

## CMSC 212 Project #4

Due 04/12/2007 8:00 PM

### Background

In this assignment, you will implement an API for a hash table that stores generic pointers (void \*). This assignment will also use function pointers to define the hash, compare, and print functions for the individual items in the table.

Your implementation will use a hash table to store hash values and generic pointers. The hash table will be stored in a dynamically allocated array. If two values hash to the same location in the array, you will use a linked list to store all of the items with the same hash value (called chained or open hashing). In addition, the array should auto-resize (and re-hash) itself as the table fills up. If you didn't re-size the hash table, it would soon get very slow to lookup items, since when many items end up at the same hash value you have to search each item in the linked list to find them.

When you re-size, you should double the size of the table (i.e. a 5 element table should double to 10 elements). The initial size of the table should be 5 elements. When the table is more than half full (i.e. when the third element is added to a table with 5 elements), it should be re-sized. When doing a re-size, all elements in the table need to be re-hashed to compute their correct location in the larger table.

### 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 function to expand (and re-hash the table). The public interface routines are defined in hashTable.h,

```
hashTable *createTable(hashFuncType hFunc, compareFuncType cFunc,  
    printFuncType pFunc, int size)
```

This function creates a table. The four parameters at the functions to compute the hash value, the function to compare two items, the function to print an item, and a size parameter that indicates how large the item to be stored is (the size must be greater than 0void). The prototypes for the functions passed to createTable are:

```
typedef int (*hashFuncType)(void *a, int size);
```

returns a hash code for between 0 and size-1

```
typedef int (*compareFuncType)(void *a, void *b);
```

compares a and b, and returns 0 if they are equal, -1 if a is bigger and 1 if b is bigger.

```
typedef void (*printFuncType)(void *a);
```

prints the item a to standard output and a newline after it.

The hash and compare functions must be defined (non-null), but the print function may be undefined (null parameter passed). On failure createTable should return NULL.

Examples of failures include malloc returning null or a user passing null pointers.

`int addTableItem(hashTable *table, void *item)`

This function adds the passed item to a table. It should use the hash function passed when the table was created to hash the passed item and store it at an appropriate location in the hash table. On success, this function should return 0, and -1 for errors. You should allocate memory to store the item (the size parameter passed when the table was created tells you how big the item is), and then copy the item into allocated memory (the standard C function `memcpy` can be used to copy arbitrary memory). If two identical items are added to the table (as determined by the compare function), they both should be stored in the table.

`int lookupTable(hashTable *table, void *item)`

Lookup the passed item in the passed table. Return a 0 if the item is in the table, and -1 if it is not found. Use the hash function associated with this table to find the correct location in the table, and the compare function to make sure it is the correct item.

`int deleteTableItem(hashTable *table, void *item)`

Delete the passed item from the table. If the item is not in the table, return a value of -1; otherwise return a value of 0. Use the hash function associated with this table to find the correct location in the table, and the compare function to make sure it is the correct item. If a duplicate item is in the table, only the first matching item should be deleted.

`int printTable (hashTable *table)`

Prints out all of the elements in the table. They can be printed in any order, but each element must be printed exactly once by calling the print function associated with the table. It should return the number of elements printed. If the print function is null, it should still return the number of elements in the table.

`void deleteTable(hashTable *table)`

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

## Additional Information

The supplied file `hashTable.h`, contains the prototypes of the function you will implement and the public type definitions for the internal representation of this table. You make not modify (add or delete) anything from this file. The internal representation of the data structures used in implementing the `hashTable` are to be put into `hashTableP.h`. We provide a skeleton version of `hashTableP.h`, but you will need to add to it. Put all of your code into the file `hashTable.c` which we supply.

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. Also don't forget to initialize any memory you allocate (or call `calloc` instead which zeros memory). `Realloc` won't work to expand the table since the hash values will change when the table size changes.

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).