线性表

2.1 概念

> 线性表：具有相同数据类型的 n ($n \geq 0$)个数据元素的有限序列
>
> $n = 0$ 为 空表

表示：

> L = $(a_1, a_2 \ldots, a_i, a_{i+1}, \ldots, a_n)$

（1）存在唯一一个被称为“第一个”的数据元素—— a_1： 表头元素

（2）存在唯一一个被称为“最后一个”的数据元素——a_n： 表尾元素

（3）除第一个之外，集合中的每个数据元素只有一个 直接前驱

（4）除最后一个外，集合中每个数据元素只有一个 直接后继

2.1.1 特点

1. 数据元素个数有限
2. 元素间 有逻辑上的顺序性
3. 表中元素 都是数据元素
   - 元素具有抽象性：只讨论元素间的逻辑关系，不考虑具体表示什么
4. 表中元素 数据类型相同
   - 单个元素占用的存储空间相同

> 当数据元素由若干数据项组成：
>
> - 记录：数据元素
> - 文件：含有大量记录的线性表

2.1.2 线性表是一种逻辑结构

> 表示元素之间一对一的相邻关系

实现线性表的两种存储结构

- 顺序存储
- 链式存储

线性表操作的实现

- 存储结构不同，算法实现也不同
- &：C++引用

2.2 线性表的存储

2.2.1 线性表的顺序存储——顺序表

位序与下标的区别

- 位序： $1 \leq i \leq length$
- 下标： $0 \leq i \leq length - 1$

动态分配

- 动态分配属于 顺序存储 结构，分配 n 个空间时仍需要 n 个连续存储空间

1. 特点

- 随机存取—— $O(1)$

  $Loc(a_i) = Loc(a_0) + (i-1) * sizeof(ElemType)$
- 存储密度高——只存数据元素
- 存储关系映射逻辑关系
- 插入删除效率低

2. 结点定义

```
# define MaxSize 20
typedef struct {/* 静态分配 */
    ElemType data[MaxSize];
    int length;
}SqList;
# define INITSIZE 10//初始空间容量
# define INCREMENT 10//增量
typedef struct {/* 动态分配 */
    ElemType *elem; //存储空间基址
    int length;     //线性表当前长度
    int listSize;   //存储容量
}SqList;
```

3. 基本操作实现

```
//动态分配 初始化一个空表
Status ListInit_sq(SqList &L){
    L.elem = (ElemType *)malloc(INITSIZE*sizeof(ElemType));
    if(!L.elem)
        return OVERFLOW;
    L.length = 0;
    L.listSize = INITSIZE;
    return OK;
}

Status ListInsert_Sq(SqList &L, int i, ElemType x) {
    //1.判断输入是否正确
    if (i < 1 || i > L.length + 1)
        return ERROR;
    //2.判断表空间是否充足
    if(L.length >= L.listSize){
        ElemType* newbase = (ElemType *)realloc(L.elem,
(L.listSize+INCREMENT)*sizeof(ElemType));
        if(!newbase)
            exit(OVERFLOW);
        L.elem = newbase;
        L.listSize += INCREMENT;
    }
    //3.插入位置元素与之后元素要后移
    ElemType* p = &(L[i-1]);
    for(ElemType* p = &(L[length-1]);p>=q;)
        *(p+1) = *p;
    //4.插入数据
    *q = x;
    L.length++;
    return OK;
}

Status ListDelete_Sq(SqList &L, int i, ElemType &x) {
    //1.判断输入是否合法
    if(i < 1 || i > L.length)
        return ERROR;
    //2.找到删除位置
    ElemType* p = &(L[i-1]);
```

```
    x = L.data[i - 1];//返回待删除元素
    //3.删除元素
    while(p<&(L[L.length-1])){
        *p = *(p+1);//从第i个元素，将其后继元素前移一位
        p++;
    }
    L.length--;
    return OK;
}
```

	插入	删除	按值查找
最好	表尾插入，不移动元素 $O(1)$	表尾删除，不移动元素 $O(1)$	遍历一次 $O(1)$
最坏	表头插入，移动 n 个元素 $O(n)$	表头删除，移动 $n-1$ 各元素 $O(n)$	表尾，$O(n)$
期望	$\sum_{i=1}^{n} p_i(n-i+1)$	$\sum_{i=1}^{n} q_i(n-i)$	
平均移动次数	$\frac{1}{n+1}\sum_{i=1}^{n}(n-i+1)=\frac{n}{2}$	$\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$	
平均时间复杂度	$O(n)$	$O(n)$	$O(n)$
空间复杂度	$O(1)$	$O(1)$	

> 假设对每个元素访问的等概率,即期望中的概率为算数平均数

4. 顺序表应用

- 插入总是发生在顺序表尾
- 顺序表的修改操作，一定会涉及遍历元素
  - 只要是顺序遍历，时间复杂度不会低于O(n)
  - 最短时间遍历一般要用 折半查找

H5 两表合并

H6 顺序表合并返回顺序表

> 思路：短表的下标为结果表的下标
>
> 时间复杂度：$O(n)$

```
bool Merge(SqList A, SqList B, SqList &C) {//合并两表
    if (A.length + B.length > C.MaxSize + 1)
        return false;//超长退出
    int i = 0, j = 0, k = 0;
    while (i < A.length && j < B.length) {
        if (A.data[i] <= B.data[j]) //两两比较，小者插入
            C.data[k++] = A.data[i++];
        else
            C.data[k++] = B.data[j++];
    }
    //有一表为遍历完情况
    while (i < A.length)
        C.data[k++] = A.data[i++];
    while (j < B.length)
        C.data[k++] = B.data[j++];

    C.length = k;
    return true;
}
```

