CompSci 261P

Data Structures

Project 1 - **Hash Methods**

Qixiang Zhang

qixianz@uci.edu

70644128

Feb 15 2019

Outline

1. Project Requirements
2. Approach Overview
3. Hashing Implementation Details
    a) Code organization
    b) Hashing algorithms
        i. General behaviors
        ii. Linear hashing
        iii. Chained hashing
        iv. Cuckoo hashing
        v. Chained hashing with Tabulation
4. Tests and Results
5. Summarize

1 - Project Requirements

In this project you are to implement several different hashing algorithms perform an empirical comparative analysis of their running times. The hashing algorithms to be implemented are:

a) Linear hashing
b) Chained hashing
c) Cuckoo hashing
d) At least one additional algorithm.

For the empirical analysis, run the algorithms on a variety of test cases to determine the relative efficiency of the algorithms as functions of various parameters, such as the number of elements and the load factor. How do your results compare with theoretical predictions of worst-case and amortized-time. You are encouraged to consider other factors as well. For example, if you generate your test cases randomly, you how do your choices of probability distribution and the distribution parameters affect your results?

2 - Project Overview

These questions will be answered with empirical tests:

*(**operations** include "insert key value pair", "search key", and "delete key")*

Factor 1 - Load Factor:

1. How does the preset load factor affect the time complexity of all operations?
2. How does the load factor affect the number of collisions?
3. How does the load factor affect the number of re-hash?
4. How does the load factor affect the time complexity of all operations?

Factor 2 - Population (number of elements):

5. How does the population affect the time complexity of all operations?
6. How does the population affect the number of collisions?
7. How does the population affect the number of re-hash?

Factor 3 - Pre-sorted keys:

8. How does pre-sorting the keys affect the time complexity of all operations?

Implementation Overview:

In this project, 4 hash methods are implemented using **Python 3x** :

1. Linear hashing
2. Chained hashing
3. Cuckoo hashing
4. Chained hashing with tabulation

Assumptions on inputs:

1. The 4 hashing implementations are mimicking Python's dictionary data structure but only for 32-bit integer keys ($2^{31} - 1$).
2. For convenience, both keys and values are positive integers. No duplicated keys or values are allowed.
3. The keys are always start with 1, then increment with a random integer between 1 and 19. For example, the first key would be 1, second key could be a number between 2 and 20, the third key would be an integer based on the second key.
4. The values will be the increments of 1. If to generate 500 key, value pairs, the values would be 0, 1, 2, ..., 499
5. The input key, value pairs are stored under /inputs/ as .txt files. Every time when any testing requires inputs, the function util.readInputs(population, shuffle_flag, random_seed) requires the number of elements to be generated, flag for sorted inputs or not, and a seed number for random generation. Every time the function is called, inputs are re-generated, and converted into integers.

3 - Hashing Implementation Details

Code Organization:

./src/
main.py

../test_script/
tests.py
void t1(const string* hash_name, const string* test_name, const string* hash_flag)
void t2t3t4(const string* hash_name, const string* test_name, const string* hash_flag)
void t5t6t7(const string* hash_name, const string* test_name, const string* hash_flag)
void t8t9t10(const string* hash_name, const string* test_name, const string* hash_flag)

../hash_methods/
Chaining.py
class Chaining
Cuckoo.py
class Cuckoo
LinearProbing.py
class LinearProbing
Tabulation.py
class Tabulation

../util/
util.py
bool isPrime(int number)
void generateInputs(int range)
list readInputs(int range, bool shuffle)

../inputs/
...(txt files of key, value pairs)

Hashing Algorithms:

Common Members	Description
num_rehash	- Number to record the count of re-hash operation - Increment when rehash function is called - during Insertion
population	- Number of stored elements
alpha_max	- The largest load factor that the hash method can take before re-sizing
cap	- The total number of slots to store the elements (start with 107; for Cuckoo is 106, 53 for both tables)
table	- The map of all elements that stores the keys and the associated values

Common Behaviors	Description
search(key)	- Search through the hash table using the key value and the hash function - Return True, location, and associated value if found, else return False, None, None
insert(key, value)	- Insert the key value pair into the hash table using the key and the hash function - The key has to be unique otherwise an exception would be thrown
delete(key)	- Delete the key value pair in the hash table using the key and the hash function - If the key is not found in the table, an exception would be thrown
get_collision()	- get the current number of collisions in the hash table
current_alpha()	- get the current load factor of the hash table

__rehash_up()	- Triggered when the table is too small or there is an infinity loop when inserting a key value pair. Increment the number of rehash - Call __sizeup() and __rehash() to do the resize operation
__sizeup()	- Call __next_prime() to compute the next appropriate table size
__next_prime(num)	- Compute an immediate larger prime number from the given number
__rehash(new_cap)	- Copy the old key value pairs into a new hash table, update the reference for the hashing object

Linear Hashing Algorithm

Hash function:

