第2章：线性表

2.2 线性表的顺序表示

2010：将一个线性表中的元素循环左移p个位置

- **题目描述**

　　设将n (n>1) 个整数存放到一维数组R中。试设计一个在时间和空间两方面都尽可能高效的算法。将R中保存的序列循环左移p (0<p<n) 个位置，即将R中的数据由$(X_0, X_1, \cdots, X_{n-1})$变换为$(X_p, X_{p+1}, \ldots, X_0, X_1, \ldots, X_{p-1})$。

- **算法思想**

　　可以将这个问题看做把数组ab转换成数组ba(a代表数组的前p个元素，b代表数组中余下的n-p个元素)，先将a逆置得到$a^{-1}b$，最后将整个b逆置得到$a^{-1}b^{-1}$，最后将整个$a^{-1}b^{-1}$逆置得到$(a^{-1}b^{-1})^{-1}=ba$。设Reserve函数将数组元素逆置的操作，对abcdefgh向左循环移动3(p=3)个位置如下：

> Reverse(0,p-1)得到cbadefgh
>
> Reserve(p,n-1)得到cbahgfed
>
> Reservr(0,n-1)得到defghabc

Reserve中的两个参数分别表示数组中待缓缓元素的始末位置。

- **算法实现**

```
void Reverse(int R[],int from,int to){
    int i,temp;
    for(i=0;i<(to-from+1)/2;i++){
        temp = R[from+i];
        R[from+i] = R[to-i];
        R[to-i] = temp;
    }
}
void Converse(int R[],int n,int p){
    Reserve(R,0,p-1);
    Reserve(R,p,n-1);
    Reserve(R,0,n-1);
}
```

- **复杂度分析**

　　上述算法中三个Reserve函数的时间复杂度分别为$O(p/2), O((n-p)/2), O(n/2)$，故算法的复杂度为$O(n)$，空间复杂度为$O(1)$

- **另解1**

　　创建大小为p的辅助数组S，将R前p个整数依次暂存在S中，同时将R中后n-p个整数左移，然后将暂存在S中的p个元素依次放回R中的后续单元。

　　时间复杂度：$O(n)$，空间复杂度：$O(n)$

- **另解2**

创建大小为n的辅助数组S，将R的前p个元素暂存在S的后p个单元中，后n-p个元素暂存在S的前n-p个单元中，最后将S中的所有元素一一赋值回R。

时间复杂度：$O(n)$，空间复杂度：$O(n)$

2011：求两个线性表的中位数

- **题目描述**

一个长度为$L(L \geq 1)$的升序序列S，处在第$\lceil L/2 \rceil$个位置的数称为S中的中位数。例如，若序列$S1=(11,13,15,17,19)$，则$S1$的中位数是15，两个序列的中位数是含它们所有元素的升序序列的中位数。例如，若$S2=(2,4,6,8,20)$，则$S1$和$S2$的中位数是11。现有两个等长升序序列A和B，试设计一个在时间和空间两方面都尽可能高效的算法，找出两个序列A和B的中位数。

