目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） 课时安排: 2学时

第五讲：堆排序 授课时间 :第五周周一第1、2节

教学目标、要求（分掌握、熟悉、了解三个层次）：**了解堆的定义；熟悉维护堆的性质；掌握建堆过程；掌握堆排序算法；了解优先队列** 教学内容**（包括基本内容、重点、难点）：

基本内容：（1）堆的定义：二叉树；最大堆；最小堆；完全二叉树。

（2）维护堆的性质：极大堆过程

（3）建堆：建最大堆过程

（4）堆排序算法：算法程序；堆排序算法时间复杂度

（5）优先队列

教学重点、难点：重点为堆排序算法；难点为维护堆的性质的算法。

教学媒体的选择：本章使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

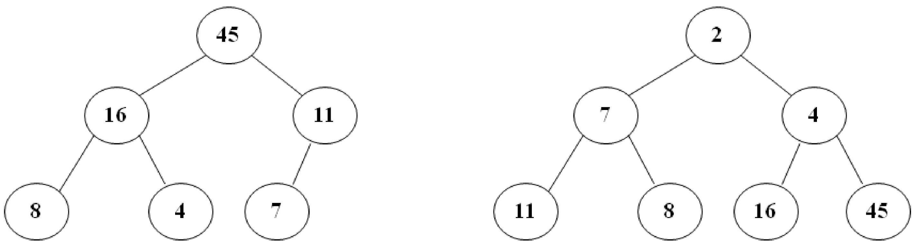
板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论；（2）举例说明；（3）程序设计与编译；（4）对本课堂进行总 结、讨论；（5）布置作业与实验报告

第五讲 堆排序

一. 堆 (Heap)

定义 1 堆是一个数组，它可以被看成一棵完全二叉树，树中每一个结点的值都不小于(或不大于)孩子结点的值.



1.最大堆：父亲结点的值不小于孩子结点的值

$$A[Parent(i)] >= A[i]$$

2.最小堆：父亲结点的值不大于孩子结点的值

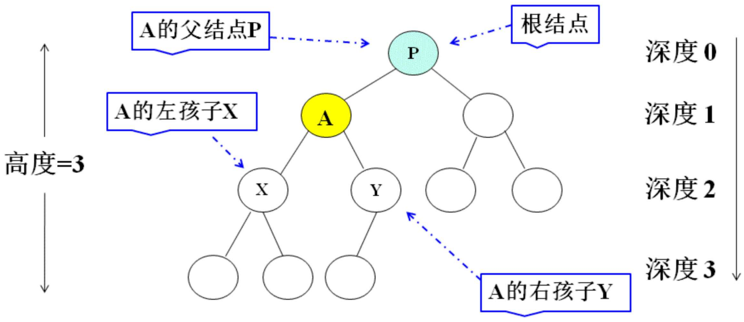
$$A[Parent(i)] <= A[i]$$

完全二叉树：

1.叶结点：没有孩子的结点;

2.内部结点：除叶结点之外的结点;

3.完全二叉树：除最下一层外，其余层的结点个数都达到当层最大结点数，且最下一层中从左至右连续存在结点，这些连续结点右边的结点全部不存在.



把堆看成一个棵树，有如下的特性：

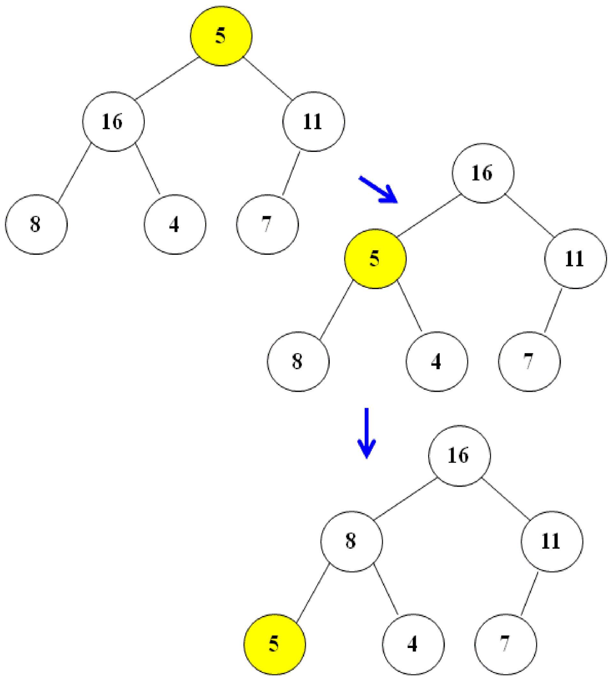
- 含有n个元素的堆的高度是lgn。
- 当用数组表示存储了n个元素的堆时，叶子节点的下标是n/2+1，n/2+2，……，n。
- 在最大堆中，最大元素该子树的根上；在最小堆中，最小元素在该子树的根上。

