3 Competitive Dynamic BSTs

In their original paper on splay trees [8], Danny Sleator and Bob Tarjan conjectured that the cost of sequence of searches in a splay tree is within a constant factor of the cost of the same sequence of searches in *any* dynamic binary search tree, even if the competing search algorithm knows the entire sequence in advance. Equivalently, in the language of online algorithms, Sleator and Tarjan conjectured that splay trees are $O(1)$-competitive against an optimal offline adversary. After more than 25 years, this ***dynamic optimality conjecture*** remains one of the most frustrating and important open problems in the field of data structures. In fact, it is not known whether any $O(1)$-competitive *offline* binary search tree exists (at least without any additional assumptions), nor whether computing the optimal offline strategy for a given access sequence is NP-hard. The depth of our ignorance is humbling.

We do have a few partial results, however.

- Blum, Chawla*, and Kalai* [1] describe a conceptually simple but computationally expensive dynamic binary search tree that is 14-competitive *if the cost of rotations is ignored*. That is, for any sequence of access, the total *search* time for their is at most 14 times the total search time for any other dynamic binary search tree, but between searches, any number of rotations may be performed for free.

- Georgakopoulos [6] defines a class of so-called *parametrically balanced* dynamic binary search trees, which includes many existing data structures such as AVL-trees and B-trees (but *not* scapegoat trees). He then proves that splay trees are $O(1)$-competitive against the optimal offline parametrically balanced tree.

- Bose, Douïeb*, and Langerman [2] consider a class of so-called *weakly bounded* skip lists. (The "weakly bounded" constraint appears to be slightly more stringent than Georgakopoulos' parametric balance condition.) Bose *et al.* describe a deterministic weakly bounded skip list that is dynamically optimal. A standard simulation of skip lists by B-trees then implies a dynamically optimal B-tree.

- Most recently, Iacono [7] presented an online dynamic binary search tree algorithm that is optimal, provided *any* online dynamic binary search tree algorithm is optimal.

One potential way to prove the dynamic optimality conjecture is to develop tighter lower bounds on the cost of the optimal offline search strategy. Several nontrivial lower bounds are currently known: the *interleave* and *alternation* bounds proved by Wilber* [10]; the *rectangle cover* bound proved by Derryberry*, Sleator, and Wang* [5]; and the SignedGreedy lower bound of Demaine, Harmon*, Iacono, Kane*, and Pătraşcu [3].

Conversely, one might approach the problem by developing new dynamic binary search trees that more closely approach the known lower bounds. Several data structures are now known to be $O(\log\log n)$-competitive, starting with the *tango trees* developed by Demaine, Harmon*, Iacono, and Pătraşcu* [4]. A slight improvement of tango trees, called *multi-splay* trees was developed by Wang*, Derryberry*, and Sleator [9]. (Every balanced binary search tree is trivially $O(\log n)$-competitive.)

3.1 Two Cost Models

Before we discuss these results in detail, let's review the rules. Without loss of generality, we assume that our binary search tree is over the set of integers $[n] = \{1, 2, \ldots, n\}$. We are given an

access sequence $X = \langle x_1, x_2, \ldots, x_m \rangle \in [n]^m$. Our search algorithm must process this sequence in order.

In the *standard model*, we search for the node with key x_i in the current tree by traversing a path from the root, and then perform some number of *arbitrary* rotations to restructure the tree. The cost of the search is the number of key comparisons (the number of nodes in each search path) plus the number of rotations.

Wilber [10] proposed an alternate *rotation-only* model. Here, each target node x_i must be rotated to the root of the search tree before the next search begins, and the total cost is just the number of rotations. We say that a sequence of rotations *executes* a search sequence X if it brings the items $x_1, x_2, \ldots, x_m$ to the root of the search tree, in that order. $OPT(X)$ is the length of the shortest sequence of rotations that executes X.

Either cost model can simulate the other with only constant overhead.

3.2 Wilber's Interleave Bound

Our first lower bound, originally defined by Wilber [10] and later simplified by Demaine *et al.* [4], is defined in terms of a fixed but arbitrary *reference tree* P over the same set $[n]$ of search keys. For each search key $y \in [n]$, we associate two contiguous subsets of search keys. The *left region* $L(y)$ consists of x and all keys that appear in the left subtree of y in P. The *right region* $R(y)$ consists of all keys that appear in the right subtree of y in P. The right region may be empty, but the left region always contains at least one element, namely x itself. The reference tree and the left and right regions of every key are fixed throughout the entire search algorithm. For each search key y, let X^y denote the subsequence of requests in $L(y) \cup R(y)$. Let $IB(X, y)$ denote the number of alternations in X^y between searches in $L(y)$ and searches in $R(y)$. Finally, let $IB(X) = \sum_y IB(X, y)$.

