S.No: 1	Exp. Name: ***Project Module***	**Date: 2024-06-13**

Aim:

Project Module

Source Code:

hello.c

```
#include <stdio.h>
#include <stdlib.h>

// Node structure for the AVL tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
};

// Function to get height of a node
int height(struct Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

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

// Function to create a new node with given data
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially added at leaf
    return node;
}

// Function to right rotate subtree rooted with y
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct 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;
}

// Function to left rotate subtree rooted with x
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct 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(struct Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to insert a new node with given key in AVL tree
struct Node* insert(struct Node* node, int data) {
    // Perform the normal BST insertion
    if (node == NULL)
        return newNode(data);

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

    // Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // Get the balance factor of this ancestor node
    int balance = getBalance(node);
```

```
    // If this node becomes unbalanced, then there are four cases

    // Left Left Case
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);

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

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

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

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

// Function to print inorder traversal of the tree
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
```

```
        printf("%d ", root->data);
    }
}
void displayTree(struct Node* root, int space) {
    int COUNT = 10;
    if (root == NULL)
        return;

    space += COUNT;

    displayTree(root->right, space);

    printf("\n");
    for (int i = COUNT; i < space; i++)
        printf(" ");
    printf("%d\n", root->data);

    displayTree(root->left, space);
}

struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;

    // Loop to find the leftmost leaf
    while (current->left != NULL)
        current = current->left;

    return current;
}
struct Node* deleteNode(struct Node* root, int data) {
    // Perform standard BST delete
    if (root == NULL)
        return root;

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

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

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

```
    else {
        // Node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node* temp = root->left ? root->left : root->right;

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

            free(temp);
        } else {
            // Node with two children: Get the inorder successor
            // (smallest in the right subtree)
            struct Node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->data = temp->data;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->data);
        }
    }

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

    // Update height of the current node
    root->height = 1 + max(height(root->left), height(root->right));

    // Get the balance factor of this node (to check whether this node
became unbalanced)
    int balance = getBalance(root);

    // 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;
}
// Driver program to test above functions
int main() {
    struct Node* root = NULL;
    int data;
    char choice;

    // Prompt the user to enter values until they choose to stop
    do {
        printf("Enter a value to insert into the AVL tree: ");
        scanf("%d", &data);
        root = insert(root, data);

        printf("Do you want to insert another value? (y/n): ");
        scanf(" %c", &choice);
    } while (choice == 'y' || choice == 'Y');


 printf("\nTree structure:\n");
    displayTree(root, 0);
printf("\nInorder traversal of the constructed AVL tree is: \n");
    inorder(root);
printf("\npreorder traversal of the constructed AVL tree is: \n");

preorder(root);
printf("\npostorder traversal of the constructed AVL tree is: \n");
postorder(root);
printf("\nEnter a deletion value from AVL tree:");
scanf("%d",&data);
root = deleteNode(root, data);
 printf("\nTree structure:\n");
```

```
    displayTree(root, 0);
    return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
Hello World

ID: 2303811710421107
2023-2027-L
K.Ramakrishnan College of Technology