线性表（很经常考）

基本定义

由n个同类型的数据元素（结点）组成的有限序列。n为线性表长度

线性结构是最常用、最简单的数据结构，线性表是典型的线性结构。
基本特点：线性表中的的元素都是有序且有限的（一对一）。

有首尾元素，有前驱、有后继。

顺序表

顺序表的定义

按照逻辑顺序一次存放在一组地址连续的存储单元。（逻辑顺序和存储顺义一致）

数组具有**随机存取**的特性（存取时间与物理位置无关）：
LOC(ai) = LOC(a1) + (i-1)*l

查找方式：

1. 按位置查找
2. 按值查找

在高级语言（例如C）数组具有随机存储的特性，可以借助数组描述顺序表。

```
/*背下下面的例子*/

#define MAXSIZE 100
typedef struct sqList{  //定义线性表结构体（此处是一个匿名结构体）
    int data[MAXSIZE];  //线性表存储元素的数组
    int length; //记录线性表长度
}sqL, *sqLP;  //线性表名称

/*
三大特性：
1. 空间数组是data
2. 最大长度是MAXSIZE
3. 当前元素个数是length
*/
```

顺序表操作（初始化、增删改查、etc.）

- 初始化

```
bool Init_SqL(sqL *L){
    L->data = (int *)malloc(MAXSIZE * sizeof(int));      //定义了一个100*4B的空间，
首地址为L->data
    if(! L->data) return false;      //分配失败（内存中没有这么大的连续空间）
    else{
        L->length = 0;
        return true;
    }
}
```

- 插入

插入位置开始到后买你的元素均需要往后移动：

1. 移动
2. 放置元素

```
int ListInser(sqL *L, int i, int e){

    /*边界检查*/
    //线性表已满
    if(L->length == MAXSIZE) return 0;
    //当i比第一位置小，后壁最后一位置的后一位置还要大时
    if(i < 1 || i > L->length+1) return 0;

    /*移动*/
    //若插入位置不在表尾
    if(i <= L->length){
        //将要插入位置后的元素向后移一位，从后往前移动
        int k;
        for(k = L->length - 1; k >= i-1; k--)
            L->data[k+1] = L->data[k];
    }

    /*将新元素插入*/
    L->data[i-1] = e;
    L->length++;

    return 1;
}

/* 综上，平均移动次数为E=n/2 */
```

- 删除

删除元素开始后的元素均需要往前移动：

1. 删除元素
2. 移动

```
int ListDelete(sqL *L, int i, ElemType *e){

    /*边界检查*/
    //线性表为空
    if(L->length == 0) return 0;
    //删除位置不正确
    if(i < 1 || i > L->length) return 0;

    /*删除并移动*/
    *e = L->data[i-1];
    //若删除位置不是最后位置
    if(i < L->length){
    //将删除位置后的元素向前移一位，从前往后依次移动
        int k;
        for(k = i; k < L->length; k++)
            L->data[k-1] = L->data[k];
    }

    /*长度-1*/
    L->lenth--;

    return 1;
}

/* 平均移动次数 E=(n-1)/2, O(n) */
```

按值查找并删除：

1. 线性查找值为x的第一个元素，记录位置
2. 从该位置从前往后移动
3. 长度-1

```
void LocateDeleteSqList(sqL *L, int x){
    int i=0;
    while(i < L->length){
        //查找值为x的第一个结点
        if(L->data[i] != x) i++;
        else{
            //找到后移动
            int k;
            for(k = i+1; k < L->length; k++)
                L->data[k-1] = L->data[k];
            L->length--;
            break;
        }
    }
}

/*
时间主要好事在比较和移动的操作
```

```
平均比较次数E=(n+1)/2
平均删除移动次数E=(n-1)/2
合计操作平均时间复杂度为E=n，即为O(n)
*/
```

顺序表总结：**空间连续，随即访问，查找容易，删除难**

单链表

链式存储：用任意存储单元存储线性表中的数据元素
除了存储每个结点的数值，还要存储直接后继结点的地址，称为指针或链：
data：存储数值；next：存储地址

为了操作方便，第一个结点之前设置一个头结点（不存储任何信息），head指向第一个结点
· 头指针：指向头结点
· 头结点：不存放数据，虽然不是必须，但一般都得有
· 第一个数据结点：第一个存放数据的结点

