Simple Confluently Persistent Catenable Lists.

(Extended Abstract)

Haim Kaplan[1], Chris Okasaki[2*], and Robert E. Tarjan[3**]

[1] AT&T labs, 180 Park Ave, Florham Park, NJ. hkl@research.att.com
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. cokasaki@cs.cmu.edu
[3] Department of Computer Science, Princeton University, Princeton, NJ 08544 and InterTrust Technologies Corporation, Sunnyvale, CA 94086. ret@cs.princeton.edu.

Abstract. We consider the problem of maintaining persistent lists subject to concatenation and to insertions and deletions at both ends. Updates to a persistent data structure are nondestructive–each operation produces a new list incorporating the change while keeping intact the list or lists to which it applies. Although general techniques exist for making data structures persistent, these techniques fail for structures that are subject to operations, such as catenation, that combine two or more versions. In this paper we develop a simple implementation of persistent double-ended queues with catenation that supports all deque operations in constant amortized time.

1 Introduction

Over the last fifteen years, there has been considerable development of *persistent* data structures, those in which not only the current version, but also older ones, are available for access (*partial persistence*) or updating (*full persistence*). In particular, Driscoll, Sarnak, Sleator, and Tarjan [5] developed efficient general methods to make pointer-based data structures partially or fully persistent, and Dietz [3] developed an efficient general method to make array-based structures fully persistent.

These general methods support updates that apply to a single version of a structure at a time, but they do not accommodate operations that combine two different versions of a structure, such as set union or list catenation. Driscoll, Sleator, and Tarjan [4] coined the term *confluently persistent* for fully persistent structures that support such combining operations. An alternative way to obtain persistence is to use strictly functional programming (By strictly functional we mean that lazy evaluation, memoization, and other such techniques are not

* Supported by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.
** Research at Princeton University partially supported by NSF Grant No. CCR-9626862.

allowed). For list-based data structure design, strictly functional programming amounts to using only the LISP functions CAR, CONS, CDR. Strictly functional data structures are automatically persistent, and indeed confluently persistent.

A simple but important problem in data structure design that makes the issue of confluent persistence concrete is that of implementing persistent double-ended queues (deques) with catenation. A series of papers [4, 2] culminated in the work of Kaplan and Tarjan [8], who developed a confluently persistent implementation of deques with catenation that has a worst-case constant time and space bound for any deque operation, including catenation. The Kaplan-Tarjan data structure and its precursors obtain confluent persistence by being strictly functional.

If all one cares about is persistence, strictly functional programming is unnecessarily restrictive. In particular, Okasaki [12, 11, 13] observed that the use of lazy evaluation in combination with memoization can lead to efficient functional (but not strictly functional) data structures that are confluently persistent. In order to analyze such structures, Okasaki developed a novel kind of debit-based amortization. Using these techniques and weakening the time bound from worst-case to amortized, he was able to considerably simplify the Kaplan-Tarjan data structure, in particular to eliminate its complicated skeleton that encodes a tree extension of a redundant digital numbering system.

In this paper we explore the problem of further simplifying the Kaplan-Tarjan result. We obtain a confluently persistent implementation of deques with catenation that has a constant amortized time bound per operation. Our structure is substantially simpler than the original Kaplan-Tarjan structure, and even simpler than Okasaki's structure: whereas Okasaki requires efficient persistent deques without catenation as building blocks, our structure is entirely self-contained. Furthermore our analysis uses a standard credit-based approach. As compared to Okasaki's method, our method requires an extension of the concept of memoization: we allow any expression to be replaced by an equivalent expression.

The remainder of this extended abstract consists of five sections. In Section 2, we introduce terminology and concepts. In Section 3, we illustrate our approach by developing a persistent implementation of deques without catenation. In Section 4, we develop our solution for deques with catenation. We conclude in Section 5 with some remarks and open problems.

2 Preliminaries

The objects of our study are lists. As in [8] we allow the following operations on lists:

MAKELIST(x):	return a new list containing the single element x.
PUSH(x, L):	return a new list formed by adding element x to the front of list L.
POP(L):	return a pair whose first component is the first element on list L and whose second component is a list containing the second through last elements of L.
INJECT(L, x):	return a new list formed by adding element x to the back of list L.
EJECT(L):	return a pair whose first component is a list containing all but the last element of L and whose second component is the last element of L.
CATENATE(L, R):	return a new list formed by catenating L and R, with L first.

