universität
innsbruck

Seminar Report

Purely Functional Real-Time Deques

Research Seminar Logic & Learning

Fabian Schaub
Supervisor: René Thiemann

16 February 2023

Abstract

Deques are a very useful and versatile data structure. In imperative programming they can be implemented as doubly linked lists. This yields a deque with constant worst case complexity for the operations *push, pop, inject* and *eject.* Also constant time catenation of two different doubly linked lists is possible. However, in purely functional programming we cannot make use of such an implementation. By only using *algebraic data types* the task of a deque with constant worst case complexities for the mentioned operations becomes non-trivial. In this report we describe and present an implementation of a purely functional real-time deque as defined by *Kaplan and Tarjan et al.* [5].

1 Introduction

A deque (short for **d**ouble **e**nded **que**ue) is a data structure representing a sequence of elements. It supports at least four operations: `push`, `pop`, `inject` and `eject`. `push` and `pop` add and remove an element at the front of the deque, respectively. Likewise `inject` and `eject` add and remove an element at the back of the deque, respectively. We can go on and define catenation of deques in terms of pop and inject, or eject and push.

An example use-case for a deque is a string-builder. String-builders are commonly used to build up large strings from sub-strings because strings are often immutable in programming languages. A natural choice for an implementation of a string-builder is a deque over strings. Relative to simply catenating strings, using such a string-builder and extracting the resulting string in the end is very efficient, provided the underlying deque operations are efficient.

From this example we can see that an implementation of a deque with constant time complexity is of great interest. Moreover, in this report we are interested in a deque data structure with constant worst case complexity that is purely functional.

In the paper *purely functional real-time deques with catenation* [5] three data structures are presented: *real-time deques*, *real-time steques with catenation* and *real-time deques with catenation*. We consider *real-time deques* in this report and give a short outlook to the other two in the end.

In the following we reference related work and present the background of our work. Going on, we present the data structure from [5] and our implementation of it. In the end we show some results and finally we give an outlook to further work.

2 Related Work

Several purely functional implementations of deques achieved constant amortized complexity, which can be found in papers by Hood [2], Gajewska [1] and Hoogerwoord [3]. The implementation presented by *Kaplan and Tarjan et al.* in *Purely functional, real-time deques with catenation* [5] extends the previous implementations and achieves constant worst case complexity. They only present a textual explanation of the data structure and algorithm however. Our contribution is a pure implementation of data structure and operations in *Haskell* [4] using *algebraic data types*.

3 Background

As previously mentioned, we are in a purely functional setting. This means that our data structure has to be persistent [6], i.e. a modification of the data structure must not destroy the previous version. We consider *algebraic data types* (ADTs) as they are persistent and native to our implementation language.

```
   ([1,2] ,  •  , [13,14,15])
             ↓
([(3, 4)] ,  •  , [(5, 6),(7, 8),(9, 10),(11, 12)])
             ↓
             ∅
```

Figure 1: A non-optimized deque representing the sequence 1 to 15. The top level deque has buffers over singleton elements and its child buffers over pairs of elements. the bottom of the deque is denoted as $\emptyset$.

The deque supports the operations `push`, `pop`, `inject` and `eject`. For these we need constant time complexity. Consider an implementation as a pair of lists [2, 1, 3]. This implementation typically uses a list splitting and reversal scheme. Amortized complexity analysis indeed gives us constant time complexity for all deque operations. Yet, in the worst case we have to perform list splitting and reversing witch takes linear time. The deque implemented in this work achieves real-time performance, meaning all operations have a worst case complexity of $O(1)$.

4 A Purely Functional Real-Time Deque

Kaplan et al. state two key ideas to implement a purely functional real-time deque. First is to use the idea from the pair of lists mentioned in Section 3. Second is not to wait to do a list splitting and reversal, but rather to move elements inside the deque at every operation. This way the balancing of the deque, which is the essence of the list splitting and reversal scheme, is spread over all deque operations. More precisely, after each deque operation we balance the deque in a constant number of steps. As long as we retain a balanced deque we can perform all deque operations on it efficiently.

This balanced state is subsumed by *Kaplan's* definition of *regularity* and *semiregularity*. Informally, a *regular* deque is balanced and a *semiregular* deque is slightly imbalanced, such that it can be balanced in a constant number of steps.

In the following subsections we present the data structure, ideas and algorithms from [5] in Subsections 4.1, 4.2 and 4.3 and then show parts of our implementation in Section 4.4. The full implementation can be found in the Appendix.

4.1 Data Structure

As in the *pair-of-lists* approach mentioned in Section 3 the deque consists of two such lists called *buffers*. The front buffer is called *prefix* and the tail buffer is called *suffix*. Buffers are constrained to hold at most 5 elements. Since buffers themselves are deques implemented by lists this is needed to achieve real-time performance. More precisely, all deque operations on buffers are real-time since each buffer has an upper bound on its size.

The *real-time deque* is then a triple of prefix, suffix and a child deque over pairs of elements (Figure 1). We call the deques in this chain of descendants levels, where level $i+1$ is the descendant of level i. The idea behind the pairing of elements is a recursive slowdown when moving data inside the deque structure. Consider the levels i and $i+1$. For every move operation on level i we move n elements. Subsequently, for every move operation on level $i+1$ we move $2n$ elements. This means for x deque operations we have to perform up to $\frac{x}{2^{i+1}}$ move operations for any level i to balance the structure.

