TREE BASED DSA(II)

B TREE

- INSERTION IN A B-TREE
- DELETION FROM A B-TREE

B+ TREE

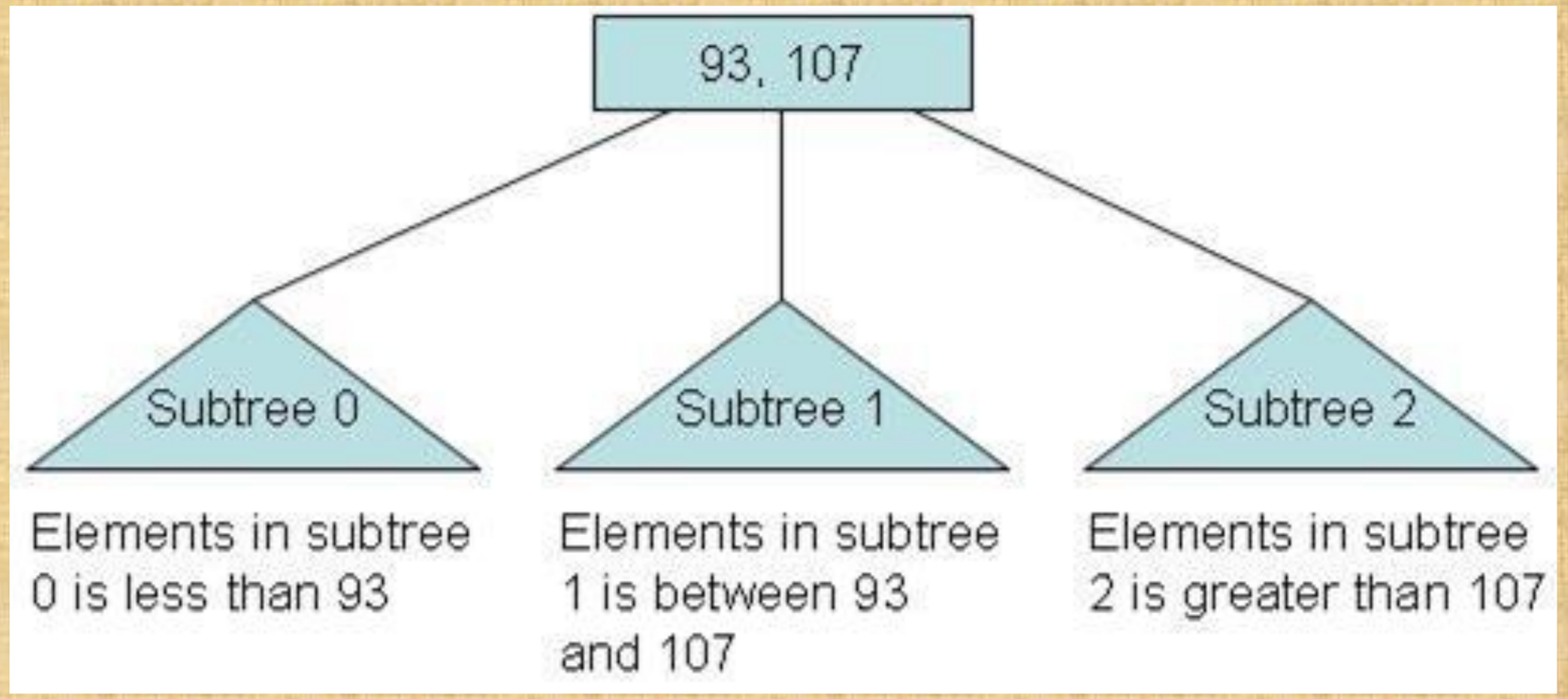
- INSERTION IN A B+ TREE
- DELETION FROM A B+ TREE

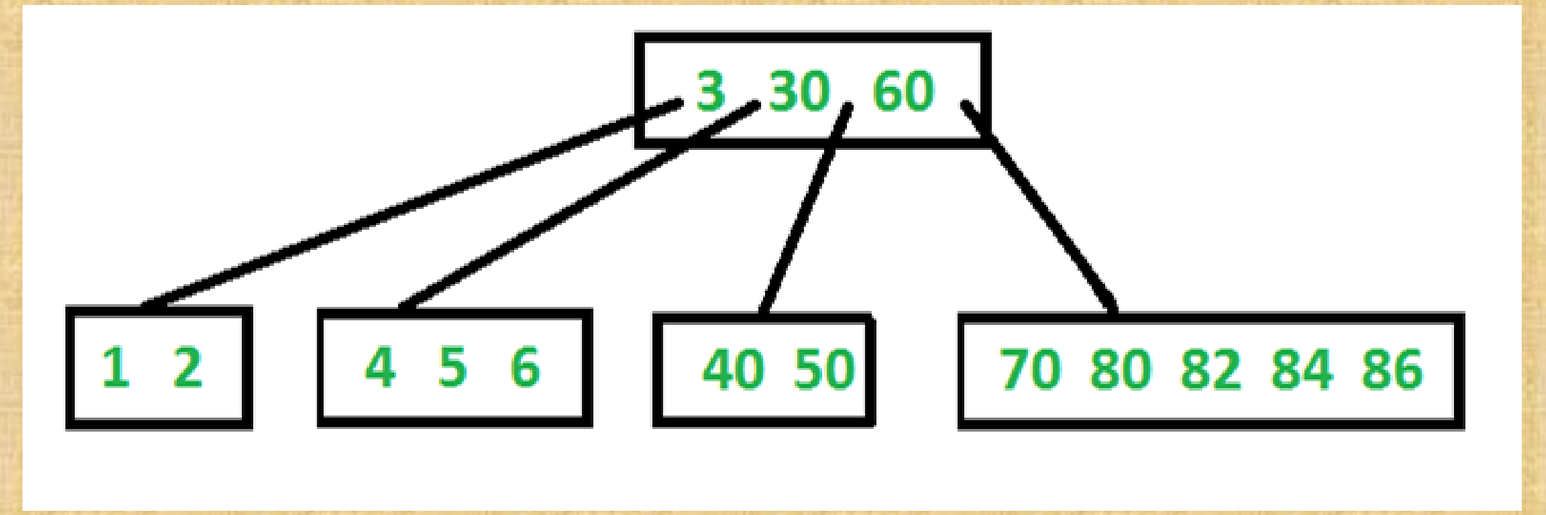
RED-BLACK TREE

- RED-BLACK TREE INSERTION
- RED-BLACK TREE DELETION

B-Tree

- B-tree - is a self-balancing tree data structure in which a node can have more than two children.
- Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order.
- Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceding key.
- A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.


93, 107
Subtree 0
Subtree 1
Subtree 2
Elements in subtree 0 is less than 93
Elements in subtree 1 is between 93 and 107
Elements in subtree 2 is greater than 107


3 30 60
1 2
4 5 6
40 50
70 80 82 84 86

B-Tree of Order m has the following properties..

- **Property #1** - All **leaf nodes** must be at **same level**.
- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **Property #3 -** All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order.**

Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

Starting from the root node, compare k with the first key of the node.

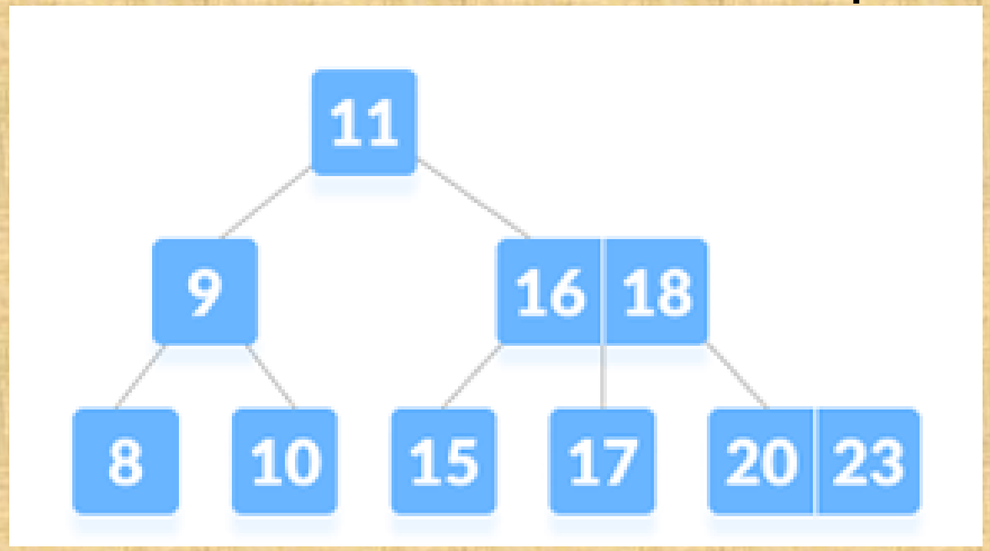
1. If k = the first key of the node, return the node and the index.
2. If k.leaf = true, return NULL (i.e. not found).
3. If k < the first key of the root node, search the left child of this key recursively.
4. If there is more than one key in the current node and k > the first key, compare k with the next key in the node.

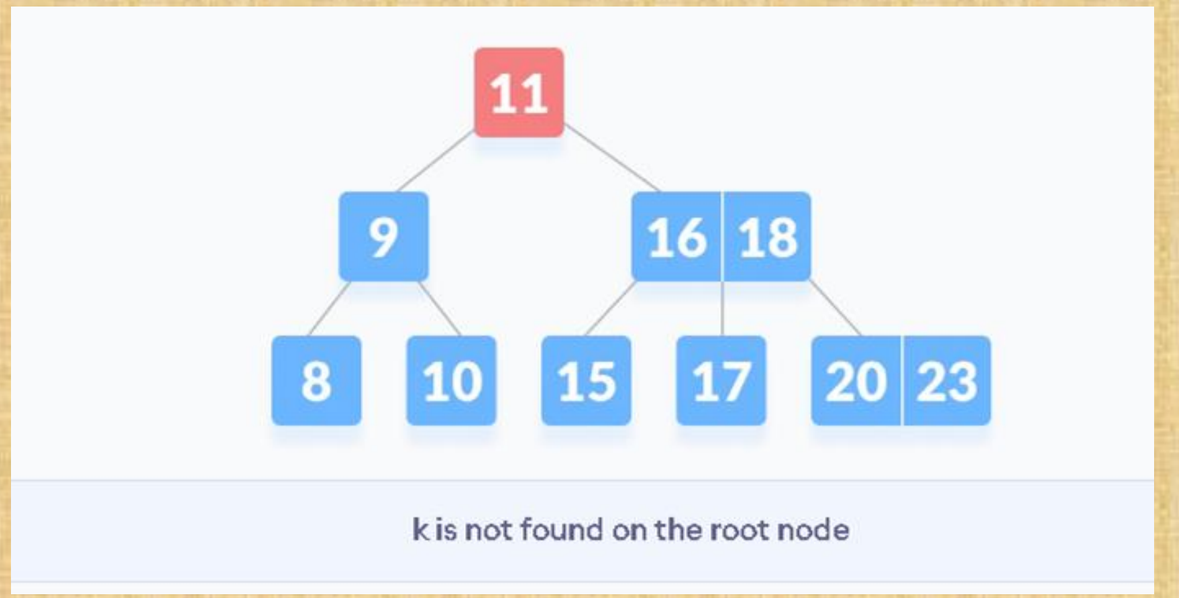
   If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).

   Else, search the right child of the key.
5. Repeat steps 1 to 4 until the leaf is reached.

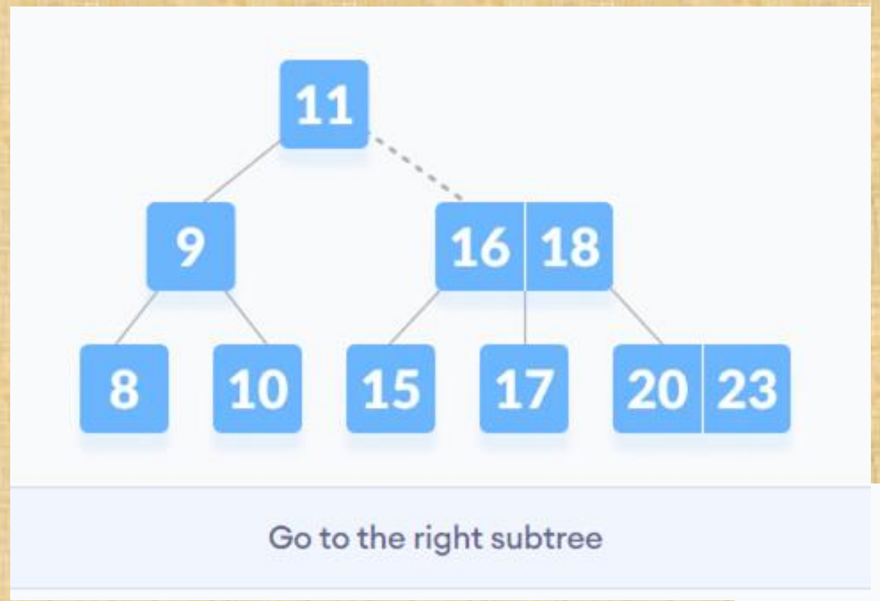
Searching Example

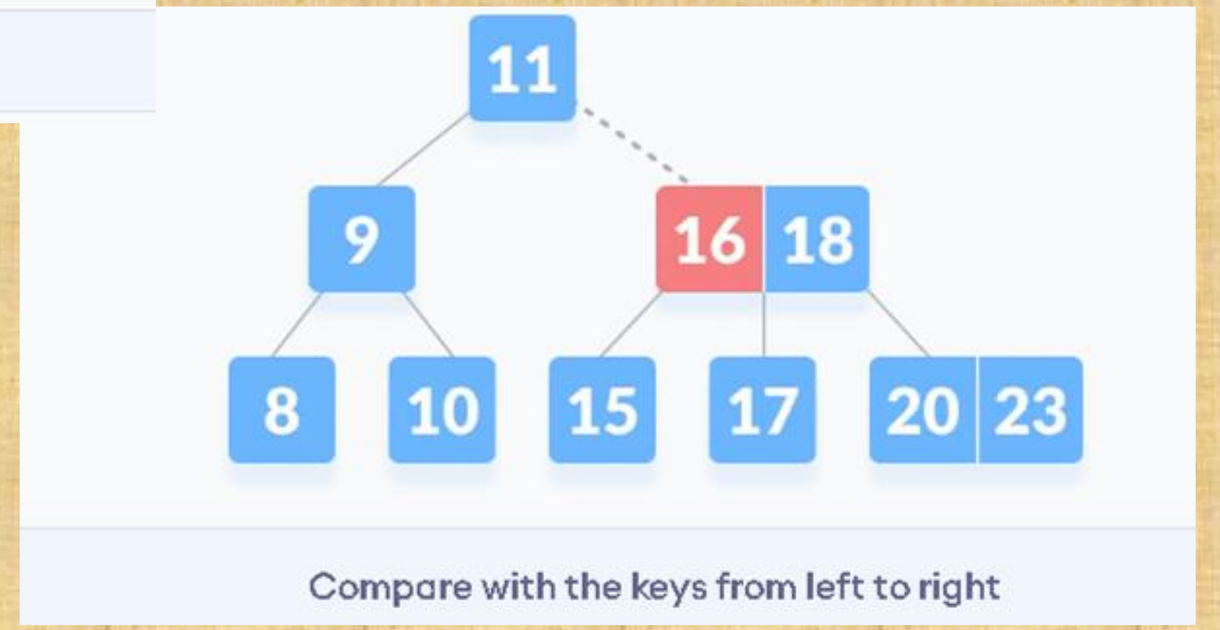
- Let us search key k = 17 in the tree below of degree 3.
- k is not found in the root so, compare it with the root key.



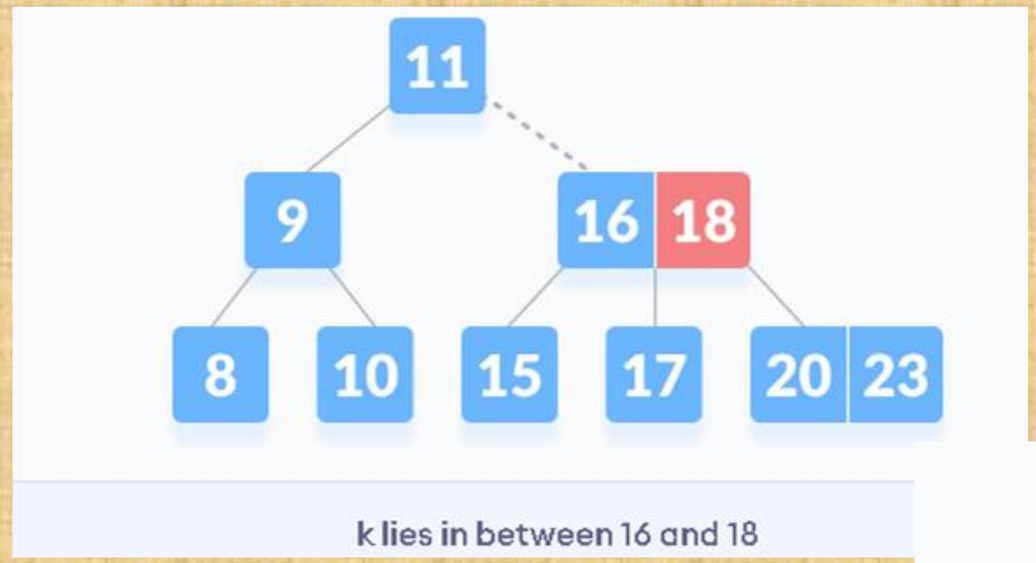


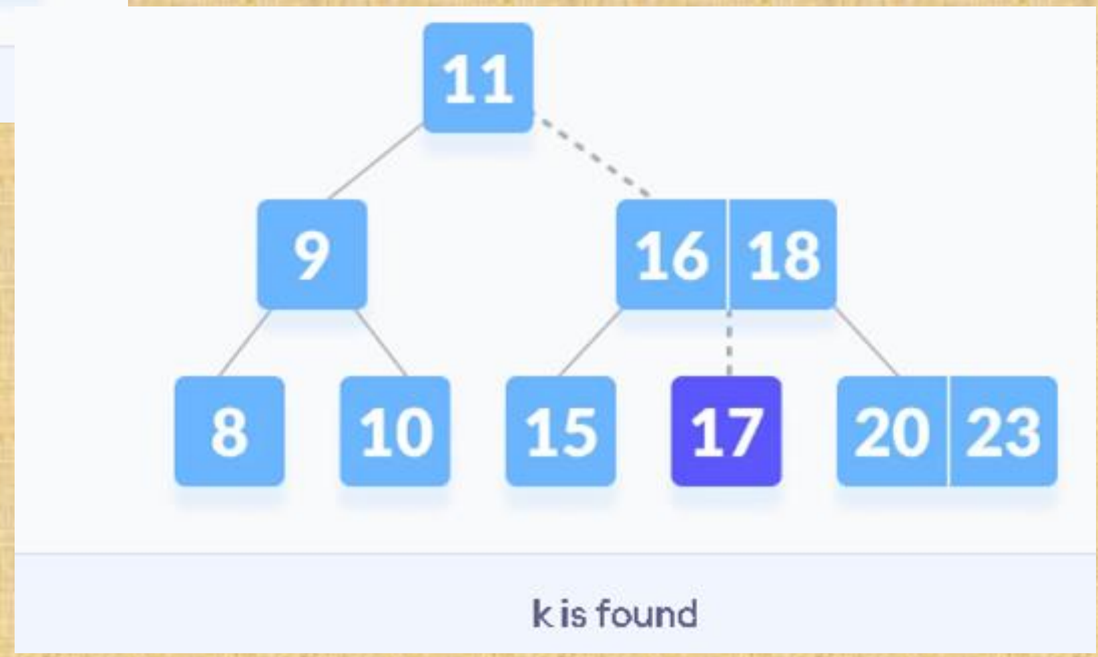
- Since k > 11, go to the right child of the root node.
- Compare k with 16. Since k > 16, compare k with the next key 18.





5. Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.
6. k is found





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

Step 1 - Check whether tree is Empty.

