

CS261P PROJECT 1 REPORT

HASHING ALGORITHMS

Contents:

Theoretical Analysis

- Definition and Explanation

- Pseudo Code

- Analysis

Experimental Analysis

- Running the test cases

- Visualization of Analysis

- General Conclusions

THEORETICAL ANALYSIS

I. Linear Hashing

The basic idea of Linear Hashing is to generate a hash value for every key in the key-value pair and check if the hash value index of the hash table is already filled. If it is not, then we simply insert the key-value pair at that index, else we check the next index. We keep on doing this till we find a position in the hash table which is not filled. We insert the key-value pair at the first unfilled position that we find. The counter wraps around the modulus so that if we traverse the array till the end we start again from the beginning.

If the hash table becomes full, no more key-value pairs can be inserted. In such a situation we need to resize our hash table and rehash all the elements so that they can be stored at the correct index in the new hash table. We use a threshold after which we resize the hash table; it is called as the load factor.

$$\text{Load Factor} = \frac{\text{number of elements in the hash table}}{\text{size of the hash table}}$$

In this hashing scheme the load factor considered is 0.5. This means that when the hash table is 50% filled we increase the size of the hash table by a factor of 2.

Removal of a key value pair can be tricky in a linear hashing scheme. First, we check whether the key-value pair at the index of the hash for a key is equal to the key that we are looking for, if that is the case we remove it and check for key-value pairs with hash values equal to the index of removed key-value pair. We then shift the key-value pairs with the same hash down by one empty space.

We can also do a lazy remove wherein we do not actually remove the element from the hash table; rather we just flag it as removed. While traversing the hash table the algorithm ignores such indices. This method is easy to implement, however as the number of elements hashed increases, we have to frequently resize the hash table since we are not actually removing the elements.

Below is the pseudo code for the three operations of this algorithm: put(), get() and remove()

a. Pseudo code:

i. *put(int hashkey, int hashvalue)*

```
def put(hashkey, hashvalue):  
    hash = h(hashkey)  
    while H[hash] is nonempty and contains a key != hashkey  
        hash = (hash + 1) mod N  
    store (hashkey, hashvalue) in H[hash]
```

ii. *get(int hashkey)*

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

iii. *remove(int hashkey)*

```
def remove(hashkey):  
    hash = h(hashkey)  
    while H[hash] is nonempty and contains a key != hashkey  
        hash = (hash + 1) mod N  
    if H[hash] is empty: exception  
    i = (hash + 1) mod N  
    while H[i] is nonempty  
        if h(H[i].hashkey) is not in the (circular) range [i+1..j]:  
            move H[i] to H[hash]  
    hash = i  
    i = i + 1  
    clear H[hash]
```

After every put() and remove() we check the load factor of the hash table. If the load factor is ≥ 0.5 for the put() operation and is ≤ 0.25 for the remove() operation. We increase and decrease respectively the size of the hash table by a factor of 2. The new hash function becomes modulo of the new table size.

b. Analysis:

i. Worst-case

Let α be the load factor of the hash table, assuming the load factor to be some maximum value, in the worst case we will have to traverse through the hash table. Therefore the worst case complexity of linear hashing is $O(N)$.

N = size of the hash table

ii. Best-case

The best-case scenario would be the element being at the index of the hash value of the key.

In such cases complexity is $O(1)$

iii. Amortized

The amortized time for all cases of a linear hashing algorithm is $O(1)$

This is the result of calculating over a sequence of operations.

II. Chained Hashing

The principle of chained hashing is similar to that of linear hashing. The only difference is how we handle collisions. Instead of going one index higher in the hash table we just link the key-value pair to the index of the hash table. In this way every index in a hash table is a collection of key value pairs.

For chained hashing, our object of a key value pair has another attribute 'nextValue' which points to the next element in the collection of key-value pairs. If the nextValue field is null for some key-value pair, it means that it is the last element in the collection of the key-value pairs in that bucket.

For the put() operation in Chained Hashing we simply add the element to the collection of key-value pairs present in the index of the hash table

For the get() operation, we need to check in the collection of key-value pairs present at the index of the hash table which is the hash value of the key. We can simply keep moving forward in the linked list till we find the key. If we do not find a matching key we return an exception.

