**1.时间复杂度和空间复杂度分析**

1.1时间复杂度：

**1.1.1分类**

表达方式：Big O notation

O(1)：常数复杂度

O(log n)：对数复杂度

O(n)：线性时间复杂度

O(n^2)：平方

O(n^3)：立方

O(2^n)：指数

O(n!)：阶乘

*注意*：*只看最高复杂度的运算*

举例：

O(N)

for(int i = 1; i <= n; i++){

System.out.print("Hey");

};

O(N^2)

for(int i = 1; i <= n; i++){

for(int j = 1; j <= n; j++){

System.out.println("Hey");

}

};

O(log(n))

for(int i = 1; i < n; i = i\*2){

System.out.println("Hey");

};

O(k^n)

int fib(int n){

if(n<2) return n;

return fib(n-1) + fib(n-2);

};

*简单的递归求****Fibonacci****非常慢*，*因为需要指数级的时间复杂度*

**1.1.2.主定理**

用于算出分治和递归函数的时间复杂度

主要分为四种

* 二分查找(Binary search)

O(log n)

一般用在有序数列查找目标数

* 二叉树遍历(Binary tree traversal)

O(n)

简化思考方式，因为二叉树遍历的话，每个节点访问一次，且仅访问一次，所以为O(n)

* 排好序的二维矩阵中进行二分查找(Optimal sorted matrix search)

O(n)

*注：如果是一维的数组进行二分查找是log n*

* 归并排序(Merge sort)

O(n log n)

**1.1.3.常见面试题**

* 二叉树遍历 - 前序、中序、后序：时间复杂度是多少？

*答：O(n)，通过主定理可得，或解释为 不管是前序中序后序它遍历二叉树的时候，每个节点会访问一次且仅访问一次，所以它的时间复杂度就是线性于二叉树的结点总数，也就是O(n)的时间复杂度*

* 图的遍历：时间复杂度是多少？

*答：O(n)，同理可得图里的每个节点访问一次，且仅访问一次，所以是O(n),n为图中的节点总数*

* 搜索算法：DFS(深度优先)、BFS(广度优先)时间复杂度是多少？

*答：O(n)，因为每个节点访问一次，其中n指的是搜索空间里面的节点总数*

* 二分查找：时间复杂度是多少？

*答：O(log n)*

## 2.数组、链表、跳表的基本实现和特性

### 2.1.数组

#### 2.1.1.常见写法

Array

* Java、C++：int a[100];
* Python：list = []
* Javascript： let x = [1,2,3]

对于数组内的元素，单元类型为泛型，即能放任何类型的数据

#### 2.1.2.硬件实现

内存管理器：每当申请数组，计算机实际上是在内存中开辟了一段连续的地址，每一个地址可以通过内存管理器进行访问，直接访问其中的任何一个元素，时间复杂度都是一样的，也就是常数时间称为O(1)

因为可以随机访问任何一个元素，所以它的访问时间非常快

#### 2.1.3.Array增加/删除元素

插入的操作不再是常数级，而是O(n)的复杂度，因为需要挪位

在源代码中，新增，如果长度不够，就扩充一倍，然后用到System.arraycopy()，将原有的数组内容复制一遍，很低效

#### 2.1.4.数组的时间复杂度

prepend O(1)

append O(1)

lookup O(1)

insert O(n)

delete O(n)

### 2.2.链表 Linked List

为了弥补数组增删的不便，通过每个元素定义value和next，next指向下一个，串成一个类似数组的结构

*注意：每个元素用class来定义，通常称为node，其中有两个成员变量，一个就是value，也可以是一个类，另一个即为next的指针，指向它的下一个元素，在java的话就是next，它是一个引用，引用到下一个元素*

