Stack & Queue

You do not need to write the Node class, push()/pop()/peek()/enqueue()/dequeue()/peek()/isEmpty() etc. functions unless specified in the problem.

Just design the function specified in the problem.

No need to write driver code.

This document is a compilation of previous semester's exam (midterm/final) problems and lab quiz problems. Try to practice relevant problems from online judges as well.

1.

Tasks from Lab 4 - Lab 4 - Secondary Data Structures.docx

Tasks from Book (Stack) - Practice Sheet 5 - Stack.pdf

Tasks from Book (Queue) - Practice Sheet 6 - Queue.pdf

2.

Consider that a **MidStack** class has been created containing the push(element), pop(), peek() and isEmpty() functions. **No need to implement MidStack class.** The MidStack class implements a singly linked list-based Stack hence overflow is not possible. The pop() and peek() functions return None in case of the underflow.

Complete the function **conditional_reverse** which will take an object of MidStack that contains some integer values. The function returns a new stack which will contain the values in reverse order from given stack with the exception that if consecutive numbers are the same, it picks only one from them. **You cannot use any other data structure except MidStack.**
Remember that a stack has no other functions than push, pop, peek, and isEmpty

Python Notation	Java Notation

def conditional_reverse(stack):
# To Do 	
	public MidStack conditional_reverse(MidStack stack) {
	// To Do
	}

Sample Input Stack (Right most is the top)	Returned Stack (Right most is the top)	Explanation
Stack: 10, 10, 20, 20, 30, 10, 50 Top = 50	Stack: 50, 10, 30, 20, 10 Top = 10	Consecutive 20 and 10 are not present in the output reversed stack

3.

You are given a **Stack** of people. They should be paired off as man with woman, woman with man. If a male and a female are consecutive in the stack, they get paired immediately. If there are consecutive females found without any subsequent males, then the spare females should be saved in a **Queue** for future pairing. If there are consecutive males found without any subsequent females, we try to pair a male with the first spare female [if available] from the **Queue**. If no spare female is available, then the male is discarded. The same logic applies to consecutive females without any subsequent males.

The stack class and the Queue class are implemented for you. **The public instance methods in the Stack class are push(), pop(), peek(), and isEmpty(). The public instance methods in the Queue class are enq(), deq(), peek(), and isEmpty().** In both classes, isEmpty() returns a boolean value, peek() and pop() returns None for underflow. All other members of both classes are private. You can only use the public instance methods. No need to consider the overflow exception for both Stack and Queue classes.

Your task is to complete the given method **dance_pair()** that takes a stack and prints every male-female pair.

Every male and female are written in this way - M_id or F_id. A method called **id_gender_extractor**(s) is implemented for you that returns the id and gender from a string,s.

You do not need to write the methods/classes that are implemented for you. You cannot use any data structure other than stack and queue.

Sample Input	Output	Explanation
st = Top → M_10 F_20 F_4 F_5 M_3 M_19 M_1 M_7 F_9 F_18	10 and 20 are paired together 5 and 3 are paired together 4 and 19 are paired together 7 and 9 are paired together	M_10 and F_20, F_5 and M_3, M_7 and F_9 are sequential in the stack. They are paired up. From F_4, there are consecutive females. So, F_4 is stored in the queue for future pairing. From M_19, there are consecutive males. M_19 is paired with spare F_4. M_1 is discarded since there is no spare female with whom M_1 can be paired.

Given:

```
def id_gender_extractor(s):
    if s != None:
        return s[0],s[2:] #gender, id
    else:
        return None, None
```

Driver Code:

```
st = Stack()
st.push('F_18')
st.push('F_9')
st.push('M_7')
st.push('M_1')
st.push('M_19')
st.push('M_3')
st.push('F_5')
st.push('F_4')
st.push('F_20')
st.push('M_10')

dance_pair(st)
```

Python Notation	Java Notation

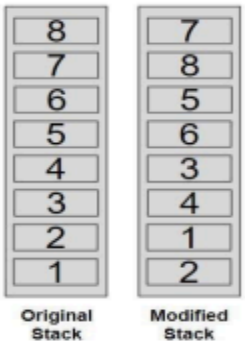
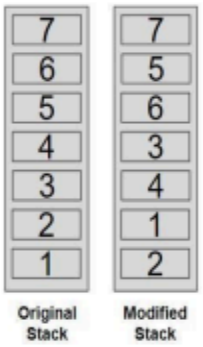
def **dance_pair**(st):
# To Do 	
	public void **dance_pair**(Stack st) {
	# To Do
	}

4.

