**HASH TABLE IMPLEMENTATION OF A CONCORDANCE**

**/\* spellcheck .c ; Spell Check implementation \*/**

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include "hashMap.h"

#define DICT\_NAME "dictionary.txt"

/\*

the getWord function takes a FILE pointer and returns you a string which was

the next word in the in the file. words are defined (by this function) to be

characters or numbers separated by periods, spaces, or newlines.

when there are no more words in the input file this function will return NULL.

this function will malloc some memory for the char\* it returns. it is your job

to free this memory when you no longer need it.

\*/

char\* getWord(FILE \*file);

/\*

Load the contents of file into hashmap ht

\*/

void loadDictionary(FILE\* file, struct hashMap\* ht);

int main (int argc, const char \* argv[]) {

clock\_t timer;

struct hashMap\* hashTable = malloc( sizeof( struct hashMap ));

int tableSize = 1000;

timer = clock();

initMap(hashTable,tableSize);

FILE\* dictionary = fopen( DICT\_NAME, "r" );

loadDictionary(dictionary,hashTable);

timer = clock() - timer;

printf("Dictionary loaded in %f seconds\n", (float)timer / (float)CLOCKS\_PER\_SEC);

char\* word = (char\*)malloc(256\*sizeof(char));

int quit=0;

while(!quit){

printf("Enter a word: ");

scanf("%s",word);

/\*

... spell checker code starts here ...

\*/

/\* Only spelled correctly if in the dictionary \*/

if( containsKey( hashTable, word ) )

{

printf( "Word in dictionary\n" );

}

else

{

printf( "Word not in dictionary\n" );

}

/\* Don't remove this. It is used for grading\*/

if(strcmp(word,"quit")==0)

quit=!quit;

}

free(word);

return 0;

/\*

... spell checker code ends here ...

\*/

}

void loadDictionary(FILE\* file, struct hashMap\* ht)

{

/\* Written this\*/

char \*word = NULL;

/\* Loop through entries \*/

while( ( word = getWord( file ) ) )

{

/\* If already in hash table, throw an error for the user \*/

if( containsKey( ht, word ) )

{

printf( "Multiple occurrences of a single word detected in dictionary.\n" );

}

/\* Else insert with occurrence count of 1 \*/

else

{

insertMap( ht, word, 1 );

}

}

}

char\* getWord(FILE \*file)

{

int length = 0;

int maxLength = 16;

char character;

char\* word = (char\*)malloc(sizeof(char) \* maxLength);

assert(word != NULL);

while( (character = fgetc(file)) != EOF)

{

if((length+1) > maxLength)

{

maxLength \*= 2;

word = (char\*)realloc(word, maxLength);

}

if((character >= '0' && character <= '9') || /\*is a number\*/

(character >= 'A' && character <= 'Z') || /\*or an uppercase letter\*/

(character >= 'a' && character <= 'z') || /\*or a lowercase letter\*/

(character == 39)) /\*or is an apostrophy\*/

{

word[length] = character;

length++;

}

else if(length > 0)

break;

}

if(length == 0)

{

free(word);

return NULL;

}

word[length] = '\0';

return word;

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* hashMap.c ; Hash map implementation \*/**

#include <stdlib.h>

#include <assert.h>

#include <string.h>

#include "hashMap.h"

/\* Compare function for hash table entries. Safety Limit 45,

\* set at longest word

\*/

#define compare( a, b ) strncmp( ( a ), ( b ), 45 )

int stringHash1(char \* str)

{

int i;

int r = 0;

for (i = 0; str[i] != '\0'; i++)

r += str[i];

return r;

}

int stringHash2(char \* str)

{

int i;

int r = 0;

for (i = 0; str[i] != '\0'; i++)

r += (i+1) \* str[i]; /\*the difference between 1 and 2 is on this line\*/

return r;

}

void initMap (struct hashMap \* ht, int tableSize)

