CompSci 261P
Data Structures

Project 1: Hashing algorithms

Name: Xingyou Ji
Date: 23 October 2019

Contents

1 Hash Chaining

1.1 Construction

In the method of hash chaining, each cell of the hash table is a linked list, with each ListNode storing a pair of key-value. Therefore, in order to implement the hash algorithm, I first implement a linked list. The supported functions and corresponding implementations as well as the amortized time are shown below.

1.1.1 Linked List Construction

For a ListNode, it has two members: a key-value pair and a pointer pointing to the next node.

The running time for creation is in $O(1)$.

```
from kv import *
class ListNode(object):
    def __init__(self, k, v):
        self.kv = KVPair(k, v)
        self.next = None
```

1.1.2 Linked List Search

To perform a search operation in a linked list, we have to traverse the whole linked list from the head node until a match is found.

The running time is in $O(n)$, where n denotes the length of the linked list.

```
def search(self, key):
    dummy = ListNode(-1, -1)
    curr = dummy
    curr.next = self.head
    while curr.next:
        if curr.next.kv.key == key:
            return curr.next.kv.val
        else:
            curr = curr.next
    return -1
```

1.1.3 Linked List Insert

There could be two situations when performing an insert operation. If the key exists in the linked list, then we should update its corresponding value. Otherwise, insert a new ListNode to the end.

The running time of insert is in $O(n)$, as we need to traverse the whole linked list.

```
def insert(self, key, val):
    dummy = ListNode(-1, -1)
    curr = dummy
    curr.next = self.head
    isExisted = False
    while curr.next:
        if curr.next.kv.key == key:
            curr.next.kv.val = val
            isExisted = True
```

```
                break
            else:
                curr = curr.next
        if not isExisted:
            node = ListNode(key, val)
            curr.next = node
        self.head = dummy.next
        return dummy.next
```

1.1.4 Linked List Remove

To perform a remove operation, we have to traverse the whole linked list, thus causing the running time in $O(n)$.

```
def remove(self, key):
    dummy = ListNode(-1, -1)
    curr = dummy
    curr.next = self.head
    while curr.next:
        if curr.next.kv.key == key:
            curr.next = curr.next.next
            break
        else:
            curr = curr.next
    self.head = dummy.next
    return dummy.next
```

1.1.5 Structures of Hash Chaining

Since the linked list is done, we can define our hash chaining in the following structure.

```
class Chaining(object):
    def __init__(self):
        self.N = 373
        self.alpha = 0
        self.cnt = 0
        self.table = [LinkList() for i in range(self.N)]
```

1.2 Supported Functions

1.2.1 Search

The idea of search is to first calculate the hash value of key, and use linked list search to check whether the key-value pair exists.

The running time of search is in $O(1 + Len(H(k)))$.

```
def search(self, key):
    k = self.hash(key)
    return self.table[k].search(key)
```

1.2.2 Insert

The idea is to first locate the cell, and do a linked list set operation.

The running time of insert is in $O(1 + Len(H(k)))$.

Notice that if too many elements have been added to the hash table, it is likely to have collisions, thus lowering the running speed. To avoid this case, we can calculate the load factor α when doing set, and choose to do rehashing accordingly. It is common and proved to be efficient to limit α within the range $[\frac{1}{4}, \frac{3}{4}]$. This feature is supported in the codes I submit in the zip file, and I would omit the description here to avoid being too tedious.

```
def insert(self, key, val):
    k = self.hash(key)
    isExisted = (self.table[k].search(key) != -1)
    self.table[k].insert(key, val)
    if not isExisted:
        self.cnt += 1
        self.alpha = self.cnt / self.N
        if self.alpha >= 1:
            self.N *= 2
            self.rehash()
```

1.2.3 Remove

The idea is to first locate the cell, and do a linked list set operation.

The running time of remove is in $O(1 + Len(H(k)))$.

```
def remove(self, key):
    k = self.hash(key)
    isExisted = (self.table[k].search(key) != -1)
    self.table[k].remove(key)
    if not isExisted:
        self.cnt -= 1
        self.alpha = self.cnt / self.N
        if self.alpha <= 0.25:
            self.N /= 2
            self.rehash()
```

1.2.4 Rehash

When doing set or delete operation, it is possible that the load factor becomes too large or too small. A large load factor indicates that there will be many collisions and a small load factor indicates that the space efficiency is low. Therefore, in this case, we need to do a rehash, namely, when $\alpha \leq 0.25$ or $\alpha \geq 0.75$.

The running time is in $O(N \cdot \max(Len(H(k))))$.

```
def rehash(self):
    all = []
    self.cnt = 0
    for lst in self.table:
        head = lst.head
        while head:
            kv = head.kv
            all.append(kv)
            head = head.next
    self.table = [LinkList() for i in range(0, self.N)]
    for kv in all:
        self.insert(kv.key, kv.val)
```

1.3 Theoretical Analysis

For hash chaining, each operation will cost a total time in $O(1 + Len(H(k)))$.
To formulate, for a pair p, the expected running time is:

$$\begin{aligned} E[\text{time/operation}] &= O(1 + E[Len(H(k))]) \\ &= O(1 + \Sigma_{q\in S-p}(H(q.k) == H(p.k))) \\ &= O(1 + \frac{n-1}{N}) \\ &= O(1+\alpha) \end{aligned}$$

1.4 Test Cases

1.4.1 Correctness Test

When checking for correctness, I first generated 20 operations of set to store 20 pairs in the hash table.
Then, I did 20 search operations. Within them, some search for non-existed pairs.
Last, I did 10 delete operations. Within them, some delete non-existed pairs, and it will cause no effect.
So far, I have tested the functionality of hash chaining algorithm. But, more efforts should still be put on corner cases, which is repeated pairs and non-existed pairs.

Therefore, I randomly generated a testcase containing 70 operations including Search, Set and Delete, each type containing multiple corner cases. All the tests are passed, indicating that there is no error inside the supported functions.

1.4.2 Efficiency Test

The load factor α is chosen from 0.05 to 0.95 with step length of 0.05. Therefore, a total of 19 records can be obtained, and that will be sufficient to obtain a linear relationship in the log-log plot.

In practice, I generate test cases each containing $(\alpha \cdot N)$ key-value pairs, and do the set operation first, search second and delete last. The randomly generated test cases will be normally distributed on each cell. Therefore, there will not be too many or too few collisions, and the result obtained will be very similar to the theoretical case.

After running tests, I collect α and running time to have a log-log plot. The reason of using log-log scale is that if we can find a linear relationship in a log-log plot, then it is very convenient to transform into a polynomial relationship. In normal sense, a polynomial relationship means high efficiency.

You can view the test cases in project/testcase/insert/. The test result is shown below.

- Insert
  Using the original data without applying a log log trick, the plot is shown below in Fig 1. However, limited information can be obtained from Fig 1.

  In order to have a linear relation, the x-axis is chosen to be $\log(\alpha)$ and the y-axis is chosen to be $\log(T)$, and the linear fitting is shown in Fig 2.

