Pseudo-codes

Linear Hashing:

The Linear Hashing algorithm for HashMap is implemented using a hash-table and its hash-function where-in every key has a hash-code which is obtained from the hash-function and the keys are inserted in the hash-table using these hash-codes. In Linear Hashing, if there exists an old key in the cell in which the new key was supposed to be inserted then the new key is simply inserted in the next empty cell after it. While inserting, if the load factor of the table reaches 1, then we re-size the table and increase its size. When removing a key, if the load factor of the table goes below 0.25, then we decrease the table size.

To implement linear hashing algorithm I will create an array of Hash-Entry objects and maintain a size variable to keep a count of keys present in the hash-table. Hash-entry object has key and value variables.

Following are the pseudo-codes for every method implemented in a HashMap:

Insert:

Find the first empty cell in the hash-table (H) starting from the hash-code index (i) obtained from the hash-function and insert the new hash-entry object (containing the key value pair) in it. Increment the size variable by 1.

```
def set(k, v):
        i = h(k)
        while H[i] is not null and contains key not equal to k
                i = (i + 1) % N
        H[i] = new HashEntry(k, v)
        size = size + 1
```

Delete:

Get the hash-code from the hash-function for the key to be removed. Find the key by traversing through the hash-table starting from hash-code index. Delete the key-value object by eager-delete method. Decrement the size variable by 1.

```
def delete(k):
        i = h(k)
        while H[i] is not null and contains key not equal to k
                i = (i + 1) % N
        if H[i] is null
                throw exception
        j = (i + 1) % N
        while H[j] is not null
                if h(H[j].key) is not in range (i+1 to j)
                        H[i] = H[j]
                        i = j
                j = (j + 1) % N
        H[i] = null
        size = size - 1
```

Search:
Get the hash-code from the hash-function for the key to be searched. Find the key by traversing through the hash-table starting from hash-code index.

```
def search(k):
        i = h(k)
        while H[i] is not null and contains a key not equal to k
                i = (i + 1) % N
        if H[i] is null
                return exception
        return value from the hashentry
```

Size:
It gives the number of keys present in the hash table. It return the size variable which has the count of keys present in the table.

```
def size:
        return size
```

IsEmpty:
Gives a boolean output of whether keys are present in the hash table.

```
def isempty:
        return size == 0
```

Chained Hashing:
The Chained Hashing algorithm for HashMap is implemented using a hash-table and its hash-function where-in every key has a hash-code which is obtained from the hash-function and the keys are inserted in the hash-table using these hash-codes. In Chained Hashing, every cell in the hash table is a linked list and all the keys having the same hash code belong to the same linked list. While inserting, if the load factor of the table reaches 20, then we resize the table and redistribute the keys. When removing a key, if the load factor of the table goes below 0.25, then we decrease the table size.

To implement chained hashing algorithm I will create an array of Linked List and maintain a size variable to keep a count of keys present in the hash-table. Hash-entry node has key, value variables and address to the next node in the linked list.

Following are the pseudo-codes for every method implemented in a HashMap:

Insert:
Find the hash-code index (i) in the hash-table (H) obtained from the hash-function and add the new hash-entry node (containing the key value pair) in the linked list if the hash-entry node does not exist

with the same key. Increment the size variable by 1. If the hash-entry node already exists, then just update the value in the node.

```
def set(k, v):
        i = h(k)
        if H[i] is null
                H[i] = new Node(k, v)
                size = size + 1
        else
                while H[i].next is not null and contains key not equal to k
                        H[i] = H[i].next
                if H[i].key == k
                        H[i].value = v
                else
                        H[i].next = new Node(k, v)
                        size = size + 1
```

Delete:

Get the hash-code from the hash-function for the key to be removed. Find the key by traversing through the linked list at the hash-code index. Delete the node from the linked list. Decrement the size variable by 1.

```
def delete(k):
        i = h(k)
        if H[i] is null
                throw exception
        else
                prevnode = null
                node = H[i]
                while H[i].next is not null and contains key not equal to k
                        prevnode = node
                        node = node.next
                if node.key is equal to k
                        delete the node and change the next address of the previous node
                        size = size - 1
                else
                        throw exception
```

Search:

Get the hash-code from the hash-function for the key to be removed. Find the key by traversing through the linked list at the hash-code index. Return the value of the node.

```
def search(k):
        i = h(k)
        if H[i] is null
                return exception
```

```
        else
                while H[i] is not null and contains key not equal to k
                        H[i] = H[i].next
                if H[i] is null
                        return exception
                return value from the node
```

Size:
It gives the number of keys present in the hash table. It return the size variable which has the count of keys present in the table.

```
def size:
        return size
```

IsEmpty:
Gives a boolean output of whether keys are present in the hash table.

```
def isempty:
        return size == 0
```

Cuckoo Hashing:
The Cuckoo Hashing algorithm for HashMap is implemented using two hash-tables and its hash-functions where-in every key has two hash-codes which are obtained from the hash-functions and the keys are inserted in the first hash-table initially using these hash-codes. If there exists another key in the hash-code of the first table, this existing key is shifted to the second hash-table using the second hash-function and this goes on if there again exists another key in the second hash-table as well until an empty hash-code is found in one of the hash table. If the original key is replaced thrice then we have reached a deadlock and that key cannot be inserted in any of the hash-table. While inserting, if the load factor of the table reaches 1 or we encounter a cycle, then we re-size the table and increase its size. When removing a key, if the load factor of the table goes below 0.25, then we decrease the table size.

