```
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
    \/\/\/\/\/\
 * CS 261 Data Structures
 * Assignment 6
 * Name: Jacob Karcz
 * Date: 11.18.2016
 *\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/
     \|/|\|/\|*/

/
    ****************************************************************************
    **************

 ******************                    File: hashMap.h
     ****************


     ***************************************************************************
     **************/

#ifndef HASH_MAP_H
#define HASH_MAP_H


#define HASH_FUNCTION hashFunction2
#define MAX_TABLE_LOAD .75

typedef struct HashMap HashMap;
typedef struct HashLink HashLink;

struct HashLink {
    char* key;
    int value;
    HashLink* next;
};

struct HashMap {
    HashLink** table;
    // Number of links in the table.
    int size;
    // Number of buckets in the table.
    int capacity;
};

HashMap* hashMapNew(int capacity);
void hashMapDelete(HashMap* map);
int* hashMapGet(HashMap* map, const char* key);
void hashMapPut(HashMap* map, const char* key, int value);
void hashMapRemove(HashMap* map, const char* key);
int hashMapContainsKey(HashMap* map, const char* key);

int hashMapSize(HashMap* map);
int hashMapCapacity(HashMap* map);
int hashMapEmptyBuckets(HashMap* map);
float hashMapTableLoad(HashMap* map);
void hashMapPrint(HashMap* map);
```

```
#endif




/
    ***************************************************************************
    **************

 ******************                    File: hashMap.c
     ****************


     ***************************************************************************
     **************/

#include "hashMap.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

int hashFunction1(const char* key) {
    int r = 0;
    for (int i = 0; key[i] != '\0'; i++) {
        r += key[i];
    }
    return r;
}

int hashFunction2(const char* key) {
    int r = 0;
    for (int i = 0; key[i] != '\0'; i++) {
        r += (i + 1) * key[i];
    }
    return r;
}

/
    ***************************************************************************
    **************
 * Creates a new hash table link with a copy of the key string.
 * param key Key string to copy in the link.
 * param value Value to set in the link.
 * param next Pointer to set as the link's next.
 * return Hash table link allocated on the heap.

     ***************************************************************************
     **************/
HashLink* hashLinkNew(const char* key, int value, HashLink* next) {
    HashLink* link = malloc(sizeof(HashLink));
    link->key = malloc(sizeof(char) * (strlen(key) + 1));
    strcpy(link->key, key);
    link->value = value;
    link->next = next;
    return link;
}

/
    ***************************************************************************
    **************
 * Free the allocated memory for a hash table link created with hashLinkNew.
 * param link

     **************************************************************************
     **************/
static void hashLinkDelete(HashLink* link) {
    free(link->key);
    free(link);
}

/
    ***************************************************************************
    **************
 * Initializes a hash table map, allocating memory for a link pointer table
     with
 * the given number of buckets.
 * param map
 * param capacity The number of table buckets.

     **************************************************************************
     **************/
void hashMapInit(HashMap* map, int capacity) {
    map->capacity = capacity;
    map->size = 0;
    map->table = malloc(sizeof(HashLink*) * capacity);
    for (int i = 0; i < capacity; i++) {
        map->table[i] = NULL;
    }
}

/
    ***************************************************************************
    **************
 * Removes all links in the map and frees all allocated memory. You can use
 * hashLinkDelete to free the links.
 * param map

     **************************************************************************
     **************/
void hashMapCleanUp(HashMap* map) {
    // FIXME: implement
        <-----------------------------------------------------------------
        <<<<<<<<<<<<<
    HashLink *current,
             *temp;
    for (int i = 0; i < map->capacity; i++) {
        current  = map->table[i];
        while (current != NULL) {
            temp = current;
            current = current->next;
            hashLinkDelete(temp);
        }    }
    free(map->table);
}

/
    ***************************************************************************
    **************
 * Creates a hash table map, allocating memory for a link pointer table with
 * the given number of buckets.
