# HashMap/Table 和 Heap

## 理解 HashTable/Map

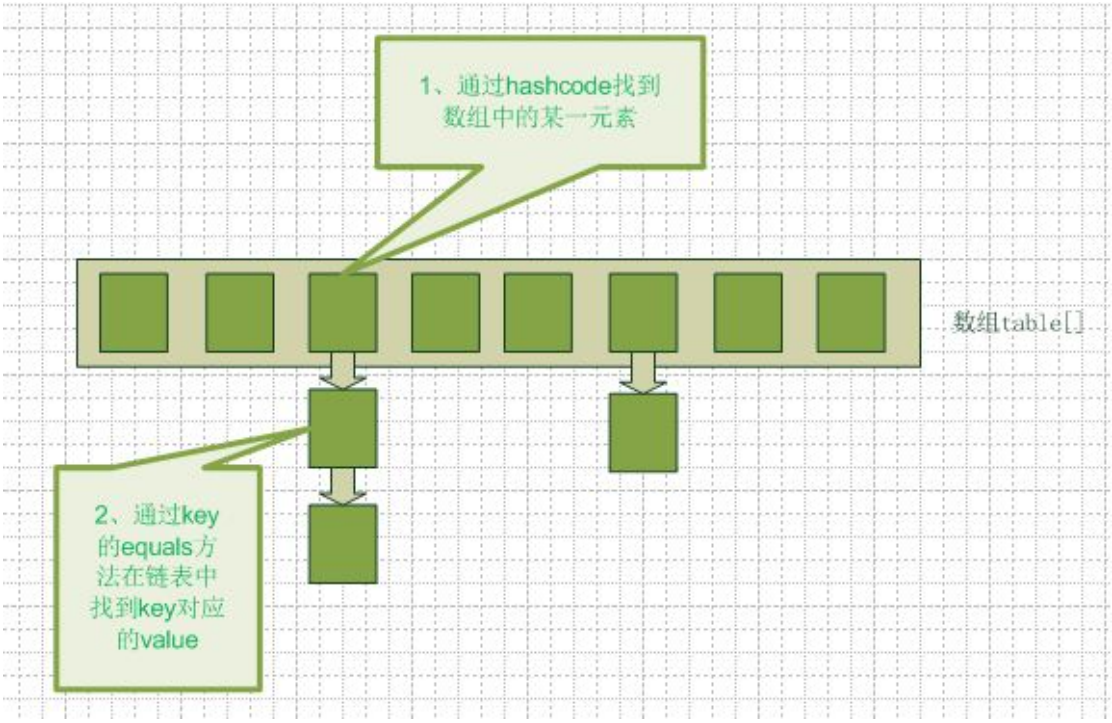
### 1. 结构

HashMap 和 HashTable 都使用哈希表来存储键值对。在数据结构上是基本相同的，都创建了一个继承自 Map.Entry 的私有的内部类 Entry，每一个 Entry 对象表示存储在哈希表中的一个键值对。

