Nancy Hernandez

Diego Aguirre

Lab 5 Report

**Introduction**

The purpose of this lab was to implement a double linked list and dictionary and fine the Least Recently Used cache.

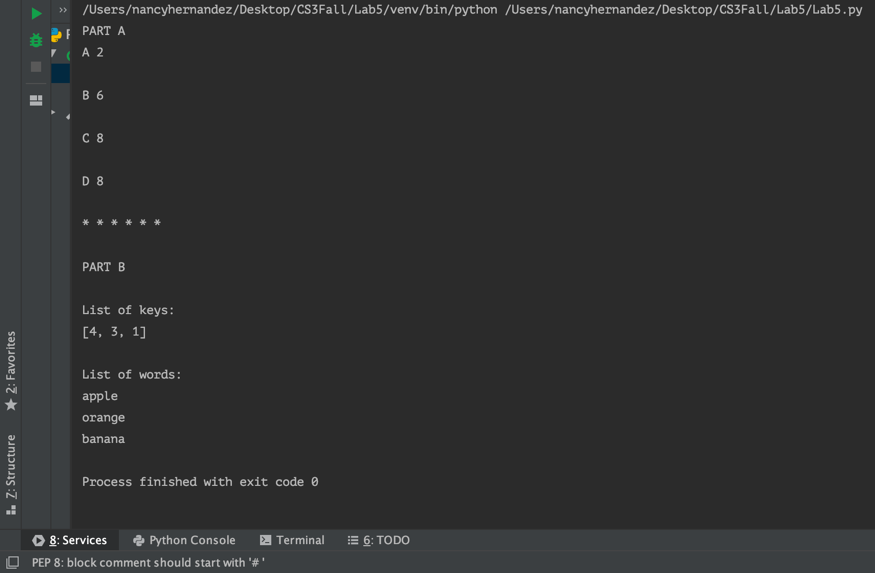
**Proposed Solution**

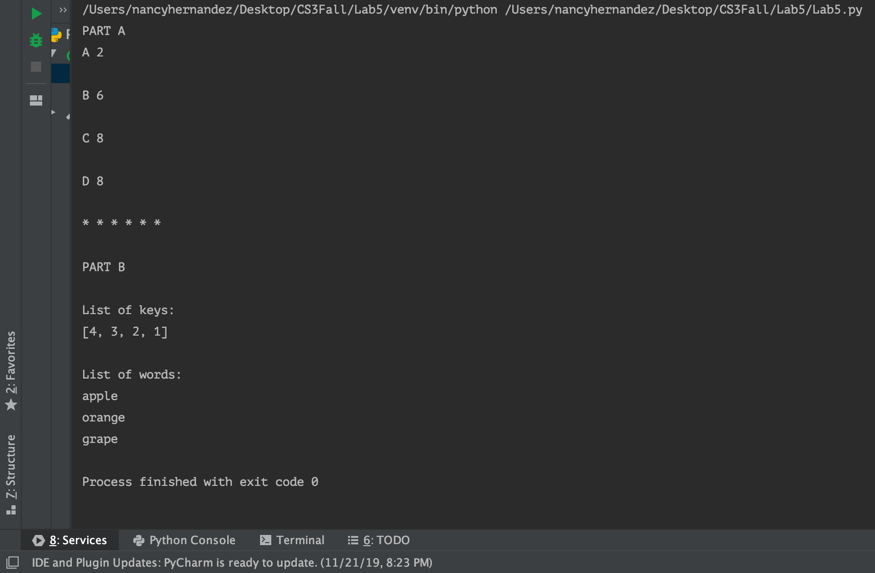
For part A I decided to create three classes, one called Node, another called Doubly liked list, and finally one called LRU. The Node class includes self.key, value, next, and prev. The Doubly Linked List head and tail. Also, in this class I implemented four different methods. One to insert at the end of the list, one to delete at the beginning of the list, one to relocate a node, and finally one to print. The insert method checks if the list is empty before anything else. Once that is checked then we create a temp and set it equal to the last node in the list. Then the new node that is going to be added is going to be set equal to temp.next. Now that that is done the tail is going to be set equal to that new node. Finally the new node.next is going to be set equal to None.

