FSE598 前沿计算技术

模块 3：　算法设计与分析
单元 2　　排序算法
第 2 讲　　堆和堆排序

本课程的部分内容是基于 Thomas H.Cormen、Charles E.Leiserson 等人的“算法简介”教材

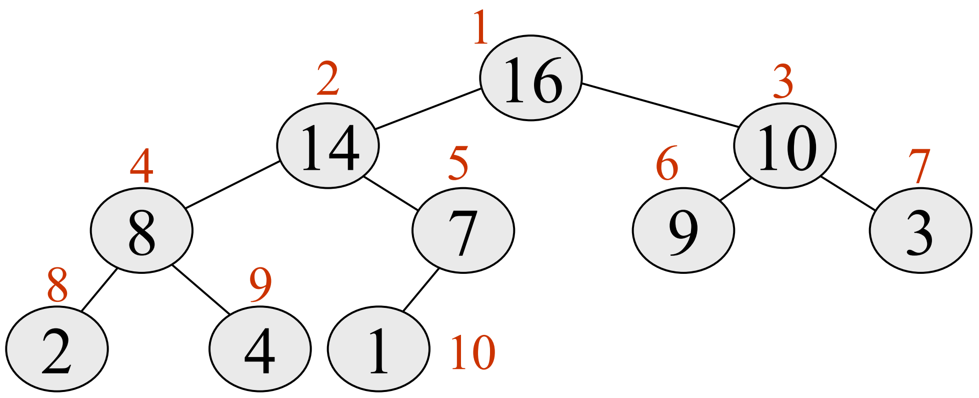
课程概要

学习内容

- ❑ 堆数据结构及其属性
- ❑ 维持堆属性
- ❑ 堆排序及其复杂度
- ❑ 通过堆实现优先队列

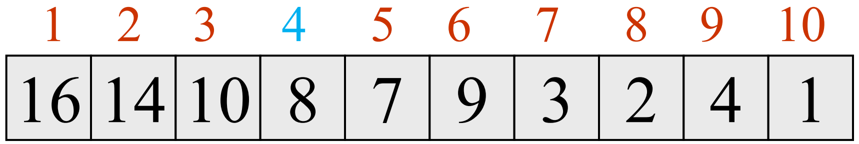
堆数据结构与堆排序

- 结合算法与数据结构
- 堆：一个可以视作**完整二叉树**的**数组**。
- 完整的二叉树指的是除最低层外(这一层只从左到右某个点进行填充)，在所有层级都进行填充的二叉树。

