Searching in Dynamic Tree-Like Partial Orders

Brent Heeringa[1], Marius Cătălin Iordan[2], and Louis Theran[3]

[1] Dept. of Computer Science. Williams College. heeringa@cs.williams.edu*
[2] Dept. of Computer Science. Stanford University. mci@cs.stanford.edu**
[3] Dept. of Mathematics. Temple University. theran@temple.edu***

Abstract. We give the first data structure for the problem of maintaining a dynamic set of n elements drawn from a partially ordered universe described by a tree. We define the Line-Leaf Tree, a linear-sized data structure that supports the operations: *insert; delete; test membership; and predecessor*. The performance of our data structure is within an $O(\log w)$-factor of optimal. Here $w \leq n$ is the width of the partial-order—a natural obstacle in searching a partial order.

1 Introduction

A fundamental problem in data structures is maintaining an ordered set S of n items drawn from a universe $\mathcal{U}$ of size $M \gg n$. For a totally ordered $\mathcal{U}$, the dictionary operations: *insert*; *delete*; *test membership*; and *predecessor* are all supported in $O(\log n)$ time and $O(n)$ space in the comparison model via balanced binary search trees. Here we consider the relaxed problem where $\mathcal{U}$ is partially ordered and give the first data structure for maintaining a dynamic partially ordered set drawn from a universe that can be described by a tree.

As a motivating example, consider an email user that has stockpiled years of messages into a series of hierarchical folders. When searching for an old message, filing away a new message, or removing an impertinent message, the user must navigate the hierarchy. Suppose the goal is to minimize, in the worst-case, the number of folders the user must consider in order to find the correct location in which to retrieve, save, or delete the message. Unless the directory structure is completely balanced, an optimal search does not necessarily start at the top—it might be better to start farther down the hierarchy if the majority of messages lie in a sub-folder. If we model the hierarchy as a rooted, oriented tree and treat the question "is message x contained somewhere in folder y?" as our comparison, then maintaing an optimal search strategy for the hierarchy is equivalent to maintaining a *dynamic* partially ordered set under insertions and deletions.

Related Work. The problem of searching in trees and partial orders has recently received considerable attention. Motivating this research are practical

* Supported by NSF grant IIS-08125414.
** Supported by the William R. Hewlett Stanford Graduate Fellowship.
*** Supported by CDI-I grant DMR 0835586 to Igor Rivin and M. M. J. Treacy.

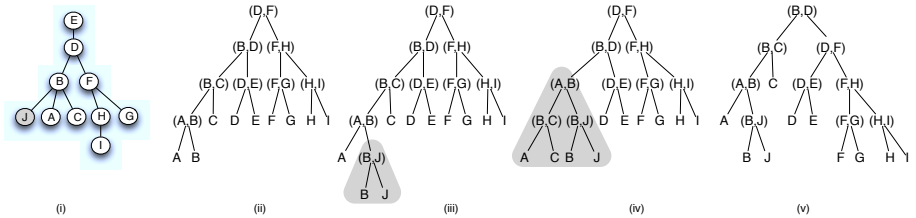

Fig. 1. (i) A partially ordered set $\{A, B, C, D, E, F, G, H, I, J\}$. A downward path from node X to node Y implies $X \prec Y$. Note that, for example, $E \prec F$ and G and I are incomparable. (ii) An optimal search tree for the set $\{A, B, \ldots, I\}$. For any query (X, Y) an answer of X means descend left and an answer of Y means descend right. (iii) After adding the element J, a standard search tree would add a new query (B, J) below (A, B) which creates an imbalance. (iv) The search tree after a rotation; the subtree highlighted in grey is not a correct search tree for the partial order (i). (v) An optimal search tree for the set $\{A, B, \ldots, J\}$.

problems in filesystem synchronization, software testing and information retrieval [1]. However, all of this work is on the *static* version of the problem. In this case, the set S is fixed and a search tree for S does not support the insertion or deletion of elements. For example, when S is totally ordered, the optimal minimum-height solution is a standard binary search tree. In contrast to the totally ordered case, finding a minimum height static search tree for an arbitrary partial order is NP-hard [2]. Because of this, most recent work has focused on partial orders that can be described by rooted, oriented trees. These are called *tree-like* partial orders in the literature. For tree-like partial orders, one can find a minimum height search tree in linear time [3–5]. In contrast, the weighted version of the tree-like problem (where the elements have weights and the goal is to minimize the *average* height of the search tree) is NP-hard [6] although there is a constant-factor approximation [7]. Most of these results operate in the edge query model which we review in Sec. 2.