The remove() operation in Chained Hashing is as simple as deleting a node from a linked list. Once we find the key-value pair to be removed, we simply manipulate the nexValue field of the previous and next entry.

a. Pseudo code:

i. *put(int hashkey, int hashvalue)*

```
def put(hashkey, hashvalue):  
    hash = h(hashkey)  
    if H[hash] does not contain hashkey  
        add (hashkey, hashvalue) to H[hash]  
    else  
        i = pair with NULL nextValue in list at H[hash]  
        store (hashkey, hashvalue) at the nextValue H[hash][i]
```

ii. *get(int hashkey)*

```
def get(hashkey):  
    hash = h(hashkey)  
    for every node i in H[hash]  
        if H[hash][i].getKey() == hashkey  
            return H[hash][i].getValue()  
    exception
```

iii. *remove(int hashkey)*

```
def remove(hashkey):  
    hash = h(hashkey)  
    for every node i in H[hash]  
        if H[hash][i].getKey() == hashkey  
            delete H[hash][i]  
    exception
```

With chained hashing we can have multiple key-value pairs on a given hash table index using a linked list like structure. Theoretically we can hash infinite number of key-value pairs, but as we increase the number of pairs the time to insert, remove and search will also increase exponentially.

b. Analysis

i. Average-case

From slides

$$\begin{aligned} E[\text{time / operation}] &= O(1 + E[\text{length of } H[h(k)]]) \\ &= O(1 + E[\text{number of keys colliding with } k]) \\ &= O(1 + \text{other keys } \sum_{\text{other keys } q} \text{probability}(q \text{ collides with } k)) \\ &= O(1 + (n-1) * \frac{1}{N}) \\ &= O(1 + \alpha) \end{aligned}$$

ii. Amortized

The amortized time for all cases of a chained hashing algorithm is $O(1)$
This is the result of calculating over a sequence of operations.

III. Cuckoo Hashing

The basic idea of cuckoo hashing is that there are two hash tables and hash functions. We store a new entry in the first hash table. If the slot in the first hash table has been occupied, then we shift the entry already in the first hash table to the second hash table with the second hash function and then occupy the slot.

The drawback of cuckoo hashing would be that if the hash functions are not selected properly we can encounter an infinite loop of changing the hash tables and therefore would have to rehash.

Therefore, the choice of the hash functions used is very crucial for cuckoo hashing. We should resize and rehash whenever we encounter a cycle.

The `get()` operation is very simple, we just check in the first hash table, if found return the value, else check in the second hash table. Exception is thrown if the value is not found in both the hash tables.

The `remove()` operation is also very similar to the `get()` operation. We search for the value in both the hash tables, wherever it is found, we return the value. An exception is thrown if the value does not exist in both the hash tables.

We describe the pseudo code of the algorithm below.

a. Pseudo code

i. *put(int hashkey, int hashvalue)*

```
def put(hashkey, hashvalue):  
    hashtable = 0  
    hashkey' = hashkey  
    count = 0  
    while (hashkey, hashvalue) is full:  
        (hashkey, hashvalue)  $\leftrightarrow H_t[h_t(\text{hashkey})]$   
        if hashkey = hashkey':  
            count = count + 1  
            if count == 3:  
                resize and rehash the table  
            hashtable = 1 - hashtable
```

ii. *get(int hashkey)*

```
def get(hashkey):  
    Look in both places:  $H_o[h_o(\text{hashkey})]$ ,  $H_1[h_1(\text{hashkey})]$   
    if found  
        return hashvalue  
    exception
```

iii. *remove(int hashkey)*

```
def remove(hashkey):  
    Look in both places. If hashkey is found in either  
    location,  
        clear that location.  
  
    exception
```

b. Analysis

i. Worst-case

We need to resize the table only in case of put() operation
The get() and remove operations are straightforward $O(1)$
From slides
If we use cuckoo hashing for a sequence of n operations

With probability of $1 - \theta \left(\frac{1}{n}\right)$ it works

With a probability of $\theta \left(\frac{1}{n}\right)$ we have to rebuild

ii. Amortized

The amortized time for all cases of a cuckoo hashing algorithm is $O(1)$

This is the result of calculating over a sequence of operations.

