Project 1
Zhecheng Li, 35688296

1 Linear Hashing (lazy delete)

1.1 Theoretical predictions (from slides)

Average search time is constant, but it depends on load factor α. Efficiency of linear hashing degrades badly when α gets close to 1.

Expected time is $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ for successful search, while $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ for unsuccessful search.

Typical worst case search time is log n.

1.2 Description of algorithm

1.2.1 Constructor

Allocating an ArrayList with size of capacity of the hash table. Each element in the ArrayList is a Pair(key, value).

1.2.2 Hashing function

Key % capacity

1.2.3 Set(key,value):

```
Index  = hashing(key)
Count = 0
While count < capacity AND element at index is not null AND key of element at index is not null AND key of element at index not equals key:
    Index = (index + 1) % capacity
    Count++
If count is not capacity OR key of element at index equals key:
    Put (key,value) at index
```

1.2.4 Search(key):

```
Index = hashing(key)
Count = 0
While count < capacity AND list at index is not null AND key of element at index is not key:
    Index = (index + 1) % capacity
    Count++
If count < capacity AND element at index is not null:
    Result = value of element at index
```

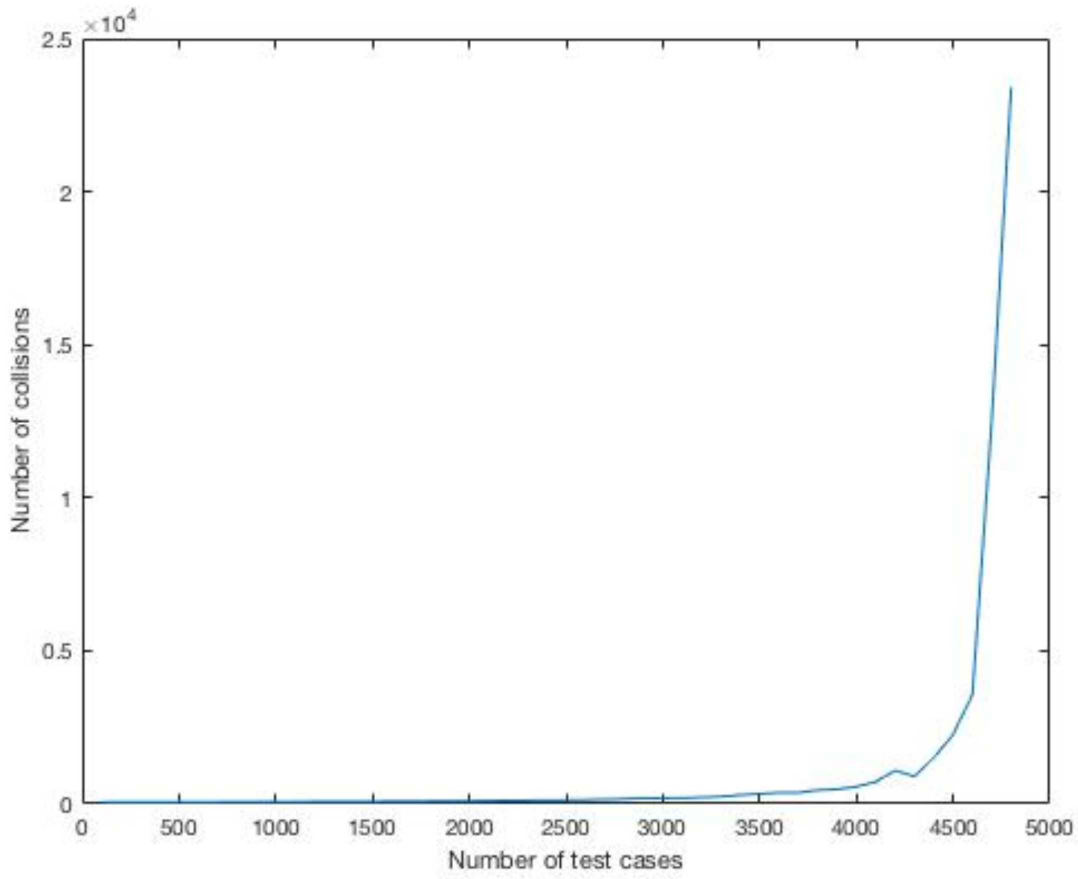
1.2.5 Delete(key)

```
Index = hashing(key)
Count = 0
While count < capacity AND element at index is not null AND (key of element at index is null OR key of element at index not equals key):
    Index = (index + 1) % capacity
    Count++
If count < capacity AND element of index is not null:
    Set key of element at index to null
```