{

int index;

if(ht == NULL)

return;

ht->table = (hashLink\*\*)malloc(sizeof(hashLink\*) \* tableSize);

ht->tableSize = tableSize;

ht->count = 0;

for(index = 0; index < tableSize; index++)

ht->table[index] = NULL;

}

void freeMap (struct hashMap \* ht)

{

/\*written this\*/

int i;

hashLink \*ptr1 = NULL;

hashLink \*ptr2 = NULL;

/\* free data \*/

for( i = 0; i < ht->tableSize; ++i )

{

if( ht->table[ i ] != NULL )

{

ptr1 = &ht->table[ i ][ 0 ];

ptr2 = ptr1->next;

while( ptr1 != NULL )

{

ptr2 = ptr1->next;

free( ptr1->key );

free( ptr1 );

ptr1 = ptr2;

}

}

}

free( ht->table );

/\* Clear hash table data \*/

ht->tableSize = 0;

ht->count = 0;

}

/\* Desc: Helper function for finding the index of a key

\* Pre: ht is not null

\* Post: index of k returned

\*/

int findKeyIndex( struct hashMap \* ht, KeyType k )

{

assert( ht != NULL );

int key\_index;

/\* Run the specified has function \*/

if( HASHING\_FUNCTION == 1 )

{

key\_index = stringHash1( k );

}

else if( HASHING\_FUNCTION == 2 )

{

key\_index = stringHash2( k );

}

/\* Reduce index to valid table index \*/

key\_index %= ht->tableSize;

/\* Ensure the index is positive \*/

if( key\_index < 0 )

{

key\_index += ht->tableSize;

}

return( key\_index );

}

void insertMap (struct hashMap \* ht, KeyType k, ValueType v)

{

/\*written this\*/

assert( ht != NULL );

int idx = findKeyIndex( ht, k );

/\* Check if key is already in table \*/

if( containsKey( ht, k ) )

{

/\* Replace the hash link as it already exists \*/

ht->table[ idx ]->value = v;

}

else

{

/\* Not in table, so create a hashLink \*/

struct hashLink \*new\_lnk = malloc( sizeof( struct hashLink ) );

/\* Fill hashLink with data \*/

new\_lnk->value = v;

new\_lnk->key = k;

new\_lnk->next = ht->table[ idx ];

/\* Add the new link to the table \*/

ht->table[ idx ] = new\_lnk;

ht->count += 1;

}

}

ValueType\* atMap (struct hashMap \* ht, KeyType k)

{

/\*written this\*/

assert( ht != NULL );

struct hashLink \*ptr = ht->table[ findKeyIndex( ht, k ) ];

while( ptr != NULL )

{

if( compare( ptr->key, k ) == 0 )

{

return( &( ptr->value ) );

}

ptr = ptr->next;

}

/\* If no value, return null \*/

return NULL;

}

int containsKey (struct hashMap \* ht, KeyType k)

{

/\*written this\*/

assert( ht != NULL );

struct hashLink \*ptr = ht->table[ findKeyIndex( ht, k ) ];

while( ptr != NULL )

{

if( compare( ptr->key, k ) == 0 )

{

return( 1 );

}

ptr = ptr->next;

}

return( 0 );

}

void removeKey (struct hashMap \* ht, KeyType k)

{

/\*written this\*/

assert( ht != NULL );

struct hashLink \*ptr = ht->table[ findKeyIndex( ht, k ) ];

/\* Check parent link \*/

if( compare( ptr->key, k ) == 0 )

{

struct hashLink \*ptr\_tmp = ptr->next;

free( ptr->key );

free( ptr );

ht->table[ findKeyIndex( ht, k ) ] = ptr\_tmp;

ht->count -= 1;

return;

}

/\* Check child links \*/

while( ptr->next != NULL )

{

if( compare( ptr->next->key, k ) == 0 )