To implement cuckoo hashing algorithm I will create a 2D array of hash-entry objects having 2 rows representing 2 hash tables and maintain a size variable to keep a count of keys present in the hash-table. Hash-entry object has key and value variables.

Following are the pseudo-codes for every method implemented in a HashMap:

Insert:
Find the key if it already exists in one of the hash-tables (H) and update the value. If this is a new key, get the hash-code (i) for the first table. If there exists another key in the hash-code of the first table, this existing key is shifted to the second hash-table using the second hash-function and this goes on if there again exists another key in the second hash-table as well until an empty hash-code is found in one of the hash table. If the original key is replaced thrice then we have reached a deadlock and that key cannot be inserted in any of the hash-table. Increment the size variable by 1.

```
def set(k, v):
        i0 = h0(k)
        i1 = h1(k)
        if H[0][i0] contains key equal to k
                H[0][i0].value = v
                return
        else if H[1][i1] contains key equal to k
                H[1][i1].value = v
                return
        t = 0
        count = 0
        while (k, v) is not null
                if (k, v) is the original key-value pair
                        count = count + 1
                if count == 3
                        throw exception for deadlock
                temp = (k, v)
                (k, v) = H[t][it]
                H[t][it] = temp
                t = 1 - t
size = size + 1
```

Delete:

Get both the hash-codes from the hash-functions for the key to be removed. Find the key in one of both the hash tables at the hash-code indices. Delete the key-value object. Decrement the size variable by 1.

```
def delete(k):
        i0 = h0(k)
        i1 = h1(k)
        if H[0][i0] contains key equal to k
                H[0][i0] = null
                size = size - 1
        else if H[1][i1] contains key equal to k
                H[1][i1] = null
                size = size – 1
        else
                throw exception
```

Search:

Get both the hash-codes from the hash-functions for the key to be searched. Find the key in one of both the hash tables at the hash-code indices. Return value of the object.

```
def search(k):
        i0 = h0(k)
        i1 = h1(k)
        if H[0][i0] contains key equal to k
                return H[0][i0].value
        else if H[1][i1] contains key equal to k
```

```
            return H[1][i_1].value
    else
            return exception
```

Size:
It gives the number of keys present in the hash table. It return the size variable which has the count of keys present in the table.

```
def size:
    return size
```

IsEmpty:
Gives a boolean output of whether keys are present in the hash table.

```
def isempty:
    return size == 0
```

Double Hashing:
The Double Hashing algorithm for HashMap is implemented using a hash-table and two hash-functions where-in every key has a hash-code which is obtained using these two hash-functions and they are inserted in the hash-table using these hash-codes. In Double Hashing, if there exists an old key in the cell in which the new key was supposed to be inserted (using only the first hash function) then the new key is inserted using new hash-code obtained by adding the second hash-function's value to the previous one. Keep on adding the second hash-function value until an empty cell is found. While inserting, if the load factor of the table reaches 1, then we re-size the table and increase its size. When removing a key, if the load factor of the table goes below 0.25, then we decrease the table size.

Double Hashing is a better method for hashing because it uses two hash-functions for inserting the key-value pair. Unlike linear probing and quadratic probing, the interval depends on the key, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering. Also, unlike cuckoo hashing, there are no chances of infinite cycles while adding a new key.

To implement double hashing algorithm I will create an array of Hash-Entry objects and maintain a size variable to keep a count of keys present in the hash-table. Hash-entry object has key and value variables.

Following are the pseudo-codes for every method implemented in a HashMap:

Insert:
Find the first empty cell in the hash-table (H) starting from the hash-code index obtained from the first hash-function and insert the new hash-entry object (containing the key value pair) in it. If the index is occupied, add second hash-code value obtained by the second hash-function to the index.

