数据结构与算法

实验报告（四）

实验题目： 平衡二叉树

姓 名： 魏靖

学 号： 2021213513

日 期： 2022.06.15

自我评分： 【 A 】90

自我评分说明：A+，A，B+，B，B-，C，D，分别对应分数95、90、85、80、75、70、60

诚信声明

本人郑重承诺：本实验程序和实验报告均是本人独立学习和工作所获得的成果。尽我所知，实验报告中除特别标注的地方外，不包含其他同学已经发表或撰写过的成果；实验程序中对代码工作的任何帮助者所作的贡献均做了明确的说明，并表达了谢意。

如有抄袭，本人原因承担因此而造成的任何后果。

特此声明。

签名：魏靖 日期：2022.05.29

程序引用说明

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 引用项 | 来源 | 相同代码行数 |
| 1 | pow（） | https://baike.so.com/doc/6702561-6916512.html |  |
| 2 | system（“CLS“） | https://baike.so.com/doc/7035728-7258633.html |  |
| 3 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 小计 | | |  |

总代码行数\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; 引用占比\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1、实验简介**

通过具体代码实现平衡二叉树的创建、插入节点、查找元素、删除节点、输出二叉树等等功能。

1. **程序框架**

struct Node {

E data;

struct Node\* LCh;

struct Node\* RCh;

int balanceFctor; //平衡因子

};

class BalanceBiTree {

public:

BalanceBiTree(Node<E>\*& T); //初始化

static void menu(); //菜单

void destory(Node<E>\*& T); //销毁二叉树

void insert(Node<E>\*& T, Node<E>\* S); //将指针S所指节点插入二叉排序中

int BiTreeDepth(Node <E>\* T); //求树的高度

int getFactor(Node<E>\* T);

void Factors(Node<E>\*& T);

void FT(Node<E>\*& T, Node<E>\*& p); //获得平衡因子为2或-2的节点

void FTFather(Node<E>\*& T, Node<E>\*& f); //获得平衡因子为2或-2的节点的父节点

void LLAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void LRAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void RLAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void RRAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void AllAdjust(Node<E>\*& T);

//更新平衡因子

void BiTreeToArray(Node <E>\* T, E A[], int i, int& count);

//二叉树转数组

void print(Node <E>\* T, E B[], int num);

//对二叉链表表示的二叉树，按从上到下，从左到右打印结点值，即按层次打印

void create(Node<E>\*& T);

//创建平衡二叉树

void search(Node <E>\*& T, Node <E>\*& p, E x);

//查找元素x

Node <E>\* getFatherpointer(Node <E>\*& T, Node <E>\*& f, E x);

//获取某个元素的父亲指针，不存在返回NULL

void getPrior(Node <E>\*& T, E& min, E& max); //获取前驱元素

Node <E>\* getPriorpointer(Node <E>\*& T);

//获取某个元素的前驱指针

void getNext(Node <E>\*& T, E& min, E& max); //获取后继元素

Node <E>\* getNextpointer(Node <E>\*& T);

//获取某个元素的后继指针

void deleteLeaf(Node <E>\*& T, Node <E>\*& p, Node <E>\*& f);

//删除叶子节点

void deleteO(Node <E>\*& T, Node <E>\*& p, Node <E>\*& f);

//删除仅有左子树或只有右子树的节点

void deleteT(Node <E>\*& T, Node <E>\*& p);

//删除既有左子树又有右子树的节点

void alldelete(Node <E>\*& T, E x);

private:

Node<E>\* root; //树根

};

1. **关键代码实现**

**3.1平衡**

平衡二叉树是这样定义的：



简而言之表现为：它的左右子树都是平衡二叉树，左右高度差不超过一。

因而实现平衡的方法在于：实时动态平衡。

插入节点前检查是否破坏当前平衡→

Yes

（1）找出最小不平衡子树（不平衡就是左右子树高度差超一的那个子树）

（2）●旋转和调整●

①LL 顺时针旋转

断去平衡因子绝对值大于1的节点与它左孩子的链接

把它左孩子上升为它的父节点

把它左孩子的右孩子更新为它的左孩子

②RR 逆时针（恰与LL相反）

断去平衡因子绝对值大于1的节点与它右孩子的链接

把它右孩子上升为它的父节点

把它右孩子的左孩子更新为它的右孩子

③LR 先逆后顺

断去平衡因子绝对值大于1的节点与它左孩子的链接

把它左孩子的右孩子上升为它的父节点

把它左孩子的右孩子的左孩子更新为它左孩子的右孩子

把它左孩子的右孩子的右孩子更新为它的左孩子

④RL 先顺后逆（恰与LR相反）

综合判断：

NO 插入

（代码太多就不再粘贴）

**3.2删除**

基于平衡的原理，删除需考虑节点的子数个数设计不同的删除步骤（比如如果是叶子结点可以直接删除，而如果有子树又需要再将子树链接到父节点）；删除之后需要再平衡（随时处于动态更新中）。

