Purely Functional, Real-Time Deques with Catenation

Haim Kaplan* Robert E. Tarjan†

November 12, 1999

Abstract

We describe an efficient, purely functional implementation of deques with catenation. In addition to being an intriguing problem in its own right, finding a purely functional implementation of catenable deques is required to add certain sophisticated programming constructs to functional programming languages. Our solution has a worst-case running time of $O(1)$ for each push, pop, inject, eject and catenation. The best previously known solution has an $O(\log^* k)$ time bound for the k^{th} deque operation. Our solution is not only faster but simpler. A key idea used in our result is an algorithmic technique related to the redundant digital representations used to avoid carry propagation in binary counting.

1 Introduction

A *persistent* data structure is one in which a change to the structure can be made without destroying the old version, so that all versions of the structure persist and can at least be accessed (the structure is said to be *partially persistent*) or even modified (the structure is said to be *fully persistent*). In the functional programming literature, fully persistent structures are often called *immutable*. Purely

*AT&T Laboratories, Florham Park, NJ. Some work done at Princeton University, supported by the Office of Naval Research, Contract No. N00014-91-J-1463, the NSF, Grants No. CCR-8920505 and CCR-9626862, and a United States-Israel Educational Foundation (USIEF) Fulbright Grant. hkl@research.att.com.

†Department of Computer Science, Princeton University, Princeton, NJ 08544 USA and InterTrust Technologies, Sunnyvale, CA. Research at Princeton University partially supported by the NSF, Grants No. CCR-8920505 and CCR-9626862, and the Office of Naval Research, Contract No. N00014-91-J-1463. ret@cs.princeton.edu.

functional[1] programming, without side effects, has the property that every structure created is automatically fully-persistent. Persistent data structures arise not only in functional programming but also in text, program, and file editing and maintenance; computational geometry; and other algorithmic application areas. (See [6, 10, 11, 12, 13, 14, 15, 16, 24, 37, 38, 39, 40, 41].)

A number of papers have discussed ways of making specific data structures, such as search trees, persistent. A smaller number have proposed methods for adding persistence to general data structures without incurring the huge time and space costs of the obvious method, which is to copy the entire structure whenever a change is made. In particular, Driscoll, Sarnak, Sleator, and Tarjan [14] described how to make pointer-based structures persistent using a technique called *node-splitting*, which is related to fractional cascading [7] in a way that is not yet fully understood. Dietz [11] described a method for making array-based structures persistent. Additional references on persistence can be found in the Driscoll et al. and Dietz papers.

These general techniques fail to work on data structures that can be combined with each other rather than just be changed locally. Driscoll, Sleator and Tarjan [13] coined the term "confluently persistent" to refer to a persistent structure in which some update operations can combine two different versions. Perhaps the simplest and probably the most important example of combining data structures is catenation (appending) of lists. Confluently persistent lists with catenation are surprisingly powerful. For example, by using self-catenation one can build a list of exponential size in linear time.

This paper deals with the problem of making persistent list catenation efficient. We consider the following operations for manipulating lists:

makelist(x): return a new list consisting of the singleton element x.

push(x, L): return the list that is formed by adding element x to the front of list L.

pop(L): return the pair consisting of the first element of list L and the list consisting of the second through last elements of L.

[1] For the purposes of this paper, a "purely functional" data structure is one built using only the LISP functions *car, cons, cdr*. Though we do not state our constructions explicitly in terms of these functions, it is routine to verify that our structures are purely functional. Our definition of purely functional is extremely strict; we do not, for example, allow techniques such as memoization. This contrasts our work with, for example, that of Okasaki [33, 34, 35, 36]. For more discussion of this issue, see Sections 2 and 7.

inject(x, L): return the list that is formed by adding element x to the back of list L.

eject(L): return the pair consisting of the last element on list L and the list consisting of the first through next-to-last elements of L.

catenate(K, L): return the list formed by catenating K and L, with K first.

Observe that *push* and *inject* are special cases of *catenate*. It will be convenient for us to treat them as separate operations, however. In accordance with convention, we call a list subject only to *push* and *pop* (or *inject* and *eject*) a *stack* and a list subject only to *inject* and *pop* (or *push* and *eject*) a *queue*. Adopting the terminology of Knuth [29], we call a list subject to all four operations *push*, *pop*, *inject*, and *eject* a *double-ended queue*, abbreviated *deque* (pronounced "deck"). In a departure from existing terminology, we call a list subject only to *push, pop*, and *inject* a *stack-ended queue*, or *steque* (pronounced "steck"). Knuth called steques *output-restricted deques*, but "stack-ended queue" is both easy to shorten and evokes the idea that a steque combines the functionalities of a stack and a queue. Steques with catenation are the same as stacks with catenation, since catenation makes *inject* (and *push*, for that matter) redundant. We call a data structure with constant worst-case time bounds for all operations a *real-time* structure.

Our main result is a real-time, purely functional (and hence confluently persistent) implementation of deques with catenation. Our data structure is both more efficient and simpler than previously proposed structures [4, 13]. In addition to being an interesting problem in its own right, our data structure provides a way to add fast catenation to list-based programming languages such as scheme, and to implement sophisticated programming constructs based on continuations in functional programming languages. See [15, 16]. A key ingredient in our result is an algorithmic technique related to the redundant digital representations devised to avoid carry propagation in binary counting.

The remainder of this paper consists of six sections. Section 2 surveys previous work dealing with problems related to that of making lists persistent and adding catenation as an efficient list operation. Section 3 motivates our approach. Section 4 describes how to make deques without catenation purely functional, thereby illustrating our ideas in a simple setting. Section 5 describes how to make stacks (or steques) with catenation purely functional, illustrating the additional ideas needed to handle catenation in the comparatively simple setting of stacks. Section 6 presents our most general result, an implementation of deques with catenation. This result uses an additional idea needed to handle an underlying tree-like recursive structure in place of a linear structure. Section 7 mentions additional related results and open problems.

A preliminary version of part of our work was presented at the 27^{th} Annual ACM Symposium on Theory of Computing [27].

2 Previous Work

Work related to ours can be found in three branches of computer science: data structures; functional programming; and, perhaps surprisingly, Turing machine complexity. We shall describe this work approximately in chronological order and in some detail, in an attempt to sort out a somewhat tangled history.

Let us put aside catenation for the moment and consider the problem of making noncatenable lists fully persistent. It is easy to make stacks persistent: we represent a stack by a pointer to a singly-linked list of its elements, the top element on the stack being the first element on the list. To push an element onto a stack, we create a new node containing the new element and a pointer to the node containing the previously first element on the stack. To pop a stack, we retrieve the first element and a pointer to the node containing the previously second element. This is just the standard LISP representation of a list.

A collection of persistent stacks represented in this way consists of a collection of trees, with a pointer from each child to its parent. Two stacks with common suffixes can share one list representing the common suffix. (Having common suffixes does not guarantee this sharing, however, since two stacks identical in content can be built by two separate sequences of *push* and *pop* operations. Maximum sharing of suffixes can be achieved by using a "hashed consing" technique in which a new node is created only if it corresponds to a distinct new stack. See [1, 42].)

Making a queue, steque, or deque persistent is not so simple. One approach, which has the advantage of giving a purely functional solution, is to represent such a data structure by a fixed number of stacks so that each operation becomes a fixed number of stack operations. That is, we seek a real-time simulation of a queue, steque, or deque by a fixed number of stacks. The problem of giving a real-time simulation of a deque by a fixed number of stacks is closely related to an old problem in Turing machine complexity, that of giving a real-time simulation of a (one-dimensional) multihead tape unit by a fixed number of (one-dimensional) one-head tape units. The two problems can be reduced to one another by noting that a deque can be simulated by a two-head tape unit, and a one-head tape unit can be simulated by two stacks; thus the deque problem can be reduced to the tape problem. Conversely, a k-head tape unit can be simulated by $k-1$ deques and two stacks, and a stack can be simulated by a one-head tape; thus the tape problem can be reduced to the deque problem. There are two gaps in these reductions. The first is that a deque element can potentially be chosen from an infinite universe, whereas the universe of tape symbols is always finite. This allows the possibility of solving the tape problem using some clever symbol encoding that might not be applicable to the deque problem. But none of the known solutions to the tape problem exploits this possibility; they all give solutions to the deque problem by the reduction above. The second gap is that the reductions do not necessarily minimize the numbers of stacks or one-head tapes in the simulation; if this is the goal, the deque or tape problem must be addressed directly.

The first step toward solving the tape simulation problem was taken by Stoss [43], who produced a linear-time simulation of a multihead tape by a fixed number of one-head tapes. Shortly thereafter, Fisher, Meyer, and Rosenberg [17] gave a real-time simulation of a multihead tape by a fixed number of one-head tapes. The latter simulation uses a tape-folding technique not directly related to the method of Stoss. Later, Leong and Seiferas [32] gave a real-time, multihead-tape simulation using fewer tapes by cleverly augmenting Stoss's idea. Their approach also works for multidimensional tapes, which is apparently not true of the tape-folding idea.

Because of the reduction described above, the deque simulation problem had already been solved (by two different methods!) by the time work on the problem began appearing in the data structure and functional programming literature. Nevertheless, the latter work is important because it deals with the deque simulation problem directly, which leads to a more efficient and conceptually simpler solution. Although there are several works [5, 8, 19, 20, 21, 22, 23, 34, 39] dealing with the deque simulation problem, they all describe essentially the same solution. This solution is based on two key ideas, which mimic the ideas of Stoss and Leong and Seiferas.

The first idea is that a deque can be represented by a pair of stacks, one representing the front part of the deque and the other representing the rear part. When one stack becomes empty because of too many *pop* or *eject* operations, the deque, now all on one stack, is copied into two stacks each containing half of the deque elements. This fifty-fifty split guarantees that such copying, even though expensive, happens infrequently. A simple amortization argument using a potential function equal to the absolute value of the difference in stack sizes shows that this gives a linear-time simulation of a deque by a constant number of stacks: k deque operations starting from an empty deque are simulated by $O(k)$ stack operations. (See [44] for a discussion of amortization and potential functions.) This simple idea is the essence of Stoss's tape simulation. The idea of representing a queue by two stacks in this way appears in [5, 20, 22]; this representation of a deque appears in [19, 21, 23, 39].

