Jumble

The Jumble program reads in a dictionary and stores the words into a hash table. The program then reads in characters from standard input and outputs all of the words that can be made from the jumbled letters.

Included Files

The following files are included:

ht.c	Hash Table module
ht.h	Hash Table header
wl.c	Word List module
wl.h	Word List header
jumble.c	Runner module
Makefile	A makefile is included to make building the program easier.

Module Organization

The assignment called for an implementation of that would handle data abstraction. In order to implement this, three modules were developed, ht.c, wl.c, and jumble.c.

ht.c

This is the hash table module. A more detailed discussion is below. The module's functions are exposed in the ht.h header file.

wl.c

This is the word list module. The module uses the hash table, but could theoretically use another data structure to store the data. A more detailed discussion is below. The module's functions are exposed in the wl.c header file.

jumble.c

This module handles the input/output and interaction with the user.

Hash Table Interface

The hash table was developed to be as generic as possible using a typedef of the stored element type in the header and requiring function pointers for operations on the data.

The initial size of the table size is defined in a `#define` statement. When the table reaches a load factor defined in another `#define` statement, the table is resized. In order to decrease the time taken during this operation, the hash value is stored with the key so that only a mod operation must be performed to determine the new bucket for the node.

Public Functions

```
extern HT ht_new(int hash_function_number);
```

This function builds and initializes the hash table. The hash function that is to be used is stored

as a function pointer for easy reuse.

```
extern ELEMTYPE* ht_get_key(HT ht, char *key);
```
This function returns a pointer to an ELEMTYPE given an existing key. If the key currently exists, it will point to the existing element. Otherwise, the key will be added to the table and a pointer to a pointer to NULL will be returned. The advantage is that the client can modify the object stored for the specified key directly, without multiple calls to the hash table. The disadvantage is that every time a key is searched for that does not exist, a new key is created, albeit with a NULL value.

```
extern void ht_free(HT ht, void (*free_elem)(const void *));
```
This function frees the hash table. In order to handle multiple data types, a function pointer to free each ELEMTYPE is passed. This is called on each key's value, so that there are no memory leaks.

```
extern void ht_print(HT ht, void (*print_function)(const void *));
```
This is used in the debugging process. In order to print generic elements, a function pointer is passed to the function that will print the element.

The following functions just return statistics about the hash table:

```
extern int ht_table_size (HT ht);
extern int ht_load_factor (HT ht);
extern int ht_num_empty_buckets (HT ht);
extern int ht_longest_list (HT ht);
extern float ht_avg_list_length (HT ht);
```

Private Functions

```
static void ht_resize(HT ht);
```
Resizes the hash table to twice its size.

```
static int pow_i(int x, int n);
```
Computes the value of x^n. Overflow is very easy with integer power, so there is no power function for integers. As overflow is acceptable in this implementation (and somewhat desirable) this is implemented. The code should not be used outside of this code though.

```
static unsigned easy_hash (char *word);
```
Computes the sum of the ASCII values in the word.

```
static unsigned better_hash (char *word);
```
Computes the sum of the ascii value of the letters times a constant raised to the position of the each letter. This can be expressed with the following formula:

$$hash(s) = \sum_{i=0}^{n-1} c^{(n-i-1)} * s_i$$

Comparison of Hash Functions

Expected Time for Lookups

The expected time for lookups for the better hash function. In the better hash, using the big sample dictionary, the average list length was ~1.3 and the longest list length was 7. For the easy hash, the longest list length was 774 with an average list length of ~116.7. Obviously the better hash function will have a much better expected time for lookups.

Worst-Case Time for Lookups

The worst case time for lookups is O(n) for both functions, if the keys all hash to the same value. This is highly unlikely for both functions, and especially unlikely as the table gets large for the better hash function.

Table Usage

This is not a very good usage of the table, especially as the table grows large. The hash values max out at around 3500 with the large sample dictionary and the minimum value being 97 (for 'a') for keys of lowercase ascii words. This could certainly be overcome with very large strings as keys, but this defeats the purpose of using strings as keys.

Word List Organization

Public Functions

```
extern WL wl_build(int function_number);
```

This function creates a new word list. It also calls ht_new, to create a new hash table to store added words.

```
extern boolean wl_free(WL wl);
```

This function frees all of the word list structs and calls the ht_free to free the hash table.

```
extern void wl_print_matches(WL wl, char *word);
```

This function prints all of the words that can be made up from the jumbled letters in *word.

```
extern boolean wl_add_word(WL wl, char *word);
```

This function adds a word to the word list.

The following functions return statistics about the word list:

```
extern int wl_total_words (WL wl);
extern int wl_table_size (WL wl);
extern int wl_num_empty_buckets (WL wl);
extern float wl_avg_list_length (WL wl);
extern int wl_load_factor (WL wl);
extern int wl_most_matches (WL wl);
```