Introdução

Uma Árvore Binária de Busca (BST) é uma estrutura de dados que armazena valores em uma estrutura semelhante a uma árvore binária, onde cada nó possui no máximo dois nós filhos. Os nós em um BST são ordenados de tal forma que, para qualquer nó dado, todos os valores em sua subárvore esquerda são menores que o valor do nó e todos os valores em sua subárvore direita são maiores que o valor do nó. Os BSTs são simples de entender e implementar, mas podem ficar desbalanceados quando a árvore cresce, o que pode resultar em baixo desempenho.

Uma árvore AVL é um tipo de BST que usa rotações para manter o equilíbrio. As árvores AVL recebem o nome de seus inventores, G.M. Adelson-Velsky e E.M. Landis. As árvores AVL garantem que a altura da árvore seja sempre logarítmica no número de elementos, o que resulta em operações rápidas de pesquisa, inserção e exclusão.

Uma árvore rubro-negra é um tipo de BST que usa o conceito de nós de coloração como vermelho ou preto para manter o equilíbrio. As árvores rubro-negras garantem que a altura da árvore seja logarítmica no número de elementos, assim como as árvores AVL. As árvores rubro-negras são mais simples de implementar do que as árvores AVL e são usadas em muitas bibliotecas e aplicativos.

Uma B-Tree é um tipo de estrutura de dados usada para armazenamento baseado em disco. B-Trees são otimizados para sistemas onde os dados precisam ser lidos e gravados em um disco e onde é lento acessar elementos individuais. As B-Trees mantêm o equilíbrio permitindo que os nós tenham mais de dois filhos e usam um método de divisão e fusão de nós para manter a altura da árvore pequena. B-Trees são usados em sistemas de banco de dados e sistemas de arquivos para armazenar grandes quantidades de dados.

Em resumo, todas essas estruturas de dados são usadas para pesquisar, inserir e excluir valores com eficiência, mas cada uma delas tem um foco diferente e é otimizada para diferentes casos de uso. Os BSTs são simples, mas podem se tornar desbalanceados, as árvores AVL garantem altura logarítmica, as árvores rubro-negras são mais simples de implementar e têm altura logarítmica e as árvores B são otimizadas para armazenamento baseado em disco.

Complexidade:

A complexidade para as operações básicas em cada uma dessas estruturas de dados é a seguinte:

Árvore de busca binária:

Busca: O(n) no pior caso (quando a árvore está desbalanceada), O(log n) no caso médio

Inserção: O(n) no pior caso (quando a árvore está desbalanceada), O(log n) no caso médio

Deleção: O(n) no pior caso (quando a árvore está desbalanceada), O(log n) no caso médio

Árvore AVL:

Pesquisa: O(log n) no pior caso e no caso médio

Inserção: O(log n) no pior caso e no caso médio

Exclusão: O(log n) no pior caso e no caso médio

Árvore Rubro-negra:

Pesquisa: O(log n) no pior caso e no caso médio

<u>Inserção: O(log n) no pior caso e no caso médio</u>

Exclusão: O(log n) no pior caso e no caso médio

Árvore B:

Pesquisa: O(log n) no caso médio

Inserção: O(log n) no caso médio

Deleção: O(log n) no caso médio

A complexidade para cada uma dessas estruturas de dados depende da altura da árvore. No pior caso, a altura da árvore pode ser proporcional ao número de elementos, o que resultaria em complexidade de tempo linear para todas as operações. No entanto, a altura é mantida logarítmica no número de elementos na maioria dos casos, o que resulta em complexidade de tempo logarítmica.

