UNIQUE BINARY SEARCH TREE REPRESENTATIONS AND EQUALITY-TESTING OF SETS AND SEQUENCES

Rajamani Sundar
Robert E. Tarjan

CS-TR-267-90

November 1989

Unique Binary Search Tree Representations and Equality-testing of Sets and Sequences

Rajamani Sundar*
Courant Institute of Mathematical Sciences
New York University

Robert E. Tarjan†
Department of Computer Science
Princeton University
and
AT&T Bell Laboratories

November 1989

Abstract

Given an ordered universe U, we study the problem of representing each subset of U by a unique binary search tree so that dictionary operations can be performed efficiently. We exhibit representations that permit the execution of dictionary operations in optimal time when the dictionary is sufficiently sparse or sufficiently dense. We apply unique representations to obtain efficient data structures for maintaining a collection of sets/sequences under queries that test the equality of a pair of objects. In the process, we devise an interesting method for maintaining a dynamic, sparse array.

*This work was begun when the author was at Bellcore and continued at DIMACS center and Courant Institute. Research partially supported by an IBM graduate fellowship and NSF grants CCR-8702271, CCR-8902221, CCR-8906949 and ONR grant N00014-85-K-0046.

†Research partially supported by NSF grant DCR-8605962 and ONR grant N00014-87-K-0467.

1 Introduction

The *unique representation problem* for an abstract data type (e.g. a dictionary) is to design an unique representation for values of the abstract data type by values of a concrete data type (e.g. a binary search tree) so that operations of the abstract data type can be performed efficiently. More concretely, consider the problem of uniquely representing a dictionary over an ordered universe by a binary search tree. The cost of searching for an item in the dictionary equals its depth in the corresponding binary search tree. The cost of performing an update operation (insert or delete) on the dictionary is the time needed to transform the tree corresponding to the initial set into the tree corresponding to the final set. Let $T(n)$ denote the worst-case cost of performing a dictionary operation on an n-element dictionary under this representation. The problem is to choose a representation that minimizes $T(n)$ for all values of n.

The unique representation problem arises in the contexts of incremental evaluation and implementation of high level programming languages. Incremental evaluation [11,12] is the technique of efficiently updating the value of a function when the input changes. A simple idea to speed up incremental evaluation is to remember the results of previous function calls and thereby avoid recomputation. If the input domain is uniquely represented and input data objects are constructed using CONS operations [2], then it becomes easy to check if the function has already been evaluated on a given input. Modern programming languages such as SETL support high level data types such as sets and sequences and permit testing whether two such objects are equal as a fundamental operation. Equality-testing can be implemented in constant time by devising an unique representation for the data type and manipulating concrete data values through CONS operations.

Before proceeding further, we need to introduce some basic terminology. A *dictionary* is a set of items selected from a totally ordered universe on which membership queries and update operations that insert or delete items can be performed. We can represent a dictionary by a binary tree containing one item per node, with the items arranged in symmetric order: each item in the tree is larger than the items in its left subtree and smaller than the items in its right subtree. This data structure is called a *binary search tree.* A binary search tree is represented in storage as a collection of records, one per node, with a pointer to the record corresponding to the root. Each record has a key field, left and right children pointers, and a bounded number of additional information and pointer fields. A *rotation* of an edge $[x,p]$ in a binary tree, where p is the parent of x, is a transformation that makes x the parent of p by transferring one of the subtrees of x to p. See Figure 1. Rotations are useful in updating binary search trees since they preserve the symmetric order of the items.

Now, we survey previous work on unique representations. Snyder [15] considers the problem of representing subsets of an ordered universe by tree-based search structures so that all sets of equal cardinality have the same underlying tree. For this class of representations he showed that $\Theta(\sqrt{n})$ time is both necessary and sufficient to perform a dictionary operation, where n is the dictionary size. Munro and Suwanda [10] examine how to implicitly represent subsets of an ordered universe by enforcing, for each set size, an unique partial order on the storage locations. They showed that $\Theta(\sqrt{n})$ time is necessary and sufficient to perform dictionary operations using such a representation. Recently, researchers have started using randomization to obtain solutions to the unique representation problem with provably good average behaviour. Pugh [11] and Pugh and Tietelbaum [12] have devised efficient randomized representations for sets, sequences and other abstract data types. Aragon and Seidel [3] show how to uniquely represent a dictionary by a binary search tree so that dictionary operations can be performed in $O(\log n)$ randomized time.

In this paper we study the deterministic complexity of uniquely representing a dictionary by a binary search tree. We consider the following three ways of updating a binary search tree during an insertion or deletion:

1. Performing a sequence of rotations: We update the tree using the traditional algorithms for binary search tree insertion and deletion [8], but permit arbitrary rotations to be performed on the tree before and after so that the final tree correctly represents the updated dictionary.

2. Changing the pointers of a set of nodes.

3. Nondestructive updating through CONS operations: At any time during the sequence of dictionary operations, we maintain all the trees created so far and their subtrees. An update operation constructs a new tree by sharing subtrees from existing trees and creating new nodes corresponding to nonexistent subtrees. A CONS operation is used to determine whether a subtree already exists, given its left and right subtrees and root item, and create the subtree if necessary. The cost of a CONS operation is assumed to be constant. Also, in anticipation of the future, the update operation may create any number of additional trees through CONS operations.

For each updating method, we devise representations that permit efficient implementation of the dictionary using that method. Culik and Wood [4] have proved that the number of rotations needed to transform any n-node binary tree into any other n-node binary tree is at most $2n-2$, and Sleator, Tarjan, and Thurston [14] have improved this bound to $2n-6$, for $n > 12$. These results imply that any unique representation supports dictionary operations in $O(n)$ time using rotations. Snyder's representation [15] requires $O(\sqrt{n})$ time per dictionary operation and uses pointer changes to update the tree. We provide a representation that supports dictionary operations in $O(\sqrt{n})$ time using CONS operations.

