组队作业 Y-fast Trie 报告

方博慧，秋闻达，徐值天

2016年4月29日

目录

0 提要

本篇报告大致会介绍我们组实现或学习的以下几种数据结构。

	Splay	X-fast Trie	Y-fast Trie	动态完美哈希	vEB
分类	平衡树	前缀树	前缀树	哈希表	树/堆
Space	$O(n)$	$O(n \log M)$	$O(n)$	$O(n)$	$O(M)$
Find	均摊$O(\log n)$	$O(\log \log M)$	$O(\log \log M)$	$O(1)$	$O(\log \log M)$
Insert	均摊$O(\log n)$	$O(\log \log M)$	$O(\log \log M)$	均摊$O(1)$	$O(\log \log M)$
Delete	均摊$O(\log n)$	$O(\log \log M)$	$O(\log \log M)$	均摊$O(1)$	$O(\log \log M)$

其中n为存储的数据个数，M为存储的数据范围。

由于Splay课本中有，就不包括在本篇报告中了。

1 X-fast Trie

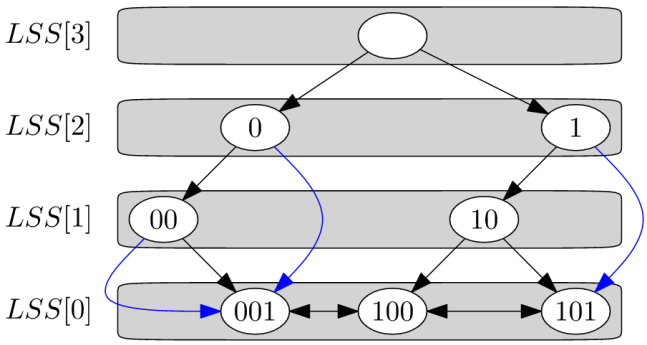
1.1 功能简介

X-fast Trie是一个用于存储有界整数的数据结构，它支持$O(\log \log M)$的查找元素、找前驱、找后继操作，使用$O(N \log M)$的空间（其中N表示存储的整数的个数，M是界的大小）。

1.2 结构

一个X-fast Trie是一个二进制前缀树，每个非叶子节点存储的值为其子树中所有叶子节点表示的值的二进制表示的公共前缀，它的左孩子为其二进制表示后添加一个0，右孩子为其二进制表示后添加一个1。

一个在0到$M-1$的整数的二进制表示需要占用$\log M$位，所以X-fast Trie的高度为$\log M$。



X-fast Trie里所有的*value*存储在叶子节点，非叶子节点存在当且仅当它的子树里面存在叶子节点。如果一个非叶子节点没有左孩子，那它会存储一个后代指针指向其右子树的最小叶子节点，。如果没有右孩子，那它会存储一个指向左子树里最大叶子节点的后代指针。每个叶子节点存储一个指针指向它的前驱和后继，从而在X-fast Trie的底层形成一个双向链表。最后，每一层使用一个哈希表存储那一层上的节点，这些哈希表形成一个层次搜索的结构记为LSS。为保证复杂度，这里采用动态完美哈希（详细说明见下一章节）的方式。最后总的空间复杂度为$O(N \log M)$,因为每个叶子节点到根的路径的长度都是$\log M$。

1.3 操作

(a) Find(key)

查找和key绑定在一起的value。

我们只需在LSS[0]这个哈希表中用常数级别的时间查询即可。

(b) Successor(key) and Predecessor(key)

Successor：查找大于等于给定key的最小的$\langle key, value\rangle$pair。

Predecessor：查找小于等于给定key的最大的$\langle key, value\rangle$pair。

我们首先找到key的最低祖先u，满足u的二进制表示是key的二进制表示的前缀且u的深度最大，这需要使用$\log \log M$的时间。

然后通过节点u的后代指针走到一个叶子节点v，根据情况选择走到v的后继或者不走或者（v的前继）来找到Successor（Predecessor）。考虑树高为$\log M$，总复杂度$\log \log M$。

(c) Insert(key,$value$)

为了插入一个$\langle key, value\rangle$pair，我们首先找到$key$的前驱和后继，然后给$key$创造一个叶子节点$u$，插入底层的双向链表。接着从根走到这个叶子节点，一路创造新的节点，维护好哈希表和后代指针。

总复杂度$O(\log M)$。

(d) Delete(key)

首先在底层的哈希表里找到这个key所对应的叶子节点u，从双向链表中删除它，然后从该叶子节点走到根，一路删除有必要删除的节点并维护好路上节点的后代指针。

类似插入操作，复杂度$O(\log M)$。

2 动态完美哈希

2.1 功能简介

动态完美哈希（Dynamic perfect hashing）是计算机科学领域中解决哈希冲突的一类技术。它占用的内存相比同类的哈希方法在常数上会变多，所以这个方法通常适用于大量元素的快速插入、删除、查询。

