堆与可并堆

twb@net9.org

优先队列

- 优先队列(Priority Queue):
  - 可以把元素加入到优先队列中, 也可以从队列中取出优先级最高的元素, 即以下ADT :
    - Insert(T, x)　　：把x加入优先队列中
    - DeleteMin(T, x)：获取优先级最高的元素x, 并把它从优先队列中删除
  - ADT　：Abstract Data Types

二叉堆(Binary Heap)

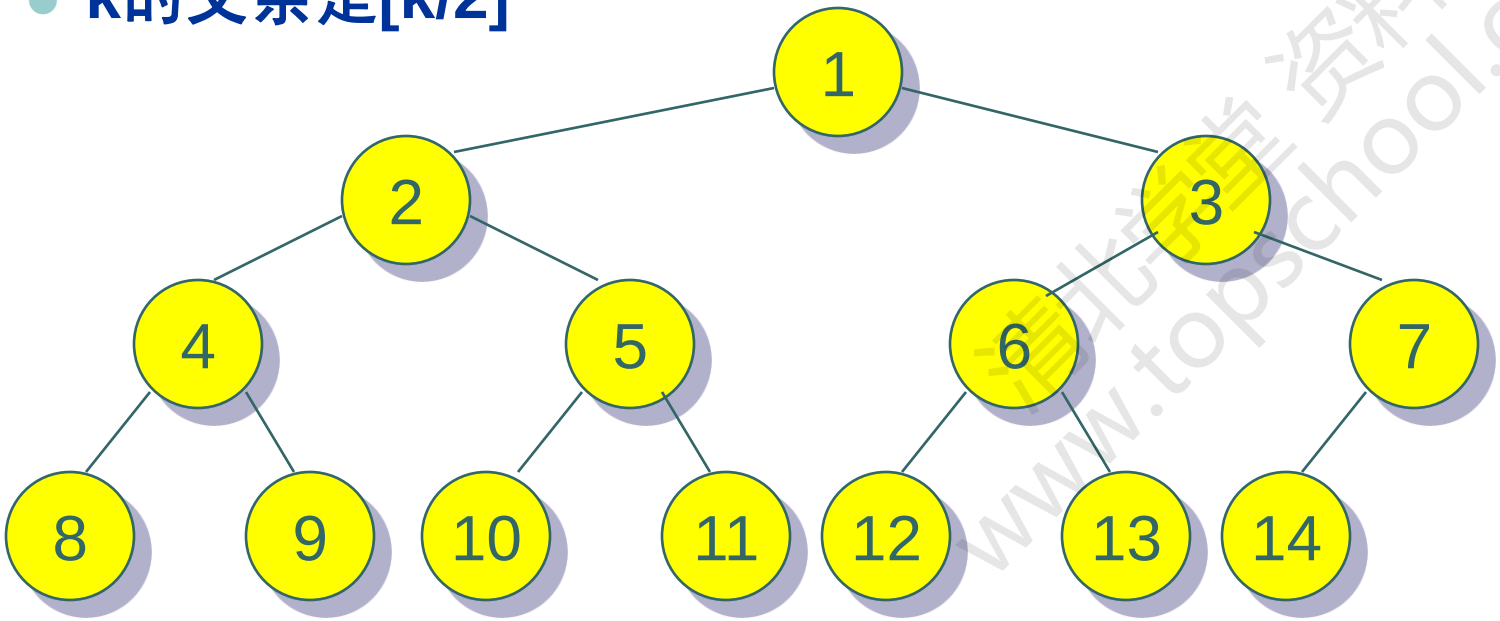
- 用二叉堆很容易实现优先队列
- 除了实现优先队列, 堆还有其他用途, 因此操作比优先队列多
  - Getmin(T, x) : 获得最小值
  - Delete(T, x) : 删除任意已知结点
  - ChangeKey(T, x, p) : 把x的优先级改为p
    - IncreaseKey(T, x, p)
    - DecreaseKey(T, x, p)
  - Build(T, x) : 把数组x建立成最小堆

堆的定义

- 堆是一棵完全二叉树
  - 所有叶子在同一层或者两个连续层
  - 最后一层的结点占据尽量左的位置
- 堆性质(递归定义)
  - 为空, 或者最小元素在根上
  - 两棵子树也是堆

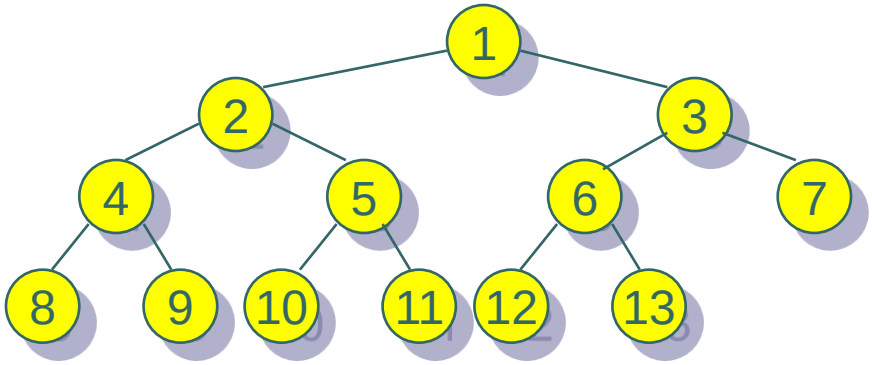
堆的存储方式

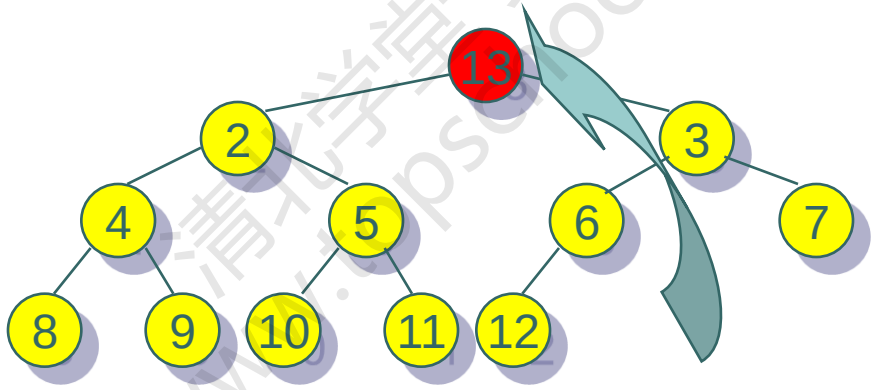
- 最小堆的元素保存在数组heap[1..heapSize]内
  - 根在heap[1]
  - k的左儿子是2k, k的右儿子是2k+1
  - k的父亲是[k/2]



