Estructuras de Datos

2020

http://xkcd.com/835/

AVL
Árboles Balanceados

AVL

Para que los árboles binarios puedan ser considerados de búsqueda deberían ser "balanceados"

Se entiende por árbol balanceado AVL (ideado por los matemáticos rusos Adelson-Velskii y Landis) aquel donde la diferencia entre el camino más largo y el más corto desde la raíz a las hojas es "EL MISMO" o al menos un valor acotado.

Los árboles binarios de búsqueda no tienen por que estar balanceados

AVL

Se debe analizar algún tipo de procedimiento que me asegure que los árboles se mantienen balanceados

Elegir “bien” la raíz. Como hacerlo?

Poner todo en “otra estructura” y luego pasarlo a un árbol? ????

Construir el árbol y cuando se desbalancea reacomodar?

Construir el árbol de otra forma de manera que quede siempre balanceado?

AVL

Al tener un AVL tenemos máxima eficiencia en la búsqueda :-)

¿Cuál es la desventaja? :-(

La propiedad de equilibrio de los árboles AVL implica una *dificultad a la hora de insertar o* eliminar elementos:

estas operaciones pueden no conservar dicha propiedad.

Algunas definiciones

AVL. Algunas definiciones

Definición de la altura de un árbol:

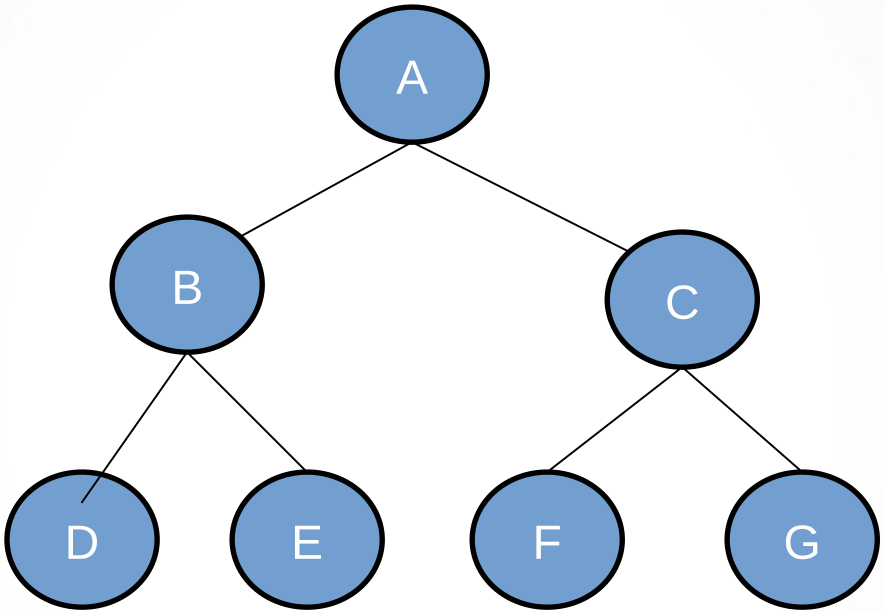
Sea T un árbol binario de búsqueda y sean T_i y T_d sus sub-árboles, su altura H(T), es:

1 si el árbol T contiene solo la raíz

1 + max(H(T_i), H(T_d)) si contiene más nodos

Nivel : El nivel de un nodo está definido por 1+ el número de conexiones entre el nodo y la raíz.

AVL. Algunas definiciones

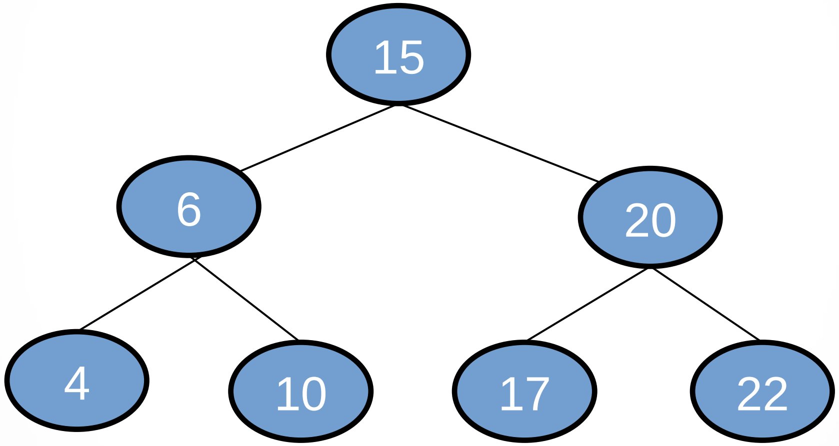


La altura del árbol es 3.

