B-Tree

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.
Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as "large key" trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.
B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

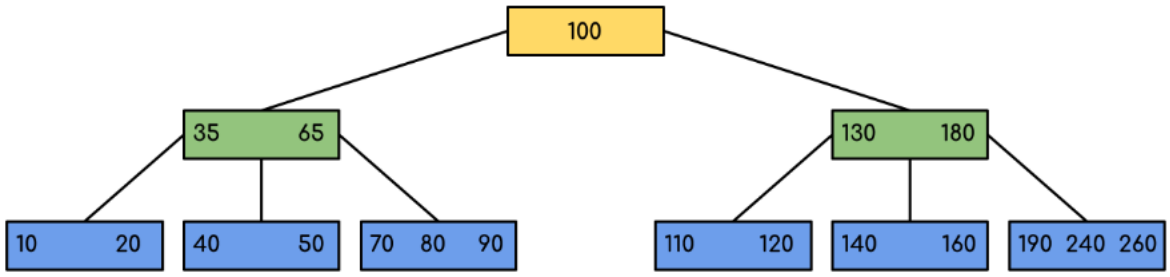
Note: "n" is the total number of elements in the B-tree

Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree '**t**'. The value of '**t**' depends upon disk block size.
- Every node except the root must contain at most t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t – 1**) keys.
- Number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.

- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of a B-Tree of minimum order 5
Note: that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leafs have no empty sub-tree and have keys one less than the number of their children.

Interesting Facts about B-Trees:

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

Traversal in B-Tree:

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 the remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree:

Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

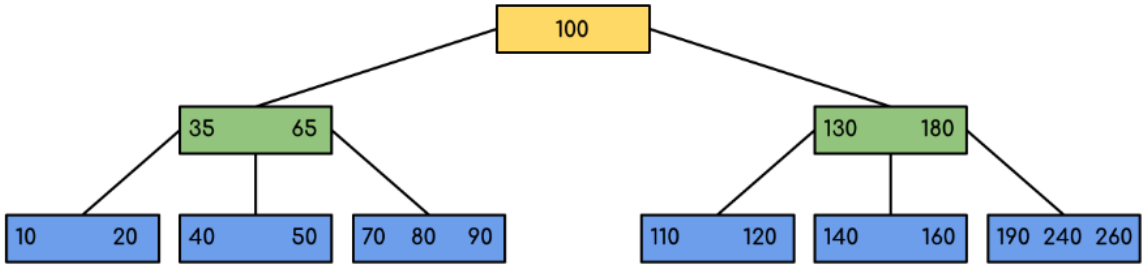
- Start from the root and recursively traverse down.
- For every visited non-leaf node,
    - If the node has the key, we simply return the node.
    - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

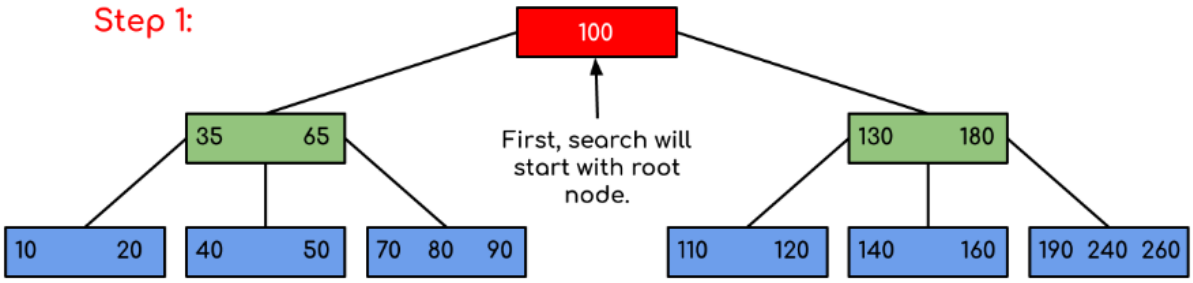
Algorithm for Searching an Element in a B-Tree:-

