SIMPLE CONFLUENTLY PERSISTENT CATENABLE LISTS.

HAIM KAPLAN*, CHRIS OKASAKI†, AND ROBERT E. TARJAN‡

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. Our implementation is functional if we allow memoization.

Key words. functional programming, data structures, persistent data structures, stack, queue, stack-ended-queue (steque), double-ended-queue (deque), memoization.

AMS subject classifications. 68P05, 68Q25

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 purely functional programming. We take here an extremely strict view of pure functionality: we disallow lazy evaluation, memoization, and other such techniques. For list-based data structure design, purely functional programming amounts to using only the LISP functions CONS, CAR, CDR. Purely 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 [2, 4] culminated in the work of Kaplan and Tarjan [11, 10], 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 purely functional.

*Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. haimk@math.tau.ac.il.

† Department of Computer Science, Columbia University, New York, NY 10027. Research at Carnegie Mellon University 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. cokasaki@cs.columbia.edu.

‡Department of Computer Science, Princeton University, Princeton, NJ 08544 and InterTrust Technologies Corporation, Sunnyvale, CA 94086. Research at Princeton University partially supported by NSF Grant No. CCR-9626862. ret@cs.princeton.edu.

If all one cares about is persistence, purely functional programming is unnecessarily restrictive. In particular, Okasaki [14, 15, 16] observed that the use of lazy evaluation in combination with memoization can lead to efficient functional (but not purely functional in our sense) 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 catenable deques: 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. We give two alternative, but closely related implementations of our method. The first uses memoization. The second, which saves a small constant factor in time and space, uses an extension of memoization in which any expression can replace an equivalent expression.

The remainder of the paper 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 extend our approach to handle stacks with catenation. In Section 5, we develop our solution for deques with catenation. We conclude in Section 6 with some remarks and open problems. An extended abstract of this work appeared in [9].

2. Preliminaries. The objects of our study are lists. As in [11, 10] 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 *purely 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 pure functionality by allowing memoization, in which a function is evaluated only once on a node; the second time the same function is evaluated on the same node, the value is simply retrieved from the previous computation. In all our constructions, there are only a constant number of memoized functions (one or two). We can implement memoization by having a node point to the results of applying each memoized function to it. Initially each such pointer is undefined. The first function evaluation fills in the appropriate pointer to indicate the result. Subsequent evaluations merely follow the pointer to the result, which takes $O(1)$ time.

We also consider the use of a more substantial extension of pure functionality, in which we allow 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.) The use of overwriting instead of memoization saves a small constant factor in running time and storage space and slightly simplifies the amortized analysis.

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 [17] 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 [11, 10] 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 representation for this representation is the obvious one: a node representing a deque d contains pointers to $pr(d)$, $c(d)$, and $sf(d)$. Note that the pointer structure of d is essentially a linked list of its descendants, since $c^i(d)$ contains a pointer to $c^{i+1}(d)$, for each i.

3.2. Operations. Implementing the deque operations is straightforward, except for maintaining the size bounds on buffers. Specifically, a PUSH on a deque is easy unless its prefix is of size three, a POP on a deque is easy unless its prefix is empty, and symmetric statements hold for INJECT and EJECT. We deal with buffer overflow and underflow in a proactive fashion, first fixing the buffer so that the operation to be performed cannot violate its size bounds and then actually doing the operation. The details are as follows.

We define a buffer to be *green* if it contains one or two elements, and *red* if it contains zero or three. We define two memoized functions on a deque: GP, which constructs a representation of the same list but with a green prefix; and GS, which constructs a representation of the same list with a green suffix. We only apply GP (GS, respectively) to a list whose prefix (suffix) is red and can be made green. Specifically, for GP, if the prefix is empty, the child deque must be nonempty, and symmetrically for GS. Below we give implementations of PUSH, POP, and GP; the implementations for INJECT, EJECT, and GS are symmetric. We denote a deque with prefix p, child deque c, and suffix s by $[p, c, s]$. As mentioned in Section 2, we can implement the memoization of GP and GS by having each node point to the nodes resulting from applying GP and GS to it; initially, such pointers are undefined.

