Data Structures - Project 1 (Hashing)

Pushkar Gupta, 13365342

April 2018

1. Analysis Summary In this project, four common hashing techniques were tested using uniform and gaussian key-value random distributions. They are - **linear, chained, cuckoo and double hashing**. To compare the performance of these hashing schemes, worst-case and amortized run time were measured for the following operations - **insert, search and delete**.

These operations were tested as a parameter of the load-factor of the hash table H, which is -

$$\alpha = \frac{\text{elements in H}}{\text{size(H)}}$$

The run time for the three operations, across four hashing schemes was measured while varying α from 0.1 to 0.9 (in increments of 0.1). The following are the specifications of the test-cases (same test cases were used to judge all the hashing schemes):

1) Key-Value pairs were generated by both uniform and gaussian (normal) random selection methods. Keys are 4 character long hexa-decimal numbers (so, the universe size is 2^{16}). The values are integers lying between 1 and 100.

2) The hash table size is $2^{10} = 1024$. The hash-table was never resized so that performance could be measured as a function of the load-factor.

3) After achieving a load-factor for which the performance has to be measured, worst-case and amortized time for another 20 operations was measured. This run-time (for 20 operations) has been plotted and compared for the various hashing schemes. For worst-case, simply the operation that took the highest time among the 20 was noted. For amortized time (over a sequence operations), the total time for the sequence was noted and then divided by 20.

Hashing Algorithm The following hashing-algorithm was used for all the hashing techniques:

$$h_a(x) = \frac{(ax)\%(U)}{U/M}$$

where U is the universe-size (2^{16} in our case), M is the table-size (2^{10}), and a is a random-odd number less than M. For this hashing algorithm the probability of a collision (for $x \neq y$), i.e.

$$Pr[h(x) = h(y)] \leq \frac{2}{M}$$

Source: Section 4.7 `https://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/slidesS14/hashing.pdf`

This hashing algorithm can be used to generate multiple hash functions (by varying a), as required in cuckoo and double hashing. It also has low probability of collision.

2. Hashing Schemes

2.1 Linear Hashing In linear hashing, every cell stores only one key-value pair and collisions are resolved using linear probing. Suppose there is a collision for a key **k**, then the following locations are searched:

$$H[h(k)+1], H[h(k)+2], H[h(k)+3], ...$$

Upto size(H), and all these positions are taken with mod of table-size, so that the search wraps around in H. In the analysis for this hashing scheme, the hashing algorithm defined in section 1 was used.

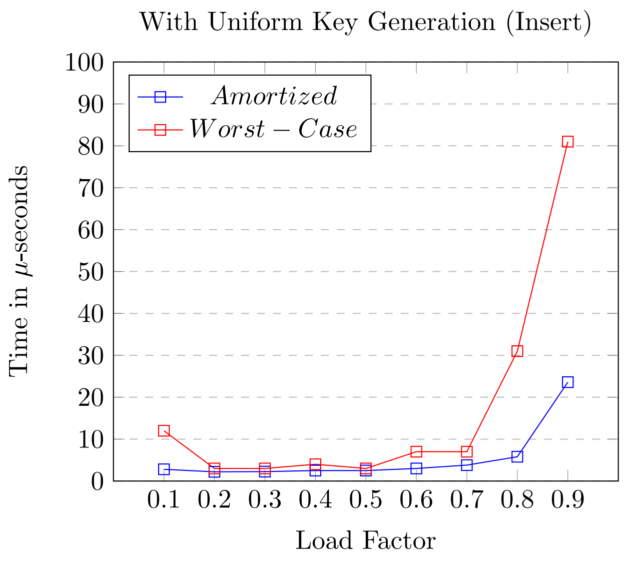
Theoretical Analysis As per the lecture slides, the typical worst-case search time for search/ delete and insert operations grows by $log(n)$, for a fixed table size. Thus theoretically the time for operations should grow by $log(\alpha)$, where α is the load-factor.

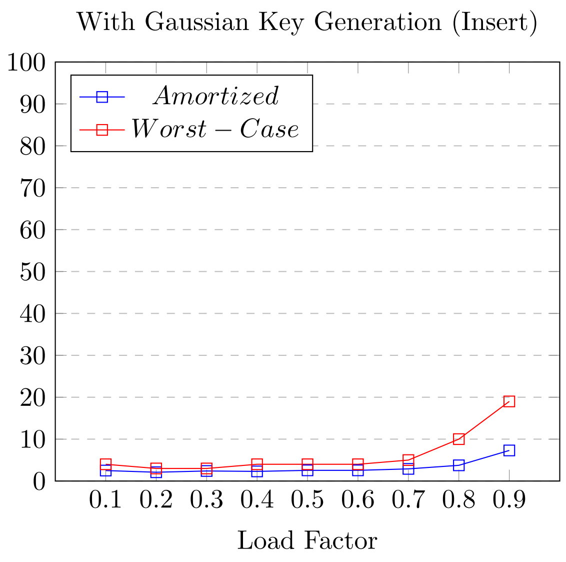
2.1.1 Insert The pseudo-code for the operation is:

```
def insert(kev, value):
    for i in range(0, size(H)):
        index = (h(key) + i) mod size(H)
        if H[index] is empty or H[index][0] == key: # cell found
            H[index] = (key, value) # insert/ update
            return True

    return False  # no more capacity
```

