Algorithms and Data Structures - Assignment 2

Queues and stacks

Task A) Theory questions

1) Which of these real-world examples correspond to a stack?

a. Waiting at the check-in line at the airport.
b. Undo a change in a text editor.
c. Taking a cookie from a box full of snacks.
d. Putting items into a shopping bag one above another.

2) What is the main reason to have a runtime stack?

a. Keep track of the point to which each active subroutine should return control when it finishes executing.
b. Keep track of the used memory by the program.
c. To calculate the CPU usage.
d. To communicate with the operating system.

3) What happens when the function dequeue() is called?

Which function? Which class?

a. An object at the rear of a queue is inserted.
b. An object at the front of a queue is inserted.
c. The element at the front of a queue is removed.
d. The element at the rear of a queue is removed.

4) Which function is not an accessor function?

a. size()
b. isEmpty()
c. front()
d. dequeue()

5) Which of the following statements are true for an array based stack?

a. Arrays are always dynamic and the stack could grow to infinity.
b. An initial index is necessary to build an array based stack.
c. Each time we use push() the index has to be incremented.
d. While using top() the index has to be decremented.

6)Which of these complexity orders is wrong?

a. $O(\log n) \subset O(n) \subset O(n^3)$
b. $O(n \log n) \subset O(n) \subset O(n)$
c. $O(1) \subset O(n^3) \subset O(3^n)$
d. $O(\log \log n) \subset O(\log n) \subset O(n)$

7) On which position stands the active element of an array-based stack of length n?

Huh? What's the active element?

a. 0
b. n
c. $n+1$
d. $n-1$

8) Which of the following denote the way elements in the stack are accessed?

a. LIFO
b. FIFO
c. FILO
d. LILO

Task B) Array-based stack

Note: in this and following exercises, the given skeleton code may at first not even compile without errors (because of lacking declarations, etc.)! Properly completing the tasks means everything will work.

We have a football card game and the deck of the game is implemented as a stack. We are not interested in the mechanics of the game, but only in the functionality of the deck. A stack provides these functions: constructor, destructor, pop(), push(), top(), empty() and size(). The corresponding files for this task are in the *Task B* folder.

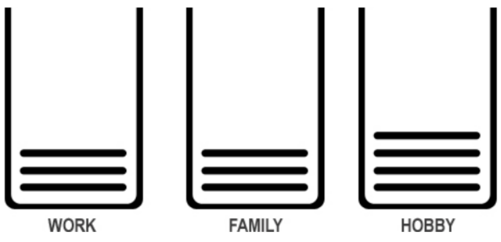
1) Implementation

The header file is already provided and *main.cpp* is filled with code to test the card deck with FC Barcelona. Your exercise consists in implementing all the functions in *stack_ ab.cpp.*

Hint: to handle stack over/underflowing, the exception classes *OverFlow()* and *UnderFlow()* are already implemented. You have to *throw* them whenever necessary.

Task C) ADT stacks

The aim of this task is to get an overview of the ADT Stack. You have three trays to classify your received letters, each for its own specific purpose (*work*, *family* or *hobby*).



Every letter belongs to one of these categories. However, your assistant made a mess with your mails while you were in holidays. This exercise deals with sorting them again.

1) printTrays()

In the file *main.cpp* (folder *Task C*) you have the skeleton of a method named **printTrays()** which should print a simple overview of the trays with the letters in it on the standard output.

2) cleanUp()

There you have also a function skeleton named **cleanUp()** which should clean up the mess your assistant made. After this method has been called, every letter should be in its correct tray.

Hint: use a temporary tray to sort the letters.

3) Ordering by priority

Finally, complete the function **orderPriority(std::stack<letter*> tray_)** for ordering the letters in each stack by priority so that the letter with the highest priority is on top.

Task D) Array-based queue

Imagine you want to develop your own simple operating system. You have already reached the point where you are coding the process management system of your OS. You have decided to use a simple array-based queue to manage the processes waiting to get access to the CPU. In this exercise, you will have to program some of the queue operations. The corresponding files for this task are in the *Task D* folder.

1) get_size() and is_empty()

In the file *process_queue.cpp* you have two function skeletons: **get_size()**, which should return the number of elements contained in the queue, and **is_empty()**, which should return a boolean value indicating if the queue is empty or not. Complete these two functions accordingly.

2) Adding elements to the queue

The skeleton of the method **add(process* my_process)** is given. The method is supposed to add the process pointed by *my_process* to the queue. Complete the function correctly, and do not forget to update class variables when necessary.

3) Removing elements from the queue

The skeleton function **remove()** is given. Complete it so that it removes the first process of the queue and returns a pointer to that removed process.

Hint: be careful with special cases and consider them too (removing from an empty queue). Also remember that after removing an element, you have to shift one position the remaining elements in the queue.

Task E) More queues

In this exercise we will learn how to work with C++ STL queues. The corresponding files for this task are in the folder *Task E*. You can find relevant information on C++ queues at *http://www.cplusplus.com/reference/stl/queue/*

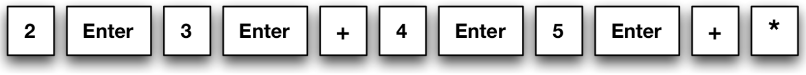
1) Traffic light

Imagine a traffic light in front of a crossroad. The arriving cars will pass the crossroad if the traffic light is green, and they will wait if the light is red. Your goal is to implement the line of cars with the help of C++ STL queues. You are already given the files *Car.cpp* and *Car.h*, that do not need further modifications.

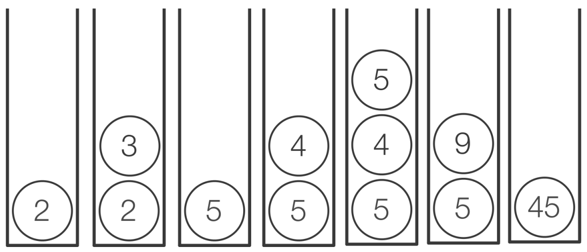
The street in our example has two lanes, so you have to use two queues. An arriving car will always wait in the shortest line. After every new arriving car, the user must have the option of switching the traffic light. As soon as the traffic light turns green, all waiting cars will leave the queue. The user can terminate the program by typing the command "stop". Simulate this behavior by filling *Street.cpp*.

Task F) The Reverse Polish Calculator

Some calculators can be used in the Reverse Polish mode. This feature, which allows to calculate long expressions without having to use parentheses, employs internally a stack (empty, at the beginning). The user can at each step enter a number, an operator or hit *Enter*. Hitting *Enter* after inputting a number puts it on top of the stack. Pressing an operator removes the last two elements of the stack, combines them using the operator and puts the result back on top of the stack. For example, the calculation $(2+3)*(4+5)$ can be encoded with the following input sequence:



which is in turn simulated internally as



with the result (45) being the last remaining element in the stack. The file *ReversePolishCalculator.cpp* (folder *Task F*) contains the skeletons of three functions, which are your task to complete. The file *main.cpp* will test your implementation with the above example, and does not need to be modified.

1) Inserting an integer in the RPC

The skeleton **introduceNumber(int input)** should insert the given parameter into the internal stack (it is equivalent to *number* + *Enter*).

2) Performing an operation

The skeleton **operate(std::string operatorString)** should compute internally an operation (which is passed as a string argument, and can be "+", "-" or "*"). Non-supported operators should make an error appear.

3) Printing the result

The skeleton **getResult()** should print the result of the calculation chain (an error should be printed if there is not exactly one element left in the internal stack).