浅谈亚 log 数据结构在 OI 中的应用

宁波市镇海中学 钱易

摘 要

数据结构是 OI 中的一类常考问题，本文介绍了压位 trie 与 vEB tree (van Emde Boas tree) 两种数据结构并且展示了其在部分问题中的应用。

引言

在 OI 中，数据结构一直是一个重要考点，而本文主要介绍了用来解决 Dynamic Predecessor Problem 的两种亚 log 数据结构[1]，目前在学术界的研究已经存在了一些比较优秀的算法，如 y-fast trie [2] 可以以 $O(n \log_2 \log_2 V)$ 的时间复杂度，$O(n)$ 的空间复杂度解决该问题，Fusion tree [3] 可以以 $O(n(\log_w n + \log_2 w))$ 的时间复杂度，$O(n)$ 的空间复杂度解决该问题。但是，这两种数据结构的常数因子非常大，且难以实现，往往不会在 OI 使用。这类问题大多数情况下，值域范围往往不会特别大，因此，本文介绍了在 OI 中更加实用的两种数据结构：压位 trie 与 vEB tree (van Emde Boas tree)。

本文第一节介绍了 Dynamic Predecessor Problem 以及常用于解决这个问题的一些 log 数据结构，第二节介绍了压位 trie，第三节介绍了 vEB tree (van Emde Boas tree)，第四节对一些数据结构的运行效率进行了比较，第五节讲解了其部分应用。

本文涉及程序的运行时间，运行时间测试环境为 64 位 Ubuntu Linux 14.04 LTS x64 操作系统，CPU 为 Intel Core i5-6500 CPU @ 3.20GHz，8 GB 内存，GCC/G++ 版本为 4.8.4，并开启 -O2 优化，测试运行多次取平均值。

1 Dynamic Predecessor Problem

写一种数据结构，来维护一些数，其中需要支持以下操作：

1. 插入 x 数（若已有 x 则不进行此操作）；

[1] 亚 log 在本文中指时间复杂度优于 $O(\log_2 V)$，其中 V 是值域范围

[2] 详见 Wikipedia：https://en.wikipedia.org/wiki/Y-fast_trie

[3] 详见 Wikipedia：https://en.wikipedia.org/wiki/Fusion_tree

2. 删除 x 数（若 x 不存在则不进行此操作）；
3. 求 x 的前趋（前趋定义为小于 x，且最大的数，若不存在则输出 -1）；
4. 求 x 的后继（后继定义为大于 x，且最小的数，若不存在则输出 -1）；

假设操作数量为 n，保证 $1 \le n, x \le 10^7$ 且其均为整数。

时间限制：1s

空间限制：32MB

1.1 常见解法

1.1.1 平衡树

这一道题目要求支持插入删除前驱后继操作，所以可以直接使用平衡树来维护。

由于不需要求第 K 大或求一个数排名，因此可以直接使用 STL 中的 set 来实现，然而由于平衡树效率较低，且使用空间较大，难以通过此题。

1.1.2 树状数组

由于本题值域不大，因此可以直接使用树状数组维护每种数字出现次数，然后采取在树状数组上跳跃的方法来求出前驱后继，时间复杂度 $O(\log_2 V)$，且常数较小，然而空间复杂度为 $O(V)$，其中 V 为值域，难以通过此题。尽管可以通过先分为若干小块的方法来压缩空间，但是代码较大，且运行速度较慢，意义不大。

1.1.3 整型压位

如果集合中的数 x 大小不大，满足 $0 \le x < w$，假设 a_i 表示数 i 是否在集合之中，因此可以使用一个整型 $s = \sum_{i=0}^{w-1} a_i \times 2^i$ 来维护集合，对于插入、删除操作，可以直接使用位运算维护。而对于查询 x 的后继，则只需要返回 $ctz(s >> x) + x$，$ctz(v)$ 表示 v 中第一个 1 所在的位，如果查询 x 的前驱，而对于查询 x 的前驱，则只需要返回 $w - 1 - clz(s \,\&\, (2^x - 1))$，$clz(v)$ 表示 v 中从高位到低位前缀 0 的个数，一般在实现之中，可以使用 GCC 内置的 ___builtin_ctzl，___builtin_clzl 函数来实现。[4] 所以如果集合中的数在 $[0, w)$ 中，以上操作均可 $O(1)$ 实现。

