Homework 10: HashTable

Released Friday
Due Friday 11/04/16 - 11:59pm

Goals:

- Learn how to implement hash table using linked list
- Be able to implement basic hash table features

Prerequisites:

- Understanding of structure of hash table
- Understanding how hash function works

Given:

A skeleton code and a hashkey function will be provided.

Introduction

Every data structure has its own special characteristics. A heap or a priority queue is used when the minimum or maximum element needs to be fetched in constant time. Similarly a hash table is used to fetch, add and remove an element in constant time. For some more backround on hashtables, go to http://visualgo.net/hashtable

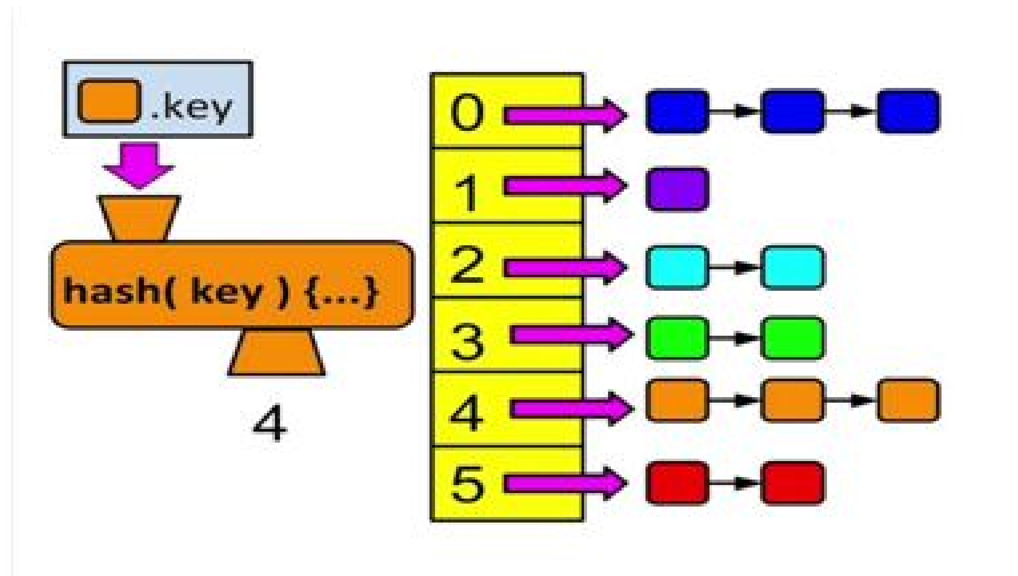
Background

Every hash-table stores data in the form of (key, value) combination. Interestingly every key is unique in a Hash Table but values can repeat which means values can be same for different keys present in it. Now as we observe in an array to fetch a value we provide the position/index corresponding to the value in that array. In a Hash Table, instead of an index we use a **key** to fetch the value corresponding to that key. Now the entire process is described below:

Every time a key is generated. The key is passed to a hash function. Every hash function has two parts a **Hash code and a Compressor**. *Hash code is an Integer number* (random or nonrandom), and it is generated by modulo operation. The entire process ensures that for any key, we get an integer position within the size of the Hash Table to insert the corresponding value.
So the process is simple, user gives a (key, value) pair set as input and based on the value generated by hash function an index is generated to where the value corresponding to the particular key is stored. So whenever we need to fetch a value corresponding to a key that is just O(1).

This picture stops being so rosy and perfect when the concept of hash collision is introduced. Imagine for different key values same block of hash table is allocated now where do the previously stored values corresponding to some other previous key go. We certainly can't replace it .That will be disastrous! To resolve this issue we will use Separate Chaining Technique.

Now what we do is make a linked list corresponding to the particular bucket of the Hash Table, to accommodate all the values corresponding to different keys who map to the same bucket.



Now there may be a scenario that all the keys get mapped to the same bucket and we have a linked list of n(size of hash table) size from one single bucket, with all the other buckets empty and this is the worst case where a hash table acts a linked list and searching is O(n).So what do we do ?

Task

For this task, you will implement a hash table. You must use chaining to handle collision. The 'table' field in the hashtable structure is a pointer to an array of '**hashtable_ent_t**' structure pointers. Each slot in the **hashtable_t** '**table**' array is a pointer to the first node in a chain of zero or more **hashtable_ent_t** structures that hold the key-value pairs. We will verify that you implement chaining. In **hashtable.c** you will implement the set of functions described in **hashtable.h:**

- hashtable_t* create_hashtable(int)
- void free_hashtable()
- int get(hashtable_t*, const char*, double*)
- int set(hashtable_t*, const char*, double)
- int key_exists(hashtable_t*, const char*)
- int remove_key(hashtable_t*, const char*)

A detailed description of each of these functions can be found in ***hashtable.h****.*

Hash Function

Your hash table will use a hash function to determine the slot in the hash table in which to store a key-value pair. You must store key-value pairs in the array entry **table[hash(key) % table_len]**. You will use the **hash()** function declared in **hashtable.h** and provided to you in compiled form in **hash.o**. To include the **hash()** function in your program, **hashtable.h** must be #include'ed in your source code files, and you must add **hash.o** to the list of *.c files being compiled using gcc.

Chaining

Since multiple key may be mapped to the same hash value, there must be a way of handling these so called 'collisions' in the hash table. For this lab you must use 'chaining', where each slot in the hash table isn't a single key-value pair, but instead is a linked list, containing zero or more key-value pairs. The **table** field in the hash table structure is a pointer to an array of pointers. If there are not key-value pairs in a given slot, that pointer at **table[hash(key) % table_len]** must be NULL to indicate that slot is empty. Otherwise, the pointer value points to the first element of a linked list of one or more **hashtable_ent** structures allocated on the heap. For example, if two key-value pairs map to the same slot in the table, then **table[hash(key) % table_len]** would contain a pointer to a **hashtable_ent** value allocated from the heap containing the first key and value, while the **next** field in that **hashtable_ent** structure would contain a pointer to a second **hashtable_ent** structure containing the second key-value pair. Finally, the second **hashtable_ent** structure would contain a NULL pointer indicating that there are no following values.