Stephanie Galvan

Date: November 22, 2019

Lab #5 – Problem A & B

CS 2302 – Datat Structures MW 1:30PM – 2:50PM

Instructor: Diego Aguirre

TA: Gerardo Barraza

Fall 2019

Lab 5 – Problem A & B

Introduction

The following lab is divided in two parts; part A consists on the implementation of a data structure called Least Recently Used (LRU) cache which contains operations with O(1) time complexity, and part B consists on the implementation of a heap to print the most frequent words based on a list. Lab 5 aims to teach the importance of understanding design decisions such as implementing the most efficient data structures for the given problem and considering the pros and cons of the algorithm solution's time and space complexities.

Proposed Solution Design and Implementation

PART A:

The Least Recently Used (LRU) cache is implemented by using a dictionary to store the values of the keys, a doubly linked list to identify the least recently used elements, and an int value to represent the maximum storage of the cache. When a new LRU is initialized, it takes the maximum size for the cache as a parameter. For this part, the following functions were required:

- get(key) – To accomplish this operation, the LRU class took advantage of its dictionary attribute representing the values of the keys on the cache. The method would first check if the key parameter exists on the dictionary, and if it did not it would return -1 else it would return its corresponding value.
- put(key, value) – First, this method checks if the maximum capacity has been reached. If it has not been reached, the key and value would be inserted inside the dictionary and the key would be inserted at the head of the doubly linked list representing the cache. If the cache is already at maximum capacity, the dictionary would delete the least recently used element (which is located in the tail) and then the tail would be removed and the new element would be inserted at the head.
- size() – The size refers to the length of the doubly linked list to represent the cache itself. The doubly linked list implementation contains a size attribute that it updates every time an element is removed or inserted.
- max_capacity() – This returns the value of the maximum attribute.

PART B:

Part B required the implementation of a max heap; the implementation of such used a list representing the list of words to be put on the tree, a list to represent the actual heap, and a dictionary to keep count of the occurrences of each word. When a new MaxHeap is initialized, it takes a list as a parameter and then proceeds to call the insert method. When the insert method is called, the list is sorted so that when the dictionary is populated, the dictionary keys would be sorted as well. After the dictionary has been populated (the keys being the words and the values being the number of ocurrences), the insert method iterates over the dictionary and calls a helped insert method (defined as _inserting) which takes a word as a parameter, appends it to the heap and percolates up.

Running-time

PART A:

As defined on the problem definition, the implementation of LRU cache has a running time of O(1). This was accomplished by taking advantage of the constant running time of operations of a dictionary and by keep track of the most recently used element and the last recently used elements; thus, when inserting or deleting the running time will remain constant.

PART B:

Since the implementation for part B is similar to a MaxHeap with some modifications, then the running time is O(n long n)

Experimental Results

PART A:

```
Max Capacity: 5
Inserting:
key:  1 val:  2

key:  2 val:  4
key:  1 val:  2

key:  3 val:  6
key:  2 val:  4
key:  1 val:  2

key:  4 val:  8
key:  3 val:  6
key:  2 val:  4
key:  1 val:  2

key:  5 val:  10
key:  4 val:  8
key:  3 val:  6
key:  2 val:  4
key:  1 val:  2
```

```
Insert after meeting max capacity

key:  7 val:  14
key:  5 val:  10
key:  4 val:  8
key:  3 val:  6
key:  2 val:  4

key:  6 val:  12
key:  7 val:  14
key:  5 val:  10
key:  4 val:  8
key:  3 val:  6
```

PART B:

```
Original list:
['sweet', 'dog', 'umbrella', 'sweet', 'hot', 'hat', 'paper', 'gadget', 'paper', 'paper', 'hot', 'dog', 'hat']

Part B Solution:
paper : 3
dog : 2
hat : 2
hot : 2
sweet : 2
gadget : 1
umbrella : 1
```

Conclusion

Due to the constant running time for basic operations (such as inserting, removing, searching, etc.) on sets and dictionaries, this allows for the design of better classes and running times when implementing code. The purpose of this lab is to expand on the importance on understanding how the running of different data structures and their primary uses help for the development of better code. Moreover, this lab also combines the understanding of heaps (and thus heapsort) and how its implementation can be improved/modified by adding the use of data structures such as dictionaries or sets.

Appendix

Source code found at https://github.com/stephypy/CS2302_LAB5

Code originally written in py charm.

Honesty Certification:

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

INITIALS: **SG** FULL NAME: **Stephanie Galvan** DATE: **November 22, 2019**