We seek implementations of these operations (or specific subsets of them) on persistent lists: any operation is allowed on any previously constructed list or lists at any time. For discussions of various forms of persistence see [5]. A *stack* is a list on which only PUSH and POP are allowed. A *queue* is a list on which only INJECT and POP are allowed. A *steque* (*stack-ended queue*) is a list on which only PUSH, POP, and INJECT are allowed. Finally, a *deque* (*double-ended queue*) is a list on which all four operations PUSH, POP, INJECT, and EJECT are allowed. For any of these four structures, we may or may not allow catenation. If catenation is allowed, PUSH and INJECT become redundant, since they are special cases of catenation, but it is sometimes convenient to treat them as separate operations because they are easier to implement than general catenation.

We say a data structure is *strictly functional* if it can be built and manipulated using the LISP functions CAR, CONS, CDR. That is, the structure consists of a set of immutable nodes, each either an atom or a node containing two pointers to other nodes, with no cycles of pointers. The nodes we use to build our structures actually contain a fixed number of fields; reducing our structures to two fields per node by adding additional nodes is straightforward. Various nodes in our structure represent lists. To obtain our results, we extend strict functionality by allowing, in addition to CAR, CONS, CDR, the operation of replacing a node in a structure by another node representing the same list. Such a replacement can be performed in an imperative setting by replacing all the fields in the node, for instance in LISP by using REPLACA and REPLACD. Replacement can be viewed as a generalization of memoization. In our structures, any node is replaced at most twice, which means that all our structures can be implemented in a write-once memory. (It is easy to convert an algorithm that overwrites any field only a fixed constant number of times into a write-once algorithm, with only a constant factor loss of efficiency.)

To perform amortized analysis, we use a standard potential-based framework. We assign to each configuration of the data structure (the totality of nodes currently existing) a *potential*. We define the amortized cost of an operation to be its actual cost plus the net increase in potential caused by performing the operation. In our applications, the potential of an empty structure is zero and

the potential is always non-negative. It follows that, for any sequence of operations starting with an empty structure, the total actual cost of the operations is bounded above by the sum of their amortized costs. See the survey paper [14] for a more complete discussion of amortized analysis.

3 Noncatenable Deques

In this section we describe an implementation of persistent noncatenable deques with a constant amortized time bound per operation. The structure is based on the analogous Kaplan-Tarjan structure [8] but is much simpler. The result presented here illustrates our technique for doing amortized analysis of a persistent data structure. At the end of the section we comment on the relation between the structure proposed here and previously existing solutions.

3.1 Representation

Here and in subsequent sections we say a data structure is *over* a set A if it stores elements from A. Our representation is recursive. It is built from bounded-size deques called *buffers*, each containing at most three elements. Buffers are of two kinds: *prefixes* and *suffixes*. A nonempty deque d over A is represented by an ordered triple consisting of a prefix over A, denoted by $pr(d)$; a (possibly empty) *child deque* of ordered *pairs* over A, denoted by $c(d)$; and a suffix over A, denoted by $sf(d)$. Each pair consists of two elements from A. The child deque $c(d)$, if nonempty, is represented in the same way. We define the set of *descendants* $\{c^i(d)\}$ of a deque d in the standard way—namely, $c^0(d) = d$ and $c^{i+1}(d) = c(c^i(d))$, provided $c^i(d)$ and $c(c^i(d))$ exist.

The order of elements in a deque is defined recursively to be the one consistent with the order of each triple, each buffer, each pair, and each child deque. Thus, the order of elements in a deque d is first the elements of $pr(d)$, then the elements of each pair in $c(d)$, and finally the elements of $sf(d)$.

In general the representation of a deque is not unique—the same sequence of elements may be represented by triples that differ in the sizes of their prefixes and suffixes, as well as in the contents and representations of their descendant deques. Whenever we refer to a deque d we actually mean a particular representation of d, one that will be clear from the context.

