计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

俞勇、张铭、陈越、韩文弢

上海交通大学、北京大学、浙江大学、清华大学

第 6 章

优先级队列

韩文弢

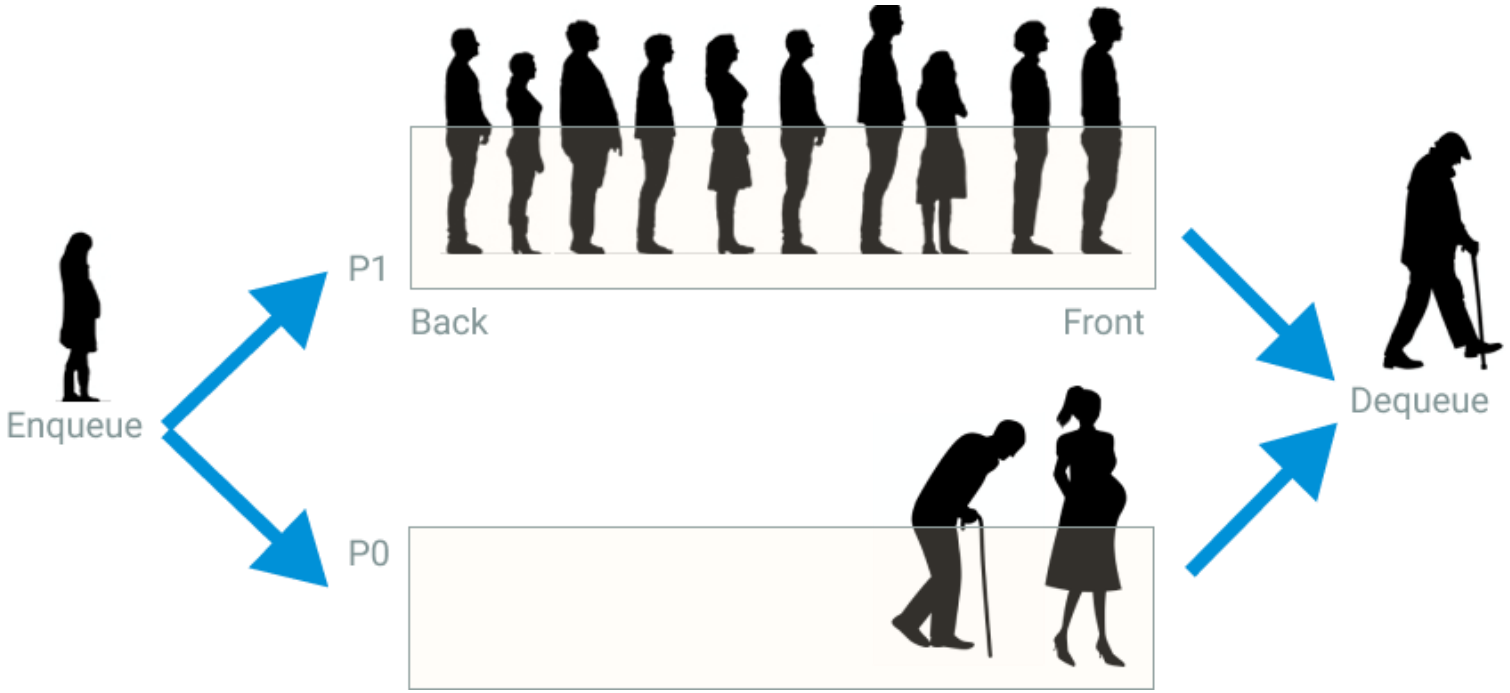
清华大学

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。

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

ExtractMax(*pq*): 从优先级队列 *pq* 中删除优先级最高（也就是值最大）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）

PeekMax(*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

优先级队列的逻辑结构**可以**用线性结构来表达。

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。

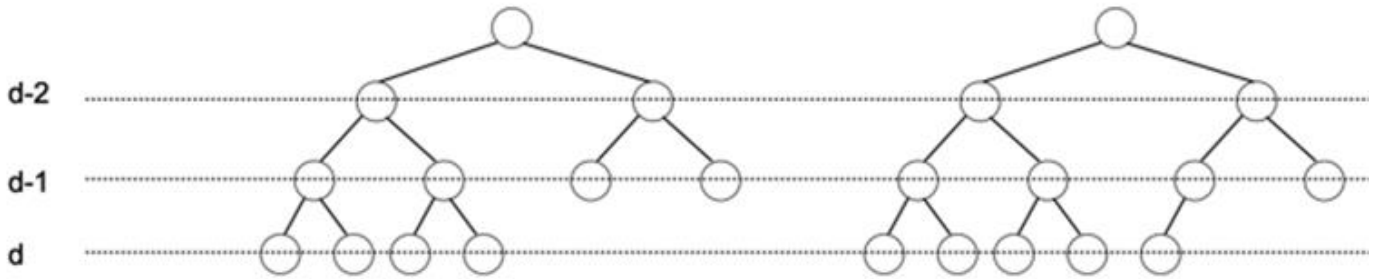
假设优先级队列中元素个数为 n，这种实现下出队操作的复杂度是 O(n)，效率较低。

一般会使用**二叉堆**等更加高效的数据结构来实现（表达）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。

- **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**。



高度$d > 3$的完全二叉树基本形态

(1) 从第1层到第$d-2$层全是度为2的中间结点

(2) 第d层的结点都是叶结点，度为0

(3) 在第$d-1$层，各结点的度从左向右单调非递增排列，同时度为1的结点要么没有，要么只有一个且该结点的左子树非空

多选题 1分

设置

对于完全二叉树，下面哪些描述是正确的？

A 含n个结点的所有二叉树中，完全二叉树的高度最小

B 如果完全二叉树的左子树非完美，则右子树一定完美，反之亦然

C 如果完全二叉树有101个结点，则树中一定没有度为1的结点

D 顺序存储n个结点的二叉树，完全二叉树需要的空间最小

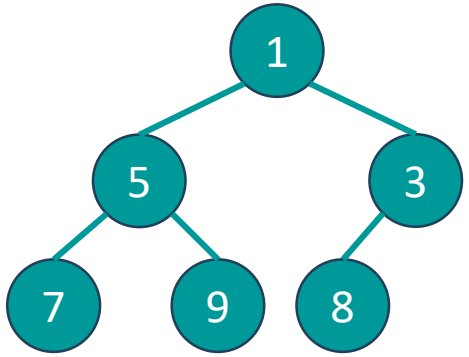
提交

6.3.1 二叉堆的定义

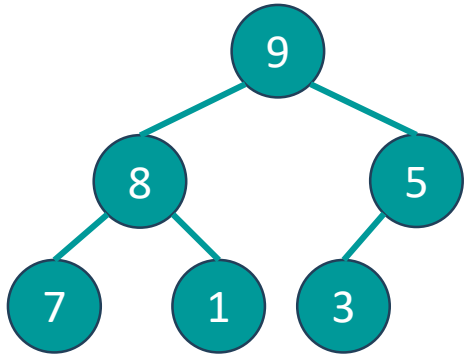
二叉堆是父结点元素和子结点元素满足一定大小关系的**完全二叉树**。根据条件不同，可分为最小堆和最大堆。

- **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。

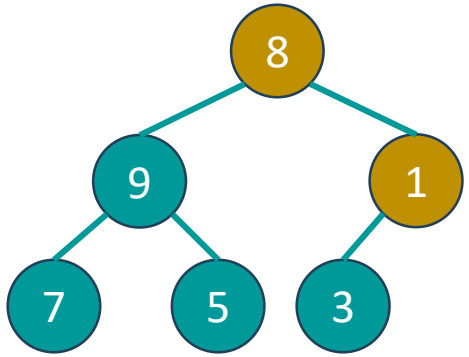
（最小堆和最大堆的区别只在于父子结点元素之间的大小关系）



最小堆



最大堆



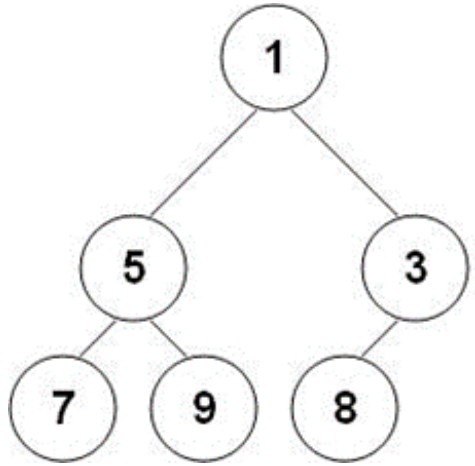
非最大堆、也非最小堆

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。



堆的逻辑结构

结点	1	5	3	7	9	8
下标	1	2	3	4	5	6

堆的存储结构

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。

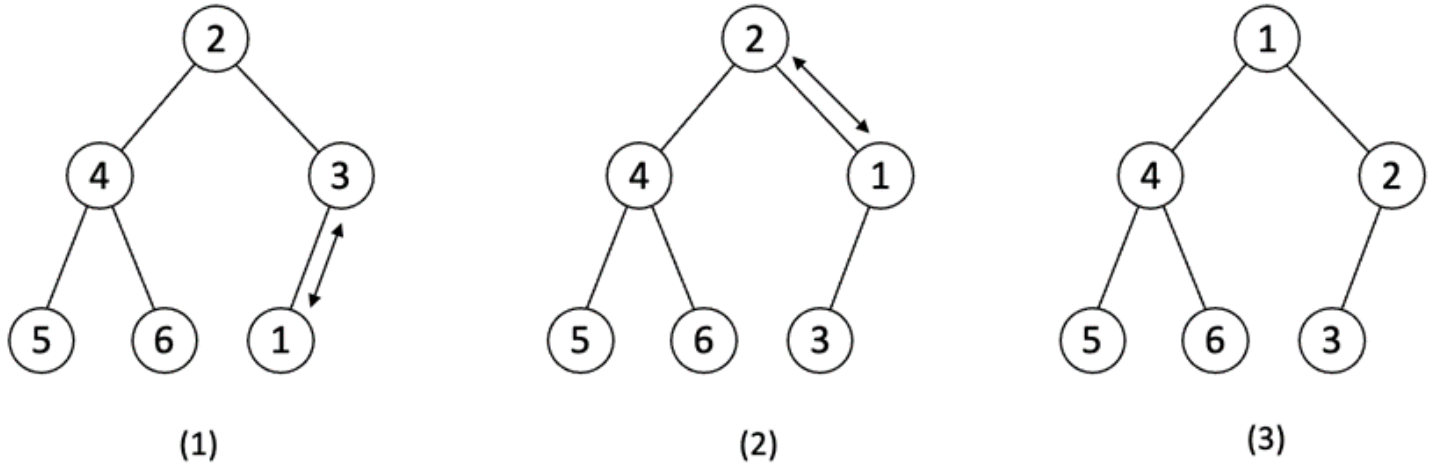
设数组 h.data[] 中保存着二叉堆中的元素。在对二叉堆进行操作的过程中，可能会出现不满足二叉堆性质的时刻，为表述方便，仍用堆来称呼此时的状态。

二叉堆的上调操作

上调(siftup)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
```

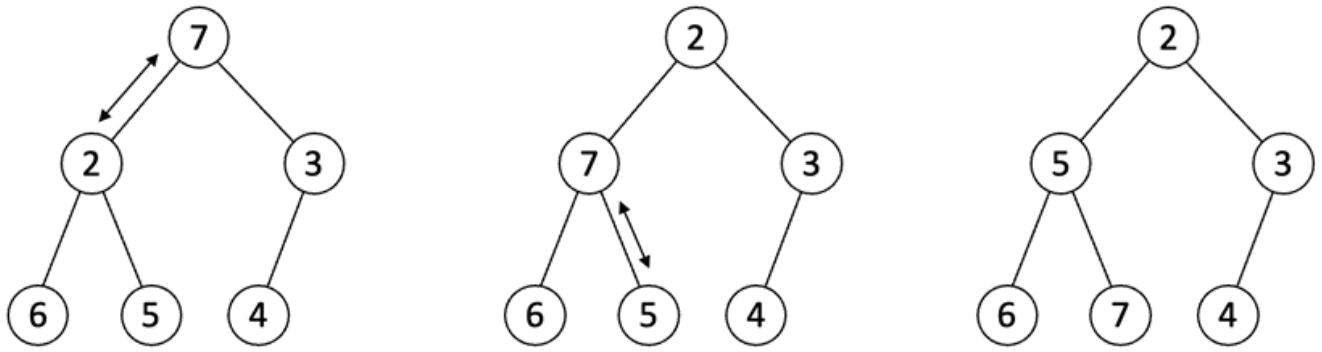
对于上调操作而言，循环的次数不会超过树的高度，因此时间复杂度为O(log n)。

二叉堆的下调操作

下调(siftdown)操作：

- 如果堆中某结点 i 大于其子结点，则要将其向“下”调整。
- 如果结点有两个子结点，且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(思考)。
- 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 //堆的长度加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*中的最小元素，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素
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*

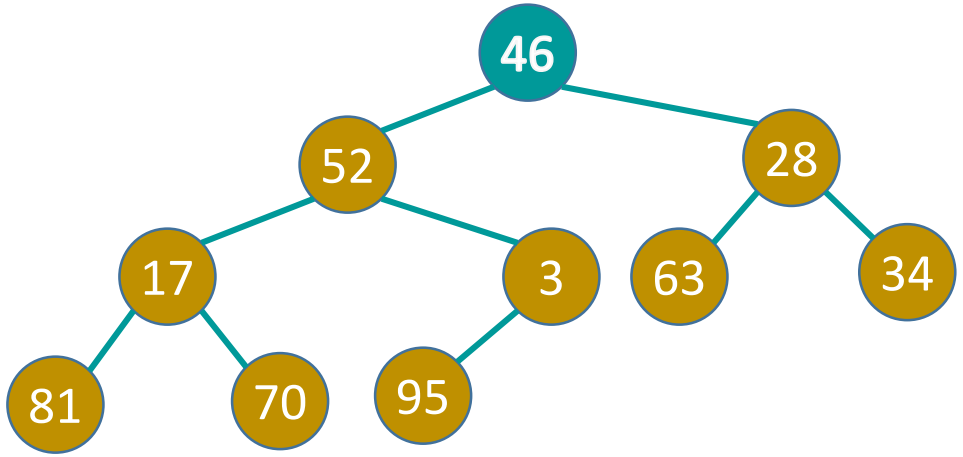
```
1. last ← h.size  //这是最后一个元素的位置
2. for i ← 2 to last do   //从第二个元素开始，依次上调
3. |  SiftUp(h, i)
4. end
```

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)。

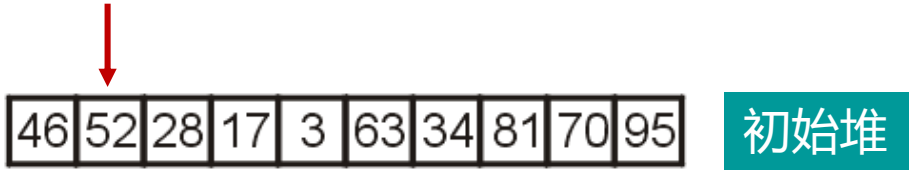
最大堆的朴素建堆操作

46	52	28	17	3	63	34	81	70	95

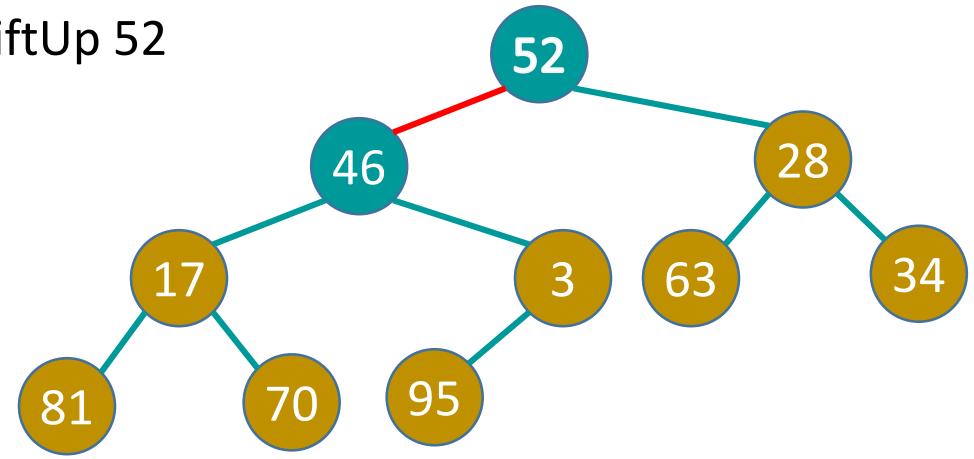
初始堆



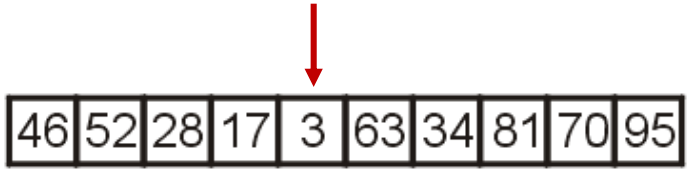
最大堆的朴素建堆操作



(1) SiftUp 52

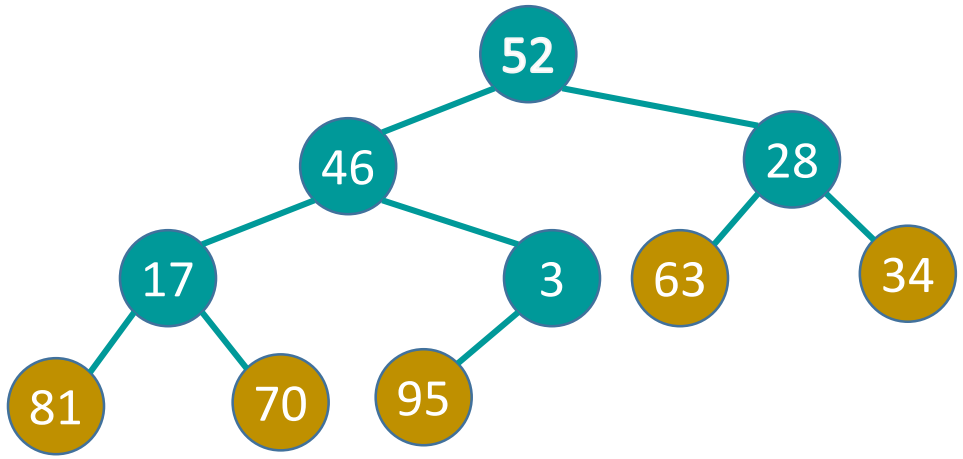


最大堆的朴素建堆操作



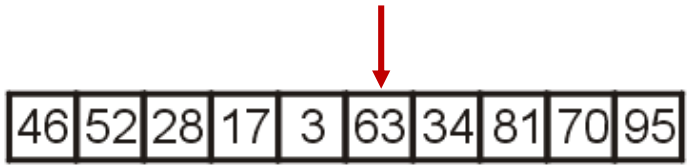
初始堆

(2) SiftUp 28
(3) SiftUp 17
(4) SiftUp 3



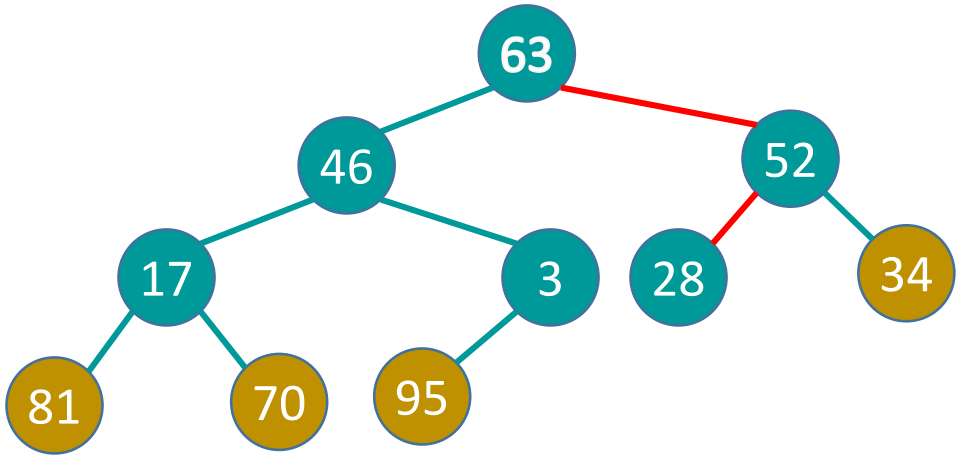
无交换或上移!

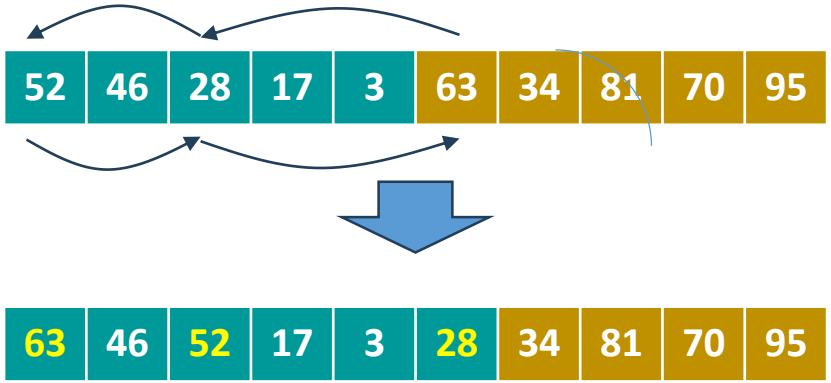
最大堆的朴素建堆操作



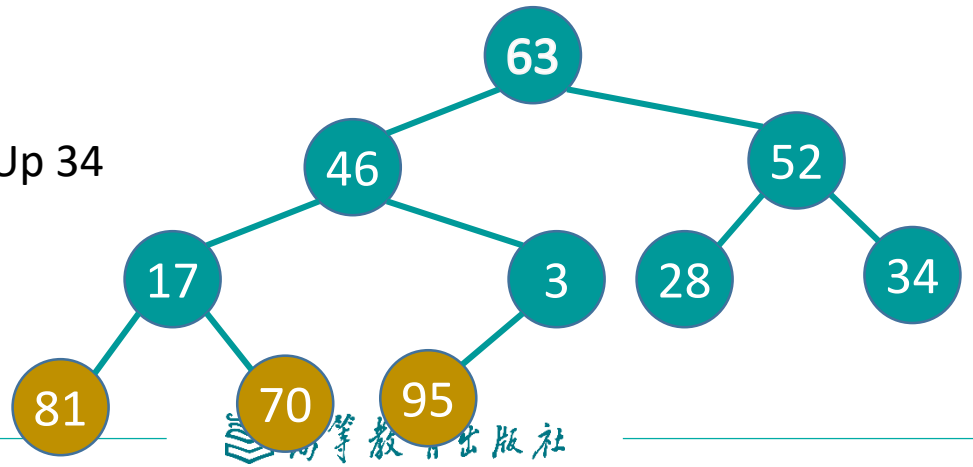
初始堆

(5) SiftUp 63

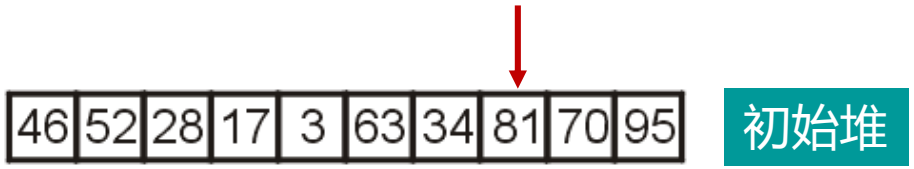




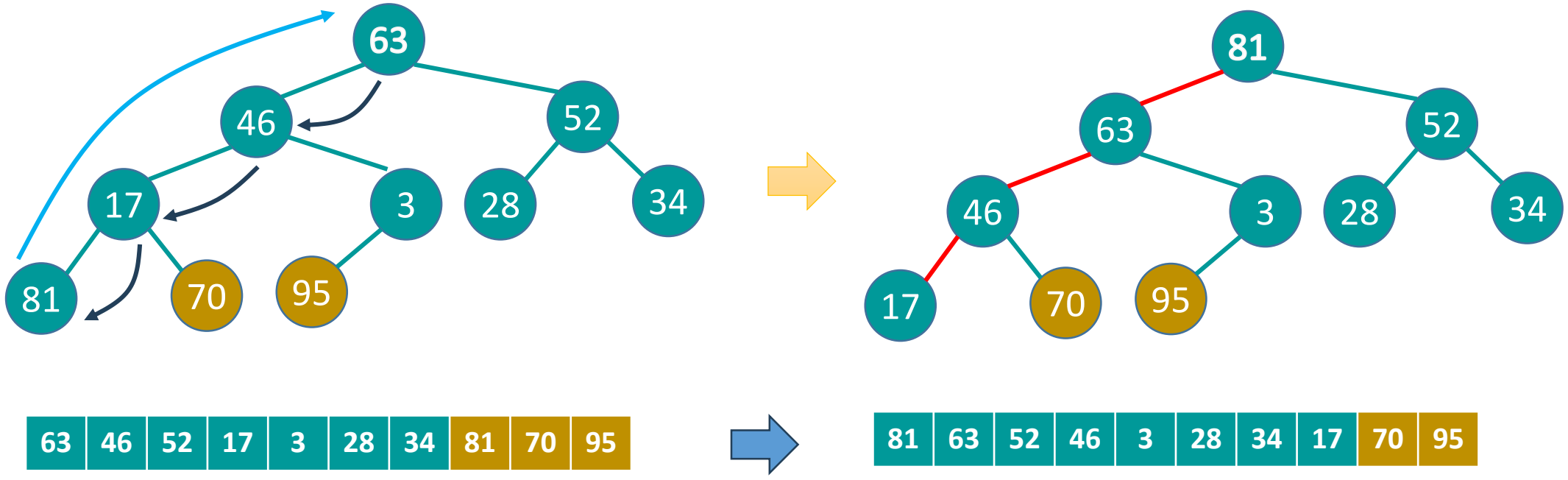
(6) SiftUp 34



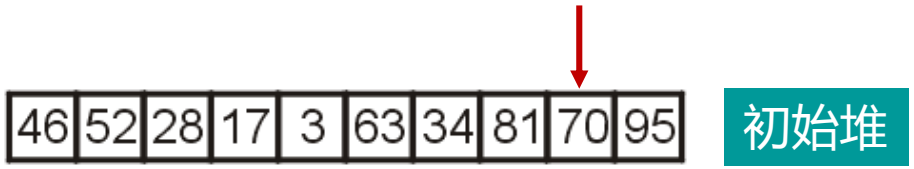
最大堆的朴素建堆操作



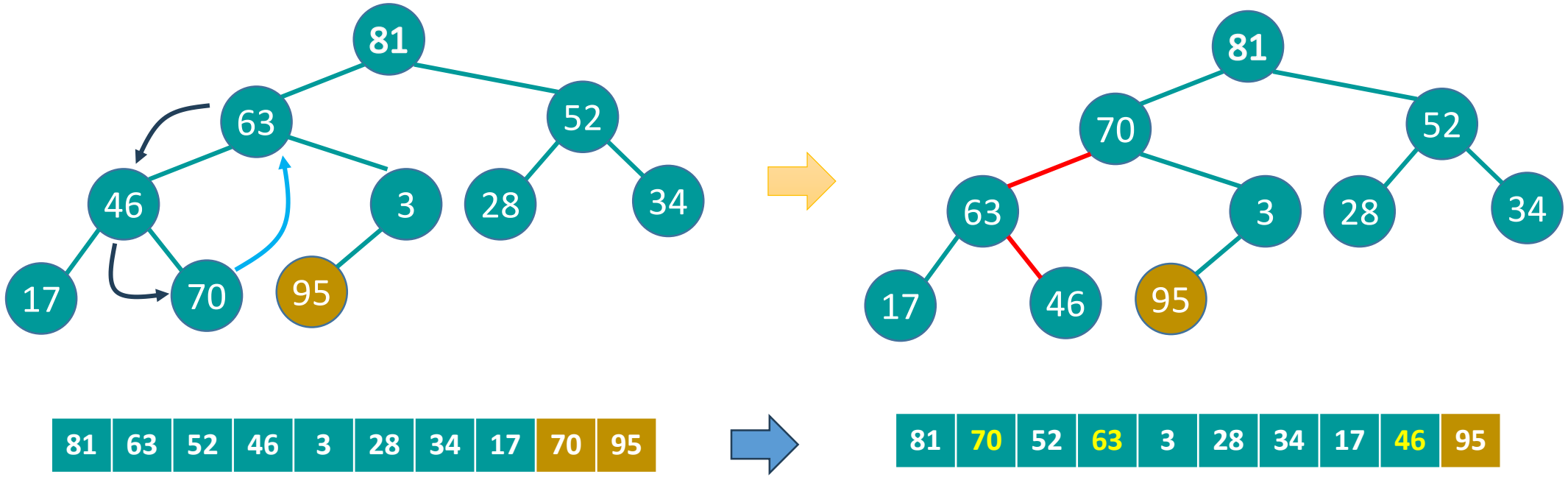
(7) SiftUp 81



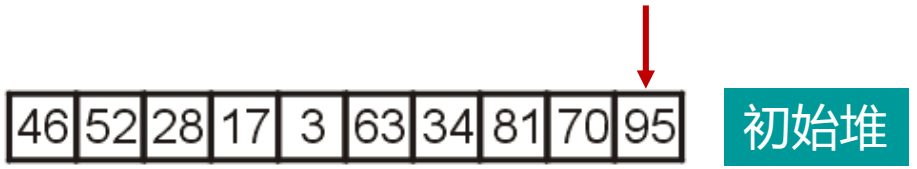
最大堆的朴素建堆操作



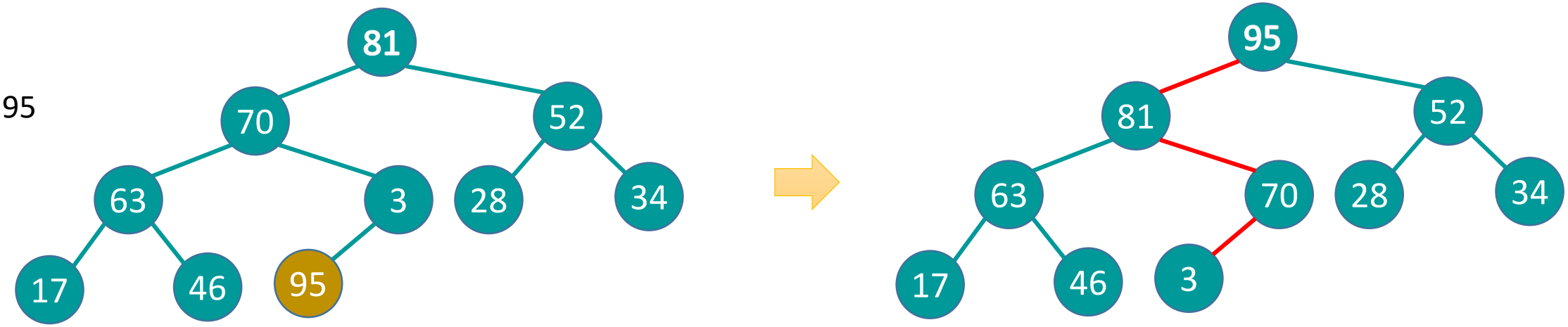
(8) SiftUp 70



最大堆的朴素建堆操作



(9) SiftUp 95



81	70	52	63	3	28	34	17	46	95

95	81	52	63	70	28	34	17	46	3

最大堆

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
```