有头结点，单链表判空条件：p->next == null

```
typedef struct Lnode{
    int data;   //数据域
    struct Lnode *next; //指针域
}Lnode, *LinkList; //结点类型，前者用于定义数据结点，后者用于定义头结点
```

结点通过动态分配和释放来实现，需要时分配，不需要时释放

实现时分别使用c提供的标准函数：

1. malloc()，分配
2. realloc()，重新分配
3. sizeof()，大小
4. free(), 释放

动态分配：
//函数malloc分配了一个类型为LNode的结点变量空间，并将其是地址放在指针变量p中
p = (LNode*)malloc(sizeof(LNode));

动态释放 free(p); 系统回收由指针变量p所指向的内存区，p必须是最近一次调用malloc函数时返回的数值

单链表操作（链式结构）【背】

赋值

```
LNode *p;
p = (LNode*)malloc(sizeof(LNode));
p->data = 20;
p->next = null;
```

```
/*
  p
| 20 | NULL |
*/
```

常见指针操作

1. q=p; // 操作前：p=>a；操作后：p=>a, q=>a。
2. q=p->next; // 操作前：p=>a, a、b；操作后：p=>a, q=>b, a、b。
    - 此时p与q的关系：p是q的直接前去，q是p的直接后继
3. p=p->next; // 操作前：p=>a, p->next=>b；操作后：p=>b。
    - 工作指针：用来挨个指向单链表中的每一个结点，以实现对单链表结点的操作
4. 插入一个结点（后插法）; // 操作前：p=>a, a、b；操作后：a、c、b。
    1. 找前驱
    2. 防断链（先右后左）
        - 对于待插入的结点c(指针q)，插入位置为指针p后面：
          q->next = p->next; //也就是让q指向原先p的下一个结点
          p->next = q;
5. 删除一个结点; // 操作前：p=>a，a、b、c；操作后：p=>a，a、c。
    - 一个会导致**内存泄漏**的操作：p->next = p->next->next; //此时原p->next成为野结点
    - 正规操作：
      q = p->next;
      p-next = q;
      free(q);

创建单链表

- 头插法建表

创建表时，每次插入的结点都作为第一个结点
创建过程： 1. 创建头 2. 起循环： 1. 创建 2. 插入

```
LNode *create_LinkList(void){
    int data;
    LNode *head, *p;
    head = (LNode *)malloc(sizeof(LNode));
    head->next = NULL;  //创建头
    while(1){
        scanf("%d", &data);
        if(data == NULL) break;
        p = (LNode *)malloc(sizeof(LNode));
```

```
        p->data = data; //数据域赋值
        /*钩链，新创建的结点总作为第一个结点。下方操作遵循先右后左*/
        p->next = head->next;
        head->next = p;
    }
    return(head);   //链表表头作为返回值
}

/*
头插法的一个致命问题：创建的链表是逆序的（简称头逆）
*/
```

- 尾插法建表

创建表时，每次插入的结点作为链表的表尾
需要引入尾指针，尾指针指向尾结点（尾结点是一个数据结点）

```
LNode *creat_LinkList(void){
    int data;
    LNode *head, *p, *q;
    head = p = (LNode *)malloc(sizeof(LNode));
    p->next = NULL; //创建头、尾结点
    while(1){
        scanf("%d", &data);
        if(data == NULL) break;
        q = (LNode *)malloc(sizeof(LNode));
        q->data = data; //数据域赋值
        /*钩链，新建的结点总是作为最后一个结点*/
        q->next = p->next;
        p->next = q;
        p = q;
    }
    return(head);   //链表表头作为返回值
}

/*
尾插法创建的表即为顺序
*/
```

单链表查找

- 按序号查找，取出链表中第i个元素（从头开始遍历表）

通过引入工作指针来一个个访问

```
int GetElem_L(LNode *L, int i){
    /*
    L为带头结点的单链表头指针
    当第i个元素存在，其赋值为e并返回OK，否则返回ERROR
```

```
    */
    LNode *p;
    p = L->next;
    int j = 1;
    /*工作指针不为空，则链表不为空*/
    while(p && j < i){
        p = p->next;
        ++j;
    }
    if(!p || i) return 0;
    int e = p->data;
    return e;
}
```

