数据结构算法汇总

目录

专题一 线性表.....1
1. 线性表基本操作.....1
2. 顺序表.....2
3. 链表.....2
4. 线性表应用.....2
专题二 栈.....6
1. 栈基本操作.....6
2. 顺序栈.....6
3. 链栈.....6
专题三 队列.....7
1. 队列基本操作.....7
2. 顺序队列.....7
3. 链式队列.....7
专题四 树.....7
1. 树.....7
2. 二叉树.....8
3. 线索二叉树.....10
4. 二叉排序树.....13
5. 二叉树应用.....14
专题五 广义表.....25
专题六 图.....26
1. 图的存储结构.....26
2. 邻接表相关.....26
专题七 排序与查找.....27

专题一 线性表

1. 线性表基本操作

```
InitList(&L)                //初始化表
Length(L)                   //求表长
LocateElem(L, e)            //按值查找操作
GetElem(L, i)               //按位查找操作
ListInsert(&L, i, e)        //插入操作
ListDelete(&L, i, &e)       //删除操作
```

```
PrintList(L)                    //输出操作
Empty(L)                        //判空操作
DestroyList(&L)                 //销毁操作
```

2. 顺序表

```
//静态分配
#define MaxSize 50
typedef struct{
    ElemType data[MaxSize];
    int length;
}SqList;
//动态分配
#define InitSize 100
typedef struct{
    ElemType *data;
    int MaxSize, length;
}SeqList;
//动态分配语句：
L.data = (ElemType*)malloc(sizeof(ElemType)*InitSize);
```

3. 链表

```
//单链表：
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
//双链表：
typedef struct DNode{
    ElemType data;
    struct DNode *prior, *next;
}DNode, *DLinklist;
//静态链表：
#define MaxSize 50
typedef struct{
    ElemType data;
    int next;
}SLinkList[MaxSize];
```

4. 线性表应用

1. 删除不带头结点单链表 L 中所有值为 x 结点

```
void Delete_x(LinkList &L, int x){
    LinkList p;                 //用于指向待删除结点
    if (L == NULL)              //递归出口
        return;
    if (L->data==x){            //若L所指结点值为x
        p = L;
        L = L->next;            //L指向下一结点
        free(p);
        Delete_x(L, x);
    }
    else Delete_x(L->next, x);
}
```

2.删除带头结点单链表 L 中所有值为 x 结点

法一:设置前驱指针，扫描

```
void Delete_x(LinkList &L, int x){
    LinkList pre, p, q;
    pre = L;p = L->next;
    while (p!=NULL){
        if (p->data == x){
            q = p;
            p = p->next;
            pre->next = p;
            free(q);
        }
        else {
            pre = p;
            p = p->next;
        }
    }
}
```

法二:尾插法建立单链表

```
void Delete_x1(LinkList &L, int x){
    LinkList p, r, q;
    p = L->next;r = L;
    while (p != NULL){
        if (p->data != x){
            r->next = p;
            r = p;
            p = p->next;
        }
        else {
            q = p;
            p = p->next;
            free(q);
```

```
            }
        }
        r->next = NULL;
    }
```

3.反向输出带头结点单链表 L 中所有结点值

```
    void R_Print(LinkList L){
        if (L->next != NULL){
            R_Print(L->next);
            cout << L->next->data << " ";
        }
    }
```

4.删除带头结点单链表 L 中最小值结点

```
    void Delete_min(LinkList &L){
        LinkList p, pre;        //p为工作指针，pre指向其前驱
        LinkList minP, minPre;  //minP指向候选最小值，minPre指向其前驱
        p = minP = L->next;
        pre = minPre = L;
        while (p!=NULL){
            if (p->data<minP->data){
                minP = p;
                minPre = pre;
            }
            pre = p;
            p = p->next;
        }
        minPre->next = minP->next;//删除最小值结点
        free(minP);
    }
```

5.就地逆置带头结点单链表 L

法一:头插法

```
    void Reverse_L(LinkList &L){
        LinkList p, q;
        p = L->next;
        L->next = NULL;
        while (p!=NULL){
            q = p;
            p = p->next;
            q->next = L->next;
            L->next = q;
        }
    }
```

法二:指针反转

```
    void Reverse_L(LinkList L){
        LinkList pre, p, r;
```

```
        p = L->next;
        r = p->next;
        p->next = NULL;
        while (r != NULL){
            pre = p;
            p = r;
            r = r->next;
            p->next = pre;  //指针反转
        }
        L->next = p;        //表头指向最后结点
    }
```

6.带头结点单链表 L 升序排列

```
    直接插入排序
    void sort(LinkList &L){
        LinkList pre, p, r;
        p = L->next;
        r = p->next;        //r指向p后继结点，保证不断链
        p->next = NULL;     //构造只含一个结点的链表
        p = r;
        while (p!=NULL){    //开始插入排序
            r = p->next;
            pre = L;
            //从表头开始扫描p的插入位置
            while (pre->next != NULL&&pre->next->data < p->data)
                pre = pre->next;
            p->next = pre->next;//将p结点插入pre后
            pre->next = p;
            p = r;
        }
    }
```

