数据结构

信息科学与工程学院

第 6 章

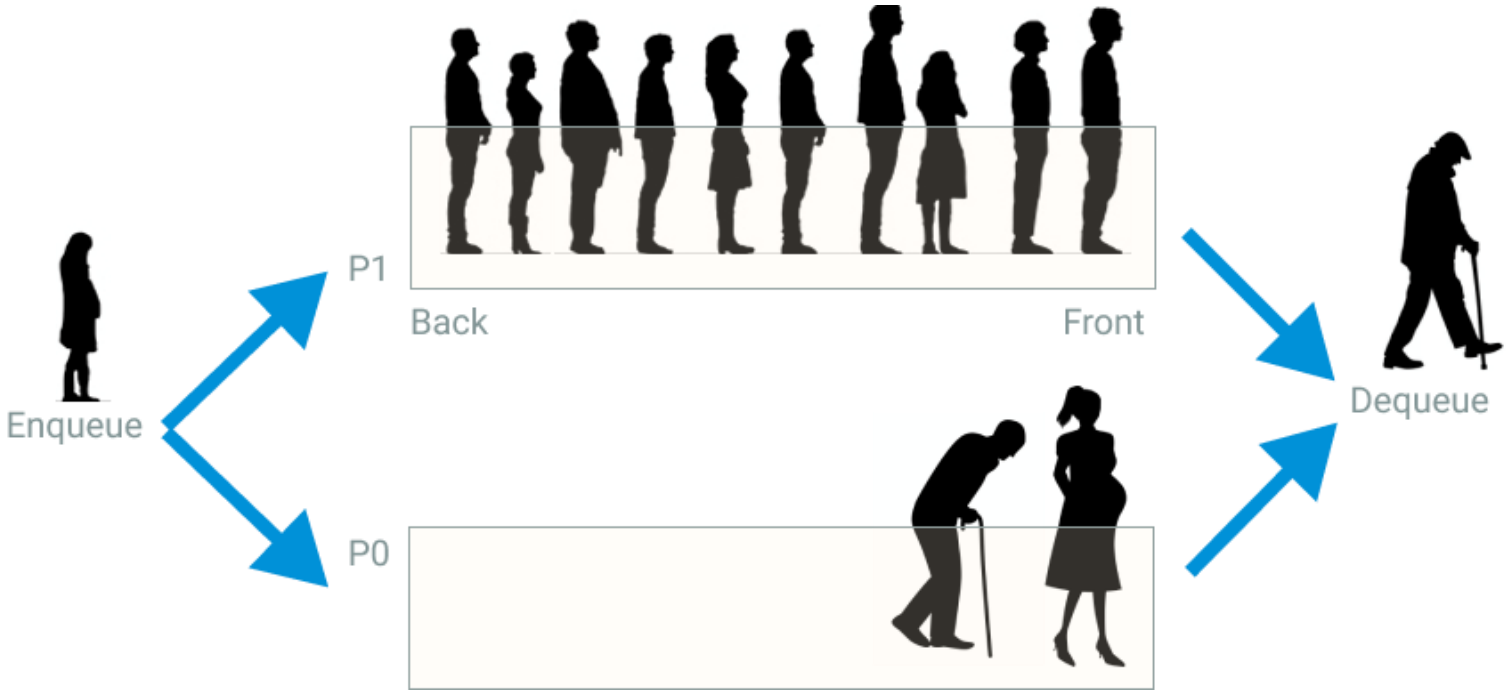
优先级队列

提纲

6.1 问题引入：带优先级的服务处理

问题：第3章介绍的**队列**能够按先到先得的方式来处理，但实际问题中（例如医院或者银行）还有可能需要考虑加急的情况，更一般地说就是**优先级**。如何按优先级进行处理？

