大家请对照课程 PPT 的第 11 页和 12 页阅读下面材料。

1 势能分析方法回顾

首先回顾下均摊分析的势能方法：

D_i：执行第 i 步操作后的数据结构；

D_0：初始数据结构；

势能函数：$\Phi : D_i \to R$，反应操作后数据结构的势能；

c_i: 将 $D_i - 1$ 变换到 D_i 的实际成本；

$\hat{c}_i$: 将 $D_i - 1$ 变换到 D_i 的均摊成本，我们有：$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$；所以，总均摊成本为：$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$

在这个总均摊成本的计算中，如果我们能够保证 $\Phi(D_n) - \Phi(D_0) \geq 0$，我们就能保证总均摊成本是总实际成本的一个上限。势能方法的关键是找到合适的势能函数。一般一个好的势能函数设计总能使 $\Phi(D_0)$ 是这个序列中的最小值。实际使用中，我们一般定义 $\Phi(D_0) = 0$。这样，只要证明上式中 $\Phi(D_i) \geq 0$，就可以证明均摊成本是实际成本的一个上限。

2 斜堆合并的均摊分析

斜堆合并的总步骤数就是两个堆的右路径节点数之和；但是斜堆右路径长度不受限，最坏情况可到 $O(n)$，所以我们需要通过均摊分析来确定 M 个操作序列的复杂度。

均摊分析的势能函数要反映斜堆合并操作的效果。考虑到合并成本为右路径总节点数，直接想法是通过右路径节点数目来定义势能函数（斜堆合并由空堆开始，因此该函数刚好从 0 开始且非负）。但该函数的问题是单调递增的，不能反映合并过程中的势能变化。

我们选择斜堆中的“重节点（heavy nodes）”数作为势能函数。如果一个节点的右子树总结点数占该节点为根的子树总结点数达一半及以上为重节点，反之为轻节点。注意到斜堆合并是由空堆开始，该势能函数满足：$\Phi(D_0) = 0$；随着合并的进展，中间任何步骤都有 $\Phi(D_0) \geq 0$。

假设要合并的两斜堆 H_1 和 H_2 右路径上的重节点数分别为 h_1 和 h_2。俩斜堆中其他重节点数目为 $\mathbf{h}$。则有：

$$\Phi(D_0) = h_1 + h_2 + \mathbf{h}$$

注意在合并过程中，除了右路径节点，其他节点不会发生轻重转变，因为它们的左右子树都没有变化。

因为合并成本为右路径总节点数，如果 H_1 和 H_2 右路径上的轻节点数分别为 l_1 和 l_2，则实际合并的总代价为：

$$\sum_{i=1}^{n} c_i = l_1 + l_2 + h_1 + h_2$$

合并后，只有右路径节点会出现轻重节点的改变：（1）右路径重节点合并后会肯定变成轻节点（左右调换后，左轻右重自然变成了左重右轻）；（2）右路径轻节点合并后有可能变成重节点。所以合并后重节点数目最多的情况就是所有右路径的轻节点都变重节点的情况。所以：$\Phi(D_N) \leq l_1 + l_2 + \mathbf{h}$。所以：$(\Phi(D_N) - \Phi(D_0)) \leq l_1 + l_2 - h_1 - h_2$。所以：

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + (\Phi(D_N) - \Phi(D_0)) \leq 2(l_1 + l_2)$$

l_1 和 l_2 为要合并的俩斜堆右路径上轻节点数量。轻节点意味着节点的子树“左重右轻”（和前面讲过的左倾堆类似），因此 $l_1 + l_2 \leq \log N_1 + \log N_2$（$N_1$ 和 N_2 分别为俩斜堆的节点数）。

THEOREM 11.1.

The amortized running times of Insert, DeleteMin, and Merge are $O(1)$, $O(\log N)$, *and* $O(\log N)$, *respectively, for binomial queues.*

PROOF:

The potential function is the number of trees. The initial potential is 0, and the potential is always nonnegative, so the amortized time is an upper bound on the actual time. The analysis for *Insert* follows from the argument above. For *Merge*, assume the two trees have N_1 and N_2 nodes with T_1 and T_2 trees, respectively. Let $N = N_1 + N_2$. The actual time to perform the merge is $O(\log(N_1) + \log(N_2)) = O(\log N)$. After the merge, there can be at most $\log N$ trees, so the potential can increase by at most $O(\log N)$. This gives an amortized bound of $O(\log N)$. The *DeleteMin* bound follows in a similar manner.

