Árbol Auto Balanceado

Son una estructura de datos informática la cual es capaz de almacenar información de una manera eficiente. Los árboles auto balanceados son una especie de árbol binario el cual es capaz de controlar su altura para no diferir más de un nivel de altura, se conocen como AVL.

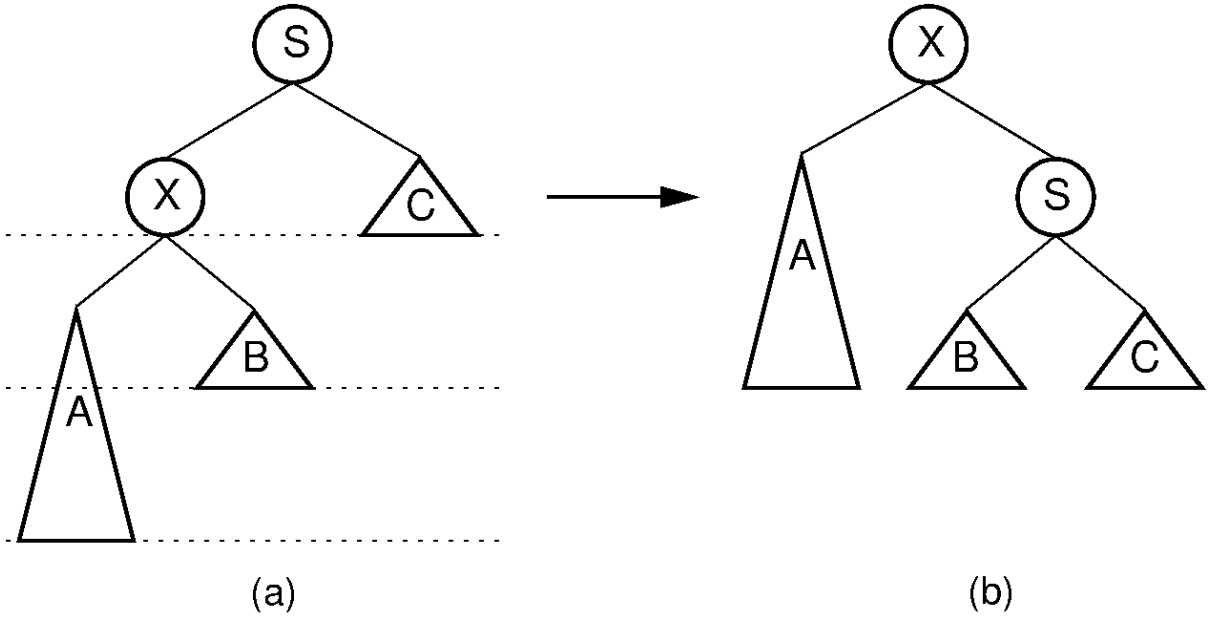

FIG 00:Un árbol autobalanceado buscando el balance.

ADVERTENCIA

LOS ÁRBOLES AUTO BALANCEADOS SE RIGEN POR LA SIGUIENTE REGLA:

- SI ES MÁS GRANDE VA A LA DERECHA.
- SI ES MENOR VA A LA IZQUIERDA.

Advertencia

Esto solo funciona con recursividad

En caso de no entender recursividad es imposible entender los árboles AVL

Por qué existen los árboles autobalanceados:

- Mejoran la eficiencia en la búsqueda. inserción y eliminación.

Un árbol autobalanceado no es más que una colección de nodos, los cuales contienen una izquierda y una derecha y almacenan un dato y a que altura se encuentra. Por ello cada nodo contiene 4 elementos:

- Izquierda.
- dato.
- altura
- Derecha.

FIG 01: representación de un nodo al lado de su respectivo código.

