Miniproyecto 3 Tabla hash en disco y ordenación por montículo binario

Alejandra Jaramillo Mejía

Cristian Ramírez Zuluaga

Universidad Nacional Sede Manizales

Estructura de datos

German Augusto Osorio Zulu

Tabla de contenido

Introducción

En el mini-Proyecto 3 realizado para la asignatura de Estructura de datos se guardan 100 registros en una tabla hash; cada uno de estos registros con un código de identificación y el nombre completo de una persona. Además, se tiene una interfaz donde se pueden visualizar algunas propiedades de la tabla hash y del montículo binario.

Para la realización de la tabla hash se utilizó una función hash denominada residuo y un método de resolución de colisiones denominado sondeo cuadrático con el objetivo de que la información de cada persona quede en una ranura única.

Mundo del problema

Antes de iniciar la descripción del proyecto este proyecto debemos dejar claro su alcance y algunos aspectos claves que consideramos de vital importancia para su realización:

Alcance:

Por medio de este proyecto se puede verificar las propiedades y los beneficios de usar las tablas hash para la búsqueda de elementos teniendo en cuenta que se almacena una cantidad elevada de ítems.

El resultado final del proyecto es el uso de una interfaz con la cual:

1. Se puede acceder al nombre de la persona a través del código de identificación
2. Listar los primeros 10 registros con base en la ordenación por el método de montículo binario.
3. Calcular el número de colisiones que se presentaron para cada tamaño de tabla 103, 113, 131, 163, 199, 251 y 311.

Conceptos Utilizados:

Tabla hash:

En la tabla hash los datos se almacenan de tal manera que el procedimiento de búsqueda de la información de una determinada persona sea mucho más rápido que utilizando los métodos de búsqueda secuencial o binaria.

Función Hash:

Para realizar una tabla hash necesitamos una función hash la cual se encarga de asignarle una ranura en la tabla a la información de determinada persona.

En este proyecto se utilizó la función hash "Residuo" la cual está definida de la siguiente manera:

```
def h(item):  return item % 11
```

Esta función nos devuelve un número entero en el rango de [0, m-1] donde m es el tamaño de la tabla y este número se convierte en el índice de la tabla hash.

Sondeo cuadrático:

Cabe aclarar que, en este proyecto es inevitable no tener colisiones por lo que tenemos una tabla hash no perfecta. Para solucionar estas colisiones utilizamos el método del **sondeo cuadrático** el cual es un esquema de direccionamiento abierto en programación informática para resolver colisiones en tablas hash . El sondeo cuadrático toma el índice hash original y agrega valores sucesivos de un polinomio cuadrático arbitrario hasta que se encuentra una ranura abierta.

Si el primer valor hash es H, los valores sucesivos son:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \ldots, H + k^2$$

Montículo binario:

Un montículo binario es basado en un árbol binario, que tiene dos restricciones adicionales:

1. Para un montículo binario max, cada nodo contiene un valor superior al de sus hijos y para un montículo binario min, cada nodo contiene un valor más pequeño que el de sus hijos.

Cabe aclarar que en este proyecto utilizamos un montículo binario min.

2. Propiedad de orden del montículo

   Significa que todos los niveles están LLENOS excepto el último nivel, y SOLO faltarán nodos a la derecha.

Descripción del proyecto

El proyecto miniProyecto3 consta de 5 archivos.

- ArchNombres1.txt
- main.py
- HashFunctions.py
- dataset.txt
- BinaryHeap.py

- ArchNombres1.txt

En este archivo de texto en disco donde se encuentran almacenados los 100 registros cada uno con el código de identificación único de 3 dígitos (entre 100 y 999) y el nombre de una persona

- HashFuctions.py

Este archivo contiene la función hash que se utiliza para insertar los elementos a la tabla hash la cual recibe el nombre de hashFunction.

Por otra parte, la función fileWriter crea un archivo llamado dataSet.txt donde se va a almacenar la tabla hash con el código hash, el número de identificación y la posición del archivo original que refleja la información de esa persona.

Además, tiene una función insertHashTable la cual recibe una lista con el número de identificación de cada persona y el tamaño de la tabla hash donde se van a introducir los elementos. Esta función crea una tabla hash del tamaño ingresado y debe insertar tuplas a la tabla las cuales contienen la identificación de la persona y posición en el archivo (lo anterior utilizando la función hash):

Primero debe detectar si existe un elemento en el índice hash (ranura). Si la ranura está vacía inserta el elemento en esa ranura o en caso contrario se realiza el método de solución de colisiones sondeo cuadrático el cual opera tomando el índice hash original y agregando valores sucesivos de un polinomio cuadrático arbitrario hasta que se encuentra una ranura abierta y esta posición es utilizada para el nuevo elemento. Adicionalmente en la función se calcula el número de colisiones que se detectaran al llenar la tabla hash dependiendo de su tamaño.