1.3 Test results

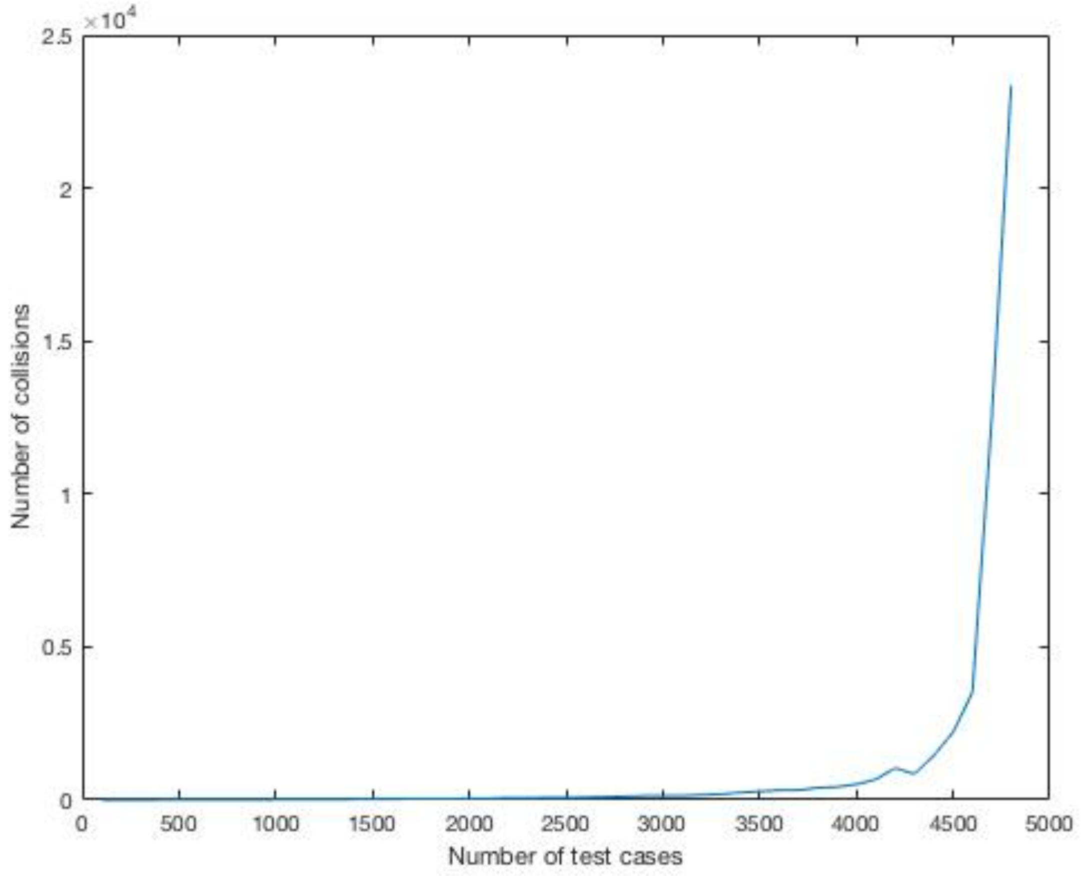
The efficiency of this algorithm is tested with different load factor from 2% to 96 %, 48 data point in total. The capacity of the hash table is 5,000. The collision times is tested every 50 operations (set, get or delete). It is tested for 100 times to get the average.

1.3.1 Set



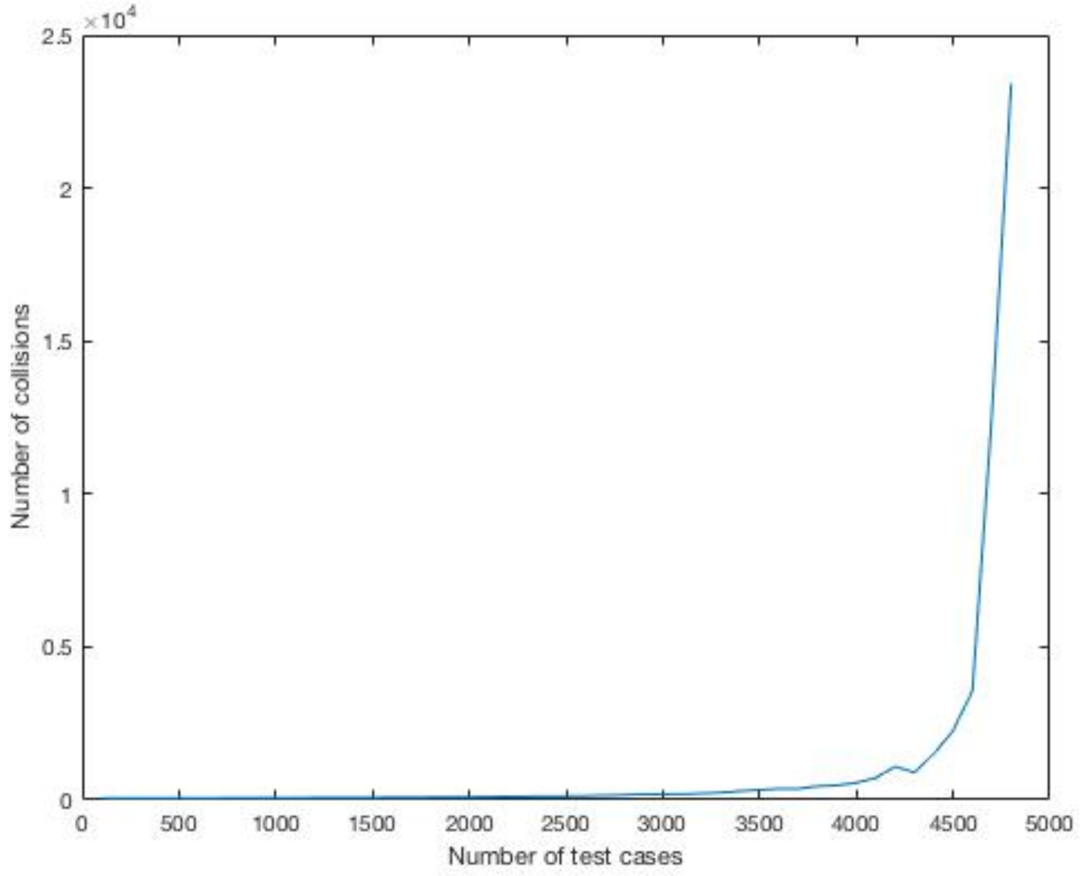
We can see that with more key-value pairs in the hash table, the collision times of set operation increases rapidly because it becomes harder to find an empty plot to put a new key-value pair.

1.3.2 Search



It is similar to plot of set operation. With a nearly full hash table, it is harder to find the target key-value pair.

1.3.3 Delete



It is similar to plot of search operation. With a nearly full hash table, it is harder to find the target key-value pair to delete.

