Project 7: Hash Tables

Due: Thursday, April 11th @ 8:00pm
This is not a team project, do not copy someone else's work.

Assignment Overview

You will be building a simple Hash Table ADT that should dynamically double in size when the load factor is greater than or equal to a threshold of 0.5. The focus of this project is understanding double hashing and how fast lookups work in hash tables.

Turning It In

Be sure to submit your project as a folder named "Project7" and include in the folder:

- hashtable.py, a Python3 file
- ReadMe.txt, a text file that includes:
  - Your name
  - Feedback on the project
  - How long it took to complete
  - A list of any external resources that you used, especially websites (make sure to include the URLs) and which function you used this information for. Please note that you can not use outside resources to solve the entire project, or use outside sources for multiple functions that leads to the solution of the project, then you are submitting someone else's work not yours. Outside sources use should not be used for more than one function, and it should not be more than few lines of code to solve that function. Please note that there is no penalty for using the programming codes shared by Onsay in Lectures and posted on D2L

Assignment Specifications

HashNode Class is completed in the skeleton file. **Do not modify this class.**
Its definition includes a key and a value. The key denotes the lookup value for all operations on the hashtable. The value represents any value assigned to a key.

For this project **keys will always be strings.**

Heap Class is partially completed in the skeleton file.
Method signatures and the following provided methods may **not be modified in any way.**

- __**init**__(self, capacity)
  - self.size - number of nodes currently in table

  - self.table - built in list that stores the nodes

- self.capacity - enforced maximum number of elements that can exist in the hash table
- self.prime_index - Maintains the index of the **largest** prime**smaller** than the capacity

- **__eq__**(self)
- **__repr__**(self)
- **_hash_1**(self, key)
- **_hash_2**(self, key)
- **primes** - A tuple containing the primes up to 1000. Used for _hash_2() method. This is a class attribute so to access the primes you must use HashTable.primes **NOT** self.primes0

Complete and implement the following methods. **Do not modify the method signatures.**

- **hash**(self, key, inserting=False)
  - Given a key string return an index in the hash table.
  - Should implement double hashing.
    - If the key exists in the hash table, return the index of the existing HashNode
    - If the key does not exist in the hash table, return the index of the next available empty position in the hash table.
    - Collision resolution should implement double hashing with hash1 as the initial hash and hash2 as the step size
    - Note - There are 2 possibilities when hashing for an index:
    - When inserting a node into the hash table we want to insert into the next available bin.
    - When performing a lookup/deletion in the hash table we want to continue until we either find the proper HashNode or until we reach a bin that has never held a value. This is to preserve the collison resolution methodology.
    - The inserting parameter should be used to differentiate between these two cases.
  - Return: type int
  - Time Complexity: Θ(1)*
  - Space Complexity: O(1)
- **insert**(self, key, value):
  - Use the key and value parameters to add a HashNode to the hash table.
  - In the event that inserting will exceed or equal a load factor of 0.5 you must grow the table to double the existing capacity.
  - Return: type None
  - Time Complexity: Θ(1)*
  - Space Complexity: O(1)*
- **get**(self, key)
  - Find the HashNode with the given key in the hash table.
  - Return: type HashNode
    - If no elements exist, return None
  - Time Complexity: Θ(1)*
  - Space Complexity: O(1)
- **delete**(self, key)
  - Removes the HashNode with the given key from the hash table .

        - If the node is found assign its key and value to None, and set the deleted flag to True
    - Return: type None

    - Time Complexity: Θ(1)*
    - Space Complexity: O(1)

- **grow**(self)
    - Double the capacity of the existing hash table.
    - Do **NOT** rehash deleted HashNodes
    - Must update self.prime_index, the value of self.prime_index should be the **index** of the largest prime **smaller** than self.capacity in the HashTable.primes tuple.
    - Return: type None

    - Time Complexity: O(N)
    - Space Complexity: O(N)

Application Problem

Given a string **S** with no spaces or punctuation and a list of words (lexicon) return the frequency of the **longest words in the string.** It is guaranteed that **S** can be split into a set of distinct words. In example 1 below both "mad" and "madness" are in the dictionary, however, we consider only the longest that is in our lexicon.
Example: word_frequency("marchmadness", ["march", "mad", "madness"]) == HashTable({"march": 1, "mad": 0, "madness": 1})

In example 2 below, the end of one word could be the beginning of the next word, but if we were to take "fall" as our full word, the remaining string "che" would not be in our lexicon so we must select "all", and "chef" as the split for this string.
Example 2: word_frequency("chefall", ["chef", "all" "fall"]) == HashTable({"chef": 1, "all": 1, "fall": 0})

- **word_frequency**(string, lexicon, table):
    - Parameters
        - string: type string
        - lexicon: type List[string]
        - table: type HashTable
    - Return: type HashTable[string: int]

    - The **table** should contain all words in the lexicon along with their frequencies.
    - Time Complexity: O(L + N^2)
        - N is len(string)
        - L is the len(lexicon)
    - Space Complexity: O(L)
        - L is the number of words in the lexicon

    - **Using a python dictionary will result in a 0.**
    - Every string will have a guaranteed exact split.
    - All words in the string will appear in the lexicon.
    - No processing/validating should or needs to be done on the string, such as stripping for whitespace, etc.

Assignment Notes

- **No use of dictionaries is permitted**
- Only allowed container/collection types (or class methods) allowed are built in lists

- As always, methods and attributes with a leading underscore should not be called outside of the class definition
- Capacity will not grow past ~1000
- Guaranteed that only keys of type string will be inserted.

Rubric

- Tests (80)
- Manual (20)
  - Time Complexity
    - insert, get, delete, grow __ / 6
  - Space Complexity
    - insert, get, delete, grow __ / 6
  - Space & Time for Application __/ 8

Project Designed by Yash Vesikar

ACTIONS