Assignment 2: Stack

PLEASE CHECK THE UPDATES HIGHLIGHTED IN YELLOW (1.1) AND GREEN (1.2)

Task 1: Stack Data structure

You need to implement a Stack ADT as defined in Table 1. There is no restriction on the programming language. However, it might be fruitful for you if you implement your assignments in C++. Please do not use C++17. If you are using C++17 or Java/other languages, please bring your laptop for smooth evaluation.

Table 1: Stack ADT definition.

Fn #	Function	Param.	Ret.	After functions execution	Comment
	[before each function execution.]			<20, 23 , 12, 15>	The values were pushed in the following order: 20, 23, 12, 15. So, TOP points to the value 15. Note that this is a logical definition without any implementation related specification.
1	clear()	-		<>	Reinitialize the stack, i.e., make it (logically) empty stack. <> means an empty stack.
2	push(item)	19		<20, 23 , 12, 15, 19>	Pushes an element.
3	pop()		15	<20, 23, 12>	Pop an element.
4	length()	-	4	<20, 23, 12, 15>	Return the number of elements in the list.

5	topValue()	-	15	<20, 23, 12, 15>	Return the current element.
6	setDirection(direction)	1		<>	This can only be called when we have an empty stack and is relevant only for the array based implementation. For the link list based implementation, it has no meaning. If -1 is given as the parameter then the stack will grow downward from the last position of the allocated array. By default, (or if +1 is given), the stack will grow upward from the first position of the allocated array.
	setDirection(direction)	-1		<>	

The implementation must support the various types of 'elements'. Use templates in C++ or something equivalent in the programming language you intend to use. You need to provide two different implementations, namely, Array Based (Arr) and Linked List (LL) Based Implementations. The size of the stack is only limited by the memory of the computing system, i.e., in case of Arr implementation, there must be a way to dynamically grow the stack size as follows: the stack should double its current size by **allocating memory dynamically**, i.e., initially the list should be able to hold X elements; as soon as the attempt is made to insert the $X+1^{th}$ element, memory should be allocated such that it can hold 2X elements and so on. Static implementation would result in deduction of marks.

Appropriate constructors and destructors must be implemented. For the Arr implementation, the constructor takes the initial size of the array with a default size available as a constant. A second constructor for the Arr implementation takes a pointer to the already allocated array and direction (1 or -1) and starts using that as an empty stack.

You may implement extra helper functions but those will not be available for programmers to use. So, while using the stack implementations, one can only use the methods listed in Table 1. Please note that during evaluation, identical main function will be used to check the functionality of both Arr and LL implementations. Also write a simple main function to demonstrate that all the functions are working properly, for both implementations. Recall that, the same main function should work for both the implementation except for the object instantiation part. Please follow the following input/output format.

Input format (for checking the Arr and LL implementations):

Follow the same format provided for Assignment 1 (List). The memory chunk size parameter is not taken anymore in the first line.

Example input:

To be provided shortly.

Output Format:

Follow the same format provided for Assignment 1 (List).

Expected Output of the example input:

To be provided shortly.

Task 2: Using the Stack (Dishwasher)

You have invited all your friends for dinner. Your mother agreed to cook a delicious meal for them, but told you that you will be in charge of dish washing. The plan is that as soon as someone finishes eating (at a time point) he will push the dish into the 'dirty' stack. You will have to pop the dish, wash it and then push it in the 'clean' stack. Each dish is associated with a size information, which will dictate the time you need to wash that. Now, you need to simulate the whole cleaning task. The menu contains a x-course meal and hence potentially, for each invitee you may need to clean upto x dishes (of varying sizes). Each course is to be eaten at most once, in order (i.e., course 1 is eaten first followed by course 2 and so on; if someone cannot eat course 3 unless he eats course 2) and on a particular size of dish which are maintained strictly. So, each invitee eats a particular course in the designated dish and then pushes the dish in the dirty stack. As soon as the first dish is pushed, your job starts and you start washing

by popping that dish and continue washing. Notably, a meal is said to be full (i.e., a full meal) if that contains all courses.