 * @param capacity The number of buckets.
 * @return The allocated map.

     **************************************************************************
     **************/
HashMap* hashMapNew(int capacity) {
    HashMap* map = malloc(sizeof(HashMap));
    hashMapInit(map, capacity);
    return map;
}

/
    ***************************************************************************
    **************
 * Removes all links in the map and frees all allocated memory, including the
 * map itself.
 * param map

     **************************************************************************
     **************/
void hashMapDelete(HashMap* map) {
    hashMapCleanUp(map);
    free(map);
}

/
    ***************************************************************************
    **************
 * Returns a pointer to the value of the link with the given key. Returns NULL
 * if no link with that key is in the table.
 *
 * Use HASH_FUNCTION(key) and the map's capacity to find the index of the
 * correct linked list bucket. Also make sure to search the entire list.
 *
 * param map
 * param key
 * return Link value or NULL if no matching link.

     **************************************************************************
     **************/
int* hashMapGet(HashMap* map, const char* key) {
    // FIXME: implement
        <-----------------------------------------------------------------
        <<<<<<<<<<<<<
    //assert (map != NULL);
    struct HashLink *link;
    int index = HASH_FUNCTION(key) % map->capacity;

    link = map->table[index];
    while (link != NULL) {
        if (strcmp(link->key, key) == 0)
            return &link->value;
        link = link->next;
    }
    return NULL;
}



/
    ***************************************************************************
    **************
 * Resizes the hash table to have a number of buckets equal to the given
 * capacity. After allocating the new table, all of the links need to be
 * rehashed into it because the capacity has changed.
 *
 * Remember to free the old table and any old links if you use hashMapPut to
 * rehash them.
 *
 * param map
 * param capacity The new number of buckets.

     **************************************************************************
     **************/
void resizeTable(HashMap* map, int capacity) {
    // FIXME: implement <--------------------------------THIS ONE
        <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    assert(map != NULL);

    //retire this map
    int oldCap = map->capacity;
    HashLink ** oldTable = map->table;

    //create a new map and rehash associations to new map
    hashMapInit(map, capacity);

    for (int i = 0; i < oldCap; i++) {
        HashLink * link = oldTable[i];
        while (link != NULL) {
            hashMapPut(map, link->key, link->value);
            link = link->next;
        }
    }

    //free memory from old table
    for (int i = 0; i < oldCap; i++) {
        HashLink* link = oldTable[i];
        while (link != NULL) {
            HashLink* temp = link;
            link = link->next;
            hashLinkDelete(temp);
        }
    }
    free(oldTable);

}

/
    ****************************************************************************
    **************
 * Updates the given key-value pair in the hash table. If a link with the given
 * key already exists, this will just update the value. Otherwise, it will
 * create a new link with the given key and value and add it to the table
 * bucket's linked list. You can use hashLinkNew to create the link.
 *
 * Use HASH_FUNCTION(key) and the map's capacity to find the index of the
 * correct linked list bucket. Also make sure to search the entire list.
 *
 * param map
 * param key
 * param value

     ***************************************************************************
     **************/
void hashMapPut(HashMap* map, const char* key, int value)
{
    // FIXME: implement
        <---------------------------------------------------------
        <<<<<<<<<<<<<<<<<<<<<

    //resize if clost to capacity
    if (hashMapTableLoad(map) >= MAX_TABLE_LOAD) {
        resizeTable(map, 2 * map->capacity);
    }

    //index
    int idx = HASH_FUNCTION(key) % (map->capacity);
    if (idx < 0) {
        idx += map->capacity;
    }

    //add to hashMap
    HashLink* curLink = map->table[idx];
    HashLink* newLink = NULL;

    if (curLink == NULL) {
        //1st link at array[index]
        newLink= hashLinkNew(key, value, NULL);
        map->table[idx] = newLink;
        map->size++;
        return;
    }

    else {
        // Bucket contains at least one link
        while (curLink != NULL) {
            if (strcmp(curLink->key, key) == 0) {
                //overwrite value of existing link
                curLink->value = value;
                return;
            }
            curLink = curLink->next;
        }
        //add to list at array[index]
        newLink = hashLinkNew(key, value, map->table[idx]);
        map->table[idx] = newLink;
        map->size++;
        return;

    }

}

/
    ***************************************************************************
    **************
 * Removes and frees the link with the given key from the table. If no such
     link
 * exists, this does nothing. Remember to search the entire linked list at the
 * bucket. You can use hashLinkDelete to free the link.