PUSH(x, d): If $|pr(d)| = 3$, let $e = \text{GP}(d)$; otherwise, let $e = d$. Push x onto $pr(e)$ to form p' and return $[p', c(e), sf(e)]$.

POP(d): If $pr(d)$ is empty and $c(d)$ is not, let $e = \text{GP}(d)$; otherwise, let $e = d$. If $pr(e)$ is nonempty, let $(x, p) = \text{POP}(pr(e))$ and return the pair $(x, [p, c(e), sf(e)])$. Otherwise ($c(e)$ must be empty), let $(x, s) = \text{POP}(sf(e))$ and return the pair $(x, [\emptyset, \emptyset, s])$.

GP(d): If $|pr(d)| = 3$, let x, y, z be the three elements in $pr(d)$. Let p be a prefix containing only x, and let $c' = \text{PUSH}((y, z), c(d))$. Return $[p, c', sf(d)]$. Otherwise ($pr(d)$ is empty and $c(d)$ is not), let $((x, y), c') = \text{POP}(c(d))$ and let p' be a prefix containing x followed by y. Return $[p', c', sf(d)]$.

3.3. Analysis. The amortized analysis of this method relies on the memoization of GP and GS. We call a node representing a deque *secondary* if it is returned by a call to GP or GS and *primary* otherwise. If a secondary node y is constructed by a call GP(x) (GS(x), respectively), the only way to access y later is via another call GP(x) (GS(x), respectively): no secondary node is returned as the result of a PUSH, POP, INJECT, or EJECT operation. This means that GP and GS are called only on primary nodes.

We devide the nodes representing deques into three states: such a node is rr if both its buffers are red, gr if exactly one of its buffers is red, and gg if both its buffers are green. We subdivide the rr and gr states: an rr node is $rr0$ if neither GP nor GS has been applied to it, $rr1$ if exactly one of GP and GS has been applied to it, and $rr2$ if both GP and GS have been applied to it; a gr node is $gr0$ if neither GP nor GS has been applied to it, and $gr1$ otherwise. By the discussion above every secondary node is $gr0$ or gg. We define $\#rr0$, $\#rr1$, and $\#gr0$ to be the numbers of *primary* nodes in states $rr0$, $rr1$, and $gr0$, respectively. We define the potential of a collection of nodes representing deques to be $4\#rr0 + 2\#rr1 + \#gr0$.

A call to PUSH is either terminal or results in a call to GP, which in turn calls PUSH. Similarly, a call to POP is either terminal or results in a call to GP, which in turn calls POP. We charge the $O(1)$ time spent in a call to GP (exclusive of the inner call to PUSH or POP) to the PUSH or POP that calls GP. A call to PUSH results in a sequence of recursive calls to PUSH (via calls to GP), of which the bottommost is *terminal* and the rest are *nonterminal*. A nonterminal PUSH has one of the following effects: it converts a primary $rr0$ node to $rr1$ and creates a new primary $gr0$ node (the result of the push) and a new secondary $gr0$ node (the result of the call to GP); it converts a primary $rr1$ node to $rr2$ and creates a new primary $gr0$ node and a new secondary $gr0$ node; or, it converts a primary $gr0$ node to $gr1$ and creates a new primary gg node and a new secondary gg node. In each case the total potential drops by one, paying for the time needed for the PUSH (excluding the recursive call). A terminal PUSH takes $O(1)$ time, creates $O(1)$ new nodes, and increases the potential by $O(1)$. We conclude that PUSH takes $O(1)$ amortized time. Analogous arguments apply to POP, INJECT, and EJECT, giving us the following theorem:

THEOREM 3.1. *Each of the operations* PUSH, POP, INJECT, *and* EJECT *defined above takes* $O(1)$ *amortized time.*

3.4. Implementation Using Overwriting. With the memoized implementation described above, a primary rr node can give rise to two secondary gr nodes representing the same list; a primary gr node can give rise to a secondary gg node representing the same list. These redundant representations exist simultanously. A gr representation, however, dominates an rr representation for performing deque operations, and a gg representation dominates a gr representation. This allows us to improve the efficiency of the implementation by using overwriting in place of memoization: when GP is called on a node, it overwrites the contents of the node with the results of the GP computation, and similarly for GS. Then only one representation of a list exists at any time, and it evolves from rr to gr to gg (via one of two alternative paths, depending on whether GP or GS is called first). Each node now needs only three fields (for prefix, child deque, and suffix) instead of five (two extra for GP and GS).

