Tree data structures

Introduction

Tree Data Structure

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

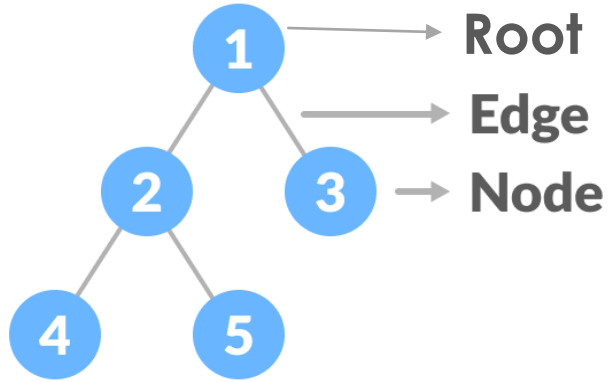
Tree Terminologies

Node

A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.

The node having at least a child node is called an **internal node**.



Edge

It is the link between any two nodes.

Root

It is the topmost node of a tree.

Degree of a Node

The degree of a node is the total number of branches of that node.

Types of Tree

- BINARY TREE
- BINARY SEARCH TREE
- B-TREE
- B+ TREE
- RED-BLACK TREE

Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Tree Traversal - inorder, preorder and postorder

- Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.
- Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.
- Every tree is a combination of
  - A node carrying data
  - Two subtrees
- Remember that our goal is to visit each node, so we need tovisit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.Depending on the order in which we do this,there can be three types of traversal.

Inorder traversal

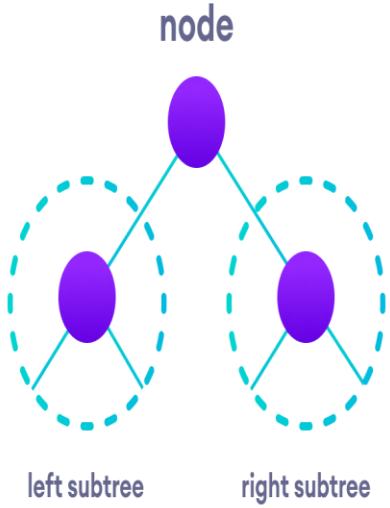
1.First, visit all the nodes in the left subtree
2.Then the root node
3.Visit all the nodes in the right subtree

Preorder traversal

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

Postorder traversal

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



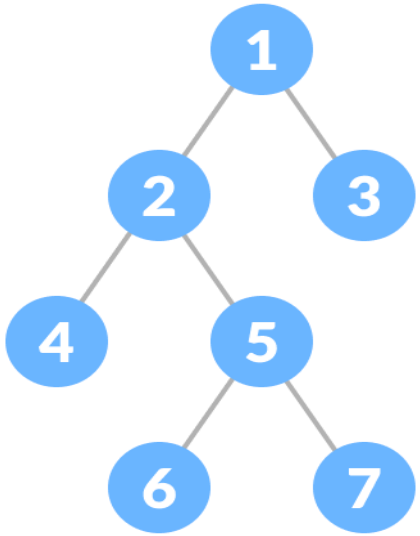
Binary Tree

- A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:
- data item
- address of left child
- address of right child

Types of Binary Tree

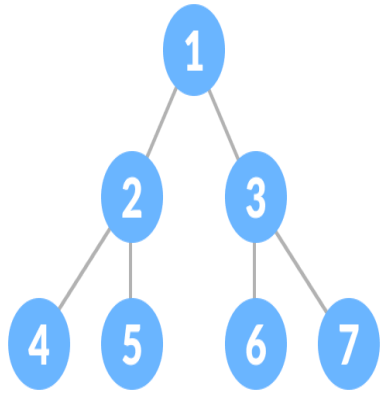
1. Full Binary Tree

- A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



2. Perfect Binary Tree

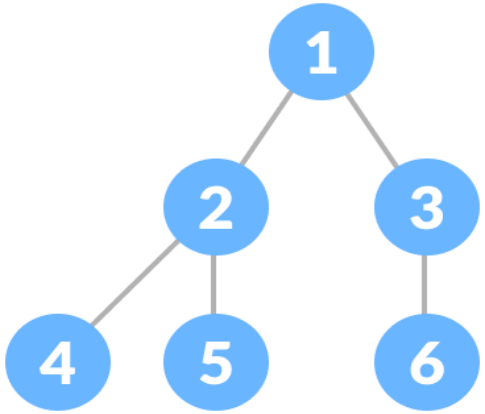
- A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

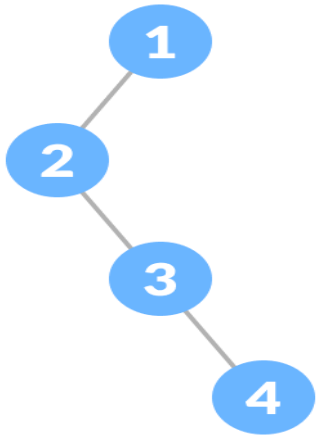
1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



4. Degenerate or Pathological Tree

- A degenerate or pathological tree is the tree having

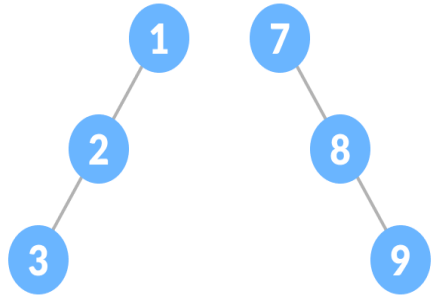
a single child either left or right.



5. Skewed Binary Tree

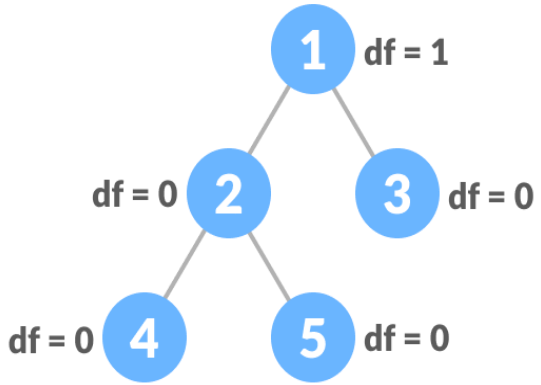
- A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree:

  left-skewed binary tree and **right-skewed binary tree**.



6. Balanced Binary Tree

- It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



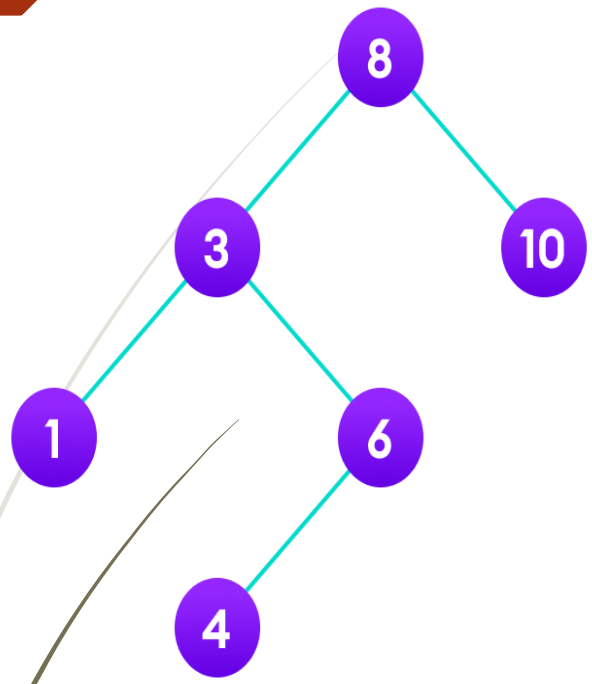
Binary Tree Applications

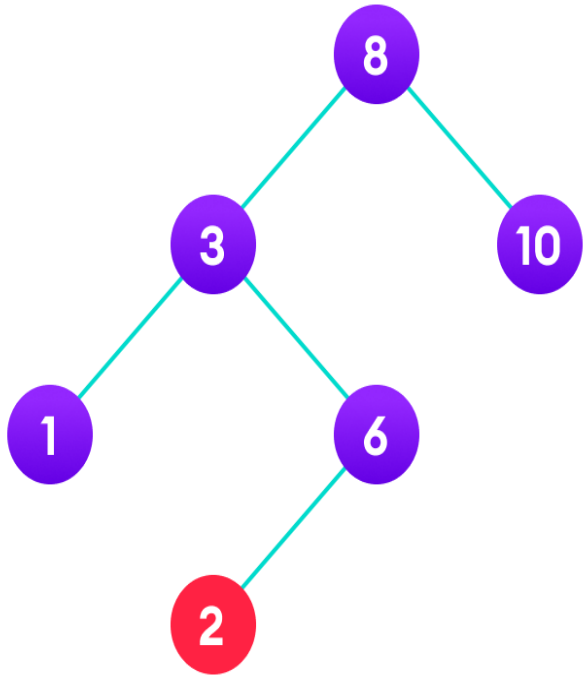
- For easy and quick access to data
- In router algorithms
- To implement heap data structure.
- Syntax tree

Binary Search Tree(BST)

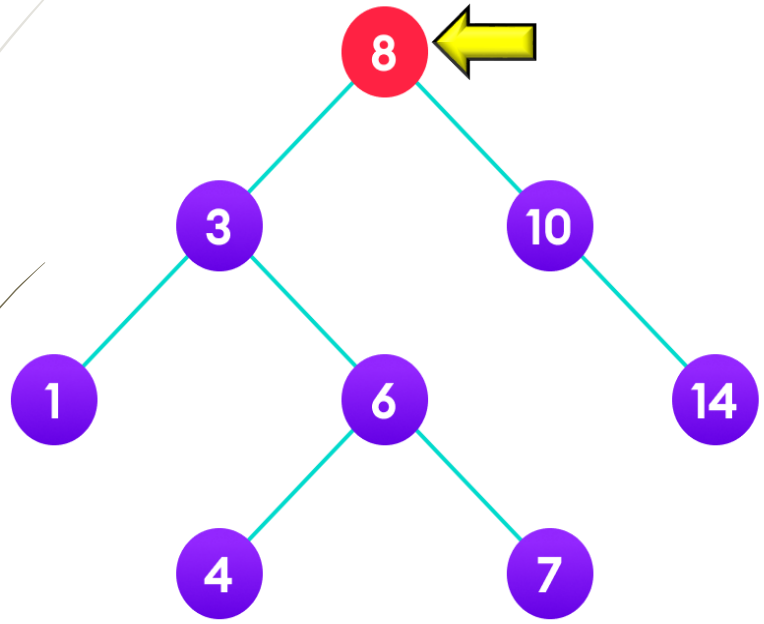
- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log n) time.
- The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties





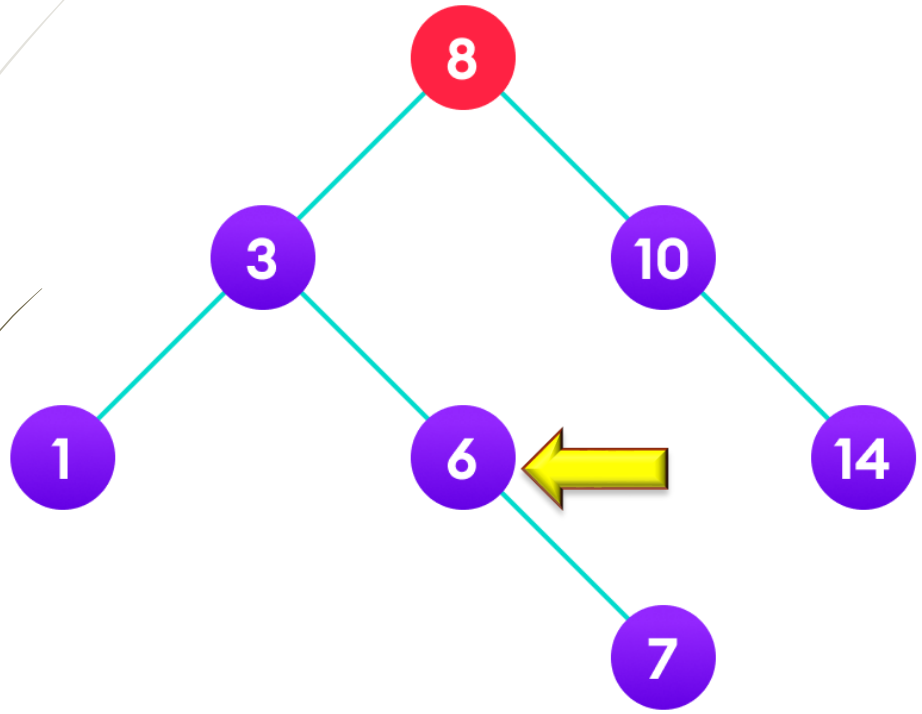
The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.



Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

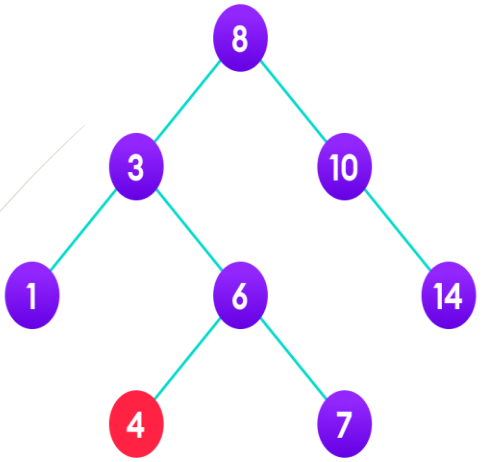
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

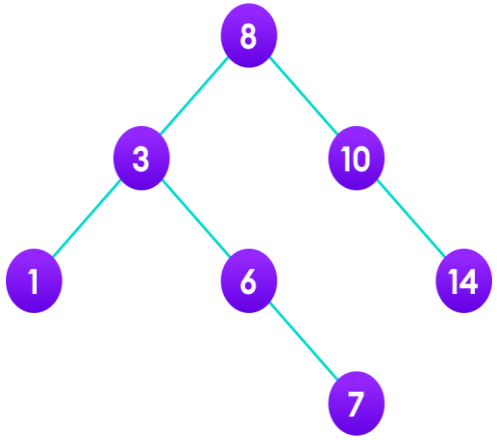


Insert Operation

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.





Deletion Operation

There are three cases for deleting a node from a binary search tree.

Case I

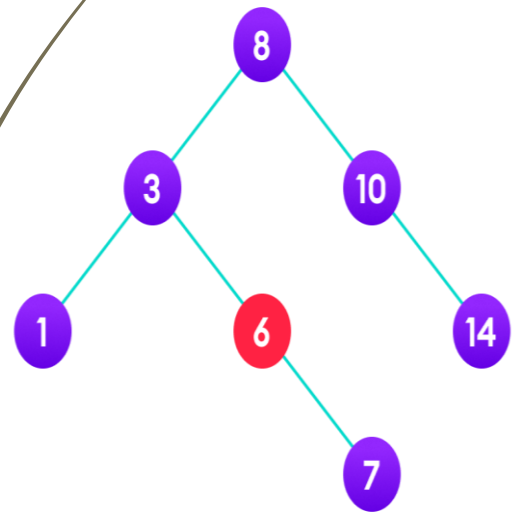
In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

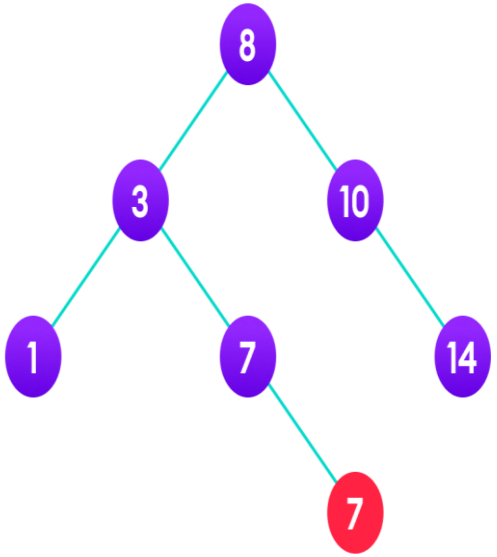
Case II

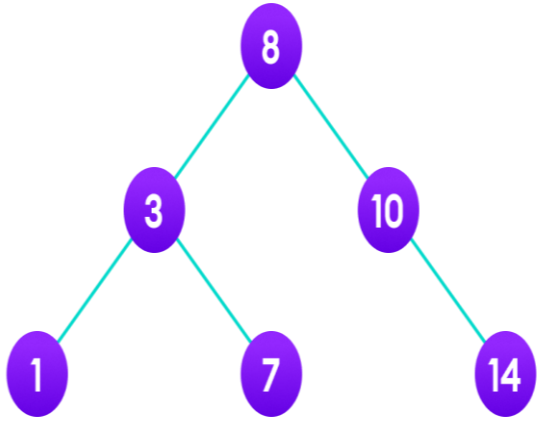
In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

Replace that node with its child node.

Remove the child node from its original position.







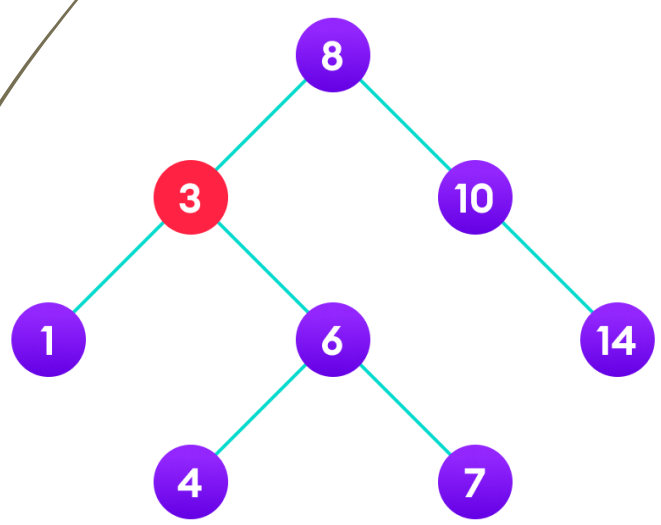
Case III

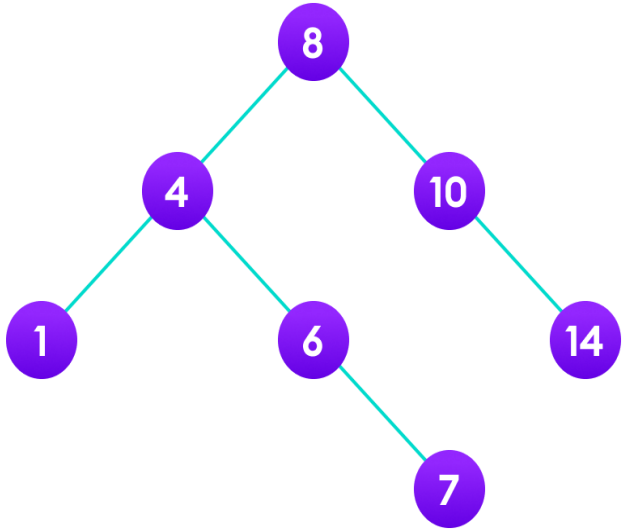
In the third case, the node to be deleted has two children. In such a case follow the steps below:

Get the inorder successor of that node.

Replace the node with the inorder successor.

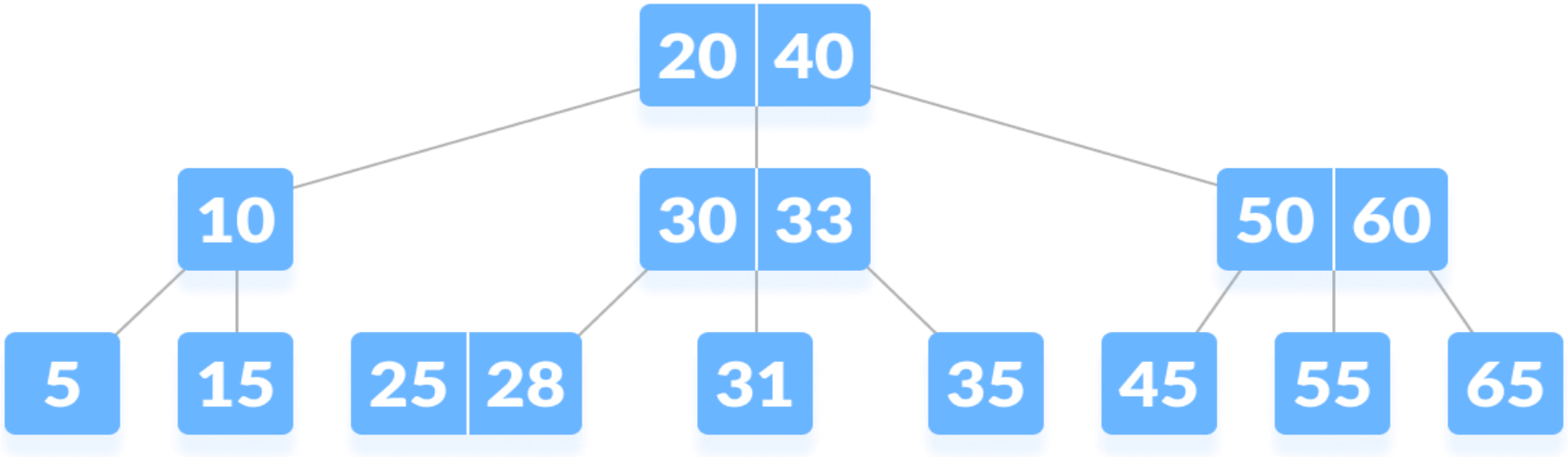
Remove the inorder successor from its original position.





