Project #4: High-Performance Hash Table ADT
Due May 10 at 11:59pm

In this project, you will build a hash table abstract data type. Your ADT will permit users to insert and search for data records in the hash table; it will also allow users to resize the hash table on-the-fly to improve data search time. Your ADT will be designed to work as the dictionary data structure in the parse.c example from lecture. In addition to building the hash table ADT, you will also rewrite the parse.c example to use your new ADT. In addition, you will run your ADT with a program that tests the resizing feature, and shows its effect on search performance.

1 Hash Table

You are to implement a hash table as an abstract data type. As described in lecture, a hash table allows users to insert and search for data in the hash table efficiently. Hash table operations require the user to provide a “key” that uniquely identifies the data the user is referring to. In this project, the hash table keys are strings. Hence, all data stored in the hash table must be associated with a string that uniquely identifies the data. A “hash function” is applied on the key to specify a bucket (linked list) within the hash table where the data is located. For insert operations, the data is added to the hashed linked list; for search operations, a linear search through the hashed linked list is performed to find the data. For performance analysis purposes, your hash table will keep track of how much data was searched during each search operation.

1.1 Data Structure Layout

Figure 1 illustrates the structures of the ADT, including the main “hash table” struct which contains a pointer to a buckets array. Each element in the buckets array points to a linked list consisting of “hash entry” structs. Each hash entry struct contains a “key,” which is a string, and a “frequency” variable, which is an integer. As mentioned above, your hash table ADT is intended to serve as the dictionary for the parse.c example from lecture. So, the “key” is a word in the dictionary, and its “frequency” is the number of times that particular word was looked up in the dictionary.
1.2 Hashing

Your hash table ADT will implement a hash function which takes as input a key (i.e., a string of any length) and returns an index that selects one of the buckets in the hash table. The hash function you implement should sum the ASCII values for every character in the key weighted by 2 raised to the power $i$, where $i$ is the index of the corresponding character, and mod by the number of buckets. In other words:

$$\text{hash}(key) = (\sum_{i=0}^{\text{strlen}(key)-1} key[i] \cdot 2^i) \mod \text{num\_buckets}$$  \hspace{1cm} (1)

This hash function will effectively distribute keys across a large number of buckets. (Hint: to reduce the chances of overflow when summing the ASCII values, it is best to use “unsigned int” types for storing the sum and the weight values).

1.3 Resizing

In addition to standard hash table functions like insert, search, and dump, your hash table ADT will also permit users to change the number of buckets at any time to control search performance. For example, the number of buckets can be increased to reduce the average number of data elements stored in each bucket. Notice, however, that the hash function from Section 1.2 depends on the number of buckets. This means that if the number of buckets changes, data elements may need to be moved to different buckets. Hence, in addition to modifying the number of buckets (i.e., creating a new pointer array in Figure 1), your ADT will also ensure all data elements are placed in the correct bucket after resizing.
1.4 ADT Interface

The interface for the hash table ADT consists of a structure template, “hash_table,” along with 7 public functions that implement the operations defined on the hash table. The structure template and public function forward declarations are defined in “hash.h,” which can be downloaded from the course website (just follow the “Project 4 Files” hyperlink).

You must implement these structure and functions in your ADT. (You may add other structures—for example a “hash_entry” structure—and private functions, but you cannot omit any of the ones defined in hash.h).

You should fill in the structure template in hash.h (the version provided is empty). Also, you should create a file, “hash.c,” that implements the 7 required public functions along with any other private functions you need. Together, the hash.h and hash.c files make up the ADT. The following describes the 7 public functions you must implement:

1. **Phash_table new_hash(int size)**

   This is the ADT constructor. This function allocates a hash table structure on the heap, and returns a pointer to it. The function takes 1 argument: the initial size (number of buckets) of the hash table. The allocated hash table should be empty (i.e., all the buckets are empty).

2. **void free_hash(Phash_table table)**

   This is the ADT destructor. This function deallocates the hash table pointed to by the argument passed into the function. All structures and arrays dynamically allocated as part of the ADT should be freed after this function is called. In particular, if the hash table is not empty, the function should deallocate all bucket linked lists.