4.2 Regularity

For an efficient procedure to move data inside the deque structure, we first define a coloring of the buffers and the deque. The colors are *red*, *green* and *yellow* and we order them as $red < yellow < green$. The coloring indicates how many deque operations can be performed on an object. In the worst case, we cannot apply a deque operation on a *red* object. On a *yellow* object we apply at least one deque operation and on a *green* object we can apply at least two deque operations.

Buffers are then colored

- *red* if they contain 0 or 5 elements,
- *yellow* if they contain 1 or 4 elements or
- *green* if they contain 2 or 3 elements.

The number of operations applicable to a deque is the number of operations applicable to the „worse“ buffer. An exception occurs on the bottommost deque. For this deque we can move things between prefix and suffix without violating the element order. This means that if one of the buffers is empty the number of operations applicable to the bottommost deque is the number of operations applicable to the non-empty buffer. In conclusion, the coloring of a deque is defined as:

$$\texttt{color d} = \begin{cases} \texttt{color (prefix d)} & \text{if } \texttt{child d} \text{ and } \texttt{suffix d} \text{ are empty} \\ \texttt{color (suffix d)} & \text{if } \texttt{child d} \text{ and } \texttt{prefix d} \text{ are empty} \\ & \\ \texttt{min (color (prefix d)) (color (suffix d))} & \text{otherwise} \end{cases}$$

An example of a deque coloring can be seen in Figure 2. Note that Level 1 in this example would not be red if its suffix was empty, since it is the bottommost deque.

A deque is *semi-regular* if for all levels i and j, if level i and j are red then there exists a level k that is green and between i and j. More formally, let $child^n(d)$ denote level n

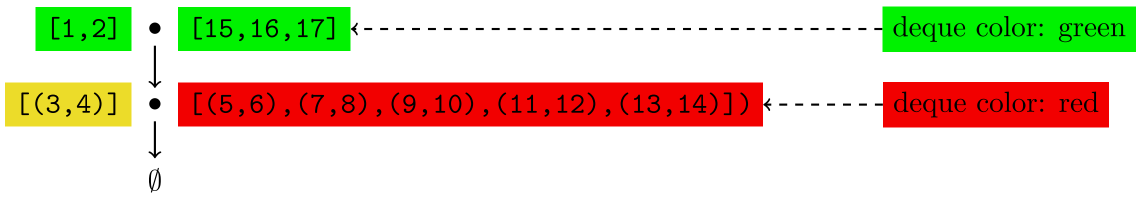

Figure 2: An example of a colored deque. Level 0 is green since both its buffers are green. Level 1 is red because its suffix is full.

in deque d. Semi-regularity is then defined as:

$$d \text{ is semi-regular}$$
$$\Updownarrow$$
$$\forall i, j.\ i < j \rightarrow \mathit{child}^i(d) \text{ and } \mathit{child}^j(d) \text{ are red} \rightarrow \exists k. i < k < j \wedge \mathit{child}^k(d) \text{ is green}$$

The idea behind semi-regularity is that any red deque can be transformed into a green one if we can move elements down or up in the deque structure. This is only the case if the descendant deque is not red. Furthermore, if there is no green descendant deque before the next red deque, such a transformation could lead to non-semi-regularity. This is why a green deque that interrupts a red chain is needed.

A deque is then *regular* if it is semi-regular and the first non-yellow deque from the top is green.

Example 4.1. Consider the color sequence of deque:

- *green* $\rightarrow$ *yellow* $\rightarrow$ *green* is regular.
- *yellow* $\rightarrow$ *yellow* $\rightarrow$ *yellow* is regular.
- *green* $\rightarrow$ *yellow* $\rightarrow$ *red* is regular.
- *red* $\rightarrow$ *green* is semi-regular. It is not regular because it starts with a red deque.
- *red* $\rightarrow$ *yellow* $\rightarrow$ *red* is not semi-regular because there is no green deque between the red deques.

Example 4.2. The deques from Figure 2, 1 and 3 are all regular.

A deque being regular means that we can always perform at least one deque operation on the top level of the deque. Furthermore, the topmost red deque is eventually transformed into a green one before we need to perform an operation on it. Consider for example a regular deque on which we push elements. Assume a topmost red level i that has a prefix buffer with 5 elements. i is not violating the regularity constraint as long as there is a

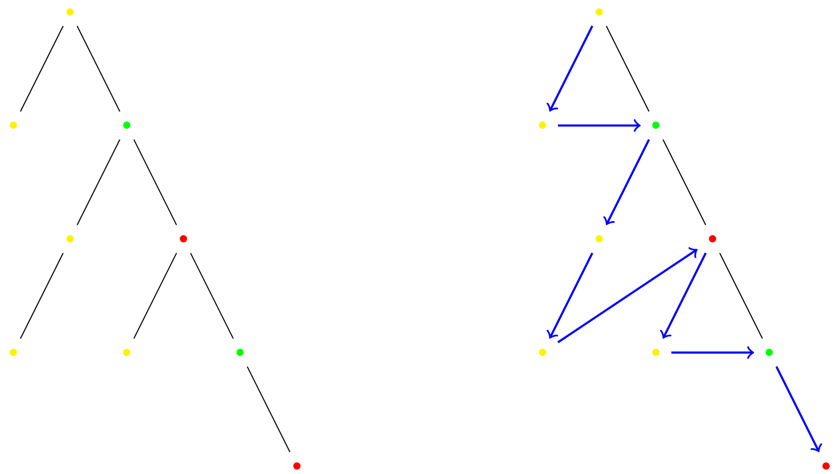

Figure 3: **Left:** The optimized deque data structure. The left sub tree are the immediate yellow descendants, the right child node the non-yellow descendant.
Right: The inorder traversal over the optimized data structure.