We also prove lower bounds on the complexity of any unique representation that are valid if the dictionary is sufficiently sparse (i.e., if n, the dictionary size, is small in relation to $|U|$). The lower bounds are derived for the following cost metric. The *cost* of an update using one of the above methods is, respectively, the number of rotations, the number of pointer changes, and the number of newly created nodes. The *cost* of searching for an item in a binary search tree is 1 plus the depth of the item in the tree. Under the assumption that the dictionary is sparse, we show that the preceding upper bounds are optimal. In contrast, Aragon and Seidel's representation requires only $O(\log n)$ randomized time per dictionary operation using any of the three updating methods. It is surprising that this problem has such widely differing deterministic and randomized complexities. We show that the sparseness assumption in the lower bounds is essential by constructing a representation that is optimal on dense dictionaries. It requires $O(\log |U|)$ time per dictionary operation using any update strategy.

Next, we apply unique representations to the problems of equality-testing of sets and sequences:

Sequence equality-testing problem: Maintain a collection of sequences over an ordered universe under the operations i) EQUAL(S,T) - Return if sequences S and T are equal, ii) INSERT(S,i,x,T) - Create a new sequence T by inserting element x into sequence S between positions $i-1$ and i, and iii) DELETE(S,i,T) - Create a new sequence T by deleting the ith element of sequence S. Initially, the collection consists of only the null sequence.

Set equality-testing problem: Maintain a collection of sets over an ordered universe under the operations EQUAL(S,T), INSERT(S,x,T), and DELETE(S,x,T), defined in the obvious fashion. Initially, the collection consists of only the empty set.

For both problems, we are interested only in data structures that test equality in constant time. We now describe previous work on these problems. Carter and Wegman [17] proposed a randomized signature-based scheme for set equality-testing that requires only constant time per operation but may declare erroneously that two unequal sets are equal with a small probability. Pugh [11] and Pugh and Tietelbaum [12] gave randomized solutions to both problems that require $O(\log n)$ expected time and $O(\log n)$ expected space per update operation, where n denotes the size of the set or sequence. Their data structures also support more powerful operations (for instance, union and intersection of sets) efficiently. However, for sequence equality-testing, the logarithmic bound is valid only if the sequences do not consist of repeated elements. Yellin [19] described two deterministic solutions for equality-testing of sets, both of which use space prohibitively. His solutions assume that the collection of sets is fixed and that the sets are updated destructively.

We obtain the following results on these problems. A straightforward solution to set equality-testing is to represent sets by binary tries and use CONS operations to update a trie. The solution requires $O((\log m)^2)$ time and $O(\log m)$ space per update, where m is the total number of operations. We reduce the time per update in this solution to $O((\log m)^{3/2})$ (amortized) by developing a technique for maintaining a dynamic sparse array efficiently. For any fixed $\epsilon > 0$, this method yields a data structure for set equality-testing requiring $O(\log m)$ time and $O(m^\epsilon)$ space per update. For sequence equality-testing, we propose two solutions. One solution requires $O(\sqrt{n \log m})$ amortized time and $O(\sqrt{n})$ amortized space per update, while the other requires $O(\sqrt{n}(\log m)^{1/4} + \log m)$ amortized time and $O(\sqrt{n}(\log m)^{1/4})$ amortized space. Here, n denotes the size of the sequence being updated and m denotes the total number of operations. The first solution is faster or slower than the second, depending on whether $m > 2^n$ or not. The update time in the first solution can be further reduced to $O(\sqrt{n})$ by increasing the space required per update to $O(\sqrt{n}m^\epsilon)$.

We note an interesting connection between the two problems. When sequences are free from repetitions, sequence equality-testing is equivalent to set equality-testing. This follows from the representation of sets by sorted sequences and the representation of sequences by sets of ordered pairs of adjacent elements. Therefore, sequence equality-testing becomes truly hard only when sequences have repetition.

The above data structures are based on a method of maintaining a dynamic sparse array efficiently.

Dynamic sparse array problem: Maintain an array of size N, all of whose entries are initially 0, efficiently under queries and updates of entries. The total number of updates is assumed to be very small relative to N.

Dynamic perfect hashing [5] gives a randomized solution requiring $O(1)$ time per query and $O(1)$ randomized amortized time and $O(1)$ amortized space per update. Tarjan and Yao [16] give a deterministic solution to the problem when the array is static, but, even then, their preprocessing time (quadratic in the number of nonzero entries) is too expensive for our applications. We propose a solution to the problem that requires $O(\sqrt{\log N})$ time per query and $O(\sqrt{\log N})$ amortized time and $O(1)$ amortized space per update. This implies that the time required by our method to store a sparse array of size n^2 having n nonzero entries is $O(n\sqrt{\log n})$, while the space required is $O(n)$. In contrast, Tarjan and Yao's static method requires $O(n^2)$ time and $O(n)$ space, but supports constant time queries. The time per operation in our solution can be reduced to $O(1)$ by increasing the space per update to $O(N^\epsilon)$.

The paper is organized as follows. In Section 2, we describe the optimal representation for sparse dictionaries that uses CONS operations. The lower bounds for representing sparse dictionaries are established in Section 3. Section 4 contains the optimal representation for dense dictionaries. In sections 5, 6, and 7, respectively, we describe the data structures for dynamic sparse arrays and set and sequence equality-testing. The last section concludes by posing open problems.