- *h1(key) = key* **mod** *current_table_size*

Pseudo code:

```
search( key ):
    loc = key mod table_size
    while table[loc] is not empty:
        if (table[loc][0] is the same key):
            return True, loc, table[loc][1]
        else:
            check the next adjacent slot if empty
    return False, loc, None

insert( key, value ):
    if (current load factor >= the load factor threshold):
        size up the table and rehash
    found, loc, value = search( key )
    if (found):
        raise Exception("Same key already exists in the hash: {}\n".format(key))
    else:
        insert the key value pair

delete( key ):
    found, loc, v = search( key )
    if (found):
        delete the key
        call function __shiftback( loc ): from the loc at the table, check the next key if the
            key value pair belong to the place until encounter empty slot. (**wrap
            around the table**)
    else:
        raise Exception('{} not found'.format(key))

get_collision():
    size = self.cap
    res = 0
    for every key k in the table:
        if k is not in the right spot:
            res += 1
    return res
```

Important Notes:

Linear hashing uses eager deletion - after deleting a key, eagerly check if the immediate next key is in the correct location. If it is not in the correction location (by computing the hash value), re-insert (delete and insert) the key-value pair. Then repeat the process until an empty slot is encountered. The process would not necessarily stop at the end of the table. It still needs to continue from the beginning of the table unless countering an empty slot.

Chained Hashing Algorithm

Hash function:

- *h1(key) = key* **mod** *current_table_size*

Pseudo code:

```
search( key ):
    loc = key mod table_size
    Loop check all the slots at table[loc]:
        if one of the slot at table[loc] is the same key):
            return True, loc, value, slot_number
    return False, loc, None, 0

insert( key, value ):
    if (current load factor >= the load factor threshold):
        size up the table and rehash
    found, loc, value, slot = search( key )
    if (not found):
        if the slot is empty,
            insert the key-value pair
        else:
            append the key-value pair
    else:
        raise exception

delete( key ):
    found, loc, value, slot_number = search( key )
    if (found):
        delete the key-value pair
    else:
        raise Exception('{} not found'.format(key))

get_collision():
    size = self.cap
    res = 0
    for every slot i in the table:
        chain_size = the number of key-value pairs at table[i]
        if chain_size > 1:
            res += (chain_size - 1)
    return res
```

Important Notes:

Chained hashing is a straight-forward hashing algorithm. Collision is handled by appending the new key-value pair at the end of the list in each slot. Therefore, there is really no limit how many elements the table can hold. However, abusing this feature can slow down the search for each key.

Cuckoo Hashing Algorithm

Hash functions:

- *h1(key) = key* **mod** *current_table_size*
- *h2(key) = floor(key*101/current_table_size)* **mod** *current_table_size*

Pseudo code:

```
search( key ):
    loc1 = h1( key )
    loc2 = h2( key )
    Check table1 at loc1 and table2 at loc2 for the key
        If found it,
            return True, table_number, slot_location, value
        If one of the table slot is None,
            return False, table_number, slot_location, None
    return False, "X", loc1, loc2

insert( key, value ):
    if the current load factor is larger than the threshold:
        call __rehash_up to size-up both tables and rehash
    Use search( key ) to get flag, table_number, slot_number, value
    if (flag)
        raise exception
    else:
        if table_number is not "X"
            insert the key
        else
            flag_success = try cuckoo_insert the key-value pair
            if (flag_success)
                do nothing (key inserted successfully)
            else
                size up the table
                flag_success = try cuckoo_insert the key-value pair

delete( key ):
    found, table_number, slot_number, value = search( key )
    if (found):
        delete the key-value pair
    else:
        raise Exception('{} not found'.format(key))

get_collision():
    return the smaller population of the two tables

__cuckoo_insert( k0, v0, table ):
    make a new table with the same size as the old table
    try insert the k0 v0 pair into the new table
        if failed (detected infinity loop):
            return False and the original table
        if succeed:
            return the new table
```

```
__rehash( new_size ):
    make a new empty table with the new_size
    try hash all the original key value pairs from the old table
        if failed (detected infinity loop):
            return False
        else:
            return True

__rehash_up():
    get the next size of the table (at least double the old size)
    call __rehash() to see if the rehash operation is a success
        if failed:
            size up and try again
```

Important Notes:

The trickiest thing for Cuckoo hashing is to handle the infinity loop when pushing elements around between the two tables. The algorithm detect the infinity loop by checking if the same key-value pair is to be inserted into the same slot second time. Additionally, since the algorithm re-size and rehash the hash tables whenever there is an infinity loop or the load factor above a threshold. If the rehash is implemented recursively, it will overflow the stack. Therefore, it must stop the process and increment the size before do the rehash operation again.

