# 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 lists 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 *expected* $\mathcal{O}(1)$ time for lookup and insert operations. There are two main strategies for organizing a hash table: open addressing and separate chaining. We saw open addressing in class, so 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 dictionary.

In lecture, we discussed two possible strategies for insertion in an association list:

1. look for the key, then either update the existing key–value pair or insert of a new one; or
2. assume there won't be any duplicate keys and always insert a new key–value pair at the front.

For this assignment, you must write a general-purpose association list, which does not assume that no one will ever use duplicate keys. Therefore, you must use the first insertion strategy.

To make writing tests less tedious (because you'll be writing many of them), we provided some methods that provide shorthands for the methods you'll be implementing. For example, instead of writing out a call to the `mem?` method, you'll be able to use DSSL2's `in` operator with your association lists to check the presence of a key. Similarly, the starter code defines array-style read and write notation using square brackets for `get` and `put`, respectively. You don't have to use these shorthands, of course, but they can save you some typing and might make your code easier to read.

## 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.

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 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.
- `SboxHash64().hash` (imported from the standard library) creates a new hash function every time it's invoked. 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 text with an sbox hash function. To create a hash table that uses an sbox hash function, you need to invoke the `HashTable` constructor like so:

```
let h = HashTable(32, SboxHash64().hash)
```

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

**Warning**: because hash functions generated using `SboxHash64` 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 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.

## Using dictionaries

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

You must fill in the `compose_menu` function, which accepts as its argument a (possibly empty) dictionary representing a menu. The function should add associations to that dictionary mapping the names of (at least) five of your friends to the name of their favorite food and the kind of cuisine the food is from. After populating the menu, the function must return it.

If your friends are busy, or if you would prefer not to share their culinary preferences, feel free to use the answers some friends of mine gave:

- Jesse: Sushi, Japanese
- Stevie: Masala dosa, Indian
- Branden: Apple pie, American
- Steve: Pizza, Italian
- Sara: Channa masala, Indian
- Iliana: Pupusas, Salvadoran

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 call it with both kinds of dictionaries that you have implemented. Be sure to test with both of them!

After you've written your `compose_menu` function, write a test case that retrieves the cuisine (and only the cuisine!) of one of your friends' (any of them) favorite dish.

## Honor code

Every homework assignment you hand in must begin with the following definition (taken from the Provost's website[1]; see that for a more detailed explanation of these points):

```
let eight_principles = ["Know your rights.",
     "Acknowledge your sources.",
     "Protect your work.",
     "Avoid suspicion.",
     "Do your own work.",
     "Never falsify a record or permit another person to do so.",
     "Never fabricate data, citations, or experimental results.",
     "Always tell the truth when discussing your work with your instructor."]
```

If the definition is not present, you receive no credit for the assignment.

**Note:** Be careful about formatting the above in your source code! Depending on your pdf reader, directly copy-pasting may not yield valid DSSL2 formatting. To avoid surprises, be sure to test your code *after* copying the above definition.

## Deliverables

The provided file `dictionaries.rkt`, containing

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

Your code will be evaluated for correctness, resource efficiency, thoroughness, code reuse, and style.

## Submission

Your homework must be submitted via Canvas.

[1] `http://www.northwestern.edu/provost/students/integrity/rules.html`