B - Tree Data Structure

- In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children.
- But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children.
- B-Tree was developed in the year 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.
- This can store large number of elements in a single node. So the height is relatively small. This is one great advantage of B-Trees.
- . It is a generalized form of the binary search tree.

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 **m-1** keys must have **m** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

- B-Tree is determined by a number of degree, which is also called "order" (m)
- The left subtree of the node will have lesser values than the right side of the subtree. This means that the nodes are also sorted in ascending order from left to right.

m = 4

Max keys: 4 – 1 = 3

- minimum keys of [m/2]-1

  For example:

  m = 4

  Min keys: 4/2-1 = 1

- The maximum number of child nodes a node can have is equal to its degree, which is m

  Node contains

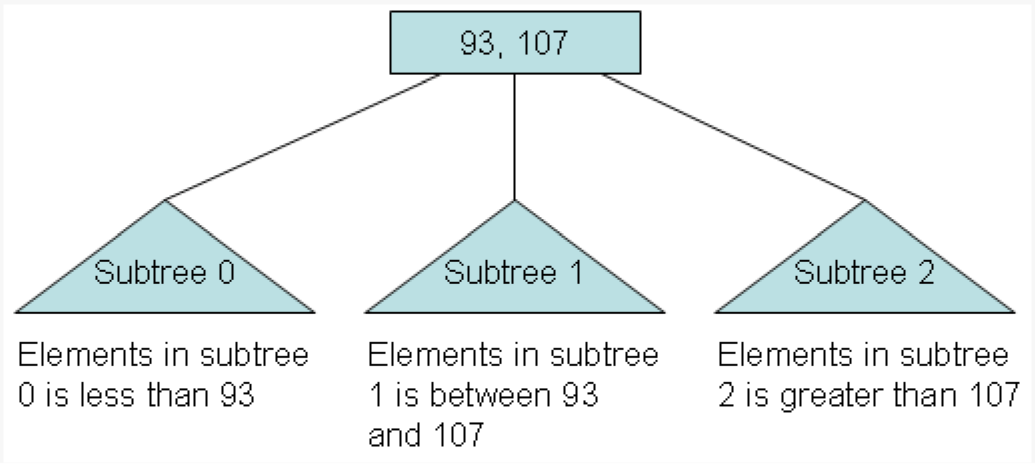
  Keys=k1,k2(k1<k2)

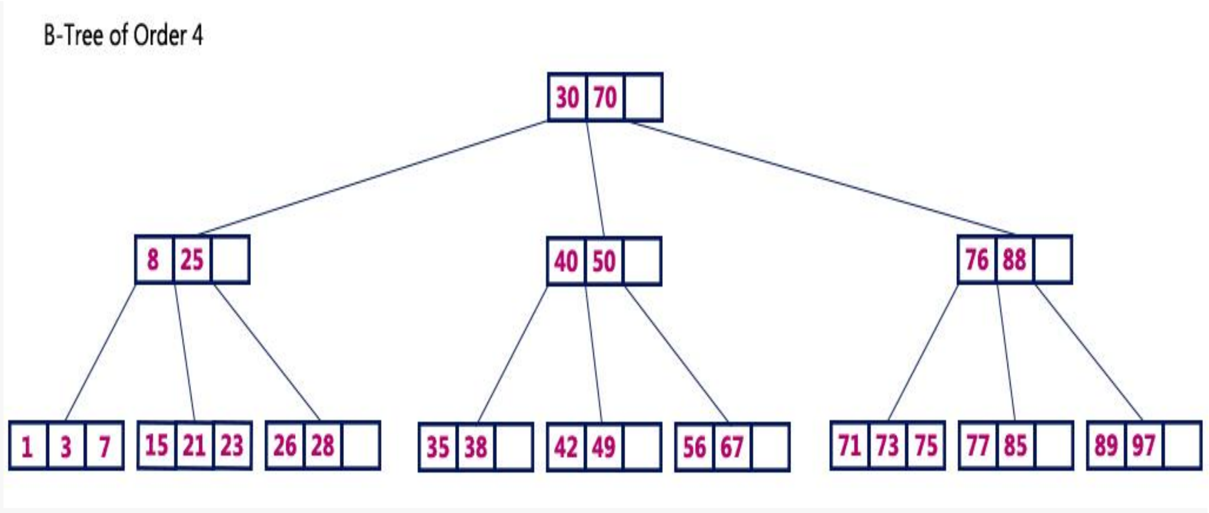
  2 keys=2+1=3 subtrees

  Subtree1 contain all the values less than k1

  Subtree2 cobtains values between k1 & k2

  Subtree 3 –values greater than k2


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



B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

Operations

The following operations are performed on a B-Tree.

1. Search
2. Insertion
3. Deletion

Insertion Operation

- 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.
- In a B-Tree, a new element must be added only at the leaf node.
- That means, the new key Value 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.

