CS119 – Module 12: Queues and Stacks

Purpose: Both the Queue and Stack ADTs are used throughout computer science. In this module you will examine the Queue ADT and then implement the Stack ADT in order to check if parentheses are matched properly in a string.

Knowledge: This module will help you become familiar with the following content knowledge:

- the Queue ADT
- the Stack ADT
- analysis of the Stack and Queue operations

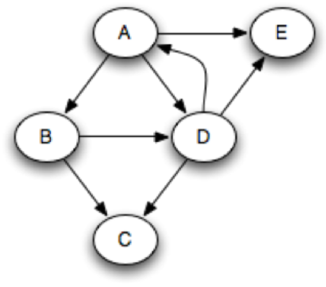
Activity: With your group perform the following tasks and answer the questions. You will need to use files in the `lab12` directory. You will be reporting your answers back to the class in 30 minutes.

Consider the abstract data types Queue and Stack:

Function	Explanation
`empty :: Queue a`	gives an empty queue
`isEmpty :: Queue a -> Bool`	determines if a queue is empty
`front :: Queue a -> a`	gives the item that is at the front of the queue
`enqueue :: a -> Queue a -> Queue a`	adds an item to the rear of the queue
`dequeue :: Queue a -> Queue a`	removes an item from the front of the queue

Function	Explanation
`empty :: Stack a`	gives an empty stack
`isEmpty :: Stack a -> Bool`	determines if a stack is empty
`top :: Stack a -> a`	gives the item that is at the top of the stack
`push :: a -> Stack a -> Stack a`	adds an item to the top of the stack
`pop :: Stack a -> Stack a`	removes an item from the top of the stack

1. A queue is a linear data structure which is FIFO (first in first out). A Stack is LIFO (last in first out). For each of the tasks below, determine whether we would want to use a stack, queue, or either.
   (a) A language runtime system that handles recursive function calls
   (b) Undo system in a text editor
   (c) A system for handling waiting printer jobs for a shared printer
   (d) Page visited history in a web browser for the back button
   (e) Operating system handling multiple processes that need execution
   (f) Visiting all the pages on a website via the hyperlinks



2. Let's examine visiting the pages on a website. Suppose we have a website with five pages A, B, C, D, and E. The diagram below indicates the hyperlinks on the pages. So page A has hyperlinks to B, D, and E.

   Consider the following algorithm:

```
visitPages :: Website a -> [a]
visitPages website = (startPage website) :
                     visit (addHyperlinks (startPage website) empty) where
        visit q = if (isEmpty q) then
                           []
                  else
                           (front q) : visit (addHyperlinks (front q) (dequeue q))

addHyperlinks page q = <code which enqueues the unvisited pages reached
                                     from this page>
```

   Complete the trace of this algorithm with the website above with start page A. (I am using `[[ ]]` to illustrate the queue ADT in the trace):

   `A : (visit [[B,D,E]])` – `visitPages` on the website creates a list starting with the starting page and the rest of the list is created by the call to `visit` on the hyperlinks

   `A : B : (visit [[D,E,C]])` – B is removed from the queue and the unvisited hyperlinks from B are added to the queue in the recursive call

   `A : B : D: (visit______________________________)`

   `A : B : D: ____ (visit______________________________)`

   `[A, B, D, ____ ,____]`

3. Change the algorithm so that instead of using a Queue it uses a Stack and trace

through it again, giving the list which would be produced.

4. Suppose that we decide to implement the Queue ADT with a list of elements as provided in the first implementation in the notes . If we have a queue with n elements, in the worst case how many elements would have to be examined in the `dequeue` operation?

   How many elements would have to be examined in the `enqueue` operation?

   Give the order of growth for both of these operations with this implementation.

5. Suppose that we decide to implement the Queue ADT with two lists of elements as provided in the second implementation in the notes . If we have a queue with n elements, in the worst case how many elements would have to be examined in the `dequeue` operation?

   How many elements would have to be examined in the `enqueue` operation?

   Give the order of growth for both of these operations with this implementation.

Complete the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

Assignment 1:
In a separate file, create a module for the ADT `Stack` and write the operations `empty`, `isEmpty`, `top`, `push`, and `pop`.

Criteria for Success: In the Example12 file import your module. Create an empty stack and verify that it is empty with the `isEmpty` function. Create a stack by pushing a couple of values on the stack and verify that it is not empty and that the top of the stack is correct. Pop a value off the stack and verify the new top. Check that repeatedly popping off all the values leads to an empty stack.

Assignment 2:
In the Example12 file implement the function
`matchingParens :: String -> Bool`

This function determines if the parentheses in the string are balanced. For example, "(()())" is balanced but "()(()" is not. We can check for balance by marching through the characters and pushing a left paren onto a stack when encountered and popping a paren off the stack on a right paren of the matching type.

To accomplish this you will want to write a helper function which takes two parameters: your string and a `Stack Char`. This function will return a boolean value on whether or not the string contains matching parens. Then all that `matchingParens` needs to do is call this helper function, passing in an empty stack.

Important: To maintain abstraction this function should not be written in the Stack.hs file nor need to know how the stack is implemented.

Criteria for Success: Test your function on "(()())", "()(()", and "())" and verify your results.

Submit all your files in Canvas for grading.