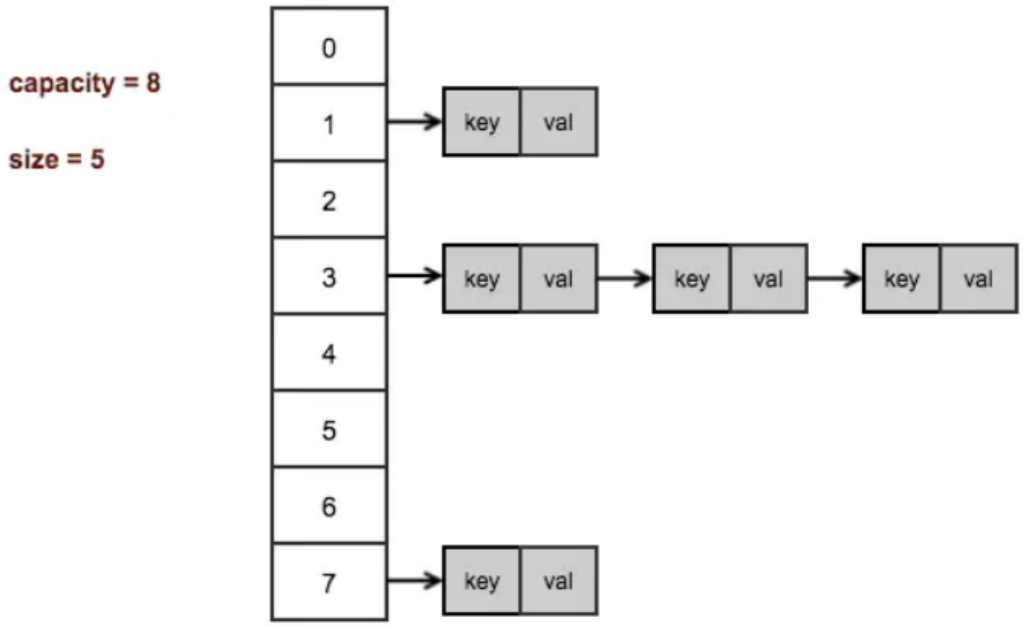
Entry 对象唯一表示一个键值对，有四个属性：

-K key 键对象
-V value 值对象
-int hash 键对象的 hash 值
-Entry entry 指向链表中下一个 Entry 对象，可为 null，表示当前 Entry 对象在链表尾部

如下图示：



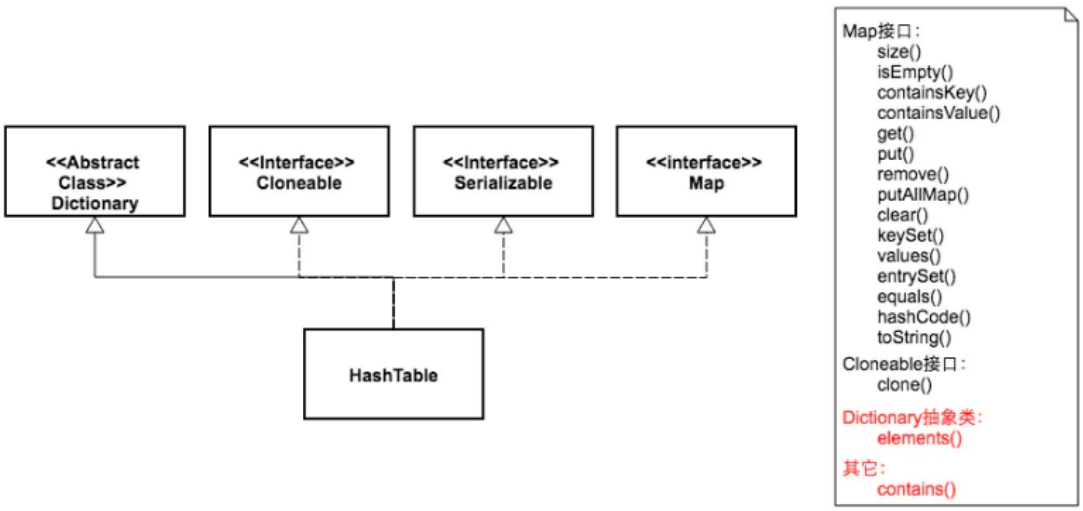

**在之前的作业中，有同学提出过相关的问题，所以这里我更深入地讲一下：**
Bucket 就是这个 table，当我们往 hashMap 或者 HashTable 中 put 元素的时候，先根据 key 的 hash 值得到这个元素在数组中的位置（即下标），然后就可以把这个元素放到对应的位置中了。