The second idea is to use incremental copying to convert this linear-time simulation into a real-time simulation: as soon as the two stacks become sufficiently unbalanced, recopying to create two balanced stacks begins. Because the recopying must proceed concurrently with deque operations, which among other things causes the size of the deque to be a moving target, the details of this simulation are a little complicated. Hood and Melville [22] first spelled out the details of this method for the case of a queue; Hood's thesis [21] describes the simulation for a deque. See also [19, 39]. Chuang and Goldberg [8] give a particularly nice description of the deque simulation. Okasaki [34] gives a variation of this simulation that uses "memoization" to avoid some of the explicit stack-to-stack copying; his solution gives persistence but is not strictly functional since memoization is a side effect.

A completely different way to make a deque persistent is to apply the general mechanism of Driscoll, et al. [14], but this solution, too, is not strictly functional, and the constant time bound per deque operation is amortized, not worst-case.

Once catenation is added as an operation, the problem of making stacks or deques persistent becomes much harder; all the methods mentioned above fail. Kosaraju has obtained a couple of intriguing results that deserve mention, although they do not solve the problem we consider here. First [30], he gave a real-time simulation of catenable deques by non-catenable deques. Unfortunately, this solution does not support confluent persistence; in particular, Kosaraju explicitly disallows self-catenation. His solution is also real-time only for a fixed number of deques; the time per deque operation increases at least linearly with the number of deques. Second [31], he gave a real-time, random-access implementation of catenable deques with the "find minimum" operation, a problem discussed in Section 7. This solution is real-time for a variable number of deques, but it does not support confluent persistence. Indeed, Kosaraju [31] states, "These ideas might be helpful in making mindeques confluently persistent."

There are, however, some previous solutions to the problem of making catenable deques fully persistent. A straightforward use of balanced trees gives a representation of persistent catenable deques in which an operation on a deque or deques of total size n takes $O(\log n)$ time. Driscoll, Sleator, and Tarjan [13] combined a tree representation with several additional ideas to obtain an implementation of persistent catenable stacks in which the k^{th} operation takes $O(\log\log k)$ time. Buchsbaum and Tarjan [4] used a recursive decomposition of trees to obtain two implementations of persistent catenable deques. The first has a time bound of $2^{O(\log^* k)}$ and the second a time bound of $O(\log^* k)$ for the k^{th} operation, where $\log^* k$ is the iterated logarithm, defined by $\log^{(1)} k = \log_2 k, \log^{(i)} k = \log\log^{(i-1)} k$ for $i > 1$, and $\log^* k = \min\{i \mid \log^{(i)} k \leq 1\}$. This work motivated ours.

3 Recursive Slow-down

In this section we describe the key insight that led to our result. Although this insight is not explicit in our ultimate construction and is not needed to understand it, the idea may be helpful in making progress on other problems, and for that reason we offer it here.

The spark for our work was an observation concerning the recurrence that gives the time bounds for the Buchsbaum-Tarjan data structures. This recurrence has the following form:

$$T(n) = O(1) + c\,T(\log n)$$

where c is a constant. An operation on a structure of size n takes a constant amount of time plus a fixed number of operations on recursive substructures of size $\log n$. In the first version of the Buchsbaum-Tarjan structure, c is a fixed constant greater than one, and the recurrence gives the time bound $T(n) = 2^{O(\log^* n)}$. In the second version of the structure, c equals one, and the recurrence gives the time bound $T(n) = O(\log^* n)$.

But suppose that we could design a structure in which the constant c were less than one. Then the recurrence would give the bound $T(n) = O(1)$. Indeed, the recurrence $T(n) = O(1) + cT(n-1)$ gives the bound $T(n) = O(1)$ for any constant $c < 1$, such as $c = 1/2$. (Frederickson [18] used a similar observation to improve the time bound for selection in a min-heap from $O(k2^{\log^* k})$ to $O(k)$.) Thus we can obtain an $O(1)$ time bound for operations on a data structure if each operation requires $O(1)$ time plus half an operation on a smaller recursive substructure. We can achieve the same effect if our data structure requires only *one* operation on a recursive substructure for every *two* operations on the top-level structure. We call this idea *recursive slow-down.*

The main new feature in our data structure is the mechanism for implementing recursive slow-down. Stated abstractly, the basic problem is to allocate work cycles to the levels of a linear recursion so that the top level gets half the cycles, the second level gets one quarter of the cycles, the third level gets one eighth of the cycles, and so on. This is exactly what happens in *binary counting.* Specifically, if we begin with zero and repeatedly add one in binary, each addition of one causes a unique bit position to change from zero to one. In every second addition this position is the one's bit, in every fourth addition it is the two's bit, in every eighth addition it is the four's bit, and so on.

Of course, in binary counting, each addition of one can change many bits to zero. To obtain real-time performance, this additional work must be avoided. One can do this by using a *redundant digital representation*, in which numbers have more than one representation and a single digit change is all that is needed to add one. Clancy and Knuth [9] used this idea in an implementation of finger search trees. Descriptions of such redundant representations as well as other applications can be found in [2, 9, 28]. The Clancy-Knuth method represents numbers in base two but using *three* digits, 0,1, and 2. A *redundant binary representation* (RBR) of a non-negative number x is a sequence of digits d_n, d_{n-1}, ..., d_0 with $d_i \in \{0, 1, 2\}$ and $x = \sum_{i=0}^{n} d_i 2^i$. Such a representation is in general not unique. We call an RBR *regular* if for every j such that $d_j = 2$ there exists an $i < j$ such that $d_i = 0$ and $d_k = 1$ for $i < k < j$. In other words, while scanning the digits from most significant to least significant, after finding a 2 we must find a 0 before finding another 2 or running out of digits. This implies in particular that $d_0 \neq 2$.

To add 1 to a number x represented by a regular RBR, we first add 1 to d_0. The result is an RBR for $x+1$, but which may not be regular. We restore regularity by finding the least significant digit d_i which is not 1, and if $d_i = 2$ setting $d_i = 0$ and $d_{i+1} = d_{i+1} + 1$. (If $d_i = 0$ we do nothing: the RBR is already regular.)

It is straightforward to show that this method correctly adds 1, and it does so while changing only a constant number of digits, thus avoiding explicit carry propagation.

Our work allocation mechanism for lists uses a three-state system, corresponding to the three digits $(0, 1, 2)$ of the Clancy-Knuth number representation. Instead of digits, we use colors. Each level of the recursive data structure is green, yellow, or red, with the color based on the state of the structure at that level. A red structure is bad but can be converted to a green structure at the cost of degrading the structure one level deeper, from green to yellow or from yellow to red. We maintain the invariant on the levels that any two red levels are separated by at least one green level, ignoring intervening yellow levels. The green-yellow-red mechanism applied to an underlying linear structure suffices to add constant-time catenation to stacks. To handle deques, we must extend the mechanism to apply to an underlying tree structure. This involves adding another color, orange. Whereas the green-yellow-red system is a very close analogue of the Clancy-Knuth number representation, the extended system is more distantly related. We postpone a discussion of this extension to Section 6, where it is used.

4 Deques without Catenation

In this section we present a real-time, purely functional implementation of deques without catenation. This example illustrates our ideas in a simple setting, and provides an alternative to the implementation based on a pair of incrementally copied stacks, which was described in Section 2. In Section 5 we modify the structure to support stacks with catenation. (We add *catenate* as an operation but remove *eject*.) Finally, in Section 6 we modify the structure to support all the catenable deque operations. This last step involves extending the work allocation mechanism as mentioned at the end of Section 3. Recall that the operations possible on a deque d are $push(x, d)$, $pop(d)$, $inject(x, d)$, and $eject(d)$. Here and in subsequent sections we say that a data structure is *over* a set A if it stores elements from A.

4.1 Representation

We represent a deque by a recursive structure that is built from bounded-size deques called *buffers*. Each buffer can hold up to five elements. Buffers are of two kinds: *prefixes* and *suffixes*. A non-empty deque d over a set A is represented by an ordered triple consisting of a prefix *prefix(d)* of elements of A, a *child deque* $\mathit{child}(d)$ whose elements are ordered pairs of elements of A, and a suffix $\mathit{suffix}(d)$ of elements of A. The order of elements within d is the one consistent with the orders of all of its component parts. The child deque $\mathit{child}(d)$, if non-empty, is represented in the same way. Thus the structure is recursive and unwinds linearly. We define the descendants $\{\mathit{child}^i(d)\}$ of deque d in the standard way, namely $\mathit{child}^0(d) = d$ and $\mathit{child}^{i+1}(d) = \mathit{child}(\mathit{child}^i(d))$ for $i \geq 0$ if $\mathit{child}^i(d)$ is non-empty.

Observe that the elements of d are just elements of A, the elements of $\mathit{child}(d)$ are pairs of elements of A, the elements of $\mathit{child}(\mathit{child}(d))$ are pairs of pairs of elements of A, and so on. One can think of each element of $\mathit{child}^i(d)$ as being a complete binary tree of depth[2] i, with elements of A at its 2^i leaves. One can also think of the entire structure representing d as a stack (of d and its descendants), each element of which is prefix-suffix pair. All the elements of d are stored in the prefixes and suffixes at the various levels of this structure, grouped into binary trees of the appropriate depths: level i contains the prefix and suffix of $\mathit{child}^i(d)$. See Figure 4.1.

Because of the pairing, we can bring *two* elements up to level i by doing *one pop* or *eject* at level $i+1$. Similarly, we can move two elements down from level i by doing one *push* or *inject* at level $i+1$. This two-for-one payoff gives the recursive slow-down that leads to real-time performance.

To obtain this real-time performance, we must guarantee that each top-level deque operation requires changes to only a constant number of levels in the recursive structure. For this reason we impose a *regularity constraint* on the structure. We assign each buffer, and each deque, a *color*, either green, yellow, or red. A buffer is *green* if it has two or three elements, *yellow* if one or four, and *red* if zero or five. Observe that we can add one element to or delete one element from a green or yellow buffer without violating its size constraint: a green buffer stays green or becomes yellow, a yellow buffer becomes green or red.

