

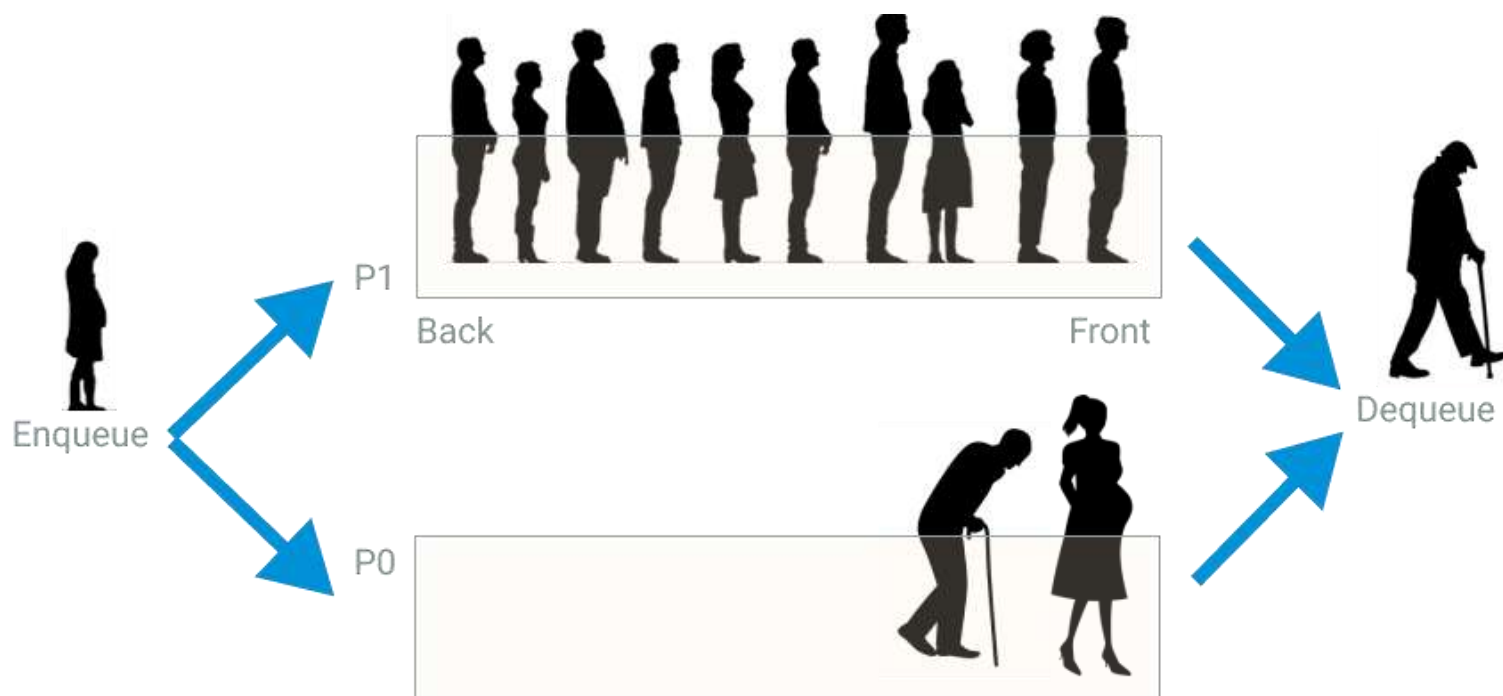
树的应用

优先级队列

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。



优先级队列

- 每次从队列中取出具有最高优先级的元素，这种队列就是优先级队列（priority queue），也称为优先权队列或优先队列。
- 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。

优先级队列的抽象数据类型定义

优先级队列是0个或多个元素的集合，每个元素都有一个与之关联的优先级。

对于优先级队列主要的操作有：

- (1) 查找优先级最高的值；
- (2) 出队优先级最高的值；
- (3) 入队一个任意优先级的值

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

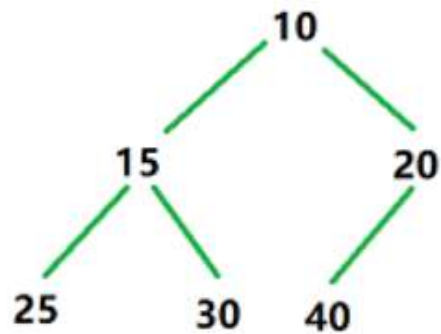
- 假设数值越小表明该元素的优先级越高，如下例所示的优先级队列中，最先出队的应该是元素10。

