```c
#include<stdio.h>
#include<stdlib.h>
#define TRUE 1
#define FALSE 0
#define ElemType int
#define  KeyType int
/* 二叉排序树的节点结构定义 */
typedef struct BiTNode
{
    int data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;



//二叉排序树查找算法
int SearchBST(BiTree T, KeyType key, BiTree f, BiTree *p) {
    //如果 T 指针为空，说明查找失败，令 p 指针指向查找过程中最后一个叶子结点，并返回查找失败的信息
    if (!T) {
        *p = f;
        return FALSE;
    }
    //如果相等，令 p 指针指向该关键字，并返回查找成功信息
    else if (key == T->data) {
        *p = T;
        return TRUE;
    }
    //如果 key 值比 T 根结点的值小，则查找其左子树；反之，查找其右子树
    else if (key < T->data) {
        return SearchBST(T->lchild, key, T, p);
    }
    else {
        return SearchBST(T->rchild, key, T, p);
    }
}




//插入节点
int InsertBST(BiTree *T, ElemType e) {
    BiTree p = NULL;
    //如果查找不成功，需做插入操作
    if (!SearchBST(*T, e, NULL, &p)) {
        //初始化插入结点
        BiTree s = (BiTree)malloc(sizeof(BiTNode));
        s->data = e;
        s->lchild = s->rchild = NULL;
        //如果 p 为NULL，说明该二叉排序树为空树，此时插入的结点为整棵树的根结点
        if (!p) {
            *T = s;
        }
        //如果 p 不为 NULL，则 p 指向的为查找失败的最后一个叶子结点，只需要通过比较 p 和 e 的值确定 s 到底是 p 的左孩子还是右孩子
        else if (e < p->data) {
            p->lchild = s;
        }
        else {
            p->rchild = s;
```

```
        }
        return TRUE;
    }
    //如果查找成功，不需要做插入操作，插入失败
    return FALSE;
}



//删除节点
int Delete(BiTree *p)
{
    BiTree q, s;
    //情况 1，结点 p 本身为叶子结点，直接删除即可
    if ((*p)->lchild==NULL && (*p)->rchild==NULL) {
        *p = NULL;
    }
    else if ((*p)->lchild==NULL) { //左子树为空，只需用结点 p 的右子树根结点代替结点 p 即可；
        q = *p;
        *p = (*p)->rchild;
        free(q);
    }
    else if ((*p)->rchild==NULL) {//右子树为空，只需用结点 p 的左子树根结点代替结点 p 即可；
        q = *p;
        *p = (*p)->lchild;//这里不是指针 *p 指向左子树，而是将左子树存储的结点的地址赋值给指针变量 p
        free(q);
    }
    else {//左右子树均不为空，采用第 2 种方式
        q = *p;
        s = (*p)->lchild;
        //遍历，找到结点 p 的直接前驱
        while (s->rchild)
        {
            q = s;
            s = s->rchild;
        }
        //直接改变结点 p 的值
        (*p)->data = s->data;
        //判断结点 p 的左子树 s 是否有右子树，分为两种情况讨论
        if (q != *p) {
            q->rchild = s->lchild;//若有，则在删除直接前驱结点的同时，令前驱的左孩子结点改为 q 指向结点的
孩子结点
        }
        else {
            q->lchild = s->lchild;//否则，直接将左子树上移即可
        }
        free(s);
    }
    return TRUE;
}



int DeleteBST(BiTree *T, int key)
{
    if (!(*T)) {//不存在关键字等于key的数据元素
        return FALSE;
    }
    else
```

```
    {
        if (key == (*T)->data) {
            Delete(T);
            return TRUE;
        }
        else if (key < (*T)->data) {
            //使用递归的方式
            return DeleteBST(&(*T)->lchild, key);
        }
        else {
            return DeleteBST(&(*T)->rchild, key);
        }
    }
}



void midOrder(BiTree t)
{
    if (t == NULL) {
        return;
    }
    midOrder(t->lchild);
    printf("%d ", t->data);
    midOrder(t->rchild);
}



int main()
{
    int i;
    int a[8] = { 3,4,2,5,9,12,8,21 };
    BiTree T = NULL;
    for (i = 0; i < 8; i++) {
        InsertBST(&T, a[i]);
    }
    printf("中序遍历二叉排序树：\n");
    midOrder(T);
    printf("\n");
    printf("请输入要删除的值：\n");
    int x;
    scanf("%d",&x);
    printf("删除%d后，中序遍历二叉排序树：\n",x);
    DeleteBST(&T,x);
    midOrder(T);
    printf("\n");
}
```