2 An optimal representation for sparse dictionaries

We describe a representation that allows search in $O(\log n)$ time and permits updating the binary search tree in $O(\sqrt{n})$ time using Cons operations. We represent the dictionary by an almost complete binary search tree. Specifically, if $n = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_k}$, such that $i_1 > i_2 > \cdots > i_k$, then the binary search tree representing an n-element dictionary comprises a k-node right path, with a sequence of k complete binary trees of respective sizes $2^{i_1} - 1, 2^{i_2} - 1, \ldots, 2^{i_k} - 1$ hanging off of it. See Figure 2a. In order to be able to update the tree efficiently, we need to maintain some additional information. Define a *j-run* to be a subset of $2^j - 1$ adjacent elements in the dictionary. For each $j \leq \lfloor \log n/2 \rfloor$, we maintain complete binary trees representing the j-runs of the dictionary, called the *j-trees*, in a sorted, doubly linked list. See Figures 2b and 2c. When n has the form $2^{2i} - 2^i + k$, where $1 \leq k < 2^i$, in addition, we also maintain the list of i-trees ($i = \lfloor \log n/2 \rfloor + 1$) corresponding to the first $k2^i$ i-runs. When an insertion causes n to increase from $2^{2i} - 1$ to 2^{2i}, this list becomes the list of $\lfloor \log n/2 \rfloor$-trees.

It is obvious that this representation supports search in $O(\log n)$ time. To insert a new element x, we first update the lists of trees, proceeding bottom up, and then create the rest of the tree from trees in the updated lists. Updating the list of j-trees, for any j, involves deleting at most $2^j - 2$ j-trees corresponding to old j-runs that contain the predecessor as well as the successor of x and inserting in their place at most $2^j - 1$ j-trees corresponding to new j-runs that contain x. Creating a new j-tree is accomplished in $O(1)$ time by performing a Cons operation on two $(j-1)$-trees that are its subtrees and its root item. In addition, if n is of the form $2^{2i} - 2^i + k$, where $1 \leq k < 2^i$, we need to extend the size of the list of i-trees from $k2^i$ to $(k+1)2^i$ by creating at most 2^i new i-trees. The foregoing discussion shows that the list of j-trees, for any j, may be updated in $O(2^j)$ time once the updated list of $(j-1)$-trees is available. The total cost of updating the lists of trees is at most $O(2 + 2^2 + \cdots + 2^{\lfloor \log n/2 \rfloor + 1}) = O(\sqrt{n})$. To create the rest of the tree, we need to create the nodes of the tree at levels $\lfloor \log n/2 \rfloor + 1, \lfloor \log n/2 \rfloor + 2, \ldots, \lfloor \log n \rfloor + 1$ and the nodes on the right path of the tree. These nodes can be created bottom up from the updated lists of trees by performing $O(\sqrt{n})$ Cons operations. It follows that the cost of an insertion is $O(\sqrt{n})$. Deletion is analogous.

This completes the description of our representation. The scheme has the drawback of using $\Omega(\sqrt{n})$ space per update. However, this is inevitable, for we will see in the next section that any unique binary search tree representation suffers from this defect if the dictionary is sparse.

3 Lower bounds for sparse dictionaries

Snyder's lower bound [15] of $\Omega(\sqrt{n})$ on the cost of a dictionary operation is applicable when sets of any particular cardinality are represented by a single underlying multiway tree and pointer changes are used to update the tree. Our lower bounds apply to the class of unique binary search tree representations in which equal-cardinality sets are allowed to have different underlying binary trees. We prove the lower bounds through an interesting application of Ramsey's theorem [7,9]. This method can be used to show that Snyder's lower bound holds even if his restriction regarding equal cardinality sets is removed. We need the following version of Ramsey's theorem to state and to prove the lower bound results:

Ramsey's theorem *Let n, k and $s \geq n$ be arbitrary positive integers and let U be an arbitrary set. There exists a number $R_n(k, s)$ with the following property: if $|U| \geq R_n(k, s)$ then, for any partition* *of the n-subsets of U into k classes, there is an s-subset S all of whose n-subsets lie in a single class.*

It is known that $R_n(k,s) = T_{n+1}(O(\log ks))$ [7,9], where $T_n(x)$ is the tower function

$$\left.2^{2^{\cdot^{\cdot^{\cdot^{2^x}}}}}\right\} n+1.$$

First, we state the lower bounds for updating the tree by means of rotations and pointer changes. Let $b_n = \binom{2n}{n}/(n+1)$ denote the number of distinct n-node binary trees [8]. Let n and d be positive integers and let U be an ordered universe of size at least $R_n(b_n, n+1)$. For any unique binary search tree representation of subsets of U we have the following trade-offs between search and update times:

Theorem 1 *If the n-subsets of U are represented by binary search trees of height at most d, then there is an n-subset on which an update operation requires $\Omega(n-2d)$ rotations.*

Theorem 2 *If the n-subsets of U are represented by binary search trees of height at most d, then there is an n-subset on which an update operation requires $\Omega(n/d)$ pointer changes.*

The Ramsey number $R_n(b_n, n+1)$ is at most $T_{n+1}(cn)$, for some constant c. Therefore these trade-offs are valid when the dictionary size is at most $\log^* |U| - o(\log^* |U|)$.

Theorem 1 implies that a dictionary operation requires $\Omega(n)$ time if the dictionary is sufficiently sparse and rotations are used to update the tree. Theorem 2 implies that a dictionary operation requires $\Omega(\sqrt{n})$ time if pointer changes are used to update the tree.

