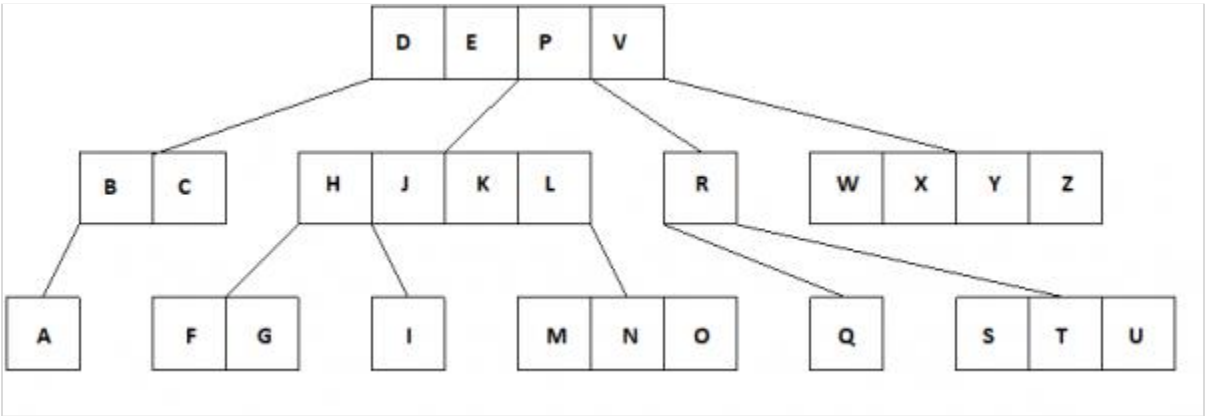
A multiway tree is defined as a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m-way tree).

As with the other trees that have been studied, the nodes in an m-way tree will be made up of m-1 key fields and pointers to children.

multiway tree of order 5



To make the processing of m-way trees easier some type of constraint or order will be imposed on the keys within each node, resulting in a multiway search tree of order m (or an m-way search tree). By definition an m-way search tree is a m-way tree in which following condition should be satisfied −

- Each node is associated with m children and m-1 key fields
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j-th key.
- The keys in the last m-j children are higher than the j-th key

m-WAY Search Trees

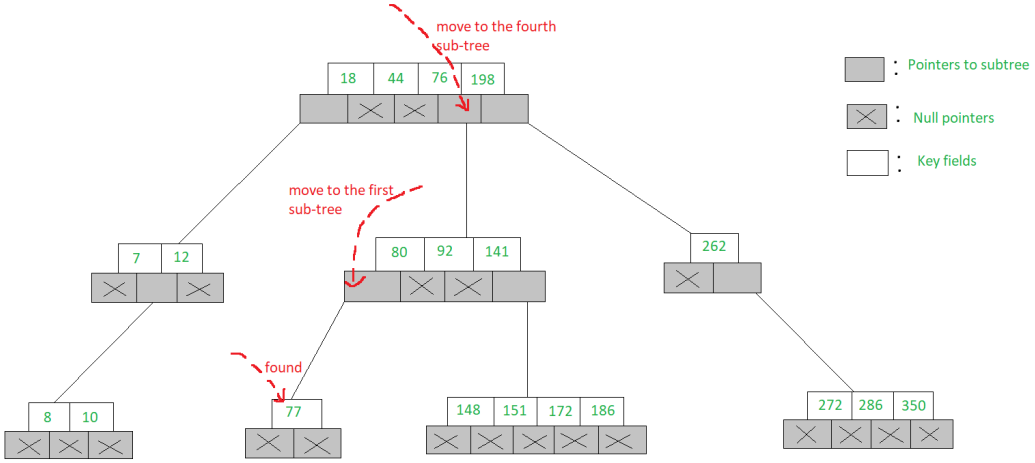
An example of a 5-Way search tree is shown in the figure below. Observe how each node has at most 5 child nodes & therefore has at most 4 keys contained in it.
The structure of a node of an m-Way tree is given below:

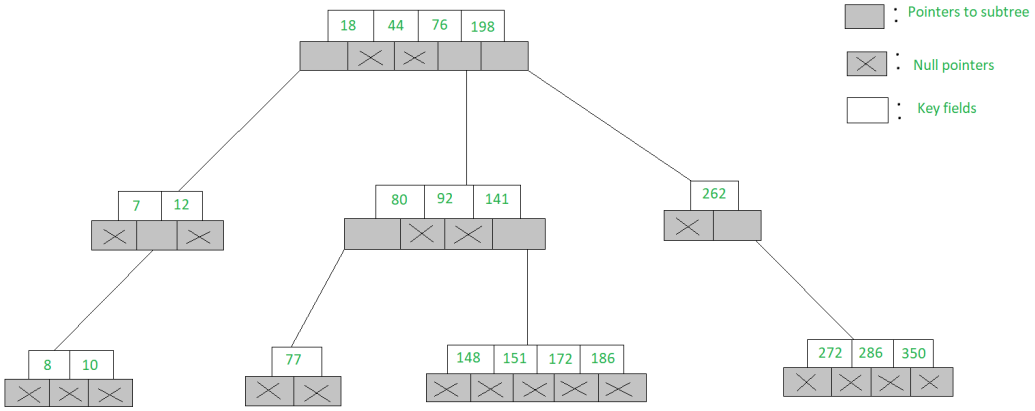
```
struct node {
    int count;
    int value[MAX + 1];
    struct node* child[MAX +
1];
};
```

- Here, **count** represents the number of children that a particular node has
- The values of a node stored in the array **value**
- The addresses of child nodes are stored in the **child** array
- The **MAX** macro signifies the maximum number of values that a particular node can contain

Searching in an m-Way search tree:

- Searching for a key in an m-Way search tree is similar to that of binary search tree
- To search for 77 in the 5-Way search tree, shown in the figure, we begin at the root & as 77> 76> 44> 18, move to the fourth sub-tree
- In the root node of the fourth sub-tree, 77< 80 & therefore we move to the first sub-tree of the node. Since 77 is available in the only node of this sub-tree, we claim 77 was successfully searched





B Tree

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

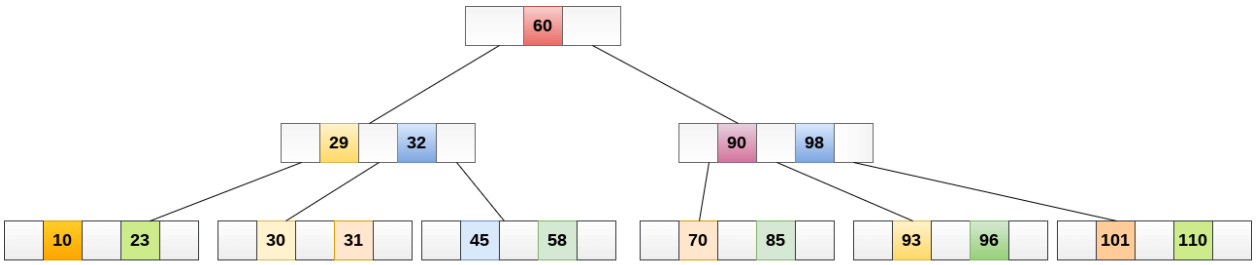
A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.

2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations

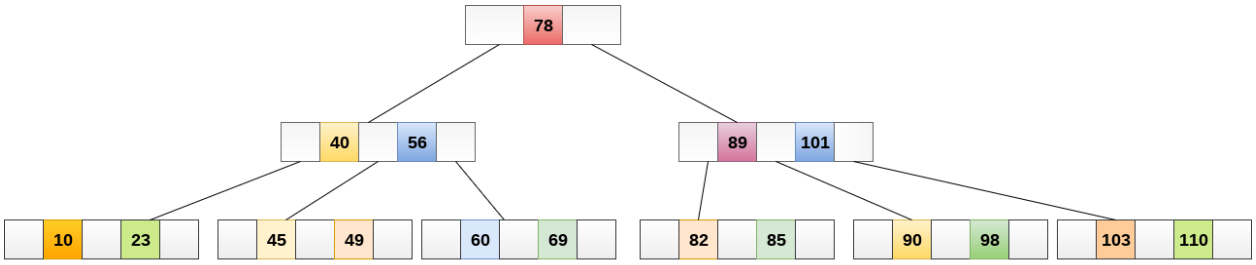
Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.

2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



Inserting

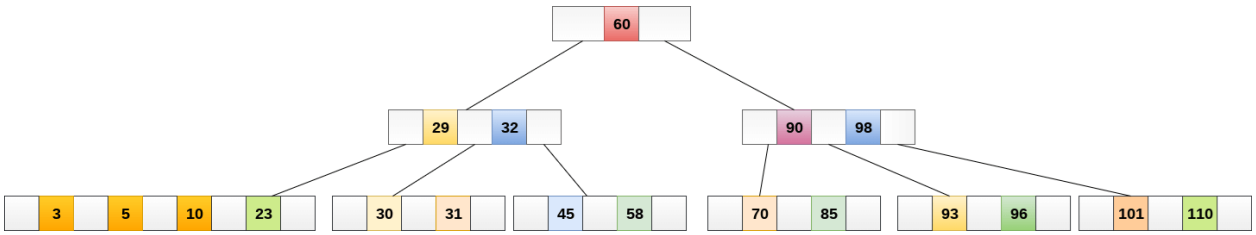
Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.

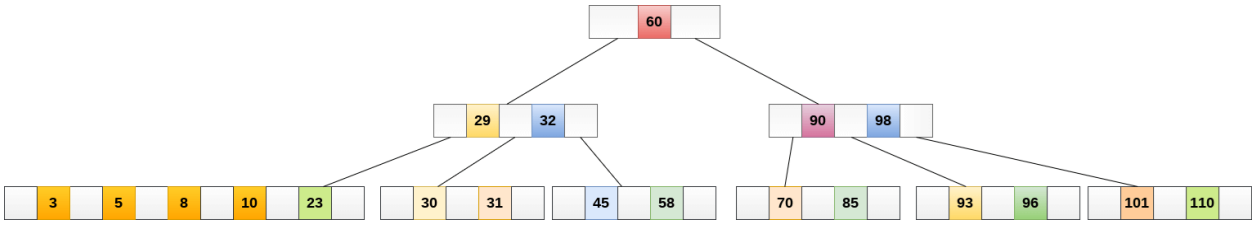
- If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

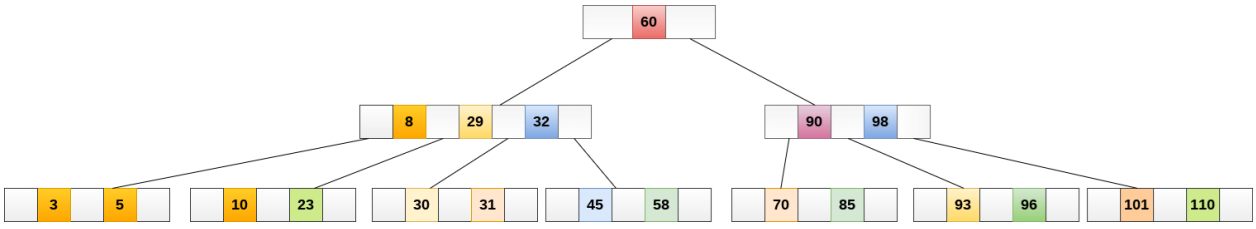
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