- **算法设计思想**

分别求出序列A和B的中位数，设为a和b，求序列A和B的中位数过程如下：

1）若a=b，则a或b即为所求的中位数，算法结束

2）若a<b，则舍弃序列A中较小的一半，同时舍弃序列B中较大的一半，要求舍弃的长度相等

3）若a>b，则舍弃序列A中较大的一半，同时舍弃序列B中较小的一半，要求舍弃的长度相等

在保留的两个升序序列中，重复1）、2）、3），直到两个序列中只含一个元素为止，较小者即为所求的中位数。

- **算法实现**

```
int M_Search(int A[],int B[],int n){
    int s1=0,d1=n-1,m1,s2=1,d2=n-1,m2;
    //分别表示序列A和B的首位数、末位数和中位数
    while(s1!=d1||s2!=d2){
        m1=(s1+d1)/2;
        m2=(s2+d2)/2;
        if(A[m1]==B[m2])
            return A[m1];    //满足条件1)
        if(A[m1]<B[m2]){    //满足条件2)
            if((s1+d1)%2==0){    //元素个数为奇数
                s1=m1;   //舍弃A中间点以前的部分且保留中间点
                d2=m2;   //舍弃B中间点以后的部分且保留中间点
            }
            else{        //元素个数为偶数
                s1=m1+1;     //舍弃A中间点及中间点以前部分
                d2=m2;       //舍弃B中间点以后部分且保留中间点
            }
        }
        else{   //满足条件3)
            if((s1+d1)%2==0){    //元素个数为奇数
                d1=m1;   //舍弃A中间点以后的部分且保留中间点
                s2=m2;   //舍弃B中间点以前的部分且保留中间点
            }
            else{        //元素个数为偶数
                d1=m1+1;     //舍弃A中间点及中间点以后部分
                s2=m2;       //舍弃B中间点以前的部分且保留中间点
            }
        }
    }
```

```
    return A[s1]<B[s2]?A[s1]:B[s2];
}
```

- **时间复杂度分析**

算法的时间复杂度为$O(\log_2 n)$,空间复杂度为$O(1)$。

- **另解1**

给序列A和B设定两个浮标 i 和 j 分别指向A、B的第一个元素，即初始值为0。A、B中的元素根据浮标所对应的元素不停的比较，若A[i] > B[j]，则i+1，否则j+1。当i+j==n-1时停止比较，此时的A和B的中位数已经找到，中位数为min{A[i],B[j]}。

时间复杂度为：$O(n)$，空间复杂度为：$O(1)$

2013：找出序列主元素

- **题目描述**

已知一个整数序列$A=(a_0,a_1,...,a_{n-1})$，其中$0\le a_i\le n(0\le i\le n)$。若存在$a_{p1}=a_{p2}=\cdots=a_{pm}=x$且$m>n/2(0\le p_k<n,1\le k\le m)$，则称$x$为主元素。例如$A=(0,5,5,3,5,7,5,5)$，则5为主元素；又如$A=(0,5,5,3,5,1,5,7)$，则$A$没有主元素。假设$A$的$n$个元素保存在一个一维数组中，请设计一个尽可能高效的算法，找出A的主元素。若存在主元素，则输出该元素；否则输出-1。

- **算法思想**

算法的策略是从前向后扫描数组元素，标记出一个可能成为主元素的元素num，然后重新计数，确认num是否是主元素。

算法可分为如下两步：

① 选取候选主元素：依次扫描所给数组中的每个整数，将第一个遇到的整数num保存到c中，记录num的出现次数为1；若遇到下一个整数仍等于num，则计数加1，否则计数减1；当计数减到0时，将遇到的下一个整数保存到c中，计数重新记为1，开始新一轮的计数，即从当前位置开始重复上述过程，知道扫描完全部数组元素。

② 判断c中元素是否是真正的主元素；再次扫描该数组，统计c中元素出现的次数，若大于n/2，则为主元素；否则，序列中不存在主元素。

- **算法实现**

```
int Majority(int A[],int n){
    int i,c,count = 1;       //c用来保存候选主元素，count用来计数
    c = A[0];                //设置A[0]为候选主元素
    for(i=1;i<n;i++){
        if(A[i]==c)
            count++;         //对A中的候选主元素计数
        else
            if(count>0)      //处理不是候选主元素
                count--
            else{
                c = A[i];
                count = 1;
            }
    }
    if(i=count=0;i<n;i++)
        if(A[i]==c)
```

```
            count++;
    if(count>n/2)
        return c;           //确定候选主元素
    else
        return -1;          //不存在主元素
}
```

- **复杂度分析**

算法时间复杂度：$O(n)$，空间复杂度：$O(1)$

- **另解1**

使用一个和序列同等长度的辅助数组**(其下标就是序列的元素，下标对应的值即序列元素出现的次数)**存储序列中的不同元素出现的个数，然后再扫描一遍辅助数组选出元素的下标，返回该下标，否则返回-1。