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

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

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

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

Step 6 - If the splitting is performed at root node then the middle value becomes

Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

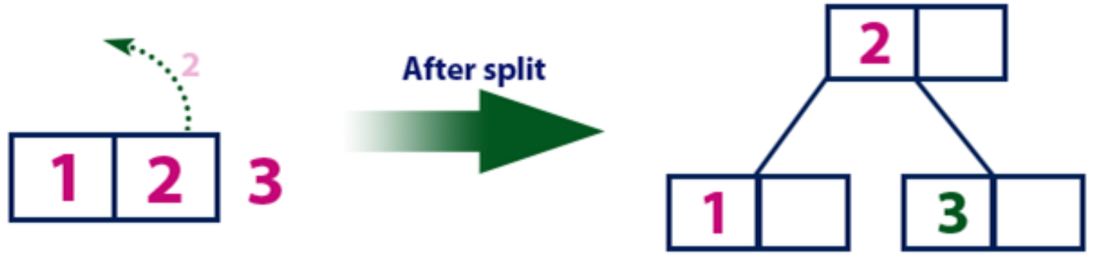
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

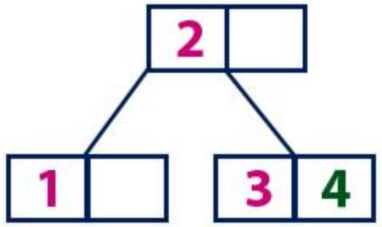
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

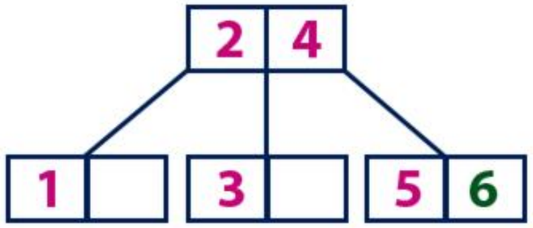


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

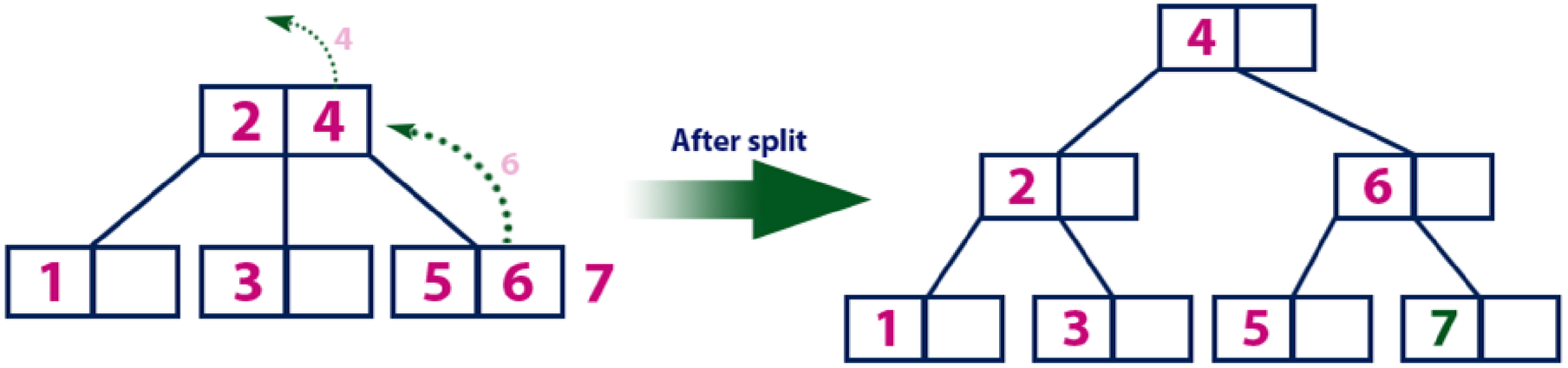
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



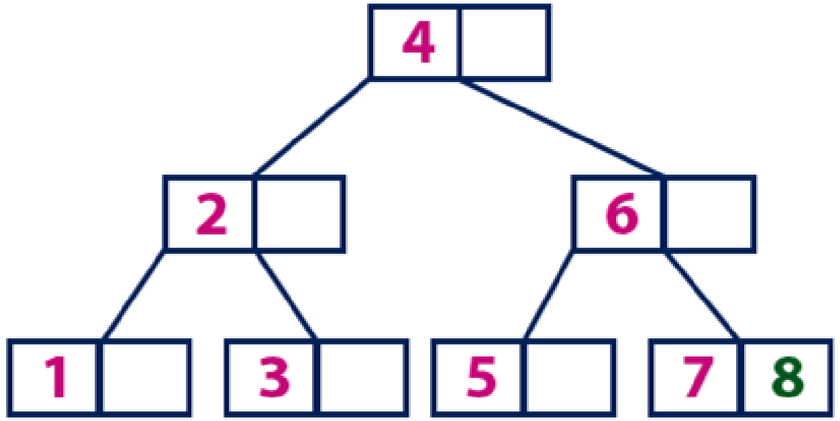
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



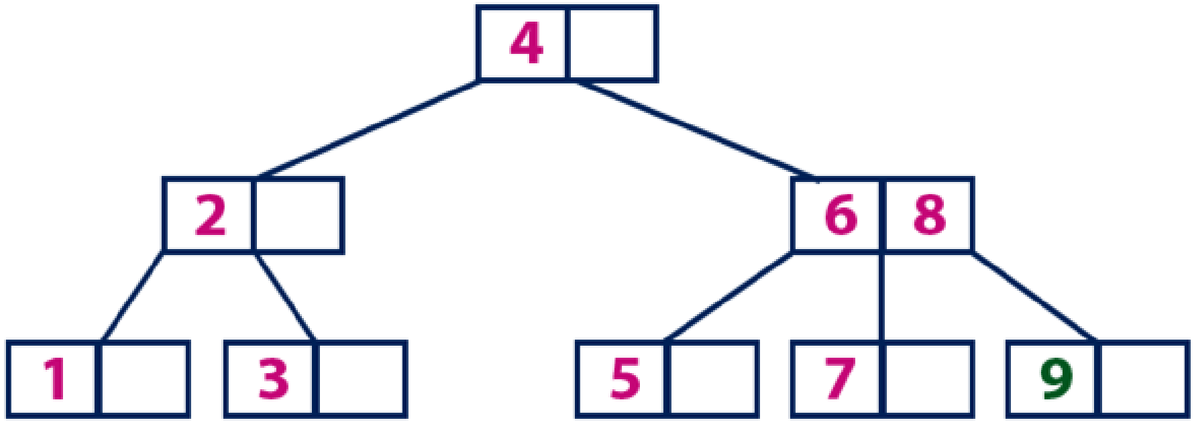
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



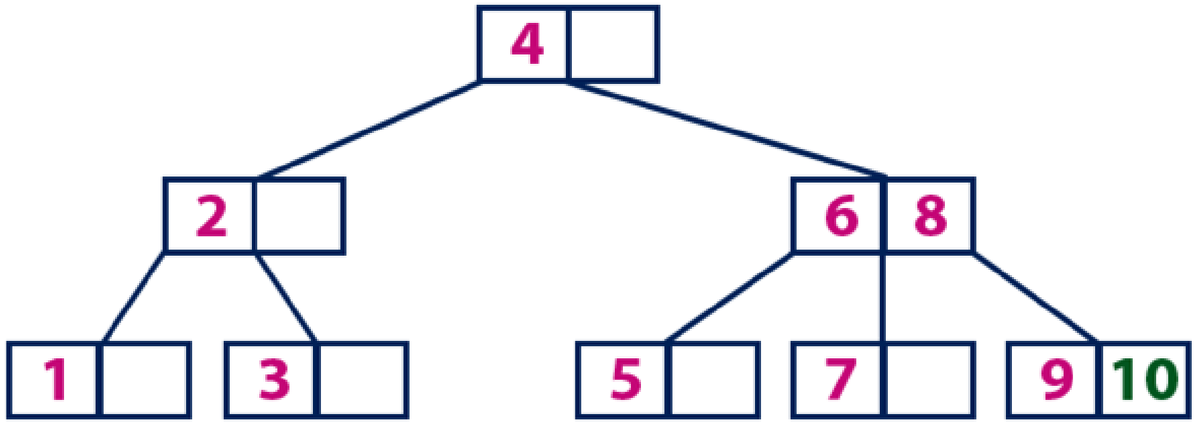
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



DELETION FROM A B-TREE

- Deleting an element on a B-tree consists of three main events: searching the node where the key to be deleted exists, deleting the key and balancing the tree if required.

While deleting a tree, a condition called underflow may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.

The terms to be understood before studying deletion operation are:

Inorder Predecessor

- The largest key on the left child of a node is called its inorder predecessor.

Inorder Successor

- The smallest key on the right child of a node is called its

Deletion Operation

01

A node can have a maximum of m children. (i.e. 3)

02

A node can contain a maximum of m - 1 keys. (i.e. 2)

03

A node should have a minimum of [m/2] children. (i.e. 2)

04

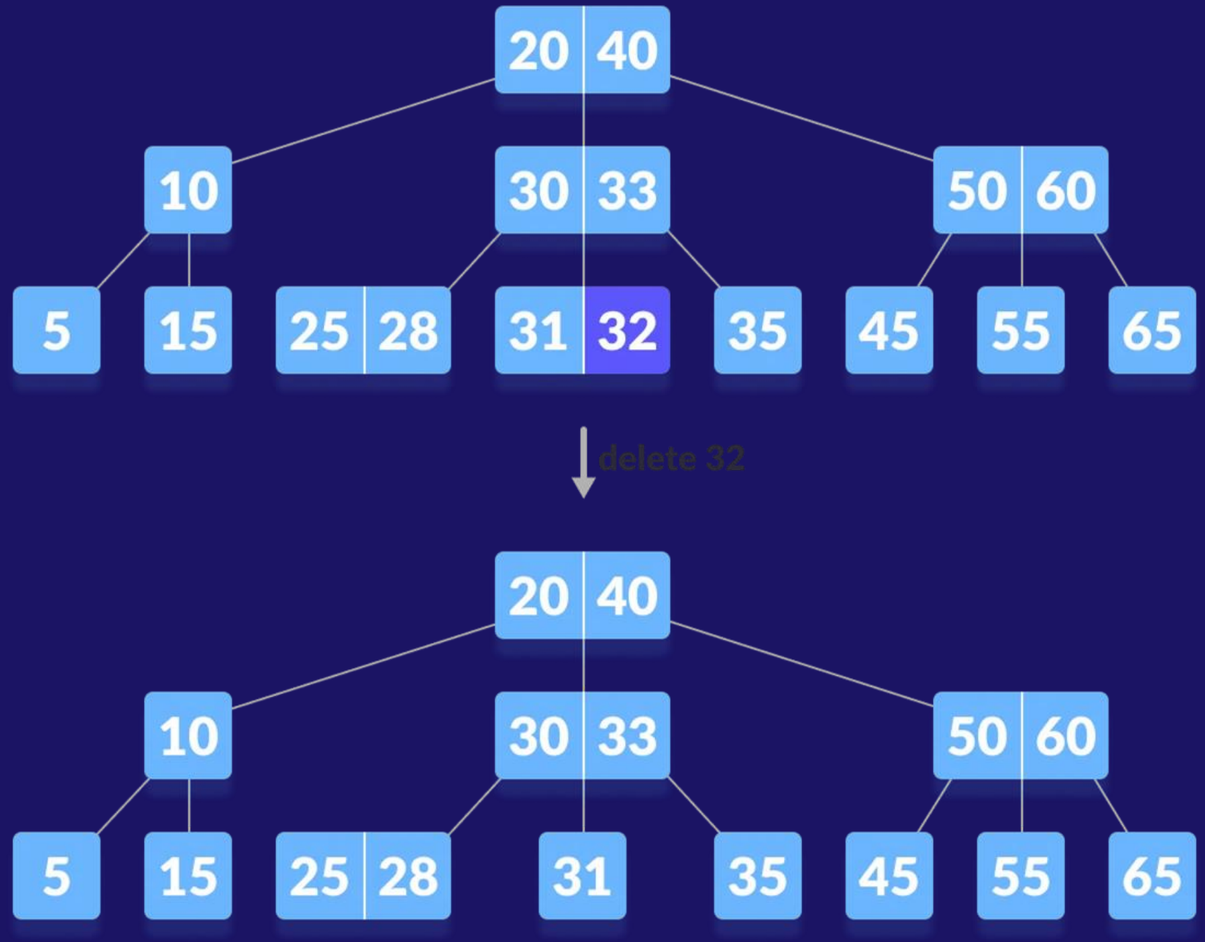
A node (except root node) should contain a minimum of [m/2] - 1 keys. (i.e. 1)

Case I

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

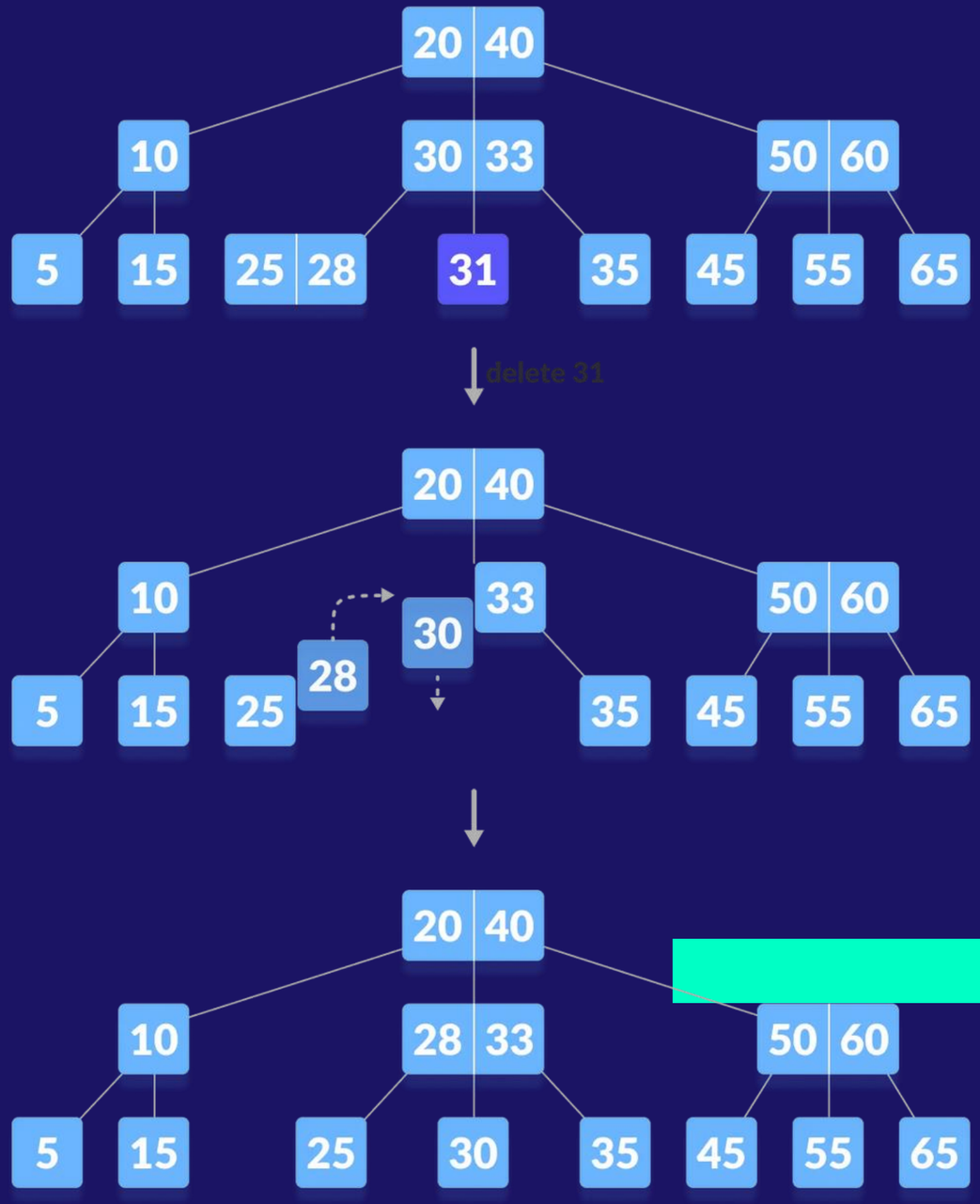
1. The deletion of the key does not violate the property of the minimum number of keys a node should hold.

In the tree example, deleting 32 does not violate the above properties



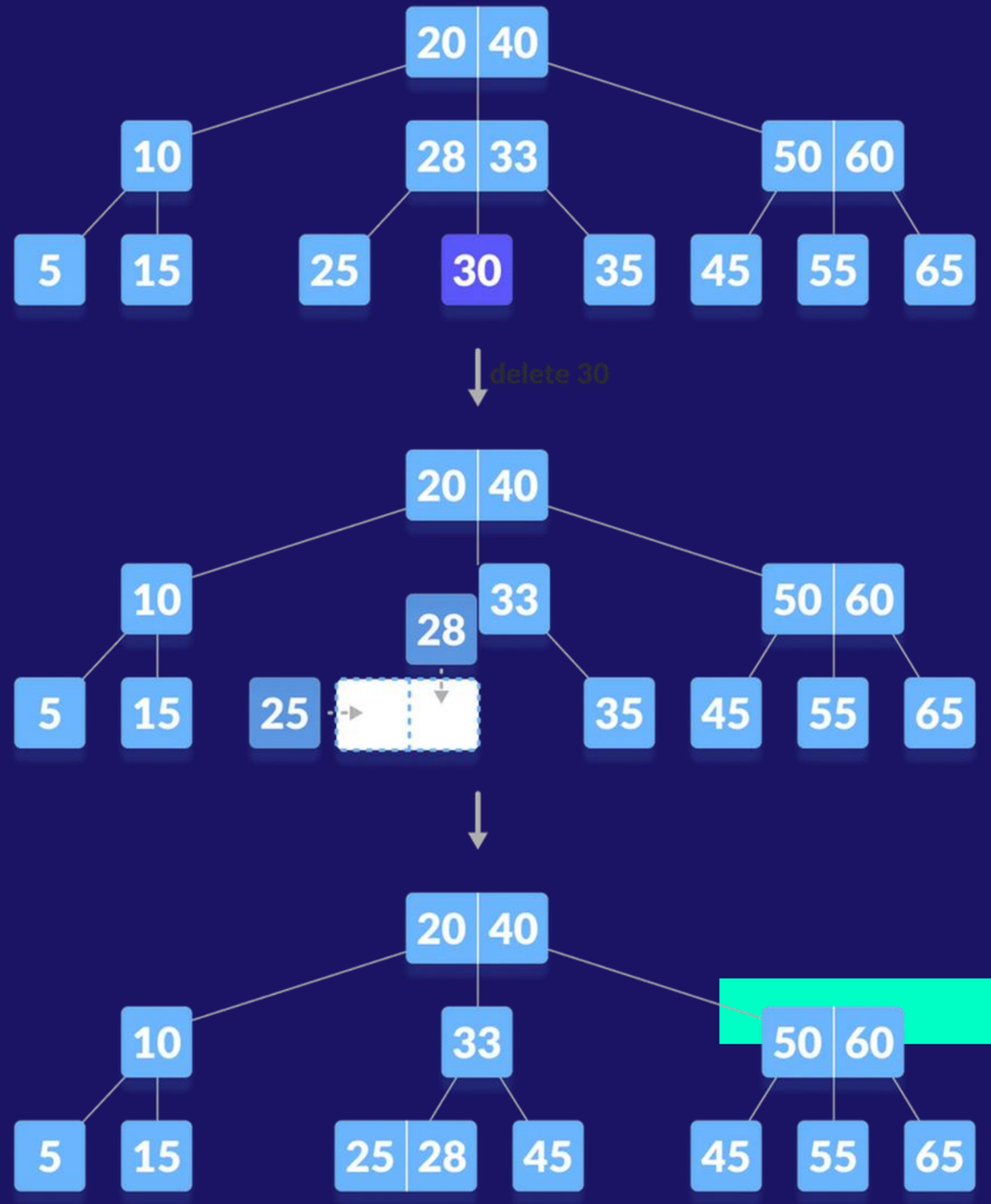
Case I

2. The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right. First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node. Else, check to borrow from the immediate right sibling node. In the tree below, deleting 31 results in the above



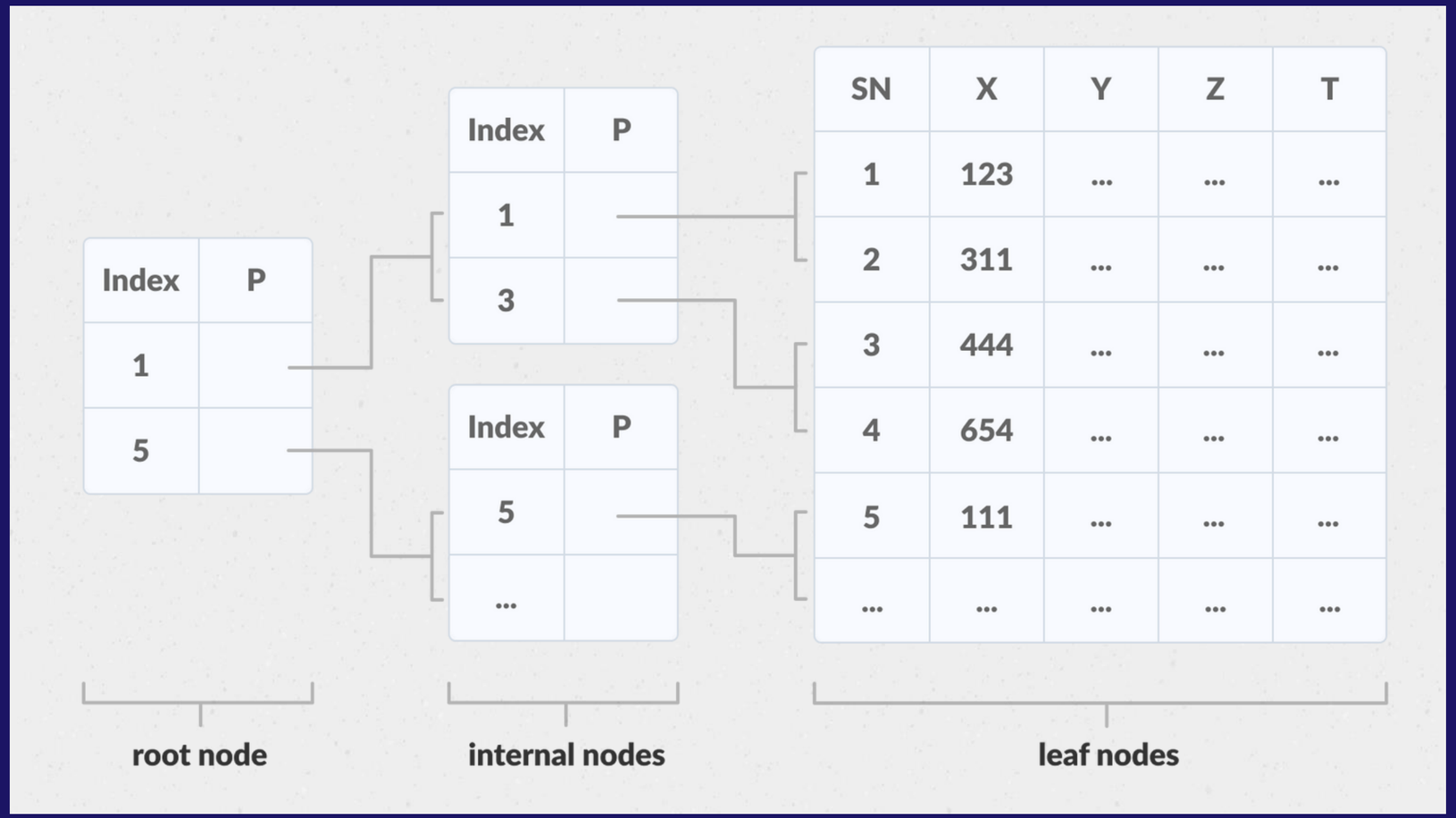
Case I

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. Deleting 30 results in the above case.



B+ TREE

- A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.



An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure above. It makes accessing the data easier and faster.

Properties of a B+ Tree

01

All leaves are at the same level.

02

The root has at least two children.

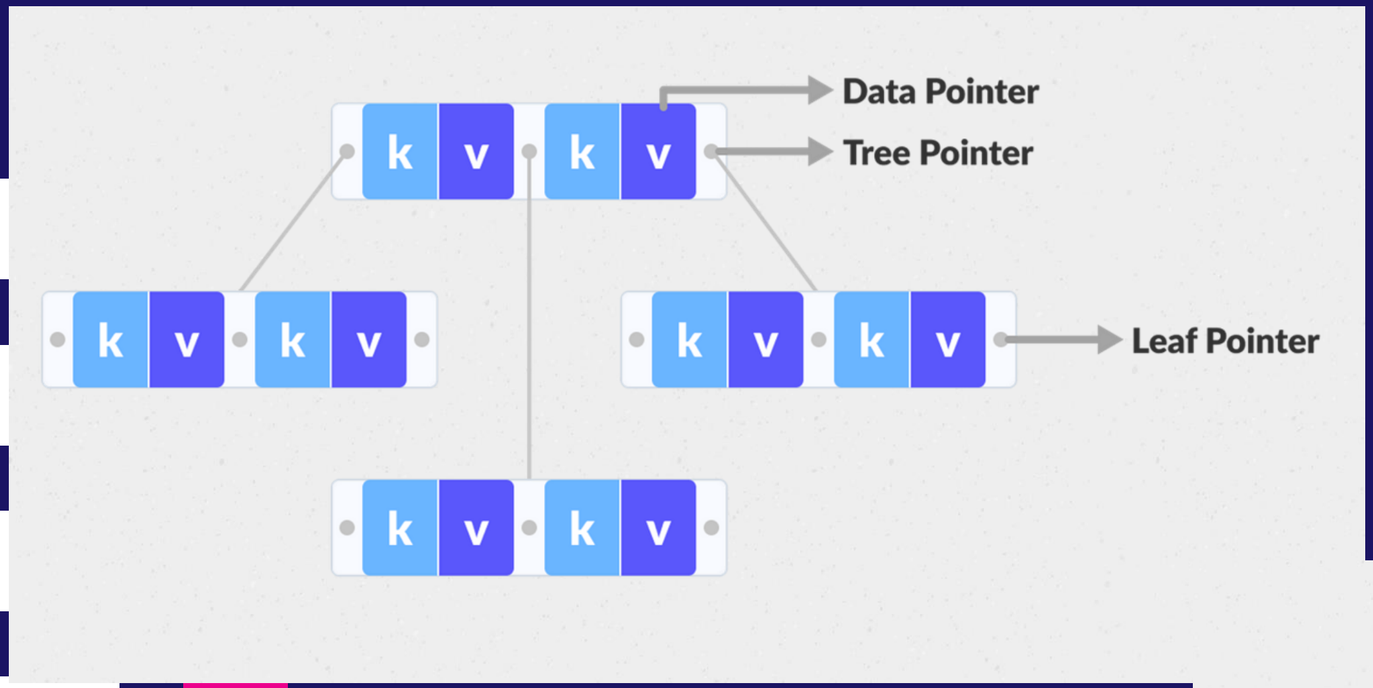
03

Each node except root can have a maximum of m children and at least m/2 children.

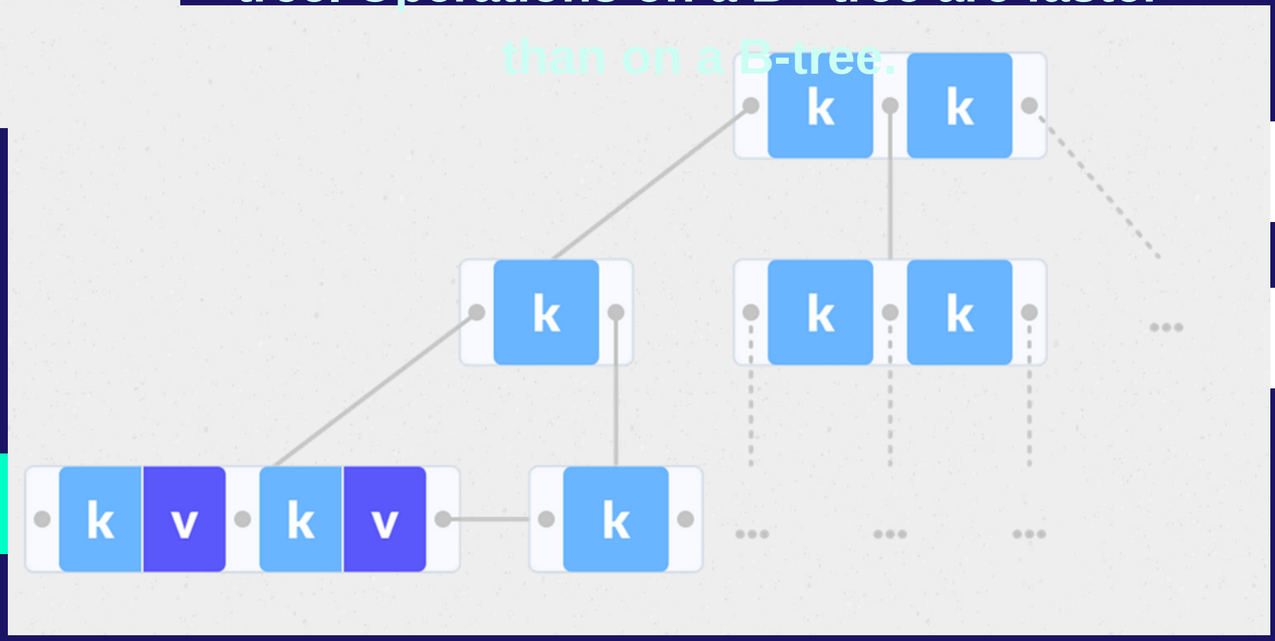
04

Each node can contain a maximum of m - 1 keys and a minimum of [m/2] - 1 keys

Comparison between a B-tree and a B+ Tree



The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree. The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree. Operations on a B+ tree are faster than on a B-tree.



Searching on a B+ Tree

Step 1

Start from the root node. Compare k with the keys at the root node [k1, k2, k3,.....km - 1]

Step 2

If k < k1, go to the left child of the root node.

Step 3

Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.

Step 4

If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.

Step 5

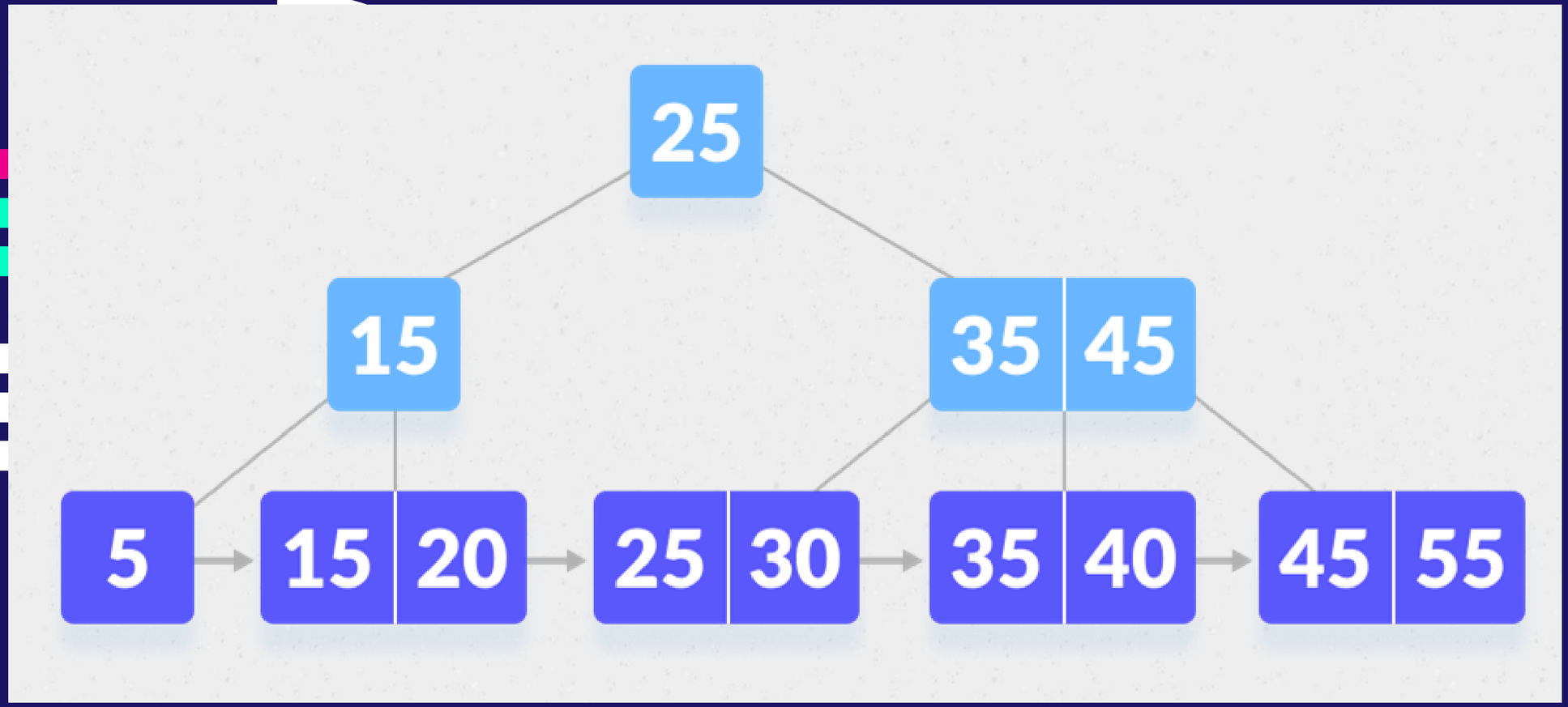
Repeat the above steps until a leaf node is reached.

Step 6

If k exists in the leaf node, return true else return false.

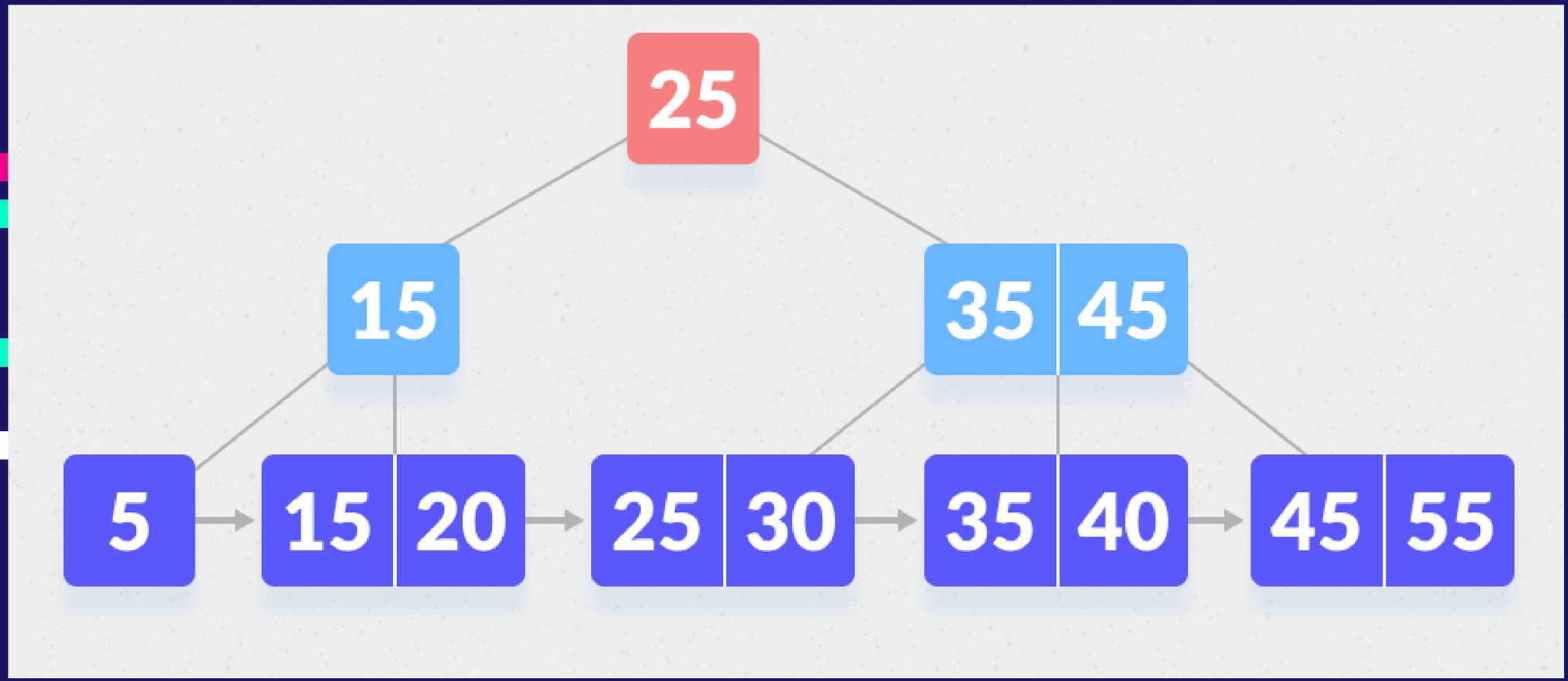
Searching Example on a B+

Let us search k = 45 on the following B+ tre e



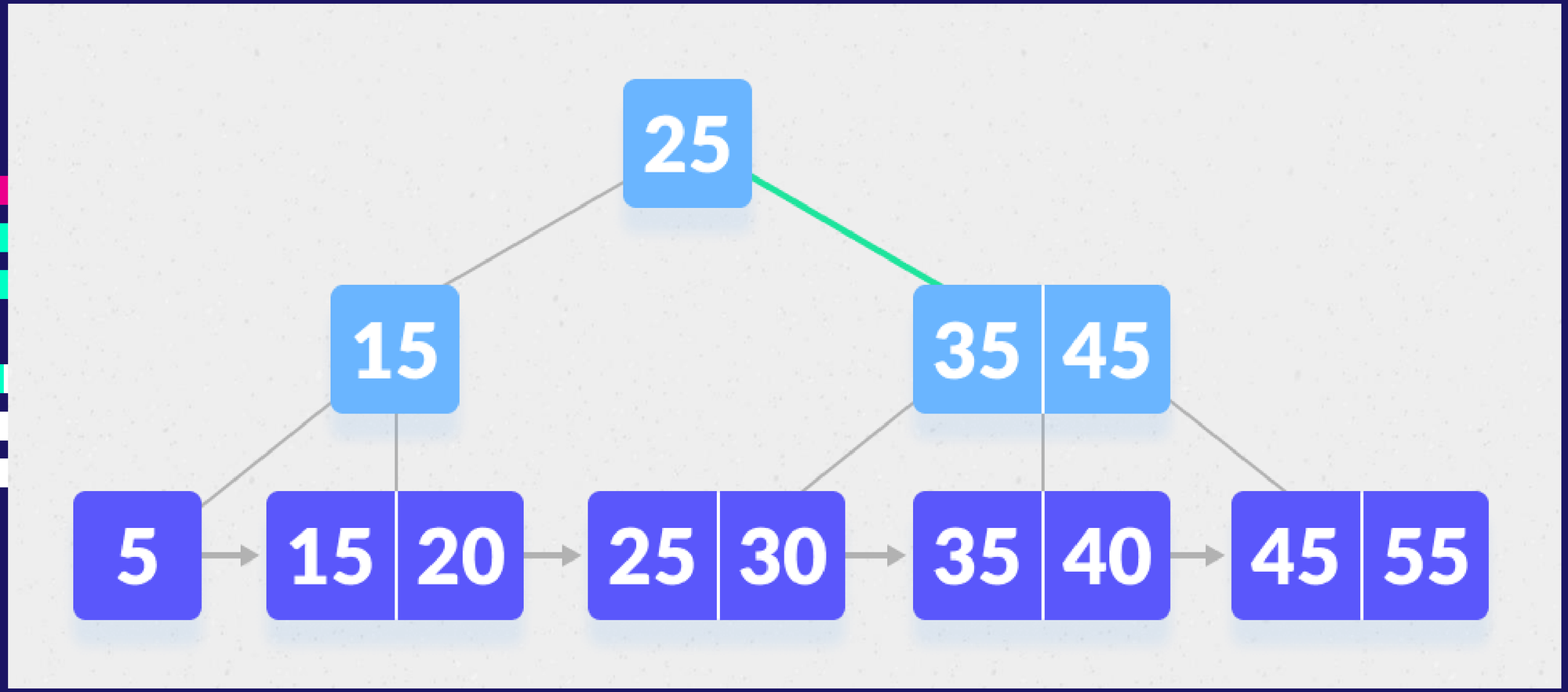
Searching Example on a B+

1. Compare k with the root node.



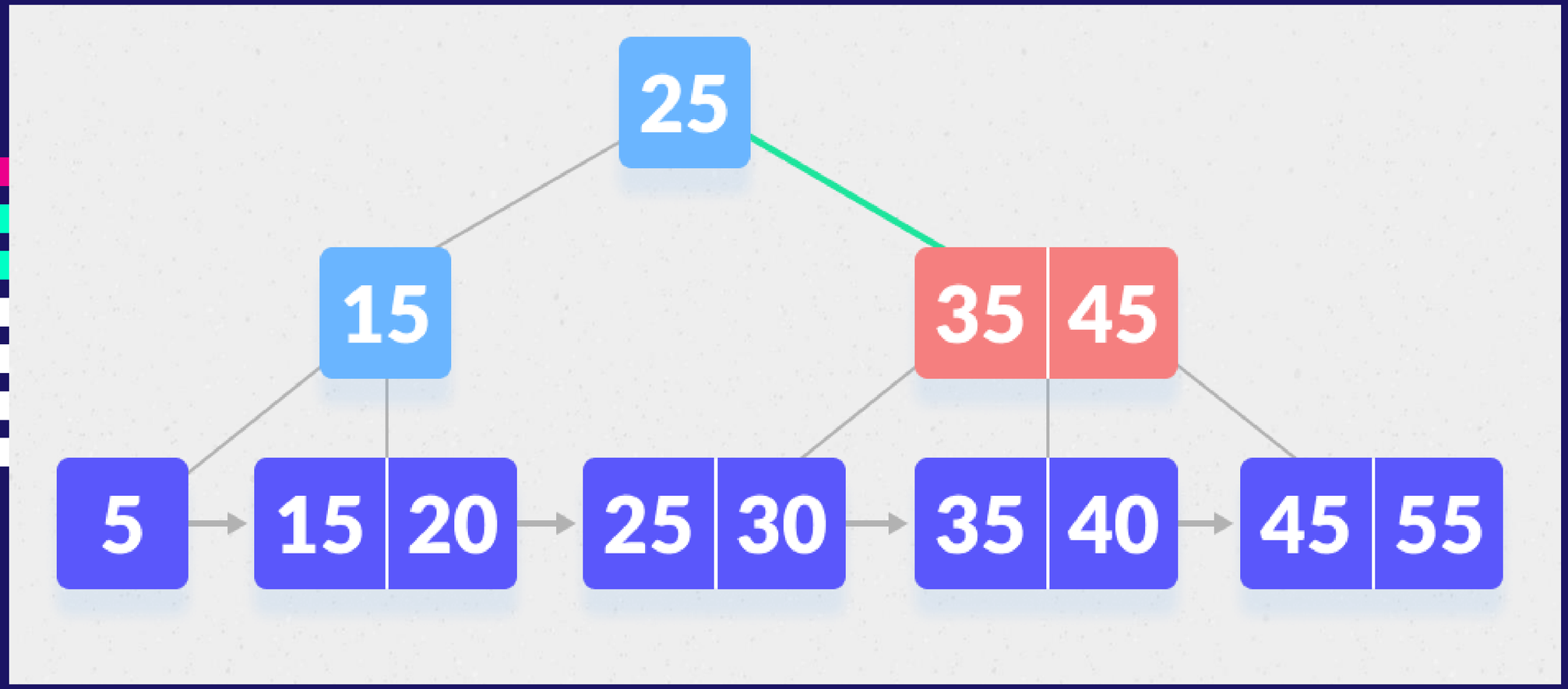
Searching Example on a B+

2. Since k > 25, go to the right child.



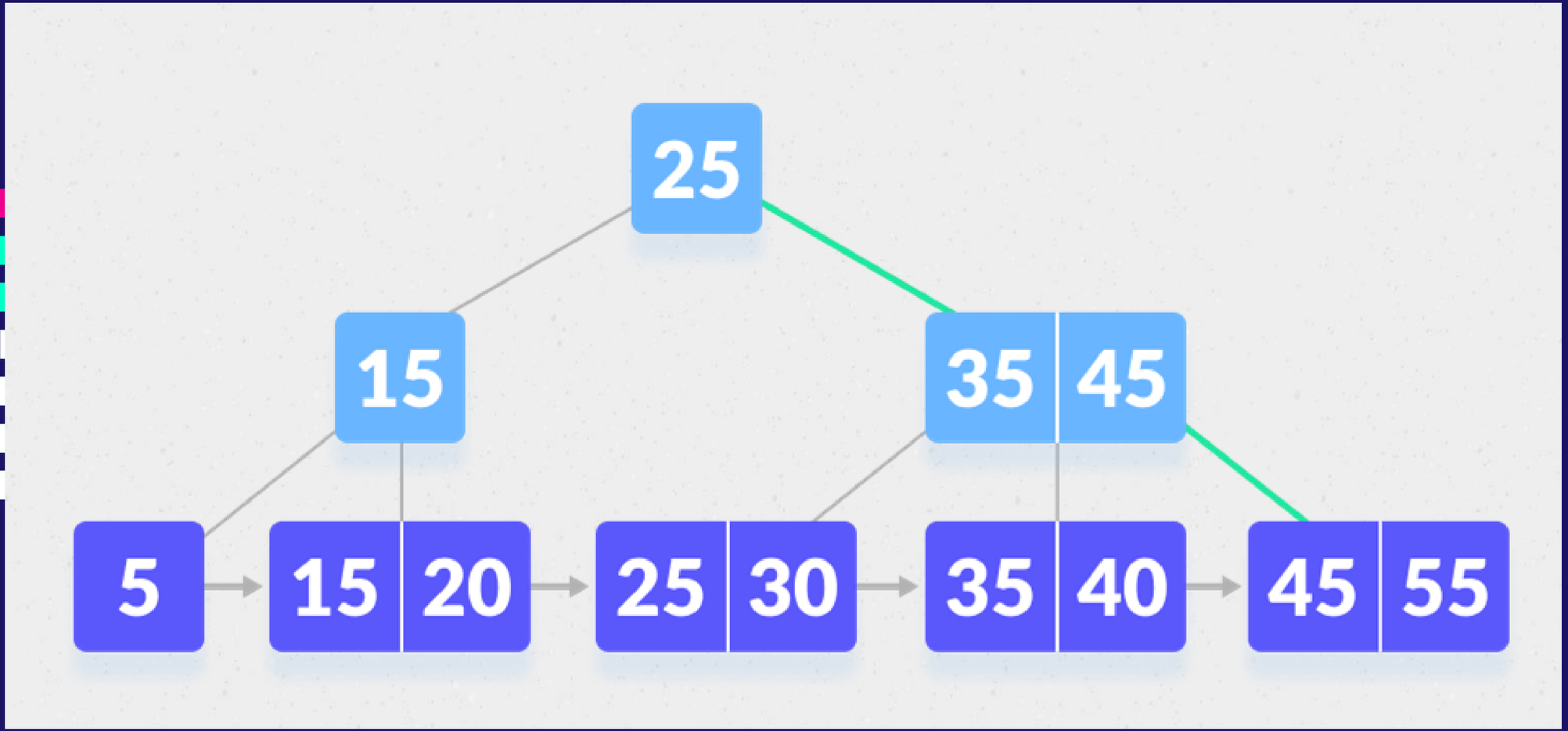
Searching Example on a B+

3. Compare k with 35. Since k > 30, compare k with 45.



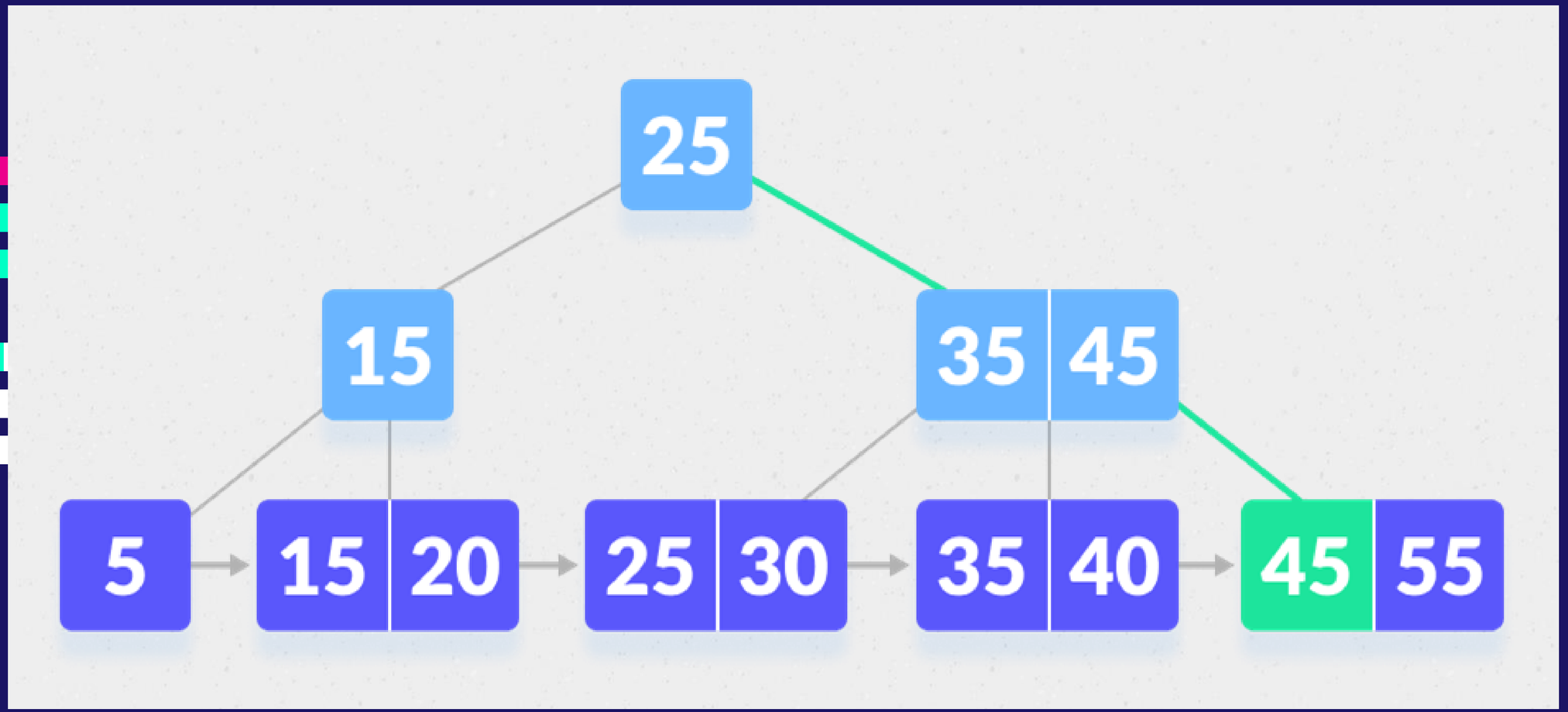
Searching Example on a B+

4. Since k ≥ 45, so go to the right child.



Searching Example on a B+

5. Key is found



INSERTION ON A B+ TREE

INSERTION ON A B+ TREE

INSERTING AN ELEMENT INTO A B+ tree consists of three main events: searching the appropriate leaf, inserting the element and balancing/splitting the tree.

INSERTION OPERATION

BEFORE INSERTING AN ELEMENT INTO A B+ TREE, THESE PROPERTIES MUST BE KEPT IN MIND.

- THE ROOT HAS AT LEAST TWO CHILDREN.
- EACH NODE EXCEPT ROOT CAN HAVE A MAXIMUM OF M CHILDREN AND AT LEAST M/2 CHILDREN.
- EACH NODE CAN CONTAIN A MAXIMUM OF M - 1 KEYS AND A MINIMUM OF ⌈M/2⌉ - 1 KEYS.

CASE I

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

CASE II

- If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way

2. Break the node at m/2th position.

3. Add m/2th key to the parent node as well.

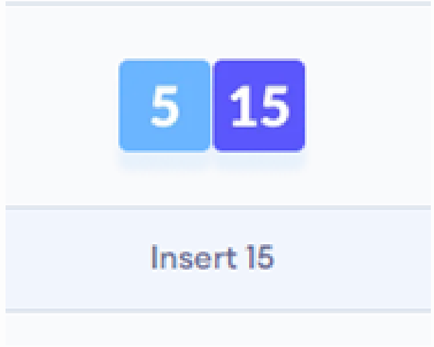
4. If the parent node is already full, follow steps 2 to 3.

Insertion Example

The elements to be inserted are 5,15, 25, 35, 45.

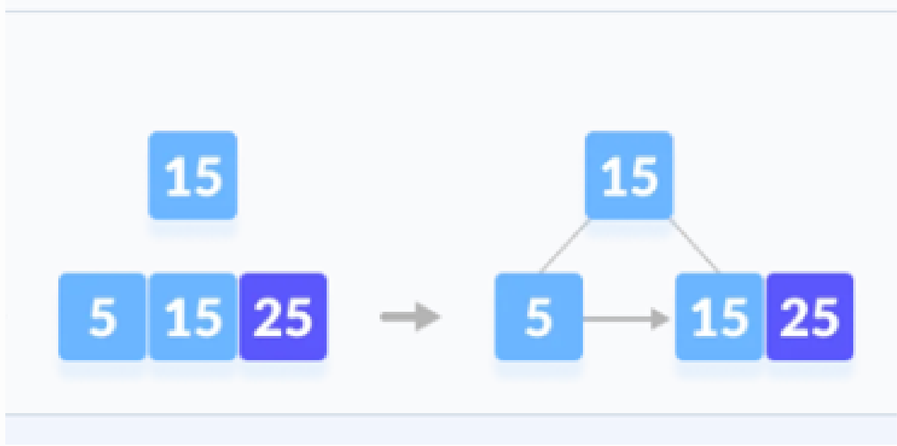
- Insert 5

2. Insert 15



3. Insert 25

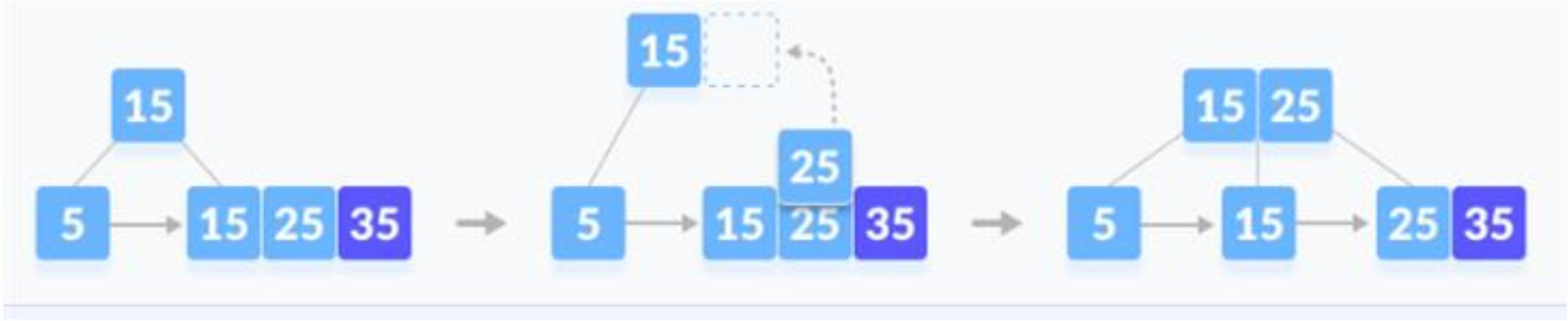
Case 2* Leaf node is full



Insertion Example

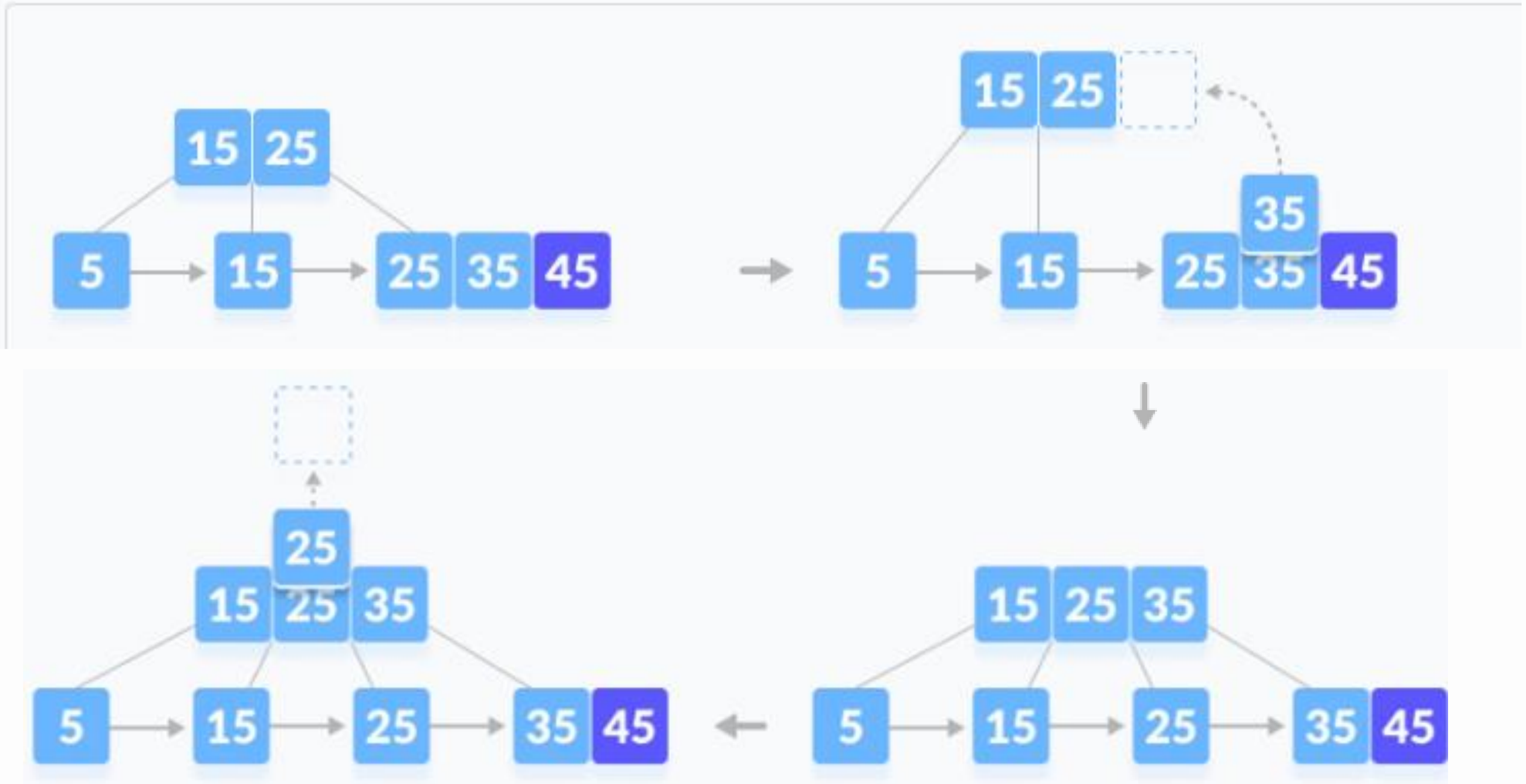
The elements to be inserted are 5,15, 25, 35, 45.

4. Insert 35



Insertion Example

The elements to be inserted are 5,15, 25, 35, 45.
5. Insert 45



Insertion Example

The elements to be inserted are 5,15, 25, 35, 45.
5. Insert 45

DELETION FROM A B+ TREE

DELETION FROM A B+ TREE

DELETING AN ELEMENT ON A B+ TREE CONSISTS OF THREE MAIN EVENTS: SEARCHING THE NODE WHERE THE KEY TO BE DELETED EXISTS, DELETING THE KEY AND BALANCING THE TREE IF REQUIRED.UNDERFLOW IS A SITUATION WHEN THERE IS LESS NUMBER OF KEYS IN A NODE THAN THE MINIMUM NUMBER OF KEYS IT SHOULD HOLD.

DELETION OPERATION

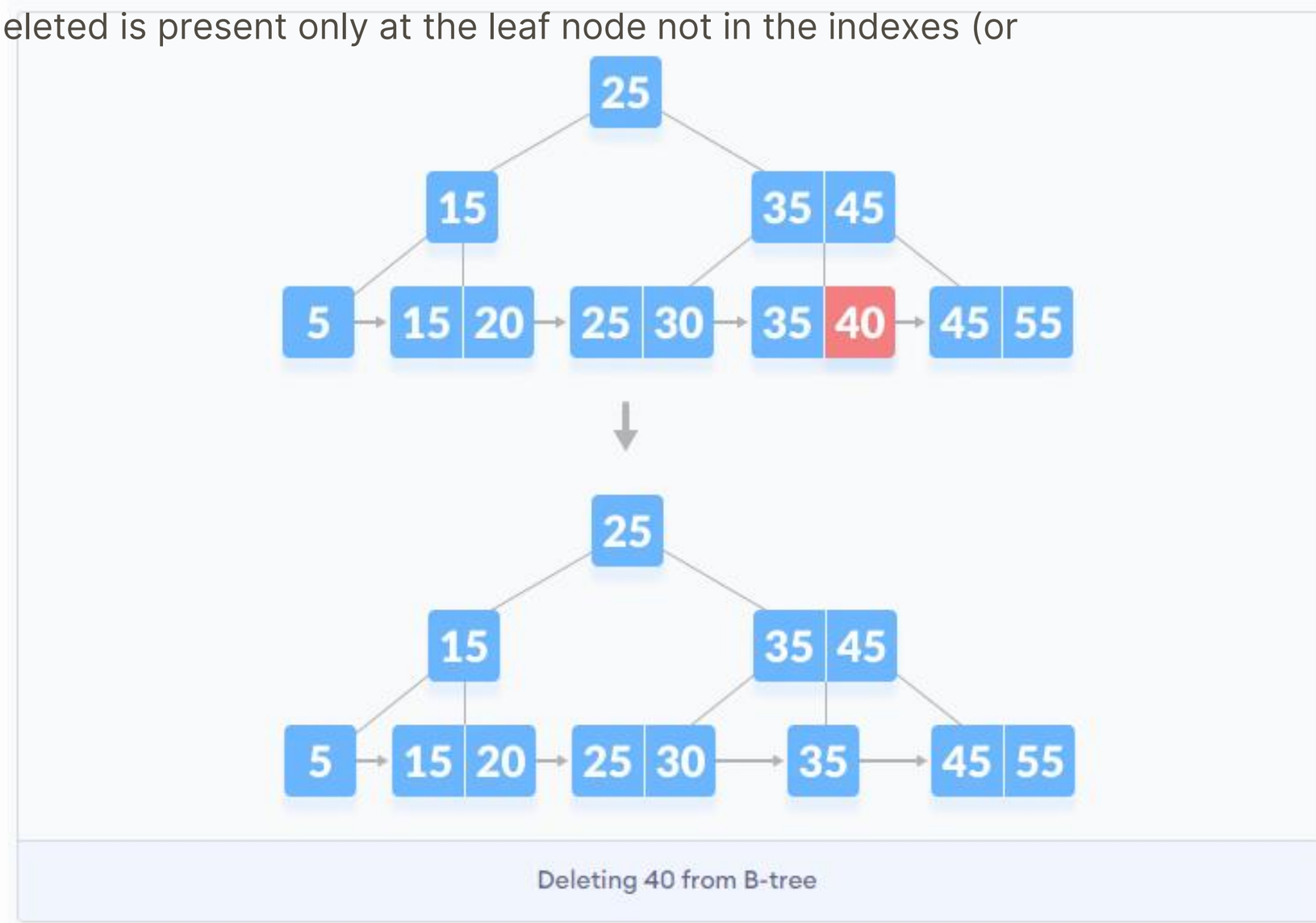
BEFORE GOING THROUGH THE STEPS BELOW, ONE MUST KNOW THESE FACTS ABOUT A B+ TREE OF DEGREE M.

