



数据结构与算法

第 6 讲：堆

韩文弢

清华大学

2024—2025 学年度春季学期

本讲内容

6.1 问题引入	2
6.2 堆的概念	5
6.3 堆的实现	14
6.4 堆的应用	29
6.5 小结	35

6.1 问题引入

6.1.1 优先级队列

优先级队列是一种常用的抽象数据类型，删除元素时按优先级高低进行，广泛应用于各种调度场景。优先级队列需要支持的主要操作包括：

- 插入元素
- 按优先级删除元素
- 获得当前优先级最高的元素

如何高效实现优先级队列？

6.1.2 优先级队列的实现思路

考虑使用顺序表的情况。

使用有序的顺序表：
(优先级从低到高排序)

- 获得优先级最高元素 $O(1)$
- 删除元素 $O(1)$
- 插入元素 $O(n)$

使用无序的顺序表：

- 获得优先级最高元素 $O(n)$
- 删除元素 $O(n)$
- 插入元素 $O(1)$

不能同时兼顾插入和删除操作的时间复杂度。如何改进？

6.2 堆的概念

定义 6.1 (完美二叉树) 深度为 $d(d \geq 1)$ ，且有 $2^d - 1$ 个结点的二叉树，称为完美二叉树 (perfect binary tree)。

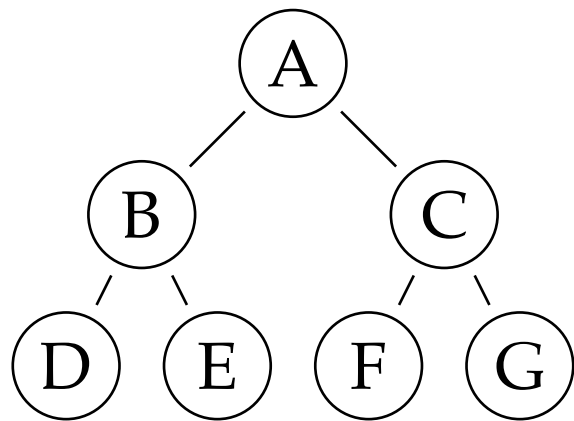


图 6.1 完美二叉树示例

定义 6.2 (完全二叉树) 除最后一层外，其他结点构成完美二叉树，且最后一层的结点从最左边开始连续排布，这种形态的二叉树称为完全二叉树 (complete binary tree)。

