组队作业秩配对堆报告

方博慧，秋闻达，徐值天

2016年5月31日

目录

0 提要

在本次组队作业中，我们学习并实现了秩配对堆（Rank-pairing Heap）。秩配对堆的发明初衷是为了结合斐波那契堆的优秀复杂度和配对堆的简洁实现。以下是几种堆的实现方式的时空复杂度对比（其中n为元素的数量）。

	二叉堆	二项堆	斐波那契堆	秩配对堆	严格斐波那契堆
Space	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Delete-min	$\Theta(\log n)$	$\Theta(\log n)$	均摊$O(\log n)$	均摊$O(\log n)$	$O(\log n)$
Insert	$\Theta(\log n)$	均摊$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	均摊$\Theta(1)$	均摊$\Theta(1)$	$\Theta(1)$
Merge	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

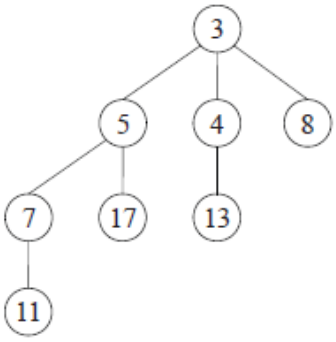
可以看到，如果不考虑均摊与严格的区别，秩配对堆是复杂度最优秀的堆之一。相比起复杂度相同的斐波那契堆，秩配对堆拥有更加简洁的实现，以及更小的运行常数。

因为基于比较的排序时间复杂度不可能优于$O(n \log n)$，所以秩配对堆的时间复杂度也是堆的极限。

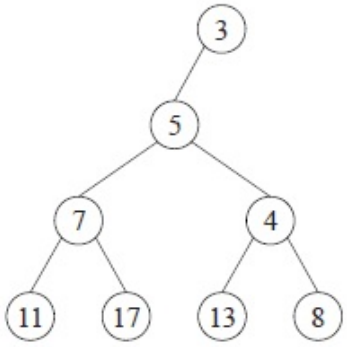
1 结构

1.1 Half Tree

- 堆结构：一棵树，每个节点的key是他所在的子树的最小值
- Half Tree结构：二叉树，每个结点的左孩子是他在堆结构中的第一个孩子，右孩子是他在堆结构中的兄弟。很明显，根节点只有左孩子。



(a) 堆结构

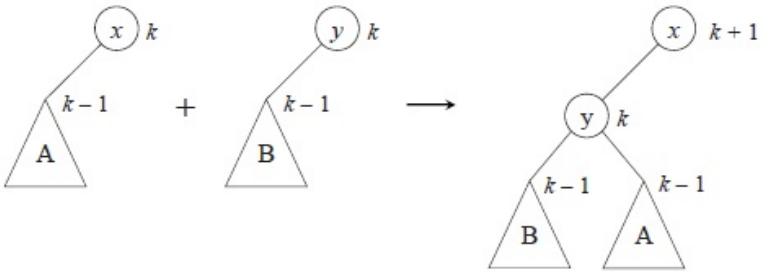


(b) Half Tree结构

在目前很多堆的实现方式中（比如斐波纳契堆），都是用了Half Tree结构，秩配对堆也是如此。秩配对堆的主要结构是许多Half Tree组成的森林。

1.2 Rank

同样地，在很多堆的实现中，我们给每个点计算了$rank$（秩）以一定的方式代表了这棵树的大小，通过合并的方式来限制森林中树的个数，从而保证了复杂度。在秩配对堆中我们定义：空的$rank$为-1，单节点的$rank$为0，$rank$均为的k的两颗Half Tree可以通过Link操作合成一颗$rank$为$k+1$的Half Tree。Link操作如下图所示，需要比较一次两棵树的树根。比较中较小的结点会成为新的树根，也称作获胜者，而另一个结点称作失败者。



(c) Link操作

秩配对堆类似配对堆，对树中rank的形态要求不是非常严格。

先给出两个定义：对于非根结点，x的rank difference为$rank(x.parent)-rank(x)$。x是$i,j-node$表示x的两个孩子（顺序无关）的rank difference是i和j。对于根节点（无右孩子），特别有$i-node$。下面记$r(x)$为$rank(x)$。

当树的形态改变（主要发生在Decrease-key）时，有两种不同的限制方式：

- Type-1：

  所有根节点为$1-node$

  所有非根节点都为$1, 1-node$或$0, i-node$，其中$i>0$，每个点的i可能不一样

- Type-2：

  所有根节点为$1-node$

  所有非根节点都为$1, 1-node$，$1, 2-node$或$0, i-node$，其中$i>1$，每个点的i可能不一样

这两种类型中，Type-1有更优秀的常数，但是复杂度证明更加麻烦；Type-2在常数上劣于Type-1，但是证明复杂度证明较为容易。在本次大作业中，我们选择了常数更优的Type-1。

