DIMACS Technical Report 93-16
March 1993

Confluently Persistent Deques via Data-Structural Bootstrapping[1]

by

Adam L. Buchsbaum[2,3]
Dept. of Computer Science
Princeton University
Princeton, NJ 08544
alb@cs.princeton.edu

Robert E. Tarjan[4,5]
Dept. of Computer Science
Princeton University
Princeton, NJ 08544
and
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
ret@cs.princeton.edu

[1]An extended abstract of this paper was presented at the 4th ACM-SIAM Symposium on Discrete Algorithms, 1993.
[2]Affiliated Graduate Student Member.
[3]Supported in part by a Fannie and John Hertz Foundation fellowship and National Science Foundation Grant No. CCR-8920505.
[4]Permanent Member.
[5]Research at Princeton University partially supported by the National Science Foundation, Grant No. CCR-8920505, and the Office of Naval Research, Contract No. N00014-91-J-1463.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

We introduce *data-structural bootstrapping*, a technique to design data structures recursively, and use it to design confluently persistent deques. Our data structure requires $O(\log^* k)$ worst-case time and space per deletion, where k is the total number of deque operations, and constant worst-case time and space for other operations. Further, the data structure allows a purely functional implementation, with no side effects. This improves a previous result of Driscoll, Sleator, and Tarjan.

1 Introduction

Consider the following operations to be performed non-destructively on linear lists of items:

makedeque(d) Create and return a new list of one element, d.

pop(X) Return an ordered pair of the first element of X and the list containing the second through the last elements of X.

eject(X) Return an ordered pair of the last element of X and the list containing the first through the second-to-last elements of X.

catenate(X, Y) Return the list containing the elements of X followed by those of Y. X may be the same list as Y.

Knuth [23] calls such lists *deques* (for **d**ouble-**e**nded **que**ues) since access is provided to both ends of the lists. The familiar *push* and *inject* operations, which insert elements into the front and rear of lists, are subsumed by catenate. We can also allow separate *first*(X) and *last*(X) operations, which simply return the first and last elements of list X, respectively, without creating a new list.

While the above operations can all be implemented destructively in $O(1)$ worst-case time using doubly-linked lists [37], providing a *non-destructive* implementation is more problematic. By *non-destructive* we mean that the operations pop, eject, and catenate leave their argument lists wholly unchanged. This falls within the realm of *persistent data structures*, which we discuss in Section 1.1. Driscoll, Sleator, and Tarjan [19] give a solution to the non-destructive *output-restricted deque* problem (no eject operation) that runs in $O(1)$ amortized time [38] and space for all the operations except catenate; catenate consumes $O(\log\log k)$ amortized time and space per operation, where k is the number of operations preceding the catenate.

We report a solution to the above problem in which pop and eject require $O(\log^* k)$ amortized time and space, where k is the number of deque operations performed so far, and all the other operations require $O(1)$ amortized time and space each. We also show how to make these resource bounds worst-case; our solution can be implemented purely functionally, with no side effects. Noting that the Driscoll, Sleator, and Tarjan [19] solution to the output-restricted deque problem provides non-constant resource bounds ($O(\log\log k)$ amortized time and space) for catenate rather than pop and eject, we can modify our data structure to provide a solution to the output-restricted deque problem with $O(\log^{(i)} k)$ amortized time and space bounds for catenate,[1] where i is any desired constant, and $O(1)$ amortized time and space bounds for the other operations.

The main technique we employ is *data-structural bootstrapping.* We use two types of bootstrapping. First, we abstract fully persistent lists (see Section 1.1) by representing them

[1] $\log^{(i)}$ is the log function iterated i times.

by their persistent version numbers; we thus reduce catenation to simple insertion. Second, we implement deques of n elements by decomposing them into collections of deques of $O(\log n)$ elements and applying this decomposition recursively. The first type of bootstrapping, which we call *structural abstraction*, is used by Driscoll, Sleator, and Tarjan [19] to implement persistent catenable lists using fully persistent (non-catenable) lists and by Buchsbaum, Sundar, and Tarjan [6] to implement catenable heap-ordered deques using non-catenable heap-ordered deques. It is similar to a technique used by Kosaraju [24] to design catenable deques by dissecting them and storing contiguous pieces on stacks. The second type of bootstrapping, which we call *structural decomposition*, is similar to an idea of Dietz [12], who recursively decomposes indexed 2-3 trees so that their leaves store smaller indexed 2-3 trees, improving their performance from $O(\log n)$ to $O(\log^* n)$ amortized time. Structural decomposition differs from ideas used to solve decomposable search problems [5] and from other ad hoc instances of data structure decomposition (e.g., [14, 15, 16, 40, 42, 43]) in that these previous results use only one level of decomposition, whereas bootstrapping is recursive.

Non-destructive (or, in the parlance of Section 1.1, *confluently persistent*) deques have many uses in high-level programming languages such as LISP, ML, and Scheme [35], in which side-effect-free list operations are fundamental. They also can be used in the implementation of continuation passing in functional programming languages [4]. Furthermore, some purely functional programming languages do not allow any reassignment of memory cells; such languages effectively implement all data structures, regardless of use, non-destructively. Therefore, any application of catenable deques becomes an application of confluently persistent deques when implemented by such languages.

Our results impact the study of persistent data structures, a brief overview of which we now provide.

1.1 Persistent Data Structures

Data structures normally provide two types of operations: *queries*, which return information without modifying the data structure, and *updates*, which may change the underlying information being represented. In the terminology of Driscoll et al. [18], a data structure is *ephemeral* if an update destroys the version of the data structure being changed; i.e., there is always one and only one valid version. A data structure that allows queries but not updates to previous versions is *partially persistent*; in this scheme, a time line of versions evolves. Allowing previous versions to be updated yields a *fully persistent* data structure and produces a tree-like notion of time.

There has been much previous work on making various data structures partially or fully persistent [8, 10, 11, 17, 22, 26, 27, 28, 29, 32, 34]. There have also been some results providing general methods for making entire classes of data structures persistent [13, 15, 18, 30, 33]. In particular, Driscoll et al. [18] provide a method by which many pointer-based data structures may be made partially or fully persistent with only a constant factor overhead in time consumption (over the ephemeral version) and using only a constant amount of space per persistent update; both bounds are amortized. Their result is subject to the important restriction that any node in the ephemeral data structure may have only a fixed number of incoming and outgoing pointers. Dietz [13] makes arrays fully persistent so that access and store operations on an array of size n require $O(\log \log n)$ time (amortized expected for store) and linear total space.

None of these previous results, however, allows updates involving more than one version; combining two versions, such as is done in deque catenation, is not possible. This type of update leads to a DAG-like concept of time, and a data structure in this setting is called *confluently persistent* by Driscoll, Sleator, and Tarjan [19]. Driscoll et al. [18] leave open solving this problem in general. While we do not offer such a general solution, to date we are aware of only Driscoll, Sleator, and Tarjan [19] as a previous result in this direction. Our work provides a solution to a more general problem (deques rather than output-restricted deques in a confluently persistent framework) and also gives asymptotically improved resource bounds ($O(\log^* k)$ versus $O(\log \log k)$ amortized time and space). Furthermore, we demonstrate the power of data-structural bootstrapping as a tool for designing high-level data structures.

We begin by presenting a data structure in which all operations require $2^{O(\log^* k)}$ amortized time and space. While $2^{O(\log^* k)}$ seems to be an unusual function, it results from a natural recurrence: $F(n) = c'F(\log n) + O(1)$. This recurrence arises from the structural decomposition of a deque of n elements into a collection of deques of $O(\log n)$ elements each. Section 2 describes the *deque tree*, a tree whose leaves represent a deque, and discusses a way to keep it balanced. Section 3 demonstrates how to implement deque trees using only lists and catenable deques. While Driscoll, Sleator, and Tarjan [19] suggest a similar decomposition of trees into paths that in turn are represented by trees, we make the observation that a decomposition of trees into paths accessible only at their ends is possible and preferable. Section 4 then introduces the two kinds of data-structural bootstrapping. The first kind, *structural abstraction*, abstracts fully persistent lists by representing them by their persistent version numbers; these version numbers effectively point to the lists and are stored in confluently persistent deques that are similarly abstracted. The second kind of bootstrapping, *structural decomposition*, uses the properties of the deque trees to show that $O(\log n)$-size confluently persistent deques suffice to implement n-size confluently persistent deque trees via the above method. We then improve our data structure in Section 5 to reduce the resource bounds for pop and eject to $O(\log^* k)$ each and those for the other operations to $O(1)$ each; we also make these bounds worst-case. Section 6 provides lower bounds on the methods given to balance deque trees, suggesting that simple extensions of the methods presented in this paper might not suffice to improve the bounds further. We conclude in Section 7 with some discussion of our results and suggestions for future work. A preliminary version of this paper appears as Buchsbaum and Tarjan [7].