Keep on adding the second hash-function value until an empty cell is found. Increment the size variable by 1.

```
def set(k, v):
        i = h1(k)
        while H[i] is not null and contains key not equal to k
                i = (i + h2(k)) % N
        H[i] = new HashEntry(k, v)
        size = size + 1
```

Delete:

Get the hash-code from the first hash-function for the key to be removed. Find the key by traversing through the hash-table using the two hash-functions (h1 + h2). Delete the key-value object by lazy-delete method. Decrement the size variable by 1.

```
def delete(k):
        i = h1(k)
        while H[i] is not null and contains key not equal to k
                i = (i + h2(k)) % N
        if H[i] is null
                throw exception
        size = size - 1
```

Search:

Get the hash-code from the hash-function for the key to be searched. Find the key by traversing through the hash-table starting from hash-code index.

```
def search(k):
        i = h1(k)
        while H[i] is not null and contains a key not equal to k
                i = (i + h2(k)) % N
        if H[i] is null
                return exception
        return value from the hashentry
```

Size:

It gives the number of keys present in the hash table. It return the size variable which has the count of keys present in the table.

```
def size:
        return size
```

IsEmpty:

Gives a boolean output of whether keys are present in the hash table.

```
def isempty:
        return size == 0
```

Experimental Analysis:

Test Case 1: (Generic example)

Insert: (17, 24, 35, 45, 84, 85, 58, 20, 65, 39, 30) with table size = 13

Search: (24, 35, 45, 84, 20, 65, 39, 30)

Delete: (24, 35, 30, 20, 45, 39, 65, 84)

Linear:

Insertions –

Time: 32468ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12
65	39			17	30	45	84	85	35	58	24	20

Search –

Time: 12830ns (8 searches)

Deletions –

Time: 21467ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12
				17		58	85					

Chained:

Insertions –

Time: 29509ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12
65 39				17 30		45 84 58	85 20		35		24	

Search –

Time: 8126ns (8 searches)

After Deletions –

Time: 7270ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12
				17		58	85					

Cuckoo:

Insertions –

Time: 38489ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12
39				30		84	20		35		24	
	17		45	58	65	85						

Search –

Time: 5560ns (8 searches)

Deletions –
Time: 3849ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12
	17			58		85						

Double:

Insertions –
Time: 28652ns (11 inserts)

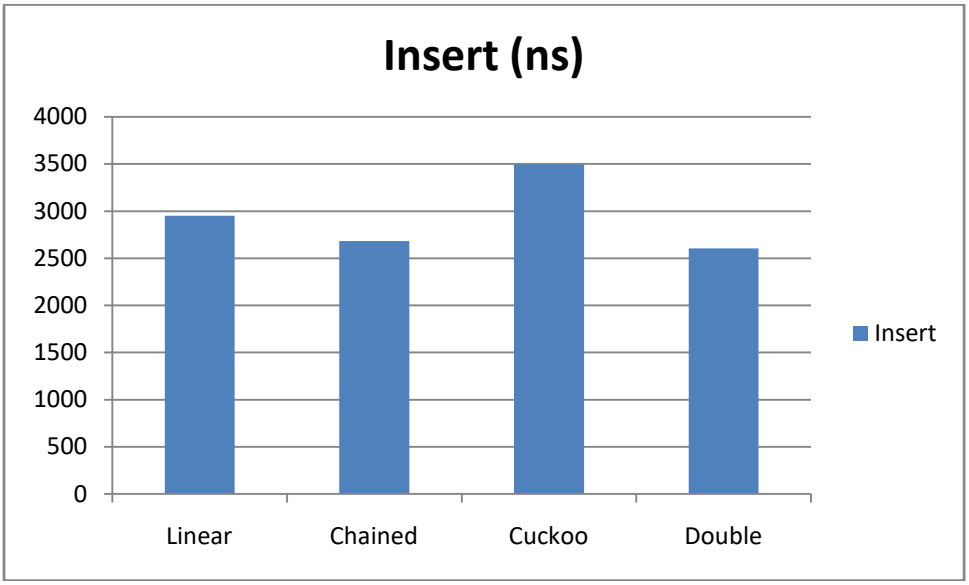
0	1	2	3	4	5	6	7	8	9	10	11	12
20	58	65	30	17	39	45	85		35	84	24	

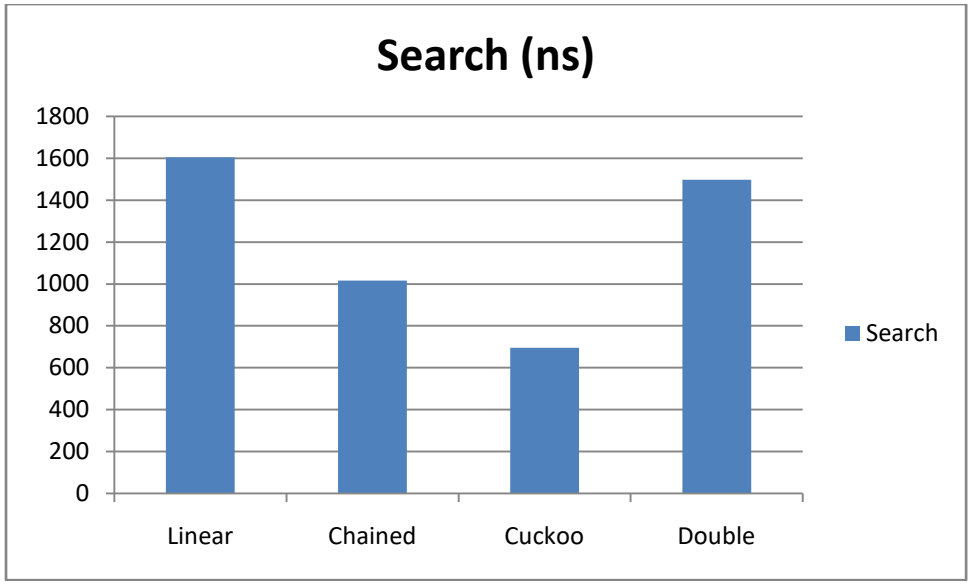
Search –
Time: 11975ns (8 searches)