The pointer structure for this representation is straightforward: a node representing a deque d contains pointers to $pr(d)$, $c(d)$, and $sf(d)$. Note that, since the node representing $c^i(d)$ contains a pointer to $c^{i+1}(d)$, the pointer structure of d is essentially a linked list of its descendants. By *overwriting* $pr(d)$, $c(d)$, or $sf(d)$ with a new prefix, child deque, or suffix respectively, we mean assigning a new value to the corresponding pointer field in d. As discussed in Section 2, we will always overwrite fields in such a way that the sequence of elements stored in d remains the same and the change is only in the representation of d. By *assembling* a deque from a prefix p, a child deque y, and a suffix s, we mean creating a new node with pointers to p, y, and s.

3.2 Operations

We describe in detail only the implementation of POP; the detailed implementations of the other operations are similar. Each operation on a buffer is implemented by creating an appropriately modified new copy.

POP(d): If $pr(d)$ is empty and $c(d)$ is nonempty, then let $((x, y), c') =$ POP$(c(d))$ and $p' =$ INJECT$(y,$ INJECT$(x, pr(d)))$. Overwrite $pr(d)$ with p' and $c(d)$ with c'. Then if $pr(d)$ is nonempty, perform $(x, p) =$ POP$(pr(d))$, return x as the item component of the result, and assemble the deque component of the result from p, $c(d)$, and $sf(d)$. Otherwise, the only part of d that is nonempty is its suffix. Perform $(x, s) =$ POP$(sf(d))$ and return x together with a deque assembled from an empty prefix, an empty child deque, and s.

Note that the implementation of POP is recursive: POP can call itself once. The implementation of EJECT is symmetric to the implementation of POP. The implementation of PUSH is as follows. Check whether the prefix contains three elements; if so, recursively push a pair onto the child deque. Once the prefix contains at most two elements, add the new element to the front of the prefix. INJECT is symmetric to PUSH.

3.3 Analysis

We call a buffer *red* if it contains zero or three elements, and *green* if it contains one or two elements. A node representing a deque can be in one of three possible states: rr, if both of its buffers are red; gr, if one buffer is green and the other red; and gg, if both buffers are green. We define $\#rr$, $\#gr$, and $\#gg$ to be the numbers of nodes in states rr, gr, and gg, respectively. Note that deques can share descendants. For instance, d and $d' =$ POP(d) can both contain pointers to the same child deque. We count each shared node only once, however. We define the potential Φ of a collection of deques to be $3 * (\#rr) + \#gr$.

To analyze the amortized cost of POP, we assume that the actual cost of a call to POP, excluding the recursive call, is one. Thus if a top level POP invokes POP recursively $k-1$ times, the total actual cost is k.

Assume that a top level POP invokes $k-1$ recursive POPs. The ith invocation of POP, for $1 \leq i \leq k-1$, overwrites $c^{i-1}(d)$, changing its state from rr to gr or from gr to gg. Then it assembles its result, which creates a new node whose state (gr or gg) is identical to the state of $c^{i-1}(d)$ after the overwriting. In summary, the ith recursive call to POP, $1 \leq i \leq k-1$, replaces an rr node with two gr nodes or a gr node with two gg nodes, and in either case decreases the potential by one. The last call, POP$(c^{k-1}(d))$, creates a new node that can be in any state, and so increases the potential by at most three. Altogether, the k invocations of POP increase the potential by at most $3-(k-1)$. Since the actual cost is k, the amortized cost is constant.

A similar analysis shows that the amortized cost of PUSH, INJECT, and EJECT is also constant. Thus we obtain the following theorem.

Theorem 1. *Each of the operations* PUSH, POP, INJECT, *and* EJECT *on the data structure defined in this section takes* $O(1)$ *amortized time.*

3.4 Related Work

The structure just described is based on the Kaplan-Tarjan structure of [8, Section 3], but simplifies it in three ways. First, the skeleton of our structure (the sequence of descendants) is a stack; in the Kaplan-Tarjan structure, this skeleton must be partitioned into a stack of stacks in order to support worst-case constant-time operations (via a redundant binary counting mechanism). Second, the recursive changes to the structure to make its nodes green are one-sided, instead of two-sided: in the original structure, the stack-of-stacks mechanism requires coordination to keep both sides of the structure in related states. Third, the maximum buffer size is reduced, from five to three. In the special case of a steque, the maximum size of the suffix can be further reduced, to two. In the special case of a queue, both the prefix and the suffix can be reduced to maximum size two.

