# MPCS 51100 - Fall 2018
# PSet 4
# Hashmaps

**Due Date: Nov 26, 2018 @ 5:30pm on Canvas**

## Introduction

- In this assignment, you will implement a variety of short programs in C based on hashing.
- Begin by cloning the github repository at `https://github.com/jtramm/51100_pset4`. The repository contains only a few test dictionary files that you will use, as well as empty files for you to fill with your code (no framework code is given). When you submit your code, you must maintain the same directory structure and filenames (e.g., p1.c, p4.c) so that your code can be compiled and run by the grader.
- In your top directory, populate the README.txt file to list your name, the assignment, and a discussion of any shortcomings your code may have. You will receive more partial credit if you are able to clearly identify faults or problems with your code, rather than letting us find them ourselves.
- Your code should compile with the included makefile. That is, using gcc and with the following flags: `gcc -Wall -O2`.
- If you are unable to implement the program correctly or fully, you may still receive partial credit, so turn in what you have done, and specify in the README where things may be problematic. If you have questions regarding expectations on the homework, please post your questions to Slack.
- Due date and extension policies are as prescribed by the course syllabus.
- The academic integrity policy for this assignment is as described in the course syllabus.

## 1 Hash Dictionary - p1.c (20 points)

For this problem, you will implement a dictionary with the same functionality described in Problem Set 2, Problem 4. However, in this problem, you will use a hashmap rather than a binary search tree as the underlying data structure. Some requirements and items to consider are:

- You must implement a *chained hashtable* using a linked this within each bucket (bin).
- Your hashtable will start as an array of 16 bins and must support dynamic resizing - specifically, doubling the number of available hash bins and rehashing when the occupancy (load factor) becomes too high. The occupancy is defined as the ratio of the number of total entries being stored in your hashtable to the total number of bins. Resize dynamically to keep the occupancy below 0.75 at all times.
- As hashtables do not preserve key order, you will need to think about how to implement the dictionary "print" functionality to display all words and definitions in alphabetical order. It is understood that your solution for printing will not be efficient, but it is expected to work correctly.
- When rehashing, deleting words, or clearing the dictionary, you must ensure that any dynamically allocated data is properly freed so that there are no memory leaks.

Implement the following three hashing functions as different options in your program:

```c
unsigned int naive_hash(char * word, int nbins)
{
    unsigned int h = 0;
    int c;
    while(c = *word++)
        h += c;
    return h % nbins;
}

unsigned int bernstein_hash(char * word, int nbins)
{
    unsigned int h = 5381;
    int c;

    while(c = *word++)
        h = 33 * h + c;

    return h % nbins;
}

unsigned int FNV_hash(char * word, int nbins)
{
    unsigned long h = 14695981039346656037lu;
    char c;
    while(c = *word++)
    {
        h = h * 1099511628211lu;
        h = h ^ c;
    }

    return h % nbins;
}
```

Enable the user to select a hashing algorithm via command line argument when the program is launched, using the following interface:

```
# use naive hash algorithm
$> ./p1 1
# use Bernstein's hash algorithm
$> ./p1 2
# use the FNV hash algorithm
$> ./p1 3
```

You should not have three copies of all your dictionary functions (`add_word(), delete_word()`, etc) for each of the hashing methods. Rather, you should simply store the desired hash function as a function pointer in your dictionary structure when initializing it, and have all your work functions that need to hash something call that function pointer.

You have been given a framework in p1.c to build off of, though you are free to change the code, add any functions, alter interfaces, modify structs, change printing techniques, or start totally fresh, as long as you provide the functionality described in the problem specifications. **Note that problems 2 and 3 will build on your Problem 1 code, rather than having separate files, so you may wish to read ahead before starting on this problem so as to plan out interfaces etc.**

## 2 Hash Dictionary Performance - p1.c (10 points)

Add into your p1.c code an extra command line operation "stats" that prints out several statistics relating to the current state of your dictionary hash table data structure. This function does not need to be efficient, so a full scan of all bins and entries in the table is acceptable. This operation should compute and print the following statistics:

- Number of possible hash values (bins) currently allowable in the hash table.
- Occupancy (load factor) of the hash table. Occupancy $= \frac{\text{Number of entries being stored by hashtable}}{\text{Number of bins allowable in hashtable}}$
- Fraction of bins that have one or more elements stored there $= \frac{\text{Number of bins being used in hashtable}}{\text{Number of bins allowable in hashtable}}$
- Maximum number of elements (words) currently stored in any single bin.

