

线性表

线性表的顺序表示和实现

```
1 //线性表格式
2 #define LIST_INIT_SIZE 100
3 typedef struct{
4     ElemType elem[LIST_INIT_SIZE] //复合型(用结构体定义的类型)用该类型的指针动态分
    配数组
5     int length;
6 }Sqlist;
7 Sqlist L;
8 L.data=(ElemType*)malloc(sizeof(ElemType)*MaxSize);
9
10 //注意! 逻辑位序与物理位序相差1
11
12 //多项式线性表
13 #define MAXSIZE 1000 //多项式可能达到的最大长度
14
15 typedef struct{ //多项式非零项的定义
16     float p; //系数
17     int e; //指数
18 }Polynomial;
19
20 typedef struct{
21     Polynomial *elem; //存储空间的基地址
22     int length; //多项式当前项的个数
23 }Sqlist; //多项式的顺序存储结构类型为Sqlist
24
25 //图书表
26 #define MAXSIZE 10000
27
28 typedef struct{
29     char no[20];
30     char name[50];
31     float price;
32 }Book;
33
34 typedef struct{
35     Book *elem;
36     int length;
37 }Sqlist;
38
39 //函数结果状态代码
40 #define TRUE 1
41 #define FALSE 0
42 #define OK 1
43 #define INFEASIBLE -1
44 #define OVERFLOW -2
45
46 //Status 是函数的类型, 其值是函数结果状态代码
47 typedef int Status;
```

顺序表基本操作

顺序表：逻辑结构与存储结构一致，可快速计算任何一个数据元素的存储位置，因此可粗略认为访问每个元素所花时间相等，这种存取元素的方法称为随机存取法。

没有占用辅助空间，空间复杂度 $S(n)=O(1)$

优点：1 存储密度大(结点本身所占存储量/结点结构所占存储量 = 1) 2 可以随机存取表中任一元素， $O(1)$

缺点：1 在插入、删除某一元素时，需要移动大量元素 2 浪费存储空间&静态存储形式，数据元素个数不能自由扩充

```

1  #define OK 1
2  #define OVERFLOW -2
3  #define ElemType xxx //依据需要填入
4  #define ERROR -1
5
6  typedef struct {
7      ElemType elem[LIST_INIT_SIZE];
8      int length;
9  } SqList;
10 //初始化线性表
11 Status InitList_Sq(SqList &L) {
12     L.elem = new ElemType[MAXSIZE];
13     if(!L.elem) exit(OVERFLOW);
14     L.length = 0;
15     return OK;
16 }
17
18 //销毁线性表
19 void DestroyList(SqList &L) {
20     if(L.elem) delete L.elem;
21 }
22
23 //清空线性表
24 void ClearList(SqList &L) {
25     L.length = 0;
26 }
27
28 //求线性表的长度
29 int GetLength(SqList L) {
30     return (L.length);
31 }
32
33 //判断线性表是否为空
34 int IsEmpty(SqList L) {
35     if(L.length == 0) return 1;
36     else return 0;
37 }
38
39 //顺序表的取值(根据位置i获取相应位置数据元素的内容)随机存取，o(1)
40 int GetElem(SqList L, int i, ElemType &e) {

