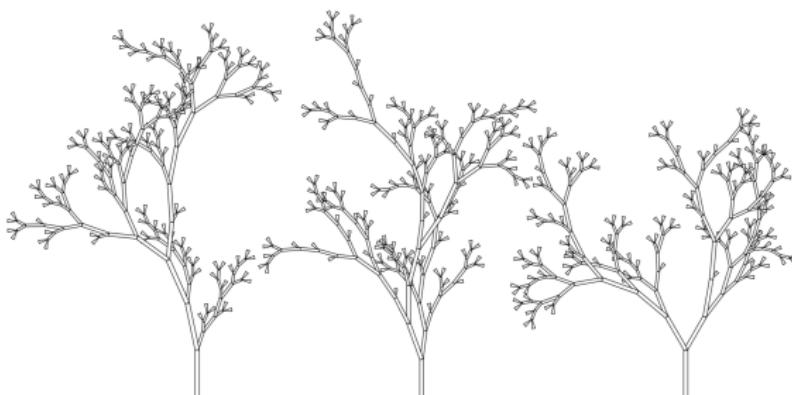


# Algoritmos y Estructuras de Datos

## Tema 6: Arboles y Arboles Binarios de Búsqueda

Grado Imat. Escuela ICAI

March 2024

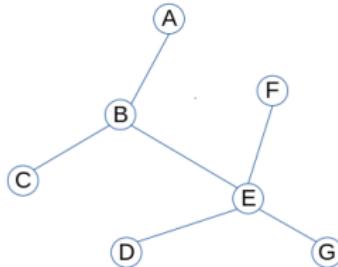


# Part I

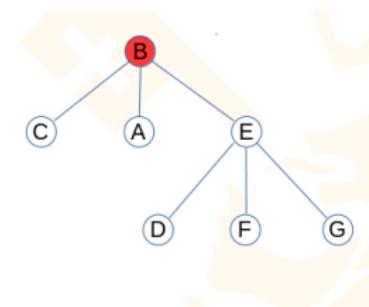
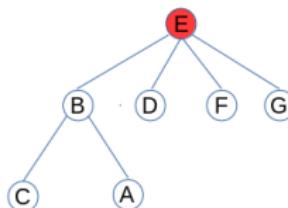
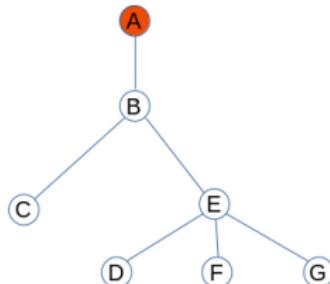
## Trees

# Árboles: Grafos acíclicos, no direccionalados

**"En general un árbol es grafo (Colección de vértices y nodos) en los que dos vértices cualquiera están conectados por un único camino.(a.k.a. No ciclos)"**



En los "Árboles enraizados" (rooted Trees) un nodo actúa como elemento de raíz. En principio la elección es arbitraria. Nodos A, E, o B.



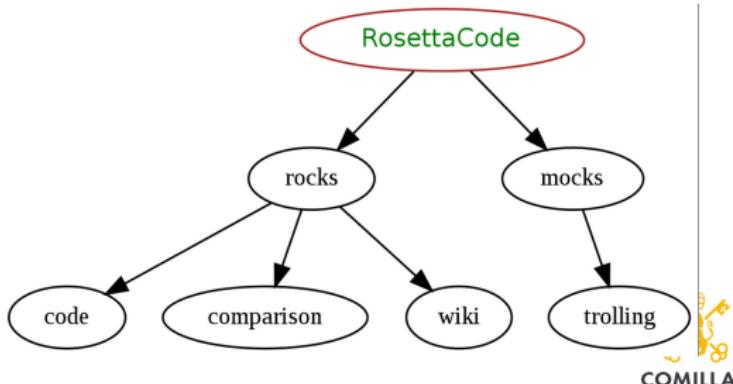
# Rooted Trees

En los Árboles con raíz, un vértice es nodo, existe una dirección contraria a la distancia al vértice, y los nodos están organizados en niveles por la profundidad de los mismos<sup>1</sup>. Esto es:

- Un nodo (vértice) es raíz
- Existe un único camino entre raíz y cualquier nodo
- Todo nodo tiene un parente (salvo el raíz)

## Terminología:

- padres, hijos, abuelos, tíos, ...
- Raíz, terminales (hojas)
- Altura, nivel
- Subárbol, camino
- Sucesor, antecesor



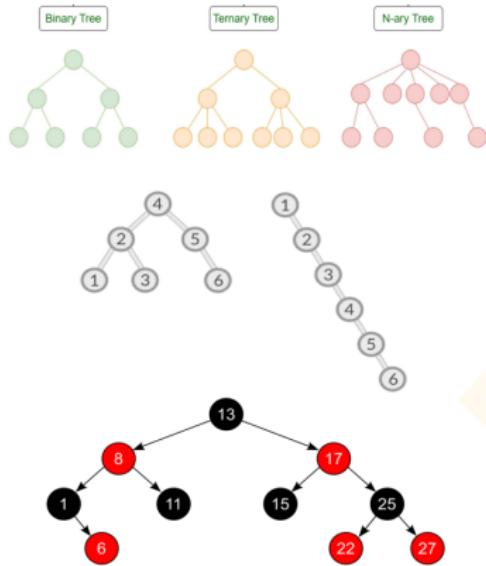
<sup>1</sup>Número de saltos desde el vértice

# Árboles: Tipos y Propiedades

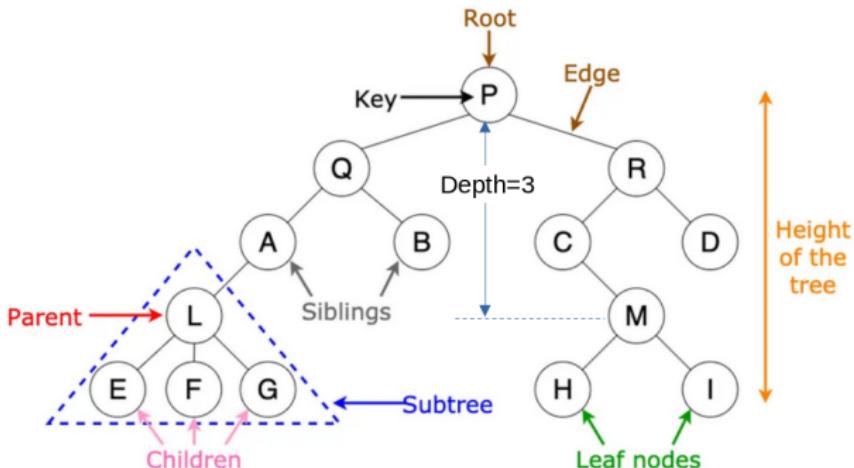
## Tipos

- Binarios, Ternarios, n-arios
  - en base al número de hijos por nodo
- Árboles Binarios de Búsqueda (ABB) o Binary Search Trees (BST)
  - propiedades de las claves que favorecen la búsqueda
- Balanceados, o Desbalanceados
  - en base a la diferencia de profundidad de las ramas
- Red-Black Tree. (Arbol Rojo-Negro ARN)
  - Coloreando los nodos para mejorar las propiedades de autobalanceo
- Hay mucho otros tipos !!

ver //www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/



# Trees: Propiedades



- ① Existe exactamente un único camino conectado 2 nodos en un árbol
- ② 2 nodos tienen al menos un ancestro común
- ③ Un árbol con N nodos tiene  $N-1$  arcos
- ④ Un árbol binario con N nodos internos tiene  $N+1$  terminales
- ⑤ La altura de un árbol binario completo con N nodos internos es aproximadamente  $\log_2(N)$