**3.3输出与交互**

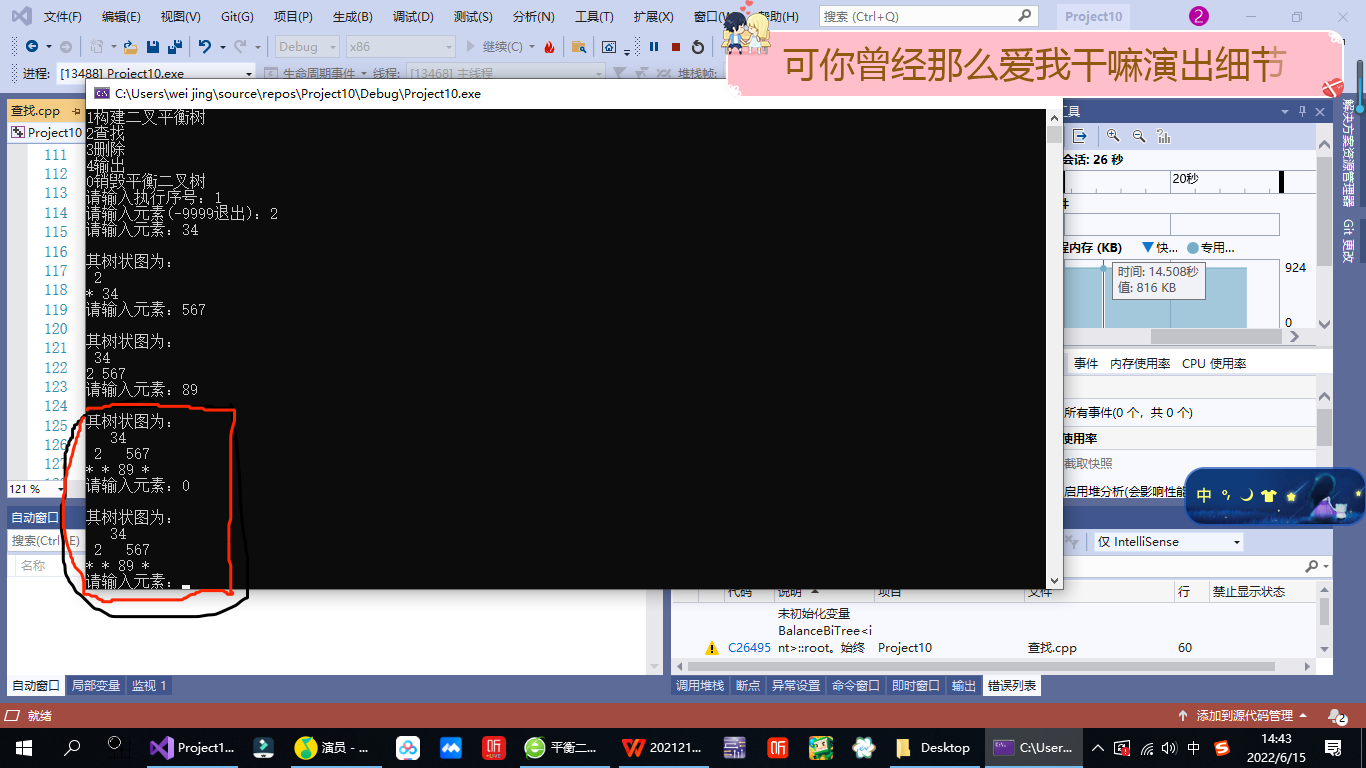
用system(“CLS”)清屏和static void menu()，在于每一次返回菜单（区别于我之前一直用的一种while（m）模式，自我感觉这样页面看上去的确会整洁一些）

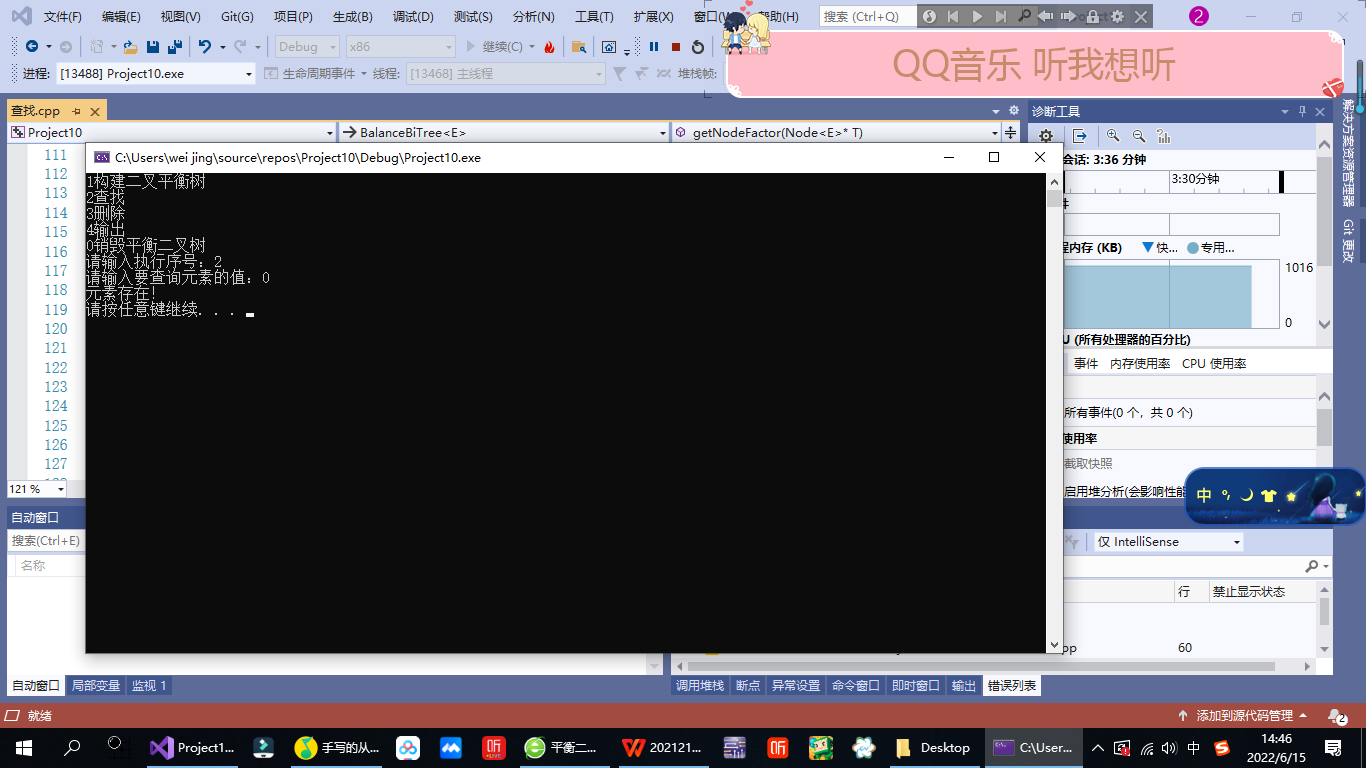
**3.4查找**

思来想去觉得还是应该把查找作为一个关键点，查找怎么最优，是我们这一章考虑的（当然本身平衡二叉树这样的结构就是便于查找的），所以想在这里再梳理一下平衡二叉树优化查找效率的机理。

