Belga B-Trees

Belga B-trees*

Erik D. Demaine† John Iacono‡ Grigorios Koumoutsos§ Stefan Langerman¶

Abstract

We revisit self-adjusting external memory tree data structures, which combine the optimal (and practical) worst-case I/O performances of B-trees, while adapting to the online distribution of queries. Our approach is analogous to undergoing efforts in the BST model, where *Tango Trees* (Demaine *et al.* 2007) were shown to be $O(\log\log N)$-competitive with the runtime of the best offline binary search tree on every sequence of searches. Here we formalize the B-Tree model as a natural generalization of the BST model. We prove lower bounds for the B-Tree model, and introduce a B-Tree model data structure, the Belga B-tree, that executes any sequence of searches within a $O(\log\log N)$ factor of the best offline B-tree model algorithm, provided $B = \log^{O(1)} N$. We also show how to transform any static BST into a static B-tree which is faster by a $\Theta(\log B)$ factor; the transformation is randomized and we show that randomization is necessary to obtain any significant speedup.

1 Introduction

Worst-case analysis does not capture the fact that some sequences of operations on data structures, often typical ones, can be executed significantly faster than worst case ones. Methods of analyzing algorithms whose performance depends on more fine-grained characteristics of the input sequence other than the size N have been coined *distribution sensitive data structures* [Iac01b, BHM13]. Two general methods to bound the performance of such a data structure exist. The first is to explicitly bound the performance by some bound. For binary search trees (BSTs) there is a rich set of such bounds (see e.g. [EFI13, CGK+16]) like the sequential access bound [Tar85], the working set bound [ST85b, Iac01a], the (weighted) dynamic finger bound [CMSS00, Col00, IL16], the unified bound [BCDI07, Iac01a] and many others [BDIL16, HIM13, CGK+18]. The other method is to compare the performance of the data structure on a sequence of operations to the performance of the best offline data structure in some model on the same sequence. Such an analysis uses the language of competitive analysis introduced in [ST85a], where the competitive ratio of an algorithm is the supremum ratio of the performance of the given algorithm to the offline optimal over all sequences of operations over a given length. A data structure which is $O(1)$-competitive in a particular model is said to be *dynamically optimal* [ST85b]. In the BST model, the best known competitive ratio is $O(\log\log N)$, first achieved by Tango trees [DHIP07]. The existence of a dynamically optimal BST is one of the most intriguing and long-standing open problems in online algorithms and data structures (see [Iac13] for a survey). The two prominent candidates to achieve dynamic optimality for BSTs are the *splay tree* of Sleator and Tarjan [ST85b] and the *greedy* algorithm [DHI+09, Luc88], but they are only known to be $O(\log N)$-competitive.

*This work was supported by the Fonds de la Recherche Scientifique-FNRS under Grant no MISU F 6001 1 and by NSF Grant CCF-1533564.

†CSAIL, Massachusetts Institute of Technology. edemaine@mit.edu

‡Université Libre de Bruxelles and New York University. johniacono@gmail.com

§Université Libre de Bruxelles. greg.koumoutsos@gmail.com

¶Directeur de Recherches du F.R.S-FNRS. stefan.langerman@ulb.ac.be

Disk-Access Model (DAM). The *external memory model*, or *disk-access model (DAM)* [AV88] is the leading way to theoretically model the performance of algorithms that can not fit all of their data in RAM, and thus must store it on a slower storage system historically known as *disk*. This model is parameterized by values M and B; the disk is partitioned into blocks of size B, of which M/B can be stored in memory at any given moment. The cost in the DAM is the number of transfers between memory and disk, called Input-Output operations (I/Os). The classic data structure for a comparison based dictionary in the DAM model, as well as in practice, is the B-Tree [BM72]. The B-Tree is a generalization of the BST, where each node stores up to $B-1$ data items, for $B \geq 2$, and the number of children is one more than the number of data items. The B-Tree supports searches in time $O(\log_B N)$ in the DAM, a $\log B$ factor faster than traditional BSTs such as red-black trees [GS78] or AVL trees [AVL62].

Dynamic Dictionaries in the DAM. Here, our goal is to explore dynamic dictionaries in the DAM and to obtain results similar to those known for BSTs.

Surprisingly, prior work in this direction is quite limited. One previous attempt was in the work of Sherk [She95] where a generalization of splay trees to what we call the B-tree model was proposed, but without any strong results. Over ten years later, Bose et. al. [BDL08] studied a self-adjusting version of skip-lists and B-Trees, where nodes can be split and merged to adapt to the query distribution by moving elements closer or farther from the root of the tree (here we call this model *classic self-adjusting* B-trees, see Section 2). They showed that dynamic optimality in this model is closely related to the working set bound. This bound captures temporal locality: for an access sequence $X = x_1, \ldots, x_m$, it is defined as $\mathrm{WS}(X) = \sum_{i=1}^{m} \log w_X(i)$, where $w_X(i)$ is the number of distinct elements accessed since the last access to the element x_i. In [BDL08] the authors presented a data structure whose cost is upper bounded by $O(\mathrm{WS}(X)/\log B)$ and obtained a matching lower bound of $\Omega(\mathrm{WS}(X)/\log B)$ for this model, which implies that their structure is dynamically optimal.

Note that the lower bound of [BDL08] shows a major limitation of B-trees with only split and merge operations: It implies there are sequences on which they are slower than BSTs. For example, repeatedly sequentially accessing all data items $1, 2, \ldots, N$ requires $O(1)$ amortized time per search for BSTs like splay trees (this is the *sequential access bound* [Tar85]) while the lower bound $\Omega(\mathrm{WS}(X)/\log B)$ implies an amortized cost $\Omega(\log_B N)$ in the classic self-adjusting model. In this work, we show that by adding just one more operation, an analogue of the rotation for B-Trees, we can overcome this limitation and obtain significant speedups with respect to standard B-trees.