删除最小值元素

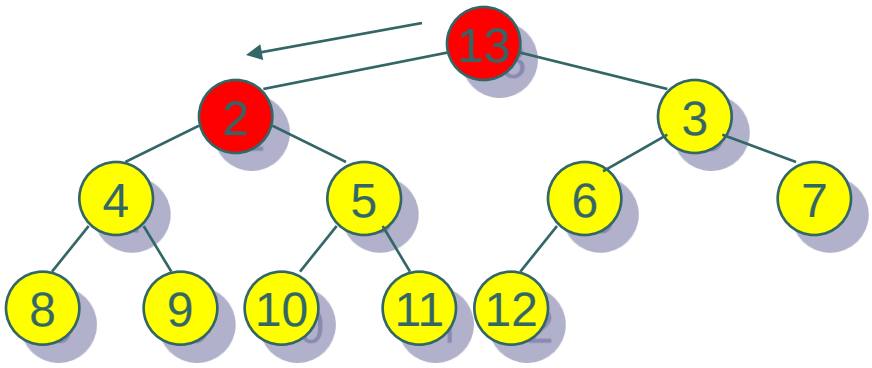
- 三步法
  - 直接删除根
  - 用最后一个元素代替根上元素
  - 向下调整

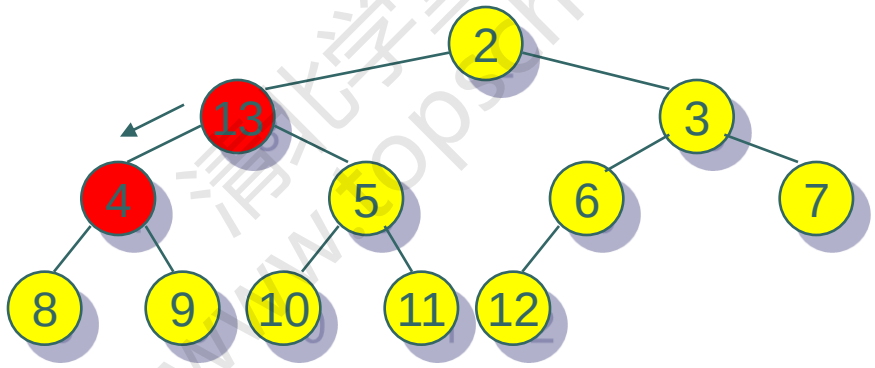




向下调整

- 当前节点的键值过大，需要向下调整
- 首先选取当前结点p的较小儿子.
  - 如果比p小, 调整停止
  - 否则交换p和儿子, 继续调整





插入元素和向上调整

- 插入元素:
  - 添加到末尾, 再向上调整
- 向上调整:
  - 当前节点键值过小, 需要向上调整
  - 比较当前结点p和父亲, 如果父亲比p小, 停止; 否则交换父亲和p, 继续调整

改变键值

- 改变一个点的键值后
  - 若键值变小, 则执行向上调整
  - 若键值变大, 则执行向下调整
- 删除任意元素
  - 用最后一个元素替代当前元素
  - 执行ChangeKey操作

堆的建立

- 从下往上逐层向下调整.
- 所有的叶子无需调整, 因此从heapSize/2开始.
- 可用数学归纳法证明循环变量为i时, 第i+1, i+2, …n均为最小堆的根

时间复杂度分析

- 向上调整/向下调整
  - 每层是常数级别, 共logn层, 因此O(logn)
- 插入/删除
  - 只调用一次向上或向下调整, 因此都是O(logn)
- 建堆
  - 高度为h的结点有$n/2^{h+1}$个,总时间为

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

堆的建立II – 插入建堆

- 每一次都插入一个新元素
- 则最坏情况每次都是log(heapSize)时间
- 总时间为O(logn!) = O(nlgn)

例1. k路归并问题

- 把k个有序表合并成一个有序表.
- 元素共有n个.

例1. k路归并问题分析

- 每个表的元素都是从左到右移入新表
- 把每个表的当前元素放入二叉堆中, 每次删除最小值并放入新表中, 然后加入此序列的下一个元素
- 每次操作需要logk时间, 因此总共需要nlogk的时间

例2. 序列和的前n小元素

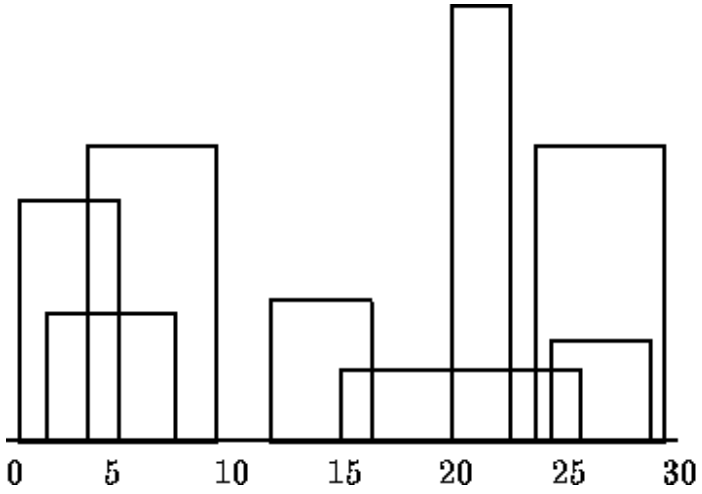
- 给出两个长度为n的有序表A和B,
- 在A和B中各任取一个, 可以得到n^2个和.
- 求这些和中最小的n个