Este archivo también contiene la función contains la cual recibe un id y un tamaño de la tabla. Su tarea es verificar si la identificación de una persona se encuentra en el archivo dataSet.txt y de ser así retorna su valor hash, la posición donde se encuentra el elemento en el archivo ArchNombres1.txt y el nombre de la persona

- main.py

En este archivo hicimos uso de las librerías tkinter y los archivos BinaryHeap y HashFunctions.

La función main toma como argumento el tamaño de la tabla hash tomado por parámetro de una lista desplegable en la interfaz gráfica.

Lo que hace esta función es ser el puente entre la función inserhashTable que además de insertar datos en la tabla hash, también cuenta las colisiones que son puestas globalmente para ser usadas posteriormente por la interface, otra de las habilidades de esta Función main,

es el hecho de insertar todos los datos en un montículo binario y retorna los primeros 10 registros con un método llamado getTenRegisters().

La función interface hace uso de la librería tkinter para crear una interface interactiva donde se puede seleccionar el tamaño de tablas hash para hacer las pruebas, la lista de los primeros 10 registros ordenados de menor a mayor con el método del montículo binario y un cuadro donde podemos ingresar un código para que nos muestre en pantalla toda la información asociada a este código.

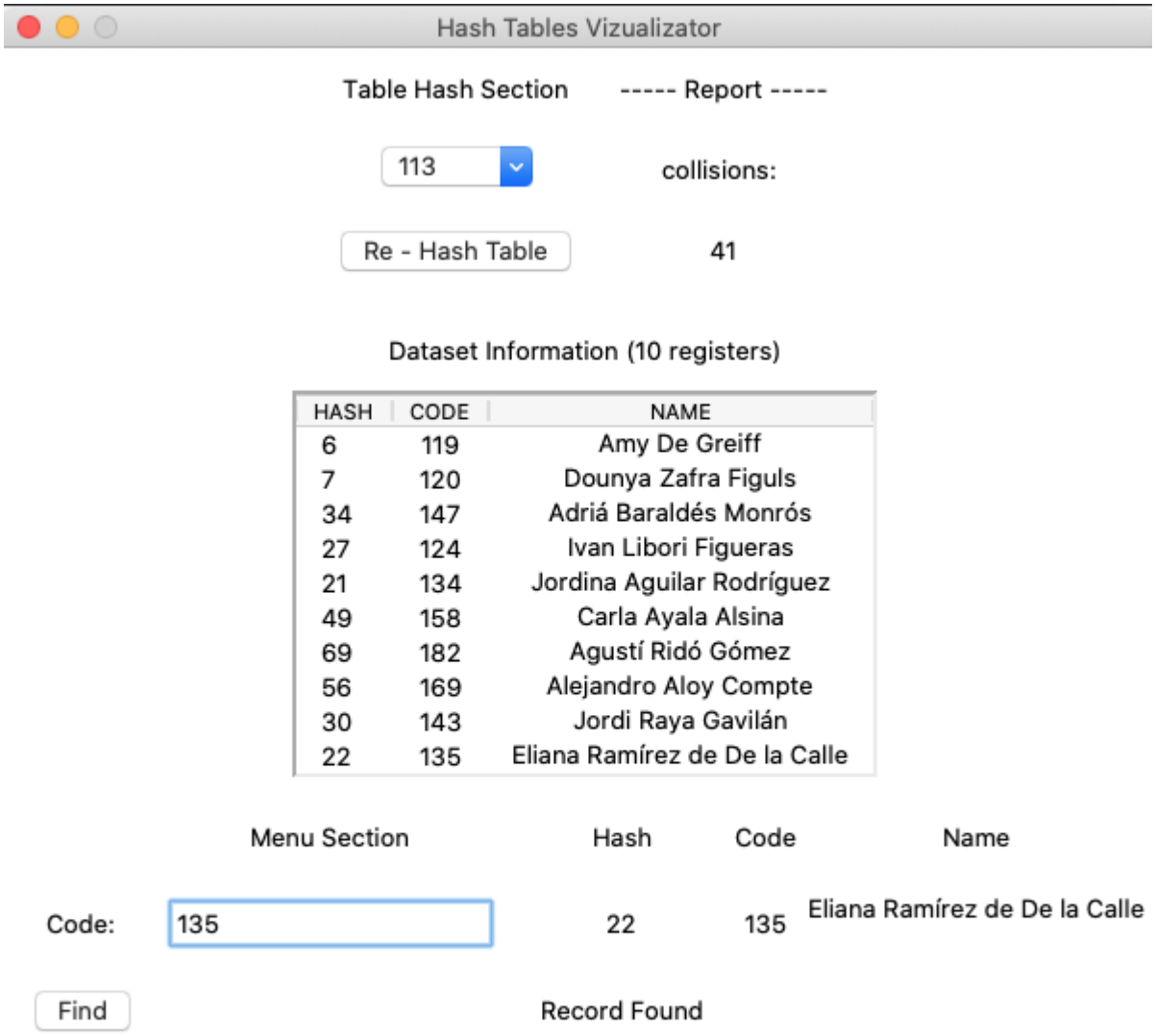
La función interface() como ya se mencionó hace uso de la tecnología tkinter, que está basada en posiciones absolutas, posiciones relativas y método de rejilla para ubicar los componentes, este último siendo el utilizado en este proyecto.

