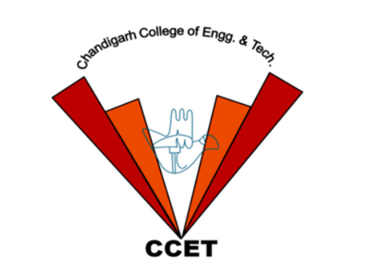
**CHANDIGARH COLLEGE OF ENGINEERING & TECHNOLOGY (DEGREE WING)**

****

Government institute under Chandigarh (UT) Administration, affiliated to Punjab University, Chandigarh

Department of Computer Science & Engineering

**Semester**: CSE 3rd

**SUBJECT:** Data Structures Practical (CS351)

**Problem 9: Case Study of Binary Tree Variants**

**Submitted by:                                                Submitted to:**

Bhavyam Dhand                                              Dr. R.B. Patel

(CO23316)                 (Professor)

**CODE**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

#define MAX\_KEYS 4

#define MIN\_KEYS 2

//AVL DATA STRUCTURE

//Representation of AVL

struct AVL

{

    int key;

    AVL\*left;

    AVL\*right;

    int height;

};

//Creating AVL node

AVL\* createAVLNode(int data) {

    AVL\* node = new AVL();

    node->key = data;

    node->left = nullptr;

    node->right = nullptr;

    node->height = 1;

    return node;

}

//Finding Height of node

int height(AVL\* node) {

    return node ? node->height : 0;

}

// Get balance factor of node

int getBalance(AVL\* &node) {

    return node ? height(node->left) - height(node->right) : 0;

}

//Rotate Leftwise

AVL\* L\_Rotate(AVL\* &x)

{

    AVL\* y = x->right;

    AVL\* T2 = y->left;

    y->left = x;

    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;

    y->height = max(height(y->left), height(y->right)) + 1;

    return y;

}

// Rotate Rightwise

AVL\* R\_Rotate(AVL\* &y)

{

    AVL\* x = y->left;

    AVL\* T2 = x->right;

    x->right = y;

    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;

    x->height = max(height(x->left), height(x->right)) + 1;

    return x;

}

// Insert a node into the AVL Tree

AVL\* insertAVL(AVL\* node,int key)

{

    if(!node)   return(createAVLNode(key));//Empty Tree

    else if (key<node->key) node->left=insertAVL(node->left,key);

    else if(key>node->key)  node->right=insertAVL(node->right,key);

    else return node;

    node->height=1+max(height(node->left),height(node->right));

    int balance=getBalance(node);

    //4 CASES

    //case 1: Right Rotation

    if (balance>1&&key<node->left->key) return(R\_Rotate(node));

    //Case 2: Left Rotation

    if(balance<-1&&key>node->right->key) return(L\_Rotate(node));

    //Case 3: Right & Left Rotation

    if(balance>1&&key>node->left->key)

    {

        node->left=L\_Rotate(node->left);

        return R\_Rotate(node);

    }

    //Case 4: Left & Right Rotation

    if(balance<-1&&key<node->right->key)

    {

        node->right=R\_Rotate(node->right);

        return L\_Rotate(node);

    }

    return node;

}

//Find the Minimum Value Node

AVL\*MinValueNode(AVL\*node)

{

    AVL\* current = node;

    // Traverse the left subtree to find the minimum value node

    while (current && current->left != nullptr)

        current = current->left;

    return current;

}

AVL\* DeleteAVL(AVL\* root, int key)

{

    if (!root) return root;

    if (key < root->key)

        root->left = DeleteAVL(root->left, key);

    else if (key > root->key)

        root->right = DeleteAVL(root->right, key);

    else

    {

        // node with only one child or no child

        if (!root->left || !root->right)

        {

            AVL\* temp = root->left ? root->left : root->right;

            if (!temp)  // No child

            {

                temp = root;

                root = nullptr;

            }

            else  // One child

                \*root = \*temp;

            delete temp;

        }

        else

        {

            // node with two children: Get the inorder successor

            AVL\* temp = MinValueNode(root->right);

            root->key = temp->key;

            root->right = DeleteAVL(root->right, temp->key);

        }

    }

    if (!root) return root;

    root->height = 1 + max(height(root->left), height(root->right));

    int balance = getBalance(root);

    // Left Left Case

    if (balance > 1 && getBalance(root->left) >= 0)

        return R\_Rotate(root);

    // Left Right Case

    if (balance > 1 && getBalance(root->left) < 0)

    {

        root->left = L\_Rotate(root->left);

        return R\_Rotate(root);

    }

    // Right Right Case

    if (balance < -1 && getBalance(root->right) <= 0)

        return L\_Rotate(root);

    // Right Left Case

    if (balance < -1 && getBalance(root->right) > 0)

    {

        root->right = R\_Rotate(root->right);

        return L\_Rotate(root);

    }

    return root;

}

AVL\* AVLsearch(AVL\* root, int key)

{

    // Base Cases: root is null or key is

    // present at root

    if (root == NULL || root->key == key)

        return root;

    // Key is greater than root's key

    if (root->key < key)

        return AVLsearch(root->right, key);

    // Key is smaller than root's key

    return AVLsearch(root->left, key);

}

void preOrder(AVL \*root)

{

    if(root != NULL)

    {

        printf("%d ", root->key);

        preOrder(root->left);

        preOrder(root->right);

    }

}

//BVL DATA STRUCTURE

struct BTreeNode {

    int keys[MAX\_KEYS];           // Array of keys

    BTreeNode\* children[MAX\_KEYS + 1]; // Array of children pointers

    int numKeys;                  // Number of keys in the node

    bool isLeaf;                  // True if the node is a leaf

    BTreeNode(bool leaf) : isLeaf(leaf), numKeys(0) {

        for (int i = 0; i < MAX\_KEYS + 1; i++) {

            children[i] = nullptr;

        }

    }

};

void splitChild(BTreeNode\* parent, int i, BTreeNode\* fullChild) {

    BTreeNode \*newNode=new BTreeNode(fullChild->isLeaf);

    newNode->numKeys = MIN\_KEYS;

    // Move the last MIN\_KEYS keys of fullChild to newNode

    for (int j = 0; j < MIN\_KEYS; j++)

        newNode->keys[j] = fullChild->keys[j + MIN\_KEYS + 1];

    // Move the last MIN\_KEYS + 1 children of fullChild to newNode

    if (!fullChild->isLeaf) {

        for (int j = 0; j < MIN\_KEYS + 1; j++)

            newNode->children[j] = fullChild->children[j + MIN\_KEYS + 1];

    }

    fullChild->numKeys = MIN\_KEYS;

    // Shift parent's children to make room for newNode

    for (int j = parent->numKeys; j >= i + 1; j--)

        parent->children[j + 1] = parent->children[j];

    parent->children[i + 1] = newNode;

    // Move the middle key of fullChild to the parent

    for (int j = parent->numKeys - 1; j >= i; j--)

        parent->keys[j + 1] = parent->keys[j];

    parent->keys[i] = fullChild->keys[MIN\_KEYS];

    parent->numKeys++;

}

// Recursive function to insert a new key

void insertNonFull(BTreeNode\* node, int key) {

    int i = node->numKeys - 1;

    if (node->isLeaf) {

        while (i >= 0 && key < node->keys[i]) {

            node->keys[i + 1] = node->keys[i];

            i--;

        }

        node->keys[i + 1] = key;

        node->numKeys++;

    } else {

        while (i >= 0 && key < node->keys[i])

            i--;

        i++;

        if (node->children[i]->numKeys == MAX\_KEYS) {

            splitChild(node, i, node->children[i]);

            if (key > node->keys[i])

                i++;

        }

        insertNonFull(node->children[i], key);

    }

}

// Function to insert a key into the B-Tree and return the root node

BTreeNode\* insertBTree(BTreeNode\* root, int key) {

    if (root->numKeys == MAX\_KEYS) {

        BTreeNode\* newRoot = new BTreeNode(false);

        newRoot->children[0] = root;

        splitChild(newRoot, 0, root);

        int i = (newRoot->keys[0] < key) ? 1 : 0;

        insertNonFull(newRoot->children[i], key);

        return newRoot;

    } else {

        insertNonFull(root, key);

        return root;

    }

}

BTreeNode \*BTreeSearch(BTreeNode\* root,int k)

{

    int i=0;

    while (i<root->numKeys&&k>root->keys[i])    i++;

    if (root->keys[i]==k)   return root;

    if(root->isLeaf)    return NULL;

    return BTreeSearch(root->children[i],k);

}

void remove(BTreeNode\* root, int key) {

    if (!root) {

        cout << "Tree is empty." << endl;

        return;

    }

    // Handle deletion for non-root node and root node

    if (root->isLeaf) {

        // Handle key deletion for leaf nodes (simple case)

        for (int i = 0; i < root->numKeys; i++) {

            if (root->keys[i] == key) {

                // Shift all keys after the deleted key

                for (int j = i; j < root->numKeys - 1; j++) {

                    root->keys[j] = root->keys[j + 1];

                }

                root->numKeys--;

                return;

            }

        }

    } else {

        // Recursive deletion logic for internal nodes

        // You would need to handle cases like merging or redistributing nodes here

    }

}

// Function to traverse and print the tree

void inorderTraversal(BTreeNode\* node) {

    if (node != nullptr) {

        // Traverse all children and print keys between them

        for (int i = 0; i < node->numKeys; i++) {

            // Recur for the left child

            if (!node->isLeaf)

                inorderTraversal(node->children[i]);

            // Print the current key

            cout << node->keys[i] << " ";

        }

        // Visit the rightmost child

        if (!node->isLeaf)

            inorderTraversal(node->children[node->numKeys]);

    }

}

int main() {

    vector<int> Data = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

    AVL\* Aroot = NULL;

    BTreeNode\* Broot = new BTreeNode(true);

    for (int value : Data) {

        Aroot = insertAVL(Aroot, value);

        Broot = insertBTree(Broot, value);

    }

    int choice, value;

    // Menu loop

    while (true) {

        cout << "\nMenu:\n";

        cout << "1. Insert into AVL Tree\n";

        cout << "2. Insert into B-Tree\n";

        cout << "3. Delete from AVL Tree\n";

        cout << "4. Delete from B-Tree\n";

        cout << "5. Display AVL Tree\n";

        cout << "6. Display B-Tree\n";

        cout << "7. Search into AVL Tree\n";

        cout << "8. Search into B-Tree\n";

        cout << "9. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                cout << "Enter value to insert into AVL Tree: ";

                cin >> value;

                Aroot = insertAVL(Aroot, value);

                cout << "Inserted " << value << " into AVL Tree." << endl;

                break;

            case 2:

                cout << "Enter value to insert into B-Tree: ";

                cin >> value;

                Broot = insertBTree(Broot, value);

                cout << "Inserted " << value << " into B-Tree." << endl;

                break;

            case 3:

                cout << "Enter value to be deleted from AVL Tree: ";cin>>value;

                DeleteAVL(Aroot,value);

                cout << "Deleted " << value << " from AVL Tree." << endl;

                break;

            case 4:

                cout << "Enter value to be deleted from B Tree: ";

                remove(Broot,value);

                cout << "Deleted " << value << " from B Tree." << endl;

                break;

            case 5:

                cout << "AVL Tree: ";

                preOrder(Aroot);

                cout << endl;

                break;

            case 6:

                cout << "B Tree: ";

                inorderTraversal(Broot);

                cout << endl;

                break;

            case 7:

                cout<<"Enter Value to be searched in AVL: "; cin>>value;

                if(AVLsearch(Aroot,value)!=NULL) cout<<"Found "<<value<<" in AVL Tree at Level: "<<height(AVLsearch(Aroot,value));

                break;

            case 8:

                cout<<"Enter Value to be searched in B Tree: "; cin>>value;

                if (BTreeSearch(Broot,value)!=NULL) cout<<"Found "<<value<<" in B Tree.";

                break;

            case 9:

                cout << "Exiting program." << endl;

                return 0;

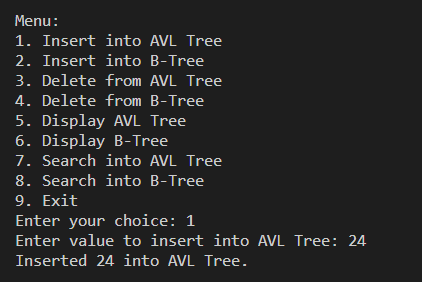
        }

    }

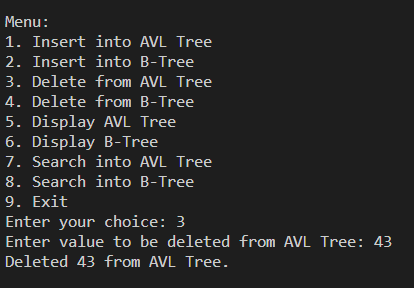
}

**CODE OUTPUT**

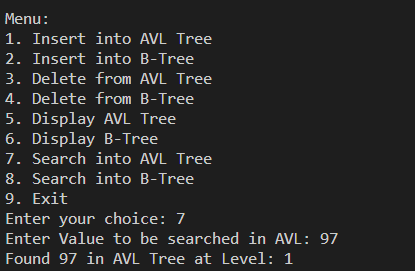
1. **AVL Tree**
2. **Insert**

****

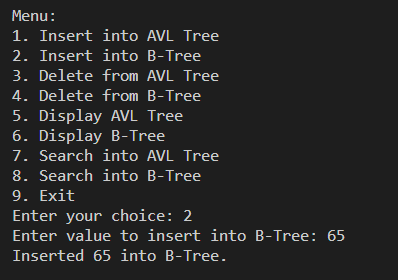
1. **Delete**

****

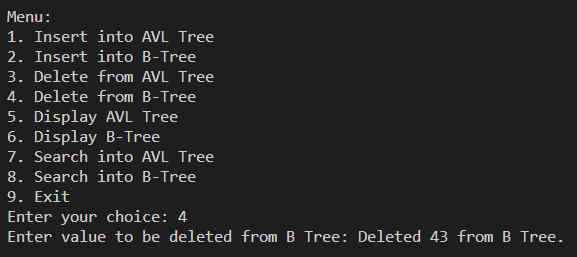
1. **Search**

****

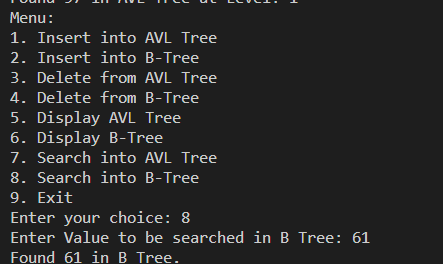
1. **B-Tree**
   1. **Insert**

****

* 1. **Delete**

****

* 1. **Search**

****