UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

ALGORITMOS Y ESTRUCTURAS DE DATOS II

HashTable

Yacante Daniel

Leg: 10341

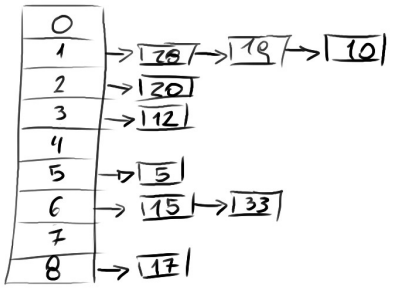
Ejercicio 1

Los siguientes valores son los que regresa la función hash para cada key indicada:

Keys: [5,28,19,15,20,33,12,17,10]

Resturn:[5, 1, 1, 6, 2, 6, 3, 8, 1]

Por lo que la tabla queda de la siguiente forma



Ejercicio 2

Primero creo la clase diccionary y la clase dictionaryElement. El diccionario va a constar de una tabla hash de 9 posiciones como esta definido por la función hash nro 1. Esta lista se llena con “none” una vez se instancia la clase. En esta tabla hash se insertaran en listas los objetos dictionaryElement que tendrán la key original y el valor a guardar.

```python
class dictionary():
    # función hash H (k) = k mod 9 => la tabla tiene 9 posiciones
    tablaHash:list
    m=9
    def fHash(self,key) -> int:
        return key%self.m
    def __init__(self):
        self.tablaHash=[None for i in range(self.m)]
    def __str__(self) -> str:
        cadena=[]
        for nodo in self.tablaHash:
            if nodo!=None:
                for elm in nodo:
                    cadena.append("%s:%s"%(elm.key,elm.value))
        return "{%s}"%("; ".join(cadena))
class dictionaryElement():
    key=None
    value=None
```

Al insertar un elemento, se calcula la posición correspondiente mediante la propia función hash del diccionario y se crea un nuevo dictionaryElement con el key y value pasados a la función.

Si el diccionario en su tabla hash la posición calculada al principio esta vacía se crea una lista con un solo elemento. Si no esta vacía, se inserta el dictionaryElement en la lista que esta en esa posición.

```python
def insert(D:dictionary,key:any,value:any):
    pos=D.fHash(key)
    newElm=dictionaryElement()
    newElm.key=key
    newElm.value=value
    if D.tablaHash[pos]==None:
        D.tablaHash[pos]=[newElm]
    else:
        D.tablaHash[pos].append(newElm)
    return D
```

Para la función search se calcula la posición determinada por la key pasada como argumento, se obtiene lo que haya en esa posición ya sea None o una lista, si es None, el elemento buscado no se encuentra, pero si

hay una lista, hay que buscar en todos lo elementos de dicha lista si el elemento buscado se encuentra. Si se recorre toda la lista y no se ha encontrado es que no esta el elemento en el diccionario.

```
def search(D:dictionary,key:any):
    pos=D.fHash(key)
    elmHash=D.tablaHash[pos]
    if elmHash!=None:
        for nodo in elmHash:
            if nodo.key==key:
                return nodo.value
        return None
    else:
        return None
```

La función Delete es similar a Search, ya que primero se calcula la posición de la tabla mediante la función hash y la key pasada. Se obtiene lo que haya en esa posición de la tabla hash, si es un único elemento, se borra de la posición de la tabla hash, sino se busca en la lista que hay en esa posición y si se encuentra se elimina de la lista.

```
def delete(D:dictionary,key:any):
    pos=D.fHash(key)
    elmHash=D.tablaHash[pos]
    if len(elmHash)==1:
        D.tablaHash[pos].clear()
    else:
        for nodo in elmHash:
            if nodo.key==key:
                elmHash.remove(nodo)
                break
    return D
```

Ejercicio 3

Para este ejercicio simplemente realice una implementación del método multiplicativo con el a y m indicado.

```
"""
Ejercicio 3
"""
print("Ejercicio 3")
def fhash(key):
    a=(sqrt(5)-1)/2
    m=1000
    return floor(m*(key*a-floor(key*a)))
print([fhash(x) for x in [61,62,63,64,65]])
```