 * param map
 * param key

     **************************************************************************
     **************/
void hashMapRemove(HashMap* map, const char* key)
{
    // FIXME: implement
        <--------------------------------------------------------
        <<<<<<<<<<<<<<<<<<<<<

    //make sure the key's in the hashMap
    if (!hashMapContainsKey(map, key)) {
        return;
    }

    int index = HASH_FUNCTION(key) % (map->capacity);

    HashLink* curLink = map->table[index];
    HashLink* lastLink = map->table[index];

    if (curLink == NULL) {
        printf("no list in table[%d]\n", index);
    }

    while (curLink != NULL) {
        if (strcmp(curLink->key, key) == 0) {
            //if its the first link
            if (curLink == map->table[index]) {
                map->table[index] = curLink->next;
                hashLinkDelete(curLink);
                map->size--;
                curLink = 0;
            }
            //if its in the list
            else {
                lastLink->next = curLink->next;
                hashLinkDelete(curLink);
                map->size--;
                curLink = 0;
            }
        }
        //else move on (no else == segFault)
        else {
            lastLink = curLink;
            curLink = curLink->next;
        }
    }
}

/
    ***************************************************************************
    **************
 * Returns 1 if a link with the given key is in the table and 0 otherwise.
 *
 * Use HASH_FUNCTION(key) and the map's capacity to find the index of the
 * correct linked list bucket. Also make sure to search the entire list.
 *
 * param map
 * param key
 * return 1 if the key is found, 0 otherwise.

     **************************************************************************
     **************/
int hashMapContainsKey(HashMap* map, const char* key) {
    // FIXME: implement
        <----------------------------------------------------------------
        <<<<<<<<<<<<<<

    //compute hash value to find the correct bucket
    int index = HASH_FUNCTION(key) % map->capacity;
    if (index < 0)
        index += map->capacity;

    HashLink *currentLink;

    //traverse list and seek testE
    currentLink = map->table[index];
    while (currentLink != NULL) {
        if (strcmp(currentLink->key, key) == 0)
            return 1;
        currentLink = currentLink->next;
    }
    return 0;

}

/
    ***************************************************************************
    **************
 * Returns the number of links in the table.
 * param map
 * return Number of links in the table.

     **************************************************************************
     **************/
int hashMapSize(HashMap* map) {
    // FIXME: implement
        <--------------------------------------------------------
        <<<<<<<<<<<<<<<<<<<<<
    return map->size;
}

/
    **************************************************************************
    **************
 * Returns the number of buckets in the table.
 * param map
 * return Number of buckets in the table.

     **************************************************************************
     **************/
int hashMapCapacity(HashMap* map) {
    // FIXME: implement
        <--------------------------------------------------------
        <<<<<<<<<<<<<<<<<<<<<
    return map->capacity;
}

/
    **************************************************************************
    **************
 * Returns the number of table buckets without any links.
 * param map
 * return Number of empty buckets.

     **************************************************************************
     **************/
int hashMapEmptyBuckets(HashMap* map) {
    // FIXME: implement
        <--------------------------------------------------------
        <<<<<<<<<<<<<<<<<<<<<
    int count = 0;

    for (int i = 0; i < map->capacity; i++) {
        HashLink* link = map->table[i];
        if (link == NULL) {
            count++;
        }
    }
    return count;
}

/
    **************************************************************************
    **************
 * Returns the ratio of (number of links) / (number of buckets) in the table.
