

# 链表 Linked list

---

## 1 用数组表达数据序列

---

### 1.1 使用数组的优点

- 方便和有效地访问序列中的任何项
  - 返回数组项中的第*i*个元素
  - 每个项都可以在固定的时间 $O(1)$ 内访问
  - 数组的这个特性被称为“随机访问”
  - 非常紧凑(就内存而言)
- 

### 1.2 数组的缺点

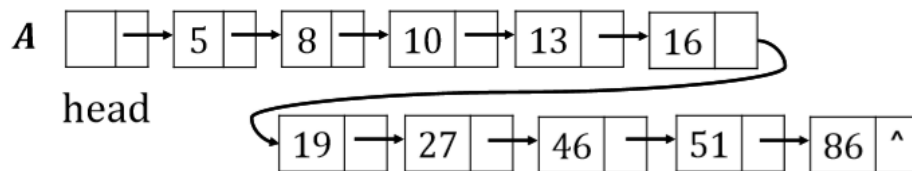
- 必须指定初始数组大小
  - 调整数组大小是可能的，但不是那么容易
  - 很难在新的位置插入/删除元素 $O(n)$
- 

## 2 基本信息

---

### 2.1 介绍

是另一种表示数据序列的方式



链表将元素序列存储在**单独的Node**中

每个Node包含:一个单独的元素，一个“**链接**”到包含下一个元素的节点

链表中最后一个节点的链接值为**NULL**

整个链表由一个变量表示，该变量持有对第一个**Node**的引用

---

### 2.2 链表的优点

它可以无限制地生长(而不是固定长度)

很容易插入/删除元素 $O(1)$

---

### 2.3 链表的缺点

它们不提供随机访问

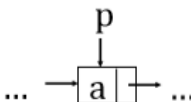
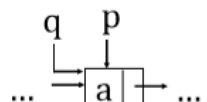


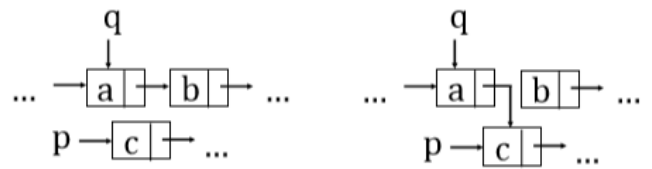
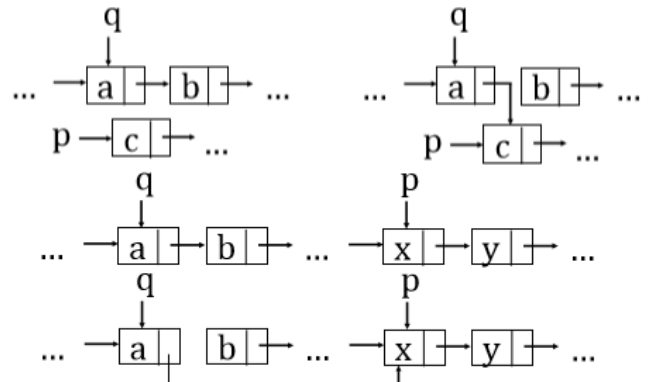
需要“**向下走**”列表访问一个项目

这些链接占用**额外的内存**

单链表的内存开销比数组大，因为初始化的不仅有Node的值，还有下一个节点的位置  
不紧凑(在内存方面)

## 2.4 操作

### 基本操作

操作	解释	图示
$q \leftarrow p$	Node p赋值给Node q	<div>Before</div>  <div>After</div> 
$q \leftarrow p.next$	Node p的下一个Node赋值给Node q	
$p \leftarrow p.next$	Node p的下一个Node赋值给Node p	
$q.next \leftarrow p$	Node q的下一个Node赋值给Node p	
$q.next \leftarrow p.next$	Node q的下一个Node赋值给Node p的下一个Node	

## 3 链表的运用

### 3.1 链表的遍历

#### ❗伪代码

```

1  Algorithm: traverse1(A):
2      if (A=NULL)
3          return
4      else
5          print A.value
6          traverse(A.next)
7
8  Algorithm: traverse2(A):
9      node trav ← A
10     while(trav != NULL)
11         print trav.value
12         trav ← trav.next

```

### 时间复杂度分析 $O(n)$

遍历整个链表需要从头到尾遍历

---

## 3.2 链表的插入操作

### 伪代码

```

1  /*
2  A: 链表初始Node的地址
3  N: 链表要插入的Node
4  i: 链表要插入Node的位置
5  */
6  Algorithm: insertNode(A, N, i):
7      count ← 0
8      p ← A // 指针
9
10     // 从i-1开始就需要更新node了!
11     while(count < i - 1)
12         p ← p.next
13         count++
14     end
15
16     // 考试重点
17     // -----
18     p1 ← p.next
19     p.next ← N
20     N.next ← p1
21     // -----
22
23     return A

```

### 时间复杂度分析 $O(n)$

插入一个Node到链表需要从头寻找到

给一个链表，任意一个位置插入一个新的值，时间复杂度 $O(n)$

---

### 3.3 链表的删除操作

#### ❗伪代码

```
1  /*
2  A: 链表初始元素的地址
3  i: 链表要删除元素的位置
4  */
5  Algorithm: deleteNode(A, i):
6    count ← 0
7    p ← A
8
9    //从i-1开始就需要更新node了!
10   while(count < i - 1)
11     p ← p.next
12     count++
13   end
14
15   p.next ← p.next.next
16
17   return A
```

#### 时间复杂度分析 $O(n)$

删除一个Node到链表需要从头寻找到

给一个链表，任意一个位置删除一个值，时间复杂度 $O(n)$

如果我们想从单链表中删除现有结点 `cur`，可以分两步完成：

- 找到 `cur` 的上一个结点 `prev` 及其下一个结点 `next`
- 接下来链接 `prev` 到 `cur` 的下一个节点 `next`

在我们的第一步中，我们需要找出 `prev` 和 `next`。使用 `cur` 的参考字段很容易找出 `next`，但是，我们必须从头结点遍历链表，以找出 `prev`，它的平均时间是  $O(N)$ ，其中  $N$  是链表的长度。因此，删除结点的时间复杂度将是  $O(N)$

空间复杂度为  $O(1)$ ，因为我们只需要常量空间来存储指针

删除指定的Linked List的节点有两步

- 使得前一个Node指向后一个Node
- 把删除的Node里面的next值指到null（删干净）

---

### 3.4 链表的查找操作

#### ❗伪代码

```

1 Algorithm: findValue(A, i):
2   count ← 0
3   p ← A
4
5   while(p != NULL)
6       if(count = i)
7           return add1
8       p ← p.next
9       count++
10  end
11
12  return -1//存在i太大找不到的情况

```

### 时间复杂度分析O(n)

查询一个Node到链表需要从头寻找到

给一个链表，查询到某一个具体的Node，时间复杂度O (n)

---

## 3.5 链表的更新操作

### 伪代码

```

1 Algorithm: updateNodes(A, value, i):
2   count ← 0
3   p ← A
4
5   while(p != NULL)
6       if(count = i)
7           p.value ← value
8       p ← p.next
9       count++
10  end
11
12  return A//存在i找不到的情况

```

### 时间复杂度分析O(n)

更新一个Node到链表需要从头寻找到

给一个链表，查询到某一个具体的Node，时间复杂度O (n)

---

## 4 高级链表

### 4.1.双向链表

双链表以类似的方式工作，但还有一个引用字段，称为 `prev` 字段。有了这个额外的字段，您就能够知道当前结点的前一个结点

与单链表类似，我们将介绍在双链表中如何访问数据、插入新结点或删除现有结点

- 我们可以与单链表相同的方式访问数据：
- 我们不能在常量级的时间内访问随机位置
- 我们必须从头部遍历才能得到我们想要的第一个结点

在最坏的情况下，时间复杂度将是  $O(N)$ ，其中  $N$  是链表的长度

对于添加和删除，会稍微复杂一些，因为我们还需要处理 `prev` 字段，在接下来的两篇文章中，我们将介绍这两个操作

---

## 4.2 哑节点

在对链表进行操作时，一种常用的技巧是添加一个哑节点（dummy node），它的 `next` 指针指向链表的头节点。这样一来，我们就不需要对头节点进行特殊的判断了。

特别地，在某些语言中，由于需要自行对内存进行管理。因此在实际的面试中，对于「是否需要释放被删除节点对应的空间」这一问题，我们需要和面试官进行积极的沟通以达成一致。下面的代码中默认不释放空间

---

## 5 链表的技巧

### 5.1 链表中的双指针

两种使用双指针技巧的情景：

1. 两个指针**从不同位置出发**：一个从始端开始，另一个从末端/某个位置开始；
2. 两个指针**以不同速度移动**：一个指针快一些，另一个指针慢一些。

对于**单链表**，因为我们只能在一个方向上遍历链表，所以第一种情景可能无法工作。然而，第二种情景，也被称为慢指针和快指针技巧，是非常有用的

本章节中，我们将重点讨论链表中的慢指针和快指针问题，并告诉你如何解决这类问题

---

## 6 链表的问题

### 6.1 链表成环问题

想象一下，有两个速度不同的跑步者。如果他们在直路上行驶，快跑者将首先到达目的地。但是，如果它们在圆形跑道上跑步，那么快跑者如果继续跑步就会追上慢跑者。

这正是我们在链表中使用两个速度不同的指针时会遇到的情况：

- 如果没有环，快指针将停在链表的末尾
- 如果有环，快指针最终将与慢指针相遇

所以剩下的问题是：

这两个指针的适当速度应该是多少？

一个安全的选择是每次移动慢指针一步，而移动快指针两步。每一次迭代，快速指针将额外移动一步。如果环的长度为  $M$ ，经过  $M$  次迭代后，快指针肯定会多绕环一周，并赶上慢指针

### 时间复杂度分析

空间复杂度分析容易。如果只使用指针，而不使用任何其他额外的空间，那么空间复杂度将是  $O(1)$ 。但是，时间复杂度的分析比较困难。为了得到答案，我们需要分析运行循环的次数。

在前面的查找循环示例中，假设我们每次移动较快的指针 2 步，每次移动较慢的指针 1 步。

如果没有循环，快指针需要  $N/2$  次才能到达链表的末尾，其中  $N$  是链表的长度。

如果存在循环，则快指针需要  $M$  次才能赶上慢指针，其中  $M$  是列表中循环的长度。

显然， $M \leq N$ 。所以我们将循环运行  $N$  次。对于每次循环，我们只需要常量级的时间。因此，该算法的时间复杂度总共为  $O(N)$ 。

---

## 6.2 交叉链表问题

空间复杂度  $O(1)$  时间复杂度为  $O(n)$

这里使用图解的方式，解释比较巧妙的一种实现。

根据题目意思

如果两个链表相交，那么相交点之后的长度是相同的

我们需要做的事情是，让两个链表从同距离末尾同等距离的位置开始遍历。这个位置只能是较短链表的头结点位置。

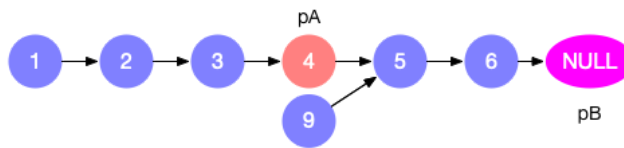
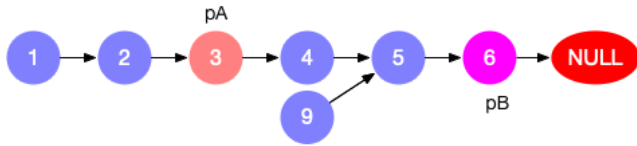
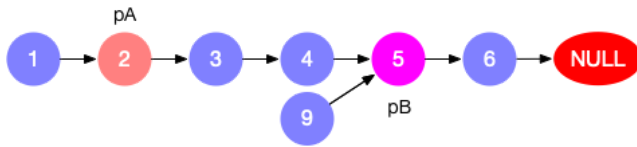
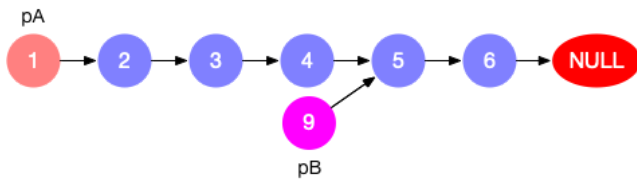
为此，我们必须消除两个链表的长度差

- 指针 pA 指向 A 链表，指针 pB 指向 B 链表，依次往后遍历
- 如果 pA 到了末尾，则  $pA = headB$  继续遍历
- 如果 pB 到了末尾，则  $pB = headA$  继续遍历

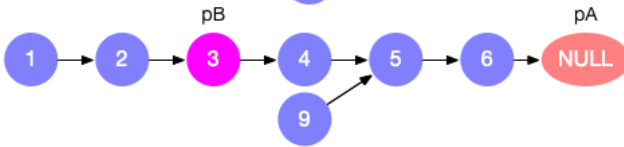
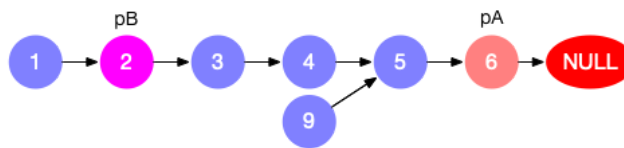
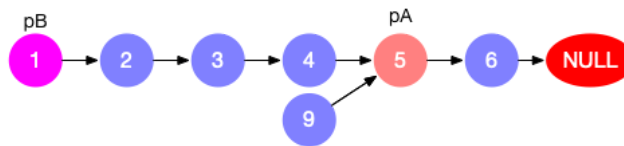
比较长的链表指针指向较短链表head时，长度差就消除了

