Project-1

Hashing Algorithms

By Sameer Ganesh Shinde, Student ID: 19644693

1. Test Suite

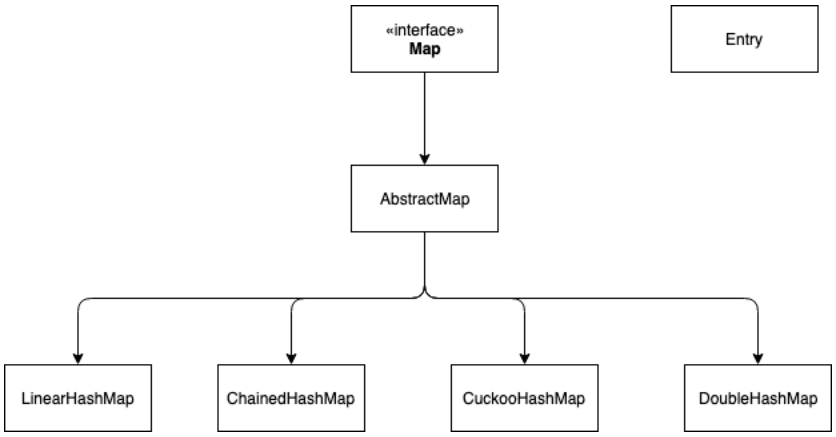
I have prepared an extensive test suite for number of test cases applicable for the mapping algorithms developed. These cases cover most of the use cases in regular usage of Hash Map.

Test Case	Description	Applicable to
0	Map Creation and simply put operation	All
1	Duplicate keys insertion	All
2	Simple get operation	All
3	Test for non-existing key	All
4	adding high number (100) of keys	All
5	adding very high number (10000) of keys	All
6	Simple remove operation	All
7	Adding 10 elements and removing 5	All
8	Adding 10000 and removing half of that	All
9	Adding 100000 and removing half of that	Linear, Chained and Double
10	Cycle resolution	Cuckoo Hashing
11	Continuous Cycle	Cuckoo Hashing

Note: I am restricting the size of keys for get to be around 30000 because, considering integer keys, the possible positive unique keys we can have is 32K. Therefore, analysis will be performed on that

How to run project

Project File structure:



Test Files Structure:

Utilities.java

- This file is responsible for populating LinearHashMap, CuckooHashMap, ChainedHashMap and DoubleHashMap.
- In addition, for testing purposes I am using HashMap from java's Utils library. All the operations performed on developed algorithms will also be performed on HashMap by java. At the end of each test case we will compare our algorithm's map with HashMap with size and elements for verifying the completeness of algorithm.

Methods	Description
populateLinearHashMap(String fileName)	This method returns the LHM with elements given in fileName
populateChainedHashMap(String fileName)	This method returns the CHM with elements given in fileName
populateCuckooHashMap(String fileName)	This method returns the CKHM with elements given in fileName

Methods	Description
populateDoubleHashMap(String fileName)	This method returns the DHM with elements given in fileName
populateOriginalHashMap(String fileName	This method returns the utils HashMap with elements given in fileName
Compare(AbstractMap map, HashMap hmap)	This method returns the comparison of AbstractMap passed and HashMap by utils for completeness.

Steps for Running Project

1. Unzip the given folder.
2. If you want to open project in eclipse, you can just import eclipse-project folder
3. Open All files together for running through command line.
4. Run

   For Chained, Double and Cuckoo hashing

   javac *.java

   mv *.class /bin

   - This bin directory will have all test cases for Double, Chained and Cuckoo Hashing

   cd bin

   run test case of your choice

   For Linear Probing,

   cd LinearProbing/

   javac *.java

   run test case of your choice

2. Linear Probing

This is a simple implementation of Linear HashMap. It doesn't allow resize and support only Integer keys and values.

Pseudo Code:

```
def get(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

def put(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]

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 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]:
			move H[j] to H[i]
			i = j
		j = j + 1
	clear H[i]
```

Algorithm Analysis:

Linear Probing - Analysis of search operation

- Worst-case time of a search operation can be really bad.
- Analyze average search time.
- Time for search(k) operation ≤ length of largest full contiguous block of cells containing H[h(k)].