ABB:

```java
import java.io.*;

// Java program for Delete a Node of BST
class GFG {

    // Given Node node
    static class node {
        int key;
        node left, right;
    };

    // Function to create a new BST node
    static node newNode(int item)
    {
        node temp = new node();
        temp.key = item;
        temp.left = temp.right = null;
        return temp;
    }

    // Function to insert a new node with
    // given key in BST
    static node insert(node node, int key)
    {
        // If the tree is empty, return a new node
        if (node == null)
            return newNode(key);

        // Otherwise, recur down the tree
        if (key < node.key) {
            node.left = insert(node.left, key);
        }
        else if (key > node.key) {
            node.right = insert(node.right, key);
        }

        // Return the node
        return node;
    }

    // Function to do inorder traversal of BST
    static void inorder(node root)
```

```java
{
    if (root != null) {
        inorder(root.left);
        System.out.print(" " + root.key);
        inorder(root.right);
    }
}
// Function that returns the node with minimum
// key value found in that tree
static node minValueNode(node node)
{
    node current = node;

    // Loop down to find the leftmost leaf
    while (current != null && current.left != null)
        current = current.left;

    return current;
}

// Function that deletes the key and
// returns the new root
static node deleteNode(node root, int key)
{
    // base Case
    if (root == null)
        return root;

    // If the key to be deleted is
    // smaller than the root's key,
    // then it lies in left subtree
    if (key < root.key) {
        root.left = deleteNode(root.left, key);
    }

    // If the key to be deleted is
    // greater than the root's key,
    // then it lies in right subtree
    else if (key > root.key) {

        root.right = deleteNode(root.right, key);
    }

    // If key is same as root's key,
    // then this is the node
    // to be deleted
    else {
```

```
            // Node with only one child
            // or no child
            if (root.left == null) {
                node temp = root.right;
                return temp;
            }
            else if (root.right == null) {
                node temp = root.left;
                return temp;
            }

            // Node with two children:
            // Get the inorder successor(smallest
            // in the right subtree)
            node temp = minValueNode(root.right);

            // Copy the inorder successor's
            // content to this node
            root.key = temp.key;

            // Delete the inorder successor
            root.right = deleteNode(root.right, temp.key);
        }
        return root;
    }

    // Driver Code
    public static void main(String[] args)
    {

        /* Let us create following BST
                50
             /      \
            30      70
           /  \    /  \
         20   40  60   80
     */
        node root = null;

        // inserting value 50
        root = insert(root, 50);

        // inserting value 30
        insert(root, 30);

        // inserting value 20
        insert(root, 20);
```

```
        // inserting value 40
        insert(root, 40);

        // inserting value 70
        insert(root, 70);

        // inserting value 60
        insert(root, 60);

        // inserting value 80
        insert(root, 80);

        // Function Call
        root = deleteNode(root, 60);
        inorder(root);
    }
}
```