2 Deque Trees

Consider an ordered tree [23] embedded in the plane with the root at the top and the leaves at the bottom. We can exploit the induced left-to-right order of the leaves of the tree to make

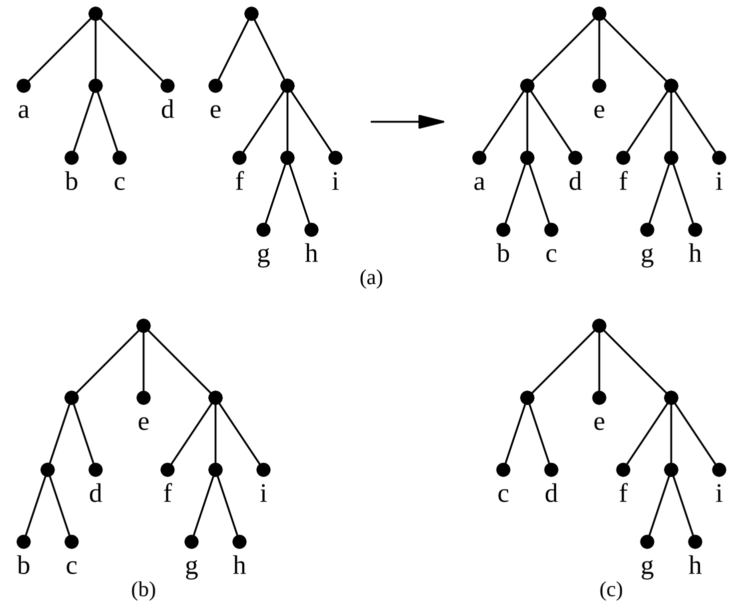

Figure 1: (a) Linking two trees; (b) After deleting the leftmost leaf of the linked tree of (a); (c) After deleting the leftmost leaf of the tree of (b).

the tree represent a list. The two operations we perform on the tree are *link*, which takes two trees and makes the root of one the new leftmost or rightmost child of the root of the other (see Figure 1(a)), and *delete*, which removes either the leftmost or the rightmost leaf of the tree (see Figure 1(b)). Link corresponds to deque catenation, and delete corresponds to pop and eject. The internal nodes of the tree can be of arbitrarily high degree, but we always ensure that they are at least binary; to do this requires replacing an internal node x by its remaining child if a deletion renders x unary. See Figure 1(c). It also requires a special case of link when linking two single-node trees; in this case, a new root node is created with the two old nodes becoming its children. Since delete always preserves the left-to-right order of the remaining leaves, it is clear that such trees can be used to implement catenable deques, and we therefore call them *deque trees*.

The remainder of this section describes a method to keep deque trees "in balance." The next two sections describe a method to implement confluently persistent deque trees (and thus deques). The efficiency of the implementation derives from the deque tree balance property.

We define a *pull* operation on deque trees just as Driscoll, Sleator, and Tarjan [19] define this operation on similar trees; Buchsbaum and Tarjan [6] define a slightly different form of pull. Let x be the leftmost non-leaf child of the root r of a tree T, and let x' be the leftmost child of x. A *pull* on T cuts the link from x' to x and makes x' a new child of r just to the left of x. Additionally, if x is now unary, it is replaced by its remaining child. See Figure 2. We can also define this as a *left pull* and define a *right pull* symmetrically, but for our needs left pulls suffice. Note that a pull preserves the left-to-right order of the leaves of T.

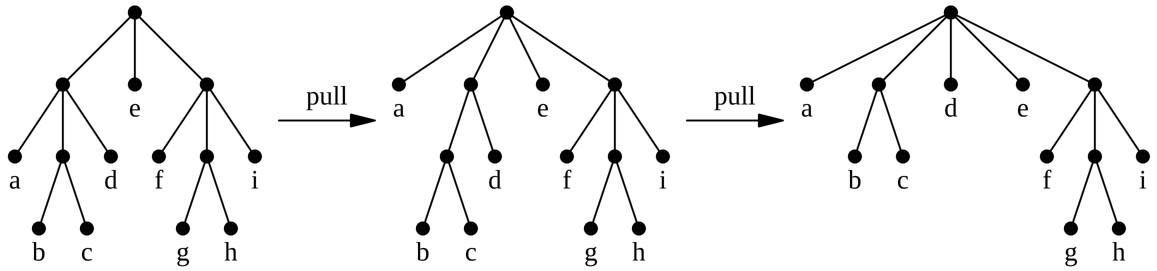

Figure 2: Two pulls applied to a tree.

We call a tree *flat* if all the children of the root are leaves or if the tree is a singleton node. A pull on a flat tree does nothing. A pull on a non-flat tree increases the degree of the root by one. In what follows, the *size* of a tree T, denoted by $|T|$, is the number of leaves in T.

We now describe how to use pulls to keep deque trees balanced, i.e., of logarithmic depth. To be precise, we define the *depth* of a deque tree to be the length of its longest root-to-leaf path. We wish to maintain the following invariant: denoting the depth of a deque tree T by d_T, we keep $|T| \geq c^{d_T}$ for some constant $c > 1$. To maintain this invariant, we augment the implementation of deque trees to perform k pulls on T every time we perform a deletion on T, for some constant k; without loss of generality, assume that the pulls precede the deletion. Also, we implement *linking by size*; i.e., when linking two trees A and B, we link the one with fewer leaves to the one with more, breaking a tie arbitrarily. This technique is also used to maintain balanced trees in the disjoint set union problem [39]. We first prove the following.

Lemma 2.1 *If $k \geq 1$ then a non-root node in a deque tree never becomes a root node of a deque tree T unless T is a singleton node.*

Proof: New root nodes are created by the makedeque operation and by linking two singleton trees together. Consider a deletion on a tree T; at least one pull precedes this deletion. Before the pull, either T is flat or else the root of T is of degree at least two. In the former case, if the deletion creates a new root node of T, then T becomes a singleton node; in the latter case, the pull increases the degree of the root of T by one, to at least three, so the deletion does not change the root of T. $\square$

Now we can prove that pulls maintain balanced deque trees.

Theorem 2.2 *If $1 < c < 2$ and $k \geq \frac{2}{2-c}$, then $|T| \geq c^{d_T}$.*

Proof: We proceed by induction on the number of deque tree operations. A singleton tree of depth zero is created by makedeque. Consider any depth-increasing link of a tree T to a tree T' yielding tree T''. We have $d_{T''} = d_T + 1 > d_{T'}$. By induction, $|T| \geq c^{d_T}$. Linking by size gives $|T'| \geq |T|$, and thus $|T''| \geq 2c^{d_T}$. As long as $c \leq 2$, the invariant holds.

Assume some deletion creates a tree X with root r such that $|X| < c^{d_X}$. We show that this produces a contradiction. Let R be the most recent tree rooted at r formed before X by a depth-increasing link. By the above analysis of linking, we know that $|R| = 2c^{d_R - 1} + x$ for some $x \geq 0$. The intermixed sequence of deletions and links that produces X from R cannot make r a non-root node; otherwise Lemma 2.1 shows that X is the singleton node r. Let T be the evolving tree, initially R, rooted at r as leaves are deleted from it and other trees linked to it. For $|X| < c^{d_X}$ to be true, at least $(2-c)c^{d_R-1} + x + y$ leaves must be deleted from T, where y is the total number of leaves added to T via links. If $k \geq \frac{2}{2-c}$, then there is a sufficient number of pulls to flatten T before T becomes X. This is because (1) each pull increases the degree of the root of T by one unless T is flat, and (2) there are always at least as many pulls remaining as leaves of T. Once T is flat, the depth invariant holds until the next depth-increasing link. So no depth violation occurs. $\square$

In particular, we can set $c = 3/2$ and $k = 4$.

Corollary 2.3 *If T is a deque tree, then $d_T = O(\log |T|)$.*

We prove in Section 6 that performing k pulls per deletion, for any constant k, can in fact result in $d_T = \Omega(\log |T|)$.

We close this section by contrasting our use of the pull operation with previous uses. Whereas Driscoll, Sleator, and Tarjan [19] employ the pull operation primarily to ensure that the left spine of any tree remains short (of constant length), we utilize it to keep the tree as a whole balanced in the sense that no leaf is of greater than logarithmic depth. Both results, however, use only the fact that pulls completely flatten the tree over some predictable period of time. For this reason, the full effect of the pull operation on the structure of the trees may be underutilized in these analyses. Buchsbaum and Tarjan [6] use the pull operation to effect path compression on trees, with no local or global balance considered.

3 Decomposition into Spines

In this section we describe how to decompose a deque tree into a collection of *spines*. We define the spines bottom-up. A *left spine* of a tree is a maximal tree path $(x_0, \ldots, x_l)$ such that x_0 is a leaf and x_i is the leftmost child of x_{i+1} for $0 \leq i < l$; we say the spine *terminates* or *ends at* x_l. A *right spine* is defined symmetrically. We show how to represent a deque tree as a set of spines so that all the deque tree operations—link, delete, and pull—require accessing the spines only at their ends. By recursively representing the spines by smaller deque trees, we obtain a bootstrapped implementation of confluently persistent deques.

Recall the deque operations from Section 1. We first describe how to implement them ephemerally using deque trees. Later we will show how to make the implementation persistent. Let *root*(T) be the root of a deque tree T. We store the left and right spines terminating at *root*(T) in the deques *ls*(T) and *rs*(T); these are called the *root spines*. For a node x with i children in T, let the children of x be $c_x^1, \ldots, c_x^i$ in left-to-right order. Let l_x^j (r_x^j) be the left (right) spine terminating at c_x^j for $1 \leq j \leq i$. We store at node x a pointer to

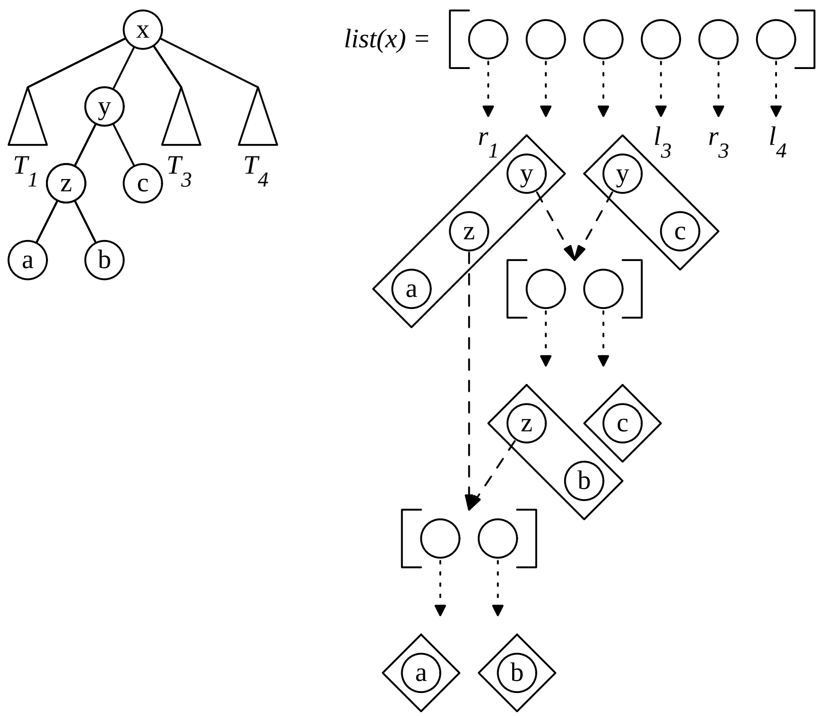

Figure 3: A subtree rooted at node x and a portion of its spine decomposition structure. Boxes surround spine deques, and s-lists are bracketed; l_i (r_i) refers to the spine deque for the left (right) spine terminating at the root of subtree T_i. Null s-lists (for leaves) are omitted.