关键：出队的顺序需要由元素本身的优先级来决定，而不是进队的顺序。



6.2 优先级队列的定义

优先级队列是一种特殊的队列。与普通队列相比，

- **相同点**：都支持进队和出队操作。
- **不同点**：优先级队列的出队顺序按事先规定的优先级顺序进行。

ADT PriorityQueue {
数据对象：
　元素取自全集 *U* 的可重集合 *E*，表示优先级队列中包含的元素。
数据关系：
　全集 *U* 中的元素须满足严格弱序。
基本操作：
　（省略初始化、销毁、清除内容、判断为空、查询元素个数等操作）
　Insert(*pq*, *x*)：在优先级队列 *pq* 中插入元素 *x*。
　ExtractMin(*pq*)：从优先级队列 *pq* 中删除优先级最高（也就是值最小）的元素，并返回。
　PeekMin(*pq*)：返回优先级队列 *pq* 中优先级最高的元素（元素仍然保留在优先级队列中）。
}

优先级队列的实现

优先级队列**可以**用线性表来实现。

然而，使用线性表实现时出队操作需要将当前优先级队列中的所有元素都检查一遍，从而找到优先级最高的元素。

假设优先级队列中元素个数为 n，这种实现下出队操作的复杂度是 O(n)，效率较低。

一般会使用**二叉堆**等更加高效的数据结构来实现优先级队列。

6.3 二叉堆

二叉堆是一种常见的堆，常用来实现优先级队列。二叉堆最早由 J. W. J. Williams 于 1964 年提出，作为支持堆排序的一种数据结构。

6.3.1 二叉堆的定义

二叉堆是父结点元素和子结点元素满足一定大小关系的**完全二叉树**。根据条件不同，可分为最小堆和最大堆。

- **最小堆**：如果完全二叉树 T 中的所有父子结点对都有父结点的元素不大于子结点的元素，则称 T 为最小堆。
- **最大堆**：如果完全二叉树 T 中的所有父子结点对都有父结点的元素不小于子结点的元素，则称 T 为最大堆。

（最小堆和最大堆的区别只在于父子结点元素之间的大小关系）

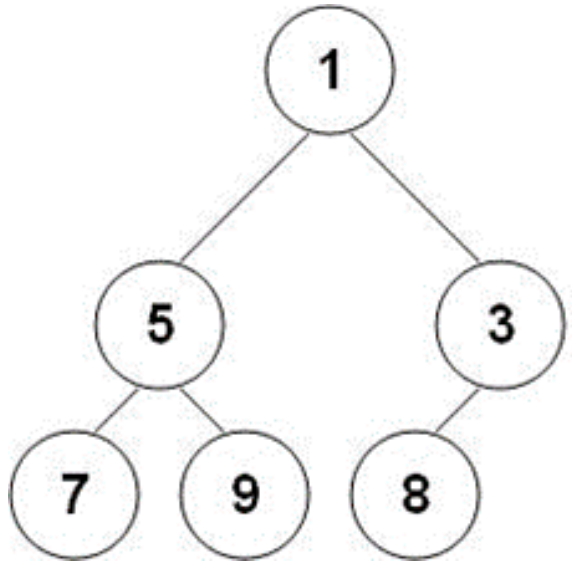
为简便起见，本章中的二叉堆（也包括其他种类的堆）都以**最小堆**为例进行讲解，最大堆的情况可类推得到。

6.3.1 二叉堆的定义

注意到二叉堆是一棵完全二叉树，可以将其保存在一个数组中（使用5.3.4节的约定，根结点是下标为1的元素），并具有以下性质：

- 结点 i 的左右子结点（如果存在）的下标分别为 2i 和 2i+1。
- 结点 i 的父结点（如果存在）的下标为 $\lfloor i/2 \rfloor$。

