CS2045M: 高级数据结构与算法分析 **2025-2026 学年秋冬学期**

Lecture 4: 左式堆与斜堆

编写人：吴一航 yhwu_is@zju.edu.cn

4.1 左式堆

4.1.1 可合并堆的引入

在数据结构基础课程中，我们已经学过最简单的堆（优先队列）结构，该结构的目标在于能够快速访问全局最优的值（优先值），从而有利于一些需要不断选取最优值的场景，例如操作系统中的任务调度，或者在下一讲将会介绍的 Dijkstra 算法加速（以及最小生成树的 Prim 算法）等。作为一个数据结构，堆自然有如下两种性质需要规定：

1. 结构性质：堆是一棵完全二叉树。并且由于是完全二叉树，故树高 h 和总结点数之间的关系显然为 $h = O(\log n)$。
2. 序性质：对于最大堆，根结点就是最大元素，然后每个结点的孩子必须小于等于自身；反之对于最小堆，根结点是最小元素，孩子应当大于等于父亲。

之前的平衡搜索树也是由这两者规定的，序性质大部分都是左小右大，B+ 树略复杂，结构性质则是各显神通，读者可以自行回顾总结。

此前已经学习了基于数组的二叉堆实现，事实上是利用数组构造了一棵完全二叉树，通过数组索引除以 2 找到结点的父亲，乘以 2 和乘以 2 加 1 获得结点的左右孩子，因此非常方便，相比于指针需要寻址，这样的实现显然更加高效。下面简要回顾一下二叉堆的一些基本操作（以最小堆为例，本节无特殊说明都默认最小堆）：

1. Insert：直接插在完全二叉树的下一个空位上，然后 percolate up 找到它应当在的位置，显然最坏情况也与完全二叉树的高度成正比，即 $O(\log n)$。
2. FindMin：直接返回根结点即可，时间 $O(1)$。
3. DeleteMin：直接用完全二叉树的最后一个元素顶替根结点，然后 percolate down 找到新根结点的归宿，时间 $O(\log n)$。
4. BuildHeap：即对 n 个元素建堆存储。这是一个比较特别的操作，最原始的方法就是连续插入 n 次，但这样时间复杂度为 $O(n \log n)$，所以要有更好的手段。更好的方法是：无需管序性质，直接

任意插入这 n 个值，然后从完全二叉树倒数第二排有孩子的结点开始，往前依次检查是否有违反序性质的，有就 percolate down 到正确的位置，循环直到根结点也调整完毕为止，可以验证这样的算法复杂度为 $O(n)$，具体可见数据结构基础的教材与 PPT。

除此之外还有一些操作，这些操作在 Dijkstra 算法加速等场景中可能有应用，因此也展开介绍：

1. DecreaseKey / IncreaseKey：非常简单，直接用 percolate up / down 实现即可。
2. Delete：用 DecreaseKey 把 key 降低到最低，percolate up 到根结点后调用 DeleteMin 即可。

需要注意的是，这里的操作要求维护需要进行操作的结点（记为 P）的位置，否则需要 $O(n)$ 的时间找到 P 然后再操作（因为堆不支持平衡搜索树那样左小右大的搜索便利性），显然是不合适的。所以这时候用指针实现是一种可行的办法，这样即使结点在一棵虚拟的树上的位置变了，但它在内存里的位置是不变的，这时以上操作的时间复杂度均为 $O(\log n)$。

看上去上面的操作需要的时间都十分完美，那么还需要在什么方面有改进呢？答案是有时候会考虑两个堆合并（merge）的问题。如果要对二叉堆进行合并，由于仍然需要保证合并后是完全二叉树，所以在基于数组的实现中能想到的最好的方法也只能是直接合并两个数组，然后调用 BuildHeap 在 $O(n)$ 时间内完成。然而我们希望合并也是一个 $O(\log n)$ 内可以实现的操作，所以需要定义新的结构（也就是接下来要介绍的三种可合并堆）来实现这一点。

4.1.2 左式堆的定义与性质

下面首先介绍左式堆（leftist heap），这一数据结构仍然保持堆的序性质，但不再要求完全二叉树的结构性质。和它的名字一样，左式堆应当是整体向左倾斜的，那么如何严格定义这样的直观呢？这样的直观又有什么好处呢？下面来逐一介绍。首先需要定义一个 null path length 的概念以便于之后进一步定义左式堆的结构：