7.删除带头结点单链表 L 中介于给定的两个值之间元素

```
    void RangeDelete(LinkList &L,int min,int max){
        LinkList p, pre;    //p为工作指针，pre指向其前驱
        p = L->next;pre = L;
        while (p!=NULL){
            if (min < p->data&&p->data < max){
                pre->next = p->next;
                free(p);
                p = pre->next;
            }
            else{
                pre = p;
                p = p->next;
            }
```

```
        }
    }
```

专题二 栈

1. 栈基本操作

```
InitStack(&S)                //初始化操作
StackEmpty(S)                //判空操作
Push(&S, x)                  //进栈操作
Pop(&S, &x)                  //出栈操作
GetTop(S, &x)                //读栈顶元素
ClearStack(&S)               //销毁操作
```

2. 顺序栈

```
#define MaxSize 50
typedef struct{
    ElemType data[MaxSize];
    int top;
}SqStack;
//栈顶指针：初始时设置S.top = -1
//栈顶元素：S.data[S.top]
//进栈操作：栈不满时，栈顶指针先加1，再送值到栈顶元素
//出栈操作：栈非空时，先取栈顶元素值，再将栈顶指针减1
//栈空条件：S.top == -1
//栈满条件：S.top == MaxSize  -  1
//栈长：S.top + 1
```

3. 链栈

```
typedef struct SNode{
    ElemType data;
    struct SNode *next;
}SNode, *LiStack;
```

专题三 队列

1. 队列基本操作

```
InitQueue(&Q)                //初始化操作
QueueEmpty(Q)                //判空操作
EnQueue(&Q, x)               //入队操作
DeQueue(&Q, &x)              //出队操作
GetHead(Q, &x)               //读队头元素
```

2. 顺序队列

```
#define MaxSize 50
typedef struct{
    ElemType data[MaxSize];
    int front, rear;
}SqQueue;
//初始条件：Q.front = Q.rear = 0
//队空条件：Q.front == Q.rear
//进队操作：队不满时，++Q.rear；Q.data[Q.rear] = x;
//出队操作：队不空时，++Q.front；x = Q.data[Q.front];
```

3. 链式队列

```
typedef struct QNode{//链式队列结点
    ElemType data;
    struct QNode *next;
}QNode;
typedef struct{ //链式队列
    QNode *front, *rear;
}LiQueue;
//当 Q.front == NULL 且 Q.rear == NULL 时，链式队列为空。
```

专题四 树

1. 树

1. 树存储结构

1.树双亲存储结构

```
#define MAX_TREE_SIZE 100
```

```
typedef struct{
    ElemType data;
    int parent;
}PTNode;
typedef struct{
    PTNode nodes[MAX_TREE_SIZE];
    int n;
}PTree;
```

2.树孩子兄弟存储结构

```
typedef struct CSNode{
    ElemType data;
    struct CSNode *firstchild,*nextsibling;
}CSNode,*CSTree;
```

2. 二叉树

1. 二叉树链式存储结构

```
#define ElemType int
#define MaxSize 1000
typedef struct BiTNode{
    ElemType data;                      //数据域
    struct BiTNode *lchild, *rchild;    //左、右孩子指针
}BiTNode,*BiTree;
```

2. 二叉树的遍历

1. 递归先序遍历

```
void PreOrder(BiTree T){
    if (T != NULL){
        visit(T);               //访问结点
        PreOrder(T->lchild);    //递归遍历左子树
        PreOrder(T->rchild);    //递归遍历右子树
    }
}
```

2. 非递归先序遍历

```
void PreOrder(BiTree T){
    InitStack(S); BiTree p = T;     //初始化栈；p是遍历指针
    if (p != NULL)                  //p不空继续执行
    {
        Push(S, p);                 //指针进栈
        while (!IsEmpty(S))         //栈不空时循环
        {
            Pop(S, p);              //指针退栈
            visit(p);               //访问结点
            if (p->rchild != NULL)  //右子树不为空
                Push(S,p->rchild);  //右子树进栈
```

```
            if (p->lchild != NULL)  //左子树不为空
                Push(S,p->lchild);  //左子树进栈
        }
    }
}
```

3.递归中序遍历

```
void InOrder(BiTree T){
    if (T != NULL){
        PreOrder(T->lchild);    //递归遍历左子树
        visit(T);               //访问结点
        PreOrder(T->rchild);    //递归遍历右子树
    }
}
```