时间复杂度：$O(n)$，空间复杂度：$O(n)$

2018：找出数组的未出现的最小正数

- **题目描述**

给定一个含 n（n≥1）个整数的数组，请设计一个在时间上尽可能高效的算法，找出数组中未出现的最小正整数。例如，数组{-5, 3, 2, 3}中未出现的最小正整数是 1；数 组{1, 2, 3}中未出现的最小正整数是 4。

- **算法思想**

题目要求算法时间上尽可能高效，因此采用空间换时间的办法。分配一个用于标记的数组 B[n]，用来记录 A 中是否出现了 1 ~ n 中的正整数，B[0]对应正整数 1，B[n-1]对应正整数 n，初始化 B 中全部为 0。由于 A 中含有 n 个整数，因此可能返回的值是 1 ~ n + 1，当 A 中 n 个数恰好为 1 ~ n 时返回 n + 1。当数组 A 中出现了小于等于 0 或大于 n 的值时，会导致 1 ~ n 中出现空余位置，返回结果必然在 1 ~ n 中，因此对于 A 中出现了小于等于 0 或大于 n 的值可以不采取任何操作。

经过以上分析可以得出算法流程：从 A[0]开始遍历 A，若 0 < A[i] <= n，则令 B[A[i]-1] = 1； 否则不进行操作。对 A 遍历结束后，开始遍历数组 B，若能查找到第一个满足 B[i] == 0 的下标 i，返回 i + 1 即为结果，此时说明 A 中未出现的最小正整数在 1 ~ n 之间。若 B[i]全部不为 0， 返回 i + 1（跳出循环时 i = n，i + 1 等于 n + 1），此时说明 A 中未出现的最小正整数是 n + 1。

- **算法实现**

```
int findMissMin(int A[],int n)
{
    int i,*B; //标记数组
    B=(int *)malloc(sizeof(int)*n);      //分配空间
    memset(B,0,sizeof(int)*n);           //赋初值为 0
    for(i=0;i<n;i++)
        if(A[i]>0&&A[i]<=n)              //若 A[i]的值介于 1~n，则标记数组 B
            B[A[i]-1]=1;
    for(i=0;i<n;i++)                     //扫描数组 B，找到目标值
        if (B[i]==0) break;
    return i+1;                          //返回结果
}
```

- **算法性能分析**

时间复杂度：遍历 A 一次，遍历 B 一次，两次循环内操作步骤为 O(1)量级，因此时间复杂度为 O(n)。

空间复杂度：额外分配了B[n]，空间复杂度为 O(n)。

2.3 线性表的链式表示

2009：寻找单链表的倒数第k个结点

- **题目描述**

已知一个带有头结点的单链表，结点结构为：

data	link

假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点(k为正整数)。若查找成功，算法输出该结点的data域值，并返回1；否则，返回0。

- **算法思想**

定义两个指针变量p和q，初始时均指向头结点的下一个结点(链表的第一个结点)。p指针沿链表移动；当p移动到第k个结点时，q指针开始与p指针同步移动；当p指针移动到最后一个结点时，q指针所指示结点为倒数第k个结点。以上过程对链表仅进行一遍扫描。