[4]如果不允许使用 GCC 的内置函数，我们也可以做到 $O(1)$ 的时间复杂度，详见：hqztrue,《位运算：进阶技巧（上）》，https://zhuanlan.zhihu.com/p/70950198

2 压位 trie

压位 trie，即 w 叉 trie，因为其较小的代码量与较小的常数，笔者一般使用其解决 Dynamic Predecessor Problem。压位 trie 各个操作时间复杂度均为 $O(\log_w V)$，且具有较强的可拓展性。

2.1 压位 trie 的结构

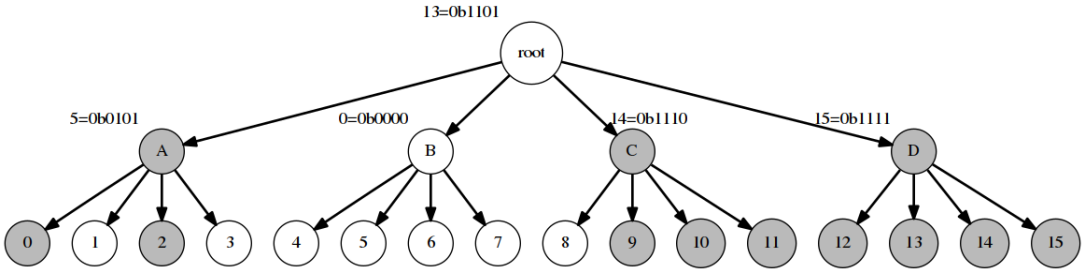
假设 $c = \log_2 w$。

一个大小为 2^k 的压位 trie 可以维护一个数字在 $[0, 2^k)$ 中的集合。

如果 $2^k \le w$，显然可以直接使用一个整型压位来解决，因此我们接下来仅讨论 $2^k > w$ 的部分。

定义 $high(x) = \lfloor \frac{x}{2^{k-c}} \rfloor$，$low(x) = x \bmod 2^{k-c}$，即 x 的高位与低位。

对于一个大小为 2^k 的压位 trie，它将有 w 个大小为 2^{k-c} 的子压位 trie $A_i (0 \le i < w)$，其中第 i 个子压位 trie 存储了高位为 i 的数的低位的集合，并且再使用一个整型 B 按位存储每个子压位 trie 中是否存在元素。



假设 $w = 4$，权值范围是 $[0, 16)$，维护的数字集合是 $\{0, 2, 9, 10, 11, 12, 13, 14, 15\}$，那么建出的压位 trie 如上图，其中节点旁边的数字代表其每个儿子中是否存在元素。

2.2 压位 trie 的插入操作

假设我们需要在一个大小为 2^k 的压位 trie 中插入元素 x。

首先，必然需要将 $low(x)$ 插入到 $A_{high(x)}$ 之中，并且标记一下 $A_{high(x)}$ 中存在元素，可以发现的是，由于其每次会进入到一个大小为原来 $\frac{1}{w}$ 的子压位 trie 之中，因此其时间复杂度为 $O(\log_2 V / \log_2 w)$，即 $O(\log_w V)$。

2.3 压位 trie 的删除操作

假设我们需要在一个大小为 2^k 的压位 trie 中删除元素 x。

首先，需要在 $A_{high(x)}$ 之中删除 $low(x)$，接着，可以通过 $A_{high(x)}$ 中是否存在一个非空子树来判断出其元素是否被删完了，如果删完了则标记一下这个子树之中不存在元素。时间

复杂度可以用类似插入的分析方法得到，时间复杂度为 $O(\log_w V)$。

2.4 压位 trie 的查询前驱操作

我们在一个大小为 2^k 的压位 trie 中查询 x 的前驱，首先先尝试在 $A_{high(x)}$ 之中查询 $low(x)$ 的前驱，如果其存在，那么就得到了答案，如果不存在，那么尝试找到其之前一个存在元素的子树 A_j，$0 \le j < high(x)$ 且 j 最大，如果存在这样一个子树，就返回这个子树中最大值，如果不存在，那么其在这个压位 trie 中也不存在前驱。实现时可以用位运算技巧直接得到 j 的值。

