B-tree data structure

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-tree Properties

1. For each node *x*, the keys are stored in increasing order.
2. If *n* is the order of the tree, each internal node can contain at most *n - 1* keys along with a pointer to each child. and at least **ceiling[n/2]**-1 keys.
3. Each node except root can have at most n children and at least **ceiling[n/2]** children.
4. All leaves have the same depth (i.e. height-h of the tree).
5. The root has at least 2 children and contains a minimum of 1 key.

Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

1. Starting from the root node, compare k with the first key of the node.
   If `k = the first key of the node`, return the node and the index.
2. If `k.leaf = true`, return *NULL* (i.e. not found).
3. If `k < the first key of the root node`, search the left child of this key recursively.
4. If there is more than one key in the current node and `k > the first key`, compare k with the next key in the node.
   If `k < next key`, search the left child of this key (ie. k lies in between the first and the second keys).
   Else, search the right child of the key.
5. Repeat steps 1 to 4 until the leaf is reached.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

insertion

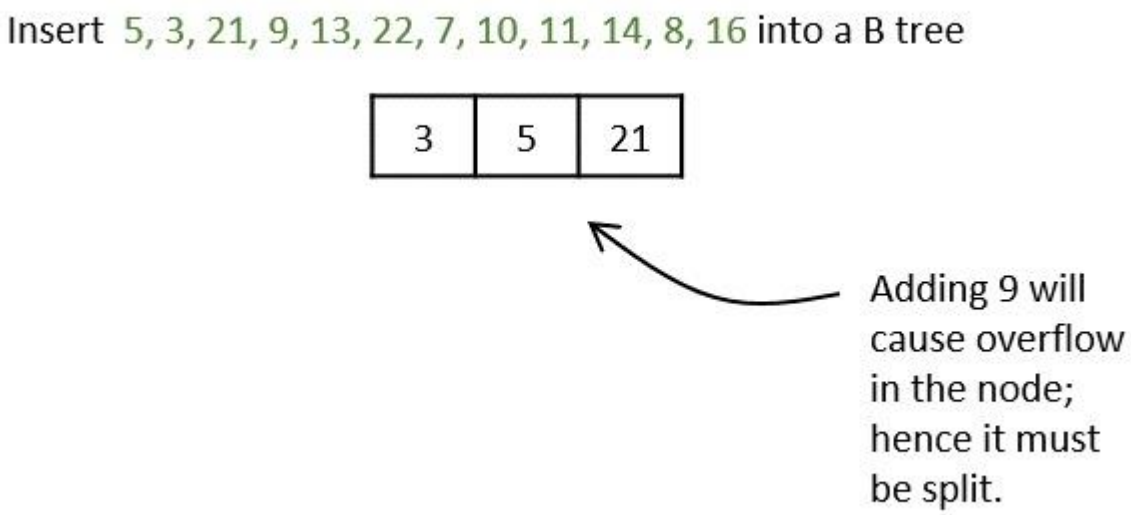
The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure −

Step 1 − Calculate the maximum\and minimum \$ number of keys a node can hold, where m is denoted by the order of the B Tree.