- **算法步骤**

① count = 0，p和q指向链表表头结点的下一个结点；

② 若p为空，转⑤；

③ 若count等于k，则q指向下一个结点；否则，count = count + 1；

④ p指向下一个结点，转②

⑤ 若count等于k，则查找成功，输出该结点的data域值，返回1；否则，说明k值超过了线性表的长度，查找失败，返回0；

⑥ 算法结束。

- **算法实现**

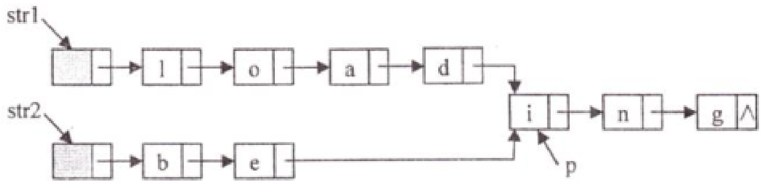
```
typedef int ElemType;
typedef struct LNode{
    ElemType data;      //数据域
    struct LNode *next; //指针域
}*LinkList,LNode;
int Search_k(LinkList list, int k){
    LinkList p = list->next,q = list->next;     //指针p,q指向链表第一个结点
    int count = 0;                              //计数器，记录是否到达第k个结点
    while(p != NULL){
        if(count < k)   count++;                //继续向前移动，表示未到达k
        else
            q = q->next;                        //与指针p同步移动
        p = p->next;
    }
    if(count < k)
        return 0;                               //查找失败
    else{
```

```
        printf("%d",q->data);                   //输出数据域
        return 1;
    }
}
```

2012：寻找保存单词的单链表的共享后缀

- **题目描述**

假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，则可共享相同的后缀存储空间，例如，“loading”和“being”的存储映像如下图所示。



设str1和str2分别指向两个单词所在单链表的头结点，链表结点结构为 data next ，请设计一个时间上尽可能高效的算法，找出由str1和str2所指向两个链表共同后缀的起始位置(如图中字符i所在结点的位置p)。

- **算法思想**

顺序遍历两个链表到尾结点时，并不能保证两个链表同时到达尾结。这是因为两个链表的长度不同。

假设一个链表比另一个链表长k个结点，我们先在长链表上遍历k个结点，之后同步遍历两个链表，这样就能保证它们同时到达最后一个结点。由于两个链表从第一个公共结点到链表的尾结点都是重合的，所以它们肯定能同时到达第一个公共结点。算法的基本设计思想如下：

① 分别求出str1和str2所指的两个链表的长度m和n；

② 将两个链表以表尾对齐：令指针p、q分别指向str1和str2的头结点，若m>=n，则使p指向链表中的第m-n+1个结点；若m<n，则使q指向链表中的第n-m+1个结点，即使指针p和q所指的结点到表尾的长度相等。

③ 反复将指针p和q同步向后移动，并判断它们是否指向同一个结点。若p和q指向同一个结点，则该结点即为所求的公共后缀的起始位置。

- **算法实现**

```
LinkNode *Find_1st_Common(LinkList str1,LinkList str2){
    int len1 = Length(str1),len2 = Length(str2);
    LinkNode *p,*q;
    for(p=str1;len1>len2;len1--)     //使p指向链表q与指向的链表等长
        p=p->next;
    for(q=str2;len1<len2;len2--)     //使q指向链表p与指向的链表等长
        q=q->next;
    while(p->next!=NULL&&p->next!=q->next){ //查找公共后缀起始点
        p=p->next;
        q=q->next;
    }
    return p->next; //返回公共后缀的起始点
}
```

- **算法复杂度**

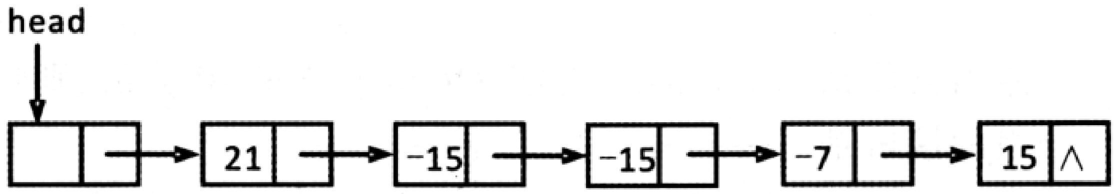
时间复杂度为：$O(\max(len1, len2))$，空间复杂度为：$O(1)$

2015：删除单链表中绝对值相等的结点

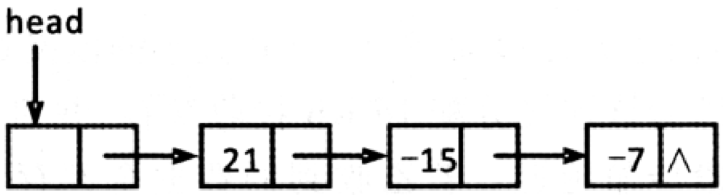
- **题目描述**

用单链表保存m个整数，结点的结构为：

，且$|data| \leq n$(n为正整数)。现要求设计一个时间复杂度尽可能高效的算法，对于链表中data的绝对值相等的结点，仅保留第一个出现的结点而删除其余绝对值相等的结点。例如，若单链表head如下：



