```
/*打印建议小 5 字体，10 磅行间距*/
typedef struct BiTNode {
	ElemType data;
	struct BiTNode *lchild, rchild;
} BiTNode, *BiTree;

//先序遍历-递归
void PreOrder(BiTree T) {
	if (T != NULL) {
		visit(T);
		PreOrder(T->lchild);
		PreOrder(T->rchild);
	}
}

//中序遍历-递归
void InOrder(BiTree T) {
	if (T != NULL) {
		InOrder(T->lchild);
		visit(T);
		InOrder(T->rchild);
	}
}

//后序遍历-递归
void PostOrder(BiTree T) {
	if (T != NULL{}) {
		PostOrder(T->lchild);
		PostOrder(T->rchild);
		visit(T);
	}
}

//中序遍历-非递归
void InOrder2(BiTree T) {
	InitStack(S);
	BiTree p = T;//p 是遍历指针
	while (p || IsEmpty(S)) {
		if (p) {
			Push(S, p);
			p = p->lchild;
		} else {
			Pop(S, p);
			visit(p);
			p = p->rchild;
		}
	}
}

//层次遍历
void LevelOrder(BiTree T) {
	InitQueue(Q);
	BiTree p;
	EnQueue(Q, T);
	while (!IsEmpty(Q)) {
		DeQueue(Q, p);
		visit(p);
		if (p->lchild != NULL)
			EnQueue(Q, p->lchild);
		if (p->rchild != NULL)
			EnQueue(Q, p->rchild);
	}
}

//线索二叉树结构体
typedef struct ThreadNode {
	ElemType data;
	struct ThreadNode *lchild, *rchild;
	int ltag, rtag;//左右线索标志
```

```
} ThreadNode, *ThreadTree;

//通过中序遍历对二叉树线索化-递归
void InThread(ThreadTree &p, ThreadTree &pre) {
    if (p != NULL) {
        InThread(p->lchild, pre);//递归，线索化左子树
        if (p->lchild == NULL) {//左子树为空，建立前驱线索
            p->lchild = pre;
            p->ltag = 1;
        }
        if (pre != NULL && pre->rchild == NULL) {
            pre->rchild = p;//建立前驱结点的后继线索
            pre->rtag = 1;
        }
        pre = p;
        InThread(p->rchild, pre);
    }
}

void CreateInThread(ThreadTree T) {
    ThreadTree pre = NULL;
    if (T != NULL) {
        InThread(T, pre);
        pre->rchild = NULL;//处理遍历的最后一个结点
        pre->rtag = 1;
    }
}

//中序线索二叉树中中序序列下的第一个结点
ThreadNode *Firstnode(ThreadNode *p) {
    while (p->ltag == 0) p = p->lchild;
    return p;
}

//中序线索二叉树中的结点 p 在中序序列下的后继结点
ThreadNode *Nextnode(ThreadNode *p) {
    if (p->rtag == 0) return Firstnode(p->rchild);
    else return p->rchild;
}

//利用上面两个算法，写出不含头结点的中序线索二叉树的中序遍历算法
void InorderOfThread(ThreadNode *T) {
    for (ThreadNode *p = Firstnode(T); p != NULL; p = Nextnode(p))
        visit(p);
}

//后序遍历的非递归算法
//使用辅助指针 r，指向最近访问过的点，也可以在结点中增加一个标识域，记录是否被访问过
/*当访问一个结点*p 的时候，栈中的结点恰好是*p 的所有祖先。从栈底到栈顶结点再加上*p，刚好构成从根结点到*p 的一条路径。可以用于求根结点到某结点的路径，求两个结点的最近公共祖先等*/
void PostOrder2(BiTree T) {
    InitStack(S);
    BiTree p = T;
    BiTree r = NULL;
    while (p || !IsEmpty(S)) {
        if (p) {
            push(S, p);
            p = p->lchild;
        } else {
            GetTop(S, p);
            if (p->rchild && p->rchild != r) {//如果右子树存在且未被访问过
                p = p->rchild;
                push(S, p);
                p = p->lchild;
            } else {
                pop(S, p);
                visit(p->data);
                r = p;//记录最近访问过的结点
                p = NULL;//访问完毕后，重置 p 指针
```

```
                }
            }
        }
}

/*自下而上，自右向左的层次遍历*/
/*利用原有层次遍历算法，将遍历的序列入栈，再出栈即可*/
void InvertLevel(BiTree bt) {
    Stack s;
    Queue Q;
    BiTree p;
    if (bt != NULL) {
        InitStack(s);
        InitQueue(Q);
        EnQueue(Q, bt);
        while (!IsEmpty(Q)) {
            DeQueue(Q, p);
            Push(s, p);
            if (p->lchild)
                EnQueue(Q, p->lchild);
            if (p->rchild)
                EnQueue(Q, p->rchild);
        }
        while (!IsEmpty(s)) {
            Pop(s, p);
            visit(p->data);
        }
    }
}

/*用非递归算法求二叉树高度*/
/*层次遍历，设置变量 level 记录当前结点的层数，设置变量 last 指向当前层最右结点，每次层次遍历出队时，与 last 指针比较，若两者相等，则 level+1，并让 last 指向下一层最右结点*/
int Btdepth(BiTree T) {
    if (!T) return 0;//树空，高度为 0
    int front = -1, rear = -1;
    int last = 0, level = 0;
    BiTree Q[maxsize];
    Q[++rear] = T;
    BiTree p;
    while (front < rear) {
        p = Q[++front];
        if (p->lchild) Q[++rear] = p->lchild;
        if (p->rchild) Q[++rear] = p->rchild;
        if (front == last) {//处理该层最右结点
            level++;
            last = rear;//last 指向下一层
        }
    }
    return level;
}//可以用栈模拟递归 ,在结构体里增加 high 变量 ,p->high=max(p->lchild->high,p->rchild->high)+1
/*用递归算法求二叉树高度*/
int Btdepth2(BiTree T) {
    if (T == NULL) return 0;
    ldep = Btdepth2(T->lchild);
    rdep = Btdepth2(T->rchild);
    if (ldep > rdep) return ldep + 1;
    else return rdep + 1;
}

/*一棵二叉树各结点的值互不相同，先序遍历后序遍历分别存放在 A[],B[]，建立该二叉树的二叉链表*/
/*根据先序序列确定树的根结点；根据根结点在中序序列中划分出二叉树的做右子树包含那些结点，然后再根据左右子树结点再先序序列中的位置确定子树根结点*/
BiTree PreInCreat(ElemType A[], ElemType B[], int l1, int h1, int l2, int h2) {
    //l1,h1 为先序的第一个和最后一个结点下标，l2,h2 为中序的第一个和最后一个结点下标
    //初始调用时，l1=l2=1,h1=h2=n
    BiTree root;
    root = (BiTNode *) malloc(sizeof(BiTNode));
    root->data = A[l1];//根结点
```

```
    for (i = l2; B[i] != root->data; i++);//根结点再中序序列中的划分
    llen = i - l2;//左子树长度
    rlen = h2 - i;//右子树长度
    if (llen)
        root->lchild = PreInCreat(A, B, l1 + 1, l1 + llen, l2, l2 + llen - 1);
    else
        root->lchild = NULL;
    if (rlen)
        root->rchild = PreInCreat(A, B, h1 - rlen + 1, h1, h2 - rlen + 1, h2);
    else
        root->rchild = NULL;
    return root;
}

/*判定二叉树是否是完全二叉树*/
/*采用层次遍历，将所有结点加入队列（包括空系结点）当遇到空结点时，查看其后是否有非空结点，若有，则不是完全二叉树*/
bool IsComplete(BiTree T) {
    BiTree p;
    InitQueue(Q);
    if (!T) return 1;//空树为满二叉树
    EnQueue(Q, T);
    while (!IsEmpty(Q)) {
        DeQueue(Q, p);
        if (p) {
            EnQueue(Q, p->lchild);
            EnQueue(Q, p->rchild);
        } else
            while (!IsEmpty(Q)) {
                DeQueue(Q, p);
                if (p) return 0;//结点非空，则二叉树为非完全二叉树
            }
    }
    return 1;
}

/*计算一棵给定二叉树的所有双分支结点的个数*/
int DSonNodes(BiTree b) {
    if (b == NULL) return 0;
    else if (b->lchild != NULL && b->rchild != NULL)//双分支结点