We order the colors red $<$ yellow $<$ green; red is bad, green is good. A "higher" buffer color

[2]The *depth* of a complete binary tree is the number of edges on a root-to-leaf path.

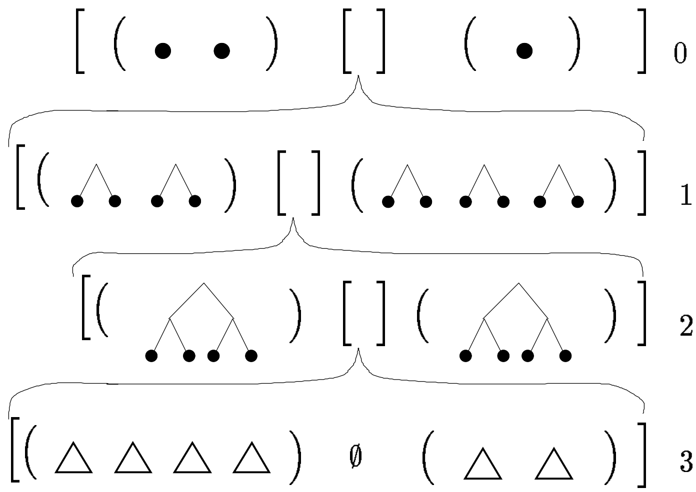

Figure 4.1: Representation of a deque. Square brackets denote the deque and its descendant deques; parentheses denote buffers. Curly brackets denote expansion of a deque into its component parts. Numbers denote levels of deques. Triangles at level three denote pairs of pairs of pairs (equivalently, complete binary trees of depth three).

indicates that more insertions or deletions on the buffer are possible before its size is outside the allowed range. We define the color of a non-empty deque to be the minimum of the colors of its prefix and suffix, unless its child and one of its buffers are empty, in which case the color of the deque is the color of its nonempty buffer.

Our regularity constraint on a deque d is a constraint on the colors of the sequence of descendant deques d, $child(d)$, $child^2(d)$, $\ldots$ We call d *semi-regular* if between any two red deques in this sequence there is a green deque, ignoring intervening yellows. More formally, d is semi-regular if, for any two red deques $child^i(d)$ and $child^j(d)$ with $i < j$, there is a k such that $i < k < j$ and $child^k(d)$ is green. We call d *regular* if d is semi-regular and if, in addition, the first non-yellow deque (if any) in the sequence is green. Observe that if d is regular or semi-regular, then $child(d)$, and indeed $child^i(d)$ for $i > 0$, is semi-regular. Furthermore if d is semi-regular and red, then $child(d)$ is regular.

Our strategy for obtaining real-time performance is to maintain the constraint that any top-level deque is regular, except possibly in the middle of a deque operation, when the deque can temporarily become semi-regular. A regular deque has a top level that is green or yellow, which means that any deque operation can be performed by operating on the appropriate top-level buffer. This may change the top level from green to yellow or from yellow to red. In either of these cases the deque may no longer be regular but only semi-regular; it will be semi-regular if the topmost non-yellow descendant deque is now red. We restore regularity by changing such a red deque to green, in the process possibly changing its own child deque from green to yellow or from yellow to red or green. Observe that such color changes, if we can effect them, restore regularity. This process corresponds to addition of 1 in the redundant binary numbering system discussed in Section 3.

In the process of changing a red deque to green, we will not change the elements it contains or their order; we merely move elements between its buffers and the buffers of its child. Thus, after making such a change, we can obtain a top-level regular deque merely by restoring the levels on top of the changed deque.

The topmost red deque may be arbitrarily deep in the recursive structure, since it can be separated from the top level by many yellow deques. To achieve real-time performance, we need constant-time access to the topmost red deque. For this reason we do not represent a deque in the obvious way, as a stack of prefix-suffix pairs. Instead, we break this stack up into substacks. There is one substack for the top-level deque and one for each non-yellow descendant deque not at the top level. Each substack consists of a top-level or non-yellow deque and all consecutive yellow proper descendant deques. We represent the entire deque by a stack of substacks of prefix-suffix pairs using this partition into substacks. An equivalent pointer-based representation is to use a node with four pointers for each non-empty descendant deque d. Two of the pointers are to the prefix and suffix at the corresponding level. One pointer is to the node for the child deque if this deque is non-empty and yellow. One pointer is to the node for the nearest non-yellow proper descendant deque, if such a deque exists and d itself is non-yellow or top-level. See Figure 4.2.

[Figure 4.2]

A single deque operation will require access to at most the top three substacks, and to at most the top two elements in any such substack. The color changes caused by a deque operation produce only minor changes to the stack partition into substacks, changes that can be made in constant time. In particular, changing the color of the top-level deque does not affect the partition into

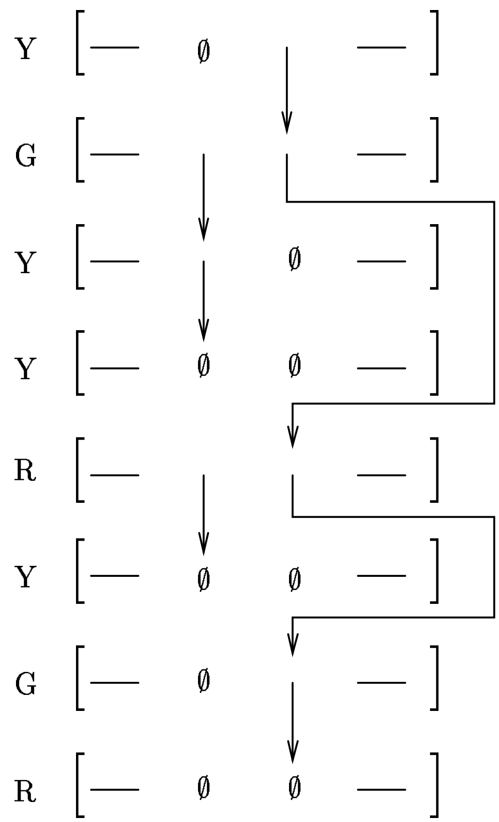

Figure 4.2: Pointer representation of stack of substacks structure. Horizontal lines denote buffers. Letters indicate deque colors. Left pointers link elements within substacks; right pointers link tops of substacks. Null pointers are denoted by $\emptyset$.

substacks. Changing the topmost red deque to green and its child from yellow to non-yellow splits one substack into its first element, now a new substack, and the rest. This is just a substack pop operation. Changing the topmost red deque to green and its child from green to yellow merges a singleton substack with the substack under it. This is just a substack push operation.

4.2 Deque Operations

All that remains is to describe the details of the buffer manipulations and verify that they produce the claimed color changes. To perform a *push* or *pop*, we push or pop the appropriate element onto or off the top-level prefix, unless this prefix and the child deque are empty, in which case we do the same to the top-level suffix. Inject and eject are symmetric. Because the original deque is regular, the top level is originally green or yellow, and any such operation can be performed without overflowing or underflowing the buffer (unless we try to pop or eject from an already empty deque). The top level may change from green to yellow, or from yellow to red, which may make the new deque semi-regular.

We restore a semi-regular deque (that is not regular) to regular as follows. Let i be the topmost red level; let P_i, P_{i+1}, S_{i+1}, S_i be the i^{th} and $i+1^{st}$-level prefixes and the $i+1^{st}$ and i^{th} level suffixes, respectively. Viewing elements from the perspective of level i, we call the elements of P_{i+1} and S_{i+1} *pairs*, since each is a pair of level-i elements. Note that if either P_{i+1} or S_{i+1} is empty, then so is d_{i+1}, since level $i+1$ cannot be red. Apply the appropriate one of the following three cases:

Two-Buffer Case: $|P_{i+1}| + |S_{i+1}| \geq 2$. If P_{i+1} is empty, pop a pair from S_{i+1} and inject it into P_{i+1}. If S_{i+1} is empty, eject a pair from P_{i+1} and push it onto S_{i+1}. If $|P_i| \geq 4$, eject two elements from P_i, pair them, and push the pair onto P_{i+1}. If $|S_i| \geq 4$, pop two elements from S_i, pair them, and inject the pair into S_{i+1}. If $|P_i| \leq 1$, pop a pair from P_{i+1} and inject its two elements individually into P_i. If $|S_i| \leq 1$, eject a pair from S_{i+1} and push its two elements onto S_i. If level $i+1$ is the bottom-most level and P_{i+1} and S_{i+1} are both now empty, eliminate level $i+1$.

One-Buffer Case: $|P_{i+1}| + |S_{i+1}| \leq 1$, and $|P_i| \geq 2$ or $|S_i| \geq 2$. If level i is the bottom-most level, create a new, empty level $i+1$. If $|S_{i+1}| = 1$, pop the pair from S_{i+1} and inject it into P_{i+1}. If $|P_i| \geq 4$, eject two elements from P_i, pair them, and push the pair onto P_{i+1}. If $|S_i| \geq 4$, pop two elements from S_i, pair them, and inject the pair into P_{i+1}. If $|P_i| \leq 1$, pop a pair from P_{i+1} and inject its two elements into P_i. If $|S_i| \leq 1$, eject a pair from P_{i+1} and push its two elements onto S_i. If P_{i+1} is now empty, eliminate level $i+1$.

No-Buffer Case: $|P_{i+1}| + |S_{i+1}| \leq 1$, $|P_i| \leq 1$, and $|S_i| \leq 1$. Among them, P_i, P_{i+1}, S_{i+1}, and S_i contain 2 or 3 level-i elements, two of which are paired in P_{i+1} or S_{i+1}. Move all these elements to P_i. Eliminate level $i+1$ if it exists.

Note: Even though each deque operation is only on one end of the deque, the regularization procedure operates on both ends of the descendant deques concurrently.

Theorem 4.1 *Given a regular deque, the method described above will perform a push, pop, inject, or eject operation in $O(1)$ time, resulting in a regular deque.*

Proof. The only non-trivial part of the proof is to verify that the regularization procedure is correct; it is then straightforward to verify that each deque operation is performed correctly and that the time bound is $O(1)$, given the stack-of-substacks representation.

