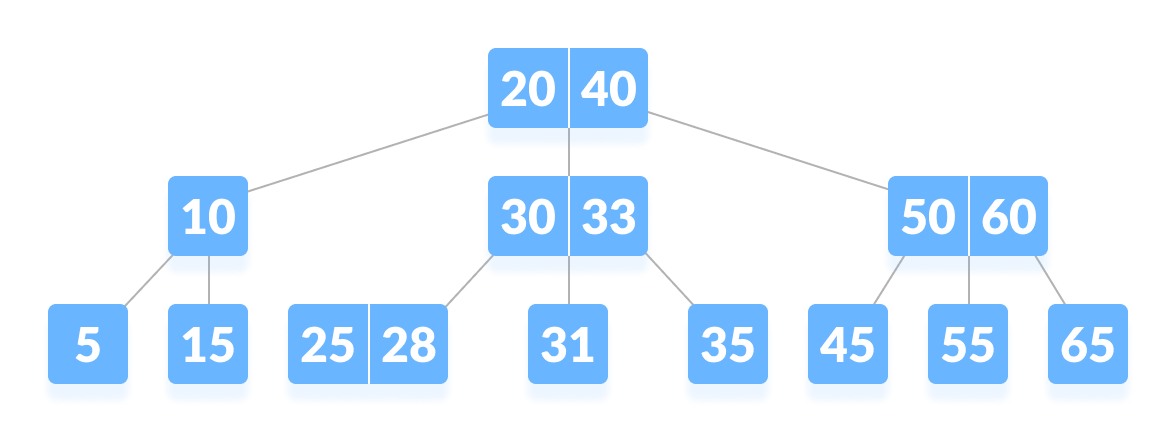
**B trees**

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.

It is also known as a height-balanced m-way tree.



**Rules for B-Tree**

Here, are important rules for creating B\_Tree

* All leaves will be created at the same level.
* B-Tree is determined by a number of degree, which is also called “order” (specified by an external actor, like a programmer), referred to as

**m**

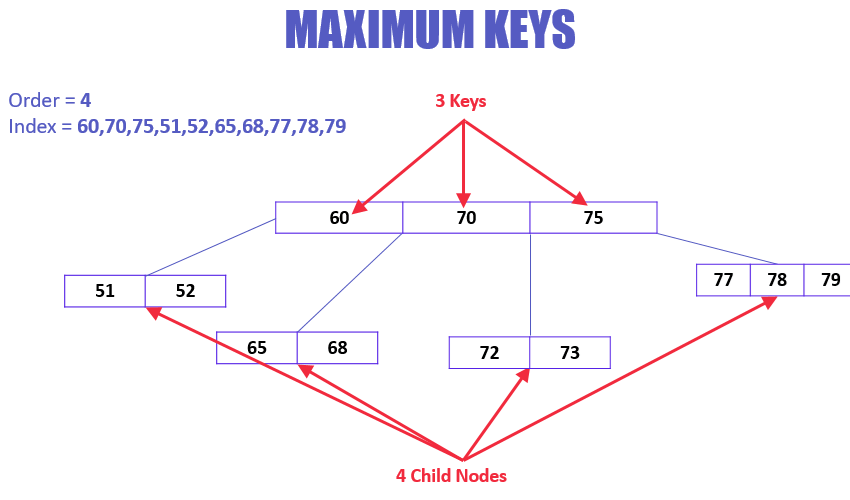
* The left subtree of the node will have lesser values than the right side of the subtree. This means that the nodes are also sorted in ascending order from left to right.
* The maximum number of child nodes, a root node as well as its child nodes can contain are calculated by this formula:

**m – 1**

For example:

**m = 4**

**max keys: 4 – 1 = 3**



Every node, except root, must contain minimum keys of

**[m/2]-1**

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

**m**

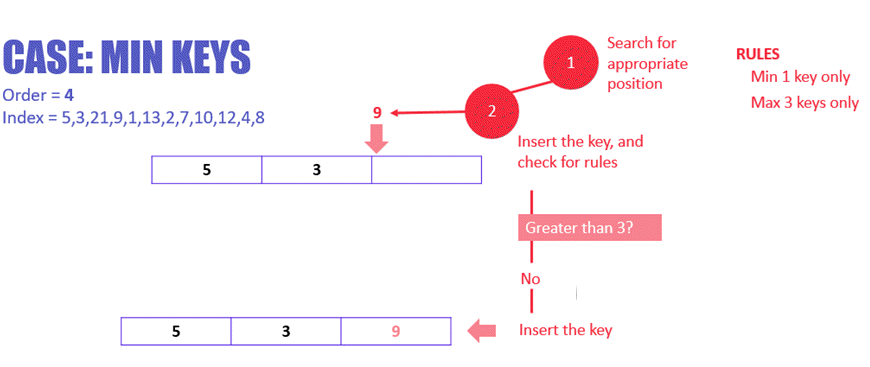
* All the keys in a node are sorted in increasing order.

