```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
// A utility function to get maximum of two integers
//int max(int,int);
// An AVL tree node
struct node
{
   int key;
   struct node *left;
   struct node *right;
   int height;
};


// A utility function to get 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 max1(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 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 = max1(height(y->left), height(y->right))+1;
    x->height = max1(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 = max1(height(x->left), height(x->right))+1;
    y->height = max1(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);
}

struct node* insert(struct node* node, int key)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

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

    /* 2. Update height of this ancestor node */
```

```
node->height = max1(height(node->left), height(node->right)) + 1;

/* 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;
}

/* 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. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

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

    return current;
}

struct node* deleteNode(struct 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) )
   {
      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->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 = max1(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);

// 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(struct node *root)
{
    if(root != NULL)
    {
            printf("%d ", root->key);
            preOrder(root->left);
            preOrder(root->right);
    }
}


/* Drier program to test above function*/
```

```
int main()
{
  struct node *root = NULL;
  //clrscr();

int ch,n;
//clrscr();
printf("1.Insert \n 2.Preorder\n 3.delenode\n 4:Exit");
do
{
printf("\nEnter your choice: ");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\n enter the data you want to insert");
scanf("%d",&n);
root = insert(root, n);
break;

case 2:preOrder(root);break;
case 3:
printf("\n enter the data you want to delete");
scanf("%d",&n);
root = deleteNode(root, n);break;
```

```
case 4:printf("Program exited");break;

default:printf("Invalid choice");

}

}while(ch!=4);

}
```