2 操作

我们使用了懒惰合并的方法来实现均摊的复杂度。

2.1 Find-min

在所有操作中维护min_root，查询时直接返回。这显然是$O(1)$的。

2.2 Merge

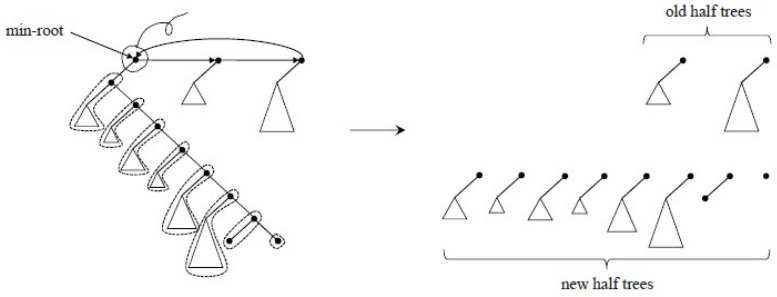
将一个堆的森林链表链接在另一个堆的森林链表后，并维护min_root。这显然是$O(1)$的。

2.3 Insert

将一个只有一个点的堆与自己合并。

2.4 Delete-min

将min_root所在的树的根删除，删除过后剩下的部分不满足Half Tree的性质。按照下图给出的方法，将余下的部分拆成多棵Half Tree，并加入到森林列表中。



(d) Delete-min操作

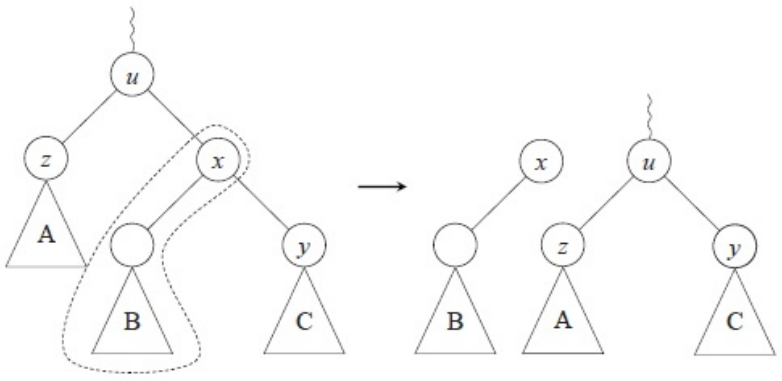
应用懒惰合并的方法，在森林中找到相同$rank$数量最多的一组k，两两合并生成若干棵$rank$为$k+1$的Half Tree，加回到森林中。

找到新的min_root。

2.5 Decrease-key

当需要Decrease-key的节点x已经是某棵Half Tree的树根时，直接修改它的key。

当结点x不是树根时，按照下图给出的方法分离出一颗根为x的新Half Tree，然后修改它的key。分离之后需要进行rank reduce，也就是对从x原来的位置到树根的路径，按照之前给出的$rank$限制(Type-1或者Type-2)，更新新的$rank$。



(e) 分离操作

操作结束后维护min_root。

实现了Decrease-key之后，我们可以用以下的方法在均摊$O(\log n)$的时间复杂度删除任意结点：先对x进行Decrease-key，将x的key改成$-\infty$，再进行delete-min操作。

3 正确性分析

可以看到，在所有操作中都满足了Half Tree的性质，也就保证了堆的正确性。

4 空间复杂度

由于每个节点只记录了他的key和$rank$，以及指针$*Left$，$*Right$，$*Parent$，复杂度是$O(n)$。

而在Delete-min中使用的临时空间大小为max_rank，考虑到均摊过程中的最坏情况，复杂度是$O(n)$。

综上两点，显然，秩配对堆的空间复杂度是$O(n)$。

5 时间复杂度

5.1 Delete-min

我们把节点分为两类，good节点和bad节点。一个节点为good节点当且仅当它是一个rank为0的根(即没有左儿子)；或者它的左儿子是一个1,1-node；或者它是一个rank为0的1,1-node(没有儿子)；或者它的儿子是1,1-nodes；或者它是一个0,1-node满足它的0-child是一个1,1-node。所有其他节点都是bad节点。在这种定义下，一次link操作的获胜者为good root，失败者为一个1,1-node，失败者good当且仅当连在一起的两个根都good，bad当且仅当至少一个连在一起的根为bad。

