

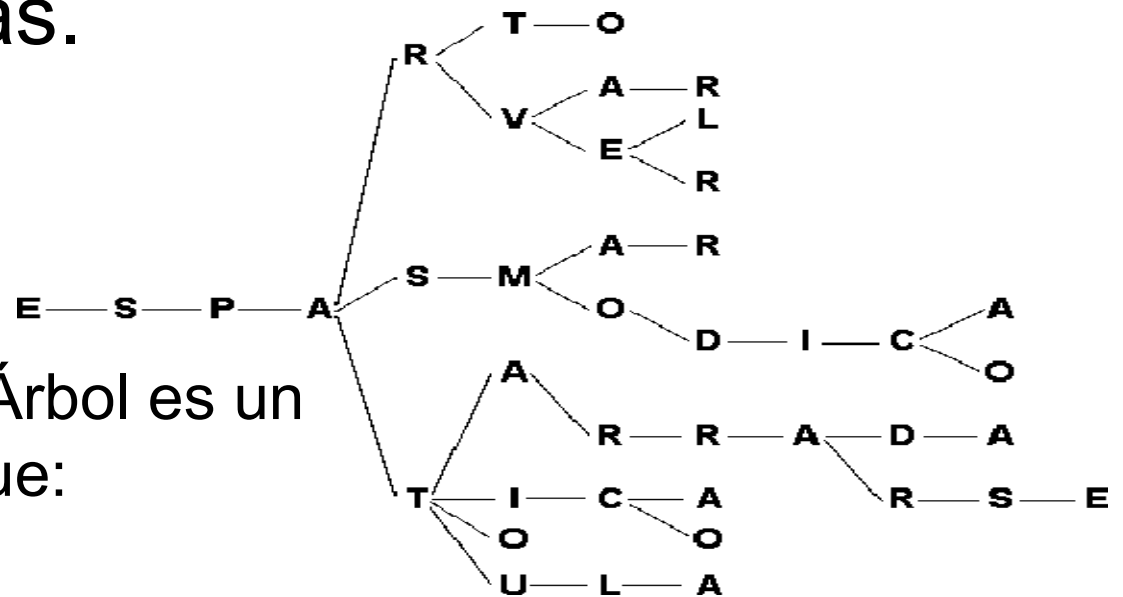
Unidad 7

Árboles Binarios

- Biblio: “Algoritmos y Estructuras de datos” de Aguilar y Martinez. Unidad 14 y 15
- Autor: Ing Rolando Simon Titiosky.

Árboles Generales

Intuitivamente el concepto de árbol implica una estructura donde los datos se organizan de modo que los elementos de información estén relacionados entre sí a través de ramas.

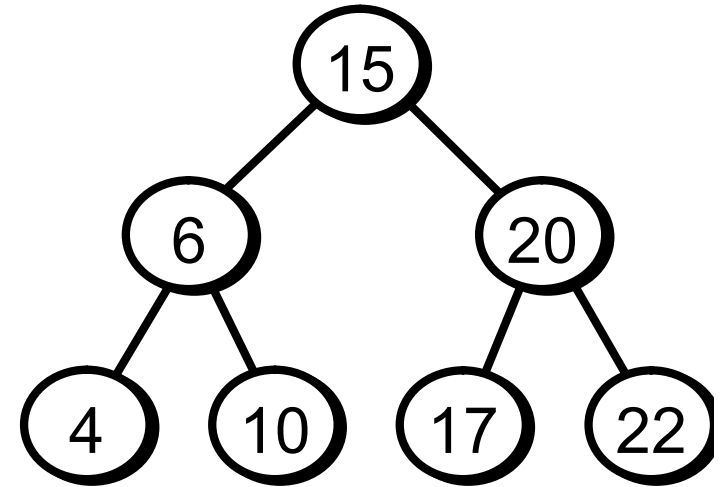


Definición recursiva: Árbol es un conjunto de nodos que:

- Es vacío, o bien,
- Tiene un nodo raíz del que descienden 0 o más subárboles.

Árboles Generales

- **Nodos:** conjunto finito de elementos.
- **Ramas:** conjunto finito de líneas dirigidas, que conectan nodos.
- **Grado del Nodo:** número de ramas descendentes con un nodo.
- **Raíz:** primer nodo de un árbol no vacío.
- **Camino:** secuencia de nodos en los que c/nodo es adyacente al siguiente.
 - Solo existe 1 camino entre la Raíz y un Nodo cualquier.
 - La distancia de un Nodo a la Raíz determina la rapidez de búsqueda.



Raíz=15

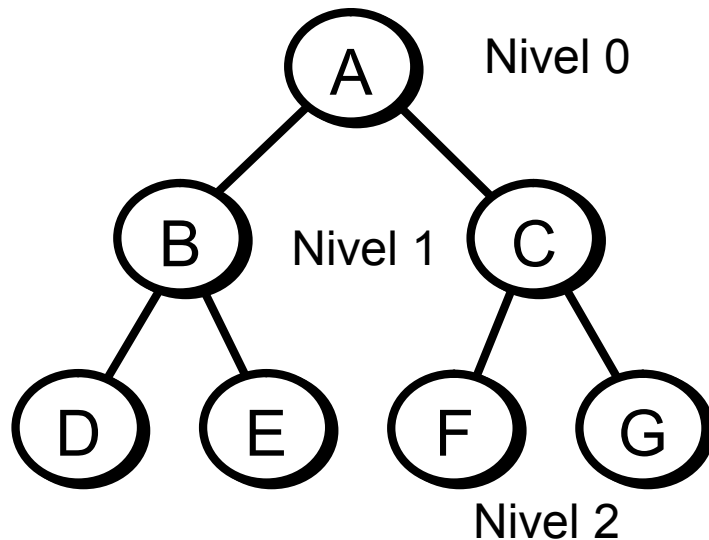
Nodos: 15, 6, 20....