Chained Hashing with Tabulation Algorithm

Hash function:

- *h1(key) = table1 [(key) & 0xff]*
- *h2(key) = table2 [(key>>8) & 0xff]*
- *h3(key) = table3 [(key>>16) & 0xff]*
- *h4(key) = table4 [(key>>24) & 0xff]*
- *h(key) = (h1* **XOR** *h2* **XOR** *h3* **XOR** *h4)* **mod** *current_table_size*

Pseudo code:

```
search( key ):
    same as Chained hashing

insert( key, value ):
    same as Chained hashing

delete( key ):
    same as Chained hashing

get_collision():
    same as Chained hashing

__getloc( key, table_size ):
    h1 = self.t1[ (key)  & 0xff ]
    h2 = self.t2[ (key>>8)  & 0xff ]
    h3 = self.t3[ (key>>16)  & 0xff ]
    h4 = self.t4[ (key>>24)  & 0xff ]
    h = h1 ^ h2 ^ h3 ^ h4
    return h % cap

__build_hashmap(self):
    t1 = [floor(random.random()*0xffffffff) for _ in range(256)]
    t2 = [floor(random.random()*0xffffffff) for _ in range(256)]
    t3 = [floor(random.random()*0xffffffff) for _ in range(256)]
    t4 = [floor(random.random()*0xffffffff) for _ in range(256)]
```

Important Notes:

Chained hashing with tabulation is just the classic chained hashing with a spin of new hash function. The key must be a 32-bit integer. To get the hash value, must generate 4 tables in the beginning with 256 random non-negative integers each. Partition the key into 4 parts of 8 bits each. Then use the 4 values compute 4 new values using bitwise AND with 255. Use the new values to look up the number in the 4 tables. Lastly, use the bitwise XOR to get a value and fit into the hash table using modulo the table_size.

4 - Tests and Results

1. How does the preset load factor affect the time complexity of all operations?
2. How does the load factor affect the number of collisions?
3. How does the load factor affect the number of re-hash?
4. How does the load factor affect the time complexity of all operations?
5. How does the population affect the time complexity of all operations?
6. How does the population affect the number of collisions?
7. How does the population affect the number of re-hash?
8. How does pre-sorting the keys affect the time complexity of all operations?

To best answer the questions above, I set up 4 tests to play around with different parameters - load factor, number of elements to be inserted, and pre-sort the keys before insertion.

To mitigate input bias, I used 3 fixed randomized seeds from Numpy to generate inputs for each test (33, 9723, 11937) Results will take the average of all three.

Factor 1 - Load Factor:

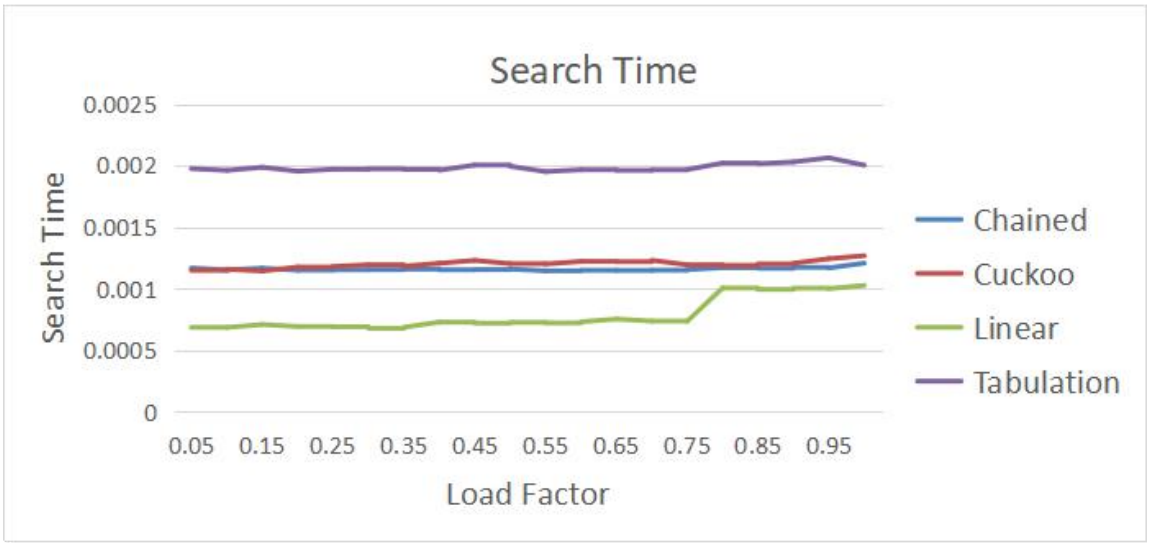
Test 1: Change load factor

This test is designed to see how does the maximum load factor affect the time complexity for all operations. The test will set different load factor thresholds for each batch operation (insertion, search, and deletion). The timer begins right before and after each batch of same operations. The test will insert, search, and delete 700 key-value pairs in each batch for each preset load factors.

- Preset load factors { 0.05, 0.10, ..., 0.90, 0.95, 1.00 }
- number of insertions, searches, deletions: 700
- randomize seed (numpy): 33, 9723, 11937

Results and Observations:

From the chart above, we can see that the insertion time drops flat as the load factor threshold increases for linear hashing, chained hashing, and chained tabulation hashing. When the load factor threshold is low (0.05), all hashing algorithms have to re-hash many times to accommodate the threshold.



The search time operation does not seem to be affected by the load factor threshold except for the linear hashing. Since the load factor is high, there are more collisions in the hash table. Linear hashing has to look through all non-empty adjacent slots to look for the key.