For the delete method, I created to if statements that check if the list is empty and to check if head.next is None. Once those things have been checked I set the head equal to head.next and temp.prev equal to None. The second to last method in this class is the relocate method which is in charge of placing the node in the correct place. I made two if statements and one else statement. If the node we want to move is the last one in the list then we just return if its not the last one and is the first one in the list then we set a temp equal to head.next and temp.prev equal to node.next. We then create a second temp and we set it equal to nod.next. after this the second temp.prev is going to equal to temp. finally we call the previous insert method. The print method is just in charge of printing the list.

For the LRU class I made five methods. The first method is called get and its in charge of getting the value of the key if the key exits in the cache, otherwise it just returns -1. Then we have the put method which inserts or replaces the value if the given key is not already in the cache. When the cache reaches its maximum capacity, it invalidates the LRU item before inserting a new item. Then we have the size\_list method which just returns the size of the list. Then follows the max\_capacity method which just returns the maximum capacity of the list. Finally we have another method which is identical to the other print method in the previous class which does the exact same thing.

For part B I created a class called MaxHeap which contains eight methods. These methods are is\_empty, parent, left\_child, right\_child, insert, \_percolate\_up, \_percolate\_down, and extract\_max. All of these methods I obtained from blackboard under the resources page. All of these methods are important when wanting to use a heap obviously. For the second part I also created a heap\_sort method which takes in the list as parameter and makes the list into a heap. I also created another method that takes care of putting everything into a heap and sorting everything.

**Results**



**Test cases**:

1. ["apple", "orange", "apple", "orange", "banana", "apple", "apple", "orange", "grape", "grape"]
2. ["apple", "orange", "apple", "orange", "banana", "apple", "apple", "orange"]

**Conclusion**

From this lab I learned how to use a heap and a dictionary when wanting to sort a list of words. I also learned how to implement the Least Recently Used cache using a double linked list.

**Appendix**

class MaxHeap(object):  
 # Constructor  
 def \_\_init\_\_(self):  
 self.tree = []  
  
 def is\_empty(self):  
 return len(self.tree) == 0  
  
 def parent(self, i):  
 if i == 0:  
 return -math.inf  
 return self.tree[(i - 1) // 2]  
  
 def left\_child(self, i):  
 c = 2 \* i + 1  
 if c >= len(self.tree):  
 return -math.inf  
 return self.tree[c]  
  
 def right\_child(self, i):  
 c = 2 \* i + 2  
 if c >= len(self.tree):  
 return -math.inf  
 return self.tree[c]  
  
 def insert(self, item):  
 self.tree.append(item)  
 self.\_percolate\_up(len(self.tree) - 1)  
  
 def \_percolate\_up(self, i):  
 if i == 0:  
 return  
 parent\_index = (i - 1) // 2  
 if self.tree[parent\_index] < self.tree[i]:  
 self.tree[i], self.tree[parent\_index] = self.tree[parent\_index], self.tree[i]  
 self.\_percolate\_up(parent\_index)  
  
 def extract\_max(self):  
 if len(self.tree) < 1:  
 return None  
 if len(self.tree) == 1:  
 return self.tree.pop()  
 root = self.tree[0]  
 self.tree[0] = self.tree.pop()  
 self.\_percolate\_down(0)  
 return root  
  
 def \_percolate\_down(self, i):  
 if self.tree[i] >= max(self.left\_child(i), self.right\_child(i)):  
 return  
 max\_child\_index = 2 \* i + 1 if self.left\_child(i) > self.right\_child(i) else 2 \* i + 2  
 self.tree[i], self.tree[max\_child\_index] = self.tree[max\_child\_index], self.tree[i]  
 self.\_percolate\_down(max\_child\_index)

**Academic Dishonesty Statement**

I, Nancy Hernandez, was not involved in any copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data in laboratory reports. Neither did I participate in any type of collusion involving collaboration with another person to commit an academically dishonesty.