a doubly-linked list *s-list*(x) whose elements left-to-right are $r_x^1, l_x^2, r_x^2, \ldots, l_x^{i-1}, r_x^{i-1}, l_x^i$; i.e., we record the right spine terminating at the leftmost child, the left and right spines terminating at the middle children, and the left spine terminating at the rightmost child of x. The "missing" spines are recorded in the s-lists of some other nodes (if they are not the root spines). All the spines are stored as deques with the leaves at the front (pop) ends. The spine deques, therefore, contain as elements nodes pointing to their respective s-lists. See Figure 3. A node in a deque tree is thus stored in precisely two spines, appearing as the top node in at least one of them.

The operations on the s-lists are (1) $insert(l, x, y)$ — insert y after x in s-list l; and (2) $delete(l, x)$ — delete x from s-list l and return x. Additionally, each element x in an s-list has a $pred(x)$ and $succ(x)$ pointing to the predecessor and successor of x respectively. Special sentinels $head(l)$ and $tail(l)$ point to the first and last elements of s-list l, and the special case insert$(l, pred(head(l)), x)$ inserts a new first element x into l. The sentinels are updated by the s-list implementation machinery. Again, assume for now that these are ephemeral operations.

To make a new deque tree T with one node x, we set $ls(T)$ and $rs(T)$ to be the singleton deque (x) and *s-list*(x) to be $\emptyset$. Now we describe how to link a deque tree A to another deque tree B, assuming at least one of them contains more than one leaf. Assume that $root(A)$ becomes the new leftmost child of $root(B)$; the other case is symmetric. First, eject

$(x, ls(T)) \leftarrow$ pop$(ls(T))$ // the leftmost leaf of T
$(y, ls(T)) \leftarrow$ pop$(ls(T))$ // the parent of x
if (y has more than 2 children) then
  delete($s\text{-}list(y), head(s\text{-}list(y))$) // discard x as the old right spine
  $l \leftarrow$ delete($s\text{-}list(y), head(s\text{-}list(y))$) // next left spine
  $ls(T) \leftarrow$ catenate(l, catenate(makedeque(y), $ls(T)$))
else if ($y \neq root(T)$) then // must delete y and promote other child
  $(g, ls(T)) \leftarrow$ pop$(ls(T))$ // the parent of y
  $l \leftarrow tail(s\text{-}list(y))$ // next left spine
  $r \leftarrow$ delete($s\text{-}list(g), head(s\text{-}list(g))$) // right spine terminating at y
  $(y, r) \leftarrow$ eject(r) // remove y
  insert($s\text{-}list(g), pred(head(s\text{-}list(g))), r$) // new right spine
  $ls(T) \leftarrow$ catenate(l, catenate(makedeque(g), $ls(T)$))
else // $y = root(T)$ and has two children
  $ls(T) \leftarrow tail(s\text{-}list(y))$ // new left spine of T
  $(y, rs(T)) \leftarrow$ eject$(rs(T))$ // delete y

Figure 4: Pseudocode for deletion of leftmost leaf of T. Double slash (//) precedes comments.

$root(B)$ from $ls(B)$ and insert the new $ls(B)$ as the new head of $s\text{-}list(root(B))$, as this is now the left spine terminating at the new second-leftmost child of B. Then insert $rs(A)$ as the new head of $s\text{-}list(root(B))$; this is now the right spine terminating at the new leftmost child of B. Finally, reset $ls(B)$ to be the catenation of $ls(A)$ and $root(B)$. The special case when A and B are both singleton deque trees is handled by creating a new structure for the resulting tree of a root and the two leaves.

A deletion is no more complicated, but it involves more steps, so we merely give pseudocode for deletion. See Figure 4; recall that Figure 1 demonstrates deletion pictorially. The code in Figure 4 is for a deletion of the leftmost leaf of T; the code for deletion of the rightmost leaf is symmetric. We note that it is crucial to the functioning of the spine decomposition that the nodes not store the first left spine or the last right spine; these are stored as parts of bigger spines in other places. This lack of duplication allows the easy implementation of the deque tree operations by simple deque and s-list operations. It is also critical that the internal nodes remain at least binary; this avoids the situation in which two spines intersect at nodes other than the top of one of them.

To implement pulls, we augment the s-lists to store as secondary information doubly-linked lists of pointers to the non-leaf children of each node. Each node also stores its degree. Using the notation above, if c_x^j is a non-leaf child, then a pointer points to l_x^j in $s\text{-}list(x)$; the exception is that if c_x^1 is a non-leaf child, then the pointer points to r_x^1. This seems unfortunate, but it turns out that we must handle the case of a pull affecting a root spine

$(r, ls(T)) \leftarrow$ eject($ls(T)$) // remove $root(T)$ temporarily
$(x, ls(T)) \leftarrow$ eject($ls(T)$) // leftmost child of $root(T)$
if (degree(x) > 2) then // x remains a child of $root(T)$
    $r_y \leftarrow$ delete($s\text{-}list(x), head(s\text{-}list(x))$) // y is leftmost child of x
    $l_z \leftarrow$ delete($s\text{-}list(x), head(s\text{-}list(x))$) // z is next
    insert($s\text{-}list(r), pred(head(s\text{-}list(r)))$, catenate($l_z$, makedeque($x$))) // new left spine to x
    insert($s\text{-}list(r), pred(head(s\text{-}list(r))), r_y$) // right spine to y
else // delete x and promote both its children
    $r_x \leftarrow$ delete($s\text{-}list(r), head(s\text{-}list(r))$) // right spine to x
    $(x, r_x) \leftarrow$ eject(r_x) // get rid of x
    insert($s\text{-}list(r), pred(head(s\text{-}list(r))), r_x$) // now right spine to z
    insert($s\text{-}list(r), pred(head(s\text{-}list(r))), tail(s\text{-}list(x))$) // left spine to z
    insert($s\text{-}list(r), pred(head(s\text{-}list(r))), head(s\text{-}list(x))$) // right spine to y
$ls(T) \leftarrow$ catenate($ls(T)$, makedeque(r)) // restore $root(T)$ to $ls(T)$

Figure 5: Pseudocode for pull on T when x is on $ls(T)$. Double slash (//) precedes comments.

slightly differently from one affecting a middle child of $root(T)$ anyway. It is straightforward to maintain this secondary information when modifying the s-lists, and for clarity we omit this part of the implementation from the description. (In fact, we obviate this secondary information in Section 5.) We merely mention that maintaining the secondary information is possible since actual s-list modifications only take place at either the ends of the s-lists or near elements in the s-lists that are themselves targets of these secondary pointers. Figure 5 shows pseudocode to implement the pull operation if the first non-leaf child x of $root(T)$ is on $ls(T)$ (a similar procedure handles the case for $rs(T)$), and Figure 6 shows pseudocode for the case of x a middle child of $root(T)$. Note that the two procedures bear definite similarities. They could easily be combined into one procedure with a proper prologue and epilogue, but keeping them distinct increases their clarity. Refer back to Figure 2 for a pictorial representation of a pull.

The above explanation shows that a constant number of ephemeral deque and s-list operations suffices to implement any ephemeral deque tree operation. In the following section, we describe how to use the spine decomposition of deque trees to implement confluently persistent deques.

4 Data-Structural Bootstrapping

Section 3 shows how to implement ephemeral deque trees using catenable deques and s-lists. We now describe how to implement confluently persistent deque trees using this method as a

Let l_x be the left spine pointed to as the first non-leaf child of $root(T)$
$r_x \leftarrow succ(l_x)$
$p \leftarrow pred(l_x)$
$r \leftarrow root(T)$
delete($s\text{-}list(r), l_x$) // left spine to x
$(x, l_x) \leftarrow$ eject(l_x) // delete x from it
if (degree(x) > 2) then // x remains a child of $root(T)$
    $r_y \leftarrow$ delete($s\text{-}list(x), head(s\text{-}list(x))$) // y is leftmost child of x
    $l_z \leftarrow$ delete($s\text{-}list(x), head(s\text{-}list(x))$) // z is next
    insert($s\text{-}list(r), p,$ catenate(l_z, makedeque(x))) // new left spine to x
    insert($s\text{-}list(r), p, r_y$) // right spine to y
    insert($s\text{-}list(r), p, l_x$) // now left spine to y
else // delete x and promote both its children
    delete($s\text{-}list(r), r_x$) // right spine to x
    $(x, r_x) \leftarrow$ eject(r_x) // now remove x from it
    $r_y \leftarrow head(s\text{-}list(x))$ // y is left child of x
    $l_z \leftarrow tail(s\text{-}list(x))$ // z is right child of x
    insert($s\text{-}list(r), p, r_y$) // right spine to y
    insert($s\text{-}list(r), p, l_x$) // now left spine to y
    insert($s\text{-}list(r), r_y, r_x$) // now right spine to z
    insert($s\text{-}list(r), r_y, l_z$) // left spine to z

Figure 6: Pseudocode for pull on T when x is a middle child of $root(T)$. Double slash (//) precedes comments.

basis. We use an idea of Driscoll, Sleator, and Tarjan [19] to bootstrap confluently persistent deques and fully persistent s-lists.