2.1.1.a Experimental run time



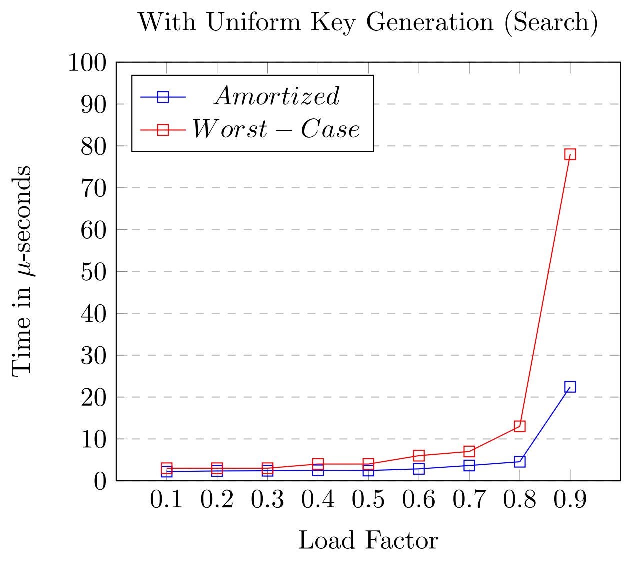


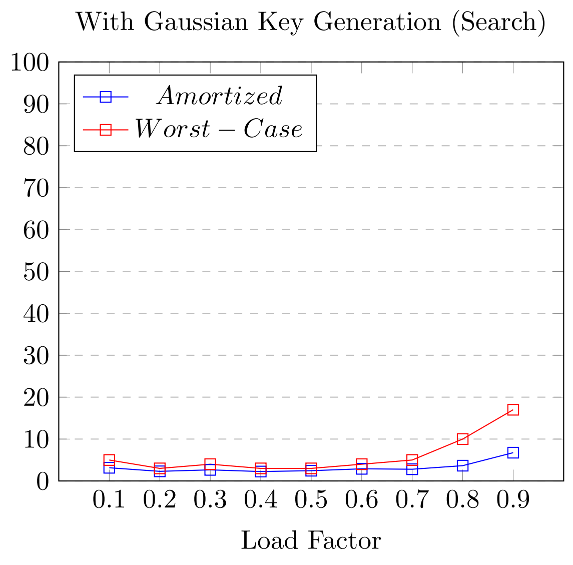
2.1.2 Search The pseudo-code for the operation is:

```
def search(key):
    for i in range(0, size(H)):
        index = (h(key) + i) mod size(H)
        if H[index] is empty:  # empty cell
            return False
        elif H[index][0] == key:  # cell found
            return (key, H[index][1])

    return False  # entire array searched
```

2.1.2.a Experimental run time





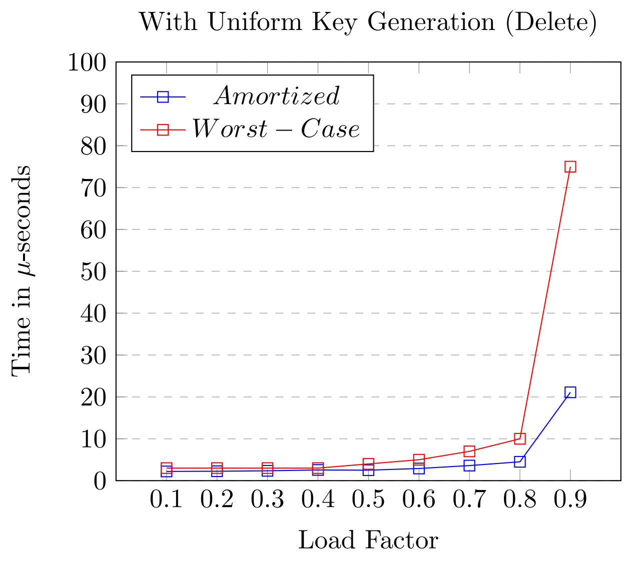
2.1.3 Delete Eager delete was implemented, the pseudo-code for the operation is (implemented exactly as in lecture slides):

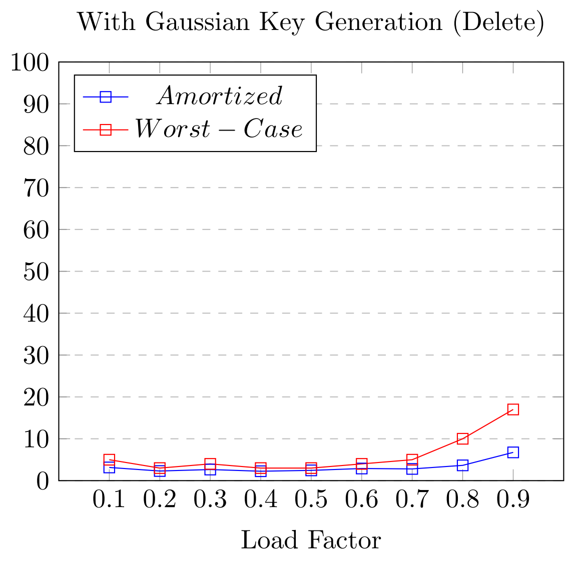
```
def delete(key):
    i = h(k)
    while H[i] is nonempty and contains a key != k
        i = (i + 1) mod size(H)
    if H[i] is empty: exception

    j = (i + 1) mod N
    while H[j] is nonempty:
        if h(H[j].key) is not in the (circular) range [i+1..j]:
            H[i] = H[j]
            i = j
        j = j + 1

    clear H[i]
    return True
```

2.1.3.a Experimental run time





2.1.4 Observation One can very clearly observe that the experimental run time for all the operations is very similar to the theoretical run-time, which is O($log(\alpha)$).

