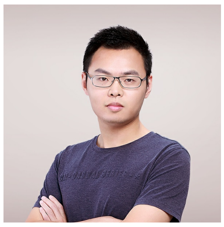
28 | 堆和堆排序：为什么说堆排序没有快速排序快？

time.geekbang.org/column/article/69913

数据结构与算法之美



王争

前Google工程师

查看详情

59308 人已学习

课程目录

已完结 73 讲

春节7天练 | Day 5：二叉树和堆

春节7天练 | Day 6：图

春节7天练 | Day 7：贪心、分治、回溯和动态规划

加餐：用户学习故事 (2讲)

祇

用户故事 | Jerry银银：这一年我的脑海里只有算法

用户故事 | zixuan：站在思维的高处，才有足够的视野和能力欣赏“美”

结束语 (1讲)

祇

结束语 | 送君千里，终须一别

嫻

数据结构与算法之美

鐙

鐙

柗

王争 2018-11-26

烁

00:00

15:34

讲述：修阳 大小：7.14M

我们今天讲另外一种特殊的树，“堆”（HeapHeap）。堆这种数据结构的应用场景非常多，最经典的莫过于堆排序了。堆排序是一种原地的、时间复杂度为 O(nlogn)O(nlog⁡n) 的排序算法。

前面我们学过快速排序，平均情况下，它的时间复杂度为 O(nlogn)O(nlog⁡n)。尽管这两种排序算法的时间复杂度都是 O(nlogn)O(nlog⁡n)，甚至堆排序比快速排序的时间复杂度还要稳定，但是，**在实际的软件开发中，快速排序的性能要比堆排序好，这是为什么呢？**

现在，你可能还无法回答，甚至对问题本身还有点疑惑。没关系，带着这个问题，我们来学习今天的内容。等你学完之后，或许就能回答出来了。

如何理解“堆”？

前面我们提到，堆是一种特殊的树。我们现在就来看看，什么样的树才是堆。我罗列了两点要求，只要满足这两点，它就是一个堆。

- 堆是一个完全二叉树；
- 堆中每一个节点的值都必须大于等于（或小于等于）其子树中每个节点的值。

我分别解释一下这两点。

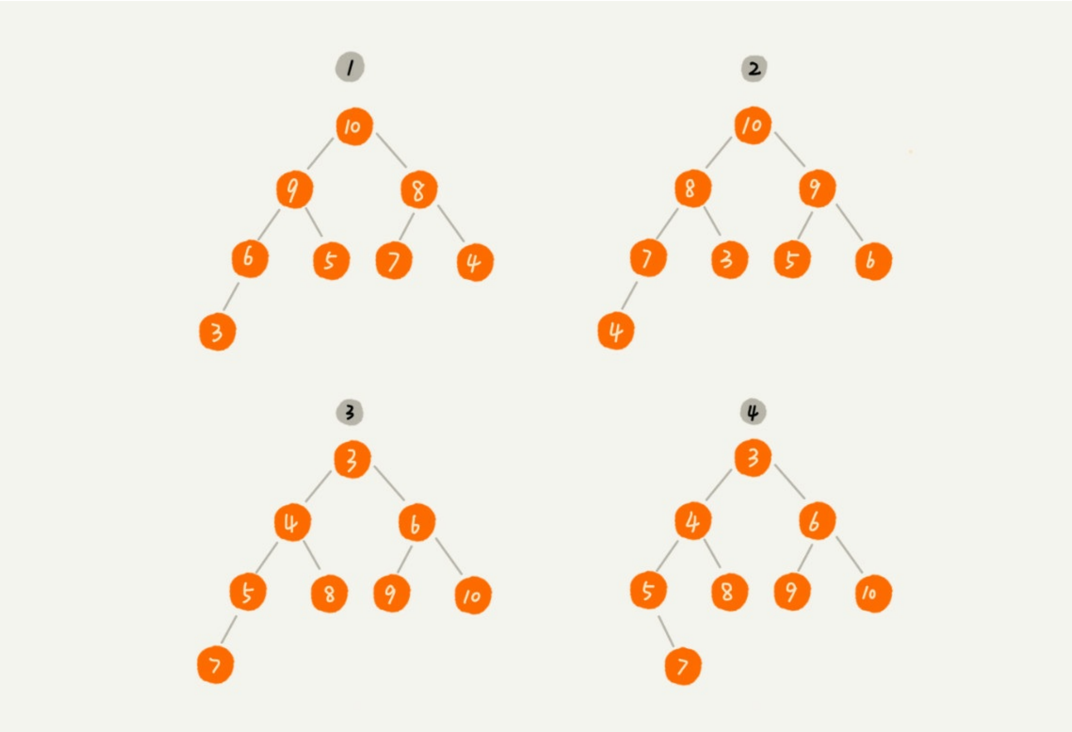
第一点，堆必须是一个完全二叉树。还记得我们之前讲的完全二叉树的定义吗？完全二叉树要求，除了最后一层，其他层的节点个数都是满的，最后一层的节点都靠左排列。

第二点，堆中的每个节点的值必须大于等于（或者小于等于）其子树中每个节点的值。实际上，

我们还可以换一种说法，堆中每个节点的值都大于等于（或者小于等于）其左右子节点的值。这两种表述是等价的。

对于每个节点的值都大于等于子树中每个节点值的堆，我们叫作“大顶堆”。对于每个节点的值都小于等于子树中每个节点值的堆，我们叫作“小顶堆”。

定义解释清楚了，你来看看，下面这几个二叉树是不是堆？



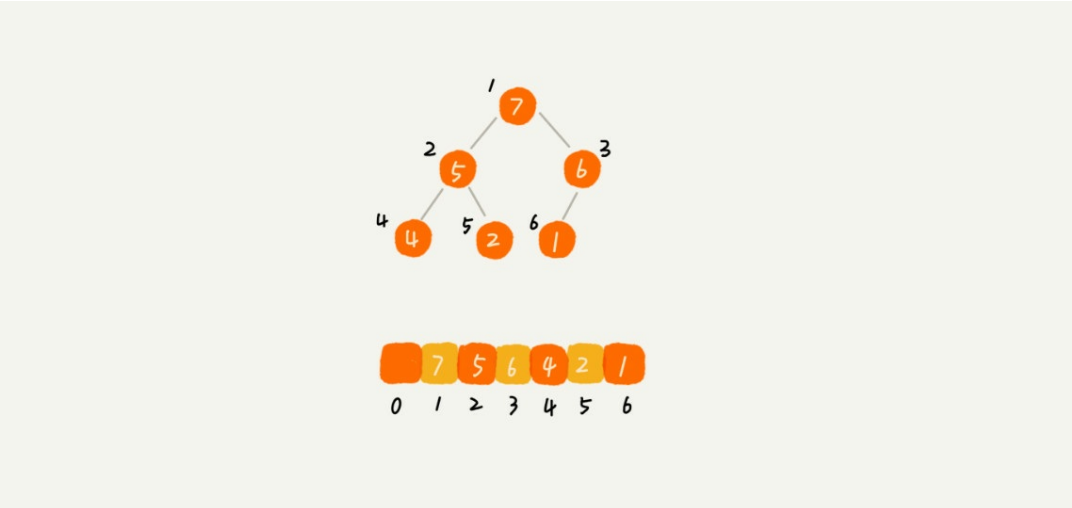
其中第 11 个和第 22 个是大顶堆，第 33 个是小顶堆，第 44 个不是堆。除此之外，从图中还可以看出来，对于同一组数据，我们可以构建多种不同形态的堆。