链表也分为单链表，双向链表，循环链表

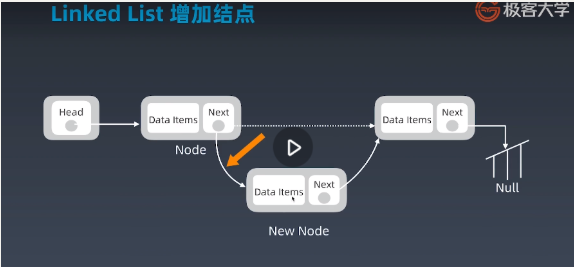
单链表：头指针head、尾指针tail、最后一个元素，next指针指向为空

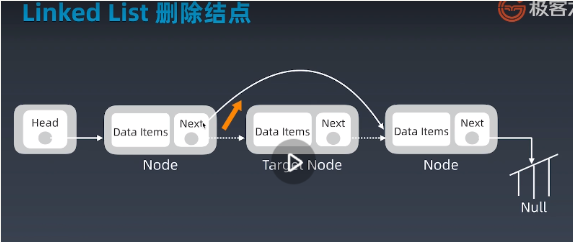
双向链表：有next，还有previous（Java的链表默认为双向链表结构）

循环链表：最后一个元素的next指针指向head

#### 2.2.1Linked List的新增/删除节点

直接对next指针进行操作，没有引起整个链表的群移操作，也不需要挪动多个元素到新的位置，所以它的移动效率和修改操作效率非常高为O(1)





所以新增、删除节点的时间复杂度都是O(1)，但查抄因为也得一个个查过来，所以为O(n)

#### 2.2.2.链表的时间复杂度

prepend O(1)

append O(1)

lookup O(n)

insert O(1)

delete O(1)

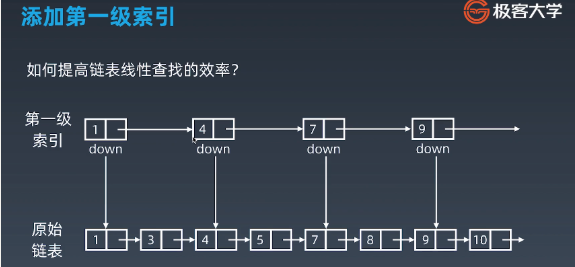
### 2.3.跳表 Skip List

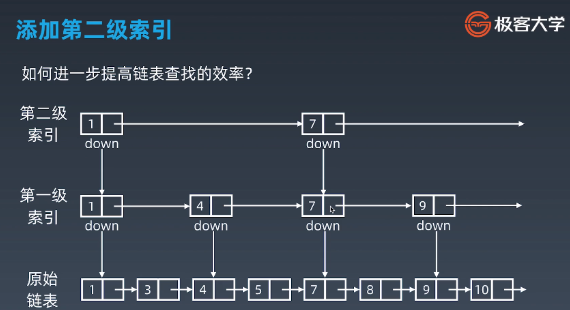
*注意：只能用于元素有序的情况*

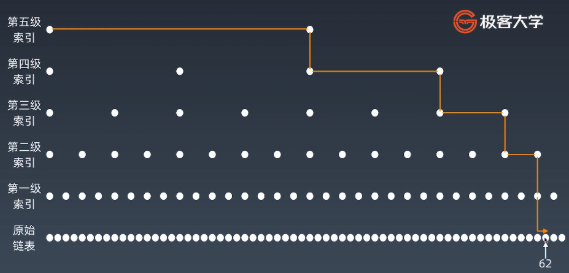
跳表对标的是平衡树(AVL Tree)和二分查找，是一种插入、删除、搜索都是O(log n)的数据结构

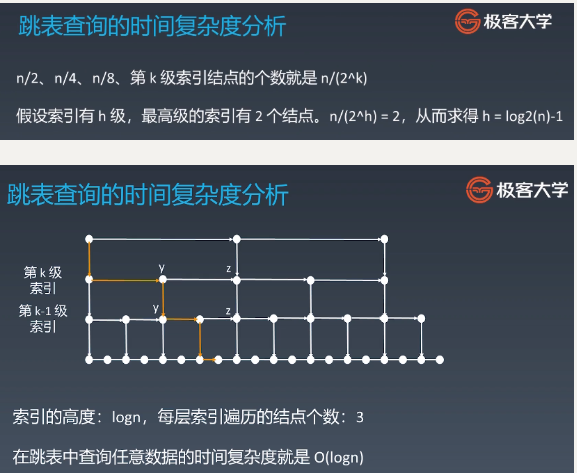
它最大的优势是原理简单、容易实现、方便扩展、效率更高。因此在一些热门的项目里用来替代平衡树，如Redis、LevelDB等