These trade-offs are actually realizable. To realize the trade-off of Theorem 2, we represent an n-element set by a complete binary tree of size n/d with chains of length d each attached at the leaves. If the depth parameter d increases "smoothly" with n and $\log n \le d \le \sqrt{n}$ always, then this representation allows update operations to be performed in $O(n/d)$ time. We leave the representation that realizes the trade-off of Theorem 1 as an (easy) exercise.

The next theorem gives the lower bound on the cost of updating the tree using CONS operations.

Theorem 3 *Let n and m be positive integers and let U be an ordered universe of size at least $R_n(b_n, m+n)$. For any unique binary search tree representation of subsets of U, there is a sequence of m update operations that involves only subsets of size at most n and causes the creation of $\Omega(m\sqrt{n})$ new nodes when* CONS *operations are used to implement updates.*

We have $R_n(b_n, m+n) \le T_{n+1}(c(n + \log m))$, for some constant c. Hence the lower bound of $\Omega(\sqrt{n})$ node creations per update holds when $n \le \log^* |U| - \log^* m - o(\log^* |U|)$.

We are now ready to prove the theorems.

Proof of Theorem 1. By Ramsey's theorem, there is a subset $S = \{x_1, x_2, \ldots, x_{n+1}\}$ of U all of whose n-subsets are represented by a single underlying binary tree, say B. We claim that $\Omega(n-2d)$ rotations are necessary to transform set $S_1 = \{x_1, x_2, \ldots, x_n\}$ into set $S_2 = \{x_2, x_3, \ldots, x_{n+1}\}$ by deleting x_1 and inserting x_{n+1}. We prove the claim using two lemmas.

For any binary tree T, let T^l denote the binary tree with T as left subtree and empty right subtree. Define T^r similarly.

Lemma 1 *For any binary tree T, at least $|T|$ rotations are needed to transform T^l into T^r.*

Proof. By induction on $|T|$.

Case 1. $|T| = 1$: Easy.

Case 2. $|T| \geq 2$: The proof has the flavor of Wilber's method [18] of deriving lower bounds on rotations in binary trees. Consider an n-node binary tree B whose nodes are labeled from 1 to n in symmetric order. An interval $[i, j]$ of nodes of B is a *block* of B. Any block $[i, j]$ of B induces a binary tree, called the *block tree* of block $[i, j]$, obtained by contracting all edges of B having an endvertex outside the block. See Figure 3. A rotation on B propagates into a rotation on a block tree of B only if both nodes of the rotation lie in the block tree; otherwise the block tree is unaffected.

Let T_1 and T_2 denote the left and right subtrees of T. Divide the nodes of T^l and T^r into a left block $[1, |T_1|+1]$ and a right block $[|T_1|+2, |T|+1]$. Then the left block trees of T^l and T^r are, respectively, T_1^l and T_1^r. See Figure 4. By the inductive hypothesis, at least $|T_1|$ left block rotations must be performed to transform T_1^l into T_1^r. Similarly, $|T_2|$ right block rotations are needed to transform the right block tree T_2^l of T^l into the right block tree T_2^r of T^r. Since the roots of T^l and T^r belong to different blocks, at least one rotation involving nodes from both blocks is performed in transforming T^l into T^r. Any single rotation on T^l falls exactly into one of the three categories. Hence, in total, at least $|T_1| + |T_2| + 1 = |T|$ rotations are needed to transform T^l into T^r. □

Lemma 2 *Let r denote the number of rotations required to transform S_1 into S_2 by deleting x_1 and inserting x_{n+1}. Then B^l can be transformed into B^r in $2r + 2d + 1$ rotations.*

Proof. Suppose that S_1 is transformed into S_2 by performing the following transformations on the underlying binary tree:

$$B \xrightarrow{\sigma_1} B_1 \xrightarrow{\text{delete } x_1} B_2 \xrightarrow{\sigma_2} B_3 \xrightarrow{\text{insert } x_{n+1}} B_4 \xrightarrow{\sigma_3} B$$

Here σ_i denotes a sequence of rotations. We transform B^l into B^r as follows:

$$B^l \xrightarrow{\sigma_1} B_1^l \xrightarrow{\substack{\text{Rotate } x_1 \text{ to the}\\ \text{root}}} B_2^{lr} \xrightarrow{\sigma_2} B_3^{lr} \xrightarrow{\substack{\text{Rotate } x_{n+1} \text{ to}\\ \text{where it is}\\ \text{inserted}}} B_4^r \xrightarrow{\sigma_3} B^r$$

The number of rotations needed to move x_1 to the root is at most $|\sigma_1| + d + 1$. Likewise, x_{n+1} can be moved to the correct position in $|\sigma_3| + d$ rotations. Hence $2|\sigma_1| + |\sigma_2| + 2|\sigma_3| + 2d + 1 \leq 2r + 2d + 1$ rotations suffice to transform B^l into B^r. □

Combining the two lemmas, it follows that $\Omega(n - 2d)$ rotations are required to transform S_1 into S_2 by deleting x_1 and inserting x_{n+1}. The insertion of x_{n+1} into set $\{x_2, x_3, \ldots, x_n\}$ and the deletion of x_{n+1} from set S_2 are inverse transformations and require the same number of pointer changes. Hence $\Omega(n - 2d)$ rotations are necessary for either the deletion of x_1 from set S_1 or the deletion of x_{n+1} from set S_2. □

Proof of Theorem 2. The proof applies Ramsey's theorem on the n-subsets of U and then uses Snyder's lower bound argument [15].