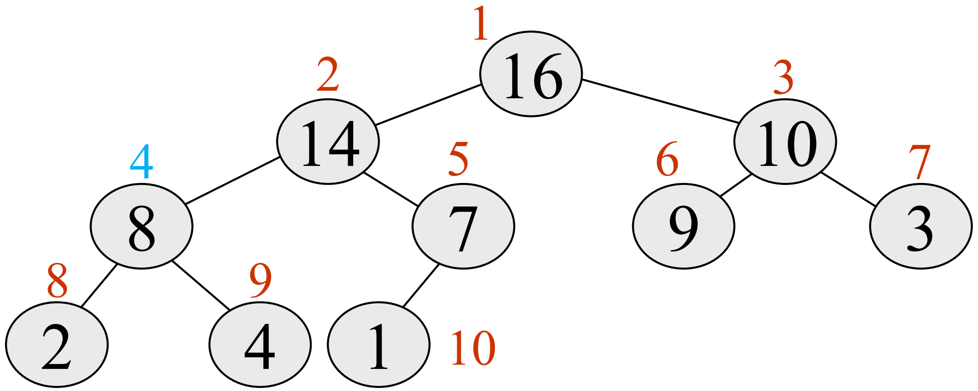


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

堆的属性

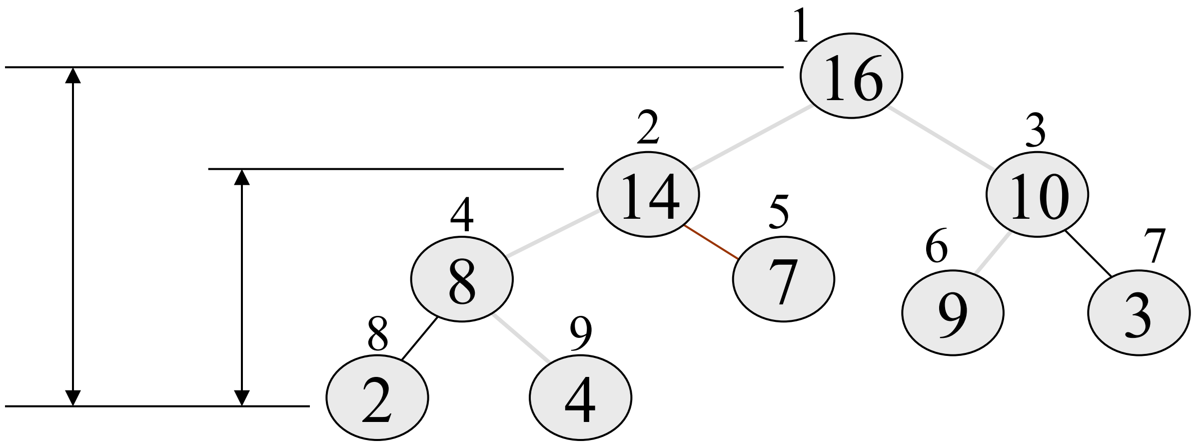


- 根节点是 $A[1]$
- 对任意一节点 i
- Parent(i)
  return $\lfloor i/2 \rfloor$ // 向右移动一位
- Left(i)
  return 2i // 向左移动一位
- Right(i)
  return 2i+1 // 向左移动一位并添加一位
- 母节点的数值要大于或等于其子节点的数值
  $A[\text{Parent}(i)] \geqslant A[i]$



堆的属性（接上页）

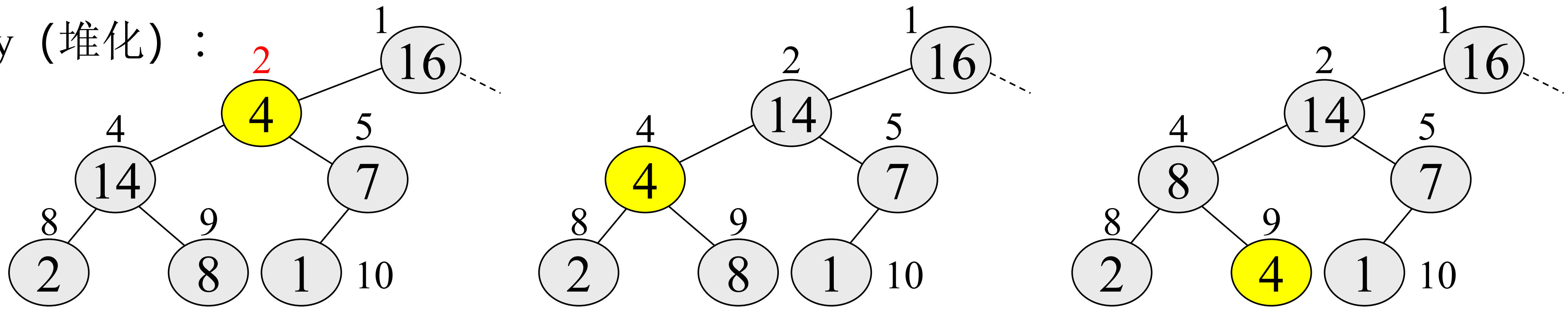
- 节点的高度是指到最远叶节点的边数。
- 树的高度是指其根的高度，即 $\Theta(\lg n)$，其中 n 是节点数。



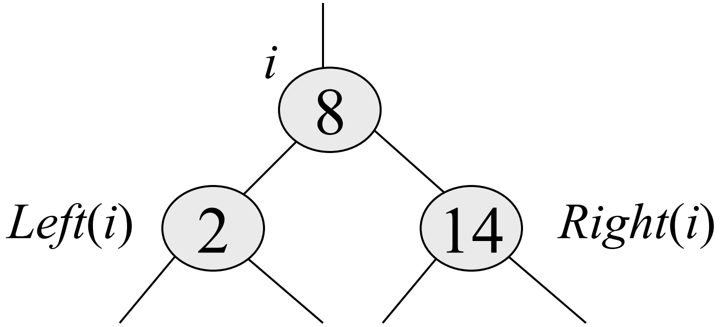
堆的排序过程

- 堆化 (A, i)，维持堆属性，假设子树 i 的两个子树是堆，但是 i 不是的情况下：O(lg$n$)
- 构建堆，从无序数组中生成堆：O(n)
- 堆排序，把数组排序： O($n$lg$n$)
- Extract-Max 和 Insert使得堆的数据结构可以作为优先队列：O(lg$n$)

Heapify（堆化）：



堆化算法的思路



1）查找具有最大值的节点

2）如果 i 具有最大值，则无需操作

如果 $Left(i)$ 具有最大值，则将其与 i 交换

如果 $Right(i)$ 具有最大值，则将其与 i 交换

3） 重复该过程，直至叶节点

堆的维持递归子程序

```
Heapify(A, i)
    L := Left(i)                                  // 左侧子节点
    R := Right(i)                                 // 右侧子节点
    if L ≤ heap-size(A) and A[L] > A[i]           // 存在左子树，而且
        then largest := L                         // 左侧子节点 > 母节点
        else largest := i                         // 否则，左侧子节点 ≤ 母节点
    if R ≤ heap-size(A) and A[R] > A[largest]     // 存在右子树，而且
        then largest := R                         // 右侧子节点 > 母节点
    if largest ≠ i                                // 一个子节点 > 母节点
        then swap (A[i], A[largest])
            Heapify(A, largest)
```

唯一递归/循环，最大迭代次数为该子树的高度：$\Theta(\lg n)$

堆的构建

- 思路是自下而上地调用 Heapify 的过程。
- 从叶节点之上的层级（叶节点总会构成一个堆）。

```
Build-Heap(A)
1.    n := length(A)
2.    for i := ⌊n/2⌋ downto 1
3.        do Heapify(A, i)
```

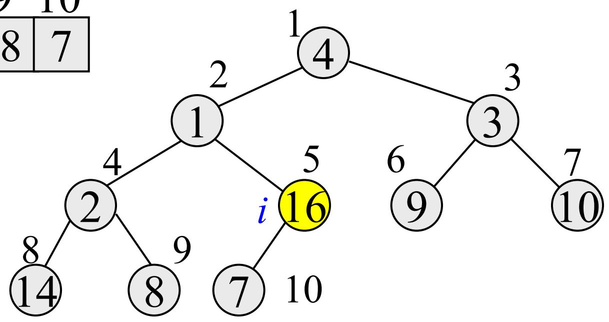
有 $n/2$ 个节点为叶节点，无需调用

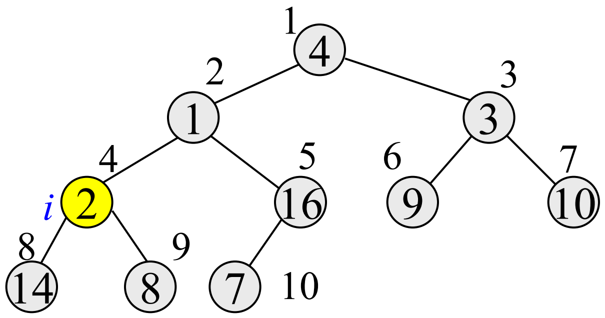
- 复杂度是：$(1/2)n\lg n = O(n\lg n)$
- 这是渐进紧界吗？
- 不是！
- 在子程序 Heapify(A, i) 中，如果 $i = n/2 ..1$，可以证明 $\Rightarrow O(n)$