如何实现一个堆？

要实现一个堆，我们先要知道，**堆都支持哪些操作**以及**如何存储一个堆**。

我之前讲过，完全二叉树比较适合用数组来存储。用数组来存储完全二叉树是非常节省存储空间的。因为我们不需要存储左右子节点的指针，单纯地通过数组的下标，就可以找到一个节点的左右子节点和父节点。

我画了一个用数组存储堆的例子，你可以先看下。



从图中我们可以看到，数组中下标为 ii 的节点的左子节点，就是下标为 i∗2i∗2 的节点，右子节点就是下标为 i∗2+1i∗2+1 的节点，父节点就是下标为 i2i2 的节点。

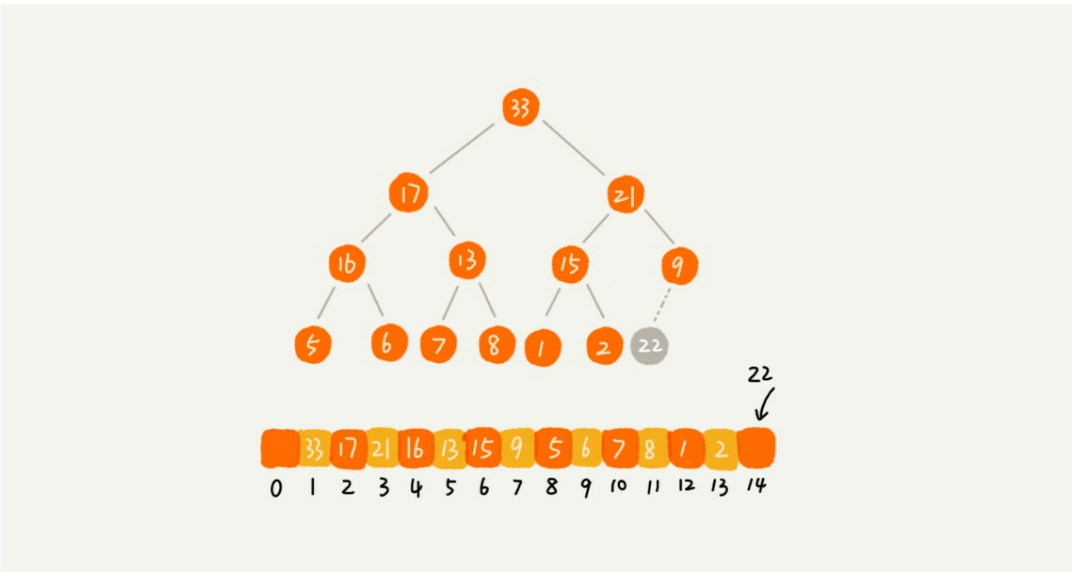
知道了如何存储一个堆，那我们再来看看，堆上的操作有哪些呢？我罗列了几个非常核心的操作，分别是往堆中插入一个元素和删除堆顶元素。（如果没有特殊说明，我下面都是拿大顶堆来讲解）。

1. 往堆中插入一个元素

往堆中插入一个元素后，我们需要继续满足堆的两个特性。

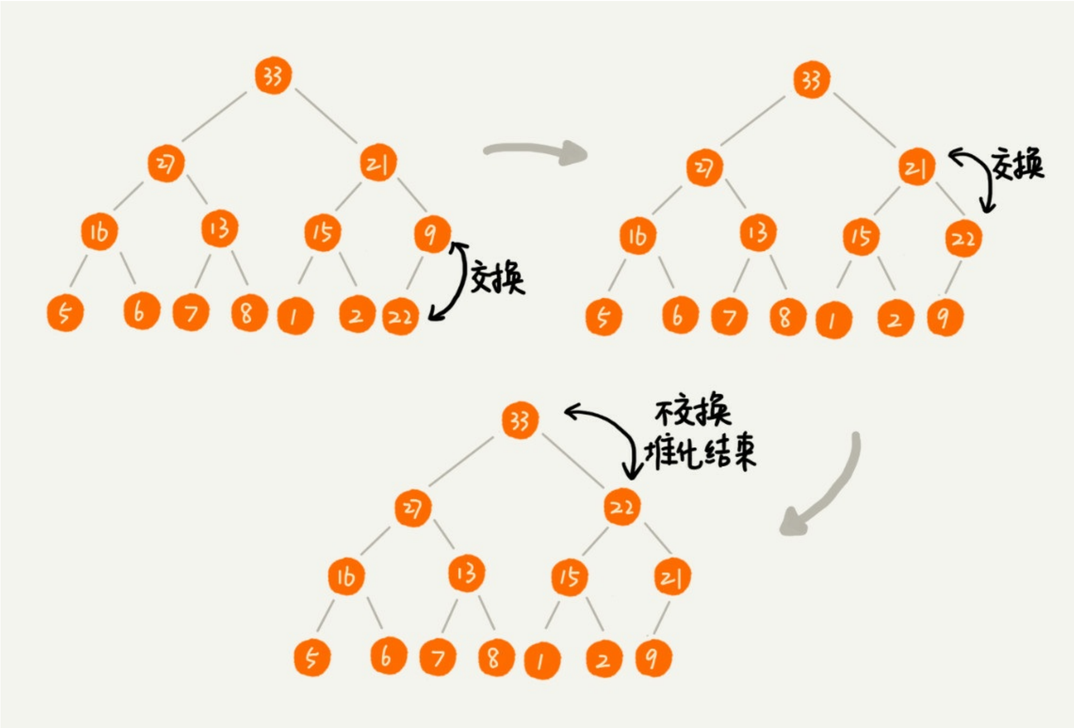
如果我们把新插入的元素放到堆的最后，你可以看我画的这个图，是不是不符合堆的特性了？于是，我们就需要进行调整，让其重新满足堆的特性，这个过程我们起了一个名字，就叫作**堆化**（heapify）。

堆化实际上有两种，从下往上和从上往下。这里我先讲**从下往上**的堆化方法。



堆化非常简单，就是顺着节点所在的路径，向上或者向下，对比，然后交换。

我这里画了一张堆化的过程分解图。我们可以让新插入的节点与父节点对比大小。如果不满足子节点小于等于父节点的大小关系，我们就互换两个节点。一直重复这个过程，直到父子节点之间满足刚说的那种大小关系。