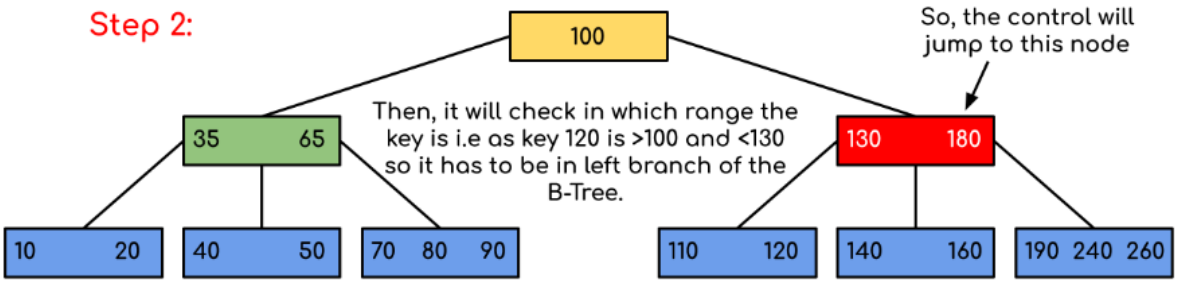
```
class Node {
    int n;
    int[] key = new int[MAX_KEYS];
    Node[] child = new Node[MAX_CHILDREN];
    boolean leaf;
}
Node BtreeSearch(Node x, int k) {
    int i = 0;
    while (i < x.n && k >= x.key[i]) {
        i++;
    }
    if (i < x.n && k == x.key[i]) {
        return x;
    }
    if (x.leaf) {
        return null;
    }
    return BtreeSearch(x.child[i], k);
}
```

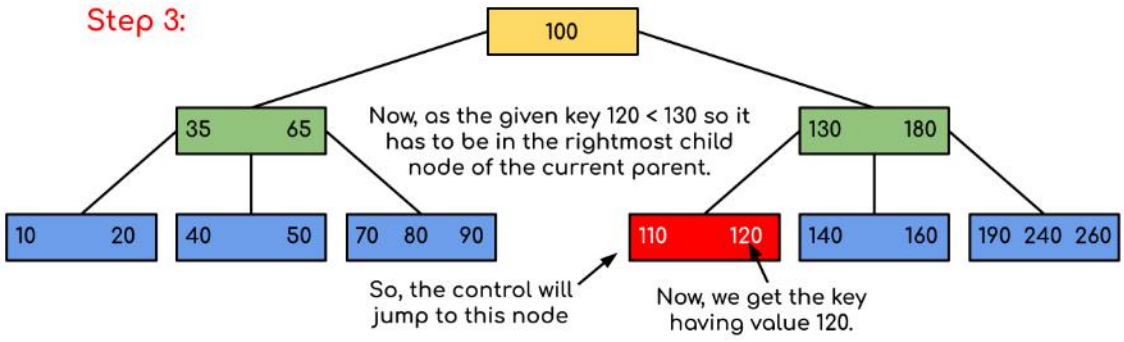
Examples:
Input: *Search 120 in the given B-Tree.*



Solution:







In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically, and therefore the control flow will go similarly as shown within the above example.

Below is the implementation of the above approach:

```
// Java program to illustrate the sum of two numbers

// A BTree

class Btree {

    public BTreeNode root; // Pointer to root node

    public int t; // Minimum degree

    // Constructor (Initializes tree as empty)

    Btree(int t)

    {

        this.root = null;

        this.t = t;

    }



    // function to traverse the tree

    public void traverse()

    {

        if (this.root != null)

            this.root.traverse();

        System.out.println();

    }

    // function to search a key in this tree

    public BTreeNode search(int k)
```

```
    {
        if (this.root == null)
            return null;
        else
            return this.root.search(k);
    }
}
// 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
    boolean
        leaf; // Is true when node is leaf. Otherwise false
    // Constructor
    BTreeNode(int t, boolean leaf)
    {
        this.t = t;
        this.leaf = leaf;
        this.keys = new int[2 * t - 1];
```

```
        this.C = new BTreeNode[2 * t];
        this.n = 0;
    }
    // A function to traverse all nodes in a subtree rooted
    // with this node
    public void traverse()
    {
        // There are n keys and n+1 children, traverse
        // through n keys and first n children
        int i = 0;
        for (i = 0; i < this.n; i++) {
            // If this is not leaf, then before printing
            // key[i], traverse the subtree rooted with
            // child C[i].
            if (this.leaf == false) {
                C[i].traverse();
            }
            System.out.print(keys[i] + " ");
        }
        // Print the subtree rooted with last child
        if (leaf == false)
            C[i].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.

    {

        // 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 the 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);

    }

}
```

Note: The above code doesn't contain the driver program. We will be covering the complete program in B-Tree Insertion.
There are two conventions to define a B-Tree, one is to define by minimum degree, second is to define by order. We have followed the minimum degree convention and

will be following the same further. The variable names used in the above program are also kept the same.