green level above it. As we push elements onto the deque level i eventually becomes the topmost non-yellow deque. At this point i violates the regularity constraint and elements are moved down from its prefix to make i green. Since i has now again free capacity, any regularization that is needed above i (which may need to modify level i) is possible. The details of the regularization procedure is explained in Subsection 4.3.

An important observation is that a deque operation on a regular deque results in a regular or a semi-regular deque. Subsequently, if we regularize after each deque operation we can retain a regular deque.

At this point we see that we need to traverse to the topmost non-yellow level for our regularization. A deque structure as depicted in Figure 1 poses a problem for this. We would need to traverse the chain of descendants which has a complexity of $O(log(n))$. Luckily, a structural optimization mitigates this problem. As seen in Figure 3 we can use a tree structure where the left sub-tree is the chain of immediate yellow descendants and the right child is the next non-yellow deque. Using this structure, the top-most non-yellow deque is the root or its right child. Moving on, we will disregard this optimization as it does not influence the principle of the regularization procedure.

4.3 Regularization

As previously mentioned we need to restore regularity of a semi-regular deque in constantly many steps after each deque operation. This is done by changing the top most red deque to a green deque. With the optimization of Figure 3 we can find this red deque within one or two steps.

Let level i be the topmost red deque and level $i+1$ its child. Furthermore, let P_i and S_i denote the prefix and the suffix of level i, respectively. Likewise, P_{i+1} and S_{i+1} denote the prefix and suffix of level $i+1$. We distinguish three cases based on how many elements are in each of these buffers. In the following we will call buffers with one or zero elements *underflowing* and buffers with four or five elements *overflowing*.

Two-Buffer-Case ($|P_{i+1}| + |S_{i+1}| \geq 2$): this case occurs if level $i+1$ contains at least two elements so we can move elements down or up to regularize. If $i+1$ is the bottommost level we move elements between P_{i+1} and S_{i+1}, such that both are non-empty. Next we move elements down if P_i or S_i or both are overflowing. In this case level $i+1$ contains at least two elements, enough to move elements up if P_i, S_i or both are underflowing. Finally, if level $i+1$ is empty we delete it.

One-Buffer-Case ($|P_{i+1}| + |S_{i+1}| \leq 1 \ \wedge \ (|P_i| \geq 2 \ \vee \ |S_i| \geq 2)$): this case occurs solely at the bottom of the deque. It occurs if level $i+1$ contains at most one element and at least one of P_i and S_i contains two or more elements. We handle this case similar as the Two-Buffer-Case, but we use only P_{i+1} on level $i+1$. We move the element on level $i+1$ to P_{i+1} if it exists. Next we move elements down to P_{i+1} from overflowing buffers on level i. Then we move elements up from P_{i+1} to underflowing buffers on level i. Finally, we delete level $i+1$ if it is empty.

No-Buffer-Case ($|P_{i+1}| + |S_{i+1}| \leq 1 \ \wedge \ |P_i| \leq 1 \ \wedge \ |S_i| \leq 1$): this case also occurs solely at the bottom of the deque. In contrast to the One-Buffer-Case, here P_i and S_i are both underflowing. This means we have at most three level-i-elements in total. We move all the elements to P_i and delete level $i+1$.

After this procedure level i is green and regularity is restored.

Example 4.3. Consider the deque d in Figure 2. The coloring is $\mathit{color}(d) = \mathit{green}$ and $\mathit{color}(\mathit{child}(d)) = \mathit{red}$, so d is regular. Assume we inject an element 16 into d. Then the top suffix gets yellow which makes the top level yellow. Now the topmost non-yellow level is level 1, which is red. Subsequently $\mathit{inject}(d, 16)$ is semi-regular. According to the regularization procedure we are in the One-Buffer-Case. Regularization now moves two elements from the suffix of level 1 to the prefix of level 1, which results in two green buffers and a green level 1. In the end we obtain $d' = \mathit{regularize}(\mathit{inject}(d, 16))$. Its coloring is $\mathit{color}(d') = \mathit{yellow}$ and $\mathit{color}(\mathit{child}(d')) = \mathit{green}$, which means d' is regular.