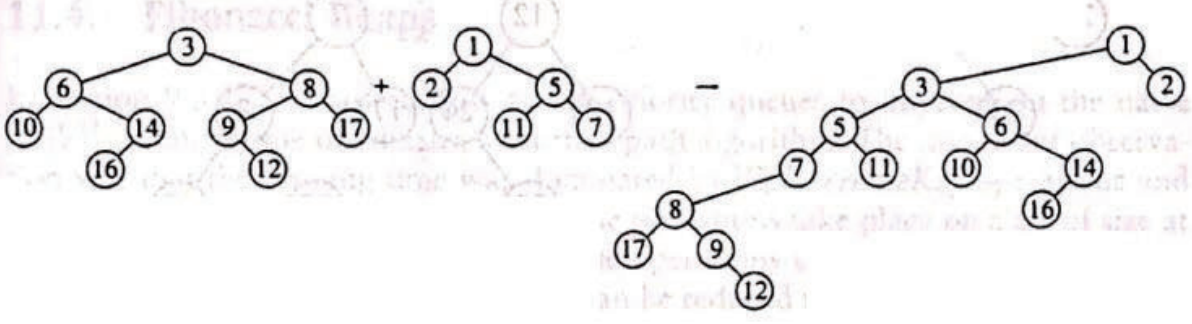
11.3. Skew Heaps

The analysis of binomial queues is a fairly easy example of an amortized analysis. We now look at skew heaps. As is common with many of our examples, once the right potential function is found, the analysis is easy. The difficult part is choosing a meaningful potential function.

Recall that for skew heaps, the key operation is merging. To merge two skew heaps, we merge their right paths and make this the new left path. For each node on the new path, except the last, the old left subtree is attached as the right subtree. The last node on the new left path is known to not have a right subtree, so it is silly to give it one. The bound does not depend on this exception, and if the routine is coded recursively, this is what will happen naturally. Figure 11.6 shows the result of merging two skew heaps.

Suppose we have two heaps, H_1 and H_2, and there are r_1 and r_2 nodes on their respective right paths. Then the actual time to perform the merge is proportional to $r_1 + r_2$, so we will drop the Big-Oh notation and charge one unit of time for each node on the paths. Since the heaps have no structure, it is possible that all the nodes in both heaps lie on the right path, and this would give a $\Theta(N)$ worst-case bound to merge the heaps (Exercise 11.3 asks you to construct an example). We will show that the amortized time to merge two skew heaps is $O(\log N)$.

Figure 11.6 Merging of two skew heaps



What is needed is some sort of a potential function that captures the effect of skew heap operations. Recall that the effect of a *Merge* is that every node on the right path is moved to the left path, and its old left child becomes the new right child. One idea might be to classify each node as a right node or left node, depending on whether or not it is a right child, and use the number of right nodes as a potential function. Although the potential is initially 0 and always nonnegative, the problem is that the potential does not decrease after a merge and thus does not adequately reflect the savings in the data structure. The result is that this potential function cannot be used to prove the desired bound.

A similar idea is to classify nodes as either heavy or light, depending on whether or not the right subtree of any node has more nodes than the left subtree.

DEFINITION: A node p is *heavy* if the number of descendants of p's right subtree is at least half of the number of descendants of p, and *light* otherwise. Note that the number of descendants of a node includes the node itself.

As an example, Figure 11.7 shows a skew heap. The nodes with keys 15, 3, 6, 12, and 7 are heavy, and all other nodes are light.

The potential function we will use is the number of heavy nodes in the (collection) of heaps. This seems like a good choice, because a long right path will contain an inordinate number of heavy nodes. Because nodes on this path have their children swapped, these nodes will be converted to light nodes as a result of the merge.

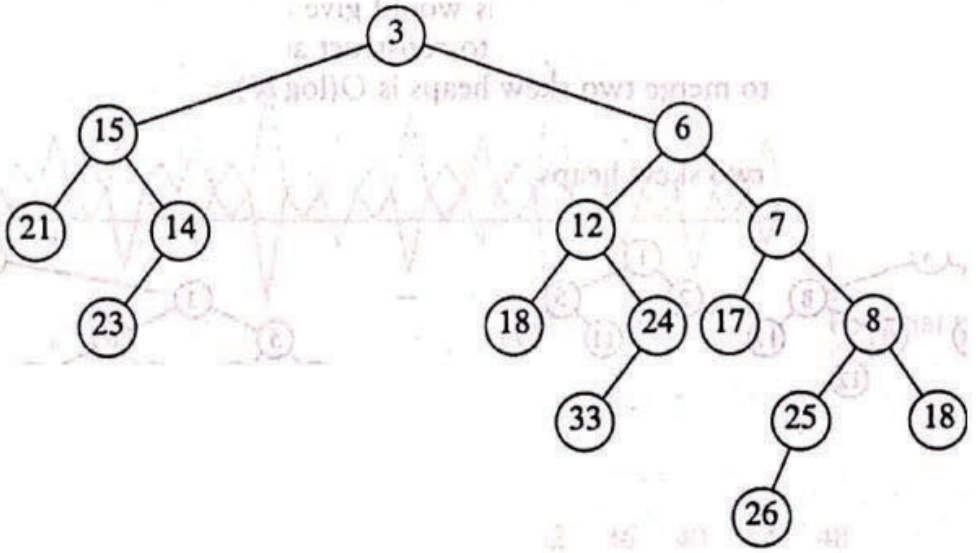
THEOREM 11.2.