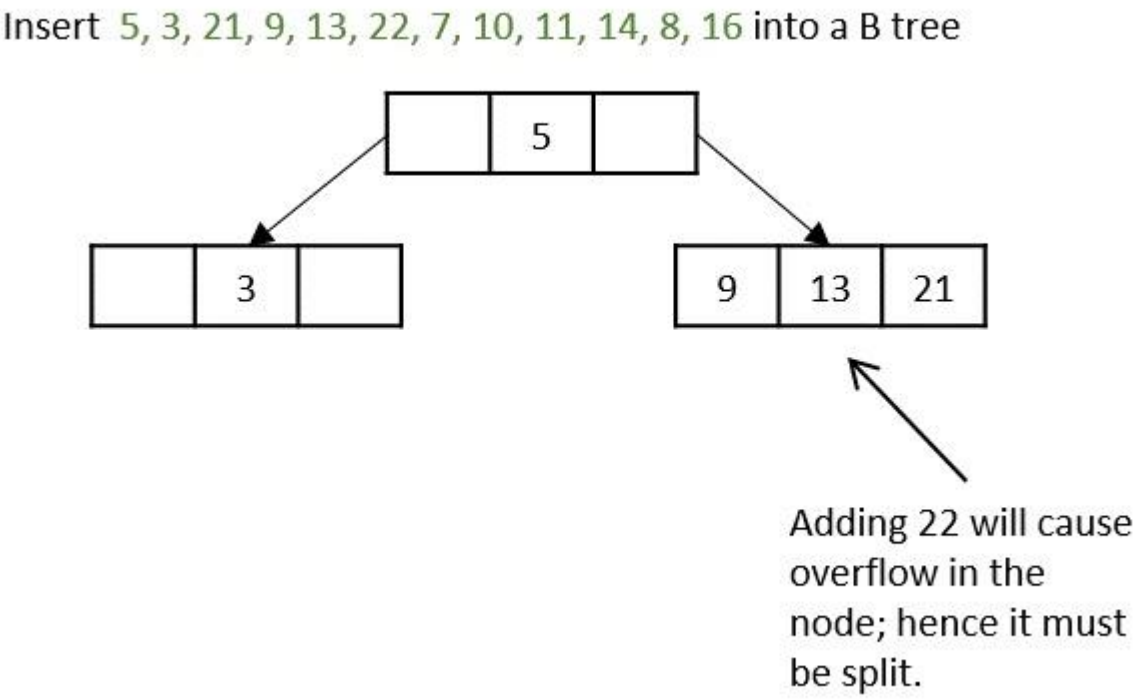
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys (m – 1) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil$) – 1 = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

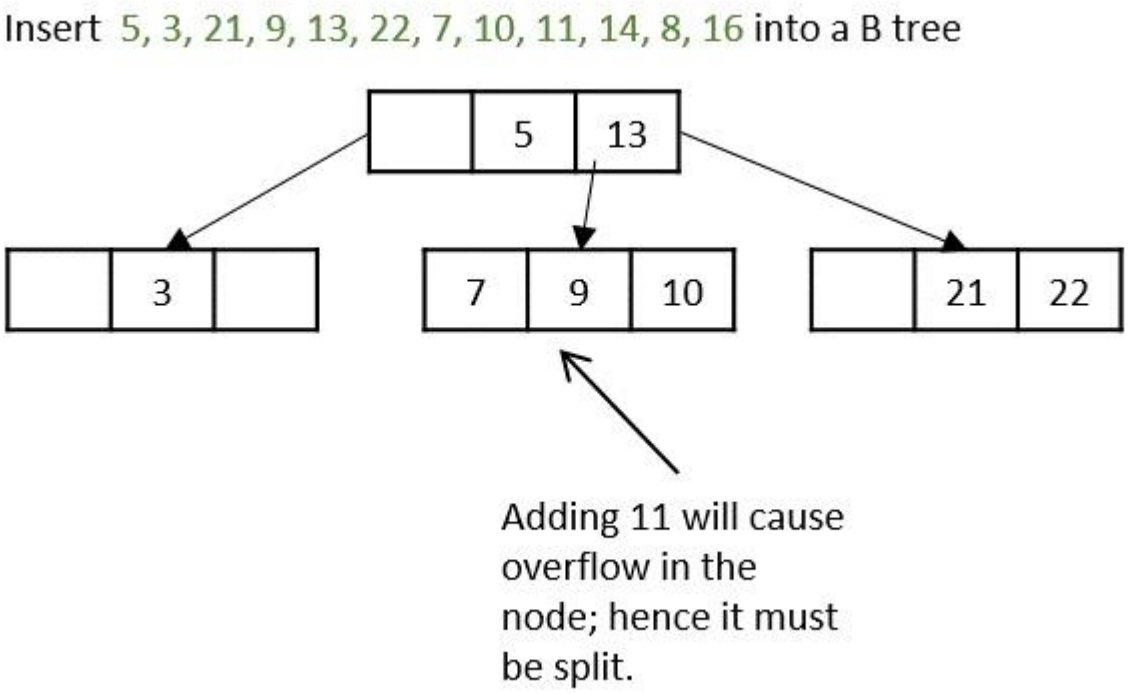
Step 2 − The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children.