{

struct hashLink \*ptr\_tmp = ptr->next->next;

free( ptr->next->key );

free( ptr->next );

ptr->next = ptr\_tmp;

ht->count -= 1;

return;

}

ptr = ptr->next;

}

}

int size (struct hashMap \*ht)

{

/\*written this\*/

assert( ht != NULL );

return ht->count;

}

int capacity(struct hashMap \*ht)

{

/\*written this\*/

assert( ht != NULL );

return ht->tableSize;

}

int emptyBuckets(struct hashMap \*ht)

{

/\*written this\*/

assert( ht != NULL );

int count = 0;

int i=0;

for( i = 0; i < ht->tableSize; ++i )

{

if( ht->table[ i ] == NULL )

{

++count;

}

}

return( count );

}

float tableLoad(struct hashMap \*ht)

{

/\*written this\*/

assert( ht != NULL );

return ((float)ht->count / (float)ht->tableSize);

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* hashMap.h \*/**

# ifndef HashMap

# define HashMap

# define KeyType char \*

# define ValueType int

# define HASHING\_FUNCTION 1

struct hashLink {

KeyType key; /\*the key is what you use to look up a hashLink\*/

ValueType value; /\*the value stored with the hashLink, an int in our case\*/

struct hashLink \* next; /\*notice how these are like linked list nodes\*/

};

typedef struct hashLink hashLink;

struct hashMap {

hashLink \*\* table; /\*array of pointers to hashLinks\*/

int tableSize; /\*number of buckets in the table\*/

int count; /\*number of hashLinks in the table\*/

};

typedef struct hashMap hashMap;

/\*the first hashing function you can use\*/

int stringHash1(char \* str);

/\*the second hashing function you can use\*/

int stringHash2(char \* str);

/\*initialize the supplied hashMap struct\*/

void initMap (struct hashMap \* ht, int tableSize);

/\*

free all memory used for your hashMap, but do not free the pointer to ht. you

can see why this would not work by examining main(). the hashMap passed into

your functions was never malloc'ed, so it can't be free()'ed either.

\*/

void freeMap (struct hashMap \* ht);

/\*

insert the following values into a hashLink, you must create this hashLink but

only after you confirm that this key does not already exist in the table. you

cannot have two hashLinks for the word "taco".

if a hashLink already exists in the table for the key provided you should

replace that hashLink (really this only requires replacing the value v).

\*/

void insertMap (struct hashMap \* ht, KeyType k, ValueType v);

/\*

this returns a POINTER to the value stored in a hashLink specified by the key k.

if the user supplies the key "taco" you should find taco in the hashTable, then

return a pointer to the value member of the hashLink that represents taco. keep

in mind we are storing an int for value, so you don't use malloc or anything.

if the supplied key is not in the hashtable return NULL.

\*/

ValueType\* atMap (struct hashMap \* ht, KeyType k);

/\*

a simple yes/no if the key is in the hashtable. 0 is no, all other values are

yes.

\*/

int containsKey (struct hashMap \* ht, KeyType k);

/\*

find the hashlink for the supplied key and remove it, also freeing the memory

for that hashlink. it is not an error to be unable to find the hashlink, if it

cannot be found do nothing (or print a message) but do not use an assert which

will end your program.

\*/

void removeKey (struct hashMap \* ht, KeyType k);

/\*

returns the number of hashLinks in the table

\*/

int size (struct hashMap \*ht);

/\*

returns the number of buckets in the table

\*/

int capacity(struct hashMap \*ht);

/\*

returns the number of empty buckets in the table, these are buckets which have

no hashlinks hanging off of them.