By Ramsey's theorem, there is a subset $S = \{x_1, x_2, \ldots, x_{n+1}\}$ of U all of whose n-subsets are represented by the same underlying binary tree. Let L denote the subset of elements that occupy the leaves in the binary search tree representation of set $S_1 = \{x_1, x_2, \ldots, x_n\}$. Note that $|L| \geq (n-1)/d$, since a binary tree with height d and $|L|$ leaves has at most $d|L| + 1$ nodes. If we delete x_1 and insert x_{n+1} into this set, we obtain the set $S_2 = \{x_2, x_3, \ldots, x_{n+1}\}$. Since S_1 and S_2 have the same underlying binary tree, every element of $L - \{x_1\}$ occupies an internal node in the binary search tree representation of S_2. The same argument shows that the leaf elements of S_2 occupy the internal nodes of the binary search tree for S_1. Therefore at least $2|L| - 2$ pointer changes are needed to transform S_1 into S_2. It follows that at least $|L| - 1 = \Omega(n/d)$ pointer changes must be performed while either deleting x_1 from set S_1 or inserting x_{n+1} into set $\{x_2, x_3, \dots, x_n\}$. The theorem is now proved in the same manner as Theorem 1. □

Proof of Theorem 3. By Ramsey's theorem, there is a subset $S = \{x_1, x_2, \dots, x_{m+n}\}$ of U all of whose n-subsets are represented by the same underlying binary tree. Let T denote an n-subset of S, to be specified later, and let B be a variable that initially denotes the binary search tree corresponding to T. The lower bound construction repeats the following cycle of operations on B until m updates have been performed on the tree:

Cycle.

1. Repeat $\sqrt{n}$ times:
   Delete the smallest element from B and insert a 'fresh element' from S that is larger than all existing elements of B. Throughout this construction, by 'fresh element', we mean an element that has never been in the tree before.
2. Call a node of B *heavy* if it has $\sqrt{n}$ or more descendents, and *minimally heavy* if it is heavy and has no proper heavy descendents. Let $y_1, y_2, \dots, y_k$ and $z_1, z_2, \dots, z_k$ denote, respectively, the minimally heavy nodes of B and their respective rightmost descendents. Replace the elements stored in nodes $z_1, z_2, \dots, z_k$ by fresh elements from S that occupy the same positions in the linear ordering of the elements of the dictionary.

For the lower bound construction to be valid, it is essential to correctly choose the elements of the initial set T and the fresh elements that are inserted into the tree each time. To make a proper choice for these elements, we first determine their relative linear ordering by executing the construction, treating the elements as unknowns. Next, we choose the elements from S, satisfying their relative ordering.

We claim that each iteration of the cycle performs at most $4\sqrt{n}$ update operations on the tree and causes the creation of at least $n - \sqrt{n}$ new nodes. The first part of the claim is trivial since each step of the cycle performs at most $2\sqrt{n}$ updates. The second part follows from two straightforward facts:

1. There are at least $\sqrt{n} - 1$ heavy nodes in any n-node binary tree.
2. Consider any iteration of Step 1 of the cycle. The subtrees of the heavy nodes of B after the iteration are all different from the subtrees of B at any time prior to the iteration. Therefore the iteration recreates all heavy nodes of B by performing Cons operations.

The theorem follows immediately from the claim. □

4 An Optimal representation for dense dictionaries

We begin with the observation that Aragon and Seidel's representation [3] always creates trees of height $\sqrt{|U|}$. They choose a static, random priority for each element in the universe and define the representation of a set of items to be the binary search tree that is heap-ordered according to priorities. It is well known [9] that any sequence of $p^2 + 1$ numbers contains a monotone subsequence of length

$p+1$. If we choose the longest monotone subsequence of the sequence of priorities of elements in the universe, then the representation of the set corresponding to this subsequence has height at least $\sqrt{|U|}$. Therefore, their representation always requires $\Omega(\sqrt{|U|})$ time in the worst case to perform a dictionary operation. Further, since it is possible to assign priorities to elements of the universe so that the longest monotone subsequence has length at most $\sqrt{|U|}+1$, there is a static priority representation in which the trees have height $O(\sqrt{|U|})$. This representation allows a dictionary operation to be performed in $O(\sqrt{|U|})$ time.

We shall now describe a more efficient representation that allows a dictionary operation to be performed in $O(\log|U|)$ time. To aid the description of our representation it is convenient to assume that $U = [1, 2^p]$. The representation is reminiscent of binary tries [8]. Let S denote the set being represented. If $S \subseteq [1, 2^{p-1}]$, then S is represented in U by the tree representing it in the universe $[1, 2^{p-1}]$. Otherwise, let $x = \min S \bigcap [2^{p-1}+1, 2^p]$. Then, x is the root of the tree representing S and its left and right subtrees are, respectively, the trees representing subsets $S \bigcap [1, 2^{p-1}]$ and $S \bigcap [x+1, 2^p]$ in the universes $[1, 2^{p-1}]$ and $[2^{p-1}+1, 2^p]$. The height of the resulting tree is at most $\log|U|$. To insert a new element x, we compare x with the root of the tree, say r. Suppose that r belongs to interval $[2^{i-1}+1, 2^i]$. We distinguish the following cases:

Case 1. $x > 2^i$: Make x the tree root with the old tree as the left subtree.

Case 2. $r < x \leq 2^i$: Insert x into the right subtree recursively.

Case 3. $2^{i-1}+1 \leq x < r$: Place x at the tree root, substituting item r, and insert r into the right subtree recursively.

Case 4. $x \leq 2^{i-1}$: Insert x into the left subtree recursively.

Deletion is analogous to insertion. It is easy to see that each operation requires only $O(\log|U|)$ time using any update mechanism.