AVL:

```
class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}

class AVLTree {

    Node root;

    // A utility function to get the height of the tree
    int height(Node N) {
        if (N == null)
            return 0;

        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b) {
        return (a > b) ? a : b;
```

```java
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        //  Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        // Return new root
        return y;
    }

    // Get Balance factor of node N
    int getBalance(Node N) {
        if (N == null)
            return 0;

        return height(N.left) - height(N.right);
    }

    Node insert(Node node, int key) {
```

```
        /* 1.  Perform the normal BST insertion */
        if (node == null)
            return (new Node(key));

        if (key < node.key)
            node.left = insert(node.left, key);
        else if (key > node.key)
            node.right = insert(node.right, key);
        else // Duplicate keys not allowed
            return node;

        /* 2. Update height of this ancestor node */
        node.height = 1 + max(height(node.left),
                              height(node.right));

        /* 3. Get the balance factor of this ancestor
              node to check whether this node became
              unbalanced */
        int balance = getBalance(node);

        // If this node becomes unbalanced, then there
        // are 4 cases Left Left Case
        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        // Right Right Case
        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node.left.key) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node.right.key) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
    }

    // A utility function to print preorder traversal
    // of the tree.
    // The function also prints height of every node
```

```java
    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    public static void main(String[] args) {
        AVLTree tree = new AVLTree();

        /* Constructing tree given in the above figure */
        tree.root = tree.insert(tree.root, 10);
        tree.root = tree.insert(tree.root, 20);
        tree.root = tree.insert(tree.root, 30);
        tree.root = tree.insert(tree.root, 40);
        tree.root = tree.insert(tree.root, 50);
        tree.root = tree.insert(tree.root, 25);

        /* The constructed AVL Tree would be
             30
            /  \
          20   40
         /  \     \
        10  25    50
        */
        System.out.println("Preorder traversal" +
                        " of constructed tree is : ");
        tree.preOrder(tree.root);
    }
}
```

```java
class Node
{
    int key, height;
    Node left, right;

    Node(int d)
    {
        key = d;
        height = 1;
    }
}

class AVLTree
{
```

```java
    Node root;

    // A utility function to get height of the tree
    int height(Node N)
    {
        if (N == null)
            return 0;
        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y)
    {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x)
    {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
```

```
        y.height = max(height(y.left), height(y.right)) + 1;

        // Return new root
        return y;
    }

    // Get Balance factor of node N
    int getBalance(Node N)
    {
        if (N == null)
            return 0;
        return height(N.left) - height(N.right);
    }

    Node insert(Node node, int key)
    {
        /* 1. Perform the normal BST rotation */
        if (node == null)
            return (new Node(key));

        if (key < node.key)
            node.left = insert(node.left, key);
        else if (key > node.key)
            node.right = insert(node.right, key);
        else // Equal keys not allowed
            return node;

        /* 2. Update height of this ancestor node */
        node.height = 1 + max(height(node.left),
                              height(node.right));

        /* 3. Get the balance factor of this ancestor
        node to check whether this node became
        Wunbalanced */
        int balance = getBalance(node);

        // If this node becomes unbalanced, then
        // there are 4 cases Left Left Case
        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        // Right Right Case
        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node.left.key)
        {
```

```java
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node.right.key)
        {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
    }

    /* Given a non-empty binary search tree, return the
    node with minimum key value found in that tree.
    Note that the entire tree does not need to be
    searched. */
    Node minValueNode(Node node)
    {
        Node current = node;

        /* loop down to find the leftmost leaf */
        while (current.left != null)
        current = current.left;

        return current;
    }

    Node deleteNode(Node root, int key)
    {
        // STEP 1: PERFORM STANDARD BST DELETE
        if (root == null)
            return root;

        // If the key to be deleted is smaller than
        // the root's key, then it lies in left subtree
        if (key < root.key)
            root.left = deleteNode(root.left, key);

        // If the key to be deleted is greater than the
        // root's key, then it lies in right subtree
        else if (key > root.key)
            root.right = deleteNode(root.right, key);

        // if key is same as root's key, then this is the node
        // to be deleted
```

```
        else
        {

            // node with only one child or no child
            if ((root.left == null) || (root.right == null))
            {
                Node temp = null;
                if (temp == root.left)
                    temp = root.right;
                else
                    temp = root.left;

                // No child case
                if (temp == null)
                {
                    temp = root;
                    root = null;
                }
                else // One child case
                    root = temp; // Copy the contents of
                                 // the non-empty child
            }
            else
            {

                // node with two children: Get the inorder
                // successor (smallest in the right subtree)
                Node temp = minValueNode(root.right);

                // Copy the inorder successor's data to this node
                root.key = temp.key;

                // Delete the inorder successor
                root.right = deleteNode(root.right, temp.key);
            }
        }

        // If the tree had only one node then return
        if (root == null)
            return root;

        // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
        root.height = max(height(root.left), height(root.right)) + 1;

        // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check
whether
        // this node became unbalanced)
        int balance = getBalance(root);
```

```java
        // If this node becomes unbalanced, then there are 4 cases
        // Left Left Case
        if (balance > 1 && getBalance(root.left) >= 0)
            return rightRotate(root);

        // Left Right Case
        if (balance > 1 && getBalance(root.left) < 0)
        {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && getBalance(root.right) <= 0)
            return leftRotate(root);

        // Right Left Case
        if (balance < -1 && getBalance(root.right) > 0)
        {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }

        return root;
    }

    // A utility function to print preorder traversal of
    // the tree. The function also prints height of every
    // node
    void preOrder(Node node)
    {
        if (node != null)
        {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    public static void main(String[] args)
    {
        AVLTree tree = new AVLTree();

        /* Constructing tree given in the above figure */
        tree.root = tree.insert(tree.root, 9);
        tree.root = tree.insert(tree.root, 5);
        tree.root = tree.insert(tree.root, 10);
```

```java
        tree.root = tree.insert(tree.root, 0);
        tree.root = tree.insert(tree.root, 6);
        tree.root = tree.insert(tree.root, 11);
        tree.root = tree.insert(tree.root, -1);
        tree.root = tree.insert(tree.root, 1);
        tree.root = tree.insert(tree.root, 2);

        /* The constructed AVL Tree would be
        9
        / \
        1 10
        / \ \
        0 5 11
        / / \
        -1 2 6
        */
        System.out.println("Preorder traversal of "+
                            "constructed tree is : ");
        tree.preOrder(tree.root);

        tree.root = tree.deleteNode(tree.root, 10);

        /* The AVL Tree after deletion of 10
        1
        / \
        0 9
        /     / \
        -1 5 11
        / \
        2 6
        */
        System.out.println("");
        System.out.println("Preorder traversal after "+
                        "deletion of 10 :");
        tree.preOrder(tree.root);
    }
}
```