如此，只需要将最短链表遍历两次即可找到位置

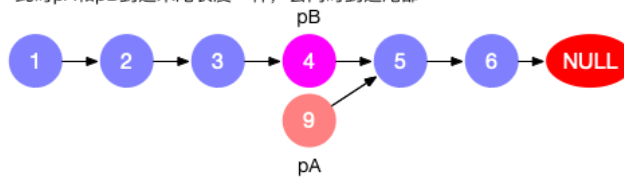
听着可能有点绕，看图最直观，链表的题目最适合看图了



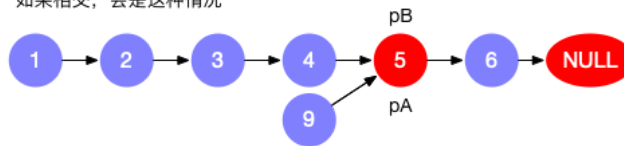
pB到了B链表末尾，指向链表A的头部，此时A和B的长度差为B链表的长度也就是3



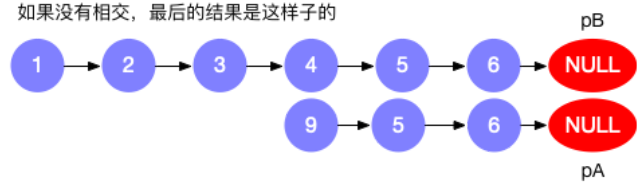
pA到了末尾，移动到B链表的头部  
此时pA和pB到达末尾长度一样，会同时到达尾部



如果相交，会是这种情况



如果没有相交，最后的结果是这样子的





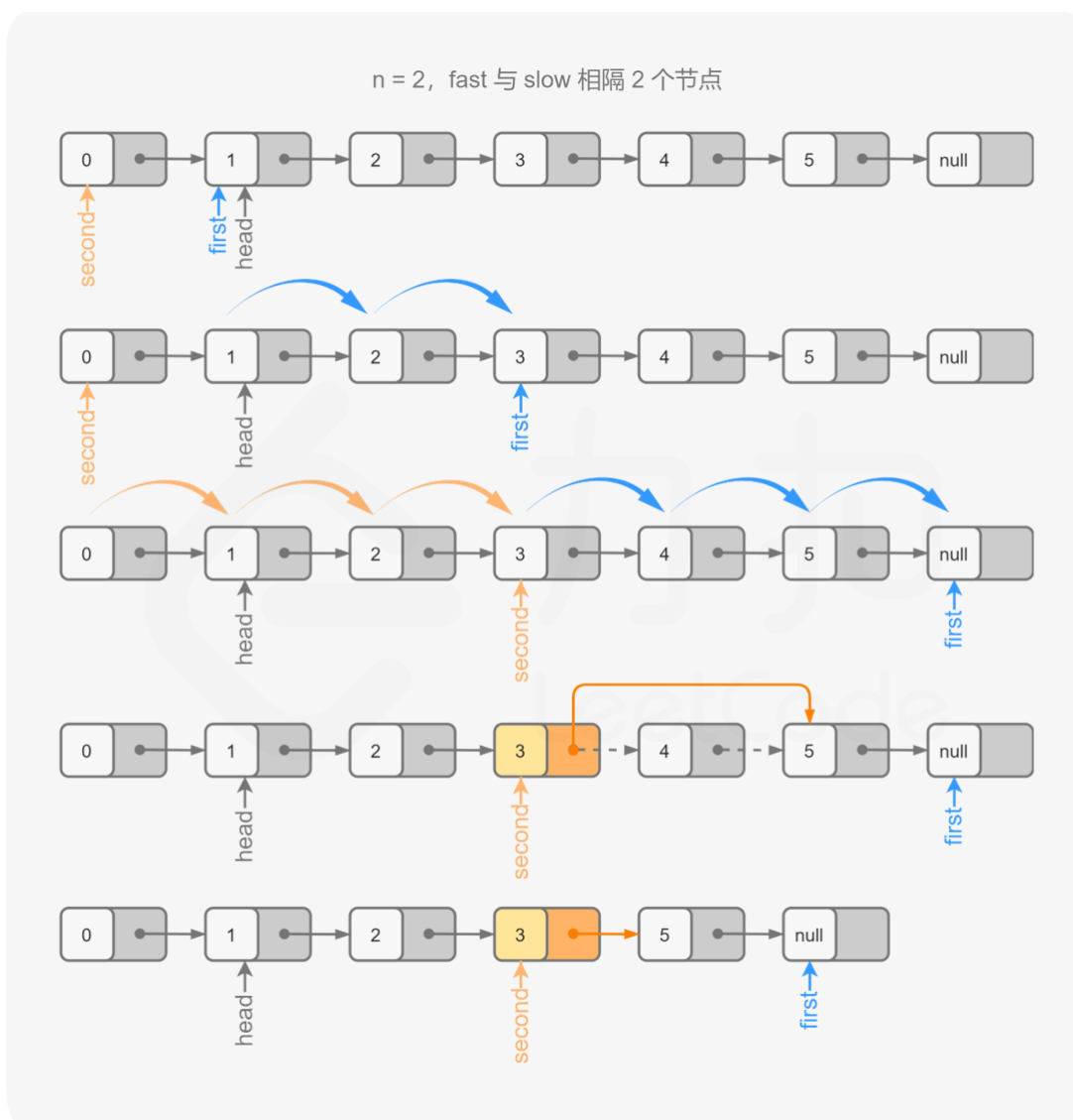
## 代码

```
1 public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
2     if (headA == null || headB == null) return null;
3     ListNode pA = headA, pB = headB;
4     while (pA != pB) {
5         pA = pA == null ? headB : pA.next;
6         pB = pB == null ? headA : pB.next;
7     }
8     return pA;
9 }
```

