# CMSC 12300: Computer Science with Applications III

The University of Chicago, Spring 2021

Quiz 3

Wednesday, June 2, 2021

This quiz has a **90-minute time limit**. This timer starts as soon as you view any material on the quiz beyond this page of instructions, or when you view the starter file in your repository.

By submitting the quiz you are implicitly certifying that you have adhered to the standards of academic honesty. In particular, you are confirming that:

- You wrote your responses in one 90-minute sitting (or more time if by prior special arrangement).
- You did not discuss the quiz content with anyone else.
- You will refrain from communicating with anyone about the quiz content until after the quiz submission deadline, and refrain from any online posts until after it has been graded and returned.
- You did not refer to any materials during the quiz other than your own class notes, the class Ed Discussion site, your own prior assignments, recordings of lectures from the class, and the class web site.

You may choose when you start the quiz, but must complete all work within 90 minutes from starting. You will need to commit your completed work to your git repository. There is no chisubmit step, but please make sure you have performed a `git add` for `quiz3.c`, a `git commit`, and a `git push`. This quiz is due Wednesday, June 2 at 11:59pm Chicago time. Your submission will be collected from your git repository at that time.

This quiz consists of 4 pages, including this one. There are 2 tasks. In total, all tasks are worth 40 points.

There is a file named `quiz3.c` in the `quiz3` directory of your repository. You must edit this file to complete the tasks and commit the updated contents to your repository.

The individual tasks are explained on the following pages.

You may run and test your code, and debug it. Please keep in mind the time limit for the overall quiz, however. Also, please be aware that test code has not been provided, and writing tests would require substantial time and effort itself. Further, for multithreaded programs, the results of tests are often highly variable and sometimes not especially probative.

To compile and run, you would need to add a `main` function, then:

```
clang -Wall -lpthread quiz3.c
./a.out
```

You may not ask clarifying questions during the test; read the text of the test, including these instructions, closely and literally, and respond as best you can. It is unfair for certain students to interact with instructors while others don't. If you feel a question is unclear, state your assumptions.

As with the prior quizzes, your solutions will be evaluated both for correctness and for whether they are designed in the best possible manner.

## Introduction

For this quiz, you will adapt existing code to make it thread-safe and have appropriate behaviors in a multithreaded environment. The file `quiz3.c` provides data type definitions and function implementations for a hash table type, `htbl`, that represents a set. This hash table supports insertion (adding an element to the set), removal (removing an element from the set), and membership testing (determining if an element is in the set).

The hash table works similarly to the hash table you implemented in CS 122. Keys are strings, and a hash function is used to determine the index within the list at which a key should be stored. If that slot is already occupied by another key, the next slot is checked, and then the one after that, and so on, wrapping around to the beginning of the list, if necessary. When inserting, a key is placed in the first available (empty) slot according to this sequence. When checking for membership, slots are examined in this order until the key is encountered or an empty slot is found; in the latter case, the key is determined not to be present.

Unlike the hash table in CS 122, this hash table implements a set rather than a dictionary. Thus, there is no separate value associated with a key, and no notion of "defval". Also, rehashing is not supported; if the hash table becomes full (which we define as all but one element occupied), we will block further insertions.

Finally, unlike the hash table in CS 122, removal is supported. Because searching for an element stops when an empty spot is encountered, removal complicates things. A spot that used to be occupied by a now-removed element is marked differently than a spot that has never been occupied. We call *empty* a spot that has never been occupied, and *empty-from-removal* a spot that was previously occupied, but now is not due to a prior removal. Insertions will place elements in empty or empty-from-removal spots. Membership tests must scan until they find the element, or reach an empty spot; they must keep going if they encounter an empty-from-removal spot. Empty spots are denoted by `NULL`, while empty-from-removal spots are stored as the empty string.

The above summarizes the code you have already been provided. You do not need to fully understand every subtlety of the implementation to complete this quiz; you only need to understand it well enough to properly add synchronization to make it thread-safe.

The implementation provided is correct, but makes various simplifying assumptions to reduce clutter and complexity. For instance, insert does not look to see if the key is already in the set, and remove malfunctions if asked to remove a key that is not actually present. You are not expected to worry about or fix these limitations; assume the functions are not called with duplicate or missing elements, respectively. Further, as the empty string is used to denote an empty-from-removal slot, the empty string should never be inserted into the hash table as an actual key, nor should its membership be checked.

Here are the functions that have been provided:

- `new_htbl` creates a new hash table, with the specified number of slots
- `hash` computes the hash value of a string, according to the formula we used in CS 122

- `empty_string` determines if a string is empty (the empty string represents an empty-from-removal entry)
- `insert` adds a key to the hash table
- `htbl_remove` removes a key from the hash table
- `member` determines if a given key is in the hash table

All of the behavior described above has already been implemented. However, this code is not thread-safe. If multiple threads have access to the same hash table, they might wish to insert keys, remove keys, and check for membership, potentially at exactly the same time as another thread is accessing the same hash table. One of your goals is to address the problems that might otherwise result.

As you work to complete the tasks:

- you may change the `struct` definition if needed
- you should not change the return types or parameters of the given functions
- you can reorganize the control flow of the functions; in other words, you are not strictly required only to add lines of code, but can change existing code and introduce or modify conditionals and loops as necessary
- while modifying the code, you should keep in mind that the existing code is largely correct, and, for the most part, should be left intact

**Task 1** [*20 points*]

Modify the provided code to be thread-safe. Specifically, if multiple threads have access to the same hash table, and each may be attempting to perform insertions, removals, and membership tests at precisely the same time, the hash table code, with your modifications, should ensure that the hash table sensibly reflects the intended modifications. You must protect shared data from race conditions, in which the contents of the hash table could become corrupted because of uncoordinated, concurrent attempts to modify it in conflicting ways.

**Task 2** [*20 points*]

The hash table will be considered full if all but one of the slots is occupied. If a user attempts to insert, using the function `insert`, into a full hash table, then the code currently quits the program (using the `exit` function). Our real intention is indeed to enforce a maximum occupancy, but to do so by making any attempt to insert into a full hash table wait until another thread removes a key from the hash table, freeing up space. Remove the `exit` call and write logic to accomplish this behavior instead of exiting. Your solution must not employ "busy waiting."