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

#include "dictionary.h"

// Representa um node, para o armazenamento de dados.
typedef struct node
{
    char word[LENGTH + 1];
    struct node *next;
}
node;

// Número de "buckets" para armazenar os dados.
// Escolhi 26, pois é o número de letras presentes no alfabeto.
#define N 26

// Hash table, array do tipo node que armazenará as palavras.
node *table[N];

// Variávevis globais
unsigned int valor_hash, palavra_count;

// Retorna verdadeiro se existe a palavra no dicionário e falso caso não
exista.
bool check(const char *word)
{
    // Chama a função hash para saber em qual valor do array a palavra
está armazenada.
    valor_hash = hash(word);

    // Aponta o cursor para o local que a palavra está.
    node *cursor = table[valor_hash];

    // Enquanto o cursor não for nulo, continua procurando.
    while(cursor != 0)
    {
        // Função strcasecmp compara duas strings independentemente se
tem variação de letra maiúscula e minúscula.
        if(strcasecmp(word, cursor->word) == 0)
        {
            // Retorna verdadeiro se a comparação for bem sucedida.
            return true;
        }
```

```
        // Se não for, o cursor para para a próxima palavra.
        cursor = cursor->next;
    }

    // Caso o cursor se torne nulo e a palavra não tenha sido encontrada,
retorna falso.
    return false;
}

// Descobre o índice no qual a palavra deve ser armazenada.
unsigned int hash(const char *word)
{
    // Retorna o valor (0-25) no qual a palavra deve ser armazenada, de
acordo com a primeira letra.
    return toupper(word[0])-'A';
}

// Carrega o dicionário na memória, retorna verdadeiro caso consiga e
falso se ocorrer algum erro.
bool load(const char *dictionary)
{
    // Abre o arquivo dicionário.
    FILE *input = fopen(dictionary, "r");

    // Caso não consiga abrir, retorna falso e uma mensagem de erro.
    if(input == NULL)
    {
        printf("Unable to open %s",dictionary);
        return false;
    }

    // Cria uma variável para armazenar a palavra lida.
    char palavra[LENGTH + 1];

    // Lê cada palavra do texto até que a função retorne EOF,
significando que chegou ao final do arquivo.
    while(fscanf(input, "%s", palavra) != EOF)
    {
        // Solicita espaço de memória para armazenar a palavra e o
pointer.
        node *n = malloc(sizeof(node));

        // Caso não tenha espaço suficiente, libera o espaço solicitado e
retorna falso.
        if(n == NULL)
        {
            unload();
            return false;
        }
```

```
        // Copia a palavra lida para a variável n.
        strcpy(n->word, palavra);

        // Chama a função hash para descobrir a onde a palavra deve ser
armazenada.
        valor_hash = hash(palavra);

        // Se for a primeira palavra armazenada nesse bucket, adiciona o
próximo pointer de n como nulo e adiciona o mesmo a n.
        if(table[valor_hash] == NULL)
        {
            n->next = NULL;
            table[valor_hash] = n;
        }
        // Caso contrario faz com que o proximo de n aponte para o locar
que a table esta apontando, e após faz a table apontar para onde n está
apontando.
        else
        {
            n->next = table[valor_hash];
            table[valor_hash] = n;
        }

        // Atualiza a variável de palavras contadas.
        palavra_count++;
    }
    // Fecha o arquivo e retorna que tudo ocorreu bem.
    fclose(input);
    return true;
}

// Retorna o tamanho do arquivo, ou seja, o número de palavras contadas.
unsigned int size(void)
{
    // Se a quantidade de palavras for maior que 0, escreve o mesmo.
    if(palavra_count > 0)
    {
        return palavra_count;
    }
    return 0;
}

// Libera a memória solicitada por meio do malloc.
bool unload(void)
{
    // Cria uma variável cursor para ajudar a não perder informações.
    node *cursor;
```

```c
    // Passa por todos os 26 "buckets"
    for(int i = 0; i < N; i++)
    {
        cursor = table[i];

        // Enquanto o cursor não for Nulo, continua repetindo.
        while(cursor != NULL)
        {
            // Cria uma variável node temporária, para ir liberando a
memória.
            node *temp = cursor;

            // O cursor vai para a próxima palavra.
            cursor = cursor->next;

            // Libera a memória.
            free(temp);
        }
    }

    // Se após tudo o cursor for = a nulo, retorna verdadeiro.
    if(cursor == NULL)
    {
        return true;
    }

    // Caso contrário retorna falso.
    return false;
}
```