对于查询一个子树里的最大值，它最高位肯定是最大的，因此仅需要找到 B 的最高位递归查询即可。

除了找最大值的部分，就是访问一条根到叶子的链，每个节点处花费的时间均为 $O(1)$，求最大值也是访问一条到叶子的链，因此时间复杂度为 $O(\log_w V)$。

2.5 压位 trie 的查询后继操作

可以发现，压位 trie 的结构是对称的，因此可以直接使用类似查询前驱的方法即可，时间复杂度为 $O(\log_w V)$。

2.6 压位 trie 的空间复杂度

假设一颗大小为 2^k 的压位 trie 的整型个数为 $F(k)$。

容易发现 $F(i) = 1(1 \le 2^i \le w), F(k) = wF(k-c) + 1$，容易得到空间复杂度为 $O(2^k)$，使用一些技巧例如调整第一层子压位 trie 个数，就可以达到 $O(\frac{2^k}{w})$ 的空间复杂度。

2.7 压位 trie 的实现

使用类似线段树的实现，自顶向下搜索节点并且更新是一种可行的方法，但是根据笔者的实现，常数相对来说较大，于是我们考虑改变实现的方法。

我们继续使用类似线段树的结构，但是我们转而使用自底向上更新，笔者这里使用了若干个数组分别记录每一层的信息，$A_{i,j}$ 维护了区间 $[j \times 2^i, (j+1) \times 2^i)$ 中的元素。

当插入时，我们自底向上更新，如果插入前该子树已经有数了，就可以不必继续向上维护信息，直接结束插入函数。

当删除时，我们自底向上更新，如果删除后子树之中还存在数，我们也不必继续向上维护信息，直接结束删除函数。

查询 x 的前驱我们仅仅需要往上寻找到第一个节点，使得子树里存在值比 x 小，然后寻找对应子树之中的最大值即可，查询后继也可以使用类似的方法实现。

该实现方法剪枝较多，所以在一般情况下表现非常优秀，如果有更好的实现方法也欢迎与笔者交流。

2.8 压位 trie 的一些拓展

在部分题目中，对信息的维护可能比较复杂，如果继续使用 w 叉可能无法快速维护，例如我们需要支持插入一个元素，查询 $< x$ 的数的个数。

我们发现新的查询无法在之前的压位 trie 上直接实现的主要原因是，我们无法快速查询前 k 个儿子中一共有几个元素。

我们假设叉数为 B，每当这个子树插入 B 个元素之后，我们重构一次，计算出每个儿子中有几个元素，并求出其前缀和，因此我们只需要快速计算最近 B 个插入的数中有几个 $< x$ 的。

我们考虑使用一种新的方法存储每个子树中的新元素个数，我们使用 1 的个数表示数字的值，由于其存在 B 个子树，所以我们使用 $B-1$ 个 0 当做数据的间隔，因此一个子树中每个儿子有几个数我们就可以使用一个整型压缩存储。而对于修改操作，我们要快速找到第 k 个子树对应的位置，这个就需要我们快速查询一个二进制第 k 个 0 的位置。我们可以使用 Method of Four Russians 来优化，即打出一张表，存储每一个不同的查询的答案。对于查询，我们也只要找到第 k 个 0 就知道前面有几个 1 了，我们就可以 $O(1)$ 支持一个子树元素个数加一，查询前 k 个子树的元素个数总和。如果我们取 $B = \frac{\log_2 n}{3}$，那么预处理复杂度将小于 $O(n)$，因此我们能以均摊 $O(\frac{\log_2 V}{\log_2 \log_2 n})$ 的时间复杂度解决这个问题，而我们也有一些手段将其变为严格 $O(\frac{\log_2 V}{\log_2 \log_2 n})$。

3 vEB tree

我们观察压位 trie 的时间复杂度，可以发现其叉数越大，效率越高，然而当叉数到了 w，由于其整数操作复杂度不再是 $O(1)$，所以其时间可能不减反增。

我们发现其使用整数操作的几个地方主要有：在插入与删除时将一个位设为 0/1，在查询前驱后继时查询一个节点第一个编号 $< x$ 或 $> x$ 的儿子，我们发现第一个操作会进行 $O(\log_w v)$ 次，第二个操作只会进行 1 次！

