# CISC-235 Data Structures W23
# Assignment 3

March 25, 2023

## General Instructions

Write your own program(s) using Python. Once you complete your assignment, place all Python files in a zip file and name it according to the same method, i.e., "235-1234-Assn3.zip". Unzip this file should get all your Python file(s).

Then upload 235-1234-Assn3.zip into Assignment 3's entry on onQ. You may upload several times if you wish. However, onQ keeps only the last uploaded file. The newly uploaded file will overwrite the old file. Please check your files after uploading. We will check the latest submission you made following the required naming.

You must ensure your code is executable and document your code to help TA mark your solution. We suggest you follow PEP8[1] style to improve the readability of your code.

All data structures involved must be implemented by yourself, except for the built-in data types, i.e., List in Python.

**An "I uploaded the wrong file" excuse will result in a mark of zero.**

## 1 Word-Frequency Hash Table (40 points)

In Assignment 2, we created a modified version of AVL tree for saving word frequency (word is the key, frequency is the value associated with key) information in a given document. In this assignment, we will redesign the solution leveraging two types of hash table, i.e., closed hash table (with linear probing) and open hash table (with lists). You can not use any built-in hash-based data structure offered by Python. You should consider the "deletion" flag for the closed hash table.

Specifically, Your tasks are:

1. Implement two classes named **ClosedHashTable** and **OpenHashTable**. Both classes should have a *load_from_file(self, file_path)* function. This

[1] `https://peps.python.org/pep-0008/`

function aims to read content from a file and save word-frequency information for all words appearing in the file in the hash table. Note that you cannot calculate the word frequency outside your hash table. In other words, when you scan the word sequence, you should insert the key to the hash table and update its frequency when you see another occurrence of the word. You can reuse your A2's code to parse a document and generate word/token sequences.

2. Test the performance of your **ClosedHashTable** and **OpenHashTable** by searching a list $S$ containing $K$ words, where half of the K words are in the hash table. You need first to call your *load_from_file(self, file_path)* function to initialize your hash table using a given sample test file (named *A3test.txt*). Varying K by setting it to be 10, 20, 30, 40, 50. Print out the number of steps required for each search. For instance, K=10, means you should create a word list containing 10 words, 5 of them exist in the A3test.txt, and 5 of them do not exist. Then you perform 10 searches on the hash table for the 10 target words. Your search function should return a value indicating whether the search target exists in the hash table and, if it exists, how many steps have been taken to find the key. Hint: you can return -1, indicating the key does not exist in the hash table.

To implement the above two functionalities, you need to implement at least an insert and search function for your hash table. You also need to design your hash function and determine the size of your hash table following the tips introduced in class. You are encouraged to explain your design in comments so TAs can quickly get your idea. If you decide to follow an existing hash function, explain that in comments as well.

# 2 Graph (60 points)

## 2.1 Random Graph Generation: 10 points

Create a Graph class (and a vertex class if needed). Your graph class should have an initialization function to create a random connected graph with a specified number of vertices, and random weights on all the edges, such that the edge-weights all come from a specified range of integers.

The construction of random graphs is a huge topic and many approaches have been developed. For the purposes of this assignment, I suggest the following very simple method:

```
Let the set of vertices be {1, 2, ..., n}
for i = 2 .. n:
  x = randint(1,i-1) # x is a random integer in the range [1 .. i-1]
  let S be a randomly selected sample of x values from the set {1, ..., i-1}
  for each s in S:
    w = random(10,100)
    add an edge between vertex i and vertex s, with weight w
```

## 2.2 Compare BFS with Prim's algorithm: 50 points

Implement two functions within the graph class you created in 2.1. One for Breadth-First Search(BFS) (10 points) and the other for Prim's Minimum Spanning Tree algorithm (20 points).

Compare the efficiency of them by performing the below experiments (20 points):

```
for n = 20, 40, 60:
    repeat k times: #k is an input parameter
      generate a random graph with n vertices
      use BFS to find a spanning tree (let its total weight be B)
      use Prim to find a spanning tree (let its total weight be P)
      compute Diff #Diff is the percentage by which B is larger than P
    compute the average of the values of Diff for this value of n
Report the average value of Diff for each value of n.
```

For example, if B=20, P=10, Diff = $(20 - 10)/10 = 100\%$