The amortized time to merge two skew heaps is $O(\log N)$.

PROOF:

Let H_1 and H_2 be the two heaps, with N_1 and N_2 nodes respectively. Suppose the right path of H_1 has l_1 light nodes and h_1 heavy nodes, for a total of $l_1 + h_1$.

Figure 11.7 Skew heap—heavy nodes are 3, 6, 7, 12, and 15



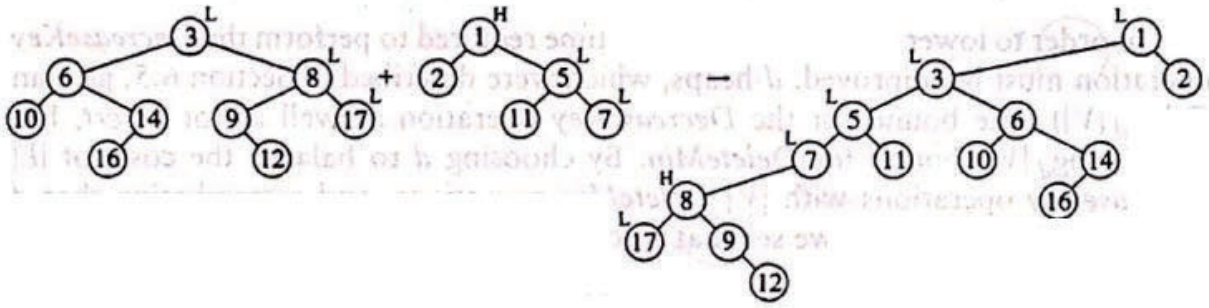


Figure 11.8 Change in heavy/light status after a merge

Likewise, H_2 has l_2 light and h_2 heavy nodes on its right path, for a total of $l_2 + h_2$ nodes.

If we adopt the convention that the cost of merging two skew heaps is the total number of nodes on their right paths, then the actual time to perform the merge is $l_1 + l_2 + h_1 + h_2$. Now the only nodes whose heavy/light status can change are nodes that are initially on the right path (and wind up on the left path), since no other nodes have their subtrees altered. This is shown by the example in Figure 11.8.

If a heavy node is initially on the right path, then after the merge it must become a light node. The other nodes that were on the right path were light and may or may not become heavy, but since we are proving an upper bound, we will have to assume the worst, which is that they become heavy and increase the potential. Then the net change in the number of heavy nodes is at most $l_1 + l_2 - h_1 - h_2$. Adding the actual time and the potential change (Equation (11.2)) gives an amortized bound of $2(l_1 + l_2)$.

Now we must show that $l_1 + l_2 = O(\log N)$. Since l_1 and l_2 are the number of light nodes on the original right paths, and the right subtree of a light node is less than half the size of the tree rooted at the light node, it follows directly that the number of light nodes on the right path is at most $\log N_1 + \log N_2$, which is $O(\log N)$.

The proof is completed by noting that the initial potential is 0 and that the potential is always nonnegative. It is important to verify this, since otherwise the amortized time does not bound the actual time and is meaningless.

Since the *Insert* and *DeleteMin* operations are basically just *Merge*s, they also have $O(\log N)$ amortized bounds.

11.4. Fibonacci Heaps

In Section 9.3.2, we showed how to use priority queues to improve on the näive $O(|V|^2)$ running time of Dijkstra's shortest-path algorithm. The important observation was that the running time was dominated by $|E|$ *DecreaseKey* operations and $|V|$ *Insert* and *DeleteMin* operations. These operations take place on a set of size at most $|V|$. By using a binary heap, all these operations take $O(\log |V|)$ time, so the resulting bound for Dijkstra's algorithm can be reduced to $O(|E| \log |V|)$.