2.2 Chained Hashing In chained hashing, every cell stores a pointer to a linked-list, which can further store any number of key-value pairs. Thus, whenever there is a collision, the new key-value pair is appended to the linked last at that cell. In the analysis for this hashing scheme, the hashing algorithm defined in section 1 was used. The empirical and theoretical analysis for insert, search and delete operations is as follows:

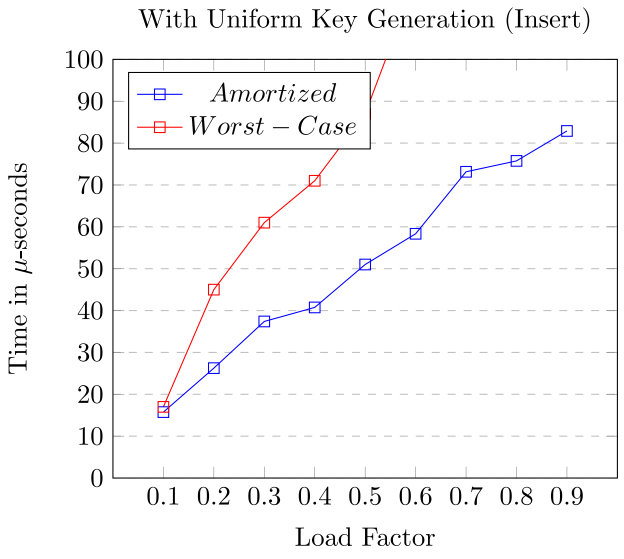
Theoretical Analysis As per the lecture slides, the expected run-time for all the operations in chained hashing is O(1+α). Hence, we should expect run-time plots to look like O(n) curves.

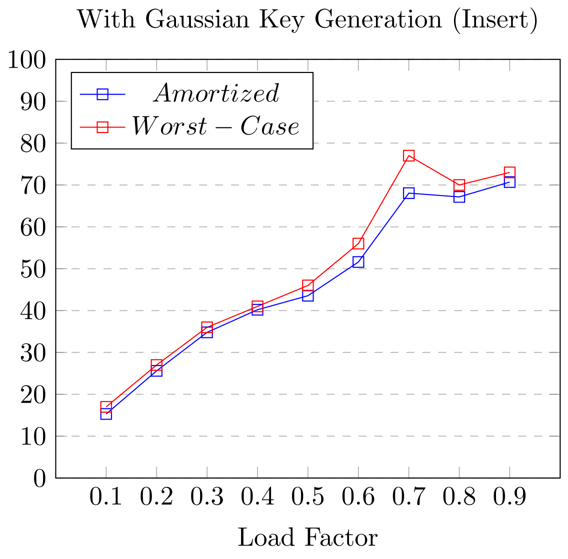
2.2.1 Insert The pseudo-code for the operation is:

```
def insert(key, value):
    linked_list = H[h(key)]
    for i in [0,1,..., len(linked_list)]:
        if linked_list[i][0] == key: linked_list[i] = (key, value)  #update
        return True

    linked_list.append((key, value))  # add new key-value tuple
    return True
```

2.2.1.a Experimental Run Time



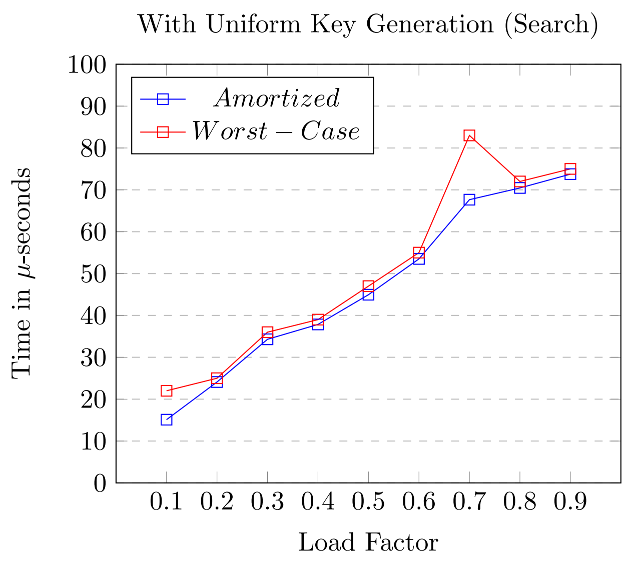


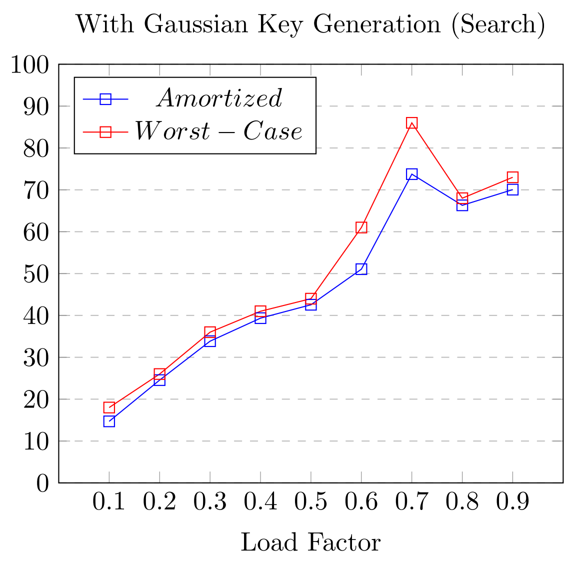
2.2.2 Search The pseudo-code for the operation is:

```
def search(key):
    linked_list = H[h(key)]
    for i in [0, 1, .... len(linked_list)]:
        if linked_list[i][0] == key:
            return (linked_list[i])  # return the key,value tuple

    return False  # not found
```