则删除结点后的head为：



- **算法思想**

算法的核心思想是用空间换时间。使用辅助数组记录链表中已经出现的数字，从而只需对链表进行一趟扫描。

因为|data|≤n，故辅助数组q的大小为$n+1$，各元素的初值为0。依次扫描链表中的各结点，同时检查q[|data|]的值，如果为0，则保留该结点，并令q[|data|]=1；否则，将该结点从链表中删除。

- **算法实现**

结点的数据类型定义：

```
typedef struct node{
    int data;
    struct node *link;
}NODE;
typedef NODE *PNODE;
```

算法实现：

```
void func(PNODE h,int n){
    PNODE p = h,r;
    int *q,m;
    q = (int*)malloc(sizeof(int)*(n+1));     //申请n+1个位置的辅助空间
    for(int i=0;i<n+1;i++)
        *(q+i) = 0;
    while(p->link!=NULL){
        m = p->link->data>0?p->link->data:-p->link->data;
        if(*(q+m)==0){   //判断该结点的data是否已出现过
```

```
            *(q+m) = 1; //保留
            p = p->link;
        }else{
            r = p->link;
            free(r);
        }
    }
    free(q);
}
```

- **算法性能分析**

时间复杂度：$O(m)$，空间复杂度：$O(n)$。

2019：重新排列单链表的各结点

- **题目描述**

设线性表 $L=\{a_1,a_2,a_3,\ldots,a_{n-2},a_{n-1},a_n\}$采用带头结点的单链表保存，链表中的 结点定义如下：

```
typedef struct node{
    int data;
    struct node *next;
}NODE;
```

请设计一个空间复杂度为 O(1)且时间上尽可能高效的算法，重新排列 L 中的各结点，得到线性表 $L'=\{a_1,a_n,a_2,a_{n-1},a_3,a_{n-2}\ldots\}$。

- **算法思想**

先观察L和L'，发现 L' 是由 L 摘 取第一个元素，再摘取倒数第一个元素......依次合并而成的。为了方便链表后半段取元素，需 要先将 L 后半段原地逆置［题目要求空间复杂度为 O(1)，不能借助栈］，否则每取最后一个结点都需要遍历一次链表。

①先找出链表 L 的中间结点，为此设置两个指针 p 和 q，指针 p 每次走一步，指针 q 每次走两步，当指针 q 到达链尾时，指针 p 正好在链表的中间结点；

②然后将 L 的后半段结点原地逆置。

③从单链表前后两段中依次各取一个结点，按要求重排。

- **算法实现**

```
void change_list(NODE*h)
{
    NODE *p,*q,*r,*s;
    p=q=h;
    while(q->next!=NULL) //寻找中间结点
    {
        p=p->next;       //p 走一步
        q=q->next;
        if(q->next!=NULL) q=q->next; //q 走两步
    }
    q=p->next;      //p 所指结点为中间结点，q 为后半段链表的首结点
    p->next=NULL;
    while(q!=NULL)   //将链表后半段逆置
```

```
    {
        r=q->next;
        q->next=p->next;
        p->next=q;
        q=r;
    }
    s=h->next; //s 指向前半段的第一个数据结点，即插入点
    q=p->next; //q 指向后半段的第一个数据结点
    p->next=NULL;
    while(q!=NULL) //将链表后半段的结点插入到指定位置
    {
        r=q->next; //r 指向后半段的下一个结点
        q->next=s->next; //将 q 所指结点插入到 s 所指结点之后
        s->next=q;
        s=q->next; //s 指向前半段的下一个插入点
        q=r;
    }
}
```

- **算法性能分析**

第 1 步找中间结点的时间复杂度为 O(n)，第 2 步逆置的时间复杂度为 O(n)，第 3 步合 并链表的时间复杂度为 O(n)，所以该算法的时间复杂度为 O(n)。

