线性表

线性表的类型定义

一个线性表是n个**数据元素**的有限序列。至于每个数据元素的具体含义，在不同的情况下各不相同，它可以不同。

在稍复杂的线性表中，一个数据元素可以由若干个**数据项**（item)组成。在这种情况下，常把数据元素称为**记录**（record），含有大量记录的线性表又称**文件**（file）。

不是一维数组。线性表的长度可以改变，更像链表

（还包含基本操作，即调用函数。在定义线性表时要定义这些操作，怎么有些像面向对象编程了）

线性表的表示和实现

顺序表示

```
/*-----线性表的动态分配顺序存储结构-----*/
#define LIST_INT_SIZE 100
#define LISTINCREMENT 10
#define OK 0
#define ERROR 1
typedef int ElemType
typedef int Status

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

Status InitList_Sq(SqList *L ) {
  // 构造一个空的线性表
    L->elem = (ElemType*) malloc (LIST_INIT_SIZE * sizeof (ElemType));
    if (!L->elem)
        return ERROR;
    L->length = 0;
    L->listsize = LIST_INIT_SIZE
    return OK;
} // InitList_Sq

Status ListInsert_Sq(SqList *L, int i, ElemType e) {// 在顺序表L的第 i 个元素之前插入新的元素e
 // i 的合法范围为  1≤i≤L.length+1
    if (i < 1 || i > L->length+1)   return ERROR; // 插入位置不合法
    if (L->length > = L->listsize) { // 当前存储空间已满，增加分配
        newbase = (ElemType *)realloc(L->elem, (L->listsize + LIST_INCREMENT) * sizeof(ElemType));
        if (!newbase)   return ERROR;// 存储分配失败
        L->elem = newbase;                 // 新基址
        L->listsize += LIST_INCREMENT; // 增加存储容量
```

```
    }
    int q = &(L->elem[i-1]);                    // q 指示插入位置
    for (p = &(L->elem[L.length-1]); p > = q;  --p)
         *(p+1) = *p;         // 插入位置及之后的元素右移
    *q = e;         // 插入e
    ++L->length;    // 表长增1
    return OK;
} // ListInsert_Sq
```

链式表示

```
typedef struct  tagLNode {
    ElemType      data;  // 数据域
    struct tagLNode   *next;  // 指针域
} LNode, *LinkList;

Status ListInsert_L(LNode L, int i, ElemType e) {
     // L 为带头结点的单链表的头指针，本算法
     // 在链表中第i 个结点之前插入新的元素 e
    LNode p = L;
    j = 0;
    while (p && j<i-1) {
        p = p->next;
        ++j;
    }   // 寻找第 i-1 个结点
    if (!p || j > i-1)    return ERROR;      // i 大于表长或者小于1
    LinkList *s = (LinkList*) malloc ( sizeof(LNode));
                                // 生成新结点
    s->data = e;
    s->next = p->next;
    p->next = s; // 插入
    return OK;
  } // LinstInsert_L

```

循环链表和双向链表类似了。

第二章 线性表

2.1 线性表的定义与特征

- **定义**：线性表 (linear list) 是由n（$n \geq 0$）个数据元素组成的**有限序列**。
- **元素**：数据元素的具体含义在不同情况下各不相同，可以是一个数、一个符号，甚至是一个记录（如学生健康登记表中的一行）。
- **逻辑特征**：
    1. 集合中必存在唯一的“**第一元素**”。
    2. 集合中必存在唯一的“**最后元素**”。
    3. 除最后元素外，所有元素均有**唯一的后继**。
    4. 除第一元素外，所有元素均有**唯一的前驱**。

抽象数据类型 (ADT) 定义

```c
ADT List {
    数据对象: D = { aᵢ | aᵢ ∈ ElemSet, i=1, 2, ..., n, n ≥ 0 }
    // n 为线性表的长度，n=0 时为空表。

    数据关系: R1 = { <aᵢ₋₁, aᵢ> | aᵢ₋₁, aᵢ ∈ D, i=2, ..., n }
    // i 称为 aᵢ 在线性表中的位序。

    基本操作:
        // 初始化与销毁
        InitList(&L);        // 构造一个空的线性表L
        DestroyList(&L);     // 销毁线性表L
        ClearList(&L);       // 将L重置为空表

        // 引用型操作 (不修改表)
        ListEmpty(L);        // 判断L是否为空
        ListLength(L);       // 返回L中元素个数
        GetElem(L, i, &e);   // 用e返回L中第i个元素的值
        LocateElem(L, e, compare()); // 返回第一个满足compare()的元素的位序
        PriorElem(L, cur_e, &pre_e); // 返回cur_e的前驱
        NextElem(L, cur_e, &next_e); // 返回cur_e的后继
        ListTraverse(L, visit());    // 遍历线性表

        // 加工型操作 (修改表)
        PutElem(&L, i, e);       // 改变第i个元素的值为e
        ListInsert(&L, i, e);    // 在第i个位置前插入元素e
        ListDelete(&L, i, &e);   // 删除第i个元素，并用e返回其值
} ADT List
```

