	Carátula para entrega de prácticas	Código	FODO-42
		Versión	01
		Página	1/1
		Sección ISO	
		Fecha de emisión	25 de junio de 2014
Secretaría/División: División de Ingeniería Eléctrica		Área/Departamento: Laboratorios de computación salas A y B	

Laboratorios de computación salas A y B

Profesor: M.C Juan Carlos Catana Salazar

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 8

Proyecto: Árbol Binario BST Auto Balanceado

Integrante(s): Barcenas Martínez Edgar Daniel

Semestre: 2017-1

Fecha de entrega: 22-Noviembre-2016

Observaciones:

CALIFICACIÓN:

_____

Sección 1. Introducción y presentación del problema.

Un árbol binario BST es una estructura de datos que permite realizar inserción, eliminación, búsquedas en tiempo eficiente sin perder la estructura o acomodo de sus elementos.

El programa es capaz de representar gráficamente en pantalla ña organización de los elementos en el BST. Se habilito una opción para que el programa lea el árbol desde un archivo y también se habilito una opción para guardar los elementos actuales de la estructura de datos

Entre sus principales operaciones tenemos que el árbol BST auto balanceado es capaz de

a) insertar un elemento

b) eliminar algún elemento

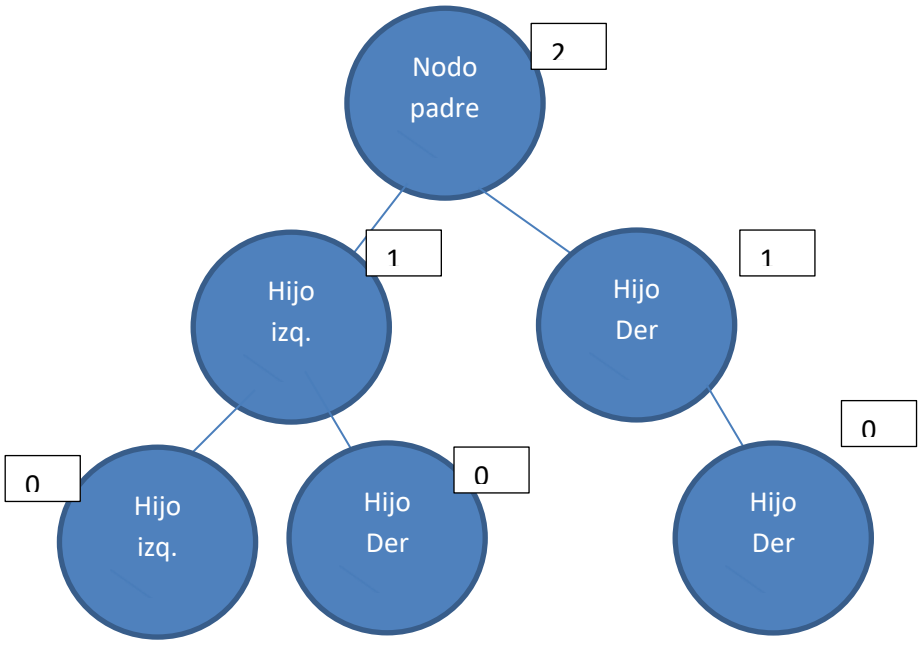
c) Buscar algún elemento

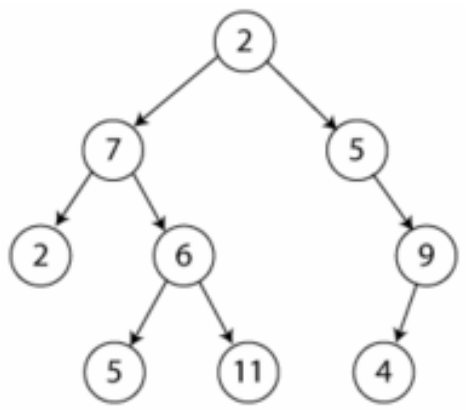
d) obtener el min y máximo del BST

e) Leer y guardar de archivo

El principal problema es lograr construir un árbol binario BST auto balanceado que a pesar de que se le hagan muchas operaciones este sea capaz de mantenerse ordenado y balanceado no excediendo más de uno de altura con respecto a otro nodo, además de ser capaz de representarse gráficamente en la pantalla para poder ser visualizado de forma fácil. También dicho programa es capaz de abrir y guardar el árbol en un archivo. Debido a esto dicho Árbol Binario BST Autobalanceable debe de ser capaz de tener operaciones como eliminar, buscar, insertar en tiempo O (log n) y su recorrido en orden O (n)

Sección 2. Definición intuitiva (ejemplo didáctico) y definición formal.



