INT202W04_排序算法，主定理

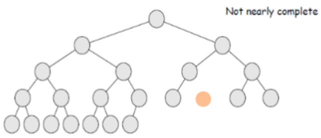
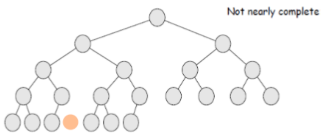
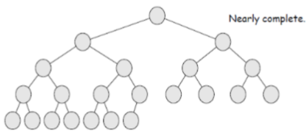
堆排序-堆（heap）

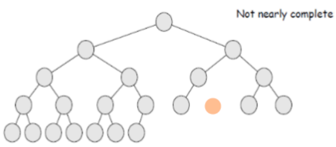
堆是 Priority Queue 的实现，它对插入和删除都有效。

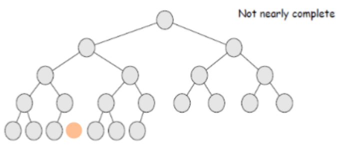
堆允许以对数时间执行插入和删除作。在堆中，元素及其键存储在几乎完整的二叉树中。二叉树的每个级别，除了最后一个级别，都将具有最大可能的子级数。

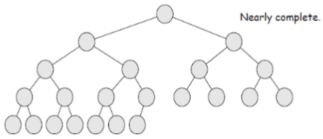
完全二叉树（Complete binary tree）

- 二叉树T是满的（full）：如果每个节点要么是叶节点或恰好拥有两个子节点
- 一个高度为h的完全二叉树在深度d上恰有2^d个节点（$0 \le d \le h$）
- 高度为h的几乎完全二叉树（nearly Complete Binary tree）有性质：在深度d上恰有2^d个节点（$1 \le d \le h-1$）；在深度d=h上节点尽可能靠左

> 
>
>
> 
>
>
> 
>

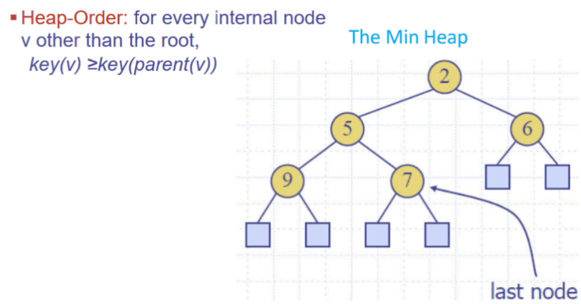






堆顺序（Heap order）

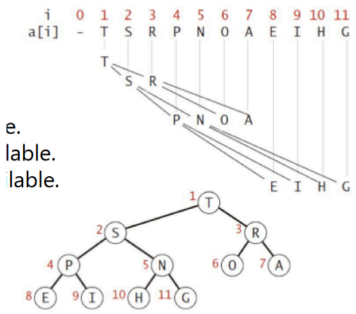
对于最小堆，对除了根节点，所有节点都比其父节点小（最大堆反之）



二叉堆（Binary heap）

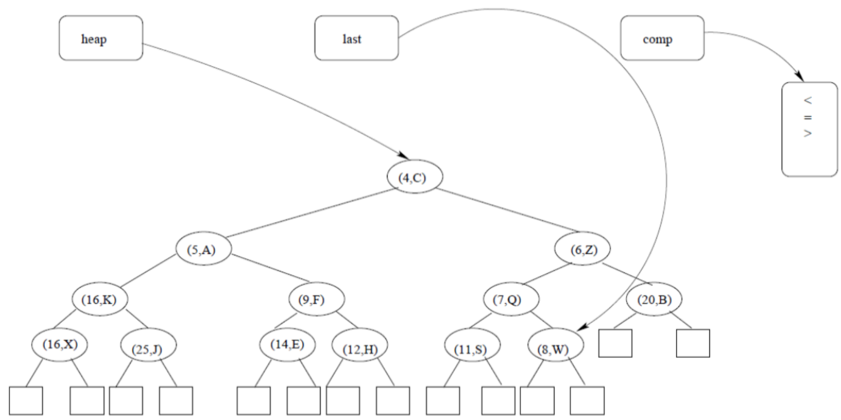
堆顺序完整二叉树的数组表示

对于在i的节点，左子节点在2i，右子节点在2i+1，父节点在[i/2]



Heap representations

优先级队列/ 堆的实现



heap：一个几乎完全的二叉树 T，包含键满足堆顺序的元素，存储在数组中。

last：对此数组表示形式中 T 的最后使用的节点的引用。

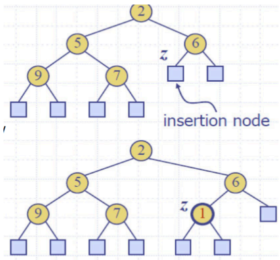
comp：一个比较器函数，它定义键上的总顺序关系，用于将最小（或最大）元素保持在 T 的根部。