IV. Double Hashing

Reasons for selecting Double Hashing:

- It is a popular collision-resolution technique used in open-addressed hash tables because it drastically reduces clusters in the input data
- It requires fewer comparisons than other hashing schemes
- Smaller hash tables can be used to implement double hashing

Double hashing is similar to linear probing and the only difference is the interval between successive key-value pairs at the same hash table index. Here, the interval between two key-value pairs at the same hash index table is computed by using two hash functions.

Let us say that the hashed index for an input record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

$\text{index} = (\text{index} + 1 * \text{indexH}) \% \text{hashTableSize};$

In the general case, the 1 is replaced by i for successive `put()` operations

a. Pseudo code

i. *put(int hashkey, int hashvalue)*

def put(hashkey, hashvalue):

hashone = h1(hashkey)

hashtwo = h2(hashkey)

i = 1


```

while H[hashone] is nonempty and contains a key !=hashkey
hashone = (hashone + i*hashtwo) mod N
i++
store (hashkey,hashvalue) in H[hashone]

```

ii. *get(int hashkey)*

```

def get(hashkey):
hashone = h1(hashkey)
hashtwo = h2(hashkey)
while H[hashone] is nonempty and contains a key != hashkey
hashone = (hashone + i*hashtwo) mod N
if H[hashone] is nonempty:
i++
return its value
else: exception

```

iii. *remove(int hashkey)*

```

def remove(hashkey):
hashone = h1(hashkey)
hashtwo = h2(hashkey)
j = 1
while H[hashone] is nonempty and contains a key != hashkey
hashone = (hashone + j*hashtwo) mod N
j++
if H[hashone] is empty: exception
i = (hashone + j*hashtwo) mod N
while H[i] is nonempty
if h(H[i].hashkey) is not in the (circular) range [i+1..j]:
move H[i] to H[hashone]
hashone = i
i = i + 1
clear H[hashone]

```

After every put() and remove() we check the load factor of the hash table. If the load factor is ≥ 0.5 for the put() operation and is ≤ 0.25 for the remove() operation. We increase and decrease respectively the size of the hash table by a factor of 2. The new hash function becomes modulo of the new table size.

c. Analysis:

i. Worst-case

Let α be the load factor of the hash table, assuming the load factor to be some maximum value, in the worst case we will have to traverse through the hash table. Therefore the worst case complexity of linear hashing is $O(N)$.

N = size of the hash table

ii. Best-case

The best-case scenario would be the element being at the index of the hash value of the key.

In such cases complexity is $O(1)$

iii. Amortized

The amortized time for all cases of a linear hashing algorithm is $O(1)$

This is the result of calculating over a sequence of operations.


In the next section we will present an experimental analysis of all the hashing techniques mentioned above.


All the hashing algorithms have been implemented in Java


EXPERIMENTAL ANALYSIS


I. Running the test cases


The 'HashOperations.java' file is an abstract class which is implemented by all the hash function files


 ChainedHashing.java


 CuckooHashing.java


 DoubleHashing.java


 HashOperations.java


 LinearHashing.java


 testcaseChainedHashing1.java


 testcaseChainedHashing2.java


 testcaseChainedHashing3.java


 testcaseChainedHashing4.java


 testcaseChainedHashing5.java


 testcaseCuckooHashing1.java


 testcaseCuckooHashing2.java


 testcaseCuckooHashing3.java


 testcaseCuckooHashing4.java


 testcaseCuckooHashing5.java


 testcaseDoubleHashing1.java


 testcaseDoubleHashing2.java


 testcaseDoubleHashing3.java


 testcaseDoubleHashing4.java


 testcaseDoubleHashing5.java

 testcaseLinearHashing1.java

 testcaseLinearHashing2.java

 testcaseLinearHashing3.java

 testcaseLinearHashing4.java

 testcaseLinearHashing5.java

'LinearHashing.java' implements linear hashing algorithm

'ChainedHashing.java' implements chained hashing algorithm

'CuckooHashing.java' implements cuckoo hashing algorithm

'DoubleHashing.java' implements double hashing algorithm

There are 5 test cases for each algorithm

The test cases are explained below

There is also a Makefile included to build the program
run the 'make' command on terminal to build the program

Any of the test case files can be run on the terminal using the command

java <testcasename>

example: *java testcaseLinearHashing1* will run the first test case of Linear Hashing

The first test case of each algorithm checks for the correctness of the algorithm

