COMP50001: Algorithm Design & Analysis

Sheet 4 (Week 5)

Exercise 4.1

1. Give the time complexity of the following *reverse* in terms of the length of a deque:

$$\begin{aligned}&reverse :: Deque\ a \to Deque\ a\\&reverse = fromList \circ reverse \circ toList\end{aligned}$$

2. Implement a *reverse* for deques that runs in $O(1)$ time.

Exercise 4.2

Supposing $xs_0 :: Deque\ a$ is the empty deque, show that the amortised complexity of each operation in the following sequence is $O(1)$:

$$xs_0 \overset{op_0}{\rightsquigarrow} xs_1 \overset{op_1}{\rightsquigarrow} xs_2 \overset{op_2}{\rightsquigarrow} \ldots \overset{op_{n-1}}{\rightsquigarrow} xs_n$$

where each $op_i \in \{tail,\ snoc,\ cons\}$.

Exercise 4.3

Suppose no invariants are imposed on *Deque* and *snoc* and *tail* are alternatively defined as *snoc′* and *tail′*.

1. Prove that the amortised complexity of each operation in a sequence of *snoc′* and *tail′* is still $O(1)$.
2. Suppose that the sequence additionally contains operation *init′*. Determine whether the amortised complexity is still $O(1)$.

Exercise 4.4

Consider the following alternative representation of *Deque*:

$$\textbf{data}\ Deque'\ a = Deque'\ Int\ [a]\ [a]$$

with an invariant $n = length\ us + length\ sv$ for any $Deque'\ n\ us\ sv$. Define *cons* and $+\!\!+$ for this representation as follows:

$$\begin{aligned}&cons :: a \to Deque'\ a \to Deque'\ a\\&cons\ u\ (Deque'\ n\ us\ sv) = Deque'\ (n+1)\ (u : us)\ sv\\&(+\!\!+) :: Deque'\ a \to Deque'\ a \to Deque'\ a\\&Deque'\ n\ us\ sv +\!\!+ Deque'\ n'\ us'\ sv'\\&\quad \mid n < n' \quad = Deque'\ (n+n')\ (us +\!\!+ reverse\ sv +\!\!+ us')\ sv'\\&\quad \mid otherwise = Deque'\ (n+n')\ us\ (sv' +\!\!+ reverse\ us' +\!\!+ sv)\end{aligned}$$

1. Give the worst-case complexity of $xs +\!\!+ ys$ in terms of *length xs* and *length ys*.

```
data Deque a = Deque [a] [a]

instance List Deque where
  toList :: Deque a → [a]
  toList (Deque xs sy) = xs ++ reverse sy

  fromList xs = Deque ys (reverse zs)
    where (ys, zs) = splitAt (length xs `div` 2) xs

  tail :: Deque a → Deque a
  tail (Deque []     []) = error "tail: empty list"
  tail (Deque []     sy) = empty
  tail (Deque [x]    sy) = fromList (reverse sy)
  tail (Deque (x:xs) sy) = Deque xs sy

  cons :: a → Deque a → Deque a
  cons x (Deque xs []) = Deque [x] xs
  cons x (Deque xs sy) = Deque (x:xs) sy

  snoc :: Deque a → a → Deque a
  snoc (Deque [] sv) x = Deque sv [x]
  snoc (Deque us sv) x = Deque us (x:sv)
```

```
snoc' :: Deque a → a → Deque a
snoc' (Deque us sv) v = Deque us (v:sv)

tail' :: Deque a → Deque a
tail' (Deque [] [])  = error "tail: empty list"
tail' (Deque [] sv)  = Deque (tail (reverse sv)) []
tail' (Deque us sv)  = Deque (tail us) sv

init' :: Deque a → Deque a
init' (Deque [] [])  = error "init: empty list"
init' (Deque us [])  = Deque [] (tail (reverse us))
init' (Deque us sv)  = Deque us (tail sv)
```

```
instance List Deque' where
  toList :: Deque' a → [a]
  toList (Deque' n us sv) = us ++ reverse sv

  fromList :: [a] → Deque' a
  fromList xs = Deque' n us sv
    where n       = length xs
          (us,vs) = splitAt (n `div` 2) xs
          sv      = reverse vs
```

