/\*树与二叉树的应用\*/

//二叉排序树非递归查找算法

typedef struct BiNode {

ElemType data;

int countOfChild;//保存以该结点为根的子树的结点个数

struct BiNode \*lchild, \*rchild;

} BSTNode, \*BiTree;

BSTNode \*BST\_Search(BiTree T, ElemType key, BSTNode \*&p) {

p = NULL;

while (T != NULL && key != T->data) {

p = T;

if (key < T->data) T = T->lchild;

else T = T->rchild;

}

return T;

}

//二叉排序树的插入

int BST\_Insert(BiTree &T, KeyType k) {

if (T == NULL) {

T = (BiTree) malloc(sizeof(BSTNode));

T->data = k;

T->lchild = T->rchild = NULL;

return 1;

} else if (k == T->data)

return 0;

else if (k < T->data)

return BST\_Insert(T->lchild, k);

else

return BST\_Insert(T->rchild, k);

}

//二叉树的构造

void Creat\_BST(BiTree &T, KeyType str[], int n) {

T = NULL;

int i = 0;

while (i < n) {

BST\_Insert(T, str[i]);

i++;

}

}

//判定给定的二叉树是否是二叉排序树

//进行中序遍历，如果能保持增序，则是二叉排序树

KeyType predt = -32767;

int JudgeBST(BiTree bt) {

int b1, b2;

if (bt == NULL) return 1;

else {

b1 = JudgeBST(bt->lchild);

if (b1 == 0 || predt >= bt->data)//若做子树返回值为0或者前驱大于当前结点

return 0;//则不是二叉排序树

predt = bt->data;

b2 = JudgeBST(bt->rchild);

return b2;

}

}

//求出给定结点在给定二叉排序树中的层次

//用n来保存查找层次，每查找一次就+1

int level(BiTree bt, BSTNode \*p) {

int n = 0;

BiTree t = bt;

if (bt != NULL) {

n++;

while (t->data != bt->data) {

if (t->data < p->data)

t = t->lchild;

else

t = t->rchild;

n++;

}

}

return n;

}

//利用二叉树遍历的思想判断二叉树是否是平衡二叉树

/\* 设置标记balance，1:平衡；0:不平衡

\* h为二叉树bt的高度

\* 1)若bt为空，则h=0,balance=1

\* 2)若bt仅有根节点，则h=1,balance=1

\* 3)否则，对左右子树递归，bt高度为最高子树+1。若左右子树高度差大于1，则balance=0;若高度差小于1，且左右子树都平衡，则balance=1,否则balance=0

\* \*/

void Judge\_AVL(BiTree bt, int &balance, int &h) {

int bl, br, hl, hr;//左右子树的平衡标志和高度

if (bt == NULL) {

h = 0;

balance = 1;

} else if (bt->lchild == NULL && bt->rchild == NULL) {//仅有根结点

h = 1;

balance = 1;

} else {

Judge\_AVL(bt->lchild, bl, hl);

Judge\_AVL(bt->rchild, br, hr);

h = (hl > hr ? hl : hr) + 1;

if (abs(hl, hr) < 2) balance = bl & br;

else balance = 0;

}

}

//求二叉排序树中最大和最小关键字

KeyType MinKey(BSTNode \*bt) {

while (bt->lchild != NULL)

bt = bt->lchild;

return bt->data;

}

KeyType MaxKey(BSTNode \*bt) {

while (bt->rchild != NULL)

bt = bt->rchild;

return bt->data;

}

//从大到小输出二叉排序树中所有值不小于k的关键字

//为了从大到小输出，先遍历右子树，再访问根结点，再遍历左子树

void OutPut(BSTNode \*bt, KeyType k) {

if (bt == NULL)

return;

if (bt->rchild != NULL)

OutPut(bt->rchild, k);

if (bt->data >= k)

printf(" %d", bt->data);

if (bt->lchild != NULL)

OutPut(bt->lchild, k);

}

//在一棵有n个结点的随机建立的二叉排序树查找第k小的元素

/\*若t->lchild为空

\* 1)若t->rchild非空，且k==1，则\*t即为第k小的元素

\* 2)若t->rchild非空，且k!=1，则第K小的元素必定在\*t的右子树

\*若t->lchild非空

\* 1)t->lchild->count==k-1,则\*t即为第k小的元素

\* 2)t->lchild->count>k-1,则第k小的元素必定在\*t的左子树，继续到左子树中查找

\* 3)t->lchild->count<k-1,则第k小的元素必定在\*t的右子树，继续道右子树中查找，寻找第k-(t->lchild->count+1)小的元素

\* \*/

BSTNode \*Search\_Small(BSTNode \*t, int k) {

if (k < 1 || k > t->countOfChild) return NULL;

if (t->lchild == NULL) {

if (k == 1) return t;

else return Search\_Small(t->rchild, k - 1);

} else {

if (t->lchild->countOfChild == k - 1) return t;

if (t->lchild->countOfChild > k - 1) return Search\_Small(t->lchild, k);

if (t->lchild->countOfChild < k - 1)

return Search\_Small(t->rchild, k - (t->lchild->countOfChild + 1));

}

}

//后序遍历的非递归算法

typedef struct {

BSTNode \*p;

int rvisited;//1:代表p所指向的结点的右结点已经被访问过

} SNode;//栈中结点定义

typedef struct {

SNode Elem[maxsize];

int top;

} SqStack;//栈结构体

void PostOrder2(BiTree T) {

SNode sn;

BSTNode \*pt = T;

InitStack(S);

while (T) { //从根结点开始，往左下方走，将路径上的每一个结点入栈

Push(pt, 0); //push到栈中：一是结点指针，另一个是其右孩子是否被访问过

pt = pt->lchild;

}

while (!S.IsEmpty()) {

sn = S.getTop();

if (sn.p->rchild != NULL || sn.rvisited) {

Pop(S, pt);

visit(pt);

} else {//若右子树存在且rvisited==0，处理其右子树

sn.rvisited = 1;

pt = sn.p->rchild;

while (pt != NULL) {//往左下方走到尽头，将路径上所有元素入栈

Push(S, pt, 0);

pt = pt->rchild;

}

}

}

}