Not only does the use of overwriting save a constant factor in running time and storage space, but it also simplifies the amortized analysis, as follows. We define $\#rr$ and $\#gr$ to be the number of nodes in states rr and gr, respectively. (There are now no secondary nodes.) We define the potential of a collection of nodes to be $3\#rr + \#gr$. A nonterminal PUSH has one of the following effects: it converts an rr

node to *gr* and creates a new *gr* node, or converts a *gr* node to *gg* and creates a new *gg* node. In either case it reduces the potential by one, paying for the $O(1)$ time required by the PUSH (excluding the recursive call). A terminal PUSH takes $O(1)$ time and can increase the potential by $O(1)$. We conclude that PUSH takes $O(1)$ amortized time. Similar arguments apply to POP, INJECT, and EJECT.

3.5. Related Work. The structure just described is based on the Kaplan-Tarjan structure of [10, Section 4], 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 [7] 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 [12].

4. Catenable Steques. In this section we show how to extend our ideas to support catenation. Specifically, we describe a data structure for catenable steques that achieves an $O(1)$ amortized time bound for PUSH, POP, INJECT, and CATENATE. The data structure is based on the same recursive decomposition of lists as that in Section 5 of [10]. The pointer structure that we need here is much simpler than that in [10], and the analysis is amortized, following the framework outlined in Section 2 and used in Section 3.

4.1. Representation. Our structure is similar to the structure of Section 3, but with slightly different definitions of the component parts. As in Section 3, we use buffers of two kinds: prefixes and suffixes. Each prefix contains two to six elements and each suffix contains one to three elements. A nonempty steque d over A is represented either by a suffix $sf(d)$ only, or by an ordered triple consisting of a prefix $pr(d)$ over A, a child steque $c(d)$ of pairs over A, and a suffix $sf(d)$ over A. In contrast to Section 3, a *pair over A* is defined to be an ordered pair containing a prefix and a possibly empty steque of pairs over A. Observe that this definition adds an additional kind of recursion (pairs store steques) to the structure of Section 3. This extra kind of recursion is what allows efficient catenation.

The order of elements in a steque is the one consistent with the order of each triple, each buffer, each pair, each steque within a pair, and each child steque. As in Section 3, there can be many different representations of a steque containing a given list of elements; when speaking of a steque, we mean a particular representation of it.

The pointer structure for this representation is straightforward. Each triple is represented by a node containing three pointers: to a prefix, a child steque, and a suffix. Each pair is represented by a node containing two pointers: to a prefix and a steque.

4.2. Operations. The implementation of the steque operations is much like the implementation of the noncatenable deque operations presented in Section 3.2. We call a prefix *red* if it contains either two or six elements, and *green* otherwise. We call a suffix *red* if it contains three elements and *green* otherwise. The prefix in a suffix-only steque is considered to have the same color as the suffix. We define two memoized functions, GP, and GS, which produce green-prefix and green-suffix representations of a steque, respectively. Each is called only when the corresponding buffer is red and can be made green. We define PUSH, POP, and INJECT to call GP or GS when necessary to obtain a green buffer. In the definitions below, we represent a steque with prefix p, child steque c, and suffix s by $[p, c, s]$.

PUSH(x, d):
Case 1: Steque d is represented by a triple. If $|pr(d)| = 6$ then let $e = \text{GP}(d)$; otherwise, let $e = d$. Let $p = \text{PUSH}(x, pr(e))$ and return $[p, c(e), sf(e)]$.
Case 2: Steque d is represented by a suffix only. If $|sf(d)| = 3$, create a prefix p containing x and the first two elements of $sf(d)$, create a suffix s containing the last element of $sf(d)$, and return $[p, \emptyset, s]$. Otherwise, create a suffix s by pushing x onto $sf(d)$ and return $[\emptyset, \emptyset, s]$.