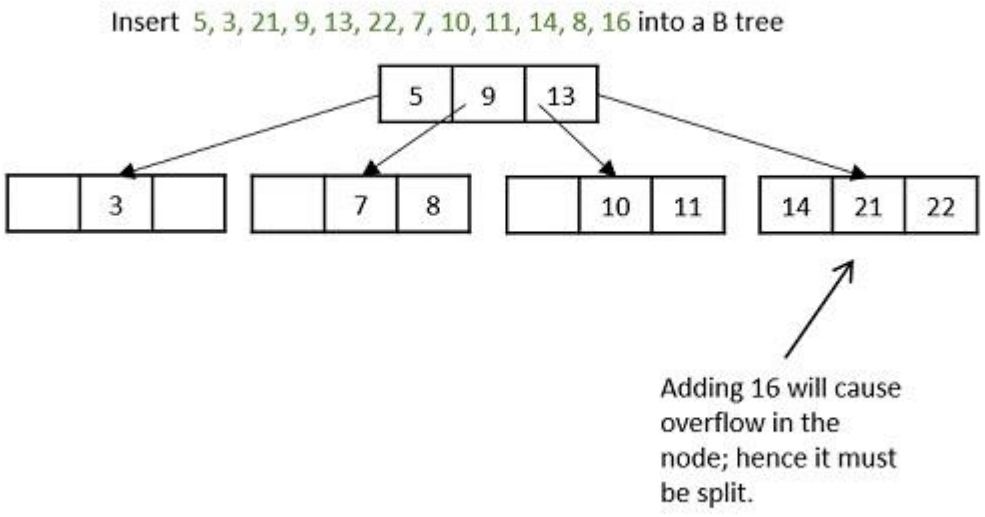
Step 3 − All the leaf nodes must be on the same level.



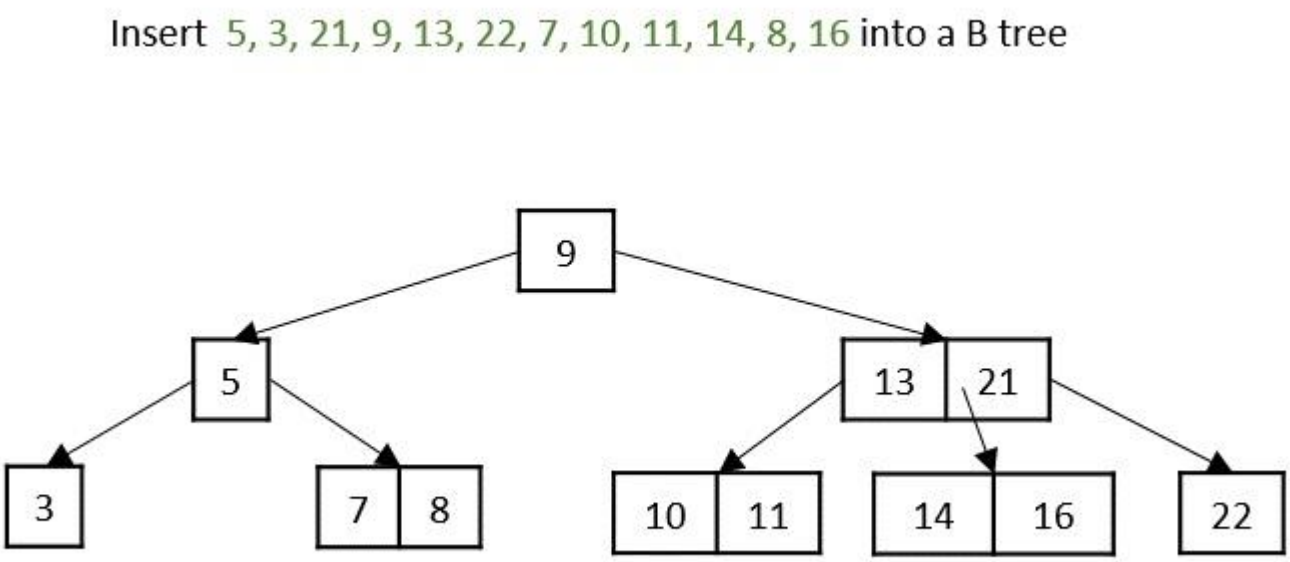
The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.



Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.



While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.



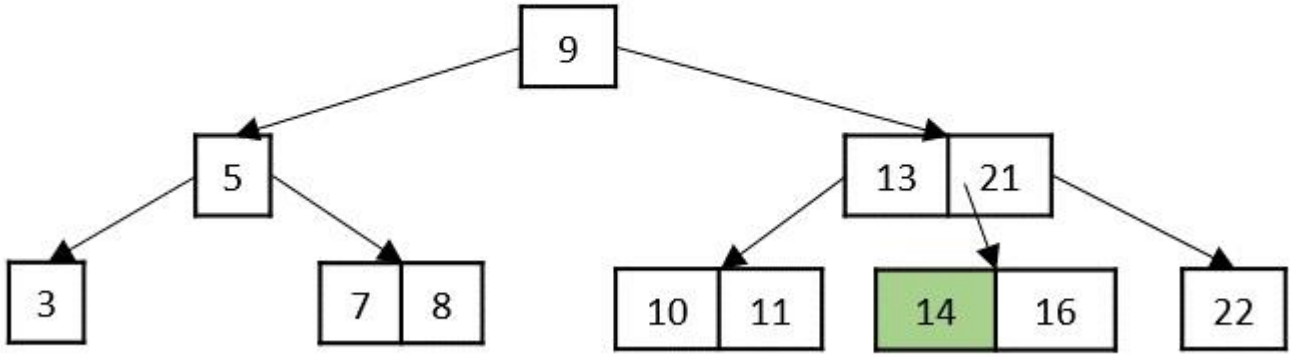
The final B tree after inserting all the elements is achieved.

Deletion

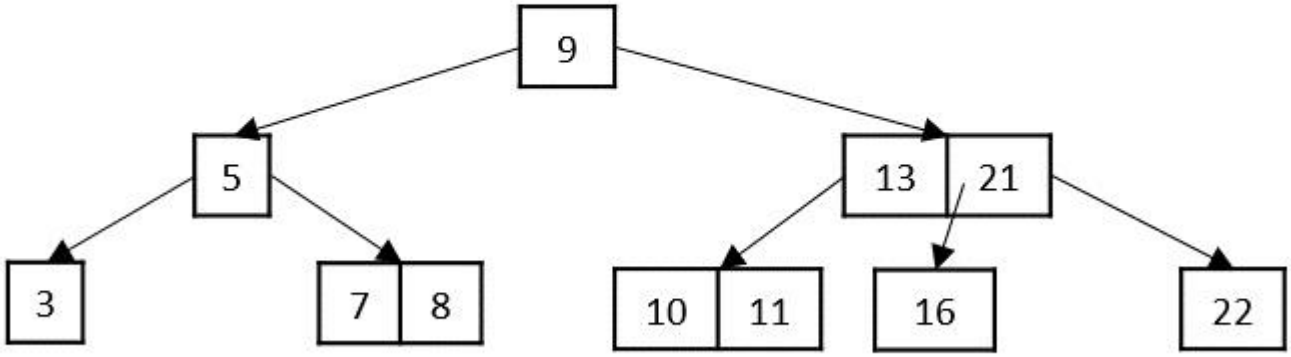
The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows −

Case 1 − If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.

