```
#
# Makefile
#
# Computer Science 50
# Problem Set 5
#


# compiler to use
CC = clang

# flags to pass compiler
CFLAGS = -ggdb3 -O0 -Qunused-arguments -std=c11 -Wall -Werror

# name for executable
EXE = speller

# space-separated list of header files
HDRS = dictionary.h

# space-separated list of libraries, if any,
# each of which should be prefixed with -l
LIBS =

# space-separated list of source files
SRCS = speller.c dictionary.c

# automatically generated list of object files
OBJS = $(SRCS:.c=.o)


# default target
$(EXE): $(OBJS) $(HDRS) Makefile
    $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)

# dependencies
$(OBJS): $(HDRS) Makefile

# housekeeping
clean:
    rm -f core $(EXE) *.o

```

```c
/**
 * dictionary.c
 *
 * Computer Science 50
 * Problem Set 5
 *
 * Implements a dictionary's functionality.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <cs50.h>
#include "dictionary.h"
/**
 * USING Kernighan and Ritchie's hash function
 */
#define HASHSIZE 1000

typedef struct node{
    char word[LENGTH+1];
    struct node* next;
}node;
node* hashtable[HASHSIZE];
unsigned int count = 0;
/**
 * Returns true if word is in dictionary else false.
 */
bool check(const char* word)
{
    int len = strlen(word);

    char* data = malloc(sizeof(char)*(len+1));
    if (data == NULL)
    {
        printf("Out of a heap memory\n");
        return false;
    }
    //converting text data to upper case
    for(int i = 0; i < len; i++)
    {
       data[i] = toupper(word[i]);
    }
    data[len] = '\0';

    unsigned int key = hash(data);
```

```
49.     node* value = hashtable[key];
50.
51.     // check if word is in dictionary
52.     while (value != NULL)
53.     {
54.         if (strcmp (value->word, data) == 0)
55.         {
56.             free(data);
57.             return true;
58.         }
59.         value = value->next;
60.     }
61.     free(data);
62.     return false;
63. }
64.
65. /**
66.  * Loads dictionary into memory.  Returns true if successful else false.
67.  */
68. bool load(const char* dictionary)
69. {
70.     // open dictionary
71.     FILE* infile = fopen(dictionary, "r");
72.     if (infile == NULL)
73.     {
74.         printf("Could not open %s.\n", dictionary);
75.         return 1;
76.     }
77.
78.     while (feof(infile)==0)
79.     {
80.         // malloc for each new word
81.         node* newnode = malloc(sizeof(node));
82.
83.         if (newnode == NULL)
84.         {
85.             printf("Memory error\n");
86.             return false;
87.         }
88.
89.         fscanf(infile, "%s", newnode->word);
90.
91.         if(feof(infile)){
92.             free(newnode);
93.             break;
94.         }
95.
96.         int len = strlen(newnode->word);
```

```
 97.
 98.         //converting dictoinary data to upper case
 99.         for(int i=0;i<len;i++){
100.             newnode->word[i] = toupper(newnode->word[i]);
101.         }
102.         unsigned int key = hash(newnode->word);
103.
104.         // adding new node to head of bucket in hashtable
105.         newnode->next = hashtable[key];
106.         hashtable[key] = newnode;
107.         //counting number of nodes in each bucket
108.         count++;
109.
110.     }
111.
112.     // close dictionary
113.     fclose(infile);
114.     return true;
115.
116. }
117.
118. /**
119.  *
120.  * Returns number of words in dictionary if loaded else 0 if not yet loaded.
121.  */
122. unsigned int size(void)
123. {
124.     return count;
125. }
126.
127. /**
128.  * Unloads dictionary from memory.  Returns true if successful else false.
129.  */
130. bool unload(void)
131. {
132.     for(int i = 0; i < HASHSIZE; i++){
133.         node* curr = hashtable[i];
134.         node* prev = NULL;
135.
136.         while(curr != NULL){
137.             prev = curr;
138.             curr=curr->next;
139.             free(prev);
140.         }
141.
142.     }
143.
144.     return true;
```

```
145. }
146. /**
147.  * USING Kernighan and Ritchie's hash function
148.  */
149.  unsigned int hash (char* word){
150.     unsigned int hashval ;
151.     for (hashval = 0; *word != '\0'; word++){
152.       hashval = *word + 31 * hashval;
153.     }
154.     return hashval % HASHSIZE;
155. }
```

```h
/**
 * dictionary.h
 *
 * Computer Science 50
 * Problem Set 5
 *
 * Declares a dictionary's functionality.
