21120491: 高级数据结构与算法分析 **2023-2024 学年春夏学期**

Lecture 5: 二项堆

教师：刘金飞，助教：吴一航 日期: 2024 年 3 月 26 日

5.1 二项堆

5.1.1 二项堆的引入与基本概念

二项堆的引入来源于我们希望插入建堆的操作有常数的平均时间，这一想法来源于二叉堆可以在 $O(n)$ 时间内实现 n 个结点的插入建堆操作，而之前讨论的左式堆和斜堆不可以。因此我们希望在保持合并的对数时间的条件下优化插入的时间复杂度。

二项堆的定义如下：

1. 结构性质：
   (a) 二项堆不再是一棵树，而是多棵树构成的森林，其中每一棵树都称为二项树（后面能看到你为什么是这个名字）；
   (b) 一个二项堆中的每棵二项树具有不同的高度（即每一个高度最多对应一棵二项树）；
   (c) 高度为 0 的二项树是一棵单节点树；高度为 k 的二项树 B_k 通过将一棵二项树 B_{k-1} 附接到另一棵二项树 B_{k-1} 的根上而构成，这一点根据 PPT 上的图非常直观。
2. 序性质：每棵二项树都保持堆的序性质，例如是最小堆（本节无特殊说明都默认最小堆）则根结点最小，孩子都比父亲大。

事实上进一步观察我们会发现更有趣的事情：二项树 B_k 实际上就是由一个带有儿子 $B_0, B_1, \cdots, B_{k-1}$ 的根组成。我们很容易根据定义归纳得到高度为 k 的二项树恰好有 2^k 个节点，而且在深度 d 处的节点数恰好就是二项系数 $\binom{k}{d}$（我们下面给出简要证明），因此二项树的名字得来非常自然。

证明: 数学归纳法。对于 $k = 0$ 的情况显然正确，设直到 k 结论都是正确的，则对于 B_{k+1}，第一层和最后一层只有一个结点是可以直接得到的，在其它层中，回忆二项堆的定义是由两个 B_k 接起来的，因此第 i 层中根据归纳假设有 $\binom{k}{i} + \binom{k}{i-1} = \binom{k+1}{i}$ 个结点，命题得证， ■

根据二项堆结构性质（特别注意其中的 b），对于具有 n 个顶点的二项堆我们可以根据其二进制表示推出这个二项堆由哪些大小的二项树组成。

5.1.2 二项堆的操作与分析

最简单的操作是 FindMin，直接遍历全部 $O(\log n)$ 棵树（回忆二进制表示）的根结点即可，因此时间复杂度是 $O(\log n)$。当然也可以通过专门记录最小的根结点来实现 $O(1)$ 的时间复杂度，只是每次 DeleteMin 后要更新这一值。

然后是插入操作，和左式堆一样，插入只是特殊的合并，因此我们分析合并（merge）操作即可。实际上两个二项堆的合并想法也非常简单：既然每个二项堆都和唯一的二进制数对应，那合并后二项堆中二项树的分布情况不就符合二进制数的加法后的结果吗？结合 PPT 第 4 页的动画，merge 操作实际上就是从最小的堆开始（对应二进制最低位），如果无需进位（表明是 1+0 或 0+1）则直接留下称为新堆的一部分，如果需要进位则表明是 1+1，因此做合并后进位与下一位做加法，如此循环直到最高位完成操作。时间复杂度分析非常简单，**在保证堆的存储顺序是按高度从小到大排列的前提下**，时间复杂度很显然就是 $O(\log n)$，因为就是二进制逐位做操作。

插入（insert）是合并的特殊情况，操作不在此赘述，时间复杂度最坏为 $O(\log n)$，但从空开始建堆的时间复杂度是常数的，我们稍后分析。DeleteMin 操作在 PPT 第 6 页也展示得很清楚了，实际上就是先用 $O(\log n)$ 的时间找到根的最小值（或 $O(1)$，但这表明最后还需要更新最小值），设根最小的堆对应 B_k，于是我们可以得到两个堆，其一是整个二项堆移除 B_k 后剩下的堆，其二是 B_k 移除根结点后得到的堆，将这两个堆合并即可，时间复杂度显然是 $O(\log n)$。

