<https://www.simplilearn.com/tutorials/data-structure-tutorial/b-tree-in-data-structure>

So far, we have assumed that we can store an entire data structure in the main memory of a computer. Suppose, however, that we have more data than can fit in main memory, and, as a result, must have the data structure reside on disk.

Modern computers execute billions of instructions per second. That is pretty fast, mainly because the speed depends largely on electrical properties. On the other hand, a disk is mechanical. Its speed depends largely on the time it takes to spin the disk and to move a disk head. Many disks spin at 7,200 RPM. Thus, in 1 min it makes 7,200 revolutions; hence, one revolution occurs in 1/120 of a second, or 8.3 ms. On average, we might expect that we have to spin a disk halfway to find what we are looking for, but this is compensated by the time to move the disk head, so we get an access time of 8.3 ms. (This is a very charitable estimate; 9–11 ms access times are more common.) Consequently, we can do approximately 120 disk accesses per second. This sounds pretty good, until we compare it with the processor speed. What we have is billions instructions equal to 120 disk accesses. Of course, everything here is a rough calculation, but the relative speeds are pretty clear: Disk accesses are incredibly expensive.

Furthermore, processor speeds are increasing at a much faster rate than disk speeds (it is disk *sizes* that are increasing quite quickly). So we are willing to do lots of calculations just to save a disk access. In almost all cases, it is the number of disk accesses that will dominate the running time. Thus, if we halve the number of disk accesses, the running time will halve. Here is how the typical search tree performs on disk: Suppose we want to access the driving records for citizens in the state of Florida. We assume that we have 10,000,000

items, that each key is 32 bytes (representing a name), and that a record is 256 bytes. We assume this does not fit in main memory and that we are 1 of 20 users on a system (so we have 1/20 of the resources). Thus, in 1 sec we can execute many millions of instructions or perform six disk accesses. The unbalanced binary search tree is a disaster. In the worst case, it has linear depth and thus could require 10,000,000 disk accesses. On average, a successful search would require 1.38 logN disk accesses, and since log 10000000 ≈ 24, an average search would require 32 disk accesses, or 5 sec. In a typical randomly constructed tree, we would expect that a few nodes are three times deeper; these would require about 100 disk accesses, or 16 sec. An AVL tree is somewhat better. The worst case of 1.44 logN is unlikely to occur,and the typical case is very close to logN. Thus an AVL tree would use about 25 disk accesses on average, requiring 4 sec.

We want to reduce the number of disk accesses to a very small constant, such as three or four. We are willing to write complicated code to do this, because machine instructions are essentially free, as long as we are not ridiculously unreasonable. It should probably be clear that a binary search tree will not work, since the typical AVL tree is close to optimal height. We cannot go below logN using a binary search tree. The solution is intuitively simple: If we have more branching, we have less height. Thus, while a perfect binary

tree of 31 nodes has five levels, a 5-ary tree of 31 nodes has only three levels, as shown in Figure 4.62. An M-ary search tree allows M-way branching. As branching increases, the depth decreases. Whereas a complete binary tree has height that is roughly log2 N, a complete M-ary tree has height that is roughly logM N. We can create an M-ary search tree in much the same way as a binary search tree. In a binary search tree, we need one key to decide which of two branches to take. In an M-ary search tree, we need M − 1 keys to decide which branch to take. To make this scheme efficient in the worst case, we need to ensure that the M-ary search tree is balanced in some way. Otherwise, like a binary search tree, it could degenerate into a linked list. Actually, we want an even more restrictive balancing condition. That is, we do not want an M-ary

search tree to degenerate to even a binary search tree, because then we would be stuck with logN accesses.

# B Tree

A B-tree is a sort of self-balancing search tree whereby each node could have more than two children and hold multiple keys. It’s a broader version of the binary search tree. It is also usually called a height-balanced m-way tree.

## What is B Tree in Data Structure?

The B Tree is a particular m-way tree which can be used to access discs in a variety of ways. At most m children and m-1 keys can be found in a B-Tree of order m. One of the main advantages of the B tree is its capacity to store a large number of keys inside a single node and huge key values while keeping the tree’s height low.

