

50001 - Algorithm Analysis and Design - Lecture 8

Oliver Killane

17/11/21

Amortized Analysis

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

Dequeues

Deque

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

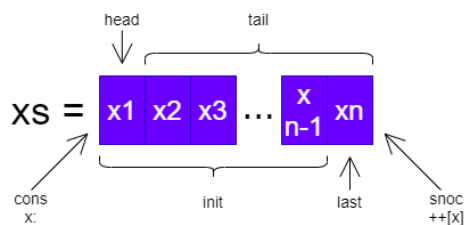
Common associated functions are:

snoc	Insert element at the back of the queue.
cons	Insert element at the front of the queue.
eject	Remove last element.
pop	remove first 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

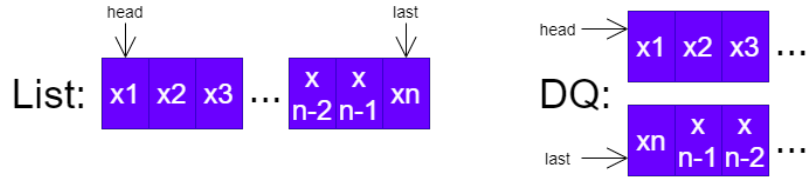


cons	$O(1)$
head	$O(1)$
tail	$O(1)$

snoc	$O(1)$
last	$O(1)$
init	$O(1)$

Deque 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:

```

1  — can ignore certain patterns due to invariant
2  {-# OPTIONS.GHC -Wno-incomplete-patterns #-}
3
4  data Dequeue a = Dequeue [a] [a]
5
6  instance List Dequeue where
7    toList :: Dequeue a -> [a]
8    toList (Dequeue us sv) = us ++ reverse sv
9
10   fromList :: [a] -> Dequeue a
11   fromList xs = Dequeue us (reverse vs)
12     where (us, vs) = splitAt (length xs `div` 2) xs
13
14   — use the invariant, if [] sv then sv = [x] or []
15
16   — O(1)
17   cons :: a -> Dequeue a -> Dequeue a
18   cons x (Dequeue us []) = Dequeue [x] us
19   cons x (Dequeue us sv) = Dequeue (x:us) sv
20
21   — O(1)
22   snoc :: Dequeue a -> a -> Dequeue a
23   snoc (Dequeue [] sv) x = Dequeue sv [x]
24   snoc (Dequeue us sv) x = Dequeue us (x:sv)
25
26   — O(1)
27   last :: Dequeue a -> a
28   last (Dequeue _ (s:_)) = s
29   last (Dequeue [u] _) = u
30   last (Dequeue [] []) = error "Nothing in the dequeue"
31
32   — O(1)
33   head :: Dequeue a -> a
34   head (Dequeue (u:_ _) _) = u
35   head (Dequeue [] [v]) = v
36   head (Dequeue [] []) = error "Nothing in the dequeue"
37
38   — O(1)
39   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 [] []

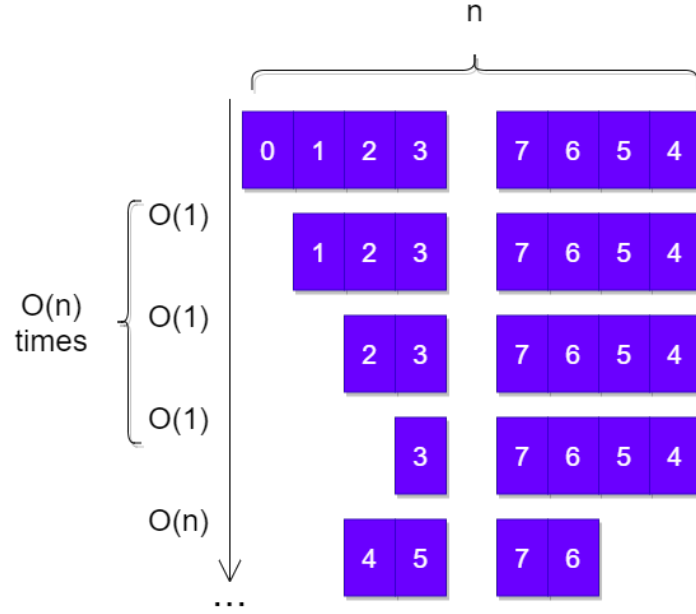
```

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

High Cost	$init(Dequeue\ us[s])$ $tail(Dequeue\ [u]sv)$	This operation is $O(n)$ complexity due to the splitAt and reverse operation done on half of a list.
Low Cost	$init(Dequeue\ us(s:sv))$ $tail(Dequeue\ (u:us)sv)$	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**.



Amortization

The complexity of **tail** is an example of **Amortized analysis**, where operation's wider context are considered when calculating the complexity.

$$xs_0 \xrightarrow{op_0} xs_1 \xrightarrow{op_1} xs_2 \xrightarrow{op_2} xs_3 \xrightarrow{op_3} \dots \xrightarrow{op_{n-1}} xs_n$$

We defined 3 parts:

1. **Cost Function**

$C_{op_i}(xs_i)$ determines the cost of operation op_i on data xs_i . Estimating how many steps it takes for each operation to execute.

2. **Amortized Cost Function**

$A_{op_i}(xs_i)$ for each operation op_i on data xs_i .

3. **Size Function**

$S(xs)$ that calculates the size of data xs

We define these functions with the goal to show that:

$$C_{op_i}(xs_i) \leq A_{op_i}(xs_i) + S(xs_i) - S(xs_{i+1})$$

The cost of the operation is smaller than the amortized cost, plus the difference in size of the data structure before and after the operation.

Once this is shown, we can infer that:

$$\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leq \sum_{i=0}^{n-1} A_{op_i}(xs_i) + S(xs_0) - S(xs_n)$$

Furthermore when $S(xs_0) = 0$ this implies:

$$\sum_{i=0}^{n-1} C_{op_i}(xs_i) \leq \sum_{i=0}^{n-1} A_{op_i}(xs_i) - S(xs_n) \Rightarrow \sum_{i=0}^{n-1} C_{op_i}(xs_i) \leq \sum_{i=0}^{n-1} A_{op_i}(xs_i)$$

This means the cost of the operations is less than the sum of the amortized costs.

For example, if $A_{op_i}(xs) = 1$ then the total cost will be bounded by $O(n)$.

We can use the **dequeue** operations as an example:

$$C_{cons}(xs) = 1 \quad C_{snoc}(xs) = 1 \quad C_{head}(xs) = 1 \quad C_{last}(xs) = 1$$

For tail we can do the following:

- | | | |
|-----|--|---|
| (1) | $C_{tail}(Dequeue\ us\ sv) = length\ sv$ | (Create a cost function of tail .) |
| (2) | $A_{op}(xs) = 2$ | (Create an arbitrary cost function.) |
| (3) | $S(Dequeue\ us\ sv) = length\ us - length\ sv $ | (Create a size function for dequeue .) |
| (4) | $Dequeue\ us\ sv$ where $length\ sv = k$ | (Worst case where us is a singleton) |
| (5) | $S(Dequeue\ us\ sv) = k - 1$
$S(Dequeue\ us'\ sv') = 1$ | (Size of the next data structure can be at most 1.) |
| (6) | $C_{tail}(Dequeue\ us\ sv) = k$ | (Calculate the worst case cost of tail .) |
| (7) | $k \leq 2 + (k - 1) - 1 = k + 2$ | (As this inequality holds, the time complexity of all n instructions is $O(n)$.) |

As the time complexity of all n instructions together is $O(n)$, the amortized cost of a single instruction is $O(1)$.