4.非递归中序遍历

```
void InOrder(BiTree T){
    //二叉树中序遍历的非递归算法，算法需要借助一个栈
    InitStack(S); BiTree p = T;     //初始化栈；p是遍历指针
    while (p || !IsEmpty(S)){       //栈不空或p不空时循环
        if (p){                     //根指针进栈，遍历左子树
            Push(S, p);             //每遇到非空二叉树先向左走
            p->lchild;
        }
        else{                       //指针退栈，访问结点，遍历右子树
            Pop(S, p); visit(p);    //退栈，访问结点
            p->rchild;              //再向右子树走
        }
    }
}
```

5.递归后序遍历

```
void PostOrder(BiTree T){
    if (T != NULL){
        PreOrder(T->lchild);    //递归遍历左子树
        PreOrder(T->rchild);    //递归遍历右子树
        visit(T);               //访问结点
    }
}
```

6.非递归后序遍历

```
void PostOrder(BiTree T){
    InitStack(S1);                          //初始化栈，S1辅助做逆后续遍历
    InitStack(S2);                          //初始化栈，S2完成顺序转换；
    BiTree p = T;                           //p是遍历指针
    if (p != NULL){                         //p不空继续执行
        Push(S1, p);                        //根指针进栈
        while (!IsEmpty(S1)){               //栈S1不空循环
```

```
                Pop(S1, p);                     //结点指针从S1退栈
                Push(S2, p);                    //结点指针入栈S2
                if (p->lchild != NULL)          //左子树不为空
                    Push(S1,p->lchild);         //左子树入栈
                if (p->rchild != NULL)          //右子树不为空
                    Push(S1, p->rchild);        //右子树入栈
            }
            while (!IsEmpty(S2)){               //栈S2不空循环
                Pop(S2, p);                     //结点指针退栈
                visit(p);                       //访问结点
            }
        }
    }
```

7.层次遍历

```
    void LevelOrder(BiTree T){
        if (T){
            InitQueue(Q);                   //初始化辅助队列
            BiTree p;
            EnQueue(Q, T);                  //将根结点入队
            while (!IsEmpty(Q)){            //队列不空循环
                DeQueue(Q, p);              //队头元素出队
                visit(p);                   //访问结点
                if (p->lchild != NULL)      //左子树不空
                    EnQueue(Q, p->lchild);  //左子树入队列
                if (p->rchild != NULL)      //右子树不空
                    EnQueue(Q, p->rchild);  //右子树入队列
            }
        }
    }
```

3.线索二叉树

1.线索二叉树存储结构

```
    typedef struct ThreadNode{
        ElemType data;                      //数据元素
        struct ThreadNode *lchild, *rchild; //左、右孩子指针
        int ltag, rtag;                     //左、右线索标志
    }ThreadNode, *ThreadTree;
    //ltag == 0, lchild域指向结点的左孩子
    //ltag == 1, lchild域指向结点的前驱结点
    //rtag == 0, rchild域指向结点的右孩子
    //rtag == 1, rchild 域指向结点的后继结点
```

2.线索二叉树构造

先序遍历建立中序线索二叉树

```
void PreThread(ThreadTree &p, ThreadTree &pre){
    //先序遍历对二叉树线索化的递归算法
    if (p != NULL){
        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;                         //标记当前结点成为刚刚访问过的结点
        if (p->ltag = 0)                 //左指针非线索继续
            PreThread(p->lchild, pre);   //递归，线索化左子树
        if (p->rtag = 0)                 //右指针非线索继续
            PreThread(p->rchild, pre);   //递归，线索化右子树
    }
}
void CreatePreThread(ThreadTree T){
    ThreadTree pre = NULL;
    if (T != NULL){                  //非空二叉树，线索化
        PreThread(T, pre);           //线索化二叉树
        pre->rchild = NULL;          //处理遍历的最后一个结点
        pre->rtag = 1;
    }
}
```

中序遍历建立中序线索二叉树

```
void InThread(ThreadTree &p, ThreadTree &pre){
//中序遍历对二叉树线索化的递归算法
    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;                     //标记当前结点成为刚刚访问过的结点
        InThread(p->rchild, pre);    //递归，线索化右子树
    }
}
void CreateInThread(ThreadTree T){
```

```
    ThreadTree pre = NULL;
    if (T != NULL){                    //非空二叉树，线索化
        InThread(T, pre);              //线索化二叉树
        pre->rchild = NULL;            //处理遍历的最后一个结点
        pre->rtag = 1;
    }
}
```

后序遍历建立中序线索二叉树

```
void PostThread(ThreadTree &p, ThreadTree &pre){
    //后序遍历对二叉树线索化的递归算法
    if (p != NULL){
        PostThread(p->lchild, pre); //递归，线索化左子树
        PostThread(p->rchild, pre); //递归，线索化右子树
        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;                    //标记当前结点成为刚刚访问过的结点
    }
}
void CreatePostThread(ThreadTree T){
    ThreadTree pre = NULL;
    if (T != NULL){                    //非空二叉树，线索化
        PostThread(T, pre);            //线索化二叉树
        pre->rchild = NULL;            //处理遍历的最后一个结点
        pre->rtag = 1;
    }
}
```