B-tree

- B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.



B tree

A B tree of order m (m = 3)contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m(3) children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 (3/2 = 2)children.
3. A node can contain a maximum of m-1 keys i.e 2
4. A node (except root node) should contain a minimum of (m/2)-1 keys i.e 1 key
5. The root nodes must have at least 2 nodes.
6. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 (2) number of nodes.

- Order m

Maximum m nodes

Minimum m/2 nodes(children)

Keys

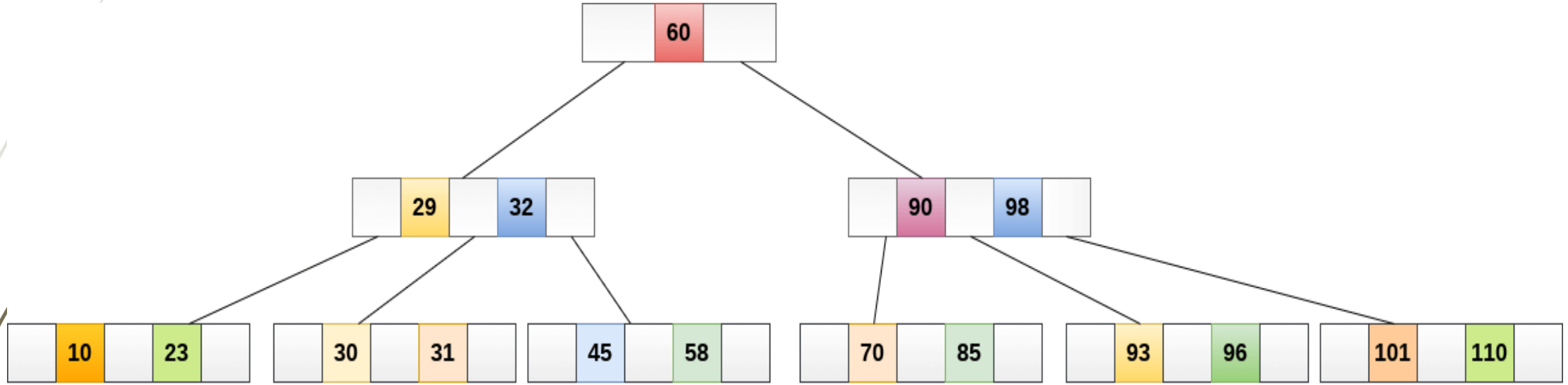
Maximum m-1 key

Minimum m/2 – 1 key

B-tree Properties

1. All leaves are at the same level.
2. The root has at least 2 children and contains a minimum of 1 key.
3. Each node except root can have at most n children and at least n/2 children.
4. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
5. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
6. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).
7. Insertion of a Node in B-Tree happens only at Leaf Node.

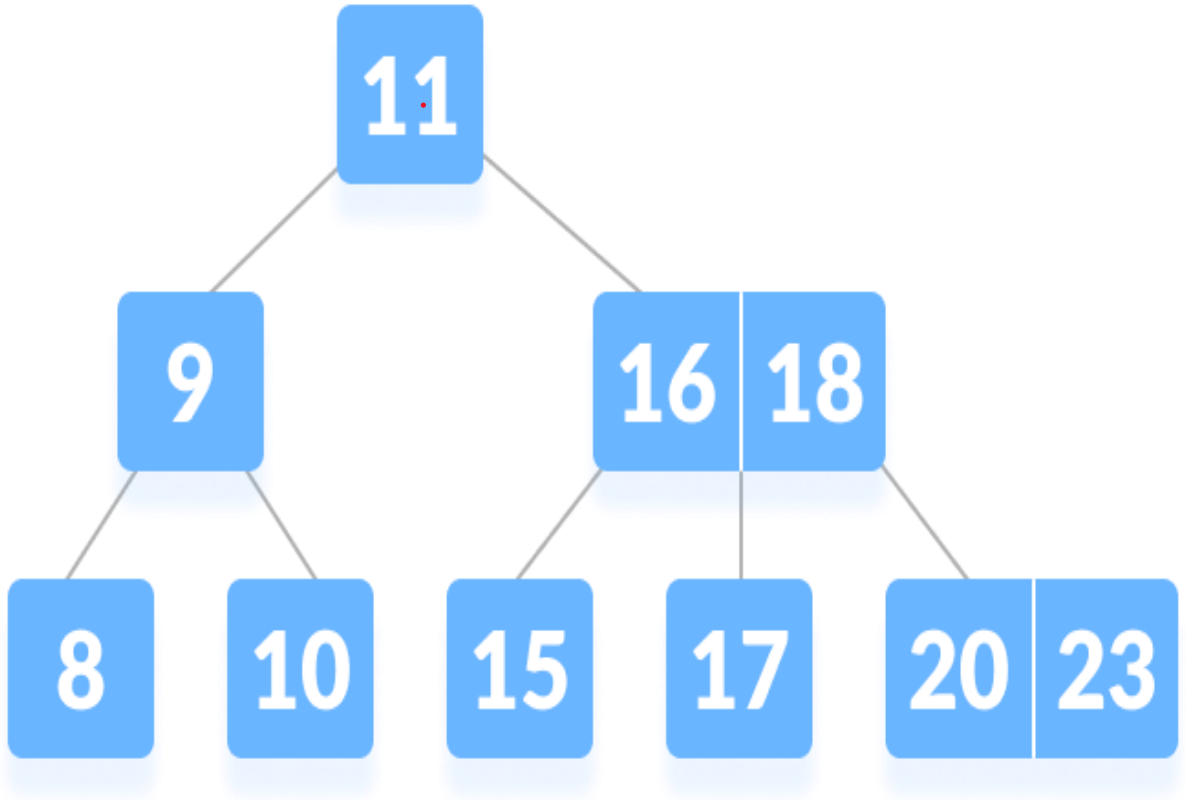
A B tree of order 3 is shown in the following image.



Create node in b-tree

```
# Create a node
class BTreeNode:
    def __init__(self, leaf=False):
        self.leaf = leaf
        self.keys = []
        self.child = []
# Tree
class BTree:
    def __init__(self, t):
        self.root = BTreeNode(True)
        self.t = t
```

Searching an element in a B-tree



Searching in B tree

Searching

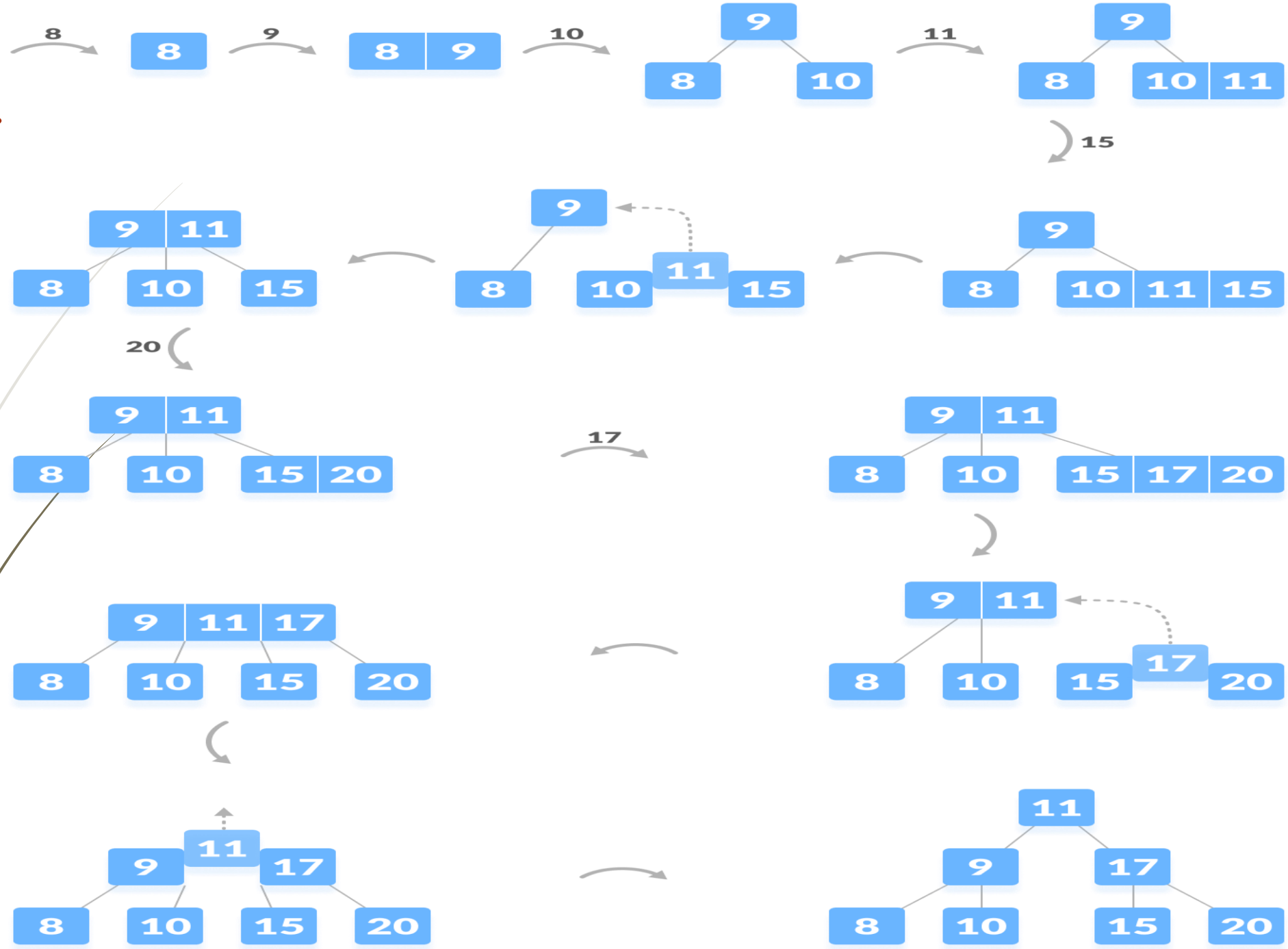
- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with first key value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that key value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7 -** If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion into a B-tree

- Inserting an element on a B-tree consists of two events: **searching the appropriate node** to insert the element and **splitting the node** if required. Insertion operation always takes place in the bottom-up approach.

- **Insertion Operation**

1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.


8
8
9
8 9
10
9
8 10
11
9
8 10 11
15
9
8 10 11 15
9
8 10 11 15
9 11
8 10 15
20
9 11
8 10 15 20
17
9 11
8 10 15 17 20
9 11 17
8 10 15 20
9 11
8 10 15 17 20
11
9 17
8 10 15 20
11
9 17
8 10 15 20

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

insert(1)

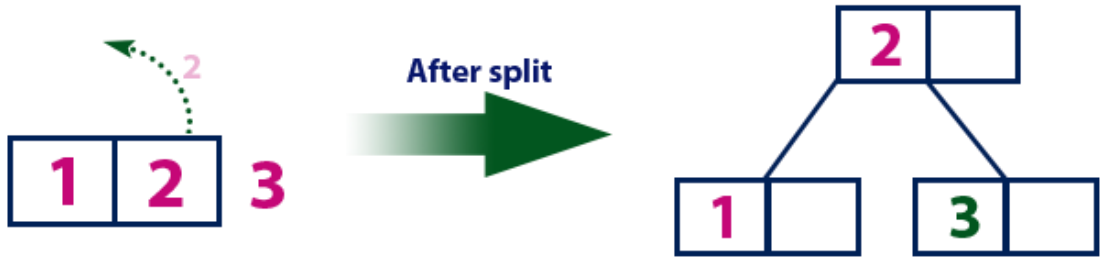
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

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.

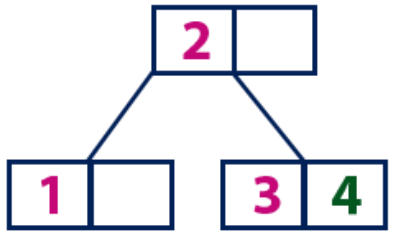
insert(3)

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.



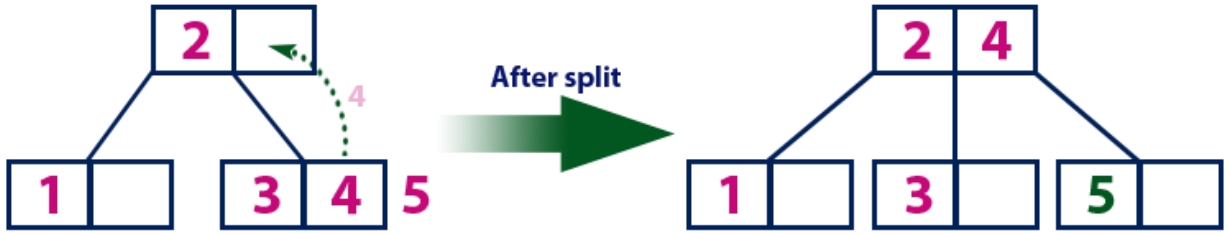
insert(4)

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.



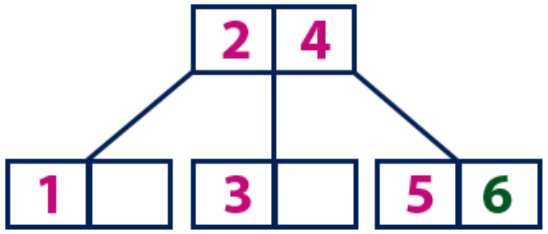
insert(5)

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.



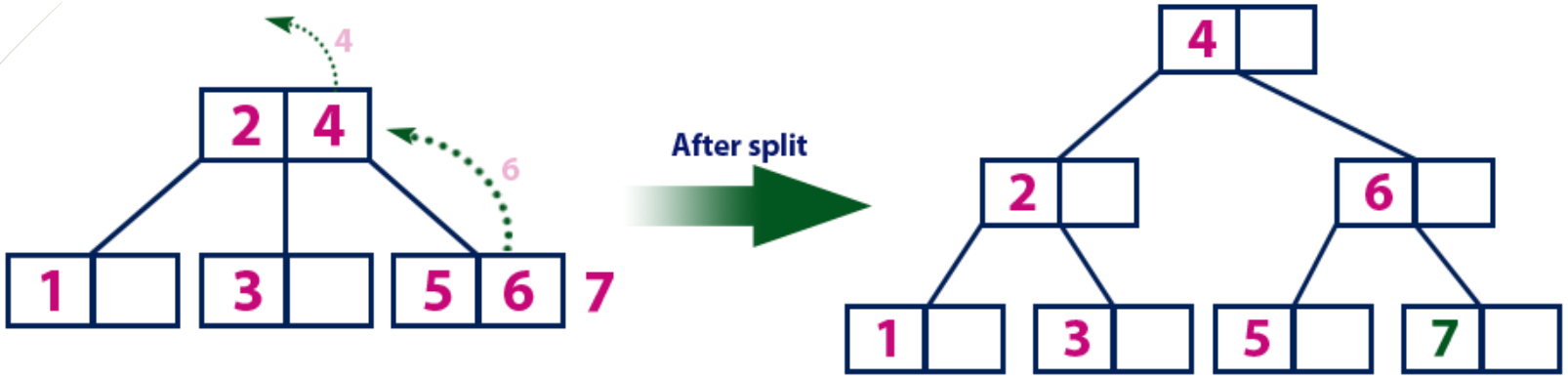
insert(6)

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.



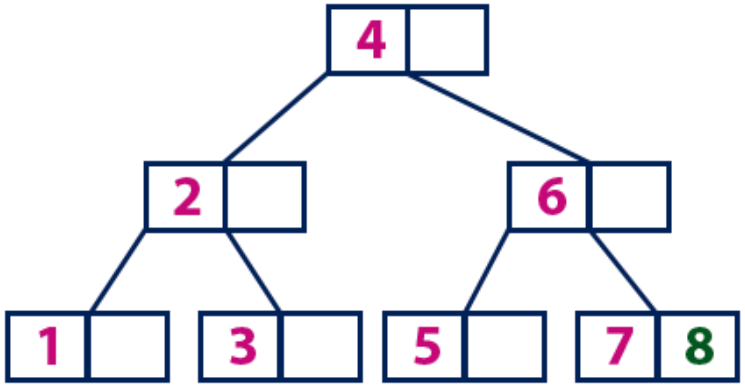
insert(7)

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.



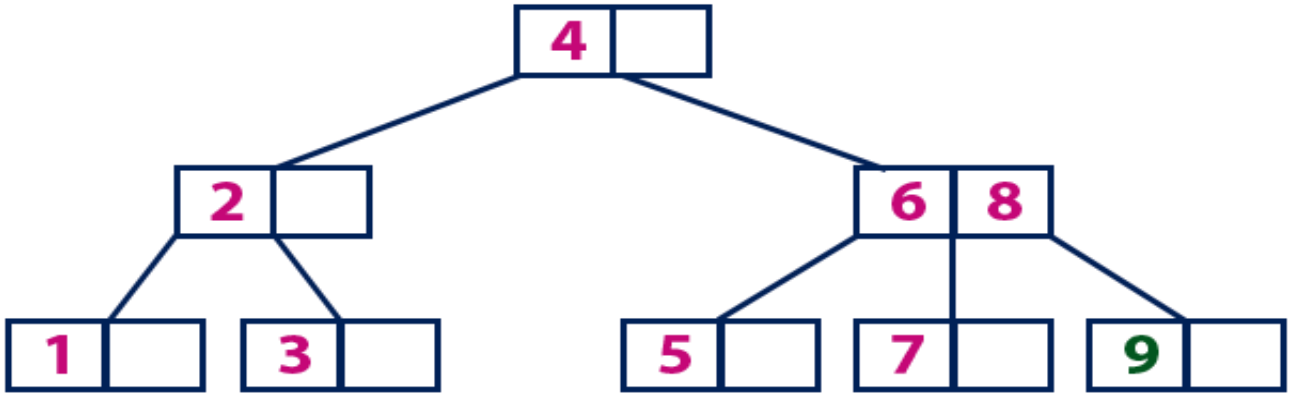
insert(8)

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.



