1.

题目：编程实现书P75  ADT BinaryTree 基本操作20个，用二叉链表结构实现；

算法：见代码

源代码：

#include<iostream>

#define TYPE int

using namespace std;

//二叉树节点类定义

class BiNode

{

public:

TYPE data;

BiNode \*lchild;//左孩子的指针

BiNode \*rchild;//右孩子的指针

};

//为便于层次遍历 再定义一个循环队列类 队列内存放节点的指针（提高效率）

class Queue

{

public:

BiNode \*\*dataPointers;//指针数组的首地址

int first,rear;//头与尾的索引

int capacity;//存储循环队列的容量

};

//队列初始化函数

void initQueue(Queue &Q,int nodeNum)

{

Q.dataPointers=new BiNode\*[nodeNum+1];//申请空间并牺牲一个空间以作为循环队列

Q.first=Q.rear=0;//队列置空

Q.capacity=nodeNum;

}

//入队函数

void enQueue(Queue &Q,BiNode \*p)

{

//由于已经申请了nodeNum+1的空间大小 故不可能队满 无需多余判断

Q.dataPointers[Q.rear]=p;

Q.rear=(Q.rear+1)%Q.capacity;

}

//出队函数 用p接收出队的指针

void deQueue(Queue &Q,BiNode \*p)

{

//由于运用时是仅当队不为空的情况 所以无需判空

p=Q.dataPointers[Q.first];

Q.first=(Q.first+1)%Q.capacity;

}

//队列判空函数

bool isQueueEmpty(Queue &Q)

{

if(Q.first==Q.rear)

{

return true;

}

return false;

}

//初始化函数 构造一个空二叉树

void initBiTree(BiNode\* &T)

{

T=NULL;//指针置空

if(!T)

{

cout<<"初始化成功"<<endl;//成功置空则成功

}

else

{

cout<<"初始化失败"<<endl;

}

}

//销毁函数 只能基于后序顺序销毁 销毁顺序为左右根 因为另两种方式会导致数据丢失

void destroyBiTree(BiNode\* &T)

{

//采用递归思想

if(T)//如果T存在再销毁 一定要加这一条！ 否则会出现内存泄漏的问题！

{

destroyBiTree(T->lchild);//销毁左子树

destroyBiTree(T->rchild);//销毁右子树

delete T;//销毁根节点

T=NULL;//根节点指针置空

}

cout<<"二叉链表销毁完毕"<<endl;

}

//创建二叉树函数 采用前序输入的方式创建

void createBiTree(BiNode\* &T)

{

char nodedata;

cin>>nodedata;

if(nodedata=='#')//注意字符一定要用单引号表示 否则会变成字符串

{

T=NULL;//该节点置空

}

else//表示数据有效 是需要存储的节点

{

T=new BiNode;

T->data=nodedata;//申请根节点的空间并存储数据

createBiTree(T->lchild);

createBiTree(T->rchild);//先左后右

}

cout<<"二叉链表创建完毕"<<endl;

}

bool biTreeEmpty(BiNode\* &T)

{

if(!T)

{

return true;

}

else

{

return false;

}

}

//求深度（高度）函数

//算法：基于后序遍历 先求左，再求右子树的高度 然后取最大值再加1

int biTreeDepth(BiNode\* &T)

{

int leftDepth;

int rightDepth;

if(T==NULL)

{

return 0;

}//若为空树则高度为0

else

{

leftDepth=biTreeDepth(T->lchild);

rightDepth=biTreeDepth(T->rchild);

return (leftDepth>=rightDepth)?(leftDepth+1):(rightDepth+1);

}

}

//返回根值函数 e用以接收

void root(BiNode\* &T,TYPE &e)

{

if(!T)

{

cout<<"为空树 根无存储数据"<<endl;

return;

}

else

{

e=T->data;

}

}

//取值函数 cur\_p表示目标节点的指针，e用以接收 注意此处cur\_p所指的节点一定是存在的

void value(BiNode\* &T,BiNode \*cur\_p,TYPE &e)

{

e=cur\_p->data;

}

//寻找双亲函数 返回值为双亲的指针 cur\_p为目标节点的指针，parent\_p用以接收 此处默认cur\_p不为空

void parent(BiNode\* &T,BiNode \*cur\_p,BiNode\* &parent\_p)

{

//采用先序递归的思路从根节点开始寻找

//先判该根节点的子代 再判左子节点的子代 再判右子节点的子代

//现将parent\_p初始化为NULL

parent\_p=NULL;

