```
#include <stdio.h>
#include <malloc.h>
#include "bst.h"

struct _Node
{                                              //type def that defines the
attributes of a node
    int value;
    Node *left;
    Node *right;
};

Node *deleteRootDown(Node *root, int value)         // this deletes a subtree by
deleting its root, then the tree is
{                                                    // rotated so that the tree
still root is replaced by a lower
    while(root != NULL)                              // node and then this process
is repeated until the root is null
    {
        root = deleteNode(root, root->value);        //           8
6            4           10
    }                                                //        6      10 -->     4
10 -->       10  -->     -->
    return root;                                     //     4
}

typedef struct
{
    Node* currentNode;                               // so that multiple values
can be returned by a function
    Node* previousNode;
    char side;
}loopReturn;

Node* insertNode (Node * root, int value)
{
    Node* currentNode = root;                            // while loop searches
for the correct position to insert the
    Node* previousNode = NULL;                           // node
    char side = 'a';                                     // side shows that the
current node is the left or right child
    while(currentNode != NULL)                           // of the previous node.
If the side = zero then it means the
    {                                                    // node to be deleted is
a root
        if(value < currentNode -> value)
        {
            previousNode = currentNode;
            side = 'l';
            currentNode = (currentNode -> left);
        }
        else if(value > currentNode -> value)
        {
            previousNode = currentNode;
            side = 'r';
            currentNode = (currentNode -> right);        // if the node already
exists in the tree then the function
        }                                                // returns null
        else
```

```
        {
            return NULL;
        }
    }

    Node* newNode = (Node*) malloc (sizeof (Node));    //allocates memory for the
new node

    if (newNode == NULL)
    {
        printf("There is not enough memory");
        return NULL;
    }
    else if (side != 'a')                              // if the node is node a
root then the previous node is updated
    {                                                  // to have the new node as
its child
        if (side == 'l')
        {
            previousNode->left = newNode;
        }
        else
        {
            previousNode->right = newNode;
        }
    }


    newNode -> value = value;                   // assign the value to the new node
    newNode -> left = NULL;
    newNode -> right = NULL;



    return newNode;                             // returns the new node
}

loopReturn *findRoot(Node *currentNode, int value) {                         //
this functions locates a node and
    loopReturn* result = (loopReturn *) malloc (sizeof (loopReturn));        //
returns the previous node; whether
    result->side = 'a';                                  // the child is the
left or right child of the previous
    result->previousNode = NULL;                         // or if the node
is  a root; and returns the node
    Node* root = currentNode;
    result->currentNode = root;                          // if the node
doesn't exist the side is set to n for
    int located = 0;                                     // null
    while (located == 0)
    {
        if (result->currentNode->value == value)
        {
            located = 1;
        }
        else if(result->currentNode->value > value)
        {
            if(result->currentNode->left == NULL)
            {
```

```
                result->currentNode = root;
                result->side = 'n';
                return result;
            }
            else
            {
                result->previousNode = result->currentNode;
                result->currentNode = result->currentNode->left;
                result->side = 'l';
            }

        }
        else if (result->currentNode->value < value)
        {
            if(result->currentNode->right == NULL)
            {
                result->currentNode = root;
                result->side = 'n';
                return result;
            }
            else
            {
                result->previousNode = result->currentNode;
                result->currentNode = result->currentNode->right;
                result->side = 'r';
            }
        }
    }
    return result;
}

Node * deleteNode(Node * root, int value)
{
    Node* permRoot = root;
    Node* currentNode = root;
    loopReturn* lr = findRoot(root, value);       // lr hold the found node, the
previous node and whether or not the
                                                  // found node is the left or
right child of the previous node or the
    if(lr->side == 'n') // node doesn't exist     // root
    {
        return lr->currentNode;
    }
    else if(lr->side == 'a') // node is root
    {
        //this else if statement carries out the necessary rotations
        if(lr->currentNode->left == NULL && lr->currentNode->right == NULL) // if
both the children of the root are null
        {
            free(lr->currentNode);
            return NULL;
        }
        else if(lr->currentNode->left == NULL) // if the left child of the root is
null
        {
            Node* newRoot = currentNode->right;
            free(lr->currentNode);
            return newRoot;
        }
```

```
        else if (lr->currentNode->right ==NULL) //if the right child of the root is
null
        {
            Node* newRoot = lr->currentNode->left;
            free(lr->currentNode);
            return newRoot;
        }
        else // if both of the children are present
        {
            Node* leftChild = lr->currentNode->left;
            Node* newRoot = NULL;
            if(leftChild->left== NULL && leftChild->right == NULL) // if both the
children of the left child of the
            {                                                       // root are
present
                leftChild->right = lr->currentNode->right;
                newRoot = leftChild;
            }
            else if(leftChild->right == NULL)                      // if only the
left child of the left child of the
            {                                                      // root is
present
                leftChild->right = lr->currentNode->right;
                newRoot = leftChild;
            }
            else
            {
                newRoot = leftChild->right;                        // if only the
right child of the left child of the
                newRoot->left = leftChild;                         // root is
present
                newRoot->right = lr->currentNode->right;
