Assignment 1 – Selected Model Answers

EXERCISE 1. For $k \in \mathbb{N}, k \geq 1$ define an operation $delete\_last(\ell, k)$, which deletes the last k elements in a list ℓ. Analyse the amortised complexity of this operation and show that it is in $\Theta(1)$, independent of k.

SOLUTION. For the deletion of the last k elements of a list (represented by an array) it suffices to decrement the *length* variable by k (or set *length* to 0 in case $k > length$). As a single execution of $delete\_last(\ell, k)$ may trigger a cascade of *deallocate* operations, it is impossible to distribute the costs for *deallocate* to *delete* and *delete_last* operations.

Instead let us modify the contribution of *append* and *insert* to become $3c$ (instead of $2c$), and $3c|\ell|$ for *concat*. In order to simply the argumentation let us use a second counter C', and let the contribution to C be still $2c$ (or $3c|\ell|$, respectively), while the remaining contribution of c (or $c|\ell|$, respectively) is added to C'.

Consequently, when the list has reached the length m, there has been at least the contribution mc to the counter C'. If we execute now an operation $delete\_last(\ell, k)$, the triggered *deallocate* operations require a cost of c for each element in the list. Subtracting these costs from C' still maintains the condition that the value of C' is c-times the actual size of the list, in particular ≥ 0.

In doing so we can have a contribution of 0 by the $delete\_last(\ell, k)$ operation, which shows that the amortised complexity is in $\Theta(1)$.

EXERCISE 2. Explore structural recursion on list objects, i.e. define an operation $src[e, f, g]$ in the following way:

- If ℓ is the empty list, then $src[e, f, g](\ell) = e$, where $e \in T'$ is some constant.
- If ℓ is a singleton list containing just one element x, then $src[e, f, g](\ell) = f(x)$, where f is a function that maps elements of a set T (the set of list elements) to elements of a set T'.
- If ℓ can be written as the concatenation of two lists, say $\ell = concat(\ell_1, \ell_2)$, then

  $$src[e, f, g](\ell) = g(src[e, f, g](\ell_1), src[e, f, g](\ell_2)) ,$$

  i.e. apply structural recursion to both sublists separately, then apply the operation $g : T' \times T' \to T'$ to the resulting pair.

(i) Discuss the conditions, under which $src[e, f, g]$ is well-defined.

(ii) Show how to use structural recursion to define operations on lists such as

- determining the length,
- applying a function to all elements of a list, and
- creating a sublist of list elements satisfying a condition φ.

(iii) Analyse the time complexity of structural recursion.

SOLUTION.

(i) From the definition of $src[e, f, g]$ we see that for a list $\ell = [x_1, \dots, x_n]$ we have to apply f to all x_i, then apply $g(f(x_i), f(x_j))$ (or $g(f(x_i), e)$) consecutively. The result is uniquely determined, if g is associative with neutral element e.

Conversely,

$$g(src[e, f, g](\ell), e) = g(src[e, f, g](\ell), src[e, f, g]([])) = src[e, f, g](\ell) ,$$

so if $src[e, f, g]$ is surjective, e must be a neutral element of g. Furthermore,

$$\begin{aligned}
&g(src[e, f, g](\ell_1), g(src[e, f, g](\ell_2), src[e, f, g](\ell_3))) \\
&\qquad = g(src[e, f, g](\ell_1), src[e, f, g](\ell_2 + \ell_3)) \\
&\qquad = src[e, f, g](\ell_1 + (\ell_2 + \ell_3)) \\
&\qquad = g(src[e, f, g](\ell_1 + \ell_2), src[e, f, g](\ell_3)) \\
&\qquad = g(g(src[e, f, g](\ell_1), src[e, f, g](\ell_2)), src[e, f, g](\ell_3))
\end{aligned}$$

so if $src[e, f, g]$ is surjective, g must be associative.

(ii) For the *length* function we have

$$\mathit{length}(\ell) = src[0, 1, +](\ell) = \begin{cases} 0 & \text{if } \ell = [] \\ 1 & \text{if } \ell = [x] \\ \mathit{length}(\ell_1) + \mathit{length}(\ell_2) & \text{if } \ell = \ell_1 + \ell_2 \end{cases}$$

For the $\mathit{map}[h]$ function applying h to every list element we have

$$\mathit{map}[h](\ell) = src[[], \mathit{single} \circ h, +](\ell) = \begin{cases} [] & \text{if } \ell = [] \\ [h(x)] & \text{if } \ell = [x] \\ \mathit{map}[h](\ell_1) + \mathit{map}[h](\ell_2) & \text{if } \ell = \ell_1 + \ell_2 \end{cases}$$

For the $\mathit{filter}[\varphi]$ function selecting the sublist of those elements x satisfying φ we have

$$\mathit{filter}[\varphi](\ell) = src[[], \alpha, +](\ell) = \begin{cases} [] & \text{if } \ell = [] \\ \alpha(x) & \text{if } \ell = [x] \\ \mathit{filter}[\varphi](\ell_1) + \mathit{filter}[\varphi](\ell_2) & \text{if } \ell = \ell_1 + \ell_2 \end{cases}$$

using the function

$$\alpha(x) = \begin{cases} [x] & \text{if } \varphi(x) \\ [] & \text{else} \end{cases} .$$

(iii) If n is the length of the input list, we have to apply the function f n-times, so if m is the size of a list element and $\bar{f}(x)$ is the complexity of computing $f(x)$, we get a complexity in $O(m \cdot n)$ for the applications of f. Furthermore, we have to apply g $(n-1)$ times, so if *barg* is a function measuring the complexity of computing g, this amounts to a complexity in $O(\sum_{i=1}^{n-1} \bar{g}(y_i))$, where the y_i are a measure of the sizes of the different inputs to g.

If we have a bound c on the size of elements in T or T', the total complexity amounts to $O(mn + n) = O(mn)$.

EXERCISE 3. Explain how to implement a FIFO queue using two stacks so that each FIFO operation takes amortised constant time.

SOLUTION. Let the two stacks be $s1$ and $s2$. For the *pushback* operation—accordingly also for *back*—we use only $s1$. If we have a *popfront* operation, we have to distinguish two cases:

Case 1. If s_2 is empty, we first move all elements from $s1$ to $s2$, where a move consists of a *pop* operation on $s1$ followed by a *push* operation on $s2$. Then proceed as in Case 2.

Case 2. If $s2$ is not empty, then the *popfront* operation becomes simply a *pop* operation on $s2$.

The operation *topfront* is handled analogously without changing $s2$ is the second case. The emptiness check amounts to checking, if both stacks are empty.

For the amortised complexity let each *pushback* operation make a constant contribution $2c$ to a counter C, where c is the cost required by any *push* or *pop* operation. Thus, moving all elements from $s1$ to $s2$ in Case 1 above requires the cost of $2nc$, where n is the number of elements moved.

When a move becomes necessary, there have been n *pushback* operations since the latest move, which altogether contributed $2nc$ to the counter C. By executing the move the counter is reset to 0. The constant contribution of the *pushback* operation guarantees that its amortised complexity remains in $\Theta(1)$. No other operation is affected.

EXERCISE 4.

(i) Implement the function *delete_last*(ℓ, k) from Exercise 1 on the class ALIST.

(ii) Implement your solution from Exercise 3 using the class STACK.

SOLUTION.

See the C++ header and program files in `Ass1_Ex4isolution.zip` and `Ass1_Ex4iisolution.zip`.