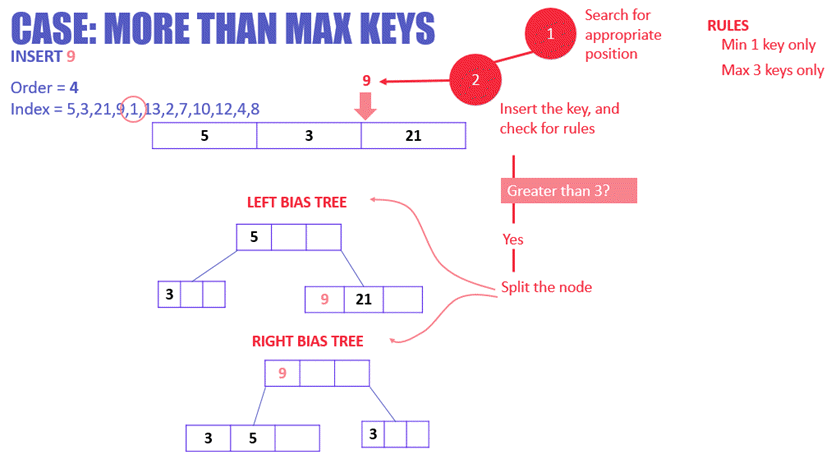
**Insert Operation**

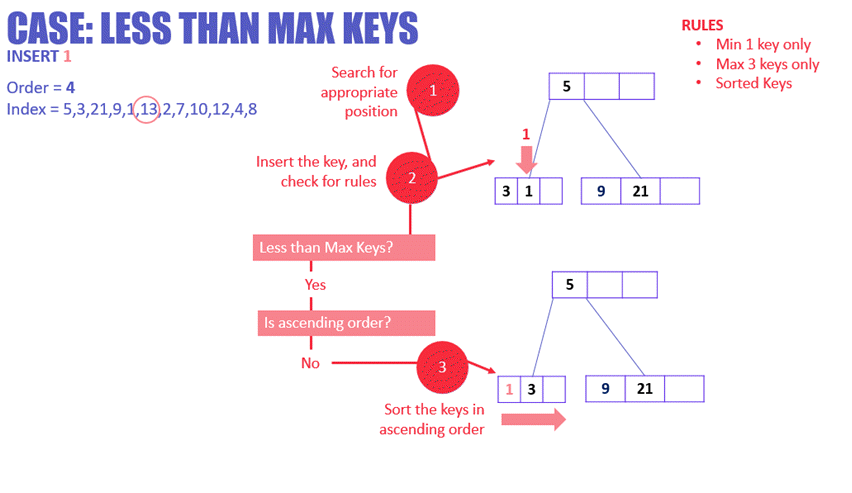
Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.

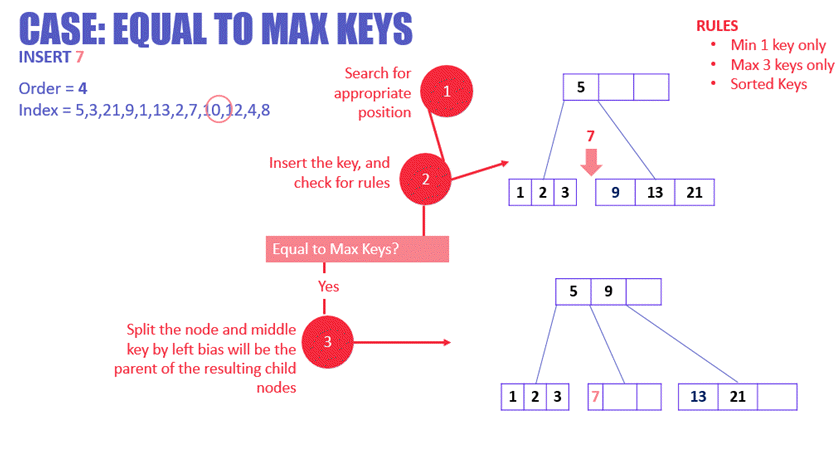
The following algorithm applies:

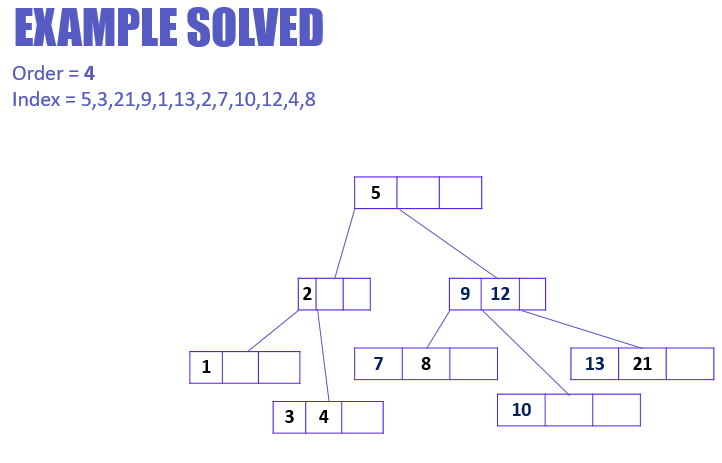
* Run the search operation and find the appropriate place of insertion.
* Insert the new key at the proper location, but if the node has a maximum number of keys already:
* The node, along with a newly inserted key, will split from the middle element.
* The middle element will become the parent for the other two child nodes.
* The nodes must re-arrange keys in ascending order.





****

****

****

**Deletion Operation**

Before going through the steps below, one must know these facts about a B tree of degree **m**.

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.
2. The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.
3. If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

**Case II**

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

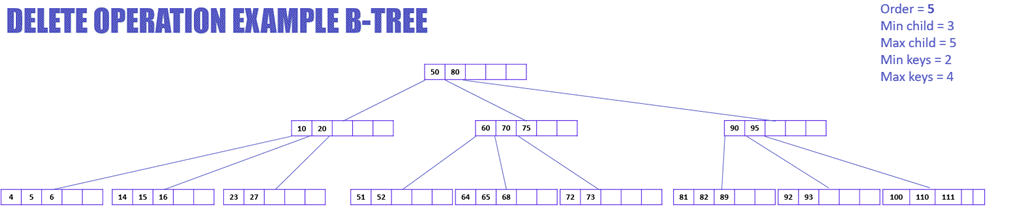
1. The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.
2. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
3. If either child has exactly a minimum number of keys then, merge the left and the right children.

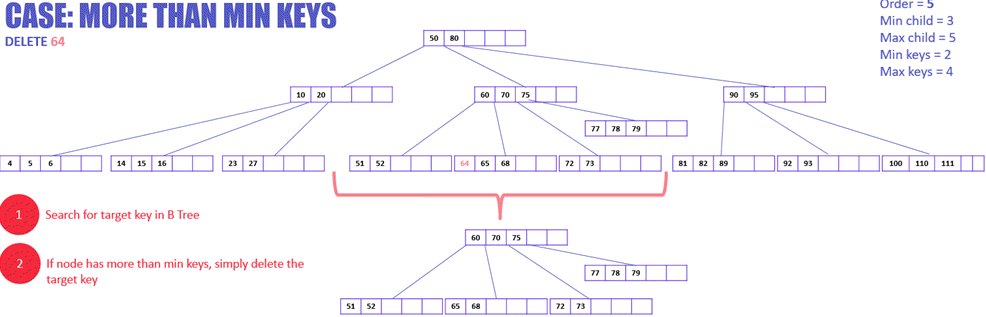
Note: After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