5 Maintaining a dynamic sparse array

First, we describe a way of maintaining a dynamic sparse array with $O(1/\epsilon)$ time per operation and $O(N^\epsilon)$ space per update, for any $0 < \epsilon < 1$. We represent the array by a trie [8] with N^ϵ branches per node. See Figure 5. Each node of the trie maintains a subset of the array entries by hashing the subset uniformly into N^ϵ buckets. The trie root maintains the entire array. If two or more nonzero entries hash into a bucket, the bucket contains a pointer to a child node that recursively maintains the entries of the bucket. Otherwise, either the bucket has exactly one nonzero entry or is empty. As we have described, the trie may contain nodes with exactly one nonempty bucket (*degree-1 nodes*). We save space by shrinking paths of degree-1 nodes and labeling each shrunk path with the string of hash values it spells out. Since the height of the trie is $O(1/\epsilon)$, the cost of looking up an array entry is $O(1/\epsilon)$. To update an entry, we first determine if the entry is 0. If so, consider the lowest bucket of the trie into which the entry hashes. If this bucket is empty, we simply add the entry to the bucket. If the bucket already contains a nonzero entry, we create a new node storing both entries and point to the node from the bucket. Finally, if the bucket points to another node, we split the edge leaving this bucket by creating a new node that holds the new entry. Creating a new node involves initializing N^ϵ new buckets, but this can be accomplished in $O(1)$ time using a standard array initialization trick [1] (see Ex. 2.12, page 71). The other case where we update a nonzero entry is handled similarly. Clearly, an update operation requires $O(1/\epsilon)$ time and $O(N^\epsilon)$ space.

Next, we improve the space per update to $O(N^\epsilon/2^{1/\epsilon})$ (amortized) by making the time per update amortized (instead of worst-case) and sacrificing a constant factor in the query time. Setting $\epsilon = 1/\sqrt{\log N}$, we obtain a strategy with $O(\sqrt{\log N})$ query time and $O(\sqrt{\log N})$ amortized time and

$O(1)$ amortized space per update. To this end, we impose the constraint that every node of the trie contains $> 2^{1/\epsilon}/\epsilon$ nonzero entries. If there are $\leq 2^{1/\epsilon}/\epsilon$ nonzero entries totally, we store these entries in a balanced tree instead of a trie. We no longer require that nodes have at least two nonempty buckets. A bucket (of a node) that contains 1 to $2^{1/\epsilon}/\epsilon$ nonzero entries stores these entries in a balanced tree, ordered according to their indices in the array. A bucket containing $> 2^{1/\epsilon}/\epsilon$ nonzero entries points to a child node that recursively maintains these entries. Looking up an array entry involves finding the lowest bucket of the trie into which the entry hashes and searching for the entry in the balanced tree stored at the bucket. This process takes $O(1/\epsilon)$ time. Updating an entry is similar, except that, if the number of nonzero entries in a bucket exceeds $2^{1/\epsilon}/\epsilon$, then we create a new child node below this bucket and hash the bucket entries into the buckets of new node. Creation of a new node may propagate at most $1/\epsilon$ times.

Now, we show that this data structure has the claimed amortized time and space bounds per update operation. Define the height of an entry in the trie to be $1/\epsilon - d$, where d denotes the depth of the lowest node in the trie containing the entry. Each nonzero entry maintains a *temporal potential* equal to h, its height in the trie, and a *spatial potential* equal to $\epsilon h N^\epsilon/2^{1/\epsilon}$. The only expensive steps in an update operation are steps that create new nodes. The creation of a new node requires N^ϵ space and time propotional to the number of nonzero entries hashing into the node. Since the creation of a new node decreases the heights of nonzeroentries hashing into that node, the decrease in the spatial and temporal potentials of these entries, respectively, pay for the space and time used to create the node. We charge the creation of spatial and temporal potentials for a new nonzero entry to the update operation that creates the entry. Thus an update operation requires $O(1/\epsilon)$ amortized time and $O(N^\epsilon/2^{1/\epsilon})$ amortized space.

This scheme works so long as nonzero entries are not made 0 through updates. When we make a nonzero entry 0, we can just treat the entry like a nonzero entry and leave it in the trie. If we wish, in order to save space, we can periodically compact the trie by eliminating all zero entries. If we perform a compaction whenever the number of zero entries becomes a constant fraction of the total number of entries stored in the trie, then it is easy to show that the amortized time per update increases by only a constant factor.

6 Set Equality-testing

We describe a data structure for set equality-testing based on our binary search tree representation for dense dictionaries in Section 4. It is also possible to base the description of the data structure on binary trie representation of sets. We number the elements seen so far in serial order and call the collection of these elements, the ***universe.*** Every set in the universe is represented by a binary search tree according to this numbering as described in Section 4. The arrival of new elements into the universe during set operations does not cause problems since new elements are assigned higher serial numbers than existing elements. Two sets are equal if and only if their roots are identical. We update a set recursively, as described in Section 4, using CONS operations. The number of CONS operations performed per update is at most $\log m$, where m is the number of update operations.

We show how to implement a CONS operation in $O(\sqrt{\log m})$ amortized time and $O(1)$ amortized space, and obtain an implementation of set updates in $O((\log m)^{3/2})$ amortized time and $O(\log m)$ amortized space. A CONS operation determines whether a binary search tree is present in a collection of trees, given its two subtrees and root element. Here, the collection is all the trees representing sets together with their subtrees. If the tree is in the collection, it is simply returned; otherwise it is added to the collection and then returned. We serially assign numbers to all trees in the collection according to their order of creation. Associating with each tree a triple (l, r, x) of numbers of its two subtrees and root element, we see that a CONS operation is equivalent to asking for a particular triple in a collection of triples. If m denotes the total number of update operations, then the coordinates of a triple corresponding to the subtrees are at most $m \log m$ and the coordinate corresponding to the root element is at most m. We maintain the collection of triples (and their corresponding trees) in an array of size $m^3(\log m)^2$. Using the techniques of previous section to implement this array, we see that a CONS operation can be implemented in $O(\sqrt{\log m})$ amortized time and $O(1)$ amortized space. This description assumes an *a priori* knowledge of m. To eliminate this requirement, we guess m initially, and double the value of the guess each time it turns out to be incorrect. Whenever we change our guess, we create a bigger sparse array and store the collection of triples in it. It is easy to check that this increases the amortized time of an update by only a constant factor.