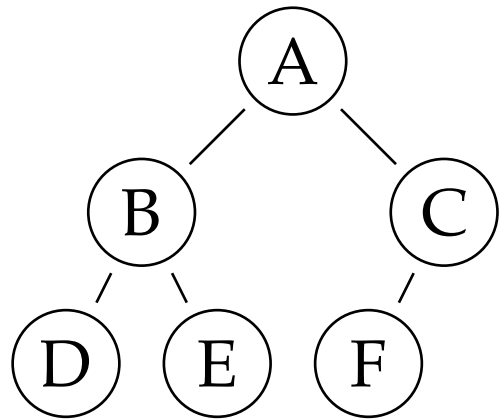


图 6.2 完全二叉树示例

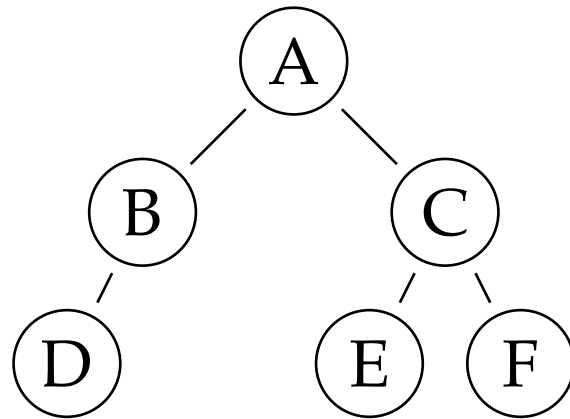
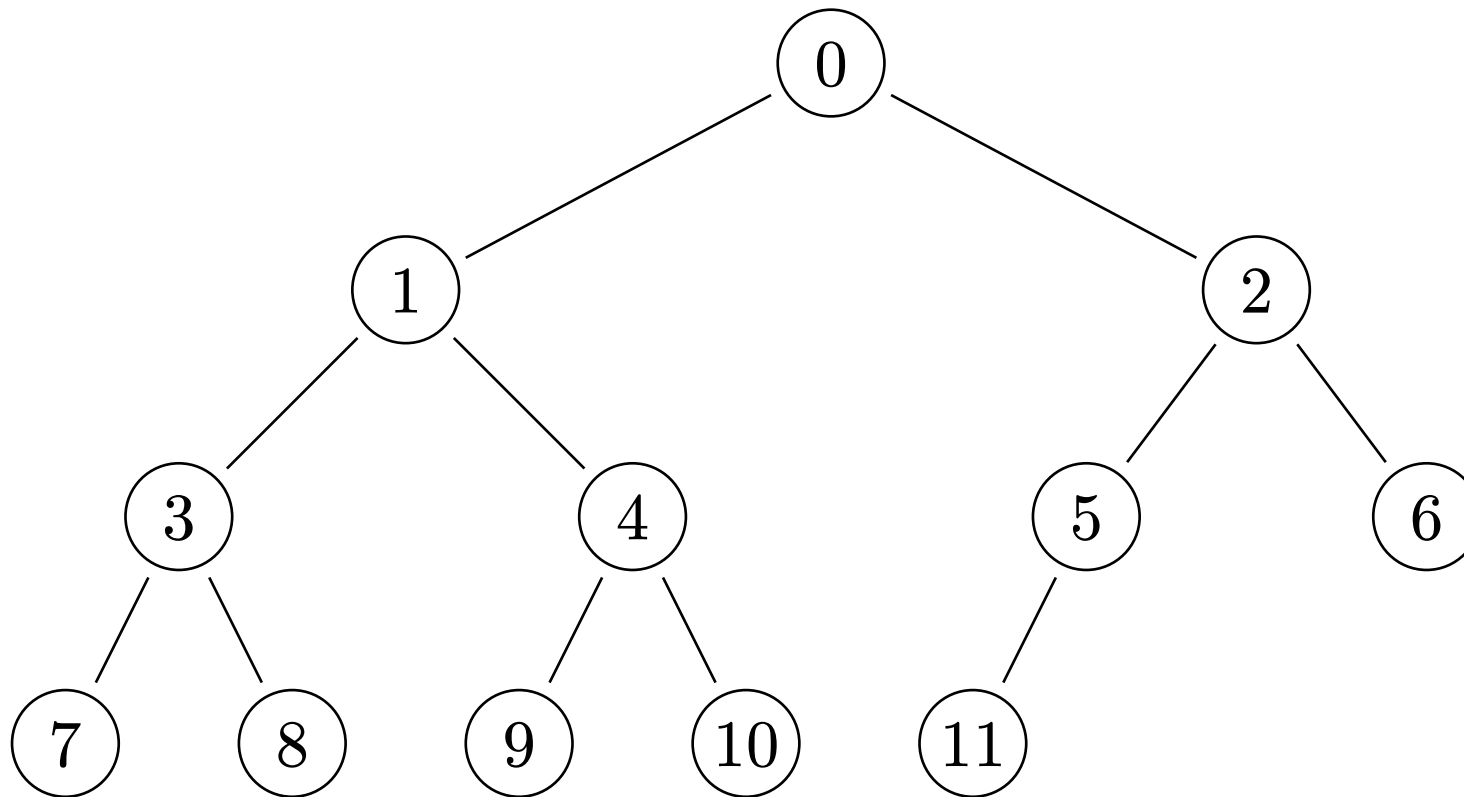


图 6.3 非完全二叉树示例

定理 6.3 设完全二叉树有 n 个结点 ($n \geq 1$), 从根结点开始从上到下、从左往右连续编号 (根结点编号为 0), 则树上任一结点 k ($0 \leq k < n$) 满足以下性质:

1. 若 $k > 0$, 则结点 k 的父结点编号为 $\lfloor \frac{k-1}{2} \rfloor$ 。
2. 若 $2k + 1 < n$, 则结点 k 的左子结点编号为 $2k + 1$, 否则结点 k 没有左子结点。
3. 若 $2k + 2 < n$, 则结点 k 的右子结点编号为 $2k + 2$, 否则结点 k 没有右子结点。