优先级	20	50	40	10	30
-----	----	----	----	----	----

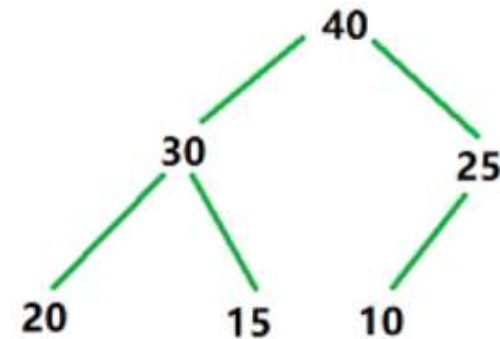
- 假设用无序表表示优先级队列，入队新元素则可以直接放在30之后，入队操作的时间复杂度可以为 $O(1)$ ，但查找和出队操作都为 $O(n)$ 。
- 如果用有序表表示优先级队列，队头元素就是优先级最高的元素，查找和出队操作效率可以为 $O(1)$ ，但入队操作的时间复杂度为 $O(n)$ 。
- 用“堆”实现优先级队列，此时入队和出队操作的时间效率都为 $O(\log_2 n)$ 。

二叉堆

- 二叉堆，简称堆（heap），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。
- 堆是**具有下列性质的完全二叉树**：每个结点的值都**小于或等于**其左右孩子结点的值，称为**小顶堆**（小根堆），或者每个结点的值都**大于或等于**其左右孩子结点的值称为**大顶堆**（大根堆）。