我将上面讲的往堆中插入数据的过程，翻译成了代码，你可以结合着一块看。

```
public class Heap {
private int[] a; // 数组，从下标 1 开始存储数据
private int n; // 堆可以存储的最大数据个数
private int count; // 堆中已经存储的数据个数

public Heap(int capacity) {
a = new int[capacity + 1];
n = capacity;
count = 0;
}

public void insert(int data) {
if (count >= n) return; // 堆满了
++count;
a[count] = data;
int i = count;
while (i/2 > 0 && a[i] > a[i/2]) { // 自下往上堆化
swap(a, i, i/2); // swap() 函数作用：交换下标为 i 和 i/2 的两个元素
i = i/2;
}
}
}
```

複复制代码

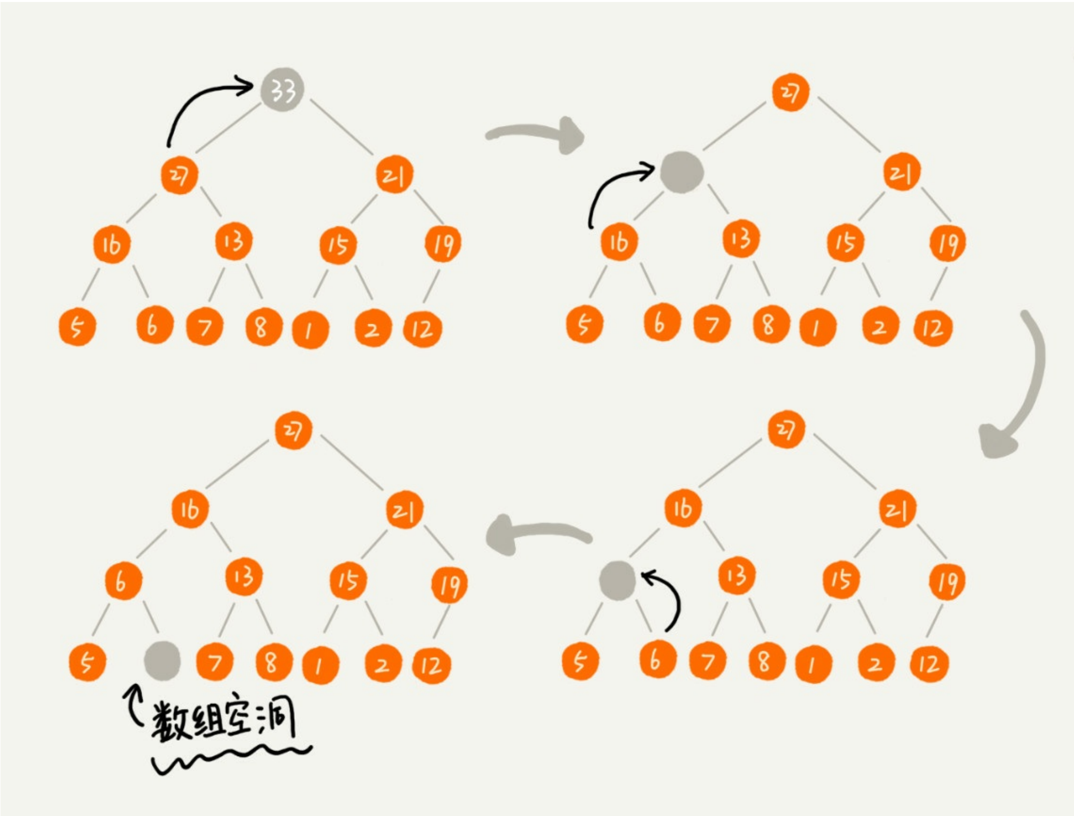
2. 删除堆顶元素

从堆的定义的第二条中，任何节点的值都大于等于（或小于等于）子树节点的值，我们可以发现，堆顶元素存储的就是堆中数据的最大值或者最小值。

假设我们构造的是大顶堆，堆顶元素就是最大的元素。当我们删除堆顶元素之后，就需要把第二

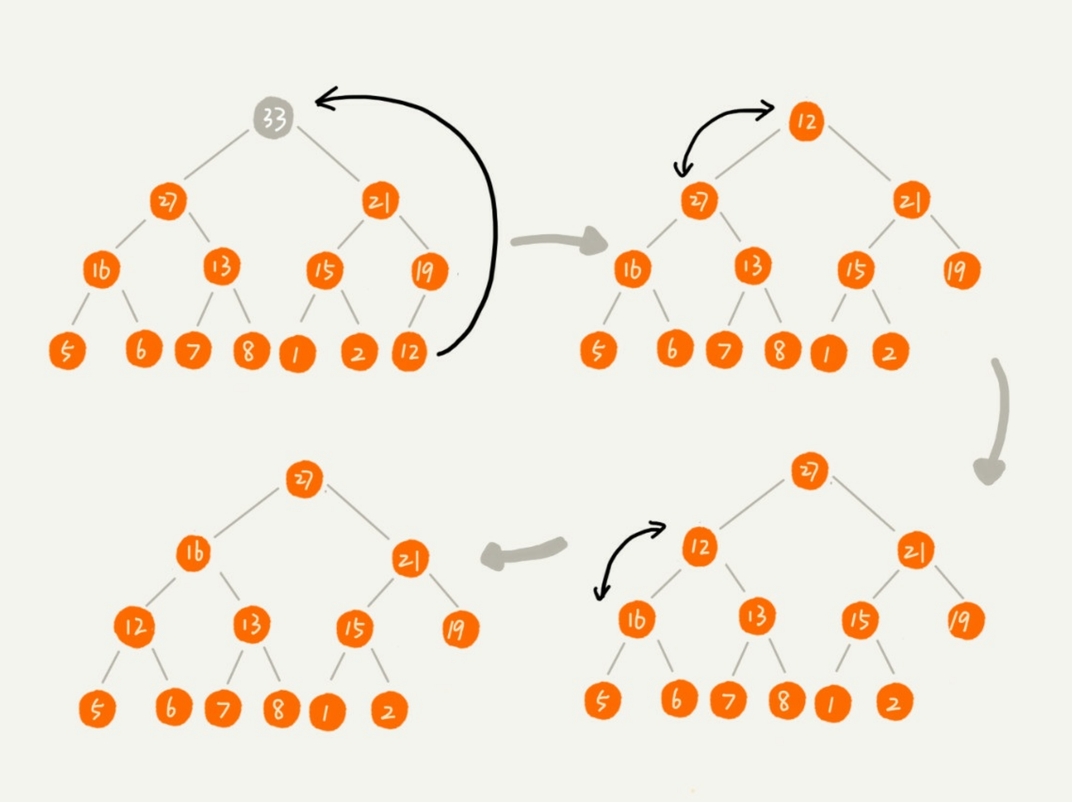
大的元素放到堆顶，那第二大元素肯定会出现在左右子节点中。然后我们再迭代地删除第二大节点，以此类推，直到叶子节点被删除。