# Part II

## Árboles Binarios



Figure: (c) Stanford U



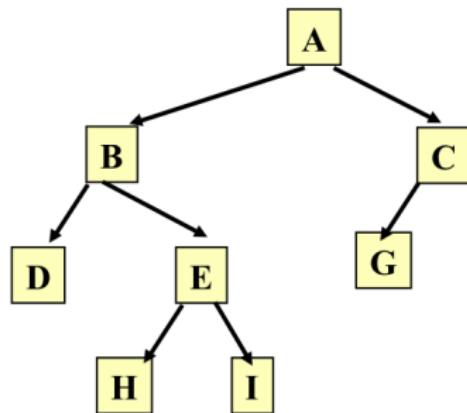
COMILLAS  
ICAI

# Arboles Binarios

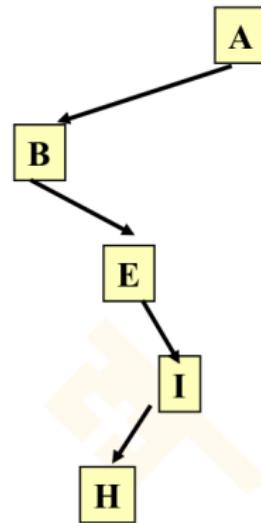
- Un árbol binario es un árbol en el que ningún nodo puede tener más de dos subárboles.
- Se conoce el nodo de la izquierda como HIJO IZQUIERDO y el nodo de la derecha como HIJO DERECHO.
- Un árbol binario es una estructura recursiva. Se divide en tres subconjuntos disjuntos:
  - ① R Nodo raíz
  - ② I<sub>1</sub>, I<sub>2</sub>, ..., I<sub>n</sub> Subárbol izquierdo de R
  - ③ D<sub>1</sub>, D<sub>2</sub>, ..., D<sub>n</sub> Subárbol derecho de R
- En cualquier nivel n, un árbol binario puede contener de 1 a 2<sup>n</sup> nodos. El número de nodos por nivel contribuye a la densidad del árbol.
- Un ÁRBOL DEGENERADO es aquél en el que existe un solo nodo hoja y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.



# Arboles Binarios



Árbol binario de profundidad 4



Árbol binario degenerado de profundidad 5



- El **FACTOR DE EQUILIBRIO** ó **BALANCE** (**Skew** en inglés) de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si la altura del subárbol izquierdo es  $h_i$  y la altura del subárbol derecho es  $h_d$ , entonces el factor de equilibrio del árbol se determina por:

$$hd - hi$$

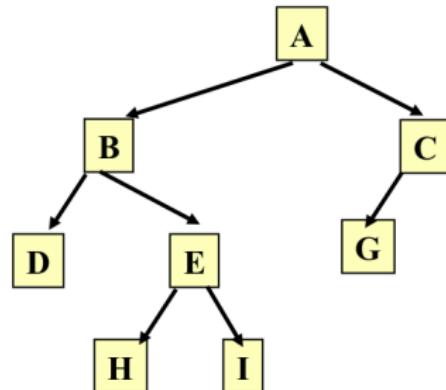
- Un árbol está **PERFECTAMENTE EQUILIBRADO** si su factor de equilibrio o balance es cero y sus subárboles están también perfectamente equilibrados.
- Un árbol está **EQUILIBRADO** si la altura de sus subárboles difiere en no más de uno (su factor de equilibrio es -1, 0 ó 1) y sus subárboles son también equilibrados.



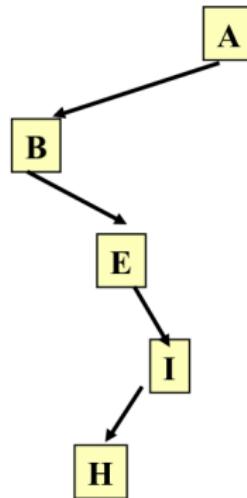
# Árbol: Equilibrio

¿Cuál es el factor de equilibrio de estos árboles?

¿Están equilibrados o perfectamente equilibrados?



Árbol 1



Árbol 2

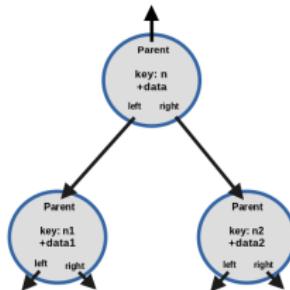


Árbol 3



# Implementación de Nodo y árbol

- Los árboles son estructuras de nodos, conectadas por "pointers" que pueden ser None
- Los Nodos tiene una Key, o clave que los identifica, y por los que "solemos" compararlos
- Los punteros, left y right no son intercambiables, y pueden ser None o apuntar a nodos hijos
- El puntero "parent" apunta al Node del que es hijo (left o right). Esto es:  $\text{childNode} \in [\text{childNode.parent.left}, \text{childNode.parent.right}]$



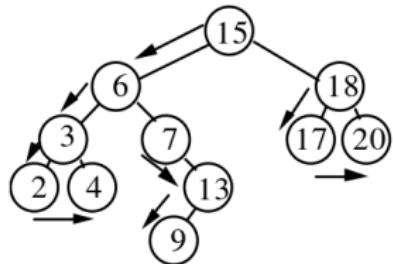
```
class Node():
    def __init__(self, key, data=None):
        # pointer to parent
        self.parent = None
        # pointer to left child
        self.left = None
        # pointer to right child
        self.right = None
        # key that identifies the node
        self.key = id
        # additional data
        self.data = data # additional data

class BTTree():
    def __init__(self, rootNode: Node):
        # a tree must have a root node !
        self.rootNode = rootNode
```

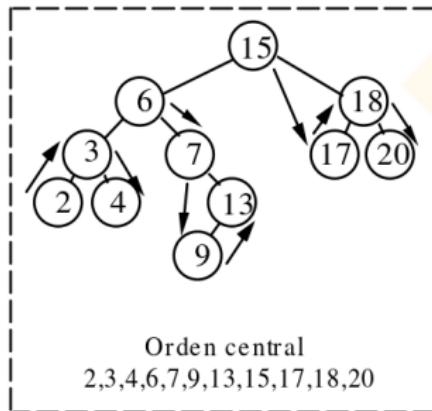


# Árbol: Ofrece una secuencia de los nodos del árbol

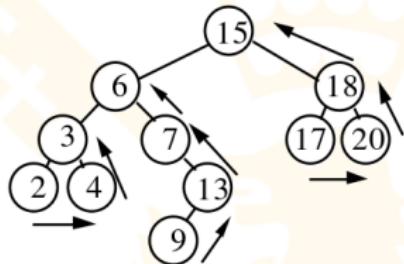
- **Preorden:** visitar 1º Padre, 2º Izq. y 3º Dcha. Se implementa sin recursividad con una pila
- **Orden central (In order):** visitar 1º Izq., 2º Padre y 3º Dcha. Es el más típico (usar pila para eliminar recursividad)
- **Postorden:** visitar 1º Izq. 2º Dcha. y 3º Padre Usar pila para evitar recursividad
- **Por Niveles (Level-Order):** arriba-abajo e Izq. a Dcha. Usar una cola



Pre-orden  
15,6,3,2,4,7,13,9,18,17,20



Orden central  
2,3,4,6,7,9,13,15,17,18,20



Post-orden  
2,4,3,9,13,7,6,17,20,18,15

# Tree Traversing: Algorithms

acorde con un **orden** determinado, los árboles ofrecen una secuencia equivalente, de pares (key, value) iterables.