6.2.4 完全二叉树结点编号示例



思考：如何证明定理 6.3？

定理 6.4 有 n 个结点 ($n \geq 1$) 的完全二叉树的深度为 $\lceil \log(n + 1) \rceil^1$ 。

证明. 设完全二叉树的深度为 d ($d \geq 1$), 由于完全二叉树的前 $d - 1$ 层构成完美二叉树, 而第 d 层至少有一个结点, 至多有 2^{d-1} 个结点, 因此结点数 n 满足

$$2^{d-1} - 1 < n \leq 2^d - 1$$

由此可得 $d - 1 < \log(n + 1) \leq d$, 即 $d = \lceil \log(n + 1) \rceil$ 。 □

¹在本课程中, 如果没有特别标注, 对数函数默认的底数是 2。

定义 6.5 (二叉堆) 二叉堆 (binary heap, 通常简称为堆, heap) 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。对于完全二叉树 T ,

- 如果 T 的所有父子结点对都满足父结点元素不大于子结点元素, 则称 T 为**最小堆**。
- 如果 T 的所有父子结点对都满足父结点元素不小于子结点元素, 则称 T 为**最大堆**。

以上性质称为**堆性质**。

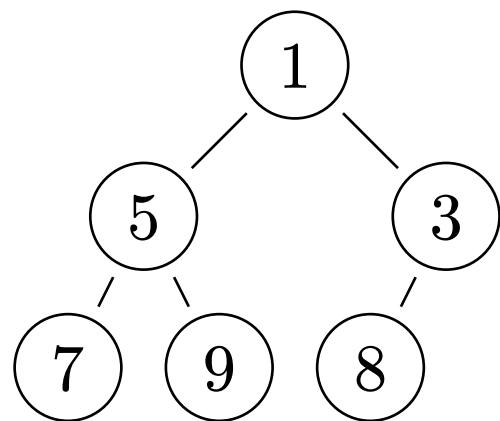


图 6.5 最小堆示例

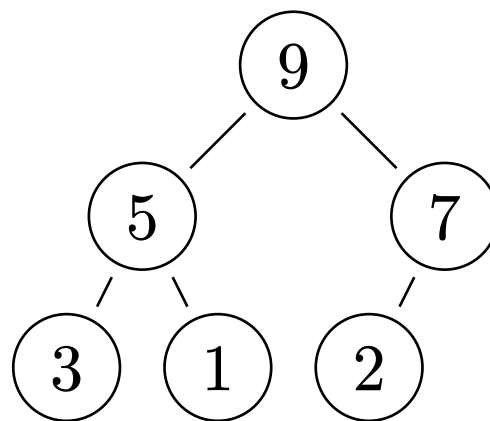


图 6.6 最大堆示例

由于最小堆和最大堆的区别只在于父子节点元素之间的大小关系，简便起见本课程默认都以**最大堆**为例进行讲解。

伪码 6.1: 堆的 ADT

元素: $D = \{a_i \mid a_i \in U, i = 0, 1, 2, \dots, n - 1\}$, U 是元素全集, n 是元素个数

关系: $R = \{(x, y) \mid x, y \in D, x < y\}$

基本操作:

Initialize(H): 初始化一个空的堆 H

Destroy(H): 销毁堆 H

Clear(H): 清空堆 H

IsEmpty(H): 返回堆 H 是否为空

Length(H): 返回堆 H 中的元素个数

Top(H): 获得堆 H 中优先级最高的元素, 若堆为空则返回 $\emptyset$

Insert(H, x): 将元素 x 插入堆 H 中

Extract(H): 从堆 H 中删除优先级最高的元素并返回, 若堆为空则返回 $\emptyset$

6.3 堆的实现

根据堆中结点编号和父子关系的性质（定理 6.3），可以不用链式存储，而用顺序表来存储堆。

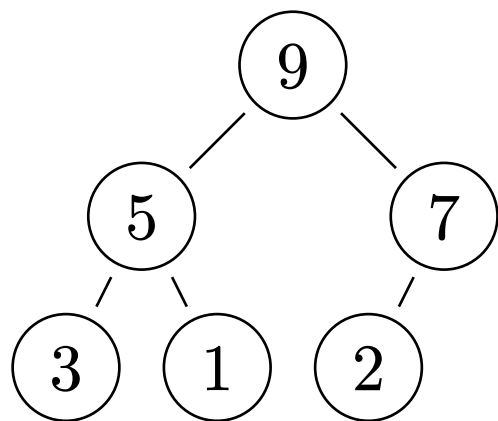


图 6.7 堆的二叉树表示

下标	0	1	2	3	4	5
元素	9	5	7	3	1	2

表 6.1 堆的顺序表表示

隐式数据结构：像堆这样使用元素下标之间的数学性质，而不是添加指针域，来表示关系的数据结构。

伪码 6.2: 堆的实现

BinaryHeap:

data: 指向数据的指针，数据连续存储

size: 当前表中实际的元素个数

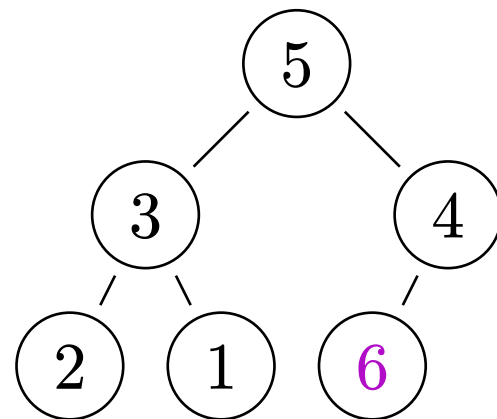
capacity: 分配的容量（最多可以存储的元素个数）

当堆中出现有结点大于它的父结点时，此时堆性质被破坏¹。

可以通过将该结点与它的父结点交换位置来恢复堆性质。

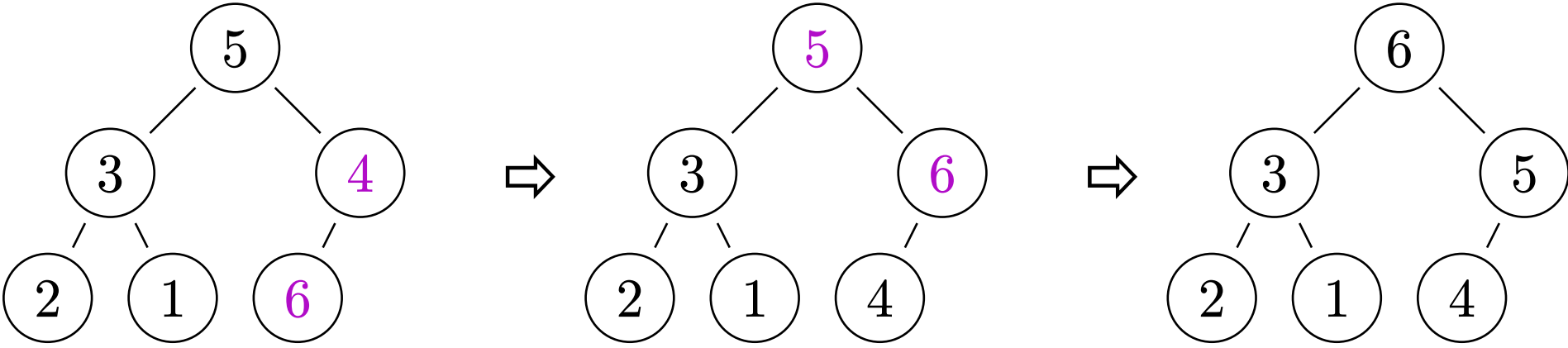
调整后继续与新的位置上的父结点比较。

以此类推，直到新的位置符合堆性质，或调整到根结点位置。



¹习惯上仍然将这棵完全二叉树称为堆。

6.3.4 堆的上调示例



伪码 6.3: 堆的上调操作

SiftUp(H, i):

```
1   $x \leftarrow H.data[i]$ 
2  while  $i > 0 \wedge x > H.data[\lfloor \frac{i-1}{2} \rfloor]$   # 当结点  $i$  大于其父结点
3       $H.data[i] \leftarrow H.data[\lfloor \frac{i-1}{2} \rfloor]$ 
4       $i \leftarrow \lfloor \frac{i-1}{2} \rfloor$ 
5   $H.data[i] \leftarrow x$ 
```

上调操作的时间复杂度是 $O(\log n)$ 。

6.3.6 堆的插入操作

有了上调操作之后，插入操作只需将元素添加到顺序表末尾，并使用上调操作将其交换至合适位置。

伪码 6.4: 堆的插入操作

Insert(H, x):

```
1  if  $H.size = H.capacity$ 
2    | error 堆空间已满
3     $H.data[H.size] \leftarrow x$ 
4    SiftUp( $H, H.size$ )
5     $H.size \leftarrow H.size + 1$ 
```