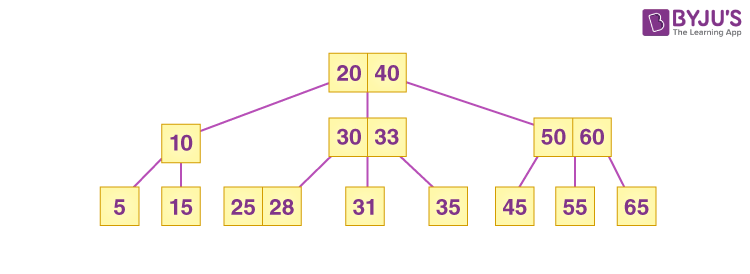
## Need of a B Tree in a Data Structure

The need for B-tree emerged as the demand for faster access to physical storage media such as a hard disk grew. With a bigger capacity, backup storage devices were slower. There was a demand for data structures that reduced the number of disc accesses.

A binary search tree, an AVL tree, a red-black tree, and other data structures can only store one key in each node. When you need to store a significant number of keys, the height of such trees grows quite vast, and the time it takes to access them grows.

B-tree, on the other hand, can store multiple keys inside a single node and has several child nodes. It considerably reduces the height, allowing for speedier disk access.

#### B Tree Example



## Properties of B Tree in DBMS

All of the features of a M way tree are present in a B tree of order m. Additionally, it has the following features:

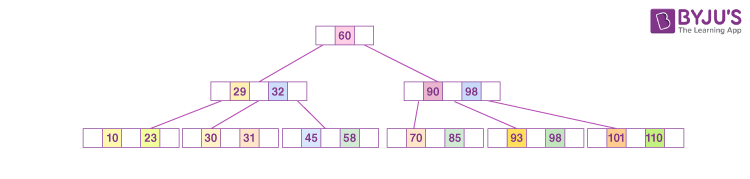
* In a B-Tree, each node has a maximum of m children.
* Except for the root and leaf nodes, each node in a B-Tree has at least m/2 children.
* There must be at least two root nodes.
* The level of all the leaf nodes should be the same.

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

**-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**.

The following graphic depicts a B tree of order 4:



Any attribute of B Tree may be violated while performing some operations on it, such as the number of minimum children each node can have. The tree may split or unite in order to keep the features of the B Tree.

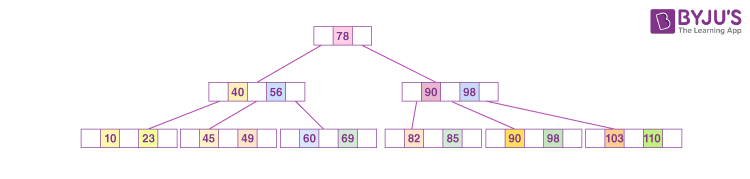
## Operations

### Search Operation

B Trees are comparable to Binary Search Trees in terms of searching. For example, let’s look for item 49 in the B Tree below. The procedure will go somewhat like this:

* Compare item 49 to node 78, which is the root node. Move to the left sub-tree since 49 < 78.
* Since 40 < 49 < 56, move to the right subtree of 40. If 49>45, move to the right, and then compare and contrast 49.
* Return if a match is found.

The height of the tree affects the search in a B tree. To search an element in a B tree, the search technique requires O(log n) time.



### Insertion Operation

At the leaf node level, insertions are made. In order to place an item into B Tree, the following algorithm must be followed:

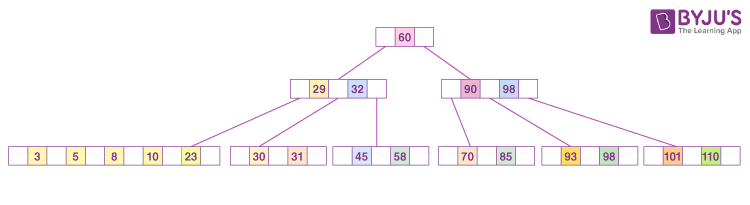
* Navigate the B Tree until you find the suitable leaf node to insert the node at.
* If there are fewer than m-1 keys in the leaf node, insert the elements in ascending order.

Otherwise, if the leaf node comprises m-1 keys, proceed to the next step.