H5 快慢指针

H6 有序顺序表去重

> 思路：快慢指针，i为慢指针，结果表游标
>
> 时间复杂度：$O(n)$
>
> 空间复杂度：$O(1)$

```
bool DeleteDuplicate(SqList &L) {//从有序顺序表删除重复值
    if (L.length == 0)
        return false;
    int i, j;//j为工作指针，遍历每个元素
    for (i = 0, j = 1; j < L.length; ++j)
        if (L.data[j] != L.data[i])
            L.data[++i] = L.data[j];
    L.length = i + 1;
    return true;
}
```

H6 删除指定元素

> 思路：直接定义快慢指针；
>
> 时间复杂度：$O(n)$

```
void DelX1(SqList &L,ElemType x) {
    //快慢指针，删除L中所有值为x的值，时间复杂度O(n),空间复杂度O(1)
    int k = 0;//慢指针，指示结果数组长度
    int i;//快指针，工作指针

    for (i = 0; i < L.length; ++i) {
        if (L.data[i] != x) {
            L.data[k++] = L.data[i];
        }
    }
    L.length = k;
}
```

> i：快指针
>
> k：记录连续为x的个数，将不为x的值插入到表尾；
>
> $i-k$：慢指针

```
void DelX2(SqList &L, ElemType x) {
    int k = 0; //x的数量
    int i = 0; //工作游标
    while(i < L.length) {
        if (L.data[i] == x)
            k++;
        else
            L.data[i-k] = L.data[i];
        i++;
    }
    L.length = L.length - k;
}
```

H6 删除有序表指定范围元素

```
bool del_s_t(SqList &L,ElemType s,ElemType t){
    int i,j;

    if(s > t || L.length == 0)
        return false;
    for(i=0;i < L.length && L.data[i] < s;i++);//找下界对应的下标
    if(i >= L.length)
        return false;
    for(j=i;j < L.length && L.data[j] <=t;j++);//找上界对应的下标
    if(j >= L.length)
        return false;

    //前移
    while(j < L.length)
        L.data[i++] = L.data[j++];//i: 慢指针; j: 快指针

    L.length = i;
}
```

H6 删除无序表指定范围元素

> i：快指针
>
> k：记录范围内值的个数，将范围内的值插入表尾
>
> $i-k$：慢指针

```
bool del_s_t(SqList &L,ElemTtpe s,ElemType t){
    int i;
    int k = 0;//k

    if(L.length == 0 || s >= t)
        return false;

    for(i = 0;i < L.length;++i){
        if(L.data[i] >= s && L.data <= t)
            k++;
        else
            L.data[i-k] = L.data[i];
    }
    L.length = k;
    return true;
}
```

H6 无序表去重

> 排序：$O(nlogn)$ 遍历：$O(n)$

```
int Partition(ElemType a[],int low,int high){
    ElemType pivot = a[low];
    while(low < high){
        while(low < high && a[high] > pivot)
            --high;//找到第一个比枢轴小的位置
        a[low] = a[high];
        while(low < high && a[low] < pivot)
            ++low;//找到第一个比枢轴大的位置
    }
    a[low] = pivot;
    return low;
}
```

```
void QuickSort(ElemType a[],int low,int high){
    if(low < high){
        int pivotpos = Partition(a,low,high);
        QucikSort(a,low,pivotpos-1);
        QucikSort(a,pivotpos+1,high);
    }
}
bool Union(SqList &L){
    QuickSort(L.data,0,L.length-1);
    for(int i = 1,j = 0;i < L.length;++i){
        //i为快指针，j为慢指针
        if(L.data[i] != L.data[j])
            L.data[++j] = L.data[i];
    }
}
```

H5 遍历

H6 删除最小值

> 思路：一次遍历，记录变量
>
> 时间复杂度：$O(n^2)$

```
bool DelMin(SqList &L ,ElemType &e) {//删除最小值,并用最后一个元素填充
    if (L.length == 0)
        return false;
    e = L.data[0];

    int pos = 0;
    for (int i = 0; i < L.length; ++i) {
        if (L.data[i] < e) {
            e = L.data[i];
            pos = i;
        }
    }
    //已知元素不重复时，可以这么做
    L.data[pos] = L.data[L.length - 1];
    L.length--;
    return true;
}
```

H6 无序顺序表去重

> 思路：sum记录表长。逐个遍历，查找结果表中是否存在
>
> 时间复杂度：$O(n^2)$

```
bool Union(SqList &L) {
    if (L.length == 0)
        return false;
    //sum 为结果串的长度，i为结果串下标,j为待和合并串下标
    int i, j, sum = 1;

    while (j < L.length) {
        //遍历结果串，看是否已存在当前值
        for (i = 0; i < sum; ++i) {
            if (L.data[i] == L.data[j])
                break;
        }
        if (i == sum)//如果不存在，则插入
            L.data[sum++] = L.data[j++];
        else//若存在则比较下一个
            j++;
    }
    L.length = sum;

    return true;
}
```