There is an alternative, much older approach that uses incremental recopying to obtain persistent deques with worst-case constant-time operations. See [8] for a discussion of this approach. The incremental recopying approach yields an arguably simpler structure than the one presented here, but our structure generalizes to allow catenation, which no one knows how to implement efficiently using incremental recopying. Also, our structure can be extended to support access, insertion, and deletion d positions away from the end of a list in $O(\log d)$ amortized time, by applying the ideas in [9].

4 Catenable Deques

In this section we show how to extend our ideas to support catenation. Specifically, we describe a data structure for catenable deques that achieves an $O(1)$ amortized time bound for PUSH, POP, INJECT, EJECT, and CATENATE. Our structure is based upon an analogous structure of Okasaki [13], but simplified to use constant-size buffers.

4.1 Representation

We use three kinds of buffers: *prefixes*, *middles*, and *suffixes*. A nonempty deque d over A is represented either by a suffix $sf(d)$ or by a 5-tuple that consists of a prefix $pr(d)$, a left deque of triples $ld(d)$, a middle $md(d)$, a right deque of triples $rd(d)$, and a suffix $sf(d)$. A *triple* consists of a *first middle buffer*, a deque of triples, and a *last middle buffer*. One of the two middle buffers in a triple must be nonempty, and in a triple that contains a nonempty deque both middles must be nonempty. All buffers and triples are over A. A prefix or suffix in a 5-tuple contains three to six elements, a suffix in a suffix-only representation contains

one to eight elements, a middle in a 5-tuple contains exactly two elements, and a nonempty middle buffer in a triple contains two or three elements.

The order of elements in a deque is the one consistent with the order of each 5-tuple, each buffer, each triple, and each recursive deque. The pointer structure is again straightforward, with the nodes representing 5-tuples or triples containing one pointer for each field.

4.2 Operations

We describe only the functions PUSH, POP, and CATENATE, since INJECT is symmetric to PUSH and EJECT is symmetric to POP. We begin with PUSH.

PUSH(x, d):
Case 1: Deque d is represented by a 5-tuple.
1) If $|pr(d)| = 6$ then create two new prefixes p' and p'' where p' contains the first four elements of $pr(d)$ and p'' contains the last two elements of $pr(d)$. Overwrite $pr(d)$ with p' and $ld(d)$ with the result of PUSH$((p'', \emptyset, \emptyset), ld(d))$.
2) Let $p =$ PUSH$(x, pr(d))$ and assemble the result from p, $ld(d)$, $md(d)$, $rd(d)$, and $sf(d)$.
Case 2: Deque d is represented by a suffix only.
If $|sf(d)| = 8$, then create a prefix p containing the first three elements of $sf(d)$, a middle m containing the fourth and fifth elements of $sf(d)$, and a new suffix s containing the last three elements of $sf(d)$. Overwrite $pr(d)$, $md(d)$, and $sf(d)$ with p, m, and s, respectively. Let $p' =$ PUSH(x, p) and assemble the result from p', $\emptyset$, m, $\emptyset$, and s. If $|sf(d)| < 8$, let $s' =$ PUSH$(x, sf(d))$ and represent the result by s' only.

Note that PUSH (and INJECT) creates a valid deque even when given a deque in which the prefix (or suffix, respectively) contains only two elements. Such deques may exist transiently during a POP (or EJECT), but are immediately passed to PUSH (or INJECT) and then discarded.