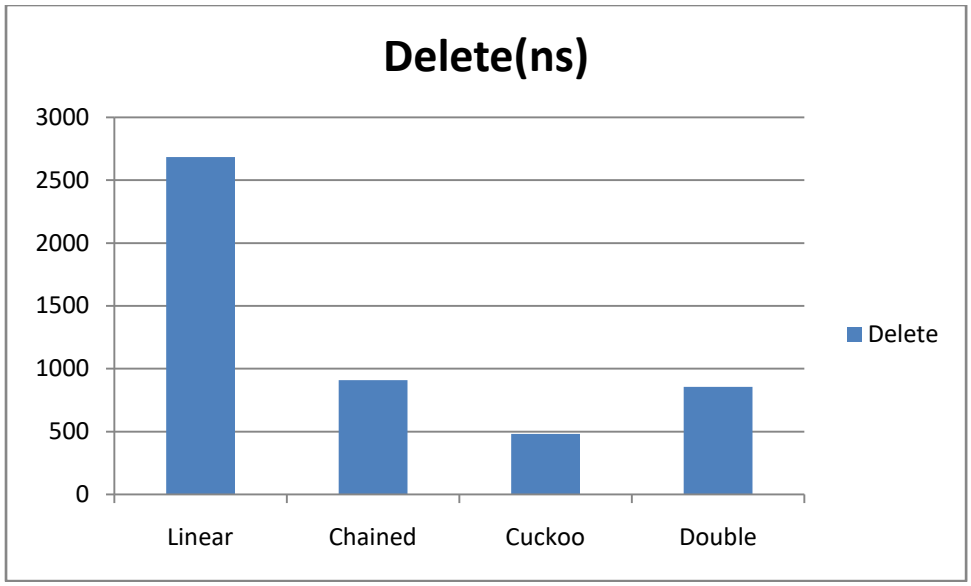
Deletions –
Time: 6843ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12
	58			17			85					

Graphs:







Observation:
While inserting the key-value pairs, Cuckoo hashing takes more time than other algorithms as it tries to rearrange existing key-value pairs as well.
For searching an existing key in the hash-table Linear hashing takes more time than others as it has to traverse through the cluster if the key is not at the right place.
While deleting, again Linear hashing takes more time as it follows the eager delete method which rearranges the existing keys after deletion.

Test Case 2: (All overlapping keys)
Insert: (20, 88, 37, 71, 54, 105, 139, 122, 156, 173, 190) with table size = 17
Search: (105, 122, 190, 20, 88, 139, 156, 37)
Delete: (105, 88, 71, 139, 122, 37, 190, 173)

Linear:

Insertions –
Time: 43620ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			20	88	37	71	54	105	139	122	156	173	190			

Search –
Time: 22666ns (8 searches)

Deletions –
Time: 30344ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			20	54	156											

Chained:

Insertions –
Time: 33425ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			20 88 37 71 54 105 139 122 156 173 190													

Search –
Time: 12395ns (8 searches)

After Deletions –
Time: 11949ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			20 54 156													

Cuckoo:

Insertions –
Time: 34152ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			190													
	20	37	54	71	88	105	122	139	156	173						

Search –
Time: 5780ns (8 searches)

Deletions –

Time: 4145ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	20		54						156							

Double:

Insertions –

Time: 34942ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			20	54		173	139	105	71	37	190	156	122	88		

Search –

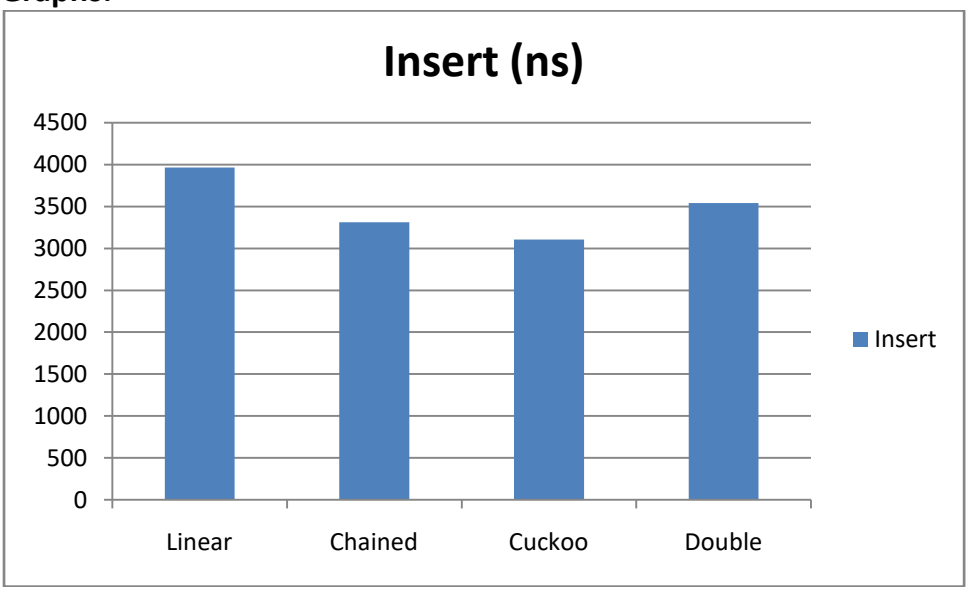
Time: 20158ns (8 searches)