The second test case of Linear and Double Hashing implements the algorithms for 1000 operations of each kind of operations put(), get() and remove() for a load factor of 0.5

The second test case of Chained and Cuckoo Hashing implements the algorithms for 100 operations of each kind of operations put(), get() and remove() for a table size of 50

The third test case of Linear and Double Hashing implements the algorithms for 1000 operations of each kind of operations put(), get() and remove() for a load factor of 0.75

The third test case of Chained and Cuckoo Hashing implements the algorithms for 1000 operations of each kind of operations put(), get() and remove() for a table size of 500

The fourth test case of Linear and Double Hashing implements the algorithms for 1000 operations of each kind of operations put(), get() and remove() for a load factor of 1.00

The fourth test case of Chained and Cuckoo Hashing implements the algorithms for 10000 operations of each kind of operations put(), get() and remove() for a table size of 5000

The fifth test case of all algorithms calculates the worst-case time for each operation

```
<terminated> testcaseLinearHashing1 [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
No Key Exists

Index of HashTable: 0 - Key: NULL Value: NULL
Index of HashTable: 1 - Key: NULL Value: NULL
Index of HashTable: 2 - Key: 50 Value: 1
Index of HashTable: 3 - Key: NULL Value: NULL
Index of HashTable: 4 - Key: NULL Value: NULL
Index of HashTable: 5 - Key: 101 Value: 5
Index of HashTable: 6 - Key: NULL Value: NULL
Index of HashTable: 7 - Key: NULL Value: NULL
Index of HashTable: 8 - Key: NULL Value: NULL
Index of HashTable: 9 - Key: 73 Value: 6
Index of HashTable: 10 - Key: NULL Value: NULL
Index of HashTable: 11 - Key: NULL Value: NULL
Index of HashTable: 12 - Key: 700 Value: 2
Index of HashTable: 13 - Key: 76 Value: 3
Index of HashTable: 14 - Key: 92 Value: 5
Index of HashTable: 15 - Key: NULL Value: NULL
Value of key just searched: 3
```

The output of 'testcaseLinearhashing1.java'

```

<terminated> testcaseChainedHashing1 [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
No Key Exists

Index of HashTable: 0 - Key: NULL Value: NULL
Index of HashTable: 1 - Key: NULL Value: NULL
Index of HashTable: 2 - Key: 50 Value: 1
Index of HashTable: 3 - Key: NULL Value: NULL
Index of HashTable: 4 - Key: NULL Value: NULL
Index of HashTable: 5 - Key: 101 Value: 5
Index of HashTable: 6 - Key: NULL Value: NULL
Index of HashTable: 7 - Key: NULL Value: NULL
Index of HashTable: 8 - Key: NULL Value: NULL
Index of HashTable: 9 - Key: 73 Value: 6
Index of HashTable: 10 - Key: NULL Value: NULL
Index of HashTable: 11 - Key: NULL Value: NULL
Index of HashTable: 12 - Key: 700 Value: 2
Index of HashTable: 12.1 - Key: 76 Value: 3
Index of HashTable: 12.2 - Key: 92 Value: 5
Index of HashTable: 13 - Key: NULL Value: NULL
Index of HashTable: 14 - Key: NULL Value: NULL
Index of HashTable: 15 - Key: NULL Value: NULL
Value of key just searched: 3

```

The output of 'testcaseChainedHashing1.java'

```

<terminated> testcaseCuckooHashing1 [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
HashTable: 1 - Index of HashTable: 0 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 1 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 2 Key: 50 Value: 1
HashTable: 1 - Index of HashTable: 3 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 4 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 5 Key: 101 Value: 5
HashTable: 1 - Index of HashTable: 6 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 7 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 8 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 9 Key: 73 Value: 6
HashTable: 1 - Index of HashTable: 10 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 11 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 12 Key: 92 Value: 5
HashTable: 1 - Index of HashTable: 13 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 14 Key: NULL Value: NULL
HashTable: 1 - Index of HashTable: 15 Key: NULL Value: NULL

HashTable: 2 - Index of HashTable: 0 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 1 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 2 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 3 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 4 Key: 76 Value: 3
HashTable: 2 - Index of HashTable: 5 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 6 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 7 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 8 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 9 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 10 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 11 Key: 700 Value: 2
HashTable: 2 - Index of HashTable: 12 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 13 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 14 Key: NULL Value: NULL
HashTable: 2 - Index of HashTable: 15 Key: NULL Value: NULL
Value of key just searched: 3