You are given a stack of some integer elements, st. Your task is to swap the adjacent elements of the stack and return the modified stack. If the number of elements in the stack is odd, keep the topmost element as it is and swap the other adjacent elements. See the input output for better understanding. Now write a function **Do_Adjacent_Swap(st)** that completes the given task and returns the modified stack.

The st is an object of the "Stack" class which is implemented by a singly linear linked list and consider that the following methods of "Stack" class are already implemented for you: **push(element), pop(), peek(), isEmpty().**

You cannot use any other data structures other than stack. The "Top" pointer of Stack class is NOT accessible to you. You can use just the above four methods of Stack class. You can create as many stacks as you need for this task.

Sample Input	Sample Output	Explanation
Original Stack: 8(Top) → 7 → 6 → 5 → 4 → 3 → 2 → 1	Modified Stack: 7(Top) → 8 → 5 → 6 → 3 → 4 → 1 → 2	
Original Stack: 7(Top) → 6 → 5 → 4 → 3 → 2 → 1	Modified Stack: 7(Top) → 5 → 6 → 3 → 4 → 1 → 2	

Python Notation	Java Notation

def Do_Adjacent_Swap(st):
  # To Do
	public Stack Do_Adjacent_Swap(Stack st): {
	  // To Do
	}

5.

You are given the start and end time of some tasks. The tasks are sorted according to their starting time. You can merge two tasks **A** and **B if they are overlapping.** That is **task B** starts before the finishing time of **task A**. In that case, it becomes a single task having the ending time of **task A** or **task B** (depending upon which task finishes later). This task can be merged further with other tasks if the above condition satisfies (overlapping). You want to merge as many tasks as possible and minimize the total number of non-overlapping tasks.

You are given a 2D array named *tasks* storing the start and end time of the task. The array is sorted according to the starting time of the tasks. Each row of the array stores two integer values. The first value indicates the start time and the second value indicates the end time of a task. More specifically, the start and end time of the i^{th} task is `tasks[i][0]` and `tasks[i][1]` respectively.

Write a function named ***`print_total_task(tasks)`*** that takes the tasks array as input and prints the start time and end time for each non-overlapping task according to the given format. Note that the tasks are printed as decreasing order of start time.

Consider that the Stack class is already available with the following methods/functions implemented: push(element), pop(), peek(), and isEmpty(). Stack Underflow and Stack Overflow exceptions return None (Python) / null(Java). There is no need to write the Stack class.

You are allowed to create as many *stacks* as you want. You cannot create any new array. To create a Stack object use the following:
stack = Stack() #(Python) **Stack stack = new Stack() //(Java)**

Example 1:	**Explanation:**
Input: *tasks* = [[1, 5], [2, 3], [4, 6], [7, 10], [9, 11], [12, 15]] **Output:** 12, 15 7, 11 1, 6	Tasks (1,5) and (2,3) are overlapping, so we can merge them creating the new task (1,5) Then task (1,5) and (4,6) are also overlapping, merging them creates the new task (1,6) We can not merge tasks (1,6) and (7,10) as they are non-overlapping However tasks (7,10) and (9,11) can be merged creating new task (7, 11) Finally tasks (7,11) and (12,15) are non-overlapping So we have 3 non-overlapping tasks (1,6), (7,11) and (12,15) Tasks are printed in descending order of start time.

Python Notation	Java Notation

`def print_total_task(tasks):`
`  # To Do`
	`public void print_total_task(int [ ][ ] tasks){`
	`  // To Do`
	`}`

6.

You are given a stack of integers. Write a function named **filter_and_sort_stack()** that takes a stack st and an integer k as input parameters. The function modifies and returns the input stack such that:

1. All elements greater than k are removed from the stack.
2. The remaining elements are sorted in ascending order in the stack.

Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:

- You can only use instances of the provided Stack class and its methods.
- No other data structures can be used apart from additional Stack instances.

Sample Input	**Sample Output**	**Explanation**
Stack: 50 40 70 20 10 80 30 k = 40	Stack: 10 20 30 40	Remove elements greater than 40 (i.e., 50, 70, and 80). The remaining elements 40, 20, 10, 30 are sorted into 10, 20, 30, 40.