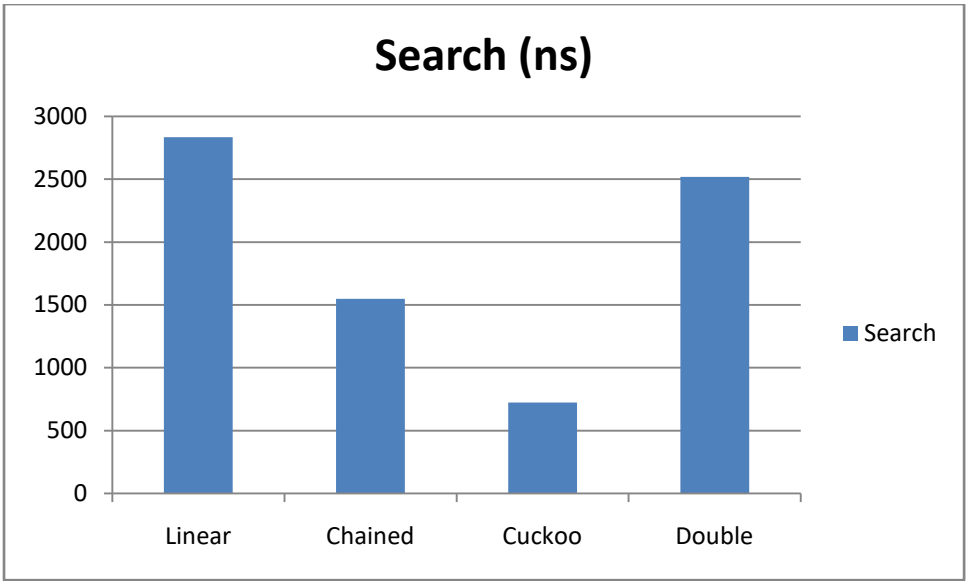
Deletions –

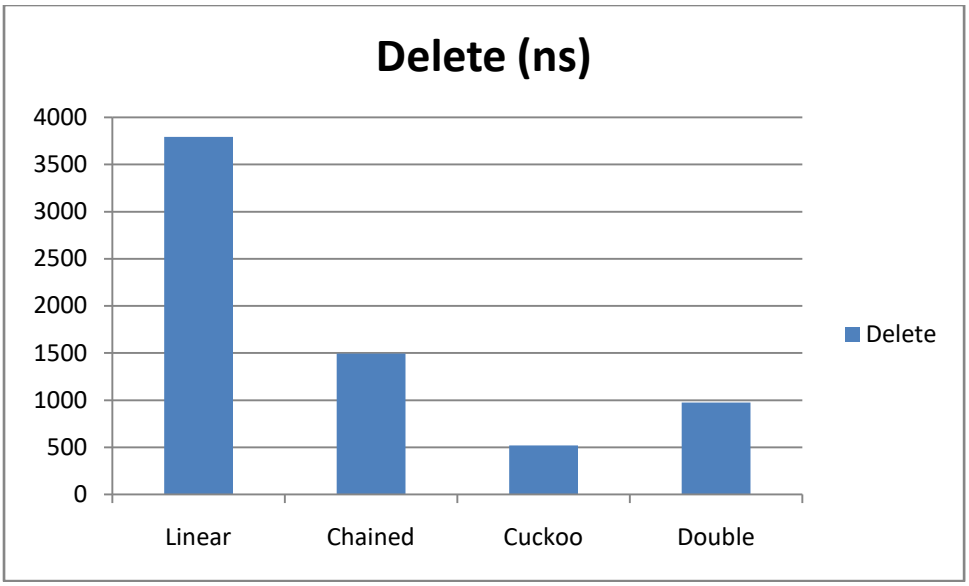
Time: 7808ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			20	54								156				

Graphs:







Observation:
When all the keys have same hash-code, Linear hashing takes more time for inserts, searches and deletes as it traverses through the cluster of keys to insert/search.