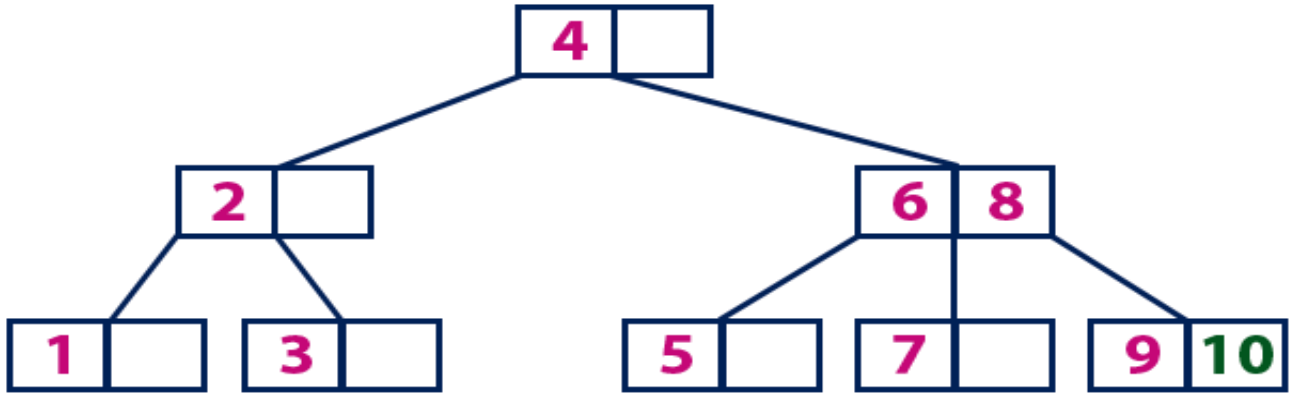
insert(9)

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:

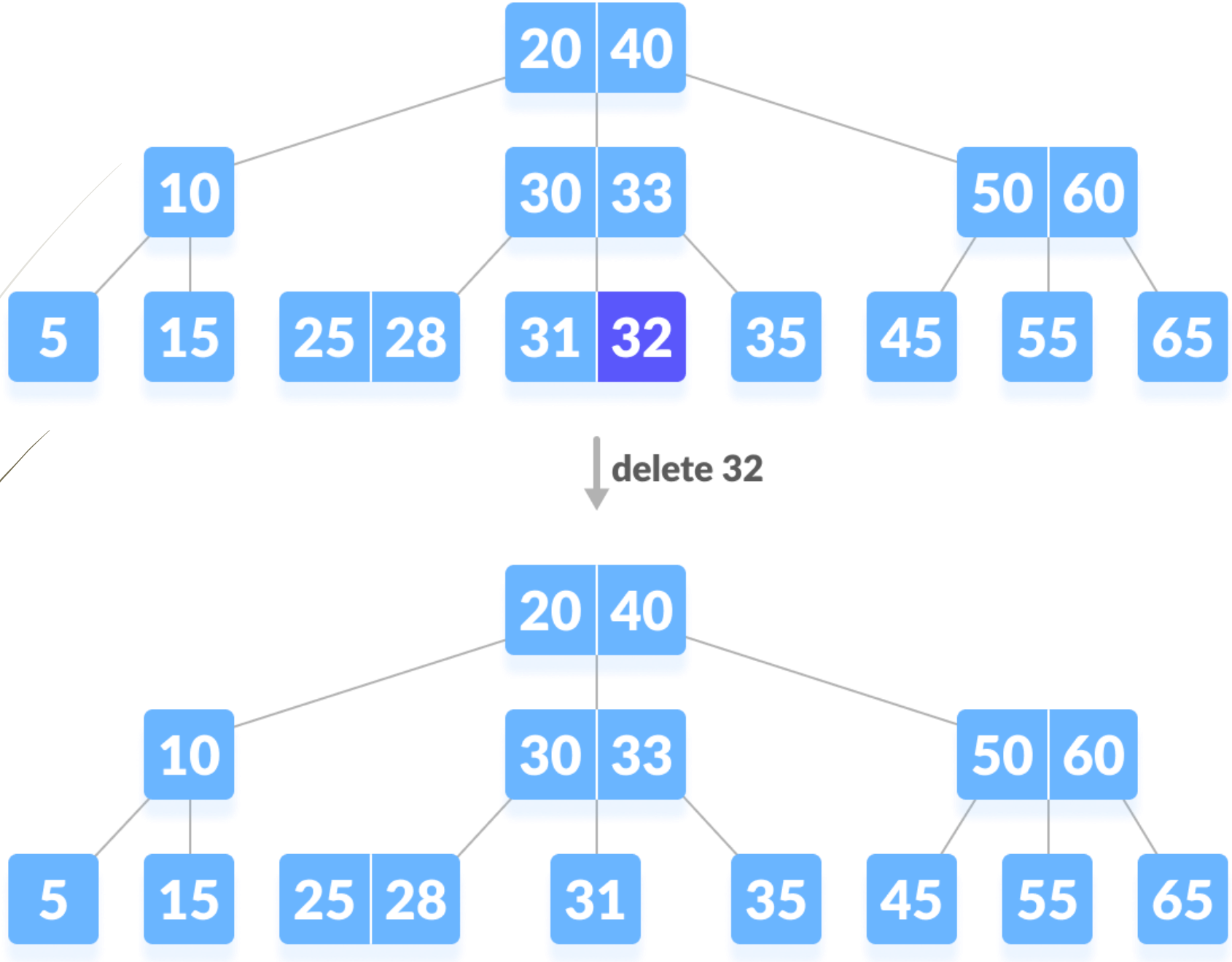
1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

- There are three main cases for deletion operation in a B tree.
- **Case I**
- The key to be deleted lies in the leaf. There are two cases for it.

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

   In the tree below, deleting 32 does not violate the above properties.

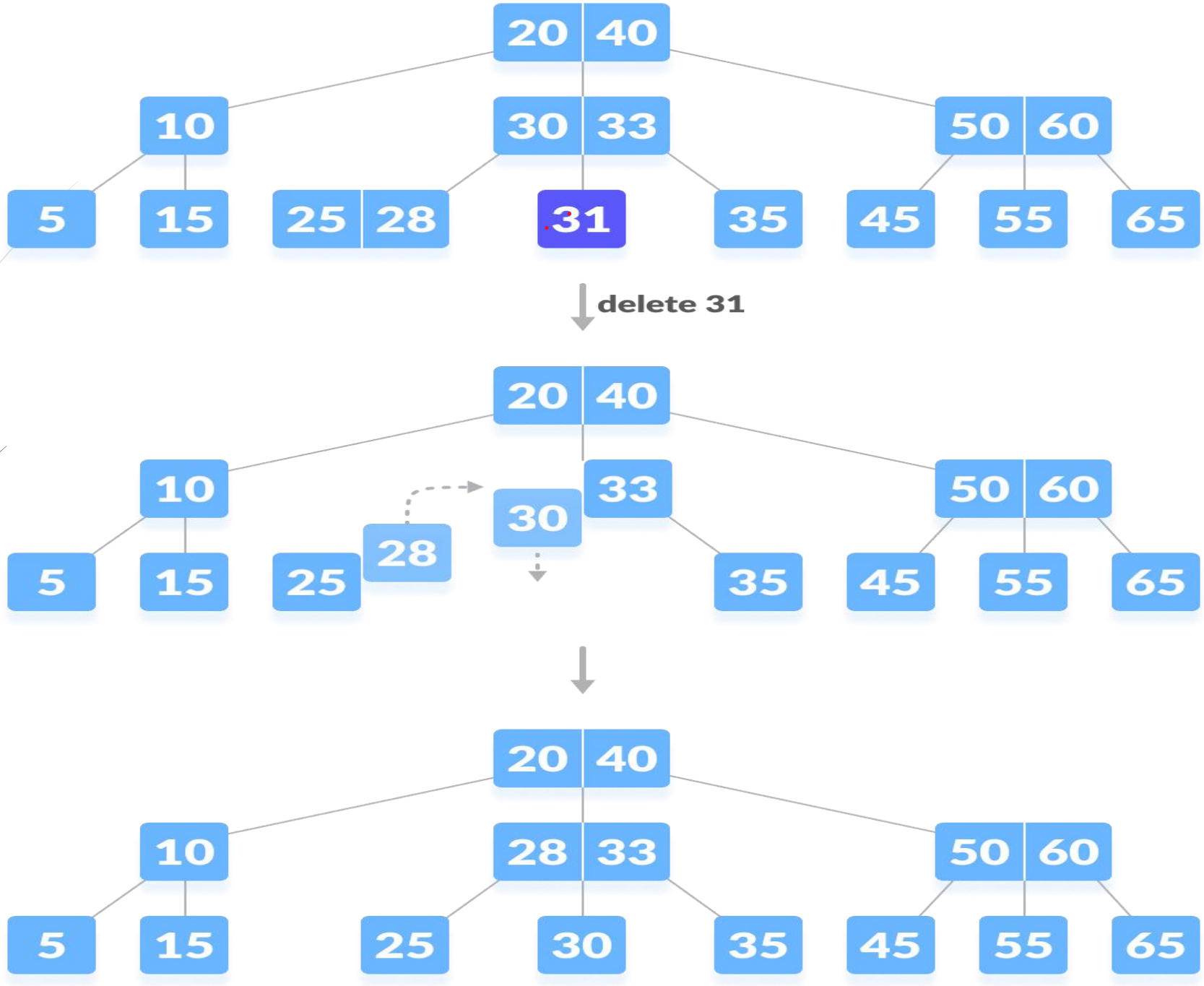

20
40
10
30
33
50
60
5
15
25
28
31
32
35
45
55
65
delete 32
20
40
10
30
33
50
60
5
15
25
28
31
35
45
55
65

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 condition. Let us borrow a key from the left sibling node.


20 40
10
30 33
50 60
5
15
25 28
31
35
45
55
65
delete 31
20 40
10
30
33
50 60
28
5
15
25
35
45
55
65
20 40
10
28 33
50 60
5
15
25
30
35
45
55
65

Case II

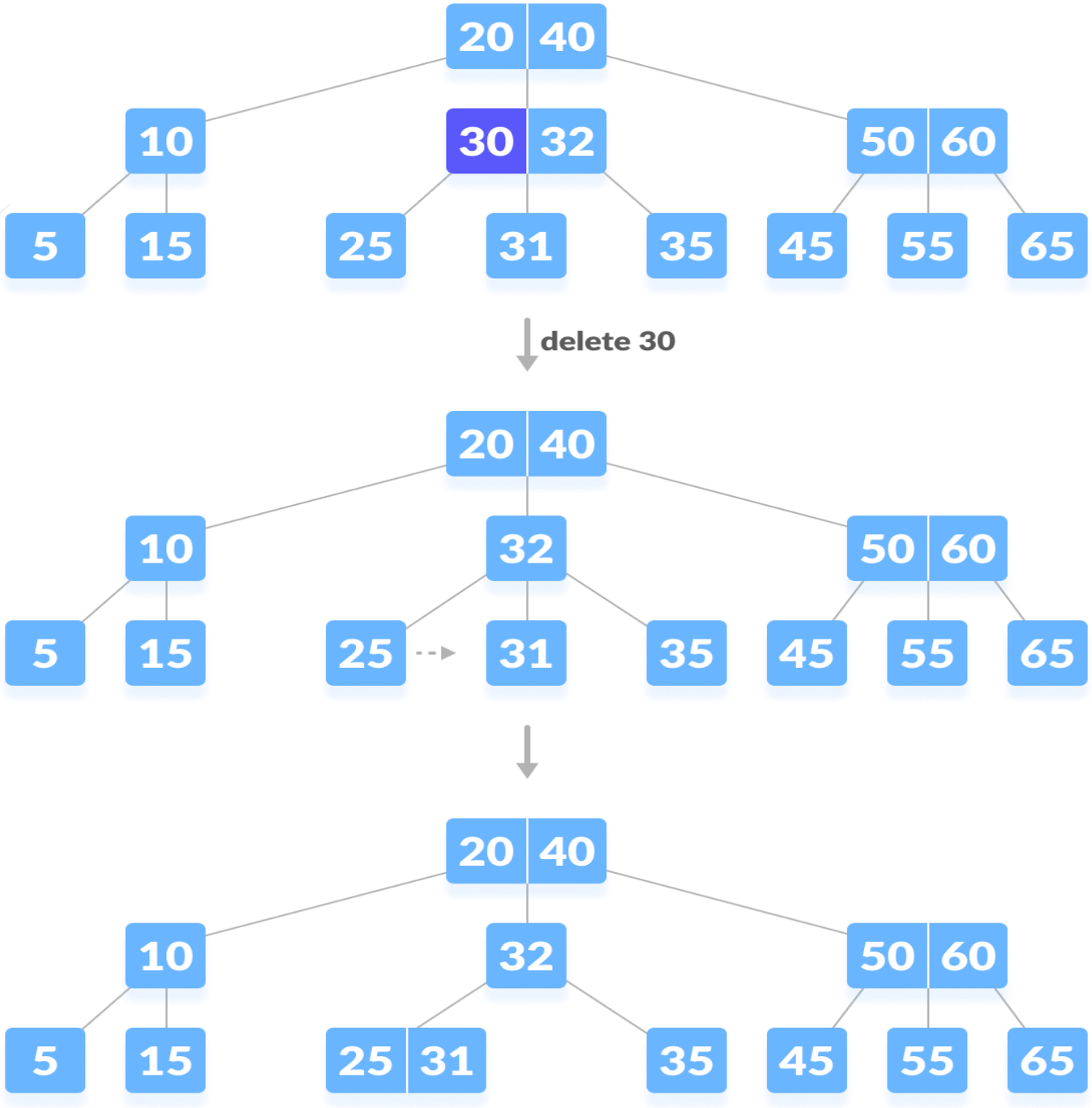
If the key to be deleted lies in the internal node, the following cases occur.

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

2.The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.

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

After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.


20 40
10
30 32
50 60
5
15
25
31
35
45
55
65
delete 30
20 40
10
32
50 60
5
15
25
31
35
45
55
65
20 40
10
32
50 60
5
15
25 31
35
45
55
65

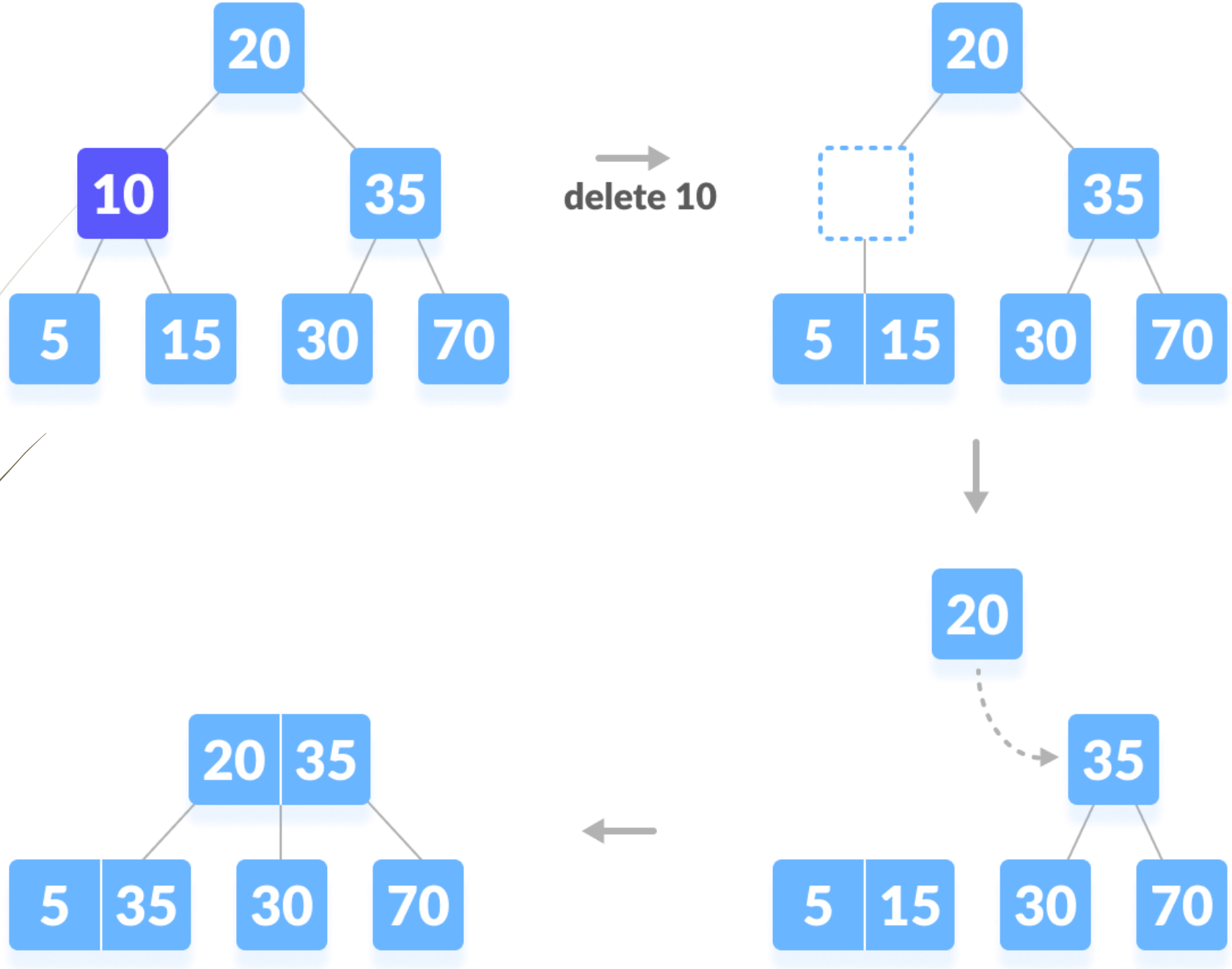
Case III

In this case, the height of the tree shrinks.

If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor.

If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(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).


20
10
35
5
15
30
70
delete 10
20
35
5
15
30
70
20
35
5
15
30
70
20
35
5
35
30
70

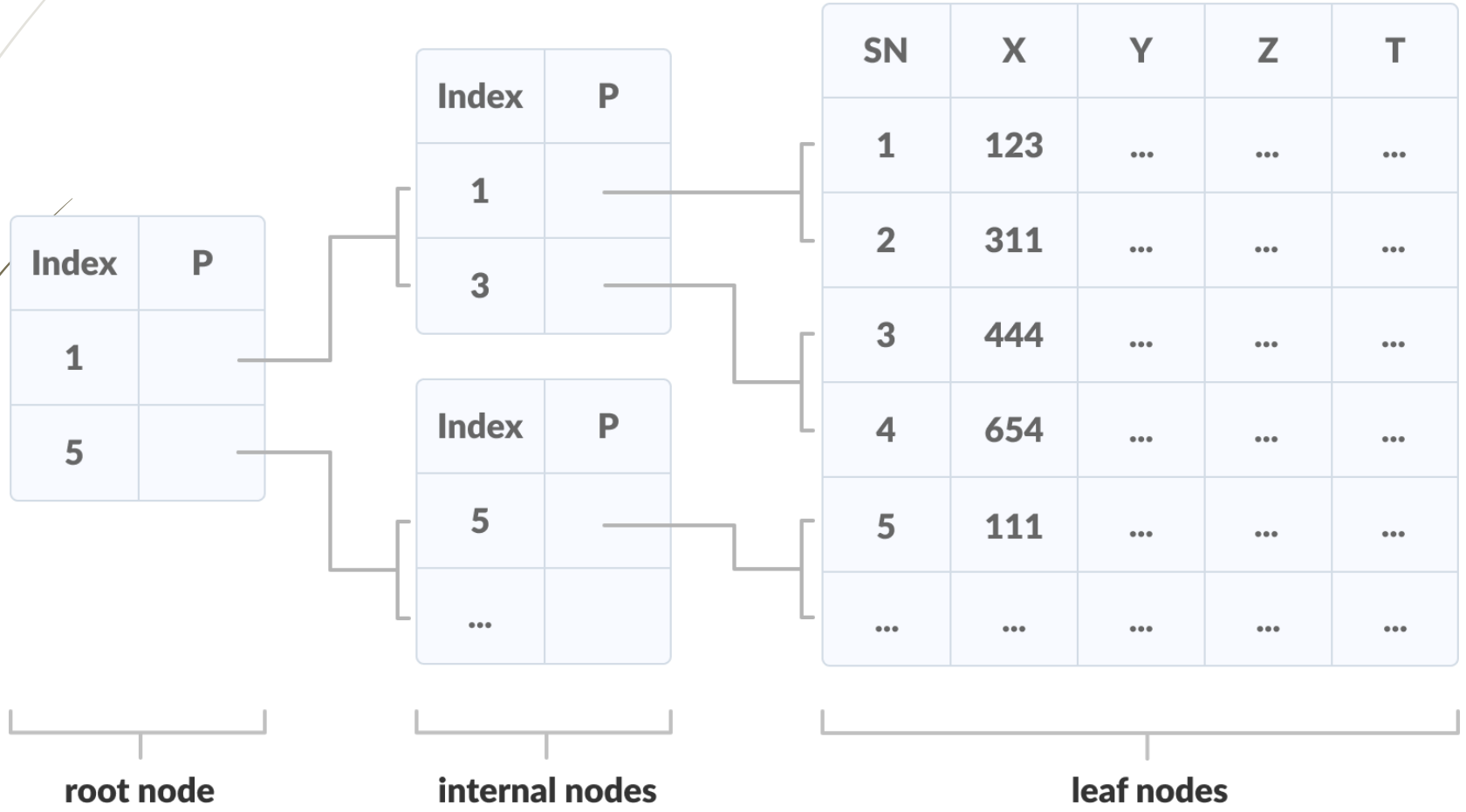
B+ Tree

- In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawbac of B-tree used for indexing, however is that it stores the data pointer (a pointer to the disk file block containin the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. Th technique, greatly reduces the number of entries that can be packed into a node of a B-tree, thereb contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record
- B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, th structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree. It ma be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily sto all the key values along with their corresponding data pointers to the disk file block, in order to access then Moreover, the leaf nodes are linked to provide ordered access to the records. The leaf nodes, therefore form th first level of index, with the internal nodes forming the other levels of a multilevel index. Some of the ke values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching o a record.
- From the above discussion it is apparent that a B+ tree, unlike a B-tree has two orders, ‘a’ and ‘b’, one for th internal nodes and the other for the external (or leaf) nodes.