INJECT(d, x):
Case 1: Steque d is represented by a triple. If $|sf(d)| = 3$, let $e = \text{GS}(d)$; otherwise, let $e = d$. Let $s = \text{INJECT}(sf(e), x)$ and return $[pr(e), c(e), s]$.
Case 2: Steque d is represented by a suffix only. If $|sf(d)| = 3$, create a suffix s containing x, and return $[sf(d), \emptyset, s]$. Otherwise, create a suffix s by injecting x into $sf(d)$ and return $[\emptyset, \emptyset, s]$.

CATENATE(d_1, d_2):
Case 1: d_1 and d_2 are represented by triples. First, catenate the buffers $sf(d_1)$ and $pr(d_2)$ to obtain p. Now, calculate c' as follows: If $|p| \leq 5$ then let $c' = \text{INJECT}(c(d_1), (p, c(d_2)))$. Otherwise, $6 \leq |p| \leq 9$. Create two new prefixes p' and p'', with p' containing the first four elements of p and p'' containing the remaining elements. Let $c' = \text{INJECT}(\text{INJECT}(c(d_1), (p', \emptyset)), (p'', c(d_2)))$. In either case, return $[pr(d_1), c', sf(d_2)]$.
Case 2: d_1 or d_2 is represented by a suffix only. Push or inject the elements of the suffix-only steque one-by-one into the other steque.

Note that both PUSH and CATENATE produce valid steques even when their second arguments are steques with prefixes of length one. Although such steques are not normally allowed, they may exist transiently during a POP. Every such steque is immediately passed to PUSH or CATENATE, and then discarded, however. In order to define the POP, GP, and GS operations, we define a NÄIVE-POP operation that simply pops its steque argument without making sure that the result is a valid steque.

NÄIVE-POP(d): If d is represented by a triple, let $(x, p) = \text{POP}(pr(d))$ and return the pair $(x, [p, c(d), sf(d)])$. If d consists of a suffix only, let $(x, s) = \text{POP}(sf(d))$ and return the pair $(x, \emptyset)$ if $|d| = 1$ or $(x, [\emptyset, \emptyset, s])$ if $|d| > 1$.

POP(d):
Case 1: Steque d is represented by a suffix only or $|pr(d)| > 2$. Return NÄIVE-POP(d).
Case 2: Steque d is represented by a triple, $|pr(d)| = 2$, and $c(d) = \emptyset$. Let x be the first element on pr(d) and y the second. If $|sf(d)| < 3$, push y onto $sf(d)$ to form s and

return $(x, [\emptyset, \emptyset, s])$. Otherwise ($|sf(d)| = 3$), form p from y and the first two elements on $sf(d)$, form s from the last element on $sf(d)$, and return $(x, [p, \emptyset, s])$.
Case 3: Steque d is represented by a triple, $|pr(d)| = 2$, and $c(d) \neq \emptyset$. Return NÄIVE-POP(GP(d)).

GP(d): If $|pr(d)| = 6$, then create two new prefixes p and p' by splitting $pr(d)$ equally in two. Let $c' =$ PUSH($(p', \emptyset), c(d)$). Return $[p, c', sf(d)]$. Otherwise ($|pr(d)| = 2$ and $c(d) \neq \emptyset$), proceed as follows. Inspect the first pair (p, d') in $c(d)$. If $|p| \geq 4$ or d' is not empty, let $((p, d'), c') =$ NÄIVE-POP($c(d)$); otherwise, let $((p, d'), c') =$ POP($c(d)$). Now inspect p.
Case 1: p contains at least four elements. Pop the first two elements from p to form p'' and inject these two elements into $pr(d)$ to obtain p'. Let $c'' =$ PUSH($(p'', d'), c'$). Return $[p', c'', sf(d)]$.
Case 2: p contains at most three elements. Push the two elements in $pr(d)$ onto p to obtain p'. Let $c'' =$ CATENATE(d', c') if d' is nonempty, or $c'' = c'$ if d' is empty. Return $[p', c'', sf(d)]$.

GS(d): (Steque d is represented by a triple with $|sf(d)| = 3$) Let p contain the first two elements of $sf(d)$ and s the last element on $sf(d)$. Let $c' =$ INJECT($c(d), (p, \emptyset)$). Return $[pr(d), c', s]$.