La altura del subárbol B es 2.

AVL. Equilibrio

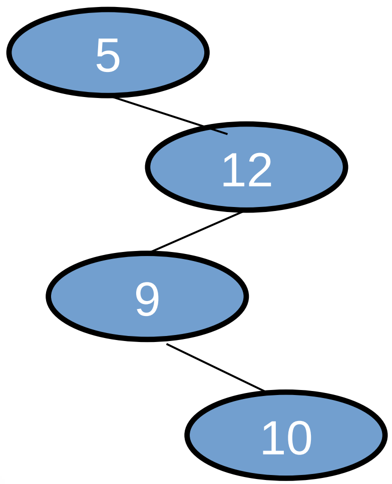
Árbol Lleno: tiene el máximo número de entradas para su altura



Árbol Lleno

AVL. Equilibrio

Árbol Degenerado: hay un solo nodo hoja (el 10) y cada nodo no hoja tiene solo un hijo. Equivalente a una lista enlazada.



Árbol Degenerado

AVL. Algunas definiciones

Definición de árbol AVL

Un árbol vacío es un árbol AVL

Si T es un árbol no vacío y T_i y T_d son sus sub-árboles, entonces T es AVL si y solo si:

T_i es AVL

T_d es AVL

$|H(T_i) - H(T_d)| <= 1$

AVL. Equilibrio

El factor de equilibrio es la diferencia entre las alturas del árbol izquierdo y el derecho:

FE = altura sub-árbol izquierdo - altura su-bárbol derecho = $H(T_i) - H(T_d)$

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

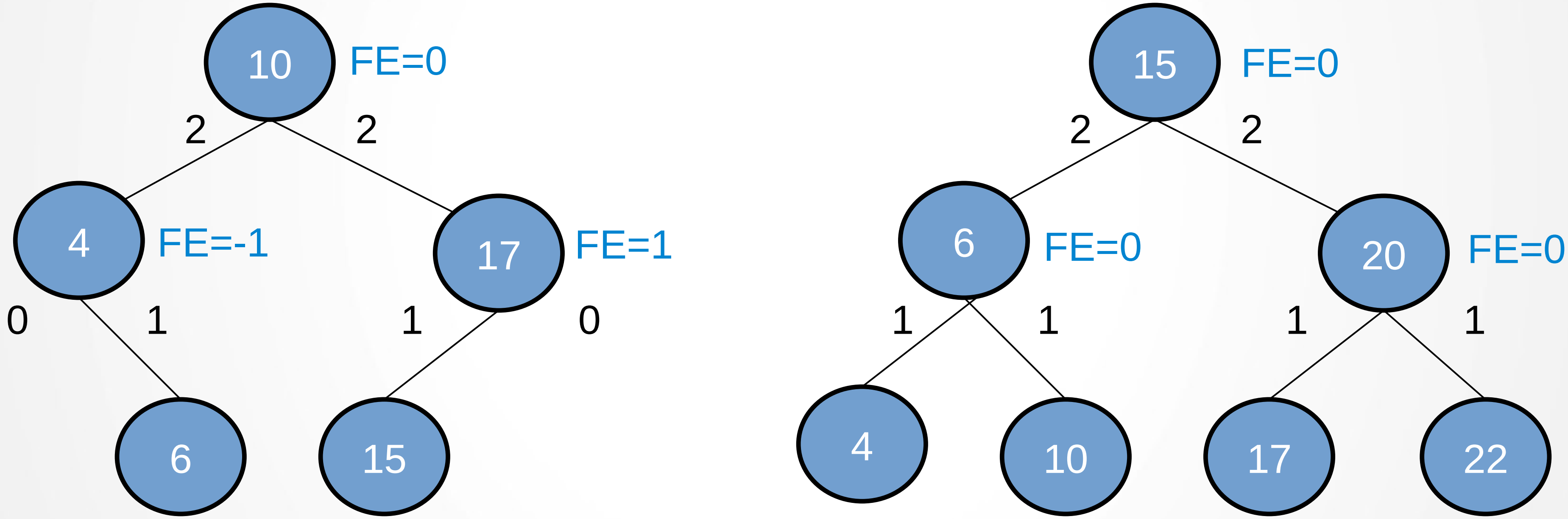
Si el factor de equilibrio de un nodo es:

- 0: El nodo está equilibrado y sus sub-árboles tienen exactamente la misma altura.
- 1: El nodo está equilibrado y su sub-árbol izquierdo es un nivel más alto.
- -1: El nodo está equilibrado y su sub-árbol derecho es un nivel más alto.
- Si el factor de equilibrio |FE| >= 2, es necesario reequilibrar el árbol cuya raíz es el nodo para el que se calculó el FE.

AVL. Equilibrio

Vamos a definir “Factor de Equilibrio” como la diferencia de altura entre los 2 subárboles de un nodo.

FE = altura(T_i) – altura(T_d).



Árbol Equilibrado: |FE| ≤ 1

Árbol Perfectamente Equilibrado: FE= 0

AVL. Equilibrio

Los árboles AVL están siempre equilibrados de tal modo que para todos los nodos:

La altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa.

Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad O(log n).

El factor de equilibrio para cada nodo debe ser computado a partir de las alturas de sus sub-árboles, por lo que resulta conveniente **almacenar directamente en cada nodo la altura que tiene el árbol del que ese nodo es la raíz.**

AVL. Especificación

¿Deberíamos cambiar algo en la especificación del ABB?

¿Cambian las operaciones?

No cambia la interfaz…

Si van a cambiar algunas implementaciones

Y el orden del” buscar”

Y la **estructura Interna**?

AVL. Estructura Interna

```
class AVLDict():
   @dataclass
   class _Node:
          key: Any  # Comparable
          value: Any
          height: int
          parent: Union['_Node', '_Root'] = None
          left: '_Node'= None
          right: '_Node'= None
    @dataclass
    class _Root:
          left: '_Node' = None
          right: '_Node' = None
          parent: '_Node' = None

    __slots__ = ['_root', '_len']
```

Altura del sub-árbol para el cuál este nodo es la raíz.

AVL. Implementación

¿Deberíamos cambiar algo en la implementación del ABB?

¿Cambian las implementaciones de las operaciones?

Van a cambiar algunas implementaciones

¿Qué implementaciones deberían cambiar?

AVL. Equilibrio

Para conseguir esta propiedad de equilibrio, tenemos que extender **la inserción** y **el borrado** de los nodos que implementamos para los ABB.

Si al realizar una operación de inserción o borrado, puede romperse la condición de equilibrio y, de ser así, hay que realizar una serie de **rotaciones de los nodos** para restablecer el equilibrio en el árbol.

AVL. Inserta

En cada nodo en el recorrido realizado para la inserción _balance_tree verifica que este balanceado

```
def insert(self, key, value=None):
   def do_insert(node, parent):
       if node is None:
            node = self._Node(key, value, 1, parent)
            coord = TreeDict._Coordinate(node)
            self._len += 1
       elif key == node.key:
            node.value = value
            coord = TreeDict._Coordinate(node)
       else:
            if key < node.key:
                node.left, coord = do_insert(node.left, node)
            else:  # key > node.key:
                node.right, coord = do_insert(node.right, node)
            node = self._balance_tree(node)
       return node, coord
  self._root.left, coord = do_insert(self._root.left,   self._root)
  return coord
```

AVL. Rotaciones

Una inserción cambia al árbol agregando un nuevo nodo.

Una eliminación lo cambia quitando un nodo del árbol.

Ambas operaciones **pueden cambiar la altura del árbol**, por lo que desde el punto donde se hizo el cambio, deberá regresarse hasta la raíz tratando de **rebalancear** cada subárbol desbalanceado.

La forma de balancear un árbol AVL consiste en hacer **rotaciones** que cambien **la posición de ciertos nodos** sin alterar el **invariante** del árbol, es decir, que todos los elementos menores al nodo actual se encuentran en el sub-árbol de la izquierda y que todos los elementos mayores están en el sub-árbol de la derecha.

