Assignment 1 – Selected Model Answers

EXERCISE 1. For $k \in \mathbb{N}, k \geq 1$ define an operation $\mathit{delete\_last}(\ell, k)$, which deletes the last k list elements in ℓ.

(i) Analyse the amortised complexity of this operation and show that it is in $\Theta(1)$, independent of k.

(ii) Implement a method $\mathit{delete\_last}(\ell, k)$ on the class ALIST or DLIST.

SOLUTION.

(i) 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 $k > \mathit{length}$). As a single execution of $\mathit{delete\_last}(\ell, k)$ 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 $\mathit{delete\_last}(\ell, k)$, 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 $\mathit{delete\_last}(\ell, k)$ operation, which shows that the amortised complexity is in $\Theta(1)$.

(ii) We omit the C++ code.

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 $\ell = \mathit{concat}(\ell_1, \ell_2)$, 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—only consider the operation, if it is well-defined.

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

- determining the length (if not stored),
- 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.

(iv) Implement structural recursion on either ALIST or DLIST.

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

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

For the $map[h]$ function applying h to every list element we have

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

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

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

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 $O(mn + n) = O(mn)$.

(iv) We omit the C++ code.

Exercise 3. Consider a sequence data structure, where we are interested in just the following four operations:

- *pushback*(ℓ, x) appends the element x at the end of the sequence;
- *pushfront*(ℓ, x) adds the element x at the front of the sequence;
- *popback*(ℓ) removes the last element x of the sequence and returns it;
- *popfront*(ℓ) removes the first element x of the sequence and returns it.

(i) Show how these operations can be expressed by the operations studied in the lectures and labs.

(ii) Provide a different implementation of a class with these methods in **C++** with improved (amortised) complexity.

Solution.

(i) *pushback*(ℓ, x) is the same as $append(\ell, x)$, and *popback*(ℓ) is the same as the sequence $y := get(\ell, length); delete(\ell, length)$.

pushfront(ℓ, x) is the same as $insert(\ell, 1, x)$, and *popfront*(ℓ) is the same as the sequence $y := get(\ell, 1); delete(\ell, 1)$.

(ii) We will see that these operations are the common deque operations. By using circular arrays the amortised complexity of *pushfront* and *popfront* can be reduced to become constant. Circular arrays are handled later in the lectures. We omit the C++ code here.

Exercise 4.

(i) Implement a function *selectionSort* on the class AList or DList for the selection sort algorithm.

(ii) Implement a function *insertionSort* on the the other of these two classes for the insertion sort algorithm. Use binary search in the inner loop of the method to minimise the number of comparisons.

(iii) Recall the definition of the bubble sort algorithm. In a nutshell, as long as there are list elements $list(i) \geq list(j)$ with $i < j$, swap them until the list is ordered. Implement the bubble sort algorithm by a function *bubbleSort* on both classes ALIST and DLIST.

SOLUTION.

This is a pure programming exercise. We omit the C++ code here.