58131 Data Structures
II exercise, week 42/2003, English translation

Exercise II.1: If we know in advance that a stack will contain at most N items at a time, then we can implements the stack without pointers by using an array $A[0 \ldots N-1]$. How?

(a) Write the corresponding stack operations in pseudocode. Show that their asymptotic time requirements are the same $\mathcal{O}(1)$ as the pointer-based version.

(b) Do part (a) for queues as well.

Exercise II.2: Consider a text written only with letters `a`,`b`,`c`,...,`z` and two kinds of parentheses: `(` together with `)`, and `[` together with `]`.

The text is *properly parenthesized* if these parenthesis pairs have been used in the natural way: an opening parenthesis `(` (`[`, respectfully) is later paired with the corresponding closing parenthesis `)` (`]`, respectfully), and the part of the text between these parentheses is also properly parenthesized.

Develop an algorithm to decide whether or not a given text is properly parenthesized. The text is given to your algorithm as a singly linked list of individual characters. Your algorithm must work in

$$\mathcal{O}(\text{length of the list})$$

time and space.

What kind of auxiliary data structure would you choose for your algorithm? Show also why your algorithm works the way it should and within the resources it should.

Exercise II.3: Lecture slides 4.4 presented the data structure *dequeue* as a doubly linked list, where additions and deletions are allowed at both ends.

Develop this presentation further, so that it supports also the operation `reverse`(dequeue) in the same $\mathcal{O}(1)$ time as its original operations. This new operation causes the dequeue to be in the *reverse* order: the left end becomes the right end, and vice versa, and so on.

Write pseudocode for all these operations.

Exercise II.4: Let $\oplus$ be any 2-place arithmetic operation: addition, subtraction, multiplication,...

(a) The *left fold* of a given list $L = (p_1, p_2, p_3, \ldots, p_n)$ of numbers is

$$(\ldots(((0 \oplus p_1) \oplus p_2) \oplus p_3)\ldots) \oplus p_n.$$

Write pseudocode for an algorithm which calculates the left fold given the list L as input. Show that it works correctly.

What are the time and memory requirements of your algorithm in terms of the length n of the input list L?

(b) Do part (a) also for the *right* fold defined as

$$p_1 \oplus (p_2 \oplus (p_3 \oplus (\ldots (p_n \oplus 0)\ldots))).$$

(c) Could such folding algorithms be useful in practical programming?

Exercise II.5: Lecture slides 4.6 presented a method for turning a recursive function into a loop and a stack. (Unfortunately this material is not in the book.)

(a) Apply the method to the following *Ackermann's function*[1]:
function Ack(m, n: $\mathbb{N}$): $\mathbb{N}$
  if $m = 0$ **then**
    return $n + 1$
  else if $n = 0$ **then**
    return Ack$(m - 1, 1)$
  else
    return Ack$(m - 1,$ Ack$(m, n - 1))$
  end if.

Optimize the tail-recursive calls.

(b) The code obtained in part (a) can be further improved by hand. In which ways? What is its final form?

(c) The method only applies to functions whose body is just an **if**-clause. How would you extend it to handle also (possibly nested) **while**-loops?

(d) How about the following *mutually* recursive functions?
function $f(x_1, x_2, x_3, \ldots, x_p)$
  Body which can call not only f but also g**.**

function $g(y_1, y_2, y_3, \ldots, y_q)$
  Body which can call not only g but also f**.**

(e) How about the following *nested* functions?
function $f(x_1, x_2, x_3, \ldots, x_p)$
  function $g(y_1, y_2, y_3, \ldots, y_q)$
    Body of inner function g**.**
  Body of outer function f**.**

Both bodies can call both functions and refer to their own parameters. In addition, the body of the inner function g can refer to the parameters x_i of the outer function f, but the body of the outer function f is not allowed to refer to the parameters y_j of the inner function.

(Total number of exercises: 5 pcs.)

[1] Different books define its details in slightly different ways.