//递归一定要设置终止条件！对该算法而言 终止时传入的T为NULL，一旦传入NULL就直接返回 否则之后的步骤会出现NULL->lchild之类的情况

if(T==NULL)

{

parent\_p=NULL;

return;//查找失败 终止

}

//若不为空树（尚未终止） 注意cur\_p为根节点的情况 返回空指针

else if(cur\_p==T)

{

cout<<"根节点无双亲 返回值为NULL"<<endl;

parent\_p=NULL;

return;

}

else//若不为根节点且不为空树

{

//先检查根节点的子代

if(T->lchild==cur\_p||T->rchild==cur\_p)

{

parent\_p=T;

return;//查找结束一定要记得返回

}

else if(parent\_p==NULL)//cur\_p不为根节点且不是根节点的子代

{

parent(T->lchild,cur\_p,parent\_p);//找左节点的子代

}

if(parent\_p==NULL)//找完左节点的子代还是没有找到

{

parent(T->rchild,cur\_p,parent\_p);//找右节点的子代

}

//由于默认cur\_p是已存在的节点的指针 所以不用考虑没有找到的情况

}

}

//找左孩子函数

void leftChild(BiNode\* &T,BiNode \*cur\_p,BiNode\* &leftchild)

{

leftchild=cur\_p->lchild;//左孩子为空的情况包含在此语句中

}

//找右孩子函数

void rightChild(BiNode\* &T,BiNode \*cur\_p,BiNode\* &rightchild)

{

rightchild=cur\_p->rchild;//左孩子为空的情况包含在此语句中

}

//找左兄弟函数

void leftBrother(BiNode\* &T,BiNode \*cur\_p,BiNode\* &leftBrother\_p)

{

BiNode \*parent\_p;

parent(T,cur\_p,parent\_p);//寻找当前节点的双亲

if(parent\_p==NULL)//若最终找到的双亲为NULL 说明cur\_p为根节点 不可能有左兄弟 直接返回空

{

cout<<"为根节点 无左兄弟"<<endl;

leftBrother\_p=NULL;

return;

}

//能执行到下面 说明不为根节点

if(cur\_p==parent\_p->lchild)

{

cout<<"该节点是左节点 无左兄弟"<<endl;

leftBrother\_p=NULL;

return;

}

else

{

leftBrother\_p=parent\_p->lchild;//包含了左兄弟本身为空的情况

return;

}

}

//找右兄弟函数

void rightBrother(BiNode\* &T,BiNode \*cur\_p,BiNode\* &rightBrother\_p)

{

BiNode \*parent\_p;

parent(T,cur\_p,parent\_p);//寻找当前节点的双亲

if(parent\_p==NULL)//若最终找到的双亲为NULL 说明cur\_p为根节点 不可能有左兄弟 直接返回空

{

cout<<"为根节点 无右兄弟"<<endl;

rightBrother\_p=NULL;

return;

}

//能执行到下面 说明不为根节点

if(cur\_p==parent\_p->rchild)

{

cout<<"该节点是右节点 无右兄弟"<<endl;

rightBrother\_p=NULL;

return;

}

else

{

rightBrother\_p=parent\_p->rchild;//包含了右兄弟本身为空的情况

return;

}

}

void preOrderTraverse(BiNode\* &T)

{

if(T)

{

cout<<T->data<<endl;

preOrderTraverse(T->lchild);

preOrderTraverse(T->rchild);

}

}

void inOrderTraverse(BiNode\* &T)

{

if(T)

{

inOrderTraverse(T->lchild);

cout<<T->data<<endl;

inOrderTraverse(T->rchild);

}

}

void postOrderTraverse(BiNode\* &T)

{

if(T)

{

postOrderTraverse(T->lchild);

postOrderTraverse(T->rchild);

cout<<T->data<<endl;

}

}

//层次遍历函数

//算法：1.创建队列

//2.根节点入队

//3.若队列不为空 出队打印data 先看左子节点 若存在 入队 再看右节点 若存在 入队

void leverOrderTraverse(BiNode\* &T)

{

int nodeNum;

cout<<"请输入节点个数"<<endl;

cin>>nodeNum;

Queue helper;

initQueue(helper, nodeNum);

enQueue(helper, T);

while(!isQueueEmpty(helper))

{

BiNode \*p=NULL;//接收返回值的指针 初始化为空

deQueue(helper,p);

cout<<p->data<<" ";//打印data

if(p->lchild)

{

enQueue(helper,p->lchild);

}

if(p->rchild)

{

enQueue(helper, p->rchild);

}

}

}

//赋值函数 cur\_p是树中某节点的指针 value为需要赋的值

void assign(BiNode\* &T,BiNode\* &cur\_p,TYPE value)

{

cur\_p->data=value;

}

//插入子树函数 cur\_p表示插入节点的位置，LR指示插入在左还是右子树，c表示需要插入的二叉树的根节点的指针

//注意c默认为没有右子树 从而插入后原来的子树作为c的右子树即可

void insertChild(BiNode\* &T,BiNode \*cur\_p,int LR,BiNode\* &c)

{

BiNode \*copyChild;//用来备份原来子树的位置

if(LR==0)

{

copyChild=cur\_p->lchild;

cur\_p->lchild=c;

c->rchild=copyChild;

}

else if(LR==1)

{

copyChild=cur\_p->rchild;

cur\_p->rchild=c;

c->rchild=copyChild;

}

else

{

cout<<"LR输入有误"<<endl;

return;

}

}

//删除子树函数 需要依次将子树上的所有节点释放 调用销毁函数 后序遍历销毁即可

void deleteChild(BiNode\* &T,BiNode \*cur\_p,int LR)

{

if(LR==0)

{

if(cur\_p->lchild)

{

destroyBiTree(cur\_p->lchild);

cout<<"左子树删除成功"<<endl;

return;

}

else

{

cout<<"指定节点左子树不存在 无法删除"<<endl;

return;

}

}

else if(LR==1)

{

if(cur\_p->rchild)

{

destroyBiTree(cur\_p->rchild);

cout<<"右子树删除成功"<<endl;

}

else

{

cout<<"指定节点右子树不存在 无法删除"<<endl;

return;

}

}

else

{

cout<<"LR输入有误"<<endl;

return;

}

}

//按值查找节点函数 x为待查找值，cur\_p为返回的指针

//基于先序遍历的算法

void findNode(BiNode\* &T,TYPE x,BiNode\* &cur\_p)

{

cur\_p=NULL;//置空表示未找到的状态

//终止时传入的T为NULL 此时需要作提前返回的处理 否则后面会内存泄露

if(T==NULL)

{

cur\_p=NULL;

return;

}

//若不为空树 首先查找根

else if(T->data==x)

{

cur\_p=T;

return;//查找结束 返回

}

else//若不为空树且x不是根节点的值

{

findNode(T->lchild,x,cur\_p);//查找左子树

if(cur\_p==NULL)//查找完左子树还是没找到

{

findNode(T->rchild,x,cur\_p);

}

}

}

int main()

{

return 0;

}

2.

题目：.实现二叉树的先序、中序、后序遍历，用递归和非递归方法；实现层次遍历。

算法：见代码

源代码：

#include<iostream>

#define TYPE char

using namespace std;

class BiNode

{

public:

TYPE data;

BiNode \*lchild;

BiNode \*rchild;

};

class Queue

{

public:

BiNode \*\*dataPointers;

int first,rear,room;

};

class Stack

{

public:

BiNode \*\*base;

BiNode \*\*top;

int stackSize;

};

void initStack(Stack &S,int nodeNum)

{

S.base=new BiNode\*[nodeNum];//无需牺牲空间

S.base=S.top;

S.stackSize=nodeNum;

}

void pushStack(Stack &S,BiNode \*p)

{

\*(S.top)=p;

S.top++;

}

void popStack(Stack &S,BiNode\* &p)

{

S.top--;

p=\*(S.top);

}

bool isStackEmpty(Stack &S)

{

if(S.top==S.base)

{

return true;

}

else

{

return false;

}

}

void initQueue(Queue &Q,int nodeNum)

{

Q.dataPointers=new BiNode\*[nodeNum+1];//记得牺牲一个空间

Q.first=Q.rear=0;

Q.room=nodeNum+1;

}

void enQueue(Queue &Q,BiNode \*p)

{

Q.dataPointers[Q.rear]=p;

Q.rear=(Q.rear+1)%Q.room;

}

void deQueue(Queue &Q,BiNode\* &p)

{

p=Q.dataPointers[Q.first];

Q.first=(Q.first+1)%Q.room;

}

bool isQueueEmpty(Queue &Q)

{

if(Q.first==Q.rear)

{

return true;

}

else

{

return false;

}

}

void initBiTree(BiNode\* &T)

{

T=NULL;

if(!T)

{

cout<<"初始化成功"<<endl;

return;