To do the above simulation, you can only use stack data structure as implemented according to the stack ADT in table 1. Other than the stack data structure, you are only allowed to use normal arrays for bookkeeping purposes. You need to provide 3 separate implementations as follows:

Impl. A: Using LL implementation of stacks

Impl. B: Using Arr implementation of stacks

Impl. C: 1 Array 2 Stack- the dirty and clean stacks must be implemented in one array.

Input format:

The first line will contain 2 parameters, n and x, where n gives the number of invitees (i.e., number of persons who could potentially eat) and x gives the number of courses in the meal. Courses should be assumed to be numbered from 1 to x.

In the second line, you will have x numbers, a_i, i \in {1..x}, space separated, each indicating the time unit taken to wash the corresponding dish for a course (i.e., a_i is the time required to wash the dish designated for course i).

In subsequent lines you will have three parameters, k (\in {1..n}), t >= 0 and s (\in {1..x}), space separated, which represents information for a dish that has been pushed at the Dirty stack. The parameter k gives you which friend is pushing the dish, t gives you a time information, i.e., when it has been pushed. It is guaranteed that the first value of t will be 0 and t will be strictly increasing for the subsequent pushes (i.e., dishes). The other parameter, s indicates which course this dish is used for.

The end of input (i.e., eating) is indicated when k is 0.

Output Format:

The first line reports an integer A, the end time when all dishes have been pushed in the clean stack. Assume that no time is needed to push the dish into the (clean) stack after it is washed. So, if the washing ends at time t, then it is also pushed at time t. Similarly, assume that as soon as the dish is put into the stack you may start popping it and start washing it (i.e., pushing and popping do not need any time).

The next line will provide the washing end time of each of the dishes in chronological order, comma separated.

The third line reports whether all of your friends have eaten all x courses of meals. Here you write either 'Y' (for yes) or 'N' (for no).

The next line will provide the ID ([1..n]) of each of your friends (comma separated) who have taken a full meal (i.e., all x courses), in reverse chronological order according to the time they completed their (full) meal. If none has completed a full meal, this should be an empty line.

Example Input 1:

3 3
2 4 1
1 0 1
3 1 1
2 5 1
1 7 2
3 8 2
3 10 3
0 0 0

Example Output 1:

15
1,3,6,10,11,15
N
3

Example Input 2:

3 3
2 4 1
1 0 1
3 1 1

2 5 1

1 7 2

3 8 2

3 10 3

1 17 3

0 0 0

Example Output 2:

17

1,3,6,10,11,15,17

N

1,3

Submission Guidelines:

1. **Create a directory with your 7-digit student id as its name**
2. **You need to create separate files for the Arr implementation code (e.g. Arr.cpp/Arr.py), LL implementation code (e.g. LL.cpp/LL.py) putting common codes in another file. Create a separate file for the main function implementing the Dishwasher simulation (e.g., dw.cpp/dw.py). Your Dishwasher simulation implementation must import/include your array and LL implementations as header files, and you must use your own array/LL implementations for the Dishwasher simulation task. No other built in data structure can be used.**
3. **Put all the source files only into the directory created in step 1. Also create a readme.txt file briefly explaining the main purpose of the source files.**
4. **Zip the directory (compress in .zip format. Any other format like .rar, .7z etc. is not acceptable)**
5. **Upload the .zip file on Moodle in the designated assignment submission link. For example, if your student id is 1905xxx, create a directory named 1905xxx. Put only your source files (.c, .cpp, .java, .h, etc.) into 1905xxx. Compress the directory 1905xxx into 1905xxx.zip and upload the 1905xxx.zip on Moodle.**

Failure to follow the above-mentioned submission guideline may result in upto 10% penalty.

Submission Deadline: 03 December 11 PM.

Evaluation Policy:

Task	Implementation	
1	Arr based	25%
	LL based	25%
2	Use LL Based Stack	20%
	Use Array based Stack	20%
	1 array 2 stack	10%