Ramas(15, 6); (20, 17)

Grado del 15= $G(15)=2$

$G(10)=0$; $G(20)=2$

Terminología

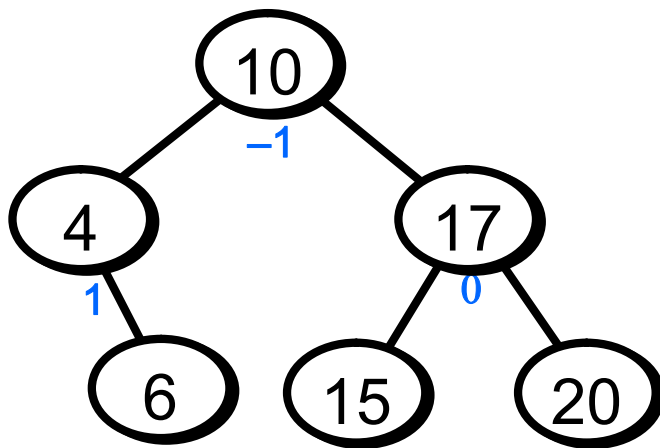


- Padres: A; B y C.
- Hijos: De A(B y C), de B (D y E)
- Descendientes de B: D y E
- Ascendientes de E: B y A.
- Hermano: {B, C}, {F, G}
- Hojas: D, E, F, G
- Altura del Árbol: 3
- Altura del Subárbol de B: 2

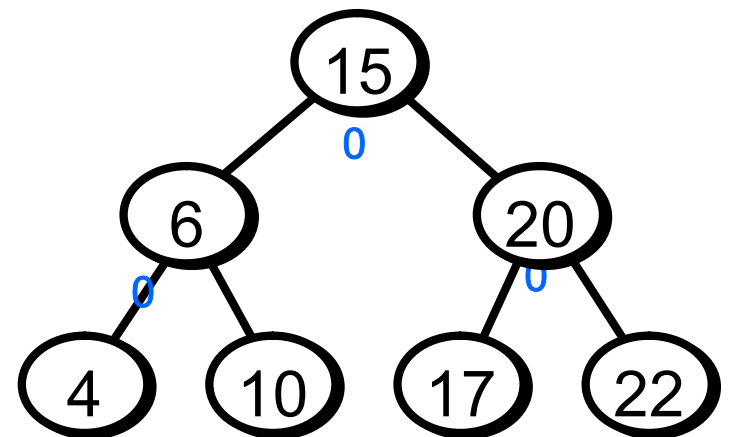
- **Padre:** tiene Nodos sucesores.
- **Hijos:** Nodos sucesores.
- **Descendientes:** Hijos de los hijos
- **Ascendientes:** los padre y abuelos de un nodo hijo.
- **Hermanos:** 2 o mas nodos del mismo padre.
- **Hojas:** nodo sin hijos .
- **Nivel de un nodo:** distancia a la raíz.
- **Altura o profundidad de un árbol:** nivel de la hoja del camino más largo desde la raíz más uno.
 - La altura de un árbol vacío es 0.
- **Subárbol:** cualquier estructura conectada por debajo del raíz.
C/nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes del nodo.

Equilibrio en Árbol Binario

- Factor de Equilibrio: diferencia de altura entre los 2 subárboles de un nodo.
 - $fe = h_d - h_i$.
 - Árbol Equilibrado: $-1 < fe < 1$. Ej: $fe = 3 - 2 = 1$
 - Árbol Perfectamente Equilibrado: $fe = 0$.



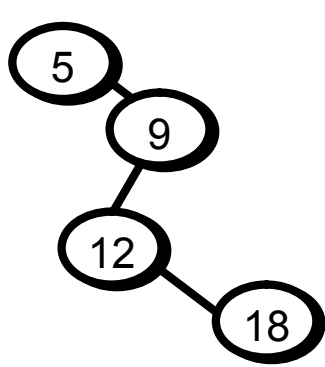
Árbol Equilibrado



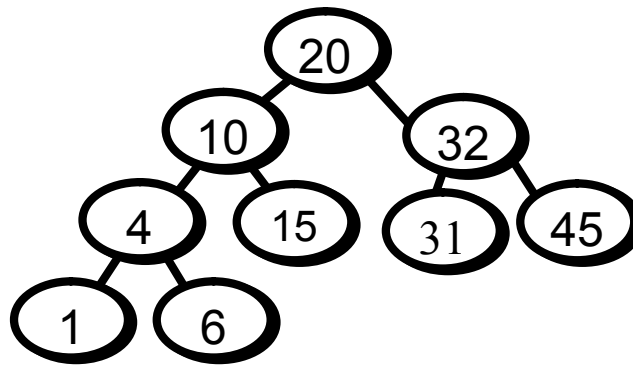
Árbol Perfectamente Equilibrado

Árbol Binario

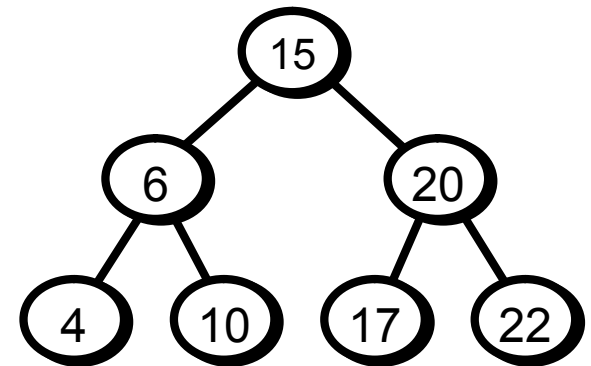
- Árbol donde ningún nodo puede tener mas de 2 subárboles.
- En cualquier nivel n , solo puede contener de 1 a $2^{n+1}-1$ nodos
- **Árbol Completo:** de **Altura n** es un árbol en el que para c/nivel, del 0 al $n-1$, **está lleno de nodos**. Todos los nodos hoja a nivel n ocupan posiciones a la izquierda.
- **Árbol Lleno:** tiene el máximo número de entradas para su altura: 2^n . **A Nivel k , tendrá $2^{k+1}-1$ nodos.** (Nivel = Profundidad-1)
 - Lleno de Profundidad 3 = Nivel 2 $\Rightarrow 2^{2+1}-1$ nodos = $2^3-1=7$ nodos
- **Árbol Degenerado:** hay un solo nodo hoja (el 18) y cada nodo no hoja tiene solo un hijo. **Equivalente a una lista enlazada.**



