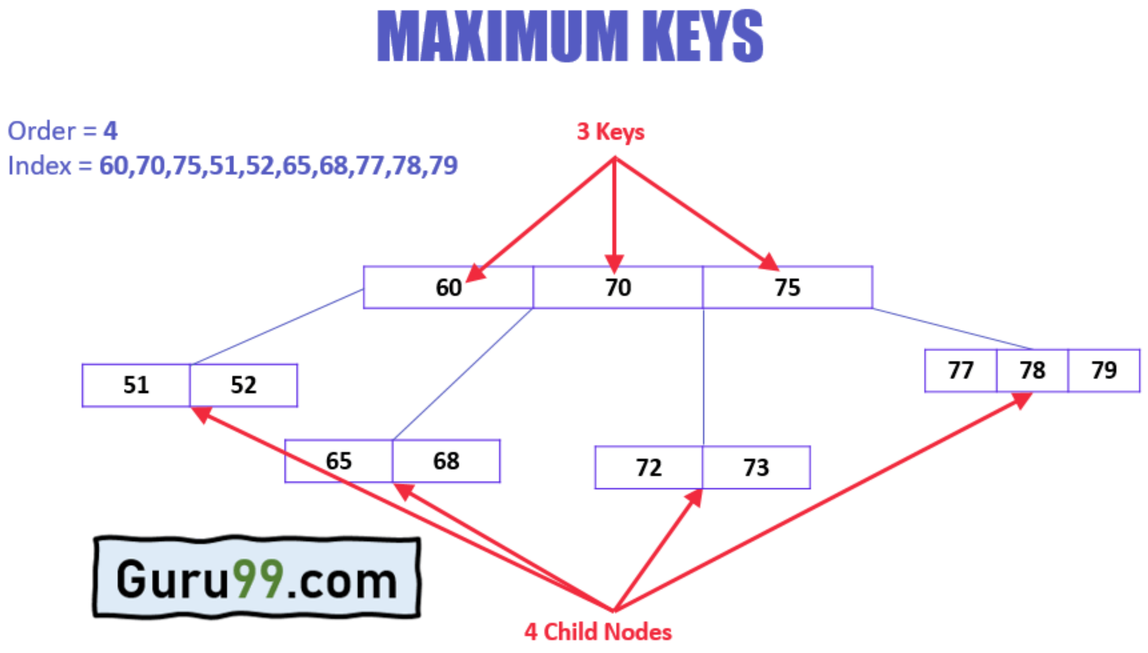
B-tree

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.

Rules for 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
- 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 = 4
max keys: 4 – 1 = 3
```

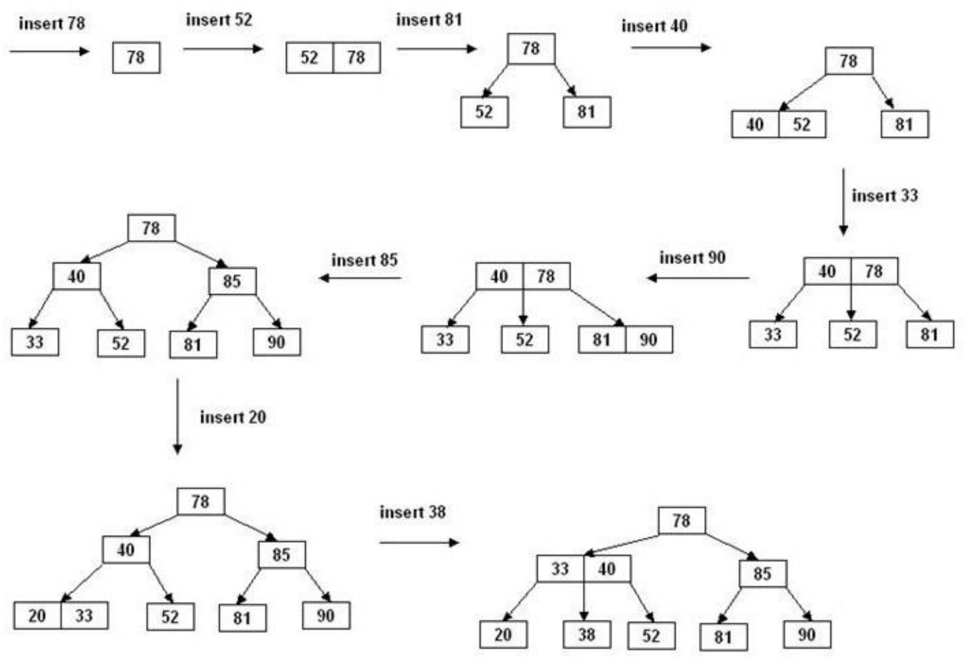


Why use B-Tree

Here, are reasons of using B-Tree

- Reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size (that is the number of child nodes) according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.
- A good choice to opt when it comes to reading and writing large blocks of data

- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



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

- **Property #1** - All **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 **atleast 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

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

- **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 matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, 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 subtree.
- **Step 6 -** If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7 -** If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
- **Step 3 -** If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- **Step 4 -** If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
- **Step 5 -** If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
- **Step 6 -** If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Deletion from a B-tree

The terms to be understood before studying deletion operation are:

1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
   - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
   - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.