Feel free to print out the statistics in any format, as long as all the values are given and labeled clearly so it's easy for a user how to interpret the results and assess the health of the datastructure. An example is given below:

```
>$ ./p5 3
>$ import small_dictionary.txt
The file "small_dictionary.txt" has been imported successfully.
>$ stats
Bins: ___  occupancy: ____  used bin fraction: _____  max entries in a bin: ____
```

Use your `stats` command to quantify how well each of the three algorithms perform compared to the naive algorithm. Test all algorithms with the "small_dictionary.txt," "medium_dictionary.txt," and "large_dictionary.txt" files and report your results (in terms of occupancy, fraction of used bins, and maximum number of items per bin) in table format. Include your table in your submission as "p2.pdf". Also include in your "p2.pdf" a brief written explanation for any observed performance deficiencies of any of the three hashing methods tested you have tested so far. Why is the naive hashing method a good or bad hash algorithm given key values composed of English language words? It is not enough to say that the max number of elements in a bin is low or high etc, you need to examine the naive hashing function itself and explain why the performance is good or bad for this specific use case.

## 3 Universal Hashing - p1.c (20 points)

In this problem, you will add another hashing technique to your dictionary code. Instead of implementing another single hashing function, you will implement a universal family of hash functions. As discussed in class, universal hashing can be very useful in an adversarial environment where someone might feed your hash table a pathological dataset so as to reduce its performance. For instance, someone who knows you are using a specific hash function on a set table size might construct a pathological set of keys that they know will all hash to the same value, so that the hash table only fills a single bin and lookup performance decays from $\mathcal{O}(1)$ to $\mathcal{O}(n)$ time. Universal hashing can also be useful if you want to try to optimize a hashtable to improve its performance characteristics.

In this problem, each hash function $h_a$ in the family $H$ will be defined by a randomized vector $a$. The family of hash functions is defined as:

$$h_a(k) = \left[\sum_{i=0}^{r} a_i k_i\right] \bmod s \tag{1}$$

where:

- $r$ is the maximum length of the input key vector (in our case, the maximum length of a string)
- $s$ is the size of hashtable (i.e., number of bins), which must be a prime number, and must be larger than $r$
- $a$ is a vector of length $r$ made up of random integer values in the range such that $0 <= a_i < s$. This randomized vector is what defines each specific hash function in the family.
- $k$ is the key vector, of length $r$, with each entry containing an integer value $k_i < s$. In our case, if a string is of length less than $r$, fill the extra entries with zeros.
- $h_a(k)$ is a specific hash function, defined by the random vector $a$, which hashes an input key vector $k$.

To help demonstrate the idea behind this universal hashing scheme, consider a specific use case of hashing string keys for our dictionary program. If we set our maximum string length is 8, then $r = 8$, and $s$ to the next highest prime number above 8, which is 11. We then begin by defining our first hash function, by creating an $a$ vector of length $r$ filled with random integers $0 <= k_i < s$:

$$a =< 9, 1, 10, 8, 2, 5, 4, 3 >$$

With $a$ selected for our first hash function, we can receive and hash our first key, "hello". The key "hello" can be converted to integer format (by casting each `char` to an `int` type), resulting in a vector:

$$k =< 104, 101, 108, 108, 111, 0, 0, 0 >$$

However, this violates our condition that each $k_i < s$, so we need to modulus our $k$ vector by $s$, such that:

$$k =< 5, 2, 9, 9, 1, 0, 0, 0 >$$

We can then run our hash function as follows:

$$h_a(k) = [9 \times 5 + 1 \times 2 + 10 \times 9 + 8 \times 9 + 2 \times 1 + 0 + 0 + 0 + 0] \bmod 11$$
$$h_a(k) = 211 \bmod 11$$
$$h_a(k) = 2$$

We can then begin to hash and add more entries to our hashtable as normal. As we are now concerned about an adversary attacking our table with pathological keys, we monitor the performance of our table after each addition to check to make sure keys are being distributed, and changing the hash function and rehashing if too many keys are ending up in the same bin.

