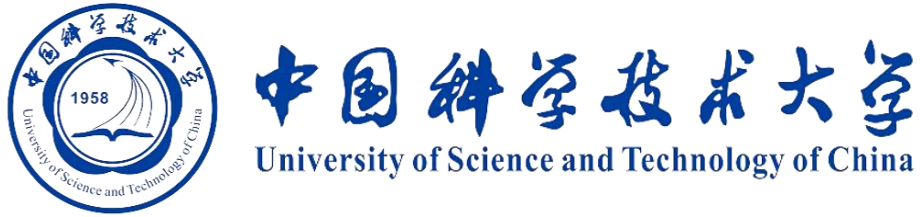


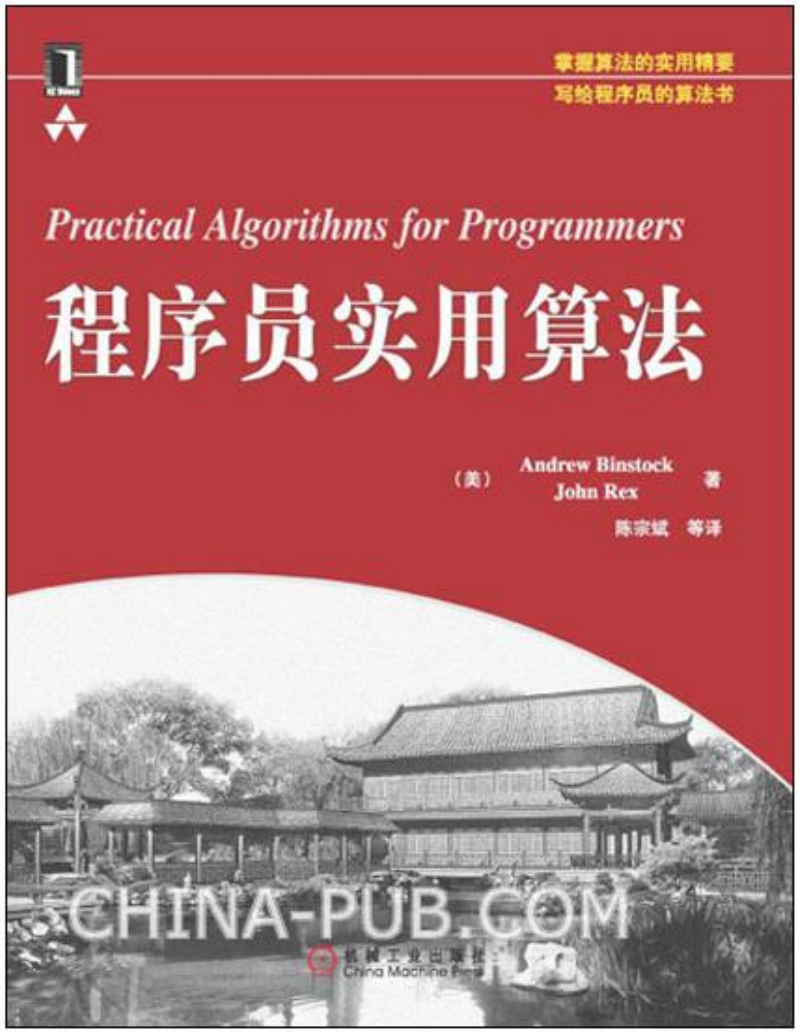
中国科学技术大学
University of Science and Technology of China

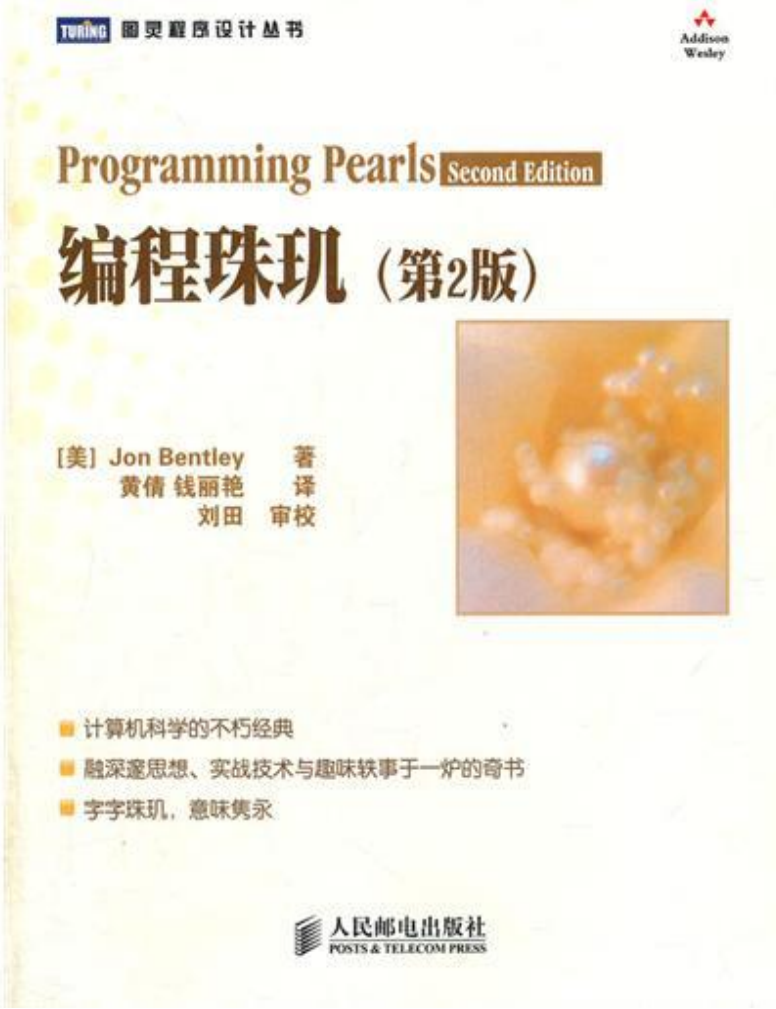
实用算法设计

➢ 线性结构

主讲：娄文启

参考资料


掌握算法的实用精要
写给程序员的算法书
Practical Algorithms for Programmers
程序员实用算法
(美) Andrew Binstock
John Rex 著
陈宗斌 等译
CHINA-PUB.COM
China Machine Press


TURING 图灵程序设计丛书
Addison Wesley
Programming Pearls Second Edition
编程珠玑（第2版）
[美] Jon Bentley 著
黄倩 钱丽艳 译
刘田 审校
计算机科学的不朽经典
融深邃思想、实战技术与趣味轶事于一炉的奇书
字字珠玑，意味隽永
人民邮电出版社
POSTS & TELECOM PRESS

本节重难点

- **重点：**
  - 理解线性表的逻辑特点；分别掌握线性表两种存储结构的特点、实现（C定义）及其基本操作的实现、适用范围；
  - 掌握算法的描述方法：伪代码和算法流程图；会粗略分析算法的时间复杂度（三种情况下）和内存开销。
- **难点：理论的落地应用。对于给定的应用需求，**
  - 会对所操作的数据对象进行分析，并判断是否需要用到DS?
  - 会判断是否适合用线性表来刻画同类数据之间的关系？（逻辑结构）
  - 会设计线性表的存储结构（包括：存储哪些数据（包括数据类型和取值）？顺序存储/链式存储的选择？）　（数据+存储结构）
  - 会编码实现数据的存储结构；
  - 会利用线性表的基本操作来解决问题。

3 线性表

3.1 数据结构

- 逻辑结构、存储结构/物理结构

3.2 什么是线性表

3.3 顺序表的定义及实现

3.4 链表的定义及实现

- 单链表、双向链表

3.5 案例分析

3.6 顺序表的延伸：位向量/位图

3.1 数据结构

- 数据结构：性质相同的数据元素的有限集合及其上的关系的有限集合。
  （数据+结构）

- 是描述现实世界实体的数据模型及其上的操作在计算机上的表示和实现。

- 数据结构包括逻辑结构和存储结构/物理结构。

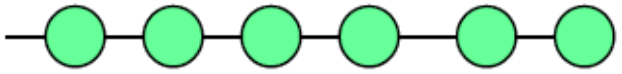
3.1 数据结构——逻辑结构

- 逻辑结构：
  - 从逻辑关系上对数据结构进行描述
  - 是从具体问题抽象出来的数据模型
  - 与数据本身的存储（数据元素的存储位置、类型和具体取值）无关。

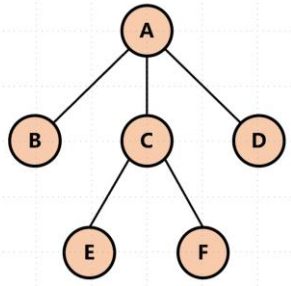
如：线性结构：线性表；
　　非线性结构：树、图；

3.1 数据结构——逻辑结构

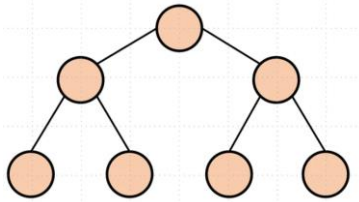
- 四类数据结构（D+S）：
  - 集合结构：
    - 元素间无任何关系，即关系集合是空集：S={}
  - 线性结构：如，线性表。
    - 元素间的关系是1:1
    - 除头结点外，所有结点有且仅有一个直接前驱；
    - 除尾结点外，所有结点有且仅有一个直接后继。
  - 树形结构：如，一般树、二叉树、森林
    - 一个结点可有多个直接后继（除叶子结点外）
    - 但只有一个直接前驱（除根结点外）；
  - 图形结构：
    - 元素间的关系是m:n；
    - 一个结点可以有多个直接后继，也有多个直接前驱。



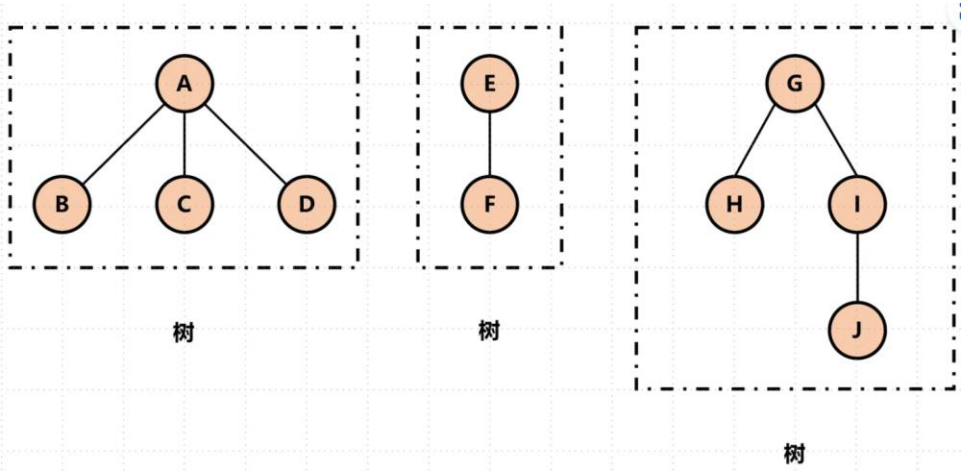
一般树



二叉树



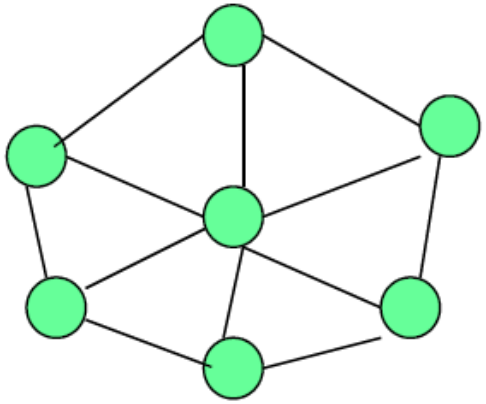
森林



3.1 数据结构——逻辑结构

- 四类数据结构（D+S）：
  - 集合结构：
    - 元素间无任何关系，即关系集合是空集：S={}
  - 线性结构：如，线性表。
    - 元素间的关系是1:1
    - 除头结点外，所有结点有且仅有一个直接前驱；
    - 除尾结点外，所有结点有且仅有一个直接后继。
  - 树形结构：如，一般树、二叉树、森林
    - 一个结点可有多个直接后继（除叶子结点外）
    - 但只有一个直接前驱（除根结点外）；
  - 图形结构：
    - 元素间的关系是m:n；
    - 一个结点可以有多个直接后继，也有多个直接前驱。

图形结构



3.1 数据结构——存储结构/物理结构

- 数据的存储结构/物理结构：
  - 数据结构在计算机中的表示（或映像）
    - 顺序映像：顺序存储结构
    - 非顺序映像：链式存储结构
    - 如，线性表有顺序表和链表两种物理存储方式
  - 它可以借助于具体某程序语言中的“数据类型” 来定义它。也可采用typedef将类型名重命名，以增加代码的可读性。

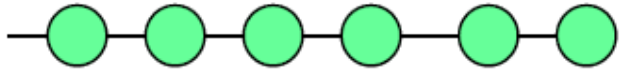
```
int Sqlist[100];
struct Node{
    int data;
    struct Node *next
};
Typedef struct Node *Link;
Link head;
```

“顺序表”数据存储结构的实现；
描述了100个int型变量组成的集合，且隐含着可利用下标[]来描述两个int型变量间的联系.

“单链表”数据存储结构的实现；
隐含着可利用指针next来描述两个struct Node类型的变量之间的联系.

3.2 线性表

- 什么是线性表?
  - 有限个数据元素组成的序列，记作(a1,a2, … , an)
- 逻辑特征：线性特征



- 存储结构：顺序表和链表两种方式
- 基本操作:
  - 创建空的线性表
  - 销毁已有线性表
  - 查找直接后继和直接前驱
  - 插入一个元素
  - 删除一个元素

例题1

- 问题1：读入一个包含城市和气温信息的数据文件，要求将记录按照气温和城市名称升序插入在表中，丢弃重复的记录，然后打印该有有序排列，并指示位于中间的条目。接着，逐渐缩短表并重新打印显示中间条目。其中，文件中每行为一条数据记录，它的前3个字符表示气温，其后的最多124个字符表示城市名称。如“-10Duluth”。

- 请指出上述问题中：
  - 1）是否需用到数据结构？
  - 2）会用到哪种类型的逻辑结构？集合、线性、树形结构、图形结构？
  - 3）存储结构是怎样的？
    - 存储哪些数据？（包括数据类型？）
    - 顺序存储/链式存储的选择？数组（多大？）

例题1 扩展

- 问题1：读入一个包含城市和气温信息的数据文件，要求将记录按照气温和城市名称升序地打印出来，并指示位于中间的条目。其中，文件中每行为一条数据记录，它的前3个字符表示气温，其后的最多124个字符表示城市名称。如“-10Duluth”。

- 请指出上述问题中：
  - 1）是否需用到数据结构?
  - 2）可以使用哪几种逻辑结构？（不限于线性结构）
  - 3）对应的存储结构分别是怎样的?
    - 存储哪些数据（包括数据类型和取值）？
    - 顺序存储/链式存储的选择?

总结

- 什么时候需要用到数据结构？
  - 操作对象为：取值为同种类型的很多数据，且这些数据间存在某种关系 或者 某些共性操作
- 若需要用到DS，那什么时候可以使用线性结构？
  - 被操作的数据对象之间没有天然的一对多 和 多对多的关系
  - 对已存储的数据进行处理时，处理顺序 有明显的唯一的先后次序关系
- 若采用线性结构，具体该使用哪种存储结构（此处，只讨论：顺序表和链表）
  - 取决于数据处理时的最频繁操作，为静态操作，还是动态操作？

3.3 顺序表的定义及实现

- 顺序表：在内存中连续存储的线性表
- 特性：
  - 逻辑上相邻的元素，物理存储地址必相邻；
  - 可随机存取：通过顺序表的名称和下标可以直接访问顺序表中的任一个元素。

- Q：对于顺序表A中下标为i的元素，它的直接前驱为？直接后继为？
- 结论：顺序表中用下标来表明线性特征。

顺序表——静态定义

- 顺序表的静态定义：利用数组。
- 方案一：

```
int Sqlist[100];
```

- 方案二：

```
#define List_Size 100 /*分配空间的大小*/
int Sqlist[List_Size ];
```

- 方案三：

```
#define List_Size 100/*分配空间的大小*/
Typedef Struct{
    int elem[List_Size ]; /*存储空间*/
    int len;       /*实际长度*/
}SqList_static;
```

(通用性最强，类似STL:vector)

1. 只需定义 单个结点和结点长度，就可以实现“顺序表”数据存储结构的定义；
2. 元素之间的关系隐式地用下标[ ]来描述。

顺序表——静态定义

- 评价：
  - 该结构比较机械：分配的内存空间大小固定。
    - List_Size 过小，会导致顺序表上溢；
    - List_Size 过大，会导致空间利用率不高
  - 在编译的时候，定义在函数内则系统在栈中分配连续的内存空间。当静态顺序表所在的函数执行完毕后，由系统来回收所开辟的内存空间。定义全局或static则分配在读写数据段
  - 程序运行时，出现上溢问题，将没法修补。

C++ Vector?

