10.2 - Week 10 - Applied - Practical

Introduction

Objectives

- Becoming better acquainted with functionality of hash tables and possible extensions
- To be able to justify some design choices made when creating hash tables (resizing, deleting, probing)

Different Deletions

In `hashtable.py`, modify the two classes `HashTableEmpty` and `HashTableReinsert` so that deletion is allowed, and is done in the following way:

- For `HashTableEmpty`, deleting should place an `ITEM_EMPTY` into the position, and probing should pass over this.
- For `HashTableReinsert`, deleting should reinsert every element after this element in the probe chain.

After doing this, discuss in `hashtable.txt` what the pros / cons of using both methods are. Would you want to use one method or the other when the load factor is large / small?

Resize, Rehash, Repeat

Consider a hash table implementation where collisions are handled with linear probing and where **rehash** is a method that is used whenever the load factor reaches or exceeds 0.5 by doubling the size of the table and reinserting each key.

The hash function used by this hash table implementation is given below (assuming it is part of the HashTable class shown in lectures and key is a number):

```
def hash(self, key):
  return (key + 7) % self.table_size
```

Question 1

Assuming the initial size of the table is 4, what is the size of the table after inserting 12 different keys into the above hash table? Explain.

No response

Question 2

During rehash, we need to reinsert every key from the old array into the new hash table by using the hash function again. Why is this necessary? What would happen if we simply copy the contents of the old array into the new resized array?

No response

Probe-Ability

For a given (partially filled) hash table, let's define the `probe-ability` as the following:

- For each index in the `hash_table`, simulate inserting a new value into the hash table (without actually inserting it). Remember the length of the probe chain on this insertion.
- Sum all of the probe chain lengths together, and then divide by `table_size` - this is the `probe-ability`.

In the hash table implementation given, implement a method `probeability(self) -> float`, that returns the probe-ability of the given hash table.

In the docstring and comments, analyse the complexity of your approach. If your approach is O(N^2), consider what strategies could be followed to achieve O(N), or identify what road-blocks might stop you from achieving this. If your approach is O(N), try to think of what ingredients we might need to add to our hash table to give us an O(1) complexity (at the very least, we'd need to be storing more information in our hash table).

Resize

Copy your hashtable from the previous task, and add the following scheme:

- on insertion, check the probe-ability.
- if the probe-ability exceeds 3, then resize & rehash to double the table-size.

Advanced - Analysing Resize

Question

For the approach given in the previous question, what is the largest the load factor could be just before resizing? What is the smallest? (Assume a tablesize of `1907` if you must, but approximates are ok). Think about what your answer would be in general, if the `3` in the question was changed to `a`.

No response