**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 an 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:

**class** **LRU\_Cache**(object):

**def** **\_\_init\_\_**(self, capacity):

*# Initialize class variables*

**pass**

**def** **get**(self, key):

*# Retrieve item from provided key. Return -1 if nonexistent.*

**pass**

**def** **set**(self, key, value):

*# Set the value if the key is not present in the cache. If the cache is at capacity remove the oldest item.*

**pass**

our\_cache = LRU\_Cache(5)

our\_cache.set(1, 1);

our\_cache.set(2, 2);

our\_cache.set(3, 3);

our\_cache.set(4, 4);

our\_cache.get(1) *# returns 1*

our\_cache.get(2) *# returns 2*

our\_cache.get(9) *# returns -1 because 9 is not present in the cache*

our\_cache.set(5, 5)

our\_cache.set(6, 6)

our\_cache.get(3) *# returns -1 because the cache reached it's capacity and 3 was the least recently used entry*

# 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().

Here is some code for the function to get you started:

**def** **find\_files**(suffix, path):

"""

Find all files beneath path with file name suffix.

Note that a path may contain further subdirectories

and those subdirectories may also contain further subdirectories.

There are no limit to the depth of the subdirectories can be.

Args:

suffix(str): suffix if the file name to be found

path(str): path of the file system

Returns:

a list of paths

"""

**return** **None**

# OS Module Exploration Code

*## Locally save and call this file ex.py ##*

*# Code to demonstrate the use of some of the OS modules in python*

**import** os

*# Let us print the files in the directory in which you are running this script*

**print** (os.listdir("."))

*# Let us check if this file is indeed a file!*

**print** (os.path.isfile("./ex.py"))

*# Does the file end with .py?*

**print** ("./ex.py".endswith(".py"))

# Overview - Data Compression

In general, a data compression algorithm reduces the amount of memory (bits) required to represent a message (data). The compressed data, in turn, helps to reduce the transmission time from a sender to receiver. The sender encodes the data, and the receiver decodes the encoded data. As part of this problem, you have to implement the logic for both encoding and decoding.

A data compression algorithm could be either **lossy** or **lossless**, meaning that when compressing the data, there is a loss (lossy) or no loss (lossless) of information. The **Huffman Coding** is a lossless data compression algorithm. Let us understand the two phases - encoding and decoding with the help of an example.

## A. Huffman Encoding

Assume that we have a string message AAAAAAABBBCCCCCCCDDEEEEEE comprising of 25 characters to be encoded. The string message can be an unsorted one as well. We will have two phases in encoding - building the Huffman tree (a binary tree), and generating the encoded data. The following steps illustrate the Huffman encoding:

### **Phase I - Build the Huffman Tree**

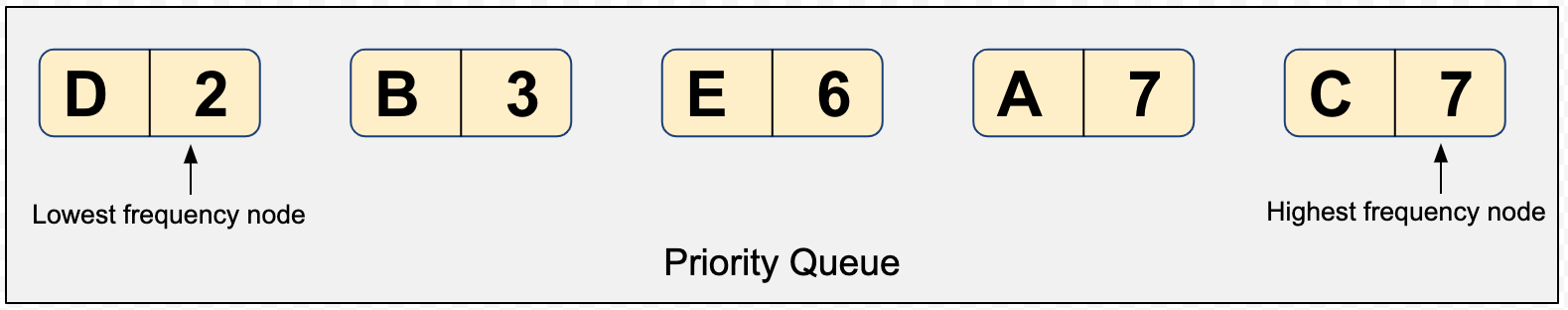
A Huffman tree is built in a bottom-up approach.