If the two-buffer case occurs, both P_{i+1} and S_{i+1} are non-empty and level $i+1$ is green or yellow after the first two steps. (Level $i+1$ starts green or yellow by semi-regularity, and making both P_{i+1} and S_{i+1} non-empty cannot make level $i+1$ red.) The remaining steps make level i green and change the sizes of P_{i+1} and S_{i+1} by at most one each. The only situation in which level $i+1$ can begin green and end red is when $|P_{i+1}| = 2$ and $|S_{i+1}| = 0$ or vice-versa initially and $|P_{i+1}| = |S_{i+1}| = 0$ finally. But in this case level $i+1$ must be the bottom-most level, and it is eliminated at the end of the case. Thus this case makes the color changes needed to restore regularity.

If the one-buffer case occurs, then since level $i+1$ cannot initially be red, it or level i must be the bottom-most level. This case makes level i green and makes level $i+1$ green, yellow, or empty, in which case it is eliminated. Thus this case, also, makes the color changes needed to restore regularity.

If the no-buffer case occurs, P_{i+1} or S_{i+1} must contain a pair, because otherwise level $i+1$ will be empty, hence non-existent, and level i will be yellow if non-empty, which contradicts the fact that level i is the topmost red level. Also at most one of P_i and S_i can contain an element. It follows that this case, too, restores regularity. □

The data structure described above can be simplified if only a subset of the four operations *push, pop, inject, eject* is allowed. For example, if *push* is not allowed, then prefixes can be restricted to be of size 0 to 3, with 0 being red, 1 yellow, and 2 or 3 green. Similarly, if *eject* is not allowed, then suffixes can be restricted to be of size 0 to 3, with 0 or 1 being green, 2 yellow, and 3 red. Thus we can represent a queue (*inject* and *pop* only) with all buffers of size at most 3. Alternatively, we can represent a steque by a pair consisting of a stack and a queue. All pushes are onto the stack and all injects into the queue. A pop is from the stack unless the stack is empty, in which case it is from the queue.

5 Real-Time Catenation

Our next goal is a deque structure that supports fast catenation. Since catenable steques (deques without eject) are easier to implement than catenable deques, we discuss catenable steques here, and delay our discussion of a structure that supports the full set of operations to Section 6. Throughout the rest of the paper we refer to a catenable steque simply as a steque.

5.1 Representation

Our representation of steques is like the structure of Section 4, with two major differences in the component parts. As in Section 4, we use buffers of two different kinds, prefixes and suffixes. Unlike Section 4, each buffer is a noncatenable steque with no upper bound on its size. Such a steque can be implemented using either the method of Section 4 or the stack-reversing method sketched in Section 2. As a possible efficiency enhancement, we can store with each buffer its size. although this is not in fact necessary to obtain constant-time operations. *We require each prefix to contain at least two elements.* There is no lower bound on the size of a suffix, and indeed a suffix can be empty.

The second difference is in the components of the pairs stored in the child steque. We define a *pair over a set* A recursively as follows: a pair over A consists of a prefix of elements of A and a possibly empty steque of pairs over A. We represent a nonempty steque s over A either as a suffix

suffix(s) of elements of A, or as a triple consisting of a prefix *prefix*(s) of elements A, a child steque *child*(s) of pairs over A, and a suffix *suffix*(s) of elements of A. The child steque, if non-empty, is represented in the same way, as is each non-empty steque in one of the pairs in *child*(s). The order of elements within a steque is the one consistent with the order in each of the component paths. See Figure 5.1.

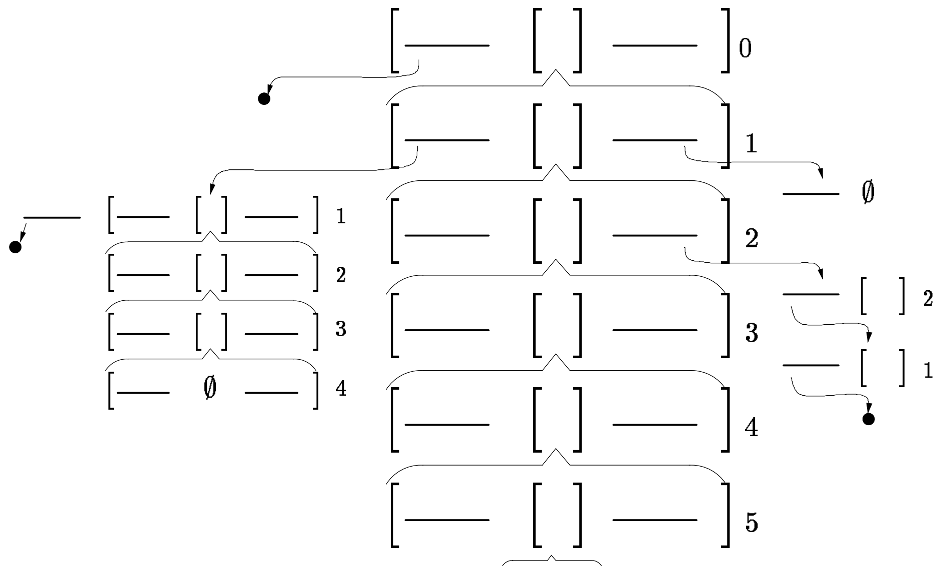

Figure 5.1: Partial expansion of the representation of a steque. Square brackets denote catenable steques; horizontal lines denote buffers. Curly brackets denote expansion of a steque into its component parts. Arrows denote membership. Circles denote elements of the base set. Numbers denote levels of steques.

[Figure 5.1]

This structure is doubly recursive; each steque in the structure is either a top-level steque, the child of another steque, or the second component of a pair that is stored in another steque. We define the *level* of a steque in this structure as follows. A top-level steque has level 0. A steque has level $i + 1$ if it is the child of a level-i steque or it is in a pair that is stored in a level-$(i + 1)$ steque. Observe that every level-i steque has the same type of elements. Namely, the elements of level-0 steques are elements of A, the elements of level-1 steques are pairs over A, the elements of level-2 steques are pairs over pairs over A, and so on. Steques to be catenated need to have the same level;

otherwise, their elements have different types.

Because of the extra kind of recursion as compared to the structure of Section 4, there is not just one sequence of descendent steques, but many: the top-level steque, and each steque stored in a pair in the structure, begins such a sequence, consisting of a steque s, its child, its grandchild, and so on. Among these descendants, the only one that can be represented by a suffix only (instead of a prefix, child, suffix triple) is the last one.

We may order the steque operations in terms of their implementation complexity as follows: *push* or *inject* is simplest, *catenate* next-simplest, and *pop* most-complicated. Each *push* or *inject* is a simple operation on a single buffer, because buffers can grow arbitrarily large, which means that overflow is not a problem. We can perform a catenate operation as just a few push or inject operations, because of the extra kind of recursion. A *pop* is the most complicated operation. It can require a *catenate*, and it may also threaten buffer underflow, which we prevent by a mechanism like that used in Section 4.

Each prefix has a *color, red* if the prefix contains two elements, *yellow* if three, and *green* if four or more. Each nonempty steque in the structure also has a color, which is the color of its prefix if it has one, and otherwise green. We call a steque s *semi-regular* if, between any pair of red steques in a descendent sequence within s, there is a green steque, ignoring intervening yellows. We call a steque s *regular* if it is semi-regular and if, in addition, the first non-yellow steque in the sequence s, $child^1(s)$, $child^2(s), \ldots$, if any, is green. As in Section 4, we maintain the invariant that any top-level steque is regular, except possibly in the middle of a steque operation, when it may be temporarily semi-regular. Observe that if s is regular, then $child(s)$ is semi-regular, and that if s is semi-regular, a steque having a green prefix and s as its child steque is regular.

Our representation of steques corresponds to that in Section 4. Namely, we represent each descendent sequence as a stack of substacks by breaking the descendent sequence into subsequences, each beginning with the first steque or a non-yellow steque and containing all consecutively following yellow steques. Each element of a substack is a pair consisting of the prefix and suffix of the corresponding steque (with a null indicator for a nonexistent prefix). Each element of a prefix or suffix is an element of the base set if the prefix or suffix is at level 0, or a pair of the appropriate type if the prefix or suffix is deeper in the structure. See Figure 5.2.

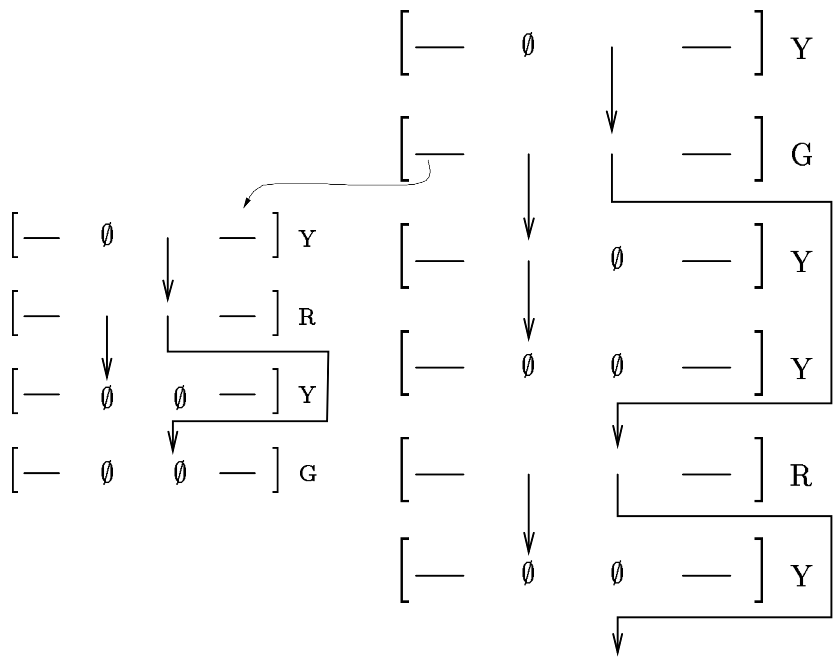

Figure 5.2: Pointer representation of the substack decomposition of part of the partially expanded steque in Figure 5.1. The sequences of descendants are shown. Letters denote steque colors. Left pointers link the elements within substacks; right pointers link the tops of substacks. Null pointers are denoted by $\emptyset$.

