Functional Queues

Kristoffer Just Andersen, 20051234

Troels Leth Jensen, 20051234

Morten Krogh-Jespersen, 20022362

Project 3, Advanced Data Structures 2013, Computer Science
January 2014
Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

Chapter 1

Introduction

needs content

1.1 Terminology

Purely functional
    Strict evelaution
    Lazy evaluation
    Memoization

Chapter 2

A purely functional list as a queue

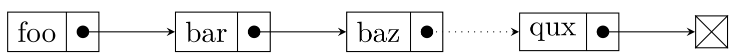

Figure 2.1: A purely functional list

Figure 2.1 shows a graphically representation of a functional list where elements can be inserted at the front by means of **cons** and the list can be traversed by following the arrows which corresponds to the **cdr** operation. The list can be constructed in constant time by a special termination construction (inductive base case) being the empty list.

Elements can be inserted and removed from the front in constant time, but all other operations require traversing the entire list. Below is all operations with the corresponding time-complexities shown:

Operation	List
makeList(x)	$O(1)$
push(x,L)	$O(1)$
pop(L)	$O(1)$
inject(x,L)	$O(n)$
eject(L)	$O(n)$
catenate(K,L)	$O(\|K\|)$

Chapter 3

Queue by a pair of lists

A simple observation to make is that if we reverse the list of section 2, INJECT(x, L) and EJECT(L) will be at the front of the list and will therefore take $\mathrm{O}(1)$ time, however, PUSH(x, L) and POP(L) will take $\mathrm{O}(n)$ time. A trick is therefore to maintain a pair of lists such that PUSH(x, L), POP(L), INJECT(x, L) and EJECT(L) for the most part perform in constant time.

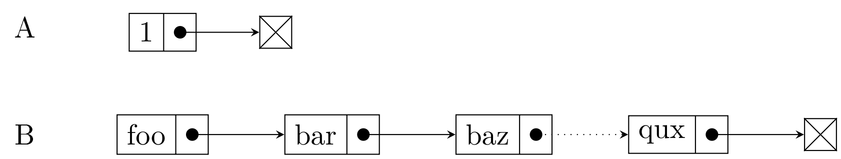

Figure 3.1: Maintaining two lists - reverse on next pop

When enqueueing nodes are pushed to the front of the list B and when elements are removed they are taken from the front of the list A which both performs in constant time. The only problem is what to do if A consists of only the empty list.

Figure 6.1 shows a configuration of the queue where a pop will result in A becoming the empty list. Subsequent pops should remove from the end of B but that would take $\mathrm{O}(n)$ for every operation. It therefore makes more sense to reverse the list once and place instead of A.

POTENTIAL

Reversing the list can be done in $\mathrm{O}(n)$ where n is the size of the list B. The length of B is the number of enqueues since the last reversal or construction of the list. By amortizing the cost of reversal over said enqueue-operations will result in all operations, except catenation, having $\mathrm{O}(1)$ time complexities:

Operation	List	Two lists (amortized)
MAKELIST(x)	$\mathrm{O}(1)$	$\mathrm{O}(1)$
PUSH(x,L)	$\mathrm{O}(1)$	$\mathrm{O}(1)$
POP(L)	$\mathrm{O}(1)$	$\mathrm{O}(1)$
INJECT(x,L)	$\mathrm{O}(n)$	$\mathrm{O}(1)$
EJECT(L)	$\mathrm{O}(n)$	$\mathrm{O}(1)$
CATENATE(K,L)	$\mathrm{O}(\lvert K\rvert)$	$\mathrm{O}(min(\lvert K\rvert, \lvert L\rvert))$

Catenation can be done in $\mathrm{O}\left(min(|K|, |L|)\right)$ although that requires maintaining the length of each queue. If K has the fewest elements. then all elements from K_B can be pushed to L_A in constant time for each node, hereafter K_B can be reversed and then pushed to L_A which in total takes linear time in the size of K. If $|L|$ is smaller then L_A can be added K_B as before and the result can be added as **cdr** to the list of L_B which in all takes linear time in the size of L. WHAT ABOUT POTENTIAL?

Chapter 4

Pair of lists with lazy evaluation

We were asked to implement a queue with lazy evaluation having amortized constant time for all operations except concatination. As so many implementations we have two lists; one representing the front and another representing the tail. At specific points we will rotate the elements from the tail onto the head by reversing the tail list and appending it. In Haskell the append and the reverse function is lazy giving us the desired evalution strategy.

The trick with two lists has already been used in the previous chapter but we will use another trigger for the lazy implementation. We rotate whenever the length of the tail list is one greater than the length of the front list.

To perform the amortized analysis we will again use the potential function, however, the potential function is no longer equivalent to the elements in the tail.