- A NODE CAN HAVE A MAXIMUM OF M CHILDREN. (I.E. 3)
- A node can contain a maximum of m - 1 keys. (i.e. 2)
- A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)
- A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)

CASE I

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

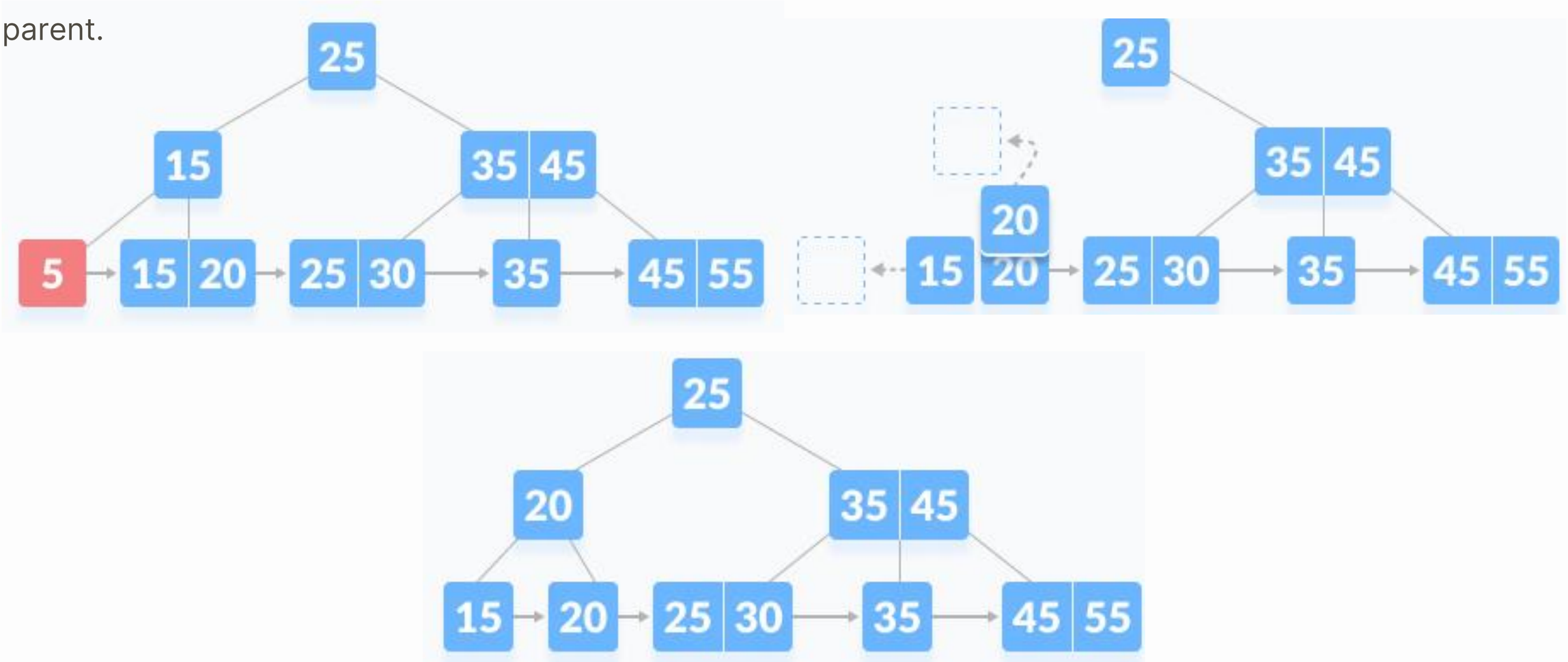
- The key to be deleted is present only at the leaf node not in the indexes (or internal nodes).



Deleting 40 from B-tree

CASE I

2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

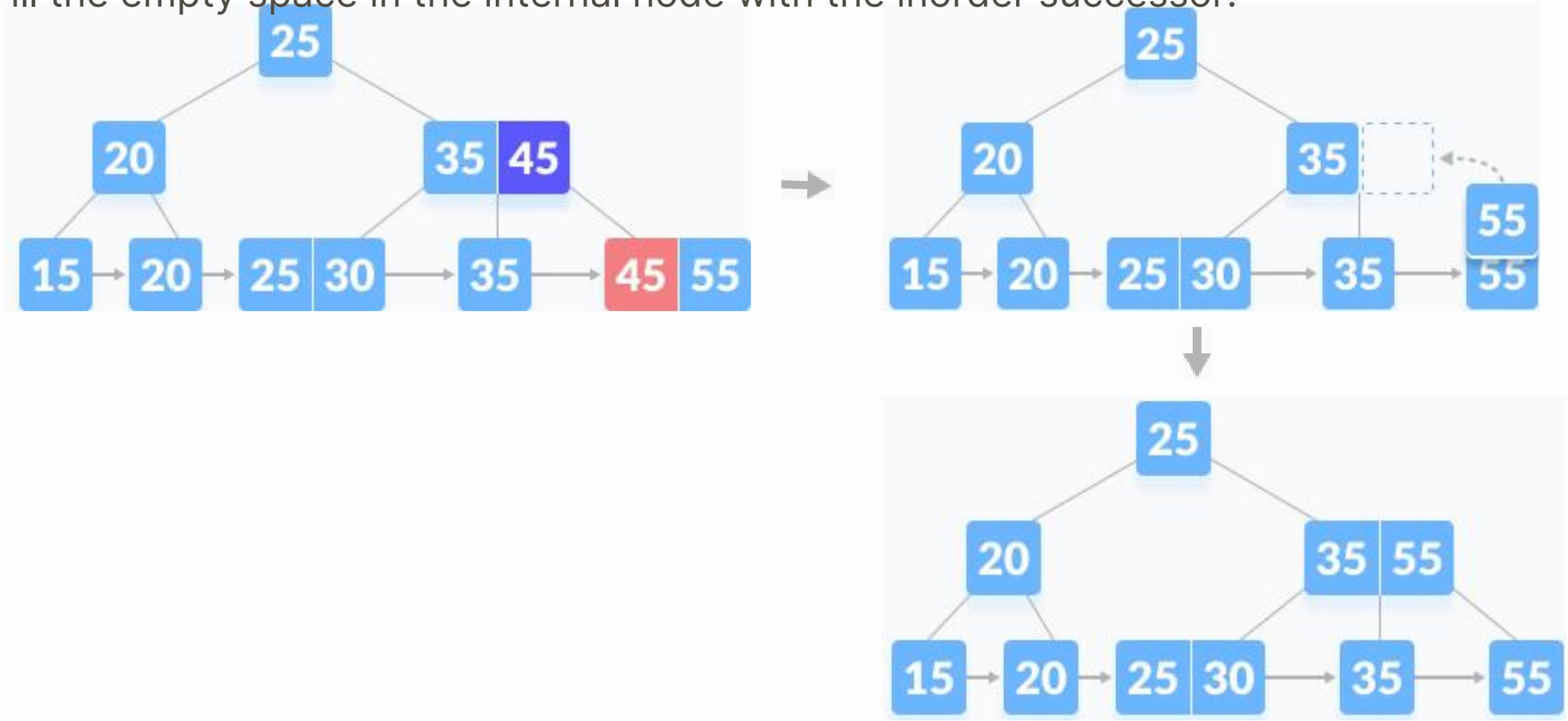


CASE II

The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well.

1.If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.

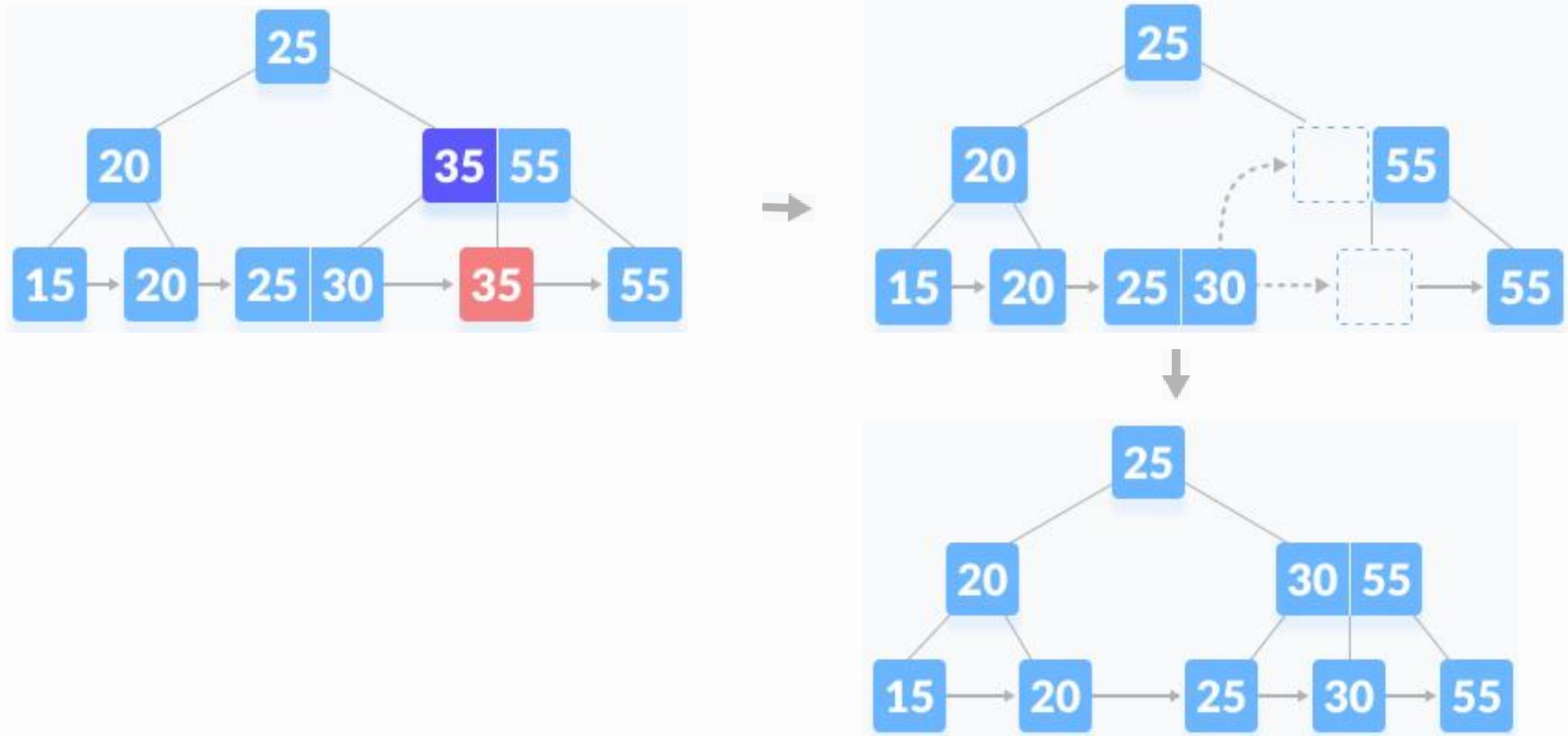
Fill the empty space in the internal node with the inorder successor.



CASE II

2.If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).

Fill the empty space created in the index (internal node) with the borrowed key.

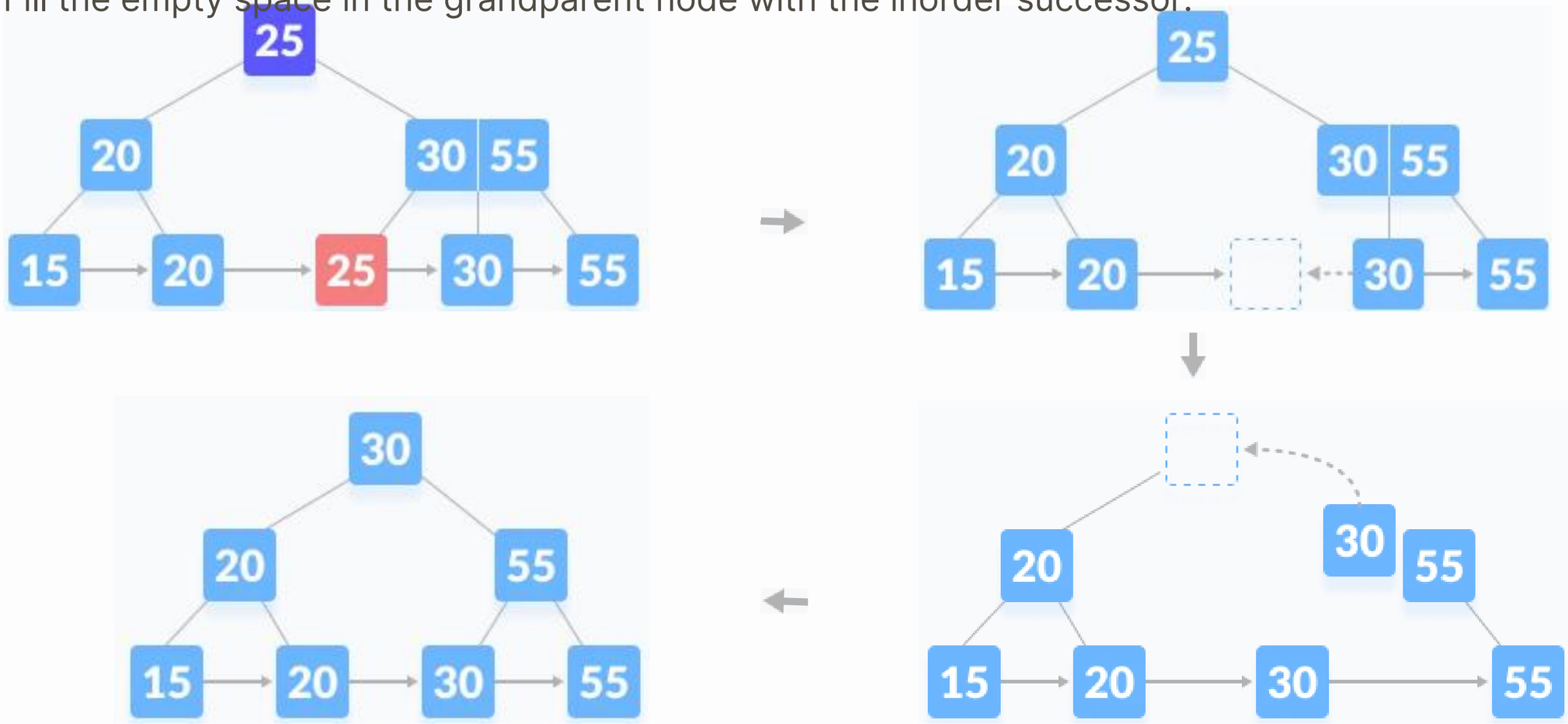


CASE II

3.This case is similar to Case II(1) but here, empty space is generated above the immediate parent node.

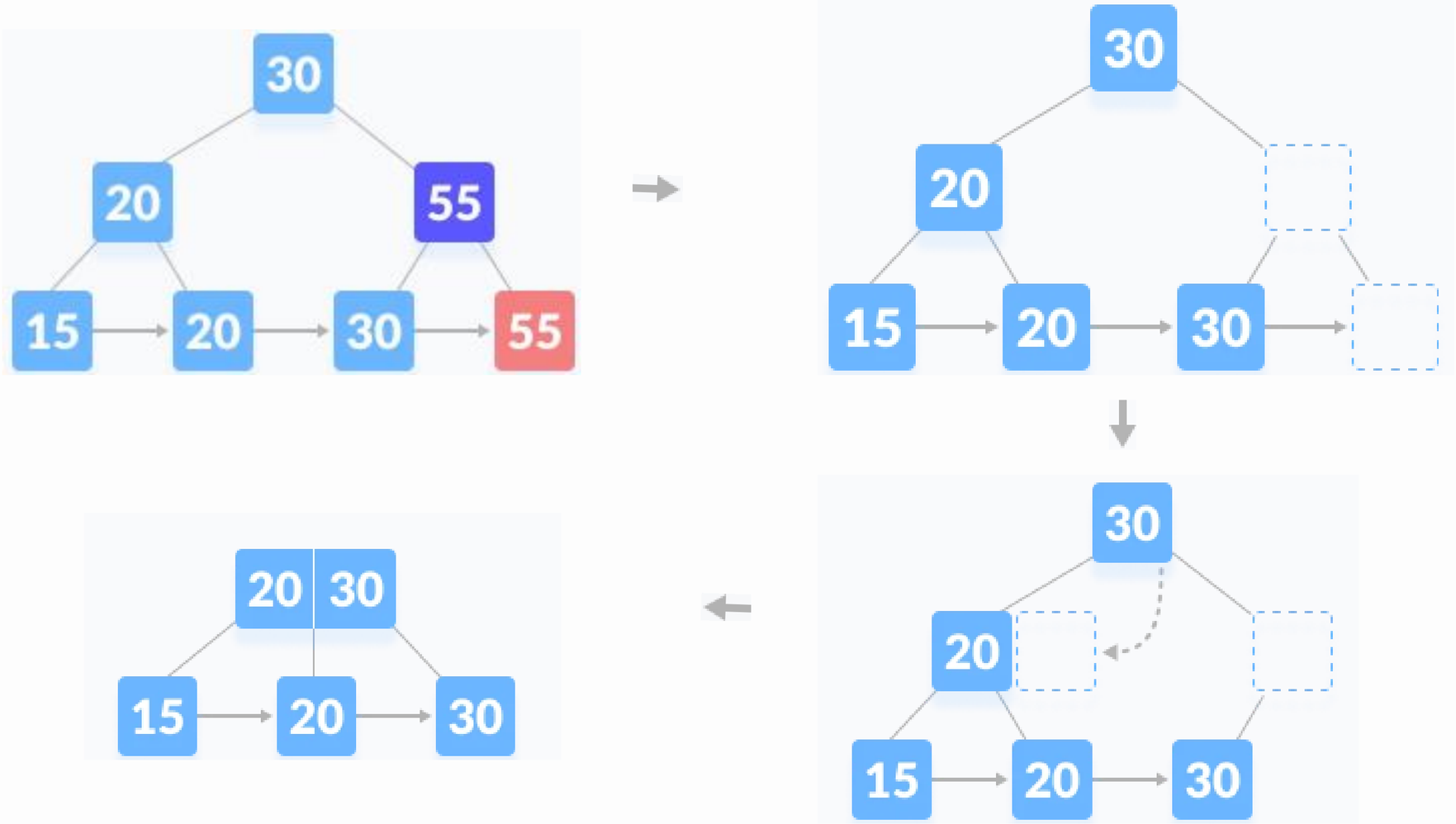
After deleting the key, merge the empty space with its sibling.

Fill the empty space in the grandparent node with the inorder successor.



CASE III

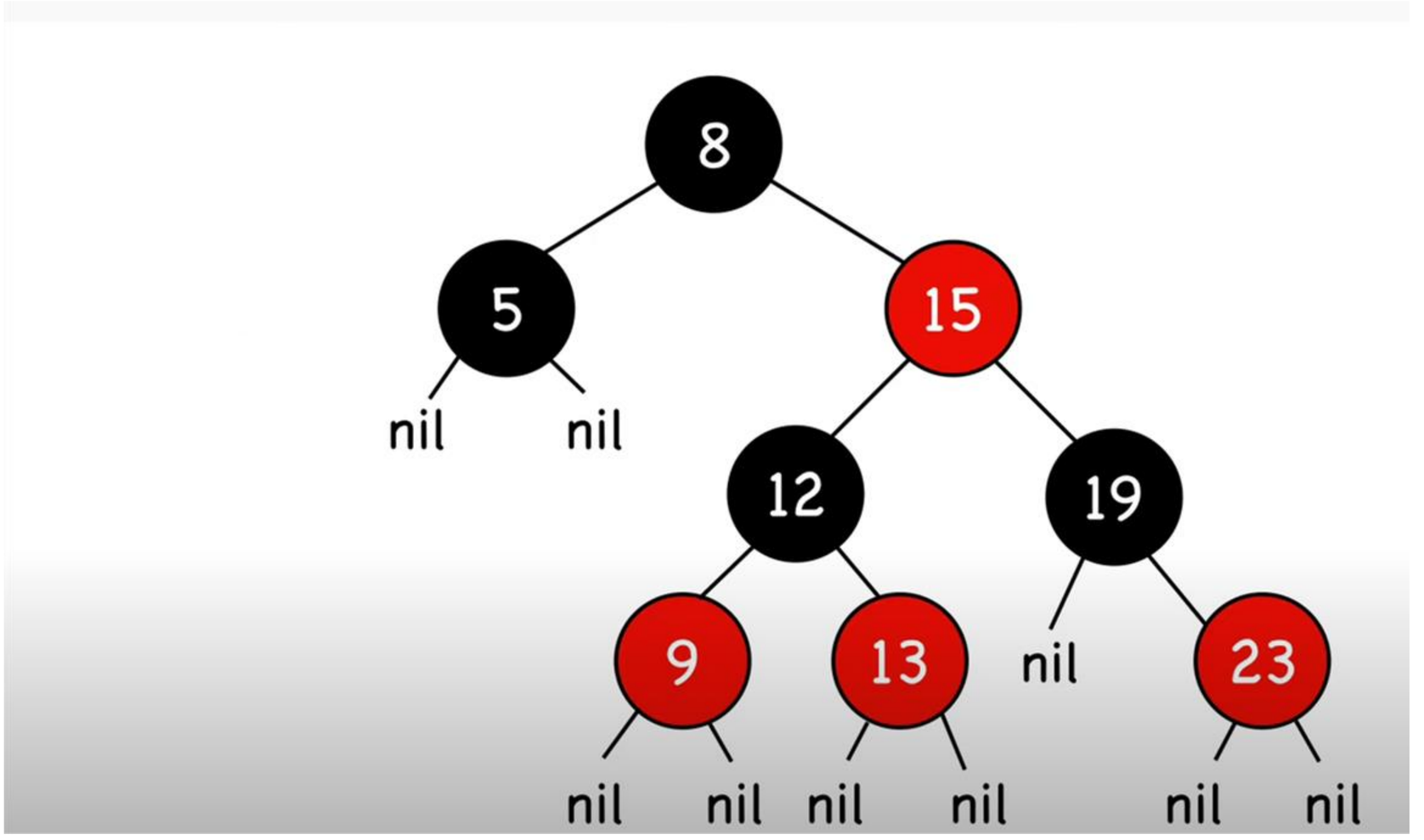
In this case, the height of the tree gets shrinked. It is a little complicated.Deleting 55 from the tree below leads to this condition.