4.4 Implementation

There are several data types which can be used as a deque. For this reason `Deque` is a type class in our implementation. Other deque operations which are not class functions like `append` and `fromList` are implemented on top of these defined class functions.

```haskell
-- | a deque as a class.
class Deque (a :: * -> *) where
  push :: b -> a b -> a b
  pop :: a b -> (b, a b)
  inject :: b -> a b -> a b
  eject :: a b -> (b, a b)
  null :: a b -> Bool
  empty :: a b
```

As the basic container of elements we use lists to realize buffers. A buffer is an instance of the `Deque` type class since it can act as one and indeed is used as one in this implementation. Note that buffers may contain at most five elements. This detail is not captured in the buffer type but asserted at runtime.

```haskell
type Buffer = []
instance Deque Buffer where
  ...
```

Our real-time deque `RTDeque` is implemented as a record type. Due to the structural optimization (Figure 3) we have to make a slight modification to the data type. Since we do not know how many immediate yellow descendants there are, we also do not know the degree of pairing in the non-yellow branch. In Haskell we have no means to encode this relationship, so we have to default to arbitrary nesting. For this reason we use buffers of balanced binary trees and do not encode the pairing in the type. This means the type `RTDeque a` does not contain information about the degree of pairing, nor does its constructors contain information about the correctness of the pairing.

Also note that we have an additional constructor `Nil` which is the empty bottom of a deque. In contrast to a deque with empty buffers `Nil` does not denote an underflowing deque but is really the bottom element.

```haskell
data RTDeque a
  = Nil
  | RTDeque
    { _prefix :: Buffer (BTree a)
    , _suffix :: Buffer (BTree a)
    , _yellows :: RTDeque a      -- ^the yellow descendants.
                                 --  these must not contain nonyellow branches.
    , _nonyellows :: RTDeque a   -- ^the nonyellow descendants.
    }
```

The notion of colors is captured in our implementation as the data type `Color` and the type class `Colored`. `Buffer` and `RTDeque` are instances of the `Colored` class.

```haskell
data Color = Red | Yellow | Green
  deriving (Ord, Eq, Show)
```

```haskell
class Colored (a :: *) where
  color :: a -> Color

instance Colored Buffer where
  ...
instance Colored RTDeque where
  ...
```

The function `restoreRegularity` uses this data definitions to implement the regularization procedure. In the source code are also checks of *regularity* and *semi-regularity* implemented. These are only for completeness and not used in the deque operations.

`restoreRegularity` is implemented as defined in the paper and is not presented at this point because it is quite long and implements the procedure shown in Subsection 4.3 or initially presented in [5]. Interested readers can find it in the Appendix. An important observation is that the regularization procedure does not contain recursive functions. The only exceptions are the functions on buffers. As we mentioned earlier buffers contain at most five elements, so these are indeed $O(1)$. Additionally, when we look at the regularization cases it becomes clear that the original description relies heavily on sequential thinking. For sake of consistency we implemented these in the same way but a non-sequential-style implementation may provide a simpler regularization function.

Finally, the instance of `Deque RTDeque` is derived by performing an operation on the appropriate top level buffer and regularizing the resulting deque.

5 Results

A measurement of `toList` and `fromList` (Figure 4) gives us the expected result. Since the two operations are implemented using `eject` and `inject`, respectively, we perform n deque operations with a list of length n. This means we expect a linear increase in runtime, which we can see in the measurement. We can also see a somewhat discrete step size in the increase in runtime. These „levels“ may correspond to the performed regularization case.

In Figure 5 we see the performance of `inject` and `eject`. These measurements match the measurements from Figure 4. The high runtime of the small deques is an effect due to our measurement setup. Included in these run times is the program start, which has more effect on smaller deque sizes. We are confident, that a repeated measurement with a more accurate setup would give us a runtimes like we see with larger deques. Unfortunately, we couldn’t do these measurements for time reasons.

6 Conclusion

We were able to implement the deque presented by *Kaplan et al.* in Haskell and confirm the real-time behavior experimentally. The biggest challenge was to bring the purely

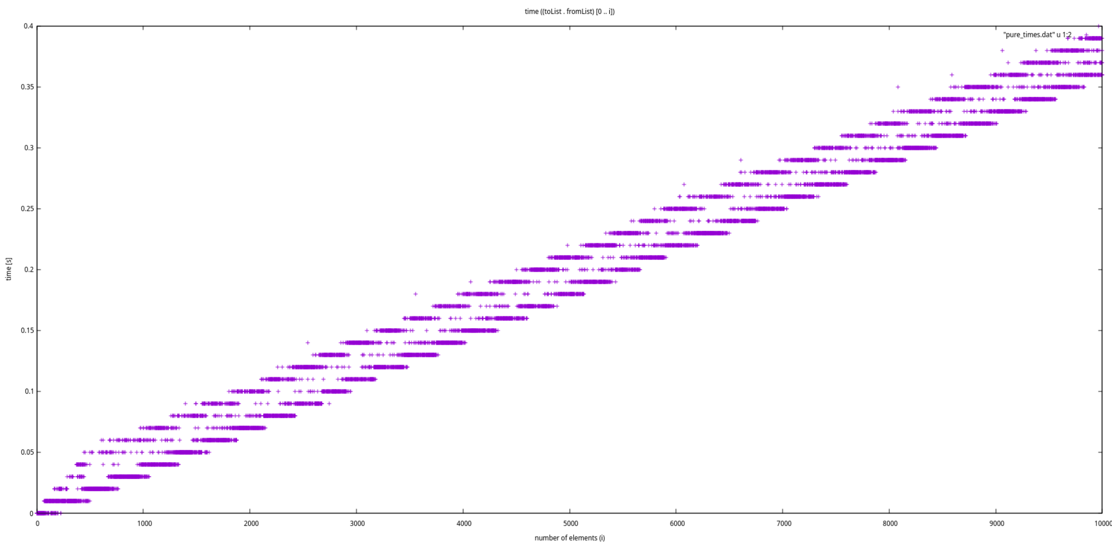

Figure 4: runtime of `toList . fromList` with lists of length zero to 10000.