## Traverse Central orden (recursive)

```
1: function CENTRALORDERSEQ(bst)
2:   seq ← empty list
3:   if bst.rootNode is not null then
4:     leftSubTree ← LEFTSUBTREE(bst)
5:     if leftsubTree is not null then
6:       seq append CENTRALORDERSEQ(leftSubTree)
7:       seq.append bst.rootNode as (key,value) pair
8:     rightSubTree ← RIGHTSUBTREE(bst)
9:     if rigtSubTree is not null then
10:      seq append CENTRALORDERSEQ(rightSubTree)
11:   return seq
```

**Ejercicio:** escribe el pseudocódigo para los otros órdenes



COMILLAS  
ICAI

# Part III

## Binary Search Trees

Tree as a data structure for searching

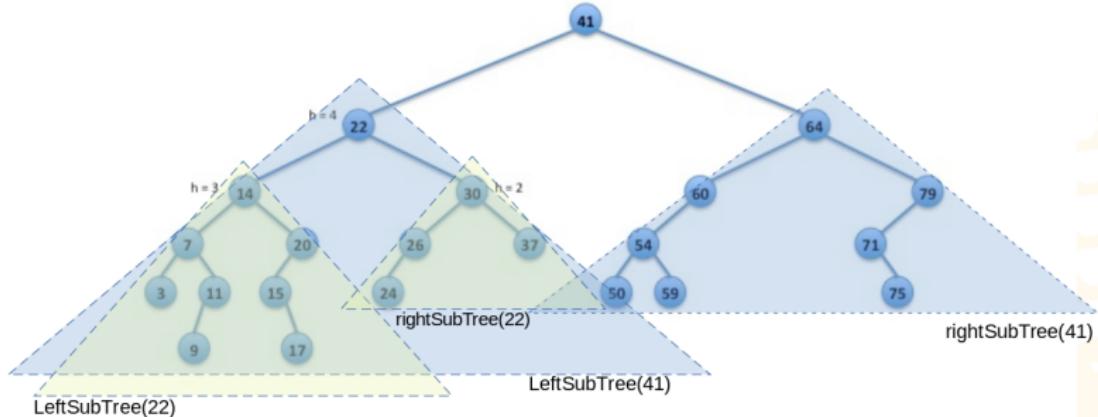


COMILLAS  
ICAI

# ABB Árbol Binario de Búsqueda // BST

Un árbol binario de búsqueda (ABB/BST) es un arbol enraizado binario, formado por nodos **comparables**, en el que para cualquier nodo del árbol éste es mayor o igual que todos los nodos de su subárbol-izquierdo, y menor o igual que cualquier nodo de su sub-árbol derecho. Esto es:

$$\forall n, n_i, n_r \in Tree \left\{ \begin{array}{ll} n_i \in leftSubTree(n) & \rightarrow n \geq n_i \\ n_r \in rightSubTree(n) & \rightarrow n \leq n_r \end{array} \right\}$$



Veremos que la complejidad de las operaciones sobre los datos

BST son la base de la estructura de datos "Colección Ordenada", manteniendo un conjunto de datos  $v_1, v_2, \dots$  acorde con un conjunto de claves  $k_1, k_2, \dots$

## Complejidad:

- Todas las operaciones son  $O(h)$ , siendo  $h$  la altura del árbol ( insert, delete, find, etc... )
- Si el arbol se mantiene balanceado,  $h \sim \log_2(\# \text{ items})$
- Las operaciones más sencillas se basan en recorrer los nodos del árbol utilizando alguna de las formas estándar (pre-orden, orden central, post-orden)
- La secuencia ordenada de items se corresponde con el orden de recorrido central
- El coste de "build" de una colección de  $n$  elementos es  $n \times \log n$



# Operaciones sobre BST

- **Operaciones de Interrogación** (búsqueda o secuencia)

- BST.find(key) → (key,value)
- BST.maxKey() → (key,value)
- BST.minKey() → (key,value)
- BST.sucesor(key) → (key,value)
- BST.previous(key) → (key,value)

- **Operaciones que modifican el conjunto dinámico**

- BST.insert(k,value)
- BST.remove(k)

- **Constructores**

- BST() → árbol vacío
- BST(keys,values) → BST con la secuencia (key, value)



COMILLAS  
ICAI

# BST: Finding a key

Hasta un máximo de  $h$  decisiones en qué sub-árbol (left/right) seguir buscando. Utilizando la estructura de los árboles BST  
Retorna la pareja key,value, o null si no lo encuentra.

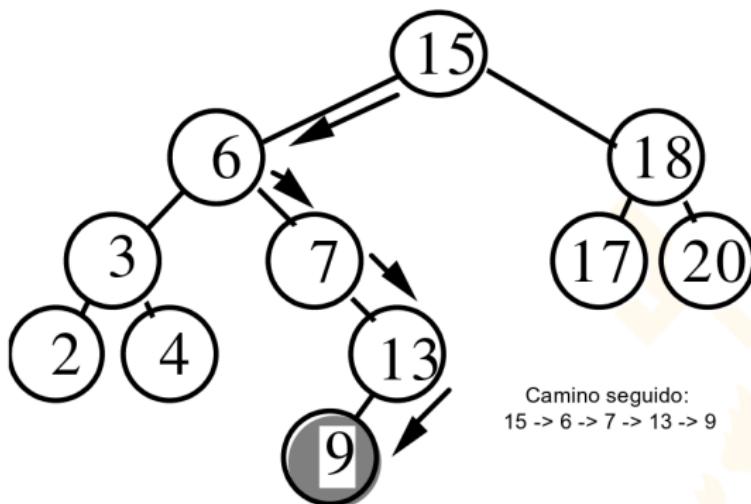


Figure: Búsqueda de key=9

# Operaciones BST: find

Find algorithm itera al estilo bisección, o de forma recursiva, por las ramas correctas del BST hasta encontrar la clave

Complejidad es **O(h)**, la altura del árbol.

## versión Iterativa

```
function FIND(bst, key)
    node ← bst.rootNode
    while node not null &
        key != node.key do
            if key < node.key then
                node ← node.left
            else if key > node.key then
                node ← node.right
            if node not Null then
                return node.key, node.value
            else
                return null      ▷ Not found
```

## versión Recursiva

```
function FINDREC(bst, key)
    node ← bst.rootNode
    if node not null then
        if key < node.key then
            ▷ Recurre sobre arbol Izq
            return FINDREC(BST(node.left),key)
        else if key > node.key then
            ▷ Recurre sobre arbol derecha
            return FINDREC(BST(node.right, key))
        else
            return key, node.value ▷ found, return key,
            value
    else
        return null
```

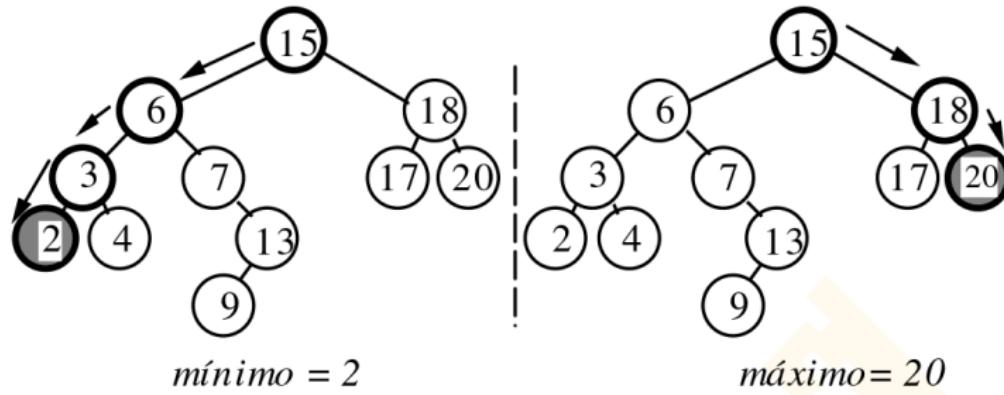


# BST: Buscando máximo y mínimo

La estructura de los BST/ABB permite encontrar el min y máx en  $O(h)$ .