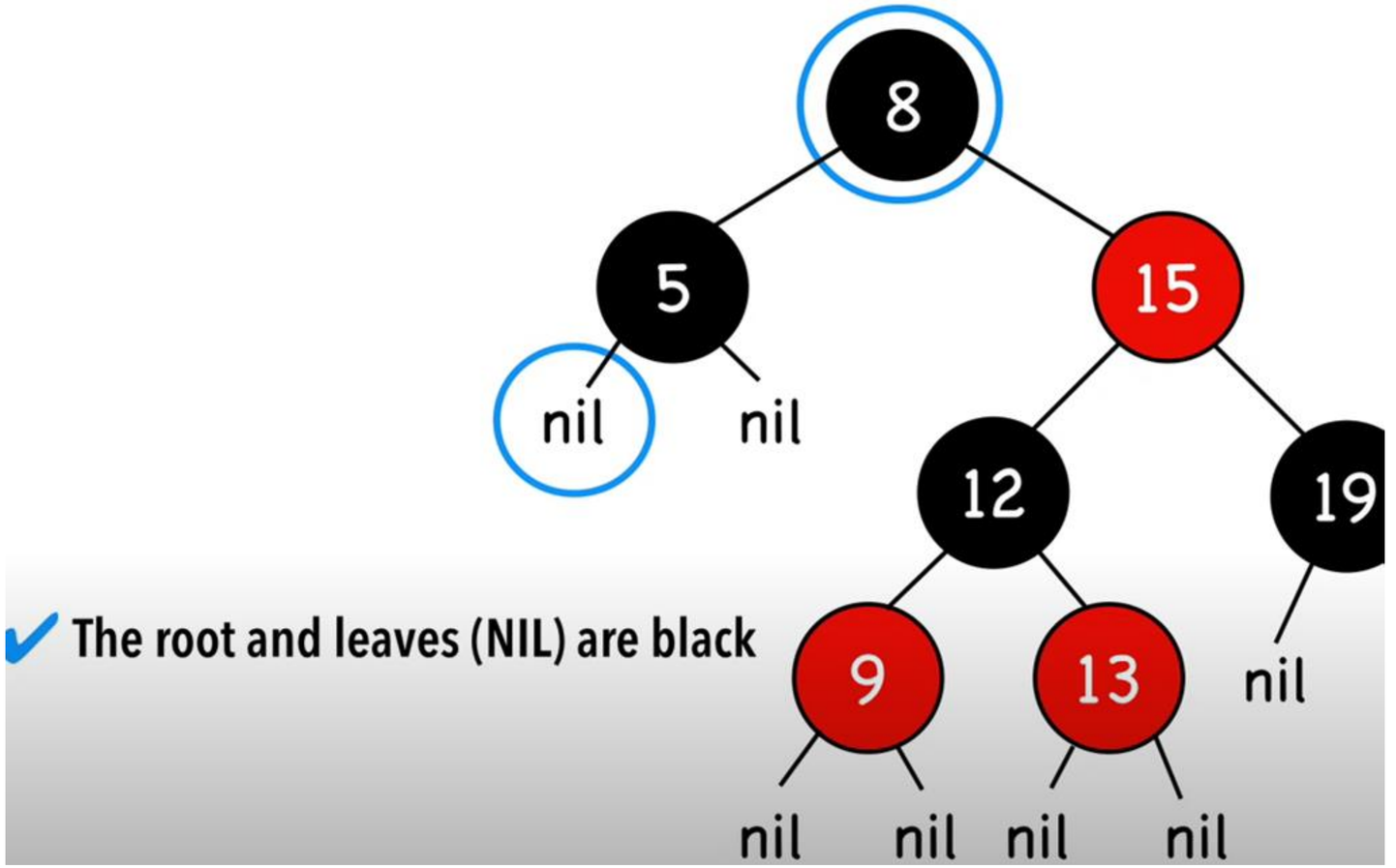


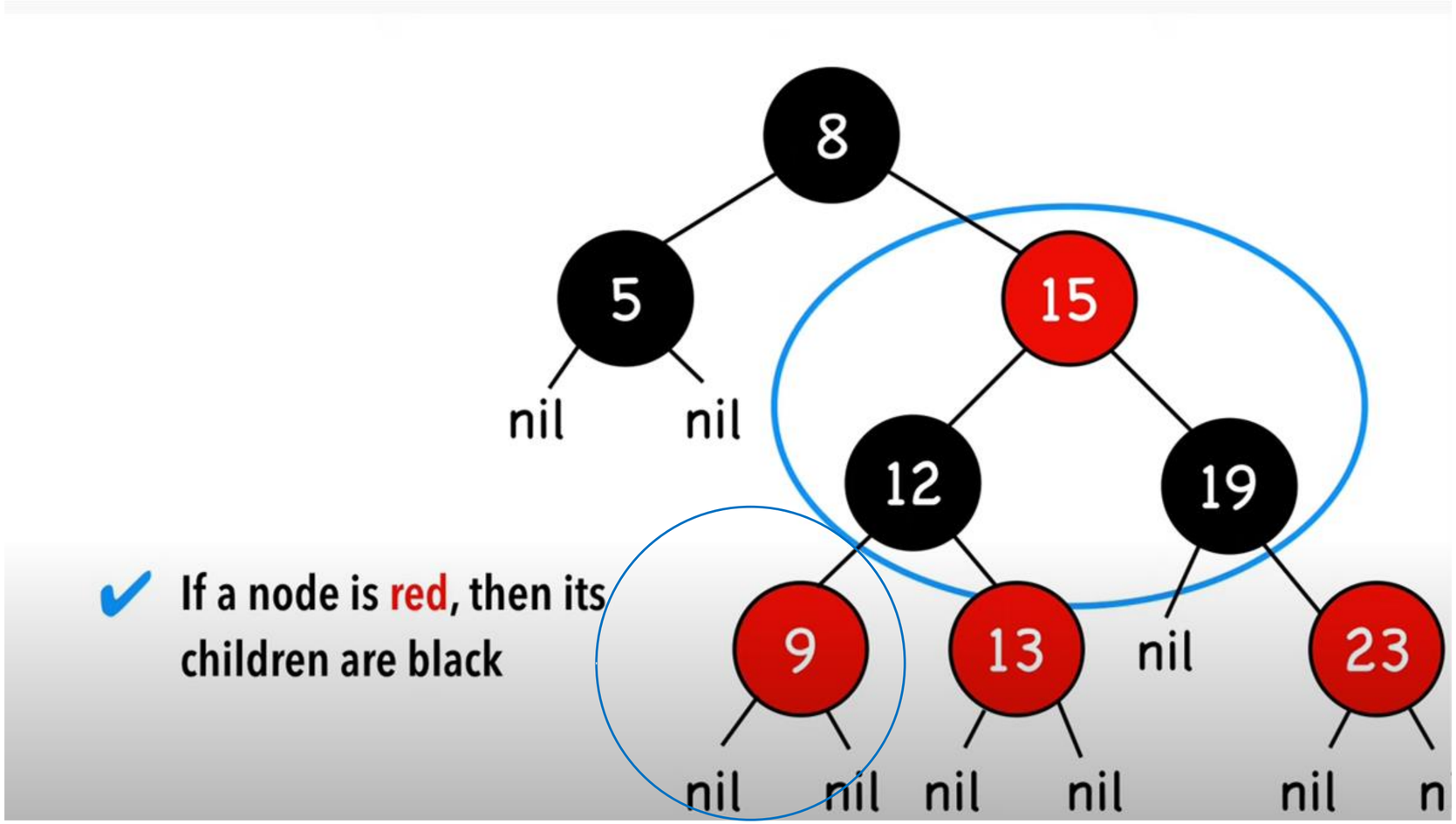
RED – BLACK TREE

red–black tree

1. A node is either red or black.
2. The root and leaves (NIL) are black.
3. If a node is red, then its children are black.
4. All paths from a node to its NIL descendants contain the same number of black nodes.


8
5
15
nil
nil
12
19
9
13
nil
23
nil
nil
nil
nil
nil
nil


8
5
15
nil
nil
12
19
9
13
nil
nil
nil
nil
nil
✔ The root and leaves (NIL) are black


8
5
15
nil
nil
12
19
If a node is red, then its children are black
9
13
nil
23
nil
nil
nil
nil
nil
n

extra notes

1. Nodes require one storage bit to keep track of color.
2. The longest path (root to farthest NIL) is no more than twice the length of the shortest path (root to nearest NIL).
   - Shortest path: all black nodes
   - Longest path: alternating red and black

operations

Search

Insert

Remove

RED BLACK TREE INSERTION

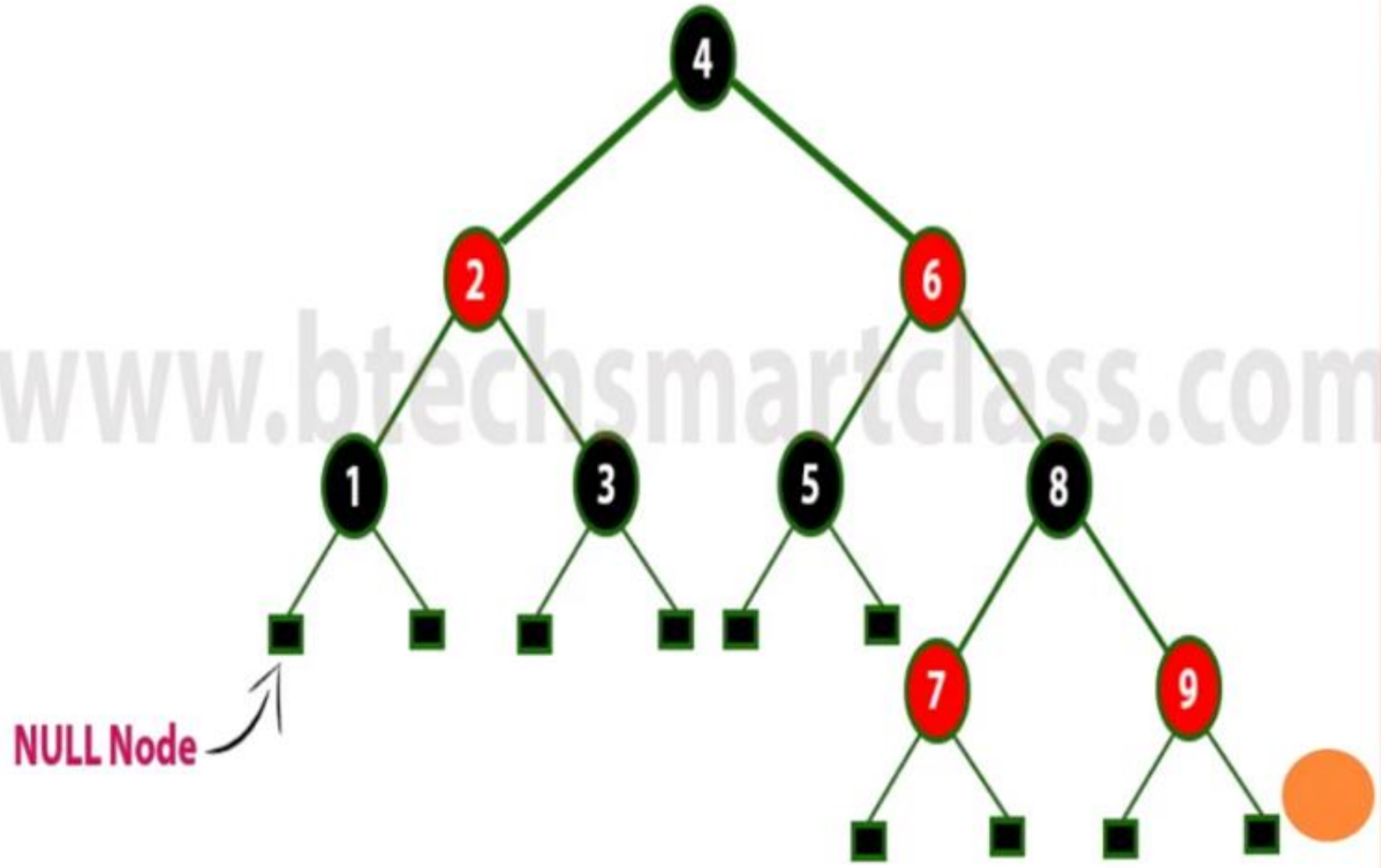
Red Black Tree

- Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.
- In a Red Black Tree the color of a node is decided based on the Red Black Tree properties.

PROPERTIES OF RED BLACK TREE

- Every node is either red or black.
- The root is black.
- Every leaf (*nil*) is black.
- If a node is red, then both its children are black.
- For each node, all paths from the node to descendant leaves contain the same number of black nodes.

EXAMPLE



INSERTION INTO RED BLACK TREE

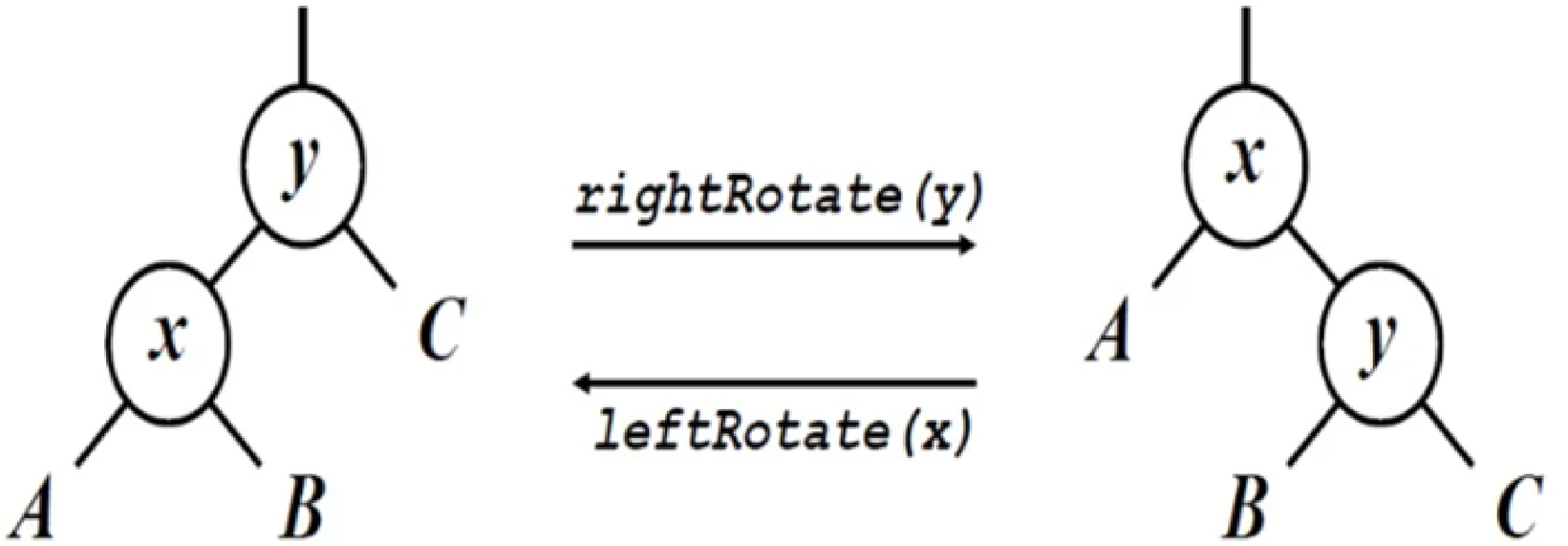
- In a Red Black Tree, every new node must be inserted with color RED.
- The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property.
- After every insertion operation, we need to check all the properties of Red Black Tree.
- If all the properties are satisfied then we go to next operation otherwise we need to perform following operation to make it Red Black Tree.

1. Recolor
2. Rotation followed by Recolor

RB TREE ROTATIONS

There are two type of rotations:

- left rotation
- right rotation



The insertion operation in Red Black tree is performed using following steps...

- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.
- **step 3:** If tree is not Empty then insert the newNode as a leaf node with Red color.
- **Step 4:** If the parent of newNode is Black then exit from the operation.
- **Step 5:** If the parent of newNode is Red then check the color of parent node's sibling of newNode.
- **Step 6:** If it is Black or NULL node then make a suitable Rotation and Recolor it.
- **Step 7:** If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.

EXAMPLE

Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

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

insert (18)

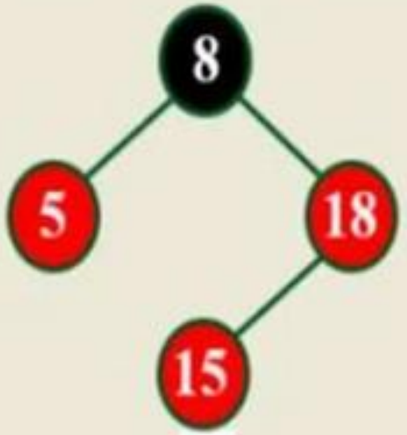
Tree is not Empty. So insert newNode with red color.

insert (5)

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

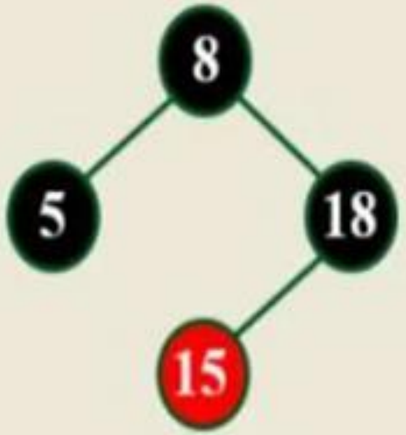
insert (15)

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.

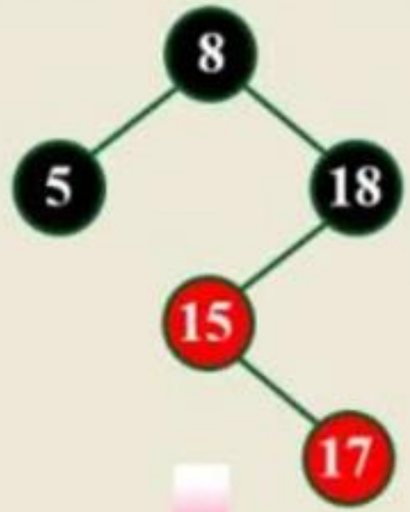
After RECOLOR



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

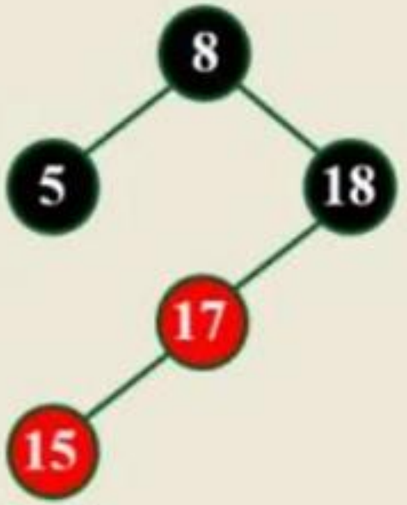
insert (17)

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

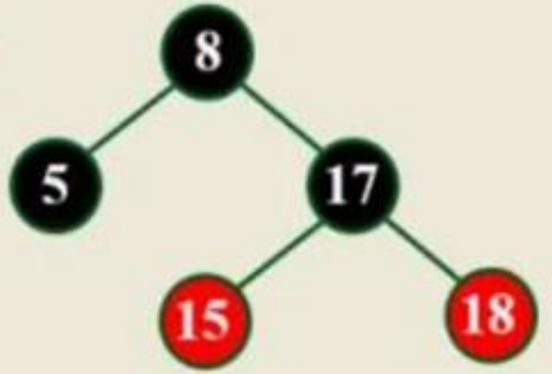


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation



After Right Rotation & Recolor

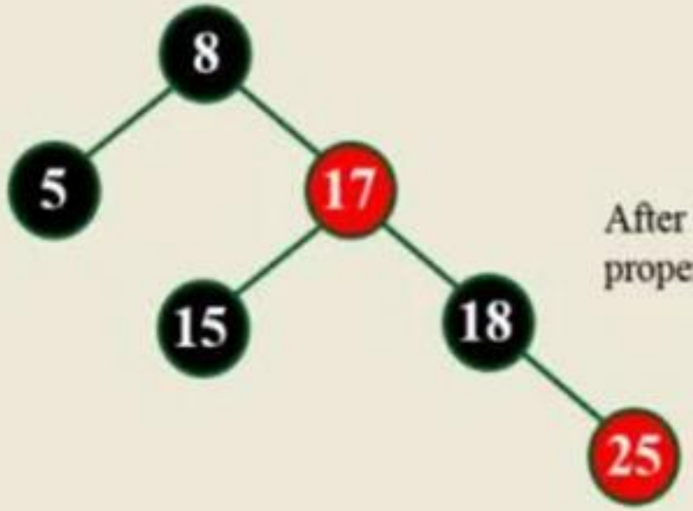


insert (25)

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.

