­­WUPS Routing Program

# Identify the algorithm that will be used to create a program to deliver the packages and meets all requirements specified in the scenario.

We’ll use classes for this routing problem we’ll have classes for each of the entities, like the driver, the truck, the package, the supervisor. Each of these classes will have some private and public members and they will also have some constant members. The Public members will have the attributes which are allowed to be visible to the people or which might be useful to real human beings. And the constant members in the classes identify the limits of the particular class, like there can be 16 packages in a truck etc.

# Code

## Main

**from** **chaininghashtable** **import** ChainingHashTable

**from** **linearprobinghashtable** **import** LinearProbingHashTable

**from** **doublehashinghashtable** **import** DoubleHashingHashTable

**from** **quadraticprobinghashtable** **import** QuadraticProbingHashTable

*# Main program to test HashTable classes*

keys = [ 35, 0, 22, 94, 220, 110, 4 ]

chaining = ChainingHashTable()

linear\_probing = LinearProbingHashTable()

qp1 = QuadraticProbingHashTable()

double\_hashing = DoubleHashingHashTable()

**for** key **in** keys:

chaining.insert(key)

linear\_probing.insert(key)

qp1.insert(key)

double\_hashing.insert(key)

*# Show tables after inserts.*

**print** ("Chaining: initial table:")

**print** (chaining)

**print**()

**print** ("Linear Probing: initial table:")

**print** (linear\_probing)

**print**()

**print** ("Quadratic Probing: initial table:")

**print** (qp1)

**print**()

**print** ("Double Hashing: initial table:")

**print** (double\_hashing)

**print**()

*# Show tables after removing item 0*

**print** ("=======================================")

chaining.remove(0)

linear\_probing.remove(0)

qp1.remove(0)

double\_hashing.remove(0)

**print** ("Chaining: after removing 0:")

**print**(chaining)

**print**()

**print** ("Linear Probing: after removing 0:")

**print**(linear\_probing)

**print**()

**print** ("Quadratic Probing: initial table:")

**print** (qp1)

**print**()

**print** ("Double Hashing: after removing 0:")

**print**(double\_hashing)

## Package

**class** **Package**:

**def** \_\_init\_\_(self, id, street\_address, city, zip\_code, weight, deadline, status):

self.id = id

self.street\_address = street\_address

self.city = city

self.zip\_code = zip\_code

self.weight = weight

self.deadline = deadline

self.status = status

**def** \_\_hash\_\_(self):

**return** self.id

## Hash Table

**import** **abc**

**class** **HashTable**(abc.ABC):

@abc.abstractmethod

**def** insert(self, item):

**pass**

@abc.abstractmethod

**def** search(self, key):

**pass**

@abc.abstractmethod

**def** remove(self, key):

**pass**

## Empty Bucket

*# class to represent an empty bucket*

**class** **EmptyBucket**:

**pass**

## Delivery Status

IN\_TRANSIT = 'IN\_TRANSIT'

DELIVERED = 'DELIVERED'

HASNT\_LEFT\_HUB = 'HASNT\_LEFT\_HUB'

## Chaining Hash Table

**from** **hashtable** **import** HashTable

*# HashTable class using chaining.*

**class** **ChainingHashTable**(HashTable):

*# Constructor with optional inital capacity parameter.*

*# Assigns all buckets with an empty list.*

**def** \_\_init\_\_(self, inital\_capacity=10):

*# initialize the hash table with empty bucket list entries.*

self.table = []

**for** i **in** range(inital\_capacity):

self.table.append([])

*# Inserts a new item into the hash table.*

**def** insert(self, item):

*# get the bucket list where this item will go.*

bucket = hash(item) % len(self.table)

bucket\_list = self.table[bucket]

*# insert the item to the end of the bucket list.*

bucket\_list.append(item)

*# Searches for an item with matching key in the hash table.*

*# Returns the item if found, or None if not found.*

**def** search(self, key):

*# get the bucket list where this key would be.*

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

*# search for the key in the bucket list*

**if** key **in** bucket\_list:

*# find the item's index and return the item that is in the bucket list.*

item\_index = bucket\_list.index(key)

**return** bucket\_list[item\_index]

**else**:

*# the key is not found.*

**return** None

*# Remove an item with matching key from the hash table.*

**def** remove(self, key):

*# get the bucket list where this item will be removed from.*

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

*# remove the item from the bucket list if it is present.*

**if** key **in** bucket\_list:

bucket\_list.remove(key)

*# Overloaded string conversion method to create a string*

*# representation of the entire hash table. Each bucket is shown*

*# as a pointer to a list object.*

**def** \_\_str\_\_(self):

index = 0

s = " --------**\n**"

**for** bucket **in** self.table:

s += "**%2d**:| ---|-->**%s\n**" % (index, bucket)

index += 1

s += " --------"

**return** s

## Linear Probing Hash Table

**from** **hashtable** **import** HashTable

**from** **emptybucket** **import** EmptyBucket

*# HashTable class definition using linear probing*

**class** **LinearProbingHashTable**(HashTable):

*# Constructor with optional initial capacity. All buckets are*

*# assigned with an EmptyBucket() instance called self.EMPTY\_SINCE\_START.*

**def** \_\_init\_\_(self, initial\_capacity=10):

*# Special constants to be used as the two types of empty buckets.*

self.EMPTY\_SINCE\_START = EmptyBucket()

self.EMPTY\_AFTER\_REMOVAL = EmptyBucket()

*# Initialize all the table buckets to be EMPTY\_SINCE\_START.*

self.table = [self.EMPTY\_SINCE\_START] \* initial\_capacity

**def** insert(self, item):

bucket = hash(item) % len(self.table)

buckets\_probed = 0

**while** buckets\_probed < len(self.table):

*# if the bucket is empty, the item can be inserted at the at index.*

**if** type(self.table[bucket]) **is** EmptyBucket:

self.table[bucket] = item

**return** True

*# the bucket was occupied, continue probing to next index in table.*

bucket = (bucket + 1) % len(self.table)

buckets\_probed = buckets\_probed + 1

*# the entire table was full and the key could not be inserted.*

**return** False

**def** remove(self, key):

bucket = hash(key) % len(self.table)

buckets\_probed = 0

**while** self.table[bucket] **is** **not** self.EMPTY\_SINCE\_START **and** buckets\_probed < len(self.table):

**if** self.table[bucket] == key:

self.table[bucket] = self.EMPTY\_AFTER\_REMOVAL

*# the bucket was occupied (now or previously), so continue probing.*

bucket = (bucket + 1) % len(self.table)

buckets\_probed = buckets\_probed + 1

*# Search for an item with a matching key in the hash table. Returns the*

*# item if found, or None if not found.*

**def** search(self, key):

bucket = hash(key) % len(self.table)

buckets\_probed = 0

**while** self.table[bucket] **is** **not** self.EMPTY\_SINCE\_START **and** buckets\_probed < len(self.table):

**if** self.table[bucket] == key:

**return** self.table[bucket]

*# the bucket was occupied (now or previously), so continue probing.*

bucket = (bucket + 1) % len(self.table)

buckets\_probed = buckets\_probed + 1

*# the entire table was probed or an empty cell was found.*

**return** None

*# Overloaded string conversion method to create a string*

*# representation of the entire hash table. Special values*

*# "E/S" and "E/R" are used to represent "EMPTY\_SINCE\_START"*

*# and "EMPTY\_AFTER\_REMOVAL".*

**def** \_\_str\_\_(self):

s = " --------**\n**"

index = 0

**for** bucket **in** self.table:

value = str(bucket)

**if** bucket **is** self.EMPTY\_SINCE\_START: value = 'E/S'

**elif** bucket **is** self.EMPTY\_AFTER\_REMOVAL: value = 'E/R'

s += '{:2}:|{:^6}|**\n**'.format(index, value)

index += 1

s += " --------"

**return** s

## Double Hashing Hash Table

**from** **hashtable** **import** HashTable

**from** **emptybucket** **import** EmptyBucket

*# HashTable class definition using double hashing.*

**class** **DoubleHashingHashTable**(HashTable):

*# Constructor with optional initial capacity. The capacity should always be*

*# set to a prime number.*

**def** \_\_init\_\_(self, initial\_capacity=11):

*# Special constants to be used as the tow types of empty cells.*

self.EMPTY\_SINCE\_START = EmptyBucket()

self.EMPTY\_AFTER\_REMOVAL = EmptyBucket()

*# Initialize all the table cells to be EMPTY\_SINCE\_START.*

self.table = [self.EMPTY\_SINCE\_START] \* initial\_capacity

*# The secondary hash function. Many different functions can*

*# be used here. The function used here is a common one, with*

*# different (usually prime number) constants used where the 7 is.*

**def** h2(self, item):

**return** 7 - hash(item) % 7

*# Inserts a new item into the hash table.*

**def** insert(self, item):

**for** i **in** range(len(self.table)):

*# calculate bucket index for the item for this value of i.*

*# hash() is used as the h1 hashing function.*

bucket = (hash(item) + self.h2(item) \* i) % len(self.table)

**if** type(self.table[bucket]) **is** EmptyBucket:

self.table[bucket] = item

**return** True

*# the entire table was full and the key could not be inserted.*

**return** False

*# Searches for an item with a matching key in the hash table.*

*# Returns the item if found, or None if not found.*

**def** search(self, key):

**for** i **in** range(len(self.table)):

*# calculate bucket index for the item for this value of i.*

*# hash() is used as the h1 hashing function.*

bucket = (hash(key) + self.h2(key) \* i) % len(self.table)

