Computer Science CSC263H
St. George Campus

July 8, 2019
University of Toronto

Homework Assignment #4

Due: July 22, 2019, by 11:59pm

- You must submit your assignment as a PDF file of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  `markus.teach.cs.toronto.edu/csc263-2019-05`

  To work with one or two partners, you and your partner(s) must form a group on MarkUs.

- If this assignment is submitted by a group of two or three students, the PDF file that you submit should contain for each assignment question:

  1. The name of the student who *wrote* the solution to this question, and
  2. The name(s) of the student(s) who *read* this solution to verify its clarity and correctness.

- The PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTeX typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can use other typesetting systems if you prefer, but handwritten documents are not accepted.

- By virtue of submitting this assignment you (and your partners, if you have any) acknowledge that you are aware of the homework collaboration policy that is stated in the csc263 course web page: `https://github.com/csc263-summer2019/csc263-s19/wiki/Course-Information-Sheet#assignment-collaboration`.

- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.

- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.

Question 1. (30 marks)

Consider a deque (its like a queue but you can insert/remove at either end) like data structure with the following operations:

- **DequeWithMax** - creates an empty object, constructor, called only once before any other functions are called.
- **pushFront(x)** - pushes x to the front of the data structure
- **popFront()** - removes and returns the front item from the data structure, nil if empty
- **pushBack(x)** -pushes x to the back of the data structure
- **popBack()** - removes and returns the back item from the data structure, nil if empty
- **maximum()** returns the biggest item in the data structure.

a. (10 marks) Describe how you would implement a data structure with the above functionality with a amortized constant run time per operation

Ensure that your description includes

- what you will store (aside from the values, do you need to store anything else to make the above run quickly?)
- how you will store it (underlying data structure used etc.)
- how each function would work

b. (10 marks) Using the aggregate method, provide a proof that the amortized cost per operation is constant.

c. (10 marks) Using the accounting method, provide a proof that the amortized cost per operation is constant.

Question 2. (20 marks)

a. (15 marks) In this question you will create an automatic maze generator.

The maze generator is given the following information:

- A maze size consisting of two values *row* and *col*
- A list of wall objects.

The maze size defines a "grid" of cells. There are *col* cells going across, *row* cells going up and down. Each cell is numbered starting with 1 at the top left, going across then onto the next row. Thus a maze of 3 rows and 4 columns would have cells numbered as follows:

```
  1 |  2 |  3 |  4
------------------
  5 |  6 |  7 |  8
------------------
  9 | 10 | 11 | 12
```

Each wall represents a separation between two "cells" by storing the cells' ID. Thus a 2, 3 represents the wall between cell 2 and cell 3. Initially all cells are separated. (ie the list of wall objects define all walls needed to separate every cell in the grid.). Thus there are ((row-1)*col + (col-1)*row) walls in the list initially.

The algorithm you will use to generate a maze is as follows:

- randomly remove a wall
- if every cell is now joined we are done
- if not repeat the process

Now clearly we want a maze, not an empty hall (formed by removing every single wall). Thus, we can only remove a wall if the two cells are not already joined.

For example, suppose we start off with following maze:

```
  1 |  2 |  3 |  4
------------------
  5 |  6 |  7 |  8
------------------
  9 | 10 | 11 | 12
```

Suppose we remove the following walls: (1,2), (2,6), (5,6). We end up with following.

```
 1    2 |  3 |  4
---     -----------
 5    6 |  7 |  8
------------------
 9 | 10 | 11 | 12
```

At this, point if we try to remove the wall between 1 and 5 we should disallow it because that would just leave a big room. We allow only walls that separate two cells that are not already joined to be joined together.

Hint: a disjoint set could be useful.

In your description be sure to provide details on any data structure that you choose to use.

b. (5 marks)

What is the worst case run time for your algorithm? Provide an explanation of why.

c. (0 marks)

For no marks feel free to implement your solution in JavaScript. You can start it off with the files in the folder a4 (to be released in a couple of days) in the course repo. Like a1, you can test your solution by opening the index.html file in a modern browser (pick Firefox or Chrome (Edge and Safari should work too))