Para poder utilizar esto necesitamos crear algo que se llama “Árbol Autobalanceado”

Un Árbol autobalanceado es una colección de nodos que nos va a permitir guardar la información, dicha clase contiene un apuntador (el principal es la raíz)… además contiene métodos para lograr un balanceo, si observa la siguiente figura usted podrá notar cómo funciona:

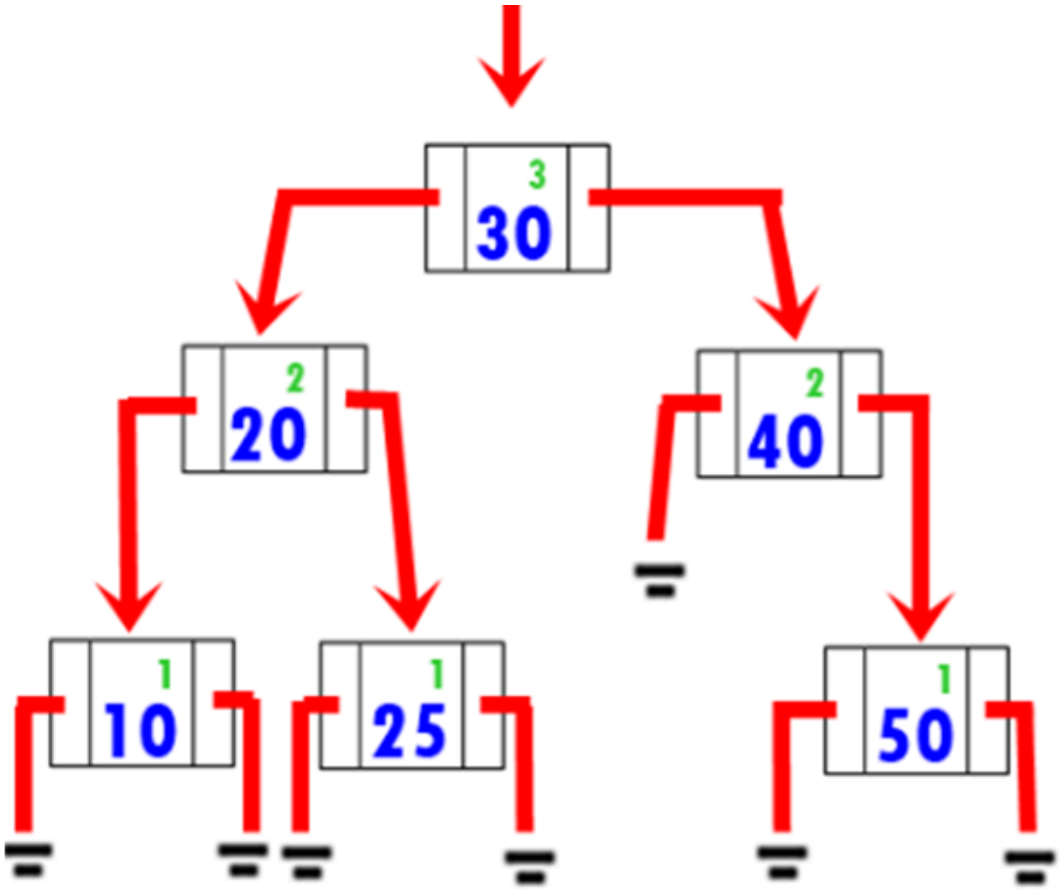

FIG 02: ejemplo visual de un árbol autobalanceado donde cada nodo contiene una altura.

Los árboles autobalanceados necesitan de los siguientes atributos y métodos para poder funcionar:

- Se necesita un pivote el cual es el encargado de moverse para leer/escribir la información.
- Se necesita un método para insertar un valor.
- Se necesita un método para calcular el equilibrio de un nodo.

- Se necesita un método para calcular la altura de un nodo.
- Se necesita un método para rotar el árbol hacia la derecha.
- Se necesita un método para rotar el árbol hacia la izquierda.
- Se necesita un método para buscar un valor.
- Se necesita un método para eliminar un valor.
- Se necesitan 3 métodos para recorrer el árbol: (In order, Pre Order y Post Orden).
- Se necesita un método para calcular la altura de un árbol.
- Se necesita un método para contar los nodos de un árbol.
- Se necesita de un método para buscar los valores mínimo y máximo.

Que es el pivote

El pivote **(raíz)** no es más que un nodo el cual vamos a utilizar para poder movernos entre los Nodos de ese árbol (Moverse entre las ramas) con este movimiento nosotros vamos a poder:

- Agregar, Eliminar, Buscar y Modificar los datos de un nodo.

El pivote es el encargado de poder ser el punto de acceso para movernos en el árbol, sin él no tendríamos una forma de recorrer el árbol.

```
class AVLTree:
    def __init__(self):
        self.root = None
```

Advertencia: El nodo principal inicia en None debido a que no existe DATA para construir el primer nodo.