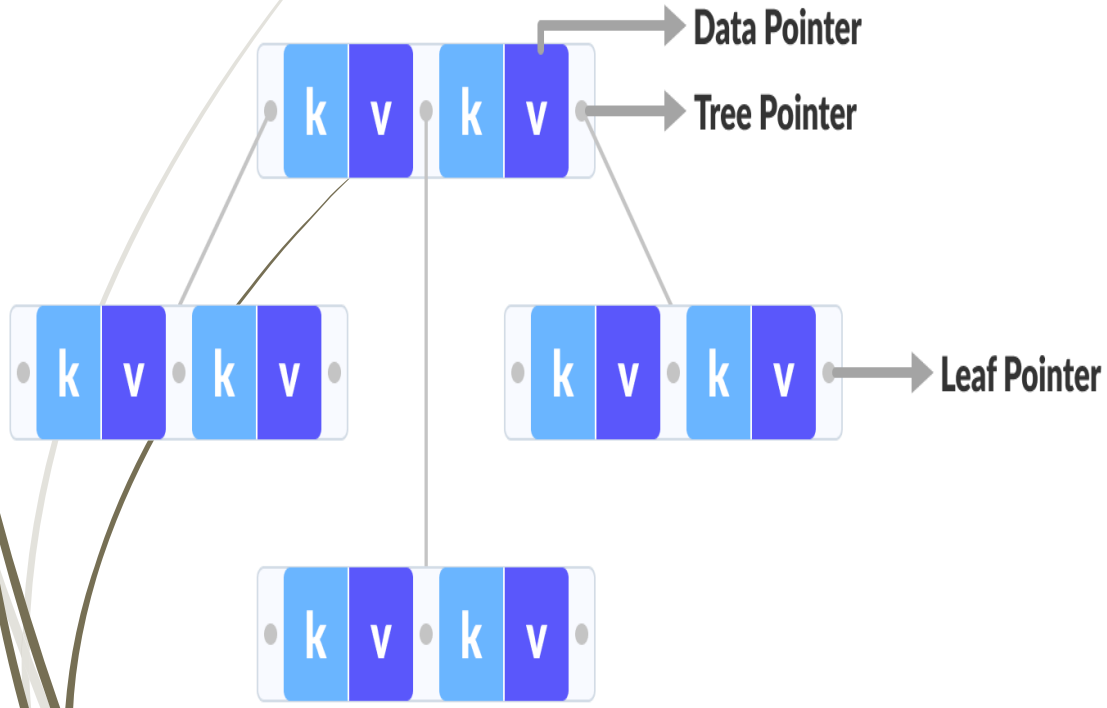
B+ tree Properties

- All leaves are at the same level.
- The root has at least two children.
- Max no keys a node can hold :- (m-1) 7-1 = 6
- Max no children's a node can have :-(m) 7
- Min Numb of keys a node can hold :- c(m/2) -1 = 4-1 = 3
- Min no child :- m/2 = 4

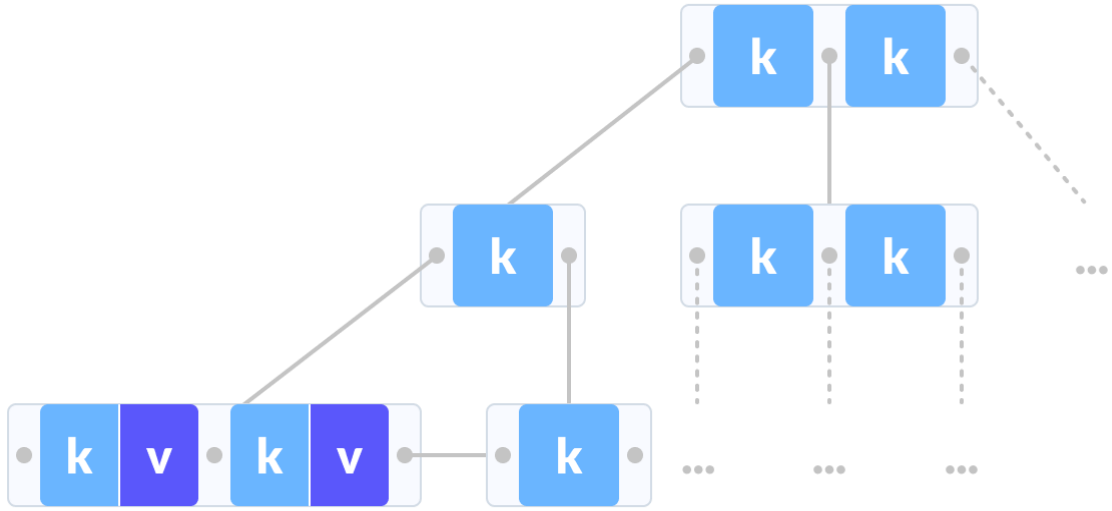
A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level. An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.



Comparison between a B-tree and a B+ Tree

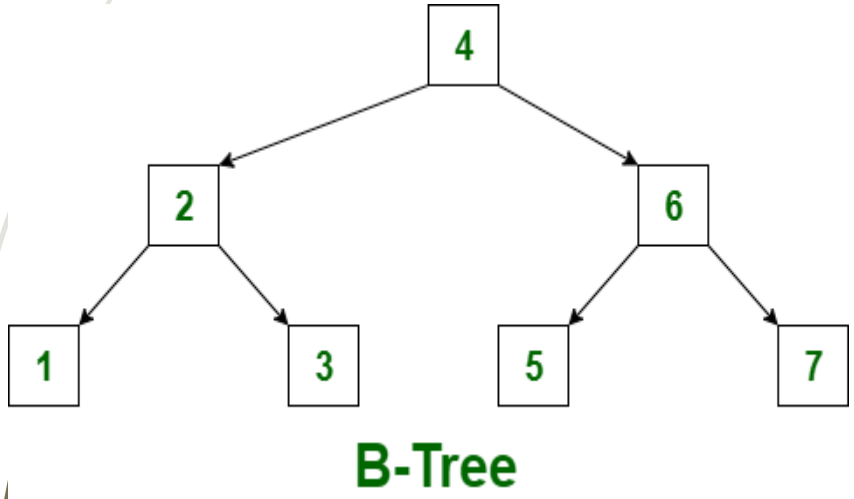


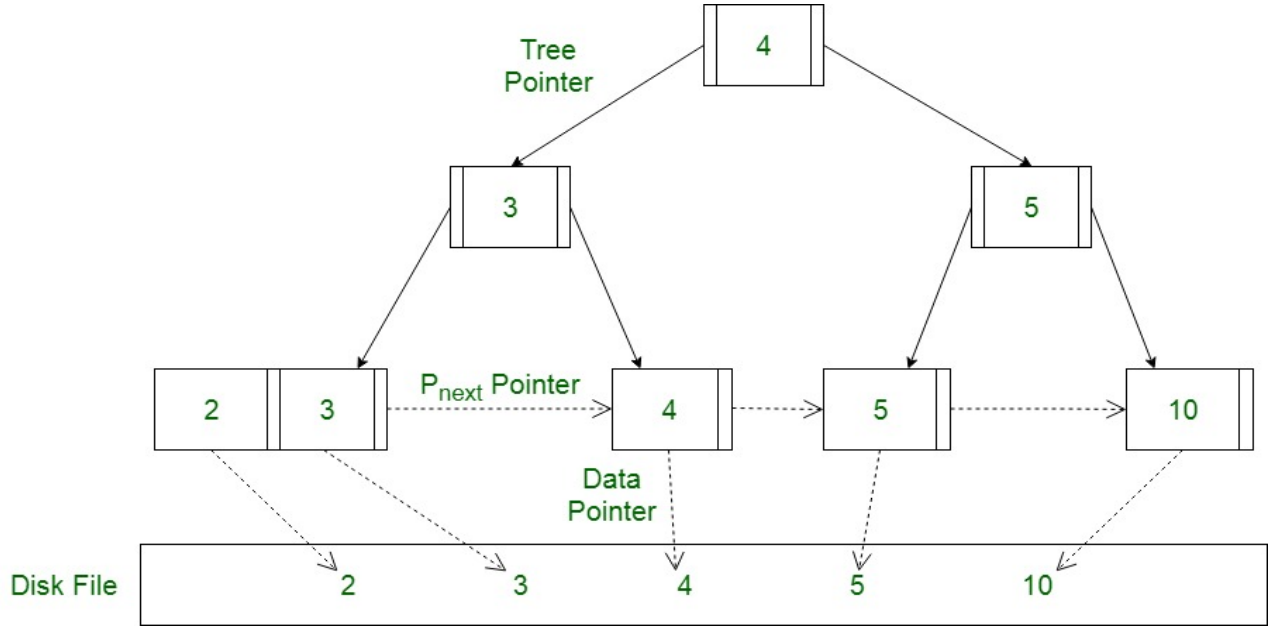
B TREE



B+ TREE

B tree and B+ tree



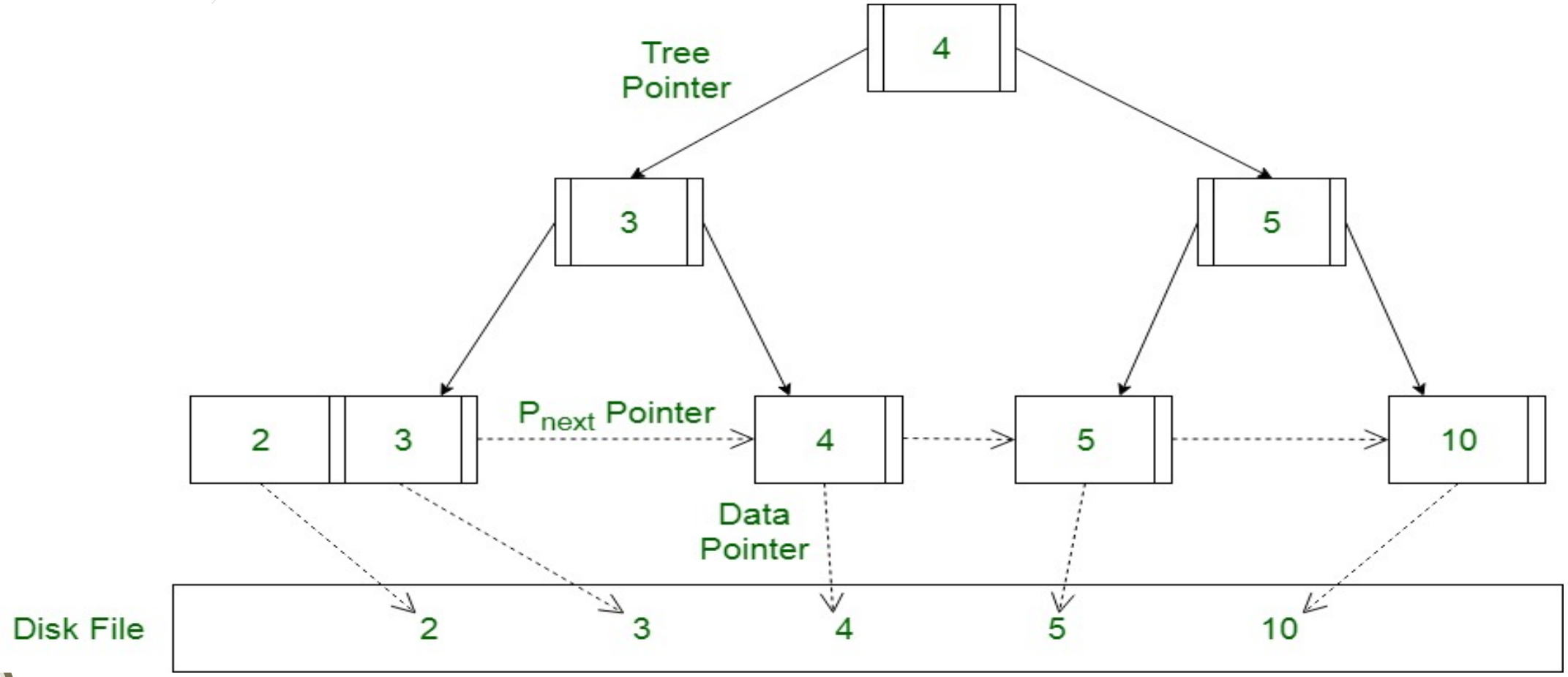


The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.

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.

A Diagram of B+ Tree

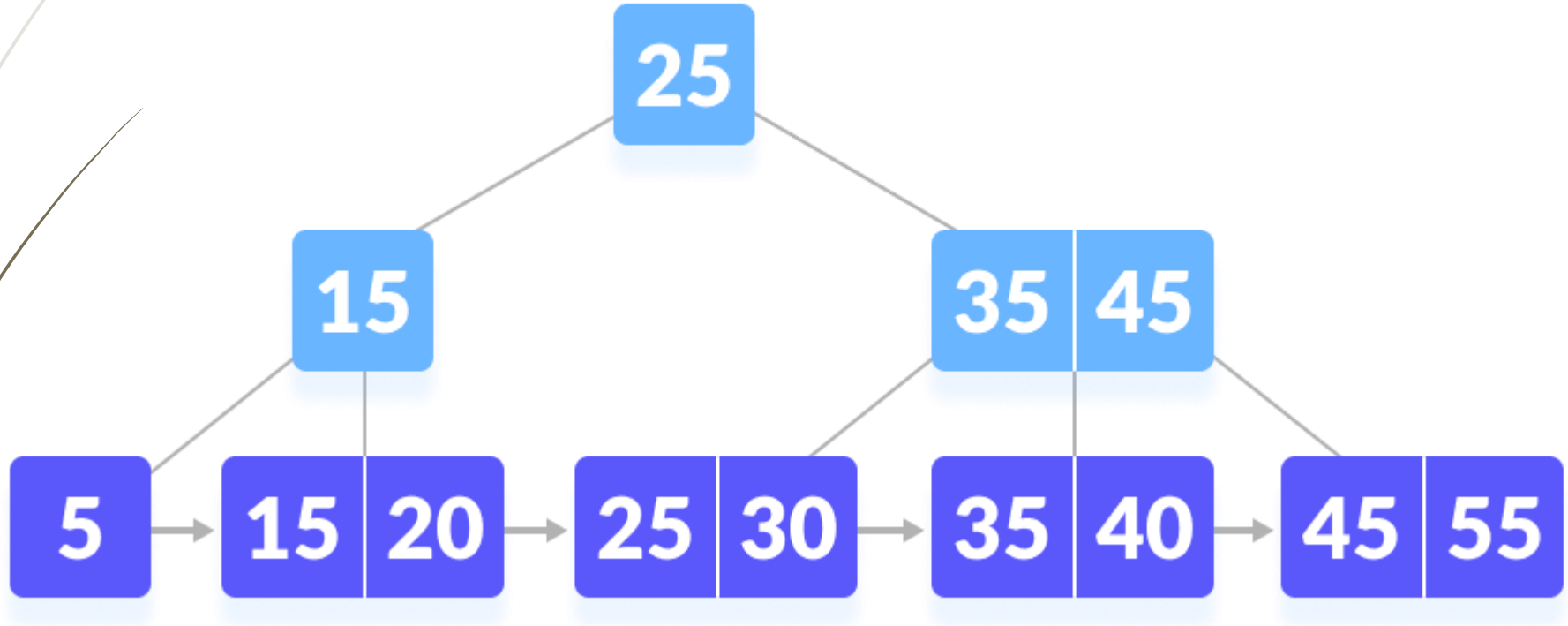


Searching on a B+ Tree

The following steps are followed to search for data in a B+ Tree of order m.
Let the data to be searched be k.

- Start from the root node. Compare k with the keys at the root node [k1, k2, k3,.....km - 1].
- If k < k1, go to the left child of the root node.
- Else if k > k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
- If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.
- Repeat the above steps until a leaf node is reached.
- If k exists in the leaf node, return true else return false.

Let us search k = 45 on the following B+ tree.



Insertion Operation

- The following steps are followed for inserting an element.

1. Since every element is inserted into the leaf node, go to the appropriate leaf node.
2. Insert the key into the leaf node.

- **Case I**

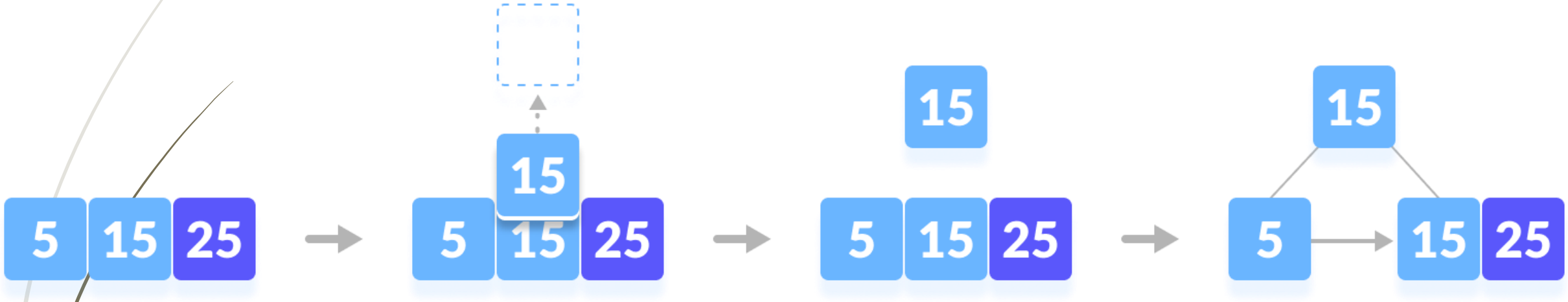
1. If the leaf is not full, insert the key into the leaf node in increasing order.

- **Case II**

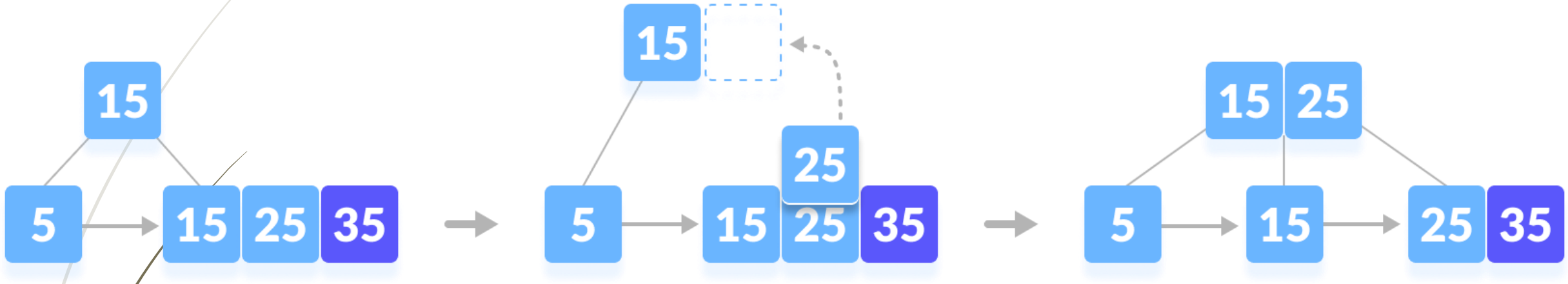
1. If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way.
2. Break the node at m/2th position.
3. Add m/2th key to the parent node as well.
4. If the parent node is already full, follow steps 2 to 3.