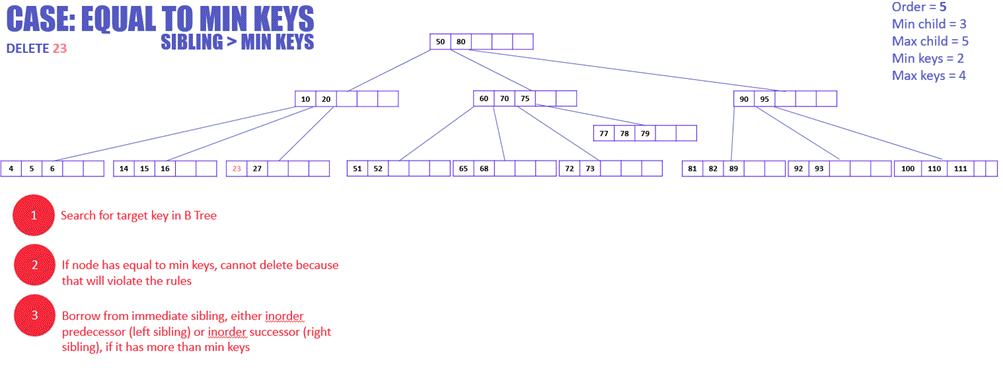
**Case III**

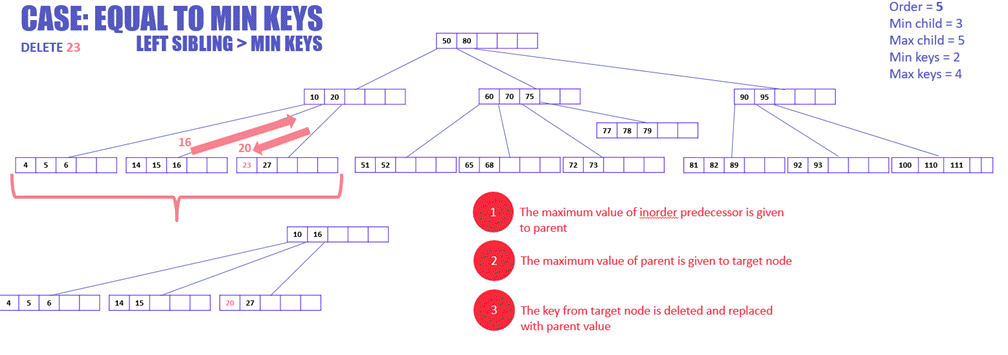
In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

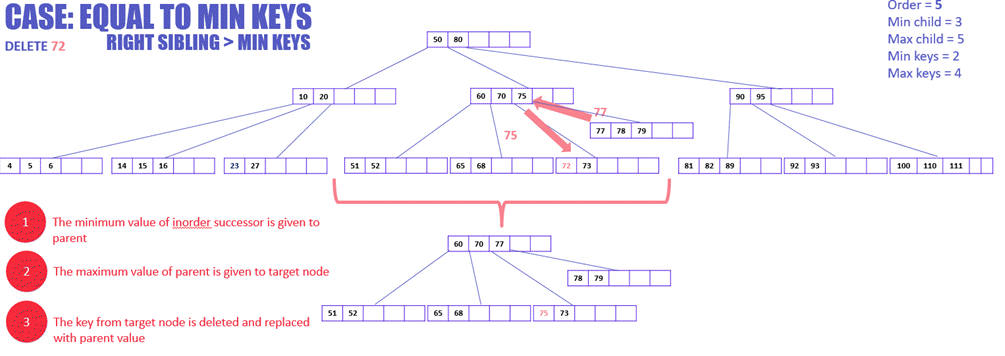
Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).

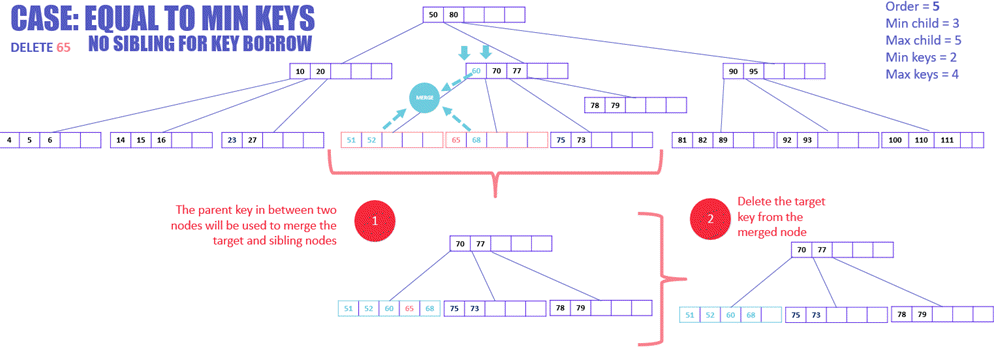












|  |
| --- |
|  **B-Trees** are more general and can vary in order, supporting larger nodes with potentially more than 3 children.   **2-3 Trees** are specifically B-trees of order 3, with each node having only 2 or 3 children and a simpler, more predictable balancing pattern. They’re typically used for simpler data structures or smaller datasets that benefit from straightforward balancing rules. |

2-3 Trees (Search, Insertion, and Deletion)

2-3 Trees

2-3 trees are the data structure same as trees, but it has some different properties like any node can have either single value or double value. So, there are two types of nodes in 2-3 trees:

Single valued

If a node is single-valued then it has two children. Left children will contain values less than parent value, and right children will contain values greater than parent value.