因此如果我们使用 bitset 维护其儿子，可能会得到一个更快的数据结构。

而 vEB tree 则采取了一定的策略，使得其叉数变大且巧妙的保证了时间复杂度，可以在 $O(\log_2 \log_2 V)$ 的时间复杂度内完成插入、删除，查询一个数的前驱、后继。

3.1 vEB tree 的结构

一个大小为 2^k vEB tree 可以维护一个数字在 $[0, 2^k)$ 中的集合。

若 $k < 2$，则可以轻易使用 2^k 个 *bool* 值来 $O(1)$ 维护这个数据结构。否则我们令 $m = \lfloor \frac{k}{2} \rfloor$。我们将定义 2^{k-m} 个大小为 2^m 的子 vEB tree $A_i (0 \le i < 2^{k-m})$。

我们定义 $high(x) = \lfloor \frac{x}{2^m} \rfloor$，$low(x) = x \bmod 2^m$，即 x 的高位与低位。对于这个 vEB tree 维护的集合，我们记录集合的最大值 *max* 与最小值 *min*（若集合为空则分别为 $-\infty$ 与 ∞），并将除了**最小值**的元素插入其子 vEB tree 当中，对于元素 x，它将在 $A_{high(x)}$ 这个子 vEB 中插入 $low(x)$。

容易发现，对于每个子 vEB tree A_i，它存储了高位为 i 的数的低位，为了加速寻找，我们应该再定义一个大小为 2^{k-m} 的 vEB tree B，其维护了出现的数的高位的集合，其可以用来加速我们的查询。

3.2 vEB tree 的简单操作

对于一个空的 vEB tree，我们插入一个元素仅仅需要将其 *min* 与 *max* 设置为要插入的 x 的数即可，时间复杂度 $O(1)$。

对于一个 vEB tree，可以通过检验其元素最大值与最小值是否相同来判断其集合大小是否为 1，数据复杂度 $O(1)$。

对于一个集合大小仅仅为 1 的 vEB tree，我们删除其最后一个元素仅需要将 *min* 与 *max* 分别设置为 ∞ 与 $-\infty$。

如果在 vEB tree A_x 中存在一个元素 y，那就说明存在元素 $x \times 2^m + y$，所以可以快速计算一个子 vEB tree 中的值在这个 vEB tree 中对应的值。

快速完成这些操作对于 vEB tree 来说是不可缺少的。

3.3 vEB tree 的插入操作

假设我们要在一个大小为 2^k 的 vEB tree 中插入元素 x。

如果该 vEB tree 为空，我们仅需要使用上述方法插入。

如果 x 比当前最小值 *min* 小，我们仅需要交换 x 与原来的 *min*，并且在后续过程中插入原来的 *min*。

如果 x 比当前最大值 *max* 大，我们仅需要更新 *max*。

首先我们需要在 $A_{high(x)}$ 这个 vEB tree 中插入 $low(x)$，并且在 B 这个 vEB tree 中插入 $high(x)$。

容易发现的是，当原来的 $A_{high(x)}$ 非空时，B 中必然已经有元素 $high(x)$，应此仅仅需要在 $A_{high(x)}$ 中进行插入，否则 $A_{high(x)}$ 为空，进行插入仅仅需要设置其 min, max，我们接下来仅仅需要在 B 中插入 $high(x)$。

容易发现，对于大小为 2^k 的 vEB tree，我们定义其插入一个元素的时间为 $F(k)$。

容易得到 $F(0) = O(1), F(1) = O(1)$，对于 $k > 1, F(k) = F(\lceil \frac{k}{2} \rceil) + O(1)$，解得 $F(k) = O(\log_2 k)$，而 k 取恰当的值就能以 $O(\log_2 \log_2 V)$ 的时间完成插入操作。

3.4 vEB tree 的删除操作

假设我们要在一个大小为 2^k 的 vEB tree 中删除元素 x。

如果该 vEB tree 集合大小为 1，我们仅需要使用上述方法删除。

如果 x 是当前最小值 min，那么我们只需要直接删除 min，并且尝试求出新的 min，显然的是，最小值的最高位必然是最小的，我们只需要找到 B 中的最小值并且在对应的子 vEB tree 中找出新的最小值，然后我们删除这个最小值并将 min 赋值为该值。

接下来，我们要在 $A_{high(x)}$ 这个 vEB tree 中删除 $low(x)$，并若 A 删除后为空就在 B 这个 vEB tree 中删除 $high(x)$。