2.2.2.a Experimental Run Time



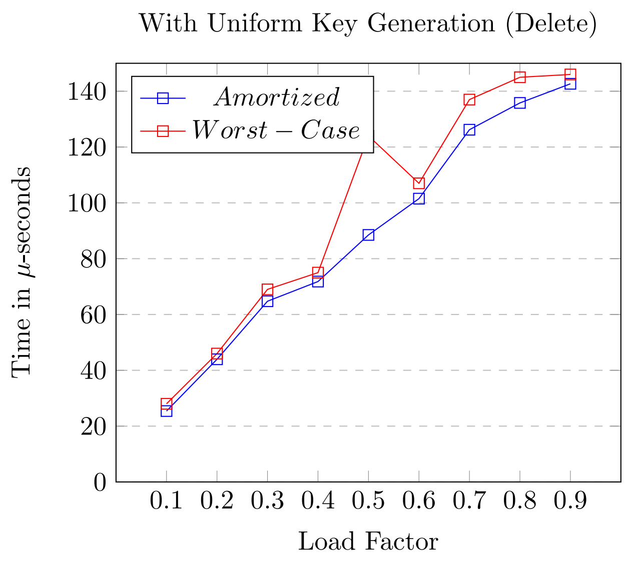


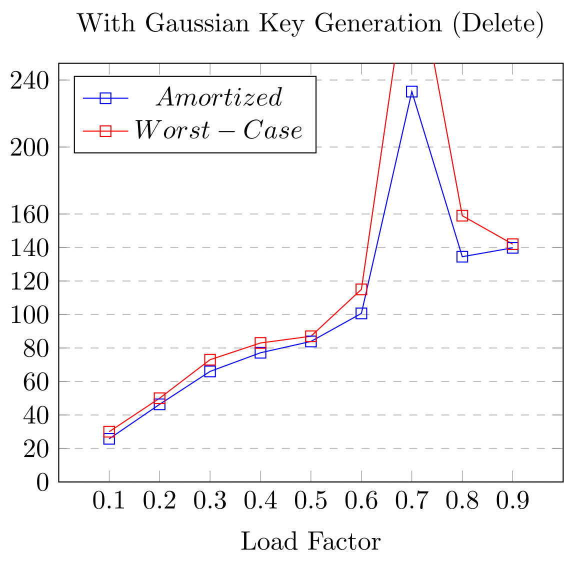
2.2.3 Delete The pseudo-code is as follows:

```
def delete(key):
    linked_list = H[h(key)]
    for i in [0, 1, ...., len(linked_list)]:
        if linked_list[i][0] == key:
            linked_list[i] = None  # mark as empty
            return True

    return False  # no element found
```

2.2.3.a Experimental run time





2.2.4 One can very clearly observe that the experimental run-time for all the operations is in accordance with the theoretical run-time of O(1 + α).

2.3 Cuckoo Hashing In Cuckoo hashing, two tables H_1 and H_2 are maintained. Each table has its own independent hashing function. While inserting a key, the key to be inserted in a table, displaces a key already in that location (if present). Then the displaced key is inserted in the other table (and the loop goes on..). A lot of these displacement operations indicates that we have to rehash all the data, using new hash functions. Search and delete occur in constant time.

Theretical Analysis Search and delete operations always occur in O(1) time. The insert amortized-time as per the lecture slides is O(n), which implies O(α) for a fixed table size.

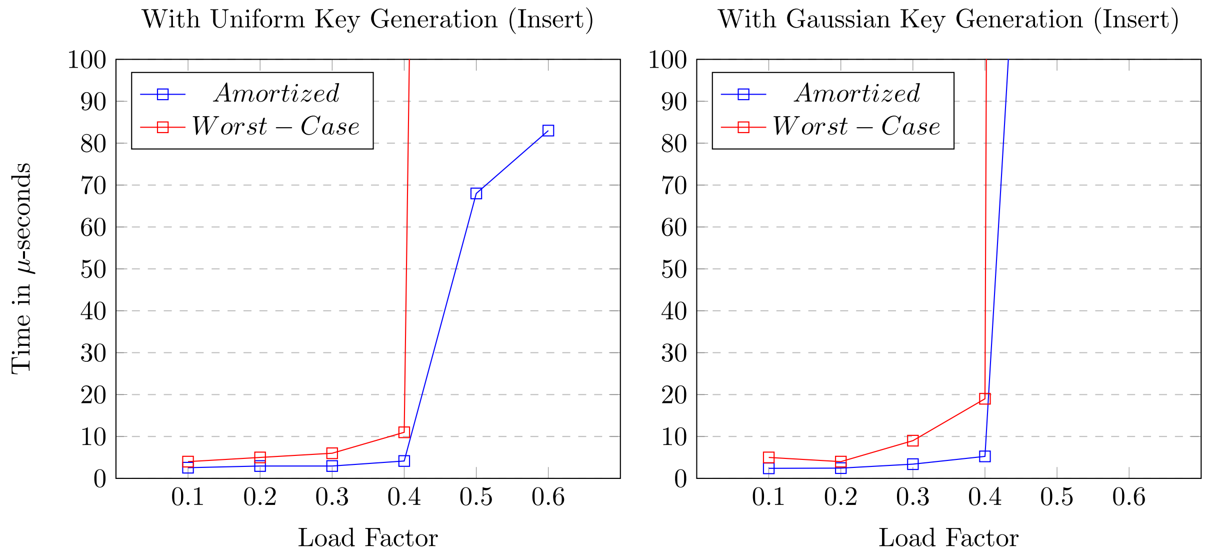
NOTE: Cuckoo hashing could only be tested till load-factor = 0.6, because after that point, no two hash functions were able to successfully rehash all the data (as per the constraints of the hashing algorithm). Also, as per the test-case specifications, the tables are never resized (so as to measure performance as a parameter of load-factor).

