7-补充 数学基础与数据结构

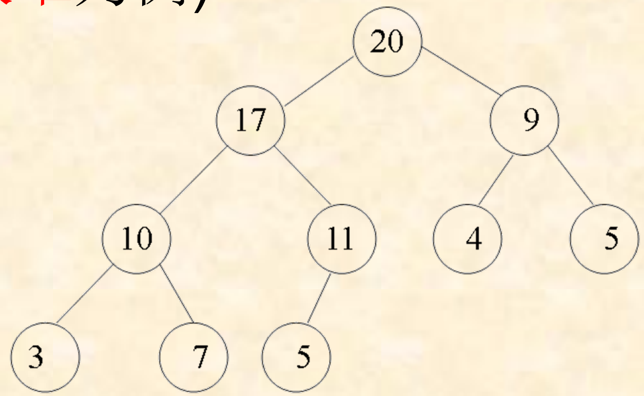
几种常用的数据结构

- **算法的实现离不开数据结构**。选择一个合适的数据结构对设计一个高效的算法有十分重要的影响。结构化程序设计创始人Niklaus Wirth (瑞士苏黎士高工)提出一个著名的论断："程序=算法+数据结构"。1984年，Wirth因开发了Euler、Pascal等一系列崭新的计算语言而荣获图灵奖, 有"结构化程序设计之父"之美誉。
- 首先学习基本的数据结构 (第三章，课本)
- 本章我们将回顾两种重要的数据结构：**堆(Heap)** 和**不相交集(Disjoint Sets)**。

堆(Heap)

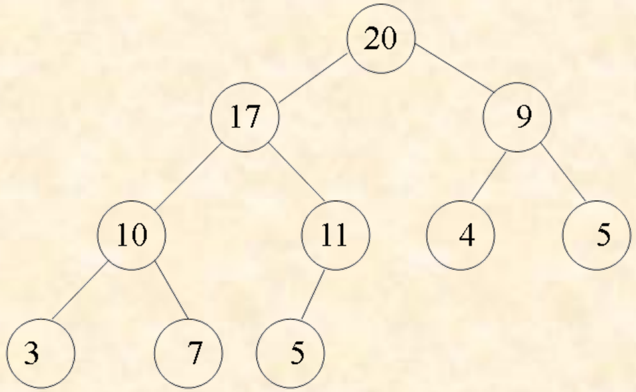
- 在许多算法中，需要大量用到如下两种基本操作：插入元素和寻找最大(小)值元素。为了提高这两种运算的效率，必须使用恰当的数据结构。
  - 普通队列：易插入元素，但求最大(小)值元素需要搜索整个队列。
  - 排序数组：易找到最大(小)值，但插入元素需要移动大量元素。
  - 堆：则是一种有效实现上述两种运算的数据结构。

- 堆的定义：堆是一个几乎完全的二叉树，每个节点都满足这样的特性：任一父节点的键值(key)不小于(大于等于)子节点的键值。(这里以最大堆为例)



- 有n个节点的堆T,可以用一个数组H[1…n]用下面的方式来表示：
  - T的根节点存储在H[1]中
  - 假设T的节点x存储在H[j]中，那么，它的左右孩子节点分别存放在H[2j]及H[2j+1]中(如果有的话)。
  - H[j]的父节点如果不是根节点，则存储在$H[\lfloor j/2 \rfloor]$中。