顺序表——动态定义

- 顺序表的动态定义：利用指针。

```
#define List_Size 100 /*分配空间的大小*/
typedef struct{
    int *elem;          /*顺序表的存储空间*/
    int len;            /*实际长度*/
    int ListSize ;      /*当前分配的空间大小*/
} Sqlist;
```

1. 数组≠顺序表.
2. 并不是只有链表中才能有指针.

- 特点：
  - 需手动分配存储空间:malloc()
  - 可以在程序运行过程中，重新分配空间：realloc()
  - 不再使用顺序表时，需手动释放所占的空间：free()
  - 可以避免“机械”，但是会增加时间开销。

顺序表——动态定义

- C中的动态分配与释放函数
  - `Void *malloc(unsigned int size)`

    `/*生成一个大小为size的结点空间，将该空间的起始地址赋给p*/`
  - `Void Free(void *p)`

    `/*回收p所指向的结点空间*/`
  - `Void *realloc(void *p, unsigned int size)`

    `/*重新分配大小为size的结点空间，并将该空间的起始地址赋给p */`

顺序表——动态定义


```
Int InitSqList(SqList *L)//构造一个空的顺序表L
{
    L->elem=(int *) malloc(List_Size *sizeof(int));
    if (L->elem==NULL)
        exit(EXIT_FAILURE);
    L->len=0;
    L->ListSize =List_Size;
    return 1;
}
```

预设的空间不够用了，可使用realloc函数重新分配

原始数据怎么处理?

顺序表——动态定义

1

```
int main() {
    // 原始内存块的大小
    size_t original_size = 10;
    // 分配初始内存
    int *original_data = (int
*)malloc(original_size * sizeof(int));
    if (original_data == NULL) {
        fprintf(stderr, "memory allocation
fail\n");
        return 1;}
```

2

```
    // 在原始内存中填充一些数据
    for (size_t i = 0; i < original_size; ++i) {
        original_data[i] = i;
    }
    // 输出原始数据
    printf("orinial data: ");
    for (size_t i = 0; i < original_size; ++i) {
        printf("%d ", original_data[i]);
    }
```

3

```
    // 调整内存大小为新的大小
    size_t new_size = 20;
    int *new_data = (int *)realloc(original_data, new_size
* sizeof(int));

// 如果内存被移动，new_data将是新分配内存的地址，original_data
将是NULL
    if (new_data != original_data) {
        printf(“address mismatch\n”); }

    // 在新内存中继续填充一些数据
    for (size_t i = original_size; i < new_size; ++i) {
        new_data[i] = i;
    }

    // 输出新数据
    printf("New Data: ");
    for (size_t i = 0; i < new_size; ++i) {
        printf("%d ", new_data[i]);
    }
    free(new_data);
}
```

顺序表——动态定义


```
orinial data: 0 1 2 3 4 5 6 7 8 9
New Data: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

没有发生数据复制

将上述原始空间从10->1024，新空间从20->2048后：

```
orinial data: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
address mismatch
New Data: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

地址不同，但之前的数据已经复制到新的内存块

继续看看Vector数据类型

顺序表——动态定义（Vector）

1

```
#include<iostream>
class Object{
public:
// 构造函数
Object(int v_):v(v_){
    std::cout << "Construct Object" << std::endl;}
    // 拷贝构造函数
    Object(const Object& other):v(other.v){
    std::cout << "Copy Object" << std::endl;}
    private:
    int v;
};
```

2

```
int fun1(Object b){
    std::cout<<"fun1"<<std::endl;  return 0;}

int fun2(Object &b){
    std::cout<<"fun2"<<std::endl;  return 0;}
```

3

```
int main(){
    Object obj1(1);
    Object obj2(obj1); cout<<endl;
    Object obj3=obj1; cout<<endl;
    fun1(obj1); cout<<endl;
    fun2(obj1);}
```

输出：？

```
Construct Object
Copy Object

Copy Object

Copy Object
fun1

fun2
```

顺序表——动态定义（Vector）


```
std::vector<Object> v;
//向vector添加一个Object
Object obj1(1);
v.push_back(obj1);
```

1

输出?

```
Construct Object
Copy Object
```

思考:

为什么不是两次拷贝,
(**传参**拷贝构造 + Vector内部**拷贝构造**)?

```
void push_back(const value_type& __x)
```

```
std::vector<Object> v;
Object obj1(1);
v.push_back(obj1);
v.push_back(obj1);
```

2

输出?

```
Construct Object
Copy Object
Copy Object
Copy Object
```

思考:

为什么多拷贝了一次,
(明明添加了两次)?

顺序表——动态定义（Vector）

当vector空间不足时:

1. 分配新的内存空间： 通常是当前容量的两倍(2的指数)。
2. 将元素从旧内存复制到新内。
3. 释放旧的内存空间。

```
std::vector<Object> v;
Object obj1(1);
v.push_back(obj1);
v.push_back(obj1);
v.push_back(obj1);
```

3

输出？

```
Construct Object
Copy Object
Copy Object
Copy Object
Copy Object
Copy Object
Copy Object
```

输出？

```
Construct Object
Copy Object
Copy Object
Copy Object
```

可以通过预先分配地址来避免额外的复制：v.reserve(3);

emplace_back可进一步改善添加临时变量时的构造后拷贝过程

顺序表——两种定义的对比总结

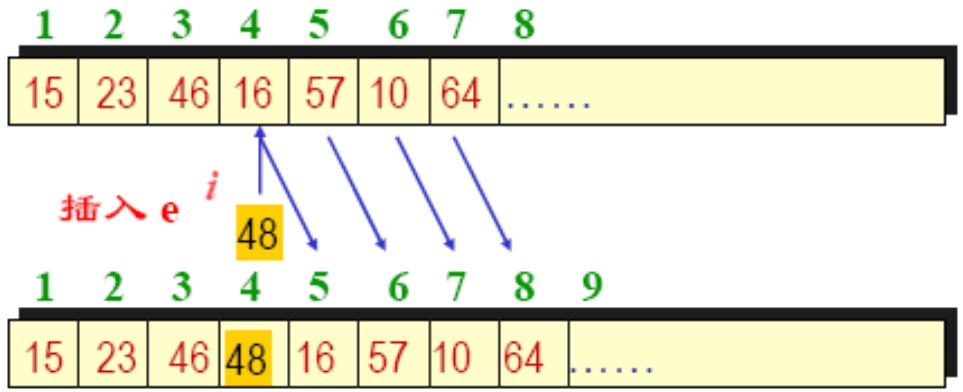
- 两种顺序表定义的对比：
  - 不同：分配存储空间的方式不同
    - 静态定义的顺序表由系统自动分配和回收存储空间；
    - 动态定义的顺序表需要手动分配和回收存储空间，但是也可以再分配。
    - 因而它们创建和销毁顺序表两种操作的实现不同，
  - 相同点：本质上都是存储在连续空间上，因而对数据的操作方式（查找，插、删）都是一样的。

例题2

- Q：分别利用静态定义的顺序表和动态定义的顺序表，来实现如下问题中的数据结构
  - 问题2：读入一个包含学生姓名和年龄的数据文件，要求将记录按照年龄和学生姓名升序地插入到一个链表中，丢弃重复的记录，然后打印该有有序链表。若键盘输入一个学生的信息为“A24张三”，则将该学生的信息插入到有序链表中，并丢弃重复的记录，然后打印该有有序链表。若键盘输入一个学生的信息为“D24张三”，则将该学生的信息从该有序链表中删除，然后打印该有有序链表。其中，文件中每行为一条数据记录，它的前2个字符表示年龄，其后的最多20个字符表示学生姓名。如“22斯琴高娃”。

顺序表——插入操作的实现


```
Status  ListInsert_Sq( SqList &L, int i, ElemType e)
```



顺序表——插入操作的实现

- 参数：顺序表&L、插入位置i、插入元素e
- 插入分析：
  - 第i个位置放e，则原来第i~L.len个数据元素必须先后移，以腾出第i个位置;
  - 后移的顺序为：从最后一个元素开始，逐个往后移

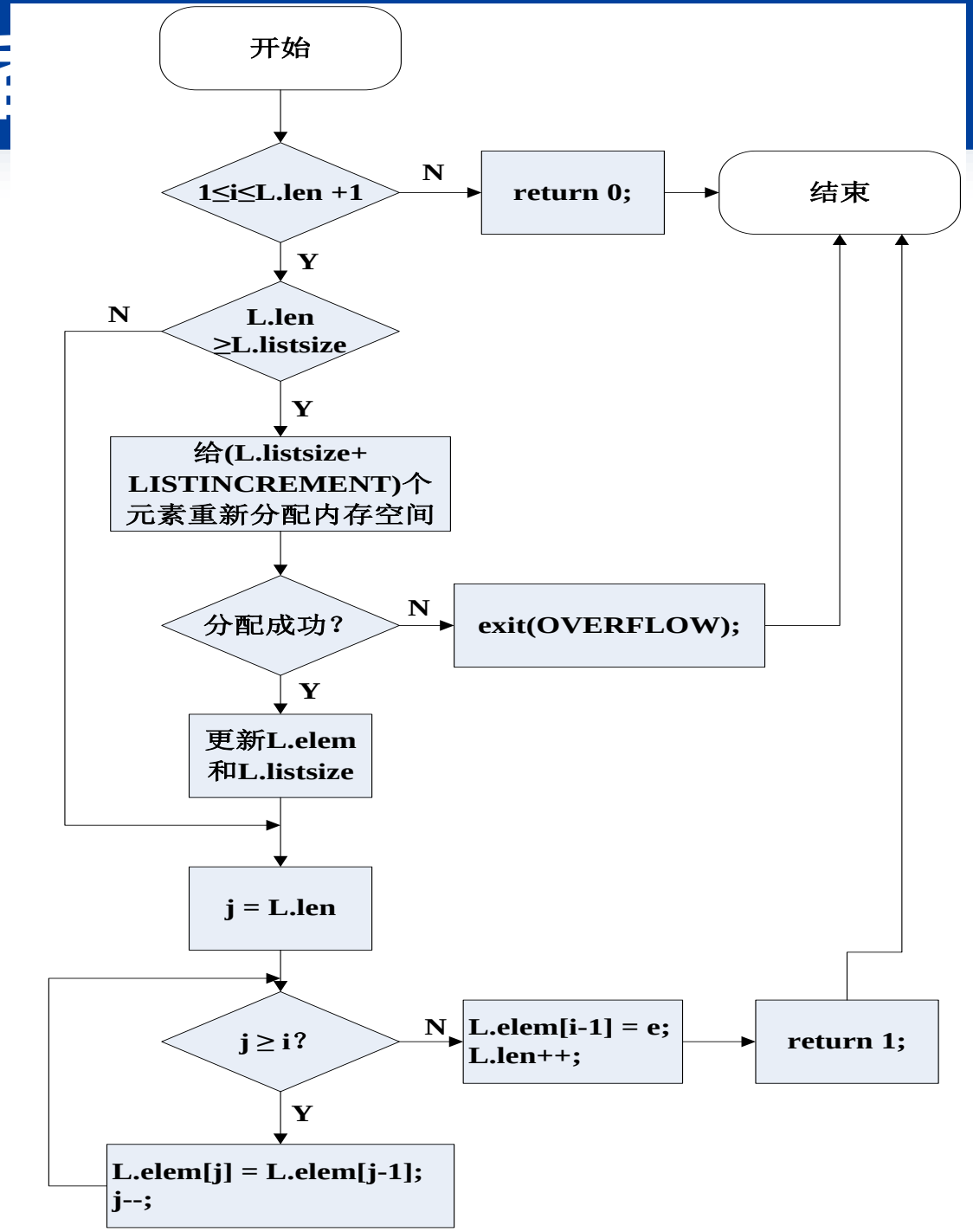
```
for ( j = L.len; j >= i; j--)
    L.elem[j] = L.elem[j-1];          // 后移
L.elem[i -1] = e; //第i个元素存放在L.elem[i-1]中，数组下标的起始为0
L.len = L.len+1;
```

- 合法的位置：i:1..L.len+1
- 上溢及处理：
  - 上溢发生的条件：L.len≥L.ListSize
  - 处理：要先申请一个有一定增量的空间：申请成功则原空间的元素复制到新空间，修改L.listsize，再进行插入工作；否则报错退出。

健壮性

顺序表——插

算法流程图：


开始
1≤i≤L.len +1
N
return 0;
结束
Y
L.len ≥L.listsize
N
Y
给(L.listsize+ LISTINCREMENT)个元素重新分配内存空间
分配成功？
N
exit(OVERFLOW);
Y
更新L.elem和L.listsize
j = L.len
j ≥ i?
N
L.elem[i-1] = e;
L.len++;
return 1;
Y
L.elem[j] = L.elem[j-1];
j--;

顺序表——插入操作的实现

- **算法的伪代码：**

		cost	times
	ListInsert_Sq(L, i, e)		
	▸ 位置合法性的判断: 1 ≤ i ≤ n+1		
1	**if** i<1 or i>length[L]+1	C1	1
2	**then** return 0	C2	0/1
	▸ 上溢时增加空间的分配		
3	**if** length[L] >= listsize[L]	C3	1
4	**then** newbase←reallocate more memory spaces	C4	0/1
5	**if** newbase = NIL	C5	0/1
6	**then** error" OVERFLOW"	C6	0/1
7	elem[L] ← newbase	C7	0/1
8	listsize[L] ← size of new spaces	C8	0/1
	▸ 插入元素		
9	**for** j ← length[L] to i	C9	n-i+2
10	**do** L[j] ← L[j-1]	C10	n-i+1
11	L[i-1] ← e	C11	1
12	length[L] ← length[L]+1	C12	1
13	return 1	C13	1

Ci表示第i条语句的实际执行时间

顺序表——插入操作的实现

- **算法的伪代码：**

		cost	times
	ListInsert_Sq(L, i, e)		
	▸ 位置合法性的判断: 1 ≤ i ≤ n+1		
1	**if** i<1 or i>length[L]+1	C1	1
2	**then** r		0/1
	▸ 上溢时		
3	**if** leng		1
4	**ther**		0/1
5			0/1
6			0/1
7	elem[L] ← newbase		0/1
8	listsize[L] ← size of new spaces	C8	0/1
	▸ 插入元素		
9	**for** j ← length[L] to i	C9	n-i+2
10	**do** L[j] ← L[j-1]	C10	n-i+1
11	L[i-1] ← e	C11	1
12	length[L] ← length[L]+1	C12	1
13	return 1	C13	1