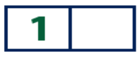
Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

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

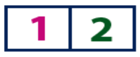
insert(1)

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



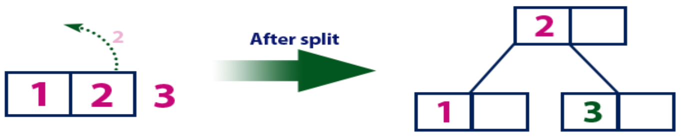
insert(2)

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.



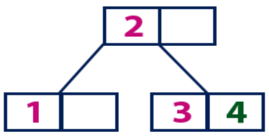
insert(3)

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.



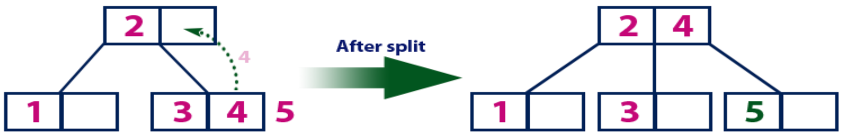
insert(4)

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.



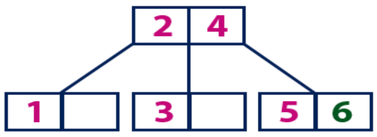
insert(5)

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.



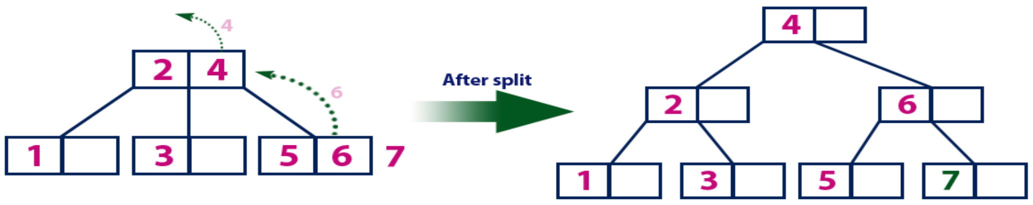
insert(6)

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)

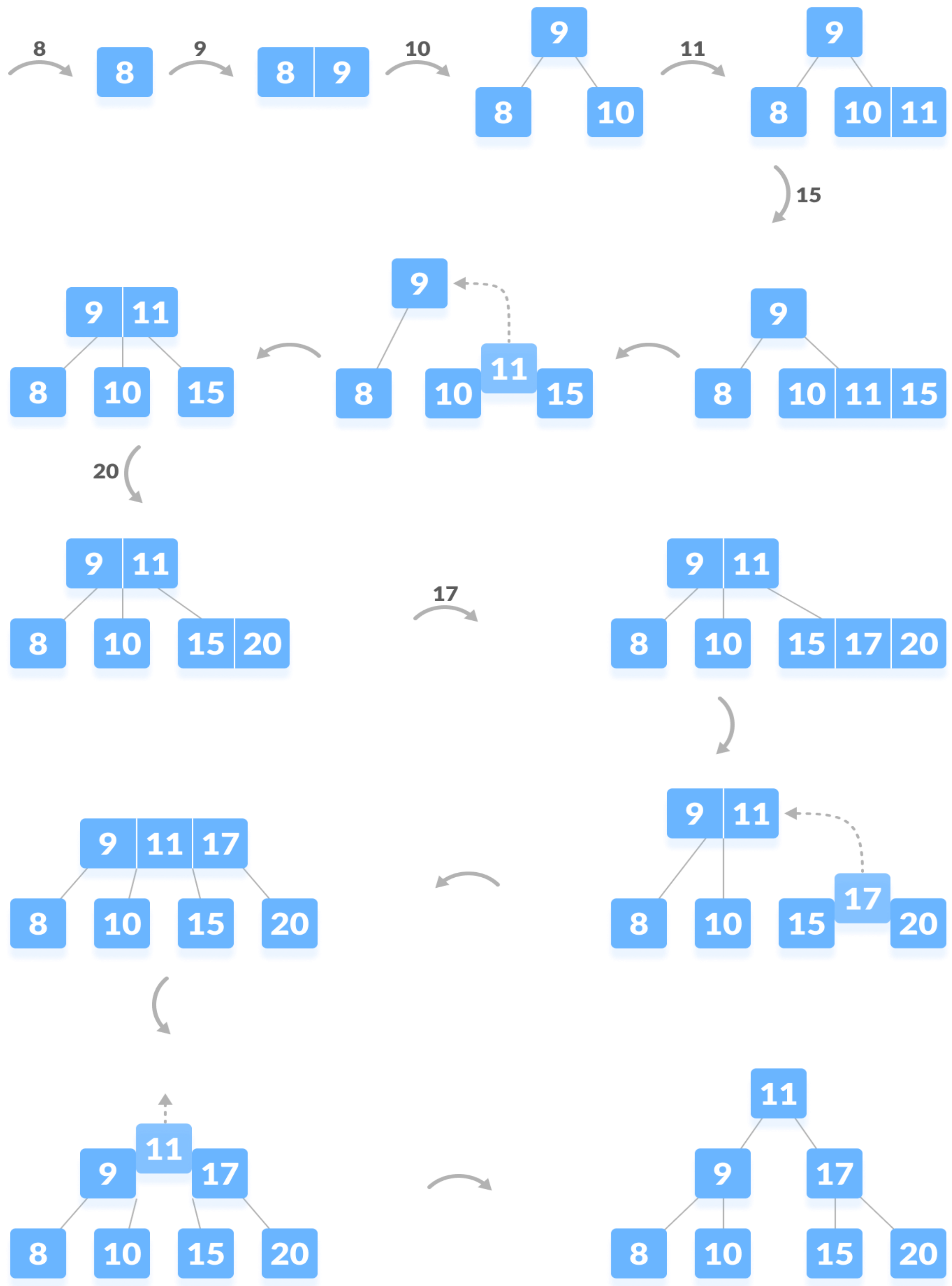
Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8)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