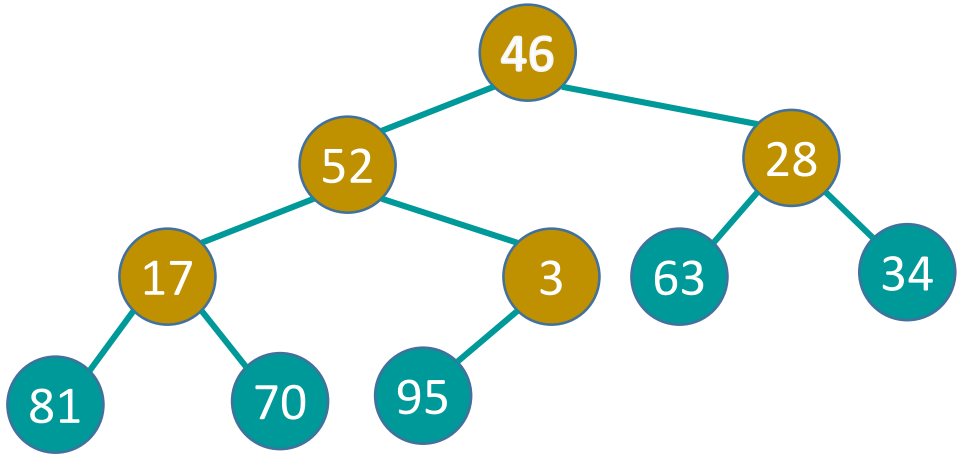
和MakeHeapUp相比，这样做的好处是在MakeHeapDown中有将近一半结点的下调操作只需要O(1)的时间，因此更加高效。具体分析如下：

