# HASHING ALGORITHMS

**DHARSHAN BIRUR JAYAPRABHU**
**54773179**

## 1) PSEUDO CODE

**LINEAR PROBING:**

Search:
```
def search(k):
i = h(k)
while H[i] is nonempty and contains a key != k
   i = (i + 1) mod N
if H[i] is nonempty:
return its value
else: exception
```

Insert:
```
def insert(k,v):
i = h(k)
while H[i] is nonempty and contains a key != k
  i = (i + 1) mod N
store (k,v) in H[i]
```

Lazy delete :

```
Def delete(k):
i=h(k);
while H[i] is nonempty and contains a key != k
         i=(i+1)mod N
if H[i] is nonempty:
  mark H[i] as nonempty but unused
else: exception
```

**CUCKOO HASHING:**

**Search:**
```
def search(k):
if H0[h0(k)] is nonempty and contains a key == k
   return its value
else if H1[h1(k)] is nonempty and contains a key == k:
```

```
    return its value
else: exception
```

Insert:

```
def insert(k,v):
t = 0
while (k,v) is a nonempty pair:
(k,v) ↔ Ht [ht(k)]
t = 1 - t
```

Delete:

```
def delete(k):
if H0[h0(k)] is nonempty and contains a key == k
   delete the key,value pair
else if H1[h1(k)] is nonempty and contains a key == k:
         delete the key, value pair
else: exception
```

**CHAINED HASHING:**

**Search:**

```
def search(k):
i = h(k)
while H[i] is nonempty contains a key == k
   if contains a key == k : return its value
   else : H[i]=H[i].next;
 exception
```

Insert:

```
def insert(k,v):
i = h(k)
store (k,v) as head of H[i]
```

Delete:

```
def delete(k,v):
i = h(k)
while H[i] is nonempty contains a key == k
   if contains a key == k : delete(k,v)
   else : H[i]=H[i].next;
 exception
```

**DOUBLE HASHING:**

Search:

```
def search(k):
if H[h0(k] is nonempty and contains a key == k
   return its value
else :
   while i is less than size of table
         if H[h1(k,i0] contains a key==k:
             return its value;
    exception
```

Insert:

```
def insert(k,v):
i = h(k)
while H[h0(k)] is empty :
  store(k,v)
else:
  while i is less than size of table
         if H[h1(k,i0] is empty:
             store(k,v) ;
```

```
Def delete(k):
if H[h0(k] is nonempty and contains a key == k
   delete (k,v)
else :
   while i is less than size of table
         if H[h1(k,i0] contains a key==k:
             delete (k,v);
    exception
```

2) **Theoretical running times of algorithms:**

$\alpha$ is the load factor

**Hash Chaining** – time/operation = O(1+ $\alpha$)

**Linear Probing** –
Average search time is constant, but it depends on load factor ($\alpha$)
Expected time for Successful Search: 1/ 2 ( 1 + 1/ (1 − $\alpha$) )
Expected time for Unsuccessful Search: 1 /2 ( 1 + 1 (1 − $\alpha$)^2)

**Cuckoo Hashing** -
The expected time for the sequence of operations if O(n) [n - # of operations]

**Double Hashing-**

Expected time for Successful Search: $1/(1 - \alpha)$
Expected time for Unsuccessful Search: $1/\alpha + (1/\alpha) \cdot \ln(1/(1 - \alpha))$

3) **Reason for choosing Double hashing algorithm:**

The more the probability that two keys will follow exactly the same probe path ,the more will different keys interfere with each other, and therefore the worse the performance of our algorithm will be. This interference phenomenon is refered to as clustering.
In linear probing the probability that two keys will follow the same probe path is identical to the probability that they will hash to the same location, which is l/m. (m-size of the hash table)
In double hashing this probability is seen to be l/m(m - 1).
Thus, double hashing results in lesser clustering as compared to double hashing.

**4)**

**Comparative experimental analysis of algorithms:**

Load factor=0.7

Hash table size=97

Hash Function – (key)%size

LINEAR HASHING:

| #elements inserted | successful Search time (ms) | unsuccessful search time | Insert time (ms) | Delete time (ms) |
|---|---|---|---|---|
| 10 | 0.011 | 0.32 | 0.755 | 0.011 |
| 100 | 0.022 | 2.355 | 0.981 | 0.024 |
| 1000 | 0.127 | 24.18 | 2.151 | 0.108 |
| 10000 | 0.761 | 1065.101 | 4.961 | 0.801 |

CHAINED HASHING:

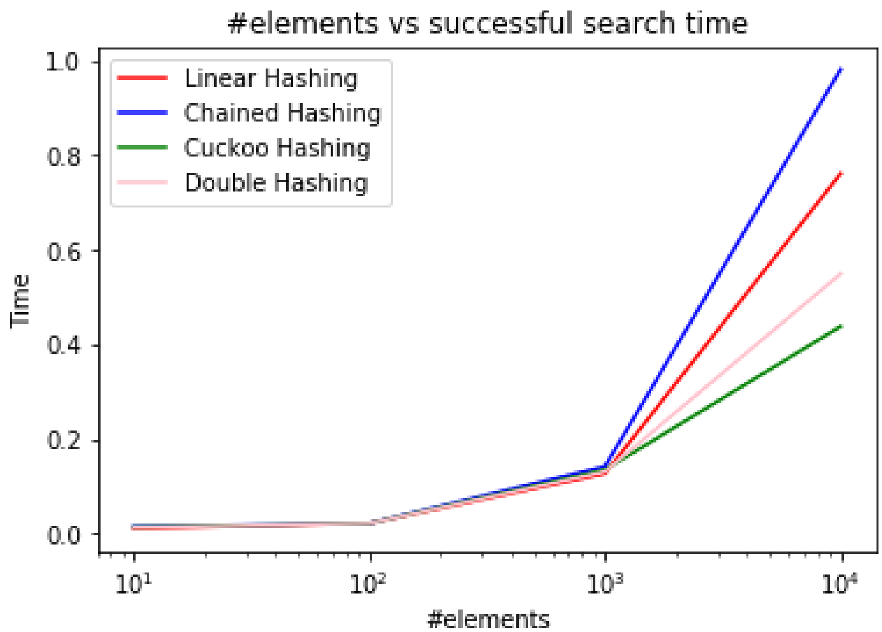
| #elements inserted | successful Search time(ms) | unsuccessful search time | Insert time(ms) | Delete time(ms) |
|---|---|---|---|---|
| 10 | 0.015 | 0.217 | 1.116 | 0.017 |
| 100 | 0.022 | 2.302 | 1.535 | 0.073 |
| 1000 | 0.142 | 11.967 | 4.793 | 0.286 |
| 10000 | 0.981 | 45.088 | 6.632 | 1.204 |

CUCKOO HASHING:

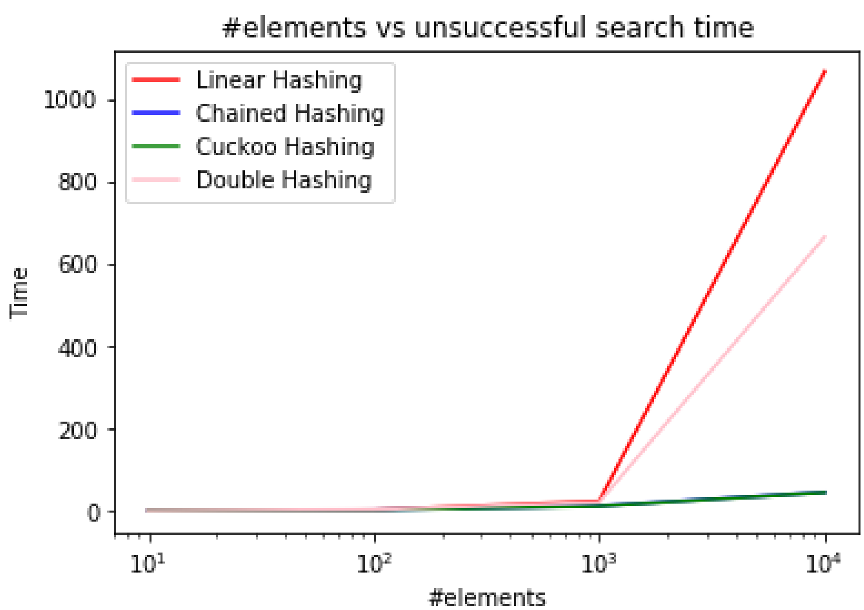
| #elements inserted | successful Search time(ms) | unsuccessful search time | Insert time(ms) | Delete time(ms) |
|---|---|---|---|---|
| 10 | 0.014 | 0.408 | 0.789 | 0.015 |
| 100 | 0.022 | 2.064 | 2.975 | 0.045 |
| 1000 | 0.135 | 11.245 | 30.454 | 0.132 |
| 10000 | 0.438 | 44.552 | 293.822 | 0.541 |

DOUBLE HASHING:

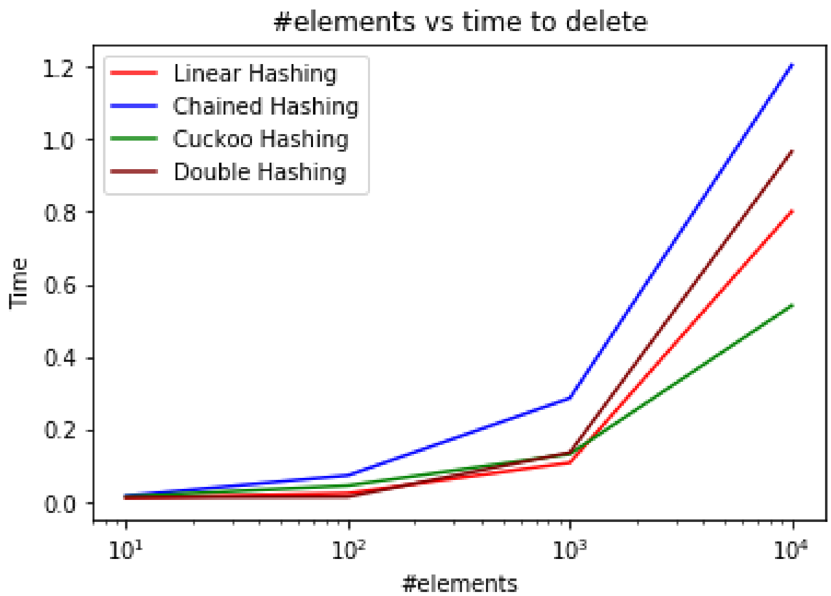
| #elements inserted | successful Search time(ms) | unsuccessful search time | Insert time(ms) | Delete time(ms) |
|---|---|---|---|---|
| 10 | 0.013 | 0.374 | 0.792 | 0.012 |
| 100 | 0.022 | 3.03 | 0.961 | 0.014 |
| 1000 | 0.131 | 20.981 | 1.834 | 0.136 |
| 10000 | 0.549 | 664.176 | 5.295 | 0.966 |



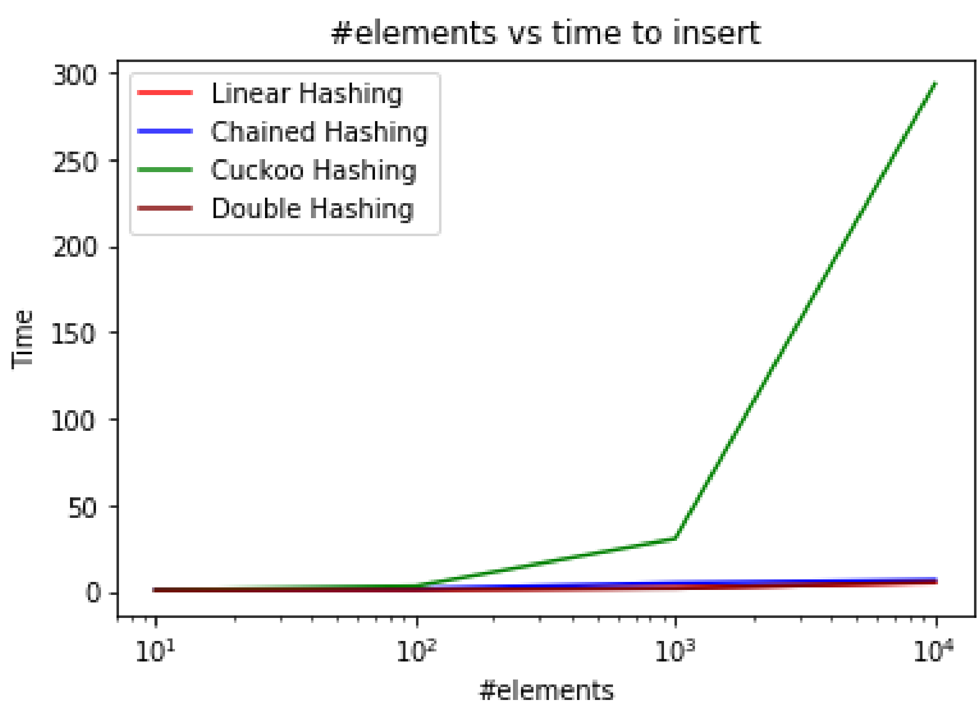

Conclusion:
Chained Hashing takes the largest amount of time for search operation .
Linear Hashing takes more time for searching as compared to Double hashing.




Conclusion:
Linear hashing takes the largest time for an unsuccessful search followed by Double Hashing.
Both Cuckoo Hashing and Chained hashing take very less time for an unsuccessful search.