#### 2.3.1.跳表实现

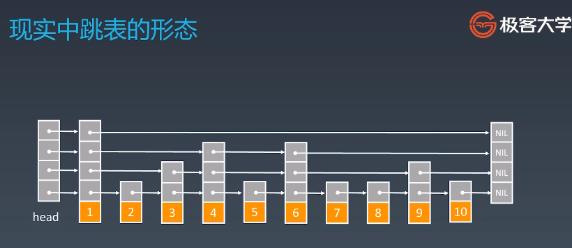






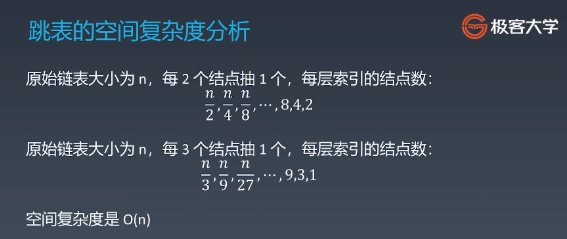


所以跳表的时间复杂度为O(log n)，同二分查找



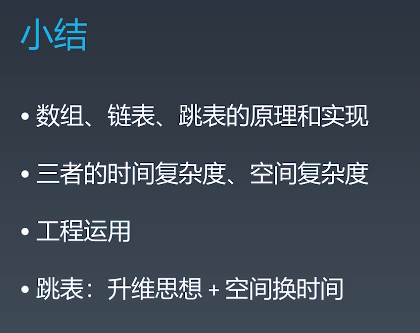
在现实情况中，由于元素的增加和删除导致索引会不完整的跨步，所以维护成本相对较高。当增加个元素时，需要把索引更新一遍，同理删除也是。在此过程中，它的时间复杂度变为log n

空间复杂度：



### 1.4.工程中的应用

* LRU Cache(最少最近使用) - Linked list： [LRU 缓存机制](http://leetcode-cn.com/problems/lru-cache)
* Redis - Skip List：[跳跃表](http://redisbook.readthedocs.io/en/latest/internal-datastruct/skiplist.html)、[为啥 Redis 使用跳表（Skip List）而不是使用 Red-Black？](http://www.zhihu.com/question/20202931)



## 2.实战题目解析思路

### 2.1.练习步骤

(1) 5-10分钟：读题和思考

(2) 有思路：自己开始做和写代码；不然，马上看题解！

(3) 默写背诵、熟练

(4) 然后开始自己写（闭卷）

### 2.2.切题四件套

* Clarification：确认题目
* Possible solutions：想多种解法，比较不同的方法间的时间空间复杂度
  + compare(time/space)
  + optimal(加强)
* Coding(多写)
* Test cases ：多写测试样例

### 2.3.五毒神掌！

* 刷题第一遍
* 五分钟：读题 + 思考
* 直接看解法：注意！多解法，比较解法优劣
* 背诵、默写好的解法
* 刷题第二遍
* 马上自己写 ----> Leetcode提交
* 多种解法比较、体会 ---> 优化！
* 刷题第三遍
* 过了一天后，再重复做题
* 不同解法的熟练程度 ---> 专项训练 （多做不熟练的题）
* 刷题第四遍
* 过了一周：反复回来练习相同的题目
* 刷题第五遍
* 面试前一周恢复性训练

### 2.4.解题时的思路

当解题没头绪的时候，可以思考有无最简暴力法，基本情况是什么

其次查找是否存在最近重复问题，数学归纳法

## 3.实战记录

夹逼法

有序数组中的双指针理论

环形链表 快慢指针