H5 折半查找

H6 折半查找

> 要求：有序线性表，查找x,
>
> - 若有，则与后继交换
> - 若无，则添加使仍为正序

```
void BinarySearch(SqList L, ElemType x) {
    int low, high = L.length - 1,mid;

    while (low <= high) {
        mid = (low + high) / 2;
        if (L.data[mid] == x)
            break;
        else if (L.data[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    if (L.data[mid] == x && mid != L.length - 1) {//中间插入
        ElemType t = L.data[mid];
        L.data[mid] = L.data[mid + 1];
        L.data[mid + 1] = t;
    }
    if (low > high) {//无，则插入
        int i;
        for (i = n - 1; i > high; i--)
            L.data[i + 1] = L.data[i];
        L.data[i + 1] = x;
    }
}
```

H6 原地转置——[折半]

> 时间复杂度：$O(n)$ 空间复杂度：$O(1)$

```
void Reverse(SqList &L) {
    ElemType e;
    for (int i = 0; i < L.length / 2; ++i) {
        e = L.data[i];
        L.data[i] = L.data[L.length - 1 - i];
        L.data[L.length - 1 - i] = e;
    }
}
```

转置应用

> 要求：$L.data[m+n]$ 中存放的元素，将$L.data$转置，然后前m个转置，后n个转置
>
> $a_1, a_2, a_3 \ldots a_n, b_1, b_2 \ldots b_m$ ---> $b_m, b_{m-1} \ldots b_1, a_n, a_{n-1} \ldots a_1$
>
> ---> $b_1, b_2 \ldots b_m, a_1, a_2, a_3 \ldots a_n$

```
bool ReverseApply(SqList &L, int left, int right) {
    //转置left->right内元素
    if (left >= right || right >= L.length)
        return false;
    int mid = (left + right) / 2;
    for (int i = 0; i <= mid - left; ++i) {
        ElemType e = L.data[left + i];
        L.data[left + i] = L.data[right - i];
        L.data[right - i] = e;
    }
}

bool Exchange(SqList &L, int m, int n) {
    //L.data[m+n]中存放的元素，前m个与后n个互换，然后m内互换，n内互换
    ReverseApply(L, 0, m - n + 1);
    ReverseApply(L, 0, n - 1);
    ReverseApply(L, n, m + n - 1);
}
```

2.2.2 线性表的链式存储

> 用一组任意的存储单元存储线性表的数据元素

- 结点：包含数据域和指针域
  - 指针域中存储的信息称作指针或链
- 头指针：指向头结点的位置
- 头结点：链表的第一个节点之前附设一个结点，称为头结点。
  - 若线性表为空表，头结点的指针域指为 `NULL`
  - 数据域不设具体数据信息

引入头结点原因：

- 空表和非空表统一
  - 无论是否为空，头指针指向都非空
- 链表的第一个位置上操作和表在其他位置的操作一致，无需特殊处理

1. 特点

- 不要求连续存储空间，逻辑上相邻的元素通过 指针 标识
- 链表同样可反映数据间的逻辑关系
- 不支持随机存取

2. 单链表

H5 1. 结点定义

```
typedef struct {
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
```

H5 2. 基本操作

H6 单链表的创建

1. 带不带头结点在代码实现中，区别在于对第一个结点的特殊处理
   - 若不带头结点， 表名 即指向第一个数据结点
   - 若带头结点， 表名 指向头结点， `表名->next` 指向第一个数据结点；
2. 必须将第一个头结点 `next` 设为NULL，因为一直向后传递，且没有尾指针
3. 每个结点插入时间为 $O(1)$ ，插入n个结点时间为 $O(n)$
4. 最后一个结点的`next`域为NULL

1. 头插法

> 多用于实现原地逆置

带头结点

```
LinkList HeadInsert(LinkList &L,int n) {
    if(n < 0)
        return ERROR;

    LNode *p;//游标
    L = (LinkList)malloc(sizeof(LNode));//创建头结点
    if (!L)
        exit(OVERFLOW);
    L->next = NULL;//初始为空链表
    L.length = 0;

    while(L.length < n){
        p = (LNode *)malloc(sizeof(LNode));//创建新结点
        if (!p)
            exit(OVERFLOW);
        scanf_s("%d", &p->data);//输入结点的值

        p->next = L->next;
        L->next = p;//将结点插入到表头，L为头指针
        L.length++;
    }

    return L;
}
```

不带头结点

```
LinkList HeadInsert2(LinkList &L,int n) {
    if(n < 0)
        return ERROR;
    LNode *p;//游标
    L.length = 0;

    while(L.length < n){
        p = (LNode *)malloc(sizeof(LNode));
        if (!p)
            exit(OVERFLOW);
        scanf("%d", &p->data);
        p->next = i==0 ? NULL : L-next;

        L = p;//将结点插入到表头
        L.length++;
    }

    return L;
}
```

2. 尾插法

- 表尾结点指针域为 `NULL`

带头结点

```
LinkList TailInsert(LinkList &L,int n) {
    L = (LinkList)malloc(sizeof(LNode));//创建头结点
    if (!L)
        exit(OVERFLOW);
    LNode *r = NULL,*p = NULL;
    L.length = 0;

    while (L.length < n) {
        p = (LNode *)malloc(sizeof(LNode));
        if (!p)
            exit(-1);
        scanf("%d",&p->data);
        p->next = NULL;
        r->next = p;
```

```
        r = p;//tail指向新的表尾结点
    }

    return L;
}
```