线性表操作的应用示例

- **示例1：合并集合 ($A = A \cup B$)**
  - **问题**：将线性表 `LB` 中存在而 `LA` 中不存在的元素插入到 `LA` 中。
  - **算法思想**：
    1. 遍历 `LB` 中的每个元素 `e`。
    2. 在 `LA` 中查找是否存在元素 `e`。
    3. 如果不存在，则将 `e` 插入到 `LA` 的末尾。
  - **核心代码逻辑**：

```c
// 该函数演示将Lb中独有的元素合并到La中
void union_list(List *La, List Lb) {
    int La_len = ListLength(*La);
    int Lb_len = ListLength(Lb);
    ElemType e;
    for (int i = 1; i <= Lb_len; i++) {
        GetElem(Lb, i, &e); // 取Lb中第i个数据元素赋给e
        // 如果在La中找不到e，返回0
        if (LocateElem(*La, e, equal) == 0) {
            // 则将e插入到La的表尾
```

```
            ListInsert(La, ++La_len, e);
        }
    }
}
```

- **示例3：合并有序表**
  - **问题**：已知两个非递减的有序表 `La` 和 `Lb`，构造一个新的非递减有序表 `Lc`，包含 `La` 和 `Lb` 的所有元素。
  - **算法思想**：
    1. 同时遍历 `La` 和 `Lb`。
    2. 比较 `La` 和 `Lb` 的当前元素，将较小者插入到 `Lc` 的末尾。
    3. 当一个表遍历完后，将另一个表中剩余的元素全部插入到 `Lc` 的末尾。
  - **核心代码逻辑**：

```c
// 合并两个有序线性表La和Lb，生成新的有序线性表Lc
void MergeList(List La, List Lb, List *Lc) {
    InitList(Lc);
    int i = 1, j = 1, k = 0;
    int La_len = ListLength(La);
    int Lb_len = ListLength(Lb);
    ElemType ai, bj;

    // 当两个表都未遍历完时，比较并插入
    while (i <= La_len && j <= Lb_len) {
        GetElem(La, i, &ai);
        GetElem(Lb, j, &bj);
        if (ai <= bj) {
            ListInsert(Lc, ++k, ai);
            i++;
        } else {
            ListInsert(Lc, ++k, bj);
            j++;
        }
    }

    // 插入La中剩余的元素
    while (i <= La_len) {
        GetElem(La, i++, &ai);
        ListInsert(Lc, ++k, ai);
    }

    // 插入Lb中剩余的元素
    while (j <= Lb_len) {
        GetElem(Lb, j++, &bj);
        ListInsert(Lc, ++k, bj);
    }
}
```

2.2 线性表的顺序实现 (顺序表)

- **定义**：用一组**地址连续**的存储单元，依次存放线性表中的数据元素。
- **地址计算公式**：$LOC(a_i) = LOC(a_1) + (i-1) \times C$

- `LOC(a₁)` 是基地址，`C` 是每个元素占用的存储量。
- 这个特性使得顺序表可以实现**随机存取**，即访问任意元素的时间复杂度为 $O(1)$。

- **C语言描述**：

```c
#define LIST_INIT_SIZE 100 // 线性表存储空间的初始分配量
#define LIST_INCREMENT 10  // 线性表存储空间的分配增量
#define OK 1
#define ERROR 0

typedef int ElemType;      // 假设元素类型为int
typedef int Status;        // 函数状态

typedef struct {
    ElemType *elem;    // 存储空间基址（一个指针）
    int      length;   // 当前长度
    int      listsize; // 当前分配的存储容量（以sizeof(ElemType)为单位）
} SqList;
```

> **[注释]**：这是一个动态分配的顺序表。当`length`接近`listsize`时，需要使用`realloc`函数增加存储空间。

顺序表基本操作实现

- **初始化操作** `InitList_Sq`

```c
Status InitList_Sq(SqList *L) {
    // 分配初始大小的存储空间
    L->elem = (ElemType*)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    // 内存分配失败
    if (!L->elem) return ERROR;
    // 空表长度为0
    L->length = 0;
    // 初始存储容量
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

- **插入操作** `ListInsert_Sq`
  - **步骤**：
    1. 检查插入位置 `i` 是否合法（$1 \le i \le length + 1$）。
    2. 检查存储空间是否已满，若满则增加分配。
    3. 将第 `i` 个位置及之后的所有元素向后移动一位。
    4. 将新元素 `e` 放入第 `i` 个位置。
    5. 表长 `length` 增1。
  - **时间复杂度**：$O(n)$。因为平均需要移动 $n/2$ 个元素。
  - **核心代码**：