```

```

41     if(i<1|| i>L.length) return ERROR;    //判断i值是否合理
42     e = L.elem[i-1];    //第i-1个单元存储第i个数据
43 }
44
45 //顺序表的查找，平均时间复杂度为O(n)
46 int LocateElem(SqList L,ElemType e) {
47 //在线性表中查找值为e的数据元素，返回其序号(是第几个元素)
48     //使用for循环
49     for(i=0; i<L.length; i++)
50         if(L.elem[i] == e) return i+1;//查找成功，返回序号
51     return 0;//查找失败，返回0
52     //使用while循环
53     i=0;
54     while(i<L.length && L.elem[i]!=e) i++;
55     if(i<L.length) return i+1;
56     return 0;
57 }
58
59 //顺序表的插入，平均时间复杂度为O(n)，插入位置+移动次数=n+1，插入位置有n+1种可能
60 Status ListInsert_Sq(SqList &L,int i,ElemType e) {
61     if(i<1|| i>L.length+1) return ERROR; //i值不合法，插入位置可以从第1个位置到第
n+1个位置，下标对应0~n
62     if(L.length==MAXSIZE) return ERROR; //当前存储空间已满
63     for(j=L.length-1; j>=i-1; j--)
64         L.elem[j+1]=L.elem[j];    //插入位置及之后的元素后移
65     L.elem[i-1]=e;    //将新元素e放入第i个位置
66     L.length++;    //表长增加1
67     return OK;
68 }
69
70 //顺序表的删除，平均时间复杂度为O(n)，删除位置+移动次数=n，删除位置有n种可能
71 Status ListDelete_Sq(SqList &L,int i) {
72     if((i<1)|| (i>L.length)) return ERROR; //i值不合法，删除位置只能是1~n
73     for(j=i; j<=L.length-1; j++)
74         L.elem[j-1]=L.elem[j];
75     L.length--;
76     return OK;
77 }

```

线性表的链式表示和实现

单链表

```

1  typedef struct LNode { //声明结点的类型和指向结点的指针类型
2      ElemType data;    //结点的数据域，多个数据项则预先定义为一个结构类型ElemType
3      struct LNode *next; //结点的指针域
4  } LNode,*LinkList;    //LinkList为指向结构体LNode的指针类型
5  //通常 定义链表L:LinkList L; 定义结点指针p:LNode *p;
6
7  //单链表的初始化(带头结点的单链表):生成新结点作头结点，用头指针L指向头结点，头结点指针域置空
8  Status InitList_L(LinkList &L) {
9      L=new LNode; //或L=(LinkList)malloc(sizeof(LNode));new获得的是指向新结点的
指针，把新结点的地址赋值给L

```

```

10     L->next=NULL;
11     return OK;
12 }
13
14 //判断链表是否为空(无元素, 头指针和头结点仍然存在)即判断头结点指针域是否为空
15 int ListEmpty(LinkList L) { //若L为空表, 则返回1, 否则返回0
16     if(L->next) //非空
17         return 0;
18     else
19         return 1;
20 }
21
22 //单链表的销毁: 链表销毁后不存在(头结点和头指针均不存在): 从头指针开始, 依次释放所有结点
23 Status DestroyList_L(LinkList &L) {
24     LNode *p;
25     while(L) { //判断头指针是否为空的简化写法, L!=NULL
26         p=L;
27         L=L->next;
28         delete p;
29     }
30     return OK;
31 }
32
33 //清空单链表: 从首元结点开始依次释放所有结点, 并将头结点指针域设置为空
34 //(链表仍存在, 但链表中无元素, 成为空链表, 头指针和头结点仍然存在)
35 Status ClearList(LinkList &L) {
36     LNode *p,*q; //或LinkList p,q;
37     p=L->next;
38     while(p) { //没到表尾
39         q=p->next;
40         delete p;
41         p=q;
42     }
43     L->next=NULL; //头结点指针域
44     return OK;
45 }
46
47 //求单链表的表长: 从首元结点开始, 依次计数所有结点
48 int ListLength_L(LinkList L) { //返回L中数据元素个数
49     LinkList p; //用LNode *p; 可读性更好
50     p=L->next; //p指向第一个结点
51     i=0;
52     while(p) { //遍历单链表, 统计结点数
53         i++;
54         p=p->next;
55     }
56     return i;
57 }
58
59 //获取线性表L中的某个数据元素的内容, 通过变量e返回(从第一个结点L->next顺链扫描, 用指针p
    指向当前扫描到的结点)
60 Status GetElem_L(LinkList L, int i, ElemType &e) {
61     p=L->next;
62     j=1; //初始化
63     while(p&& j<i) { //向后扫描, 直到p指向第i个元素或p为空