20	17	9	10	11	4	5	3	7	5
1	2	3	4	5	6	7	8	9	10



观察：

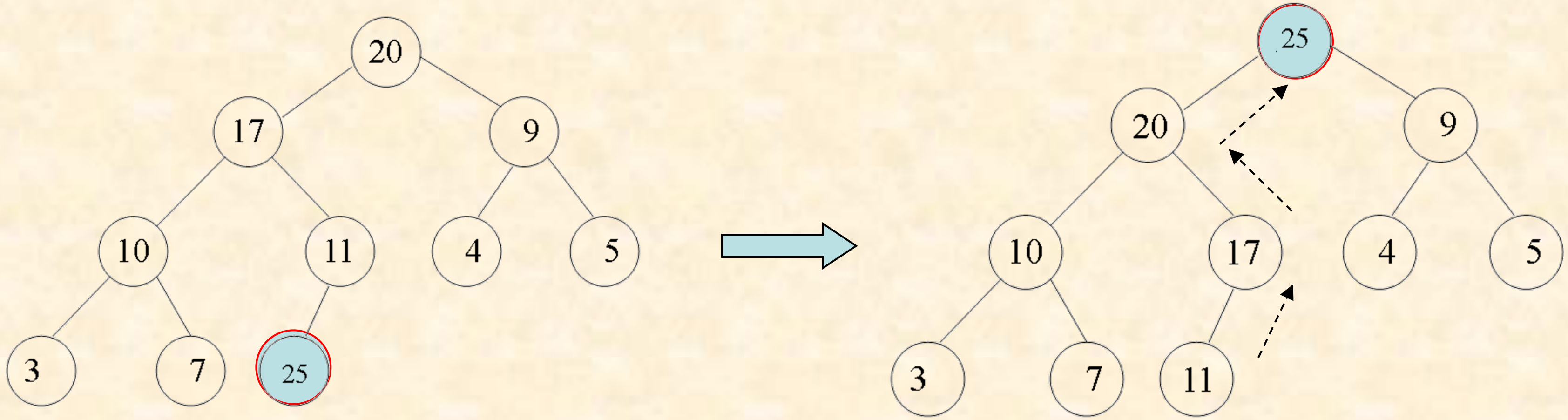
- 根节点键值最大，叶子节点键值较小。从根到叶子，键值以非升序排列。
- 节点的左右孩子节点键值并无顺序要求。
- 堆的数组表示呈“基本有序”状态（所以，堆又称为优先队列）。此外，并非节点的高度越高，键值就越大。

堆的基本操作

- make-heap(A): 从数组A创建堆
- insert(H, x): 插入元素x到堆H中
- delete(H, i): 删除堆H的第i项
- delete-max(H): 从非空堆H中删除最大键值并返回数据项

辅助运算Sift-up (以最大堆为例)

- 若某个节点H[i]的键值大于其父节点的键值，就违背了堆的特性，需要进行调整。
- 调整方法：上移。
- 沿着H[i]到根节点的唯一一条路径，将H[i]移动到合适的位置上：比较H[i]及其父节点$H[\lfloor i/2 \rfloor]$的键值，若$key(H[i])>key(H[\lfloor i/2 \rfloor])$，则二者进行交换，直到H[i]到达合适位置。



过程 Sift-up(H,i)

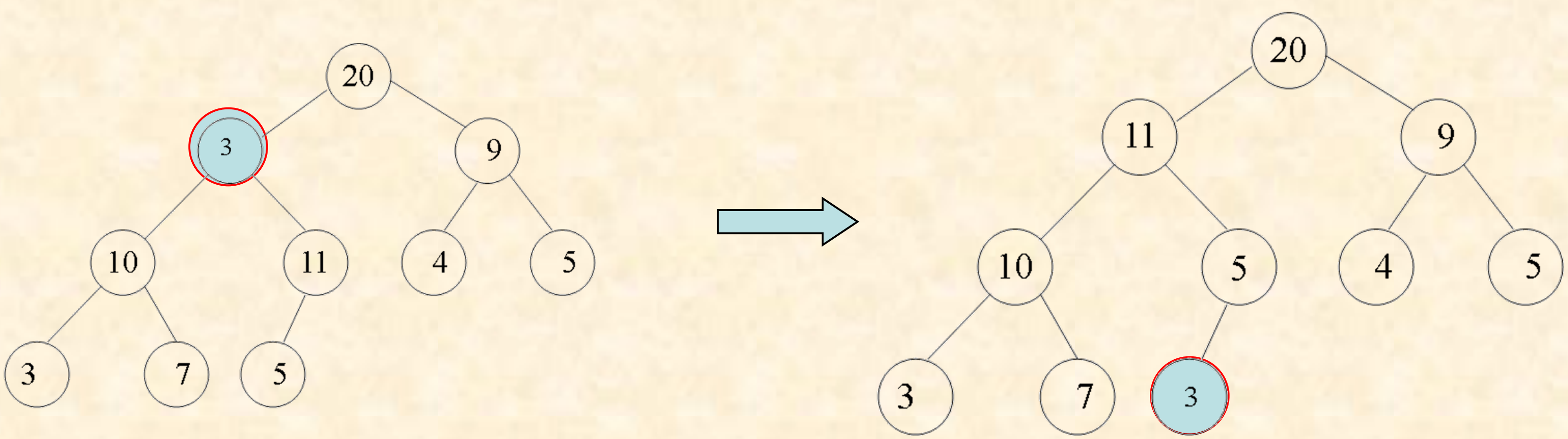
输入：数组H[1…n]，索引i ($1\leq i\leq n$)