First, note that the s-list structure is a *bounded indegree* structure, in the terminology of Driscoll et al. [18]. Furthermore, each $s\text{-}list(x)$ has only four *access pointers*, i.e., entry points to the s-list where access can start. These four points are the two ends and the nodes corresponding to the leftmost and rightmost non-leaf children of x. Thus, we can use the methods of Driscoll et al. [18] to make s-lists fully persistent so that a non-destructive s-list operation takes $O(1)$ amortized time and space. For now, assume the existence of confluently persistent deques.

The bootstrapping idea of Driscoll, Sleator, and Tarjan [19] is to make each element in a persistent s-list point to its respective spine via a *version number* referring to the corresponding persistent deque. Similarly, each element of a persistent deque contains the version number of the persistent s-list of the corresponding node. We call this *structural abstraction.*

The confluently persistent deque tree T is then fully described by two numbers: the version numbers of $ls(T)$ and $rs(T)$. We mention that the cases in the above implementation in which a node is popped off a spine only to be recatenated to the spine are present in fact to record the new version number of the node being recatenated to the spine. One implementation detail remains: updating a spine deque creates a new version of that deque, which must be stored in any s-list pointing to the spine; this creates a new version of the s-list, which must be reflected in a "higher" spine pointing to the list, and so on. This is further complicated by the fact that a node occurs in two spines, and when an s-list changes, both copies of the node that points to it must be updated.

We fix the second problem by storing the version number of any *s-list*(x) only in the copy of node x that occurs in the middle of a spine; the copy occurring at the top of a spine is left blank. If both copies occur at the tops of spines, one is chosen arbitrarily to store the version number of *s-list*(x). This works for the following reason. Consider how spines and s-lists are accessed: any time an *s-list*(x) is accessed via the copy of node x occurring at the top of a spine, the spine is either a root spine or a spine actually pointed to by *s-list*(r), where r is the root of a tree. In either case, both copies of x are readily available via a constant number of deque and s-list operations and so the version number of *s-list*(x) may be retrieved. The important fact is that no reverse pointers are needed. These would invalidate the methods of Driscoll et al. [18] in making the s-lists fully persistent, since too many access pointers would need to be accommodated. Now note that any modifications to spines and s-lists occur close to a root spine. I.e., updating a spine might cause an update to an s-list, but the node pointing to this s-list will be on $ls(T)$ or $rs(T)$. Therefore, no cascading sequence of version updates occurs. As an aside, we note that Driscoll, Sleator, and Tarjan [19] use the pull operation to ensure that any modifications to their structure occur close to the roots of their trees, thereby keeping cascading sequences of updates short.

While it is clear that s-lists can be made fully persistent using previously known techniques [18], the method described above relies upon the existence of confluently persistent deques in their own implementation. Note that to represent a confluently persistent deque of size n, however, we need only be able to implement confluently persistent deques of size $O(\log n)$; this is due to Corollary 2.3. This is the second type of bootstrapping we employ: namely, decomposition of a data structure into smaller pieces represented in the same fashion (and some auxiliary data structures, in this case the s-lists). We call this *structural decomposition.*

Theorem 4.1 *Any operation on a confluently persistent deque of size n can be performed in $2^{O(\log^* n)}$ amortized time and space.*

PROOF: We represent small confluently persistent deques (those of size less than some suitable constant, e.g., four) by doubly-linked lists. Operations on such a small deque are performed by copying the entire deque and modifying the new copy. Any time a deque increases to a size above this threshold, we implement it via the deque tree spine-decomposition structure described in Section 3.

An operation on a confluently persistent deque tree of n leaves implemented via the spine-decomposition structure requires $O(1)$ fully persistent s-list operations, each taking $O(1)$

amortized time and space [18]. It also requires $O(1)$ operations on confluently persistent deques that by Corollary 2.3 are of size $O(\log n)$. Therefore, for some constants c_1 and c_2, the amortized time and space required by an operation on a confluently persistent deque of size n is $F(n)$, where $F(n)$ is described by the following recurrence:

$$F(n) = \begin{cases} O(1) & \text{if } n \leq c_1 \\ c_2 F(\log n) + O(1) & \text{if } n > c_1 \end{cases}$$

The solution to this recurrence is $F(n) = 2^{O(\log^* n)}$. □

We note that confluently persistent deques support "self-catenation," i.e., forming a new deque by catenating an old one to itself. Therefore, k catenations can generate a deque of size 2^k. This potential size explosion is addressed by Driscoll, Sleator, and Tarjan [19] with a "guessing trick" that involves guessing how many operations will be performed and keeping the lists truncated to an appropriate size, doubling or squaring the guess and rebuilding the lists each time the number of operations exceeds the previous guess. This technique is important in this previous work, since the resource bounds obtained there of $O(\log n)$ and $O(\log \log n)$ would otherwise become $O(k)$ and $O(\log k)$ respectively, which would be poor amortized results. Our method requires no such additional machinery, however, since our resource bound grows sufficiently slowly in terms of n: for $n = O(2^k)$ we have $2^{O(\log^* n)} = 2^{O(\log^* k)}$.

Finally, we conclude this section by mentioning how to improve the performance of catenate, first, and last. The latter two operations can easily be made to take $O(1)$ worst-case time by storing with each deque its actual first and last elements. This does not degrade the performance of the data structure. We improve catenate as follows. Treat the root spines ($ls(T)$ and $rs(T)$) specially by not storing the actual node $root(T)$ in them. We can now implement the standard push and inject operations by directly modifying the spine-decomposition structure of the affected deque tree using only s-list operations, rather than calling catenate. Now use push and inject in place of catenate wherever possible in the implementation. By doing all this, we remove the eject call from the link operation, and catenate becomes a non-recursive call, utilizing only a constant number of inject and s-list operations and thus taking $O(1)$ amortized time and space per call. In the next section, we shall further improve our data structure to reduce the cost of deletion to $O(\log^* k)$ amortized time and space.

5 Improving the Data Structure

The only non-constant-cost operation in our data structure is deletion. In Sections 2–4, we described how to implement a deletion on a deque of size n via a number of deletions on deques of size $O(\log n)$, yielding a $2^{O(\log^* n)}$ bound. In this section we explain how to modify our data structure so that a deletion from an n-size deque requires at most one deletion from an $O(\log n)$-size deque, thereby reducing our resource bound to $O(\log^* n)$. The bounds for the other operations are unchanged.

For clarity, we use the term *deletion* to refer to the corresponding operation on a deque of n items or a deque tree of n leaves, and we use *subdeletion* to refer to the operation on a smaller ($O(\log n)$-size) structure.

In our original implementation, one deletion can require four subdeletions to modify the spine decomposition of the deque tree plus some number of pulls (previously four), each of which potentially requires three subdeletions. In Section 5.1, we modify the representation of the spine decomposition to reduce to one the number of subdeletions required to update the spine decomposition of a deque tree undergoing a deletion. In order to reduce our resource bound to $O(\log^* n)$, we must then eliminate all of the subdeletions required by pulls. We do this in Sections 5.2 and 5.3 by eliminating the pulls altogether and replacing them with another mechanism to maintain balanced deque trees. Section 5.4 then provides the details on how to make the new implementation persistent.

5.1 Modified Representation

As previously mentioned, the present implementation can require as many as four subdeletions per deletion in order to update the spine decomposition of the deque tree:

1. to remove x, the node being deleted;
2. to remove or update y, the parent of x;
3. to update the parent of y, say g;
4. to remove the terminating node (y) of the right spine terminating at the leftmost child of g.

Our new implementation eliminates three of these subdeletions.

We keep the same spine decomposition as described in Section 3; i.e., the root spines and s-lists still record the same spines as before. We increase the degree invariant maintained at each non-root internal node so that each such node now has degree at least three instead of two; the root node still has degree at least two. Our biggest change, though, is how we store the spines. Whereas before each spine was stored as a deque, we now store the leaf node and the terminating node of each spine separately and keep only the non-leaf and non-terminating (*middle*) nodes of the spine in a deque. Section 5.2 makes use of these changes to provide efficient traversals of deques. Here we investigate the impact of the modifications on the algorithm for maintaining the spine decomposition. Given the record s for any spine, $leaf(s)$, $deq(s)$, and $term(s)$ respectively give us the leaf node, middle deque, and terminating node of s.

First, we no longer need to maintain the actual first and last elements of each deque, since we can access these elements as $leaf(ls(T))$ and $leaf(rs(T))$ each in $O(1)$ time. (T is the deque tree corresponding to the deque.) Additionally, we can now *change in place* the first and second (and last and second-to-last) elements of any deque; that is, we can modify them without performing subdeletions on the spines of T. To change the first element in place

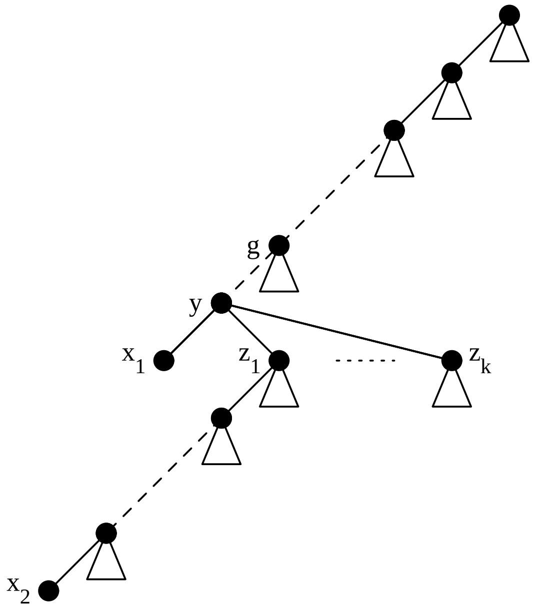

Figure 7: A deque tree T. Dashed lines are paths of nodes that comprise middle deques of spines; triangles represent other subtrees descending from the nodes. The first element of the deque is $x_1 = \mathit{leaf}(\mathit{ls}(T))$; $y = \text{first}(\mathit{deq}(\mathit{ls}(T)))$ is the parent of x_1 in T, and $g = \text{second}(\mathit{deq}(\mathit{ls}(T)))$ is the parent of y in T. Let $l = \mathit{succ}(\mathit{head}(\textit{s-list}(y)))$; l is the left spine terminating at the second child (z_1) of y. Then $x_2 = \mathit{leaf}(l)$ is the second element of the deque. We can change x_1 and x_2 in place in T in $O(1)$ time.

