current node.
2. Visit the left subtree.
3. Visit the right subtree.

In other words, for each node, the current node is visited first, then its left subtree is traversed, followed by its right subtree.

Preorder traversal is useful for creating a copy of a tree, as it can be used to generate a copy of the tree by visiting nodes in the order they should be copied.



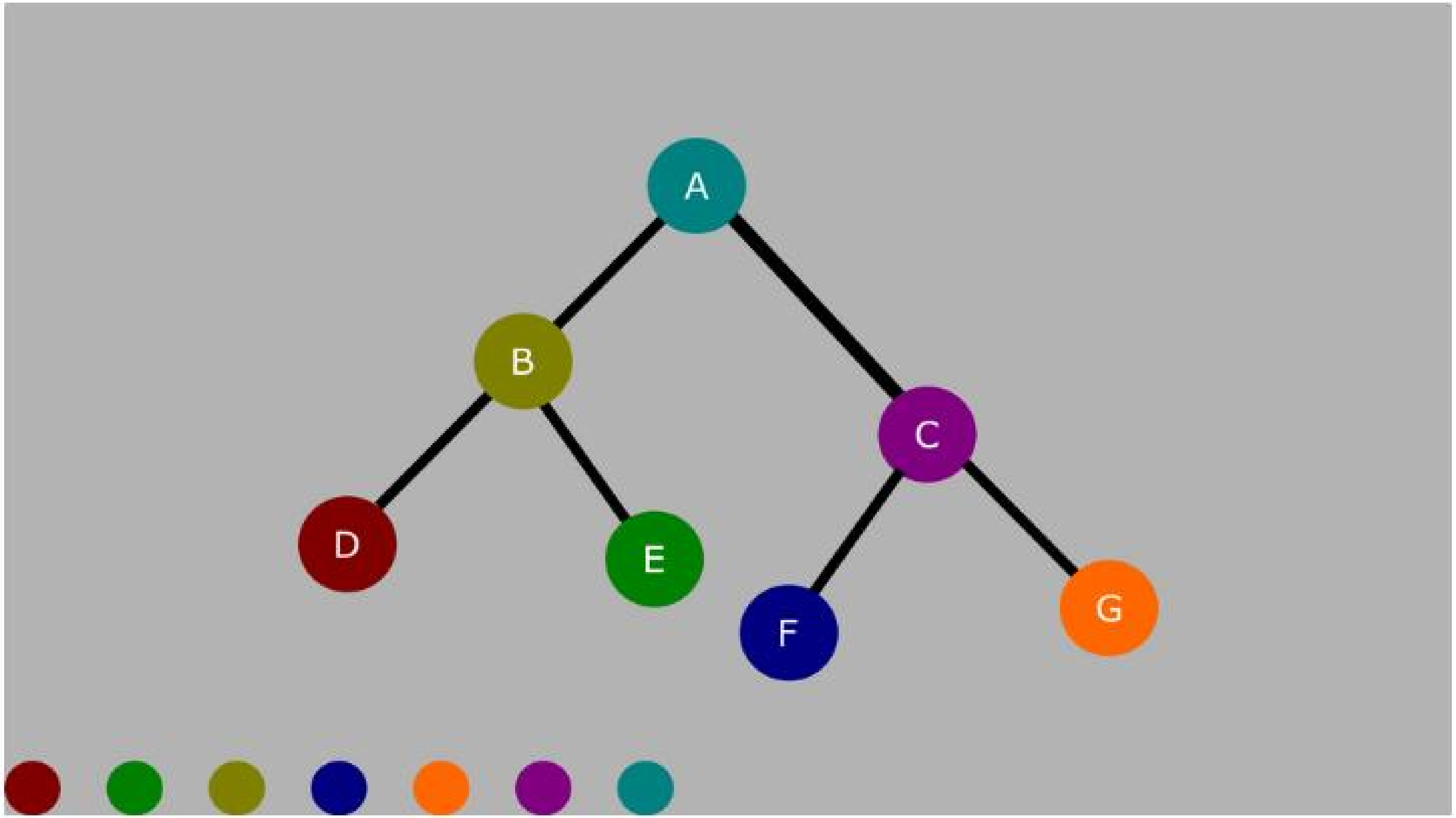
Preorder traversal

Postorder Traversal: In this traversal, nodes are visited in the order left subtree - right subtree - node itself.
In postorder traversal, the nodes of the binary tree are visited in the following order:

1. Visit the left subtree.
2. Visit the right subtree.
3. Visit the current node.

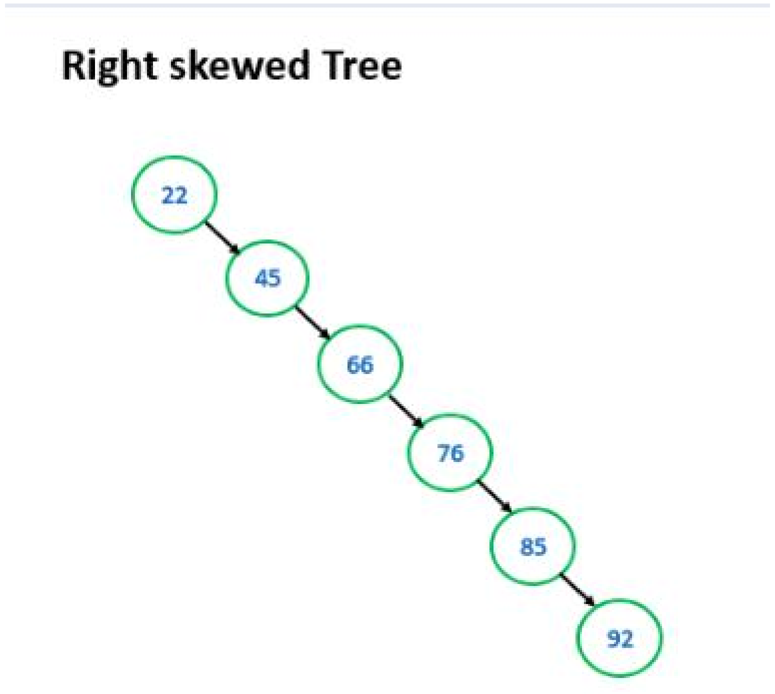
In other words, for each node, the left subtree is traversed first, then the right subtree is traversed, and finally, the current node is visited.

Postorder traversal is useful for deleting a tree, as it ensures that a node is deleted only after deleting its children.



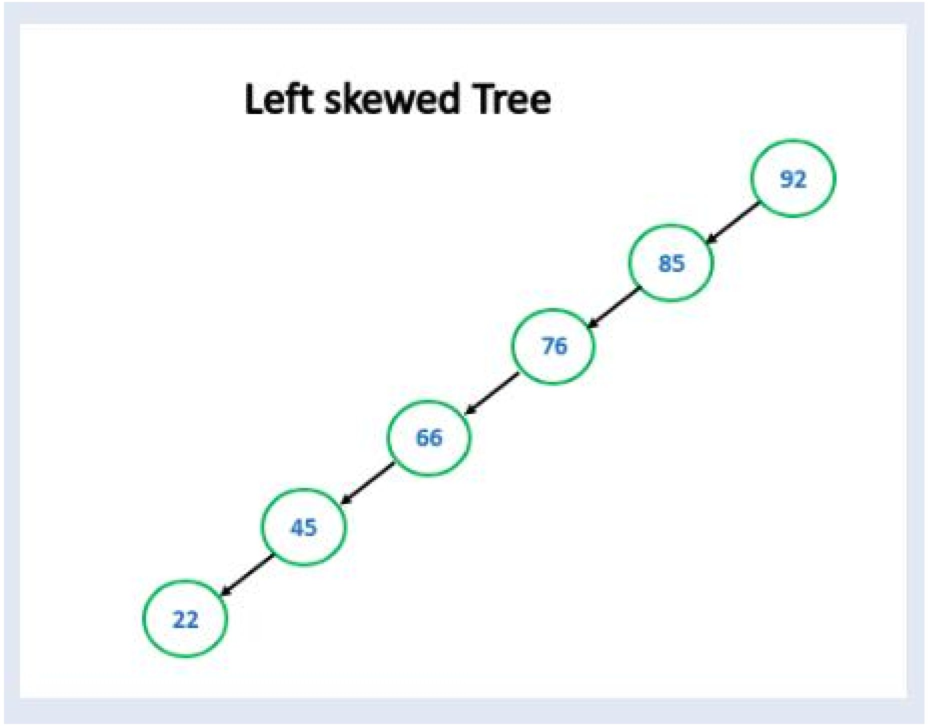
Postorder traversal

Imagine starting with an empty binary search tree and inserting 22, 45, 66, 76, 85, 92 into the binary search tree in that order.
The output of the tree would be