Como se agregan los elementos a un árbol autobalanceado

Lo que vamos a ver a continuación es una mezcla de recursividad, balance y maromas donde podremos ver como un árbol actualiza su estructura cada vez que se inserta un nuevo nodo. A continuación vamos a detallar los conceptos básicos para agregar un elemento al árbol.

Vamos a añadir los elementos 10,20,30,40,50 y 25 a un árbol autobalanceado, la cosa es que este árbol contiene casos base y transformaciones. Pero el secreto está en que la estructura del árbol siempre se está instanciando y actualizando medidores.

Caso 01: "El árbol no tiene datos"

```
def addData(self, data):
    self.root = self._addData(self.root, data)
def _addData(self, pivot, data):
    if pivot == None:
        new_node = Node(data)
        return new_node
```

En ese caso lo que tenemos que hacer es igualar la raíz a un método recursivo el cual se encargará de crear un nuevo nodo. En pocas palabras como la raíz es nula se construye un nuevo nodo.

Caso 02: "El árbol tiene datos e insertar el nuevo nodo no rompe el balance"

```
def addData(self, data):
    self.root = self._addData(self.root, data)
def _addData(self, pivot, data):
    if pivot == None:
        new_node = Node(data)
        return new_node
    if pivot.data < data:
        pivot.right = self._addData(pivot.right, data)
    else:
        pivot.left = self._addData(pivot.left, data)
```

Cuando se necesite agregar un nuevo elemento al árbol usted tendrá que notar que la diferencia es que estamos declarando el siguiente del pivote… y al momento de declararlo llamamos recursivamente y la asignación se hace y luego al continuar con los pendientes el método se actualizarán valores como: altura y equilibrio.

```
pivot.height = 1 + max(self.getNodeHeight(pivot.left), self.getNodeHeight(pivot.right))
```

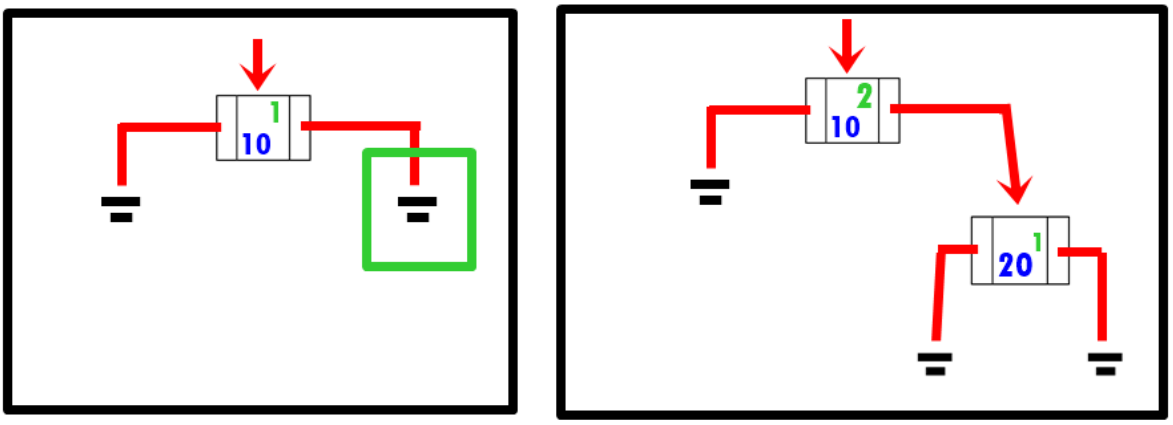

FIG 03: Insertar un valor a la derecha de un nodo.

Caso 03: “El árbol tiene datos e insertar un nuevo nodo rompe el balance”

```
def addData(self, data):
    self.root = self._addData(self.root, data)
def _addData(self, pivot, data):
    if pivot == None:
        new_node = Node(data)
        return new_node
    if pivot.data < data:
        pivot.right = self._addData(pivot.right, data)
    else:
        pivot.left = self._addData(pivot.left, data)

    pivot.height = 1 + max(self.getNodeHeight(pivot.left), self.getNodeHeight(pivot.right))
    f_equilibrium = self.getNodeEquilibrium(pivot)

    if f_equilibrium > 1:
        if data < pivot.left.data:
            return self.RR(pivot)
        else:
            pivot.left = self.RL(pivot.left)
            return self.RR(pivot)

    if f_equilibrium < -1:
        if data > pivot.right.data:
            return self.RL(pivot)
        else:
            pivot.right = self.RR(pivot.right)
            return self.RL(pivot)

    return pivot
```