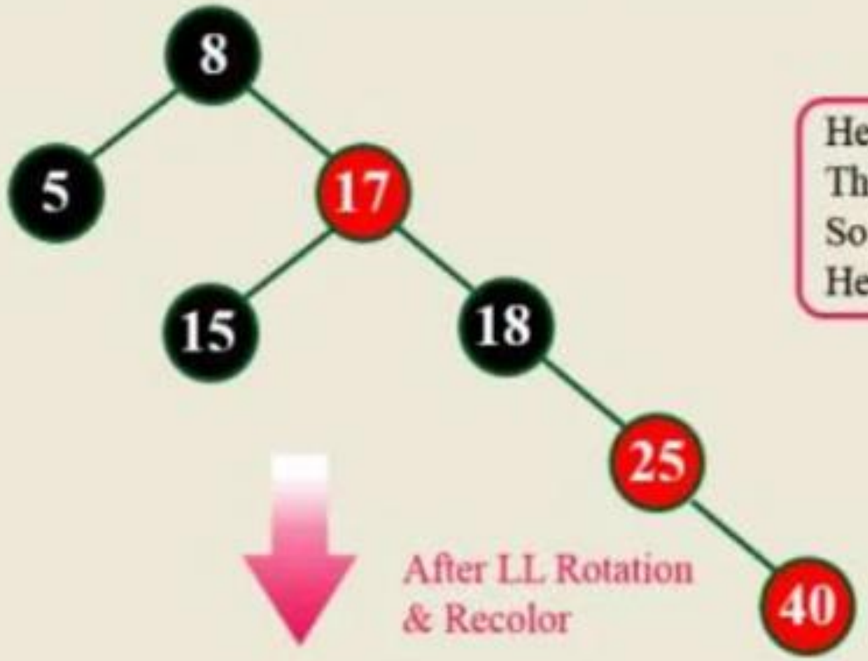
After Recolor



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

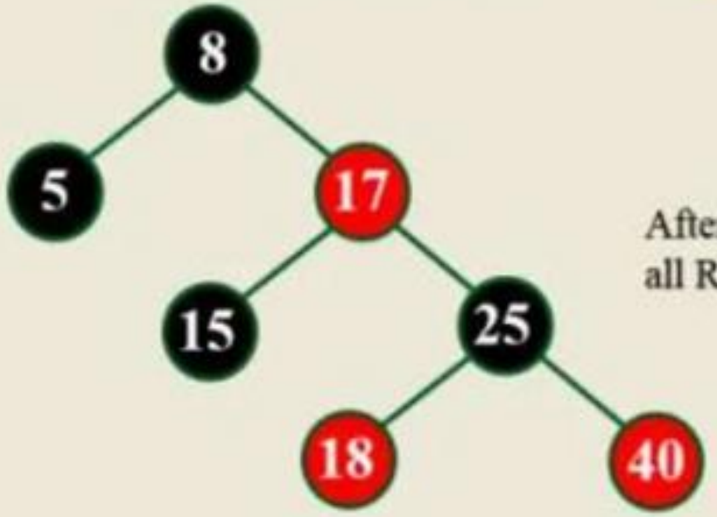
insert (40)

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.

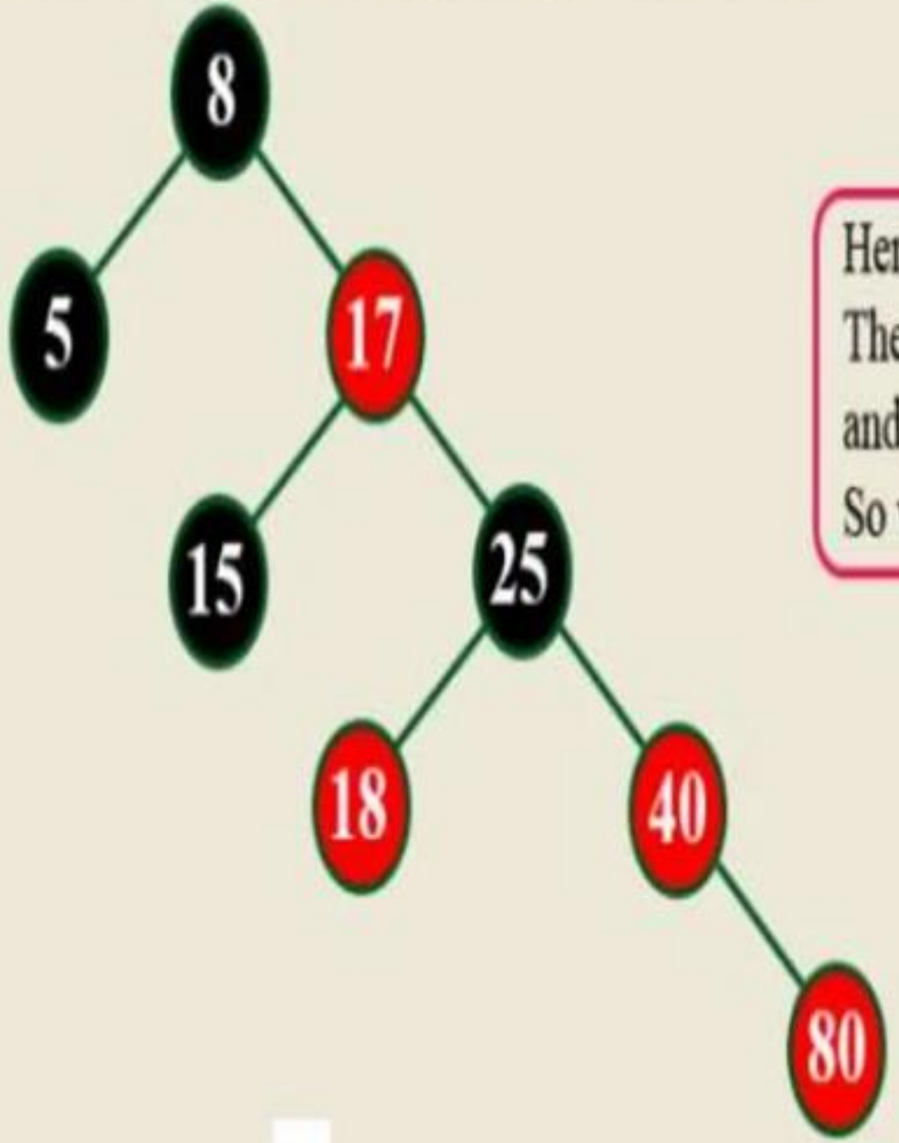
After LL Rotation
& Recolor



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

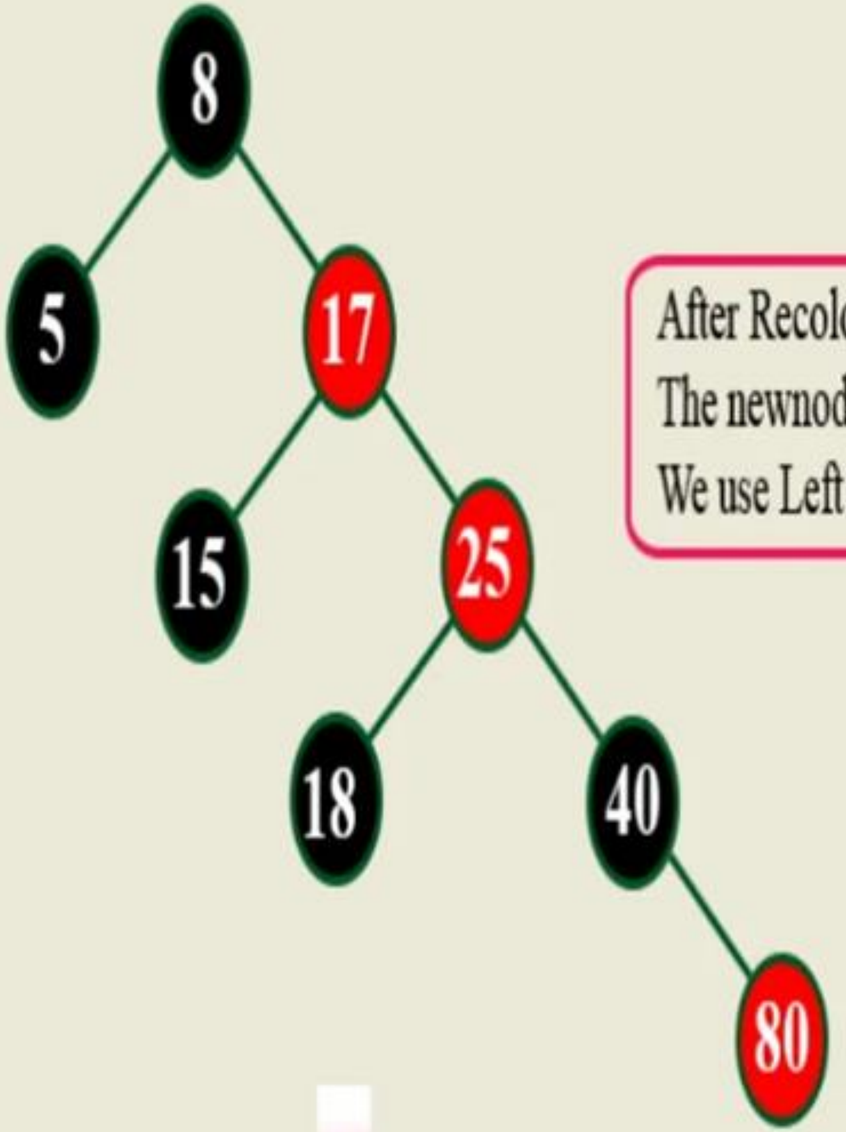
insert (80)

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



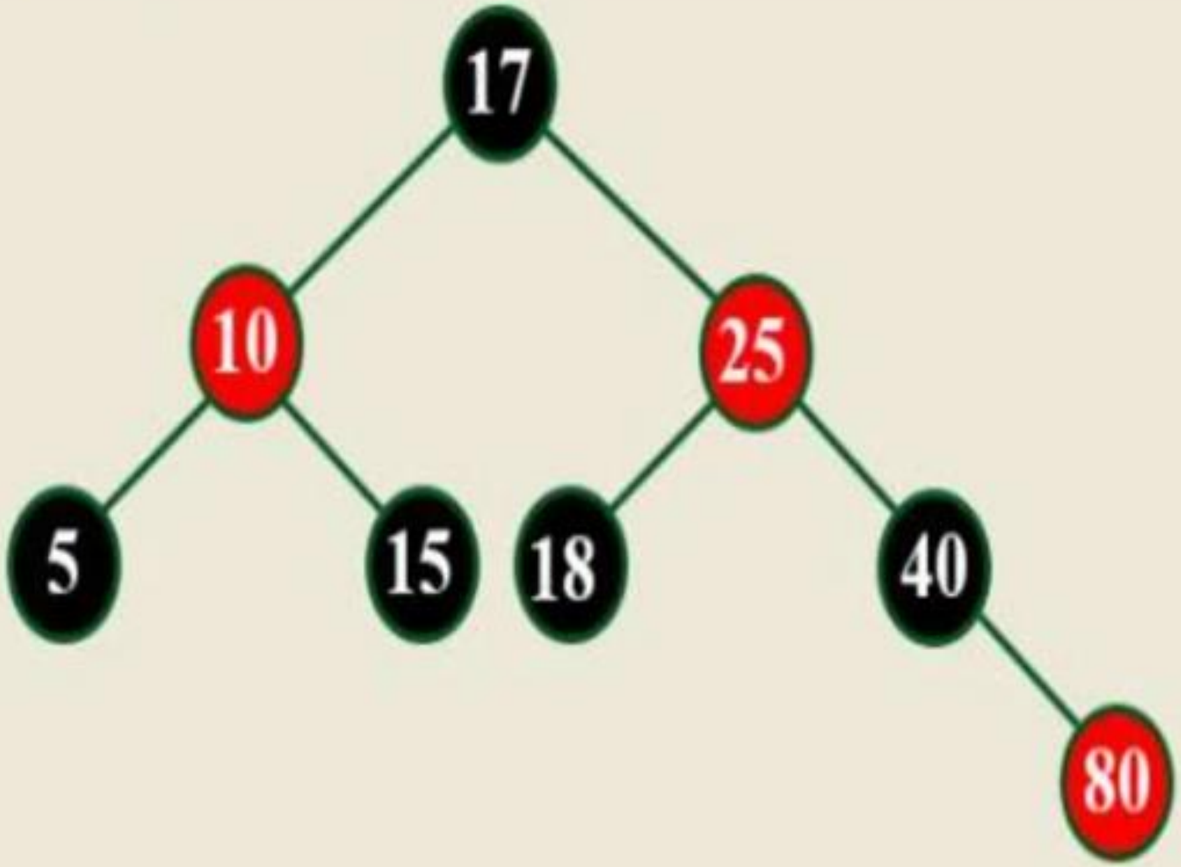
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

- =>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.

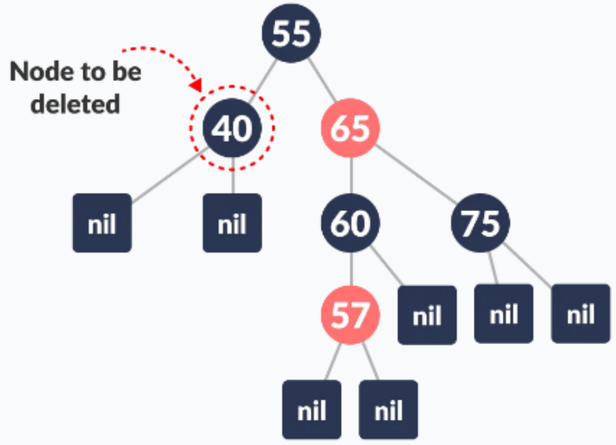
- =>After Left Rotation & Recolor



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

Red Black Tree Deletion

1. Let the nodeToBeDeleted be:



Node to be deleted

2. Save the color of `nodeToBeDeleted` in `origrinalColor`.

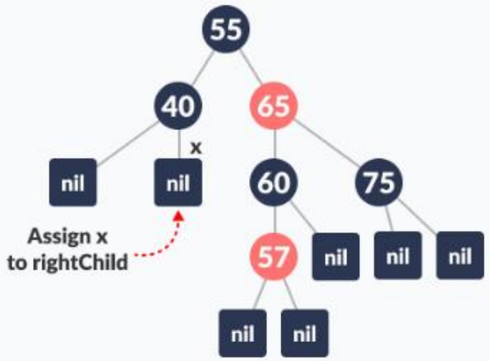
originalColor = Black

Saving original color

Red Black Tree Deletion

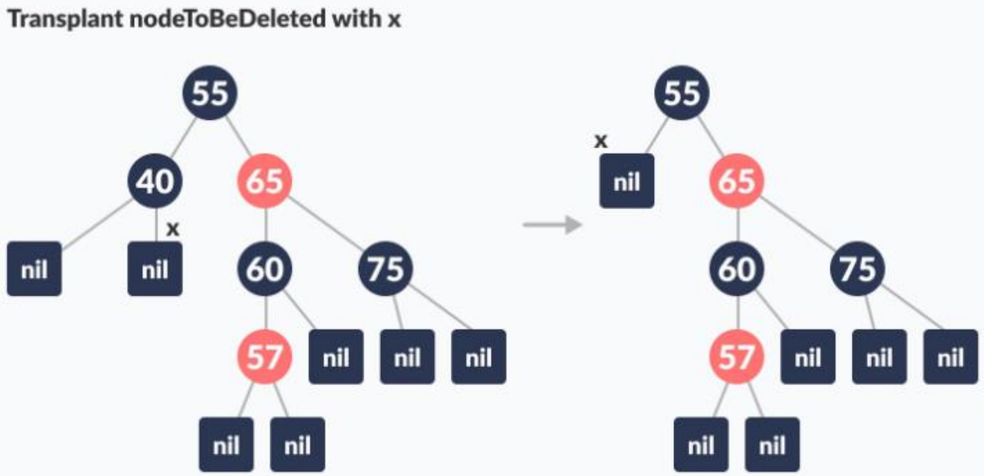
3. If the left child of `nodeToBeDeleted` is `NULL`

   a. Assign the right child of `nodeToBeDeleted` to x.



Assign x to the rightChild

b. Transplant `nodeToBeDeleted` with `x`.



Transplant nodeToBeDeleted with x

Red Black Tree Deletion

4. Else if the right child of nodeToBeDeleted is NULL
    - Assign the left child of nodeToBeDeleted into x.
    - Transplant nodeToBeDeleted with x.
5. Else
    - Assign the minimum of right subtree of noteToBeDeleted into y.
    - Save the color of y in originalColor.
    - Assign the rightChild of y into x.
    - If y is a child of nodeToBeDeleted, then set the parent of x as y.
    - Else, transplant y with rightChild of y.
    - Transplant nodeToBeDeleted with y.
    - Set the color of y with originalColor.
6. If the originalColor is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node x (which is occupying y's original position) has an extra black. This makes node x neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.

However, the color attribute of x is not changed rather the extra black is represented in x's pointing to the node.

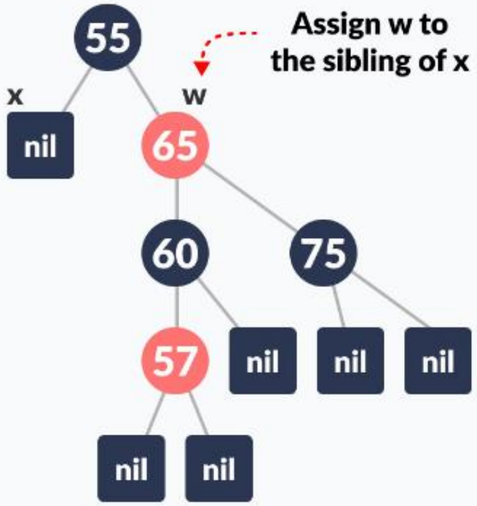
The extra black can be removed if

- It reaches the root node.
- If x points to a red-black node. In this case, x is colored black.

Algorithm to maintain Red-Black property after deletion

Following algorithm retains the properties of a red-black tree.

- Do the following until the x is not the root of the tree and the color of x is BLACK
- If x is the left child of its parent then,
  - Assign w to the sibling of x.

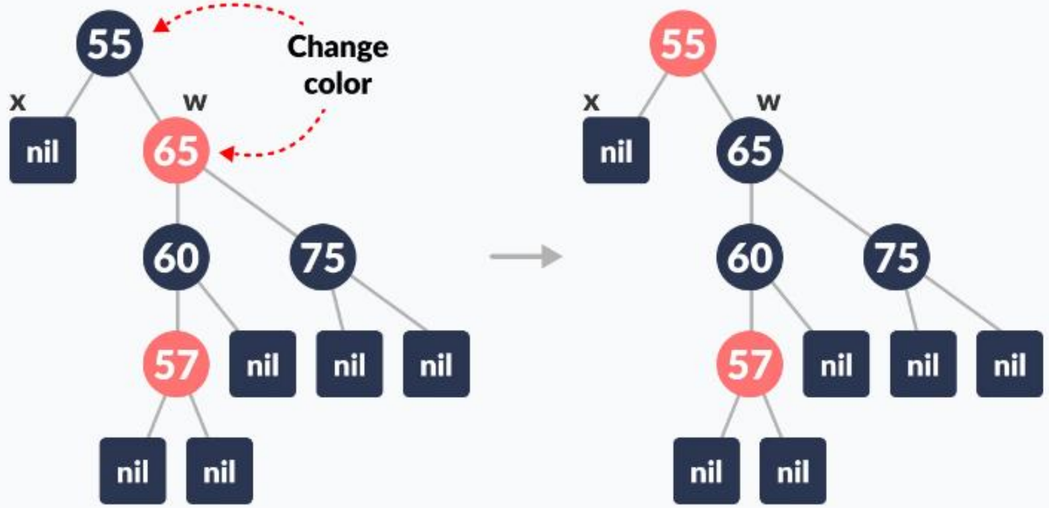


Assigning w

b.If the sibling of x is RED,

Case-I:

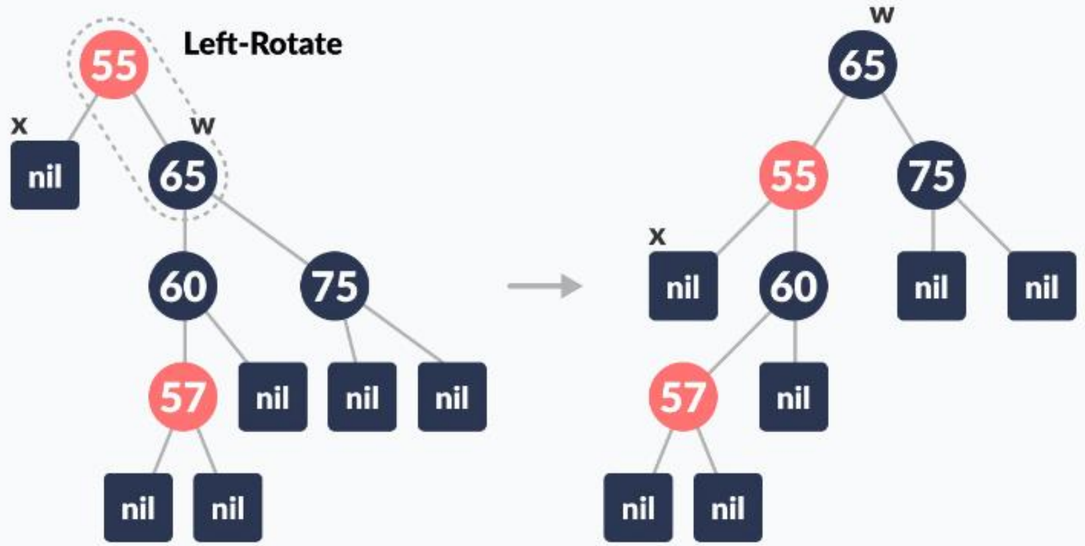
- Set the color of the right child of the parent of x as BLACK.
- Set the color of the parent of x as RED.