RB:

```java
import java.io.*;

// considering that you know what are red-black trees here is the
implementation in java for insertion and traversal.
// RedBlackTree class. This class contains subclass for node
```

```java
// as well as all the functionalities of RedBlackTree such as -
rotations, insertion and
// inoredr traversal
public class RedBlackTree
{
    public Node root;//root node
    public RedBlackTree()
    {
        super();
        root = null;
    }
    // node creating subclass
    class Node
    {
        int data;
        Node left;
        Node right;
        char colour;
        Node parent;

        Node(int data)
        {
            super();
            this.data = data;   // only including data. not key
            this.left = null; // left subtree
            this.right = null; // right subtree
            this.colour = 'R'; // colour . either 'R' or 'B'
            this.parent = null; // required at time of rechecking.
        }
    }
    // this function performs left rotation
    Node rotateLeft(Node node)
    {
        Node x = node.right;
        Node y = x.left;
        x.left = node;
        node.right = y;
        node.parent = x; // parent resetting is also important.
        if(y!=null)
            y.parent = node;
        return(x);
    }
    //this function performs right rotation
    Node rotateRight(Node node)
    {
        Node x = node.left;
        Node y = x.right;
        x.right = node;
```

```
        node.left = y;
        node.parent = x;
        if(y!=null)
            y.parent = node;
        return(x);
    }


    // these are some flags.
    // Respective rotations are performed during traceback.
    // rotations are done if flags are true.
    boolean ll = false;
    boolean rr = false;
    boolean lr = false;
    boolean rl = false;
    // helper function for insertion. Actually this function performs
all tasks in single pass only.
    Node insertHelp(Node root, int data)
    {
        // f is true when RED RED conflict is there.
        boolean f=false;

        //recursive calls to insert at proper position according to
BST properties.
        if(root==null)
            return(new Node(data));
        else if(data<root.data)
        {
            root.left = insertHelp(root.left, data);
            root.left.parent = root;
            if(root!=this.root)
            {
                if(root.colour=='R' && root.left.colour=='R')
                    f = true;
            }
        }
        else
        {
            root.right = insertHelp(root.right,data);
            root.right.parent = root;
            if(root!=this.root)
            {
                if(root.colour=='R' && root.right.colour=='R')
                    f = true;
            }
        // at the same time of insertion, we are also assigning
parent nodes
        // also we are checking for RED RED conflicts
```

```
        }

        // now lets rotate.
        if(this.ll) // for left rotate.
        {
            root = rotateLeft(root);
            root.colour = 'B';
            root.left.colour = 'R';
            this.ll = false;
        }
        else if(this.rr) // for right rotate
        {
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.rr  = false;
        }
        else if(this.rl)  // for right and then left
        {
            root.right = rotateRight(root.right);
            root.right.parent = root;
            root = rotateLeft(root);
            root.colour = 'B';
            root.left.colour = 'R';

            this.rl = false;
        }
        else if(this.lr)  // for left and then right.
        {
            root.left = rotateLeft(root.left);
            root.left.parent = root;
            root = rotateRight(root);
            root.colour = 'B';
            root.right.colour = 'R';
            this.lr = false;
        }
        // when rotation and recolouring is done flags are reset.
        // Now lets take care of RED RED conflict
        if(f)
        {
            if(root.parent.right == root)  // to check which child is
the current node of its parent
            {
                if(root.parent.left==null ||
root.parent.left.colour=='B')  // case when parent's sibling is black
                {// perform certaing rotation and recolouring. This
will be done while backtracking. Hence setting up respective flags.
                    if(root.left!=null && root.left.colour=='R')
```

```
                        this.rl = true;
                    else if(root.right!=null &&
root.right.colour=='R')
                        this.ll = true;
                }
                else // case when parent's sibling is red
                {
                    root.parent.left.colour = 'B';
                    root.colour = 'B';
                    if(root.parent!=this.root)
                        root.parent.colour = 'R';
                }
            }
            else
            {
                if(root.parent.right==null ||
root.parent.right.colour=='B')
                {
                    if(root.left!=null && root.left.colour=='R')
                        this.rr = true;
                    else if(root.right!=null &&
root.right.colour=='R')
                        this.lr = true;
                }
                else
                {
                    root.parent.right.colour = 'B';
                    root.colour = 'B';
                    if(root.parent!=this.root)
                        root.parent.colour = 'R';
                }
            }
            f = false;
        }
        return(root);
    }

    // function to insert data into tree.
    public void insert(int data)
    {
        if(this.root==null)
        {
            this.root = new Node(data);
            this.root.colour = 'B';
        }
        else
            this.root = insertHelp(this.root,data);
    }
```

```
    // helper function to print inorder traversal
    void inorderTraversalHelper(Node node)
    {
        if(node!=null)
        {
            inorderTraversalHelper(node.left);
            System.out.printf("%d ", node.data);
            inorderTraversalHelper(node.right);
        }
    }
    //function to print inorder traversal
    public void inorderTraversal()
    {
