Project 1: Hashing Algorithms

Yifeng Zhang
21406209
`yifengz7@uci.edu`

November 1, 2019

1 Project Overview

1.1 Background

In this project, two different hashing algorithms are implemented, and an empirical comparative analysis of their performance are performed on these two hashing algorithms.

For this project, the hashing algorithm were:

1. Chained Hashing
2. Cuckoo Hashing

1.2 How to Execute the Program

The algorithms were implemented and tested under Python 3.7, and Jupyter IPython Notebook was used to run the test programs and generate figures.

The script used to test the algorithms are stored in the .ipynb files, and can be directly ran by opening them with Jupyter Notebook.

I also exported the .ipynb files as .py files. The .py files can be directly run by the Python interpreter.

2 Assumption

2.1 Input

For convenience, all the keys and values are non-negative integers with in one million. The keys and values are generated randomly and distinctively using `random.sample(range(x), length)`, where `x` is the maximum possible number generated, and `length` is the number of numbers generated.

2.2 Initialization

For both Chained Hashing and Cuckoo Hashing, the table length used to evaluate the performance is 1000.

2.3 Performance Evaluation

The performance are evaluated at each load factor. In explanation, in each iteration, a pair is inserted, and a key is searched. The amount of time used for each operation is recorded to generate the figures.

Set/Insert The metric for the performance of insertions is designed to be the cumulative running time versus the current load factor. In other words, it measures the time used to achieve the current load factor. Note that for Cuckoo Hashing, the time used to rehash is also counted as a part of insertion.

The number of insertions performed to test the performance is ten times of the length of the table.

Intuitively, when highly occupied, the performance of the hash tables are expected to drop.

Searching The metric for the performance of searches is designed to be the cumulative running time versus the total number of searched performed. In other words, it measures the time used to complete the current number of searches.

The number of searches performed to test the performance is ten times of the length of the table.

Intuitively, when highly occupied, the performance of the hash tables are expected to drop.

2.4 Load Factor

The load factor α is defined to be the number of elements (key-value pairs) in the hash table divided by the total number of cells in the hash table.

$$\alpha = \frac{len(table.pairs())}{len(table)}$$

3 Chained Hashing

3.1 Introduction

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.

In this method, each bucket is independent, and has a linked list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the linked list operation.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per bucket—that is, it is roughly proportional to the load factor.

For this reason, chained hash tables remain effective even when the number of table entries n is much higher than the number of slots.

The worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number n of entries in the table.

Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache ineffective.

3.2 Implementation

The implementation is pretty straightfoward, with a Node class and a ChainedHashTable class. Note that the hash function is chosen to be

$$h(x) = x \mod len(table)$$

. Given that the input data are randomly generated and are expected to distribute uniformly, this saves much time.

```
class Node:
    def __init__(self, key: int = None, val: int = None):
        self.key = key
        self.val = val
        self.next = None

    def __str__(self):
        return f"({self.key}, {self.val})"


class ChainedHashTable:

    def __init__(self, length: int):
        self.array = [Node() for _ in range(length)]
        self.num_items = 0
        self.h = lambda x: x % length

    def __str__(self):
        s = ""
        for i in range(len(self.array)):
            s += f"[{i}]"
            it = self.array[i]
            while it:
                if it.val is None:
                    s += "head->"
                else:
                    s += str(it) + "->"
                it = it.next
            s += "end"
            s += "\n"
        return s

    def insert(self, key: int, val: int):
        index = self.h(key)
        it = self.array[index]
        while it.next:
            it = it.next
        it.next = Node(key, val)
        self.num_items += 1
        return 0

    def lookup(self, key: int):
        index = self.h(key)
        it = self.array[index]
        while it:
            if it.key == key:
                return it.val
        return False
```

3.3 Analysis

1. $t/num_{operations} = O(1 + length(H[h(k)]))$
2. We make random hash function assumption:
   - For each k, $h(k)$ is uniformly distributed over indices.
   - For any two keys k_1, k_2, $h(k_1)$ and $h(k_2)$ are independent.
3. E[time/operation]
   = O(1+len(H[h(k)]))
   = O(1+ E[number of keys colliding with k])
   = O(1+ $\sum_q$ Pr[q collides with k])
   = O(1 + (n-1)/n)
   = O(1+a)

4 Cuckoo Hashing

4.1 Introduction

Cuckoo hashing ensures constant lookup and deletion time in the worst case, and constant amortized time for insertions (with low probability that the worst-case will be encountered). It uses two or more hash functions, which means any key-value pair could be in two or more locations. For lookup, the first hash function is used; if the key-value pair is not found, then the second hash function is used, and so on.

If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets. By combining multiple hash functions with multiple cells per bucket, very high space utilization can be achieved.

4.2 Design

This implementation of Cuckoo Hashing does not resize automatically. The reason is that if the hash table is resized, then the load factor after resizing would not be consistent with that before resizing.

4.3 Implementation

The trickiest thing for Cuckoo hashing is to handle the infinity loop when pushing elements around between the two tables. The algorithm detects the infinity loop by checking if the same key-value pair is to be inserted the third time.

In the implementation of Cuckoo Hashing, the two hash functions are

$$h_0(x) = ((a_0x^3 + b_0x^2 + c_0x + d_0) \mod p) \mod len(table)$$

$$h_1(x) = ((a_1x^3 + b_1x^2 + c_1x + d_1) \mod p) \mod len(table)$$

, where a, b, c, d are randomly generated integers between $[1, 10000]$ and p is a prime number larger than $len(table)$.

For the consistency of the load factor, in this implementation of Cuckoo Hashing, no resizing is performed.

Instead, when seeing a loop in keys, this implementation simply rehashes exactly once, and tries to insert the key again. If the rehashing or the insertion fails, abort the insertion and return 1. This can prevent the program from entering a endless loop of finding different hashing functions and reinsert the elements (since when the load factor is relatively high, it is highly likely that there will be loops among keys).

```
class CuckooHashTable:

    def __init__(self, length: int):
        self.arrays = [[None for i in range(length)] for j in range(2)]
        self.length = length
        self.h = None
        self.reset_hash()

    def get_load_factor(self):
        return len([i for i in self.arrays[0] if i is not None] + [i
           for i in self.arrays[1] if i is not None]) / (2 * self.
           length)

    def __str__(self):
        return f"[0]: {self.arrays[0]}\n[1]: {self.arrays[1]}"

    def reset_hash(self):
        a1, b1, c1, d1 = get_random_numbers(4)
        a2, b2, c2, d2 = get_random_numbers(4)
        p1 = get_random_prime(self.length)
        p2 = get_random_prime(self.length)
        self.h = [lambda x: ((a1*x**3 + b1*x**2 + c1*x + d1) % p1) %
           self.length, lambda x:  ((a2*x**3 + b2*x**2 + c2*x + d2) %
           p2) % self.length]

    def search(self, key):
        for i in range(2):
            if self.arrays[i][self.h[i](key)] is not None:
                return self.arrays[i][self.h[i](key)]
        return None

    def set(self, key, val):
        ret = self._set(key, key, val, 0)
        if ret == 1:
            rehash_failed = self.rehash()
            if rehash_failed == 1:
                print(f"rehashing failed")
                return 1
            ret = self._set(key, key, val, 0)
            if ret == 1:
                print(f"after rehashing, insertion failed")
        return 0

    def _set(self, key, first_key, val, time, array_id=0):
        if key == first_key:
            time += 1
            if time == 3:
                return 1
        if self.arrays[array_id][self.h[array_id](key)] is not None:
            dis = self.arrays[array_id][self.h[array_id](key)]
            self.arrays[array_id][self.h[array_id](key)] = (key, val)
            return self._set(dis[0], first_key, dis[1], time, 0 if
                array_id == 1 else 1)
        else:
            self.arrays[array_id][self.h[array_id](key)] = (key, val)
            return 0

    def rehash(self):

        old_arrays = copy.deepcopy(self.arrays)
        old_hash = copy.deepcopy(self.h)
        elems = []
        for i in range(2):
            for j in range(self.length):
                if self.arrays[i][j] is not None:
                    elems.append(self.arrays[i][j])
                    self.arrays[i][j] = None

        self.reset_hash()
        ret = 0
        for e in elems:
            ret = self._set(e[0], e[0], e[1], 0)
            if ret == 1:
                break

        if ret == 1:
            self.arrays = old_arrays
            self.h = old_hash
            print("abort, restore to previous status")
            return 1
        else:
            print("rehashing done")
            return 0
```

4.4 Analysis

If Cuckoo hashing is used over a sequence of n operations:

1. With probability 1-$\theta(1/n)$, it works.
2. With probability $\theta(1/n)$, rehashing is needed. Thus we have the following:
3. Probability of 1 rehash: $O(1/n)$
4. Probability of 2 rehash: $O(1/n^2)$
5. Probability of 3 rehash: $O(1/n^3)$

Thus the expected number of rehashes is O(S), where S = $1/n$ + $2/n^2$ + ... = O(1)

Since the rehash can be done in O(n) time, the expected time for the sequence operations is O(n).

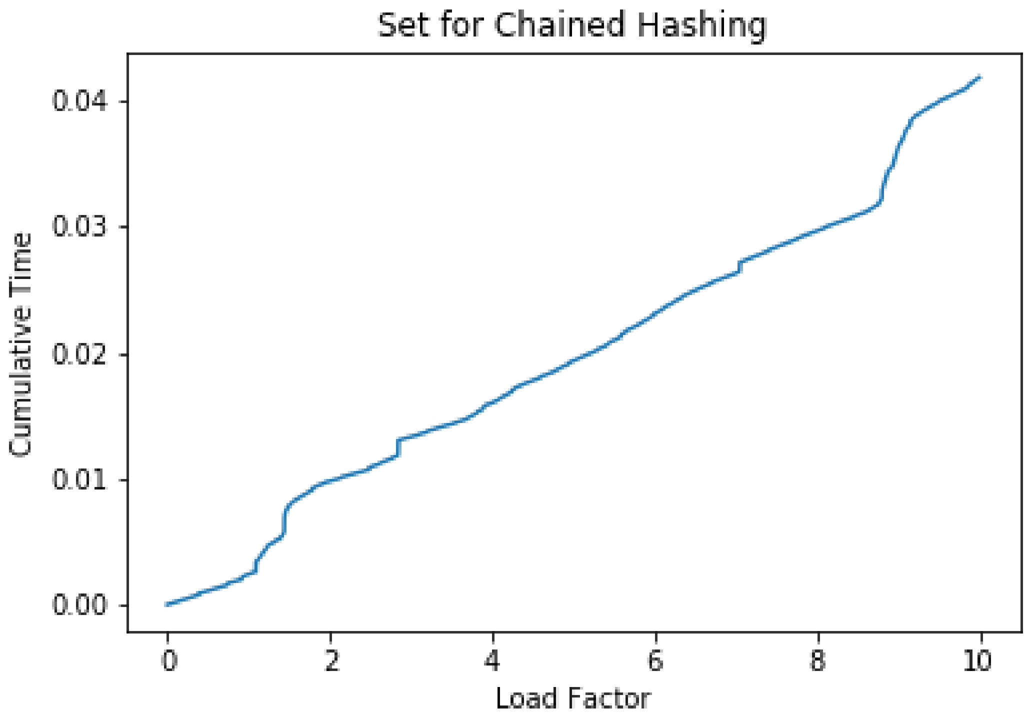
Note that after going into a loop, new hash functions and rehashing are needed.

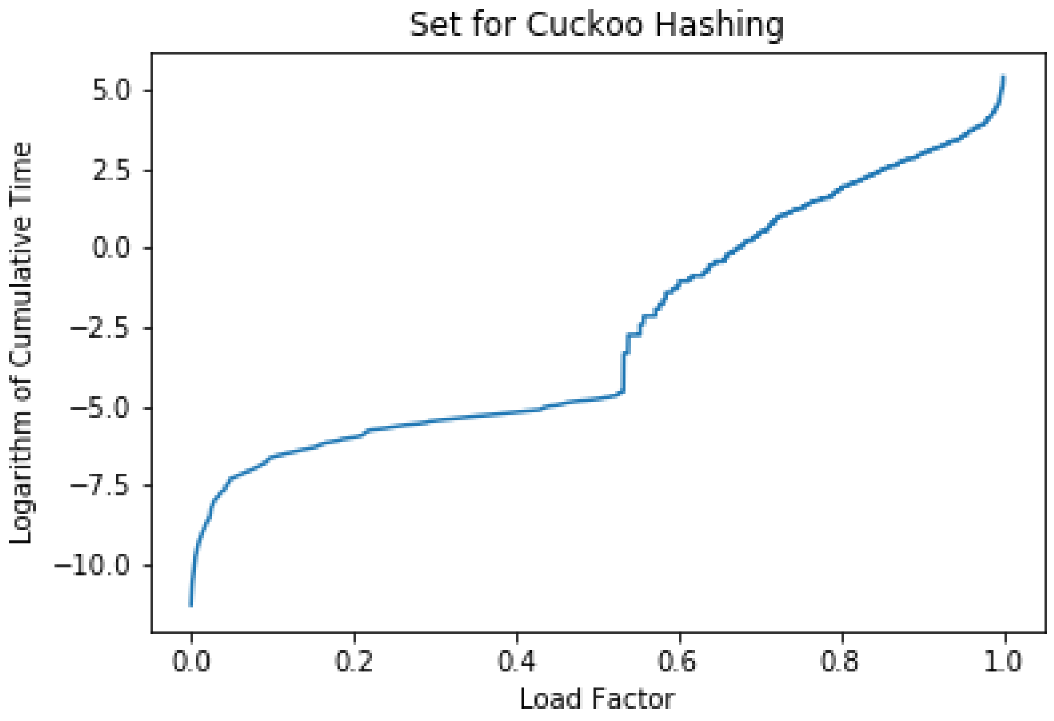
5 Testing

The figures generated are not in log-log form, since in my metric, log-log figures are not very easy to read.

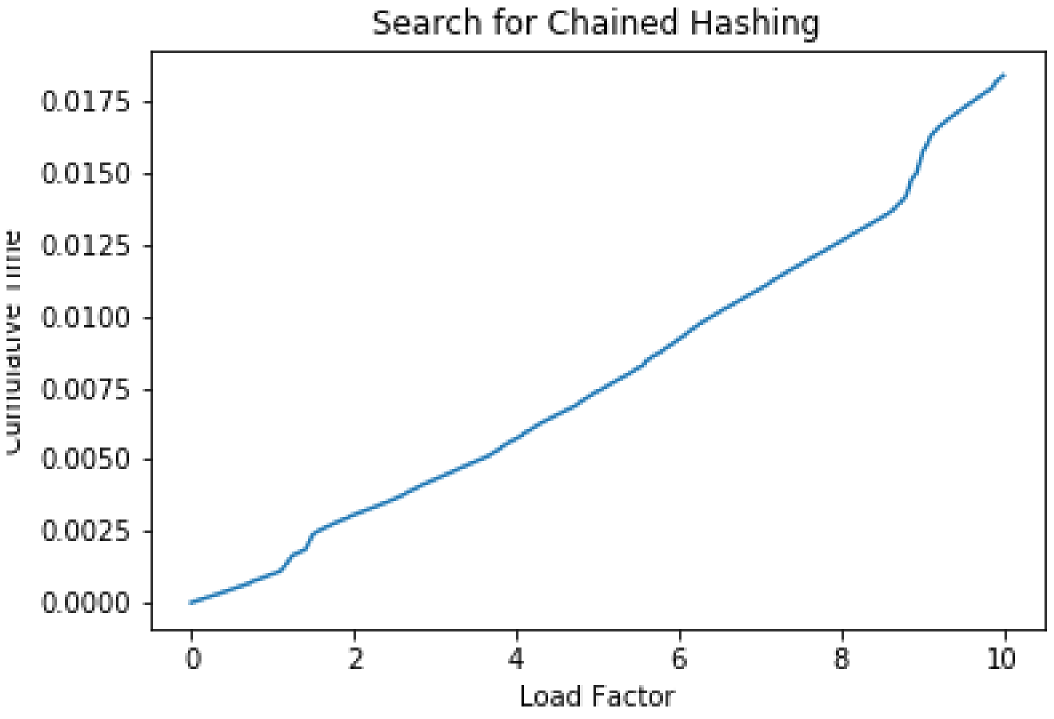
The specific form for each figure is stated in the labels of the axis.

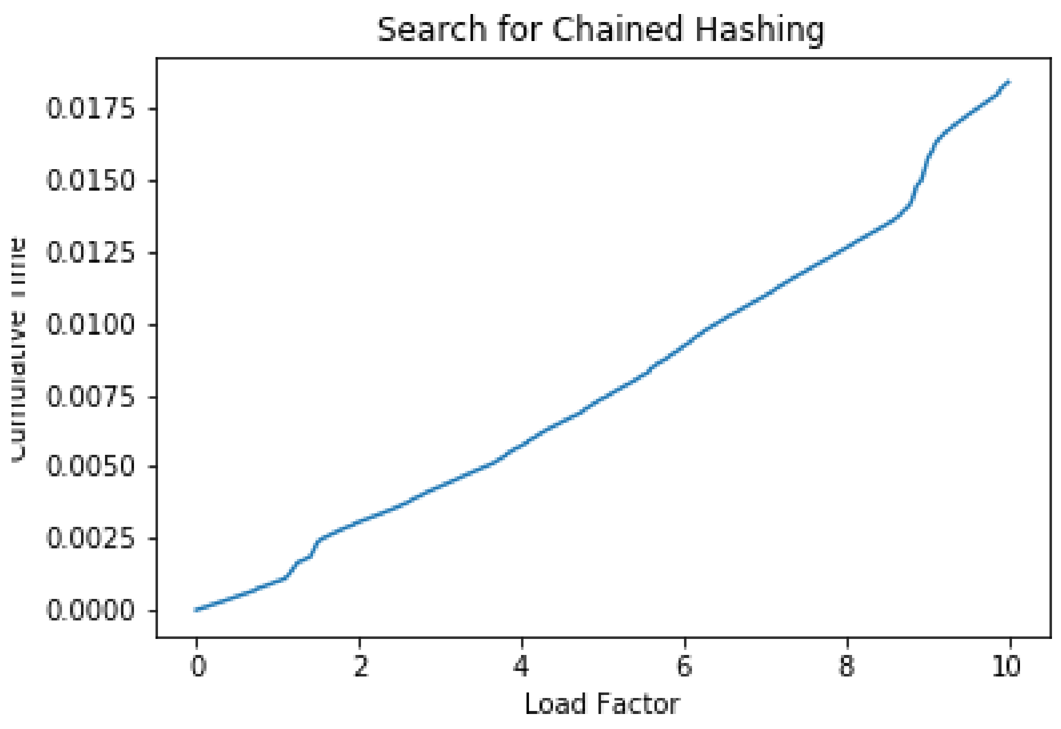
5.1 Insert


Set for Chained Hashing
Cumulative Time
Load Factor
0.00
0.01
0.02
0.03
0.04
0
2
4
6
8
10


Set for Cuckoo Hashing
Logarithm of Cumulative Time
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0.0
0.2
0.4
0.6
0.8
1.0
Load Factor

5.2 Search


Search for Chained Hashing
Cumulative Time
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
2
4
6
8
10
Load Factor



6 Important Observations

6.1 Chained Hashing

Chained Hashing has impressive performance even with a huge load factor.

6.2 Cuckoo Hashing

The main problem with cuckoo hashing is Cycle. It is very important to choose effective hash functions which would distribute the data evenly across all buckets.

Even after attempting to do so, there would be continuous cycles. At that point, my implementation of the algorithm would try to rehash exactly once, and if it does not succeed, abort. This prevents endless loops.

Note that there is a vertical line in the set for Cuckoo Hashing. It means that the load factor did not change, but time is used to set the value. This is caused by rehashing.

6.3 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. Cuckoo hashing is 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. Chained Hashing, on the other hand, is not that sensitive to the load factor.

Operation	Worst Case	Amortized
Chained Hashing - Search	$O(1 + max(len(bucket)))$	$O(1)$
Chained Hashing - Insert	$O(1 + max(len(bucket)))$	$O(1)$
Cuckoo Hashing - Search	$O(1)$	$O(1)$
Cuckoo Hashing - Insert	$O(n)$	$O(1)$

6.4 Time Complexity

7 Conclusion

1. The Big-O notation and amortized time may not always apply From this project, we can see that the run time for insertion and search 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 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. 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.
3. Resizing (including scaling up and down) can be implemented for both hashing algorithms as extension. Resizing might make them perform better, but make it harder to analyze.