

# Java集合相关面试题

## 导读

本文共分为三部分，第一个是数据结构的普及，让大家快速掌握常见数据结构的一些特点。第二个是集合相关的面试题，这个也是本文的重点部分，包含了常见面试题的必问部分，比如：`ArrayList`和`HashMap`相关等等。第三个是面试现场或者叫做真实面试还原场景，我会以面试官和候选人的角度去提出问题和解答问题，希望能帮助到你！

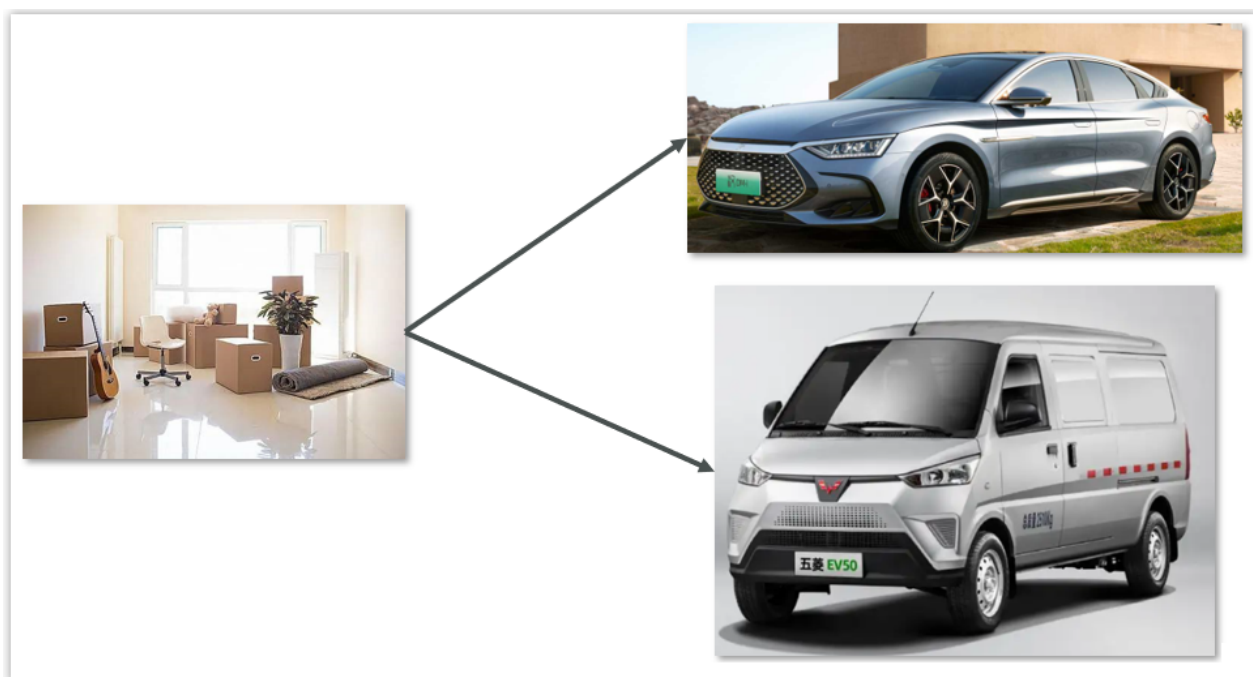
## 1 数据结构

### 1.1 算法复杂度分析

#### 1.1.1 概念

1. 数据结构是指一组数据的存储结构
2. 算法就是操作数据的方法

举个例子：



比如，你要搬家，有一堆货物，这个时候你可以选择使用小轿车拉走货物，你可以选择小货车拉走货物。其实现在你选择哪辆车装载货物就相当于选择了哪种数据结构。

你选择小货车拉走货物，但是货物依然很多，你这个时候需要规整一下，难看怎么着更能节省空间，更能节省效率，这个动作就是算法了

清楚了这些概念之后，如果只是单独讲数据结构和算法是不合适的，它们两个是相辅相成的。

### 1.1.2 算法复杂度

复杂度也叫渐进复杂度，包括时间复杂度和空间复杂度，用来分析算法执行效率与数据规模之间的增长关系，可以粗略地表示，越高阶复杂度的算法，执行效率越低。

复杂度描述的是算法执行时间或占用内存空间随数据规模的增长关系。

举例：

有200人需要从成都到北京，可以选择很多交通工具，每个交通工具的载客量和时间都不相同

- 大型载人客车，50人每车，需要4辆车，时间大概为30小时
- 普通火车，载客量超大，满足200人的需求，时间大概为20小时
- 高铁，载客量超大，满足200人的需求，时间大概为9小时
- 飞机，载客量适中，满足200人的需求，时间大概为3小时

算法的执行效率，粗略地讲，就是算法代码执行的时间，那如何在不直接运行代码的前提下粗略的计算执行时间呢？

分析以下代码

```

/**
 * 求1~n的累加和
 * @param n
 * @return
 */
public static int sum(int n){
    int sum = 0;
    for (int i = 1; i < n; ++i) {
        sum = sum + i;
    }
    return sum;
}

```

假设每行代码执行时间都一样为：timer

此代码的执行时间为：(3n+3) timer

总结：所有代码的执行时间 **$T(n)$** 与代码的执行次数成正比。

按照该思路我们接着看下面一段代码

```

public static int sum2(int n){
    int sum = 0;
    for (int i = 1; i < n; ++i) {
        for (int j = 1; j < n; ++j) {
            sum = sum + i * j;
        }
    }
    return sum;
}

```

同理，此代码的执行时间为：(3  $n^2$  + 3n + 3) \* timer

因此有一个重要结论：代码的执行时间 **$T(n)$** 与总的执行次数相关，我们可以把这个规律总结成一个公式。

**$T(n) = O(f(n))$**

$T(n)$ 表示代码的执行时间， $n$ 表示数据规模的大小， $f(n)$ 表示了代码执行的总次数，它是一个公式因此用 $f(n)$ 表示， $O$ 表示了代码执行时间与 $f(n)$ 成正比

### 1.1.3 大O复杂度表示法

大  $O$  时间复杂度实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，所以，也叫作渐进时间复杂度，简称时间复杂度。

当 $n$ 很大时，公式中的低阶，常量，系数三部分并不左右其增长趋势，因此可以忽略，我们只需要记录一个最大的量级就可以了，因此如果用大 $O$ 表示刚刚的时间复杂度可以记录为

第一个例子中的 $T(n)=O(3n+3) \rightarrow T(n)=O(n)$

第二个例子中的 $T(n)=O(3n^2 + 3n + 3) \rightarrow T(n)=O(n^2)$

常见的复杂度

描述	表示形式
常数	$O(1)$
线性	$O(n)$
对数	$O(\log n)$
线性对数	$O(n * \log n)$
平方	$O(n^2)$
立方	$O(n^3)$
k次方	$O(n^k)$
指数	$O(2^n)$
阶乘	$O(n!)$

### 1.1.4 $O(1)$

```
public int test01(int n){  
    int i=0;  
    int j = 1;  
    return i+j;  
}
```

代码只有三行，它的复杂度也是 $O(1)$ ，而不是 $O(3)$

再看如下代码：

```
public void test02(int n){  
    int i=0;  
    int sum=0;  
    for(;i<100;i++){  
        sum = sum+i;  
    }  
    System.out.println(sum);  
}
```

整个代码中因为循环次数是固定的就是100次，这样的代码复杂度我们认为也是 $O(1)$

一句话总结：只要代码的执行时间不随着 $n$ 的增大而增大，这样的代码复杂度都是 $O(1)$

### 1.1.5 $O(n)$

这个大家已经不陌生了，就是刚才上面的两个例子

```

/**
 * 求1~n的累加和
 * @param n
 * @return
 */
public int sum(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    return sum;
}

```

一层for循环的时间复杂度是  $O(n)$

```

public static int sum2(int n){
    int sum = 0;
    for (int i = 1; i < n; ++i) {
        for (int j = 1; j < n; ++j) {
            sum = sum + i * j;
        }
    }
    return sum;
}

```

两层for循环的时间复杂度是  $O(n^2)$

### 1.1.6 $O(\log n)$

对数复杂度非常的常见，但相对比较难以分析，代码：

```


public void test04(int n){
    int i=1;
    while(i<=n){
        i = i * 2;
    }
}

```

分析这个代码的复杂度，我们必须再强调一个前提：复杂度分析就是要弄清楚代码的执行次数和数据规模 $n$ 之间的关系

以上代码最关键的一行是：`i = i * 2`，这行代码可以决定这个while循环执行代码的行数，`i`的值是可以无限接近`n`的值的。如果`i`一旦大于等于了`n`则循环条件就不满足了。也就说达到了最大的行数。我们可以分析一下`i`这个值变化的过程

分析过程如下：

代码执行次数：	1	2	3	4			$2^x = n$ $x = \log_2 n$	
变量 <i>i</i> 的值：	2	4	8	16				
	$2^1$	$2^2$	$2^3$	$2^4$				
					x表示第几次		x记作以2为底n的对数	

由此可知，代码的时间复杂度表示为 $O(\log n)$

### 1.1.7 $O(n * \log n)$

分析完 $O(\log n)$ ，那 $O(n * \log n)$ 就很容易理解了，比如下列代码：

```
public void test05(int n){
    int i=0;
    for(;i<=n;i++){
        test04(n);
    }
}

public void test04(int n){
    int i=1;
    while(i<=n){
        i = i * 2;
    }
}
```

### 1.1.8 空间复杂度

空间复杂度全称是渐进空间复杂度，表示算法占用的额外存储空间与数据规模之间的增长关系

看下面代码

```
public void test(int n){
    int i=0;
    int sum=0;
    for(;i<n;i++){
        sum = sum+i;
    }
    System.out.println(sum);
}
```

代码执行并不需要占用额外的存储空间，只需要常量级的内存空间大小，因此空间复杂度是 $O(1)$

再来看一个其他例子：

```
void print(int n) {
    int i = 0;
    int[] a = new int[n];
    for (i; i < n; ++i) {
        a[i] = i * i;
    }
    for (i = n-1; i >= 0; --i) {
        System.out.println(a[i]);
    }
}
```

传入一个变量 $n$ ，决定申请多少的`int`数组空间内存，此段代码的空间复杂度为 $O(n)$

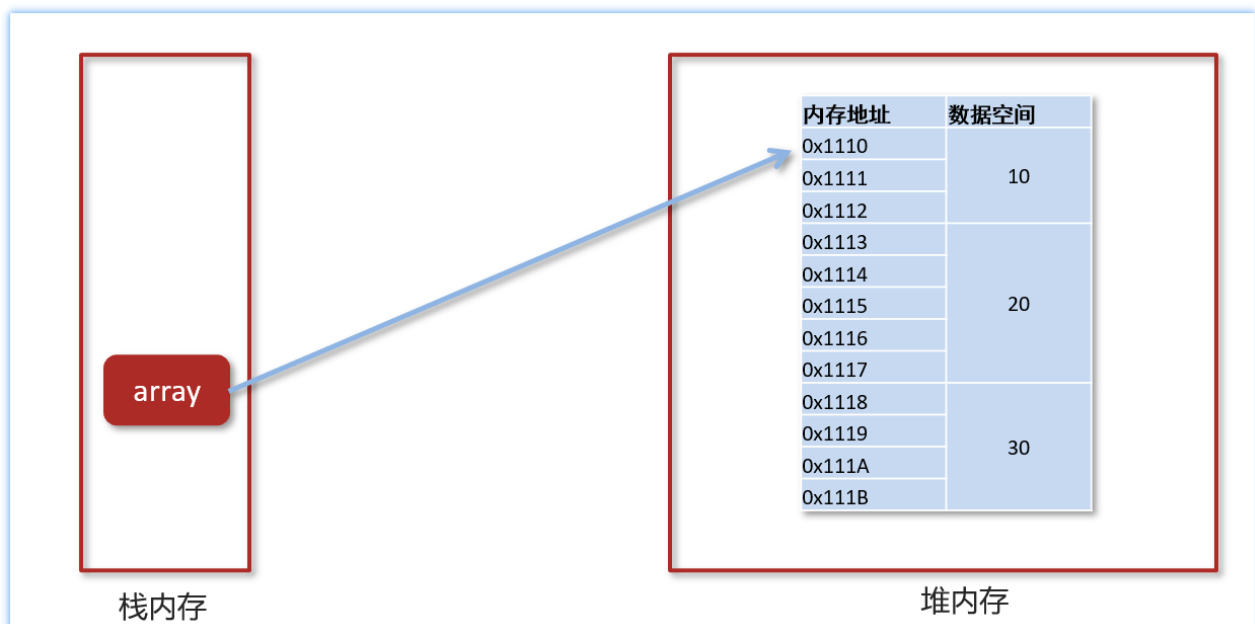
我们常见的空间复杂度就是 $O(1)$ , $O(n)$ , $O(n^2)$ ，其他像对数阶的复杂度几乎用不到，因此空间复杂度比时间复杂度分析要简单的多。