- 按值查找

```
LNode *Locate_Node(LNode *L, int key){
    LNode *p = L->next;
    while(p!=NULL && p->data != key) p = p->next;
    if(p->data == key) return p;
    else{
        printf("索要查找的结点不存在");
        return(NULL);
    }
}
```

插入

插入位置为i，需要找到第i-1个元素，以i-1元素为直接前去，做插入

```
int FindElem_L(LNode L, int i){
    LNode *p = L;
    int j = 0;
    while(p && j < i-1){
        //寻找位置
        p = p->next;
        ++j;
    }
    if(!p || j > i-1) return 0;
    /*执行插入操作*/
    //创建节点
    LNode *s;
    s = (LinkedList)malloc(sizeof(LNode));
    s->data = e;
    //插入
    s->next = p->next;
    p->next = s;
    return 1;
}
```

删除

- 按位置删除

找前驱，防断链。删除位置为i，找到i-1个元素，然后做删除

```
int DeleteElem_L(LNode L, int i){
    LNode *p = L;
    LNode *q;
    int j = 0;
    while(p->next && j < i-1){
        //寻找位置
        p = p->next;
        ++j;
    }
    if(!(p->next) || j > i-1) return 0; //删除位置不合理
    /*执行插入操作*/
    q = p->next;
    p->next = q->next;
    free(q);
    return 1;
}
```

- 按值删除

找前驱，防断链，引入结对指针：让p是q的直接前驱，让q是p的直接后继

```
void Delete_LinkList(LNode *L, int key){
    LNode *p = L, *q = L->next;
    while(q != NULL && q->data != key){
        p = q;
        q = q->next;
    }
    if(q->data == key){
        p->next = q->next;
        free(q);
    } else {
        printf("所要删除的节点不存在\n");
    }
}
```

- 变形1：把所有值为key的结点都删除

对每个结点进行检查，值为key则删除，然后继续检查下一个结点，知道所有节点都被检查到。

```
void Delete_LinkList_Node(LNode *L, int key){
    LNode *p = L, *q = L->next;
```

```
    while(q != NULL){
        if(q->data == key){
            //在此处对匹配的结点删除
            p->next = q->next;
            free(q);
            q = p->next;
        } else {
            //若不匹配，则结对指针后移
            p = q;
            q = q->next;
        }
    }
}
```

- 变形2：把所有相等的元素去掉

```
void Delete_Node_value(LNode *L){
    LNode *p = L->next, *q, *ptr;
    /*在该算法里，p为基准（key），q和ptr作为结对指针*/
    while(p != NULL){
        q = p, ptr=p->next;
        while(ptr != NULL){
            if(ptr->data == p->data){
                q->next = ptr->next;
                free(ptr);
                ptr = q->next;
            } else {
                q = ptr;
                ptr = ptr->next;
            }
        }
        p = p->next;
    }
}
```

单链表合并

现有两个有序单链表La、Lb，合并为Lc为表头的有序链表

- 双指针，不回溯
- 尾插法：
    - 谁小，谁尾插
    - 谁小，谁后移
    - 相等即删除

```
LNode *erge_linkList(LNode *La, LNode *Lb){
    LNode *Lc, *pa, *pb, *pc, *ptr;
    Lc = La;
```

```
    pc = La;
    pa = La->next;
    pb = Lb->next;
    while(pa != NULL && pb != NULL){
        /*谁小谁尾插，谁小谁后移*/
        if(pa->data < pb->data){
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        } else if(pa->data > pb->data){
            pc->next = pb;
            pc = pb;
            pb=pb->next;
        } else if(pa->data == pb->data){
            //相等即删除
            pc->next = pa;
            pc = pa;
            pa = pa->next;
            ptr = pd;
            pb = pb->next;
            free(ptr);
        }
    }
    if(pa != NULL) pc->next = pa;
    else pc->next=pd;   //将剩余结点连接上
    free(Lb);
    return(Lc);
}
```