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

最大堆的快速建堆操作

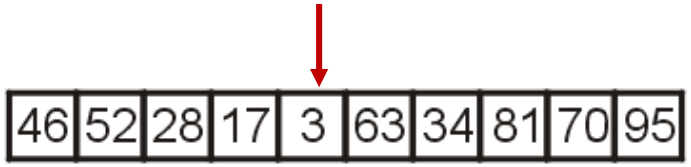
46	52	28	17	3	63	34	81	70	95

初始堆



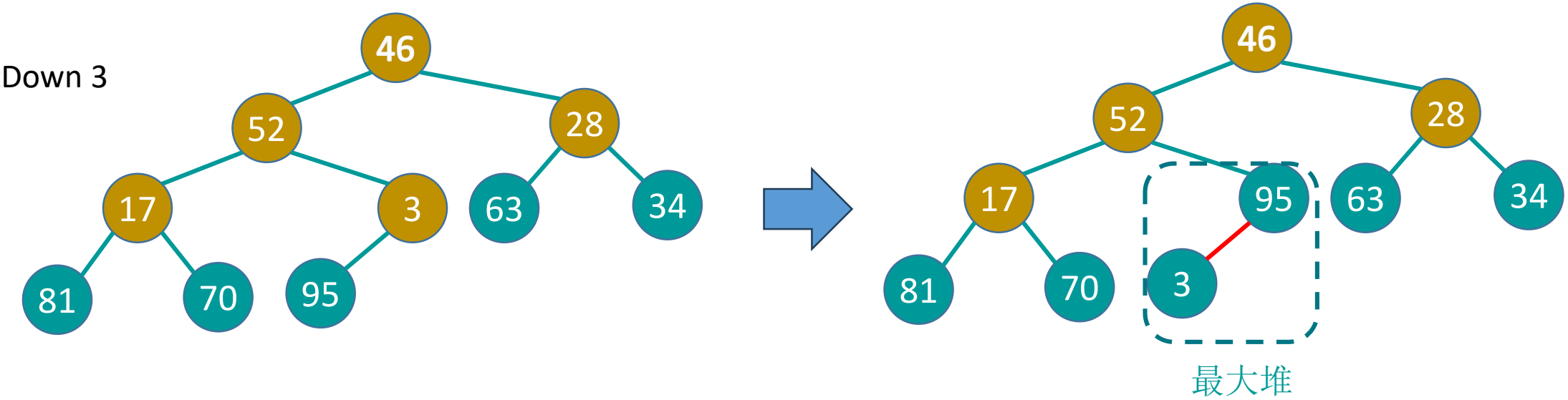
所有叶结点没有子结点，不需要下调（满足最大堆性质？）

最大堆的快速建堆操作



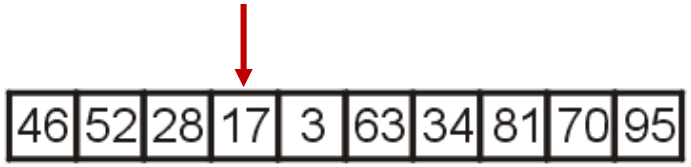
初始堆

(1) SiftDown 3



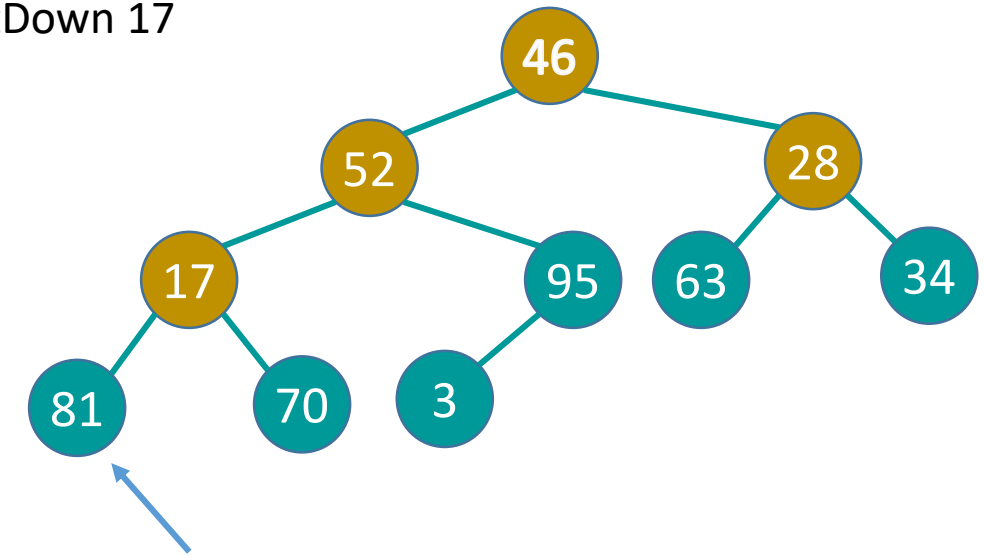
最大堆

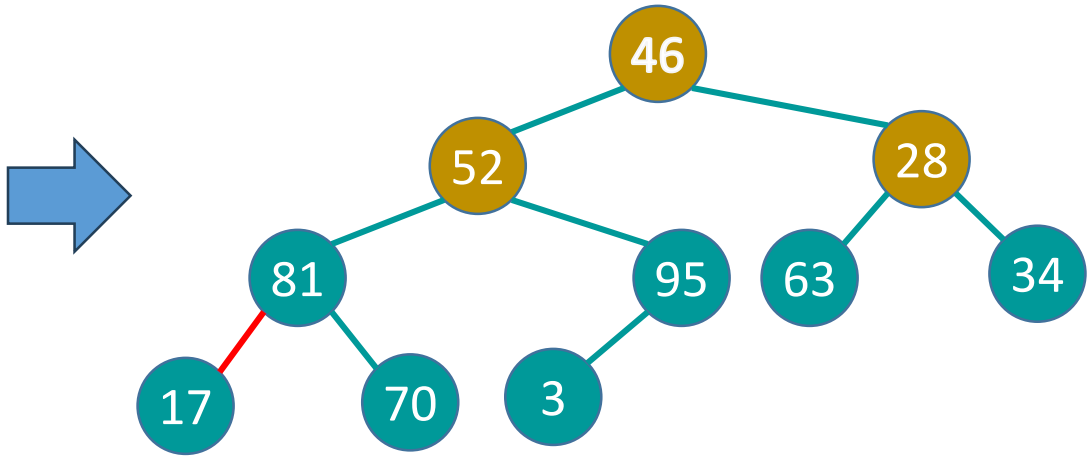
最大堆的快速建堆操作



初始堆

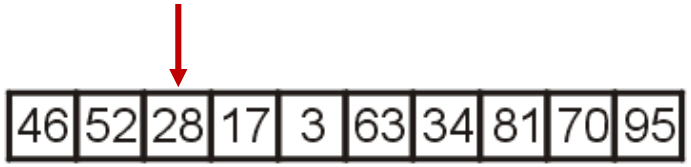
(2) SiftDown 17





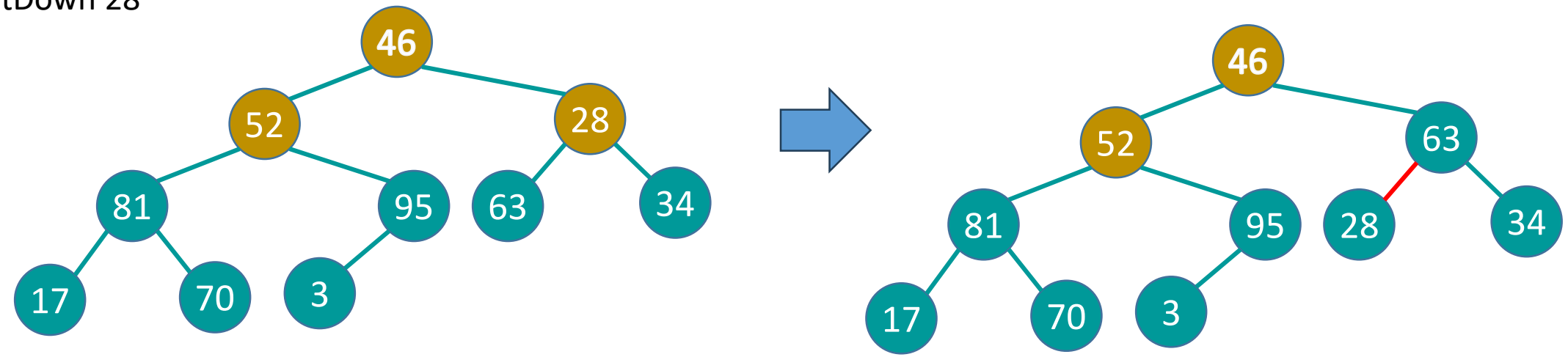
与子结点中较大值交换！

最大堆的快速建堆操作

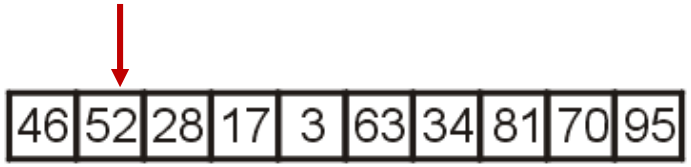


初始堆

(3) SiftDown 28

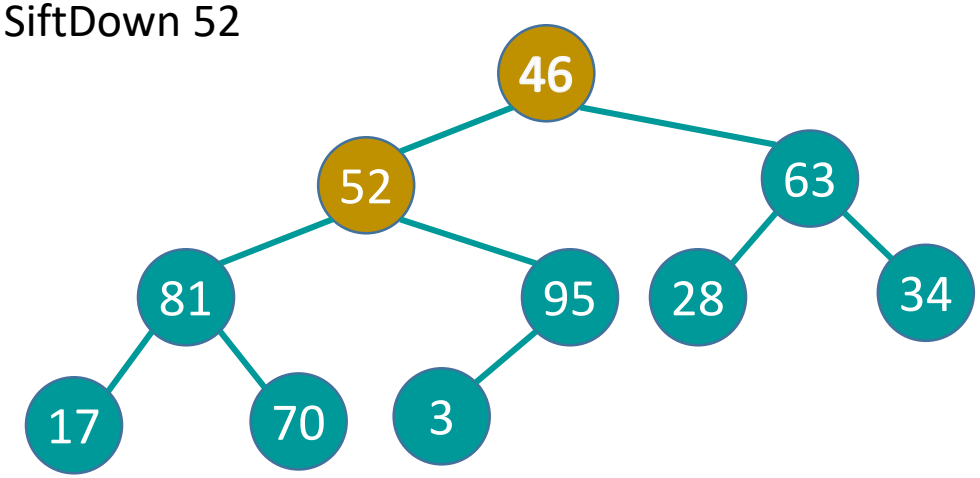


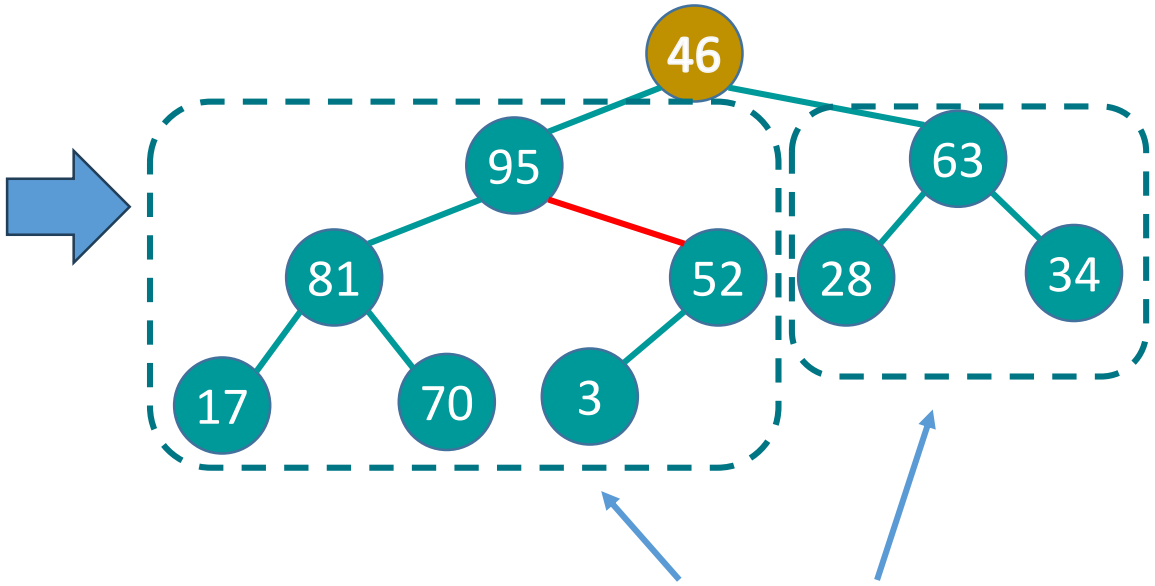
最大堆的快速建堆操作



初始堆

(4) SiftDown 52





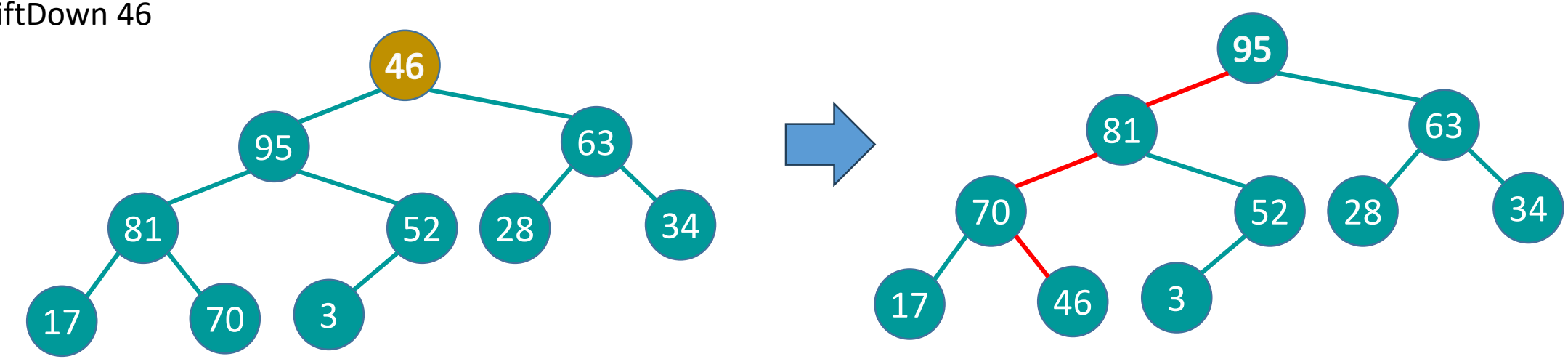
左右子树都成最大堆

最大堆的快速建堆操作

46	52	28	17	3	63	34	81	70	95

初始堆

(5) SiftDown 46



95	81	63	70	52	28	34	17	46	3

最大堆

6.5 可并堆*

一些算法需要高效地支持堆的**合并**操作，也就是把两个堆的元素合并到一个堆中，同时保持堆的性质。如果采用二叉堆结构，合并操作的实现方式是将其中一个堆（一般是元素较少的那个）中的全部元素逐一插入到另一个堆中，或者直接将两个堆的元素连接在一起再执行一次建堆操作，复杂度都比较高。

能够高效支持合并操作的堆被称为**可并堆**。常见的可并堆有**左堆**、**斜堆**、**二项堆**等。与二叉堆相比，可并堆的形状往往是不规则的，因此在实现时需要用指针来表示结点之间的连接关系。

值得一提的是，多数可并堆会将合并操作作为最基本的操作，插入操作可由原有堆与待插入元素本身构成的单元素堆的合并操作来完成，删除操作则是将原有堆删除结点后所产生的所有分离的子树进行合并。

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。

6.6 优先级队列应用：堆排序

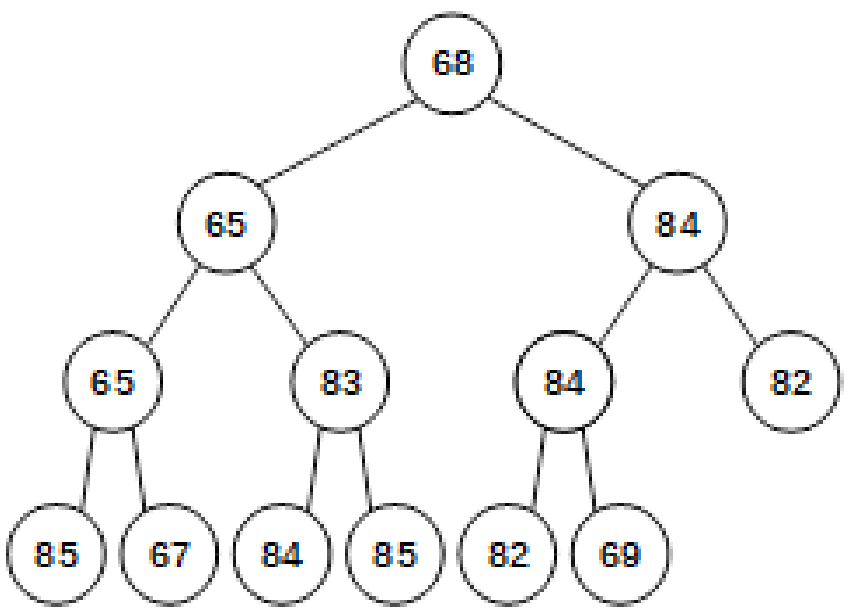
基本思想：在简单选择排序中，从长度为k的待排序序列中选出键值最小的项时，所需的时间复杂度为$O(k)$。将待排序序列组成优先级队列，则可以优化这一步骤

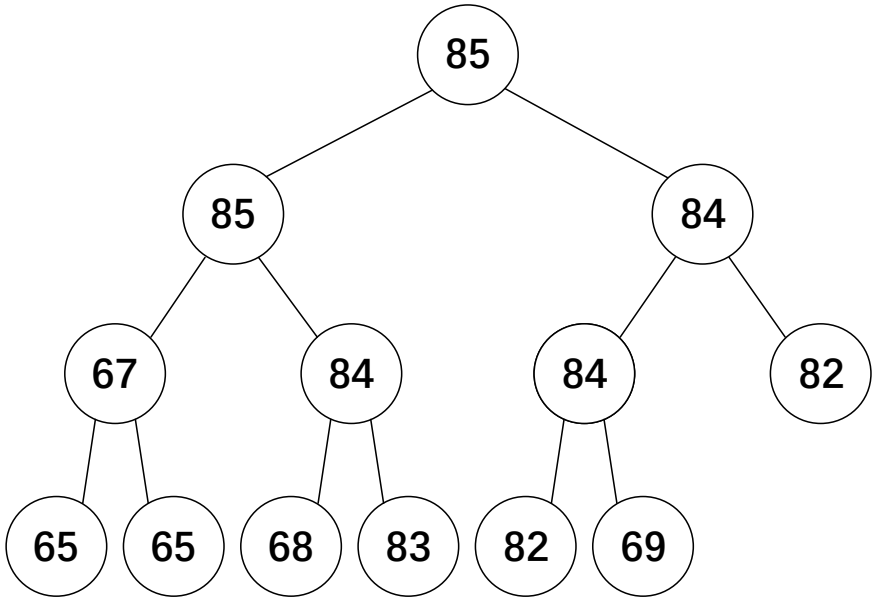
排序步骤：假设待排序序列为已经建好堆的序列$a_1, a_2, \ldots, a_n$，依次对$i = n, n-1, \ldots, 2$分别执行如下的选择步骤：在$a_1, a_2, \ldots, a_i$中选择一个键值最大的项a_1（堆顶），然后将a_i与a_1交换并修复堆。

6.6 优先级队列应用：堆排序

基本思想：在简单选择排序中，从长度为k的待排序序列中选出键值最小的项时，所需的时间复杂度为$O(k)$。将待排序序列组成优先级队列，则可以优化这一步骤

排序步骤：假设待排序序列为已经建好堆的序列$a_1, a_2, \ldots, a_n$，依次对$i = n, n-1, \ldots, 2$分别执行如下的选择步骤：在$a_1, a_2, \ldots, a_i$中选择一个键值最大的项a_1（堆顶），然后将a_i与a_1交换并修复堆。