4.3. Analysis. The analysis of this method is similar to the analysis in Section 3.3. We define primary and secondary nodes, node states, and the potential function exactly as in Section 3.3: the potential function, as there, is $4\#rr0 + 2\#rr1 + \#gr0$, where $\#rr0$, $\#rr1$, and $\#gr0$ are the numbers of primary nodes in states $rr0$, $rr1$, and $gr0$, respectively.

As in Section 3.3, we charge the $O(1)$ cost of a call to GP or GS (excluding the cost of any recursive call to PUSH, POP, or INJECT) to the PUSH, POP, or INJECT that calls GP or GS. The amortized costs of PUSH and INJECT are $O(1)$ by an argument identical to that used to analyze PUSH in Section 3.3. Operation CATENATE calls PUSH and INJECT a constant number of times and creates a single new node, so its amortized cost is also $O(1)$.

To analyze POP, assume that a call to POP recurs to depth k (via intervening calls to GP). By an argument analogous to that for PUSH, each of the first $k - 1$ calls pays for itself by decreasing the potential by one. The terminal call to POP can result in a call to either PUSH or CATENATE, each of which has $O(1)$ amortized cost. It follows that the overall amortized cost of POP is $O(1)$, giving us the following theorem:

THEOREM 4.1. *Each of the operations* PUSH, POP, INJECT, *and* CATENATE *defined above takes $O(1)$ amortized time.*

We can improve the time and space efficiency of the steque data structure by constant factors by using overwriting in place of memoization, exactly as described in Section 3.4. If we do this, we can also simplify the amortized analysis, again exactly as described in Section 3.4.

4.4. Related work. The structure presented in this section is analogous to the Kaplan-Tarjan structure of [10, Section 5] and the structure of [8, Section 7], but simplifies them as follows. First, the buffers are of constant-bounded size, whereas the structure of [10, Section 5] uses noncatenable steques as buffers, and the structure of [8, Section 7] uses noncatenable stacks as buffers. These buffers in turn must be represented as in Section 3 of this paper or by using one of the other methods mentioned there. In contrast, the structure of the present section is entirely self-contained. Second, the skeleton of the present structure is just a stack, instead of a stack of stacks as in [10] and [8]. Third, the changes used to make buffers green are applied in a one-sided, need-driven way; in [10] and [8], repairs must be made simultaneously to both sides of the structure in carefully chosen locations.

Okasaki [14] has devised a different and somewhat simpler implementation of confluently persistent catenable steques that also achieves an $O(1)$ amortized bound per operation. His solution obtains its efficiency by (implicitly) using a form of path reversal [18] in addition to lazy evaluation and memoization. Our structure extends to the double-ended case, as we shall see in the next section; whether Okasaki's structure extends to this case is an open problem.

5. Catenable Deques. In this section we show how to extend our ideas to support all five list operations. 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 [16], but simplified to use constant-size buffers.

5.1. 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.

5.2. Operations. 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. We introduce two memoizing functions functions GP and GS as in Sections 3.2 and 4.2, which produce green-prefix and green-suffix representations of a deque, respectively, and which are called only when the corresponding buffer is red but can be made green. Below we give the implementations of PUSH, POP, GP, and CATENATE; the implementations of INJECT, EJECT, and GS are symmetric to those of PUSH, POP, and GP, respectively. We denote a deque with prefix p, left deque l, middle m, right deque r, and suffix s by $[p, l, m, r, s]$.

PUSH(x, d):
Case 1: Deque d is represented by a 5-tuple. If $|pr(d)| = 6$, then let $e = \text{GP}(d)$; otherwise, let $e = d$. Return $[\text{PUSH}(x, pr(e)), ld(e), md(e), rd(e), sf(e)]$.
Case 2: Deque d is represented by a suffix only. If $sf(d) < 8$, return a suffix-only deque with suffix $\text{PUSH}(x, sf(d))$. Otherwise, push x onto $sf(d)$ to form s, with nine elements. Create a new prefix p with the first four, a middle with the next two, and a suffix s with the last three. Return $[p, \emptyset, m, \emptyset, s]$.

As in Section 4.2, the implementation of POP uses NÄIVE-POP.

POP(d):
Case 1: Deque d is represented by a suffix only or $|pr(d)| > 3$. Return NÄIVE-POP(d).