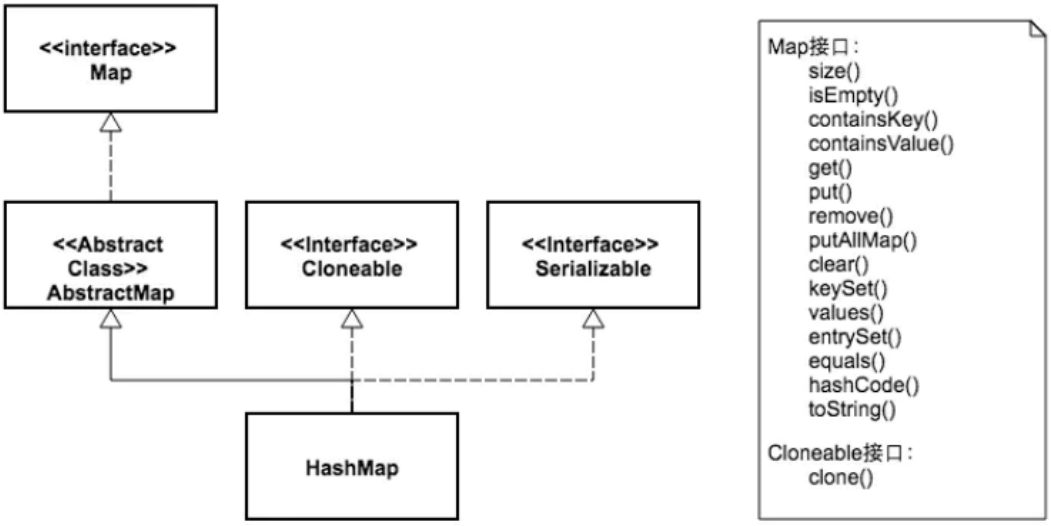



上图画出的是一个桶数量为 8，存有 5 个键值对的 HashMap/HashTable 的内存布局情况。可以看到 HashMap/HashTable 内部创建有一个 Entry 类型的引用数组，用来表示哈希表，数组的长度，即是哈希桶的数量。而数组的每一个元素都是一个 Entry 引用，从 Entry 对象的属性里，也可以看出其是链表的节点，每一个 Entry 对象内部又含有另一个 Entry 对象的引用。

## 2. HashMap 和 HashTable 的区别

1. 继承不同：
   Hash Table 继承了 Dictionary class，implements Map 接口
   Hash Map 继承了 AbstractMap class，implements Map 接口
   具体的图可以看这个：






可惜，Dictionary 类是一个已经被废弃的类（见其源码中的注释）。父类都被废弃，自然而然也没人用它的子类 Hash table 了。

2. 对于 put 的值的要求不同:
   Hashtable 既不支持 Null key 也不支持 Null value。Hashtable 的 put()方法的注释中有说明。HashTable 在遇到 null 时，会抛出 NullPointerException 异常

```
//以下代码及注释来自 java.util.HashTable

public synchronized V put(K key, V value) {

    // 如果 value 为 null，抛出 NullPointerException
    if (value == null) {

            throw new NullPointerException();
    }
    // 如果 key 为 null，在调用 key.hashCode()时抛出 NullPointerExceptio
n // ...

}
```

在 HashMap 中，null 可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为 null。当 get()方法返回 null 值时，即可以表示 HashMap 中没有该键，也可以表示该键所对应的值为 null。因此，在 HashMap 中不能由 get()方法来判断 HashMap 中是否存在某个键， 而应该用 containsKey()方法来判断。HashMap 在实现时对 null 做了特殊处理，将 null 的 hashCode 值定为了 0，从而将其存放在哈希表的第 0 个 bucket 中。我们以 put 方法为例，看一看代码的细节:

```
//以下代码及注释来自 java.util.HasMap

public V put(K key, V value) {
    if (table == EMPTY_TABLE) {
            inflateTable(threshold);
    }
    // 当 key 为 null 时，调用 putForNullKey 特殊处理

    if (key == null)
            return putForNullKey(value);
            // ...
 }

private V putForNullKey(V value) {
     // key 为 null 时，放到 table[0]也就是第 0 个 bucket 中
     for (Entry<K,V> e = table[0]; e != null; e = e.next) {
         if (e.key == null) {
           V oldValue = e.value; e.value = value;
           e.recordAccess(this);
           return oldValue;
         }
     }
     modCount++;
     addEntry(0, null, value, 0);
     return null;
}
```

从这里我们可以看出，对于 HashMap 来说，所有的 null value 都被放在了 table 的 index0 的位置，所有其他的 Entry 都是从 index 为 1 开始一个个往下找的

3. 初始的 capacity 和扩容的方式：
   HashTable 默认的初始大小为 11，之后每次扩充为原来的 2n+1。HashMap 默认的初始化大小为 16，之后每次扩充为原来的 2 倍。如果在创建时给定了初始化大小，那么 HashTable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小。也就是说 HashTable 会尽量使用素数、奇数。而 HashMap 则总是使用 2 的幂作为哈希表的大小。

   所以单从这一点上看，HashTable 的哈希表大小选择，似乎更高明些。但另一方面我们又知道，在取模计算时，如果模数是 2 的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从 hash 计算的效率上，又是 HashMap 更胜