如图展示了一个最小堆。其中结点 1 的子结点为结点 5 和结点 3，结点 3 的子结点只有结点 8。可以验证，其中任意一个结点上的元素都不大于其子结点元素。



结点	1	5	3	7	9	8
下标	1	2	3	4	5	6

6.3.2 二叉堆的操作

二叉堆的基本操作是堆元素的**上调**和**下调**。（这里的“上”和“下”是指用一般习惯画出二叉堆的树表示后元素在调整过程中的走向）

在上调和下调操作的基础上，可实现堆元素的插入、删除，以及建堆操作。

设数组 h.data[] 中保存着二叉堆中的元素。在对二叉堆进行操作的过程中，可能会出现不满足二叉堆性质的时刻，为表述方便，仍用堆来称呼此时的状态。

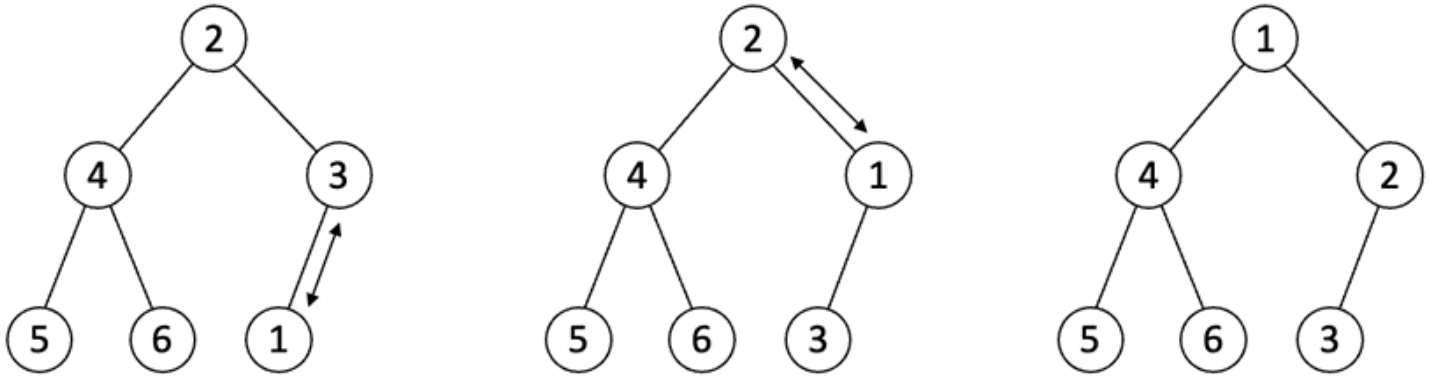
二叉堆的上调操作

上调操作：

如果堆中某结点 i 小于其父结点 p，此时可以交换结点 i 和结点 p 的元素，也就是把结点 i 沿着堆的这棵树往“上”调整。

此时，再看新的父结点与它的大小关系。重复该过程，直到结点 i 被调到根结点位置或者和新的父结点大小关系满足条件。

如图演示了一次二叉堆的上调操作的过程，元素1从开始的叶结点位置一直调整到了根结点。



二叉堆的上调操作

在实现时，可以通过以下方法避免交换，从而减少赋值操作的次数。SiftUp 先将结点 i 的元素保存在临时变量中，随着调整将父结点的元素往“下”移动，最后再将原来结点 i 的元素填入合适的位置。由此得到上调操作的算法如下：

算法6-1： 二叉堆的上调操作 SiftUp(*h*, *i*)
输入： 堆 *h* 和上调起始位置 *i*
输出： 上调后满足堆性质的*h*

```
1.  elem← h.data[i]
2.  while i > 1 且 elem < h[i / 2] do  //当前结点小于其父结点
3.  |  h.data[i] ← h.data[i / 2]  //将i的父结点元素下移
4.  |  i ← i / 2  //i指向原结点的父结点，即向上调整
5.  end
6.  h.data[i] ← elem
```