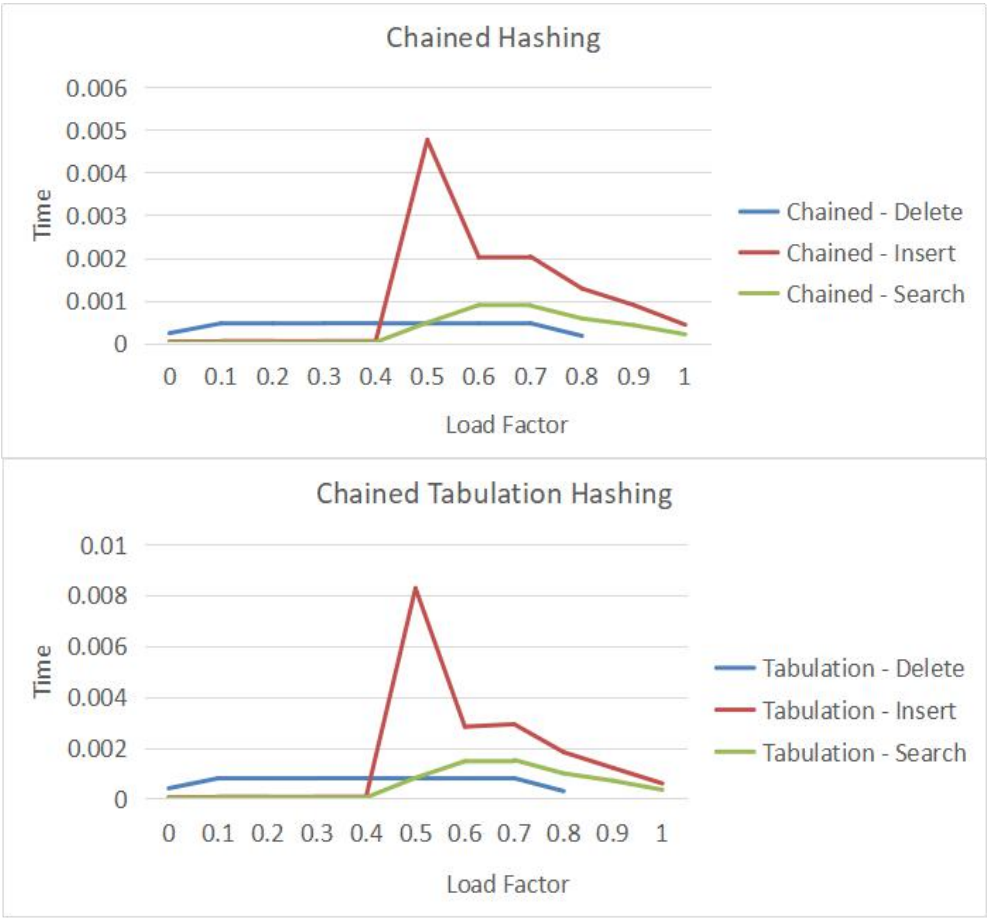
From the chart above, similar results occur comparing to the search time chart. The eager deletion effect gets amplified here when the load factor is close to 1.

Test 2-3-4: Single operation time

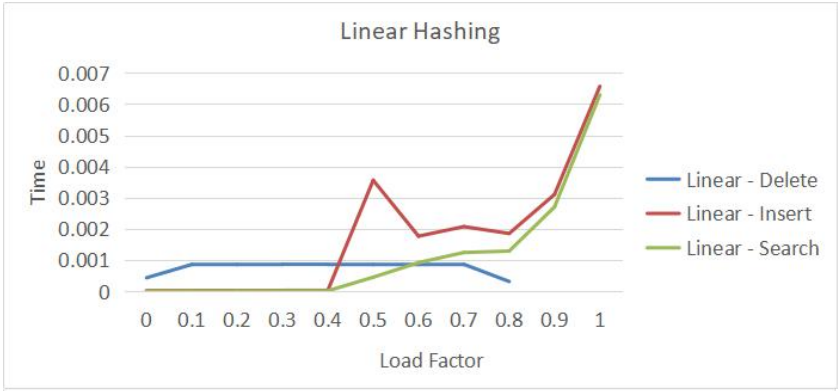
This test is designed to see how does the immediate load factor has any correlations with insertion, deletion, and search runtime for different hashing algorithms, as well as number of collisions and count of re-hash operations. Different from test 1, this test measures single operations instead of batch operation. Time measure is summed manually after each single operation (without counting any irrelevant instructions from the test itself). The max load factor is set at 1. The immediate load factor is measured by calling the current_alpha() function in each hash class.

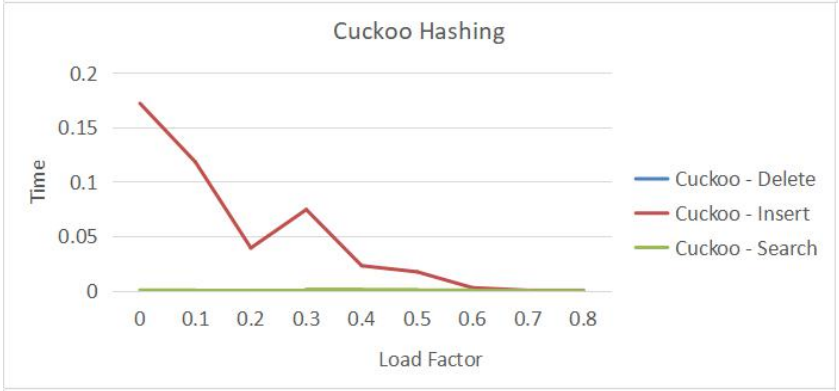
- max load factor: 1.0
- min load factor: 0.0
- time measured by accumulating every single operation (e.g. insert 1 element)
- number of insertions, search, deletions: 1435
- randomize seed (numpy): 33, 9723, 11937

