

CMSC 212, Fall 2005
Project #2

Due 9/30/2005 midnight

Background

In this assignment, you will implement an API for a text oriented associative memory. In an associative memory, you can lookup values based on keyword they are associated with. For example, if you had an associative memory of state capitals, a lookup of “Maryland” would return “Annapolis”.

Your implementation will use a hash table to store keyword and value pairs where each of keywords and values are variable length character arrays. The hash table will be stored in a dynamically allocated array. If two values hash to the same location in the array, you will search for a free location in the following elements of the array (wrapping around to the 0th element if you reach the end of the array). In addition, the array should auto-resize (and re-hash) itself whenever less than 10% of the elements are free. When you re-size, you should pick the first prime number that is at least twice the size of the previous table (i.e. 11, 23, 47, 97, 197, 397, 797, 1597, 3203, 6421). You should use an initialized array in your program to represent the possible table sizes. The maximum size of your table should be 6,421 elements.

Functions To Implement

Here are the required public functions exported by this interface. In addition to these, you will likely write several additional helper functions. Likely helper functions include: a hash function, and a function to expand (and re-hash the table).

`assocTable *createTable(int size)`

This function creates a table that can hold at least the passed number of elements. The minimum array size is 11 elements, and you should pick the next biggest table size closest to passed size, if the passed size is not a “standard” table size. If no such size is available, return NULL.

`int addTableItem(assocTable *table, char *keyword, char *value)`

This function adds the passed keyword value to a table. The passed character arrays (keyword and value) need to be stored in dynamically allocated memory so that subsequent changes to their values will not alter your table. If a keyword is already in the table, the new value replaces the old one. Attempting to insert either a NULL key or value is an error. On success, this function should return 0, and -1 for errors.

`char *lookupTable(assocTable *table, char *keyword)`

Lookup the passed keyword in the passed table. Return the value associated with the keyword if it is stored in the table or NULL otherwise.

```
int deleteTableItem(assocTable *table, char *keyword)
```

Delete the keyword (and its associated value) from the table. If the keyword is not in the table, return a value of -1; otherwise return a value of 0. Don't forget to think about how deleting an item that collides with another item in the table should work.

```
void deleteTable(assocTable *table)
```

Delete the passed table and all of its stored keywords, values, and arrays.

Additional Information

The supplied file `assocMemory.h`, contains the prototypes of the function you will implement and the type definitions for the internal representation of this table. You may not modify (add or delete) anything from this file. In fact, *the only file you may modify is `assocMemory.c`*.

This program will make extensive use of `malloc`, `free`, and `calloc` to manage storage. It is critical that you properly free any space when it is no longer being used.

Start with simple tests cases to create and store a few items into a table and then look them up. Proceed onto test cases that will require expanding the table. Next, return to the smaller tests and try some calls to `deleteTableItem` and `deleteTable`. Finally, return the larger examples and try deleting items.

We have supplied several test cases (each one is a C main program that calls into the API you are implementing) to help you debug your code. However, these will not be the entire set of test cases we will use to grade your API. You should think about what other things might need testing, and write those test cases (though you will not be graded on the test cases themselves).

In principal you could use any hash function to implementation this project. However, for this assignment, use the following hash function (adapted from Aho, Sethi, Ulman *Compilers*, Second Edition, pp. 436):

```
int hashValue = 0;
foreach character of the string until null
    hashValue is bit shifted 4 bits to the left and add in the current character
    if hashValue has a non-zero bit in the 4 leftmost (highest) bits
        g = (hashValue & 0xf0000000);
        hashValue = hashValue ^ (g >> 24);
        hashValue = hashValue ^ g;
return hashValue mod the size of the table (# of elements)
```