2.3.1 Insert The pseudo-code for the operation is:

```
def insert(k, v):
    t = 0
    ($k_i$, $v_i$) = (k, v)
    do:
        if (k, v) is empty: return True
        swap (k, v) and $H_t[h_t(k)]$
        t = 1-t
    while: not ((k, v) == ($k_i$, $v_i$) and t==0)

    # loop found! so, rehash
    rehash($H_1$, $H_2$)
    return insert(k, v)
```

2.3.1.a Experimental Run Time



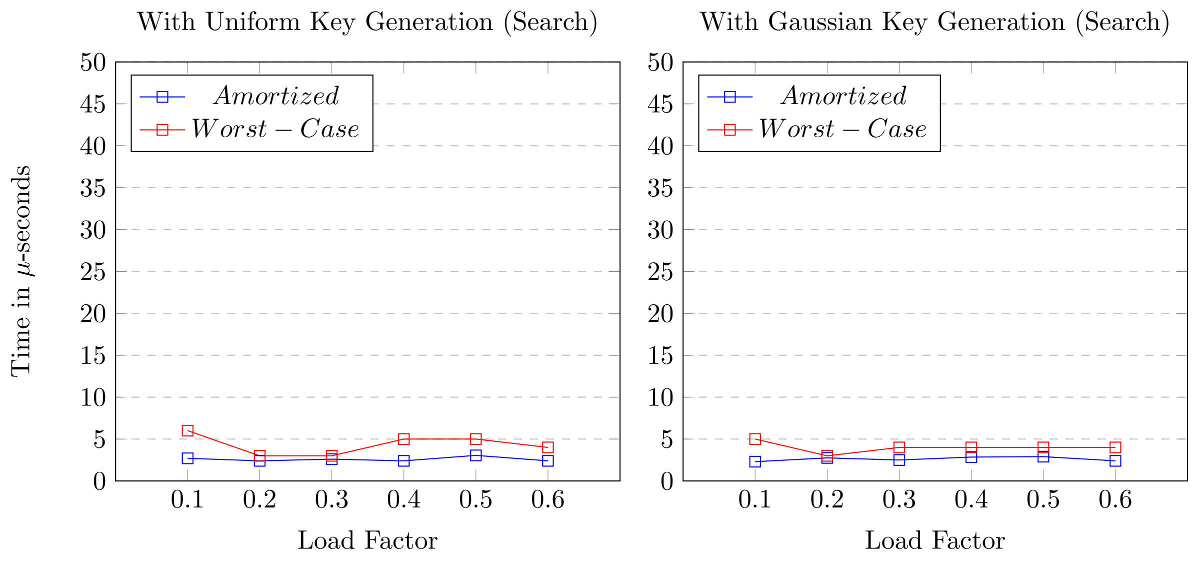
2.3.2 Search The pseudo-code for the operation is:

```
def search(key):
    if $H_0[h_0(key)]$[0] == key: # search in 1st store
        return $H_0[h_0(key)]$ # a key,val tuple

    if $H_1[h_1(key)]$[0] == key: # search in 2nd store
        return $H_1[h_1(key)]$ # a key,val tuple

    return False  # not found
```

2.3.2.a Experimental run time



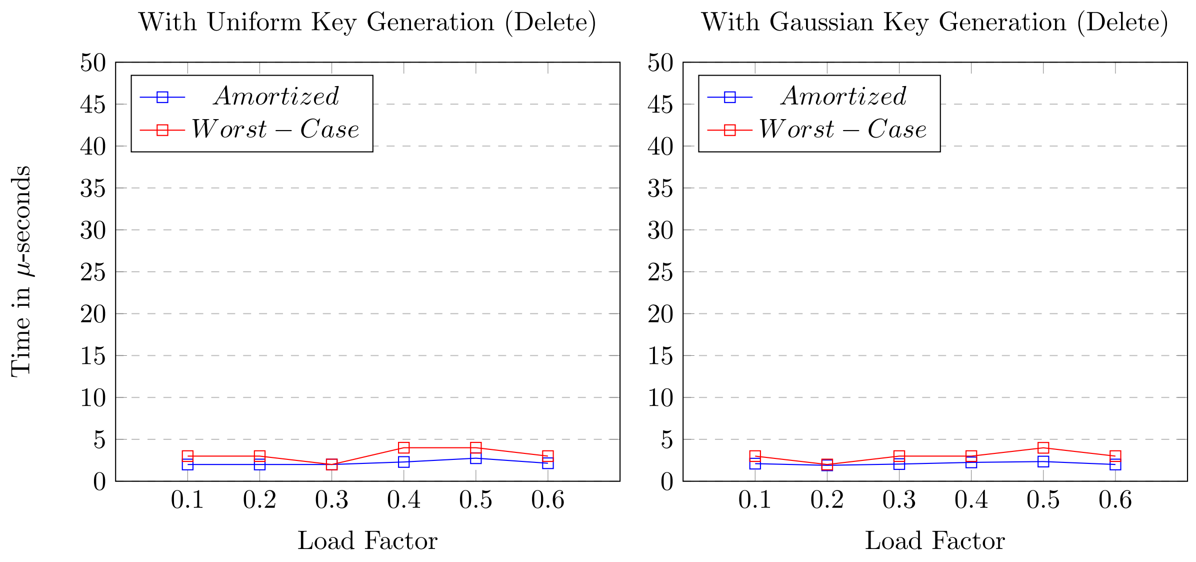
2.3.3 Delete The pseudo-code for the operation is:

```
def delete(key):
    if $H_0[h_0(key)]$[0] == key: # search in 1st store
        $H_0[h_0(key)]$ = None
        return True

    if $H_1[h_1(key)]$[0] == key: # search in 2nd store
        $H_1[h_1(key)]$ = None
        return True

    return False  # not found
```

2.3.3.a Experimental run time



2.3.4 The experimental search and delete time are constant as expected theoretically. The insert time grows with the load-factor with O(α)). After load-factor = 0.6, the number of rehashes grows to a large number, and there is no successful insert. Cuckoo hashing could only be used till that point, given the constraints of the hashing function (as per Section 1).