Continuing with our present example, we then hash several other incoming key words, say "fast", "delta", "commute", and "file", and add them to our table. We find that these words all also hash to a value of 2, so that bin has begun to fill up with 5 entries while the other bins are empty, and we can see our table is beginning to approach $\mathcal{O}(n)$ lookup time. We surmise that we are under attack, and decide to rehash. We generate a new random vector $a$:

$$a =< 6, 6, 10, 8, 3, 4, 0, 9 >$$

and rehash all of our entries. We now find that all of these five words hash to different values, meaning our table has returned to $\mathcal{O}(1)$ lookup time. We then continue using this new hash function as we add new values until we detect poor hash distribution or need to resize, at which time we can generate a new randomized vector $a$.

It is important to note is that when our occupancy becomes high, and we need to increase the size of the table and rehash, we must pick both a new vector $a$ as well as a new size $s$. As $s$ must be a prime number, we should double the previous $s$, then run a test on subsequent numbers after it to check if they're prime. For your implementation, you will need to write a function to scan and test for the next prime number. It does not need to be an efficient solution, but it should work.

Implement this universal hashing scheme as another hashing option in your dictionary code in p1.c. Each time a new word is added, your code should test if the distribution of keys is bad and rehash with a new function if necessary. While there are a variety of ways to determine what counts as a "bad" distribution, for the purposes of this problem we will define this as being when the following two conditions both hold: 1) more than 25% of entries are stored in a single bin, and 2) there are more than 10 entries in the hashtable. The "bad distribution" check must be accomplished efficiently in time, so as to still achieve $\mathcal{O}(1)$ "add word" operations (that is, you should not scan all entries in the dictionary after each add operation).

Building on the interface from problem 1, allow your universal hash function to be selected with the following interface:

```
# use naive hash algorithm
$> ./p1 1
# use Bernstein's hash algorithm
$> ./p1 2
# use the FNV hash algorithm
$> ./p1 3
# use the stochastic universal hash algorithm
$> ./p1 4
```

Make sure it also provides `stats` command functionality as described in problem 2 (which does not need to be efficient).

## 4 Universal Hashing Without "a" - p4.c (10 points)

Copy your solution to problem 3 into p4.c. Then, alter it so that it provides the same functionality but without having to explicitly store the $a$ vector. This problem may require some creative thinking and perhaps some light research into how random numbers are generated on computers.

## 5 Optimized Universal Hashing - p5.c (10 points)

Copy your solution to problem 3 into p5.c. Then, alter it to add a new user command `optimize`, that works to attempt to optimize the hash function for the current dictionary so as to create the fastest possible lookups. For the purposes of this problem, we will optimize by trying to find an $a$ vector that results in a table with the lowest maximum number of entries in any bin. Your code should perform this optimization in a stochastic manner, testing many different $a$ vectors and selecting the vector that results in the best performance. You should not change the size (number of bins) of the hash table, you should only change the $a$ vector when optimizing. An example of a run is as follows:

```
>$ ./p5 4
>$ import small_dictionary.txt
The file "small_dictionary.txt" has been imported successfully.
>$ stats
Bins: 163  occupancy: 0.613  used bin fraction: 0.460  max entries in a bin: 3
>$ optimize
>$ stats
Bins: 163  occupancy: 0.613  used bin fraction: 0.491  max entries in a bin: 2
>$ quit
```

Your code should try at least 100 different hash functions for each call to `optimize`. In the event that you are unable to find a more optimal vector than your starting vector, you may want to adjust your random number seed so that it changes each run by initializing it with the system time via a call to `time(NULL)`;

## 6 Hash Filter - p6.c (20 points)

Write a function that uses hashing to efficiently filter out duplicates from an array of objects. The function should be written generically so as to work for an array of any datatype, provided user supplied comparator and hash functions. Your function should have the following prototype:

```c
int uniq(void * f, int n, int sz, int (*equals)(void  *, void *),
         unsigned int (*h)(const void *key));
```

where:

- `f` is a pointer to the original array of objects
- `n` is the number of objects in the original array
- `sz` is the size (in bytes) of each object
- `equals` is a function pointer to the user supplied function that returns non-zero when two objects are equal, and zero when they are not
- `h` is a function pointer to the user supplied hash function
- the return `int` indicates the number of objects in the filtered array

Your function should run in $\mathcal{O}(n)$ time assuming a reasonably efficient hashing function is used.
Test your function on the following two data types:

```c
typedef struct{
    double x;
    double y;
    double z;
    double v_x;
    double v_y;
    double v_z;
    int state;
} Particle;

typedef struct{
    int age;
    char name[64];
    double gpa;
} Student;
```

Write a program in p6.c that initializes an array of `Particle` objects and an array of `Student` objects, fills them with a mixture of unique and duplicate values, and then prints the results before and after your `uniq()` function is called on them. You will need to develop hashing functions and comparator functions to go along with our two test types. You are welcome to adapt any (reasonably efficient) hashing function you like for these purposes.

