CS221 Lab17 – Hash Table

Paul Haskell

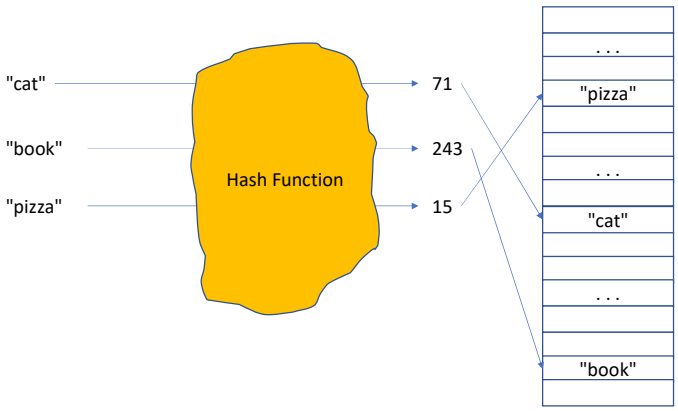
INTRODUCTION

A **Hash Function** is a function that takes the values of some data object and maps them to a number. For example, a simple (and bad) hash function for strings is:

```
unsigned simpleHash(const char* input) {
      unsigned retval = 0;
      while (*input != 0) {
            retval += *input++; // Add up ASCII values
      }
      return retval;
}
```

A Hash Function can operate on a `struct` by processing all the members of the `struct`.

A **Hash Table** is a table—or array—that "indexes" (stores and retrieves) data objects based on their hash-function value.



Frequently, pairs of objects are stored in a Hash Table, and the hash function is only computed on one of the objects, the "key". This makes a HashTable act like a Dictionary, where we look up the "value" by the "key".

Suppose we want to store a large number of strings in some data structure so we can retrieve them quickly.

- We could use an array, but either we would have to sort the values every time we add a new one (which is slow), or would have to search the entire array every time we search (which is slow).
- We could use a Linked List, but we have the same slowness problems.

But with the Hash Table, we can look up a value (to see if it occurs in the table or not) as quickly as we can compute the Hash Function on the value!

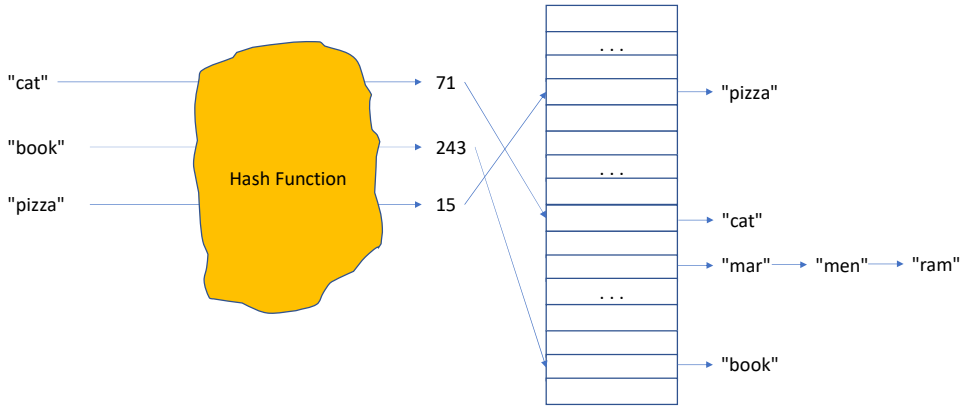
Really?? Well, that's only true if the size of our Hash Table is much larger than the amount of data we want to store. Otherwise, multiple **different input values will have the same Hash Function value**.

In fact, using a Hash Table is fast only if we have a good Hash Function also. The `simpleHash()` function is not very "good": lots of different words map to the same Hash Function value:

"mar", "men", "ram", "sal", "safe", "rage"…

A good Hash Function should give different values for different inputs, even if only the order of the values in the input differ. The design of "good" Hash Functions takes a lot of sophisticated math, and we won't cover it in CS221.