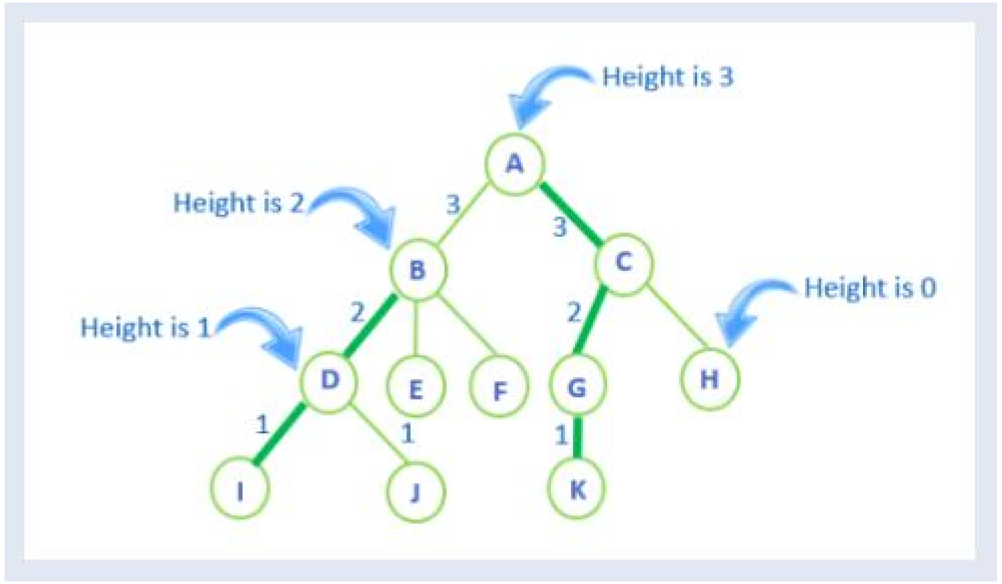
The tree shown above is called a "Right Skewed Tree" as all the nodes have the right child without corresponding left child.

Imagine starting with an empty binary search tree and inserting 92, 85, 76, 66, 45, 22 into the binary search tree in that order.
The output of the tree would be



The tree shown above is called a "Left Skewed Tree" as all the nodes have the left child without corresponding right child.

In both the cases , the tree is not a branching tree, but a linear tree or skewed tree. Because of this behavior, in the worst case each of the operations (search, insertion and deletion) takes time O(n) compared to O(log n).
To solve the above problem, the concept of height balanced trees is introduced.
Height-balanced trees
Recollecting , The height of a node in a tree is the length of the longest path from that node downward to a leaf, counting both the start and end vertices of the path. The height of all leaf nodes is 0. The height of a tree is the height of its root.



Height-balancing requirement

A node in a tree is height-balanced if the heights of its sub trees differ by no more than 1. (That is, if the sub trees have heights h1 and h2, then $|h1 - h2| \le 1$.).

The difference of heights of left sub tree and right sub tree of a node is also called as "Balance Factor"

BalanceFactor(N) := Height(RightSubtree(N)) – Height(LeftSubtree(N))

A tree is height-balanced if all of its nodes are height-balanced.

There are several popular height balanced trees. There are :

AVL tree

B-tree

Red-Black tree

Splay tree

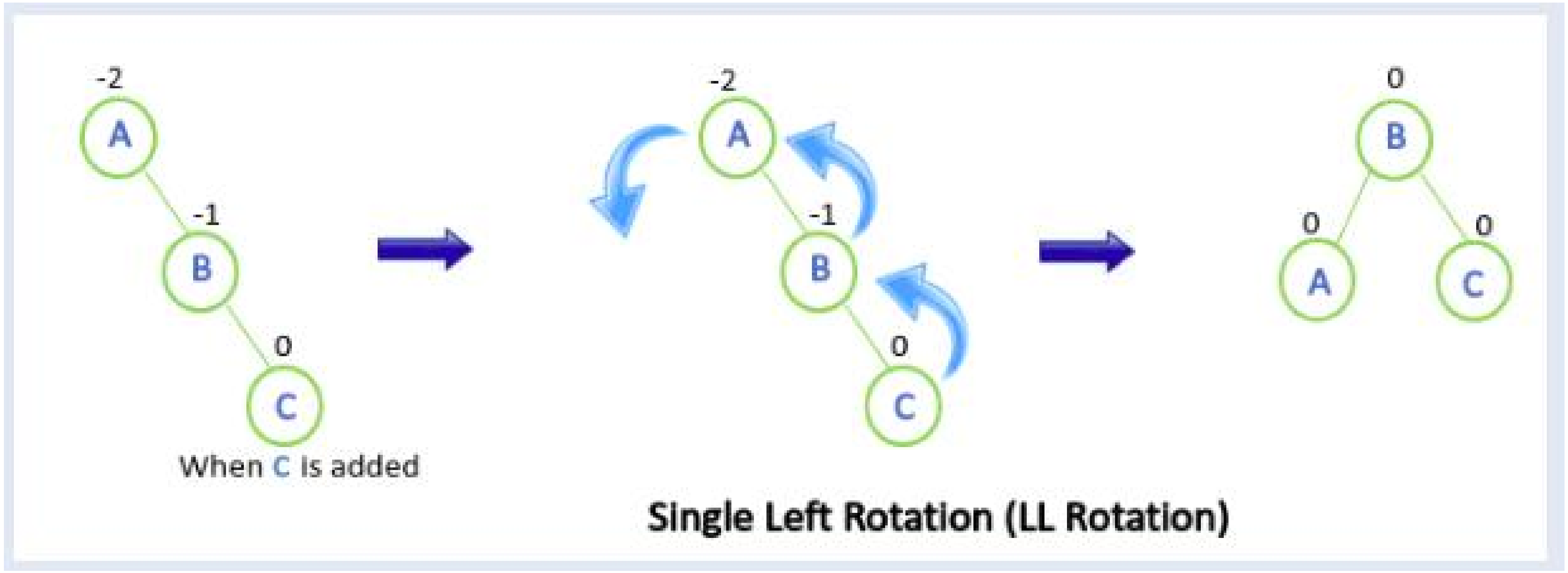
2-3 tree

AVL Trees:

In AVL Trees, rotations are like rearranging books on our shelf but in a specific way. These rotations are the key to maintaining balance. There are four types of rotations: Left-Left (LL), Right-Right (RR), Left-Right (LR), and Right-Left (RL).

LL Rotation (Right Rotation):

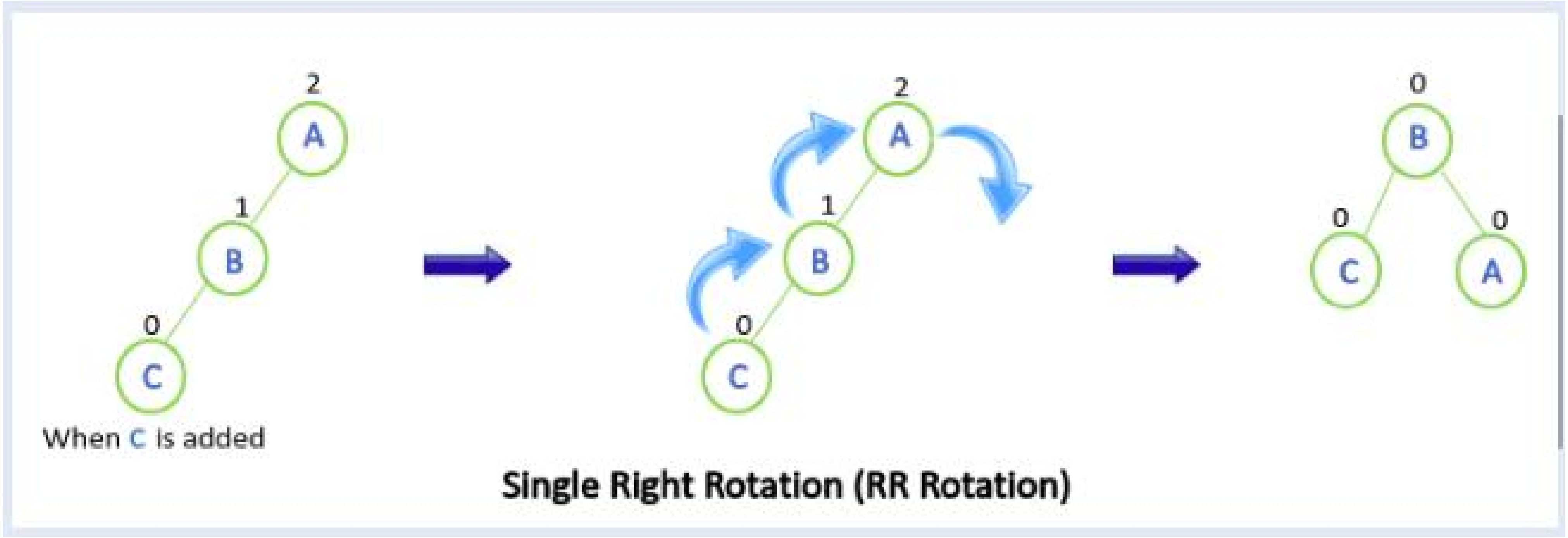
Imagine your books are leaning too much to the left. To balance the shelf, you would perform a Right Rotation. This is known as LL Rotation in AVL Trees.



