```c
// Implements a dictionary's functionality

#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <cs50.h>

#include "dictionary.h"

// Represents a node in a hash table
typedef struct node
{
    char word[LENGTH + 1];
    struct node *next;
}
node;

// TODO: Choose number of buckets in hash table
// Yu : May use the first two letter to hash
// TODO(20220306): When set N = 26, it works, but 26*26 do not work , why ?
const unsigned int N = 26 * 26 * 26 * 26 ;

// Hash table
node *table[N];

// Counting word up
unsigned int count = 0;
void word_count(node *node);

// Search in list
bool search(node *list, const char *word);

// Returns true if word is in dictionary, else false
bool check(const char *word)
{
    // TODO
    unsigned int hash_value = hash(word);
    return search(table[hash_value], word);
}

// Hashes word to a number
```

```c
unsigned int hash(const char *word)
{
    // TODO: Improve this hash function
    // Yu : What's the problem of it ?
/*      return toupper(word[0]) - 'A';
 */
    if ((strlen(word) == 1) | (word[1] == '\''))
    {
        return toupper(word[0]) - 'A';
    }
/*      return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)- 1 ;
 */

    if ((strlen(word) == 2) | (word[2] == '\''))
    {
        return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1) - 1;
    }
/*      return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
    * (toupper(word[2]) - 'A' + 1) - 1 ;
 */

    if ((strlen(word) == 3) | (word[3] == '\''))
    {
        return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
        * (toupper(word[2]) - 'A' + 1) - 1 ;
    }

     return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
    * (toupper(word[2]) - 'A' + 1) * (toupper(word[3]) - 'A' + 1) - 1 ;

/*      if ((strlen(word) == 4) | (word[4] == '\''))
    {
        return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
        * (toupper(word[2]) - 'A' + 1) * (toupper(word[3]) - 'A' + 1) - 1 ;
    }
 */
/*      return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
    * (toupper(word[2]) - 'A' + 1) * (toupper(word[3]) - 'A' + 1)
    * (toupper(word[4]) - 'A' + 1)- 1 ;
 */
/*      if ((strlen(word) == 5) | (word[5] == '\''))
    {
        return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
        * (toupper(word[2]) - 'A' + 1) * (toupper(word[3]) - 'A' + 1)
```

```
        * (toupper(word[4]) - 'A' + 1) - 1 ;
    }
    return (toupper(word[0]) - 'A' + 1) * (toupper(word[1]) - 'A' + 1)
    * (toupper(word[2]) - 'A' + 1) * (toupper(word[3]) - 'A' + 1)
    * (toupper(word[4]) - 'A' + 1) * (toupper(word[5]) - 'A' + 1) - 1 ;
 */


}

// Loads dictionary into memory, returning true if successful, else false
bool load(const char *dictionary)
{
    // TODO
    // Open dictionary
    FILE *file = fopen(dictionary, "r");
    if (file == NULL)
    {
        printf("Could not load %s.\n", dictionary);
        return false;
    }

    // Add word into hash table
    int index = 0;
    char word[LENGTH + 1];
    char c;
    int hash_value = 0;
    while (fread(&c, sizeof(char), 1, file))
    {
        // Allow only alphabetical characters and apostrophes
        if (isalpha(c) || (c == '\'' && index > 0))
        {
            // Append character to word
            word[index] = c;
            index++;

            // Ignore alphabetical strings too long to be words
            if (index > LENGTH)
            {
                // Consume remainder of alphabetical string
                while (fread(&c, sizeof(char), 1, file) && isalpha(c));

                // Prepare for new word
                index = 0;
```

```
            }
        }

        // Ignore words with numbers (like MS Word can)
        else if (isdigit(c))
        {
            // Consume remainder of alphanumeric string
            while (fread(&c, sizeof(char), 1, file) && isalnum(c));

            // Prepare for new word
            index = 0;
        }

        // Now, we get a whole brand new word
        else if (index > 0)
        {
            // Terminate a word
            word[index] = '\0';
            hash_value = hash(word);

            // Get a memory location
            node *tmp = malloc(sizeof(node));
            if (tmp == NULL)
            {
                printf("Memory allocation error!");
                return false;
            }

            // Add word to the linked list
            strcpy(tmp->word, word);
            tmp->next = table[hash_value];
            table[hash_value] = tmp;
            index = 0;
        }
    }

    // Check whether there was an error
    if (ferror(file))
    {
        fclose(file);
        printf("Error loading %s.\n", dictionary);
        return false;
    }
```

```c
    // Close text
    fclose(file);

    return true;
}

// Returns number of words in dictionary if loaded, else 0 if not yet loaded
unsigned int size(void)
{
    // TODO:
    for (int i = 0; i < N; i++)
    {
        word_count(table[i]);
    }

    return count;
}

// Unloads dictionary from memory, returning true if successful, else false
bool unload(void)
{
    // TODO
    // Free list
    for (int i = 0; i < N; i++)
    {
        while (table[i] != NULL)
        {
            node *tmp = table[i]->next;
            free(table[i]);
            table[i] = tmp;
        }
    }
    return true;
}

// Counting word up
void word_count(node *pnode)
{
    if (pnode == NULL)
    {
        return;
    }

    // Recursion
```

```
    word_count(pnode->next);
    count++;
    return;
}

bool search(node *list, const char *word)
{
    if (list == NULL)
    {
        return false;
    }
    else if (strcasecmp(list->word, word) == 0)
    {
        return true;
    }
    else
    {
        return search(list->next, word);
    }
}
```