5.2. Steque Operations

As noted above, *push* and *inject* operations are the simplest steque operations to implement: each changes only a single buffer, increasing it in size by one. Specifically, to inject an element x into a steque s, we inject x into *suffix*(s). To push an element x onto a steque s, we push x onto *prefix*(s) unless s has no prefix, in which case we push x onto *suffix*(s). A push may change the color of the top-level steque from red to yellow or from yellow to green, but this only helps the regularity constraint and it does not change the substack decomposition.

A catenate operation is somewhat more complicated but consists of only a few *push* and *inject* operations. Specifically, to form the catenation s_3 of two steques s_1 and s_2, we apply the appropriate one of the following three cases:

Case 1: s_1 is a triple. If *suffix*(s_1) contains at least two elements, inject the pair (*suffix*$(s_1), \emptyset)$ into *child*(s_1). (This converts *suffix*(s_1) into a prefix.) Otherwise, if *suffix*(s_1) contains one element, push this element onto s_2. If s_2 is a triple, inject the pair (*prefix*(s_2), *child*(s_2)) into *child*(s_1). Let s_3 be the triple (*prefix*(s_1), *child*(s_1), *suffix*(s_2)).

Case 2: s_1 is a suffix only and s_2 is a triple. If $|$*suffix*$(s_1)| \geq 4$, push the pair (*prefix*$(s_2), \emptyset)$ onto *child*(s_2) and let the result s_3 be the triple (*suffix*(s_1), *child*(s_2), *suffix*(s_2)). (This makes *suffix*(s_1) into a green prefix.) Otherwise, pop the at most three elements on *suffix*(s_1), push them in the opposite order onto *prefix*(s_2), and let s_3 be (*prefix*(s_2), *child*(s_2), *suffix*(s_2)).

Case 3: Both s_1 and s_2 are suffixes only. If $|$*suffix*$(s_1)| \geq 4$, let s_3 be (*suffix*$(s_1), \emptyset,$ *suffix*(s_2)). (This makes *suffix*(s_1) into a green prefix.) Otherwise, pop the at most three elements on *suffix*(s_1), push them in the opposite order onto *suffix*(s_2), and let s_3 be *suffix*(s_2).

Lemma 5.1 *If s_1 and s_2 are semi-regular, then s_3 is semi-regular. If in addition s_1 is regular, then s_3 is regular.*

Proof. In Case 3, the only steque in s_3 is the top-level one, which is green. Thus s_3 is regular. In Case 2, the push onto *child*(s_2), if it happens, preserves the semi-regularity of *child*(s_2), and the prefix of the result steque s_3 is green. Thus s_3 is regular. In Case 1, both *child*(s_1) and *child*(s_2) are semi-regular. The injections into *child*(s_1) preserve its semi-regularity. Steque s_1 has the same prefix as s_1 and the same child steque as s_1, save possibly for one or two injects. Thus s_3 is semi-regular if s_1 is, and is regular if s_1 is. $\square$

A *pop* is the most complicated steque operation. To pop a steque that is a suffix only, we merely pop the suffix. To pop a steque that is a triple, we pop the prefix. This may result in a steque that is no longer regular, but only semi-regular. We restore regularity by modifying the nearest red descendant steque, say s_1, of the top-level steque, as follows. If $child(s_1)$ is empty, pop the two elements on $prefix(s_1)$, push them in the opposite order onto $suffix(s_1)$, and represent s_1 by its suffix only. Otherwise, pop a pair, say (p, s_2) from $child(s_1)$, pop the two elements on $prefix(s_1)$ and push them in the opposite order onto p, catenate s_2 and $child(s_1)$ to form s_3, and replace s_1 by the triple $(p, s_3, suffix(s_1))$.

Lemma 5.2 *The restoration method described above converts a semi-regular steque s to regular. Thus the implementaiton of* pop *is correct.*

Proof. Let s_1 be the nearest red descendant steque of s. If $child(s_1)$ is empty, s_1 is replaced by a green steque with no child, and the result is a regular steque. Suppose $child(s_1)$ is non-empty. Then $child(s_1)$ before the pop is regular, because it is semi-regular since s_1 is semi-regular and since s_1 is red the nearest non-yellow descendant of $child(s_1)$ must be green. Hence $child(s_1)$ is at least semi-regular after a pop. The triple $(p, s_3, suffix(s_1))$ replacing s_1 has p green and s_3 semi-regular, which means it is regular. $\square$

Theorem 5.1 *A* push, pop, *or* inject *on a regular steque takes $O(1)$ time and results in a regular steque. A catenation of two regular steques takes $O(1)$ time and results in a regular steque.*

Proof. The $O(1)$ time bound per steque operation is obvious if the stack of substacks representation is used. Regularity is obvious for *push* and *inject*, is true for *catenate* by Lemma 5.1, and for *pop* by Lemma 5.2. $\square$

For an alternative way to build real-time catenable steques using noncatenable stacks as buffers, see [25].

6 Catenable Deques

Finally, we extend the ideas presented in the previous two sections to obtain a data structure that supports the full set of deque operations, namely *push*, *pop*, *inject*, *eject*, and *catenate*, each in $O(1)$ time. We omit certain definitions that are obvious extensions of those in previous sections.

A common feature of the two data structures presented so far is an underlying linear skeleton (the sequence of descendants). Our structure for catenable deques replaces this linear skeleton by a binary-tree skeleton. This seems to be required to efficiently handle both *pop* and *eject*. The branching skeleton in turn requires a change in the work-allocation mechanism, which must funnel computation cycles to all branches of the tree. We add one color, orange, to the color scheme, and replace the two-beat rhythm of the green-yellow-red mechanism by a three-beat rhythm. We obtain an $O(1)$ time bound per deque operation essentially because $2/3 < 1$; the "2" corresponds to the branching factor of the tree structure, and the "3" corresponds to the rhythm of the work cycle. The connection to redundant numbering systems is much looser than for the green-yellow-red scheme used in Sections 4 and 5. Nevertheless, we are able to show directly that the extended mechanism solves our problem.

6.1 Representation

Our representation of deques uses two kinds of buffers: *prefixes* and *suffixes*. Each buffer is a non-catenable deque. We can implement the buffers either as described in Section 4 or by using the incremental stack-reversing method outlined in Section 2. Henceforth by "deque" we mean a catenable deque unless we explicitly state otherwise. As in Section 5, we can optionally store with each buffer its size, which may provide a constant-factor speedup.

We define a *triple* over a set A recursively as a prefix of elements of A, a possibly empty deque of triples over A, and a suffix of elements of A. Each triple in the deque we call a *stored triple*. We represent a non-empty deque d over A either by one triple over A, called an *only triple*, or by an ordered pair of triples over A, the *left triple* and the *right triple*. The deques within each triple are represented recursively in the same way. The order of elements within a deque is the one consistent with the order in each of the component parts.

We define a parent-child relation on the triples as follows. If t = (*prefix, deque, suffix*) is a triple with *deque* $\neq \emptyset$, the children of t are the one or two triples that make up *deque*. We define ancestors and descendants in the standard way. Under this relation, the triples group into trees, each of whose nodes is unary or binary. Each top-level triple and each stored triple is the root of such a tree, and a deque is represented by the one or two such trees rooted at the top-level triples. See Figure 6.1.

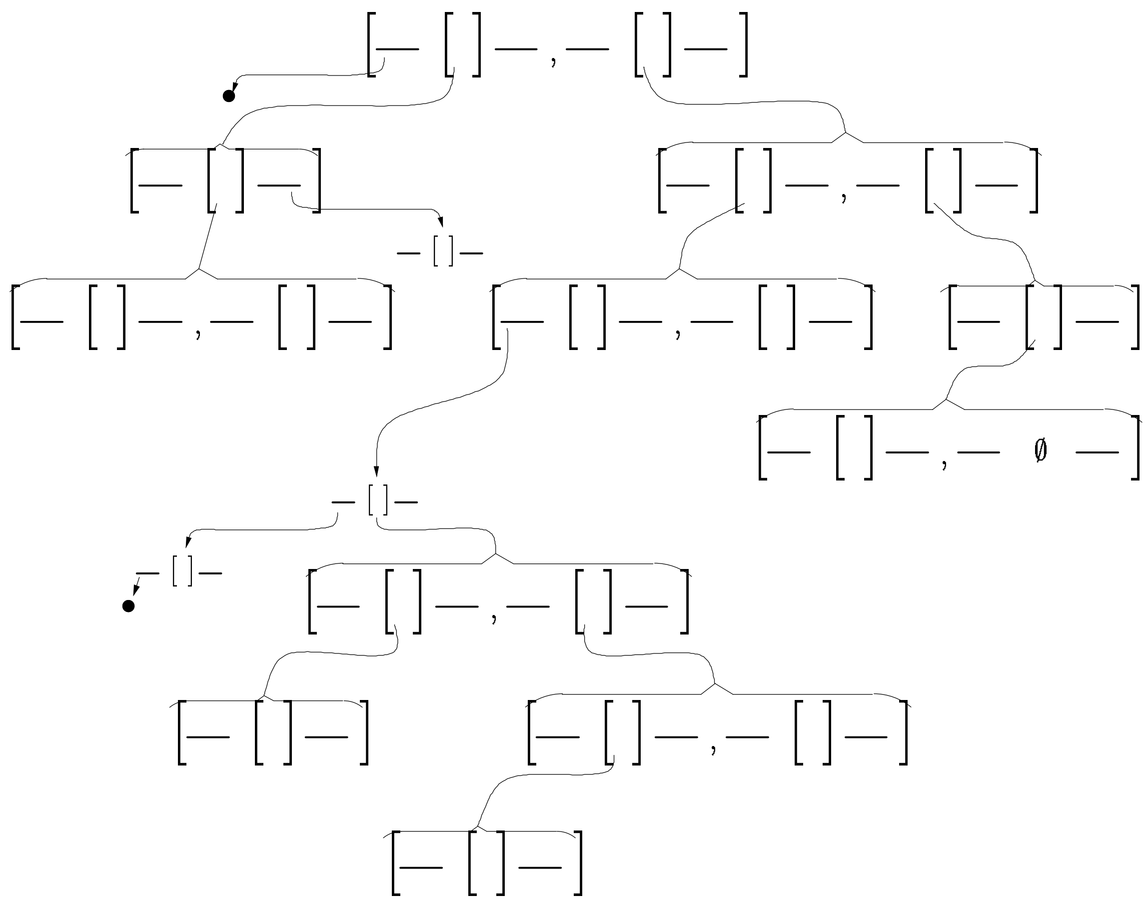

Figure 6.1: Partial expansion of the representation of a catenable deque. Conventions are as in Figure 5.1, with two triples comprising a deque separated by a comma.

