Y-fast trie

In computer science, a **y-fast trie** is a data structure for storing integers from a bounded domain. It supports exact and predecessor or successor queries in time O(log log *M*), using *O*(*n*) space, where *n* is the number of stored values and *M* is the maximum value in the domain. The structure was proposed by Dan Willard in 1982[1] to decrease the *O*(*n* log *M*) space used by an x-fast trie.

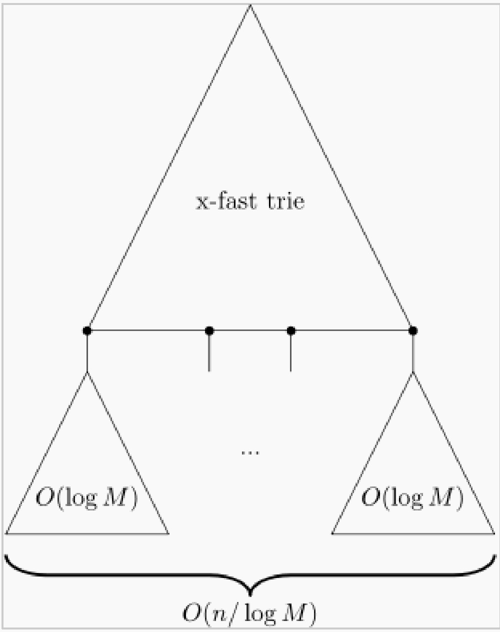
Y-fast trie	
Type	Trie
Invented	1982
Invented by	Dan Willard
Asymptotic complexity in big O notation	
Space	*O*(*n*)
Search	*O*(log log *M*)
Insert	*O*(log log *M*) amortized
Delete	*O*(log log *M*) amortized

Contents

1 Structure
2 Operations
2.1 Find
2.2 Successor and Predecessor
2.3 Insert
2.4 Delete
3 References
4 External links

Structure [edit]

A y-fast trie consists of two data structures: the top half is an x-fast trie and the lower half consists of a number of balanced binary trees. The keys are divided into groups of *O*(log *M*) consecutive elements and for each group a balanced binary search tree is created. To facilitate efficient insertion and deletion, each group contains at least (log *M*)/4 and at most 2 log *M* elements.[2] For each balanced binary search tree a representative *r* is chosen. These representatives are stored in the x-fast trie. A representative *r* need not to be an element of the tree associated with it, but it does need be an integer smaller than the successor of *r* and the minimum element of the tree associated with that successor and greater than the predecessor of *r* and the maximum element of the tree associated with that predecessor. Initially, the representative of a tree will be an integer between the minimum and maximum element in its tree.



An example of a y-fast trie. The nodes shown in the x-fast trie are the representatives of the *O*(*n* / log *M*) balanced binary search trees.

Since the x-fast trie stores *O*(*n* / log *M*) representatives and each representative occurs in *O*(log *M*) hash tables, this part of the y-fast trie uses *O*(*n*) space. The balanced binary search trees store *n* elements in total which uses *O*(*n*) space. Hence, in total a y-fast trie uses *O*(*n*) space.

Operations [edit]

Like van Emde Boas trees and x-fast tries, y-fast tries support the operations of an *ordered* associative array. This includes the usual associative array operations, along with two more *order* operations, *Successor* and *Predecessor*:

- *Find*(*k*): find the value associated with the given key
- *Successor*(*k*): find the key/value pair with the smallest key larger than or equal to the given key
- *Predecessor*(*k*): find the key/value pair with the largest key less than or equal to the given key
- *Insert*(*k*, *v*): insert the given key/value pair
- *Delete*(*k*): remove the key/value pair with the given key

Find [edit]

A key k can be stored in either the tree of the smallest representative r greater than k or in the tree of the predecessor of r since the representative of a binary search tree need not be an element stored in its tree. Hence, we first find the smallest representative r greater than k in the x-fast trie. Using this representative, we retrieve the predecessor of r. These two representatives point to two balanced binary search trees, which we both search for k.