INSERT 5

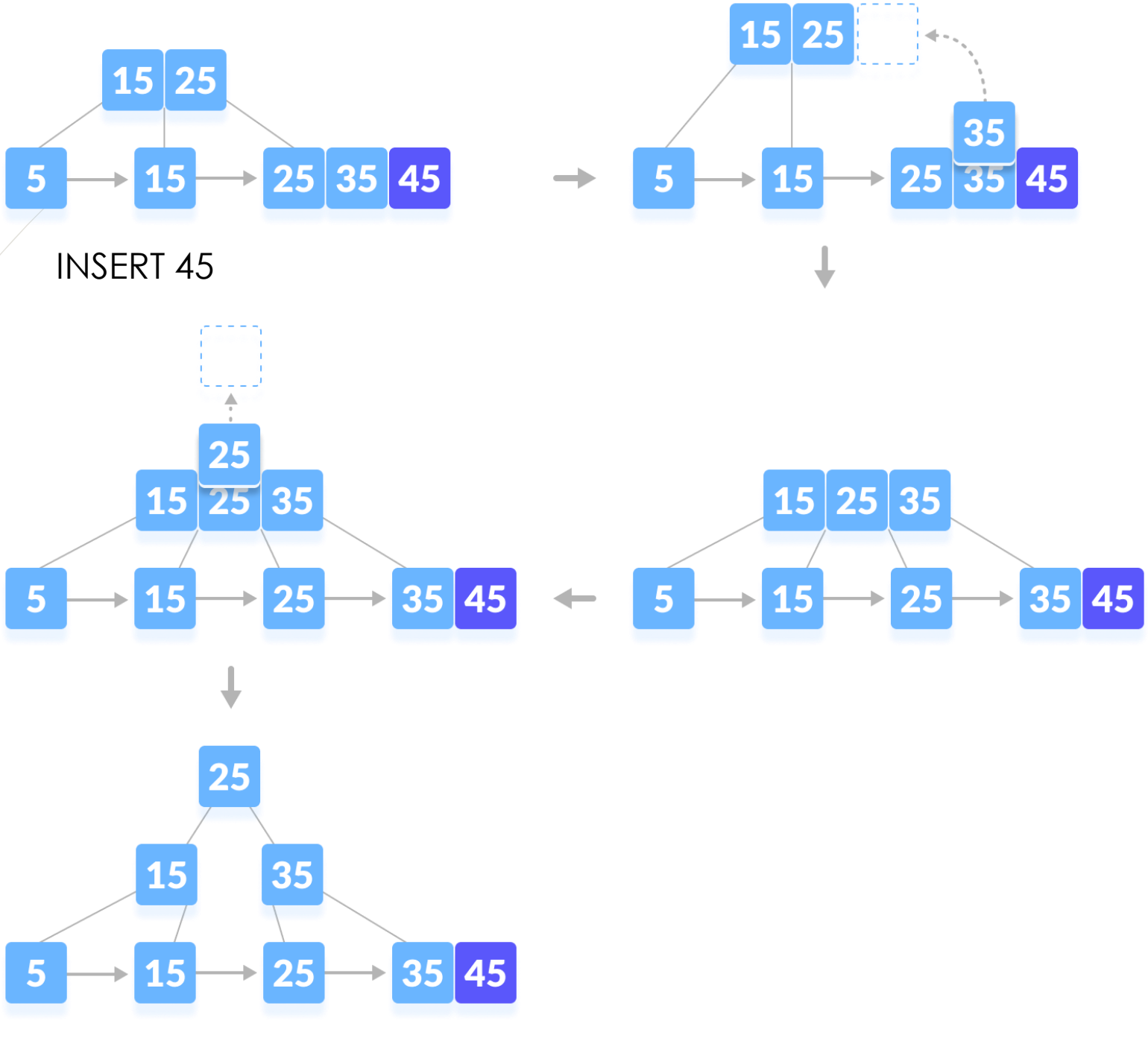
INSERT 15



INSERT 25



INSERT 35


15 25
5 15 25 35 45
INSERT 45
15 25
35
5 15 25 35 45
25
15 25 35
5 15 25 35 45
15 25 35
5 15 25 35 45
25
15 35
5 15 25 35 45

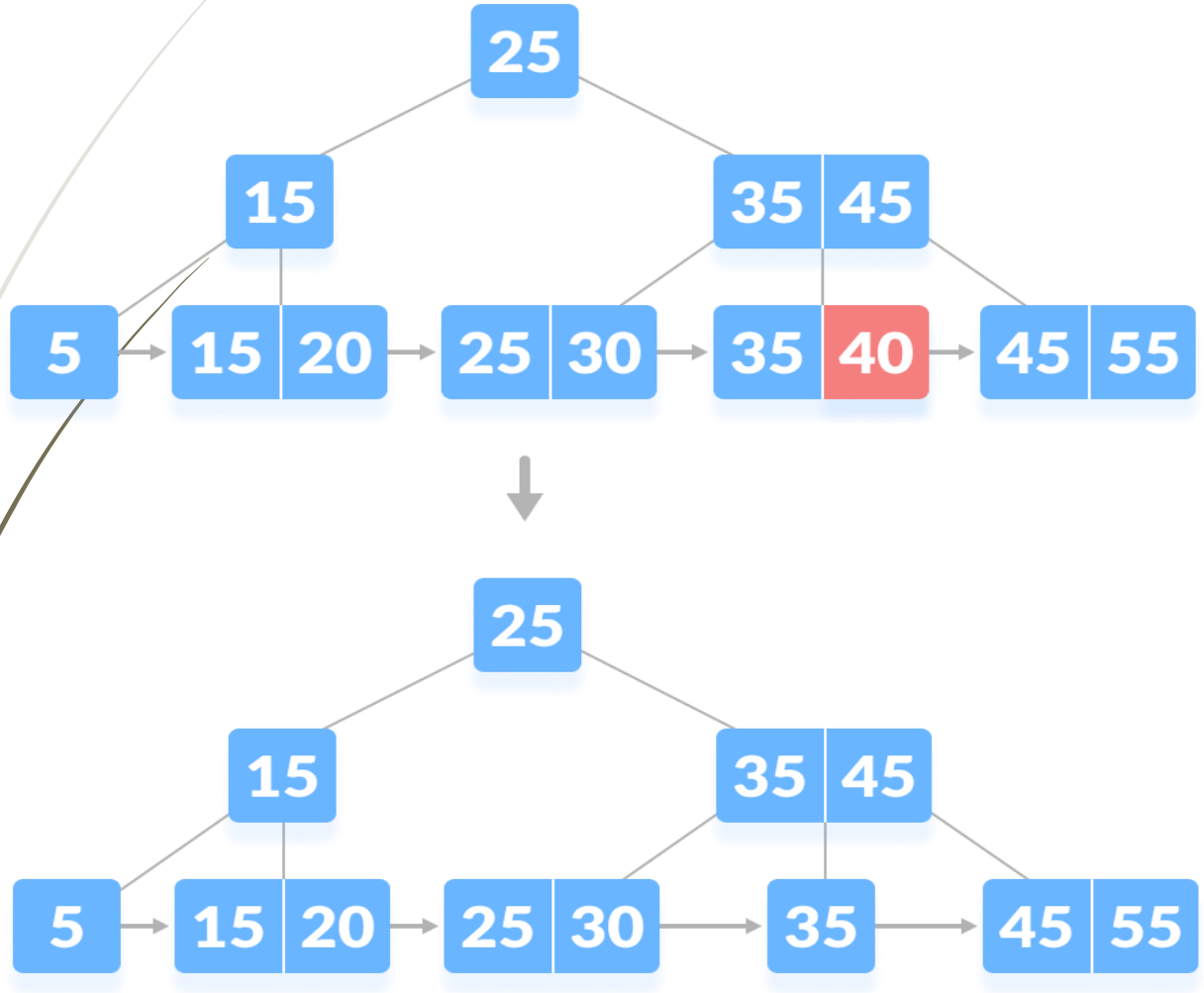
Deletion from a B+ Tree

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

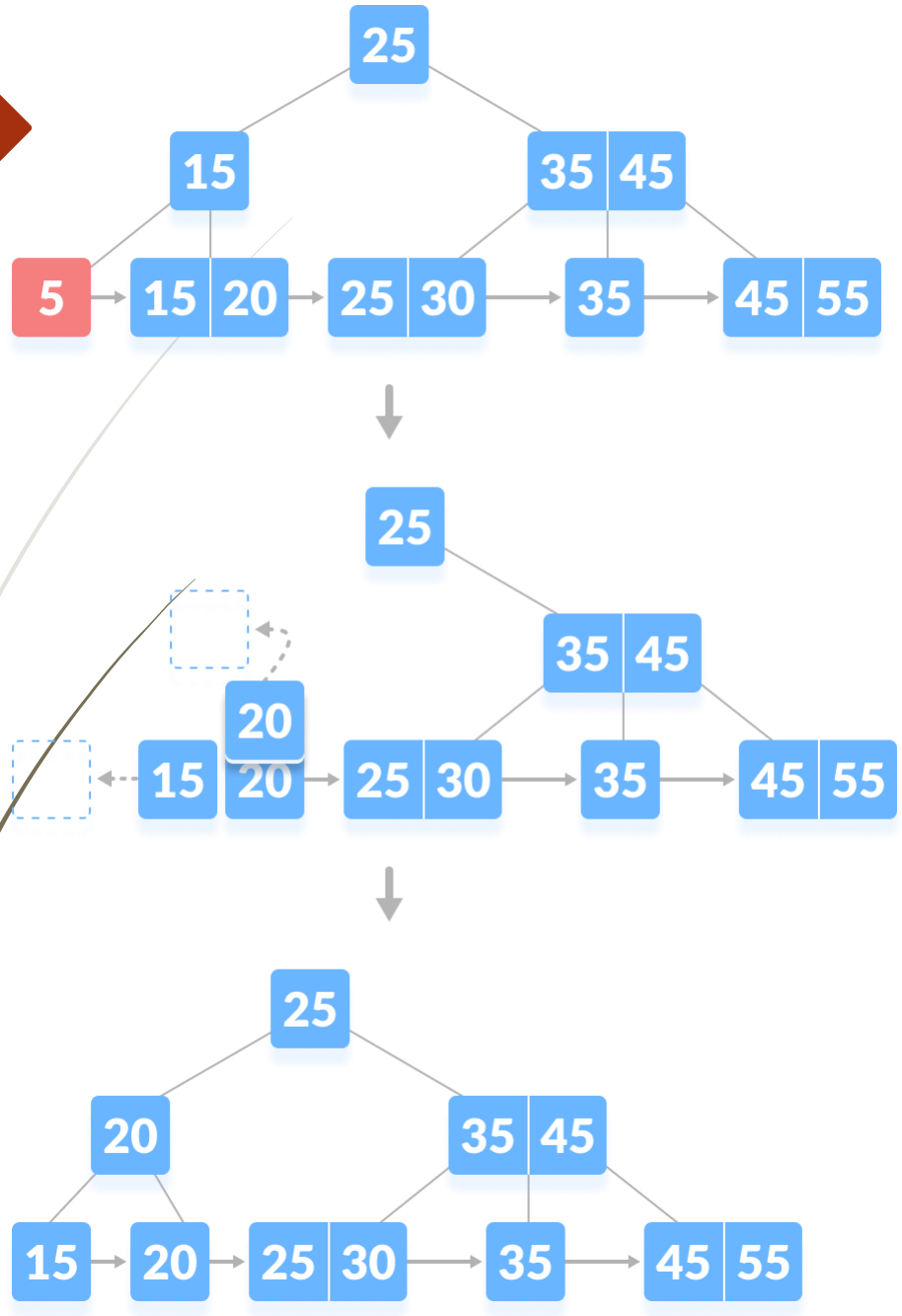
While deleting a key, we have to take care of the keys present in the internal nodes (i.e. indexes) as well because the values are redundant in a B+ tree. Search the key to be deleted then follow the following steps.



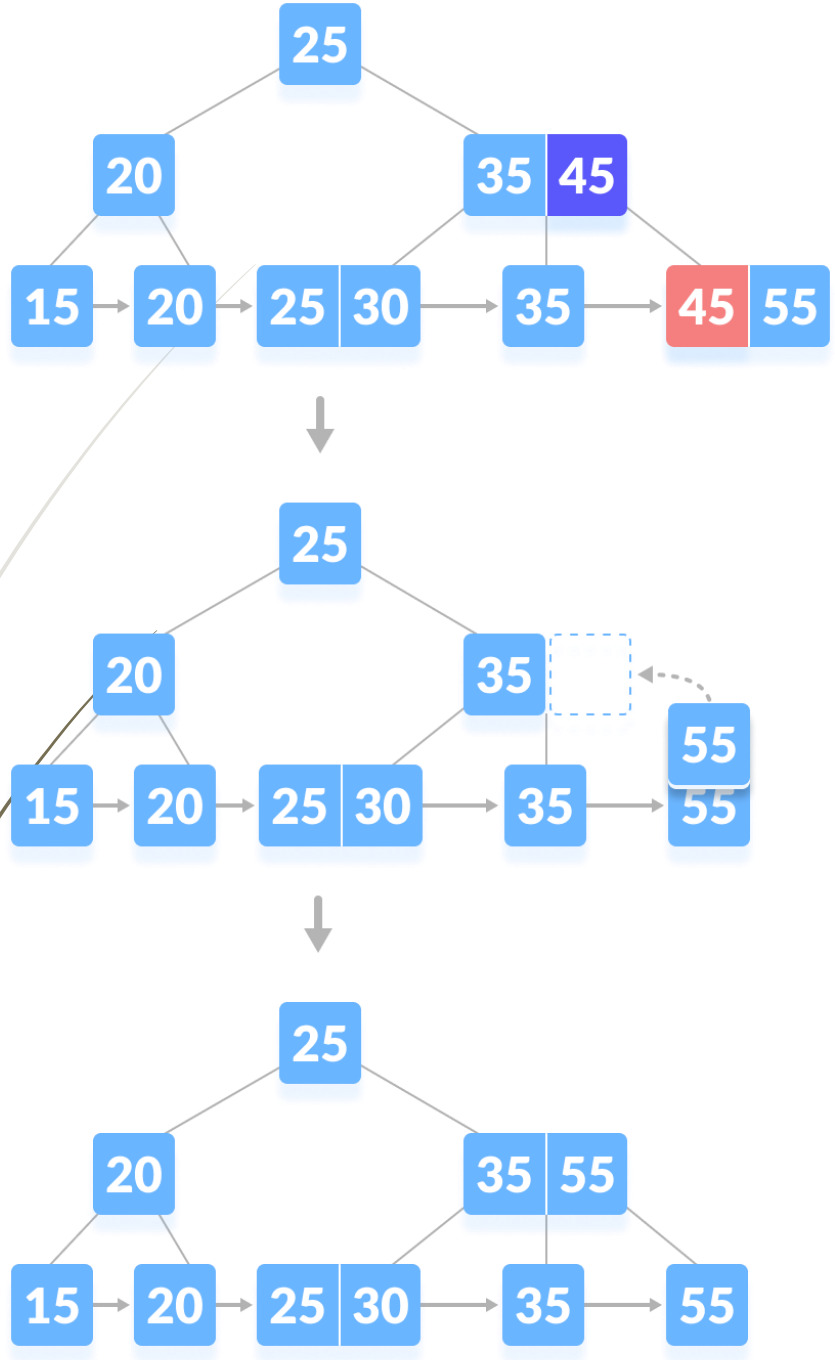
Case I:

The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:

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



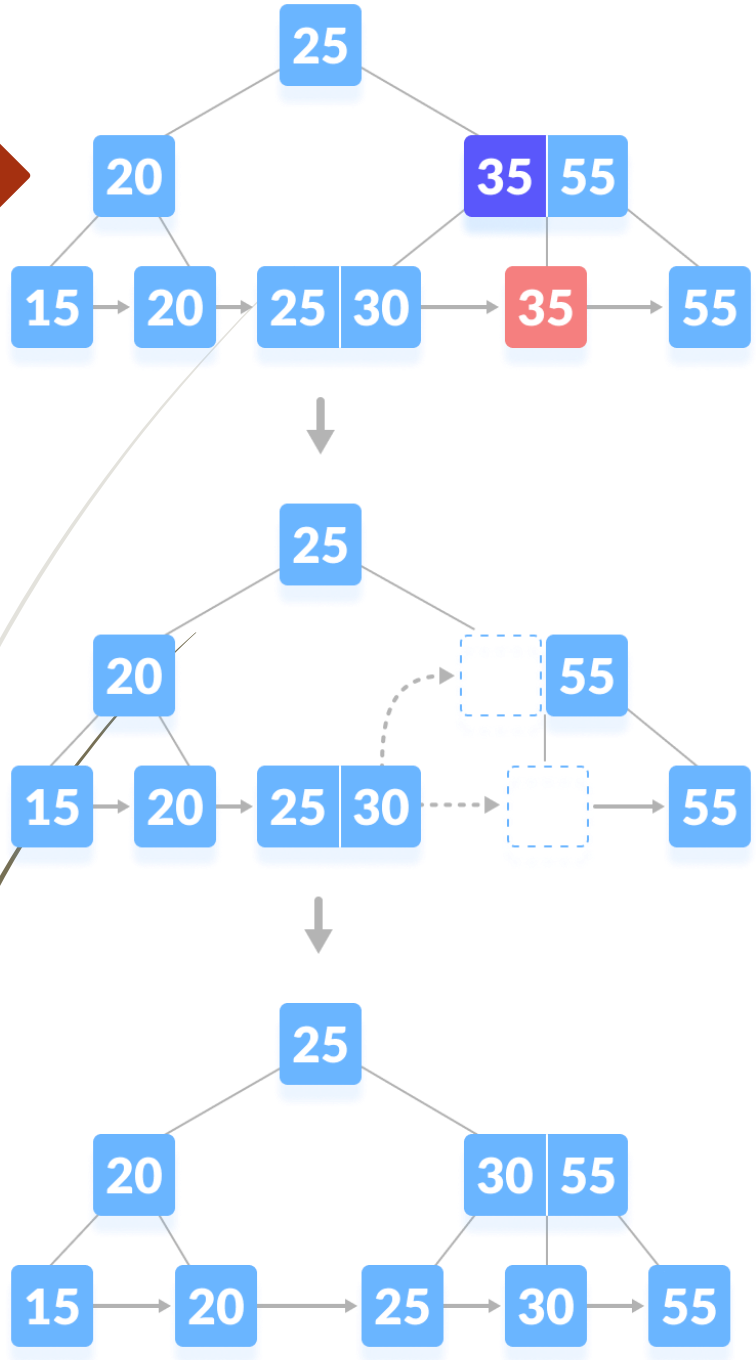
There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.



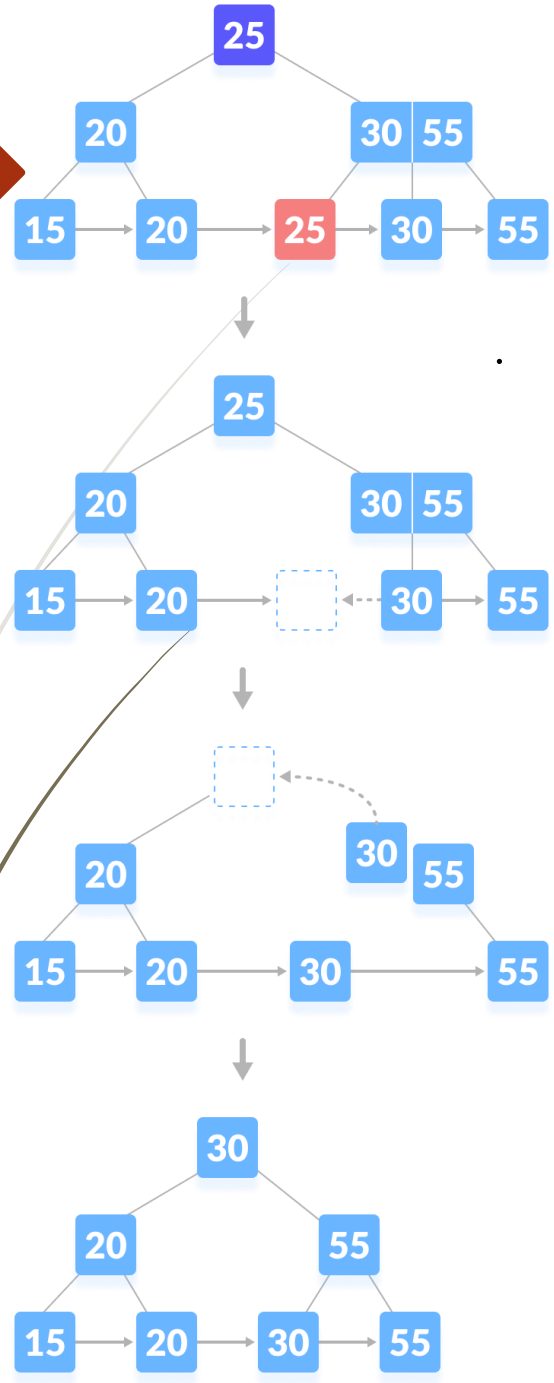
Case II

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. There are the following cases for this situation.

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.

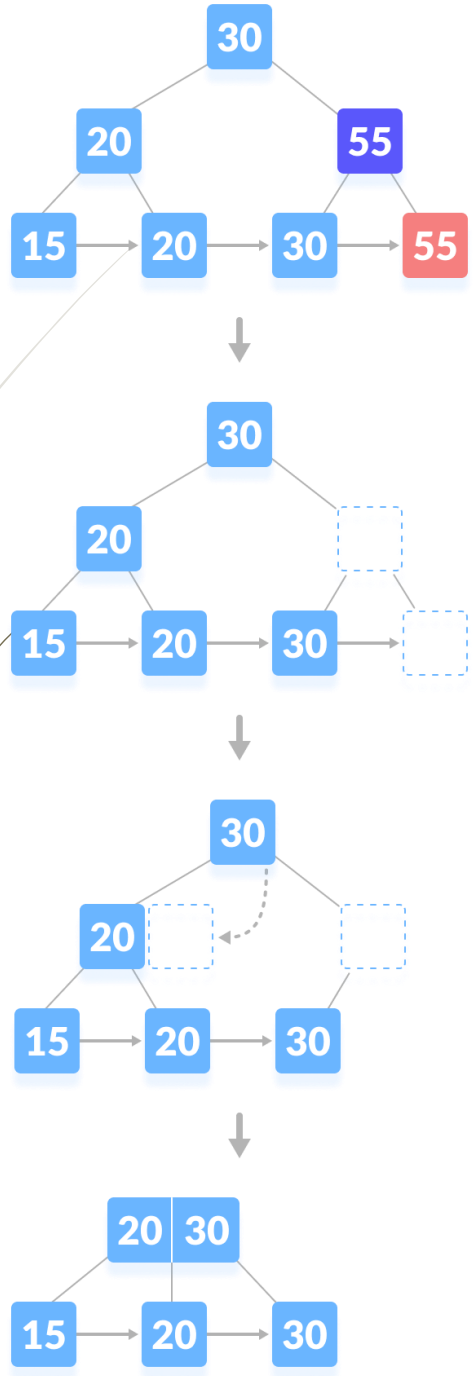


If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent). Fill the empty space created in the index (internal node) with the borrowed key.