如果 $A_{high(x)}$ 集合大小为 1，则可以 $O(1)$ 删除其中的元素并且在 B 中删除 $high(x)$。

如果 $A_{high(x)}$ 集合大小非 1，那么我们仅仅需要在 $A_{high(x)}$ 中删除 $low(x)$。

容易发现删除的时间复杂度与插入类似，同样是 $O(\log_2 \log_2 V)$。

3.5 vEB tree 的查询前驱操作

假设我们要在一个大小为 2^k 的 vEB tree 中查询 x 的前驱。

首先，x 的前驱与 x 高位相同当且仅当 $A_{high(x)}$ 中的 min 比 $low(x)$ 小，因此可以直接判断它前驱是否在 $A_{high(x)}$ 当中。如果前驱在 $A_{high(x)}$ 当中，我们直接在这个 vEB tree 中查询 x 的前驱。

否则，x 的前驱的高位一定是比 $high(x)$ 小的数中最大数，因此，我们在 B 中查询 $high(x)$ 的前驱 y，而它前驱的低位必然是 A_y 中最大的，即 A_y 的 max。

需要注意的是，如果 B 中的最小值如果大于等于 $high(x)$，那么 x 前驱将会是这个 vEB tree 的 min。

3.6 vEB tree 的查询后继操作

假设我们要在一个大小为 2^k 的 vEB tree 中查询 x 的后继。

首先，x 的后继与 x 高位相同当且仅当 $A_{high(x)}$ 中的 max 比 $low(x)$ 大，因此可以直接判断它后继是否在 $A_{high(x)}$ 当中。如果后继在 $A_{high(x)}$ 当中，我们直接在这个 vEB tree 中查询 x 的后继。

否则，x 的后继的高位一定是比 $high(x)$ 大的数中最小数，因此，我们在 B 中查询 $high(x)$ 的后继 y，而它后继的低位必然是 A_y 中最小的，即 A_y 的 min。

需要注意的是，最小值不在这棵压位 trie 的子结构之中，所以如果 x 比这个压位 trie 中的最小值要小，那么需要直接返回最小值，不然就可能会返回错误的结果。

3.7 vEB tree 的一些简单优化

容易发现的是，当 $k < 2$ 时再使用 2^k 个 bool 来维护集合时，vEB tree 深度常数可能会比较大大。而当 $2^k \le w$ 时，就可以使用一个整型轻松的维护一个大小为 2^k 的集合。如果 $w = 64$，我们维护一个大小为 2^{24} 的集合都仅需要递归两层即可。大大的优化了常数。

在部分使用 $w = 64 = 2^6$ 为测试机器的地方，我们定义大小为 2^{24} 的 vEB tree 的话基本可以满足大部分使用需求，且在其它操作中仅仅需要递归两层，大大提升了效率。

3.8 vEB tree 的实现

由于 vEB tree 的结构相对复杂，难以使用数组直接实现，故可以使用 C++ 中的 template 语法来定义不同大小的 vEB tree。并且特化了大小较小的 vEB tree，使用压位整型来实现，优化常数。使用此方法后代码难度相对降低，较为容易实现。如果有更加优秀的实现方法欢迎与笔者交流。

