***Binary Search Trees, Traversals and Balancing in Python***

Bob es un Senior Back-end, que tiene la tarea de desarrollar una estructura de datos, para administrar la información de un perfil de usuario (Nombre de usuario, nombre real, correo electrónico) esto para 100 millones de usuarios.

La estructura de datos debe:

1. Insertar un nuevo usuario con toda la información antes mencionada.
2. Encontrar la información del usuario, a través del nombre de usuario.
3. Actualizar la información del usuario, a través del nombre de usuario.
4. Organizar o listar, a todos los usuarios, ordenados alfabéticamente por su nombre de usuario.

> Puedes asumir que los nombres de usuario son únicos.

*1. Plantee el problema claramente. Identificar los formatos de entrada y salida.*

***Problema:***

Necesitamos una estructura de datos que pueda almacenar 100 millones de registros y realizar las operaciones de insertar, encontrar, actualizar y organizar.

***Input:***

Los perfiles de usuario que contengan la información que se mencionó en el problema: (Nombre de usuario, nombre real, correo electrónico), en este caso, una clase seria una excelente forma de crear este usuario con esos datos.

class User:

    def \_\_init\_\_ (self, username, full\_name, email):

        self.username = username

        self.full\_name = full\_name

        self.email = email

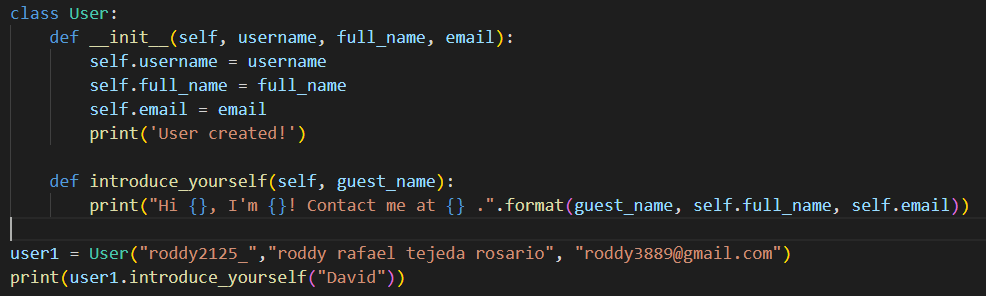
print ('User created!')

ejemplo de cómo se vería:

Texto

Descripción generada automáticamente

En este caso, se le ha agregado una presentación:



También como vamos a implementar, las funcionalidades y ya sabemos cuáles son, podemos enumerarlas como funciones en código:  
Texto

Descripción generada automáticamente

***¿Cuál es el propósito de `\_\_str\_\_` y `\_\_repr\_\_` en las clases?, ¿En qué se diferencian las dos funciones?, Muestra ejemplos.***

***`\_\_str\_\_`***

Sirve para imprimir los datos de una Class, sin tener que usar un print para cada self.

Sin el ***\_\_str\_\_:***

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente



Con el ***\_\_str\_\_:***

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media



También se puede poner en forma de formato:

def \_\_str\_\_(self):

        return f"Nombre de Usuario: {self.username} Nombre Completo: {self.full\_name} Email: {self.email}"

otra forma de llamar al método:

print(str(user1))

`\_\_repr\_\_`

Devuelve un STR, pero en más precisa a nivel interno.

Pasara exactamente los mismos ejemplos antes mostrados en esta parte.

La diferencia radica en que \_\_str\_\_ devuelve un valor que es entendible para el usuario mientras que \_\_repr\_\_ devuelve un valor entendible para el programador o la máquina.



***2. Proponga algunos ejemplos de entradas y salidas.***

Los inputs, son los perfiles de ejemplo:

# perfiles de prueba

rafael = User('rafael', 'rafael Rai', 'rafael@example.com')

rosario = User('rosario', 'rosario Das', 'rosario@example.com')

tejeda = User('tejeda', 'tejeda Jain', 'tejeda@example.com')

yosi = User('yosi', 'yosi Verma', 'yosi@example.com')

ana = User('ana', 'ana Sinha', 'ana@example.com')

jenny = User('jenny', 'jenny Kumar', 'jenny@example.com')

romeo = User('romeo', 'romeo Goel', 'romeo@example.com')

***Ejercicio: Enumerar escenarios para probar la funcionalidad de la estructura de datos:***

1. Insertar:

1. Insertar en una base de datos vacía de usuarios.

2. Intentar insertar un usuario con un nombre de usuario que ya existe.

3. Insertar un usuario con un nombre de usuario que no existe.

2. Encuentros:

1. encontrar en una lista vacía.