插入（上堆冒泡 Up-heap bubbling）

方法`insertltem`将键为k的对象插入优先级队列，包括三步：

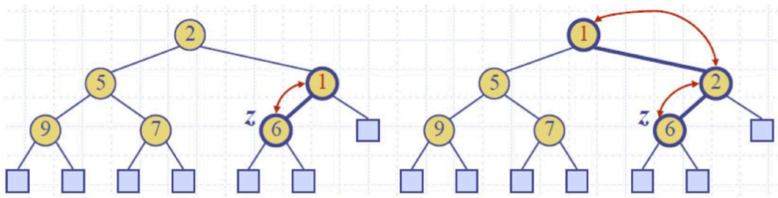
找到插入节点z；

将k储存在z处，并扩展为内部节点；



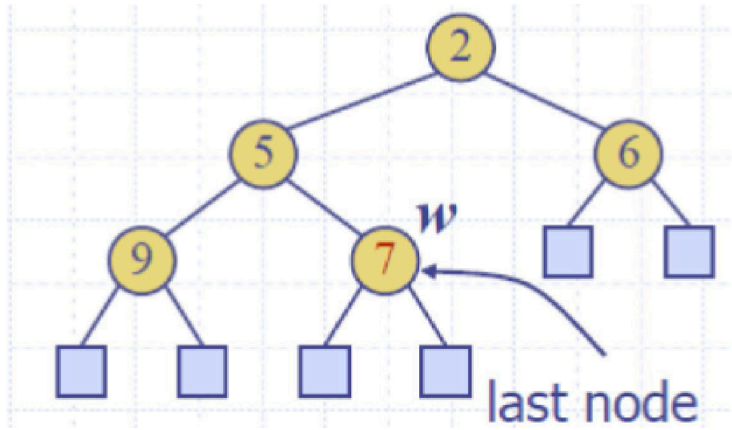
恢复heap-order。

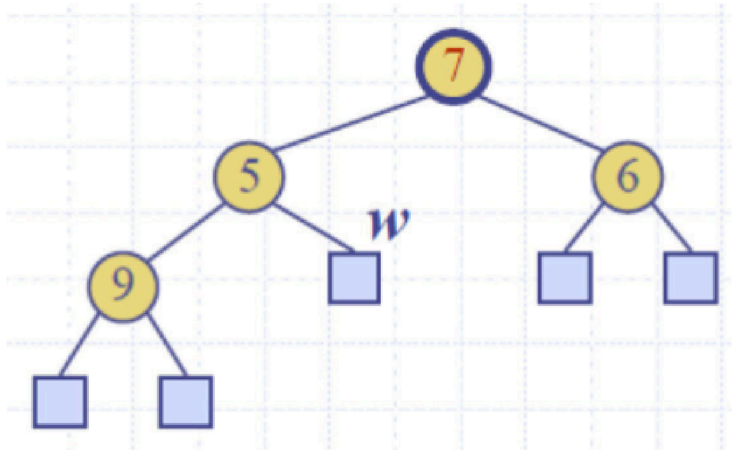
由于插入新节点后可能违反heap order，需要进行交换，由于高度是logn，故复杂度为O(log n)



移除（下堆冒泡 Down-heap bubbling）

如移除根节点（`removeMin`），将根节点替换为最后一个节点，然后向下冒泡





使用基于堆的优先级队列，我们可以在 $O(nlogn)$时间内对 n 个元素的序列进行排序。结果算法称为堆排序（heap-sort）

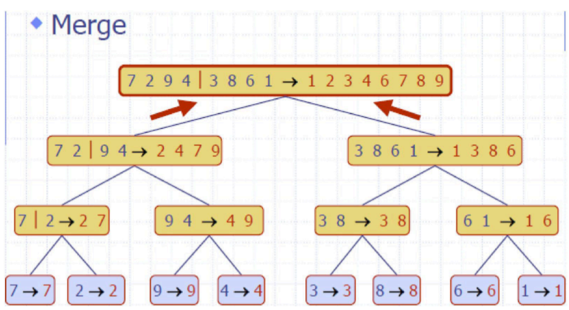
归并排序（MergeSort）

三个步骤：

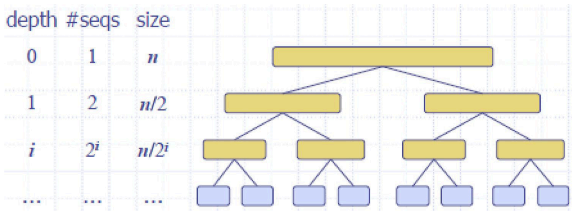
划分divide：把S分为S_1和S_2，各自拥有n/2个元素

递归recur：递归排序S_1和S_2

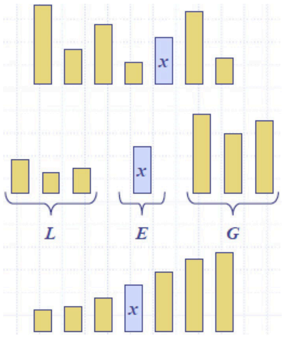
合并Conquer：将S_1和S_2合并为唯一的排序序列



复杂度：每次处理n个比较，工作logn次，故O(nlogn)