3.9 vEB tree 的空间复杂度

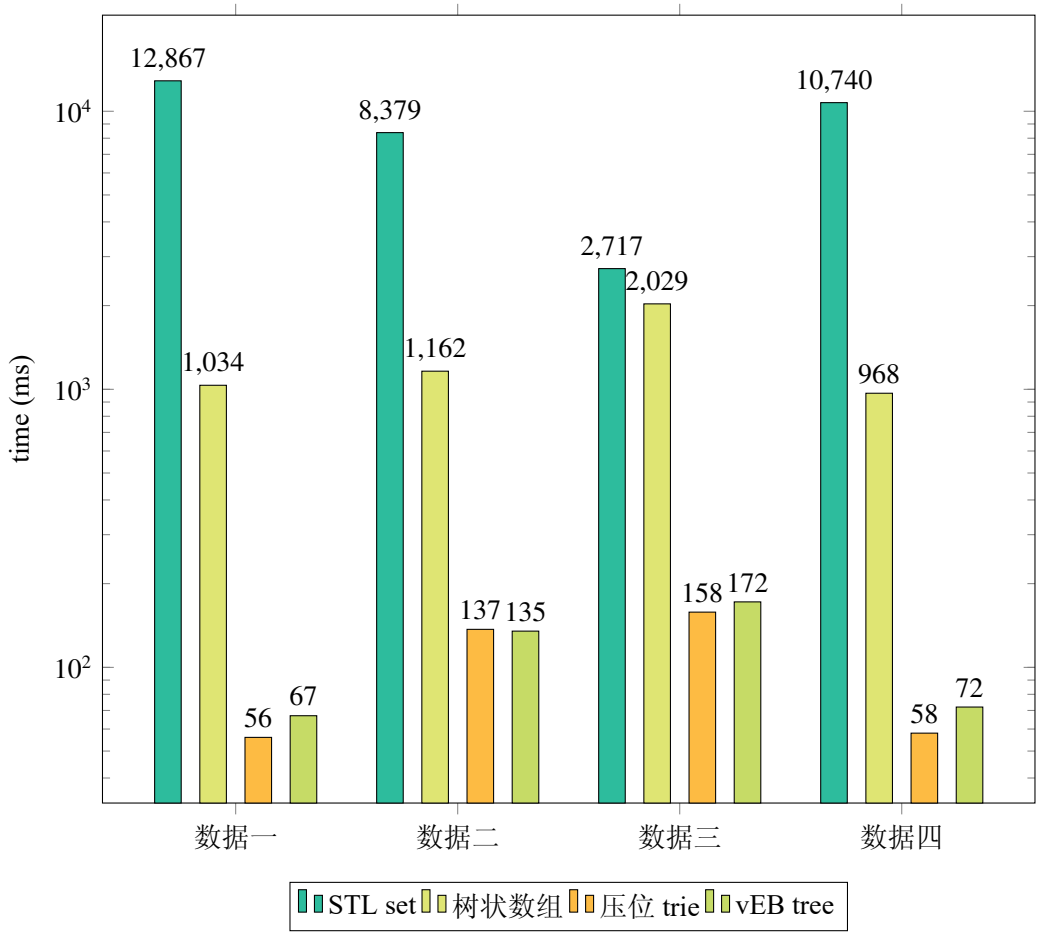
假设一个大小为 2^k 的 vEB tree 的中整型个数为 $F(x)$。

容易得到：$F(i) = 1(1 \le 2^i \le w), F(i) = F(\lceil \frac{i}{2} \rceil) + 2^{\lceil \frac{i}{2} \rceil} F(\lfloor \frac{i}{2} \rfloor) + 2$。

我们可以证明 $F(x)$ 可能会达到 $O(\frac{2^k}{\sqrt{w}})$，不过可以通过调整 m 的大小即 vEB tree 的叉数能够得到 $O(\frac{2^k}{w})$ 的空间复杂度。

如果值域范围特别大，我们也可以考虑使用 hash 表等方法存储 vEB tree 以得到可以接受的空间复杂度。

4 对于部分数据结构效率的一些测试[5]



在所有数据中，操作个数均为 10^7，值域范围为 $[0, 2^{24})$。

其中第一组数据为：插入 10^7 个不同的数字。

其中第二组数据为：插入 10^6 个不同的数字后进行 9×10^6 次前驱/后缀查询。

其中第三组数据为：插入 10^5 个不同的数字后进行 99×10^5 次前驱/后缀查询。

其中第四组数据为：插入 5×10^6 个不同的数字后再删除这 5×10^6 个数，顺序随机。

所有数字均随机生成。

从时间角度来说：可以发现 vEB tree 和压位 trie 在时间上还是远远快于其它两个 log 数据结构，在集合数字比较稀疏的情况下的查询甚至比树状数组快了 10 倍多。相对来说，vEB tree 的插入删除略慢于压位 trie，但是其查询效率还是可以的，但是这仅仅是随机数据，在更加强力的数据下 vEB tree 可能会有相对更高的查询效率。但是压位 trie 已经完全能够满足我们的使用需求。

