Multi-Splay Trees and Tango Trees

Christian Altamirano and Parker Rule

May 18, 2021

1 Introduction and literature review

This project is an exploration of two data structures developed in the study of *dynamic optimality*. Recall that a binary search tree is *dynamically optimal* if, for every access sequence X, the tree executes the sequence in $O(\text{OPT}(X))$ operations, where $\text{OPT}(X)$ is the minimum number of unit-cost operations required to access X in *any* binary search tree [Dem+07]. Splay trees [ST85] are conjectured to be dynamically optimal, but this conjecture is still unresolved after nearly four decades. However, several binary search trees have been shown to achieve $O(\log\log n \cdot OPT(X))$ performance—that is, they are $O(\log\log n)$-competitive. (Every balanced binary search tree is trivially $O(\log n)$-competitive.) Tango trees [Dem+07] and multi-splay trees [SW04; WDS06; DSW09] both achieve $O(\log\log n)$-competitiveness by encoding *preferred paths* as trees of size $O(\log n)$ within a tree-of-trees representation. We have implemented multi-splay trees and made significant progress in the implementation of tango trees.

2 Multi-splay implementation

At a high level, a multi-splay tree is a tree made of splay trees where nodes have several augmentations. Each node has a *marked* bit that tells whether or not this node is a root of its own splay tree. Splaying is done at the splay subtree level by rotating up until reaching a marked node.

Our implementation follows the C++ starter code shown in the appendix in [SW04], with a few modifications. First, in our implementation we don't include a few of the augmentations and instead compute some properties on the fly. This extra computation seems to cause at most a constant overhead for the runtime, and our experiments show that the runtime matches the asymptotic bounds for $n \leq 2^{20}$. We also make a subtle modification the `Query` recursive function to deal with nodes with infinite keys. Finally, we splay the nodes top-down rather than bottom-up—splaying nodes bottom-up as originally described in [SW04] appears to be incorrect, as top nodes would have their pointers modified improperly.

All trees implemented for this project use a common base implementation of `BST` and `BSTNode` (code). These classes provide a common interface for inserts, deletes, and queries and are generic over key, value, and metadata types.

2.1 Benchmarks: splay trees vs. multi-splay trees

For testing, we use the universe made of keys in $[0, 2^{20} - 2]$ and sample random trees containing all of these keys. (We insert elements in random order to avoid starting with a pathological splay tree.) For splay trees (code), we simply insert all the possible keys to an empty tree and then perform random queries. For multi-splay trees (code), we make a perfect binary search tree and perform random queries to modify all the preferred children. After this initialization, we test how multi-splay trees perform compared to splay trees using several query sequences (code).

For our benchmarks, we used the sequence of all keys in increasing order, a sequence of random keys, and a sequence of all keys in bit reversal order. It is known that for a bit reversal sequence X of length n, $\text{OPT}(X) = \Theta(n \log n)$ [Wil89].

Our results show that multi-splay tree queries seem to take $O(\log n)$ amortized time per query, which matches the asymptotic bounds in [SW04]. Querying the increasing sequence is much faster than a random sequence. Asymptotically, sequential accesses in splay trees run in linear time [Tar85]. Our experiments also show that for the increasing sequence query, the number of rotations needed is linear, always less than $3.8N$ for $N = 2^{20} - 1$. This matches the results in [SW04].

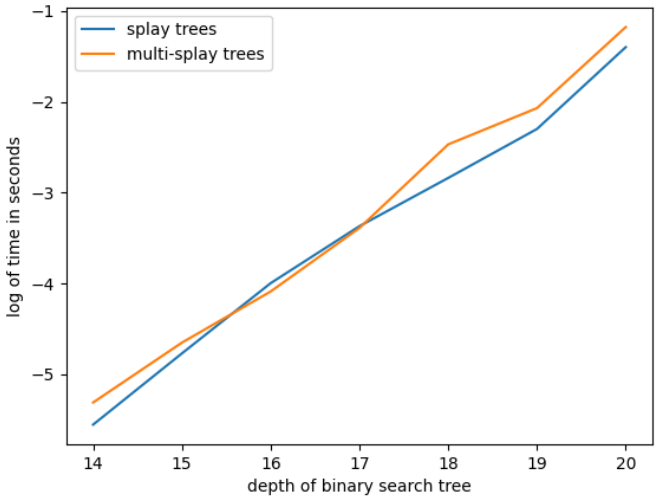

Figure 1: Increasing sequence query.

