Data Structures, Lab
CRN: 19897

Salvador Robles Herrera
ID: 80683116

Lab #5 Report

Problem 1

This problem consists on implementing a Least Recently Used Cache, which is a data structure called cache that supports get(key), put(key, value), size(), max_capacity() all in O(1) time complexity.

The main idea of this implementation is to use a doubly linked list to know the order in which the nodes are being accessed, that way the operations would be O(1) when moving or removing or adding a node. The other thing is to use a dictionary since the LRU deals with key, value pairs, so the dictionary can help to map keys to nodes in the doubly linked list. This is how I implemented the doubly linked list separated by methods:

- First create a class for the nodes that stores the key, the value mapped to that key on the dictionary, the previous node and the next node.
- The class of the doubly linked list at first contains a head and a tail with a "None" reference since there is nothing in the list at first.
- **insertEnd(self, node)** method: This just inserts a node passed as an argument to the end of the doubly linked list, which is O(1) time. If the head and tail is "None", the linked list is empty and therefore the node inserted should be the tail and the head at the same time. If not, just set the reference to the next node of the tail as the node being passed and updating that node as the tail.
- **changeHead(self, node)** method: This just sets the head to the node being passed, by checking if the doubly linked list is empty or with just one element, then just set the previous reference to the node next to the head as the passed argument and updating the reference to head.
- **moveToEnd(self, node)** method: This is not a node that is not in the linked list like "insertEnd" method, this one changes a node position that is already in the doubly linked list to the end of linked list. There are multiple scenarios of this method, like the node being the head which needs to update the head to the node next to it, or when the node is the tail (which there are no needed changes), if not just update the reference to the next and previous node of the node being changed of position and insert it to the end.
- **removeHead(self)** method: This just update the head as the next node to head.

This is how I implemented the LRU cache using the methods from the doubly linked list:

- The LRU cache class has the properties max_capacity (maximum capacity of the linked list), keys (the set that contains all the keys), the pairs (the dictionary pairing the keys and the values), the doubly linked list previously implemented and the size of the linked list
- **get(self, key)** method: If the key is not in the set of keys, then it is not in the doubly linked list so it should return "None", if not then just return the value paired to the key but

Salvador Robles Herrera
ID: 80683116

before it update the doubly linked list by moving it to the end with the method "moveToEnd", since that node should be the most recently used in the cache.

- **put(self, key, value)** method: I check if the key is already in the LRU by checking the set of keys, if it is then just update the value with the value being passed and moving the node to the end of the linked list. If not in the set, then create a note with key and value passed, add it to the set, increase the size by one, pair the key with node in the dictionary and inserting it to the end with the insertEnd method.
- **size(self) and max_capacity(self)** methods: Just returns the properties size and max_capacity of the LRU cache class.

The O(1) requisite should maintain since we are using a doubly linked list and adding only to the end or at the beginning of the linked list which are constant operations.

This are the test cases I used which passed:

```
#LRU test cases with maximum capacity of 4
lru2 = LRUCache()
print("What I got: ", lru2.max_capacity(), "Expected: 4")
print("What I got: ", lru2.get(0), " Expected: None")
lru2.put(0,10)
print("What I got: ", lru2.get(0), " Expected: 10")
print("What I got: ", lru2.size(), " Expected: 1")
lru2.put(1,20)
lru2.put(2,500)
lru2.put(3,70)
print("What I got: ", lru2.get(0), lru2.get(1), lru2.get(2), lru2.get(3),
" Expected: 10 20 500 70")

lru2.put(3,100)
lru2.put(2,100)
lru2.put(0,100)

print("What I got: ", lru2.size(), " Expected: 4")
print("What I got: ", lru2.get(0), lru2.get(2), lru2.get(3), " Expected: 1
00 20 100 100")
```

Problem 2

For this problem we need to get the k most used passwords in a given data set and returns the list in descending order, this relates to Lab 2 where there is a set of 10 million passwords, but now it is necessary for this solution to use a heap and a dictionary.

The dictionary is used to store the password and pair with the number of times the passwords has appear in the dataset, this is done by looping thorough all the passwords and

incrementing it to one in the dictionary if the password is already in the dictionary and if not then add it with a value of one.

The main idea in my implementation of the class Heap is that I created a property of the Heap called “max” which is the maximum number of elements that the tree can have and the size of the tree, so if someone wants to know the top 3 used passwords out of the 10 million passwords, there is no necessity to put all those 10 million passwords in the Heap, it is better to check if a password is more used than the elements already in the Heap and if not then not add it to, if it is then add it, but if the heap is already full then replace the minimum used passwords with the current password. This are the methods for the Heap class:

- Note that I decided to use a min Heap since its easier to get the minimum value of the passwords already in the tree in O(1) so it is easier to check if the password should be added to the tree or not since its bigger to at least one of the elements.
- **remove(self, pairs)** method: First get the first element in the list containing the passwords of the Heap, this since the value would be the least used in the tree and then return it, but before returning it replace with the last element is the Heap and then perlocating it down to find its correct position in the tree.
- **parent(self, i)** method: Return the parent given by (i-1)/2, note that if i equal to zero then it has no parent.
- **leftChild(self, i) rightChild(self, i)** method: It is given by 2*i + 1 and 2*i+ 2 respectively. If it is not in the bounds of the tree, then return math.inf.
- **insert(self, value, password, pairs)** method: If it is bigger than the first element in the tree then it is necessary to add it to the heap by appending to the end and then perlocating up until its correct position, note that if the tree is full then it is needed to remove the first element with the method “remove(self,pairs)”.
- **perlocate_up(self, i, pairs)** method: If the current node is bigger than its parent then the min Heap property satisfies, if not it is needed to swap values and then keep perlocating up with the parent as the new i.
- **perlocate_down(self, i, pairs)** method: Here the password should keep perlocating down if the left and the right child are less than current “i”/parent, this since the min Heap property doesn’t satisfy, then it would be needed to perlocate down with the smaller child and swap there values. Note that there could be a case where the number of times the passwords that are in the directory is the same, therefore it would be needed to check those strings alphabetically (just like in perlocate_up and insert it would be needed to check the passwords at some point). If it is not necessary to swap positions in the tree, then don’t.

The method that will use the class Heap and the dictionary with all the passwords and times it has appear is like the following:

- **heap_sort(password_dict, k)** method: It creates a list and a Heap with maximum capacity of k elements. Then loop thorough all the passwords in the dictionary and insert if to the Heap with the method “insert”, note that the Heap will have at maximum k elements with the k most used password. Then it would be need to remove the k elements from the

Salvador Robles Herrera
ID: 80683116

heap with the method “remove” and inserting at the beginning of the created list, this should return a list with a descending order of the k most used elements.

This are my test cases, testing when the dictionary contains passwords with the same value paired to it:

```
sub_dictionary = {}
sub_dictionary2 = {}
for i in range(5):
  sub_dictionary[passwords_lst[i]] = password_dict[passwords_lst[i]]
  sub_dictionary2[passwords_lst[i+5]] = password_dict[passwords_lst[i+5]]

sub_dictionary3 = {'hello': 1, 'hellos': 1, 'hellp': 1, 'helli': 1, 'hella
': 1}

print("Sub: ", sub_dictionary)
print("Sub2: ", sub_dictionary2)
print("Sub2: ", sub_dictionary3)

ret = heap_sort(sub_dictionary, 5)
print("Returned list: ", ret)

ret2 = heap_sort(sub_dictionary2, 5)
print("Returned list: ", ret2)

ret3 = heap_sort(sub_dictionary3, 5)
print("Returned list: ", ret3)
```