        inorderTraversalHelper(this.root);
    }
    // helper function to print the tree.
    void printTreeHelper(Node root, int space)
    {
        int i;
        if(root != null)
        {
            space = space + 10;
            printTreeHelper(root.right, space);
            System.out.printf("\n");
            for ( i = 10; i < space; i++)
            {
                System.out.printf(" ");
            }
            System.out.printf("%d", root.data);
            System.out.printf("\n");
            printTreeHelper(root.left, space);
        }
    }
    // function to print the tree.
    public void printTree()
    {
        printTreeHelper(this.root, 0);
    }
    public static void main(String[] args)
    {
        // let us try to insert some data into tree and try to
visualize the tree as well as traverse.
        RedBlackTree t = new RedBlackTree();
        int[] arr = {1,4,6,3,5,7,8,2,9};
        for(int i=0;i<9;i++)
        {
            t.insert(arr[i]);
            System.out.println();
```

```
            t.inorderTraversal();
        }
        // you can check colour of any node by with its attribute
node.colour
        t.printTree();
    }
}
```

```
#include <iostream>
#include <queue>
using namespace std;

enum COLOR { RED, BLACK };

class Node {
public:
  int val;
  COLOR color;
  Node *left, *right, *parent;

  Node(int val) : val(val) {
    parent = left = right = NULL;

    // Node is created during insertion
    // Node is red at insertion
    color = RED;
  }

  // returns pointer to uncle
  Node *uncle() {
    // If no parent or grandparent, then no uncle
    if (parent == NULL or parent->parent == NULL)
      return NULL;

    if (parent->isOnLeft())
      // uncle on right
      return parent->parent->right;
    else
      // uncle on left
      return parent->parent->left;
  }

  // check if node is left child of parent
  bool isOnLeft() { return this == parent->left; }

  // returns pointer to sibling
```

```cpp
  Node *sibling() {
    // sibling null if no parent
    if (parent == NULL)
      return NULL;

    if (isOnLeft())
      return parent->right;

    return parent->left;
  }

  // moves node down and moves given node in its place
  void moveDown(Node *nParent) {
    if (parent != NULL) {
      if (isOnLeft()) {
        parent->left = nParent;
      } else {
        parent->right = nParent;
      }
    }
    nParent->parent = parent;
    parent = nParent;
  }

  bool hasRedChild() {
    return (left != NULL and left->color == RED) or
           (right != NULL and right->color == RED);
  }
};

class RBTree {
  Node *root;

  // left rotates the given node
  void leftRotate(Node *x) {
    // new parent will be node's right child
    Node *nParent = x->right;

    // update root if current node is root
    if (x == root)
      root = nParent;

    x->moveDown(nParent);

    // connect x with new parent's left element
    x->right = nParent->left;
    // connect new parent's left element with node
    // if it is not null
```

```
    if (nParent->left != NULL)
      nParent->left->parent = x;

    // connect new parent with x
    nParent->left = x;
  }

  void rightRotate(Node *x) {
    // new parent will be node's left child
    Node *nParent = x->left;

    // update root if current node is root
    if (x == root)
      root = nParent;

    x->moveDown(nParent);

    // connect x with new parent's right element
    x->left = nParent->right;
    // connect new parent's right element with node
    // if it is not null
    if (nParent->right != NULL)
      nParent->right->parent = x;

    // connect new parent with x
    nParent->right = x;
  }

  void swapColors(Node *x1, Node *x2) {
    COLOR temp;
    temp = x1->color;
    x1->color = x2->color;
    x2->color = temp;
  }

  void swapValues(Node *u, Node *v) {
    int temp;
    temp = u->val;
    u->val = v->val;
    v->val = temp;
  }

  // fix red red at given node
  void fixRedRed(Node *x) {
    // if x is root color it black and return
    if (x == root) {
      x->color = BLACK;
      return;
```

```
  }

  // initialize parent, grandparent, uncle
  Node *parent = x->parent, *grandparent = parent->parent,
       *uncle = x->uncle();

  if (parent->color != BLACK) {
    if (uncle != NULL && uncle->color == RED) {
      // uncle red, perform recoloring and recurse
      parent->color = BLACK;
      uncle->color = BLACK;
      grandparent->color = RED;
      fixRedRed(grandparent);
    } else {
      // Else perform LR, LL, RL, RR
      if (parent->isOnLeft()) {
        if (x->isOnLeft()) {
          // for left right
          swapColors(parent, grandparent);
        } else {
          leftRotate(parent);
          swapColors(x, grandparent);
        }
        // for left left and left right
        rightRotate(grandparent);
      } else {
        if (x->isOnLeft()) {
          // for right left
          rightRotate(parent);
          swapColors(x, grandparent);
        } else {
          swapColors(parent, grandparent);
        }

        // for right right and right left
        leftRotate(grandparent);
      }
    }
  }
}

// find node that do not have a left child
// in the subtree of the given node
Node *successor(Node *x) {
  Node *temp = x;

  while (temp->left != NULL)
    temp = temp->left;
```

```
    return temp;
  }

  // find node that replaces a deleted node in BST
  Node *BSTreplace(Node *x) {
    // when node have 2 children
    if (x->left != NULL and x->right != NULL)
      return successor(x->right);

    // when leaf
    if (x->left == NULL and x->right == NULL)
      return NULL;

    // when single child
    if (x->left != NULL)
      return x->left;
    else
      return x->right;
  }

  // deletes the given node
  void deleteNode(Node *v) {
    Node *u = BSTreplace(v);

    // True when u and v are both black
    bool uvBlack = ((u == NULL or u->color == BLACK) and (v->color ==
BLACK));
    Node *parent = v->parent;

    if (u == NULL) {
      // u is NULL therefore v is leaf
      if (v == root) {
        // v is root, making root null
        root = NULL;
      } else {
        if (uvBlack) {
          // u and v both black
          // v is leaf, fix double black at v
          fixDoubleBlack(v);