merely requires updating $\mathit{leaf}(\mathit{ls}(T))$. To change the second element in place, we first access $y = \mathit{leaf}(\mathit{deq}(\mathit{ls}(T)))$; this is the parent of $\mathit{leaf}(\mathit{ls}(T))$ in T. We then take from $\textit{s-list}(y)$ the left spine l terminating at the second child of y. The second element of T is $\mathit{leaf}(l)$, and we can modify this directly. This produces a new left spine l', which we store in the proper place in $\textit{s-list}(y)$. In a persistent setting, we now have a new version of y, which we can store in $\mathit{ls}(T)$ by changing $y = \mathit{first}(\mathit{deq}(\mathit{ls}(T)))$ in place; of course, we also have a new version of $\mathit{ls}(T)$. See Figure 7. Changing the last and second-to-last elements of a deque is symmetric. By the same reasoning, we can also use $\mathit{second}(T)$ and $\textit{second-to-last}(T)$ to access the second and second-to-last elements of a deque in $O(1)$ time.

We exploit the new representation of a spine and changing in place to remove three of the four subdeletions cited above. In particular, we no longer have to pop x off the left spine of T, since x is stored apart from that spine. Similarly, any updates to y (the parent of x) and g (the parent of y) can be done *in place* on $\mathit{deq}(\mathit{ls}(T))$. If y is removed (to preserve the degree invariant), rather than pop it off $\mathit{deq}(\mathit{ls}(T))$, we merely substitute a new node for it by changing it *in place*.

The new procedure is best illustrated when we have to delete y to prevent its degree from falling too low. This now happens when the degree of y is exactly three. Let z_1 and z_2 be the middle and rightmost children, respectively, of y. Note that y is the terminating node of the right spine terminating at the leftmost child (y) of g. We eject z_2 from the middle deque of

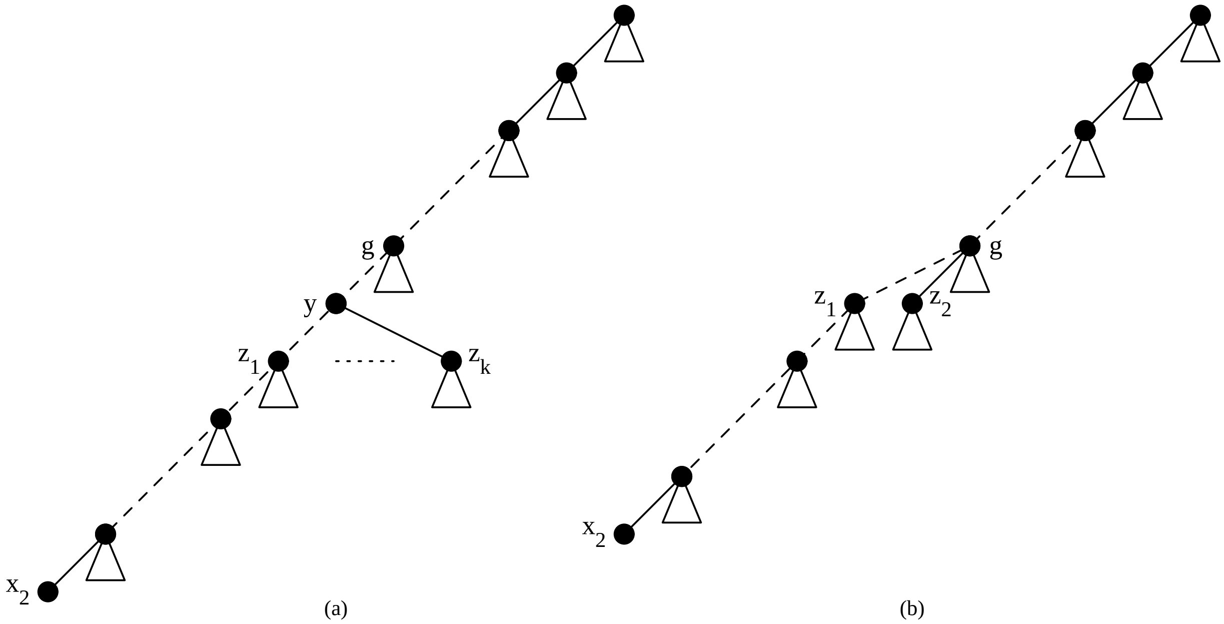

Figure 8: The deque tree of Figure 7 after deleting the leftmost node x_1 if (a) $k > 2$, and (b) $k = 2$. Again, dashed lines represent paths of nodes comprising middle deques of spines. In case (b), y is removed and its children made children of g.

that spine, and z_2 becomes the new terminating node. We also have to change g *in place* to reflect the changes to *s-list*(g): the replacement of the right spine ending at the leftmost child of g and the addition of the left spine terminating at z_2 and the right spine terminating at z_1. As z_1 is the terminating node of the left spine ending at the second-leftmost child of y, we replace y with z_1 *in place* in $deq(ls(T))$. The new section of $ls(T)$ comes from the leaf node and middle deque of the left spine terminating at z_1, and we can catenate the appropriate segments. In fact, this case is the only time a subdeletion occurs. If y remains on the spine, then no subdeletions occur. This is depicted in Figure 8 and described via pseudocode in Figures 9–11. We also provide the pseudocode for push (and inject) and catenate. Note that the higher degree invariant slightly complicates linking deque trees: if the root of the smaller tree being linked has degree two, then we must make each of its children a child of the root of the larger tree; otherwise, we can simply link roots as before.

There are a couple of special cases that the pseudocode does not address. First, in the above case (where y is replaced by z_1), if z_1 is a leaf node then we would have to pop y from $deq(ls(T))$, for in this case z_1 would become $leaf(ls(T))$. To avoid this second subdeletion, however, we instead replace y *in place* with z_2, catenate the middle deque of the left spine terminating at z_2 to $deq(ls(T))$, and add z_1 as the new leftmost child $root(T)$; this last step is just a push. If z_2 is also a leaf, then we do the pop from $deq(ls(T))$ (making $g = \text{first}(ls(T))$), but no eject from the right spine terminating at y is necessary. So, again, only one subdeletion occurs; in this case, it is subdeletion (2) instead of (4). The second special case is when a spine has fewer than three elements and thus cannot support the leaf,

Let l_x (r_x) be the left (right) spine containing the single node x.
// Note they can be constructed in $O(1)$ time.
insert($\textit{s-list}(\textit{root}(T)), \textit{pred}(\textit{head}(\textit{s-list}(\textit{root}(T)))), \textit{ls}(T)$)
insert($\textit{s-list}(\textit{root}(T)), \textit{pred}(\textit{head}(\textit{s-list}(\textit{root}(T)))), r_x$)
$\textit{ls}(T) \leftarrow l_x$

Figure 9: Pseudocode for pushing element x onto the deque described by tree T; the code for inject is symmetric. Double slash (//) precedes comments.

middle deque, and terminating node representation. In this case, all the operations on the spine can be accomplished by dealing with the elements themselves.

5.2 Deque Traversal

We described in Section 5.1 a new implementation of the spine decomposition of a deque tree. We also showed how it could be used to implement new operations on deque trees (such as changing in place). Here we introduce the notion of *traversing* a deque and show how our modified representation facilitates such an action. In Section 5.3 we use deque traversal to help eliminate the pulls.

A deque *traversal* is a sequence of steps that makes available the elements of the deque in head-to-tail or tail-to-head order. We extend this notion to deque trees, and say that a deque tree *traversal* makes available the leaves of the deque tree in left-to-right or right-to-left order. To traverse a deque implemented via spine decomposition, we need to traverse its deque tree, which involves traversing the smaller deques that make up its spines, and so on. Ephemeral deques allow traversal in $O(1)$ time per element. With our original data structure, it is not clear that this is possible. Our modified representation, however, allows deque traversal in $O(1)$ amortized time per element.

We traverse the deque tree in symmetric, i.e., depth-first left-to-right (or right-to-left), order. To visit the leftmost or rightmost child of a node requires performing some part of the traversal of the corresponding spine deque. Visiting a middle child of a node merely requires accessing the s-list of the node (in left-to-right or right-to-left order) to obtain and begin traversing the appropriate spine deque. Since we do not have parent pointers to facilitate the backtracking part of the traversal, we maintain a *traversal stack* that records the state of the traversal; backtracking then involves popping the top record off the stack and continuing the traversal at the node (and spine) indicated.

A deque tree with n leaves has only $c_r n$ internal nodes, for some $0 \leq c_r \leq 1$. We call c_r the *reduction factor*. Since we now store the leaves outside of the spine deques, only these internal nodes occur in smaller deques. Furthermore, since we now also store the terminating nodes of the spines outside of the spine deques, each internal node occurs in at most one smaller deque.

```
if (degree(root(A)) < 3) then // must move all children to preserve degree invariant
    insert(s-list(root(B)), pred(head(s-list(root(B)))), ls(B))
    insert(s-list(root(B)), pred(head(s-list(root(B)))), rs(A))
    insert(s-list(root(B)), pred(head(s-list(root(B)))), tail(s-list(root(A))))
        // it must be that degree(root(A)) = 2
    insert(s-list(root(B)), pred(head(s-list(root(B)))), head(s-list(root(A))))
    ls(T) ← ls(A)
else // can just link roots
    insert(s-list(root(B)), pred(head(s-list(root(B)))), ls(B))
    deq(rs(A)) ← inject(term(rs(A)), deq(rs(A)))
        // A becomes new leftmost child of B
    term(rs(A)) ← root(A)
    insert(s-list(root(B)), pred(head(s-list(root(B)))), rs(A))
    deq(ls(A)) ← inject(term(ls(A)), deq(ls(A)))
    term(ls(A)) ← root(A)
    ls(B) ← ls(A)
```

Figure 10: Pseudocode for linking tree A to the left of tree B. Double slash (//) precedes comments.

