	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Jesús Cruz Navarro

Asignatura: Estructuras de Datos y Algoritmos 2

Grupo: 1

No de Práctica(s): 5

Integrante(s): Ugalde Velasco Armando

No. de Equipo de cómputo empleado:

No. de Lista o Brigada: 32

Semestre: 2021-1

Fecha de entrega: 8 de noviembre de 2020

Observaciones:

CALIFICACIÓN: _____

PRÁCTICA 5: BÚSQUEDA 2

Objetivo: El estudiante identificará el comportamiento y características de algunos algoritmos de búsqueda por comparación de llaves.

1. Desarrollar un programa en python3 que:

a) Implemente la Tabla Hash vista en clase con **chaining** para el manejo de colisiones y función hash universal con las siguientes operaciones:

- **CrearTablaHash**(tamaño) **->** regresa **tabla**
- **Insertar**(tabla, llave, valor) **->** regresa **void**
- **Busca**(tabla, llave) **->** regresa **valor**

Se decidió implementar la estructura en una clase con el mismo nombre. El constructor, en este caso, es el equivalente a la función solicitada **CrearTablaHash**. Ésta se encarga de instanciar una tabla, creando una lista con el tamaño indicado. De no proporcionarse este argumento, se utiliza el valor default: **10**. Además, se obtienen los valores aleatorios a utilizar en la **función hash**. Para obtener el valor primo **p**, se utilizó la librería **pycryptodome**.

```
class HashTable:
    def __init__(self, initialSize=10):
        self.list = [None] * initialSize
        self.size = 0

        # Obtener a, b y p
        self.LARGE_PRIME = getPrime(PRIME_BITS)
        self.MAX_KEY_VALUE = self.LARGE_PRIME - 1
        self.A = random.randint(1, self.MAX_KEY_VALUE)
        self.B = random.randint(0, self.MAX_KEY_VALUE)
        print("Valor para a:", self.A)
        print("Valor para b:", self.B)
        print("Valor para p:", self.LARGE_PRIME)
        print("Valor key máximo:", self.MAX_KEY_VALUE, "\n")
```

Constructor de la clase HashTable

Además, se definió una clase auxiliar **Node** que contendría las llaves y datos de los elementos a insertar.

```
class Node:
    def __init__(self, key, satelliteInformation):
        self.key = key
        self.satelliteInformation = satelliteInformation
```

Clase Node

En el método **insert,** primero se comprobó que no existiera ningún elemento dentro de la tabla con la misma llave que el elemento a insertar. Posteriormente, se utilizó la clase antes mencionada para instanciar el elemento a insertar. Se utilizó la técnica de **table doubling** para mantener el factor de carga con un valor menor o igual a uno, y, finalmente, se insertó el elemento utilizando el método auxiliar **insertNodeWithoutSizeChecking**.

```
def insert(self, key, value):
    if self.search(key) is not None:
        raise ValueError("Duplicate key")
    newNode = self.Node(key, value)
    # Asegurarnos de que el factor de carga
    # siempre sea menor o igual a uno
    if self.size == self.getListSize():
        self.doubleSizeAndRehash()

    self.insertNodeWithoutSizeChecking(newNode)

def insertNodeWithoutSizeChecking(self, node):
    computed_hash = self.getHash(node.key)

    if self.list[computed_hash] is None:
        self.list[computed_hash] = []

    self.list[computed_hash].append(node)
    self.size += 1
```

Método insert e insertNodeWithoutSizeChecking

Por último, se implementó el método **search**. En éste, primero se obtiene el **hash** de la llave proporcionada utilizando el método auxiliar **getHash**. Posteriormente, se comprueba si existe una lista presente en la posición del hash calculado. De lo contrario, es evidente que el elemento no se encuentra dentro de la tabla, por lo tanto, se retorna el valor **None**. Finalmente, se recorre la lista mencionada para comprobar si el elemento deseado se encuentra en ella. De ser así, se retorna la información que contiene; de lo contrario, se retorna el valor **None**.

```
def search(self, key):
    computed_hash = self.getHash(key)

    if self.list[computed_hash] is None:
        return None

    for node in self.list[computed_hash]:
        if node.key == key:
            return node.satelliteInformation

    return None
```