1. First, determine the frequency of each character in the message. In our example, the following table presents the frequency of each character.

| **(Unique) Character** | **Frequency** |
| --- | --- |
| A | 7 |
| B | 3 |
| C | 7 |
| D | 2 |
| E | 6 |

1. Each row in the table above can be represented as a node having a character, frequency, left child, and right child. In the next step, we will repeatedly require to pop-out the node having the lowest frequency. Therefore, build and sort a list of nodes in the order lowest to highest frequencies. Remember that a list preserves the order of elements in which they are appended.

We would need our list to work as a [**priority queue**](https://en.wikipedia.org/wiki/Priority_queue), where a node that has lower frequency should have a higher priority to be popped-out. The following snapshot will help you visualize the example considered above:

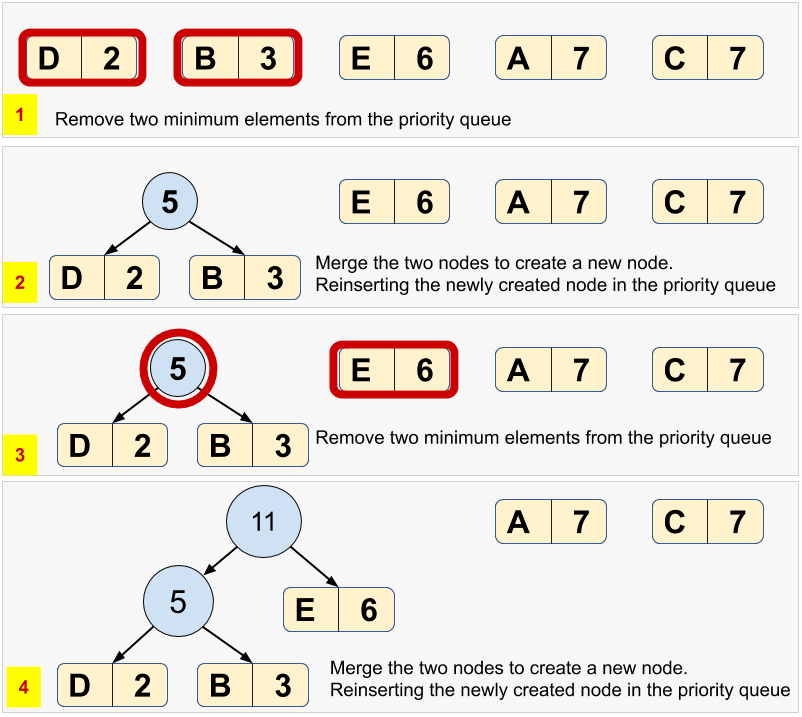


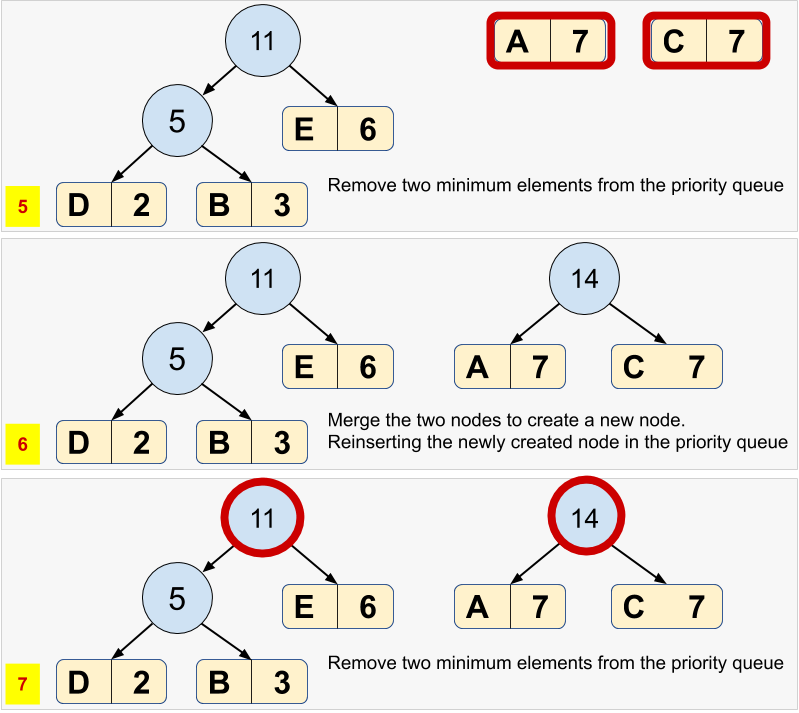
Can you come up with other data structures to create a priority queue? How about using a min-heap instead of a list? You are free to choose from anyone.

