#define BinNodePosi(T) BinNode<T>\* //节点位置

#define stature(p) ((p) ? (p)->height : -1) //节点高度（与“空树高度为-1”的约定相统一）

typedef enum { RB\_RED, RB\_BLACK} RBColor; //节点颜色

template <typename T> struct BinNode { //二叉树节点模板类

// 成员（为简化描述起见统一开放，读者可根据需要进一步封装）

T data; //数值

BinNodePosi(T) parent; BinNodePosi(T) lc; BinNodePosi(T) rc; //父节点及左、右孩子

int height; //高度（通用）

int npl; //Null Path Length（左式堆，也可直接用height代替）

RBColor color; //颜色（红黑树）

// 构造函数

BinNode() :

parent ( NULL ), lc ( NULL ), rc ( NULL ), height ( 0 ), npl ( 1 ), color ( RB\_RED ) { }

BinNode ( T e, BinNodePosi(T) p = NULL, BinNodePosi(T) lc = NULL, BinNodePosi(T) rc = NULL,

int h = 0, int l = 1, RBColor c = RB\_RED ) :

data ( e ), parent ( p ), lc ( lc ), rc ( rc ), height ( h ), npl ( l ), color ( c ) { }

// 操作接口

int size(); //统计当前节点后代总数，亦即以其为根的子树的规模

BinNodePosi(T) insertAsLC ( T const& ); //作为当前节点的左孩子插入新节点

BinNodePosi(T) insertAsRC ( T const& ); //作为当前节点的右孩子插入新节点

BinNodePosi(T) succ(); //取当前节点的直接后继

template <typename VST> void travLevel ( VST& ); //子树层次遍历

template <typename VST> void travPre ( VST& ); //子树先序遍历

template <typename VST> void travIn ( VST& ); //子树中序遍历

template <typename VST> void travPost ( VST& ); //子树后序遍历

// 比较器、判等器（各列其一，其余自行补充）

bool operator< ( BinNode const& bn ) { return data < bn.data; } //小于

bool operator== ( BinNode const& bn ) { return data == bn.data; } //等于

/\*DSA\*/

/\*DSA\*/BinNodePosi(T) zig(); //顺时针旋转

/\*DSA\*/BinNodePosi(T) zag(); //逆时针旋转

};

template <typename T> //统计二叉树子树规模

int BinTree<T>::size ( BinNodePosi(T) x ) //assert: x为二叉树中的合法位置

{ return x ? size ( x->lc ) + size ( x->RChild ) : 0; }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* BinNode状态与性质的判断

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define IsRoot(x) ( ! ( (x).parent ) )

#define IsLChild(x) ( ! IsRoot(x) && ( & (x) == (x).parent->lc ) )

#define IsRChild(x) ( ! IsRoot(x) && ( & (x) == (x).parent->rc ) )

#define HasParent(x) ( ! IsRoot(x) )

#define HasLChild(x) ( (x).lc )

#define HasRChild(x) ( (x).rc )

#define HasChild(x) ( HasLChild(x) || HasRChild(x) ) //至少拥有一个孩子

#define HasBothChild(x) ( HasLChild(x) && HasRChild(x) ) //同时拥有两个孩子

#define IsLeaf(x) ( ! HasChild(x) )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 与BinNode具有特定关系的节点及指针

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#define sibling(p) /\*兄弟\*/ \

( IsLChild( \* (p) ) ? (p)->parent->rc : (p)->parent->lc )

#define uncle(x) /\*叔叔\*/ \

( IsLChild( \* ( (x)->parent ) ) ? (x)->parent->parent->rc : (x)->parent->parent->lc )

#define FromParentTo(x) /\*来自父亲的引用\*/ \

( IsRoot(x) ? \_root : ( IsLChild(x) ? (x).parent->lc : (x).parent->rc ) )

template <typename T> BinNodePosi(T) BinNode<T>::insertAsLC ( T const& e )

{ return lc = new BinNode ( e, this ); } //将e作为当前节点的左孩子插入二叉树

template <typename T> BinNodePosi(T) BinNode<T>::insertAsRC ( T const& e )

{ return rc = new BinNode ( e, this ); } //将e作为当前节点的右孩子插入二叉树

#include "BinNode.h" //引入二叉树节点类

template <typename T> class BinTree { //二叉树模板类

protected:

int \_size; BinNodePosi(T) \_root; //规模、根节点

virtual int updateHeight ( BinNodePosi(T) x ); //更新节点x的高度

void updateHeightAbove ( BinNodePosi(T) x ); //更新节点x及其祖先的高度

public:

BinTree() : \_size ( 0 ), \_root ( NULL ) { } //构造函数

~BinTree() { if ( 0 < \_size ) remove ( \_root ); } //析构函数

int size() const { return \_size; } //规模

bool empty() const { return !\_root; } //判空

BinNodePosi(T) root() const { return \_root; } //树根

BinNodePosi(T) insertAsRoot ( T const& e ); //插入根节点

BinNodePosi(T) insertAsLC ( BinNodePosi(T) x, T const& e ); //e作为x的左孩子（原无）插入

BinNodePosi(T) insertAsRC ( BinNodePosi(T) x, T const& e ); //e作为x的右孩子（原无）插入

BinNodePosi(T) attachAsLC ( BinNodePosi(T) x, BinTree<T>\* &T ); //T作为x左子树接入

BinNodePosi(T) attachAsRC ( BinNodePosi(T) x, BinTree<T>\* &T ); //T作为x右子树接入

int remove ( BinNodePosi(T) x ); //删除以位置x处节点为根的子树，返回该子树原先的规模

BinTree<T>\* secede ( BinNodePosi(T) x ); //将子树x从当前树中摘除，并将其转换为一棵独立子树

template <typename VST> //操作器

void travLevel ( VST& visit ) { if ( \_root ) \_root->travLevel ( visit ); } //层次遍历

template <typename VST> //操作器

void travPre ( VST& visit ) { if ( \_root ) \_root->travPre ( visit ); } //先序遍历

template <typename VST> //操作器

void travIn ( VST& visit ) { if ( \_root ) \_root->travIn ( visit ); } //中序遍历

template <typename VST> //操作器

void travPost ( VST& visit ) { if ( \_root ) \_root->travPost ( visit ); } //后序遍历

bool operator< ( BinTree<T> const& t ) //比较器（其余自行补充）

{ return \_root && t.\_root && lt ( \_root, t.\_root ); }

bool operator== ( BinTree<T> const& t ) //判等器

{ return \_root && t.\_root && ( \_root == t.\_root ); }

/\*DSA\*/

/\*DSA\*/void stretchToLPath() { stretchByZag ( \_root ); } //借助zag旋转，转化为左向单链

/\*DSA\*/void stretchToRPath() { stretchByZig ( \_root, \_size ); } //借助zig旋转，转化为右向单链

}; //BinTree

template <typename T> int BinTree<T>::updateHeight ( BinNodePosi(T) x ) //更新节点x高度

{ return x->height = 1 + max ( stature ( x->lc ), stature ( x->rc ) ); } //具体规则，因树而异

template <typename T> void BinTree<T>::updateHeightAbove ( BinNodePosi(T) x ) //更新高度

{ while ( x ) { updateHeight ( x ); x = x->parent; } } //从x出发，覆盖历代祖先。可优化

template <typename T> BinNodePosi(T) BinTree<T>::insertAsRoot ( T const& e )

{ \_size = 1; return \_root = new BinNode<T> ( e ); } //将e当作根节点插入空的二叉树

template <typename T> BinNodePosi(T) BinTree<T>::insertAsLC ( BinNodePosi(T) x, T const& e )

{ \_size++; x->insertAsLC ( e ); updateHeightAbove ( x ); return x->lc; } //e插入为x的左孩子

template <typename T> BinNodePosi(T) BinTree<T>::insertAsRC ( BinNodePosi(T) x, T const& e )

{ \_size++; x->insertAsRC ( e ); updateHeightAbove ( x ); return x->rc; } //e插入为x的右孩子

template <typename T> //二叉树子树接入算法：将S当作节点x的左子树接入，S本身置空

BinNodePosi(T) BinTree<T>::attachAsLC ( BinNodePosi(T) x, BinTree<T>\* &S ) { //x->lc == NULL

if ( x->lc = S->\_root ) x->lc->parent = x; //接入

\_size += S->\_size; updateHeightAbove ( x ); //更新全树规模与x所有祖先的高度

S->\_root = NULL; S->\_size = 0; release ( S ); S = NULL; return x; //释放原树，返回接入位置

}

template <typename T> //二叉树子树接入算法：将S当作节点x的右子树接入，S本身置空

BinNodePosi(T) BinTree<T>::attachAsRC ( BinNodePosi(T) x, BinTree<T>\* &S ) { //x->rc == NULL

if ( x->rc = S->\_root ) x->rc->parent = x; //接入

\_size += S->\_size; updateHeightAbove ( x ); //更新全树规模与x所有祖先的高度

S->\_root = NULL; S->\_size = 0; release ( S ); S = NULL; return x; //释放原树，返回接入位置

