

## Lab 5: Implementing a Separate Chaining Hash Table

For this lab you will explore an implementation of a hash table with a separate chaining collision resolution strategy.

Define a class **MyHashTable** that supports separate chaining collision resolution

- **\_\_init\_\_(self, table\_size=11):** this function takes no parameters and returns a MyHashTable object, having initialized an empty hash table. The table\_size parameter (default value of 11) is the starting size of the table (number of “slots” in the table). This function should initialize the hash table (use a Python list) and any other instance variables used in your MyHashTable class (e.g. num\_items, num\_collisions).
- **insert(self, key, item):** this function takes a key, and an item. **Keys are valid Python non-negative integers.** The function will insert the key-item pair into the hash table based on the hash value of the key mod the table size ( $\text{hash\_value} = \text{key} \% \text{table\_size}$ ). If the key-item pair being inserted into the hash table is a duplicate key, the old key-item pair will be replaced by the new key-item pair. If the insert would cause the load factor of the hash table to become greater than 1.5, the number of slots in the hash table should be increased to twice the old number of slots, plus 1 ( $\text{new\_size} = 2 * \text{old\_size} + 1$ ). After creating the new hash table, the items in the old hash table need to be rehashed into the new table.
- **get(self, key):** this function takes a key and returns the item (key, item) pair from the hash table associated with the key. If no key-item pair is associated with the key, the function raises a **LookupError** exception.
- **remove(self, key):** this function takes a key, removes the key-item pair from the hash table and returns the key-item pair. If no key-item pair is associated with the key, the function raises a **LookupError** exception.
- **size(self):** this function returns the number of key-item pairs currently stored in the hash table.
- **load\_factor(self):** this function returns the current load factor of the hash table.
- **collisions(self):** this function returns the number of collisions that have occurred during insertions into the hash table. A collision occurs when an item is inserted into the hash table at a location where one or more key-item pairs has already been inserted. When the table is resized, do not increment the number of collisions unless a collision occurs when the new key-item pair is being inserted into the resized hash table.

Demo a file **sep\_chain\_ht.py** containing the above and a file **sep\_chain\_ht\_tests.py** containing your set of test cases.

Note: You may want to implement a “list of lists” for this lab. As an example, to initialize a list of 11 empty lists, you can use the following Python statement:

```
hash_table = [[] for _ in range(11)]
```

In this case, hash\_table would be set to:

```
[[], [], [], [], [], [], [], [], [], [], []]
```

The individual lists can then be accessed with indexing, for example:

```
hash_table[5].append((16, 'cat'))
```

Would yield:

```
[[], [], [], [], [], [(16, 'cat')], [], [], [], [], []]
```