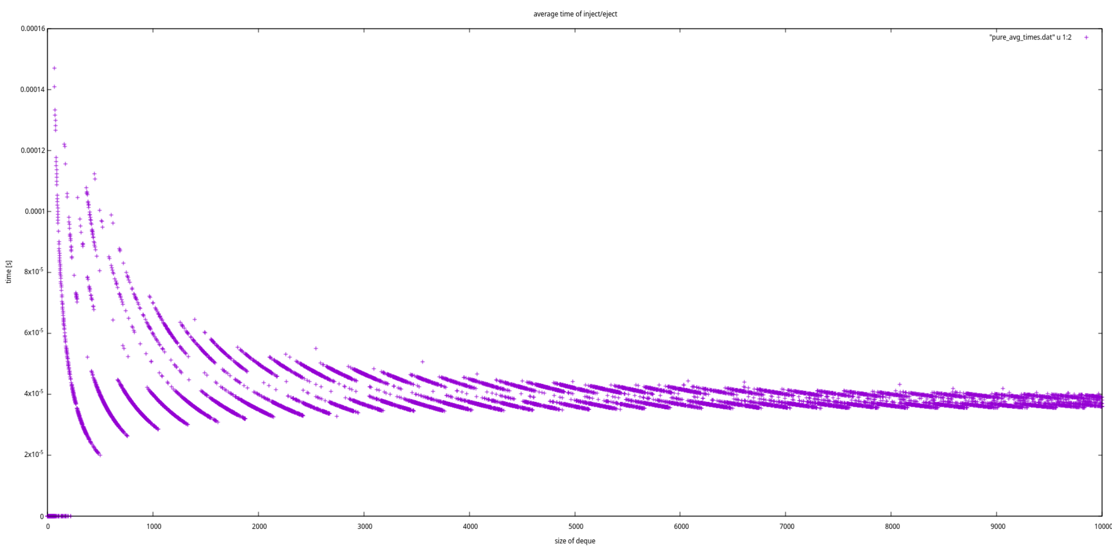

Figure 5: runtime of `inject` and `eject` on deques of length zero to 10000.

textual explanation to a level that allowed us to implement the data structure and regularization procedure. Subsequently, our goal was not to provide a production-ready high performance but rather a conceptual and concise implementation. Though we have a working implementation there are some things which can be further simplified, for example the regularization cases and the handling of the deque shape.

In the process of bringing the data structure to code, we also implemented a monadic version that counts the cost of each deque operation. Since we saw later on that we can actually implement everything without recursion, the question of complexity became trivial and we moved to the now implemented pure version.

7 Further Work

In the paper this report is based on [5] there are two more data structures: *purely functional real-time steques with catenation* and *purely functional real-time deques with catenation*. The overall goal is to implement and formalize all three data structures. This would be a step to make them available to interactive theorem provers and dependently typed languages such as *Agda*, *Coq*, *Isabelle/HOL* and *Lean*, as well as simply typed functional languages like *Haskell* and *OCaml*. Moreover, dependent typing may give us the opportunity to index on the nested pairs in the deque datatype and refine our current solution with arbitrary binary trees. To which degree this is possible is an open question for us.

Moving away from pure functional programing to imperative programming the third data structure, the deque with real-time catenation, may also be of use. It solves the problem of self-catenation since it is persistent. In implementations like doubly linked lists self catenation may introduce non-termination if not handled properly because it leads to circular traversal. Although, one must also mention that despite being real-time these purely functional implementations are not as straight forward and performant as imperative ones. This is a downside of using deques based on persistent data structures in imperative programming. The gain of safety and the ability to self-catenation however could be useful in some domains.

8 Acknowledgements

I thank my supervisor Professor Dr. René Thiemann for helping me in conducting the project and producing the resulting material.

References

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

[2] Robert T Hood. The efficient implementation of very-high-level programming language constructs. Technical report, Cornell University, 1982.

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

[4] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[5] Haim Kaplan and Robert E Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM (JACM)*, 46(5):577–603, 1999.

[6] Neil Ivor Sarnak. *Persistent data structures.* New York University, 1986.