```

```

64     p=p->next;
65     ++j;
66 }
67 if(!p||j>i) return ERROR; //第i个元素不存在(元素位置超过长度p==NULL或是元素位置<1)
68 e=p->data; //取第i个元素
69 return OK;
70 }
71
72 //单链表的查找(时间复杂度O(n))
73
74 //按值查找:根据指定数据获取该数据所在的位置(该数据的地址)
75 LNode *LocateElem_L(LinkList L,ElemType e) {
76 //在线性表L中查找值为e的数据元素,找到则返回其地址,失败则返回NULL
77     p=L->next;
78     while(p&&p->data!=e)
79         p=p->next;
80     return p;
81 }
82
83 //按值查找:根据指定数据获取该数据所在的位置序号(是第几个数据元素)
84 int LocateElem_L(LinkList L,ElemType e) {
85 //返回L中值为e的数据元素的位置序号,查找失败返回0
86     p=L->next;
87     j=1;
88     while(p&&p->data!=e) {
89         p=p->next;
90         j++;
91     }
92     if(p) return j;
93     else return 0;
94 }
95
96 //插入和删除时间复杂度O(1),但由于要从头查找前驱结点,所耗时间复杂度为O(n)
97
98 //单链表的插入:在第i个结点前插入值为e的新结点
99 Status ListInsert_L(LinkList &L,int i,ElemType e) {
100     p=L;
101     j=0;
102     while(p&&j<i-1) { //寻找第i-1个结点
103         p=p->next;
104         ++j;
105     }
106     if(!p||j>i-1) return ERROR; //i大于表长+1或者小于1,插入位置非法
107     s=new LNode;
108     s->data=e; //生成新结点s,将结点s的数据域置为e
109     s->next=p->next; //将结点s插入L中
110     p->next=s;
111     return OK;
112 }
113
114 //单链表删除第i个结点:首先找到第i-1个结点的存储位置p(根据需要保存第i个结点的值),再令p->next指向第i+1个结点,释放第i个结点的空间
115 Status ListDelete_L(LinkList &L,int i,ElemType &e) {
116     p=L;

```

```

117     j=0;
118     while(p->next&& j<i-1) {
119         p=p->next;    //寻找第i个结点，并令p指向其前驱
120         ++j;
121     }
122     if(!(p->next)||j>i-1) return ERROR; //删除位置不合理
123     q=p->next; //临时保存被删结点的地址以备释放
124     p->next=q->next; //改变删除结点前驱结点的指针域
125     e=q->data; //保存删除结点的数据域
126     delete p; //释放删除结点的空间
127     return OK;
128 }
129
130 //单链表的建立
131
132 //头插法 时间复杂度O(n) 原先头结点后的所有结点接在新结点后，再将新结点接在头结点后
133 void CreateList_H(LinkList &L,int n) {
134     L=new LNode;
135     L->next=NULL; //先建立一个带头结点的单链表
136     for(i=n; i>0; --i) {
137         p=new LNode;
138         cin>>p->data;
139         p->next=L->next; //插入到表头
140         L->next=p;
141     }
142 }
143
144 //尾插法 时间复杂度O(n) 尾指针r指向尾结点，新结点插入到尾结点后，r指向新结点
145 void CreateList_R(LinkList &L,int n) { //正位序输入n个元素的值，建立带头结点的
    单链表L
146     L=new LNode;
147     L->next=NULL;
148     r=L; //尾指针r指向头结点
149     for(i=0; i<n; ++i) {
150         p=new LNode;
151         cin>>p->data; //生成新结点，输入元素值
152         p->next=NULL;
153         r->next=p; //插入到表尾
154         r=p; //r指向新的尾结点
155     }
156 }

```

循环链表

循环链表：是一种头尾相接的链表(即表中最后一个结点的指针域指向头结点，整个链表形成一个环)

优点：从表中任一结点出发均可找到表中其他结点

注意：由于循环链表中没有NULL指针，故涉及遍历操作时，其终止条件就不再像非循环链表那样判断p或p->next是否为空，而是判断它们是否等于头指针。p!=L 和 p->next!=L