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

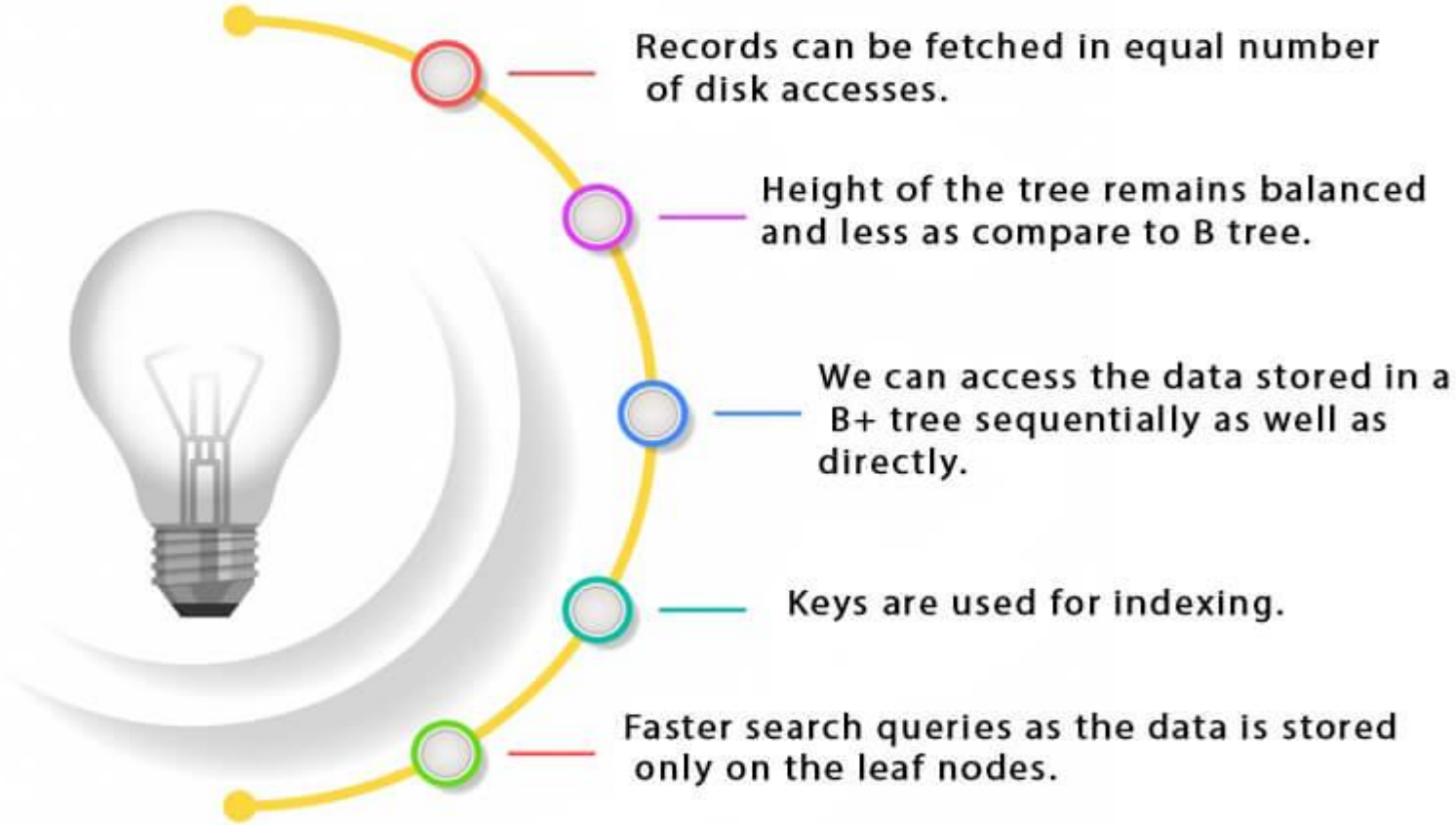


In this case, the height of the tree gets shrinked . It is a little complicated . Deleting 55 from the tree below leads to this condition. It can be understood in the illustrations below.

Advantage

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.
6. Multilevel Indexing
7. Faster operations on the tree (insertion, deletion, search)
8. Database indexing

Advantages of B+ Tree



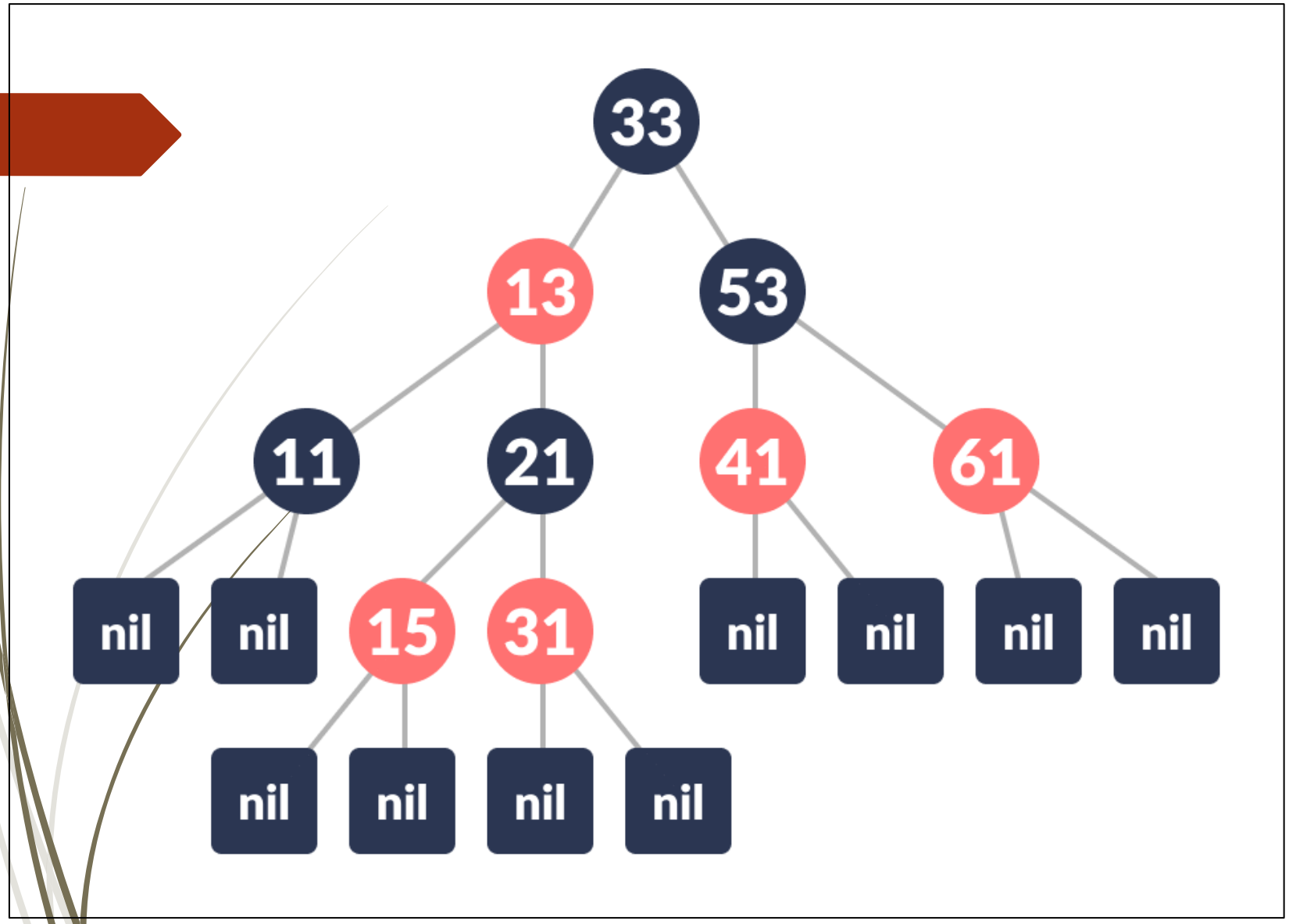
Let’s see the difference between B-tree and B+ tree:

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

Red-Black Tree

- Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.
- A red-black tree satisfies the following properties:

1. Red - Black Tree must be a Binary Search Tree.
2. **Red/Black Property:** Every node is colored, either red or black.
3. **Root Property:** The root is black.
4. **Leaf Property:** Every leaf (NIL) is black.
5. **Red Property:** If a red node has children then, the children are always black.
6. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).



Each node has the following attributes:

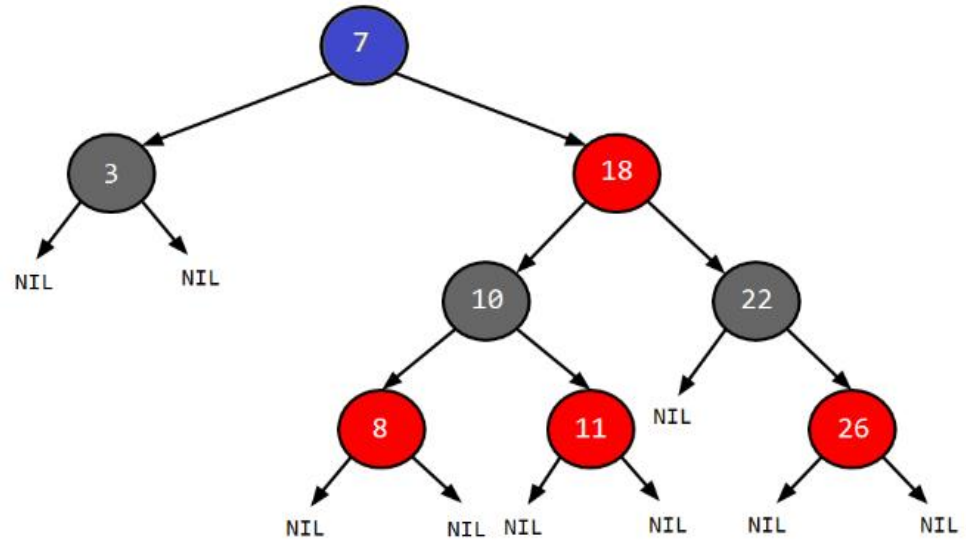
- Color
- Key
- Left Child
- Right Child
- Parent (except root node)

Search Operation in Red-black Tree:

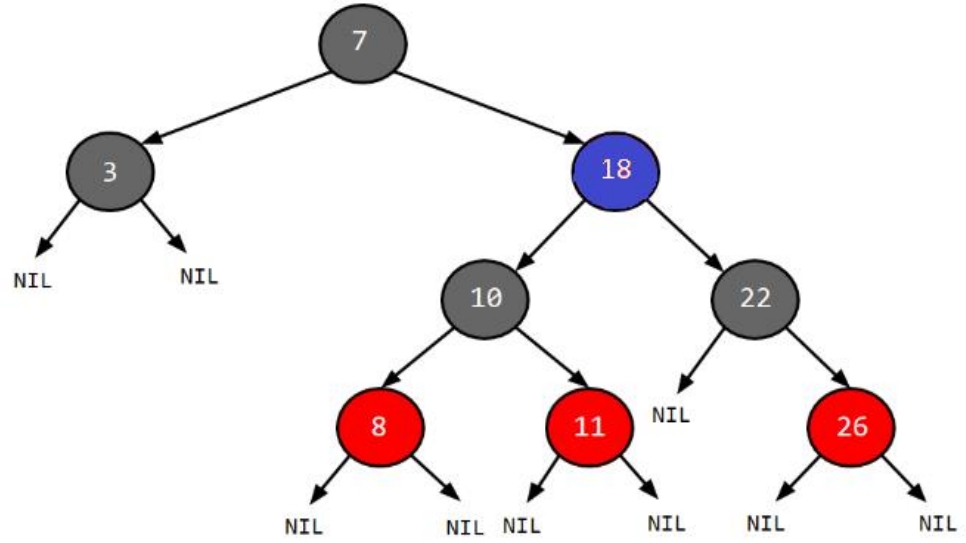
As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

```
searchElement (tree, val)

Step 1: If tree -> data = val OR tree = NULL
            Return tree
        Else If val < data
            Return searchElement (tree -> left, val)
        Else
            Return searchElement (tree -> right, val)

         [ End of if ]

Step 2: END
```

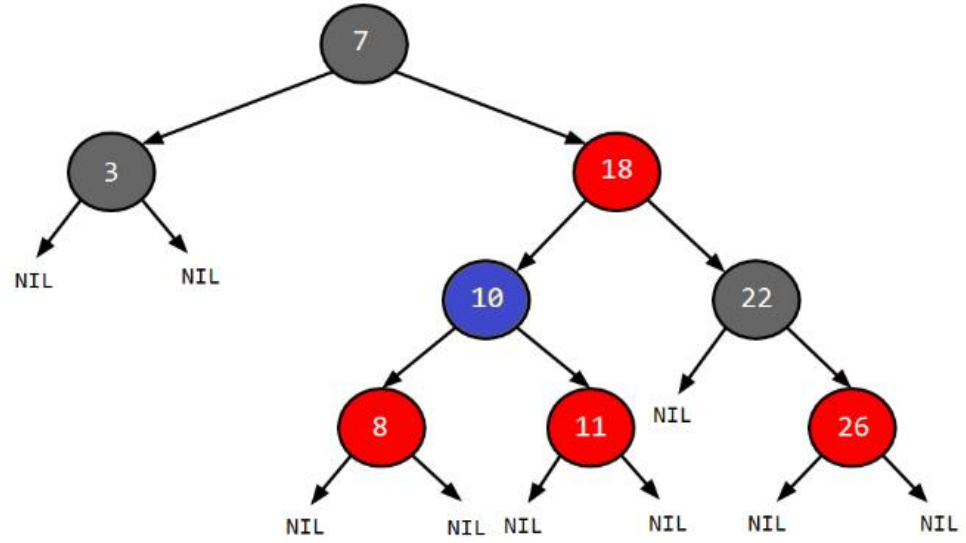
Step-1



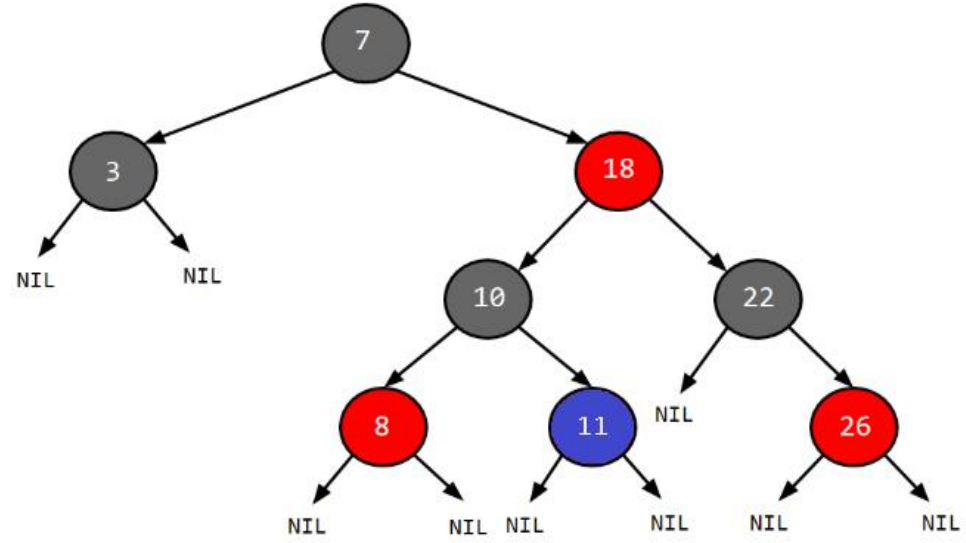
Step-2



Step-3



Step-4



Operations on a Red-Black Tree

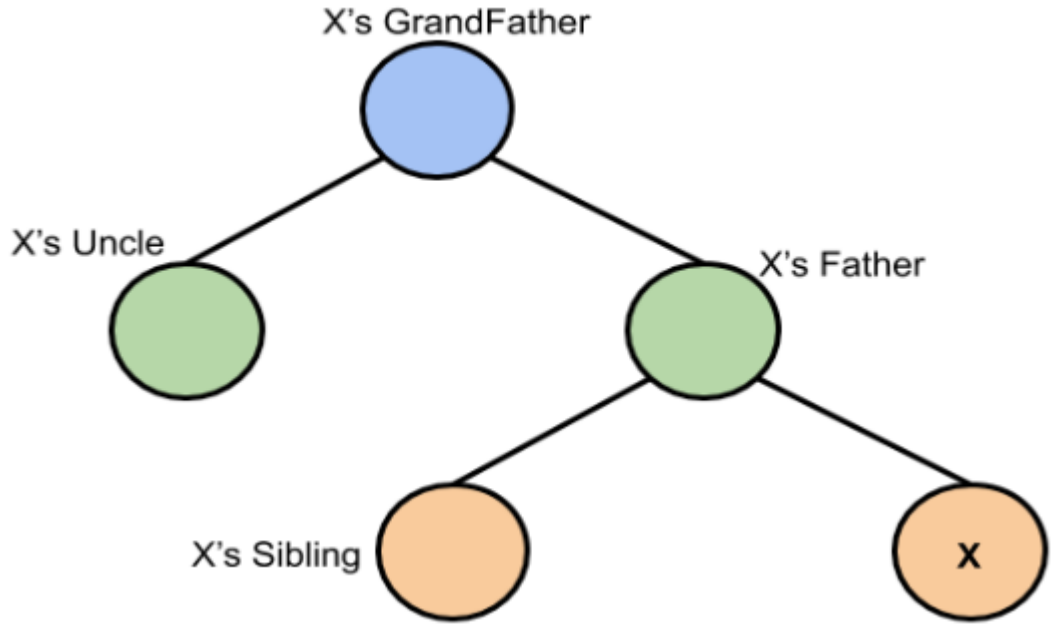
Various operations that can be performed on a red-black tree are:

Rotating the subtrees in a Red-Black Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

Rotation operation is used for maintaining the properties of a red-black tree when they are violated by other operations such as insertion and deletion.

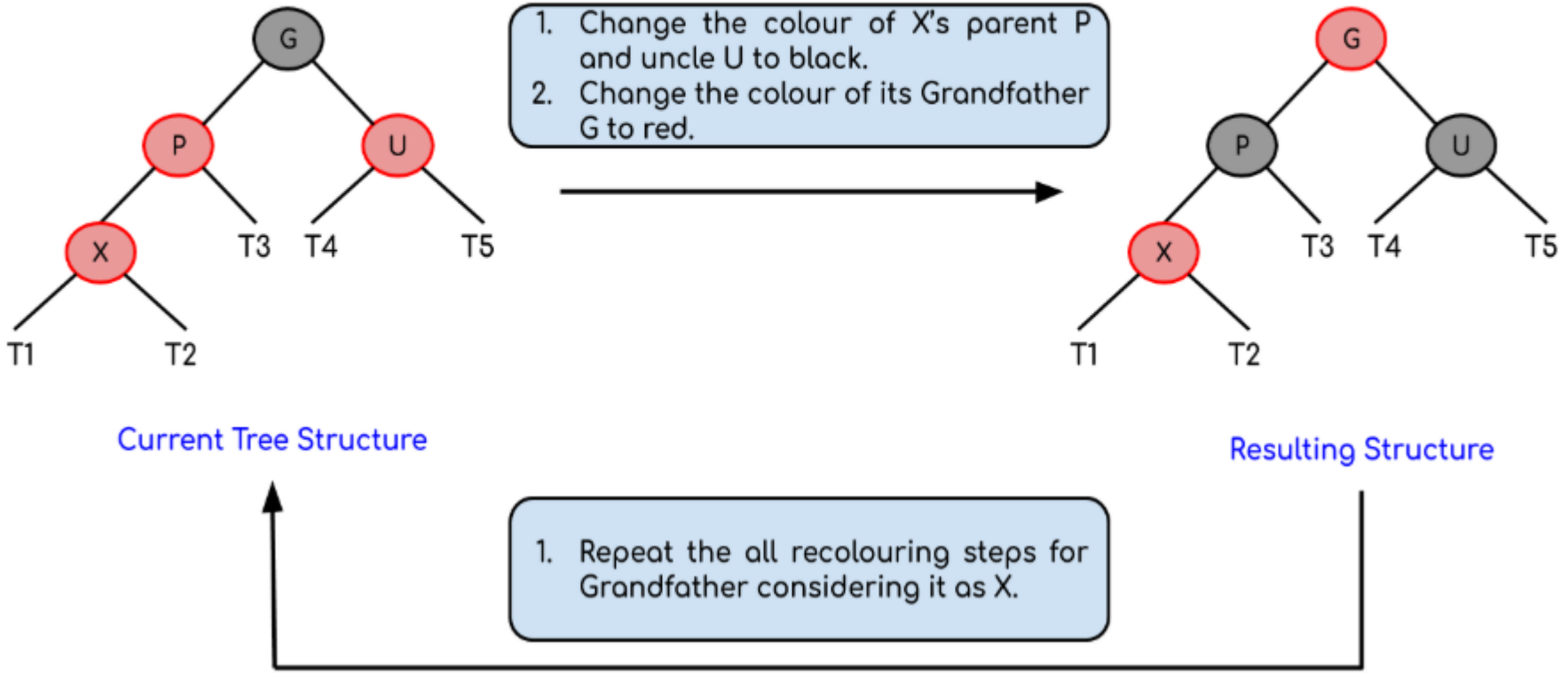
There are two types of rotations: Left rotation and Right rotation