对于从空开始插入建堆的时间复杂度我们需要做特殊的分析。因为我们是要平均时间的最坏情况，事实上也就是摊还代价，因此我们回想起三种方法。

聚合法：聚合法需要每一步的操作复杂度，实际上我们随便模拟几步再结合之前讨论的合并和二进制加法之间的关系就可以发现，插入的整个操作与二进制数加 1 有完全的对应关系：若是遇到了某一位是 1+1，则用常数操作完成简单的合并即可，如果遇到 0+1，那么当前所有的二项树合起来就是最后的结果。基于这一观察我们知道，因为 0+1 对应将 0 置 1，1+1 对应 1 置 0，这两种情况都对应于堆的常数时间操作，因此从空树连续插入 n 的顶点的时间复杂度与 $0+1+1\cdots$（n 个 1）的过程中数据二进制表示中 0 和 1 比特翻转的次数总和。

于是算法复杂度就很好计算了，因为我们知道 n 对应于 $\lfloor \log n \rfloor + 1$ 个二进制位，事实上最低位每次加 1 都会反转比特，次低位每两次运算反转比特，倒数第三位每 4 次运算反转比特..... 以此类推，n 次操作的整体时间复杂度与

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{\lfloor \log n \rfloor + 1}}$$

成正比，根据等比数列求和可知上述求和是小于 $2n$ 的（取 $n \to \infty$ 才能到 $2n$），所以单步操作的常数摊还时间也就得到了。

核算法：由于 PPT 上没有给出这一解法，我们在此留一个讨论题：

讨论 1：使用核算法证明二项堆从空堆开始连续插入 n 个结点，每一步操作的摊还代价为 $O(1)$（提示：考虑二进制加法从 0 置 1（置位）和从 1 置 0（复位）两种操作）。

势能法：相信经历了这么多次势能函数的构造，我们对于势能函数构造的方向应该很清楚了，那就是考虑进行复杂度最大的操作时定义怎样的势能函数能使得这一步势能下降很大。这里我们同样直接映射到二进制加法问题，我们发现复杂度很大的操作都对应于很多的复位和一个置位，复位导致 1 变 0，因此我们直接设势能函数等于加 1 后二进制中 1 的个数即可，对应于堆就是堆中插入后二项树的个数。正确性验证非常容易，此处不在赘述，如果还是不会验证那可能说明在摊还分析上还存在较大的困难。

在本讲结束后我们会有堆加速 Dijkstra 算法的实验，在那里我们需要 DecreaseKey 的操作，事实上这在二项堆中是再简单不过的事情，因为每棵树都是二项树，所以类似于普通二叉堆的 DecreaseKey 操作（当然需要维护每个结点的指针，否则还要先找到这一结点，这是很耗时间的事情），我们只需要在违反序性质时将做了 DecreaseKey 的结点与父亲不断交换直到恢复序性质（percolate up）即可，时间复杂度显然也是 $O(\log n)$ 的。那删除任意结点操作显然也是和普通二叉堆一样顺其自然可以支持的，就是将想要删除的结点通过 DecreaseKey 将 key 值降低到最小，然后调用 DeleteMin 即可，显然时间复杂度为 $O(\log n)$。

5.1.3 二项堆的代码实现

因为每个结点的孩子数量可能不只有 2 个，因此我们使用 LeftChild 和 NextSibling 的组合实现。直观上来看用 LeftChild 和 NextSibling 是让二项树翻转了：原先是根的子树从左到右高度依次增大，现在依次减小了。并且为了方便索引每棵二项树，我们用一个数组存储每棵二项树的根，其中数组的索引就对应二项树的高度（见 PPT 第 7 页）。

我们知道插入、DeleteMin 的关键操作都是 merge，因此我们主要讨论 merge，而 merge 中我们经常需要合并两棵大小相同的二项树，因此我们首先介绍 combine 的实现。在 combine（PPT 第 9 页）合并两棵相同大小的二项树时，结合堆的序性质以及 LeftChild 和 NextSibling 的实现，我们很容易想到其实只需要直接用根结点小的根作为新的根，根结点大的整棵树作为 LeftChild，NextSibling 接上根结点小的树除去根结点的其它部分即可。这里的时间复杂度显然是常数的。

接下来进入比较复杂的 merge 阶段（PPT 第 10 页）。前面的内容很显然，我们重点分析循环中的 8 个 case。首先来看 case 是如何决定的，这里我们使用了双!，使用!! 的目的显然就是讲可能不是 0 或 1 的值 bool 化，例如如果 T_1 存在（即第一个堆对应高度为 i 的二项树存在），则 $!!T_1$ 做了两次取非操作后就变成了 1，如果 T_1 不存在，那么 $!!T_1$（T_1 实际上是 NULL）就等于 0，T_2 和进位 carry 也是同理。