        } else {
          // u or v is red
          if (v->sibling() != NULL)
            // sibling is not null, make it red"
            v->sibling()->color = RED;
        }

        // delete v from the tree
```

```
        if (v->isOnLeft()) {
          parent->left = NULL;
        } else {
          parent->right = NULL;
        }
      }
      delete v;
      return;
    }

    if (v->left == NULL or v->right == NULL) {
      // v has 1 child
      if (v == root) {
        // v is root, assign the value of u to v, and delete u
        v->val = u->val;
        v->left = v->right = NULL;
        delete u;
      } else {
        // Detach v from tree and move u up
        if (v->isOnLeft()) {
          parent->left = u;
        } else {
          parent->right = u;
        }
        delete v;
        u->parent = parent;
        if (uvBlack) {
          // u and v both black, fix double black at u
          fixDoubleBlack(u);
        } else {
          // u or v red, color u black
          u->color = BLACK;
        }
      }
      return;
    }

    // v has 2 children, swap values with successor and recurse
    swapValues(u, v);
    deleteNode(u);
  }

  void fixDoubleBlack(Node *x) {
    if (x == root)
      // Reached root
      return;

    Node *sibling = x->sibling(), *parent = x->parent;
```

```
    if (sibling == NULL) {
      // No sibiling, double black pushed up
      fixDoubleBlack(parent);
    } else {
      if (sibling->color == RED) {
        // Sibling red
        parent->color = RED;
        sibling->color = BLACK;
        if (sibling->isOnLeft()) {
          // left case
          rightRotate(parent);
        } else {
          // right case
          leftRotate(parent);
        }
        fixDoubleBlack(x);
      } else {
        // Sibling black
        if (sibling->hasRedChild()) {
          // at least 1 red children
          if (sibling->left != NULL and sibling->left->color == RED)
{
            if (sibling->isOnLeft()) {
              // left left
              sibling->left->color = sibling->color;
              sibling->color = parent->color;
              rightRotate(parent);
            } else {
              // right left
              sibling->left->color = parent->color;
              rightRotate(sibling);
              leftRotate(parent);
            }
          } else {
            if (sibling->isOnLeft()) {
              // left right
              sibling->right->color = parent->color;
              leftRotate(sibling);
              rightRotate(parent);
            } else {
              // right right
              sibling->right->color = sibling->color;
              sibling->color = parent->color;
              leftRotate(parent);
            }
          }
          parent->color = BLACK;
        } else {
```

```
        // 2 black children
        sibling->color = RED;
        if (parent->color == BLACK)
          fixDoubleBlack(parent);
        else
          parent->color = BLACK;
      }
    }
  }
}

// prints level order for given node
void levelOrder(Node *x) {
  if (x == NULL)
    // return if node is null
    return;

  // queue for level order
  queue<Node *> q;
  Node *curr;

  // push x
  q.push(x);

  while (!q.empty()) {
    // while q is not empty
    // dequeue
    curr = q.front();
    q.pop();

    // print node value
    cout << curr->val << " ";

    // push children to queue
    if (curr->left != NULL)
      q.push(curr->left);
    if (curr->right != NULL)
      q.push(curr->right);
  }
}

// prints inorder recursively
void inorder(Node *x) {
  if (x == NULL)
    return;
  inorder(x->left);
  cout << x->val << " ";
  inorder(x->right);
```

```
  }

public:
  // constructor
  // initialize root
  RBTree() { root = NULL; }

  Node *getRoot() { return root; }

  // searches for given value
  // if found returns the node (used for delete)
  // else returns the last node while traversing (used in insert)
  Node *search(int n) {
    Node *temp = root;
    while (temp != NULL) {
      if (n < temp->val) {
        if (temp->left == NULL)
          break;
        else
          temp = temp->left;
      } else if (n == temp->val) {
        break;
      } else {
        if (temp->right == NULL)
          break;
        else
          temp = temp->right;
      }
    }

    return temp;
  }

  // inserts the given value to tree
  void insert(int n) {
    Node *newNode = new Node(n);
    if (root == NULL) {
      // when root is null
      // simply insert value at root
      newNode->color = BLACK;
      root = newNode;
    } else {
      Node *temp = search(n);

      if (temp->val == n) {
        // return if value already exists
        return;
      }
```

```
    // if value is not found, search returns the node
    // where the value is to be inserted

    // connect new node to correct node
    newNode->parent = temp;

    if (n < temp->val)
      temp->left = newNode;
    else
      temp->right = newNode;

    // fix red red voilaton if exists
    fixRedRed(newNode);
  }
}

// utility function that deletes the node with given value
void deleteByVal(int n) {
  if (root == NULL)
    // Tree is empty
    return;

  Node *v = search(n), *u;

  if (v->val != n) {
    cout << "No node found to delete with value:" << n << endl;
    return;
  }

  deleteNode(v);
}

// prints inorder of the tree
void printInOrder() {
  cout << "Inorder: " << endl;
  if (root == NULL)
    cout << "Tree is empty" << endl;
  else
    inorder(root);
  cout << endl;
}

// prints level order of the tree
void printLevelOrder() {
  cout << "Level order: " << endl;
  if (root == NULL)
    cout << "Tree is empty" << endl;
```

```
    else
      levelOrder(root);
    cout << endl;
  }
};

int main() {
  RBTree tree;

  tree.insert(7);
  tree.insert(3);
  tree.insert(18);
  tree.insert(10);
  tree.insert(22);
  tree.insert(8);
  tree.insert(11);
  tree.insert(26);
  tree.insert(2);
  tree.insert(6);
  tree.insert(13);

  tree.printInOrder();
  tree.printLevelOrder();

  cout<<endl<<"Deleting 18, 11, 3, 10, 22"<<endl;

  tree.deleteByVal(18);
  tree.deleteByVal(11);
  tree.deleteByVal(3);
  tree.deleteByVal(10);
  tree.deleteByVal(22);

  tree.printInOrder();
  tree.printLevelOrder();
  return 0;
}
```

