第七周实验报告

沈家成

2017 年 11 月 3 日

1　1215 Bernouli 树

1.1　Bernouli 树

Bernouli 树是由许多个含有 2^n 个结点的树组成的森林，可以看作二进制方式储存的优先级队列。一般储存方式是用孩子兄弟表示法储存每棵树，用数组储存每棵树的根结点。

1.2　性质

根结点是每棵树的最小结点，通过遍历根结点数组，就可以找到森林中最小结点。

每棵树都有 2^n 个结点，而且正好根结点的孩子们有 $2^{n-1}, 2^{n-2}, ..., 2^0$ 个结点，这为删除根结点后的归并提供了方便。

1.3 具体实现

1.3.1 插入元素

插入元素时，一要保证根结点是树中最小的元素，二要保证 Bernouli 树的森林特性。插入一个元素，可以看作原森林与一个只有一个元素的森林归并。

根结点数组的第一位应该储存结点数为 1 的树，如果不存在该树，直接插入即可。如果存在，就要把较小的结点作为根结点，并把该位树指针归零，再比较下一位，直到有空位可以放进去为止。

那么较大的元素放哪里呢？原本试过将其作为根结点的兄弟结点，但是数据量稍微增大，就出现了重复替换兄弟结点，失去了数据。后来根据 Bernouli 树的特性，将较大的结点插入根结点的孩子们中，即根结点的孩子指针指向该结点，该结点的兄弟指针指向根结点的原孩子，就可以保持 Bernouli 树的结构了。

这是一个不断迭代的过程，为了保持操作的一致性，设置了两个指针，一个名叫 new root, 另一个叫做 new child，首先将要插入的结点赋给 new root, 再在循环中根据不同情况更新 new root 和 new child，就可以完成插入操作了。

1.3.2 查找最小值

由于根结点是每棵树的最小元素，遍历根结点数组即可。

1.3.3 删除最小值

删除操作的主要难点在于删除根结点后，会留下一堆孩子，怎么安顿他们就成了问题。

根据之前的铺垫，根结点的孩子们刚好都是 2^n 结点数的树，因此问题就转换成了删除根结点后归并一堆新树。

首先，要设置一个数组存放这个新树。这时，孩子兄弟储存法的优势就体现出来了，通过不断地指向下一个兄弟，就可以完成孩子们的遍历。值得注意的是，遍历完成后，要将兄弟指针置零，表示这些孩子们已经“分家”了，否则就会出现“恶性争夺财产”的错误。

然后，就要进行归并。一个 2^n 的树删除根结点后留下的孩子们结点数刚好是一个 $2^0, 2^1, ..., 2^{n-1}$ 的等比数列，因此可以直接找到对应结点数的根结点数组。再对插入一个元素的方法进行推广，就可以完成余留树的分配了。再次强调的是，为了操作的一致性，还是有必要单独设置两个指针 new root 和 new child，并先把要插入的树赋值给 new root 再进行迭代，否则就迭代时更新的数据就会混乱。

2 1579 LCS

2.1 矩阵的看法

寻找最大公共子序列是一个动态规划的问题，为了便于分析过程，可以用矩阵表达。

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

(1)

矩阵中的元素代表从左上角到这个位置所能生成的最大公共子序列的长度。通过构建一个这样的矩阵，每次生成一个新元素，就只需要利用其上方、左方和左上方的数据。因此，就可以比较容易地找到最大公共子序列的长度。

那么新元素怎么生成呢。先设想理想的情况，如果这个位置刚好两个数列的元素相等，就意味着这段的最大子序列就是之前的最大子序列再加上这个元素。具体到矩阵的运算，就是取左上角的元素加一。

那么如果是更普遍的不相等呢？那就是在该元素之前生成的最长子序列中选择最大的。具体到矩阵，就是选择上方、左方和左上方的最大值。

这样，整个矩阵就生成了。之后只需要找到最大值，就找到了最大子序列长度。

3 1580 LIS

3.1 $O(N^2)$ 的简单办法

寻找最长递增子序列也是一个动态规划的问题。考虑其中一个元素，它的最长递增子序列应该是，在比它小的元素中，已经存在的最长递增子序列再加一。因此，需要设置一个数组来储存从开始到每个元素的最长子序列。

$$\begin{array}{cccccccccc} 17 & 15 & 61 & 17 & 21 & 61 & 100 & 97 & 69 & 7 \\ 1 & 1 & 2 & 2 & 3 & 4 & 5 & 5 & 5 & 1 \end{array} \tag{2}$$

最后，只要找出最大值就寻找到了最长递增子序列的最大值。但是，这种方法的时间复杂度为 $O(N^2)$，需要更低时间复杂度的算法。

3.2 $O(NlogN)$ 维护最长递增子序列

为了降低时间复杂度，需要摒弃上一个算法的顺序遍历，而递增子序列是有顺序的，因此需要维护一个最长递增子序列用于二分查找。

先考虑最简单的情况。如果新加入的元素大于递增子序列的最大值，直接加入即可。

但是如果新元素处于递增子序列之中呢? 首先是肯定不能直接抛弃的，因为可能后面就跟着一个更长的递增子序列。但是如果直接放进去，一是放哪儿呢，二是怎么判定要不要选择它作为新的递增子序列。因此，要思考这个最长递增子序列要怎么维护。

当已经存在一个递增子序列时，如果要有新的子序列来替代当中的一部分，新的子序列必须有更大的贡献，也就是它比替换的子序列更长。

因此，只需要用该元素替换最小的比该元素大的递增子序列元素（可以用二分查找），就可以在保持子序列顺序的情况下，将该元素进行“缓存”。如果有足够多的既小又多还递增的元素进来，最终就会完全替换掉部分递

增序列，这样就实现了递增子序列的维护。

$$
\begin{array}{l}
17\ \ 15\ \ 61\ \ 17\ \ 21\ \ 61\ \ 100\ \ 97\ \ 69\ \ 7 \\
17 \\
15 \\
15\ \ 61 \\
15\ \ 17 \\
15\ \ 17\ \ 21 \\
15\ \ 17\ \ 21\ \ 61 \\
15\ \ 17\ \ 21\ \ 61\ \ 100 \\
15\ \ 17\ \ 21\ \ 61\ \ 97 \\
15\ \ 17\ \ 21\ \ 61\ \ 69 \\
7\ \ 17\ \ 21\ \ 61\ \ 69
\end{array}
\tag{3}
$$

由于二分查找的时间复杂度为$O(logN)$,所以总的时间复杂度为$O(NlogN)$,达到了想要的效果。