这里我也画了一个分解图。不过这种方法有点问题，就是最后堆化出来的堆并不满足完全二叉树的特性。



实际上，我们稍微改变一下思路，就可以解决这个问题。你看我画的下面这幅图。我们把最后一个节点放到堆顶，然后利用同样的父子节点对比方法。对于不满足父子节点大小关系的，互换两个节点，并且重复进行这个过程，直到父子节点之间满足大小关系为止。这就是**从上往下的堆化方法**。

因为我们移除的是数组中的最后一个元素，而在堆化的过程中，都是交换操作，不会出现数组中的“空洞”，所以这种方法堆化之后的结果，肯定满足完全二叉树的特性。



我把上面的删除过程同样也翻译成了代码，贴在这里，你可以结合着看。

```
public void removeMax() {
if (count == 0) return -1; // 堆中没有数据
a[1] = a[count];
--count;
heapify(a, count, 1);
}

private void heapify(int[] a, int n, int i) { // 自上往下堆化
while (true) {
int maxPos = i;
if (i*2 <= n && a[i] < a[i*2]) maxPos = i*2;
if (i*2+1 <= n && a[maxPos] < a[i*2+1]) maxPos = i*2+1;
if (maxPos == i) break;
swap(a, i, maxPos);
i = maxPos;
}
}
```

榔复制代码

我们知道，一个包含 nn 个节点的完全二叉树，树的高度不会超过 log2nlog2n。堆化的过程是顺着节点所在路径比较交换的，所以堆化的时间复杂度跟树的高度成正比，也就是 O(logn)O(logn)。插入数据和删除堆顶元素的主要逻辑就是堆化，所以，往堆中插入一个元素和删除堆顶元素的时间复杂度都是 O(logn)O(logn)。

如何基于堆实现排序？

前面我们讲过好几种排序算法，我们再来回忆一下，有时间复杂度是 O(n2)O(n2) 的冒泡排序、插入排序、选择排序，有时间复杂度是 O(nlogn)O(nlogn) 的归并排序、快速排序，还有线性排

序。

这里我们借助于堆这种数据结构实现的排序算法，就叫作堆排序。这种排序方法的时间复杂度非常稳定，是 $O(n\log n)$，并且它还是原地排序算法。如此优秀，它是怎么做到的呢？

我们可以把堆排序的过程大致分解成两个大的步骤，**建堆**和**排序**。

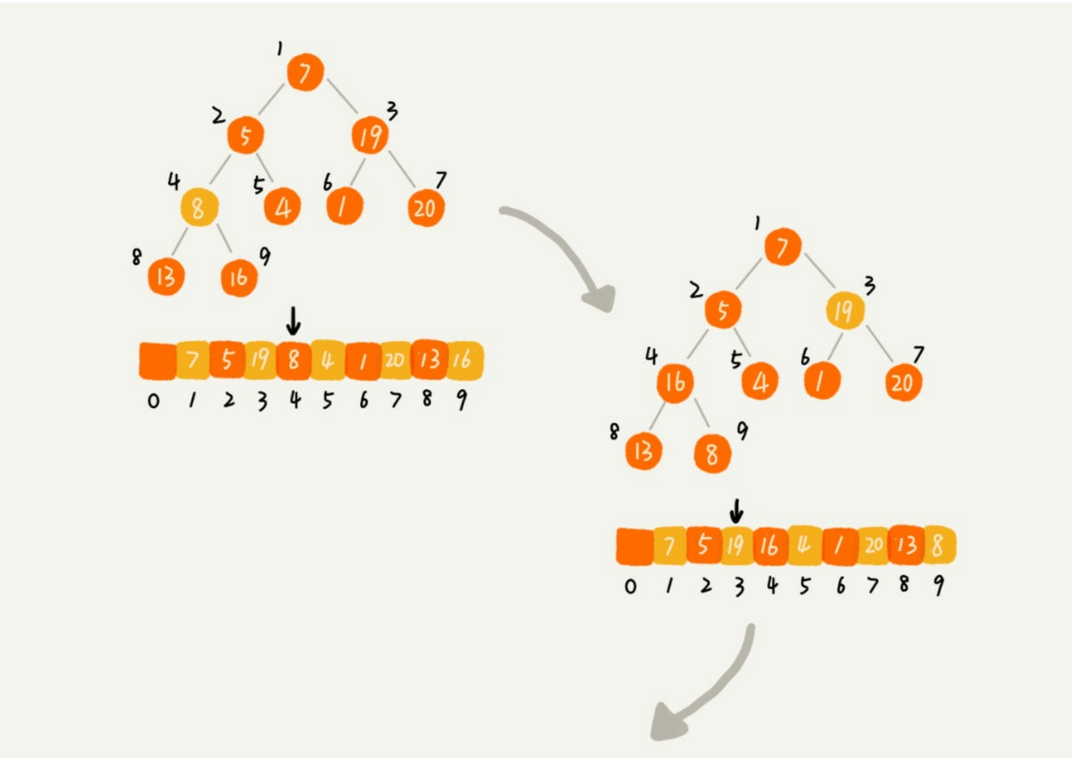
1. 建堆

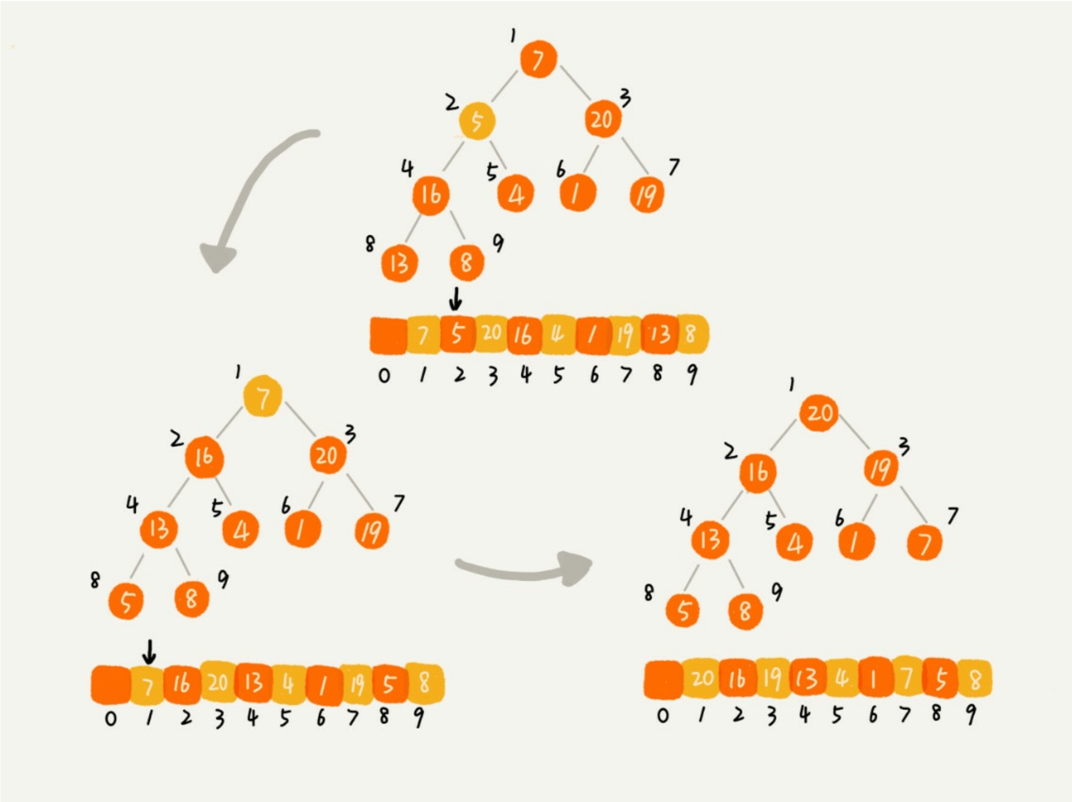
我们首先将数组原地建成一个堆。所谓“原地”就是，不借助另一个数组，就在原数组上操作。建堆的过程，有两种思路。