Test Case 3: (No overlapping keys)
Insert: (20, 34, 52, 70, 90, 109, 24, 40, 100, 63, 81) with table size = 17
Search: (24, 52, 40, 109, 100, 63, 90, 34)
Delete: (100, 70, 52, 81, 63, 34, 24, 20)

Linear:

Insertions –
Time: 24678ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	52	70	20		90	40	24			112		63	81		100	

Search –
Time: 4230ns (8 searches)

Deletions –
Time: 5241ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
					90	40				112						

Chained:

Insertions –
Time: 24118ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	52	70	20		90	40	24			112		63	81		100	

Search –
Time: 4186ns (8 searches)

After Deletions –
Time: 4212ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
					90	40				112						

Cuckoo:

Insertions –
Time: 24684ns (11 inserts)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	52	70	20		90	40	24			112		63	81		100	

Search –
Time: 4191ns (8 searches)

Deletions –
Time: 4209ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
					90	40				112						

Double:

Insertions –
Time: 24234ns (11 inserts)

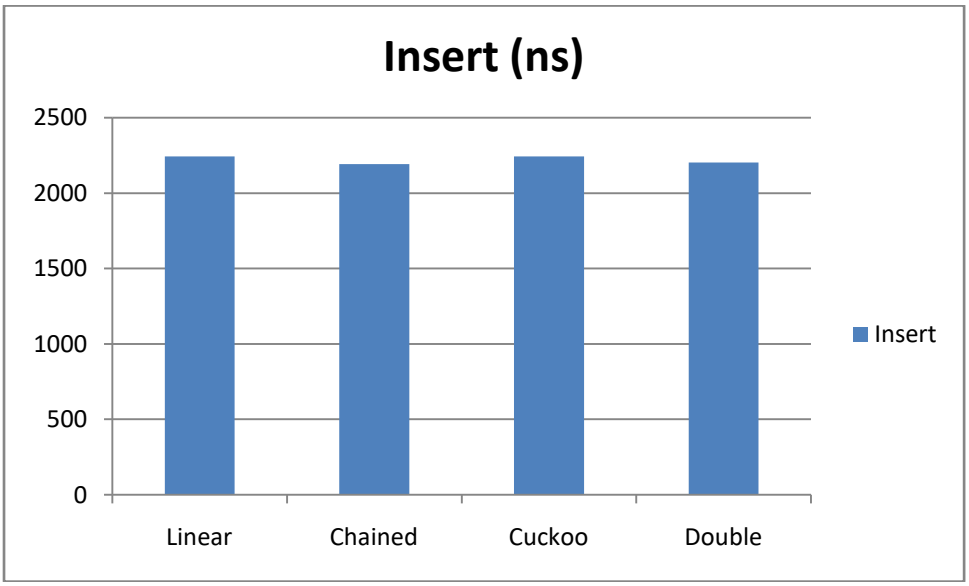
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	52	70	20		90	40	24			112		63	81		100	

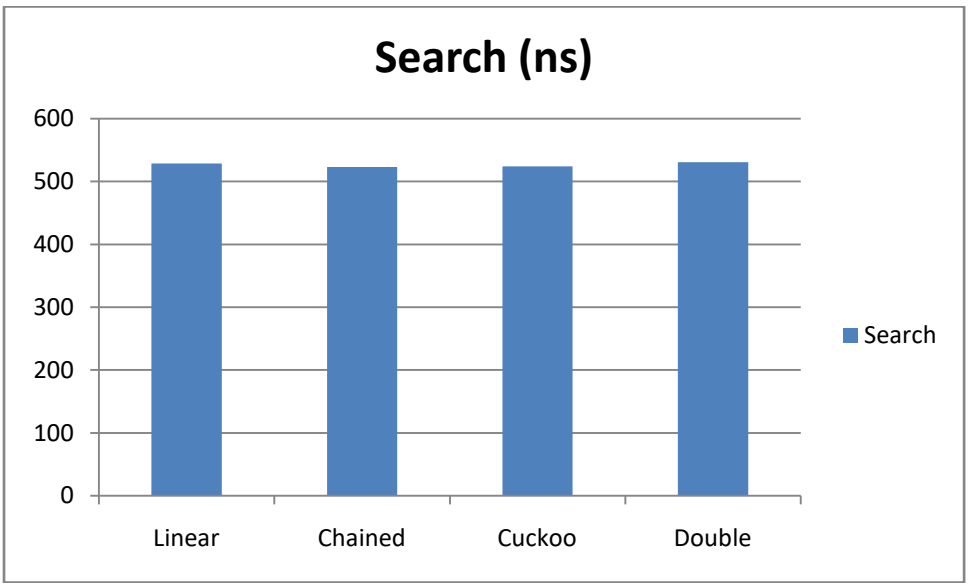
Search –
Time: 4245ns (8 searches)