Degenerado de profundidad 4



Completo de profundidad 4



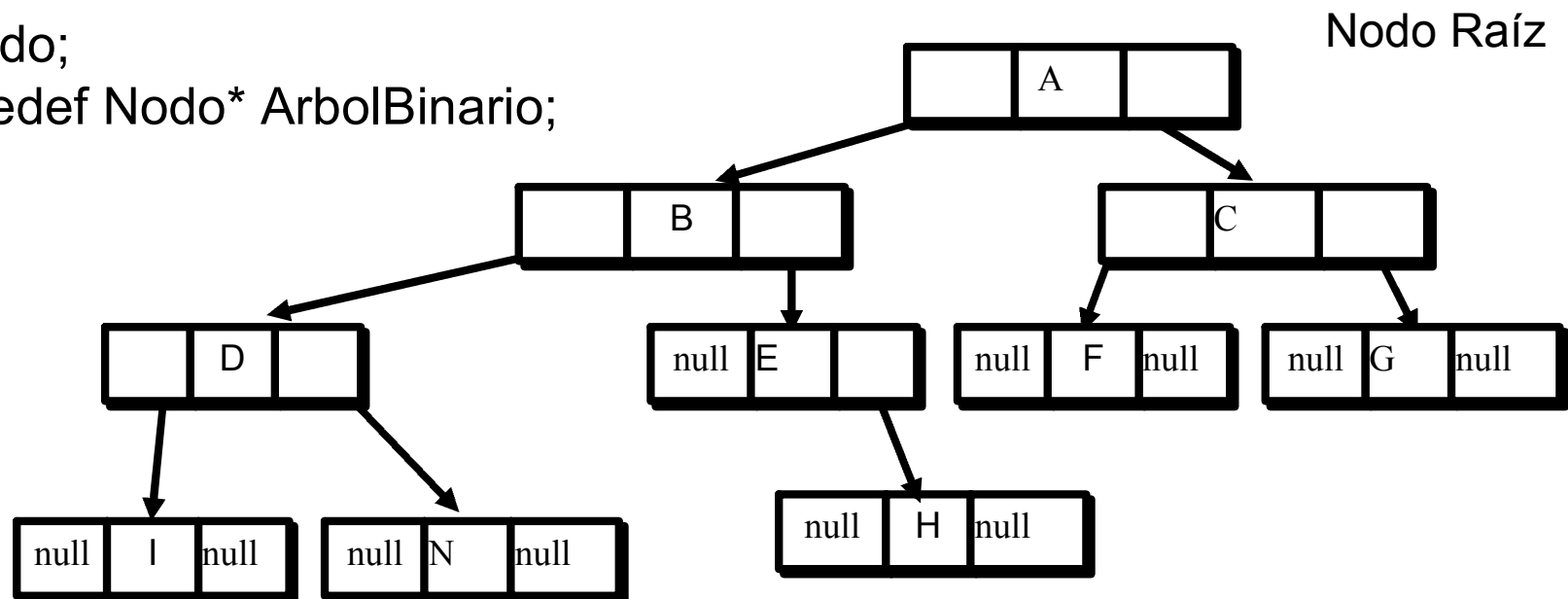
Lleno de profundidad 3

TAD Árbol Binario: Especificación

- Tipo de Dato
 - Nodos del Árbol (se especifica en filmina siguiente)
- Operaciones
 - Altura: de un árbol
 - Borrar: Elimina del árbol a un nodo dado
 - Búsqueda: Buscar un elemento en un Árbol de Búsqueda
 - Construir: crea un árbol con un elemento raíz y dos ramas.
 - Copiar: crear una copia del árbol
 - CrearArbol: Inicia un árbol vacío
 - Derecho: da la rama derecha de un árbol dado.
 - Elementos: determina el número de elementos del árbol
 - EsVacio: comprueba si el árbol tiene nodos
 - Iguales: determinar si dos árboles son idénticos
 - Insertar: inserta un nodo dentro de un árbol
 - Izquierdo: da la rama izquierda de un árbol dado.
 - Pertenece: Determina si un elemento pertenece a un árbol.
 - Recorrer: el árbol de acuerdo algunos de los criterios
 - Profundidad: determina la profundidad de un árbol dado
 - Raiz: devuelve el nodo raíz.
 -

Estructura de Datos del Árbol Binario

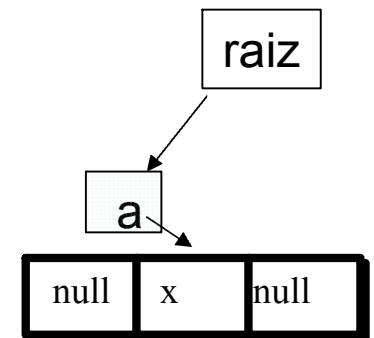
```
Typedef int TipoElemento;  
Typedef struct nodo  
{  
    TipoElemento    dato;  
    struct nodo    *izq;  
    struct nodo    *der;  
} Nodo;  
Typedef Nodo* ArbolBinario;
```



Creación del Árbol

```
Void nuevoArbol(ArbolBinario* raíz, ArbolBinario ramalq,  
    TipoElemento x, ArbolBinario ramaDer)  
{ *raíz=crearNodo(x);  
  (*raíz)->izq=ramalq;  
  (*raíz)->der=ramaDer;  
}
```

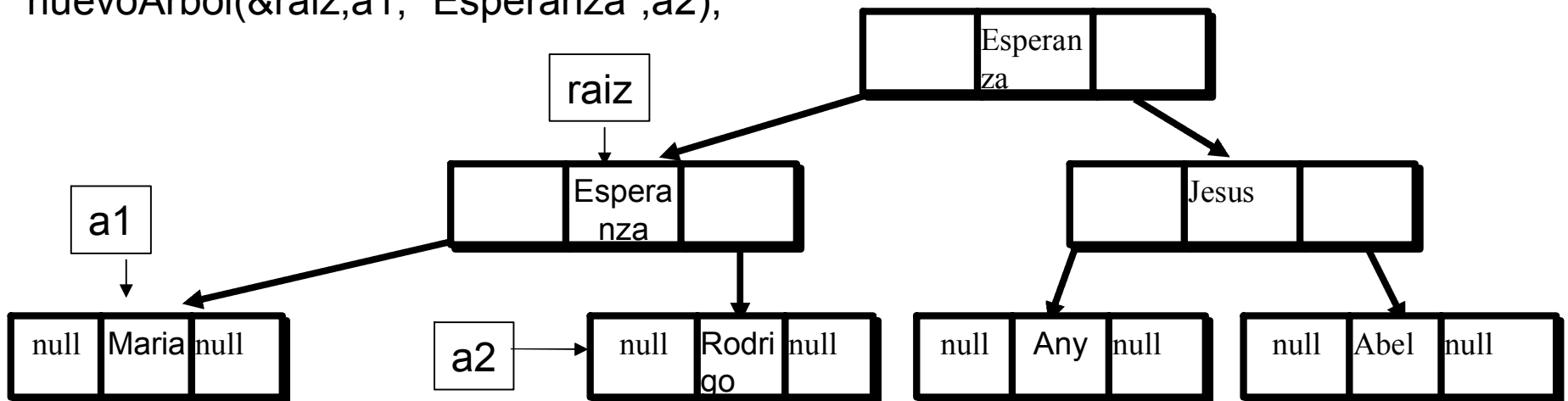
```
ArbolBinario crearNodo(TipoElemento x)  
{ ArbolBinario a;  
  a=(ArbolBinario) malloc(sizeof(Nodo));  
  a.dato=x;  
  a.lzq=a.Der=NULL;  
  return a;  
}
```



