Operations on 2-3-4 Trees

All the operations take $O(\log n)$ time since the height of the tree is in the logarithmic order.

Search Operation

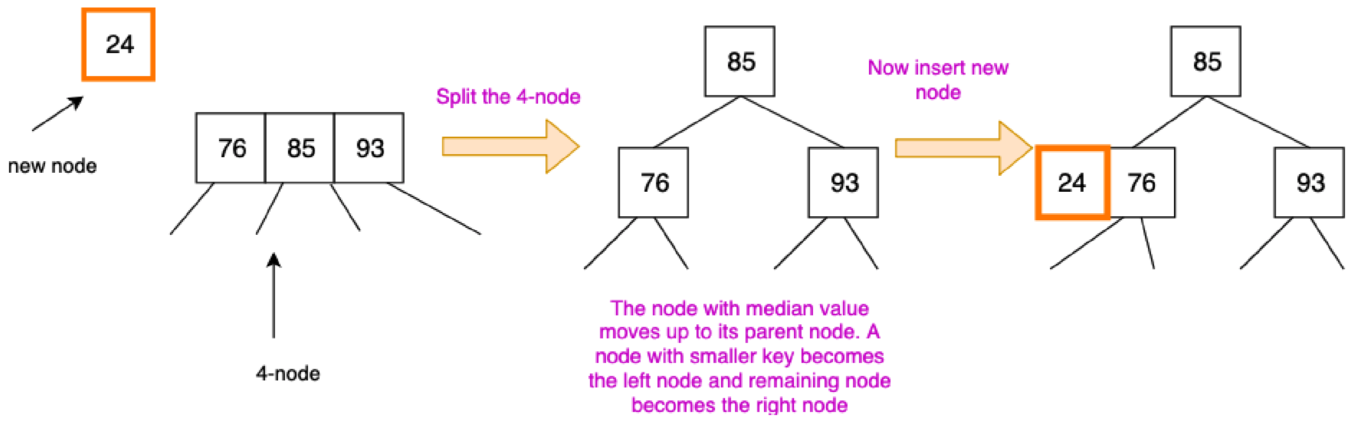
Compare the item to be searched with the keys of the node and move to the appropriate direction. Unlike BST where we move either to the left child or to the right child, we need to make choice among three or four different paths. Since the keys are sorted, it is obvious to choose the path following the rule given in Figure 1.

Insert operation

A node cannot hold more than three keys. If a node is full before insertion, we split the node so that the new node can be inserted.

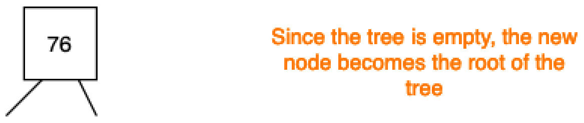
- The insertion takes place always on the leaf nodes.
- again, *we never insert a new node on the internal nodes even if they have room to accommodate*.
- Therefore, we perform the search operation on the tree until we reach the leaf node.
- If the leaf node is a 2-node, we insert the item and make it a 3-node.
- Similarly, if the leaf node is a 3-node, we make it a 4-node. But what if the node is a 4-node?
- We can not insert a new node in this node right? The node is already full. In this case, we split the node that splits into nodes with a smaller number

of keys and inserts the new node in the appropriate child node.

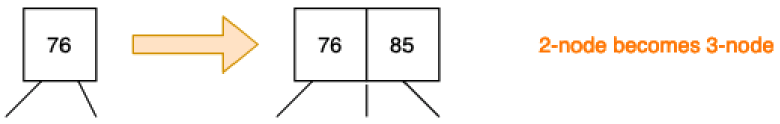

24
new node
76
85
93
4-node
Split the 4-node
85
76
93
The node with median value moves up to its parent node. A node with smaller key becomes the left node and remaining node becomes the right node
Now insert new node
85
24
76
93

The splitting process can, sometimes, go up to the root node. This happens when all the nodes on the path from the root to the leaf node are 4-nodes. When the root node splits, it increases the height of the tree by 1.