一筹。

4. Hash 值的计算不同：
   HashTable 使用的是 hash function，我们可以直接看 java 自带的 jdk 代码：

```
//以下代码及注释来自 java.util.HashTable
// hash 不能超过 Integer.MAX_VALUE
//所以要取其最小的 31 个 bit int hash = hash(key);

int index = (hash & 0x7FFFFFFF) % tab.length;

// 直接计算 key.hashCode()
private int hash(Object k) {
    // hashSeed will be zero if alternative hashing is disabled.
    return hashSeed ^ k.hashCode();
}
```

   HashMap 为了加快 hash 的速度，将哈希表的大小固定为了 2 的幂。当然这引入了哈希分布不均匀的问题，所以 HashMap 为解决这问题，又对 hash 算法做了一些改动。HashMap 由于使用了 2 的幂次方，所以在取模运算时不需要做除法，只需要位的与运算就可以了。但是由于引入的 hash 冲突加剧问题，HashMap 在调用了对象的 hashCode 方法之后，又做了一些位运算在打散数据。我们可以看一下代码：

```
//以下代码及注释来自 java.util.HashMap

int hash = hash(key);
int i = indexFor(hash, table.length);

// 在计算了 key.hashCode()之后，做了一些位运算来减少哈希冲突

final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
            return sun.misc.Hashing.stringHash32((String) k);
    }
    h ^= k.hashCode();
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
```

```
    // number of collisions (approximately 8 at default load facto
r).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

// 取模不再需要做除法

static int indexFor(int h, int length) {

// assert Integer.bitCount(length) == 1 : "length must be a non-zer
o power of 2";
    return h & (length-1);
}
```

如果你有细心读代码，还可以发现一点，就是 HashMap 和 HashTable 在计算 hash 时都用到了一个叫 hashSeed 的变量。这是因为映射到同一个 hash 桶内的 Entry 对象，是以链表的形式存在的，而链表的查询效率比较低，所以 HashMap/HashTable 的效率对哈希冲突非常敏感，所以可以额外开启一个可选 hash（hashSeed），从而减少哈希冲突。但是这个优化在 JDK 1.8 中已经去掉了，因为 JDK 1.8 中，映射到同一个哈希桶（数组位置）的 Entry 对象在 LinkedList 的长度大于 8 的时候，使用了红黑树来存储。但是在 HashTable 中却是始终使用 LinkedList 来储存 hashCode 相等的那些 Entry 的

### 3. HashTable/Map 的作用和算法 example

以空间换时间复杂度，增加所需要的空间，但是大幅度地减小时间上的开销

**题目：**
**给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target 的那 两个 整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。**

**Example1：**
输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1]

**Example2：**
输入：nums = [3,2,4], target = 6
输出：[1,2]

**Example3：**

输入：nums = [3,3], target = 6

输出：[0,1]

**思路：**

最容易想到的方法是枚举数组中的每一个数 x，寻找数组中是否存在 target - x。当我们使用遍历整个数组的方式寻找 target - x 时，需要注意到每一个位于 x 之前的元素都已经和 x 匹配过，因此不需要再进行匹配。而每一个元素不能被使用两次，所以我们只需要在 x 后面的元素中寻找 target – x, 时间复杂度为 O(n^2)

```
public int[] twoSum(int[] nums, int target) {
        int n = nums.length;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i, j};
                }
            }
        }
        return new int[0];
}
```

1. 由于哈希查找的时间复杂度为 O(1)，所以可以利用哈希容器 map 降低时间复杂度

2. 遍历数组 nums，i 为当前下标，每个值都判断 map 中是否存在 target-nums[i] 的 key 值

3. 如果存在则找到了两个值，如果不存在则将当前的 (nums[i],i) 存入 map 中，继续遍历直到找到为止，如果最终都没有结果则抛出异常

```
public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> res = new HashMap<Integer, Integer>();
        for (int i = 0; i < nums.length; ++i) {
            if (res.containsKey(target - nums[i])) {
                return new int[]{res.get(target - nums[i]), i};
            }
            res.put(nums[i], i);
        }
        return new int[0];
```

```
}
```