2 Chained Hashing

2.1 Theoretical predictions (from slides)

time/operation = O(1 + length(H[h(k)])

E[time/operation] = $O(1 + \alpha)$

Typical worst case search time is (log n/log (log n))

2.2 Description of algorithm

2.2.1 Constructor

Allocating an ArrayList with size of the capacity of hash table. Each element of the ArrayList is a LinkedList. Each element of a LinkedList is a Pair(key, value).

2.2.2 Hashing function

Key % capacity

2.2.3 Set (key, value)

```
Index = hashing(key)
If LinkedList at index is null:
    New a LinkedList
    Add a Pair(key, value) into the LinkedList
```

```
        Put the LinkedList at index
    Else:
        Foreach pair in chain at index:
            If pair.key equals key:
                Update the value
        If not found pair in chain with same key:
            Add the Pair(key, value) to the LinkedList at index
```

2.2.4 Search (key)

```
    Index = hashing(key)
    If element at index is not null:
        Foreach pair in the LinkedList:
            If key of the pair equals key:
                Return the pair
```

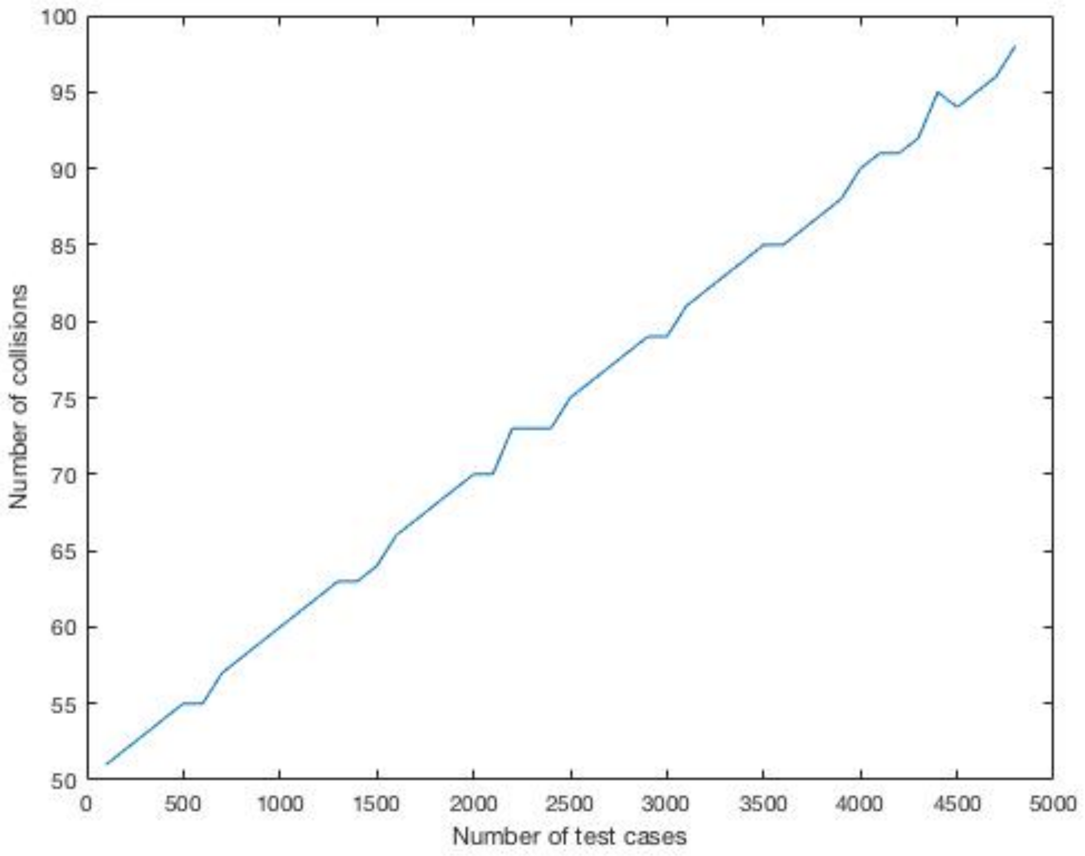
2.2.5 Delete (key)

```
    Index = hashing(key)
    If LinkedList at index is not null:
        Foreach pair in the LinkedList:
            If key of the pair equals key:
                delete the pair
```