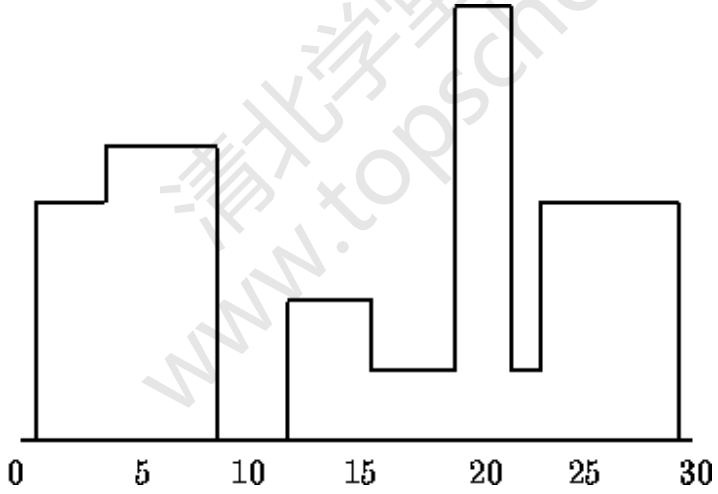
分析

- 可以把这些和看成n个有序表:
  - A[1]+B[1] <= A[1]+B[2] <= A[1]+B[3] <=…
  - A[2]+B[1] <= A[2]+B[2] <= A[2]+B[3] <=…
  - …
  - A[n]+B[1] <= A[n]+B[2] <= A[n]+B[3] <=…
- 类似刚才的算法, 每次O(logn), 共取n次最小元素,共O(nlogn)

例3. 轮廓线

- 每一个建筑物用一个三元组表示(L, H, R), 表示左边界, 高度和右边界
- 轮廓线用X, Y, X, Y…这样的交替式表示
- 右图的轮廓线为: (1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0)
- 给N个建筑，求轮廓线





分析

- 算法一: 用数组记录每一个元线段的高度
  - 离散化, 有n个元线段
  - 每次插入可能影响n个元线段, O(n), 共$O(n^2)$
  - 从左到右扫描元线段高度, 得轮廓线
- 算法二：每个建筑的左右边界为事件点
  - 把事件点排序, 从左到右扫描
  - 维护建筑物集合, 事件点为线段的插入删除
  - 需要求最高建筑物, 用堆, 共O(nlogn)

例4. 丑数

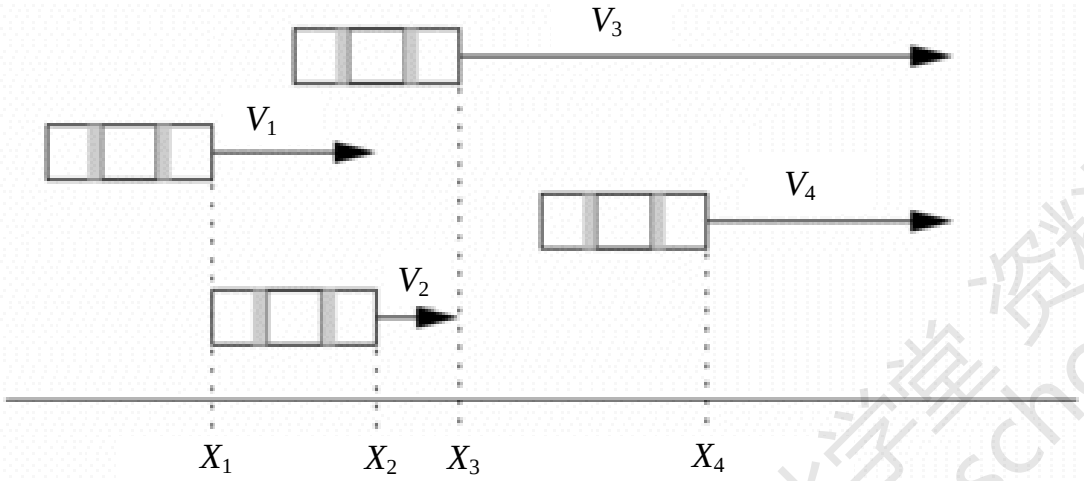
- 素因子都在集合{2, 3, 5, 7}的数称为ugly number
- 求第n大的丑数

分析

- 初始：把1放入优先队列中
- 每次从优先队列中取出一个元素k，把2k, 3k, 5k, 7k放入优先队列中
- 从2开始算，取出的第n个元素就是第n大的丑数
- 每取出一个数，插入4个数，因此任何堆里的元素是O(n)的，时间复杂度为O(nlogn)

例5. 赛车

- 有n辆赛车从各不相同的地方以各种的速度(速度$0<v_i<100$)开始往右行驶，不断有超车现象发生。



- 给出n辆赛车的描述（位置x_i，速度v_i），赛车已按照位置排序（$x_1<x_2<\ldots<x_n$）
- 输出超车总数以及按时间顺序的前m个超车事件

分析

- 事件个数$O(n^2)$, 因此只能一个一个求
- 给定两辆车，超越时刻预先可算出
- 第一次超车可能在哪些辆车之间？
  - 维护所有车的前方相邻车和追上时刻
    - 局部：此时刻不一定是该车下个超车时刻！
    - 全局：所有时刻的最小值就是下次真实超车时刻
- 维护：超车以后有什么变化？
  - 相对顺序变化…改变三个车的前方相邻车
  - 重新算追上时刻，调整三个权
  - 简单的处理方法：删除三个再插入三个

例6. 可怜的奶牛

- 农夫John有n（n≤100 000）头奶牛，可是由于它们产的奶太少，农夫对它们很不满意，决定每天把产奶最少的一头做成牛肉干吃掉。但还是有一点舍不得，John打算如果不止有一头奶牛产奶最少，当天就大发慈悲，放过所有的牛。
- 由于John的奶牛产奶是周期性的，John在一开始就能可以了解所有牛的最终命运，不过他的数学很差，所以请你帮帮忙，算算最后有多少头奶牛可以幸免于难。每头奶牛的产奶周期T_i可能不同，但不会超过10。在每个周期中，奶牛每天产奶量不超过200。