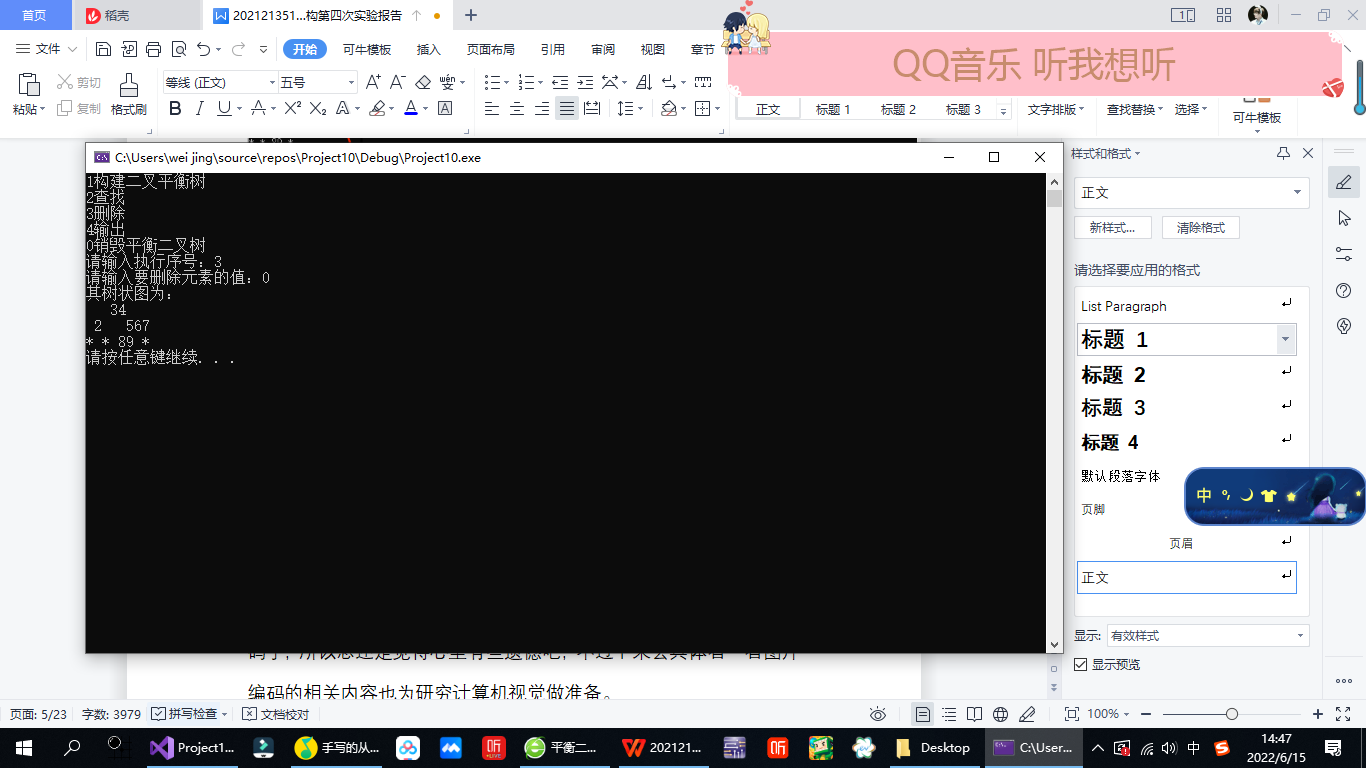
首先最为直观的是，就平衡二叉树的结构特征来说，他减少了二叉排序树的层次（让树的高度减小，查找算法的时间复杂度就会有相应的减小）。其次，二叉树本身来说就结合了数组查找快和链表插入快（我们在算法中也设计了 void BiTreeToArray(Node <E>\* T, E A[], int i, int& count)树转数组）的优点。基于此，平衡二叉树用于查找非常棒。

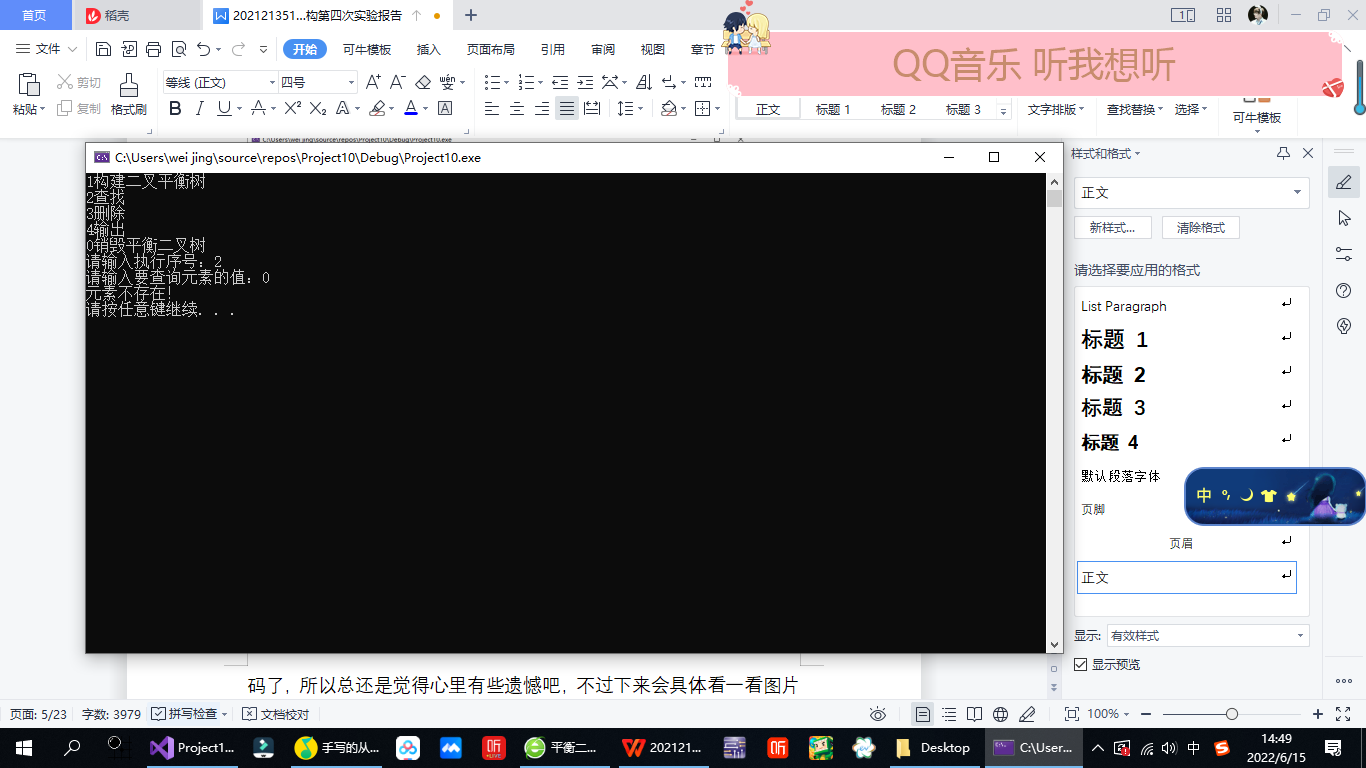
1. **不足**
2. 因为转化数组默认值的问题，无法插入值为0的节点（仅只无法在输出时表现出来，但是查找依然能够顺利找到和删除，如下）。





删除元素0再次查找发现：





1. 只能实现一次插入全部需插入值（因为是写完了才考虑到这个问题，所以后面也不大方便去大改了）。
2. 查找时只设计了找到元素而没有设计当时Huffman那种将位置输出出来。