不带头结点

```
LinkList TailInsert(LinkList &L,int n) {
    LNode *r = NULL,*p = NULL;
    L.length = 0;

    while (L.length < n) {
        p = (LNode *)malloc(sizeof(LNode));
        if (!p)
            exit(OVERFLOW);
        scanf("%d",&p->data);
        p->next = NULL;
        if(!r){//第一个结点
            L = p;
        }else{
            r->next = p;
        }
        r = p;
    }

    return L;
}
```

H6 查找

1. 按序号查找

> 当第 `i` 个元素存在是，将钙元素的值赋给 `e` 并返回 `OK`，否则返回 `ERROR`

```
Status GetElem_L(LinkList &L,int i,ElemType &e){
    //L是带头结点的单链表头的头指针
    LNode *p = L->next;//游标
    int k = 1;//计数器
    while(p && k < i){
        p = p->next;
        k++;
    }//循环退出的条件是游标指向第i个元素或者到达表尾
    if(!p || k > i)//当i为小于0的违法输入是，不会进入while循环。此时k=1>i
        return ERROR;

    e = p->data;
    return OK;
}
```

2. 按值查找

```
//按值查找
LNode *LocateElem(LinkList L, ElemType e) {
    LNode *p = L->next;

    while (p != NULL && p->data != e)//从第一个结点开始查找data域为e的结点
        p = p->next;

    return p;//找到后返回该结点指针
}
```

H6 插入

对第 `i` 个结点前插 <==> 对第 `i-1` 个结点后插

1. 插入到第 `i` 个位置

> 只知道插入位序
>
> - 查找位序时间 $O(n)$
> - 插入时间 $O(1)$

```
//前一个后插
Status ListInsert_L(LinkList &L,int i, ElemType e) {
    LNode *p = L->next;
    k = 0;
    while(p && k < i-1){
        p = p->next;
        k++;
    }//结束：p到表尾或指向第i-1个元素

    if(!p || k > i-1)//违法输入或遍历到表尾
        return ERROR;

    LNode *s = (LNode *)malloc(sizeof(LNode));
    if(!s) return OVERFLOW;
    s->data = e;
    s->next = p->next;
    return OK;
}
```

2. 后插&交换

> 当已知插入结点p时
>
> - 时间复杂度 $O(1)$

```cpp
//后插后交换
bool InsBefore2(LinkList &L, LNode *p, LNode *s) {//在p之前插入s结点
    //将s插入到p后
    s->next = p->next;
    p->next = s;
    //交换数据域。使s换到p之前
    ElemType tmp = s->data;
    s->data = p->data;
    p->data = tmp;

    return true;
}
```

H6 删除

1. 已知索引删除

从链表头开始顺序查找到 `p` 的前驱结点，执行删除操作

> - 时间复杂度为 $O(n)$

```
//已知索引删除
Status LinkListDelete_L(LinkList &L, int i, ElemType &e) {
    if (i < 1 || i > Length(L))
        return ERROR;

    int j = 0;
    LNode *s, *p = L;

    while (p && j < i - 1) {//寻找第 i 个结点的前驱结点
        p = p->next;
        j++;
    }
    if (p && p->next) {
        s = p->next;
        p->next = s->next;
        e = s->data;
        free(s);//释放s结点

        return OK;
    }
}
```

2. 已知结点删除

> 删除结点 `p` 的后继结点实现
>
> - 时间复杂度为 $O(1)$

```
//已知结点删除
bool Del2(LinkList &L, LNode *p) {
    LNode *q = p->next;//指向p后的结点

    p->data = p->next->data;
    p->next = q->next;
    free(q);

    return true;
}
```

H6 求长度

> 对不带头结点的单链表，表为空时，要单独处理

带头结点

```
int Length(LinkList L) {
    int n = 0;
    LNode *p = L->next;
    while (p) {
        n++;
        p = p->next;
    }
    return n;
}
```

不带头结点

```c
int Length2(LinkList L) {//不带头结点
    if (L == NULL) {
        return 0;
    }
    int n = 0;
    LNode *p = L;
    while (p) {
        n++;
        p = p->next;
    }

    return n;
}
```

H5 公共序列

H6 找公共 结点

> 两个链表有公共结点，即从第一个公共结点开始，它们的 `next` 域都指向同一个结点。从第一个公共结点开始，之后它们的所有结点都是重合的，不可能出现分叉。即只能是 Y ，不可能是 X。

1. 暴力

> 空间复杂度：$O(len1 * len2)$

```c
LNode *Search(LinkList L1,LinkList2){
    LNode *pa = L1->next,*pb = L2->next;

    while(pa != NULL){
        pb = L2->next;
        while(pb != NULL){
            if(pa->data == pb->data)
                break;
            pb = pb->next;
        }
        if(pb == NULL)//没有找到公共结点
            pa = pa->next;
    }

    return pa;
```

```
}
```

