**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. 5**

**Title:** Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique. Standard Operations: Insert(key, value), Find(key), Delete(key).

**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.

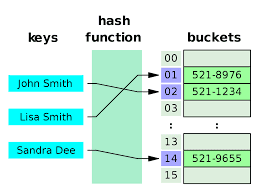
**Theory:**

**ADT(Abstract Data Type):**

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations. ... It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

**Hashing:**

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) format.

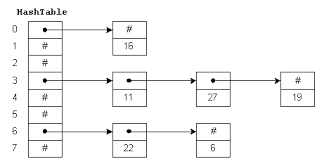
****

**Hash Function:**

* A fixed process converts a key to a hash key is known as a **Hash Function.**
* This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
* Hash value represents the original string of characters, but it is normally smaller than the original.
* It transfers the digital signature and then both hash value and signature are sent to the

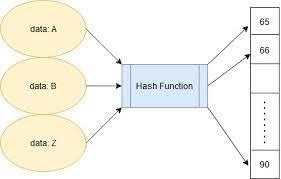
Hash Table:

* Hash table or hash map is a data structure used to store key-value pairs.
* It is a collection of items stored to make it easy to find them later.
* It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
* It is an array of list where each list is known as bucket.



**Collisions:**

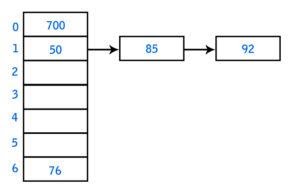
When one or more hash values compete with a single hash table slot, collisions occur. To resolve this, the next available empty slot is assigned to the current hash value. The most common methods are open addressing, chaining, probabilistic hashing, perfect hashing and coalesced hashing technique.

****

**Collision Handling Techniques:**

**a) Chaining:**

This technique implements a [linked list](https://www.vibrantpublishers.com/2019/11/16/linked-list-operations/) and is the most popular collision resolution techniques. Below is an example of a chaining process.



**b) Open Addressing:**

This technique depends on space usage and can be done with linear or quadratic probing techniques. As the name says, this technique tries to find an available slot to store the record. It can be done in one of the 3 ways –

**Linear probing**:

Here,In, open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.The next probe interval is fixed to 1. It supports best caching but miserably fails at clustering.

**Quadratic probing**:

the Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index. probe distance is calculated based on the quadratic equation. This is considerably a better option as it balances clustering and caching.

**Double Hashing:**

 Here, Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

**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 key and value:  10 “Hello There”  11 “General”  20 “Kanobi” | Element added | pass |
| 2 | Search 20 | Element found at index 0 | pass |
| 3 | Delete 20 | Element deleted from list | pass |

**Conclusion:**

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

**Code:**

|  |
| --- |
| class Node:      def \_\_init\_\_(self,key,value):          self.key = key          self.value = value  class Hash:      def \_\_init\_\_(self):          self.size = int (input("Enter the Size of Hash Table : "))          self.HashTable = [[]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,key):          return key % self.size      def InsertElement(self,Node):          if self.isTableFull():              print("Hash Table Is Full")              return False          position = self.HashFun(Node.key)          self.HashTable[position].append(Node)      def search(self,key):          hash\_key = self.HashFun(key)          bucket = self.HashTable[hash\_key]          for i in bucket:              if i.key == key:                  return i.value      def delete(self,key):          hash\_key = self.HashFun(key)          key\_exist = False          bucket = self.HashTable[hash\_key]          for i in bucket:              if i.key == key:                  key\_exist = True                  break          if key\_exist:              bucket.remove(i)              print("Key {} Deleted",format(key))          else:              print("Key {} Not Found",format(key))      def display(self):          print("---------------------------")          for i in range(len(self.HashTable)):              print(i, end = " ")              for j in self.HashTable[i]:                  print("-->", end = " ")                  print(j.value, end = " ")              print()  # main  hash\_object = Hash()  hash\_object.InsertElement(Node(10,"Hello"))  hash\_object.InsertElement(Node(11,"There"))  hash\_object.InsertElement(Node(12,"General"))  hash\_object.InsertElement(Node(20,"Kanobi"))  hash\_object.display()  print(hash\_object.search(20))  hash\_object.delete(20)  hash\_object.display() |

**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 |