2. encontrar un usuario con un nombre que ya existe.

3. encontrar un usuario que no existe.

3. Actualización:

1. cuando la lista está vacía.

2. actualizar un usuario cuando hay dos con el mismo nombre.

3. actualizar un usuario que no existe.

4. Organizar:

1. organizar una lista vacía.

2. organizar unos usuarios con el mismo nombre.

3. organizar un usuario que no existe

***3. Proponga una solución correcta. Dígalo en inglés sencillo.***

Almacenar los usuarios en una lista ordenada por nombres de usuario.

1. \*\*Insertar\*\*: recorra la lista y agregue el nuevo usuario en una posición que mantenga la lista ordenada.
2. \*\*Buscar\*\*: recorra la lista y encuentre el objeto de usuario con el nombre de usuario que coincida con la consulta.
3. \*\*Actualizar\*\*: recorra la lista, encuentre el objeto de usuario que coincida con la consulta y actualice los detalles.
4. \*\*Ordenar\*\*: Devuelve la lista de objetos de usuario.

Podemos usar los nombres de usuario de hecho, que son cadenas que se pueden comparar usando los operadores `<`, `>` y `==` en Python.

***4. Implemente la solución y pruébela usando entradas de ejemplo.***

class UserDatabase:

    #Lista que almacenara los usuarios

    def \_\_init\_\_(self):

        self.users = []

    def insert(self, user):

        i = 0

        while i < len(self.users):

            # si el username de la lista es alfabéticamente mayor al username nuevo

            if self.users[i].username > user.username:

# el codigo se detiene y el nuevo usuario se inserta en la siguiente posición (i += 1)

                break

            i += 1

        # la funcion Insert agrega un elemento, tomando la posicion, el elemento a insertar

        self.users.insert(i, user)

        return print(self.users,'\nUser created!')

    def find(self, username):

        for user in self.users:

            if user.username == username:

                return user

    def update(self, user):

        # buscamos el usuario en la lista

        target = self.find(user.username)

        target.full\_name, target.email = user.full\_name, user.email

        print(target)

    def list\_all(self):

        return print(f"all users\n{self.users}")

USUARIOS:

# perfiles de prueba

users\_list = {

    "roddy" : User("roddy2125\_","roddy rafael tejeda rosario", "roddy3889@gmail.com"),

    "rafael" : User('rafael', 'rafael Rai', 'rafael@example.com'),

    "rosario" : User('rosario', 'rosario Das', 'rosario@example.com'),

    "tejeda" : User('tejeda', 'tejeda Jain', 'tejeda@example.com'),

    "yosi" : User('yosi', 'yosi Verma', 'yosi@example.com'),

    "ana" : User('ana', 'ana Sinha', 'ana@example.com'),

    "jenny" : User('jenny', 'jenny Kumar', 'jenny@example.com'),

    "romeo" : User('romeo', 'romeo Goel', 'romeo@example.com')

}

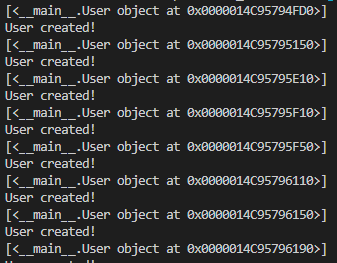
Implementación:

#INSERTAR TODOS LOS USUARIOS

for i in users\_list:

    database = UserDatabase()

    database.insert(users\_list[i])



#ENCONTRAR EL USUARIO (SI USAS UN LOOP FOR SALE NONE, PORQUE FOR LOS ALMACENA EN OTRA LISTA)

found = database.find(users\_list['roddy'].username)

print(found)



#ACTUALIZAR UN USUARIO

database.update(User(username='roddy2125\_', full\_name='Siddhant U', email='siddhantu@example.com'))



#IMPRIMIR TODA LA LISTA

database.list\_all()



***5. Analizar la complejidad del algoritmo e identificar ineficiencias***

Las DEF insertar, buscar, actualizar iteran sobre la lista, en el peor de los casos, tomaría N iteraciones para devolver un resultado, list\_all, solo devuelve una lista que fue creada.