Our Contribution. In this work we initiate a systematic study of dynamic B-trees. First, we formally define the (dynamic) B-Tree model of computation (§2). Second, we show how to produce lower bounds in the B-Tree model (§3). Then, we introduce a data structure, which we call the *Belga B-Tree*[1], which is $O(\log \log N)$ competitive with any dictionary in the B-Tree model of computation, when $B = O(\log^{O(1)} N)$ (§4).

More generally, we conjecture the following in §6: any BST-model algorithm can be transformed into a (randomized) B-Tree model algorithm with a $\Theta(\log B)$ factor cost savings. This would imply that BST model algorithms such as the splay tree [ST85b] or greedy [DHI+09, Luc88] would have B-Tree model counterparts, and that a dynamically optimal BST-model algorithm would imply a dynamically optimal algorithm in the B-Tree model. We leave this conjecture open, but in §5 we do resolve the case of a static (no rotations allowed) BSTs by

[1]The Tango tree was invented on an overnight flight from JFK airport en route to Buenos Aires, Argentina. The work on the Belga B-Tree has been substantially completed at Cafe Belga, Ixelles, Belgium.

showing a randomized transformation from a static BST to a static B-Tree such that any algorithm in the static BST model would have factor $\Theta(\log B)$ speedup in the B-Tree model. We also show that no $\omega(1)$-factor speedup is possible for a deterministic transformation in general.

2 The B-Tree model of computation

In this section, we define the tree models discussed in this paper. In all cases, we consider data structures supporting searches over a universe of N elements $\mathcal{U} = \{1, 2, \ldots, N\}$ which we refer to as *keys*. The input is a valid tree T_0 and request sequence of searches $X = x_1, x_2 \ldots, x_m$, where $x_i \in \mathcal{U}$ is the ith item to be searched.

2.1 The BST Model

In a Binary Search Tree (BST) data structure, each node stores a single key and three pointers, indicating its parent and its (left and right) children. The key value of a node is larger than all keys in its left subtree and smaller than all keys in its right subtree. To execute each request to search for element x_i, a BST algorithm initializes a single pointer at the root (at unit cost) then may perform any sequence of the following unit-cost operations:

- Move the pointer to the parent or to the left or right child of the current node the pointer points to (if such a destination node exists).
- Perform a *rotation* of the edge between current node and its parent (if not the root).

Whenever the pointer moves to or it is initialized to a node v, we say that node v is *touched*. A BST-model search algorithm is correct if during each search, the element x_i that is being searched for is touched. The cost of a BST algorithm on the search sequence X equals the total number of unit-cost operations performed to execute the searches in the sequence. This model was formally defined in [DHIP07] and it is known to be equivalent up to constant factors to several alternative models which have been considered (e.g. [DHI$^+$09, Wil89]).

A BST data structure can be augmented such that each node stores $O(\log N)$ additional bits of information. The running time of such BST data structures in the RAM model is dominated by the number of unit-cost operations. A *static* BST is a restricted version of the BST model where rotations are not allowed and thus the shape of the tree never changes.

2.2 The B-tree model

We define the B-tree model to be a generalization of the BST model which allows more than one key to be stored in each node. The B-tree model is parameterized by a positive integer $B \geq 2$ which represents the maximum number of children of each node[2]; in the case where $B = 2$ the B-tree model will be equivalent to the BST model. We denote by $n(v)$ the number of keys stored in a node v. Every node v has $n(v) \leq B - 1$ and $n(v) + 1$ child pointers (some of which could be null). A node v which stores exactly $n(v) = B - 1$ keys is called *full*.

Suppose $x_1, \ldots, x_{n(v)}$ are the keys stored at node v and $c_1, \cdots, c_{n(v)+1}$ are the children of v. Keys satisfy the in-order condition, i.e. $x_1 < \ldots < x_{n(v)}$ and for any key k_i stored in the subtree T_{c_i} rooted at c_i, we have that $k_1 < x_1 < \cdots < k_i < x_i < k_{i+1} < \cdots < k_{n(v)} < x_{n(v)} < k_{n(v)+1}$.

Similar to the BST model, to execute each search there is a single pointer initialized to the root of the tree at unit cost. To execute a search for x_i, a B-tree algorithm performs a sequence of the following unit-cost operations which are described formally later:

[2]Recall that in the external memory model (defined in Section 1) B denotes the block size. Each B-tree node has at most B children, contains $O(B)$ words and thus it can be stored in $O(1)$ blocks of size B.

- Move the pointer to a child or to the parent of the current node.
- Split a node containing at least three keys.
- Join two sibling nodes storing no more than $B-2$ keys in total.
- Rotate the edge between the current node and its parent.

B-tree model algorithms that only use the first type of operations are referred to as *static* as the shape of the B-tree does not change. We now fully describe the unit-cost operations of rotating, splitting and joining:

Rotations: Consider a (non-root) node u and let $p(u)$ be its parent. Let $P = \{p_1, \ldots, p_m\}$ be the union of all keys stored in u and $p(u)$. The keys stored at u define an interval $[p_\ell, p_r]$ in P. A rotation of the edge $(p(u), u)$ essentially updates this interval to $[p_{\ell'}, p_{r'}]$, moving the keys as needed. Depending on the values of ℓ, ℓ' and r, r' we characterize a rotation as a promote/demote left — promote/demote right rotation. For example, a rotation of the type promote left k — demote right k' sets $\ell' = \ell + k$ (i.e. the k leftmost keys of u are promoted to $p(u)$) and $r' = r + k'$ (i.e. keys $p_{r+1}, \ldots, p_{r+k'}$ are demoted to u). Values k and k' should be non-negative and satisfy that after the rotation both u and $p(u)$ have at most $B-1$ keys. Rotations of the type demote left — promote right, promote left — promote right and demote left — demote right can be defined analogously. As an example, figure 1 shows a rotation of type demote left - promote right.

Splitting a node: Let u be a node (except the root) containing at least three keys and let $p(u)$ be its non-full parent. Splitting node u at key u_m (which is not the smallest or the largest key stored at u) consists of promoting u_m to $p(u)$ and replacing u by 2 nodes u_L, u_R such that keys smaller than u_m are contained in u_L and keys larger than u_m are in u_R. To split the root (given that it stores at least three keys), we create an empty B-tree node, make it the parent of the root (i.e. the new root) and then perform a split operation as defined above.

