ESC190 Lab 4

Hash Tables

Due: March 31, 2020 at 23:59

Grading: This lab is double weighted.

Background

You are part of an underground organization called “The Resistance”. It has come to light that the government has been attempting to infiltrate the organization. You have intelligence that the government spies are using fake identities and applying to jobs at Resistance-affiliated businesses. However, you have the upper hand as the government does not know you have amassed a large amount of personal information which you can use to cross-reference the information of new recruits. To ensure this information is readily available in the future, you have decided to implement a hash table to store, update, and look up the Social Insurance Numbers (SINs), Passport Numbers, and Bank Account Numbers of each person.

Compiling and Linking

To compile e.g. the utilities file, run:

```
gcc -lm -g -o utilities.o -c utilities.c
     -c: stop at the compiling stage, i.e. do not attempt to link
     -o: target output file
     -lm: link math library
     -g: include debugging information
```

If you want to link multiple files which have been compiled as shown above:

```
     gcc -o hash_table utilities.o lab4_part1.o lab4_part2.o main.o
```

If you want to compile and link multiple files in one step:

```
     gcc -o hash_table utilities.c lab4_part1.c lab4_part2.c main.c
```

VALGRIND

Running the following:

```
valgrind -v --leak-check=full ./hash_table
```

should produce no memory leaks and no errors. This will be worth 20% of your lab grade.

DEFINITIONS

- Key: $x \in \mathrm{K}$, SIN number which is an integer.
- Number of buckets: b
- Number of keys: n
- Load factor: $\alpha = \frac{n}{b}$
- $W = 4294967296$, $PHI = 1.618033988749895$
- $a = \frac{W}{PHI}$ (rounded, **not down,** to the nearest integer).

HASH FUNCTIONS

Trivial hash	$h_0(x)$: $x \bmod b$
Pearson hash	$h_1(x)$: Accumulate the result of the hash function as follows. T represents an array of randomly organized integers from 0 to 255 which is given in utilities. *1.* Initialize the result, h, to 0. *2.* Select the least significant digit of x and represent it in ASCII as d_{ASCII}. *3.* Compute the result of h ^ d_{ASCII} *4.* Index T at the result of step (3) *5.* Assign h the result of (4) *6.* Repeat steps (3)-(5) with the next least significant digit; repeat until all digits have been expended. **EXAMPLE:** Consider the case when `SIN` is `123.digit` in the first, second and third iterations of the loop will take on the values of '3', '2' and '1' with ASCII values of `51`, `50` and `49` respectively. The algorithm would yield: `T[T[T[0 xor 51] xor 50] xor 49] mod num_buckets` For an explanation of the XOR operator (^) see Appendix A.
Fibonacci Hash	$h_3(x)$: $\left(\frac{a * x \ \bmod W}{\frac{W}{b}}\right)$ Rounded **down** to the nearest integer.

Collision Handling

Consider a hash function $h(x)$. Consider that we are trying to insert a key $k \in \mathrm{K}$ at bucket b^*. A collision occurs if bucket b^* contains a key from a prior insert. In the event of a collision at bucket b^*, the following methods of collision handling are implemented in this lab: chaining, linear/quadratic probing, and cuckoo hashing.

Chaining

Build a linked list in the bucket. Insert all keys hashed to the bucket into the linked list.

Linear/Quadratic Probing

Refer to the lecture slides for a graphical representation.

1. Let $p = 0$. p represents the p^{th} probe iteration.
2. Compute $b^* = \left(h(k) + c(p)\right) \bmod b$, where $c(p) = p$ for linear probing and $c(p) = p^2$ for quadratic probing.
3. If bucket b^* is empty, insert k and terminate the algorithm. Otherwise, continue to step 4.
4. If bucket b^*contains a key, a collision has occurred. Increase p by 1 and repeat steps (2)-(3), unless, after the increment, $c(p) \geq b$.
5. If no empty buckets have been found by performing the steps above, discard key k.