Single Left Rotation (LL Rotation)

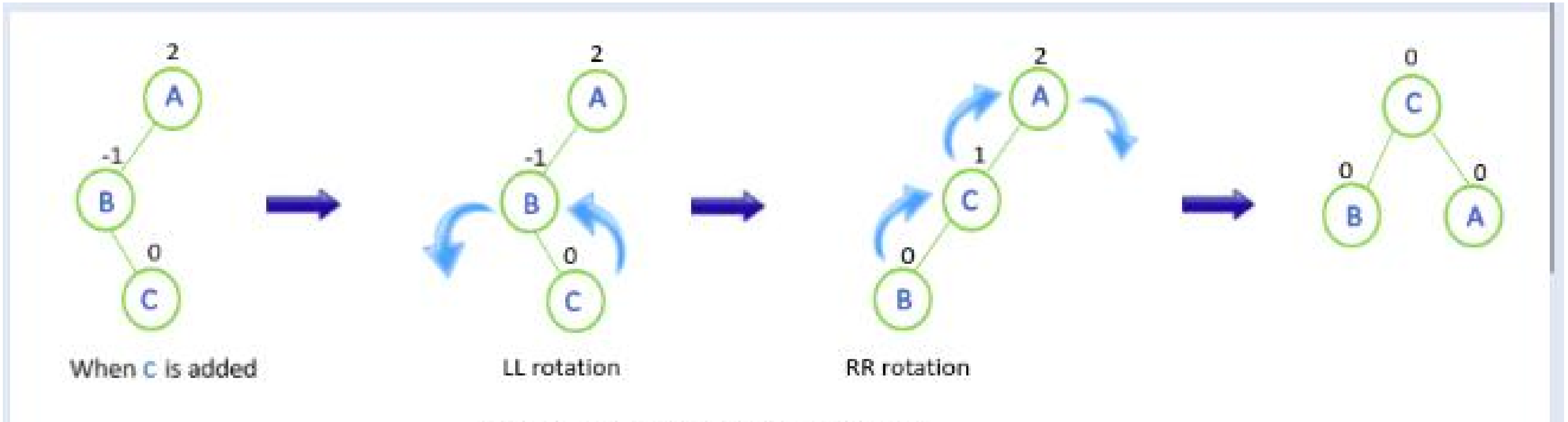
RR Rotation (Left Rotation):

Conversely, if your books are leaning too much to the right, you would perform a Left Rotation or RR Rotation in AVL Trees.



LR Rotation (Left-Right Rotation):

If the books are leaning left and a bit forward, you perform a combination of Left and Right Rotations. This is called LR Rotation.

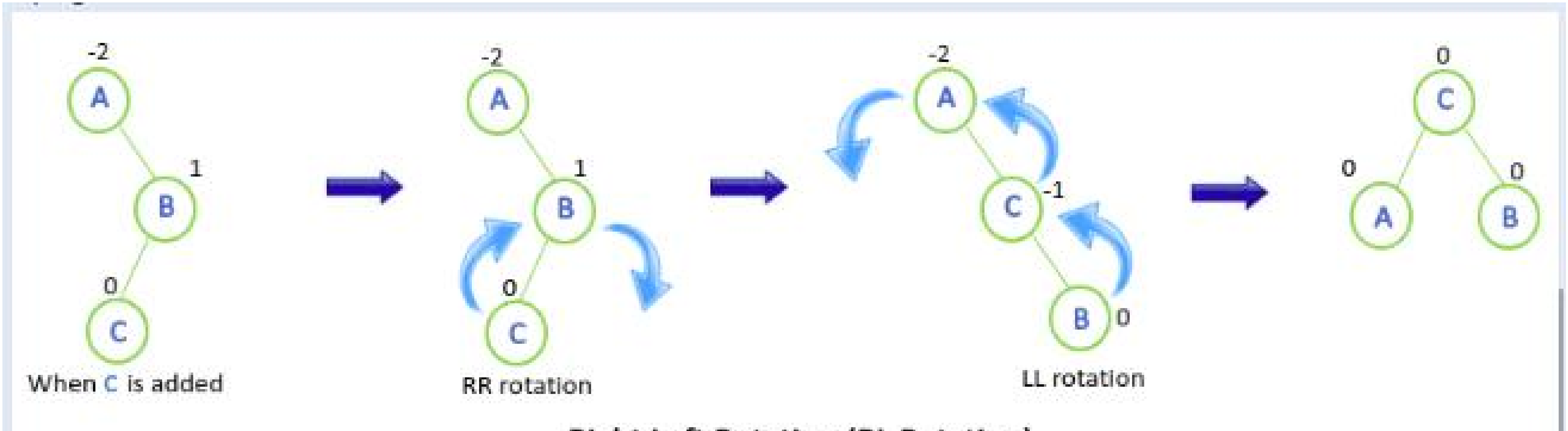


Left Right Rotation (LR Rotation)

RL Rotation (Right-Left Rotation):

Similarly, if the books are leaning right and a bit forward, you perform a combination of Right and Left Rotations, known as RL Rotation.

Rotations may seem complex, but they are like neat tricks to keep our tree balanced, ensuring it stays upright.



Right Left Rotation (RL Rotation)

Insertion Operation in AVL Trees:

Perform BST Insertion: Initially, the new node is inserted into the AVL tree just like in a normal binary search tree according to the rules of binary search.

Update Heights: After insertion, update the height of each node starting from the newly inserted node's parent up to the root of the tree.

Check Balance Factor: After updating heights, check the balance factor of each node (the difference in heights between the left and right subtrees).

Rebalance if Necessary:

- If the balance factor of any node violates the AVL property (i.e., becomes greater than 1 or less than -1), then the tree needs rebalancing.
- Perform rotations (single or double rotations) to restore the balance of the tree while maintaining the binary search tree property.

Deletion Operation in AVL Trees:

Perform BST Deletion: Initially, the node to be deleted is removed from the AVL tree just like in a normal binary search tree according to the rules of binary search.

Update Heights: After deletion, update the height of each node starting from the parent of the deleted node up to the root of the tree.

Check Balance Factor: After updating heights, check the balance factor of each node along the path from the deleted node's parent up to the root.

Rebalance if Necessary:

- If the balance factor of any node violates the AVL property, perform rotations (single or double rotations) to restore the balance of the tree while maintaining the binary search tree property.

Splay Tree:

Splay tree is also self-balancing binary search tree like AVL Tree and Red-Black tree.
The main idea of splay tree is to bring 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.

These trees are very optimal in a case where only a few keys are frequently accessed, which is prominent in many applications.

All splay tree operations run in O(log n) time on average, where n is the number of nodes in the tree. Any single operation can take Θ(n) time in the worst case.

NOTE : Every operation (insertion, deletion or search) on a splay tree performs the splaying operation.

The insertion operation first inserts the new element as it inserted into the binary search tree, after insertion the newly inserted element is splayed so that it is placed at root of the tree.

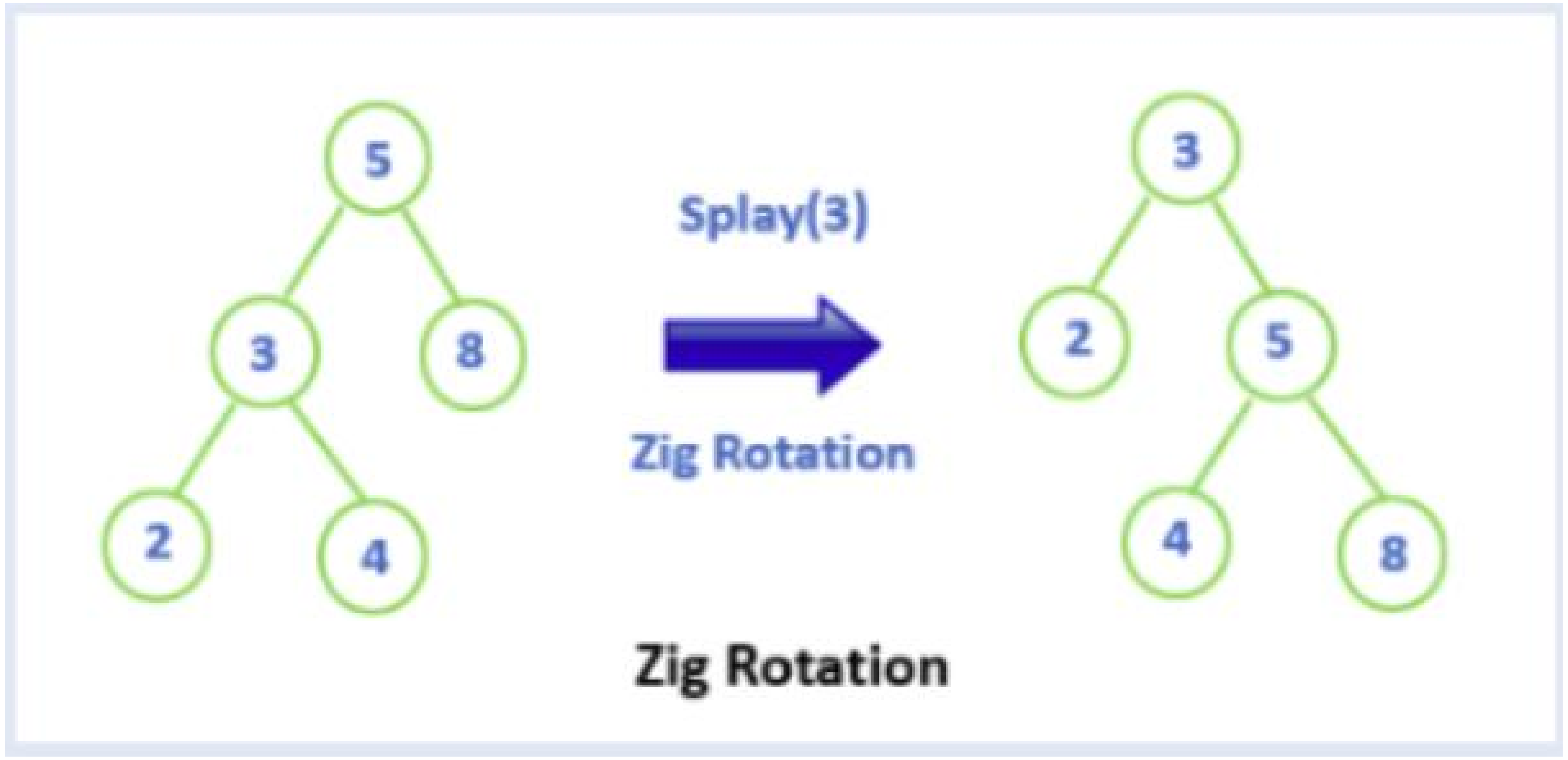
To splay an element to the root, rotations that are similar to that of an AVL tree.