Join: This operation is the inverse of a split. Let u and v be two sibling nodes and let p be their parent, such that there exists a unique key p_j in p such that p_j is larger than all keys stored at u and smaller than all keys stored at v. Joining nodes u and v (given that they store no more than $B-2$ keys in total) consists of demoting p_j to u (and deleting it from p), adding all elements of v (including the pointers to children) to u and deleting v. Note that after a join operation p might become empty (in case p_j was the unique key of p). In that case, we set the parent of u to be the parent of p (if it exists) and we delete p. If p is empty and it is the root, then we just delete p and u becomes the new root of the tree.

A B-tree can be augmented with additional $O(B \log N)$ bits of information for each node. The performance of B-trees in the *external memory model* with blocks of size B, is within a constant factor of the sum of the unit-cost operations as we have defined them.

Relation with other B-tree models. The classic structure of B-trees first appeared in [BM72]. In this framework, all leaves have the same depth and no join, split and rotate operations are performed during searches (to be precise, restricted versions of split and join were defined in order to support insertions and deletions and were not allowed for performing search operations, see [CLRS09] for an extensive treatment). We call this framework the *classic B-tree* model.

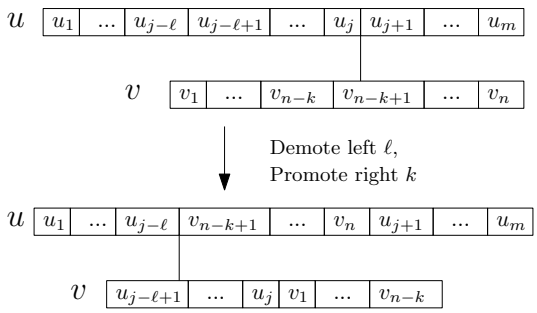

Figure 1: A rotation of a B-tree edge (u, v) of the type demote left ℓ — promote right k: From the left of v, the ℓ neighboring keys of u, $u_{j-\ell+1}, \ldots, u_j$ are getting demoted to v. From the right, the k last elements of v, $v_{n-k+1}, \ldots, v_n$ are getting promoted to u.

A more flexible model of B-trees was considered in [BDL08]: We start with a classic B-tree and an algorithm is allowed to perform joins and splits, but not rotations. Note that by performing join and split operations, the property that all leaves of the tree have the same depth is maintained throughout the whole execution. This model was called "self-adjusting B-trees". To avoid confusion with our dynamic B-tree model, we call this model *classic self-adjusting* B-trees, in order to emphasize that all leaves have the same depth, as in classic B-trees. The self-adjustment relies on the fact that using joins and splits the algorithm might choose to bring an item closer to the root or demote it farther from the root. Also, note that the number of nodes in a B-tree on N keys is not fixed (as opposed to BSTs where we always have exactly N nodes) and the split/join operations might increase/decrease the number of nodes of the tree, changing thus its shape.

For the rest of this paper, whenever we use the term B-tree we refer to our B-tree model, unless stated otherwise.

3 Lower bounds: simulating dynamic B-trees using BSTs.

In this section we show how to simulate a dynamic B-tree algorithm using a BST-model algorithm with an $O(\log B)$ overhead in the cost. This will allow us to transform lower bounds from the BST model into lower bounds for the B-tree model.

Notation. For a search sequence X, we denote $\mathrm{OPT}_{\mathrm{BST}}(X)$ and $\mathrm{OPT}_{\text{B-Tree}}(X)$ the optimal (offline) cost to serve X using a BST-model and a B-tree-model data structure respectively.

Theorem 3.1. *For any search sequence X, $\mathrm{OPT}_{\mathrm{BST}}(X) = O(\mathrm{OPT}_{\text{B-Tree}}(X) \cdot \log B)$.*

Proof. We simulate a B-tree execution of X using a BST in the following way: Each node of the B-tree is simulated by a red-black tree of depth $O(\log B)$. Thus our BST is a tree of red-black trees. We also augment the red-black tree data structure such that each node stores a counter on the number of keys in its subtree. Note that in this tree-of-trees, leaves of a red-black tree might have children, which are the roots of other red-black trees. To distinguish the leaves of each tree, we mark the root of each red-black tree. We also use the parent-child terminology for those red-black trees, i.e., if U and V are red-black trees corresponding to B-tree nodes u and v respectively such that u is a child of v, we will say that "tree U is a child of tree V".

It remains to show that each unit-cost B-tree operation can be simulated in time $O(\log B)$ using our tree-of-trees BST data structure. Moving the pointer from a B-tree node to an

adjacent node corresponds to moving the BST pointer from the root of one red-black tree to the root of its child/parent. This can be done in $O(\log B)$ time, since the depth of our red-black trees is $O(\log B)$. For the other unit-cost operations showing this is more complicated. In order to keep the presentation as simple as possible, we proceed as follows: we first describe some basic properties of red-black trees, we then use them to develop operations of merging and separating red-black which will be useful in out tree-of-trees construction and finally we show how to implement the B-tree unit-cost operations using all those tools.

Background on red-black trees. We note that red-black trees on k nodes support split and concatenate operations, as well as finding the ℓth largest (or smallest key) in time $O(\log k)$ [CLRS09]. We now describe those operations.

- The *split* operation of a red-black tree T at a node x re-arranges the tree such that x is the root and the left and right subtrees are red-black trees including keys of smaller and larger values than x respectively.
- *Concatenating* two red-black trees T_1, T_2 whose roots are children of a common node x, consists of re-arragning the subtree of x to form a red-black tree on all keys of $T_1 \cup T_2 \cup \{x\}$. This operation is also referred as *concatenating at* x and it can be defined even if one of T_1, T_2 is empty. Particularly, in our tree of trees construction, if we concatenate at a node x whose left (right) child is marked, then we treat its left (right) subtree as empty.
- *Find the key with a given rank:* Given an augmented red-black tree on k nodes, where each node stores the number of keys in its subtree and a value $\ell < k$, we can find its ℓth largest (or smallest) key in $O(\log k)$ time (see e.g. [CLRS09, Chapter 14]).