2. 维护堆的性质(个别调整)

堆个关键操作过程是如何保持堆的特有性质，给定一个节点i，要保证以i为根的子树满足堆性质。书中以最大堆作为例子进行讲解，并给出了递归形式的保持最大堆性的操作过程MAX-HEAPIFY。先从看一个例子，操作过程如下图所示：

伪代码

```
 Max-Heapify (A, i)
k=Left ( i )
r=Right ( i )
if  k<=A.heap-size
        and  A[k] > A[i]
    largest = k
else largest = i
if  r<= A.heap-size
         and A[r] > A[largest]
    largest = r
if  largest != i
 exchange A[i]  with  A[largest]
 Max-Heapify(A, largest)
```



堆维护的C语言程序

In [ ]:

```
#C语言代码:
//对heap数组在[low, high]范围进行向下调整
//其中low为欲调整结点的数组下标，high为最后元素下标
void  downAdjust (int  low,  int  high)  {
       int  i = low,  j = i*2;          //i为欲调整结点, j 为其左孩子
       while (j <= high)  {            //存在孩子结点
             //如果右孩子存在，且右孩子的值大于左孩子
             if (j+1 <= high  &&  heap[j+1] > heap[j] )
                   { j = j+1; }                 //让 j 存储右孩子下标
             //如果孩子中最大的权值比欲调整结点i大
             if (heap[j] > heap[i])
                  { swap (heap[j], heap[i]);  //交换最大值的孩子与结点  i
                   i = j;
                   j = i*2; }
              else { break; }
         }}
```

Max-Heapify的时间复杂度:

1.调整$A[i]$和$A[Left(i)]$和$A[Right(i)]$的时间代价为$\Theta(1)$；

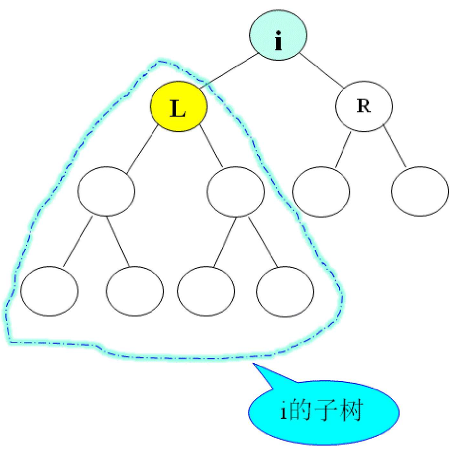
2.以i的一个孩子为根结点的$Max-Heapify$的时间代价至多$2n/3$(最低层半满).

从而Max-Heapify的运行时间为:

$$T(n) \leq T(2n/3) + \Theta(1)$$

利用主定理：$T(n) = O(lgn)$.

因为树高$h = lgn$, 一般地，$T(n) = O(h)$.



3. 建堆

建立最大堆的过程是自底向上地调用最大堆调整程序将一个数组 $A[1.....N]$ 变成一个最大堆。将数组视为一颗完全二叉树，从其最后一个非叶子节点（ $n/2$ ）开始调整。调整过程如下图所示：