        return DSonNodes(b->lchild) + DSonNodes(b->rchild) + 1;
    else
        return DSonNodes(b->lchild) + DSonNodes(b->rchild);
}//也可以设置全局变量 NUM，遍历并判断，且维护 NUM

//互换 b 的左右子树
void swap(BiTree b) {
    if (b) {
        swap(b->lchild);
        swap(b->rchild);
        temp = b->lchild;
        b->lchild = b->rchild;
        b->rchild = temp;
    }
}

/*求先序遍历第 k 个结点的值*/
int i = 1;

ElemType PreNode(BiTree b, int k) {
    if (b == NULL) return '#';
    if (i == k) return b->data;
    i++;
    ch = PreNode(b->lchild, k);
    if (ch != '#') return ch;
    ch = PreNode(b->rchild, k);
    return ch;
}
```

```
/*对于树中每一个元素值为 x 的结点，删去以它为根的子树，并释放空间*/
/*删除 x 结点，意味着应该将其父结点的左（右）子女指针置空，用层次遍历易于找到父节点*/
void DeleteXTree(BiTree bt) {
    if (bt) {
        DeleteXTree(bt->lchild);
        DeleteXTree(bt->rchild);
        free(bt);
    }
}

void Search(BiTree bt, ElemType x) {
    BiTree Q[];//队列
    BiTree p;
    if (bt) {
        if (bt->data == x) {//若根结点值为 x，则删除整棵树
            DeleteXTree(bt);
            exit(0);
        }
        InitQueue(Q);
        EnQueue(Q, bt);
        while (!IsEmpty(Q)) {
            DeQueue(Q, p);
            if (p->lchild)
                if (p->lchild->data == x) {
                    DeleteXTree(p->lchild);
                    p->lchild = NULL;
                } else
                    EnQueue(Q, p->lchild);
            if (p->rchild)
                if (p->rchild->data == x) {
                    DeleteXTree(p->rchild);
                    p->rchild = NULL;
                } else
                    EnQueue(Q, p->rchild);
        }
    }
}

/*查找值为 x 的结点，打印其所有祖先，假设值为 x 的结点不多于一个*/
typedef struct {
    BiTree t;
    int tag;
} stack;//tag=0 表示左子女被访问，tag=1 表示右子女被访问
void SearchX(BiTree bt, ElemType x) {
    stack s[];
    top = 0;
    while (bt != NULL || top > 0) {
        while (bt != NULL && bt->data != x) {//结点入栈
            s[++top].t = bt;
            s[top].tag = 0;
            bt = bt->lchild;//沿左分支向下
        }
        if (bt->data == x) {
            for (i = 1; i <= top; i++)
                printf("%d", s[i].t->data);
            exit(1);
        }
        while (top != 0 && s[top].tag == 1)
            top--;//退栈所有已遍历右子树的结点
        if (top != 0) {
            s[top].tag = 1;
            bt = s[top].t->rchild;
        }
    }
}

/*找出二叉树中任意 p,q 两个结点的最近公共祖先 r
 * 采用后序非递归算法，栈中存放着二叉树结点的指针，访问到某结点时，栈中元素为其祖先。假设 p 在
q 左边，先遍历到 p，将栈复制到另一个辅助栈，继续遍历到 q，然后将两个栈匹配，第一个匹配的元素即
```

```
为最近公共祖先*/
typedef struct {
    BiTree t;
    int tag;//tag=0 表示左子女已经访问，tag=1 表示右子女已经访问
} stack;
stack s[], s1[];

BiTree Ancestor(BiTree Root, BiTNode *p, BiTNode *q) {
    top = 0;
    BiTree bt = Root;
    while (bt != NULL || top > 0) {
        while (bt != NULL && bt != p && bt != q)
            while (bt != NULL) {
                s[++top].t = bt;
                s[top].tag = 0;
                bt = bt->lchild;
            }
        while (top != 0 && s[top].tag == 1) {
            //假定 p 在 q 的左侧，遇到 p 时，栈中元素均为 p 的祖先
            if (s[top].t == p) {
                for (i = 1; i <= top; i++)
                    s1[i] = s[i];
                top1 = top;
            }
            if (s[top].t == q)
                for (i = top; i > 0; i--)
                    for (j = top1; j > 0; j--)
                        if (s1[j].t == s[i].t)
                            return s[i].t;
            /*if (s[top].t == q){
                i=j=min(top1,top);
                for(;i>0;i--,j--)
                    if(s1[j].t==s[i].t)
                        return s[i].t;
            }效率高一些，因为找到一个公共祖先，那么两个栈内元素相同，必然 i=j*/
            top--;//退栈
        }
        if (top != 0) {
            s[top].tag = 1;
            bt = s[top].t->rchild;
        }
    }
    return NULL;
}

/*求非空二叉树宽度*/
typedef struct {
    BiTree data[MaxSize];
    int level[MaxSize];
    int front, rear;
} Qu;

int BTWidth(BiTree b) {
    BiTree p;
    int k, max, i, n;
    Qu.front = Qu.rear = -1;//队列为空
    Qu.rear++;
    Qu.data[Qu.rear] = b;//根结点入队
    Qu.level[Qu.rear] = 1;//根结点层次为 1
    while (Qu.front < Qu.rear) {
        Qu.front++;
        p = Qu.data[Qu.front];
        k = Qu.level[Qu.front];//出队结点的层次
        if (p->lchild != NULL) {
            Qu.rear++;
            Qu.data[Qu.rear] = p->lchild;
            Qu.level[Qu.rear] = k + 1;
        }
        if (p->rchild != NULL) {
```

```
                Qu.rear++;
                Qu.data[Qu.rear] = p->rchild;
                Qu.level[Qu.rear] = k + 1;
            }
        }
        max = 0;
        i = 0;
        k = 1;
        while (i <= Qu.rear) {
            n = 0;//n 统计第 k 层结点个数
            while (i <= Qu.rear && Qu.level[i] == k) {
                n++;
                i++;
            }
            k = Qu.level[i];
            if (n > max) max = n;
        }
        return max;
}

/*设有一棵满二叉树，结点值均不同，已知先序序列为 pre，求后续序列 post*/
/*将 pre[l1,h1]转化为后序 post[l2,h2],递归模型
 * 当 h1<l1 时：f(pre,l1,h1,post,l2,h2) 无动作
 * 其他情况：
 * f(pre,l1,h1,post,l2,h2) post[h2]=pre[l1]
 * 取中间位置 half=(h1-l2)/2
 * 将 pre[l1+1,l1+half]左子树转化为 post[l2,l2+half-1]
 * 即 f(pre,l1+1,l1+half,post,l2,l2+half-1)
 * 将 pre[l1+half+1,h1],右子树转化为 post[l2+half,h2-1]
 * 即 f(pre,l1+half+1,h1,post,l2+half,h2-1)
 * 其中，post[h2]=pre[l1]表示后序序列的最后一个结点(根结点)等于先序序列的第一个结点(根结点)
*/
void PreToPost(ElemType pre[], int l1, int h1, ElemType post[], int l2, int h2) {
        int half;
        if (h1 >= l1) {
            post[h2] = pre[l1];
            half = (h1 - l1) / 2;
            PreToPost(pre, l1 + 1, l1 + half, post, l2, l2 + half - 1);//转换左子树
            PreToPost(pre, l1 + half + 1, h1, post, l2 + half, h2 - 1);//转换右子树
        }
}

/*设计一个算法将二叉树的叶结点从左到右连成一个单链表，表头指针为 head，叶结点的右指针存放单链表指针*/
/*采用中序递归遍历*/
LinkedList head, pre = NULL;

LinkedList InOrder(BiTree bt) {