4. **复杂度分析:**

   **时间复杂度：O(N)，其中 N 是数组中的元素数量。对于每一个元素 x，我们可以 O(1) 地寻找 target – x**

   **空间复杂度：O(N)，其中 N 是数组中的元素数量。**

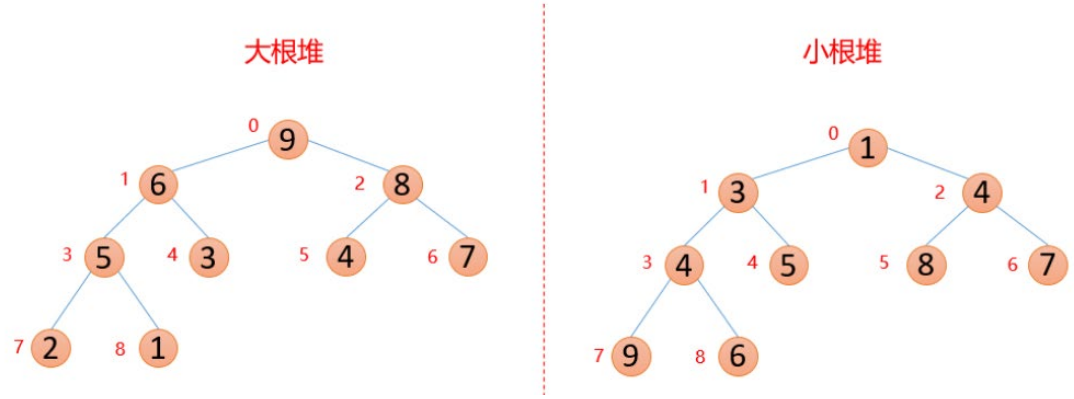
# Heap：

1. **什么是 Heap：**

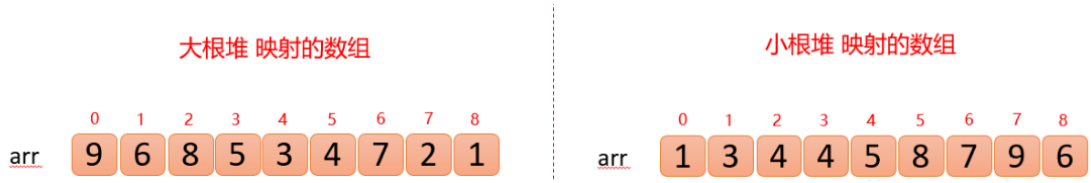
   堆的结构可以分为大根堆（MaxHeap）和小根堆（MinHeap），是一个完全二叉树，而堆排序是根据堆的这种数据结构设计的一种排序

2. **Heap 的结构与 array**

   性质：每个结点的值都大于其左孩子和右孩子结点的值，称之为大根堆；每个结点的值都小于其左孩子和右孩子结点的值，称之为小根堆。如下图：




我们对上面的图中每个数都进行了标记，上面的结构映射成数组就变成了下面这个样子：




3. **Heap 在 array 中的 index**

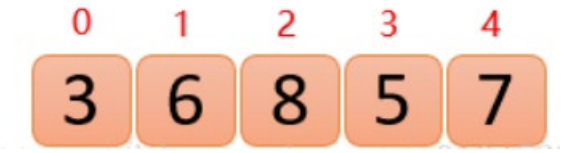
   1. 父结点索引：( I – 1 ) / 2（这里计算机中的除以 2，省略掉小数）
   2. 左孩子索引：2 * I + 1

3. 右孩子索引：2 * I + 2

4. **构造 Heap 和 heapSort**

**1. 首先将待排序的数组构造成一个大根堆，此时，整个数组的最大值就是堆结构的顶端**
**2. 将顶端的数与末尾的数交换，此时，末尾的数为最大值，剩余待排序数组个数为 n-1**
**3. 将剩余的 n-1 个数再构造成大根堆，再将顶端数与 n-1 位置的数交换，如此反复执行，便能得到有序数组**

