UNIVERSIDAD DE COSTA RICA

IE-0217

Estructuras Abstractas de Datos y Algoritmos para Ingeniería

PROFESOR: JUAN CARLOS COTO

Proyecto: *Algoritmo Open-Addressing para Hash Table*

Estudiante: Michelle Gutiérrez

Carné: B43195

14/12/20

Índice

1. Introducción

Las Tablas de Hash corresponden a una estructura de datos que facilitan el almacenamiento de datos, específicamente el acceso a los datos ya almacenados. Por ejemplo, si se utiliza un array para almacenar diferentes datos, para buscar un dato en específico sería necesario recorrer todo el array hasta encontrar la posición del array con el contenido buscado, sin embargo, si la posición en la que se almacena un dato depende de un criterio en específico, se puede usar este mismo criterio para saber la posición en la que debería estar almacenado y buscarlo de forma directa. Utilizar una función para calcular la posición (es decir, el índice) en el que será almacenado un dato en un array, es lo que hace una Tabla Hash, la función utilizada es llamada función de Hash. Esta estructura de datos se compone principalmente de las siguientes funciones:

- *Declaración o inicialización:* De esta forma se define el array que será usado para costruir la tabla y además su capacidad.
- *Hash:* Está función recibe el valor o dato a ser almacenado y aplica la función hash para calcular la posición.
- *Agregar:* Se encarga de incluir nuevos datos en la tabla hash según el resultado de hash.
- *Buscar:* Se encarga de buscar un elemento de la tabla en particupar como se mencinó antes, utilizando hash también.
- *Borrar:* Se encarga de eliminar un elemento de la tabla.

Una buena función de Hash debería tener ciertas características para considerarse buena, como ser fácil de calcular, debería de producir resultados de índices que se distribuyan de forma uniforme sobre el array y debería utilizar todo el elemento para obtener todo el resultado, solo una parte, entre otros. [1].

Si el resultado de la función Hash es el mismo para dos elementos diferentes entonces se produce una colisión. Una función de Hash que no permita colisiones sería perfecta, sin embargo, esto no sucede y eventualmente suceden colisiones. Para manejar las colisiones se han desarrollado diferentes algoritmos. Entre ellos se encuentran: Chaining, Open Adressing y Double Hash, entre otros. [2]. Al aplicar estos algoritmos se vuelve necesario modificar la estructura básica de las funciones básicas y en algunos casos podría ser necesario agregar nuevas, como se podrá ver más adelante al comparar la implementación de una Tabla Hash Básica con la de una Tabla Hash con Open Adressing.

2. Discusión

Como se mencionó antes existen diferentes algoritmos utilizados para lidiar con las colisiones en una Tabla Hash, el enfoque de este trabajo es el algoritmo de Open Adressing. Este algoritmo es también conocido como Linear Probing, en este todos los elementos se guardan de forma simple en la Tabla Hash, es decir, cada elemento de la Tabla contiene está ocupado o es nulo, no contiene otras listas, como el método de Chaining. Al darse una colisión el algoritmo empieza a buscar en los demás espacios disponibles hasta encontrar uno desocupado, de forma lineal, de esto el nombre. Se deben tener las consideraciones necesarias para que se busque en los espacios que están antes y después del que se encontró ocupado. [3]

El código de la implementación en python de una Tabla Hash sin control de colisiones se muestra en la

Figura 1. Aquí se puede ver que las funciones básicas de la Tabla son bastante sencillas, especialmente al implementarlo en Python.

```
In [19]: # Clase Hash Table
         class HashTable:
             def __init__(self):
                 self.MAX = 10
                 self.arr = [None for i in range(self.MAX)]
             def hash(self,key):
                 return key % self.MAX
             def add(self,key,value):
                 x = self.hash(key)
                 self.arr[x] = value
             def get(self,key):
                 x =self.hash(key)
                 return self.arr[x]
             def delete(self,key):
                 x = self.hash(key)
                 self.arr[x] = None
```

Figura 1: Implementación en python de una Tabla Hash sin manejo de colisiones

Si se corre esta implementación, provocando una colisión (el índice que resulta para 89 y 19 es 9), se puede observar en la Figura 2 que efectivamente el último dato suplanta el que estaba en la posición de colisión originalmente.

```
In [20]: o = HashTable()
         o.add(89,7)
         print(o.arr)
         o.add(19,79)
         print(o.arr)

[None, None, None, None, None, None, None, None, None, 7]
[None, None, None, None, None, None, None, None, None, 79]
```

Figura 2: Resultados de una Tabla Hash sin manejo de colisiones

Para implementar el algoritmo de Open Adressing será necesario modificar algunas funciones y agregar también. Las funciones de inicialización y de hash no se modifican, la función de agregar (add) ahora verifica primero que en la posición calculada por la función Hash (el índice) no está ocupada, si no lo está, asigna el elemento como corresponde, si la posición está ocupada (colisión) se dispone a buscar el nuevo índice que tenga una disposición disponible, se puede utilizar una función aparte para

encontrar el espacio disponible. Cuando encuentra el nuevo índice, lo utiliza para guardar el valor correspondiente (tanto valor como la llave). En caso de no encontrar un espacio vacío disponible, levanta una exception indicando que la tabla está llena. La modificación realizada a esta función y la función nueva para buscar un espacio vacío se puede ver en la Figura 3.

```
def add(self,key,value):
    x = self.hash(key)
    if self.arr[x] is None:
        self.arr[x] = (key,value)
    else:
        new_x = self.free_space(key, x)
        self.arr[new_x] = (key,value)
def free_space(self,key,x):
    trange = [*range(0,x)] + [*range(x, len(self.arr))]
    for x in trange:
        if self.arr[x] is None:
            return x
        else:
            raise Exception("Tabla llena")
```

Figura 3: Función add modificada y nueva función free$_s$*pace*

Por otro lado, las funciones get (buscar) y delete (borrar) deben buscar en el rango correspondiente de la tabla por medio de la key (llave) directamente hasta encontrar el elemento específico que se desea obtener o eliminar, ya que si solo se utiliza la función hash podría existir ambiguedad por las colisiones. Si el elemento deseado no es encontrado, ambas levantan excepciones indicando que el elemento no fue encontrado. Estas funciones me pueden ver en la Figura 4.

```
    def get(self,key):
        x =self.hash(key)
        trange = [*range(0,x)] + [*range(x, len(self.arr))]
        for x in trange:
            item = self.arr[x]
            if item is None:
                raise Exception("Item no encontrado")
            if item[0] == key:
                return item[1]
    def delete(self,key):
        x =self.hash(key)
        trange = [*range(0,x)] + [*range(x, len(self.arr))]
        for i in trange:
            if self.arr[i][0] == key:
                self.arr[i]= None
            else:
                raise Exception("Item no encontrado")
```

Figura 4: Funciones get y delete modificadas

3. Ejemplo

Para probar la implementación del algoritmo se utilizará el mismo caso que se mencionó anteriormente en el que las llaves 89 y 19 generan una colisión en el índice 9. En la Figura 5 se muestra el resultado del código para la Tabla Hash con Open Adressing, en esta se puede ver como esta vez a pesar de que existe una colisión, ambos datos son almacenados dentro de la Tabla.

```
In [138]: o = HashTable()
          o.add(89,7)
          print(o.arr)
          o.add(19,79)
          print(o.arr)

          [None, None, None, None, None, None, None, None, None, (89, 7)]
          [(19, 79), None, None, None, None, None, None, None, None, (89, 7)]
```

Figura 5: Resultados de una Tabla Hash con manejo de colisiones

4. Conclusiones

- Las tablas Hash son una buena opción para implementar un diccionario, ya que permiten el acceso directo a la información deseada.
- Existen diferentes técnicas para lidiar con las colisiones en las Tablas Hash, unas tienen ventajas sobre las otras dependiendo de los requerimientos o casos particulares.
- Fue posible familiarizarse con la sintaxis básica y el concepto de clase en python. Además las excepciones ayudan a saber que está pasando en el programa.

Referencias

[1] Putambekar A. *Advanced Data Structures and Algorithms.* Technical Publications Pune, 2008.

[2] Kolev V. Nakov S. *Fundamentals of Computer Programming with C: The Bulgarian C Book.* Faber Publishing, 2013.

[3] Charles E Leiserson Ronald L Rivest Clifford Stein Thomas H.. Cormen, Thomas H Cormen. *Introduction To Algorithms.* MIT Press, 2001.

5. Código

```
# Michelle Gutiérrez Muñoz - B43195
# Proyecto del curso Estructuras Abstractas de Datos y Algoritmos para Ingeniería
# Implementación de Hash Table con Open Adressing para manejar las colisiones


# Clase Hash Table
class HashTable:
    def __init__(self):
        self.MAX = 10
        self.arr = [None for i in range(self.MAX)]
    def hash(self,key):
        return key % self.MAX
    def add(self,key,value):
        x = self.hash(key)
        if self.arr[x] is None:
            self.arr[x] = (key,value)
        else:
            new_x = self.free_space(key, x)
            self.arr[new_x] = (key,value)
    def free_space(self,key,x):
        trange = [*range(0,x)] + [*range(x, len(self.arr))]
        for x in trange:
            if self.arr[x] is None:
                return x
            else:
                raise Exception("Tabla llena")
    def get(self,key):
        x =self.hash(key)
        trange = [*range(0,x)] + [*range(x, len(self.arr))]
        for x in trange:
            item = self.arr[x]
            if item is None:
                raise Exception("Item no encontrado")
            if item[0] == key:
                return item[1]
    def delete(self,key):
        x =self.hash(key)
        trange = [*range(0,x)] + [*range(x, len(self.arr))]
        for i in trange:
            if self.arr[i][0] == key:
                self.arr[i]= None
            else:
                raise Exception("Item no encontrado")
```