小根堆



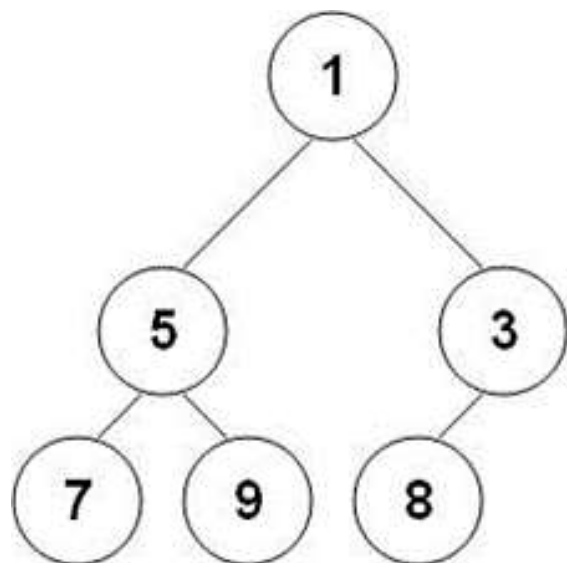
大根堆







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 $\lfloor i/2 \rfloor$ 。

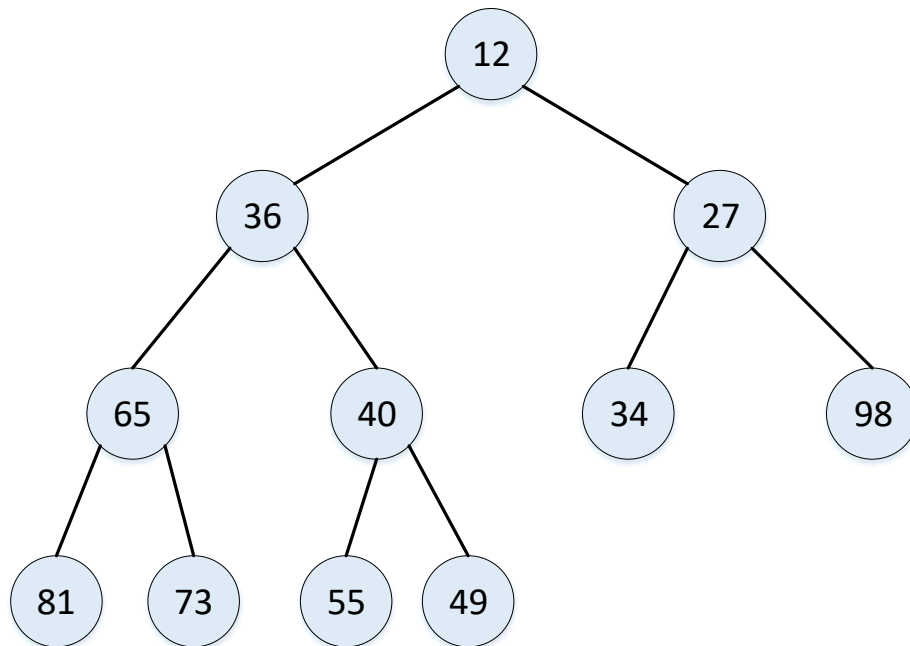
如图展示了一个最小堆。其中结点 1 的子结点为结点 5 和结点 3，结点 3 的子结点只有结点 8。可以验证，其中任意一个结点上的元素都不大于其子结点元素。



结点						
下标	1	2	3	4	5	6

二叉堆

- 如下图所示的完全二叉树是一个小顶堆，也可以说序列(12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49)是一个小顶堆。



二叉堆

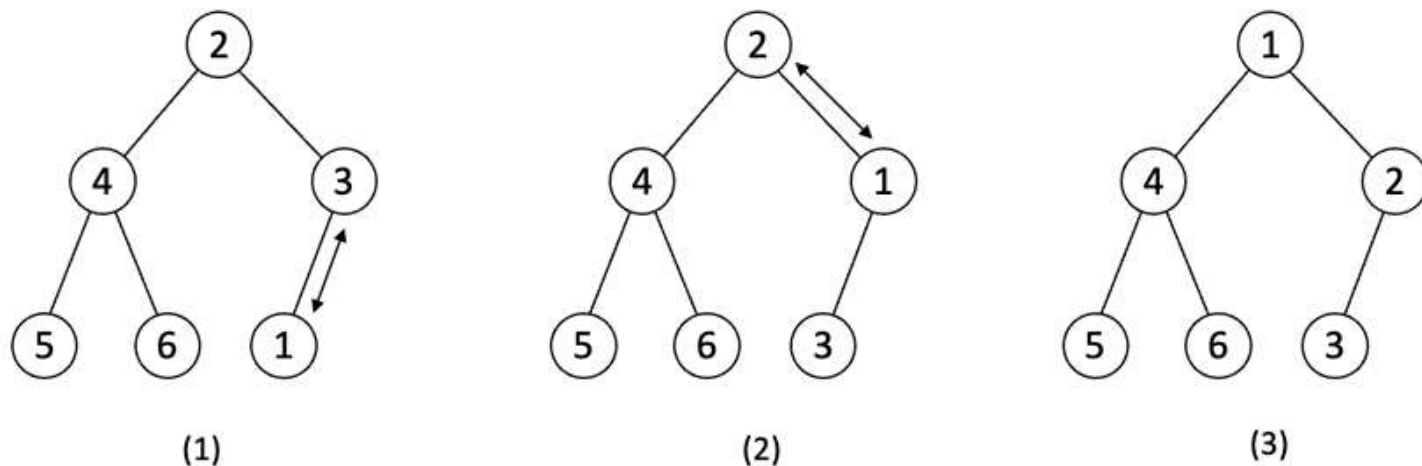
- 堆主要用于实现优先级队列，优先级队列的操作特点是每次出队的元素是优先级最高的元素，在无序表结构下出队算法的性能为 $O(n)$ ，而利用堆实现优先级队列，入队和出队算法性能都为 $O(\log_2 n)$ 。
 - 在第8章中，将介绍如何利用堆实现堆排序。
- 二叉堆的基本操作是堆元素的**上调**和**下调**（这里的“上”和“下”是指用一般习惯画出二叉堆的树表示后元素在调整过程中的走向）。
- 在上调和下调操作的基础上，可实现堆元素的插入、删除，以及建堆操作。下面以小顶堆为例介绍。

