Data Structures BCS-4F
FAST-NU, Lahore, Spring 2021

Homework 1

Due date: March 31, 1155 PM **200 pts**

Give C++ codes for each of the following problems. Codes should be in five separate .cpp files. All code files must be submitted as a single .zip file on Google Classroom.

Problem 1 “Enhanced Queue”

Implement a C++ class with the following definition:

```
template <typename T>
class EnhancedQueue{
private:
   //Only the following members are allowed
   T * eqptr;
   int front;
   int rear;
   int n, cap;
   //you can add utility methods here
public:
   EnhancedQueue();
   void enqueueAtFront(T & obj);
   void dequeueAtFront();
   void enqueueAtRear(T & obj);
   void dequeueAtRear();
   T peekFront();
   T peekRear();

   int size();
   bool empty();
};
```

As you can guess from this definition, the EnhancedQueue allows enqueues and dequeues at both the front and rear of the queue. This means that elements can be added at the front or the rear of the queue as well as removed from the front or the rear using one of the four enqueue/dequeue methods. It’s also evident from the code above that you are only using a single dynamically allocated array for this purpose. Just like the ordinary queue, this queue needs to be circular to save space. When the space fills up, the capacity of the array should be doubled. When more than half of the array becomes empty, the size of the array should be reduced by one-half.

Test your code extensively after coding it. We promise we will do so during evaluations ☺

Problem 2 “Invertible Stack”

Add the following methodology to the Stack Class written in the lecture on Stack:

Add a method called flipStack. This method should work in O(1). Its effect should be such that the whole stack should be logically inverted, i.e. the oldest element becomes the newest and vice versa. So the next pop will remove the element that was at the bottom of the stack before it had been flipped. Notice that the stack may be flipped again and again by using the flipStack method repeatedly. Make sure that no slot of the array is wasted (Hint: Circular). As always, when the stack fills up we double its capacity; and when more than half of it is empty, we halve the capacity.

Once again, test your code extensively.

Problem 3 “The Undo Stack”

Create a class called UndoStack. This stack is intended to be used for the undo operation in another application. It should have the property that it remembers *at most* the last 100 elements pushed into it. So, for example, if there were 101 push operations it would “forget” the oldest of these and if a pop happens the 2nd pushed element will be popped, etc. Write complete C++ code for this class.

Problem 4 “Working with stack and queue limitations”

In the following, remember that the only methods available to you in a stack are push, pop and peek. Similarly, the only methods in a queue are enqueue, dequeue and peek. No extra methods can be added to the Stack or Queue classes for this problem.

(a) Transfer the elements of a stack s to a stack t so that the elements of t are in the same order as in s. Do this while:
    (i) Using one additional stack.
    (ii) Using no additional stack, but only some additional variables.

    You cannot use any other data structures in either (i) or (ii).

(b) Order the elements of a queue in the ascending order from front to rear, but to do so, you can only use:
    (i) Two additional queues
    (ii) One additional queue.

    Obviously, you can also use a few more simple variables in your code, but not another data structure.

Problem 5 “A stack of queues”

In this problem, you will use the built-in STL string, vector, stack and queue. You may add these to your program by using the lines:

```
#include <string>
#include <vector>
#include <stack>
```

```
#include <queue>
using namespace std;
```

(a) Create an object of the STL stack of type character. Use this stack to detect whether a string of 0's and 1's entered by the user is such a string that it is a repetition of a certain number of zeroes following by an equal number of ones. For example, the following strings are all of this type:
01
001101
00110100011101
The following are invalid:
001
0
10
001100011110

(b) The power of templates is that you can create an object with any type of data you wish. For example, you can make a vector of integers, a stack of floats, a queue of characters etc. You can even make a vector of queues of integers, using just one line of STL code:

```
vector< queue<int> > sqn; //a stack of queues of numbers
```

Where might you need to use such a data structure? Imagine, you are making network 'receiver' node that manages multiple recourses of a certain type; these could be printers, transmission links, files, and so on. Then the node should be able to receive requests for each resource; make these requests wait in queues (a separate queue for requests for a specific resource) and serve these request in FIFO order.

Create a class Receiver with the following definition:

```
class Receiver{
  vector<queue<int>> sqn;

public:

 void addRequestforResource(int rid, int reqno);
//adds a new quest with number reqno to the resource sqn[rid]
//rid is between 0 and 4, as we have 5 resources

void serviceRequestatResource(int rid);
//services the request  (dequeues) at front of sqn[rid]

void printQueues()
  //prints all queues line by line, numbers separated by spaces
```

}

Now write a loop that **simulates** this process of request reception and servicing and shows how the queues grow longer and shorter.

First ask the user to provide 5 ‘servicing rates’ for the 5 resources. For example, if the user provides 250 for resource 1, this means that resource 1 services its request after every 250 milliseconds, i.e. after every 250 milliseconds it removes a request from the front of its queue sqn[1]. Based on these servicing rates you should remove a request from the respective queue, shortening it.

Meanwhile, you should add a new request with a random reqid to a random resource id (between 0 and 5) after every 500 milliseconds (this addition rate is fixed).

Use the printQueues method to show the status of all queues after every 250 milliseconds. Do this by clearing the screen each time and printing the queues at the top left of the screen.

Important Notes:

- You can make a loop make one iteration per 1 millisecond by adding the line Sleep(1) at the end of the loop. This will add a delay of 1 millisecond after each iteration. This means that after 250 iterations of the loop, exactly 250 ms would have passed.
- For the first 100 iterations do not wait for 30 seconds and only add requests without servicing any (no dequeues).

---*** END ***