分析

- 如果采用最笨的方法，每次先求出每头牛的产奶量，再求最小值，则每天的复杂度为O(n)，总复杂度为O(Tn)，其中T是模拟的总天数。由于周期不超过10，如果有的牛永远也不会被吃掉，那么我们需要多模拟2520天（1，2，3，…，10的最小公倍数）才能确定
- 周期同为t的奶牛在没有都被吃掉之前，每天的最小产奶量也是以t为周期的。因此如果把周期相同的奶牛合并起来，每天只需要比较10类奶牛中每类牛的 最小产奶量就可以了，每天的复杂度为O(k)，其中k为最长周期

分析

- 假设周期为6的牛有4头，每次只需要比较k组牛的“代表”就可以了，每天模拟的时间复杂度为$O(k)$。

项　目	第6n+1天	第6n+2天	第6n+3天	第6n+4天	第6n+5天	第6n+6天
牛1	2	5	3	5	7	4
牛2	3	1	6	7	5	4
牛3	5	3	3	5	3	9
牛4	4	4	3	8	8	2
合并结果	2（牛1）	1（牛2）	3（多）	5（多）	3（牛3）	2（牛4）

分析

- 只要周期为6的牛都不被吃掉，这个表一直是有效的。但是在吃掉一头奶牛后，我们需要修改这个表，使它仍然记录着每天的最小产奶量
  - 方法一: 重新计算，时间O(h)，其中h是该组的牛数
  - 方法二: 把一个周期中每天的最小产奶量组织成堆，每次删除操作的复杂度是O(klogh)
- 由于每头奶牛最多被吃掉一次，因此用在维护“最小产奶量结构”的总复杂度不超过O(nklogn)。每天复杂度为O(k)，总复杂度为O(Tk+nklogn)

例7. 黑匣子

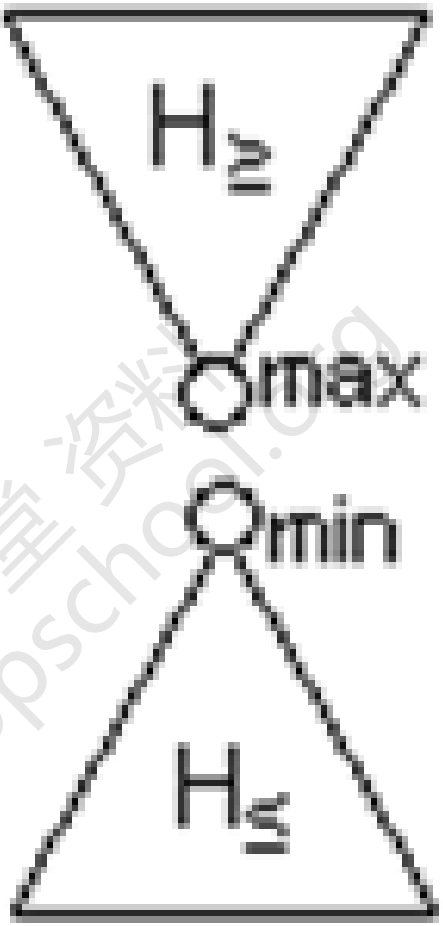
- 我们使用黑匣子的一个简单模型。它能存放一个整数序列和一个特别的变量*i*。在初始时刻，黑匣子为空且*i*等于0。这个黑匣子执行一系列的命令。有两类命令：
- ADD(*x*)：把元素*x*放入黑匣子；
- GET：*i*增1的同时，输出黑匣子内所有整数中第*i*小的数。牢记第*i*小的数是当黑匣子中的元素以非降序排序后位于第*i*位的元素

例7. 黑匣子

编号	命令	i	黑匣子内容	输出
1	ADD(3)	0	3	
2	GET	1	3	3
3	ADD(1)	1	1, 3	
4	GET	2	1, 3	3
5	ADD(-4)	2	-4, 1, 3	
6	ADD(2)	2	-4, 1, 2, 3	
7	ADD(8)	2	-4, 1, 2, 3, 8	
8	ADD(-1000)	2	-1000, -4, 1, 2, 3, 8	
9	GET	3	-1000, -4, 1, 2, 3, 8	1
10	GET	4	-1000, -4, 1, 2, 3, 8	2
11	ADD(2)	4	-1000, -4, 1, 2, 2, 3, 8	

分析

- 降序堆$H_{\geq}$和升序堆$H_{\leq}$如图放置
- $H_{\geq}$根节点的值$H_{\geq}[1]$在堆$H_{\geq}$中最大，$H_{\leq}$根节点的值$H_{\leq}[1]$在堆$H_{\leq}$中最小，并满足
  - $H_{\geq}[1] \leq H_{\leq}[1]$
  - size[$H_{\geq}$]=*i*
- ADD(*x*): 比较*x*与$H_{\geq}[1]$，若$x \geq H_{\geq}[1]$，则将*x*插入$H_{\leq}$，否则从$H_{\geq}$中取出$H_{\geq}[1]$插入$H_{\leq}$，再将*x*插入$H_{\geq}$
- GET: $H_{\leq}[1]$就是待获取的对象。输出$H_{\leq}[1]$，同时从$H_{\leq}$中取出$H_{\leq}[1]$插入$H_{\geq}$，以维护条件(2)



可并优先队列

- 二叉堆很好的实现了优先队列的各种操作
  但是却很难将两个堆合并起来
  - 只能将其中一个堆的元素一个一个取出来插入另一个堆
  - 如果两个堆的元素都有n个，需要花nlogn的时间

