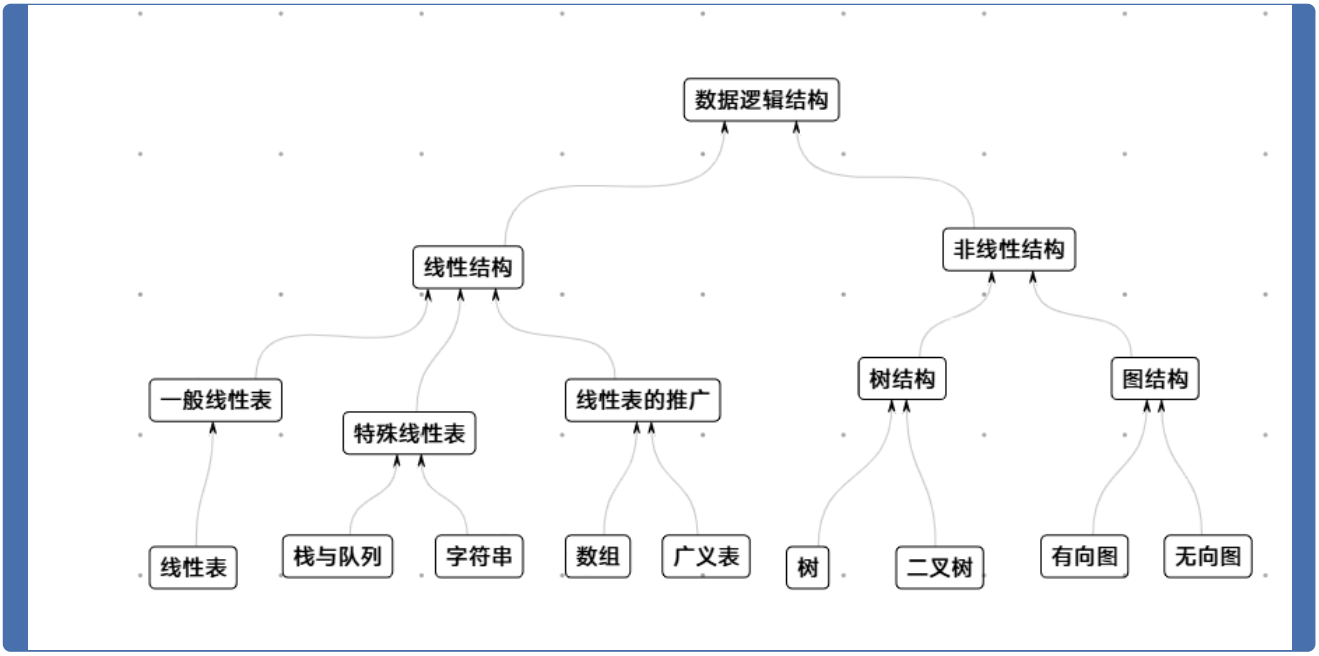
线性表



1.线性表

简称表，时零个或多个元素的有穷序列，通常可以表示为K0，K1...

- 表目：线性表中的元素（可包含多个数据项，记录）
- 索引（下表）：i称为表目Ki的索引或下标
- 表的长度：线性表中所含元素个数n
- 空表：长度为0的线性表

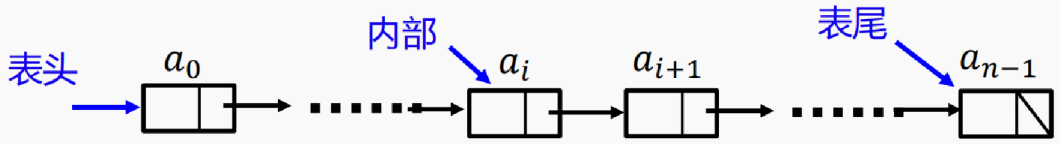
线性表特点：

- 操作灵活，长度可以增长、缩短

线性结构

- 二元组$B=(K,R)$ $K=\{a_0,a_1,\ldots,a_{n-1}\}$ $R=\{r\}$
  - 有一个唯一的**开始结点**，它没有前驱，有一个唯一的直接后继
  - 一个唯一的**终止结点**，它有一个唯一的直接前驱，而没有后继
  - 其它的结点皆称为**内部结点**，每一个内部结点都有且仅有一个唯一的直接有前驱，也有一个唯一的直接后继

    $<a_i,a_{i+1}>$ a_i是a_{i+1}的前驱，a_{i+1}是a_i的后继
  - 前驱/后继关系r，具有 **反对称性** 和 **传递性**



传递性示例：K1是K2的前驱，K2是K3的前驱，K1也是K3的前驱。

线性结构特点

- 均匀性：同一线性表中的元素具有相同的数据结构和长度
- 有序性：表里的元素都有一定的顺序

线性结构

- 按复杂程度：
  - 简单：线性表，栈，队列，散列表（索引+运算）
  - 高级：广义表，多维数组，文件...
- 按访问方式：
  - 直接访问型direct access
  - 顺序访问型sequential access
  - 目录索引型directory access
- 按操作划分
  - 线性表
    - 所有表目都是同一类型节点的线性表
    - 不限制操作形式
    - 根据储存不同分为：顺序表，链表

- 栈(LIFO, Last In First Out) 插入和删除操作都限制在表的同一端进行
- 队列 (FIFO, First In First Out) 插入操作在一端，删除操作在另一端

三个方面

- 线性表的逻辑结构
- 线性表的储存结构
- 线性表运算

逻辑结构

- 主要属性
  - 线性表长度
  - 表头
  - 表尾tail
  - 当前位置current position

储存结构

顺序表

- 按索引值**从小到大**存放在一片**相邻连续的区域**
- 紧凑结构，存储密度为1

链表

- 单链表
- 双链表
- 循环链表

线性表的运算

建立、清除、插入、删除、修改、排序、检索

```
template <class T> class List {
 void clear(); // 置空线性表
 bool isEmpty(); // 线性表为空时，返回 true
 bool append(const T value);
```

```
    // 在表尾添加一个元素 value，表的长度增 1
    bool insert(const int p, const T value);
    // 在位置 p 上插入一个元素 value，表的长度增 1
    bool delete(const int p);
    // 删除位置 p 上的元素，表的长度减 1
    bool getPos(int& p, const T value);
    // 查找值为 value 的元素并返回其位置
    bool getValue(const int p, T& value);
    // 把位置 p 元素值返回到变量 value
    bool setValue(const int p, const T value);
    // 用 value 修改位置 p 的元素值
};
```

线性表基本操作(运算)

```
Intlist(&L):初始化表。创造一个空的线性表L，分配储存空间。
DestoryList(&L):销毁操作。销毁线性表，并释放线性表L所占用的储存空间。

Listlnser(&L,i,e):插入操作。在第i个位置后插入指定元素e
ListDelete(&l,i, &e):删除操作。删除第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中察州具有给定关键字值的元素。
GetElem(L,i):按位查找。获取第i个位置的元素的值。

Length(L):求表长，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，返回true 1，否则返回false。
```

2.顺序表

- 也称向量，采用一定长度的一维数组储存结构
- 主要特性
  - 数据类型相同
  - 元素顺序地储存在连续地储存空间中，每个元素有位移索引值
  - 使用常熟作为向量长度
- 数组储存

- 读写其他元素很方便，通过下表即可指定位置
  - 只要确定了首地址，线性表中任何数据元素都可以随机存取

$$--Loc(ki) = Loc(k0) + c \times i, c = sizeof(ELEM)$$

顺序表的实现——静态分配

顺序表的长度从开始确认后就无法改变（存储空间是静态的。）

```
#define MaxSize 10//定义最大长度
typedef struct{
   ElemType data[MaxSize];//用静态”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）
```

顺序表的实现——动态分配

```
#define InitSize 10//顺序表的初始长度
typedef struct{
    ElemType *data;//只是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
}SqList;//顺序表的类型定义（动态分配）
```

C——malloc、free函数（`stdlib.h`）——动态申请和释放内存空间。

```
L.data=(ElemType*)malloc(sizeof(ElemType)*InitSize);
//malloc函数返回一个指针，需要强制转型为自己定义的数据元素类型指针
//realloc重新分配，增加动态分配数组长度
```

C++——new、delete函数

顺序表的实现——初始化表

```
Status InitList(SqList &L){
    L.elem= new ElemType[MAXSIZE];
    if(!L.elem) exit (OVERFLOW);
    L.length=0;
    return OK;
}
```

【算法步骤】
1.为顺序表L动态分配一个预定义大小的数组空间，使`elem`指向这段空间的基地址。
2.将表的当前长度设为0。

顺序表的插入

```
Status ListInsert (SqList &L,int i,ElemType e){
    if((i<1)||(i>L.length+1)) return ERROR;
    if(L.length==MAXSIZE) return EEROR;
    for(j=L.length-1;j≥i-1;j--)
        L.elem[j+1]=L.elem[i];
    L.elem[i-1]=e;
    ++L.length;
    return OK;
}
```

[算法步骤]

- 判断插入位置l是否合法(i 值的合法范围是$1 \leq i \leq n+1$), 若不合法则返回ERROR。
- 判断顺序表的存储空间是否已满，若满见肤返回ERROR。
- 将第n个至第i个位置的元素依次向后移动一个位置，空出第i个位置(i =n+1时无需移动)。
- 将要插入的新元素e放入第i个位置。
- 表长加1。

顺序表的删除

```
Status ListDelete(SqList &L,int i){
    if((i<1)||(i>L.length)) return ERROR;
    for(j=i;j≤L.length-1;j++)
        L.elem[j-1]=L.elem[j];
    --L.length;
    renturn OK;
}
```

[算法步骤]

- 判断删除位置i是否合法（合法值为$1 \leq i \leq n$), 若不合法则返回ERROR。
- 将第i+1个至第n个的元素依次向前移动一个位置(i = n时无需移动)。
- 表长减1。

顺序表的查找（按值查找）

```
int LocateElem(SqList L,ElemType e){
    for(i=0;i<L.length;i++)
        if(L.elem[i]==e)
            return i+1;
        else
            renturn 0;
}
```

【算法步骤】
1.从第一个元素起，依次和e相比较，若找到与e相等的元素 `L.elem[i]`，则查找成功，返回该元素的序号i+l。
2.若查遍整个顺序表都没有找到，则查找失败， 返回0。

顺序表的取值(按位查找)

```
Status GetElem(SqList L,int i,ElemType &e){
    if(i<1||i>L.length) return ERROR;//判断i值是否合理
    e=L.elem[i-1];
    return OK;
```

注意：

- 插入——移动n-i个元素
- 删除——移动n-i-1个元素

（n是元素个数，i是索引值从0开始）

i 的位置上插入和删除的概率分别是 p_i 和 p_i'

- 插入的平均移动次数为

$$M_i = \sum_{i=0}^{n}(n-i)p_i$$

- 删除的平均移动次数为

$$M_d = \sum_{i=0}^{n-1}(n-i-1)p_i'$$

如果在顺序表中每个位置上插入和删除元素的概率相同，即 $p_i = \frac{1}{n+1}$，$p_i' = \frac{1}{n}$

$$M_i = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(\sum_{i=0}^{n}n - \sum_{i=0}^{n}i)$$

$$= \frac{n(n+1)}{n+1} - \frac{n(n+1)}{2(n+1)} = \frac{n}{2}$$

$$M_d = \frac{1}{n}\sum_{i=0}^{n}(n-i-1) = \frac{1}{n}(\sum_{i=0}^{n}n - \sum_{i=0}^{n}i - n)$$

$$= \frac{n^2}{n} - \frac{(n-1)}{2} - 1 = \frac{n-1}{2}$$

时间代价为$O(n)$

优点：

1. 随机访问效率高
2. 储存密度高
3. 操作简单

缺点：

1. 插入和删除效率低

3.链表

通过指针把它的一串储存结点链接成一个链。

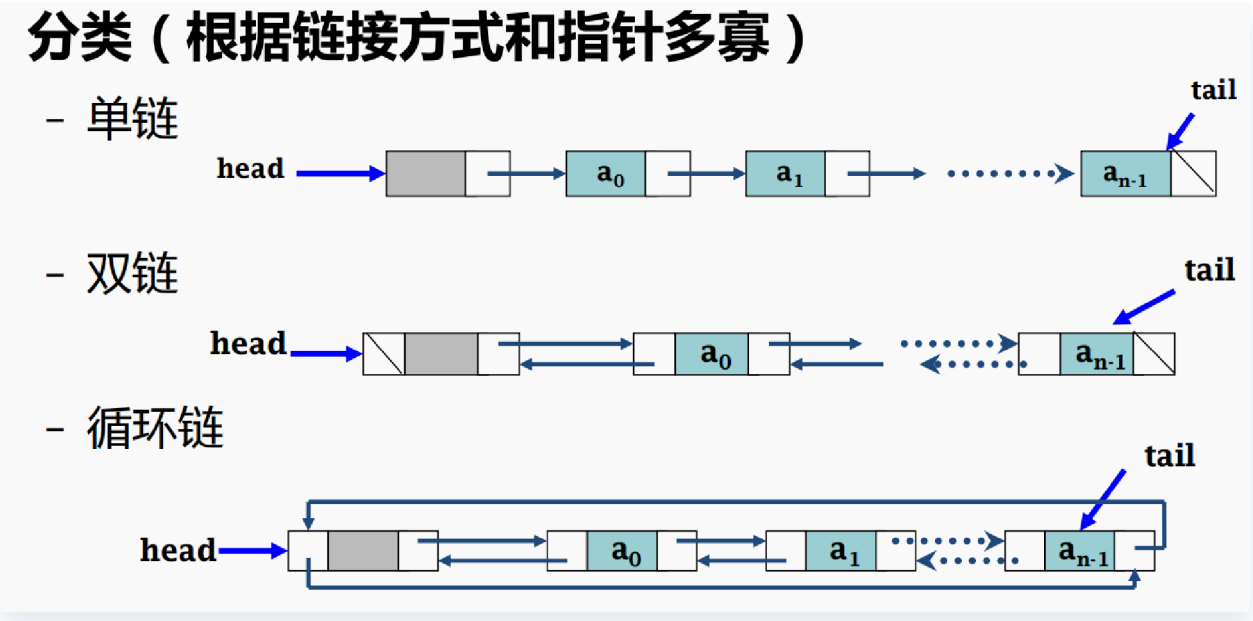
> 单链表的节点在存储空间中的分布可以是离散的，因此进行增删更改容量更方便。但查找某个元素时只能单个节点依次查找，效率低。
>
> 不支持随机存取。

储存节点由两部分组成；

数据域+指针域（后继地址）

data	next

分类



- 单链
- 双链
- 循环链

链表	简单的单链表	带头结点的单链表
整个单链表	head	head
第一个节点	head	head->next(虚结点，无意义)
空表判断	head=NULL	head->next==NULL
当前结点	`curr`	fence->next(`curr` 隐含)

单链表

单链表的定义

```
struct LNode{//定义单链表节点类型
    ElemType data;//每个几点存放一个数据元素
    struct LNode *next;//指针指向下一个节点
};
```

```
Sturct LNode *p=(struct LNode*)malloc(sizeod(struct LNode))
```

```
typedef<数据类型><别名>//数据类型的重命名

typedef struct LNode LNode;//省略struct，直接使用LNode
```

课本标准简洁代码

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList; //LinkList 为指向结构体LNode 的指针类型

//要表示单链表时只需要声明一个头指针L，指向单链表的第一个节点。
LNode * L;//声明一个指向单链表第一个节点的指针
//或者
LinkList L;//声明一个指向单链表第一个节点的指针，代码可读性更强


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

LNode * GetElem(LinkList L,int i){
//使用LNode*,强调这是一个结点；使用LinkList，强调这是一个单链表。
    int j=1;
    LNode *p=L-next;
    if(i==0)
        return L;
    if (i<1)
        return NULL;
    while(p≠NULL&&j<1){
        p=p→next;
        j++;
    }
    return p;
```

```
}
```

为了提高程序的可读性，在此对同一结构体指针类型起了两个名称， `LinkList` 与 `LNode*` ，两者本质上是等价的。通常习惯上用 `LinkList` 定义单链表，强调定义的是某个单链表的头指针；用 `LNode *` 定义指向单链表中任意结点的指针变量。

> 例如，若定义 `LinkList L` ,则L 为单链表的头指针，若定义 `LNode * p` , 则p 为指向单链表中某个结点的指针，用 *p 代表该结点。

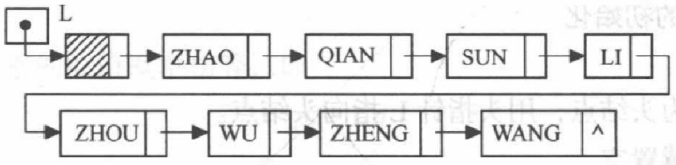


图 2.9　增加头结点的单链表的逻辑状态

下面对首元结点、头结点、头指针三个容易混淆的概念加以说明。

（1）首元结点是指链表中存储第一个数据元素 a_1 的结点。如图 2.8 或图 2.9 所示的结点“ZHAO”。

（2）头结点是在首元结点之前附设的一个结点，其指针域指向首元结点。头结点的数据域可以不存储任何信息，也可存储与数据元素类型相同的其他附加信息。例如，当数据元素为整数型时，头结点的数据域中可存放该线性表的长度。

（3）头指针是指向链表中第一个结点的指针。若链表设有头结点，则头指针所指结点为线性表的头结点；若链表不设头结点，则头指针所指结点为该线性表的首元结点。

链表增加头结点的作用如下。

(1)便于首元结点的处理

增加了头结点后，首元结点的地址保存在头结点（即其“前驱”结点）的指针域中，则对链表

的第一个数据元素的操作与其他数据元素相同，无需进行特殊处理。

(2)便于空表和非空表的统一处理

> 当链表不设头结点时，假设L 为单链表的头指针，它应该指向首元结点，则当单链表为长度n 为0 的空表时， L 指针为空（判定空表的条件可记为：L==NULL)。

```
p=L;//p指向头结点
p=L→next;//p指向首元结点
p=p→next;//p指向下一个结点
```

定义单链表

不带头节点的单链表

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

//初始化一个空的单链表
bool InitList(LinkList &L){
    L=NULL;//空表，暂时还没有任何节点
    return true;
}

void test(){
    LinkList L;//声明一个指向单链表的指针
    //初始化一个空表
    InitList(L);
    // ...后续代码...
}


//判断单链表是否为空
bool Empty(LinkList L){
    if(L=NULL)
        return true;
    else
        return false;
}
```

```
//或:
bool Empty(LinkList L){
    return (L=NULL);
}
```

带头节点的单链表

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

//初始化一个单链表（带头节点）
bool InitList (LinkList &L){
    L=(LNode *)malloc(sizeof(LNode));
    if(L==NULL)//内存不足，分配失败
        return false;
    L->next=NULL;//头节点之后暂时还么有节点
    return true;
}

void test(){
    LinkLIst L;//声明一个指向单链表的指针
    InitList(L);//初始化一个空表
    //。。。。
}

//判断单链表是否为空
bool Empty(LinkList L){
    if(L->next==NULL)
        return true;
    else
        return false;
}
```

示例

```
//储存学生学号、姓名、成绩的单链表结点类型的定义
typedef struct{
    char num[8];
    char name[8];
    int score;
}ElemType;

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

单链表初始化

```
Status InitList_L(LinkList &L){
    L=new LNode;//C语言: L=(LNode *)malloc(sizeof(LNode));
    L->next=NULL;
    return OK;
}
```

单链表的销毁

```
status DestroyList_L(LinkList &L){
    Lnode *p;//或LinkList p;
    while(L){//L≠NULL
        p=L;//p指向头节点
        L=L->next;
        delete p;
    }
    return OK;
}
```

单链表的清空

单链表仍存在，但单链表中无元素，成为空链表（头指针和头结点仍然存在）。

```
status ClearList_L(LinkList &L){
    Lnode *p,*q;
    p=L->next;
    while(p){//结束条件p==NULL
        q=p->next;
        delete p;
        p=q;
    }
    L->next=NULL;
    return Ok;
}
```

单链表——求表长

```
status ListLength_L(LinkList &L){
    Lnode *p;
    p=L->next;
    i=0;
    while(p){//p≠NULL;
        i++;
        p=p->next;
    }
    return i;
}
```

> 算法思路：从首元节点开始，依次计数所有结点。

单链表——按位查找

```
Status GeyElem_L(LinkList L,int i,ElemType &e){
    //初始化
    P=L->next;
    j=1;
    while(p&&j<i){//向后扫描，直到p指向第i个元素或p为空
        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;
}
```

单链表——按值查找

```
LNode *LocateElem_L(LinkList L,Elemtype e){
    p=L->next;
    while(p && p->data≠e)//运行条件：p不为空，当前结点所指地域的值不是要查找的值
        p=p->next;
    return p;
}
```

时间复杂度O(n)

按值查找获取位置序号

```
int LocalElem_L(LinkList L,Elemtype e){
    p=L->next;
    j=1;
    while(p&&p->data!=e)
    {
        p=p->next;
        j++;
    }
    if(p)
        return j++;
    else
        return 0;
}
```

单链表的插入

```
status ListInsert_L(LinkList &L,int i,ElemType e){
    p=L;
    j=0;
    while(p&&j<i-1){//寻找第i-1个结点，p指向i-1结点
        p=p->next;
        ++j;
    }
    if(!p||j>i-1) return ERROR;//判断插入位置是否非法
    s=new LNode;
    s->data=e;//生成新结点s，并将e赋值到s的数据域
    s->next=p->next;//将结点s插入L中
    p->next=s;
    return OK;
}
```

时间复杂度O(1)。

单链表的删除

```
Status ListDelete_L(LinkList &L,int i,ElemType &e){
    p=L;
```

```
    j=0;
    while(p->next&&j<i-1){
        p=p->next;
        ++j;
    }
    if(!(p->next)||j>i-1) return ERROR;
    q=p->next;//临时保存被删结点地址以备释放
    p->next=q->next;//改变删除结点前去结点的指针域
    e=q->data;//保存删除结点的数据域
    delete q;//释放删除结点空间
    return OK;
}
```

单链表的建立

头插法

元素插入在链表头部，又称前插法

- 从一个空表开始，重复读入数据
- 生成新结点，将读入数据存放到新结点的数据域中
- 从最后一个结点开始，一次将个结点插入到链表的前端

```
void CreateList_H(LinkList &L,int n){
    L=new LNode;
    L->next=NULL;
    for(i=n;i>0;--i){
        p=new LNode;
        cin>>p->data;
        p->next=L->next;//插入到表头
        L->next=p;
    }
}
```

尾插法

元素插入到链表尾部，也叫后插法

```
void CreatList_R(LinkList &L,int n){
    L=new LNode;
    L->next=NULL;
    r=L;
    for(i=0;i<n;i++){
        p= new LNode;
        cin>>p->data;
        p->next=NULL;
        r->next=p;//插入到尾部
        r=p;
    }
}
```

循环链表

定义及特点

循环链表：是一种头尾相连接的链表（即：表中最后一个结点的指针域指向头节点，整个链表形成环）

> 优点：从表中任一结点出发均可找到表中其他结点。

> 循环单链表的操作和单链表基本一致，差别仅在于：当链表遍历时，**判别当前指针p是否指向表尾结点的终止条件不同**。在单链表中，判别条件为 $p\neq NULL$ 或 $p\rightarrow next\neq NULL$,而循环单链表的判别条件为他们是否等于头指针 $p\neq L$ 或 $p\rightarrow next\neq L$ 。

- 头指针表示单循环链表
  - 找到A1的时间复杂度O(1)
  - 找到An的时间复杂度为O(n)，不方便。
- 尾指针表示单循环链表
  - A1的储存位置：R->next->next
  - An的储存位置：R

- 时间复杂度均为O(1)

> 所以使用带尾指针的循环列表更常用。

将带有尾指针的循环列表合并

- p存表头结点
- Tb表头连接到Ta表尾
- 释放Tb表头结点
- 修改指针

```
LinkList Connect(LinkList Ta,LinkList Tb){
    //Ta、Tb均为非空单循环链表
    p=Ta→next;//p存表头结点
    Ta→next=Tb→next→next;//Tb表头连接到Ta表尾
    delete Tb→next;//释放Tb表头结点
    Tb→next=p;//修改指针
    return Tb;
}
```

双向链表

双向链表：在单链表的每一个结点里再增加一个指向其直接前驱的指针域prior，这样链表中就形成了有两恶搞方向不同的链。

双向链表结点结构

prior	data	next

双向链表定义

```
Typedef struct DuLNode{
    ElemType data;
    struct DuLNode *prior,*next;
}DuLNode,*DuLinkList;
```

双向循环链表

- 让头节点的前驱指针指向链表的最后一个结点
- 让最后一个结点的后继指针指向头结点

特性

1. 对称性 `p->prior->next = p = p->next->prior`
2. 操作仅插入和删除时与单链表不同，需要同时修改两个方向上的指针，两者操作的时间复杂度均为O(n)。

双向链表的插入

```
void ListInsert_DuL(DuLinkList &L,int i,ElemType e){
    if(!(p=GetElemP_DuL(L,i))) return ERROR;
    s=new DuLNode;
    s->data=e;
    s->prior=p->prior;//1
    p->prior->next=s;//2
    s->next=p;//3
    p->prior=s;//4
    return OK;
}
```

双向链表的删除

```
void ListDelete_DuL(DuLink &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;
    delete p;
    return OK;
}
//O(1)
```

4.区分对比

顺序表和链表对比

特性	顺序表	链表
存储结构	连续的内存空间	离散的节点通过指针连接
类型	-	**单链表**、**双链表**、**循环链表**（可基于单链表或双链表实现）
特点	- 元素物理地址连续 - 支持随机访问	- 元素物理地址不连续 - 动态分配内存 - 插入/删除效率高
优点	1. 访问速度快（O(1)） 2. 内存占用少（仅存储数据）	1. 插入/删除灵活（O(1)） 2. 无需预分配空间，动态扩展 3. 双链表支持双向遍历 4. 循环链表适合环形结构需求
缺点	1. 插入/删除需移动元素（O(n)） 2. 大小固定（需预分配或扩容）	1. 访问速度慢（O(n)） 2. 额外存储指针占用空间
访问方式	随机访问（直接通过下标）	顺序访问（从头节点遍历）
插入/删除时间复杂度	平均 O(n)（需移动元素）	已知位置时 O(1)（修改指针） 查找位置时 O(n)
空间分配	静态分配或动态分配	动态分配节点
适用场景	查询频繁、数据量固定或变化较小	频繁插入/删除、数据量动态变化 - 双链表：需双向操作（如撤销） - 循环链表：轮询任务管理

链表子类对比

链表类型	单链表	双链表	循环链表
节点结构	数据 + **指向下一节点**的指针	数据 + **指向前后节点**的两个指针	尾节点指向头节点（单/双链表均可实现循环）

链表类型	单链表	双链表	循环链表
遍历方向	单向	双向	单向或双向（取决于底层链表类型）
优点	结构简单，内存占用少	支持双向遍历，操作更灵活	适合循环操作（如轮询调度）
缺点	无法逆向遍历，删除节点需前驱指针（O(n)）	占用更多内存（多一个指针）	需注意循环终止条件（避免死循环）

总结

- **顺序表**：适合**读多写少**的场景（如数组操作）。
- **链表**：适合**频繁插入/删除**的场景（如队列、图邻接表）。
- **双链表**：需要双向操作时使用（如浏览器前进后退）。
- **循环链表**：适合环形数据处理（如操作系统进程调度）。