定义 4.1 *把任一结点 X 的零路径长（null path length, NPL）Npl(X) 定义为从 X 到一个没有两个儿子的结点的最短路径的长。因此，具有 0 个或 1 个儿子的结点的 Npl 为 0，且规定 Npl(null) = -1。*

根据这一定义可知，对于一个堆（或者二叉树），计算每个结点的 Npl 应当从叶子结点出发向上计算，因为**每个结点的 Npl 就等于它的两个孩子的 Npl 的最小值 + 1**。这一结论适用于没有两个孩子的结点，因为定义中 Npl(null) = -1。

定义 4.2 *左式堆的结构性质是：每个结点的左孩子的 Npl 都要大于等于其右孩子的 Npl。注意这一定义适用于没有两个孩子的结点，因为有定义 Npl(null) = -1。*

左式堆的例子和非例子可以参考 PPT。总结而言，我们基于 Npl 定义出了左式堆的结构性质，将向左倾斜的直观有了严格的表达。那么下一步需要研究的问题就是，向左倾斜有什么好处？

定理 4.3 *在右路径上有 r 个结点的左式堆必然至少有 $2^r - 1$ 个结点（右路径指从根结点出发一路找右孩子直到找到叶子的路径）。*

讨论 1：证明这一定理（提示：使用数学归纳法）。

证明: 使用数学归纳法证明。若 $r = 1$，则显然至少存在 1 个结点。设定理对右路径上有小于等于 r 个结点的情况都成立，现在考虑在右路径上有 $r+1$ 个结点的左式堆。此时，根的右子树恰好在右路径上有 r 个结点，因此右子树大小至少为 $2^r - 1$。考虑左子树，根据左式堆定义左子树的 Npl 必须大于等于 $r - 1$，事实上 Npl 大于等于 $r - 1$ 的树右路径至少有 r 个结点，因此左子树大小也至少为 $2^r - 1$，因此整棵树的结点数至少为 $1 + (2^r - 1) + (2^r - 1) = 2^{r+1} - 1$。 ■

从这个定理立刻得到，共有 N 个结点的左式堆的右路径最多含有 $\lfloor \log(N+1) \rfloor$ 个结点。因此我们的想法是，将左式堆的操作的所有的工作都放到右路径上进行，因为它是 $O(\log n)$ 的。当然很显然的一点是，对右路径的 Insert 和 Merge 可能会破坏左式堆性质，但接下来的分析将表明，恢复左式堆的性质是非常容易的。

有一个直觉是值得特别说明的，就是平衡搜索树中通常要求树越平衡越好，但堆却似乎不需要这一点，这是为什么呢。这是因为堆不支持 find 操作，所以左式堆左边的结点在操作中可以完全不被访问，而接下来的讨论会表明只在右路径上操作也完全可以解决插入、删除最小值和合并，因此完全不需要保持树的平衡。

4.1.3 左式堆的操作与实现

接下来的讨论主要解决三个核心操作：Insert、DeleteMin 和 Merge，其它操作和二叉堆区别不大或者没有必要。PPT 的 5-7 页详细展示了操作的动态过程以及代码实现，可以参考，特别是代码实现很可能作为考试中的程序填空题出现，请务必多加注意。

接下来简要地说明三种操作：

1. Merge：最核心的操作，事实上是接下来两种操作的基础。
2. Insert：可以视为一个堆和一个单结点的堆的 Merge，因此问题转化为 Merge。
3. DeleteMin：两个步骤实现：首先删除根结点，然后只需要将根结点的两个子树 Merge 即可，因此关键问题还是 Merge。

所以这三种操作最终都归结于 Merge，时间复杂度也完全来源于 Merge，所以接下来只对 Merge 进行详细的分析。下面介绍两个版本的解决方案，其一是递归，其二是迭代，事实上它们从结果上来看是等价的，我们分别来看它们在分析中的优劣。

【注：Delete 和 DecreaseKey 在另一份附件中有介绍，感兴趣的同学可以参考。】

4.1.3.1 递归实现

递归实现分为如下步骤：

1. 如果两个堆中至少有一个是空的，那么直接返回另一个即可；
2. 如果两个堆都非空，比较两个堆的根结点 key 的大小，key 小的是 H_1，key 大的是 H_2；
3. 如果 H_1 只有一个顶点（根据左式堆的定义，只要它没有左孩子就一定是单点），直接把 H_2 放在 H_1 的左子树就完成任务了（很容易验证这样得到的结构符合左式堆性质，此时 Npl 也没有变化）；
4. 如果 H_1 不只有一个顶点，则将 H_1 的右子树和 H_2 合并（这是递归的体现，在 base case 设计良好，其它步骤也都合理的情况下你完全可以相信这一步递归帮你做对了），成为 H_1 的新右子树；
5. 如果 H_1 的 Npl 性质被违反，则交换它的两个子树；
6. 更新 H_1 的 Npl，结束任务。