2.4 Double Hashing In double hashing, collisions are resolved using another independent hash-function h2. Whenever there is a collision at location H[h_1(k)], the following locations are checked (for all insert/ search/ delete):

$$H[h_1(k) + 1 * h_2(k)], H[h_1(k) + 2 * h_2(k)], H[h_1(k) + 3 * h_2(k)], ...$$

These successive locations are checked upto a limit (that is pre-determined) and after taking mod with table-size, so that the search "wraps" around. After this limit, the entire table is rehashed, and two new hash functions h_1 and h_2 are used, like in Cuckoo hashing. This is done so that the time for search and delete remains constant (we will only check upto "limit" positions). The insert-time could increase in spikes due to the rehash table operation.

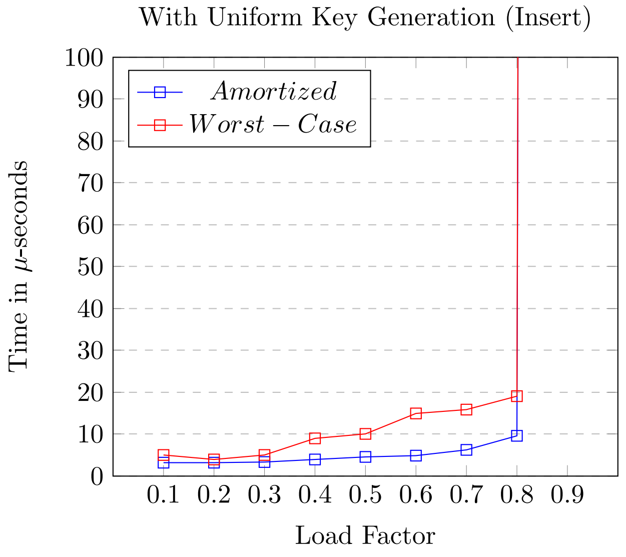
Theoretical Analysis Search and delete time will always be constant (as we only check a specific number of successive locations). Insert time is constant, but if we include the time for rehashing, it shoudl grow by O(n) or O(α), for a fixed-table size.

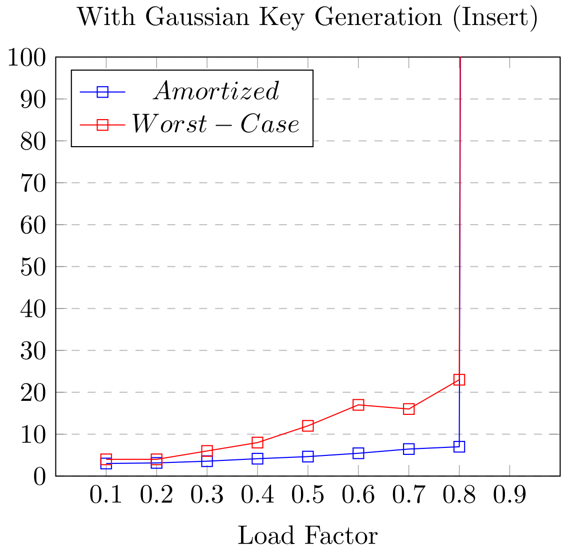
2.4.1 Insert The pseudo-code for the operation is:

```
def insert(key, value):
for i in [0, 1, ...., limit]:
    index = $h_1$(key) + i*$h_2$(key)
    if H[index] is empty or has same key:
        H[index] = (key, value)
        return True

# no space
rehash(H)
return insert(key, value)
```