## 1.2 数组

### 1.2.1 数组概述

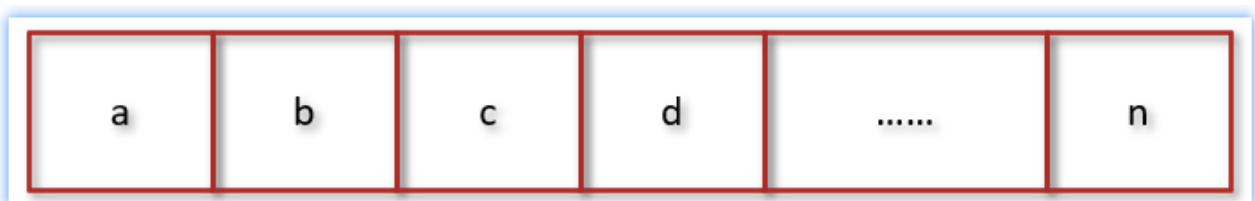
数组（Array）是一种用连续的内存空间存储相同数据类型数据的线性数据结构。



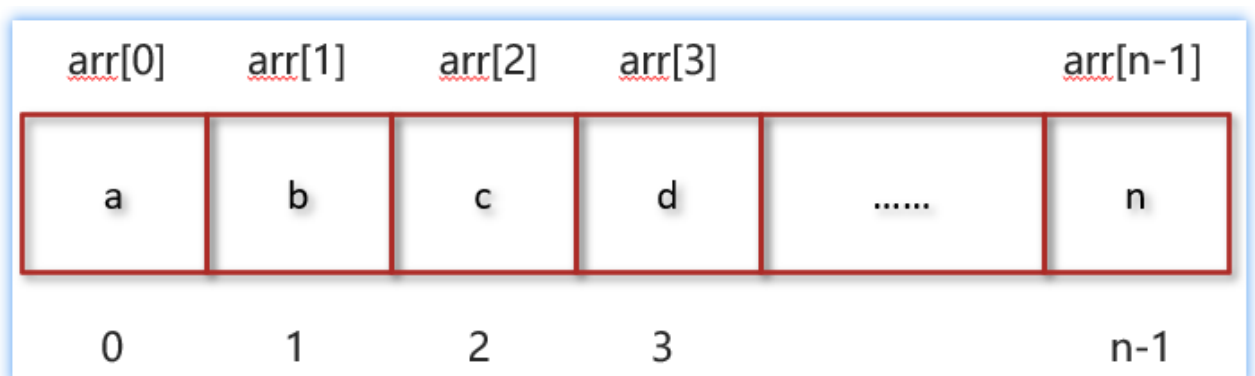


数组的表示方式：使用下标来获取数组元素数据

设有一个字符串数组arr，元素个数为n



通过下标来表示和获取元素，数组下标从0开始



思考：操作平台是如何根据下标来找到对应元素的内存地址的呢？

我们拿一个长度为10的数组来举例，`int [] a= new int[10]`，在下面的图中，计算机给数组分配了一块连续的空间，100-139，其中内存的起始地址为  
`baseAddress=100`

int[] a	内存地址
a[0]	100-103
a[1]	104-107
a[2]	108-111
...	...
a[9]	136-139

计算机给每个内存单元都分配了一个地址，通过地址来访问其数据，因此要访问数组中的某个元素时，首先要经过一个寻址公式计算要访问的元素在内存中的地址：

```
a[i] = baseAddress + i * dataTypeSize
```

`dataTypeSize`代表数组中元素类型的大小，在这个例子中，存储的是int型的数据，因此`dataTypeSize=4`个字节

## 1.2.2 数组的特点

### 1.查询O(1)

数组元素的访问是通过下标来访问的，计算机通过数组的首地址和寻址公式能够很快速的找到想要访问的元素

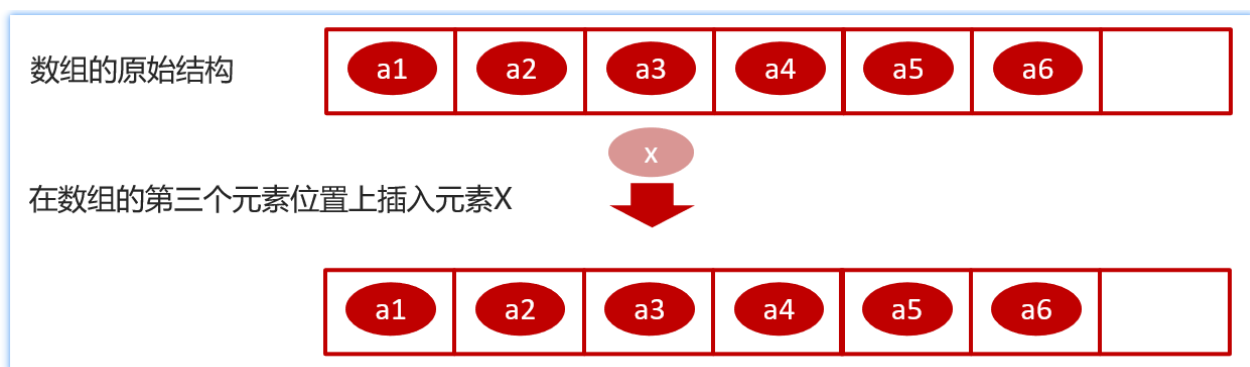
```
public int test01(int[] a,int i){  
    return a[i];  
    // a[i] = baseAddress + i * dataSize  
}
```

代码的执行次数并不会随着数组的数据规模大小变化而变化，是常数级的，所以查询数据操作的时间复杂度是 $O(1)$

## 2.插入 $O(n)$

数组是一段连续的内存空间，因此为了保证数组的连续性会使得数组的插入和删除的效率变的很低。

假设数组的长度为  $n$ ，现在如果我们需要将一个数据插入到数组中的第  $k$  个位置。为了把第  $k$  个位置腾出来给新来的数据，我们需要将第  $k \sim n$  这部分的元素都顺序地往后挪一位。如下图所示：



新增之后的数据变化，如下



所以：

插入操作，最好情况下是 $O(1)$ 的，最坏情况下是 $O(n)$ 的，平均情况下的时间复杂度是 $O(n)$ 。

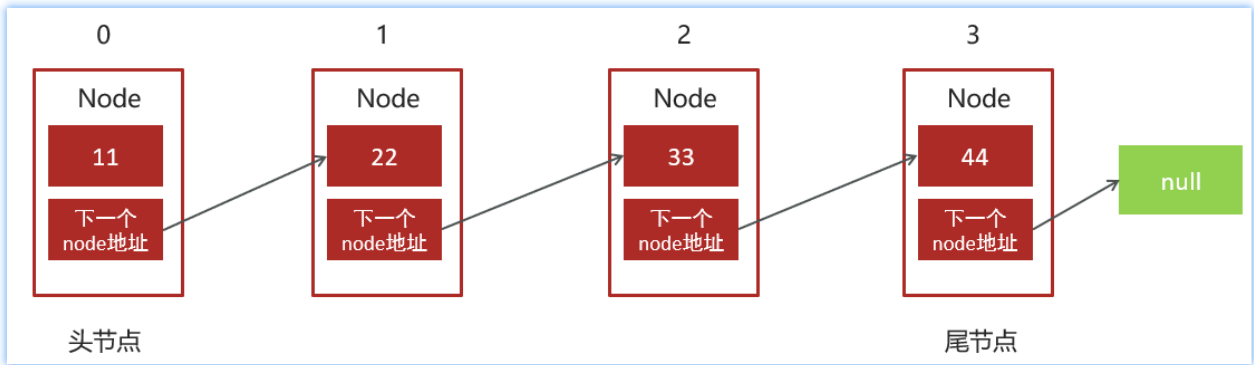
## 3.删除 $O(n)$

同理可得：如果我们要删除第  $k$  个位置的数据，为了内存的连续性，也需要搬移数据，不然中间就会出现空洞，内存就不连续了，时间复杂度仍然是 $O(n)$ 。

# 1.3 链表

## 1.3.1 链表概述

链表（**Linked list**）是一种物理存储单元上非连续、非顺序的存储结构，链表中的每一个元素称之为结点（**Node**），结点之间用指针（引用）连接起来，指针的指向顺序代表了结点的逻辑顺序，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

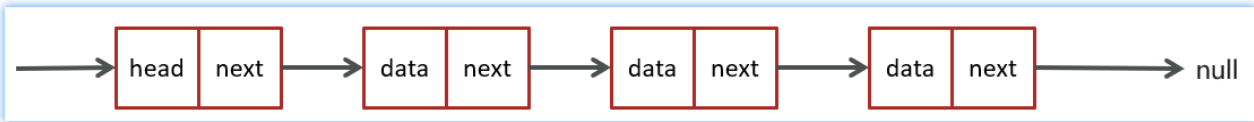


## 1.3.2 链表分类

- 单链表
- 双向链表
- 循环链表

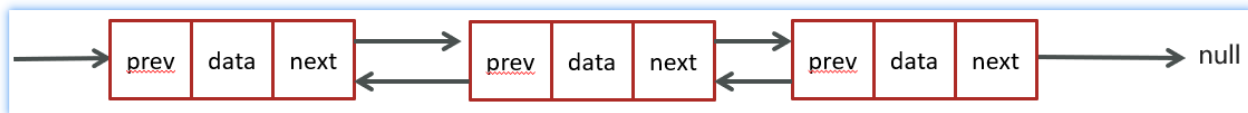
## 1.3.3 单链表

单链表就是我们刚刚讲到的链表的最基本的结构，链表通过指针将一组零散的内存块串联在一起。。如图所示，我们把这个记录下个结点地址的指针叫作后继指针 **next**，如果链表中的某个节点为 **p**，**p** 的下一个节点为 **q**，我们可以表示为：  
 $p.next=q$



### 1.3.4 双向链表

单向链表只有一个方向，结点只有一个后继指针 `next`。而双向链表，顾名思义，它支持两个方向，每个结点不止有一个后继指针 `next` 指向后面的结点，还有一个前驱指针 `prev` 指向前面的结点，如图所示

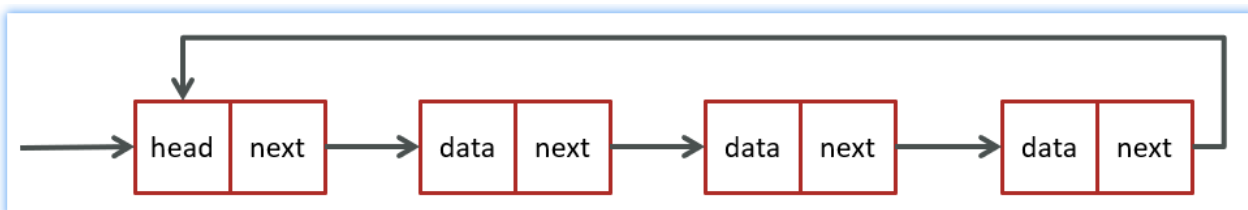


特点：

- 双向链表需要额外的两个空间来存储后继结点和前驱结点的地址
- 支持双向遍历，这样也带来了双向链表操作的灵活性
- 可以在 $O(1)$ 时间内找到给定结点的前驱节点，而对于单链表则需要 $O(n)$ 的时间
- 根据索引来查找元素时可极大提升查找效率

### 1.3.5 循环链表

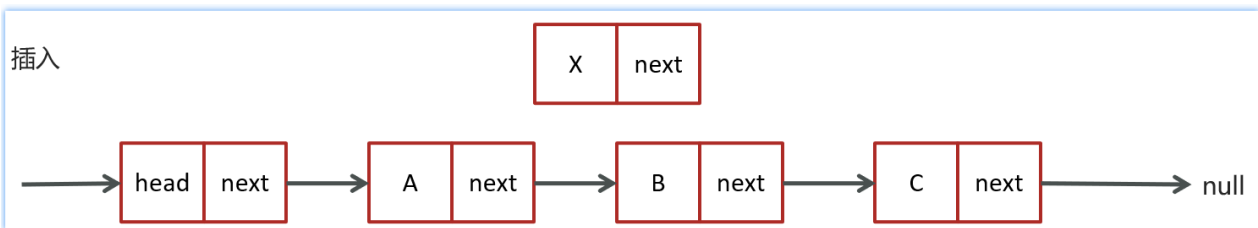
循环链表是一种特殊的单链表。循环链表的尾结点指针是指向链表的头结点。它像一个环一样首尾相连，所以叫作“循环”链表，和单链表相比，循环链表的优点是从链尾到链头比较方便。当要处理的数据具有环型结构特点时，就特别适合采用循环链表，循环链表的结构如图所示