7.

You are given a stack of integers. Write a function named ***conditional_reverse*** that takes a stack *st* and an integer *n* as input parameters. The function modifies and returns the input stack where the input stack is reversed from the top to *n-th* element.

Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:
You are only allowed to use instances of the provided Stack class, which supports the provided methods. No other data structures can be used other than Stack.

[Hint: You can create multiple instances of the Stack class]

Sample Input	**Sample Output**	**Explanation**
Stack: 10 20 30 40 50 60 n = 4	Stack: 10 20 60 50 40 30	The top element of the input stack is 60. Reversing 4 elements from the top results in Stack: 10 20 60 50 40 30

8.

You are given a stack of integers. Write a function named ***sum_stack*** that takes a stack st as input and modifies it such that the resulting stack contains the sum of corresponding elements from the **top** and **bottom** of the original stack. Specifically, the last element (top) of the stack is summed with the 0th element (bottom), the second last element is summed with the 1st element, and so on. It is ensured that there will be an even number of elements in the original stack.
Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:

- You are only allowed to use instances of the provided Stack class, which supports the provided methods
- No other data structures can be used other than Stack.

[Hint: You can create multiple instances of the Stack class to assist in solving the problem.]

Sample Input:	**Sample Output:**	**Explanation:**
Input Stack: **10 15 20 50 25 35**	**Output Stack:** **70 40 45**	**Pair 1: 35 (Top) + 10 (Bottom) = 45** **Pair 2: 25 (Second from top) + 15 (Second from bottom) = 40** **Pair 3: 50 (Third from top) + 20 (Third from bottom) = 70**

9.

A stack of integers is given to you. Now you have to rearrange the given stack in such a way so that on top there will be an ODD integer and the EVEN integers will come after an ODD integer. You have to maintain the relative sequence of the even numbers and the odd numbers.

There can be **TWO** scenarios in the given Stack:

1. There will be the same number of odd and even integers in the given stack if the stack length is EVEN [odd.count = even.count] **OR**
2. The total number of the odd integers will be one extra from the total number of even integers if the stack length is ODD [odd.count+1 = even.count]

You need to solve the above problem using **Stack class**. You cannot use other methods than pop(), peek(), push(), isEmpty() methods of Stack. Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:

- No other data structures can be used other than Stack.

[Hint: You can create multiple instances of the Stack class to assist in solving the problem.]

Sample Input	Sample Output	Explanation
Stack: | 11 | | 22 | | 24 | | 35 | | 41 |	Stack: | 11 | ← top | 22 | | 35 | | 24 | | 41 |	There are 3 odd integers(11,35,41) and 2 even integers(22,24). On the top 11 is kept as it is an even integer. Then comes the odd integer then again comes the odd integer. The arrival sequence is maintained for even and odd integers.
Stack: | 8 | |10 | | 7 | | 5 | |12 | | 3 |	Stack: | 7 | ← top | 8 | | 5 | |10 | | 3 | |12 |	

10.

You are given a stack of integers and an integer **k**. Write a function named ***rotate_stack*** that takes a stack and an integer as input. The function will modify the stack to rotate it ***upward*** by k position(s). You have to return the modified stack.
Assume the Stack class is already given and provides standard methods: **push**, **pop**, **peek**, and **isEmpty**.
Constraints:

- You are only allowed to use instances of the provided Stack class, which supports the provided methods
- No other data structures can be used other than Stack.
- Note that k can be very large. $0 \leq k \leq 1,000,000,000$

Sample Input 1:	**Sample Output 1:**
Input Stack: **10 (Top)** **20** **30** **40** **50** **K = 2**	**Output Stack:** **30 (Top)** **40** **50** **10** **20**

Sample Input 2:	**Sample Output 2:**
Input Stack: **10 (Top)** **20** **30** **40** **50** **K = 6**	**Output Stack:** **20 (Top)** **30** **40** **50** **10**

11.

You are given a stack of integers. Write a function named **retain_and_reverse_stack()** that takes a stack st and an integer m as input parameters. The function modifies and returns the input stack such that:

1. Only the top m elements are retained in the stack (discard the rest).
2. The retained elements are reversed in the stack.

Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:

- You are only allowed to use instances of the provided Stack class and its methods.
- No other data structures can be used apart from additional Stack instances.

Sample Input	Sample Output	Explanation
Stack: 10 20 30 40 50 60 n = 4	Stack: 40 30 20 10	Retain the top 4 elements (10, 20, 30, 40) and discard the rest (50, 60). Reverse the retained elements to get 40, 30, 20, 10.

12.

You are given a stack of integers and an integer **k**. Write a function named ***rotate_stack*** that takes a stack and an integer as input. The function will modify the stack to rotate it ***downward*** by k position(s). You have to return the modified stack.
Assume the Stack class is already given and provides standard methods: **push**, **pop**, **peek**, and **isEmpty**.
Constraints:

- You are only allowed to use instances of the provided Stack class, which supports the provided methods
- No other data structures can be used other than Stack.
- Note that k can be very large. $0 \leq k \leq 1,000,000,000$

Sample Input 1:	**Sample Output 1:**
Input Stack: **10 (Top)** **20** **30** **40** **50** **K = 2**	**Output Stack:** **40 (Top)** **50** **10** **20** **30**

Sample Input 2:	**Sample Output 2:**
Input Stack: **10 (Top)** **20** **30** **40** **50** **K = 6**	**Output Stack:** **50 (Top)** **10** **20** **30** **40**

13.

You are given a stack of integers. Write a function named ***sub_stack*** that takes a stack st as input and modifies it such that the resulting stack contains the difference of corresponding elements from the **top** and **bottom** of the original stack. Specifically, the 0th element (bottom) of the stack is subtracted from the last element (top), the 1st element is subtracted from the second last element, and so on. It is ensured that there will be an even number of elements in the original stack.
Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:

- You are only allowed to use instances of the provided Stack class, which supports the provided methods
- No other data structures can be used other than Stack.

[Hint: You can create multiple instances of the Stack class to assist in solving the problem.]

Sample Input:	**Sample Output:**	**Explanation:**
Input Stack: **10 20 30 40 50 60**	**Output Stack:** **10 30 50**	**Pair 1: 60 (Top) - 10 (Bottom) = 50** **Pair 2: 50 (Second from top) - 20 (Second from bottom) = 30** **Pair 3: 40 (Third from top) - 30 (Third from bottom) = 10**

14.

A stack of integers is given to you. Now perform summation of the odd integers and even integers to rearrange the given stack in such a way so that if the summation of the even integers is more than the summation of the odd integers then keep the odd integers at the bottom and the even integers at the top of the given stack. Otherwise, keep the odd integers at the top and even integers at the bottom. You have to maintain the relative sequence of the even numbers and the odd numbers.

You need to solve the above problem using **Stack class**. You cannot use other methods than pop(), peek(), push(), isEmpty() methods of Stack. Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:

- No other data structures can be used other than Stack.

[Hint: You can create multiple instances of the Stack class to assist in solving the problem.]

Sample Input	Sample Output	Explanation
Stack: | 11 | | 22 | | 24 | | 35 | | 41 |	Stack: | 11 | ← top | 35 | | 41 | | 22 | | 24 |	The summation of odd integers 11+35+41=87 is greater than the summation of even integers 22+24=46. So,all the even integers will be placed after the odd integers.
Stack: | 8 | |10 | | 7 | | 5 | |12 | | 3 |	Stack: | 8 | ← top |10 | |12 | | 7 | | 5 | | 3 |	The summation of even integers 8+10+12=30 is greater than the summation of odd integers 7+5+3=15. So,all the odd integers will be placed after the even integers.

15.

You are given a stack of integers. Write a function named ***conditional_reverse*** that takes a stack *st* and an integer *n* as input parameters. The function modifies and returns the input stack where the input stack is reversed from the bottom to *n-th* element.

Assume the Stack class is already given and provides standard methods: push, pop, peek, and isEmpty.

Constraints:
You are only allowed to use instances of the provided Stack class, which supports the provided methods. No other data structures can be used other than Stack.

[Hint: You can create multiple instances of the Stack class]

Sample Input	**Sample Output**	**Explanation**
Stack: 10 20 30 40 50 60 n = 4	Stack: 40 30 20 10 50 60	The bottom element of the input stack is 10. Reversing 4 elements from the bottom results in Stack: 40 30 20 10 50 60