1. 总的实际执行时间 T = ？（分三种情况考虑）
2. 估算最频繁语句的执行次数，用O来描述该算法的时间复杂度。（分三种情况考虑）

Ci表示第i条语句的实际执行时间

顺序表——插入操作的实现

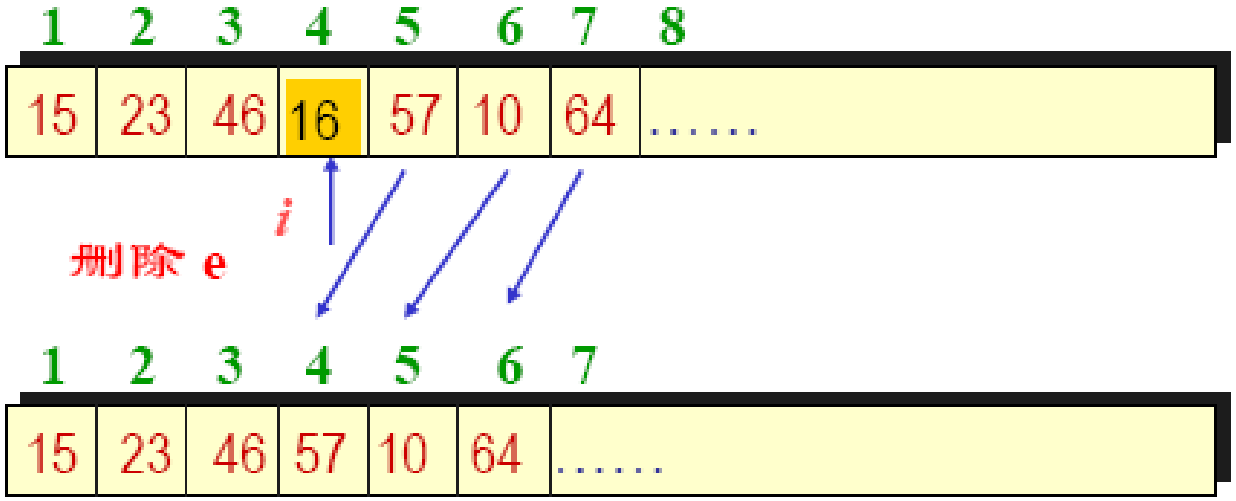
- 时间复杂度分析：
  - 频次最高的操作：移动元素。
  - 上溢时的空间的再分配与复制(realloc操作)的时间复杂度与realloc的算法以及当前的表长相关，至少为O(n)；但它仅在插入元素会引起上溢时才执行。
  - 若线性表的长度为n，则：
  - *最好情况*：插入位置i为n+1，此时无须移动元素，时间复杂度为O(1);
  - *最坏情况*：插入位置i为1，此时须移动n个元素，时间复杂度为O(n);
  - *平均情况*：假设pi为在第i个元素之前插入一个元素的概率，则：
    插入时移动次数的期望值：等概率时，即，
    ∴ T(n) = O(n)

顺序表——插入操作的实现


```
Status  ListInsert_Sq( SqList &L, int i, ElemType e)
{
    // 位置合法性的判断
    if ( i<1 || i>L.len +1 )    return 0;
    // 上溢时增加空间的分配
    if( L.len >= L.listsize)
    {
        newbase = (ElemType *) realloc(L.elem,
                    (L.listsize+ LISTINCREMENT)*sizeof(ElemType));
        if ( newbase == NULL ) exit(OVERFLOW);
        L.elem = newbase;
        L.listsize += LISTINCREMENT;
    }
    // 插入元素
    for ( j = L.len; j >= i; j--)   L.elem[j] = L.elem[j-1];
    L.elem[i-1] = e;
    L.len++;
    return 1;
}
```

顺序表——删除操作的实现


```
Status  ListDelete_Sq( SqList &L, int i, ElemType &e)
```



顺序表——删除操作的实现

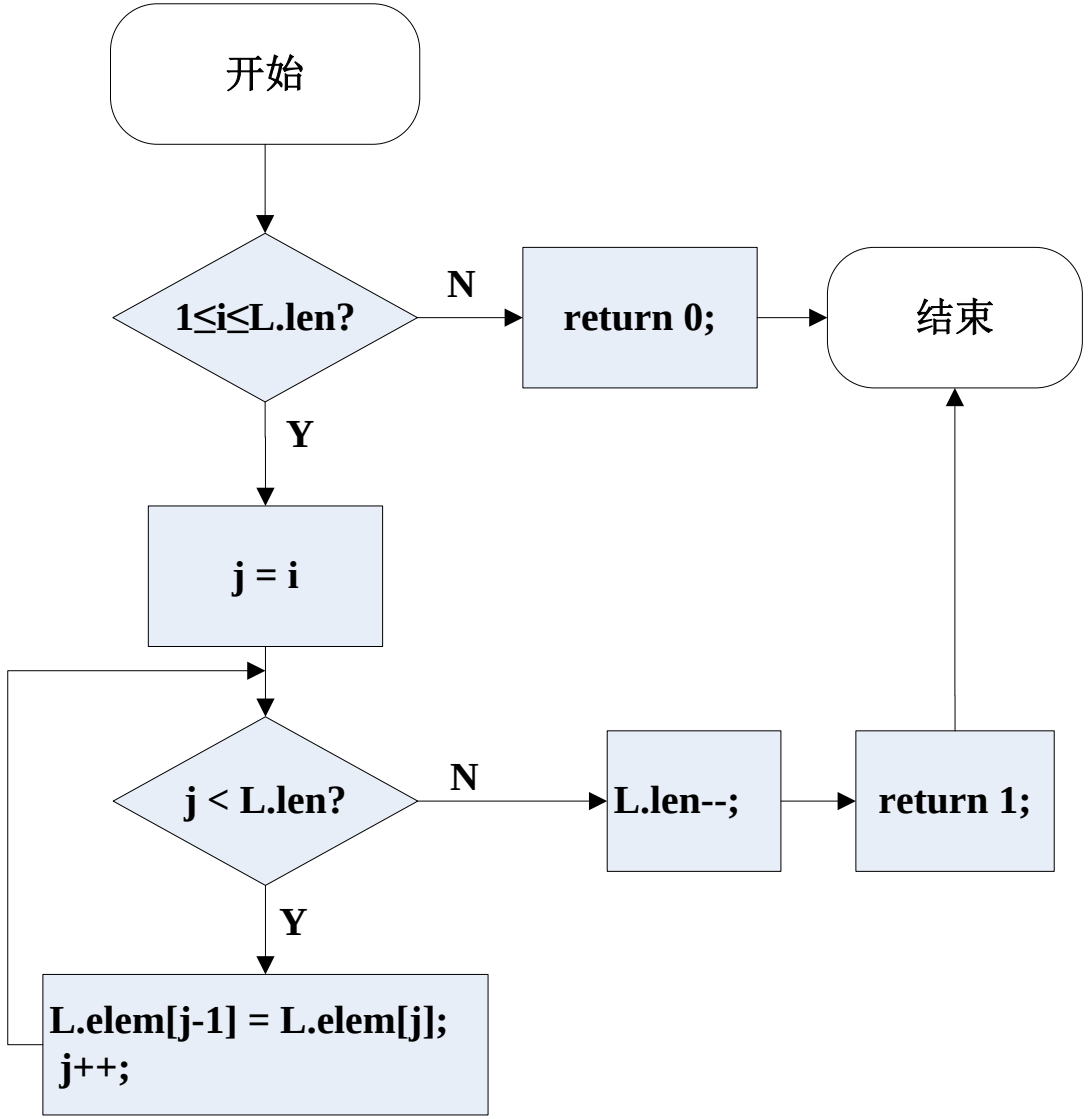
- **参数：**顺序表&L、删除位置i
- **删除分析：**
  - **去掉第i个元素，则原来第i+1~L.len个数据元素须前移，以覆盖第i个位置；**
  - **前移的顺序为：**

```
for ( j = i; j < L.len ; j++)
    L.elem[j-1] = L.elem[j];
L.len = L.len-1
```

- **合法的位置：**i:1..L.len
- **下溢：**L.len≤0 —— 隐含在i的条件中

顺序表——删除操作的实现

- 算法流程图为：



顺序表——删除操作的实现

- 算法伪代码为：

```
Status  ListDelete_Sq( SqList &L, int i) {
    // 位置合法性的判断
    if ( i<1 || i>L.len )   return 0;
    // 删除
    for ( j = i; j < L.len ; j++)
        L.elem[j-1] = L.elem[j];
    L.len--;
    return 1;
}
```

顺序表——删除操作的实现

- 时间复杂度分析：
  - 频次最高的操作：移动元素
  - 若线性表的长度为n，则：
    - 最好情况：删除位置i为n，此时无须移动元素，时间复杂度为O(1);
    - 最坏情况：删除位置i为1，此时须前移n-1个元素，时间复杂度为O(n);
    - 平均情况：假设qi为删除第i个元素的概率，则删除时移动次数的期望值：

$$E_{de} = \sum_{i=1}^{n} q_i (n - i)$$

等概率时，即，

$$q_i = \frac{1}{n} \qquad E_{de} = \frac{1}{n} \sum_{i=1}^{n} (n - i) = \frac{n-1}{2}$$

- ∴ T(n) = O(n)

例题 4

给你一个顺序表 L 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

例如：L = [3, 2, 2, 3], val = 3。新的长度为 2，且数组前两个元素均为 2。

例题 4

给你一个顺序表 L 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

解法 1：循环遍历数组，每当遇到 val 时，调用删除操作

```
for (int i = 0; i < L.len; i++) {
    if (L.elem[i] == val) {
        ListDelete_Sq(L, i);
    }
}
```

时间复杂度分析：删除操作的复杂度为 O(n)，外层循环 n 次，因此时间复杂度为 $O(n^2)$

例题 4

给你一个顺序表 L 和一个值 val，你需要 原地 移除所有数值等于 val 的元素，并返回移除后数组的新长度。

解法 2：双指针法

快指针：寻找新数组的元素（值不为val）

慢指针：指向新数组的更新位置

```
int slowIndex = 0;
for (int fastIndex = 0; fastIndex < L.len; fastIndex++)
{
    if (val != L.elem[fastIndex]) {
        L.elem[slowIndex++] = L.elem[fastIndex];
    }
}
return slowIndex;
```

时间复杂度分析：循环内的复杂度为 O(1)，因此时间复杂度为 O(n)。

课堂练习

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。

解法 1：双指针法

快指针：表示遍历数组到达的下标位置

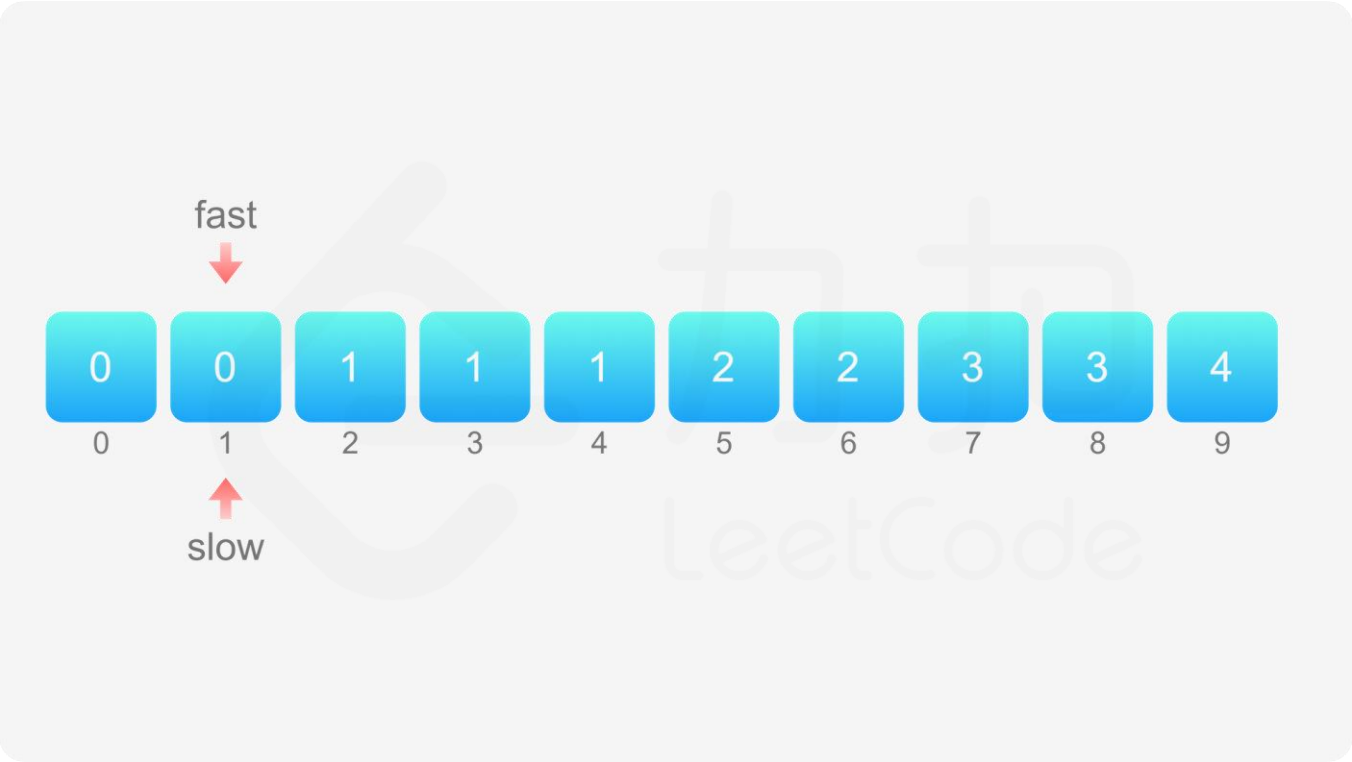
慢指针：表示下一个不同元素要填入的下标位置

如果 nums[fast] != nums[fast−1]，说明 nums[fast] 和之前的元素都不同，因此将 nums[fast] 的值复制到 nums[slow] ， 然后将 slow 的值加 1，即指向下一个位置。

遍历结束之后，slow 即为新的长度。

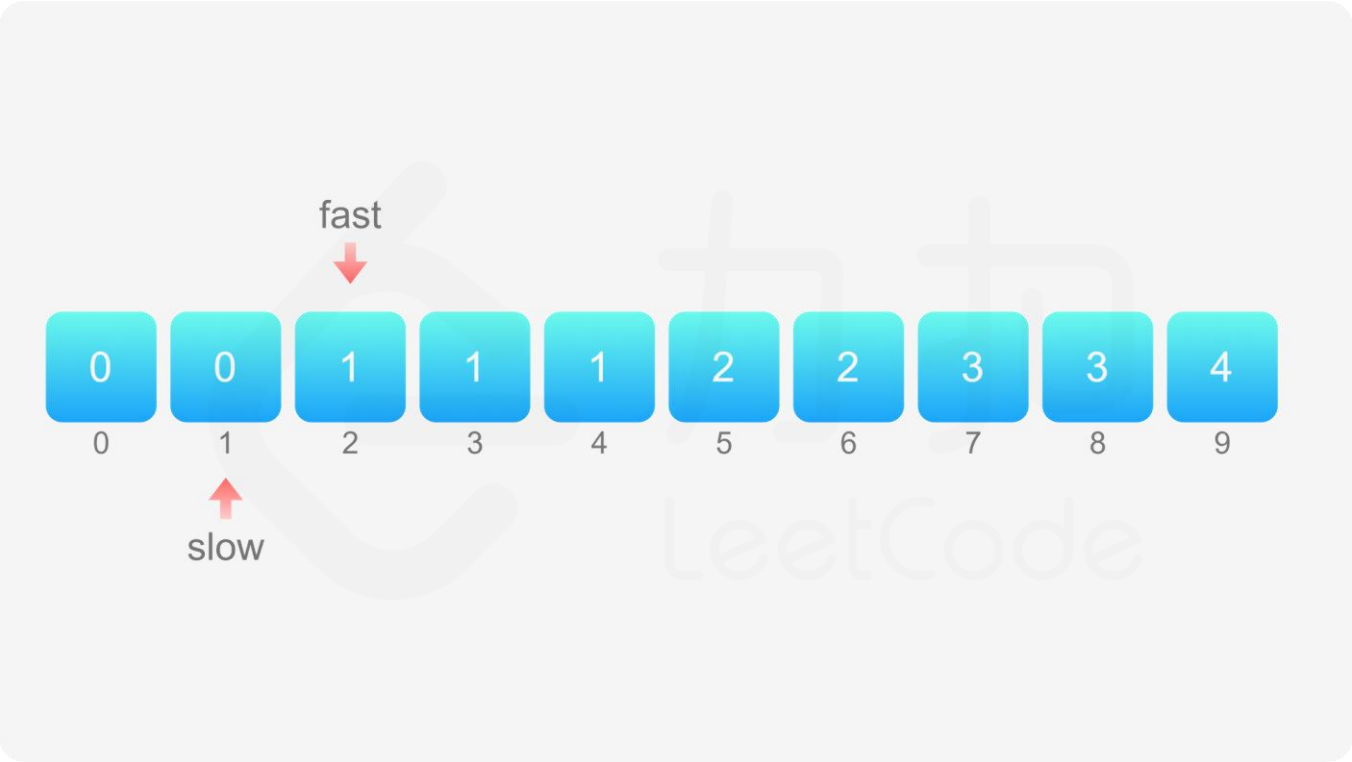
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



