Real–Time Deques, Multihead Turing Machines, and Purely Functional Programming

Tyng–Ruey Chuang and Benjamin Goldberg
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York, NY 10012, USA
chuang@cs.nyu.edu, goldberg@cs.nyu.edu

Abstract

We answer the following question: Can a deque (double–ended queue) be implemented in a purely functional language such that each push or pop operation on either end of a queue is accomplished in $O(1)$ time in the worst case? The answer is yes, thus solving a problem posted by Gajewska and Tarjan [14] and by Ponder, McGeer, and Ng [25], and refining results of Sarnak [26] and Hoogerwoord [18].

We term such a deque *real–time*, since its constant worst–case behavior might be useful in real time programs (assuming real–time garbage collection [3], *etc.*) Furthermore, we show that no restriction of the functional language is necessary, and that push and pop operations on *previous* versions of a deque can also be achieved in constant time.

We present a purely functional implementation of real–time deques and its complexity analysis. We then show that the implementation has some interesting implications, and can be used to give a real–time simulation of a multihead Turing machine in a purely functional language.

1 Introduction and Survey

In a functional program, if an aggregate data structure is updated then both the original version and the updated version of the aggregate must be preserved, preferably at a small cost, to maintain referential transparency. It is generally regarded as too expensive to make a complete copy of the aggregate that differs from the original only in the updated position. There have been various approaches to solve the aggregate update problem. If compile–time program analysis or run–time tests can determine that the original version of an aggregate will not be referenced following an update, then the update can be performed in place [6,7,19,24,28,29].

A functional language can also provide language primitives for writing single–threaded programs such that they can easily be recognized and implemented by the compiler. A program is single–threaded if all operations on aggregates only refer to their newest versions. Thus, all update operations in a single–threaded program can be performed in place because previous versions of aggregates will never be needed and can be safely overwritten [2,16,20,31,32].

ACM-FPCA'93-6/93/Copenhagen, DK

Another general approach is to design efficient algorithms to make aggregate data structures fully persistent (*i.e.*, purely functional), such that after a sequence of updates the newest version and all previous versions of the aggregate are still accessible [1,4,5,10,11,17,22,23,26,27].

General techniques for making aggregate data structures fully persistent are described by Driscoll, Sarnak, Sleator, and Tarjan [11]. However, the techniques often rely on side–effects to achieve good time and space performance. It is not clear how they can be implemented in a purely functional language without losing their efficiency. For an advocate of functional programming, the challenge is to demonstrate that certain aggregates can be implemented efficiently in purely functional languages.

When discussing and comparing efficient implementations of fully persistent aggregate data structures, it is helpful that we ask the following questions regarding the implementations:

- Is the implementation a *purely functional* one? Or, must it use *side–effects* and be implemented in a side–effecting language?
- Is the stated cost of each aggregate operation a *worst–case* cost or an *amortized* one?
- Is the implementation good for *multi–threaded* applications? Or is it only suitable for *single–threaded* applications?

While a worst–case analysis measures the cost of an operation in an isolated context, an amortized analysis measures the cost of an operation averaged over a worst–case sequence of operations [30]. Amortized analysis should not be confused with average–case analysis, which is often based on some probabilistic assumptions. An implementation is said to be *real–time* if each operation costs only constant overhead in the worst case.

Notice that a purely functional implementation of an aggregate data structure automatically makes it fully persistent because no side–effect is used in the implementation and previous versions of an aggregate are always accessible. The problem is that the implementation may be inefficient, even under the assumption that it will only be used for single–threaded applications. Also, an implementation (either in purely functional or side–effecting languages) with good amortized performance does not necessarily mean it is good for multi–threaded applications, because the amortized cost may be measured over a single–threaded sequence of operations.

Hood and Melville [17] show that a FIFO queue can be implemented in pure Lisp (where no side–effect is allowed) such that, in the worst case, push or pop operations on the queue cost only $O(1)$ time and consume $O(1)$ space. This is an interesting result for several reasons. First of all, the $O(1)$ complexity applies not only to the newest version of a queue, but to previous versions as well. This makes their implementation suitable for multi–threaded applications. Secondly, it had not previously been shown, even for languages allowing side-effects, how a fully persistent real–time queue could be implemented. For example, suppose that the queue is implemented in a side–effecting language as a single–linked list with access pointers to both ends and the standard technique of [11] is used to make the single–linked list fully persistent. Each of the pop or push operations to any version of a queue will need $O(1)$ *amortized* time and space.

We extend in this paper the result of Hood and Melville such that using a purely functional language, each of the push or pop operations on either end of any version of a deque costs only $O(1)$ time in the worst case. Efficient implementation of deques in a purely functional language was considered both by Gajewska and Tarjan [14] and by Ponder, McGeer, and Ng [25], but without giving a solution. Our results show that it can be done.

The techniques we use are not new. We will use two stacks to represent a deque, with each stack top corresponding to one of the two open ends of the deque. The two stacks are balanced at all time in that the bigger stack is never more than three times the size of the smaller stack. A pop operation on a non–empty deque is performed directly on one of the stacks. The challenge is to make the design as simple as possible — such that a purely functional implementation becomes straightforward — and to get its complexity analysis right. The two–stacks representation is used by Gries [15, pages 250–251] to design purely functional queues with constant amortized cost per operation. Hoogerwoord [18] shows that the representation can be extended to purely functional deques with constant amortized cost per operation. However, these two designs are not suitable for multi–threaded applications because the constant cost for each operation is amortized along a single thread of operations. Hood and Melville [17] show that, by incrementally copying elements from one stack to another, real–time queues can be implemented in a purely functional way. A design for fully persistent deques with constant worst–case time per operation is described in Sarnak's Ph.D. thesis [26]. However, his design is not purely functional because side–effects are used in the implementation. His method, and the associated complexity analysis, is also more complicated than ours. Nevertheless, the basic idea is similar in his and our design: a deque is represented as two stacks and they are kept balanced by incrementally moving elements between them.

In section 2, we outline Hood and Melville's purely function method for real–time FIFO queues. We then describe in section 3 a purely functional implementation of deques with good amortized performance. This design is also described by both Sarnak and Hoogerwoord. Our purely functional implementation of deques with real–time performance is presented in section 4. Section 5 describes its implications in purely functional list processing. It also shows how to simulate in real–time, in a purely functional language, a multihead Turing machine by using multiple real–time deques. Section 6 contains some remarks on real–time processing in a purely functional setting. Section 7 discusses related work and future work.

In this paper, we will use the notation $P = (p_1, p_2, \ldots, p_m)$ to describe a sequence P of m elements $p_1, p_2, \ldots, p_m$. The concatenation of sequences P and Q is denoted by PQ. The notation $P^{\blacktriangleleft}$ describes a stack consisting of the sequence P, with p_1 at the top of the stack and p_m at the bottom. Similarly, $P^{\blacktriangleright}$ is also a stack consisting of the sequence P, but with p_m at the top of the stack and p_1 at the bottom. A deque consisting of the sequence P is denoted by $P^{\diamond}$. Notice that $(p_1, p_2, \ldots, p_m)^{\blacktriangleleft} = (p_m, p_{m-1}, \ldots, p_1)^{\blacktriangleright}$. The size of a sequence P is denoted by $|P|$. Similarly, $|P^{\blacktriangleleft}|, |P^{\blacktriangleright}|$, and $|P^{\diamond}|$ are respectively the sizes of stacks $P^{\blacktriangleleft}, P^{\blacktriangleright}$, and deque $P^{\diamond}$.

2 Functional Queues with Good Real–Time Performance

Hood and Melville [17] describe a pure LISP implementation of FIFO queues with good amortized performance. They then modify it to get an implementation, also in pure LISP, with good real–time (*i.e.* worst–case) performance. Their idea is to represent a queue by two disjoint stacks, where the input stack $I^{\blacktriangleright}$ is used to receive the sequence of elements being pushed and the output stack $O^{\blacktriangleleft}$ is used to store the sequence of elements to be popped. We will denote the configuration of a queue as $\langle O^{\blacktriangleleft}, I^{\blacktriangleright} \rangle$. The sequence OI represents the entire sequence of elements of the queue. When the output stack becomes empty and a pop operation is executed, the sequence of elements in the input stack are all transferred to the output stack by reversing the sequence in the input stack to form a new output stack. The input stack is then replaced by an empty stack. That is, the configuration of the queue is transformed from $\langle ()^{\blacktriangleleft}, I^{\blacktriangleright} \rangle$ into $\langle I^{\blacktriangleleft}, ()^{\blacktriangleright} \rangle$. The transfer takes time linear to size of the input stack and can be implemented in a purely functional way.

Starting from an empty queue (with both the input and output stacks empty), a single–threaded sequence of s queue operations will transfer at most s elements from the input stack to the output stack. This is based on the observation that an element will be transferred at most once from the input stack to the output stack after it is pushed onto the queue. Since there are at most s push operations in a sequence of s queue operations, the total cost of transfer is bounded by s. This results in $O(s)/s = O(1)$ amortized cost per queue operation for the transfer, in addition to the $O(1)$ actual cost per queue operation for implementing the push and pop operations on the input and out stacks respectively.

However, the above implementation has two drawbacks. It is not real–time, and, more severely, does not suit multi–threaded applications, where pop or push operations may be performed not only on the newest version of a queue, but on previous versions as well. For example, let queue $P = \langle ()^{\blacktriangleleft}, (p_1, p_2, \ldots, p_m)^{\blacktriangleright} \rangle$ be a queue formed by a sequence of m push operations. A single pop operation on P will take $O(m)$ time to get element p_1 and form a new queue $Q = \langle (p_2, p_3, \ldots, p_m)^{\blacktriangleleft}, ()^{\blacktriangleright} \rangle$. Suppose that the next operation in the sequence is again a pop operation on P rather than on the newer version, Q. Then $O(m)$ time has to be spent again to get element p_1 and to form a queue identical to Q. Note that in general we have no way to tell whether or not two pop operations will be performed on the same queue.

Hood and Melville improve the above naive implementation by a simple idea, that the transfer of elements from the input stack to the output stack need not be carried out all at once when the output stack becomes empty. The transfer of elements can be carried out *incrementally* over a sequence

of queue operations whenever a substantial number of elements have been accumulated in the input stack, even when the output stack is not empty.

Suppose that a queue has the configuration $\langle O^{\blacktriangleleft}, I^{\blacktriangleright}\rangle$ and is to be transformed into the configuration $\langle (OI)^{\blacktriangleleft}, ()^{\blacktriangleright}\rangle$. Suppose also that the transfer of elements from the input stack $I^{\blacktriangleright}$ to the output stack $O^{\blacktriangleleft}$ is initiated only when the invariant $|O^{\blacktriangleleft}| \geq |I^{\blacktriangleright}|$ is violated. At the beginning of the transfer, we can assume $|O^{\blacktriangleleft}| = n$ and $|I^{\blacktriangleright}| = n+1$ for some $n \geq 0$. The new output stack $(OI)^{\blacktriangleleft}$ can be constructed in three stages: reverse stack $I^{\blacktriangleright}$ creating a stack $I^{\blacktriangleleft}$, reverse stack $O^{\blacktriangleleft}$ creating a stack $O^{\blacktriangleright}$, and then pop all the elements from stack $O^{\blacktriangleright}$ and push them onto stack $I^{\blacktriangleleft}$ one by one. The whole process will take $3n+4$ steps of time, where each step either moves the topmost element from one stack to another, or is a test (at the bottom of the recursion) to see if a stack is empty. Because the original output stack $O^{\blacktriangleleft}$ will become empty after n pop operations, we must accomplish the $3n+4$ steps in n regular queue operations. We then allocate 4 steps to the pop or push operation that initiates the transfer, and allocate 3 steps to each of the n queue operations that follows. Since each queue operation performs at most 4 extra steps to aid the construction of the new output stack, each queue operation takes $O(1)$ time and $O(1)$ space.

In the actual implementation, a normal queue is represented by two stacks, one for pop operations and one for push operations. However, an in–transition queue is represented by the old output stack (for pop operations), a new input stack (for receiving elements from push operations), and additional intermediate stacks for incremental construction of the new output stack. Once the new output stack is completely rebuilt, it is paired with the new input stack to represent a normal queue. The old output stack and those intermediate stacks are discarded. A normal queue enters the in–transition mode whenever its output stack becomes smaller than its input stack.

There are some remaining details in Hood and Melville's algorithm, though. In the construction of a new output stack, we need not copy all the elements in the original output stack $O^{\blacktriangleleft}$ because some of elements may have been popped during the course of the transfer. A counter is used to insure that we will not over–copy. Also, it can be shown that the queue is never empty during its transfer of elements, and after the transfer of elements, the invariant $|O^{\blacktriangleleft}| \geq |I^{\blacktriangleright}|$ still holds for the new input stack $I^{\blacktriangleright}$ and the new output stack $O^{\blacktriangleleft}$.

3 Functional Deques with Good Amortized Performance

A deque is a linear buffer where push and pop operations can be performed on either end of the buffer. We can modify the naive implementation of FIFO queues described in the above section to get a purely functional implementation of deque with good amortized performance. Similar schemes are also described by Sarnak [26] and Hoogerwoord [18].

A pair of stacks, $\langle L^{\blacktriangleleft}, R^{\blacktriangleright}\rangle$, is used to represent a deque, similar to that for a queue. The difference is that pop or push operations can be performed on stack $L^{\blacktriangleleft}$ or stack $R^{\blacktriangleright}$. We call $L^{\blacktriangleleft}$ the the left–hand–side (lhs) stack and $R^{\blacktriangleright}$ the right–hand–side (rhs) stack, instead of using the terms output stack and input stack previously used. Push and pop operations on the left side of the deque are performed on the lhs stack, and operations on the right side of the deque are performed on the rhs stack. The problem is that an lhs pop operation may be executed while the lhs stack is empty and the rhs stack contains some elements, and vice versa.

We solve this problem by transferring the bottom half of the rhs stack to the empty lhs stack, with the bottommost elements of the rhs stack becoming the topmost elements in the lhs stack after the transfer. That is, for an lhs pop operation on queue $\langle ()^{\blacktriangleleft}, R^{\blacktriangleright}\rangle$, we first transform the configuration of the queue from $\langle ()^{\blacktriangleleft}, R^{\blacktriangleright}\rangle$ to $\langle R_1{}^{\blacktriangleleft}, R_2{}^{\blacktriangleright}\rangle$, where $R = R_1R_2$ and $|R_1| = \lceil |R|/2 \rceil$. Then we perform an lhs pop operation on $\langle R_1{}^{\blacktriangleleft}, R_2{}^{\blacktriangleright}\rangle$. A symmetrical treatment also applies to an rhs pop operation when the rhs stack is empty but the lhs stack is not.

Does the above strategy provide a good implementation? If a sequence of deque operations is single–threaded and starts with an empty deque, then, by a simple credit–debit argument [30], it can be shown that each operation takes $O(1)$ amortized time and space. For the argument, we associate with a deque an imaginary "bank account" of units of time, whose balance reflects exactly the difference between the sizes of the two stacks in the the deque. That is, the balance will be $\big|\,|L^{\blacktriangleleft}| - |R^{\blacktriangleright}|\,\big|$ for a configuration of $\langle L^{\blacktriangleleft}, R^{\blacktriangleright}\rangle$. We also allocate two units of time for each deque operation, which will be the amortized cost of the operation.

A pop or push operation will change the difference between the sizes of the two stacks by one if it does not initiate the transfer process. For such an operation, one unit of time is used to pop/push an element from/to its corresponding stack, and the other unit of amortized time is deposited into the bank account if the pop or push operation increases the size difference. If the size difference is decreased by the push or pop operation, then one unit of time is withdrawn from the account and is discarded along with the other unit of time allocated for the deque operation. By doing so, we make sure that the bank account is consistent with the difference between the sizes of the two stacks.

On the other hand, an lhs pop operation on the deque $\langle ()^{\blacktriangleleft}, R^{\blacktriangleright}\rangle$ will have to transfer the bottom half of the rhs stack $R^{\blacktriangleright}$ to the empty lhs stack, before the pop operation can be performed. The transfer will require $|R^{\blacktriangleright}|$ units of time, which are withdrawn from the bank account. The bank account will have contained exactly $|R^{\blacktriangleright}|$ units before the transfer. After the transfer, the difference of the stack sizes is at most one (with the size of the lhs stack larger than or equal to the size of the rhs stack) and the bank account has a balance of zero. The lhs pop operation then proceeds as if the transfer had never happened. Inconsistency occurs after the transfer process if the stack size difference is one but the bank account balance is zero. In such a case, the lhs pop operation will decrease the size difference by one. However, it will not withdraw and discard one credit from the bank account this time, as would be required in the usual situations. That is, the bank account balance again reflects exactly the difference between the stack sizes after the lhs pop operation. A symmetrical analysis also applies to the case of rhs pop operations.

We conclude that $O(1)$ amortized time suffices for each deque operation in a single–threaded sequenced of operations. However, this implementation suffers the same drawbacks of the naive amortized implementation for FIFO queues in section 2. It is not real–time and does not suit multi–threaded applications. We will address these problems in the next section.

4 Functional Deques with Good Real-Time Performance

For an implementation of deques to be real-time and functional, the implementation must insure that each of the push or pop operations on either end of a deque is completed in $O(1)$ time in the worst case, and that the push or pop operation returns a new version of the deque without destroying the old version. This section describes such an implementation written in a purely functional language.