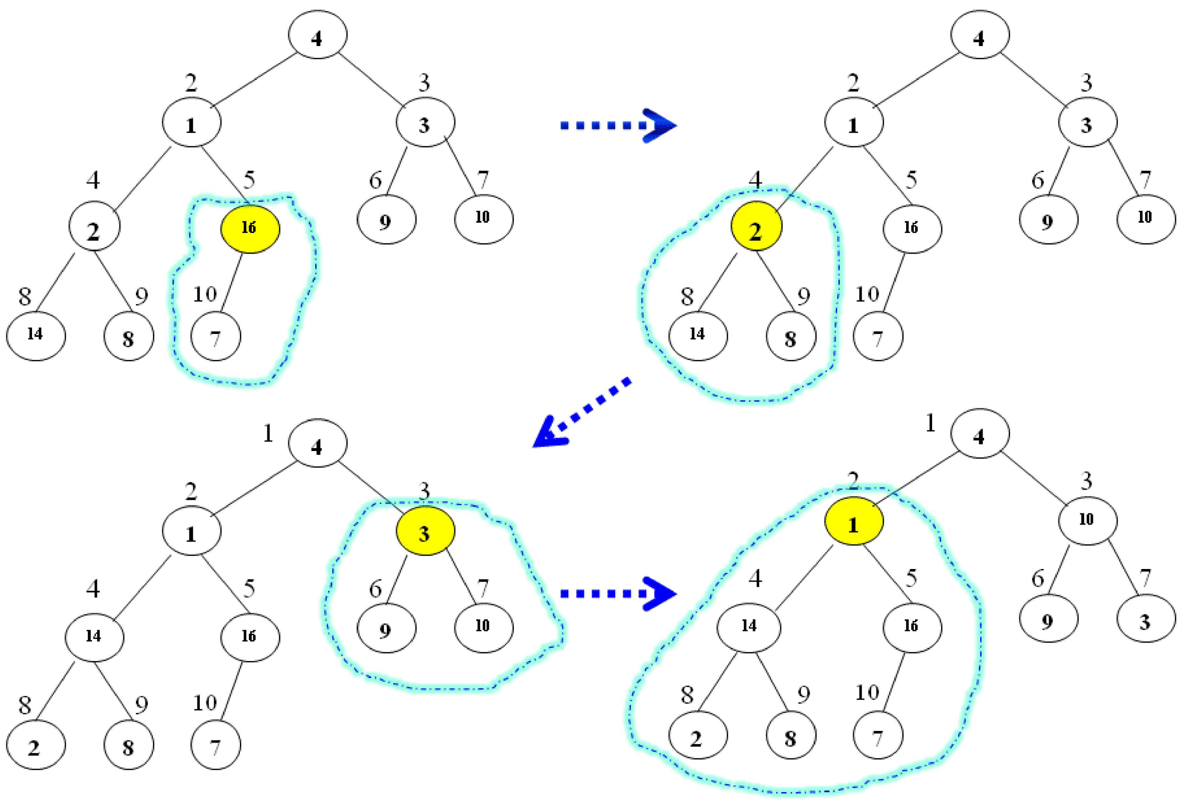
伪代码：

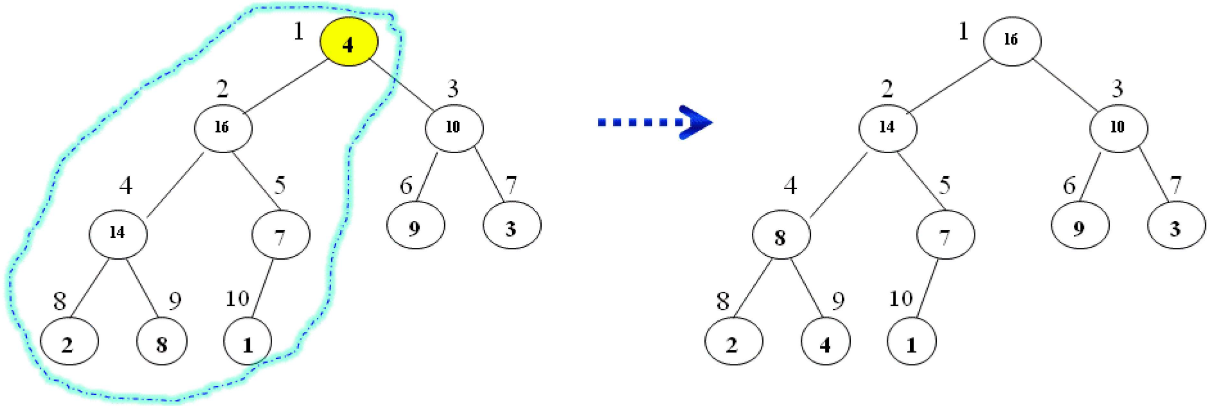
```
Build-Max-Heap(A)
A.heap-size = A.length
for  i= A.length/2 downto 1
       Max-Heapify (A,i)
```

C语言代码：

```
void  createHeap ( )  {
   for (int  i = n/2; i >= 1; i--) {
       downAdjust(i, n); } }
```

例1 设 A={4, 1, 3, 2, 16, 9, 10, 14, 8, 7},构造最大堆





注记:

1.以上过程将无序数组A={4, 1, 3, 2, 16, 9, 10, 14, 8, 7}构造成一个最大堆;

2.我们也可一通过调用Build-Min-Heap来构建最小堆.