Rotations of splay tree

1. Zig rotation
2. Zag rotation
3. Zig-Zig rotation
4. Zag-Zag rotation
5. Zig-Zag rotation
6. Zag-Zig rotation

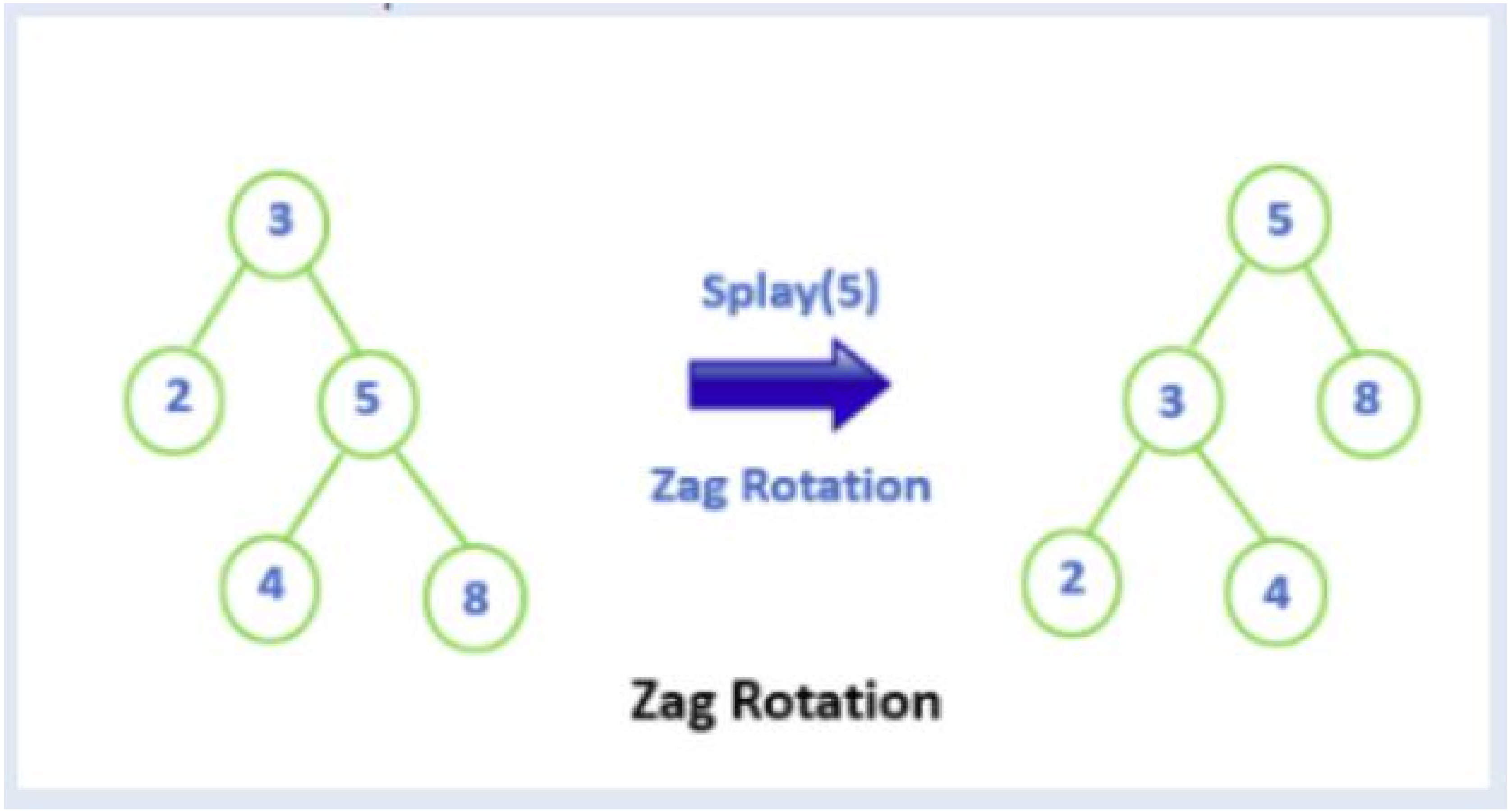
Zig rotation

Zig rotation is performed when we want to splay an element in the immediate left subtree. Zig rotation is similar to a single right rotation in an AVL tree. In a Zig rotation all the elements move one position to the right from its current position.



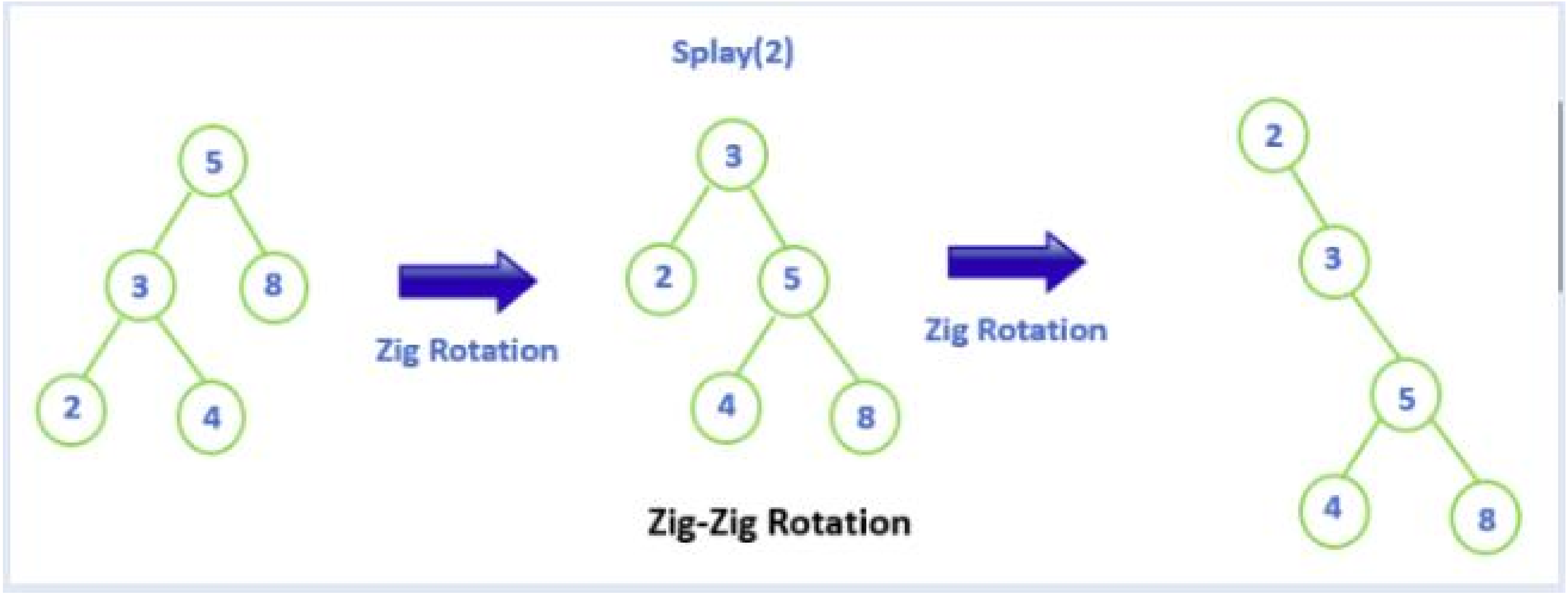
Zag rotation

Zag rotation is performed when we want to splay an element in the immediate right subtree. Zag rotation is similar to a single left rotation in an AVL tree. In a Zag rotation all the elements move one position to the left from its current position.



