第二章、线性表

2.1、线性表的定义和基本操作

知识总览

线性表的定义：具有 *相同数据类型* 的n（n≥0）个数据元素的 *有限* 序列，其中n为表长，当n = 0时线 性表是一个空表。

注意：

a_i是线性表中的“第i个”元素线性表中的 *位序* ，位序从1开始 数组下标从0开始

a_1是表头元素；a_n是表尾元素。

除第一个元素外，每个元素有且仅有一个直接前驱；除最后一个元素外，每个元素有且仅有一个直接后继

线性表的基本操作：

```
InitList(&L)：初始化表。构造一个空的线性表L，分配内存空间。
DestroyList(&L)：销毁操作。销毁线性表，并释放线性表L所占用的内存空间。
ListInsert(&L,i,e)：插入操作。在表L中的第i个位置上插入指定元素e。
ListDelete(&L,i,&e)：删除操作。删除表L中第i个位置的元素，并用e返回删除元素的值。
LocateElem(L,e)：按值查找操作。在表L中查找具有给定关键字值的元素。
GetElem(L,i)：按位查找操作。获取表L中第i个位置的元素的值。
其他常用操作：
Length(L)：求表长。返回线性表L的长度，即L中数据元素的个数。
PrintList(L)：输出操作。按前后顺序输出线性表L的所有元素值。 Empty(L)：判空操作。若L为空表，则返回true，否则返回false。
```

Tips:

①对数据的操作（记忆思路） —— 创销、增删改查

②C语言函数的定义 —— <返回值类型> 函数名 (<参数1类型> 参数1，<参数2类型> 参数2，……)

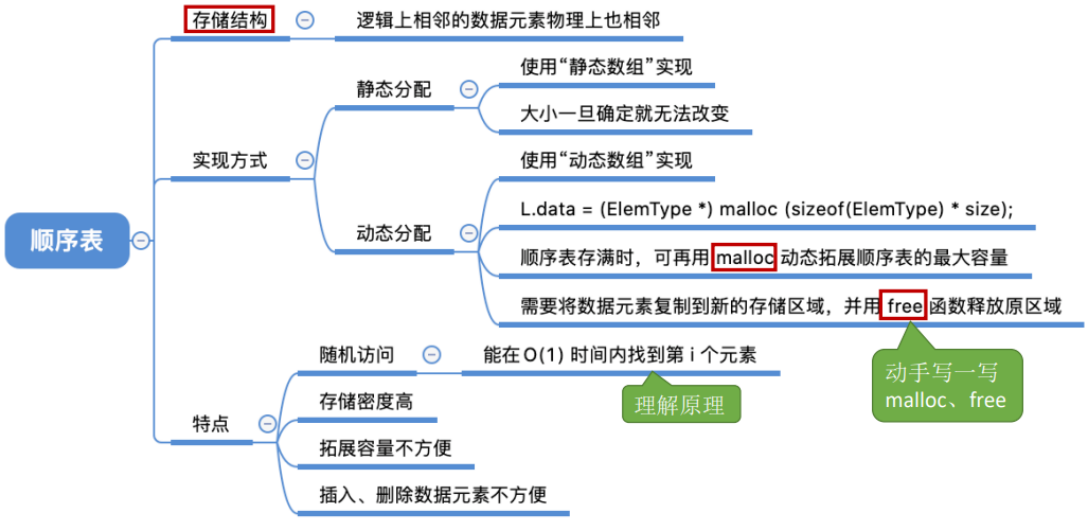
③实际开发中，可根据实际需求定义其他的基本操作

④函数名和参数的形式、命名都可改变（Reference：严蔚敏版《数据结构》）

⑤什么时候要传入引用“&” —— 对参数的修改结果需要“带回来”

2.2.1、顺序表的定义

知识总览：



> **顺序表**：用*顺序存储*的方式实现线性表顺序存储。把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来体现。

- 顺序表—静态分配

```
#define MaxSize 10 //定义最大长度
typedef struct{
    ElemType data[MaxSize]; //用静态的“数组”存放数据元素
    int length; //顺序表的当前长度
}SqList; //顺序表的类型定义（静态分配方式）
// 初始化
void InitList(SqList &L){
    L.length = 0;
}
```

