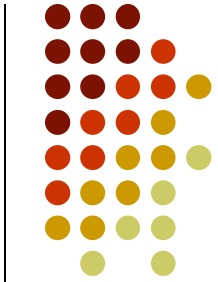


PROGRAMACION 3

TUDAI

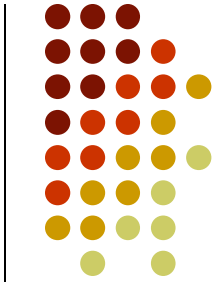
Tema 2 - CURSADA 2024

Algoritmos de Ordenamiento
Árboles

Prof. Federico Casanova
Facultad de Ciencias Exactas - UNICEN

Parte I

El problema de ordenar



Se tiene un conjunto de elementos, sobre los cuales se establece una relación de orden.
Se quiere luego ordenarlos en forma creciente o decreciente.

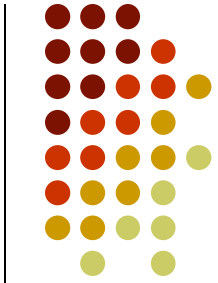
Generalmente los elementos son números y la relación de orden viene dada por los operadores < , > , =

También es común requerir otros tipos de orden sobre elementos, por ejemplo el orden lexicográfico sobre palabras.

Mantener los datos ordenados nos permite en general acelerar las búsquedas en el conjunto, y brindar la posibilidad de listar ordenado.

Así es que desde la década del 50 se estudian algoritmos para ordenar conjuntos de datos en tiempos buscando computacionales cada vez menores.

Ordenamiento
Algoritmo de Burbujeo



Consiste en comparar pares de elementos adyacentes en un array y si están desordenados intercambiarlos hasta que estén todos ordenados.
El elemento mayor sube como una burbuja hacia la posición más alta.

6	5	3	1	8	7	2	4
6	5	3	1	8	7	2	4
5	6	3	1	8	7	2	4
5	3	6	1	8	7	2	4
5	3	1	6	8	7	2	4
5	3	1	6	7	8	2	4
5	3	6	1	7	2	8	4
5	3	6	1	7	2	4	8
3	5	6	1	7	2	4	8
3	5	6	1	7	2	4	8

……

Ordenamiento
Algoritmo de Burbujeo

Consiste en comparar pares de elementos adyacentes en un array y si están desordenados intercambiarlos hasta que estén todos ordenados. El elemento mayor sube como una burbuja hacia la posición más alta.

```
public static void burbujeo(int [ ] A) {
        int i, j, aux;
        for ( i=0; i < A.length - 1; i++)
            for ( j=0; j < A.length – i – 1 ; j++)
                if (A[ j ] > A[ j+1 ]) {
                   aux = A[ j+1 ];
                   A[ j+1 ] = A[ j ];
                   A[ j ] = aux;
                 }
}
```

$O(n^2)$

```
public void bubbleSortAdapt(int[] arr) {
        boolean swapped = true;
        int j = 0;
        int tmp;
        while (swapped) {
            swapped = false;
            j++;
            for (int i=0; i<arr.length - j; i++) {
                if (arr[i] > arr[i + 1]) {
                    tmp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = tmp;
                    swapped = true;

                }
            }
        }
}
```

Mejorado (Adaptativo)
Para caso por ej:
8 5 7 9 11 15 25 32 41 50

Ordenamiento
Algoritmo Mergesort

Se pueden mezclar eficientemente dos secuencias **ordenadas** de tamaño n/2 en otra de tamaño n que también resulte ordenada ?
Cómo lo haría ?

1-12-14-15-25-30 4-5-10-17-20-45

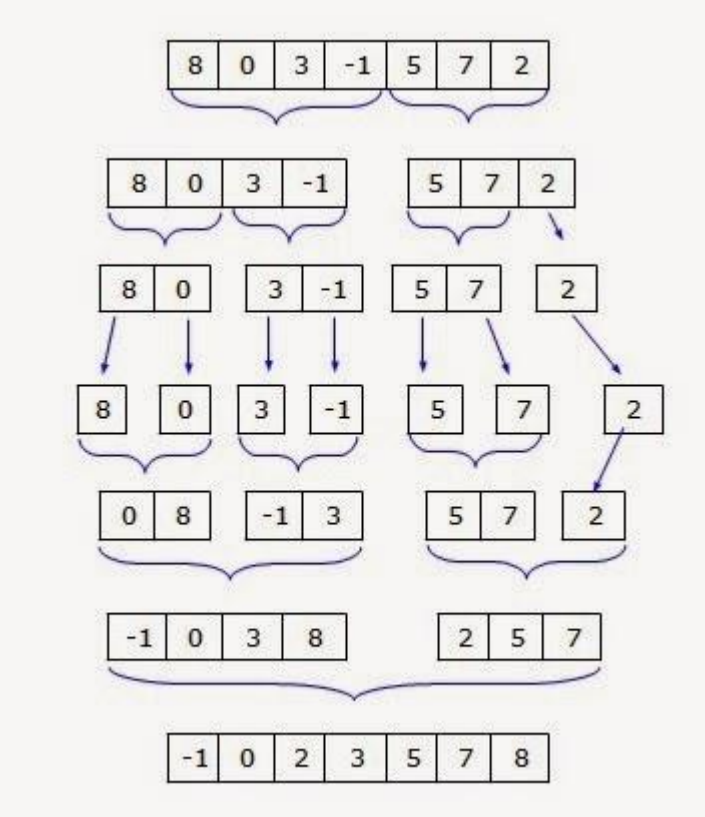
1-4-5-10-12-14-15-17-20-25-30-45

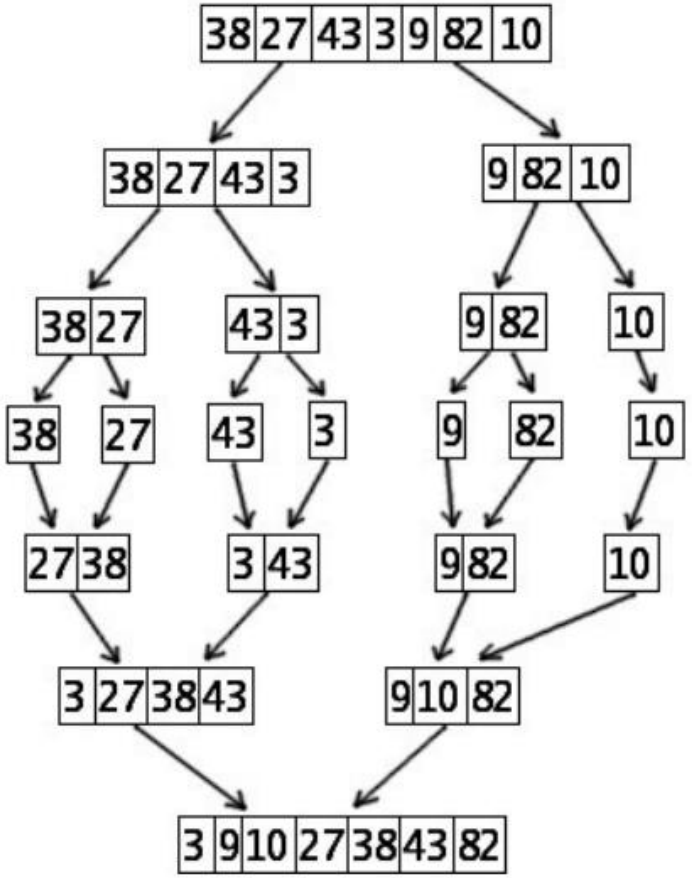
El tiempo que tarda (complejidad temporal) dependerá de la cantidad de elementos total n ? **O(n)**

Entonces si sabemos mezclar eficientemente dos mitades ordenadas.
Podríamos repetir el algoritmo y decir que el problema de ordenar un array de tamaño n, se puede resolver si recursivamente ordenamos la primer mitad de n/2 elementos, luego la segunda mitad de n/2 elementos y luego las mezclamos para obtener el array ordenado de n elementos.
El caso base (o de corte) cuál será ?
Será cuando lleguemos a un array de un solo elemento.

Cuántas veces se puede ir dividiendo por 2 hasta obtener el caso base ? **$\log_2 n$**

Ordenamiento
Algoritmo Mergesort





Tanto para este algoritmo como para los otros de ordenamiento les recomiendo ver animaciones tales como:
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Ordenamiento
Algoritmo Mergesort

```
public class Mergesort {
  private int[ ] numbers;
  private int[ ] helper;
  private int size;

  public void sort(int[ ] values) {
    this.numbers = values;
    size = values.length;
    this.helper = new int[size];
    mergesort(0, size - 1);
   }

  private void mergesort(int low, int high) {
    // si low es menor que high continua el ordenamiento
    // si low no es menor que high entonces el array está ordenado
    // ya que es el caso base donde el array tiene un solo elemento.
    if (low < high) {
     // obtener el indice del elemento que se encuentra en la mitad
     // al ser int redondea el resultado al entero menor
     int middle = (low + high) / 2;
     // ordenar la mitad izquierda del array – llamada recursiva
     mergesort(low, middle);
     // ordenar la mitad derecha del array – llamada recursiva
     mergesort(middle + 1, high);
     // combinar ambas mitades ordenadas
     merge(low, middle, high);
    }
  }
```

Requiere memoria auxiliar adicional

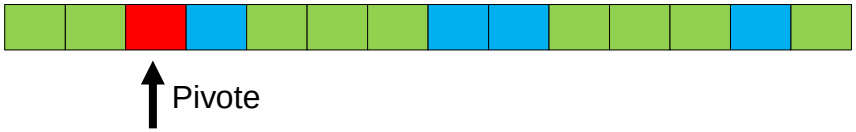
$O(n.\log_2 n)$

```
private void merge(int low, int middle, int high) {

  // copiar ambas partes al array helper
  for (int i = low; i <= high; i++) {
    helper[i] = numbers[i];
   }

  int i = low;
  int j = middle + 1;
  int k = low;
  // copiar de manera ordenada al array original los valores de la
  // mitad izquierda o de la derecha
  while (i <= middle && j <= high) {
     if (helper[ i ] <= helper[ j ]) {
        numbers[ k ] = helper[ i ];
        i++;
     } else {
        numbers[ k ] = helper[ j ];
        j++;
     }
   k++;
  }
  // si quedaron elementos copiarlos al array original
  while (i <= middle) {
    numbers[ k ] = helper[ i ];
    k++;
    i++;
  }
  while (j <= high) {
    numbers[ k ] = helper[ j ];
    k++;
    j++;
  }
 }
}
```