二叉堆的上调操作 (SiftUp)

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。



设数组 $h.data[]$ 中保存着二叉堆中的元素。

在实现时，为避免多次交换，可以先将结点 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，实现插入元素的“上调”（SiftUp函数）。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. $elem \leftarrow h.data[i]$
2. **while** $i > 1$ 且 $elem < h[i / 2]$ **do** //当前结点小于其父结点
3. | $h.data[i] \leftarrow h.data[i / 2]$ //将 i 的父结点元素下移
4. | $i \leftarrow i / 2$ // i 指向原结点的父结点，即向上调整
5. **end**
6. $h.data[i] \leftarrow elem$

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 $O(\log_2 n)$ 。

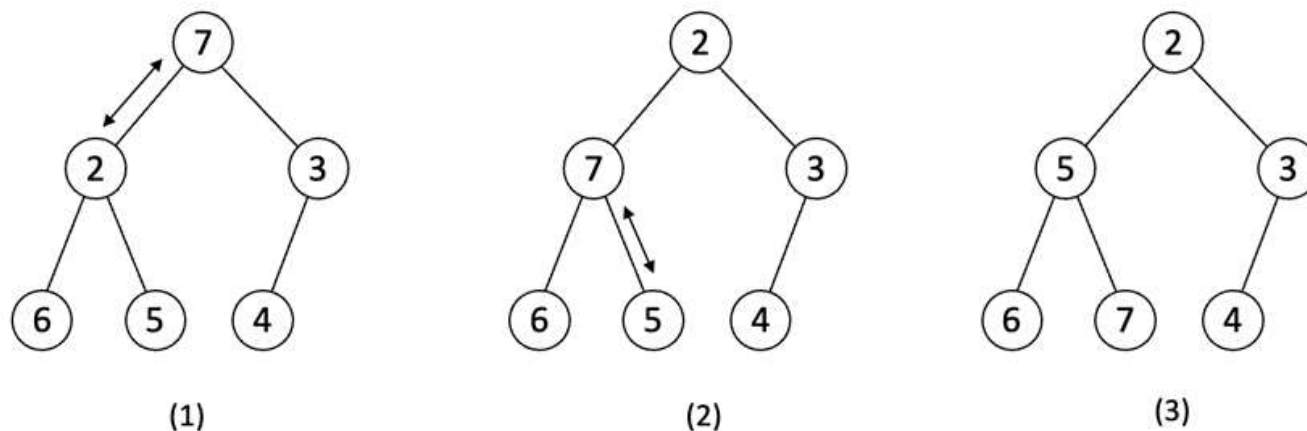
二叉堆的下调操作 (SiftDown)