7 Sequence Equality-testing

In this section we present two data structures for testing equality of sequences.

7.1 Data structure 1

We represent a sequence by an almost complete binary tree as described in Section 2. The ith node of the tree in symmetric order stores the ith element of the sequence. As before, for each $j \leq \lfloor \log n/2 \rfloor$, we maintain a list of j-trees of the sequence. One problem with this representation is that a node can belong to several sequences and hence to several lists of j-trees. Since a node can have several left and right pointers corresponding to the lists of j-trees that contain it, how do we navigate the list of j-trees of a particular sequence efficiently?

We solve this problem using a result of Driscoll et al. [6] for making pointer-based data structures persistent. A data structure is *fully persistent* if it maintains all versions of the data structure created by the update operations so far and permits accesses and updates (that create new versions) to any existing version. Driscoll et al.'s result [6] is that any pointer-based data structure in which nodes have constant bounded indegree can be made fully persistent at the expense of weakening the time and space per update operation to amortized bounds (if the original bounds are amortized, they remain amortized). In our data structure nodes have indegree at most 4, since a node has at most two parents and at most two adjacent nodes in its list of j-trees that point to it. Therefore, our data structure has a fully persistent counterpart that maintains all sequences created so far and permits accessing and updating any of them.

One remaining problem is the incorporation of CONS operations into the fully persistent version of our data structure. We implement CONS operations as in the previous section, by serially numbering all the elements and the trees and maintaining triples corresponding to trees in a sparse dynamic array. This gives rise to two fresh problems:

1. We are using random access to access nodes of the data structure via CONS operations, whereas only purely pointer-based data structures can be made fully persistent.

2. The method used by Driscoll et al. to achieve full persistence is to split a node when it has to maintain too many pointers corresponding to different versions of the data structure. Therefore, a node of the data structure that is the root of a single binary tree can have many different copies representing the same tree. How do we know that two nodes represent the same tree? The answer to this question is required to determine if two sequences are equal, given the roots of their trees, and to implement CONS operations.

The second problem is solved easily by maintaining in each node the serial number of the tree it represents. Therefore, to test the equality of two sequences, we simply compare the serial numbers of the roots of their binary trees. When a node is split, the two resulting copies get its serial number.

We solve Problem 1 by modifying the implementation of a Cons operation as follows. Previously, a Cons operation created a new node corresponding to the root of a tree only if the tree was not already in the collection. Now, a Cons operation always creates a node representing the root of a tree irrespective of whether the tree is in the collection or not. If the tree already exists, its serial number is stored in the newly created node besides pointers to nodes that represent its two subtrees. Otherwise the tree is given the next available serial number, its triple is added to the collection of triples, and its number is stored in the newly created node. This solves problem 1, since we no longer access nodes of the data structure using random access. Random access is used only to obtain the serial number of a triple. Since a serial number is an information field in a node and we are free to perform any computation on the information fields of nodes of the data structure, the data structure can be made fully persistent without any difficulty.

The fully persistent version of the data structure solves the sequence equality-testing problem. Now, we analyze its performance. The cost of testing equality of two sequences is $O(1)$. The time to update a sequence of length n is $O(\sqrt{n \log m})$ (amortized), where m denotes the number of update operations, and the space required is $O(\sqrt{n})$ (amortized). If we do not know m in advance, as before we guess m and keep updating our guess, thereby creating bigger and bigger sparse arrays. This makes our bound on the amortized time per update slightly worse. It is easy to show that the new bound is $O(\sqrt{\bar{n} \log m})$, where $\bar{n}$ denotes the maximum length of a sequence.

7.2 Data structure 2

We combine the idea of maintaining lists of j-trees for a sequence with another representation of sequences. We associate a parameter $K = 2^p - 1$ with each sequence, to be specified later. For each sequence, we maintain the lists of of its j-trees, for $j \leq p$. We store the lists of j-trees of all sequences in a fully persistent manner similar to Data structure 1. We also maintain a sequence of serial numbers of its *representative* p-trees in left-to-right order. The *representative* p-trees of a sequence consist of every Kth p-tree and the last p-tree of the sequence. The sequence of serial numbers is called the *signature* of a sequence. We store the signatures of all sequences in a *lexicographic splay tree* [13]. The amortized cost of accessing a signature or inserting a new signature into a lexicographic splay tree (*signature tree,* henceforth) is $O(l + \log N)$, where l denotes the length of the signature and N denotes the number of signatures in the tree. This completes the representation.

Two sequences are identical if and only if their signatures are identical and this can be checked in $O(1)$ time if each sequence keeps track of the location of its signature in the signature tree. To update a sequence, we update the lists of j-trees of the sequence bottom up using Cons operations. This is accomplished as in Data structure 1. Then, we compute and insert the signature of the new sequence into the signature tree. The amortized time required by an update is $O(K\sqrt{\log m}+n/K+\log m)$, where n is the length of the updated sequence and m is the total number of update operations. This follows from the facts that updating the lists of j-trees requires $O(K)$ Cons operations and that creation and insertion of the signature of a new sequence into the signature tree requires $O(n/K+\log m)$ time. The amortized space needed by an update is $O(K + n/K)$. Setting $K = 2^p - 1 \geq \sqrt{n}/(\log m)^{1/4} \geq 2^{p-1}$, it follows that an update requires $O(\sqrt{n}(\log m)^{1/4} + \log m)$ amortized time and $O(\sqrt{n}(\log m)^{1/4})$ amortized space.