Min: moverse todo lo posible a la izquierda

Max: moverse todo lo posible a la derecha



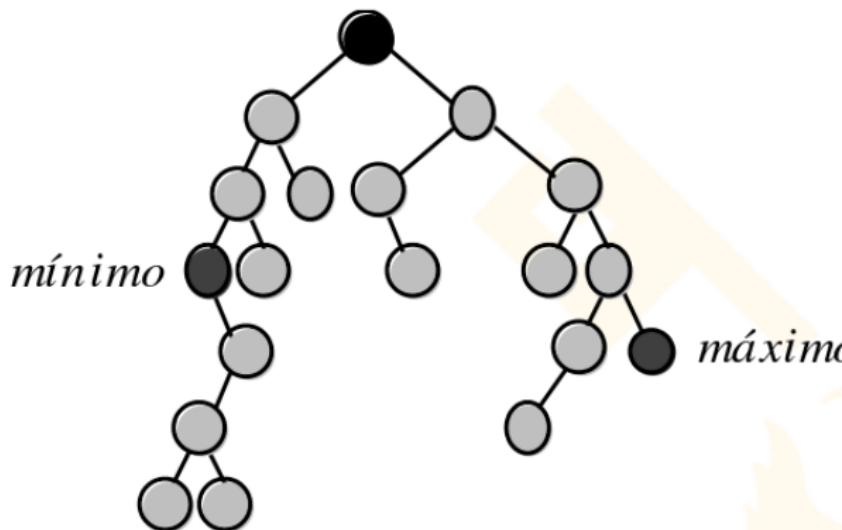
Pero no es siempre tan sencillo....



COMILLAS  
ICAI

Dada la estructura del  
árbol:

- El nodo con clave mínima será el nodo más a la izquierda que no tenga hijo izquierdo
- El nodo con clave máxima será el nodo más a la derecha que no tenga hijo derecho



# Operaciones BST: Max y Min

**Max** selecciona left Child de forma recursiva hasta que el nodo carece de hijo izquierdo,

**Min** selecciona right Child de forma recursiva hasta que el nodo carece de hijo derecho.

⇒ Operación es  $O(h) \sim \log(n)$

---

## Mínimo

```
function MIN(bst)
    node ← bst.rootNode
    if node not null then
        while node.left not null do
            node ← node.left
    else
        return null
    return node.key, node.value
```

---

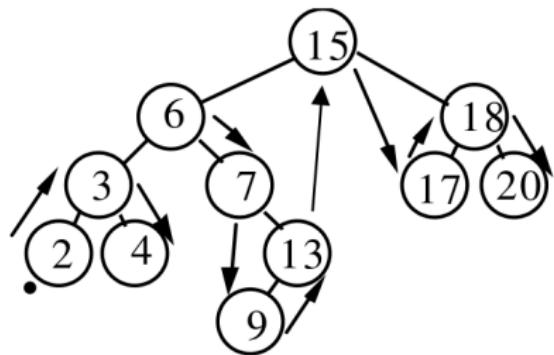
## Máximo

```
function MAX(bst)
    node ← bst.rootNode
    if node not null then
        while node.right not null do
            node ← node.right
    else
        return null
    return node.key, node.value
```



# Sucesor o previo (next key, prev key)

Acorde a un criterio de secuenciación (Central, normalmente)<sup>2</sup>



Secuencia Acorde a Orden Central  
2,3,4,6,7,9,13,15,17,18,20

Ejemplos:

- El sucesor del nodo con clave 6 es el 7, del 7 el 9, del 13 el 15 ...
- El predecesor del 7 es el 6, del 9 el 7, del 15 el 13 ...

<sup>2</sup>podríamos utilizar otro !)



# Sucesor/Previo How To

- **Fuerza bruta:**

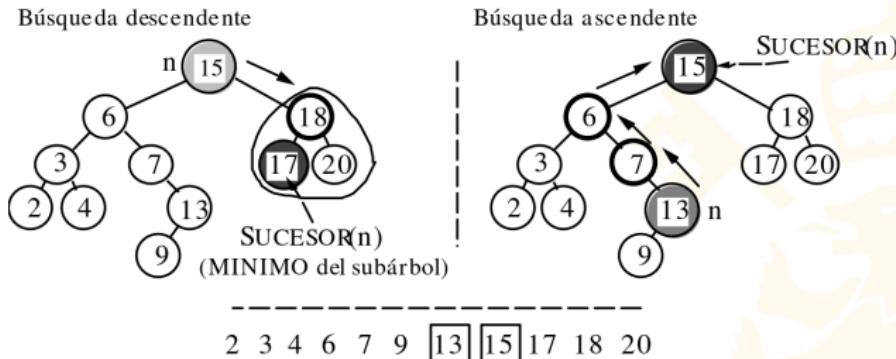
Aplicar la secuenciación completa ( $O(n)$ ) y coger el siguiente (sucedor) o el anterior (predecesor) en la lista resultante

- **Forma óptima:**

Recorrer sólo la parte del árbol que sea estrictamente necesaria y encontrar el nodo buscado indicado ( $O(\log n)$ )

## Sucesor:

- Si el nodo tiene hijo derecho, será el mínimo del subárbol derecho.  
Si no, escalar hacia arriba hasta acceder al padre como hijo izquierdo



# Sucesor: algoritmo

## Sucesor

```
1: function SUCESOR(bst, key)
2:   node ← FINDNODE(bst, key)           ▷ First, find the node with the given key first
3:   if node not null then
4:     if node.right not null then    ▷ Min or first, of right subtree, color = BLACK present
5:       return MIN(BST(node.right))
6:     else
7:       parentNode ← node.parent      ▷ Otherwise jump upwards till root or right turn
8:       while parentNode not null & node = parentNode.right do
9:         node ← parentNode
10:        parentNode ← node.parent
11:        if parentNode not null then          ▷ If not at root, return node content
12:          return parentNode.key, parentNode.value
13:   return null                         ▷ Any other case, return null
```

## Predecesor:

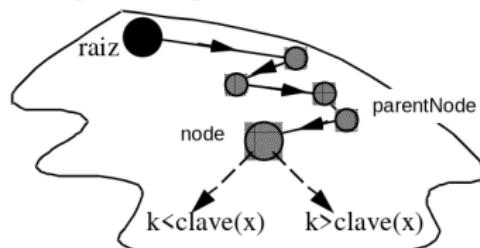
- Si el nodo tiene hijo izquierdo , será el máximo (último) del subarbol izquierdo. Si no, escalar hacia arriba hasta acceder al padre como hijo derecho.