Appendix

Source Code

```haskell

{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE TypeSynonymInstances #-}




module Deque
where


import Control.Monad ((>=>), (<=<), (=<<), join)
import Control.Monad.Identity
import Control.Arrow (first, second, (>>>))
import Data.Bifunctor (bimap)
import Data.Functor ((<&>))
import Data.Foldable (foldl')
import Data.Function ((&))

import Prelude hiding (null)
import qualified Prelude


{-|
Implementation of
  "Purely Functional, Real-Time Deques with Catenation, KAPLAN and TARJAN et. al.,
  Journal of the ACM, Vol. 46, No. 5, September 1999, pp. 577-603."


This module implements real-time deques without catenation.
we ensure the real-time complexity by restricting ourselves mainly to non-recursive functions.
If we need to use recursive functions, we add a justification why this function is still constant time or why it has no effect
on the behavior of the real-time deque.
-}

-- * Deques
```

```haskell

-- | a deque as a class.
class Deque (a :: * -> *) where
  push :: b -> a b -> a b
  pop :: a b -> (b, a b)
  inject :: b -> a b -> a b
  eject :: a b -> (b, a b)
  null :: a b -> Bool
  empty :: a b
  append :: a b -> a b -> a b
  append xs ys = foldl' (flip inject) ys $ toList xs
  len :: a b -> Int
  len = length . toList



-- ** Deque packing and unpacking
--
-- note that fromList and toList are not scope of the realtime behaviour.

toList :: (Deque a) => a b -> [b]
toList = go []
  where
    go acc d
      | null d    = acc
      | otherwise = uncurry go $ first (:acc) $ eject d

fromList :: Deque a => [b] -> a b
fromList xs = do
  foldl' (flip inject) empty xs


-- * nonempty, leaf only binary trees

data BTree a = Leaf !a | Node (BTree a) (BTree a)

instance Show a => Show (BTree a) where
  show (Node x y) = "(" ++ show x ++ ", " ++ show y ++ ")" -- ^this method is not used in the deque
  show (Leaf v) = show v

-- ** convenience functions
```

```haskell

combine :: BTree a -> BTree a -> BTree a
combine = Node

split :: BTree a -> (BTree a, BTree a)
split (Leaf _)     = error "split: cannot split leaf"
split (Node l r)   = (l, r)

unleaf :: BTree a -> a
unleaf (Node _ _) = error "unleaf: cannot unpack node"
unleaf (Leaf x)    = x


-- * Colors

-- | Colors as defined in the paper
data Color = Red | Yellow | Green
  deriving (Ord, Eq, Show)

-- | Class for colored things
class Colored (a :: *) where
  color :: a -> Color

-- * Buffers

-- | we use lists as Buffers. note that they must fit at most 5 elements.
-- subsequently, if we use non-constant linear functions in this context they become constant time.
type Buffer = []

instance Deque Buffer where
  push = (:)
  pop (x:xs) = (x, xs)
  inject x xs = xs ++ [x]
  eject xs = (last xs, take (length xs - 1) xs)
  null = Prelude.null
  empty = []


instance Colored (Buffer a) where
  color buf = case len buf of
    0 -> Red
```

```haskell
    1 -> Yellow
    2 -> Green
    3 -> Green
    4 -> Yellow
    5 -> Red
    _ -> error "color: buffer overflow"


-- * Real time Deques

-- | the real time deque.
data RTDeque a
  = Nil
  | RTDeque
    { _prefix :: Buffer (BTree a)
    , _suffix :: Buffer (BTree a)
    , _yellows :: RTDeque a     -- ^the yellow descendants. these must not contain nonyellow branches.
    , _nonyellows :: RTDeque a  -- ^the nonyellow descendants.
    }
  deriving (Show)


-- | an empty instance of a RTDeque
emptyRTDeque :: RTDeque a
emptyRTDeque = RTDeque
  { _prefix = empty
  , _suffix = empty
  , _yellows = Nil
  , _nonyellows = Nil
  }

-- ** guarded getters, modifiers and setters

prefix, suffix :: RTDeque a -> Buffer (BTree a)
prefix Nil = empty
prefix d   = _prefix d
suffix Nil = empty
suffix d   = _suffix d

yellows, nonyellows :: RTDeque a -> RTDeque a
yellows Nil     = Nil
yellows d       = _yellows d
```

```haskell
nonyellows Nil  = Nil
nonyellows d    = _nonyellows d

onPrefix, onSuffix :: (Buffer (BTree a) -> Buffer (BTree a)) -> RTDeque a -> RTDeque a
onPrefix f Nil = emptyRTDeque { _prefix = f empty }
onPrefix f deq = deq { _prefix = f (_prefix deq) }
onSuffix f Nil = emptyRTDeque { _suffix = f empty }
onSuffix f deq = deq { _suffix = f (_suffix deq) }
setPrefix, setSuffix :: Buffer (BTree a) -> RTDeque a -> RTDeque a
setPrefix = onPrefix . const
setSuffix = onSuffix . const


onNonyellows, onYellows :: (RTDeque a -> RTDeque a) -> RTDeque a -> RTDeque a
onNonyellows f Nil = emptyRTDeque { _nonyellows = f emptyRTDeque }
onNonyellows f deq = deq { _nonyellows = f $ _nonyellows deq }
onYellows f Nil = emptyRTDeque { _yellows = f emptyRTDeque }
onYellows f deq = deq { _yellows = f $ _yellows deq }

setNonyellows, setYellows :: RTDeque a -> RTDeque a -> RTDeque a
setNonyellows = onNonyellows . const
setYellows    = onYellows . const

-- ** auxiliary functions

-- | check if a RTDeque is a bottom element (i.e does not contain any values).