快排（QuickSort）

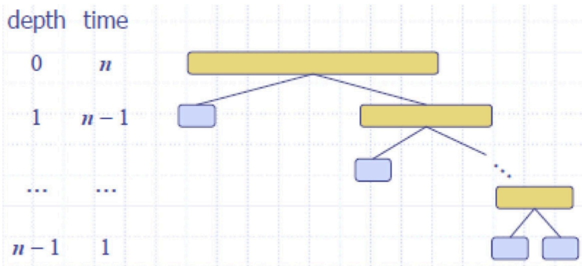


在一个无序的序列中选取一个任意的基准元素（pivot）E，将序列分为两部分，L部分元素小于等于E，G部分元素大于等于E。然后进行递归分别对前后两部分重复上述操作，直到将无序序列排列成有序序列。

不是一种稳定的排序算法，即多个相同的值的相对位置可能在算法结束时变动。

算法每搜索一次耗费n，在理想情况下（随机选择的中间数恰好等分序列）经过logn次划分可以把子表的长度切到1，故$O(nlog(n))$；

最坏情况是选择到了最小或最大值，需要划分n遍（n, n-1, ..., 2, 1），故$O(n^2)$



主定理

主定理（Master Theorem）是用于分析递归算法时间复杂度的重要工具。

> https://blog.csdn.net/cold_code486/article/details/134109090
>
> 有稍不一样的推广形式，2可以多乘log

由递推式得到复杂度：

$$\text{假设有递推式}T(n)=aT(\frac{n}{b})+f(n) \tag{1}$$
$$\text{其中：生成子问题数量}a>0\text{，子问题规模缩小倍数}b>0\text{，非递归开销}f(n) \tag{2}$$
$$\text{主定理比较非递归开销}f(n)\text{和递归开销}(n^{log_b a})\text{的大小确定谁占主导：} \tag{3}$$

$$1.\text{若有常数}\epsilon>0\text{能使}f(n)=O(n^{log_b(a-\epsilon)})\text{，则}T(n)=\Theta(n^{log_b a}) \tag{4}$$

递归调用开销占主导，子问题数量增长比额外处理开销快。大多数计算都发生在递归调用中，处理开销相对较小。

假设有$T(n)=4T(\frac{n}{2})+n,a=4,b=2,f(n)=n,n^{log_b a}=n^2,O(n^2)$

$$2.\text{若恰好}f(n)=\Theta(n^{log_b a})\text{，则}T(n)=\Theta(n^{log_b a}log(n)) \tag{5}$$

递归调用的总开销与非递归开销平分秋色，那么两部分一起决定复杂度，加一个额外的对数因子。

如归并排序$T(n)=2T(\frac{n}{2})+O(n)\Rightarrow f(n)=n=n^{log_2 2}=\Theta(n^{log_b a})\Rightarrow O(nlog(n))$

如$T(n)=T(2n/3)+1,a=1,b=2/3,f(n)=1,n^{log_b a}=1,O(logn)$

$$3.\text{若有常数}\epsilon>0\text{能使}f(n)=\Omega(n^{log_b(a+\epsilon)})\text{，且对足够大的}n\text{有常数}c<1\text{满足}af(\frac{n}{b})\leq cf(n) \tag{6}$$
$$\text{则}T(n)=\Theta(f(n)) \tag{7}$$

非递归开销f(n)占主导（且足够大），**非递归部分的增长比递归部分的累积增长更快。**

如$T(n)=T(n/3)+n$，$O(n)$

主定理不适用的情况：

1. 复杂度非几何倍数：$n^{log_b a}$比$f(n)$增长的快或慢，但没有快或慢$O(n^{\epsilon})$倍

如对递推式$T(n)=2T(n/2)+nlogn$，其中$f(n)=nlogn$，$n^{log_b a}=n$两者不成次数不可比。

2. 增长率不可比

代换技巧：

令$k=logn$，对递推式$\mathbf{T(n)=2T(n^{0.5})+logn}\Rightarrow T(2^k)=2T(2^{\frac{k}{2}})+k$

令$S(k)=T(2^k)\Rightarrow S(k)=2S(\frac{k}{2})+k$

$f(n)=k$，$k^{log_b a}=k$，$O(klog(k))\Rightarrow O(logn\cdot log(logn))$