可并优先队列

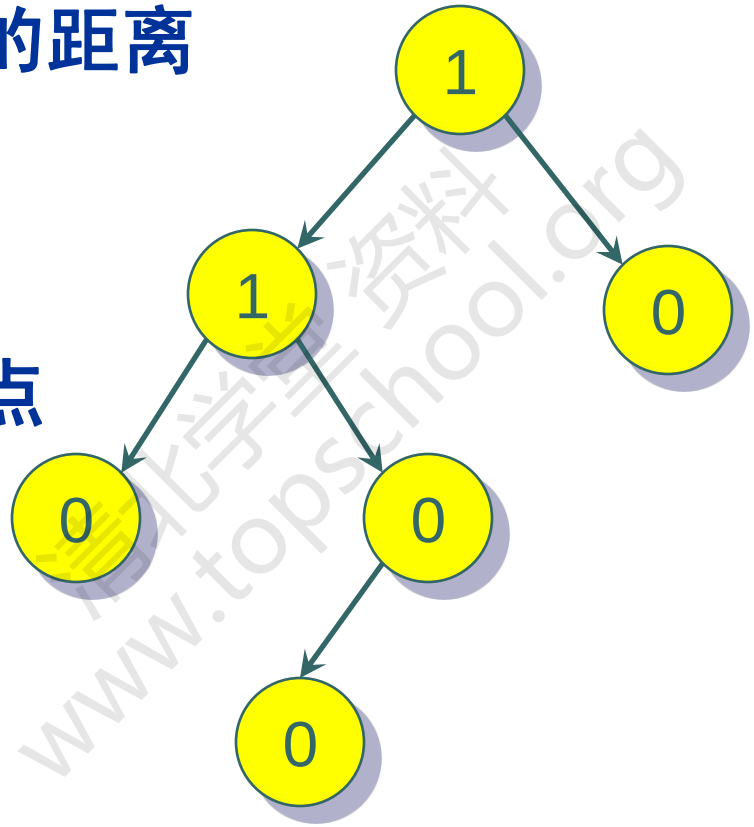
- 常用的可并优先队列有四种
  - 左偏树与斜堆
    - 实现简单, 实用性高
  - 二项堆
    - 实现有一定难度, 是Fibnacci堆的基础
  - Fibonacci堆
    - 理论时间复杂度非常优秀. 但是实现难度相当大

左偏树

- 左偏树的定义:
  - 是一棵二叉树(左子树与右子树都是左偏树)
  - 满足堆性质 ：根的键值小于等于儿子键值
  - 满足左偏性质(Leftist Property)

Null Path Length

- 一个节点的*NPL*(*Null path length*)定义为
  - 到达子孙中最近的外节点的距离
- 外节点
  - 两个儿子不同时存在的节点

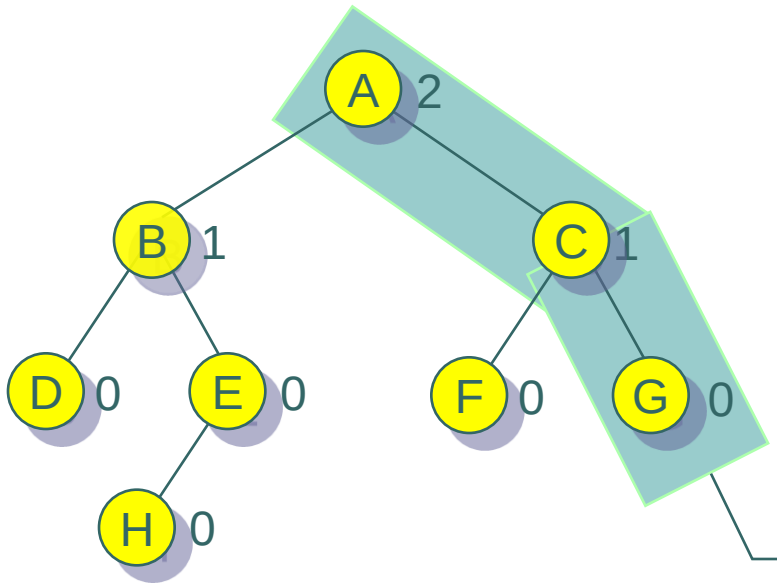


左偏性质

- 记左偏树的根为root
  - root.left.*NPL* ≥ root.right.*NPL*　(左偏性质)
  - 注意这个性质是递归的,因为根的左子树和右子树也是左偏树
- 由左偏性质可知:
  - 一个点的*NPL*即为其最右路径的长度

左偏树的性质

- 定理: 若一棵左偏树有N个节点, 则该左偏树的*NPL*不超过⌊log(N+1)⌋-1.



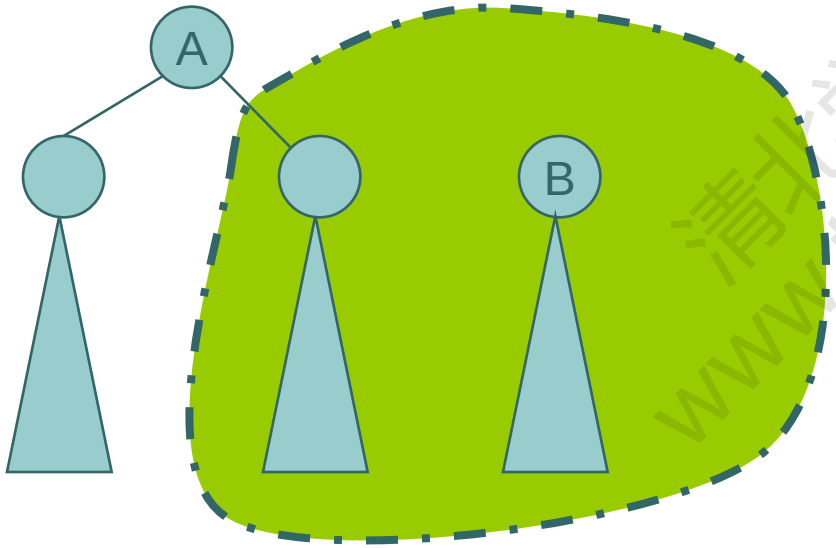
最右路径：A—C—G
最右路径长度 = 2

8个节点的左偏树
⌊log(8+1)⌋-1 = 2

最右路径长度即为该点*NPL*

基本操作：合并

- 若A或B为空，要返回另外一棵树，否则
  - 第一步：假设A的根≤B的根(否则交换A和B),把A的根作为新树的根，合并A.right和B
  - 第二步：如果合并后A.right>A.left(*NPL*)，交换
  - 第三步：更新A.*NPL*=A.right.*NPL* + 1

