Y-Fast Trie Algorithm

汪苏轶

PKU

2025 年 12 月 10 日

Introduction

使用一个数据结构来维护一个由非负整数构成的集合，支持以下几种操作：

- 向集合中插入一个整数。
- 从集合中删除一个整数。
- 查询某数在集合中的前驱（集合中小于等于给定数的最大数）。
- 查询某数在集合中的后继（集合中大于等于给定数的最小数）。
- 查询集合中的最大值与最小值。

不妨令插入数值的值域为 V，集合中数的个数为 n。

Related Algorithm

为了维护上述内容，我们通常会使用以下几种数据结构：平衡树(以 STL 中的 set 为例)、树状数组或权值线段树、van Emden Boas Tree（下简称 vEB），以及将要介绍的 Y-Fast Trie。

下表为这几种算法在该问题上的时空复杂度对比：

	平衡树	**权值线段树**	**vEB Tree**	**Y-Fast Trie**
ins/del	$O(\log V)$	$O(\log V)$	$O(\log\log V)$	$O(\log\log V)$
pre/suc	$O(\log V)$	$O(\log V)$	$O(\log\log V)$	$O(\log\log V)$
max/min	$O(1)$	$O(\log V)$	$O(1)$	$O(1)$
space	$O(n)$	$O(V)$	$O(V)$	$O(n)$

表: 各种算法在该问题上的时空复杂度对比

Related Algorithm

由此可见，Y-Fast Trie 在各种操作的时间复杂度均较为优秀，且能够做到与插入元素数量线性的空间复杂度。

Problem Simplification

以下算法均建立在“假设我们能够实现一个支持 $O(1)$ 查询元素值的哈希表（即散列表）”的前提之下。

首先对问题进行一些简化。我们尝试使用一个双向链表来按照大小顺序维护出当前在集合中的所有元素，用一个哈希表来维护值到链表节点指针的映射。

如果我们能够实现如下操作：给定一个数，查询集合中这个数的前驱**或**后继之一。那么我们便可以用 $O(1)$ 的额外时间来完成原问题的所有其他操作。

Trie Algorithm

在介绍 Y-Fast Trie 算法之前，首先需要引入 Trie 和 X-Fast Trie 两种算法。

Trie 即字典树，在此处我们仅考虑 01Trie。其结构是一棵二叉树，且所有叶子节点的深度均相同。

对于一个叶子节点，表示一个整数，其代表的整数是从根节点向下走到该节点所经过所有边的边权拼接而成所形成的二进制数。

对于一个非叶子节点，其代表的即为一段二进制数的前缀，其子树内的所有叶子节点代表包含这一段前缀的所有二进制数。

Trie Approach

将所有集合中的数插入到一棵 Trie 中，并在每个节点统计该节点的子树内有多少个叶子节点的值存在于集合中，同时维护子树内的最大值和最小值。

每次查询前驱后继时，从根出发从高到低考虑每个二进制位。假设当前位于 Trie 上的节点是 p，查询数对应的二进制位是 x：

- 如果 p 有 c 儿子，且该儿子的子树内元素数量非零，那么从 p 移动到对应的儿子，考虑下一位。
- 否则 p 的 $c \oplus 1$ 儿子一定非空。找到对应儿子内的最大值或最小值即为所要查询的前驱或后继。

不难发现该算法插入删除、查找前驱后继的复杂度均为 $O(\log V)$。

X-Fast Trie Algorithm

对普通的 Trie 进行一些简单优化可以得到 X-Fast Trie 算法。

考虑对与 Trie 的每一层节点分别维护一个散列表。该散列表维护从节点值（即表示的二进制前缀）到节点的映射。该散列表可以在新建 Trie 节点的同时轻松维护。

在查询操作时，考虑使用二分查找的办法。对二进制前缀长度进行二分查找，这样容易判断查询值的一段二进制前缀在 Trie 中对应的节点是否为空。这样只需经过 $O(\log \log V)$ 轮查找过程即可定位到能够走到的最深的 Trie 树节点。

使用优化后的算法被称为 X-Fast Trie 算法，在上述问题中，该算法插入删除元素复杂度仍为 $O(\log V)$，但是查找前驱后继复杂度优化为 $O(\log \log V)$。

Y-Fast Trie Algorithm

可以发现，在使用 X-Fast Trie 算法时，瓶颈在于将一个树插入 Trie 需要 $O(\log V)$ 的时间。Y-Fast Trie 算法在 X-Fast Trie 算法的基础上，使用了底层分块的思想来进行进一步优化。

