Adv. Data Structures: Functional queues

Anders Ingemann (20052979)
Peter E. Hvidgaard (20062546)

December 16, 2011

Project

In this project we perform an experimental study of functional data structures. The implementations are done in Haskell, which is a lazy language. Due to this we have to make sure that our results are actually used, so the evaluations are not postponed.

We implement a queue using

1. A standard Haskell list.
2. A pair of lists, with amotized $O(1)$ gurantee if the same queue will never be argument to repeated queue operations.
3. A pair of lists, exploiting the properties of lazy evaluation of list concatenation to guranteed amortized $O(1)$ per queue operation.
4. A $O(1)$ list, with a worst-case gurantee of $O(1)$ per queue operation (if executed strictly).

We then design and perform experiements comparing the different implementations, where we cover the worst-case scenario for every queue

Remarks

IF ANY

Queue implementations

A list in Haskell is nothing more than a singly linked list. So in other words, it's cheap to add or remove the first element, but very expensive to add an element to the tail, or remove the last element. Throughout the description of the queues, we will omit talking about trivial cases for the remove operation when we have an empty queues. This is because the only sensible action is to return nothing.

Teoretiske overvejelser og overfladisk analyse af deres running times

A standard Haskell list

Using a standard Haskell list is a very simple data structure. Unfortunately it is only usable as a queue if you gurantee that it only contain a few elements. Because of how Haskell represent lists, when inserting at the end of the queue, it will have to traverse the entire list, before appending the new element. Thus the complexity for inserting is $O(n)$. Removing on the other hand, is fairly simple. You can simply remove the first element in constant time. Thus $O(1)$ for remove.

Worst-case input

The worst-case test is fairly straight forward. Simply insert n elements and that is it. Note that the lazy nature of append, require us to force strict behaviour **HOW DO WE DO THIS?**

A pair of lists

Using a pair of lists, we can obtain amotized $O(1)$ as long as we do not repeat expensive operations. The queue works by having a pair of lists, a $left$ and a $right$. When inserting, we will add the element to the head of $right$. Removing on the other hand is a bit more complicated. There are 2 cases:

1. $left$ contain 1 or more elements.

2. $left$ is empty and *right* contains 1 or more elements.

For (1), the remove operation will simply remove the head of $left$. For (2) it's a bit more involved. The list *right* is reversed, become $left$ and the first element is removed, this makes sense, because that will be the oldest element in the queue. The running time of the insert operation is still $O(1)$, however the remove operation can be $O(n)$, especially, the first time er remove an element, we will reverse *right*. If we gurantee that we do not repeat the expensive operation, we have a $O(1)$ amotized (for every element we have in the reverse operation, we will do a remove paying for that element). However, it is common in the functional paradigm to reuse functions. Therefore one should take care to not use expensive operation more than once. This also provide us with the worst-case behaviour.

Worst-case input

Just as with the single list queue, the worst-case test is straight forward: insert n elements and repeat the remove of the first element. Note for this to be true, we have to avoid memoirzation.

A pair of lists, exploiting laziness

As described in the paper *Simple and Efficient Purely Functional Queues and Deques* by Chris Okasaki, we can exploit lazyness to achieve the same $O(1)$ amotized bound, but with an improved worst-case to $O(\log n)$.

To achieve this, we want to calculate the reverse of *right* in an incremental fashion, i.e. $left$ ++ rev(*right*), where ++ is the append operation. Since append is lazy, we have to make the reverse lazy as well. To do this, we replace $<$*left*, *right*$>$ with $<$$left$ ++ rev(*right*), []$>$ periodically. This is called a rotation, denoted *rot*, and is a fusion of append and reverse. The function *rot* takes as input 3 lists, $left$, *right* and *acc* - it will then peel of the elements of $left$ one by one. For each element it will also take off an element from *right* and append it to *acc*. Assuming that $left$ and *right* have the same length, once the recursion have removed all elements from $left$, the content of *acc* will be the reversed *right*. Because append operation is lazy, the recursive calls to *rot* will be lazy as well.

The details of this can be found in the before mentioned paper.

The data structure works by maintaining an invariant, $|right| \leq |left|$. We maintain this invariant by calling *rot* whenever *right* become strictly larger than *left*, i.e. when $|right| = |left| + 1$. So, when inserting we make a call to *rot* every time we have inserted $|left| + 1$ elements. But the lazy nature of *rot* means the actual evaluation will be deferred until needed. This makes insert $O(\log 1)$. When we call remove, it is possible that *left* consists of a series of deferred *rot* calls, all of which have to be evaluated. We know that each of those calls will reduce the size of *left* in half, thus the recursion will have depth of $\log n$. Again the lazy nature of *rot* ensure that only $\log n$ work will be done, giving the $O(\log n)$ bound.

Worst-case input

We have already mentioned how *left* should be to enforce the worst-case behaviour. By inserting $1 + 2^i$ for any i will leave us with such a senarios for the first time we call remove. Note, due to memoization repeating this call will not be expensive for any successive calls. This is importent when timing several runs to gain an average.

A $O(1)$ list

Worst-case input

Experiments with worst-case

A standard Haskell list

A pair of lists

A pair of lists, exploiting laziness

A $O(1)$ list