Homework 3: Dictionaries

The *dictionary* abstract data type is possibly the most generally useful ADT. It allows one to store values associated with keys, and to look up these values given the corresponding keys. As with any ADT, multiple concrete data structures can be used to represent a dictionary. In this assignment, you will implement two of them: an *association list*, and a *hash table*.

An association list is a data structure which uses a linked list to store key-value pairs. Most of its operations rely on linear search, which gives them a worst-case time complexity of $\mathcal{O}(n)$. Still, association lists can be a good choice when we expect n to be small.

A hash table is a data structure that has *average* $\mathcal{O}(1)$ time for lookup and insert operations. We saw two main strategies for organizing a hash table in class: open addressing and separate chaining. In this assignment, you will implement a separate chaining hash table.

In `dictionaries.rkt` I've supplied stubs for the two classes you'll need to write, along with some frankly embarrassing excuses for tests. Your job is to fill in the methods and write a bunch more tests, as well as to write a small piece of code that uses your dictionaries.

Dictionaries

The starter code defines an interface, `DICT`, which both your association list and 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`, provides five methods, which should behave as follows:

- `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 the key is present, or calls `error` otherwise.
- `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.

Association list

The first kind of dictionary you must implement is an association list.

Association lists use linked lists of key-value pairs as their representation; you will need to figure out how to represent that in your code.

You will also need to implement the five methods from the `DICT` interface as well as a constructor, which does not take any arguments (aside from `self`) and initializes an empty association list.

There are a few possible strategies for insertion in an association list, some of which we discussed in class:

1. assume there won't be any duplicate keys and always insert a new key–value pair at the front;
2. look for the key, then only insert the new key–value pair if the key doesn't exist, otherwise throw an error; or
3. look for the key, then either update the existing key–value pair with a new value or insert a new key-value pair.

For this assignment, you must write a general-purpose association list which does not assume that no one will ever use duplicate keys and which allows updating the values of existing keys. Therefore, you must use the third insertion strategy.

Hash table

The second dictionary you must implement is a separate chaining hash table.

To help you get started, the starter code provides you with definitions for the fields you will need, as well as a partial definition for the constructor:

```
class HashTable[K, V] (DICT):
    let _hash
    let _size
    let _data
    def __init__(self, nbuckets: nat?, hash: FunC[AnyC, nat?]):
        self._hash = hash
    …
```

In this code, the `_hash` field stores the hash function, which you will use to hash keys into natural numbers. The `_size` field is for storing the number of key-value mappings in the dictionary. The `_data` field is intended to store your vector of *buckets*. In a separate chaining hash table, each bucket is a linked list of key-value pairs, but the exact representation is up to you. You are welcome to add more fields and change the body constructor as you see fit, but you must leave the constructor's signature (*i.e.,* its parameters) as is, so that we can test your code.

Warning: The DSSL2 vector comprehension syntax (*i.e.,* `[init ; size]`) evaluates `init` only once! Beware aliasing.

Your job is to finish implementing the representation of your hash table, as well as to implement all the operations from the `DICT` interface. And to write high-quality tests, of course.

Ordinarily, to achieve $\mathcal{O}(1)$ average complexity as we discussed in class, 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 hash tables

I've provided two different hash functions to help you test 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()` (imported) creates a new hash function every time it gets called. These hash functions can hash keys of any kind.

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 also test with an sbox hash function. To create a hash table that uses an sbox hash function, call the `HashTable` constructor like so:

```
let h = HashTable(32, make_sbox_hash())
```

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

Warning: because hash functions generated using `make_sbox_hash()` are random, it's possible for two keys to collide with one such hash function, but not with another. In turn, if your hash table implementation does not handle collisions correctly, you may get intermittent test failures. Or worse, you may not see failures when you run your tests, but we will when grading. To avoid surprises, be sure to have some tests that force collisions using a predictable hash function.

Advice: if you find yourself writing code that does the same thing (or similar things) in multiple places in your program, you should seriously consider writing it only once (e.g., as a helper function, or a helper class), then reusing it elsewhere. By having a single version of the code (as opposed to copy-pasting it around), you avoid the risk of multiple pieces of code getting out of sync if you need to make a change, when fixing a bug for example. In general, it is good programming practice to reuse code wherever possible.

Using dictionaries

The final part of your task is to write a short piece of code that uses the dictionaries you just wrote.

You are planning a trip abroad to a country where English is not the primary language. Thinking ahead, you are building a short phrasebook to map various words in the language spoken at your destination to both their English translation and their pronounciation.

You must write the `compose_phrasebook` function, which accepts as its argument a (possibly empty) dictionary representing a phrasebook. The function should add key-value pairs to that dictionary mapping at least five words in a non-English language of your choice to their translation and pronounciation.

In case you're not feeling inspired, feel free to use the following Czech words:

Kolo:	Bicycle,	KOH-loh
Peněženku:	Wallet,	PEH-ehh-zhehn-koo
Záchod:	Toilet,	ZAHH-khoht
Mýdlo:	Soap,	MOOD-loh
Nebezpečí:	Emergency,	NEH-behz-peh-chee

This function should use only operations from the `DICT` interface (enforced by the `DICT!` contract on its argument), which means you will be able to test it with both kinds of dictionaries that you have implemented.

After you've written your `compose_phrasebook` function, write a test case that retrieves the pronounciation (only the pronounciation!) of one of your words.

Grading

Please submit your completed version of `dictionaries.rkt`, containing:

- definitions for the two classes and one function described above,
- sufficient tests to be confident of your code's correctness,
- and the honor code.

Be sure to remove any leftover debugging printing code, comment out any code that would cause infinite loops, and make sure that your submission can run successfully. If your submission produces excessive output, loops infinitely, or crashes, we will not be able to give either you feedback on it or credit for it. **Please run your code one last time before submitting!**

Functional Correctness

We will use four separate test suites to test your submission:

- **Basic association list**: `len`, `mem?`, `get`, and `put` in the non-update case.
- **Advanced association list**: `put` in the update case, `del`, and stress tests (to test computational complexity).
- **Basic hash table**: `len`, `mem?`, `get`, and `put` in the non-update case, including correct collision handling.
- **Advanced hash table**: `put` in the update case, `del`, and stress tests (to test computational complexity), including correct collision handling.

To get credit for a test suite, your submission must pass *all* its tests.

The outcome your submission will earn will be determined as follows:

- **Got it**: passes all four test suites.
- **Almost there**: passes both basic test suites, and fails a single advanced test suite.
- **On the way**: either passes both basic test suites, or passes both the basic and advanced test suite for one of the two topics.
- **Not yet**: does not achieve “on the way” requirements.
- **Cannot assess**: we could not successfully grade your submission.

Non-Functional Correctness

For this assignment, the self-evaluation will be specifically looking for:

- Thorough testing, including edge cases
- Rigorous checking of error cases
- Efficiency from the use of the correct representation and operations
- Reuse of code (*i.e.*, not copy-paste) where pertinent
- Good data representation choices