尾指针表示单循环链表：a1的存储位置是：R->next->next an的存储位置是：R 时间复杂度均为O(1) 头指针表示则不够方便，找an的时间复杂度：O(n)

```

1 //带尾指针循环链表的合并(将Tb合并到Ta之后): p存表头结点; Tb表头连接到Ta表尾; 释放Tb表头;
  修改指针
2 LinkList Connect(LinkList Ta,LinkList Tb){ //假设Ta、Tb都是非空的单循环链表
3     p=Ta->next; //1、p存表头结点
4     Ta->next=Tb->next->next; //2、Tb表头连接到Ta表尾
5     delete Tb->next; //3、释放Tb表头结点
6     Tb->next=p; //修改指针
7     return Tb;
8 }//时间复杂度为O(1)

```

双向链表

双向链表：在单链表的每个结点里再增加一个指向其直接前驱的指针域prior，这样链表中就形成了有两个方向不同的链，故称为双向链表。

```

1 typedef struct DuLNode{
2     ElemType data;
3     struct DuLNode *prior,*next;
4 }DuLNode,*DuLinkList;
5
6 //双向链表的插入
7 void ListInsert_DuL(DuLinkList &L,int i,ElemType e){ //在带头结点的双向循环链表
  L中第i个位置之前插入元素e
8     if(!(p=GetElemP_DuL(L,i))) return ERROR;
9     s=new DuLNode; s->data=e;
10    s->prior=p->prior; p->prior->next=s;
11    s->next=p; p->prior=s;
12    return OK;
13 }
14
15 //双向链表的删除 //时间复杂度O(n)
16 void ListDelete_DuL(DuLinkList &L,int i,ElemType &e){ //删除带头结点的双向循环
  链表L的第i个元素，并用e返回
17     if(!(p=GetElemP_DuL(L,i))) return ERROR;
18     e=p->data;
19     p->prior->next=p->next;
20     p->next->prior=p->prior;
21     delete p;
22     return OK;
23 }

```

线性表的应用

线性表的合并

【问题描述】假设利用两个线性表La和Lb分别表示两个集合A和B，现要求一个新的集合A=A∪B。如：
La=(7,5,3,11), Lb=(2,6,3)→La=(7,5,3,11,2,6)

【算法步骤】依次取出Lb中的每个元素，执行以下操作：

- 1 在La中查找该元素
- 2 如果找不到，则将其插入La的最后

算法的时间复杂度是： $O(\text{ListLength}(\text{La}) * \text{ListLength}(\text{Lb}))$

```
1 void union(List &La, List Lb){
2     La_len=ListLength(La);
3     Lb_len=ListLength(Lb);
4     for(i=1; i<=Lb_len; i++){
5         GetElem(Lb, i, e);
6         if(!LocateElem(La, e)) ListInsert(&La, ++La_len, e);
7     }
8 }
```

有序表的合并

【问题描述】已知线性表La和Lb中的数据元素按值非递减有序排列，现要求将La和Lb归并为一个新的线性表Lc，且Lc中的数据元素仍按值非递减有序排列。如：La=(1,7,8), Lb=(2,4,6,8,10,11) → Lc=(1,2,4,6,7,8,8,10,11)

【算法步骤】

- 1 创建一个空表Lc
- 2 依次从La或Lb中"摘取"元素值较小的结点插入到Lc表的最后，直至其中一个表变空为止
- 3 继续将La或Lb其中一个表的剩余结点插入在Lc表的最后

用顺序表实现

【注意】：先取值，后自增。因为顺序表中元素在物理上也相邻，可以使用指针++，在链表中则不可以。

算法的时间复杂度是： $O(\text{ListLength}(\text{La}) + \text{ListLength}(\text{Lb}))$ 因为合并后没有剔除相同元素(或理解成取最坏情况)