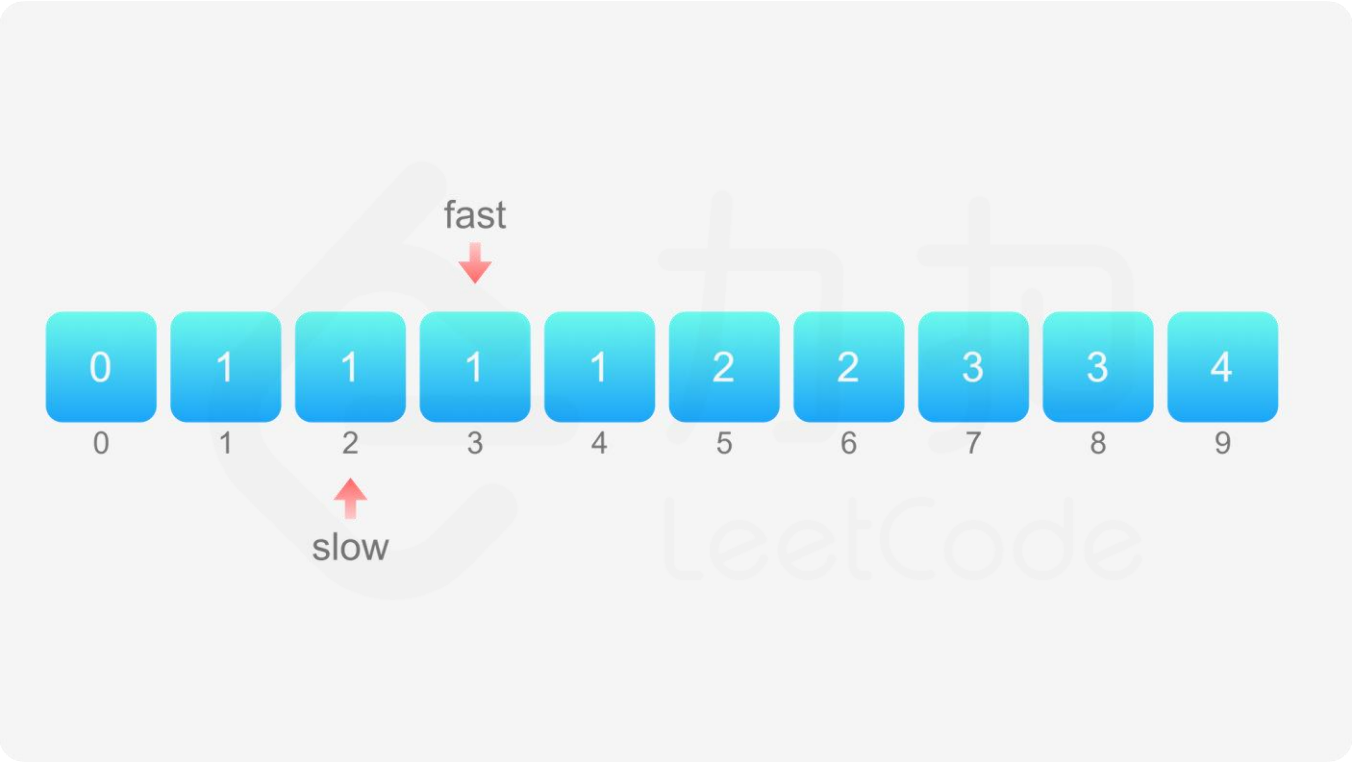
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



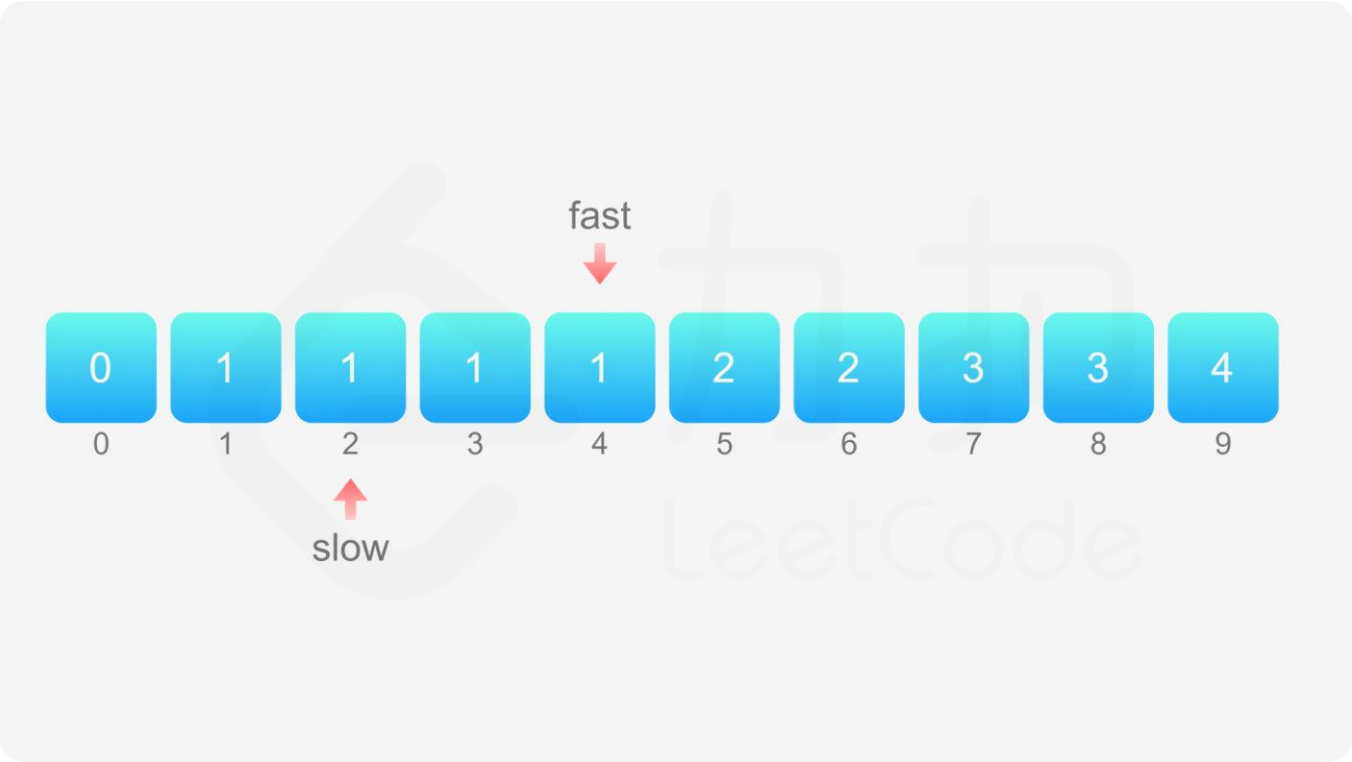
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



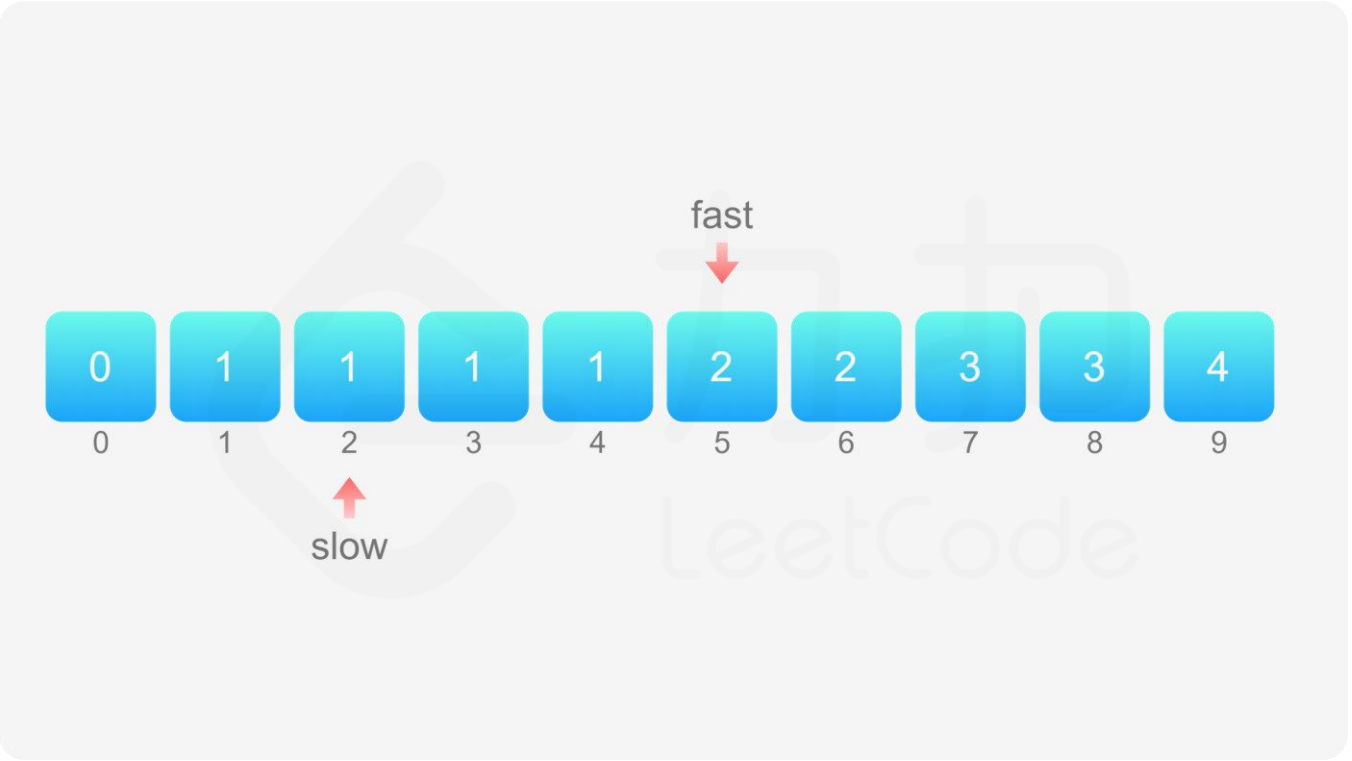
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



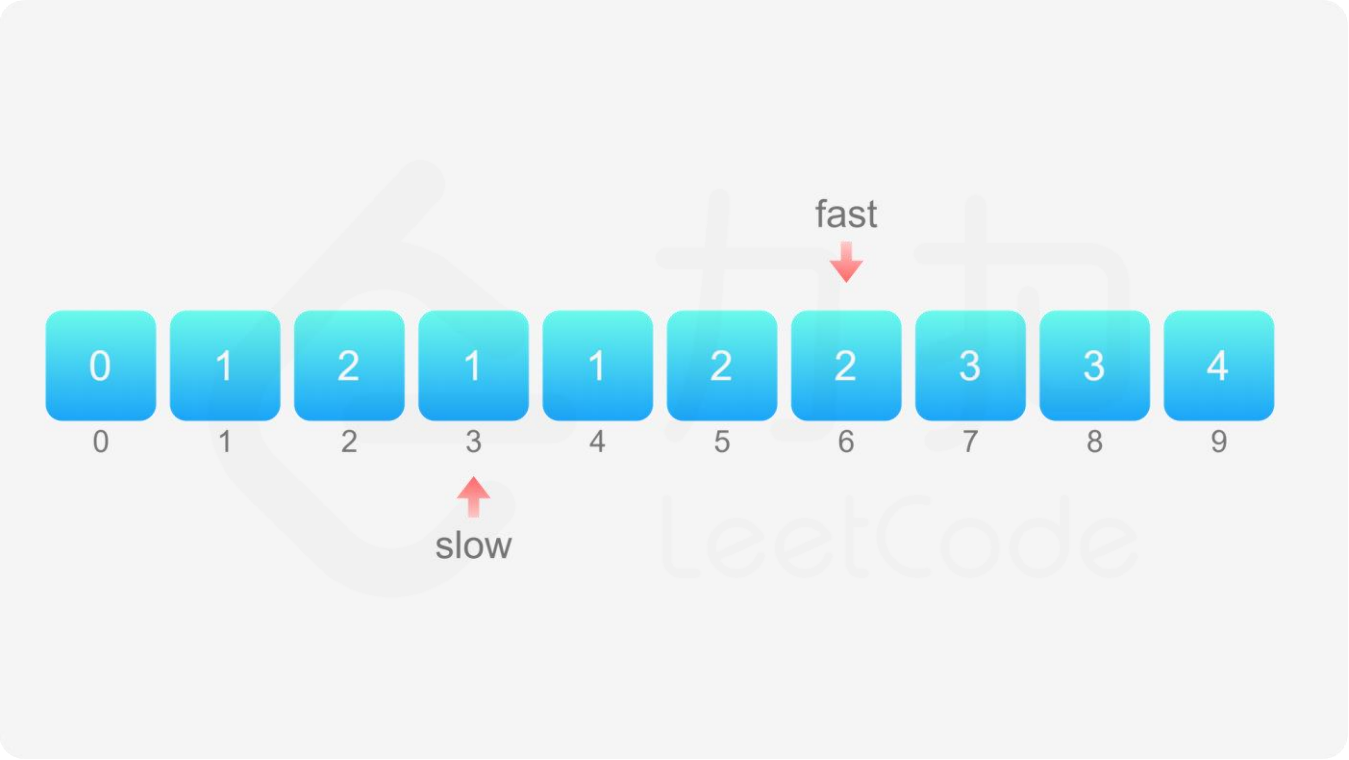
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



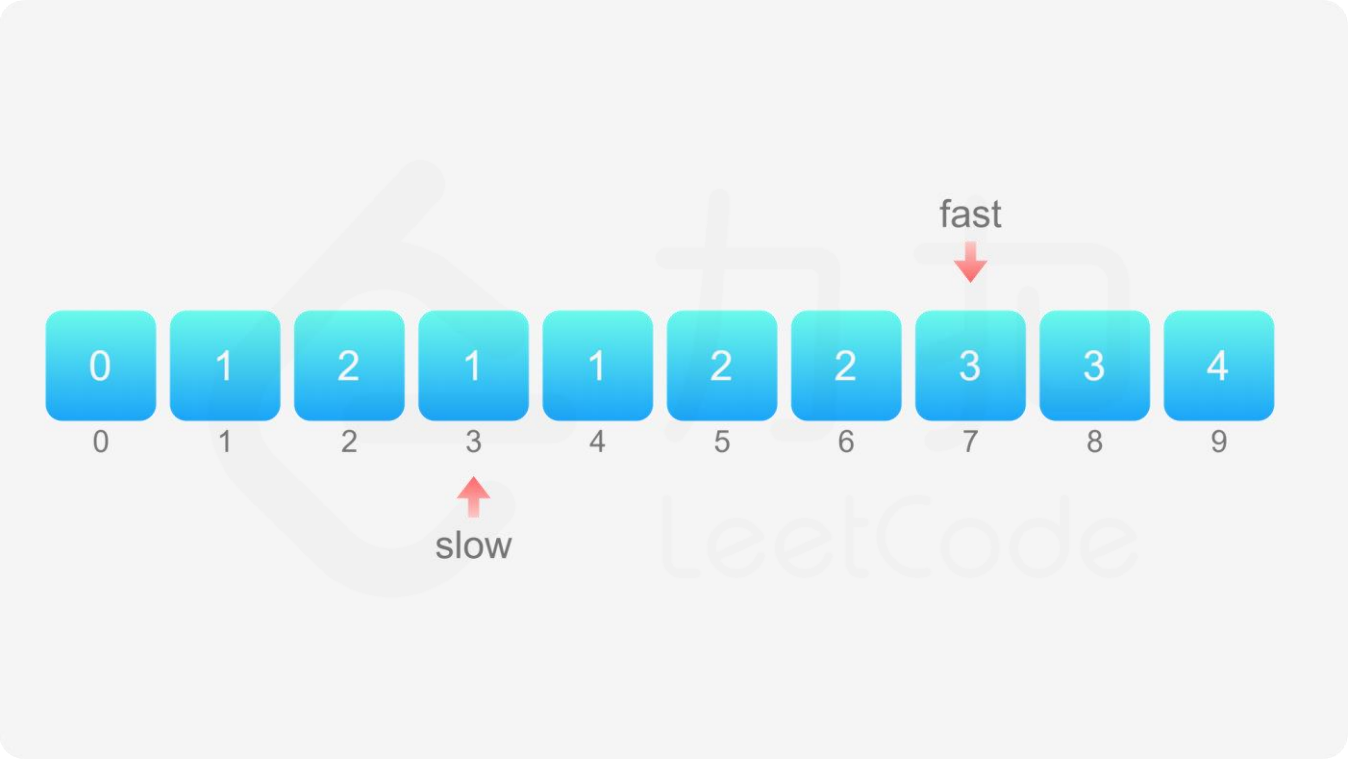
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