        if (bt) {
            InOrder(bt->lchild);
            if (bt->lchild == NULL && bt->rchild == NULL)
                if (pre == NULL) {
                    head = bt;
                    pre = bt;
                } else {
                    pre->rchild = bt;
                    pre = bt;
                }
            InOrder(bt->rchild);
            pre->rchild = NULL;
        }
        return head;
}

/*判断两个树 T1,T2 是否相似*/
/* 1)f(T1,T2)=1;若 T1==T2==NULL
 * 2)f(T1,T2)=0;若 T1 和 T2 之一为 NULL,另一个不是 NULL
 * 3)f(T1,T2)=f(T1->lchild,T2->lchild)&&f(T1->rchild,T2->rchild)若 T1T2 均不为 NULL*/
int similar(BiTree t1, BiTree t2) {
```

```
    int leftS, rightS;
    if (t1 == NULL && t2 == NULL)//两树皆空
        return 1;
    else if (t1 == NULL || t2 == NULL)//只有一树为空
        return 0;
    else {
        leftS = similar(t1->lchild, t2->lchild);
        rightS = similar(t1->rchild, t2->rchild);
        return leftS && rightS;
    }
}

/*写出在中序线索二叉树里查找指定结点在后序的前驱结点的算法*/
/*  在后序序列，
 *  若结点 p 有右子女，则右子女是其前驱
 *  若无右子女而有左子女，则左子女是其前驱
 *  若结点 p 左右子女均无，设其中序左线索指向某祖先结点 f(p 是 f 右子树中按中序遍历的第一个结点)
 *  --若 f 有左子女，则其左子女是结点 p 在后序下的前驱
 *  --若 f 无左子女，则顺其前驱找双亲的双亲，一直找到双亲有左子女(这时左子女是 p 的前驱)
 *  若 p 是中序遍历的第一个结点，则 p 在中序和后序下均无前驱*/
ThreadTree InPostPre(ThreadTree t, ThreadTree p) {
    //在中序线索二叉树 t 中，求指定结点 p 在后序下的前驱结点 q
    ThreadTree q;
    if (p->rtag == 0)//若 p 有右子女，则右子女是其后序前驱
        q = p->rchild;
    else if (p->ltag == 0)//若 p 只有左子女，左子女是其后序前驱
        q = p->lchild;
    else if (p->lchild == NULL)//p 是中序序列的第一结点，无后序前驱
        q = NULL;
    else {//顺左线索向上找 p 的祖先，若存在，再找祖先的左子女
        while (p->ltag == 1 && p->lchild != NULL)
            p = p->lchild;
        if (p->ltag == 0)//p 结点的祖先的左子女是其后序前驱
            q = p->lchild;
        else
            q = NULL;//仅有单支树(p 是叶子),已到根结点，p 无后序前驱
    }
    return q;
}

/*求 WPL(带权路径长度)*/
/*基于先序递归遍历
 *  用一个 static 变量记录 wpl，把每个结点的深度作为递归函数的参数传递
 *  若该结点是叶结点，那么 wpl 加上该结点的深度与权值之积
 *  若该结点非叶结点，那么若左子树不为空，递归调用；若右子树不为空，递归调用。深度参数均为本届
点的深度参数+1*/
typedef struct BiTNode {
    int weight;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;

int WPL(BiTree root) {
    return wpl_PreOrder(root, 0);
}

int wpl_PreOrder(BiTree root, int deep) {
    static int wpl = 0;
    if (root->lchild == NULL && root->rchild == NULL)//若为叶结点
        wpl += deep * root->weight;
    if (root->lchild != NULL)//若左子树不为空，对左子树递归
        wpl_PreOrder(root->lchild, deep + 1);
    if (root->rchild != NULL)//若右不为空，递归
        wpl_PreOrder(root->rchild, deep + 1);
    return wpl;
}
/*基于层次遍历
 *  当遍历到叶结点，累积 wpl
 *  当遍历到非叶结点，把该结点的子树加入队列
 *  当某结点为该层最后一个结点时，层数+1*/
```

```
#define MaxSize 100

int wpl_LevelOrder(BiTree root) {
    BiTree q[MaxSize];
    int front, rear;
    front = rear = 0;
    int wpl = 0, deep = 0;
    BiTree lastNode;//记录当层最后一个结点
    BiTree newlastNode;//下一层的最后一个结点
    lastNode = root;//初始化为根结点
    newlastNode = NULL;
