CMSC 216 Project #2 Spring 2016

Hash Table I **Due: Wednesday March 2, 2016, 8PM**

In this project you will implement a hash table with the open addressing scheme (with linear probing).

1 Overview

1.1 Obtain the project files

To obtain the project files copy the folder project2 available in the 216 public directory to your 216 directory.

We have supplied three files for your use in this project: (a) `hashtable.h`, which provides prototypes for the functions your hash table must implement; (b) `.submit`, which is necessary to submit the project to the submit server; and a (c) `Makefile`, which will help you work on your project more efficiently.

1.2 Implement the hash table

Create a file named `hashtable.c` to contain your implementation of the hash table. Your hash tables need to use an open addressing, linear probing approach to resolving hash collisions. Both the keys and the values to be inserted into your hash tables are strings (`char` pointers). You *must* use the hash code algorithm defined in Section 2.2.2 to generate the hash codes for your keys, since we will be inspecting the state of your hash table, and using a different algorithm will generate incorrect results.

2 Specifications

2.1 Use of a `Makefile`

Makefiles will be covered more in-depth in class, but instead of issuing `gcc` commands every time you want to recompile your program, you can simply execute "`make`" from the command line. Make recompiles the public tests provided.

2.2 Hash Table

The hash table you will implement is a fixed-size hash table using open addressing (with linear probing to handle collisions).

2.2.1 Linear probing

This is intended as a brief review of the concept of linear probing.

Linear probing is a method to deal with the occurrence of collisions when hashing keys to buckets in a hash table. When an insertion of an element is attempted into a full bucket (located at an index i), the linear probing algorithm attempts to insert the element into the next bucket in the table (i.e., the bucket at index $i + 1$). The algorithm repeats the probing with subsequent buckets (IMPORTANT if you reach the end of the table you need to go back to the beginning of the hash table) until it either finds an empty bucket that can hold the element, or finds that all buckets are full.

Because this algorithm maps elements to a bucket that is not necessarily at index $h \pmod n$ (where h is the hash code of the key and n the total number of buckets), you need to take this possibility into account in performing search and delete operations on the hash table. Consider inserting two elements a and b that map to the same index, i. a is placed in the bucket at index i, but b goes into index $i + 1$. But if a is subsequently deleted, b remains at index $i + 1$; this displacement requires you to introduce a new bucket state, in addition to *empty* and *full*, called *deleted*, meaning that a bucket was once full, but no longer is. With the additional state, you can adapt the search and delete operations to find displaced elements in the table. Learning how to adapt those operations is part of your assignment for the project.

2.2.2 Hash code algorithm

As mentioned earlier, you are hashing strings in this project. You must write a function to generate a hash code for a string; this hash code (modulo the size of the table) provides the starting point for finding a position to insert or find a string. This is the algorithm you need to implement in the `hash_code()` function that is part of your `hashtable.c` implementation:

1. an empty string (`""`) has a hash code of 0.
2. non-empty strings have a hash code determined by finding the hash code of the string obtained by removing the last character, multiplying that code by 65599, then adding the ASCII value of the removed character.

For example, the hash code (HC) of `"ice"` is 451,845,518,507:

$$\begin{aligned}
HC(\texttt{"ice"}) &= HC(\texttt{"ic"}) \times 65599 + ASCII(\texttt{'e'}) \\
&= (HC(\texttt{"i"}) \times 65599 + ASCII(\texttt{'c'})) \times 65599 + ASCII(\texttt{'e'}) \\
&= [(HC(\texttt{""}) \times 65599 + ASCII(\texttt{'i'})) \times 65599 + ASCII(\texttt{'c'})] \times 65599 + ASCII(\texttt{'e'}) \\
&= [(0 \times 65599 + 105) \times 65599 + 99] \times 65599 + 101 \\
&= [105 \times 65599 + 99] \times 65599 + 101 \\
&= 6887994 \times 65599 + 101 \\
&= 451845518507
\end{aligned}$$

For long strings, this algorithm could result in a number that exceeds the maximum value of an `unsigned long`. Therefore, you should assume that all multiplications and additions are done modulo (`ULONG_MAX`[1] + 1); this can be done by just assigning the results of the multiplications and additions into an `unsigned long` and the hardware will perform the necessary conversions via truncation.

2.2.3 Hash table functions

You must implement the functions described below. Notice that for 0 and -1 you should use the macros SUCCESS and FAILURE defined in hashtable.h.

1. `void init_table(Table *table)`

   Initialize the table to be empty. You can assume that the table has capacity `NUM_BUCKETS`, as defined in `hashtable.h`. Do nothing if `t` is `NULL`.

2. `void reset_table(Table *table)`

   Reset the table to an empty state, meaning that all buckets are in the empty state. Do nothing if `table` is `NULL`. For this project, the functionality of this function and `init_table` are the same (this will not be the case for future projects).

3. `int insert(Table *table, const char *key, const char *val)`

   Insert a key/value pair into the table. Insertion fails if: a. either `key` or `val` is `NULL`; b. either the key or value string is longer than the `MAX_STR_SIZE` defined in `hashtable.h`; c. `t` is `NULL`; or d. the table is full.