Applications of B-Trees:

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

Advantages of B-Trees:

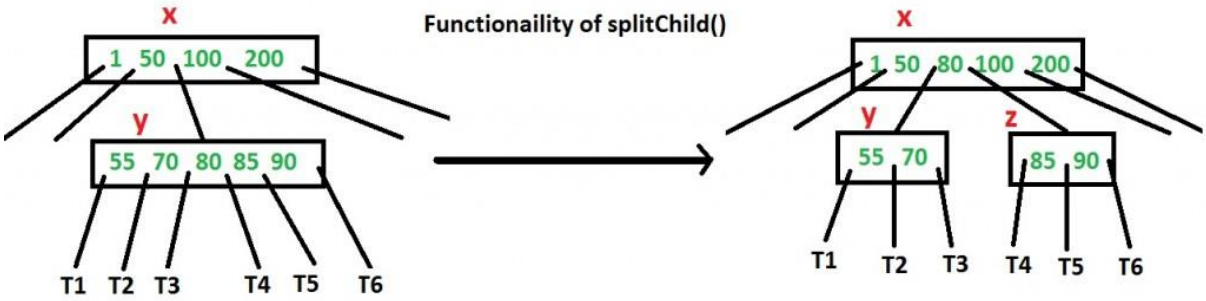
- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- Slow in comparison to other data structures.

Insert Operation in B-Tree:

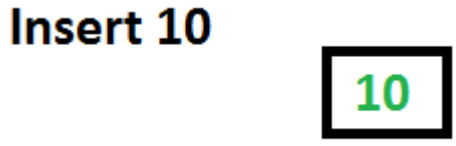
A new key is always inserted at the leaf node. Let the key to be inserted be k. Like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.
How to make sure that a node has space available for a key before the key is inserted? We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up, unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is the complete algorithm.

Insertion
1) Initialize x as root.
2) While x is not leaf, do following
..**a)** Find the child of x that is going to be traversed next. Let the child be y.
..**b)** If y is not full, change x to point to y.
..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.
Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if a new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from the root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because the parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.
Initially root is NULL. Let us first insert 10.

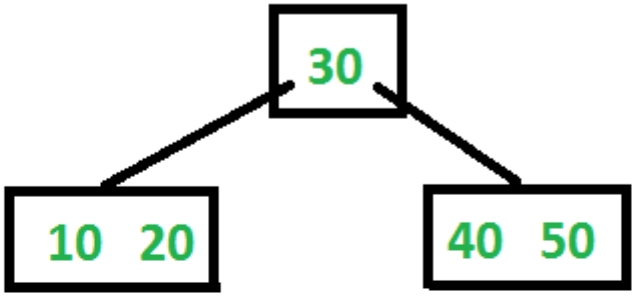


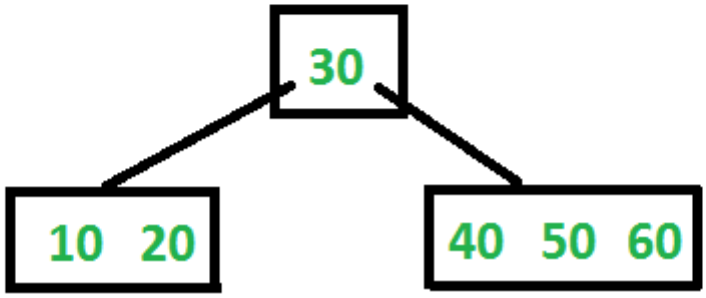
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50

Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

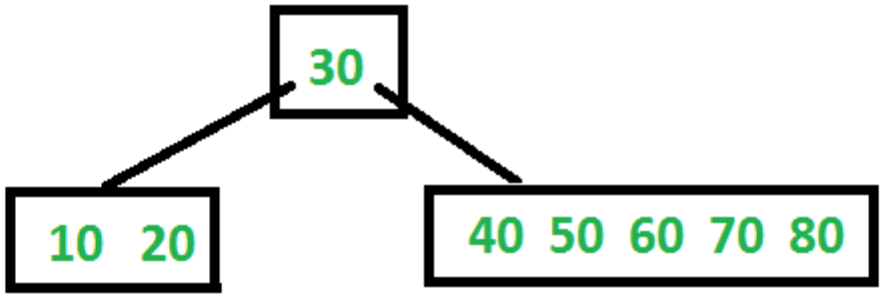
Insert 60



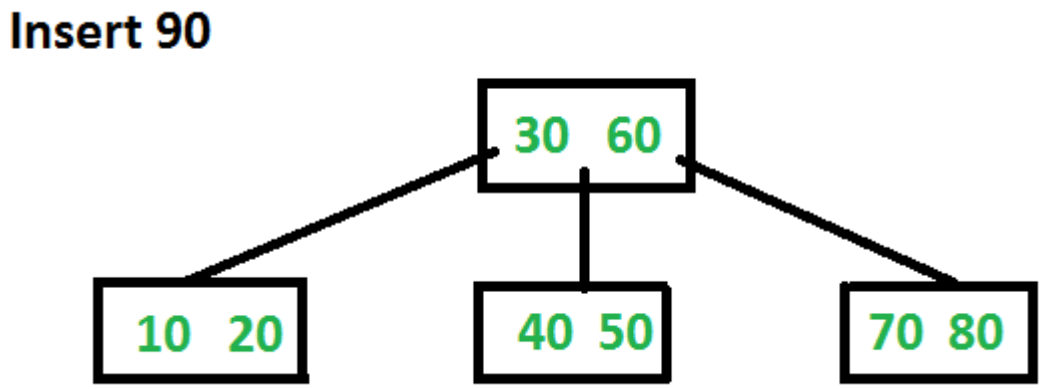


Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.



Following is C++ implementation of the above proactive algorithm.

```
// 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;

}
```

Output:

```
Traversal of the constructed tree is  5 6 7 10 12 17 20 30

Present

Not Present
```

Deletion Process in B-Trees:

Deletion from a B-tree is more complicated than insertion because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number t-1 of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t. Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

Various Cases of Deletion

Case 1: If the key k is in node x and x is a leaf, delete the key k from x.
Case 2: If the key k is in node x and x is an internal node, do the following.

- If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k with k0 in x. (We can find k0 and delete it in a single downward pass.)
- If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k with k0 in x. (We can find k0 and delete it in a single downward pass.)
- Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

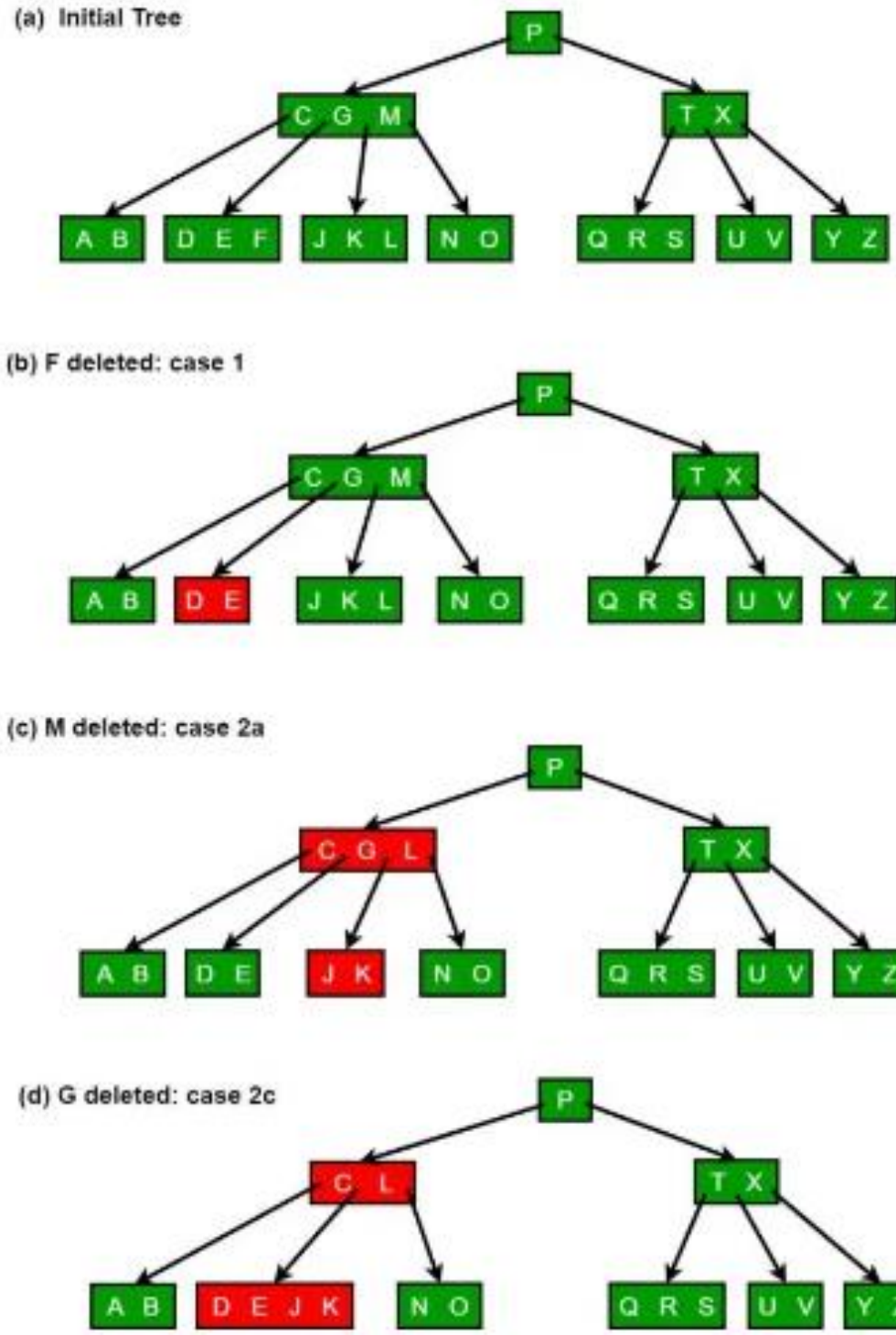
Case 3: If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute steps 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

