咸鱼注：大家可以在自己电脑上新建一个 C++ 项目，并把这段代码复制到 main 函数前边，然后在你的 main 函数里调用 test_BinarySearchTree() 这个函数。

提示：用电脑网页版看，体验更好一些

```
// 个人微博：@王道咸鱼学长-计算机考研
// 这段代码实现了二叉查找树BST 的 插入、删除、查找，前中后序遍历、层序遍历；求树的深度
// 同时，printBST 函数也可以利用层序遍历在你的终端打印二叉树的样子

#ifndef CODE_BINARYSEARCHTREE_H
#define CODE_BINARYSEARCHTREE_H

#include "stdio.h"
#include "stdlib.h"

//二叉排序树结点
typedef struct BSTNode{
    int key;                        //数据域
    struct BSTNode *lchild,*rchild;   //左、右孩子指针
}BSTNode,*BSTree;

//平衡二叉树结点
typedef struct AVLNode{
    int key;       //数据域
    int balance;    //平衡因子
    struct AVLNode *lchild,*rchild;
}AVLNode,*AVLTree;

//访问结点p
void visit(BSTNode * p){
    printf("%d,", p->key);
}

//先序遍历
void PreOrder(BSTree T){
    if(T!=NULL){
        visit(T);               //访问根结点
        PreOrder(T->lchild);       //递归遍历左子树
        PreOrder(T->rchild);       //递归遍历右子树
    }
}

//中序遍历
void InOrder(BSTree T){
    if(T!=NULL){
        InOrder(T->lchild);        //递归遍历左子树
        visit(T);            //访问根结点
        InOrder(T->rchild);        //递归遍历右子树
    }
}

//后序遍历
void PostOrder(BSTree T){
    if(T!=NULL){
        PostOrder(T->lchild);  //递归遍历左子树
        PostOrder(T->rchild);  //递归遍历右子树
        visit(T);            //访问根结点
    }
}

//求树的深度
int treeDepth(BSTree T){
    if (T == NULL) {
        return 0;
    }
    else {
        int l = treeDepth(T->lchild);
        int r = treeDepth(T->rchild);
        //树的深度=Max(左子树深度，右子树深度)+1
        return l>r ? l+1 : r+1;
    }
}

//在树T中找到结点p的父节点
BSTNode * findFather(BSTree T, BSTNode * p) {
    //检查T是否是p的父节点
    if (T==NULL)
        return NULL;
```

```
    if (T->lchild==p || T->rchild==p)
        return T;

    //在左子树找p的父节点
    BSTNode * l = findFather(T->lchild, p);
    if (l != NULL)
        return l;

    //在右子树找p的父节点
    BSTNode * r = findFather(T->rchild, p);
    if (r != NULL)
        return r;

    //左右子树中都没找到父节点。或者，根节点也没有父节点
    return NULL;
}

//在二叉排序树中查找值为 key 的结点（非递归实现）
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){    //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild;     //小于，则在左子树上查找
        else  T=T->rchild;               //大于，则在右子树上查找
    }
    return T;
}

//在二叉排序树中查找值为 key 的结点（递归实现）
BSTNode *BSTSearch(BSTree T,int key){
    if (T==NULL)
        return NULL;     //查找失败
    if (key==T->key)
        return T;        //查找成功
    else if (key < T->key)
        return BSTSearch(T->lchild, key);    //在左子树中找
    else
        return BSTSearch(T->rchild, key);    //在右子树中找
}


//在二叉排序树插入关键字为k的新结点（递归实现）
int BST_Insert(BSTree &T, int k){
    if(T==NULL){              //原树为空，新插入的结点为根结点
        T=(BSTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rchild=NULL;
        return  1;           //返回1，插入成功
    }
    else if(k==T->key)        //树中存在相同关键字的结点，插入失败
        return  0;
    else if(k<T->key)         //插入到T的左子树
        return  BST_Insert(T->lchild,k);
    else                     //插入到T的右子树
        return  BST_Insert(T->rchild,k);
}

//在二叉排序树 T 中删除结点 p（不考虑p为根节点的情况）
int BST_DeleteNode(BSTree &T, BSTNode * p){
    if (p==NULL)
        return 0;

    //先找到 p 的父节点 father
    BSTNode *f = findFather(T, p);
