Building More Complex Data Types in C

Hash Map

For this assignment, you will use the `struct` mechanism in C to implement a data type that represents a hash table or unordered map, modeled after Java's `Map` interface and `HashMap` type. Fortunately, we've already done a lot of the work required to create a hash map type. Our `hashmap_t` will use the `bytes_t` type from a previous assignment as a building block:

```c
// See the header file for more explanation
struct hashmap_t
{
	// Function pointers, used to make our hashmap_t generic.
	// We'll discuss these later in the pdf.
	size_t (*hash)(const uint8_t * const);

	int (*cmp)(const uint8_t * const, const uint8_t * const);

	void (*print)(FILE *, const uint8_t * const);

	// The size, in bytes, of the key/value pair, provided by the user.
	size_t pair_size;

	// The members that make up the actual hashmap:

	// The hash table, an array of bytes_t variables.
	// Each bytes_t holds zero or more key value pairs and
	// is called a "bucket".
	bytes_t * table;

	// The number of buckets in table (the array length).
	size_t buckets;

	// The number of key/value pairs in the hash map.
	size_t size;

	// The biggest the load factor can be before resizing the table.
	// The load factor is size / buckets, and when the load factor
	// is >= to the max_load_factor the table should be resized and
	// its elements rehashed and stored.
	double max_load_factor;
};

typedef struct hashmap_t hashmap_t;
```

This is both a blessing and a curse, if your `bytes_t` type operates correctly, many operations in the hash map can be simplified. However, this means you may have to track down small bugs in the `bytes_t` type. Hopefully, you've been following good programming practices and your code is well spaced and indented, as well as commented.

While you implement your hash map type, you **may use any of the C standard library string/memory functions** we discussed in class. **You may include string.h and use any function declared within.**

Requirements
You must use your `bytes_t` code to complete this assignment. You may assume `bytes.h` will be present when your code is compiled. The TAs will manually examine your code to make sure you are following these requirements. **A score of 0 will be assigned if you don't follow the requirements.**

This is a purely individual assignment!

Hash Tables

Our `hashmap_t` will really be a hash table, here's Wikipedia's definition: "a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found. " Commonly, each bucket in the hash table holds a list or an array to accommodate collisions in the hash table, i.e. if two keys map to the same bucket, the inner list or array would be able to hold them both. For this assignment, we'll represent our hash table using a dynamic array (`bytes_t * table`), where each bucket is a `bytes_t` variable (also a dynamic array) that contains all of the key/value pairs.

Anytime you want to set or get something in the hash table, you hash a key to produce an integer, then you mod (%) with the size of the table to get the right bucket:

```
sample_pair_t p;
hashmap_t hm;

strcpy(p.key, "sample");
p.value = 10;

hm->table = malloc(7 * sizeof(bytes_t));

// More on the hash function later.
size_t h = hm->hash((uint8_t *)&p);

// If the key is in the map, it will be stored in bucket b.
size_t b = h % hm->buckets;

// Look for the key/value pair in hm->table[b], maybe using your bytes_t functions?
. . .
```

You then have to search the `bytes_t` to see if the key/value pair is in the hash table. In the average case this has a (amortized) constant look up time (zero or one item in a bucket). In the worst case all the table entries map to the same bucket and you have to perform a linear search through every item in the map to find the key/value pair.

Remember: Each index is a `bytes_t`, and will store the bytes for each key/value pair. So, `hm->table[b]` is a `bytes_t` and contains its own `dflt` array, `data` pointer, `usage`, and `dim`.

Further, let's say there are two key value pairs in `hm->table[b]`, each with `sizeof(p) = 20`, so 20 bytes per key/value pair, then `hm->table[b].data` points to an array of 40 bytes, the first 20 bytes are the first key/value pair, the second 20 bytes is next key/value pair, etc.

Resizing Hash Tables

To keep look ups fast, we want to avoid the buckets filling up with multiple values. Ideally, each bucket has at most one element, unfortunately, this is rarely the case and key/value pairs tend to cluster as more values are added. To accommodate this, hash tables normally keep track of something called the load factor, which is the number of elements in the hash table divided by the number of buckets. When this load factor exceeds some predefined threshold, the table is resized. For the C++ `unordered_map` type (another hash table) the default threshold is 1.0 (or an average of 1 element per bucket) and in the Java `HashMap` it's 0.75.