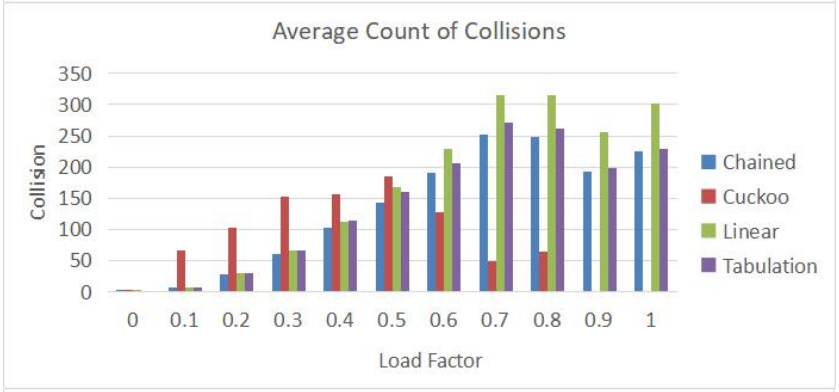
Results and Observations:

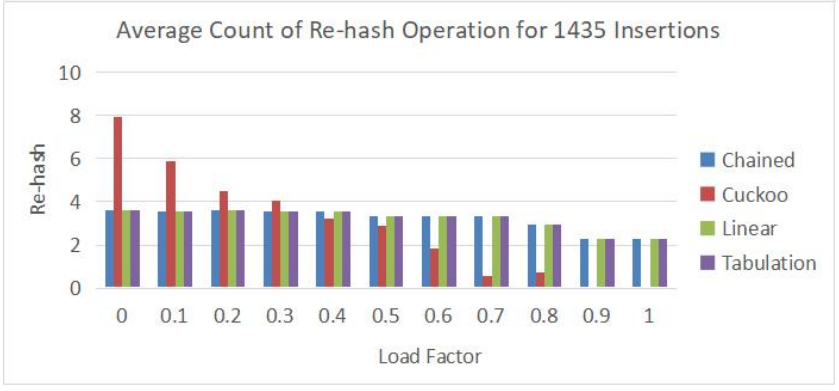


From the two charts above, the shape of the regular chained hashing and the chained hashing with tabulation almost look identical. Chained Tabulation hashing takes a little longer to run for each operation. For insertion, time peaked when the load factor is at 0.5. The only reason that happened is because it is probably when the algorithm decided to do re-hash operation.









Linear hashing's search and delete runtime peaked when the load factor is close to 1. When the load factor is high, there are more collisions and cause the linear hashing search time to increase, same applies to the linear hashing deletion operation. Cuckoo hashing's insert time decreases as the load factor increases. The second chart correlates with the bottom two charts in terms of number of collisions and number of re-hash operation. Cuckoo has to do many re-hash operations when the load factor is very low.

