Analyzing Search Times of Hash Maps
Lab #9

By
Michael Harden

CS 303 Algorithms and Data Structures
November 9, 2021

1. Problem Specification

The goal of this lab is to implement a Hash Map data structure and populate it with pre-sorted and randomly ordered data. Then, test the implementation. Also, allow for users to their own data to create and search a Hash Tables. And finally, to calculate the time it takes to look up values in a hash map, for both a hash table populated with the pre-sorted and randomly sorted data and compare the times. Also examine how changing the array size alters the look up time.

The Hash Map implementation was to provide the ability to use three probing strategies.

1) linear probing – (H(x) + 1) mod <array size>.
2) Quadratic probing – (H(x) + i^2 mod <array size>.
3) A variant of the linear probing strategy, (7H(x) + 1) mod <array size>.

2. Program Design

This program contains two classes. First is the HashMap class. This class implements the Hash Map data structure It provides methods to put and get values from the table, The second class is the Driver class. This class provides methods to parse a csv file, time the get function for HashMap, and allows users to create and search their own tables in the command line.

The following methods are provided in the HashMap class:

a) __init__(size, probing):

   Initializes a new Hash Map. The size parameter sets the size of the table, and probing assigns which of the above probing strategies should be used.

   The three probing strategies are stored as class properties and can are implemented as lambda functions. When constructing a new hash map, the probing function you wish to use must be passed as the probing argument as one of the following options.

   - HashMap.hash is the variant of the linear probing
   - HashMap.linear is linear probing
   - HashMap.quadratic is quadratic probing.

   This is built in such a way that a client can use their own probing strategy method by means of a lambda function in the parameter, probing

b) put(key, value):

   The put function, first the corresponding hash to the key provided is computed by taking the mod of the key. If the calculated index is already populated, the index is rehashed by taking the previous index calculated as the parameter to the prescribed probing function. This process is repeated until ether an empty spot is found, or the function hashes back to where it began. Once ether case is met, the index is used to insert the key value pair in to the hash map.

c) get(key):
The get function works in a very similar way to the put function. First the corresponding hash to the key provided is computed by taking the mod of the key. If the calculated index is already populated with a tuple containing a key other than the key provided, the index is rehashed by taking the previous index calculated as the parameter to the prescribed probing function. This process is repeated until ether an empty spot is found, a tuple is found containing the key provided, or the function hashes back to where it began. In the former and later cases, False is returned from the function, signaling the value is not present. If the second case is met, the key value pair is returned from the function.

The following methods are provided in the Driver class:

a) start ():
Called at the beginning of the program. Similar to a main() function in java. This function also allows for user input.

b) load_file(file_path):
takes a csv file and parses the data into an array of tuples with the key value pairs

c) load_keys(file_path):
takes a csv file and parses the keys into an array of integers

d) time():
Times how long it takes to get the values stored for list of keys for all three probing strategies on hash map array sizes of 50, 100, 500, 1,000. Both the average time per key, and the total time for the list of keys is returned

3. Testing

To test the HashMap class, a hash map with size 8 was created and populated with a list of key value pairs (1-9, ‘a’-‘i’) The following test cases were used to verify that values were being properly over written and to make sure the hash function was finding the correct values.

Test cases:

Key	Expected Description
1	False
9	(9, ‘i’)
5	(5, ‘e’)

A hash map of length 5 was also created that contained the inputs
[(0, ‘a’), (8, ‘b’), (16, ‘c’), (24, ‘d’), (32, ‘e’)] to verify the probing was working correctly. When get() was called with each key, the proper value was returned.

All three probing methods were tested in the same way and worked correctly

4. Analysis and Conclusions

In conclusion, The hash map is a very fast storage strategy. With look up times for all probing strategies in the range of microseconds per element on average, it appears to be an almost instant option, and the fasted explored so far. One hesitation I have is in the size of the array. With over 200,000 elements being put into the hash map, values are being over written so many times there is a 99.95% probability you would have to explore the entire hash map before realizing the value is not in the table, when the hash map is 100 elements long. If the map contains 1,000 elements, there is a 99.5% probability you will have to search the entire hash map before realizing the value is not in the table. However because the hash maps are so small it would explain why there are still fast times, yet would also go to explain with why it is taking more time to search through the bigger hash maps than the smaller ones. Since the time complexity to compute a hash is only O(1) it would not make since for an increase in table size to also increase searching time, unless you have to look at every element which is O(n) time complexity.

One idea is if the hash maps size gets closer to 50,000 indices to store all 200,000 pairs, then for each key to search for, there is a 25% probability that it is not overwritten giving a quicker look up time on average since there is less probability that a value would be over written meaning the whole hash map should be searched. This method was attempted, however, the time it took to build the data structure, and search for values that were not present was to intense for the computer it was being run on.

The times to search for the given values are as follows

Searching for Keys Times

In Order

		Hash Probing Function					
		7H(x) + 1		Linear		Quadratic	
		Avg time (secs)	Total Time (secs)	Avg time (secs)	Total Time (secs)	Avg time (secs)	Total Time (secs)
Table Size	50	1.55E-06	5.41E-05	1.54E-05	5.38E-04	7.06E-06	2.47E-04
	100	1.51E-06	5.29E-05	3.39E-05	1.19E-03	7.51E-06	2.63E-04

500	4.52E-06	1.58E-04	1.10E-04	3.86E-03	7.49E-05	2.62E-03
1,000	5.00E-06	1.75E-04	1.95E-04	6.81E-03	1.47E-04	5.14E-03

Random Order

	Hash Probing Function					
	7H(x) + 2		Linear		Quadratic	
Table Size	Avg time (secs)	Total Time (secs)	Avg time (secs)	Total Time (secs)	Avg time (secs)	Total Time (secs)
50	2.80E-06	9.80E-05	1.52E-05	5.32E-04	4.01E-06	1.40E-04
100	1.74E-06	6.08E-05	2.93E-05	1.03E-03	8.37E-06	2.93E-04
500	7.94E-06	2.78E-04	1.73E-04	6.05E-03	1.24E-04	4.33E-03
1,000	6.68E-06	2.34E-04	3.32E-04	1.16E-02	2.58E-04	9.02E-03

5. Credits

- The example lab report provided on canvas was used as a template for this lab report.