   If the key is already in the table, its corresponding value in the table should be overwritten with the new value. Note that in this case, even if the table is full the insert succeeds.

   Return 0 if insertion is successful, -1 otherwise.

4. `int search(Table *table, const char *key, char *val)`

   Search for a key in the table. If the key is present and the parameter `val` is not `NULL`, the value that is paired with the key in the table should be copied into the buffer `val` points to (you may assume the buffer is large enough to hold the value). If either `table` or `key` is `NULL`, the search is defined as failing. If the key is present and the parameter `val` is NULL, the function will not copy the string and return 0.

[1] `ULONG_MAX` is a constant defined in `<limits.h>`, which is equal to the largest value an `unsigned long` variable can hold. As described above, you do not need to actually include `limits.h` in your program to perform operations modulo `ULONG_MAX`.

Return 0 if the key is in the table, -1 if the search fails.

5. `int delete(Table *table, const char *key)`

   Attempt to remove the key from the table. If either `table` or `key` is `NULL`, or if the key does not exist in the table, the attempt fails.

   Return 0 if the deletion succeeds, -1 otherwise.

6. `int key_count(Table *table)`

   Return the number of keys present in the table, or -1 if `table` is `NULL`.

7. `int bucket_count(Table *table)`

   Return the number of buckets the table has; return -1 if `table` is `NULL`.

8. `unsigned long hash_code(const char *str)`

   Return the hash code of the string, as defined by the algorithm in Section 2.2.2. If `str` is `NULL`, return 0.

2.3 Important Points and Hints

1. You must use the hash code algorithm described in Section 2.2.2. If you do not get the results described in the example above, or fail the public tests that check the hash algorithm, make sure you fix it before continuing.
2. Your search operations must start at the hash_code() value and not at index 0.
3. You must use the constant `NUM_BUCKETS` whenever you need to use the number of buckets in the table in your code. Our tests will define this value and your table needs to behave correctly when the value is set.
4. Some of the required functions in the hash table perform similar operations as part of fulfilling their specifications. Your style grade depends in part on your ability to avoid unnecessary code duplication, either by creating helper functions to perform common operations, or reusing the required functions in other functions.
5. Data should only be allocated statically. You may not use `malloc()` etc.
6. Make sure you use diff (rather than visual inspection) in order to check the output of your project against the expected output. If you think you are passing a test in grace, but it does not work in the submit server, probably the problem is a small typo (e.g., missing a period in the output).
7. If you pass a test in grace, but not in the submit server you may have a problem with your variable declarations. For example, you are not declararing an array that is large enough, or you are passing to a function a pointer to a character, instead of a pointer to an array of characters. You may also have variables that are not initialized (in particular set to 0) or you may have arrays that are supposed to be strings, but you don't have a null character in them. For example, if you implement the hash_code method recursively, make sure the array you pass in the recursive call is nul terminated (i.e. with the string termination character).
8. Use the key_ct field to keep track of the number of keys in the hash table.
9. If you are having trouble with your code read the debugging guide available at:

   http://www.cs.umd.edu/class/spring2016/cmsc216/content/resources/debugging.html
10. Before posting in Piazza or heading to office hours, check the debugging guide.

3 Grading Criteria

Your project grade will be determined with the following weights:

Results of public tests	20%
Results of secret tests	70%
Code style grading	10%

3.1 Style grading

For this project, your code is expected to conform to the following style guidelines:

- Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).
- No lines longer than 80 columns are allowed. You can check your code's line lengths using the `linecheck` program in grace. Just run "`linecheck filename.c`" and it will report any lines that are too long.
- Use reasonable and consistent indentation.
- Use descriptive and meaningful identifiers.
- Do not use global variables.
- Feel free to use helper functions for this project. Just make sure to define them as static.
- Each function must have, at a minimum, a comment describing its purpose and operation. If you use a complicated algorithm to implement a function, you definitely need an extra comment explaining the complicated steps of your algorithm.
- Follow the C style guidelines available at:

  `http://www.cs.umd.edu/class/spring2016/cmsc216/content/resources/coding-style.html`

4 Submission

4.1 Deliverables

The only file we will grade is hashtable.c. We will use our versions of all header files to build our tests, so do not make any changes to the header files.

4.2 Procedure

You can submit your project by executing, in your project directory (project2), the **submit** command.

Immediately after copying the project2 folder, try to submit your project (even if you have not started). Do not wait until the day the project is due in order to try the submission process.

4.3 Possible problem with submit command

If you try to submit your project in grace, and you get the error:

"Exception in thread 'main' java.lang.OutOfMemoryError: unable to create new native thread"

then close all terminals windows except one, and try to submit again.

5 Academic Integrity

Please see the syllabus for project rules and academic integrity information. All programming assignments in this course are to be written individually (unless explicitly indicated otherwise in a written project handout). Cooperation between students is a violation of the Code of Academic Integrity.