AVL. Rotaciones

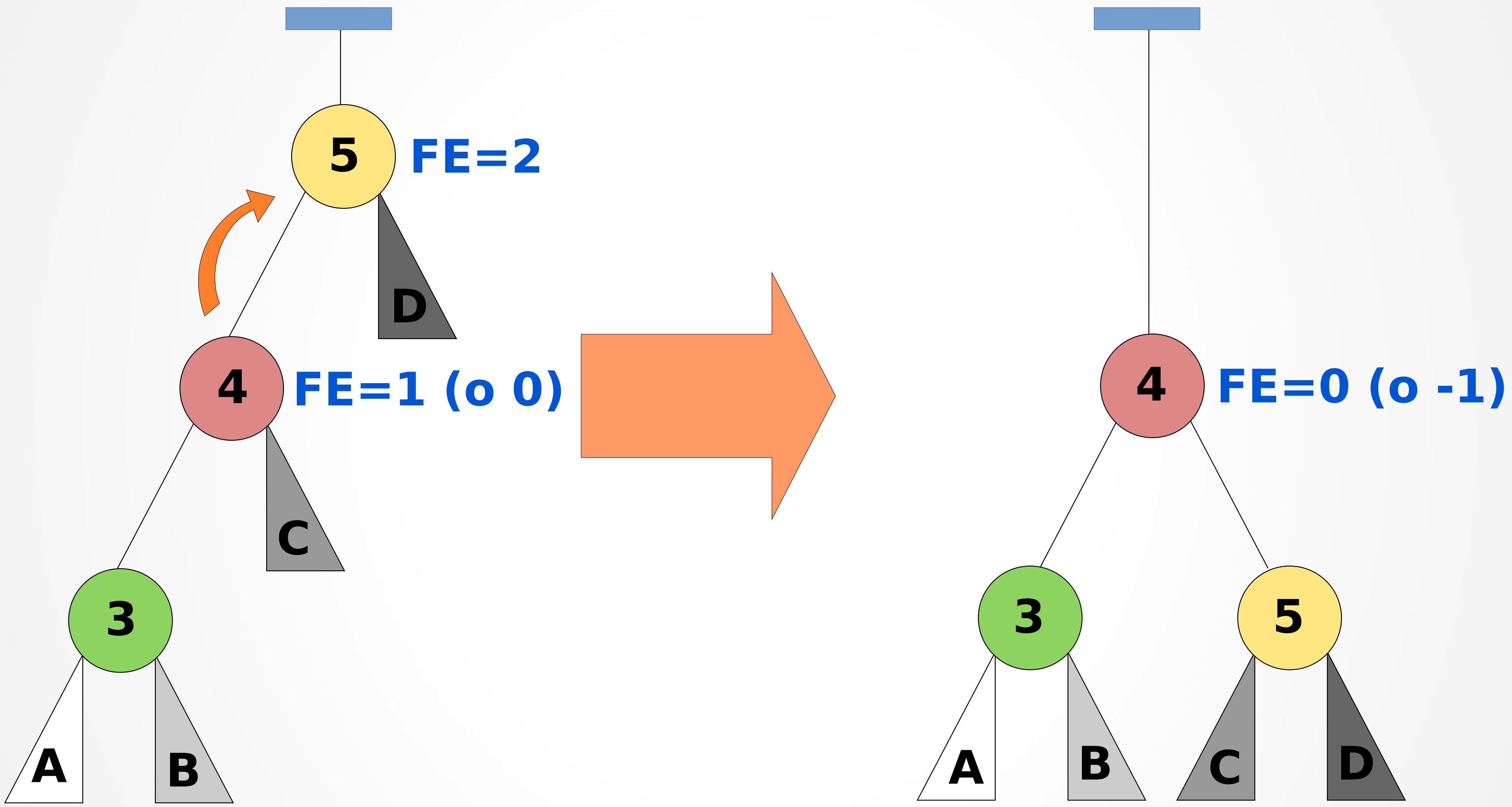
Pueden darse dos casos:

1- Rotación simple

2- Rotación doble

A su vez, ambos casos pueden ser hacia la derecha o hacia la izquierda.