Combining and Separating red-black trees. We now develop two procedures that will be useful in our implementation of B-tree unit cost operations. In particular we show how to merge and separate red-black trees in $O(\log k)$ time, where k is the total number of nodes in the trees involved.

(i) *Merge(S,T)*: Given two red-black trees S and T such that T is a child of S, merge them into one valid red-black tree. We describe an implementation of this operation in $O(\log k)$ time, where k is the total number of nodes of S and T. Let y_T be the root of T. We can find the predecessor ℓ and the successor r of y_T in S in $O(\log k)$ time, by searching for the key value of y_T in S. Note that either ℓ or r might not exist. We split S at ℓ (if it exists) and then split the right subtree in r (if it exists). Now, T is the left subtree of r (if r does not exist, T is just the right subtree of ℓ). Unmark the root of T. Then, concatenate at r (skip this step if r does not exist) and finally concatenate at ℓ (if it exists). The result is a valid red-black tree containing all keys of S and T. We used a constant number of $O(\log k)$-time operations.

(ii) *Separate(T,ℓ,r):* Given a red-black tree T, separate keys with values in the interval $[\ell, r]$, i.e. split T into two trees T_1, T_2 where T_2 contains keys with values in the interval $[\ell, r]$ and T_1 is a parent of T_2. In case ℓ is not specified ($\ell = null$), we think of ℓ as being the minimum key value in T and this operation separates all keys with value at most r. Symmetrically, if r is *null*, we think of r as being the maximum key value in T and this operation separates keys with value at least ℓ. We implement this as follows. Let ℓ' be the predecessor of ℓ in T (if exists) and r' the successor or r (if exists). Split T in ℓ' (skip this step if ℓ' does not exist) and then split the subtree with values larger than ℓ' at r' (skip this step if r' does not exist). As a result the left subtree of r' (or the right subtree of ℓ'

if r' does not exists) is the tree T_2 containing all keys in $[\ell, r]$. Mark the root of T_2. Then concatenate at r' (if exists) and finally concatenate at ℓ' (if exists). As a result we get a valid red-black tree T_1 which is the parent of red-black tree T_2 containing all keys of the interval $[\ell, r]$.

Simulating the unit-cost operations. We now proceed on showing how to simulate B-tree rotations, splits and joins using our tree of red-black trees data structure with cost $O(\log B)$. In all cases, the total number of keys in the trees involved is $O(B)$ and we perform a constant number of operations which take time $O(\log B)$.

- **Rotations.** We show how to implement a rotation of the form demote left ℓ - promote right k (assuming valid values of ℓ and k). The other operations are defined analogously. Let (u, v) be the B-tree edge which is rotated, where node u is parent of v and let U and V be the augmented red-black trees corresponding to u and v. Let $u_1, \ldots, u_j, u_{j+1}, \ldots, u_m$ and $v_1, \ldots, v_n$ be the key values stored in u and v respectively such that for all v_i we have that $u_j < v_i < u_{j+1}$, similar to the example in figure 1. The rotation corresponds to promoting to U the k largest keys of V, i.e. $v_{n-k+1}, \ldots, v_n$ and demoting to V the keys $u_{j-\ell+1}, \ldots, u_j$. We implement such a rotation as follows (see figure 2 for an illustration): We start by promoting the k elements to U. Find v_{n-k}, i.e. the $(k+1)$th largest key stored at V. Then, Separate$(V, null, v_{n-k})$ to get a tree V_1 containing keys $v_{n-k+1}, \ldots, v_n$ and a tree V_2 with the rest keys of V. V_2 is a child of V_1 and V_1 is a child of U. Now, we merge U and V_1 to get a new tree U', such that V_2 is a child of U'. It remains to demote $u_{j-\ell+1}, \ldots, u_j$ to V_2. To do that, we split U' at v_{n-k+1}. Let U_L and U_R be the two subtrees of v_{n-k+1} in U'. Note that $u_{j-\ell+1}, \ldots, u_j$ are the ℓ largest keys of U_L. Find $u_{j-\ell+1}$, i.e., the ℓth largest key of U_L and Separate$(U_L, u_{j-\ell+1}, null)$. We get a separate tree U_{L_2} containing $u_{j-\ell+1}, \ldots, u_j$. Mark the root of U_{L_2}. Now, V_2 is a child of U_{L_2}, so we can merge them to form V'', the tree corresponding to B-tree node v. Finally we concatenate at the root v_{n-k+1}, to form the final tree corresponding to u, denoted by U'', where V'' is a child of U''.

- **Splitting a node of a B-tree.** Let u be the node which we want to split and $p(v)$ its parent. Let also U and P the corresponding red-black trees, where U is a child of P. Let u_m be the median key value of U. We split U at u_m, so that u_m is the root with subtrees U_L and U_R. Mark the roots of U_L and U_R and then merge u_m (which is a single-node red-black tree) with P. Clearly all those operations can be performing in $O(\log B)$ time.

- **Joining two sibling nodes.** This is the inverse operation of splitting so the sequence of operations can be seen as the symmetric of the ones performed in splitting. Let u and v be the sibling B-tree nodes that we want to join, and p their parent, with U, V and P the corresponding red-black trees in our binary search tree. U and V are children of P and there is a unique key p_j in P such that keys stored at U are smaller than p_j and keys stored at V are larger. Thus, p_j is the successor of the root of U in P and we can find it in $O(\log B)$ time. We then Separate(P, p_j, p_j). Now we get a new tree P_1 containing all keys of P except from p_j, and p_j is a single-node red-black tree, child of P_1. U and V are the left and right children of p_j. We unmark the roots of U and V and concatenate at p_j, to get a new tree U' and mark its root. Now U' corresponds to the join node of u and v, and it is a child of the red-black tree P' which corresponds to the parent node in the B-tree. We performed a constant number of operations each of which takes time $O(\log B)$.