从空间角度来说：四个数据结构使用的空间分别为：455MB, 64MB, 2.06MB, 2.03MB。

[5]测试代码可以在 https://skip2004.blog.uoj.ac/blog/6551 查看

可以见压位 trie 与 vEB tree 在操作数和值域基本同阶的情况下，空间也是非常占优势的。而树状数组的空间也相对来说较小。

从实现难度来说，笔者认为压位 trie, vEB tree 实现难度均在可接受范围之内，而压位 trie 相对来说更加容易实现，且在很多情况下效率更高。

总的来说，STL set 虽然容易实现，但是消耗时间空间都较大，笔者比较推荐使用树状数组或者压位 trie 来解决这一类问题。而在下面一些例题中可以看到，压位 trie 在实际中的应用还是比较广泛的。

5 简单应用

5.1 【北大集训 2018】ZYB 的游览计划

5.1.1 题目大意

有一棵 N 个顶点的树，和一个 $1 \sim N$ 的排列 P。

定义一个整数区间 $[L,R]$ 的权值为从 1 号点出发遍历完 $P_L, P_{L+1}, \cdots, P_R$ （不一定按顺序）中的所有点最后回到 1 号点需要经过的最少边数。

现在将 $[1,N]$ 划分成 K 个区间，问权值和的最大值。

数据保证 $2 \le K \le N \le N \times K \le 2 \times 10^5$。

时间限制：7s

空间限制：512MB

5.1.2 做法

对于本题，我们首先可以使用决策单调性来优化 DP，由于内容和本文无关，这里略去不讲。我们要解决的问题是：维护一个集合 S，支持往里面插入一个点与删除集合内的点，查询从 1 号点出发遍历完集合内所有点需要经过的最小边数，而其中的插入删除次数会达到 $O(nk \log_2 n)$，相对来说较大。

容易发现的是，我们按 dfs 序以此访问就是最优的答案了，因此我们需要维护 dfs 序中相邻两个点的距离之和以及 1 号点与 dfs 序最小和最大的点的距离。

我们在里面插入一个点 b，首先求出这个点 dfs 序之前的点 a 和 dfs 序之后的点 c，我们发现插入点 b 之后的答案会增加 $dist(a,b) + dist(b,c) - dist(a,c)$。

我们在里面删除一个点 b，首先求出这个点 dfs 序之前的点 a 和 dfs 序之后的点 c，我们发现删除点 b 之后的答案会减小 $dist(a,b) + dist(b,c) - dist(a,c)$。

如果其 dfs 序之前或者之后的点不存在，那么我们将其设置为 1 号点即可。

因此我们可以使用 vEB tree 来维护集合内点的 dfs 序集合，使用 ST 表 + 欧拉序求 lca 以及点对距离即可做到 $O(nk\log_2\log_2\log_2 n)$ 的时间复杂度。

相对于原题给出的 $O(nk\log_2^2 n)$ 做法，上述做法的复杂度与常数更加优秀。笔者使用压位 trie 实现了该题目，经过测试可以在 150ms 左右内解决这个问题，远小于原题所要求的 7s 的时间限制。

5.2 【ZJOI 2019】语言

5.2.1 题目大意

有一棵 n 个顶点的树，和 q 条树上的链 (u_i, v_i)。

对于一条链，链上的任意不同的两个点之间都可以展开贸易活动。

询问一共有多少点对之间可以开展贸易活动。

时间限制：3s

空间限制：512MB

5.2.2 做法

我们先假设一个城市能与自己展开贸易活动，然后考虑计算有序点对数量 (u, v) 为 ans，那么答案就是 $\frac{ans-n}{2}$。

我们考虑计算可以和每个点开展贸易活动的点数。

容易发现，能和一个点展开贸易活动的点集（包括其自己）是覆盖它的若干条链的并，因此其为一个连通块，我们只需要包含它的链的端点，求出包含这些端点的最小连通块大小。

在常规做法之中，我们使用动态开点线段树来维护点集内点的 dfn 序集合，我们要求的连通块大小显然就是 dfn 序相邻的点距离加上 dfn 序最小的点与最大的点的距离之和的一半，然后线段树每个节点记录区间 dfn 最小与最大的点以及这个区间相邻的点的距离和，然后可以轻易上传信息，再使用线段树合并就可以达到 $O(n\log_2 n)$ 的时间与空间复杂度。

然而线段树做法时间空间常数都略大，我们考虑继续优化。

我们考虑使用动态开点压位 trie 维护 dfn 序集合。

使用一个数组 $ch[x][i]$ 表示一个压位 trie x 的第 i 个子压位 trie 的编号，$w[x]$ 表示压位 trie x 有哪些子压位 trie 非空，本题还需要记录子树最小最大值。