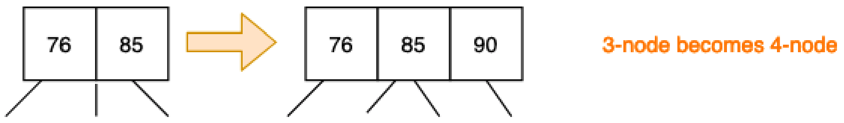
Insert 76


76
Since the tree is empty, the new node becomes the root of the tree

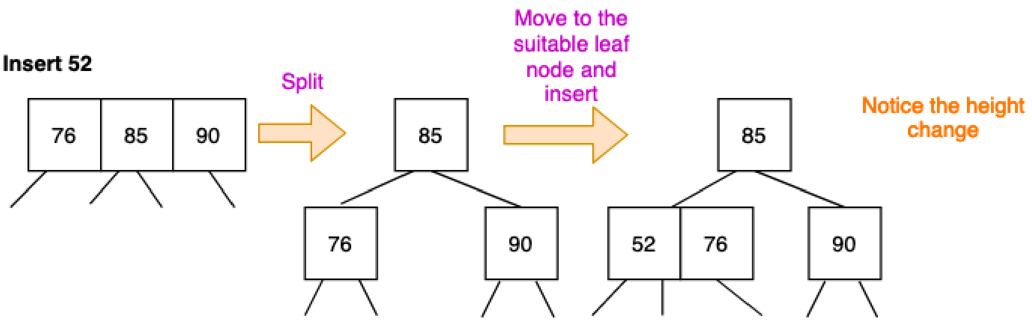
Insert 85



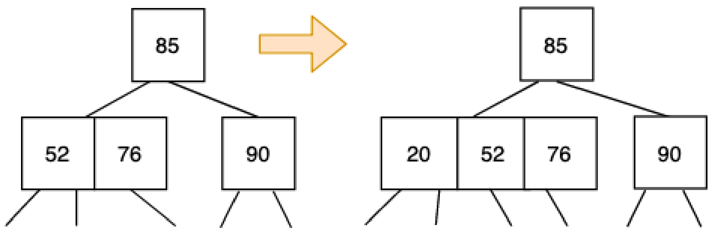
Insert 90



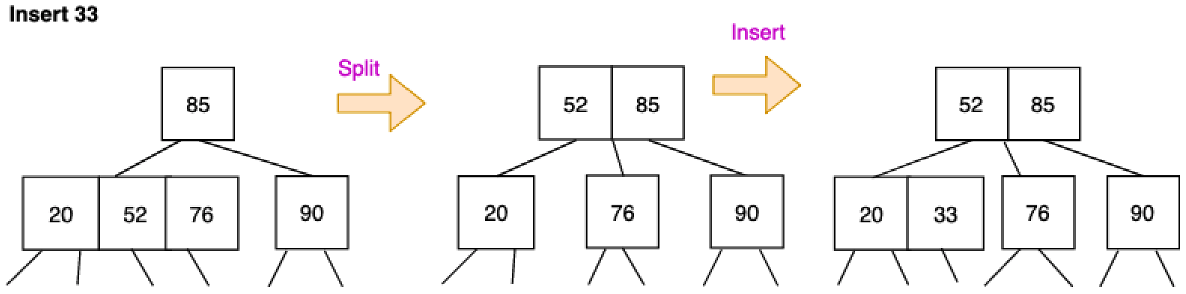
Insert 52



Insert 20



Insert 33



B+ tree of degree **m** PROPERTIES=

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)

Delete Operation

Delete is a bit tricker than insert operation. Depending upon the location of the node containing the target (x) to be deleted, we need to consider several cases. I am going to explain each of the cases one by one.

Case 1: If x is in a leaf node

This has further two cases.

