50001 - Algorithm Analysis and Design - Lecture 8

Oliver Killane

17/11/21

Lecture Recording

Lecture recording is available here

Amortized Analysis

So far we have studied complexity of a single, isolated run of an algorithm. **Amortizsed Analysis** is about understanding cost in a wider context (e.g averaged over many calls to a routine).

Dequeues

Dequeue

A Dequeue is a double ended queue. An abstract datatype that generalises a queue. Elements can be added or removed from either end.

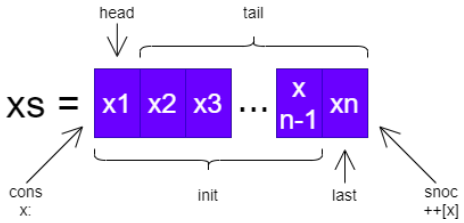
Common associated funtions are:

snoc	Insert element at the back of the queue.
cons	Insert element at the front of the queue.
eject	Remove last element.
pop	remove fist element.
peek	Examine but do not remove first element.

Dequeues are also called head-tail linked lists or symmetric lists.

we use a dequeue when we want to reduce the time taken to perform certain operations.

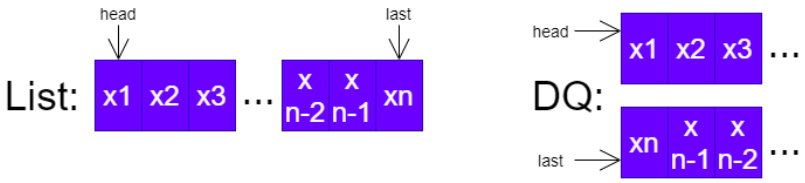
List Operation Complexity



cons	O(1)	**snoc**	O(1)
head	O(1)	**last**	O(1)
tail	O(1)	**init**	O(1)

Dequeue Structure

To achieve $O(1)$ complexity in the **snoc**, **init** and **last** we use two lists.



One list starts contains the start of the list, and the other the end (reversed).

We keep two invariants for *Dequeue us sv*:

$$null\ us \Rightarrow null\ sv \vee single\ sv$$

$$null\ sv \Rightarrow null\ us \vee single\ us$$

In other words, if one list is empty, the other can contain at most 1 element.

An example implementation in haskell is below:

```haskell
-- can ignore certain patterns due to invariant
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}

data Dequeue a = Dequeue [a] [a]

instance List Dequeue where
  toList :: Dequeue a -> [a]
  toList (Dequeue us sv) = us ++ reverse sv

  fromList :: [a] -> Dequeue a
  fromList xs = Dequeue us (reverse vs)
    where (us,vs) = splitAt (length xs `div` 2) xs

  -- use the invariant, if [] sv then sv = [x] or []

  -- O(1)
  cons :: a -> Dequeue a -> Dequeue a
  cons x (Dequeue us []) = Dequeue [x] us
  cons x (Dequeue us sv) = Dequeue (x:us) sv

  -- O(1)
  snoc :: Dequeue a -> a -> Dequeue a
  snoc (Dequeue [] sv) x = Dequeue sv [x]
  snoc (Dequeue us sv) x = Dequeue us (x:sv)

  -- O(1)
  last :: Dequeue a -> a
  last (Dequeue _ (s:_)) = s
  last (Dequeue [u] _) = u
  last (Dequeue [] []) = error "Nothing in the dequeue"

  -- O(1)
  head :: Dequeue a -> a
  head (Dequeue (u:_) _) = u
  head (Dequeue [] [v]) = v
  head (Dequeue [] []) = error "Nothing in the dequeue"

  -- O(1)
  tail :: Dequeue a -> Dequeue a
```

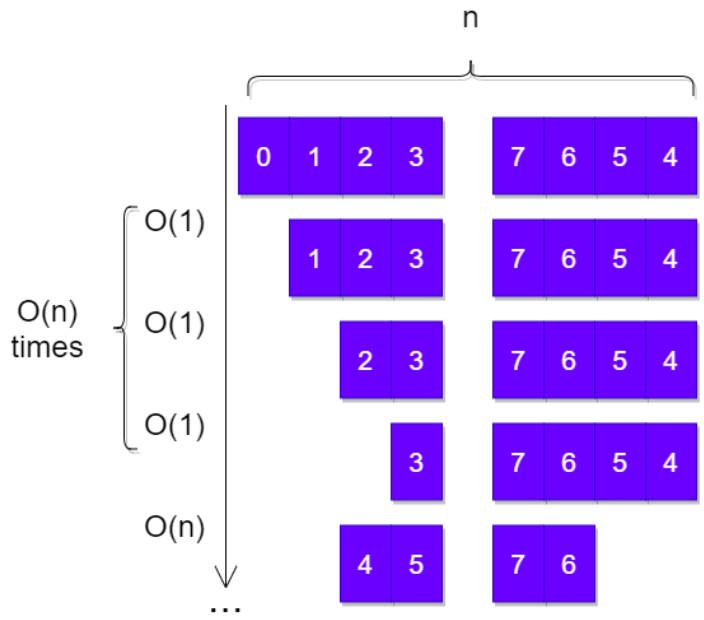
```
40  tail (Dequeue [] []) = error "Nothing in the dequeue"
41  tail (Dequeue [] _) = empty
42  tail (Dequeue _ []) = empty
43  tail (Dequeue [u] sv) = Dequeue (reverse su)
44    where
45      n = length sv
46      (su, sv') = splitAt (n `div` 2) sv
47      -- note: could also do a fromList (reverse sv) but less efficient
48
49  tail (Dequeue (u:us) sv) = Dequeue us sv
50
51  -- O(1)
52  init :: Dequeue a -> Dequeue a
53  init (Dequeue [] []) = error "Nothing in the dequeue"
54  init (Dequeue [] _) = empty
55  init (Dequeue _ []) = empty
56  init (Dequeue us [s]) = fromList us
57  init (Dequeue us (s:sv)) = Dequeue us sv
58
59  isEmpty :: Dequeue a -> Bool
60  isEmpty (Dequeue [] []) = True
61  isEmpty _ = False
62
63  empty :: Dequeue a
64  empty = Dequeue [] []
```

When considering the cost of **tail** and **init** we must consider that there are two possibilities:

High Cost	$init(Dequeueus[s])$ $tail(Dequeue[u]sv)$	This operation is $O(n)$ complexity due to the **spitAt** and **reverse** operation done on half of a list.
Low Cost	$init(Dequeueus(s:sv))$ $tail(Dequeue(u:us)sv)$	Low cost $O(1)$ operation as it requires only a pattern match on the first element.

As both of these operations rebalance the **Dequeue** to to be balanced (half the queue on each list), we these operations can have an amortized cost of $O(1)$.

We know this as the average cost is of order $O(1)$. The $O(n)$ cost is incurred every $n/2$ calls to **tail**/**init**.


n
0 1 2 3
7 6 5 4
O(1)
1 2 3
7 6 5 4
O(n)
times
O(1)
2 3
7 6 5 4
O(1)
3
7 6 5 4
O(n)
4 5
7 6
...