B - Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree.

B-Tree can be defined as follows...

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

Here, number of keys in a node and number of children for a node is depending on the order of the B-Tree. Every B-Tree has order.

B-Tree of Order m has the following properties...

Property #1 - All the 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.

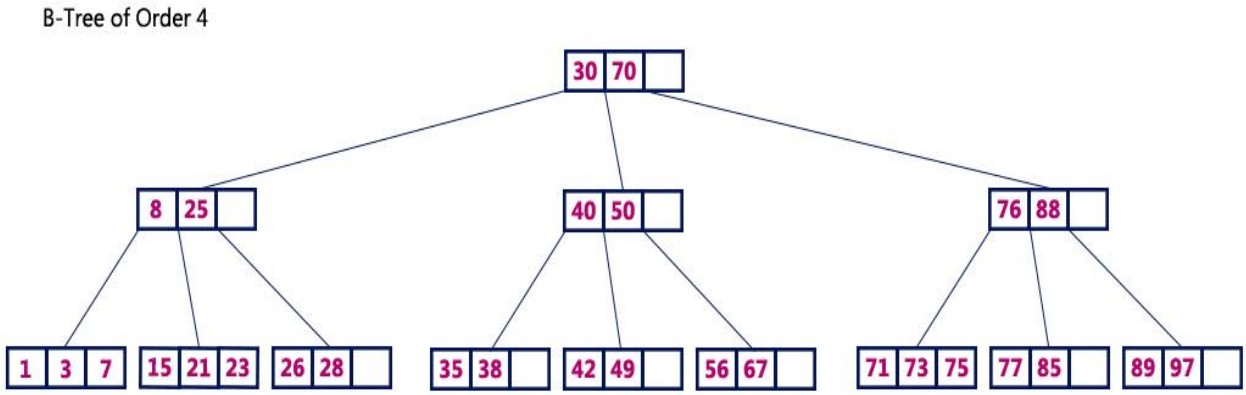
Property #4 - If the root node is a non leaf node, then it must have at least 2 children.

Property #5 - A non leaf node with n-1 keys must have n number of children.

Property #6 - All the key values within a node must be in Ascending Order.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

Example



Operations on a B-Tree

The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

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 matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, 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 sub tree.

Step 6: If search element is larger, then compare with next key value in the same node and repeat step 3, 4, 5 and 6 until we found exact match or comparisons completed with last key value in a leaf node.

Step 7: If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new key Value is attached to 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 into the tree as a root node.

Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.

Step 4: If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

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

Step 6: If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

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

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

insert(1)

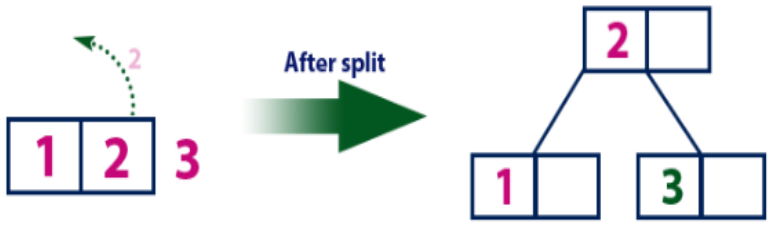
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

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.

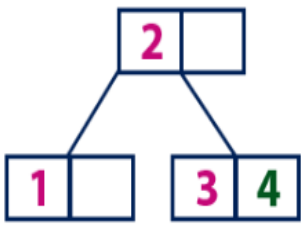
insert(3)

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.

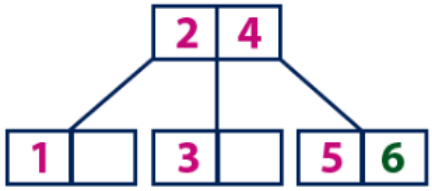


insert(5)

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.

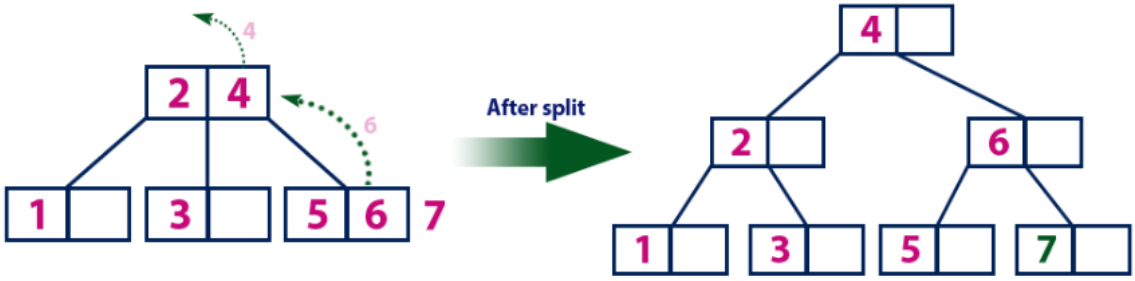
insert(6)

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.



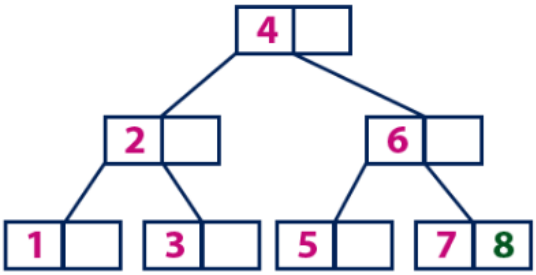
insert(7)

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.



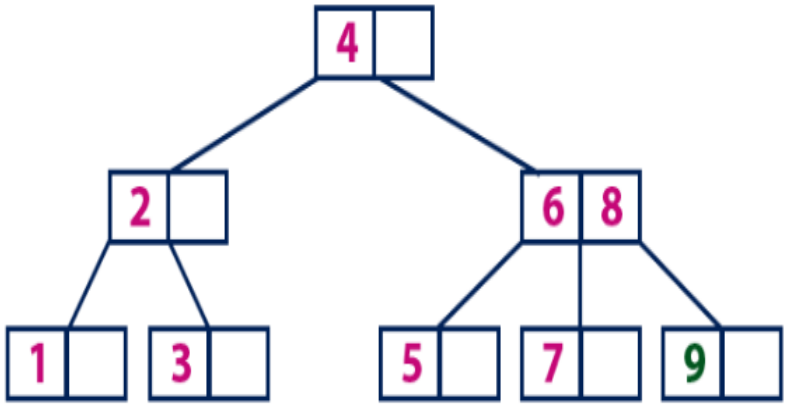
insert(8)

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.



insert(9)

Element '9' 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'. '9' 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 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' 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 values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.