As in section 3, an lhs stack and an rhs stack together are used to represent a deque. The lhs pop or push operations are performed on the lhs stack, and the rhs pop or push operations are performed on the rhs stack. Problems arise when a pop operation encounters an empty stack. Our strategy is to make sure that such problems never arise. To able to do so, we maintain the following invariant between the two stacks:

$$|B| \geq |S| \geq 1, \text{and } 3|S| \geq |B| \qquad (1)$$

where B is the bigger stack of two stacks and S is the smaller stack. That is, each of the two stacks has at least one element, and the size of the bigger stack is never larger than three times the size of the smaller stack.

The above invariant can be violated in two ways. The first is if one of the stacks becomes empty due to a pop operation. However, if the invariant has been maintained all along, we know that the other stack has at most three elements. Thus the deque as a whole has at most three elements. At that point, the usual representation of the deque is replaced by a list of its elements. This list will contain at most three elements, and all push and pop operations afterward will be handled in an ad hoc way by looking at the head and tail of the list. If the size of the list grows to four by subsequent push operations, we then break the list into equally sized lhs and rhs stacks and usual representation is subsequently used. It is clear that each of the above ad hoc treatments consumes only a constant amount of time and space.

The other case in which the invariant can be violated is by a pop operation on the smaller stack, or by a push operation on the bigger stack, such that the size of the resulting bigger stack is larger than three times the size of the resulting smaller stack. Let S be the smaller stack, and B be the bigger stack, after the violating pop or push operation. If the invariant has been maintained all along and $|S| = m$, then $|B| = 3m + k$, where $m \geq 1$ and k is either 1, 2, or 3. Without loss of generality, we assume that the resulting deque has the configuration $\langle S, B \rangle$, with $S = (p_1, p_2, \ldots, p_m)^{\blacktriangleleft}$ and $B = (q_1, q_2, \ldots, q_{3m+k})^{\blacktriangleright}$, where k may be 1, 2, or 3. We then transfer the bottommost $m + 1$ elements of stack B to the bottom of stack S such that, after the transfer, we have two new stacks, $newS = (p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_{m+1})^{\blacktriangleleft}$ and $newB = (q_{m+2}, q_{m+3}, \ldots, q_{3m+k})^{\blacktriangleright}$, as the representation of the deque. It is clear that that transfer needs at most $O(m)$ time and space. We then distribute the transfer process evenly over the next m deque operations. Because each of the following m deque operations need only $O(m)/m = O(1)$ extra cost to rebuild the new stacks, each of of deque operations takes only constant time and consumes only constant space.

We now describe the details. The construction of the new stacks $newS$ and $newB$ can be accomplished by the following procedures:

a. Pop and reverse the topmost $2m + k - 1$ elements of stack B into an auxiliary stack $auxB$ such that $B = (q_1, q_2, \ldots, q_{m+1})^{\blacktriangleright}$ and $auxB = (q_{m+2}, q_{m+3}, \ldots, q_{3m+k})^{\blacktriangleleft}$.

b. Reverse stack S into an auxiliary stack $auxS$ such that $auxS = (p_1, p_2, \ldots, p_m)^{\blacktriangleright}$.

c. Reverse stack $auxB$ into a new stack $newB$ such that $newB = (q_{m+2}, q_{m+3}, \ldots, q_{3m+k})^{\blacktriangleright}$.

d. Reverse stack B into a new stack $newS$ such that $newS = (q_1, q_2, \ldots, q_{m+1})^{\blacktriangleleft}$.

e. Reverse stack $auxS$ onto stack $newS$ such that $newS = (p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_{m+1})^{\blacktriangleleft}$.

At the end of the transfer, stacks $newS$ and $newB$ will replace the roles of stack S and B, respectively, in the representation of the deque. Note that procedures a and b above can be carried out concurrently, which takes no more than $2m + 3$ steps. A step costs two units of time and space by moving one element from B to $auxB$ and moving one element from S to $auxS$. Similarly, procedure c can be carried out concurrently with d and e, taking a total of at most $2m + 3$ steps. In total, $4m + 6$ steps are sufficient to complete the transfer process. Since the transfer process will be distributed evenly over the next m deque operations, we allocate 6 steps to the deque operation that violates the invariant, and 4 steps to each of the m deque operations that follow.

During the course of the transfer, pop operations can be performed by the user on the original stacks S and B. Therefore, we must take care not to copy those discarded elements back onto the new stack $newS$ and $newB$ from the auxiliary stacks $auxS$ and $auxB$. A counter is used to keep track of the number of remaining elements in the original stack. Push operations during the course of transfer also cause problems. The elements pushed to stack S, for example, must be kept in a separate place $extraS$, to be annexed with $newS$ at the end of the transfer process to become the real new stack S. We cannot simply append $newS$ to $extraS$ for this will ruin the real-time performance at the very last step. Instead, we use the pair $\langle ExtraS, newS \rangle$ as the representation of the stack S when the transfer process is completed. This complicates pop and push operations on stacks. Pop and push operations on stack S have to examine $ExtraS$ first to see if it is empty. If it is not empty, then the pop or push operation is performed on it, otherwise the operation is performed on $newS$. But, in exchange, we maintain real-time performance. This implies that, in the normal representation of a deque as an lhs stack and an rhs stack, each of the two stacks is in fact a pair of lists in our implementation.

To summarize, we now describe below the high-level implementation of the deque operations. Only the lhs operations are described; the rhs cases are symmetrical to the lhs cases.

- $push_{lhs}$ (e, dq): Depending on the representation of dq, perform one of the following actions,
  - dq is a list of (less than 4) elements $\Longrightarrow$ "$Cons\ e\ dq$". If the resulting list has 4 elements, then split it in half into an lhs stack and an rhs stack.
  - dq is a pair of stacks, but is not currently transferring elements between the two stacks $\Longrightarrow$ Push e into the lhs stack. If the resulting lhs stack is

three times larger than the rhs stack, then initiate the transfer process.