Logic:

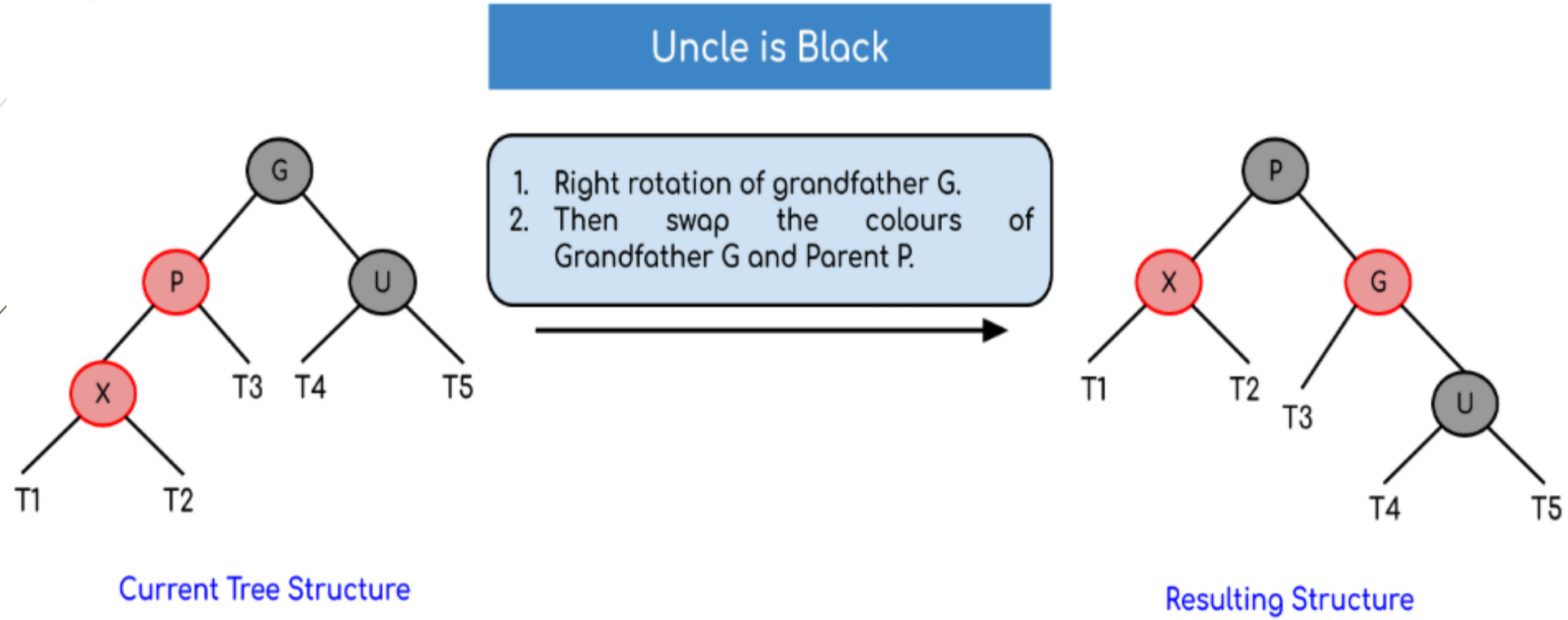
First, you have to insert the node similarly to that in a binary tree and assign a red colour to it. Now, if the node is a root node then change its colour to black, but if it does not then check the colour of the parent node. If its colour is black then don't change the colour but if it is not i.e. it is red then check the colour of the node's uncle. If the node's uncle has a red colour then change the colour of the node's parent and uncle to black and that of grandfather to red colour and repeat the same process for him (i.e. grandfather).

Uncle is Red

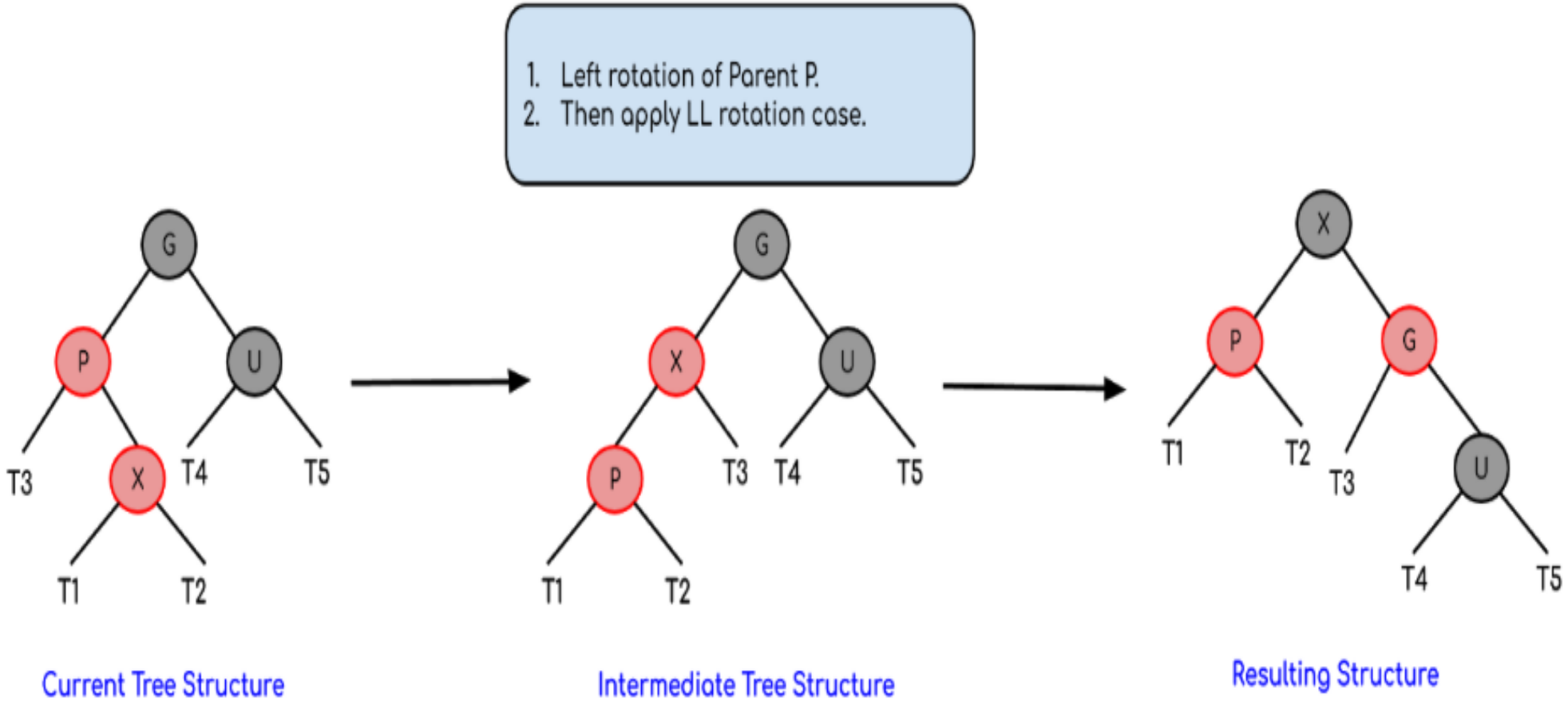

G
P
U
X
T3
T4
T5
T1
T2
1. Change the colour of X's parent P and uncle U to black.
2. Change the colour of its Grandfather G to red.
G
P
U
X
T3
T4
T5
T1
T2
Current Tree Structure
Resulting Structure
1. Repeat the all recolouring steps for Grandfather considering it as X.

But, if the node's uncle has black colour then there are 4 possible cases:

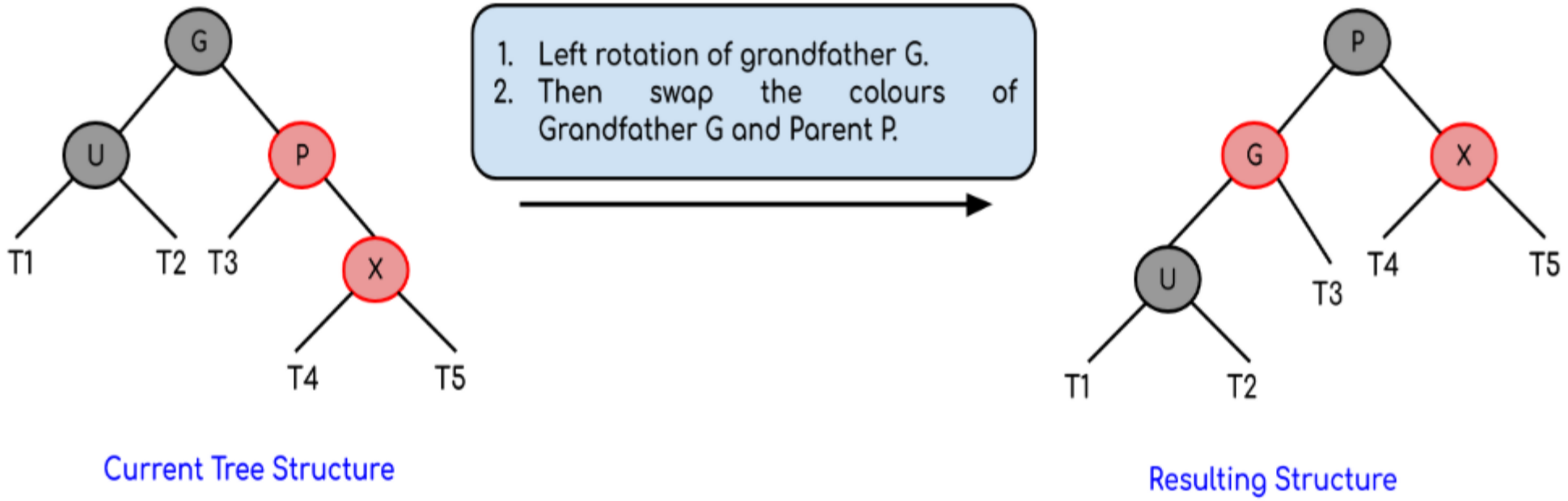
•Left Left Case (LL rotation):



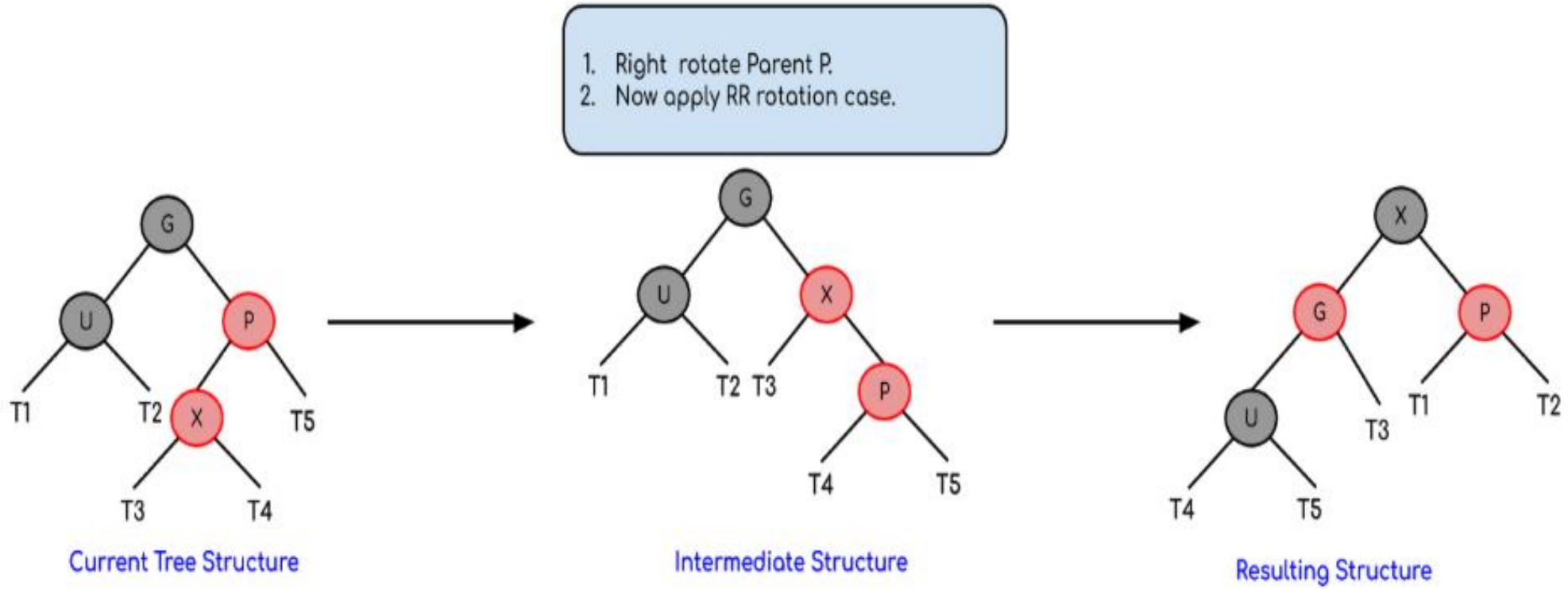
•Left Right Case (LR rotation):



•Right Right Case (RR rotation):



•Right Left Case (RL rotation):



Insertion in Red-Black Tree

In a Red-Black Tree, every new node must be inserted with the 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 perform the following operation to make it Red Black Tree.

- **1. Recolor**
- **2. Rotation**
- **3. Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the 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 leaf node with color Red.
- **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 parentnode's sibling of newNode.
- **Step 6 -** If it is colored Black or NULL then make suitable Rotation and Recolor it.
- **Step 7 -** If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

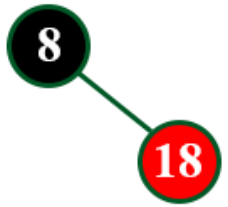
Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

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

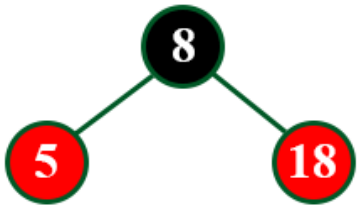
insert (18)

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



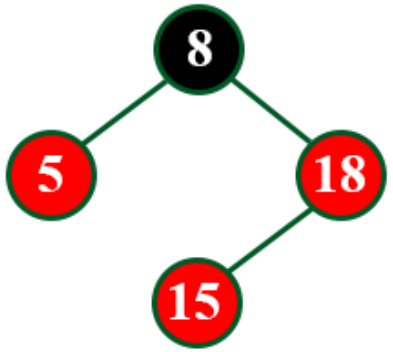
insert (5)

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



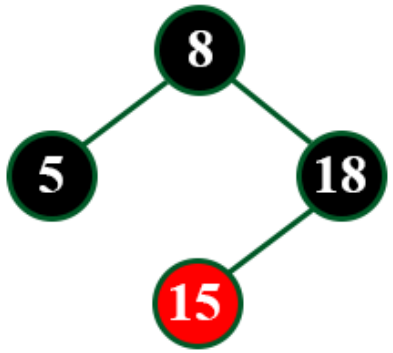
insert (15)

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.

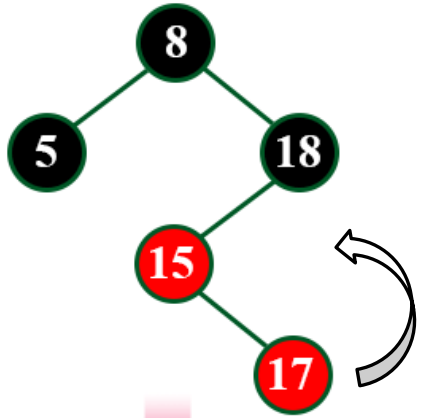
After RECOLOR



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

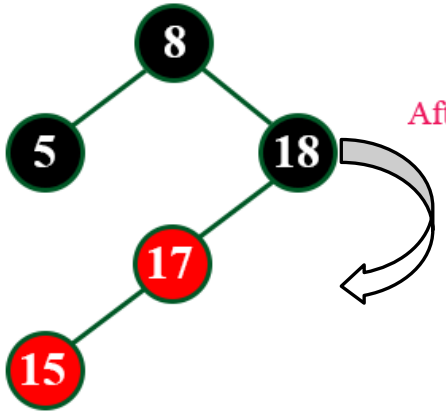
insert (17)

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

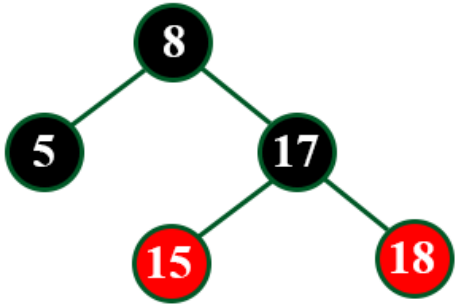


Here there are two consecutive Red nodes (15 & 17). The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

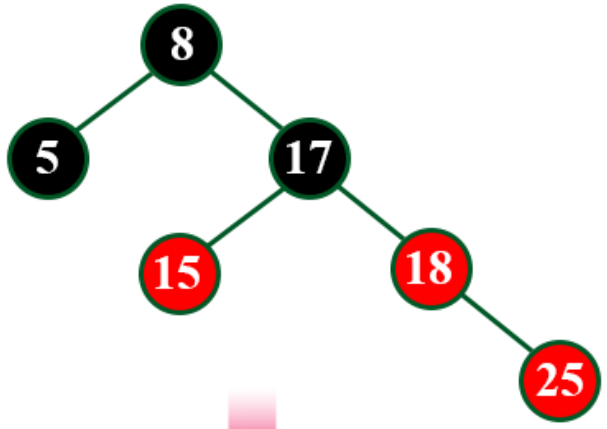


After Right Rotation & Recolor



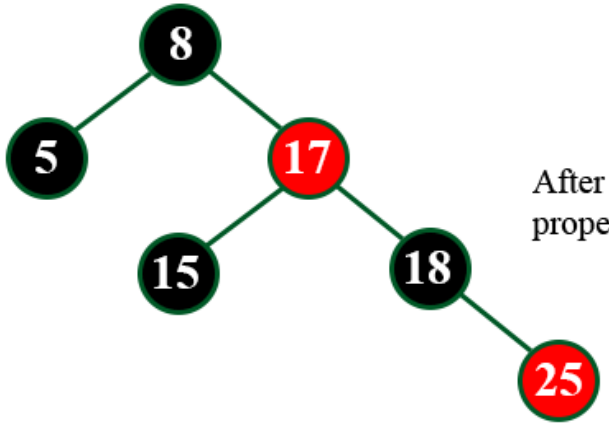
insert (25)

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.

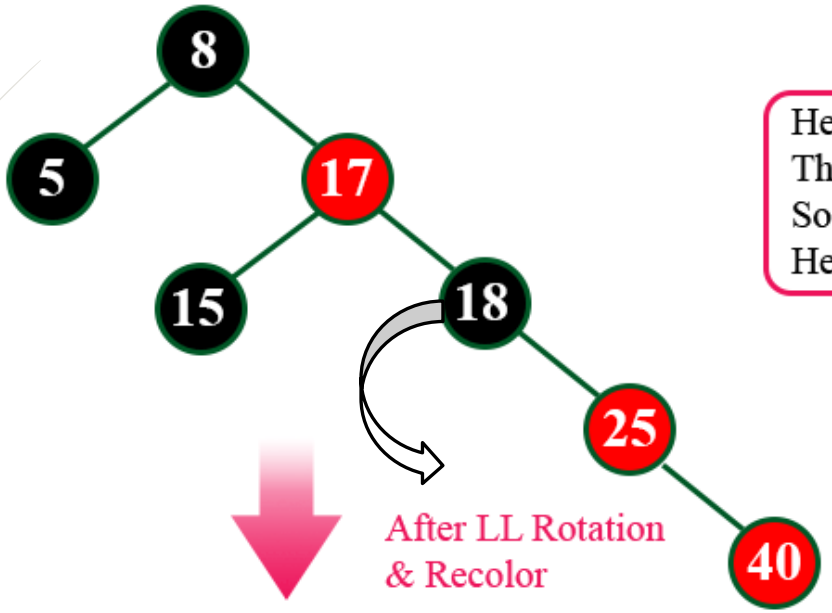
After Recolor



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

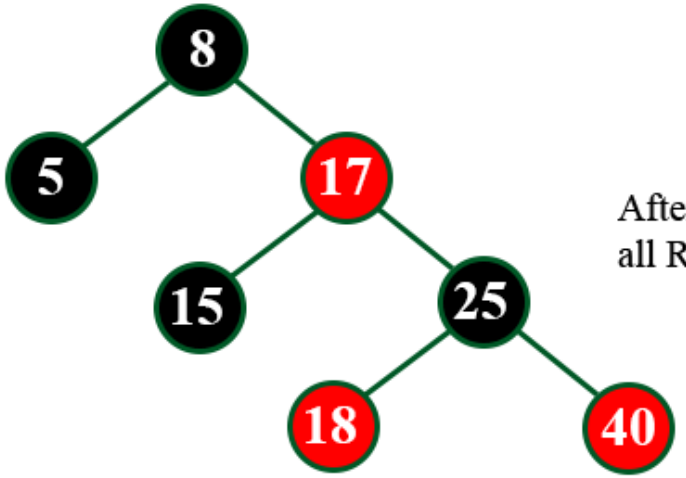
insert (40)

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.