示例：构建一个堆

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

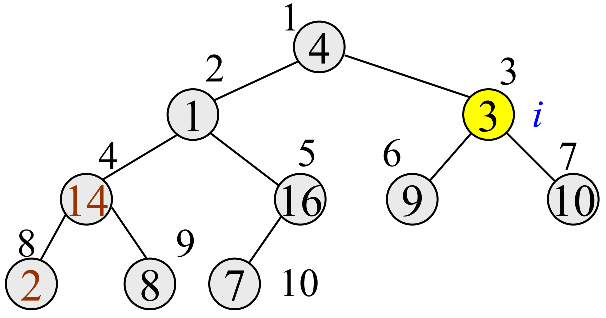
n/2 = 5

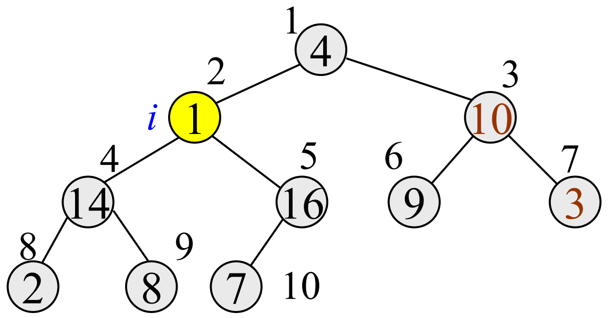




n/2-1 = 4

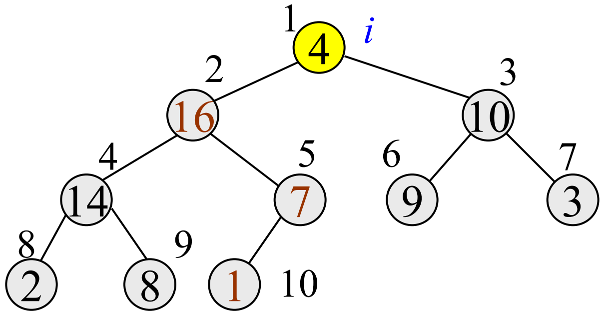
n/2-2 = 3

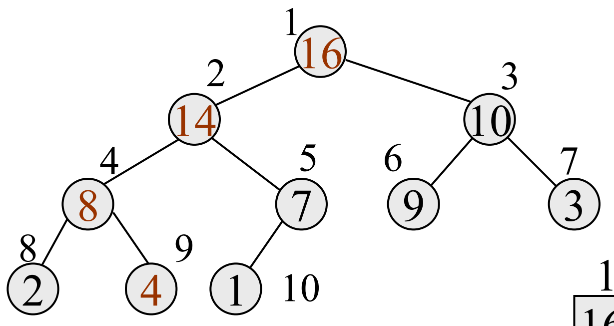




n/2-3 = 2

n/2-3 = 1





1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

堆排序算法

1. $A[1]$ 为最大数字 ⇒ $A[n]$
2. 将堆 A[1..n-1] 维护成堆
3. 重复第二步，直至所有元素均已排序

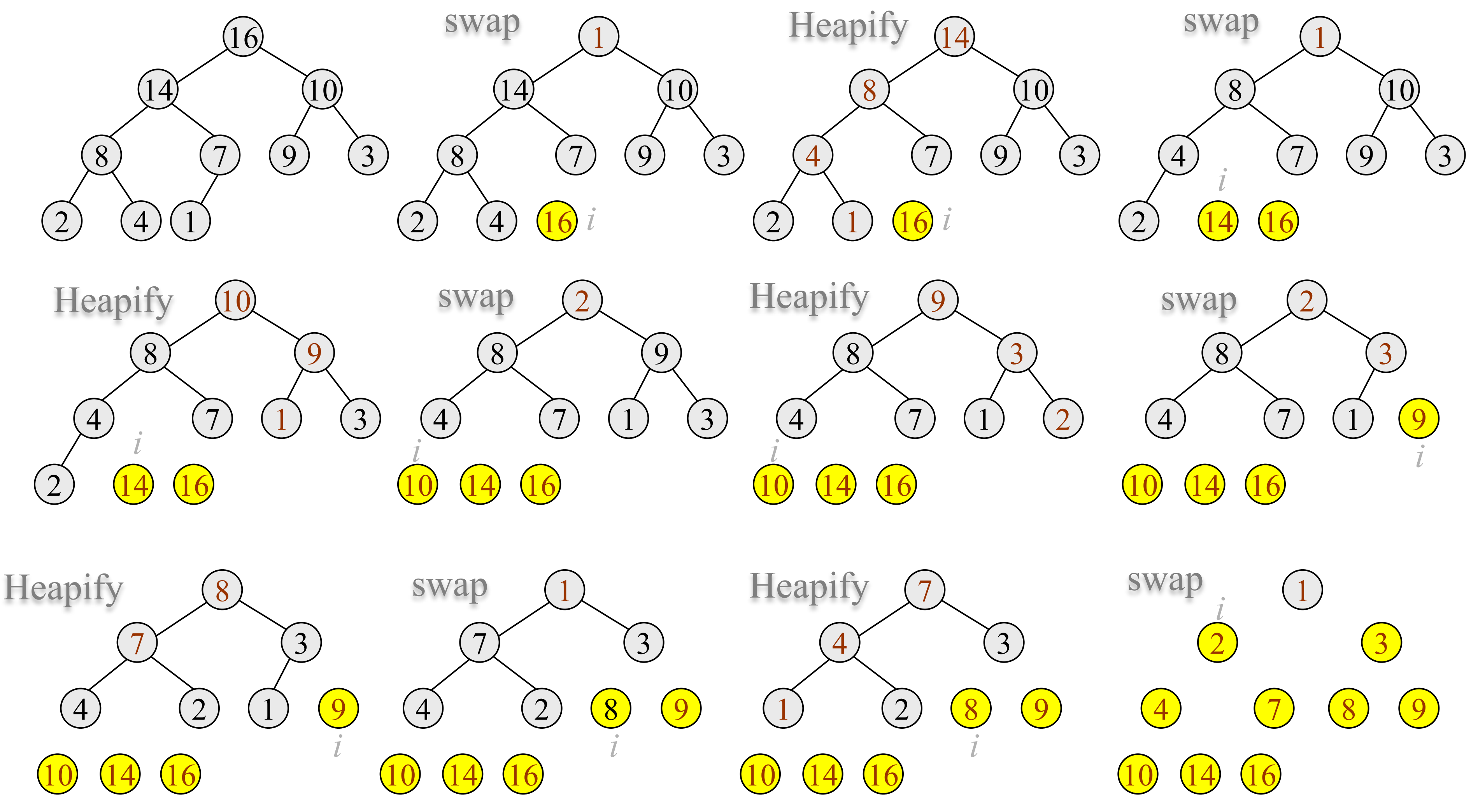
Heapsort(A)

1. Build-Heap(A) ⟶ $O(n)$
2. **for** i := n **downto** 2 **do** ⟶ n
3. swap(A[1], A[i]) ⟶ n-1
4. Heapify(A, i) ⟶ $(n\text{-}1)O(\lg n)$

- 复杂度是：

$$O(n) + n + (n - 1) + (n - 1)O(\lg n)) = O(n\lg n)$$

示例：堆排序



1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16

比较排序算法

算法	最坏情况运行时间	是否原地排序？
插入排序	$O(n^2)$	是
归并排序	$O(n\lg n)$	否（需要更多空）
快速排序	$O(n^2)$	是
堆排序	$O(n\lg n)$	是

堆排序

- 最坏情况的运行时间与归并排序的相同
- 与快速排序一样原地排序

快速排序

- 性能要优于 $O(n^2)$ 很多。
- 平均情况运行时间为 $O(n\lg n)$，而不是 $O(n^2)$。
- 在实践中，堆排序的性能要优于所有其他排序算法（隐藏常量较小）

基于堆的优先队列

优先队列是用于维持一个元素队列的数据结构，其中，每个元素都有一个称为“**key**”（优先级）的关联值

优先队列支持：

- Insert(S, x)：把 x 插入到队列 S 中 // S := S ∪ {x}
  O(lg*n*)
- Maximum(S)：返回带有最大值的元素
  O(1)
- Extract-Max(S)：返回并移除带有最大值的元素。O(lg*n*)

在操作系统中，要完成的任务被安排在优先队列中

- 任务到达：Insert(S, x)
- 任务分派：Extract-Max(S)

通过堆实现优先队列的操作

优先队列可以通过堆轻松实现

Heap-Maximum(A)

    return A[1] → $\Theta(1)$

Heap-Extract-Max(A)

1. **if** heapsize(A) < 1
2. **then** error("heap underflow")
3. max := A[1]
4. A[1] := A[heapsize(A)] // 把最后一个元素放到第一
5. heapsize(A) := heapsize(A) – 1 // 忽略最后一个元素
6. Heapify(A, 1) → $\Theta(\lg n)$ → 复杂度是 $\Theta(\lg n)$
7. return max

实现优先队列的操作（接上页）

Heap-Insert(A, key)

1. heapsize(A) := heapsize(A) + 1
2. i = heapsize(A)
3. **while** i > 1 and A[Parent(i))] < key **do**
   A[i] := A[Parent(i)]
4. i := Parent(i)
5. A[i] := key

$\Theta(\lg n)$

树的高度