- *dq* is a pair of stacks, and is transferring elements between them $\Longrightarrow$ Push e into the lhs *extra* list, then perform 4 incremental steps of the transfer process. If the transfer process is now complete, then pair–up the *extra* list with the *new* list to form a new stack, both for the lhs and the rhs.

- pop_{lhs} (*dq*): Depending on the representation of *dq*, perform one of the following actions,
  - *dq* is a list of (less than 4) elements $\Longrightarrow$ Return the "car" and "cdr" of the list.
  - *dq* is a pair of stacks, but is not currently transferring elements between the two stacks $\Longrightarrow$ Pop the lhs stack. If it results in an empty lhs stack, then the deque is now represented as a list of the elements in the rhs stack. If the rhs stack is three times larger than the resulting lhs stack, then initiate the transfer process.
  - *dq* is a pair of stacks, and is transferring elements between them $\Longrightarrow$ Pop the lhs *extra* list if it is not empty, otherwise pop the old lhs stack. Perform 4 incremental steps of the transfer process. If the transfer process is now complete, then pair–up the *extra* list with the *new* list to form a new stack, both for the lhs and the rhs.

- *new*: Return an empty list.

- *empty dq*: Return true if *dq* is an empty list, otherwise, return false.

The actual code, written in the purely functional subset of SML (of New Jersey), along with some annotations, is included in Appendix A.

Before we conclude this section, it remains to be shown that the invariant (1) is maintained after the transfer process. Suppose for the moment that the m deque operations that carry out the entire transfer process do not interfere with the rebuilding of the two new stacks. In such a case, the resulting deque will have one stack of size $2m+1$ and the other stack of size $2m+k-1$ in its representation, where k may be $1, 2$, or 3. The sizes of the two stacks will become most unbalanced if the m deque operations that carry out the transfer process all happen to be pop operations aimed at the smaller of the above two stacks. This occurs when $k = 1$ and the stack of size $2m+k-1$ are popped m times. The resulting smaller stack will have m elements, and the resulting bigger stack will have size $2m+1$. Since $3 \cdot m \geq 2m+1$ for all $m \geq 1$, the invariant (1) has been maintained.

5 Applications

Being able to add real–time deques to a purely functional language can lead to some interesting results. We show in this section two applications. The first is some implications in purely functional list processing. The other is a real–time simulation of a multihead Turing machine in a purely functional language.

5.1 Purely Functional List Processing

Here is a quiz: How much time will it take to reverse a list of n elements in a (sequential) purely functional language? A conventional method will take $O(n)$ time. But a list can be implemented, in a purely functional way, as a real–time deque without losing its functionality or its constant time performance. Furthermore, we can reverse a deque in constant time because of its symmetric nature. It turns out that a list of an arbitrary number of elements can be reversed in constant worst–case time if it is implemented as a real–time deque and an additional orientation tag is attached to the deque.

This gives us a new way to concatenate two lists. The usual method will take $O(|X|)$ time to concatenate a list Y to the rear of a list X. However, if lists X and Y are implemented as real–time deques with the lhs of the deque as the front of the list and the rhs as the rear, then their concatenation can be implemented in the following way: If $|Y| < |X|$, then pop elements from the lhs of Y and push them to the rhs of X; otherwise, pop elements from the rhs of X and push them to the lhs of Y. This takes $O(min(|X|, |Y|))$ worst–case time, which is better than the usual $O(|X|)$ time.

5.2 Real–Time Simulation of a Multihead Turing Machine in a Purely Functional Language

A multihead Turing machine is a Turing machine with a single two–way linear tape and multiple read/write heads upon the tape. In one move the multihead Turing machine, depending on the symbols scanned by the heads and the state of the finite control, changes its state and either prints a symbol on the cell scanned by one head or moves one head left or right one cell. Let Q be the finite set of states in the machine, Σ be the finite set of tape symbols, and K be the finite set of heads $\{h_1, h_2, \ldots, h_k\}$. The next move function δ is a mapping from $Q \times \Sigma^k$ to $Q \times K \times (\Sigma \cup \{L, R\})$, where L and R will move the head left and right one cell respectively. Notice that the machine can move or write using only one of its heads at a time. The machine starts with an initial state and an initial sequence of symbols on the tape. It proceeds according to the next move function, and halts if it reaches a final state. The resulting sequence of symbols on the tape is the output of the machine.

We are interested in multihead Turing machines because they provide a model for limited random access memory. A k–head Turing machine can be viewed as a linear memory machine with multiple program counters. These program counters differ from the program counter in a usual random access machine in that they cannot be arbitrarily reset to point to any memory locations. Only increment, decrement, read, and write instructions are available to manipulate these program counters. To compensate the loss of the arbitrary jump instruction, k program counters are made available such that the machine can memorize k different locations in the memory. It would be interesting to see how well the purely functional model (which assumes non–linear memory and admits no side–effects) can simulate this limited model of random access memory (which assumes linear memory and admits side–effects).

We now describe how to simulate in real–time a multihead Turing machine in a purely functional language. The contents of the tape and the positions of the heads can be implemented by a sequence of deques with each deque representing a segment of the tape delimited by its two surround-

ing heads. Let the entire contents of the tape be described by a sequence of symbols $S = (s_0, s_2, \ldots, s_n)$ with the k heads located at symbol $s_{l_i}, 1 \leq i \leq k$. Without loss of generality, we assume that $0 \leq l_i \leq l_{i+1} \leq n$ for all $1 \leq i \leq k-1$. Recall that $S^\diamond$ denotes the deque containing exactly the elements of sequence S, with elements s_0 and s_n at the two ends of the deque. The tape, along with the head positions, can be represented by a sequence of deques $(D_0, D_1, \ldots, D_k)$, where $D_0 = (s_0, s_1, \ldots s_{l_1-1})^\diamond$, $D_i = (s_{l_i}, s_{l_i+1}, \ldots s_{l_{i+1}-1})^\diamond$ for $1 \leq i \leq k-1$, and $D_k = (s_{l_k}, s_{l_k+1}, \ldots s_n)^\diamond$. Because several heads may be positioned over the same cell on the tape, some of the deques may be empty. Also, head h_i, the ith head in K, need not be positioned over symbol s_{l_i}. Rather, there is a permutation function f over $\{1, 2, \ldots k\}$ such that head h_i will position at symbol $s_{l_{f(i)}}$. It is clear that every move of a multihead Turing machine (printing a symbol under a head, moving a head left or right one cell), can be accomplished in real-time by performing the operations at the corresponding deques.

