DEPARTMENT OF COMPUTING AND INFORMATION SYSTEMS

COMP20007 Design of Algorithms, Semester 1, 2016

Assignment 2: Hashing (Update 11 May)

Due: 5pm Friday 20th May

Objective

In this project, you'll explore more about what makes a good or bad hash function, how to quantify its goodness or badness, and why different kinds of properties are needed in different situations.

This assignment extends Workshop 8, offering an alternative implementation for a hash table, reusing the linked-list module from assignment 1. The skeleton code is on the LMS. You'll need to read it and Dasgupta *et al.* 1.5.2 carefully.

For the purposes of this assignment we'll use separate chaining. Assume that all the elements that clash at a particular location are stored in a linked list, in the order they arrive unless explicitly stated otherwise.

The assignment includes both coding instructions (marked with letters) and theory questions (marked with numbers). Please include the answers to the theory questions in a separate file entitled "report.pdf"

Files: Please download the skeleton code from the LMS. **Do not alter filenames or structure**. The files are as follows.

`Makefile`	Download from LMS and edit to include your student ID.
`main.c`	Download from LMS and do not change.
`hashtable.c, hashtable.h`	"
`list.c, list.h`	"
`array.c, array.h`	"
`types.h`	"
`hash.h`	"
`extra.h`	"
`types.c`	Replace with your own implementation.
`hash.c`	"
`extra.c`	"

Theory of hash functions and hash tables

Let `size` be the size of the table and n the number of elements to be inserted.

1. Suppose the hash function hashes everything to the same bucket. Using big-O notation, write the complexity of insert and search in terms of `size` and n. Justify your answer.
2. Suppose the hash function spreads the input perfectly evenly over all the buckets. Using big-O notation, write the complexity of insert and search in terms of `size` and n. Justify your answer.

3. Suppose that the hash function never hashes two different inputs to the same bucket. Using big-O notation, write the complexity of insert and search in terms of `size` and n. Justify your answer.

Working with strings

The Workshop exercise required the keys to be `int`s. We've recreated the hash table to work with generic pointers, but provided interfaces to work with `int`s. The purpose of this assignment is to explore the universal families of hash functions from Dasgupta 1.5.2.

A0) Data type: The functions `int_eq()`, `int_print()`, and the standard library function `atoi` enable the hash table to work with `int`s, by treating the pointers as `int`s.

Implement the functions `str_eq()`, `str_copy()`, and `str_print()` in `types.c` to enable the hash table to work with strings by treating the input generic pointers as `char *`.

A) Code size: The existing code simply makes the hash table the same size as the command-line argument "size". Now assume that the command-line input is the maximum expected number of elements, implement `determine_size()` in `extra.c` to choose a hash table `size` equal to the next prime after `2*size`. If you like, you can reuse the code at `http://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/isprime.c`, with proper attribution of course.

Linear probing with double hashing

The rest of the assignment assumes the hash table uses separate chaining with linked lists, but *this question only* considers linear probing. (The workshop contrasted open addressing with double hashing approach against the separate chaining collision resolution methods.)

B) Linear probing: The hash table of the assignment also implements double hashing. Implement the function `linear_probing` in `hash.c` so when it's used as the second hash function of a double hashing hash table it performs linear probing as its collision resolution method.

Using randomness from the C standard library

We'll use the pseudorandom number generator from the C standard library. A pseudorandom number generator has two main operations:

- A *seed* operation, in which the pseudorandom number generator is initialised with some source of "real" randomness. We'll use a file.
- A *get* operation, which is a deterministic operation that takes the internal state and returns some "random-looking" bytes, then updates the internal state (for next time).

So for example if `seed` were a number, the following fragment would initialise the pseudorandom number generator with it, then print ten random-looking integers.

```
srand(seed);
for (i = 0; i < 10; i++) {
   printf("%d\n",rand());
}
```

Implementing and testing the hash table

You can see we've added a command-line argument `-s seed` to main.

Remember a key property of hash functions is that they are consistent. This means we get the same result every time we apply it on a given input.