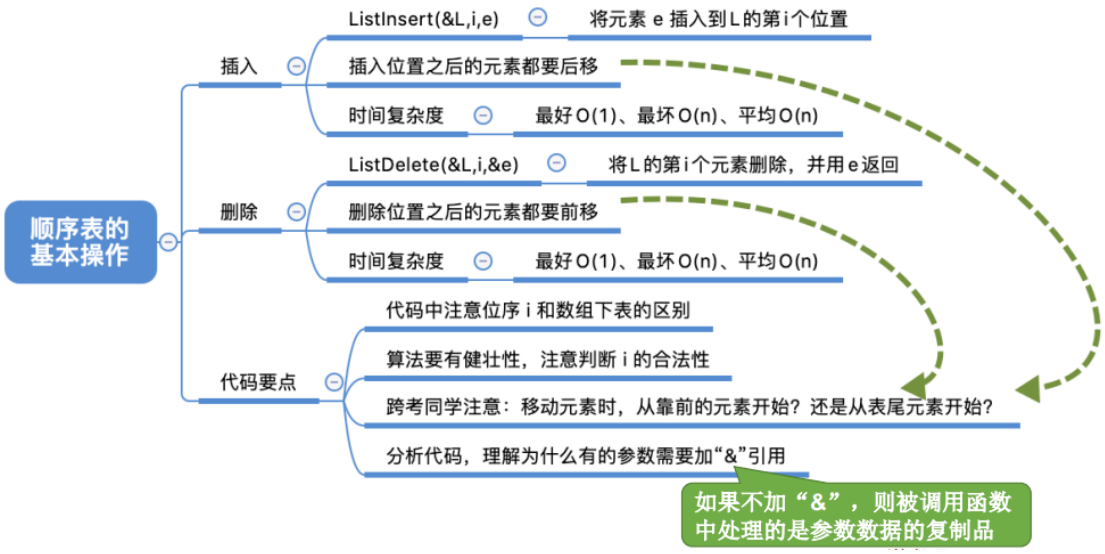
- 顺序表—动态分配

```
#define InitSize 10 //顺序表的初始长度
typedef struct{
    ElemType *data; //指示动态分配数组的指针
    int MaxSize; //顺序表的最大容量
    int length; //顺序表的当前长度
} SeqList; //顺序表的类型定义（动态分配方式）
// 初始化
```

```
void InitList(SqList &L){
    L.data=(ElemType *)malloc(InitSize*sizeof(ElemType));
    L.length = 0;
    L.MaxSize = InitSize;
}
// 动态增加数组长度
void IncreaseSize(SqList &L,int len){
    ElemType *p = L.data;
    L.data = (ElemType *)malloc((InitSize+len)*sizeof(ElemType));
    for(int i = 0;i<L.length;i++){
        L.data[i] = p[i];
    }
    L.MaxSize = L.MaxSize+len;
    free(p);
    p=NULL;
}
```

2.2.2.1、顺序表的插入删除

知识总览：



数据：

```
#define MaxSize 10 //定义最大长度
typedef struct{
    ElemType data[MaxSize]; //用静态的“数组”存放数据元素
    int length; //顺序表的当前长度
}SqList; //顺序表的类型定义（静态分配方式）
```

插入：

```
bool ListInsert(SqList &L,int i,ElemType e){
    if(i<1||i>(L.length+1))
        return false;
    if(L.length >= MaSize)
        return false;
    for(int j=L.length;j>=i;j--)
        L.data[j] = L.data[j-1];
    L.data[i-1] = e;
    L.length++;
}
```

时间复杂度：

最好情况：新元素插入到表尾，不需要移动元素
i = n+1，循环0次；最好时间复杂度 = O(1)
最坏情况：新元素插入到表头，需要将原有的 n 个元素全都向后移动
i = 1，循环 n 次；最坏时间复杂度 = O(n);
平均情况：假设新元素插入到任何一个位置的概率相同，即 i = 1,2,3, ... , length+1 的概率都是 $p = \frac{1}{n+1}$
i = 1，循环 n 次；i=2 时，循环 n-1 次；i=3，循环 n-2 次 ..... i =n+1时，循环0次

平均循环次数 = $np + (n-1)p + (n-2)p + ... + 1\cdot p = \frac{n(n+1)}{2}\frac{1}{n+1} = \frac{n}{2}$ ➡ 平均时间复杂度 = O(n)

删除：

```
bool ListDelete(SqList &L,int i,ElemType e){
    if(i<1||i>L.length)
        return false;
    e = L.data[i-1];
    for(int j=i;j<L.length;j++)
        L.data[j-1]=L.data[j];
    L.length--;
    return true;
}
```