For your `Student` specific functions, assume that the name is stored as a regular C style null terminated string. Only hash and compare the used portions of the name array, i.e., do not hash or compare character values after the null terminator.

An example run is given below (you do not need to use these values/names, or format, but your code's output should be clear so that it's easy for the grader to confirm that it's working correctly).

```
>$ ./p6
Original List:
Position: (0.20, 0.98, 0.67) Velocity: (0.73, 0.10, 0.08) State: 1
Position: (0.52, 0.53, 0.10) Velocity: (0.48, 0.06, 0.13) State: 1
Position: (0.87, 0.28, 0.36) Velocity: (0.75, 0.68, 0.94) State: 0
Position: (0.06, 0.57, 0.32) Velocity: (0.93, 0.01, 0.68) State: 2
Position: (0.26, 0.32, 0.18) Velocity: (0.41, 0.48, 0.49) State: 0
Position: (0.19, 0.91, 0.56) Velocity: (0.90, 0.53, 0.22) State: 0
Position: (0.26, 0.32, 0.18) Velocity: (0.41, 0.48, 0.49) State: 0
Position: (0.46, 0.77, 0.18) Velocity: (0.16, 0.93, 0.11) State: 2
Position: (0.29, 0.45, 0.17) Velocity: (0.05, 0.01, 0.59) State: 4
Position: (0.46, 0.77, 0.18) Velocity: (0.16, 0.93, 0.11) State: 2
Unique'd List:
Position: (0.20, 0.98, 0.67) Velocity: (0.73, 0.10, 0.08) State: 1
Position: (0.52, 0.53, 0.10) Velocity: (0.48, 0.06, 0.13) State: 1
Position: (0.87, 0.28, 0.36) Velocity: (0.75, 0.68, 0.94) State: 0
Position: (0.06, 0.57, 0.32) Velocity: (0.93, 0.01, 0.68) State: 2
Position: (0.26, 0.32, 0.18) Velocity: (0.41, 0.48, 0.49) State: 0
Position: (0.19, 0.91, 0.56) Velocity: (0.90, 0.53, 0.22) State: 0
Position: (0.46, 0.77, 0.18) Velocity: (0.16, 0.93, 0.11) State: 2
Position: (0.29, 0.45, 0.17) Velocity: (0.05, 0.01, 0.59) State: 4
Original List:
Age: 23 GPA: 3.90 Name: Isaac Newton
Age: 25 GPA: 2.40 Name: Albert Einstein
Age: 29 GPA: 4.00 Name: Enrico Fermi
Age: 20 GPA: 3.20 Name: Johannes Kepler
Age: 31 GPA: 3.60 Name: Nicolaus Copernicus
Age: 29 GPA: 4.00 Name: Enrico Fermi
Age: 34 GPA: 2.90 Name: Galileo Galilei
Age: 34 GPA: 2.90 Name: Archimedes
Age: 20 GPA: 3.20 Name: Johannes Kepler
Age: 29 GPA: 4.00 Name: Enrico Fermi
Unique'd List:
Age: 23 GPA: 3.90 Name: Isaac Newton
Age: 25 GPA: 2.40 Name: Albert Einstein
Age: 29 GPA: 4.00 Name: Enrico Fermi
Age: 20 GPA: 3.20 Name: Johannes Kepler
Age: 31 GPA: 3.60 Name: Nicolaus Copernicus
Age: 34 GPA: 2.90 Name: Galileo Galilei
Age: 34 GPA: 2.90 Name: Archimedes
```

## 7 Code Cleanliness & Coding Habits (10 points)

- Your code should be submitted with the same filenames and directory structure as the cloned github repository, all contained within a single zip file.
- We should be able to compile all problems using the included makefile with gcc.
- We are not grading to any particular coding standard and we do not have any strict requirements in terms of style. However, your code should be human readable, which means it should be very clean, well commented, and easy for the grader to understand.