-- due to our regularization procedure we know that a deque is bottom if the buffers are empty.
bottom :: RTDeque a -> Bool
bottom Nil = True
bottom deq = all null [prefix deq, suffix deq]


-- | replace RTDeque with Nil if it is the bottom of the deque
truncate :: RTDeque a -> RTDeque a
truncate deq
  | bottom deq = Nil
  | otherwise  = deq
```

```haskell

-- | apply a function on the descendant of the RTDeque
withNext :: (RTDeque a -> b) -> RTDeque a -> b
withNext fun deq
  | bottom (yellows deq)  = fun $ nonyellows deq
  | otherwise             = fun $ yellows deq

instance Colored (RTDeque a) where
  color deq =
    let
      bot = withNext bottom deq
      pnull = null $ prefix deq
      snull = null $ suffix deq
    in
      case (bot, pnull, snull) of
        (True, True, _    ) -> color $ suffix deq
        (True, _    , True) -> color $ prefix deq
        (_    , _    , _    ) -> min (color $ prefix deq) (color $ suffix deq)


-- * Regularity
--
-- The regularity checks are implemented for completeness. they are not used in the actual RTDeque since regularity and semiregularity is
-- invariant to the functions. This also means that the checks have no effect on the complexity.


-- | semiregularity check
-- note that we also check if the partition (yellows/nonyellows) is correct (is an error case)
semiregular :: RTDeque a -> Bool
semiregular deque = bottom deque
  ||  ( semiregular (nonyellows deque)
        && allyellow (yellows deque)
        &&  ( Red /= color deque || greenBeforeRed (nonyellows deque)
            )
      )
  where
    greenBeforeRed d =  case (bottom d, color d) of
      (False, Red    ) -> False
```

```haskell
      (_     , Yellow) -> error "semiregular: yellows deque in the nonyellows stack"
      (_     , _      ) -> True
    allyellow d = bottom d
      ||  ( color d == Yellow
            && bottom (nonyellows d)
            && allyellow (yellows d)
          )
      || error "semiregular: nonyellows deque in yellows stack"


-- | regularity check
regular :: RTDeque a -> Bool
regular deque
  | bottom deque               = True
  | not (semiregular deque)    = False
  | otherwise                  = case (color deque, bottom (nonyellows deque), color (nonyellows deque)) of
    (Green, _    , _     ) -> True
    (Red  , _    , _     ) -> False
    (_    , True, _      ) -> True
    (_    , _    , Green) -> True
    (_    ,_     , _     ) -> False


-- * Restoring Regularity


-- | restore a semiregular deque to a regular deque
--
-- this function is constant time, since it is not recursive and does not use any recursive functions.
restoreRegularity :: RTDeque a -> RTDeque a
restoreRegularity deque
  | bottom deque                                                    = deque
  | color deque == Green                                            = deque
  | color deque == Red                                              = withNext (restore deque) deque
  | color deque == Yellow && color (nonyellows deque) == Green  = deque
  | color deque == Yellow && color (nonyellows deque) == Red    = onNonyellows (\ny -> withNext (restore ny) ny) deque
  | otherwise                                                       = error "restoreRegularity: nonexhaustive matching"
```

```haskell
  where
    restore deque child =
      let
        p1 = len $ prefix deque
        s1 = len $ suffix deque
        p2 = len $ prefix child
        s2 = len $ suffix child
        two_buffer_case_cond  = p2 + s2 >= 2
        one_buffer_case_cond  = p2 + s2 <= 1 && (p1 >= 2 || s1 >= 2)
        no_buffer_case_cond   = p2 + s2 <= 1 && (p1 <= 1 && s1 <= 1)
        case_fun = case (two_buffer_case_cond, one_buffer_case_cond, no_buffer_case_cond) of
          (True, _    , _   ) -> twoBufferCase
          (_   , True, _    ) -> oneBufferCase
          (_   , _    , True) -> noBufferCase
          (_   , _    , _   ) -> error "restoreRegularity: no case applicable"
      in
        combineDeqs $ case_fun buffers
      where
        -- the buffers to modify
        buffers = (prefix deque, suffix deque, prefix child, suffix child)
        -- moving elements between buffers
        p1_to_p2,p2_to_p1,p2_to_s2,p2_to_s1,s2_to_p2,s2_to_s1,s1_to_s2,s1_to_p2 :: (Buffer (BTree a), Buffer (BTree a), Buffer (BTree a), Buffer (BTree a)) -> (Buffer (BTree a), Buffer (BTree a), Buffer (BTree a), Buffer (BTree a))
        p1_to_p2 (p1, s1, p2, s2) = let (y,(x,p1')) = second eject $ eject p1 ; p2' = push (combine x y) p2   in (p1', s1  , p2' , s2 )
        p2_to_p1 (p1, s1, p2, s2) = let ((x,y),p2') = first split $ pop p2    ; p1' = inject y $ inject x p1  in (p1', s1  , p2' , s2 )
        p2_to_s2 (p1, s1, p2, s2) = let (x,p2')     = eject p2 ; s2' = push x s2                    in (p1 , s1  , p2' , s2')
        p2_to_s1 (p1, s1, p2, s2) = let ((x,y),p2') = first split $ eject p2  ; s1' = push x $ push y s1     in (p1 , s1' , p2' , s2 )