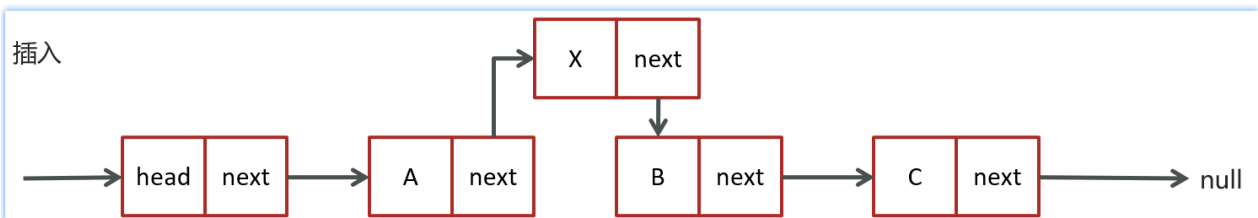
### 1.3.6 链表时间复杂度分析

针对链表的插入和删除操作，我们只需要考虑相邻结点的指针改变，所以插入删除的时间复杂度是  $O(1)$ 。

插入操作，解析X想要插入到AB节点之间

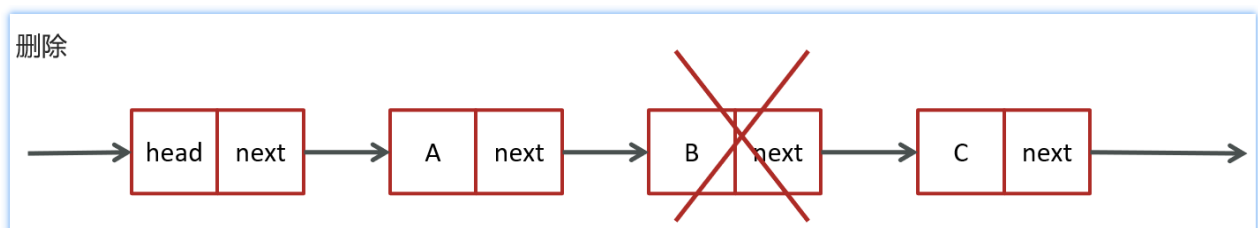


移动之后的效果，如下：

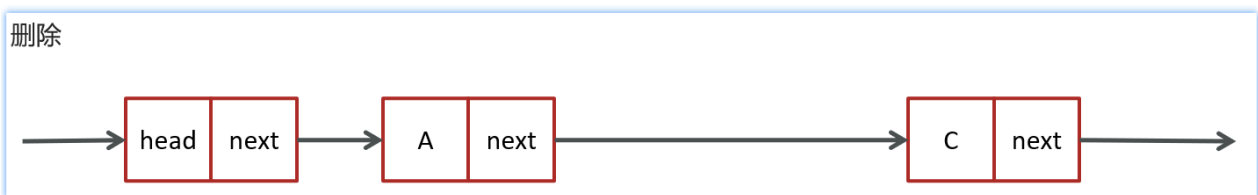


删除操作

想要删除B节点



删除之后的效果，如下：



查询的复杂度

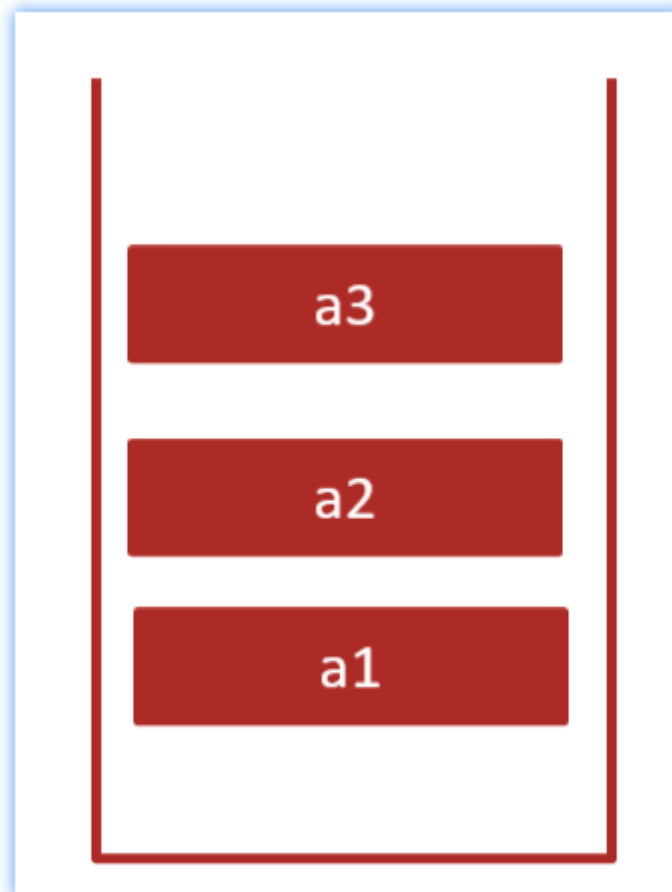
链表要想随机访问第  $k$  个元素，就没有数组那么高效了。因为链表中的数据并非连续存储的，所以无法像数组那样，根据首地址和下标，通过寻址公式就能直接计算出对应的内存地址，而是需要根据指针一个结点一个结点地依次遍历，直到找到相应的结点，所以，链表随机访问的性能没有数组好，查询的时间复杂度是  $O(n)$ 。



## 1.4 栈和队列

### 1.4.1 栈

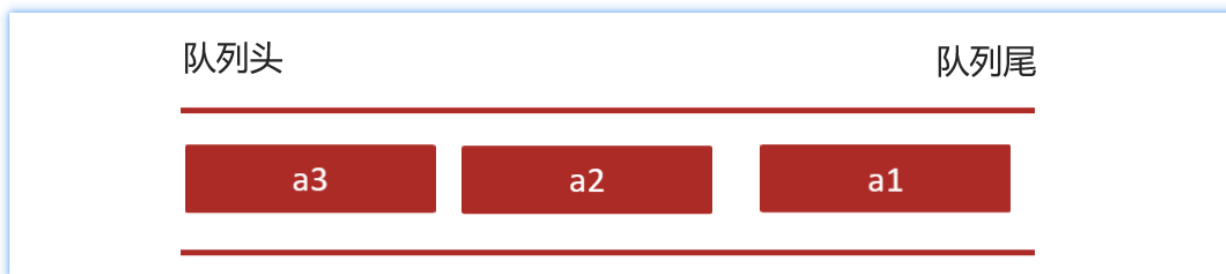
栈（Stack）并非指某种特定的数据结构，它是有着相同典型特征的一类数据结构的统称，因为栈可以用数组实现，也可以用链表实现。该典型特征是：后进先出；英文表示为：**Last In First Out**即 **LIFO**，只要满足这种特点的数据结构我们就可以说这是栈，为了更好的理解栈这种数据结构，我们以一幅图的形式来表示，如下：



时间复杂度均为 $O(1)$

### 1.4.1 队列

队列（Queue）和栈一样，代表具有某一类操作特征的数据结构，队列先进先出的特点英文表示为：**First In First Out**即**FIFO**，为了更好的理解队列这种数据结构，我们以一幅图的形式来表示，如下：



入队列和出队列操作的时间复杂度均为 $O(1)$

## 1.5 散列表

### 1.5.1 散列表（Hash Table）

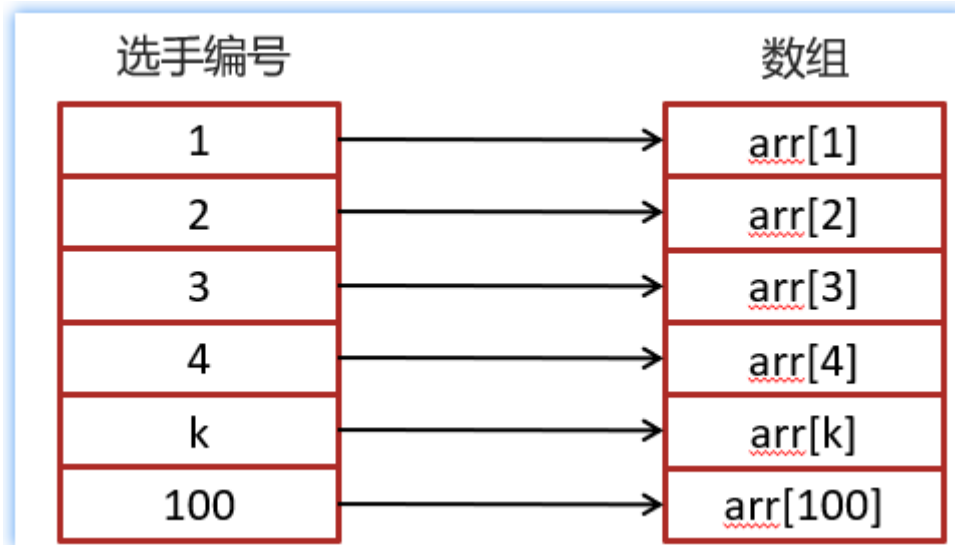
散列表(Hash Table)又名哈希表/Hash表，是根据键（Key）直接访问在内存存储位置值（Value）的数据结构，它是由数组演化而来的，利用了数组支持按照下标进行随机访问数据的特性

例子1:





假设有100个人参加马拉松，编号是1-100，如果要编程实现根据选手的编号迅速找到选手信息？

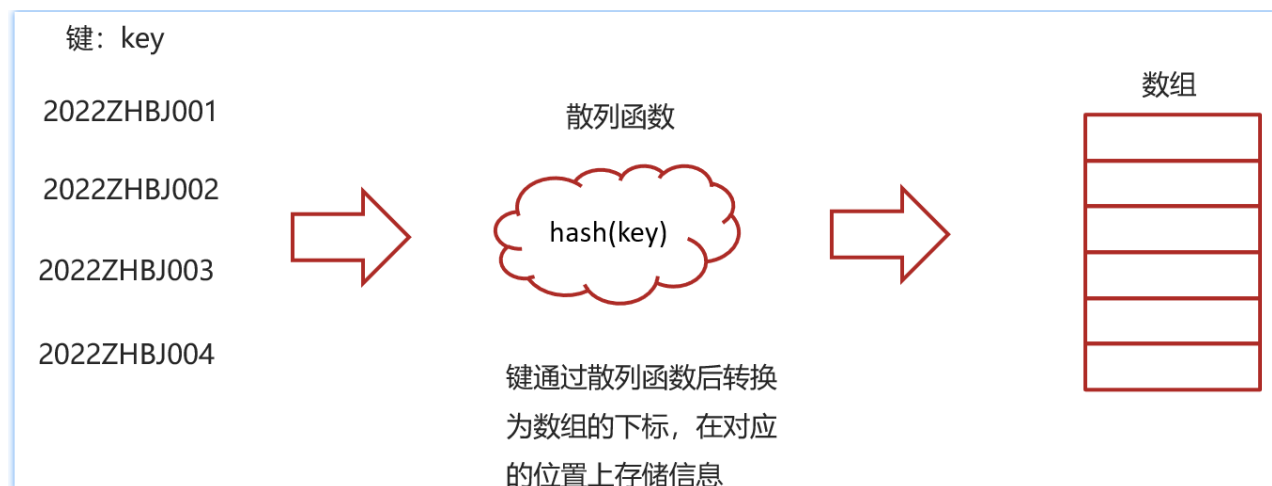


#### 实现思路：

我们可以让选手的编号存储到数组中，其中选手的编号就是数组的下标，我们需要查询某一个选手的时候，只需要根据选手编号作为数组下标查询即可，时间复杂度为 $O(1)$ ，效率很高

#### 例子2：

假设有100个人参加马拉松，不采用1-100的自然数对选手进行编号，编号有一定的规则比如：2022ZHBj001，其中2022代表年份，ZH代表中国，BJ代表北京，001代表原来的编号，那此时的编号2022ZHBj001不能直接作为数组的下标，此时应该如何实现呢？



我们让选手的编号（2022ZHBj001）进行hash计算，hash计算后得到一个整数值，可以做为数组的下标进行存储。

## 1.5.2 散列函数

将键(key)映射为数组下标的函数叫做散列函数。可以表示为：**hashValue = hash(key)**

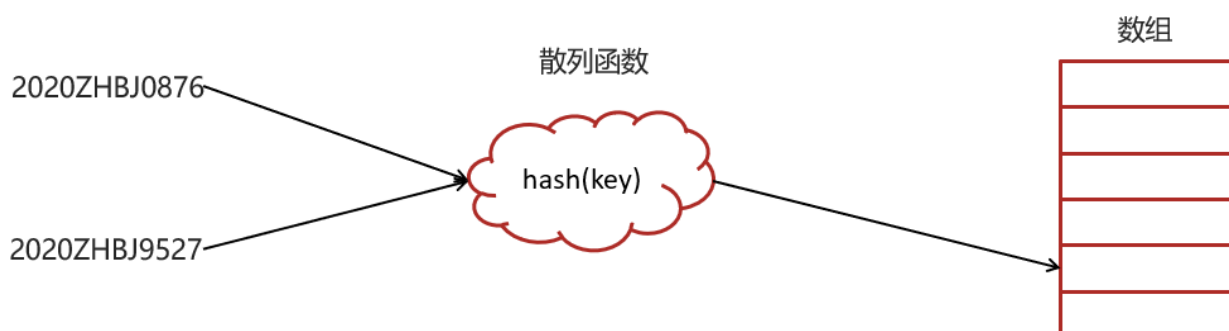
散列函数的基本要求：

- 散列函数计算得到的散列值必须是大于等于0的正整数，因为hashValue需要作为数组的下标。
- 如果 $key1 == key2$ ，那么经过hash后得到的哈希值也必相同即： $hash(key1) == hash(key2)$
- 如果 $key1 \neq key2$ ，那么经过hash后得到的哈希值也必不相同即： $hash(key1) \neq hash(key2)$

上面要求的前2条很容易理解，也很容易实现，但是第三条是不太容易实现的，因而会产生散列冲突

## 1.5.3 散列冲突

第三个要求看起来没有任何问题，但是在实际的情况下想找一个散列函数能够做到对于不同的key计算得到的散列值都不同几乎是不可能的，即便像著名的MD5,SHA等哈希算法也无法避免这一情况，这也就是我们即将要说到的散列冲突(或者哈希冲突，哈希碰撞，就是指多个key映射到同一个数组下标位置)

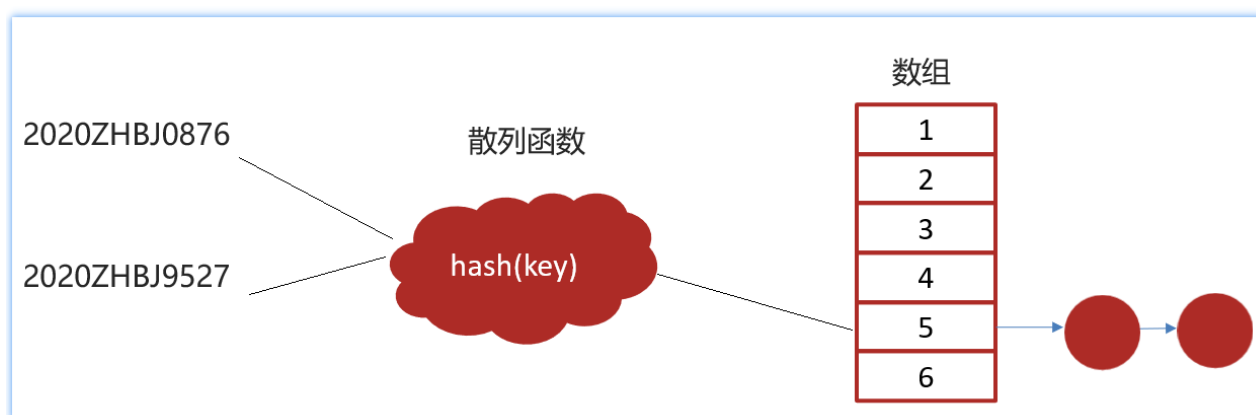


如上图所示

上图中的选手编号，如果进行hash计算后得到的值是一样的，则会计算到数组中的同一个下标中

### 1.5.4 散列冲突-链表法（拉链）

在散列表中，数组的每个下标位置我们可以称之为桶（**bucket**）或者槽（**slot**），每个桶(槽)会对应一条链表，所有散列值相同的元素我们都放到相同槽位对应的链表中。