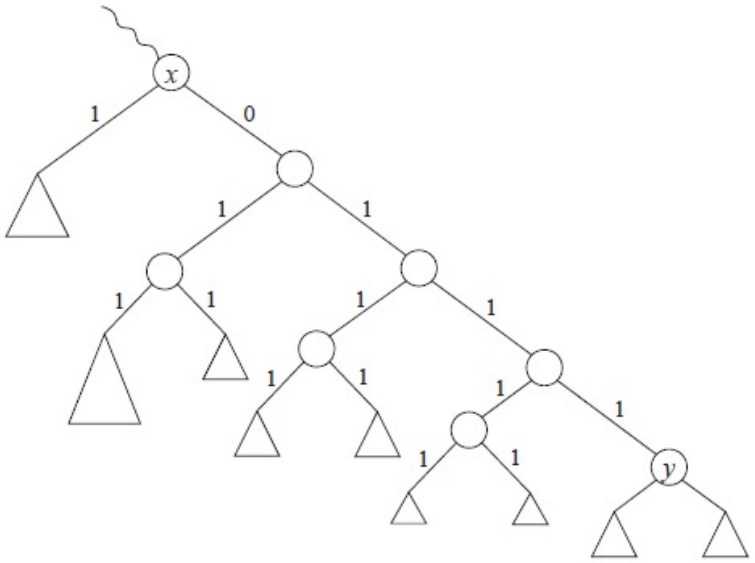
我们定义一个堆的势能函数为它所有节点的势能之和。我们定义一个点的势能函数为它所有儿子的“rank difference”的和，如果是一个根就加二，如果是good节点就减一，如果是bad节点就加三。这个定义不仅适用于那些遵守“rank rule”的节点，也适用于在一次“key decrease”的过程中违背了这个“rank rule”的节点。

定理：在type-1的rp-heap中，以下时间均摊均为$O(1)$:make-heap, find-min, insert, merge, decrease-key，只有delete-min是$O(\log n)$的。

证明：make-heap、find-min、merge操作均为最坏$O(1)$且不改变势能；一次insert操作需要$O(1)$时间且让势能上升了2，因为新的根是good节点。因此所有这些操作是均摊$O(1)$的。

考虑delete-min。在分裂以及连接新产生的half tree的时候，我们给每个新的half tree一个4的“临时势能”（无论是good还是bad节点）代替它本身的势能函数（如果是good则为2，bad则为6）。我们声称如果我们这样做，分裂引发的势能的上升最多是4lgn + 4。只有新根可能有势能上升。每个bad节点有至少4的势能，所以如果它变成了一个根，它有它需要的4的临时势能。考虑一个good节点x变成了根。有两种情况。如果x是一个1,1-node，或者x是一个0,1-node且右儿子是1-child，那么我们给新根x的r(x)充入小于等于4的它所需要的临时势能。如果x是一个0,1-node（右儿子是一个0-child），那么我们给r(y)充入4能量，使x成为新根，y是在x的右链上的最高的bad节点。如下图。如果没有这样的y，那么我们给min-delete充入x所需要的4能量。如果y存在，那么从x到y的路径上的每个节点是一个good的1,1-node，除了x和y。因为y是bad节点，r(y)只会被充能一次为了一个0,1-node，并且他不会被一个good 的1,1-node或者一个0,1-node(右儿子是1-child)充能。

如果y不存在，那么每个x的右链上的节点(除去x)都是一个good 的1,1-node，因此x是唯一一个能给min-delete产生充能的节点。这验证了我们的声称。



(f) x到y的路径

现在我们考虑连接新的half tree。每次这样的连接都会把一个根转换成一个1,1-node并且让剩下的根变为good。在link操作以前，这些节点有8的势能(4 + 4)。我们给在link后保留下来的根的势能为它自身的正确势能2。在link后，两个节点的势能和最大为7（2+5），所以连接操作使得势能至少减少1。

在所有的新的half tree的link结束以后，每个rank最多存在一个新的half tree。我们给每个那样的half tree它自己的正确势能:如果是good就是2，如果是bad就是6。对于小于最大rank 的每个rank，这将最多提升2个势能，所以室总和是$2\log n$（这个上升的上界也正是我们限制link 的原因）。然后我们做剩下的link。每个那样的link使得势能减小1:如果至少有一个被link的根是bad 的，那么在link之前两个根有至少$6 + 2 = 8$的势能，在link之后最多有2（胜利者）+ 5（失败者）= 7的势能；如果两个节点都是good的，在link前他们有$2 + 2 = 4$的势能，之后只有$2 + 1 = 3$的势能，因为失败者是一个good的1,1-node。