1. Pop-out two nodes with the minimum frequency from the priority queue created in the above step.
2. Create a new node with a frequency equal to the sum of the two nodes picked in the above step. This new node would become an internal node in the Huffman tree, and the two nodes would become the children. The lower frequency node becomes a left child, and the higher frequency node becomes the right child. Reinsert the newly created node back into the priority queue.

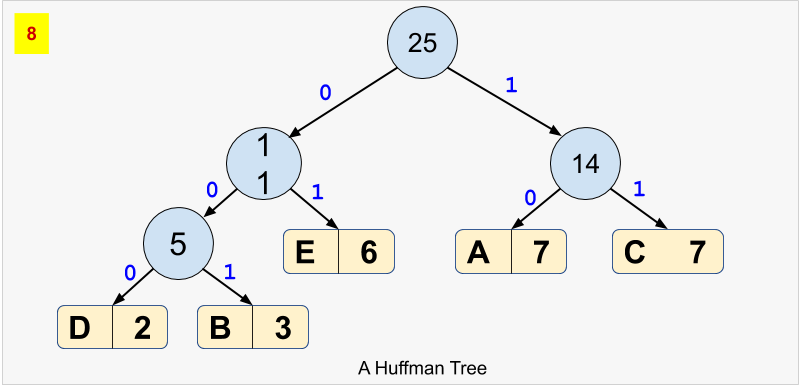
**Do you think that this reinsertion requires the sorting of priority queue again?** If yes, then a min-heap could be a better choice due to the lower complexity of sorting the elements, every time there is an insertion.

1. Repeat steps #3 and #4 until there is a single element left in the priority queue. The snapshots below present the building of a Huffman tree.





1. For each node, in the Huffman tree, assign a bit 0 for left child and a 1 for right child. See the final Huffman tree for our example:



### **Phase II - Generate the Encoded Data**

1. Based on the Huffman tree, generate unique binary code for each character of our string message. For this purpose, you'd have to traverse the path from root to the leaf node.

| **(Unique) Character** | **Frequency** | **Huffman Code** |
| --- | --- | --- |
| D | 2 | 000 |
| B | 3 | 001 |
| E | 6 | 01 |
| A | 7 | 10 |
| C | 7 | 11 |

**Points to Notice**

* Notice that the whole code for any character is **not** a prefix of any other code. Hence, the Huffman code is called a [***Prefix code***](https://en.wikipedia.org/wiki/Prefix_code).
* Notice that the binary code is shorter for the more frequent character, and vice-versa.
* The Huffman code is generated in such a way that the entire string message would now require a much lesser amount of memory in binary form.
* Notice that each node present in the original priority queue has become a leaf node in the final Huffman tree.

This way, our encoded data would be 1010101010101000100100111111111111111000000010101010101

## B. Huffman Decoding

Once we have the encoded data, and the (pointer to the root of) Huffman tree, we can easily decode the encoded data using the following steps:

1. Declare a blank decoded string
2. Pick a bit from the encoded data, traversing from left to right.
3. Start traversing the Huffman tree from the root.
   * If the current bit of encoded data is 0, move to the left child, else move to the right child of the tree if the current bit is 1.
   * If a leaf node is encountered, append the (alphabetical) character of the leaf node to the decoded string.
4. Repeat steps #2 and #3 until the encoded data is completely traversed.

You will have to implement the logic for both encoding and decoding in the following template. Also, you will need to create the sizing schemas to present a summary.

**import** sys

**def** **huffman\_encoding**(data):

**pass**

**def** **huffman\_decoding**(data,tree):

**pass**

**if** \_\_name\_\_ == "\_\_main\_\_":

codes = {}

a\_great\_sentence = "The bird is the word"

**print** ("The size of the data is: {}\n".format(sys.getsizeof(a\_great\_sentence)))

**print** ("The content of the data is: {}\n".format(a\_great\_sentence))

encoded\_data, tree = huffman\_encoding(a\_great\_sentence)

**print** ("The size of the encoded data is: {}\n".format(sys.getsizeof(int(encoded\_data, base=2))))

**print** ("The content of the encoded data is: {}\n".format(encoded\_data))

decoded\_data = huffman\_decoding(encoded\_data, tree)