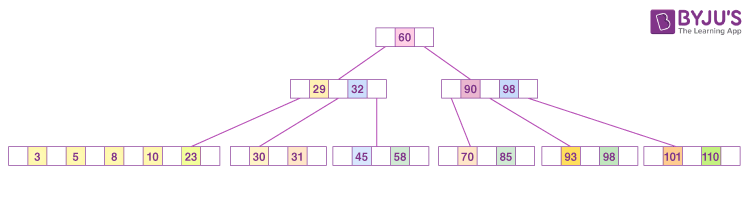
* In the rising sequence of elements, add the new element.
* In the middle, divide the node into two nodes.
* The median element should be pushed up to its parent node.
* If the parent node has the same m-1 number of keys as the child node, split it as well using the very same steps.

#### Example

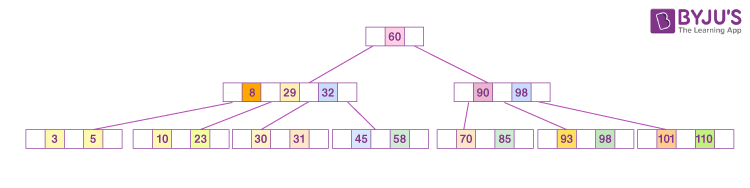
In the B Tree of order 5, insert node 8 as indicated in the diagram.



Because 8 will be put to the right of 5, enter 8.



There are now 5 keys in the node, which is more than (5 -1 = 4) keys. As a result, divide the node from the median, which is number 8, and push it up to its parent node, as illustrated below.



### Deletion Operation

At the leaf nodes, deletion is also conducted. It can be a leaf node or an inside node that needs to be destroyed. In order to delete a node from a B tree, use the following algorithm:

1. Determine the location of the leaf node.

2. If the leaf node has more than m/2 keys, delete the required key from the node.

3. If the leaf node lacks m/2 keys, fill in the gaps with the element from the eighth or left sibling.

* If the left sibling has more than m/2 elements, shift the intervening element downwards to the node wherever the key is deleted and push the largest element back to its parent.
* If the right sibling has more than m/2 items, shift the intervening element downwards to the node wherever the key is deleted, then push the smallest element up to the parent.

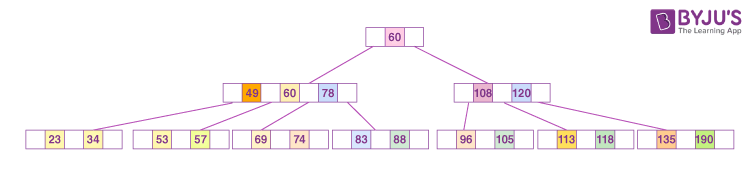
4. Create a new leaf node by merging two leaf nodes and the parent node’s intervening element if neither sibling has more than m/2 elements.

5. If the parent has less than m/2 nodes, repeat the operation on the parent as well.

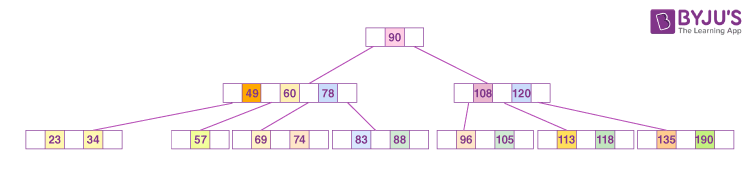
If the node to be destroyed is an internal node, its in-order successor or predecessor should be used instead. The process will be the same as the node is deleted from the leaf node because the successor/predecessor would always be on the leaf node.

#### Example 1

Remove node 53 from B Tree of the order 5, as indicated in the diagram below:

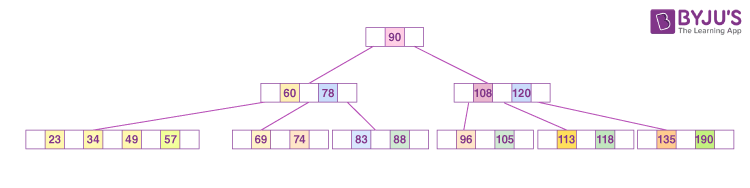


Element 49’s right child contains the number 53. Remove it.



Now that 57 is the only element left in the node, the minimal number of elements required in a B tree of rank 5 is 2. Because the elements in its left and right subtrees are similarly insufficient, combine it with the left sibling of the intervening element of the parent, i.e. 49.

The following is the final B tree:



## B Tree Application

Because accessing values held in a huge database, saved on a disc, is a very time-consuming activity, the B tree is used to index the data and enable quick access to the actual information stored on the discs.