Insertar: O(N)

Hallar: O(N)

Actualización: O(N)

Lista: O(1)

Ejercicio: Verifica que la complejidad espacial de cada operación es O(1).

En el siguiente ejemplo, se toma en cuenta que 100 millones de usuarios se inserten, solo es un ejemplo.

%%time

for i in range(100000000):

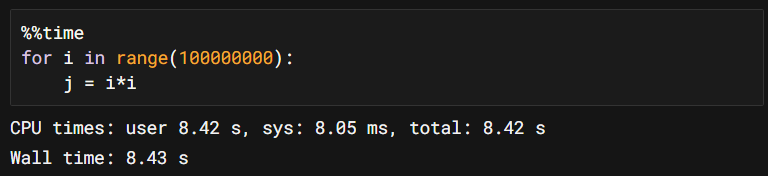
    j = i\*i

Toma alrededor de 10 segs ejecutar las iteraciones.

Un retraso de 10 segs para obtener los perfiles dará lugar a una exp de usuario no tan buena, puede que algunos no usen la plataforma por lo lenta que es.

El tiempo de 10 segs para cada perfil, limitaría la cantidad de usuarios que podrían acceder a la plataforma, los costos en la nube serian de millones.

Como ingeniero senior de back-end, ¡debe crear una estructura de datos más eficiente! Es evidente que una lista ordenada de usuarios podría no ser la mejor estructura de datos para organizar la información de perfil de millones de usuarios.

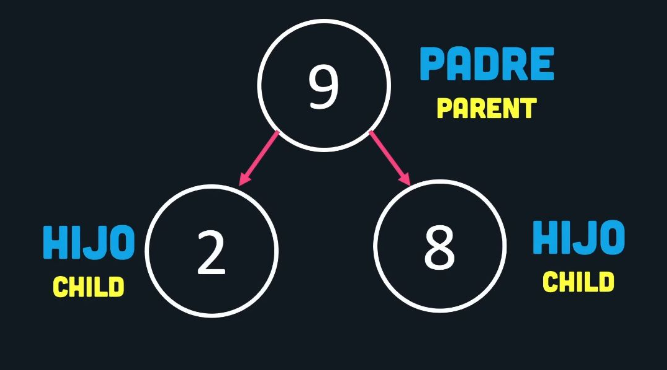


***6. Aplicar la técnica adecuada para superar la ineficiencia***

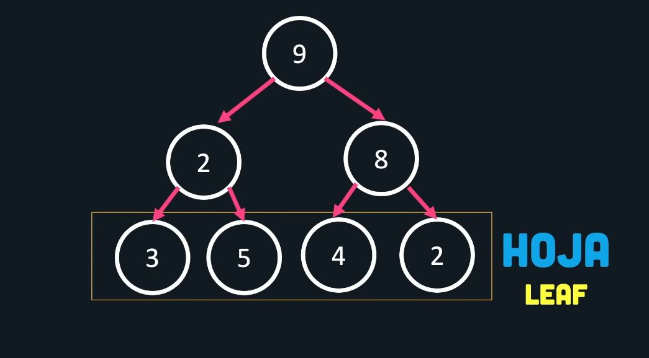
La estructura se llama ARBOL BINARIO.

Se dice que es binario porque cada NODO tiene un máximo de 2 Hijos (izquierdo o derecho).

El nodo principal es llamado NODO RAIZ y es el que almacena las operaciones (Búsqueda, Insertar, Actualizar, ect..).

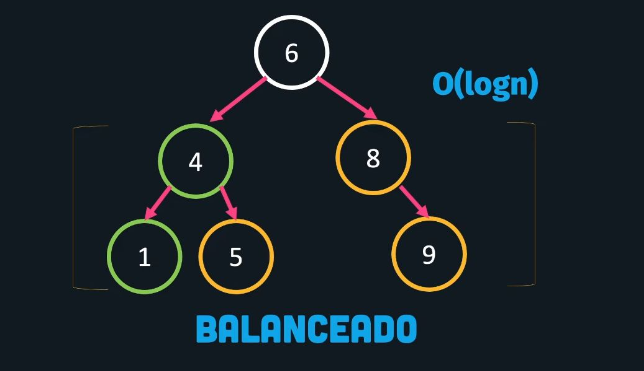


Los nodos que no tienen hijos se llaman Hojas.

