HW3: Hash Table

Due: Thursday, October 25, at 11:59 PM, via 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.

Orientation

The starter code provides an interface, `DICT`, which 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) -> VoidC
    def del(self, key: K) -> VoidC
```

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

- `len` returns the number of mappings in the dictionary.
- `mem?` returns whether a particular key is present in the dictionary.
- `get` returns the value associated with a key, if found, or calls `error` otherwise.
- `put` associates a key with a value in the dictionary, replacing the key's value if already present.
- `del` removes a key and its value, if present.

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

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

[1] `http://goo.gl/GBt9qN`

```
    def __init__(self, nbuckets: nat?, hash: HashFunctionC(K)):
        self._hash = hash
        self._size = 0
        self._data = [ nil(); nbuckets ]

    …
```

Field `_hash` contains the hash function, which hashes keys into numbers; field `_size` stores the number of associations in the hash table; and field `_data` is a vector of *buckets*, where each bucket is a singly-linked list of key–value associations. The constructor for `HashTable` initializes `_hash` to the supplied hash function, `_size` to 0, and `_data` to a vector of size `nbuckets`, filled with empty linked lists.

The linked list in each bucket is made out of `nil` and `cons` structs, defined as follows:

```
struct nil: pass

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 list are pairs associating each key with its value:

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

Here is an example of a bucket containing two associations:

```
let EX_BUCKET = cons(assoc('hello', 5),
                     cons(assoc('goodbye', 7),
                          nil()))
```

Your task

Your job is to complete the definition of the `HashTable` class by writing 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._hash`; the resulting hash code modulo `self._data.len()` (the number of buckets) tells you which bucket to look in. Then, it searches

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 searches 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 updates the associated value to the given value; otherwise, it conses a new association onto the list in the appropriate bucket. In the latter case, it also increments the size.
5. `HashTable.del` also hashes the key to find out which bucket to look in. Then it searches the list in the bucket, and if an association with the given key is present, it removes the association and decrements the size.

Testing

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

- `first_char_hasher` is a hash function for strings that hashes each string to the code of its first character.
- `make_sbox_hash` is a function of no arguments, that, when applied, generates a new hash function for strings.

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 `first_char_hasher('apple')` returns 97. You can use this, modulo the number of buckets, to predict which bucket a key should hash to.

The latter *generates* a good hash function, suitable for storing a large number of associations. You should use an sbox hash function 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, make_sbox_hash())
```

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

Deliverables

The provided file `hashtable.rkt`, containing

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