Espero que tal cantidad de código no cause estrés al lector, le recomiendo conservar la calma y revisar con atención:

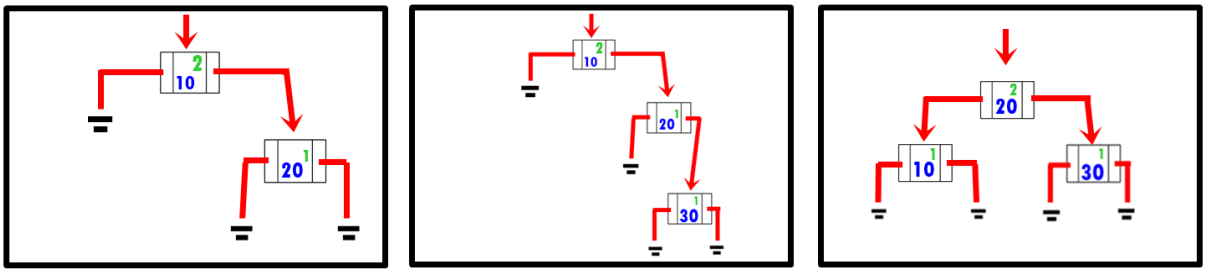

FIG 04: Insertar un elemento y luego balancear

1 - cuando se ingresa el dato 30 al árbol hay un problema de equilibrio. Primero vamos a tratar que es un problema de equilibrio: “Se trata de que un nodo contiene más de un nivel de diferencia con respecto a sus hijos” para este caso el problema del equilibrio pasa en el nodo 10. a continuación vamos a ver el factor de equilibrio de nuestros nodos:

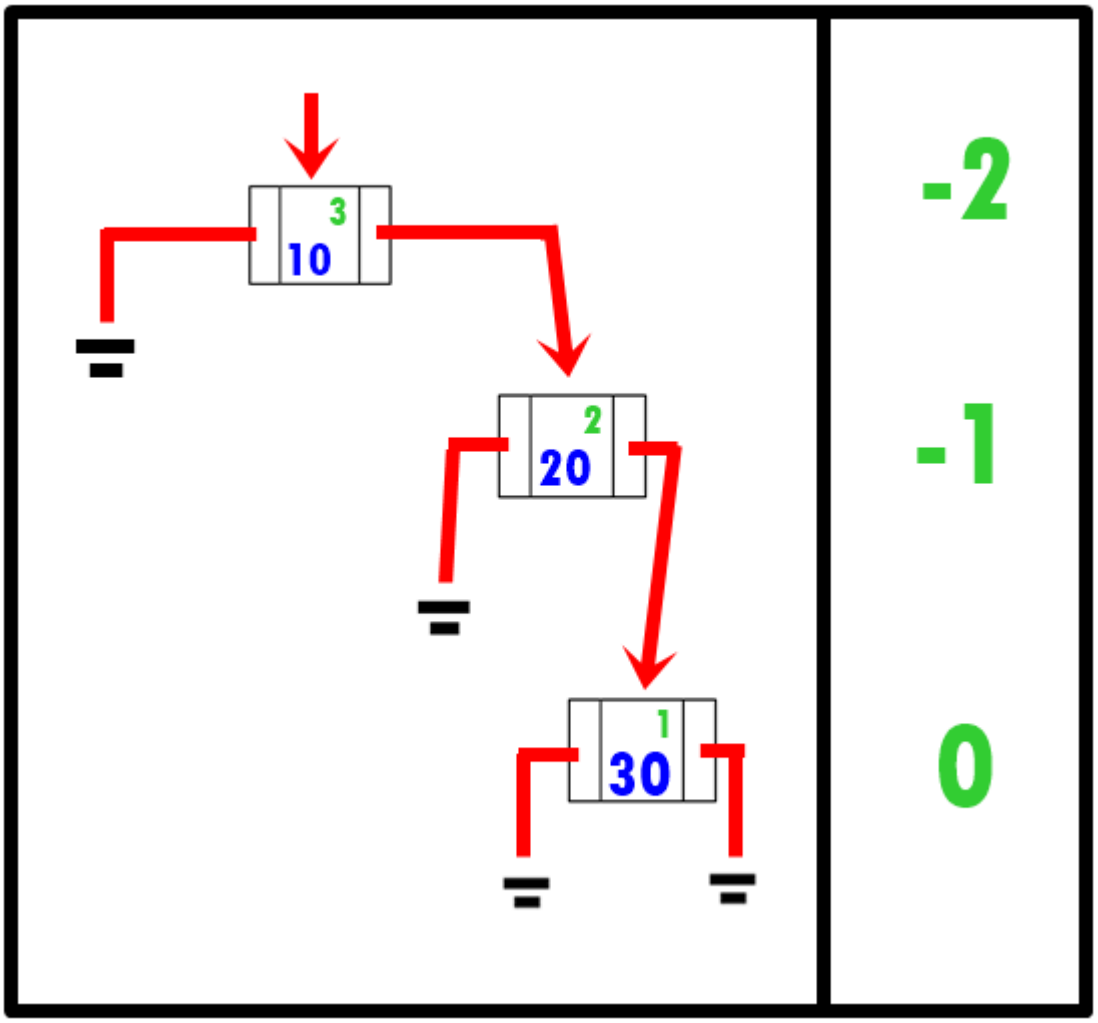

FIG 05: valores de equilibrio respectivos de cada nodo

2 - para solucionar un problema de equilibrio hay que hacer algo llamado rotación, y la rotación consiste en reasignación de hijos y conexiones. Existen 2 tipos de rotaciones a la izquierda y a la derecha. Para este caso tenemos que hacer una rotación a la izquierda:

Pasos para hacer una rotación a la izquierda:

- identificar cual es el nodo que tiene problemas.
- Se crea un nodo Y el cual será la derecha del nodo que tiene problemas.
- Se crea un nodo T2 el cual será la izquierda de Y.
- Se asigna como izquierda de Y al nodo que tiene problemas.
- Se asigna como derecha del nodo que tiene problemas a T2.
- Se actualiza la variable de altura.
- Se retorna Y

Cuando nosotros retornamos Y lo que pasará es que la raíz se asigna gracias a esta línea de código:

```
def addData(self, data):
    self.root = self._addData(self.root, data)
```

Y con ello la raíz del árbol cambia.

Cómo calcular la altura de un nodo

Cuando un nodo es creado tiene un altura de 1 por defecto y cada vez que se inserte un descendiente a ese nodo se procederá a aumentar +1 a cada nodo que tenga relación con esa dependencia.

```
class Node:
    def __init__(self, data) -> None:
        self.left = None
        self.data = data
        self.height = 1
        self.right = None
```

Cómo calcular el equilibrio

El equilibrio solo consiste en hacer una resta de sus descendientes, lo de la izquierda se le resta lo de la derecha. No se hace de manera recursiva solo se usa la variable de altura que tiene el nodo:

```
def getNodeEquilibrium(self, node):
    if node == None:
        return 0
    else:
        return self.getNodeHeight(node.left) - self.getNodeHeight(node.right)
```

Ejemplo de factores de equilibrio de un árbol:

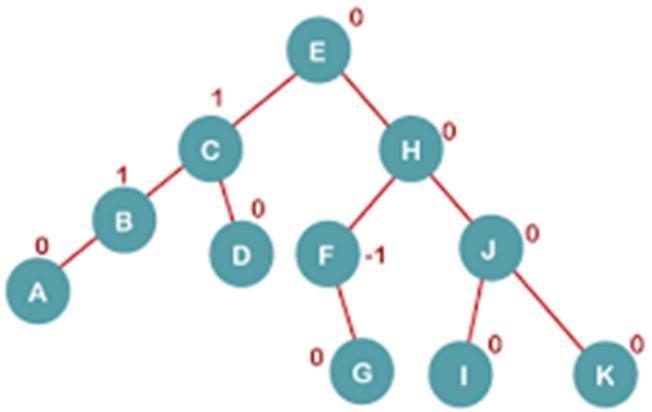

FIG 06: ejemplo de un árbol AVL con sus respectivos valores de equilibrio.

