University of Calgary
Faculty of Science
*Department of Computer Science*

**CPSC 331 - Data Structures, Algorithms and their Analysis**
Assignment 2
Application of Stacks and Queues
*Winter 2024*

# 1 Objectives

In this assignment you will gain experience in manipulating two essential ADT: stacks and queues either. As part of the learning experience, you will also implement the basic operations on these ADT as well as several algorithms to use these ADT in various applications. You will need to provide some analysis of the time complexity of these algorithms.

# 2 Background

Stacks and queues are fundamental data structures in computer science that play important roles in various algorithms and applications which provide efficient ways to store and manage data based on specific principles.

**Stack: LIFO (Last-In-First-Out)**

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It can be visualized as a stack of objects, where the last object placed on top is the first one to be removed. Think of a real-life stack of plates, where you can only remove the topmost plate.

**Queue: FIFO (First-In-First-Out)**

A queue is another linear data structure that follows the First-In-First-Out (FIFO) principle. It can be visualized as a line of people waiting for a service, where the person who arrives first is served first. In a queue, elements are added to the rear and removed from the front.

# 3 Questions

You need to solve the following problems.

## 3.1 Exercise 1 - Bonus

1. Given a directed graph and two vertices - source and destination, determine if the destination vertex is reachable from the source vertex. The solution should return true if a path exists from the source vertex to the destination vertex, false otherwise. Note: Edge (x, y) represents an edge from x to y.

   Follow the specific input-output format: Input: Graph [edges = [(0, 6), (6, 7), (7, 3), (3, 5), (4, 6)], n = 6], src = 4, dest = 5 Output: true Explanation: There exists a path [4 — 6 — 7 — 3 — 5] from vertex 4 to vertex 5.

   Input: Graph [edges = [(0, 6), (6, 7), (7, 3), (3, 5), (4, 6)], n = 6], src = 5, dest = 0 Output: false Explanation: There is no path from vertex 5 to any other vertex.

2. Given a directed graph, two vertices - source and destination, and a positive number m, find the total number of routes to reach the destination vertex from the source vertex with exactly m edges. Note: Edge (x, y) represents an edge from x to y.

   Follow the specific input-output format:

   Input: Graph [edges = [(0, 6), (0, 1), (1, 6), (1, 9), (1, 5), (5, 3), (3, 4), (9, 5), (9, 3), (9, 4), (6, 9), (7, 6), (7, 1)], n=8], src = 0, dest = 3, m = 4

   Output: 3

   Explanation: The graph has 3 routes from source 0 to destination 3 with 4 edges.

   0 — 1 — 9 — 5 — 3

   0 — 1 — 6 — 9 — 3

   0 — 6 — 9 — 5 — 3

## 3.2 Exercise 2

Refer to the figure below to write a Java program to do following[1]:

1. Generate a similar queue as the initial queue of the figure above. Your task is to generate an updated queue reversing the initial queue where you should append the last value of the initial queue to the first index of the updated queue, the 2nd last last value of the initial queue to the second index of the updated queue and so on. For example, in the figure, the last value of the initial queue is 25 and the second last value is 5. In the updated queue, 25 comes in the first index of the queue and 5 in the second index.

2. Consider the updated queue. Your task is to divide the value of each index with its following index. If the value of the previous index < then the value of following index, divide the value of the following index with the previous index. Generate a new queue with the resultant values. Consider the whole number if the result is decimal, i.e.

[1]Hint: An array-based implementation is recommended here.

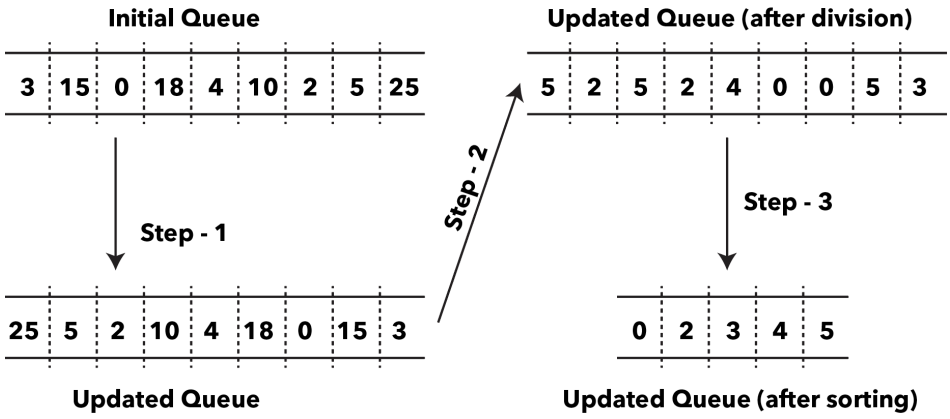


Figure 1: Queue

integer division. For example, in the previously updated queue, the value of the 1st index = 25, the value of the 2nd index = 5, result = 25/5 = 5. Therefore, in the new updated queue (after division) the value of the first index will be 5. Repeat the same for the value of the 2nd index = 5, value of the 3rd index = 2, result = 5/2 = 2. Therefore, in the new updated queue the value of the 2nd index is 2. Similarly, the value of the 3rd index = 2, value of the 4th index = 10, as $2 < 10$, result = 10/2 = 5. And so on.

