# Combining Binary Search Trees*

Erik D. Demaine[1], John Iacono[2,**],
Stefan Langerman[3,***], and Özgür Özkan[2,†]

[1] Massachusetts Institute of Technology
[2] Polytechnic Institute of New York University
[3] Université Libre de Bruxelles

**Abstract.** We present a general transformation for combining a constant number of binary search tree data structures (BSTs) into a single BST whose running time is within a constant factor of the minimum of any "well-behaved" bound on the running time of the given BSTs, for any online access sequence. (A BST has a *well-behaved* bound with $f(n)$ overhead if it spends at most $\mathcal{O}(f(n))$ time per access and its bound satisfies a weak sense of closure under subsequences.) In particular, we obtain a BST data structure that is $\mathcal{O}(\log\log n)$ competitive, satisfies the working set bound (and thus satisfies the static finger bound and the static optimality bound), satisfies the dynamic finger bound, satisfies the unified bound with an additive $\mathcal{O}(\log\log n)$ factor, and performs each access in worst-case $\mathcal{O}(\log n)$ time.

## 1 Introduction

Binary search trees (BSTs) are one of the most fundamental and well-studied data structures in computer science. Yet, many fundamental questions about their performance remain open. While information theory dictates the worst-case running time of a single access in an $n$ node BST to be $\Omega(\log n)$, which is achieved by many BSTs (e.g., [2]), BSTs are generally not built to execute a single access, and there is a long line of research attempting to minimize the overall running time of executing an online access sequence. This line of work was initiated by Allen and Munro [1], and then by Sleator and Tarjan [15] who invented the splay tree. Central to splay trees and many of the data structures in the subsequent literature is the BST model. The BST model provides a precise model of computation, which is not only essential for comparing different BSTs, but also allows the obtaining of lower bounds on the optimal offline BST.

In the BST model, the elements of a totally ordered set are stored in the nodes of a binary tree and a BST data structure is allowed at unit cost to manipulate the tree by following the parent, left-child, or right-child pointers at each node or rotate the node with its parent. We give a formal description of

* See [9] for the full version.

** Research supported by NSF Grant CCF-1018370.

*** Directeur de Recherches du F.R.S.-FNRS. Research partly supported by the F.R.S.-FNRS and DIMACS.

† Research supported by GAANN Grant P200A090157 from the US Department of Education.

F.V. Fomin et al. (Eds.): ICALP 2013, Part I, LNCS 7965, pp. 388–399, 2013.

the model in Section 1.1. A common theme in the literature since the invention of splay trees concerns proving various bounds on the running time of splay trees and other BST data structures [4, 6, 7, 12, 15]. Sleator and Tarjan [15] proved a number of upper bounds on the performance of splay trees. The *static optimality* bound requires that any access sequence is executed within a constant factor of the time it would take to execute it on the best static tree for that sequence. The *static finger* bound requires that each access $x$ is executed in $\mathcal{O}(\log d(f, x))$ amortized time where $d(f, x)$ is the number of keys between any fixed finger $f$ and $x$. The *working set* bound requires that each access $x$ is executed in $\mathcal{O}(\log w(x))$ amortized time where $w(x)$ is the number of elements accessed since the last access to $x$. Cole [6] and Cole et al. [7] later proved that splay trees also have the *dynamic finger* bound which requires that each access $x$ is executed in $\mathcal{O}(\log d(y, x))$ amortized time where $y$ is the previous item in the access sequence. Iacono [14] introduced the *unified* bound, which generalizes and implies both the dynamic finger and working set bounds. Bose et al. [4] presented layered working set trees, and showed how to achieve the unified bound with an additive cost of $\mathcal{O}(\log \log n)$ per access, by combining them with the skip-splay trees of Derryberry and Sleator [12].

A BST data structure satisfies the dynamic optimality bound if it is $\mathcal{O}(1)$-competitive with respect to the best offline BST data structure. Dynamic optimality implies all other bounds of BSTs. The existence of a dynamically optimal BST data structure is a major open problem. While splay trees were conjectured by Sleator and Tarjan to be dynamically optimal, despite decades of research, there were no online BSTs known to be $o(\log n)$-competitive until Demaine et al. invented Tango trees [8] which are $\mathcal{O}(\log \log n)$-competitive. Later, Wang et al. [17] presented a variant of Tango trees, called multi-splay trees, which are also $\mathcal{O}(\log \log n)$-competitive and retain some bounds of splay trees. Bose et al. [3] gave a transformation where given any BST whose amortized running time per access is $\mathcal{O}(\log n)$, they show how to deamortize it to obtain $\mathcal{O}(\log n)$ worst-case running time per access while preserving its original bounds.

**Results and Implications.** In this paper we present a structural tool to combine bounds of BSTs from a certain general class of BST bounds, which we refer to as well-behaved bounds. Specifically, our method can be used to produce an online BST data structure which combines well-behaved bounds of all known BST data structures. In particular, we obtain a BST data structure that is $\mathcal{O}(\log \log n)$ competitive, satisfies the working set bound (and thus satisfies the static finger bound and the static optimality bound), satisfies the dynamic finger bound, satisfies the unified bound with an additive $\mathcal{O}(\log \log n)$, and performs each access in worst-case $\mathcal{O}(\log n)$ time. Moreover, we can add to this list any well-behaved bound realized by a BST data structure.

Note that requiring the data structures our method produces to be in the BST model precludes the possibility of a trivial solution such as running all data structures in parallel and picking the fastest.

Our result has a number of implications. First, it could be interpreted as a weak optimality result where our method produces a BST data structure which is $\mathcal{O}(1)$-competitive with respect to a constant number of given BST data structures whose actual running times are well-behaved. In comparison, a dynamically optimal BST data structure, if one exists, would be $\mathcal{O}(1)$-competitive with