3.线索二叉树遍历

1.中序线索二叉树遍历

```
//求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;    //最左下结点
    return p;
}
//求中序线索二叉树中结点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;                //rtag==1直接返回后继线索
}
```

```
//中序线索二叉树遍历算法
void Inorder(ThreadNode *T){
    for (ThreadNode *p = Firstnode(T); p != NULL; p = Nextnode(p))
        visit(p);
}

//求中序线索二叉树中中序序列下的最后一个结点
TBTNode *Lastnode(ThreadNode *p){
    while (p->rtag == 0)
        p = p->rchild;  //最右下结点
    return p;
}
//求中序线索二叉树中结点p在中序序列下的前驱结点
TBTNode *PreNode(ThreadNode *p){
    if (p->ltag == 0)
        return LastNode(p->lchild);
    else return p->lchild;      //ltag == 1直接返回前驱线索
}
```

4. 二叉排序树

1. 二叉排序树的

非递归查找

```
BSTNode *BST_Search(BiTree T, ElemType key, BSTNode *&p){
    p = NULL;
    while (T!=NULL&&key!=T->data){
        p = T;          //指向被查找结点双亲，用于插入或删除操作
        if (key < T->data)T = T->lchild;
        else T = T->rchild;
    }
    return T;
}
```

递归查找

```
BTSNode *BST_Search(BiTree *T, ElemType  key){
    if (T == NULL) return NULL;
    else{
        if (T->data == key)
            return T;
        else if (key < T->data)
            return BST_Search(BT->lchild, key);
        else
            return BST_Search(BT->rchild, key);
    }
}
```

2.二叉排序树的插入

```
int BST_Insert(BiTree &T, KeyType k){
    if (T == NULL){
        T = (BiTree)malloc(sizeof(BSTNode));
        T->key = k;
        T->lchild = T->rchild = NULL;
        return 1;
    }
    else if (k == T->key)
            return 0;
    else if (k < T->key)
        return BST_Insert(T->lchild, k);
    else
        return BST_Insert(T->rchild, k);
    }
}
```

3.二叉排序树的构造

```
void Create_BST(BiTree &T, KeyType str[], int n){
    T = NULL;int i = 0;
    while (i < n){
        BST_Insert(T, str[i]);
        i++;
    }
}
```

5.二叉树应用

1.层次遍历求二叉树宽度、高度

```
void MaxWidthAndLevel(BiTree T,int &MaxWidth,int &level){
    MaxWidth = 0;level = 0;
    if (T != NULL){
        int front = -1, rear = -1;      //front 出队指针   rear 入队指针
        int last = 0, level = 0;        //last 每一层的最右指针
        BiTree Q[MaxSize];              //模拟队列
        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){         //front==last一层遍历结束
```

```
                level++;                //level++
                last = rear;            //last指向下层最后一个节点
                MaxWidth = MaxWidth>(last - front) ? MaxWidth : (last - front);
            }
        }
    }
}
```

2.根结点左右子树叶子结点最远距离

```
    //即左右子树深度和
    int Depth(BiTree T){
        if (T == NULL)return 0;
        int ldepth = Depth(T->lchild) + 1;
        int rdepth = Depth(T->rchild) + 1;
        return ldepth > rdepth ? ldepth : rdepth;
    }
    int Dist = Depth(root->lchild) + Depth(root->rchild);
```

3.递归求二叉树宽度

```
    int count[MaxSize];     //全局数组
    int max = -1;           //全局变量
    void Width(BiTree T, int k){
        if (T == NULL)
            return  ;
        count[k]++;         //该层节点数++
        if (max<count[k])
            max = count[k];
        Width(T->lchild, k + 1);
        Width(T->rchild, k + 1);
    }
```

4.输出根结点到每个叶结点路径

```
    typedef struct Queue{
        struct BTNode *data[MaxSize];
        int rear = 1;
        int front = 1;
    };//定义队列
    void RootToLeaf(BiTree bt, Queue q){
        if (bt!=NULL){
            q.data[q.rear++] = bt;
            if (bt->lchild == NULL&&bt->rchild == NULL){
                while (q.front<q.rear)
                    cout << q.data[q.front++]->key << " ";
                cout << endl;
            }
            RootToLeaf(bt->lchild,q);
            RootToLeaf(bt->rchild,q);
```

```
        }
    }
```

输出指定结点祖先

```
    int Ancestors(BiTree root, ElemType x){
        if (!root)  return 0;
        if (root->data == x)    return 1;
        //如果子树中可以找到匹配值 那么此节点肯定是祖先结点
        if (Ancestors(root->lchild, x) || Ancestors(root->rchild, x)){
            printf("%c ", root->data);
            return 1;
        }
        return 0;
    }
```