3. Sort the updated queue (after division) in descending order. If any value appears more than once (duplicate), just append once in the queue.

   You must follow the specific output format:

   Step-1 :

   Initial queue - [3,15,0,18,4,10,2,5,25]

   Updated queue - [25,5,2,10,4,18,0,15,3]

   Step-2:

   Updated queue - [25,5,2,10,4,18,0,15,3]

   Updated queue (after division) - [5,2,5,2,4,0,0,5,3]

   Step-3:

   Updated queue (after division) - [5,2,5,2,4,0,0,5,3]

   Updated queue (after sorting) - [0,2,3,4,5]

## 3.3 Exercise 3

Write a Java program to reverse a stack using the queue data structure. From the reversed stack, find the unique elements and generate a new queue with the elements following FIFO

order.

Hints to reverse a stack using queue data structure:

1. Pop elements one by one from the stack.
2. Enqueue every popped element into the queue.
3. Do it till the stack is empty.
4. Then, Dequeue elements one by one from the queue.
5. Push every dequeued element into the stack.
6. Do it till the queue is empty.
7. The stack is now reversed.

You may follow the pseudo-code for reversing the stack:

ReverseStack(stack, queue):
// Step 1: Pop elements from the stack and enqueue into the queue
while stack is not empty:
element = Pop(stack)
Enqueue(queue, element)

// Step 2: Dequeue elements from the queue and push into the stack
while queue is not empty:
element = Dequeue(queue)
Push(stack, element)
// The stack is now reversed

Use the following algorithm to solve the problem. Provide a brief analysis of the complexity for it and justify its Big-Oh notation.

**GenerateUniqueQueue(stack)**
Input: A stack of elements
Output: A queue containing the unique elements in FIFO order

1. Initialize an empty set called uniqueSet to track unique elements.
2. Initialize an empty queue called uniqueQueue.
3. Iterate over the elements in the stack, one by one.
4. For each element in the stack, do the following

   Check if the element is already present in uniqueSet.

   If the element is not present in uniqueSet, add it to uniqueSet.

   Enqueue the element into uniqueQueue.

5. Return uniqueQueue, which will contain the unique elements in FIFO order.

You must follow the specific output format:

Input Stack: [2 - 9 - 3 - 1 - 8 - 9 - 0 - 7 - 8 - 4 - 5 - 3]
Output Reversed Stack: [3 - 5 - 4 - 8 - 7 - 0 - 9 - 8 - 1 - 3 - 9 - 2]
Output Queue with unique values: [3 - 5 - 4 - 8 - 7 - 0 - 9 - 1 - 2]

## 3.4 Exercise 4

Suppose you are shopping to buy shoes, and you have a shopping cart. You can add or remove items from your cart using the concept of a stack, based on the following preferences:

You have a dictionary containing tuples for each shoe item. Each tuple consists of three elements:

- x: The brand name of the shoe.

- y: The original price of the shoe.

- z: The discount on the original price of the shoe.

Here is the dictionary of shoe items:

('Nike', 450, 25), ('Adidas', 400, 10), ('Puma', 600, 30), ('Sorel', 360, 0), ('Aldo', 680, 15), ('Skechers', 380, 0)

In this dictionary, you can find information about various shoe brands, their original prices, and the discounts available. Each entry in the dictionary represents one shoe item. Using a stack, you can add or remove items to your shopping cart.

The following algorithm can be used to solve the problem. Provide a brief analysis of the complexity for it and justify its Big-Oh notation.

1. Start by adding the first item from the dictionary to your shopping cart.

2. Calculate the price of the first item after any applicable discount. Let's call this calculated price "prev".

3. For each remaining item in the dictionary, repeat steps 4-7.

4. Move on to the next item in the dictionary.

5. Calculate the price of the new item after any discount and call this calculated price "next".

6. Compare the calculated prices "prev" and "next" to determine which item has a lower price.

7. If "next" is lower than "prev", update the existing item in your shopping cart with the new item that has the lower price. Otherwise, keep the existing item in your shopping cart as it already has the lower price.

8. If "prev" and "next" are the same even after applying the discount, choose the item that has a discount and keep it in your shopping cart.