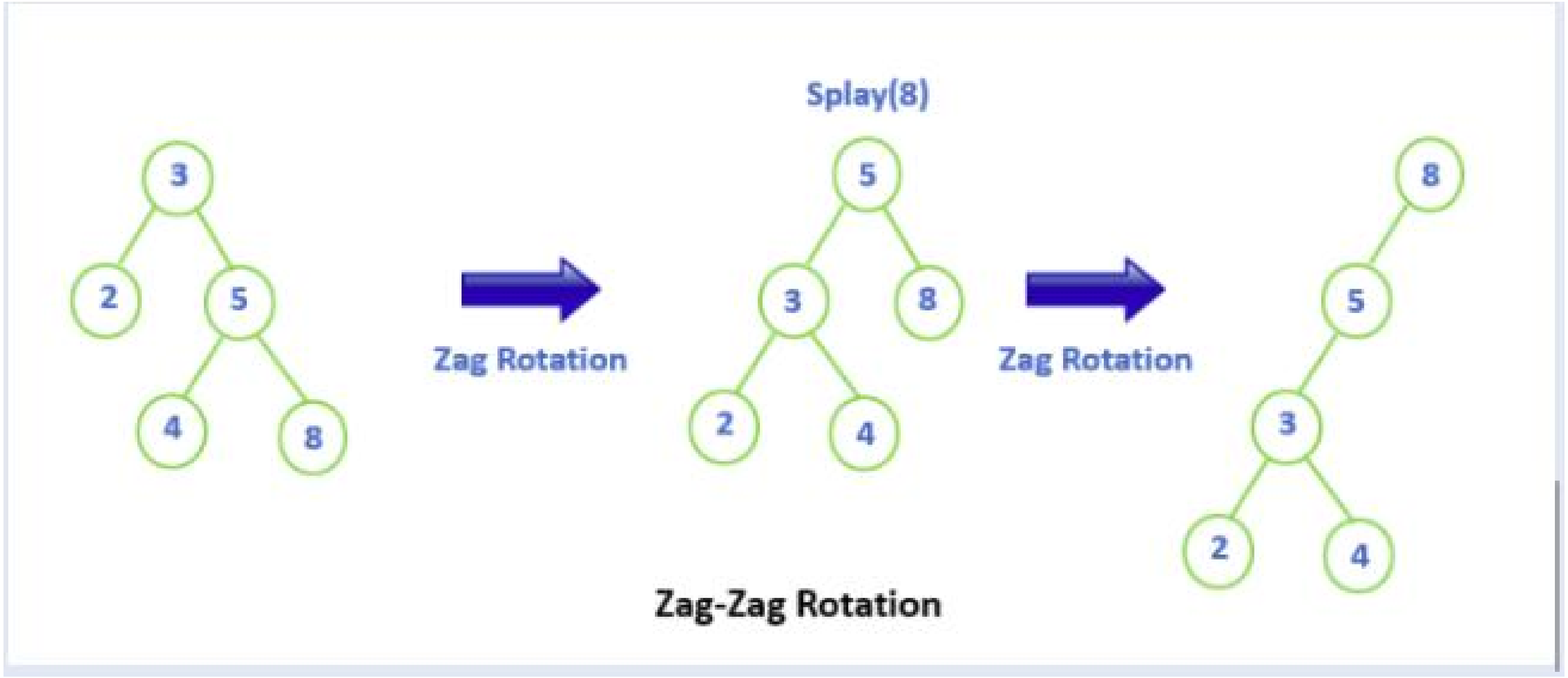
Zig-Zig rotation:

The Zig-Zig rotation in a splay tree is a double zig rotation i.e performing a zig rotation two times. In Zig-Zig rotation every element moves two position to the right from its current position. Zig-Zig rotation is performed when we want to splay an element which is in the left of the left subtree.



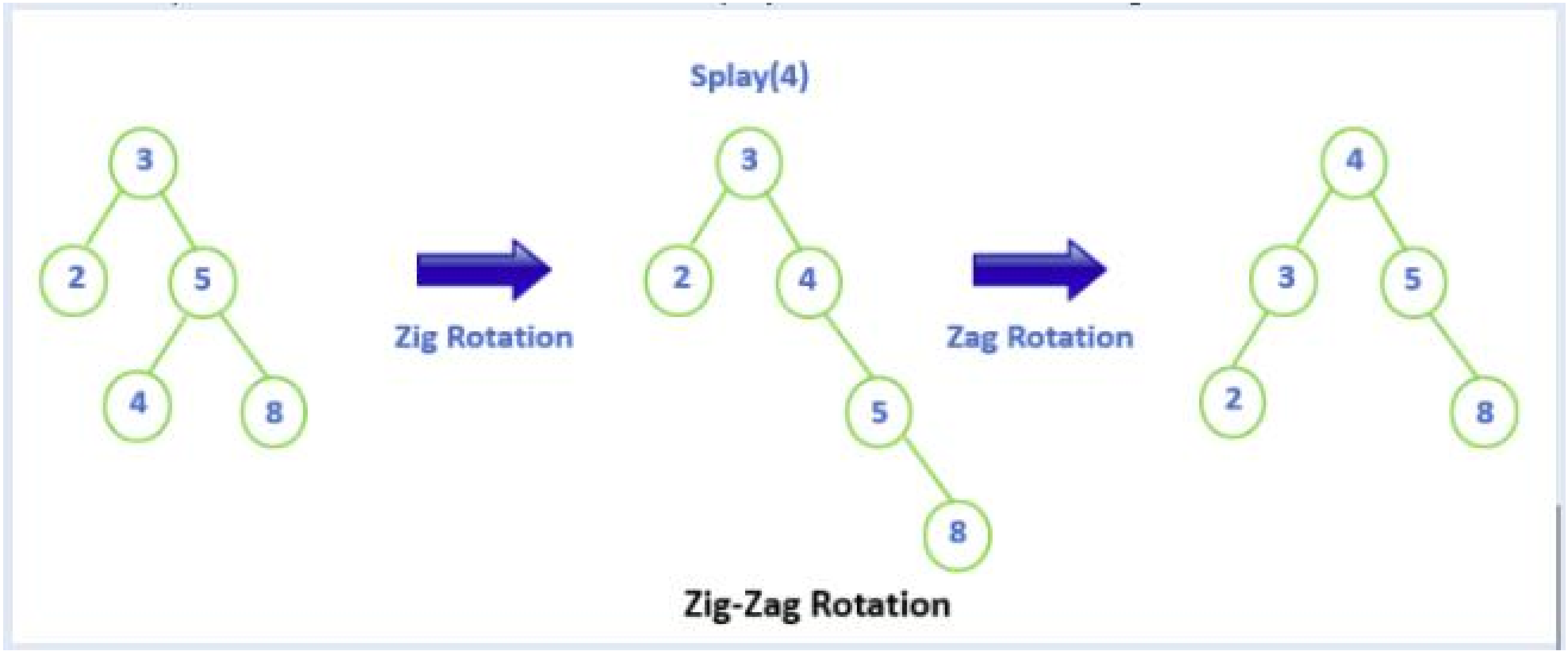
Zag-Zag rotation

The Zag-Zag rotation in a splay tree is a double zag rotation i.e performing zag rotation two times. In Zag-Zag rotation every node moves two position to the left from its current position. Zag-Zag rotation is performed when we want to splay an element that is right of the right subtree.



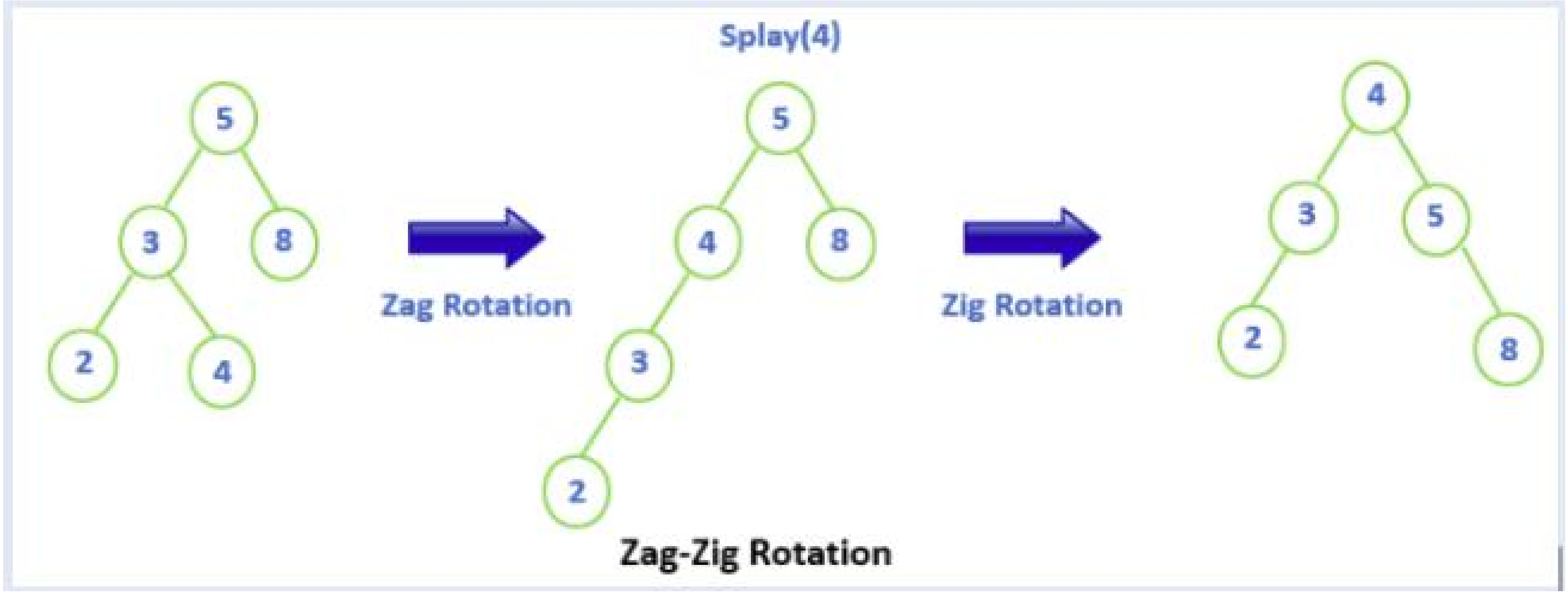
Zig-Zag rotation

In Zig-Zag rotation, a zig rotation is followed by a zag rotation. In Zig-Zag rotation every element moves one position to the right followed by one position to the left from its current position. Zig-Zag rotation is performed when the element to be splayed is in the left of the right subtree.