2.3 Test results

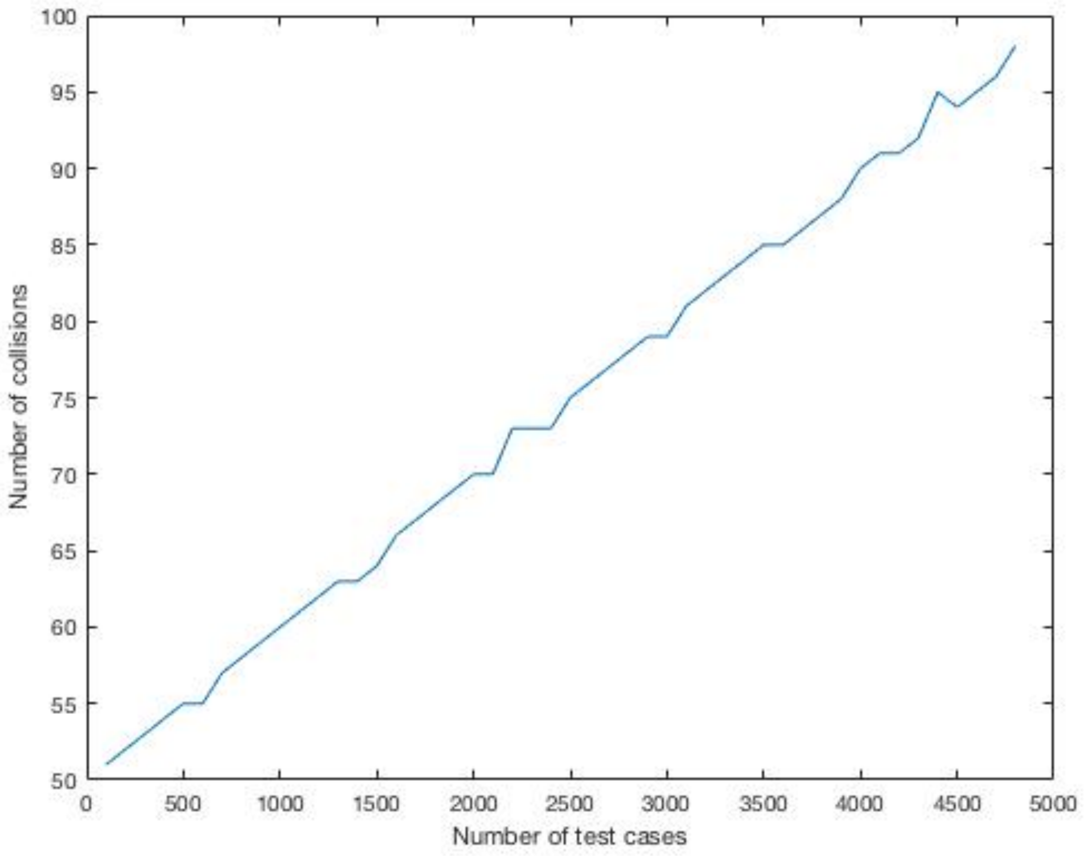
The efficiency of this algorithm is tested with different load factor from 2% to 96 %, 48 data point in total. The capacity of the hash table is 5,000. The collision times is tested every 50 operations (set, get or delete). It is tested for 100 times to get the average.

2.3.1 Set



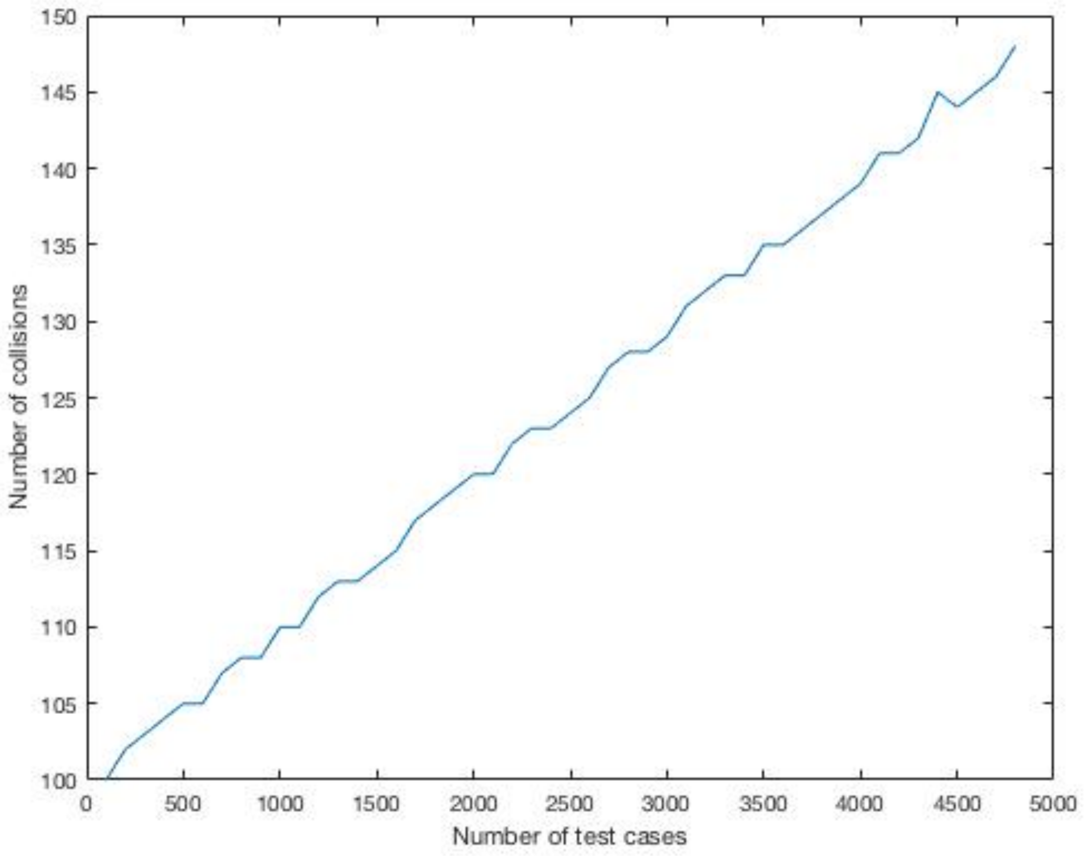
We can see that with more key-value pair in the hash table, there are more collision to put a new pair since we need to check if the pair is already in the table. The collision times is related to α.

2.3.2 Search



The plot is similar to that of set operation. It is harder to find a pair with more pairs in the hash table.

2.3.3 Delete



The plot is similar to that of search operation since we need to find the key then to delete it.

3 Cuckoo Hashing

3.1 Theoretical predictions (from slides)

Guaranteed O(1) search

The expected time for the sequence of operations is O(n)

3.2 Description of algorithm

3.2.1 Constructor

Allocating two ArrayList with size of capacity of the hash table. Elemnt of an ArrayList is a Pair(key, value). Initialize two magicHashNum.