68	65	84	65	83	84	82	85	67	84	85	82	69

待排序序列

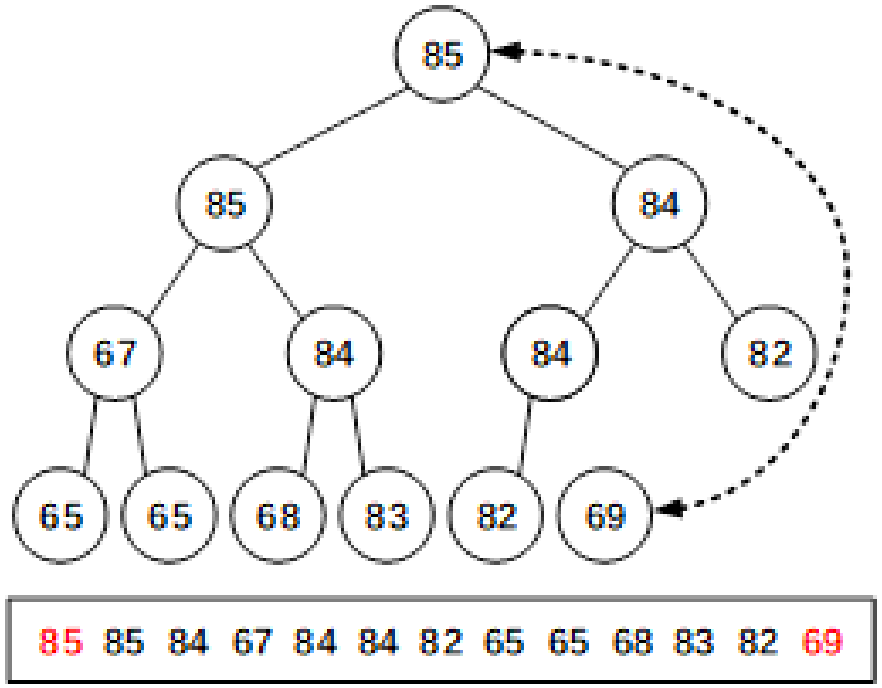
85	85	84	67	84	84	82	65	65	68	83	82	69

最大堆

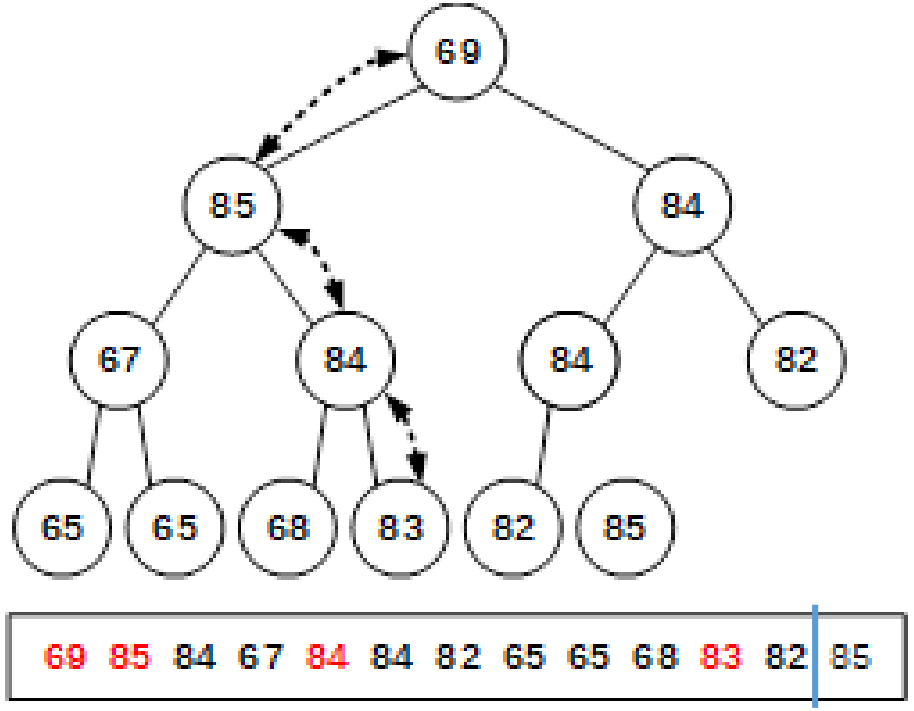
6.6 优先级队列应用：堆排序

基本思想：在简单选择排序中，从长度为k的待排序序列中选出键值最小的项时，所需的时间复杂度为$O(k)$。将待排序序列组成优先级队列，则可以优化这一步骤

排序步骤：假设待排序序列为已经建好堆的序列$a_1, a_2, \ldots, a_n$，依次对$i = n, n-1, \ldots, 2$分别执行如下的选择步骤：在$a_1, a_2, \ldots, a_i$中选择一个键值最大的项a_1（堆顶），然后将a_i与a_1交换并修复堆。



(1) 堆顶调整（交换堆顶和堆尾元素）

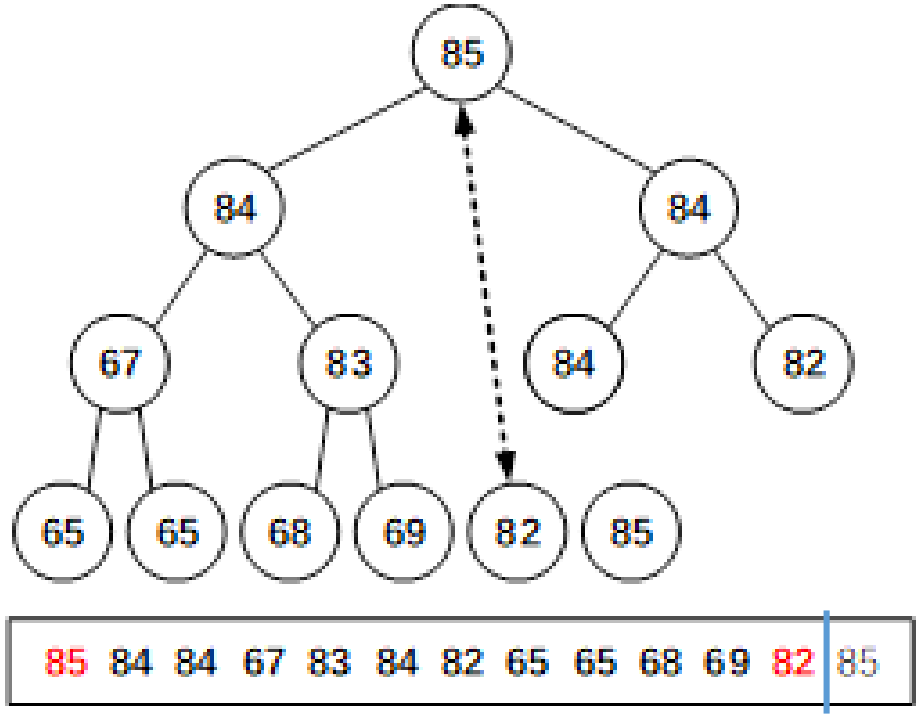


(2) 修复堆（删除堆尾元素，然后下调根结点！）

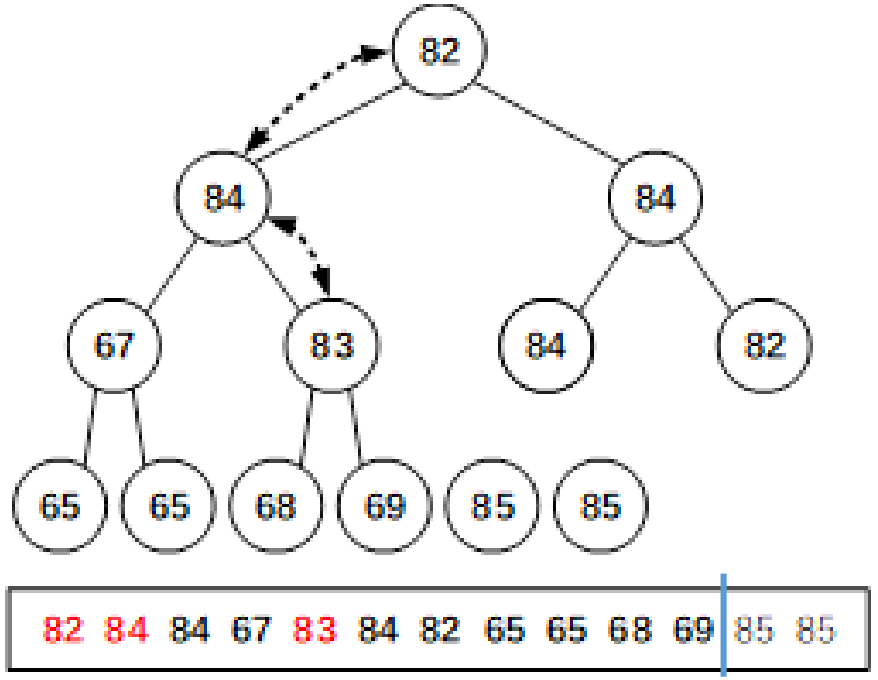
6.6 优先级队列应用：堆排序

基本思想：在简单选择排序中，从长度为k的待排序序列中选出键值最小的项时，所需的时间复杂度为$O(k)$。将待排序序列组成优先级队列，则可以优化这一步骤

排序步骤：假设待排序序列为已经建好堆的序列$a_1, a_2, \ldots, a_n$，依次对$i = n, n-1, \ldots, 2$分别执行如下的选择步骤：在$a_1, a_2, \ldots, a_i$中选择一个键值最大的项a_1（堆顶），然后将a_i与a_1交换并修复堆。



(3) 堆顶调整

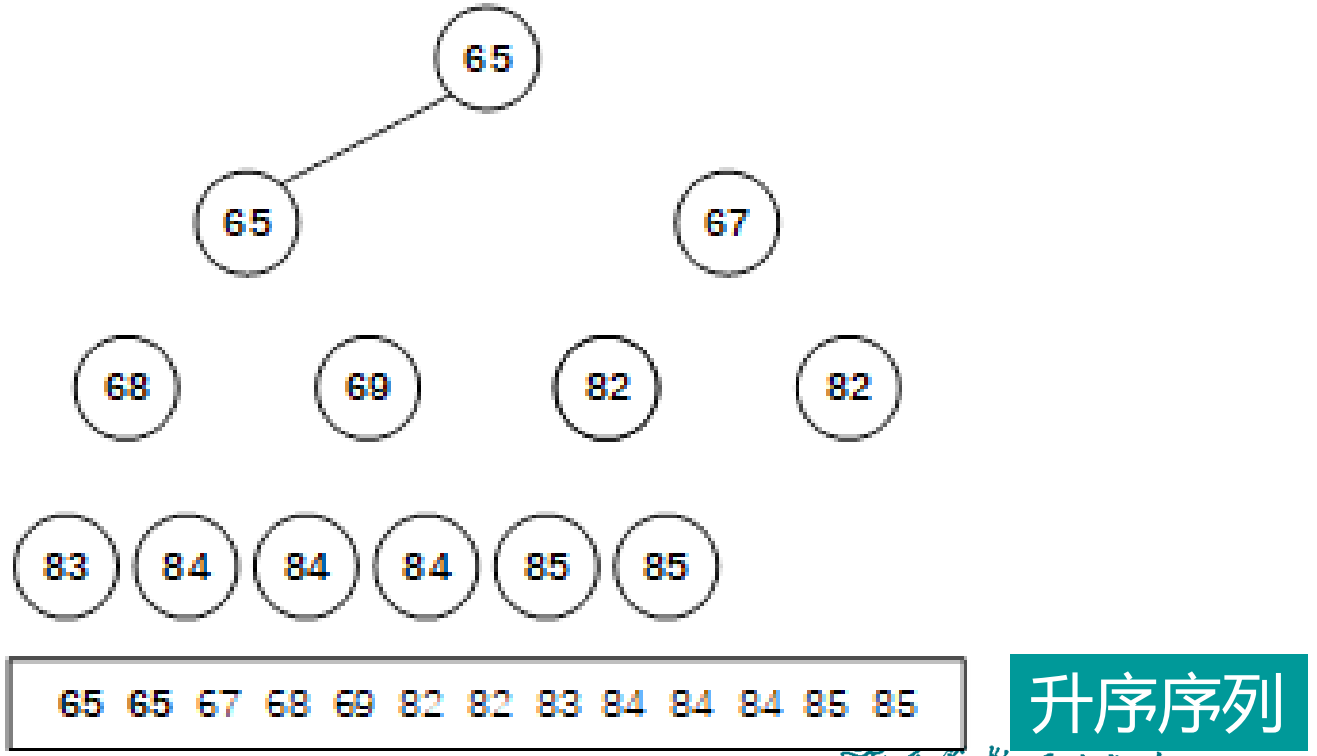


(4) 修复堆

6.6 优先级队列应用：堆排序

基本思想：在简单选择排序中，从长度为k的待排序序列中选出键值最小的项时，所需的时间复杂度为$O(k)$。将待排序序列组成优先级队列，则可以优化这一步骤

排序步骤：假设待排序序列为已经建好堆的序列$a_1, a_2, \ldots, a_n$，依次对$i = n, n-1, \ldots, 2$分别执行如下的选择步骤：在$a_1, a_2, \ldots, a_i$中选择一个键值最大的项a_1（堆顶），然后将a_i与a_1交换并修复堆。



(N) 堆顶调整

6.6 优先级队列应用：堆排序

算法： 堆排序 $\mathrm{HeapSort}(a, l, r)$

输入：序列a，左端点下标l，右端点下标r

输出：调整$a_l, a_{l+1}, \ldots, a_r$元素顺序，使元素按照非递减顺序排列

```
1  n ← r − l + 1
2  MakeHeapDown(⟨a_l, ..., a_r⟩)    //建最大堆
3  while n > 1 do  //基于堆的排序
4  |  t ← a_l
5  |  a_l ← a_{l+n−1}
6  |  a_{l+n−1} ← t
7  |  n ← n − 1
8  |  SiftDown(⟨a_l, ..., a_{l+n−1}⟩, l)
9  end
```

6.6 优先级队列应用：堆排序

时间代价：$O(n \log n)$

对n个元素进行堆排序，建立初始堆需要线性时间，即时间复杂度为$O(n)$，排序过程中需要的比较次数至多为$2n \log n$，因此总的时间复杂度为$O(n \log \mathrm{n})$。

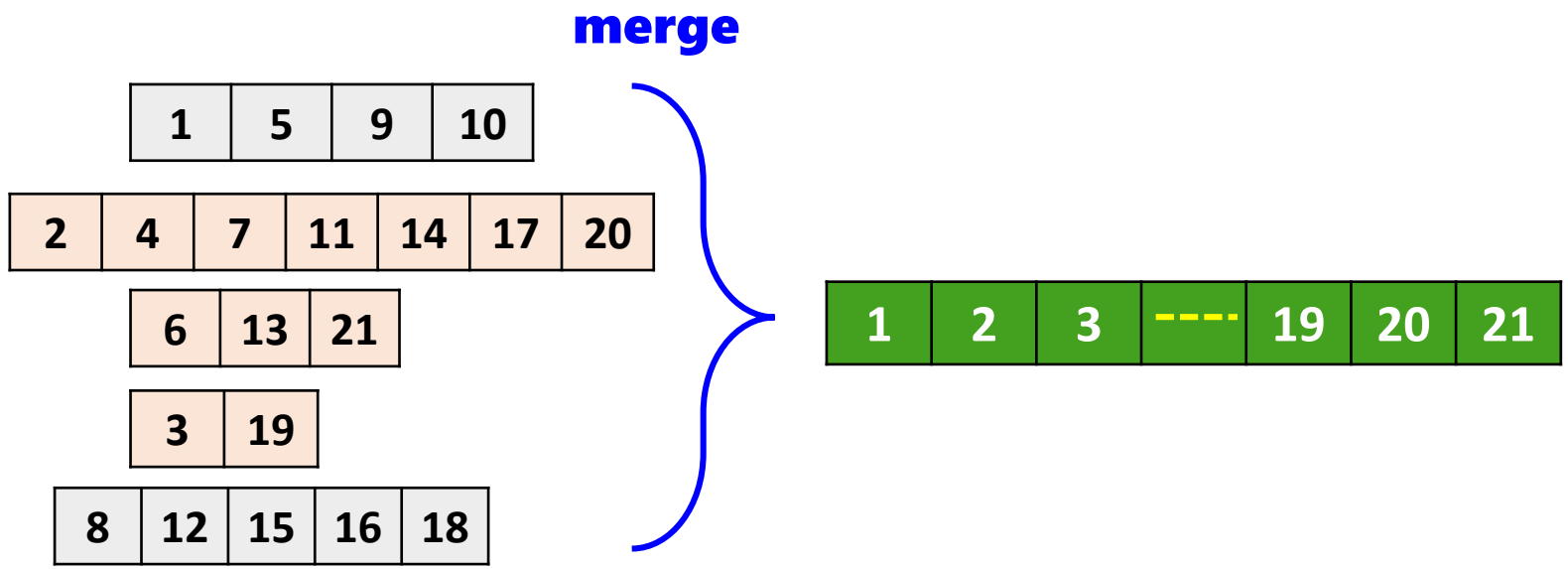
空间代价：$O(1)$

稳定性：不稳定

在堆排序中，会将**堆顶元素与数组末尾元素**进行**交换**，这一操作会**破坏**数组的稳定性，故堆排序是不稳定排序。

6.6 优先级队列应用：多路归并

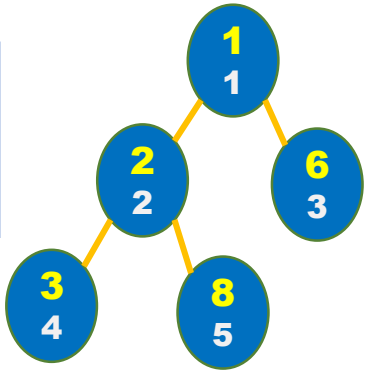
问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。简单起见，假设合并长度为x和y的两个序列需要x+y次比较，求比较次数最少的合并方案。