 * Remember that the buckets are linked lists, so this ratio tells you nothing
 * about the number of empty buckets. Remember also that the load is a floating
 * point number, so don't do integer division.
 * param map
 * return Table load.

     ***************************************************************************
     **************/
float hashMapTableLoad(HashMap* map) {
    // FIXME: implement
        <---------------------------------------------------------
        <<<<<<<<<<<<<<<<<<<<<
    float load = map->size / (double) map->capacity;
    return load;
}

/
    ****************************************************************************
    **************
 * Prints all the links in each of the buckets in the table.
 * param map

     ***************************************************************************
     **************/
void hashMapPrint(HashMap* map)
{
    for (int i = 0; i < map->capacity; i++)
    {
        HashLink* link = map->table[i];
        if (link != NULL)
        {
            printf("\nBucket %i ->", i);
            while (link != NULL)
            {
                printf(" (%s, %d) ->", link->key, link->value);
                link = link->next;
            }
        }
    }

    printf("\n");
}
```

```
/
```

```
    ***************************************************************************
    **************

 ******************                  File: spellChecker.c
     ****************


     ***************************************************************************
     **************/

#include "hashMap.h"
#include <assert.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * Allocates a string for the next word in the file and returns it. This string
 * is null terminated. Returns NULL after reaching the end of the file.
 * @param file
 * @return Allocated string or NULL.
 */
char* nextWord(FILE* file)
{
    int maxLength = 16;
    int length = 0;
    char* word = malloc(sizeof(char) * maxLength);
    while (1)
    {
        char c = fgetc(file);
        if ((c >= '0' && c <= '9') ||
            (c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            c == '\'')
        {
            if (length + 1 >= maxLength)
            {
                maxLength *= 2;
                word = realloc(word, maxLength);
            }
            word[length] = c;
            length++;
        }
        else if (length > 0 || c == EOF)
        {
            break;
        }
    }
    if (length == 0)
    {
        free(word);
        return NULL;
    }
    word[length] = '\0';
    return word;
}

/
    ***************************************************************************
    **************
 * HashFunction2, takes a word (string) and retuns an int hash value of the
     word (key)
 * param string
 * returns int

     **************************************************************************
     **************/
int hashFunctionz(const char* key) {
    int r = 0;
    for (int i = 0; key[i] != '\0'; i++) {
        r += (i + 1) * key[i];
    }
    return r;
}

/
    ***************************************************************************
    **************
 * Loads the contents of the file into the hash map.
 * param file
 * param map

     **************************************************************************
     **************/
void loadDictionary(FILE* file, HashMap* map) {
    // FIXME: implement
        <-----------------------------------------------------------------------
        ---------

    char * word = nextWord(file);
    int hash;

    while (word != NULL) {

        //compute hash value
        hash = hashFunctionz(word);

        if (hashMapContainsKey(map, word)) {
            //word is already in hashMap "dictionary"
        }
        else {
            //add word to hashMap
            hashMapPut(map, word, hash);
        }
        free(word);
        word = nextWord(file);
    }
}

/**
 * Prints the concordance of the given file and performance information. Uses
 * the file input1.txt by default or a file name specified as a command line
 * argument.
 * @param argc
 * @param argv
 * @return
 */
int main(int argc, const char** argv)
{
    // FIXME: implement
    HashMap* map = hashMapNew(1000);

    FILE* file = fopen("dictionary.txt", "r");
    clock_t timer = clock();
    loadDictionary(file, map);
    timer = clock() - timer;
    printf("Dictionary loaded in %f seconds\n", (float)timer / (float)
        CLOCKS_PER_SEC);
    fclose(file);

    char inputBuffer[256];
    int quit = 0;
    while (!quit)
    {
        printf("Enter a word or \"quit\" to quit: ");
        scanf("%s", inputBuffer);

        // Implement the spell checker code here..