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。



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 $O(\log_2 n)$ 。

算法6-2：二叉堆的下调操作 $\text{SiftDown}(h, i)$

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.  $last \leftarrow h.size$  //这是最后一个元素的位置
2.  $elem \leftarrow h.data[i]$ 
3. while true do
4. |  $child \leftarrow 2i$  //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. | |  $child \leftarrow child + 1$  //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. | |  $h.data[i] \leftarrow h.data[child]$  //将较小的孩子结点上移
12. | |  $i \leftarrow child$  // $i$ 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.  $h.data[i] \leftarrow elem$ 
```

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 $\text{Insert}(h, x)$

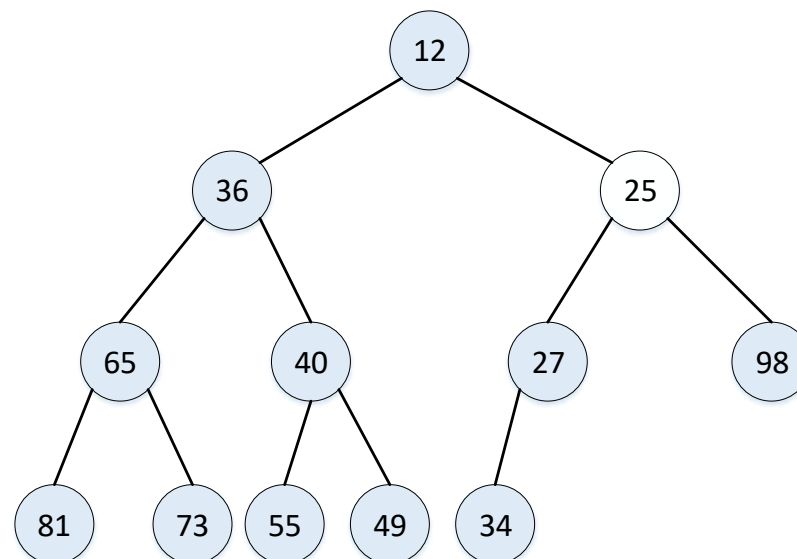
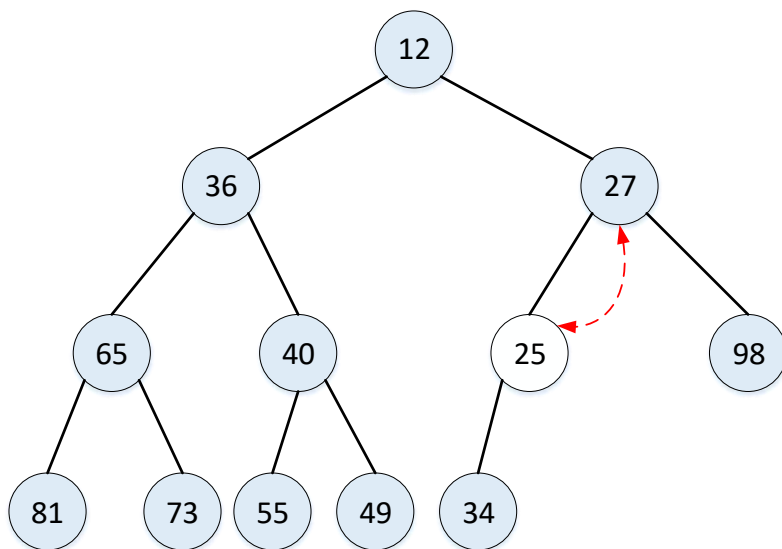
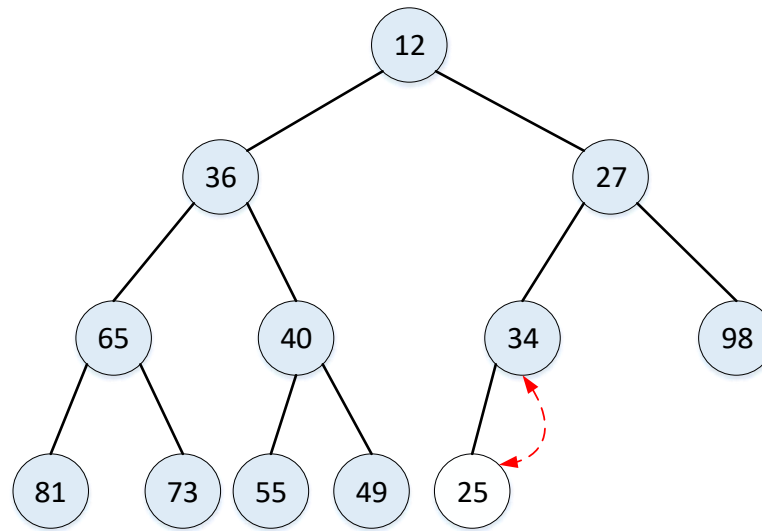
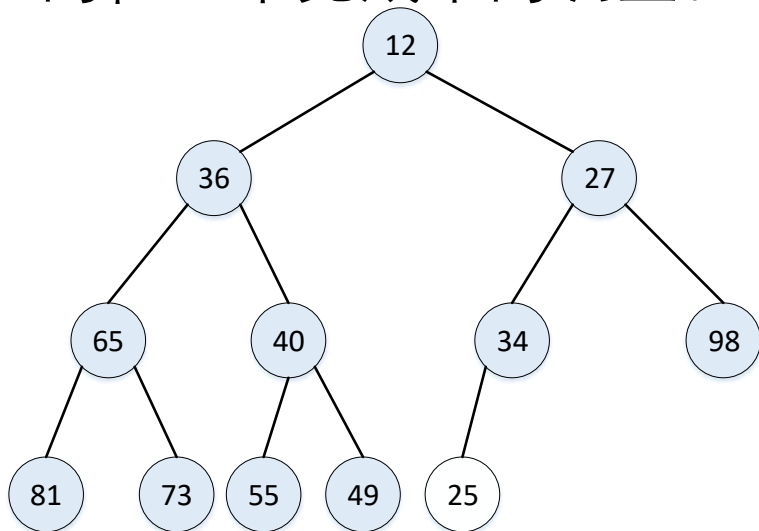
输入：堆 h 和待插入元素 x

输出：将元素插入后的堆

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

二叉堆的插入操作

堆的插入操作为向堆中追加待插入的元素，可以用“上调”操作将其调整到合适的位置来完成堆的调整。



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 $O(\log n)$ 。堆的删除操作的算法如下：

算法6-4：二叉堆的删顶操作 $\text{ExtractMin}(h)$

输入：堆 h

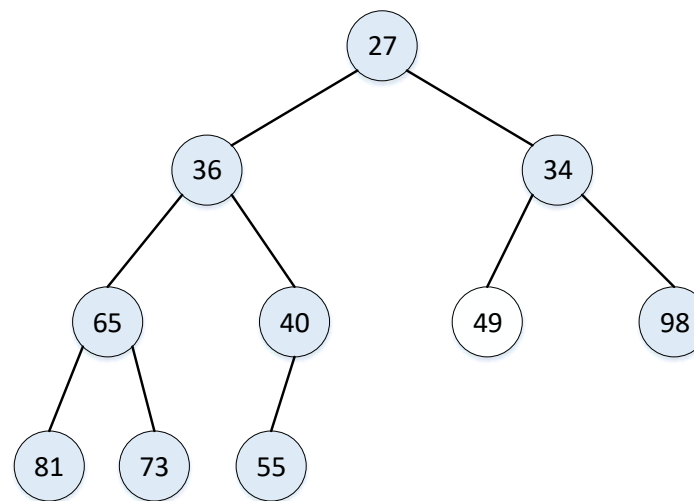
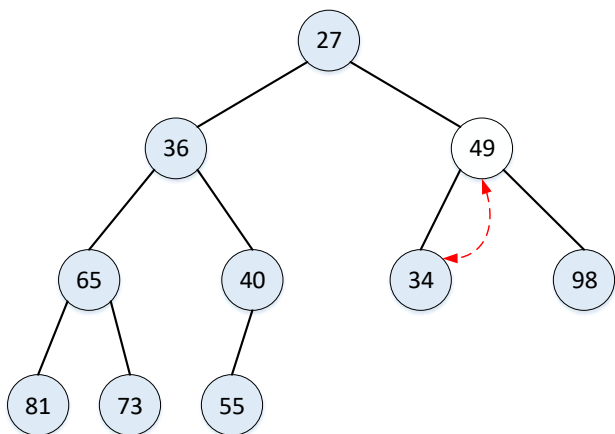
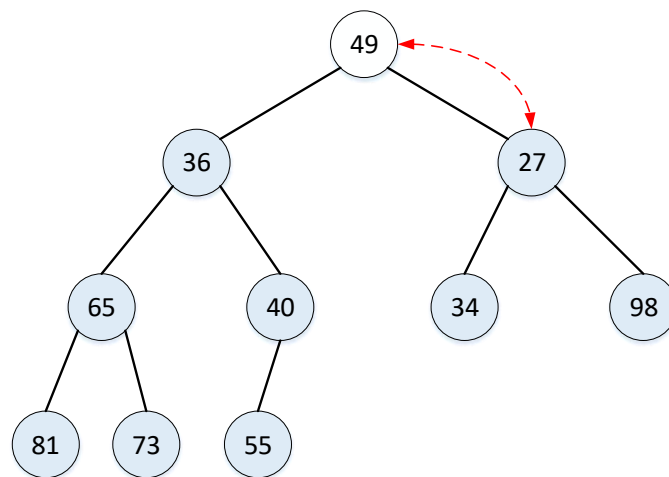