9. Repeat steps 4-8 until all items in the dictionary have been processed.

You must follow the specific output format:

Shoes info: ('Nike', 450, 25), ('Adidas', 400, 10), ('Puma', 600, 30), ('Sorel', 360, 0), ('Aldo', 680, 15), ('Skechers', 390, 0)

1st step: [('Nike', 450, 25)]
2nd step: [('Adidas', 400, 10)] as 400 X 10 percent = 360 is less than 450
3rd step: [('Adidas', 400, 10)] as 600 X 30 percent = 420 is greater than 360
4th step: [('Adidas', 400, 10)] as 360 X 0 percent = 360 = 360 and 'Adidas has a discount'
5th step: [('Adidas', 400, 10)] as 680 X 15 percent = 578 is greater than 360
6th step: [('Adidas', 400, 10)] as 300 X 0 percent = 390 is greater than 360

## 3.5 Exercise 5

You are tasked with simulating asteroid collisions using stacks.Implement a Java program that takes a sequence of asteroids represented as integers (positive for asteroids moving to the right, negative for asteroids moving to the left) and determines the final state after all possible collisions have occurred. Use a stack to simulate the collisions.

You have to implement the following class:

```
Class: AsteroidCollision
Methods:
   public static int[] simulateCollisions(int[] asteroids):
```

This method takes an array of integer asteroids representing the sequence of asteroids. It simulates the collisions and returns the final state of the asteroids after all possible collisions. The returned array should only contain the asteroids that survived the collisions. The array should preserve the order of the asteroids. Note the following:

1. Asteroids moving in the same direction will never collide e.g. (5, 15) will never collide.

2. When two asteroids collide, the smaller one will explode, and the larger one will continue moving in the same direction e.g. (16, -7); here, -7 will explode, and only 16 is saved.

3. If two asteroids have the same size and opposite direction (2, -2), they both explode, and none will be saved.

When dealing with asteroids, the negative sign in front of a value does not indicate that the value itself is negative. Instead, it signifies the direction in which the asteroid is moving. For

example, consider the pair of asteroids (-21, 2). This simply indicates that the first asteroid has a magnitude value of 21 but moving in the opposite direction of the asteroid with a value of 2.

**Algorithm: AsteroidCollisions(stack)**

Input: A stack of asteroid values
Output: The resulting stack after asteroid collisions

1. Create an empty stack called resultStack to store the resulting asteroids.

2. Iterate over the asteroids in the input stack, one by one.

   Let the current asteroid be represented by the variable 'curr'.

   If resultStack == empty or the current asteroid 'curr' is moving in the same direction as the top asteroid in resultStack: push 'curr' onto resultStack. If the top asteroid in resultStack is moving in the opposite direction of 'curr', compare their sizes to determine the outcome:

   If 'curr' has a larger size, continue pushing 'curr' onto resultStack.

   If 'curr' has a smaller size, discard 'curr' and continue to the next asteroid in the input stack. If 'curr' and the top asteroid in resultStack have the same size, pop the top asteroid from resultStack, discarding both asteroids.

3. Return the resulting stack after all asteroid collisions, represented by resultStack.

   You must follow the specific output format:

   Input asteroids = [7, 16, -16, -7, 5, -5, -21, 2, 2, 34, -9]

   Step-1 : []

   Step-2: [7]

   Step-3: [7,16]

   Step-4: [7]

   Step-5: []

   Step-6: [5]

   Step-7: []

   Step-8: [-21]

   Step-9: [-21]

   Step-10: [-21]

   Step-11: [34]

   Step-12: Final asteroid = [34]

# 4 Grading

The entire assignment carries 100 points: 25 points for each of the questions 2, 3, 4, and 5. Exercise 1 will carry an additional 10 points bonus. For exercise 2, 10 points will be given to the first part, 10 points to the second part, and 5 points to the last part.

# 5 Deadline

The assignment is due by March 4 at 11:59 PM, after that the deduction policy of 20 percent daily (part of the day is considered as a full day) will be applied. The first 15 minutes of the penalty day is a grace period and will not result in deduction.

# 6 Submission

Please note that for most of the questions, you will need to implement code in java, and use your own implementations of the ADT queues and stacks. You also need to type out your solutions for the time complexity analysis questions. Make sure to be consistent with the formatting. Please ensure that you have numbered the questions, use the same input and print exactly the same output as mentioned (when applicable). Correctly and double check your work before submitting it. While typing out a certain solution, make sure all the solutions are typed out in a similar manner. Please upload the PDF version of the typed documents.

1. Exercise-1 (IF ANY), Exercise-2 and Exercise-5 should be in the form of Java Files (3 Files).

2. Exercise-3, Exercise-4, there will be both java files (for implementation) and pdf (for time complexity) (4 files)

   To submit, create a new folder called "dir name" and place all files you wish to submit in this folder. Use the command "tar -zcvf zip name.tar.gz dir name" and upload the zip name.tar.gz. The Zip File name should be the student name+ID (example: Name = John Doe, Student ID = 12345678, FILENAME = john doe 12345678). Ensure that you have added all the files while zipping.