对于插入/删除一个节点，我们在压位 trie 上直接寻找要插入的儿子编号，与动态开点线段树类似，若不存在该儿子我们则新建一个节点，插入之后我们使用类似上一题的方法更新答案。

对于合并两个压位 trie，我们考虑使用类似线段树合并的方式实现：

若两个节点中有一个是空节点，则返回另外一个。

我们选取儿子个数大的节点作为新的根，并在之后把另外一个节点删除，在删除之前，我们需要将还有一个节点的儿子与这个节点的儿子合并。

对于它们的公共儿子，我们直接枚举并且合并，这里可以通过与运算来快速获得它们的公共儿子。

对于第一个节点没有但第二个节点有的儿子，直接暴力赋值。

我们考虑分析其时间复杂度。首先合并两个节点的公共儿子部分时间复杂度与线段树合并类似，每次合并都会少一个节点，所以这部分总时间复杂度是 $O(n\log_w n)$ 的。而对于暴力赋值对于第一个节点没有但第二个节点有的儿子的部分，可以发现第二个节点的儿子里必定至少有一个元素，而赋值的子树是互不相交的，因此赋值次数不会超过两个压位 trie 大小的较小值，类似于启发式合并，这部分总复杂度为 $O(n\log_2 n)$，但常数非常小，复杂度瓶颈部分仅仅为一些赋值语句。

本题还需要在合并的同时改变答案，例如合并两个节点公共儿子时需要注意这个子树中第一个元素和最后一个元素改变后需要重新计算其与前驱后继的距离。

但是，它还存在一些弊端，其空间复杂度较大，需要 $O(n\log_w n)$ 个节点，每个节点需要 $O(w)$ 个整型来存储儿子编号，总空间复杂度为 $O(nw\log_w n)$，因此我们需要继续优化空间复杂度。

我们首先使用重链剖分，在树上 dfs 时候第一次往重儿子走，然后将当前 trie 设为重儿子的 trie，可以发现这样最多同时存在 $O(\log_2 n)$ 棵 trie，使用较好的实现，一棵 trie 中只会存在 $O(\frac{n}{w})$ 个节点，其占用的空间最多为 $O(n)$，使用内存回收[6]可以做到 $O(n\log_2 n)$ 的空间复杂度，且常数较小。

因此，我们使用压位 trie 得到了一个时间空间常数更加小的做法，并且这种压位 trie 合并的思想可以拓展到一些类似的题目之中。

总结

本文介绍了压位 trie 和 vEB tree 两种可以用于解决 Dynamic Predecessor Problem 的数据结构，并且给出了其在部分 OI 题目中的应用。并且本文介绍的以及其它亚 log 数据结构可能还有更多有趣的作用，故希望本文能起到抛砖引玉的作用，希望有兴趣的读者能继续研究，得到更多有趣的应用。

感谢

感谢中国计算机学会提供学习和交流的平台。

感谢国家集训队教练高闻远的指导。

[6]内存回收时需要注意清空的时间复杂度，不要使用类似于 memset 的方式。

感谢父母对我的培养和教育。

感谢学校的栽培，符水波老师，应平安老师，郁庭老师，林乃杰老师，董毅老师的教导和同学们的帮助。

感谢虞皓翔同学，潘佳奇同学，翁伟捷同学，戴江齐同学，与我交流讨论、给我启发。

感谢罗恺同学，施开成同学，孙睿泽同学为本文审稿。

参考文献

[1] Wikipedia, the free encyclopedia,“Predecessor problem”, https://en.wikipedia.org/wiki/Predecessor_problem.

[2] Wikipedia, the free encyclopedia,“Van Emde Boas tree”, https://en.wikipedia.org/wiki/Van_Emde_Boas_tree.

[3] Wikipedia, the free encyclopedia,“Method of Four Russians”, https://en.wikipedia.wikimirror.org/wiki/Method_of_Four_Russians.

[4] Peter van Emde Boas，*Preserving order in a forest in less than logarithmic time*，Proceedings of the 16th Annual Symposium on Foundations of Computer Science 10: 75-84, 1975

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*

[6] 中国计算机学会, 2018 全国信息学奥林匹克年鉴