Practice 03:

Implementation of hashing using quadratic probing approach

Code:

class hashTable:

def \_\_init\_\_(self):

self.size = int(input("Enter the Size of the hash table : "))

# initialize table with all elements 0

self.table = list(0 for i in range(self.size))

self.elementCount = 0

self.comparisons = 0

# method that checks if the hash table is full or not

def isFull(self):

if self.elementCount == self.size:

return True

else:

return False

def hashFunction(self, element):

return element % self.size

# method to resolve collision by quadratic probing method

def quadraticProbing(self, element, position):

posFound = False

limit = 50

i = 1

while i <= limit:

newPosition = position + (i\*\*2)

newPosition = newPosition % self.size

if self.table[newPosition] == 0:

posFound = True

break

else:

i += 1

return posFound, newPosition

# method that inserts element inside the hash table

def insert(self, element):

# checking if the table is full

if self.isFull():

print("Hash Table Full")

return False

isStored = False

position = self.hashFunction(element)

# checking if the position is empty

if self.table[position] == 0:

# empty position found , store the element and print the message

self.table[position] = element

print("Element " + str(element) + " at position " + str(position))

isStored = True

self.elementCount += 1

# collision occured hence we do linear probing

else:

print("Collision has occured for element " + str(element) + " at position " + str(position) + " finding new Position.")

isStored, position = self.quadraticProbing(element, position)

if isStored:

self.table[position] = element

self.elementCount += 1

return isStored

# method that searches for an element in the table

def search(self, element):

found = False

position = self.hashFunction(element)

self.comparisons += 1

if(self.table[position] == element):

return position

# if element is not found at position returned hash function then we search element using quadratic probing

else:

limit = 50

i = 1

newPosition = position

# start a loop to find the position

while i <= limit:

# calculate new position by quadratic probing

newPosition = position + (i\*\*2)

newPosition = newPosition % self.size

self.comparisons += 1

# if element at newPosition is equal to the required element

if self.table[newPosition] == element:

found = True

break

elif self.table[newPosition] == 0:

found = False

break

else:

# as the position is not empty increase i

i += 1

if found:

return newPosition

else:

print("Element not Found")

return found

# method to remove an element from the table

def remove(self, element):

position = self.search(element)

if position is not False:

self.table[position] = 0

print("Element " + str(element) + " is Deleted")

self.elementCount -= 1

else:

print("Element is not present in the Hash Table")

return

# method to display the hash table

def display(self):

print("\n")

for i in range(self.size):

print(str(i) + " = " + str(self.table[i]))

print("The number of element is the Table are : " + str(self.elementCount))

# main function

table1 = hashTable()

# storing elements in table

table1.insert(12)

table1.insert(26)

table1.insert(31)

table1.insert(17)

table1.insert(90)

table1.insert(28)

table1.insert(88)

table1.insert(40)

table1.insert(77) # element that causes collision at position 0

# displaying the Table

table1.display()

print()

# printing position of elements

print("The position of element 31 is : " + str(table1.search(31)))

print("The position of element 28 is : " + str(table1.search(28)))

print("The position of element 90 is : " + str(table1.search(90)))

print("The position of element 77 is : " + str(table1.search(77)))

print("The position of element 1 is : " + str(table1.search(1)))

print("\nTotal number of comaprisons done for searching = " + str(table1.comparisons))

print()

table1.remove(90)

table1.remove(12)

table1.display()