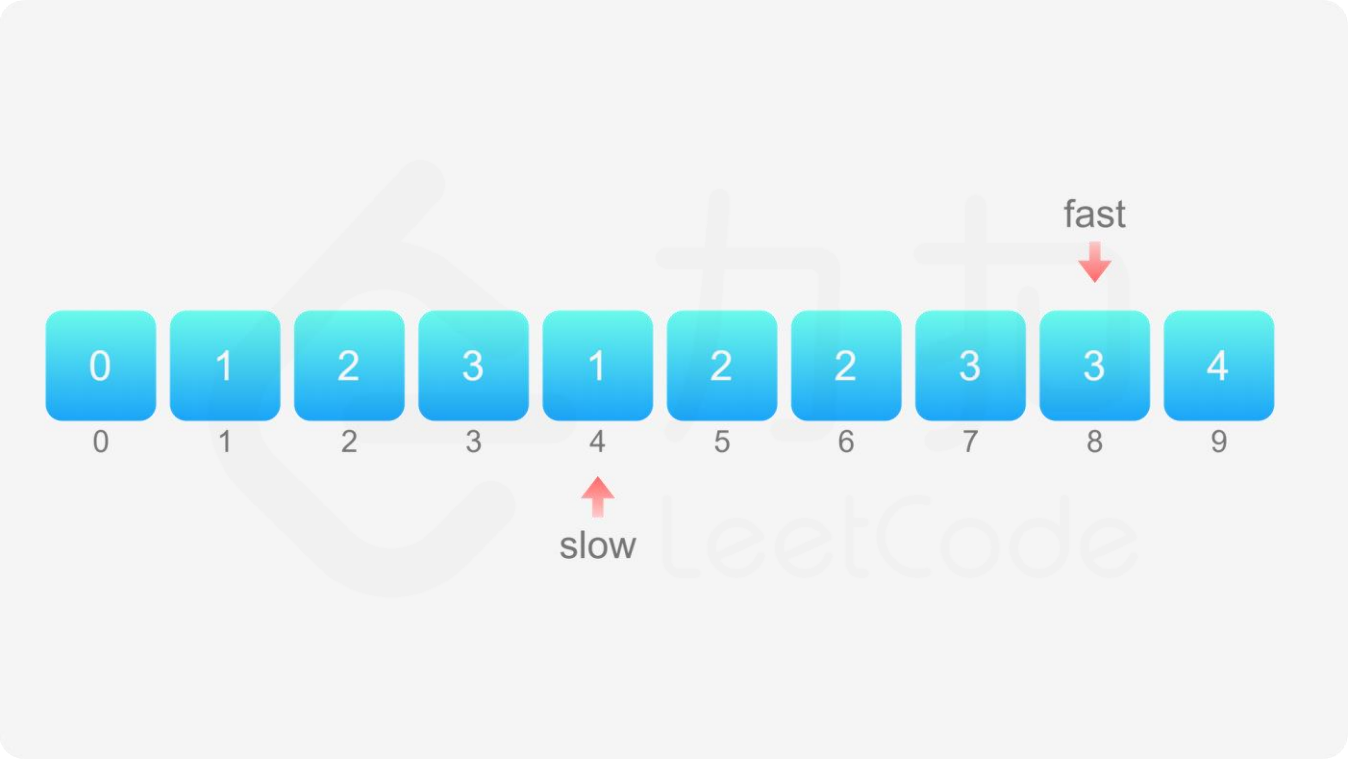
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



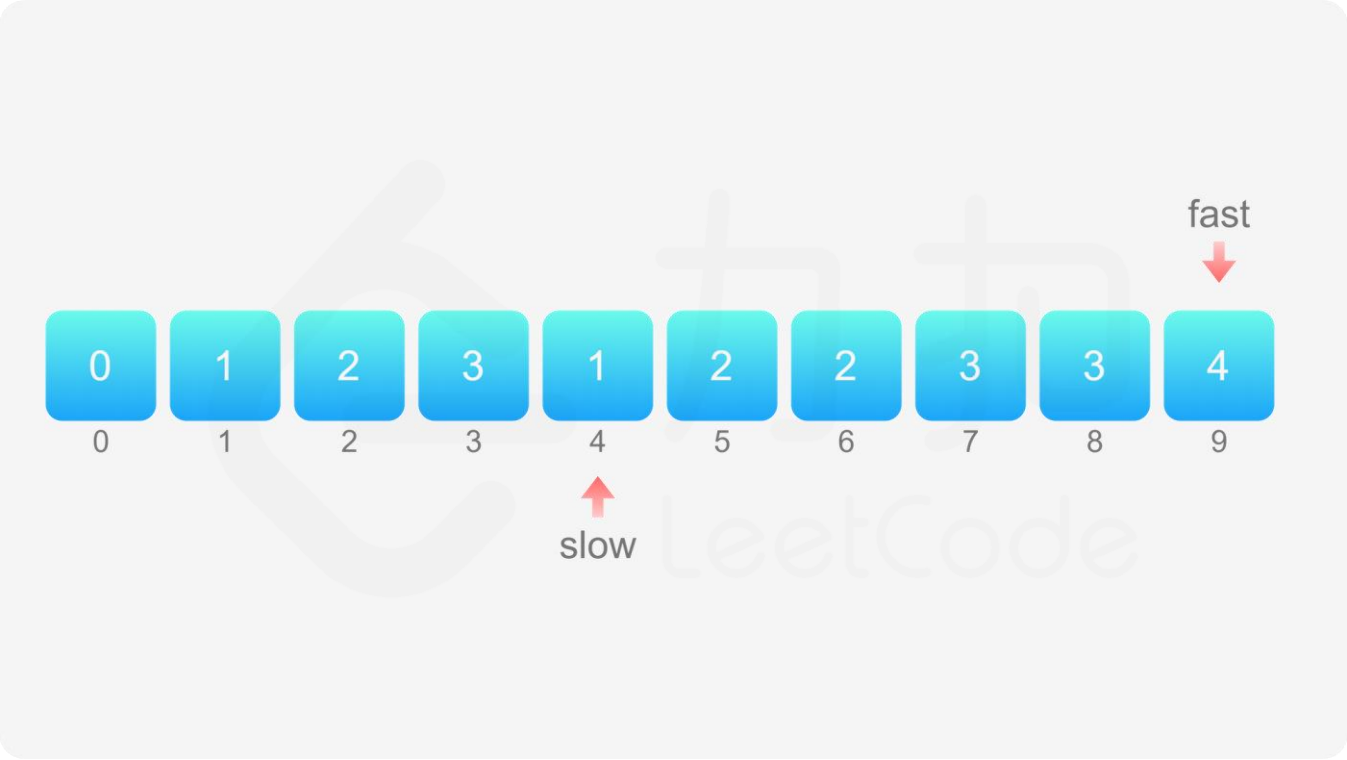
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



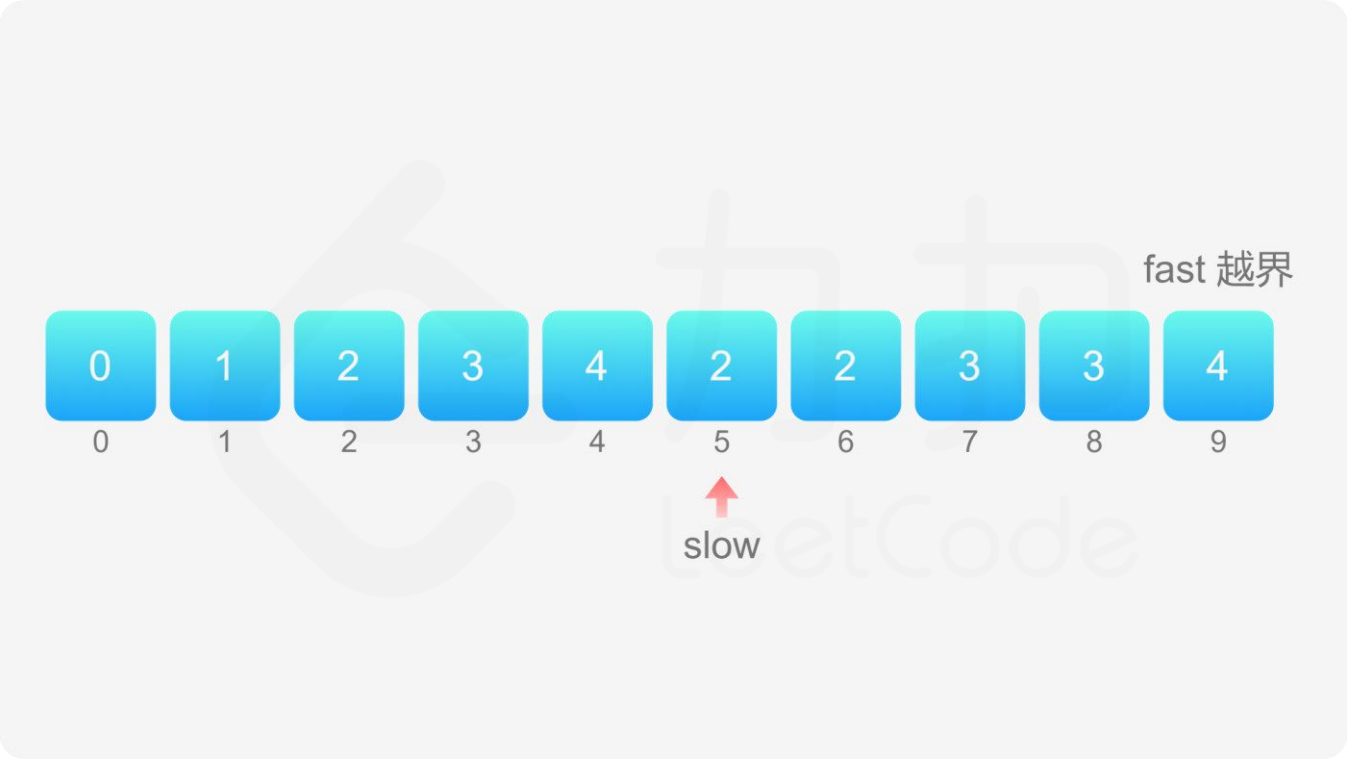
课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。



课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。

课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。

```
int removeDuplicates(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) {
        return 0;}
    int fast = 1, slow = 1;
    while (fast < n) {
        if (nums[fast] != nums[fast - 1]) {
            nums[slow] = nums[fast];
            ++slow;
        }
        ++fast;
    }
    return slow;
}
```

课堂练习 5

给你一个 **非严格递增排列** 的数组 nums ， 请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次** ， 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 nums 中唯一元素的个数。

时间复杂度：O(n)，其中 n 是数组的长度。快指针和慢指针最多各移动 n 次。

课堂练习 6

给你一个 **有序** 数组 nums ， 请你 **原地** 删除重复出现的元素，使得出现次数超过两次的元素 **只出现两次** ， 返回删除后数组的新长度。

解法 1：双指针法,遍历数组检查每一个元素是否应该被保留，如果应该被保留，就将其移动到指定位置。
快指针：表示已经检查过的数组的长度
慢指针：表示处理出的数组的长度

因为本题要求相同元素最多出现两次而非一次，所以我们需要检查上上个应该被保留的元素 nums[slow−2] 是否和当前待检查元素 nums[fast] 相同。当且仅当它们相同时当前待检查元素不应该被保留（为什么？）。

最后，slow 即为处理好的数组的长度。

课堂练习 6

给你一个 **有序** 数组 nums ， 请你 **原地** 删除重复出现的元素，使得出现次数超过两次的元素 **只出现两次** ， 返回删除后数组的新长度。

```
int removeDuplicates(vector<int>& nums) {
        int n = nums.size();
        if (n <= 2) {
            return n;
        }
        int slow = 2, fast = 2;
        while (fast < n) {
            if (nums[slow - 2] != nums[fast]) {
                nums[slow] = nums[fast];
                ++slow;
            }
            ++fast;
        }
        return slow;
    }
```

时间复杂度：O(n)，其中 n 是数组的长度。
同上题一样，快指针和慢指针最多各移动 n 次。

顺序表——查找操作

- 直接利用下标[ ]
- Q1：对于第i个元素，如何查找其直接前驱和直接后继？
- Q2：查找指定的第i个元素。

顺序表

- 顺序表结构
  - 优点：随机存取。
  - 弱点：

    1）空间利用率不高（预先按最大空间分配）

    2）表的容量不可扩充（针对顺序表的静态定义方案）

    3）即使表的容量可扩充（针对顺序表的动态定义方案），由于其空间再分配和复制的开销，因而也不允许它频繁地使用

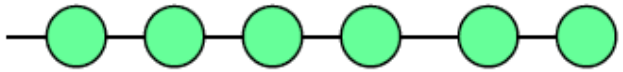
    4）插入或删除时需移动大量元素。

顺序表——适用环境

- 例1：顺序文件的查找。（有序顺序表+二分查找）
- 例2：顺序文件的查找，并在文件中添加一个元素。（基于二叉平衡树的查找）
- 结论：顺序表适用于输入数据的大小已知，且无太多动态操作的应用问题。

3.4 链表的定义及实现

- 链表是链式存储的线性结构。



  - 由一连串结点组成，结点之间通过链串起来。
- 结点：
  - 是链表元素的存储映像。
  - 每个结点包括数据域和指针域。
- 指针/链：它用于将结点们联系起来。也可以用于唯一定位一个结点。
- 头指针：链表存取的开始。
- 单链表：只有一个指针域的链表。
- 双链表：有两个指针域的链表。
- Q：对于单链表中指针P所指向的结点，它的直接前驱为？直接后继为？对于双链表呢，情况又如何？

结论：链表中用链来表明线性特征。

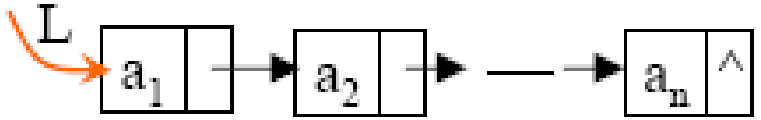
3.4 单链表——定义


```
struct Node{
    int data;
    struct Node *next};
};
Typedef struct Node *Link;
Link head;
```

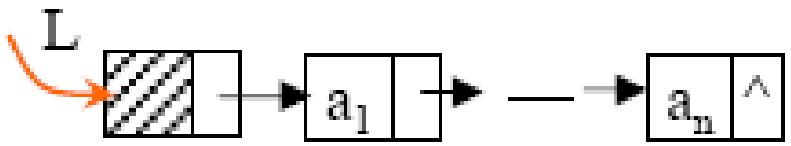
1. 只需要定义结点，就可以实现“单链表”数据存储结构的定义；
2. 结点之间的关系隐式地用指针next来描述。

3.4 单链表——定义

- 无头结点的单链表



  - 头指针为L，则空表时，L == NULL
  - 由于第一个结点无前驱结点，所以只能通过某指针变量来指向，如L；其余结点均有前驱结点，故可通过其直接前驱结点的next域来指向，即……->next；
  - 表示方法的不同，会造成对结点的操作处理的不同。
- 有头结点的单链表
  - 空表时，L指向一结点(称为头结点)，该结点的数据域可以不存储信息，也可存储如表长等的附加信息，结点的指针域存放NULL，即L->next == NULL。
  - 第一个结点和其余结点均可统一表示为其直接前驱结点的next域所指向的结点，即……->next。

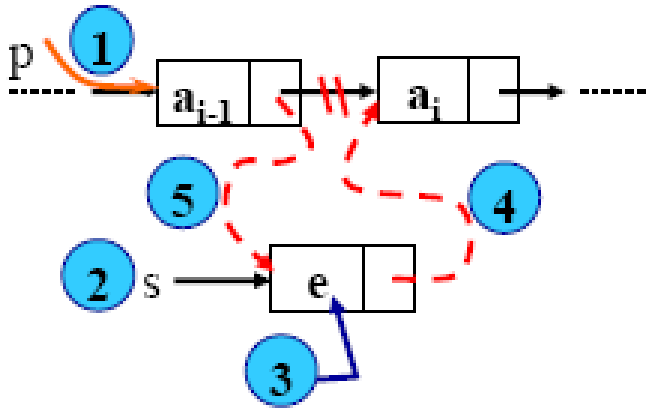


3.4 单链表——插入操作

- **ListInsert_L(LinkList &L, int i, ElemType e)**

【设计思路】

- **相关结点：**ai-1和ai
- **结点的表示：**引入指针变量LinkList p;
  ∵ 链表结点的指针域是指向该结点的直接后继
  ∴ 当p指向ai-1时，ai可用*(p->next)表示
- **关键步骤：**
  ①找到ai-1 的位置，即使p指向ai-1 结点，可参考GetElem_L( )的处理
  若p≠NULL，则② s = (LinkList)malloc(sizeof(LNode))
  ③ s->data = e
  ④ s->next = p->next
  ⑤ p->next = s
- **注意：④和⑤不能交换，否则会导致ai的位置无法获取。**



- **频度最高的操作：确定ai-1 的位置**
- **若线性表长度为n，则：**
  - ***最好情况*: T(n)=O(1)**
  - ***最坏情况*: T(n)=O(n)**
  - ***平均情况*: T(n)=O(n)**

3.4 单链表——插入操作

- 有头结点的单链表中的插入算法

```
Status ListInsert(LinkList &L, int i, ElemType e){
    // 有头结点，无须对i为1的插入位置作特殊处理
    p = L; j = 0;                    // 对p,j初始化; *p为L的第j个结点
    while( p != NULL && j<i-1){
        p = p->next;
        j++;                         // 寻找第i-1个结点的位置
    }
    if( p == NULL || j>i-1) return ERROR;    // i小于1或大于表长
    s = (LinkList )malloc(sizeof(LNode));    // 生成新结点
    if ( s == NULL )
        exit(OVERFLOW);                      // 空间分配不成功，报错返回
    s->data = e; s->next = p->next;          // 插入L中
    p->next = s;
    return OK;
}
```

3.4 单链表——插入操作

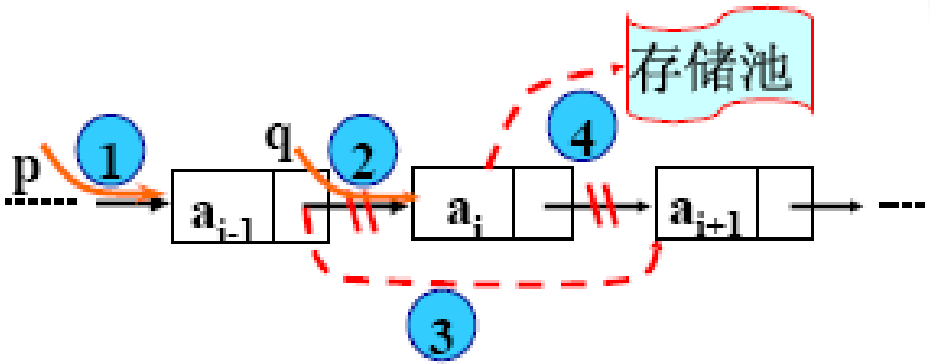
- 无头结点的单链表中的插入算法

```
Status ListInsert(LinkList &L, int i, ElemType e) {
    // 无头结点，须对i为1的插入位置作特殊处理
    if ( i==1){
        s = (LinkList )malloc(sizeof(LNode));    // 生成新结点
        if ( s == NULL )    exit(OVERFLOW);      // 空间分配不成功，报错返回
        s->data = e; s->next = L;                // 插入到链表L中
        L = s;                                   // 修改链头指针L
    }else{
        p = L; j = 1;                            // 对p,j初始化；*p为链表的第j个结点
        while( p != NULL && j<i-1){
            p = p->next;
            j++;                                 // 寻找第i-1个结点的位置
        }
        if( p == NULL || j>i-1) return ERROR;    // i小于1或大于表长
        s = (LinkList )malloc(sizeof(LNode));    // 生成新结点*s
        if ( s == NULL )    exit(OVERFLOW);      // 空间分配不成功，报错返回
        s->data = e; s->next = p->next;          // 插入到链表L中
        p->next = s;
    }
    return OK;
}
```

【思考】对比无头结点和有头结点在插入算法上的不同，分析其中的原因。

3.4 单链表——删除操作

相关结点：ai-1、ai和ai+1
结点表示：*p, *(p->next),*(p->next->next)



- 设计思路：
  - 关键步骤：

①找到ai-1 的位置，即使p指向ai-1 结点, 可参考GetElem_L( )的处理

p->next≠NULL（有待删除的结点），则

② q = p->next (记录待释放结点的位置)

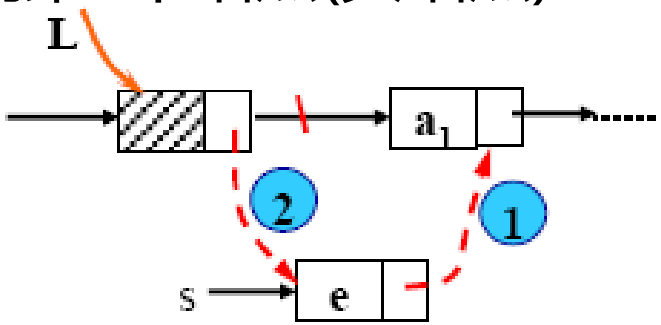
③ p->next = p->next->next

④ free(q)

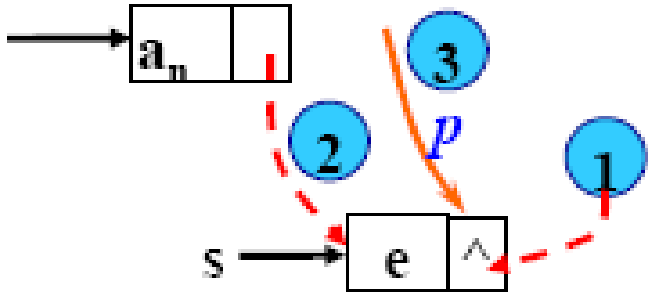
注意：必须在③④前增加②步，否则在执行了③后要释放的结点无法标识。

单链表——创建操作

- 头插法
  - 思想：每次将待插结点*s插入到第一个结点之前；当有头结点时，待插结点也可视为插入到第0个结点(头结点)之后。



- 尾插法：
  - 思想：待插结点*s插入到最后一个结点之后



单链表——头插法创建操作

- **思想：**每次将待插结点*s插入到第一个结点之前；当有头结点时，待插结点也可视为插入到第0个结点(头结点)之后。
- **插入步骤：**以单链表中有头结点为例，单个结点的构造和插入步骤如下

  ① s = (LinkList)malloc(sizeof(LNode))

  ② scanf(&s->data)

  ③ s->next = L->next ④ L->next = s
- **算法时间复杂度分析：**每次插入一个结点所需的时间为O(1)

  ∴头插法创建单链表的时间复杂度 **T(n) = O(n)**

单链表——尾插法创建操作

- **思想：**待插结点*s插入到最后一个结点之后
- **插入步骤：**

  ① 获得最后一个结点的位置,使p指向该结点

  ② p->next = (LinkList)malloc(sizeof(LNode))

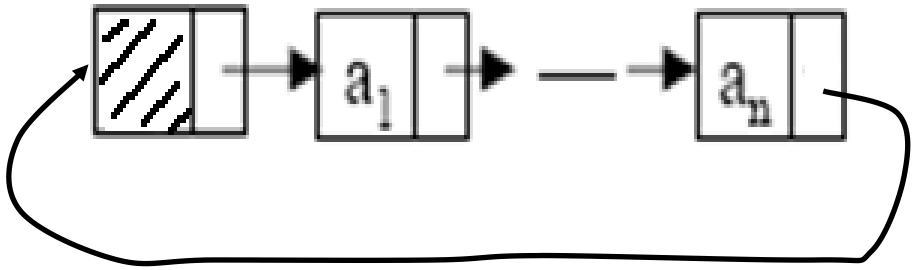
  ③ p = p->next

  ④ scanf(&p->data)

  ⑤ p->next = NULL
- **算法时间复杂度分析：**要想获取最后一个结点的位置，必须从链头指针开始顺着next链搜索链表的全部结点，该过程的时间复杂度是 O(n)。如果每次插入都按此方法获取最后一个结点的位置，则整个创建算法的时间复杂度为$T(n) = O(n^2)$。

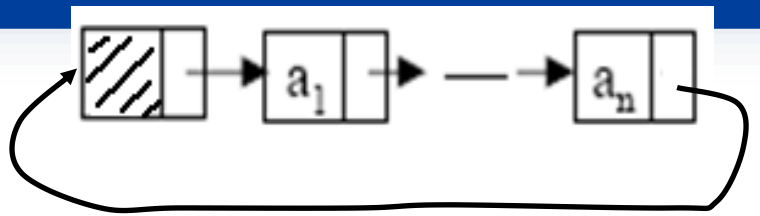
循环链表

- 循环链表
- 循环链表 VS. 单链表：
  1) 定义相同；
  2) 最后一个结点不同：
     - 循环量表中最后一个结点的指针不为空，而是指向表头结点（头结点/第一个结点）；
  3) 判断表尾结点的条件不同：
     - 循环链表中，判断当前是否到达表尾结点的条件，不再是 看其是否为NULL，而是看其**是否等于头指针**

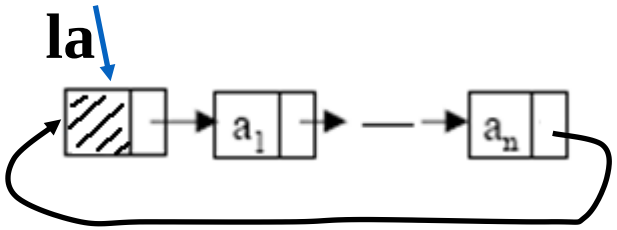


- 问题1：如何从一个结点出发，访问到链表中的全部结点?

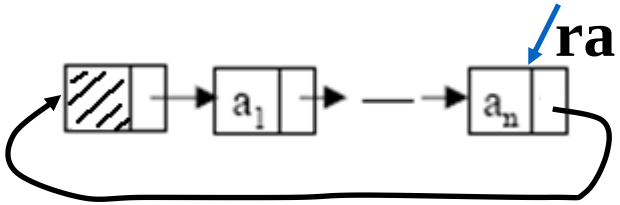
循环链表



- **问题2：对于循环链表，**如何在O(1)时间内由链表指针访问到第一个结点和最后一个结点?
  - **头指针表示法：**
    - **第一个结点：*(la->next)　　T(n)=O(1)**
    - **最后一个结点：需从表头搜索到表尾　T(n)=O(n)**



  - **尾指针表示法：**
    - **第一个结点：*(ra->next->next)　　T(n)=O(1)**
    - **最后一个结点：*ra　　T(n)=O(1)**

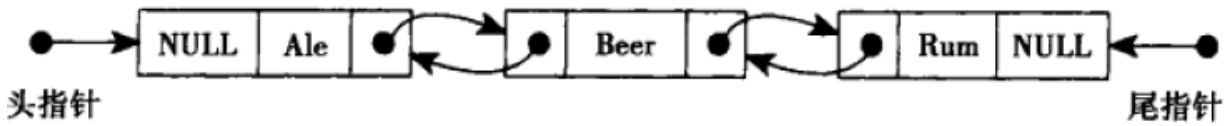


双向链表

- 单链表的缺点：只知其直接后继结点，不知当前结点的直接前驱。
- 如何O(1)时间内找到当前结点的直接前驱？
- 巧妙的解决方案：
  - Link prev; curr=prev->next;

- 本质上克服单链表的弱点——双向链表

- 双向链表：



  - 每个结点有2个链：分别指向逻辑相邻的2个结点。
  - 可在O(1)时间内找到一个结点的直接前驱结点和直接后继结点
  - 广泛应用，如STL:list

双向链表


```
Struct Node{
    ElemType    data;
    struct Node     *prior;
    struct Node     *next;
};
typedef struct Node * Link;
typedef struct {
    Link    head, tail;
    int     len;
}DLinkList;
```

- 特点：若d指向表中的一个内部结点，则d->next->prior == d->prior->next == d
- 双向循环链表：存在两个环。

异或链表

- 双向链表节点内需要两个指针
- 如何**仅使用一个指针的内存大小**实现双向链表的功能?
- 巧妙的解决方案:
  - Link ptr = prev ^ next
- 正向遍历时，前节点的地址和当前节点的 ptr 异或得到后节点的地址
- 反向遍历时，后节点的地址和当前节点的 ptr 异或得到前节点的地址
- 多一次计算

异或链表

- 异或操作：相同为0，不同为1
- 异或操作的性质

交换律： 对于任意两个值，异或操作满足交换律，即 a XOR b = b XOR a。

结合律： 异或操作也满足结合律，即 (a XOR b) XOR c = a XOR (b XOR c)。

自反性： 对于任意值 a，a XOR a = 0，这是因为异或一个值两次会回到原始值。

零值性质： 对于任意值 a，a XOR 0 = a，因为异或零不会改变原始值

A XOR B = C，B XOR C = ?

异或链表

1

```
class XorNode {
public:
    ElemType data;
    XorNode<ElemType> *xorPtr;
    XorNode(ElemType data):data(data) { }
};
```

2

```
    while (i < pos && curNode != NULL) {
        tmpNode = curNode; // current
        curNode = xor_func(prevNode, curNode->xorPtr);
        prevNode = tmpNode; // prev
        i++;
    }
```

3

```
    // insert the newNode before pos (current)
    XorNode<ElemType> *newNode = new XorNode<ElemType>(e);
    newNode->xorPtr    = xor_func( prevNode,curNode);
    prevNode->xorPtr   = xor_func(xor_func(prevNode->xorPtr, curNode), newNode);
    curNode->xorPtr    = xor_func(newNode, xor_func(prevNode, curNode->xorPtr));
    size++;
```

顺序表和链表的特性对比

	插入/删除（时间复杂度）	查询（时间复杂度）	适用场景
数组	O(n)	O(1)	数据量固定，频繁查询，较少增删
链表	O(1)	O(n)	数据量不固定，频繁增删，较少查询

链表例题-1

描述：给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

例如，[1, 2, 3, 4, 5], n = 2
输出　[1, 2, 3, 5]

尝试使用一趟扫描实现?

链表例题-1

描述：给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

双指针法：a、b指针

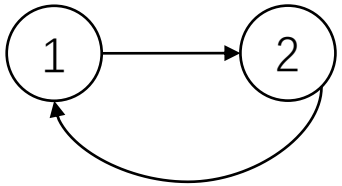
1. **a指针先移动 n 步**，
2. a指针和b指针同时移动，
3. a指针指向链表末尾时，**删除b指针指向的节点**。

```
ListNode* dummyHead = new ListNode(0);
dummyHead->next = head; // 使用头节点方便删除
ListNode* slow = dummyHead;
ListNode* fast = dummyHead;
while(n-- && fast != NULL) {
    fast = fast->next;}
// fast再提前走一步，让slow指向删除节点的上一个节点
fast = fast->next;
while (fast != NULL) {
    fast = fast->next;
    slow = slow->next;}

ListNode *tmp = slow->next;
slow->next = tmp->next;
delete tmp;
return dummyHead->next;
```

链表例题-2 （双指针）

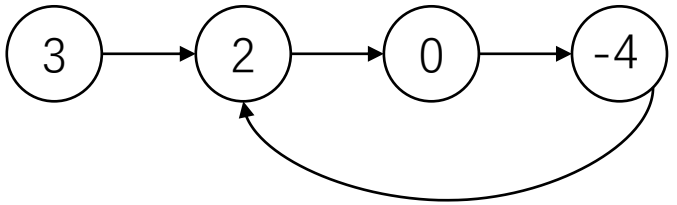
问题描述：给你一个链表的头节点 head ，判断链表中是否有环。如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环

示例 1：



输入： head = [1, 2], pos = 0
输出： true

示例 2：



输入： head = [3, 2, 0, -4], pos = 1
输出： true

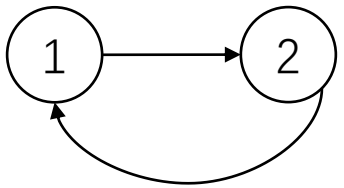
链表例题-2 （双指针）


```
bool hasCycle(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return false;
        }
        ListNode* slow = head;
        ListNode* fast = head->next;
        while (slow != fast) {
            if (fast == nullptr || fast->next == nullptr) {
                return false;
            }
            slow = slow->next;
            fast = fast->next->next;
        }
        return true;
    }
```

链表例题-3（双指针）

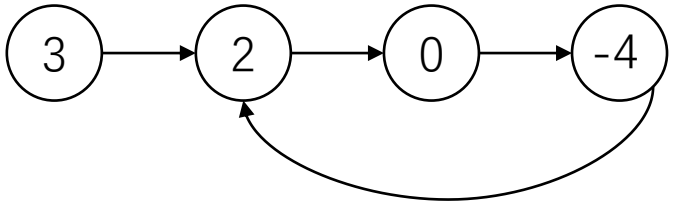
问题描述：给定一个链表，返回链表**开始入环的第一个节点**。 从链表的头节点开始沿着 next 指针进入环的第一个节点为环的入口节点。如果链表无环，则返回 null

示例 1：



输入： head = [1, 2], pos = 0
输出： 返回索引为 0 的链表节点
解释： 链表中有一个环，其尾部连接到第一个节点。

示例 2：



输入： head = [3, 2, 0, -4], pos = 1
输出： 返回索引为 1 的链表节点
解释： 链表中有一个环，其尾部连接到第二个节点。

链表例题-3 （双指针）

问题描述：给定一个链表，返回链表**开始入环的第一个节点**。 从链表的头节点开始沿着 next 指针进入环的第一个节点为环的入口节点。如果链表无环，则返回 null

算法流程:

双指针第一次相遇：设两指针 fast，slow 指向链表头部 head，fast 每轮走 2 步，slow 每轮走 1 步;

1. **第一种结果：**fast 指针走过链表末端，说明链表无环，直接返回 null；
    - TIPS：若有环，两指针一定会相遇。因为每走 1 轮，fast 与 slow 的间距 +1，fast 终会追上 slow;

链表例题-3 （双指针）

问题描述：给定一个链表，返回链表开始入环的第一个节点。 从链表的头节点开始沿着 next 指针进入环的第一个节点为环的入口节点。如果链表无环，则返回 null

算法流程:

2. 第二种结果：当 fast == slow 时，两指针在环中**第一次相遇**。下面分析此时 fast 与 slow 走过的**步数关系**:

- 设链表共有 a+b 个节点，其中**链表头部到链表入口**有 a 个节点（不计链表入口节点），**链表环**有 b 个节点（这里需要注意，a 和 b 是未知数，例如图解上链表 a = 4，b = 5）；设两指针分别走了 f，s步,则有:
  - fast 走的步数是 slow 步数的 2 倍，即 f = 2s；(解析：fast 每轮走 2 步)
  - fast 比 slow 多走了 n 个环的长度，即 f = s + nb；(解析：双指针都走过 a 步，然后在环内绕圈直到重合，重合时 fast 比 slow 多走**环的长度整数倍**）；
- 以上两式相减得：f = 2nb，s = nb，即 fast 和 slow 指针分别走了 2n，n 个**环的周长**（注意：n是未知数，不同链表的情况不同）。

链表例题-3（双指针）

问题描述：给定一个链表，返回链表开始入环的第一个节点。 从链表的头节点开始沿着 next 指针进入环的第一个节点为环的入口节点。如果链表无环，则返回 null

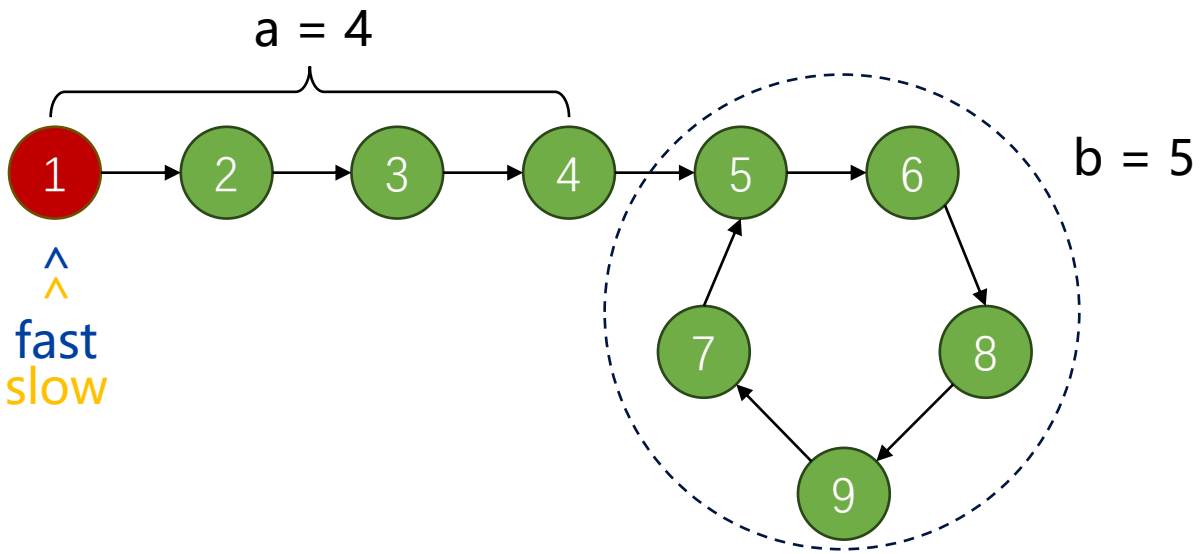
目前情况分析:

- 如果让指针从链表头部一直向前走并统计步数 k，那么所有走到链表入口节点时的步数是：k = a + nb（先走 a 步到入口节点，之后每绕 1 圈环（b步）都会再次到入口节点)。
- 而目前，slow 指针走过的步数为 nb 步。因此，我们只要想想办法让 slow 再走 a 步停下来，就可以到环的入口。
- 但是我们不知道 a 的值，该怎么办？依然是使用双指针法。我们沟建一个指针，此指针需要有以下性质：此指针和 slow 一起向前走 a 步后，两者在入口节点重合。那么从哪里走到入口节点需要 a 步？答案是链表头部 head。

链表例题-3（双指针）

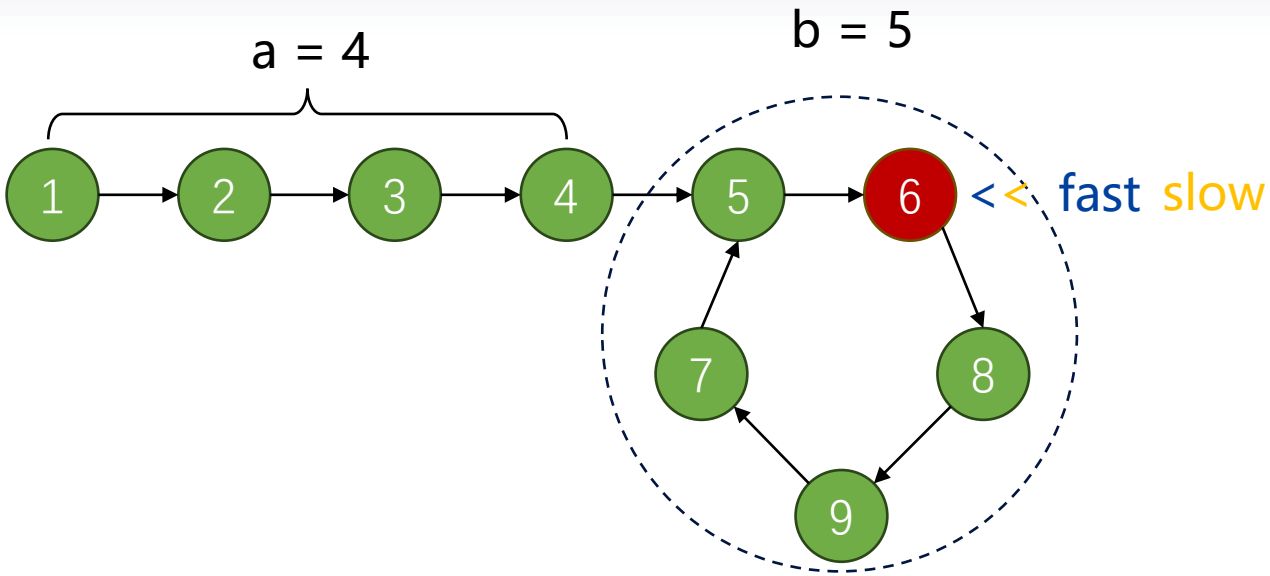
双指针第二次相遇：

- slow 指针**位置不变**，将 fast 指针重新**指向链表头部**节点；slow 和 fast 同时每轮向前走 1 步；
  - TIPS：此时 f = 0, s = nb;
- 当 fast 指针走到 f = a 步时，slow 指针走到 s = a + nb，此时**两指针重合，并同时指向链表环入口。**
- 第二次相遇后，返回 slow 指针指向的节点即可。



链表例题-3（双指针）

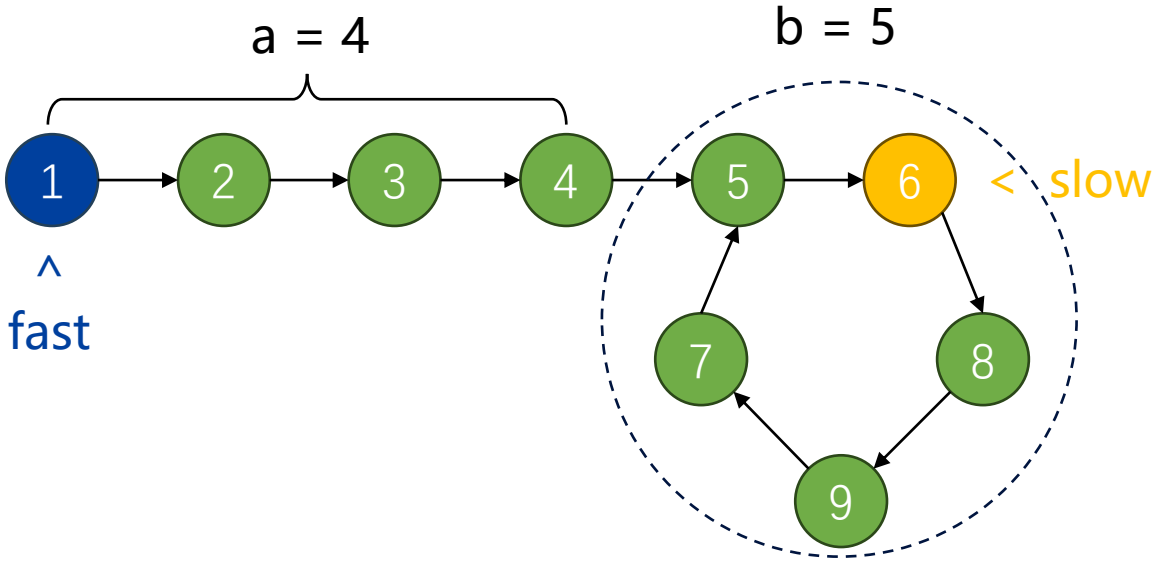
第一轮相遇



fast移动至head

步幅为1

等待第二次相遇



链表例题-3（双指针）

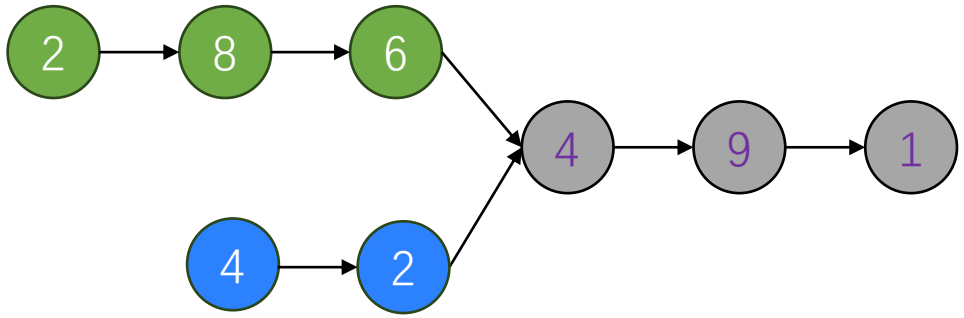

```
ListNode *detectCycle(ListNode *head) {
        ListNode *fast = head, *slow = head;
        while (true) {
            if (fast == nullptr || fast->next == nullptr) return nullptr;
            fast = fast->next->next;
            slow = slow->next;
            if (fast == slow) break;
        }
        fast = head;
        while (slow != fast) {
            slow = slow->next;
            fast = fast->next;
        }
        return fast;
    }
```

链表例题-4（双指针）

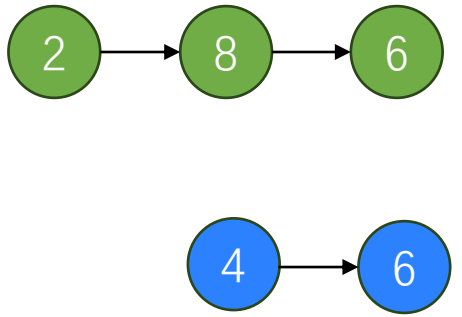
问题描述：给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null

题目数据 保证 整个链式结构中不存在环，链表必须 保持其**原始结构**

示例 1：



示例 2：



链表例题-4（双指针）

问题描述：给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null

题目数据 保证 整个链式结构中不存在环，链表必须 保持其**原始结构**

暴力法

O(mn)

```
ListNode *getIntersectionNode(ListNode *headA,
ListNode *headB) {
        ListNode *tmp1 = headA;
        while(tmp1!=nullptr){
        ListNode *tmp2 = headB;
        while(tmp2!=nullptr){
            if (tmp2==tmp1) {return tmp2;}
            tmp2 = tmp2->next;
        }
        tmp1 = tmp1->next;
        }
        return nullptr;}
```

链表例题-4（双指针）

问题描述：给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null

题目数据 保证 整个链式结构中不存在环，链表必须 保持其**原始结构**

哈希法

1. 遍历链表 headA，并将链表 headA 中的每个节点加入哈希集合中。
2. 遍历链表 headB，对于遍历到的每个节点，判断该节点是否在哈希集合中.
   - 如果当前节点不在哈希集合中，则继续遍历下一个节点;
   - 如果当前节点在哈希集合中，则后面的节点都在哈希集合中，即从当前节点开始的所有节点都在两个链表的相交部分

O(m+n)　　**额外空间为O(m)**

```
unordered_set<ListNode *> uoset;
    ListNode *tmp = headA;
    while (tmp != nullptr) {
        uoset.insert(tmp);
        tmp = tmp->next;}
    tmp = headB;
    while (tmp != nullptr) {
        if (uoset.count(tmp)) {
            return tmp;}
        temp = temp->next;}
    return nullptr;
```

链表例题-4（双指针）

问题描述：给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null

题目数据 保证 整个链式结构中不存在环，链表必须 保持其**原始结构**

双指针法

1. **初始**：指针 pA 和 pB，分别指向头节点 headA 和 headB，依次遍历。
2. 每步操作需要同时更新指针 pA 和 pB（1次1步）
3. pA 和 pB指针移动到nullptr时，分别移动到对方的head开始，继续后移动
4. **终止条件**：指向相同节点返回，或者都为空时nullptr

O(M+N)　　**额外空间为O(1)**

含义

c为共同的节点个数
链表A: m=a+c
链表B: n = b+c

走到第一次相交节点时的步数
A: 走a步　B: 走b步

构造相遇时的步数恰好为第一个相交节点
sA：a+ c +b　相当于B走了b步
sB:　b+ c +a　相当于A走了a步
A=B

链表例题-4（双指针）

问题描述：给你两个单链表的头节点 headA 和 headB ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null

题目数据 保证 整个链式结构中不存在环，链表必须 保持其**原始结构**

双指针法

```
ListNode *getIntersectionNode(ListNode *headA, ListNode
*headB) {
    if (headA == nullptr || headB == nullptr) {
            return nullptr;}

        ListNode *pA = headA, *pB = headB;
        while (pA != pB) {
            pA = pA == nullptr ? headB : pA->next;
            pB = pB == nullptr ? headA : pB->next;
        }
        return pA;
    }
```

顺序表例题-1

问题描述：给你一个整数数组 nums ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

子数组是数组中的一个连续部分。

示例 1：

输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6 。

示例 2：

输入: nums = [5,4,-1,7,8]
输出: 23
解释: 连续子数组 [5,4,-1,7,8] 的和最大，为 23

暴力求解

方法1

- 基本思路：
  - 对所有满足0≤i≤j<n的整数对(i,j)进行检查，判断x[i..j]的总和Sum(x[i..j])是否最大？
  - 其中$Sum(x[i..j])=\sum_{k=i}^{j} x[k]$

```
float alg1()
{
int i, j, k;
float sum, maxsofar = 0;
for (i = 0; i < n; i++)

   for (j = i; j < n; j++)
      {
         sum = 0;
         for (k = i; k <= j; k++)
              sum += x[k];
         if (sum > maxsofar)
             maxsofar = sum;
      }
 return maxsofar;   }
```

