15-122: Principles of Imperative Computation — Spring 2018

Lab 8: Hashing — 27 February

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-hash .
% cd lab-hash
```

Grading: You should finish (1.a), (1.b), and (1.c) for two points. Additionally, finish (2.a) for three points.

Finding collisions in hash functions

Recall that a hash function $h(k)$ takes a key k as its argument and returns some integer, a *hash value*; we can then use $\text{abs}(h(k)\%m)$ as an index into our hash table. In this lab you will be examining various hash functions and exploiting their inefficiencies to make them collide.

It will be convenient to will denote a string of length n (for $n > 0$) as $s_0 s_1 s_2 \ldots s_{n-2} s_{n-1}$, where s_i is the ASCII value of character i in string s. (A partial ASCII table is given to the right.) We define four hash functions as follows:

hash_add: $h(s) = s_0 + s_1 + s_2 + \cdots + s_{n-2} + s_{n-1}$

hash_mul32:

$$h(s) = (\ldots((s_0 \times 32 + s_1) \times 32 + s_2) \times 32 \cdots + s_{n-2}) \times 32 + s_{n-1}$$

hash_mul31:

$$h(s) = (\ldots((s_0 \times 31 + s_1) \times 31 + s_2) \times 31 \cdots + s_{n-2}) \times 31 + s_{n-1}$$

hash_lcg:

$$h(s) = f(f(\ldots f(f(f(s_0) + s_1) + s_2) \cdots + s_{n-2}) + s_{n-1})$$

where $f(x) = 1664525 \times x + 1013904223$

Partial ASCII Table

32	20	␣	64	40	@	96	60	‘
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	’	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C		
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			

These four hash functions have been implemented for you and can be run from the command line like this, for example:

```
% hash_add
Enter a string to hash: bar
   hash value = 311
   hashes to index 311 in a table of size 1024
Another? (empty line quits):
```

Note that the command line hashing tool also reports where the element with the given key will hash given a table size of 1024. This is important because hash tables have a limited size, so we want to minimize collisions within said size.

The first exercise requires you to mathematically reverse-engineer one of the simpler hash functions:

(1.a) Find three or more strings, each string containing three or more characters, that would always collide because they have the same hash value using `hash_add`.

1pt

Now, let's work through a more complicated real-world example: hashing an entire dictionary. We would like to know which hashing function would be the best to hash the Scrabble dictionary. We define a hashing function to be "better" based on how efficiently it spreads out the words over the buckets. Obviously, this depends on the size of our hash table: if we have a smaller hash table, there will naturally be more collisions. That's why we can use a visualizer (implemented for you in file `visualizer.c0`) to see how many words hash to each bucket for a given hash function.

(1.b) Implement your own version of `hash_mul32` in `hash-a.c0` so that the function `hash_string(s)` returns an integer representing the hash value for `s` using the formula given on the previous page. The string library may be helpful in this. You can compile your code and run it with the following command:

```
% cc0 hash-a.c0 hash-dictionary.c0 visualizer.c0
% ./a.out -o mul32.png
% display mul32.png
```

This will output a graphical visualization of your hash function on the dictionary for a table of size 1024, with the vertical lines showing how many values hashed to that index in the table. You can run your program with the `-n` flag followed by a different table size if you like. You can see just how ineffective `hash_mul32` is!

(1.c) Now, similarly implement `hash_lcg` in `hash-b.c0`, and compile it for the dictionary:

```
% cc0 hash-b.c0 hash-dictionary.c0 visualizer.c0
```

Run it like above to see how well it hashes the dictionary. Compare this to `hash_mul32`.

2pt

Hashing Faculty

In file `profs.txt`, there is a list of CS faculty info, which we will parse for you into the following structs. We would like to hash such structs into a hash table (some fields may be blank):

```
typedef struct prof prof_t;
struct prof {
  string name;  string title;  string office;  string email;
  int area_code;     // 0 if no phone number
  int phone;         // 0 if no phone number
};
```

(2.a) Implement two different ways of hashing faculty within a function called `hash_prof(prof_t* p)` in two different C0 files. Try and think of two different strategies. You can use your code for hashing a string if you would like to hash the strings separately.

Compile and run your code as follows:

```
% cc0 <your hash file>.c0 hash-profs.c0 visualizer.c0
% ./a.out -o mullcg.png
```

Run them through the visualizer as you did earlier and compare the results. You and your neighbor can each write one and then compare. Try and understand what makes a better hashing function! **Hint:** try lowering the bucket size as there is way less faculty than words in the dictionary.

3pt