**PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE - 411043 Department of Computer Engineering** 

**S.No.-27, Pune Satara Road, Dhankawadi, Pune-411043**

**Data Structures and Algorithms Laboratory**

**Batch-IV (H4)**

**Name: Rushikesh Vidye**

**Roll No.: 21471**

**Class: SE4**

**Assignment No. 4**

**Title:** Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers (use linear probing with replacement and without replacement)

**Software Requirement:**

a) OS : Microsoft Windows 10.

b) Browser: Google Chrome.

c) VS Code.

**Hardware Requirement:**

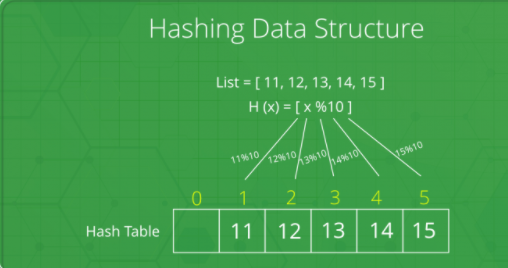
a) Processor: Intel Core i5-8265U.

b) Ram: 8 GB DDR4 2800Mhz.

**Theory:**

Hashing is a technique or process of mapping keys, values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value  at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



**What is Collision?**   
Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

**How to handle Collisions?**   
There are mainly two methods to handle collision:   
1) Separate Chaining   
2) Open Addressing   
In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post.

**Separate Chaining:**   
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

**Advantages:**   
1) Simple to implement.   
2) Hash table never fills up, we can always add more elements to the chain.   
3) Less sensitive to the hash function or load factors.   
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

**Disadvantages:**   
1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.   
2) Wastage of Space (Some Parts of hash table are never used)   
3) If the chain becomes long, then search time can become O(n) in the worst case.   
4) Uses extra space for links.

**Open Addressing**   
Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as “deleted”.   
The insert can insert an item in a deleted slot, but the search does not stop at a deleted slot.

* **Algorithm**

Step 1:-Start

Step 2:-Display the options on screen

Step 3:-Take the input of option from user

Step 4:-For 1 Option Insert Function:

Take the input of Phone number from user and insert in hash table

Step 5:-For 2 Option Display Function:

Display all the number insert in the hash table as per the count of digits in     number

Step 6:-For 3 Option Search Function:

Enter the number to be searched

Check if the entered number is present in hash table

Case 1: Display Element found if the element is present

Case 2:Display element Not found if the element is not present in hash table

Step 7:-For 4 Option Delete Function:

Take the input for the number to be deleted from the Hash table

Check if entered element is present in hash table

Case 1: If the element is present Delete the element from the table and       display element deleted

Case 2: If element is not present then display element not found.

Step 8: For 5 Option Exit Function:

Terminate the execution of program

Step 9: Stop

**Time Complexity:**

|  |  |  |
| --- | --- | --- |
| Sr.No | Methods | Complexity |
| 1 | Insert() | O(1) |
| 2 | Display() | O(n) |
| 3 | Search() | O(n) |
| 4 | Delete() | O(n) |

**Test Cases:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr.no | Input | Output | Result |
| 1 | Enter mobile number:  9898989898 | Element added | pass |
| 2 | Search 9898989898 | Element found at index 8 | pass |
| 3 | Delete 9898989898 | Element deleted from list | pass |

**Conclusion:**

Hence we learnt the implementation of hash table and also successfully implemented collision handling techniques.

**Code:**

|  |
| --- |
| class Hash:      def \_\_init\_\_(self):          self.size = int (input("Enter the Size of Hash Table : "))          self.HashTable = list(0 for i in range(self.size))          self.num\_of\_number = 0          self.comparison = 0      def isTableFull(self):          if self.num\_of\_number == self.size:              return True          else:              return False      def HashFun(self,element):          return element%self.size      def InsertElement\_Linear(self,element):          if self.isTableFull():              print("Hash Table Is Full")              return False          OccupiedStatus = False          position = self.HashFun(element)          if self.HashTable[position] == 0:              self.HashTable[position] = element              print("Telephone Number "+str(element) +" at position "+str(position))              OccupiedStatus = True          else:              print("Collision has occurred for Telephone Number "+str(element)+" at index "+str(position))              position = self.LinearProbing(element,position)              self.HashTable[position] = element              OccupiedStatus = True              self.num\_of\_number += 1          return OccupiedStatus      def LinearProbing(self,element,position):          while self.HashTable[position] != 0:              position += 1              if position >= self.size:                  position = 0          return position      def Search(self,element):          found = False          position = self.HashFun(element)          self.comparison += 1          if self.HashTable[position] == element:              return position              isFound = True          else :              temp = position - 1              while position < self.size:                  if self.HashTable[position] != element:                      position += 1                      self.comparison += 1                  else:                      return position              position = temp              while position > 0:                  if self.HashTable[position] != element:                      position -= 1                      self.comparison += 1                  else :                      return position              if not found:                  print("Element Not Found")                  return False      def display(self):          print("---------------------------")          print("Position\tTelephone Number\n")          for i in range(self.size):              print("\t"+str(i) + " -->"+str(self.HashTable[i]))  # main  hash\_object = Hash()  hash\_object.InsertElement\_Linear(79)  hash\_object.InsertElement\_Linear(29)  hash\_object.InsertElement\_Linear(85)  hash\_object.InsertElement\_Linear(40)  hash\_object.InsertElement\_Linear(33)  hash\_object.InsertElement\_Linear(77)  hash\_object.InsertElement\_Linear(67)  hash\_object.InsertElement\_Linear(66)  hash\_object.InsertElement\_Linear(75)  hash\_object.InsertElement\_Linear(88)  hash\_object.display()  print("Position of 88 is : " + str(hash\_object.Search(88)))  print("Number of Comparisons done while searching : "+str(hash\_object.comparison)) |

**Output:**

|  |
| --- |
| Enter the Size of Hash Table : 10  0 --> Hello --> Kanobi  1 --> There  2 --> General  3  4  5  6  7  8  9  Kanobi  Key {} Deleted 20  0 --> Hello  1 --> There  2 --> General  3  4  5  6  7  8  9 |