  General fact about expected values: for any random variable V

  $E[\mathrm{V}] = \sum_x x.\Pr[\mathrm{V} = \mathrm{x}]$

  E[search time] ≤

  ≤ E[Largest full contiguous block of cells containing H[h(k)]

$= \sum_L L.\Pr$[There is a maximal full contiguous block of length L containing H[h(k)]]
$\leq \sum_L L.\Pr$[∃ full contiguous block of length L containing H[h(k)]]

- There are L different blocks of length L containing H[h(k)]
- The probability that any particular contiguous block of length L is full is $1/C^L$, where c > 1 is a constant that depends on the load factor α.
- The probability that at least one of L particular contiguous block of length L is full is ≤ L/C^L

E[search time] ≤
≤ $\sum_L L.\Pr$[∃ full contiguous block of length L containing H[h(k)]
≤ $\sum_L L.\frac{L}{C^L}$
= $\sum_{L=1}^{n} \frac{L^2}{C^L}$
= $\sum_{L=1}^{\infty} \frac{L^2}{C^L}$
= O(1).

Run Time Analysis:

size	put	get	remove
3	5058	19980	1 removal: 3666
12	8225	33775	-
7	10114	1 Unsuccessful Search: 100517 1 successful search: 748	Removal of 5 keys: 8776
100 for put /58 for get	55580	70793	-
10000 for put/ 1000 for get /500 for remove	3046222	433818	673445
100000 for put/31k for get and 15k for remove	18695048	5632984	4367559

3. Chained Hashing

This implementation of Chained HashMap automatically resize based on the load factor. When load factor exceeds 0.75 it doubles the size of buckets and when it is below 0.25 it halves the size of buckets. Rehashing is performed based on the new size of buckets. This class is implemented using Templates therefore it supports any type of Key and Value.

Pseudo Code:

```
def get(k):
	i = h(k)
	head = bucket[i]
	while head!=null
		if head.key == k
			return head.value
		head = head.next
	throw RunTimeException(“Key not found”)

def put(k,v):
	i = h(k)
	head = bucket[i]
	if head == null
		bucket[i] = (k,v)
		size++
	else
		while head !=null
			if head.key == k
				head.value = v
				break
			head = head.next
		if head == null
			head=bucket[i];
			(k,v).next=head;
			bucket[i] = (k,v);
			size++;
	if loadFactor > 0.75
		resize()

def delete(k):
	i = h(k)
	head=bucket[i]
	if head == null
		return null
	if head.key == k
		val=head.value
		head=head.next
		bucket[i] = head
```

```
        size--
        checkResize()
        return val
    else
        prev=null
        while head!=null
            if head.key == k
                prev.next=head.next;
                size--
                checkResize()
                return head.value
            prev=head;
            head=head.next;
        size--
        checkResize()
        return null
```

Algorithm Analysis:

- time/operation = O(1 + length(H[h(k)])
- We make random hash function assumption:
    - For each k, h(k) is uniformly distributed over indices
    - For any two keys k1 and k2: h(k1) and h(k2) are independent

  E[time / operation] =

  = O (1 + E [length of H[h(k)]])

  = O (1 + E [number of keys colliding with k])

  = O($1 + \sum_{other\ keys\ q} \Pr[\text{q collides with k}]$)

  = O (1 + (n − 1) $\cdot \frac{1}{n}$)

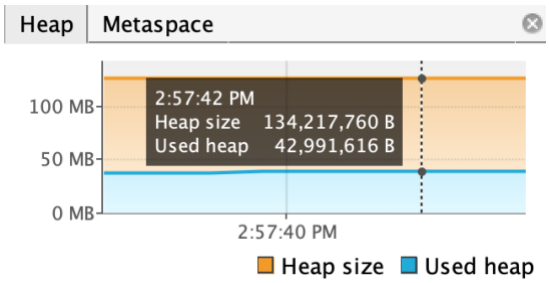
  = O (1 + α)

Run Time Analysis:

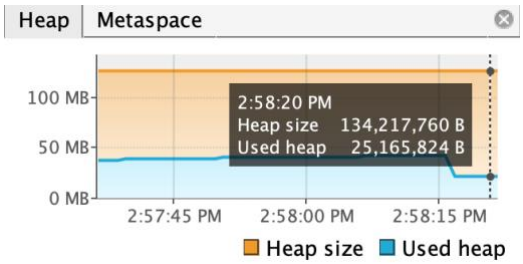
size	put	get	remove
3	281367	278	1 removal: 43128
12	319932	269	-
7	376893	1 Unsuccessful Search: 107684 1 successful search: 1874	Removal of 5 keys: 10420
100 for put /58 for get	1098404	274	-
10000 for put/ 1000 for get/ 500 for remove	7591134	500	524676
100000 for put/31000 for get/15000 for remove	41314710	566	2631357

Space Complexity:

Used Heap Size after adding 100000 elements (31k Unique)



Used Heap Size after deleting 15k keys



4. Cuckoo Hashing

This implementation of Cuckoo Hash Map automatically resizes based on the load factor. When load factor exceeds 0.75 it doubles the size of buckets and when it is below 0.25 it halves the size of buckets. While calculating the load factor since we are using two arrays, we consider size/2 in calculating load factor. Rehashing is performed based on the new size of buckets. This class is implemented using Templates therefore it supports any type of Key and Value.

Algorithm:

- Two tables: H0, H1. Maintained by dynamic array resizing. So, load factor $\alpha = n/N < 1$.
- Two hash functions: h0, h1
  h0 = no.of elements % length of array
  h1 = (no. of elements/11) % length of array
- Search(k): Look in both places: H0[h0(k)], H1[h1(k)]
- Delete(k): Look in both places, clear if found. Set(k,v):

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

- If we repeat the while loop more than c log n times for an appropriately chosen c, rebuild the entire structure with a new hash function

Algorithm Analysis:

If we use cuckoo hashing over a sequence of n operations:

- With probability 1 − Θ(1/n), it works.
- With probability Θ(1/n), rebuild. Hence,
- Probability of 1 rebuild: O(1/n)
- Probability of 2 rebuilds: O($\frac{1}{n^2}$)
- Probability of 3 rebuilds: O($\frac{1}{n^3}$)
- ...

So the expected number of rebuilds is O(S), where

S = 1 · 1 n + 2 ·

+ · · · = O(1)

Since we can rebuild in O(n) time, the expected time for the

sequence of operations is O(n)

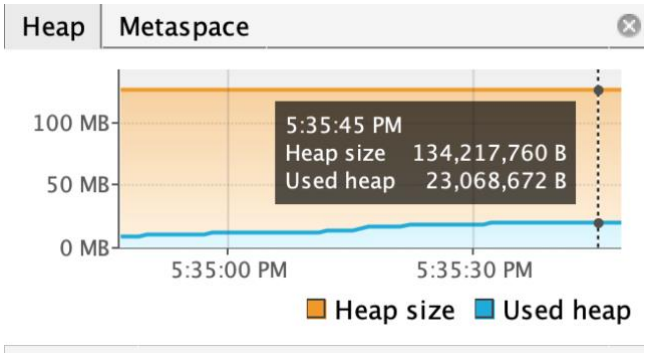
*Note: In the implemented Cuckoo Hashmap, after going in continuous loop for cycle, we have to rehash with a different Hashing function. That part is kept for exploring further

Run Time Analysis:

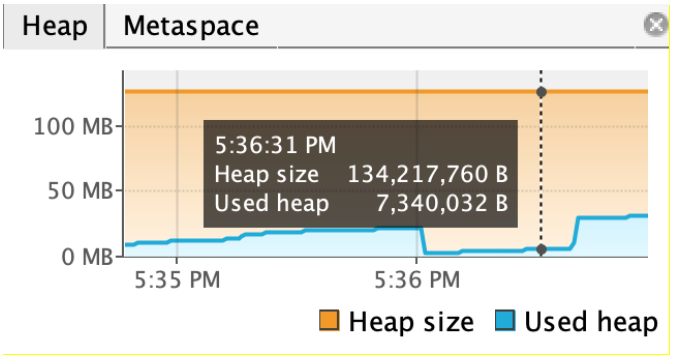
size	put	get	remove
3	272181	273	1 removal: 15850
12	397106	452	-
7	379266	1 Unsuccessful Search: 55541 1 successful search: 5864	Removal of 5 keys: 6859
100 for put /58 for get	1732775	327	-
10000 for put/ 1000 for get/ 500 for remove	44234237	400	440