Finding the smallest representative r greater than k in the x-fast trie takes $O(\log \log M)$. Using r, finding its predecessor takes constant time. Searching the two balanced binary search trees containing $O(\log M)$ elements each takes $O(\log \log M)$ time. Hence, a key k can be found, and its value retrieved, in $O(\log \log M)$ time.[1]

Successor and Predecessor [edit]

Similarly to the key k itself, its successor can be stored in either the tree of the smallest representative r greater than k or in the tree of the predecessor of r. Hence, to find the successor of a key k, we first search the x-fast trie for the smallest representative greater than k. Next, we use this representative to retrieve its predecessor in the x-fast trie. These two representatives point to two balanced binary search trees, which we search for the successor of k.[3]

Finding the smallest representative r greater than k in the x-fast trie takes $O(\log \log M)$ time and using r to find its predecessor takes constant time. Searching the two balanced binary search trees containing $O(\log M)$ elements each takes $O(\log \log M)$ time. Hence, the successor of a key k can be found, and its value retrieved, in $O(\log \log M)$ time.[1]

Searching for the predecessor of a key k is highly similar to finding its successor. We search the x-fast trie for the largest representative r smaller than k and we use r to retrieve its predecessor in the x-fast trie. Finally, we search the two balanced binary search trees of these two representatives for the predecessor of k. This takes $O(\log \log M)$ time.

Insert [edit]

To insert a new key/value pair (k, v), we first need to determine in which balanced binary search tree we need to insert k. To this end, we find the tree T containing the successor of k. Next, we insert k into T. To ensure that all balanced binary search trees contain $O(\log M)$ elements, we split T into two balanced binary trees and remove its representative from the x-fast trie if it contains more than $2 \log M$ elements. Each of the two new balanced binary search trees contains at most $\log M + 1$ elements. We pick a representative for each tree and insert these into the x-fast trie.

Finding the successor of k takes $O(\log \log M)$ time. Inserting k into a balanced binary search tree that contains $O(\log M)$ elements also takes $O(\log \log M)$ time. Splitting a binary search tree that contains $O(\log M)$ elements can be done in $O(\log \log M)$ time. Finally, inserting and deleting the three representatives takes $O(\log M)$ time. However, since we split the tree at most once every $O(\log M)$ insertions and deletions, this takes constant amortized time. Therefore, inserting a new key/value pair takes $O(\log \log M)$ amortized time.[3]

Delete [edit]

Deletions are very similar to insertions. We first find the key k in one of the balanced binary search trees and delete it from this tree T. To ensure that all balanced binary search trees contain $O(\log M)$ elements, we merge T with the balanced binary search tree of its successor or predecessor if it contains less than $(\log M)/4$ elements. The representatives of the merged trees are removed from the x-fast trie. It is possible for the merged tree to contain more than $2 \log M$ elements. If this is the case, the newly formed tree is split into two trees of about equal size. Next, we pick a new representative for each of the new trees and we insert these into the x-fast trie.

Finding the key k takes $O(\log \log M)$ time. Deleting k from a balanced binary search tree that contains $O(\log M)$ elements also takes $O(\log \log M)$ time. Merging and possibly splitting the balanced binary search trees takes $O(\log \log M)$ time. Finally, deleting the old representatives and inserting the new representatives into the x-fast trie takes $O(\log M)$ time. Merging and possibly splitting the balanced binary search tree, however, is done at most once for every $O(\log M)$ insertions and deletions. Hence, it takes constant amortized time. Therefore, deleting a key/value pair takes $O(\log \log M)$ amortized time.[3]

References [edit]

1. ^ a b c Willard, Dan E. (1983). "Log-logarithmic worst-case range queries are possible in space Θ(N)". *Information*

Processing Letters (Elsevier) **17** (2): 81–84. doi:10.1016/0020-0190(83)90075-3. ISSN 0020-0190.

2. ^ Bose, Prosenjit; Douïeb, Karim; Dujmović, Vida; Howat, John; Morin, Pat (2010), "Fast Local Searches and Updates in Bounded Universes", (PDF), Proceedings of the 22nd Canadian Conference on Computational Geometry (CCCG2010), pp. 261–264 http://cccg.ca/proceedings/2010/paper69.pdf Missing or empty |title= (help)
3. ^ *a b c* Schulz, André; Christiano, Paul (2010-03-04). "Lecture Notes from Lecture 9 of Advanced Data Structures (Spring '10, 6.851)" (PDF). Retrieved 2011-04-14.

External links [edit]

- Open Data Structure - Chapter 13 - Data Structures for Integers

v · t · e	Tree data structures
Search trees (dynamic sets/associative arrays)	2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^x · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB · Weight-balanced
Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas
Tries	Hash · Radix · Suffix · Ternary search · X-fast · **Y-fast**
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top

Categories: Trees (data structures)