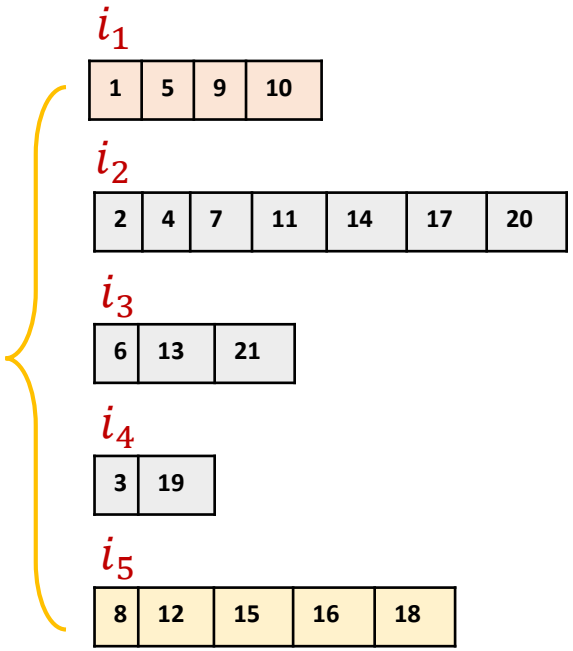
6.6 优先级队列应用：多路归并

问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。

- 建立存放所有$\langle A_x[i_x], x\rangle$的最小堆 $(1 \le x \le n)$



MIN HEAP



用指针i_x指向序列A_x 的先头元素（最小值）$(1 \le x \le n)$

算法：

① 取出堆顶元素$\langle A_y[i_y], y\rangle$并将$A_y[i_y]$放入合并后的序列（数组）

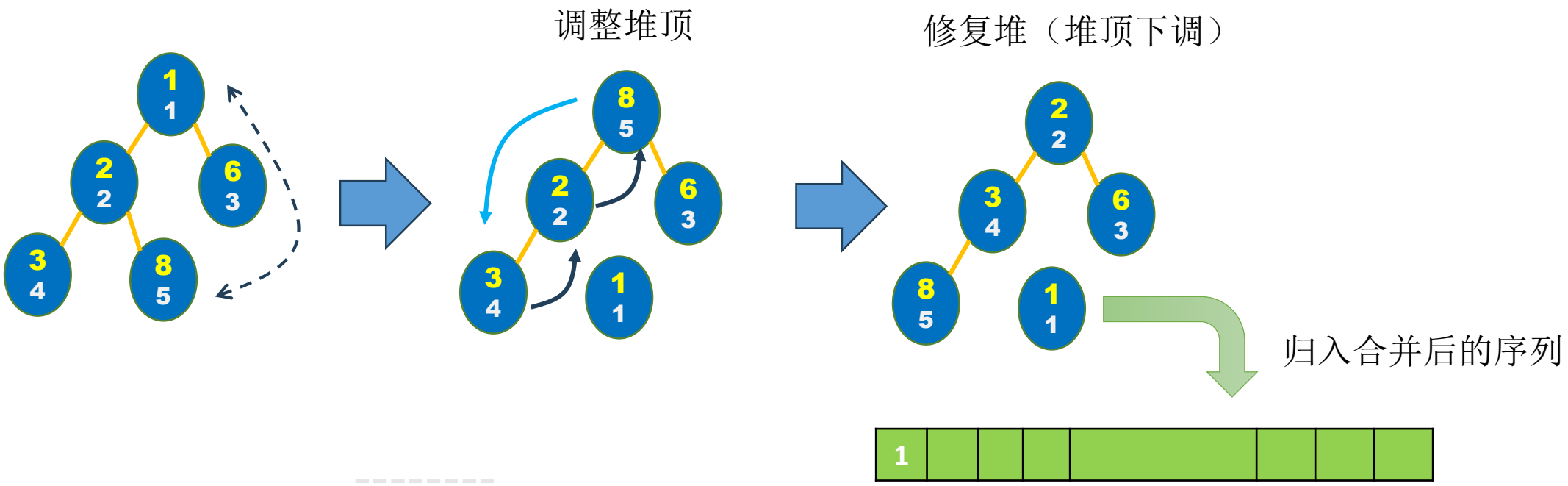
② 如果$i_y < |A_y|$，指针$i_y = i_y + 1$，插入$\langle A_y[i_y], y\rangle$至堆

③ 重复上述处理，直到合并完所有元素（堆变空！）

6.6 优先级队列应用：多路归并

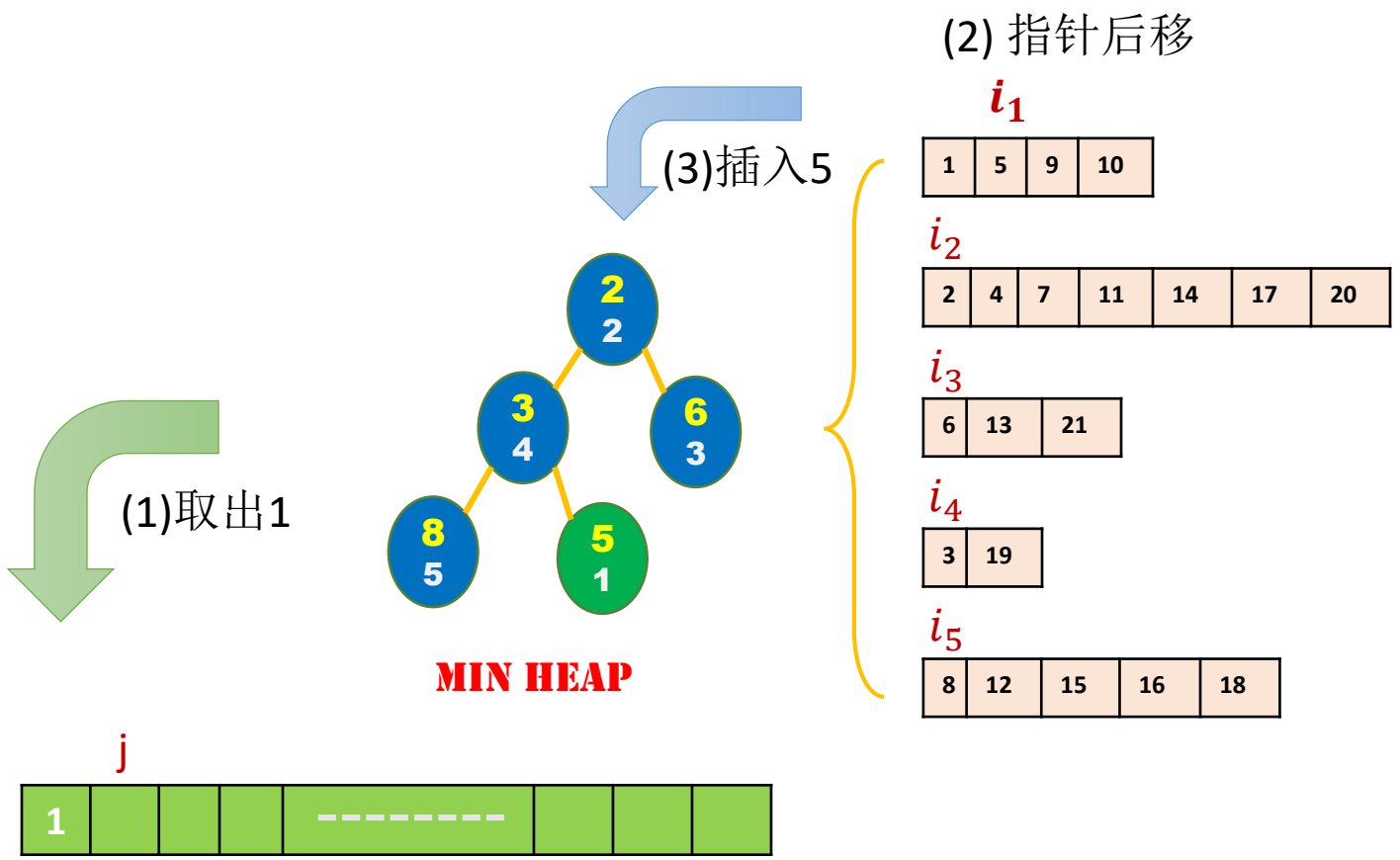
问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。

(1)取出堆顶元素



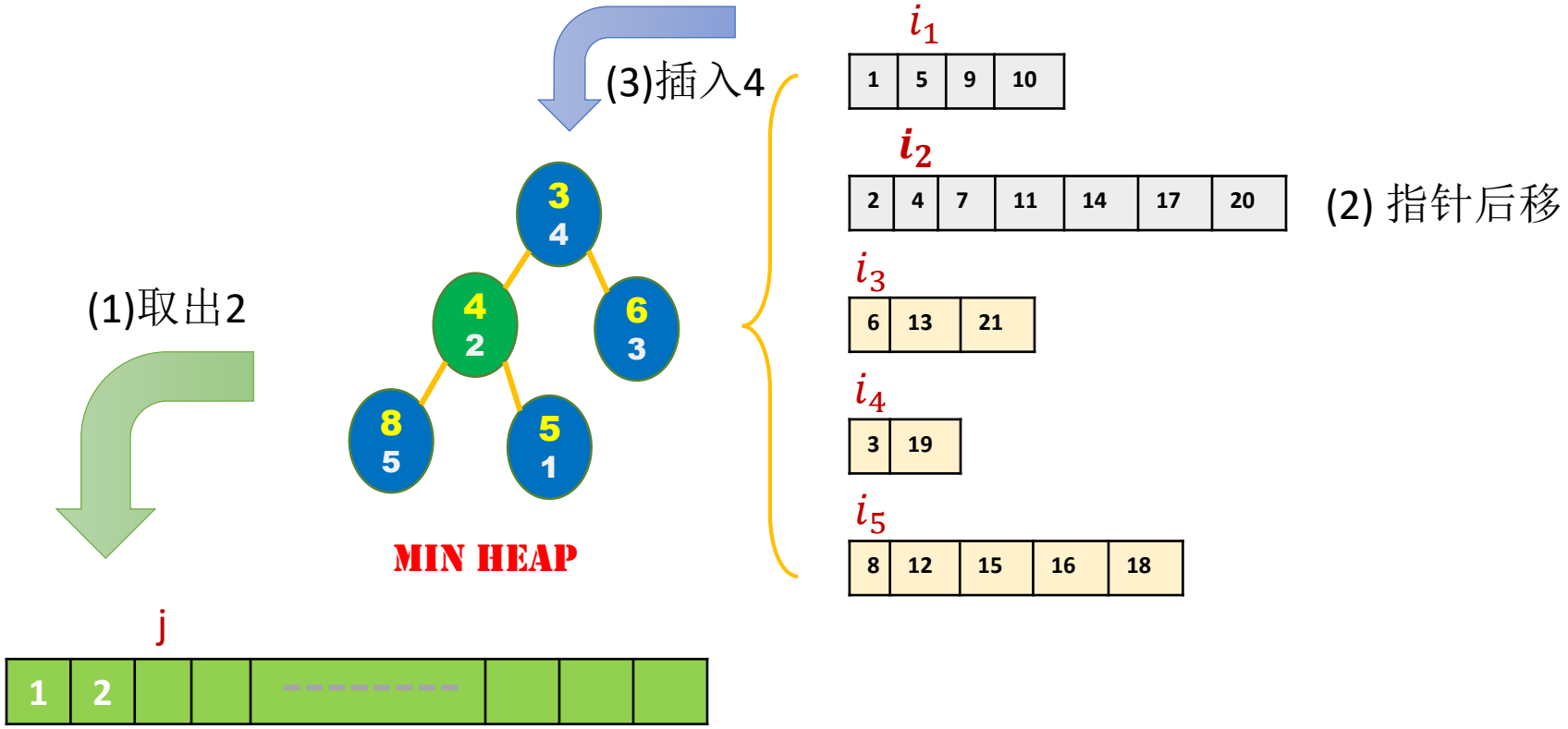
6.6 优先级队列应用：多路归并

问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。



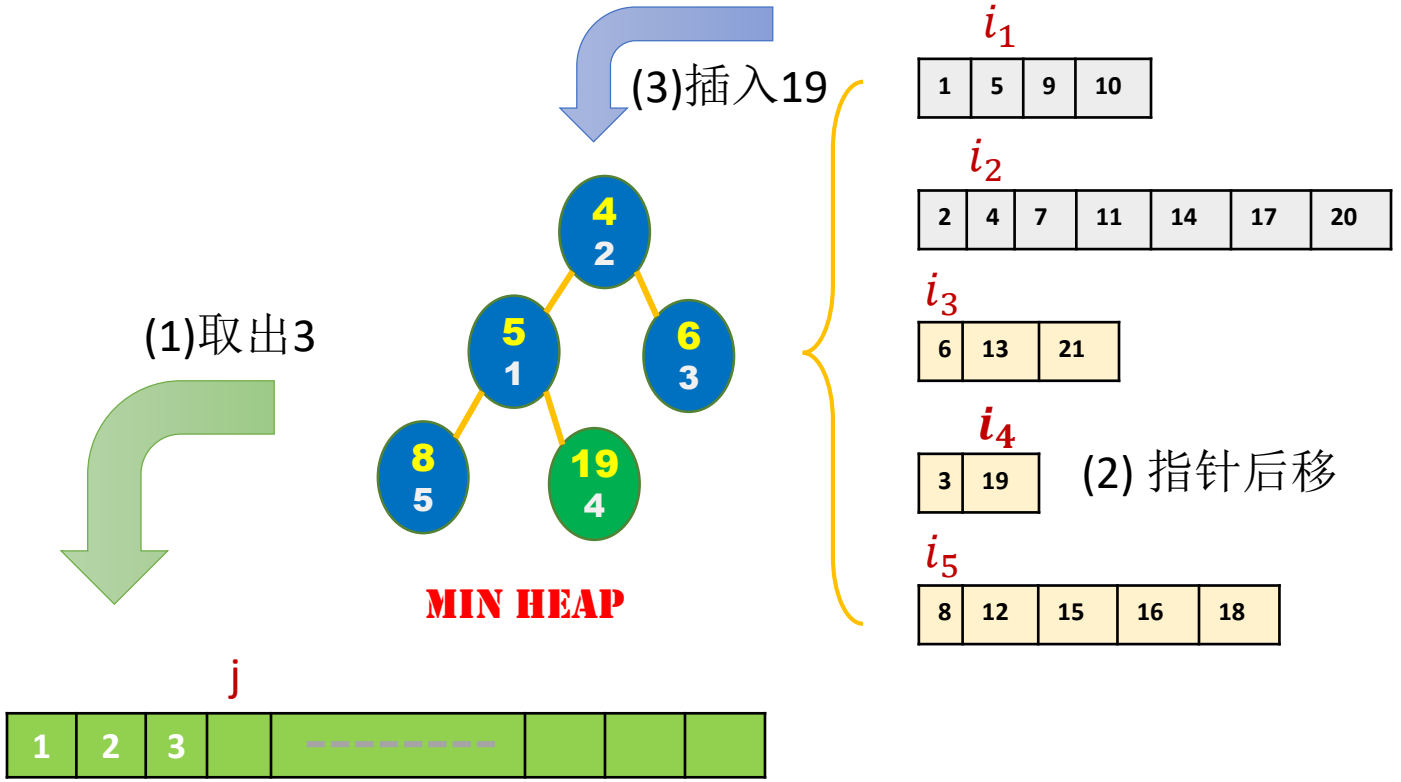
6.6 优先级队列应用：多路归并

问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。



6.6 优先级队列应用：多路归并

问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。



6.6 优先级队列应用：多路归并

问题：把n个升序序列: $A_1, A_2, \ldots, A_n$，合并成一个升序序列。

时间复杂度：

- 用指针i_x指向数组A_x $(1 \le x \le n)$中最小值位置 $(i_x = 1)$

- 设计存放所有$\langle A_x[i_x], x\rangle$的最小堆 $(1 \le x \le n)$

- 取出堆顶元素$\langle A_y[i_y], y\rangle$并放入合并后的数组；如果$i_y < |A_y|$，插入$\langle A_y[i_y + 1], y\rangle$至堆，指针$i_y = i_y + 1$。重复该处理，直到合并完所有元素

$$T\left(\sum_{1\le i\le n}|A_i|\right) = \left(\sum_{i=1}^{n}|A_i|\right) * \log(n)$$

单选题 1分

设置

2031个互异数据构成最大堆，则最小值在堆（数组）中的位置（下标）可以是

A 1

B 1014

C 1015

D 1016

提交

单选题 1分

设置

最小堆的层序遍历结果是<1，3，2，5，4，7，6>，使用朴素建堆法将其调整为最大堆，则树的层序遍历结果是

A 7，6，5，4，3，2，1

B 7，4，6，1，3，2，5

C 7，5，4，3，2，6，1

D 7，1，3，4，6，2，5

提交

单选题 1分

设置

最小堆的层序遍历结果是<1，3，2，5，4，7，6>，使用快速建堆法将其调整为最大堆，则树的中序遍历结果是

A 3，5，4，2，6，1，7

B 1，4，3，7，2，6，5

C 3，5，4，7，2，6，1

D 4，1，3，7，6，2，5

提交

多选题 1分

设置

关于最大堆和哈夫曼树，下面正确的是

A 没有度为1的结点

B 如果树中两个子树的根结点权重或数据相同，可以交换

C 任何结点的权重或数据大于等于它的所有子孙结点的值

D 堆是完全二叉树，而哈夫曼树不是完全二叉树

提交

6.7 拓展延伸

优先级队列和堆有很多可以拓展的方面，下面介绍**双端优先级队列**和**对顶堆**。

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。

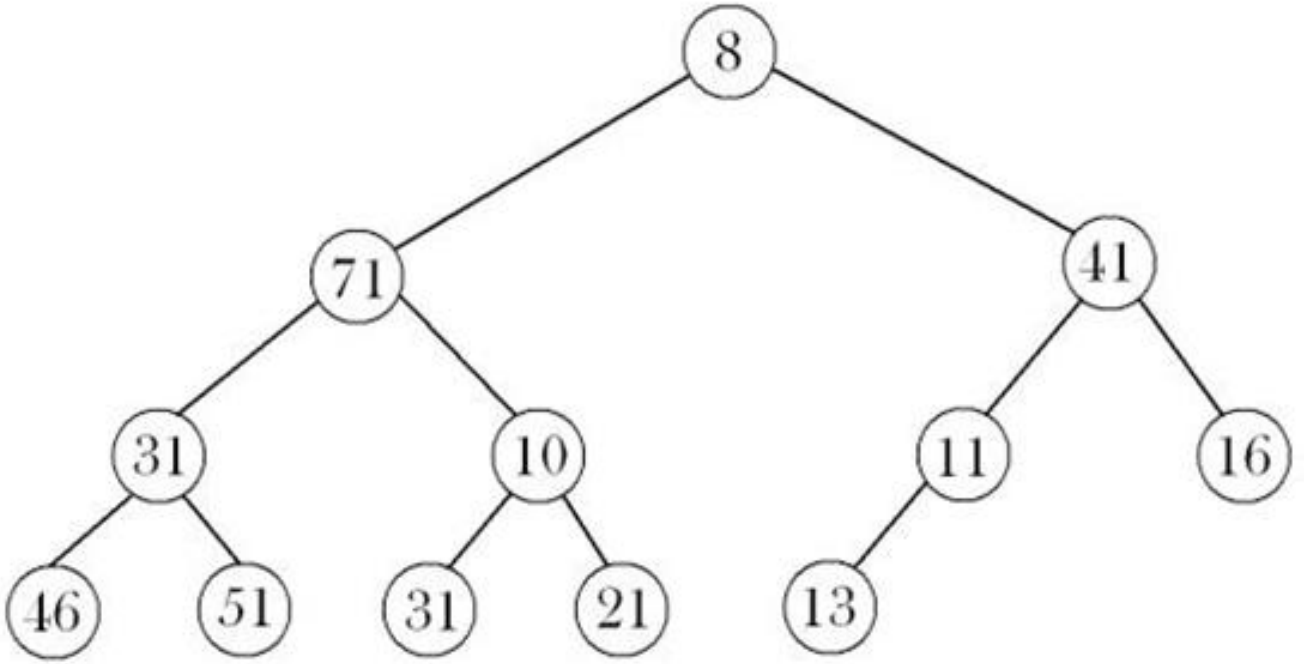
双端优先级队列**可以**使用一个最大堆和一个最小堆配合来实现。在实现时，这两个堆中的元素要分别增加指向另外一个堆中对应元素的指针域，并在元素插入或者删除时做好相应的维护。第12章要介绍的自平衡二叉搜索树（如红黑树、AA树、伸展树等）也可以用来实现双端优先级队列。

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**偶数层**（0、2、4、……）的结点小于它的所有后代结点，而位于**奇数层**（1、3、5、……）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 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \leq K \leq N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 子序列<3,7,4>, 最小元素：3,4,7，返回14
- 子序列<3,7,4,6>, 最小元素：3,4,6，返回13
- 子序列<3,7,4,6,5>, 最小元素：3,4,5，返回12
- 子序列<3,7,4,6,5,1>, 最小元素：1,3,4，返回8
- 子序列<3,7,4,6,5,1,2>, 最小元素：1,2,3，返回6

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

算法1：插入排序

- 对子序列A[1...K]按升序（非降序）排序，计算元素和值并输出
- 从 j = K+1开始，执行以下操作：
  （1）将A[j] 添加至上述升序（非降序）序列末尾，作插入排序
  （2）删除序列末尾的最大值
  （3）输出序列中K个元素的和值
- 重复上述操作，直到 j = N为止，即处理完A[1...N]中所有数据

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 对子序列<3, 7, 4>排序：<3, 4, 7>，和值14
- 添加6：<3, 4, 7, 6>；插入排序：<3, 4, 6, 7>；去掉7：<3, 4, 6>，和值13
- 添加5：<3, 4, 6, 5>；插入排序：<3, 4, 5, 6>；去掉6：<3, 4, 5>，和值12
- 添加1：<3, 4, 5, 1>；插入排序：<1, 3, 4, 5>；去掉5：<1, 3, 4>, 和值8
- 添加2：<1, 3, 4, 2>；插入排序：<1, 2, 3, 4>；去掉4：<1, 2, 3>，和值6

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值$(1 \le K \le N)$，并返回这K个最小值的和值。

算法1：插入排序

- 对子序列A[1...K]按升序（非降序）排序，计算元素和值并输出
- 从 j = K+1开始，执行以下操作：
  （1）将A[j] 添加至上述升序（非降序）序列末尾，作插入排序
  （2）删除序列末尾的最大值 (?)
  （3）输出序列中K个元素的和值（思考：如何快速求和）
- 重复上述操作，直到 j = N为止，即处理完A[1...N]中所有数据

时间复杂度：$O(K\log(K)) + O((N-K)K)$

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值$(1 \le K \le N)$，并返回这K个最小值的和值。

算法2：堆

- 算法1使用有序序列保存最小的K个元素，从末尾添加新元素时，容易造成序列的调整时间长（与序列长度成正比例），效率低!
- 因此，可以考虑用堆来维护最小的K个元素

思考：维护最小的K个元素，应该使用最小堆 or 最大堆?

投票 最多可选1项

设置

假设序列中不断有新元素加入，每添加一个元素都需要动态记录当前序列中最小的K个元素，这种情况下使用哪种堆方便处理？

A 最小堆

B 最大堆

C 最大最小堆

D 最小最大堆

提交

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

算法2：最大堆

- 对子序列A[1...K]建最大堆，计算元素和值并输出
- 从 j = K+1开始，执行以下操作：
  （1）将A[j] 添加至最大堆末尾，并上调
  （2）取出堆头元素（最大值），并作下调操作
  （3）输出堆中K个元素的和值
- 重复上述操作，直到 j = N为止，即处理完A[1...N]中所有数据

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

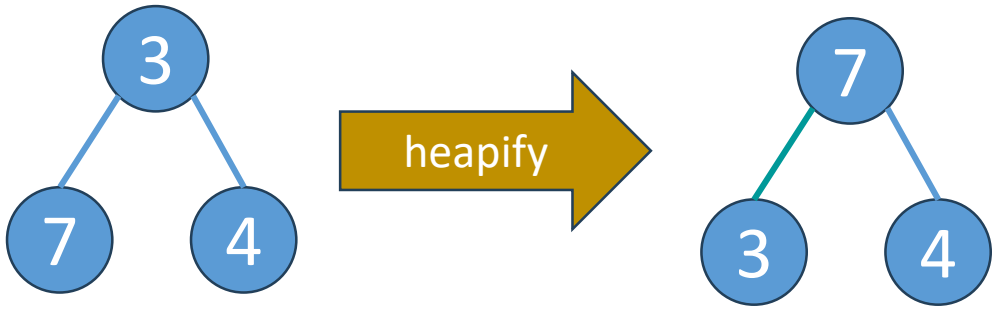
- 对子序列<3, 7, 4>建最大堆：<7, 3, 4>，和值14
- 添加6：<7, 3, 4, 6>；上调：<7, 6, 4, 3>；去堆头：<6, 3, 4>，和值13
- 添加5：<6, 3, 4, 5>； 上调：< 6 ,5, 4, 3>；去堆头：<5, 3, 4>，和值12
- 添加1：<5, 3, 4, 1>；上调： <5, 3, 4, 1>；去堆头： <4, 3, 1>，和值8
- 添加2：<4, 3, 1, 2>；上调： < 4, 3, 1, 2>；去堆头： <3, 2, 1>，和值6

6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 对子序列<3, 7, 4>建最大堆：<7, 3, 4>，和值14