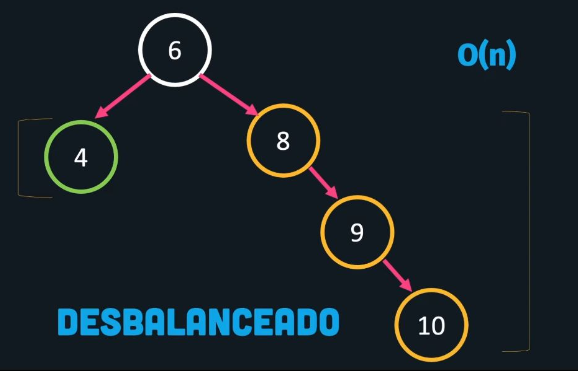


En los árboles de búsquedas binarios se aplica lo de que a la izquierda son menores que el raíz y a la derecha son mayores que el raíz y que es como se ve un árbol Balanceado, un árbol balanceado, solo puede tener máximo 1 nivel de diferencia.

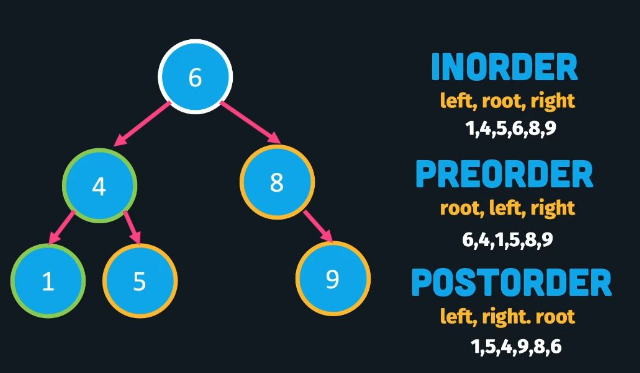
Su complejidad es 0(LOG N).



Al contrario de un árbol desbalanceado, en este caso hay dos niveles de diferencia, haciendo esto un 0(N).



Ordenes de recorrido de un árbol binario.



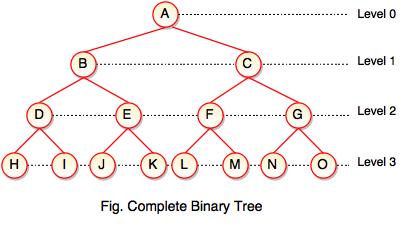
***Árboles de búsqueda binarios equilibrados***

Para este árbol Binario en especial necesitamos que tenga algunos requerimientos:

1. Clave (un nombre de usuario) y valor (un objeto `Usuario`) (DICT), un árbol que tiene claves y valores se llama mapa de árbol.
2. Árbol de búsqueda binaria, los nodos del lado izquierdo son menores al raíz, y el derecho mayores al nodo raíz.
3. Árbol equilibrado, cuando el árbol no difiere en más de 1 nivel.

***Altura de un árbol binario***

El número de niveles en un árbol se llama su altura. Como puede ver en la imagen de arriba, cada nivel de un árbol contiene el doble de nodos que el nivel anterior.



Cada nivel tiene una capacidad de Nodos.

Nivel 0: `1`Nodo.

Nivel 1: `2` Nodos.

Nivel 2: `4` Nodos, es decir, `2^2`.

Nivel 3: `8` Nodos, es decir, `2^3`.

Esto es Binario porque para aplicar las funciones antes mencionadas, necesitamos dividir la lista para elegir un tramo, y esto es 0(LOG N).

La forma de saber cuántos niveles necesitamos para almacenar una X cantidad de usuarios es: ***N + 1 = 2^k***

Ejemplo:

Necesitamos una estructura de datos para almacenar información de 15 usuarios, podemos calcular la altura del árbol de la siguiente manera:

N = 15 (Usuarios)

N + 1 = 16

16 = 2^k (Debemos calcular la potencia que, de 16, en este caso.) (K = el número de niveles)

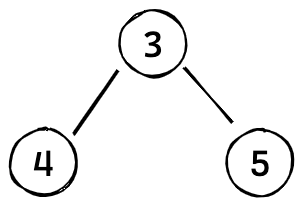
16 = 2^4

16 = 16

Por ende, necesitamos de un árbol binario de 4 niveles para almacenar 15 usuarios.