Método search

b) Cree al inicio una Tabla Hash con la Base de Datos de los 1000 usuarios proporcionada **(Database.py).**

Para cumplir con el objetivo planteado, se implementó el siguiente método:

```
def initializeHashTable(table, initialList=None):
    if initialList is None:
        initialList = Usuario.GetUsuariosDB(1000)
    for userEntry in initialList:
        table.insert(userEntry.username, userEntry)
```

Método initializeHashTable

Éste tiene dos parámetros: **table**, la tabla a inicializar con los valores de usuarios; e **initialList**, una lista opcional con éstos. Como se logra observar, en caso de no proporcionar el argumento anterior, se recupera la lista de usuarios de la clase **Usuario** presente en el archivo **Database.py.** Finalmente, se insertan los valores tomando como llaves los nombres de usuario. Esta funcionalidad se utiliza finalmente en **testLogin**, mostrada a continuación.

c) Implemente una función **Login** que busque un usuario en la tabla hash, compare el **password** y, dependiendo del resultado, imprima en consola uno de los siguientes mensajes:

- Si el usuario puede hacer **login** (usuario existe y contraseña correcta). **"Acceso Autorizado: Bienvenido nombre_completo"**

- Si la contraseña del usuario es incorrecta (el usuario existe, pero contraseña incorrecta). "**Acceso Denegado: Contraseña Incorrecta"**

- Si el usuario no existe función (el usuario no existe). **"Usuario no encontrado."**

La función **Login** debe tener la siguiente firma**: Login**(**tablaHash, username, password**) **->** regresa **void**

Se implementó la función solicitada como se muestra a continuación:

```python
def login(tablaHash, username, password):
    user = tablaHash.search(username)

    if user is None:
        return print("Usuario no encontrado")

    if user.password == password:
        return print("Acceso autorizado, bienvenido/a", user.fullname)
    else:
        return print("Acceso denegado: Contraseña incorrecta")
```

Función login

Primero, se buscó el elemento en la tabla proporcionada, utilizando el **username** como la llave. Posteriormente, se consideraron los tres casos: el usuario no se encontró, el usuario fue encontrado y proporcionó una contraseña correcta, o bien, se encontró pero proporcionó una contraseña incorrecta.

d) Pruebe todos los casos de la función **Login** para diferentes usuarios e imprima los parámetros elegidos para la función hash y el factor de carga de esta.

Para cumplir con el objetivo se implementó la función **testLogin**, donde, primero se creó la **tabla hash** con el tamaño deseado **(1000),** se inicializaron sus valores, se imprimieron los valores pertinentes, y, finalmente, se ejecutó la función **login** de tal forma que se presentaran los tres casos posibles:

```
def testLogin():
    hashTable = HashTable(1000)
    initializeHashTable(hashTable)
    print("Factor de carga:", hashTable.getLoadFactor())
    print("Número máximo de colisiones:", hashTable.getMaxCollisions(), "\n")

    login(hashTable, "brevierk", "lF4RnE")
    login(hashTable, "brevierk", "Contraseña incorrecta")
    login(hashTable, "Usuario inexistente", "Contraseña")
    print()
```

Función testLogin

```
Valor para a: 887129561714546137356521875419469923700339275388044830580915
Valor para b: 10500158586973422306337320999168213016067486015016369087099 51
Valor para p: 15216603228207921905005113866018842923976925316256815080786 17
Valor key máximo: 152166032282079219050051138660188429239769253162568150807861 6

Factor de carga: 1.0
Número máximo de colisiones: 5

Acceso autorizado, bienvenido/a Baldwin Revie
Acceso denegado: Contraseña incorrecta
Usuario no encontrado
```

Salida del programa

e) Además, compare la búsqueda lineal con la búsqueda en la tabla hash, y obtenga el **PROMEDIO** de los tiempos de ejecución para **10** casos (caso promedio), usando la **BD** de los **1000** usuarios. Imprima el tiempo de ejecución para cada caso