Space Complexity:

Heap Size after adding 10000 elements (1000 Unique)



Heap Size after deleting 500 keys



5. Double Hashing

This implementation of Double HashMap automatically resizes based on the load factor. When load factor exceeds 0.75 it doubles the size of buckets and when it is below 0.25 it halves the size of buckets. Rehashing is performed based on the new size of buckets. This class is implemented using Templates therefore it supports any type of Key and Value.

Pseudo Code:

```
def get(k):
	h1 = h1k(k)
	h2 = h2k(k)
	i = h1
	count = 1
	do
		entry = bucket[i]
		if entry !=null && entry.key == k
			return entry.value
		i = (h1 + h2* count) % numBuckets
		count++
	while(i!=h1)
	throw RunTimeException(“Key not found”)

def put(k,v):
	h1 = hf1(k)
```

```
        h2 = hf2(k)
        i = h1
        count = 1
        do
                temp = bucket[i]
                if temp == null
                        bucket[i] = (k,v)
                        size++
                        break
                if temp.key == k
                        bucket[i] = (k,v)
                        return
                i = (h1 + h2* count) % numBuckets
                count++
        while i!=h1

def delete(k):
        h1 = h1k(k)
        h2 = h2k(k)
        i = h1
        count = 1
        do
                entry = bucket[i]
                if entry !=null && entry.key == k
                        ret = bucket.get(i).value;
                        bucket.set(i, null);
                        return ret
                i = (h1 + h2* count) % numBuckets
                count++
        while(i!=h1)
        throw RunTimeException("Key not found")
```

Algorithm Analysis:

Double hashing:

- In addition to h(k), compute a secondary hash function h2(k).
- If H[h(k)] is full, try (in order)
- H[h(k) + 1 · h2(k)], H[h(k) + 2 · h2(k)], H[h(k) + 3 · h2(k)], . . .Wrap around mod N.
- The major difference between Linear probing and double hashing is,
    - During the collision the Linear probing will have large continuous chunk
    - The worst-case search operation complexity in this case is dominated by the size of largest continuous chunk
    - On the other hand, in double hashing, we provide arithmetic progressive displacement with h2 which reduces the length of longest consecutive chunk

Why Double Hashing?

- Quadratic probing reduces the effect of clustering but introduces another problem of secondary clustering.

- While primary and secondary clustering affects the efficiency of linear and quadratic probing, clustering is completely avoided with double hashing.

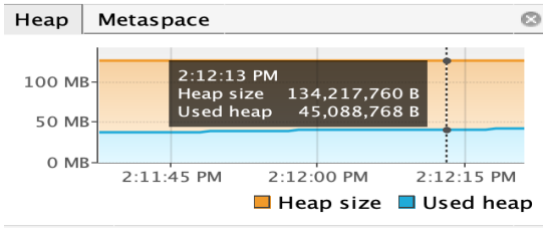
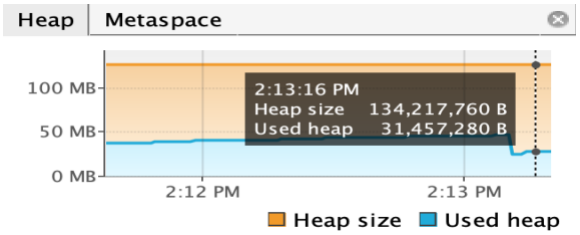
- This makes double hashing most efficient as far as clustering is concerned.