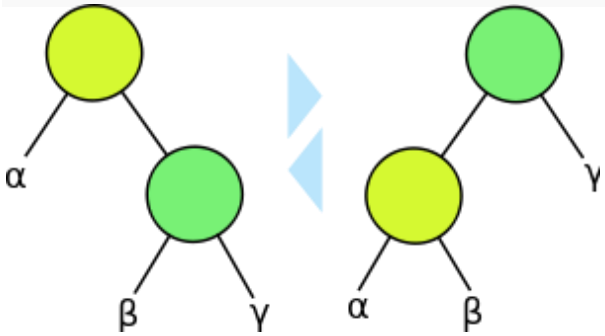


Dicho Árbol BST para que este auto balanceado debe ser capaz de mantener la altura entre dos nodos hijos no mayor a 1 por lo cual si se tiene que un nodo es mayor a otro nodo entonces el arbolo no está balanceado, si es que el árbol no está balanceado entonces se debo de rotar hacia la derecha o izquierda dependiendo del nodo que se encuentre con una altura mayor a 1 respecto a otro nodo. Además dicho árbol debe cumplir con la propiedad de que todos los nodos que están al lado izquierdo deben de ser menores al nodo padre y el nodo padre debe de ser menor a los nodos del lado derecho por lo que este tipo de árbol siempre tiene sus nodos ordenados.

Además el Árbol Binario Auto balanceado cumple con operaciones como búsqueda, inserción, Eliminación de orden O (log n) con una iteración de orden O(n), lo cual significa que este tipo de árboles son muy eficientes y rápidos

Operación	Tiempo en cota superior asintótica
Búsqueda	O(log *n*)
Inserción	O(log *n*)
Eliminación	O(log *n*)
Iteración en orden	O(*n*)

Rotaciones

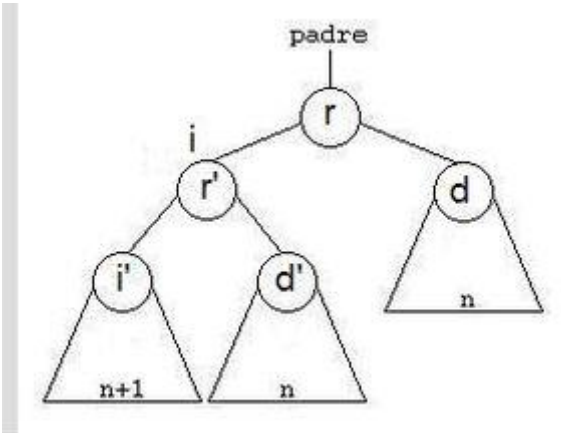


Las rotaciones internas en árboles binarios son operaciones internas comunes utilizadas para mantener el balance perfecto.Un árbol balanceado permite operaciones en tiempo logarítmico

El reequilibrado se produce de abajo hacia arriba sobre los nodos en los que se produce el desequilibrio. Pueden darse dos casos: rotación simple o rotación doble; a su vez ambos casos pueden ser hacia la derecha o hacia la izquierda.

Rotación simple a la derecha

De un árbol de raíz (r) y de hijos izquierdo (i) y derecho (d), lo que haremos será formar un nuevo árbol cuya raíz sea la raíz del hijo izquierdo, como hijo izquierdo colocamos el hijo izquierdo de i (nuestro i') y como hijo derecho construimos un nuevo árbol que tendrá como raíz, la raíz del árbol (r), el hijo derecho de i (d') será el hijo izquierdo y el hijo derecho será el hijo derecho del árbol (d).



Definición formal:

Un **árbol binario** es un **árbol** en el que ningún nodo puede tener más de dos subárboles. En un **árbol binario** cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho.

Árbol binario de búsqueda

Sea *A* un árbol binario de raíz *R* e hijos izquierdo y derecho (posiblemente nulos) H_I y H_D , respectivamente.