La interface está dividida en 3 grandes frames que a su vez contienen los minicomponentes como botones, labels, o inputs box.

Los frames principales son menu Frame, que tiene todo lo referente con la búsqueda de información por su código único gracias al botón que lleva a una función que se encarga de esto.

El frame Table Hash que es el que contiene la lista desplegable con los tamaños de tabla sugeridos en la descripción del proyecto y el botón re-hash table que crea la tabla hash de nuevo y retorna la cantidad de colisiones.

Y por último el frame dataset que muestra la información ordenada por el método de montículo binario de menor a mayor, siendo los padres menores a los hijos.



BinaryHeap.py

La Clase BinaryHeap es calcada del libro Problem Solving with Algorithms and Data Structures using Python. Sin embargo, se implementó algo un poco distinto, en lugar de guardar un solo dato en cada espacio del montículo, para nuestro caso guarda una tupla con el código de la persona y el index o posición en archivo original para poder traer la información con facilidad a la hora de mostrarla en la interface; además de una función que trae de la

estructura de datos los primeros 10 registros teniendo claro que esta estructura tiene el primer espacio vacío.

- dataset.txt

La información de la tabla hash queda registrada en un archivo txt llamado dataSet.txt.

conclusiones:

- Se eligen tablas hash que tengan como tamaño un número primo.
- Al probar el programa con tamaños de *tabla hash* de 103, 113, 131, 163, 199, 251 y 311 en todos los casos se presentan colisiones por lo que siempre se tiene una tabla hash no perfecta.
- Mientras la tabla hash tenga un tamaño mayor, menos colisiones se van a presentar al insertar los elementos o al buscarlos.

Tamaño tabla hash	Número de colisiones
103	50
113	41
131	36
163	31
199	27
251	22
311	13

Vemos que hay una diferencia bastante significativa de colisiones entre una tabla hash de tamaño de 103 y una tabla hash de tamaño 311 (diferencia de 37 colisiones en total).

- Evidenciamos que a mayor tamaño menos colisiones se van a presentar pero más recursos se usarán por lo que es recomendable encontrar un punto medio entre estos dos parámetros.
- Mientras más grande sea la tabla hash la búsqueda de elementos en esta se acercará al O (1) (orden uno)

- A medida que aumentan las colisiones al insertar un elemento más se demora la búsqueda con la tabla hash.

- El sondeo cuadrático puede ser un algoritmo más eficiente en una tabla de direccionamiento abierta , ya que evita de una mejor manera el problema de agrupamiento que puede ocurrir con la prueba lineal , aunque lo anterior no significa que evite colisiones.

- El tamaño de la tabla hash siempre debe ser mayor al número de elementos .