Run Time Analysis:

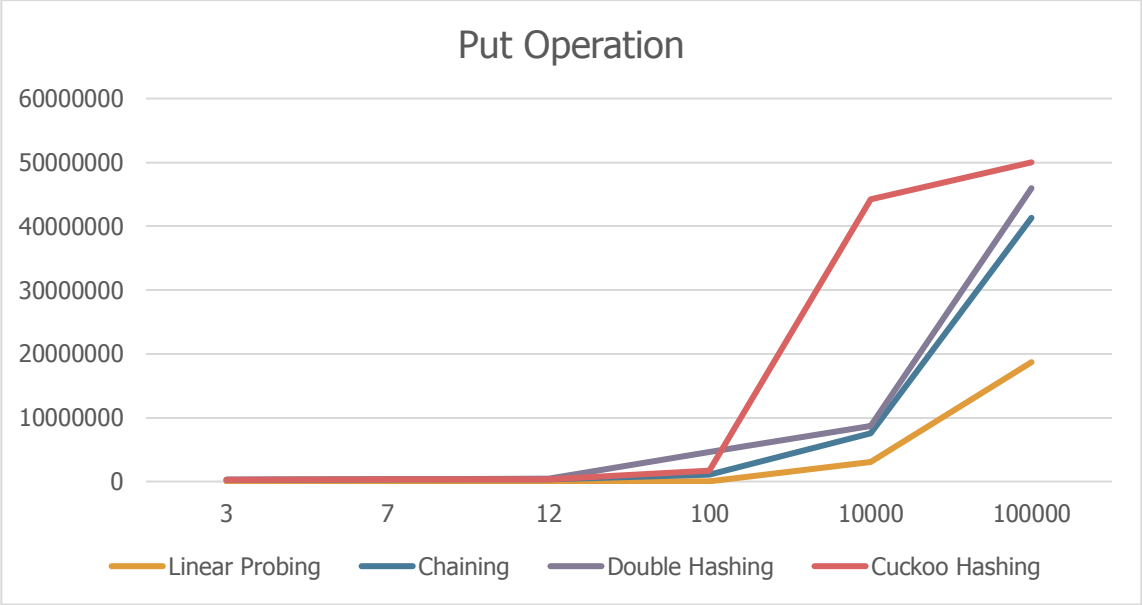
size	put	get	remove
3	276920	289	1 removal: 59548
12	429734	292	-
7	317437	1 Unsuccessful Search: 84170 1 successful search: 1897	Removal of 5 keys: 6589
100 for put /58 for get	4673251	328	-
10000 for put/ 1000 for get/ 500 for remove	8716147	416	405107

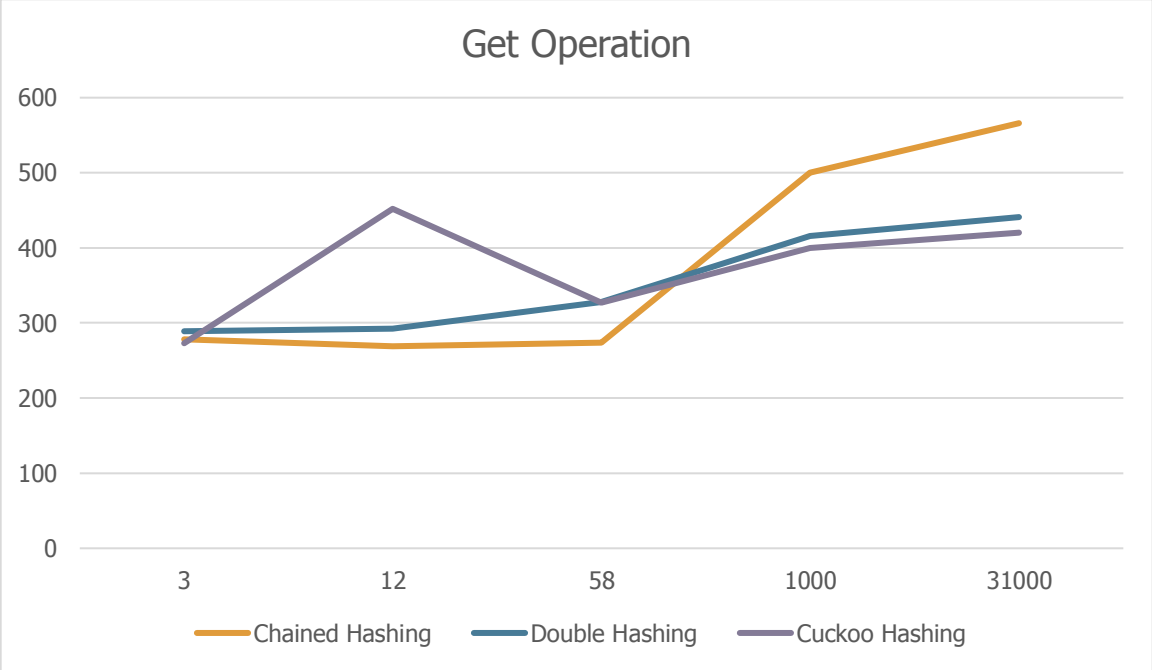
size	put	get	remove
100000 for put/ 31164 for get/ 15582 for remove	45955939	441	3149106