For example, for the access sequence $X = \langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 3, 2, 3, 8, 4, 6, 2, 7 \rangle$ and the reference tree P shown below, we have $IB(X) = 26$. Specifically, $L(7) = \{6, 7\}$, $R(7) = \{8, 9\}$, $X^7 = \langle 9_R, 6_L, 8_R, 9_R, 7_L, 8_R, 6_L, 7_L \rangle$, and $IB(X, 7) = 5$.

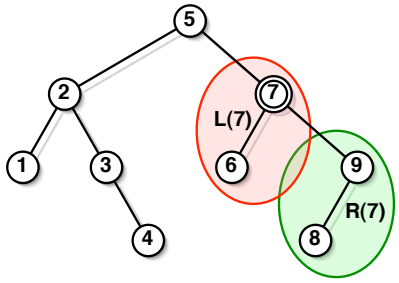

Figure 1. A reference tree with nine nodes, showing the left and right regions for node 7.

Interleave Bound [10, 4]. *In the standard model,* $OPT(X) \geq \frac{1}{2} IB(X) - n$*, for any access sequence* X *and any reference tree* P*.*

Proof (Sketch): Let T_i denote our dynamic search tree immediately before the ith search. For any key $y \in [n]$, define the *transition point* $tp_i(y)$ to be the shallowest node in T_i whose ancestors include a node in $L(y)$ and a node in $R(y)$. If the transition point exists, it can be uniquely determined as follows. Let $\ell_i(y)$ be the least common ancestor in T_i of the nodes in $L(y)$, and let $r_i(y)$ be the least common ancestor in T_i of the nodes in $R(y)$. Because $L(y)$ and $R(y)$

are contiguous intervals, one of these two nodes must be a proper ancestor of the other. The transition point is the *deeper* of these two nodes. (If $R(y)$ is empty, then the transition point is undefined, but in that case $IB(X, y) = 0$.)

Suppose the search algorithm does not touch $tp_i(y)$ during the ith search. Without loss of generality, assume that $tp_i(y) = r_i(y)$. Then the search algorithm does not touch any node in $R(y)$, so $r_{i+1}(y) = r_i(y)$. The node $\ell_i(y)$ must be outside the subtree of T_{i+1} rooted at $r_{i+1}(y)$, and so $\ell_{i+1}(y)$ is the least common ancestor of $\ell_i(y)$ and $r_{i+1}(y)$ in T_{i+1}. It follows that $tp_{i+1}(y) = r_{i+1}(y) = r_i(y) = tp_i(y)$. We conclude the transition point can only change if the search algorithm touches it.

A straightforward but boring case analysis, which I won't repeat here, implies that distinct keys have different transition points.

Now consider the cost of the optimal search tree for access sequence X. We easily observe[1] that

$$\begin{aligned}
OPT(X) &\geq \sum_i \#\text{nodes touched by } i\text{th search in the optimal search tree} \\
&\geq \sum_i \#\text{transition points touched by } i\text{th search in the optimal search tree} \\
&= \sum_y \sum_i [tp_i(y) \text{ touched by } i\text{th search in the optimal search tree}] \\
&\geq \sum_i \sum_y [tp_i(y) \neq tp_{i+1}(y) \text{ in the optimal search tree}].
\end{aligned}$$

Fix an integer y, and consider a maximal subsequence $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_p} \rangle$ of X that alternates between keys in $L(y)$ and keys in $R(y)$. By definition, $IP(X, y) = p - 1$. Consider the contiguous subsequence of searches $\langle x_{i_{2j-1}}, \ldots, x_{i_{2j}} \rangle$ for some j. Without loss of generality, the (i_{2j-1})th search touches $\ell(y)$, and the (i_{2j-1})th search touches $r(y)$. If neither of these searches touches $tp(y)$, then $tp(y)$ must be changes (and therefore touched) by some intermediate search. Thus, $tp(y)$ must be touched at least $\lfloor p/2 \rfloor = \lfloor (IB(X, y) + 1)/2 \rfloor \geq IB(X, y)/2$ times. We conclude that

$$OPT(x) \geq \sum_y \frac{IB(X, y)}{2} = \frac{IB(X)}{2}. \qquad \square$$

3.3 Tango and Multisplay Trees

Like the alternation bound, tango trees are defined in terms of a fixed, *perfectly-balanced* reference tree P. (It is possible to implement the data structure without actually constructing or maintaining P, but the description is somewhat simpler if we imagine actually maintaining it.) Left and right regions are defined in terms of this fixed subtree, just as in the definition of Wilber's interleave bound. Every node v in P has a *preferred child*, indicating whether $L(v)$ or $R(v)$ contains the most recent search target in $L(v) \cup R(v)$. (If no node in $L(v) \cup R(v)$ has been accessed yet, the preferred child of v is undefined.) A *preferred path* is a maximal path in which every node except the first is the preferred child of its parent. Because P is perfectly balanced, each preferred path contains $O(\log n)$ nodes.