Given the next move function δ from $Q \times \Sigma^k$ to $Q \times K \times (\Sigma \cup \{L, R\})$ and the representation of the tape described above, we can implement in a purely functional way a next move function Δ from $Q \times \mathit{Tape}$ to $Q \times \mathit{Tape}$ such that each function application of Δ costs $O(k)$ time and space. *Tape* is the data type that describes exactly the current tape configuration of the machine, including the the contents of the tape, the positions of the heads, and the permutation function f. The permutation function f is needed because if two heads cross each other on the tape, their ordering on the tape will change and the permutation function f must be changed accordingly. This results in a real-time simulation in a purely functional language of a multihead Turing machine, where each move of the machine is accomplished in $O(k)$ time and space. Note that the complexity of the simulation, $O(k)$ per move, depends only on k, the number of heads, not on n, the number of symbols on the tape. Also note that every configuration of the execution history of a multihead Turing machine is equally accessible in our real-time simulation. This means that our real-time simulation can be multi-threaded, which is very useful if we want to simulate a nondeterministic multihead Turing machine where there may be several eligible next moves at any given machine configuration.

6 Remarks

Throughout our discussion in this paper, we implicitly assume that the purely functional language used to implement the deques is a strict language rather than non-strict language in which the evaluation of function arguments are delayed until needed. A strict language is required because the incremental nature of the algorithm requires that elements be transferred between stacks well in advance of the use of those elements. A non-strict language would delay most of the transfer steps when a deque operation occurs. However, a subsequent deque operation (such as a popping an element and performing a strict operation on the element) would cause many of the delayed transfer steps to execute. Naturally, this would destroy the real-time nature of the algorithm.

Are the data structures described in this paper still attractive if implemented in a non-strict language or implemented within a system that does not provide real-time garbage collection? Even though the resulting program will not have real-time properties, it is still important to have efficient implementations of aggregate data structures, such as deques, which are suitable for multi-threaded applications. Using a non-strict functional language, the performance of the deques described in section 4 will degrade to $O(1)$ amortized time, instead of $O(1)$ real-time, per deque operation in a multi-threaded program. This is better than the naive implementation of deques described in section 3, which achieves $O(1)$ amortized complexity only in single-threaded programs.

7 Related Work and Future Work

A real-time simulation of a multihead Turing machine in a purely functional language does not come as a big surprise. Fischer, Meyer, and Rosenberg [13] and Leong and Seiferas [21] has shown that a multihead Turing machine can be simulated in real-time by a multitape Turing machine with only one head per tape. A single tape, single head Turing machine can be simulated in real-time in a purely functional language, by two stacks and a finite control. Thus, it is clear that a multihead Turing machine can be simulated in real-time in a purely functional language. Nevertheless, the simulations of [13] and [21] are complicated, and their resulting simulations of a multihead Turing machine in a purely functional language will probably not be as simple as ours. Furthermore, our objective is to demonstrate that purely functional languages can be used to implement non-trivial aggregate data structures, such as deques and multihead Turing machines, efficiently and straightforwardly, which is different from the objectives of [13] and [21].

In our purely functional implementation of real-time deques, two techniques are used: a deque is represented as two balanced stacks, and incremental method is used to balance the two stacks. Incremental methods have been used in many contexts to improve algorithms of amortized performance to worst-case performance. Baker's algorithm for real-time garbage collection is a good example [3]. Baker took a copying garbage collection algorithm which had $O(1)$ amortized cost per storage allocation (this cost includes the high cost of the stop-and-copy collection) and modified it to be $O(1)$ worst-case cost.

Both Sarnak [26] and Hoogerwoord [18] use two stacks to represent a deque. Sarnak's design of fully persistent real-time deques is not purely functional because side-effects are used. His scheme is also more complicated than ours. Hoogerwoord's design is purely functional, but not real-time, and is not suitable for multi-threaded applications. While not aware of the above two works, we start with the result of Hood and Melville [17] on purely functional real-time queues, and extend it to purely functional real-time deques. The underlying idea, that a deque is represented as two balanced stacks, is the same.

An open problem is whether deque concatenation can also be implemented in real-time in a purely functional way, in addition to the real-time pop and push operations. Buchsbaum and Tarjan [9] recently show that deques can be made confluently persistent (*i.e.*, concatenateable) with constant cost for each concatenation and push operation, but at the cost of $O(\log^* n)$ for each pop operation, where n

is the deque size.[1] All costs are amortized. This improves a previous result of Driscoll, Sleator, and Tarjan [12]. Using our purely functional implementation of real-time deques, their implementation can be made purely functional, with amortized complexity improved to worst-case complexity for each operation [8]. It would be interesting to see if a purely functional implementation of deques exists such that each concatenation, pop, and push operation costs only constant worst-case time.

8 Acknowledgments

We would like to thank Adam Buchsbaum for pointing to us the deque results in Sarnak's Ph.D. thesis, which we were not aware of, after reading a draft of this paper. We also thank him and the referees for their comments.

This research has been supported, in part, by the National Science Foundation (#CCR-8909634) and DARPA (DARPA/ONR #N00014-92-J1719). The first author is also supported in part by a Dean's dissertation fellowship from the Graduate School of Arts and Science, New York University, during the 1992-93 academic year.

A SML code with Annotations

The SML signatures for the stack and deque structures are described in Figure 1. An implementation of the stack structure is described in Figure 2, where each operation, including pack and unpack, costs $O(1)$ time. An implementation of real-time deques is described in two parts. Figure 3 describes the local declarations of the deque functor. It includes the definition of states according to the transfer procedures described in section 4, and the functions to manipulate them. The actual implementations of deque operations are given in Figure 4.

