A truly efficient implementation of LRU stacks

0 Operations on an LRU stack

We consider a finite and nonempty list of natural numbers in the range $[0, N)$, for some given constant N. All numbers in the list are assumed to be different. The problem to be solved is the implementation of this list and the following three operations on it:

(0) return (the value of) the *last* –right most– element of the list;

(1) delete a given number from the list;

(2) add a given number to the list in such a way that it becomes the list's *first* –left most– element.

I assume that these operations are used in such a way that the list remains nonempty, so operation (0) is well-defined. I also assume that numbers to be deleted do occur in the list and that numbers to be added do not occur in the list. As a consequence of the latter, all numbers in the list remain different. Put differently, each number from the range $[0, N)$ occurs in the lists *at most once*; therefore, every list element is uniquely identified by its value. We shall exploit this in the following solution.

* * *

Operation (0) is the only one by means of which the list can be *inspected*: if it were not for operation (0) there would be no need to store any information at all. Its implementation is easy: we use a variable L to hold the *last* element of the list. Operation (0) then boils down to returning the value of L.

As far as variable L is concerned deletion of a number n, $0 \leq n < N$, from the list amounts to:

```
if  n=L  →  L := "the predecessor of L"
[]  n≠L  →  skip
fi
```

Because the list is assumed to remain nonempty deletions only take place when the list intially has at least 2 elements; hence, "the predecessor of L" exists. For its implementation we introduce an array $p(i : 0 \leq i < N)$ with the following interpretation. For every number i occurring in the list, p satisfies:

$$p[i] = \text{"the predecessor of } i\text{"} \quad \vee \quad i = F \quad ,$$

where variable F, which we will need anyway, represents the *first* element of the list. For numbers i not occurring in the list, $p[i]$ is irrelevant, as is the value of $p[F]$. Notice that the viability of this representation crucially depends on the fact that all list elements are different: here we exploit that every list element is uniquely identified by its value.

Now $L :=$ "the predecessor of L" can be encoded as $L := p[L]$, provided of course that $L \neq F$; because delete operations are only performed on lists with at least two elements, $L \neq F$ is indeed a precondion of the delete operation.

The introduction of variable p brings about the obligation to update it whenever the list is changed. The effect of deleting element i is that the unique element of which i was the predecessor, the *successor* of i, has no longer i as its predecessor: after the deletion, the predecessor of i's successor is $p[i]$ instead of i. So, we need an administration of the successors as well and we introduce an array $s(i : 0 \leq i < N)$ with the following interpretation. For every number i occurring in the list, s satisfies:

$$s[i] = \text{"the successor of } i\text{"} \quad \vee \quad i = L \quad .$$

In a very similar way –as far as deletion is concerned, the problem is symmetric– we obtain the obligation to update variables s and F. Thus we obtain the following program fragment for deletion of number n:

$$\begin{array}{lll} \textsf{if} \;\; n{=}L & \rightarrow & L := p[L] \\ [] \;\; n{\neq}L \wedge n{\neq}F & \rightarrow & p[s[n]] := p[n] \;\; ; \;\; s[p[n]] := s[n] \\ [] \;\; n{=}F & \rightarrow & F := s[F] \\ \textsf{fi} & & \end{array}$$

The implementation of operation (2) in terms of this representation poses no problems whatsoever; addition of number n as the list's first element boils down to:

$$p[F] := n \;\; ; \;\; s[n] := F \;\; ; \;\; F := n$$

* * *

The datastructure consisting of the two arrays p, s and the numbers F, L is, of course, known as a *doubly linked list*. The above shows that we need not know this, because the whole design is of the kind only-one-thing-you-can-do. The three resulting programs are simple and obviously have $\mathcal{O}(1)$ time complexity.

In LRU-applications of this datastructure yet another operation is needed, which could be called an *update* of list element n, but this is just deletion of n followed by addition of n.

That I have used natural numbers is not very relevant: for every i, the values $p[i]$ and $s[i]$ could also be stored together as a record, and i could then be considered as a *pointer* to that record.

1 Epilogue

I designed this datastructure some 8 years ago, to use it in a (so-called) disk-cache program. I would never dream of devoting a publication to it, simply because I consider the design exercise as elementary and well within reach of any competent programmer. (The problem is so simple that a formal treatment of it, which is certainly possible, would be overdoing it.)

Surprisingly enough, in a recent publication [1] L. Barriga and R. Ayani present a solution for the same problem. They claim their solution to be efficient, but it is not and it is needlessly complicated. (Besides, they only support their claim by performance *measurements* instead of a performance *analysis*.) My solution shows that their paper should never have been written.

References

[1] L. Barriga and R. Ayani, Lazy update: An efficient implementation of LRU stacks, *Information Processing Letters* **54** (1995) 81–84.

Eindhoven, 11 may 1995

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven