B Trees (M-way Trees)

A B-Tree is a special type of M-way search tree.

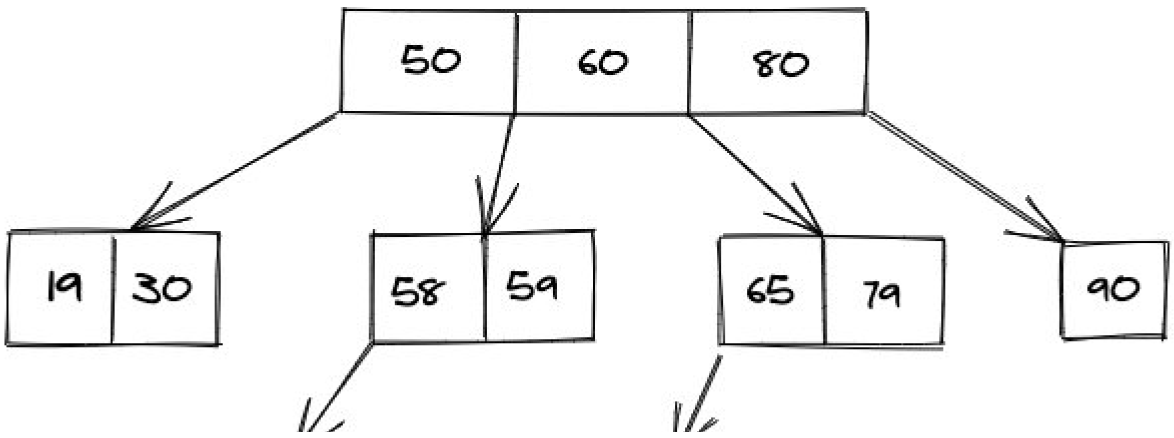
M-way Trees

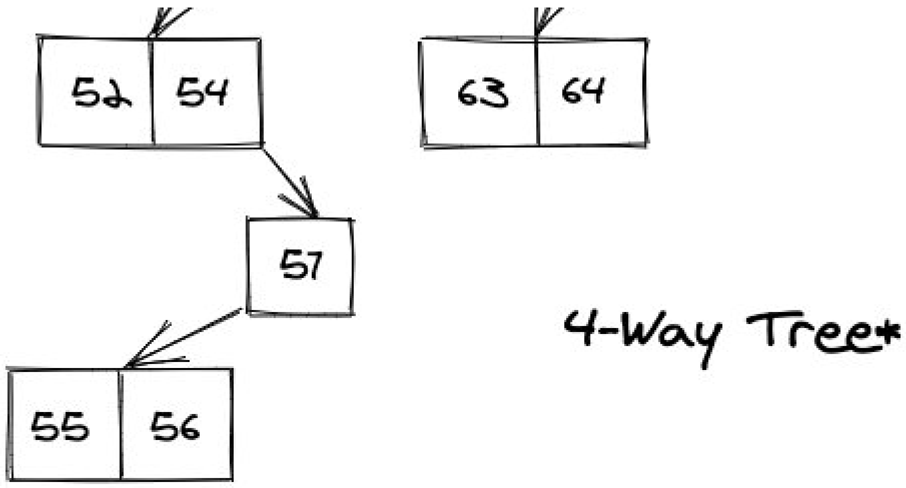
Before learning about B-Trees we need to know what M-way trees are, and how B-tree is a special type of M-way tree. An M-way(multi-way) tree is a tree that has the following properties:

- Each node in the tree can have at most **m** children.
- Nodes in the tree have at most **(m-1)** key fields and pointers(references) to the

≡ Index

Consider the pictorial representation shown below of an **M-way tree**.





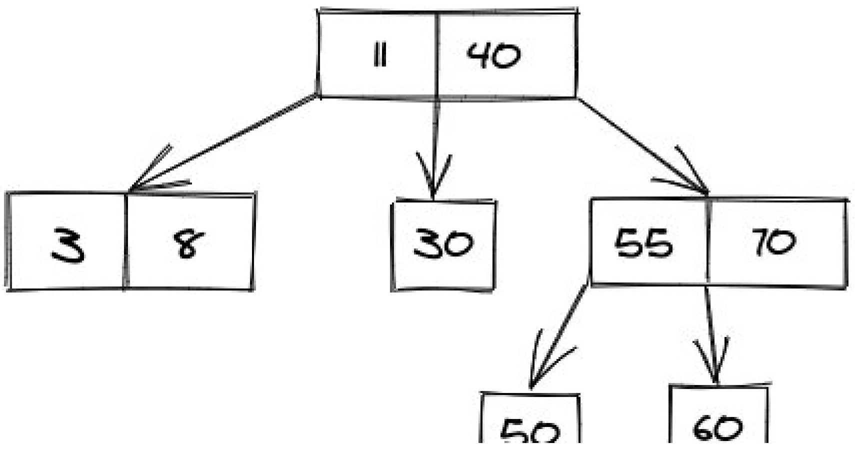
The above image shows a **4-way tree**, where each node can have at most **3(4-1) key fields** and at most **4** children. It is also a **4-way search tree**.