You may wish to look at how the `static` keyword may be used on variables within functions in C to achieve consistency of hash functions.

C) A not-quite-right start: We could consider using the `rand()` function from the C standard library, seeded with `_seed`, to generate a random coefficient $a \in \{0, 1, \ldots, \texttt{size-1}\}$, and using it like this:

```
bad_hash(char *, unsigned int size) {
     /* Generate random a here */
    return a * key[0] % size;
}
```

A stub for `bad_hash` resides in `hash.c`. Complete it.

NEW: Assume that `bad_hash` only has to work with string input. (Of course, it should fail gracefully if not.)

4. Why is this a bad hash function? Give some example input on which the hash table would behave badly if it was using this hash function.

D) Implementing universal hashing: Review the definition of universal hashing from Dasgupta *et al.* 1.5.2. You will construct a universal hash function that takes each character of the string separately. Think of a string as an array of `unsigned char`. You can assume a maximum string length, `MAXSTRLEN`. Assuming that `srand` has been seeded with `seed` as above, generate a list of random integers $r_0, r_1, \ldots, r_{\texttt{MAXSTRLEN}-1}$ in the range $\{0, 1, \ldots, \texttt{size} - 1\}$.

Define `universal_hash(Key k, unsigned int size)` to implement:

$$\textit{universal-hash(string)} = \left(\sum_{i=0}^{\textit{length(string)}-1} r_i * \textit{string}[i] \right) \bmod \texttt{size}$$

Note that string[i] means the i-th character of the string, starting from 0.

Tip: To generate random values in the range $\{0, 1, \ldots \texttt{size} - 1\}$, use `rand() % size`. This isn't strictly perfect (think about whether it wraps around an integer number of times), but it will do fine.

5. **Average-case analysis** Assume that $r_0, r_1, \ldots$ are chosen randomly and independently of the list of inputs. Using big-O notation, write the *expected* complexity of insert and search in terms of `size` and n. Justify your answer.

E) Generating collisions by trial and error: Suppose you're a malicious annoyance who wants to slow down the hash table to make other users' lives difficult. Suppose also that the r_i values for universal-hash are made public. Write a function `collide_dumb(unsigned int size, unsigned int seed, int n)` that generates n strings that `universal_hash()` hashes to 0. For this question, do this the unintelligent way: simply enumerate a lot of strings and keep any that hash to 0, until you have at least n of them. The output should be printed to `stdout` one line per string. To avoid suspicion, your list must not repeat any strings, even padded with zeros. You may, of course, call `universal-hash`.

NEW: Output format: write your output to `sdtout.` It should contain:

- `Num_randoms` (The number of `r[i]` values to appear), followed by $'\backslash n'$
- the random elements `r[0], r[1],` ..., `r[Num_randoms-1]` in order, one per line,
- your clashing strings, one per line.

6. Explain briefly (in a few sentences) how your algorithm works. Using big-O notation, write the expected running time of your algorithm, for generating 2 hashes to 0 in a table of size `size`.

F) Generating collisions by clever maths: Look at the extended Euclid algorithm on p.21 of Dasgupta *et al.* Write a new function `collide_clever(unsigned int size, unsigned int seed, int n)` that achieves the same thing as `collide_dumb()`, but more efficiently. ~~For full credit, your strings must have only ASCII alphanumeric characters.~~

Note: your function need not necessarily work for large values of n. Full marks as long as it works for at least $n = 2$ and fails gracefully if n is too large.

NEW: Two significant simplifications to make this easier.

1. Assume that `size` is less than the maximum value of char. (so at most 255)
2. Never mind about alphanumeric chars. Let's just get some strings of some chars that hash to the same value.

There's also a big hint in an announcement from 10 May.

Partial credit for two strings that hash together, but not to zero. Partial credit also for an algorithm that works for many, but not all, possible values of `r[0], r[1], ...`.

NEW: Output format: same as (E).

3. Explain briefly (about half a page) how your algorithm works. Using big-O notation, write the expected complexity of your algorithm in terms of `size`. Justify your answer. You can assume that integer operations (like + and *) take a constant amount of time.