 */

#ifndef DICTIONARY_H
#define DICTIONARY_H

#include <stdbool.h>

// maximum length for a word
// (e.g., pneumonoultramicroscopicsilicovolcanoconiosis)
#define LENGTH 45

/**
 * Returns true if word is in dictionary else false.
 */
bool check(const char* word);

/**
 * Loads dictionary into memory.  Returns true if successful else false.
 */
bool load(const char* dictionary);

/**
 * Returns number of words in dictionary if loaded else 0 if not yet loaded.
 */
unsigned int size(void);

/**
 * Unloads dictionary from memory.  Returns true if successful else false.
 */
bool unload(void);

unsigned int hash (char*);

#endif // DICTIONARY_H

```

```
0.  An artificial term referring to a lung disease caused by silica dust, sometimes cited as one of the longest words in the English language
1.  getrusage() returns resource usage measures for who, which can be one of the following:
       RUSAGE_SELF
              Return resource usage statistics for the calling process, which is the sum of resources used by all threads in the process.
       RUSAGE_CHILDREN
              Return  resource  usage  statistics  for  all children of the calling process that have terminated and been waited for.  These statistics will
              include the resources used by grandchildren, and further removed descendants, if all of the intervening descendants waited on their terminated
              children.
       RUSAGE_THREAD (since Linux 2.6.26)
              Return resource usage statistics for the calling thread.
2.  16 members as follows:
    struct rusage {
                struct timeval ru_utime; /* user CPU time used */
                struct timeval ru_stime; /* system CPU time used */
                long   ru_maxrss;        /* maximum resident set size */
                long   ru_ixrss;         /* integral shared memory size */
                long   ru_idrss;         /* integral unshared data size */
                long   ru_isrss;         /* integral unshared stack size */
                long   ru_minflt;        /* page reclaims (soft page faults) */
                long   ru_majflt;        /* page faults (hard page faults) */
                long   ru_nswap;         /* swaps */
                long   ru_inblock;       /* block input operations */
                long   ru_oublock;       /* block output operations */
                long   ru_msgsnd;        /* IPC messages sent */
                long   ru_msgrcv;        /* IPC messages received */
                long   ru_nsignals;      /* signals received */
                long   ru_nvcsw;         /* voluntary context switches */
                long   ru_nivcsw;        /* involuntary context switches */
            };
3.  As the getrusage() expects pointer to a structure as its argument.Also, paas by reference is more efficient than value
4.  1.main declares variables and an array for storing a word (word[LENGTH+1]) for spell-check.
    2.It checks each word in text, using fgetc() until the end of the file. fgetc() gets the next character from the file and advances to the next character in the stream.
    3.It checks if the characters are alphabetical and apostrophes and stores them in the array word.
    If the word is too long or has digits, it is ignored and next word is checked.
    4.After all characters are checked, it terminates current word and updates counter of words. check() returns true if word is in dictionary else false.
    misspelled stores the word, if check() returns false.

5.  fgetc() reads character-by-character from a stream and helps keep track of the number of chars read.
    fscanf detects end of string either by encountering \n or \0. In the given problem, words could be separated by , or other characters which cannot be detected
    by fsacnf.
6. As the values do not and should not be changed.They are simply to be read by other functions.

7. I am using a hashtable that has linked list mapped to each haskekey.The nodes have a char array word[LENGTH+1] and a node pointer.
```

word contains the dictonary word mapped to the hashkey linklist.Hash function used for mapping is Rithchie & Kernigan's

8. At first I was applying sorting to the hashtable linkist in the load function.This was taking more time as compared to an unsorted list.
9. I removed sorting code from the linklist and am simply adding each new word to the beginning of the list
10. Not any that I can find.

```c
/**
 * speller.c
 *
 * Computer Science 50
 * Problem Set 5
 *
 * Implements a spell-checker.