- If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).
- If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

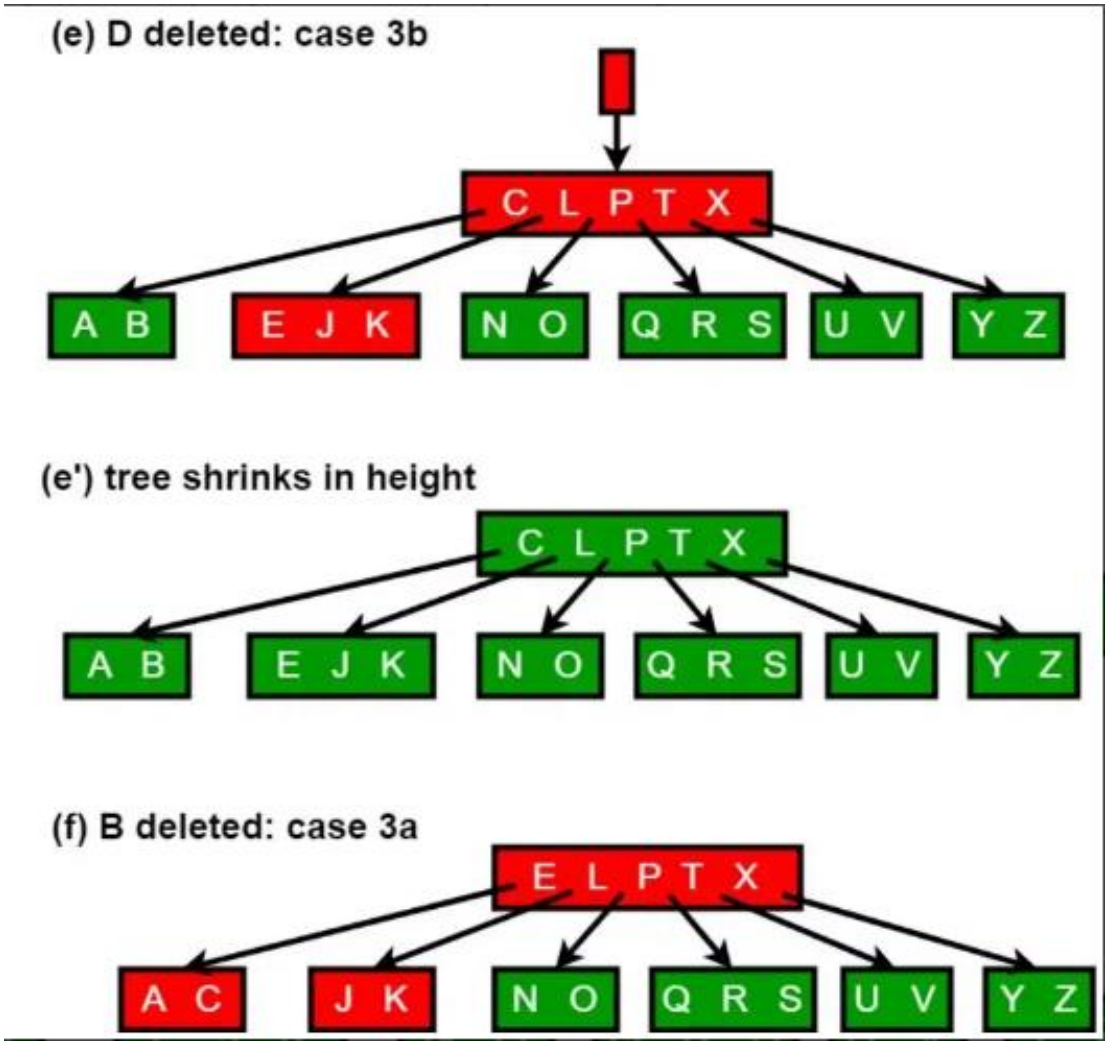
Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which