Decimos que *A* es un árbol binario de búsqueda (ABB) si y solo si se satisfacen las dos condiciones al mismo tiempo:

- "H_I es vacío" ("*R* es *mayor* que todo elemento de H_I" "H_I es un ABB").
- "H_D es vacío" ("*R* es *menor* que todo elemento de H_D" "H_D es un ABB").

Donde " " es la conjunción lógica "y", y " " es la disyunción lógica "o".

Para una fácil comprensión queda resumido en que es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores.

Un **árbol binario de búsqueda auto-balanceable** o **equilibrado** es un árbol binario de búsqueda que intenta mantener su *altura*, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente. Esto es importante, ya que muchas operaciones en un árbol de búsqueda binaria tardan un tiempo proporcional a la altura del árbol, y los árboles binarios de búsqueda ordinarios pueden tomar alturas muy grandes en situaciones normales, como cuando las claves son insertadas en orden. Mantener baja la altura se consigue habitualmente realizando transformaciones en el árbol, como la rotación de árboles, en momentos clave.

Definición formal

Definición de la altura de un árbol

Sea T un árbol binario de búsqueda y sean T_i y T_d sus subárboles, su altura $H(T)$, es:

- 0 si el árbol T contiene solo la raíz
- $1 + \max(H(T_i), H(T_d))$ si contiene más nodos

Definición de árbol AVL

- Un árbol vacío es un árbol AVL
- Si T es un árbol no vacío y T_i y T_d sus subárboles, entonces T es AVL si y solo si:
  - T_i es AVL
  - T_d es AVL
  - $|H(T_i) - H(T_d)| <= 1$

Sección 3. Descripción de él o los algoritmos implementados en el proyecto.

Búsqueda Binaria:

Fue implementada para realizar la búsqueda de algún elemento en el Árbol binario BST auto balanceado. Este algoritmo debe de tener una clave y una posición para buscar si el elemento buscado se encuentra en el nodo si es así entonces nos imprime el nodo y su correspondiente altura. De lo contrario se verifica si el elemento a buscar es menor o es mayor, si es menor vuelve a llamar a la función de forma recursiva pasándole la clave y una nueva posición, si es mayor entonces se va hacia los hijos del lado derecho vuelve a pasar el valor a buscar y una nueva posición.

```
def buscar(raiz, clave):
# busca el valor clave dentro del arbol
if raiz == None:
    print 'No se encuentra'
else:
    #
    if clave == raiz.dato:
       print 'El valor ',clave,' se encuentra en el ABB'
    elif clave < raiz.dato:
        # lado izquierdo
       return buscar(raiz.izq, clave)
    else:
       # lado derecho
       return buscar(raiz.der, clave)
```

Sección 3.1. Explicación del código del proyecto (secciones importantes).

Se crea una clase Vertex la cual ayudara a crear las hojas del Árbol

```
class Vertex:
    def __init__(self,vertex):
            self.id=vertex #vertice == id
            self.padre=None
            self.hizq=None
            self.hder=None
            self.altura=-1
            self.parent=None
```

Se Crea la clase Árbol la cual contendrá atributos como raíz, nodos a rotar y un vértice que contiene un diccionario.

```
class Arbol:
    def __init__(self):
        self.raiz=None
        self.nodesToRotate=[]
        self.vertices={}
```

Si tenemos inicialmente como parámetro un árbol vacío se crea un nuevo nodo como único contenido el elemento a insertar. Si no lo está, se comprueba si el elemento dado es menor que la raíz del árbol inicial con lo que se inserta en el subárbol izquierdo y si es mayor se inserta en el subárbol derecho.

```
def agregar(self,current,vertex): #se reciben el nodo raiz, vertice
    if current.id < vertex.id : # En este caso existe algun hijo
        if current.hder != None:#no tiene hijo derecho
            self.agregar(current.hder,vertex)#se vuelve a llamar la funcion
        else:
            vertex.parent=current #se agrega el hijo
            current.hder=vertex #asigno al objeto de la clase vertex hder al vertex
    else:
        if current.hizq != None: #si no existe hijo
            self.agregar(current.hizq,vertex)
        else:
            vertex.parent=current#se agrega el hijo
            current.hizq=vertex #asigno al objeto de la clase vertex izq al vertex
```

```
def agregarVertice(self,v):

    vertex=Vertex(v)
    self.vertices[v]=vertex  #si no existe vertice entonces se crea
    if self.raiz == None :
        self.raiz=vertex
    else:
        self.agregar(self.raiz,vertex) #si existe vertice entonces se agrega el elemento

def crearArbol(self,listOfVertices):
    for value in listOfVertices:
        self.agregarVertice(value) #se crean los vertices dependiendo
                                   #del numero de elementos del arbol
```

