Assignment 6

Hash Table. Hash Functions. Collision resolution.

A *table* is a collection of elements of the same kind, which are identified by *keys*. The elements stored in a hash table are also called *records*. For example, in our movie tree assignment, the records were movies that were sorted by title.

A *hash function h* transforms a key into a natural number called a *hash value*.

$$h : K \rightarrow H,$$

where *K* is the set of keys and *H* is a set of natural numbers. Function *h* is a many-to-one function. If two different keys, say *k1* and *k2* (*k1 != k2)* have the same hash value (*h(k1) = h(k2)*), then the two keys are said to *collide* and the corresponding records are called *synonyms*. Two restrictions are imposed on *f*:

- For any $k \in K$ the value should be obtained as fast as possible.
- It must minimize the number of collisions.

In this assignment, write a program to manage a hash table, using *collision resolution by chaining*, where the keys are movie titles. Your code should provide create, insert, find and delete operations on that table.

What the program needs to do
Display a menu with the following options:
1. Insert movie
2. Delete movie
3. Find movie
4. Print table contents

Insert movie:
This menu option should prompt the user for the Title and Year of the movie. For testing, you can use the movie data in the Assignment6movies.txt file from the movie tree assignments, or make up movie titles and years. We're not using all parameters for each movie, just the Title and Year. The movie should be inserted at the end of the chain for its key. You do not need to alphabetize the movie titles.

There is no additional output for the Insert option.

Delete movie:
This menu option should prompt the user for the Title, search for the movie in the hash table, and delete it if found. You will need to update the pointers in your hash

table, setting them to NULL where needed, or deleting the movie from the linked list or vector.

There is no additional output for the Delete option.

Find movie:
This menu option should prompt the user for a Title, search the hash table for the movie, and display the table index where the movie is found, Title and Year for the movie. If the movie is not found, print "not found".

Print table contents:
The output for Print should display all movies stored in the table in the order they are stored, i.e. the movies at hashTable[0] should be printed before the movies at hashTable[10]. The movies do not need to be in alphabetical order. For debugging purposes, you may want to print the index for each movie title. Just remember to remove that code before submitting to COG.

Chaining
You can use a vector or a linked list in your hash table to implement the collision resolution by chaining algorithm.

Hash Function
Use the *hashSum* function from the lecture notes and a table size of 10. If two movies have the same hash code, they should be stored in the order they are added for that hash code.

Implementation and book algorithms
The algorithms in your book are based on an implementation where the hash table is an array of empty hash table elements and contained in each element is a next and previous pointer to other elements in a linked list. For a hash table, *T*, to check if there are any records stored for a particular hash value, you check if *T[index].next = NULL*, where *T* is the name of the array and *index* is the hash value. When an element is added to the hash table, the *next* pointer is updated to point to the head of a linked list.

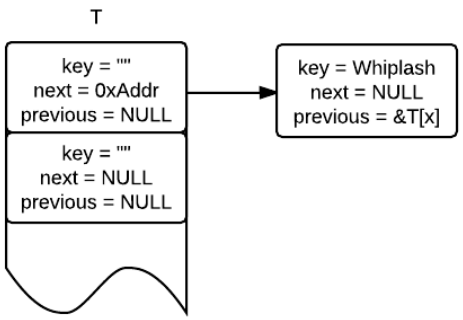

Figure 1. The hash table array contains empty hash table elements, each with a key, and next and previous pointers. Two elements of the array are shown here. The first element in the array points to a node in a linked list, where 0xAddr signifies the memory address.

The image in Figure 1 shows an example of a hash table using the empty records approach. When the first element is added to the chain for a particular index, the *next* pointer in the hash table is updated to

point to that new node and the previous pointer for the new node points to the hash table array at the given index. Adding additional nodes to the chain, and removing nodes from the chain, for an index would involve updating the next and previous pointers for the surrounding nodes.

I presented a slightly different approach in lecture where each element in the hash table array is a pointer to a linked list (Figure 2). In this implementation, the hash table array stores a pointer to a hash table element for a particular index. When there are no entries for an index, *T[x] = NULL*. The first node for an index is added to the chain by changing the *T[x]* pointer to point to the node.

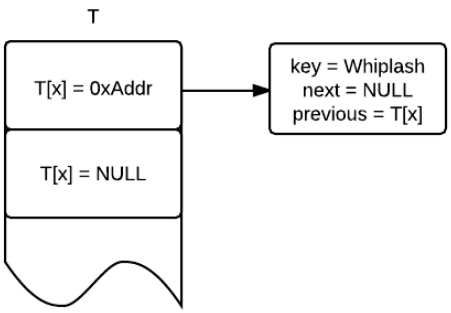

Figure 2. In this hash table example, the hash table is an array of pointers to the first node in a linked list.

How to submit your work

To submit your work, zip all files together and submit the zip file as Assignment 6 .zip. Your zip file should include HashTable.h, HashTable.cpp, and Assignment 6 .cpp. There is a sample HashTable.h on Moodle. Submit your work to COG. If you don't get the assignment working on COG, you will have an option for an interview grade.

Appendix A – cout statements that COG expects

Menu Display:

```
cout << "======Main Menu======" << endl;
cout << "1. Insert movie" << endl;
cout << "2. Delete movie" << endl;
cout << "3. Find movie" << endl;
cout << "4. Print table contents" << endl;
cout << "5. Quit" << endl;
```

1. Insert movie

   ```
   cout << "Enter title:" << endl;
   cout << "Enter year:" << endl;
   ```

2. Delete movie

   ```
   cout << "Enter title:" << endl;
   ```

3. Find movie

   ```
   cout << "Enter title:" << endl;
   ```

   Once the movie is found:

```
cout << index << ":" << title << ":" << year << endl;
```

If the movie is not found:

```
cout << "not found" << endl;
```

4. Print table contents.

```
cout<<temp->title<<":"<<temp->year<<endl;
```

if the hash table is empty:

```
cout << "empty" << endl;
```

5. Quit

```
cout << "Goodbye!" << endl;
```