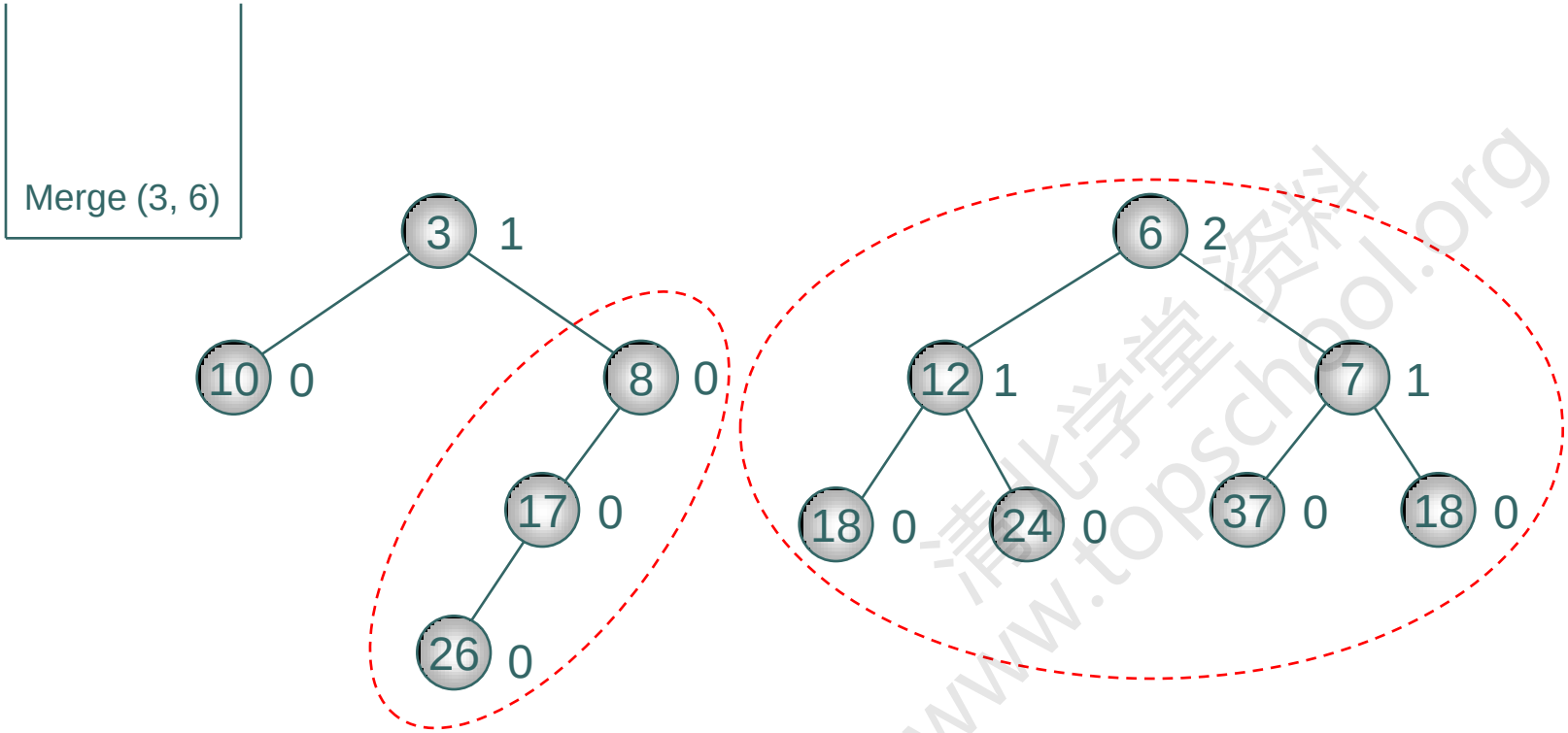


合并伪代码

```
Function Merge(A, B)
        If A = NULL Then return B
        If B = NULL Then return A
        If B.key < A.key Then swap(A, B)
        A.right ← Merge(A.right, B)
        If A.right.NPL > A.left.NPL Then
                swap(A.left, A.right)
        If A.right = NULL Then A.NPL ← 0
        Else A.NPL ← A.right.NPL + 1
        return A
End Function
```