Let $T(n)$ be the time required to traverse a deque tree T with n leaves. Such a traversal requires one traversal of the deque of each spine of T plus a constant amount c_w (the *work constant*) of work per node in the deque tree, thus yielding

$$T(n) \leq \sum_i T(x_i) + c_w(1 + c_r)n$$

where x_i is the size of the deque containing the non-leaf/non-terminating node(s) of the ith spine of T. It is easy to show that $T(n) \leq c_t n$ if $c_t \geq \frac{c_w(1+c_r)}{1-c_r}$. We call c_t the *traversal constant*. For the analysis to be valid, we must also have $0 \leq c_r < 1$.

To finish the analysis, we must show how we obtain a suitable reduction factor. Recall that we do not store the leaves or the terminating nodes of spines of the deque tree in the spine deques but instead maintain them separately. Thus a spine consists of a terminating node, a deque, and a leaf node. Also, we have that the degree of a non-root internal node is at least c_d for some $c_d \geq 2$; in Section 5.1 we used $c_d = 3$. These modifications yield $c_r \leq \sum_{i=1}^{\infty} 1/c_d^i = \frac{1}{c_d - 1}$. In particular, $c_d \geq 3$ allows us to achieve $c_t = 3c_w$ using the above traversal strategy. Note that we can achieve $c_t = c_d c_w$. In fact it is straightforward to show that for any traversal method involving t traversals of each smaller deque, we can achieve $c_t = c_d c_w$ by setting $c_d = t + 2$. We use linear-time deque traversal in the next section.

$x \leftarrow leaf(ls(T))$ // leftmost leaf of T
$y \leftarrow$ first$(deq(ls(T)))$ // the parent of x in T
if (degree$(y) > 3$) then
    delete$(s\text{-}list(y), head(s\text{-}list(y)))$ // right spine to x
    $l_z \leftarrow$ delete$(s\text{-}list(y), head(s\text{-}list(y)))$ // left spine to z
    [update y *in place* on $deq(ls(T))$]
    $deq(ls(T)) \leftarrow$ catenate$(deq(l_z), \text{push}(term(l_z), deq(ls(T))))$
    $leaf(ls(T)) \leftarrow leaf(l_z)$
else // degree$(y) = 3$ and must promote both other children z_1 and z_2 of y
    $g \leftarrow$ second$(deq(ls(T)))$ // the parent of y in T
    delete$(s\text{-}list(y), head(s\text{-}list(y)))$ // right spine to x
    $l_{z_1} \leftarrow$ delete$(s\text{-}list(y), head(s\text{-}list(y)))$ // left spine to z_1
    $r_{z_1} \leftarrow$ delete$(s\text{-}list(y), head(s\text{-}list(y)))$ // right spine to z_1
    $l_{z_2} \leftarrow$ delete$(s\text{-}list(y), head(s\text{-}list(y)))$ // left spine to z_2
    $r_{z_2} \leftarrow$ delete$(s\text{-}list(g), head(s\text{-}list(g)))$ // right spine to y
    $(term(r_{z_2}), deq(r_{z_2})) \leftarrow$ eject$(deq(r_{z_2}))$ // delete y from spine, now right spine to z_2
    insert$(s\text{-}list(g), pred(head(s\text{-}list(g))), r_{z_2})$
    insert$(s\text{-}list(g), pred(head(s\text{-}list(g))), l_{z_2})$
    insert$(s\text{-}list(g), pred(head(s\text{-}list(g))), r_{z_1})$
    [update g *in place* on $deq(ls(T))$]
    [change y to $term(l_{z_1}) = z_1$ *in place* on $deq(ls(T))$]
    $deq(ls(T)) \leftarrow$ catenate$(deq(l_{z_1}), deq(ls(T)))$
    $leaf(ls(T)) \leftarrow leaf(l_{z_1})$

Figure 11: Pseudocode for deletion of leftmost leaf of T. Double slash (//) precedes comments.

5.3 Global Rebuilding

All that remains is to eliminate the subdeletions incurred by the pulls. We accomplish that in this section by replacing the pull operations with a new mechanism to ensure that the deque trees are balanced.

The new technique is *global rebuilding*. Global rebuilding and the related idea of *partial rebuilding*, elucidated by Overmars [31], are tools initially developed and used to dynamize static data structures [1, 2, 9, 21]. They have also been used to turn amortized bounds into worst case bounds [3, 10, 16, 20, 22] and to improve the space requirements of a data structure [41]. The basic idea of global rebuilding is to maintain with each data structure a secondary copy of the data structure that is being gradually "rebuilt." While updates are made to the primary data structure, reducing its balance, the secondary structure is being created a few steps at a time. By the time enough updates are made to the primary structure to violate its balance condition, the secondary structure is ready; i.e., the secondary structure is then a balanced (perhaps "perfectly" so) version of the primary structure, and it then replaces the primary structure. In his monograph, Overmars [31] describes how to use global rebuilding to dynamize data structures that allow insertions and deletions. His methods are not immediately extensible to allow generalized catenable data structures to be globally rebuilt. We can apply his basic idea, however, to rebuild deque trees while they are being subjected to catenations and deletions.

Globally rebuilding a deque tree entails traversing it over a sequence of operations and constructing a new, flat deque tree containing the same leaves. With each deque tree T, we associate a secondary tree $sec(T)$ and a buffer BUF, the latter implemented as a queue. We maintain an *ongoing traversal* of T; that is we perform some number of steps of the traversal with each operation on T and continue the traversal during the next such operation. We describe in detail how to do this in Section 5.4.

With every operation (link or delete) on T, we perform some number (k) of *rebuilding steps*, to be described below. Also, we add the operation performed on T to the back of BUF. That is, we inject into BUF a record describing the operation (e.g., pop, eject, or catenate—with the latter also storing the respective deque being catenated). With each such injection we perform another k rebuilding steps.

A *rebuilding step* is as follows. If the traversal of T is not complete, then we perform one unit of work on that traversal. Each leaf encountered is copied to $sec(T)$, and $sec(T)$ is built as a flat tree using the spine decomposition structure. If the traversal of T is complete, then we remove (pop) the front element from BUF. If it is a pop (eject), we perform a left (right) deletion on $sec(T)$. Similarly, if it is a push (inject), we add the new element as the new leftmost (rightmost) child of the root of $sec(T)$. Otherwise, the operation is a catenate of some tree S to T. In this case, we begin traversing S, adding its leaves to the left (right) of the leftmost (rightmost) leaf of $sec(T)$ if S was linked to the left (right) of T. Again, during this traversal, operations on T are queued onto BUF. It is interesting to note that any accumulated rebuilding of S is ignored by our algorithm; as S is traversed and its leaves added to $sec(T)$, we merely continue buffering operations on T. The ability to do this and have $sec(T)$ rebuilt in time to take over from T is what allows our data structure to undergo global rebuilding; the requirement that the rebuilding of the linked data structure somehow be saved and used is what makes global rebuilding hard to apply in general to catenable data structures.

It is straightforward to show via induction that if we finish a traversal and BUF is empty, then $sec(T)$ is a *flat version of* T; that is, $sec(T)$ contains the same leaves in left-to-right order as T, and each leaf in $sec(T)$ is a child of the root of $sec(T)$. Furthermore, since $sec(T)$ is always flat, we can always make the appropriate modification to $sec(T)$ in $O(1)$ time. Once $sec(T)$ is a flat version of T, we replace T by $sec(T)$ and discard the old T. We begin creating a new $sec(T)$ with the first link to the new T.

We showed in Section 5.2 that it takes no more than $c_t n$ time to traverse a deque tree with n leaves, for some some traversal constant c_t. We assume that it takes unit time either to buffer an operation or to pop and perform an operation from BUF. The proof that the above scheme maintains balanced deque trees is analogous to the analysis of the effect of pulling. Depth-increasing links provide a cushion of leaves that must be deleted before depth violations can occur, and catenations merely delay the violations. Meanwhile, the rebuilding progresses quickly enough to finish before any depth violation occurs. Whereas pulling is a local operation, however, affecting only the area around the root of a tree, global rebuilding is, as the name implies, global. The analogue to Lemma 2.1 therefore requires a more complicated proof.

Lemma 5.1 *If $k \geq 2c_t$ then a non-root node in a deque tree never becomes a root node of a deque tree T unless T is a singleton node.*

Proof: New root nodes are created by the makedeque operation and by linking two singleton trees together. Consider when a root node r of some tree becomes a non-root node due to a link. Assume that at some point in the future r becomes a root node of some tree X. Now consider the most recent tree S formed before X by linking a tree R containing r to another tree R'. Let r' be the root of R' (and thus of S), and note that $|R'| \geq |R|$. By this definition, r' never becomes a non-root node before X is created, but it may acquire new children as a result of future links. Denote by T the evolving tree rooted at r', initially S, as leaves are deleted from it and other trees linked to it.