Daskalakis et al. have recently studied the problem of *sorting* partial orders [8, 9] and, in [9], ask for analogues of balanced binary search trees for dynamic partially ordered sets. We are the first to address this question.

Rotations do not preserve partial orders. Traditional data structures for dynamic ordered sets (*e.g.*, red black trees, AVL trees) appear to rely on the total order of the data. All these data structures use binary tree rotations as the fundamental operations; applied in an unrestricted manner, rotations *require* a totally ordered universe. For example, consider Figure 1 (ii) which gives an optimal search tree for the elements $\{A, B, \ldots, I\}$ depicted in the partial order of Figure 1 (i). If we insert node J (colored grey) then we must add a new test (B, J) below (A, B) which creates the sub-optimal search tree depicted in Figure 1 (iii). Using traditional rotations yields the search tree given in Figure 1 (iv) which does not respect the partial order; the leaf marked C should appear under the right child of test (A, B). Figure 1 (v) denotes a correct optimal search for the

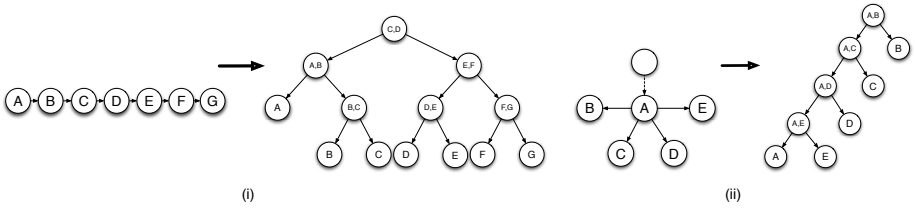

Fig. 2. Examples of (i) a line contraction where we build a balanced binary search tree from a path and (ii) a leaf contraction where we build a linear search tree from the leaves of a node.

set $\{A, B, \ldots, J\}$. The key observation is that, if we imagine the leaves of a binary search tree for a total order partitioning the real line, rotations preserve the order of the leaves, but not any kind of subtree relations on them. As a consequence, blindly applying rotations to a search tree for the static problem does not yield a viable dynamic data structure. To sidestep this problem, we will, in essence, decompose the tree-like partial order into totally ordered chains and totally incomparable stars.

Our Techniques and Contributions We define the Line-Leaf Tree, the first data structure that supports the fundamental dictionary operations for a *dynamic* set $S \subseteq \mathcal{U}$ of n elements drawn from a universe equipped with a *partial order* $\preceq$ described by a rooted, oriented tree.

Our dynamic data structure is based on a static construction algorithm that takes as input the *Hasse diagram* induced by $\preceq$ on S and in $O(n)$ time and space produces a Line-Leaf Tree for S. The Hasse diagram H_S for S is the directed graph that has as its vertices the elements of S and a directed edge from x to y if and only if $x \prec y$ and no z exists such that $x \prec z \prec y$. We build the Line-Leaf Tree inductively via a natural contraction process which starts with H_S and, ignoring the edge orientations, repeatedly performs the following two steps until there is a single node:

1. Contract paths of degree-two nodes into balanced binary search trees (which we can binary search efficiently); and
2. Contract leaves into linear search structures associated with their parents (since the children of an interior node are mutually incomparable).

One of these steps always applies in our setting since H_S is a rooted, oriented tree. We give an example of each step of the construction in Figure 2. We show that the contraction process yields a search tree that is provably within an $O(\log w)$-factor of the minimum-height static search tree for S. The parameter w is the *width* of S—the size of the largest subset of mutually incomparable elements of S—which represents a natural obstacle when searching a partial order. We also show that our analysis is tight. Our construction algorithm and analysis appear in Section 3.

To make the Line-Leaf Tree *fully dynamic*, in Section 4 we give procedures to update it under insertions and deletions. All the operations, take

$O(\log w) \cdot OPT$ comparisons and RAM operations where OPT is the height of a minimum-height static search tree for S. Additionally, *insertion* requires only $O(h)$ comparisons, where h is the height of the Line-Leaf Tree being updated. (The non-restructuring operations *test membership* and *predecessor* also require at most $O(h)$ comparisons since the Line-Leaf Tree is a search tree). Because w is a property of S, in the dynamic setting it changes under insertions and deletions. However, the Line-Leaf Tree maintains the $O(\log w) \cdot OPT$ height bound *at all times*. This means it is well-defined to speak of the $O(\log w) \cdot OPT$ upper bound without mentioning S.