            <----------------------------------------------------
        if (hashMapContainsKey(map, inputBuffer)) {
            printf("%s is spelled correctly\n\n", inputBuffer);
        }
        else if (! hashMapContainsKey(map, inputBuffer)) {
            printf("%s is either incorrect or not in the dictionary.\n\n",
                inputBuffer);
        }

        if (strcmp(inputBuffer, "quit") == 0)
        {
            quit = 1;
        }
    }

    hashMapDelete(map);
    return 0;
}







/
    ***************************************************************************
    **************
 ******************                    File: spellChecker.c
     ****************


     ***************************************************************************
     **************/
/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
    \/\/\/\/\/\
 * CS 261 Data Structures
 * Assignment 6
 * Name: Jacob Karcz
 * Date: 11.18.2016
 *\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/\|/|\|/
     \|/|\|/\|*/

#include "hashMap.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <assert.h>

/**
 * Allocates a string for the next word in the file and returns it. This string
 * is null terminated. Returns NULL after reaching the end of the file.
 * @param file
 * @return Allocated string or NULL.
 */
char* nextWord(FILE* file)
{
    int maxLength = 16;
    int length = 0;
    char* word = malloc(sizeof(char) * maxLength);
    while (1)
    {
        char c = fgetc(file);
        if ((c >= '0' && c <= '9') ||
            (c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            c == '\'')
        {
            if (length + 1 >= maxLength)
            {
                maxLength *= 2;
                word = realloc(word, maxLength);
            }
            word[length] = c;
            length++;
        }
        else if (length > 0 || c == EOF)
        {
            break;
        }
    }
    if (length == 0)
    {
        free(word);
        return NULL;
    }
    word[length] = '\0';
    return word;
}

/**
 * Prints the concordance of the given file and performance information. Uses
 * the file input1.txt by default or a file name specified as a command line
 * argument.
 * @param argc
 * @param argv
 * @return
 */
int main(int argc, const char** argv)
{
    // FIXME: implement
    const char* fileName = "input1.txt";
    if (argc > 1)
    {
        fileName = argv[1];
    }
    printf("Opening file: %s\n", fileName);

    clock_t timer = clock();

    //FILE* file = fopen("dictionary.txt", "r");

    HashMap* map = hashMapNew(10);

    // --- Concordance code begins here ---
    // Be sure to free the word after you are done with it here.
        <--------------------------------------

    FILE *filePtr;
    filePtr = fopen(fileName, "r"); //"r" == read
    if (filePtr == NULL) {
        printf("could not open file\n\n");
        return 0;
    }


    char *word;
    int count;

    //tally up the words
    word = nextWord(filePtr);
    while (word != NULL) {
        if (hashMapContainsKey(map, word)) {
            count = *(hashMapGet(map, word)) + 1;
            hashMapPut(map, word, count);
        }
        else {
            hashMapPut(map, word, 1);
        }
```

```c
        free(word);
        word = nextWord(filePtr);
    }
    fclose(filePtr);

    //print occurrences
    HashLink *travLink;
    for (int i = 0; i < map->capacity; i++) {
        travLink = map->table[i];
        if (travLink != NULL) {
            while (travLink != NULL) {
                printf ("%s: %d \n", travLink->key, travLink->value);
                travLink = travLink->next;
            }
        }
    }


    // --- Concordance code ends here ---

    hashMapPrint(map);

    timer = clock() - timer;
    printf("\nRan in %f seconds\n", (float)timer / (float)CLOCKS_PER_SEC);
    printf("Empty buckets: %d\n", hashMapEmptyBuckets(map));
    printf("Number of links: %d\n", hashMapSize(map));
    printf("Number of buckets: %d\n", hashMapCapacity(map));
    printf("Table load: %f\n", hashMapTableLoad(map));

    hashMapDelete(map);
    return 0;
}
```