6.3.7 堆的下调操作

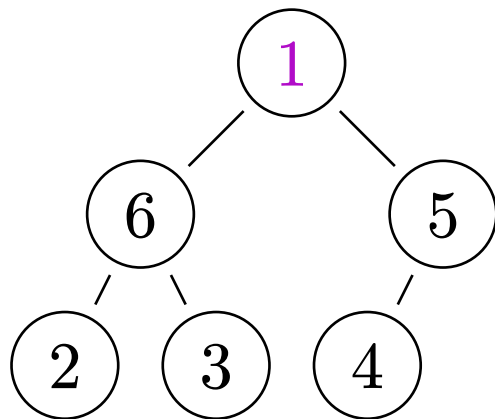
当堆中出现有元素小于它的子结点时，此时堆性质也被破坏。

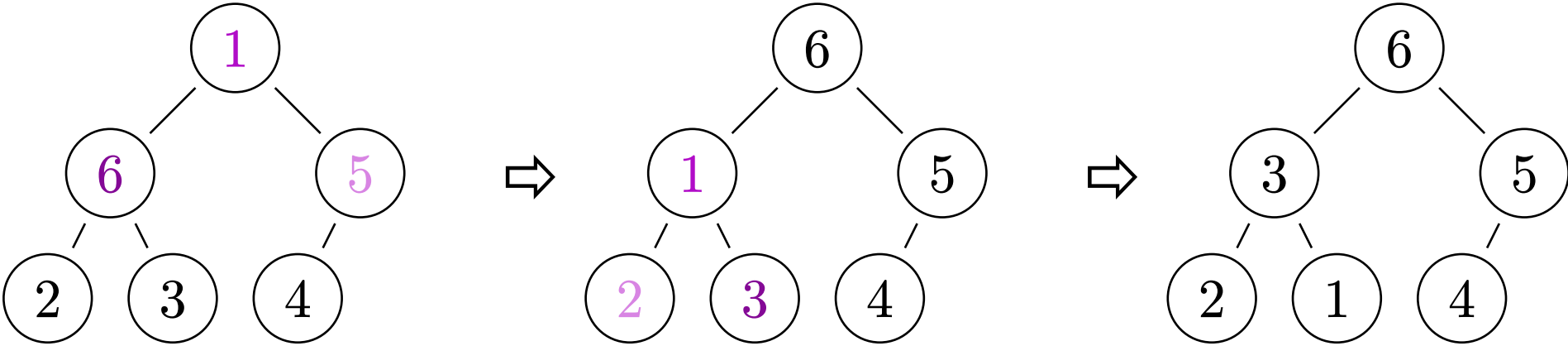
可以通过将该结点与它的子结点交换位置来恢复堆性质。

思考：如果两个子结点均大于该结点怎么办？

调整后继续与新的位置上的子结点比较。

以此类推，直到新的位置符合堆性质，或调整到叶结点位置。





伪码 6.5: 堆的下调操作

```
SiftDown( $H, i$ ):  
1   $n \leftarrow H.size$   
2   $x \leftarrow H.data[i]$   
3  while true  
4       $j \leftarrow 2i + 1$   
5      if  $j \geq n$  # 如果是叶结点则结束  
6          | break  
7      if  $j + 1 < n \wedge H.data[j + 1] > H.data[j]$  # 如果右子结点存在, 比较左右子结点谁更大  
8          |  $j \leftarrow j + 1$   
9      if  $H.data[j] > x$  # 如果大的子结点比该结点的值大, 则交换位置并继续, 否则下调结束  
10         |  $H.data[i] \leftarrow H.data[j]$   
11         |  $i \leftarrow j$   
12     else  
13         | break  
14   $H.data[i] \leftarrow x$ 
```

6.3.10 堆的删除操作

下调操作的时间复杂度也是 $O(\log n)$ 。

删除操作会导致顺序表中首元素（根结点）空缺，如何解决？

可将末元素交换至首元素位置，然后进行下调操作。

这样，堆的插入操作和删除操作的时间复杂度都是 $O(\log n)$ 。

伪码 6.6: 堆的删除操作

Extract(H):

```
1  if  $H.size = 0$ 
2    | return  $\emptyset$ 
3   $x \leftarrow H.data[0]$ 
4   $H.size \leftarrow H.size - 1$ 
5  if  $H.size > 0$ 
6    |  $H.data[0] \leftarrow H.data[H.size]$ 
7    | SiftDown( $H, 0$ )
8  return  $x$ 
```

6.3.12 建堆操作

把一个任意序列调整成堆的操作称为**建堆**。朴素的建堆思路是将元素依次插入到堆中。

伪码 6.7: 朴素建堆算法

HeapifyNaive(H):

```
1  | for  $i \leftarrow 1$  to  $H.size - 1$   
2  |   SiftUp( $H, i$ )
```