3. **void insert_hash(Phash_table table, char *key)**

   This function inserts a data record into a hash table according to a string-based key. The function takes 2 arguments: a pointer to the hash table, and a pointer to the string that is the key for insertion. When creating the new hash entry, initialize the “frequency” field to “1” since this is the first time the key (i.e., dictionary word) has been referenced in the hash table. Note, when inserting a new hash entry into a bucket, you should always insert at the head of the bucket’s linked list. (This will be necessary to match the output in the public test vectors).

4. **int find_hash(Phash_table table, char *key)**

   This function searches for a hash entry in a hash table according to a string-based key. The function takes 2 arguments: a pointer to the hash table, and a pointer to the string that is the key for the search. If the key is found in the hash table, then you should increment the corresponding “frequency” field by 1, and return the “frequency” value. If the key is not found in the hash table, then you should return “-1”.

3
5. void stat_hash(Phash_table table, int *total, float *average_search, int *worst_search)

This function returns 3 statistics associated with a hash table: the total number of hash entries stored in the hash table, the average number of data records searched and the worst-case number of data records searched across calls to “find_hash”. The function takes 4 arguments: a pointer to the hash table, and the 3 statistics passed in by reference. The function should return the 3 statistic values to the user through the pass-by-reference arguments. (Notice the data types for the 3 statistics). The last 2 statistics should take into consideration all calls to “find_hash”. In particular, calls that resulted in no hash entry found should contribute a count equal to the total number of hash entries in the searched bucket.

6. void dump_hash(Phash_table table)

This function mirrors the “dump_dictionary” function from the parse.c example. In particular, the function should traverse every hash entry in the hash table and print to standard output those hash entries whose “frequency” field is greater than “1”. The function should print the hash entries starting from the first bucket and ending at the last bucket. Within each bucket, the function should print the hash entries in the order that they appear in the bucket’s linked list (from head to tail). The function takes a single argument which is a pointer to the hash table that should be dumped.

7. void resize_hash(Phash_table table, int size)

This function changes the number of buckets in a hash table. It takes two arguments: a pointer to the hash table, and the new number of buckets to be used. As discussed earlier, this function must ensure all data records are stored in the correct bucket after resizing. (Hint: while there are many ways to accomplish this, one approach is to simply re-insert all data records into an empty hash table employing the correct size using a hash function that takes into consideration the new number of buckets). This function should also clear any counters associated with the performance statistics returned by “stat_hash” (i.e., “average_search” and “worst_search”) so that subsequent calls to “stat_hash” only reflect the search performance of the resized hash table.

2 Applications

Now that you have implemented your hash table ADT, you will use it in two different applications.

2.1 parse.c

First, you are to rewrite the parse.c program discussed in class (see handout #14) to use your hash table ADT.
Your new parse.c program should replace the “dictionary” and “frequency” arrays with a hash table that keeps track of the words encountered thus far. The hash table you create should contain 1000 buckets. (For this application, we will not use the resizing feature of your ADT).

Once you have created the hash table ADT, you should modify the parse.c program to use them. At the beginning of the parse program, you should create a hash table by calling “new_hash.” In the parse loop, instead of calling “insert_word,” you will call “insert_hash.” Instead of calling “find_word,” you will call “find_hash.” Lastly, in the “dump_dictionary” function, you should first call “stat_hash” and print the 3 statistics values returned by that function. Then, you should call “dump_hash” to print the contents of the hash table.

2.2 performance-test.c

Second, you are to use your hash table ADT in the “performance-test” program which tests your “resize_hash” function, and illustrates the performance benefits of tuning the number of buckets. All of the code has been provided to you–simply download the “performance-test.c” and “words.txt” files from the course website. Compile the “performance-test.c” module with your “hash.c” module, and run the program.

“performance-test” inserts all of the words in the “words.txt” file into a hash table (there are a very large number of words), and then resizes the hash table several times. After each resizing, the program calls “find_hash” on every inserted word and then calls “stat_hash” to report how quickly data could be found in the hash table. Notice how performance improves as the hash table size is increased.