# Inserción de una nueva pareja (key,value)

Se quiere insertar en el árbol, un nuevo nodo con la pareja (key, value)  
El procedimiento consta de dos partes claras:

- ① búsqueda del punto de inserción  $\Rightarrow \mathbf{O}(h) \sim \log(n)$
- ② inserción del nuevo nodo con (key,value)  $\Rightarrow \mathbf{O}(1)$

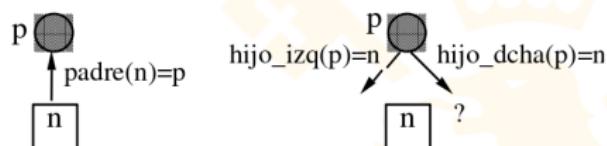
Búsqueda del punto de inserción



Descenso hasta llegar a un nodo

Se van actualizando node v parentNode

Inserción del nodo



Se actualizan dos punteros

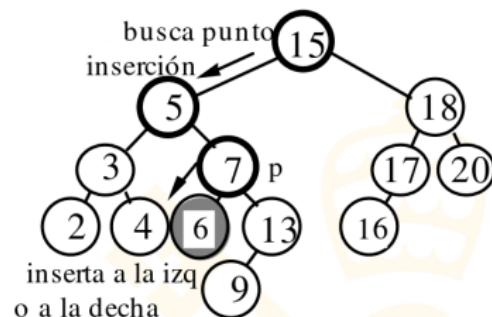
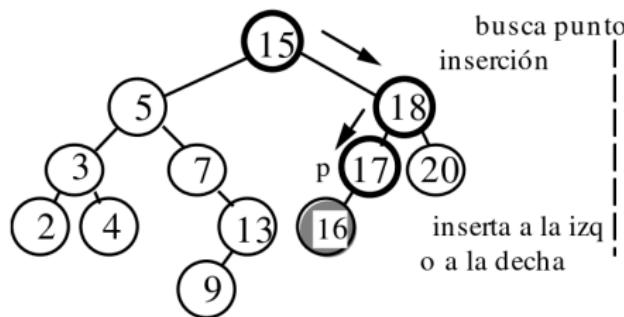
En un Arbol Binario de Búsqueda, la inserción ya es realizada en la ubicación correcta. Sólo es necesario mantener el "Balance" del árbol para asegurar que la altura del mismo ( $h$ ) se mantenga  $\sim \mathbf{O}(\log(n))$



# Inserción: Ejemplo

Se desciende por el árbol hasta encontrar la posición en la que debe ir (una que no esté ya ocupada por un nodo)

- Insertando Node( $k=16$ ) en el árbol. Punto de inserción en 17 con `leftChild=None`
- Insertando Node( $k=6$ ) en el árbol. Punto de insercion en Node( $k=7$ )



# BST: Insert nodo

## InsertNode

```
1: function INSERT(bst,(key,value))
2:   newNode ← NODE(key,value)
3:   node ← bst.rootNode
4:   if node is Null then
5:     Set bst.rootNode ← newNode
6:   else
7:     ▷ desciende por el árbol con node buscando punto inserción
8:     while node not Null do
9:       parentNode ← node
10:      if key < clave(x) then
11:        node ← node.left
12:      else
13:        node ← node.right
14:      ▷ insert node with key,value
15:      Set parentNode as parent of newNode
16:      if key < parentNode.key then
17:        Set newNode as parent.left
18:      else
19:        Set newNode as parent.right
20:   Return
```



# BST: Eliminación de un nodo

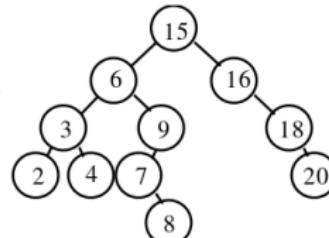
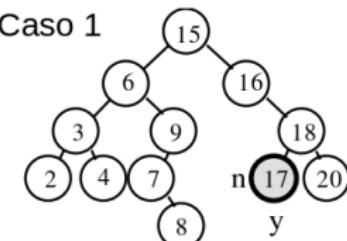
- Es un proceso más complicado que la inserción. Se quiere eliminar el nodo n.
  - NO se quiere eliminar aquél nodo que tenga una clave dada, esto requiere primero buscar...
- Existen tres casos diferentes:
  1. El nodo n no tiene hijos
  2. El nodo n sólo tiene un hijo
  3. El nodo n tiene dos hijos
- Casos ordenados de más sencillo a más complejo:
  1. se resuelve simplemente borrando el nodo n
  2. necesita actualizar punteros, (similar al borrado en listas)
  3. necesita actualizar punteros y copiar...



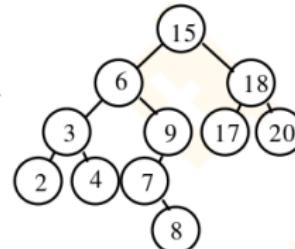
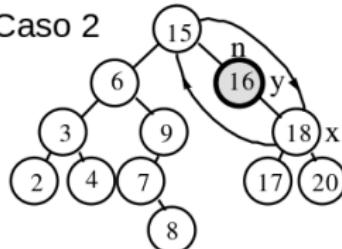
COMILLAS  
ICAI

# BST: Eliminación de un nodo II

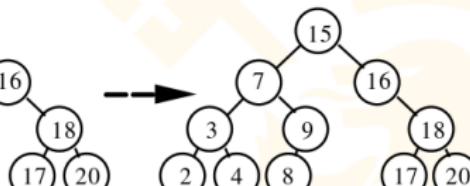
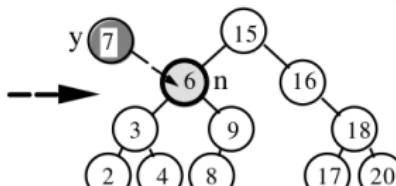
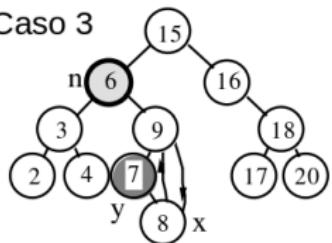
Caso 1



Caso 2



Caso 3



MILLAS  
ICAI

# BST: Borrar Nodo con clave

## Delete Node with Key in tree

```
function DELETE(bst,key)
    node ← FIND(bst,key)
    if node is Null then
        return
    ▷ case 1, leaf node
    if node is Leaf then
        if node is leftChild then
            Set node.parent.left to Null
        else
            Set node.parent.right to Null
        Set node.parent to Null
        return
    ▷ case 2, just one child
    if node.left is null or node.right is Null then
        if node.left not Null then
            child ← node.left
        else
            child ← node.right
        child.parent ← node.parent ▷ connect child to
        parent of to-delete node
        parent ← node.parent
        if node is right Child then
            Set child as right child of parent
        else
            Set child as left child of parent
        Return
    .../..
```

```
function DELETE(bst,key) (continúa)
    ▷ Case 3 two children. Copia el sucesor sobre el nodo
    a borrar, y elimina el sucesor
    sucNode ← SUCESOR(bst,node)
    tmpNode ← NODECLONE(sucNode)
    DELETE(bst,sucNode)
    SWAPCONTENTS(node, tmpNode)
    return
```

```
function NODECLONE(aNode)
    ▷ Crea un nuevo nodo con la misma clave y datos
    return NODE(aNode.key, aNode.data)
```

```
function SWAPCONTENTS(node1, node2)
    ▷ Intercambia keys y data, dejando pointers intactos
    node1.key ↔ with node2.key
    node1.data ↔ node2.data
    return
```

# BSTs,... para qué ??

## • Ventajas de los Árboles binarios de búsqueda

- ① Búsqueda eficiente. Soportando búsqueda binaria de forma nativa
- ② Secuenciación flexible, que permite utilizar el in-orden, pre- y post-orden de forma sencilla
- ③ Eficiente en memoria. Sólo dos punteros por nodo
- ④ Inserción y borrado rápidos, comparados con otras estructuras de datos
- ⑤ fácil de implementar
- ⑥ soporte de operaciones de ordenación, como heapSort

## • Desventajas

- ① Sólo dos hijos por nodo, que puede no ser de aplicación en casos desbalanceo, que limita su rapidez
- ② Caso peor muy desfavorable ( ej. inserción de claves ordenadas)
- ③ algoritmos de balanceo complejos

## • Usos de los árboles Binarios

- ① Algoritmos de búsqueda y ordenación
- ② Sistemas de Bases de Datos
- ③ Gestión de sistemas de ficheros
- ④ Algoritmos de ruteado, Excel y hojas de cálculo.