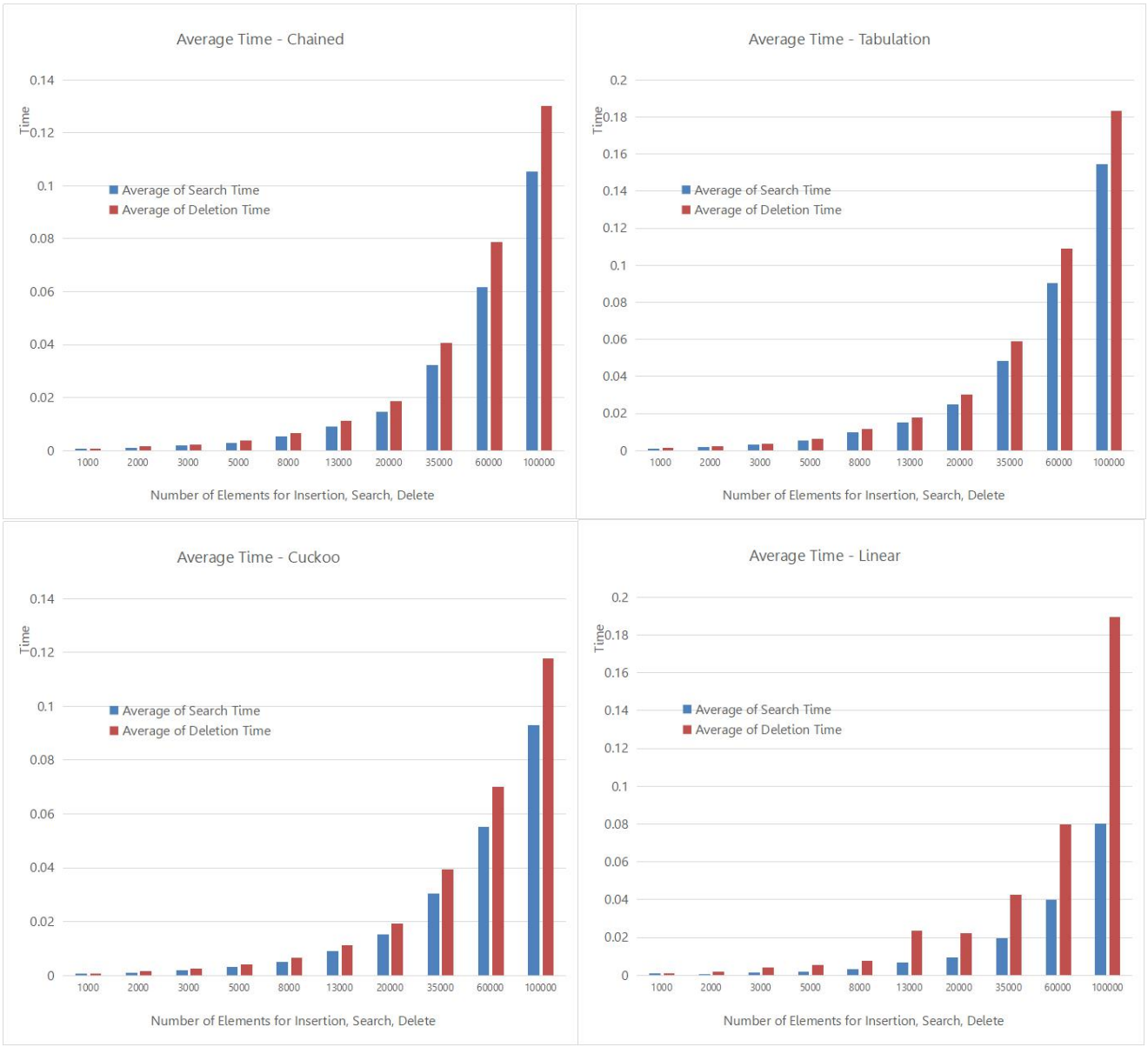
Factor 2 - Population (number of elements):

Test 5-6-7: Change Population

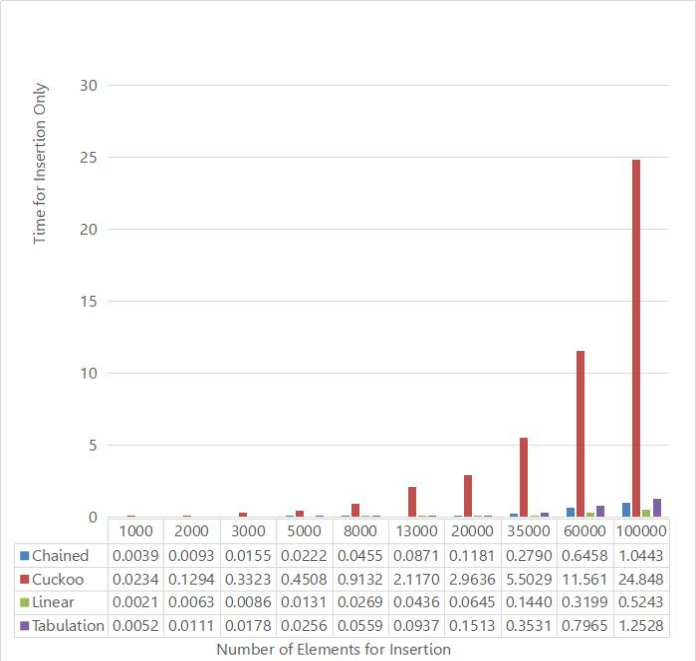
This test is designed to see how the number of elements to be inserted, searched, and deleted affects the length of runtime, number of collisions, and number of re-hash. The results below take the average of each load factor threshold and across each random seed.

- Fixed max load factor { 0.2, 0.5, 0.8 }
- Populations { 1k, 2k, 3k, 5k, 8k, 13k, 20k, 35k, 60k, 100k }
- randomize seed (numpy): 33, 9723, 11937

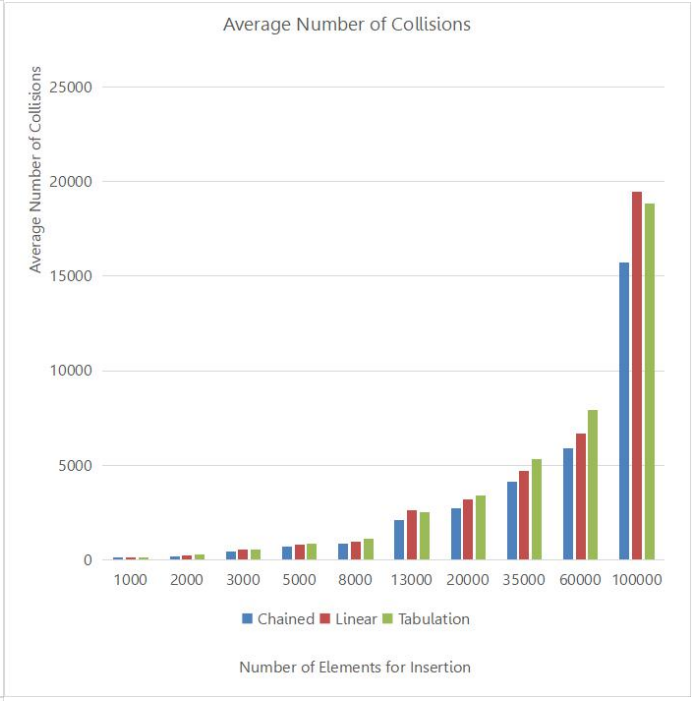
Results and Observations:

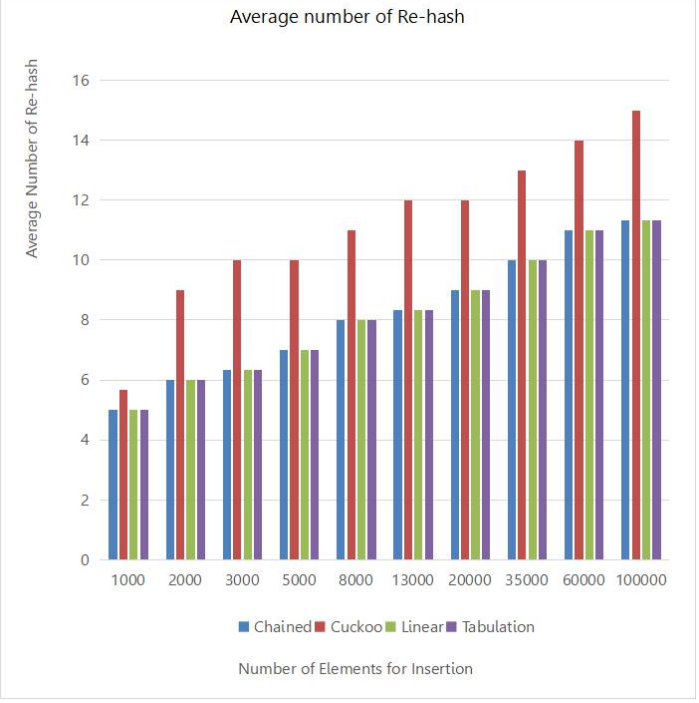


From the 4 charts above, we can conclude that the increase number of elements will results in increase length of search and deletion time. Among all 4 tests, Cuckoo takes least amount of time to search and delete. Tabulation Chained hashing takes a little longer than the regular chained hashing. Since linear hashing uses eager deletion, the deletion time is always about twice as longer than the search time.



	1000	2000	3000	5000	8000	13000	20000	35000	60000	100000
Chained	0.0039	0.0093	0.0155	0.0222	0.0455	0.0871	0.1181	0.2790	0.6458	1.0443
Cuckoo	0.0234	0.1294	0.3323	0.4508	0.9132	2.1170	2.9636	5.5029	11.561	24.848
Linear	0.0021	0.0063	0.0086	0.0131	0.0269	0.0436	0.0645	0.1440	0.3199	0.5243
Tabulation	0.0052	0.0111	0.0178	0.0256	0.0559	0.0937	0.1513	0.3531	0.7965	1.2528





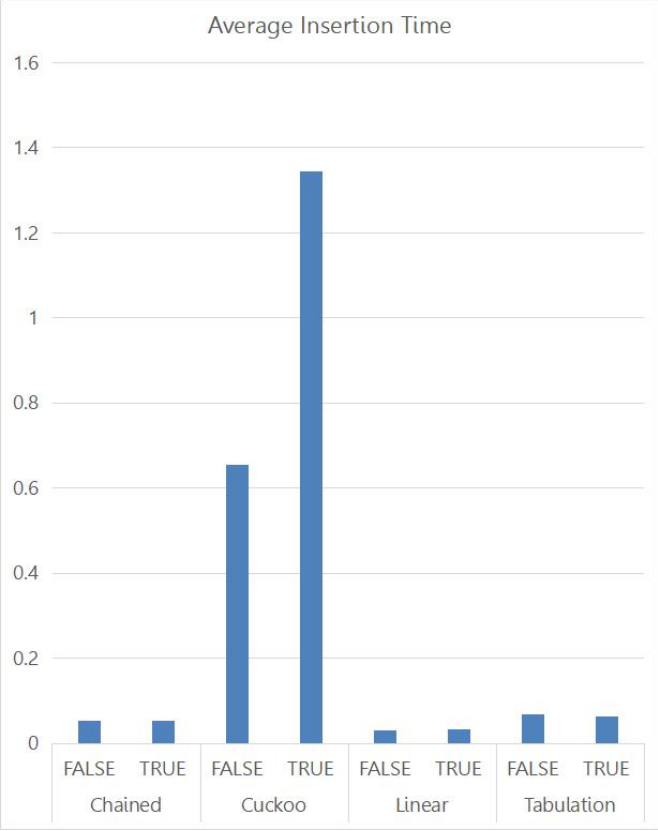
Similar theory can be deducted based on the above charts. High population in a hash table leads to higher number of collisions, therefore, more re-hash operations during the insertion operation, which leads to higher insertion time. Cuckoo's insertion time gets amplified since it keeps looping between two tables finding an empty slot.