Código Fuente

HashFunctions.py

```
#this file contains all functions related to hash methods

#Hash Function based on % mod operator, in a id from a person and hashTableSize
this parameter is selected from Gui
def hashFunction(id, hashTableSize):
   return id % hashTableSize

def fileWriter(myHashDict : dict):
   fileName = 'dataset.txt'
   myFile = open(fileName, 'w', encoding='utf-8')
   for key in myHashDict:


myFile.write(str(key)+';'+str(myHashDict[key][0])+';'+str(myHashDict[key][1])+'\n')

#Insert to hash table function, write on a file the hash Value, original position
on file like a tuple and return amounth of collisions to write a simple value
def insertHashTable(idList:list, hashTableSize):
   myHashDict = {i:(None,None) for i in range(hashTableSize)}
   collisions = 0
   for id in idList:
       hashValue = hashFunction(id,hashTableSize)
       if ( myHashDict[hashValue][0] == None):
           myHashDict[hashValue] = (id,idList.index(id))
       else:
           collisions += 1
           a=hashValue
           jump = 1
           while(myHashDict[hashValue][0] != None ):
               hashValue = hashFunction((a + (jump)**2 ), hashTableSize)
               if (hashValue >= hashTableSize):
                   hashValue -= hashTableSize
               jump+=1
           myHashDict[hashValue] = (id, idList.index(id))
   fileWriter(myHashDict)
   return collisions

# this Function contains, return information of a person by his id, and

def contains(id, hashTableSize):
   namesDatabase = open('ArchNombres1.txt', encoding='utf-8').readlines()
   datasetFile = open('dataset.txt',encoding='utf-8').readlines()
```

```
    hashValue=hashFunction(id, hashTableSize)
    if ( int(datasetFile[hashValue].split(';')[1]) == id):
        return (hashValue,
namesDatabase[int(datasetFile[hashValue].split(';')[2])].split(',')[0],
namesDatabase[int(datasetFile[hashValue].split(';')[2])].split(',')[1])
    else:
        a=hashValue
        jump = 1
        while(datasetFile[hashValue].split(';')[1] != 'None' ):
                hashValue = hashFunction((a + (jump)**2),hashTableSize)
                if (hashValue >= hashTableSize):
                    hashValue -= hashTableSize
                try:
                    if (int(datasetFile[hashValue].split(';')[1]) == id):
                        return (hashValue,
namesDatabase[int(datasetFile[hashValue].split(';')[2])].split(',')[0],
namesDatabase[int(datasetFile[hashValue].split(';')[2])].split(',')[1])
                except:
                    pass
                jump+=1
        return "0"
```

BinaryHeap.py

```
class BinaryHeap():

    def __init__(self):
        self.heapList = [(0, 0)]
        self.currentSize = 0

    def insert(self, k, index):
        self.heapList.append((k, index))
        self.currentSize += 1
        self.percUp(self.currentSize)

    def findMin(self):
        return self.heapList[1]

    def isEmpty(self):
        return not bool(self.heapList)

    def buildHeap():
        pass
```

```
def minChild(self, i):
    if((i * 2 + 1) > self.currentSize):
        return i * 2
    else:
        if(self.heapList[i*2][0]< self.heapList[i*2+1][0]):
            return i * 2
        else:
            return i * 2 + 1

def percUp(self, i):
    while ( i // 2 > 0):
        if(self.heapList[i][0] < self.heapList[i // 2][0]):
            tmp = self.heapList[i // 2]
            self.heapList[i // 2] = self.heapList[i]
            self.heapList[i] = tmp
        i = i // 2

def percDown(self, i):
    while((i * 2) <= self.currentSize):
        mc = self.minChild(i)
        if(self.heapList[i][0]>self.heapList[mc][0]):
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc

def delMin(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.currentSize]
    self.currentSize -= 1
    self.heapList.pop()
    self.percDown(1)
    return retval

def getTenRegisters(self):
    return self.heapList[1:11]
```

