线性结构的特点

在数据元素的非空有限集中：

- 存在唯一的一个被称作第一个的数据元素
- 存在唯一的被称作最后一个的数据元素
- 除第一个外，集合中的每个数据元素均只有一个前驱
- 除最后一个外，集合中的每个数据元素均只有一个后继

线性表的类型定义

线性表：具有相同特性的数据元素的一个有限序列

$$(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$

- a_1：起始结点
- i：下标，是元素的序号，表示元素在表中的位置
- n：元素总个数，即表长，$n=0$时称为空表

抽象类型线性表的定义
ADT List{
数据对象：$D = \{a_i | a_i \in Elemset, (i = 1, 2, \cdots, n, n \geq 0)\}$
数据关系：$R = \{< a_{i-1}, a_i > \| a_{i-1}, a_i \in D, (i = 1, 2, \cdots, n)\}$
基本操作：
InitList(&L)
操作结果：构造一个空的线性表L
DestroyList(&L)
初始条件：线性表L已存在
操作结果：销毁线性表L
ClearList(&L)
初始条件：线性表L已存在
操作结果：将L重置为空表
ListEmpty(L)
初始条件：线性表L已存在
操作结果：若L为空表则返回TRUE，否则返回FALSE
ListLength(L)
初始条件：线性表L已存在
操作结果：返回L中数据元素个数
GetElem(L, i, &e)
初始条件：线性表L已存在，$1 \leq i \leq ListLength(L)$

FxMarkBrown's Blog

操作结果：用e返回L中第i个数据元素的值
LocateElem(L, e, compare())
初始条件：线性表L已存在，compare()是数据元素判定函数
操作结果：返回L中第1个与e满足compare()的数据元素的位序。若这样的数据元素不存在，则返回值为0
PriorElem(L, cur_e, &pre_e)
初始条件：线性表L已存在
操作结果：若cur_e是L的数据元素，且不是第一个，则用pre_e返回它的前驱，否则操作失败，pre_e无定义
NextElem(L, cur_e, &next_e)
初始条件：线性表L已存在
操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继，否则操作失败，next_e无定义
ListInsert(&L, i, e)
初始条件：线性表L已存在，$i \leq i \leq ListLength(L)+1$
操作结果：在L中第i个位置之前插入新的数据元素，e,L的长度加1
ListDelete(&L, i, &e)
初始条件：线性表L已存在且非空，$i \leq i \leq ListLength(L)$
操作结果：删除L的第i个数据元素，并用e返回其值，L的长度减1
ListTraverse(L, vist())
初始条件：线性表L已存在
操作结果：依次对L的每个数据元素调用vist()，一旦vist()失败，则操作失败
} ADT List

其他操作比如求两个表的并集

```C++
void union(List &La, List Lb) {
   //将所有在线性表Lb中但不在La中的数据元素插入到La中
   La_len = ListLength(La); Lb_len = ListLength(Lb); // 求线性表的长度
   for(i = 1;i <= Lb_len, i++) {
       GetElem(Lb, i, e); // 取Lb中第i个数据元素赋给e
       if(!LocateElem(La, e, equal)) ListInsert(La, ++La_len, e);
   }
}
```

假设GetElem和ListInsert两个操作执行时间和表长无关，LocateElem执行时间和表长成正比，则复杂度$O(ListLength(LA) \times ListLength(LB))$

另外考虑数据元素均按值非递减有序排列的线性表LA和LB的合并，归并到一个新的表LC，且LC中的数据元素扔按值非递减有序排列，如

$$LA = (3,5,8,11)$$
$$LB = (2,6,8,9,11,15,20)$$

FxMarkBrown's Blog

则

$$LC = (2, 3, 5, 6, 8, 9, 11, 11, 15, 20)$$

```
void MergeList(List La, List Lb, List &Lc) {
    // 已知线性表La和Lb中的数据元素按值非递减排列
    // 归并La和Lb得到新的线性表Lc，Lc的数据元素也按值非递减排列
    InitList(Lc);
    i = j =1;k = 0;
    La_len = ListLength(La); Lb_len = ListLength(Lb);
    while((i <= La_len) && j <= Lb_len) { // La和Lb非空
        GetElem(La, i, ai); GetElem(Lb, i, bj);
        if(ai <= bj) { ListInsert(Lc, ++k, ai); ++i; }
        else { ListInsert(Lc, ++k, bj); ++j; }
    }
    while(i <= La_len) {
        GetElem(La, i++, ai); LisInsert(Lc, ++k, ai);
    }
    while(j <= Lb_len) {
        GetElem(Lb, j++, bj); ListInsert(Lc, ++k, bj);
    }
}
```

假设GetElem和ListInsert两个操作的执行时间与表长无关，LocateElem的执行时间和表长成正比，又假设La内最大的元素，即排在末尾的元素a_m比Lb内任意元素b_j要小，且La比Lb长度要短，则第一个while的复杂度为$O(ListLength(La))$，第二个while不执行（因为La内的元素已经全部插入进去），第三个while的复杂度为$O(ListLength(Lb))$，总复杂度为$O(ListLength(La) + ListLength(Lb))$