The actual tango tree T consists of several balanced binary search trees, one for each preferred path in P, joined together into a single master binary search tree. The choice of component BST

[1] I'm using Iverson's bracket notation: $[A] = 1$ if proposition A is true, and $[A] = 0$ if A is false.

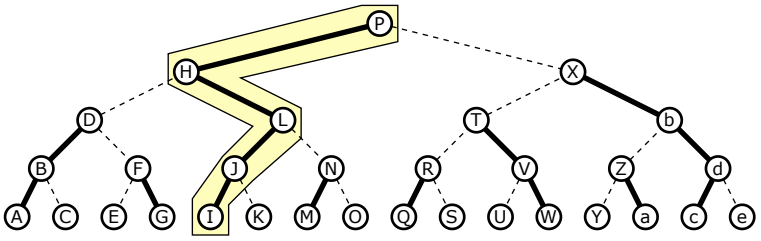

Figure 2. A 31-node reference tree. Bold edges indicate preferred children. One preferred path is highlighted.

is fairly arbitrary—the original formulation of Demaine *et al.* [4] uses red-black trees; the more recent multi-splay trees of Wang *et al.* [9] use splay trees—provided it supports the following operations on an m-node tree in $O(\log m)$ amortized time:

- SEARCH(T, x): Search for key x in T.
- SPLIT(T, x): Split T into two search trees, one containing all keys less than or equal to x, the other containing all keys greater than x.
- MERGE$(T_{\leq}, T_{>})$: Merge two search trees, where one tree's keys are all smaller than the other's, into a single search tree.

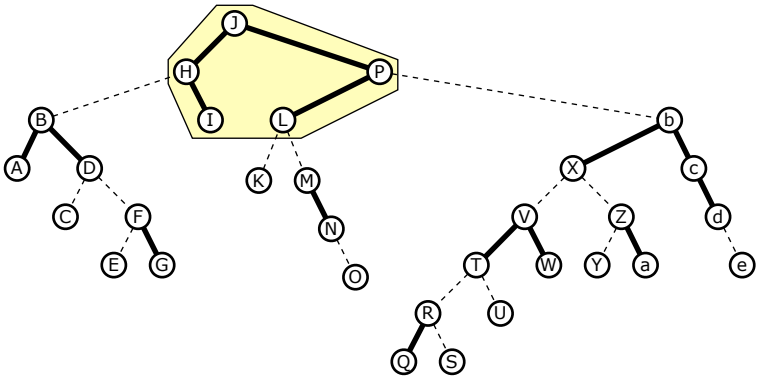

Figure 3. A tango tree corresponding to the reference tree in Figure 2, with one component tree highlighted.

Suppose the search path to x_i in P intersects k preferred paths, or equivalently, visits $k-1$ *non*-preferred children. (For example, for the reference tree shown in Figure 2, if $x_i = \mathsf{C}$ then $k = 3$.) To find x_i in the actual tango tree T_i, we apply the standard binary search tree algorithm, but for purposes of analysis, we can partition the search into k SEARCHes in component trees. Since each component tree has at most $\lceil \lg n \rceil$ nodes, the total (amortized) search time is $O(k \log \log n)$.

After each search, the search path in P becomes a preferred path, so we must restructure the corresponding components of the tango tree. To this end, we must be able to split any preferred path into two paths, and to merge two preferred paths where the deepest node in one path is the parent of the shallowest node in the other. If we examine the nodes in any preferred path in sorted order, we find that the nodes in any suffix (that is, any lower subpath) form a contiguous block. Thus, we can simulate splitting a preferred path by SPLITting the corresponding component tree twice. Similarly, we can model merging two paths by one call to SPLIT and two calls to MERGE

to combine two component trees. Each SPLIT and MERGE takes $O(\log\log n)$ time, and we must preform $O(k)$ of them. Thus, the total time for restructuring the tango tree is $O(k \log\log n)$.

