Homework 3: Hash Table

Code due: Tue., Oct. 22 at 11:59 PM (via GSC)
Self-eval due: Thu., Oct. 24 at 11:59 PM (on GSC)

You may work on your own or with one (1) partner.

The *hash table* is a data structure that implements the dictionary abstract data type, with *expected* $\mathcal{O}(1)$ time for lookup and insert operations. There are two main ways to organize a hash table: open addressing and separate chaining. In this homework assignment, you will implement a separate chaining hash table.

In `hashtable.rkt`[1], I've supplied headers for the methods that you'll need to write, along with one sorry excuse for a test. Your job is to fill in the methods and write a bunch more tests.

Orientation

The starter code defines an interface, `DICT`, that your hash table will implement:

```
interface DICT[K, V]:
    def len(self) -> nat?
    def mem?(self, key: K) -> bool?
    def get(self, key: K) -> V
    def put(self, key: K, value: V) -> NoneC
    def del(self, key: K) -> NoneC
```

That is, a `DICT`, for some key contract `K` and some value contract `V`, specifies five methods:

- `len` returns the number of associations in the dictionary.
- `mem?` returns whether a particular key is present in the dictionary.
- `get` returns the value associated with a key if the key is present, or calls `error` if the key is absent.

[1] `https://bit.ly/2VUzcUF`

- `put` associates a key with a value in the dictionary, replacing the key's previous value if already present.
- `del` removes a key and its associated value if the key present and has no effect if the key is absent.

The starter code also defines the representation (fields) and initializer method for the `HashTable` class:

```
class HashTable[K, V] (DICT):
    let _data
    let _size
    let _hasher

    def __init__(self, nbuckets: nat?, hasher: HASHER!):
        self._data   = [ None; nbuckets ]
        self._size   = 0
        self._hasher = hasher

    …
```

Field `_data` is the table itself, a vector of *buckets*, where each bucket is a singly-linked list of key–value associations. Field `_size` stores the number of associations in the hash table (which is not the same as the number of buckets, `self._data.len()`). And field `_hasher` contains the hasher, which you'll need to use to hash keys into integers; it's in the form of an object with a `.hash` method.

The `__init__` method for `HashTable` initializes `_data` to a vector of size `nbuckets` filled with empty linked lists, `_size` to 0, and `_hasher` to the supplied hasher object.

The linked list in each bucket is made out of a single `None` preceded by some number of `cons` structs, defined as follows:

```
struct cons:
    let car
    let cdr
```

(These structs are not defined in the starter code directly, but rather imported from the standard library with the line `import cons`.)

The elements of each linked list are pairs that associate each key with its value:

```
struct assoc:
    let key
    let value
```

Here's an example of a bucket containing two associations:

```
let EX_BUCKET = cons(assoc('hello', 5),
                     cons(assoc('goodbye', 7),
                          None))
```

Your task

Your job is to complete the definition of the `HashTable` class by implementing the five methods of the `DICT` interface:

1. `HashTable.len` returns the number of mappings in the hash table, which is just `self._size`.
2. `HashTable.mem?` searches the table for a key as follows. First, it hashes the key using `self._hasher.hash`; the resulting hash code modulo `self._data.len()` (the number of buckets) tells you which bucket to look in. Then, it scans the list in that bucket and returns whether any of the associations contains the given key.
3. `HashTable.get`, like `HashTable.mem?`, hashes the key and scans the indicated bucket for an association with that key. If found, it returns the value of the association; if not, it calls `error`.
4. `HashTable.put` also hashes the key to find out which bucket to look in. If the key is already in the appropriate bucket, then it replaces the associated value with the given value; otherwise, it conses a new association onto the appropriate bucket's association list. In the latter case but not the former, it also increments the size.
5. `HashTable.del` also hashes the key to find out which bucket to look in. Then it searches the linked list in the appropriate bucket, and if an

association with the given key is present, it removes that association and decrements the size.

Ordinarily hash tables are dynamically sized, which means that the `put` method is responsible for maintaining a reasonable load factor by growing the table and rehashing as needed. For this assignment, however, you do not need to implement growing and rehashing—we will assume that the initially allocated capacity suffices.

Testing

I've provided two different hash functions for testing your hash table:

- `FirstCharHasher()` constructs a hasher object whose `hash` method takes non-empty strings to the integer codes of their first characters.
- `SboxHash64()` constructs a randomly-generated hasher object whose `hash` method enjoys good properties.

The former is a bad hash function, but it can be useful for debugging because it's predictable. For example, the ASCII code for lowercase letter 'a' is 97, so if you define `h` as a `FirstCharHasher` then `h.hash('apple')` will return 97. That means that if your hash table contains the key `'apple'` then it belongs in the bucket at `self._data[97 % self._data.len()]`.

The latter class, `SboxHash64`, *generates* a good hash function that is suitable for storing a very large number of associations. You should use the sbox hasher for testing. To create a hash table that uses an sbox hasher, you need to invoke the `HashTable` constructor as follows:

```
let h = HashTable(100, SboxHash64())
```

One test is included in the starter code, but it's not nearly comprehensive, and you need to write more.

Deliverables

The provided file `hashtable.rkt`[2], containing

- definitions of the five methods described above, and
- sufficient tests to be confident of your code's correctness.

[2] `https://bit.ly/2VUzcUF`