5.层次遍历判断完全二叉树

算法思想：采用层次遍历的算法，将所有结点(包括空结点)加入队列。当遇到空结点时，查看其后是否有非空结点。若有，则二叉树不是完全二叉树。

```
    bool IsComplete(BiTree T){
        InitQueue(Q);
        if (!T)return 1;            //空树为满二叉树
        EnQueue(Q, T);
        while (!IsEmpty(Q)){
            DeQueue(Q, p);
            if (p){                 //结点非空，将其左、右子树入队列
                EnQueue(Q, p->lchild);
                EnQueue(Q, p->rchild);
            }
            else{                   //结点为空，检查其后是否有非空结点
                while (!IsEmpty(Q)){
                    DeQueue(Q, p);
                    if (p)return 0; //结点非空，则非完全二叉树
                }
            }
        }
        return 1;
    }
```

6.两棵二叉树 A、B，判断 B 是否 A 的子结构(结构和结点指均需相同)。

```
    //从某一相同结点开始，判断B是否A子结构
    bool partOfTree(BiTree A, BiTree B){
        if (!A)return false;                //A为空，B未匹配完成，B不是A子结构
        if (!B)return true;                 //匹配完毕，B是A的子结构
        if (A->data != B->data)return false;//A、B存在结点不匹配，B不是A子结构
        //继续递归匹配下一结点
        return
        partOfTree(A->lchild, B->lchild) && partOfTree(A->rchild, B->rchild);
```

```
}
//B的根结点可能在A中出现多次，递归匹配
bool hasSubTreeCore(BiTree A, BiTree B){
    bool result = false;
    if (A->data == B->data)      //找到同一起始结点，开始匹配
        result = partOfTree(A, B);
    if (!result&&A->lchild)      //本次匹配失败，A左子树不空，继续左子树匹配
        result = hasSubTreeCore(A->lchild, B);
    if (!result&&A->rchild)      //本次匹配失败，A右子树不空，继续右子树匹配
        result = hasSubTreeCore(A->rchild, B);
    return result;
}
bool hasSubTree(BiTree A, BiTree B){
    if (!B)return true;          //B为空，满足B是A子结构
    if (!A&&B)  return false;    //A空，B不空，B不是A子结构
    return hasSubTreeCore(A, B);
}
```

7.二叉树自下而上、从右到左层次遍历。

算法思想：进行原有层序遍历，通过栈辅助，最后统一出栈，即解。

```
void LevelOrder(BiTree T){
    if (T){
        InitStack(S);                    //初始化栈，栈内存放结点指针
        InitQueue(Q);                    //初始化辅助队列
        EnQueue(Q, T);                   //将根结点入队
        while (!IsEmpty(Q)){             //队列不空循环，从上至下层序遍历
            DeQueue(Q, p);               //队头元素出队
            Push(S, p);                  //元素入栈
            if (p->lchild != NULL)       //左子树不空
                EnQueue(Q, p->lchild);   //左子树入队列
            if (p->rchild != NULL)       //右子树不空
                EnQueue(Q, p->rchild);   //右子树入队列
        }
        while (IsEmpty(S)){              //自下而上、从右到左层次遍历
            Pop(S, p);
            visit(p);
        }
    }
}
```

8.通过先序序列、中序序列的数组建立二叉树。

```
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下标；l2,h2为后序的第一个和最后一个下标
    int i;
    root = (BiTree)malloc(sizeof(BiTNode));      //建根结点
    root->data = A[l1];                          //根结点值
```

```
    for (i=l2; B[i]!= root->data;i++);          //于中序序列寻找根结点
    //左子树长度不为0，递归建立左子树
    if (i-l2)root->lchild = PreInCreat(A,B,l1+1,l1+i-l2,l2,i-1);
    else root->lchild = NULL;
    //右子树长度不为0，递归建立右子树
    if (h2-i)root->rchild = PreInCreat(A,B,l1+i-l2+1,h1,i+1,h2);
    else root->rchild = NULL;
    return root;
}
```

9. 满二叉树先序序列数组获得后续序列。

```
void PreToPost(ElemType pre[],int L1,int R1,ElemType post[],int L2,int R2){
    int half;
    if (L1 <= R1){
        post[R2] = pre[L1];//将pre的第一个元素放到post最后
        half = (R1 - L1) / 2;
        //递归处理前半部分
        PreToPost(pre,L1+1,L1+half,post,L2,L2+half-1);
        //递归处理后半部分
        PreToPost(pre,L1+half+1,R1,post,L2+half,R2-1);
    }
}
```

10. 统计二叉树结点数、叶子结点数、双分支结点数。

```
//统计所有结点数
int n1 = 0;
void countAllNode(BiTree p){
    if (p != NULL){
        ++n1;
        countAllNode(p->lchild);
        countAllNode(p->rchild);
    }
}
//统计叶子结点数
int n2 = 0;
void countLeafNode(BiTree p){
    if (p != NULL){
        if (p->lchild == NULL&&p->rchild == NULL)//判断是否叶子结点
            ++n2;
        countLeafNode(p->lchild);
        countLeafNode(p->rchild);
    }
}
//统计双分支结点数
int n3 = 0;
void countDoubleNode(BiTree p){
```

```
    if (p != NULL){
        if (p->lchild != NULL&&p->rchild != NULL)//判断是否双分支结点
            ++n3;
        countDoubleNode(p->lchild);
        countDoubleNode(p->rchild);
    }
}
```