the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

The following figures explain the deletion process.



The next processes are shown below in the figure.



Deletion Operation in B+ Trees

Implementation

Following is the C++ implementation of the deletion process.

```
#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 function to traverse all nodes in a subtree rooted with this
node
    void traverse();
    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.
    // A function that returns the index of the first key that is
    // greater or equal to k
    int findKey(int k);
    // 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 wrapper function to remove the key k in subtree rooted with
// this node.
void remove(int k);
 // A function to remove the key present in idx-th position in
// this node which is a leaf
void removeFromLeaf(int idx);
 // A function to remove the key present in idx-th position in
// this node which is a non-leaf node
void removeFromNonLeaf(int idx);
 // A function to get the predecessor of the key- where the key
// is present in the idx-th position in the node
int getPred(int idx);
 // A function to get the successor of the key- where the key
// is present in the idx-th position in the node
int getSucc(int idx);
 // A function to fill up the child node present in the idx-th
// position in the C[] array if that child has less than t-1 keys
void fill(int idx);
```

```
    // A function to borrow a key from the C[idx-1]-th node and place

    // it in C[idx]th node

    void borrowFromPrev(int idx);

     // A function to borrow a key from the C[idx+1]-th node and place it

    // in C[idx]th node

    void borrowFromNext(int idx);

     // A function to merge idx-th child of the node with (idx+1)th child
of

    // the node

    void merge(int idx);

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

    // this class in BTree functions

    friend class 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;
    }
    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);
    // The main function that removes a new key in this B-Tree
    void remove(int k);
};
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;
}
 // A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreeNode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}
 // A function to remove the key k from the sub-tree rooted with this
node
void BTreeNode::remove(int k)
```

```
{
    int idx = findKey(k);
    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {
        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else
    {
        // If this node is a leaf node, then the key is not present in
tree
        if (leaf)
        {
            cout<<"The key "<<k<<" is does not exist in the tree\n";
            return;
        }
```

```
        // The key to be removed is present in the sub-tree rooted with
this node

        // The flag indicates whether the key is present in the sub-tree
rooted

        // with the last child of this node

        bool flag = ( (idx==n)? true : false );

        // If the child where the key is supposed to exist has less that
t keys,

        // we fill that child

        if (C[idx]->n < t)

            fill(idx);

        // If the last child has been merged, it must have merged with
the previous

        // child and so we recurse on the (idx-1)th child. Else, we
recurse on the

        // (idx)th child which now has atleast t keys

        if (flag && idx > n)

            C[idx-1]->remove(k);

        else

            C[idx]->remove(k);

    }

    return;

}
```

```
// A function to remove the idx-th key from this node - which is a leaf
node

void BTreeNode::removeFromLeaf (int idx)

{

    // Move all the keys after the idx-th pos one place backward

    for (int i=idx+1; i<n; ++i)

        keys[i-1] = keys[i];

    // Reduce the count of keys

    n--;

    return;

}

// A function to remove the idx-th key from this node - which is a non-
leaf node

void BTreeNode::removeFromNonLeaf(int idx)

{

    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,

    // find the predecessor 'pred' of k in the subtree rooted at

    // C[idx]. Replace k by pred. Recursively delete pred

    // in C[idx]

    if (C[idx]->n >= t)

    {
```

```
        int pred = getPred(idx);

        keys[idx] = pred;

        C[idx]->remove(pred);

    }

    // If the child C[idx] has less that t keys, examine C[idx+1].

    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in

    // the subtree rooted at C[idx+1]

    // Replace k by succ

    // Recursively delete succ in C[idx+1]

    else if (C[idx+1]->n >= t)

    {

        int succ = getSucc(idx);

        keys[idx] = succ;

        C[idx+1]->remove(succ);

    }

    // If both C[idx] and C[idx+1] has less that t keys,merge k and all
of C[idx+1]

    // into C[idx]

    // Now C[idx] contains 2t-1 keys

    // Free C[idx+1] and recursively delete k from C[idx]

    else

    {
```

```
        merge(idx);

        C[idx]->remove(k);

    }

    return;

}

// A function to get predecessor of keys[idx]

int BTreeNode::getPred(int idx)

{

    // Keep moving to the right most node until we reach a leaf

    BTreeNode *cur=C[idx];

    while (!cur->leaf)

        cur = cur->C[cur->n];

    // Return the last key of the leaf

    return cur->keys[cur->n-1];

}

int BTreeNode::getSucc(int idx)

{

    // Keep moving the left most node starting from C[idx+1] until we
reach a leaf

    BTreeNode *cur = C[idx+1];

    while (!cur->leaf)
```

```
        cur = cur->C[0];

    // Return the first key of the leaf

    return cur->keys[0];

}

// A function to fill child C[idx] which has less than t-1 keys

void BTreeNode::fill(int idx)