□

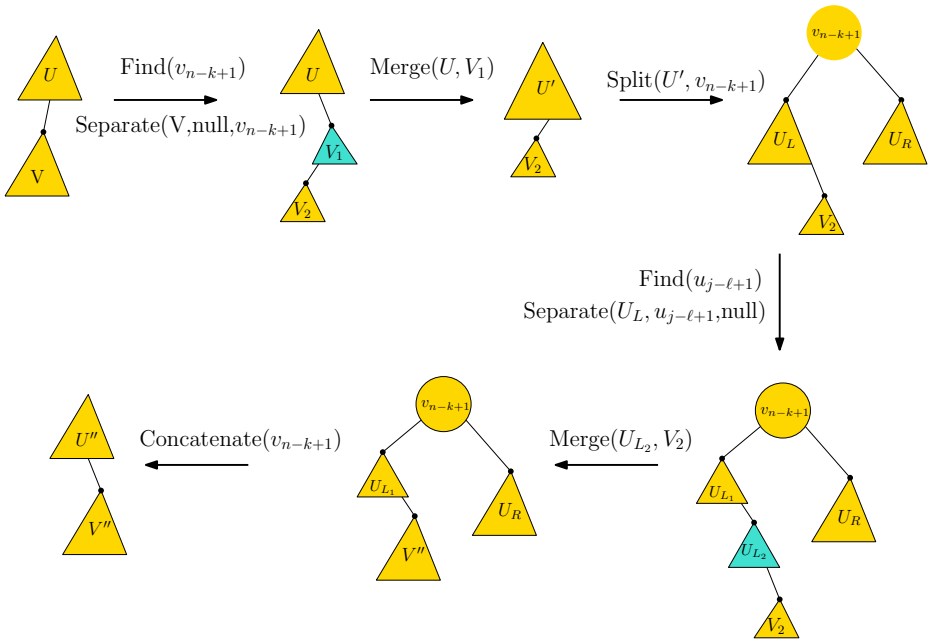

Figure 2: Simulating a rotation of a B-tree edge (u, v) of the type demote left ℓ - promote right k in the BST model using red-black tree operations of merge, separate, split, concatenate and find a key with a given rank.

Theorem 3.1 implies that we can transform any lower bound for binary search trees to a lower bound for dynamic B-trees, as shown in the following corollary.

Corollary 3.2. *Let X be a search sequence and let* $\mathrm{LB}(X)$ *be any lower bound on the cost of executing X in the BST model. Then we have that* $\mathrm{OPT}_{\text{B-Tree}}(X) = \Omega\left(\frac{\mathrm{LB}(X)}{\log B}\right)$.

Proof. Since $\mathrm{LB}(X)$ is a lower bound on $\mathrm{OPT}_{\mathrm{BST}}(X)$, we have that $\mathrm{LB}(X) \leq \mathrm{OPT}_{\mathrm{BST}}(X) = O(\log B) \cdot \mathrm{OPT}_{\text{B-Tree}}(X)$, which implies $\mathrm{OPT}_{\text{B-Tree}}(X) = \Omega\left(\frac{\mathrm{LB}(X)}{\log B}\right)$. □

4 Belga B-trees

In this section, we develop a dynamic B-tree data structure yclept *Belga B-tree* that achieves a competitive ratio of $O(\log \log N)$, for search sequences of length $\Omega(N)$, provided that $1 + \log_B \log N = O(\log_B \log N)$, i.e. $B = (\log N)^{O(1)}$. Our construction is built upon the ideas used in [DHIP07] to get a similar competitive ratio for binary search trees. Particularly, we crucially connect the cost of our algorithm to the *interleave lower bound*. For completeness, we present here the setup and the necessary background regarding this lower bound.

Interleave Lower Bound and preferred paths (See Figure 3). Let $\{1, \ldots, N\}$ be the keys stored in our B-tree. Let P be a (fixed) complete binary search tree on those keys. For each internal node v in P, we define its left region to be v together with the subtree rooted at its left child and its right region to be the subtree rooted at its right child. Node v has a *preferred child*, which is left or right, depending on whether the last search for a node in its subtree was in its left or right region (if no node of the subtree rooted at v has been searched, then v has no preferred child).

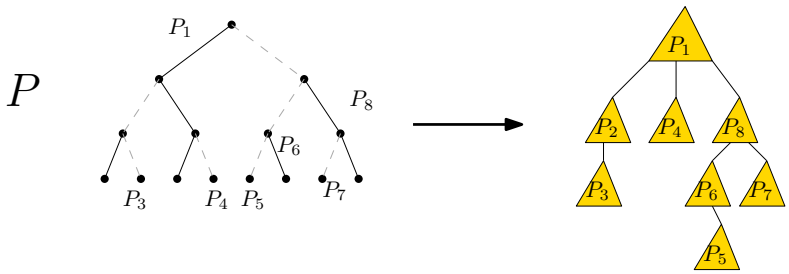

Figure 3: Example of a reference tree P and the tree-of-trees representation of its preferred paths $P_1, \ldots, P_8$. Edges connecting different preferred paths are dashed gray.

We define a *preferred path* in P as follows: Start from a node that is not the preferred child of its parent (including the root) and perform a walk by following the preferred child of the current node, until reaching a node with no preferred child. Clearly, a preferred path contains $O(\log N)$ keys.

Note that during a search for a key, the preferred child of some nodes that are ancestors of the node with the key being searched might change. Each change of preferred child, changes also the preferred paths of P. For a search sequence X, the interleave lower bound $\mathrm{IB}(X)$ equals the total number of changes of preferred child from left to right or from right to left, over all nodes of P. We use the following lemma of [DHIP07], which is a slight variant of the first lower bound of [Wil89]:

Lemma 4.1 (Lemma 3.2 in [DHIP07])**.** *The cost to execute X in the BST model is $\Omega(\mathrm{IB}(X))$ if $|X| = \Omega(N)$.*

High-level overview of our structure. We store each preferred path in a balanced classic B-tree. We call such classic B-trees *auxiliary trees*. Our dynamic B-tree will be a tree of classic B-trees. Recall that Lemma 4.1 essentially tells us that the number of preferred paths touched during a request sequence is a lower bound on the value of $\mathrm{OPT}_{\mathrm{BST}}$. The idea here is to show that for each preferred path touched, and thus unit of lower bound incurred, we can perform search and all update operations (cutting and merging preferred paths) with an overhead factor $O(\log_B \log N) = O(\frac{\log \log N}{\log B})$. This will imply that we have a dynamic B-tree with cost $O(\frac{\log \log N}{\log B} \cdot \mathrm{IB}(X))$. This combined with Lemma 4.1 and Corollary 3.2 implies that the cost of our dynamic B-tree data structure is $O(\log \log N) \cdot \mathrm{OPT}_{\text{B-Tree}}$.