第一种是借助我们前面讲的，在堆中插入一个元素的思路。尽管数组中包含 n 个数据，但是我们可以假设，起初堆中只包含一个数据，就是下标为 1 的数据。然后，我们调用前面讲的插入操作，将下标从 2 到 n 的数据依次插入到堆中。这样我们就将包含 n 个数据的数组，组织成了堆。

第二种实现思路，跟第一种截然相反，也是我这里要详细讲的。第一种建堆思路的处理过程是从前往后处理数组数据，并且每个数据插入堆中时，都是从下往上堆化。而第二种实现思路，是从后往前处理数组，并且每个数据都是从上往下堆化。

我举了一个例子，并且画了一个第二种实现思路的建堆分解步骤图，你可以看下。因为叶子节点往下堆化只能自己跟自己比较，所以我们直接从第一个非叶子节点开始，依次堆化就行了。





对于程序员来说，看代码可能更好理解一些，所以，我将第二种实现思路翻译成了代码，你可以看下。

```
private static void buildHeap(int[] a, int n) {
for (int i = n/2; i >= 1; --i) {
heapify(a, n, i);
}
}

private static void heapify(int[] a, int n, int i) {
while (true) {
int maxPos = i;
if (i*2 <= n && a[i] < a[i*2]) maxPos = i*2;
if (i*2+1 <= n && a[maxPos] < a[i*2+1]) maxPos = i*2+1;
if (maxPos == i) break;
swap(a, i, maxPos);
i = maxPos;
}
}
```

櫛复制代码

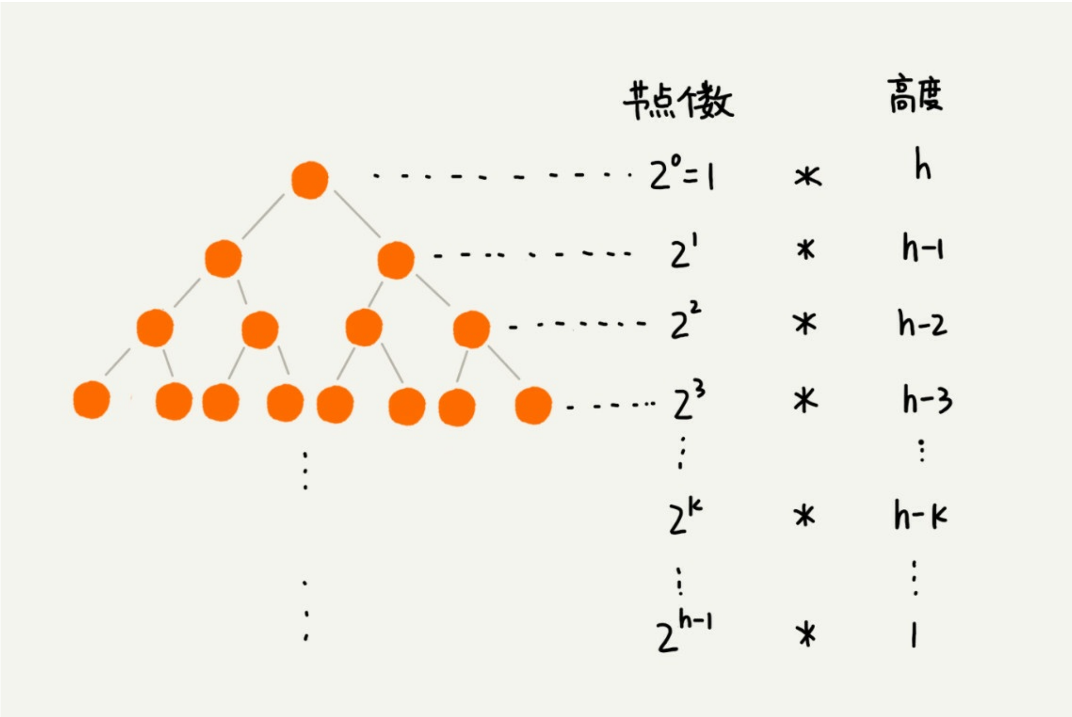
你可能已经发现了，在这段代码中，我们对下标从 n2n2 开始到 11 的数据进行堆化，下标是 n2+1n2+1 到 nn 的节点是叶子节点，我们不需要堆化。实际上，对于完全二叉树来说，下标从 n2+1n2+1 到 nn 的节点都是叶子节点。

现在，我们来看，建堆操作的时间复杂度是多少呢？

每个节点堆化的时间复杂度是 O(logn)O(log⁡n)，那 n2+1n2+1 个节点堆化的总时间复杂度是不是就是 O(nlogn)O(nlog⁡n) 呢？这个答案虽然也没错，但是这个值还是不够精确。实际上，堆排序的建堆过程的时间复杂度是 O(n)O(n)。我带你推导一下。

因为叶子节点不需要堆化，所以需要堆化的节点从倒数第二层开始。每个节点堆化的过程中，需要比较和交换的节点个数，跟这个节点的高度 kk 成正比。

我把每一层的节点个数和对应的高度画了出来，你可以看看。我们只需要将每个节点的高度求和，得出的就是建堆的时间复杂度。



我们将每个非叶子节点的高度求和，就是下面这个公式：

$$S_1 = 1 * h + 2^1 * (h-1) + 2^2 * (h-2) + \cdots + 2^k * (h-k) + \cdots + 2^{h-1} * 1$$