}

cout<<"初始化失败"<<endl;

return;

}

void createBiTree(BiNode\* &T)

{

char nodeData;

cin>>nodeData;

if(nodeData=='#')

{

T=NULL;

}

else

{

T=new BiNode;

T->data=nodeData;

createBiTree(T->lchild);

createBiTree(T->rchild);

}

}

//先序递归

void preOrderTraverse\_Recursion(BiNode\* &T)

{

//终止时传入的T为NULL 所以要设置非空的条件

if(!T)

{

return;

}

else//不为空

{

cout<<T->data<<" ";//先打印根

preOrderTraverse\_Recursion(T->lchild);

preOrderTraverse\_Recursion(T->rchild);

}

}

//中序递归

void inOrderTraverse\_Recursion(BiNode\* &T)

{

if(!T)

{

return;

}

else

{

inOrderTraverse\_Recursion(T->lchild);

cout<<T->data<<" ";

inOrderTraverse\_Recursion(T->rchild);

}

}

//后序递归

void postOrderTraverse\_Recursion(BiNode\* &T)

{

if(!T)

{

return;

}

else

{

postOrderTraverse\_Recursion(T->lchild);

postOrderTraverse\_Recursion(T->rchild);

cout<<T->data<<" ";

}

}

//层次遍历

void levelOrderTraverse(BiNode\* &T,int nodeNum)

{

Queue Q;

initQueue(Q, nodeNum);

enQueue(Q, T);

while(!isQueueEmpty(Q))

{

BiNode \*p;

deQueue(Q, p);

cout<<p->data<<" ";

if(p->lchild)

{

enQueue(Q, p->lchild);

}

if(p->rchild)

{

enQueue(Q, p->rchild);

}

}

}

//先序非递归

//算法：1.将根节点入栈

//2.当栈不为空时出栈 并依次检索出栈节点的右和左子树 若存在 依次入栈

void preOrderTraverse\_Non\_Recursion(BiNode\* &T,int nodeNum)

{

Stack S;

initStack(S, nodeNum);

pushStack(S, T);

while(!isStackEmpty(S))

{

BiNode \*p;

popStack(S, p);

cout<<p->data<<endl;

if(p->rchild)

{

pushStack(S, p->rchild);

}

if(p->lchild)

{

pushStack(S, p->lchild);

}

}

}

//中序非递归

void inOrderTraverse\_Non\_Recursion(BiNode\* &T,int nodeNum)

{

Stack S;

initStack(S, nodeNum);

BiNode \*p=T;//遍历辅助指针

while(p||!isStackEmpty(S))//当p不为空或者栈不为空时 都需要继续遍历

{

if(p)

{

pushStack(S, p);

p=p->lchild;//这两步操作相当于将根节点与所有左节点依次入栈 由此出栈访问时得以从最底下开始

}

else//一旦左节点全部入栈 访问完了，就开始出栈打印节点data 然后访问右子树 重复之前操作

{

popStack(S, p);//令p接收出栈值

cout<<p->data<<endl;

p=p->rchild;

}

}

}

int main()

{

return 0;

}

3.

题目：设二叉树采用二叉链表存储，编写函数，对二叉树中每个元素值为x的结点，删除以它为根的子树，并释放相应空间。（习题集6.45）

算法：采用层次遍历 遇到x即删除子树

源代码：

//算法：采用层次遍历或前序遍历 若采用中序或后序可能会导致删除次数增加

#include<iostream>

#define TYPE char

using namespace std;

class BiNode

{

public:

TYPE data;

BiNode \*lchild;

BiNode \*rchild;

};

class Queue

{

public:

BiNode \*\*dataPointers;

int first,rear;

int room;

};

void initQueue(Queue &Q,int nodeNum)

{

Q.dataPointers=new BiNode\*[nodeNum+1];

Q.room=nodeNum+1;

Q.first=Q.rear=0;

}

void enQueue(Queue &Q,BiNode \*p)

{

Q.dataPointers[Q.rear]=p;

Q.rear=(Q.rear+1)%Q.room;

}

void deQueue(Queue &Q,BiNode\* &p)

{

p=Q.dataPointers[Q.first];

Q.first=(Q.first+1)%Q.room;

}

void createBiTree(BiNode\* &T)

{

char ch;

cin>>ch;

if(ch=='#')

{

T=NULL;

}

else

{

T=new BiNode;

T->data=ch;

createBiTree(T->lchild);

createBiTree(T->rchild);

}

}

void destroyBiTree(BiNode\* &T)