        s2_to_p2 (p1, s1, p2, s2) = let (x,s2')     = pop s2 ; p2' = inject x p2                    in (p1 , s1  , p2' , s2')
        s2_to_s1 (p1, s1, p2, s2) = let ((x,y),s2') = first split $ eject s2  ; s1' = push x $ push y s1     in (p1 , s1' , p2  , s2')
        s1_to_s2 (p1, s1, p2, s2) = let (x,(y,s1')) = second pop $ pop s1     ; s2' = inject (combine x y) s2 in (p1 , s1' , p2  , s2')
```

```
    s1_to_p2 (p1, s1, p2, s2) = let (x,(y,s1')) = second pop $ pop s1     ; p2' = inject (combine x y) p2 in (p1 , s1' , p2' , s2 )
    -- combine deque and child with the new buffers.
    -- the parent deque has changed from red to green.
    -- we need to rotate the tree if child changed from yellow to nonyellow or vice versa.
    combineDeqs (p1, s1, p2, s2) =
      let
        child' = truncate $ setPrefix p2 $ setSuffix s2 child
        deque' = truncate $ setPrefix p1 $ setSuffix s1 deque
      in
        case (color child, color child') of
          (Yellow, Yellow) -> setYellows child' deque'
          (Yellow, _     ) -> let child'' = truncate $ setNonyellows (nonyellows deque') child'
                              in setYellows Nil $ setNonyellows child'' deque'
          (_     , Yellow) -> setYellows child' $ setNonyellows (nonyellows child') deque'
          (_     , _     ) -> setNonyellows child' deque'
    -- two buffer case: p2 + s2 >= 2
    twoBufferCase =
      let
        balance_lower                (p1, s1, p2, s2)
          | len p2 == 0   = s2_to_p2 (p1, s1, p2, s2)
          | len s2 == 0   = p2_to_s2 (p1, s1, p2, s2)
          | otherwise     =          (p1, s1, p2, s2)
        prop_prefix_down             (p1, s1, p2, s2)
          | len p1 >= 4   = p1_to_p2 (p1, s1, p2, s2)
          | otherwise     =          (p1, s1, p2, s2)
        prop_prefix_up               (p1, s1, p2, s2)
          | len p1 <= 1   = p2_to_p1 (p1, s1, p2, s2)
          | otherwise     =          (p1, s1, p2, s2)
        prop_suffix_down             (p1, s1, p2, s2)
          | len s1 >= 4   = s1_to_s2 (p1, s1, p2, s2)
          | otherwise     =          (p1, s1, p2, s2)
        prop_suffix_up               (p1, s1, p2, s2)
          | len s1 <= 1   = s2_to_s1 (p1, s1, p2, s2)
          | otherwise     =          (p1, s1, p2, s2)
      in
        balance_lower
        >>> prop_prefix_down
        >>> prop_suffix_down
```

```
              >>> prop_prefix_up
              >>> prop_suffix_up
        -- one buffer case: p2 + s2 <= 1 && (p1 >= 2 || s1 >= 2)
        oneBufferCase =
          let
            move_lower_to_prefix          (p1, s1, p2, s2)
              | len s2 == 1   = s2_to_p2  (p1, s1, p2, s2)
              | otherwise     =           (p1, s1, p2, s2)
            prop_prefix_down              (p1, s1, p2, s2)
              | len p1 >= 4   = p1_to_p2  (p1, s1, p2, s2)
              | otherwise     =           (p1, s1, p2, s2)
            prop_prefix_up                (p1, s1, p2, s2)
              | len p1 <= 1   = p2_to_p1  (p1, s1, p2, s2)
              | otherwise     =           (p1, s1, p2, s2)
            prop_suffix_down              (p1, s1, p2, s2)
              | len s1 >= 4   = s1_to_p2  (p1, s1, p2, s2)
              | otherwise     =           (p1, s1, p2, s2)
            prop_suffix_up                (p1, s1, p2, s2)
              | len s1 <= 1   = p2_to_s1  (p1, s1, p2, s2)
              | otherwise     =           (p1, s1, p2, s2)
          in
            move_lower_to_prefix
            >>> prop_prefix_down
            >>> prop_suffix_down
            >>> prop_prefix_up
            >>> prop_suffix_up
        -- no buffer case: p2 + s2 <= 1 && (p1 <= 1 && s1 <= 1)
        noBufferCase =
          let
            move_s2_to_p2               (p1, s1, p2, s2)
              | len s2 == 1 = s2_to_p2  (p1, s1, p2, s2)
              | otherwise   =           (p1, s1, p2, s2)
            prop_prefix_up              (p1, s1, p2, s2)
              | len p2 == 1 = p2_to_p1  (p1, s1, p2, s2)
              | otherwise   =           (p1, s1, p2, s2)
          in
            move_s2_to_p2 >>> prop_prefix_up



-- | the deque instance for Real Time deques
instance Deque RTDeque where
  push x = restoreRegularity . onPrefix (push (Leaf x))
```

```haskell
  pop deq =
    let
      pnull = null (prefix deq) -- if the prefix is empty we pop the suffix
      buf = if pnull then suffix deq else prefix deq
      (x, buf') = pop buf
      deq' = restoreRegularity $ (if pnull then setSuffix else setPrefix) buf' deq
    in (unleaf x, deq')
  inject x = restoreRegularity . onSuffix (inject (Leaf x))
  eject deq =
    let
      snull = null $ suffix deq -- if the suffix is empty we eject the prefix
      buf = if snull then prefix deq else suffix deq
      (x, buf') = eject buf
      deq' = restoreRegularity $ (if snull then setPrefix else setSuffix) buf' deq
    in (unleaf x, deq')
  null = bottom
  empty = Nil
```