Areege Chaudhary
10197607
March 19, 2017

CISC235 Assignment 3: Hashing

Part 1: Converting code words into integers

For this assignment, I chose to use one of the functions that were given in the course lecture notes. Using a for loop, the function takes a character from the code word string and uses the Ord() function to return the Unicode representation of that character. It then adds that value to the current sum multiplied by some constant. The final value that is returned is a summation of the calculation done with every character in the string. I chose to use the value 5381 as the starting sum and 33 as the constant. 5381 is used because it has shown (through testing by other individuals) to result in fewer collisions and better "avalanching" than other numbers. Avalanching is a term used in cryptography that refers to the avalanche effect: a property of an algorithm that allows it to significantly change an output with only a slight change in input. As well, the 33 increases efficiency of the algorithm (because it allows for bit shifts instead of actual multiplication). These characteristics were important for me to consider when selecting the proper method to use, as I wanted to create a combination of functions for this assignment that were the least time consuming during execution, and allowed me to find the smallest table size for the hash functions I decided to use.

Part 2: Quadratic Probing experimentation

For both parts 2 and 3, I created charts that summarized the experimenting that I did. To start, I experimented with different hashing functions (all from the course notes). The mid-square method and the multiplication method did not work out so well. The mid-square method was giving me an average comparison rate of around 7-8 comparisons per execution. And no matter what I did to the C1/C2 values or the table size, I could not get past this. I wasted a lot of time trying to figure out how to meet the probe depth requirement, and eventually realized that I needed to change the hash function itself as what I was using was not well suited for the characteristics of the keys I was inputting. I then looked at the multiplication method and despite spending quite a bit of time trying to get it to work, I decided to just move on to the sum of digits method instead. I had a bit more luck with this. After a few iterations of running the program with the recommended C1,C2 values, I realized that the table size always had to be the maximum value that the hash function outputted. For example, if my key was 2034 and running it through my hash function resulted in the value 15, I knew that I had to have index 15 in the array available. Therefore, the array had to have a size of at least 16 (to account for array[0] and array[15]). So then, I started calculating the max sum outputted by the hash function and using that to figure out the minimum table size I needed. To decrease the table size, I then started dividing the sum(highlighted in blue in my table) to be outputted by the hash function. I kept increasing the divisor while keeping the same C1, C2 values until I reached a point where increasing the divisor resulting in not meeting the probe depth requirement.

Areege Chaudhary
10197607
March 19, 2017

Hash Function	**Hash Function Code**	**C1**	**C2**	**Table Size (works if it meets requirements of probe depth)**

mid square	`s = key*key`
	`x = s // 1000`
	`a = x % 1000`
	`return a`
		1, 2	1, 0.5	Doesn't fit requirements

Multiplication method	`Temp = 0.61803398875*key`
	`frac, whole = math.modf(temp)`
	`x = frac`
	`return floor(len(array)*x)`
				Doesn't work because key integers are too large. It might have worked if I spent more time on it, but I decided to move on to the next type of hash function

Sum of digits	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    return sum`
		1, 2	1, 0.5	Error at Table Size: 300 000 (hashfunc output is too large)
				
				Table Size: 1 000 000 works! (max sum output is 563933) therefore, table size: 563934 works as well

	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    sum = sum // 2`
	`    return sum`
		1, 2	1, 0.5	Table size: 281967 works

Areege Chaudhary
10197607
March 19, 2017

	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    sum = sum // 10`
	`    return sum`
		1, 2	1, 0.5	Table size: 56394 works

	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    sum = sum // 50`
	`    return sum`
		1, 2	1, 0.5	Table size: 11279 works

	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    sum = sum // 55`
	`    return sum`
		1, 2	1, 0.5	Table size: 10254 works

	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    sum = sum // 56`
	`    return sum`
		1	1	Table size: Doesn't work, reaches 11 comparisons

		2	0.5	Table size: 10071 works

	`sum = 0`
	`while key:`
	`    sum = sum*2 + key % 10`
	`    key = key // 10`
	`    sum = sum // 65`
	`    return sum`
		2	2	Table size: 8675
				
				After this division, greater divisions do not meet the requirement of <= 10 comparisons so

Areege Chaudhary
10197607
March 19, 2017

				I moved on to part 3

Part 3: Double Hashing Experimentation

For this experiment, I did the same as I stated above with dividing the output of H'(k) with increasing values, but I also tried out three different functions for H''(k). The results of my experimentation are summarized below.

H'(k) code	**H''(k) code**	**Table Size (works if it meets requirements of probe depth)**

`sum = 0`
`while key:`
`    sum = sum*2 + key % 10`
`    key = key // 10`
`    sum = sum // 80`
`    return sum`
	`key = key**2`
	`return key`
		7049

`sum = 0`
`while key:`
`    sum = sum*2 + key % 10`
`    key = key // 10`
`    sum = sum // 90`
`    return sum`
	`key = key**2`
	`return key`
		6265

`sum = 0`
`while key:`
`    sum = sum*2 + key % 10`
`    key = key // 10`
`    sum = sum // 99`
`    return sum`
	`key = key**2`
	`return key`
		5696
		
		After this, implementing greater divisors resulted in a hashing that did not meet the requirement

`sum = 0`
`while key:`
`    sum = sum*2 + key % 10`
`    key = key // 10`
`    sum = sum // 120`
`    return sum`
	`s = key % size`
	`return s`
		4699
		
		This is the lowest possible table size value that I found through my experiments. Dividing sum by greater values resulted in hashing that did not meet the requirement. I am sure that if I used a different combination of h'(k) and h''(k)

Areege Chaudhary
10197607
March 19, 2017

		functions, I would get to a table size that was even lower but due to time constraints, I was unable to get further in progress than this.

`sum = 0`
`while key:`
`      sum = sum*2 +`
`key % 10`
`      key = key // 10`
`      sum = sum // 125`
`      return sum`
	`s = (k + 31)**2`
	`return s`
		Does not meet comparison requirement

`sum = 0`
`while key:`
`      sum = sum*2 +`
`key % 10`
`      key = key // 10`
`      sum = sum // 70`
`      return sum`
	`s = (k + 31)**2`
	`return s`
		8056
		
		I could not get to a value lower than this by changing the sum divisor.

Part 4: Discussion

Overall, my experiments do seem to show that it is much easier to get smaller table sizes using double hashing than it is when using quadratic probing when dealing with collisions. As shown above, after a variety of different tests, I only got to 8675 as the minimum table size when using quadratic probing. However, when I started testing for double hashing, I immediately started getting results that were lower than anything I could get to with quadratic probing. These results obviously varied depending on the different functions (for double hashing) and C1/C2 values I used (for quadratic probing) but in almost every test I did, I got table size values that were lower than the lowest size I got to in the previous experiment (8675). I am sure that with more time, I would have been able to find combinations that would get me to table sizes lower than 4699. There's just so many different variables involved with getting to the results that unless you get a table size of 2500, you cannot say that you have found the most efficient solution.

I confirm that this submission is my own work and is consistent with the Queen's regulations on Academic Integrity.