{

if(!T)

{

return;

}

else

{

destroyBiTree(T->lchild);

destroyBiTree(T->rchild);

delete T;

T=NULL;

}

}

void deleteXChildren(BiNode\* &T,int nodeNum,int x)

{

Queue Q;

initQueue(Q, nodeNum);

enQueue(Q, T);

while(Q.first!=Q.rear)

{

BiNode \*p;

deQueue(Q, p);

if(p->data==x)

{

destroyBiTree(p->lchild);

destroyBiTree(p->rchild);

continue;

}

if(p->lchild)

{

enQueue(Q, p->lchild);

}

if(p->rchild)

{

enQueue(Q, p->rchild);

}

}

}

int main()

{

return 0;

}

4.

题目：判断完全二叉树

算法：基于层次遍历 对度为0，1，2的点分别判断是否满足完全二叉树的条件

源代码：

//算法：基于层次遍历 依次判断完全二叉树应满足的条件

#include<iostream>

#define TYPE char

using namespace std;

class BiNode

{

public:

TYPE data;

BiNode \*lchild;

BiNode \*rchild;

};

class Queue

{

public:

BiNode \*\*dataPointers;

int first,rear,room;

};

void createBiTree(BiNode\* &T)

{

char ch;

cin>>ch;

if(ch=='#')

{

T=NULL;

}

else

{

T=new BiNode;

T->data=ch;

createBiTree(T->lchild);

createBiTree(T->rchild);

}

}

void initQueue(Queue &Q,int nodeNum)

{

Q.dataPointers=new BiNode\*[nodeNum+1];

Q.room=nodeNum+1;

Q.first=Q.rear=0;

}

void enQueue(Queue &Q,BiNode \*p)

{

Q.dataPointers[Q.rear]=p;

Q.rear=(Q.rear+1)%Q.room;

}

void deQueue(Queue &Q,BiNode\* &p)

{

p=Q.dataPointers[Q.first];

Q.first=(Q.first+1)%Q.room;

}

bool isComplete(BiNode\* &T,int nodeNum)

{

Queue Q;

initQueue(Q, nodeNum);

//根节点入队

enQueue(Q,T);

//声明一个变量记录已访问节点中是否有度数小于等于1的节点 初始化为假

bool existSpecialNode=false;

//层次遍历

while(Q.first!=Q.rear)

{

//队头出队

BiNode \*p;//接收出队节点

deQueue(Q, p);

//下根据出队节点的度数进行判别

//1.0度点

if(!p->lchild&&!p->rchild)

{

//无法判别 直接改变状态量即可

existSpecialNode=true;

//无子节点 访问后无需将子节点入队

}

//2.1度点

//分只有左没有右 和 只有右没有左 两种情况

else if(p->rchild&&!p->lchild)//有右无左

{

return false;//非完全 结束

}

else if(p->lchild&&!p->rchild)//有左无右

{

//在出现1度点之前 不可能出现度数<=1的点

if(existSpecialNode)//存在度数……的点

{

return false;//非完全 结束

}

else//不存在度数……的点

{

//改变状态量

existSpecialNode=true;

//左子节点入队

enQueue(Q, p->lchild);

}

}

//3.2度点

else

{

//2度点不可能出现在度数<=1的点之后

if(existSpecialNode)

{

return false;//非完全 结束

}

else

{

//子代入队 继续遍历

enQueue(Q, p->lchild);

enQueue(Q, p->rchild);

}

}

}

//若在整个循环过程中没有返回 则是完全二叉树

return true;

}

int main()

{

return 0;

}

5.

题目：俄罗斯方块是俄罗斯人阿列克谢·帕基特诺夫发明的一款休闲游戏。

游戏在一个15行10列的方格图上进行，方格图上的每一个格子可能已经放置了方块，或者没有放置方块。每一轮，都会有一个新的由4个小方块组成的板块从方格图的上方落下，玩家可以操作板块左右移动放到合适的位置，当板块中某一个方块的下边缘与方格图上的方块上边缘重合或者达到下边界时，板块不再移动，如果此时方格图的某一行全放满了方块，则该行被消除并得分。

在这个问题中，你需要写一个程序来模拟板块下落，你不需要处理玩家的操作，也不需要处理消行和得分。

具体的，给定一个初始的方格图，以及一个板块的形状和它下落的初始位置，你要给出最终的方格图。

输入格式：

输入的前15行包含初始的方格图，每行包含10个数字，相邻的数字用空格分隔。如果一个数字是0，表示对应的方格中没有方块，如果数字是1，则表示初始的时候有方块。输入保证前4行中的数字都是0。