[Figure 6.1]

There are four different kinds of triples: stored triples, only triples, left triples, and right triples. We impose size constraints on the buffers of a triple depending upon what kind it is. If $t = (p, d, s)$ is a stored triple, we require that both p and s contain at least three elements unless d and one of the buffers is empty, in which case the other buffer must contain at least three elements. If t is an only triple, we require that both p and s contain at least five elements, unless d and one of the buffers is empty, in which case the other buffer can contain any non-zero number of elements. If t is a left triple, we require that p contain at least five elements and s *exactly* two. Symmetrically, if t is a right triple, we require that s contain at least five elements and p exactly two.

We assign colors to the triples based on their types and their buffer sizes, as follows. Let $t = (p, d, s)$ be a triple. If t is a stored triple or if $d = \emptyset$, t is green. If t is a left triple and $d \neq \emptyset$, t is green if p contains at least eight elements, yellow if p contains seven, orange if six, and red if five. Symmetrically, if t is a right triple and $d \neq \emptyset$, t is green if s contains at least eight elements, yellow if seven, orange if six, and red if five. If t is an only triple with $d \neq \emptyset$, t is green if both p and s contain at least eight elements, yellow if one contains seven and the other at least seven, orange if one contains six and the other at least six, and red if one contains five and the other at least five.

The triples are grouped into trees by the parent-child relation. We partition these trees into paths as follows. Each yellow or orange triple has a *preferred child*, which is its left child or only child if the triple is yellow and its right child or only child if the triple is orange. The preferred children define preferred paths, each starting at a triple that is not a preferred child and passing through successive preferred children until reaching a triple without a preferred child. Thus each preferred path consists of a sequence of zero or more yellow or orange triples followed by a green or red triple. (Every triple with no children is green.) We assign each preferred path a color, green or red, according to the color of its last triple.

We impose a regularity constraint on the structure, like those in Sections 4 and 5 but a little more complicated. We call a deque *semi-regular* if both of the following conditions hold:

(1) Every preferred path that starts at a child of a red triple is a green path.

(2) Every preferred path that starts at a non-preferred child of an orange triple is a green path.

This definition implies that if a deque is semi-regular, then all the deques in its constituent triples are semi-regular. We call a deque *regular* if it is semi-regular and if, in addition, each preferred path that starts at a top-level triple (one of the one or two representing the entire deque) is a green path. We maintain the invariant that any top-level deque is regular, except possibly in the middle of a deque operation, when it may temporarily be only semi-regular. Note that an empty deque is regular.

We need a representation of the trees of triples that allows us to shortcut preferred paths. To this end, we introduce the notions of an *adopted child* and its *adoptive parent.* Every green or red triple that is on a preferred path of at least three triples is an adopted child of the first triple on this path, which is its adoptive parent. That is, there is an adoptive parent-adopted child relationship between the first and last triples on each preferred path containing at least three triples.

We define the *compressed forest* by the parent-child relation on triples, except that each adopted child is a child of its adoptive parent instead of its natural parent. In the compressed forest, each triple has at most three children, one of which may be adopted. We represent a deque by its compressed forest, with a node for each triple containing the prefix and suffix of the triple and pointers to the nodes representing its child triples. See Figure 6.2.

[Figure 6.2]

The operations that we describe in the next section rely on the following property of the compressed forest representation. Given the node of a triple $t = (p, d, s)$, we can extract in constant time a pointer to a compressed forest representation for d when t is a top-level triple, a stored triple, or the color of t is either red or green.

6.2 Deque Operations

The simplest deque operations are *push* and *inject.* Next is *catenate*, which may require a *push* or an *inject* or both. The most complicated operations are *pop* and *eject*, which can violate regularity and may force a repair deep in the forest of triples (but shallow in the compressed forest).

We begin by describing *push*; *inject* is symmetric. Let d be a deque onto which we wish to push an element. If d is empty, we create a new triple t to represent the new deque, with one nonempty buffer containing the pushed element. If d is nonempty, let $t = (p_1, d_1, s_1)$ be its left triple or its only triple. If p_1 is nonempty, we push the new element onto p_1; otherwise, we push the new element onto s_1.

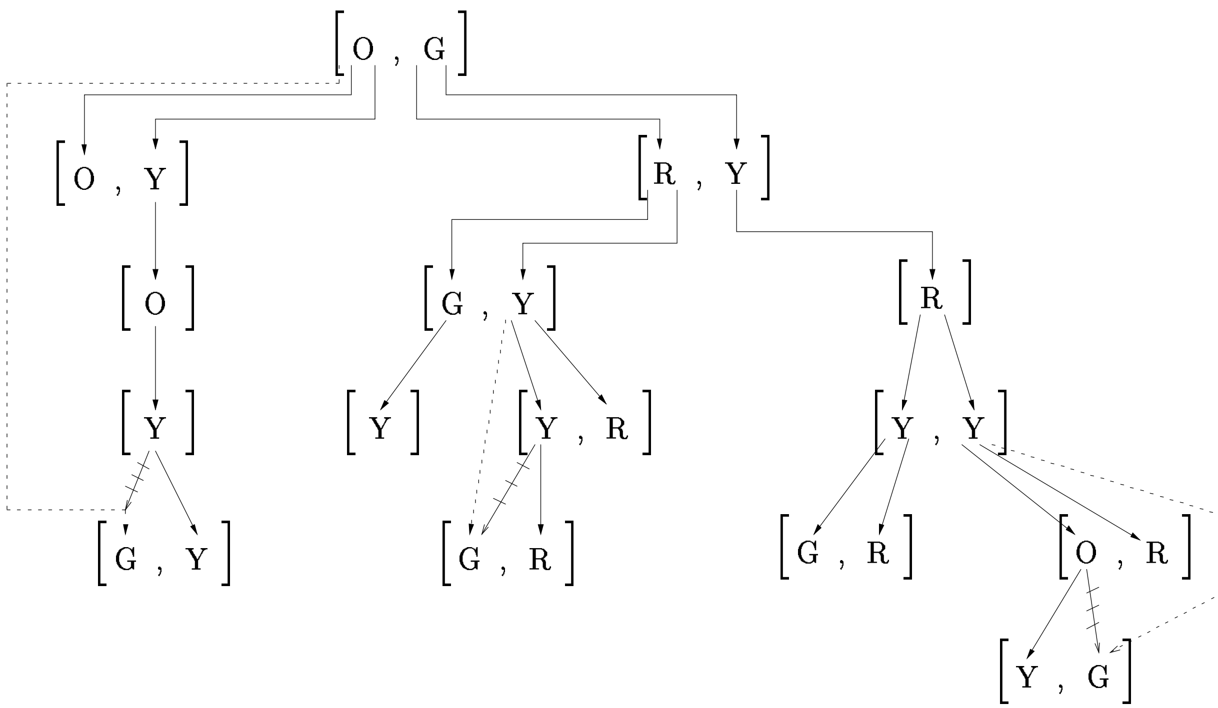

Figure 6.2: Top-level trees in the compressed forest representation of a deque. Letters denote triples of the corresponding colors. Dashed arrows denote adoptive-parent, adoptive-child relationships that replace the natural parent-child relationships marked by hatched arrows. The complete compressed forest representation (not shown) would include the buffers of the triples and the lower-level compressed trees rooted at the stored triples.

Lemma 6.1 *A push onto a semi-regular deque produces a semi-regular deque; a push onto a regular deque produces a regular deque.*

Proof. If the push does not change the color of t, the lemma is immediate. If the push does change the color of t, it must be from yellow to green, from orange to yellow, or from red to orange. (Red-to-orange can only happen if the original deque is semi-regular but not regular.) The yellow-to-green case obviously preserves both semi-regularity and regularity. In the orange-to-yellow case, let u be the non-preferred child of t before the push if t has a non-preferred child. If u exists, semi-regularity implies that the preferred path containing u is a green path. The push adds t to the front of this path. This means that the push preserves both semi-regularity and regularity. If u does not exist, then the push does not change any of the preferred paths but only changes t from orange to yellow. In this case also the push preserves both semi-regularity and regularity. In the red-to-orange case, before the push every child of t starts a preferred path that is green, which means that after the push the non-preferred child of t, if it exists, starts a preferred path that is green. Thus the push preserves semi-regularity. $\Box$

Note that the only effect a push has on the preferred path decomposition is to add t to or delete t from the front of a preferred path (or both). This means that the compressed forest can be updated in $O(1)$ time during a push.

Next, we describe *catenate*. Let d and e be the two deques to be catenated. Assume both are nonempty; otherwise the *catenate* is trivial. To catenate d and e, we apply the appropriate one of the following four cases:

Case 1. All the buffers in the two, three, or four top-level triples of d and e are nonempty. The new deque will consist of two triples t and u, with t formed from the top-level triple or triples of d, and u formed from the top-level triple or triples of e. There are four subcases in the formation of t.

Subcase 1a. Deque d consists of two triples $t_1 = (p_1, d_1, s_1)$ and $t_2 = (p_2, d_2, s_2)$, with $d_1 \neq \emptyset$. Combine s_1 and p_2 (each containing exactly two elements) into a single buffer p_3. Eject the last two elements from s_2 and add them to a new buffer s_3; let s_2' be the rest of s_2. Inject (p_3, d_2, s_2') into d_1 to form d_1'. Let $t = (p_1, d_1', s_3)$.

Subcase 1b. Deque d consists of two triples $t_1 = (p_1, \emptyset, s_1)$ and $t_2 = (p_2, d_2, s_2)$. Inject the elements in s_1 and p_2 into p_1 to form p_1'. Replace the representation of d by the only triple (p_1', d_2, s_2) and apply Subcase 1c or 1d as appropriate.