B:

```
class BTreeNode:
    def __init__(self, t, leaf):
        self.keys = [None] * (2 * t - 1) # An array of keys
        self.t = t # Minimum degree (defines the range for number of
keys)
        self.C = [None] * (2 * t) # An array of child pointers
```

```
        self.n = 0 # Current number of keys
        self.leaf = leaf # Is true when node is leaf. Otherwise false

    # 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
    def insertNonFull(self, k):
        i = self.n - 1
        if self.leaf:
            while i >= 0 and self.keys[i] > k:
                self.keys[i + 1] = self.keys[i]
                i -= 1
            self.keys[i + 1] = k
            self.n += 1
        else:
            while i >= 0 and self.keys[i] > k:
                i -= 1
            if self.C[i + 1].n == 2 * self.t - 1:
                self.splitChild(i + 1, self.C[i + 1])
                if self.keys[i + 1] < k:
                    i += 1
            self.C[i + 1].insertNonFull(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
    def splitChild(self, i, y):
        z = BTreeNode(y.t, y.leaf)
        z.n = self.t - 1
        for j in range(self.t - 1):
            z.keys[j] = y.keys[j + self.t]
        if not y.leaf:
            for j in range(self.t):
                z.C[j] = y.C[j + self.t]
        y.n = self.t - 1
        for j in range(self.n, i, -1):
            self.C[j + 1] = self.C[j]
        self.C[i + 1] = z
        for j in range(self.n - 1, i - 1, -1):
            self.keys[j + 1] = self.keys[j]
        self.keys[i] = y.keys[self.t - 1]
        self.n += 1

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