Case 2: $|pr(d)| = 3$ and $ld(d) \cup rd(d) \neq \emptyset$. Return NÄIVE-POP(GP($d$)).
Case 3: $|pr(d)| = 3$ and $ld(d) \cup rd(d) = \emptyset$. Let x be the first element on $pr(d)$. If $|sf(d)| = 3$, create a new suffix s containing all the elements in $pr(d)$, $md(d)$, and $sf(d)$ except x, and return the pair consisting of x and the deque represented by s only. Otherwise, form p from $pr(d)$ by popping x and injecting the first element on $md(d)$, form m' from $md(d)$ by popping the first element and injecting the first element on $sf(d)$, form s from $sf(d)$ by popping the first element, and return $(x, [p, \emptyset, m', \emptyset, s])$.

GP(d): If $|pr(d)| = 6$ create two new prefixes p and p', with p containing the first four elements of $|pr(d)|$ and p' the last two; return $[p, \text{PUSH}((p', \emptyset, \emptyset), ld(d)), md(d), rd(d), sf(d)]$. Otherwise ($|pr(d)| = 3$ and $ld(d) \cup rd(d) \neq \emptyset$), proceed as follows.
Case 1: $ld(d) \neq \emptyset$. Inspect the first triple t on $ld(d)$. If either the first nonempty middle buffer in t contains 3 elements or t contains a nonempty deque, let $(t, l) =$ NÄIVE-POP($ld(d)$); otherwise let $(t, l) =$ POP($ld(d)$). Let $t = (x, d', y)$ and assume that x is nonempty if t consists of only one nonempty middle buffer. Apply the appropriate one of the folowing two subcases.

Case 1.1: $|x| = 3$. Form x' from x and p from $pr(d)$ by popping the first element from x and injecting it into $pr(d)$. Return $[p, \text{PUSH}((x', d', y), l), md(d), rd(d), sf(d)]$.
Case 1.2: $|x| = 2$. Inject both elements in x into pr(d) to form p. If d' and y are empty, return $[p, l, md(d), rd(d), sf(d)]$. Otherwise ($d'$ and y are nonempty) let $l' = \text{CATENATE}(d', \text{PUSH}((y, \emptyset, \emptyset), l))$ and return $[p, l', md(d), rd(d), sf(d)]$.

Case 2: $ld(d) = \emptyset$ 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, let $(t, r) =$ NÄIVE-POP($rd(d)$); otherwise, let $(t, r) =$ POP($rd(d)$). Let $t = (x, d', y)$ and 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: $|x| = 3$. Form p, m', and x' from $pr(d)$, m, and x by popping an element from m and injecting it into $pr(d)$ to form p, popping an element from m and injecting the first element from x to form m', and popping the first element from x to form x'. Return $[p, \emptyset, m', \text{PUSH}((x', d', y), r), sf(d)]$.
Case 2.2: $|x| = 2$. Inject the two elements in $md(d)$ into $pr(d)$ to form p. Let $r' = r$ if d' and y are empty or $r' = \text{CATENATE}(d', \text{PUSH}((y, \emptyset, \emptyset), r))$ otherwise. Return $[p, \emptyset, x, r', sf(d)]$.

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 $ld_1' = \text{INJECT}(ld(d_1), (md(d_1), rd(d_1), s_1'))$. If $s_1'' \neq \emptyset$ then let $ld_1'' = \text{INJECT}(ld_1', (s_1'', \emptyset, \emptyset))$; 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 $rd_2' = \text{PUSH}((p_2'', ld(d_2), md(d_2)), rd(d_2))$. If $p_2' \neq \emptyset$ let $rd_2'' = \text{PUSH}((p_2', \emptyset, \emptyset), rd_2')$; otherwise, let $rd_2'' = rd_2'$. Return $[pr(d_1), ld_1'', m, rd_2'', sf(d_2)]$.
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.

5.3. Analysis. To analyze this structure, we use the same definitions and the same potential function as in Sections 3.3 and 4.3. The amortized costs of PUSH, INJECT, CATENATE, and POP are $O(1)$ by an argument analogous to that in Section

4.3. The amortized cost of EJECT is $O(1)$ by an argument symmetric to that for POP. Thus we obtain the following theorem:

THEOREM 5.1. *Each of the operations* PUSH, POP, INJECT, EJECT, *and* CATENATE *defined above takes* $O(1)$ *amortized time.*

Just as in Sections 3.4 and 4.3, we can improve the time and space constant factors and simplify the analysis by using overwriting in place of memoization. Overwriting is the preferred implementation, unless one is using a functional programming language that supports memoization but does not easily allow overwriting.