#### 时间复杂度

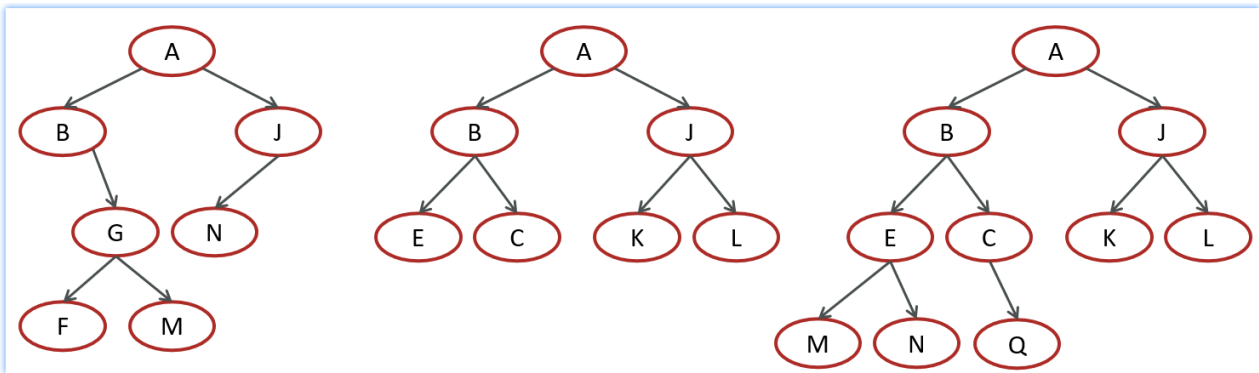
- 插入操作，通过散列函数计算出对应的散列槽位，将其插入到对应链表中即可，插入的时间复杂度是  $O(1)$
- 当查找、删除一个元素时，我们同样通过散列函数计算出对应的槽，然后遍历链表查找或者删除
  - 平均情况下基于链表法解决冲突时查询的时间复杂度是  $O(1)$
  - 如果散列函数设计得不好，或者装载因子过高，都可能导致散列冲突发生的概率升高，都散列到同一个槽里,散列表就会退化为链表,查询的时间复杂度就从  $O(1)$  急剧退化为  $O(n)$
  - 如果链表法稍加改造，可以实现一个更加高效的散列表。我们将链表法中的链表改造为其他高效的动态数据结构，比如跳表、红黑树，查询的时间复杂度也只不过是  $O(\log n)$

# 1.6 二叉树

## 1.6.1 二叉树

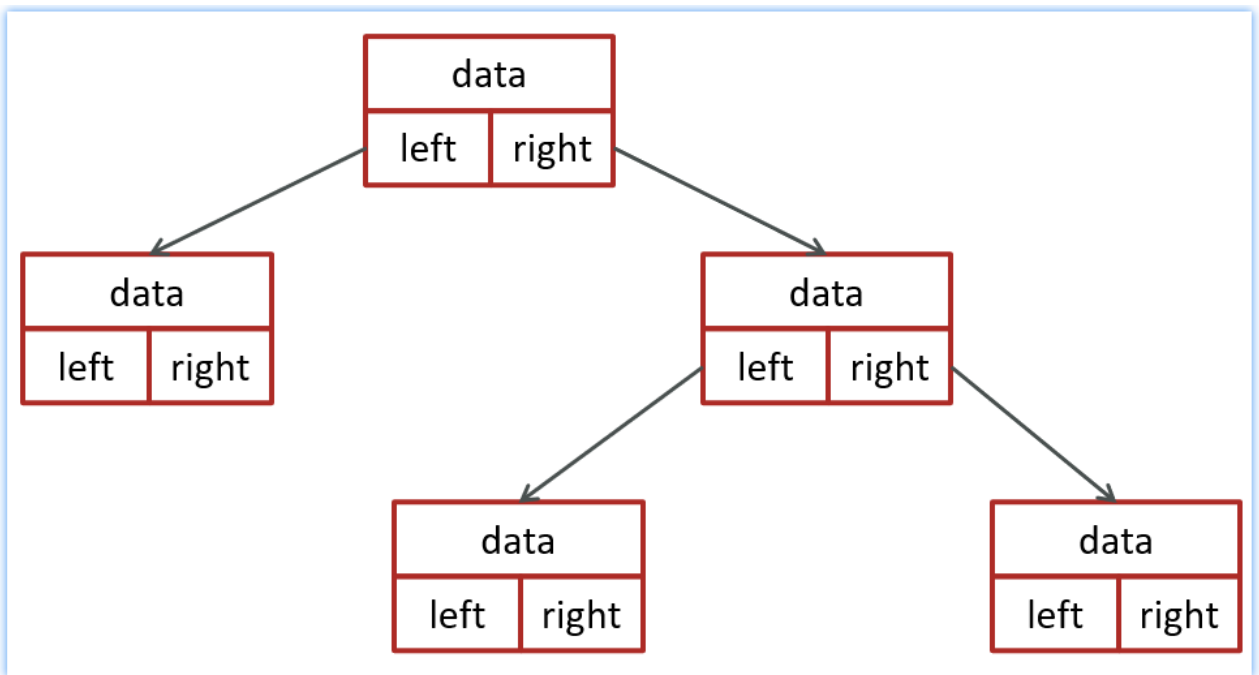
二叉树，顾名思义，每个节点最多有两个“叉”，也就是两个子节点，分别是左子节点和右子节点。不过，二叉树并不要求每个节点都有两个子节点，有的节点只有左子节点，有的节点只有右子节点。

二叉树每个节点的左子树和右子树也分别满足二叉树的定义。



很多其他高级数据结构都是基于二叉树，他们的操作特点各有不同，但从存储上来说底层无外乎就是两种：数组存储，链式存储。

基于链式存储的树的节点可定义如下：

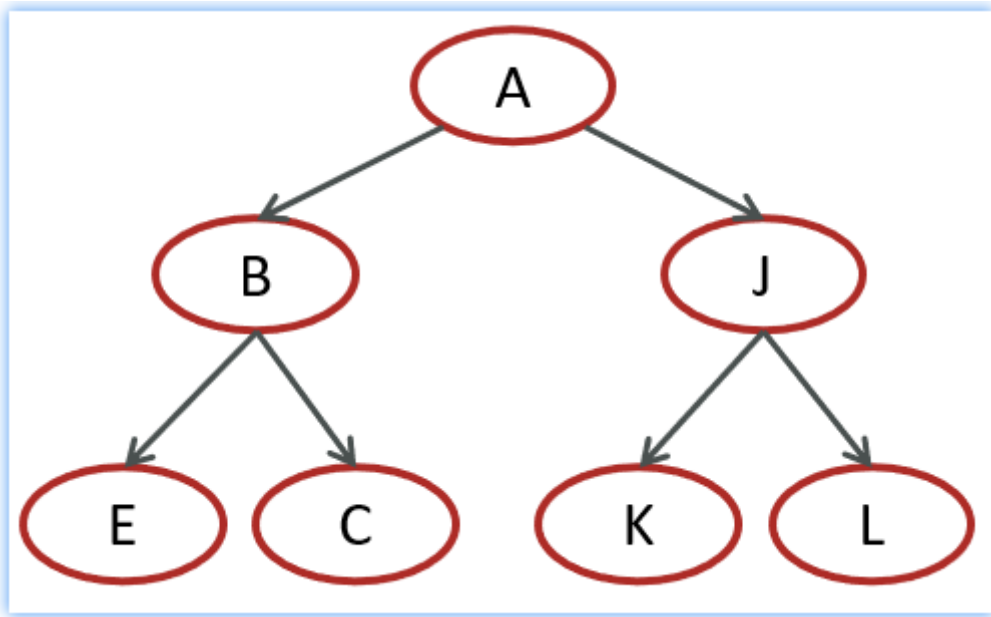


## 1.6.2 二叉树分类

在二叉树中，有几种特殊的情况，分别叫做：满二叉树，完全二叉树

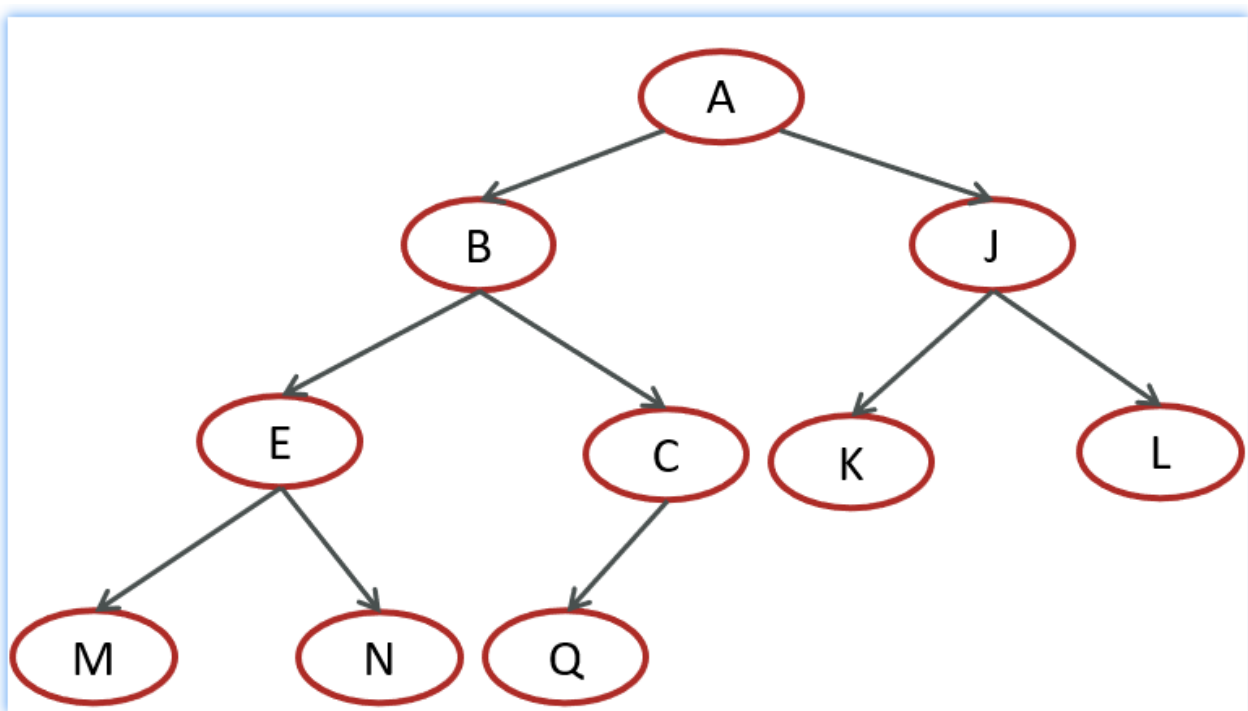
### 1、满二叉树

叶子节点全都在最底层，除了叶子节点之外，每个节点都有左右两个子节点，这种二叉树就叫作满二叉树。



### 2、完全二叉树

- 叶子节点都在最底下两层
- 最后一层的叶子节点都靠左排列(某个节点只有一个叶子节点的情况下)，
- 并且除了最后一层，其他层的节点个数都要达到最大，这种二叉树叫作完全二叉树。