We conclude that a search that changes k preferred child pointers in the reference tree P can be executed in $O(k \log\log n)$ time in the tango tree T. But the total number of changes to preferred children is precisely Wilber's interleave bound! It follows that tango trees are $O(\log\log n)$-competitive with the best dynamic binary search tree.

Unfortunately, as Wang *et al.* [9] observed that, the worst–case time for searching in a tango tree is $O(\log n \log n \log n)$, not $O(\log n)$ as for other balanced binary search trees. To recover the optimal search time, at least in an amortized sense, they replaced the red-black component trees with splay trees. Wang *et al.* prove that multisplay trees enjoy several other properties of splay trees, including working se and sequential access lemmas. Unlike tango trees, multisplay trees also support insertions and deletions while remaining $O(\log\log n)$-competitive.

3.4 Wilber's Working-Set Alternation Bound

Wilber's second lower bound is a variation of the working set bound. Let x_i and x_j be two consecutive searches for the same key. Recall that the *working set* for x_j is the set of unique keys in the range $\{x_{i+1}, \ldots, x_{j-1}\}$. Imagine incrementally constructing an *unbalanced* binary search tree by inserting the keys $x_{i+1}, x_{i+2}, \ldots, x_{j-1}, x_j$ in that order (ignoring duplicates) into an initially empty tree. Let $WA(X, j)$ denote the number of turns—the number of alternations between left and right children—in the search path to x_j in this tree. If x_j is the first occurrence of a key in X, then $WA(X, j) = 0$. Finally, we define $WA(X) = \sum_{j=1}^{M} WA(X, j)$.

We can equivalently define $WA(X, j)$ as follows. Imagine computing the largest open interval on the real line that contains the search key x_j but no element of its working set. We start with the entire real line $(-\infty, \infty)$. As we scan through the working set in order, the interval occasionally shrinks, sometimes from below, sometimes from above. $WA(X, j)$ is the number of alternations between increasing the lower bound and decreasing the upper bound.

For example, for the access sequence $X = \langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7\rangle$, we have $WA(X) = 11$. Specifically, for the pair $x_{14} = x_{23} = 7$, we obtain the search tree shown below, or the nested sequence of intervals $(-\infty, \infty) \to (-\infty, \mathbf{9}) \to (\mathbf{3}, 9) \to (3, \mathbf{8}) \to (\mathbf{4}, 8) \to (6, 8)$, implying that $WA(X, 22) = 3$.

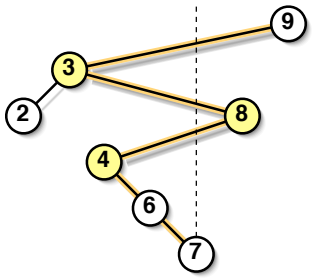

Figure 4. The working-set alternation tree for $\langle 9, 3, 2, 3, 8, 4, 6, 2, 7\rangle$.

Working-Set Alternation Bound [10]. *$OPT(X) = \Omega(WA(X))$ for any access sequence X.*

3.5 The Geometric View

3.6 Independent Rectangles

3.7 The Rectangle Cover Bound

For the final lower bound, we interpret our given access sequence X as a set of points in the plane, where the x-axis represents *rank* and the y-axis represents *time*. Specifically, the ith request in X corresponds to the point (x_i, i). At the risk of confusing the reader, I will also call this point set X.

A *box* is an axis-aligned rectangle with points of X in two opposite corners. Boxes are either *left-handed* or *right-handed*, depending on which *bottom* corner contains a point. A left-handed box has a point from X in its bottom-*left* and top-right corners; a right-handed box has points in the top-left and bottom-*right* corners. Each box contains a *divider*—a vertical line segment that splits it into two smaller rectangles at some non-integral x-coordinate. We say that two boxes *conflict* if they have the same handedness and each box intersects the other box's divider. Finally, let $BO(X)$ ('box overlap of X'?) be the maximum number of pairwise non-conflicting boxes for a given access sequence X.

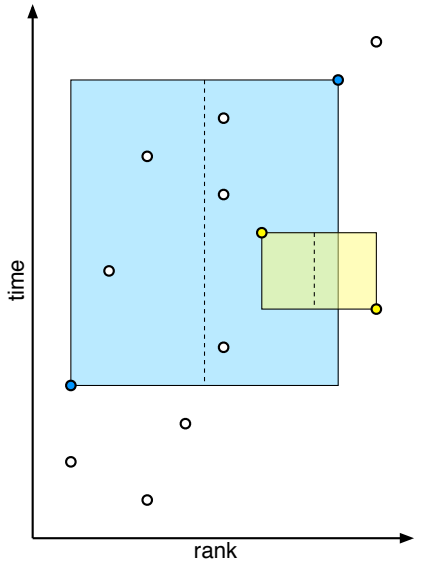

Figure 5. Points corresponding to the access sequence $\langle 3,1,4,1,5,9,2,6,5,3,5,8,9\rangle$. A large right-handed box and a small left-handed box are shaded.

