EX1

Dynamic memory Allocation, Generic Programming, Function Pointers and Libraries

You may use dynamic memory allocation; however it is prohibited in this course to use arrays of dynamic size (VLA). In order to prevent you from using VLA you must compile your code with the flag –Wvla (this is addition to the –Wall flag used to ensure that compilation is without warnings). You also must compile with the –g flag to enable memory management inspection with valgrind.

Background: Hash Tables

Hash Tables are data structures that support these three operations:

- Adding an element
- Removing an element
- Searching for an element

All of the operations listed above are performed in an average time of O(1). In your data structures course you have encountered hash tables. It is acceptable and common to implement hash tables using an array. In this exercise the size of the array is dynamic, i.e. the size of the array can grow. Mapping of an object to the appropriate location in this array is accomplished using a hash function:

$$i=d*H(k,m)$$

We can assume that for each object there is a key – k, which identifies the object. The hash function receives the key and the original size of the table - m (array size), and returns a pointer to the location that this object is supposed to be stored in - i. this pointer is multiply by d, which is the ratio between the current size of the table to its original size, initially, d=1.

Handling Collisions

The size of an array is generally smaller than the number of possible keys given. Thus, more than one key can be mapped by a specific hash function to a single location in the array which forms the hash table. There are various ways to deal with this issue. In this exercise we will use two simple methods together: linked lists and dynamic expansion of the hash table.

a. **Linked Lists**

   In each element in the hash table there is a pointer to the head of a list. Any new object that maps to this element in the hash table, will be added to the end of this list.

b. **Table Duplication**

We will build an implementation in which no more than *t* elements can be mapped to the same entry in the hash table. When we need to add an element to an index *i* where there are already t elements, go to the first location after i where there is room. If no place was found before the next original location in the table, or the end of the table was reached, the entire table size will be cloned (doubling the size of the table) in the following manner:
All existed entries of the table 0-(n-1) will be mapped to the respective **even** entries in the new table (this is accomplished by multiplying the original index by 2). In the event that an additional expansion of this table is required, this process is repeated. In this manner the distance between entries of the original table are always a power of two, e.g. after one expansion the original entries are at location 0,2,4,6… after two expansions 0,4,8,12,16… after three expansions 0,8,16,24… etc.

Example:
Given a hash table of size 4, three objects were inserted named X, Y, Z. The first row of the table shown here is the table's indices.

0	1	2	3
Y		X	
Z			

Now a new element W is mapped by the hash function to location 0.

In this case we need to expand the table and copy existing values, so that it looks like this:

0	1	2	3	4	5	6	7
Y				X			
Z							

And after W is inserted (described below):

0	1	2	3	4	5	6	7
Y	W			X			
Z							

Interface

The interface for the hash table shall be defined in the file GenericHashTable.h. You shall implement a hash table that supports the following operations

- Adding an element
- Removing an element
- Searching for an object
- Doubling the Size of the table

In the interface file GenericHashTable.hthat you get from us, there are various types of declarations,however the declarations of the names and types are missing in the structs *Table* and *Object* (although there are function declarations that use these objects). You need to define the structures Table and Object in GenericHashTable.h . The struct Table represents the table described above. The structure Object is an element in the table.

Objects:

The table is designed for use with general objects. To do this one must declare a *struct* (called Object) which includes a void* pointer named key as the object's key. The *struct* named Object can include additional fields as necessary in your implementation.

Implement the “constructor” of this object that is defined in the header file (function CreateObject), you may implement other constructors as you wish

The function freeObjectis designed to release the object's memory.

Table Representation

You shall define a struct named Table which includes the relevant fields of the struct. In the header there is a constructor which returns a pointer to the table (CreateTable). This function you must implement. You may implement other constructors as you wish.

Mapping of elements to the table is done with a key as described above. Implementation of insertion to be done as described above.

Print Table

The function printTable will print out the table in the following format:
[idx] \t elm1 \t → \t elm2 \t → …
The left column contains a number in brackets which is the index, and the index points to the strings/integers that are pointed to by this index.
For example, the line:
[0] 20 --> 40 -->
Means that the numbers 20 and 40 are chained to entry 0 in the table.

Table Helper Functions

In the file GenericHashTable.hthere are three functions that are used by the hash table. They are - the hash function, key printing function and a function that checks equality between keys.

The data of each object can be either integer or string, in the case of a string, the key points to the block of memory that is to be released by the free call.

You should implement hash functions for integers and for strings. For integer the hash function computes $i = k \bmod n$ where k is the key and n is the number of cells in the original table. You should insert only positive numbers to the hash (>=0).
For strings, the hash function computes $i = k \bmod n$ where k is the ascii sum of the characters of the string (the key) and n is the number of cells in the original table.

Functions that compare the value of keys shall return 0 if the keys are not equal and a non-zero value of your choosing if the keys are equal.

For integers, the function printTableprints the value without spaces. For strings, the function printTableprints the string as is.

When searching a key, if the key is in the table, you should print the index in the table where it was found, say i, and the index of it in the list of entry number i.
For example, we search for the key 56 and it's the third element of the list in entry 17 of the table, the output should be:
17\t3

Error Handling

There are error scenarios for each function (e.g. freeing memory, return values etc). Please implement accordingly.

Programming Assignment

Implement in the file GenericHashTable.cthe interface described in the file GenericHashTable.

Testing

You should write a main tester to test your work. Use all the function in the interface.

The compilation and linkage must run without warnings or errors.

Miscellany

You may not use prepared data structures in C for the linked list. You must implement the linked list yourselves.
Use const for variables and parameters when appropriate.
You can add any functions or const to the header file but NOT change the existing prototypes and names in it.
When a function receives a parameter which is a pointer you must check if the pointer is not NULL.

The exercise will be checked for proper memory management.

Run Valgrind on the program to check for memory leaks before you submit.

Submission

Submit a file named ex1.tar which contains the following:

README – Documentation of the program and implementation GenericHashTable.c–

Implementation of GenericHashTable.h

GenericHashTable.h – header file with your changes included.

makefile – A makefile that supports this application

Makefile

Your makefile shall support the following:

1. Creation of an executable htable based on
2. Running:
   make clean - erases without any further intervention the above executable

You can add source files as you wish but you do NOT submit a main file.
Compilation and Linkage must complete without errors or warnings

BEST OF LUCK TO ALL OF YOU