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.

Example

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.Order is 3



- **Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.**

The following algorithm applies:

- **Run the search operation and find the appropriate place of insertion.**
- **Insert the new key at the proper location, but if the node has a maximum number of keys already:**

  - ✓ **The node, along with a newly inserted key, will split from the middle element.**
  - ✓ **The middle element will become the parent for the other two child nodes.**
  - ✓ **The nodes must re-arrange keys in ascending order.**

Search Operation

- The search operation in B-Tree is similar to the search operation in Binary Search Tree.
- In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree).
- In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has.
- The search operation is the simplest operation on B Tree.

STEPS

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

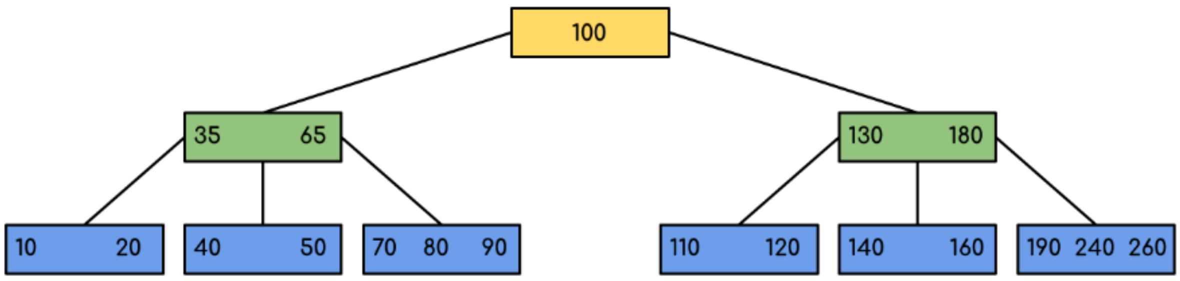
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

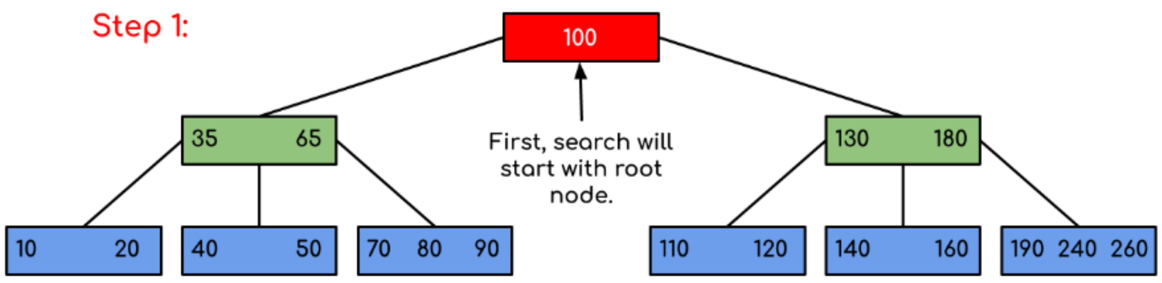
Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

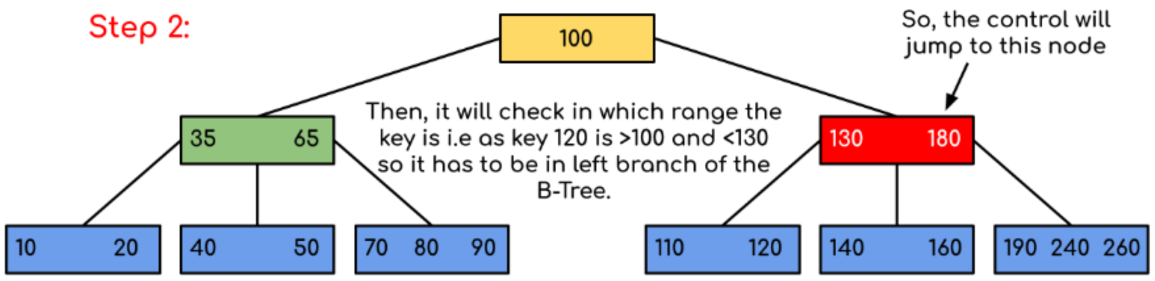
Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