```c
Status ListInsert_Sq(SqList *L, int i, ElemType e) {
    // 检查插入位置i的合法性
    if (i < 1 || i > L->length + 1) return ERROR;

```

```
    // 如果当前存储空间已满，则增加分配
    if (L->length >= L->listsize) {
        ElemType *newbase = (ElemType *)realloc(L->elem, (L->listsize + LIST_INCREMENT) * sizeof(ElemType));
        if (!newbase) return ERROR; // 存储分配失败
        L->elem = newbase;          // 新基址
        L->listsize += LIST_INCREMENT; // 增加存储容量
    }

    // q 指向插入位置
    ElemType *q = &(L->elem[i-1]);

    // 从最后一个元素开始，将插入位置及之后的元素右移一位
    for (ElemType *p = &(L->elem[L->length - 1]); p >= q; --p) {
         *(p + 1) = *p;
    }

    *q = e;          // 插入元素e
    ++L->length;     // 表长增1
    return OK;
}
```

- **删除操作** `ListDelete_Sq`
  - **步骤**:
    1. 检查删除位置 `i` 是否合法（$1 \le i \le length$）。
    2. 用`e`返回被删除元素的值。
    3. 将第`i+1`个位置及之后的所有元素向前移动一位。
    4. 表长`length`减1。
  - **时间复杂度**：$O(n)$。因为平均需要移动 $(n-1)/2$ 个元素。
  - **核心代码**:

```
Status ListDelete_Sq(SqList *L, int i, ElemType *e) {
    // 检查删除位置i的合法性
    if (i < 1 || i > L->length) return ERROR;

    // p 指向被删除元素的位置
    ElemType *p = &(L->elem[i-1]);
    // 将被删除元素的值赋给 e
    *e = *p;
    // q 指向表尾元素的位置
    ElemType *q = L->elem + L->length - 1;

    // 从被删除元素的下一个位置开始，所有元素左移一位
    for (++p; p <= q; ++p) {
        *(p - 1) = *p;
    }

    --L->length;     // 表长减1
    return OK;
}
```

2.3 线性表的链式实现 (单链表)

- **定义**：用一组**地址任意**的存储单元存放线性表中的数据元素，通过指针将它们连接起来。
- **结点 (Node)**：由**数据域**和**指针域**组成。
- **头指针 (Head Pointer)**：指向链表中第一个结点的指针。
- **头结点 (Head Node)**：为了操作方便，在第一个元素结点之前附加的一个结点。头结点的数据域可以不存储信息，但它的指针域指向第一个元素结点。

> **[注释]**：使用头结点的好处是，对第一个元素结点的操作（如插入、删除）与对其他结点的操作逻辑上可以统一，代码更简洁。空表和非空表的处理也一致。

- **C语言描述**：

```c
typedef struct LNode {
  ElemType      data;         // 数据域
  struct LNode  *next;        // 指针域
} LNode, *LinkList;           // LinkList为指向LNode的指针类型
```

单链表基本操作实现

- **按位查找** `GetElem_L`
  - **思想**：由于链表的非随机存取特性，必须从头指针开始，顺着`next`指针域逐个向后查找。
  - **时间复杂度**：$O(n)$。
  - **核心代码**：

```c
Status GetElem_L(LinkList L, int i, ElemType *e) {
    // L是带头结点的链表的头指针
    // p从第一个实际结点开始
    LinkList p = L->next;
    int j = 1; // 计数器

    // 顺着链表向后找，直到p指向第i个元素或p为空
    while (p && j < i) {
        p = p->next;
        ++j;
    }

    // 如果i值不合法（小于1或大于表长）或链表为空，则p会为空
    if (!p || j > i) return ERROR;

    // 取得第i个元素的值
    *e = p->data;
    return OK;
}
```

- **插入操作** `ListInsert_L`
  - **思想**：首先找到第`i-1`个结点`p`，然后生成新结点`s`，再修改指针完成插入。
  - **时间复杂度**：$O(n)$，主要耗时在查找第`i-1`个结点。
  - **核心代码**：

```c
Status ListInsert_L(LinkList L, int i, ElemType e) {
    // L为带头结点的单链表的头指针
```

```
    // p从头结点开始，寻找第i-1个结点
    LinkList p = L;
    int j = 0;
    while (p && j < i - 1) {
        p = p->next;
        ++j;
    }

