**Hashing:**

Hashing is one of the efficient technique used to perform dictionary operations such as insert, search, delete. This technique uses hash tables to implement. Hash table is usually an array with index and stores the values according the values generated by the hash function.

This hashing technique is implemented with the help of python programming language.

**Implementation:**

Initially, the hash table of size m is declared and set to empty with the help of init() function. While inserting values into the hash table, we check if there are any keys in the table and insert if it is null. If there is a value existing in that position, the key is moved to the next position. In order to insert keys at certain positions, we evaluate hash functions.

Linear probing hash function:

Hash value= key % table size

New hash value = (hash value+1) % table size.

Quadratic probing:

Hash value= key % table size

New hash value = ((hash value+1) \*\*2) % table size.

Double probing:

Hash value= key % table size

Hash value1 = (hash value+1) % table size.

New hash value = (hash value + hash value 1) % table size

**Observations:**

For some random set of integers generated between 1 to 10,000, we have the following observations for different values of m.

|  |  |  |  |
| --- | --- | --- | --- |
| Table size(m) | Linear hashing | Quadratic hashing | Double hashing |
| 1223 | 4422 | 3138 | 3126 |
| 1831 | 1377 | 1376 | 1339 |
| 2447 | 1195 | 1260 | 1189 |

From the above table generated after evaluating the number of probes for each value of m indicates the following:

1. linear hashing has the highest number of collisions or the probes.
2. Double has the least number of probes.
3. Also, during the execution of the program, different number of probes are generated when the same table size and the same hashing technique are used and executed multiple times. This is because, the library used from the python library for generating the random numbers, generates different set of numbers for different table sizes.
4. The library “random” generates one set of random integers between the given range and stores them in the given table size. When the table size is given same continuously during multiple executions, the table takes the values that are stored in the table during the first execution.

**Appendices:**

Source code:

import random as rand

#import math

class HashTable:

def \_\_init\_\_(hasht, m, method):

hasht.size = m

hasht.slots = [None] \* hasht.size

hasht.data = [None] \* hasht.size

hasht.operations = 0

hasht.method = method

def put(hasht,key,data):

hashvalue = hasht.hashfunction(key,len(hasht.slots))

if hasht.slots[hashvalue] == None:

hasht.slots[hashvalue] = key

hasht.data[hashvalue] = data

hasht.operations+= 1

else:

if hasht.slots[hashvalue] == key:

hasht.data[hashvalue] = data #replace

hasht.operations+= 1

else:

nextslot = hasht.rehash(hashvalue,len(hasht.slots))

while hasht.slots[nextslot] != None and \

hasht.slots[nextslot] != key:

hasht.operations+= 1

nextslot = hasht.rehash(nextslot,len(hasht.slots))

if hasht.slots[nextslot] == None:

hasht.slots[nextslot]=key

hasht.data[nextslot]=data

hasht.operations+= 1

else:

hasht.data[nextslot] = data #replace

hasht.operations+= 1

def hashfunction(hasht,key,size):

return key%size

def rehash(hasht,prevhash,size):

if hasht.method == 1:

return (prevhash+1) %size

if hasht.method == 2:

return ((prevhash+1) \*2) %size

if hasht.method == 3:

hash2 = (prevhash+1) %size

return (prevhash+hash2) %size

def \_\_setitem\_\_ (hasht,key,data):

hasht.put (key,data)

m = int(input("Enter The size of the hash table for the experiment 'm' :\n"))

method = int (input("Choose the method for hashing:\n1.Linear Hashing\n2.Quadratic Hashing\n3.Double Hashing\n"))

H=HashTable (m,method)

for i in range (1,1100):

H[rand.randint (1,10001)]= 1

Print ("Number Of Operations taken for the above chosen method and array size is:\n",H.operations)