Auxiliary trees. Our auxiliary trees are augmented classic B-trees. Each auxiliary tree stores a preferred path. With each key x we also store its depth in the reference tree P. We call this value depth of key x. Also, each node stores the minimum and maximum depth of a key in its subtree. Last, a node may be marked or unmarked, depending on whether it is the root of an auxiliary tree or not. Note that P is just a reference tree used for the analysis. We do not need to store P explicitly in order to implement our algorithm. All necessary information about P is stored in our dynamic B-tree data structure.

During an execution of a search sequence we need to perform the following operations on a preferred path:

(i) Search for a key.

(ii) Cut the preferred path into two paths, one consisting of keys of depth at most d and the other of keys of depth greater than d.

(iii) Merge two preferred paths P_1 and P_2, where the bottom node of P_1 is the parent of the top node of P_2.

We will show that we can perform those operations using our auxiliary trees in time $O(1 + \log_B k)$, where k is the number of keys in the involved preferred paths. We defer this proof to the end of this section and we now proceed to the description and analysis of Belga B-trees, assuming that those operations can be done in time $O(1 + \log_B k)$. For the rest of this section, whenever we refer to cutting/merging operations on auxiliary trees, we mean the implementation of cutting/merging the corresponding preferred paths in our B-tree data structure.

Our Algorithm. A Belga B-tree is a tree of auxiliary classic B-trees, where each auxiliary tree stores a preferred path. Initially we transform the input tree T_0 to a valid Belga B-tree. Upon a request for a key x_i, we start from the root and search for x_i. Whenever we reach a marked node v (i.e. a root of an auxiliary tree), we have to update the preferred paths. Let Q be the preferred path stored in the auxiliary tree of the parent of v and R the preferred path in the auxiliary tree rooted at v. We update the preferred paths using the cut and merge operations of auxiliary trees. Particularly, if d is the minimum depth of a key of R (this value is stored at node v of our B-tree), we cut the auxiliary tree storing Q at depth $d-1$. This gives us two preferred paths Q_{d-} and Q_{d+}, where the first stores keys of Q of depth at most $d-1$ and the second keys of Q of depth greater than d. We mark the roots of the auxiliary trees corresponding Q_{d-} and Q_{d+}. We then merge the auxiliary tree storing Q_{d-} with the auxiliary tree rooted at v (which stores R). We mark the root of the new tree and continue the search for x_i.

Note that the only part where our algorithm needs to perform rotations is the initial step of transforming the input tree into a Belga B-tree.

Bounding the cost. We now compare the cost of our Belga B-tree data structure to that of the optimal offline B-tree. The following lemma makes the essential connection between the number of preferred paths touched during a search and the cost of our algorithm.

Lemma 4.2. *Let ℓ be the number of preferred child changes during a search for key x_i. Then the cost of Belga B-tree for searching x_i is $O((\ell + 1)(1 + \log_B \log N))$.*

Proof. To search for x_i, we touch exactly $\ell + 1$ preferred paths. We account separately for the search cost and the update cost.

For each preferred path touched, the search cost is $O(\lceil \log_B \log N \rceil)$, since we are searching a balanced B-tree on $O(\log N)$ keys. Thus the total search cost is clearly $O((\ell+1)(1+\log_B \log N))$.

We now account for the update cost. Recall that we can cut and merge preferred paths on k keys in time $O(1 + \log_B k)$. Since each preferred path has at most $O(\log N)$ keys, we can perform those updates in time $O(1 + \log_B \log N)$. There are ℓ preferred path changes, and for each change we perform one cut and and one merge operation, we get that the total time for merging and cutting is $O(\ell \cdot (1 + \log_B \log N))$. The lemma follows. □

We now combine this lemma with Corollary 3.2 to get the competitive ratio of Belga B-trees.

Theorem 4.3. *For any search sequence of length $m = \Omega(N)$, Belga B-trees are $O(\log \log N)$-competitive.*

Proof. We account only for the cost occured during searches, since the cost of transforming the input tree into a Belga B-tree is just a fixed additive term which does not depend on the input sequence.

The total number of preferred path changes is at most $\mathrm{IB}(X)+N$. The additive N accounts for the fact that initially each node has no preferred child, so its first change from null to either left or right is not counted in $\mathrm{IB}(X)$. Using Lemma 4.2 and summing up over all search requests, we get that the cost of Belga B-trees is $O((\mathrm{IB}(X)+N+m)(1+\log_B \log N))$. By our assumption on the value of B, we have that $1+\log_B \log N = O(\log_B \log N)$, thus the cost is in $O((\mathrm{IB}(X)+N+m)\cdot \frac{\log\log N}{\log B})$. By Lemma 4.1 this is bounded by $(\mathrm{OPT}_{\mathrm{BST}}(X)+N+m)\cdot \frac{\log\log N}{\log B}$. Using Corollary 3.2 we get that cost of Belga B-tree is

$$O\left((\log B \cdot \mathrm{OPT}_{\text{B-Tree}} + N + m)\cdot \frac{\log\log N}{\log B}\right).$$

Note that for any request sequence $\mathrm{OPT}_{\text{B-Tree}} \geq m$. Since $m = \Omega(N)$, we have that $\log B \cdot \mathrm{OPT}_{\text{B-Tree}} + N + m = O(\log B \cdot \mathrm{OPT}_{\text{B-Tree}})$. We get that the total cost is upper bounded by

$$O\left(\log B \cdot \mathrm{OPT}_{\text{B-Tree}} \cdot \frac{\log\log N}{\log B}\right) = O(\mathrm{OPT}_{\text{B-Tree}} \cdot \log\log N).$$

□

Operations on auxiliary trees in logarithmic time. We now show that our auxiliary B-trees support search, cut and merge in time $O(1+\log_B k)$, where k is the total number of nodes in the trees which are involved.