As always, it should be the case that we add potential on inserts and release potential when deleting. We therefore define the potential to be:

$$\Phi(Q) = tail(Q) + notreversed(Q)$$

Appending the reversed elements from the tail onto the head makes no change to the potential. Potential is released when a lazy list is demanded. Since any lazy list demanded will origin from the tail, the number of operations is completely specified by the amount of inserts and thus the release of potential pays for the reverting the list.

The above analysis assumes that when a demand for the first element of a lazy list not yet reversed, the list is reversed and all subsequent calls to the list will work in constant time. The latter part assumes therefore that lazy lists are memoized. We can now present the time complexities for the lazy algorithm:

Operation	List	Two lists (amortized)	Lazy (amortized)
MAKELIST(x)	O (1)	O (1)	O (1)
PUSH(x,L)	O (1)	O (1)	O (1)
POP(L)	O (1)	O (1)	O (1)
INJECT(x,L)	O (n)	O (1)	O (1)
EJECT(L)	O (n)	O (1)	O (1)
CATENATE(K,L)	O ($\|K\|$)	O ($min(\|K\|, \|L\|)$)	O (1)

Catenating two queues can be done in constant time by appending the head of L and the reversed tail of L onto the head of K because append and reverse are lazy thus the functions returns immediately. We assume that all unreleased potential of L can be transferred to K.

Chapter 5

$\mathrm{O}(1)$ lists with worst case $\mathrm{O}(1)$ enqueue and dequeue with strict evaluation

All of the queues described above have worst case running times $\mathrm{O}(n)$ which is unacceptable in some situations thus there exist a need for real times queues where all operations have worst case running times $\mathrm{O}(1)$. We have chosen to implement the Hood-Melville real time queue[REF].

There is a few key insights to this algorithm. First, reversing a list incrementally can be done by having two lists and transferring elements from one to the other. Second, one can incrementally append two lists by applying the trick. If one will would like to append two lists xs and ys, one can reverse xs to xs' and then reverse xs' onto ys. Additionally, one can reverse xs on to the reverse of ys by reversing xs and ys in parallel and then continue as before.

Initially, the queue will have two lists forming the head H and tail T and two integers describing the length of each. When we decide to rotate we use the observation above and do the following:

1. Reverse T forming the tail of the new resulting head H'
2. Reverse H into H_R
3. Reverse H_R onto H'

One should be able to convince himself (or herself) that queue order is preserved after the third step. However, the queue will not remain constant when performing the rotation (which is a weird phrase in [REF] since the incremental rotation only occurs when changes occur), but basically, it means that we have to maintain the state of the current rotation and provide the user the ability to still make alterations to the queue.

Allowing the user to enqueue elements is easily done by just having a new tail list. Dequeueing is bit harder because we are currently reversing the H into H_R. The answer is two have two lists; one being the a working copy of the old list and one being the list we are reversing. This again introduce some maintenance because removing an element from the working copy somehow has to be synchronized with the list we are reversing. To correct this, we use a counter that describe how many valid elements to be copied from H_R. A total of six lists is therefore necessary for this data structure.

The recopying (or rotation) should be completed before the first element is needed. We will rotate elements when the tail list becomes one longer than the head list, therefore, rotating elements will take $2m+1$ operations where m is the number of elements in head and tail. Therefore $2m+1$ incremental operations must be performed in at most m queue operations. By performing the two first rotation steps starting the rotation process and hereafter perform two incremental steps for each queue operation. In total, this gives $2(m+1) = 2m+2$ steps, which is greater than $2m+1$. We therefore perform constant work for every queue operation sans catenation:

Operation	List	Two lists (amortized)	Lazy (amortized)	Hood-Melville
MAKELIST(x)	$O(1)$	$O(1)$	$O(1)$	$\Theta(1)$
PUSH(x,L)	$O(1)$	$O(1)$	$O(1)$	$\Theta(1)$
POP(L)	$O(1)$	$O(1)$	$O(1)$	$\Theta(1)$
INJECT(x,L)	$O(n)$	$O(1)$	$O(1)$	$\Theta(1)$
EJECT(L)	$O(n)$	$O(1)$	$O(1)$	$\Theta(1)$
CATENATE(K,L)	$O(\lvert K\rvert)$	$O(min(\lvert K\rvert, \lvert L\rvert))$	$O(1)$	

Chapter 6

Functional lists with index lookup

This exercise was posted in the theoretical project but solving this seemed natural considering this project is all about functional data structures.

We are asked to describe a strict functional data structure supporting PUSH(x, L), POP(L) in $\mathrm{O}(1)$ and LOOKUP(d) in time $\mathrm{O}(\log n)$ where n is the number of elements in the list.

By using a standard list composed by cons we will have a PUSH(x, L) and POP(L) in constant time, so how do we solve lookups in $\mathrm{O}(\log n)$? A binary tree would give us the desired running times. The problem is how to combine the two. Okasaki solved this problem by having a collection of complete binary trees.