COMILLAS  
ICAI

# BST: qué puede ir mal ??

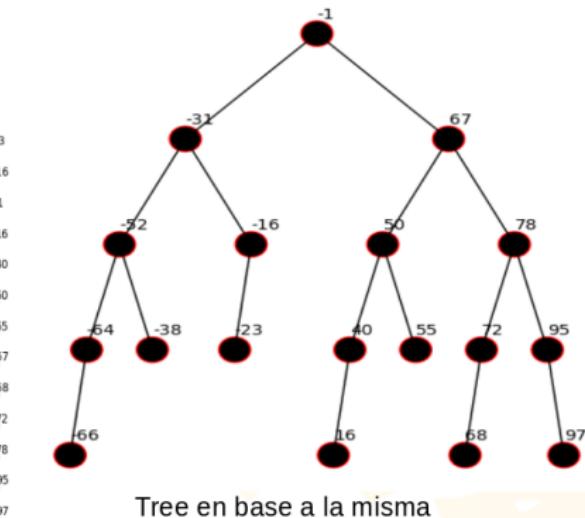
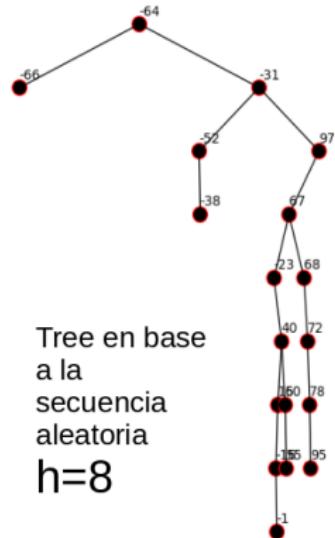
## Performance:

- Hemos visto que las operaciones "complicadas" de inserción y borrado, así como las búsquedas con de  $O(h)$  donde  $h$  es la altura del árbol.
- Y, alegremente, hemos supuesto que  $h \sim \log(n)$  donde  $n$  es el número de nodos (o claves) en el árbol.
- Por tanto depende del equilibrado del árbol, esto es de la diferencia entre profundidad entre los caminos alternativos desde la raíz a los nodos hoja.

## Es esto siempre cierto ??

- La profundidad del árbol depende de la secuencia de inserción de las claves en el nodo.
  - Se puede probar que con claves aleatorias, el árbol resultante es "razonablemente equilibrado"
  - Pero en el caso peor, de claves ordenadas, el árbol resultante es equivalente a una lista enlazada  $\Rightarrow h = n$  y la performance de las operaciones es  $O(N)$

# BST Organización, profundidad y Complejidad



Secuencia: [-64, -31, 97, 67, 68, 72, -23, 40, -52, 78, 16, -16, 50, -1, 55, -38, -66, 95]

# BST: mejorando el equilibrio

La solución es **Mejorar el equilibrio de los árboles de búsqueda**

Dos opciones (entre muchas) en el mercado:

**AVL Trees** Arboles "augmented" con la información de equilibrio de cada subárbol asociado a cada nodo, y que, mediante rotaciones, aseguran en todo momento el equilibrio del árbol

**Red Black Trees** Arboles, "augmented" con un bit de color, que cumplen ciertas condiciones que aseguran un "cierto" eqilibrio.<sup>3</sup>

En base a garantizar cierto equilibrio en el árbol, se evita el Worst-case lineal y se garantiza la consistencia del tiempo de respuesta de las operaciones sobre árboles en tiempo logarítmico con el número de nodos.

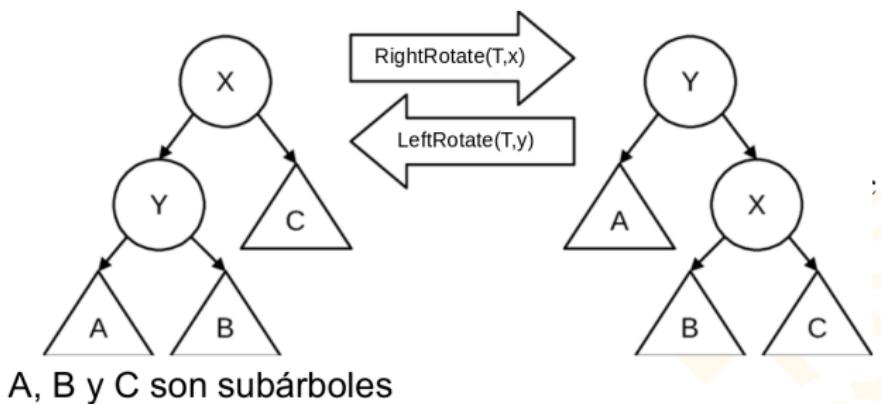


---

<sup>3</sup>Veremos a continuación los RBTrees en cierto detalle

# Equilibrando los árboles: Rotaciones

La rotación (derecha/izquierda) es una transformación que, manteniendo la secuencia de ordenación de claves natural en el BST, modifica el balanceo o equilibrio del mismo