Our implementation will also be able to grow as more entries are added to the table. To resize a hash table first a new, larger table is created, then each element in the old table needs to be inserted into the new table. This involves rehashing each element and remapping the element using mod (%) with the new table's size. This rehashing makes the element order unpredictable in a `HashMap` or (the logically named) `unordered_map` type. There are ways to impose or keep track of the element order but our implementation won't incorporate them.

Function Pointers
So far in class we've seen pointers to data, but you can also have pointers to code, or function pointers. Function pointers are used in many contexts, and can be stored in variables or arrays and passed as parameters. User interface code, or code that handles an event like a click, often use function pointers. Maybe most familiar, they'll look a bit like method calls from an object oriented language.

Our hash map uses 3 function pointers, the hash function, the comparison function, and a print function. By letting the user provide these functions, rather than hard coding them into the hash map, we are able to handle keys and values of any type. They will be set for you, and you shouldn't have to change them, but you will have to know how to call them:

```
// Sample structure, its exact contents aren't relevant.
sample_pair_t p1, p2;

strcpy(p1.key, "sample");
p1.value = 10;

strcpy(p2.key, "something else");
p2.value = 10;

// Calling the hash function:
size_t h = hm->hash((uint8_t *)&p1);
// Calling the comparison function, 0 means the keys are equal:
int e = hm->hash((uint8_t *)&p1, (uint8_t *)&p2);
// Calling the print function:
size_t h = hm->print((uint8_t *)&p1);

//If p1 and p2 are already uint8_t *, like inside of the hashmap_t functions:

// Calling the hash function:
size_t h = hm->hash(p1);
// Calling the comparison function, 0 means the keys are equal:
int e = hm->cmp(p1, p2);
// Calling the print function:
size_t h = hm->print(p1);
```

Operations
For this assignment an initialization function, a de-allocation function, and a print function are provided, your task is to complete the rest of the implementation. The required functions are:

```
// Status functions, these are one-liners:
size_t hashmap_size(const hashmap_t * const hm);
bool hashmap_empty(const hashmap_t * const hm);
size_t hashmap_buckets(const hashmap_t * const hm);
double hashmap_get_curr_load_factor(const hashmap_t * const hm);
double hashmap_get_max_load_factor(const hashmap_t * const hm);

// More complex functions that provide common operations for the bytes_t.
bool hashmap_set_max_load_factor(hashmap_t * const hm, const double new_max);
bool hashmap_get(const hashmap_t * const hm, uint8_t * const pair);
bool hashmap_put(hashmap_t * const hm,  const uint8_t * const pair);
bool hashmap_rehash(hashmap_t * const hm, const size_t new_buckets);
bool hashmap_erase(hashmap_t * const hm, const uint8_t * const pair);
```

This is a purely individual assignment!

What to Submit
You will submit a single `.tar` file, containing nothing but your c files: `bytes.c`, and `hashmap.c`. Be sure to conform to the specified function interfaces and do not include anything else in the archive you submit!

Your submission will be compiled with a test driver and graded according to how many cases your solution handles correctly. This assignment will be graded automatically. You will be allowed up to ten submissions for this assignment. Test your functions thoroughly before submitting it. Make sure that your function produces correct results for every test case you can think of.

The above requirements will be checked manually by a TA after the assignment is closed. Any shortcomings will be assessed a penalty, which will be applied to your score from the Curator. The course policy is that the submission that yields the highest score will be checked. If several submissions are tied for the highest score, the latest of those will be checked.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found on the course website.

Pledge:
Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//    On my honor:
//
//    - I have not discussed the C language code in my program with
//      anyone other than my instructor or the teaching assistants
//      assigned to this course.
//
//    - I have not used C language code obtained from another student,
//      or any other unauthorized source, either modified or unmodified.
//
//    - If any C language code or documentation used in my program
//      was obtained from an allowed source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    <Student Name>
```

Failure to include this pledge in a submission is a violation of the Honor Code.

This is a purely individual assignment!