线性表的顺序表示和实现

线性表的元素

- 地址连续
- 依次存放
- 随机存取
- 类型相同

FxMarkBrown's Blog

定义

```C++
#define LIST_INIT_SIZE 100 // 线性表存储空间的初始分配量
#define LISTINCREMENT 10 // 线性表存储空间的分配增量
typedef struct {
    ElemType *elem //存储空间基址
    int length //当前长度
    int listsize //当前分配的存储容量（以sizeof(ElemType)为单位）
} SqList
```

初始化

```C++
Status InitList_Sq(SqList &L) {
    // 构造一个空的
    L.elem = (ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
    if(!L_elem) exit(OVERFLOW); // 存储分配失败
    L.length = 0; // 空表长度为0
    L.listsize = LIST_INIT_SIZE; // 初始存储容量
    return OK;
}
```

插入

```C++
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
    // 在顺序线性表L中第i个位置之前插入新的元素e
    // i的合法值为1<=i<=ListLength_Sq(L) + 1
    if(i < 1 || i > L.length + 1) return ERROR; // i值不合法
    if(L.length >= L.listsize) {
        // 存储空间已满，增加分配
        newbase = (ElemType*)realloc(L.elem, (L.listsize +
LISTINCREMENT)*sizeof(ElemType));
        if(!newbase) exit(OVERFLOW); // 存储分配失败
        L.elem = newbase; //新基址
        L.listsize += LSITINCREMENT; // 增加存储容量

    q = &(L.elem[i - 1]) // q为插入位置
    for(p = &(L.elem[L.length - 1]); p>= q, --p) *(p+1) = *p
    *q = e; //插入e
    ++L.length; //表长+1
    return OK;
}
```

计算其平均复杂度，最耗时的操作在移动元素上：

FxMarkBrown's Blog

- 最好情况：插在表尾，后移0个元素
- 最坏情况：插在表头，后移n个元素
- 概率：n元素可以插入的位置有n+1个，$p_i = \frac{1}{n+1}$

则期望为

$$E_{is} = \sum_{k=0}^{n} p_i \cdot 1 = \sum_{i=1}^{n+1} p_i \cdot (n-i+1)$$

计算结果为$\frac{n}{2}$，则平均时间复杂度为$O(n)$

删除

```C++
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
    // 在顺序表L中删除第i个元素，并用e返回其值
    // i的合法值为 1<=i<=ListLength_Sq(L)
    if (i < 1 || (i > L.length)) return ERROR; // i值不合法
    p = &(L.elem[i-1]); // 被删除元素的位置
    e = *p; // 被删除的元素赋给e
    q = L.elem+L.length-1; //指针右移到表尾元素
    for(++p;p <= q; ++p) *(p-1) = *p;
    --L.length; // 表长减1
    return OK;
}
```

计算其平均复杂度，最耗时的操作在移动元素上：

- 最好情况：删在表尾，前移0个元素
- 最坏情况：删在表头，前移n-1个元素
- 概率：n元素可以删的有n个，$q_i = \frac{1}{n}$

则期望为

$$E_{dl} = \sum_{k=0}^{n-1} q_i \cdot 1 = \sum_{i=1}^{n} q_i \cdot (n-i)$$

计算结果为$\frac{n-1}{2}$，则平均时间复杂度为$O(n)$

FxMarkBrown's Blog

查找

```C++
int LocateElem_Sq(SqList L, ElemType e, Status (*compare)(ElemType, ElemType)){
    // 在顺序线性表L中查找第1个值与e满足compare()的元素的位序
    // 若找到，则返回位序，否则返回0
    i = 1; // i的初值为第1个元素的位序
    p = L.elem; // p的初值为第一个元素的存储位置
    while(i <= L.length && !(* compare)(*p++, e)) ++i;
    if (i <= L.length) return i;
    else return 0;
}
```

计算其平均复杂度，最耗时的操作在遍历元素上：

- 最好情况：查在表尾，遍历1个元素
- 最坏情况：查在表头，遍历n个元素
- 概率：n元素可以查的有n个，$q_i = \frac{1}{n}$

则期望为

$$E_{loc} = \sum_{k=1}^{n} q_i \cdot 1 = \frac{n+1}{2}$$

计算结果为$\frac{n+1}{2}$，则平均时间复杂度为$O(n)$

合并

```C++
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {
    // 已知顺序线性表La和Lb的元素按值非递减排列
    // 归并La和Lb得到新的顺序表Lc，Lc的元素也按值非递减排列
    pa = La.elem; pb = Lb.elem;
    Lc.listsize = Lc.length = La.length+ Lb.length;
    pc = Lc.elem = (ElemType *)malloc(Lc.listsize * sizeof(ElemType));
    if(!Lc.elem) exit(OVERFLOW); // 存储分配失败
    pa_last = La.elem + La.length -1;
    pb_last = Lb.elem + Lb.length -1;
    while(pa <= pa_last && pb <= pb_last) { // 归并
        if(*pa <= *pb) *pc++ = *pa++;
        else *pc++ = *pb++;
    }
    while(pa <= pa_last) *pc++ = *pa++;
    while(pb <= pb_last) *pc++ = *pb++;
}
```