M-way Search Trees

An M-way search tree is a more constrained m-way tree, and these constrain mainly apply to the key fields and the values in them. The constraints on an **M-way** tree that makes it an M-way search tree are:

- Each node in the tree can associate with **m** children and **m-1** key fields.
- The keys in any node of the tree are arranged in a sorted order(**ascending**).
- The keys in the first **K** children are **less than** the **Kth** key of this node.
- The keys in the last **(m-K**) children are higher than the **Kth** key.

Consider the pictorial representation shown below of an M-way search tree:



3-way search tree

M-way search trees have the same advantage over the M-way trees, which is making the search and update operations much more efficient. Though, they can become unbalanced which in turn leaves us to the same issue of searching for a key in a skewed tree which is not much of an advantage.

Searching in an M-way Search Tree:

If we want to search for a value say **X** in an M-way search tree and currently we are at a node that contains key values from **Y1, Y2, Y3,.....,Yk**. Then in total 4 cases are possible to deal with this scenario, these are:

- If **X < Y1**, then we need to recursively traverse the left subtree of **Y1**.
- If **X > Yk**, then we need to recursively traverse the right subtree of **Yk**.
- If **X = Yi**, for some **i**, then we are done, and can return.
- Last and only remaining case is that when for some **i** we have **Yi < X < Y(i+1)**, then in this case we need to recursively traverse the subtree that is present in between **Yi** and **Y(i+1)**.

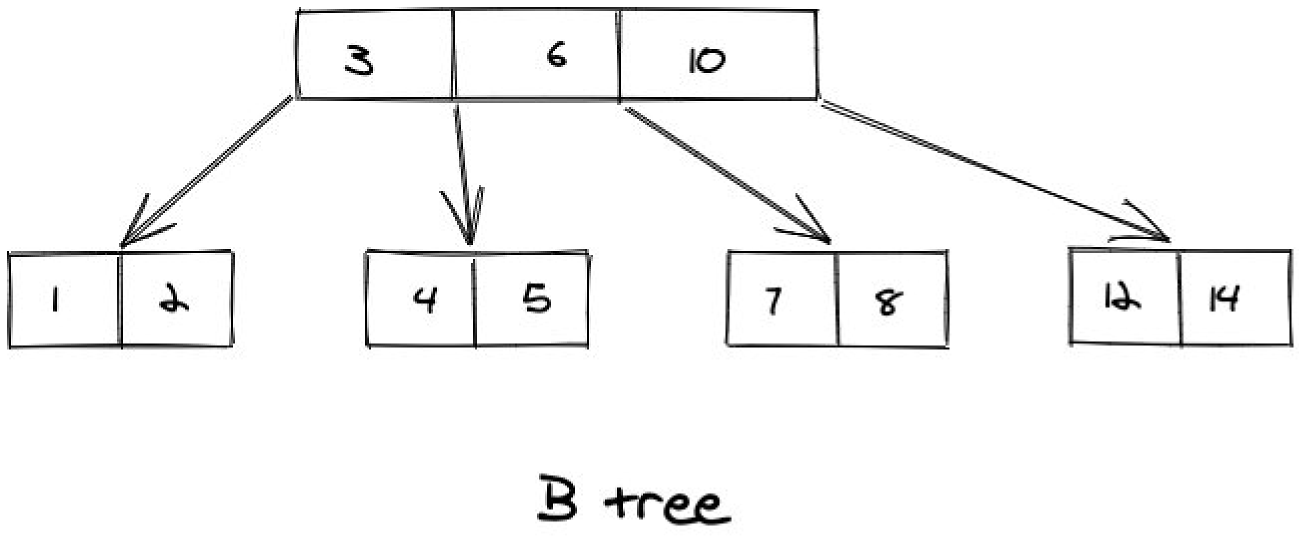
For example, consider the 3-way search tree that is shown above, say, we want to search for a node having key(X) equal to 60. Then, considering the above cases, for the root node, the second condition applies, and **(60 > 40)** and hence we move on level down to the right subtree of **40**. Now, the last condition is valid only, hence we traverse the subtree which is in between the **55** and **70**. And finally, while traversing down, we have our value that we were looking for.

B Trees Data Structure:

A B tree is an extension of an M-way search tree. Besides having all the properties of an M-way search tree, it has some properties of its own, these mainly are:

- All the leaf nodes in a B tree are at the same level.
- All internal nodes must have **M/2** children.
- If the root node is a non leaf node, then it must have at least two children.
- All nodes except the root node, must have at least **[M/2]-1** keys and at most **M-1** keys.