如上的步骤对应于 PPT 的伪代码，直观的解释在伪代码的前一页也有展现，核心是上面的 4-5 步。需要注意的是，PPT 上的伪代码比较特别，它把整个过程拆成了两个互相调用的递归函数，实际上直接把第二个递归函数合并进第一个函数的代码中就会发现这就是一个普普通通的递归过程，这里只是更进一步地模块化，实际上是和普通递归完全等价的。

还有两个问题值得讨论，其一是一定要注意更新 Npl，否则所有的结点 Npl 都将是初始值 0，因此整个堆无论长什么样都是左式的，这显然不合理（得到的堆序性质仍满足但结构性质不满足）。其二是这一算法的复杂度如何，事实上可以走一遍这个递归流程，例子就是 PPT 第 5 页的例子：

1. 首先比较发现 $3 < 6$，因此递归要求 H_1 的右子树 H_1' 与 H_2 合并，H_1 的左子树不动；
2. H_1' 的根结点 $8 > 6$，因此递归要求 H_2 的右子树 H_2' 与 H_1' 合并，H_2 的左子树不动；
3. H_1' 根结点 8 大于 H_2' 根结点 7，因此递归要求 H_2' 的右子树 H_2'' 与 H_1' 合并，H_2' 的左子树不动；
4. H_1' 根结点 8 小于 H_2'' 根结点 18，因此递归要求 H_1' 的右子树 H_1'' 与 H_2'' 合并，H_1' 的左子树不动；
5. 发现 H_1'' 是 null，因此直接连接上 H_2'' 即可，递归开始返回；
6. 1-4 步的每一步都在实现递归中的 `Merge(H1->Right, H2)` 步骤，因此接下来只需要将 merge 后的树接到父亲的右子树上，实现 `H1->Right = Merge(H1->Right, H2)`，然后判断是否需要交换子树并且更新 Npl 即可继续返回上一层递归，知道所有递归函数都返回就实现了整个过程。

在将递归过程展开之后，就可以很清楚地分析时间复杂度。首先分析递归的最大深度。不难发现，在 1-5 步的递归过程中，产生的递归层数不会超过两个左式堆的右路径长度之和，因为每次递归都会使得

两个堆的其中一个（根结点 key 更小的）向着右路径上下一个右孩子推进，并且直到其中一个推到了 null 结点就不再加深递归。注意加深一层的过程中的操作是常数的，因为只需要简单的大小比较和找孩子，加上右路径长度的限制，因此递归向下的过程是 $O(\log n)$ 的。这一点可以更严谨地展开：假设 H_1 大小为 N_1，H_2 大小为 N_2，两者路径之和

$$O(\log N_1 + \log N_2) = O(\log N_1 N_2) = O(\log \sqrt{N_1 N_2}) = O(\log(N_1 + N_2)),$$

上面的推导用到了基本不等式 $a + b \geqslant 2\sqrt{ab}$。总而言之，两个堆右路径长度之和仍然是两个堆大小的对数级别，因此递归层数是 $O(\log n)$ 的是准确的。

接下来分析递归返回的操作，事实上每一层的操作也是常数的，因为只需要接上新的指针，判断、交换子树以及更新 Npl，所以也是 $O(\log n)$ 的，因此总的时间复杂度就是 $O(\log n)$ 的。

4.1.3.2 迭代实现

有趣的一点是，上面递归过程的展开实际上就等价于迭代算法的流程：每一次递归向下对应迭代中保留根结点更小的堆的左子树（就像是左子树不动，右子树等着接下来合并的结果），直到最后一次与 null 合并直接接上，递归返回过程实际上就是逐个检查新的右路径上的结点是否有违反 Npl 性质的并且更新 Npl 即可，其它结点无需关心是因为它们根本就不受影响（可以看 PPT 第 7 页的例子辅助理解这一过程）。因为已经说明了迭代和递归的每一步都是有对应关系的，只不过递归是最后返回时才接上每个结点的右子树，迭代过程中就已经接好了，因此二者时间复杂度是一样的。