The insertion and deletion algorithms maintain the invariant that the updated Line-Leaf Tree is structurally equivalent to the one that we would have produced had the static construction algorithm been applied to the updated set S. In fact, the heart of insertion and deletion is *correcting* the contraction process to maintain this invariant. The key structural property of a Line-Leaf Tree—one that is not shared by constructions for optimal search trees in the static setting—is that its sub-structures essentially represent either paths or stars in S, allowing for updates that make only local changes to each component search structure. The $O(\log w)$-factor is the price we pay for the additional flexibility. The dynamic operations, while conceptually simple, are surprisingly delicate. We devote detailed attention to them in the full version of this paper [10].

In Section 5 we provide empirical results on both random and real-world data that show the Line-Leaf Tree is strongly competitive with the static optimal search tree.

2 Models and Definitions

Let $\mathcal{U}$ be a finite set of M elements and let $\preceq$ be a partial order, so the pair $(\mathcal{U}, \preceq)$ forms a *partially ordered set*. We assume the answers to $\preceq$-queries are provided by an oracle. (Daskalakis, et al. [8] provide a space-efficient data structure to answer $\preceq$-queries in $O(1)$ time.)

In keeping with previous work, we say that $\mathcal{U}$ is *tree-like* if $H_{\mathcal{U}}$ forms a rooted, oriented tree. Throughout the rest of this paper, we assume that $\mathcal{U}$ is tree-like and refer to the vertices of $H_{\mathcal{U}}$ and the elements of $\mathcal{U}$ interchangeably. For convenience, we add a dummy minimal element ν to $\mathcal{U}$. Since any search tree for a set $S \subseteq \mathcal{U}$ embeds with one extra comparison into a corresponding search tree for $S \cup \{\nu\}$, we assume from now on that ν is always present in S. This ensures that the Hasse diagram for S is always connected.

Given these assumptions it is easy to see that tree-like partial orders have the following properties:

Property 1. Any subset S of a tree-like universe $\mathcal{U}$ is also tree-like.

Property 2. Every non-root element in a tree-like partially ordered set $S \subseteq \mathcal{U}$ has exactly one predecessor in H_S.

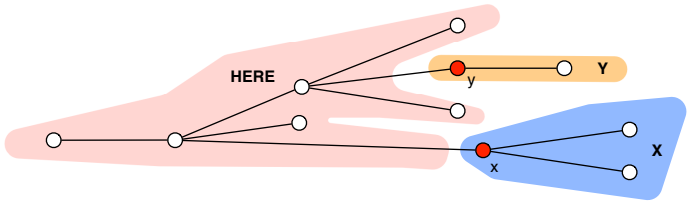

Fig. 3. Given two nodes x and y in S and a third node $u \in \mathcal{U}$, a dynamic edge query on (x, y) with respect to u can answer (i) Y, in which case u falls somewhere in the shaded area labelled Y; (ii) X, in which case u falls somewhere in the shaded area labelled X; or (iii) HERE, in which case u falls somewhere in the shaded area labelled HERE. Notice that if (x, y) forms an actual edge then the query reduces to a standard edge query

Let T_S be the rooted, oriented tree corresponding to the Hasse diagram for S. We extend edge queries to *dynamic edge queries* by allowing queries on arbitrary pairs of nodes in T_S instead of just edges in T_S.

Definition 1 (Dynamic Edge-Queries). *Let u be an element in $\mathcal{U}$ and x and y be nodes in T_S. Let $S' = S \cup \{u\}$ and consider the edges (x, x') and (y, y') bookending the unique path from x to y in $T_{S'}$. Define $T^x_{S'}$, $T^y_{S'}$ and $T^{\text{HERE}}_{S'}$ to be the three connected components of $T_{S'} \setminus \{(x, x'), (y, y')\}$ containing x, y, and neither x nor y, respectively. A* dynamic edge query *on (x, y) with respect to u has one of the following three answers:*

1. X*: if $u \in T^x_{S'}$ (u equals or is closer to x)*
2. Y*: if $u \in T^y_{S'}$ (u equals or is closer to y)*
3. HERE*: if $u \in T^{\text{HERE}}_{S'}$ (u falls between, but is not equal to either, x or y)*

Figure 3 gives an example of a dynamic edge query. Any dynamic edge query can be simulated by $O(1)$ standard comparisons when H_S is tree-like. This is not the case for more general orientations of H_S and an additional data structure is required to implement either our algorithms or algorithms of [3, 4]. Thus, for a tree-like S, the height of an optimal search tree in the dynamic edge query model and the height of an optimal decision tree for S in the comparison model are always within a small constant factor of each other. For the rest of the paper, we will often drop *dynamic* and refer to *dynamic edge queries* simply as *edge queries*.

