EX1 – Generic Hashtable

Background

In this exercise you need to create the generic HashTables.

Let’s start with a little bit of background about HashTables.

HashTables are data structures that support these three operations:

1. Adding an element
2. Removing an element
3. Searching for an element

All of the operations listed above are performed in an average time of O(1) or O(t) where t is a constant value.

In many cases HashTable is implemented using an array, where each cell is either contains a single value or a pointer to a list of values.

Collision occurs where more than one value has to be placed in the same cell in the array. If more than one value is allowed for each index, then, both values are listed to the same index in the array. But note that in order to get performance of O(t), we need to limit the length of these lists.

Generic HashTable

In this exercise you will create a generic HashTable. The HashTable should be implemented using an array that at each index of the array, there should be a pointer to a linked list.

The size of the array is dynamic, i.e. it can double it size when there is not enough room to insert a new value.

Definitions

M - the original size of the HashTable’s array. This value is given as a parameter to the program.

N – the current size of the HashTable’s array. Initially, N=M.

D - the ratio between the current size of the table to its original size. For example, if M=4 and N=32, then D=8. Initially, D=1.

K – the key that is calculated based on the Object to be inserted. More on that later.

H(k,M) – is the hash function that is based on two parameters, the key k and the original size M.

Mapping an Object

Mapping of an object to the appropriate location in this array is accomplished using the following:

i=D*H(k,M)

The hash function receives the key and the **original** size of the table - M, and returns a pointer to the location that this object is supposed to be stored in the original table. This pointer is then multiplied by D to get the location in the current table.

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 HashTable array. 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:

Each element in the hash table holds a pointer to a 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

In order to get performance of O(t) for all operations, no more than t elements should be mapped to the same entry in the hash table. When an element has to be inserted to an index i in the array and there are already t elements in the pointed list, this element should be added to first location j after i where there is room but j cannot reach an index of the original size.

If no place was found before the next original location in the table, or the end of the table was reached, the table size will be doubled 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). 0->0, 1->2, 2->4, 3->6, etc.
- This process is repeated whenever an expansion of the table is required.
- 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… (distance is 2) after two expansions 0,4,8,12,16… (distance is 4) after three expansions 0,8,16,24… (distance is 8) etc.

Example:

Given a hash table with M=4, and t=2, three objects were inserted named X, Y, Z. The first row of the table shown here is the table's indices.

Green – indicated entries of the original table.

Yellow – indicated entries that created by expansion.

0	1	2	3
Y		X	
Z			

Now a new element W should be mapped to location 0. There are already 2 elements at location 0 and all entries at this stage are original entries. Therefore, the table must be doubled, and existing values are copied by the following rule: if i is the current index of an element, this element is copied to index i*2.

The new table looks like this:

0	1	2	3	4	5	6	7
Y				X			
Z							

And W is inserted to the next index which is not part of the original indices (described below):

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

Let's say that R was also inserted to the entry number 1 and now H must be inserted to entry 0. So after R was inserted and before H is inserted, the table looks like this:

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

Since both entries 0 and 1 are full, the table must be doubled and looks like this after H is inserted:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Y	H	W						X							
Z		R													

Interface

The interface for the hash table shall be defined in the file GenericHashTable.h.

This file contains two structures, Object that represent an element in the table and Table that represent the hast table itself. You should add members to those classes as needed for your implementation.

Struct Object already has a pointer to generic data (void*), in this exercise it can be either integer or string.

Struct Table already has a pointer to the array with is an array where each entry contains list of Objects (Object**)

You shall implement a hash table that supports the following operations

1. Adding an element
2. Removing an element
3. Searching for an object
4. Doubling the Size of the table

See GeneralHashTable.h for all the functions that you need to implement. You may add more functions that you need for your own implementation as you wish but you must implement all functions in the file. We will add our own main file that will call these functions. You should NOT change the prototypes of the functions in the file.

You will find two hah functions in the file:

intHashFun – hash function for integers

strHashFun – hash function for strings

you will also find two constant values, INT_TYPE and STR_TYPE.

You should implement the hash functions as described in the .h file and use the correct function based on the constant value. Note that table can include either elements of type integer or elements of type string, not both. The type of the elements in the table is determined during table creation in the function createTable.

Print Table:

The function printTable should print out the table to stdout in the following format:

[idx] \t elm1 \t --> \t elm2 \t ...

Where idx is the index to an entry in the table and elm are the strings or integers that appear in the list pointed by this entry.

For example, the line: [0]	20	-->	40	-->

means that the numbers 20 and 40 appear in the list that pointed by entry 0 in the table.

Note that there is only a tab between the index to the first element and between the elements to the arrow. The spaces in the format line exist only for readability.

General instructions:

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.

What you need to submit:

You should submit a tar file containing the following files:

README

GenericHashTable.h

GenericHashTable.c

You can add more src files if you want (not needed in this exercise)

You should NOT submit main file or Makefile.

In order to create your tar file, you must go to the same directory where your files are and type:

```
tar -cvf ex1_123456.tar README GenericHashTable.h GenericHashTable.c
```

Where 123456 is your id.

GOOD LUCK