Delete key 14

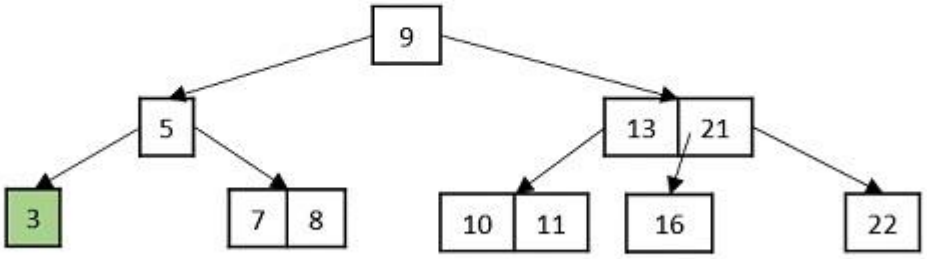


Delete key 14

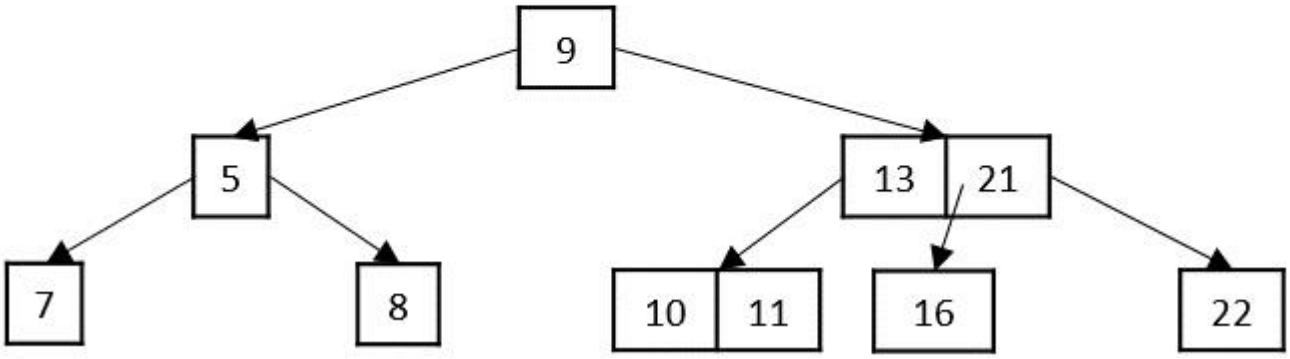


Case 2 − If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its **left sibling** or **right sibling**. In case if both siblings have exact minimum number of keys, **merge** the node in either of them.