3.2.2 Hashing function

One function: (key + magicHashNum1) % capacity

Another function: ((key + magicHashNum2) / capacity) % capacity

3.2.3 newHashFunc()

randomly choose magicHashNum1

randomly choose magicHashNum2

make sure these two numbers are different

3.2.4 findPlace(toPutPair) function

3.2.5 origin = toPutPair

```
t = 0
count = 0
index = hashing(t, toPutPair.key)
alreadyExist = false
while element of ArrayList t at index is not null AND count < clog(capacity):
    if current element's key equals to origin's key but they are different:
        alreadyExist = true
        break
    exchange current element with toPutPair
    t = 1 – t
    index = hashing(t, toPutPair.key)
    count++

if ArrayList.t.index is null OR alreadyExist is true:
    return current position and toPutPair
else:
    return -1 (to rehash) and toPutPair
```

3.2.6 rehash (toPutPair)

```
backupList = this.list
noConflict = true
newHashCount = 0
do
    noConflict = true
    if newHashCount < 10:
        newHashFunc()
        newHashCount++
    else:
        capacity *= 2
        newHashCount = 0
    allocate a new ArrayList
    put Pair of oldList into new ArrayList, if conflict set nonConflict to false
    put toPutPair into new ArrayList, if conflict set nonConflict to false
while noConflict is false
```

3.2.7 Set(key, value):

```
findPlace(toPutPair)
if need rehash:
    rehash()
else:
    put toPutPair to the empty slot of the ArrayLists
```

3.2.8 Search(key)

```
Try to find the key at position (hashing result) in each ArrayList.
If found:
    Return it
Else:
    Return null
```

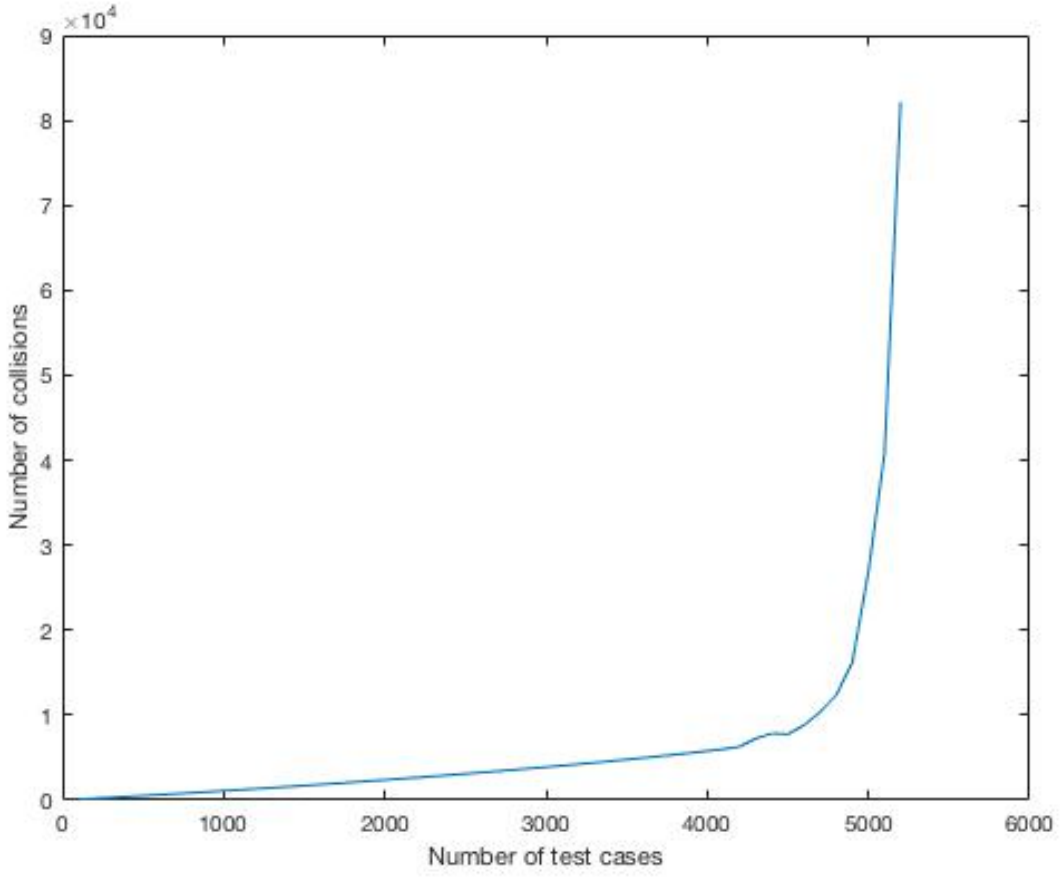
3.2.9 Delete(key)

```
    Try to find the key at position (hashing result) in each ArrayList.
    If found:
        Delete it
```