Conclusion:
Cuckoo Hashing takes the least amount of time for the delete operation.
Chained Hashing takes the maximum time for performing delete operation.




Conclusions:
Cuckoo Hashing takes the largest amount of time for performing insert operation .

**Conclusions Summary:**

Chained Hashing takes the largest amount of time for search operation .
Linear Hashing takes more time for searching as compared to Double hashing.
Linear hashing takes the largest time for an unsuccessful search followed by Double Hashing.
Both Cuckoo Hashing and Chained hashing take very less time for an unsuccessful search.
Cuckoo Hashing takes the least amount of time for the delete operation.
Chained Hashing takes the maximum time for performing delete operation.
Cuckoo Hashing takes the largest amount of time for performing insert operation .

All the conclusions made through experimental analysis are inline with the theoretical running time analysis.

**Analysis based on varying load factor**:

Number of elements inserted- 10000
Hash table initial size=97
Hash Function – (key)%size

Linear Hashing:

| Load factor | successful Search time (ms) | unsuccessful search time (ms) | Insert time (ms) | Delete time (ms) |
|---|---|---|---|---|
| 0.4 | 1.836 | 2514.579 | 11.292 | 1.444 |
| 0.6 | 2.484 | 2500.283 | 11.581 | 2.152 |
| 0.8 | 1.745 | 2521.875 | 11.777 | 3.181 |
| 1 | 1.805 | 2511.788 | 11.373 | 1.932 |

Chained Hashing:

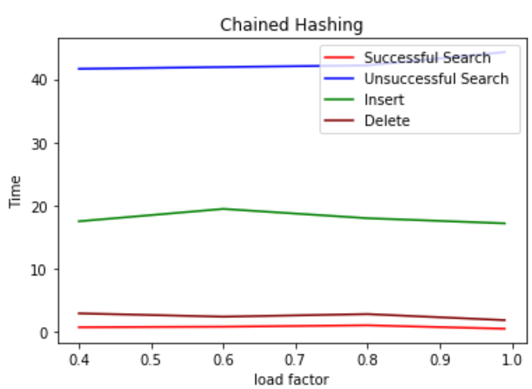
| Load factor | successful Search time (ms) | unsuccessful search time (ms) | Insert time (ms) | Delete time (ms) |
|---|---|---|---|---|
| 0.4 | 0.711 | 41.64 | 17.491 | 2.907 |
| 0.6 | 0.798 | 41.922 | 19.452 | 2.385 |
| 0.8 | 1.018 | 42.184 | 17.978 | 2.793 |
| 1 | 0.479 | 44.286 | 17.168 | 1.836 |

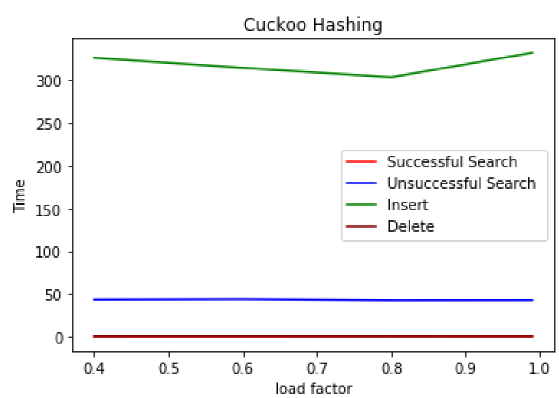
Cuckoo Hashing:

| Load factor | successful Search time (ms) | unsuccessful search time (ms) | Insert time (ms) | Delete time (ms) |
|---|---|---|---|---|
| 0.4 | 0.576 | 43.499 | 325.748 | 0.662 |
| 0.6 | 0.553 | 44.026 | 314.044 | 0.673 |
| 0.8 | 0.627 | 42.597 | 303.077 | 0.657 |
| 1 | 0.631 | 42.77 | 331.672 | 0.612 |

Double Hashing:
h1(k)=(key)%size;
hash=prime-(prime)%size
h2(k)=(hash+i*hash)%size

| Load factor | Successful Search time (ms) | unsuccessful search time (ms) | Insert time (ms) | Delete time (ms) |
|---|---|---|---|---|
| 0.4 | 0.579 | 634.369 | 4.894 | 0.497 |
| 0.6 | 0.422 | 590.813 | 5.262 | 0.498 |
| 0.8 | 0.418 | 628.386 | 4.608 | 0.993 |
| 1 | 0.595 | 635.699 | 4.894 | 0.966 |




Conclusion:
The experimental running time for successful search, unsuccessful search , delete, insert with respect to the load factor is inline with the theoretical analysis of load factor and the various operations.