2. 最优

> 由于从公共结点开始到最后一个结点是相同的，所以从最后一个结点回溯，可以找到第一个公共结点。若截取长链表多出来部分，并不影响公共部分。
>
> 时间复杂度：$O(len1+len2)$

```
LNode *Search(LinkList L1,LinkList L2){
    LinkList longList,shortList;
    int dist;
    if(L1.length > L2.length){
        longList = L1.length;
        shortList = L2.length;
        dist = L1.length-L2.length;
    }else{
        shortList = L1.length;
        longList = L2.length;
        dist = L2.length-L1.length;
    }

    while(dis--)
        longList = longList->next;

    while(!longList){
        if(longList == shortList)
            return longList;
        else{
            longList = longList->next;
            shortList = shortList->next;
        }
    }

    return NULL;
}
```

H6 找公共 序列

> 实质上是模式匹配，A为主串，B为模式串

1. 暴力法

> 时间复杂度：$O(len1 * len2)$

```
int pattern(LinkList A,LinkList B){
    LNode *p = A->next,*q = B->next;
    LNode *pre = p;//记录每轮比较A的起始点

    while(p && q){
        if(p->data==q->data){
            p = p->next;
            q = q->next;
        }else{
            pre = pre->next;
            p = pre;
            q = B->next;
        }
    }

    if(q == NULL)
        return 1;//表示匹配成功
    return 0;
}
```

2. KMP算法

H5 合并

> 实质是链表的遍历

H6 归并

> 两个递增链表，合并为一个递减链表
>
> 使用头插法

```
void Merge(LinkList &La,LinkList &Lb){
    LNode *pa = La->next,*pb = Lb->next;
    LNode *q;
    La->next = NULL;//La为结果表

```

```
    while(pa && pb){
        if(pa->data <= pb->data){//La当前结点元素小于Lb
            q = pa->next;//暂存La的后继链，防止断链
            pa->next = La->next;
            La->next = pa;

            pa = q;
        }else{//Lb当前结点小于Lb
            q = Lb->next;
            pb->next = La->next;
            La->next = pb;

            pb = q;
        }
        free(q);
    }

    //将剩余结点插入结果表
    if(pa)
        pb = pa;
    while(pb){
        q = pb->next;
        pb->next = La->next;
        La->next = pb;
        pb = q;
        free(q);
    }
    free(Lb);
}
```

H6 求两链表交集

> 只有同时出现在两链表中的元素才链接到新表
>
> 将 `L1` 作为新表，`L2` 释放

```
LinkList Union(LinkList L1,LinkList L2){
    LNode *pa = L1->next,*pb = L2->next;
    LNode *r;//指向待释放结点
    LinkList pc = (LinkList)malloc(sizeof(LNode));
    pc = pa;//pc为结果表游标，指向表尾元素

```

```
    while(pa && pb){
        if(pa->data == pb->data){
            pc->next = pa;//pc指向L1中结点
            pc = pa;
            pa = pa->next;
            //释放L2中结点
            r = pb;
            pb = pb->next;
            free(r);
        }else if(pa->data < pb->data){
            r = pa;
            pa = pa->next;
            free(r);
        }else(pa->data > pb->data){
            r = pb;
            pb = pb->next;
            free(r);
        }
    }

    while(pa){
        r = pa;
        pa = pa->next;
        free(r);
    }
    while(pb){
        r = pb;
        pb = pb->next;
        free(r);
    }
    pc->next = NULL;
    free(L2);
    return L1;
}
```

H6 有序表找公共元素

> 要求不破坏原链表
>
> 值不等，则将值小的指针后移；值相等，创建一个新结点，尾插法到新表尾。

```
LinkList getCommon(LinkList L1,LinkList2){
```

```
    LNode *pa = L1->next,*pb = L2->next;
    LinkList L3 = (LinkList)malloc(sizeof(LNode));
    if(!L3)
        exit(OVERFLOW);
    LNode *r = L3;//指向新结点表尾

    while(pa != NULL && pb != NULL){
        if(pa->data < pb->data)
            pa = pa->next;
        else if(pb->data < pa->data)
            pb = pb->next;
        else{
            LNode *s = (LNode *)malloc(sizeof(LNode));
            s->data = p->data;
            r-next = s;
            r = s;
            pa = pa->next;
            pb = pb->next;
        }
    }
    r->next = NULL;//表尾指针置为NULL
}
```

H5 3. 相关思路

H6 头插法

- 用于实现逆置

1. 逆序输出

```
void RPrint(LinkList L){
    LinkList A = (LinkList)malloc(sizeof(LNode));
    A->next = NULL;
    LNode *p = L->next;

    while(!p){//头插法建立带头结点的A
        LNode *q = (LNode *)malloc(sizeof(LNode));
        q->data = p->data;
        q->next = A->next;
        A->next = q;

        p = p->next;
```

```
    }

    p = A->next;
    while(!p)//遍历A
        print(p->data);
}
```

2. 就地逆置

```
void Reverse(LinkList &L){
    LNode *p,*q;
    p = L->next;
    L->next = NULL;

    while(!p){
        q = p->next;//暂存p的后继链
        p->next = L->next;
        L->next = p;
    }
}
```

H6 尾插法

1. 删除指定值

```
void DelX(LinkList &L,ElemType e){
    LNode *p = L->next;//游标
    LNode *r = L;//指向尾结点
    LNode *q; //暂存待删除结点

    while(!p){
        if(p->data!=e){//p结点值不为x时将其链接到L表尾
            r->next = p;
            r = p;
            p = p->next;
        }else{
            q = p;
            p = p->next;
            free(p);
        }
    }
```

```
    r->next = NULL;//表尾结点指针域置NULL
}
```