FxMarkBrown's Blog

顺序存储结构的特点

优点

- 逻辑关系相邻，物理位置相邻
- 可随机存取任一元素
- 存储空间使用紧凑

缺点

- 插入、删除操作需要移动大量的元素
- 分配空间需按固定大小分配、利用不充分

线性表的链式表示和实现

介绍

- 用一组任意的存储单元存储线性表的数据元素
- 利用指针表示数据元素之间的逻辑关系
- 每个数据元素，除存储本身信息外，还需存储其逻辑关系（直接后继元素）

链表的结点包括：

- 数据域：元素本身信息
- 指针域：指示直接后继的存储位置

特点

1. 结点在存储器上的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相等
2. 访问时只能通过头指针进入链表，并通过每个结点的指针域依次向后顺序扫描其余结点，所以寻找第一个结点和最后一个结点所花费的时间不同（顺序存储）

单链表

介绍

结点中只含一个指针域的链表

```c++
typedef struct LNode {
   ElemType data; // 结点的数据域
   struct LNode *next; // 结点的指针域
} LNode, *LinkList;
```

定义链表：`LinkList L`;
定义结点指针：`LNode *p`

FxMarkBrown's Blog

基本运算

随机取元素

```c++
Status GetElem_L(LinkList L, int i, ElemType &e) {
    // L为带头结点的单链表的头指针
    // 当第i个元素存在时，其值赋给e并返回OK，否则返回ERROR
    p = L->next; j = 1;
    while(p && j < i) {
        p = p->next;
        ++j;
    }
    if(!p || j > i) return ERROR; // 第i个元素不存在
    e = p->data; // 取第i个元素
    return OK;
}
```

时间复杂度：$O(n)$

插入元素

```c++
Status ListInsert_L(LinkList &L, int i, ElemType e) {
    p = L; j = 0;
    while(p && j < i-1) {
        p = p->next;
        ++j;
    }
    if(!p || j > i-1) return ERROR;
    s = (LinkList) malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return OK;
}
```

不用移动元素，但由于不能随机存取元素，故需查找第$i-1$个元素，算法的时间复杂度为$O(n)$

FxMarkBrown's Blog

删除元素

```C++
Status ListDelete(LinkList &L, int i, ElemType &e) {
    p = L; j = 0;
    while(p && j < i-1) {
        p = p->next;
        ++j;
    }
    if(!p || j > i-1) return ERROR;
    q = p->next;
    p->next = q->next;
    e = q->data;
    free(q);
    return OK;
}
```

动态建立链表（头插法）

```C++
void CreateList_L(LinkList &L, int n) {
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    for(i = n;i > 0; --i) {
        p = (LinkList)malloc(sizeof(LNode)); // 生成新结点
        scanf(&p->data);
        p->next = L->next; // 插入到表头
        L->next = p;
    }
}
```

两链表合并为一个链表

```C++
void MergeList_L(LinkList &La, &Lb, LinkList &Lc) {
    pa = La->next; pb = Lb->next;
    Lc = pc = La; // 用La的头结点作为Lc的头结点
    while(pa && pb) {
        if(pa->data <= pb->data) {
            pc->next = pa; pc = pa; pa = pa->next;
        } else {
            pc->next = pb; pc = pb; pb = pb->next;
        }
    }
    pc->next = pa ? pa : pb;
    free(Lb);
}
```

FxMarkBrown's Blog

循环链表（带尾指针，从尾部结点访问）

基本运算

查找

查首尾：$O(1)$

合并

复杂度$O(1)$

双向链表

定义

```C++
typedef struct DuLNode{
    ElemType data;
    struct DuLNode* prior;
    struct DuLNode* next;
} DuLNode, *DuLinkList;
```

基本运算

查找

查前继：$O(1)$

删除

```C++
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
    if(!(p = GetElemP_DuL(L, i))) return ERROR;
    e = p->data;
    p->prior->next = p->next;
    p->next-prior = p->prior;
    free(p);
    return OK;
}
```

FxMarkBrown's Blog

插入

```C++
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e) {
    if(!(p = GetElemP_DuL(L, i))) return ERROR;
    if(!(s = (DuLinkList)malloc(sizeof(DuLNode)))) return ERROR;
    s->data = e;
    s->prior = p->prior;
    p->prior->next = s;
    s->next=p;
    p->prior = s;
    return OK;
}
```

双向循环链表

头结点的前驱指针指向链表的最后一个结点
最后一个结点的后继指针指向头结点

静态链表

具有脸是存储的主要优点：插入删除时不需要移动元素
用一维数组表示，包含数据元素和指向下一个结点的索引数

应用：一元多项式的表示及相加

考虑多项式P、Q和相加得到的多项式R
例如$P(x)=10+5x-4x^2+3x^3+2x^4$
用数组表示，依次表示系数即可
但是对于稀疏多项式，比如$s(x)=1+3x^{100000}+2x^{200000}$
为了避免浪费，可以只存非零系数项，同时存储相应指数
此时线性表$P=((p_1,e_1),(p_2,e_2),\cdots,(p_m,e_m))$

FxMarkBrown's Blog