    //判断 p 是 f 的左孩子 还是 右孩子
    bool isLchild = false;      //isLchild == false 表示 p 是右孩子
    if (f->lchild == p)
        isLchild = true;        //isLchild == true 表示 p 是左孩子

    // p 是叶子，直接删除，同时要修改其父节点的指针
    if (p->lchild==NULL && p->rchild==NULL) {
        if (isLchild){          //p是f的左孩子
            f->lchild = NULL;   //父节点f的左指针指向NULL
        } else {                //p是f的右孩子
            f->rchild = NULL;   //父节点f的右指针指向NULL
        }
        free(p);                //释放结点 p
        return 1;               //删除成功，返回1
    }

    // p 只有左子树，则让左子树顶替p即可
    if (p->lchild!=NULL && p->rchild==NULL) {
        if (isLchild){              //p是f的左孩子
```

```
            f->lchild = p->lchild;  //父节点f的左指针指向p的左子树
        } else {                    //p是f的右孩子
            f->rchild = p->lchild;  //父节点f的右指针指向p的左子树
        }
        free(p);                //释放结点 p
        return 1;               //删除成功，返回1
    }

    // p 只有右子树，则让右子树顶替p即可
    if (p->lchild==NULL && p->rchild!=NULL) {
        if (isLchild){              //p是f的左孩子
            f->lchild = p->rchild;  //父节点f的左指针指向p的右子树
        } else {                    //p是f的右孩子
            f->rchild = p->rchild;  //父节点f的右指针指向p的右子树
        }
        free(p);                //释放结点 p
        return 1;               //删除成功，返回1
    }

    //上面几种条件都不满足，说明 p 既有左子树，也有右子树。此时，可让p的后继来顶替p，然后删除其后继结点
    BSTNode * q = p->rchild;
    //找到 p的右子树中，最左下角的一个结点，这个结点就是p的后继
    while(q->lchild!=NULL){
        q = q->lchild;
    }
    p->key = q->key;            //用后继结点的关键字顶替被删除节点
    BST_DeleteNode(T, q);       //转化为删除节点q的操作
    return 1;
}

//从二叉排序树 T 中删除关键字为 k 的结点
int BST_DeleteKey(BSTree &T, int k){
    BSTNode * p = BST_Search(T, k);
    if (p==NULL){
        printf("【删除失败】，不存在这个关键字 %d\n", k);
        return  0;          //返回0，删除失败
    }
    if (p==T){
        printf("【删除失败】，关键字 %d 是根节点。答应我，我们不删除根节点，好吗？\n（注：删除根节点会更麻烦一点，有能力的同学可以
自己动手实现）", k);
        return  0;          //返回0，删除失败
    }
    BST_DeleteNode(T, p);   //删除节点p

    printf("【删除成功】，关键字为 %d\n", k);
}

//按照 str[] 中的关键字序列建立二叉排序树
void Creat_BST(BSTree &T,int str[],int n){
    T=NULL;           //初始时T为空树
    int i=0;
    while(i<n){        //依次将每个关键字插入到二叉排序树中
        BST_Insert(T,str[i]);
        i++;
    }
}


//链式队列结点，用于辅助实现层序遍历
typedef struct LinkNode{
    BSTNode * data;
    struct LinkNode *next;
}LinkNode;

typedef struct{
    LinkNode *front,*rear; //队头队尾
}LinkQueue;

//初始化队列(带头结点)
void InitQueue(LinkQueue &Q){
    //初始时 front、rear 都指向头结点
    Q.front=Q.rear=(LinkNode*)malloc(sizeof(LinkNode));
    Q.front->next=NULL;
}

//判断队列是否为空（带头结点）
bool IsEmpty(LinkQueue Q){
    if(Q.front==Q.rear)
        return true;
    else
        return false;
```

```
}

//新元素入队（带头结点）
void EnQueue(LinkQueue &Q,BSTNode * x){
    LinkNode *s=(LinkNode *)malloc(sizeof(LinkNode));
    s->data=x;
    s->next=NULL;
    Q.rear->next=s;     //新结点插到表尾之后
    Q.rear=s;           //修改表尾指针
}

//队头元素出队（带头结点）
bool DeQueue(LinkQueue &Q,BSTNode * &x){
    if(Q.front==Q.rear)
        return false;            //空队
    LinkNode *p=Q.front->next;  //p指向此次出队的结点
    x=p->data;                   //用变量x返回队头元素