总结一下，总minimum deletion操作造成的净势能上升最多只有$6\log n+$

4，再减去link操作的数量。令h为删除之后的所有half tree的数量。所以总的删除操作需要h-1的比较次数，即 $O(h+1)$的时间。把它所放到$h+O(1)$。在所有link操作结束之后，最多有$\log n+1$的half trees，所以会至少进行$h-\log n-1$的link操作。所以minimum deletion会增加最多$7\log m-h+1$的势能，所以给出了均摊$O(\log n)$的时间以及最多$7\log n+5$的均摊比较次数。

5.2 Decrease-key

在节点x进行key decrease的分析，我们需要证明key decrease仅会使$O(1)$的nodes变为bad。一个good 1,1-node不会变为bad，它只可能变成一个good 0,1-node。一个0,1-node不可能减少它的rank，所以如果它变为bad它就是最后一个进行rank reduce的节点。如果节点x成为了一个根，那么它变为bad当且仅当它之前是一个带有right 0-child的good 0,1-node，在这种情况下没有rank会发生改变且x是唯一一个变bad的节点。考虑包含节点x的old half tree的根变为bad的情况，它的左儿子必须是一个1,1-node并且old root是唯一一个变bad的节点。综上可以得出key decrease仅能使一个节点变为bad。如果我们给一个变bad的节点4个单位的势能(在rank-reduce之前)，那么该节点每次rank的减小必会伴随至少一点势能的减小。key decrease也可以创造一个新的根，从而使势能增加2。因此key decrease的均摊操作复杂度为$O(1)$。

在一个Rank-pairing Heap里key decrease的最坏复杂度为$O(n)$，类似Fibonacci heaps。我们可以将这个复杂度减小至$O(1)$(通过延迟每一个key decrease的操作直到一个minimum deletion操作的出现)。这需要维护可能有最小*key*的节点的集合，也就是所有满足自从上一次minimum deletion以来他们的key减小了的roots和nodes。

6 一些实现细节

- 我们实现了指向元素的*iterator*，并在每次push之后返回该元素的*iterator*。这样做能保证*decrease_key*均摊$O(1)$的复杂度能最好地得以利用。
- 我们没有实现任意元素的删除，因为我们使用的*key*是用户传进来的类模板。我们并不知道这个类的$-\infty$。保留*decrease_key*给用户可以让使用更加自由。

- 类似的的原因，我们在实现$decrease_key$时，传进来的是修改后的新key，而不是减少的Δkey，因为我们不知道key的类型是否支持减法。但是相应地，我们需要用户自己遵守key值减少的约定。

7 应用

7.1 排序

- 场景：将n个数排序
- 操作：新建一个堆，插入n个元素，把它们pop出来，得到的元素顺序即为排序后的结果。
- 复杂度：$O(n \log n)$。

基于比较的排序算法不可能低于$O(n \log n)$，因此该复杂度已经达到下界。

7.2 最短路径

- 场景：在一张n个点，m条边的图中，寻找s到t的最短路径。使用堆优化Dijkstra算法。
- 操作：在堆中维护n个点到源点的距离。在做Dijkstra算法的时候，如果某个点i的距离被更新了，只需要调用Decrease-key(i)即可。
- 复杂度：$O(m + n \log n)$。

秩配对堆没有广泛应用于Dijkstra算法的原因：秩配对堆的复杂度和常数都非常优秀，在同等复杂度的堆中写法也很简洁，但是存在另外一些复杂度不是最优、而写起来很容易的替代做法：

- 每次在用某个点i的距离更新其他点的时候，对堆内元素不进行修改，而是将新的距离push进去，等到需要取出堆顶元素的时候，再不停pop到堆顶元素的距离与堆顶的最新距离相同即可。
- 在堆内存储边，然后重载比较函数，对于一条a到b距离为v的边，用$dist[a]+v$作为比较值。

然而以上这两种做法的复杂度均是$O(m + n \log m)$，不如使用Decrease-key(i)的算法优秀，但是面对目前的问题，其表现也已足够令人满意。

8 改进

秩配对堆的作者提出了两个问题：

- 我们能不能像二项堆一样，寻找一种常数更小的，一棵树的实现方法来替代森林？
- Decreas-key的实现以及证明能不能更加简洁？

对于这两个问题，作者的答案都是"No"和"Maybe"。由于时间有限，我们没有得到答案。但这确实是秩配对堆可能改进的方向。

参考文献

[1] Haeupler B, Sen S, Tarjan R E. Rank-pairing heaps[J]. SIAM Journal on Computing, 2011, 40(6): 1463-1485.

[2] Wikipedia：Heap `https://en.wikipedia.org/wiki/Heap_(data_structure)`