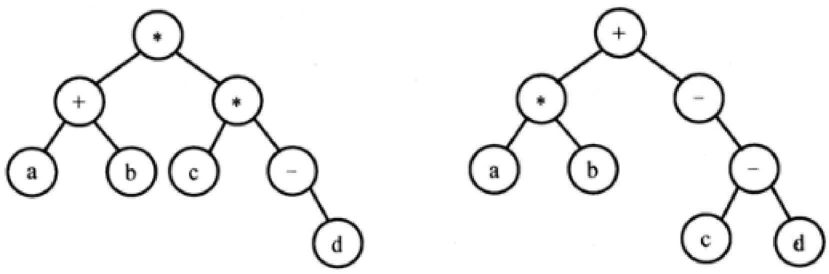
第4章：树与二叉树

4.3 二叉树的遍历和线索树

2017：二叉树转换为中缀表达式

- **题目描述**

　　请设计一个算法，将给定的表达式树(二叉树)转换为等价的中缀表达式(通过括 号反映操作符的计算次序)并输出。例如，当下列两棵表达式树作为算法的输入时：



输出的等价中缀表达式分别为(a+b) * (c * (-d))和(a * b)+(-(c-d))。

二叉树结点定义如下：

```
typedef struct node
{
    char data[10];      //存储操作数或操作符
    struct node *left,*right;
}BTree
```

- **算法思想**

表达式树的中序序列加上必要的括号即为等价的中缀表达式。可以基于二叉树的中序遍历策略得到所需的表达式。表达式树中分支结点所对应的子表达式的计算次序，由该分支结点所处的位置决定。为得到正确的中缀表达式，需要在生成遍历序列的同时，在适当位置增加必要的括号。显然，表达式的最外层(对应根结点)及操作数(对应叶结点)不需要添加括号。

- **算法实现**

```
void BtreeToE(BTree *root){
    BtreeToExp(root,1);         //根的高度为1
    void BtreeToExp(BTree *root,int deep){
        if(root == NULL) return;        //树空
        else if(root->left == NULL && root->right == NULL)  //若为叶子结点
            printf("%s",root->data);    //输出操作数
        else{
            if(deep>1)  print("(");     //若有子表达式则加1层括号
            BtreeToExp(root->left,deep+1);
            printf("%s",root->data);    //输出操作符
            BtreeToExp(root->right,deep+1);
            if(deep>1) printf(")");     //若有子表达式则加1层括号
        }
    }
}
```

4.5 树与二叉树的应用

2014：求二叉树的带权路径长度WPL

- **题目描述**

二叉树的带权路径长度(WPL)是二叉树所有叶结点的带权路径长度之和。给定一颗二叉树T，采用二叉树链表存储，结点结构为：

left	weight	right

其中叶结点的weight域保存该结点的非负权值。设root为指向T的根结点指针，请设计T的WPL的算法。

- **算法思想**

① 基于先序递归遍历的算法是用一个static变量记录wpl，把每个结点的深度作为递归函数的一个参数传递，算法步骤如下：

若该结点是叶子结点，那么变量wpl加上该结点的深度与权值之积；

若该结点是非叶子结点，那么若左子树不为空，对左子树调用递归算法，若右子树不为空，对右子树调用递归算法，深度参数均为本结点的深度参数加一。

最后返回计算出的wpl即可。

② 基于层次遍历的算法是使用队列进行层次遍历，并记录当前的层数，

当遍历到叶子结点时，累计wpl；

当遍历到非叶子结点时对该结点的把该结点的子树加入队列；

当某结点为该层的最后一个结点时，层数自增1；

队列空时遍历结束，返回wpl。

- **二叉树结点数据类型定义**

```
typedef struct BiTNode{
    struct Node *left,*right;
    int weight;
}BiTNode,*BiTree;
```

