CSE 109: Systems Programming

Fall 2017

Program 7: **Due on Tuesday, November 21st at 9pm on CourseSite.**

Collaboration Reminder:

1. You must submit your own work.
2. In particular, you may not:
   (a) Show your code to any of your classmates
   (b) Look at or copy anyone else's code
   (c) Copy material found on the internet
   (d) Work together on an assignment

Assignment: Preparation

1. Make a *Prog7* directory in your class folder.
2. You will be writing a *Hash.h*, *Hash.cpp* and *useHash.cpp* file for this assignment.

   This assignment is in C++, you must use a **class** and it's appropriate forms.
3. All source code files must have the comment block as shown at the end of this document. All files must be contained in your *Prog7* directory.

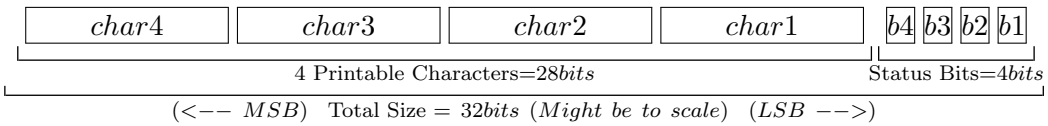
Background:

- The basic idea of a Chaining Hash Table is that we maintain a set of buckets (an array of buckets) and each bucket acts as an array of elements.

  When we store data, it goes into a bucket based on how the data is hashed by some hash function. A particular piece of data will always go to the same bucket. If we want to look for something, we hash it and look in the corresponding bucket.

- A hash function takes data and generates some non-reconstructable representation of it. For a given data, we will always generate the same result. This allows us to organize our data by what the resulting hash is.
- In this program, you will initially just store unsigned integers. Hash them, find out what bucket they go into: insert/remove/find/output.

  You should first work on getting this version of the Hash to work.
- However, we are going to allow the same Hash Table to be used to store printable characters. A nice thing about ASCII is that we only need 7 bits to store the usual characters (only positive values of a signed char). What this means for us is that within each unsigned int of the Hash, we can store 4 characters and have 4 bits leftover. We can use those 4 bits to indicate which character slots are empty!



The layout above shows how we can use an unsigned integer to store 4 printable chars. This means we can adapt our hash to work with this!

Compacting Chars:

This section only applies when dealing with storing characters within the Hash. Be aware that if the user tries to store both unsigned ints and characters, the results are undefined. However, you should not crash unless it can not be avoided.

- When looking through our Hash, we must look at each element as if it could be 4 characters.
- Use the status bits to determine whether or not a character exists.
- A status bit value of 1 indicates a particular character in the element exists, a 0 indicates it does not (see the diagram).
- You will need to "compress", if multiple empty spots exist, you should move characters from other locations into the empty spots so you can free up entire unsigned ints.

You don't have to "compress" until it is possible to remove the entire unsigned int from the Hash.

Alternatively, you can try to always compress.

Assignment:

In this assignment, you will be creating a chaining hash table data structure. The nature of the hash table is as follows:

- The elements stored in the hash can be of any either unsigned ints or chars. This is determined by usage. Your hash will not be aware of what data type it is handling. This means a user can try to insert an unsigned int and then a char, however, that will probably ruin the hash.
- The number of buckets in the hash is determined at construction time and can not be changed, it must be a const size_t value (this will be called *width*.
- The hash function will be provided at construction time as a function pointer. Otherwise, use a default hash function (described later).

  A pointer to a function? Oh dear.

1. Define a Hash_t object:
   (a) It will not know what type of objects it holds (unsigned int or char).
   (b) The number of buckets in the Hash_t must be made a public member of the class called *width*. It must be declared const. You will need to use an initializer list to give it a value.

   This must be done no matter how you handle the buckets even though it won't be needed in some cases.

   This const value will make it difficult to create an overloaded assignment operator. Therefore, you will not make one in this assignment.
   (c) Each bucket is a list of elements. You should use *vector* for this.
   (d) A function pointer that will be used to hash an object. The function pointer is for a function that takes *(void *, size_t)* and returns an int. If the constructor does not provide this, assign it to *basic_hash* function you will create later on.

