Exercises: Stacks and Queues (Theory)

Stacks:

(1) Modify the array implementation of a stack given so that (1) the bottom will be anchored to the low-indexed end of the array (position 1), and (2) the top will be the higher-indexed element of the array. Analyze the efficiency of the operations.

(2) Modify the array implementation of a stack so that the top will always be the first position of the array. Compare its running time with the standard implementation.

(3) Extend the specification of stack to include a function called *searchstack* that searches for a given element on a stack and returns its position in relation with the top. Write the function. Consider the case in which the element is not found. Provide the algorithm for the array implementation as well as for the pointer implementation. Provide also an implementation based on the standard operations of a stack. Estimate the running time of each version to determine the best option.

(4) Extend the specification of stack to include a function called *extractfromstack* that searches and returns an element *e* from a stack. Write that function. Any other element must remain on the stack respecting their order. Consider the case in which the element is not found. Provide algorithms for the array implementation as well as for the pointer implementation. Provide also an implementation based on the standard operations of a stack. Estimate the running time of each version to determine the best option.

(5) Write a program that checks whether an input string is a palindrome or not using a stack.

(6) Write an algorithm for evaluating a postfix expression using a stack.

(7) Write an algorithm for evaluating an infix expression (not necessarily fully parenthesized) using stacks. The priority of operators shall be taken into account. Describe the input, stack and operation performed on each iteration to evaluate the expression `(2 * 5 - 1 * 2) / (11 - 9)`.
Tip: You would probably need two stacks (a stack of numbers and stack of characters)

(8) Write a program for evaluating a prefix expression. Describe the input, stack and operations performed on each iteration to evaluate the expression `/ - * 2 5 * 1 2 - 11 9`.

(9) Write an algorithm for converting a non-parenthesized infix expression to a postfix expression. The priority of operators shall be considered. Show the steps to convert the expression `a+b*c-d`.

(10) It is possible to keep two stacks in a single array, if one grows from position 1 of the array, and the other grows from the last position. Write a specification of the ADT. Write a function *push(x, s)* that pushes element *x* into stack *s*, where *s* is one or the other stack. Include all error checks in your function.

Queues:

(1) Define an ADT to support the operations *enqueue*, *dequeue* and *onqueue*. *onqueue(x)* is a function returning true or false depending on whether *x* is on the queue. Write also *onqueue* and compute its running time.

(2) Rewrite the *makenull* operation of the queue ADT so that any element existing on the queue is conveniently disposed. Dequeue operation can be reused. What is the running time of the new *makenull* operation.

(3) Extend the ADT queue to include the following operations:
   - *last* that will return the rear element of the queue (which shall not be deleted from the queue)
   - *length* that will return the number of elements on the queue.
   - *enqueueN* that will enqueue an array of N elements on the queue
   - *dequeueN* that will dequeue and return an array of N elements from the queue

   Write an implementation for each operation and compute their running time.

(4) Extend the ADT queue to include an operation called *searchqueue* that searches for an item on the queue and returns its position. Write a program for the new function and compute its running time.

(5) Extend the specification of queue to include a function called *extractfromqueue* that searches and returns an element e from a queue. Implement that function. Any other element must remain on the queue respecting their order. Consider the case in which the element is not found. Provide also an implementation based on the standard queue operations. Estimate the running time of each version to determine the best option.

(6) Provide a pointer implementation of a queue that does not kept a reference to the rear element (i.e. that does not have any pointer to the rear element). Compare the running time of each operation with the standard implementation.

(7) Implement the ADT that provides a circular implementation of a queue. Extend the queue ADT to include a function that returns the number of elements on the queue. Implement such a function.

(8) Another linked-list implementation of queues is to use a dummy cell. A dummy cell is an empty cell that always is in the first position of the queue. If the queue is empty then *front* and *rear* point to the dummy cell. Implement the queue operations for this representation. How does this implementation compare with the list implementation given in class in terms of speed (running time) and space utilization?

(9) A *dqueue* (doubled-ended queue) is a list from which elements can be inserted or deleted at either end. Present a specification for the *dqueue* ADT and develop array as well as pointer implementations of it.