***Árbol binario***

Creamos un árbol binario simple, sin las implementaciones del problema:



class TreeNode:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

    def \_\_str\_\_(self):

        return str(self.key)

Ahora, vamos a crear Class que representen los Nodos de la IMG:

node0 = TreeNode(3)

node1 = TreeNode(4)

node2 = TreeNode(5)

forma de llamar a los nodos:

print(node0.key)

print(node0)

Vinculamos el nodo raíz con los otros nodos que fueron creados, esto usando las variables: ***LEFT*** y ***RIGHT***

# IGUALAMOS LOS DOS NODOS ANTERIORMENTE CREADOS AL NODO RAIZ

node0.left = node1

node0.right = node2

print(node0)

print (node0.left)

print(node0.right)



Ahora Podemos crear una variable que contendrá todo el árbol

tree = node0

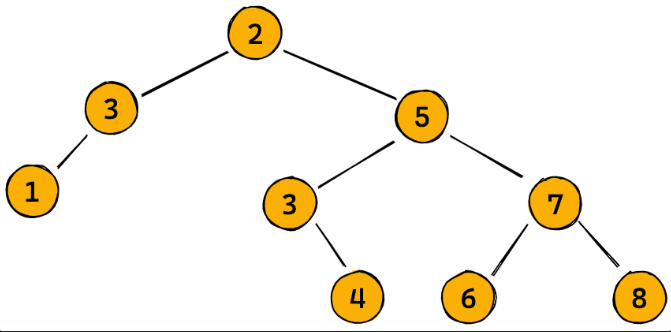
print(tree.key)

print(tree.left)

print(tree.right)



***Cree el siguiente árbol binario utilizando la clase `TreeNode` definida anteriormente.***



Rama LEFT:

nodo\_2 = TreeNode(2)

nodo\_3 = TreeNode(3)

nodo\_3.left = TreeNode(1)

print (nodo\_2)

print (nodo\_3 )

print (nodo\_3.left)

Rama RIGHT:

nodo\_5 = TreeNode(5)

nodo\_5.left = TreeNode(3)

nodo\_5.left.right = TreeNode(4)

nodo\_5.right = TreeNode(7)

nodo\_5.right.right = TreeNode(8)

nodo\_5.right.left = TreeNode(6)

print ("--- RAMA RIGHT ---")

print (nodo\_5)

print ("--- RAMA RIGHT (LEFT) ---")

print (nodo\_5.left)

print (nodo\_5.left.right)

print ("--- RAMA RIGHT (RIGHT) ---")

print (nodo\_5.right)

print (nodo\_5.right.right)

print (nodo\_5.right.left)

así se ve por consola.

Texto

Descripción generada automáticamente

VERSION ALTERNATIVA DE IMPRIMR EL ARBOL:

print ("--- NIVEL 0 ---","\n")

print ("\t", nodo\_2)

print ("--- NIVEL 1 ---","\n")

print (nodo\_3, "\t","\t", nodo\_5,"\n", end= " ")

print ("--- NIVEL 2 ---")

print (nodo\_3.left,"\t", nodo\_5.left, "\t","\t", nodo\_5.right,"\n", end=" ")

print ("--- NIVEL 3 ---")

print (nodo\_5.left.right, "\t","\t", nodo\_5.right.left, "\t","\t", nodo\_5.right.right,"\n", end=" ")

Texto

Descripción generada automáticamente

Ahora con una función, La forma de aplicar el árbol en una función es con recursividad:

tree\_tuple = ((1,3, None), 2, ((None, 3, 4), 5, (6, 7, 8)))

ORDEN DE LECTURA DE LA TUPLA: 2 > (1,3, None) > 5 > (None, 3, 4) > (6, 7, 8) [FIJATE EN LOS PARENTESIS]

def parse\_tuple(data):

    if isinstance(data, tuple) and len(data) == 3:

        node = TreeNode(data[1])

        node.left = parse\_tuple(data[0])

        node.right = parse\_tuple(data[2])

    elif data is None:

        node = None

    else:

        node = TreeNode(data)

    return node

full\_tree = parse\_tuple(tree\_tuple)

Como la Variable: FULL\_TREE contiene todos los elementos que fueron creados, en ese código, se pueden llamar por claves.

Print (full\_tree.key)



print (full\_tree.left.key, full\_tree.right.key)



print (full\_tree.left.left.key, full\_tree.left.right, full\_tree.right.left.key, full\_tree.right.right.key)



print (full\_tree.right.left.right.key, full\_tree.right.right.left.key, full\_tree.right.right.right.key)



Ahora vamos a invertir el proceso: