## Data Structure Questions

For this project, you will answer the **six** questions laid out in the next sections. The questions cover a variety of topics related to the data structures you've learned in this course. You will write up a clean and efficient answer in Python, as well as a text explanation of the efficiency of your code and your design choices.

A qualified reviewer, such as a software or data engineer, will look over your answer and give you detailed, written feedback on areas you're doing very well in and areas that could be improved. For example, is your solution the most efficient one possible? Are you doing a good job of explaining your thoughts? Is your code elegant and easy to read?

Feel free to write this code in your local development environment. In case you do not have the ability to use your local environment or wish to use the classroom to write your code a Jupyter notebook is provided below.

1. Least Recently Used Cache

We have briefly discussed caching as part of a practice problem while studying hash maps.

The lookup operation (i.e., get ()) and put () / set () is supposed to be fast for a cache memory.

While doing the get () operation, if the entry is found in the cache, it is known as a cache hit. If, however, the entry is not found, it is known as a cache miss.

When designing a cache, we also place an upper bound on the size of the cache. If the cache is full and we want to add a new entry to the cache, we use some criteria to remove an element. After removing an element, we use the put () operation to insert the new element. The remove operation should also be fast.

For our first problem, the goal will be to design a data structure known as a **Least Recently Used (LRU) cache**. An LRU cache is a type of cache in which we remove the least recently used entry when the cache memory reaches its limit. For the current problem, consider both get and set operations as a use operation.

Your job is to use an appropriate data structure(s) to implement the cache.

* In case of a cache hit, your get () operation should return the appropriate value.
* In case of a cache miss, your get () should return -1.
* While putting an element in the cache, your put () / set () operation must insert the element. If the cache is full, you must write code that removes the least recently used entry first and then insert the element.

All operations must take O (1) time.

For the current problem, you can consider the size of cache = 5.

Here is some boiler plate code and some example test cases to get you started on this problem:

**2. Finding Files**

For this problem, the goal is to write code for finding all files under a directory (and all directories beneath it) that end with ".c"

Here is an example of a test directory listing, which can be downloaded [**here**](https://s3.amazonaws.com/udacity-dsand/testdir.zip):

./testdir

./testdir/subdir1

./testdir/subdir1/a.c

./testdir/subdir1/a.h

./testdir/subdir2

./testdir/subdir2/.gitkeep

./testdir/subdir3

./testdir/subdir3/subsubdir1

./testdir/subdir3/subsubdir1/b.c

./testdir/subdir3/subsubdir1/b.h

./testdir/subdir4

./testdir/subdir4/.gitkeep

./testdir/subdir5

./testdir/subdir5/a.c

./testdir/subdir5/a.h

./testdir/t1.c

./testdir/t1.h

Python's os module will be useful—in particular, you may want to use the following resources:

[**os.path.isdir(path)**](https://docs.python.org/3.7/library/os.path.html#os.path.isdir)

[**os.path.isfile(path)**](https://docs.python.org/3.7/library/os.path.html#os.path.isfile)

[**os.listdir(directory)**](https://docs.python.org/3.7/library/os.html#os.listdir)

[**os.path.join(...)**](https://docs.python.org/3.7/library/os.path.html#os.path.join)

**Note:** os.walk() is a handy Python method which can achieve this task very easily. However, for this problem you are not allowed to use os.walk().

# 3. Huffman Coding

A **Huffman code** is a type of optimal prefix code that is used for compressing data. The Huffman encoding and decoding schema is also lossless, meaning that when compressing the data to make it smaller, there is no loss of information.

The Huffman algorithm works by assigning codes that correspond to the relative frequency of each character for each character. The Huffman code can be of any length and does not require a prefix; therefore, this binary code can be visualized on a binary tree with each encoded character being stored on leafs.

There are many types of pseudocode for this algorithm. At the basic core, it is comprised of building a Huffman tree, encoding the data, and, lastly, decoding the data.

Here is one type of pseudocode for this coding schema:

* Take a string and determine the relevant frequencies of the characters.
* Build and sort a list of tuples from lowest to highest frequencies.
* Build the Huffman Tree by assigning a binary code to each letter, using shorter codes for the more frequent letters. (This is the heart of the Huffman algorithm.)
* Trim the Huffman Tree (remove the frequencies from the previously built tree).
* Encode the text into its compressed form.
* Decode the text from its compressed form.

You then will need to create encoding, decoding, and sizing schemas.

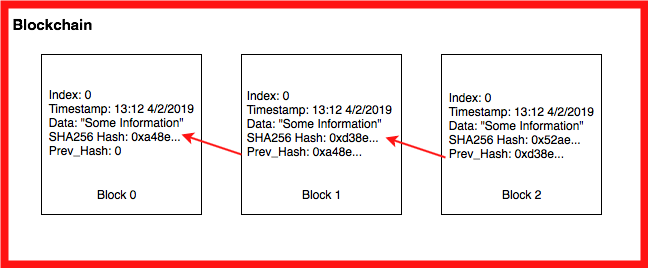
# 4. Active Directory

In Windows Active Directory, a group can consist of user(s) and group(s) themselves. We can construct this hierarchy as such. Where User is represented by str representing their ids. Write a function that provides an efficient look up of whether the user is in a group.

**5. Block chain**

A [Block chain](https://en.wikipedia.org/wiki/Blockchain) is a sequential chain of records, similar to a linked list. Each block contains some information and how it is connected related to the other blocks in the chain. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. For our block chain we will be using a [SHA-256](https://en.wikipedia.org/wiki/SHA-2) hash, the [Greenwich Mean Time](https://en.wikipedia.org/wiki/Greenwich_Mean_Time) when the block was created, and text strings as the data.

Use your knowledge of linked lists and hashing to create a block chain implementation.

**[[](https://classroom.udacity.com/nanodegrees/nd256/parts/b835ca8d-4269-4ca3-b911-c8ceb9cc0aa0/modules/a5f68248-862f-4a72-8682-24b86e2f6d61/lessons/a640374a-90af-40ad-85ff-1c6ce3948219/concepts/24216d22-1e4d-48f5-b224-9191fd5e5941)](https://classroom.udacity.com/nanodegrees/nd256/parts/b835ca8d-4269-4ca3-b911-c8ceb9cc0aa0/modules/a5f68248-862f-4a72-8682-24b86e2f6d61/lessons/a640374a-90af-40ad-85ff-1c6ce3948219/concepts/24216d22-1e4d-48f5-b224-9191fd5e5941)**

We can break the block chain down into three main parts.

First is the information hash:

We do this for the information we want to store in the block chain such as transaction time, data, and information like the previous chain.

The next main component is the block on the block chain:

Above is an example of attributes you could find in a Block class.

Finally you need to link all of this together in a block chain, which you will be doing by implementing it in a linked list. All of this will help you build up to a simple but full block chain implementation!

# 6. Union and Intersection of Two Linked Lists

Your task for this problem is to fill out the union and intersection functions. The union of two sets A and B is the set of elements which are in A, in B, or in both A and B. The intersection of two sets A and B, denoted by A ∩ B, is the set of all objects that are members of both the sets A and B.

You will take in two linked lists and return a linked list that is composed of either the union or intersection, respectively. Once you have completed the problem you will create your own test cases and perform your own run time analysis on the code.