11. 交换二叉树左、右子树。

```
void swap(BiTree T){
    if (T){
        swap(T->lchild);            //递归交换左子树
        swap(T->rchild);            //递归交换右子树
        BiTree temp = T->lchild;    //交换左、右子树
        T->lchild = T->rchild;
        T->rchild = temp;
    }
}
```

12. 求二叉树先序序列中第 k 个结点值。

```
int num = 0;
void preNode(BTNode *p,int k){
    if (p != NULL){
        num++;
        if (k == num){
            cout << p->data << " ";
            return;
        }
        preNode(p->lchild, k);
        preNode(p->rchild, k);
    }
}
```

13. 删除二叉树所有值为 x 的结点的子树。

```
void DeleteXTree(BiTree T){
    if (T){
        DeleteXTree(T->lchild); //递归删除左子树
        DeleteXTree(T->rchild); //递归删除右子树
        free(T);                //释放被删除结点所占空间
    }
}
void Search(BiTree T,ElemType x){
    if (T){
        if (T->data == x){
            DeleteXTree(T);                 //若根结点为x，则删除整棵树
            exit(0);
        }
```

```
            InitQueue(Q);                          //初始化辅助队列
            BiTree p;
            EnQueue(Q, T);                         //将根结点入队
            while (!IsEmpty(Q)){                   //队列不空循环
                DeQueue(Q, p);                     //队头元素出队
                if (p->lchild){
                    if (p->lchild->data == x){     //左子树符合，删除右子树
                        DeleteXTree(p->lchild);
                        p->lchild = NULL;          //父结点左子女置空
                    }
                    else EnQueue(Q, p->lchild);    //左子树入队列
                }
                if (p->rchild){
                    if (p->rchild->data == x){     //右子树符合，删除右子树
                        DeleteXTree(p->rchild);
                        p->rchild = NULL;          //父结点右子女置空
                    }
                    else EnQueue(Q, p->rchild);    //右子树入队列
                }
            }
        }
    }
```

14. 查找 p、q 结点最近公共祖先 r。

```
    typedef struct Stack{
        BiTree data[MaxSize];
        int top = 0;
    }Stack;//用于存储指定结点的祖先结点

    //查找指定元素的祖先
    int Ancestors(BiTree T, BiTree x, Stack &S){
        if (!T)  return 0;
        if (T == x)   return 1;
        //如果子树中可以找到匹配值 那么此节点肯定是祖先结点
        if (Ancestors(T->lchild, x,S) || Ancestors(T->rchild, x,S)){
            S.data[++S.top] = T;
            return 1;
        }
        return 0;
    }
    void ComAncestors(BiTree T, BiTree p, BiTree q,BiTree &r){
        Stack S1, S2;
        int flag = 0;                          //找到最近公共祖先置为1 结束循环
        Ancestors(T, p, S1);                   //查找p的祖先
        Ancestors(T,q, S2);                    //查找q的祖先
```

```
    for (int i = 1; i <= S1.top; i++)    //查找p、q最近公共祖先
        if (!flag)
            for (int j = 1; j <= S2.top; j++)
                if (S1.data[i] == S2.data[j]){
                    r = S1.data[i];
                    flag = 1;
                    break;
                }
}
```

15. 将二叉树叶子结点从左往右顺序串连。

```
void linkLeafNode(BiTree p, BiTree &head, BiTree &tail){
    if (p != NULL){
        if (p->lchild == NULL&&p->rchild == NULL){//叶子结点
            if (head == NULL){
                head = p;
                tail = p;
            }else{
                tail->rchild = p;
                tail = p;
            }
        }
        linkLeafNode(p->lchild, head, tail);
        linkLeafNode(p->rchild, head, tail);
    }
}
```

16. 判断两棵二叉树是否相似(结构相同)。

```
int similar(BiTree T1,BiTree T2){
    int leftS, rightS;
    if (T1 == NULL&&T2 == NULL)          //两树均空
        return 1;
    else if (T1 == NULL || T2 == NULL)   //只有一树为空
        return 0;
    else{                                //递归判断
        leftS = similar(T1->lchild, T2->lchild);
        rightS = similar(T1->rchild, T2->rchild);
        return leftS&&rightS;
    }
}
```

17. 中序线索二叉树查找指定结点后序的前驱结点。

算法思想：在后序序列中，若 p 有右子女，则右子女为其前驱；若无右子女但有左子女，则左子女为其前驱；若 p 左、右子女均无，则其中序左线索指向的某祖先结点 f，若 f 有左子女，则该左子女为 p 前驱；若 f 无左子女，则顺其前驱一直找到双亲有左子女，该左子女既为 p 前驱；若 p 是中序遍历第一个结点，则结点 p 无前驱。

```
BiThrTree InPostPre(BiThrTree T, BiThrTree p){
```

```
    BiThrTree q;
    if (p->rtag == 0)           //若p有右子女，右子女为其后序前驱
        q = p->rchild;
    else if (p->ltag == 0)      //若p只有左子女，左子女为其后序前驱
        q = p->lchild;
    else if (p->lchild == NULL) //p为中序第一个结点，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)
            q = p->lchild;  //p结点的祖先的左子女是其后序前驱
        else
            q = NULL;
    }
    return q;
}
```

17.计算叶结点 WPL。

先序遍历：

```
int WPL(BiTree T, int deepth){
    static int wpl = 0;     //定义静态变量存储wpl
    if (T->lchild == NULL&&T->rchild == NULL)//为叶结点，累积wpl
        wpl += deepth*T->data;
    if (T->lchild != NULL)  //左子树不空，递归遍历左子树
        WPL(T->lchild, deepth + 1);
    if (T->rchild != NULL)  //右子树不空，递归遍历右子树
        WPL(T->rchild, deepth + 1);
    return wpl;
}
```

层序遍历：

```
int WPL(BiTree T){
    static int wpl = 0;
    int level = 0;
    if (T != NULL){
        int front = -1, rear = -1;      //front 出队指针   rear 入队指针
        int last = 0, deepth = 0;       //last 每一层的最右指针
        BiTree Q[MaxSize];              //模拟队列
        Q[++rear] = T;
        BiTree p;
        while (front<rear){
            p = Q[++front];
            if (p->lchild == NULL&&p->rchild == NULL)//为叶结点，累积wpl
                wpl += deepth*p->data;
            if (p->lchild)
```

```
                Q[++rear] = p->lchild;
            if (p->rchild)
                Q[++rear] = p->rchild;
            if (front == last){          //front==last一层遍历结束
                deepth++;                //deepth++
                last = rear;             //last指向下层最后一个节点
            }
        }
    }
    return wpl;
}
```

18.输出给定表达式树的中缀表达式(通过括号反映计算次序)。

```
BtreeToExp(T, 1);
void BtreeToExp(BiTree T, int deep){
    if (T == NULL)return;
    else if (T->lchild == NULL&&T->rchild == NULL)
        cout << T->data ;        //输出操作数，不加括号
    else{
        if (deep>1)cout << "("; //有子表达式，加一层括号
        BtreeToExp(T->lchild, deep + 1);
        cout << T->data;         //输出操作符
        BtreeToExp(T->rchild, deep + 1);
        if (deep>1)cout << ")"; //有子表达式，加一层括号
    }
}
```

19.判断二叉树是否二叉排序树。

```
ElemType predt = -0xFFFF;        //保存当前结点中序前驱，初值负无穷
int JudgeBST(BiTree T){
    int b1, b2;
    if (T == NULL) return 1;     //空树
    else{
        b1 = JudgeBST(T->lchild);//判断左子树是否二叉排序树
        if (b1 == 0 || predt >= T->data)//若左子树返回值为0
            return 0;                   //或前驱大于等于当前结点
        predt = T->data;                //保存当前结点关键字
        b2 = JudgeBST(T->rchild);       //判断右子树
        return b2;                      //返回右子树结果
    }
}
```

20.指定结点在二叉排序树层次。

```
int level(BiTree T, BiTree p){
    int level = 0;                  //统计层数
    BiTree t = T;
    if (t != NULL){
```

```
            ++level;
            while (t->data!=p->data){
                if (t->data < p->data)  //在右子树中寻找
                    t = t->rchild;
                else t = t->lchild;     //在左子树寻找
                ++level;                //层数+1
            }
        }
        return level;
    }
```

21.判断二叉树是否平衡二叉树。

后序遍历算法思想：

若 T 为空，balance=1;

若 T 仅有根结点，balance=1;

否则，对 T 左右子树执行递归算法，返回左右子树的高度和平衡标记，T 的高度为最高子树高度+1.若左右子树高度差>1,则 balance=0;若左右子树高度差<=1，且左右子树均平衡，则 balance=1,否则 balance=0.

```
    void JudgeAVL(BiTree T, int &balance, int &h){
        int bl = 0, br = 0, hl = 0, hr = 0;//左右子树平衡标记和高度
        if (T == NULL){     //空树
            h = 0;
            balance = 1;
        }
        else if(T->lchild==NULL&&T->rchild==NULL){//仅有根结点
            h = 1;
            balance = 1;
        }
        else{
            JudgeAVL(T->lchild, bl, hl);        //递归判断左子树
            JudgeAVL(T->rchild, br, hr);        //递归判断右子树
            h = (hl > hr ? hl : hr) + 1;
            if (abs(hl - hr) < 2)       //若子树高度差绝对值<2
                balance = bl&&br;       //左右子树均平衡才平衡
            else balance = 0;
        }
    }
```