2.2 原理说明

由FKS Scheme可知，在静态问题中，进行两次哈希。对x个插入操作，最好在第一维哈希时，取第一维桶的个数为

$$s = 2 \cdot (x - 1)$$

对于每个第一维哈希后被插入$k[i]$个元素的桶i，令其第二维容量为$k[i]^2$，并使用全域哈希函数使得第二维$k[i]$个元素，经过这个二次哈希后不发生冲突。设第二维哈希表占总空间为T，只要第一维哈希函数经过特别选取，即可满足

$$T < s + 4 \cdot x$$

Fredman,Komlós和 Szémeredi证明过，只要给出一系列全域哈希（a universal hashing family of hash functions），那么其中至少一半都具有这个性质。因此，该静态问题的空间是$O(n)$的，时间是$O(1)$的。

在动态问题中，Dietzfelbinger et al.证明过，内存查询是常数时间所以最坏$O(1)$的，总内存需求是$O(n)$的，插入与删除复杂度均摊是$O(1)$的。与静态问题不同的是，在当前总操作次数超过一个界M的时候，我们便对所有的元素进行重新哈希，可证明总插入和删除操作次数均摊是$O(1)$的，通常取M为总元素数的常数倍，重新哈希后计数器制为当前元素数。

2.3 具体实现

(a) 重哈希

取出所有元素，重置计数器，计算M。不停寻找第一维的哈希函数，直至得到满足，$s(M)$代表有元素的桶的个数。再对每个桶，重开空间，找到各自的哈希函数使得内部不冲突。

(b) 检查

计数器++，如果计数器超过M，重新哈希。

(c) 插入

检查。如果插入不冲突，直接插入；否则对这个桶进行哈希函数的重新选取，直至内部不存在冲突。每个桶内部也存在一个界$m[j]$，如果元素个数超过$m[j]$，则倍长$m[j]$，内存大小为