}

template <typename T> //删除二叉树中位置x处的节点及其后代，返回被删除节点的数值

int BinTree<T>::remove ( BinNodePosi(T) x ) { //assert: x为二叉树中的合法位置

FromParentTo ( \*x ) = NULL; //切断来自父节点的指针

updateHeightAbove ( x->parent ); //更新祖先高度

int n = removeAt ( x ); \_size -= n; return n; //删除子树x，更新规模，返回删除节点总数

}

template <typename T> //删除二叉树中位置x处的节点及其后代，返回被删除节点的数值

static int removeAt ( BinNodePosi(T) x ) { //assert: x为二叉树中的合法位置

if ( !x ) return 0; //递归基：空树

int n = 1 + removeAt ( x->lc ) + removeAt ( x->rc ); //递归释放左、右子树

release ( x->data ); release ( x ); return n; //释放被摘除节点，并返回删除节点总数

}

template <typename T> //二叉树子树分离算法：将子树x从当前树中摘除，将其封装为一棵独立子树返回

BinTree<T>\* BinTree<T>::secede ( BinNodePosi(T) x ) { //assert: x为二叉树中的合法位置

FromParentTo ( \*x ) = NULL; //切断来自父节点的指针

updateHeightAbove ( x->parent ); //更新原树中所有祖先的高度

BinTree<T>\* S = new BinTree<T>; S->\_root = x; x->parent = NULL; //新树以x为根

S->\_size = x->size(); \_size -= S->\_size; return S; //更新规模，返回分离出来的子树

}

template <typename T, typename VST> //元素类型、操作器

void travPre\_R ( BinNodePosi(T) x, VST& visit ) { //二叉树先序遍历算法（递归版）

if ( !x ) return;

visit ( x->data );

travPre\_R ( x->lc, visit );

travPre\_R ( x->rc, visit );

}

template <typename T, typename VST> //元素类型、操作器

void travPost\_R ( BinNodePosi(T) x, VST& visit ) { //二叉树后序遍历算法（递归版）

if ( !x ) return;

travPost\_R ( x->lc, visit );

travPost\_R ( x->rc, visit );

visit ( x->data );

}

template <typename T, typename VST> //元素类型、操作器

void travIn\_R ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（递归版）

if ( !x ) return;

travIn\_R ( x->lc, visit );

visit ( x->data );

travIn\_R ( x->rc, visit );

}

template <typename T, typename VST> //元素类型、操作器

void travPre\_I1 ( BinNodePosi(T) x, VST& visit ) { //二叉树先序遍历算法（迭代版#1）

Stack<BinNodePosi(T)> S; //辅助栈

if ( x ) S.push ( x ); //根节点入栈

while ( !S.empty() ) { //在栈变空之前反复循环

x = S.pop(); visit ( x->data ); //弹出并访问当前节点，其非空孩子的入栈次序为先右后左

if ( HasRChild ( \*x ) ) S.push ( x->rc ); if ( HasLChild ( \*x ) ) S.push ( x->lc );

}

}

template <typename T, typename VST> //元素类型、操作器

static void visitAlongLeftBranch ( BinNodePosi(T) x, VST& visit, Stack<BinNodePosi(T)>& S ) {

while ( x ) {

visit ( x->data ); //访问当前节点

S.push ( x->rc ); //右孩子入栈暂存（可优化：通过判断，避免空的右孩子入栈）

x = x->lc; //沿左分支深入一层

}

}

template <typename T, typename VST> //元素类型、操作器

void travPre\_I2 ( BinNodePosi(T) x, VST& visit ) { //二叉树先序遍历算法（迭代版#2）

Stack<BinNodePosi(T)> S; //辅助栈

while ( true ) {

visitAlongLeftBranch ( x, visit, S ); //从当前节点出发，逐批访问

if ( S.empty() ) break; //直到栈空

x = S.pop(); //弹出下一批的起点

}

}

template <typename T> //从当前节点出发，沿左分支不断深入，直至没有左分支的节点

static void goAlongLeftBranch ( BinNodePosi(T) x, Stack<BinNodePosi(T)>& S ) {

while ( x ) { S.push ( x ); x = x->lc; } //当前节点入栈后随即向左侧分支深入，迭代直到无左孩子

}

template <typename T, typename VST> //元素类型、操作器

void travIn\_I1 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（迭代版#1）

Stack<BinNodePosi(T)> S; //辅助栈

while ( true ) {

goAlongLeftBranch ( x, S ); //从当前节点出发，逐批入栈

if ( S.empty() ) break; //直至所有节点处理完毕

x = S.pop(); visit ( x->data ); //弹出栈顶节点并访问之

x = x->rc; //转向右子树

}

}

template <typename T> BinNodePosi(T) BinNode<T>::succ() { //定位节点v的直接后继

BinNodePosi(T) s = this; //记录后继的临时变量

if ( rc ) { //若有右孩子，则直接后继必在右子树中，具体地就是

s = rc; //右子树中

while ( HasLChild ( \*s ) ) s = s->lc; //最靠左（最小）的节点

} else { //否则，直接后继应是“将当前节点包含于其左子树中的最低祖先”，具体地就是

while ( IsRChild ( \*s ) ) s = s->parent; //逆向地沿右向分支，不断朝左上方移动

s = s->parent; //最后再朝右上方移动一步，即抵达直接后继（如果存在）

}

return s;