Consider the pictorial representation of a B tree shown below:

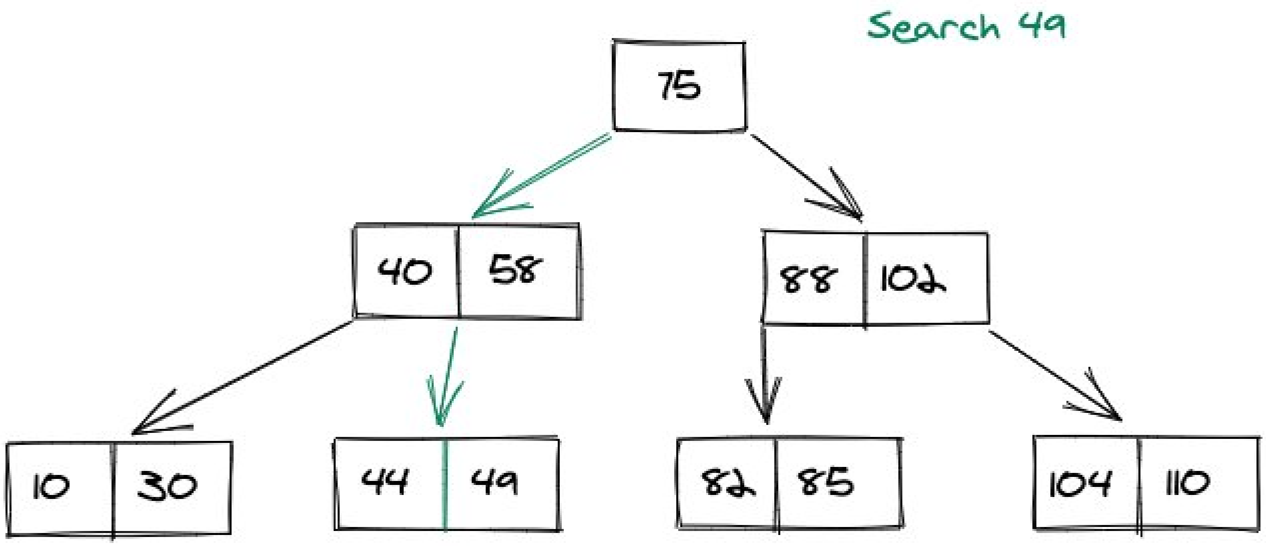


Searching in a B Tree:

Searching for a key in a B Tree is exactly like searching in an M-way search tree, which we have seen just above. Consider the pictorial representation shown below of a B tree, say we want to search for a key 49 in the below shown B tree. We do it as following:

- Compare item **49 with root node 75**. Since **49 < 75** hence, move to its left sub-tree.
- Since, **40<49<58**, traverse right sub-tree of 40.
- **49>44**, move to right. Compare **49**.
- We have found 49, hence returning.

Consider the pictorial representation shown below:

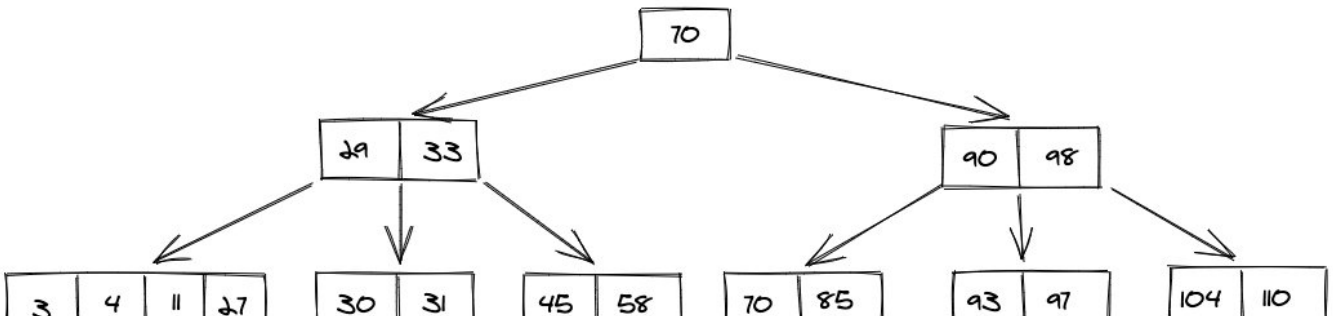


Inserting in a B Tree:

Inserting in a B tree is done at the leaf node level. We follow the given steps to make sure that after the insertion the B tree is valid, these are:

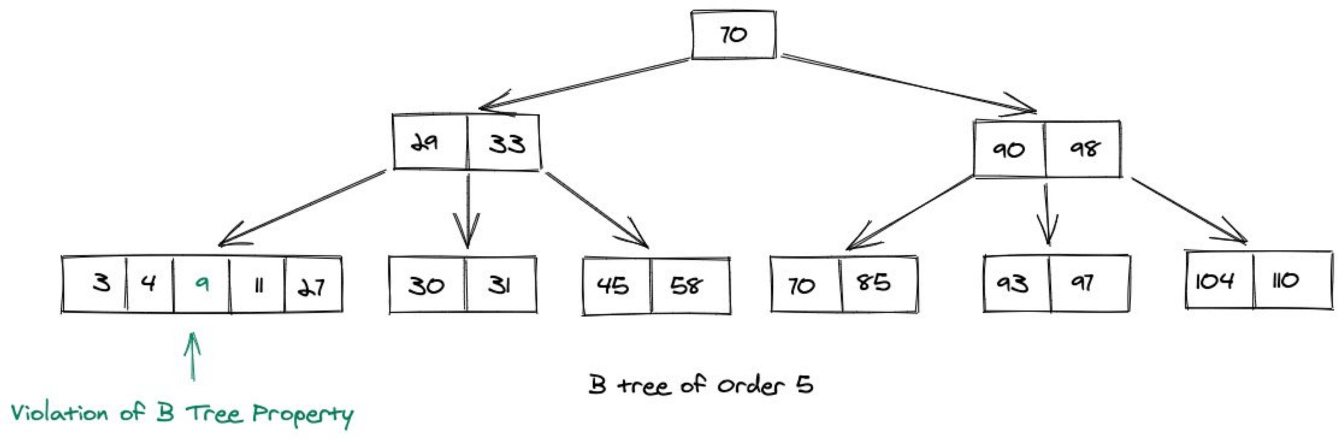
- First, we traverse the B tree to find the appropriate node where the to be inserted key will fit.
- If that node contains less than **M-1** keys, then we insert the key in an increasing order.
- If that node contains exactly **M-1** keys, then we have two cases ? Insert the new element in increasing order, split the nodes into two nodes through the median, push the median element up to its parent node, and finally if the parent node also contains **M-1 keys**, then we need to repeat these steps.