当然在完成上面的流程后会有一个观察，就是在递归向下之后，或者说交换孩子调整左式堆性质之前，合并得到的堆的右路径是原来两个堆的右路径合并排序的结果，例如 PPT 上的 3、8 和 6、7、18 合并为了 3、6、7、8、19。通过上面的过程很容易证明这一结论是通用的，因为每次都在比较两个堆的右路径上两个点的大小，然后把小的作为根插入。有了这一规律，做题会更快捷一些，因为你只需要把两条右路径从小到大排序，然后从小到大依次带着左子树接入到新的右路径即可（但要注意在此之后还需要调整使得满足左式堆结构性质）。因此用代码实现迭代版本也并没有想象中复杂，只需要对两个堆右路径从小到大便利操作，然后再从右路径最后一个点返回根结点，过程中检查结构性质并更新 Npl 即可。

4.2 斜堆

4.2.1 简介

斜堆与左式堆的关系就像是 splay 树和 AVL 树之间的关系。回顾 splay 树，它并不需要维护 AVL 树中的 bf 属性，只需要在访问一个结点之后就无脑地将它用 zig / zig-zig / zig-zag 三种情况将它翻到根结点即可。

斜堆也是类似的想法，它不用再维护 Npl，因此在递归过程中左式堆所有维护结构性质以及更新 Npl 的

操作不再需要，取而代之的是如下操作：

1. 在 base case 是处理 H 与 null 连接的情况时，左式堆直接返回 H 即可，但斜堆必须看 H 的右路径，要求 H **右路径上除了最大结点之外都必须交换其左右孩子**。
2. 在非 base case 时，若 H_1 的根结点小于 H_2，如果是左式堆，需要合并 H_1 的右子树和 H_2 作为 H_1 的新右子树，最后再判断这样是否违反性质决定是否交换左右孩子，斜堆直接无脑交换，也就是说每次这种情况都把 H_1 的左孩子换到右孩子的位置，然后把新合并的插入在 H_1 的左子树上。

可以看 PPT 第 8 页的例子，如果像前面分析左式堆那样展开递归的每一步，前面的过程很好理解，就是无脑交换根的 key 更小的堆的左右孩子，关键在于当递归到最深的一层是 merge 一个 null 堆和一个 18 为根、35 为 18 的左孩子的堆，看上面操作的第一条，这个堆的右路径上除了最大结点外都要交换左右孩子，但幸运的是，这个堆右路径只有 18 一个结点，它是最大的，所以无需交换。这也就是 PPT 上说从递归角度来看这里不用交换的本质原因，**特别注意务必以这上面说的方法为准，在维基百科等地方的斜堆 base case 之后都无需操作，但这里可能还有操作（尽管这个例子没有，但作业题有），作业和考试务必按照这里以及 PPT 所说的操作为准！**

所以类似于左式堆，最后合并出的堆的左路径上包含两个原始堆的右路径排序后的结果，当然后面还可能连着原始堆右路径最大值的一些左孩子（因为这些左孩子是不被交换的），因此做题的时候这一规律也是可以利用的，熟练之后不一定要按照前面递归的方式逐步分析。

讨论 2：判断以下两个说法是否正确，若正确，请给出详细的证明；若不正确，请举出反例：

1. 按顺序将含有键值 $1, 2, \cdots, 2^k - 1$ 的结点从小到大依次插入左式堆，那么结果将形成一棵完全平衡的二叉树。
2. 按顺序将含有键值 $1, 2, \cdots, 2^k - 1$ 的结点从小到大依次插入斜堆，那么结果将形成一棵完全平衡的二叉树。

本题两个结果都是正确的，证明只需要找规律归纳即可，没有什么特殊性。除此之外，斜堆的 Delete 和 DecreaseKey 操作一般不讨论。

4.2.2 摊还分析

根据上面的介绍，我们发现斜堆的好处很多，例如不需要多余空间存储 Npl，也不需要有很多判断、更新操作，但它真的能像我们希望的那样，像 splay 树一样具有 $O(\log n)$ 的摊还操作代价吗？接下来开始分析。

首先需要明确的一点是，insert 和 delete 还是以 merge 为核心，所以一系列 insert、delete、merge 操作分析的关键还是 merge，因此只需分析 merge。事实上，因为斜堆的定义没有限制住右路径的长度，因此无法像左式堆那样直接用右路径长度 bound 住它的时间复杂度。

下面使用最强大的摊还分析工具：势函数法。为了定义这一势函数，需要给出一个辅助的定义：

定义 4.4 *称一个结点 P 是重的（**heavy**），如果它的右子树结点个数至少是 P 的所有后代的一半（后代包括 P 自身）。反之称为轻结点（**light node**）。*