{

    // If the previous child(C[idx-1]) has more than t-1 keys, borrow a
key

    // from that child

    if (idx!=0 && C[idx-1]->n>=t)

        borrowFromPrev(idx);

    // If the next child(C[idx+1]) has more than t-1 keys, borrow a key

    // from that child

    else if (idx!=n && C[idx+1]->n>=t)

        borrowFromNext(idx);

    // Merge C[idx] with its sibling

    // If C[idx] is the last child, merge it with its previous sibling

    // Otherwise merge it with its next sibling

    else

    {
```

```
        if (idx != n)
            merge(idx);
        else
            merge(idx-1);
    }
    return;
}
// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
void BTreeNode::borrowFromPrev(int idx)
{
    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx-1];
    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus,
the  loses
    // sibling one key and child gains one key
    // Moving all key in C[idx] one step ahead
    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];
    // If C[idx] is not a leaf, move all its child pointers one step
ahead
```

```
    if (!child->leaf)
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }
     // Setting child's first key equal to keys[idx-1] from the current
node
    child->keys[0] = keys[idx-1];
     // Moving sibling's last child as C[idx]'s first child
    if(!child->leaf)
        child->C[0] = sibling->C[sibling->n];
     // Moving the key from the sibling to the parent
    // This reduces the number of keys in the sibling
    keys[idx-1] = sibling->keys[sibling->n-1];
     child->n += 1;
    sibling->n -= 1;
     return;
}
 // A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreeNode::borrowFromNext(int idx)
```

```
{

     BTreeNode *child=C[idx];

    BTreeNode *sibling=C[idx+1];

     // keys[idx] is inserted as the last key in C[idx]

    child->keys[(child->n)] = keys[idx];

     // Sibling's first child is inserted as the last child

    // into C[idx]

    if (!(child->leaf))

        child->C[(child->n)+1] = sibling->C[0];

     //The first key from sibling is inserted into keys[idx]

    keys[idx] = sibling->keys[0];

     // Moving all keys in sibling one step behind

    for (int i=1; i<sibling->n; ++i)

        sibling->keys[i-1] = sibling->keys[i];

     // Moving the child pointers one step behind

    if (!sibling->leaf)

    {

        for(int i=1; i<=sibling->n; ++i)

            sibling->C[i-1] = sibling->C[i];

    }
```

```
    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    // respectively
    child->n += 1;
    sibling->n -= 1;
     return;
}
 // A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreeNode::merge(int idx)
{
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx+1];
     // Pulling a key from the current node and inserting it into (t-1)th
    // position of C[idx]
    child->keys[t-1] = keys[idx];
     // Copying the keys from C[idx+1] to C[idx] at the end
    for (int i=0; i<sibling->n; ++i)
        child->keys[i+t] = sibling->keys[i];
     // Copying the child pointers from C[idx+1] to C[idx]
    if (!child->leaf)
```

```
    {
        for(int i=0; i<=sibling->n; ++i)
            child->C[i+t] = sibling->C[i];
    }
    // Moving all keys after idx in the current node one step before -
    // to fill the gap created by moving keys[idx] to C[idx]
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];
    // Moving the child pointers after (idx+1) in the current node one
    // step before
    for (int i=idx+2; i<=n; ++i)
        C[i-1] = C[i];
    // Updating the key count of child and the current node
    child->n += sibling->n+1;
    n--;
    // Freeing the memory occupied by sibling
    delete(sibling);
    return;
}
// 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 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;

}

// 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);
}
```

```
void BTree::remove(int k)

{

    if (!root)

    {

        cout << "The tree is empty\n";

        return;

    }



    // Call the remove function for root

    root->remove(k);



    // If the root node has 0 keys, make its first child as the new root

    //  if it has a child, otherwise set root as NULL

    if (root->n==0)

    {

        BTreeNode *tmp = root;

        if (root->leaf)

            root = NULL;

        else

            root = root->C[0];
```

```
        // Free the old root
        delete tmp;
    }
    return;
}


// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minimum degree 3
     t.insert(1);
    t.insert(3);
    t.insert(7);
    t.insert(10);
    t.insert(11);
    t.insert(13);
    t.insert(14);
    t.insert(15);
    t.insert(18);
    t.insert(16);
    t.insert(19);
```

```
    t.insert(24);

    t.insert(25);

    t.insert(26);

    t.insert(21);

    t.insert(4);

    t.insert(5);

    t.insert(20);

    t.insert(22);

    t.insert(2);

    t.insert(17);

    t.insert(12);

    t.insert(6);

     cout << "Traversal of tree constructed is\n";

    t.traverse();

    cout << endl;

     t.remove(6);

    cout << "Traversal of tree after removing 6\n";

    t.traverse();

    cout << endl;

     t.remove(13);

    cout << "Traversal of tree after removing 13\n";
```

```
    t.traverse();

    cout << endl;

     t.remove(7);

    cout << "Traversal of tree after removing 7\n";

    t.traverse();

    cout << endl;

     t.remove(4);

    cout << "Traversal of tree after removing 4\n";

    t.traverse();

    cout << endl;

     t.remove(2);

    cout << "Traversal of tree after removing 2\n";

    t.traverse();

    cout << endl;

     t.remove(16);

    cout << "Traversal of tree after removing 16\n";

    t.traverse();

    cout << endl;

     return 0;

}
```