2. Consider a sequence of operations creating and manipulating multiple deques

$$D_0 \overset{op_0}{\rightsquigarrow} D_1 \overset{op_1}{\rightsquigarrow} D_2 \overset{op_2}{\rightsquigarrow} \ldots D_{n-1} \overset{op_{n-1}}{\rightsquigarrow} D_n$$

where each D_i is a *multiset* of deques and $D_0 = \emptyset$. Each op_i is only one of the following forms:

(a) $xs_i = empty$, and in this case

$$D_{i+1} = D_i \cup \{xs_i\},$$

(b) $xs_i = cons\ x\ xs_j$ where $xs_j \in D_i$, and in this case

$$D_{i+1} = (D_i \setminus \{xs_j\}) \cup \{xs_i\},$$

(c) $xs_i = xs_j +\!\!+ xs_k$ where $xs_j, xs_k \in D_i$ and $j \neq k$, and in this case

$$D_{i+1} = (D_i \setminus \{xs_j,\ xs_k\}) \cup \{xs_i\}.$$

For case (c) $xs_i = xs_j +\!\!+ xs_k$, if xs_{small} is the one in xs_j and xs_k with the smaller length, the elements in xs_{small} is said to be *merged into a larger deque*. Explain why every element can only be merged into a larger deque at most $\lceil \log_2 n \rceil$ times, where n is the length of the sequence of operations.

3. Prove that each operation has amortised complexity $O(\log_2 n)$ with the following size function:

$$S(D) = sum\ [length\ xs \times log_2\ (n\ /\ (length\ xs)) \mid xs \leftarrow D, length\ xs > 0]$$

and explain the intuition of this size function. (The cost incurred by operations $\cup$ and $\setminus$ on the multiset does not need to be considered in the analysis.)

Exercise 4.5

Define $dec :: Binary \rightarrow Binary$ that decrements a binary number discussed in the lecture and show that the amortised complexity of repeated applications of *dec* is $O(1)$. Determine if the amortised complexity of each operation in a sequence op_i where $0 \leqslant i < n$ and $op_i \in \{inc,\ dec\}$ is still $O(1)$.

```
type Binary = [Digit]
data Digit = O | I

inc :: Binary → Binary
inc []       = [I]
inc (O : bs) = I  : bs
inc (I : bs) = O : (inc bs)
```

Exercise 4.6

Given is the data type *Tree* which is an instance of *List*. The tree $t :: Tree\ a$ is *balanced* iff $t = Tip$, $t = Leaf\ a$, or $t = Node\ n\ l\ r$ with balanced subtrees l and r such that $size\ l = size\ r$.

Show that $(!!) :: Tree\ a \rightarrow Int \rightarrow a$ takes $O(\log_2 n)$ time for balanced binary trees using a recurrence relation, where n is the number of elements in the tree.

```
data Tree a = Tip
            | Leaf a
            | Node Int (Tree a) (Tree a)

size (Tip)        = 0
size (Leaf _)     = 1
size (Node n _ _) = n
```

Exercise 4.7

Consider the definition of *RAList a*.

1. Letting $head = (!!0)$, give the best-case and worst-case time complexities of *head xs* where $xs :: RAList\ a$.
2. Implement $tail :: RAList\ a \to RAList\ a$ such that the amortised complexity of a sequence of *tail* is $O(1)$.
3. Determine if the amortised complexity of a sequence of operations where each operation is either *tail* or *cons* is still $O(1)$. (Hint: compare with *inc* and *dec* for binary numbers.)

```
newtype RAList a = RAList [Tree a]
instance List RAList where
  fromList :: [a] → RAList a
  fromList xs = foldr cons empty xs
  toList :: RAList a → [a]
  toList (RAList ts) = (concat ∘ map toList) ts
  (!!) :: RAList a → Int → a
  RAList (t : ts) !! k
    | k < size t  = t !! k
    | otherwise = RAList ts !! (k − size t)
```