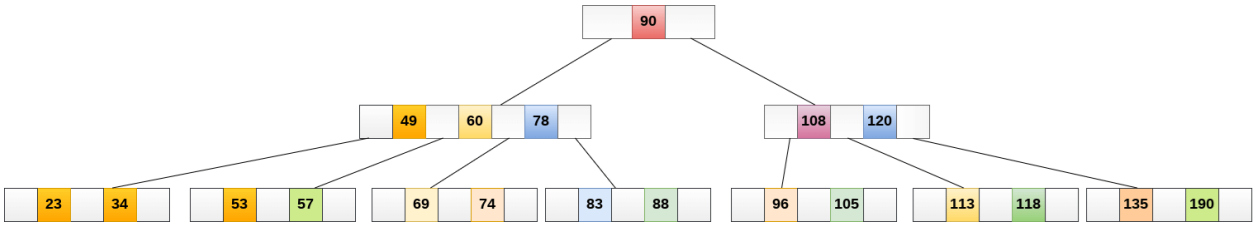
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

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.

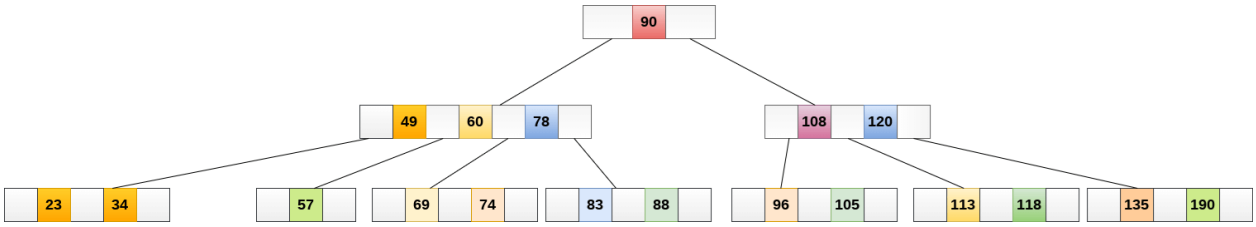
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

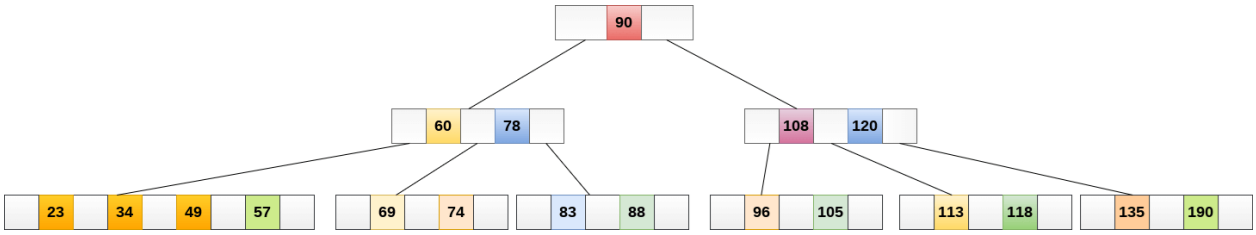


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Notes:

1. The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is: $h_{min} = \lceil \log_m(n+1) \rceil - 1$
2. The maximum height of the B-Tree that can exist with n number of nodes and d is the minimum number of children that a non-root node can have is: $h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$ and $t = \lceil \frac{m}{2} \rceil$ OBJ and OBJOBJ and OBJ

2-3 Trees | (Search and Insert)

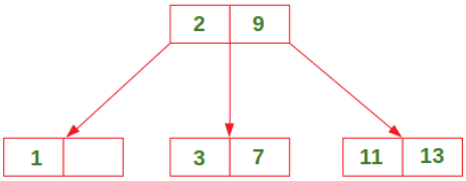
2-3 tree is a tree data structure in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. If a node contains one data element **leftVal**, it has two subtrees (children) namely **left** and **middle**. Whereas if a node contains two data elements **leftVal** and **rightVal**, it has three subtrees namely **left**, **middle** and **right**.

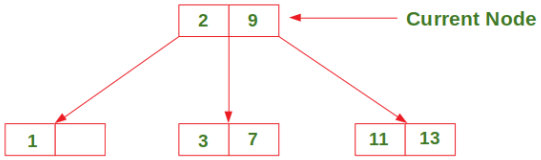
The main advantage with 2-3 trees is that it is balanced in nature as opposed to a binary search tree whose height in the worst case can be O(n). Due to this, the worst case time-complexity of operations such as search, insertion and deletion is $O(log(n))$ as the height of a 2-3 tree is $O(log(n))$.