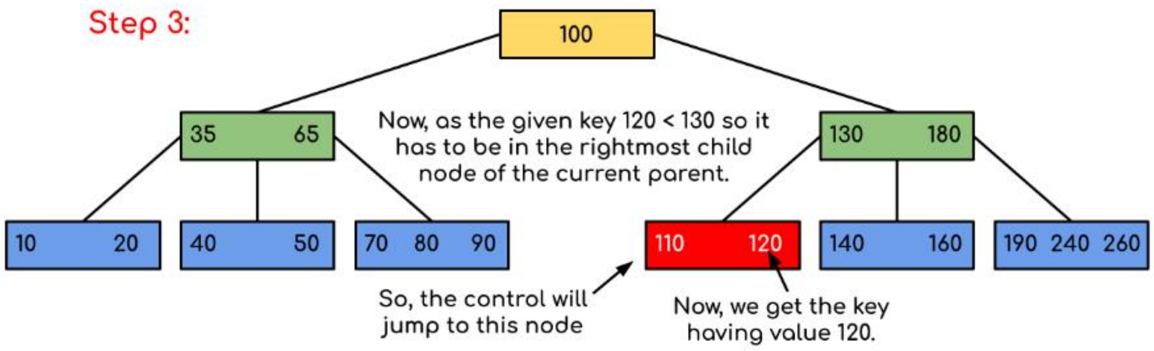
Example: Searching 125 in the given B-Tree.


100
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260

Solution:


Step 1:
100
First, search will start with root node.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260


Step 2:
100
So, the control will jump to this node
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260


Step 3:
100
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

B-Tree –Deletion

- 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.

- Before going through the steps below, one must know these facts about a B tree of degree m.(m=3)

1. **A node can have a maximum of m children. (i.e. 3)**
2. **A node can contain a maximum of m - 1 keys. (i.e. 2)**
3. **A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)**
4. **A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)**

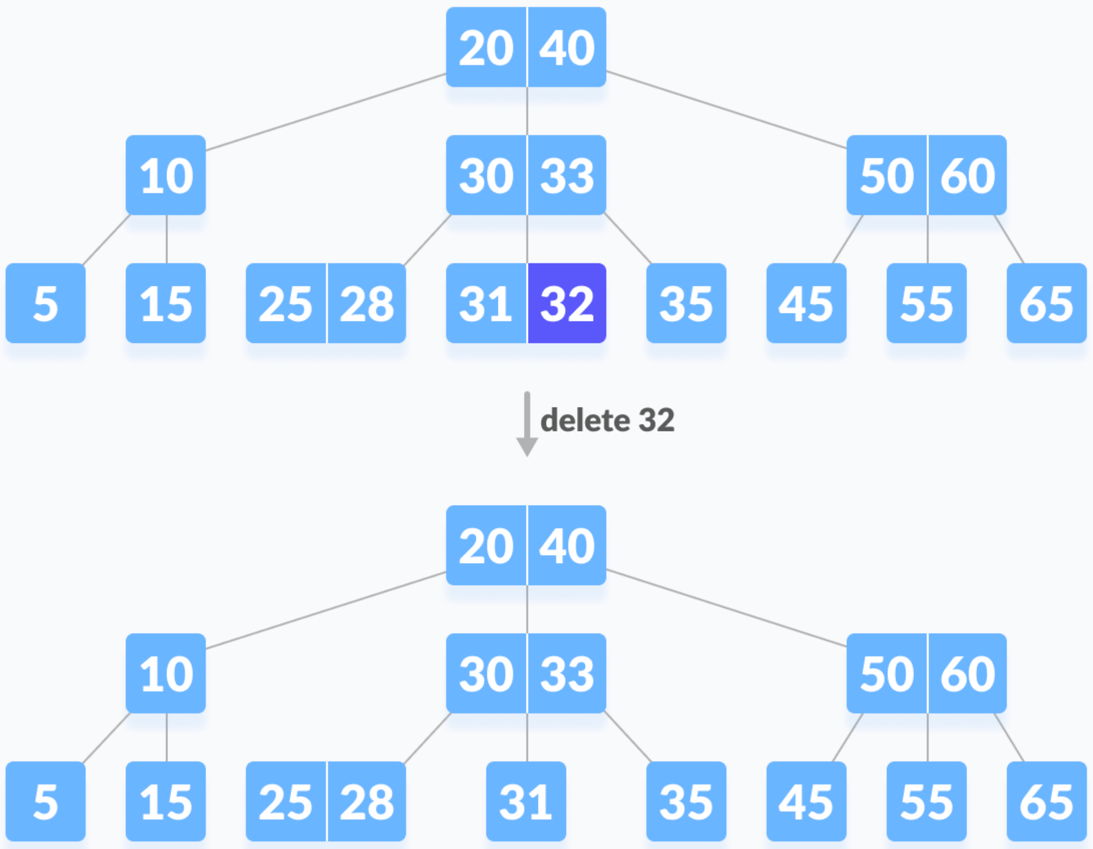
There are three main cases for deletion operation in a B tree.