Cuckoo Hashing

Cuckoo hashing is not a mandatory component of this lab. It is a bonus component.

Using Cuckoo hashing, each key can be placed (and consequently, found) in one of three possible buckets. The three locations are determined by the 3 different hash functions you will implement. We will refer to those as h_0 h_1, and h_2:

h_0: Trivial hash

h_1: Pearson hash

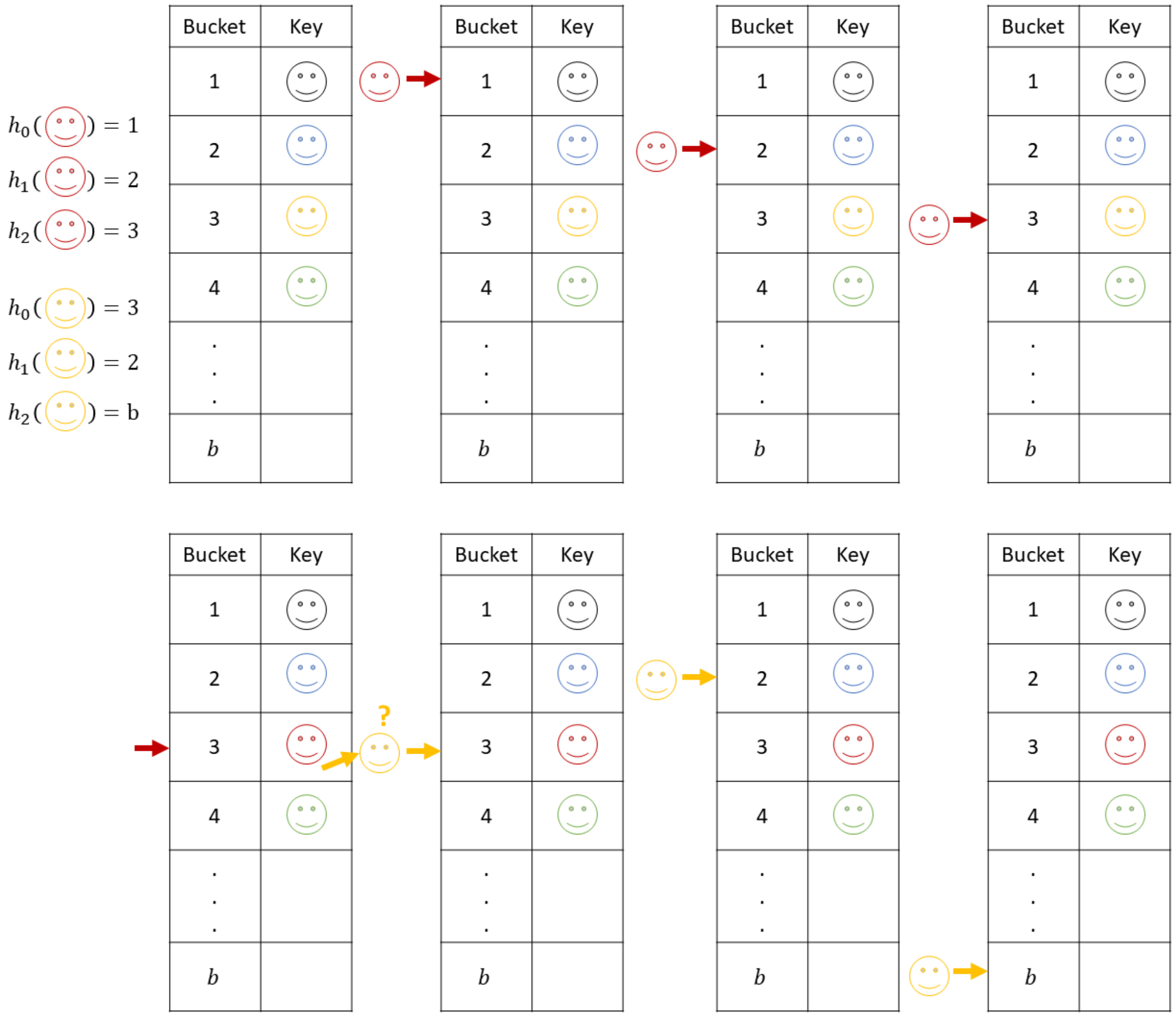
h_2: Fibonacci hash

We will use i to index our hash functions, where $i_{MAX} = 2$.

When inserting a new key k, use the following algorithm:

1. Set $i = 0$. Hash the key with h_i; i.e. compute $h_0(k)$. This result is called b^*.
2. Check bucket b^*. If the bucket is empty, insert k in bucket b^* and terminate the algorithm. Otherwise, continue to step 3.
3. Re-hash the key with h_{i+1}; i.e. compute $h_{i+1}(k)$. This result is the new b^*.
4. Repeat steps (2)-(3) until you successfully insert the key (terminate algorithm) or $i = i_{MAX}$. (reached your last hash function). If the latter occurs, proceed to step 5.
5. Take the key in the most recent bucket b^*. We will refer to this key as k^*. Remove k^* from b^*. Insert k at b^*. Restart the algorithm at step (1), this time inserting key k^*. If k^* collides with k upon reaching $i = i_{MAX}$, delete key k^*.

An example of the algorithm is shown below. Note that this only illustrates one case of the algorithm and you are encouraged to trace it out for other cases.



Prelab

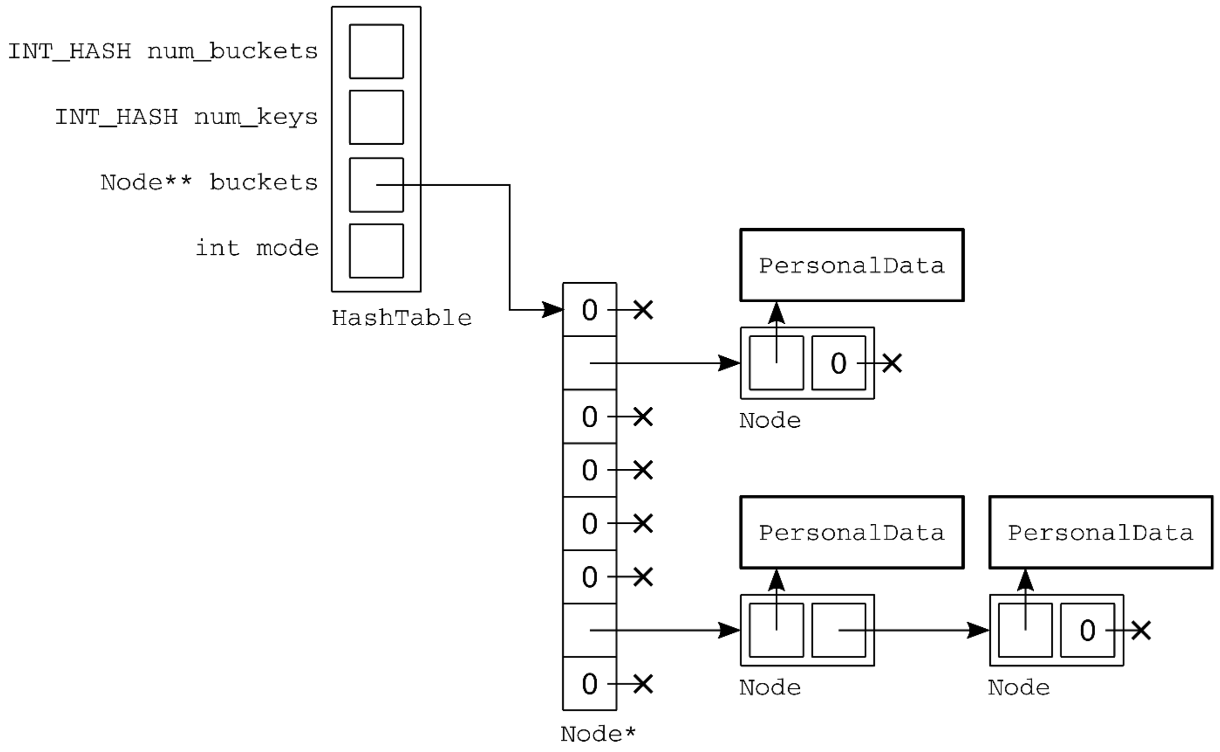
Compute the hashes for the data based on the ASCII representation of the keys. Let the number of buckets be the variable b.

Key (SIN)	Trivial Hash	Pearson Hash	Fibonacci Hash
1			
2			b(0.2360679772682488) rounded down
10		ASCII values of ‘0’ and ‘1’ are 48 and 49 respectively. T[T[0^48]^49]%b =T[39^49]%b=T[22]%b=223%b	
123	123%b		

Fill in how the keys would be placed into the 8-bucket open addressing tables with linear probing, quadratic probing, and cuckoo hashing. Key ‘a’ is placed first, followed by keys ‘b’ and ‘c’.

Bucket	0	1	2	3	4	5	6	7
Linear probing								
Quadratic probing								
Cuckoo Hashing								

Provided for your reference is an illustration of the implementation we will use for a hash table with chaining, where “X” denotes NULL



Illustrate what a hash table with open addressing will look like. (Hint: look at `utilities.h`)

PROVIDED CODE

In `utilities.c` and `utilities.h`, the following have been provided:

Constants	
`MAX_LOAD_FACTOR`	The maximum allowed load factor of a hash table before resizing
`INT_SIN` `INT_HASH`	Predefined datatypes to be used for representing a hash and SIN
`W` `PHI`	Constants for Fibonacci hashing.

`MAX_NAME_LEN`
`BANK_ACC_LEN`
`PASSPORT_LEN`
	Assumptions you may make on char length. You will see how they are used in the `PersonalData Struct`

`PEARSON_LOOKUP`	The array T of randomly organized integers from 0 to 255 to be used in Pearson hashing
Structures	
`PersonalData`	Structure for data being stored
`HashTable`	Structure for hash tables
`Node`	Structure needed for constructing `HashTable`
Functions	

`print_status`
`print_buckets`
`print_personal_data`
	Helper functions for debugging

TASK OVERVIEW

You will implement insert, lookup, and delete on the hash tables you implement by hashing the Social Insurance Number (SIN). You will insert/delete the structured personal data at the appropriate location. You will implement 3 hash functions and 4 types of hash tables. Mode 3 (Cuckoo hashing) is **not** mandatory and is a bonus part of the lab (+1% of entire grade).

Mode		**Trivial Hash**	**Pearson Hash**	**Fibonacci Hash**
0	Closed addressing: Linked Lists	✓		
1	Open addressing: Linear probing		✓	
2	Open Addressing: Quadratic probing			✓
3	Open Addressing: Cuckoo Hashing	✓	✓	✓

TESTING YOUR CODE

You are provided with some code which will run test cases for your code, found in the directory `/test`. These are **not** comprehensive and are only a starting point. You should thoroughly test your code for each function.

PART I: HASH FUNCTIONS

Complete the following functions in lab4_part1.c

```
INT_HASH trivial_hash(INT_SIN SIN, INT_HASH num_buckets);
```

Return the remainder of `SIN` when divided by `num_buckets`.

```
INT_HASH pearson_hash(INT_SIN SIN, INT_HASH num_buckets);
```

Implement the Pearson hash as described above.

```
INT_HASH fibonacci_hash(INT_SIN SIN, INT_HASH num_buckets);
```

Return the hash of `SIN` with the Fibonacci hash function and round **down** to the nearest integer.

PART II: HASH TABLE CORE OPERATIONS

Complete the following functions in lab4_part2.c

Each of these functions should consider when the hash table is in one of modes 0, 1, 2, or 3. You will be evaluated for each mode separately. It is recommended that you work on the modes in the following order:

- Mode 0; closed addressing (chaining as linked list)
- Mode 1; open addressing with linear probing
- Mode 2; open addressing with quadratic probing
- Mode 3; open addressing with cuckoo hashing (bonus)

```
HashTable* create_hash_table(int m, int mode);
```

Return a pointer to an empty, fully initialized, and closed addressed hash table with 2^m buckets, where `mode` is an integer value of 0, 1, 2, or 3.

```
void update_key_without_resize(PersonalData* data,
                                         HashTable* table);
```

Add or update the data as appropriate in the hash table with the `data` being supplied. You may assume the table does not require resizing. Get the `SIN` directly from `data`. This function may be called in `update_key`.

```
HashTable* resize_table(HashTable* table);
```

Create a new hash table which is double the size of the existing hash table (regardless of the load factor) indicated by the pointer `table`. Rehash all keys and insert them in the new hash table. Return a pointer to the new hash table. This function may be called in `update_key`.

```
void delete_table(HashTable* table);
```

Delete the table freeing any heap memory allocated. Only free memory that you allocated in your other functions - do **NOT** attempt to free any other memory. The `main()` function will free the memory allocated in `main()`. This function may be called in `update_key`.

```
void update_key(PersonalData* data, HashTable** table);
```

Add or update the data as appropriate in the hash table with the `data` being supplied and resize the table if needed to prevent MAX_LOAD_FACTOR from being exceeded. Carefully consider the best order of these operations.

Considering calling other hash table functions.

```
int delete_key(INT_SIN SIN, HashTable* table);
```

Delete `PersonalData` associated with `SIN` from the hash table. Return `1` if successful and `0` if the key is not found.

```
PersonalData* lookup_key(INT_SIN SIN, HashTable* table);
```

Return the pointer to the `PersonalData` associated with the provided `SIN`. If the `SIN` is not found, return a `NULL` pointer.

PART III: LOAD AND INSERT DATA

A subset of data to start your testing:

- `test_data_LOAD.txt`
- `test_data_UPDATE.txt`
- `test_data_VALIDATE.txt`

And their respective full datasets:

- `full_data_LOAD.txt`
- `full_data_UPDATE.txt`
- `full_data_VALIDATE.txt`

Now that you have tested out various types of hash tables and probing modes and decided which one is best, you need to use it to store data you have stolen from various sources as well as validate identities of new recruits. Complete the following functions in lab4_part3.c

```
PersonalData** parse_data(char* fn);
```

Return a pointer to `PersonalData*` array parsed from the file with filename `fn`. The memory allocated for these pointers should be dynamically sized based on the size of the file. You may not make any assumptions on the number of data entries. It can be very large. **Hint**: use a helper function to count the number of lines as a `long int`.

```
void counter_intelligence(char* load, char* update,
                              char* validate, char* outfile);
```

Given authentic personal identities in the files with filenames `load` and `update`, verify the authenticity of the data in `validate` and write each `SIN` associated with a fake identity on a new line in the file `outfile`. This function should call `parse_data.`

Example:

```
char* load_fn = "test_data_LOAD.txt";
char* update_fn = "test_data_UPDATE.txt";
char* validate_fn = "test_data_VALIDATE.txt";
char* spies_fn = "spies.txt";
counter_intelligence(load_fn, update_fn,
                             validate_fn, spies_fn);
```

spies.txt

```
781192679
399673474
```

APPENDIX A: XOR OPERATOR

The bitwise XOR operator, represented using ^, takes 2 1-bit inputs A and B and produces results as shown in the table below. Note that the result of the XOR is bit-level high when exactly **one** of its inputs is bit-level high; thus it is known as the **exclusive or** operator.

$\boldsymbol{A}$	$\boldsymbol{B}$	$\boldsymbol{A}$ ^ $\boldsymbol{B}$
0	0	0
0	1	1
1	0	1
1	1	0

For inputs with multiple bits, XOR operates in an element-wise fashion, e.g. 1110_2 ^ $1011_2 = 0101_2$.