Case 1.1: If x is either in a 3-node or 4-node

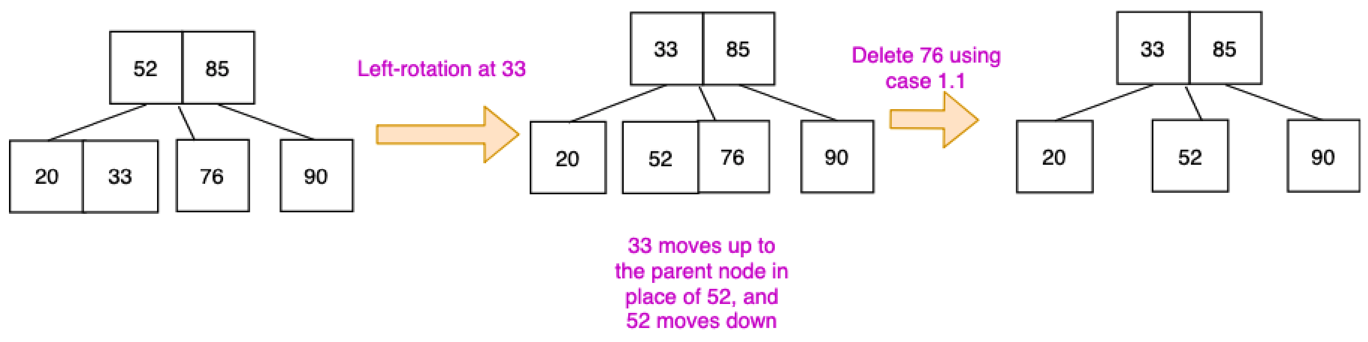
Delete x. If the node is a 3-node, it becomes 2-node and if the node is a 4-node, it becomes 3-node.

Case 1.2: If x is in a 2-node

Underflow can occur. To resolve this, we need to consider further three cases.

Case 1.2.1: If the node containing x has 3-node or 4-node siblings

Convert the 2-node into a 3-node by stealing the key from the sibling. This can be done by left or right rotation. If the left sibling is a 3-node (or 4-node), do the left rotation otherwise do the right rotation.

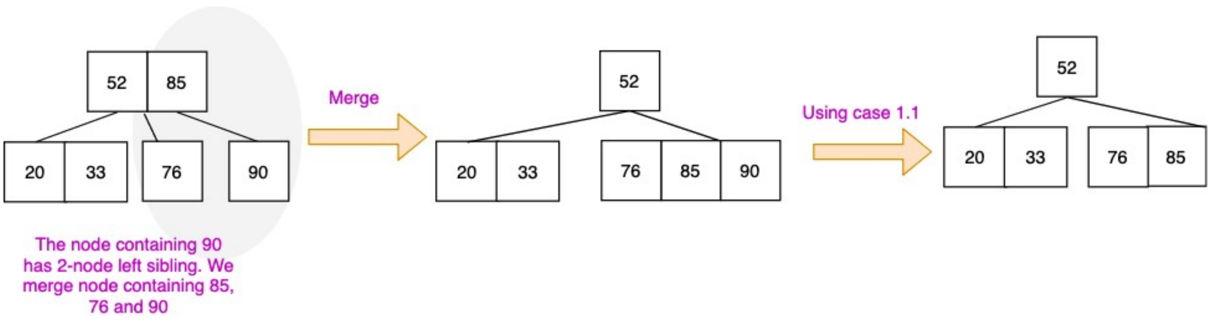


Case 1.2.2: If both the siblings are 2-node but the parent node is either a 3-node or a 4-node

In this case, we convert the 2-node into a 4-node using the merge (or fusion) operation. We merge the following three nodes.

1. The current 2-node containing x
2. The left or right sibling node (which is also a 2-node).
3. The parent node corresponding to these two nodes.

