Associated arrays are data structures that map keys to values. You can think of a regular array as mapping ints to values of whatever type you have stored in that array. If you’ve programmed in Java, the HashMap class is an associative array, as is the dictionary type in Python.

In this class, we’ll be covering hashtables, which are an implementation of an associative array. We have a hash function, which takes a key as input and gives a nonnegative int, which we use to index into an array. That int will need to be between 0 and the size of the array, m, so if you get a value outside that range, you return the index hash(k) % m.

It might be the case that two or more elements, k_i, k_j hash(k_i) % m == hash(k_j) % m. This is called a collision. We resolve this problem by using separate chaining, which goes as follows:

- We find the index of our value by hashing the key.
- If there is NULL at the index, then we know that the value is not stored in the hashtable.
- If there is a linked list, we traverse it until either we find our value - which we get by comparing keys - or the list runs out (it’s not in there).

That traversal sounds a lot like linear search, which we know to be O(n), where n is the number of elements in the list. But we want a data structure that lets us have O(1) lookups in expectation. We want to reason, then, that each chain is not particularly long. We can measure the fullness of our hashtable as its load factor, n/m, where n is the number of hashed elements, and m is the number of indices in the array. The higher the load factor, the higher the number of collisions. In the case of a particularly high load factor, we’ll want to dynamically resize the hashtable, similarly to how we did with unbounded arrays.

Library vs. Client

We’ve talked about interface versus implementation, but now we look at another distinction - library versus client.

The hashtable library is the set of functions that every hashtable should come with, no matter what type of values we’re hashing. In particular, we have the following definitions:

```c
struct ht_header; // Known to the client by the typedef
typedef struct ht_header* ht;

ht ht_new(int capacity)
//@requires capacity > 0;
    ;
 elem ht_lookup(ht H, key k); /* O(1) avg. */
 void ht_insert(ht H, elem e) /* O(1) avg. */
//@requires e != NULL;
    ;
```

On the client side, we consider the details specific to the values of the client’s type, the values that we’re hashing. We have the following functions on the client side:
typedef struct wcount* elem;
typedef string key;

int hash(key k, int m)
 //@requires m > 0;
//@ensures 0 <= \result && \result < m;

bool key_equal(key k1, key k2);

key elem_key(elem e)
//@requires e != NULL;

The client and the library are each aware of the other’s interface, but none of the implementation details. We want it to be the case that the library is general for all possible clients, and as long as the client has functions that behave as they should, it can use those functions to work for that client. Similarly, the client knows the library’s interface, so as long as its own functions conform to the expectations of the library’s functions, things are good.

Hashtable Code

Look at the hashtable code from class - http://www.cs.cmu.edu/~fp/courses/15122-f12/lectures/14-interfaces/ht.c0

Notice the library-client division. It’s not as distinct in C0 as we’d like it to be - other languages will enforce the fact that the client should not know about the innards of the library structure. Keep that in mind as you implement your own code in the upcoming assignment.