Ordenamiento
Algoritmo Quick-Sort

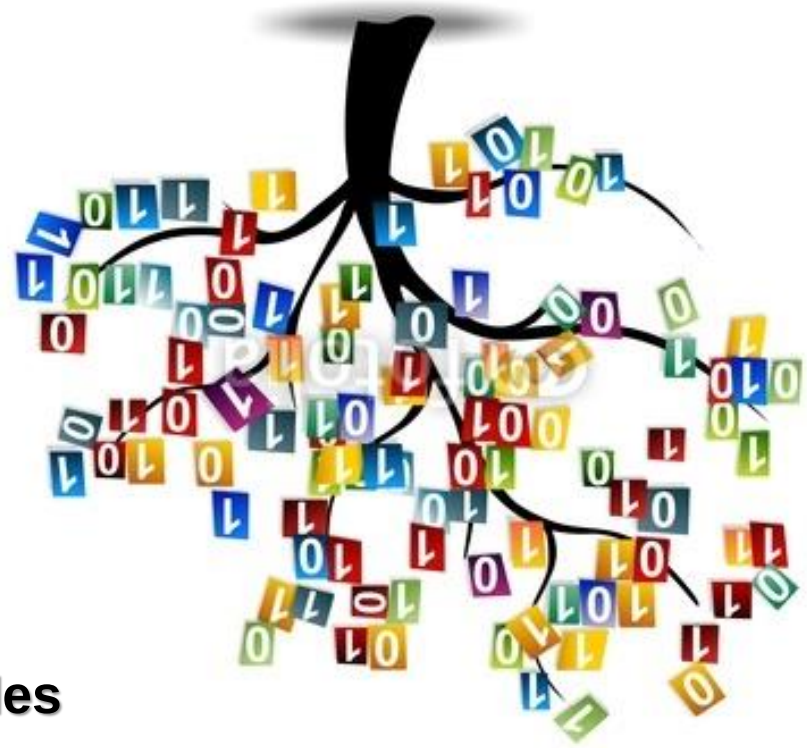
El algoritmo consiste en seleccionar uno de los elementos del array como valor pivote alrededor del cual el resto de los elementos serán reordenados.
Todo lo que sea menor que el pivote es movido a la izquierda del pivote (a la partición izquierda).
De forma similar todo lo que sea mayor que el pivote es movido a la partición derecha.
Notar que el pivote quedó en el lugar correcto del array final ordenado.

Una vez hecho esto realizar un QuickSort de cada partición.



$O(n^2)$ **$\Omega(n.\log_2 n)$** *No requiere memoria adicional !*

Buscar e implementar !

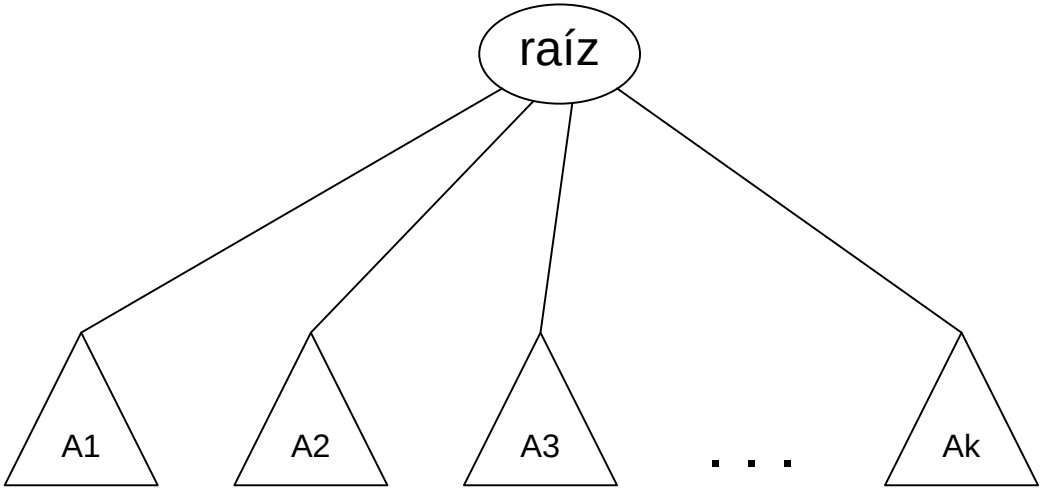


Árboles

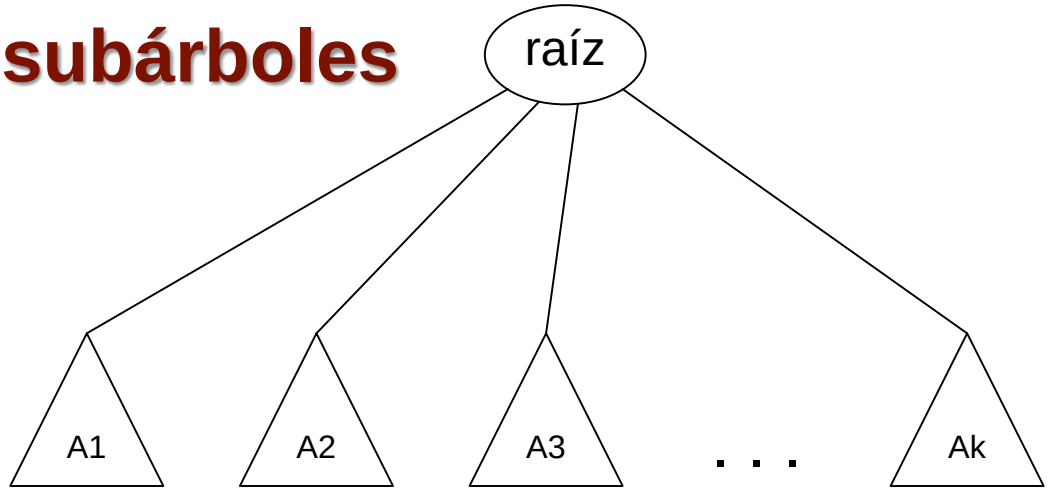
Árboles

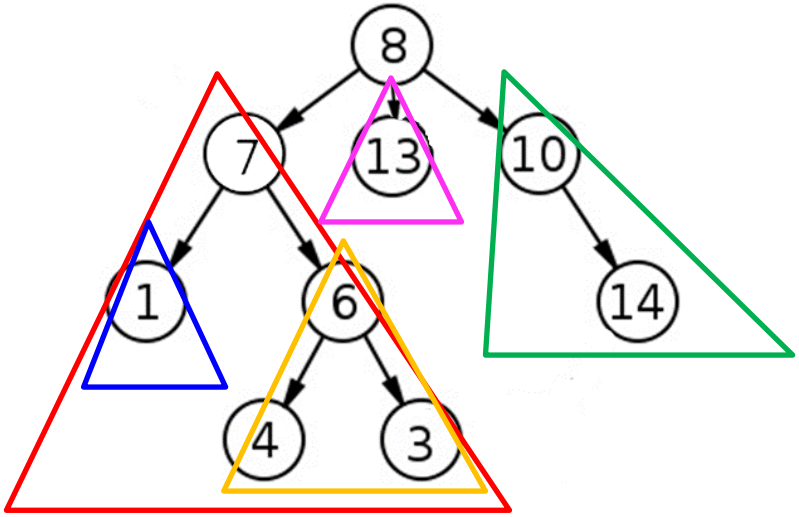
Definición general

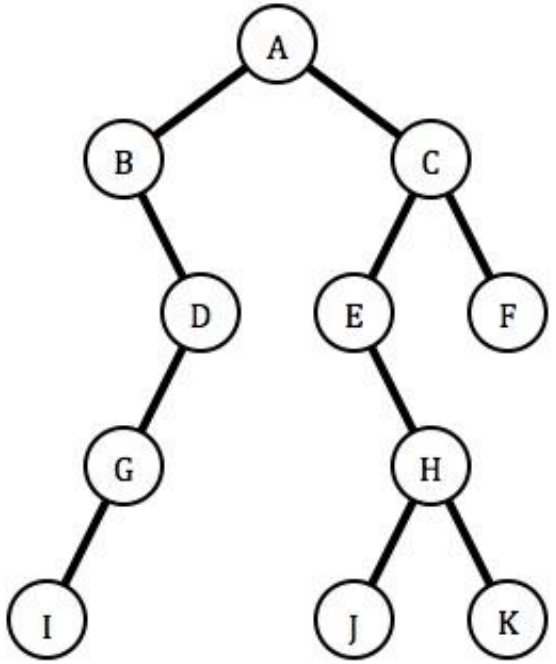
- Si no está vacío, es un conjunto de uno o más nodos, tal que existe un nodo especial llamado raíz (r) y donde los restantes nodos están separados en k≥0 conjuntos disjuntos, cada uno de los cuales es a su vez un árbol (subárboles) A_1, A_2, ..., A_k. Para cada subárbol, su raíz está conectada a la raíz r por medio de un arco. (es una definición recursiva).
- El árbol puede ser vacío.
- Entonces un árbol no vacío tiene una raiz, y un conjunto de árboles.



Árbol y subárboles







Terminología

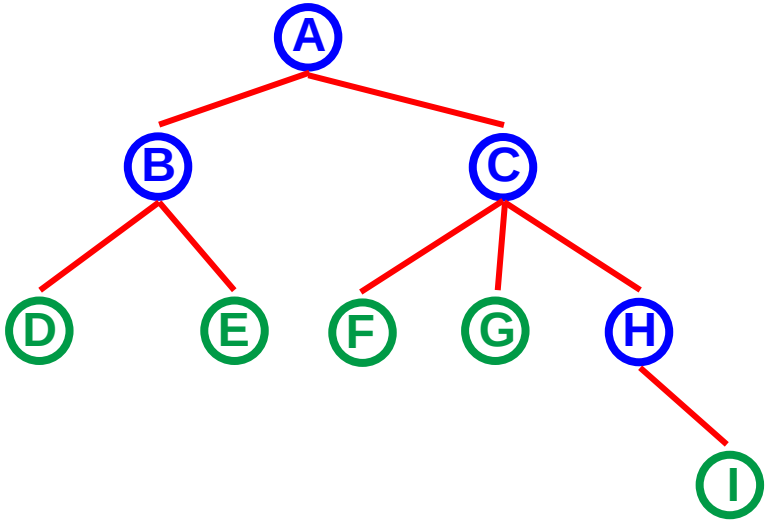
- *Hijo* y *Padre*
  - Cada nodo, excepto la raíz, tiene **un** padre.
  - Cada nodo puede tener un número arbitrario de hijos (dependerá del tipo de árbol).
- *Hojas (ó nodos externos)*
  - Nodos sin hijos.
- *Vecino o hermano*
  - Nodos con el mismo padre.

Terminología

- *Camino*
  - Conjunto de arcos desde un nodo padre a cualquier nodo de alguno de sus subarboles.
- *Ancestros* y *descendientes*
  - Nodos que anteceden en un camino y nodos que suceden.
- *Longitud de un camino*
  - Cantidad de arcos de un camino.
- *Profundidad de un nodo*
  - Longitud del **único** camino desde la raíz a ese nodo.
- *Altura de un árbol (h):*
  - La profundidad de la más lejana de sus hojas.

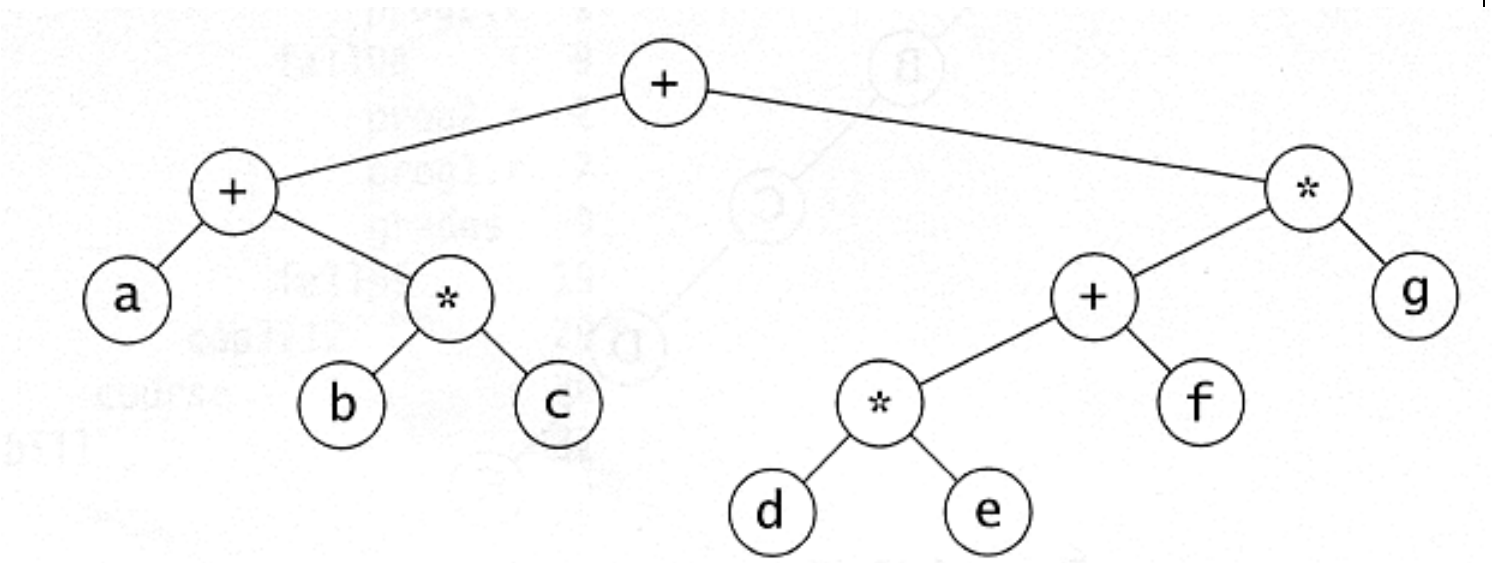
Terminología - Ejemplo

- A es el nodo raíz. Qué profundidad tendrá?
- B es el padre de D y E
- D y E son los hijos de B
- C es el hermano o vecino de B
- D, E, F, G, I son nodos externos u hojas
- A, B, C, H son nodos internos
- La profundidad (nivel) de E es 2
- La altura del árbol es 3 (= profundidad de I)
- El camino entre C e I es de longitud 2
- A y C son ancestros de G



- Propiedad: (#arcos) = (#nodos) - 1

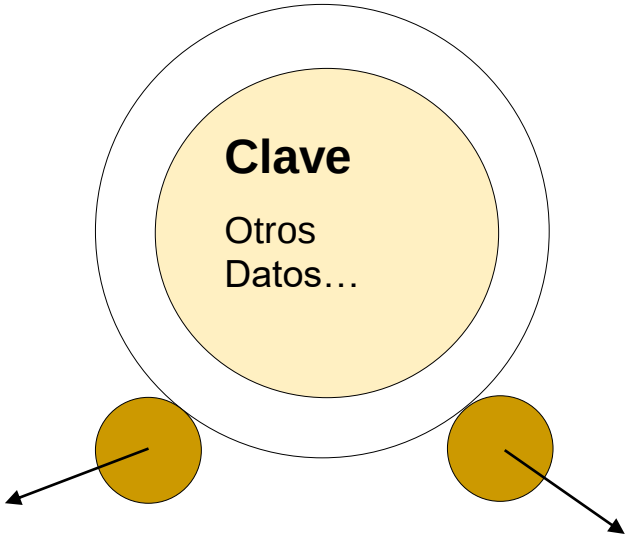
Ejemplo: Árboles de Expresiones

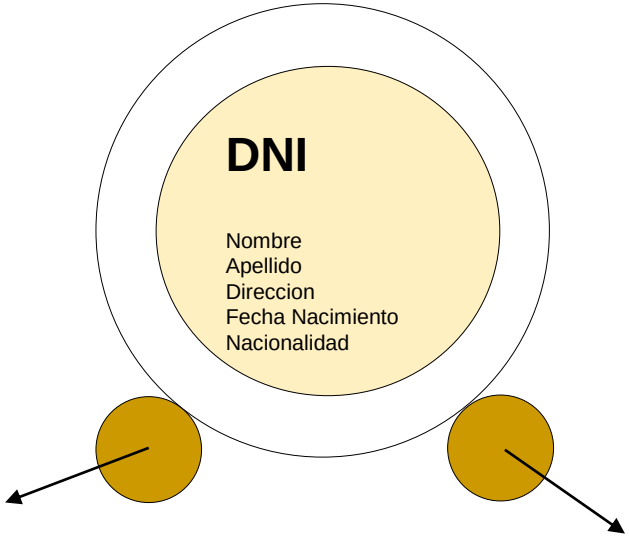


Árbol de expresión para (a+(b*c)) + (((d*e)+f)*g)

- Hojas → operandos (constantes o variables)
- Nodos internos→ operadores

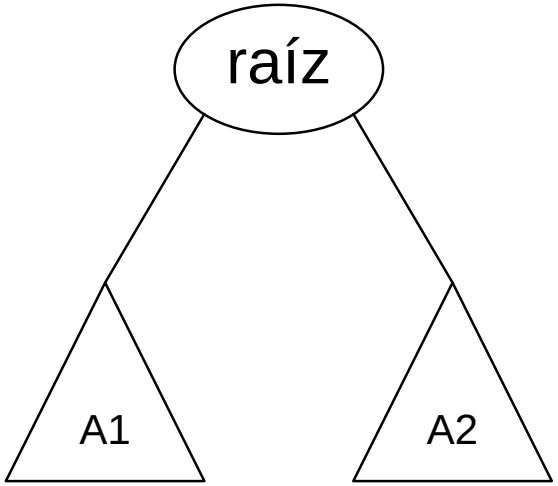
Ejemplo: Árboles con información





Árboles Binarios

- Es un árbol cuyos nodos no tienen más de 2 hijos



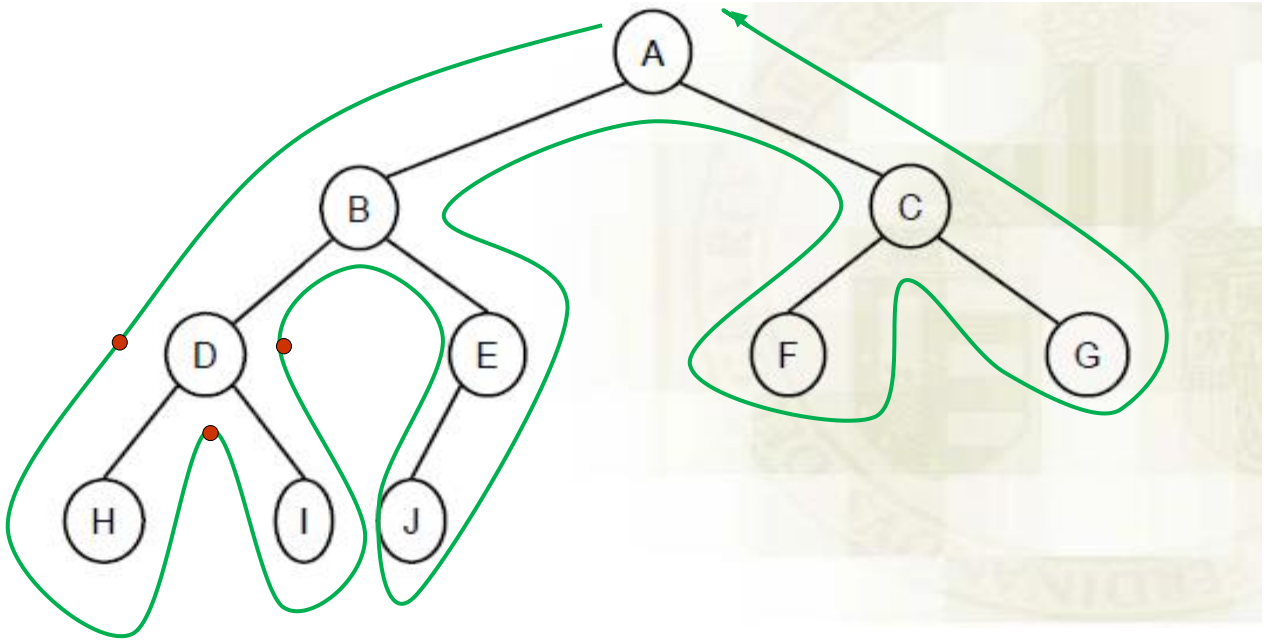

```
class Nodo
{
    int clave;
    Nodo izq, der;

    public Nodo(int c)
    {
        clave = c;
        izq = der = null;
    }
}
```

En la práctica veremos distintas formas de implementarlo.