Output:

```
Traversal of tree constructed is
 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 6
 1 2 3 4 5 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 13
 1 2 3 4 5 7 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 7
 1 2 3 4 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 4
 1 2 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 2
 1 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 16
 1 3 5 10 11 12 14 15 17 18 19 20 21 22 24 25 26
```

Summary:

The B-tree is a data structure that is similar to a binary search tree, but allows multiple keys per node and has a higher fanout. This allows the B-tree to store a large amount of data in an efficient manner, and it is commonly used in database and file systems.

The B-tree is a balanced tree, which means that all paths from the root to a leaf have the same length. The tree has a minimum degree t, which is the minimum number of keys in a non-root node. Each node can have at most 2t-1 keys and 2t children. The root can have at least one key and at most 2t-1 keys. All non-root nodes have at least t-1 keys and at most 2t-1 keys.

The B-tree supports the following operations:

Search(k): search for a key k in the tree.
Insert(k): insert a key k into the tree.
Delete(k): delete a key k from the tree.
The search operation is similar to that of a binary search tree. The insert operation is more complicated, since inserting a key can cause a node to become full. If a node is full, it must be split into two nodes, and the median key moved up to the parent node. The delete operation is also more complicated, since deleting a key can cause a node to have too few keys. If a node has too few keys, it can be merged with a sibling node or borrow a key from a sibling node.

The B-tree has a number of advantages over other data structures. It has a higher fanout than binary search trees, which means that fewer disk accesses are required to search for a key. It is also a balanced tree, which means that all operations have a

worst-case time complexity of O(log n). Finally, the B-tree is self-adjusting, which means that it can adapt to changes in the data set without requiring expensive rebalancing operations.