We have tacitly ignored the problem that, when the parameter K of a sequence suddenly increases due to an insertion, we have to create the entire list of p-trees for the new sequence. In order to avoid this expensive operation, for each sequence, we also maintain a partial list of its $(p+1)$-trees and add a new $(p+1)$-tree to this list during each insertion, unless the list is complete. Note that this list has to be updated during an update operation. We claim that whenever K increases, the complete list of $(p+1)$-trees of the updated sequence is available. This is because, for the quantity $\sqrt{n/\log m}$ to double, n must quadruple. In other words, we must perform a series of $3n$ insertions between two successive increases of parameter K, and after the first n insertions the construction of the list of $(p+1)$-trees is complete. Therefore, when K increases during an insertion, the list of $(p_{old}+1)$-trees of the old sequence becomes the list of p_{new}-trees of the new sequence. It is easy to check that this only increases the time and space per update by a constant factor.

Once again, we have assumed that we know m beforehand. If this is not the case, we keep guessing m, and each time we change our guess, we increase the size of the array of triples and reexecute the whole sequence of update operations so far for the new value of m. This is necessary since the parameter K of a sequence depends on m. This weakens our bound on the amortized time per update to $O(\sqrt{\bar{n}}(\log m)^{1/4} + \log m)$, where $\bar{n}$ is the maximum length of a sequence.

8 Conclusion

Several interesting open problems are raised by our work:

1. **Unique binary search tree representations:** We have seen some ways of uniquely representing a dictionary by a binary search tree that are optimal when the dictionary is sparse ($|U| \geq T_{n+1}(cn)$) or dense ($|U| \leq n^c$). Determine the complexity of unique representations when the dictionary is of intermediate density ($n^c < |U| < T_{n+1}(\bar{c}n)$).

2. **Dynamic sparse array maintenance:** Dynamic sparse arrays are useful in efficiently implementing CONS operations in high level languages like LISP and SETL. It would be of interest to obtain sharp time-space tradeoffs on the complexity of maintaining a dynamic sparse array.

3. **Set and Sequence equality-testing:** Obtain sharp time-space tradeoffs for set and sequence equality-testing. Another attractive avenue of research is to determine the randomized complexities of these problems.

4. **More powerful data types and operations:** Programming language LISP allows generalized lists which are a generalization of sequences and SETL allows sets and sequences to be themselves composed of other sets and sequences. Join and split are natural operations for sequences and the natural operations for sets are union, intersection, and set difference. It is a challenging problem to devise an efficient implementation of these data types and operations. Since this problem may not have an efficient deterministic solution, randomized solutions might be worth exploring.

Acknowledgement

The first author is much indebted to Mike Fredman for valuable discussions and encouragement. We thank Richard Cole for comments on an earlier draft.

References

1. A.V.Aho, J.E.Hopcroft, and J.D.Ullman, Design and Analysis of Computer Algorithms, Addison Wesley Publishing Co., 1974.

2. J.Allen, Anatomy of LISP, McGraw Hill Publishing Co., 1978.

3. C.R.Aragon and R.G.Seidel, "Randomized search trees", *Proc. 30th IEEE FOCS* (1989), 540-545.

4. K.Culik II and D.Wood, "A note on some tree similarity measures", *Info. Proc. Lett.* 15 (1982), 39-42.

5. M.Dietzfelbinger, A.Karlin, K.Mehlhorn, F.Meyer auf der Heide, H.Rohnert, and R.E.Tarjan, "Dynamic perfect hashing: Upper and lower bounds", *Proc. 29th IEEE FOCS* (1988), 524-531.

6. J.R.Driscoll, N.Sarnak, D.D.Sleator, and R.E.Tarjan, "Making data structures persistent", *J. Comp. Sys. Sci.* 38 (1989), 86-124.

7. R.L.Graham, B.L.Rothschild and J.Spencer, Ramsey Theory, John Wiley & sons, 1980.

8. D.E.Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley Publishing Co., 1973.

9. L.Lovász, Combinatorial Problems and Exercises, North-Holland Publishing Co., 1979.

10. J.I.Munro and H.Suwanda, "Implicit data structures for fast search and update," *J. Comp. Sys. Sci.* 21 (1980), 236-250.

11. W.Pugh, "Incremental computation and the incremental evaluation of functional programming," Ph.D. Thesis, Cornell University, 1988.

12. W.Pugh and T.Tietelbaum, "Incremental computation via function caching," *Proc. 16th ACM POPL* (1989), 315-328.

13. D.D.Sleator and R.E.Tarjan, "Self-adjusting binary search trees", *J. ACM* 32 (1985), 652-686.

14. D.D.Sleator, R.E.Tarjan, and W.P.Thurston, "Rotation distance, triangulations, and hyperbolic geometry," *J. AMS* (1988), 647-682.

15. L.Snyder, "On uniquely representable data structures," *Proc. 18th IEEE FOCS* (1977), 142-146.

16. R.E.Tarjan and A.C.Yao, "Storing a sparse table", *Comm. ACM* 22 (1979), 606-611.

17. M.N.Wegman and J.L.Carter, "New hash functions and their use in authentication and set equality", *J. Comp. Sys. Sci.* 22 (1981), 265-279.

18. R.Wilber, "Lower bounds for accessing binary search trees with rotations," *SIAM J. on Computing,* 18 (1989), 56-67.

19. D.Yellin, "Representing sets with constant time equality testing", IBM Tech. Rept., April 1989.