    // i值不合法，或链表长度不够
    if (!p || j > i - 1) return ERROR;

    // 生成新结点s
    LinkList s = (LinkList)malloc(sizeof(LNode));
    if (!s) return ERROR; // 内存分配失败
    s->data = e;

    // 插入操作 (关键两步)
    s->next = p->next; // 1. s的后继指向p原来的后继
    p->next = s;       // 2. p的后继指向s
    return OK;
}
```

- **删除操作** `ListDelete_L`
  - **思想**：首先找到第`i-1`个结点`p`，然后修改指针跳过第`i`个结点`q`，并释放`q`的空间。
  - **时间复杂度**：$O(n)$，主要耗时在查找第`i-1`个结点。
  - **核心代码**：

```
Status ListDelete_L(LinkList L, int i, ElemType *e) {
    // L为带头结点的单链表的头指针
    // p从头结点开始，寻找第i-1个结点
    LinkList p = L;
    int j = 0;
    while (p->next && j < i - 1) { // p->next确保第i个结点存在
        p = p->next;
        ++j;
    }

    // 删除位置不合理
    if (!(p->next) || j > i - 1) return ERROR;

    // q指向待删除的第i个结点
    LinkList q = p->next;
    // p的后继指向q的后继，即跳过q
    p->next = q->next;
    // 用e返回被删除元素的值
    *e = q->data;
    // 释放q的空间
    free(q);
    return OK;
}
```

- **创建链表 (头插法)**
  - **思想**：逆位序输入n个元素，每次生成一个新结点，都将其插入到头结点之后。

- **核心代码**：

```c
void CreateList_L(LinkList *L, int n) {
    // 先建立一个带头结点的空链表
    *L = (LinkList)malloc(sizeof(LNode));
    (*L)->next = NULL;

    // 循环n次，每次创建一个新结点
    for (int i = 0; i < n; i++) {
        LinkList p = (LinkList)malloc(sizeof(LNode));
        printf("请输入第 %d 个元素值: ", n - i);
        scanf("%d", &(p->data)); // 假设输入整型数据

        // 将新结点p插入到头结点之后
        p->next = (*L)->next;
        (*L)->next = p;
    }
}
```

顺序表与链表的比较

特性	顺序表	链表
存取方式	随机存取 ($O(1)$)	顺序存取 ($O(n)$)
存储密度	密度高，等于1	密度低，有指针开销
空间分配	静态分配或一次性动态分配，可能造成浪费	动态分配，按需申请，空间利用率高
插入/删除	效率低 ($O(n)$)，需移动大量元素	效率高 ($O(1)$)，只需修改指针（不考虑查找）

2.4 其它形式的链表

- **循环链表**：表中最后一个结点的指针域指向头结点，形成一个环。
- **双向链表**：结点中除`next`指针外，还有一个`prior`指针指向其前驱结点。
  - **优点**：可以方便地找到前驱结点，对某个结点p进行操作（如删除）时，不需要像单链表那样必须从头查找p的前驱。
  - **缺点**：空间开销更大，插入和删除操作的指针修改更复杂。
  - **C语言描述**：

```c
typedef struct DuLNode {
    ElemType data;
    struct DuLNode *prior; // 指向前驱的指针
    struct DuLNode *next;  // 指向后继的指针
} DuLNode, *DuLinkList;
```

2.5 一元多项式的表示

- **问题**：对于稀疏多项式，如 $S(x) = 1 + 3x^{10000} - 2x^{20000}$，用一个大数组表示会浪费大量空间。
- **解决方案**：用一个线性表只存储非零项。每一项是一个二元组 `(系数, 指数)`。
- **实现**：可以使用一个**有序链表**来表示。链表按指数的升序或降序排列。
  - **优点**：节省空间，便于进行多项式的加法、减法等运算。
  - **C语言描述**：

```c
// 定义多项式的项
typedef struct {
    float coef; // 系数
    int   expn; // 指数
} Term;

// 多项式可以用一个有序链表来表示
// (其结点的数据域类型为Term)
// typedef OrderedLinkList Polynomial;
```

- **多项式加法** `AddPolyn`
  - **思想**：类似于有序表的合并。同时遍历两个多项式链表 `Pa` 和 `Pb`。
    1. 若当前两项指数相等，则系数相加，若结果不为0，则将新项插入到结果多项式 `Pc` 中。
    2. 若 `Pa` 的当前项指数小，则将其插入到 `Pc` 中，`Pa` 后移。
    3. 若 `Pb` 的当前项指数小，则将其插入到 `Pc` 中，`Pb` 后移。
    4. 当一个多项式遍历完后，将另一个多项式剩余的项全部插入到 `Pc` 中。