输入的第16至第19行包含新加入的板块的形状，每行包含4个数字，组成了板块图案，同样0表示没方块，1表示有方块。输入保证板块的图案中正好包含4个方块，且4个方块是连在一起的（准确的说，4个方块是四连通的，即给定的板块是俄罗斯方块的标准板块）。

第20行包含一个1到7之间的整数，表示板块图案最左边开始的时候是在方格图的哪一列中。注意，这里的板块图案指的是16至19行所输入的板块图案，如果板块图案的最左边一列全是0，则它的左边和实际所表示的板块的左边是不一致的（见样例）

算法：创建对比下一状态的比对函数 若下一位置已经有图案 在当前位置停止 若没有 递归下移

源代码：

//算法：模拟一个下移一格的函数 如果下一格位置有1则放置在当前位置 直到放置到结尾

#include<iostream>

#include<vector>

using namespace std;

void printBlocks(vector<vector<int>> &blocks)

{

for(int i=0;i<15;i++)

{

for(int j=0;j<10;j++)

{

cout<<blocks[i][j]<<" ";

}

cout<<endl;

}

}

//找到板块图案在游戏界面中的初始坐标

void findIndexsInBlocks(vector<pair<int,int>> &indexsOfPatterns,int startCol)

{

//行坐标无需变更 只需变更列坐标

for(vector<pair<int,int>>::iterator it=indexsOfPatterns.begin();it!=indexsOfPatterns.end();it++)

{

(\*it).second+=startCol;

}

}

//检查冲突函数

bool isAttacking(int row,int col,vector<vector<int>> &blocks)

{

if(blocks[row][col]==1)

{

return true;

}

else

{

return false;

}

}

//下移函数 检查下移后的坐标中是否有1 如果有 将当前位的格子赋值为1 如果没有 继续下移 直到达到界面底端

//递归思想

void downOneBlock(vector<pair<int,int>> &indexOfPatterns,vector<vector<int>> &blocks,int bottomOfPatterns)

{

//如果已经到达了界面底部 直接返回

if(bottomOfPatterns==14)

{

return;

}

else//如果尚未达到底部

{

//检查

for(vector<pair<int,int>>::iterator it=indexOfPatterns.begin();it!=indexOfPatterns.end();it++)

{

if(isAttacking((\*it).first+1, (\*it).second, blocks))

{

return;//如果有冲突 直接返回

}

}

//如果前面没有返回 意味着没有发生冲突的情况 即图案可以下移

//下移

for(vector<pair<int,int>>::iterator it=indexOfPatterns.begin();it!=indexOfPatterns.end();it++)

{

(\*it).first+=1;

}

//底部下移

bottomOfPatterns++;

//递归

downOneBlock(indexOfPatterns, blocks, bottomOfPatterns);

}

}

int main()

{

//游戏界面

vector<vector<int>> blocks;

//输入图形 直接存储所有1块在板块图形中的坐标

vector<pair<int,int>> indexsOfPatterns;

//记录图案底部的坐标

int bottomOfPatterns=-1;

//初始列 注意是整个板块的初始列

int startCol;

//预设空间

blocks.resize(15);

for(int i=0;i<15;i++)

{

blocks[i].reserve(10);

}

//输入

for(int i=0;i<15;i++)

{

for(int j=0;j<10;j++)

{

int x;

cin>>x;

blocks[i].push\_back(x);

}

}

for(int i=0;i<4;i++)

{

for(int j=0;j<4;j++)

{

int x;

cin>>x;

if(x==1)

{

indexsOfPatterns.emplace\_back(i,j);//存储有效方块在图案中的坐标

if(i>bottomOfPatterns)

{

bottomOfPatterns=i;//更新底部坐标

}

}

}

}

cin>>startCol;

//注意到此处的startCol是实际列数 需要在此基础上自减

startCol--;

//输入完成

//找到在界面中的坐标

findIndexsInBlocks(indexsOfPatterns, startCol);

downOneBlock(indexsOfPatterns, blocks, bottomOfPatterns);

//此时的indexOfPatterns存储有最终的添加点坐标

//修改原界面

for(vector<pair<int,int>>::iterator it=indexsOfPatterns.begin();it!=indexsOfPatterns.end();it++)

{

blocks[(\*it).first][(\*it).second]=1;

}

//输出答案

printBlocks(blocks);

return 0;

}