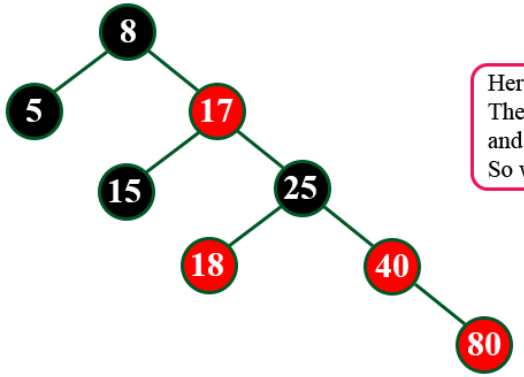
After LL Rotation
& Recolor



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

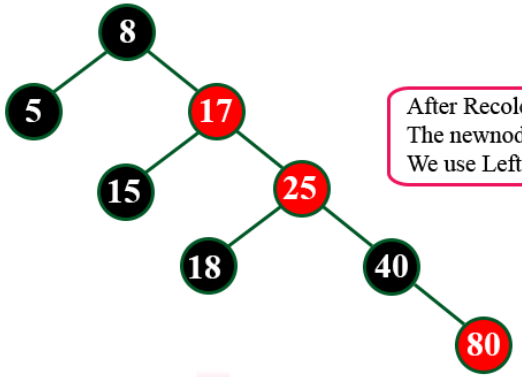
insert (80)

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



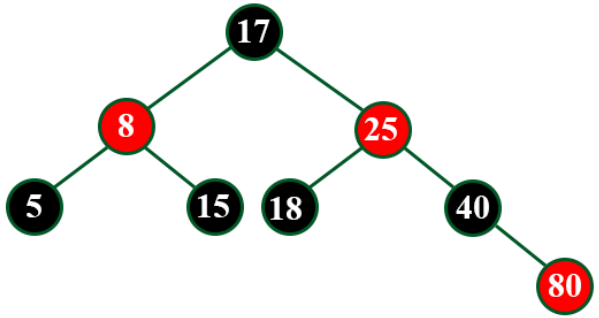
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

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



Deletion in Red-Black Tree

Insertion Vs Deletion:

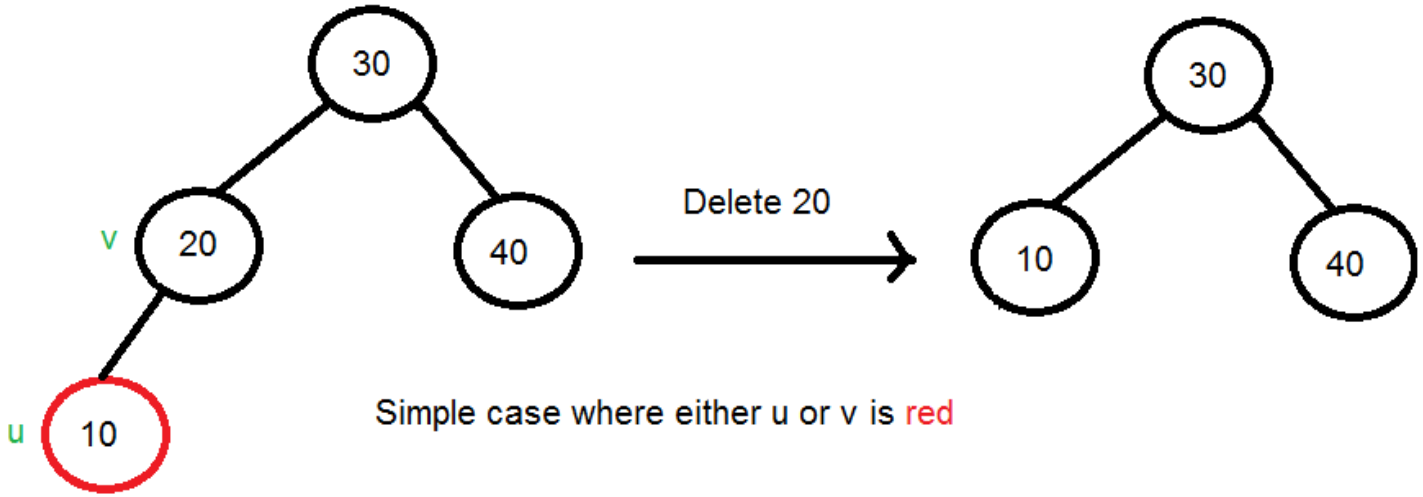
Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.
In the insert operation, we check the color of the uncle to decide the appropriate case.
In the delete operation, ***we check the color of the sibling*** to decide the appropriate case.
The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.
Deletion is a fairly complex process. To understand deletion, 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.

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Deletion Steps

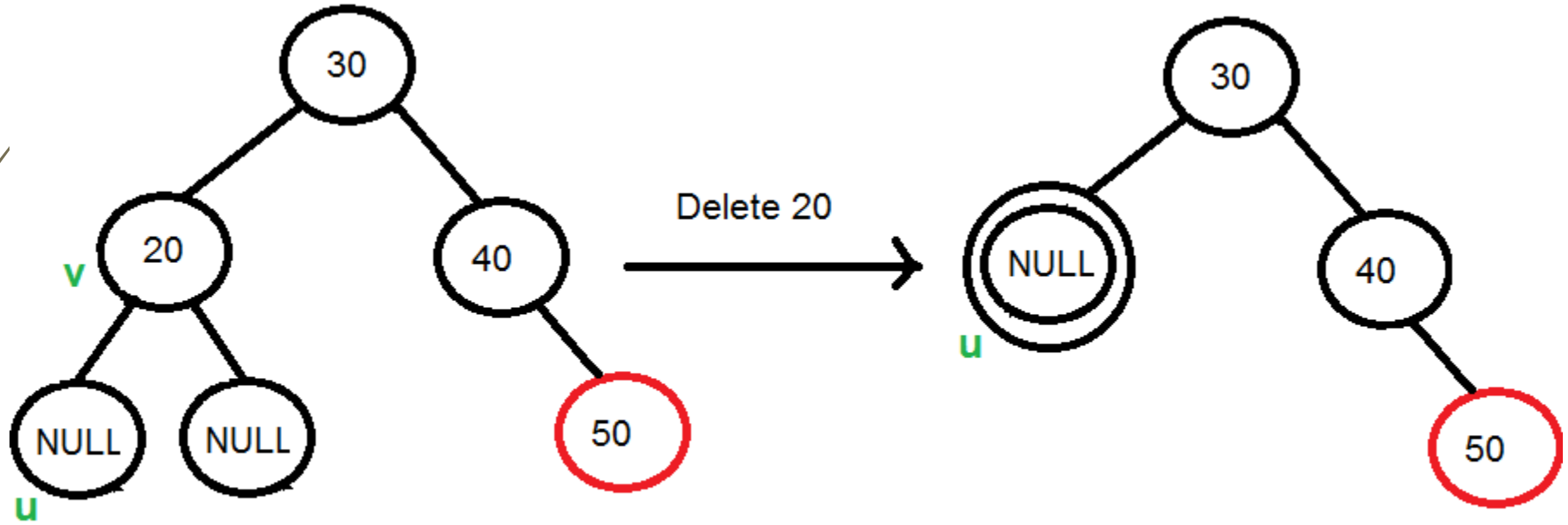
Following are detailed steps for deletion.
1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered black. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.

Note that deletion is not done yet, this double black must become single black

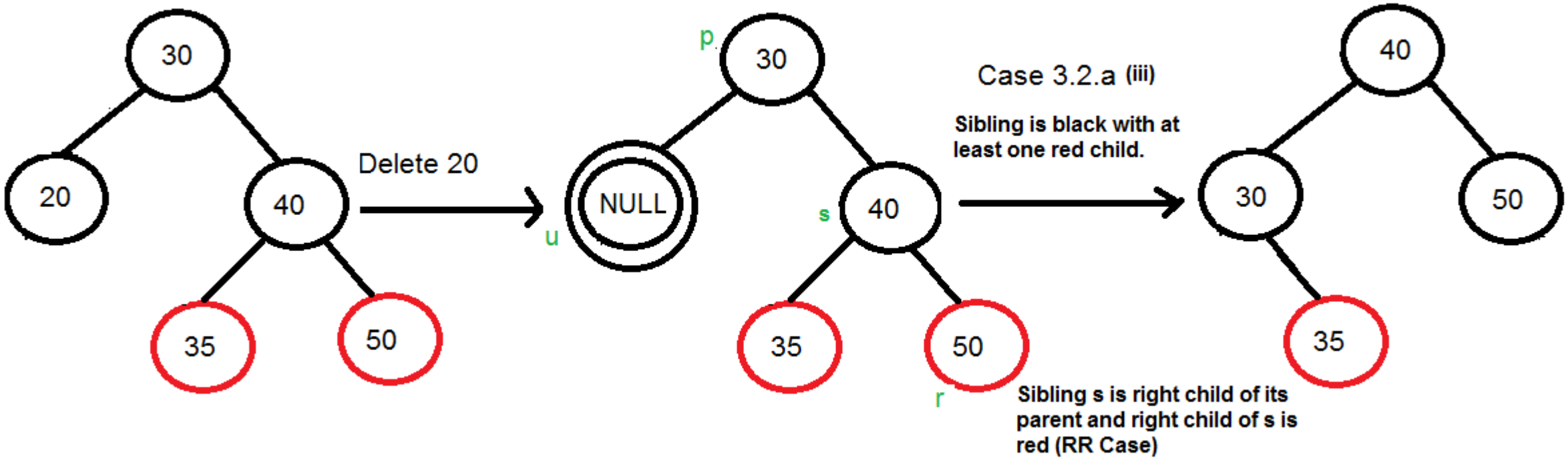
3.2) Do following while the current node u is double black, and it is not the root. Let sibling of node be **s**.

….**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

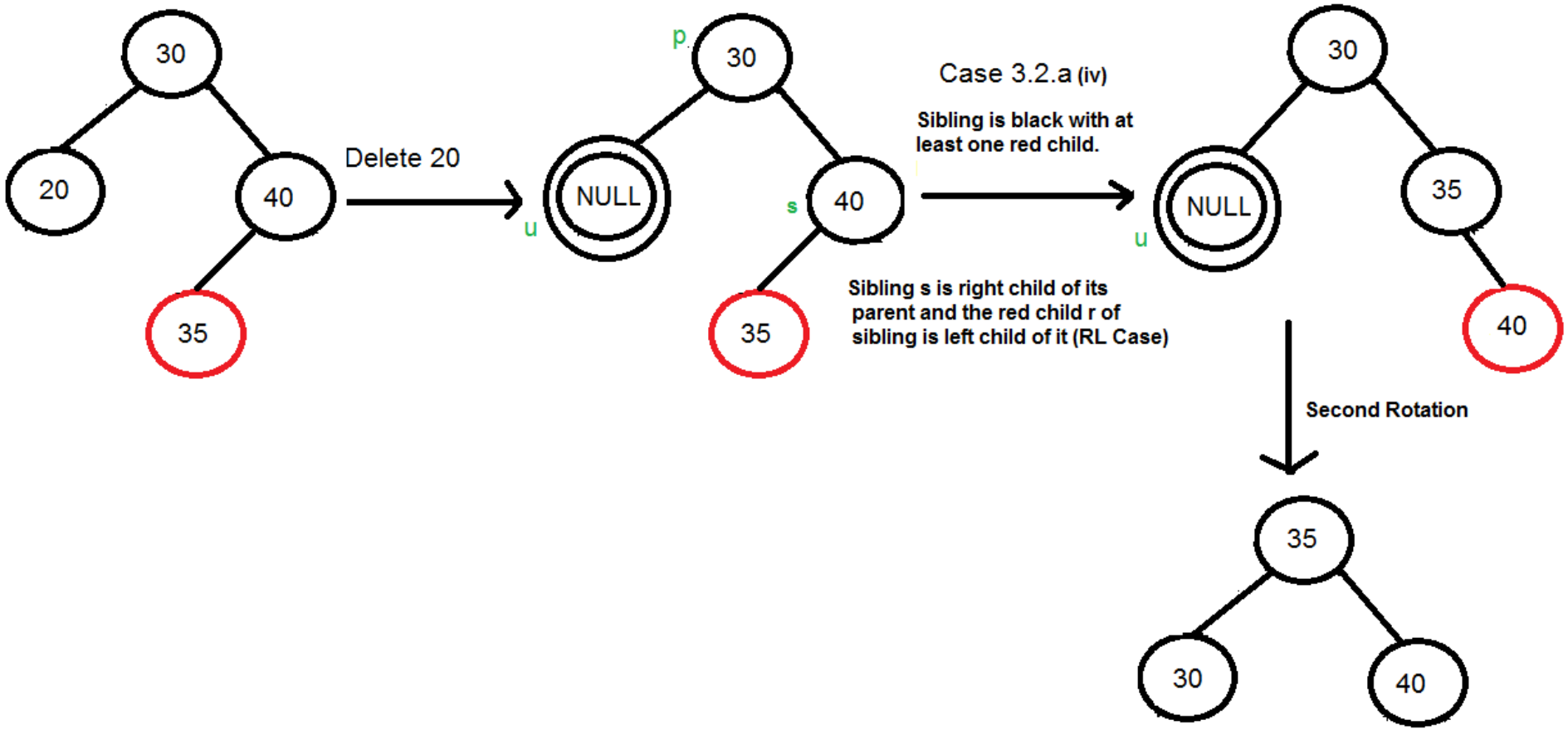
……**(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

……**(ii)** Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

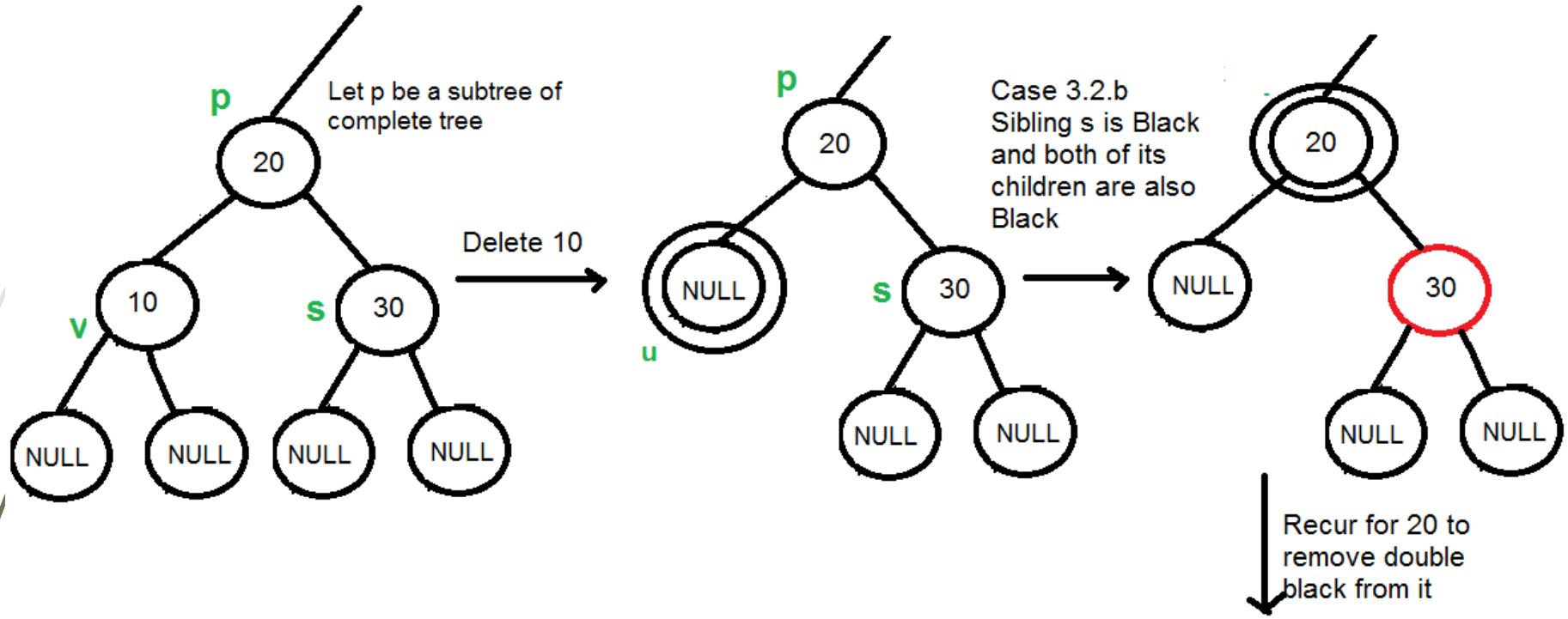
……**(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



Right Left Case (s is right child of its parent and r is left child of s)

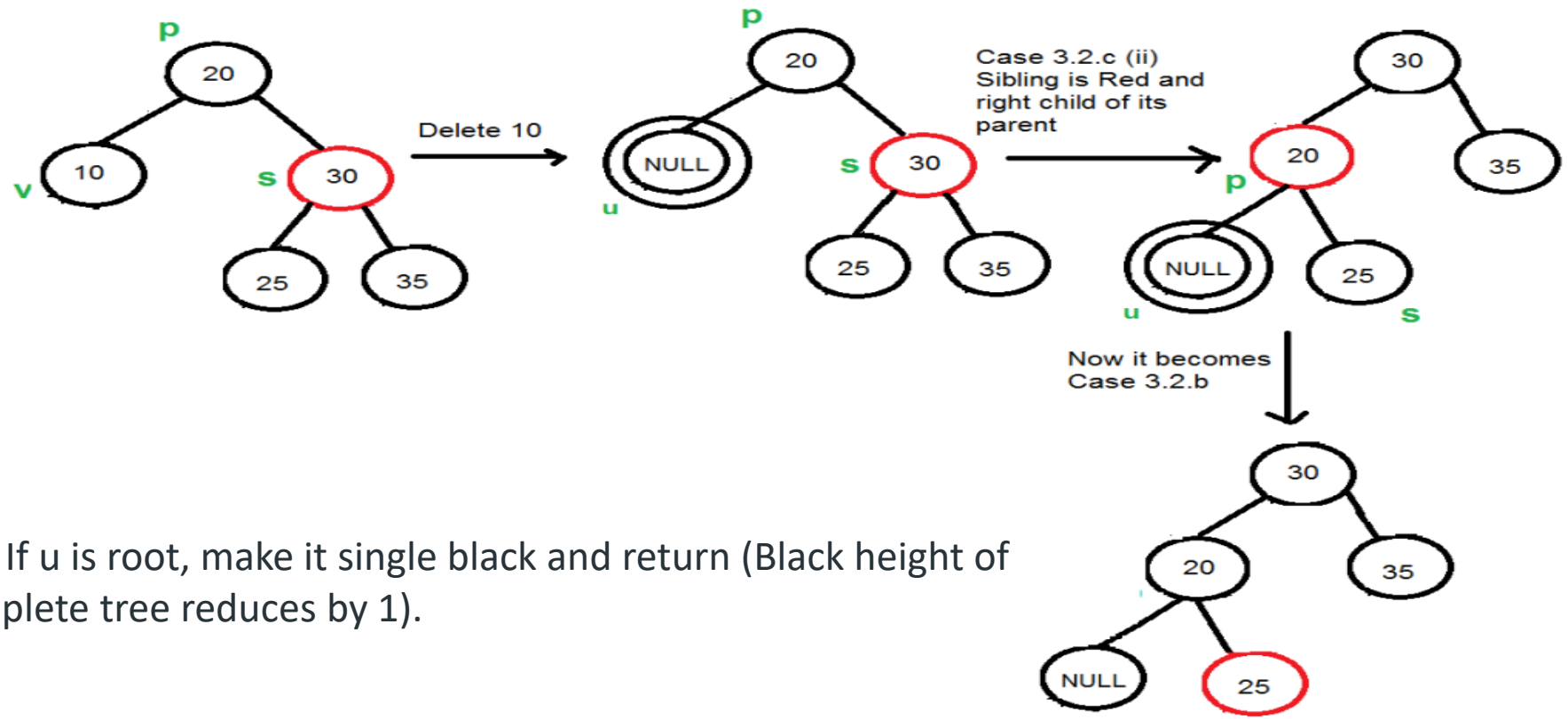


(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.



In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.
……..**(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.
……..**(iii)** Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

Red-Black Tree Applications

1.To implement finite maps
2.To implement Java packages: java.util.TreeMap and java.util.TreeSet
3.To implement Standard Template Libraries (STL) in C++: multiset, map, multimap
4.In Linux Kernel