CATENATE(d_1, d_2):
Case 1: Both d_1 and d_2 are represented by 5-tuples.
Let y be the first element in $pr(d_2)$, and let x be the last element in $sf(d_1)$. Create a new middle m containing x followed by y. Partition the elements in $sf(d_1) - \{x\}$ into at most two buffers s_1' and s_1'' each containing two or three elements in order, with s_1'' possibly empty. Let $ld_1' =$ INJECT$((md(d_1), rd(d_1), s_1'), ld(d_1))$. If $s_1'' \neq \emptyset$ then Let $ld_1'' =$ INJECT$((s_1'', \emptyset, \emptyset), ld_1')$; otherwise, let $ld_1'' = ld_1'$. Similarly partition the elements in $pr(d_1) - \{y\}$ into at most two prefixes p_2' and p_2'' each containing two or three elements in order, with p_2' possibly empty. Let $rd_2' =$ PUSH$((p_2'', ld(d_2), md(d_2)), rd(d_2))$. If $p_2' \neq \emptyset$ let $rd_2'' =$ PUSH$((p_2', \emptyset, \emptyset), rd_2')$; otherwise, let $rd_2'' = rd_2'$. Assemble the result from $pr(d_1)$, ld_1'', m, rd_2'', and $sf(d_2)$.
Case 2: d_1 or d_2 is represented by a suffix only.
Push or inject the elements of the suffix-only deque one by one into the other deque.

In order to define the POP operation, we define a NÄIVE-POP procedure that simply pops its argument without making sure that the result is a valid deque.

NÄIVE-POP(d): If d is represented by a 5-tuple, let $(x, p) = \text{POP}(pr(d))$ and return x together with a deque assembled from p, $ld(d)$, $md(d)$, $rd(d)$, and $sf(d)$. If d consists of a suffix only, let $(x, s) = \text{POP}(sf(d))$ and return x together with a deque represented by s only.

POP(d):
If deque d is represented by a suffix only, or if $|pr(d)| > 3$, then perform $(x, d') = \text{NÄIVE-POP}(d)$ and return (x, d'). Otherwise, carry out the appropriate one of the following three cases to increase the size of $pr(d)$; then perform $(x, d') = \text{NÄIVE-POP}(d)$ and return (x, d').
Case 1: $|pr(d)| = 3$ and $ld(d) \neq \emptyset$.
Inspect the first triple t in $ld(d)$. If either the first nonempty middle buffer in t contains 3 elements or t contains a nonempty deque, then perform $(t, l) = \text{NÄIVE-POP}(ld(d))$; otherwise, perform $(t, l) = \text{POP}(ld(d))$. Let $t = (x, d', y)$ and w.l.o.g. assume that x is nonempty if t consists of only one nonempty middle buffer. Apply the appropriate one of the following two subcases.
Case 1.1: $|x| = 3$.
Pop the first element of x and inject it into $pr(d)$. Let x' be the buffer obtained from x after the pop and let p' be the buffer obtained from $pr(d)$ after the inject. Overwrite $pr(d)$ with p' and overwrite $ld(d)$ with the result of $\text{PUSH}((x', d', y), l)$.
Case 1.2: $|x| = 2$.
Inject all the elements from x into $pr(d)$ to obtain p'. Then, if d' and y are null, overwrite $pr(d)$ with p' and overwrite $ld(d)$ with l. If on the other hand, d' and y are not null, let $l' = \text{CATENATE}(d', \text{PUSH}((y, \emptyset, \emptyset), l))$, and overwrite $pr(d)$ with p' and $ld(d)$ with l'.
Case 2: $|pr(d)| = 3$, $ld(d) = \emptyset$, and $rd(d) \neq \emptyset$.
Inspect the first triple t in $rd(d)$. If either the first nonempty middle buffer in t contains 3 elements or t contains a nonempty deque, then perform $(t, r) = \text{NÄIVE-POP}(rd(d))$; otherwise, perform $(t, r) = \text{POP}(rd(d))$. Let $t = (x, d', y)$ and w.l.o.g. assume that x is nonempty if t consists of only one nonempty middle buffer. Apply the appropriate one of the following two subcases.
Case 2.1: $|x| = 3$.
Pop an element from $md(d)$ and inject it into $pr(d)$, Let m be the buffer obtained from $md(d)$ after the pop and p the buffer obtained from $pr(d)$ after the inject. Pop an element from x and inject it into m to obtain m'. Let x' be the buffer obtained from x after the pop, and let $r' = \text{PUSH}((x', d', y), r)$. Overwrite $pr(d)$, $ld(d)$, $md(d)$, and $rd(d)$ with p, $\emptyset$, m', and r' respectively.
Case 2.2: $|x| = 2$
Inject the two elements in $md(d)$ into $pr(d)$ to obtain p. Overwrite $pr(d)$, $md(d)$, and $rd(d)$ with p, x, and r', where $r' = r$ if d' and y are empty and $r' = \text{CATENATE}(d', \text{PUSH}((y, \emptyset, \emptyset), r))$ otherwise.
Case 3: $|pr(d)| = 3$, $ld(d) = \emptyset$, and $rd(d) = \emptyset$.
If $|sf(d)| = 3$, then combine $pr(d)$, $md(d)$, and $sf(d)$ into a single buffer s and overwrite the representation of d with a suffix-only representation using s. Oth-