对于上调操作而言，循环的次数不会超过树的高度，因此时间复杂度为O(log n)。

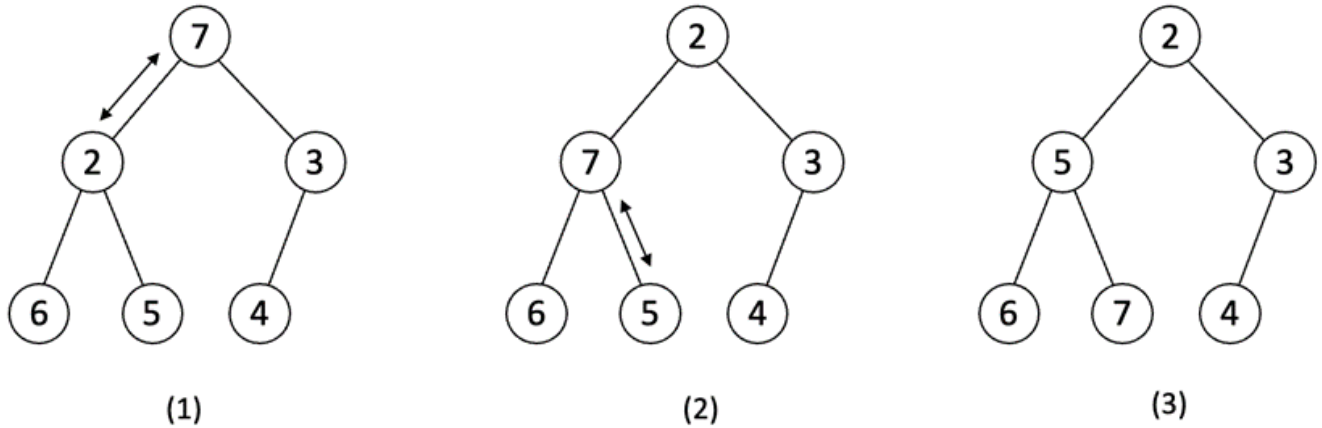
二叉堆的下调操作

下调操作：

如果堆中某结点 i 大于其子结点，则要将其向“下”调整。调整时需要注意，对于有两个子结点的情况，如果两个子结点均小于结点 i，交换时应选取它们中的较小者，只有这样才能保证调整之后三者的关系能够满足堆的性质。

重复该过程，直到结点 i 被调到叶结点位置或者和新的子结点大小关系满足条件。

如图演示了一次二叉堆的下调操作的过程，元素7从开始的根结点位置一直调整到了叶结点，注意每次要和左右子结点中值较小的元素进行交换。



二叉堆的下调操作

下调操作同样可以使用上调操作的方法来避免交换操作，使用该方法实现的下调操作算法如下。对于下调操作而言，循环的次数也不会超过树的高度，因此时间复杂度为O(log n)。

算法6-2： 二叉堆的下调操作 SiftDown(*h*, *i*)
输入： 堆 *h* 和下调起始位置 *i*
输出： 下调后满足堆性质的*h*

```
1.  last ← h.size  //这是最后一个元素的位置
2.  elem ← h.data[i]
3.  while true do
4.  | child ← 2i  //child当前是i的左孩子的位置
5.  | if child<last 且h.data[child+1]<h.data[child] then //如果i有右孩子并且右孩子更小
6.  | | child ← child + 1  //child更新为i的右孩子的位置
7.  | else if child>last  //如果i是叶子结点
8.  | | break  //已经调整到底，跳出循环
9.  | end
10. | if h.data[child] < elem then  //若较小的孩子比elem小
11. | | h.data[i] ← h.data[child]  //将较小的孩子结点上移
12. | | i ← child  //i指向原结点的孩子结点，即向下调整
13. | else  //若所有孩子都不比elem小
14. | | break  //则找到了elem的最终位置，跳出循环
15. | end
16. end
17. h.data[i] ← elem
```

二叉堆的插入操作

插入操作：

有了上调与下调两个基本操作后，堆的插入操作就可以实现为向堆中追加待插入的元素，然后用上调操作将其调整到合适的位置来完成堆的调整，时间复杂度同样是O(log n)。插入操作算法如下所示。

算法6-3： 二叉堆的插入操作 Insert(*h*, *x*)
输入： 堆 *h* 和待插入元素 *x*
输出： 将元素插入后的堆

1. *h.size* ← *h.size* + 1
2. *last* ← *h.size*
3. *h.data*[*last*] ← *x* //暂时将*x*放入最后一个元素的位置
4. SiftUp(*h*, *last*) //从最后一个位置上调