Zag-Zig rotation

In Zag-Zig rotation, a Zag rotation is followed by a Zig rotation. In Zag-Zig rotation every element moves one position to the left followed by one position to the right from its current position. Zig-Zag rotation is performed when the element to be splayed is in the left of the right subtree.



Insertion Operation in Splay Trees:

BST Insertion: Initially, the new node is inserted into the splay tree just like in a normal binary search tree according to the rules of binary search.

Splay Operation: After insertion, the newly inserted node is splayed to the root of the tree. Splaying brings the newly inserted node to the root, promoting it closer to the root for quicker access in future operations.

Deletion Operation in Splay Trees:

BST Deletion: Initially, the node to be deleted is removed from the splay tree just like in a normal binary search tree according to the rules of binary search.

Splay Operation (Optional): Some implementations of splay trees perform a splay operation on the parent of the deleted node after deletion. This operation brings the parent node closer to the root, potentially improving the tree's future access pattern.

Insertion:

Suppose we have an empty splay tree. We insert elements 5, 3, 8, 2, 4, and 7 into the tree. After each insertion, the newly inserted node is splayed to the root.

- Insert 5: Tree: 5
- Insert 3: Tree: 3 (splay 3)
- Insert 8: Tree: 8 (splay 8)
- Insert 2: Tree: 2 (splay 2)
- Insert 4: Tree: 4 (splay 4)
- Insert 7: Tree: 7 (splay 7)

Deletion:

Suppose we delete node 2 from the splay tree. After deletion, the parent of node 2 is splayed to the root.

Delete 2: Tree: 4 (splay 4's parent)

Red-Black Tree

Red-Black Tree is a self balancing binary search tree in which all the nodes of the follow 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.

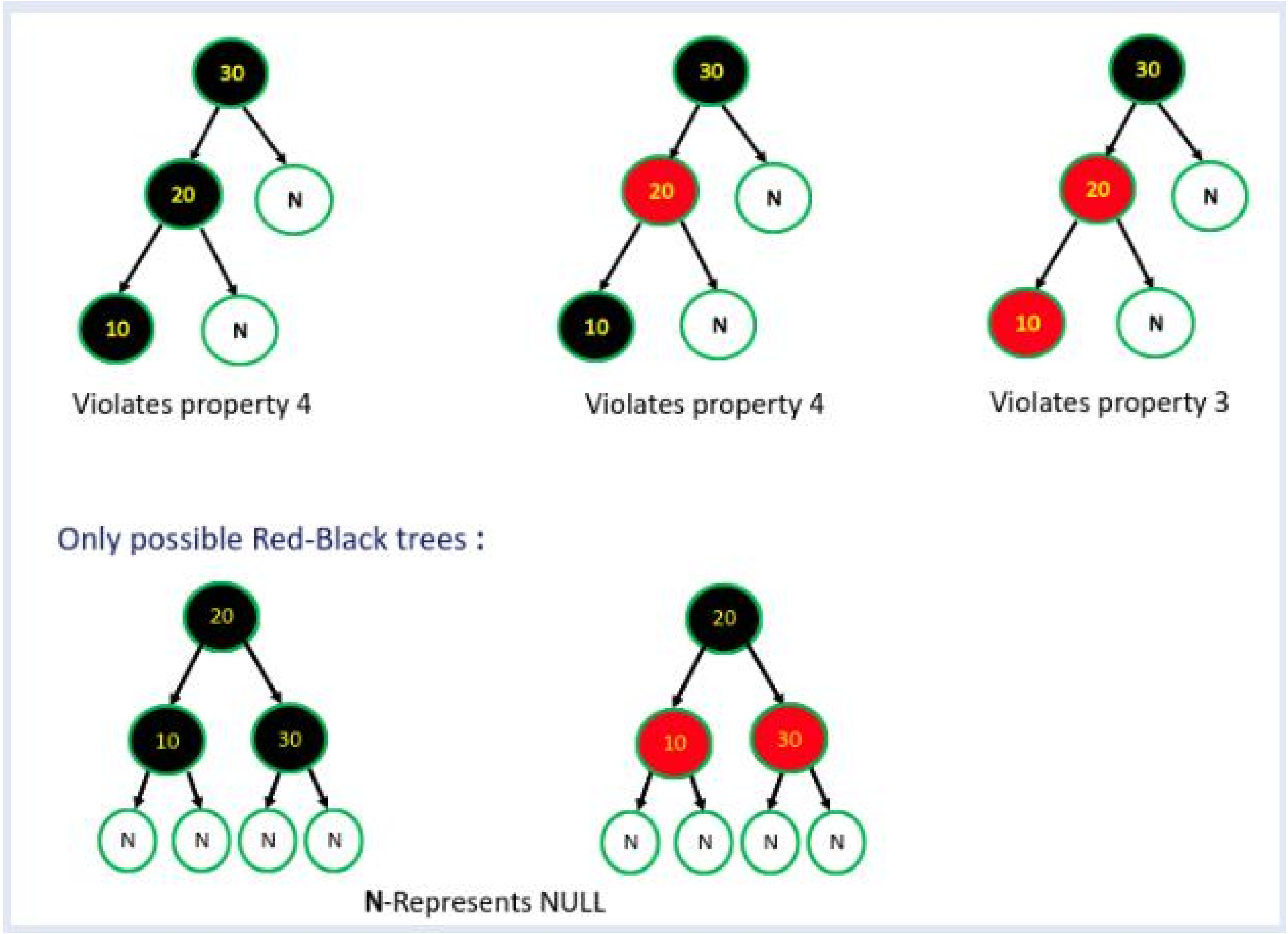
How does Red-Black tree ensure balance

Basic principle behind the balancing in Red-Black trees is that the above four properties mentioned ensure that a chain of 3 nodes is not possible.

Explanation :

Considering 3 nodes 30,20 and 10 in the Binary Search tree, In all the possible combinations of color [Red/Black as per property 1] given to non root nodes [Root node is always black as per property 2] only balanced trees are the ones that satisfy both property 3 and 4.

A visual representation is given below.



Red-Black trees Vs. AVL trees

The AVL trees are more balanced trees compared to Red-Black trees.

Number of rotations during insertion and deletion are more in the cases of AVL trees.

So in a place that involves many frequent insertions and deletions Red-Black trees should be preferred and in a place where the insertions and deletions are less frequent and search is a more frequent operation, AVL tree should be preferred over Red-Black tree.

Insertion Operation in Red-Black Trees:

BST Insertion: Initially, the new node is inserted into the red-black tree just like in a normal binary search tree according to the rules of binary search.

Coloring: The color of the newly inserted node is set to red.

Fix-Up: After insertion, the tree may violate one or more red-black properties. To restore these properties, a series of rotations and color changes (fix-up operations) are performed starting from the newly inserted node and moving upwards towards the root.

- If the parent of the newly inserted node is black, then no further action is needed.
- If the parent of the newly inserted node is red, there are several cases to consider:

Case 1: The uncle of the newly inserted node is also red (the parent's sibling). In this case, recolor the parent, uncle, and grandparent nodes appropriately.

Case 2: The uncle of the newly inserted node is black and the newly inserted node is an inside child (right child of left child, or left child of right child). Perform a rotation and recoloring to transform the situation into Case 3.
Case 3: The uncle of the newly inserted node is black and the newly inserted node is an outside child (left child of left child, or right child of right child). Perform a rotation and recoloring to maintain the red-black properties.

Deletion Operation in Red-Black Trees:

BST Deletion: Initially, the node to be deleted is removed from the red-black tree just like in a normal binary search tree according to the rules of binary search.

Fix-Up: After deletion, the tree may violate one or more red-black properties. To restore these properties, a series of rotations and color changes (fix-up operations) are performed starting from the sibling of the deleted node and moving upwards towards the root.

- If the sibling of the deleted node is red, perform rotations and recoloring to adjust the tree structure.
- If the sibling of the deleted node is black, there are several cases to consider:

Case 1: Both children of the sibling are black. Recolor the sibling to red and move the violation up to the parent.
Case 2: The sibling's outside child (farthest from the deleted node) is red. Perform rotations and recoloring to transform the situation into Case 3.
Case 3: The sibling's outside child is black and the sibling's inside child is red. Perform rotations and recoloring to maintain the red-black properties.

Rotations in Red-Black Trees:

Left Rotation: Rotate the subtree rooted at a given node to the left.
Right Rotation: Rotate the subtree rooted at a given node to the right.

- By performing these operations and rotations, red-black trees maintain balance and ensure that the height of the tree remains logarithmic, providing efficient performance for all operations.