算法的空间复杂度是： $O(\text{ListLength}(\text{La}) + \text{ListLength}(\text{Lb}))$ 因为Lc中元素的个数是两表元素之和

```
1 void MergeList_Sq(SqList LA, SqList LB, SqList &LC){ //合并结果通过LC返回
2     pa=LA.elem;
3     pb=LB.elem; //指针pa和pb的初值分别指向两个表的第一个元素
4     LC.length=LA.length+LB.length; //新表长度为待合并两表的长度之和
5     LC.elem=new ElemType[LC.length]; //为合并后的新表分配一个数组空间
6     pc=LC.elem; //指针pc指向新表的第一个元素
7     pa_last=LA.elem+LA.length-1; //指针pa_last指向LA表的最后一个元素
8     pb_last=LB.elem+LB.length-1; //指针pb_last指向LB表的最后一个元素
9     while(pa<=pa_last && pb<=pb_last){ //两个表都非空
10         if(*pa<=*pb) *pc++=*pa++; //依次"摘取"两表中值较小的结点
11         else *pc++=*pb++;
12     }
13     while(pa<=pa_last) *pc++=*pa++; //LB表已到达表尾，将LA中剩余元素加入LC
14     while(pb<=pb_last) *pc++=*pb++; //LA表已到达表尾，将LB中剩余元素加入LC
15 }
```


用链表实现

用La的头结点作为Lc的头结点

注意：插入剩余段的操作 `pc->next=pa?pa:pb;` 相当于 `if(pa) pc->next=pa; else pc->next=pb;`

算法的时间复杂度是： $O(\text{ListLength}(La) + \text{ListLength}(Lb))$ 考虑最坏情况，La和Lb中元素依次轮流加到Lc中

算法的空间复杂度是： $O(1)$ 因为直接在原先链表上利用指针进行操作(修改指针)，无需额外空间

```
1 void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc){
2     pa=La->next; pb=Lb->next;
3     pc=Lc=La; //用La的头结点作为Lc的头结点
4     while(pa && pb){
5         if(pa->data==pb->data){
6             pc->next=pa;
7             pc=pa;
8             pa=pa->next;
9         }
10        else{
11            pc->next=pb;
12            pc=pb;
13            pb=pb->next;
14        }
15    }
16    pc->next=pa?pa:pb; //插入剩余段
17    delete Lb; //释放的头结点
18 }
```

案例分析与实现

一元多项式的运算：实现两个多项式加、减、乘运算

$$P_n(x) = p_0 + p_1x + p_2x^2 + \cdots + p_nx^n$$

思路：将多项式的系数视为线性表 $P = (p_0, p_1, p_2, \cdots, p_n)$ 每一项的指数 i 隐含在其系数 p_i 的序号中

```
1 void PolyOperate(SqList &L1, SqList &L2, SqList &L3){
2     for (int i=0; i<L1.length && i<L2.length; ++i){
3         L3.elem[i]=L1.elem[i]+L2.elem[i];
4         L3.length+=1;
5     }
6     if (L1.length<=L2.length){
7         for (int j=L1.length; j<L2.length; ++j){
8             L3.elem[j]=L2.elem[j];
9             L3.length+=1;
10        }
11    }
12    else{
13        for (int j=L2.length; j<L1.length; ++j){
14            L3.elem[j]=L1.elem[j];
15            L3.length+=1;
16        }
17    }
```

```
17     }
18 }
```

稀疏多项式的运算

顺序存储结构

思路：只存储系数不为0的项，避免浪费空间。每一项存储 系数`a[i]` 和 指数，将其构成的序列记为线性表 $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_mx^{e_m}$$

【算法步骤】

- 1 创建一个新数组c
- 2 分别从头遍历比较a和b的每一项 { ✓ 指数相同，对应系数相加，若其和不为0，则在c中增加一个新项 × 指数不相同，则将指数较小的项复制到c中 }
- 3 一个多项式已遍历完毕时，将另一个剩余项依次复制到c中即可