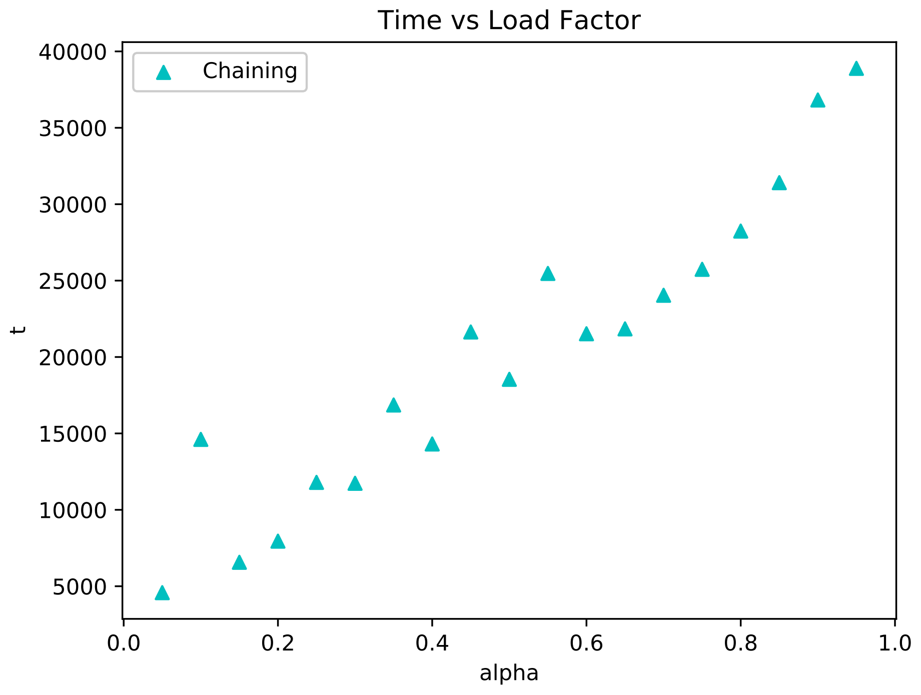

Figure 1: Hashing chaining (Insert).

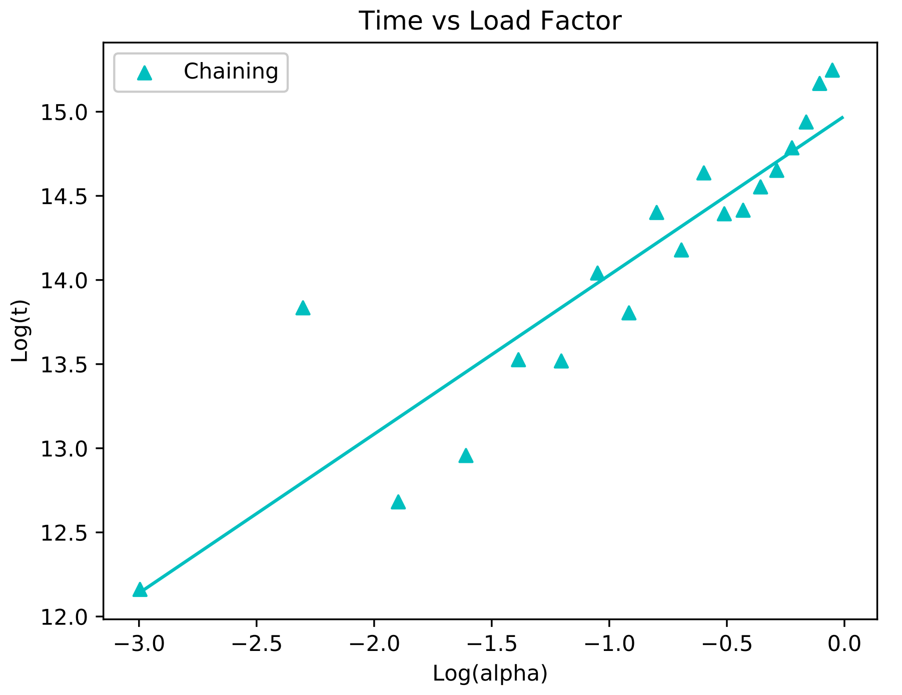

Figure 2: Log-log plot for hashing chaining (Insert).

I use the scipy.optimize.curve_fit function in Python, and the result is:

$$\log(T) = 0.945 \cdot \log(\alpha) + 14.974$$

which can be further simplify to polynomial form:

$$T = \alpha^{0.945} \cdot 2^{14.974}$$

Recall that the expected running time is in $O(1+\alpha)$, which is a first order polynomial of α, the result is close and thus makes sense.

2 Cuckoo Hashing

2.1 Construction

2.1.1 Structures of Cuckoo Hashing

For a cuckoo hashing, it contains two hash tables and two hash functions. In order to prevent infinite loops, I set a limit to detect whether the cuckoo hashing requires rehash.

The structures of the cuckoo hashing class is shown below.

```
class Cuckoo(object):
    def __init__(self):
        self.N = 373
        self.alpha = 0
        self.limit = 300
        self.pos = [nan for i in range(0, 2)]
        self.table = [[KVPair(nan, nan) for i in range(0, self.N)], [KVPair(nan, nan) for j in range(0, self
         .N)]]
    def hash(self, func, key):
        if func == 1:
            return key % self.N
        elif func == 2:
            return floor(key / self.N) % self.N
```

2.2 Supported Functions

2.2.1 Search

When performing a search operation, we need to check two places in two hash tables.

The running time is in $O(1)$.

```
def search(self, key):
    for i in range(0, 2):
        k = self.hash(i + 1, key)
        if self.table[i][k].key == key:
            return self.table[i][k].val
```

2.2.2 Insert

The insert operation in cuckoo hashing is quite different from other hashing methods.

First, we do a search in the two tables to see whether we only need to do an update. If so, update the corresponding value and return.

If the pair to insert is new, then we have to check whether its corresponding position in table 1 is availbale. If so, put the new pair there and return.

If not, put the new pair in its corresponding position in table 1, and do a insert operation for the pair which was in that cell originally. The insert operation for this pair starts to check table two. It is a recursive process, and during each step, we update the counter. Once the counter reaches the limit, it means there is an infinite loop and return.

Since the insert operation only requires to check certain cell in two hash tables, the running time is in $O(1)$.

```
def helper(self, key, val, table_id, cnt):
    if cnt == self.limit:
        self.N *= 2
        self.rehash()
    for i in range(0, 2):
        self.pos[i] = self.hash(i + 1, key)
        if self.table[i][self.pos[i]].key == key:
            self.table[i][self.pos[i]].val = val
            return
    if self.table[table_id][self.pos[table_id]].key is not nan and self.table[table_id][self.pos[table_id]].val
     is not nan:
        kv = self.table[table_id][self.pos[table_id]]
        self.table[table_id][self.pos[table_id]] = KVPair(key, val)
        self.helper(kv.key, kv.val, (table_id + 1) % 2, cnt + 1)
    else:
        self.table[table_id][self.pos[table_id]] = KVPair(key, val)

def insert(self, key, val):
    self.helper(key, val, 0, 0)
```

2.2.3 Remove

The idea of delete is to check two hash tables in order whether the target cell has a pair. If so, delete the pair. In my code implementation, it is to make the cell return to its initial state.

Since it only requires to look up in two hash tables, the running time is in $O(1)$.

```
def remove(self, key):
    for i in range(0, 2):
        k = self.hash(i + 1, key)
        if self.table[i][k].key == key:
            self.table[i][k] = KVPair(nan, nan)
```

2.2.4 Rehash

In the insert function, if we reach the limit of search, it indicates that there are too many collisions now, and we should do a rehash.

The basic idea is to twice the cell number and insert all the elements again. The running time is in $O(N)$.

```
def rehash(self):
    all = []
    for i in range(0, 2):
        for kv in self.table[i]:
            if kv.key is not nan and kv.val is not nan:
                all.append(kv)
    self.table = [[KVPair(nan, nan) for i in range(0, self.N)], [KVPair(nan, nan) for i in range(0, self.N)
        ]]
    for kv in all:
        self.insert(kv.key, kv.val)
```

2.3 Theoretical Analysis

For an arbitrary sequence of operations, it could contain multiple search, insert and delete. As I have included in the previous section, search, insert and delete all have $O(1)$ running time since the basic idea is to look up certain cells in two hash tables.

Suppose a sequence contains n operations, based on the previous analysis, the total running time of cuckoo hashing is in $O(n)$.

Recall the definition of load factor $\alpha := \frac{n}{N}$, hence $n = \alpha N$. Therefore, the running time can be rewritten as $O(\alpha N)$.

2.4 Test Cases

2.4.1 Correctness Test

When checking for correctness, I first generated 50 operations of insert to store 50 pairs in the two hash tables.

Then, I did 25 search operations. Within them, some search for non-existed pairs.

Last, I did 25 delete operations. Within them, some delete non-existed pairs, and it will cause no effect.

So far, I have tested the functionality of cuckoo hashing algorithm. I still need to test corner cases, which is repeated pairs and non-existed pairs.

Therefore, I randomly generated a testcase containing 70 operations including Search, Set and Delete, each type containing multiple corner cases. All the tests are passed, indicating that there is no error inside the supported functions.

2.4.2 Efficiency Test

- Set

  Using the original data without applying a log log trick, the plot is shown below in Fig 1. However, limited information can be obtained from Fig 3.

