

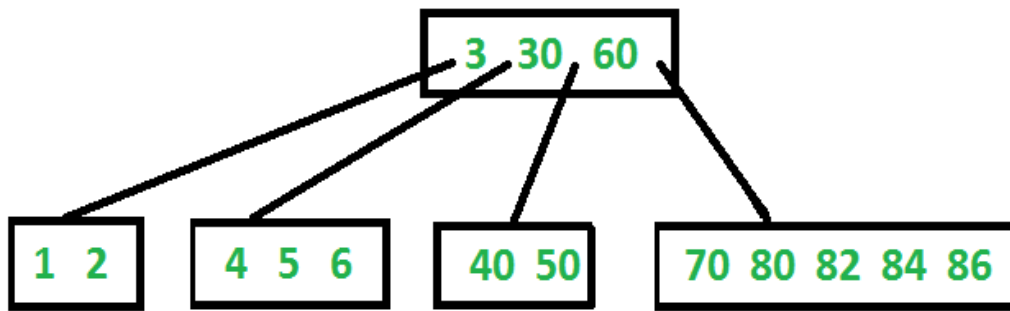
# B-Tree | Set 1 (Introduction)

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like **AVL** and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc ) require  $O(h)$  disk accesses where  $h$  is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since  $h$  is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

## Properties of B-Tree

- 1) All leaves are at same level.
- 2) A B-Tree is defined by the term *minimum degree* 't'. The value of  $t$  depends upon disk block size.
- 3) Every node except root must contain at least  $t-1$  keys. Root may contain minimum 1 key.
- 4) All nodes (including root) may contain at most  $2t - 1$  keys.
- 5) Number of children of a node is equal to the number of keys in it plus 1.
- 6) All keys of a node are sorted in increasing order. The child between two keys  $k_1$  and  $k_2$  contains all keys in range from  $k_1$  and  $k_2$ .
- 7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- 8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is  $O(\log n)$ .

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



## Search

Search is similar to search in Binary Search Tree. Let the key to be searched be  $k$ . We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find  $k$  in the leaf node, we return NULL.

## Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

```
// C++ implementation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys; // An array of keys
    int t;     // Minimum degree (defines the range for
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with
    void traverse();

    // A function to search a key in subtree rooted with
```

```

BTreeNode *search(int k); // returns NULL if k is not present

// Make BTree friend of this so that we can access private
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, traverse through

```

```

// and first n children
int i;
for (i = 0; i < n; i++)
{
    // If this is not leaf, then before printing key
    // traverse the subtree rooted with child C[i].
    if (leaf == false)
        C[i]->traverse();
    cout << " " << keys[i];
}

// Print the subtree rooted with last child
if (leaf == false)
    C[i]->traverse();
}

// Function to search key k in subtree rooted with this
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

```

The above code doesn't contain driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in [Cormen book](#)), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept

same as Cormen book for better readability.

## Insertion and Deletion

[B-Tree Insertion](#)

[B-Tree Deletion](#)

## References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)