No Hash Function is perfect, and a Hash Table must handle the problem of different inputs mapping to the same Hash Function value, and thus to the same Hash Table entry. An easy way to do this is to store a Linked List of values in each Hash Table entry:



Of course, with a Linked List at every Hash Table entry, we need a little more time to search the table to see if a value is stored. What Hash Tables really let us do is to **trade off memory size and speed**: more memory gives more speed, less memory gives less (but still pretty good) speed. Often a Hash Table is the most efficient way to store and retrieve values.

Part 1

For Part 1 of this lab, you will build a Hash Table of strings. See the Appendix of this PDF for a Hash Function that maps a string to an `unsigned` value.

There are over 4 billion possible unsigned values, and we do not want 4 billion entries in our Hash Table. We will only have 256 entries. So each entry stores a range of Hash Function values.

Each entry in the Hash Table shall store three values:

- `lowest` value of the Hash Function for this Table entry
- `highest` value of the Hash Function for this Table entry
- Linked List of strings whose Hash Function value is between `lowest` and `highest`

You must write the Linked List software.

Provide a method to initialize the Hash Table, setting each entry's `lowest` and `highest` values to evenly cover the range of possible `unsigned` values.

Provide an `add(const char* word)` method that adds `word` to the correct entry in your Hash Table. If the same word is added more than once, the LinkedList should keep track of the "count" of how many times the word was added.

Provide a `find(const char* word)` method that returns the word's "count" or 0, depending on whether or not `word` is found in your Hash Table.

Please write a program **TestHash.c** that creates a Hash Table. The program should read the name of a text file from the command line, and then should read each word from the text file, adding each word to your Hash Table.

The program then should prompt the user for words to search for. For each user-entered word, print the "count" or 0. The program should exit cleanly when the user ends the keyboard input by typing ctrl-D (Mac) or ctrl-Z (Windows).

Part 2 - OPTIONAL

The time needed to add or search for a value in a Hash Table will get slow if the inputs being stored all map to the same entry in the Hash Table. This can happen if different values map to the same Hash Table entry.

For Part 2 of this project, every time a new value is added to the Hash Table, **rebalance** the Hash Table.

What's rebalancing?

Rebalancing is simply changing the Hash Table so the longest Linked List in the Hash Table is made shorter, and the shortest Linked List is made longer. But we want to keep the size of the Hash Table array to be 256. How can we do this?

- Find the two adjacent Hash Table entries whose total list lengths are shortest (among all adjacent Hash Table entries). Merge those entries into one. This requires you manipulate the `lowest` and `highest` values and also the Linked Lists of those two entries, to make one merged entry.
- Find the Hash Table entry whose Linked List is longest, and split it in half. To keep things simple, just divide the entry's `lowest`-to-`highest` range in half. This also requires proper setting of the new entries' `lowest` and `highest` values, and also their Linked Lists.

Wow—it took me a few hours of testing and debugging to get this right. Good luck!

CONCLUSION:

Please push your program to GitHub before the project deadline. The deadline for completion of Lab 17 is 11:59pm Monday April 7.

I have no idea how difficult this project will be, so please see me or the TA's if you get stuck.

Task	Score, points

Part 1's **TestHash.c** compiles and runs properly	25
Software quality of Part 1	15
Part 2's **TestHash.c** rebalances properly, and continues to add and search for strings properly	Just bragging rights

APPENDIX

```
// A hash function that maps strings to their hash values.
// Use the MPEG-2 CRC32 operation. As Paul about MPEG :)
// See https://stackoverflow.com/
//      questions/54339800/how-to-modify-crc-32-to-crc-32-mpeg-2
unsigned hash(const char* val) {
    unsigned hashVal = 0xffffffff;
    while (*val != 0) {
        hashVal ^= ((const unsigned) *val) << 24;
        for (int j = 0; j < 8; j++) { // each bit in this char
            unsigned msb = hashVal >> 31;
            hashVal <<= 1;
            hashVal ^= (0 - msb) & 0x04C11DB7;
        }
        val++; // each char in the string
    }
    return hashVal;
}
```