为了证明后面的主要定理，在此先给出一个引理

引理 4.5 *对于右路径上有 l 个轻结点的斜堆，整个斜堆至少有 2^l-1 个结点，这意味着一个 n 个结点的斜堆右路径上的轻结点个数为 $O(\log n)$。*

证明: 这里的证明和前面证明左式堆性质是十分类似的。对于 $l=1$ 显然成立，现在设小于等于 l 都成立，对于 $l+1$ 的情况，找到右路径上第二个轻结点，那么它所在子树大小根据归纳假设至少有 2^l-1 个结点。现在考虑第一个轻结点，根据轻结点定义它的左子树更大，而右路径上第二个轻结点所在的子树在其右子树中，因此它的左子树至少有 2^l-1 个结点。故整个堆至少有 $1+(2^l-1)+(2^l-1)=2^{l+1}-1$ 个结点。 ■

定理 4.6 *若有两个斜堆 H_1 和 H_2，它们分别有 n_1 和 n_2 个结点，则合并 H_1 和 H_2 的摊还时间复杂度为 $O(\log n)$，其中 $n=n_1+n_2$。*

证明: 定义势函数 $\Phi(H_i)$ 等于堆 H_i 的重结点（heavy node）的个数，并令 H_3 为合并后的新堆. 设 $H_i(i=1,2)$ 的右路径上的轻结点数量为 l_i，重结点数量为 h_i，因此真实的合并操作最坏的时间复杂度为 $c_i=l_1+l_2+h_1+h_2$（所有操作都在右路径上完成）。因此根据摊还分析可知摊还时间复杂度为

$$\hat{c_i}=c_i+\Phi(H_3)-(\Phi(H_1)+\Phi(H_2)).$$

事实上，在 merge 前可以记

$$\Phi(H_1)+\Phi(H_2)=h_1+h_2+h,$$

其中 h 表示不在右路径上的重结点个数。现在要考察合并后的情况，事实上有两个非常重要的观察：

1. 只有在 H_1 和 H_2 右路径上的结点才可能改变轻重状态，这是很显然的，因为其它结点合并前后子树是完全被复制的，所以不可能改变轻重状态；
2. H_1 和 H_2 右路径上的重结点在合并后一定会变成轻结点，这是因为右路径上结点一定会交换左右子树，并且后续所有结点也都会继续插入在左子树上（这也表明轻结点不一定变为重结点）。

结合以上两点可知，合并后原本不在右路径上的 h 个重结点仍然是重结点，在右路径上的 h_1+h_2 个重结点全部变成轻结点，l_1+l_2 个轻结点不一定都变重，因此合并后有

$$\Phi(H_3)\leqslant l_1+l_2+h,$$

代入数据计算可得

$$\hat{c_i} \leqslant (l_1 + l_2 + h_1 + h_2) + (l_1 + l_2 + h) - (h_1 + h_2 + h) = 2(l_1 + l_2).$$

根据前面的引理，$l_1 + l_2 = O(\log n_1 + \log n_2) = O(\log(n_1 + n_2)) = O(\log n)$（这里的等号之前有完全一样的说明），并且注意到初始（空堆）势函数一定为 0。且之后总是非负的，所以这一势函数定义满足要求，因此证明也就完成了。 ■

回忆摊还分析中强调的，势函数的想法就是使得复杂度大的操作恰好是能较多地降势能的，这里右路径很长的时候操作时间复杂度大，但很幸运的是，根据引理，右路径上的轻结点个数总是不可能太多，因此这时重结点更多，而操作之后右路径上重结点全部变轻，势函数又规定为重结点个数，因此非常完美地抵消了真实的复杂度。

需要说明的是，有一种很直观的势函数定义方法在此是行不通的。我们知道，一次合并的效果是处在右路径上的每一个结点都被移到左路径上，而其原左儿子变成新的右儿子。所以自然的一种想法是把每一个结点算为右结点或左结点来分类，**这是根据结点是右儿子还是不是右儿子来决定的**。将右结点的个数作为势函数，虽然这一势函数初始为 0 并且总是非负，但是问题在于，如果考虑一种最坏的情况，就是右路径长度为 $O(n)$ 时，我们希望势函数减小很多使得摊还代价是对数的，然而很显然能找到例子使得在这种情况下操作之后右结点数量不变，所以这种情况下右结点并不适合于解决问题。因此考虑另一种基于轻重的分类来解决斜堆的摊还分析问题。