erwise, overwrite $pr(d)$, $md(d)$, and $sf(d)$ with the results of shifting one element from the middle to the prefix, and one element from the suffix to the middle.

4.3 Analysis

We call a prefix or suffix in a 5-tuple *red* if it contains either three or six elements and *green* otherwise. We call a suffix in a suffix-only representation *red* if it contains eight elements and *green* otherwise. The prefix of a suffix-only deque is considered to have the same color as the suffix. A node representing a deque can be in one of three states: rr, if both the prefix and suffix are red, gr, if one buffer is green and the other red, or gg, if both buffers are green. We define the potential Φ of a collection of deques exactly as in the previous section: $\Phi = 3 * (\#rr) + \#gr$ where $\#rr$ and $\#gr$ are the numbers of nodes that are in states rr and gr, respectively.

The amortized costs of PUSH and INJECT are $O(1)$ by an argument identical to that given in the analysis of POP in the previous section. CATENATE calls PUSH and INJECT a constant number of times and assembles a single new node, so its amortized cost is also $O(1)$.

Finally, we analyze POP. Assume that a call to POP recurs to depth k. By an argument analogous to the one given in the analysis of POP in the previous section, each of the first $k-1$ calls to POP pays for itself by decreasing the potential by one. The last call to POP may invoke PUSH or CATENATE, and excluding this invocation has a constant amortized cost. Since the amortized cost of PUSH and CATENATE is constant, we conclude that the the amortized cost of POP is constant.

In summary we have proved the following theorem:

Theorem 2. *Our deque representation supports* PUSH, POP, INJECT, EJECT, *and* CATENATE *in* $O(1)$ *amortized time.*

4.4 Related Work

The structure presented in this section is analogous to the structures of [13, Section 8] and [7, Section 9] but simplifies them as follows. First, the buffers are of constant size, whereas in [13] and [7] they are noncatenable deques. Second, the skeleton of the present structure is a binary tree, instead of a tree extension of a redundant digital numbering system as in [7]. The amortized analysis uses the standard potential function method of [14] rather than the more complicated debit mechanism used in [13].

For catenable steques (EJECT is not allowed) we have a simpler structure that has a stack as its skeleton rather than a binary tree. It is based on the same recursive decomposition of lists as in [8, Section 4]. Our new structure simplifies the structure of [8] because we use constant size buffers rather than noncatenable stacks, and our pointer structure defines a stack rather than a stack of stacks. We will describe this structure in the full version of the paper.

5 Further Results and Open Questions

If the universe A of elements over which deques are constructed has a total order, we can extend the structures described here to support an additional heap order based on the order on A. Specifically, we can support the additional operation of finding the minimum element in a deque (but not deleting it) while preserving a constant amortized time bound for every operation, including finding the minimum. We merely have to store with each buffer, each deque, and each pair or triple the minimum element in it. For related work see [1, 2, 6, 10].

We can also support a *flip* operation on deques. A flip operation reverses the linear order of the elements in the deque: the ith from the front becomes the ith from the back, and vice-versa. For the noncatenable deques of Section 3, we implement flip by maintaining a *reversal bit* that is flipped by a flip operation. If the reversal bit is set, a push becomes an inject, a pop becomes an eject, an inject becomes a push, and an eject becomes a pop. To support catenation as well as flip we use reversal bits at all levels. We must also symmetrize the definition in Section 4 to allow a deque to be represented by a prefix only, and extend the various operations to handle this possibility. The interpretation of reversal bits is cumulative. That is, if d is a deque and x is a deque inside of d, x is regarded as being reversed if an odd number of reversal bits are set to 1 along the path of actual pointers in the structure from the node for d to the node for x. Before performing catenation, if the reversal bit of either or both of the two deques is 1, we push such bits down by flipping such a bit of a deque x to 0, flipping the bits of all the deques to which x points, and swapping the appropriate buffers and deques (the prefix and suffix exchange roles, as do the left deque and right deque). We do such push-downs of reversal bits by assembling new deques, not by overwriting the old ones.