**if** self.table[bucket] == key:

**return** self.table[bucket]

*# the entire table was full and the key was not found*

**return** None

*# Remove an item with a matching key from the hash table,*

*# if found.*

**def** remove(self, key):

**for** i **in** range(len(self.table)):

*# calculate bucket index for the item for this value of i.*

*# hash() is used as the h1 hashing function.*

bucket = (hash(key) + self.h2(key) \* i) % len(self.table)

**if** self.table[bucket] == key:

self.table[bucket] = self.EMPTY\_AFTER\_REMOVAL

*# Overloaded string conversion method to create a string*

*# representation of the entire hash table. Special values*

*# "E/S" and "E/R" are used to represent "EMPTY\_SINCE\_START"*

*# and "EMPTY\_AFTER\_REMOVAL".*

**def** \_\_str\_\_(self):

s = " --------**\n**"

index = 0

**for** bucket **in** self.table:

value = str(bucket)

**if** bucket **is** self.EMPTY\_SINCE\_START: value = 'E/S'

**elif** bucket **is** self.EMPTY\_AFTER\_REMOVAL: value = 'E/R'

s += '{:2}:|{:^6}|**\n**'.format(index, value)

index += 1

s += " --------"

**return** s

## Quadratic Probing Hash Table

**from** **hashtable** **import** HashTable

**from** **emptybucket** **import** EmptyBucket

*# Hashtable class definition using linear probing*

**class** **QuadraticProbingHashTable**(HashTable):

*# Constructor with optional initial capacity. All buckets are*

*# assigned with an EmptyBucket() instance called self.EMPTY\_SINCE\_START.*

**def** \_\_init\_\_(self, initial\_capacity=10, c1=0, c2=1):

*# Special constants to be used as the two types of empty buckets.*

self.EMPTY\_SINCE\_START = EmptyBucket()

self.EMPTY\_AFTER\_REMOVAL = EmptyBucket()

*# Initialize all the table buckets to be EMPTY\_SINCE\_START.*

self.table = [self.EMPTY\_SINCE\_START] \* initial\_capacity

self.c1 = c1

self.c2 = c2

*# Inserts a new item into the hashtable.*

**def** insert(self, item):

i = 0

bucket = (hash(item) + self.c1 \* i + self.c2 \* i\*\*2) % len(self.table)

buckets\_probed = 0

**while** buckets\_probed < len(self.table):

*# if the bucket is empty, the item can be inserted at the at index.*

**if** type(self.table[bucket]) **is** EmptyBucket:

self.table[bucket] = item

**return** True

*# the bucket was occupied, continue probing to next index in table.*

i = i + 1

bucket = (hash(item) + self.c1 \* i + self.c2 \* i\*\*2) % len(self.table)

buckets\_probed = buckets\_probed + 1

*# the entire table was full and the key could not be inserted.*

**return** False

*# Removes an item with a matching key from the hashtable.*

**def** remove(self, key):

i = 0

bucket = (hash(key) + self.c1 \* i + self.c2 \* i\*\*2) % len(self.table)

buckets\_probed = 0

**while** self.table[bucket] **is** **not** self.EMPTY\_SINCE\_START **and** buckets\_probed < len(self.table):

**if** self.table[bucket] == key:

self.table[bucket] = self.EMPTY\_AFTER\_REMOVAL

**return** True

*# the bucket was occupied (now or previously), so continue probing.*

i = i + 1

bucket = (hash(key) + self.c1 \* i + self.c2 \* i\*\*2) % len(self.table)

buckets\_probed = buckets\_probed + 1

**return** False

*# Searches for an item with a matching key in the hashtable. Returns the*

*# item if ound, or None if not found.*

**def** search(self, key):

i = 0

bucket = (hash(key) + self.c1 \* i + self.c2 \* i\*\*2) % len(self.table)

buckets\_probed = 0

**while** self.table[bucket] **is** **not** self.EMPTY\_SINCE\_START **and** buckets\_probed < len(self.table):

**if** self.table[bucket] == key:

**return** self.table[bucket]

*# the bucket was occupied (now or previously), so continue probing.*

i = i + 1

bucket = (hash(key) + self.c1 \* i + self.c2 \* i\*\*2) % len(self.table)

buckets\_probed = buckets\_probed + 1

*# the entire table was probed or an empty cell was found.*

**return** None

*# Overloaded string conversion method to create a string*

*# representation of the entire hashtable. Special values*

*# "E/S" and "E/R" are used to represent "EMPTY\_SINCE\_START"*

*# and "EMPTY\_AFTER\_REMOVAL".*

**def** \_\_str\_\_(self):

s = " --------**\n**"

index = 0

**for** bucket **in** self.table:

value = str(bucket)

**if** bucket **is** self.EMPTY\_SINCE\_START: value = 'E/S'

**elif** bucket **is** self.EMPTY\_AFTER\_REMOVAL: value = 'E/R'

s += '{:2}:|{:^6}|**\n**'.format(index, value)

index += 1

s += " --------"

**return** s

# Output