References

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structure for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

[5] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, August 1991.

[1] The function $\log^*$ is defined for every integer $n \geq 0$ such that

$$\begin{cases} \log^* n = 0 & \text{if } n = 0, \\ \log^* n = 1 + \log^* \lceil \log n \rceil & \text{otherwise.} \end{cases}$$

Intuitively, $\log^* n$ is the number of repeated logarithm operations to take n to 0. For example, $\log^* 1 = 1, \log^* 2 = 2$, and $\log^* 2^{2^{2^2}} = 5$.

[6] Adrienne Gael Bloss. *Path Analysis and the Optimization of Non-Strict Functional Languages.* PhD thesis, Department of Computer Science, Yale University, May 1989. Also appears as report YALEU/DCS/RR-704.

[7] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Functional Programming Languages and Computer Architecture*, pages 26–38. ACM/Addison-Wesley, September 1989. Imperial College, London, UK.

[8] Adam L. Buchsbaum. Private communication, 1993.

[9] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 155–164. Austin, Texas, USA, January 1993.

[10] Tyng-Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In Bernd Krieg-Brückner, editor, *4th European Symposium on Programming*, pages 110–129. Rennes, France, February 1992. Lecture Notes in Computer Science, Volume 582, Springer-Verlag.

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

[12] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 89–99, January 1991. San Francisco, California, USA.

[13] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Real-time simulation of multihead tape units. *Journal of the Association for Computing Machinery*, 19(4):590–607, October 1972.

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

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

[16] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of 5th Annual IEEE Symposium on Logic in Computer Science*, pages 333–343. IEEE, June 1990.

[17] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, 1981.

[18] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.

[19] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 300–314. ACM, January 1985. New Orleans, Louisiana, USA.

[20] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84. Charleston, South Carolina, USA, ACM, January 1993.

[21] Benton L. Leong and Joel I. Seiferas. New real-time simulations of multihead tape units. *Journal of the Association for Computing Machinery*, 28(1):166–180, January 1981.

[22] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.

[23] Eugene W. Myers. Efficient applicative data types. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75. ACM, January 1984. Salt Lake City, Utah, USA.

[24] Martin Odersky. How to make destructive updates less destructive. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 25–36. Orlando, Florida, USA, ACM, January 1991.

[25] Carl G. Ponder, Patrick C. McGeer, and Antony P.-C. Ng. Are applicative languages inefficient? *SIGPLAN Notices*, 23(6):135–139, June 1988.

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

[27] Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis, Deaprtment of Mathematics and Computing Science, Eindhoven University of Technology, September 1992.

[28] J. T. Schwartz. Optimization of very high level languages — i. value transmission and its corollaries. *Computer Languages*, 1(2):161–194, June 1975.

[29] J. T. Schwartz. Optimization of very high level languages — ii. deducing relationships of inclusion and membership. *Computer Languages*, 1(3):197–218, September 1975.

[30] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[31] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. Nice, France, ACM, June 1990.

[32] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluation and and Semantics-Based Program Manipulation*, pages 255–273. New Haven, Connecticut, USA, ACM, June 1991. The proceeding appears as *SIGPLAN Notices*, 26(9), September 1991.

```
signature STACK =
sig
    exception Empty

    type 'a Stack

    val new   : 'a Stack
    val empty : 'a Stack -> bool
    val push  : 'a -> 'a Stack -> 'a Stack
    val pop   : 'a Stack -> 'a * 'a Stack

    val pack  : 'a list -> 'a list -> 'a Stack
    val unpack : 'a Stack -> 'a list * 'a list
end


signature DEQUE =
sig
    exception Empty

    datatype Side = LHS | RHS

    type 'a Deque

    val new    : 'a Deque
    val empty  : 'a Deque -> bool
    val push   : Side -> 'a -> 'a Deque -> 'a Deque
    val pop    : Side -> 'a Deque -> 'a * 'a Deque
    val length : 'a Deque -> int
end
```

Figure 1: The signatures of the stack and deque structures in SML.
NOTE. The objective is to implement each of their operations in constant time and constant space. Note that we will implement a stack by a pair of lists, by using the pack and unpack functions. The reason for this is stated in section 4.

```
functor Stack () : STACK =
struct
    exception Empty

    type 'a Stack = 'a list * 'a list

    val new = ([], [])
    fun empty ([], []) = true
      | empty _        = false

    fun push e (x,     y)     =     (e::x, y)
    fun pop    (x::xs, ys)    = (x, (xs,   ys))
      | pop    ([],    y::ys) = (y, (ys,   []))
      | pop    ([],    [])    = raise Empty

    fun   pack x y  = (x, y)
    fun unpack pair = pair
end
```