 */

#include <ctype.h>
#include <stdio.h>
#include <sys/resource.h>
#include <sys/time.h>

#include "dictionary.h"
#undef calculate
#undef getrusage

// default dictionary
#define DICTIONARY "dictionaries/large"

// prototype
double calculate(const struct rusage* b, const struct rusage* a);

int main(int argc, char* argv[])
{
    // check for correct number of args
    if (argc != 2 && argc != 3)
    {
        printf("Usage: speller [dictionary] text\n");
        return 1;
    }

    // structs for timing data
    struct rusage before, after;

    // benchmarks
    double time_load = 0.0, time_check = 0.0, time_size = 0.0, time_unload = 0.0;

    // determine dictionary to use
    char* dictionary = (argc == 3) ? argv[1] : DICTIONARY;

    // load dictionary
    getrusage(RUSAGE_SELF, &before);
    bool loaded = load(dictionary);
    getrusage(RUSAGE_SELF, &after);

    // abort if dictionary not loaded
```

```c
    if (!loaded)
    {
        printf("Could not load %s.\n", dictionary);
        return 1;
    }

    // calculate time to load dictionary
    time_load = calculate(&before, &after);

    // try to open text
    char* text = (argc == 3) ? argv[2] : argv[1];
    FILE* fp = fopen(text, "r");
    if (fp == NULL)
    {
        printf("Could not open %s.\n", text);
        unload();
        return 1;
    }

    // prepare to report misspellings
    printf("\nMISSPELLED WORDS\n\n");

    // prepare to spell-check
    int index = 0, misspellings = 0, words = 0;
    char word[LENGTH+1];

    // spell-check each word in text
    for (int c = fgetc(fp); c != EOF; c = fgetc(fp))
    {
        // 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 ((c = fgetc(fp)) != EOF && isalpha(c));

                // prepare for new word
                index = 0;
            }
        }

        // ignore words with numbers (like MS Word can)
```

```c
        else if (isdigit(c))
        {
            // consume remainder of alphanumeric string
            while ((c = fgetc(fp)) != EOF && isalnum(c));

            // prepare for new word
            index = 0;
        }

        // we must have found a whole word
        else if (index > 0)
        {
            // terminate current word
            word[index] = '\0';

            // update counter
            words++;

            // check word's spelling
            getrusage(RUSAGE_SELF, &before);
            bool misspelled = !check(word);
            getrusage(RUSAGE_SELF, &after);

            // update benchmark
            time_check += calculate(&before, &after);

            // print word if misspelled
            if (misspelled)
            {
                printf("%s\n", word);
                misspellings++;
            }

            // prepare for next word
            index = 0;
        }
    }

    // check whether there was an error
    if (ferror(fp))
    {
        fclose(fp);
        printf("Error reading %s.\n", text);
        unload();
        return 1;
    }

    // close text
```

```c
    fclose(fp);

    // determine dictionary's size
    getrusage(RUSAGE_SELF, &before);
    unsigned int n = size();
    getrusage(RUSAGE_SELF, &after);

    // calculate time to determine dictionary's size
    time_size = calculate(&before, &after);

    // unload dictionary
    getrusage(RUSAGE_SELF, &before);
    bool unloaded = unload();
    getrusage(RUSAGE_SELF, &after);

    // abort if dictionary not unloaded
    if (!unloaded)
    {
        printf("Could not unload %s.\n", dictionary);
        return 1;
    }

    // calculate time to unload dictionary
    time_unload = calculate(&before, &after);

    // report benchmarks
    printf("\nWORDS MISSPELLED:     %d\n", misspellings);
    printf("WORDS IN DICTIONARY:  %d\n", n);
    printf("WORDS IN TEXT:        %d\n", words);
    printf("TIME IN load:         %.2f\n", time_load);
    printf("TIME IN check:        %.2f\n", time_check);
    printf("TIME IN size:         %.2f\n", time_size);
    printf("TIME IN unload:       %.2f\n", time_unload);
    printf("TIME IN TOTAL:        %.2f\n\n",
     time_load + time_check + time_size + time_unload);

    // that's all folks
    return 0;
}

/**
 * Returns number of seconds between b and a.
 */
double calculate(const struct rusage* b, const struct rusage* a)
{
    if (b == NULL || a == NULL)
    {
        return 0.0;
```

```
193.     }
194.     else
195.     {
196.         return ((((a->ru_utime.tv_sec * 1000000 + a->ru_utime.tv_usec) -
197.                   (b->ru_utime.tv_sec * 1000000 + b->ru_utime.tv_usec)) +
198.                  ((a->ru_stime.tv_sec * 1000000 + a->ru_stime.tv_usec) -
199.                   (b->ru_stime.tv_sec * 1000000 + b->ru_stime.tv_usec)))
200.                  / 1000000.0);
201.     }
202. }
203.
```