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.



Deleting a leaf key (32) from B-tree

2. The deletion of the key violates the property of the minimum number of keys a node should hold.

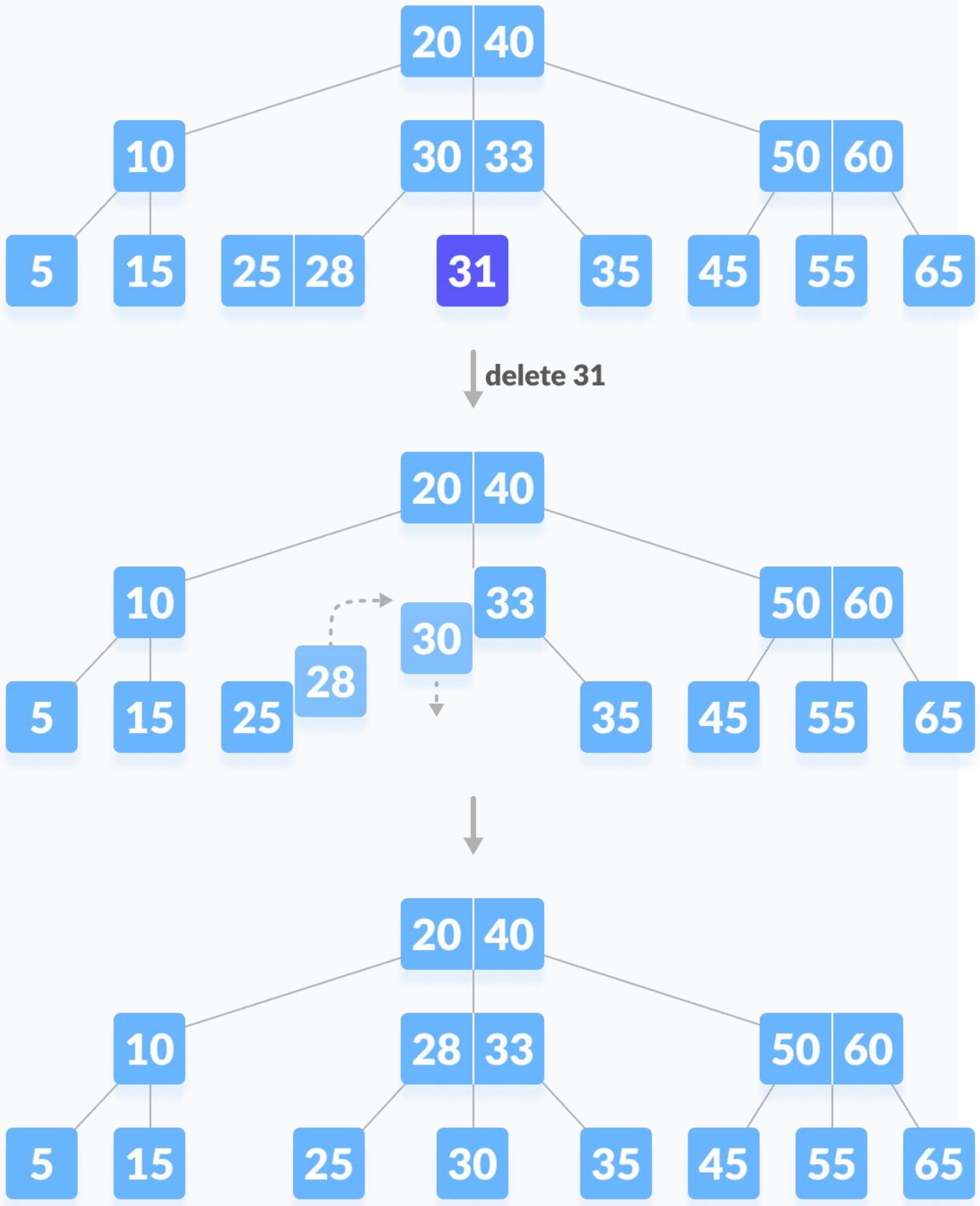
   ✓ In this case, we borrow a key from its immediate neighbouring sibling node in the order of left to right.

     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.