这个公式的求解稍微有点技巧，不过我们高中应该都学过：把公式左右都乘以 22，就得到另一个公式 S2S2。我们将 S2S2 错位对齐，并且用 S2S2 减去 S1S1，可以得到 SS。

$$S_1 = 1 * h + 2^1 * (h-1) + 2^2 * (h-2) + \cdots + 2^k * (h-k) + \cdots + 2^{h-1} * 1$$

$$S_2 = 2^1 * h + 2^2 * (h-1) + \cdots + 2^k * (h-k+1) + \cdots + 2^{h-1} * 2 + 2^h * 1$$

$$S = S_2 - S_1 = -h + \underbrace{2 + 2^2 + 2^3 + \cdots + 2^k + \cdots + 2^{h-1}}_{\text{等比数列}} + 2^h$$

SS 的中间部分是一个等比数列，所以最后可以用等比数列的求和公式来计算，最终的结果就是下面图中画的这个样子。

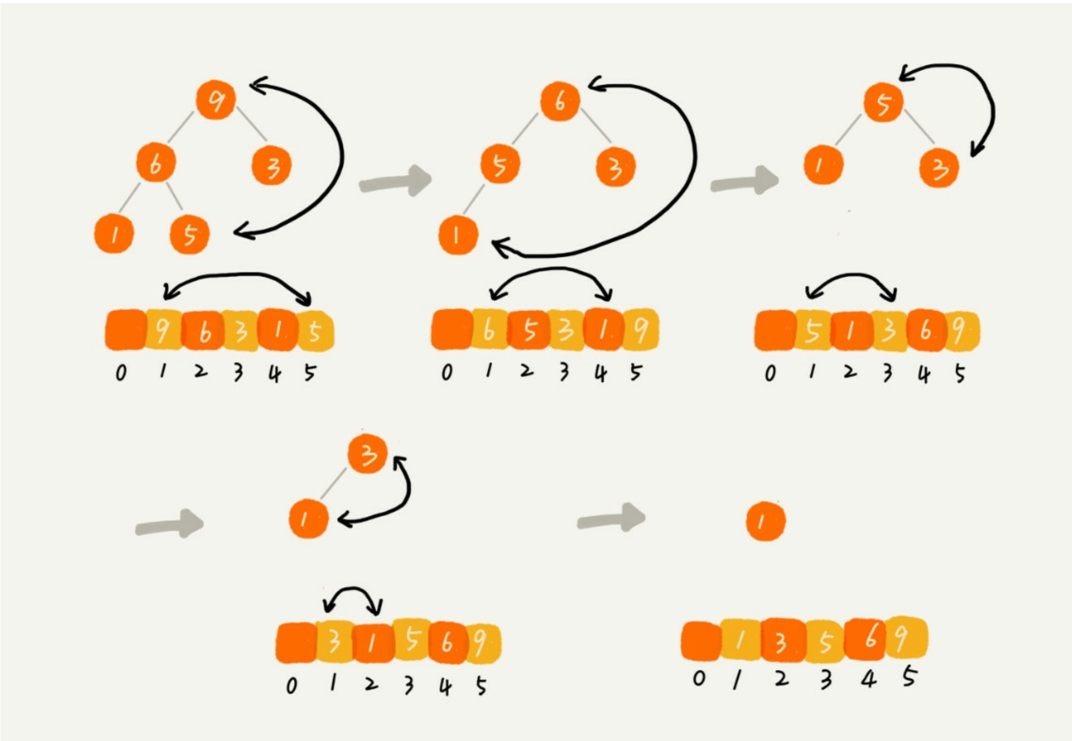
$$S=-h+(2^h-2)+2^h=2^{h+1}-h-2$$

因为 h=log2nh=log2n，代入公式 SS，就能得到 S=O(n)S=O(n)，所以，建堆的时间复杂度就是 O(n)O(n)。

2. 排序

建堆结束之后，数组中的数据已经是按照大顶堆的特性来组织的。数组中的第一个元素就是堆顶，也就是最大的元素。我们把它跟最后一个元素交换，那最大元素就放到了下标为 nn 的位置。

这个过程有点类似上面讲的“删除堆顶元素”的操作，当堆顶元素移除之后，我们把下标为 nn 的元素放到堆顶，然后再通过堆化的方法，将剩下的 n−1n−1 个元素重新构建成堆。堆化完成之后，我们再取堆顶的元素，放到下标是 n−1n−1 的位置，一直重复这个过程，直到最后堆中只剩下标为 11 的一个元素，排序工作就完成了。



堆排序的过程，我也翻译成了代码。结合着代码看，你理解起来应该会更加容易。

```
// n 表示数据的个数，数组 a 中的数据从下标 1 到 n 的位置。
public static void sort(int[] a, int n) {
buildHeap(a, n);
int k = n;
while (k > 1) {
swap(a, 1, k);
--k;

heapify(a, k, 1);
}
}
```

櫛复制代码

现在，我们再来分析一下堆排序的时间复杂度、空间复杂度以及稳定性。

整个堆排序的过程，都只需要极个别临时存储空间，所以堆排序是原地排序算法。堆排序包括建堆和排序两个操作，建堆过程的时间复杂度是 O(n)O(n)，排序过程的时间复杂度是 O(nlogn)O(nlogn)，所以，堆排序整体的时间复杂度是 O(nlogn)O(nlogn)。

堆排序不是稳定的排序算法，因为在排序的过程，存在将堆的最后一个节点跟堆顶节点互换的操作，所以就有可能改变值相同数据的原始相对顺序。

今天的内容到此就讲完了。我这里要稍微解释一下，在前面的讲解以及代码中，我都假设，堆中的数据是从数组下标为 1 的位置开始存储。那如果从 00 开始存储，实际上处理思路是没有任何变化的，唯一变化的，可能就是，代码实现的时候，计算子节点和父节点的下标的公式改变了。