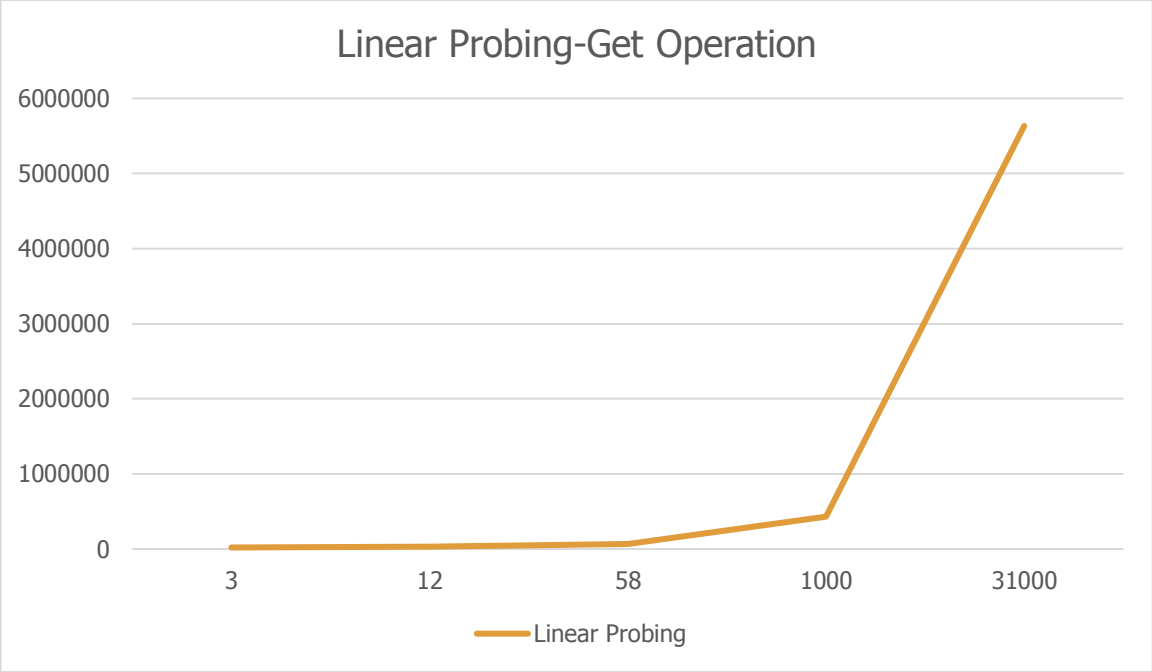
Space Complexity:

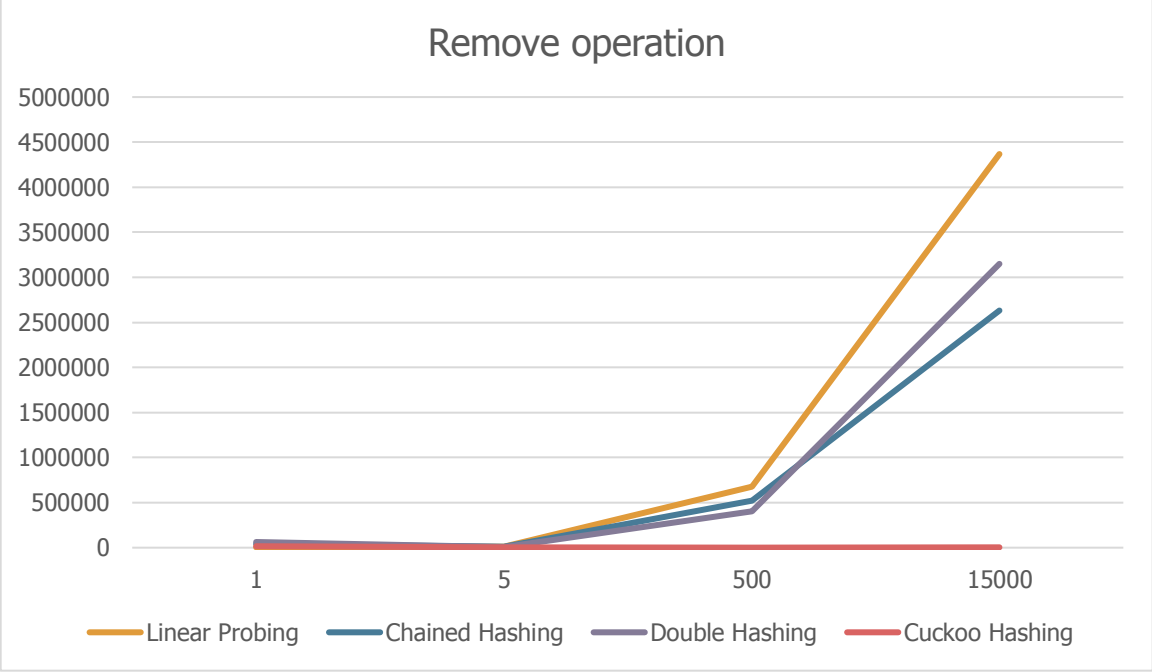
Used Heap Size after adding 100000 elements (31k Unique)	Used Heap Size after deleting 15k keys
	

6. Comparative Analysis






Linear Probing-Get Operation
6000000
5000000
4000000
3000000
2000000
1000000
0
3
12
58
1000
31000
Linear Probing


Remove operation
5000000
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1
5
500
15000
Linear Probing
Chained Hashing
Double Hashing
Cuckoo Hashing

Important observations

Linear Probing Vs Double Hashing:

- We can see that, because we have provided arithmetic progressive displacement in Double Hashing Unsuccessful search time is lesser on double hashing than on linear probing.

Chained Hashing:

- Important factor in Chained hashing's success is the rehashing and resizing when load factor is. increased beyond 0.75 and decreased beyond 0.25.
- In absence of rehashing and resizing, performance of chained hashing will degrade greatly.

Cuckoo Hashing:

- The main problem with cuckoo hashing is Cycle.
- It is very important to choose an effective hash function which would distribute the data evenly across all buckets.
- Even after choosing that, there would be continuous cycles. At that point as we have studied, we have to choose c so that we will rebuild HashMap after c log n cycles.
- In my experience, I have generated such sequence to generate cycles.
- In that case, a different effective hash function is yet to be explored.

Ideal hash maps for different use-cases

From this analysis, we can say that, based on different use cases we have to choose different algorithms.

Size of keys	Use cases	Ideal Hash map
Very huge	Efficient get, remove, but slow put	Cuckoo Hashing
<500	Efficient removes	Double Hahsing
<100	Efficient gets	Chained Hashing
Very Huge	Fast Puts	Chained Hashing
Small number of elements	Fast Puts	Linear Hashing
Small number of elements	Fast get, put and remove	Double Hashing
Medium size	Fast get, put and remove	Chained Hashing

Algorithm	Worst Case	Amortized
Linear Probing- Search	**O(n)**	**O(1)**
Linear Probing- Remove	**O(n)**	**O(1)**
Linear Probing- Put	**O(n)**	**O(1)**
Chained Hashing- Search	**O(1+ longest list in a bucket)**	**O(1)**
Chained Hashing- Remove	**O(1+ longest list in a bucket)**	**O(1)**
Chained Hashing- Put	**O(1+ longest list in a bucket)**	**O(1)**
Double Hashing- Search	**O(n)**	**O(1)**
Double Hashing- Remove	**O(n)**	**O(1)**
Double Hashing- Put	**O(n)**	**O(1)**
Cuckoo Hashing- Search	**O(n)**	**O(1)**
Cuckoo Hashing- Remove	**O(1)**	**O(1)**
Cuckoo Hashing- Put	**O(1)**	**O(1)**