Before proceeding to this proof we note that classic B-trees on k nodes support search, split and concatenate (similar to the ones we presented in previous section for red-black trees) operations in time $O(1+\log_B k)$ (see [CLRS09], Chapter 18). For completeness we describe here the split and concatenate operations:

- Splitting a B-tree at a key value x consists of creating a tree where the root contains only x, its left subtree is a B-tree on keys with value smaller than x and the right subtree is a B-tree on keys greater than x.
- Concatenating two classic B-trees T_1, T_2 with a key value k such that all keys in T_1 are smaller than k and all keys in T_2 are greater, consists of creating a new classic B-tree T which contains all key values contained in T_1, T_2 and k.

Search can be clearly performed in time $O(1+\log_B k)$. We now describe the cut and merge operations on preferred paths.

Cut a preferred path at depth d: Let R be the tree storing the preferred path. Let ℓ and r be the smallest and the largest key value respectively stored at depth greater than d in the path. We wish to find ℓ and r in the tree R. This can be easily done using the maximum depth value of subtree stored in the nodes. We show how to find ℓ and for r is symmetric. Start from the root and move to the leftmost child whose maximum depth is greater than d. When we reach a node v such that all its children have maximum depth smaller than d, then ℓ is the smallest key in v with depth greater than d. Let ℓ' predecessor of ℓ in R (if it has one) and r' the successor of r in R (if it has one). Split R at ℓ' (skip this step if ℓ' does not exist) and then split the right subtree at r' (skip this step if r' does not exist). Now, the left subtree of r' contains all keys with depth greater than d. Let us call this tree D. Mark the root of D (and change values of depths, max depth, min depth in time $O(1+\log_B k)$) and then use concatenate operations at the tree rooted at r' (if it exists) and then at the tree rooted at ℓ' (if it exists) to make the remaining of R a valid classic B-tree.

Merge two preferred paths: Let P_1 and P_2 be the preferred paths that we want to merge, where the bottom node of P_1 is the parent of the top node of P_2. Merging is the inverse operation

of a cut. Let U and V be the auxiliary trees storing P_1 and P_2 respectively, i.e U is a parent of V in our tree-of-trees construction and the key values stored at U have are of smaller depth in P than the key values stored in V. Pick a key from the root of V and find its predecessor ℓ and its successor r in U. Split U in ℓ (skip this step if ℓ does not exist) and then split the right subtree at r (skip this step if r does not exist). Now the left subtree of r is V. Unmark the root of V. Then, concatenate at r to get a resulting tree R which is the right subtree of the root ℓ (skip this step if r does not exist). Then, concatenate at ℓ (if it exists), to get a valid B-tree which contains all keys of U and V. In each of the last two steps (if not skipped), updates of the values of depth, maximum depth, minimum depth take time $O(1 + \log_B k)$.

5 Transforming any static BST into the B-Tree model

In this section we focus on static trees, with the goal to simulate a static BST using a static B-tree and achieving a speedup by a factor of $\Theta(\log B)$. In the static BST and B-Tree models, all that is allowed in each operation is to move a single pointer around the tree, starting at the root, each time moving to a neighboring node, at unit cost per move. We refer to a sequence of moves of a single pointer as a *walk*. In particular, given a BST we wish to convert it to a B-Tree so that if a walk in the BST costs k, a walk in the B-Tree T_B that touches the same keys costs as little as possible in terms of k; k is clearly possible since a BST is a B-tree, but when can we achieve $o(k)$?

We note that the results of this section allow the pointer to move arbitrarily in a static BST/B-tree, i.e., it can visit nodes that are outside the path from the root to the searched node. In the case where only a search path of length D is considered, the worst-case cost has been completely characterized in [DIL15] as $\Theta\left(\frac{D}{\lg(1+B)}\right)$ when $D = O(\lg N)$, $\Theta\left(\frac{\lg N}{\lg\left(1+\frac{B\lg N}{D}\right)}\right)$, when $D = \Omega(\lg N)$ and $D = O(B \lg N)$, and $\Theta\left(\frac{D}{B}\right)$ when $D = \Omega(B \lg N)$.

Block-Connected Mappings. The most natural approach to achieve our goal is to try to map a static BST T into a static B-tree T_B such that each node of T_B corresponds to a connected subtree of T. We call such a mapping $f : T \to T_B$, *block-connected*. Observe that in order to achieve a $\Omega(\log B)$ speedup for the B-tree model T_B, it is necessary that a block-connected mapping f should satisfy that every node at depth d in T is at depth $O(\frac{k}{\log B})$ in T_B. However, as we will see, this is not sufficient.

The next theorem shows that, perhaps surprisingly, this approach fails to give any super-constant factor improvement, given that the mapping is deterministic. Afterwards, we show how to achieve an $\Omega(\log B)$ factor speedup using randomization.

Theorem 5.1. *There does not exist a block-connected mapping $f : T \to T_B$ such that any walk P on T of length k corresponds to a walk of length $o(k)$ in T_B.*

Proof. We proceed by contraction. Assume an f and $N = 2^i - 1$ for some integer i, and let T be the perfectly balanced tree with N nodes and thus $\ell = \frac{N+1}{2}$ leaves. Consider some BST model sequence of operations E which is an inorder traversal of T. Let b be the number of different blocks (i.e. B-tree nodes) that $f(T)$ stores the leaves of T in, which must be at least $\frac{\ell}{B}$. Let E' be the sequence of operations where the inorder traversal does not recurse whenever it encounters a node stored in the same block as a leaf. E' will still visit all b blocks containing leaves, but its length will be exactly $2b-1$. This happens because the block-connected property ensures that E' will never visit two nodes, both of which are in the same block as a leaf of T, as that would imply they would have an LCA also in the block, which would mean E' would

not visit them. Thus E' has a BST cost of $2b-1$ and a B-tree cost of $\Theta(b)$, where $b = \Omega(\frac{N}{b})$ which proves the theorem. □

Randomized Construction. Theorem 5.1 above is based on an adversarial argument and relies crucially on the knowledge of the layout of the B-tree. To overcome this issue, we use randomization.

Theorem 5.2. *For any BST T, there is a randomized block-connected mapping which produces a static B-tree T_R such that for any walk of length k in T, there exists a corresponding walk in T_R with expected cost $O\left(\frac{k}{\log B}\right)$.*

Proof. We construct the B-tree T_R as follows. We choose uniformly at random an integer h in $[0, \lfloor \log B \rfloor - 1]$. The root node of T_R contains the key values of the first h levels of T. Then, we build the rest of the tree in a deterministic way, by storing $\lfloor \log B \rfloor - 1$ levels of each subtree in a B-tree node, recursively. Consider any walk P of k operations on T that starts at the root. We assume that the block containing the root and the current location of the walk are stored in memory. Whenever P passes through an edge e of T, the probability that this move corresponds to a unit cost operation equals the probability that the endpoints of e belong to different B-tree nodes in T_R and equals $1/\lfloor \log B \rfloor$.

We thus obtain that the expected cost of the corresponding sequence of operations in in T_R is $k/\lfloor \log B \rfloor$. Since $\lfloor \log B \rfloor = \Omega(\log B)$ for any $B \geq 3$, we get that the expected cost is $O(\frac{k}{\log B})$. □

6 Open Problems

We conclude with some open problems. The first is that our Belga B-trees are $O(\log \log N)$-competitive only when $B = \log^{O(1)} N$, and thus the case of large B where $B = \log^{\omega(1)} N$ remains open. The main impediment is to figure out how to fit multiple preferred paths into one block.

A more general open problem is to resolve the following conjecture: Is it possible to convert any BST-model algorithm into a B-Tree model algorithm such that if an algorithm costs $O(k)$ in the BST model, it costs $O(\frac{k}{\log B} + 1)$ in the B-Tree model? Special cases of this theorem, when applied to, for example, splay trees and greedy future, would also be interesting should the general conjecture prove too difficult to resolve.

A third open problem is whether, given two B-tree model algorithms, can you achieve the runtime that is the minimum of them; this would be the B-Tree model analogue of the BST result of [DILÖ13]. It would also allow one to then combine Belga B-trees with other B-tree model algorithms to get stronger results, like, for example [BDL08] to add the working-set bound; in the BST model [WDS06] gave a $O(\log \log N)$-competitive BST with the working set bound.

References

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[AVL62] G. M. Adelson-Velskiĭ and E. M. Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

[BCDI07] Mihai Badoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theor. Comput. Sci.*, 382(2):86–96, 2007.

[BDIL16] Prosenjit Bose, Karim Douïeb, John Iacono, and Stefan Langerman. The power and limitations of static binary search trees with lazy finger. *Algorithmica*, 76(4):1264–1275, 2016.

[BDL08] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Symposium on Discrete Algorithms, SODA*, pages 1106–1114, 2008.

[BHM13] Prosenjit Bose, John Howat, and Pat Morin. A history of distribution-sensitive data structures. In Brodnik et al. [BLRV13], pages 133–149.

[BLRV13] Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors. *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*. Springer, 2013.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[CGK+16] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *CoRR*, abs/1603.04892, 2016.

[CGK+18] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Multi-finger binary search trees. In *29th International Symposium on Algorithms and Computation, ISAAC*, pages 55:1–55:26, 2018.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[CMSS00] Richard Cole, Bud Mishra, Jeanette P. Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part I: splay sorting log n-block sequences. *SIAM J. Comput.*, 30(1):1–43, 2000.

[Col00] Richard Cole. On the dynamic finger conjecture for splay trees. part II: the proof. *SIAM J. Comput.*, 30(1):44–85, 2000.

[DHI+09] Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, and Mihai Patrascu. The geometry of binary search trees. In *Symposium on Discrete Algorithms, SODA*, pages 496–505, 2009.

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality - almost. *SIAM J. Comput.*, 37(1):240–251, 2007.

[DIL15] Erik D. Demaine, John Iacono, and Stefan Langerman. Worst-case optimal tree layout in external memory. *Algorithmica*, 72(2):369–378, 2015.

[DILÖ13] Erik D. Demaine, John Iacono, Stefan Langerman, and Özgür Özkan. Combining binary search trees. In *ICALP 2013, Part I*, pages 388–399, 2013.

[EFI13] Amr Elmasry, Arash Farzan, and John Iacono. On the hierarchy of distribution-sensitive properties for data structures. *Acta Inf.*, 50(4):289–295, 2013.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[HIM13] John Howat, John Iacono, and Pat Morin. The fresh-finger property. *CoRR*, abs/1302.6914, 2013.

[Iac01a] John Iacono. Alternatives to splay trees with o(log n) worst-case access times. In *Symposium on Discrete Algorithms (SODA)*, pages 516–522, 2001.

[Iac01b] John Iacono. *Distribution Sensitive Data Structures*. PhD thesis, Ph.D. Thesis. Rutgers, The State University of New Jersey, 2001.

[Iac13] John Iacono. In pursuit of the dynamic optimality conjecture. In Brodnik et al. [BLRV13], pages 236–250.

[IL16] John Iacono and Stefan Langerman. Weighted dynamic finger in binary search trees. In *Symposium on Discrete Algorithms, SODA*, pages 672–691, 2016.

[Luc88] Joan M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, 1988.

[She95] Murray Sherk. Self-adjusting k-ary search trees. *J. Algorithms*, 19(1):25–44, 1995.

[ST85a] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[ST85b] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[Tar85] Robert Endre Tarjan. Sequential access in play trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

[WDS06] Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. $O(\log \log n)$-competitive dynamic binary search trees. In *Symposium on Discrete Algorithms, SODA*, pages 374–383, 2006.

[Wil89] Robert E. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM J. Comput.*, 18(1):56–67, 1989.