22.二叉排序树最大、最小值。

```
    ElemType MinData(BiTree T){
        while (T->lchild)
            T = T->lchild;
        return T->data;
    }
    ElemType MaxData(BiTree T){
        while (T->rchild)
```

```
            T = T->rchild;
        return T->data;
    }
```

23.从大到小输出二叉排序树不小于 k 的值。

```
    void OutPut(BiTree T, ElemType k){
        if (T == NULL) return;
        if (T->rchild != NULL)
            OutPut(T->rchild, k);//递归输出右子树结点
        if (T->data >= k)
            cout << " " << T->data;//是输出大于k的值
        if (T->lchild != NULL)
            OutPut(T->lchild, k);//递归输出右子树结点
    }
```

24.以 O(log2n)查找随机二叉排序树第 k 个结点(结点还有 count 域表示当前结点子树个数)。

```
    BiTree Search(BiTree T,int k){
        if (k<1 || k>T->count)return NULL;
        if (T->lchild == NULL){//左为空
            if (k == 1)return T;//查找成功
            else return Search(T->rchild, k - 1);//在右子树
        }
        else{
            if (T->lchild->count == k - 1)//查找成功
                return T;
            if (T->lchild->count > k - 1)//在左子树
                return Search(T->lchild, k);
            if (T->lchild->count < k - 1)//在右子树
                return Search(T->rchild, k - (T->lchild->count + 1));
        }
    }
```

专题五 广义表

1.取表头

```
    int Head(GList ls, GList h, ElemType x){
        GList p;
        if (ls == NULL)return 0;
        else{
            p = ls->hp;
            if (p->tag == 0){
                h = NULL; x = p->data;
            }
            else { h = p; x = NULL; }
```

```
        }
    }
```

2.取表尾

```
    int Tail(GList ls, GList h){
        if (ls == NULL)return 0;
        else h = ls->tp;
    }
```

专题六 图

1.图的存储结构

1.邻接矩阵

```
    #define MaxVertexNum 100                        //顶点数目的最大值
    typedef char VertexType;                        //顶点的数据类型
    typedef int EdgeType;                           //带权图中边权值的数据类型
    typedef struct{
        VertexType Vex[MaxVertexNum];               //顶点表
        EdgeType edge[MaxVertexNum][MaxVertexNum];  //邻接矩阵，边表
        int vexnum, arcnum;                         //图的当前顶点数和弧数
    }MGraph;
```

2.邻接表

```
    #define MaxVertexNum 100                    //顶点数目的最大值
    typedef char VertexType;                    //顶点的数据类型
    typedef int EdgeType;                       //带权图中边上权值的数据类型
    typedef struct ArcNode{                     //边表结点
        int adjvex;                             //该弧所指向的顶点的位置
        struct ArcNode *nextarc;                //指向下一条弧的指针
    }ArcNode;
    typedef struct VNode{                       //顶点表结点
        VertexType data;                        //顶点信息
        ArcNode *firstarc;                      //指向第一条依附该顶点弧的指针
    }VNode, AdjList[MaxVertexNum];
    typedef struct{
        VNode adjlist[MaxVertexNum];            //邻接表
        int vexnum, arcnum;                     //图的顶点数和弧数
    }ALGraph;                                   //ALGraph 是邻接表存储的图类型
```

2.邻接表相关

1.二叉链表存储转邻接表存储

```
    void BtTreeToALGraph(BiTree bt, int n, ALGraph alg){
```

```
    //bt为二叉树的根，n为结点数，alg为邻接表
    BiTree p; ArcNode *s;
    if (bt != NULL){
        INITQUEUE(Q); ENQUEUE(Q, bt);
        int i = 1;//i记录目前正在处理的编号
        int j = 1;//j记录目前最后一个结点的编号
        while (!EMPTY(Q)){
            p = DEQUEUE(Q);
            //顶点表结点赋值
            alg.adjlist[i].data = p->data;
            alg.adjlist[i].firstarc = NULL;
            if (p->lchild != NULL){
                ENQUEUE(Q, p->lchild);
                j = j + 1;
                s = (ArcNode*)malloc(sizeof(ArcNode*));
                s->adjvex = j;
                //头插法插入边表结点
                s->nextarc = alg.adjlist[i].firstarc;
                alg.adjlist[i].firstarc = s;
            }
            if (p->rchild != NULL){
                ENQUEUE(Q, p->rchild);
                j = j + 1;
                s = (ArcNode*)malloc(sizeof(ArcNode*));
                s->adjvex = j;
                s->nextarc = alg.adjlist[i].firstarc;
                alg.adjlist[i].firstarc = s;
            }
            i = i + 1;
        }
    }
}
```

专题七 排序与查找