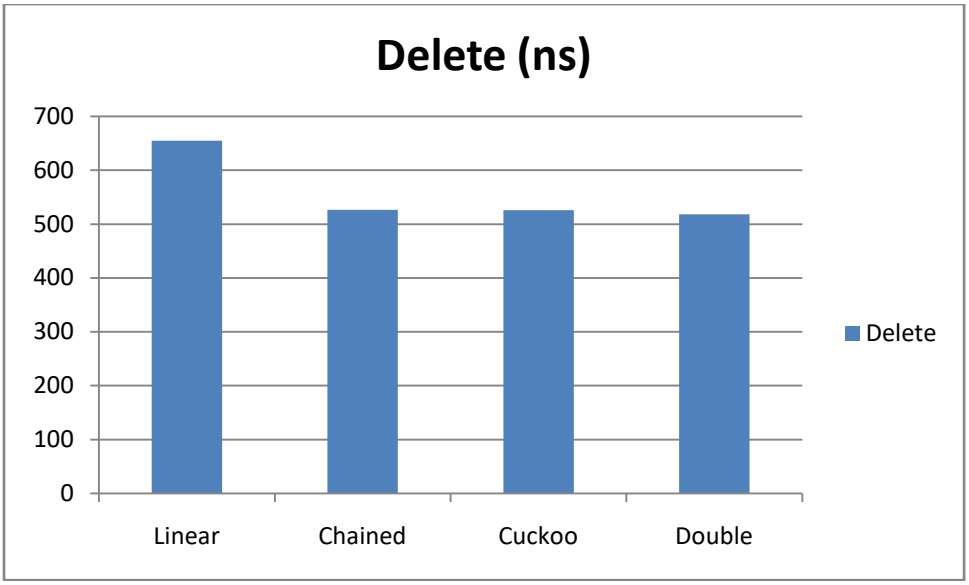
Deletions –
Time: 4143ns (8 deletes)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
					90	40				112						

Graphs:







Observation:
When there is no overlapping between the keys, i.e. all the keys have unique hash-codes, all the hashing techniques have the same complexity and run time.

Test Case 4: (Multiple random keys)
Insert: 500 key-value pairs with table size = 2017
Search: 300 keys
Delete: 300 keys

Linear:

Insertions – Time: 725912ns (500 inserts)
Search – Time: 382830ns (300 searches)
Deletions – Time: 581467ns (300 deletes)

Chained:

Insertions – Time: 689520ns (500 inserts)
Search – Time: 288126ns (300 searches)
Deletions – Time: 380140ns (300 deletes)

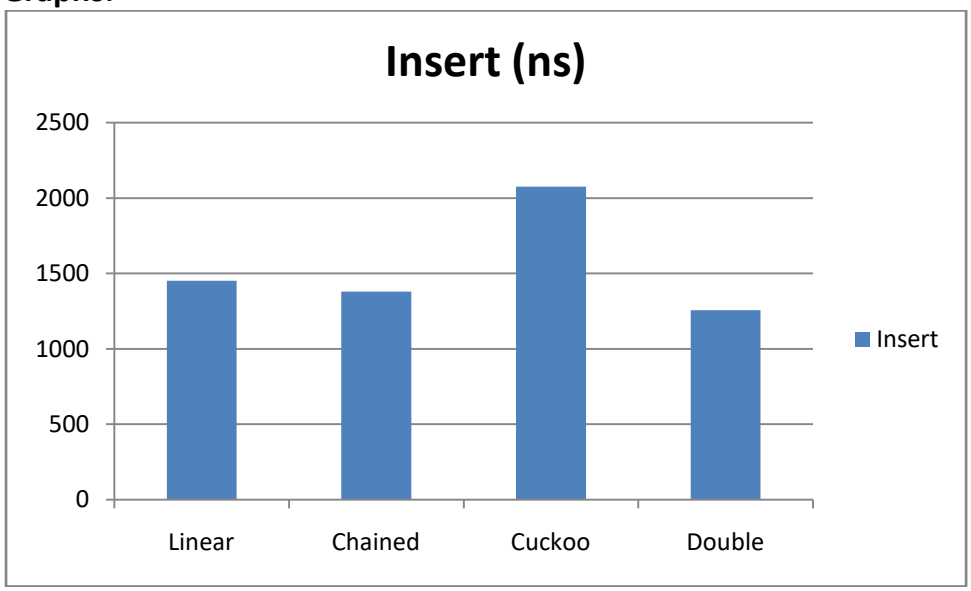
Cuckoo:

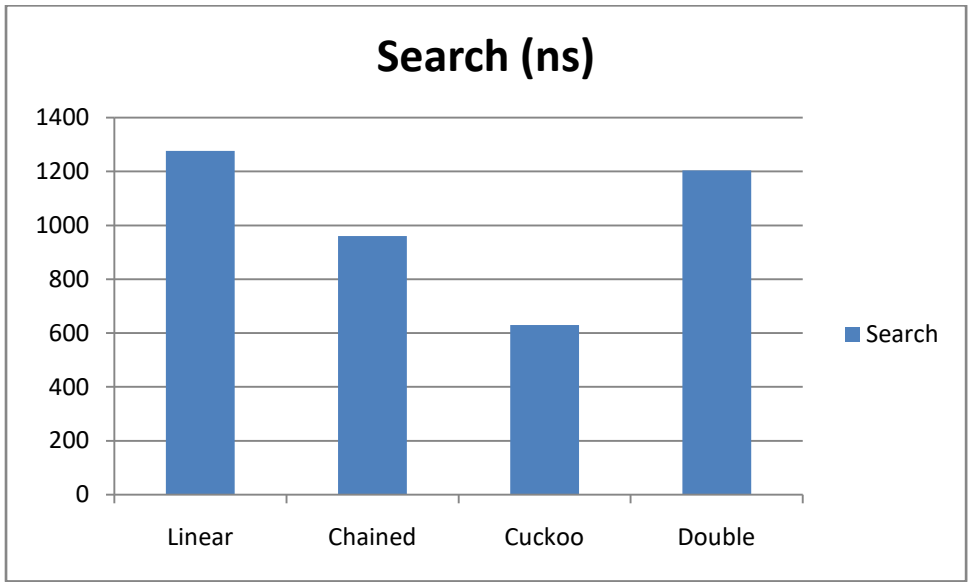
Insertions – Time: 1038411ns (500 inserts)
Search – Time: 189060ns (300 searches)
Deletions – Time: 280549ns (300 deletes)

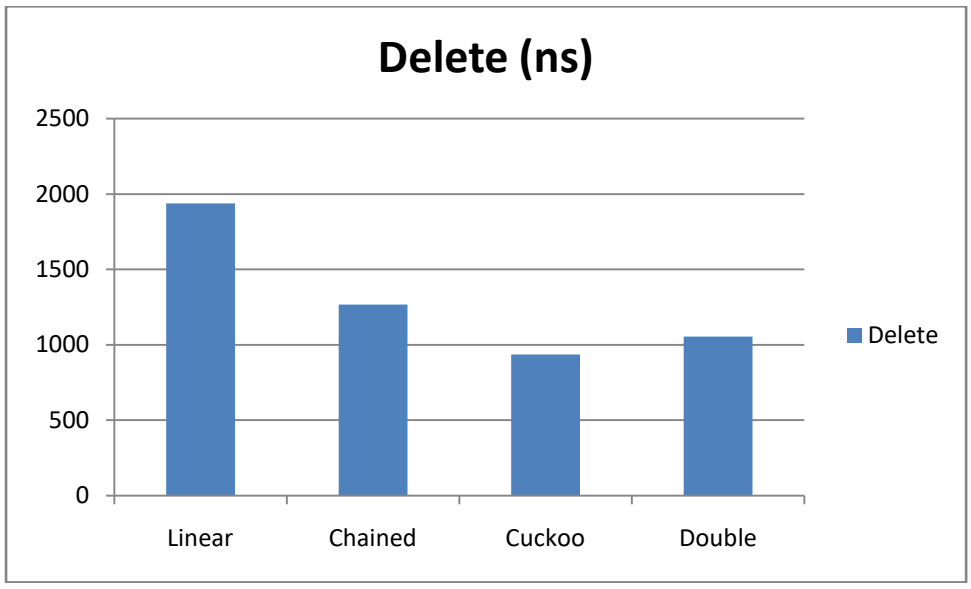
Double:

Insertions – Time: 628761ns (500 inserts)
Search – Time: 361195ns (300 searches)
Deletions – Time: 316343ns (300 deletes)

Graphs:


Insert (ns)
2500
2000
1500
1000
500
0
Linear
Chained
Cuckoo
Double
Insert


Search (ns)
1400
1200
1000
800
600
400
200
0
Linear
Chained
Cuckoo
Double
Search


Delete (ns)
2500
2000
1500
1000
500
0
Linear
Chained
Cuckoo
Double
Delete

Observation:
While inserting the key-value pairs, Cuckoo hashing takes more time than other algorithms as it tries to rearrange existing key-value pairs as well.
For searching an existing key in the hash-table Linear hashing takes more time than others as it has to traverse through the cluster if the key is not at the right place.
While deleting, again Linear hashing takes more time as it follows the eager delete method which rearranges the existing keys after deletion.

Conclusion:

Linear Hashing:
Insert: Average time complexity: O(1) and worst time complexity: O(n)
Search: Average time complexity: O(1) and worst time complexity: O(n)
Delete: Average time complexity: O(1) and worst time complexity: O(n)

Chained Hashing:
Insert: Average time complexity: O(1) and worst time complexity: O(1 + length of longest linked list)
Search: Average time complexity: O(1) and worst time complexity: O(1 + length of longest linked list)
Delete: Average time complexity: O(1) and worst time complexity: O(1 + length of longest linked list)

Cuckoo Hashing:
Insert: Average time complexity: O(1) and worst time complexity: O(n)
Search: Average time complexity: O(1) and worst time complexity: O(1)
Delete: Average time complexity: O(1) and worst time complexity: O(1)

Double Hashing:
Insert: Average time complexity: O(1) and worst time complexity: O(n)
Search: Average time complexity: O(1) and worst time complexity: O(n)
Delete: Average time complexity: O(1) and worst time complexity: O(n)