    Q.front->next=p->next;       //修改头结点的 next 指针
    if(Q.rear==p)                //此次是最后一个结点出队
        Q.rear=Q.front;          //修改 rear 指针
    free(p);                     //释放结点空间
    return true;
}

//层序遍历
void LevelOrder(BSTree T){
    LinkQueue Q;
    InitQueue(Q);               //初始化辅助队列
    BSTree p;
    EnQueue(Q,T);               //将根结点入队

    while(!IsEmpty(Q)){              //队列不空则循环
        DeQueue(Q, p);           //队头结点出队
        visit(p);                //访问队头元素
        if(p->lchild!=NULL)
            EnQueue(Q,p->lchild);  //左孩子入队
        if(p->rchild!=NULL)
            EnQueue(Q,p->rchild);  //右孩子入队
    }
}


//基于层序遍历打印出树的样子
void printBST(BSTree T){
    LinkQueue Q;
    InitQueue(Q);               //初始化辅助队列
    BSTree p;
    EnQueue(Q,T);               //将根结点入队

    //打印h层二叉树
    int h=treeDepth(T);
    int maxLayer = pow(2, h-1); //最下边一层有几个结点
    printf("\n树的亚子：\n");

    for (int i=1; i<=h; i++){
        int sum = pow(2, i-1);  //本层的结点数
        int gap = maxLayer/sum;      //本层各节点之间的间隔距离
        for (int j=0; j<gap/2; j++)
            printf("\t");
        //打印本层结点
        for (int j=0; j<sum; j++){
            DeQueue(Q, p);           //队头结点出队
            if (p==NULL){
                printf("空");
            }else{
                printf("%d", p->key);       //打印结点的值
            }

            //放入下一级的孩子
            if (p==NULL){
                EnQueue(Q,NULL);    //占两个位置
                EnQueue(Q,NULL);    //占两个位置
            } else {
                EnQueue(Q,p->lchild);  //左孩子入队
                EnQueue(Q,p->rchild);  //右孩子入队
            }

            for (int k=0; k<gap; k++)
                printf("\t");
        }
```

```
        printf("\n");
    }
}

//【测试代码】：测试二叉排序树的创建、遍历、求深度、插入、查找、删除 操作
int test_BinarySearchTree() {
    int str[] = {985,211,996,45,12,24};
    BSTree root;                    //定义一颗空的而叉查找树
    Creat_BST(root, str, 6);     //按照 str[] 数组的顺序依次往二叉查找树中插入元素，其中 6 是数组的长度，大家可自行更改

    printf("\n中序遍历：");
    InOrder(root);

    printf("\n先序遍历：");
    PreOrder(root);

    printf("\n后序遍历：");
    PostOrder(root);

    printf("\n层序遍历：");
    LevelOrder(root);

    printf("\n树的深度=%d", treeDepth(root));

    printBST(root);          //打印出树的样子

    //下面这段代码，用于测试插入操作
    BST_Insert(root, 666);  //往二叉排序树插入关键字666
    printBST(root);          //打印出树的样子
    BST_Insert(root, 1000);  //往二叉排序树插入关键字1000
    printBST(root);          //打印出树的样子

    //下面这段代码，用于测试查找操作（非递归实现）
    int key1 = 2333;
    BSTNode * result1;   //用于保存查找结果
    result1 = BST_Search(root, key1);       //在排序树中查找关键字 666
    if (result1!=NULL)
        printf("关键字%d查找成功（非递归实现）\n", key1);
    else
        printf("关键字%d查找失败（非递归实现）\n", key1);


    //下面这段代码，用于测试查找操作（非递归实现）
    int key2 = 996;
    BSTNode * result2;   //用于保存查找结果
    result2 = BST_Search(root, key2);       //在排序树中查找关键字 666
    if (result2!=NULL)
        printf("关键字%d查找成功（递归实现）\n", key2);
    else
        printf("关键字%d查找失败（递归实现）\n", key2);


    //下面这段代码，用于测试二叉排序树的删除操作。同学们可以自己更改要删除的关键字
    BST_DeleteKey(root, 211);
    printBST(root);          //删除之后打印出树的样子

    return 0;
}

#endif //CODE_BINARYSEARCHTREE_H
```