### 6.3 单向链表删除列表的倒数第n个节点

由于我们需要找到倒数第  $n$  个节点，因此我们可以使用两个指针 `first` 和 `second` 同时对链表进行遍历，并且 `first` 比 `second` 超前  $n$  个节点。当 `first` 遍历到链表的末尾时，`second` 就恰好处于倒数第  $n$  个节点。

如果我们能够得到的是倒数第  $n$  个节点的前驱节点而不是倒数第  $n$  个节点的话，删除操作会更加方便。因此我们可以考虑在初始时将 `second` 指向哑节点，其余的操作步骤不变。这样一来，当 `first` 遍历到链表的末尾时，`second` 的下一个节点就是我们需要删除的节点



## 6.4 多项式运算问题

### 多项式相加

如果做多项式加减的时候，用链表会比数组快，因为数组一定要存好每一项（即使系数是0）的情况，而链表不需要

### 多项式相乘

假设有 $p(x)$ 和 $r(x)$ ，它们的级数是 $d_1$ 和 $d_2$ ，有 $n_1$ 和 $n_2$ 项数

- 如果要用数组做多项式乘法，要做 $d_1 * d_2$ 次操作
- 如果要用链表做多项式乘法，要做 $n_1 * n_2$ 次操作

## 7 数组与链表

### 7.1 在内存中的区别

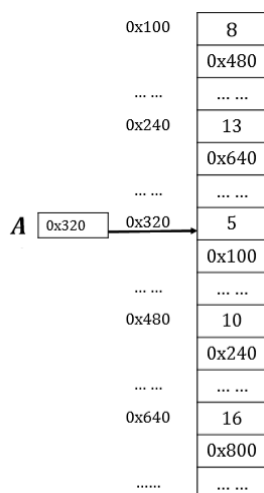
在数组中，元素占据连续的内存位置

5	8	10	13	16	19	27	46	51	86
0x100	0x104	0x108	0x112	0x116	0x120	0x124	0x128	0x132	0x136

在链表中，每个Node都是一个**独立的对象**

#### Node不必在内存中相邻

这就是为什么我们需要从一个Node到下一个Node的链接



给一个数组 $n$ ，把前 $k$ 个数移动到后，最优算法是什么

Linked List的操作与相同在Array的操作里

如果给了一个数组是sorted的，我们一定要想sorted是不是有什么意义，快速的

Given two sorted arrays A and B in ascending order, with  $n$  and  $m$  integers respectively, All  $m+n$  integers are distinct, design an algorithm to find the median in all  $m+n$  integers

Brute force  $O(m+n)$

Binary search  $O(\log n * \log m)$

Divide and Conquer  $O(\log(n+m))$

Median property  $O(\log(\min\{n,m\}))$