```

The output of 'testcaseCuckooHashing1.java'

```
<terminated> testcaseDoubleHashing1 [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe
```

```
No Key Exists
```

```
Index of HashTable: 0 - Key: NULL Value: NULL
```

```
Index of HashTable: 1 - Key: 101 Value: 5
```

```
Index of HashTable: 2 - Key: 50 Value: 1
```

```
Index of HashTable: 3 - Key: NULL Value: NULL
```

```
Index of HashTable: 4 - Key: NULL Value: NULL
```

```
Index of HashTable: 5 - Key: 85 Value: 4
```

```
Index of HashTable: 6 - Key: NULL Value: NULL
```

```
Index of HashTable: 7 - Key: NULL Value: NULL
```

```
Index of HashTable: 8 - Key: 92 Value: 5
```

```
Index of HashTable: 9 - Key: 73 Value: 6
```

```
Index of HashTable: 10 - Key: NULL Value: NULL
```

```
Index of HashTable: 11 - Key: NULL Value: NULL
```

```
Index of HashTable: 12 - Key: 700 Value: 2
```

```
Index of HashTable: 13 - Key: 76 Value: 3
```

```
Index of HashTable: 14 - Key: NULL Value: NULL
```

```
Index of HashTable: 15 - Key: NULL Value: NULL
```

```
Value of key just searched: 3
```

The output of 'testcaseDoubleHashing1.java'

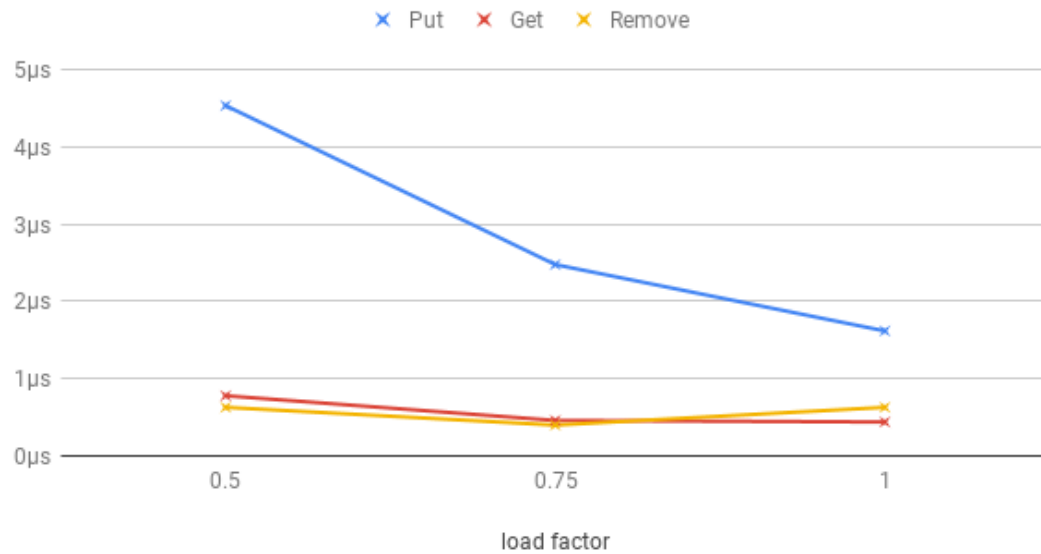
II. Visualization of Analysis

Several graphs have been plotted to compare the load factor and number of inputs to the execution time per operation