二叉堆的删除操作

删除操作：

删除操作所要提取的最小元素就是堆中的第一个元素，然后可以把堆中的最后一个元素挪到第一个位置，并通过下调操作将这个元素调整到合适的位置，时间复杂度是O(log n)。删除操作算法如下所示。

算法6-4： 二叉堆的删顶操作 ExtractMin(*h*)
输入： 堆 *h*
输出： *h*中的最小元，以及删除了最小元素后的堆*h*

1. *min_key* ← *h.data*[1] //这是将要返回的最小元
2. *last* ← *h.size* //这是删除前最后一个元素的位置
3. *h.size* ← *h.size* - 1
4. *h.data*[1] ← *h.data*[*last*] //暂时将删除前最后一个元素放入根的位置
5. SiftDown(*h*, 1) //从根结点下调
6. **return** *min_key*

二叉堆的朴素建堆操作

朴素**建堆**操作：

对于任意一组元素，可以通过逐个上调的方式把它们转化为一个堆，相当于依次插入堆中，算法如下所示。

算法6-5：二叉堆的朴素建堆操作 MakeHeapUp(*h*)
输入：存储在 *h* 中的数据
输出：满足堆性质的堆*h*

```
1.  last ← h.size  //这是最后一个元素的位置
2.  for i ← 2 to last do
3.  |  SiftUp(h, i)
4.  end
```

使用上述方法进行建堆，堆中后一半的元素进行上调时都可能需要O(log n)的时间，因此总的时间复杂度是O(n log n)。

二叉堆的快速建堆操作

快速建堆操作：

也可以用逐个下调的方式把它们转化为一个堆。由于可以把叶结点跳过，因此是从最后一个有叶子的结点（大概是一半的位置）开始操作，算法如下所示。

算法6-6： 二叉堆的快速建堆操作 MakeHeapDown(*h*)
输入： 存储在 *h* 中的数据
输出： 满足堆性质的堆*h*

```
1. last ← h.size  //这是最后一个元素的位置
2. for i ← last/2 downto 1 do  // last/2是最后一个元素的父结点的位置
3. |  SiftDown(h, i)
4. end
```

和MakeHeapUp相比，这样做的好处是在MakeHeapDown中有将近一半结点的下调操作只需要O(1)的时间，因此更加高效。具体分析如下：

$$\sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} O(k) = O\left(n \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k}\right) = O\left(n \sum_{k=0}^{\infty} \frac{k}{2^k}\right) = O(n)$$

下调步数为k的层结点数为$\frac{n}{2^k}$

6.4 多叉堆

多叉堆，也称为 d 堆（d-ary heap 或 d heap），是二叉堆的推广形式。除边界情况外，二叉堆的每个结点有两个子结点，而多叉堆则有 d 个子结点。多叉堆由 Donald B. Johnson 于 1975 年提出。

与二叉堆相似，多叉堆也可以用数组来保存堆中的元素，元素的下标之间存在数学关系。从数学公式的简洁优雅考虑，多叉堆用数组表示时下标一般从0开始。在多叉堆的数组表示中，元素 0 是根结点，元素 1~d 是根结点的子结点，而紧接着的 d^2 个元素是根结点的孙子结点，以此类推。

于是，可以得到多叉堆结点之间的下标关系如下：

- 结点 i 的 d 个子结点的下标分别为 di+j，其中1⩽j⩽d。
- 结点 i 的父结点的下标为 $\lfloor (i-1)/d \rfloor$。

多叉堆的上调操作

可以根据以上性质扩展二叉堆的 SiftUp 操作，使得它支持多叉堆。

算法6-7：多叉堆的上调操作 SiftUpD(*h*, *d*, *i*)
输入：堆 *h*、堆的分叉数 *d* 和上调起始位置 *i*
输出：上调后满足*d*叉堆性质的*h*

```
1. elem ← h.data[i]
2. while i > 0 and elem < h.data[(i−1) / d] do
3. |  h.data[i] ← h.data[(i−1) / d]
4. |  i ← (i−1) / d
5. end
6. h.data[i] ← elem
```

多叉堆的下调操作

同理，也可以扩展二叉堆的 SiftDown 操作，要注意向下比较时要看所有的子结点。

算法6-8：多叉堆的下调操作 SiftDownD(*h*, *d*, *i*)
输入：堆 *h*、堆的分叉数 *d* 和下调起始位置 *i*
输出：将堆 *h* 中的元素下调以满足堆性质

```
1.  last ← h.size - 1  //这是最后一个元素的位置
2.  elem ← h.data[i]
3.  while true do
4.  | child ← i
5.  | for k ← 1 to d do  //找所有孩子中最小的
6.  | | if d×i+k ≤ last 且h.data[d×i+k]<h.data[child] then
7.  | | | child ← d×i+k  //child更新为更小的孩子的位置
8.  | | end
9.  | end
10. | if child = i then  //前面for循环未执行，i是叶结点
11. | | break  //已经调整到底，跳出循环
12. | end
13. | if h.data[child]<elem then  //若最小的孩子比elem小
14. | | h.data[i] ← h.data[child]  //将最小的孩子结点上移
15. | | i ← child  //i指向原结点的孩子结点，即向下调整
16. | else  //若所有孩子都不比elem小
17. | | break  //则找到了elem的最终位置，跳出循环
18. | end
19. end
20. h.data[i] ← elem
```

多叉堆的分析

使用类似的方法可以分析多叉堆操作的复杂度，其中 SiftUpD 为 $O(\log_d n)$，SiftDownD 为 $O(d \log_d n)$ 。

与二叉堆的对应操作相比，多叉堆的上调操作性能会更好一些，而下调操作则会变差。对于建堆操作，可以通过数学证明其复杂度仍然为 O(n)。

多叉堆可用于上调操作比下调操作频繁的应用场景，包括图论中的很多常用算法。此外，与二叉堆相比，多叉堆有更好的高速缓存特性，因此实际使用中能够在现代计算机系统结构上取得更好的运行效率。

6.6 优先级队列应用：哈夫曼树的构建

第5.5节讲了哈夫曼树。在构建哈夫曼树的过程中，算法CreateHuffmanTree会持续地从一个二叉树集合中取出带权路径长度最小和次小的两棵二叉树，并把合并后的树加回到集合里。

这个过程是优先级队列的典型应用，可以将二叉树的带权路径长度定为优先级（带权路径长度越小优先级越高），分别使用ExtractMin和Insert来完成从集合取出最小、次小以及加回到集合的操作。

为了提高算法的效率，通常使用**二叉堆**作为优先级队列的实现。由于二叉堆的插入和删除元素操作都是O(logn)的时间复杂度，建堆的时间复杂度是O(n)，整个算法在建堆之后一共要进行2n-2次删除和n-1次插入操作，因此总的复杂度为O(n log n)。

用二叉堆实现的优先级队列能够高效地支持诸如构建哈夫曼树这样的算法。

6.7.1 双端优先级队列

有的应用需要同时支持获取集合中的最大和最小元素的操作，我们把这样的数据结构称为双端优先级队列。与普通的优先级队列相比，双端优先级队列的ADT增加了ExtractMax和PeekMax操作。

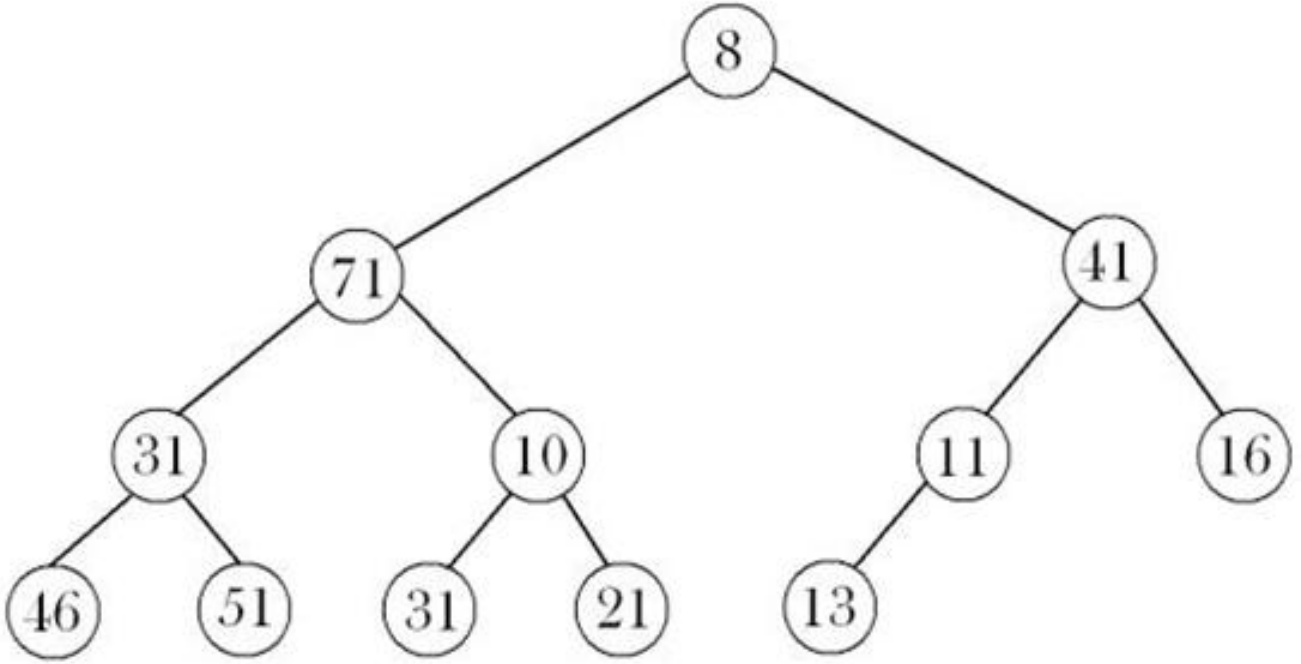
双端优先级队列**可以**使用一个最大堆和一个最小堆配合来实现。在实现时，这两个堆中的元素要分别增加指向另外一个堆中对应元素的指针域，并在元素插入或者删除时做好相应的维护。第12章要介绍的自平衡二叉搜索树（如红黑树、AA树、伸展树等）也可以用来实现双端优先级队列。

此外，双端优先级队列也有一些专用的方法，下面简单介绍**最小最大堆**。

6.7.1 双端优先级队列

最小最大堆与二叉堆的基本原理相似，本身是一棵完全二叉树的结构。

它结合了最小堆和最大堆的特性：位于**奇数层**（1、3、5、……）的结点小于它的所有后代结点，而位于**偶数层**（2、4、……）的结点大于它的所有后代结点。如图所示。同理，还可以定义最大最小堆。



6.7.1 双端优先级队列

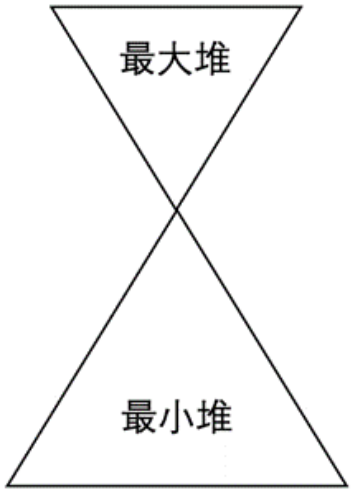
- **插入操作Insert**：向最小最大堆插入元素时，先将待插入的元素置于数组末尾，然后根据情况上调该元素在堆中的位置。调整时，与二叉堆的区别在于二叉堆每次都和上一层的元素（父结点）进行比较，而最小最大堆在和父结点比较一次之后会进行连续的跨层比较（也就是和祖父结点进行比较），直到根结点。
- **查询最小元素PeekMin**：最小最大堆的最小元素一定是根结点。
- **查询最大元素PeekMax**：最小最大堆的最大元素一定是根结点的两个子结点之一。
- **删除最小元素ExtractMin**：由于最小元素在最小最大堆的根结点，在删除时将数组最后一个元素交换至根结点位置，然后下调该元素在堆中的位置。在下调时，需要注意挑选交换的候选元素时要往下看两层。
- **删除最大元素ExtractMax**：与ExtractMin类似，最大元素是根结点的两个子结点之一，将数组最后一个元素交换至最大元素位置，然后进行下调操作。

6.7.2 对顶堆

普通的堆只能解决最大或者最小的问题，对于第k大（或者第k小）这样的问题，可以用两个堆配合解决。

以获取当前所有元素中第k小的元素为例，可以使用一个最大堆维护当前最小的k个元素，用一个最小堆维护剩下的元素。

如果将最大堆和最小堆看作两个三角形，上述方法好像是把两个三角形顶点对顶点扣在一起，因此被形象地称为**对顶堆**，如图所示。



6.7.2 对顶堆

下面分别讨论需要支持的三种操作。

- **获取第k小的元素**：根据两个堆的定义，最大堆的堆顶元素就是所有元素中第k小的元素。
- **插入元素**：如果待插入元素小于最大堆的堆顶元素，那么它应该放在最大堆中，此时将最大堆的堆顶元素取出并插入到最小堆中，然后将带插入元素插入到最大堆中。否则，直接将待插入元素插入到最小堆中。
- **删除元素**：如果待删除元素在最大堆中，将其删除的同时把最小堆的堆顶元素补充到最大堆中。否则，直接从最小堆中删除元素。

6.8 应用场景：离散事件模拟

离散事件模拟是一种重要的计算机模拟技术，在其关注的离散事件系统中，系统的所有状态只在特定的、离散的时间点发生变化，而这些变化都可被抽象为一系列定义好的事件。离散事件模拟被广泛应用于各种领域，如工业制造、交通系统、医疗保健、金融等。它可以分析并预测系统的行为，以便优化系统的性能、改进流程、减少成本和风险。

离散事件模拟有三个重要的组成部分：**时钟**、**状态**和**事件列表**。时钟表示系统当前的模拟进度，它可以对应到现实时间（如把每秒作为一个模拟的单位），也可以只是逻辑的概念；状态反映了整个系统的所有需要关注的属性；而事件列表中包含了所有将来要发生的事件和它们会发生的时间点。

6.8 应用场景：离散事件模拟

在每个模拟时钟的时间点上，模拟器都需要从事件列表中取出所有当前时刻发生的事件，并根据这些事件对应修改系统的状态，向事件列表中增加新的（会在将来发生的）事件。因此，**优先级队列**非常适合用来维护事件列表：只要以“事件发生的时间点”作为元素的优先级维护队列，并不断出队、处理事件，直到事件列表为空，或者达到了预定的模拟时间点即可停止。

以简单的仓储管理为例，假设需要建设一个容量有限的仓库用于保存物品，随时可能有卡车前来装卸物品。每辆车到来的时间、可以运来或者运走的量都是随机的，但遵循某种统计分布。通过离散事件模拟，可以获得在不同容量下仓库的预期使用效率、物品的运输效率，以实现建设成本的最佳利用。在这一系统中，时钟可以是离散化的现实时间（例如以一分钟为模拟单位），状态是仓库的剩余容量和已经成功运输的货物量，而事件是后续可能到来的卡车和容量（可以来自现实的统计数据，也可以在模拟过程中随机生成）。

小结

- **优先级队列**是一种常用的数据结构，提供了比栈和队列更加丰富的处理顺序，可用于很多真实场景。
- **堆**是优先级队列的一类实现方式，由基本的堆性质衍生出了多种不同的堆的设计，它们在堆的基本操作上具有不同的复杂度。这其中，**二叉堆**是一种隐式数据结构，它将元素的逻辑结构蕴含在存储结构中，避免了额外的指针域空间开销，实现起来也比较简洁，因此得到了广泛的应用。

谢谢观看