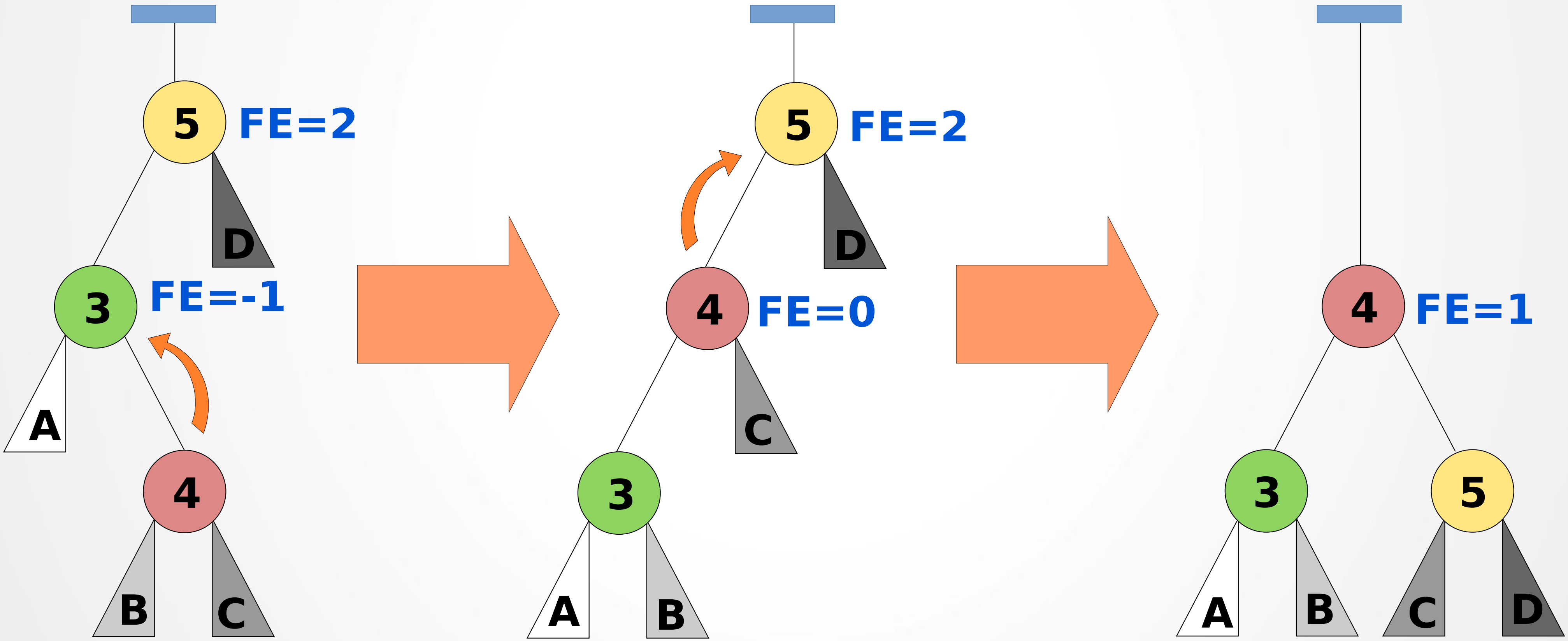
AVL



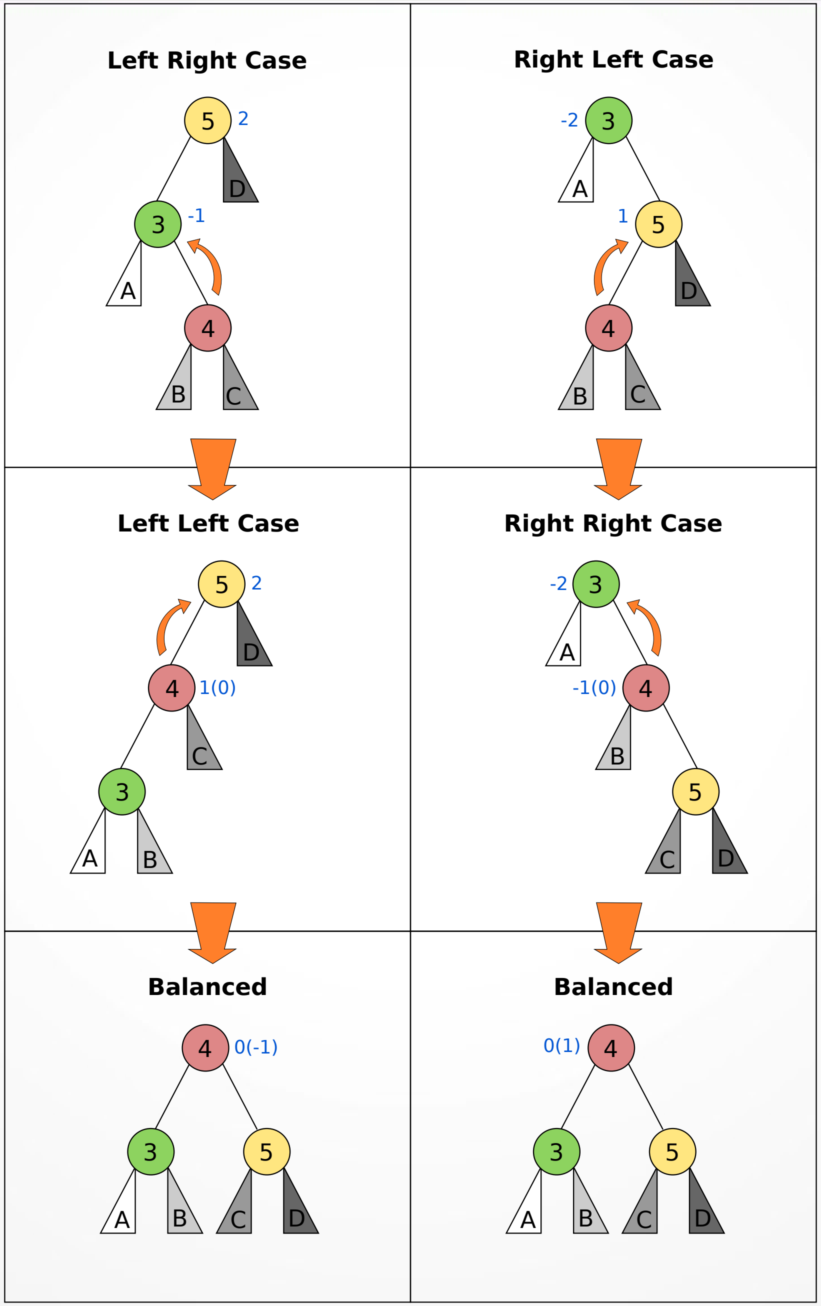
Si FE del hijo izquierdo del nodo desequilibrado es 0 o 1, una rotación simple a derecha balancea al árbol cuya raíz es 5.

AVL

Rotación a izquierda seguida de derecha:



AVL



Left Right Case
5
2
D
3
-1
A
4
B
C
Right Left Case
-2
3
A
1
5
D
4
B
C
Left Left Case
5
2
D
4
1(0)
C
3
A
B
Right Right Case
-2
3
A
-1(0)
4
B
5
C
D
Balanced
4
0(-1)
3
5
A
B
C
D
Balanced
0(1)
4
3
5
A
B
C
D

AVL.. Balanceamos

```
def _balance_tree(self, root):

  #funciones internas: balance_factor, rotate_left, rotate_right

    bf = balance_factor(root)
    if bf == 2:
         if balance_factor(root.left) == -1:
             root.left = rotate_left(root.left)
         root = rotate_right(root)
    elif bf == -2:
         if balance_factor(root.right) == 1:
             root.right = rotate_right(root.right)
         root = rotate_left(root)
    else:
         self._update_height(root)
    return root
```

Con las rotaciones pudo haber cambiado la altura de los hijos...

AVL.. Calculamos el FE

```
def balance_factor(node):

            factor = 0
            if node is not None:

                if node.left is not None:
                    factor += node.left.height

                if node.right is not None:
                    factor -= node.right.height

            return factor
```

En caso de ser node None Retornamos 0

AVL.. Rotamos a derecha

```
def rotate_right(root):

        left_tree = root.left
        root.left = left_tree.right
        self._assign_parent(root.left, root)
        left_tree.right = root
        left_tree.parent = root.parent
        root.parent = left_tree
        root = left_tree
        self._update_height(root.right)
        self._update_height(root)

        return root
```