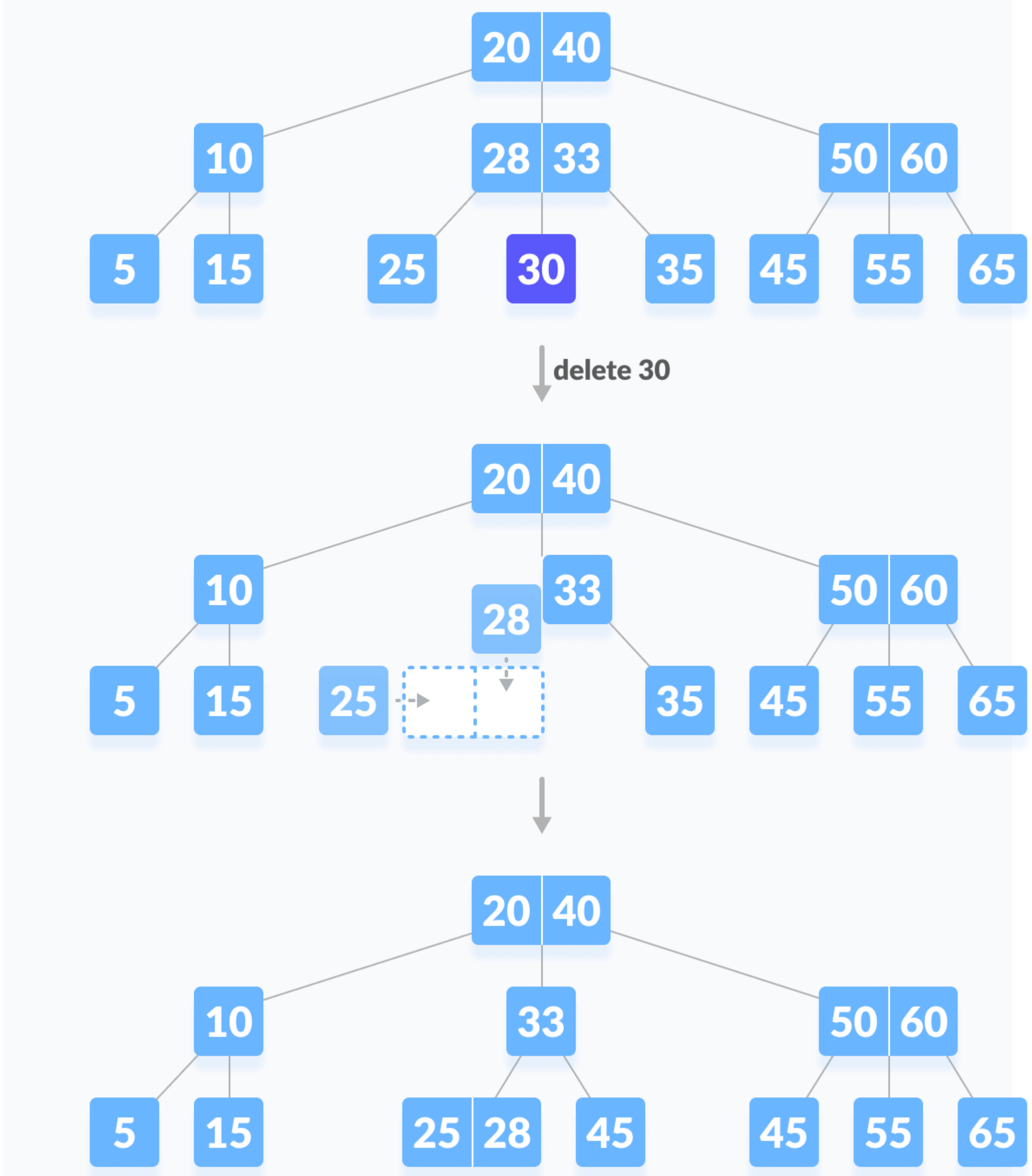
- ✓ Let us borrow a key from the left sibling node.



Deleting a leaf key (31)

- ✓ 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.