缺点：存储空间分配不灵活，运算的空间复杂度高

链式存储结构

思路：将每一项视为链表中的一个结点，利用指针进行操作

```
1 typedef struct PNode{
2     float coef; //系数
3     int expn; //指数
4     struct PNode *next; //指针域
5 }PNode,*Polynomial;
```

【算法步骤】

- 1 创建一个只有头结点的空链表
- 2 根据多项式的项的个数n，循环n次执行以下操作
 - 生成一个新结点 *s
 - 输入多项式当前项的系数和指数赋给新结点 *s 的数据域
 - 设置一前驱指针 pre，用于指向待找到的第一个大于输入项指数的结点的前驱，pre 初值指向头结点
 - 指针 q 初始化，指向首元结点
 - 循链向下逐个比较链表中当前结点与输入项指数，找到第一个大于输入项指数的结点 *q
 - 将输入项结点 *s 插入到结点 *q 之前

```
1 void CreatePolyn(Polynomial &P,int n){ //输入m项的系数和指数，建立表示多项式的有序
    列表P
2     P=new PNode;
3     P->next=NULL; //先建立一个带头结点的单链表
4     for(i=1;i<=n;++i){ //依次输入n个非零项
5         s=new PNode; //生成新结点
6         cin>>s->coef>>s->expn; //输入系数和指数
7         pre=P; //pre用于保存q的前驱，初值为头结点
8         q=P->next; //q初始化，指向首元结点
9         while(q && q->expn<s->expn){ //找到第一个大于输入项指数的项*q
```

```

10         pre=q;
11         q=q->next;
12     }
13     s->next=q; //将输入项s插入到q和其前驱结点pre之间
14     pre->next=s;
15 }
16 }

```

3 多项式相加

- 指针p1和p2初始化，分别指向Pa和Pb的首元结点
- p3指向和多项式的当前结点，初值为Pa的头结点
- 当指针p1和p2均未到达相应表尾时，则循环比较p1和p2所指结点对应的指数值 (p1->expn与p2->expn)，有下列3种情况:
 - 当p1->expn == p2->expn时，则将两个结点中的系数相加 { ☒ 若和不为零，则修改p1所指结点的系数值，同时删除p2所指结点 ☐ 若和为零，则删除p1和p2所指结点 }
 - 当p1->expn < p2->expn时，则应摘取p1所指结点插入到“和多项式”链表中去
 - 当p1->expn > p2->expn时，则应摘取p2所指结点插入到“和多项式”链表中去
- 将非空多项式的剩余段插入到p3所指结点之后
- 释放Pb的头结点

```

1 void SPO_II(LinkList &La,LinkList &Lb,LinkList &Lc){
2     LNode *pa=La->next;
3     LNode *pb=Lb->next;
4     Lc=La;
5     LNode *pc=Lc;
6     while(pa && pb){
7         if(pa->data.index<pb->data.index){
8             pc->next=pa;
9             pc=pc->next;
10            pa=pa->next;
11        }
12        else if(pa->data.index>pb->data.index){
13            pc->next=pb;
14            pc=pc->next;
15            pb=pb->next;
16        }
17        else if(pa->data.index==pb->data.index){
18            pa->data.coef+=pb->data.coef;
19            pc->next=pa;
20            pc=pc->next;
21            pa=pa->next;
22            pb=pb->next;
23        }
24    }
25    pc->next=(pa?pa:pb);
26    delete Lb;
27 }

```

图书信息管理系统

图书顺序表

elem[0]	elem[1]	elem[2]	...	elem[length-1]	空闲区	
a ₁	a ₂	a ₃	...	a _{length}		

ISBN	书名	价格
------	----	----

图书链表



```
1 struct Book{
2     char id[20]; //ISBN
3     char name[50]; //书名
4     int price; //定价
5 };
```

顺序表

```
1 typedef struct{
2     Book *elem;
3     int length;
4 }SqList;
```

链表

```
1 type struct LNode{
2     Book data;
3     struct LNode *next;
4 }LNode,*LinkList;
```