For each entry in our list we store a tuple consisting of the size of the tree and a complete binary tree. Thus the size of the trees is on the form $2^k - 1$. Whenever two adjacent elements are at the front of the list and another element is pushed, the trees are joined with the newly added element as the root which gives us that the elements in any of the binary trees is stored in preorder. Elements 1 to $\lfloor n/2 \rfloor$ will be in the left part of the tree and $\lceil n/2 \rceil$ to $n-1$ will be in the right part.

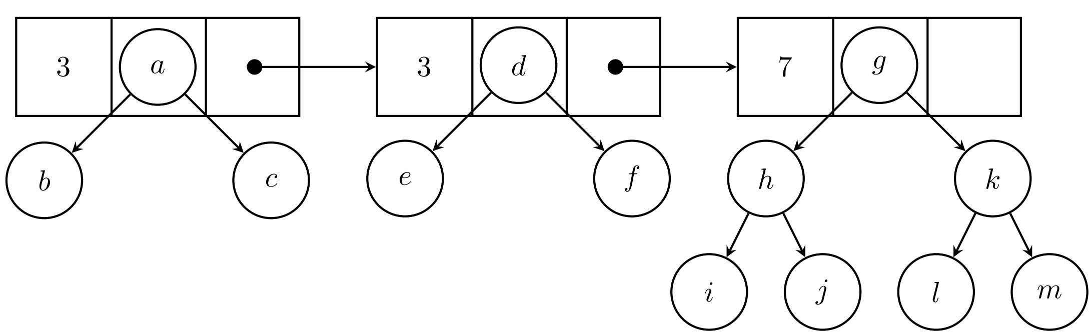

Figure 6.1: A list consisting of three complete binary trees, two of size 3 and one of size 7. An added node would result in two complete binary trees of size 7 and an additional node would result in one tree of size 15.

Inserting new elements can at most join two trees. Since the left child of the resulting tree will be the first tree in the list and the right child will be the second tree, the tree can be constructed in constant time. Similarly, it can be destructed into two parts by just removing the root and cons the right child followed by the left child onto the list.

Looking up and element by index can be done by simply iterating the list until the containing tree has been found. If the index to search for is smaller than the size of the tree at the current position said tree will contain the element, otherwise subtract the size from the index and continue searching.

If we only have on complete binary tree looking up an element can clearly be done in $\mathrm{O}(\log n)$ time so we have to show that the length of the list will never be longer than $\log n$.

An integer on the form $2^k - 1$ is called a skew-binary term, and if t is such a term then the next skew-binary term will be $2t + 1$. A decomposition of an integer n greater or equal to zero will be a multiset of skew-binary terms $\{t_1, t_2, \cdots, t_m\}$ where $n = t_1 + t_2 + \cdots + t_m$. Such a decomposition is said to be greedy if the largest term is as large as possible and the remaining part of the decomposition is greedy as well. In other words a collection of skew-binary terms is greedy if no subset of lesser terms sums to a larger term. We will describe such a greedy decomposition of n as $G(n)$.

Property 1. *Every integer $n \geq 0$ has a unique greedy decomposition.*

Proof. $G(0)$ will have the empty set which is unique. For $G(n)$ where $n > 0$, we know, that no subset of skew-binary terms may sum to a larger term thus the it must include the largest possible term t such that $t \leq n$. We add t to the set and continue searching for $G(n - t)$. Notice, that no subset of $G(n - t)$ can sum to t otherwise it would not be the largest term respecting $t \leq n$. □

Property 2. *A skew-binary decomposition is greedy iif every term is unique except the two smallest: $t_1 \leq t_2 < t_3 < \cdots < t_m$.*

Proof.
⇒ We have a greedy decomposition and say we have two terms of size $2^k - 1$ and a third term of equal or lesser size. Then the three terms would be greater or equal to another term because the next term would be at least as large as $2(2^k - 1) + 1$. But then this would not be greedy and we have a contradiction.
⇐ Again, because the next skew-binary term for any term t is on the form $2t + 1$, no two terms can sum up to another term if we only repeat the smallest ones twice and all others are unique. The decomposition would therefore be greedy. □

Theorem 1. $|G(n)| \leq \lceil \log(n + 1) \rceil$.

Proof. Clearly if $n = 0$ it holds so we turn our attention to $n > 0$. Assume, to obtain a contradiction, that $|G(n)|$ contains more than $k = \lceil \log(n + 1) \rceil$. Then, by 2 and by how a greedy decomposition is defined, it must be that $G(n)$ contains a term at least as large as $2^k - 1 \geq n$. Then, because $|G(n)|$ contains more than k terms there is at least one more term and therefore the sum of terms in $G(n)$ will strictly exceed n and we obtain a contradiction. □

Except for the two first trees in our list every size of the complete binary trees in the list is unique and in increasing order by construction. Therefore, the length of the list can not exceed $\lceil \log(n+1) \rceil$ by 1. Therefore, lookup will traverse at most $\mathrm{O}(\log n)$ elements in the list and perform a search in a single complete binary tree of height at most $\log n$.