Salida por consola

```
Ejercicio 3
[700, 318, 936, 554, 172]
```

Ejercicio 4

Para ver si una cadena es permutación de la otra, lo primero que hago es pasar los caracteres de la cadena a una lista, la cual se ordenara de forma alfabética. Una vez ordenadas si ambas listas no son del mismo tamaño ya se sabe que no van a ser permutaciones, pero si tienen la misma longitud se obtienen los valores key correspondientes a la cadena, el cual utiliza un método de ponderación por posición del carácter. Si dichas key son iguales es que las cadenas lo eran al principio. La complejidad temporal esta determinada por

el tiempo de ordenamiento de la lista, por lo que el algoritmo tiene O(logn) como complejidad temporal.

```
"""
Ejercicio 4
"""
print("Ejercicio 4")
def sonPermutaciones(cad1:str,cad2:str):
    listCad1=[c for c in cad1]
    listCad2=[c for c in cad2]
    listCad1.sort()
    listCad2.sort()
    if len(listCad1)==len(listCad2):
        key1=strToKey(listCad1)
        key2=strToKey(listCad2)
        if key1==key2:
            return True
        else:
            return False
    else:
        return False

def strToKey(key:list):
    suma=0
    for i in range(len(key)):
        suma+=ord(key[i])*pow(128,i)
    return suma

print(sonPermutaciones("hola","aloh"))
print(sonPermutaciones("casa","aloh"))
```

Ejercicio 5

Para este ejercicio se verifica que la lista pasada sea un set o conjunto, en el cual no existen elementos repetidos, la forma mas sencilla de hacerlo es crear un diccionario en el cual iremos insertando los elementos si es que no se han insertado previamente, ya que si esto ocurre es que ese elemento esta repetido en la lista, si al insertarse todos los elementos en el diccionario no hay ninguno repetido, será un set o conjunto. La complejidad temporal será el de O(n) ya que la búsqueda e inserción se harán en O(1) debido a que el diccionario se han implementado tablas hash.

```
"""
Ejercicio 5
"""
print("Ejercicio 5")
def isSet(l:list):
    conjunto=dictionary.dictionary()
    for item in l:
        if dictionary.search(conjunto,item)==None:
            dictionary.insert(conjunto,item,item)
        else:
            return False
    return True

print(isSet([1,5,12,1,2]))
print(isSet([1,5,12,3,2]))
```

Ejercicio 6

Para este ejercicio la forma de encontrar la key que utilice es a todas las cadenas aplicarles la función strToKey, que toma los caracteres y los pasa a base 128, y luego se suma la parte numérica y toda la parte alfabética convertida a un número. Con esa key se aplica una función hash multiplicativa con m=1013, un numero primo “grande” aproximadamente del tamaño de la sumatoria de la key.

```
"""
Ejercicio 6
"""
print("Ejercicio 6")
def zipHash(codigo:str):
    if len(codigo)<9:
        m=1013
        a=(sqrt(5)-1)/2
        funcHash=lambda key: floor(m*(key*a-floor(key*a)))
        c1=codigo[0]
        c2=codigo[1:5]
        c3=codigo[5:]
        key=strToKey(c1)+int(c2)+strToKey(c3)
        pos=funcHash(key)
        print(pos)
        return pos

    else:
        return None
zipHash("A9514BFJ")
```

Ejercicio 7

Para acortar la cadena, simplemente se recorre dicha cadena viendo cambios de caracteres, cada vez que se cambia de carácter se agrega a la cadAcort (variable que va a almacenar la cadena acortada) la letra y el valor máximo que se alcanzo en el acumulador, luego de hacer esto se reinicia el acumulador a 1 y se actualiza el charAct que tiene el carácter actual que se está contando.