5.4. Related Work. The structure presented in this section is analogous to the structures of [16, Chapter 11] and [8, Section 9] but simplifies them as follows. First, the buffers are of constant size, whereas in [16] and [8] 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 [8]. Also, our amortized analysis uses the standard potential function method of [17] rather than the more complicated debit mechanism used in [16]. Another related structure is that of [10, Section 5], which represents purely functional, real-time deques as pairs of triples rather than 5-tuples, but otherwise is similar to (but simpler than) the structure of [8, Section 9]. It is straightforward to modify the structure presented here to use pairs of triples rather than 5-tuples.

6. 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, 13].

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 5 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; the order of elements in the prefix and suffix is reversed as well.) 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. We do this by memoizing the POP and EJECT operations and avoiding creating a new structure with a green prefix (suffix, respectively) representing the original deque when performing POP (EJECT, respectively). Using a more complicated potential function than the ones used in earlier sections, we can show that such an 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 5. 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 has prefix $pr(d_1)$, child deque c', and suffix $sf(d_2)$. 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 either 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 [12]; it is purely functional, and the time bounds are worst-case.

Acknowledgment. We thank Michael Goldwasser for a detailed reading of this paper, and Jason Hartline for discussions that led to our implementations using memoization.

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 (1995), pp. 1190–1206.

[2] A. L. Buchsbaum and R. E. Tarjan, *Confluently persistant deques via data structural bootstrapping*, J. of Algorithms, 18 (1995), pp. 513–547.

[3] P. F. Dietz, *Fully persistent arrays*, in Proceedings of the 1989 Workshop on Algorithms and Data Structures (WADS'89), Springer, 1995, pp. 67–74. LNCS 382.

[4] J. Driscoll, D. Sleator, and R. Tarjan, *Fully persistent lists with catenation*, Journal of the ACM, 41 (1994), pp. 943–959.

[5] J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, *Making data structures persistent*, J. of Computer and System Science, 38 (1989), pp. 86–124.

[6] H. Gajewska and R. E. Tarjan, *Deques with heap order*, Information Processing Letters, 12 (1986), pp. 197–200.

[7] R. Hood, *The efficient implementation of very-high-level programming language constructs*, PhD thesis, TR 82-503, Dept. of Computer Science, Cornell University, Ithaca, NY, 1982.

[8] H. Kaplan, *Purely functional lists*, PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ 08544, 1997.

[9] H. Kaplan, C. Okasaki, and R. E. Tarjan, *Simple confluently persistent catenable lists (extended abstract)*, in Proc. Scandinavian Workshop on Algorithm Theory (SWAT), Springer-Verlag, 1998, pp. 119–130.

[10] H. Kaplan and R. E. Tarjan, *Purely functional, real-time deques with catenation*, J. Assoc. Comput. Mach. to appear.

[11] ———, *Persistent lists with catenation via recursive slow-down*, in Proceedings of the 27th Annual ACM Symposium on Theory of Computing (Preliminary Version), ACM Press, 1995, pp. 93–102.

[12] ———, *Purely functional representations of catenable sorted lists*, in Proceedings of the 28th Annual ACM Symposium on Theory of Computing, ACM Press, 1996, pp. 202–211.

[13] S. R. Kosaraju, *An optimal RAM implementation of catenable min double-ended queues*, in Proc. 5th ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 195–203.

[14] C. Okasaki, *Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking*, in Proc. 36th Symposium on Foundations of Computer Science, IEEE, 1995, pp. 646–654.

[15] ———, *Simple and efficient purely functional queues and deques*, J. Functional Progamming, 5 (1995), pp. 583–592.

[16] ———, *Purely functional data structures*, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1996.

[17] R. E. Tarjan, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 306–318.

[18] R. E. Tarjan and J. V. Leeuwen, *Worst case analysis of set union algorithms*, Journal of the ACM, 31 (1984), pp. 245–281.