（4）因为时间关系也没有去拓展更多的功能，但其实基于我们对前驱、后继节点和元素的定位，可以实现“显示前一个”“显示最大/最小值”等等功能。

1. **心得体会**

（1）平衡二叉树的平衡过程其实并不复杂，关键在于弄清楚平衡二叉树的定义和本质。

（2）在做实验之前是觉得这一章讲得比较快，可能基础部分还需要进一步理解和应用，所以没有选择扩展实验（加之扩展还有另外一种编码方式），但基于平衡二叉树的一系列操作设计（尤其是平衡），对它也有了一定的想法，会在假期进行实践。

（3）重温了在第一次试验时所提到的模板（记得当时没搞懂所以就没有用模板写，直到现在才大致明白了模板的妙用），最直观的感受就是把前面一部分都放在头文件里，那源代码就相对来说更简单易读，长篇代码比较冗杂（虽然每条函数定义前都得有“template< class E>”）。其次在于如果要把代码操作扩展出去，就会显得十分方便（不用大段复制粘贴去说明，只需要include这个头文件）。

**代码：（将头文件混一起了，没单独提出来）**

#include <iostream>

#include <Cstdlib>

#include <iostream>

#include <cmath>

using namespace std;

template< class E>

struct Node {

E data;

struct Node\* LCh;

struct Node\* RCh;

int balanceFctor; //平衡因子

};

template< class E>

class BalanceBiTree {

public:

BalanceBiTree(Node<E>\*& T); //初始化

static void menu(); //菜单

void destory(Node<E>\*& T); //销毁二叉树

void insert(Node<E>\*& T, Node<E>\* S); //将指针S所指节点插入二叉排序中

int BiTreeDepth(Node <E>\* T); //求树的高度

int getFactor(Node<E>\* T);

void Factors(Node<E>\*& T);

void FT(Node<E>\*& T, Node<E>\*& p); //获得平衡因子为2或-2的节点

void FTFather(Node<E>\*& T, Node<E>\*& f); //获得平衡因子为2或-2的节点的父节点

void LLAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void LRAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void RLAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void RRAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f);

void AllAdjust(Node<E>\*& T);

//更新平衡因子

void BiTreeToArray(Node <E>\* T, E A[], int i, int& count);

//二叉树转数组

void print(Node <E>\* T, E B[], int num);

//对二叉链表表示的二叉树，按从上到下，从左到右打印结点值，即按层次打印

void create(Node<E>\*& T);

//创建平衡二叉树

void search(Node <E>\*& T, Node <E>\*& p, E x);

//查找元素x

Node <E>\* getFatherpointer(Node <E>\*& T, Node <E>\*& f, E x);

//获取某个元素的父亲指针，不存在返回NULL

void getPrior(Node <E>\*& T, E& min, E& max); //获取前驱元素

Node <E>\* getPriorpointer(Node <E>\*& T);

//获取某个元素的前驱指针

void getNext(Node <E>\*& T, E& min, E& max); //获取后继元素

Node <E>\* getNextpointer(Node <E>\*& T);

//获取某个元素的后继指针

void deleteLeaf(Node <E>\*& T, Node <E>\*& p, Node <E>\*& f);

//删除叶子节点

void deleteO(Node <E>\*& T, Node <E>\*& p, Node <E>\*& f);

//删除仅有左子树或只有右子树的节点

void deleteT(Node <E>\*& T, Node <E>\*& p);

//删除既有左子树又有右子树的节点

void alldelete(Node <E>\*& T, E x);

private:

Node<E>\* root; //树根

};

template< class E>

BalanceBiTree<E>::BalanceBiTree(Node<E>\*& T)

{

T = NULL;

}

template< class E>

void BalanceBiTree<E>::menu()

{

cout << "1构建平衡二叉树" << endl;

cout << "2查找" << endl;

cout << "3删除" << endl;

cout << "4输出" << endl;

cout << "0销毁平衡二叉树" << endl;

}

template< class E>

void BalanceBiTree<E>::destory(Node<E>\*& T)

{

if (T)

{

destory(T->LCh);

destory(T->RCh);

delete T;

}

}

template< class E>

void BalanceBiTree<E>::insert(Node<E>\*& T, Node<E>\* S)

{

if (T == NULL)

T = S;

else if (S->data < T->data)

insert(T->LCh, S);

else

insert(T->RCh, S);

}

template< class E>

int BalanceBiTree<E>::BiTreeDepth(Node <E>\* T)

{

int m, n;

if (T == NULL)

return 0; //空树

else {

m = BiTreeDepth(T->LCh); //求左子树高度

n = BiTreeDepth(T->RCh);

if (m > n)

{

return m + 1;

}

else {

return n + 1;

}

}

}

template< class E>

int BalanceBiTree<E>::getFactor(Node<E>\* T)

{

int m = 0, n = 0;

if (T)

{

m = BiTreeDepth(T->LCh);

n = BiTreeDepth(T->RCh);

}

return m - n;

}

template< class E>

void BalanceBiTree<E>::Factors(Node<E>\*& T)

{

if (T)

{

T->balanceFctor = getFactor(T);

Factors(T->LCh);

Factors(T->RCh);

}

}

template< class E>

void BalanceBiTree<E>::FT(Node<E>\*& T, Node<E>\*& p)

{

if (T)

{

if (T->balanceFctor == 2 || T->balanceFctor == -2)

{

p = T;

}

FT(T->LCh, p);

FT(T->RCh, p);

}

}

template< class E>

void BalanceBiTree<E>::FTFather(Node<E>\*& T, Node<E>\*& f)

{

if (T)

{

if (T->LCh != NULL)

{

if (T->LCh->balanceFctor == 2 || T->LCh->balanceFctor == -2)

{

f = T;

}

}

if (T->RCh != NULL)

{

if (T->RCh->balanceFctor == 2 || T->RCh->balanceFctor == -2)

{

f = T;

}

}

FTFather(T->LCh, f);

FTFather(T->RCh, f);

}

}

template< class E>

void BalanceBiTree<E>::LLAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f)

{

Node<E>\* r;

if (T == p)

{

T = p->LCh; //将P的左孩子为新的根节点

r = T->RCh;

T->RCh = p; //p为其左孩子的右孩子

p->LCh = r; //将p原来的左孩子的右孩子连接其左孩子

}

else {

if (f->LCh == p)

{

f->LCh = p->LCh;

r = f->LCh->RCh;

f->LCh->RCh = p;

p->LCh = r;

}

if (f->RCh == p)

{

f->RCh = p->LCh;

r = f->RCh->RCh;

f->RCh->RCh = p;

p->LCh = r;

}

}

}

template< class E>

void BalanceBiTree<E>::LRAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f)

{

Node<E>\* l, \* r;

if (T == p) //->balanceFctor==2&&T->LCh->balanceFctor!=2

{

T = p->LCh->RCh; //将P的左孩子的右孩子为新的根节点

r = T->RCh;

l = T->LCh;

T->RCh = p;

T->LCh = p->LCh;

T->LCh->RCh = l;

T->RCh->LCh = r;

}

else {

if (f->RCh == p)

{

f->RCh = p->LCh->RCh;

r = f->RCh->RCh;

l = f->RCh->LCh;

f->RCh->RCh = p;

f->RCh->LCh = p->LCh;

f->RCh->LCh->RCh = l;

f->RCh->RCh->LCh = r;

}

if (f->LCh == p)

{

f->LCh = p->LCh->RCh;

r = f->LCh->RCh;

l = f->LCh->LCh;

f->LCh->RCh = p;

f->LCh->LCh = p->LCh;

f->LCh->LCh->RCh = l;

f->LCh->RCh->LCh = r;

}

}

}

template< class E>

void BalanceBiTree<E>::RLAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f)

{

Node<E>\* l, \* r;

if (T == p)

{

T = p->RCh->LCh;

r = T->RCh;

l = T->LCh;

T->LCh = p;

T->RCh = p->RCh;

T->LCh->RCh = l;

T->RCh->LCh = r;

}

else {

if (f->RCh == p)

{

f->RCh = p->RCh->LCh;

r = f->RCh->RCh;

l = f->RCh->LCh;

f->RCh->LCh = p;

f->RCh->RCh = p->RCh;

f->RCh->LCh->RCh = l;

f->RCh->RCh->LCh = r;

}

if (f->LCh == p)

{

f->LCh = p->RCh->LCh;

r = f->LCh->RCh;

l = f->LCh->LCh;

f->LCh->LCh = p;

f->LCh->RCh = p->RCh;

f->LCh->LCh->RCh = l;

f->LCh->RCh->LCh = r;

}

}

}

template< class E>

void BalanceBiTree<E>::RRAdjust(Node<E>\*& T, Node<E>\*& p, Node<E>\*& f)

{

Node<E>\* l;

if (T == p)

{

T = p->RCh;

l = T->LCh;

T->LCh = p;

p->RCh = l;

}

else {

if (f->RCh == p)

{

f->RCh = p->RCh;

l = f->RCh->LCh;

f->RCh->LCh = p;

p->RCh = l;

}

if (f->LCh == p)

{

f->LCh = p->RCh;

l = f->LCh->LCh;

f->LCh->LCh = p;

p->RCh = l;

}

}

}

template< class E>

void BalanceBiTree<E>::AllAdjust(Node<E>\*& T)

{

Node<E>\* f = NULL, \* p = NULL;

Factors(T);

FTFather(T, f);

FT(T, p);

while (p)

{

Factors(T);

if (p->balanceFctor == 2 && (p->LCh->balanceFctor == 1 || p->LCh->balanceFctor == 0))

{

LLAdjust(T, p, f);

Factors(T);

}

else if (p->balanceFctor == 2 && p->LCh->balanceFctor == -1)

{

LRAdjust(T, p, f);

Factors(T);

}

else if (p->balanceFctor == -2 && p->RCh->balanceFctor == 1)

{

RLAdjust(T, p, f);

Factors(T);

}

else if (p->balanceFctor == -2 && (p->RCh->balanceFctor == -1 || p->RCh->balanceFctor == 0)) //||p->RCh->balanceFctor==0

{

RRAdjust(T, p, f);

}

f = NULL;

p = NULL;

FTFather(T, f);

FT(T, p);

}

}

template<class E>

void BalanceBiTree<E>::BiTreeToArray(Node <E>\* T, E A[], int i, int& count)

{

if (T != NULL)

{

A[i] = T->data;

if (i > count)

count = i;

BiTreeToArray(T->LCh, A, 2 \* i, count);

BiTreeToArray(T->RCh, A, 2 \* i + 1, count);

}

}

template<class E>

void BalanceBiTree<E>::print(Node <E>\* T, E B[], int num)

{

int n, i, j, t, q, s, p, m = 0, k = 0;

n = (int)((log(num) / log(2)) + 1);

p = n;

for (i = 0; i < n; i++)

{

k += pow(2, m) ;

t = pow(2, m);

j = pow(2, p - 1) - 1;

q = pow(2, p) - 1;

s = q;

for (j; j > 0; j--)

{

cout << " ";

}

for (t; t <= k; t++)

{

if (B[t] == 0)

{

cout << "\*";

for (q; q > 0; q--)

cout << " ";

q = s;

}

else {

cout << B[t];

for (q; q > 0; q--)

cout << " ";

q = s;

}

}

m++;

p--;

j = n - i - 1;

cout << endl;

}

}

template< class E>

void BalanceBiTree<E>::create(Node<E>\*& T)

{

int level = 1;

int i = 1, j = 0;

int A[100] = { 0 };

int length = 0;

E x;

Node<E>\* S, \* p;

T = new Node<E>;

T->balanceFctor = 0;

T->LCh = NULL;

T->RCh = NULL;

p = T;

cout << "请输入元素(-9999退出)：";

cin >> x;

T->data = x;

while (x != -9999)

{

cout << "请输入元素：";

cin >> x;

if (x == -9999)

return;

S = new Node<E>;

S->data = x;

S->balanceFctor = 0;

S->LCh = NULL;

S->RCh = NULL;

insert(p, S);

AllAdjust(T);

p = T;

cout << endl;

BiTreeToArray(T, A, i, length);

cout << "其树状图为：" << endl;

print(T, A, length);

j = 0;

for (j; j < 100; j++)

A[j] = 0;

level = 1;

i = 1;

}

}

template<class E>

void BalanceBiTree<E>::search(Node <E>\*& T, Node <E>\*& p, E x)

{

if (T)

{

if (T->data == x)

p = T;

search(T->LCh, p, x);

search(T->RCh, p, x);

}

}

template<class E>

Node <E>\* BalanceBiTree<E>::getFatherpointer(Node <E>\*& T, Node <E>\*& f, E x)

{

if (T)

{

if (T->LCh != NULL)

{

if (T->LCh->data == x)

f = T;

}

if (T->RCh != NULL)

{

if (T->RCh->data == x)

f = T;

}

getFatherpointer(T->LCh, f, x);

getFatherpointer(T->RCh, f, x);

}

return f;

}

template<class E>

void BalanceBiTree<E>::getPrior(Node <E>\*& T, E& min, E& max)

{

if (T)

{

min = T->data;

if (min > max)

max = min;

getPrior(T->LCh, min, max);

getPrior(T->RCh, min, max);

}

}

template<class E>

Node <E>\* BalanceBiTree<E>::getPriorpointer(Node <E>\*& T)

{

Node <E>\* p;