- A1 y A2 son árboles binarios también
- La altura de un árbol binario promedio es considerablemente menor que la cantidad de nodos N (si fuera N a qué estructura se asemejaría ?).

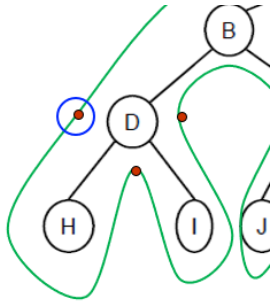
Recorridos de Arboles Binarios

- El recorrido más genérico de un árbol binario visita cada nodo tres veces (desde la izquierda, desde abajo y desde la derecha). Los puntos rojos indican estas visitas para el nodo D.
- Se denomina recorrido de Euler.

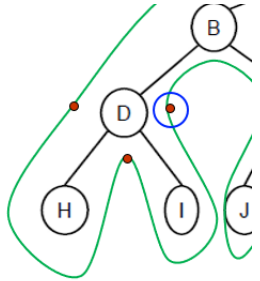


Recorridos de Árboles Binarios

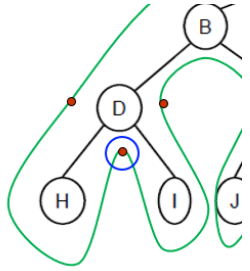
- Usado para obtener los contenidos de los nodos en un orden determinado
- Son casos especiales del recorrido de Euler.
- Recorrido Pre-orden
  - Imprimir el dato de la raíz
  - Recursivamente obtener los datos del subárbol izquierdo
  - Recursivamente obtener los datos del subárbol derecho



- Recorrido Post-orden
  - Recursivamente obtener los datos del subárbol izquierdo
  - Recursivamente obtener los datos del subárbol derecho
  - Imprimir el dato de la raíz



- Recorrido En-orden
  - Recursivamente obtener los datos del subárbol izquierdo
  - Imprimir el dato de la raíz
  - Recursivamente obtener los datos del subárbol derecho



Recorridos de Árboles Binarios

```
private void imprimirPreOrden(Nodo nodo)
   {
      if (nodo == null)
         return;

      System.out.print(nodo.clave + " ");

      imprimirPreOrden(nodo.izq);

      imprimirPreorden(nodo.der);
}
```

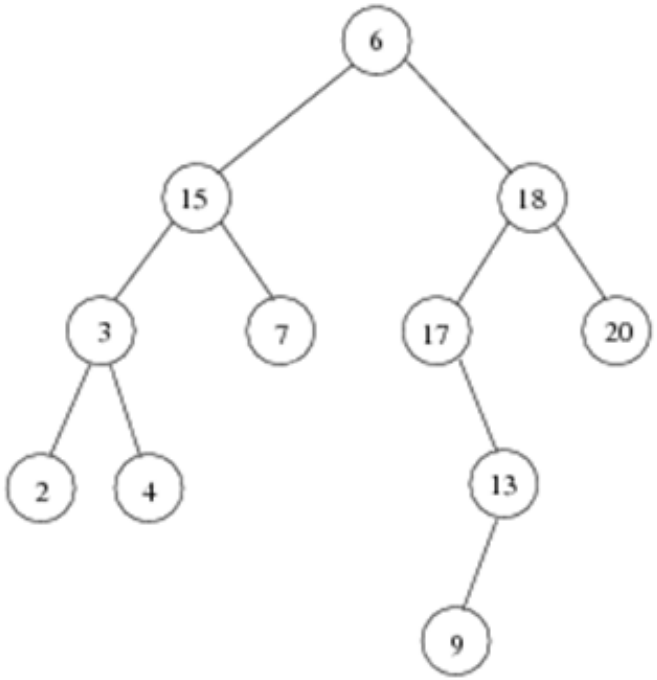
```
private void imprimirEnOrden(Nodo nodo)
   {
      if (nodo == null)
         return;

      imprimirEnOrden(nodo.izq);

      System.out.print(nodo.clave + " ");

      imprimirEnOrden(nodo.der);
}
```

```
private void imprimirPosOrden(Nodo nodo)
   {
      if (nodo == null)
         return;

      imprimirPosOrden(nodo.izq);

      imprimirPosOrden(nodo.der);

      System.out.print(nodo.clave + " ");

 }
```

O(n)

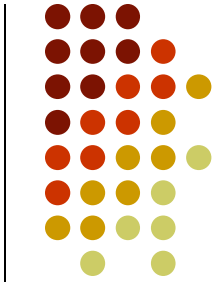
Ya que haremos un acceso (llamada recursiva) por cada nodo del árbol.

Recorridos de los árboles



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

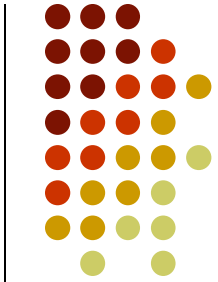
Operaciones en árboles binarios



Sea **p** el puntero (o referencia) a un nodo.

- **crear(x)**. Crea un nuevo árbol binario formado por un solo nodo con clave *x* y retorna un puntero a ese nodo.
- **info(p)**. Retorna el valor del nodo.
- **izq(p)**. Retorna un puntero al hijo izquierdo del nodo.
- **der(p)**. Retorna un puntero al hijo derecho del nodo.
- **buscar(p_raíz, x).** Busca el nodo que tiene clave x.
- **insertar(p_raíz, x).** Inserta un nodo con clave x en el árbol.
- **borrar(p_raíz, x).** Remueve el nodo que tiene clave x.

Operaciones en árboles binarios

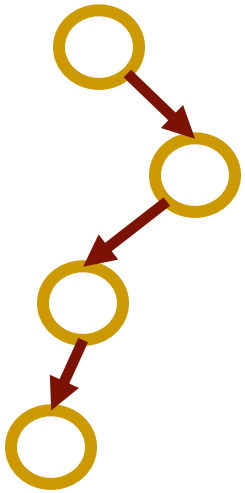


Cómo hacer estas operaciones?

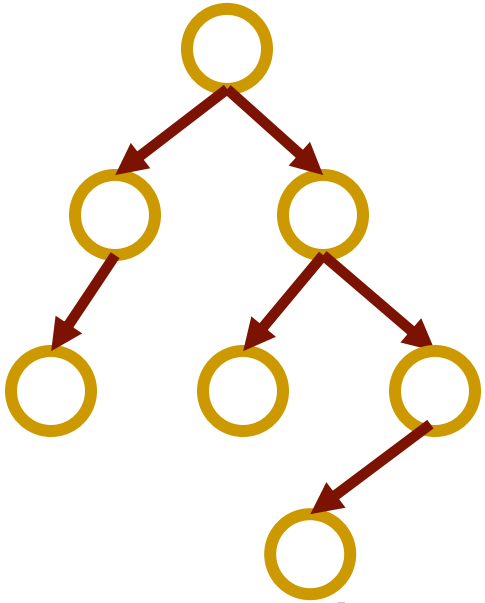
- **buscar(p_raíz, x).** Busca el nodo que tiene clave x.
- **insertar(p_raíz, x).** Inserta un nodo con clave x en el árbol, con su valor.
- **borrar(p_raíz, x).** Remueve el nodo que tiene clave x.

Estructuras Especiales de Árboles Binarios

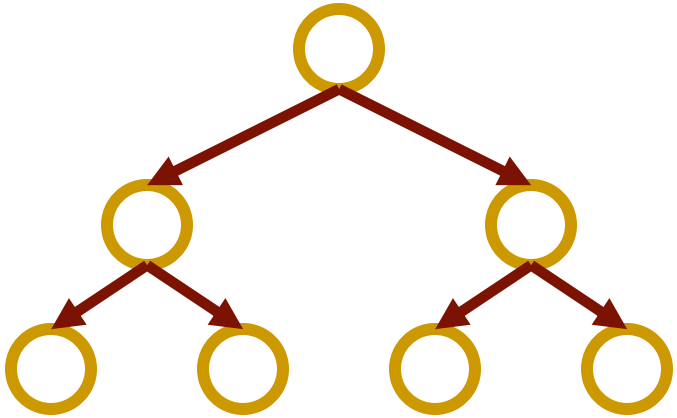
- Arboles Degenerados (enredaderas) → nodos internos con sólo un hijo.
- Balanceados → las alturas de los dos subarboles de cualquier nodo difieren a lo sumo en 1.
- Completos → todos sus nodos tienen dos hijos, excepto los del último nivel que no tienen hijos.