循环链表

尾指针指向头结点

判断是否是空链表：head->next == head;
判断是否表尾结点：p->next == head;

双向链表

每个节点中设立两个指针域，prior指向前驱，next指向后继
(若在此基础上首位连接也可构成双向循环链表)

prior | data | next

(p->prior)->next == p == (p->next)->prior

插入操作

- 后插

1. 找前驱
2. 防断（先右后左）
    1. p->next->prior = s;

2. s->next = p->next;
3. s->prior = p;
4. p->next = s;

- 前插

1. 找后继
2. 防断链（先左后右）
    1. p->prior->next = s;
    2. s->prior = p->prior;
    3. p->prior = s;
    4. s->next = p;

删除操作

断链，释放节点

```
p->prior->next = p->next;
p->next->prior = p->prior;
free(p);
```

- 后删

删除其直接后继

1. 找前驱
2. 防断链

```
q = p->next;
p->next = q->next;
p = q->next->prior;
free(q);
```

- 前删

```
q = p->prior;
p->prior = p->prior;
q->prior->next = p;
free(q);
```

静态链表（顺序结构和链式结构的组合体）

用数组来实现链表

1. 使用结构体数组，内含指针域cur和数据域data
2. 一个数组分量表示一个结点，用cur表示结点在数组中相对位置

增删改查只需修改指针即可

链式结构和顺序结构的比较

比较内容	链式结构（重点）	顺序结构
实现形式	结构体	数组
存储空间	可以连续，可以离散	连续
存储效率	低	高
插入元素	无需移动元素	需要移动元素
删除元素	无需移动元素	需要移动元素
查找	顺序查找	顺序和随机存储
扩展	按需扩展	扩展困难

补充细节内容

该部分内容是做例题时的错题知识点

- 线性链表访问第i个元素的时间复杂度为O(n)
- 顺序表访问节点是随机访问，时间复杂度为O(1)；增加、删除结点需要移动大量元素，时间复杂度为O(n)
- 一个时间复杂度为O(1)的顺序表逆置算法

```
/*
该算法利用收尾元素下标和为length-1的特性进行处理
数组逆置策略
*/
void reverse(SqList &L){
    ElemType x;
    for(int i=0; i<L.length/2; i++){
        x = L.data[i];
        L.data[i] = L.data[L.length-i-1];
        L.data[L.length-i-1]=x;
    }
}

/*
如果做的是部分逆序
例如A[to, from]
则是 (to+i,  from-i) 的区间
*/
for(int i=0; i<(to-from+1)/2; i++){
    sawp(); //交换操作
}
```

- 一个高效的奇数提前、偶数放后的算法

```
/*
从左往右找偶数，从右往左找奇数
两边都找到后做交换
直到二者相遇
*/
void oddPreEven(SqList &L){
    int i=0; j=L.length-1, k;
    ElemType temp;
    while(i <= j){
        while(L.data[i]%2 == 1) i++;    //指向偶数
        while(L.data[j]%2 == 0) j--;    //指向奇数
        if(i<j){
            //交换
            temp = L.data[i];
            L.data[i] = L.data[j];
            L.data[j] = temp;
        }
    }
}
```

- 去除单链表中的重复元素

```
void deleteNodeValue(LNode *L){
    LNode *p = L->next, *q, *qtr;
    while(p != NULL){
        *q = p, *ptr = p->next;
        while(ptr != NULL){
            if(ptr->data == p->data){
                q->next = ptr->next;
                free(ptr);
                ptr = q->next;
            } else {
                q = ptr;
                ptr = ptr->next;
            }
        }
        p = p->next;
    }
}
```

- 将一个单链表逆序，可以考虑头插法逆序的特点，来将表倒置
- 时间复杂度为O(1)的顺序表删除指定元素的算法

```
void delLnode(SqList &L, int x){
    int k=0, i=0;
    while(i<L.length){
        if(L.data[i] == x) k++; //如果是要删除的元素，继续遍历
        //如果不是删除元素，则将其放在已经处理好的在最后一个不为x的位置
```

```
        else L.data[i-k] = L.data[i];
        i++;
    }
    L.length -= k;  //删除k个元素，则长度-k
}
```