E min = 0, max = -9999;

getPrior(T, min, max);

search(T, p, max);

return p;

}

template<class E>

void BalanceBiTree<E>::getNext(Node <E>\*& T, E& min, E& max)

{

if (T)

{

max = T->data;

if (min > max)

min = max;

getNext(T->LCh, min, max);

getNext(T->RCh, min, max);

}

}

template<class E>

Node <E>\* BalanceBiTree<E>::getNextpointer(Node <E>\*& T)

{

Node <E>\* p;

E min = 9999, max = 0;

getNext(T, min, max);

search(T, p, min);

return p;

}

template<class E>

void BalanceBiTree<E>::deleteLeaf(Node <E>\*& T, Node <E>\*& p, Node <E>\*& f)

{

if (p == NULL)

{

cout << "此节点不存在，不能删除" << endl;

return;

}

if (T == p) //根节点即为叶子节点

{

delete p;

T = NULL;

}

else { //删除节点为非根节点的叶子节点

if (f->LCh == p)

{

delete p;

f->LCh = NULL;

}

if (f->RCh == p)

{

delete p;

f->RCh = NULL;

}

}

}

template<class E>

void BalanceBiTree<E>::deleteO(Node <E>\*& T, Node <E>\*& p, Node <E>\*& f)

{

if (p == NULL)

{

cout << "此节点不存在，不能删除" << endl;

return;

}

if (T == p)

{

if (T->LCh == NULL && T->RCh != NULL)

{

T = p->RCh;

delete p;

}

if (T->RCh == NULL && T->LCh != NULL)

{

T = p->LCh;

delete p;

}

}

else {

if (p->LCh != NULL)

{

if (f->LCh == p)

f->LCh = p->LCh;

else

f->RCh = p->LCh;

}

if (p->RCh != NULL)

{

if (f->LCh == p)

f->LCh = p->RCh;

else

f->RCh = p->RCh;

}

}

}

template<class E>

void BalanceBiTree<E>::deleteT(Node <E>\*& T, Node <E>\*& p)

{

Node <E>\* f, \* next, \* prior;

if (p == NULL)

{

cout << "此节点不存在，不能删除" << endl;

return;

}

if (p->balanceFctor == 1) //p的平衡因子为1时，用p的前驱节点代替p

{

prior = getPriorpointer(p->LCh); //获得x的前驱指针

if (prior->LCh != NULL && prior->RCh == NULL) //前驱节点只有左孩子

{

p->data = prior->data;

prior->data = prior->LCh->data;

delete prior->LCh;

prior->LCh = NULL;

}

if (prior->LCh == NULL && prior->RCh == NULL) //前驱节点为叶子节点

{

getFatherpointer(T, f, prior->data); //得到前驱节点的父节点

p->data = prior->data;

delete prior;

f->RCh = NULL;

}

}

else if (p->balanceFctor == -1) //p的平衡因子为-1时，用p的后继节点代替p

{

next = getNextpointer(p->RCh); //获得x的后继指针

cout << next->data;

int level = 1;

if (next->RCh != NULL && next->LCh == NULL) //后继节点只有右孩子

{

p->data = next->data;

next->data = next->RCh->data;

delete next->RCh;

next->RCh = NULL;

}

else if (next->RCh == NULL && next->LCh == NULL) //后继节点为叶子节点

{

getFatherpointer(T, f, next->data); //得到后继节点的父节点

p->data = next->data;

delete next;

f->LCh = NULL;

}

}

else if (p->balanceFctor == 0) //p的平衡因子为0时，用p的前驱节点代替p

{

prior = getPriorpointer(p->LCh);

if (prior->LCh != NULL && prior->RCh == NULL)

{

p->data = prior->data;

prior->data = prior->LCh->data;

delete prior->LCh;

prior->LCh = NULL;

}

if (prior->LCh == NULL && prior->RCh == NULL)

{

getFatherpointer(T, f, prior->data);

p->data = prior->data;

delete prior;

if (p == f)

f->LCh = NULL;

else

f->RCh = NULL;

}

}

}

template<class E>

void BalanceBiTree<E>::alldelete(Node <E>\*& T, E x)

{

Node <E>\* f, \* p = NULL;

search(T, p, x);

getFatherpointer(T, f, x);

if (p == NULL)

{

cout << "不存在此节点，删除失败！" << endl;

return;

}

if (p->LCh == NULL && p->RCh == NULL) //删除节点为叶子节点

{

deleteLeaf(T, p, f);

if (T != NULL)

AllAdjust(T);

}

else if ((p->LCh == NULL && p->RCh != NULL) || (p->LCh != NULL && p->RCh == NULL))

{

deleteO(T, p, f);

if (T != NULL)

AllAdjust(T);

}

else

{

deleteT(T, p);

if (T != NULL)

AllAdjust(T);

}

}

void initArray(int A[])

{

int i = 0;

for (i; i < 100; i++)

A[i] = 0;

}

int main()

{

int x, y;

int i = 1;

int level = 1;

int A[100] = { 0 };

int B[100] = { 0 };

int length = 0; //存储数组A的有效元素个数

Node<int>\* root;

Node<int>\* p;

BalanceBiTree<int> T(root);

BalanceBiTree<int>::menu();

cout << "请输入执行序号：";

cin >> x;

while (x != 0)

{

switch (x)

{

case 1:

T.create(root);

break;

case 2:

cout << "请输入要查询元素的值：";

cin >> x;

T.search(root, p, x);

if (p != NULL)

{

if (p->data == x)

cout << "元素存在！" << endl;

else

cout << "元素不存在！" << endl;

}

else {

cout << "元素不存在！" << endl;

}

break;

case 3:

i = 1;

initArray(A);

level = 1;

cout << "请输入要删除元素的值：";

cin >> x;

T.alldelete(root, x);

T.BiTreeToArray(root, A, i, length);

cout << "其树状图为：" << endl;

T.print(root, A, length);

break;

case 4:

i = 1;

initArray(A);

T.AllAdjust(root);

T.BiTreeToArray(root, A, i, length);

cout << "其树状图为：" << endl;

T.print(root, A, length);

break;

}

system("pause");

system("CLS");

BalanceBiTree<int>::menu();

cout << "请输入执行序号：";

cin >> x;

}

if (root != NULL)