Main.py

```
from HashFunctions import insertHashTable, contains
from tkinter import *
from tkinter import ttk
from BinaryHeap import *

collisions = 0

def main(hashTableSize):
    myBinaryHeap = BinaryHeap()
    global collisions
    fileName = 'ArchNombres1.txt'
    myFile = open(fileName, encoding='utf-8')
    idList = []
    content= myFile.readlines()
    index = 0
    for register in content:
        id = int(register.split(',')[0])
        idList.append(id)
        myBinaryHeap.insert(id, index)
        index +=1
    collisions = insertHashTable(idList, hashTableSize)
    return myBinaryHeap.getTenRegisters()



def interface():

    #main windows Section
    mainWindow = Tk()
    mainWindow.title('Hash Tables Vizualizator')
    mainWindow.geometry('600x600')
    mainWindow.resizable(0,0)
    mainWindow.config(bg='white')

    #menu frame Section
    recordFound = StringVar()
    hashValue = StringVar()
    codeValue = StringVar()
    nameValue = StringVar()
    def recordFoundFunction():
        information = contains(int(inputNumber.get()),int(hashTableSize.get()))
        if(len(information)>1):
            recordFound.set('Record Found')
            hashValue.set(information[0])
            codeValue.set(information[1])
```

```
            nameValue.set(information[2])

        else:
            recordFound.set('Record Not Found')
            hashValue.set('')
            codeValue.set('')
            nameValue.set('')




    menuFrame = Frame(mainWindow)
    menuFrame.grid(row=2, column=0, padx=10,pady=0)
    menuFrame.config(bg='white')
    menuFrame.config(width=800, height=350)
    menuLabel = Label(menuFrame, text='Menu Section')
    menuLabel.grid(row=0, column=1, padx=10,pady=10)
    inputNumber = Entry(menuFrame)
    inputNumber.grid(row=1, column=1, padx=10,pady=10)
    inputLabel = Label(menuFrame, text='Code: ')
    inputLabel.grid(row=1, column=0, padx=10,pady=10)
    findButton = Button(menuFrame, text='Find', command=recordFoundFunction)
    findButton.grid(row=2, column=0)
    hashLabel = Label(menuFrame, text= 'Hash')
    hashLabel.grid(row=0, column=2)
    codeLabel = Label(menuFrame, text= 'Code')
    codeLabel.grid(row=0, column=3)
    nameLabel = Label(menuFrame, text= 'Name')
    nameLabel.grid(row=0, column=4)
    hashValueLabel = Label(menuFrame, textvariable=hashValue)
    hashValueLabel.grid(row=1, column=2)
    codeValueLabel = Label(menuFrame, textvariable=codeValue)
    codeValueLabel.grid(row=1, column=3)
    nameValueLabel = Label(menuFrame, textvariable=nameValue)
    nameValueLabel.grid(row=1, column=4)
    recordFoundLabel= Label(menuFrame, textvariable=recordFound)
    recordFoundLabel.grid(row=2, column=2, padx=10,pady=10)


    #table Frame Section
    collisionsAmounth = StringVar()
    collisionsAmounth.set(collisions)
    def collisionsAmounthFunction():
        tenRegisters = main(int(hashTableSize.get()))
```

```
        collisionsAmounth.set(collisions)
        dataNames = open('ArchNombres1.txt', encoding='utf-8').readlines()
        treeview.delete(*treeview.get_children())
        for register in tenRegisters:
            treeview.insert('',
END,text=contains(int(register[0]),int(hashTableSize.get()))[0],
values=(register[0],dataNames[register[1]].split(',')[1]))


    tableFrame = Frame(mainWindow)
    tableFrame.grid(row=0, column=0, padx=10,pady=0)
    tableFrame.config(bg='white')
    tableFrameLabel = Label(tableFrame, text='Table Hash Section')
    tableFrameLabel.grid(row=0, column=0, padx=10,pady=10)
    hashTableSize =StringVar(tableFrame)
    hashTableSize.set('103')
    options = ['103','113','131','163','199','251','311']
    dropDown = OptionMenu(tableFrame, hashTableSize, *options)
    dropDown.grid(row=1, column=0)
    reportButton = Button(tableFrame, text='Re - Hash Table',
command=collisionsAmounthFunction)
    reportButton.grid(row=2, column=0)
    reportLabel = Label(tableFrame, text='----- Report -----')
    reportLabel.grid(row=0, column=2, padx=10,pady=10)
    collisionsLabel= Label(tableFrame, text='collisions: ')
    collisionsLabel.grid(row=1, column=2, padx=10,pady=10)
    collisionsAmounthLabel= Label(tableFrame, textvariable=collisionsAmounth)
    collisionsAmounthLabel.grid(row=2, column=2, padx=10,pady=10)

    #Dataset Frame Section this section contains 10 registers above the find
register

    datasetFrame = Frame(mainWindow)
    datasetFrame.grid(row=1, column= 0, padx=10, pady=10)
    datasetFrame.config(bg='white', width=300)
    datasetFrameLabel = Label(datasetFrame, text='Dataset Information (10
registers)')
    datasetFrameLabel.grid(row=0, column=0, padx=10, pady=10)
    treeview = ttk.Treeview(datasetFrame, columns=('col1','col2'))
    treeview.column('#0', width=50, anchor=CENTER)
    treeview.column('col1', width=50, anchor=CENTER)
    treeview.column('col2', width=200, anchor=CENTER)
    treeview.heading('#0', text='HASH', anchor=CENTER)
    treeview.heading('col1', text='CODE', anchor=CENTER)
    treeview.heading('col2', text='NAME', anchor=CENTER)
```

```
    treeview.grid(row=1)
    mainWindow.mainloop()

interface()
```

Bibliografía

- Brad Miller and David Ranum, Luther College (2006). Problem Solving with Algorithms and Data Structures using Python.