Para imprimir el árbol entonces se debe de recibir el nodo raíz, después se mueve al hijo izquierdo para imprimir sus elementos y luego a su hijo derecho esto es de forma recursiva.

```
def printing(self,current): #recibe el nodo raiz
    if current != None: #si arbol no esta vacio
        self.printing(current.hizq) #se mueve al hijo izquierdo y este es el nodo padre
        if current.id != 0: #imprime el valor del nodo + la altura
            print("("+str(current.id)+", "+str(current.altura)+")")
        self.printing(current.hder) #se mueve mueve al hijo derecho
    else:
        print("arbol Vacio")
```

Para buscar se utiliza la búsqueda binaria de forma recursiva.

```
def buscar(self,current,clave):
    if current == None:
        print ("No existe Arbol")
    if current != None:
        if int(current.id) == int (clave): #si la clave es igual al nodo
            print("("+str(current.id)+", "+str(current.altura)+")") #imprime nodo + altu
        elif int (clave) <int (current.id): #si la clave es menor al nodo actual
            return self.buscar(current.hizq,clave) #se llama a la funcion de forma recur
        elif int (clave)>int (current.id):# si la clave es mayor al nodo actual
            return self.buscar(current.hder,clave) #se llama ala funcion buscar los
                                                   #el elemento derecho
        else:
            ("No encontrado") #no existe elemento
```

Eliminar

Carga nuevamente la lista que contiene al árbol y remueve el valor seleccionado para después crear nuevamente el arbol

```
def eliminar(self,lista,clave):
    print(lista)
    lista.remove(clave)
    return lista
```

MinHeap

Se Recorre el árbol hacia los hijos izquierdos hasta encontrar el ultimo nodo izquierdo que no tenga hijos, el cual se imprime con su valor y altura

```
def min(self,current,aux):
    if current != None: #verifica que el arbol no este vacio
        if int (current.id) < int (aux):
            aux=current.id
            if current.hizq==None: #si nodo izquierdo no tiene hijos
                print("("+str(current.id)+", "+str(current.altura)+")")#imprime
            else:
                current=current.hizq #si tiene  hijos
                if current.hizq==None: #si ya no tiene hijos izquierdos
                    print("("+str(current.id)+", "+str(current.altura)+")") #imprime
                    return 0
                else:
                    return self.min(current,aux) #retorna el valor min con su altura
```

MaxHeap

Se Recorre el árbol hacia los hijos derechos hasta encontrar el ultimo nodo derecho que no tenga hijos, el cual se imprime con su valor y altura

```
def max(self,current,aux):
    if current != None:
        if int (current.id) > int (aux):
            aux=current.id
            if current.hder==None:
                print("("+str(current.id)+", "+str(current.altura)+")")
            else:
                current=current.hder
                if current.hder==None:
                    print("("+str(current.id)+", "+str(current.altura)+")")
                    return 0
                else:
                    return self.max(current,aux)
```

Altura

El método altura recibe el nodo raíz y si no está vacía vuelve a llamarse a sí misma para obtener los nodos izquierdos y derechos, luego con el método Max se busca cuál de los 2 nodos hijo es el mayor para luego con un if ver cuál es el nodo que se encuentra desbalanceado y así poder agregarlo a una lista de nodos desbalanceados

```
def altura(self, current):
    if current!=None:
        global a3 #llamamos al arreglo global
        a1=self.altura(current.hizq) # nodo izquierdo
        a2=self.altura(current.hder) # nodo derecho
        current.altura=max([a1,a2])+1 #Se busca el que tiene mayor altura
        if a1-a2>1: #si su altura es mayor a 1
            a3.append(current.id) # se agrega a una lista
        if a2-a1>1:#si su altura es mayor a 1
            a3.append(current.id) #se agrega a la lista
        a3.sort() #se ordena a3 con los elementos desbalanceados
        return current.altura  #se regresa la altura de cada nodo
    else:
        return -1
```