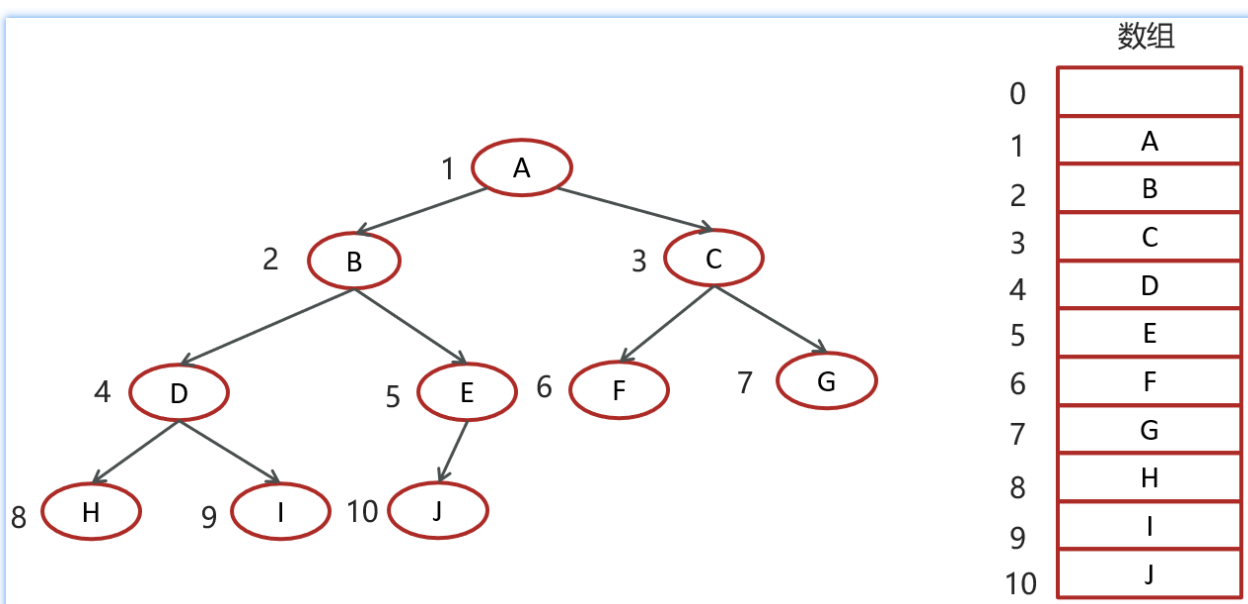


思考：为什么偏偏把最后一层的叶子节点靠左排列的叫完全二叉树？

完全二叉树更倾向采用基于数组的顺序存储方式，特征如下：

任意一个节点在数组下标为 $k$ 的位置

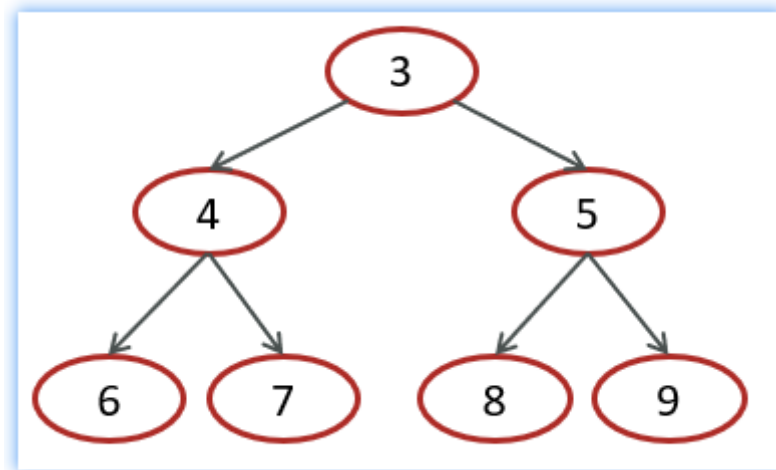
- 下标 $2*k$ 的位置存储的它的左子节点
- 下标 $2*k + 1$ 的位置存储的它的右子节点



数组存储完全二叉树能够有效利用存储空间

### 1.6.3 二叉树遍历

经典的三种遍历方式：前序遍历，中序遍历，后序遍历



- 前序遍历：对于树中的任意节点来说，先打印这个节点，然后再打印它的左子树，最后打印它的右子树。

遍历结果：3467589

- 中序遍历：对于树中的任意节点来说，先打印它的左子树，然后再打印它本身，最后打印它的右子树。

遍历结果：6473859

- 后序遍历：对于树中的任意节点来说，先打印它的左子树，然后再打印它的右子树，最后打印这个节点本身。

遍历结果：6748953

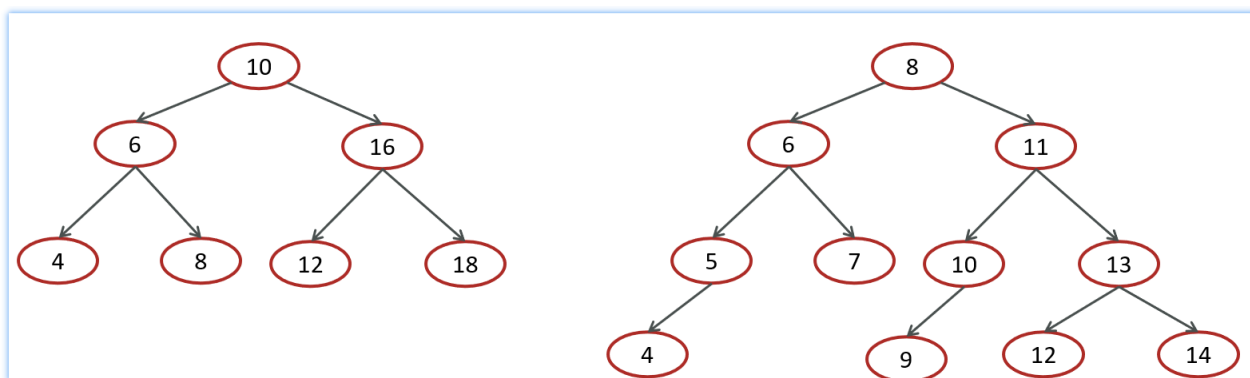
总结：

通过我们分析的二叉树的遍历流程我们可以发现，遍历二叉树的时间复杂度跟二叉树节点的个数 $n$ 成正比，因此，二叉树遍历的时间复杂度是 $O(n)$ 。

### 1.6.4 二叉搜索树

二叉搜索树(Binary Search Tree,BST)又名二叉查找树，有序二叉树或者排序二叉树，是二叉树中比较常用的一种类型

二叉查找树要求，在树中的任意一个节点，其左子树中的每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值



详细可以分为以下4点：

1. 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
3. 任意节点的左、右子树也分别为二叉查找树；
4. 没有键值相等的节点。