合并示例

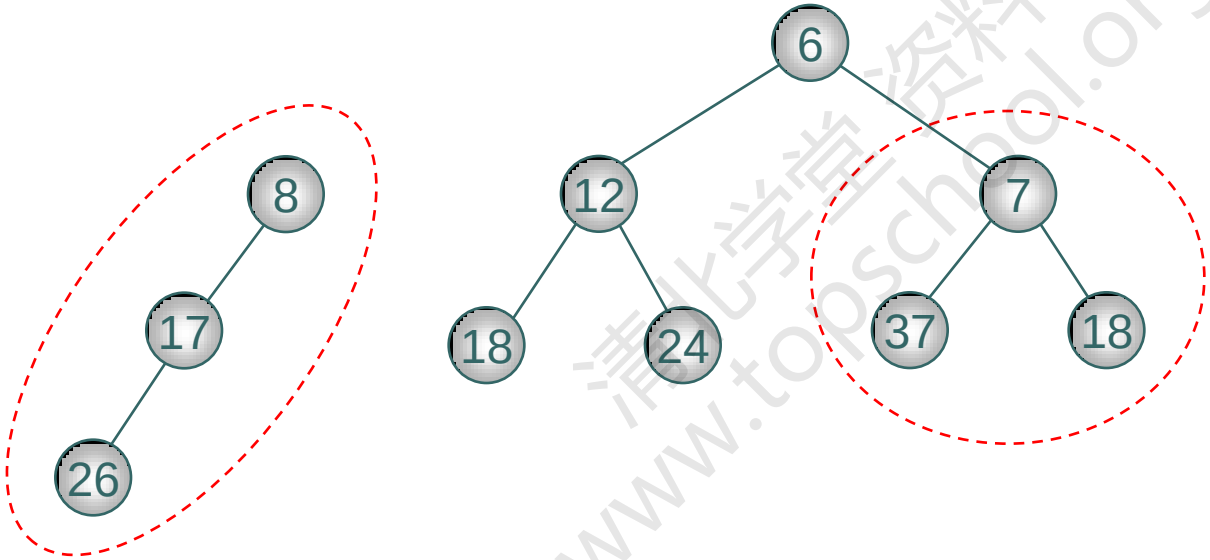
- 下面是一个合并的例子：



合并示例

- 下面是一个合并的例子：

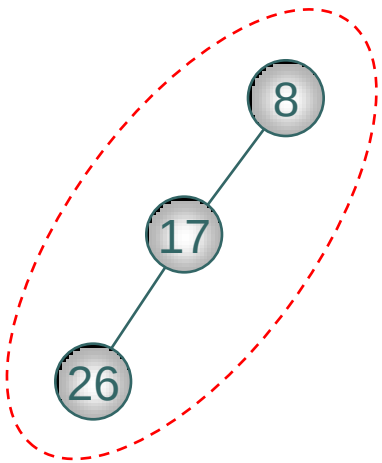
Merge (8, 6)
Merge (3, 6)

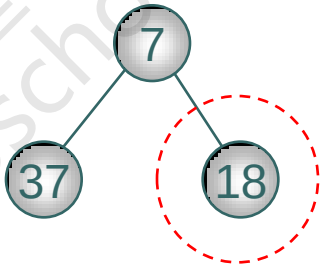


合并示例

- 下面是一个合并的例子：

Merge (8, 7)
Merge (8, 6)
Merge (3, 6)





合并示例

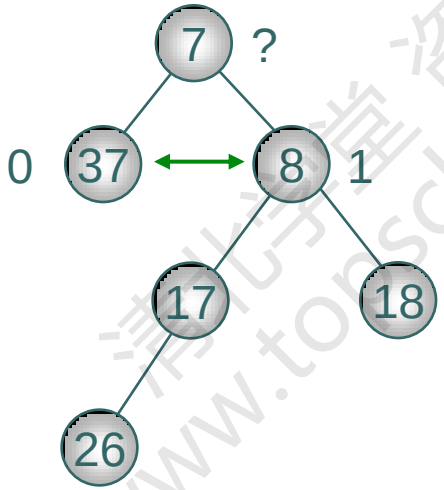
- 下面是一个合并的例子：

Merge (8,18)
Merge (8, 7)
Merge (8, 6)
Merge (3, 6)

合并示例

- 下面是一个合并的例子：

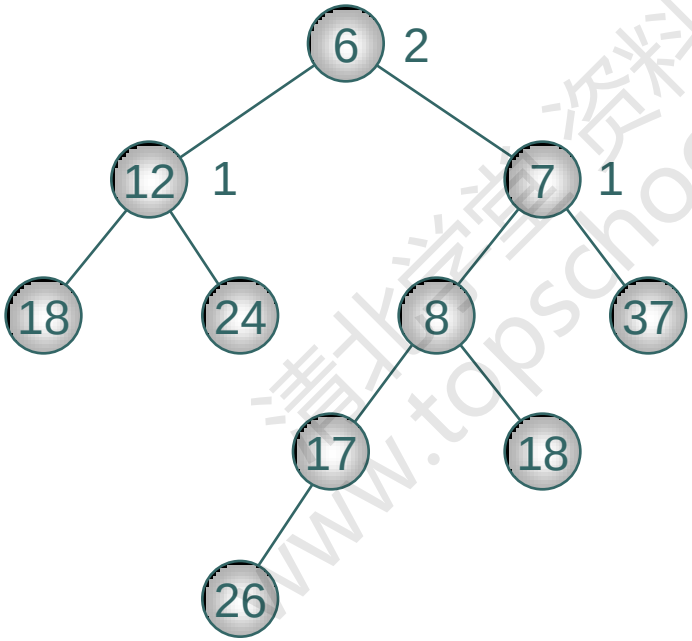
Merge (8, 7)
Merge (8, 6)
Merge (3, 6)



合并示例

- 下面是一个合并的例子：

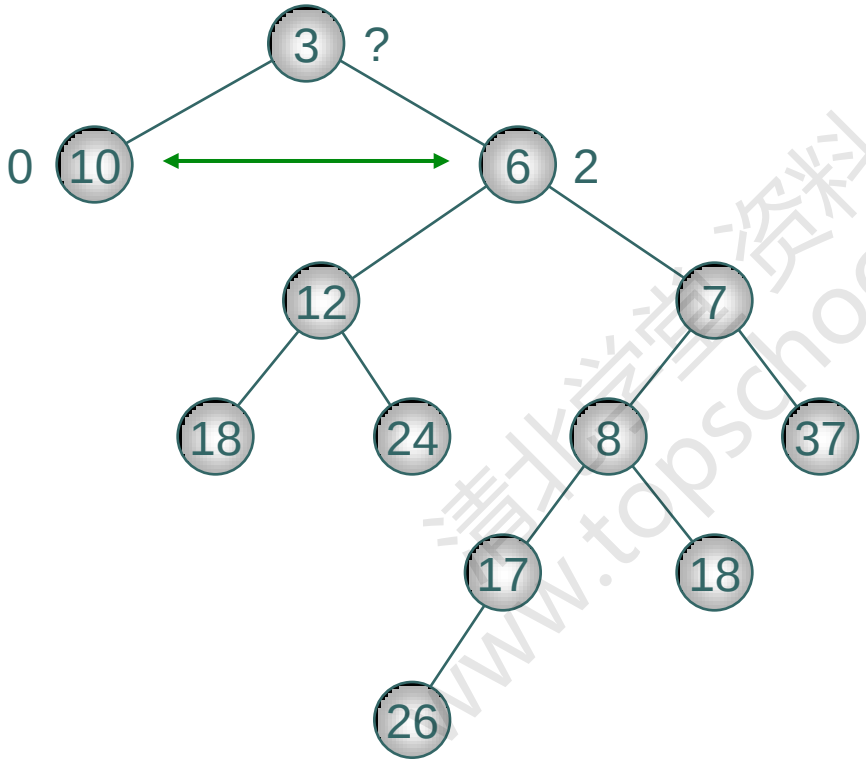
Merge (8, 6)
Merge (3, 6)