BalanceBST

Primero se buscan los nodos desbalanceados y si no tiene hijo izquierdo entonces balancea del lado izquierdo y sucede lo mismo cuando altura es mayor al nodo derecho, si no tiene hijo derecho entonces balancea del lado derecho y sucede lo mismo cuando altura es mayor al nodo izquierdo

```
def balanceBST(self,current):
    if current in self.nodesToRotate:
        for i in self.nodesToRotate:#se busca los nodos desvalanceados
            #print(i.id)
            if i.hizq == None:
                self.ll(i) #balancea del lado izquierdo
            elif i.hder == None:
                self.rr(i) #balancea del lado derecho
            elif(i.hizq.altura < i.hder.altura): #si altura derecha es menor a izquierda
                self.ll(i)#balancea del lado izquierdo
            elif(i.hder.altura > i.hizq.altura):#si altura derecha es mayor a izquierda
                self.rr(i)#balancea del lado derecho
```

RR

```
def rr(self,current):#se recibe nodo raiz
    newroot=current.hizq #se asigna hizq al newroot
    current.hizq=newroot.hder #se asigna el hijoderecho del hijo izquierdo
    newroot.hder=current #nodo raiz es asignado a h.der
    newroot.parent=current.parent#se asigna none a newroot.none
    current.parent=newroot #hizq es asignado a current none
    if current.hizq != None:
        current.hizq.parent=current
    if newroot.parent != None :
        newroot.parent.hizq=newroot
    if self.raiz == current:
        self.raiz=newroot
```

LL

```
def ll(self,current):
    newroot=current.hder
    current.hder=newroot.hizq
    newroot.hizq=current
    newroot.parent=current.parent
    current.parent=newroot
    if current.hder != None:
        current.hder.parent=current
        #asignar al nodo cambiado el nuevo padre
    if newroot.parent != None:
        newroot.parent.hder=newroot
        #asignar si no es la raiz el nodo rrotado asignarle a la raiz ese nodo
    if self.raiz == current:
        self.raiz=newroot
        #si el nodo rotado se convierto la raiz cambiar la raiz
```

Class Aplication

```
class Aplicacion():
    def __init__(self):
        self.ventana = Tk()
        # Define la dimensión de la ventana
        self.ventana.geometry("600x300")
        self.ventana.title("Arbol BST")
        c = Canvas(self.ventana,width=600,height=300)
        c.place(x=0,y=0)
        c.create_rectangle(0,0,600,300,fill="blue")
        #c.create_oval(120,120,200,200,fill="red")

        self.fuente=font.Font(family="Helvetica",size=12,weight="bold")
        self.etiq1=Label(self.ventana,text="ARBOL BST AUTOBALANCEADO",font=self.fuente).pla

        self.mensa = StringVar()
        self.mensa2 = StringVar()
        self.mensa3=StringVar()

        self.etiq3 = ttk.Label(self.ventana,text="Dato ingresado:",font=self.fuente, foregr
        self.etiq4 = ttk.Label(self.ventana,textvariable=self.mensa2,font=self.fuente, fore
        self.etiq5 = ttk.Label(self.ventana,textvariable=self.mensa3,font=self.fuente, fore

        self.insertar=IntVar()
```

Sección 4. Conclusiones del trabajo.

En este proyecto se pusieron a prueba los conocimientos adquiridos durante el curso en especial el uso de los arboles binarios BST auto balanceados en donde se lleva a cabo un gran uso de la lógica para poder implementar dicho árbol.

La elaboración de esta proyecto me ayudo a reforzar los conocimientos necesarios para implementar la elaboración de un RD (rotación doble o Double Rotation) o también el saber cómo obtener la altura de los vértices de un árbol y con esto determinar aquellos vértices que sean los hijos, es decir, su nivel de es 0.

En este proyecto aprendimos a implementar un árbol BST en el cual implementamos métodos para obtener la altura de cada nodo del árbol y de esta manera lograr balancear el árbol para que no tenga más de uno en la diferencia respecto a otro nodo y de esta forma poder llevar acabo búsquedas, inserciones, eliminar, encontrar los subárboles no balanceados, para después balancearlos a la derecha o izquierda dependiendo del tipo de árbol