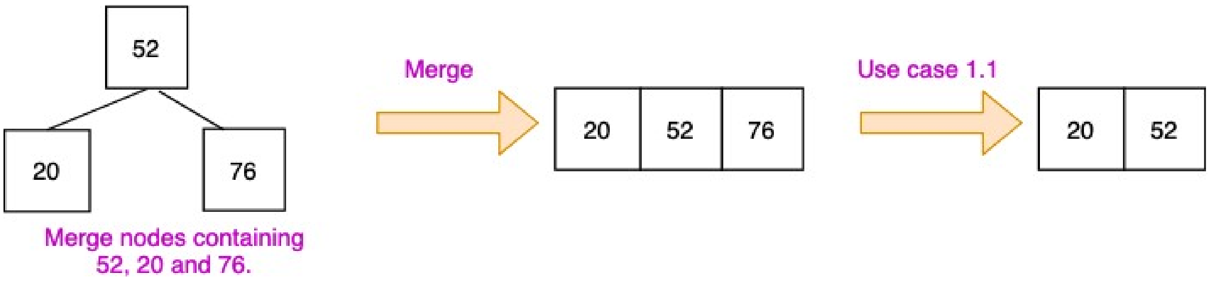
After the fusion, the keys in the parent node decreases by 1. This is illustrated in figure where a 2-node containing 90 is being deleted.


52
85
20
33
76
90
The node containing 90 has 2-node left sibling. We merge node containing 85, 76 and 90
Merge
52
20
33
76
85
90
Using case 1.1
52
20
33
76
85

After the merge, we use case 1.1 to delete x

Case 1.2.3: If both siblings and the parent node are a 2-node

We encounter this scenario rarely. In this particular case, the parent node must be a root. Just like fusion, we combine both the siblings and the parent node to make it a 4-node. This process shrinks the height of the tree by 1. Figure illustrates this. The node containing 76 is being deleted.

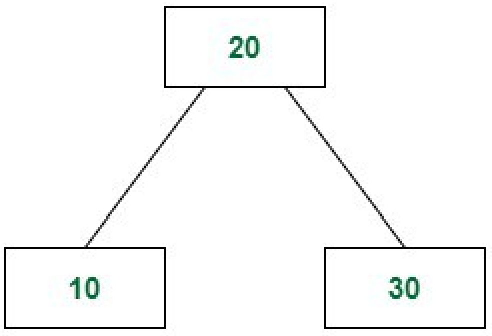


Case 2: If x is in an internal node

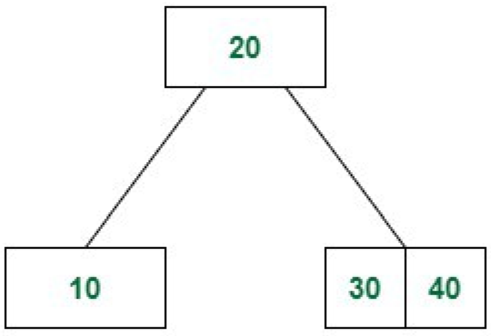
In this case, we do the following

1. Find the predecessor of the node containing x. The predecessor is always a leaf node.
2. Exchange the node containing x with its predecessor node. This moves the node containing x to the leaf node.
3. Since x is in a leaf node, this is case 1. Use the rules given in case 1 to delete it.