合并示例

- 下面是一个合并的例子：

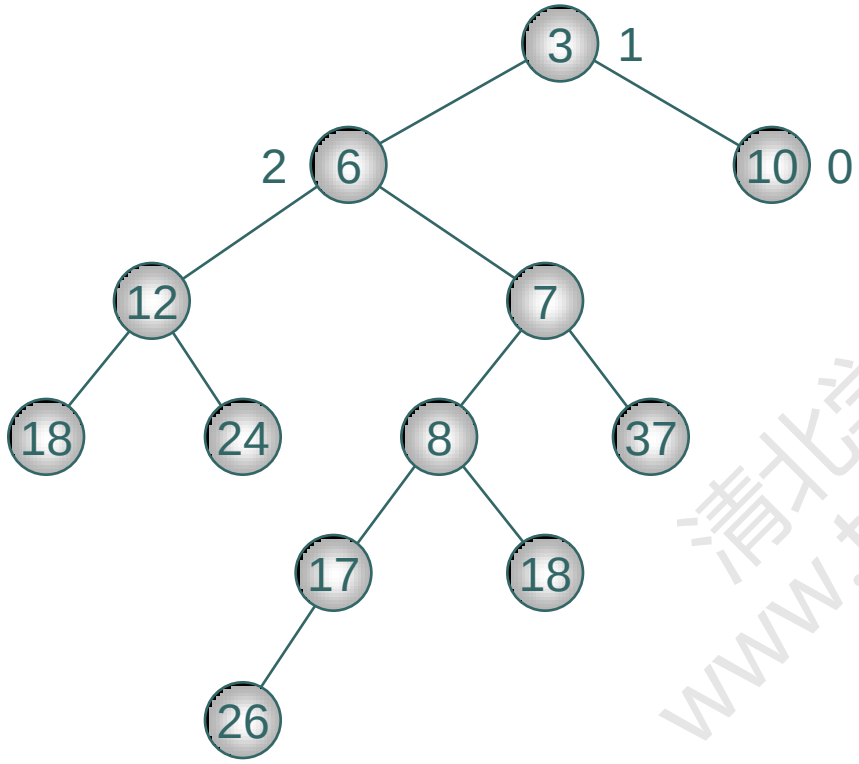
Merge (3, 6)



合并示例

- 下面是一个合并的例子：

Merge (3, 6)



合并操作的分析

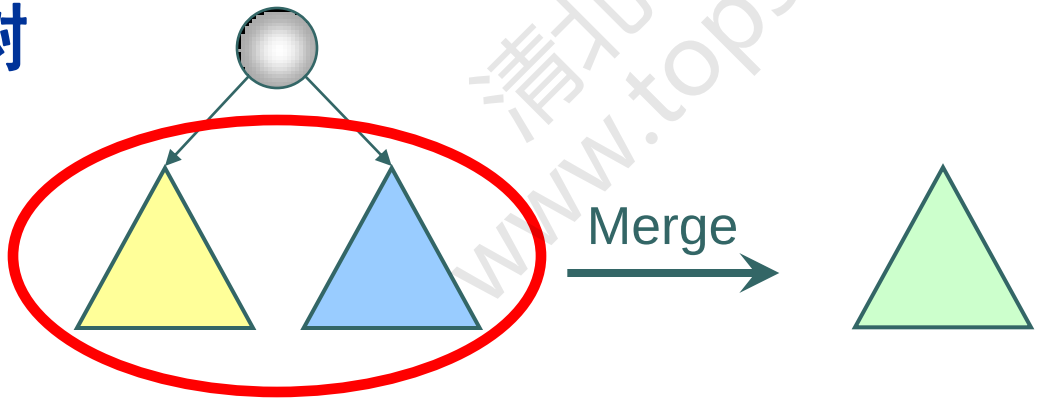
- 每次递归合并时分解右子树
- 它的距离至少减少1

- 故时间复杂度为$O(\log N_1+\log N_2)$

左偏树的其他操作

- 插入新节点
  - 将单独一个点作为一棵左偏树，进行合并操作

- 删除最小点
  - 合并左右子树



左偏树性质定理的证明

- 定理: 若一棵左偏树有N个节点, 则该左偏树的*NPL*不超过⌊log(N+1)⌋-1.

- 要证明这个定理, 我们只需要证明另一个引理即可:
  - 若一棵左偏树的*NPL*(最右路径长度)为r , 则该树包含至少$2^{r+1}-1$个节点

归纳证明

- 如果r = 0 , 左偏树至少包含一个节点
- 否则我们假设上述定理对于r=0..k均成立
- 考虑一棵最右路径长度为k+1的左偏树
- 由定义知根的右子树的*NPL*为k, 而左子树的*NPL*大于等于k.
- 由归纳假设, 左右子树均至少包含$2^{k+1}-1$个节点
- 所以当前这棵左偏树至少包含$(2^{k+1}-1)*2+1 = 2^{k+2}-1$个节点, 得证.

[例]Monkey King(ZOJ2334)

- 丛林中生活着N只好斗的且互不认识的猴子
- 两只不认识的猴子之间免不了要发生冲突,那个时候,两只猴子会分别邀请他们认识的最强壮的猴子来进行一次决斗
- 决斗之后,这两群猴子互相都认识了
- 每一个猴子有一个强壮值, 而两个猴子在参与决斗之后强壮值将减半

[例]Monkey King(ZOJ2334)

- 现在我们得到了M个信息, 表示哪两只猴子发生了冲突
- 对于每一个冲突信息
  - 如果这两只猴子早就认识, 则输出-1(非法)
  - 否则输出决斗后这群猴子(他们互相都认识了)中最强壮的猴子的强壮值

[例]Monkey King分析

- 用并查集维护连通情况(认识与否)
- 用左偏树维护每一个群体(互相认识的猴子)内的强壮值.需要用到的操作即为:
  - 删除最大值
  - 插入新值
  - 合并两棵左偏树
- 时间复杂度: O(MlgN)

THE END

Thank You.

Reference & 感谢

- 部分PPT来自于刘汝佳的讲稿
- 部分图片&动画来自黄源河同学的论文