Subcase 1c. Deque d consists of an only triple $t_1 = (p_1, d_1, s_1)$ with $d_1 \neq \emptyset$. Eject the last two elements from s_1 and add them to a new buffer s_2. Let the remainder of s_1 be s_1'. Form a new triple $(\emptyset, \emptyset, s_1')$ and inject it into d_1 to form d_1'. Let $t = (p_1, d_1', s_2)$.

Subcase 1d. Deque d consists of an only triple $t_1 = (p_1, \emptyset, s_1)$. If s_1 contains at most eight elements, move all but the last two elements of s_1 to p_1 to form p_1'; let the remaining two elements of s_1 form s_1'. Let $t = (p_1', \emptyset, s_1')$. Otherwise ($s_1$ contains more than eight elements), move the first three elements on s_1 into p_1 to form p_1', move the last two elements on s_1 into a new buffer s_2, and let the remainder of s_1 be s_1'. Push the triple $(\emptyset, \emptyset, s_1')$ onto an empty deque to form the deque d_2. Let $t = (p_1', d_2, s_2)$.

Operate symmetrically on e to form u.

Case 2. Deque d consists of an only triple $t_1 = (p_1, d_1, s_1)$ with only one nonempty buffer, and all the buffers in the top-level triple or triples of e are nonempty. Let $t_2 = (p_2, d_2, s_2)$ be the left or only triple of e. We combine t_1 and t_2 to form a new triple t, which is the left or only triple of the new deque; the right triple of e, if it exists, is the right triple of the new deque. To form t, let p_3 be the nonempty one of p_1 and s_1. If p_3 contains less than eight elements, push all these elements onto p_2 to form p_2', and let $t = (p_2', d_2, s_2)$. Otherwise, form a triple $(p_2, \emptyset, \emptyset)$, push it onto d_2 to form d_2', and let $t = (p_3, d_2', s_2)$.

Case 3. Deque e consists of an only triple with only one nonempty buffer, and all the buffers in the top-level triple or triples of d are nonempty. This case is symmetric to Case 2.

Case 4. Deques d and e each consist of an only triple with a single nonempty buffer. Let p be the nonempty buffer of d and s the nonempty buffer of e. If either p or s contains fewer than eight elements, combine them into a single buffer b and let $t = (b, \emptyset, \emptyset)$. Otherwise, let $t = (p, \emptyset, s)$.

Lemma 6.2 *A catenation of two semi-regular deques produces a semi-regular deque. A catenation of two regular deques produces a regular deque.*

Proof. Consider Case 1. We shall show that, in each subcase, triple t and its descendants satisfy the semi-regularity or regularity constraints as appropriate. The symmetric argument applies to u, which gives the lemma for Case 1.

In Subcase 1d, triple t is green and either has a green child and no grandchildren or no child at all. In either case t satisfies the regularity constraints. Consider Subcase 1c. Deque d_1' is formed from a semi-regular deque d_1 by an injection and hence is semi-regular by Lemma 6.1. The color of triple $t = (p_1, d_1', s_2)$ is at least as good as the color of triple $t_1 = (p_1, d_1, s_1)$, since the color of t depends only on the size of p_1, whereas the color of t_1 depends on the minimum of the sizes of p_1 and s_1. We must consider several cases, depending on the color of t_1 and on whether we are trying to verify regularity or only semi-regularity. If t_1 is green, t and its descendants satisfy the regularity constraints. If t_1 is red, the semi-regularity of d implies that d_1 and hence d_1' is regular, and t and its descendants satisfy the semi-regularity constraints. If t_1 is orange and d is regular, then d_1 and hence d_1' must be regular, and t and its descendants satisfy the regularity constraints. If t_1 is orange and d is only semi-regular, then the non-preferred child of t_1, if it exists, starts a green path. The corresponding non-preferred child of t also starts a green path, by an argument like that in Lemma 6.1. This means that t and its descendants satisfy the semi-regularity constraints. If t_1 is yellow, the semi-regularity of d_1' implies that t and its descendants satisfy the semi-regularity constraints. Finally, if t_1 is yellow and d is regular, then the preferred child of t_1 is on a green path, as is the corresponding child of t, again by an argument like that in Lemma 6.1. Thus t and its descendants satisfy the regularity constraints.

Subcase 1b creates a one-triple representation of d that is semi-regular if the original representation is and regular if the original one is. Subcase 1b is then followed by an application of 1c or 1d as appropriate. In this case, too, triple t and its descendants satisfy the semi-regularity or regularity constraints as appropriate.

The last subcase is Subcase 1a. As in Case 1c, the argument depends on the color of $t_1 = (p_1, d_1, s_1)$ and whether we are trying to verify regularity or semi-regularity. In this case, t_1 and triple $t = (p_1, d_1', s_2)$ have exactly the same color. Deque d_1' is semi-regular by Lemma 1, since d_1 and d_2 are semi-regular. The remainder of the argument is exactly as in Subcase 1c.

Consider Case 2. If p_3 contains less than eight elements, then t is formed by doing up to seven pushes onto t_2, so t satisfies regularity or semi-regularity by Lemma 6.1. Otherwise, deque d_2' is formed from deque d_2 by doing a push, and triple t is either green or has the same color as triple t_2. The remainder of the argument is exactly as in Subcase 1c.

Case 3 is symmetric to Case 2. Case 4 obviously preserves both semi-regularity and regularity. $\square$

A catenate changes the colors and compositions of triples in only a constant number of levels at the top of the compressed forest structure. Hence this structure can be updated in constant time during a catenate.

We come finally to the last two operations, *pop* and *eject*. We shall describe *pop*; *eject* is symmetric. A *pop* consists of two parts. The first removes the element to be popped and the second repairs the damage to regularity caused by this removal. Let t be the left or only triple of the deque d to be popped. The first part of the pop consists of popping the prefix of t, or popping the suffix if the prefix is empty, and replacing t in d by the triple t' resulting from this pop, forming d'. As we shall see below, d' may not be regular but only semi-regular, because the preferred path starting at t' may be red. In this case let u be the red triple at the end of this preferred path. Using the compressed forest representation, we can access u in constant time. The second part of the pop replaces u and its descendants by a tree of triples representing the same elements but which has a green root v and satisfies the regularity constraints. This produces a regular representation of d' and finishes the pop.

To repair $u = (p_1, d_1, s_1)$, we apply the appropriate one of the following cases. Since u is red, $d_1 \neq \emptyset$.

Case 1. Triple u is a left triple. Pop the first triple (p_2, d_2, s_2) from d_1 (without any repair); let d_1' be the rest of d_1.

> **Case 1a.** Both p_2 and s_2 are nonempty. Push $(\emptyset, \emptyset, s_2)$ onto d_1', forming d_1''. Push the elements on p_1 onto p_2, forming p_2'. Catenate deques d_2 and d_1'', forming d_3. Let $v = (p_2', d_3, s_1)$.
>
> **Case 1b.** One of p_2 and s_2 is empty. Combine p_1, p_2, and s_2 into a single buffer p_3. Let $v = (p_3, d_1', s_1)$.

Case 2. Triple u is an only triple. Apply the appropriate one of the following three cases.

Case 2a. Suffix s_1 contains at least eight elements. Proceed as in Case 1, obtaining $v = (p_4, d_4, s_1)$ with p_4 containing at least eight elements.

Case 2b. Prefix p_1 contains at least eight elements. Proceed symmetrically to Case 1, obtaining $v = (p_1, d_4, s_4)$ with s_4 containing at least eight elements.

Case 2c. Both p_1 and s_1 contain at most seven elements. Pop the first triple (p_2, d_2, s_2) from d_1 (without any repair); let d_1' be the rest of d_1. If $d_1' = \emptyset$, combine p_1 and p_2 to form p_4, combine s_2 and s_1 to form s_4, and let $v = (p_4, d_2, s_4)$. Otherwise, eject the last triple (p_3, d_3, s_3) from d_1' (without any repair); let d_1'' be the rest of d_1'. If one of p_2 and s_2 is empty, combine p_1, p_2, and s_2 into a single buffer p_4 and let $d_4 = d_1''$. Otherwise, push $(\emptyset, \emptyset, s_2)$ onto d_1'', forming d_1'''; push the elements on p_1 onto p_2, forming p_4; and catenate d_2 and d_1''' to form d_4. Symmetrically, if one of p_3 and s_3 is empty, combine p_3, s_3, and s_1 into a single buffer s_4, and let $d_5 = d_4$. Otherwise, inject $(p_3, \emptyset, \emptyset)$ into d_4, forming d_4'; inject the elements on s_1 into s_3, forming s_4; and catenate d_4' and d_3 to form d_5. Let $v = (p_4, d_5, s_4)$.

Lemma 6.3 *Removing the first element (from the first buffer) in a regular deque produces a semi-regular deque whose only violation of the regularity constraint is that the preferred path containing the left or only top-level triple may be red. Removing the first and last elements (from the first and last buffers, respectively) in a regular deque produces a semi-regular deque.*

Proof. Let d be a regular deque, and let $t = (p_1, d_1, s_1)$ be its left or only triple. Let t' be formed from t by popping p_1, and let d' be formed from d by replacing t by t'. If t is green, yellow, or orange (t cannot be red by regularity), then t' can be yellow, orange, or red, respectively. (One of these transitions will occur unless both t and t' are green, in which case d' is regular since d is.) In each case it is easy to verify that the regularity of d implies that triple t' satisfies the appropriate semi-regularity constraint; so do all other triples since their colors don't change. The only possible violation of regularity is that the preferred path containing t' may be red. An analogous argument shows that if the last element of d' is removed to form d'' then d'' will still be semi-regular: if t is the only triple of d, the two removals can degrade its color by only one color; if t is a left triple, an argument symmetric to that above applies to its sibling. $\square$

Lemma 6.4 *Popping a regular deque produces a regular deque.*

Proof. Let d be the deque to be popped, and let d' be the deque formed by removing the first element from the first buffer of d. Let t' be the left or only triple of d'. By Lemma 6.3, d' is semi-regular, and the only violation of regularity is that the preferred path containing t' may be red. If this preferred path is green, then d' is regular, the pop is finished, and the lemma is true. Suppose, on the other hand, that this preferred path is red. Let $u = (p_1, d_1, s_1)$ be the red triple on this path. Since d' is semi-regular and u is red, d_1 must be regular. We claim that the repair described above in Cases 1 and 2 replaces u and its descendants by a tree of triples with a green root satisfying the semi-regularity constraints, which implies that the deque d'' resulting from the repair is regular, thus giving the lemma.

