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

// An AVL tree node
struct Node
{
	int key;
	struct Node *left;
	struct Node *right;
	int height;
};

// A utility function to get the height of the tree
int height(struct 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;
}
```

```
/* Helper function that allocates a new node with the given key and
        NULL left and right pointers. */
struct Node* newNode(int key)
{
        struct Node* node = (struct Node*)
                                                malloc(sizeof(struct Node));
        node->key = key;
        node->left = NULL;
        node->right = NULL;
        node->height = 1; // new node is initially added at leaf
        return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
        int key;
        struct Node *left;
        struct Node *right;
```

```
        int height;
};

// A utility function to get the height of the tree
int height(struct 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;
}

/* Helper function that allocates a new node with the given key and
        NULL left and right pointers. */
struct Node* newNode(int key)
{
        struct Node* node = (struct Node*)
                                        malloc(sizeof(struct Node));
        node->key = key;
        node->left = NULL;
```

```
        node->right =  given above.
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;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
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 *N)
{
        if (N == NULL)
                return 0;
        return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
```

```
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key)
{
	/* 1. Perform the normal BST insertion */
	if (node == NULL)
		return(newNode(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 are not allowed in BST
		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
void preOrder(struct Node *root)
{
        if(root != NULL)
        {
                printf("%d ", root->key);
                preOrder(root->left);
                preOrder(root->right);
        }
}

/* Driver program to test above function*/
int main()
{
struct Node *root = NULL;

/* Constructing tree given in the above figure */
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
ro
```

```
/* The constructed AVL Tree would be
                        30
                    / \
                    20 40
                    / \        \
            10 25 50
*/

printf("Preorder traversal of the constructed AVL"
                    " tree is \n");
preOrder(root);

return 0;
}
ot = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);

/* The constructed AVL Tree would be
                        30
                    / \
                    20 40
                    / \        \
            10 25 50
*/
```

```
printf("Preorder traversal of the constructed AVL"
                        " tree is \n");
preOrder(root);


return 0;
}
```