B-Trees

B-trees and B+ trees are balanced tree data structures commonly used for organizing and storing large amounts of data efficiently on disk or in memory. They are particularly well-suited for databases and file systems due to their ability to minimize the number of disk accesses required for operations. Here are the basic operations of B-trees and B+ trees:

Basic Operations of B-Trees:

Search: To search for a key in a B-tree, start at the root and recursively search down the tree. At each node, perform a binary search to find the appropriate child node to continue the search.

Insertion: To insert a new key into a B-tree, start by searching for the appropriate leaf node where the key should be inserted. If the leaf node is full, split it into two nodes and promote the median key to the parent node. Continue this process recursively until no node overflows.

Deletion: To delete a key from a B-tree, first search for the key in the tree. If the key is found in a leaf node, remove it. If the leaf node becomes underflowed after deletion, perform various restructuring operations to balance the tree.

Traversal: In-order traversal of a B-tree will result in keys being visited in sorted order.

Basic Operations of B+ Trees:

Search: Searching in a B+ tree is similar to searching in a B-tree. Start at the root and recursively search down the tree until the appropriate leaf node is found. Keys are only stored in leaf nodes in B+ trees.

Insertion: Insertion in a B+ tree is similar to insertion in a B-tree. Start by searching for the appropriate leaf node to insert the new key. If the leaf node is full, split it into two nodes and update the parent node accordingly. If necessary, propagate the split operation recursively until the root is reached.

Deletion: Deletion in a B+ tree is similar to deletion in a B-tree. Search for the key to be deleted, remove it from the leaf node, and perform restructuring operations if necessary to maintain balance.

Range Queries: B+ trees are well-suited for range queries due to their structure. Range queries involve searching for keys within a specific range. By starting at the leaf node containing the smallest key in the range and following the linked list of leaf nodes, all keys within the range can be efficiently retrieved.

Traversal: In a B+ tree, in-order traversal involves traversing the linked list of leaf nodes, resulting in keys being visited in sorted order.

Both B-trees and B+ trees offer efficient search, insertion, and deletion operations, making them suitable for a wide range of applications where efficient storage and retrieval of large datasets are required. B+ trees, in particular, excel in scenarios where range queries are common due to their structure that facilitates efficient range-based retrieval.

Insertion Operation:

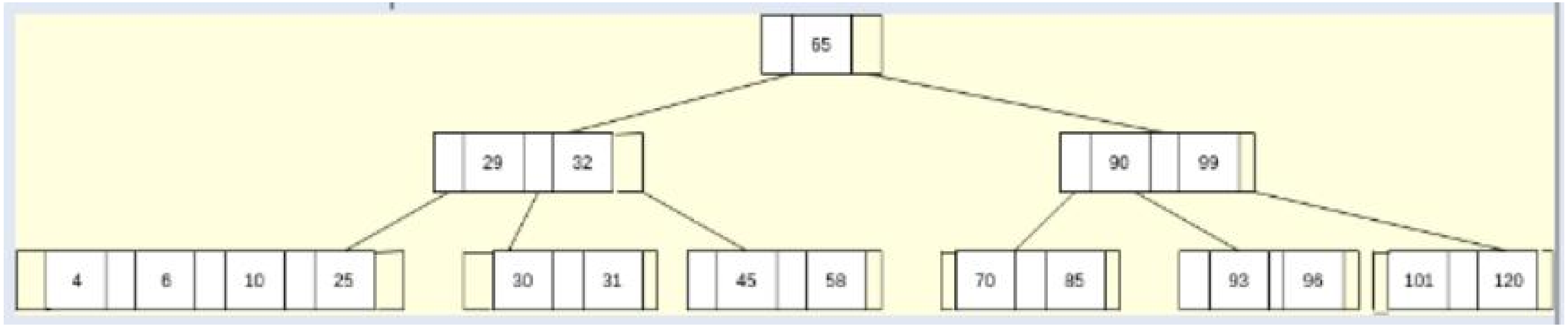
B-Tree Insertion:

1. Start at the root and recursively search down the tree to find the appropriate leaf node where the key should be inserted.
2. If the leaf node is not full, insert the key into the leaf node in sorted order.
3. If the leaf node is full, split it into two nodes and promote the median key to the parent node.
4. Continue this process recursively until reaching a node that is not full, or until reaching the root and splitting it.

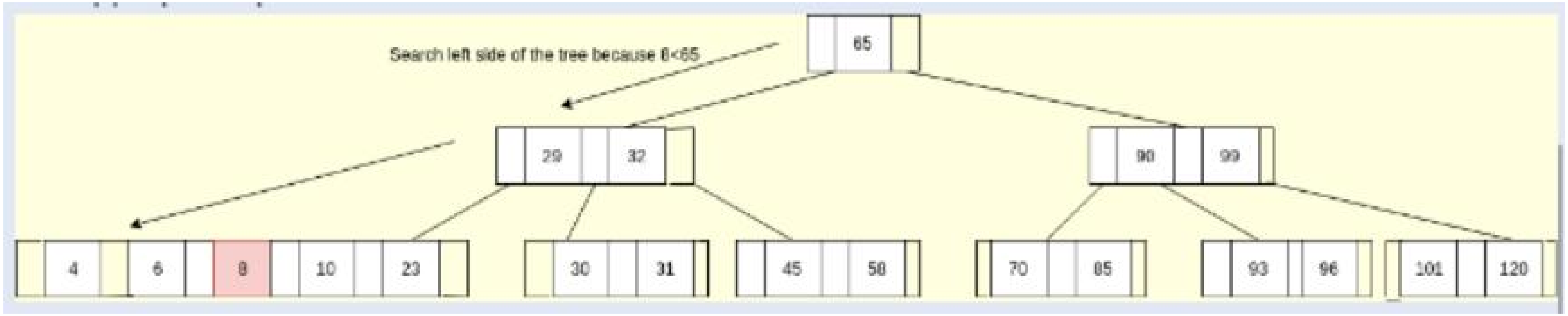
B+ Tree Insertion:

1. Start at the root and recursively search down the tree to find the appropriate leaf node where the key should be inserted.
2. If the leaf node is not full, insert the key into the leaf node in sorted order.
3. If the leaf node is full, split it into two nodes and update the parent node accordingly.
4. If necessary, propagate the split operation recursively until the root is reached.

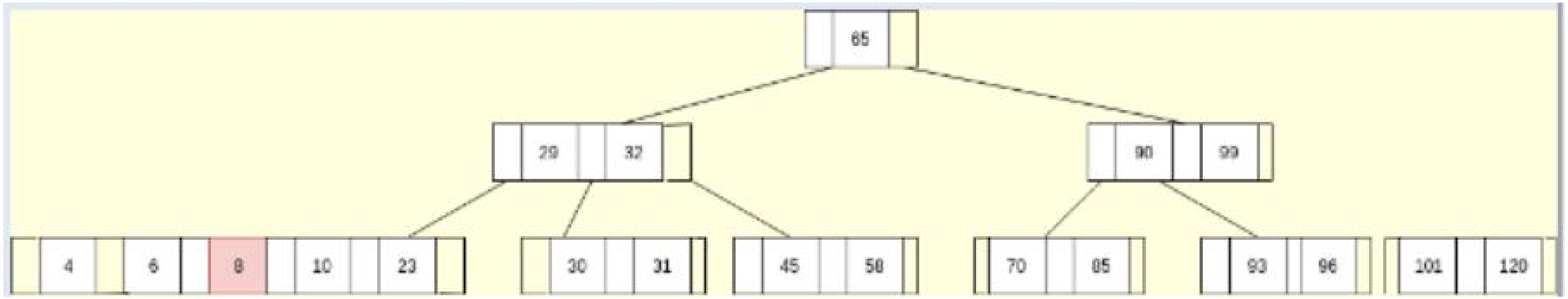
Let us consider an example to understand the insertion in B tree. Below is the B tree of order 5



Now we are inserting element 8 into the B Tree. since 8 < 65 so it traverses towards the left side of the tree. In the second level 8 < 29 again it traverses towards the left and inserts the element 8 into the appropriate position as shown below



But the order of the tree m = 5, it contains m-1 keys i.e 5 - 1 = 4 keys so split the node at the middle position and place the middle element 8 to its parent node as shown



Deletion Operation:

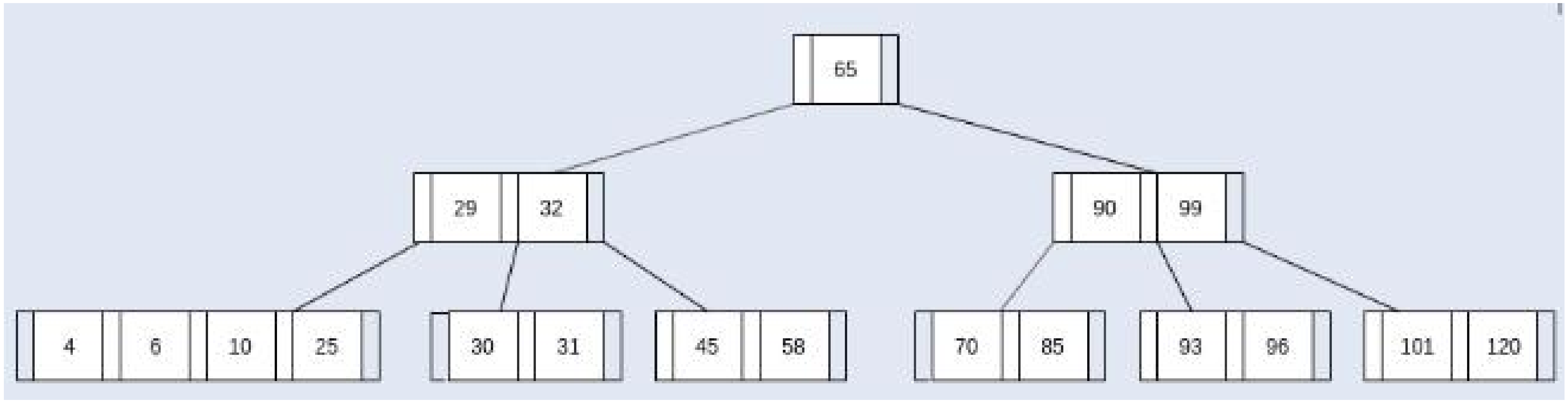
B-Tree Deletion:

1. Search for the key to be deleted in the tree.
2. If the key is found in a leaf node, remove it from the node.
3. If the leaf node becomes underflowed after deletion, borrow a key from a sibling or perform a merge operation to balance the tree.
4. Continue propagating the underflow condition upwards until reaching the root or until the tree is balanced.