Ejemplo trivial

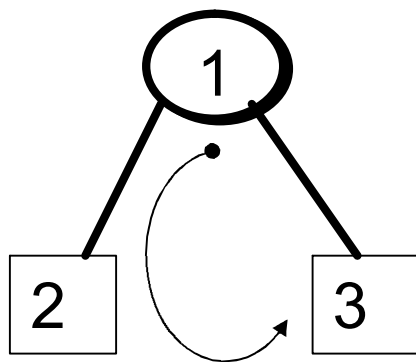
```
ArbolBinario raíz, a1, a2;  
Pila pila;  
nuevoArbol(&a1,NULL, "Maria",NULL);  
nuevoArbol(&a2,NULL, "Rodrigo",NULL);  
nuevoArbol(&raíz,a1, "Esperanza",a2);  
Insertar(&pila, raíz);  
nuevoArbol(&a1,NULL, "Any",NULL);  
nuevoArbol(&a2,NULL, "Abel",NULL);  
nuevoArbol(&raíz,a1, "Jesus",a2);  
Insertar(&pila, raíz);  
a2=quitar(&pila);  
a1=quitar(&pila);  
nuevoArbol(&raíz,a1, "Esperanza",a2);
```

```
Void nuevoArbol(ArbolBinario* raíz,  
ArbolBinario ramalq, TipoElemento x,  
ArbolBinario ramaDer)
```

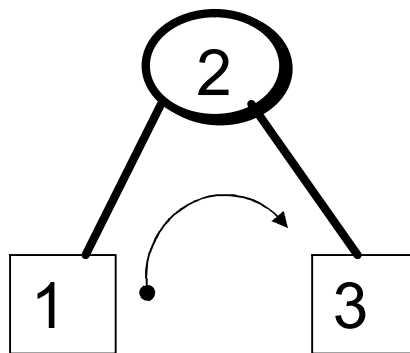


Recorrido del Árbol

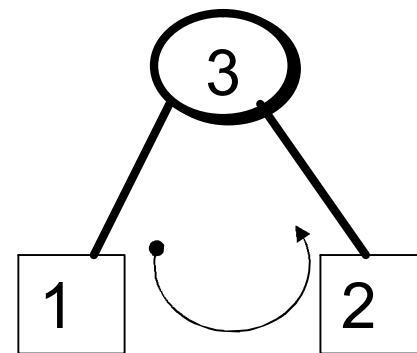
- Recorrer el Árbol significa que cada nodo sea procesado una vez y solo una en un secuencia determinada. Existen 2 enfoques generales
 - **Recorrido en Profundidad:** el proceso exige alcanzar las profundidades de un camino desde la raíz hacia el descendiente mas lejano del primer hijo, antes de proseguir con el segundo.
 - **Recorrido en Anchura:** el proceso se realiza horizontalmente desde la raíz a todos su hijos antes de pasar con la descendencia de alguno de ellos.



Preorden RID
Raiz–IZQ–DER



EnOrden IRD
IZQ–Raiz–DER



PostOrden IDR
IZQ–Raiz–DER

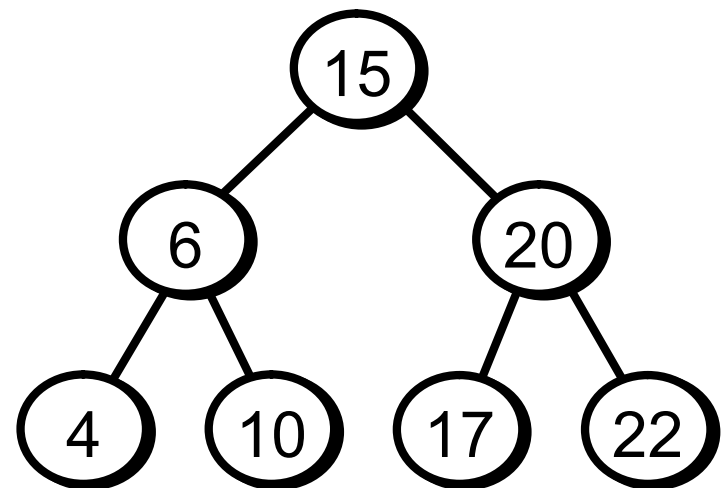
Recorrido PreOrden (RID)

```
void preOrden(ArbolBinario raiz)
{ if(raiz)
  {  visitar(raiz->dato);
    preOrden(raiz->izq);
    preOrden(raiz->der);
  }
}
```

```
void visitar(TipoElemento x)
{printf(" %i ", x);}
}
```

El recorrido en PreOrden del
árbol es el siguiente:

15, 6, 4, 10, 20, 17, 22

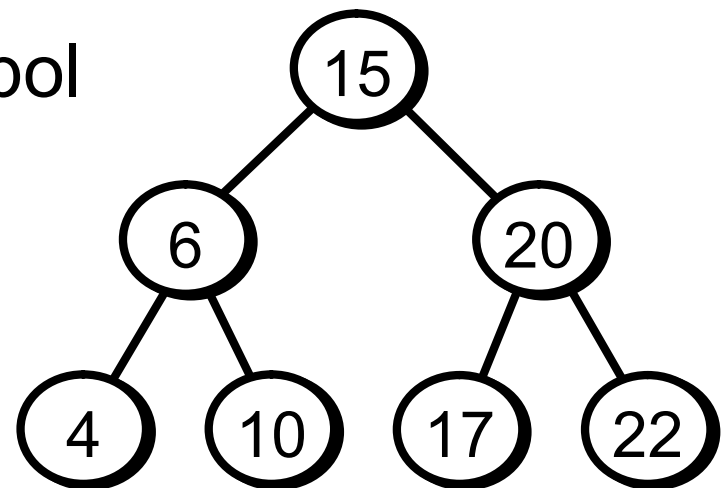


Recorrido EnOrden (IRD)

```
void enOrden(ArbolBinario raíz)
{ if(raíz)
  { enOrden(raíz->izq);
    visitar(raíz->dato);
    enOrden(raíz->der);
  }
}
```