3.3 Test results

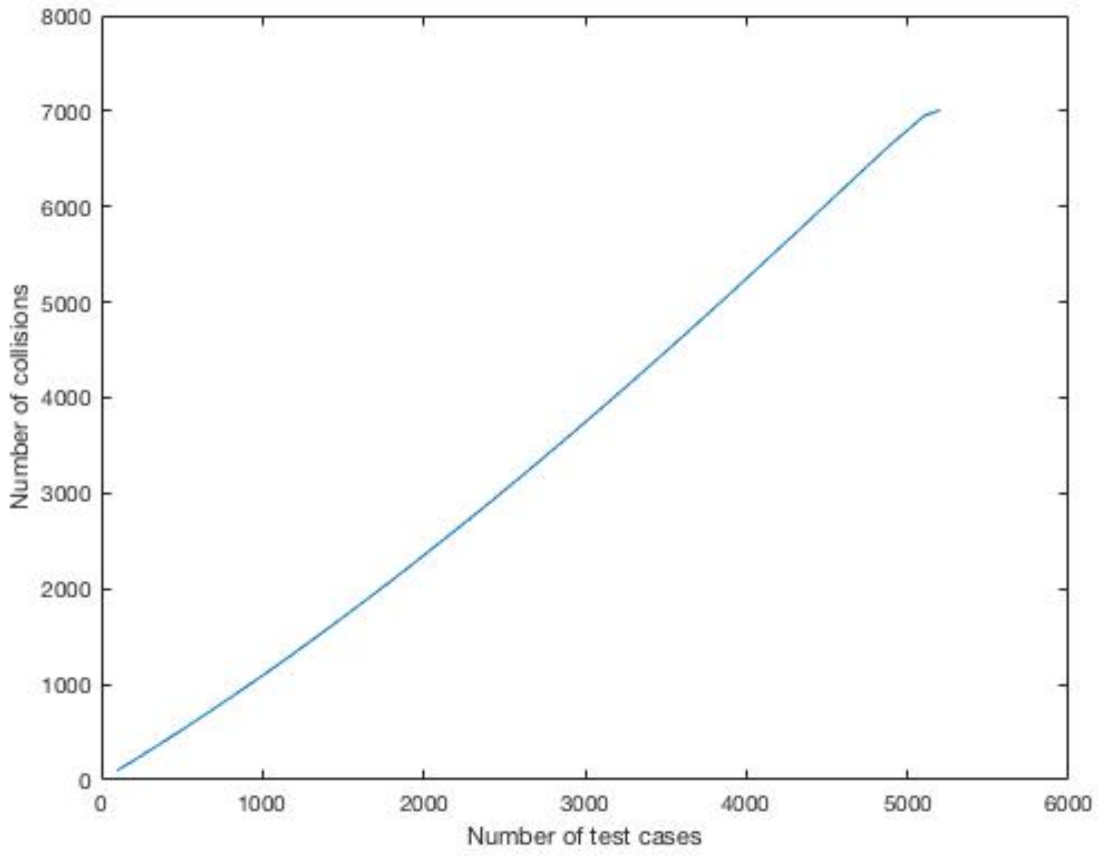
The capacity of the hash table is 5,000. I test sequence collision times of operations (set, get, delete) from 100 to 5,300. I do the test for 200 times to get the average.

3.3.1 Set



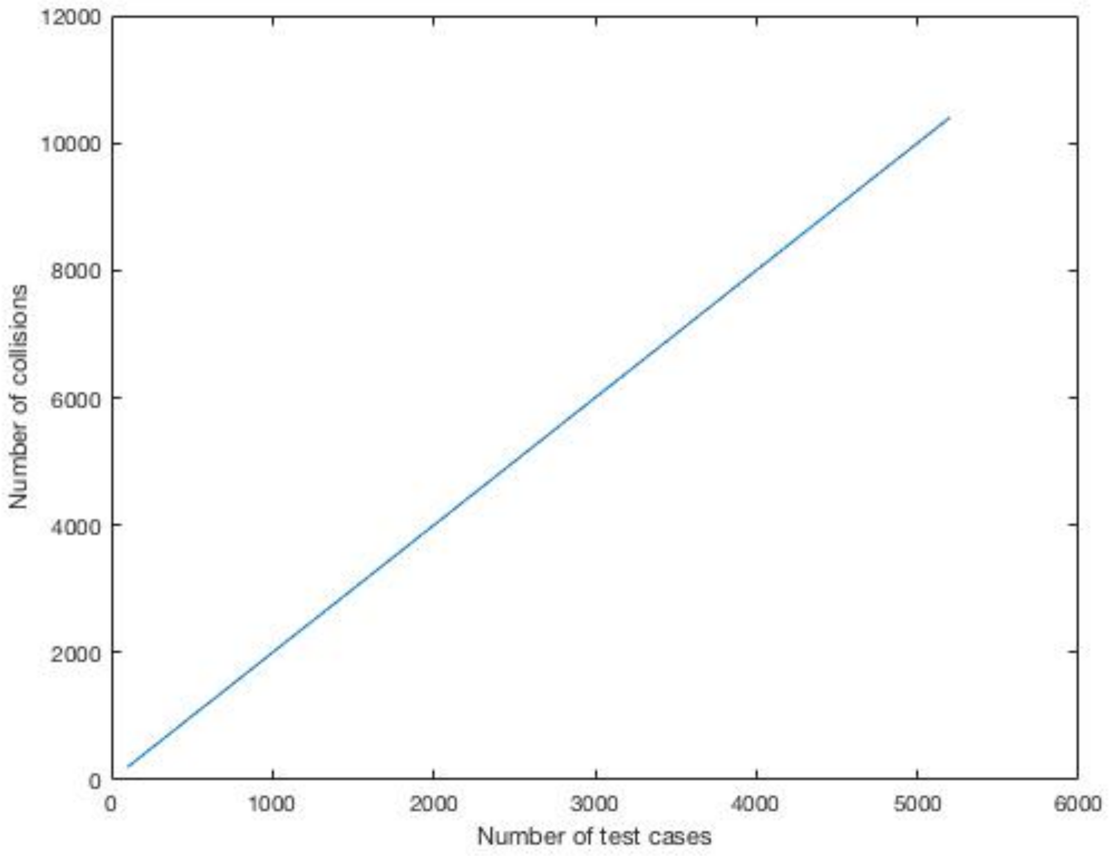
With more key-value pairs in the table, the collision times increases rapidly because it may cause rehashing when while loop is more than clog n times.

3.3.2 Search



The collision times to search is constant because it takes just one or two operations. O(1) complexity for each operation.

3.3.3 Delete



The collision times to delete is constant because it takes just one or two operations. O(1) complexity for each operation.

4 Quadratic Hashing (lazy delete)

4.1 Theoretical predictions

Efficiency is similar to linear probing but is better since the key-value pairs are more uniformly distributed in the table.

4.2 Reason for choosing it

Quadratic probing is better than linear probing since it avoids the clustering problem.

4.3 Description of algorithm

4.3.1 Constructor

Allocating an ArrayList with size of capacity of the hash table. Each element in the ArrayList is a Pair(key, value).

4.3.2 Hashing function

Key % capacity

(hash result should be: key % capacity + i^2, it is completed in each operation function)

4.3.3 Set(key,value):

Count = 1

```
hashVal = hashing(key)
Index  = (hashVal + count * count) % capacity
While count < capacity + 1 AND element at index is not null AND key of element at
index is not null AND key of element at index not equals key:
        Index  = (hashVal + count * count) % capacity
        Count++
If count < capacity + 1 OR key of element at index equals key:
        Put (key,value) at index
```

4.3.4 Search(key):

```
Count = 1
hashVal = hashing(key)
Index = (hashVal + count * count) % capacity
While count < capacity + 1 AND element at index is not null AND (key of element at
index is null OR key of element at index not equals key):
        Index = (hashVal + count * count) % capacity
        Count++
If count < capacity + 1 AND element at index is not null:
        Return value of element at index
```

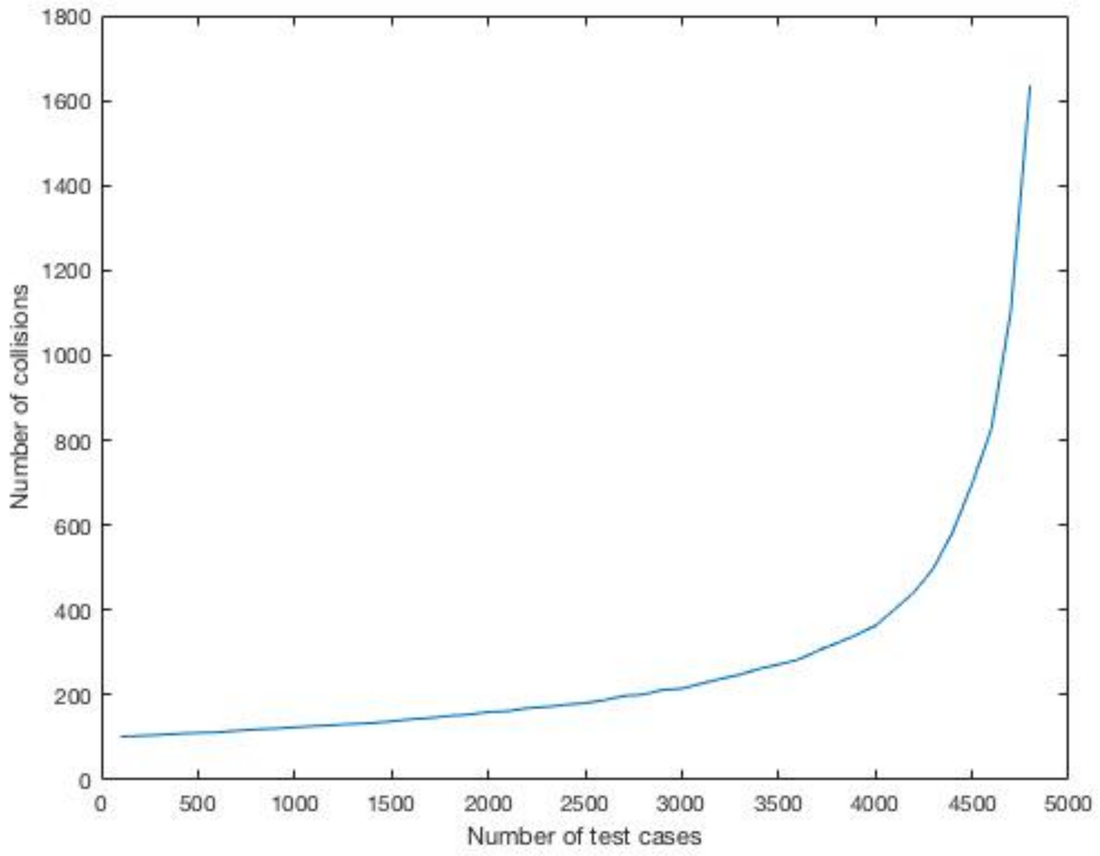
4.3.5 Delete

```
Count = 1
hashVal = hashing(key)
Index = (hashVal + count * count) % capacity
While count < capacity + 1 AND element at index is not null AND (key of element at
index is null OR key of element at index not equals key):
        Index = (hashVal + count * count) % capacity
        Count++
If count < capacity + 1 AND element at index is not null:
        Set key of element at index to null
```

