Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Српски / srpski
Edit links

Tango tree

From Wikipedia, the free encyclopedia

A **Tango tree** is a type of binary search tree proposed by Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu in 2004.[1]

It is an online binary search tree that achieves an $O(\log \log n)$ competitive ratio relative to the optimal offline binary search tree, while only using $O(\log \log n)$ additional bits of memory per node. This improved upon the previous best known competitive ratio, which was $O(\log n)$.

Contents [hide]

Structure [edit]

Tango trees work by partitioning a binary search tree into a set of *preferred paths*, which are themselves stored in auxiliary trees (so the tango tree is represented as a tree of trees).

Reference Tree [edit]

To construct a tango tree, we simulate a complete binary search tree called the *reference tree*, which is simply a traditional binary search tree containing all the elements. This tree never shows up in the actual implementation, but is the conceptual basis behind the following pieces of a tango tree.

Preferred Paths [edit]

First, we define for each node its *preferred child*, which informally is the most-recently touched child by a traditional binary search tree lookup. More formally, consider a subtree *T*, rooted at *p*, with children *l* (left) and *r* (right). We set *r* as the preferred child of *p* if the most recently accessed node in *T* is in the subtree rooted at *r*, and *l* as the preferred child otherwise. Note that if the most recently accessed node of *T* is *p* itself, then *l* is the preferred child by definition.

A preferred path is defined by starting at the root and following the preferred children until reaching a leaf node. Removing the nodes on this path partitions the remainder of the tree into a number of subtrees, and we recurse on each subtree (forming a preferred path from its root, which partitions the subtree into more subtrees).

Auxiliary Trees [edit]

To represent a preferred path, we store its nodes in a balanced binary search tree, specifically a red-black tree. For each non-leaf node *n* in a preferred path *P*, it has a non-preferred child *c*, which is the root of a new auxiliary tree. We attach this other auxiliary tree's root (*c*) to *n* in *P*, thus linking the auxiliary trees together. We also augment the auxiliary tree by storing at each node the minimum and maximum depth (depth in the reference tree, that is) of nodes in the subtree under that node.

Algorithm [edit]

Searching [edit]

To search for an element in the tango tree, we simply simulate searching the reference tree. We start by searching the preferred path connected to the root, which is simulated by searching the auxiliary tree corresponding to that preferred path. If the auxiliary tree doesn't contain the desired element, the search terminates on the parent of the root of the subtree containing the desired element (the beginning of another preferred path), so we simply proceed by searching the auxiliary tree for that preferred path, and so forth.

Updating [edit]

In order to maintain the structure of the tango tree (auxiliary trees correspond to preferred paths), we must do some updating work whenever preferred children change as a result of searches. When a preferred child changes, the top part of a preferred path becomes detached from the bottom part (which becomes its own preferred path) and reattached to another preferred path (which becomes the new bottom part). In order to do this efficiently, we'll define *cut* and *join* operations on our auxiliary trees.

Join [edit]

Our *join* operation will combine two auxiliary trees as long as they have the property that the top node of one (in the reference tree) is a child of the bottom node of the other (essentially, that the corresponding preferred paths can be concatenated). This will work based on the *concatenate* operation of red-black trees, which combines two trees as long as they have the property that all elements of one are less than all elements of the other, and *split*, which does the reverse. In the reference tree, note that there exist two nodes in the top path such that a node is in the bottom path if and only if its key-value is between them. Now, to join the bottom path to the top path, we simply *split* the top path between those two nodes, then *concatenate* the two resulting auxiliary trees on either side of the bottom path's auxiliary tree, and we have our final, joined auxiliary tree.

Cut [edit]

Our *cut* operation will break a preferred path into two parts at a given node, a top part and a bottom part. More formally, it'll partition an auxiliary tree into two auxiliary trees, such that one contains all nodes at or above a certain depth in the reference tree, and the other contains all nodes below that depth. As in *join*, note that the top part has two nodes that bracket the bottom part. Thus, we can simply *split* on each of these two nodes to divide the path into three parts, then *concatenate* the two outer ones so we end up with two parts, the top and bottom, as desired.

Analysis [edit]

In order to bound the competitive ratio for tango trees, we must find a lower bound on the performance of the optimal offline tree that we use as a benchmark. Once we find an upper bound on the performance of the tango tree, we can divide them to bound the competitive ratio.

Interleave Bound [edit]

Main article: Interleave lower bound

To find a lower bound on the work done by the optimal offline binary search tree, we again use the notion of preferred children. When considering an access sequence (a sequence of searches), we keep track of how many times a reference tree node's preferred child switches. The total number of switches (summed over all nodes) gives an asymptotic lower bound on the work done by any binary search tree algorithm on the given access sequence. This is called the *interleave lower bound*.[1]

Tango Tree [edit]

In order to connect this to tango trees, we will find an upper bound on the work done by the tango tree for a given access sequence. Our upper bound will be $(k+1)O(\log \log n)$, where *k* is the number of interleaves.

The total cost is divided into two parts, searching for the element, and updating the structure of the tango tree to maintain the proper invariants (switching preferred children and re-arranging preferred paths).

Searching [edit]

To see that the searching (not updating) fits in this bound, simply note that every time an auxiliary tree search is unsuccessful and we have to move to the next auxiliary tree, that results in a preferred child switch (since the parent preferred path now switches directions to join the child preferred path). Since all auxiliary tree searches are unsuccessful except the last one (we stop once a search is successful, naturally), we search $k+1$ auxiliary trees. Each search takes $O(\log\log n)$, because an auxiliary tree's size is bounded by $\log n$, the height of the reference tree.

Updating [edit]

The update cost fits within this bound as well, because we only have to perform one *cut* and one *join* for every visited auxiliary tree. A single *cut* or *join* operation takes only a constant number of searches, *splits*, and *concatenates*, each of which takes logarithmic time in the size of the auxiliary tree, so our update cost is $(k+1)O(\log\log n)$.

Competitive Ratio [edit]

Tango trees are $O(\log\log n)$-competitive, because the work done by the optimal offline binary search tree is at least linear in *k* (the total number of preferred child switches), and the work done by the tango tree is at most $(k+1)O(\log\log n)$.

See also [edit]

- Splay tree
- Optimal binary search tree
- Red-black tree
- Tree (data structure)

References [edit]

1. ^ *a* *b* Demaine, E. D.; Harmon, D.; Iacono, J.; Pătraşcu, M. (2007). "Dynamic Optimality—Almost". *SIAM Journal on Computing* **37**: 240. doi:10.1137/S0097539705447347.

Categories: Binary trees