3 Line-Leaf Tree Construction and Analysis

We build a Line-Leaf Tree $\mathcal{T}$ inductively via a contraction process on T_S. Each contraction step builds a *component search structure* of the Line-Leaf Tree. These component search structures are either linear search trees or balanced binary search trees. A linear search tree $LST(x)$ is a sequence of dynamic edge queries, all of the form (x, y) where $y \in S$, that ends with the node x. A balanced binary search tree $BST(x, y)$ for a path of contiguous degree-2 nodes between,

but not including, x and y is a tree that *binary searches* the path using edge queries.

Let $T_0 = T_S$. If the contraction process takes m iterations total, then the final result is a single node which we label $\mathcal{T} = T_{2m}$. In general, let T_{2i-1} be the partial order tree after the line contraction of iteration i and T_{2i} be the partial order tree after the leaf contraction of iteration i where $i \geq 1$. We now show how to construct a LINE-LEAF TREE for a fixed tree-like set S.

Base Cases Associate an empty balanced binary search tree $BST(x, y)$ with every actual edge (x, y) in T_0. Associate a linear search tree $LST(x)$ with every node x in T_0. Initially, $LST(x)$ contains just the node itself.

Line Contraction Consider the line contraction step of iteration $i \geq 1$: If $x_2, \ldots, x_{t-1}$ is a path of contiguous degree-2 nodes in $T_{2(i-1)}$ bounded on each side by non-degree-2 nodes x_1 and x_t respectively, we contract this path into a balanced binary search tree $BST(x_1, x_t)$ over the nodes $x_2, \ldots, x_{t-1}$. The result of the path contraction is an edge labeled (x_1, x_t). This edge yields a dynamic edge query.

Leaf Contraction Consider the leaf contraction step of iteration $i \geq 1$: If $y_1, \ldots, y_t$ are all degree-1 nodes in T_{2i-1} adjacent to a node x in T_{2i-1}, we contract them into the linear search tree $LST(x)$ associated with x. Each node y_j contracted into x adds a dynamic edge query (x, y_j) to $LST(x)$. If nodes were already contracted into $LST(x)$ from a previous iteration, we add the new edge queries to the front (top) of the LST.

After m iterations we are left with $\mathcal{T} = T_{2m}$ which is a single node. This node is the root of the LINE-LEAF TREE.

Searching a LINE-LEAF TREE for an element u is tantamount to searching the component search structures. A search begins with $LST(x)$ where x is the root of $\mathcal{T}$. Searching $LST(x)$ with respect to u serially questions the edge queries in the sequence. Starting with the first edge query, if (x, y) answers X then we move onto the next query (x, z) in the sequence. If the query answers HERE then we proceed by searching for u in $BST(x, y)$. If it answers Y, then we proceed by searching for u in $LST(y)$. If there are no more edge queries left in $LST(x)$, then we return the actual element x. When searching $BST(x, y)$, if we ever receive a HERE response to the edge query (a, b), we proceed by searching for u in $BST(a, b)$. That is, we leave the current BST and search in a new BST. If the binary search concludes with a node x, then we proceed by searching $LST(x)$. Searching an empty BST returns NIL.

Implementation Details The LINE-LEAF TREE is an index into H_S but not a replacement for H_S. That is, we maintain a separate DAG data structure for H_S across insertions and deletions into S. This allows us, for example, to easily identify the predecessor and successors of a node $x \in S$ once we've used the LINE-LEAF TREE to find x in H_S. The edges of H_S also play an essential role in the implementation of the LINE-LEAF TREE. Namely, an edge query (x, y) is actually two pointers: $\lambda_1(x, y)$ which points to the edge (x, a) and $\lambda_2(x, y)$ which points to the edge (b, y). Here (x, a) and (b, y) are the actual edges bookending

the undirected path between x and y in T_S. This allows us to take an actual edge (x, a) in memory, rename x to w, and indirectly update all edge queries (x, z) to (w, z) in constant time. Here the path from z to x runs through a. Note that we are not touching the pointers involved in each edge query (x, z), but rather, the actual edge in memory to which the edge query is pointing.

Edge queries are created through line contractions so when we create the binary search tree $BST(x, y)$ for the path $x, a, \ldots, b, y$, we let $\lambda_1(x, y) = \lambda_1(x, a)$ and $\lambda_2(x, y) = \lambda_2(b, y)$. We assume that every edge query (x, y) corresponding to an actual edge (x', y') has $\lambda_1(x, y) = \lambda_2(x, y) = (x', y')$.

Node Properties We associate two properties with each node in S. The *round* of a node x is the iteration i where x was contracted into either an LST or a BST. We say $\textsc{round}(x) = i$. The *type* of a node represents the step where the node was contracted. If node x was *line contracted*, we say $\textsc{type}(x) = \textsc{line}$, otherwise we say $\textsc{type}(x) = \textsc{leaf}$.

In addition to ROUND and TYPE, we assume that both the linear and binary search structures provide a PARENT method that operates in time proportional to the height of the respective data structure and yields either a node (in the case of a leaf contraction) or an edge query (in the case of a line contraction). More specifically, if node x is leaf contracted into $LST(a)$ then $\textsc{parent}(x) = a$. If node x is line contracted into $BST(a, b)$ then $\textsc{parent}(x) = (a, b)$. We emphasize that the PARENT operation here refers to the Line-Leaf Tree and not T_S. Collectively, the ROUND, TYPE, and PARENT of a node help us recreate the contraction process when inserting or removing a node from S.

Approximation Ratio The following theorem gives the main properties of the static construction.

Theorem 1. *The worst-case height of a* Line-Leaf Tree $\mathcal{T}$ *for a tree-like* S *is* $\Theta(\log w) \cdot OPT$ *where* w *is the width of* S *and* OPT *is the height of an optimal search tree for* S*. In addition, given* H_S*,* $\mathcal{T}$ *can be built in* $O(n)$ *time and space.*

Proof. We prove the upper bound here and leave the tight example to the full version [10]. We begin with some lower bounds on OPT.

Claim. $OPT \geq \max\{\Delta(S), \log n, \log D, \log w\}$ where $\Delta(S)$ is the maximum degree of a node in T_S, n is the size of S, D is the diameter of T_S and w is the width of S.

Proof. Let x be a node of highest degree $\Delta(S)$ in T_S. Then, to find x in the T_S we require at least $\Delta(S)$ queries, one for each edge adjacent to x [11]. This implies $OPT \geq \Delta(S)$. Also, since querying any edge reduces the problem space left to search by at most a half, we have $OPT \geq \log n$. Because n is an upper bound on both the width w of S and D, the diameter of T_S we obtain the final two lower bounds. □

Recall that the width w of S is the number of leaves in T_S. Each round in the contraction process reduces the number of remaining leaves by at least half:

round i starts with a tree T_{2i} on n_i nodes with w_i leaves. A line-contraction produces a tree T_{2i+1}, still with w_i leaves. Because T_{2i+1} is full, the number of nodes neighboring a leaf is at most $w_i/2$. Round i completes with a leaf contraction that removes all w_i leaves, producing T_{2i+2}. As every leaf in T_{2i+2} corresponds to an internal node of T_{2i+1} adjacent to a leaf, T_{2i+2} has at most $w_i/2$ leaves. It follows that the number of rounds is at most $\log w$. The length of any root-to-leaf path is bounded in terms of the number of rounds. The following lemma follows from the construction.

Lemma 1. *On any root-to-leaf path in the* Line-Leaf Tree *there is at most one BST and one LST for each iteration i of the construction algorithm.*

For each LST we perform at most $\Delta(S)$ queries. In each BST we ask at most $O(\log D)$ questions. By the previous lemma, since we search at most one BST and one LST for each iteration i of the contraction process and since there at most $\log w$ iterations, it follows that the height of the Line-Leaf Tree is bounded above by: $(\Delta(S) + O(\log D)) \log w = O(\log w) \cdot OPT$. We now prove the time and space bounds. Consider the line contraction step at iteration i: we traverse $T_{2(i-1)}$, labeling paths of contiguous degree-2 nodes and then traverse the tree again and form balanced BSTs over all the paths. Since constructing balanced BSTs is a linear time operation, we can perform a complete line contraction step in time proportional to the size of size of $T_{2(i-1)}$. Now consider the leaf contraction step at iteration i: We add each leaf in T_{2i-1} to the LST corresponding to its remaining neighbor. This operation is also linear in the size of T_{2i-1}. Since we know the size of T_{2i} is halved after each iteration, starting with n nodes in T_0, the total number of operations performed is $\sum_{i=0}^{\log n} O(\frac{n}{2^i}) = O(n)$. Given that the construction takes at most $O(n)$ time, the resulting data structure occupies at most $O(n)$ space. □

4 Operations

Test Membership. To test whether an element $A \in \mathcal{U}$ appears in $\mathcal{T}$, we search for A in $LST(x)$ where x is the root of $\mathcal{T}$. The search ends when we reach a terminal node. The only terminal nodes in the Line-Leaf Tree are either leaves representing the elements of S or Nil (which are empty BSTs). So, if we find A in $\mathcal{T}$ then test membership returns True, otherwise it returns False. Given that test membership follows a root-to-leaf path in $\mathcal{T}$, the previous discussion constitutes a proof of the following theorem.

Theorem 2. Test Membership *is correct and takes $O(h)$ time.*

Predecessor. Property 1 guarantees that each node $A \in \mathcal{U}$ has exactly one predecessor in S. Finding the predecessor of A in S is similar to test membership. We search $\mathcal{T}$ until we find either A or Nil. Traditionally if A appears in a set then it is its own predecessor, so, in the first case we simply return A. In the latter case, A is not in $\mathcal{T}$ and Nil corresponds to an empty binary search tree $BST(y, z)$ for the actual edge (y, z) where, say, $y \prec z$. We know that A

falls between y and z (and potentially between y and some other nodes) so y is the predecessor of A. We return y. Given that PREDECESSOR also follows a root-to-leaf path in $\mathcal{T}$, the previous discussion yields a proof of the following theorem.

Theorem 3. PREDECESSOR *is correct and takes* $O(h)$ *time.*

Insert. Let $A \notin S$ be the node we wish to insert in $\mathcal{T}$ and let $S' = S \cup \{A\}$. Our goal is to transform $\mathcal{T}$ into $\mathcal{T}'$ where $\mathcal{T}'$ is the search tree produced by the contraction process when started on $T_{S'}$. We refer to this transformation as *correcting* the LINE-LEAF TREE and divide INSERT into three corrective steps: *local correction*, *down correction*, and *up correction*. Local correction repairs the contraction process on $\mathcal{T}$ for elements of S that appear near A at some point during the contraction process. Down correction repairs $\mathcal{T}$ for nodes with round at most ROUND(A). Up correction repairs $\mathcal{T}$ for nodes with round at least ROUND(A). Our primary result is the following theorem.

Theorem 4. INSERT *is correct and takes* $O(h)$ *time.*

A full proof of Theorem 4 appears in the full version [10]. Here we give a detailed outline of the insertion procedure. Let X be a node such that $LST(X)$ has t edge queries $(X, Y_1) \dots (X, Y_t)$ sorted in descending order by ROUND(Y_i). That is, Y_1 is the last node leaf-contracted into $LST(X)$, Y_t is the first node leaf-contracted into $LST(X)$ and Y_i is the $(t-i+1)^{th}$ node contracted into $LST(X)$. Define $\rho_i(X) = Y_i$ and $\mu_i(X) =$ ROUND(Y_i). If $i > t$, then let $\mu_i(X) = 0$.

Local Correction. We start by finding the predecessor of A in T_S. Call this node B. In $H_{S'}$, A potentially falls between B and any number of $children(B)$. Thus, A may replace B as the parent of a set of nodes $D \subseteq children(B)$. We use D to identify two other sets of nodes C and L. The set C represents nodes that, in T_S, were leaf-contracted into B in the direction of some edge (B, D_j) where $D_j \in D$. The set L represents nodes that were involved in the contraction process of B itself. Depending on TYPE(B), the composition of L falls into one of the following two cases:

1. if TYPE(B) = LINE then let PARENT$(B) = (E, F)$. Let D_E and D_F be the two neighbors of B on the path from E to F. If D_E and D_F are in D then $L = \{E, F\}$. If only D_E is in D, then $L = \{E\}$. If only D_F is in D, then $L = \{F\}$. Otherwise, $L = \emptyset$.
2. If TYPE(B) = LEAF then let PARENT$(B) = E$. Let D_E be the neighbor of B on the path $B \dots E$. Let $L = \{E\}$ if D_E is in D and let $L = \emptyset$ otherwise.

If C and L are both empty, then A appears as a leaf in $T_{S'}$ and ROUND$(A) = 1$. In this case, we only need to correct $\mathcal{T}$ upward since the addition of A does not affect nodes contracted in earlier rounds. However, if either C or L is non-empty, then A is an interior node in $T_{S'}$ and A essentially acts as B to the stolen nodes in C. Thus, for every edge query (B, C_i) where $C_i \in C$, we remove (B, C_i) from

$LST(B)$ and insert it into $LST(A)$. In addition, we create a new edge (B, A) and add it to H_S which yields $H_{S'}$. This ends local correction.

Removing edge queries from $LST(B)$ and inserting them into $LST(A)$ may cause changes in the contraction process that reverberate upward and downward in the Line-Leaf Tree. Let $P = A$ and $Q = B$ when $\text{ROUND}(A) \leq \text{ROUND}(B)$ and let $P = B$ and $Q = A$ otherwise. Broadly, there are two interesting cases. If $\mu_1(P) \neq \mu_2(P)$ then P was potentially line contracted between $\rho_1(P)$ and Q at some earlier round. If this is the case then we must correct the contraction process *downward* on $BST(\rho_1(P), P)$ and $BST(P, Q)$. Likewise, when $\mu_1(P) = \mu_2(P)$ then $\text{ROUND}(Q)$ might increase, which in turn may affect later rounds of the contraction process. If this is the case then we must correct the contraction process *upward* on Q.

Down Correction. Here we know that P was line contracted between $\rho_1(P)$ and Q at some earlier round. The main idea of Down Correct is to float P down to the BST created in the same round as P. We do this by examining the rounds when $BST(\rho_1(P), P)$ and $BST(P, Q)$ were created and recursively calling Down Correct until we arrive at the BST with correct round.

Up Correction. In this case, we know that P increases the round of Q by one which can affect the contraction process for nodes contracted in later rounds. If Q was leaf-contracted into E (*i.e.*, $\text{TYPE}(Q) = \text{LEAF}$ and $\text{PARENT}(Q) = E$) then P replaces Q in the edge query (Q, E) since Q is now line-contracted between P and E in the iteration before. If Q was line-contracted into $BST(E, F)$ (*i.e*, $\text{TYPE}(Q) = \text{LINE}$ and $\text{PARENT}(Q) = (E, F)$) then $BST(E, F)$ is now split into $BST(E, Q)$ and $BST(Q, F)$. The interesting case is when, in $\mathcal{T}$, E was leaf-contracted into F. In $\mathcal{T}'$, the edge query (E, Q) now appears in $LST(Q)$ and we're in a position to recursively correct the contraction process upwards with Q and F replacing P and Q respectively in the recursive call.

Delete. Deletion removes a node A from a Line-Leaf Tree $\mathcal{T}$ assuming A appears in $\mathcal{T}$. As with insertion, the goal is to repair $\mathcal{T}$ so that it mimics $\mathcal{T}'$ where $\mathcal{T}'$ is the result of running the contraction process on $T_{S'}$ where $S' = S \setminus \{A\}$. Deletion is a somewhat simpler operation than insertion. This is because when we delete A, *all* of the successors of A become successors of A's predecessor B. If A outlasted B in the new contraction process, then B essentially plays the role of A in $\mathcal{T}'$. If B outlasted A, then its role does not change. The only problem is that B no longer has A as a neighbor which may create problems with nodes contracted later in the process. Repairing these problems is the technical crux of deletion. A thorough description of deletion, as well as a proof of the following Theorem also appear in the full version [10].

Theorem 5. Delete *is correct and takes* $O(\log w) \cdot OPT$ *time.*

5 Empirical Results

To conclude, we compare the height of a Line-Leaf Tree to the height of an optimal static search tree in two experimental settings: random tree-like partial

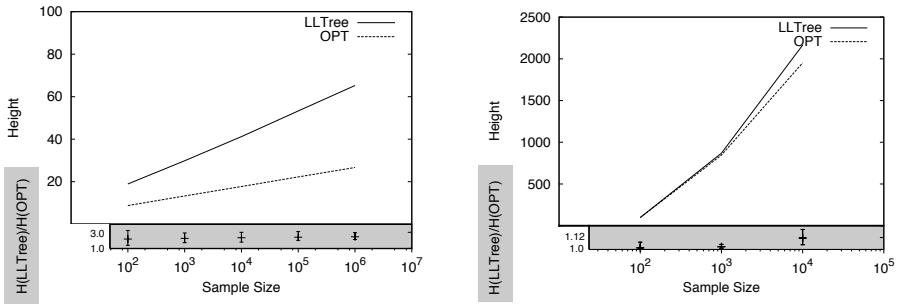

Fig. 4. Results comparing the height of the Line-Leaf Tree to the optimal static search search tree on (a) random tree-like partial orders; and (b) a large portion of the UNIX filesystem. The non-shaded areas show the average height of both the Line-Leaf Tree and optimal static algorithm. The shaded area shows their ratio (as well as the min and max values over the 1000 iterations).

orders and the UNIX directory structure. For these experiments, we consider the height of a search tree to be the maximum number of edge queries performed on any root-to-leaf path. So any dynamic edge query in a Line-Leaf Tree counts as two edge queries in our experiments.

In the first experiment, we examine tree-like partial orders of increasing size n. For each n, we independently sample 1000 partial-orders uniformly at random from all tree-like partial orders with n nodes [12] (this distributions give a tree of height $\theta(\log n)$, w.h.p. [13–15]). The non-shaded area of Figure 4 (a) shows the heights of the Line-Leaf Tree and the optimal static tree averaged over the samples. The important thing to note is that both appear to grow linearly in $\log n$. We suspect that the differing slopes come mainly from the overheard of dynamic edge queries, and we conjecture that the Line-Leaf Tree performs within a small constant factor of OPT with high probability in the uniform tree-like model. The shaded area of Figure 4 (a) shows the average, minimum, and maximum approximation ratio over the samples.

Although the first experiment shows that the Line-Leaf Tree is competitive with the optimal static tree on *average* tree-like partial orders, it may be that, in practice, tree-like partial orders are distributed non-uniformly. Thus, for our second experiment, we took the `/usr` directory of an Ubuntu 10.04 Linux distribution as our universe $\mathcal{U}$ and independently sampled 1000 sets of size $n = 100$, $n = 1000$, and $n = 10000$ from $\mathcal{U}$ respectively. The `/usr` directory contains 23,328 nodes, of which 17,340 are leaves. The largest directory is `/usr/share/doc` which contains 1551 files. The height of `/usr` is 12. We believe that this directory is somewhat representative of the use cases found in our motivation. As with our first experiment, the shaded area in Figure 4 (b) shows the ratio of the height of the Line-Leaf Tree to the height of the optimal static search tree, averaged over all 1000 samples for each sample size. The non-shaded

area shows the actual heights averaged over the samples. The Line-Leaf Tree is again very competitive with the optimal static search tree, performing at most a small constant factor more queries than the optimal search tree.

Acknowledgements We would like to thank T. Andrew Lorenzen for his help in running the experiments discussed in Section 5.

References

1. Ben-Asher, Y., Farchi, E., Newman, I.: Optimal search in trees. SIAM J. Comput. **28** (1999) 2090–2102
2. Carmo, R., Donadelli, J., Kohayakawa, Y., Laber, E.S.: Searching in random partially ordered sets. Theor. Comput. Sci. **321** (2004) 41–57
3. Mozes, S., Onak, K., Weimann, O.: Finding an optimal tree searching strategy in linear time. In: SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2008) 1096–1105
4. Onak, K., Parys, P.: Generalization of binary search: Searching in trees and forest-like partial orders. In: FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (2006) 379–388
5. Dereniowski, D.: Edge ranking and searching in partial orders. Discrete Appl. Math. **156** (2008) 2493–2500
6. Jacobs, T., Cicalese, F., Laber, E.S., Molinaro, M.: On the complexity of searching in trees: Average-case minimization. In: ICALP 2010. (2010) 527–539
7. Laber, E., Molinaro, M.: An approximation algorithm for binary searching in trees. In: ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Berlin, Heidelberg, Springer-Verlag (2008) 459–471
8. Daskalakis, C., Karp, R.M., Mossel, E., Riesenfeld, S., Verbin, E.: Sorting and selection in posets. In: SODA '09: Proceedings of the Nineteenth Annual ACM-SIAM SODA, Philadelphia, PA, USA, SIAM (2009) 392–401
9. Daskalakis, C., Karp, R.M., Mossel, E., Riesenfeld, S., Verbin, E.: Sorting and selection in posets. CoRR **abs/0707.1532** (2007)
10. Heeringa, B., Iordan, M.C., Theran, L.: Searching in dynamic tree-like partial orders. CoRR **abs/1010.1316** (2010)
11. Laber, E., Nogueira, L.T.: Fast searching in trees. Electronic Notes in Discrete Mathematics **7** (2001) 1–4
12. Meir, A., Moon, J.W.: On the altitude of nodes in random trees. Canadian Journal of Mathematics **30** (1978) 997–1015
13. Bergeron, F., Flajolet, P., Salvy, B.: Varieties of increasing trees. In: CAAP '92: Proceedings of the 17th Colloquium on Trees in Algebra and Programming, London, UK, Springer-Verlag (1992) 24–48
14. Drmota, M.: The height of increasing trees. Annals of Combinatorics **12** (2009) 373–402 10.1007/s00026-009-0009-x.
15. Grimmett, G.R.: Random labelled trees and their branching networks. J. Austral. Math. Soc. Ser. A **30** (1980/81) 229–237