**print** ("The size of the decoded data is: {}\n".format(sys.getsizeof(decoded\_data)))

**print** ("The content of the encoded data is: {}\n".format(decoded\_data))

# 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.

**class** **Group**(object):

**def** **\_\_init\_\_**(self, \_name):

self.name = \_name

self.groups = []

self.users = []

**def** **add\_group**(self, group):

self.groups.append(group)

**def** **add\_user**(self, user):

self.users.append(user)

**def** **get\_groups**(self):

**return** self.groups

**def** **get\_users**(self):

**return** self.users

**def** **get\_name**(self):

**return** self.name

parent = Group("parent")

child = Group("child")

sub\_child = Group("subchild")

sub\_child\_user = "sub\_child\_user"

sub\_child.add\_user(sub\_child\_user)

child.add\_group(sub\_child)

parent.add\_group(child)

Write a function that provides an efficient look up of whether the user is in a group.

**def** **is\_user\_in\_group**(user, group):

"""

Return True if user is in the group, False otherwise.

Args:

user(str): user name/id

group(class:Group): group to check user membership against

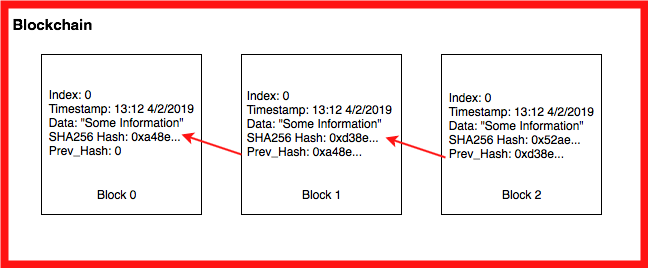
"""

**return** **None**

# Blockchain

A [**Blockchain**](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 blockchain 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 blockchain implementation.



We can break the blockchain down into three main parts.

First is the information hash:

**import** hashlib

**def** **calc\_hash**(self):

sha = hashlib.sha256()

hash\_str = "We are going to encode this string of data!".encode('utf-8')

sha.update(hash\_str)

**return** sha.hexdigest()

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 blockchain:

**class** **Block**:

**def** **\_\_init\_\_**(self, timestamp, data, previous\_hash):

self.timestamp = timestamp

self.data = data

self.previous\_hash = previous\_hash

self.hash = self.calc\_hash()

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 blockchain implementation!

# 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.

We have provided a code template below, you are not required to use it:

**class** **Node**:

**def** **\_\_init\_\_**(self, value):

self.value = value

self.next = **None**

**def** **\_\_repr\_\_**(self):

**return** str(self.value)

**class** **LinkedList**:

**def** **\_\_init\_\_**(self):

self.head = **None**

**def** **\_\_str\_\_**(self):

cur\_head = self.head

out\_string = ""

**while** cur\_head:

out\_string += str(cur\_head.value) + " -> "

cur\_head = cur\_head.next

**return** out\_string

**def** **append**(self, value):

**if** self.head **is** **None**:

self.head = Node(value)

**return**

node = self.head

**while** node.next:

node = node.next

node.next = Node(value)

**def** **size**(self):

size = 0

node = self.head

**while** node:

size += 1

node = node.next

**return** size

**def** **union**(llist\_1, llist\_2):

*# Your Solution Here*

**pass**

**def** **intersection**(llist\_1, llist\_2):

*# Your Solution Here*

**pass**

*# Test case 1*

linked\_list\_1 = LinkedList()

linked\_list\_2 = LinkedList()

element\_1 = [3,2,4,35,6,65,6,4,3,21]

element\_2 = [6,32,4,9,6,1,11,21,1]

**for** i **in** element\_1:

linked\_list\_1.append(i)

**for** i **in** element\_2:

linked\_list\_2.append(i)

**print** (union(linked\_list\_1,linked\_list\_2))

**print** (intersection(linked\_list\_1,linked\_list\_2))

*# Test case 2*

linked\_list\_3 = LinkedList()

linked\_list\_4 = LinkedList()

element\_1 = [3,2,4,35,6,65,6,4,3,23]

element\_2 = [1,7,8,9,11,21,1]

**for** i **in** element\_1:

linked\_list\_3.append(i)

**for** i **in** element\_2:

linked\_list\_4.append(i)

**print** (union(linked\_list\_3,linked\_list\_4))

**print** (intersection(linked\_list\_3,linked\_list\_4))