2.4.1.a Experimental run time



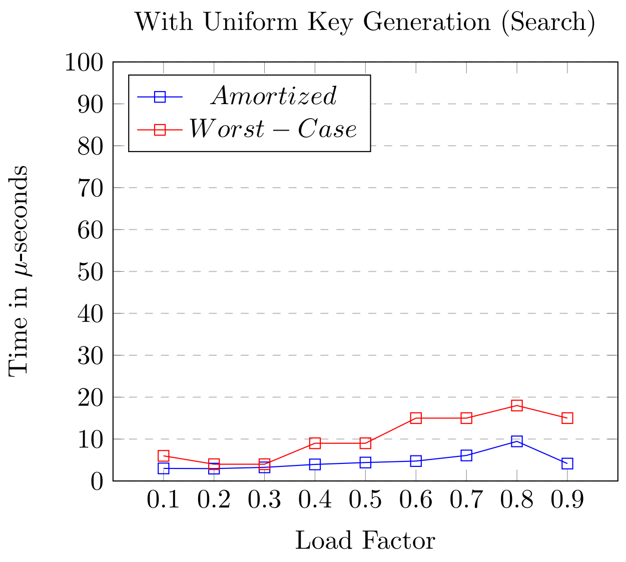


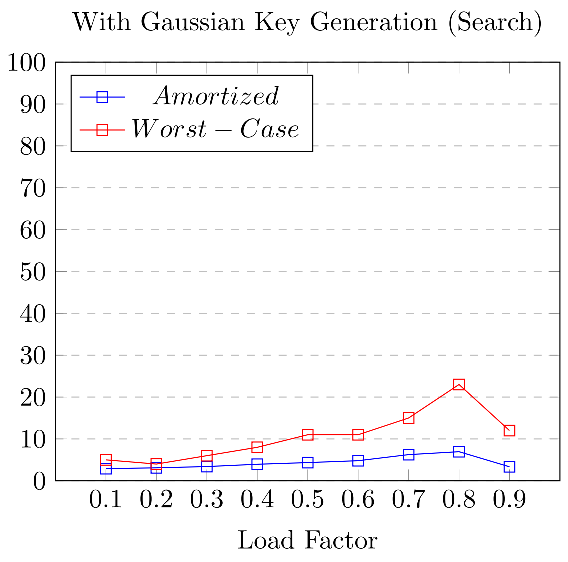
2.4.2 Search The pseudo-code for the operation is:

```
def search(key):
for i in [0, 1, ...., limit]:
    index = $h_1$(key) + i*$h_2$(key)
    if H[index][0] == key:
        return H[index]  # a key,value tuple

return False  # not found
```

2.4.2.a Experimental run time



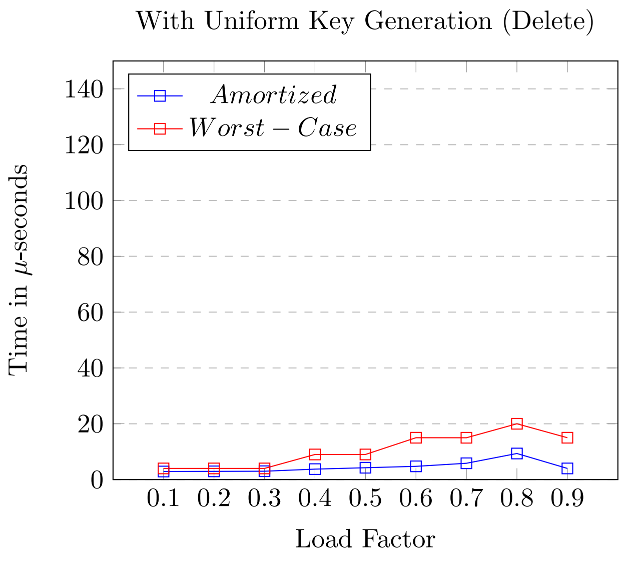


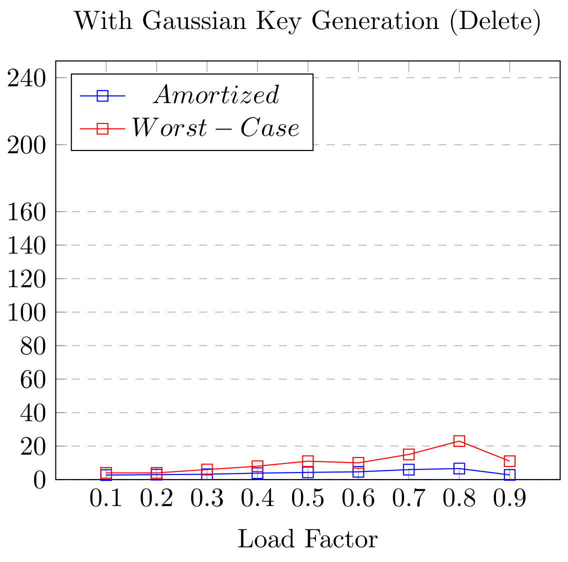
2.4.3 Delete The pseudo-code for the operation is:

```
def delete(key):
for i in [0, 1, ...., limit]:
    index = $h_1$(key) + i*$h_2$(key)
    if H[index][0] == key:
        H[index] = None  # mark as empty
        return True

return False  # not found
```