- **算法实现**

① 基于先序遍历的算法：

```
int WPL(BiTree root){
    return wpl_PreOrder(root,0);
}
int wpl_PreOrder(BiTree root,int deep){
    static int wpl = 0;          //定义一个static变量存储wpl
    if(root->lchild == NULL && root->rchild == NULL)    //若为叶字结点，累计wpl
        wpl+=deep*root->weight;
    if(root->lchild != NULL)                    //若左子树不空，对左子树递归
        wpl_PreOrder(root->lchild,deep+1);
    if(root->rchild != NULL)                    //若右子树不空，对左子树递归
        wpl_PreOrder(root->rchild,deep+1);
    return wpl;
}
```

② 基于层次遍历的算法

```
#define MaxSize 100                  //设置队列的最大容量
int wpl_LevelOrder(BiTree root){
    BiTree q[MaxSize];               //声明队列，end1 为头指针，end2 为尾指针
    int end1, end2;                  //队列最多容纳 MaxSize-1 个元素
    end1 = end2 = 0;                 //头指针指向队头元素，尾指针指向队尾的后一个元素
    int wpl = 0, deep = 0;           //初始化 wpl 和深度
    BiTree lastNode;                 //lastNode 用来记录当前层的最后一个结点
    BiTree newlastNode;              //newlastNode 用来记录下一层的最后一个结点
    lastNode = root;                 //lastNode 初始化为根节点
    newlastNode = NULL;              //newlastNode 初始化为空
    q[end2++] = root;                //根节点入队
    while(end1 != end2){                 //层次遍历，若队列不空则循环
        BiTree t = q[end1++];            //拿出队列中的头一个元素
        if(t->lchild == NULL & t->lchild == NULL){
            wpl += deep*t->weight;
        }                                //若为叶子结点，统计 wpl
        if(t->lchild != NULL){           //若非叶子结点把左结点入队
            q[end2++] = t->lchild;
            newlastNode = t->lchild;
        }                                //并设下一层的最后一个结点为该结点的左结点
        if(t->rchild != NULL){           //处理叶节点
            q[end2++] = 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;                   //层数加 1
```

```
        }
    }
    return wpl;                          //返回 wpl
}
```

第7章：排序

7.3 交换排序

2016：划分集合

- **题目描述**

已知由$(n \geq 2)$个正整数构成的集合$A=\{a_k\}, 0 \leq k < n$，将其划分为两个不相交的子集A_1和A_2，元素个数分别为n_1和n_2，A_1和A_2中元素之和分别为S_1和S_2。设计一个尽可能高效的划分算法，满足$|n_1-n_2|$最小且$|S_1-S_2|$最大。

- **算法思想**

将最小的$\lfloor n/2 \rfloor$个元素放在A_1中，其余的元素放在A_2中，分组结果即可满足题目要求。仿照快速排序的思想，基于枢轴将n个整数划分为两个子集。根据划分后枢轴所处的位置i分别处理：

① 若$i=\lfloor n/2 \rfloor$，则分组完成，算法结束；

② 若$i<\lfloor n/2 \rfloor$，则枢轴及以前的所有元素均属于A_1，继续对i之后的元素进行划分；

③ 若$i>\lfloor n/2 \rfloor$，则枢轴及之后的所有元素均属于A_2，继续对i之前的元素进行划分；

- **算法实现**

```
int setPartition(int a[],int n){
    int pivotkey,low=0,low0=0,high=n-1,high0=n-1,flag=1,k=n/2,i;
    int s1=0,s2=0;
    while(flag){
        pivotkey = a[low];      //选择枢轴
        while(low<high){
            while(low<high && a[high]>=pivotkey)        //基于枢轴对数据进行划分
                --high;
            if(low!=high)   a[low] = a[high];
            while(low<high && a[low]<=pivotkey)
                ++low;
            if(low!=high)   a[high]=a[low];
        }
        a[low]=pivotkey;
        if(low==k-1)        //若枢轴是第n/2个元素，划分成功
            flag=0;
        else{
            if(low<k-1){
                low0=++low;
                high=high0;
            }
            else{
                high0=--high;
```

```
                low=low0;
            }
        }
    }
}
```

- **算法性能分析**

平均时间复杂度：$O(n)$，空间复杂度为$O(1)$