Consider the pictorial representation shown below:



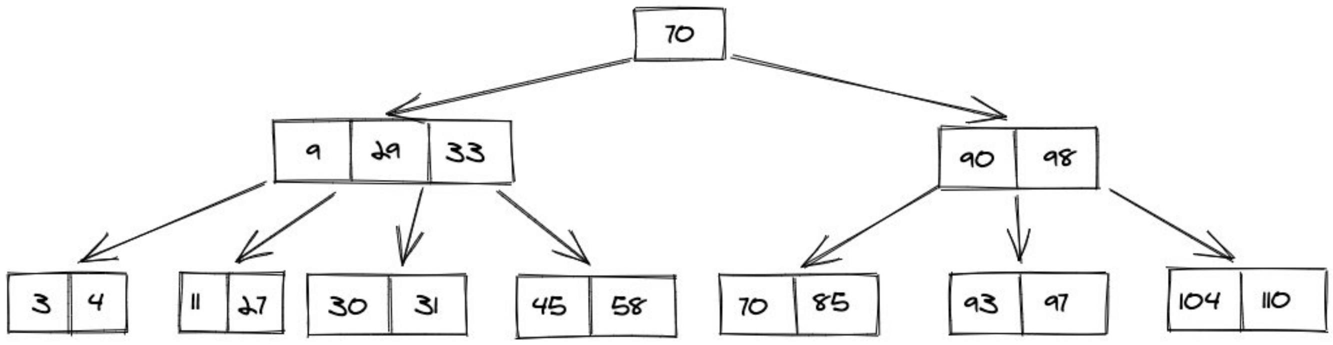
B tree of order 5

Now, consider that we want to insert a key 9 into the above shown B tree, the tree after inserting the key 9 will look something like this:



B tree of order 5

Since, a violation occurred, we need to push the median node to the parent node, and then split the node in two parts, hence the final look of B tree is:



B tree of order 5

Deletion in a B Tree:

Deletion of a key in a B tree includes two cases, these are:

- **Deletion of key from a leaf node**

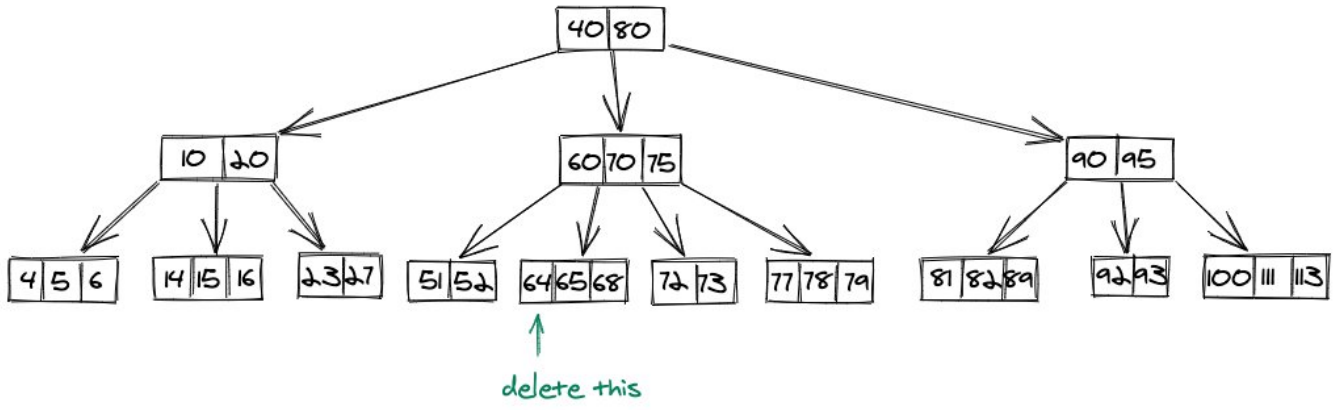
- **Deletion of a key from an Internal node**

Deletion of Key from a leaf node:

If we want to delete a key that is present in a leaf node of a B tree, then we have two cases possible, these are:

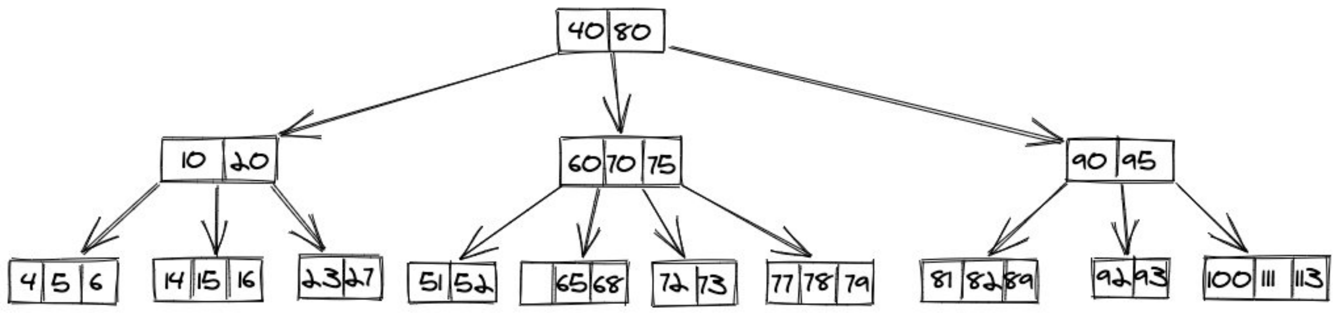
- If the node that contains the key that we want to delete, in turn **contains more than the minimum number of keys required** for the valid B tree, then we can simply delete that key.

Consider the pictorial representation shown below:



Say, we want to delete the key 64 and the node in which 64 is present, has more than minimum number of nodes required by the B tree, which is 2. So, we can simply delete this node.

The final tree after deletion of 64 will look like this:

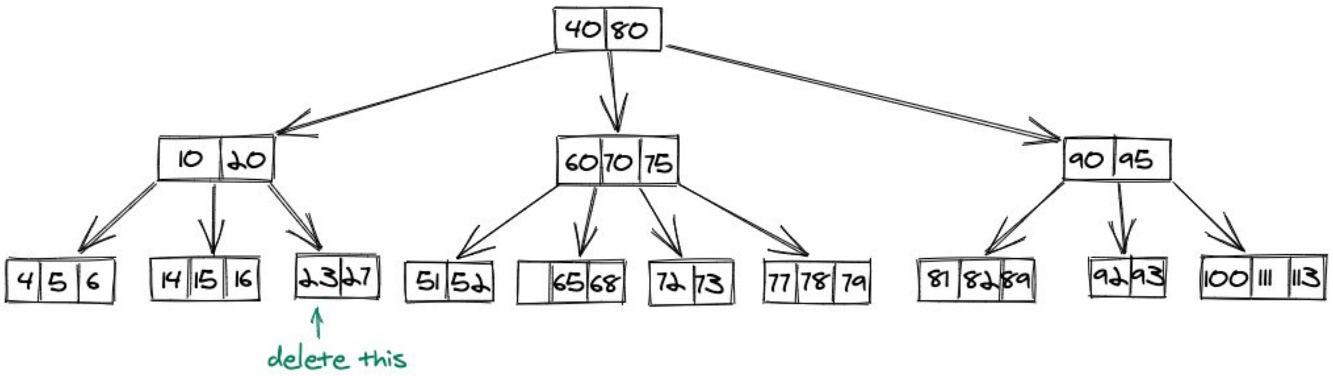


- If the node that contains the key that we want to delete, in turn **contains the minimum**

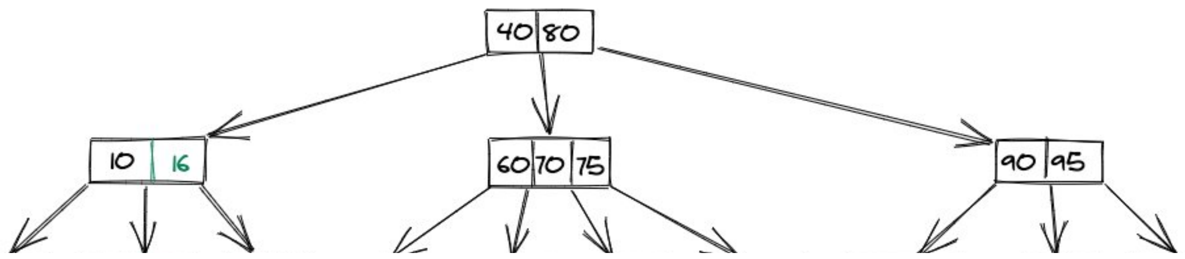
number of keys required for the valid B tree, then three cases are possible:

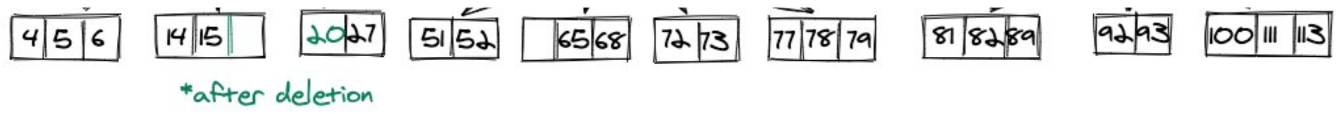
- In order to delete this key from the B Tree, we can borrow a key from the immediate left node(left sibling). The process is that we move the highest value key from the left sibling to the parent, and then the highest value parent key to the node from which we just deleted our key.
- In another case, we might have to borrow a key from the immediate right node(right sibling). The process is that we move the lowest value key from the right sibling to the parent node, and then the highest value parent key to the node from which we just deleted our key.
- Last case would be that neither the left sibling or the right sibling are in a state to give the current node any value, so in this step we will do a merge with either one of them, and the merge will also include a key from the parent, and then we can delete that key from the node.