(e) A size_t called *_size* to track the number of elements within the hash.

2. Constructors:

   (a) Default constructor: Assume bucket size is 10, uses default hash function.

   (b) Hash_t(size_t buckets): Use the bucket size given, use default hash function.

   (c) Hash_t(size_t buckets, int(*hashfunc)(void*, size_t): Use the bucket size given and use *hashfunc* as the function for hashing.

   This will be used to provide the basic_hash function to your Hash_t object.

   Initialize your buckets. Initialize *_size* to 0 - use this to keep track of how many elements are in the hash.

3. Copy constructor: Deep copies a Hash_t object.

4. bool insert(unsigned int): Inserts an unsigned int element into the hash. True if inserted. No duplicates are allowed. Insert the data at the end of the bucket (since you have to search the bucket for a possible duplicate first).

5. bool insert(char): Inserts an char element into the hash. True if inserted. No duplicates are allowed. Insert the data at the end of the bucket (since you have to search the bucket for a possible duplicate first).

6. bool find(unsigned int) const: Finds an unsigned int element in the hash. True if found.

7. bool find(char) const: Finds a char element in the hash. True if found.

8. bool remove(unsigned int): Removes an unsigned int element from the hash. True if removed.

   Only remove the first occurence.

9. bool remove(char): Removes a char element from the hash. True if removed.

   Only remove the first occurence.

10. string toString() const: Creates a string with all of the bucket data, see sample output. This assumes the hash holds unsigned ints

11. string toString(unsigned int) const: Same as toString().

12. string toString(char) const: Creates a string with all of the bucket data, see sample output. This assumes the hash holds chars.

13. size_t size() const: Returns the number of elements in the hash.

14. operator<<(): Overloaded output operator for the hash, uses toString().

15. private: getIndex(unsigned int) const: Calls the hash function for the hash object and mods it by the number of buckets.

16. private: getIndex(char) const: Calls the hash function for the hash object and mods it by the number of buckets.

17. You will probably want to make private methods that can disassmble and reassembled the unsigned ints into chars.

Default Hash:

```
1 int defaultHash(void *data, size_t size)
2 {
3     int bytes = 0;
4     for(size_t i=0; i<size; i++)
5     {
6         bytes += *(((char *)data) + i);
7     }
8     return bytes;
9 }
```

basic_hash:

The function basic_hash will take *(void *, size_t)* and return an *int*. The returned *int* will represent the SHA1 hash of the data pointed to by the *void * * that is of size specified by the *size_t*.

1. Use *man 3 EVP_DigestInit* for details on using the built-in message digest functionality.

2. You will need to add *-lssl -lcrypto* to your compilation line in order to link in the SSL and Crypto libraries.

3. The code given in the man page works if you add an include for *string.h*. You can rip code straight from that page, instead of *argv[1]* you would use *"sha1"*.

4. The resulting SHA1 hash (digest) is not an integer (it is an array of something). But you need to return an integer. Initialize the return value to 0, treat the digest as an integer array and *exclusive or* each integer in that array with your return value. Return the resulting integer.

Sample output for operator<<

Note the space before the single-digit indices. The indices should be lined up neatly based on the width of the largest index value. You do not need to line up the data elements within the buckets.

```
 0: (Empty)
 1: (Empty)
 2: (Empty)
 3: (Empty)
 4: (Empty)
 5: (Empty)
 6: (Empty)
 7: 18
 8: 71 19
 9: 33 20
10: 34
11: (Empty)
12: 23
```

```
 0: Z L B g
 1: a O W U F
 2: f S e A X
 3: Q b
 4: ‘ K ^
 5: N P d G
 6: H I D V
 7: [ ] M
 8: T C h R J E _
 9: \ c Y
```

Submission:

1. Once ready to submit, you can package up the assignment as a .tgz file

   tar -czvf Prog7.tgz Prog7

   You must use this command in the directory that contains the *Prog7* folder, not within the directory.

2. Transfer *Prog7.tgz* to the Program 7 submission area of CourseSite.

Comment Block:

```
/*
    CSE 109: Fall 2017
    <Your Name>
    <Your user id (Email ID)>
    <Program Description>
    Program #7
*/
```