$$s[j] = 2m[j] \cdot (m[j] - 1)$$

(d) 删除

找到对应位置，存在即删除。检查。

(e) 查询

找到对应位置，返回答案。

2.4 实现细节

(a) 哈希函数的选取

我们最后在网上选取了6种不同的哈希方法，后来又将被哈希的数字转化成k进制数，再哈希，k也是要每次重新随机，这样即解决了不同的哈希函数经常等价的问题。

(b) 打标签

由于被插入的元素还有一维用以存储的$value$值，如果每次删除都真的

删除，调用析构函数可能会使得性能无形中下降，因此在插入、删除时开一个布尔数组代表这个位置是否有元素即可。

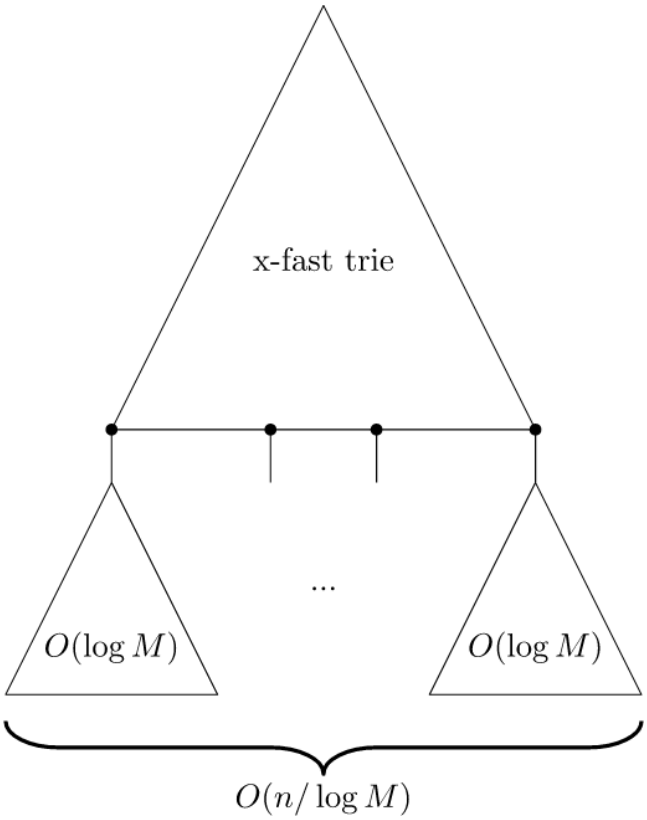
3 Y-fast Trie

3.1 功能简介

Y-fast Trie类似X-fast Trie，只不过它的空间复杂度为$O(N)$。

3.2 结构

Y-fast Trie由一个X-fast Trie和许多平衡二叉树（本次作业中我们组选用了Splay）组成。



我们把key分成$\log M$组连续的元素，每一组是一个平衡二叉树。为了保证复杂度，每个平衡二叉树的大小保持在$\frac{\log M}{4}$和$2 \cdot \log M$之间。在每个平衡二叉树中我们选择一个代表元r将其存储在X-fast Trie中。代表元r不一定要是平衡二叉树中的值，只要其在平衡二叉树的最小值最大值之间。考虑X-fast Trie存储$\frac{n}{\log M}$个代表元并且每个代表元出现在$\log M$个哈希表中，以及所有的平衡二叉树总共存储n个元素，所以Y-fast Trie使用的空间为

$$\frac{n}{\log M} \cdot \log M = O(N)$$

3.3 操作

(a) Find(key)

我们在X-fast Trie中找到key的前驱和后继，然后在这两个点所对应的平衡二叉树中查找key即可。

复杂度为$O(\log \log M)$。

(b) Predecessor(key) and Successor(key)

类似find操作，我们在X-fast Trie中找到key的前驱和后继，在那两个点所对应的平衡二叉树中查找$predecessor$和$successor$，注意可能在后继的后继或者前驱的前驱对应的二叉树中。

复杂度为$O(\log \log M)$。

(c) Insert(key,$value$)

首先我们找到包含key的后继的平衡二叉树T，将key插入其中。

如果之后T的$size$大于$2 \cdot \log M$，那么将树T分裂成两个平衡二叉树，满足其$size$差至多为1。先在X-fast Trie中删除树T对应的叶子节点，然后在新生成的两个平衡树中选出两个代表元插入X-fast Trie中。整个过程中注意维护好哈希表和双向链表、后代指针。

复杂度为$O(\log \log M)$。

(d) Delete(key)

类似插入操作，首先找到key在哪个平衡二叉树中，然后先在平衡二叉树中删除那个数。如果平衡二叉树的$size$为0，那么在X-fast Trie中删除这个节点，如果其$size$小于$\frac{\log M}{4}$，那么将其和其前驱或者后继合并，合并后如果$size$大于$2 \cdot \log M$，那么还要执行分裂操作。整个过程中注

意维护好底层的双向链表，哈希表以及所有X-fast Trie中节点的后代指针。

复杂度为$O(\log\log M)$。

4 vEB树

4.1 功能简介

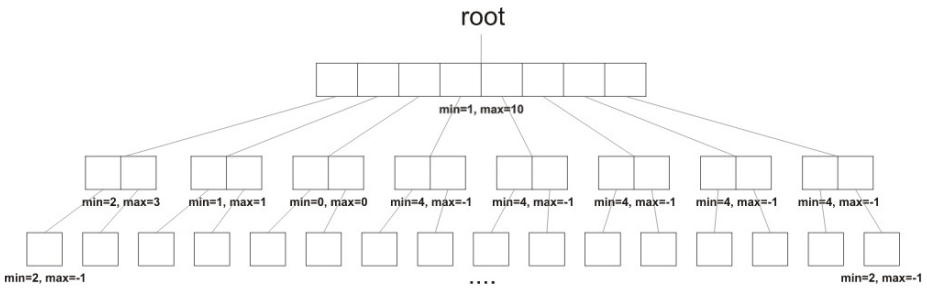
vEB树由Peter van Emde Boas在1975年发明。

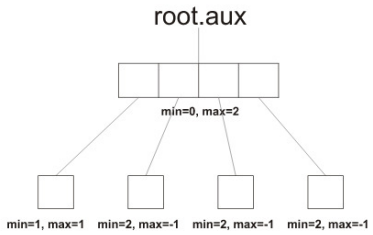
类似于Y-fast Trie，vEB树（或称vEB堆），是一种存储m位二进制整数的数据结构。他能支持以下操作：

- Insert：插入一个$\langle key, value\rangle$pair，其中$key$为$m$位二进制整数
- Erase：删除一个给定key的$\langle key, value\rangle$pair
- Find：查询给定key的$value$
- Successor：查询给定key的后一个key
- Predecessor：查询给定key的前一个key
- Min and Max：查询最小值和最大值

4.2 结构

vEB树由树T和树$T.aux$组成。记$M = 2^m$，所以vEB树存储的范围是$[0, M-1]$内的整数。





树T的每个结点负责一段区间$[0, M-1]$，每个结点的孩子$T.children[i]$指向的是一颗负责区间$[i\sqrt{M}, (i+1)\sqrt{M}-1]$的子树。每个结点记录的是这段区间内存在的最小值$T.min$和最大值$T.max$。为了方便起见，如果这棵子树为空，则令$T.min = M$，$T.max = -1$。