Case 1 pictorial representation:

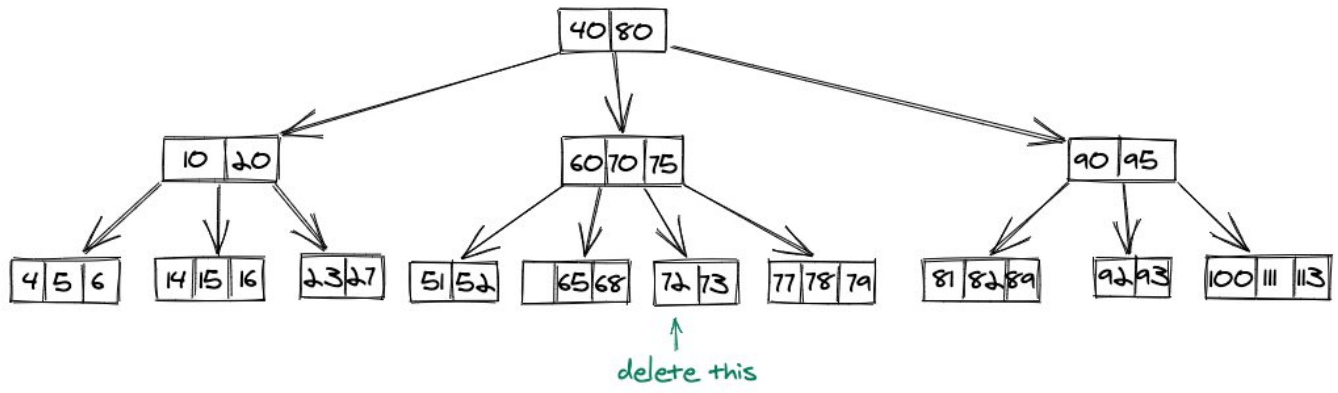


After we delete 23, we ask the left sibling, and then move 16 to the parent node and then push 20 downwards, and the resultant B tree is:

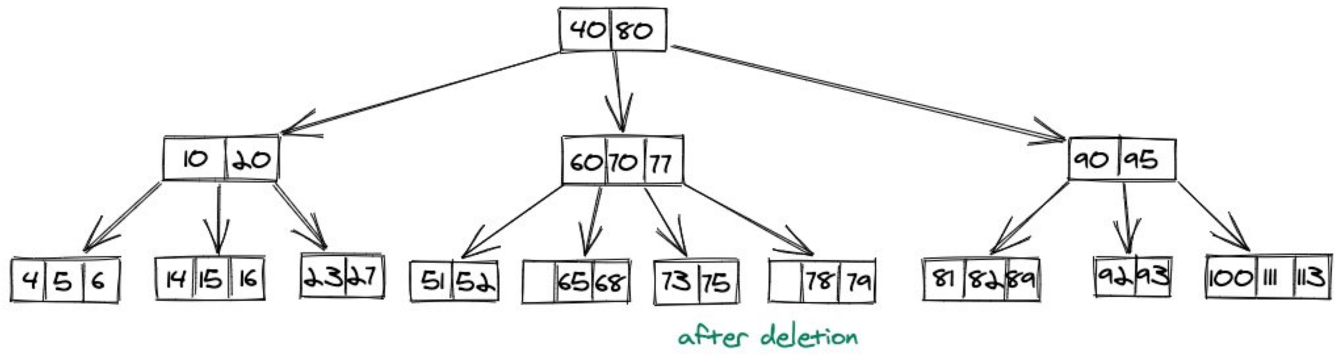




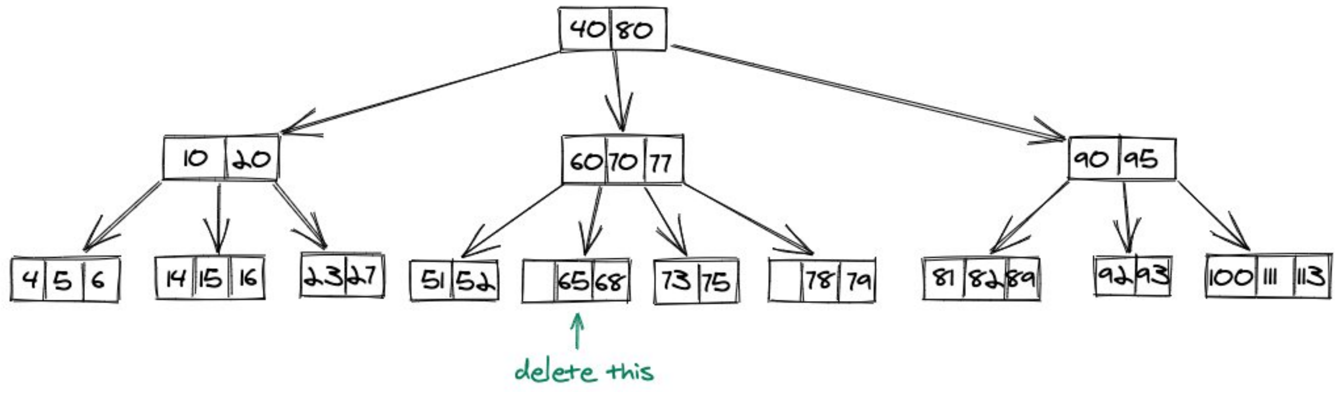
Case 2 pictorial representation:



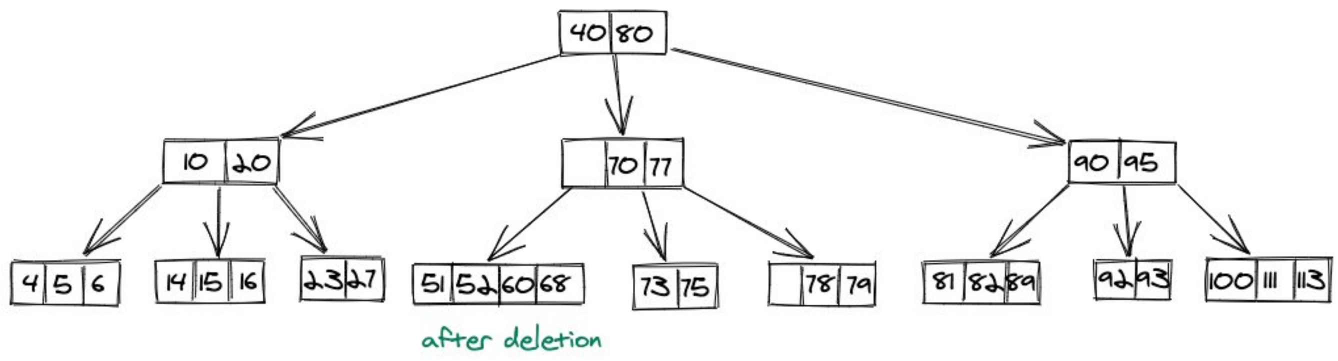
After we delete 72, we ask the right sibling, and then move the 77 to the parent node and then push the 75 downwards, and the resultant B tree is:



Case 3 pictorial representation:



After deleting 65 from the leaf node, we will have the final B tree as:

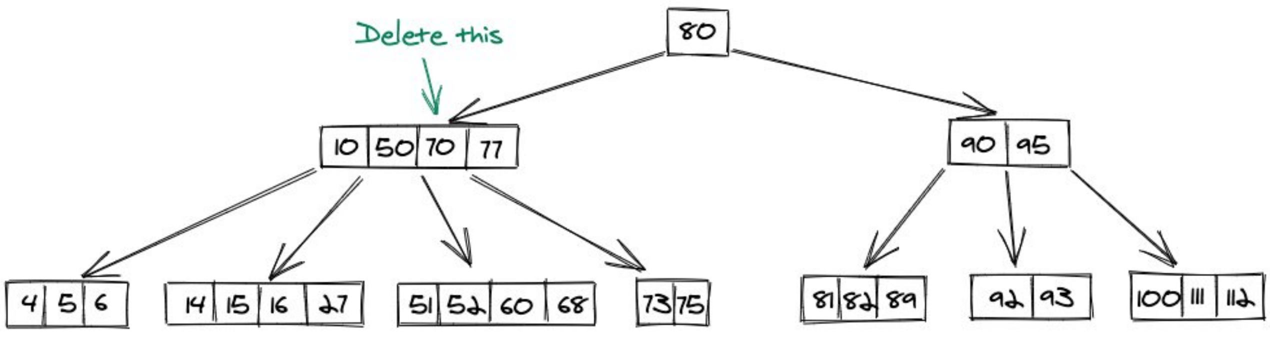


Deletion of Key from an Internal node:

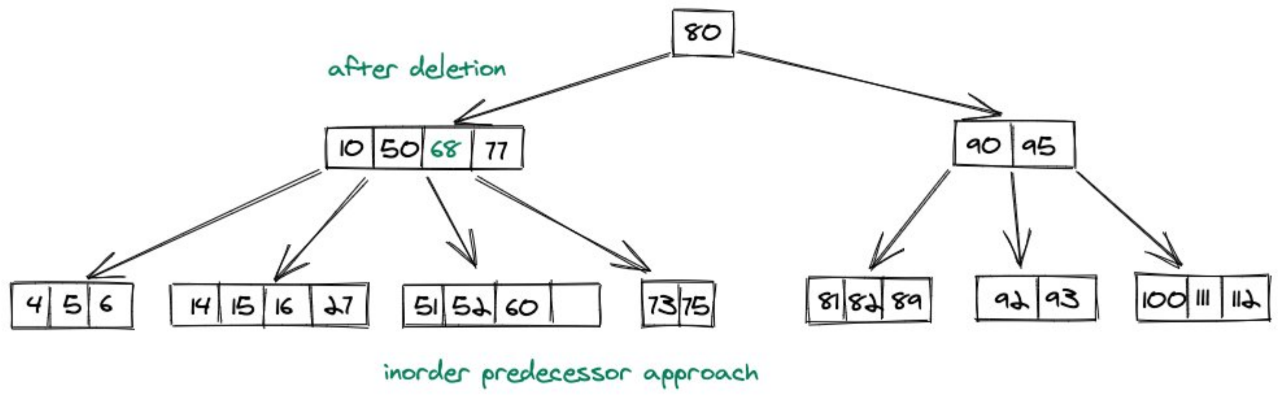
- If we want to delete a key that is present in an internal node, then we can either take the value which is in order **predecessor** of this key or if taking that inorder predecessor violates the B tree property we can take the **inorder successor** of the key.
- In the inorder predecessor approach, we extract the highest value in the left children node of the node where our key is present.
- In the inorder successor approach, we extract the lowest value in the right children node of the node where our key is present.

Pictorial Representation of the above cases:

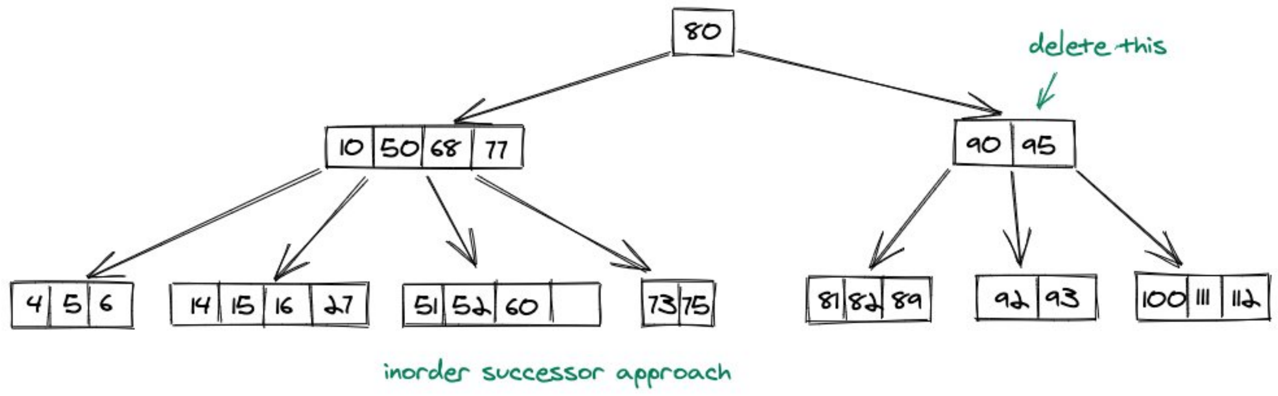
- **Internal predecessor approach**



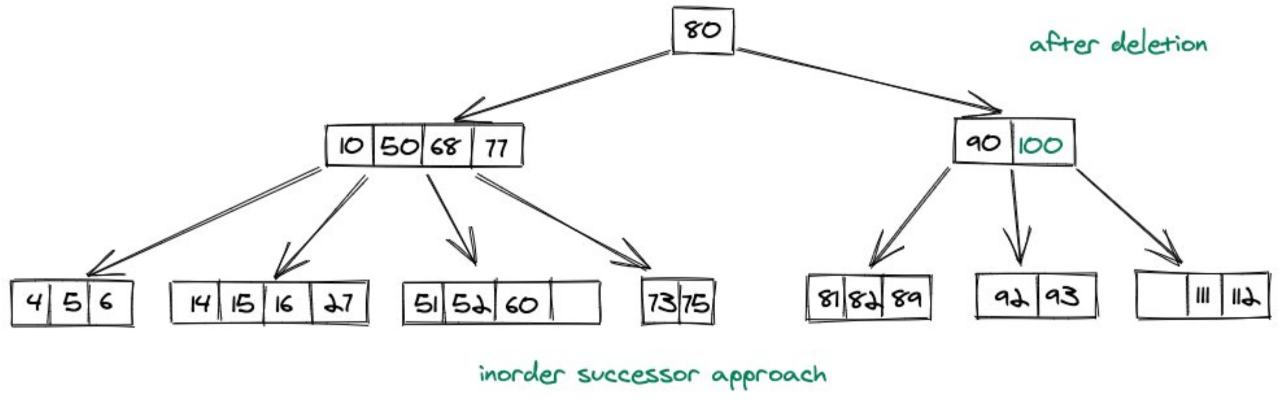
After deletion, our B tree:



- **Internal successor approach**



After deletion of 95, our tree will look like this:



Key Points:

- The time complexity for search, insert and delete operations in a B tree is **O(log n).**
- The minimum number of keys in a B tree should be **[M/2] - 1.**

- The maximum number of keys in a B tree should be **M-1.**
- All the leaf nodes in a B tree should be at the same level.
- All the keys in a node in a binary tree are in increasing order.
- B Trees are used in SQL to improve the efficiency of queries.
- Each node in a B Tree can have at most **M** children.