    q[rear++] = root;
    while (front != rear) {
        BiTree t = q[front++];
        if (t->lchild == NULL && t->rchild == NULL)
            wpl += deep * t->weight;
        if (t->lchild != NULL) {
            q[rear++] = t->lchild;
            newlastNode = t->lchild;
        }
        if (t->rchild != NULL) {
            q[rear++] = t->rchild;
            newlastNode = t->rchild;
        }
        if (t == lastNode) {//若该结点为本层最后一个结点，更新 lastNode
            lastNode = newlastNode;
            deep += 1;
        }
    }
    return wpl;
}
```

```
/*树和森林*/
//双亲表示法
#define MAX_TREE_SIZE 100
typedef struct {
    ElemType data;
    int parent;
} PTNode;
typedef struct {
    PTNode nodes[MAX_TREE_SIZE];
    int n;
} PTree;

//孩子兄弟表示法
typedef struct CSNode {
    ElemType data;
    struct CSNode *firstchild, *nextsibling;
} CSNode, *CSTree;

//并查集
#define SIZE 100
int UFSets[SIZE];

//并查集的初始化操作（s 即为并查集）
void Initial(int S[]) {
    for (int i = 0; i < size; ++i) {
        s[i] = -1;
    }
}

//Find 操作（函数在并查集 S 中查找并返回包含元素 x 的树的节点
int Find(int S[], int x) {
    while (S[x] >= 0)
        x = S[x];
    return x;
}

//Union 操作（函数要求两个不相交的子集）
void Union(int S[], int Root1, int Root2) {
    //要求 Root1 和 Root2 是不同的，且表示子集的名字
    S[Root2] = Root1;
}

//求以孩子兄弟表示法存储的树的高度
int Height(CSTree bt) {
    int hc, hs;
    if (bt == NULL)
        return 0;
    else {
        hc = Height(bt->firstchild);
        hs = Height(bt->nextsibling);
        if (hc + 1 > hs)
            return hc + 1;
        else
            return hs;
    }
}

//求以孩子兄弟表示法存储的森林的叶子结点数
//以孩子兄弟表示法存储时，若结点没有左子树，则为叶结点
typedef struct node {
    ElemType data;   //数据域
    struct node *fch, *nsilb;
} *Tree;

int Leaves(Tree t) {
    if (t == NULL)
        return 0;
    if (t->fch == NULL)
        return 1 + Leaves(t->nsilb);
```

```
    else
        return Leaves(t->fch) + Leaves(t->nsilb);
}

//已知一棵树的层次序列以及每个结点的度，构造此树的孩子-兄弟链表
//pointer[]，存储新建树的各结点的地址
typedef struct CSNode {
    ElemType data;
    struct CSNode *firstchild, *nextsibling;
} CSNode, *CSTree;
#define maxNodes 15

void createCSTree_Degree(CSTree &t, DataType e[], int degree[], int n) {
    //根据树的层次遍历序列 e[]和各个结点的度 degree[]构造，n 是结点个数
    CSNode *pointer = new CSNode[maxNodes];
    int i, j, d, k = 0;
    for (i = 0; i < n; ++i) {
        pointer[i] = new CSNode;
        pointer[i].data = e[i];
        pointer[i].firstchild = pointer[i].nextsibling = NULL;
    }
    for (i = 0; i < n; ++i) {
        d = degree[i];
        if (d) {
            k++;      //k 为子女结点序号
            pointer[i].firstchild = pointer[k];//建立 i 与子女 k 的链接
            for (j = 2; j <= d; ++j)
                pointer[k - 1].nextsibling = pointer[k];
        }
    }
    T = pointer[0];
    delete[] pointer;
}
```

```
/*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;
            }
        }
    }
}
```