需要特别注意的是，每个数x真正存储的位置是在结点T_i，其中$T_i.min = x$。

树$T.aux$与树T类似，只不过树$T.aux$只存储那些非空的结点。

4.3 操作

(a) Insert(T,key)

- T为空：$T.min = T.max = key$
- $key < T.min$：将$T.min$插入到负责其所在的子树中，并将$T.min$改成key
- $key > T.max$：将key插入到负责其所在的子树中，并将$T.max$改成key
- $T.min < key < T.max$：将key插入到负责其所在的子树中，并维护$T.aux$

- 否则：出现了重复的key

(b) Erase(T,key)

- $key = T.min = T.max$：按照约定将$T.min$置为M，$T.max$置为-1
- $key = T.min$：找到T中存储的第二小的key'并赋给$T.min$，之后在对应子树中删除key'。第二小的key值可以由$T.aux.min$或是$T.max$在$O(1)$时间内得到。
- $key = T.max$：在对应子树中删除key，回溯更新$T.max$（或者由$T.children[T.aux.max].max$或$T.min$在$O(1)$时间内直接得到）
- $T.min < key < T.max$且对应子树不为空：在对应子树中删除key
- $T.min < key < T.max$且对应子树为空：不存在这个key

注意：以上所有情况均需要同时维护$T.aux$。

(c) Successor(T,key)
有以下几种情况：

- $key \leq T.min$: 查询结果为$T.min$
- $key > T.max$: 无后继结点
- 否则：令$i = key/\sqrt{M}$，如果$key < T.children[i].max$，则递归进入$T.children[i]$继续以上步骤，否则我们只需要找到第一个j，使得$T.children[j]$，那么$T.children[j].min$即为答案。该过程可以用$T.aux$加速。

(d) Min and Max
只需要查询树跟的$T.min$和$T.max$即可。

由于Predecessor和Find与Successor极其类似，在此不做赘述。

4.4 时空复杂度

用$S(M)$表示存储范围为M的最朴素的vEB树的空间复杂度。根据树的结构，我们可以得出

$$S(M) = O(\sqrt{M}) + (\sqrt{M} + 1) \cdot S(\sqrt{M})$$

Rex, A.使用归纳法证明了

$$S(M) = M - 2$$

用$T(M)$表示存储范围为M的最朴素的vEB树的时间复杂度。所以最坏情况下每次操作的复杂度为

$$T(M) = T(\sqrt{M}) + O(1)$$

如果记$2^m = M$，即

$$T(m) = T(m/2) + O(1)$$

所以有

$$T(m) = \log m$$

也就是

$$T(M) = \log \log M$$

类似Y-fast Trie，vEB也可以组合一些其他的数据结构或技巧来优化$O(M)$的空间复杂度。

参考文献

[1] Willard, Dan E. (1983). "Log-logarithmic worst-case range queries are possible in space Θ(N)". Information Processing Letters (Elsevier) 17 (2): 81–84.

[2] Bose, Prosenjit; Doueb, Karim; Dujmovi, Vida; Howat, John; Morin, Pat (2010), Fast Local Searches and Updates in Bounded Universes, Proceedings of the 22nd Canadian Conference on Computational Geometry (CCCG2010), pp. 261–264.

[3] Schulz, André ; Christiano, Paul (2010-03-04). "Lecture Notes from Lecture 9 of Advanced Data Structures (Spring '10, 6.851)". Retrieved 2011-04-14.

[4] Kementsietsidis, Anastasios; Wang, Min (2009), Provenance Query Evaluation: What's so special about it?, Proceedings of the 18th ACM conference on Information and knowledge management, pp. 681–690.

[5] Fredman, M. L., Komló s, J., and Szemeré di, E. 1984. Storing a Sparse Table with 0(1) Worst Case Access Time. J. ACM 31, 3 (Jun. 1984), 538-544.

[6] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. 1994. "Dynamic Perfect Hashing: Upper and Lower Bounds". SIAM J. Comput. 23, 4 (Aug. 1994), 738-761.

[7] Erik Demaine, Jeff Lind. 6.897: Advanced Data Structures. MIT Computer Science and Artificial Intelligence Laboratory. Spring 2003.

[8] Yap, Chee. "Universal Construction for the FKS Scheme". New York University. New York University. Retrieved 15 February 2015.

[9] 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)

[10] Rex, A. "Determining the space complexity of van Emde Boas trees". Retrieved 2011-05-27.