建堆的时间复杂度:

1.包含n个元素的堆的高度为 $\lfloor lgn \rfloor$;

2.高度为h的堆最多包含 $\lceil n/2^{h+1} \rceil$;

3.在高度为h的结点运行$Max-Heapity$的代价$O(h)$;

$$\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h}) = O(n)$$

建堆的时间复杂度为 $O(n)$

4. 堆排序算法

伪代码:

```
HeapSort(A)
Build-Max-Heap(A)
For  i = A.length  downto  2
Exchange A[1]  with  A[i]
A.heap-size = A.heap-size  -  1
Max-Heapify (A, 1)
```

C语言代码:

```
void  heapSort( )  {
     createHeap ( );
     for (int  i = n; i > 1; i--)  {
           swap(heap[i], heap[1]);
           downAdjust(1, i-1);  } }
```

堆排序Python代码：

In [4]:

```python
#堆排序python程序
import math
def print_tree(array):
    '''
        前空格元素间
    170
    237
    313
    4   01
    '''
    index = 1
    depth = math.ceil(math.log2(len(array))) # 因为补0了，不然应该是math.ceil(
    sep = '  '
    for i in range(depth):
        offset = 2 ** i
        print(sep * (2 ** (depth - i - 1) - 1), end='')
        line = array[index:index + offset]
        for j, x in enumerate(line):
            print("{:>{}}".format(x, len(sep)), end='')
            interval = 0 if i == 0 else 2 ** (depth - i) - 1
            if j < len(line) - 1:
                print(sep * interval, end='')
        index += offset
        print()
```

In [6]:

```python
# Heap Sort
# 为了和编码对应，增加一个无用的0在首位
origin = [0, 30, 20, 80, 40, 50, 10, 60, 70, 90]
total = len(origin) - 1  # 初始待排序元素个数，即n
print(origin)
print_tree(origin)
print("="*50)
def heap_adjust(n, i, array: list):
    '''
    调整当前结点(核心算法)
    调整的结点的起点在n//2，保证所有调整的结点都有孩子结点
    :param n: 待比较数个数
    :param i: 当前结点的下标
    :param array: 待排序数据
    :return: None
    '''
    while 2 * i <= n:
        # 孩子结点判断 2i为左孩子，2i+1为右孩子
        lchile_index = 2 * i
        max_child_index = lchile_index  # n=2i
        if n > lchile_index and array[lchile_index + 1] > array[lchile_index]
            max_child_index = lchile_index + 1  # n=2i+1
        # 和子树的根结点比较
        if array[max_child_index] > array[i]:
            array[i], array[max_child_index] = array[max_child_index], array[i
            i = max_child_index  # 被交换后，需要判断是否还需要调整
        else:
            break
    # print_tree(array)
```

```
[0, 30, 20, 80, 40, 50, 10, 60, 70, 90]
              30
      20              80
  40      50      10      60
70  90
==================================================
```

In [8]:

```
# 构建大顶堆、大根堆
def max_heap(total,array:list):
    for i in range(total//2,0,-1):
        heap_adjust(total,i,array)
    return array
print_tree(max_heap(total,origin))
print("="*50)
# 排序
def sort(total, array:list):
    while total > 1:
        array[1], array[total] = array[total], array[1] # 堆顶和最后一个结点交
        total -= 1
        if total == 2 and array[total] >= array[total-1]:
            break
        heap_adjust(total,1,array)
    return array
print_tree(sort(total,origin))
print(origin)
```

```
                90
        80              70
    40      50      60      30
20  10
==================================================
                10
        20              30
    40      50      60      70
80  90
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

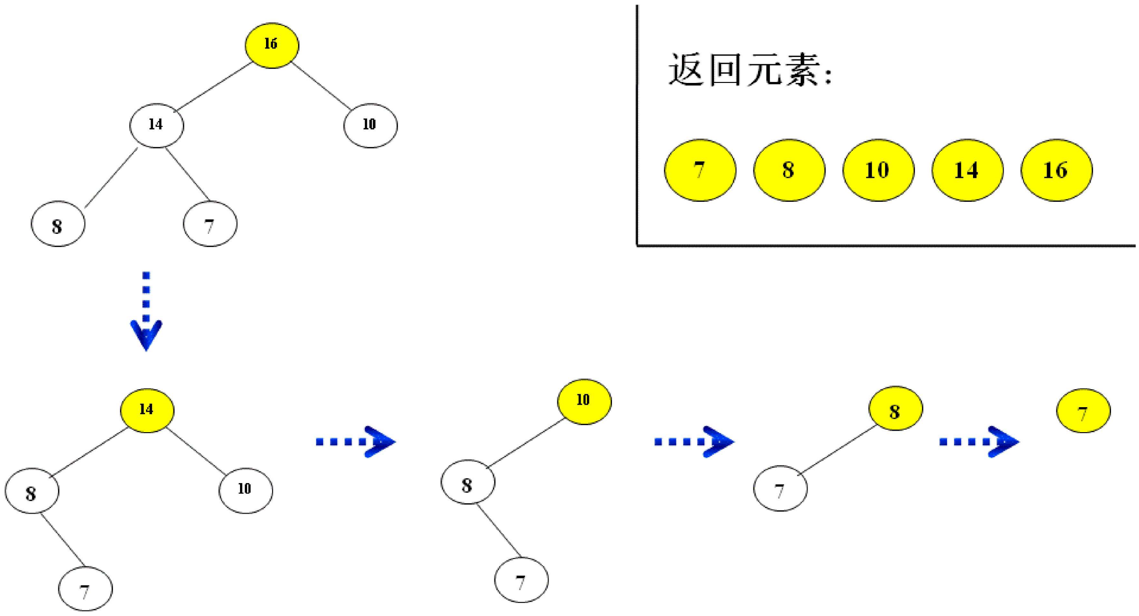
堆排序的时间复杂度：

- n个元素的建堆的时间复杂度为 $O(n)$；
- k个结点时，$A[1]$ 与 $A[k]$ 交换时间复杂度为 $\Theta(1)$；
- k个结点运行Max-Heapity的代价为 $O(lgk)$.

$$O(n) + \sum_{k=2}^{n}(O(lgk) + O(1)) = O(nlgn)$$

堆排序时间复杂度为 $O(nlgn)$

堆排序-操作过程：



5. 优先队列

FIFO 队列： 先到先出，先来先服务；

优先队列： 按重要性提供服务，分最大优先级和最小优先级.

关键字（key）： 表示优先级

最大优先队列支持以下操作：

1. 堆插入
2. 返回最大值（优先级最高者）
3. 删除最大者
4. 堆增值操作

堆提取（返回堆顶最大者）：

伪代码：

```
Heap-Extract-Max(A)
if  A.heap-size < 1
     error “heap  underflow”
max = A[1]
A[1] = A[A.heap-size]
A.heap-size=A.heap-size - 1
Max-Heapify (A, 1)
Return  max
```

C语言代码:

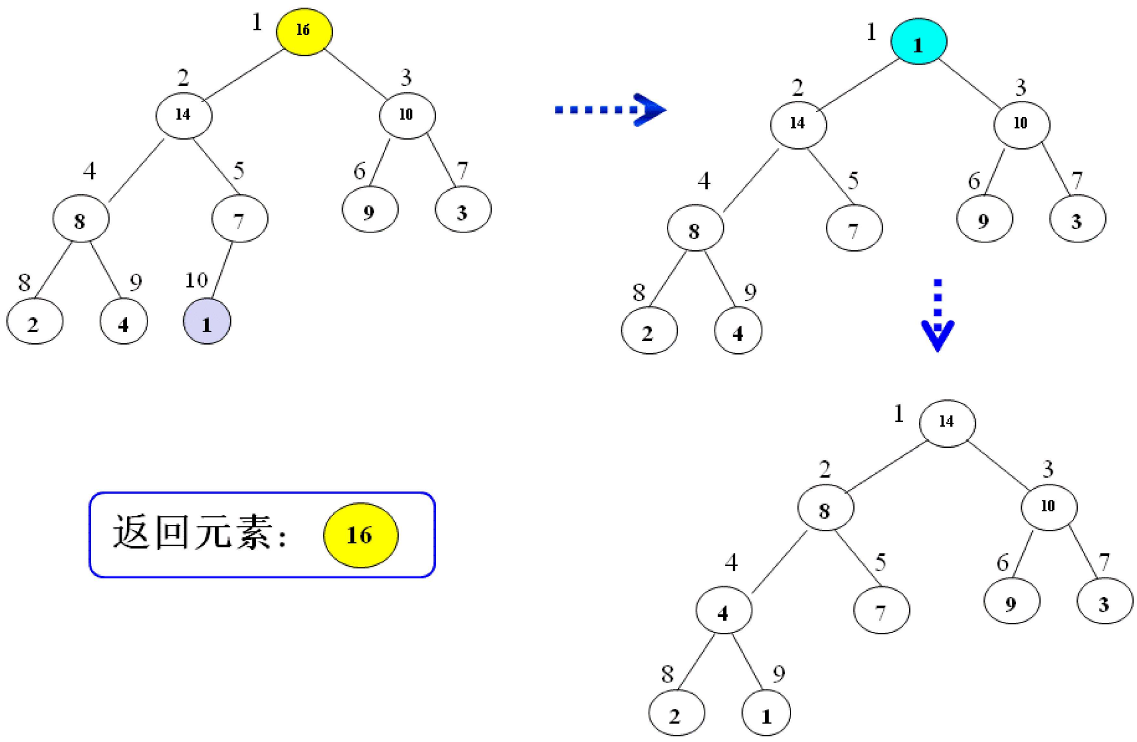
```
void  deleteTop( )  {
     heap[1]=heap[n--]
     downAdjust(1, n);    }}
```

堆提取的时间复杂度:

- n个元素的Max-Heapify的时间复杂度为 $O(lgn)$；
- 其余操作时间复杂度为 $\Theta(1)$.

堆提取时间复杂度为$O(lgn)$

堆提取最大者-操作过程:



堆增值、堆插入操作:

增值-伪代码:

```
Heap-Increase-Key (A, i, key)
if  key < A[i]
error  “new key is smaller”
A[i] = key
while  i>1
and  A[Parents(i)] > A[i]
exchange A[i] with A[parents(i)]
i = Parents[i]
```

插入-伪代码：

```
Max-Heap-Insert( A, key)
A.heap-size = A.heap-size -1
A[A.heap-size] = - ∞
Heap-Increase-Key(A, A.heap-size, key)
```

优先队列的时间复杂度：

- n个元素的堆增值的时间复杂度为 $O(lgn)$;
- n个元素的堆插入的时间复杂度为 $O(lgn)$.

优先队列运行时间为：$T(n) = O(lgn)$

```
#堆增值、堆插入操作：
#增值-C语言代码：
//对堆在[low, high]范围向上调整
//其中low一般设置为1
void upAdjust (int low, int high) {
   int i=high,  j = i/2;
   while (j >= low)  {
       if (heap[j] < heap[i])  {
           swap(heap[j], heap[i]);
           i = j;
           j = i/2;
       } else {
           break; } } }

#插入-C语言代码：
void  insert( int x) {
      heap[++n] = x;
      upAdjust (1, n);  }
```

引用及参考:

[1]《Python数据结构与算法分析》布拉德利.米勒等著，人民邮电出版社，2019年9月.
[2] https://www.cnblogs.com/Anker/archive/2013/01/23/2873422.html (https://www.cnblogs.com/Anker/archive/2013/01/23/2873422.html)

课后练习

1. 写出堆排序算法的完整Python代码或 C语言代码。
2. 举例说明堆排序的完整操作过程，并用Python代码实现该序列堆排序。
3. 证明堆排序算法的时间复杂度是 $\Omega(nlgn)$。

讨论、思考题、作业:

参考资料（含参考书、文献等）：算法导论. Thomas H. Cormen等，机械工业出版社，2017.

授课类型（请打√）：理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）：复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）：讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）：多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。

In [ ]:

```
1
```