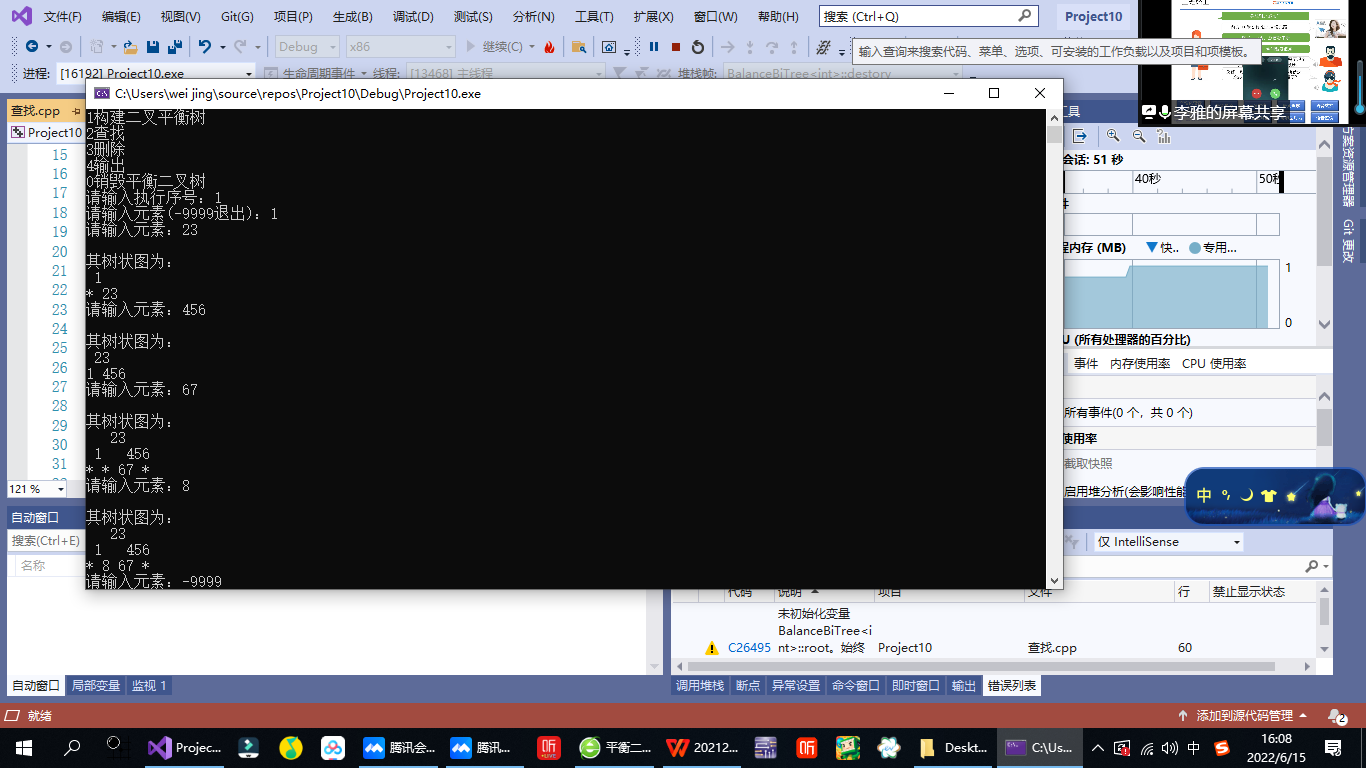
T.destory(root);

return 0;