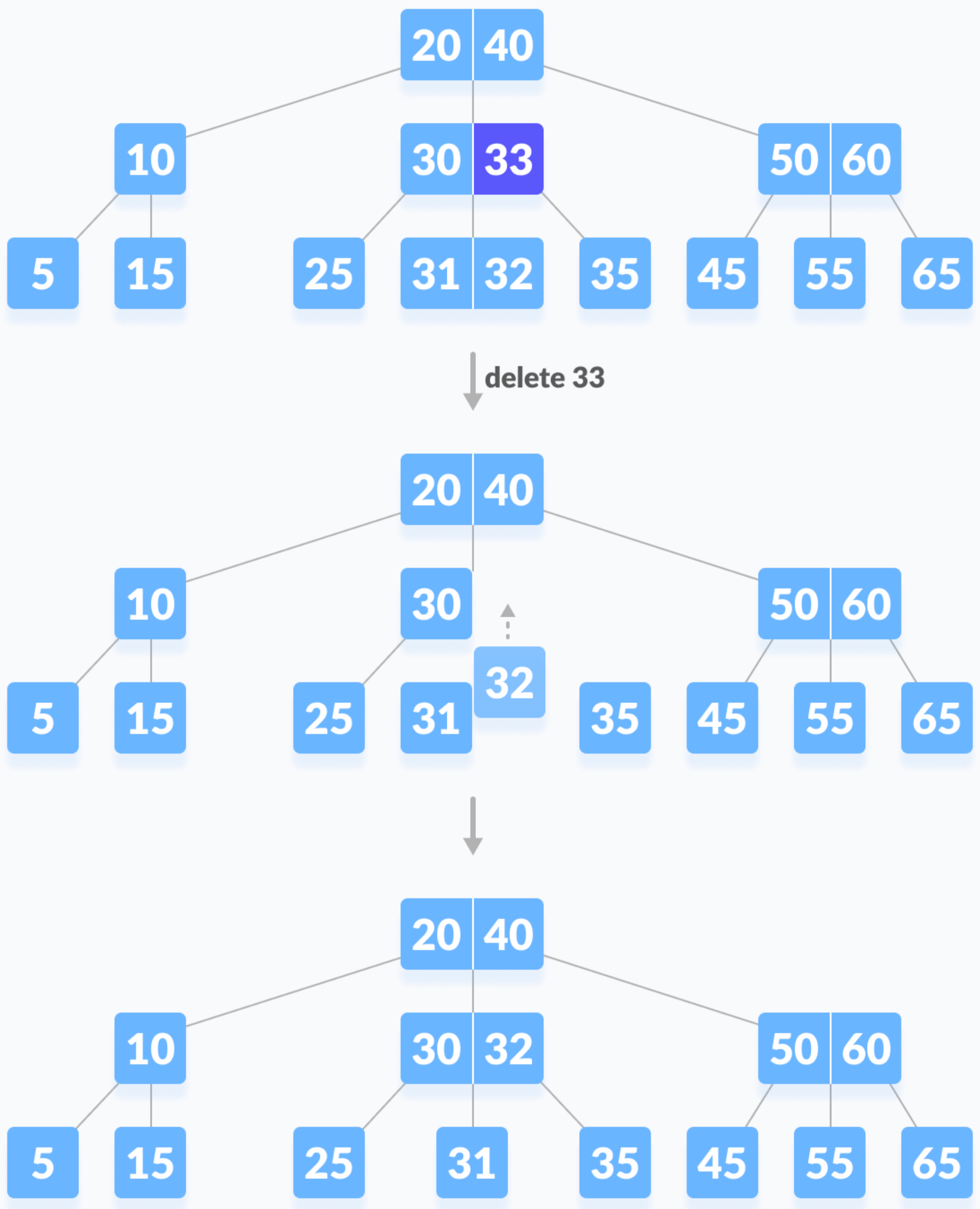


Delete a leaf key (30)

Case II

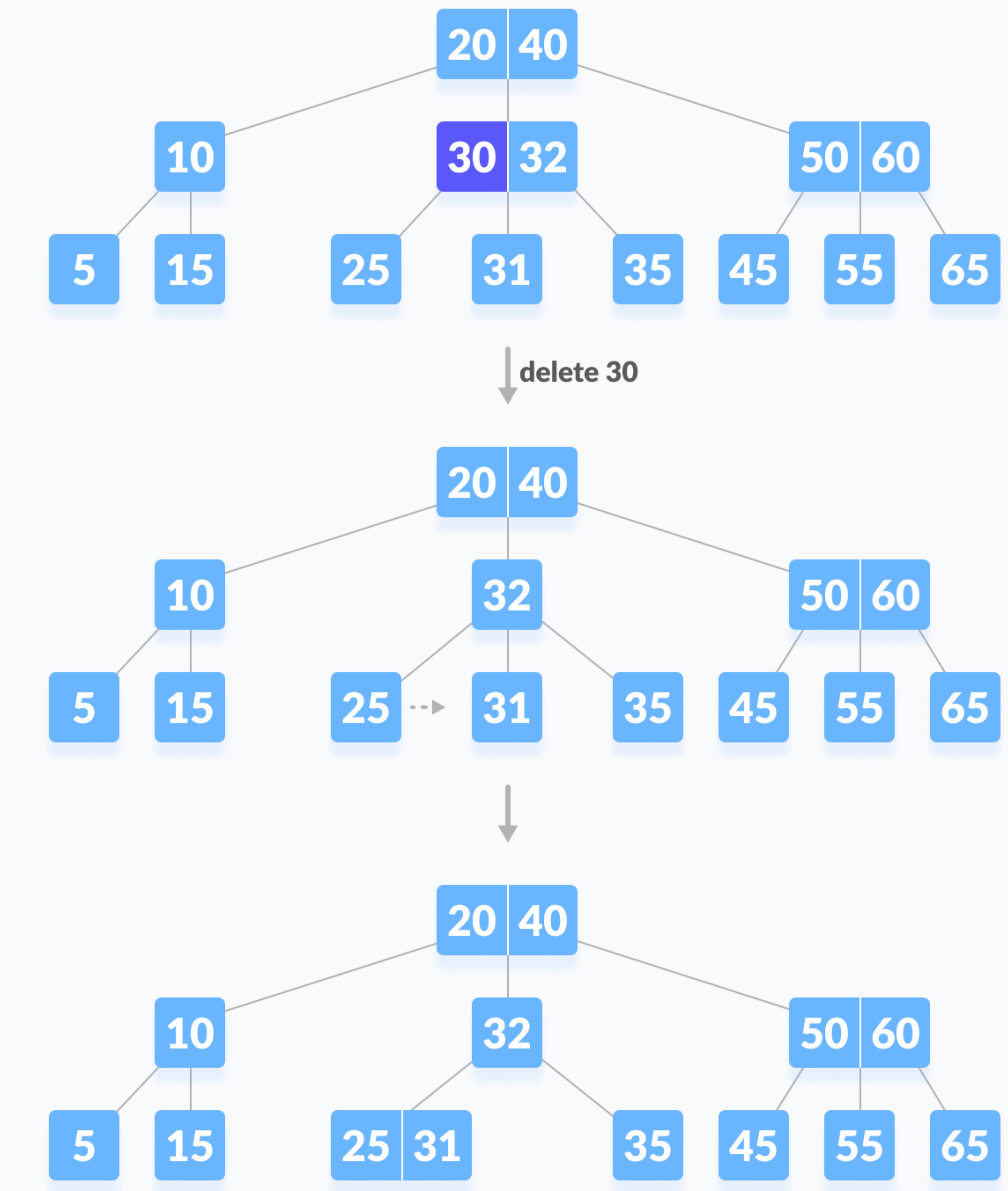
If the key to be deleted lies in the internal node, the following cases occur.