Actualizamos las alturas

AVL.. Rotamos a izquierda

```
def rotate_left(root):

        right_tree = root.right
        root.right = right_tree.left
        self._assign_parent(root.right, root)
        right_tree.left = root
        right_tree.parent = root.parent
        root.parent = right_tree
        root = right_tree
        self._update_height(root.left)
        self._update_height(root)

        return root
```

Actualizamos las alturas

AVL.

```
def _assign_parent(self, node, parent):
        if node is not None:
            node.parent = parent



def _update_height(self, node):

        left_height = 0
        if node.left is not None:
            left_height = node.left.height
        right_height = 0
        if node.right is not None:
            right_height = node.right.height
        node.height = 1 + max(left_height, right_height)
```

AVL. Eliminar

¿Qué cambiaríamos en el borrar implementado para un ABB?

Hay que ver cuando hace falta balancear...

Índices

Tenemos una lista de alumnos ordenada alfabéticamente pero queremos acceder a los alumnos de manera rápida por DNI y por número de legajo…

¿Cómo organizamos los datos?

Se escuchan ofertas :-)

índices

índices

Colocar índices a un archivo o **indexar**, es realizar algo análogo a lo que se hace con los libros: tener una lista ordenada que sirva de referencia para acceder en forma más rápida a la información buscada:

Índices

Podemos tener:

-La lista con los datos ordenada alfabéticamente

-un AVL ordenado por DNI (en cada nodo tenemos DNI y posición del dato en la lista)

-un AVL ordenado por Número de legajo (en cada nodo tenemos Número de legajo y posición del dato en la lista)

Índices

Debemos estar atentos a los cambios en la lista de datos (altas o bajas)

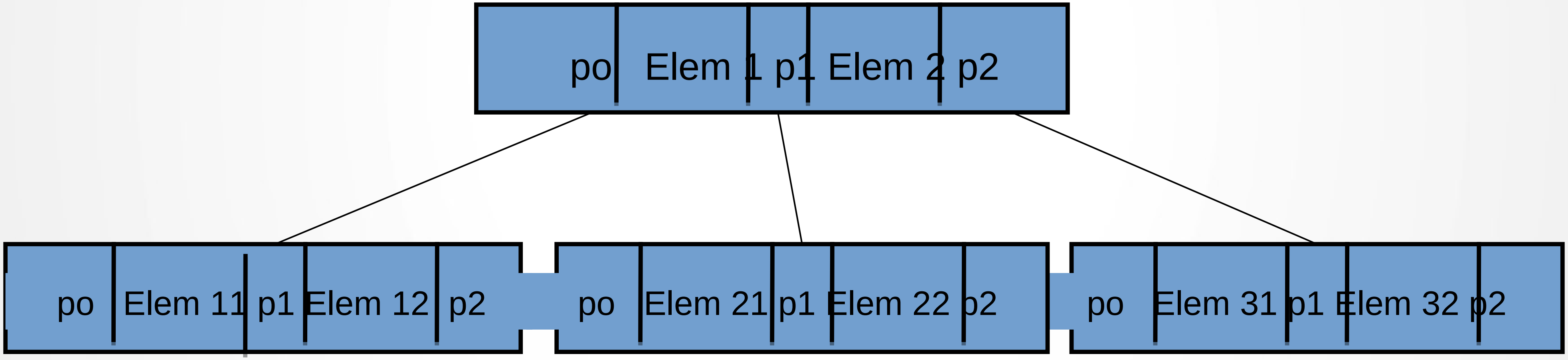
En caso agregar o borrar un dato tenemos que actualizar los índices (**reindexar**).

Árboles B, B^+ y B^*

Árboles n-arios

Árboles n-arios

La estructura del Nodo está compuesta por N Punteros y (N-1) Claves.



Árboles multicamino

Generalización de árboles binarios:

Cada nodo tiene k punteros y (k-1) claves (o registros), disminuye la profundidad del árbol,

Orden del árbol: cantidad máxima de descendientes posibles por nodo

Árboles B

Árboles B

Un árbol balanceado es un árbol multicamino, que cumple las siguientes propiedades.

Propiedades de un árbol B de orden M:

Ningún nodo tiene más de M hijos

Cada nodo (menos raíz y los terminales) tienen como mínimo

[M/2] hijos

La raíz tiene como mínimo 2 hijos (o sino ninguno)

Todos los nodos terminales están a igual nivel

Nodos no terminales con K hijos contienen K-1 registros.