Y-Fast Trie Algorithm

考虑将集合中的所有数按照从小到大的顺序，以 $\log V$ 的大小为一块进行分块。在每一块内，我们使用一棵红黑树来维护块中的 $\log V$ 个元素。

对于每一个块，我们选定一个代表元。这个代表元不一定在这个块的元素中，但是一定位于这个块的最大值与下一个块的最小值之间。

我们再使用一棵 X-Fast Trie 来维护所有块的代表元。

Predecessor and Successor

如果我们能够动态维护出这样一个结构，那么查询前驱后继的操作就非常简单：首先在 X-Fast Trie 中查询对应值的前驱或后继，随后在对应代表元所在块中查询对应值的前驱后继。

查询前驱后继的时间复杂度：顶层 X-Fast Trie 的复杂度为 $O(\log \log V)$，在块中红黑树内只有 $O(\log V)$ 个元素，因此复杂度也是 $O(\log \log V)$。

Insert Operation

问题在于，在插入和删除元素时，我们无法时刻保证每个块的大小始终为 $\log V$。因此，我们不再要求每个块的大小恰好为 $\log V$，而是限定每个块的大小始终在 $(\frac{\log V}{2}, 2\log V)$ 之间。不难发现，这样的限定不会影响查询前驱后继的复杂度。

对于插入操作，首先找到该值所对应的块，并插入到对应的红黑树中。如果插入后红黑树大小仍然小于 $2\log V$，那么符合条件。插入复杂度 $O(\log\log V)$。

否则，红黑树大小必然恰好为 $2\log V$，我们将红黑树内的所有元素取出来，按大小划分为两个大小均为 $\log V$ 的集合，重新建立两棵红黑树，并将代表元在顶层的 X-Fast Trie 中进行更新。这样操作一次时间复杂度为 $O(\log V)$。

Delete Operation

再考虑删除操作。同样找到删除值所属的块，并将对应值从对应的红黑树中删除。时间复杂度 $O(\log \log V)$。

如果删除后块的大小仍大于 $\frac{\log V}{2}$，那么合法。否则找到这个块的下一个块，将这两个块进行合并，合并后块的大小必然在 $(\log V, \frac{5}{2} \log V)$ 之间。

- 如果块的大小在 $(\frac{3}{2} \log V, \frac{5}{2} \log V)$ 之间，那么就再进行一次分裂操作，变成两个大小在 $(\frac{3}{4} \log V, \frac{5}{4} \log V)$ 之间的块。
- 否则块大小必然在 $(\log V, \frac{3}{2} \log V)$ 之间。

不难发现，一次删除最多会导致两次块之间的分裂与合并。单次分裂合并复杂度均为 $O(\log V)$。

Time Complexity

分析上述算法中插入与删除元素的时间复杂度。如果没有导致块的分裂与合并，那么复杂度显然是 $O(\log \log V)$。

如果导致了块的分裂与合并，发现操作后块的大小必然会在 $(\frac{3}{4}\log V, \frac{3}{2}\log V)$ 之间，那么接下来的 $\frac{1}{4}\log V$ 次在该块的插入与删除元素操作必然不会导致块的重构。

将所有块的分裂合并操作均摊到每一次操作上可以发现，对于一次操作可以认为该复杂度是 $O(1)$ 的。

Space Complexity

分析空间复杂度：底层红黑树节点个数即为集合中元素的个数。而每加入 $O(\log V)$ 个元素会导致新建一个顶层 X-Fast Trie 的节点，消耗 $O(\log V)$ 的额外空间。

故均摊空间复杂度为 $O(n)$。

Comparison of Trie, X-Fast Trie and Y-Fast Trie

比较三种 Trie 的变种在该问题上的时空复杂度效率：

	Trie	**X-Fast Trie**	**Y-Fast Trie**
ins/del	$O(\log V)$	$O(\log V)$	$O(\log \log V)$
pre/suc	$O(\log V)$	$O(\log \log V)$	$O(\log \log V)$
max/min	$O(1)$	$O(1)$	$O(1)$
space	$O(n \log V)$	$O(n \log V)$	$O(n)$

表: 各种算法在该问题上的时空复杂度对比

Summary

Y-Fast Trie 在处理该问题的理论时空复杂度是所有算法中较为优秀的。

但是该算法在实际应用中仍然存在一些缺陷，例如依赖于散列表的实现，以及底层红黑树，顶层 X-Fast Trie 的结构导致常数较大，在处理小规模数据的情况下表现不如其他的几种算法等。

Bibliography

[1]. Stanford University, Department of Computer Science. (n.d.). CS166 Lecture Slides: Data Structures [Slides]. Stanford University.