Delete key 3

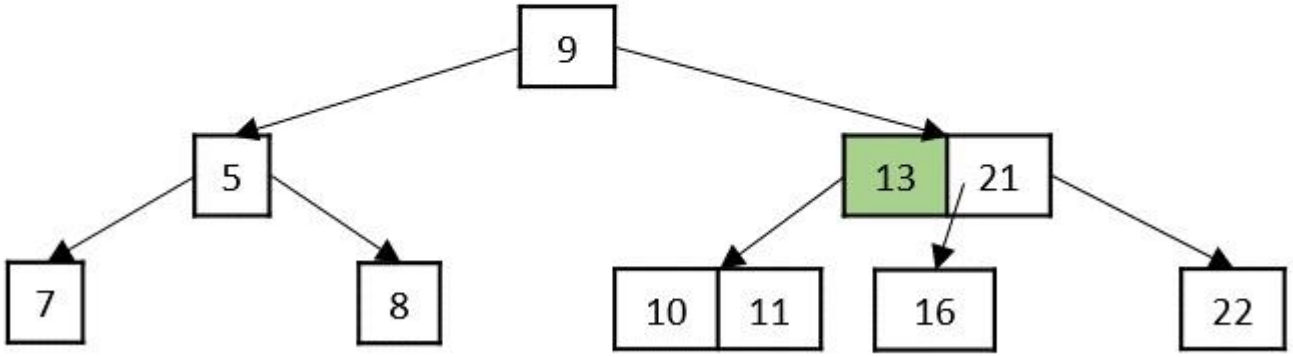


Delete key 3

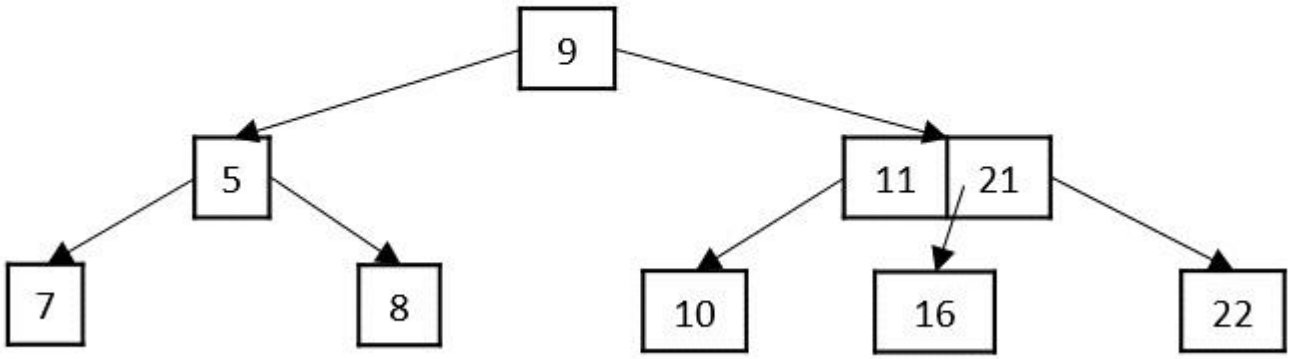


Case 3 – If the key to be deleted is in **an internal node**, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, they're **merged** together.

Delete key 13

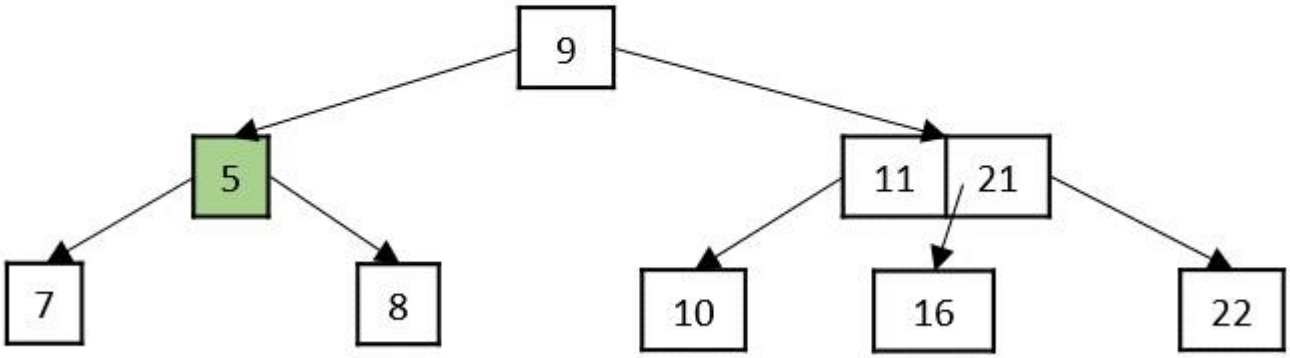


Delete key 13



Case 4 – If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its **sibling with its parent**.

Delete key 5



Delete key 5