```python
def create(m):
    return [None] * m


def hashf(k, m):
    if type(k) is str:
        return sum(ord(c) for c in k) % m
    if type(k) is int:
        return k % m
    raise TypeError("hashf error.")


def insert(D, key, value):
    index = hashf(key, len(D))
    lst = D[index]

    if not lst:
        D[index] = [(key, value)]
        return D

    for i, (k, v) in enumerate(lst):
        if k == key:  # if pair it was already on list, updates
            lst[i] = (key, value)
            return D
    lst.append((key, value))  # else, adds it
    return D


def search(D, key):
    index = hashf(key, len(D))
    lst = D[index]

    if not lst:
        return None

    for i, (k, v) in enumerate(lst):
        if k == key:
            return v

    return None  # not found on that index


def delete(D, key):
    index = hashf(key, len(D))
    lst = D[index]

    if not lst:
        return D

    for i, (k, v) in enumerate(lst):
        if k == key:
            lst.pop(i)
            if len(lst) == 0:
                D[index] = None
            break

    return D


def print_hash(D):
    print("\n==========> HASH")
    for index in range(len(D)):
        slot = D[index]
        print(slot)
    print("")
    return


if __name__ == "__main__":
    dicc = create(5)
    insert(dicc, "A", 20)
    insert(dicc, 20, "G")
    print(dicc)
    print(search(dicc, 20))
    print(search(dicc, "Q"))
    delete(dicc, 20)
    print(dicc)

    # ejercicio1
    D = create(9)
    for e in [5, 28, 19, 15, 20, 33, 12, 17, 10]:
        insert(D, e, "hola")
    print_hash(D)
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio3.py

```
from math import sqrt, floor

from dictionary import *


def hashf(k, m):
    A = (sqrt(5) - 1) / 2
    return floor(m * (k * A % 1))


if __name__ == "__main__":
    dicc = create(1000)
    for num in range(61, 66):
        print(hashf(num, 1000))
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio4.py

```python
import string

from dictionary import *


def insert_count(D, k):
    """Assign the value field as the num of times it has appeared"""
    count = search(D, k)
    if not count:
        insert(D, k, 1)
    else:
        insert(D, k, count + 1)
    return k


def is_permutation(s1, s2):
    """O(n), porque son 3 bucles separados que usan una hash ideal"""
    d1 = create(len(string.ascii_letters))
    d2 = create(len(string.ascii_letters))

    for let in s1:
        insert_count(d1, let)
    for let in s2:
        insert_count(d2, let)

    for let in s1:
        if search(d1, let) != search(d2, let):
            return False
    return True


if __name__ == "__main__":
    print(is_permutation("hola", "ahlo"))
    print(is_permutation("hola", "ahdo"))
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio5.py

```python
from dictionary import *


def unique_elements(lst):
    """O(n) porque es un solo bucle, y la hash la asumo como promedio"""
    dicc = create(40)  # tamaño arbitrario
    for ele in lst:
        if search(dicc, ele):
            return False
        insert(dicc, ele, ele)
    return True


if __name__ == "__main__":
    print(unique_elements([1, 5, 12, 1, 2]))
    print(unique_elements([1, 5, 8, 2]))
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio6.py

```python
from dictionary import create


def hash_code_postal(code, m):
    p, dddd, ccc = code[0], code[1:5], code[5:]

    # Convertir p a su valor ASCII y restar 96 para obtener un número del 1 al 26
    hash_p = ord(p.lower()) - 96

    # Convertir dddd a un número entero
    hash_dddd = int(dddd)

    # Convertir ccc a un número sumando los valores ASCII de cada carácter
    hash_ccc = sum(ord(c) for c in ccc.lower())

    # Calcular el hash final como una combinación ponderada de hash_p, hash_dddd y hash_ccc
    w1, w2, w3 = 100, 10, 1  # pesos
    hash_final = (w1 * hash_p + w2 * hash_dddd + w3 * hash_ccc) % m
    print(hash_p, hash_dddd, hash_ccc, hash_final)

    return hash_final


def insert(D, key, value):
    index = hash_code_postal(key, len(D))
    lst = D[index]

    if not lst:
        D[index] = [(key, value)]
        return D

    for i, (k, v) in enumerate(lst):
        if k == key:  # if pair it was already on list, updates
            lst[i] = (key, value)
            return D
    lst.append((key, value))  # else, adds it
    return D


if __name__ == "__main__":
    P = create(100000)
    for cod, msg in [("M5501EAD", "Factura NRO..."), ("C0001ZXC", "Estimado Carlos..."), ("M5500AAB", "Hola...")]:
        insert(P, cod, msg)
    pass
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio7.py