El recorrido en EnOrden del árbol
es el siguiente:

4, 6, 10, 15, 17, 20, 22

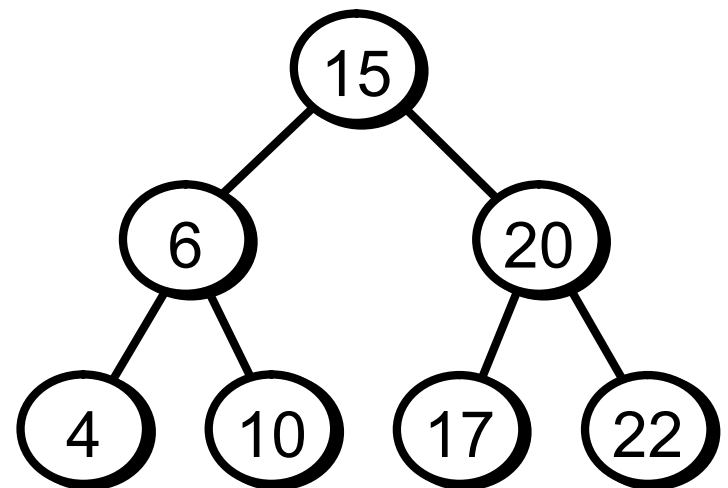


Recorrido PostOrden (IDR)

```
void PostOrden(ArbolBinario raíz)
{ if(raíz)
  { PostOrden(raíz->izq);
    PostOrden(raíz->der);
    visitar(raíz->dato);
  }
}
```

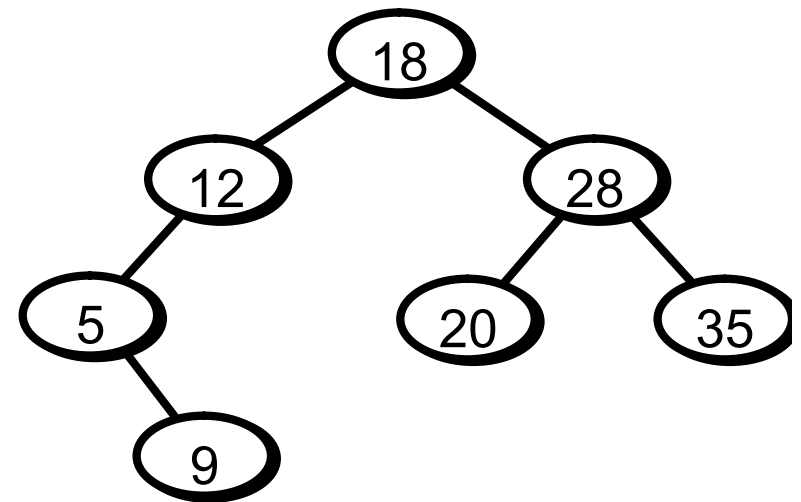
El recorrido en PostOrden del
árbol es el siguiente:

4, 10, 6, 17, 22, 20, 15



Comparemos Recorridos

- PreOrden RID
 - 18, 12, 5, 9, 28, 20, 35
- EnOrden IRD
 - 5, 9, 12, 18, 20, 28, 35
- PostOrden IDR
 - 9, 5, 12, 20, 35, 28, 18



Árbol Binario de Búsqueda

Como en el Ejemplo, dado un Nodo cualquiera del Árbol, todos los datos de su Subárbol IZQ son menores, y los del Subárbol DER son mayores: $I < R < D$. Por esto, solo el recorrido IRD, dará la secuencia ordenada del árbol.

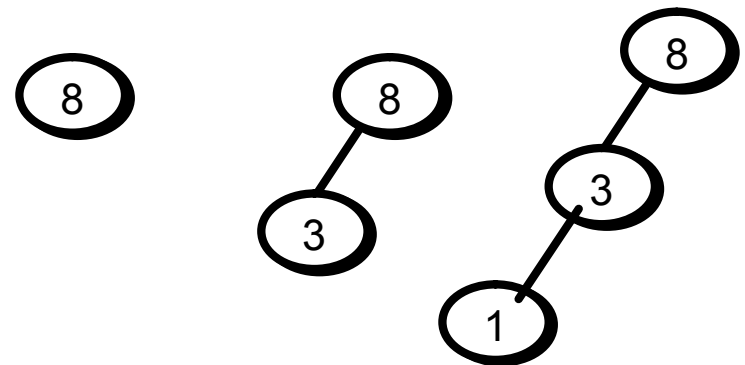
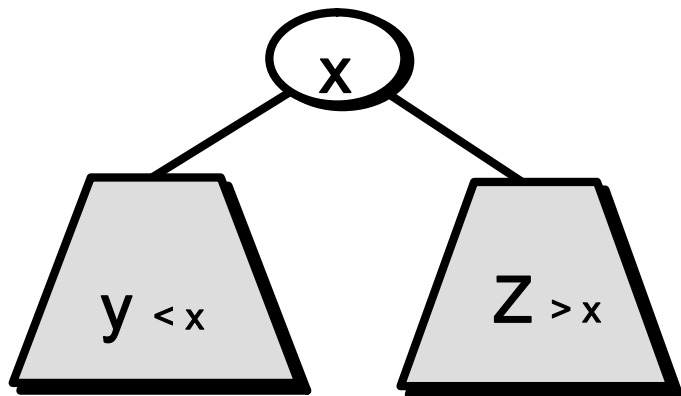
Profundidad de un Árbol Binario

```
int profundidad (ArbolBinario raíz)
{ if (!raíz) return 0 ; /*Árbol vacío*/
  else
  {   int profundidadI = profundidad (raíz -> izq);
      int profundidadD = profundidad (raíz -> der);
      if (profundidadI > profundidadD)
          return profundidadI + 1;
      else
          return profundidadD + 1;
  }
}
```

Altura o profundidad de un árbol: nivel de la hoja del camino más largo desde la raíz más uno. La altura de un árbol vacío es 0.

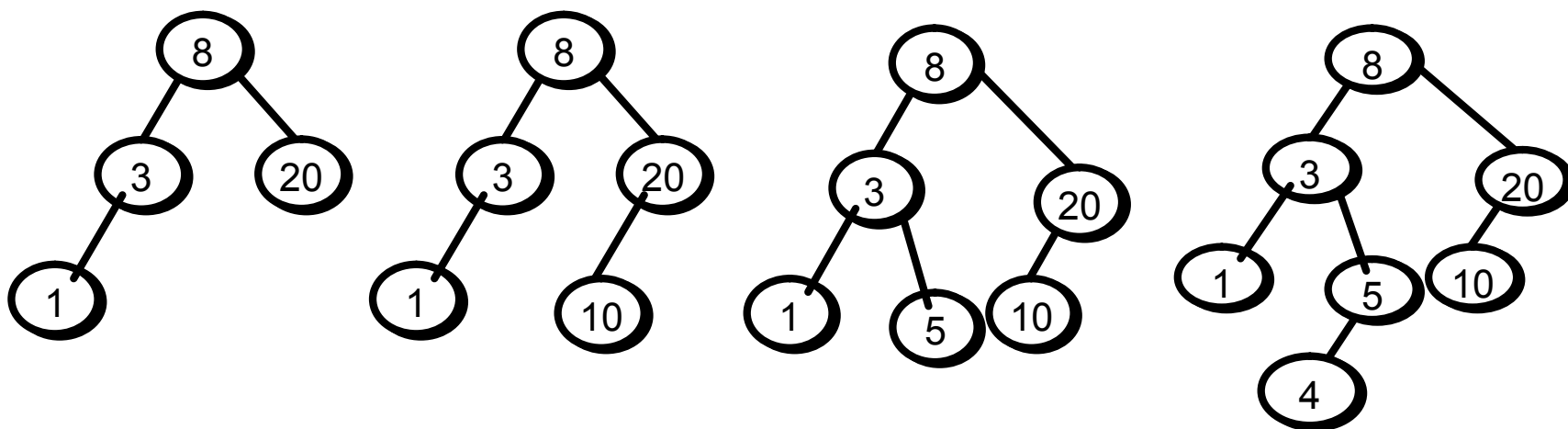
Creación de un Árbol Binario de Búsqueda

- Se desean almacenar los números 8, 3, 1, 20, 10, 5 y 4 en un Árbol Binario de Búsqueda.
- La Única Regla es que $L < R < D$.
- Inicialmente el Árbol está vacío y la única opción para el 8 es ser el Nodo Raíz.
- Luego viene 3, es menor a 8: va al subárbol IZQ
- Luego el 1, menor al 8 y al 3.



Creación de un Árbol Binario de Búsqueda (cont.)

- Se desean almacenar los números 8, 3, 1, 20, 10, 5 y 4 en un Árbol Binario de Búsqueda.
- Luego el 20, mayor al 8
- Así hasta insertar todos los elementos.
- *PROPIEDAD de los Árboles Binarios de Búsqueda:*
 - *No son únicos y depende del orden en el cual se fueron insertando los elementos*



Búsqueda en Árbol de Búsqueda

```
Nodo *buscar(Nodo *raiz, TipoElemento buscado)
{
    if(!raiz)      return 0;    /*Árbol vacío*/
    else if (buscado==raiz->dato) return raiz;
    else if (buscado<raiz->dato)
        return buscar(raiz->izq, buscado);
    else
        return buscar(raiz->der, buscado);
}
```

Insertar en Árboles de Búsqueda

- Es una extensión de la operación de Búsqueda
 - Si el árbol está vacío se inserta directamente
 - SINO: Buscará en el árbol el lugar correcto de Inserción

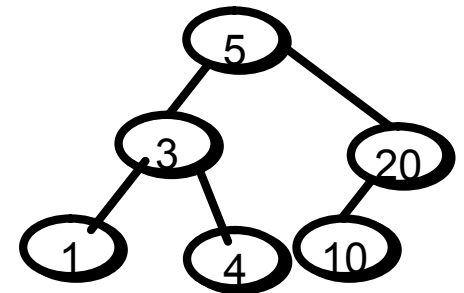
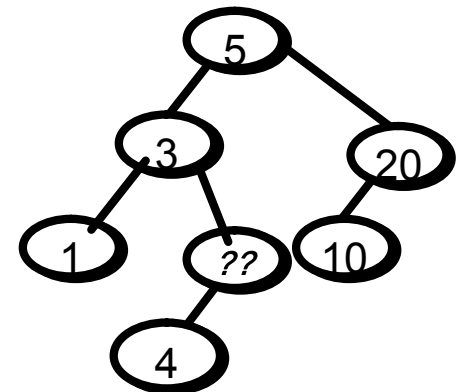
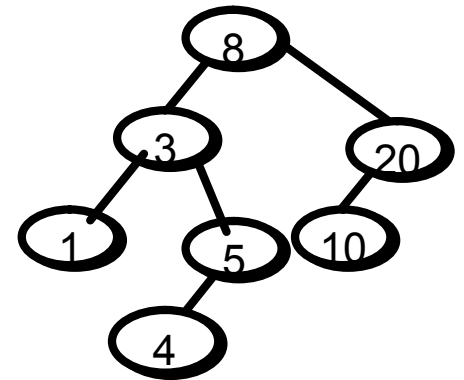
```
void insertar (Nodo** raiz, TipoElemento dato)
{  if (!(*raiz))
    *raiz = crearNodo(dato);
  else if (dato < (*raiz) -> dato)
    insertar (&((*raiz) -> izdo), dato);
  else
    insertar (&((*raiz) -> dcho), dato);
}
```

Eliminar de un Árbol Binario de Búsqueda

- Es una extensión de la operación de búsqueda.
- Mantendrá la estructura de Árbol Binario de Búsqueda.
- Presenta 2 casos bien diferenciados
 - 1er: El Nodo a eliminar es Hoja o Hijo único
 - Se Asignará al Padre del nodo a eliminar, su único descendiente. (El Abuelo se encargará del Nieto)
 - 2do: El nodo a borrar tiene sus dos hijos. Se Puede
 - Reemplazar el dato del nodo con el Menor de las Claves Mayores del Subárbol Derecho.
 - Reemplazar el dato del nodo con el Mayor de las Claves Menores del Subárbol Izquierdo.
 - Bajaremos al 1er nodo de la Rama Izquierda
 - Como la clave mayor está a la derecha, se continúa bajando a derecha hasta encontrar un nodo hoja (El Mayor de los Menores que reemplazará el dato a eliminar).