Si luego de esto la cadena acortada es mayor a la cadena original se regresa la cadena original, sino la cadena acortada.

```
"""
Ejercicio 7
"""
print("Ejercicio 7")
def acortarString(S:str):
    charAct=S[0]
    cont=1
    cadAcort=""
    for char in S[1:]:
        if char!=charAct:
            cadAcort+=f"{charAct}{cont}"
            cont=1
            charAct=char
        else:
            cont+=1
    cadAcort+=f"{charAct}{cont}"
    if len(cadAcort)<len(S):
        return cadAcort
    else:
        return S
print("Acortado de 'aabcccccaaa'")
print(acortarString("aabcccccaaa"))
print("Acortado de 'ab'")
print(acortarString("ab"))
```

Ejercicio 8

Para resolver este problema se implementó un algoritmo similar al de Rabin–Karp que compara el valor que regresa la función hash con lo que se desea buscar con lo que regresa la función hash del pedazo de cadena de igual longitud, si estos valores son iguales, se ve si las cadenas son iguales. La complejidad sería un O(m-n) ya que se harán m-n comprobaciones de hash.

```python
"""
Ejercicio 8
"""
print("Ejercicio 8")
def findString(pattern:str, string:str):
    funcHash=lambda key: key%1013 #m=1013, numero primo elegido
    keyPattern=funcHash(strToKey([c for c in pattern]))
    n=len(pattern)
    m=len(string)
    for i in range(m-n+1):
        keyStringCut=funcHash(strToKey(string[i:i+n]))
        if keyPattern==keyStringCut:
            if pattern==string[i:i+n]:
                return i
    return None

print(findString("cada","abradacadabra"))
```

Ejercicio 9

Para resolver este ejercicio lo que hace el algoritmo es insertar en una hashTable los elementos de T, la función hash implementada es del método multiplicativo.

Luego de insertar todos los elementos de T, para cada valor de S se usa la misma función hash para buscar el lugar en el que esta insertado, si el valor de S se encuentra en la tabla, se continua con el próximo valor de S.

Si un valor no se llega a encontrar de S, la bandera de HC (Hay Coincidencia) hará que se regrese False, ya que no se cumple con que S este incluido en T…es decir todos los valores de S están en T.

La complejidad esta en recorrer los elementos de T, O(n), mas el insertarlos en la tabla hash, O(1), mas recorrer todos los elementos de S, O(m), mas el buscar los elementos insertados, O(fc), donde se trata de que el fc sea lo mas pequeño posible. Por lo tanto la complejidad total es O(m+n).

```python
"""
Ejercicio 9
Complejidad:
    Insertar elementos de T en la tabla: O(n), insertar es O(1)
    Buscar los elementos de S en la tabla: O(m), calculo de key O(1), búsqueda en la posición indicada O(n/m)
    Total: O(n+m)

"""

def checkSubSet(setS:list,setT:list):
    setS=set(setS)
    setT=set(setT)
    hashTable=[None for i in range(len(setT))]
    m=len(hashTable)
    a=(sqrt(5)-1)/2
    funcHash= lambda key: floor(m*(key*a-floor(key*a)))
    for elem in setT:
        pos=funcHash(elem)
        if hashTable[pos]==None:
            hashTable[pos]=[elem]
        else:
            hashTable[pos].append(elem)
    # print(hashTable)
    for elem in setS:
        HC=False
        pos=funcHash(elem)
        if hashTable[pos]!=None:
            for val in hashTable[pos]:
                if val==elem:
                    HC=True
            if not HC:
                return False
        else:
            return False
        return True

print("Random S y T")
s1=[randint(1,50) for i in range(5)]
s2=[randint(1,50) for i in range(10)]
print("S: ",s1,"\nT:",s2)
print(checkSubSet(s1,s2))
print("S y T a mano")
s1=[9,11,5,3,1]
s2=[8,4,3,1,2,9,11,5,21,27]
print("S: ",s1,"\nT:",s2)
print(checkSubSet(s1,s2))
```

