```c
//Implement a Binary Search Tree(create, insert, delete, number of nodes, height,
traversal, copy)
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int key;
    node *left, *right, *parent;
};

node *get_new_node()
{
    node *p;
    p = (node *)malloc(sizeof(node));
    return p;
}

int max(int a,int b)
{
    if (a>b)
        return a;
    else
        return b;
}

void tree_insert(node **bst, int val)
{
    node *p = *bst, *q;
    while(p != NULL)
    {
        q = p;
        if(p->key < val)
            p = p->right;
        else
            p = p->left;
    }
    p = get_new_node();
    p->key = val;
    p->left = NULL;
    p->right = NULL;
    p->parent = q;
    if (q == NULL)
    {
        *bst = p;
    }
    else
    {
        if(q->key < val)
            q->right = p;
        else
            q->left = p;
    }
}

node * tree_minimum(node *bst)
{
    node *p = bst;
    while(p->left !=NULL)
    {
        p = p->left;
    }
    return p;
}

node * tree_successor(node *x)
{
    if (x->right != NULL)
        return tree_minimum(x->right);
```

```
    node *y=x->parent;
    while(y!= NULL && x == y->right)
    {
        x = y;
        y = y->parent;
    }
    return y;
}

node * search(node *bst, int key)
{
    node *p = bst;
    while(p!=NULL)
    {
        if(p->key > key)
            p = p->left;
        else if(p->key < key)
            p = p->right;
        else
            return p;
    }
}

node *copy(node *bst)
{
    if(bst == NULL)
        return NULL;
    node *bst_copy = get_new_node();
    bst_copy->key = bst->key;
    bst_copy->left = copy(bst->left);
    bst_copy->right = copy(bst->right);
    return bst_copy;
}

void tree_delete(node **bst, node *z)
{
    node *y;
    if (z->left == NULL || z->right == NULL)
    {
        if(z ->left == NULL && z->right == NULL)
            y = NULL;
        else if(z->left != NULL)
            y = z->left;
        else if(z->right != NULL)
            y = z->right;
        if (z->parent->left == z)
        {
            z->parent->left = y;
        }
        else
        {
            z->parent->right = y;

        }
        free(z);
    }
    else
    {
        y = tree_successor(z);
        z->key = y->key;
        tree_delete(bst, y);
    }

}

void inorder(node *bst)
{
    if(bst != NULL)
    {
```

```
        inorder(bst->left);
        printf("%6d", bst->key);
        inorder(bst->right);
    }
}

void preorder(node *bst)
{
    if(bst != NULL)
    {
        printf("%6d", bst->key);
        preorder(bst->left);
        preorder(bst->right);
    }
}

int count_nodes(node *bst)
{
    if (bst == NULL)    return 0;
    return 1 + count_nodes(bst->left) + count_nodes(bst->right);
}

int height(node *bst)
{
    if(bst == NULL) return 0;
    return 1 + max(height(bst->left), height(bst->right));
}

int main()
{
    node *bst;
    bst = NULL;
    tree_insert(&bst, 5);
    tree_insert(&bst, 3);
    tree_insert(&bst, 2);
    tree_insert(&bst, 7);
    tree_insert(&bst, 8);
    printf("Original Preorder\n");
    preorder(bst);
    printf("\n");
    tree_delete(&bst, search(bst,5));
    printf("After Deletion Preorder\n");
    preorder(bst);
    printf("\n");
    printf("Total Nodes = %d\n", count_nodes(bst));
    printf("Height = %d\n", height(bst));

}

/*OUTPUT
Original Preorder
     5     3     2     7     8
After Deletion Preorder
     7     3     2     8
Total Nodes = 4
Height = 3
*/
```