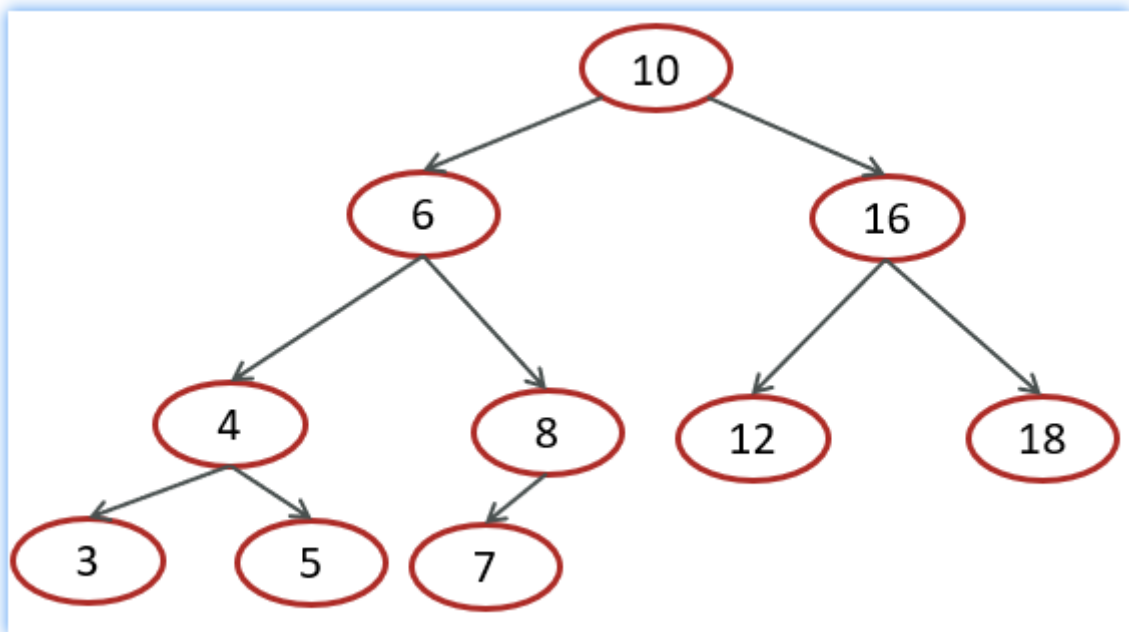
对于二叉查找树而言，它的中序遍历结果是一个递增的序列

### 1.6.5 二叉搜索树-时间复杂度分析

实际上由于二叉查找树的形态各异，时间复杂度也不尽相同，我画了几棵树我们来看一下插入，查找，删除的时间复杂度

#### (1) 完全二叉查找树

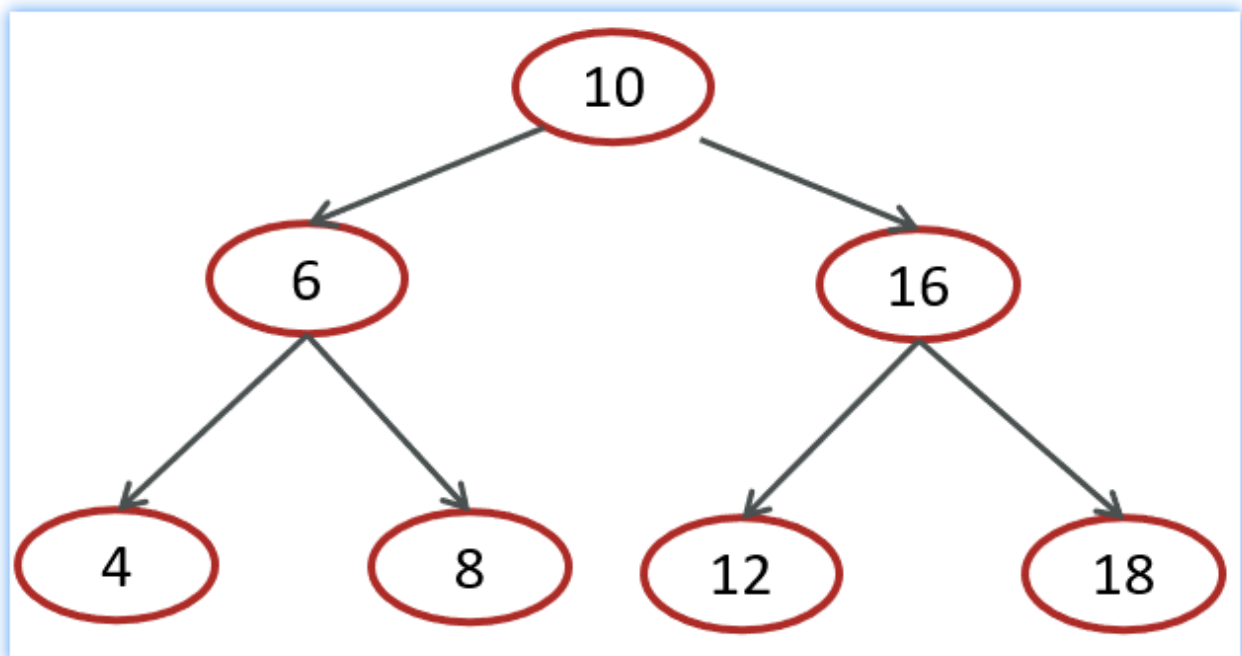




插入，查找，删除的时间复杂度其实和树的高度成正比，那也就是说时间复杂度为 $O(\text{height})$

**$O(\text{height}) == O(\log n)$**

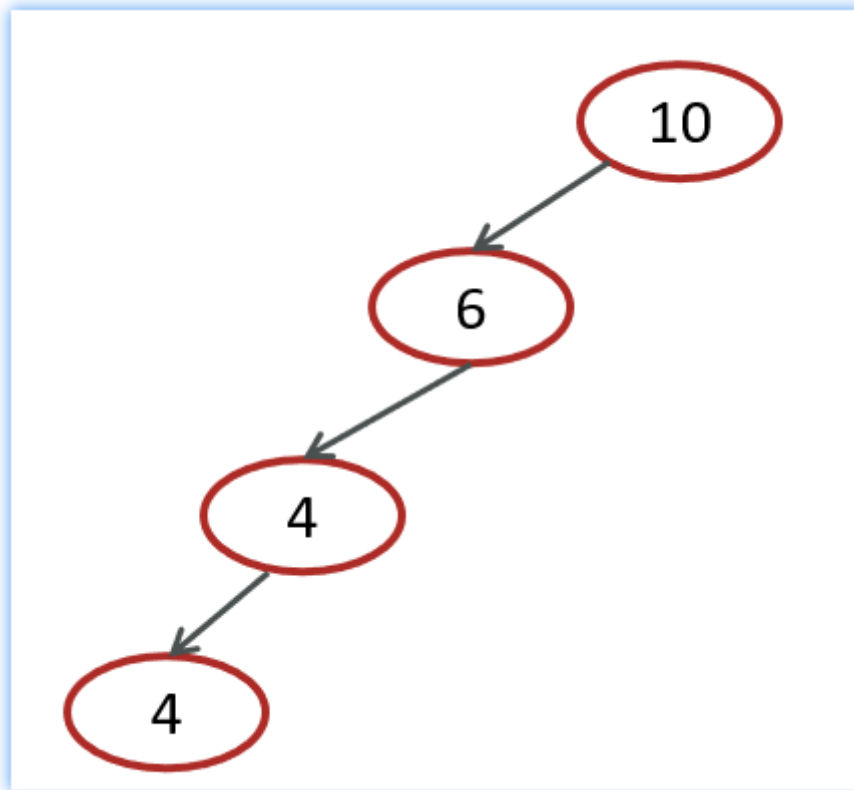
(2) 满二叉查找树



插入，查找，删除的时间复杂度其实和树的高度成正比，那也就是说时间复杂度为 $O(\text{height})$

**$O(\text{height}) == O(\log n)$**

### （3）极端情况



对于图中这种情况属于最坏的情况，二叉查找树已经退化成了链表，左右子树极度不平衡，此时查找的时间复杂度肯定是 $O(n)$ 。

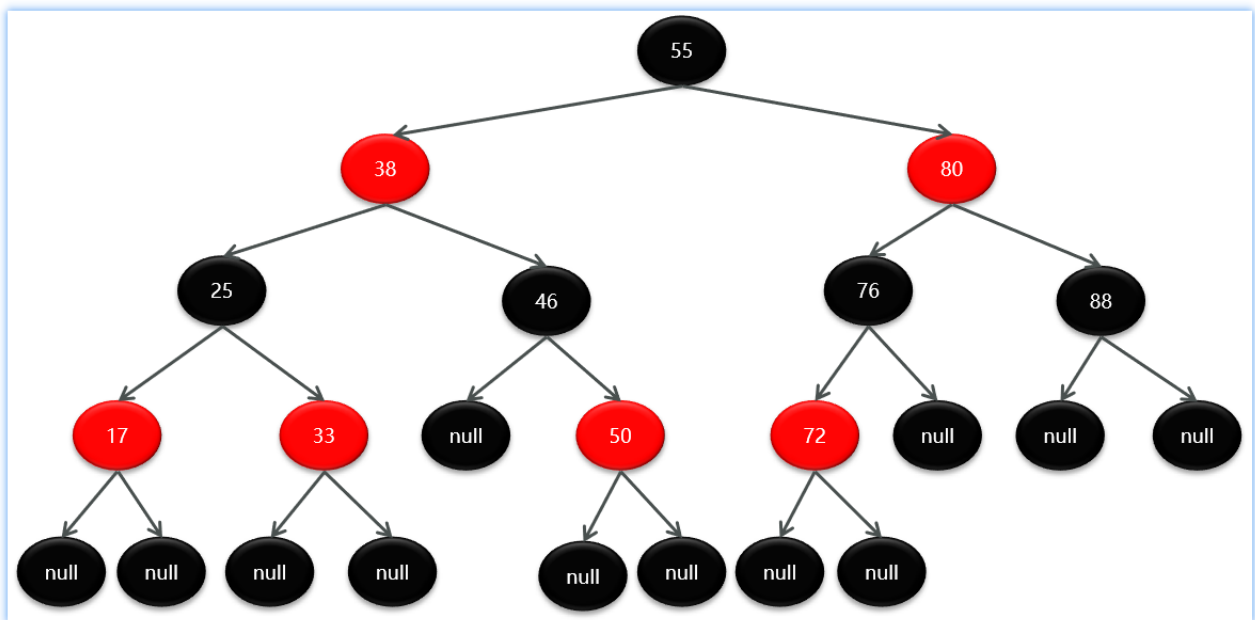
总结：

二叉查找树也可以叫做平衡二叉查找树。平衡二叉查找树的高度接近 $\log n$ ，所以插入、删除、查找操作的时间复杂度也比较稳定，是 $O(\log n)$ 。

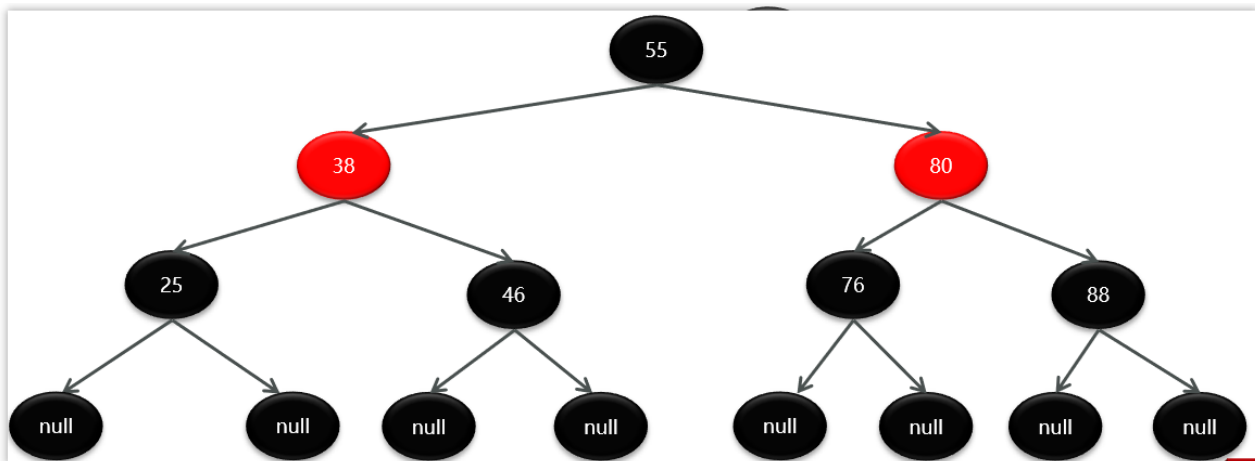
## 1.7 红黑树

### 1.7.1 红黑树-概述

红黑树（**Red Black Tree**）：也是一种自平衡的二叉搜索树(BST)，之前叫做平衡二叉B树（Symmetric Binary B-Tree）



### 1.7.2 红黑树的特质



性质1：节点要么是红色,要么是黑色

性质2：根节点是黑色

性质3：叶子节点都是黑色的空节点

性质4：红黑树中红色节点的子节点都是黑色

性质5：从任一节点到叶子节点的所有路径都包含相同数目的黑色节点

在这些规则的约束下，红黑树能够保证平衡

### 1.7.3 红黑树的复杂度

- 查找：

红黑树也是一棵BST（二叉搜索树）树，查找操作的时间复杂度为： $O(\log n)$

- 添加：

添加先要从根节点开始找到元素添加的位置，时间复杂度 $O(\log n)$

添加完成后涉及到复杂度为 $O(1)$ 的旋转调整操作

故整体复杂度为： $O(\log n)$

- 删除：

首先从根节点开始找到被删除元素的位置，时间复杂度 $O(\log n)$

删除完成后涉及到复杂度为 $O(1)$ 的旋转调整操作

故整体复杂度为： $O(\log n)$

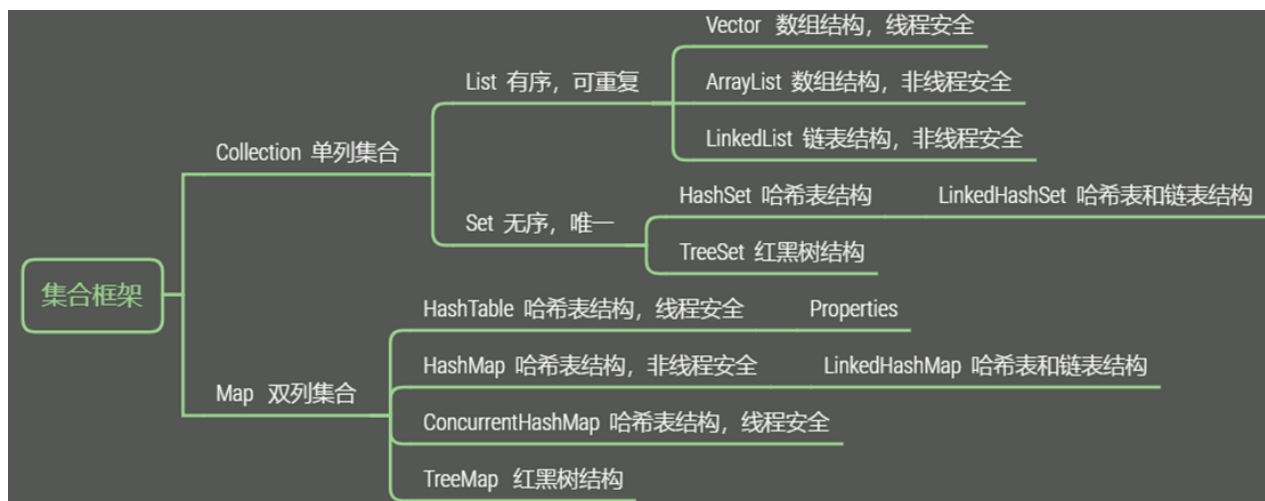
## 2 集合常见面试题

### 2.1 Java常见的集合类

#### 2.1.1 说一说（画一画）Collection结构图

难易程度：☆☆

出现频率：☆☆☆☆



Map接口和Collection接口是所有集合框架的父接口：

- 1.Collection接口的子接口包括：Set接口和List接口
- 2.Set接口的实现类主要有：HashSet、TreeSet、LinkedHashSet等
- 3.List接口的实现类主要有：ArrayList、LinkedList、Stack以及Vector等
- 4.Map接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap以及 Properties等

## 2.2 ArrayList

ArrayList 底层实现是数组，是动态数组。与Java中的数组相比，它的容量能动态增长。

它的算法时间复杂度与数组是一致的。

- 查询的复杂度为：O(1)
- 新增和删除的复杂度为：O(n)

### 2.2.1 ArrayList list=new ArrayList(10)中的list扩容几次

难易程度：☆☆

出现频率：☆☆

```
/**  
 * 构造一个具有指定初始容量的空列表。
```

```

* 参数: initialCapacity - 列表的初始容量
* 抛出: IllegalArgumentException - 如果指定的初始容量为负
*/
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
            initialCapacity);
    }
}
}

```

该语句只是申明和实例了一个 ArrayList，指定了容量为 10，未扩容

## 2.2.2 如何实现数组和List之间的转换

难易程度：☆☆

出现频率：☆☆

```

//数组转 List , 使用 JDK 中 java.util.Arrays 工具类的 asList 方法
public static void testArray2List() {
    String[] strs = new String[]{"aaa", "bbb", "ccc"};
    List<String> list = Arrays.asList(strs);
    for (String s : list) {
        System.out.println(s);
    }
}

//List 转数组
public static void testList2Array() {
    List<String> list = Arrays.asList("aaa", "bbb", "ccc");
    String[] array = list.toArray(new String[list.size()]);
    for (String s : array) {
        System.out.println(s);
    }
}
}

```

- 数组转 List，使用 JDK 中 `java.util.Arrays` 工具类的 `asList` 方法
- List 转数组，使用 List 的 `toArray` 方法。无参 `toArray` 方法返回 Object 数组，传入初始化长度的数组对象，返回该对象数组

## 2.3 LinkedList

底层是双向链表实现的List

- 非线程安全的
- 元素允许为null，允许重复元素
- 实现了栈和队列的操作方法，因此也可以作为栈、队列和双端队列来使用
- 时间复杂度：

因此插入删除效率高，复杂度为 $O(1)$ ，查找效率低，复杂度为： $O(n)$

### 2.3.1 ArrayList 和 LinkedList 的区别是什么？

难易程度：☆☆☆

出现频率：☆☆☆☆☆

参考回答：

1. 数据结构实现：ArrayList 是动态数组的数据结构实现，而 LinkedList 是双向链表的数据结构实现。
2. 随机访问效率：ArrayList 比 LinkedList 在随机访问的时候效率要高，因为 LinkedList 是线性的数据存储方式，所以需要移动指针从前往后依次查找。
3. 增加和删除效率：在非首尾的增加和删除操作，LinkedList 要比 ArrayList 效率要高，因为 ArrayList 增删操作要影响数组内的其他数据的下标。
4. 内存空间占用：LinkedList 比 ArrayList 更占内存，因为 LinkedList 的节点除了存储数据，还存储了两个引用，一个指向前一个元素，一个指向后一个元素。
5. 线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

## 2.4 HashMap

### 2.4.1 说一下HashMap的实现原理？

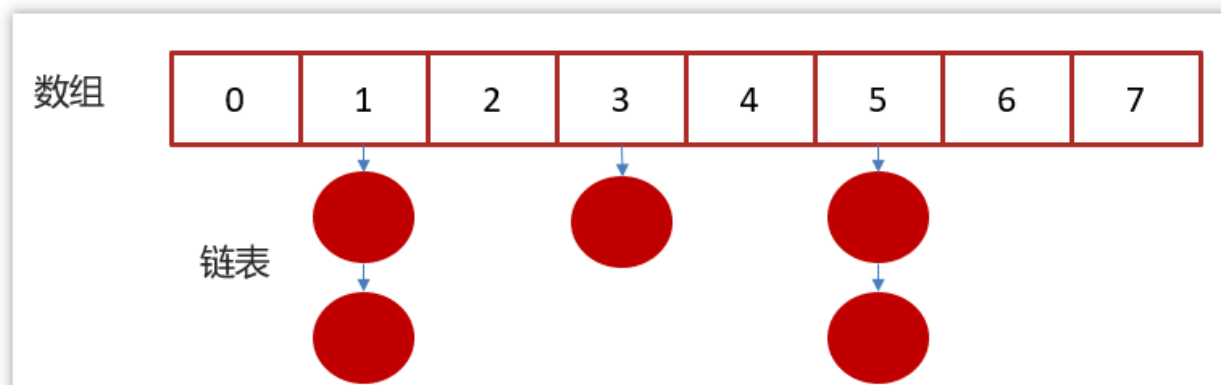
难易程度：☆☆☆

出现频率：☆☆☆☆☆

HashMap的数据结构：底层使用hash表数据结构，即数组和链表的结合体。

HashMap 基于 Hash 算法实现的

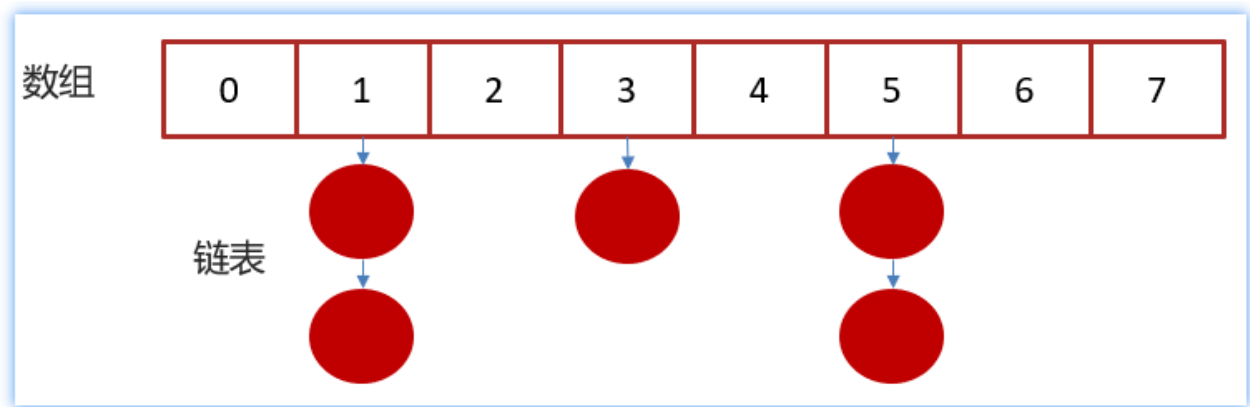
1. 当我们往HashMap中put元素时，利用key的hashCode重新hash计算出当前对象的元素在数组中的下标
2. 存储时，如果出现hash值相同的key，此时有两种情况。
  - 如果key相同，则覆盖原始值；
  - 如果key不同（出现冲突），则将当前的key-value放入链表中
3. 获取时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。