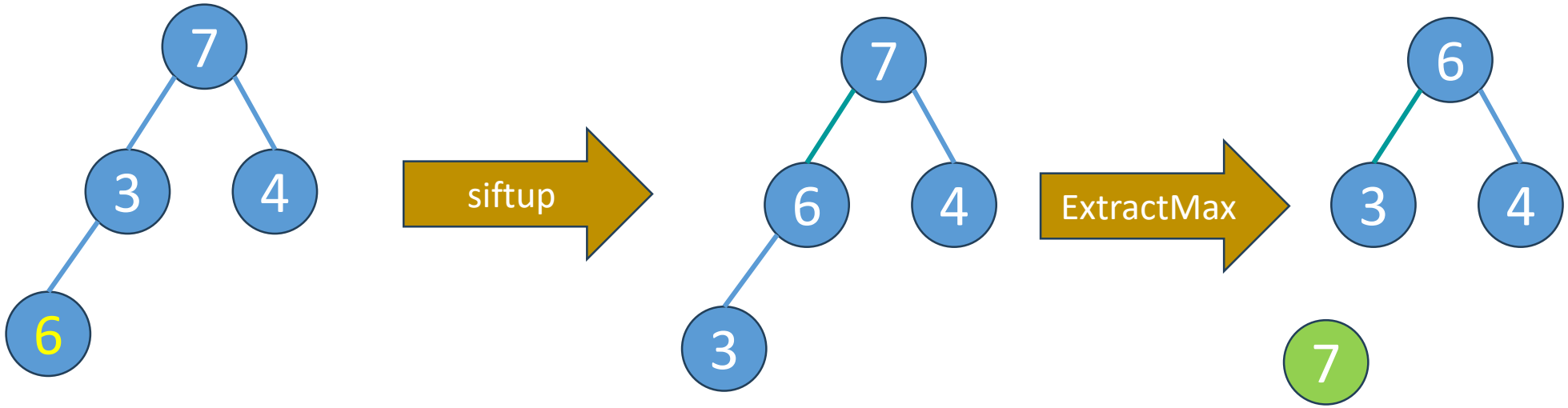


6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 添加6：<7, 3, 4, 6>；上调：<7, 6, 4, 3>；去掉堆头：<6, 3, 4>，和值13

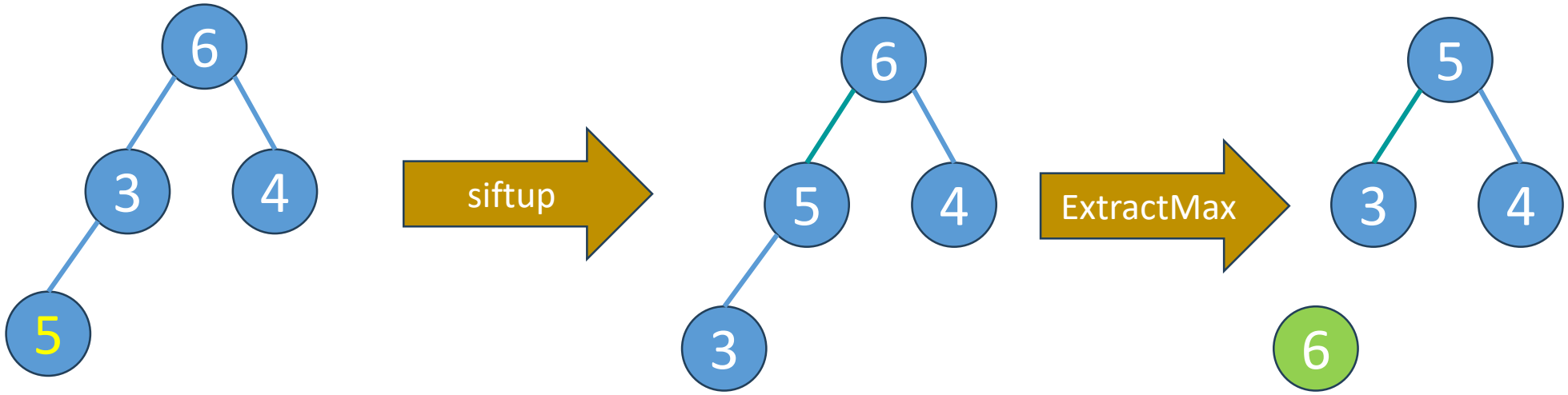


6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 添加5：<6, 3, 4, 5>；上调：<6, 5, 4, 3>；去掉堆头：<5, 3, 4>，和值12

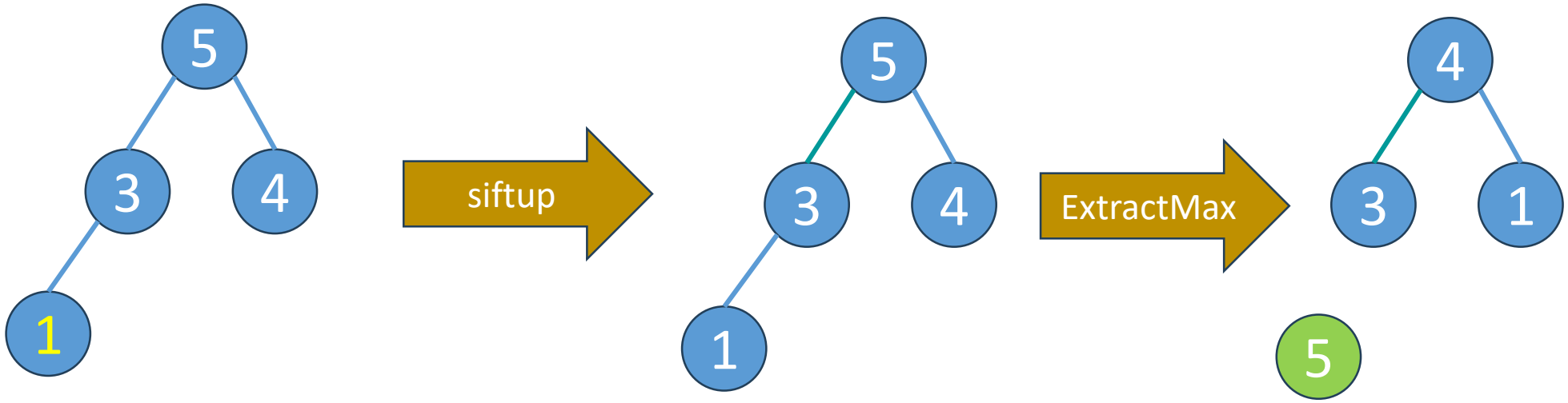


6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 添加1：<5, 3, 4, 1>；上调：<5, 3, 4, 1>；去掉堆头：<4, 3, 1>, 和值8

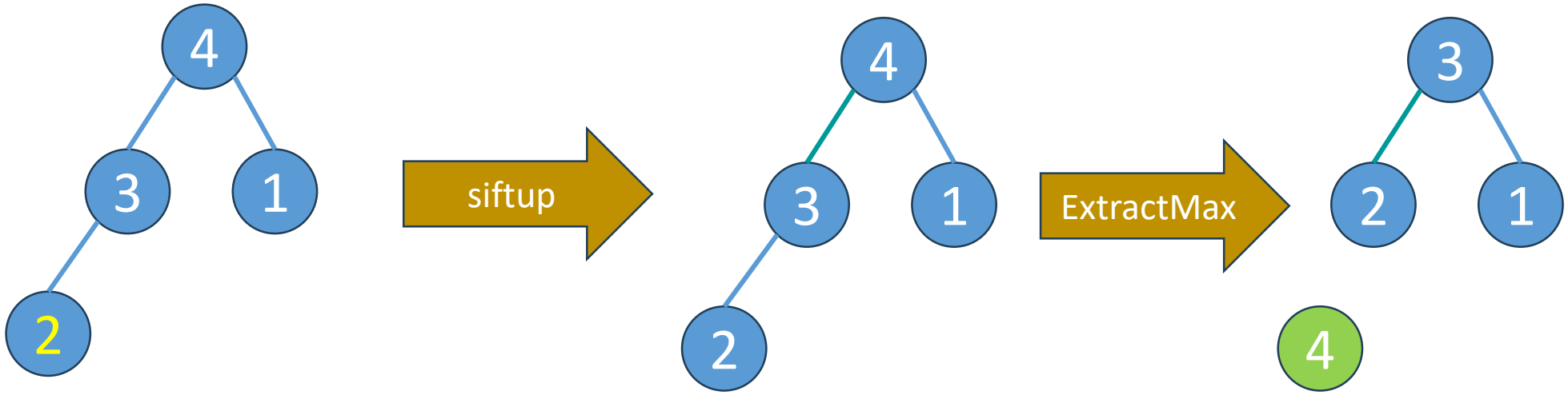


6.7.2 堆的应用：K个最小元素和

问题描述：给定长度为N的整数序列A[1...N]，依次查找子序列（前缀）A[1...K], A[1...K+1],...,A[1...N]中最小的K个数值($1 \le K \le N$)，并返回这K个最小值的和值。

示例：A=<3, 7, 4, 6, 5, 1, 2>, K=3

- 添加2：<4, 3, 1, 2>；上调：<4, 3, 1, 2>；去掉堆头：<3, 2, 1>，和值6

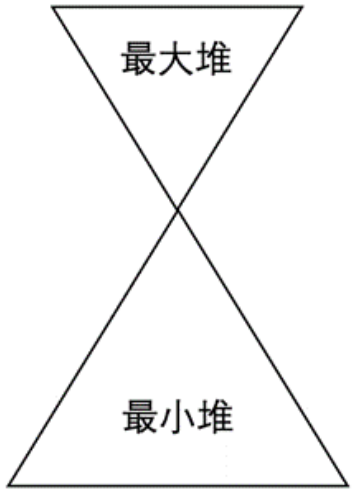


6.7.2 对顶堆*

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

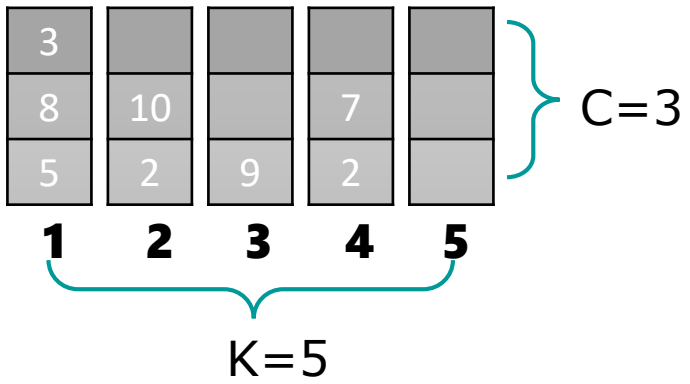
在每个模拟时钟的时间点上，模拟器都需要从事件列表中取出所有当前时刻发生的事件，并根据这些事件对应修改系统的状态，向事件列表中增加新的（会在将来发生的）事件。因此，**优先级队列**非常适合用来维护事件列表：只要以“事件发生的时间点”作为元素的优先级维护队列，并不断出队、处理事件，直到事件列表为空，或者达到了预定的模拟时间点即可停止。

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）。

6.8 离散事件模拟示例：*装箱问题（LeetCode1172）

问题描述：K个箱子从左向右排成一列，最左边箱子编号1，每个箱子最多只能装C件商品（商品用正数ID表示），且装入的商品按照先进后出的顺序取出（栈）。开始时所有箱子为空。重复N次操作，每次操作可执行的指令有三种：push, pop, popAt，其中

- void push(int id)---把商品id放入从左向右第一个未满的箱子
- int pop()---取出从右往左第一个非空箱子里最上面的商品，如果所有栈为空，返回-1
- int popAt(int k)---取出第k个箱子里最上面的商品，如果栈为空，返回-1



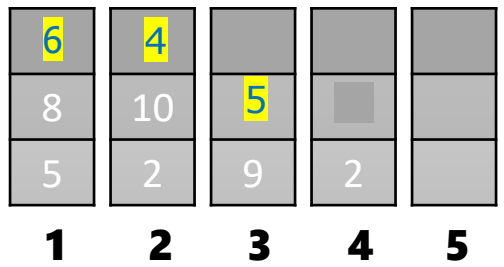
思考：如果没有PopAt操作，商品会如何在K个箱子中排列?

6.8 离散事件模拟示例：*装箱问题（LeetCode1172）

问题描述：K个箱子从左向右排成一列，最左边箱子编号1，每个箱子最多只能装C件商品（商品用正数ID表示），且装入的商品按照先进后出的顺序取出（栈）。开始时所有箱子为空。重复N次操作，每次操作可执行的指令有三种：push, pop, popAt，其中

- void push(int id)---把商品id放入从左向右第一个未满的箱子
- int pop()---取出从右往左第一个非空箱子里最上面的商品，如果所有栈为空，返回-1
- int popAt(int k)---取出第k个箱子里最上面的商品，如果栈为空，返回-1

push(4)
push(5)
pop()
popAt(1)
push(6)



6.8 离散事件模拟示例：*装箱问题（LeetCode1172）

指令（事件）	功能需求	算法及数据结构
➢ push:	需要实时跟踪各栈是否装满，并快速查找最左边的非满栈	• 用最小堆记录各栈是否装满，让最左边的非满栈在栈顶 • push操作后，根据堆顶栈的新状态调整堆
➢ pop:	需要实时跟踪各栈是否为空，并快速查找最右边的非空栈	• 用最大堆记录各栈是否为空，让最右边的非空栈在栈顶 • pop操作后，根据堆顶栈的新状态调整堆
➢ popAt:	直接更新指定栈的状态	• 直接操作指定栈，根据新状态调整最大堆和最小堆

关键：如何把满、空的状态以及左右的位置关系定义（量化）成优先级？

6.8 离散事件模拟示例：*装箱问题（LeetCode1172）

- 用最小堆记录各栈是否装满，让最左边的非满栈在栈顶

最小堆用优先级比较函数

函数： MinHeapPriority(stack_a, stack_b, index_a, index_b)
输入： 栈stack_a, stack_b, 以及各自在序列中的位置index
输出： 判断栈a是否比栈b优先

```
if IsFull(stack_a) = false and IsFull(stack_b) = false then
|   return index_a < index_b     //如果栈a和栈b都未装满，编号小（靠左）优先
end
return IsFull(stack_b)          //如果栈b装满，栈a优先；否则栈b优先
```

6.8 离散事件模拟示例：*装箱问题（LeetCode1172）

- 用最大堆记录各栈是否为空，让最右边的非空栈在栈顶

最大堆用比较函数

函数：MaxHeapPriority(stack_a, stack_b, index_a, index_b)
输入：栈stack_a, stack_b, 以及各自在序列中的位置index
输出：判断栈a是否比栈b优先

```
if IsEmpty(stack_a) = false and IsEmpty(stack_b) = false then
|   return index_a > index_b     //如果栈a和栈b都非空，编号大（靠右）优先
end
return IsEmpty(stack_b)          //如果栈b是空栈，栈a优先；否则栈b优先
```

6.9 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。

下一章开始介绍图形结构，包括图、图应用。

谢谢观看