Rectangle Cover Bound [5]. *$OPT(X) = \Omega(BO(X))$ for any access sequence X.*

Proof: We work in Wilber's rotation-only model, in which every access must bring the requested item to the root of the search tree. Let B be any set of pairwise non-conflicting boxes for X, and let R be any sequence of rotations that executes the access sequence X. For each box in B, we will associate a unique rotation from R, thereby proving that $|R| \geq |B|$. In particular, this inequality holds for the *largest* set of boxes and the *smallest* set of rotations, so $OPT(X) \geq BO(X)$.

Let $\text{lca}(a, b)$ denote the least common ancestor of two nodes a and b in the current search tree, where $a < b$. We must have $a \leq \text{lca}(a, b) \leq b$. Let c be any node in the tree, and let p be its parent. A rotation at c makes c the new parent of p. It is not hard to see that this rotation changes $\text{lca}(a, b)$ if and only if the old lca was p and the new lca is c.

We can visualize each rotation in R as a horizontal line segment, whose x-coordinates represent the rotated node and its (old) parent, and whose y-coordinate is the 'time' of the rotation. The exact positions are unimportant, as long as the segments do not overlap and all rotations needed to execute the ith search have y-coordinates in the open interval $(i-1, i)$.

Now consider a left-handed box $\square_k \in B$ with corner requests (a, i) and (b, j), where $i < j$ and (therefore) $a < b$. After the ith search, a is at the root of the search tree, so $\text{lca}(a, b) = a$. Similarly, after the jth search, b is at the root, so $\text{lca}(a, b) = b$. Thus, between those two searches, there must be a *right* rotation that moves $\text{lca}(a, b)$ across the divider of $\square_k$, from the left side to the right side. Let r_k be the earliest such rotation. The segment corresponding to r_k lies entirely within $\square_k$.

If two left-handed boxes $\square$ and $\square'$ claim the same right rotation, the intersection $\square \cap \square'$ must intersect both dividers, so the boxes would conflict. Thus, every left-handed box in B claims a unique right rotation in R. Similarly, every right-handed box in B claims a unique left rotation in R. We conclude that $|R| \geq |B|$, as promised. $\square$

Derryberry *et al.* proved that the rectangle cover bound implies both Wilber bounds.

References

[1] A. Blum, S. Chawla*, and A. Kalai*. Static optimality and dynamic search-optimality in lists and trees. *Algorithmica* 36(3):249–260, 2003.

[2] P. Bose, K. Douïeb*, and S. Langerman. Dynamic optimality for skip lists and B-trees. *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms*, 1106–1114, 2008.

[3] E. D. Demaine, D. Harmon*, J. Iacono, D. Kane*, and M. Pătraşcu. The geometry of binary search trees. *Proc. 20th Ann. ACM-SIAM Symp. Discrete Algorithms*, 496–505, 2009.

[4] E. D. Demaine, D. Harmon*, J. Iacono, and M. Pătraşcu**. Dynamic optimality—almost. *Proc. 45th Annu. IEEE Sympos. Foundations Comput. Sci.*, 484–490, 2004.

[5] J. Derryberry*, D. D. Sleator, and C. C. Wang*. A lower bound framework for binary search trees with rotations. Tech. Rep. CMU-CS-05-187, Carnegie Mellon Univ., Nov. 2005. ⟨http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-187.pdf⟩.

[6] G. F. Georgakopoulos. Splay trees: A reweighing lemma and a proof of competitiveness vs. dynamic balanced trees. *J. Algorithms* 51(1):64–76, 2004.

[7] J. Iacono. In pursuit of the dynamic optimality conjecture. *Space-Efficient Data Structures, Streams, and Algorithms*, 236–250, 2013. Lecture Notes Comput. Sci. 8066, Springer-Verlag.

[8] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985.

[9] C. C. Wang*, J. Derryberry*, and D. D. Sleator. $O(\log \log n)$-competitive dynamic binary search trees. *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 374–383, 2006.

[10] R. E. Wilber*. Lower bounds for accessing binary search trees with rotations. *SIAM J. Computing* 18(1):56–67, 1987.

*Starred authors were graduate students at the time that the cited work was published. **Double-starred authors were undergraduates.