如果节点的下标是 ii，那左子节点的下标就是 2∗i+12∗i+1，右子节点的下标就是 2∗i+22∗i+2，父节点的下标就是 i−12i−12。

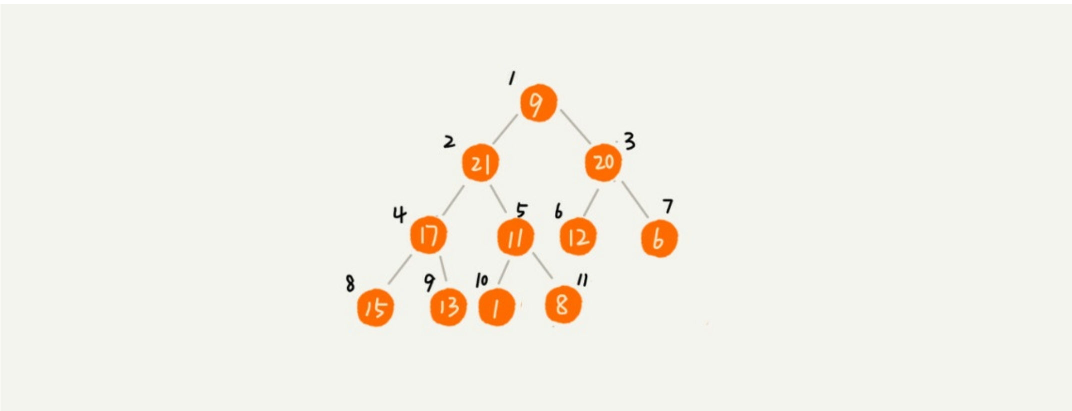
解答开篇

现在我们来看开篇的问题，在实际开发中，为什么快速排序要比堆排序性能好？

我觉得主要有两方面的原因。

第一点，堆排序数据访问的方式没有快速排序友好。

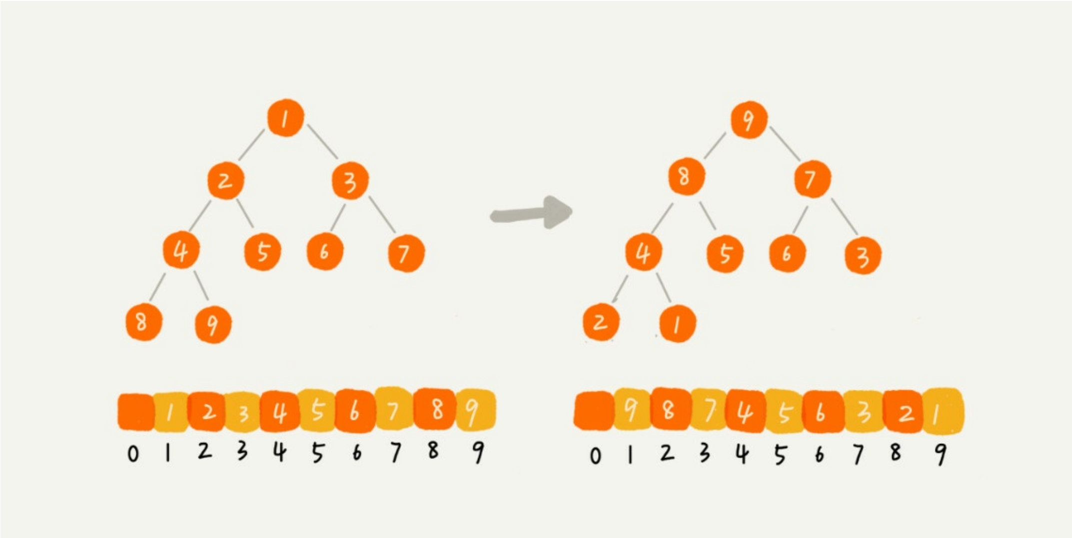
对于快速排序来说，数据是顺序访问的。而对于堆排序来说，数据是跳着访问的。 比如，堆排序中，最重要的一个操作就是数据的堆化。比如下面这个例子，对堆顶节点进行堆化，会依次访问数组下标是 1，2，4，81，2，4，8 的元素，而不是像快速排序那样，局部顺序访问，所以，这样对 CPU 缓存是不友好的。



第二点，对于同样的数据，在排序过程中，堆排序算法的数据交换次数要多于快速排序。

我们在讲排序的时候，提过两个概念，有序度和逆序度。对于基于比较的排序算法来说，整个排序过程就是由两个基本的操作组成的，比较和交换（或移动）。快速排序数据交换的次数不会比逆序度多。

但是堆排序的第一步是建堆，建堆的过程会打乱数据原有的相对先后顺序，导致原数据的有序度降低。比如，对于一组已经有序的数据来说，经过建堆之后，数据反而变得更无序了。



对于第二点，你可以自己做个试验看下。我们用一个记录交换次数的变量，在代码中，每次交换的时候，我们就对这个变量加一，排序完成之后，这个变量的值就是总的数据交换次数。这样你就能很直观地理解我刚刚说的，堆排序比快速排序交换次数多。

内容小结

今天我们讲了堆这种数据结构。堆是一种完全二叉树。它最大的特性是：每个节点的值都大于等于（或小于等于）其子树节点的值。因此，堆被分成了两类，大顶堆和小顶堆。

堆中比较重要的两个操作是插入一个数据和删除堆顶元素。这两个操作都要用到堆化。插入一个数据的时候，我们把新插入的数据放到数组的最后，然后从下往上堆化；删除堆顶数据的时候，我们把数组中的最后一个元素放到堆顶，然后从上往下堆化。这两个操作时间复杂度都是 O(logn)O(log⁡n)。

除此之外，我们还讲了堆的一个经典应用，堆排序。堆排序包含两个过程，建堆和排序。我们将下标从 n2n2 到 11 的节点，依次进行从上到下的堆化操作，然后就可以将数组中的数据组织成堆这种数据结构。接下来，我们迭代地将堆顶的元素放到堆的末尾，并将堆的大小减一，然后再堆化，重复这个过程，直到堆中只剩下一个元素，整个数组中的数据就都有序排列了。

课后思考

1. 在讲堆排序建堆的时候，我说到，对于完全二叉树来说，下标从 n2+1n2+1 到 nn 的都是叶子节点，这个结论是怎么推导出来的呢？
2. 我们今天讲了堆的一种经典应用，堆排序。关于堆，你还能想到它的其他应用吗？

欢迎留言和我分享，我会第一时间给你反馈。

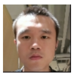
涛强

Command + Enter 发表

0/2000字

提交留言

精选留言(113)

- 

  Jerry银银

  ## 第一题：

  使用数组存储表示完全二叉树时，从数组下标为1开始存储数据，数组下标为i的节点，左子节点为2i, 右子节点为2i + 1. 这个结论很重要（可以用数学归纳法证明），将此结论记为『原理1』，以下证明会用到这个原理。

  为什么，对于完全二叉树来说，下标从n/2 + 1 到 n的节点都是叶子节点？ 使用反证法证明即可：

  如果下标为n/2 + 1的节点不是叶子节点，即它存在子节点，按照『原理1』，它的左子节点为：2(n/2 + 1) = n + 2，大家明显可以看出，这个数字已经大于n + 1，超出了实现完全二叉树所用数组的大小（数组下标从1开始记录数据，对于n个节点来说，数组大小是n + 1），左子节点都已经超出了数组容量，更何况右子节点。以此类推，很容易得出：下标大于n/2 + 1的节点肯定都是也叶子节点了，故而得出结论：对于完全二叉树来说，下标从n/2 + 1 到 n的节点都是叶子节点

  备注下：用数组存储表示完全二叉树时，也可以从下标为0开始，只是这样做的话，计算左

子节点时，会多一次加法运算

第二题：

堆的应用除了堆排以外，还有如下一些应用：
1. 从大数量级数据中筛选出top n 条数据； 比如：从几十亿条订单日志中筛选出金额靠前的1000条数据
2. 在一些场景中，会根据不同优先级来处理网络请求，此时也可以用到优先队列(用堆实现的数据结构)；比如：网络框架Volley就用了Java中PriorityBlockingQueue，当然它是线程安全的
3. 可以用堆来实现多路归并，从而实现有序，leetcode上也有相关的一题：Merge K Sorted Lists

