算法基础 HW3

PB18111697 王章瀚

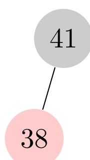
2020 年 11 月 5 日

1

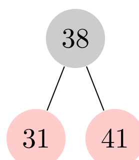
红黑树

(a). 将关键字41, 38, 31, 12, 19, 8 连续地插入一棵初始为空的红黑树之后, 试画出该结果树.

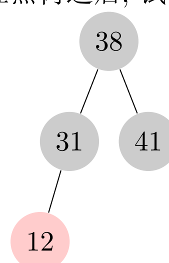
插入38之后:



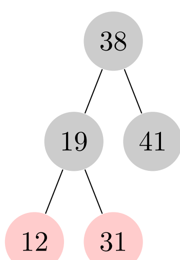
插入31之后:



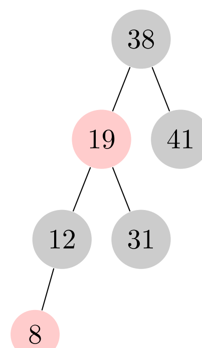
插入12之后:



插入19之后:

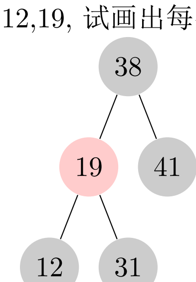


插入8之后(也就是最终结果):



(b). 对于(a)中得到的红黑树, 依次删除8,12,19, 试画出每次删除操作后的红黑树.

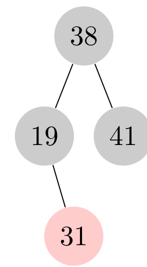
删除8, 因为是红色的, 直接删除就行:

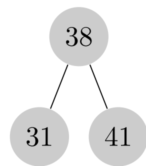


删除12,

(1). 其下面接上来的结点x(NIL)是双黑结点，其兄弟是黑叶子，按case2来fix

(2). 之后x.p是红，所以改为黑，结束



删除19, 接上来红黑结点(31), 变黑即可: 

2

假设我们希望记录一个区间集合的最大重叠点, 即被最多数目区间所覆盖的那个点.

(a). 证明: 在最大重叠点中一定存在一个点是其中一个区间的端点

证明如下:

假设找到了一个最大重叠点x, 但它并不是某区间的端点. 假设它在以下区间中:

$$
\begin{aligned}
&[a_1, b_1] \\
&[a_2, b_2] \\
&\cdots \\
&[a_k, b_k]
\end{aligned}
$$

那么必然有

$$a_{(1)} \leq a_{(2)} \leq \cdots \leq a_{(k)} \leq x \leq b_{(1)} \leq b_{(2)} \leq \cdots \leq b_{(k)}$$

则显然, 包含x的这k个区间也同样包含$a_{(k)}$和$b_{(1)}$. 而这两个数则是区间的端点. 由此就构造出了区间端点使得其位最大重叠点之一. $\square$

(b). 设计一个数据结构, 使得它能够有效地支持Interval-Insert, Interval-Delete, 以及返回最大重叠点的Find-Pom操作

(1). 构造的整体描述:

只需要构造一颗红黑树, 其key为所有2n个区间端点, 但多加两个个域

- lr: (即lefr right)表示其为左端点还是右端点, 若为左端点则$lr = +1$, 若为右端点则$lr = -1$
- $v(x)$: 表示x为根节点的子树中所有结点的 lr 值之和. 它可以这样递归求出:

  $$v(x) = v(x.left) + x.lr + v(x.right)$$

- $m(x)$: 即以x为根节点的子树的最大重叠点的相关信息, 用 m.val 表示最大重叠点, 用 m.count 表示最大重叠数.

  m.count 的表达式为:

  $$m(x).count = \max_i S(leftest(x), i)$$

  这里的i是x的所有后代， $leftest(x)$ 表示 x 所有后代中最小的,
  而$S(i, j)$则是在考虑范围内, 按顺序从 i 到 j 结点的 lr 值之和.