```python
def compress(s):
    """Returns a compressed string. O(n), just a while with no iterative functions inside."""
    if len(s) <= 1:
        return s

    comp = ""
    count = 1
    for i in range(1, len(s)):
        if s[i] == s[i - 1]:
            count += 1
        else:
            comp += s[i - 1] + str(count)
            count = 1

        if len(comp) >= len(s):
            return s

    comp += s[-1] + str(count)
    return comp if len(comp) < len(s) else s


if __name__ == "__main__":
    print(compress("aabcccccaaa"))
    print(compress("aabcccccmmsadfaaaa"))
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio8.py

```python
from dictionary import hashf, create, search


def insert_once_first(D, key, value):
    index = hashf(key, len(D))
    lst = D[index]

    if not lst:
        D[index] = [(key, value)]
        return D

    for i, (k, v) in enumerate(lst):
        if k == key:  # EDIT from original: if pair it was already on list, returns, we just want 1st time appeared
            return D
    lst.append((key, value))  # else, adds it
    return D


def first_app(p, s):
    """O(k-l+1 + k/l) => O(k/l), con k y l las longitudes de p y s, respectivamente"""
    if len(s) > len(p):
        return None
    dicc = create(len(p))
    for i in range(len(p) - len(s) + 1):  # O(k-l+1) => O(k), siempre
        part = p[i:i + len(s)]
        insert_once_first(dicc, part, i)
    return search(dicc, s)  # O(k/l), ya que depende de la longitud de l, y la naturaleza de p, ya que si en p hay
    # permutaciones de s, sin ser alguna estrictamente p, la lista enlazada puede extenderse (hasta tener len = k??)


if __name__ == "__main__":
    print(first_app("abracadabra", "cada"))
    print(first_app("hola, estoy buscando esta palabra, no esta", "esta"))
    print(first_app("aaxxaaaxxxaaaxxxaaaxxxxax", "xax"))  # caso ineficiente
    print(first_app("abcdefgh", "X"))
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio9.py

```python
from dictionary import create, insert, search


def contains(setS, setT):
    """Verifica si setS  setT.
    Caso Promedio: O(t + s): se tiene que crear el hash de T y la búsqueda de los s siempre se hace"""
    if len(setS) > len(setT):
        return False
    dicT = create(len(setT))
    for t in setT:
        insert(dicT, t, t)
    for s in setS:
        if search(dicT, s) is None:
            return False
    return True


if __name__ == "__main__":
    print(contains([2, 0], [0, 1, 2, -1000, 5]))
    print(contains([0], [0]))
    print(contains([0], [1, 2]))
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio10.py

```python
from ejercicio11 import create, insert, print_hash

if __name__ == "__main__":
    lst = [10, 22, 31, 4, 15, 28, 17, 88, 59]
    D = create(11)
    for ele in lst:
        insert(D, ele, ele)  # edit each insert function to get the desired hash method
    print_hash(D)
    pass

"""
1. Linear probing
(22, 22) True
(88, 88) True
None None
None None
(4, 4) True
(15, 15) True
(28, 28) True
(17, 17) True
(59, 59) True
(31, 31) True
(10, 10) True

2. Quadratic probing con c1 = 1 y c2 = 3
(22, 22) True
(17, 17) True
(59, 59) True
(88, 88) True
(4, 4) True
None None
(28, 28) True
(15, 15) True
None None
(31, 31) True
(10, 10) True

3. Double hashing con h1(k) = k y h2(k) = 1 +(k mod ( m - 1))
(22, 22) True
None None
(59, 59) True
(17, 17) True
(4, 4) True
(15, 15) True
(28, 28) True
(88, 88) True
None None
(31, 31) True
(10, 10) True


"""
```

```
"""¿La lista debe estar doblemente enlazada o con una simplemente enlazada alcanza?
Con que sea simplemente enlazada basta, en ese caso es necesario reiniciar el puntero una vez llega al índice máximo
y bien mantener un contador (o una referencia de partida) para evitar bucles infinitos.
Para el caso de la doblemente enlazada es más sencillo, ya que la búsqueda acaba cuando ambos punteros se tornan None"""


class slot:
    val = None
    flag = None
    """flag as 'isOccupied', possible values: (None, True, False) = (empty, occupied, deleted)"""


def create(m):
    """Each tuple (a, b) represents (key-value-pair, slot-flag)"""
    table = []
    for i in range(m):
        new = slot()
        table.append(new)
    return table


def hashf(k):
    return k


def linear_probing(k, i, m):
    return (hashf(k) + i) % m


def quadratic_probing(k, i, m, c1=1, c2=2):
    return (hashf(k) + c1 * i + c2 * i) % m


def double_hashing(k, i, m):
    def hash_1(k):
        return k

    def hash_2(k):
        return 1 + (k % (m - 1))

    return (hash_1(k) + i * hash_2(k)) % m


def insert(D, key, value):
    i = 0
    while True:
        if i >= len(D):
            raise Exception("hash table is full")

        index = linear_probing(key, i, len(D))
        if index >= len(D):  # resets pointer to beginning of list
            index = 0

        slot = D[index]
        if slot.flag is None or slot.flag is False:
            slot.val = (key, value)
            slot.flag = True
            return D
        else:
            k, v = slot.val
            if k == key:
                slot.val = (key, value)  # overwrites if key already exists

        i += 1


def search(D, key):
    i = 0
    while True:
        if i >= len(D):
            return None

        index = linear_probing(key, i, len(D))
        if index >= len(D):  # resets pointer to beginning of list
            index = 0

        slot = D[index]
        if slot.flag is True:
            k, v = slot.val
            if k == key:
                return v
        elif slot.flag is None:
            return None

        i += 1


def delete(D, key):
    i = 0
    while True:
        if i >= len(D):
            return None

        index = linear_probing(key, i, len(D))
        if index >= len(D):  # resets pointer to beginning of list
            index = 0

        slot = D[index]
        if slot.flag is True:
            k, v = slot.val
            if k == key:
                slot.val = None
                slot.flag = False
                return D
        elif slot.flag is None:  # key not found
            return D
        else:  # slot occupied
            pass

        i += 1
```

File - /home/admin1/Documents/Algoritmos2/practicas/tp-hashtable/code/ejercicio11.py

```
def print_hash(D):
    print("\n==========> HASH")
    for index in range(len(D)):
        slot = D[index]
        print(slot.val, slot.flag)
    print("")
    return


if __name__ == "__main__":
    table = create(4)
    insert(table, 2, 2)
    insert(table, 8, 3)
    insert(table, 11, "ASD")
    print_hash(table)
    # insert(table, 3, 5)
    print(search(table, 5))
    print(search(table, 8))
    delete(table, 11)
    print_hash(table)
    pass
```

```python
"""La tabla hash resultante es la c).
Primero se introduce 12, cuyo módulo 10 es 2, en i=2
Luego se introduce 18, cuyo módulo 10 es 8, en i=8
Luego se introduce 13, cuyo módulo 10 es 3, en i=3
Luego se introduce 2, cuyo módulo 10 es 2, en i=2...
   ...pero está ocupado => Saltea a i+1=3, pero está ocupado => Saltea a i+1=4
   aquí fallan b) y d), b porque no saltea y d porque usa chaining
   a) falla porque reescribe en los slots donde cae
"""

# De paso compruebo
from ejercicio11 import create, insert, print_hash

if __name__ == "__main__":
    lst = [12, 18, 13, 2, 3, 23, 5, 15]
    D = create(10)
    for ele in lst:
        insert(D, ele, ele)  # edit each insert function to get the desired hash method
    print_hash(D)
    # None None
    # None None
    # (12, 12) True
    # (13, 13) True
    # (2, 2) True
    # (3, 3) True
    # (23, 23) True
    # (5, 5) True
    # (18, 18) True
    # (15, 15) True
    pass
```

```
"""
(A) 46, 42, 34, 52, 23, 33: falla al insertar 52, porque si bien es en i=2, que está ocupado, i=3 no lo está
(B) 34, 42, 23, 52, 33, 46: falla al insertar 33, porque si bien es en i=3 e i=3...5 están ocupados, i=6 no lo está
(C) 46, 34, 42, 23, 52, 33: opción correcta
(D) 42, 46, 33, 23, 34, 52: falla al insertar 33, porque si bien se inserta en i=3, la tabla lo muestra en i=7
"""
```