In the worst situation, searching an unindexed and unsorted database with n key values takes O(n) time. Also, if we use B Tree for indexing this DB, it will be searched in about O(log n) time.

# 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

The search operation in B-Tree is similar to the search operation in Binary Search Tree. In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but here we make an n-way decision every time. Where 'n' is the total number of children the node has. In a B-Tree, 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 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.

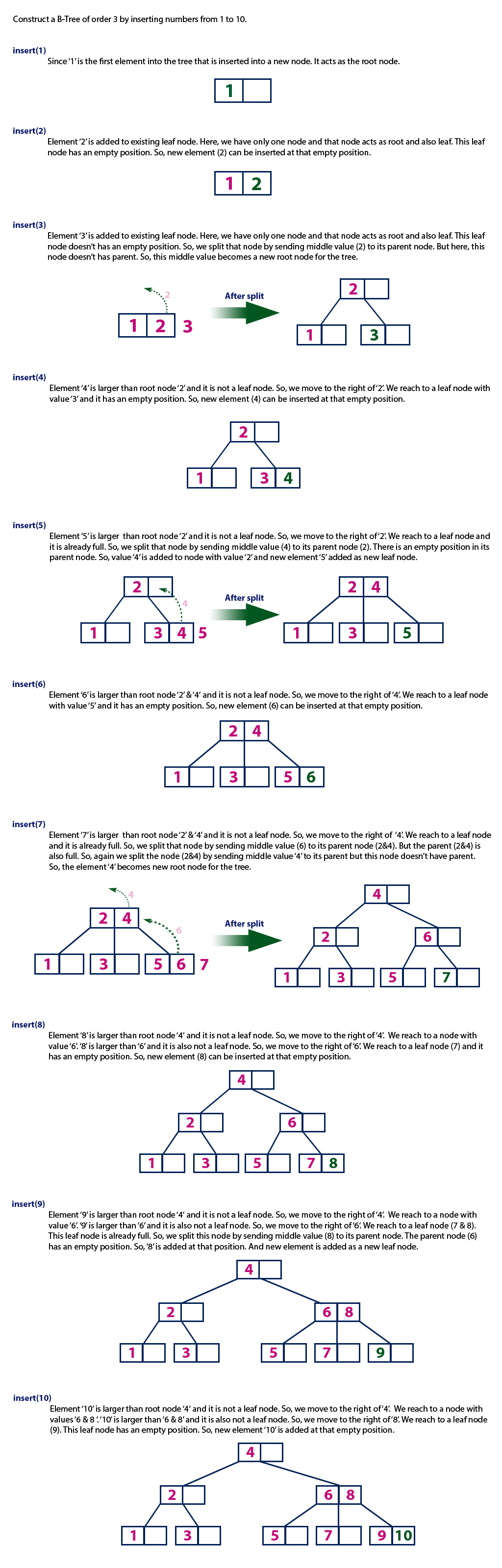
# Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the 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 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.

##### Example

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



// C++ program for B-Tree insertion

#include<iostream>

using namespace std;

// A BTree node

class BTreeNode

{

    int \*keys;  // An array of keys

    int t;      // Minimum degree (defines the range for number of keys)

    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 utility function to insert a new key in the subtree rooted with

    // this node. The assumption is, the node must be non-full when this

    // function is called

    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of y in

    // child array C[].  The Child y must be full when this function is called

    void splitChild(int i, BTreeNode \*y);

    // A function to traverse all nodes in a subtree rooted with this node

    void traverse();

    // A function to search a key in the subtree rooted with this node.

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

// Make BTree friend of this so that we can access private members of this

// 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); }

    // The main function that inserts a new key in this B-Tree

    void insert(int k);

};

// Constructor for BTreeNode class

BTreeNode::BTreeNode(int t1, bool leaf1)

{

    // Copy the given minimum degree and leaf property

    t = t1;

    leaf = leaf1;

    // 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 this node

void BTreeNode::traverse()

{

    // There are n keys and n+1 children, traverse through n keys

    // and first n children

    int i;

    for (i = 0; i < n; i++)

    {

        // If this is not leaf, then before printing key[i],

        // 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 node

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 main function that inserts a new key in this B-Tree

void BTree::insert(int k)