2. 删除最小值

> 遍历：一趟简单选择排序
>
> 时间复杂度：$O(n)$

```
void DelMin(LinkList &L){
    LNode *pre = L,*p = pre->next;//快慢指针
    LNode *minpre = pre,*minp = p;//minpre指向当前最小值结点的前驱

    while(!p){
        if(p->data < minp->data){//更新
            minp = p;
            minpre = pre;
        }
        pre = p;
        p = p->next;
    }

    minpre->next = minp->next;//删除最小值结点
    free(minp);
}
```

3. 有序表去重

> 有序表，则前后结点数据不相等，则不重复
>
> 若为无序表，则遍历有序表后才可判断是否重复，时间复杂度为$O(n^2)$

```
void DelDuplicate(LinkList &L){
    LNode *p = L->next,*ra = p;
    LNode *r;
    p = p->next;

    while(p != NULL){
        if(p->data != ra->data){
```

```
            ra->next = p;
            ra = p;
            p = p->next;
        }else{//若相等，则释放该结点
            r = p;
            p = p->next;
            free(r);
        }
    }
}
```

H6 快慢指针

> 去重

1. 删除指定值

> `p`：快指针
>
> `pre`：慢指针
>
> 时间复杂度：$O(n)$
>
> 空间复杂度：$O(1)$

```
void DelX(LinkList &L,ElemType e){
    LNode *q;//暂存待删除结点
    LNode *p = L->next, *pre = L;

    while(!p){
        if(p->data == e){
            q = p;
            p = p->next;
            pre->next = p;
            free(q);
        }else{//同时后移
            pre = p;
            p = p->next;
        }
    }
}
```

2. 删除无序链表指定范围值

```
void RangeDel(LinkList &L,ElemType min,ElemType max){
    LNode *pre = L,*p = pre->next;//快慢指针
    while(!p){
        if(p->data > min && p->data < max){
            pre->next = p->next;
            p = p->next;
            free(p);
        }else{
            pre = p;
            p = p->next;
        }
    }
}
```

3. 有序表去重

> 时间复杂度：$O(n)$
>
> 空间复杂度：$O(1)$

```
void DelDulplicate(LinkList &L){
    LNode *p = L->next,LNode *q;
    if(p == NULL)
        return ;
    while(p->next != NULL){
        q = p->next;
        if(p->data == q->data){
            p->next = q->next;
            free(q);
        }else
            p = p->next;
    }
}
```

H6 快慢指针实现指针反转

1. 就地逆置

> `pre`：指向慢指针指向的前一个结点

`q`：慢指针，初始指向头结点

`p`：快指针，初始指向头结点的下一个结点

```
void Reverse(LinkList &L){
    LNode *pre, *q = L->next,*p = q->next;
    q->next = NULL;
    while(!p){
        //指针后移
        pre = q;
        q = p;
        p = p->next;
        //修改指针指向
        q->next = pre;
    }
}
```

1. 排序

思路：

基于 直接插入排序 思想，此时前后指针是为了不断链

时间复杂度：$O(n^2)$

```
void sort(LinkList &L){
    LNode *p = L->next,*pre;//pre为有序表游标
    LNode *r = p->next;//r指向p的下一个结点
    p->next = NULL;//构建一个只有一个结点的链表

    p = r;
    while(!p){
        r = p->next;

        pre = L;
        while(pre->next != NULL && pre->data < p->data)
            pre = pre->next;
        p->next = pre->next;
        pre->next = p;

        p = r;
```

```
    }
}
```

2. 递增输出

> 简单选择排序思想，若不影响原链表，则需要进行复制
>
> 时间复杂度：$O(n^2)$

```
void AscPrint(LinkList L){
    LNode *r;//指向被每轮被释放结点
    while(L->next){
        LNode *pre = L;//最小值结点的前驱结点，确保不断链
        LNode *p = pre->next;

        while(p){
            if(p->next->data < pre->next-data)
                pre = p;
            p = p->next;
        }
        print(p->next-data);
        r = pre->next;
        pre->next = r->next;
        free(r);
    }

    free(L);
}
```

3. 拆分

> 奇数位结点元素入 `A` 尾，偶数位结点元素入 `B` 尾，结果表中元素相对位置不变
>
> 指针赋值，会使他们指向同一个存储单元。对两个指针的操作会修改同一个结点的属性
>
> 时间复杂度：$O(n)$
>
> 空间复杂度：$O(1)$

```
LinkList DisCreate(LinkList &A){
    int cnt = 0;//计数
    B = (LinkList)malloc(sizeof(LNode));
    if(!B)
        exit(OVERFLOW);
    LNode *ra = A,*rb = B;//结果表表尾指针
    LNode *p = ra;//游标
    A->next = NULL;

    while(p){
        cnt++;
        if(cnt%2==0){
            rb->next = p;
            rb = p;
        }else{
            ra->next = p;
            ra = p;
        }
        p = p->next;
    }
    ra->next = NULL;
    rb->next = NULL;

    return B;
}
```

```
LinkList DisCreate(LinkList &A){
    B = (LinkList)malloc(sizeof(LNode));
    if(!B)
        exit(OVERFLOW);
    LNode *p = A;//游标
    A->next = NULL;
    B->next = NULL;
    LNode *ra = A,*rb = B;//结果表表尾指针

    while(p){
        ra->next = p;
        ra = p;
        p = p->next;

        if(p != NULL){
            rb->next = p;
```

```
            rb = p;
            p = p->next;
        }
    }
    ra->next = NULL;
    rb->next = NULL;

    return B;
}
```

> 奇数位结点元素入 `A` 尾，偶数位结点元素入 `B` 头

```
LinkList DisCreate(LinkList &A){
    LinkList B = (LinkList)malloc(sizeof(LNode));
    B->next = NULL;
    LNode *p = A->next,q;
    LNode *ra = A;//指向A尾结点

    while(p != NULL){
        ra->next = p;
        ra = p;//将p链接到A表尾
        p = p->next;
        if(p != NULL)
            q = p->next;//暂存p的后续链，防止断链
        p->next = B->next;
        B->next = p;
        p = q;
    }

    ra->next = NULL;
    return B;
}
```

H6 空间换时间

1. 逆序输出

> 时间复杂度：$O(n)$
>
> 空间复杂度：$O(n)$

```
void RPrint(LinkList L){
    Stack s;
    LNode * p = L->next;

    while(!p)
        Push(s,p->data);
    while(!EmptyStack(s)){
        ElemType e;
        Pop(s,e);
        print(e);
    }
}
```

2. 链表排序

> 将链表数据复制到数组中，采用 $O(nlonn)$ 的排序算法排序，然后将数组元素插入到链表中
>
> 时间复杂度：$O(n)$

```
void sort(LinkList &L){
    LNode *p = L->next;
    ElemType a[MaxSize];
    int i = 0;

    while(!p){
        a[i++] = p->data;
        p = p->next;
    }

    QuickSort(a,0,L.length-1);
    //修改指针域
    p = L->next;
    for(i = 0;i < L.length;++i){
        p->data = a[i];
        p = p->next;
```

```
    }
}

int Partition(ElemType a[],int low,int high){
    ElemType pivot = a[low];
    while(low < high){
        while(low < high && a[high] > pivot)
            --high;//找到第一个比枢轴小的位置
        a[low] = a[high];
        while(low < high && a[low] < pivot)
            ++low;//找到第一个比枢轴大的位置
    }
    a[low] = pivot;
    return low;
}

void QuickSort(ElemType a[],int low,int high){
    if(low < high){
        int pivotpos = Partition(a,low,high);
        QucikSort(a,low,pivotpos-1);
        QucikSort(a,pivotpos+1,high);
    }
}
```

3. 递增输出

> 将链表中数据复制到数组中，排序后输出

H6 递归

- 递归 = 出口 + 调用
- 为保证一致性：递归从第一个有数据的结点开始

1. 删除指定值

> 时间复杂度：$O(n)$
>
> 递归栈深度：$O(n)$

```
void DelX(LinkList &L,ElemType e){
    //L不带头结点
    LNode *p = NULL;//暂存待删除结点
    if(L==NULL)// 递归出口
        return ;
    if(L->data == e){
        p = L;//p指向待删除结点
        L = L->next;
        free(p);//释放空间
        DelX(L,x);
    }else
        DelX(L->next,x);
}
```

2. 逆序输出

```
void RPrint(LinkList L){
    if(!L->next)
        RPrint(L->next);

    if(!L)//递归出口
        print(L-data);
}

void IgnoreHead(LinkList){
    if(!L)
        RPrint(L->next);
}
```

3. 双链表

- 优化：访问前驱结点 —— $O(1)$
  - $O(n)$ <-- 单链表

H5 1. 结点定义

```
typedef struct DulNode{//定义双链表结点类型
    ElemType data;//数据域
    struct DulNode *prior,*next;//前驱和后继指针
}DulNode,*DulLinkList;
```

H5 2. 基本操作

H6 插入

```
//s为待插入结点，p为其前驱结点
s->next = p->next;
p->next->prior = s;
s->prior = p;
p->next = s;
```

H6 删除

```
//p为前驱，q为待删除结点
p->next = q->next;
q->next->prior = p;
free(q);
```

H5 最近最高频访问

> 双向链表中查找到值为 `x` 的结点，查找到后，将结点从链表摘下，然后顺着结点的前驱找到该结点的插入位置（频度递减，且排在同频度的第一个。即向前找到第一个比他大的结点，插在该结点位置之后）

```
DLinkList Locate(DLinkList &L,ElemType e){
    DNode *p = L->next,*pre;
    while(p && p->data != x)
```

```
        p = p->next;
    if(!p){
        //结点不存在
        return NULL;
    }else{
        p->freq++;
        //修改结点的前驱后继
        if(p->next != NULL)
            p->next->pred = p->pred;
        p->pred->next = p->next;
        //寻找插入位置
        pre = p->pred;
        while(pre != L && pre->freq <= p->freq)
            pre = pre->pred;//最后一轮，pre指向插入位置的前驱
        //插入
        p->next = pre->next;
        pre->next->pred = p;
        p->pred = pre;
        pre->next = p;
    }

    return p;
}
```