Color change

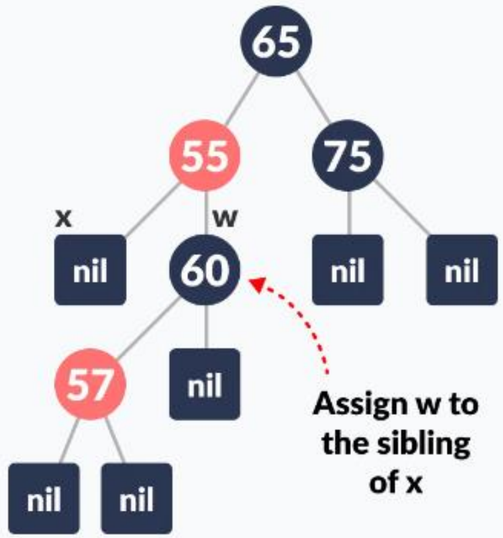
Algorithm to maintain Red-Black property after deletion

c. Left-Rotate the parent of x.



Left-rotate

d. Assign the rightChild of the parent of x to w.



Reassign w

Algorithm to maintain Red-Black property after deletion

c. If the color of both the right and the leftChild of w is BLACK,

Case-II:

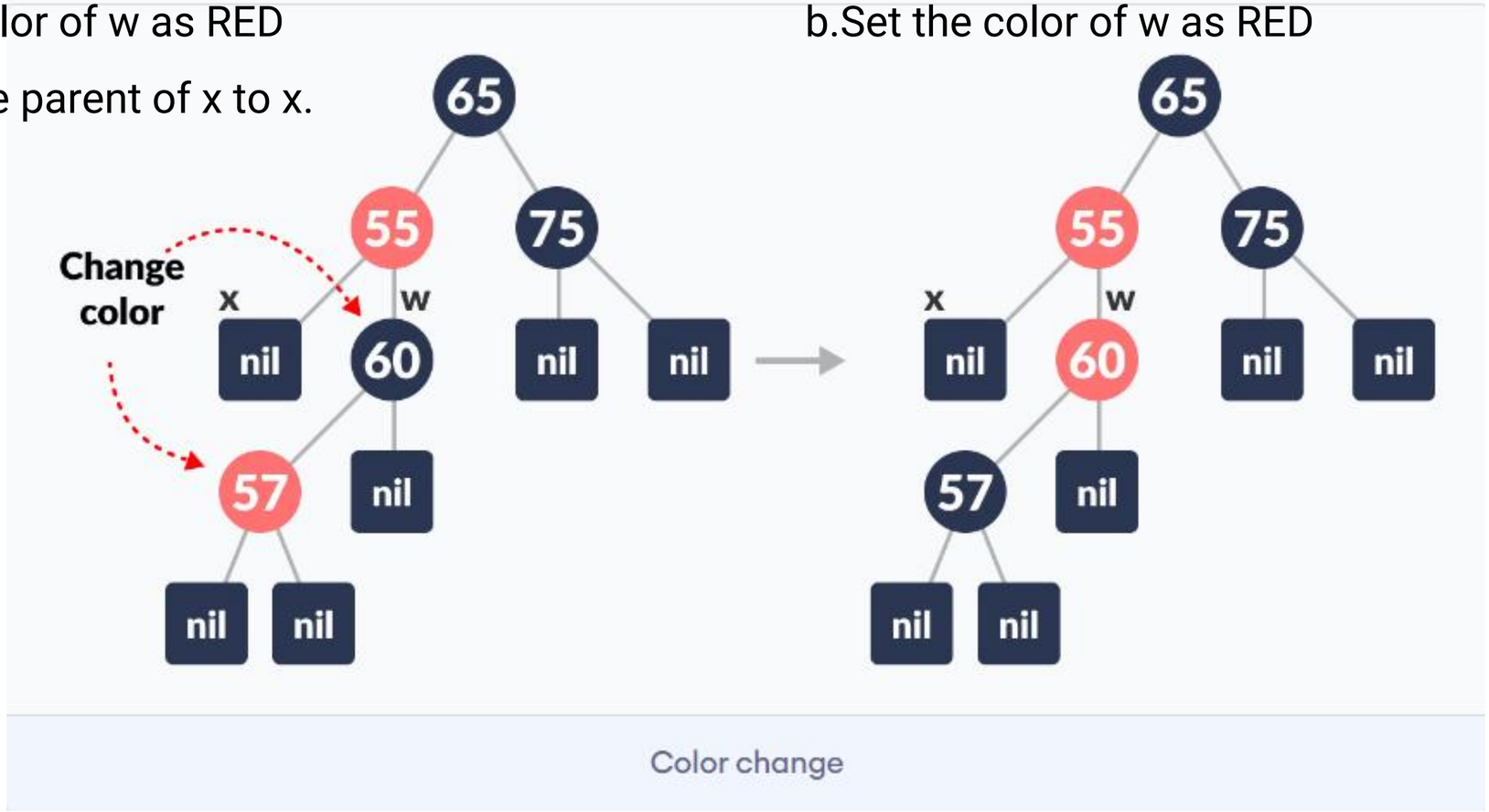
- Set the color of w as RED
- Assign the parent of x to x.

d. Else if the color of the rightChild of w is BLACK

Case-III:

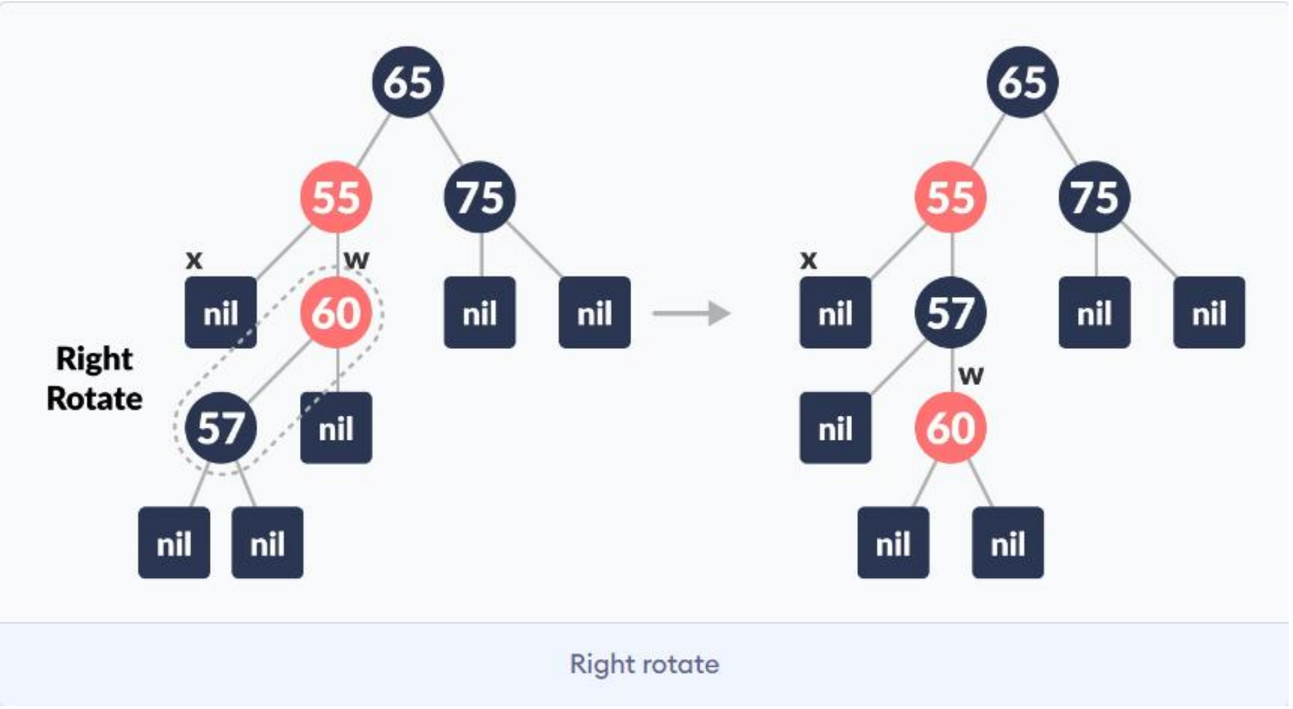
a. Set the color of the leftChild of w as BLACK

b.Set the color of w as RED



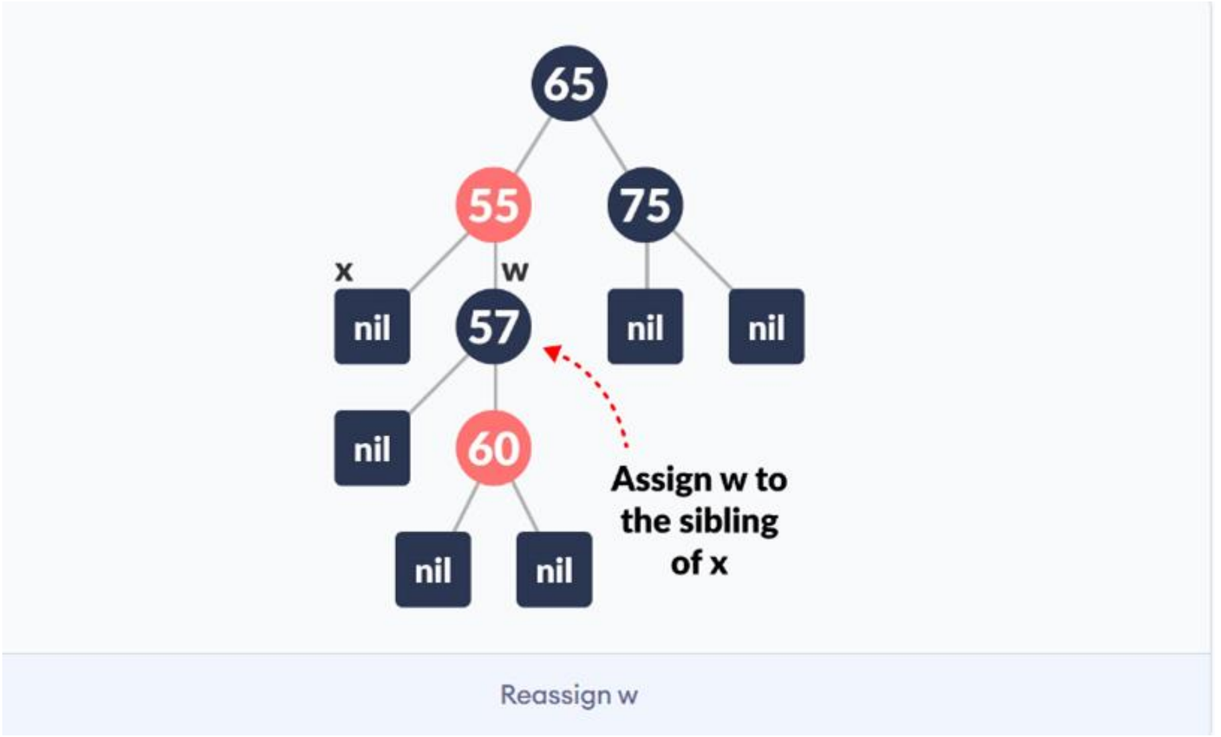
Algorithm to maintain Red-Black property after deletion

c. Right-Rotate w.



Right rotate

d. Assign the rightChild of the parent of x to w.



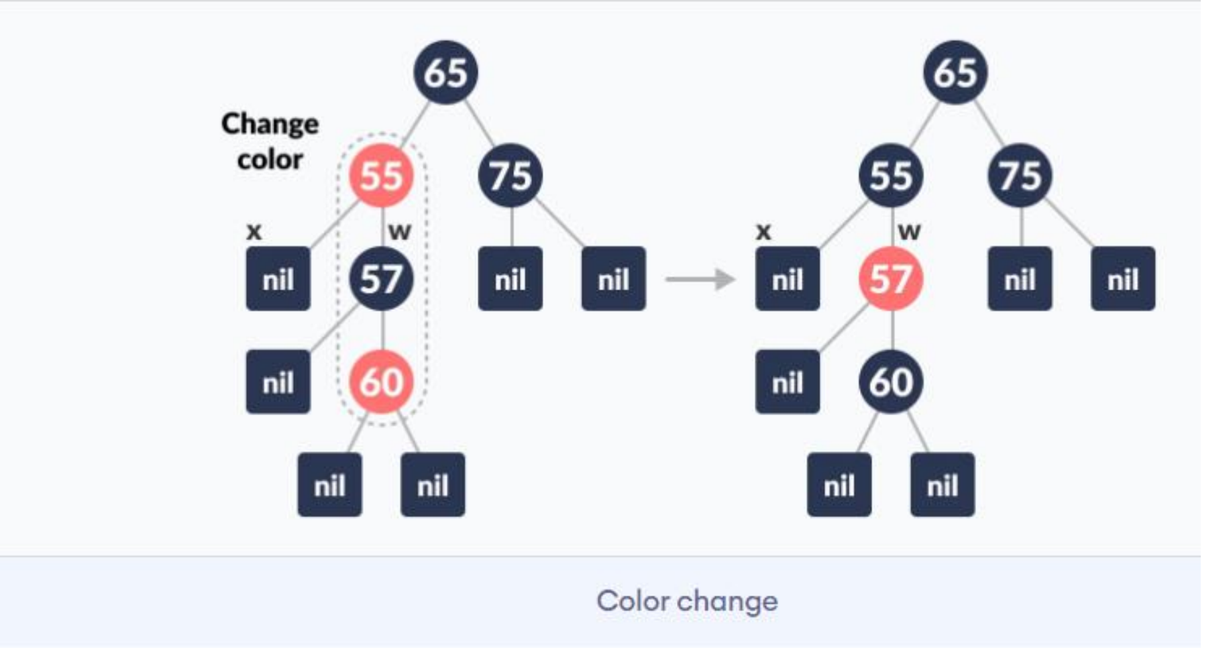
Reassign w

Algorithm to maintain Red-Black property after deletion

e. If any of the above cases do not occur, then do the following.

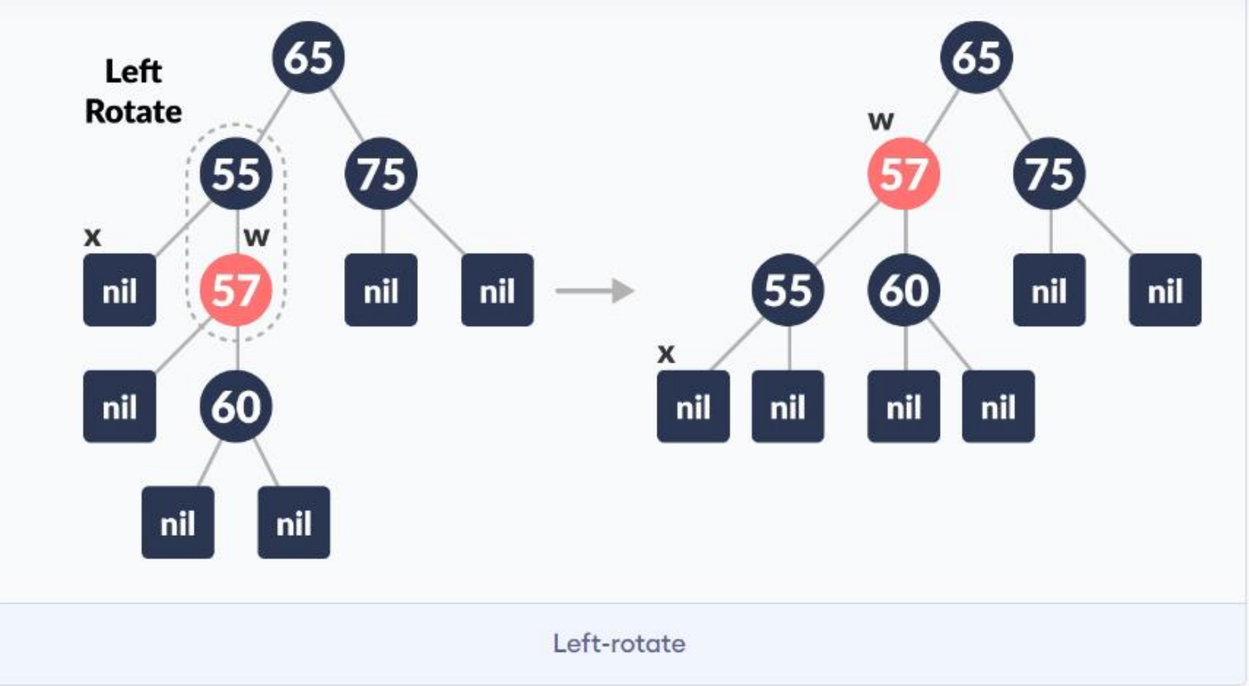
Case-IV:

- Set the color of w as the color of the parent of x.
- Set the color of the parent of parent of x as BLACK.
- Set the color of the right child of w as BLACK.



Color change

d. Left-Rotate the parent of x.



Left-rotate

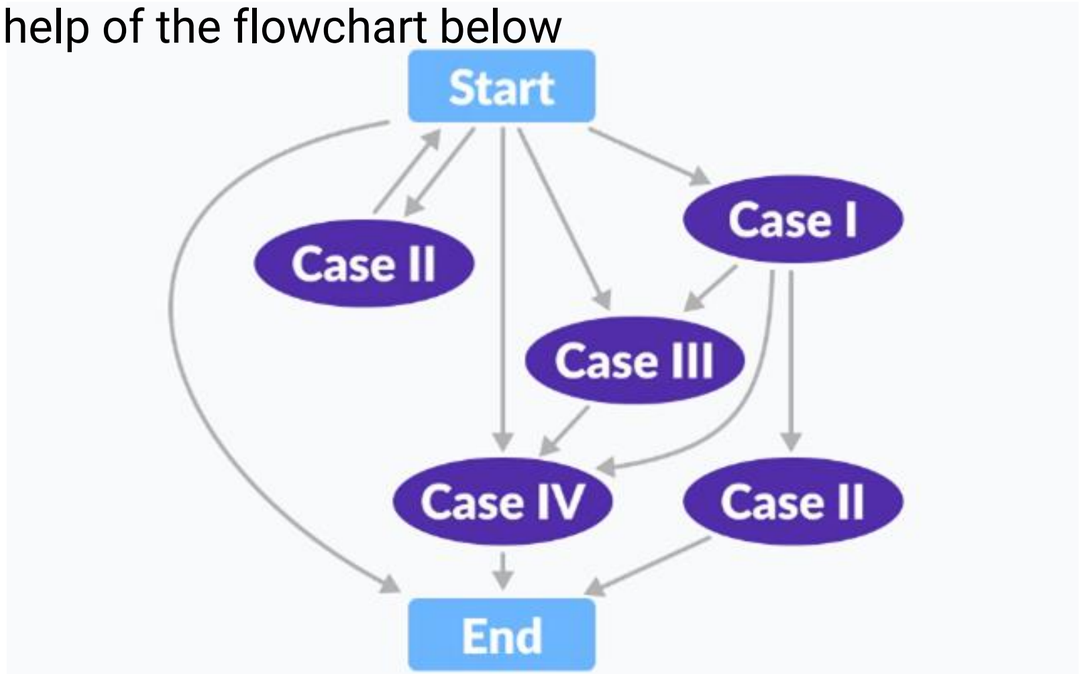
Algorithm to maintain Red-Black property after deletion

e. Set x as the root of the tree.

Set x as root

- Else same as above with right changed to left and vice versa.
- Set the color of x as BLACK.

The workflow of the above cases can be understood with the help of the flowchart below



Flowchart for deletion operation

THANK YOU!