输出：上移H[i] (如果需要)，使它的键值不大于父节点的键值

```
done ←false
if i=1 then exit {根节点，无需上移，直接退出}
repeat
    if key(H[i])>key(H[⌊i/2⌋]) then 互换 H[i] 和H[⌊i/2⌋]
    else done←true {调整过程至此已经满足要求，可退出}
    i←⌊i/2⌋
until i=1 or done {调整进行到根节点，或到某一节点终止}
```

辅助运算Sift-down

- 假如某个内部节点H[i] ($i \le \lfloor n/2 \rfloor$)，其键值小于其孩子节点的键值,即key(H[i])<key(H[2i])或key(H[i] <key(H[2i+1]) (如果右孩子存在)，违背了堆特性，需要进行调整。
- 调整方法：下渗。
- 沿着从H[i]到孩子节点(可能不唯一，则取其键值较大者)的路径，比较H[i]与孩子节点的键值，若key(H[i]) < max(H[2i], H[2i+1])则交换之。这一过程直到其成为叶子节点或满足堆特性为止。



```
过程 Sift-down(H,i)
输入：数组H[1…n]，索引i (1≤ i≤n)
输出：下渗H[i] (若它违背了堆特性)，使H满足堆特性
done←false
if 2i>n, then exit {叶子节点，无须下渗}
repeat
    i←2i
    if i+1<=n and key(H(i+1))> key(H(i)) then i=i+1
                          //有右孩子,取左右孩子中较大者
    if  key(H[⌊i/2⌋])< key(H[i]) then 互换 H[i] 和 H[⌊i/2⌋]
    else done ← true {调整过程至此已经满足堆特性，可退出}
    end if
until 2i>n or done {调整进行到叶节点，或到某一节点终止}
```

操作insert(H,x): 插入元素x到堆H中

- 思路：先将x添加到H的末尾，然后利用Sift-up，调整x在H中的位置，直到满足堆特性。

输入：堆H[1…n]和元素x
输出：新堆H[1…n+1]，x是其中元素之一。
1. $n \leftarrow n+1$ {堆大小增1}
2. $H[n] \leftarrow x$;
3. Sift-up(H,n) {调整堆}

树的高度为$\lfloor \log n \rfloor$,所以将一个元素插入大小为n的堆所需要的时间是O(logn)。

操作 delete(H,i)

- 思路：先用H[n]取代H[i]，然后对H[i]作Sift-up或Sift-down)，直到满足堆特性。

输入：非空堆H[1…n]，索引i，$1 \le i \le n$.
输出：删除H[i]之后的新堆H[1…n-1].

1. $x \leftarrow H[i]$; $y \leftarrow H[n]$;
2. $n \leftarrow n-1$; {堆大小减1}
3. if $i=n+1$ then exit {要删除的刚好是最后一个元素，叶节点}
4. $H[i] \leftarrow y$; {用原来的H[n]取代H[i]}
5. if key(y) ≥key(x) then Sift-up(H,i)
6. else Sift-down(H,i);
7. end if

类似地，容易知道delete操作所需要的时间是O(logn).

操作delete-max(H)

输入：堆H[1…n]
输出：返回最大键值元素，并将其从堆中删除
1. x←H[1]
2. delete(H,1)
3. return x

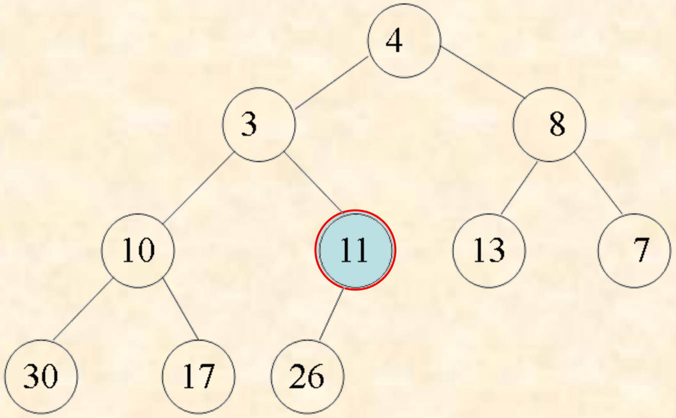
make-heap(A): 从数组A创建堆

- 方法1：从一个空堆开始，逐步插入A中的每个元素，直到A中所有元素都被转移到堆中。
- 时间复杂度为O(nlogn).为什么？(自行阅读教材)