Cryptographic hashing—Optional unmarked section for those who got the openssl library to compile

A cryptographic hash function has all the properties of a "good" hash function, plus some extras. Cryptographic hash functions should also be collision-resistant and preimage-resistant, as described in

lectures. You just showed that universal-hash is not collision-resistant, given public randomness. (It's not preimage resistant either.)

One popular cryptographic hash function is SHA-256. The '256' part means that it outputs a digest of 256 binary digits. (Recent advice suggests that SHA-256 is a little outdated, and that we should be switching to at least SHA-384, but we'll stick with 256 for this project.)

4. (Optional) How many guesses would you have to make before you found, with at least 50% probability, a message that SHA-256 hashes to the string of all 1's? How about a string with 30 zeros at the start?

Optional. Cryptographic hash implementation MINGW's openssl suite includes a SHA-256 implementation. Take the example code, `sha256test.c` from the LMS. Check that you have the right libraries installed—on MinGW, you might need to add `msys-libcrypt, msys-libopenssl` and `msys-openssl`, including the dev version when there is one. You may also need to take some care when compiling. On a default MinGW setup under windows, you can use the `-I` option to tell gcc where to look for a `#include`'d file, and the `-L` option to tell it where to find a library to link. A command something like this should work. `gcc -Wall -IC:/MinGW/msys/1.0/include -o sha256test sha256test.c -lcrypto -LC:/MinGW/msys/1.0/lib`

Bitcoin's proofs of work use SHA-256 to prove that someone has guessed lots of hashes. Roughly, you make a block with some transactions ("Eve pays Rao $50" let's say) and concatenate a random element r. If you can find a value of r for which

$$\textit{SHA-256("Eve pays Rao \$50"}||r) < \textit{threshold}$$

you've mined a block and earned some money.[1]

Optional: Generate hash collisions Adapt the example code to test through random input values to SHA-256 until you get 12 zeros at the start. How many tries do you need on average? How long does it take? Use the timing techniques from Workshop 8. Do some online research to find the current threshold for bitcoin. How long would it take you to use your code to mine a real block?

Optional: Blockchain manipulation Suppose that instead of SHA-256, the bitcoin blockchain used universal-hashing in which the r_i values were public. Explain how you could spend the same coin twice.

Submission

Submission is via LMS under COMP20007, "Assignment 2". Submissions will close automatically at the deadline. NB. Machine and network load/problems right before the deadline is not a sufficient excuse for a poor or missing submission. As per the Subject Information Sheet from Lecture 1: "The late penalty is 20% of the available marks for that project for each day (or part thereof) overdue. Submissions more than three days late will not be accepted."

Submit a single archive file (e.g. `.tar.gz` or `.zip`) via the LMS containing the four files: `types.c`, `hash.c`, `extra.c`, `report.pdf`. It should unpack into a folder, where the folder is named using your student id. To make this easy, we've added a submission target to the Makefile. You need to locate the line `STUDENTID = <STUDENT-ID>` and modify it with your student id. Then you can type `make submission` to create the required archive file.

Submissions not adhering to these requirements will be subject to a 2 point penalty.

[1] Actually, to be inserted into the blockchain you have to be the *first* to do this, but never mind.

Marking: Half the marks are for your code and half are for your report.

The seven coding parts (A0 through F) of the assignment will be marked for correctness and clarity. You will lose a mark if your submission does not identify you in the opening comment of each submitted file; you will lose a mark if your solution is incorrect in some way (breaks on certain inputs, has memory leaks, requires fixing to work at all); you will lose a mark if your solution is outrageously inefficient when there was a more efficient solution; and you will lose a mark if your solution is difficult to interpret (minimal or unhelpful comments, obscure variable names).

In your report, you should include clear answers to the numbered questions. These will be marked for clarity and correctness.

Academic honesty: All work is to be done on an individual basis. You may discuss ideas with other students but you must write your code and your report individually and independently. Any code or ideas sourced from third parties must be attributed. Please see the Subject Information Sheet for more information. Where academic misconduct is detected, students will be referred to the School of Engineering for handling under the University Discipline procedures.