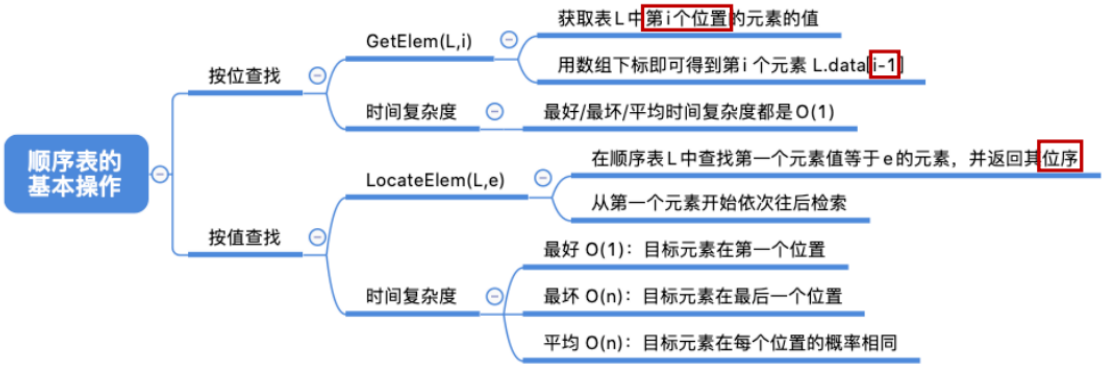
时间复杂度：

最好情况：删除表尾元素，不需要移动其他元素
i = n，循环 0 次；最好时间复杂度 = O(1)
最坏情况：删除表头元素，需要将后续的 n-1 个元素全都向前移动
i = 1，循环 n-1 次；最坏时间复杂度 = O(n);
平均情况：假设删除任何一个元素的概率相同，即 i = 1,2,3, ... , length 的概率都是 $p = \frac{1}{n}$
i = 1，循环 n-1 次；i=2 时，循环 n-2 次；i=3，循环 n-3 次 ..... i =n 时，循环0次

平均循环次数 = $(n-1)p + (n-2)p + ... + 1\cdot p = \frac{n(n-1)}{2}\frac{1}{n} = \frac{n-1}{2}$ ➡ 平均时间复杂度 = O(n)

2.2.2.2、顺序表的查找

知识总览：



数据：

```c
#define InitSize 10 //顺序表的初始长度
typedef struct{
    ElemType *data; //指示动态分配数组的指针
    int MaxSize; //顺序表的最大容量
    int length; //顺序表的当前长度
} SeqList; //顺序表的类型定义（动态分配方式）
```

按位查找：

```c
ElemType GetElem(SeqList L, int i){
    return L.data[i-1];
}
```

时间复杂度：

$O(1)$

tip:由于顺序表的各个数据元素在内存中连续存放， 因此可以根据起始地址和数据元素大小立即找到 第 i 个元素——“随机存取”特性

按值查找：

```c
//在顺序表L中查找第一个元素值等于e的元素，并返回其位序
int LocateElem(SeqList L,ElemType e){
    for(int i=0;i<L.length;i++)
        if(L.data[i]==e)
            return i+1; //数组下标为i的元素值等于e，返回其位序i+1
    return 0; //退出循环，说明查找失败
}
```

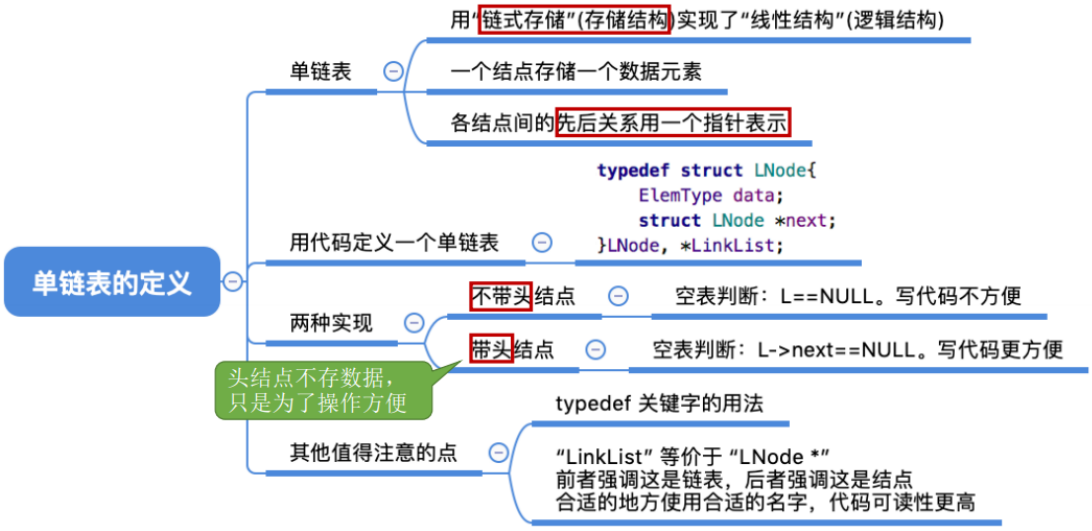
时间复杂度：

最好情况：目标元素在表头
循环1次；最好时间复杂度 = O(1)
最坏情况：目标元素在表尾
循环 n 次；最坏时间复杂度 = O(n);
平均情况：假设目标元素出现在任何一个位置的概率相同，都是 $\frac{1}{n}$
目标元素在第1位，循环1次；在第2位，循环2次；……；在第 n 位，循环 n 次

平均循环次数 = $1\cdot\frac{1}{n}+2\cdot\frac{1}{n}+3\cdot\frac{1}{n}+\cdots\cdots+n\cdot\frac{1}{n}=\frac{n(n+1)}{2}\frac{1}{n}=\frac{n+1}{2}$ ⇒ 平均时间复杂度 = O(n)

2.3.1、单链表

知识总览：



什么是单链表： 每个结点除了存放数据 元素外，还要存储指向 下一个节点的指针

定义单链表：

```
struct LNode{       // 定义单链表结点类型
    ElemType data;  // 每一个结点存放一个数据
    struct LNode *next; // 指针指向下一个结点
}LNode,*LinkList;
// 增加一个新的结点：在内存中申请一个结点所需空间，并用指针 p 指向这个结点
struct LNode * p = (struct LNode *) malloc(sizeof(struct LNode));
```

强调这是一个单链表——使用$LinkList$

强调这是一个结点——使用$LNode*$

初始化单链表：

```
// 带头结点
bool InitList(LinkList &L){
    L = (LNode *)malloc(sizeof(LNode));// 分配头结点
    if(L==NULL)
        return false;
    L->next = NULL;
    return true;
}
```

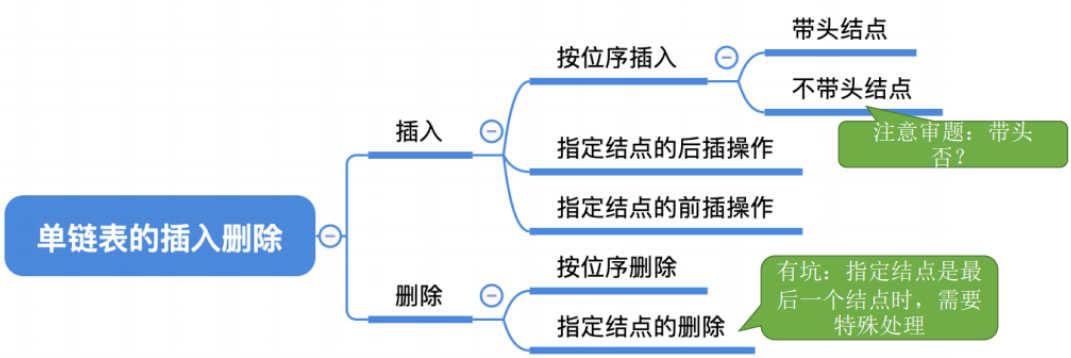
判空：

```
bool Empty(LinkList L){
    if(L->next == NULL)
        return true;
    else
        return false;
}
```

2.3.2.1、单链表的插入、删除、查找、创建

> 插入和删除

知识总览：



按位序插入：

```
// 在第i个位置插入元素e(带头结点)
bool ListInsert(LinkList &L,int i,ElemType e){
    if(i<1)
        return false;
    LNode *p;
    p=L;
    int j = 0;      // 标记当前指向第几个结点
    while(p!=NULL&&j<i-1){
        p=p->next;
        j++;
    }
    if(p==NULL)     // 当i值不合法的时候
        return false;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return true;
}
```

时间复杂度：

最好：$O(1)$

最坏：$O(n)$

平均：$O(n)$

指定节点的后插操作：

```
bool InsertNextNode(LNode *p,ElemType e){
    if(p==NULL)
        return false;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    if(s==NULL)
        return false;
    s->data = e;
    s->next = p->next;
    p->next = s;
    return true;
}
```

指定节点的前插操作：

```
// 按后插操作的方式插入结点，并交换两个数据域的数据
bool InsertPriorNode(LNode *p,ElemType e){
    if(p == NULL)
        return false;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    if(s==NULL)
        return false;
    s->next = p->next;
    p->next = s;
    ElemType temp = p->data;
    p->data = s->data;
    s->data = temp;
    return true;
}
```

按位序删除：

```
bool ListDelete(LinkList &L,int i,ElemType &e){
    if(i<1)
        return false;
    LNode *p;
    p = L;
    int j = 0;
    while(p!=NULL&&j<i-1){
        p = p->next;
        j++;
    }
    if(p==NULL)
        return false;
    if(p->next == NULL)
        return false;
    LNode *q = p->next;
    e = q->data;
    p->next = q->next;
    free(q);
    return true;
```

```
20  }
```

时间复杂度：

最坏、平均时间复杂度：O(n)

最好时间复杂度：O(1)

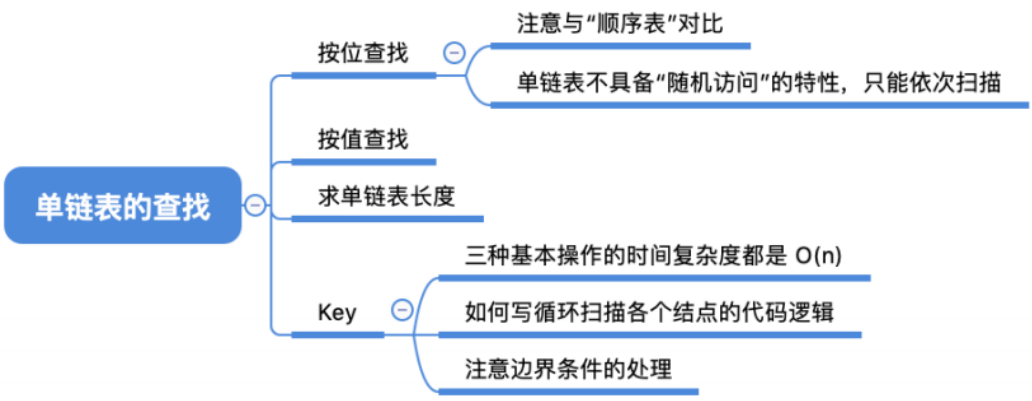
指定节点的删除：

```
1   bool DeleteNode(LNode *p){
2       if(p == NULL)
3           return false;
4       LNode *q;
5       q = p->next;
6       p->data = q->data;
7       p->next = q->next;
8       free(q);
9       return true;
10  }
```

注意：如果p是最后 一个结点...、只能从表头开始依 次寻找p的前驱， 时间复杂度 O(n)

单链表的查找

知识总览：



按位查找：

```
1   LNode * GetElem(LinkList L,int i) {
2       if(i<0)
3           return NULL;
4       LNode *p;
5       p = L;
6       int j = 0;
7       while(p!=NULL&&j<i){
8           p = p->next;
9           j++;
10      }
11      if(p == NULL)
12          return NULL;
13      return p;
14  }
```

平均时间复杂度： $O(n)$

按值查找：

```
LNode *LocateElem(LinkList L,ElemType e){
    LNode *p = L->next;
    while(p!=NULL&&p->data != e)
        p=p->next;
    return p;
}
```

平均时间复杂度： $O(n)$

求长度：

```
int length(LinkList L){
    int len = 0;
    LNode *p = L;
    while(p->next != NULL){
        p=p->next;
        len++;
    }
    return len;
}
```

平均时间复杂度： $O(n)$

单链表的创建：

尾插法：

```
LinkList List_TailInsert(LinkList &L){ //正向建立单链表
    int x; //设ElemType为整型
    L=(LinkList)malloc(sizeof(LNode)); //建立头结点
    LNode *s,*r=L; //r为表尾指针
    scanf("%d",&x); //输入结点的值
    while(x!=9999){ //输入9999表示结束
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s; //r指向新的表尾结点
        scanf("%d",&x);
    }
    r->next=NULL; //尾结点指针置空
```

```
    return L;
}
```

头插法：

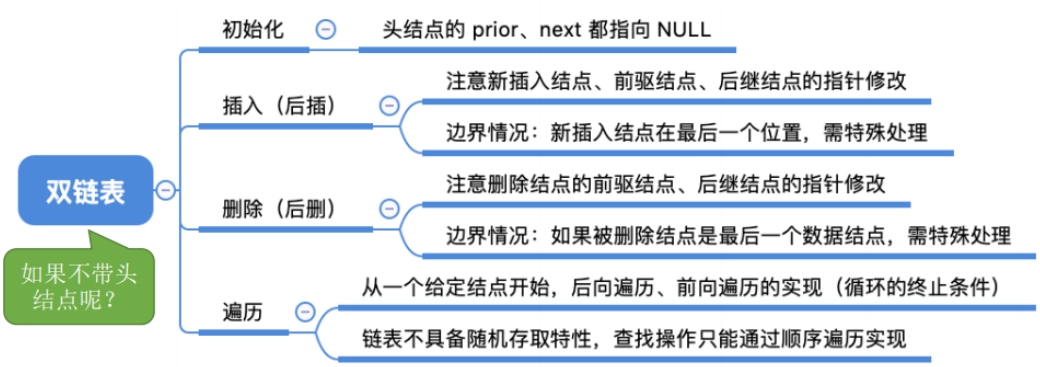
```
LinkList List_HeadInsert(LinkList &L){ //逆向建立单链表
    LNode *s;
    int flag;
    L=(LinkList)malloc(sizeof(LNode)); //创建头结点
    L->next=NULL; //初始为空链表
    scanf("%d",&flag); //输入结点的值
    while(flag!=9999){ //输入9999表示结束
        s=(LNode *)malloc(sizeof(LNode)); //创建新结点
        s->data=flag;
        s->next=L->next;
        L->next=s; //将新结点插入表中，L为头指针
        scanf("%d",&flag);
    }
    return L;
}
```

头插法的重要应用：链表的逆置

2.3.2、双链表、循环链表、静态链表

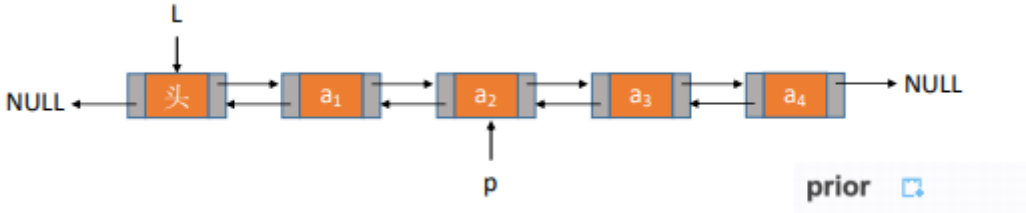
1.双链表

知识总览：



定义：

```
typedef int ElemType;
typedef struct DNode {
    ElemType data;
    struct DNode *next,*prior;
}DNode,*DLinkList;
```



基本操作：

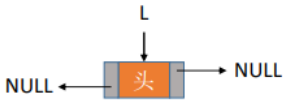
双链表的初始化（带头结点）

```
//初始化双链表
bool InitDLinkList(DLinklist &L){
    L = (DNode *) malloc(sizeof(DNode));    //分配一个头结点
    if (L==NULL)                            //内存不足，分配失败
        return false;
    L->prior = NULL;                        //头结点的 prior 永远指向 NULL
    L->next = NULL;                         //头结点之后暂时还没有节点
    return true;
}
```

```
typedef struct DNode{
    ElemType data;
    struct DNode *prior,*next;
}DNode, *DLinklist;
```

DLinklist 等价 DNode *

```
//判断双链表是否为空（带头结点）
bool Empty(DLinklist L) {
    if (L->next == NULL)
        return true;
    else
        return false;
}
```

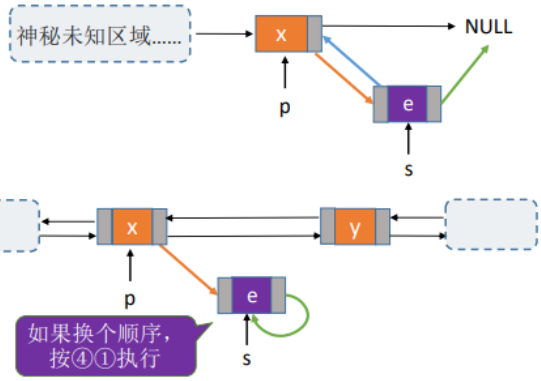


插入：

双链表的插入

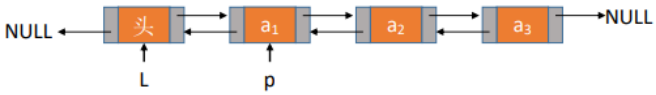
```
//在p结点之后插入s结点
bool InsertNextDNode(DNode *p, DNode *s){
    if (p==NULL || s==NULL)     //非法参数
        return false;
①   s->next=p->next;
②   if (p->next != NULL)       //如果p结点有后继结点
        p->next->prior=s;
③   s->prior=p;
④   p->next=s;
    return true;
}
```