La secuencia natural [A, y, B, x, C] es invariante ante las rotaciones

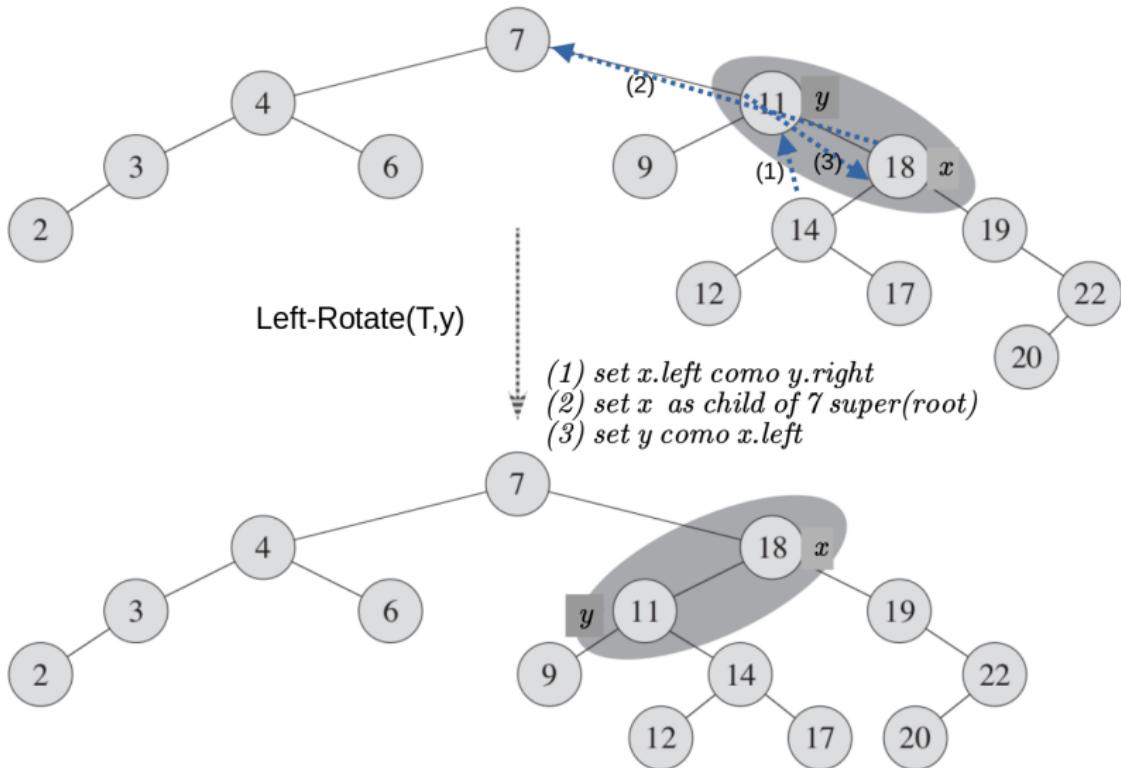
# Rotation: Processo

- en Right Rotate:
  - B, que es el predecesor de X, pasa a ser el leftChild de X
  - Y se convierte en raiz local, asignando como parent de y, el parent de X
  - X toma el lugar como rightchild de Y
- en LeftRotate,
  - B, que es el sucesor de Y, pasa a ser el rightChild de y
  - X se convierte en raiz local, asignando como parent de X, el parent de Y
  - Y toma el lugar como leftChild de X



COMILLAS  
ICAI

# Rotation: How TO



# Rotations: Algorithm

```
function LEFTROTATE(bst,y)
    x ← y.right                                ▷ sets x
    y.right ← x.left   ▷ Fija sucesor de y como su hijo derecho
    if x.left ≠ null then
        x.left.parent ← y

    x.parent ← y.parent ▷ fijar x como root local o del arbol...
    if y.parent == Null then                    ▷ y era el root !
        bst.rootNode ← y
    else if y is leftChild then               ▷ set as right or left child
        y.parent.left ← x
    else
        y.parent.right ← x

    x.left ← y                                  ▷ Fija y como leftChild de x
    y.parent ← x
    return
```

RighRotate es totalmente simétrico...

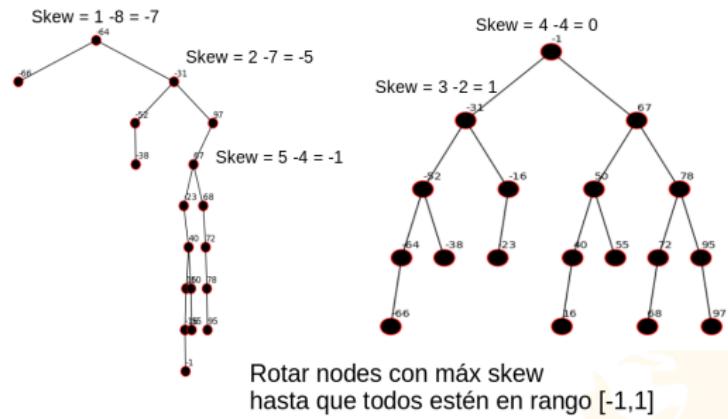


## Skew y rebalanceo de árboles

- The **skew** de un nodo es la diferencia entre las "alturas" de sus subárboles derecho e izquierdo.

$$skew = height(righSubtree) - height(leftSubtree)$$

- El nodo está equilibrado sii  $skew \in [-1, 0, 1]$ 
    - Si  $skew < -1$  entonces puede ser corregido con Right-Rotate( $T, node$ )
    - Si  $skew > +1$  entonces puede ser corregido con Left-Rotate( $T, node$ )



La secuencia apropiada de rotaciones puede transformar árboles complementariamente

## Part IV

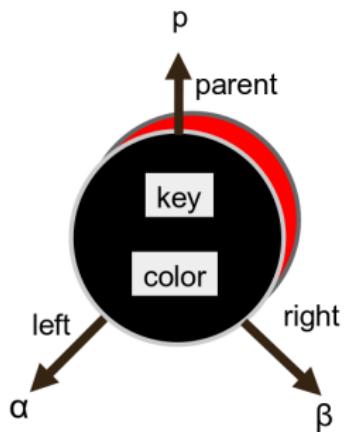
# RedBlack Trees : Árboles Rojo-Negros



COMILLAS  
ICAI

# Red-Black Trees (árboles rojo-Negros)

- Es un árbol BINARIO de búsqueda
- Son árboles (aproximadamente) EQUILIBRADOS
- El equilibrio se consigue utilizando unas restricciones en la forma de colorear los nodos
- Cada nodo tiene, además, un color asociado, Rojo o Negro



p: parent Node ( or null)  
 $\alpha$ : Left subtree ( or null)  
 $\beta$ : right subtree (or null)

# RB Trees: Condiciones

Un árbol Rojo Negro, es un Árbol Binario de Búsqueda que, además cumple 5 condiciones adicionales:

- P1 Todo nodo es rojo o negro.
- P2 Toda hoja (NULL) es siempre NEGRA.
- P3 Si un nodo es ROJO entonces sus dos hijos son NEGROS<sup>4</sup>
- P4 Todo camino sencillo desde cada nodo a una hoja de sus subárboles contiene el mismo número de nodos NEGROS.<sup>5</sup>
- P5 El nodo raíz es siempre Negro (para evitar indeterminaciones)

**Estas condiciones aseguran el equilibrio de los árboles Rojo-Negros**

---

<sup>4</sup>pero no necesariamente al revés

<sup>5</sup>lo que se denomina black height



COMILLAS  
ICAI

# RB Tree: Ejemplo y visualización

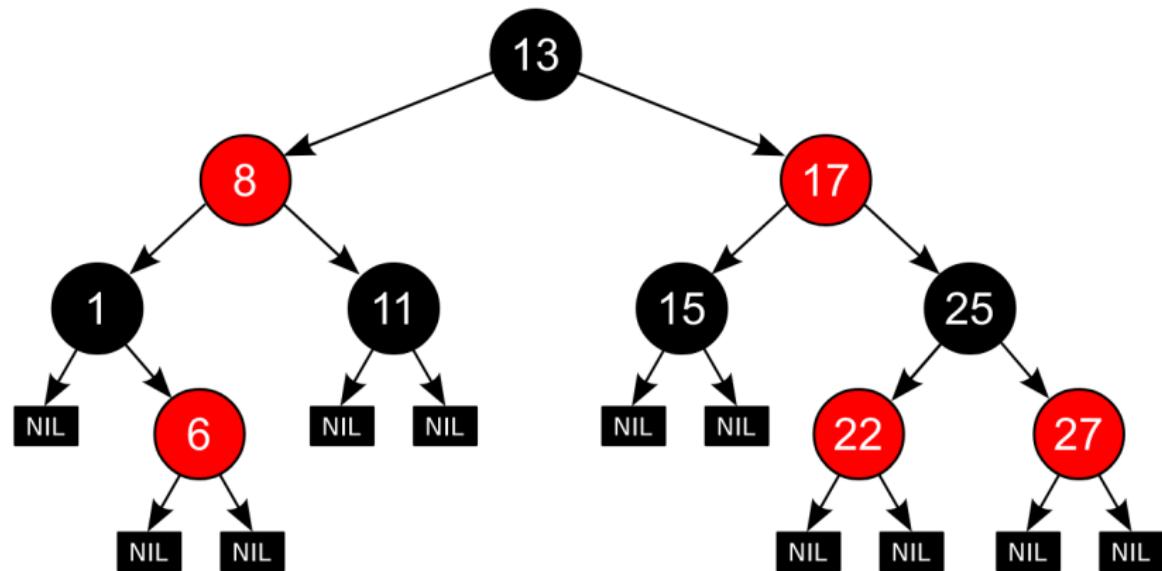


Figure: Source:Wikipedia Commons

Las hojas del árbol se consideran como un tipo especial nil/Null que no tienen clave<sup>6</sup>.