暂时只能想到以上三种常见的应用场景，其它的，希望老师补充！

2018-11-26

 2

 178

- 

  Jessie

  强烈建议，在进入课程的左侧，做一个目录，这样就不用每次都从最新的滑到最下面了。例如：目前是学到了第35课，已进入课堂，就是35课，假如我想看第一课，就得使劲滑。如果学到100课，那得滑老半天……所以强烈建议给左侧添加一个目录。可以连接到每一节课。！！！

  2018-12-13

   2

   115

- 

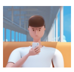
  WhoAmWe

  应用:
  1.topK
  2.流里面的中值
  3.流里面的中位数

  作者回复: 👍

  2018-11-26

   39

- 

冯选刚

不知道有没有人很容易看懂原理思路，就是不愿意看代码

2018-11-27

𤢴

𡛷 36

-

  无心拾贝

  这TM估计是我唯一看的懂的数据结构与算法吧。 谢谢老师！

  2018-11-26

  𤢴

  𡛷 24

- 

  猫头鹰爱拿铁

  思考题1:堆是完全二叉树，求最后的非叶子节点即是求最大的叶子节点的父节点。最大的叶子节点下标为n，他的父节点为n/2，这是最后一个非叶子节点，所以n/2+1到n都是叶子节点。
  思考题2:堆排序的应用-topk问题，例如求数据频率最高的k个数，建立k个数的最小顶堆，然后剩余数据和堆顶比较，如果比堆顶大则替换堆顶并重新调整最小顶堆。

  2018-11-26

  𤢴 1

  𡛷 20

- 

  insist

  这种数据结构堆和java内存模型中的堆内存有什么关系呢？

  作者回复: 完全没关系的。

  2018-11-26

  𤢴

  𡛷 16

- 

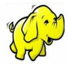
  Smallfly

堆应该可以用于实现优先级队列。

2018-11-26

璊

娯 10

- 

朝夕心

思考题1证明
结论：对于完全二叉树来说，下标从(n/2)+1到n都是叶子节点
证明：
假设堆有n个节点
假设满二叉树有h层 则满二叉树的总节点数 2^0+2^1...+2^(h-2)+2^(h-1)=(2^h)-1> n n为h层完全二叉树节点数
堆为完全二叉树，相同高度，完全二叉树总结点数小于满二叉树节点数，即n<(2^h)-1， 即(2^h)>n+1 -----①
完全二叉树1到h-1层节点的数量总和： 2^0+2^1...+2^(h-2)=(2^(h-1))-1=(2^h)/2 -1 -----②
如果数组的第0位也存储数据，由②可知，完全二叉树的第h层开始的节点的下标为i=(2^h)/2 -1，由①，i>((n+1)/2)-1=(n/2)+1
结论1：如果数组的第0位也存储数据，完全二叉树的节点下标至少开始于(n/2)+1
如果数组的第0位不存储数据，则由②可知，完全二叉树的第h层开始的节点的下标为j=(2^h)/2，由①，j>(n+1)/2=(n/2)+2
结论2：如果数组的第0位不存储数据，完全二叉树的节点下标至少开始于(n/2)+2
综上，堆（完全二叉树）的叶子节点的下标范围从(n/2)+1到n-1或从(n/2)+2到n，也即堆的叶子节点下标从(n/2)+1到n
欢迎指正
--不为别的，就为成为更合格的自己

作者回复: 👍

2019-02-26

璊 1

娯 9

- 

鲍勃

linux内核内存中的堆和这个有关系吗？

作者回复: 没关系 完全是两个东西

2018-11-30

璊

娯 7

- 

Brandon

排序队------时间复杂度
堆满足两条：1、完全二叉树（可以很方便的使用数组存储），2、父节点大于或小于子节点-----
插入元素-先放入队尾，再进行堆化（heapify）
删除元素-从最后取一个元素放到删除元素位置，从上往下调整

快排比堆排性能好的原因有二：1、堆排序数据访问的方式有没有快速排序友好；
2、对于同样的数据，在排序的过程中堆排序算法的交换次数多于快速排序

2018-12-14

㺍

娊 4

- 

  .

  "对堆顶节点进行堆化，会依次访问数组下标是 1，2，4，8"。这里图画错了吧，数组下标2 (20)和数组下标3(21)的位置应该是弄反了。如果按原图对堆顶元素堆化的话顺序应该是1,3,6不应该是1,2,4,8

  作者回复: 嗯嗯 多谢指正

  2018-12-14

  㺍

  娊 4

- 

  若星

  删除堆顶元素的代码第二行return -1。。

  2018-12-12

  㺍

  娊 4

- 

  李建轰

  老师你好～
  heapify方法好像有点问题？
  假如第一个非叶子节点是5，左叶子节点是7，右叶子节点是6
  然后入heapify方法的这段代码
  ```

  while (true) {
  ```

```
    int maxPos = i;
    if (i*2 <= n && a[i] < a[i*2]) maxPos = i*2;
    if (i*2+1 <= n && a[i] < a[i*2+1]) maxPos = i*2+1
    if (maxPos == i) break;
    swap(a, i, maxPos);
    i = maxPos;
}
```
就会变成第一个非叶子节点是6，左叶子节点是7，右叶子节点是5，因为swap只会执行一次。
我觉得swap方法在前面两个if里面都得有，并且第二个if必须用if，不能用else if。
斗胆提问，请老师答疑~

2018-11-30

璑 1

娯 4

- 

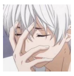
  yaya

  最后一个结点的父节点是n/2.这是最后一个非叶子结点所以，叶子结点是n/2+1到n。
  优先队列的实现用的就是堆

  2018-11-26

  璑

  娯 4

- 

  博予liutxer

  堆排序和快速排序相比实际开发中不如后者性能好，那堆排序在哪些场景比较有优势呢？

  作者回复: 时间复杂度比较稳定 有些排序函数会使用这种排序算法

  2018-12-09

  璑

  娯 3

- 

  小二黑

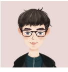
  老师，请问堆化自上而下，那段代码，节点和子节点比较大小，是用if判断的吗

  作者回复: if判断是什么意思呢

  2019-03-27

璊

娊 2

- 

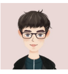
  Rephontil

  这一节课看了两遍，看得清清楚楚，明明白白😊

  2019-01-07

  璊

  娊 2

- 

  Rephontil

  老师，“我们将每个非叶子节点的高度求和，就是下面这个公式”，这个公式的末尾部分2^(h-1) * 1应该是不需要的吧，因为这个2^(h-1) * 1是最底层叶子节点的高度。

  2019-01-05

  璊

  娊 2

- 

  惟新

  好久没看了，又开始重新看了。另外还在线画了思维导图。

  2019-01-04

  璊

  娊 2

收起评论祇

璊99+

鍮99+

挊

岲

忛

毖

[攡]

椈

箈