Rotar el árbol a la izquierda

```python
def RL(self, node):
    y = node.right
    T2 = y.left
    y.left = node
    node.right = T2
    node.height = 1 + max(self.getNodeHeight(node.left), self.getNodeHeight(node.right))
    y.height = 1 + max(self.getNodeHeight(y.left), self.getNodeHeight(y.right))

    return y
```

Cuando ocurre un desbalance hacia la derecha se necesita realizar una corrección, y para poder balancear necesitamos rotar a la izquierda.

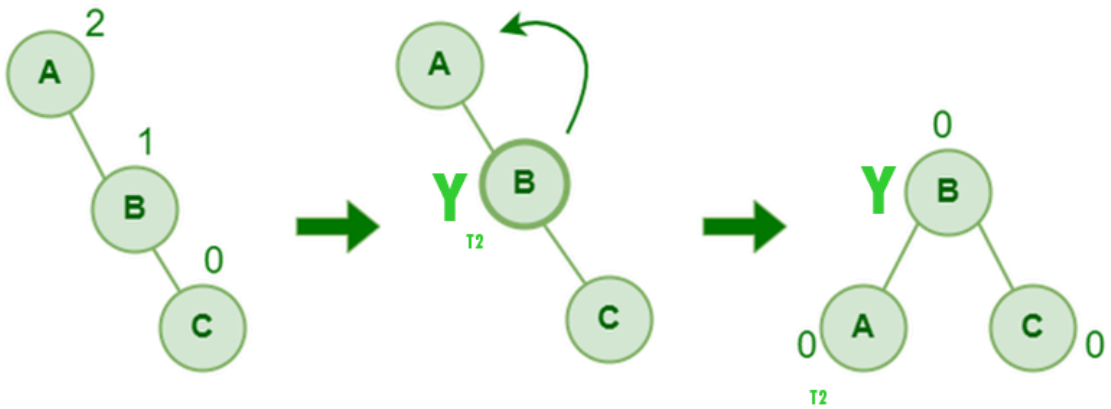

FIG 07: Se realiza una rotación a la izquierda del nodo A.

Cuando hacemos esto se cumplen los siguientes pasos:

- Se crea un nodo Y el cual será la derecha del nodo que tiene problemas.
- Se crea un nodo T2 el cual será la izquierda de Y.
- Se asigna como izquierda de Y al nodo que tiene problemas.
- Se asigna como derecha del nodo que tiene problemas a T2.
- Se actualizan las variables de altura.
- Se retorna Y

Rotar el árbol a la derecha

```
def RR(self, node):
    y = node.left
    T3 = y.right
    y.right = node
    node.left = T3
    node.height = 1 + max(self.getNodeHeight(node.left), self.getNodeHeight(node.right))
    y.height = 1 + max(self.getNodeHeight(y.left), self.getNodeHeight(y.right))

    return y
```

Cuando hay un desbalance a la izquierda se necesita realizar un movimiento a la derecha para lograr el equilibrio.

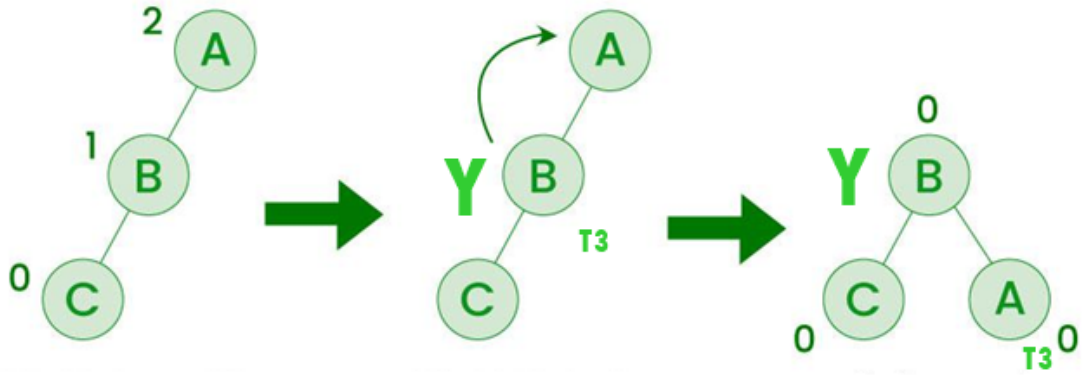

FIG 08: Se realiza una rotación a la derecha del nodo A.