4. 循环链表

- 优化：对表尾操作—— $O(1)$
  - 单链表为 $O(n)$
  - 单链表删除最后一个元素 需要将最后一个元素空指 —— $O(n)$

H5 1. 循环单链表

- 表尾 `r->next` 指向头指针（判空条件）
- 插入删除操作
  - 表尾 `next->L`

应用

> 若操作多为在表头和表尾 插入 时， 设尾指针
>
> - 头指针对表尾操作为 $O(n)$

Note

- 若对 表尾删除 操作，单链表寻找其前驱结点为 $O(n)$
  - 需要采用 循环双链表

H5 约瑟夫问题

```c
# include<stdio.h>
# include<stdlib.h>

typedef struct LNode{
    int no;
    unsigned int pwd;
    struct LNode *next;
}LNode,*LinkList;

LinkList CreateLinkList(int n);
void playing(LinkList tail,int n,int m);

//7 5
//3 8 1 22 4 9 15
//===5 2 6 7 4 3 1
int main(){
    LinkList tail;
    int n,it;

    scanf("%d%d",&n,&it);//输入初始数量与初始密码
    tail = CreateLinkList(n);//创建不带头结点的单循环链表
    if(tail)
        playing(tail,n,it);

    return 0;
}

LinkList CreateLinkList(int n){
    LNode *p,*r;
    p = (LNode *)malloc(sizeof(LNode));
    if(!p)
```

```
        exit(-1);
    scanf("%d",&p->pwd);
    p->no = 1;
    p->next = p;
    r = p;
    for(int i = 2;i <= n;++i){
        p = (LNode *)malloc(sizeof(LNode));
        if(!p)
            exit(-1);
        scanf("%d",&p->pwd);
        p->no = i;
        p->next = r->next;
        r->next = p;
        r = p;
    }

    return r;
}

void playing(LinkList tail,int n,int m){
    //L为环， n为环中结点数量，m为初始密码
    LNode *pre,*p;
    m = m%n ? m%n : n;//检验m为合法输入
    pre  = tail;
    p = pre->next;
    int k = 1;//计数器

    while(n > 1){//环中人数多余1时
        if(k==m){//数到需要出圈的人
            printf("%4d",p->no);
            pre->next = p->next;
            n--;
            m = p->pwd%n ? p->pwd%n : n;
            free(p);
            p = pre->next;
            k = 1;
        }else{
            k++;
            pre = p;
            p = p->next;
        }
    }
    printf("%4d",p->no);
```

```
}
```

将两个循环单链表连接

```
LinkList link(LinkList &L1,LinkList &L2){
    LNode *p = L1,*q = L2;
    while(p->next != L1)
        p = p->next;
    while(q->next != L2)
        q = q->next;

    p->next = L2;
    q->next = L1;
    return L1;
}
```

循环单链表选择排序

```
void Del(LinkList &L){
    LNode *p,*pre,*minp,*minpre;
    while(L->next != L){
        p = L->next;
        pre = L;
        minp = p;
        minpre = pre;
        while(p!=L){//寻找最小元素
            if(p->data < minp->data){
                minp = p;
                minpre = pre;
            }
            pre = p;
            p = p->next;
        }
        print(minp->data);
        minpre-next = minp->next;
        free(minp);
    }
    free(L);
```

```
21  }
```

H5 **2. 循环双链表**

1. 空表条件
   - 头结点 `p->pre == p->next = L`
2. 便于进行各种修改操作，但占有较大指针域，存储密度不高

H6 **判断循环双链表是否对称**

> `p` 从左向右扫描，`q` 从右向左扫描，若相等，则一直比较下去，直到指向同一结点(`p == q`)或者相邻(`p->next==q` 或 `q->prior ==p`)；否则，返回0。

```
1  int Symmetry(DLinkList L){
2      DNode *p = L->next,*q = L->prior;//两头工作指针
3      while(p != q && q->next != p){
4          if(p->data == q->data){
5              p = p->next;
6              q = q->prior;
7              //当数据结点为偶数时，最后一轮遍历完q在p指针前，所以判断退
   出条件是q->next != p
8          }else
9              return 0;
10     }
11
12     return 1;
13 }
```

5. 静态链表

> 预先分配连续的内存空间

- 指针 <==> 游标
- `next == -1` 为表尾

结点定义

```
# define MaxSize 50 //静态链表的最大长度
typedef struct{//静态链表结构类型定义
    ElemType data;//存储数据元素
    int next;//下一个元素的数组小标
}SLinkList[MaxSize];
```

2.3 顺序表与链表比较

	存取方式	逻辑&物理结构	查找	插&删	空间分配
顺序表	顺序存取	逻辑相邻 存储相邻	无序： O(n)； 有序： O(logn)	O(n)	静态分配： 过大：浪费； 过小：内存溢出
	随机存取		按序号： O(1)		动态分配： 效率低，需要移动大量元素
链表	顺序存取	逻辑关系通过 指针表示 存储密度低	O(n)	O(1)	按需分配，灵活高效

2.4 存储结构的选择

- 较稳定——顺序存储
- 频繁修改——链式存储

	基于存储	基于运算	基于运算
顺序表	适用于有存储密度的要求	常用操作为按序号访问	不支持指针的语言；易于实现
链表	适用于难以估计存储规模	常用操作为插入删除	基于指针

Note：插入删除

> 链表按位序查找为 $O(n)$，但主要进行比较操作；
>
> 顺序表主要操作是移动数据元素；

> 虽然时间复杂度同样为 $O(n)$，但显然比较操作相对优与移动操作