Salida por consola:

```
Random S y T
S:  [18, 22, 29, 40, 17]
T: [26, 1, 43, 8, 45, 25, 48, 48, 47, 37]
False
S y T a mano
S:  [9, 11, 5, 3, 1]
T: [8, 4, 3, 1, 2, 9, 11, 5, 21, 27]
True
```

Ejercicio 10

Implementación de los algoritmos de linear-probin, quadratic-probin y double hash. Para cada valor de key, se inserta el valor en la tabla mediante el método elegido. Dicha tabla se muestra al final por consola.

```python
def ej10():
    keys=[10,22,31,4,15,28,17,88,59]
    print("Keys:\n",keys)
    m=11
    c1=1
    c2=3
    h1=lambda key: key
    h2=lambda key: 1+(key%(m-1))
    fHashLin=lambda key: key%m
    fHashCuad=lambda key,i: (key+c1*i+c2*pow(i,2))%m
    fHashHash=lambda key,i: (h1(key)+i*h2(key))%m
    tHashLin=[None for i in range(m)]
    tHashCuad=[None for i in range(m)]
    tHashHash=[None for i in range(m)]

    for i in range(len(keys)):
        for j in range(m):
            pos=fHashLin(keys[i]+j)
            if tHashLin[pos]==None:
                tHashLin[pos]=keys[i]
                break
            else:
                continue
        for j in range(m):
            pos=fHashCuad(keys[i],j)
            if tHashCuad[pos]==None:
                tHashCuad[pos]=keys[i]
                break
            else:
                continue
        for j in range(m):
            pos=fHashHash(keys[i],j)
            if tHashHash[pos]==None:
                tHashHash[pos]=keys[i]
                break
            else:
                continue
    print("Linear\n",tHashLin)
    print("Cuadrática\n",tHashCuad)
    print("Doble Hash\n",tHashHash)
```

Salida por consola:

```
Ejercicio 10
Keys:
 [10, 22, 31, 4, 15, 28, 17, 88, 59]
Linear
 [22, 88, None, None, 4, 15, 28, 17, 59, 31, 10]
Cuadrática
 [22, None, 88, 17, 4, None, 28, 59, 15, 31, 10]
Doble Hash
 [22, None, 59, 17, 4, 15, 28, 88, None, 31, 10]
```

Se puede observar que las posiciones para cada método quedan ocupadas por valores distintos según el método.

Ejercicio 12

Llaves: 12, 18, 13, 2, 3, 23, 5,15

h(12)=2; h(18)=8; h(13)=3; h(2)=2; h(3)=3; h(23)=3; h(5)=5; h(15)=5

Primero el valor de 12 se inserta en la posición 2, 18 en la posición 8, 13 en la 3, al llegar 2 y al estar la posición 2 ocupada por 12, se busca la próxima libre, que será la posición 4, algo similar ocurre con el valor de 3, la posición de 3 esta ocupada por 13, la 4 por el 2 así que la próxima libre es la 5. Al llegar el 23 las posiciones 3, 4 y 5 ya están ocupadas, por lo que se inserta en la 6ta. Al insertar el 5 la posición 5 y 6 están ocupadas por lo que se inserta en la posición 7 que esta libre. Con el valor 15 todas las posiciones del 5 al 8 están ocupadas, así que la próxima posición libre es la 9.

La opción que corresponde a esta secuencia es la C.

Ejercicio 13

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Al analizar la tabla se puede ver que para que sea la secuencia correcta en linear-probing, el 42 debe haber sido insertado antes que el 52, y el 23 antes que el 33, ya que estos valores devuelven el mismo índice en la tabla, pero el que primero se insertó encontró el lugar correcto. Con esto se descarta la opción D, ya que el valor 33 se insertaría primero que el 23.

De las 3 restantes, se descarta la opción B ya que al seguir el linear-probing de la secuencia el valor de 33 se hubiera insertado en la posición 6 y no 7 como esta en la figura. Y luego se descarta la opción A, ya que el valor 52 se hubiera insertado en la posición 3 ya que esta estaba vacía en ese momento. Por lo que la opción correcta es la C.