Double valued

If a node has two values then it will have three children. Left children will contain values lesser than the left parent value, and middle children will contain values greater than the left parent value but less than the right parent value. Right children will contain a value greater than the right parent's value. Since, each node has either two children or three children, that's why it is called 2-3 trees. It is a height-balanced tree, and the reason is all the leaf nodes will be at the same level.

Since, it looks like a binary search tree it also has very good time complexity in terms of searching the element. The real-life application of this data structure is into string files in file systems and database systems. In the worst case, a binary search tree can cost O(n) operation if the height of the tree is near equal to the number of elements, but in the case of a 2-3 tree, there will be O(log N) time complexity.

**There are three operations in this tree:**

1. Search

Search is the operation where we are given the root node and target value. If the value is available in the tree, it returns true; else, it will return false.

We can use recursion to search for any element in the tree.

**Case 1:**

If the current node is single-valued and the value is lesser than the node's value, then call the recursive function for the left child. Else, call the recursive function for the right child.

**Case 2:**

If the current node is double valued, and if the value is lesser than the left value, then call the recursive function for the left child. If the target element is greater than the current node's left value and lesser than the current node's right value, then call the recursive function for middle children. Else, call the recursive for the right child.

**Base case:**

As we know, the recursion base case is the most important condition which helps to terminate the recursive calls. If the current node is null, then we will return false, or if the current node's value is equal to the target element, then we will return true.

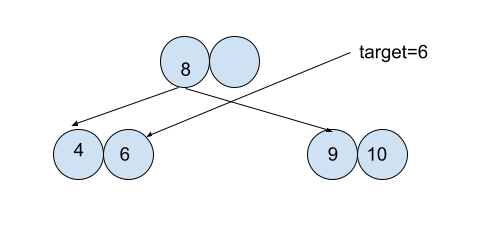
2. Insertion

If we want to insert any element in the tree, then we will find its correct position, and then we will insert it. We can have three cases in the insertion operation:

**Case1:**

Suppose the node at which we want to put the element contains a single value. In this case, we will simply insert the element.

**For example:**

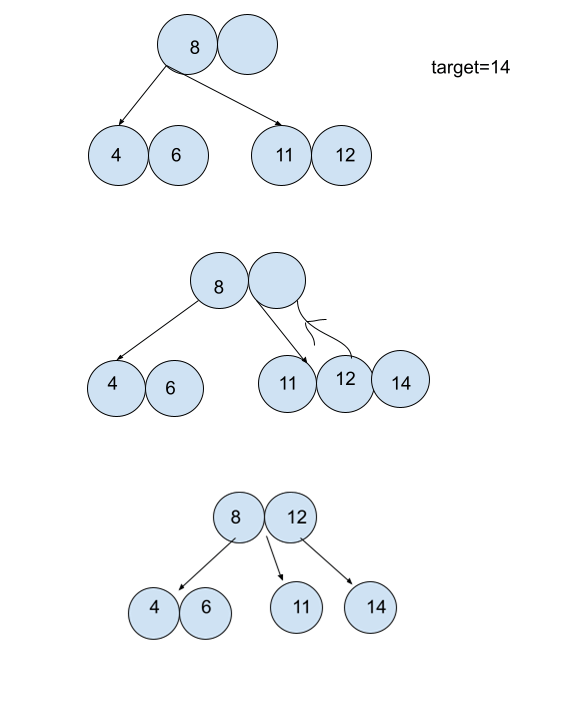


In the above example, we simply put the target in its correct position.

**Case2:**

If the node at which we want to insert the target element contains the double value and its parent is single-valued, then we will put the element at the node, and the middle value of the node will be shifted to its parent. And the current node will be split into two separate nodes.

**For example:**



**Explanation**

In the above example, we have target element 14, and the node where it will be inserted already has two values which are 11 and 12. So in the first step, we will insert the value at its correct position, and now our node has three values which are 11, 12 and 14, which should not be done. So the middle value is 12, and its parent has a single value, which means we can insert one more value into its parent node. So we will insert 12 to the parent node and split the current node into two nodes where one node will have a value of 11, and another node will have a value of 14.

**Case 3:**

If the node at which we want to insert is a double-valued node and its parent is also a double-valued node. We will shift the middle element to the parent node and split the current node. Now its parent has three values, so it will also shift its middle element to its parent node and split the parent node into two separate nodes.

3. Deletion

A value is removed after being replaced by its in-order successor in order to be deleted. Two nodes must be combined together if a node has less than one data value remaining. After removing a value, a node is merged with another if it becomes empty.