#### HashMap JDK1.8之前

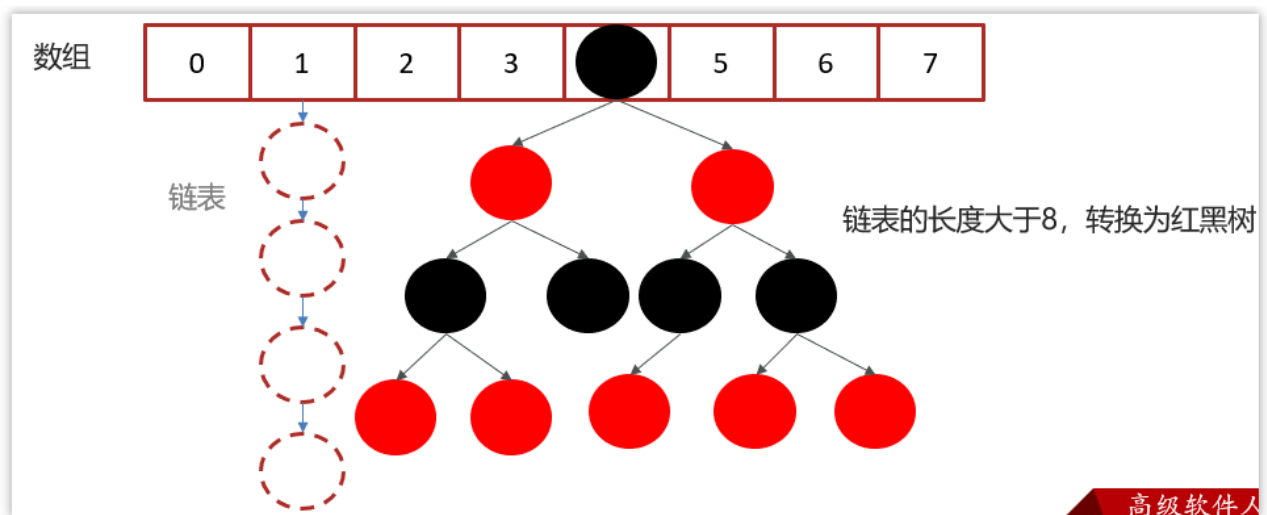
JDK1.8之前采用的是拉链法。拉链法：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。





## HashMap JDK1.8之后

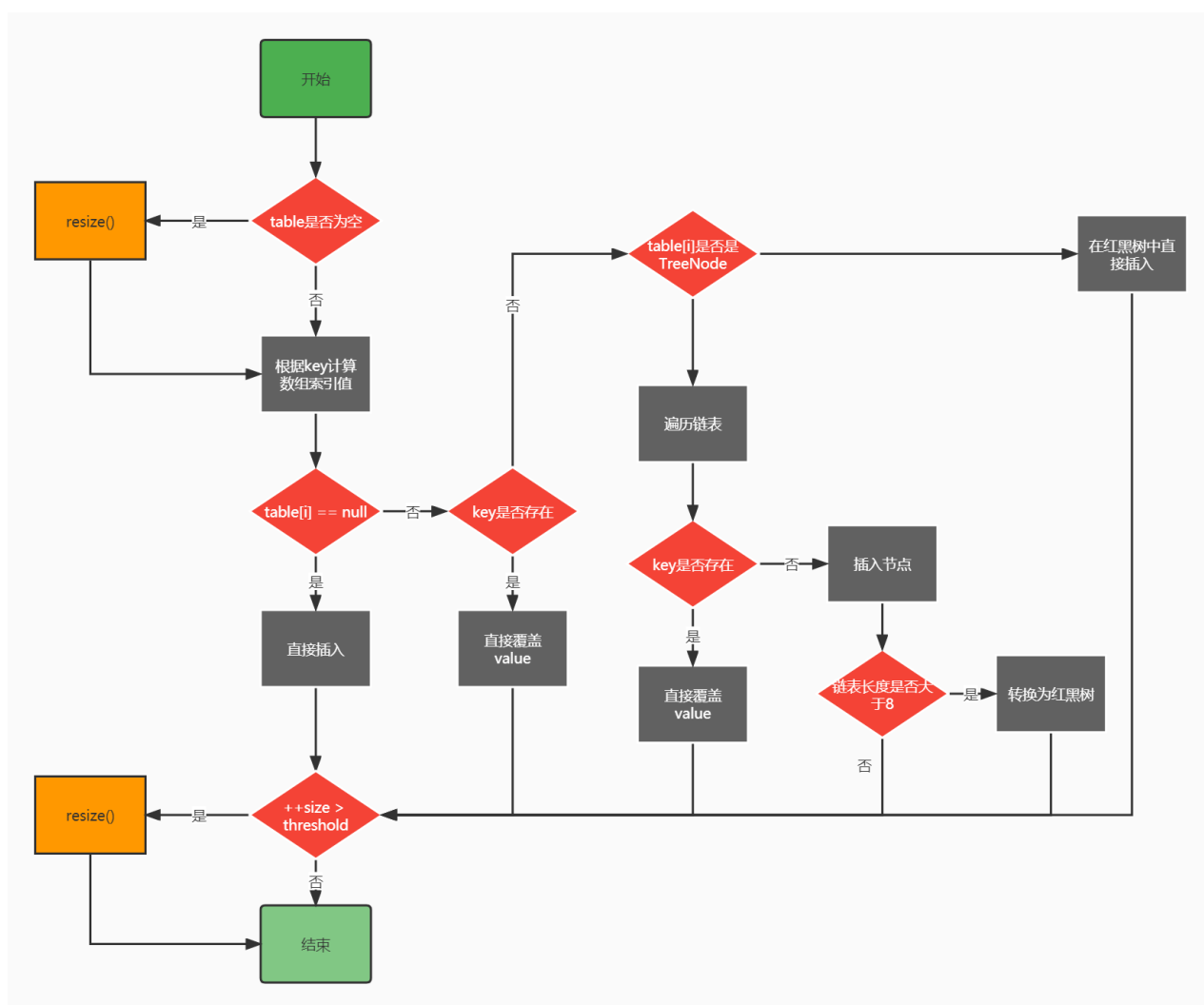
相比于之前的版本，jdk1.8在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。扩容 `resize()` 时，红黑树拆分成的树的结点数小于等于临界值6个，则退化成链表。



## 2.4.2 HashMap的put方法的具体流程？

难易程度：☆☆☆

出现频率：☆☆☆☆



参考回答：

1. 判断键值对数组`table[i]`是否为空或为`null`，否则执行`resize()`进行扩容；
2. 根据键值`key`计算`hash`值得到插入的数组索引`i`，如果`table[i]==null`，直接新建节点添加，转向⑥，如果`table[i]`不为空，转向③；
3. 判断`table[i]`的首个元素是否和`key`一样，如果相同直接覆盖`value`，否则转向④，这里的相同指的是`hashCode`以及`equals`；
4. 判断`table[i]` 是否为`treeNode`，即`table[i]` 是否是红黑树，如果是红黑树，则直接在树中插入键值 对，否则转向⑤
5. 遍历`table[i]`，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现`key`已经存在直接覆盖`value`即可；
6. 插入成功后，判断实际存在的键值对数量`size`是否超多了最大容量`threshold`，如果超过，进行扩容。

## 2.4.3 hashMap的寻址算法

难易程度：☆☆☆☆

出现频率：☆☆☆☆

我们先研究下key的哈希值是如何计算出来的。key的哈希值是通过上述方法计算出来的。

这个哈希方法首先计算出key的hashCode赋值给h,然后与h无符号右移16位后的二进制进行按位异或得到最后的 hash值。计算过程如下所示：

```
static final int hash(Object key)
{
    int h;
    /*
        1) 如果key等于null:
            可以看到当key等于null的时候也是有哈希值的，返回的是0.
        2) 如果key不等于null:
            首先计算出key的hashCode赋值给h,然后与h无符号右移16位后
            的二进制进行按位异或得到最后的 hash值
    */
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>>
16);
}
```