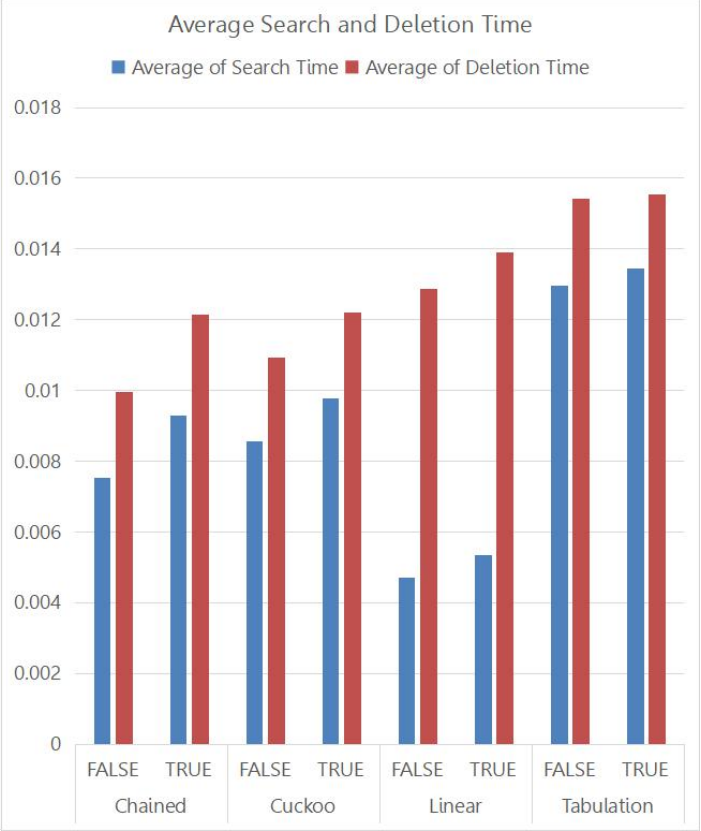
Factor 3 - Pre-sorted keys:

Test 8: Pre-sort the keys

This test is designed to observe when pre-sorting the inputs will have any effects on the regular operation's runtime. The results will take the average across multiple max load factors and random seeds. Each batch operation takes 10000 key-value pairs (one set of pre-sorted and the other randomized). This can be achieved by setting the shuffle flag to True or False. True means the inputs are randomized. False means the inputs are pre-sorted in ascending order.

- Fixed max load factor { 0.2, 0.5, 0.9 }
- Shuffle flag { False, True }
- Number of inserts, search, deletions: 10000
- randomize seed (numpy): 33, 9723, 11937





For average insertion time, there is no obvious difference among chained hashing, linear hashing, and chained hashing with tabulation. However, for Cuckoo hashing, the insertion time doubled when the input keys are randomized. For average search time and deletion time, all 4 hashing algorithm behaves similarly to sorted and unsorted input key-value pairs. All 4 hashing methods take less time to do search or delete on sorted inputs than unsorted inputs.

5 - Summarize

Factor 1 - Load Factor:

Immediate load factor indicates how dense the hash table is. Max load factor sets the threshold which decides when the algorithm does re-hash operation during insertion. Both Cuckoo hashing and Linear hashing operating time are sensitive to the load factor. When the load factor is close to 1, the tables are becoming more full and with high number of collisions. It will take longer to insert an item into the table.

Factor 2 - Population (number of elements):

The population has the most direct effect on the runtime of each hashing algorithm. Higher population leads to longer operation time. It also leads to higher number of collisions and, therefore, more re-hash operations.

Factor 3 - Pre-sorted keys:

If pre-sort the keys before inserting into the hash table, the insertion time may not be reduced. The reasons is that there may be gaps between keys. For this project, keys are integers with 1 to 19 gaps in between. However, if to insert keys that are with gaps of only 1 and in sorted order, I suspect that the time would reduce dramatically since there will be less collision (assuming the hash function is just Key Modulo Table_size).

Conclusion:

1. **The Big-O notation and amortized time may not always apply**

From this project, we can see that the runtime for insertion, search, and deletion varies differently and cannot be easily generalized by the Big-O notation or amortized time. There are other factors including the load factor, population, and the specification on the input keys.

2. **Choosing a good hashing method**

Choosing the best hashing method depends on the need and usage of the hashing. If the search time is strictly limited, Cuckoo hashing is recommended since it has a guaranteed constant search time (only needs to search between 2 places). However, the draw back is that it takes a long time to insert all the keys. Linear hashing is a very easy algorithm to implement but it is always a good idea to keep the load factor very well below 1. Chained hashing is usually the default hashing method when the requirements are not clear. Its behavior is predictable and the hash function can be modified to minimize the number of collisions. Tabulation with chained hashing behaves nearly the same as regular chained hashing. It takes a bit longer to compute the hash value but may stamp down the number of collisions.

3. **Project Extension**

Re-sizing is only implemented when the tables do not have enough space. Sizing down can also be considered to be put into the deletion operation at the cost of increasing deletion time. I am also curious about exploring better hash functions.