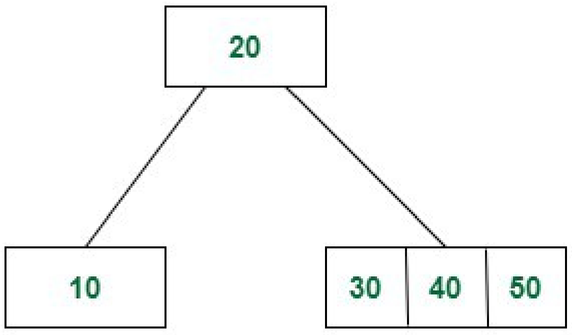
INSERTION



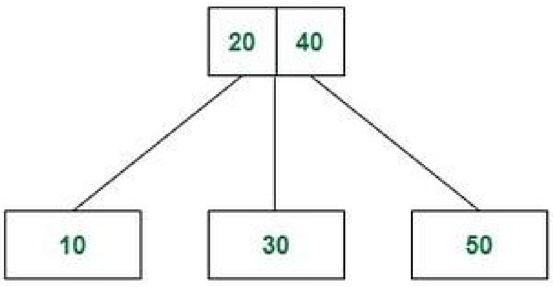
Split the 4 node



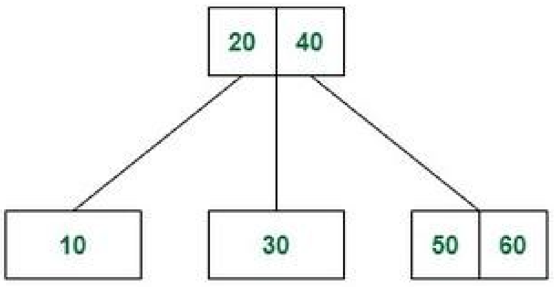
Insert 40



Insert 50

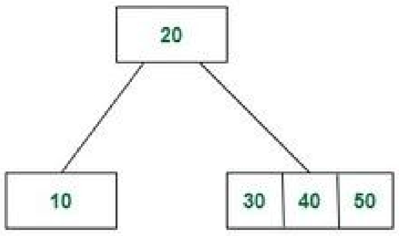


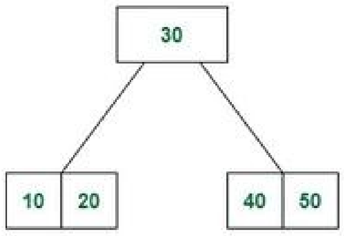
Split the 4 node

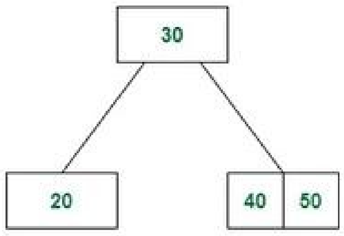


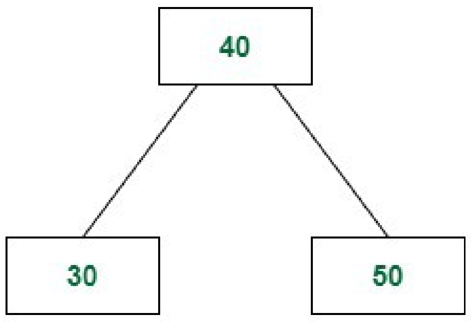
Insert 60

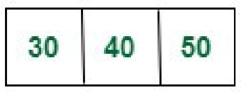
Deletion of a node from 2-3-4 Tree:

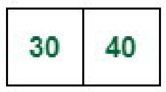


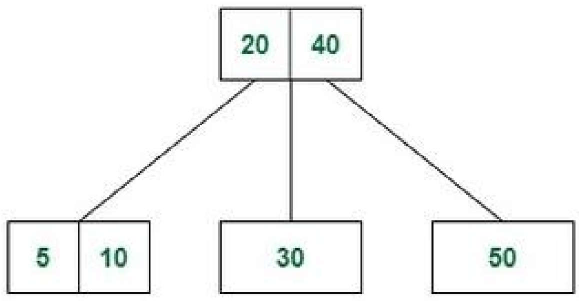

Left rotation

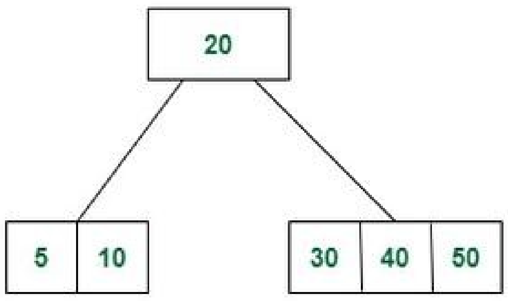

Delete 10




Merge all 2-nodes


Delete 50




Merge two 2-nodes

Complexity Analysis of 2-3-4 trees:

- Searching, insertion, and deletion all take **O(logN)** time complexity in 2-3-4 trees. Since the 2-3-4 is always balanced. By repeating inserting to initialize the 2-3-4 tree, we may say the time cost of init is O(n*log(n)).
- **Height:** In the worst case in **2-3-4 trees** the height is **logN** and in the best case the height is **1/2 * logN** (It is the condition when all nodes are 4 nodes)