Enredadera
(se asemeja a una lista vinculada)



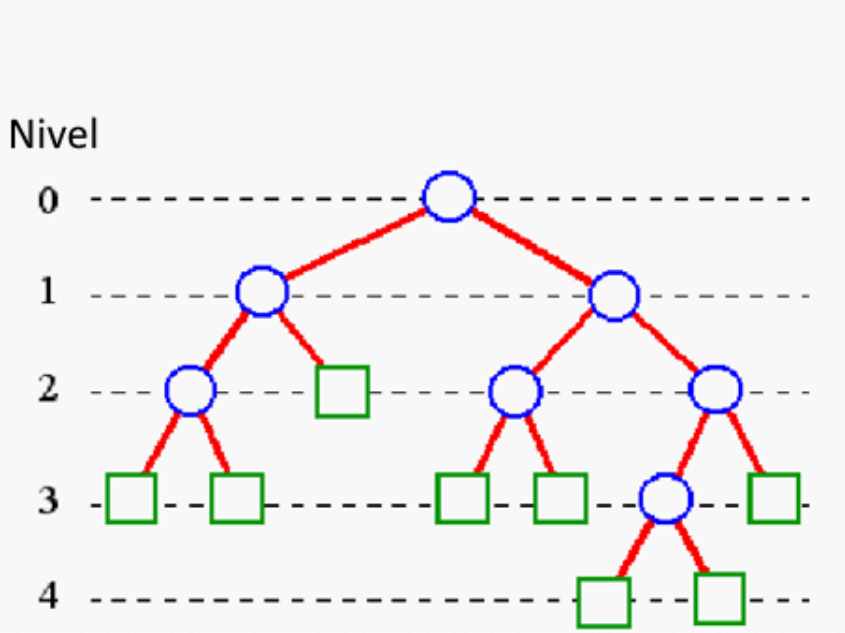
Balanceado



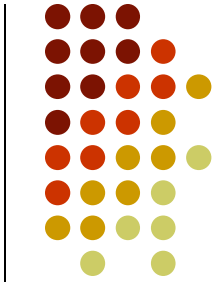
Completo

Propiedades de Arboles Binarios

- #nodos externos = #nodos internos + 1
- #nodos i-ésimo nivel $\leq 2^{i}$ donde i = 0, …, altura
- #nodos externos $\leq 2^{\text{altura}}$
- $\log_2$ (#nodos externos) $\leq$ altura $\leq$ #nodos internos



Propiedades



- Arbol Enredadera
  - Calcular h ε O(N) para N nodos
  - EL peor caso es similar a una lista lineal vinculada.

- Arboles Balanceados y Completos
  - Calcular h ε O (log_2 (N)) para N nodos.

Pero

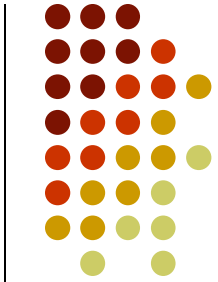
- Los recorridos por ejemplo si quiero buscar un elemento, es similar a explorar listas ya que en el peor caso voy a tener que pasar por todos los nodos O(n)

  … donde está entonces la ventaja respecto de utilizar listas vinculadas???

Pausa de 10 minutos…

Parte II

Motivación



- El tiempo de búsqueda en las listas vinculadas (LINEAL depende de n) es elevado… sin embargo, una vez encontramos la posición el costo de agregar y borrar es O(1) o sea tiempo constante.

- Array ordenado: para las búsquedas $O(\log_2 N)$ usando búsqueda binaria .. Pero inserciones (con desplazamientos) y borrados (sin huecos) implican O(N) ….

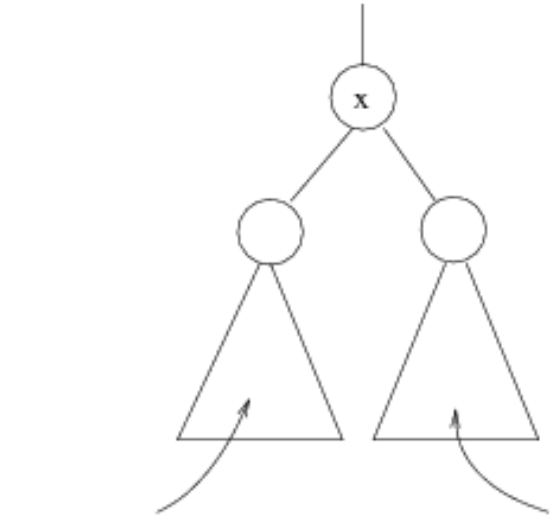
Buscamos una solución que tome lo mejor de ambas estructuras

Árboles Binarios de Búsqueda

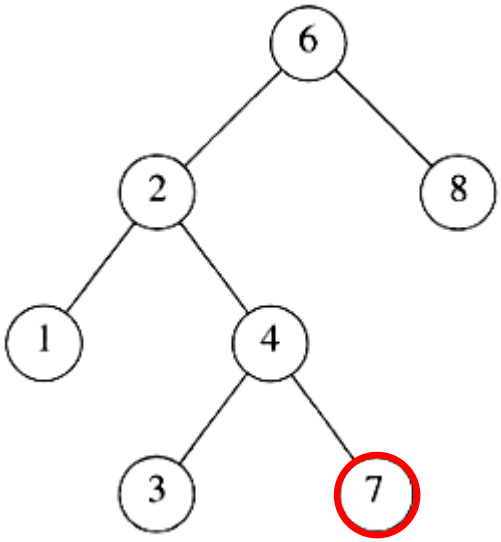
Un Árbol Binario **de Búsqueda** (ABB) es un árbol binario que tiene un orden entre los nodos, utilizando el valor de la clave de cada nodo como campo de ordenamiento (clave de bifurcación).

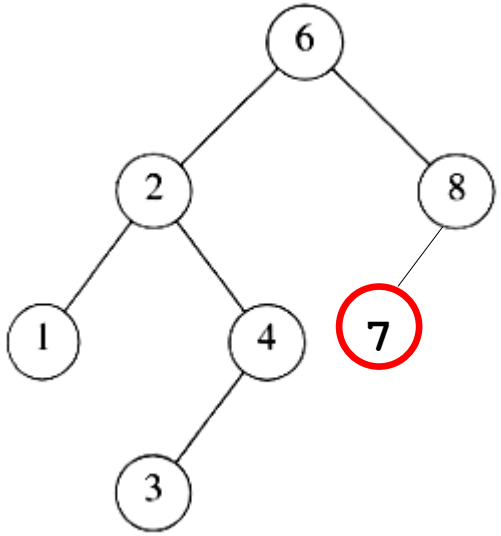
La propiedad de los ABB:

- Para cada nodo del ABB, todas las claves del subárbol izquierdo son menores que la de la raíz, y todas las del derecho son mayores.



Árboles Binarios de Búsqueda





Es un árbol binario

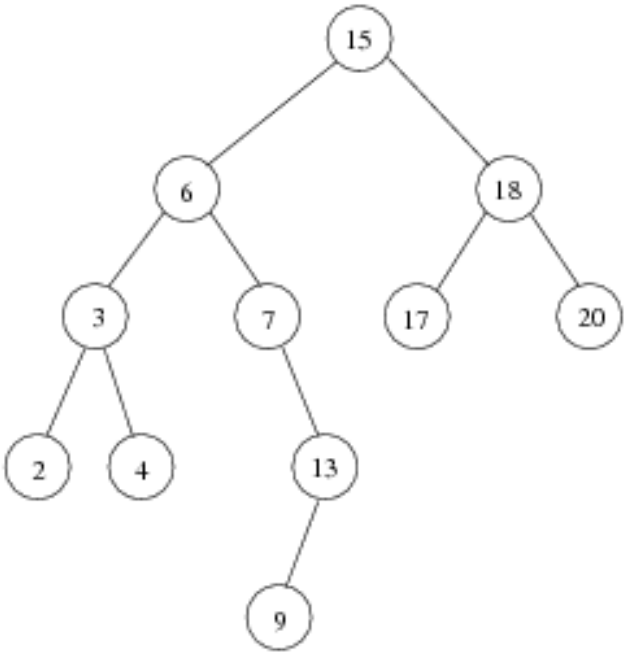
Pero no un ABB.

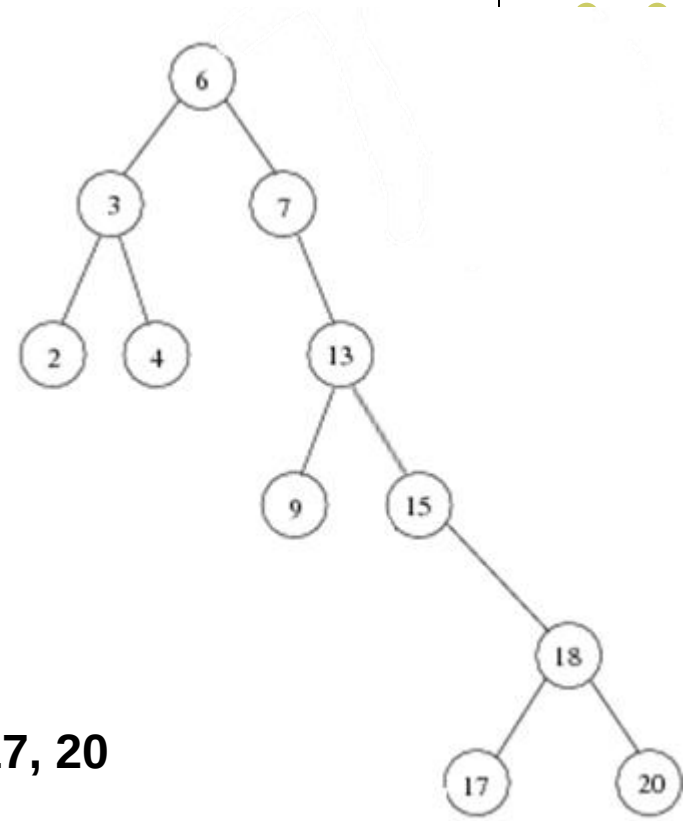
Es un ABB

Al contenido del nodo que sirve para establecer la relación de orden lo llamaremos CLAVE.

Las claves deben ser únicas en el árbol.

Recorridos en ABB





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

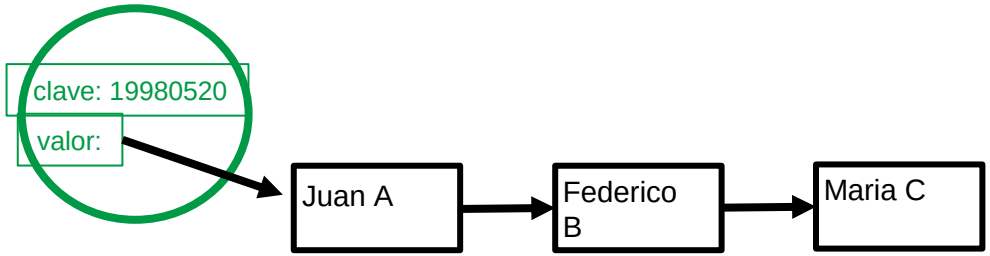
¿Si tuviera el mismo conjunto de claves en ABB diferente, cambian los recorridos?

Claves y valores

- Los árboles binarios de búsqueda están ordenados según la clave (de bifurcación) contenida en cada nodo del árbol.
- La clave se utiliza para analizar el ordenamiento del nodo y proceder en las búsquedas.

- Puede resultar útil permitir que cada nodo contenga un valor (información) además de la clave.
- El valor es un dato adicional del nodo, asociado a la clave.

- Si en nuestro dominio existieran claves repetidas, podemos asociar al nodo con dicha clave una lista de valores (llamada lista de factoreo). Por ej. Si la clave es la fecha de nacimiento de los alumnos de la facultad.



Operaciones en ABB

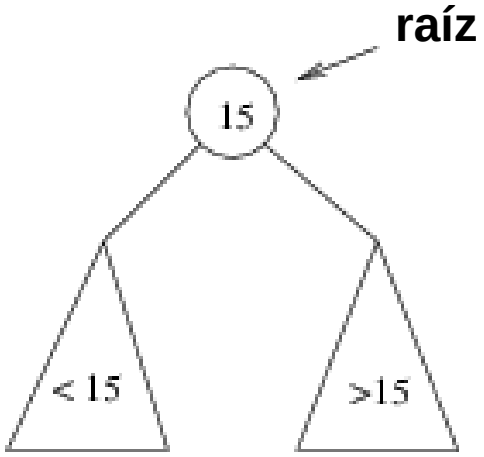
Para comparar…

Cómo cambian estas operaciones, dado el ordenamiento de árbol?

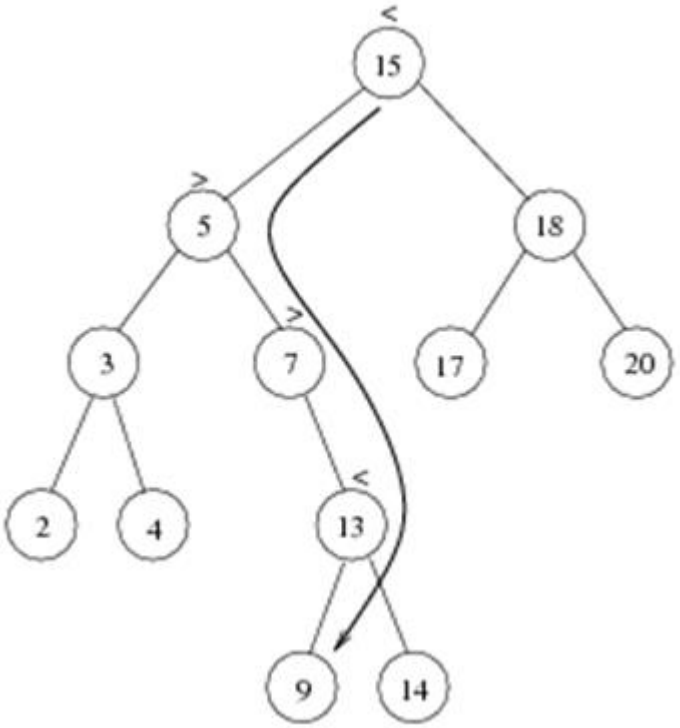
- **buscar(p_raíz, x).** Busca el nodo que tiene clave x.
- **insertar(p_raíz, x).** Inserta un nodo con clave x en el árbol, con su valor.
- **borrar(p_raíz, x).** Remueve el nodo que tiene clave x.

Búsqueda en ABB

- Búsqueda de clave X = 15 → se visita la raíz y termina.
- Búsqueda de una clave Y < 15 → buscar en el subárbol izquierdo.
- Búsqueda de una clave Z > 15 → buscar en el subárbol derecho.



Búsqueda en ABB



Buscar la clave 9:

Comparar 9 con 15 → avanzar hacia el subárbol izquierdo

Comparar 9 con 5 → avanzar hacia el subárbol derecho

Comparar 9 con 7 → avanzar hacia el subárbol derecho

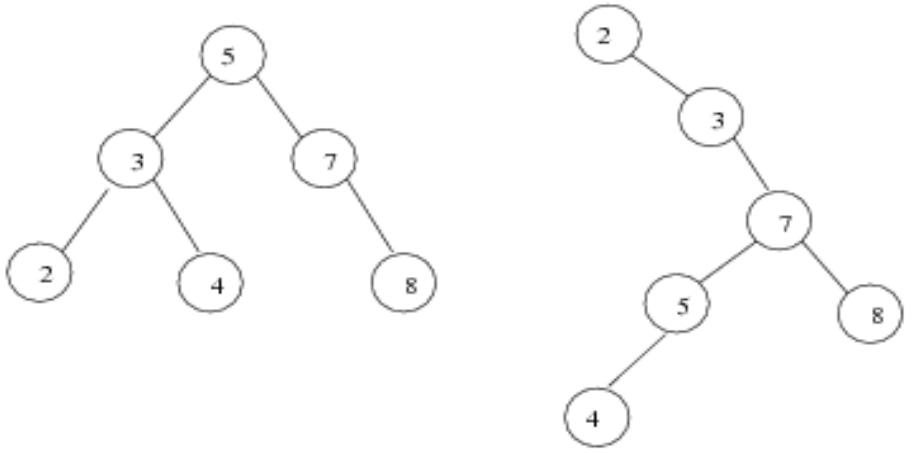
Comparar 9 con 13 → avanzar hacia el subárbol izquierdo

Comparar 9 con 9 → éxito!!

Y si hubiera que buscar la clave 6 ???

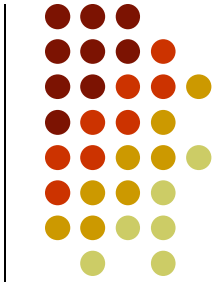
Nodo mínimo y máximo

- Cómo obtener el nodo con el menor valor y cómo el de mayor valor?
  - → Nodo más izquierdo (NMI): trato de ir a la izquierda hasta que no se pueda más.
  - → Nodo más derecho (NMD): trato de ir a la derecha hasta que no se pueda más.
- Complejidad temporal ε O(h)



NMI=2 y NMD=8 en ambos casos

Construcción de un ABB

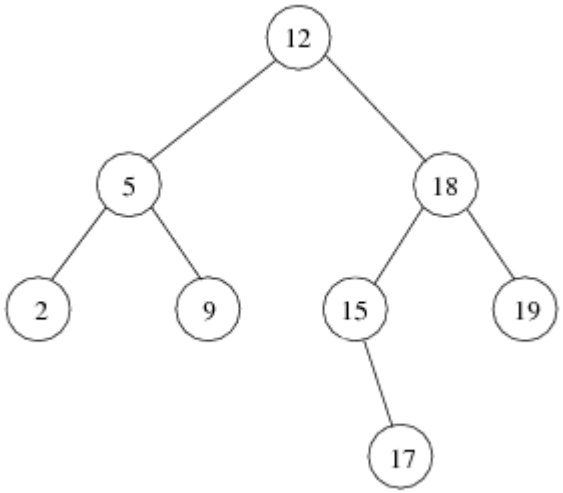


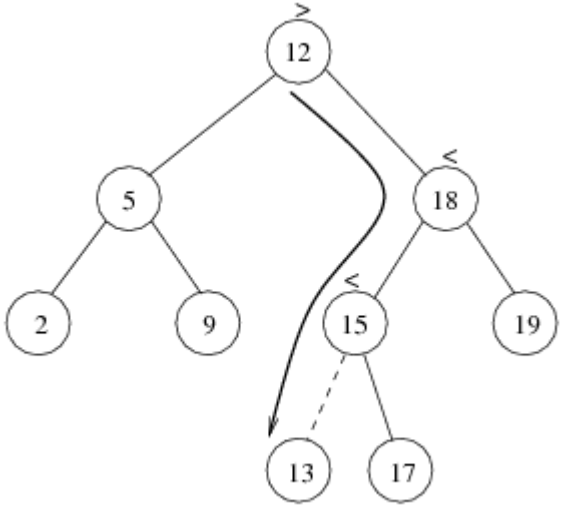
- Inserciones
- Borrados

- IMPORTANTE: se debe mantener la propiedad de árbol binario de búsqueda.
  - Valores menores a la izquierda
  - Valores mayores a la derecha

Inserción en un ABB

- Buscar el lugar para insertar X
- Si se encuentra X, termina … o … se inserta nuevamente??
- Si no se encuentra, insertar X en el último lugar alcanzado en la búsqueda
- Por ej. insertamos el 13:

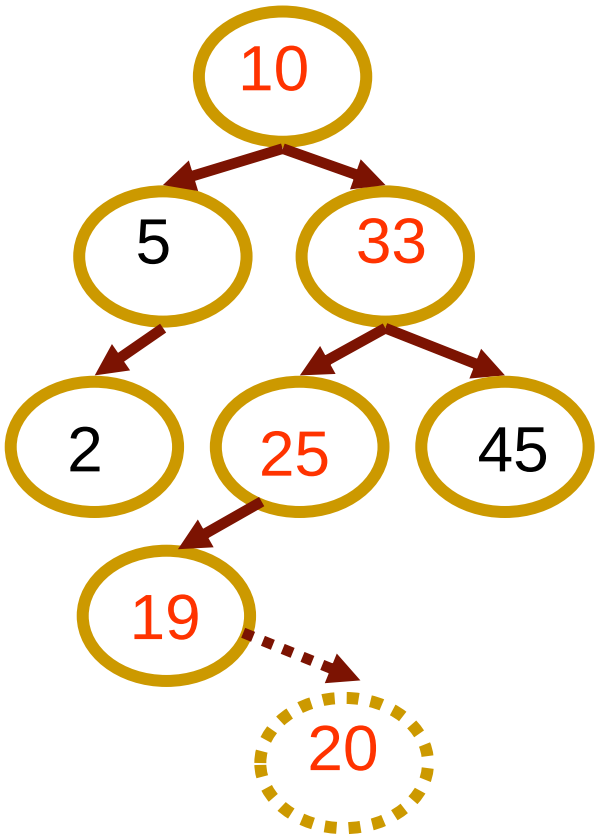




- Complejidad temporal ε O(h), donde h es la altura.