  m.count 的递归计算方法如下:

  $$m(x).count = \max \begin{cases} m(x.left).count & \text{即这个最大重叠点在x的左子树} \\ v(x.left) + p(x) & \text{即这个最大重叠点就是x} \\ v(x.left) + p(x) + m(x.right).count & \text{即这个最大重叠点在x的右子树} \end{cases}$$

  至于 m.val 则依据 m.count 的计算方法取出即可: 当上述求m.count的递归式确定了这个最大重叠点在左子树还是右子树还是就是x的同时, 也就确定了这个最大重叠点(如果在左(右)子树, 就是左(右)子树的最大重叠点, 否则就是x); 至于叶子结点, 就记m.val为它本身, 记m.count=1

(2). 对Interval-Insert, Interval-Delete, Find-Pom操作的支持

- Interval-Insert

  其基础还是红黑树的插入, 特别之处在于, 寻找插入点的时候, 其路径上的所有结点上的v值应当加上这个新插入结点的lr值; 此外其m值和v值也应该相应改变(根据上述递归公式). 而旋转操作也需要有特殊的变换以维持 lr, m 和 v 的值

- Interval-Delete

  同样, 基础也是红黑树的删除, 特别之处在于寻找删除点时, 其路径上的所有结点上的v值应当减去这个要删除结点的lr值, 此外其m值(要和插入的点的)和v值也应该相应改变(根据上述递归公式), 而旋转操作同样也要特殊变换

- Find-Pom:

  直接可以通过 m 的值取出, 是 $O(1)$ 的.

- 左旋转(右旋转类似)

  旋转后显然被旋转结点的子树们的m值和v值不会被改变, 但旋转了的结点需要重新按照上述递归方法计算其m值和v值.

3

(斐波那契堆删除操作的另一种实现) Pisano教授提出了下面的Fib-Heap-Delete 过程的一个变种, 声称如果删除的结点不是由 $H.min$ 指向的结点, 那么该程序运行地更快.

Algorithm 1 Pisano-Delete(H, x)

```
1: if x == H.min then
2:     Fib-Heap-Extract-Min(H)
3: else
4:     y = x.p
5:     if y ≠ NIL then
6:         CUT(H, x, y)
7:         Cascading-Cut(H, y)
8:     add x's child list to the root list of H
9:     remove x from the root list of H
```

(a). 该教授的声称是基于第8行可以在 O(1) 实际时间完成的这一假设, 它的程序可以运行得更快. 该假设有什么问题吗?

- x的孩子数目并非常数, 而是$\log n$的, 因此这一假设并不正确.

(b). 当 x 不是由 H.min 指向时, 给出 Pisano-Delete 实际时间的一个好(紧凑)上界. 你给出的上界应该以 $x.degree$ 和调用 Cascading-Cut 的次数 c 这两个参数来表示.
先不考虑consolidate部分[1](因为题目也没让考虑这个)

- 从**简单的分析**来看, c次调用每次O(1), 加x的孩子到根上总共要x.degree, 所以应该总的上界是O(c+x.degree)
- 从**摊还的角度**来看, Pisano-Delete实际时间的一个紧凑上界应该可以从这几个部分分析:

(1). 一些杂七杂八的操作: O(1)

(2). Cascading-Cut: 这里题目说可以用其次数c来表示. 显然, 除了最后一次调用, 其他 $c-1$次调用都减少了一个mark结点(但最后一次可能又标记了一个), 且每次调用都会产生新的一棵树连在 root list 中. 总之, 它引起的势变化的一个紧凑上界为:

$$\Delta t + \Delta 2m = c - 2 \times (c-2) = 4 - c$$

(3). 将x的孩子加到 root list 中的势增的上界类似上述分析有:

$$(t(H') + 2m(H')) - (t(H) + 2m(H)) = t(H') - t(H) = x.degree$$

综上所述, 当x不是由H.min指向时, 摊还代价的上界可以考虑为:

$$O(1) + O(c) + 4 - c + x.degree$$

于是就可以得出, 其一个好上界可认为是$O(x.degree)$

[1] 如果还要考虑上consolidate, 则还有一部分$O(D(n))$的, 但它则是$O(\log n)$的, 因此总的会变成$O(\log n + x.degree)$