Figure 2: SML code for stacks.

```
functor Deque (stack : STACK) : DEQUE =
struct
local
    open stack

    type 'a Current = 'a list * int * 'a Stack * int

    datatype 'a State = NORM  of 'a Stack * int
                      | RevB  of 'a Current * 'a Stack * 'a list * int
                      | RevS1 of 'a Current * 'a Stack * 'a list
                      | RevS2 of 'a Current * 'a list * 'a Stack * 'a list * int
                      | COPY  of 'a Current * 'a list * 'a list * int

    fun head stack = let  val (element, _) = pop stack  in  element  end
    fun tail stack = let  val (_, stack)   = pop stack  in  stack    end

    fun put e (extra, added, old, remained) = (e::extra, added+1, old, remained)
    fun get   ([],    added, old, remained) = (head old, ([], added,   tail old, remained-1))
      | get   (e::es, added, old, remained) = (e,        (es, added-1, old,      remained))

    fun top current = let  val (element, _) = get current  in  element  end
    fun bot current = let  val (_, current) = get current  in  current  end

    fun normalize (state as COPY ((extra, added, _, remained), _, new, moved)) =
        if moved = remained
           then NORM (pack extra new, added + moved)
           else state
      | normalize state = state

    fun tick state =
        case state of
             NORM _ => state
           | RevB (current, Big, auxB, count) =>
             RevB (current, tail Big, (head Big)::auxB, count-1)
           | RevS1 (current, Small, auxS) =>
             if empty Small
                then state
                else RevS1 (current, tail Small, (head Small)::auxS)
           | RevS2 (current, auxS, Big, newS, count) =>
             if empty Big
                then normalize (COPY  (current, auxS, newS, count))
                else RevS2 (current, auxS, tail Big, (head Big)::newS, count+1)
           | COPY (current as (_, _, _, remained), aux, new, moved) =>
             if moved < remained
                then normalize (COPY (current, tl aux, (hd aux)::new, moved+1))
                else normalize state

    fun ticks (RevB (currentB, Big, auxB, 0), RevS1 (currentS, _, auxS)) =
              (normalize (COPY (currentB, auxB, [], 0)), RevS2 (currentS, auxS, Big, [], 0))
      | ticks (RevS1 (currentS, _, auxS), RevB (currentB, Big, auxB, 0)) =
              (RevS2 (currentS, auxS, Big, [], 0), normalize (COPY (currentB, auxB, [], 0)))
      | ticks (lhs, rhs) = (tick lhs, tick rhs)

    fun steps 0 pair = pair
      | steps n pair = steps (n - 1) (ticks pair)
in
```

Figure 3: SML code for real-time deques, Part 1.

NOTE. Datatype Current is used to hold the "current" stack when elements are being transferred between the two stacks of a deque. It has four fields: the list of newly pushed elements and its count, and the old stack and its remaining count. Datatype State has five states whose functions are described in section 4. They are NORM (when no element is being transferred), RevB (for performing procedure a), RevS1 (for procedure b), RevS2 (for procedure d), and COPY (for procedure c or e). Functions put and get perform push and pop operations, respectively, on datatype Current. State transitions are carried out by function tick, which is called by ticks, which is again called by steps.

```
    exception Empty

    datatype    Side  = LHS | RHS
    datatype 'a Deque = LIST of 'a list
                      | PAIR of 'a State * 'a State

    val new = LIST []
    fun empty (LIST []) = true
      | empty _         = false

    fun swap (LIST l)          = LIST (rev l)
      | swap (PAIR (lhs, rhs)) = PAIR (rhs, lhs)

    fun pop' (NORM  (a, b))          = (head a, NORM  (tail a, b-1))
      | pop' (RevB  (a, b, c, d))    = (top  a, RevB  (bot  a, b, c, d))
      | pop' (RevS1 (a, b, c))       = (top  a, RevS1 (bot  a, b, c))
      | pop' (RevS2 (a, b, c, d, e)) = (top  a, RevS2 (bot  a, b, c, d, e))
      | pop' (COPY  (a, b, c, d))    = (top  a, COPY  (bot  a, b, c, d))

    fun pop _   (LIST []) = raise Empty
      | pop LHS (LIST l)  = (hd l, LIST (tl l))
      | pop LHS (PAIR (NORM (L, l), rhs as NORM (R, r))) =
        let  val (h, L) = stack.pop L  in
        if 3*(l-1) >= r
           then (h, PAIR (NORM (L, l-1), rhs))
           else if l >= 2
                  then (h, PAIR (steps 6 (RevS1 (([], 0, L, 2*l-1), L, []),
                                          RevB  (([], 0, R, r-l),   R, [], r-l))))
                  else (h, LIST (rev ((op @) (unpack R))))  end
      | pop LHS (PAIR (L, R)) = let  val (e, L) = pop' L  in  (e, PAIR (steps 4 (L, R)))  end
      | pop RHS deque         = let  val (e, deque) = pop LHS (swap deque)  in  (e, swap deque)  end

    fun push' z (NORM  (a, b))          = NORM  (push z a, b+1)
      | push' z (RevB  (a, b, c, d))    = RevB  (put  z a, b, c, d)
      | push' z (RevS1 (a, b, c))       = RevS1 (put  z a, b, c)
      | push' z (RevS2 (a, b, c, d, e)) = RevS2 (put  z a, b, c, d, e)
      | push' z (COPY  (a, b, c, d))    = COPY  (put  z a, b, c, d)

    fun push LHS e (LIST l) =
        if length l <= 2
           then LIST (e::l)
           else PAIR (NORM (pack [e,              hd l]      [], 2),
                      NORM (pack [hd (tl (tl l)), hd (tl l)] [], 2))
      | push LHS e (PAIR (NORM (L, l), rhs as NORM (R, r))) =
        let  val L = stack.push e L  in
        if 3*r >= l+1
           then PAIR (NORM (L, l+1), rhs)
           else PAIR (steps 6 (RevB  (([], 0, L, l-r),   L, [], l-r),
                               RevS1 (([], 0, R, 2*r+1), R, [])))  end
      | push LHS e (PAIR (L, R)) = PAIR (steps 4 (push' e L, R))
      | push RHS e deque         = swap (push LHS e (swap deque))

    fun length' (NORM  (_, l))                    = l
      | length' (RevB  ((_, a, _, r), _, _, _))   = a + r
      | length' (RevS1 ((_, a, _, r), _, _))      = a + r
      | length' (RevS2 ((_, a, _, r), _, _, _, _)) = a + r
      | length' (COPY  ((_, a, _, r), _, _, _))   = a + r

    fun length (LIST l)      = List.length l
      | length (PAIR (L, R)) = length' L + length' R
end
end
```

Figure 4: SML code for real-time deques, Part 2.

NOTE. A Deque datatype is either a list of (less than 4) elements, or a pair of states representing the lhs and rhs stacks. Function swap exchanges the two sides of a deque. As described in section 4, functions pop and push will initiate the transfer of elements between the two stacks of a deque iff the resulting bigger stack is more than three times the size of the resulting smaller stack.