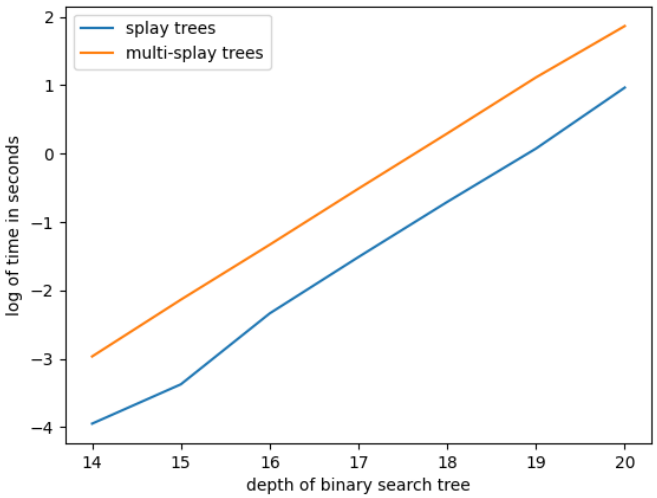

Figure 2: Random sequence query.

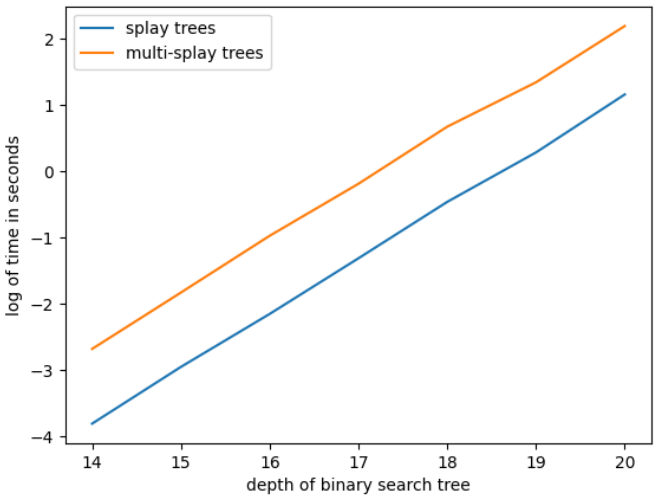

Figure 3: Bit reversal sequence query.

3 Partial tango implementation and future work

The implementation details of tango trees are significantly more subtle than those of multi-splay trees, even though the data structures are conceptually similar. In [Dem+07], an augmented red-black tree is used as the basis of the tree-of-trees representation. For standard red-black tree implementation details, including split and concatenate operations, we follow [Tar83; Sed90; Cor+09; Wei05; Wik21]. *Marked nodes* represent the roots of individual trees, and the red-black invariants are maintained separately within each of these trees, and all of the standard tree operations—such as rotations and height queries—must be updated to respect marking. In most cases, marked nodes should be treated as null pointers (though preserved in rotations), but this rule can be problematic when the main root is marked. Additionally, [Dem+07] requires nodes to be augmented with the minimum and maximum perfect depths within their subtrees, and these augmentations must be updated at each tree operation while respecting marked nodes. (Nodes also store their fixed depths in the perfect tree.) In our implementation, we reuse the base `BST` and `BSTNode` classes (code) used in the multi-splay tree implementation.

As [Dem+07] describes tango trees for a static universe of keys, we add root-level `lock()` and `unlock()` operations. In the unlocked phase, the tree supports inserts and deletes (handled lazily), but not queries; in the locked phase, the tree is converted to a tango tree and allows only queries. The tree is rebuilt into a perfect tree at each lock and unlock operation.

Unfortunately, our implementation (code) is incomplete; implementing tree operations that respect the nuances of marked nodes proved more difficult than anticipated under our time constraints. However, we believe that our work may be a useful starting point for future 6.851-ers. To this end, we list the next steps necessary to complete the implementation below.