COMILLAS  
ICAI

<sup>6</sup>Normalmente no se visualizan

# RB Trees: Complejidad de operaciones

- Existe un lema que dice:

*La altura (normal) de un árbol RN con N nodos internos (sin incluir hojas) es  $\leq 2 \times \log(N + 1)$  donde N es el número de claves.*
- Esta es la razón por la que dichos árboles son buenos árboles de búsqueda
- Una consecuencia inmediata de dicho lema es que todas las operaciones de petición/consulta se pueden hacer en  $\mathbf{O}(\log(N))$  puesto que  $\mathbf{O}(h) = \mathbf{O}(\mathbf{O}(\log N)) = \mathbf{O}(\log N)$ 
  - Las operaciones que modifican (inserción y borrado) también son  $\mathbf{O}(\log N)$ , pero no se puede deducir por mera aplicación de dicho lema, (son más complicadas), ya que han de preservar los condicionantes de los ARN.
  - La operación de borrado es algo más complicada que la de inserción, por idénticas razones



COMILLAS  
ICAI

# RBTrees: Inserción

Insertar un nodo en un árbol Rojo-Negro:

- ① Insertar un nodo utilizando el algoritmo de inserción en BST
- ② Fijar el nodo nuevo como de **color Rojo**
- ③ Restaurar las 5 condiciones (P1,P2, ...,P5) de los árboles Rojo-negros (Fix-RBTree) mediante ajustes de colores y rotaciones.
  - P1, y P2 se cumplen siempre
  - al Insertar un nodo rojo, P4 se sigue cumpliendo
  - P5 siempre se cumple, salvo que el árbol estuviera vacío  
⇒ En primera instancia, será P3 la condición no cumplida, i.e. cuando nuevo nodo tiene un parent del mismo color

---

## Insertion in an RB-Tree

```
function RB-INSERT(rbt, key)
    ▷ insert as in BST, but use RBNode for newNode creation
    newNode = BST-INSERT(rbt, key)
    newNode.color = RED
    ▷ paint newNode as red
    RB-FIXUP(rbt, newNode)
    ▷ Restore RB conditions
    return newNode
```

---



# RBTREE Inserción: RBT-FIXUP

Hay tres casos, que se duplican cada uno en función de que el padre del nodo insertado sea un RightChild, o un leftChild. Consideraremos tres elementos:

- **z** → El nodo en consideración
- **z.parent** → el parent
- **z.parent.parent.left/right** → *uncle* ()según que z.parent sea un rightChild, o sea un leftChild )

Caso	Condicion	Proceso Solución (Itera mientras que z.parent.color == RED)
Caso 1	Uncle es RED	(1) pintar parent y tío de BLACK (2) pintar abuelo de ROJO (3) fijar z en el abuelo
Caso 2	Uncle es BLACK y z es rightChild	(1) fijar z en el parent (2) Left Rotate(z) (3) Saltar a Caso 3
Caso 3	Uncle es BLACK y z es rightChild	(1) Pinta parent de BLACK (2) pinta al Abuelo de RED (3) Right-Rotate(abuelo)

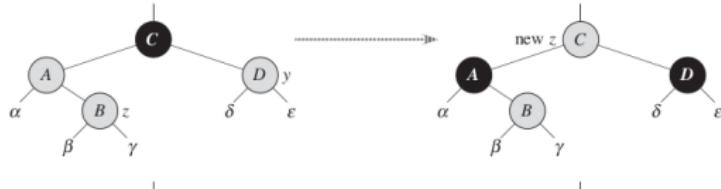


# RBTREE Insert FIXUP

Iterar mientras que el padre de z sea Rojo ( violación ). Tres Casos<sup>7</sup>

## Caso 1:

- Recolorear, padre, tío de Black
- Color abuelo en Rojo
- Fijar z en Rojo e iterar

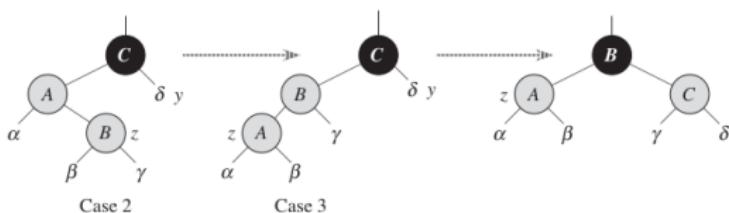


## Caso 2

- shift Z al padre y Left-Rotate(z), avanza a 3

## Caso 3:

- padre en Black, abuelo en Rojo
- Right-Rotate abuelo. Iterar



nodos gris son nodos rojos

<sup>7</sup> similar y simétrico en caso de que z sea left-Child

# RBTREE FIXUP: algorithm

## RB-Tree Inserción Fix-Up

```
function RB-INSERT-FIXUP(rbt,z)
    while z.parent is RED do
        if z.parent is leftChild then
            y = z.parent.parent.right
            if y is RED then
                z.parent.color = BLACK
                y.color = BLACK
                z.parent.parent.color = RED
                z = z.parent.parent
            else
                if z.parent is rightChild then
                    z = z.parent
                    LEFT-ROTATE(rbt,z)

                    z.parent = BLACK
                    z.parent.parent.color = RED
                    RIGHTROTATE(rbt,z.parent.parent)
                else:
                    ...
    rootNode.color = BLACK
    return
```

▷ Iterate mientras conflicto de colores  
▷ is z parent a left child? Cases 1, to 3  
    ▷ get uncle  
    ▷ Case 1, uncle is RED  
        ▷ set parent Black  
        ▷ set uncle Black  
        ▷ set grandParent Red  
    ▷ consider now at grandParent position  
    ▷ case 2 and 3, Uncle is black  
        ▷ Case 2 parent is right Child  
    ▷ place consideration point to parent  
    ▷ do left Rotate to turn to case 3  
        ▷ Case 3: parent is left Child  
            ▷ set parent black  
            ▷ set grandparent Red  
            ▷ RR sobre abuelo  
    ▷ symmetric, cases 4 to 6, when z.parent is right child



# Insercion en RBTree: Complejidad

- Análisis intuitivo:
  - La altura de un árbol RN es  $O(\lg N)$
  - Una llamada a Insertar\_Arbol\_Binario ( $T, x$ ), necesita  $O(\log N)$
  - El while externo puede ejecutarse varias veces si aparece el CASO 1.
  - En la Peor de las situaciones subir todo el árbol el  $n^{\text{o}}$  de veces que se ejecuta el while será de  $O(\log N)$   
⇒ Luego será  $O(\log N)$
- Acotado
  - Nunca más de dos rotaciones, ya que se sale del bucle while cuando se cumpla CASO 2 ó 3



COMILLAS  
ICAI

# RB Tree, eliminación de clave

# The Tree of Knowledge



COMILLAS  
ICAI