Se implementó la función **searchAndGetTimes**, que busca el elemento **toSearch** dentro de la tabla y utilizando el algoritmo de búsqueda lineal dentro de la lista **arr**. Se calculan los tiempos respectivos y se retorna una tupla con éstos.

```python
# Función que calcula el tiempo utilizado por los algoritmos definidos
# Devuelve una tupla con los tiempos correspondientes
def searchAndGetTimes(arr, toSearch, table):
    print("Username:", toSearch)

    t1 = time.perf_counter()
    table.search(toSearch)
    t2 = time.perf_counter()
    tableSearchTime = t2 - t1
    print("Hash table search:", tableSearchTime)

    t1 = time.perf_counter()
    linearSearchUsers(arr, toSearch)
    t2 = time.perf_counter()
    linearSearchTime = t2 - t1
    print("Linear search:", linearSearchTime, "\n")

    return tableSearchTime, linearSearchTime
```

Función searchAndGetTimes

Finalmente, en la función **testTimes**, se calculan los tiempos de ejecución para **10** casos y se calcula el promedio para cada algoritmo.

```python
def testTimes():
    # Lista con los tiempos de ejecución para cada caso
    TIMES = []
    users = Usuario.GetUsuariosDB(math.inf)
    TOTAL_USERS = len(users)
    hashTable = HashTable(TOTAL_USERS)
    initializeHashTable(hashTable, users)

    # Calcular los tiempos de cada algoritmo y agregar la tupla respectiva a la lista
    for i in range(NUMBER_OF_CASES):
        randomUser = users[random.randint(0, TOTAL_USERS - 1)]
        TIMES.append(searchAndGetTimes(users, randomUser.username, hashTable))

    # Calcular el promedio de tiempo para cada algoritmo
    firstMean = numpy.mean([t[0] for t in TIMES])
    print("Tiempo promedio búsqueda por tabla hash", firstMean)
    secondMean = numpy.mean([t[1] for t in TIMES])
    print("Tiempo promedio búsqueda lineal", secondMean, "\n")
```

Función testTimes

```
Username: hmomfordq2
Hash table search: 9.09999999998412e-06
Linear search: 0.00010430000000000161

Username: rhandysidei5
Hash table search: 5.199999999982996e-06
Linear search: 6.349999999999412e-05

Username: kortiga72
Hash table search: 4.499999999962867e-06
Linear search: 2.359999999995699e-05

Username: prangellr4
Hash table search: 4.400000000015503e-06
Linear search: 9.47000000000312e-05

Username: abernlicv
Hash table search: 4.299999999957116e-06
Linear search: 4.4399999999999995e-05

Username: canthonsencl
Hash table search: 4.800000000027005e-06
Linear search: 4.2200000000047755e-05

Username: njorenu
Hash table search: 4.2000000000097515e-06
Linear search: 8.019999999997474e-05

Username: jgoleyco
Hash table search: 4.100000000006876e-06
Linear search: 4.270000000000662e-05
```

```
Username: adislee3u
Hash table search: 4.400000000015503e-06
Linear search: 1.3899999999955615e-05

Username: brysdale6q
Hash table search: 4.300000000012627e-06
Linear search: 2.199999999996649e-05

Tiempo promedio búsqueda por tabla hash 4.929999999997436e-06
Tiempo promedio búsqueda lineal 5.314999999999071e-05
```

Salida del programa

CONCLUSIONES

Las **tablas hash** son una de las estructuras de datos más importantes en las *Ciencias de la Computación*. Sin duda alguna, forman parte fundamental de la implementación de numerosos sistemas y algoritmos en la actualidad. Como ya se analizó durante la clase y en el desarrollo de la práctica, éstas permiten realizar búsquedas en una forma muy rápida. En palabras más concretas, se determinó que la complejidad asintótica de esta última operación es **O(1)** en la mayor parte de los casos, si se implementa de forma correcta.

Algunos factores que influyen en su rendimiento es el factor de carga y la función hash utilizada. Mientras menor sea el primero, menor será la probabilidad de colisión, y, por lo tanto, los tiempos de búsqueda serán menores. Por otro lado, es importante elegir una función hash que distribuya uniformemente el universo de llaves a los índices válidos.

Además, cabe mencionar que el **hashing universal** es una técnica que podemos utilizar para prevenir que un *"adversario"* logre mermar el rendimiento de nuestra estructura. Al elegir nuestra función hash de forma aleatoria, prevenimos que éste conozca los valores de las llaves para los cuales se presentarían más colisiones.