We have devised an alternative implementation of catenable deques in which the sizes of the prefixes and suffixes are between 3 and 5 instead of 3 and 6. To achieve this we have to use two additional pointers in each node. For a node that represents a deque d, one additional pointer, if not null, points to the result of $\textsc{pop}(d)$; and the other, if not null, points to the result of $\textsc{eject}(d)$. The implementation of push and catenate is essentially as in Section 4. The changes in pop (and eject) are as follows. While popping a deque d with a prefix of size 3, if the pointer to $\textsc{pop}(d)$ is not null we read the result from there. Otherwise, we carry out a sequence of operations as in Section 4 but instead of overwriting the buffers of d before creating the result we create the result and record it in the additional pointer field of the node representing d. Using a more complicated potential function than the one used in Section 4 we can show that this implementation runs in $O(1)$ amortized time per operation.

One direction for future research is to find a way to simplify our structures further. Specifically, consider the following alternative representation of catenable deques, which uses a single recursive subdeque rather than two such subdeques. A nonempty deque d over A is represented by a triple that consists of a prefix $pr(d)$, a (possibly empty) child deque of triples $c(d)$, and a suffix $sf(d)$. A *triple* consists of a nonempty *prefix*, a deque of triples, and a nonempty *suffix*,

or just of a nonempty prefix or suffix. All buffers and triples are over A. The operations PUSH, POP, INJECT, and EJECT have implementations similar to their implementations in Section 4. The major difference is in the implementation of CATENATE, which for this structure requires a call to POP. Specifically, let d_1 and d_2 be two deques to be catenated. CATENATE pops $c(d_1)$ to obtain a triple (p, d', s) and a new deque c, injects $(s, c, sf(d_1))$ into d' to obtain d'' and then pushes $(p, d'', pr(d_2))$ onto $c(d_2)$ to obtain c'. The final result is assembled from $pr(d_1)$, c', and $sf(d_2)$. It is an open question whether this algorithm runs in constant amortized time per operation for any constant upper and lower bounds on the buffer sizes.

Another research direction is to design a confluently persistent representation of sorted lists such that accesses or updates d positions from an end take $O(\log d)$ time, and catenation takes $O(1)$ time. The best structure so far developed for this problem has a doubly logarithmic catenation time [9]; it is strictly functional, and the time bounds are worst-case.

References

1. A. L. Buchsbaum, R. Sundar, and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues. *SIAM J. Computing*, 24(6):1190–1206, 1995.
2. A. L. Buchsbaum and R. E. Tarjan. Confluently persistant deques via data structural bootstrapping. *J. of Algorithms*, 18:513–547, 1995.
3. P. F. Dietz. Fully persistent arrays. In *Proceedings of the 1989 Workshop on Algorithms and Data Structures (WADS'89)*, pages 67–74. Springer, 1995. LNCS 382.
4. J. Driscoll, D. Sleator, and R. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.
5. J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *J. of Computer and System Science*, 38:86–124, 1989.
6. Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 12(4):197–200, 1986.
7. H. Kaplan. *Purely functional lists.* PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ 08544, 1997.
8. H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slowdown. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (Preliminary Version)*, pages 93–102. ACM Press, 1995. Complete version submitted to Journal of the ACM.
9. H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 202–211. ACM Press, 1996.
10. S. R. Kosaraju. An optimal RAM implementation of catenable min double-ended queues. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 195–203, 1994.
11. C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *Proc. 36th Symposium on Foundations of Computer Science*, pages 646–654. IEEE, 1995.

12. C. Okasaki. Simple and efficient purely functional queues and deques. *J. Functional Progamming*, 5(4):583–592, 1995.
13. C. Okasaki. *Purely functional data structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1996.
14. R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.