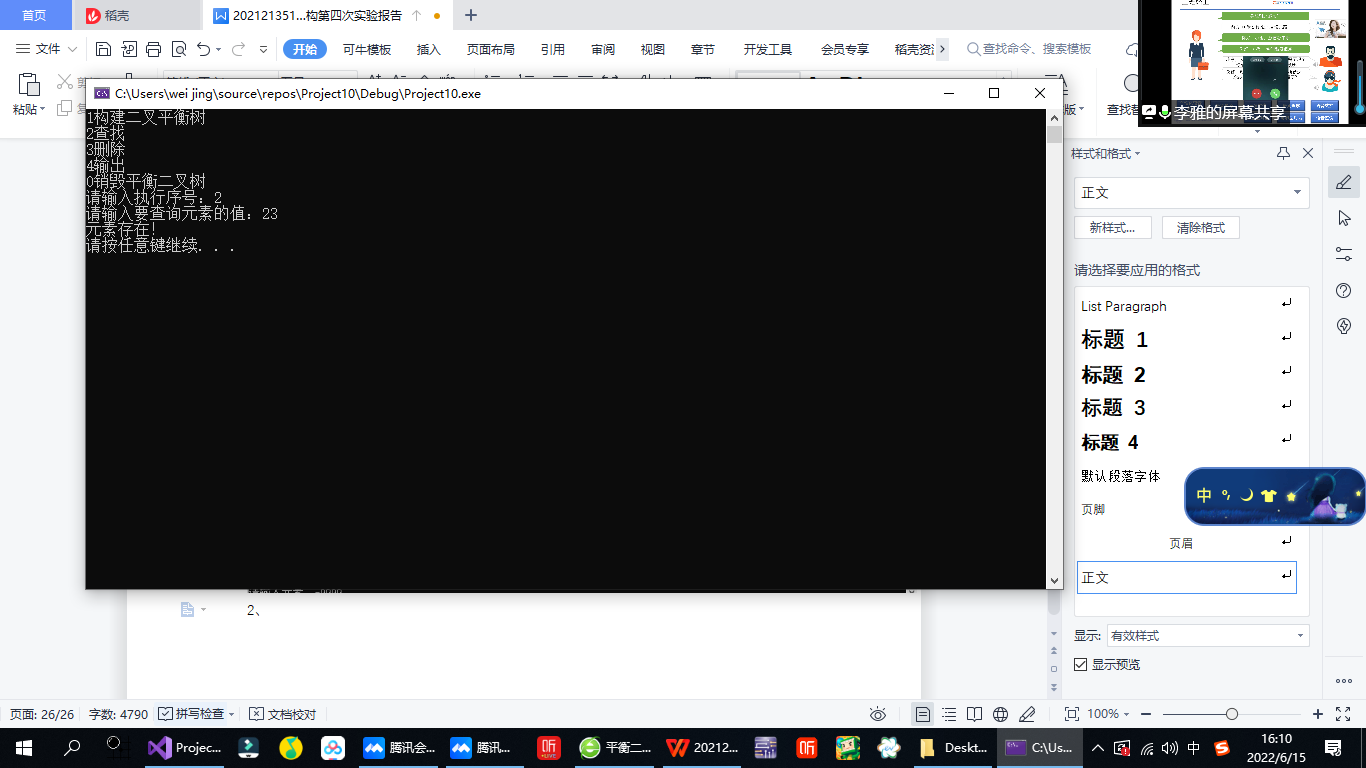
}

**调试结果：**

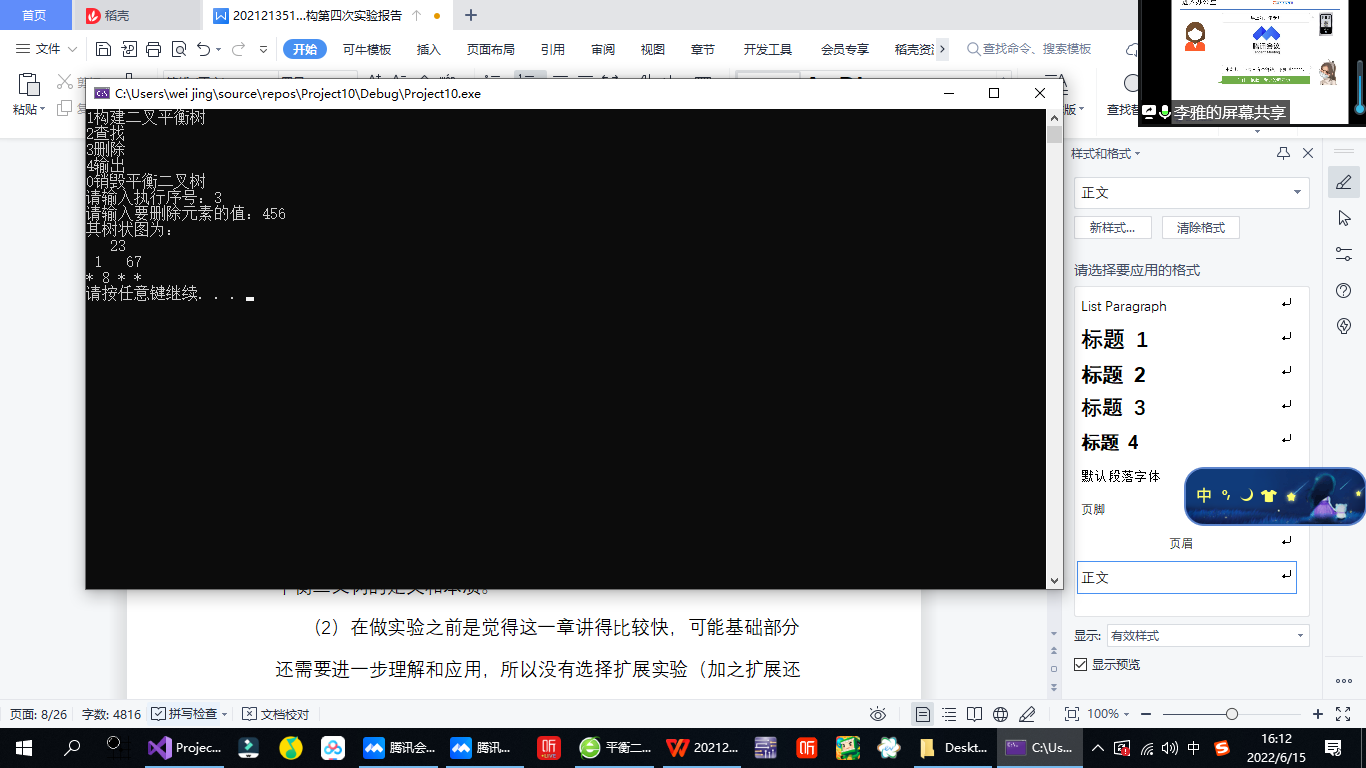
**1、创建和插入**



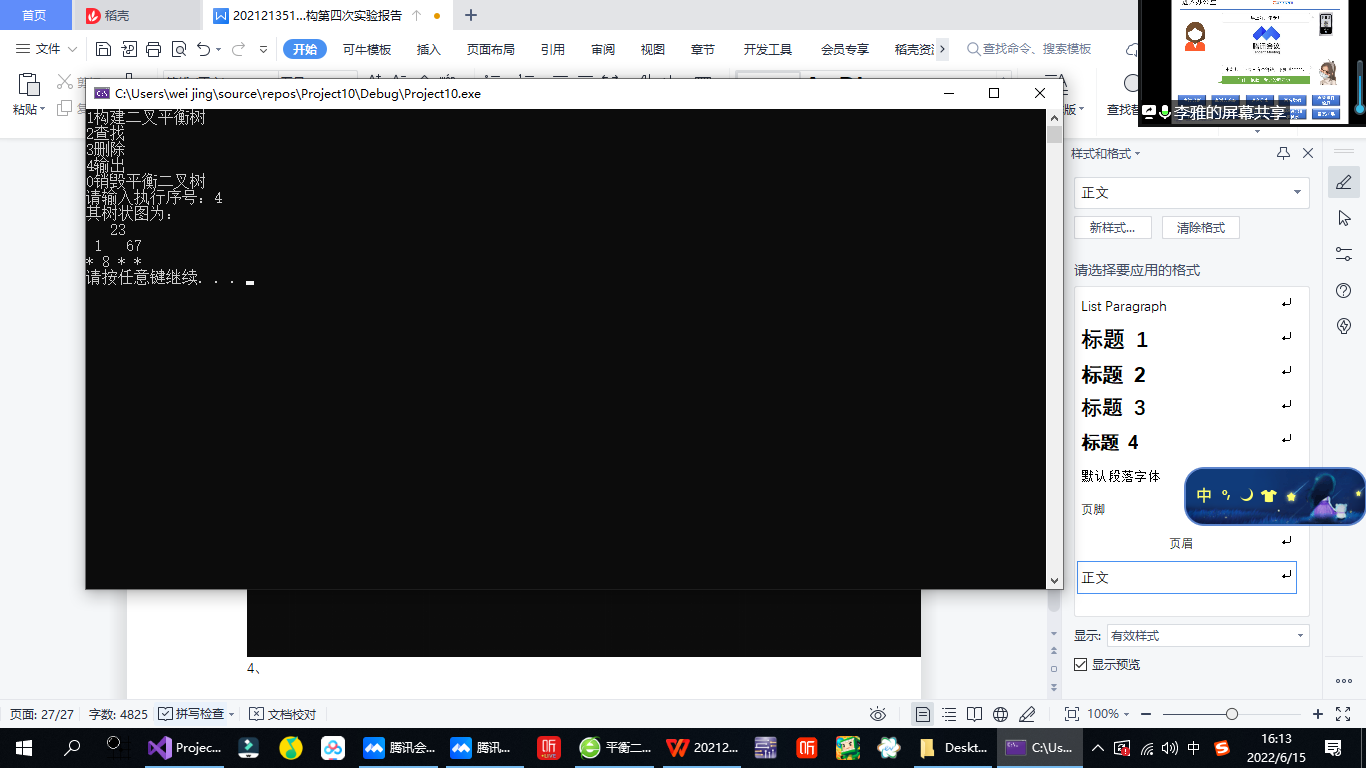
**2、查找**



**3、删除**



**4、输出**



**5、销毁**