然后我们就能理解 $4*!!Carry + 2*!!T_2 + !!T_1$ 的含义了，事实上这就是一个三位二进制数（当然 case 的标号还是十进制的，但我们心里要转化为二进制来分析），最高位表示是否有 carry，即之前的合并是否带来了进位（从堆的角度看也就是之前合并出了一棵新的更高的二项树），第二位代表第二个堆 H_2 是否有高度为 i 的二项树，最后一位代表 H_1 是否有高度为 i 的二项树。因此接下来的匹配过程实际上就是匹配竖式加法的各种情况，因为其中一位来源于 H_1，一位来源于 H_2，一位来源于进位，我们来仔细分析各个情况（我们这里按照 0-7 的顺序，PPT 上先易后难把 3 和 4 顺序对调了，一定要看清楚）：

1. 000：什么都不用做，只需等待循环结束后结束合并；

2. 001：什么都不用做，只需等待循环结束后结束合并；
3. 010：因为我们最后是要返回 H_1（也就是说 H_1 是合并后的结果），清空 H_2，因此此时我们只需要将 H_2 中的树转移到 H_1 然后把 H_2 对应位置改为 NULL，最后等待循环结束后结束合并即可；
4. 011：从加法操作来看此时没有进位，但会产生进位，当前位需要置 0，对应于堆操作就是，H_1 和 H_2 当前位变为 NULL，进位等于两个堆该高度的二项树合并后的结果；
5. 100：类似于 010 的情况，但此时是把 carry 接到 H_1 上，然后只需等待循环结束后结束合并即可；
6. 101：从加法操作看此时会产生进位，当前位需要置 0，因此堆操作就是 H_1 当前位变为 NULL，新的 carry 等于 T_1 和当前的 carry 合并的结果；
7. 110：与 101 类似，只是 H_2 当前位变为 NULL，新的 carry 等于 T_2 和当前的 carry 合并的结果；
8. 111：此时是 1+1 还要加上进位 1，因此求和后有新的进位，当前位也是 1，因此让 H_1 当前位变为 carry，新的 carry 等于 T_1 和 T_2 合并的结果，最后给 H_2 当前位变为 NULL 即可。

如此分析后我们发现，尽管代码看起来很复杂，但如果将以上情况与竖式加法一一对应，实际上是非常容易理解的。有一个细节是 for 循环的终止条件，是由变量 j 控制的，这里需要稍微理解一下，实际上很简单，就是等价地确定最多会循环多少次。最后是 DeleteMin 的实现（PPT 第 11 页），这里与先前讨论的步骤完全对应，所以并没有什么好强调的。

讨论 2：本题希望你设计支持可合并堆操作的 2-3-4 堆。这里的 2-3-4 与我们课上学习的 2-3-4 的 B+ 树是类似的，但也有部分不同：

1. 只有叶子结点存放 key，并且每个叶子结点只存放一个 key，并且叶子结点中的 key 也是无序的，不像 B+ 树中从左到右是从小到大的；
2. 每个非叶结点（包括当根结点不为叶子结点时的根结点）x 存放以 x 为根的子树中叶子结点存储的最小 key；
3. 根结点除了第 2 点外还需存储树的高度。

最后，2-3-4 堆设计放在内存中，磁盘读写是不需要的。你的任务是实现下面的 2-3-4 堆操作，在一个有 n 个元素的 2-3-4 堆上，1-5 每个操作都应当在 $\log n$ 时间内完成，6 的合并应当在 $\log n$ 时间内完成，其中 n 为两个堆元素个数之和。

1. FindMin：返回指向堆中 key 最小的结点的指针；
2. DecreaseKey：将给定结点的 key 减小为给定值；
3. Insert：插入一个给定 key 的叶结点；

4. Delete：删除一个给定 key 的叶结点；

5. DeleteMin：删除最小结点；

6. Merge：合并两个 2-3-4 堆，返回唯一的 2-3-4 堆，销毁原先的 2-3-4 堆。

（本题的可能实现肯定不止一种，不同方案之间可能有少部分细节的差别，合理即可）

5.2 堆加速 Dijkstra 算法

5.2.1 回顾 Dijkstra 算法

Dijkstra 算法是著名的求解单源最短路径的算法，当然有一个重要的条件是没有负边。我们会在动态规划一讲中介绍存在负边的单源最短路径问题的解决思路，以及所有结点对的最短路径问题。