We can observe some interesting trends

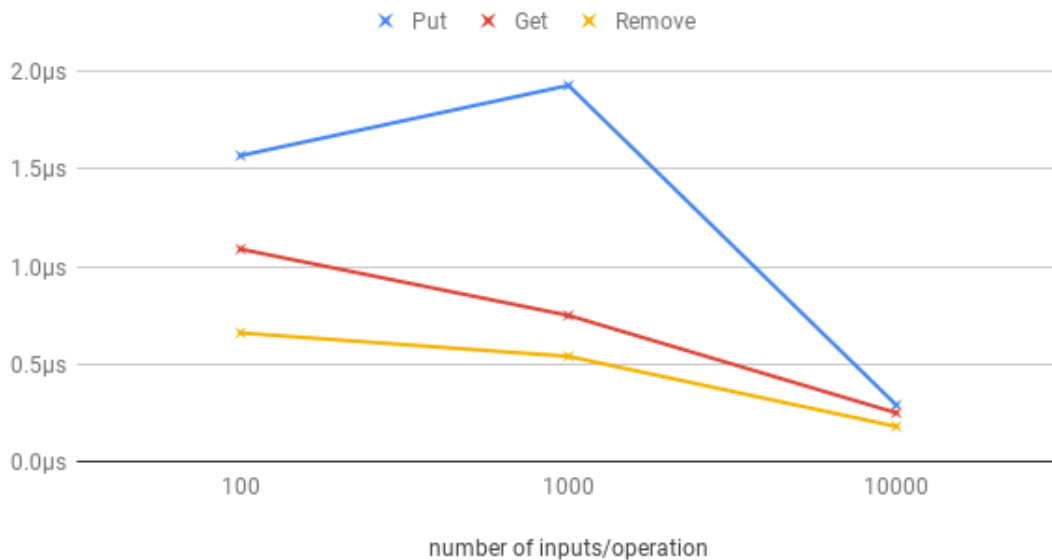
The graphs are presented below

Linear Hashing: Comparison at load factors



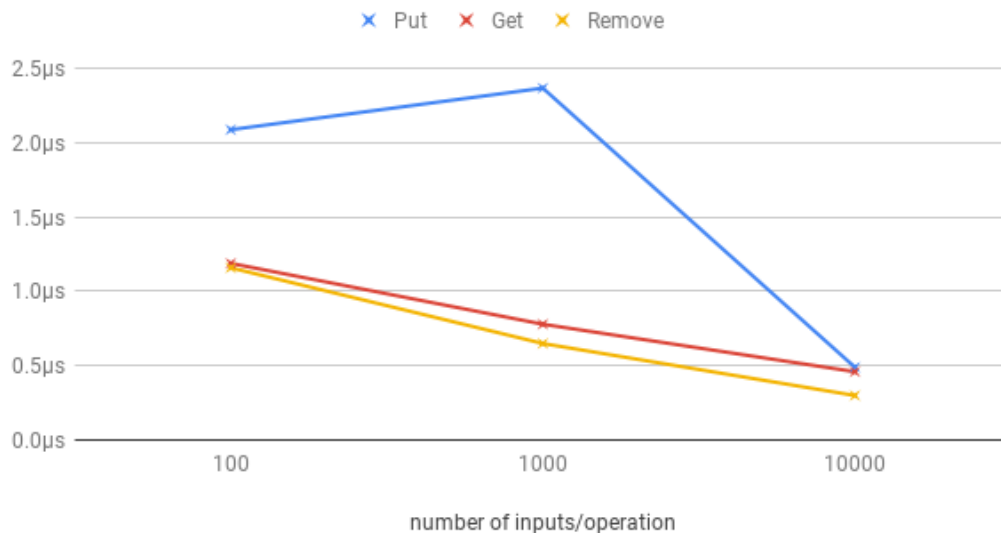
We can see that at a load factor of 0.75 for linear hashing the execution times for each operation of put(), get() and remove() are the lowest
Therefore, the load factor of 0.75 is best

Chained Hashing: comparison of input sizes



We can see that as the number of input values increase the execution times decrease
Therefore, this algorithm works best for larger inputs
We have considered the table size to be constant at half the number of inputs

Cuckoo Hashing: Comparison of input sizes



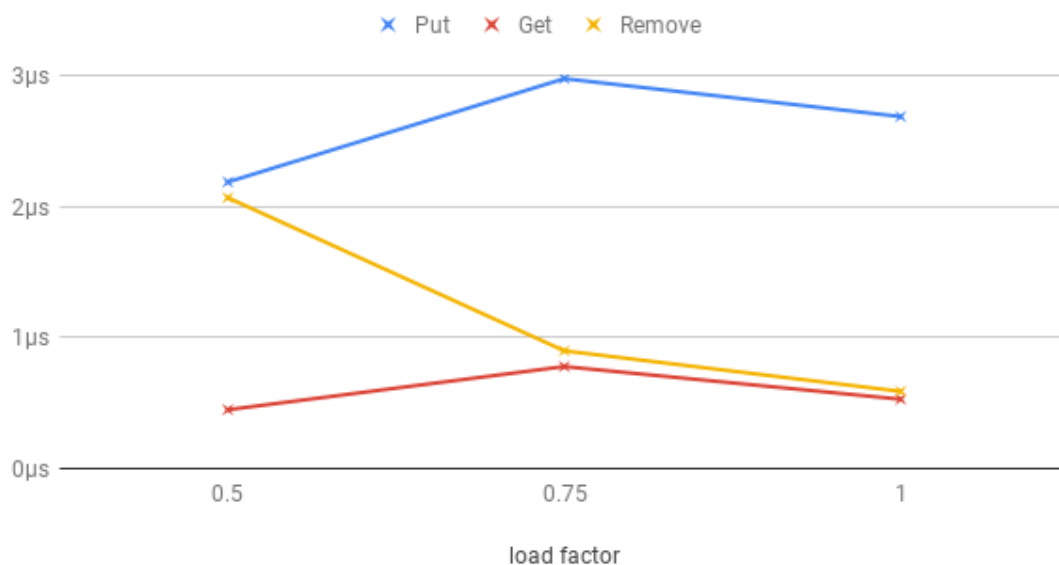
We can see that as the number of input values increase the execution times decrease

Therefore, this algorithm works best for larger inputs

We have considered the table size to be constant at half the number of inputs

Insertions take massive amounts of time compared to other operations in certain cases due to the fact that when cycles are detected the table is resized

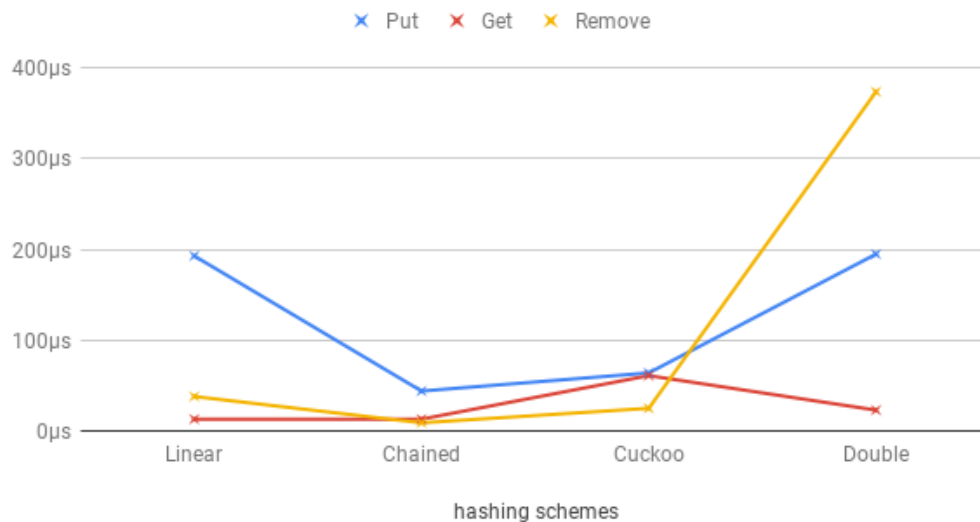
Double Hashing: Comparison at load factors



We can see that at a load factor of 0.50 for double hashing the execution times for most of the operations are the lowest

Therefore, the load factor of 0.50 is best

Worst-case time per operation comparison



This graph compares the worst-case time per operation of put(), get() and remove() for all the hashing algorithms.

We can see that Chained Hashing proves to give the lowest times compared to all the other approaches. This is the case because the size of input was 1000. Therefore at small input sizes the Chained Hashing works well, but for larger input sizes it will be slower.

III. General Conclusions

- All the hashing algorithms have an Amortized time of $O(1)$ for all the operations, but in certain specific cases some algorithms perform better than others
- Chained Hashing is a good approach when the input size is small
- Varying the load factor for hash algorithms can also change the way the algorithm works, in general a load factor between 0.5 – 0.75 should be optimal in most cases
- For the remove() operation a load factor of 0.25 works out well enough
- In general hashing is a great way to store data if you want a constant amortized time. Searching is very fast when the data is hashed and therefore hash tables are widely used in industry applications.