Consider Case 1 above. Since d_1 is regular, the deque d_1' formed from d_1 by popping the triple (p_2, d_2, s_2) is semi-regular by Lemma 6.3. In Case 1a, the push onto d_1' to form d_1'' leaves d_1'' semi-regular by Lemma 6.1. Deque d_2 is semi-regular since d_1 is regular, and by Lemma 6.2 the deque d_3 formed by catenating d_2 and d_1'' is semi-regular. The triple $v = (p_2', d_3, s_1)$ is green. This gives the claim. In Case 1b, the triple $v = (p_3, d_1', s_1)$ is green, and d_1' is semi-regular, again giving the claim.

Consider Case 2 above. The same argument as in Case 1 verifies the claim in Cases 2a and 2b. In Case 2c, if $d_1' = \emptyset$, v is green and d_2 is semi-regular, which gives the claim. In Case 2c, d_1'' is semi-regular by Lemma 6.3, deque d_5 is semi-regular by appropriate applications of Lemmas 6.1 and 6.2, and v is green. Again the claim is true. $\square$

As with the other operations, a pop changes only a constant number of levels at the top of the compressed forest and hence can be performed in constant time.

Theorem 6.1 *Each of the deque operations takes $O(1)$ time and preserves regularity.*

Proof. It is straightforward to verify that the compressed forest representation allows each of the deque operations to be performed as described in $O(1)$ time. Lemmas 6.1, 6.2, and 6.4 give preservation of regularity. $\square$

The deque representation we have presented is a hybrid of two alternative structures described in [25], one based on pairs and quadruples and the other, suggested by Okasaki [34], based on triples and quintuples. The present structure offers some conceptual simplifications over these alternatives. The buffer size constraints in our representation can be reduced slightly, at the cost of making the structure less symmetric. For example, the lower bounds on the suffix sizes of right triples and only triples can be reduced by one, while modifying the definition of colors appropriately.

7 Further Results and Open Problems

We conclude in this section with some additional results and open problems. We begin with two extensions of our structures, then mention some recent work, and finally give some open problems.

If the set A of elements to be stored in a deque has a total order, we can extend all 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). Each operation remains constant-time, and the implementation remains purely functional. We merely have to store with each buffer, each deque, and each pair the minimum element contained in it. For related work see [3, 4, 19, 31].

We can also support a *flip* operation on deques, for each of the structures in Sections 4 and 6. 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 4, 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 requires a little more work. We need to symmetrize the structure and add reversal bits at all levels. The only non-symmetry in the structure is in the definition of preferred children: the preferred child of a yellow triple is its left child and the preferred child of an orange triple is its right child. Flipping exchanges left and right, but we do *not* want this operation to change preferred children; we want the partition of the compressed forest into preferred paths to be unaffected by a flip. Thus when we create a brand-new triple we designate its current left child to be its preferred child if it is yellow and its current right child to be the preferred child if it is orange. When a triple changes from orange to yellow or yellow to orange, we switch its preferred child, irrespective of current left and right.

To handle flipping, we add a reversal bit for every deque and every buffer in the structure. A reversal bit set to 1 means that the entire deque or buffer is flipped. Reversal bits are cumulative along paths of descendants in the compressed forest: for a given deque or buffer, it is reversed if an odd number of its ancestors (including itself) have reversal bits set to 1. To flip an entire deque, we flip its reversal bit. Whenever doing a deque operation, we push reversal bits down in the structure so that each deque actually being manipulated is not reversed; for reversed buffers, push and inject, and pop and eject, switch roles. The details are straightforward.

Now we turn to recent related work. In work independent of ours, Okasaki [33, 35] has devised a confluently persistent implementation of catenable stacks (or steques). His implementation is not real-time but gives constant amortized time bounds per operation. It is also not purely functional, but uses memoization. Okasaki uses rooted trees to represent the stacks. Elements are popped using a memoized version of the *path reversal* technique previously used in a data structure for the disjoint set union problem [45]. Though Okasaki's solution is neither real-time nor purely functional, it is simpler than ours. Extending Okasaki's method to the case of deques is an open problem.

After seeing an early version of our work [27], Okasaki [35, 36] observed that if amortized time bounds suffice and memoization is allowed, then all of our data structures can be considerably simplified. The idea is to perform fixes in a lazy fashion, using memoization to record the results. This avoids the need to maintain the "stack of stacks" structures in our representations, and also allows the buffers to be shorter. Okasaki called the resulting general method "implicit recursive slow-down". He argues that the standard techniques of amortized analysis [44] do not suffice in this case because of the need to deal with persistence. His idea is in fact much more general than recursive slow-down, however, and the standard techniques [44] do indeed suffice for an analysis. Working with Okasaki, we have devised even simpler versions of our structures that need only constant-size buffers and take $O(1)$ amortized time per deque operation, using a replacement operation that generalizes memoization [26].

Finally, we mention some open problems. As noted above, one is to extend Okasaki's path reversal technique to deques. A second one is to modify the structure of Section 6 to use buffers of bounded size. We know how to do this for the case of stacks, but the double-ended case has unresolved technicalities. Of course, one solution is to plug the structure of Section 4 in-line into the structure of Section 6 and simplify to the extent possible. But a more direct approach may well work and lead to a simpler solution. Another open problem is to devise a version of the structure in Section 6 that uses only one subdeque instead of two, thus leading to a linear recursive structure. A final open problem is to devise a purely functional implementation of finger search trees (random-access lists) with constant-time catenation. Our best solution to this problem has $O(\log \log n)$ catenation time [28].

Acknowledgements

We thank Adam Buchsbaum, David Wagner, Ian Munro, and Chris Okasaki for their vital contributions to this paper. Adam Buchsbaum engaged in extensive and fruitful discussions concerning our ideas. David Wagner suggested the idea of the color invariant as an alternative to the explicit use of binary counting as a work allocation mechanism. Ian Munro, after seeing a presentation of our ideas, pointed out the connection of the color invariant to the redundant binary representation of [9]. Chris Okasaki provided valuable comments on drafts of our work. We also thank the referees for their insightful and valuable suggestions.

References

[1] J. Allen. *Anatomy of LISP*. Mcgraw-Hill Computer Science Series. McGraw-Hill, New York, 1978.

[2] G. S. Brodal. Worst-case efficient priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, 1996.

[3] 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.

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

[5] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, 1982.

[6] B. Chazelle. How to search in history. *Information and control*, 64:77–99, 1985.

[7] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[8] Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead turing machines, and purely functional programming. In *Proc. of the Conference on Functional Programming and Computer Architecture*, pages 289–298, Copenhagen, 1993.

[9] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University, Palo Alto, 1977.

[10] R. Cole. Searching and storing similar lists. *J. of Algorithms*, 7:202–220, 1986.

[11] P. F. Dietz. Fully persistent arrays. In *Proceedings of the 1989 Workshop on Algorithms and Data Structures, LNCS 382*, pages 67–74, 1995.

[12] D. P. Dobkin and J. I. Munro. Efficient uses of the past. *J. of Algorithms*, 6:455–465, 1985.

[13] J. Driscoll, D. Sleator, and R. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.

[14] J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *J. of Computer and System Sciences*, 38:86–124, 1989.

[15] M. Felleisen. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[16] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proc. Conference on Lisp and Functional Programming*, pages 52–62, 1988.

[17] Patrick C. Fisher, Albert R. Meyer, and Arnold L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the ACM*, 19(4):590–607, 1972.

[18] G. N. Frederickson. Optimal selection in a min-heap. *Information and Compution*, 104:197–214, 1993.

[19] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 12(4):197–200, 1986.

[20] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[21] R. Hood. *The efficient implementation of very-high-level programming language constructs.* PhD thesis, Dept. of Computer Science, Cornell University, 1982.

[22] R. Hood and R. Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13:50–54, 1981.

[23] Rob R. Hoogerwood. A symmetric set of efficient list operations. *J. Functional Progamming*, 2(4):505–513, 1992.

[24] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 158–168, 1988.

[25] H. Kaplan. *Purely Functional Lists.* PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ, 1996.

[26] H. Kaplan, C. Okasaki, and R. E. Tarjan. Simple confluently persistent catenable lists (extended abstract). In *Proc.* 6^{th} *Scandanavian Workshop on Algorithm Theory*, pages 119–130, 1998.

[27] H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 93–102. ACM Press, 1995.

[28] 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.

[29] Donald E. Knuth. *Fundamental Algorithms, Volume 1 of The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[30] S. R. Kosaraju. Real-time simulation of concatenable double-ended queues by double-ended queues. In *Proc. 11th ACM Symposium on Theory of Computing*, pages 346–351, 1979.

[31] 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.

[32] B. L. Leong and J. I. Seiferas. New real-time simulations of multihead tape units. *Journal of the ACM*, 28(1):166–180, 1981.

[33] 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.

[34] C. Okasaki. Simple and efficient purely functional queues and deques. *J. Functional Progamming*, 5(4):583–592, 1995.

[35] C. Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996; also Cambridge University Press, 1998.

[36] C. Okasaki. Catenable double-ended queues. In *Proc. International Conference on Functional Programming*, pages 66–74, 1997.

[37] M. H. Overmars. Searching in the past, I. Technical Report RUU-CS-81-7, Department of Computer Science, University of Utrecht, Utrecht,The Netherlands, 1981.

[38] M. H. Overmars. Searching in the past, II: General transforms. Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1981.

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

[40] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[41] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation: An International Journal*, 3:67–99, 1990.

[42] J. M. Spitzer, K. N. Levitt, and L. Robinson. An example of hierarchical design and proof. *Communications of the ACM*, 21(12):1064–1075, 1978.

[43] Hans-Jörg Stoss. K-band simulation von k-kopf-turing-maschinen. *Computing*, 6(3):309–317, 1970.

[44] R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods*, 6(2):306–318, 1985.

[45] R. E. Tarjan and J. Van Leeuwen. Worst case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.