{

    // If tree is empty

    if (root == NULL)

    {

        // Allocate memory for root

        root = new BTreeNode(t, true);

        root->keys[0] = k;  // Insert key

        root->n = 1;  // Update number of keys in root

    }

    else // If tree is not empty

    {

        // If root is full, then tree grows in height

        if (root->n == 2\*t-1)

        {

            // Allocate memory for new root

            BTreeNode \*s = new BTreeNode(t, false);

            // Make old root as child of new root

            s->C[0] = root;

            // Split the old root and move 1 key to the new root

            s->splitChild(0, root);

            // New root has two children now.  Decide which of the

            // two children is going to have new key

            int i = 0;

            if (s->keys[0] < k)

                i++;

            s->C[i]->insertNonFull(k);

            // Change root

            root = s;

        }

        else  // If root is not full, call insertNonFull for root

            root->insertNonFull(k);

    }

}

// A utility function to insert a new key in this node

// The assumption is, the node must be non-full when this

// function is called

void BTreeNode::insertNonFull(int k)

{

    // Initialize index as index of rightmost element

    int i = n-1;

    // If this is a leaf node

    if (leaf == true)

    {

        // The following loop does two things

        // a) Finds the location of new key to be inserted

        // b) Moves all greater keys to one place ahead

        while (i >= 0 && keys[i] > k)

        {

            keys[i+1] = keys[i];

            i--;

        }

        // Insert the new key at found location

        keys[i+1] = k;

        n = n+1;

    }

    else // If this node is not leaf

    {

        // Find the child which is going to have the new key

        while (i >= 0 && keys[i] > k)

            i--;

        // See if the found child is full

        if (C[i+1]->n == 2\*t-1)

        {

            // If the child is full, then split it

            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and

            // C[i] is splitted into two.  See which of the two

            // is going to have the new key

            if (keys[i+1] < k)

                i++;

        }

        C[i+1]->insertNonFull(k);

    }

}

// A utility function to split the child y of this node

// Note that y must be full when this function is called

void BTreeNode::splitChild(int i, BTreeNode \*y)

{

    // Create a new node which is going to store (t-1) keys

    // of y

    BTreeNode \*z = new BTreeNode(y->t, y->leaf);

    z->n = t - 1;

    // Copy the last (t-1) keys of y to z

    for (int j = 0; j < t-1; j++)

        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z

    if (y->leaf == false)

    {

        for (int j = 0; j < t; j++)

            z->C[j] = y->C[j+t];

    }

    // Reduce the number of keys in y

    y->n = t - 1;

    // Since this node is going to have a new child,

    // create space of new child

    for (int j = n; j >= i+1; j--)

        C[j+1] = C[j];

    // Link the new child to this node

    C[i+1] = z;

    // A key of y will move to this node. Find the location of

    // new key and move all greater keys one space ahead

    for (int j = n-1; j >= i; j--)

        keys[j+1] = keys[j];

    // Copy the middle key of y to this node

    keys[i] = y->keys[t-1];

    // Increment count of keys in this node

    n = n + 1;

}// Driver program to test above functions

int main()

{

    BTree t(3); // A B-Tree with minimum degree 3

    t.insert(10);

    t.insert(20);

    t.insert(5);

    t.insert(6);

    t.insert(12);

    t.insert(30);

    t.insert(7);

    t.insert(17);

    cout << "Traversal of the constructed tree is ";

    t.traverse();

    int k = 6;

    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    k = 15;

    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    return 0;}

## Practice Problems on B Tree

**1.**The B-trees of the order 4 and height 3 would have a maximum of \_\_\_\_\_\_\_\_ keys.

**a.**188

**b.**127

**c.**63

**d.**255

**Answer- (d)**255

**2.**If five node splitting operations occur when some entry is inserted in a B-tree, how many nodes would be written?

**a.**5

**b.**11

**c.**7

**d.**14

**Answer- (b)**11

**3.**AVL tree and B-tree consist of a similar worst case time complexity for deletion and insertion.

**a.**True

**b.**False

**Answer- (a)**True

**4.**2-3-4 trees refer to the B-trees of order 4. These are an isometric of the \_\_\_\_\_\_\_ trees.

**a.**AA

**b.**AVL

**c.**Red-Black

**d.**2-3

**Answer- (c)**Red-Black