假设存在以下数组：



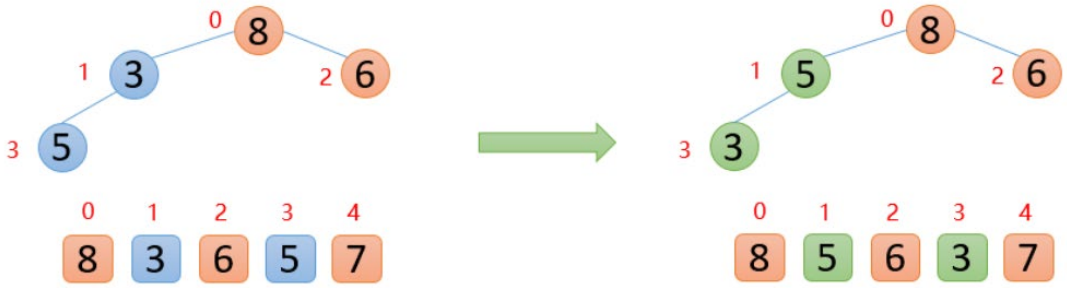

**主要思路：**
**第一次保证 0~0 位置满足 heap 要求**
**第二次保证 0~1 位置满足 heap 要求**
**第三次保证 0~2 位置满足 heap 要求...**
**直到保证 0~n-1 位置（每次新插入的数据都与其父结点进行比较，如果插入的数比父结点大，则与父结点交换，否则一直向上交换，直到小于等于父结点，或者来到了顶端）**

下面为对应的流程图：

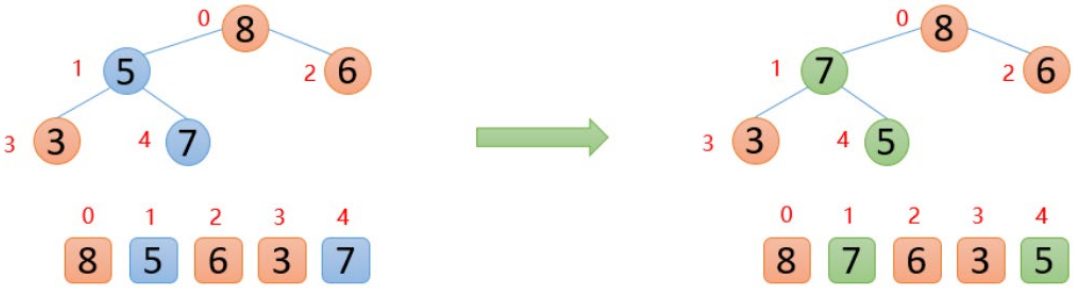
1. 插入 6 的时候，6 大于他的父结点 3，即 arr(1)>arr(0)，则交换；此时，保证了 0~1 位置是大根堆结构，如下图：


2. 插入 8 的时候，8 大于其父结点 6，即 arr(2)>arr(0),则交换；此时，保证了 0~2 位置是大根堆结构，如下图：

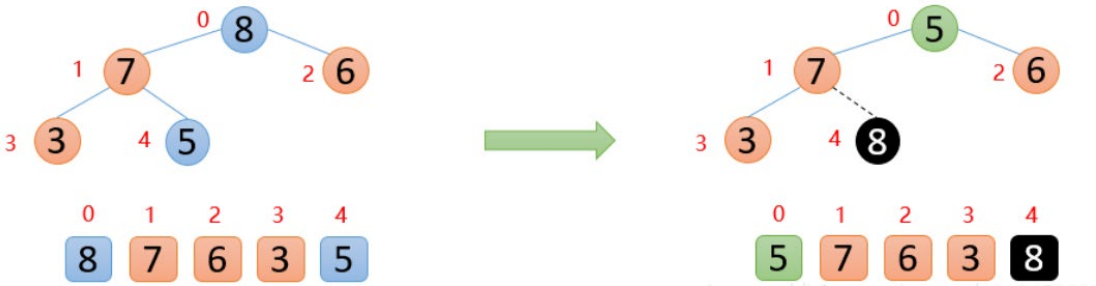
3. 插入 5 的时候，5 大于其父结点 3，则交换，交换之后，5 又发现比 8 小，所以不交换；此时，保证了 0~3 位置大根堆结构，如下图：