Inserción (ej.)

- Insertar (20)



10 < 20, derecha

33 > 20, izquierda

25 > 20, izquierda

19 < 20, derecha

Insertar 20 a la derecha

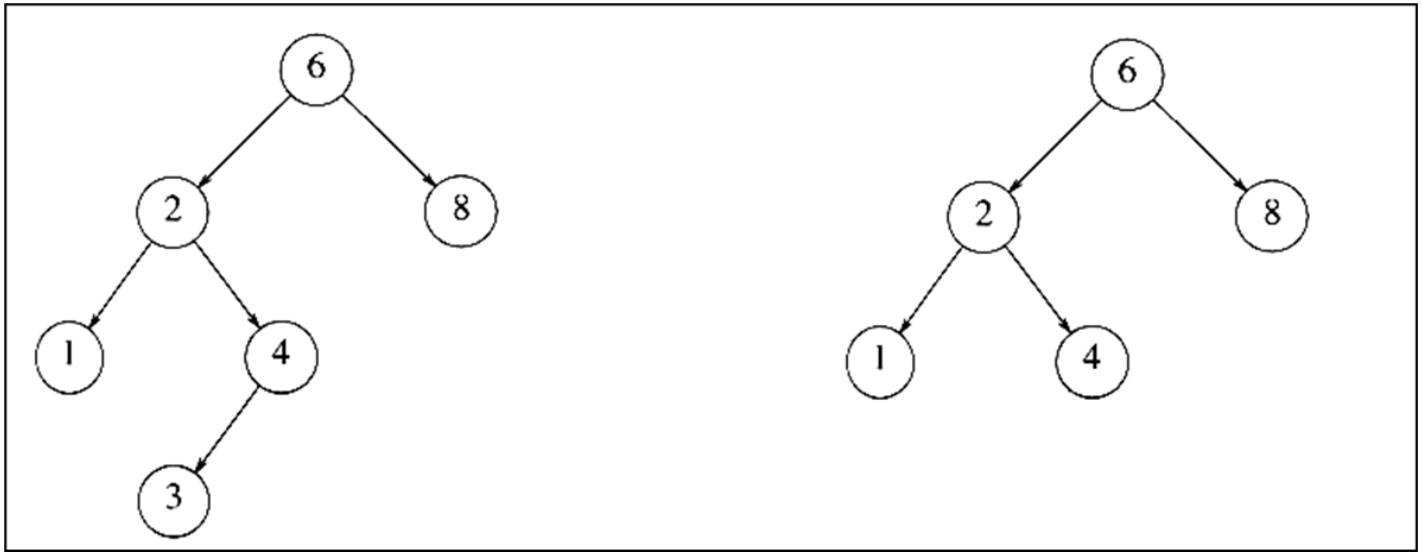
Borrado en un ABB

Podemos encontrar 3 casos:

Caso 1) El nodo es una hoja:

- Borrar el nodo.

Por ej borrar el 3:

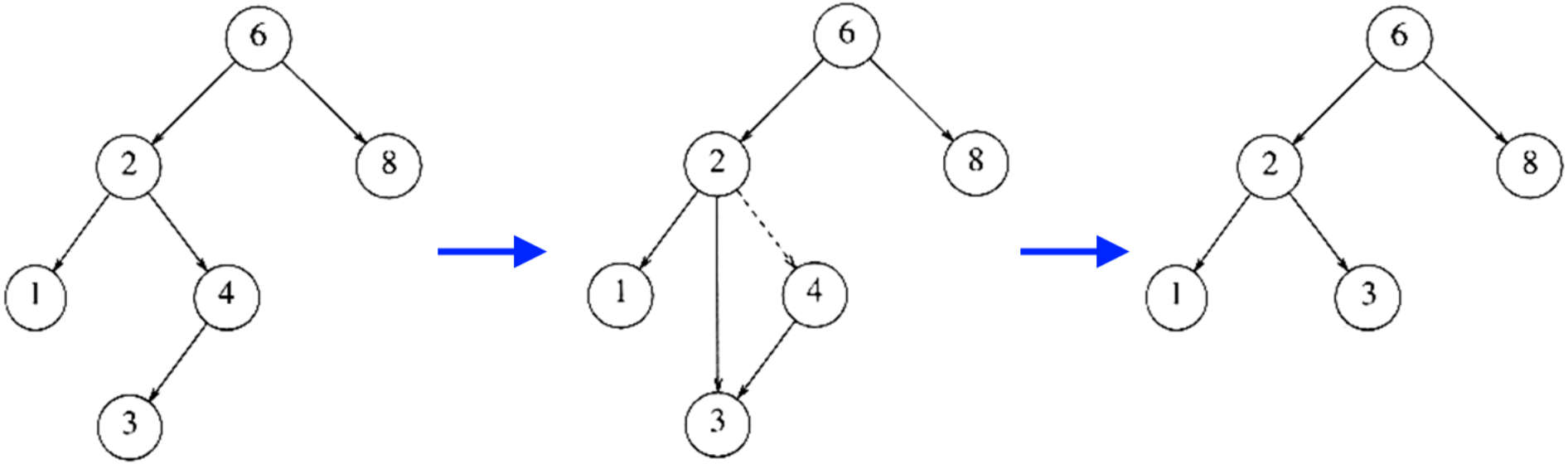


Borrado en un ABB

Caso 2) El nodo tiene un solo hijo :

- Acomodar el puntero para ignorar el nodo borrado y apuntar al hijo.

Por ej borrar el 4:



Borrado en un ABB

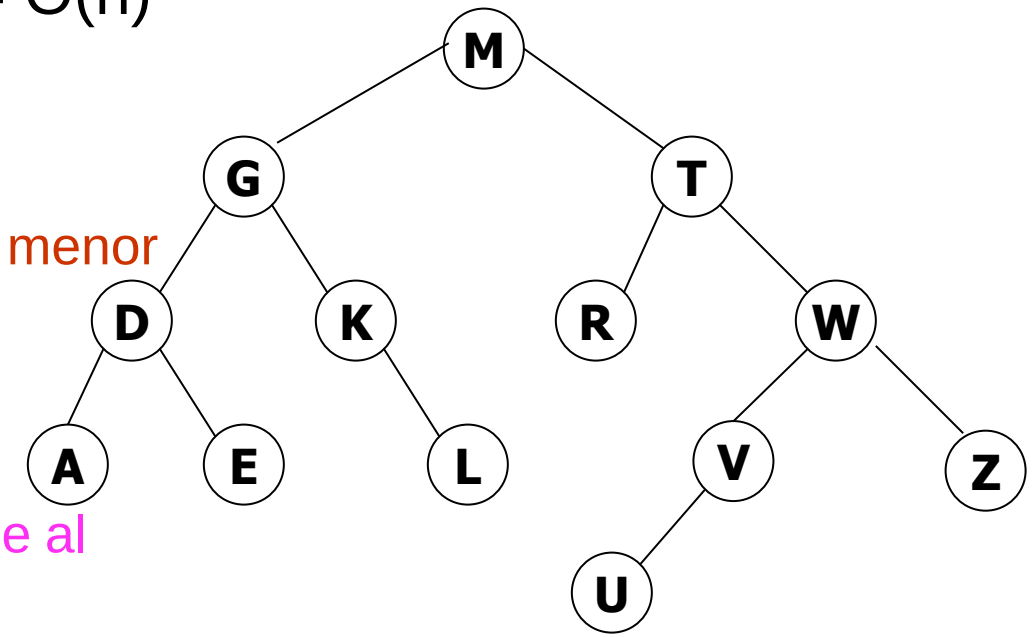
Caso 3) El nodo tiene sus 2 hijos:

- Reemplazar con el NMI del subárbol derecho del nodo a borrar.
- Borrar el NMI del subárbol derecho.
  - Se reduce a aplicar caso 1 o caso 2.
- Complejidad temporal = O(h)

Qué es ese valor ??

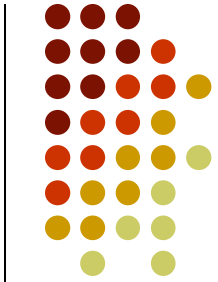
El NMI del subárbol derecho, es el menor de los mayores a esa clave.

También podría usarse el NMD del subárbol izquierdo.
En este caso ese nodo corresponde al mayor de los menores a esa clave.



