Introduction

Remember Assignment 5? Seems like a long time ago! In that assignment, we built a CList data type inspired by Python's use of lists.

Having progressed through more complex data structures, we are now ready to build CDict: A dictionary similar to Python's built-in `dict` type, written in C and based on a hash table.

The following code shows how a caller might use the CDict module:

```
CDict dict = CD_new();

CD_store(dict, "Atlanta", "Hawks");
CD_store(dict, "Boston", "Celtics");
CD_store(dict, "Los Angeles", "Lakers");
CD_store(dict, "Denver", "Nuggets");

printf("The Denver team is called the %s\n", CD_retrieve(dict, "Denver"));
printf("  current load factor: %.2f\n", CD_load_factor(dict) );

CD_delete(dict, "Boston");
printf("  current size: %d\n", CD_size(dict));

printf("Printing the entire dictionary:\n");
CD_print(dict);

// overwrite an existing value
CD_store(dict, "Denver", "Broncos");
printf("After updating Denver to 'Broncos':\n");
CD_print(dict);

// add two more elements to force a rehash
CD_store(dict, "New York", "Knicks");
CD_store(dict, "Miami", "Head");

printf("After adding New York and Miami:\n");
CD_print(dict);

CD_free(dict);
```

The code above produces the following output:

```
The Denver team is called the Nuggets
  current load factor: 0.50
  current size: 3
Printing the entire dictionary:
*** capacity: 8  stored: 3  deleted: 1  load_factor: 0.50
    00: unused
    01: DELETED
    02: IN_USE key=Atlanta hash=2 value=Hawks
    03: unused
```

```
      04: unused
      05: unused
      06: IN_USE key=Los Angeles hash=6 value=Lakers
      07: IN_USE key=Denver hash=6 value=Nuggets
  After updating Denver to 'Broncos':
  *** capacity: 8  stored: 3  deleted: 1  load_factor: 0.50
      00: unused
      01: DELETED
      02: IN_USE key=Atlanta hash=2 value=Hawks
      03: unused
      04: unused
      05: unused
      06: IN_USE key=Los Angeles hash=6 value=Lakers
      07: IN_USE key=Denver hash=6 value=Broncos
  After adding New York and Miami:
  *** capacity: 16  stored: 5  deleted: 0  load_factor: 0.31
      00: unused
      01: unused
      02: unused
      03: IN_USE key=New York hash=3 value=Knicks
      04: unused
      05: unused
      06: IN_USE key=Los Angeles hash=6 value=Lakers
      07: unused
      08: IN_USE key=Miami hash=8 value=Head
      09: unused
      10: IN_USE key=Atlanta hash=10 value=Hawks
      11: unused
      12: unused
      13: unused
      14: IN_USE key=Denver hash=14 value=Broncos
      15: unused
```

CDicts are based on hash tables. A new CDict will have a hash table with 8 slots. Items can be added or deleted; slots that previously held items that have been deleted appear in the hash table with the type DELETED. When the load factor of the hash table exceeds 0.6, the CDict is automatically rehashed into a new hash table with double the number of slots. Rehashing reclaims deleted slots.

Our Data Structure

Our ExprTree implementation will be based the following data structure:

```
#define DEFAULT_DICT_CAPACITY 8
#define REHASH_THRESHOLD  0.6

typedef enum {
  SLOT_UNUSED = 0,
  SLOT_IN_USE,
  SLOT_DELETED
} CDictSlotStatus;
```

```
struct _hash_slot {
  CDictSlotStatus status;
  CDictKeyType    key;
  CDictValueType  value;
};

struct _dictionary {
  unsigned int num_stored;
  unsigned int num_deleted;
  unsigned int capacity;
  struct _hash_slot *slot;
};
```

CDict uses the term *capacity* to refer to the total number of slots that currently exist.

Assignment Detail

The CDict library has eleven functions which you will need to write:

Function	Description
CD_new	Create a new, empty CDict object
CD_free	Destroy a previously created CDict
CD_size	Return the number of elements currently held in the dictionary
CD_capacity	Return the capacity of the dictionary, which is the current size of the underlying hash table
CD_contains	Return true or false based on whether a key is found in the dictionary
CD_store	Store a (key, value) pair in the dictionary, overwriting any previous value stored with that key
CD_retrieve	Find the value associated with a key
CD_delete	Delete a key from the dictionary
CD_load_factor	Return the current load factor for the dictionary
CD_print	For debugging: Print all slots of the hash table
CD_foreach	Cause a function (supplied by the caller) to be called for each element in the dictionary

Each of these functions is fully documented in the header file `cdict.h`.

I have provided a hash function called _CD_hash, which is found in the file `cdict.c`. You should not modify this function.

I have provided the following files for you:

- **cdict.h** – the header file which documents the interface to all the functions. You should not need to modify this file.
- **cdict.c** – the implementation file. You should add your code as indicated in the comments. You should not need to modify the code I have provided, but you are free to do so.
- **cdict_test.c** – This file contains an automated test frameworks for the CDict module. You should add your test code here.
- **Makefile** – A makefile to build the project. You should be able to type 'make' from the command line in your working directory, and Make will compile the executable cdict_test.

Implementation Notes

For this assignment, we will again assume that malloc always returns a valid memory block. You should ensure this by calling assert immediately after every call to malloc, but you do not otherwise need to worry about handling the out-of-memory error case.

I have again added -fsanitize=address to the gcc flags in the Makefile. As before, your code must be free of memory leaks.

Testing

A large part of this assignment is for you to think through not just how to implement your code, but how to *prove to yourself and everyone else* that your implementation is solid. Therefore, I want YOU to write good test cases. When writing your test cases, try to think through all the possible corner cases.

We have again made scottycheck available, but I do not want you to use it in place of your own tests. Therefore scottycheck use is throttled to once every 20 minutes. To run:

```
scottycheck isse-10 <list all your .c and .h files>
```

You should not attempt scottycheck until you are passing your own test cases. If scottycheck indicates failures that are not caught by your own tests, you should spend time thinking about how to enhance your tests (as well as fixing whatever problems scottycheck identified).

Grading and Submission

To submit your code, you should create a private GitHub repo that contains your code and a README.md file; the README should (at minimum) document how to build your code. Prior to the submission deadline, you should then submit the URL of the GitHub repo on Canvas. We will compute a grade based on the state of your repo as of the submission deadline, so feel free to make changes to it up to that point.

Points will be awarded as follows:

Points	Item
1	Did you submit your GitHub URL in your Canvas submission?
4	Does your code compile without warnings? [Points are all-or-nothing here.]
10	Code legibility: Is your code indented reasonably and are variable names well selected? Are there appropriate comments? Does the code use #defines appropriately? Subjectively, does this look like good, professional code?
70	Correctness: Based on running the automated tests used by scottycheck [7 tests, 10 points per test]
15	Memory leaks: Does your code have memory leaks as reported by -fsanitize=address? [score is computed as 15 – num_blocks_leaked × 2]

Good luck and happy coding!