Eliminar de un Árbol Binario de Búsqueda

- Eliminar el Nodo 8.
- Bajaremos al 1er nodo de la Rama Izquierda: Nodo 3
- Como la clave mayor está a su derecha, se continúa bajando a derecha hasta encontrar un nodo sin rama DER: El nodo 5.
- Se reemplaza el 8 por el 5.
- Queda un caso de Hijo Único.
- El Abuelo (3) se encargará del Nieto (4).



Eliminar de un Árbol Binario de Búsqueda

```
void eliminar (Nodo** r, TipoElemento
    dato)
{
    if (!(*r)) puts("Nodo no encontrado");
    else if (dato < (*r) -> dato)
        eliminar(&(*r) -> izdo, dato);
    else if (dato > (*r) -> dato)
        eliminar(&(*r) -> dcho, dato);
    else /* Nodo encontrado */
    {
        Nodo* q; /* puntero al nodo a suprimir */
        q = (*r); /* r es el ptr del nodo Padre */
        if (q -> izdo == NULL)
            (*r) = q -> dcho;
        else if (q -> dcho == NULL)
            (*r) = q -> izdo;
        else /* tiene rama izquierda y derecha */
            reemplazar(&q);
        free(q);
    }
}
```

```
void reemplazar(Nodo** act)
{
    Nodo* a, *p;
    p = *act;
    a = (*act) -> izdo; /* menores a IZQ */
    while (a -> dcho)
    {
        p = a; /* buscamos el Mayor a DER */
        a = a -> dcho;
    }

    /* Cambio de Campo Datos */
    (*act) -> dato = a -> dato;
    /* Al Abuelo p, se hace cargo de nieto IZQ */
    if (p == (*act))
        p -> izdo = a -> izdo;
    else
        p -> dcho = a -> izdo;
    (*act) = a;
} /* Ojo: No puede tener nada a Derecha,
    pues abríamos bajado en el while. */
```

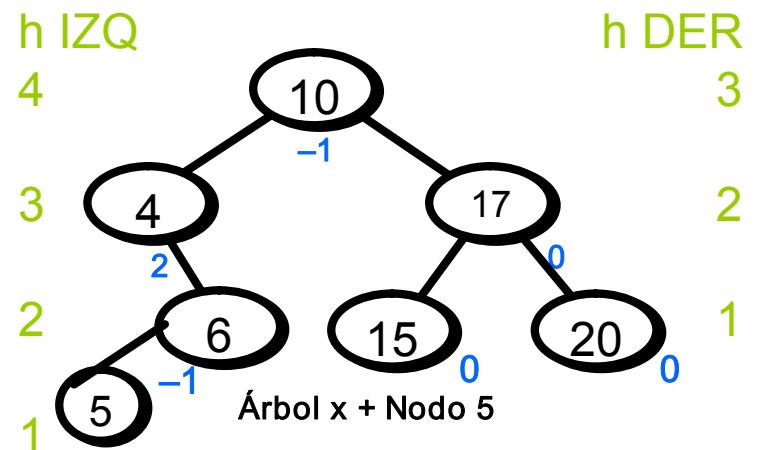
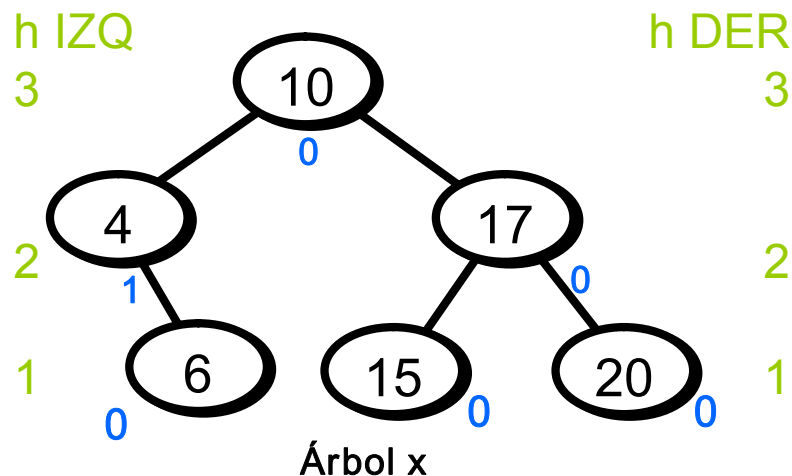
AVL: Árbol Binario Ordenado y Balanceado

- Considerando a un árbol de n nodos.
 - Eficiencia en Búsquedas en árboles Equilibrados $F(n)=\text{Log } n$
 - En los demás: $\log n \leq F(n) \leq n$
- Para Optimizar las Búsquedas, se intentará mantener los Árboles Binarios Ordenados $I < R < D$ y Equilibrados.
 - Llamados *Árboles AVL* (Adelson–Velskii–Landis)
 - Se caracterizan porque la Altura $B = h_d - h_i : -1 \leq B \leq 1$
 - Restringe la altura de cada uno de los Subárboles
 - Se necesita un campo “FE” (Factor de Equilibrio) en cada nodo.

```
struct nodo
{
    TipoElemento  dato;
    int fe;
    struct nodo   *izq, *der;
} Nodo;
```


Inserción en AVL

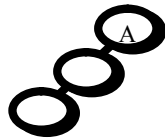
- La Inserción se hace siguiendo el camino de búsqueda
- Puede aumentar la altura de una rama, de manera que cambie el factor de equilibrio de dicho nodo.
 - Implica que se retornará por el camino de búsqueda para actualizar el FE de c/nodo
 - Se puede llegar a Desbalancear (Altura=2) => ReBalanceo.
 - O Puede mejorar: Si al Árbol X se le Inserta el 3, resultará en Perfectamente Balanceado.
- El Proceso termina cuando se llega a la Raiz o cuando termina el Rebalanceo del mismo.