在putVal函数中使用到了上述hash函数计算的哈希值：

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    . . . . .
    if ((p = tab[i = (n - 1) & hash]) == null)//这里的n表示数组长
度16
    . . . . .
}
```

计算过程如下所示：

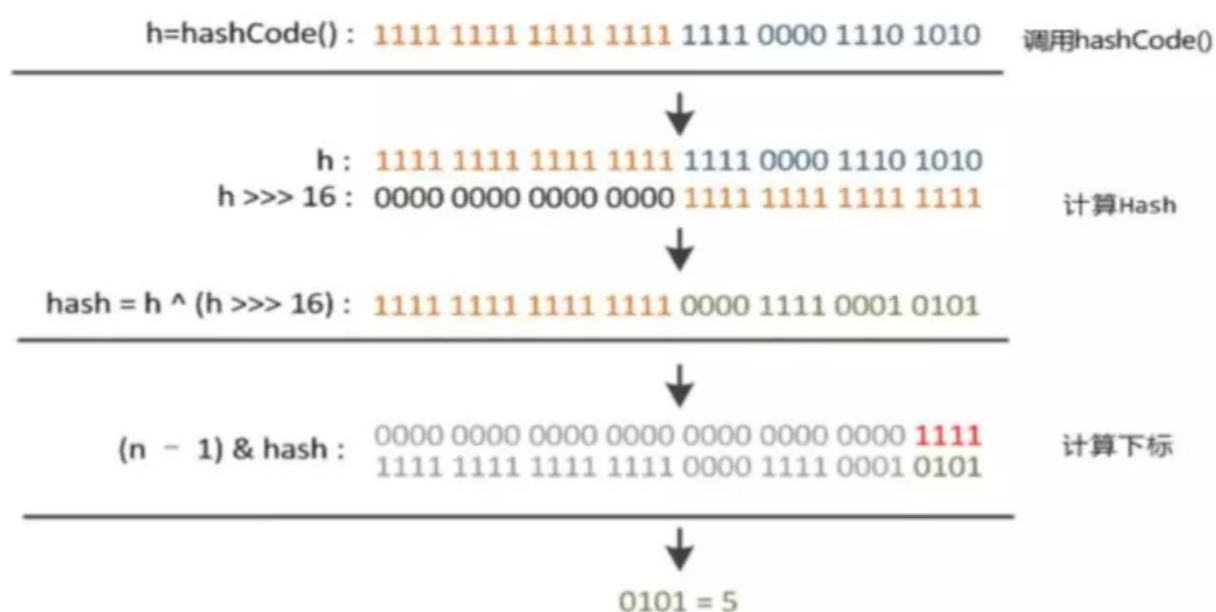
说明：

1) `key.hashCode()`: 返回散列值也就是hashcode。假设随便生成的一个值。

2) `n`表示数组初始化的长度是16

3) `&` (按位与运算): 运算规则: 相同的二进制数位上, 都是1的时候, 结果为1, 否则为零。

4) `^` (按位异或运算): 运算规则: 相同的二进制数位上, 数字相同, 结果为0, 不同为1。



## 2.4.4 讲一讲HashMap的扩容机制

难易程度: ☆☆☆

出现频率: ☆☆☆☆

参考回答:

1. 在jdk1.8中, `resize`方法是在hashmap中的键值对大于阈值(0.75)时或者初始化时, 就调用`resize`方法进行扩容;
2. 每次扩展的时候, 都是扩展2倍;
3. 扩展后Node对象的位置要么在原位置, 要么移动到原偏移量两倍的位置。

增强补充：

在put过程中，我们看到在这个函数里面使用到了2次resize()方法，resize()方法表示的在进行第一次初始化时会对其进行扩容，或者当该数组的实际大小大于其临界值(第一次为12)，这个时候在扩容的同时也会伴随的桶上面的元素进行重新分发，这也是JDK1.8版本的一个优化的地方

在1.7中，扩容之后需要重新去计算其Hash值，根据Hash值对其进行分发，但在1.8版本中，则是根据在同一个桶的位置中进行判断( $e.hash \& oldCap$ )是否为0，重新进行hash分配后，该元素的位置要么停留在原始位置，要么移动到原始位置+增加的数组大小这个位置上

## 2.4.5 为何HashMap的数组长度一定是2的次幂？

难易程度：☆☆☆

出现频率：☆☆☆

1. 计算索引时效率更高：如果是2的n次幂可以使用位与运算代替取模
2. 扩容时重新计算索引效率更高： $hash \& oldCap == 0$ 的元素留在原来位置，否则新位置 = 旧位置 + oldCap

## 2.4.6 hashmap在1.7情况下的多线程死循环问题

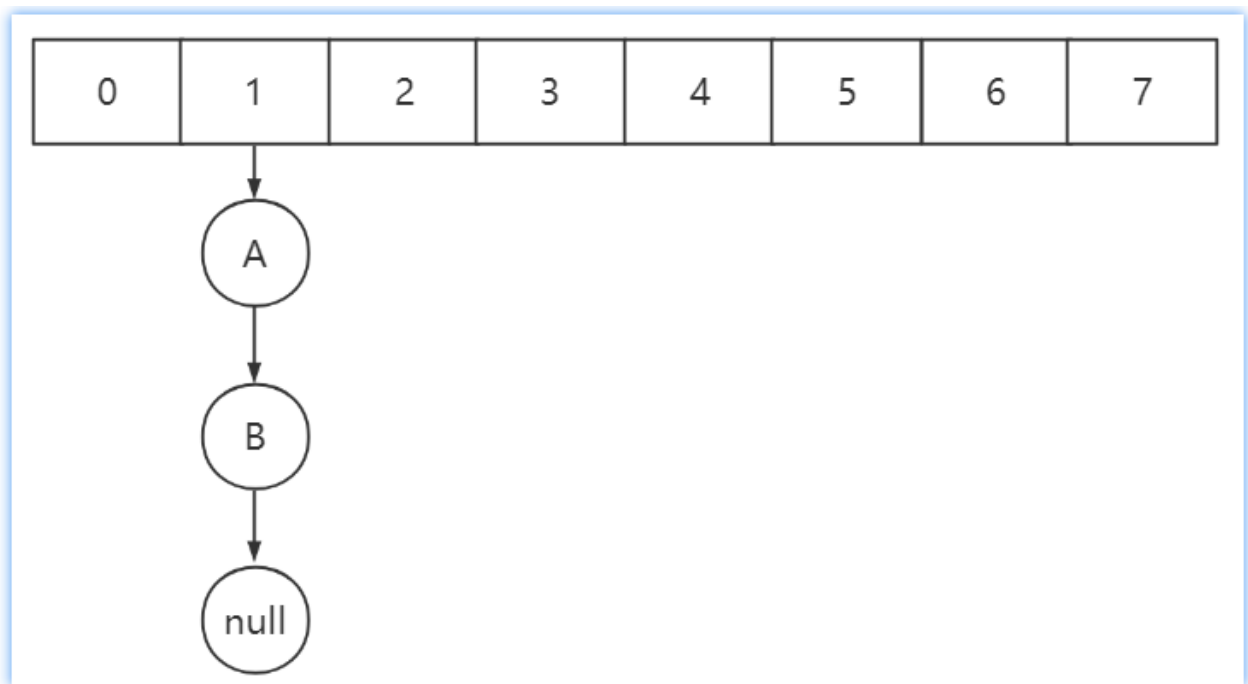
难易程度：☆☆☆

出现频率：☆☆

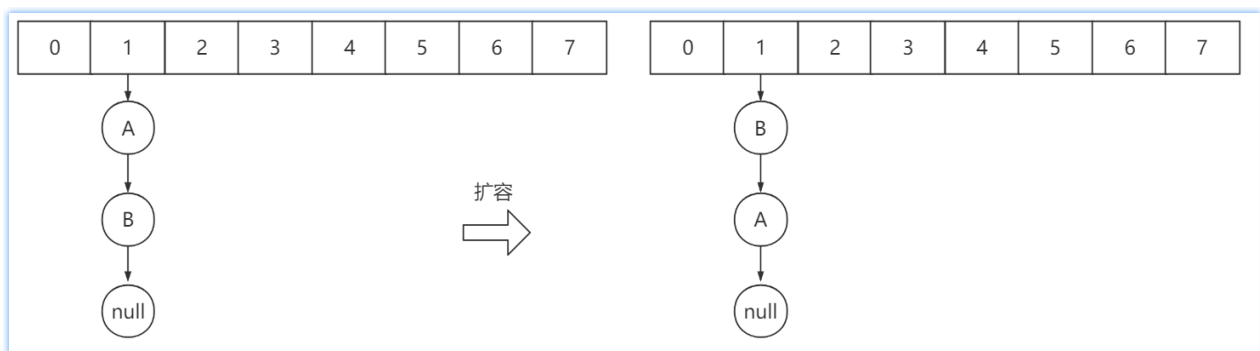
jdk7的的数据结构是：数组+链表

在数组进行扩容的时候，因为链表是头插法，在进行数据迁移的过程中，有可能导致死循环

线程一：读取到当前的hashmap情况，在准备扩容时，线程二介入

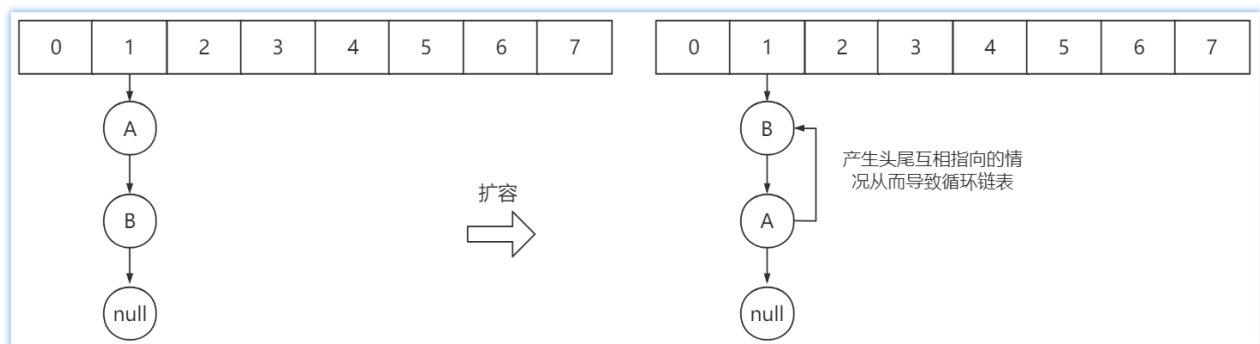


线程二：读取hashmap，进行扩容



线程一：继续执行

此线程先将A移入新的链表，再将B插入到链头，由于另外一个线程的原因，B的next指向了A，所以B->A->B,形成循环。



JDK 8 将扩容算法做了调整，不再将元素加入链表头（而是保持与扩容前一样的顺序），尾插法

## 2.5 HashSet

1. 不允许重复（底层是**HashMap**，用key储存元素，value统一都是PRESENT），可以为null，无顺序
2. HashSet就是为了提高查找效率的（在查找是否存在某个值时，ArrayList需要遍历才能确定某个值的位置，而HashSet可以通过HashCode快速定位）

### 2.5.1 HashSet与HashMap的区别

难易程度：☆☆

出现频率：☆☆

(1)HashSet实现了Set接口, 仅存储对象; HashMap实现了 Map接口, 存储的是键值对.

(2)HashSet底层其实是用HashMap实现存储的, HashSet封装了一系列HashMap的方法. 依靠HashMap来存储元素值,(利用hashMap的key键进行存储), 而value值默认为Object对象. 所以HashSet也不允许出现重复值, 判断标准和HashMap判断标准相同, 两个元素的hashCode相等并且通过equals()方法返回true.

## 3 真实面试还原

### 3.1 Java常见的集合类

面试官：说一说Java提供的常见集合？（画一下集合结构图）

候选人：

嗯~~，好的。

在java中提供了量大类的集合框架，主要分为两类：

第一个是Collection 属于单列集合，第二个是Map 属于双列集合

- 在Collection中有两个子接口List和Set。在我们平常开发的过程中用的比较多像list接口中的实现类ArrayList和LinkedList。在Set接口中有实现类HashSet和TreeSet。
- 在map接口中有很多的实现类，平时比较常见的是HashMap、TreeMap，还有一个线程安全的map:ConcurrentHashMap

## 3.2 ArrayList

面试官：ArrayList list=new ArrayList(10)中的list扩容几次

候选人：

是new了一个ArrayList并且给了一个构造参数10，对吧？(问题一定要问清楚再答)

面试官：是的

候选人：

好的，在ArrayList的源码中提供了一个带参数的构造方法，这个参数就是指定的集合初始长度，所以给了一个10的参数，就是指定了集合的初始长度是10，这里面并没有扩容。

---

面试官：如何实现数组和List之间的转换

候选人：

嗯，这个在我们平时开发很常见

数组转list，可以使用jdk自动的一个工具类Arrars，里面有一个asList方法可以转换为数组

List 转数组，可以直接调用list中的toArray方法，需要给一个参数，指定数组的类型，需要指定数组的长度。



### 3.3 LinkedList

面试官：ArrayList 和 LinkedList 的区别是什么？

候选人：

嗯，它们两个主要是底层使用的数据结构不一样，ArrayList 是动态数组，LinkedList 是双向链表，这也导致了它们很多不同的特点。

查询操作是ArrayList较快，它是根据数组下标获取的。LinkedList 由于是双向列表，需要移动指针从前往后依次查找，所以比较慢

增加和删除是LinkedList 效率更好，只需要挪动节点的指针即可；ArrayList 增删操作比较慢，它要影响数组内的其他数据的下标。

内存空间：也主要是因为数据结构不一样，LinkedList 比 ArrayList 更占内存

线程安全：ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全（看情况回答，一般回答这个问题，就会引出下一个问题，看下面）

面试官：嗯，好的，刚才你说了ArrayList 和 LinkedList 不是线程安全的，你们在项目中是如何解决这个的线程安全问题的？

候选人：

嗯，是这样的，主要有两种解决方案：

第一：我们使用这个集合，优先在方法内使用，定义为局部变量，这样的话，就不会出现线程安全问题。

第二：如果非要在成员变量中使用的话，可以使用线程安全的集合来替代

ArrayList可以通过Collections 的 synchronizedList 方法将 ArrayList 转换成线程安全的容器后再使用。

LinkedList 换成ConcurrentLinkedQueue来使用

## 3.4 HashMap

面试官：说一下HashMap的实现原理？

候选人：

嗯。它主要分为了一下几个部分：

当我们往HashMap中put元素时，利用key的hashCode重新hash计算出当前对象的元素在数组中的下标。计算key时，有可能会先两种情况，一是key之前是存在的，则覆盖原始值；第二是key不同则将当前的key-value放入链表中。

其实在它的底层实现中jdk1.7和1.8是不一样的。JDK1.8之前采用的是拉链法。拉链法：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。

jdk1.8在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时且数组长度大于64时，将链表转化为红黑树，以减少搜索时间。扩容时，红黑树拆分成的树的结点数小于等于临界值6个，则退化成链表。

后期使用map获取值时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。

面试官：好的，你能说下HashMap的put方法的具体流程吗？

候选人：

嗯好的。

在往hashmap中添加元素的时候。首先判断键值对数组是否为空或为null，否则执行扩容；根据键值key计算hash值得到插入的数组索引，如果数组为null，直接新建节点添加，插入成功后，判断实际存在的键值对数量size是否超多了最大容量，如果超过，进行扩容

如果数组不为空，判断数组的首个元素是否和key一样，如果相同直接覆盖value。

这个时候还要判断数组的下标元素是否是一个红黑树，如果是红黑树则需要向树中添加键值对

如果不是红黑树，则需要走链表的逻辑，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

面试官：好的，刚才你说的通过hash计算后找到数组的下标，是如何找到的呢，你了解HashMap的寻址算法吗？

候选人：

这个哈希方法首先计算出key的hashCode值，然后通过这个hash值右移16位后的二进制进行按位异或运算得到最后的hash值。

在putValue的方法中，计算数组下标的时候使用hash值与数组长度取模得到存储数据下标的位置，hashmap为了性能更好，并没有直接采用取模的方式，而是使用了数组长度-1得到一个值，用这个值按位与运算hash值，最终得到数组的位置。

面试官：好的，刚才你多次介绍了hsahmap的扩容，能讲一讲HashMap的扩容机制吗？

候选人：

好的，其实在jdk1.8中，resize方法是在hashmap中的键值对大于阈值(0.75)时或者初始化时，就调用resize方法进行扩容；每次扩展的时候，都是扩展2倍；扩展后Node对象的位置要么在原位置，要么移动到原偏移量两倍的位置。

面试官：为何HashMap的数组长度一定是2的次幂？

候选人：

嗯，好的。hashmap这么设计主要有两个原因：

第一：

计算索引时效率更高：如果是2的n次幂可以使用位与运算代替取模

第二：

扩容时重新计算索引效率更高：在进行扩容是会进行判断 hash值按位与运算 旧数组长度是否  $\neq 0$

如果等于0，则把元素留在原来位置，否则新位置是等于旧位置的下标+旧数组长度

面试官：好的，我看你对hashmap了解的挺深入的，你知道hashmap在1.7情况下的多线程死循环问题吗？

候选人：

嗯，知道的。是这样

jdk7的数据结构是：数组+链表

在数组进行扩容的时候，因为链表是头插法，在进行数据迁移的过程中，有可能导致死循环

比如说，现在有两个线程

线程一：读取到当前的hashmap数据，数据中一个链表，在准备扩容时，线程二介入

线程二也读取hashmap，直接进行扩容。因为是头插法，链表的顺序会进行颠倒过来。比如原来的顺序是AB，扩容后的顺序是BA，线程二执行结束。

当线程一再继续执行的时候就会出现死循环的问题。

线程一先将A移入新的链表，再将B插入到链头，由于另外一个线程的原因，B的next指向了A，所以B->A->B,形成循环。

当然，JDK 8 将扩容算法做了调整，不再将元素加入链表头（而是保持与扩容前一样的顺序），尾插法，就避免了jdk7中死循环的问题。

面试官：好的，hashmap是线程安全的吗？

候选人：不是线程安全的

面试官：那我们想要使用线程安全的map该怎么做呢？

候选人：我们可以采用ConcurrentHashMap进行使用，它是一个线程安全的HashMap

面试官：那你能聊一下ConcurrentHashMap的原理吗？

候选人：好的，请参考《多线程相关面试题》中的ConcurrentHashMap部分的讲解

---

面试官：HashSet与HashMap的区别？

候选人：嗯，是这样。

HashSet底层其实是用HashMap实现存储的, HashSet封装了一系列HashMap的方法. 依靠HashMap来存储元素值,(利用hashMap的key键进行存储), 而value值默认为Object对象. 所以HashSet也不允许出现重复值, 判断标准和HashMap判断标准相同, 两个元素的hashCode相等并且通过equals()方法返回true.