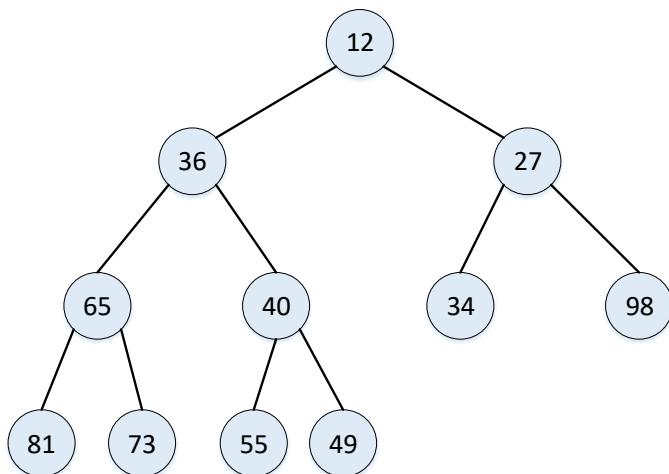
输出： h 中的最小元，以及删除了最小元素后的堆 h

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

对于“下调”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 $O(\log_2 n)$ 。

二叉堆的删顶操作

删除操作所要提取的最小元素就是堆中的第一个元素，**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。



二叉堆的删顶操作

删除操作算法如下，对于“下调”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 $O(\log_2 n)$ 。

算法6-4：二叉堆的删顶操作 $\text{ExtractMin}(h)$

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. $\text{min_key} \leftarrow h.\text{data}[1]$ //这是将要返回的最小元
2. $\text{last} \leftarrow h.\text{size}$ //这是删除前最后一个元素的位置
3. $h.\text{size} \leftarrow h.\text{size} - 1$
4. $h.\text{data}[1] \leftarrow h.\text{data}[\text{last}]$ //暂时将删除前最后一个元素放入根的位置
5. $\text{SiftDown}(h, 1)$ //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 $\text{MakeHeapUp}(h)$

输入：存储在 h 中的数据

输出：满足堆性质的堆 h

1. $last \leftarrow h.size$ //这是最后一个元素的位置
2. **for** $i \leftarrow 2$ **to** $last$ **do**
3. | $\text{SiftUp}(h, i)$
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 $O(\log_2 n)$ 的时间，因此总的时间复杂度是 $O(n \log_2 n)$ 。

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 $\text{MakeHeapDown}(h)$

输入：存储在 h 中的数据

输出：满足堆性质的堆 h

1. $last \leftarrow h.size$ //这是最后一个元素的位置
2. **for** $i \leftarrow last/2$ **downto** 1 **do** // $last/2$ 是最后一个元素的父结点的位置
3. | $\text{SiftDown}(h, i)$
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：

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



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 $O(\log n)$ 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。

多叉堆(自学)

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 $1 \leq j \leq d$ 。
- 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 $\text{SiftUpD}(h, d, i)$

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. $elem \leftarrow h.data[i]$
2. **while** $i > 0$ **and** $elem < h.data[(i-1)/d]$ **do**
3. | $h.data[i] \leftarrow h.data[(i-1)/d]$
4. | $i \leftarrow (i-1)/d$
5. **end**
6. $h.data[i] \leftarrow elem$

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.  $last \leftarrow h.size - 1$  //这是最后一个元素的位置
2.  $elem \leftarrow h.data[i]$ 
3. while true do
4.    $child \leftarrow i$ 
5.   for  $k \leftarrow 1$  to  $d$  do //找所有孩子中最小的
6.     if  $d \times i + k \leq last$  且  $h.data[d \times i + k] < h.data[child]$  then
7.        $child \leftarrow d \times i + k$  //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.     $h.data[i] \leftarrow h.data[child]$  //将最小的孩子结点上移
15.     $i \leftarrow child$  //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.  $h.data[i] \leftarrow elem$ 
```


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。