```
    def traverse(self):
        for i in range(self.n):
            if not self.leaf:
                self.C[i].traverse()
            print(self.keys[i], end=' ')
        if not self.leaf:
            self.C[i + 1].traverse()


    # A function to search a key in the subtree rooted with this
node.
    def search(self, k):
        i = 0
        while i < self.n and k > self.keys[i]:
            i += 1
        if i < self.n and k == self.keys[i]:
            return self
        if self.leaf:
            return None
        return self.C[i].search(k)

# A BTree
class BTree:
    def __init__(self, t):
        self.root = None # Pointer to root node
        self.t = t  # Minimum degree

    # function to traverse the tree
    def traverse(self):
        if self.root != None:
            self.root.traverse()

    # function to search a key in this tree
    def search(self, k):
        return None if self.root == None else self.root.search(k)

    # The main function that inserts a new key in this B-Tree
    def insert(self, k):
        if self.root == None:
            self.root = BTreeNode(self.t, True)
            self.root.keys[0] = k # Insert key
            self.root.n = 1
        else:
            if self.root.n == 2 * self.t - 1:
                s = BTreeNode(self.t, False)
                s.C[0] = self.root
                s.splitChild(0, self.root)
                i = 0
```

```python
                if s.keys[0] < k:
                    i += 1
                s.C[i].insertNonFull(k)
                self.root = s
            else:
                self.root.insertNonFull(k)

# Driver program to test above functions
if __name__ == '__main__':
    t = BTree(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)

    print("Traversal of the constructed tree is ", end = ' ')
    t.traverse()
    print()

    k = 6
    if t.search(k) != None:
        print("Present")
    else:
        print("Not Present")

    k = 15
    if t.search(k) != None:
        print("Present")
    else:
        print("Not Present")
```

```cpp
#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();
}
```

```cpp
// 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;
}
```

```cpp
// 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;
}
```