$T(n)=O(n^3)$

算法2

- 基本思路：
  - 对所有满足0≤i≤j<n的整数对(i,j)进行检查，判断x[i..j]的总和sum(x[i..j])是否最大？

– 如何计算**sum(x[i..j])**？
  - **sum(x[i..j]) = sum(x[i..j-1])+x[j]**
  - **sum(x[i..j]) = sum(x[0..j])-sum(x[0..i-1])**

```
float alg2()
{
int i, j, k;
float sum, maxsofar = 0;
for (i = 0; i < n; i++)
   {                                  T(n)=O(n²)
      sum = 0;
      for (j = i; j < n; j++){
         sum += x[j];
         if (sum > maxsofar)
            maxsofar = sum;  }
   }
 return maxsofar;
}
```

如何计算sum(x[i..j])?

sum(x[i..j]) = sum(x[0..j])-sum(x[0..i-1])

```
float cumvec[MAXN+1];
float alg2b()
{
int i, j, k;
float *cumarr, sum, maxsofar = 0;
cumarr = cumvec+1; // to access cumarr[-1]
cumarr[-1] = 0;

for (i = 0; i < n; i++)
        cumarr[i] = cumarr[i-1] + x[i];

for (i = 0; i < n; i++)
    {   for (j = i; j < n; j++)
        {
          sum = cumarr[j] - cumarr[i-1];
            if (sum > maxsofar)
                maxsofar = sum;
        }
    }
 return maxsofar;}
```

$T(n)=O(n^2)$

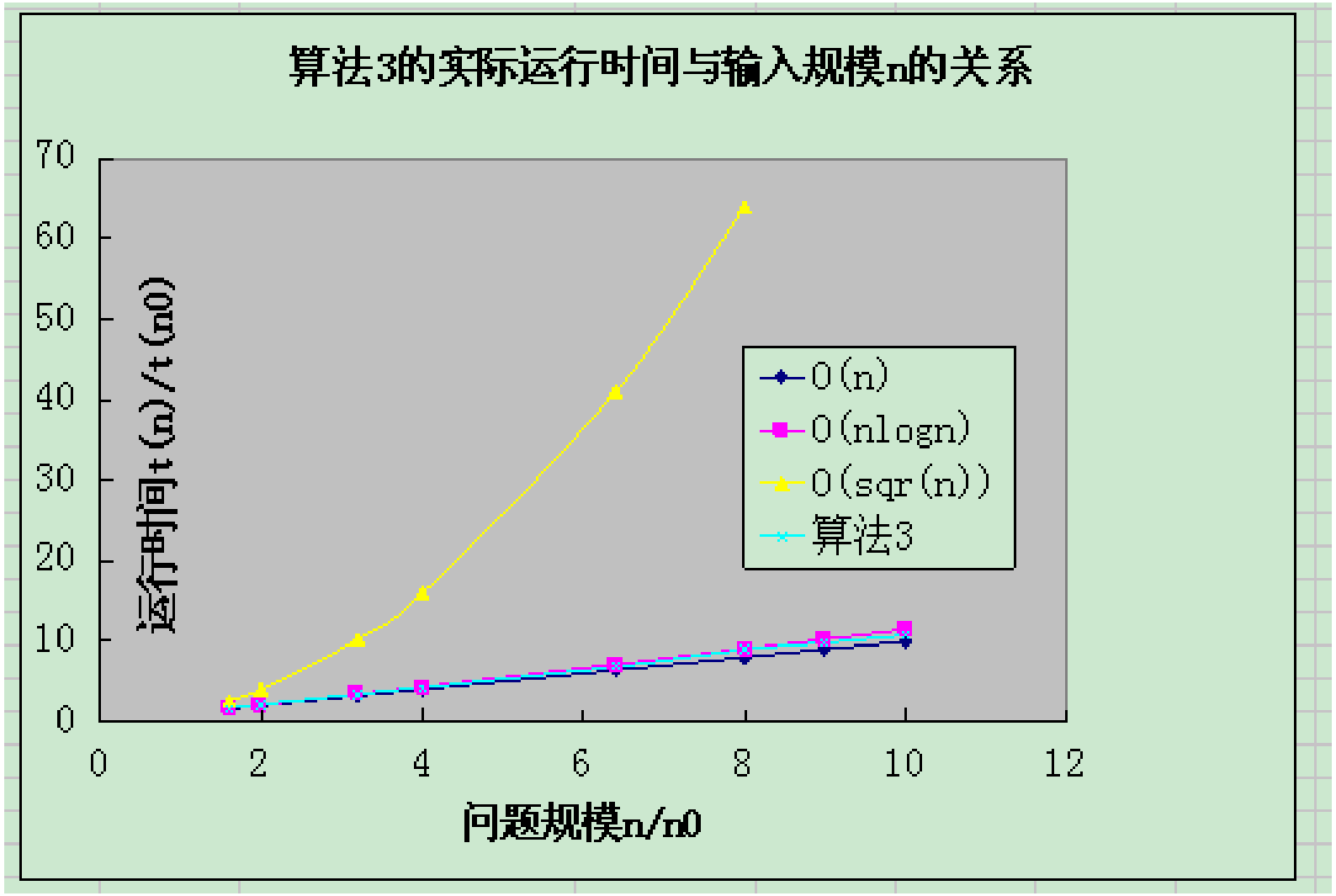
$m_x = \max(m_a, m_b, m_c)$,
其中 $m_c = \max_{0 \le i < n/2-1}(\text{sum}(x[i..n/2-1])) + \max_{n/2 < j < n}(\text{sum}(x[n/2..j]))$

```
float maxfun(float a, float b)
{   return a > b ? a : b;
}

#define max(a, b) maxfun(a, b)

float alg3()
{   return recmax(0, n-1);
}
```

$T(n)=2T(n/2)+O(n)$

=> $T(n)=O(n\log n)$

```
float recmax(int l, int u)
{   int i, m;
    float lmax, rmax, sum;
    if (l > u)  /* zero elements */
                        return 0;
    if (l == u)  /* one element */
                        return max(0, x[l]);
    m = (l+u) / 2;
// find max crossing to left
    lmax = sum = 0;
    for (i = m; i >= l; i--) {
            sum += x[i];
            if (sum > lmax)
                lmax = sum;  }
    rmax = sum = 0;
    for (i = m+1; i <= u; i++) {
            sum += x[i];
            if (sum > rmax)
                        rmax = sum;    }
    return max(lmax + rmax,max(recmax(l,
m), recmax(m+1, u)));   }
```


算法3的实际运行时间与输入规模n的关系
运行时间t(n)/t(n0)
问题规模n/n0
0
10
20
30
40
50
60
70
0
2
4
6
8
10
12
O(n)
O(nlogn)
O(sqr(n))
算法3

2-动态规划

思路：

假设 nums 数组的长度是 n，下标从 0 到 n−1。**我们用 f(i) 代表以第 i 个数结尾的「连续子数组的最大和」，那么我们要求的答案就是：max{f(i)}，(0≤i≤n−1)**。我们只需要求出每个位置的 f(i)，然后返回 f 数组中的最大值即可。

可以考虑 nums[i] 单独成为一段还是加入 f(i−1)对应的那一段，这取决于 nums[i] 和 f(i−1)+nums[i] 的大小，我们希望获得一个比较大的，于是可以写出这样的动态规划转移方程：

$$f(i)=\max\{f(i-1)+nums[i],nums[i]\}$$

```
int maxSubArray(vector<int>& nums) {
    int size = nums.size();
    vector<int> dp(size,0);
    int result = dp[0] = nums[0];//防止仅有一个元素
    for(int i=1;i<size;i++){
        dp[i] = max(dp[i - 1] + nums[i], nums[i]);
        result = max(result,dp[i]);
    }
    return result;}
```

```
int maxSubArray(vector<int>& nums) {
    int pre,size = nums.size();
    int result = pre = nums[0];//防止仅有一个元素
    for(int i=1;i<size;i++){
        pre = max(pre + nums[i], nums[i]);
        result = max(result,pre);
    }
    return result;}
```

3-分治法

思路：

我们定义一个操作 get(a, l, r) 表示直询 a 序列 [l，r] 区间内的最大子段和，那么最终我们要求的答案就是 get(nums, 0，nums.size()-1)。

如何分治实现这个操作呢?

对于一个区间 [l, r]，我们取 m = $\left\lfloor \frac{l+r}{2} \right\rfloor$，对区间 [l, m] 和 [m+1, r] 分治求解。当递归逐层深入直到区间长度缩小为 1 的时候，递归 **开始回升**。这个时候我们考虑如何通过 [1, m] 区间的信息和 [m+1, r] 区间的信息合并成区间 [1, r] 的信息。

最关键的两个问题是:

- 我们要维护区间的哪些信息呢?
- 我们如何合并这些信息呢?

3-分治法

思路：

对于一个区间 [l, r]，我们可以维护四个量：

- **lSum** 表示 [l, r] 内以 l 为左端点的最大子段和；
- **rSum** 表示 [l, r] 内以 r 为右端点的最大子段和；
- **mSum** 表示 [l, r] 内的最大子段和；
- **iSum** 表示 [l, r] 的区间和。

简称 [l, m] 为 [l, r] 的 “左子区间”，
[m + 1, r] 为 [l, r] 的 “右子区间”

我们考虑如何维护这些量呢（如何通过左右子区间的信息合并得到 [l, r] 的信息）？

对于长度为 1 的区间 [i, i]，四个量的值都和 nums [i] 相等。

3-分治法

- 首先最好维护的是 iSum，区间 [l, r] 的 iSum 就等于“左子区间” 的 iSum 加上“右子区间”的 iSum。

$$iSum[l,r] = iSum[l,m] + iSum[m+1,r]$$

- 对于 [l, r] 的 lSum，存在两种可能，它要么等于 “左子区间” 的 lSum，要么等于 “左子区间” 的 iSum 加上 “右子区间” 的 lSum，二者取大。

$$lSum[l,r] = \max(lSum[l,m], iSum[l,m] + lSum[m+1,r])$$

- 对于 [l, r] 的 rSum，同理，它要么等于 “右子区间” 的 rSum，要么等于 “右子区间”的 iSum 加上 “左子区间” 的 rSum，二者取大。

$$rSum[l,r] = \max(rSum[m+1,r], iSum[m+1,r] + rSum[l,m])$$

3-分治法

- 当计算好上面的三个量之后，就很好计算 [l, r] 的 mSum 了。我们可以考虑 [l, r] 的 mSum 对应的区间是否跨越 m，

1. 它可**能不跨越 m**，也就是说 [l, r] 的 mSum 可能是 “左子区间” 的 mSum 和 “右子区间” 的 mSum 中的一个；
2. 它也**可能跨越 m**，可能是 “左子区间” 的 rSum 和 “右子区间” 的 lSum 求和。三者取大。

$$mSum = \max(\max(mSum[l,m], mSum[m+1,r]), rSum[l,m] + lSum[m+1,r])$$

3-分治法

```cpp
struct Status {
    int lSum, rSum, mSum, iSum;
};

Status pushUp(Status l, Status r) {
    int iSum = l.iSum + r.iSum;
    int lSum = max(l.lSum, l.iSum + r.lSum);
    int rSum = max(r.rSum, r.iSum + l.rSum);
    int mSum = max(max(l.mSum, r.mSum), l.rSum + r.lSum);
    return (Status) {lSum, rSum, mSum, iSum};
};
Status get(vector<int> &a, int l, int r) {
    if (l == r) {
        return (Status) {a[l], a[l], a[l], a[l]};
    }
    int m = (l + r) >> 1;
    Status lSub = get(a, l, m);
    Status rSub = get(a, m + 1, r);
    return pushUp(lSub, rSub);
}
int maxSubArray(vector<int>& nums) {
    return get(nums, 0, nums.size() - 1).mSum;
}
```

复杂度分析

- 时间复杂度：假设我们把递归的过程看作是一颗二叉树的先序遍历，那么这颗二叉树的深度的渐进上界为 O (log n)。这里的总时间相当于遍历这颗二叉树的所有节点，故总时间的渐进上界是 O (n)，故渐进时间复杂度为 O (n)。
- 空间复杂度：递归会使用 O (log n) 的栈空间，故渐进空间复杂度为 O (log n)。

合并两个有序线性表

1. 数组

给定两个排序后的数组 A 和 B。 编写一个方法，将 B 合并入 A 并排序。初始化 A 和 B 的元素数量分别为 *m* 和 *n*。

前向遍历(双指针)

```
void merge(vector<int>& A, int m, vector<int>& B, int n) {
    int idx_a = 0, idx_b = 0;
    int tmp[m+n],idx=0;
    while(idx_a < m || idx_b < n){
        if     (idx_a==m)  { tmp[idx++] = B[idx_b++]; }
        else if(idx_b==n)  { tmp[idx++] = A[idx_a++]; }
        else{
            if(A[idx_a]>B[idx_b]) tmp[idx++] = B[idx_b++];
            else                  tmp[idx++] = A[idx_a++];
        }}
    for(int i=0;i<m+n;i++)//赋值给A数组
        A[i] = tmp[i];
```

- **时间复杂度：O(m+n)**
- **空间复杂度：O(m+n)**

合并两个有序线性表

1. 数组

给定两个排序后的数组 A 和 B，其中 A 的**末端有足够的缓冲空间容纳 B**。 编写一个方法，将 B 合并入 A 并排序。初始化 A 和 B 的元素数量分别为 *m* 和 *n*。

思路：A 的后半部分预留了B的空间，若将比较后的结果**存放在A末尾而不会影响结果则可以不使用额外空间**。

证明：
对于idx_a， *A* 数组中有 **m−idx_a−1**个值 放入了A数组的后部。
对于 idx_b， 有**n-idx_b -1**个值放入了A数组后部。

A数组可用的空间为 **m+n-idx_a-1** (idx_a = m-1时有n个)，隐含地表达了a中访问过的元素，位置是可用的。

m+n-idx_a-1显然大于 m−idx_a−1 + n-idx_b -1，等价于 idx_b > -1显然成立。

预期的复杂度

- **时间复杂度：O(m+n)**
- **空间复杂度：O(1)**

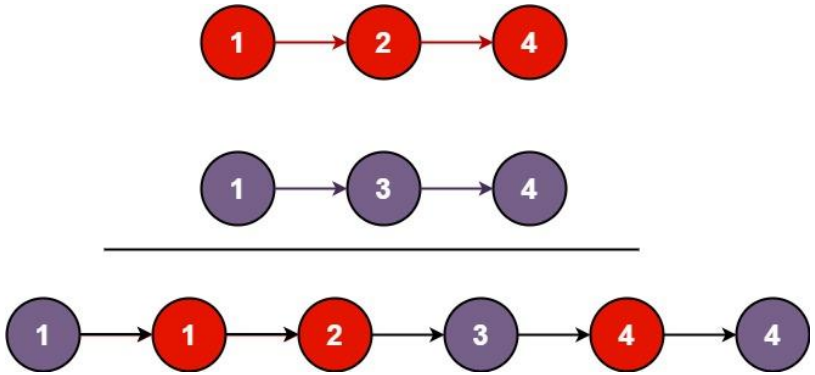
合并两个有序线性表


```
void merge(vector<int>& A, int m, vector<int>& B, int n) {
    int idx = m+n-1, idx_a=m-1, idx_b = n-1;
    while(idx_a >=0 && idx_b >= 0 ){
        if(A[idx_a]<B[idx_b]){
            A[idx--] = B[idx_b--];
        }
        else
            A[idx--] = A[idx_a--];
    }
    //处理未完成的B数组
    for(;idx_b>=0;) {
        A[idx--] = B[idx_b--];
    }
}
```

合并两个有序线性表

1. 数组
2. **链表**

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。



思路：

我们可以用迭代的方法来实现上述算法。当 l1 和 l2 都不是空链表时，判断 l1 和 l2 哪一个链表的头节点的值更小，将较小值的节点添加到结果里，当一个节点被添加到结果里之后，将对应链表中的节点向后移一位。

合并两个有序线性表

这里我们仍然采用有头节点的插入方法

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode* preHead = new ListNode(-1);

    ListNode* prev = preHead;
    while (l1 != nullptr && l2 != nullptr) {
        if (l1->val < l2->val) {
            prev->next = l1;
            l1 = l1->next;
        } else {
            prev->next = l2;
            l2 = l2->next;
        }
        prev = prev->next;
    }
    // 合并后 l1 和 l2 最多只有一个还未被合并完，我们直接将链表末尾指向未合并完的链表即可
    prev->next = l1 == nullptr ? l2 : l1;

    return preHead->next;
}
```

中国科学技术大学
University of Science and Technology of China

谢谢！