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.



Deleting an internal node (33)

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.



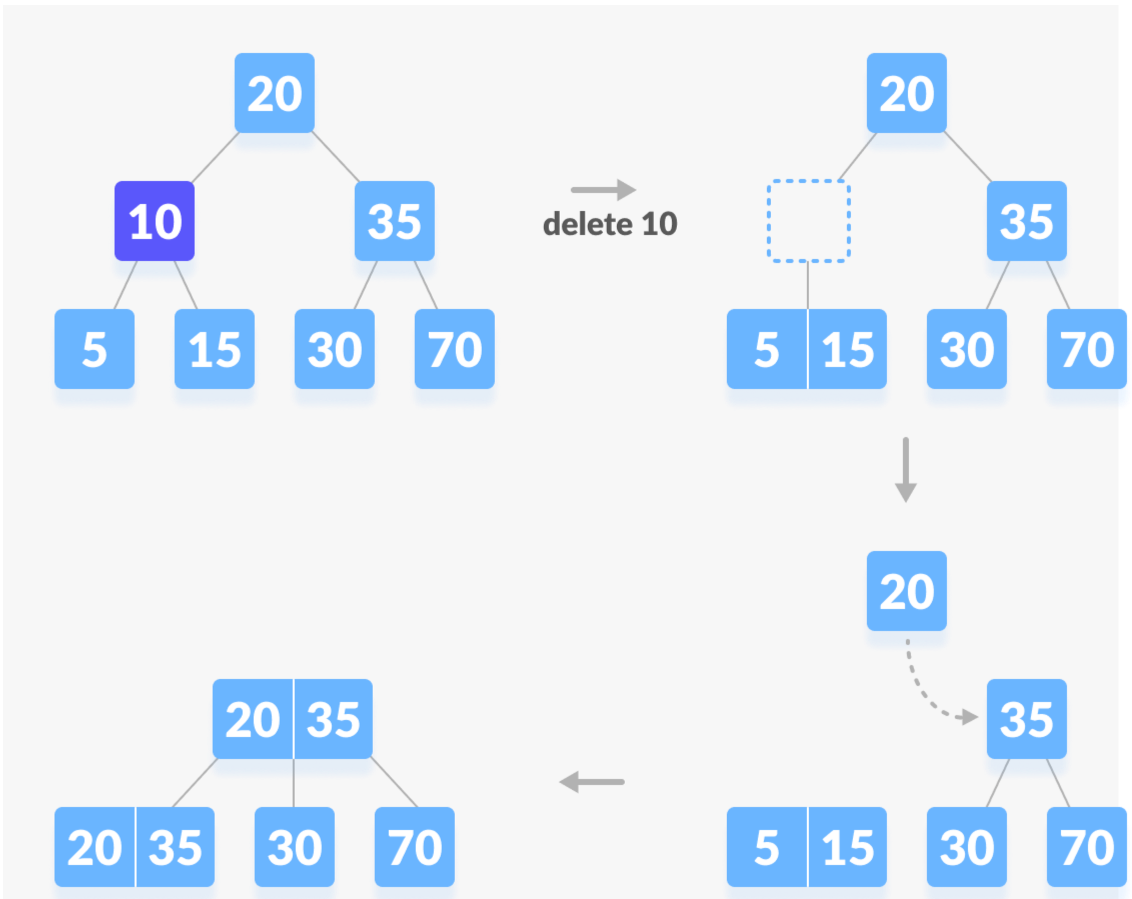
Deleting an internal node (30)

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

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).



Deleting an internal node (10)