4.4 Test results

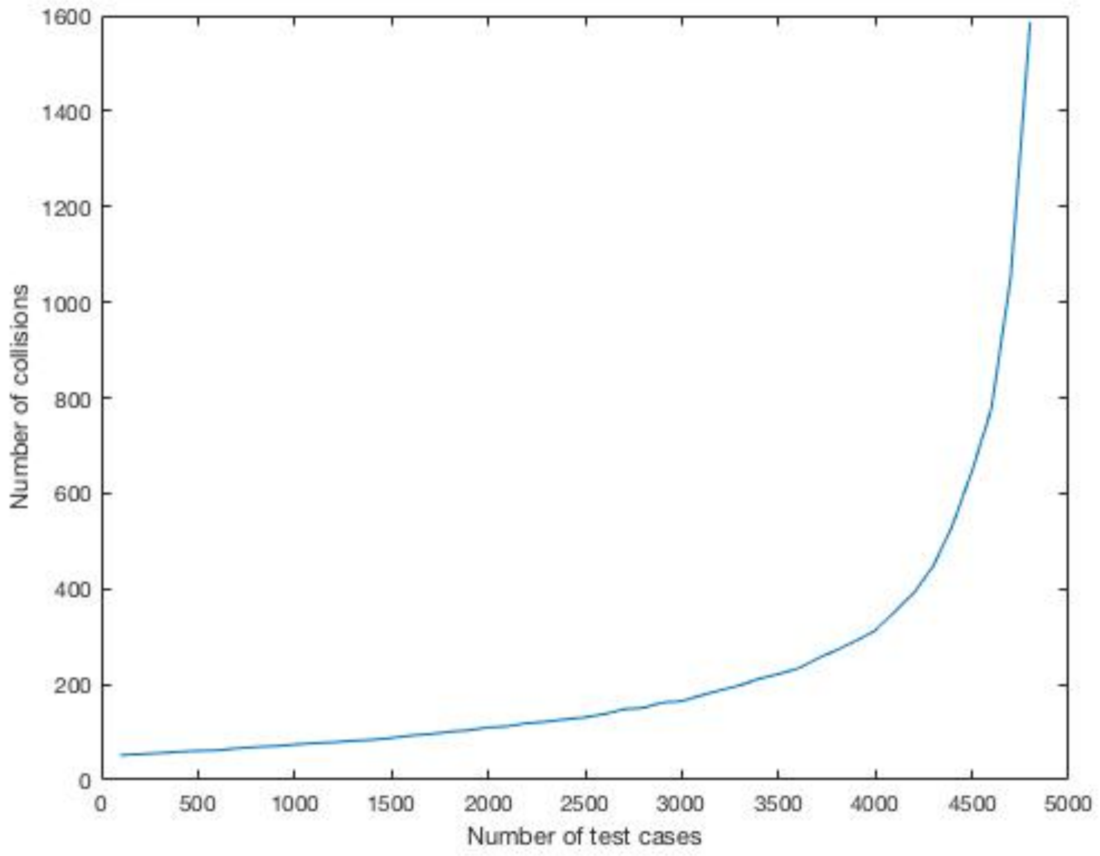
The efficiency of this algorithm is tested with different load factor from 2% to 96 %, 48 data point in total. The capacity of the hash table is 5,000. The collision times is tested every 50 operations (set, get or delete). It is tested for 100 times to get the average.

4.4.1 Set



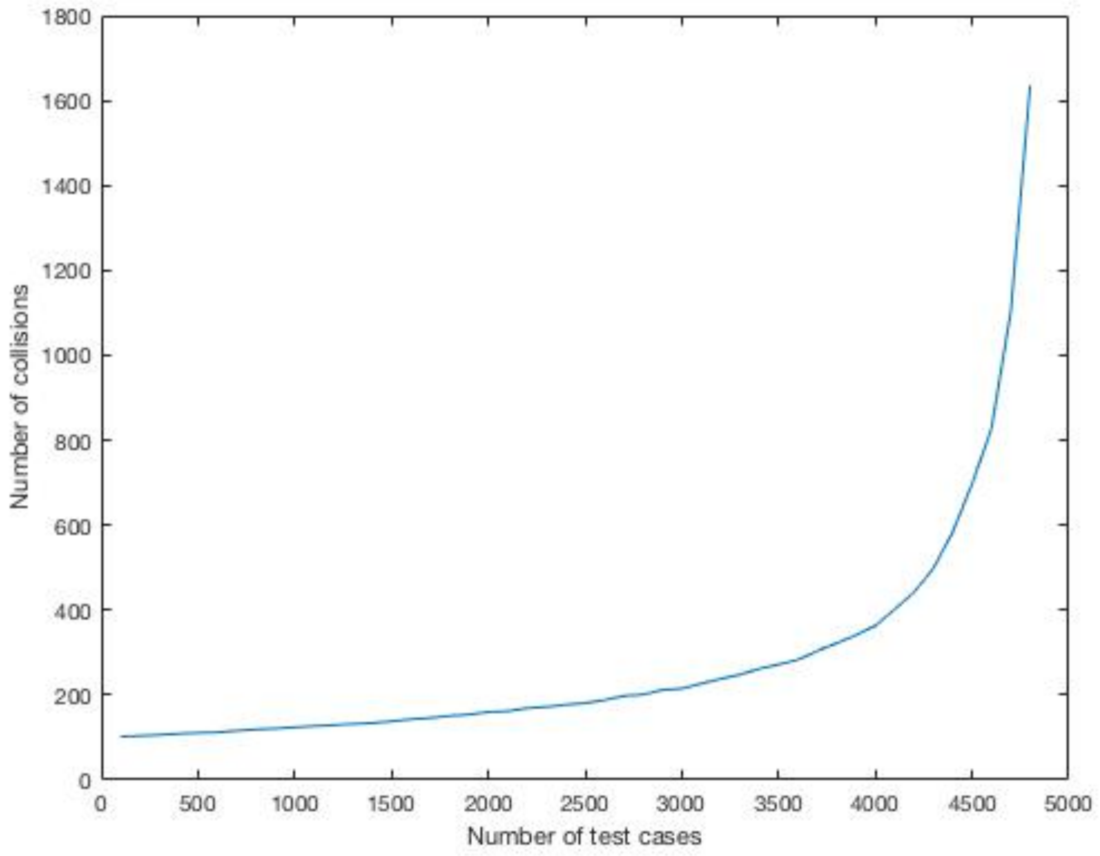
The plot is similar to that of linear probing since they are implemented similarly. With more key-value pairs in the table, the probability of collision increases.

4.4.2 Search



It is similar to plot of set operation. With a nearly full hash table, it is harder to find the target key-value pair.

4.4.3 Delete



It is similar to plot of search operation. With a nearly full hash table, it is harder to find the target key-value pair to delete.