3.1 Completed tasks

- **Base red-black tree implementation**, as described in [Cor+09].
- **Perfect tree rebuilding.** After an initial series of inserts and deletes, we lock the tree and rebuild into a perfect tree in the obvious recursive way. We update each node's metadata and record perfect depths.
- **Split and concatenate operations.** These operations are distinct from the standard split and join operations used to implement set operations in C++ in two ways: both operations must respect marked nodes, and the `concatenate()` operation is a special case of the general join operation where the trees to be joined share a root in the tree of trees.
- **Tree augmentation.** We recompute minimum and maximum depths in each subtree as necessary (this entails following parent pointers up to marked nodes).

3.2 Future work

- $P_i \rightarrow T_i$ **conversion.** One major advantage of multi-splay trees is that (up to marking) a splay tree is also a valid multi-splay tree. However, this is not the case for nontrivial tango trees—after rebuilding an arbitrary red-black tree to form a perfects tree, we must additionally "tangoize" by converting the perfect tree representation P_i to a tree-of-trees representation T_i. Of course, T_i is determined by both the elements in P_i and a preferred path through P_i. It would suffice to hardcode an initial element to access (say, the leftmost element) as the last step in `lock()`.

- **Full cut and join operations.** We have implemented a basic version of the `cut()` operation but have not thoroughly tested it due to the lack of $P_i \rightarrow T_i$ conversion. (The `join()` operation is largely symmetric.)

- **Tango algorithm.** This is simply the application of `cut()` and `join()` operations to reflect preferred path changes.

- **Thorough tests and invariant checks.** The implicit tree-of-tree representation has a number of complicated tree invariants (it typically contains many red-black trees of differing height); globally, it must correspond to a valid preferred path and be a valid binary search tree.

Beyond these implementation steps, there are a number of interesting possible extensions.[1] It would be informative to extend the benchmarks in section 2.1 to directly compare multi-splay trees and tango trees. It is known that the tango algorithm runs in $\Theta(\log n \log \log n)$ time per query for random access sequences [WDS06], but perhaps there are sequences (such as sequential or near-sequential access) for which they perform similarly in practice.

Acknowledgements

We thank George Tang and Sebastian Mendez for contributing some early scaffolding code to this project.

[1] Inspired by [Oka99], we briefly considered using a purely functional language such as Haskell, which would likely yield a more compact representation of the tree operations. However, it is not clear that the trees' asymptotic performance guarantees would remain the same in an immutable setting.

References

[Cor+09] Thomas H Cormen et al. *Introduction to Algorithms*. MIT Press, 2009.

[Dem+07] Erik D Demaine et al. "Dynamic optimality—almost". In: *SIAM Journal on Computing* 37.1 (2007), pp. 240–251. URL: http://erikdemaine.org/papers/Tango_SICOMP/.

[DSW09] Jonathan Derryberry, Daniel Sleator, and Chengwen Chris Wang. "Properties of multi-splay trees". In: (2009). URL: http://reports-archive.adm.cs.cmu.edu/anon/2009/CMU-CS-09-171.pdf.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[Sed90] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[ST85] Daniel D Sleator and Robert E Tarjan. "Self-adjusting binary search trees". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686. URL: https://dl.acm.org/doi/pdf/10.1145/3828.3835.

[SW04] Daniel D Sleator and Chengwen Chris Wang. "Dynamic optimality and multi-splay trees". In: (2004). URL: https://kilthub.cmu.edu/ndownloader/files/12095432.

[Tar83] Robert E Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[Tar85] Robert E Tarjan. "Sequential access in splay trees takes linear time". In: *Combinatorica* 5.4 (1985), pp. 367–378.

[WDS06] Chengwen Chris Wang, Jonathan Derryberry, and Daniel D Sleator. "O (log log n)-competitive dynamic binary search trees". In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Vol. 122. SIAM. 2006, p. 374. URL: https://www.cs.cmu.edu/~jonderry/multi-splay.pdf.

[Wei05] Ron Wein. *Efficient implementation of red-black trees with split and catenate operations*. Tech. rep. Tel Aviv University, 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.4875&rep=rep1&type=pdf.

[Wik21] Wikipedia contributors. *Red-black tree — Wikipedia, The Free Encyclopedia*. [Online; accessed 2021-05-19]. 2021. URL: https://en.wikipedia.org/wiki/Red-black_tree.

[Wil89] Robert Wilber. "Lower bounds for accessing binary search trees with rotations". In: *SIAM journal on Computing* 18.1 (1989), pp. 56–67. URL: https://epubs.siam.org/doi/pdf/10.1137/0218004.