方法2：

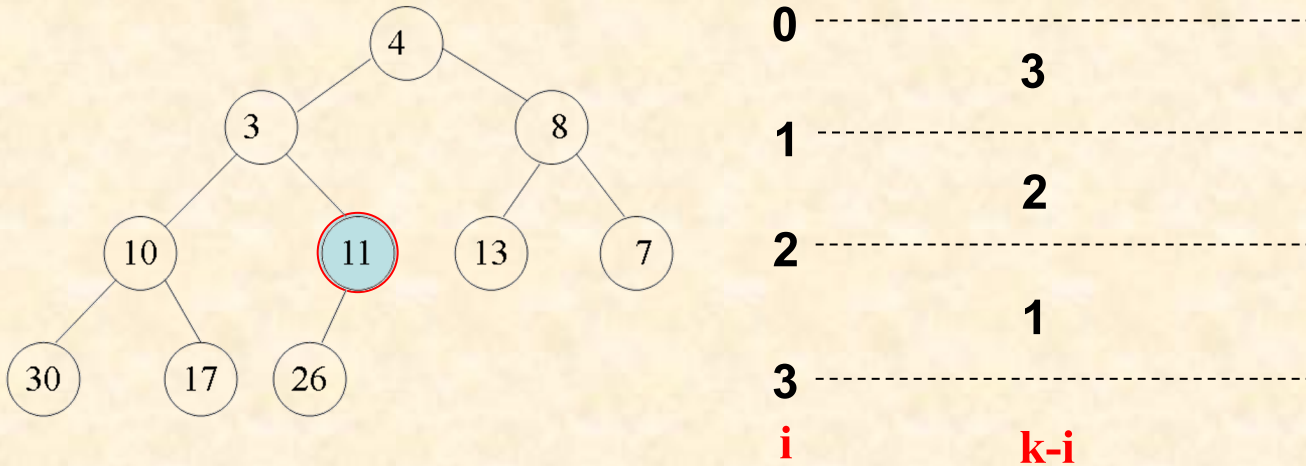
MAKEHEAP (创建堆)
输入：数组A[1…n]
输出：将A[1…n]转换成堆
1. for i←⌊n/2⌋ down to 1   //从第一个非叶子节点开始
2.   Sift-down(A, i) {使以A[i]为根节点的子树调整成为堆，故调用down过程}
3. end for

例：给定数组A[1…10] = {4, 3, 8, 10, 11, 13, 7, 30, 17, 26}



例：给定数组A[1…10] = {4, 3, 8, 10, 11, 13, 7, 30, 17, 26}

复杂度分析



- 树高k=$\lfloor \log n \rfloor$，第i层正好2^i个节点, $0 \le i < k$,(不含最深的叶子节点层), 每个节点的down过程最多执行k-i次,故down过程执行次数上限为

$$\sum_{i=0}^{k-1}(k-i)2^i = \sum_{j=k}^{1} j2^{k-j} \text{(令} k-i=j\text{)}$$

$$= 2^k \sum_{j=1}^{k} j2^{-j} = 2^k \Theta(1)$$

$$\le n \cdot \Theta(1)$$

- 时间复杂度为O(n).

$$\sum_{j=1}^{k} j2^{-j} = 1\cdot\frac{1}{2} + 2\cdot\frac{1}{2^2} + 3\cdot\frac{1}{2^3} + \cdots + k\cdot\frac{1}{2^k}$$

$$= \left\{ \begin{array}{r} 1\cdot\frac{1}{2} + 1\cdot\frac{1}{2^2} + 1\cdot\frac{1}{2^3} + \cdots + 1\cdot\frac{1}{2^k} \\ 1\cdot\frac{1}{2^2} + 1\cdot\frac{1}{2^3} + \cdots + 1\cdot\frac{1}{2^k} \\ 1\cdot\frac{1}{2^3} + \cdots + 1\cdot\frac{1}{2^k} \\ \vdots \qquad \\ 1\cdot\frac{1}{2^k} \end{array} \right.$$