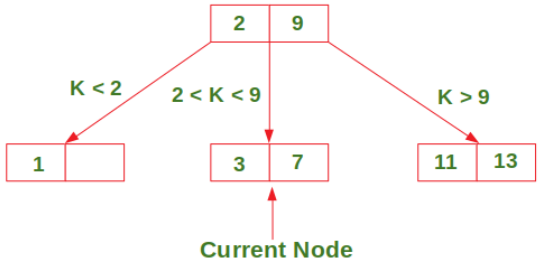
Search operation:

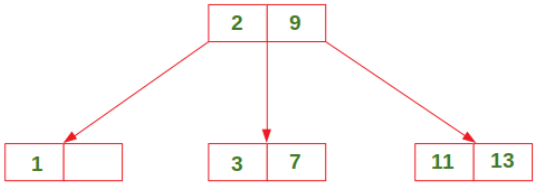
Consider the following example:

Search 5 in the following 2-3 Tree:







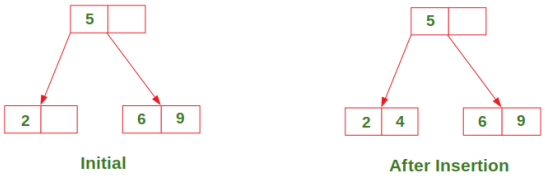


5 Not Found. Return False

Insertion: There are 3 possible cases in insertion which have been discussed below:

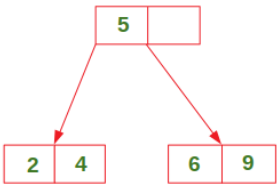
Case 1: Insert in a node with only one data element

Insert 4 in the following 2-3 Tree:

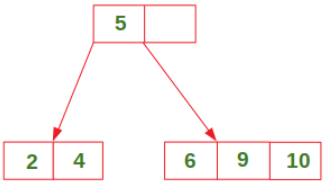


Case 2: Insert in a node with two data elements whose parent contains only one data element.

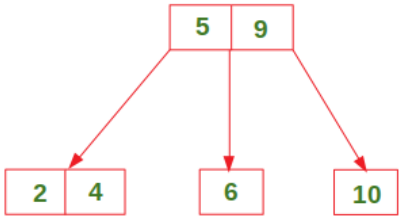
Insert 10 in the following 2-3 Tree:



Initial



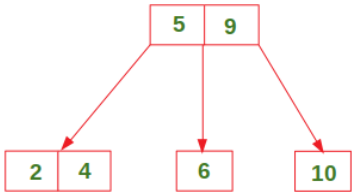
Temporary Node with 3 data elements



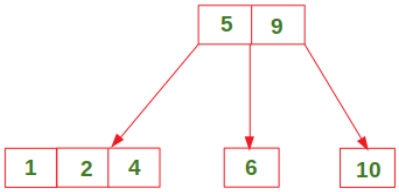
Move the middle element to
parent and split the current Node

Case 3: Insert in a node with two data elements whose parent also contains two data elements.

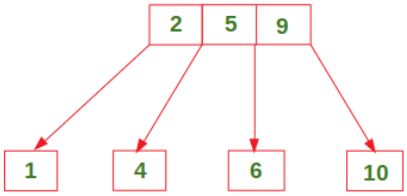
Insert 1 in the following 2-3 Tree:



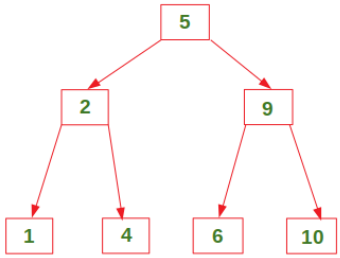
Initial



Temporary Node with 3 data elements



Move the middle element to the parent and split the current Node



Move the middle element to the parent and split the current Node