2.4.3.a Experimental run time

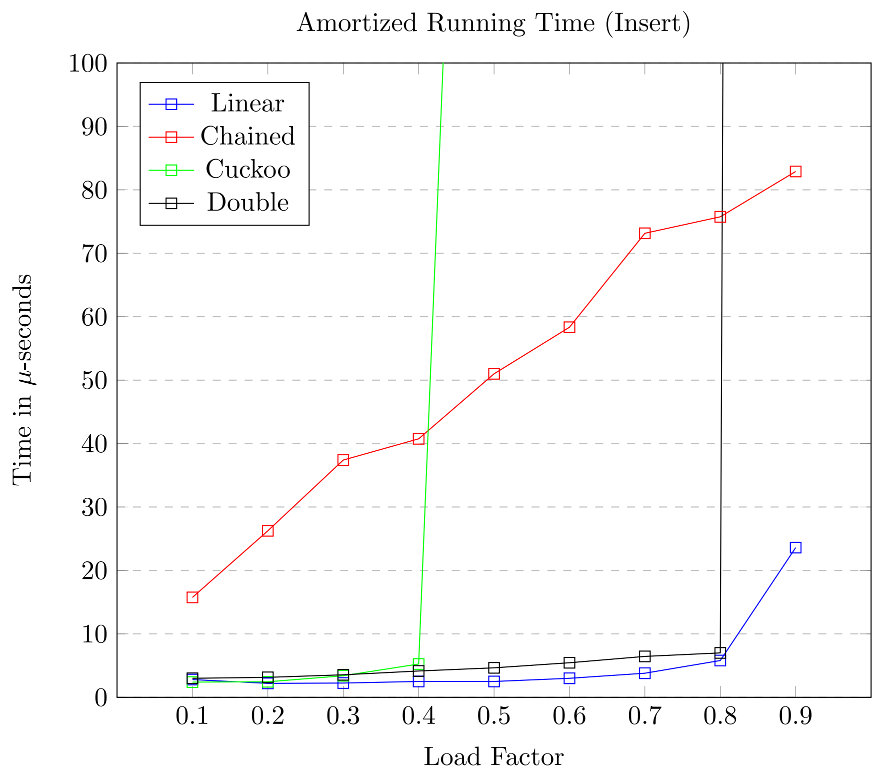




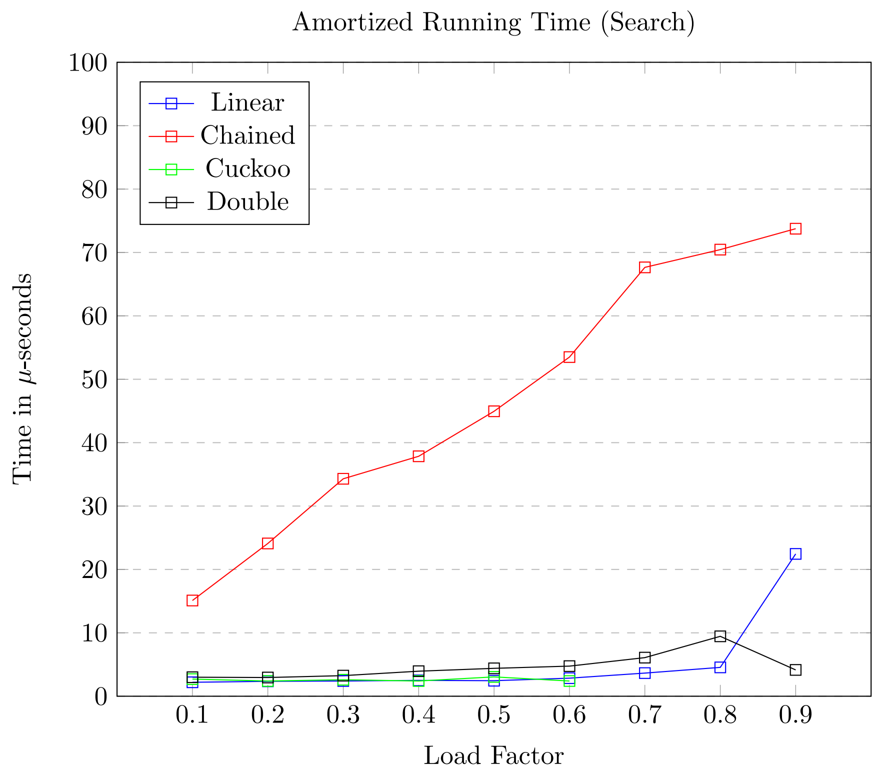
2.4.4 The experimental search and delete time are constant as expected theoretically. The insert time also grows linearly by O(α). At load factor = 0.9, there is a huge spike, as the number of rehashes is very large.

3. Comparitive Analysis Draw plots of various hashing schemes.

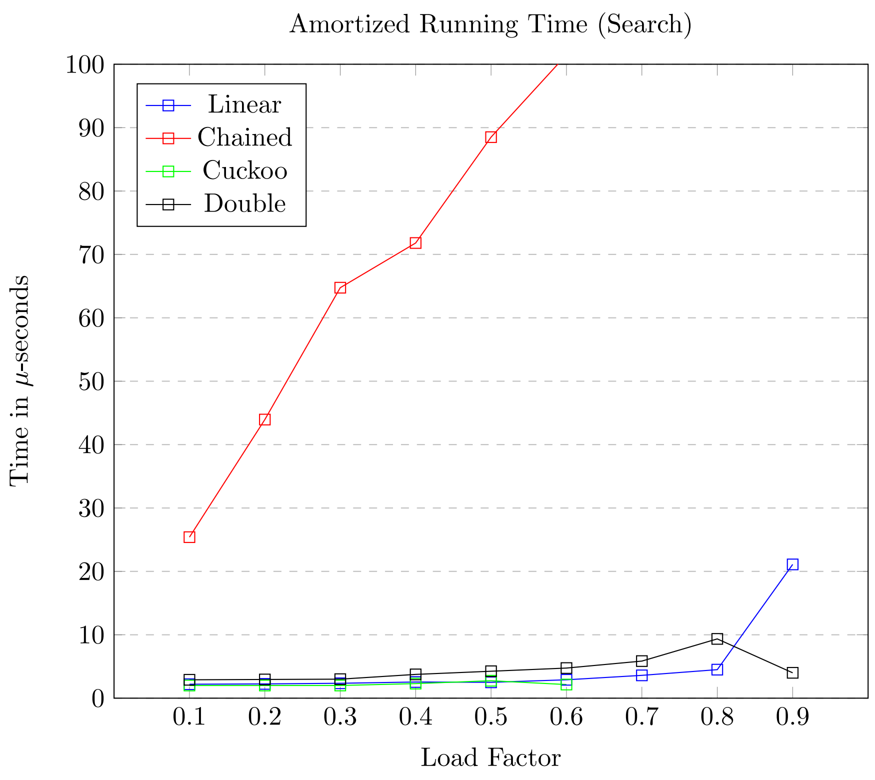
3.1 Insert with Uniform Key Generation



3.2 Search with Uniform Key Generation



3.3 Delete with Uniform Key Generation



3.4 Comparative performance summary It looks like Double Hashing performs the best across all the operations, against the other hashing schemes. The search and delete time are constant (though the constant is bigger than that for Cuckoo). The insert time is the best for linear-probing, but again Double hashing is sufficiently close. Cuckoo hashing is useless after load-factor of 0.6, and chained hashing clearly grows linearly for every operation.