4. 插入 7 的时候，7 大于其父结点 5，则交换，交换之后，7 又发现比 8 小，所以不交换；此时**整个数组已经是大根堆结构**

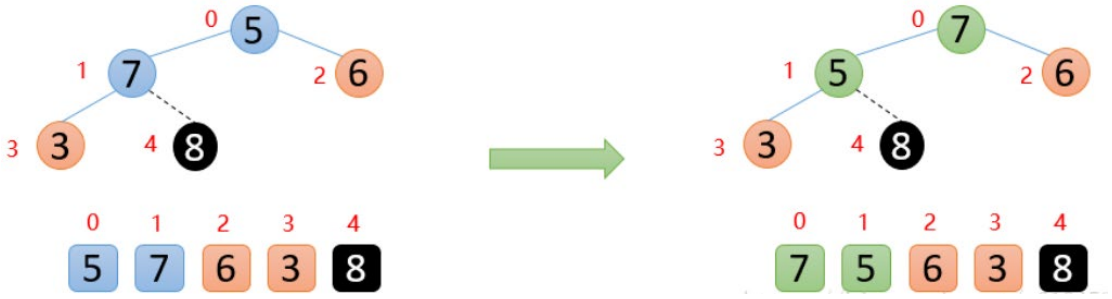



5. 此时，我们已经得到一个大根堆，下面将顶端的数与最后一位数交换，然后将剩余的数再构造成一个大根堆：



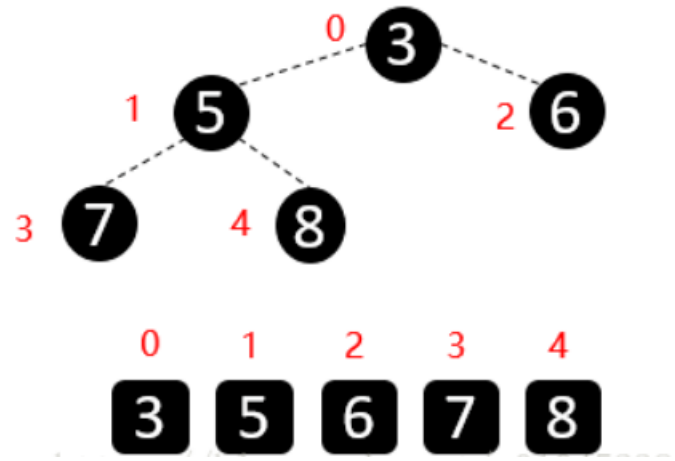

6. 此时最大数 8 已经来到末尾，则固定不动，后面只需要对顶端的数据进行操作即可，拿顶端的数与其左右孩子较大的数进行比较，如果顶端的数大于其左右孩子较大的数，则停止，如果顶端的数小于其左右孩子较大的数，则交换，然后继续与下面的孩子进行比较

   下图中，5 的左右孩子中，左孩子 7 比右孩子 6 大，则 5 与 7 进行比较，发现 5<7，则交换；交换后，发现 5 已经大于他的左孩子，说明剩余的数已经构成大根堆，后面就是重复固定最大值，然后构造大根堆




7. 如下图：顶端数 7 与末尾数 3 进行交换，固定好 7


8. 剩余的数开始构造大根堆 ， 然后顶端数与末尾数交换，固定最大值再构造大根堆，重复执行上面的操作，最终会得到有序数组




5. **总结**

**对于上面的操作，我们有如下的总结：**
**1、首先将无需数组构造成一个大根堆（新插入的数据与其父结点比较）**
**2、固定一个最大值，将剩余的数重新构造成一个大根堆，重复这样的过程**

6. **算法和应用**

**PriorityQueue 是 Heap 的一个应用，它能自动地给每次添加的元素经行排序，然后 poll 的时候会自动 poll 出目前最小的那个元素，也就是说实现了 MinHeap 的思想**

**题目：**

**给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。**
**请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。**

**Example 1：**
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

**Example 2：**
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4

**思路：**
**把数组全部排序好，这样就可以拿到第 k 大的元素，这样是一种解法，但是我们是需要第 K 大的元素，不一定要全部排序好再去拿，只针对部分元素进行排序，这样的复杂度显然可以降低的，也就是可以转化为：使用堆排序来解决这个问题——建立一个大顶堆，做 k−1 次删除操作后,堆顶元素就是我们要找的答案（堆排序过程中，不全部下沉，下沉 nums.length-k+1,然后堆顶可以得到我们 top k 答案了）**

**代码示例：**

```
public int findKthLargest(int[] nums, int k) {
      //构造小顶堆
    PriorityQueue<Integer> queue = new PriorityQueue<Integer>((a,
b)->a - b);
       for(int i = 0;i < nums.length;i ++){
            //当队列 size 小于 k 时，就加入元素；
           if(queue.size() < k){
              queue.add(nums[i]);
           }
            //否则判断小顶堆队首元素是否小于当前元素，如果小于说明有更大的元
素要加进来；
           else if(queue.peek() <= nums[i]){
              queue.poll();
              queue.add(nums[i]);
           }
       }
    //返回队首元素
       return queue.peek();
}
```