朴素建堆算法的时间复杂度为 $O(n \log n)$ 。

6.3.13 快速建堆算法

朴素建堆算法比较慢的原因是：对占结点数量一半左右的叶结点，每个结点要花费 $O(\log n)$ 的时间来调整，而只有一个的根结点只需要 $O(1)$ 的时间。思考能否改进？

伪码 6.8: 快速建堆算法

Heapify(H):

```
1  | for  $i \leftarrow \lfloor \frac{H.size-2}{2} \rfloor$  downto 0
2  |   | SiftDown( $H, i$ )
```

与朴素建堆算法相比，在快速建堆算法中，近一半的结点只需要 $O(1)$ 的时间，而根结点是 $O(\log n)$ 。

快速建堆算法的时间复杂度为：

$$\begin{aligned}\sum_{k=1}^{\lceil \log(n+1) \rceil} \frac{n}{2^{k-1}} O(k) &= n \cdot O\left(\sum_{k=1}^{\lceil \log(n+1) \rceil} \frac{k}{2^{k-1}}\right) \\ &= n \cdot O\left(\sum_{k=1}^{\infty} \frac{k}{2^{k-1}}\right) \\ &= O(n)\end{aligned}$$

6.4 堆的应用

问题. 给出一个序列 A , 要求将 A 中的元素按升序排列。

例: 给出 $A = (3, 1, 4, 1, 5)$, 则输出为 $(1, 1, 3, 4, 5)$ 。

朴素想法: 每次从还没有排好序的元素中挑出最大的元素, 将其放到末尾。如此处理所有元素, 则排序完成。例如:

1. 第一趟从 A 中选出 5, 将其放到末尾, 得到 $(3, 1, 4, 1, 5)$
2. 第二趟从前 4 个元素中选出 4, 将其放到末尾, 得到 $(3, 1, 1, 4, 5)$
3. 第三趟从前 3 个元素中选出 3, 将其放到末尾, 得到 $(1, 1, 3, 4, 5)$
4. 第四趟从前 2 个元素中选出 1, 将其放到末尾, 得到 $(1, 1, 3, 4, 5)$

这样的排序算法称为**选择排序**。

伪码 6.9: 朴素选择排序

SelectionSort(A):

```
1   $n \leftarrow A.size$ 
2  for  $i \leftarrow n - 1$  downto 1
3       $k \leftarrow i$  # 当前找到最大元素的下标
4      for  $j \leftarrow 0$  to  $i - 1$ 
5          if  $A[j] > A[k]$ 
6               $k \leftarrow j$ 
7       $A[i] \leftrightarrow A[k]$ 
```

在朴素选择排序中，每一趟选最大元素时，没有充分利用之前选择过程产生的信息。

如果将序列分块，每块各自选择最大元素，那么全局最大元素只要在上一趟每块最大元素中选择即可。对于产生全局最大元素的块，要重新选择。

设序列长度为 n ，分块长度为 k ，假设能整除，块数 $m = \frac{n}{k}$ 。分析这样做的时间复杂度：

- 初始时：每块选择 $O(k)$ ，总共是 $m \cdot O(k) = O(n)$ 。

6.4.3 优化思路

- 每趟：在所有块中选出全局最大元素需要 $O(m)$ ，重新选择该块的最大元素需要 $O(k)$

总的时间复杂度为 $O(n) + (n - 1)(O(m) + O(k)) = O(n(m + k))$ ，当 $k = \sqrt{n}$ 时最小，为 $O(n\sqrt{n})$ 。

分块的思路可以分层进行，例如分两层，第一层每块长度是 $\sqrt[3]{n}$ ，第二层每块长度是 $n^{\frac{2}{3}}$ ，则总的排序可以做到 $O(n\sqrt[3]{n})$ 。

这样不断分层分块的极限是什么？堆！

伪码 6.10: 堆排序

Heapsort(A):

```
1   $n \leftarrow A.size$ 
2  Heapify( $A$ )
3  for  $i \leftarrow n - 1$  downto 1
4       $A[0] \leftrightarrow A[i]$ 
5       $A.size \leftarrow A.size - 1$  # 缩小堆的长度
6      SiftDown( $A, 0$ )
7   $A.size \leftarrow n$  # 恢复序列长度
```

6.5 小结

6.5.1 本讲小结

- 堆的概念：完全二叉树、最小堆、最大堆
- 堆的实现：隐式数据结构、上调操作、下调操作、建堆
- 堆的应用：堆排序（选择排序的优化）