修改指针时要注意顺序



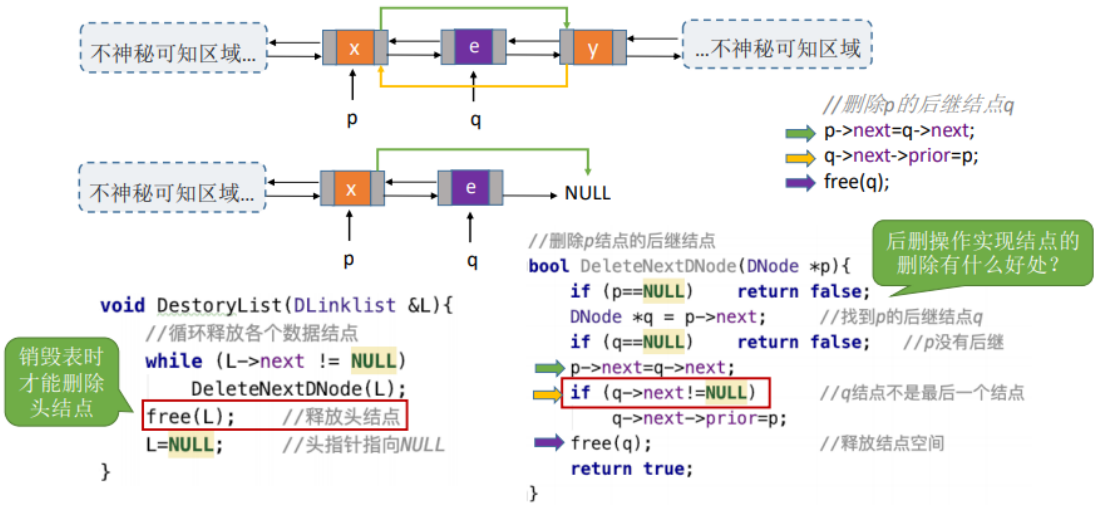
用后插操作实现结点的插入有什么好处？

按位序插入前插操作



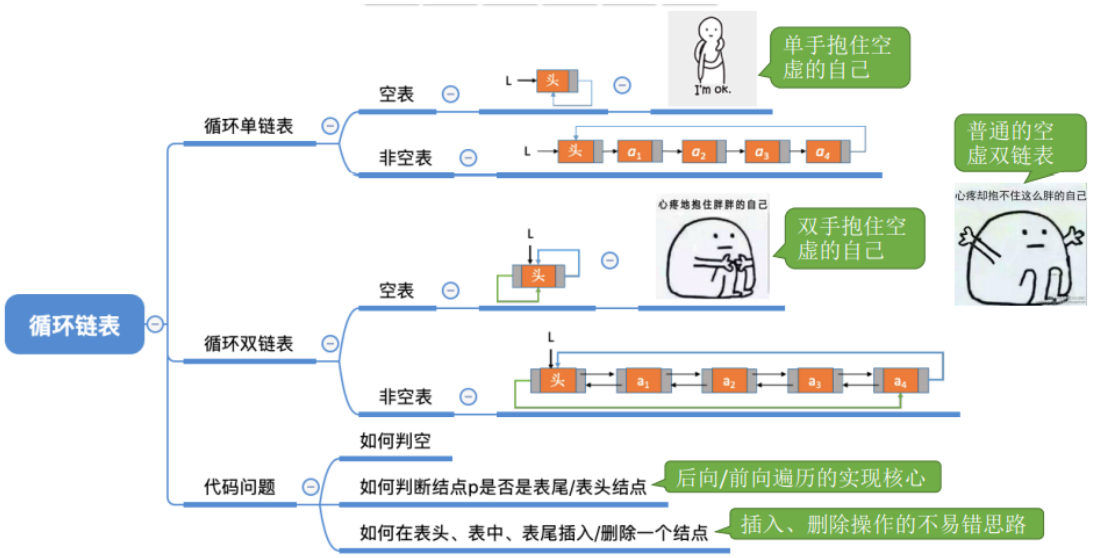
删除：

双链表的删除



2.循环链表

知识总览:

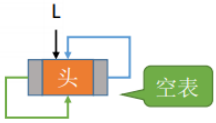


基本操作:

循环双链表的初始化

```
//初始化空的循环双链表
bool InitDLinkList(DLinklist &L){
    L = (DNode *) malloc(sizeof(DNode));    //分配一个头结点
    if (L==NULL)                            //内存不足，分配失败
        return false;
    L->prior = L;          //头结点的 prior 指向头结点
    L->next = L;       //头结点的 next 指向头结点
    return true;
}
```

```
typedef struct DNode{
    ElemType data;
    struct DNode *prior,*next;
}DNode, *DLinklist;
```



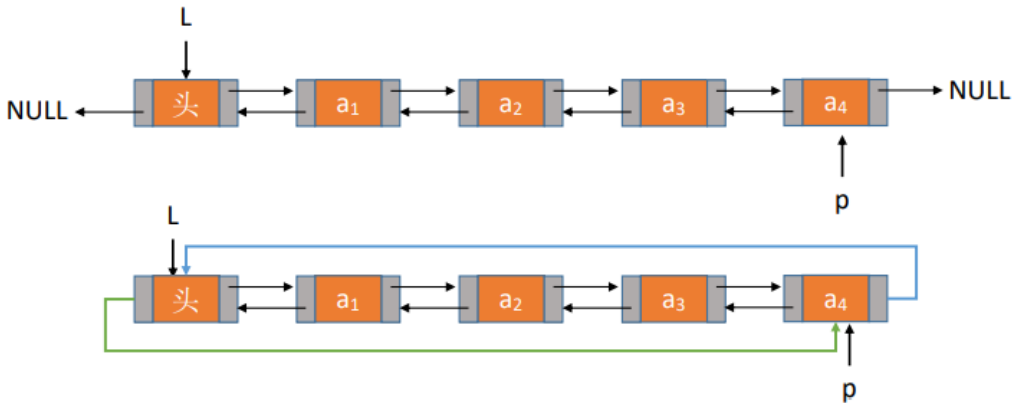

```
//判断循环双链表是否为空
bool Empty(DLinklist L) {
    if (L->next == L)
        return true;
    else
        return false;
}
```

```
//判断结点p是否为循环单链表的表尾结点
bool isTail(DLinklist L, DNode *p){
    if (p->next==L)
        return true;
    else
        return false;
}
```

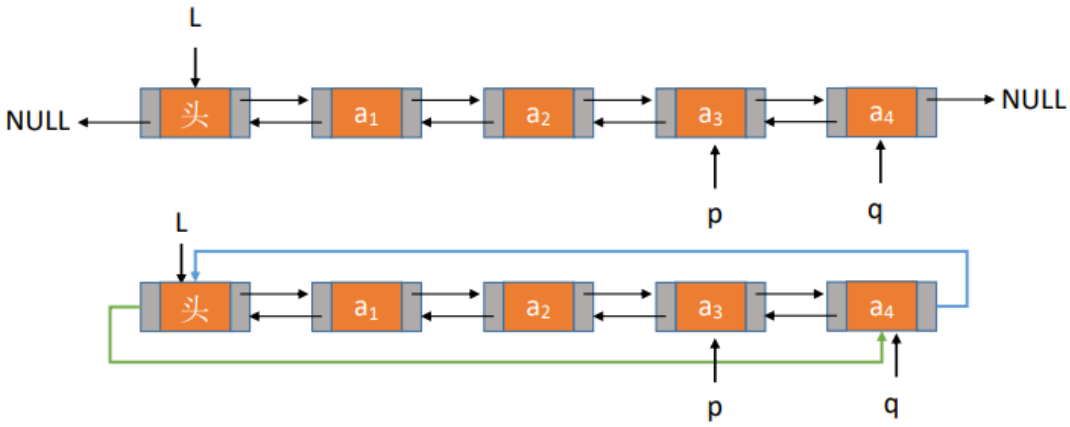
插入:

双链表的插入

```
//在p结点之后插入s结点
bool InsertNextDNode(DNode *p, DNode *s){
    s->next=p->next;     //将结点*s插入到结点*p之后
    p->next->prior=s;
    s->prior=p;
    p->next=s;
}
```



删除：



3.静态链表

- 分配一整片连续的内存空间，各个结 点集中安置。

定义：

```
#define MaxSize 10          //静态链表的最大长度
typedef struct {            //静态链表结构类型的定义
    ElemType data;          //存储数据元素
    int next;               //下一个元素的数组下标
} SLinkList[MaxSize];
```


```
#define MaxSize 10          //静态链表的最大长度
struct Node{                //静态链表结构类型的定义
    ElemType data;          //存储数据元素
    int next;               //下一个元素的数组下标
};
typedef struct Node SLinkList[MaxSize];
```