\*/

int emptyBuckets(struct hashMap \*ht);

/\*

returns the ratio of: (number of hashlinks) / (number of buckets)

this value can range anywhere from zero (an empty table) to more then 1, which

would mean that there are more hashlinks then buckets (but remember hashlinks

are like linked list nodes so they can hang from each other)

\*/

float tableLoad(struct hashMap \*ht);

# endif

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* main.c \*/**

#include <assert.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "hashMap.h"

/\*

the getWord function takes a FILE pointer and returns you a string which was

the next word in the in the file. words are defined (by this function) to be

characters or numbers separated by periods, spaces, or newlines.

when there are no more words in the input file this function will return NULL.

this function will malloc some memory for the char\* it returns.

\*/

char\* getWord(FILE \*file);

int main (int argc, const char \* argv[]) {

const char\* filename;

struct hashMap hashTable;

int tableSize = 1000;

clock\_t timer;

char\* word;

FILE \*inputFile;

int i;

/\*

this part is using command line arguments, you can use them if you wish

but it is not required.

if you wish not to use command line arguments manually type in your

filename and path in the else case.

\*/

if(argc == 2)

filename = argv[1];

else

filename = "input.txt"; /\*specify your input text file here\*/

printf("opening file: %s\n", filename);

/\* open the input file \*/

inputFile = fopen("input.txt", "r");

if(inputFile != 0)

printf("Opened Successfully\n");

else printf("Failed to open\n");

timer = clock();

initMap(&hashTable, tableSize);

/\*

... concordance code starts here ...

\*/

/\* Loop through entries \*/

while( ( word = getWord( inputFile ) ) )

{

/\* If already in hash table, increment number of occurrences \*/

if( containsKey( &hashTable, word ) )

{

ValueType \*val = atMap( &hashTable, word );

assert( val != NULL );

( \*val )++;

/\* Not adding word to hash table, so free it here \*/

free( word );

}

/\* Else insert with occurrence count of 1 \*/

else

{

insertMap( &hashTable, word, 1 );

}

}

/\* Print hash map \*/

for( i = 0; i < hashTable.tableSize; ++i )

{

struct hashLink \*lnk = &hashTable.table[ i ][ 0 ];

while( lnk != NULL )

{

printf( "%s: %d\n", lnk->key,lnk->value );

lnk = lnk->next;

}

}

timer = clock() - timer;

printf("concordance ran in %f seconds\n", (float)timer / (float)CLOCKS\_PER\_SEC);

printf("closing file...\n");

fclose(inputFile);

freeMap( &hashTable );

return 0;

/\*

... concordance code ends here...

\*/

}

char\* getWord(FILE \*file)

{

int length = 0;

int maxLength = 16;

char character;

char\* word = (char\*)malloc(sizeof(char) \* maxLength);

assert(word != NULL);

while( (character = fgetc(file)) != EOF)

{

if((length+1) > maxLength)

{

maxLength \*= 2;

word = (char\*)realloc(word, maxLength);

}

if((character >= '0' && character <= '9') || /\*is a number\*/

(character >= 'A' && character <= 'Z') || /\*or an uppercase letter\*/

(character >= 'a' && character <= 'z') || /\*or a lowercase letter\*/

character == 39) /\*or is an apostrophy\*/

{

word[length] = character;

length++;

}

else if(length > 0)

break;

}

if(length == 0)

{

free(word);

return NULL;

}

word[length] = '\0';

return word;

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* input.txt; please use your keys as present in the dictionary\*/**

It was the best of times, It was the worst of

Times

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* dictionary. txt ; please add keys as required from a to z \*/**

a

aah

aahed

aahing

aahs

aardvark

aardvarks

aardwolf

baaed

baaing

baal

baalism

baalisms

baals

baas

baba

babas

babbitting

babble

babbled

babbler

babblers

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**/\* Makefile.txt \*/**

all: prog

hashMap.o: hashMap.c hashMap.h

gcc -g -Wall -std=c99 -c hashMap.c

main.o: main.c hashMap.h

gcc -g -Wall -std=c99 -c main.c

prog: hashMap.o main.o

gcc -g -Wall -std=c99 -o prog hashMap.o main.o

clean:

rm prog

rm hashMap.o

rm main.o

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