【提醒：这里并不是详解堆加速 Dijkstra 算法，只是叙述一下大致思路，具体实现还需靠各位读者自己努力或者查阅带图示的讲解加深理解】

现在我们需要简单回顾 Dijkstra 算法的流程，并从中找到我们需要用堆加速的理由。实际上 Dijkstra 算法的资料非常丰富，无论是上学期的 PPT 还是教材，还是网络上寻找都可以了解，例如维基百科，这里只把大致思想简单描述，不熟悉的读者请务必阅读其它地方更详细的版本。从算法的第一步开始，Dijkstra 算法维护两个集合，X 和 $V-X$（假设 V 是所有顶点的集合），X 中存的就是 Dijkstra 算法已经访问过的结点，也就是已经确定了最短路径的结点。同时算法维护每个顶点的一个 Dijkstra 得分，也就是当前状态下源结点到该顶点的最短路径长度，暂未到达的长度为 $+\infty$。算法的每一步都要从 $V-X$ 中选择一个 Dijkstra 得分最低的点，也就是从当前状态下算法认为的 $V-X$ 中与源结点距离最近的顶点，将其加入 X 中。就这样逐步加入，直到 $V=X$ 结束。

5.2.2 堆加速 Dijkstra 算法

读者应当不难发现，在算法的执行过程中，每一轮循环我们都需要找到集合 $V-X$ 中 Dijkstra 得分最低的顶点（也就是距离最近的顶点），将其取出，于是我们就会很自然地想到最小堆的 DeleteMin 操作。因此我们可以维护一个存有 $V-X$ 中所有顶点的最小堆，每个顶点的 key 就是当前的 Dijkstra 得分值。每一轮迭代我们可能需要对其中的顶点进行 DecreaseKey 操作，因为一个新的顶点加入 X 后可能使得 $V-X$ 中顶点的 Dijkstra 得分降低（也可以直接删除然后插入新的，如果你没有支持 DecreaseKey 操作，例如左式堆中不应当有 DecreaseKey）。然后我们再调用 DeleteMin 取出最小顶点加入 X。

需要注意的是，我们可以回顾上一讲的讲义，DecreaseKey 操作需要我们的维护每个顶点的位置信息（例如用指针），否则我们还需要首先用线性时间找到它再进行下一步操作。

接下来我们讨论 Dijkstra 的算法复杂度。事实上，无论我们用何种数据结构，Dijkstra 算法的复杂度都可以表达为如下形式：

$$O(|E|\cdot T_{\mathrm{dk}} + |V|\cdot T_{\mathrm{em}}),$$

其中 T_{dk} 和 T_{em} 分别表示 DecreaseKey 操作和 DeleteMin 操作的时间复杂度。这一点并不难验证，因为每一条边的加入都可能导致一次 DecreaseKey 操作，而每一个结点都会在被 DeleteMin 操作取出。于是我们可以分别代入以下两种存储顶点的数据结构，得到如下结论：

1. 数组：用数组存储顶点，DecreaseKey 只需要降低数值即可，因此是 $O(1)$ 的，但 DeleteMin 操作是 $O(|V|)$ 的，因此整体复杂度为 $O(|E|+|V|^2) = O(|V|^2)$。
2. 堆：显然两种操作都是 $O(\log|V|)$ 的，因此整体复杂度为 $O((|E|+|V|)\log|V|)$，如果所有结点都可以从源结点到达，这一复杂度为 $O(|E|\log|V|)$。

当图不太稠密的时候，$|E| = O(|V|^2/\log|V|)$，使用堆的方法时间复杂度会更低（很容易验证），这也是堆加速的意义所在，因为在实际大型路线规划等场景中，我们更容易遇到稀疏图的情况，所以使用堆会使得算法性能更好。

5.2.3 斐波那契堆

如果我们希望在任何时候堆加速的 Dijkstra 算法都能在理论上优于数组版本，那么我们就要求 DecreaseKey 操作是常数的。这很难直接实现，但斐波那契度在理论上实现了 DecreaseKey 操作的常数摊还时间复杂度（但实际上常数非常大，因此实际性能完全不如其它操作简单的堆，同学们在 project 中会体会到这一点）。

如果你要完成本次 project 或者对这一内容感兴趣，那么关于斐波那契堆的具体设计与实现请参考《算法导论》等书籍、资料，这里不对现有资料做无趣的摘抄。下表是总结的各种堆的操作时间，注意表中的 $O(1)$ 都代表摊还代价，二叉堆的建堆 $O(1)$ 时间没有体现在表格中，因为回顾建堆操作，事实上这并不是通过普通的插入实现的。斜堆的删除和 DecreaseKey 操作我们不做讨论，因此也不要求大家实现。

	二叉堆	左式堆	斜堆	二项堆	斐波那契堆
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Merge	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
DeleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$		$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(\log n)$		$O(\log n)$	$O(1)$

讨论 3： 回忆最小生成树的 Prim 算法，写出堆优化的 Prim 算法的大致思路（可以以伪代码形式呈现，

或者有详细的文字描述)，关键问题在于堆加速是加速了什么步骤，应当把什么存在堆中，需要对堆做什么操作。