  In order to have a linear relation, the x-axis is chosen to be $\log(\alpha)$ and the y-axis is chosen to be $\log(T)$. I use the scipy.optimize.curve_fit in Python to get the linear fitting as shown in Fig 4. The result is:

  $$\log(T) = 0.894 \cdot \log(\alpha) + 15.332$$

  which can be further simplify to polynomial form:

  $$T = \alpha^{0.894} \cdot 2^{15.332}$$

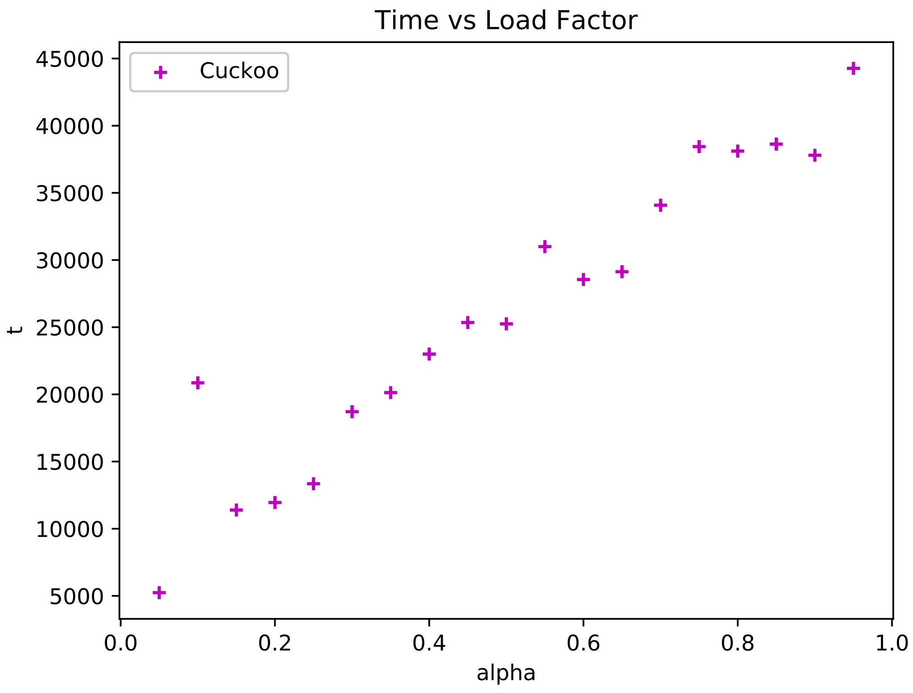

Figure 3: Cuckoo hashing (Insert).

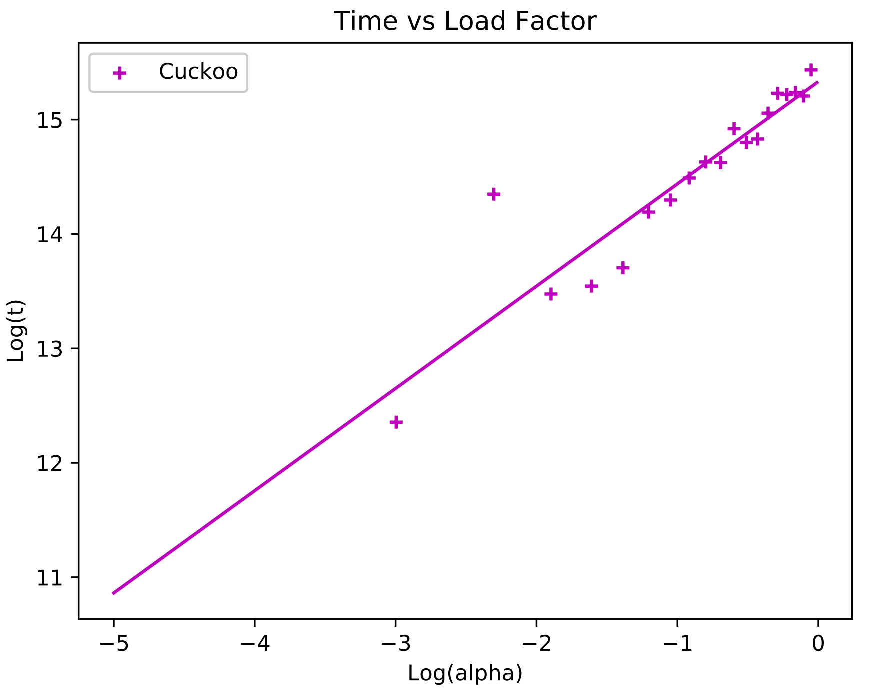

Figure 4: Log-log plot for cuckoo hashing (Insert).

Recall that the expected running time is in $O(\alpha N)$, which is a first order polynomial of α, the result is close and thus makes sense.

3 Comparative Analysis

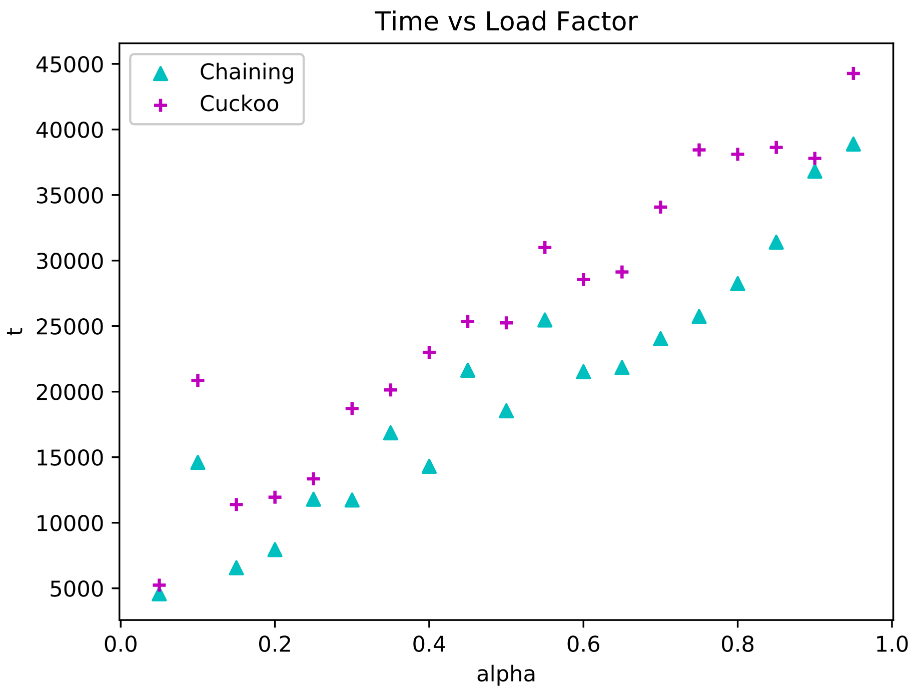

Figure 5: Hashing (Insert).

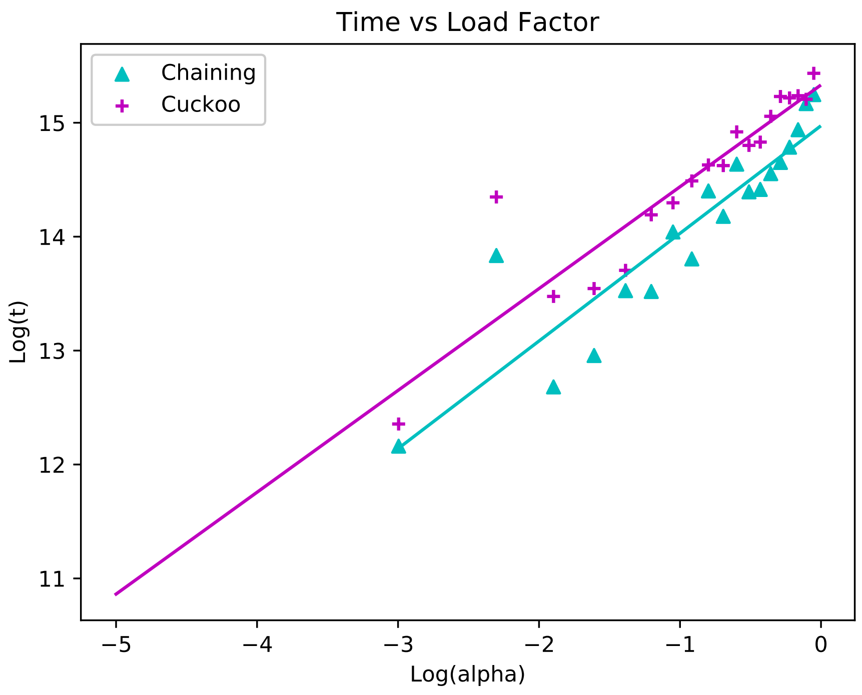

Figure 6: Log-log plot for both hashing (Insert).