Los nodos terminales tienen:

Mínimo (M/2) –1 registros

Máximo (M – 1) registros

Árboles B. Inserción

Los registros se insertan **en un nodo Terminal**

Casos posibles:

El registro tiene lugar en el nodo Terminal (no se produce overflow): solo se hacen reacomodaminetos internos en el nodo.

El registro no tiene lugar en el nodo Terminal (se produce overflow): el nodo se divide y los elementos se reparten entre los nodos, hay una promoción al nivel superior, y esta puede propagarse y generar una nueva raíz.

Performance de la inserción

Mejor caso (sin overflow) - H lecturas + 1 escritura

Peor caso (overflow hasta la raíz, aumenta en uno el nivel del árbol) H lecturas - 2H + 1 escrituras (dos por nivel más la raíz)

Árboles B. Búsqueda

Comienza desde el nodo raíz

Busca la clave en el nodo.

- Si la localiza, se encontró la clave buscada.
- Si no la localiza, se toma el puntero anterior a la primer clave mayor.
  - Si no es puntero nulo, se toma ese nodo y se repite del principio.
  - Si es puntero nulo, el elemento no se encuentra en el árbol.

Performance

Orden M, # de nodos terminales N, N+1 punteros nulos.

Accesos:

Mejor caso: 1 lectura

Pero caso: h lecturas (con h altura del árbol)

Árboles B. Eliminar

Eliminar

Nodo Terminal o Nodo no Terminal (llevar a un nodo Terminal)

Dos soluciones

Redistribuir: Cuando un nodo tiene underflow puede trasladarse claves de un nodo adyacente hermano (en caso que este tenga suficientes elementos)

Concatenar: Si un nodo adyacente hermano está al mínimo (no le sobra ningún elemento) no se puede redistribuir, se concatena con un nodo adyacente disminuyendo el # de nodos (y en algunos casos la altura del árbol)

Árboles B. Eliminar

Mejor caso: borra un elemento del nodo y no produce underflow, solo reacomodos (# elementos >= [M/2]-1)

Peor caso: se produce underflow, #elementos < [M/2] – 1

Performance de la eliminación

Mejor caso (borra de un nodo Terminal) - H lecturas - 1 escritura

Peor caso (concatenación lleva a decrementar el nivel del árbol en 1) –

2h – 1 lecturas o H + 1 escrituras

Árboles B*

Árboles B*

La redistribución podría posponer la creación de nuevos nodos. Podemos generar árboles B* más eficientes en términos de utilización de espacio.

El Árbol B* es un Árbol B especial en que cada nodo está lleno por lo menos en 2/3 partes

Propiedades (orden M):

Cada nodo tiene máximo M descendientes

Cada nodo, menos la raíz y las hojas, tienen al menos [(2M – 1) / 3] descendientes

La raíz tiene al menos dos descendientes (o ninguno)

Todas las hojas aparecen en igual nivel

Un nodo que no sea hoja si tiene K descendientes contiene K-1 claves

Una hoja contiene por lo menos [(2M – 1) / 3] –1 claves, y no más de M-1.

Árboles B*. Operaciones

Búsqueda

Igual que el árbol B

Inserción

Tres casos posibles:

-Derecha: redistribuir con nodo adyacente hermano de la derecha (o izq. Si es el último)

-Izquierda o derecha : si el nodo de la derecha está lleno y no se puede redistribuir, se busca el de la izquierda.

-Izquierda y derecha: busca llenar los tres nodos, estos tendrán un ¾ partes llena.

Borrado:

similar (concatenación, redistribución)

Árboles B+

Árboles B+

Consiste en un conjunto de grupos de registros ordenados por clave en forma secuencial, junto con un conjunto de **índices**, que proporciona acceso rápido a los registros.

Propiedades

Cada nodo tiene máximo M descendientes

Cada nodo, menos la raíz y las hojas, tienen entre [M/2] y M hijos

La raíz tiene al menos dos descendientes (o ninguno)

Todas las hojas aparecen en igual nivel

Un nodo que no sea hoja si tiene K descendientes contiene K-1 claves

Los nodos terminales representan un conjunto de datos y son enlazados juntos.

Los nodos no terminales **no tienen datos sino punteros a los datos**.

Árboles B+