                leftChild->right = NULL;
            }
            free(lr->currentNode);
            return newRoot;
        }
    }
    else if(lr->currentNode->right == NULL && lr->currentNode->left == NULL) // the
node is not a root
    {                                                                         // if
the right and left children of the
        free(lr->currentNode);                                                //
node are NULL
        if(lr->side == 'l')
        {                                                                     //if
the current node is the left child
            lr->previousNode->left = NULL;                                    // of
the previous node
        }
        else
        {
            lr->previousNode->right = NULL;                                   // if
the current node is the right child
        }                                                                     // of
the previous node
    }
    else if (lr->currentNode->left != NULL && lr->currentNode->right !=NULL) // if
the left and right children of the
```

```
    {                                                                           //
node is null
        if(lr->side == 'l')
        {                                                                       // if
the node to be deleted is the left
            lr->previousNode->left = lr->currentNode->left;                     //
child of the previous node
            Node* newCurrentNode = lr->previousNode->left;
            newCurrentNode->right = lr->currentNode->right;
            free(lr->currentNode);
        }
        else
        {
            lr->previousNode->right = lr->currentNode->right;                  // if
the node to be deleted is the right
            Node* newCurrentNode = lr->previousNode->right;                    //
child of the previous node
            newCurrentNode->left = lr->currentNode->left;
            free(lr->currentNode);
        }
    }
    else if (lr->currentNode->left != NULL)                                    // if
only the left child of the node to be
    {                                                                          //
deleted is present
        if (lr->side == 'l')
        {                                                                      // if
the node to de deleted is the left
            lr->previousNode->left = lr->currentNode->left;                    // child
of the previous node
            free(lr->currentNode);
        }
        else
        {
            lr->previousNode->right = lr->currentNode->left;                   // if
the node to be deleted is the right
            free(lr->currentNode);                                             // child
of the previous node
        }

    }
    else if (lr->currentNode->right != NULL)                                   // if
only the right child of the node to be
    {                                                                          //
deleted is present
        if (lr->side == 'r')
        {
            lr->previousNode->right = lr->currentNode->right;                 // if the
node to be deleted is the right
            free(lr->currentNode);                                            // child
of the previous node
        }
        else
        {
            lr->previousNode->left = lr->currentNode->right;                  // if the
node to be deleted is the left child
            free(lr->currentNode);                                            // of the
node tp be deleted
        }
```

```
    }
    return permRoot;                                                       // return
the root
}


void printSubtree(Node * N)
{
    if (N == NULL)             //recursively prints the subtree
    {
        return;
    }
    else
    {
        printSubtree(N -> left);
        printf("%d\n", N -> value);
        printSubtree(N -> right);
    }

}


int countLeaves(Node * N)
{
    if (N->left == NULL && N->right == NULL)    // recursively counts the leaves
    {                                           // if the node is a leaf return 1
        return 1;
    }                                           // is the left or right subtree is
null return only the number of leaves
    else if(N->left == NULL)                    // from the not null subtree
    {
        return countLeaves(N->right);
    }
    else if(N->right == NULL)
    {
        return countLeaves(N->left);
    }
    else
    {
        return countLeaves(N->left) + countLeaves(N->right);    // returns the sum
of the leaves of the left and right
    }                                                           // subtree

}

Node * deleteSubtree(Node * root, int value)
{

    loopReturn* lr = NULL;
    lr = findRoot(root, value); // findRoot returns the desired node as well as the
necessary data about it

    if (lr->side == 'r')    // if the root of the subtree to be deleted is the
right child of the previous node
    {
        lr->previousNode->right = deleteRootDown(lr->currentNode, lr->currentNode-
>value);
    }                       // deletes the whole right subtree of the previous node
from that root using deleteRootDown
```

```
    else if (lr->side == 'l')
    {                       // deletes the whole left subtree of the previous node
from that root using deleteRootDown
        lr->previousNode->left = deleteRootDown(lr->currentNode, lr->currentNode-
>value);
    }
    else if (lr->side == 'a')    // if the subtree to the deleted is the root of
the entire tree the whole tree is
    {                            // deleted using deleteRootDown and then NULL is
returned
        Node* temp = deleteRootDown(lr->currentNode, lr->currentNode->value);
        return NULL;
    }
    return root;     // returns the new root

}

int depth (Node * R, Node * N)
{
    int depth = 0;

    Node* currentNode = R;

    int located = 0;
    while (located == 0)
    {
        if (currentNode->value == N->value) // if the current node is the node we
are looking for return the depth
        {
            return depth;
        }
        else if(currentNode->value > N->value)     // if the node we are looking
for greater the value we now look left
        {
            if(currentNode->left == NULL)           // if the left node of the
current node is null then the node we
            {                                       // are looking for doesn't
exist
                return -1;
            }
            else
            {
                currentNode = currentNode->left;    // if not then we go down a
level in the tree and incrementing depth
                depth ++;
            }

        }
        else if (currentNode->value < N->value)     // exact same as above but this
time we look right
        {
            if(currentNode->right == NULL)
            {
                return -1;
            }
            else
            {
                currentNode = currentNode->right;
                depth++;
```

```
                }
            }
        }
    }
```