}

template <typename T, typename VST> //元素类型、操作器

void travIn\_I2 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（迭代版#2）

Stack<BinNodePosi(T)> S; //辅助栈

while ( true )

if ( x ) {

S.push ( x ); //根节点进栈

x = x->lc; //深入遍历左子树

} else if ( !S.empty() ) {

x = S.pop(); //尚未访问的最低祖先节点退栈

visit ( x->data ); //访问该祖先节点

x = x->rc; //遍历祖先的右子树

} else

break; //遍历完成

}

template <typename T, typename VST> //元素类型、操作器

void travIn\_I3 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（迭代版#3，无需辅助栈）

bool backtrack = false; //前一步是否刚从右子树回溯——省去栈，仅O(1)辅助空间

while ( true )

if ( !backtrack && HasLChild ( \*x ) ) //若有左子树且不是刚刚回溯，则

x = x->lc; //深入遍历左子树

else { //否则——无左子树或刚刚回溯（相当于无左子树）

visit ( x->data ); //访问该节点

if ( HasRChild ( \*x ) ) { //若其右子树非空，则

x = x->rc; //深入右子树继续遍历

backtrack = false; //并关闭回溯标志

} else { //若右子树空，则

if ( ! ( x = x->succ() ) ) break; //回溯（含抵达末节点时的退出返回）

backtrack = true; //并设置回溯标志

}

}

}

template <typename T, typename VST> //元素类型、操作器

void travIn\_I4 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历（迭代版#4，无需栈或标志位）

while ( true )

if ( HasLChild ( \*x ) ) //若有左子树，则

x = x->lc; //深入遍历左子树

else { //否则

visit ( x->data ); //访问当前节点，并

while ( !HasRChild ( \*x ) ) //不断地在无右分支处

if ( ! ( x = x->succ() ) ) return; //回溯至直接后继（在没有后继的末节点处，直接退出）

else visit ( x->data ); //访问新的当前节点

x = x->rc; //（直至有右分支处）转向非空的右子树

}

}

template <typename T> //在以S栈顶节点为根的子树中，找到最高左侧可见叶节点

static void gotoHLVFL ( Stack<BinNodePosi(T)>& S ) { //沿途所遇节点依次入栈

while ( BinNodePosi(T) x = S.top() ) //自顶而下，反复检查当前节点（即栈顶）

if ( HasLChild ( \*x ) ) { //尽可能向左

if ( HasRChild ( \*x ) ) S.push ( x->rc ); //若有右孩子，优先入栈

S.push ( x->lc ); //然后才转至左孩子

} else //实不得已

S.push ( x->rc ); //才向右

S.pop(); //返回之前，弹出栈顶的空节点

}

template <typename T, typename VST>

void travPost\_I ( BinNodePosi(T) x, VST& visit ) { //二叉树的后序遍历（迭代版）

Stack<BinNodePosi(T)> S; //辅助栈

if ( x ) S.push ( x ); //根节点入栈

while ( !S.empty() ) {

if ( S.top() != x->parent ) //若栈顶非当前节点之父（则必为其右兄），此时需

gotoHLVFL ( S ); //在以其右兄为根之子树中，找到HLVFL（相当于递归深入其中）

x = S.pop(); visit ( x->data ); //弹出栈顶（即前一节点之后继），并访问之

}

}

template <typename T> template <typename VST> //元素类型、操作器

void BinNode<T>::travLevel ( VST& visit ) { //二叉树层次遍历算法

Queue<BinNodePosi(T)> Q; //辅助队列

Q.enqueue ( this ); //根节点入队

while ( !Q.empty() ) { //在队列再次变空之前，反复迭代

BinNodePosi(T) x = Q.dequeue(); visit ( x->data ); //取出队首节点并访问之

if ( HasLChild ( \*x ) ) Q.enqueue ( x->lc ); //左孩子入队

if ( HasRChild ( \*x ) ) Q.enqueue ( x->rc ); //右孩子入队

}

}