Let a total of x leaves be added to T via links after the formation of S; let m be the number of operations that transform T from S to X. For r to become a root node, all the leaves of R' plus all these x leaves must first be deleted from T, resulting in $k(|R'|+x+m)$ rebuilding steps being applied to T. No more than $c_t(|S|+x)+m$ rebuilding steps are needed to make $sec(T)$ a flat version of T, however. Linking by size yields $|R'| \geq |S|/2$. Thus, if $k \geq 2c_t$, there are always at least as many rebuilding steps remaining as required to produce a flat version of T, and so $sec(T)$ takes over as a flat version of T before T becomes X. If r occurs in $sec(T)$, then it does so as a leaf. For r to become the root of $sec(T)$, therefore, all the other leaves of $sec(T)$ must first be deleted, leaving the single node r as tree X. □

Theorem 5.2 *If $1 < c < 2$ and $k \geq \frac{2c_t}{2-c}$, then $|T| \geq c^{d_T}$.*

Proof: Apply the same argument as in the proof of Theorem 2.2. Using Lemma 5.1 and adding the new terms to account for buffered updates and the traversal constant yields the desired bounds. □

In summary, we have shown how to use global rebuilding in place of pulls to keep the deque trees balanced. This eliminates all of the subdeletions associated with the pull operations. Note that therefore it is no longer necessary to maintain a secondary list of non-leaf children with each node. The operations on s-lists simply become deque operations (without catenate). Thus, a deque of n elements is decomposed into a collection of only deques, and the deques that need to be catenated are all of size logarithmic in n.

5.4 Persistence Details

We now consider making the improved implementation of deque trees confluently persistent. As before, we use structural abstraction to refer to the spines and s-lists by their version numbers. The spines are now persistent records, each containing the version numbers of its leaf, middle confluently persistent deque, and terminating node. Again an internal node occurs in at most two spines, but now in at most one confluently persistent deque. Each such node contains the version number of its s-list, and again we store the actual version number in the copy of the node that occurs in the middle deque (to prevent cascading update sequences). If the node occurs as the terminating node of both spines containing it, we store the version number of its s-list in the terminating node of the left spine. Again, an inspection of how we access these nodes reveals that if we ever need to access an s-list via the terminating node of a right spine, we have immediate access to the corresponding left spine as well. Thus no further deque operations are necessary. As before, all operations occur in the immediate proximity of the left spines or right spines of the trees, so cascading update sequences do not occur. In particular, the operation of changing in place can be performed as described above without causing cascading updates.

To implement global rebuilding, we store with each persistent deque tree T the version number of $sec(T)$, a pointer to the place in $sec(T)$ where the last leaf was added, the version number of the tree being traversed and copied to $sec(T)$, a persistent stack to implement the traversal of that tree, and a persistent queue to represent the BUF associated with T. Stacks and queues are bounded indegree structures, so they can be made persistent by the methods of Driscoll et al. [18]; each persistent stack or queue operation thus requires $O(1)$ amortized time and space. Recall that traversing a deque tree is a recursive operation: we are also traversing the deques representing the middle nodes of spines and so on. Each such traversal requires a persistent traversal stack, and the global state of the traversal of the top-level tree is recorded in the global persistent stack (containing records "pointing to" the persistent traversal stacks via version numbers) for that tree. While a traversal step might percolate all the way through the global stack, we know that the total number of traversal steps is linear in the size of the top-level deque tree. This is the key to global rebuilding. Each time we update T, we perform the required rebuilding steps and also update the traversal stacks, $sec(T)$, etc. All of these operations are done persistently. When $sec(T)$ becomes a flattened version of T, we replace the appropriate version (the one just created) of T by $sec(T)$. To do this, we simply return the version numbers describing $sec(T)$: those for its spines and the new $sec(T)$, BUF, etc. that we begin creating.

As in our original implementation, structural decomposition yields an efficient data structure.

Theorem 5.3 *A pop or eject on a confluently persistent deque of size n can be performed in $O(\log^* n)$ amortized time and space; catenate and makedeque each require $O(1)$ amortized time and space; first and last require $O(1)$ time.*

PROOF: By the above discussion and similar arguments as used in the proof of Theorem 4.1. □

Corollary 5.4 *Pop and eject require $O(\log^* k)$ amortized time and space, where k is the total number of deque operations performed so far.*

PROOF: Since $n = O(2^k)$, $\log^* n = O(\log^* k)$. □

Chuang and Goldberg [10] show how to implement fully persistent deques so that each fully persistent deque operation takes $O(1)$ worst-case time; Gajewska and Tarjan [20], in fact, anticipate this result. Chuang and Goldberg [10] break such deques into two fully persistent stacks, much like Gajewska and Tarjan [20] break heap-ordered deques into two heap-ordered stacks, and use global rebuilding to keep the stacks non-empty. We can use these fully persistent deques in our implementation, making all of our resource bounds worst-case.

Theorem 5.5 *A pop or eject on a confluently persistent deque of size n can be performed in $O(\log^* k)$ worst-case time and space; catenate and makelist each require $O(1)$ worst-case time and space; first and last require $O(1)$ worst-case time.*

PROOF: As mentioned in Section 5.3, our modified representation uses only persistent deques, not s-lists, to implement the spine decomposition. Therefore, we can apply the Chuang and Goldberg [10] implementation of fully persistent deques to our data structure. □

In fact, using the Chuang and Goldberg [10] fully persistent deques, our solution to the confluently persistent deque problem can be implemented purely functionally, with no side effects.

We conclude by noting that the confluently persistent output-restricted deques of Driscoll, Sleator, and Tarjan [19] require $O(\log\log k)$ amortized time and space for catenate and $O(1)$ amortized time and space for the other operations. We can implement output-restricted deques using a *left-spine decomposition*, i.e., storing with each node a list of left spines terminating at its non-leftmost children. No right spines are stored. With this representation, no eject operations are needed; we only pop from the spine deques (in the case above where y is removed to preserve the degree invariant and z_1 and z_2 are leaves, thereby necessitating popping y from $deq(ls(T))$). In this way, we may recurse any number of levels in our structure and then substitute the output-restricted deques of Driscoll, Sleator, and Tarjan [19] for the middle deques in the spines. We can thus obtain an $O(\log^{(i)} k)$ amortized time and space bound for catenate, for any desired constant i, and $O(1)$ amortized time and space bounds for the other operations. This yields strictly improved bounds for the output-restricted case in addition to our solution to the general deque problem.

6 Lower Bounds for Pulls

6.1 Linking by Size

Here we show that performing k pulls per deletion, for any constant k, can yield deque trees with logarithmic-length spines when we link deque trees by size. The construction

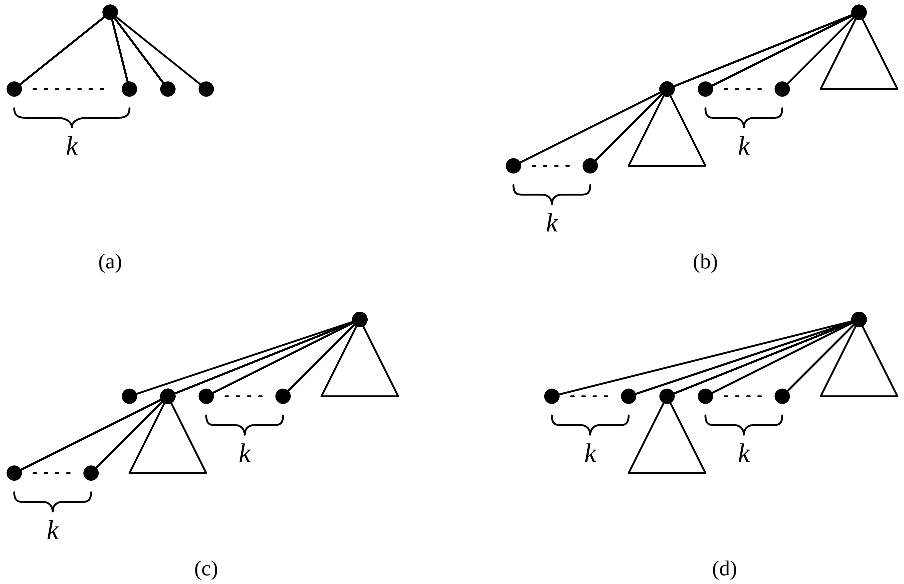

Figure 12: (a) A tree S_0; (b)–(d) Constructing a tree S_i by (b) linking two S_{i-1} trees, (c) adding a new leftmost child to the root, and (d) performing k pulls and a deletion. Triangles denote subtrees.

extends to the global rebuilding case. Therefore, further improvements to the resource bounds will require either a deeper analysis of the structure of the lower levels of the deque trees (the smaller trees representing spine deques) or a different data structure. The following construction is due to Sleator [36].

Theorem 6.1 *Linking by size and performing k pulls per deletion, for $k = O(1)$, can yield an infinite set of deque trees such that any deque tree T in the set has a spine of length $\Omega(\log |T|)$.*

PROOF: Consider the tree S_0 consisting of a root node with $k+2$ children, as in Figure 12(a). We recursively construct a sequence of trees as follows: tree S_i (for $i > 0$) is formed by linking two S_{i-1} trees, adding a new leftmost child to the root of the resulting tree, and then performing k pulls and a deletion. See Figure 12(b)–(d). It is easy to prove by induction that:

1. $|S_i| = 2^i(k+2)$;

2. The length of the left spine terminating at the leftmost non-leaf child of the root of S_i is i.

□

As an aside, we need two extra children of the root of S_0 so that the k pulls applied to the construction of S_1 (in Figure 12(d)) do not make the root of S_0 unary and thus remove it.

The above construction can be extended to handle the case in which k_1 *left pulls* (like the previous pull operations) and k_2 *right pulls* (the symmetric operations on the right side) are performed per deletion, for constants k_1 and k_2. For this case, S_0 is the tree containing a root with $k_1 + k_2 + 2$ children (k_1 to the left and k_2 to the right of the arbitrary subtree in the middle). The construction also shows the same lower bound for global rebuilding: start with a root with ck children for some appropriate constant c. Since each link-insert-delete sequence doubles the size of the tree while doing only a constant amount of rebuilding, the rebuilding never finishes, and the primary tree always has a logarithmic-length spine.