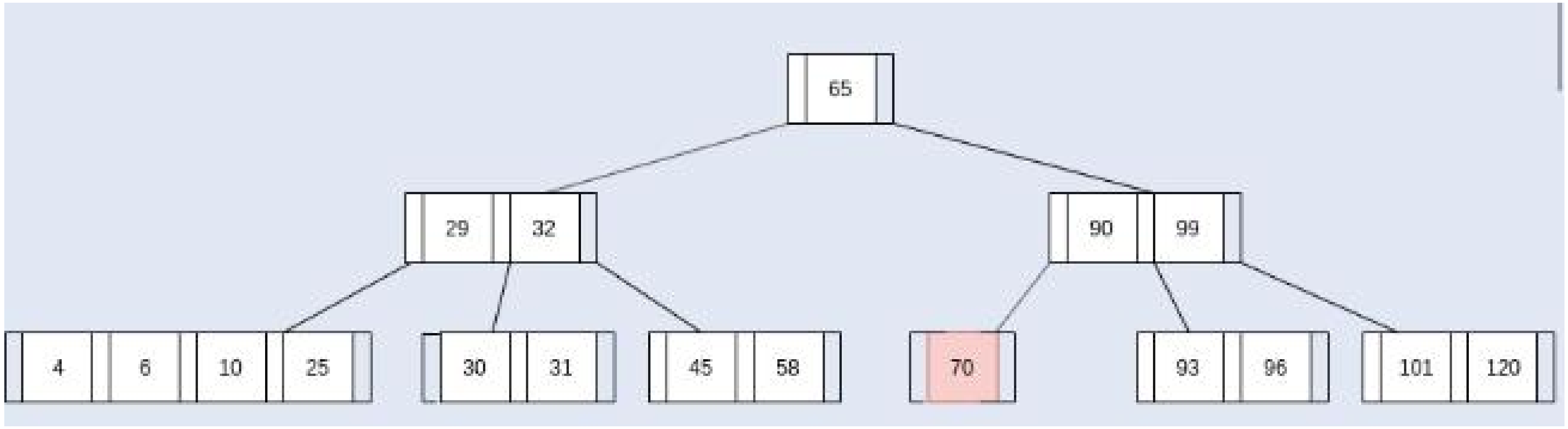
B+ Tree Deletion:

1. Search for the key to be deleted in the tree.
2. If the key is found in a leaf node, remove it from the node.
3. If necessary, perform restructuring operations to maintain balance in the tree.
4. If a leaf node becomes underflowed after deletion, borrow a key from a sibling or merge with a sibling and update the parent accordingly.
5. Propagate the underflow condition recursively until the root is reached or until the tree is balanced.

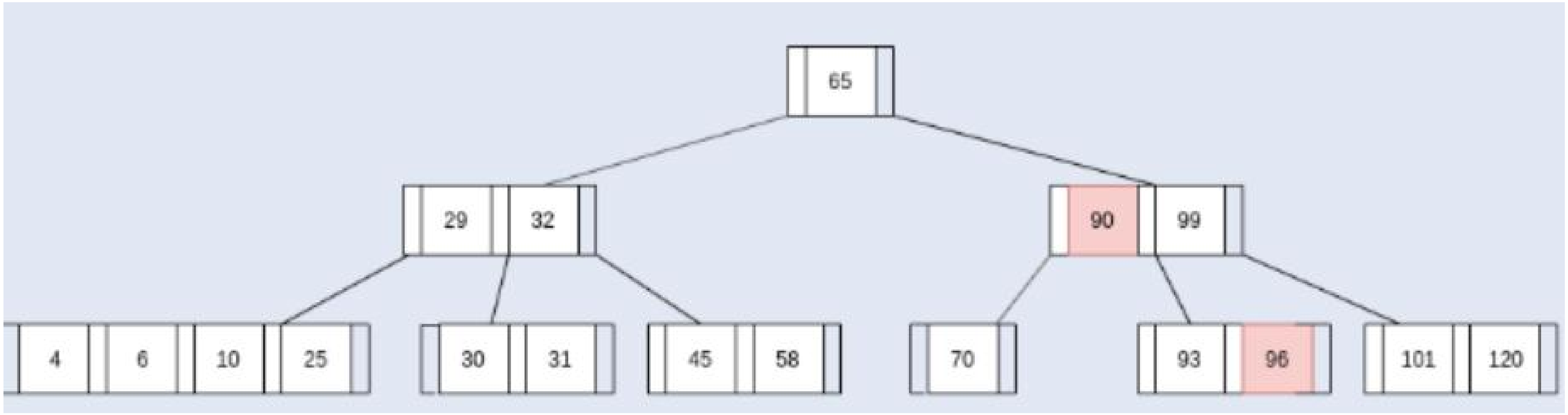
In the below example, we can learn how to delete an element from the B-Tree



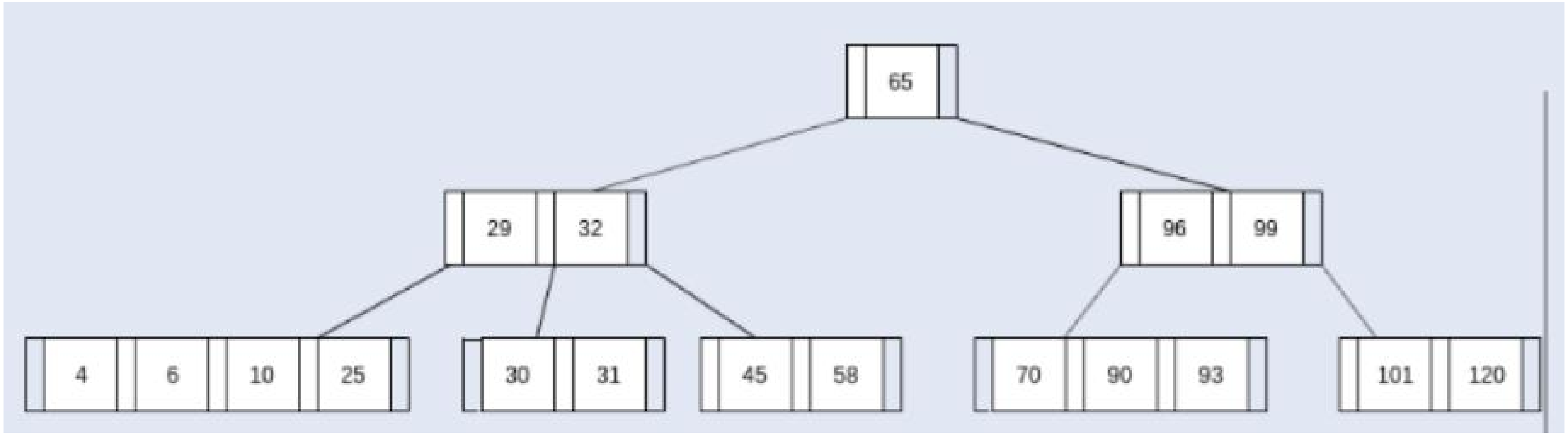
From the above example, assume the element 85 is to be deleted. The resultant B-Tree after the deletion of 85.



From the above diagram after deleting 85, we can observe the B-Tree property is violated at element 70. As the node should contain at least 2 elements. The element 70 will be combined with the sibling node.



The property of B-Tree is, the left node should be lesser than the parent node and the right node should be greater than the parent node. The greatest element 96 in the right node will be pushed to the parent node, and move the intervening element 90 to the child node. The resultant tree will be as shown below:



Notes:

- In both B-trees and B+ trees, splitting and merging operations are key to maintaining balance.
- B+ trees often require less restructuring after insertions and deletions compared to B-trees, as only leaf nodes contain keys in B+ trees.
- Both B-trees and B+ trees maintain the properties of a balanced tree, ensuring efficient search, insertion, and deletion operations.
- These operations ensure that B-trees and B+ trees maintain their properties, such as minimum degree and balanced heights, resulting in efficient storage and retrieval of data.