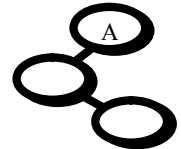
Reestructuración AVL

Hay 4 casos posibles a tener en cuenta, según donde se hizo la Inserción.

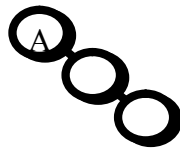
1. Inserción en el Subárbol IZQ De la Rama IZQ de A



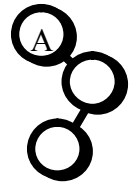
2. Inserción en el Subárbol DER De la Rama IZQ de A



3. Inserción en el Subárbol DER De la Rama DER de A



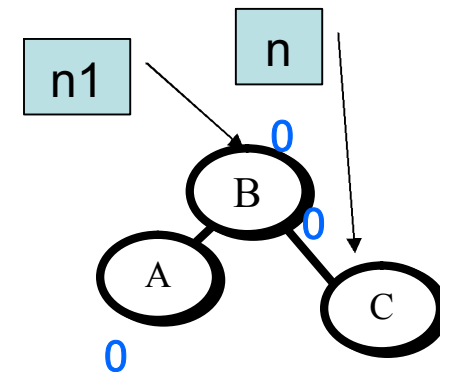
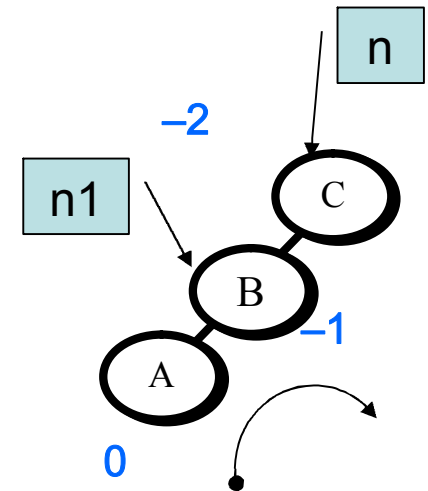
4. Inserción en el Subárbol DER De la Rama IZQ de A



- Solución: La ROTACION le devuelve el equilibrio al árbol.
 - Rotación Simple: Caso 1 y 3: Implica a A y su descendiente
 - Rotación Doble: Caso 2 y 4: Implica a los 3 nodos
- Soluciones simétricas: En c/caso, las ramas están opuestas.

AVL: Rotación Simple

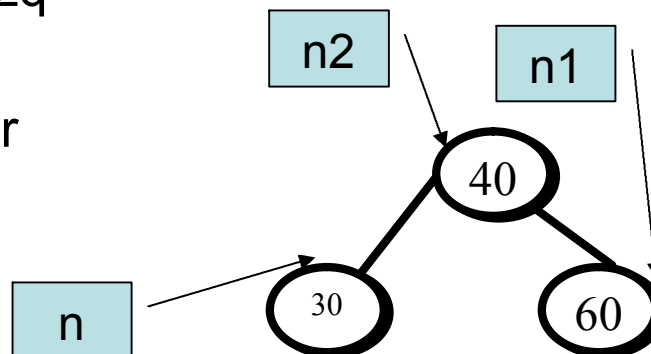
- Luego de Insertar el Nodo A el árbol quedó desbalanceado.
- Según que rama ha crecido, la Rotación Simple puede ser de izquierda–izquierda (II) o derecha–derecha(DD).
- ***Movimientos de Ptrs.***
 - n es ptro al nodo problema: $Fe=-2$
 - IZQ (nuestro ej). $N1$ apunta a rama IZQ
 - $n \rightarrow izq = n1 \rightarrow der$
 - $n1 \rightarrow der = n$
 - $n = n1$
 - DER. $N1$ apunta a rama DER
 - $n \rightarrow der = n1 \rightarrow izq$
 - $n1 \rightarrow izq = n$
 - $n = n1$



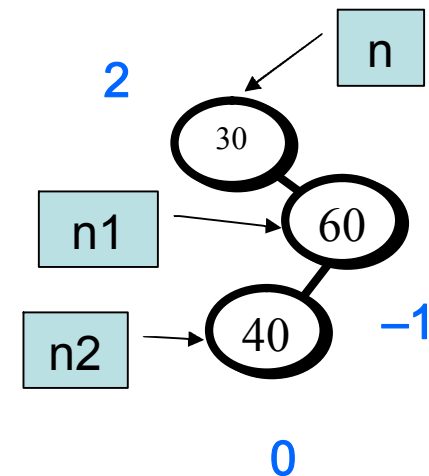
AVL: Rotación Doble

Movimientos de Ptrs.

- n apunta al nodo problema ($Fe=-2$); $n1$ al hijo de n con problemas, $n2$ al hijo de $n1$
- *D*(nuestro ej). Derecha–izquierda
 - $n1 \rightarrow izq = n2 \rightarrow der$
 - $n2 \rightarrow der = n1$
 - $n \rightarrow der = n2 \rightarrow izq$
 - $n2 \rightarrow izq = n$
 - $n = n2$
- *ID*: izquierda–derecha
 - $n1 \rightarrow der = n2 \rightarrow izq$
 - $n2 \rightarrow izq = n1$
 - $n \rightarrow izq = n2 \rightarrow der$
 - $n2 \rightarrow der = n$
 - $n = n2$



La solución consiste en subir el 40, pender el 30 a su izquierda y el 60 a su derecha.



Eliminación de un Nodo AVL

- Luego de eliminar un nodo con cierta clave, el árbol resultante debe seguir siendo AVL
- El Algoritmo puede descomponerse en dos partes bien diferenciadas
 1. Eliminar el Nodo: según la estrategia que vimos en la Eliminación de Nodo en Árbol de Búsqueda
 2. Balancear: se debe seguir el camino de búsqueda en sentido inverso:
 - 2.1. Arreglando el Factor de Equilibrio
 - 2.1. Restaurar el Equilibrio allí donde se rompe.

Trabajos Prácticos Unidad 7

COMO MINIMO, REALIZAR:

- De la Bibliografía
 - Del Capitulo 14:
 - Ejercicios: 14.1 al 14.8, 14.11
 - Problemas: 14.2, 14.3, 14.7
 - Del Capitulo 15:
 - Ejercicios: 15.1, 15.6, 15.10
 - Problemas: 15.1
- Complementarios:
 - Equilibrar los Árboles de 14.1 al 14.8, 14.11