6.2 Linking on the Right

Driscoll, Sleator, and Tarjan [19] use a different linking strategy to produce confluently persistent output-restricted deques. They always use *right linking*; i.e., they always make the root of the right tree the new rightmost child of the root of the left tree. They define a tree to be *c-collapsible* if performing c pulls per deletion maintains that the leftmost child of the root is a leaf. If a tree T_r is linked to the right of a c-collapsible tree T_l and $|T_r| \leq (c-1)|T_l|$, they prove that the resulting tree is also c-collapsible. By maintaining an output-restricted deque as a sequence of c-collapsible trees of geometrically increasing size, they obtain an $O(\log \log n)$ resource bound on the output-restricted deque operations.

The c-collapsible trees in the sequence are themselves leaves of a top-level search tree. Here we show that this hybridization is crucial. In particular, we prove that as long as $k = O(1)$, performing k pulls per deletion when right linking trees can always produce a left root spine of arbitrary length. Therefore, pulls alone do not suffice to produce an efficient data structure, even for the output-restricted case. The following construction is also due to Sleator [36].

Theorem 6.2 *Right linking and performing k pulls per deletion, for $k = O(1)$, can yield an infinite set of deque trees such that any deque tree T in the set has a left root spine of length $\Omega(|T|)$.*

PROOF: Let the tree L_0 be the singleton node. For $i > 0$, the root of tree L_i has the root of tree L_{i-1} as its left child and a leaf node as its right child. See Figure 13(a). Tree T_i, for $i > 0$, consists of tree L_i augmented with $2k$ extra leaf nodes as the leftmost children of the root; see Figure 13(b). Clearly, we can construct tree T_1 by right linking $2k + 2$ singleton nodes together.

We construct tree T_{i+1}, for $i \geq 1$, as follows. Link tree T_i to the right of tree L_1; perform k pulls and a deletion, and then perform another k pulls and deletion, forming tree X_{i+1}; finally, link tree L_0 to the right of tree X_{i+1}. See Figure 13(c)–(e).

The length of the left spine terminating at the leftmost non-leaf child of the root of tree T_n is $n - 1$; further, $|T_n| = n + 2k + 1$. For $n > 2k^2$, performing $2k$ deletions (and thus $2k^2$ pulls as well) on tree T_n yields a tree of $n + 2k + 1 - 2k^2 = O(n)$ nodes with a left root spine of length $n - 2k^2 = \Omega(n)$. □

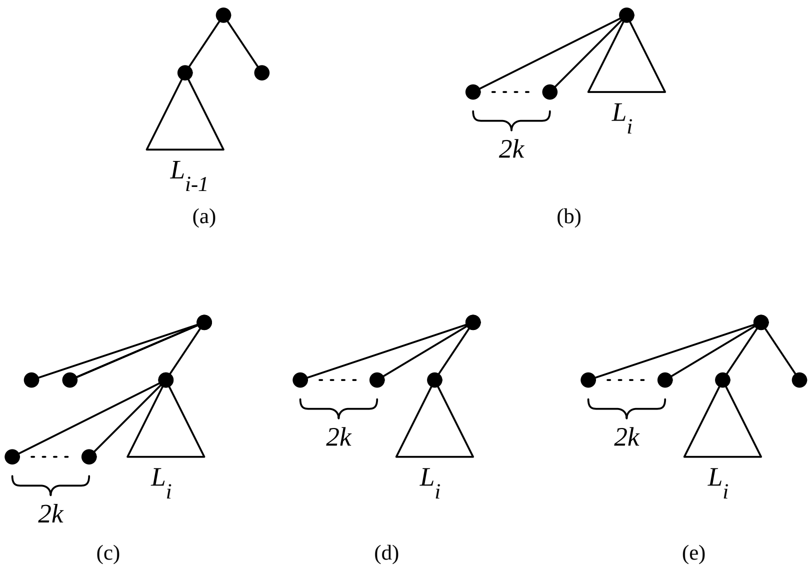

Figure 13: (a) Tree L_i; (b) Tree T_i; (c)–(e) Constructing tree T_{i+1} by (c) linking tree T_i to the right of tree L_1, (d) performing k pulls and a deletion twice, and (e) right linking tree L_0 to the result of step (d).

7 Conclusion

We have shown how to make deques confluently persistent so that pop and eject each require $O(\log^* k)$ amortized time and space and the other operations each take constant amortized time and space. We can also make these bounds worst-case, and we can implement our data structure purely functionally, with no side effects. We utilize two types of *data-structural bootstrapping*—*structural abstraction* and *structural decomposition*—to achieve our results, thus further showing the usefulness of this technique in designing high-level data structures. We mention four avenues for further exploration.

First, still open is the problem of implementing confluently persistent deques with better resource bounds. There is currently no reason to believe that constant time and space per operation is not possible; if it is not, however, perhaps an $O(\alpha(k))$ solution is attainable, where α is a functional inverse of the Ackermann function [37]. Section 6 suggests that different analyses or data structures are necessary for such improvements. Kosaraju [25] has an $O(1)$ worst-case time per operation solution to the related catenable heap-ordered deque problem described by Buchsbaum, Sundar, and Tarjan [6]. While his methods are not immediately extensible to the confluently persistent deque problem, we suggest investigating his data structure as a first step.

Second, data-structural bootstrapping seems to be a powerful tool for use in designing high-level data structures out of more basic pieces. We use two types of it in this work, while other authors (e.g., [6, 12, 19, 24]) also use it in one form or another. Formalizing this technique and finding more applications of it seems a worthwhile goal.

Third, providing general techniques for making various data structures confluently persistent remains an open problem. Of course, defining the notion of confluent persistence is the first step, as it is not well-defined for all data structures. Some linking operation is necessary for it to be considered; it might make no sense, for instance, to talk about confluently persistent arrays. ("Linking" operations that could make arrays confluently persistent include such things as summing the respective entries of each array position and performing convolutions.)

Finally, as mentioned in Section 1, some purely functional languages effectively implement all data structures persistently, regardless of the actual application. Using techniques that make data structures persistent to facilitate the implementation of such languages is an important open area for future research.

Acknowledgements

We thank Jim Driscoll and Danny Sleator for some early ideas and discussions that motivated this work. In particular, the notion of decomposing deque trees into paths and somehow working with these paths is suggested by Driscoll, Sleator, and Tarjan [19].

References

[1] A. Andersson. Improving partial rebuilding by using simple balance criteria. In *Proc. 1st Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer-Verlag, 1989.

[2] A. Andersson. Maintaining α-balanced trees by partial rebuilding. *International Journal of Computer Mathematics*, 38(1-2):37–48, 1991.

[3] A. Andersson, C. Icking, R. Klein, and T. Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28(2):165–78, 1990.

[4] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[5] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1:301–58, 1980.

[6] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 40–9, 1992.

[7] A. L. Buchsbaum and R. E. Tarjan. Confluently persistent deques via data structural bootstrapping. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 155–64, 1993.

[8] B. Chazelle. How to search in history. *Information and Control*, 77(1-3):77–99, 1985.

[9] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structure technique. *Algorithmica*, 1(2):133–62, 1986.

[10] T.-R. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. To appear in *Proc. 6th ACM Conf. on Functional Programming Languages and Computer Architecture*, 1993.

[11] R. Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202–20, 1986.

[12] P. F. Dietz. Maintaining order in a linked list. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 122–7, May 1982.

[13] P. F. Dietz. Fully persistent arrays (extended abstract). In *Proc. 1st Workshop on Algorithms and Data Structures*, number 382 in Lecture Notes in Computer Science, pages 67–74. Springer-Verlag, August 1989. Submitted to *J. Alg.*

[14] P. F. Dietz. Finding level-ancestors in dynamic trees. In *Proc. 2nd Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 32–40. Springer-Verlag, 1991.

[15] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 78–88, 1991.

[16] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–72, May 1987.

[17] D. P. Dobkin and J. I. Munro. Efficient uses of the past. *Journal of Algorithms*, 6(4):455–65, 1985.

[18] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[19] J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan. Fully persistent lists with catenation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 89–99, 1991. Submitted to *J. ACM.*

[20] H. Gajewska and R. E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, April 1986.

[21] I. Galperin and R. L. Rivest. Scapegoat trees. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–74, 1993.

[22] R. Hood and R. Melville. Real-time queue operations in pure LISP. *Information Processing Letters*, 13(2):50–4, 1981.

[23] D. E. Knuth. *The Art of Computer Programming*, volume 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA, second edition, 1973.

[24] S. R. Kosaraju. Real-time simulation of concatenable double-ended queues by double-ended queues. In *Proc. 11th ACM Symp. on Theory of Computing*, pages 346–51, 1979.

[25] S. R. Kosaraju. Private communication. 1992.

[26] E. W. Myers. AVL dags. Technical Report 82-9, U. Arizona Dept. of Computer Science, Tucson, 1982.

[27] E. W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–8, 1983.

[28] E. W. Myers. Efficient applicative data types. In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 66–75, 1984.

[29] M. H. Overmars. Searching in the past, I. Technical Report RUU-CS-81-7, U. Utrecht Dept. of Computer Science, 1981.

[30] M. H. Overmars. Searching in the past, II: General transforms. Technical Report RUU-CS-81-9, U. Utrecht Dept. of Computer Science, 1981.

[31] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[32] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–77, 1983.

[33] N. Sarnak. *Persistent Data Structures*. PhD thesis, Department of Computer Science, New York University, 1986.

[34] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–79, 1986.

[35] R. Sethi. *Programming Languages Concepts and Constructs*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1989.

[36] D. D. Sleator. Private communication. 1992.

[37] R. E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[38] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–18, 1985.

[39] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[40] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–12, 1984.

[41] M. J. van Kreveld and M. H. Overmars. Union-copy structures and dynamic segment trees. Technical Report RUU-CS-91-5, U. Utrecht Dept. of Computer Science, February 1991. To appear in *J. ACM*.

[42] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17:81–4, August 1983.

[43] D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–94, 1984.