Supongamos queremos borrar la G, el NMISD es K, y el NMDSI es E

Borrado en un ABB

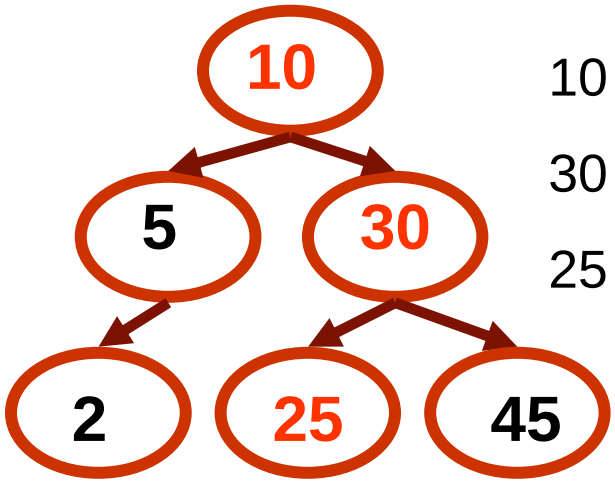


- Algoritmo: Suprimir (X, A)
  1. Buscar la clave X -> nodo N que la contiene o null. Si no es null:
  2. Si el nodo N es hoja, borrar N.
  3. Si el nodo N tiene un sólo hijo, borrar el nodo N y reacomodar puntero.
  4. Si el nodo N tiene dos hijos:
     a) Reemplazar N con el NMI del subárbol derecho a N.
     b) Suprimir (NMISD, N.der)

- Observación:
  - La operación es O(h) es lo que lleva hacer la búsqueda en el peor caso

Borrado de una hoja

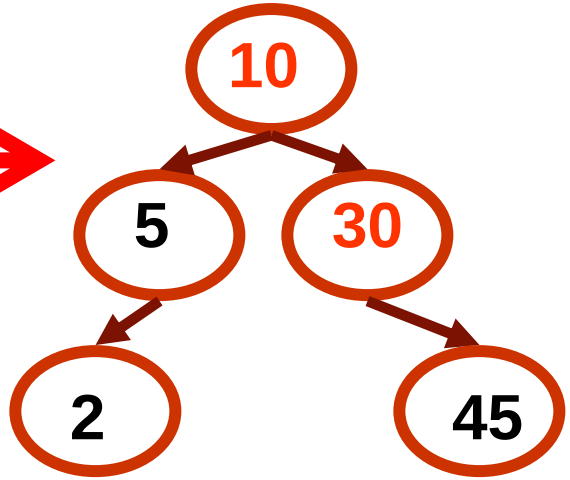
- Borrar (25)



10 < 25, derecho

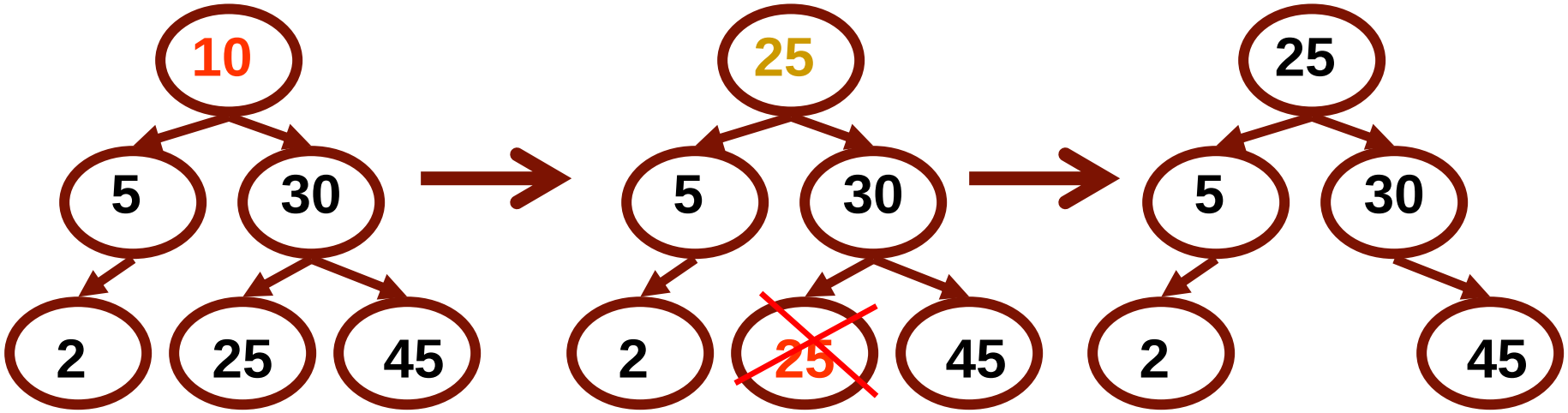
30 > 25, izquierdo

25 = 25, es hoja borrar



Borrado de un nodo interno

- Borrar (10)



Reemplazar 10 por el NMI del subárbol derecho

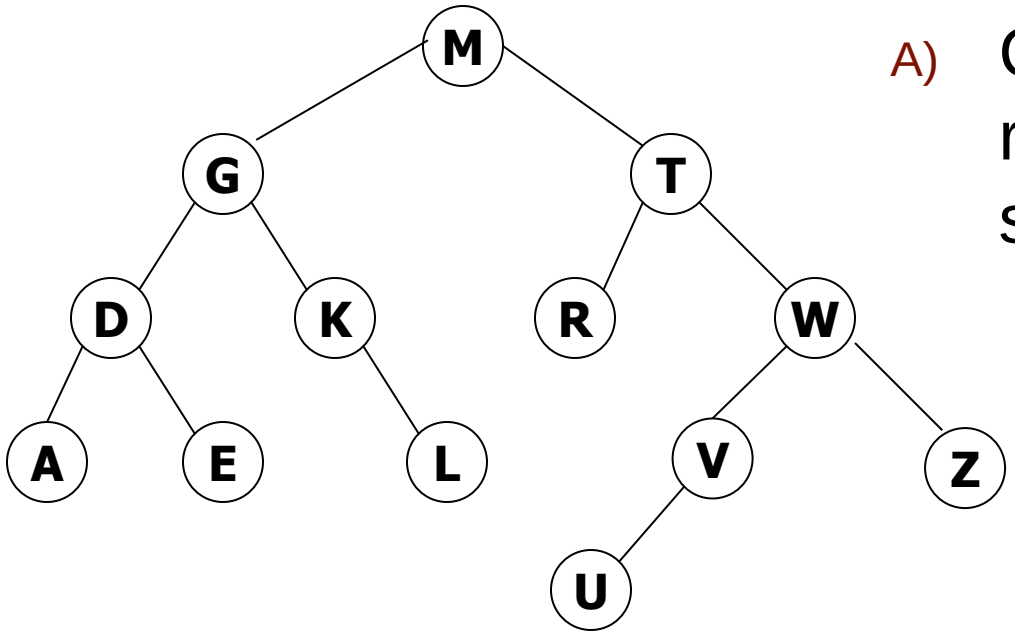
Borrar el NMI del subárbol derecho (25)

Resultado

Tarea:

- Obtener el ABB resultante de ingresar los siguientes elementos:
- A) J, N, B, D, W, E, T, Y, H, M, F
- B) B, D, E, F, H, J, N, M, T, W, Y
- C) Efectúe los diferentes recorridos del árbol obtenido en A) compare con los correspondientes en B)

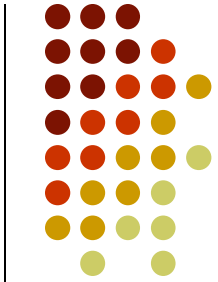
Ejercicios II



A) Cuál es el árbol resultante luego de suprimir W, T, A, y M?

Usar NMISD

Algunas propiedades de los ABBs



- Un árbol binario **completo** de altura $h \geq 0$ tiene

  $N = 2^{h+1} - 1$ nodos $\Rightarrow$

- El máximo número N de nodos que puede tener un árbol binario de altura h es $N \leq 2^{h+1} - 1$ $\Rightarrow$

- La altura mínima en un árbol binario de N nodos es $\lfloor \log_2(N+1) \rfloor - 1 \leq h$

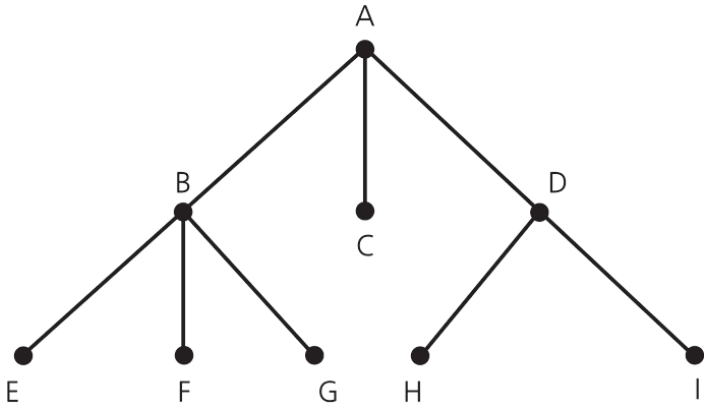
- La altura de un ABB particular depende del orden en el cual se producen las inserciones y borrados → **esto define la estructura resultante.**

  Mejor Caso → completo, Peor Caso→ enredadera

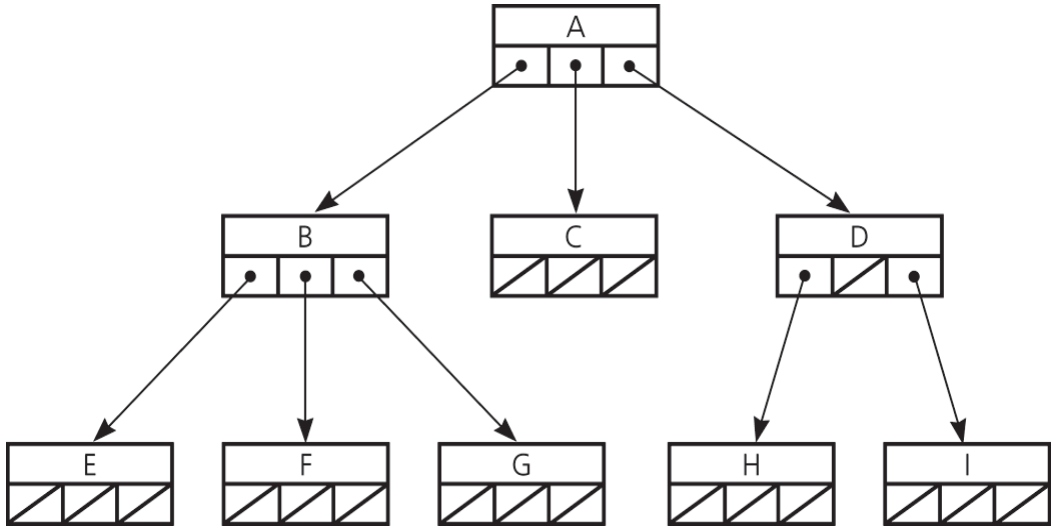
  ¿Cómo atacar este problema ? → Técnicas de Balanceo (ej.: AVL).

Árboles n-arios en general

- Es una generalización de un árbol binario cuyos nodos pueden tener hasta n hijos.



Un árbol ternario (3-ario)



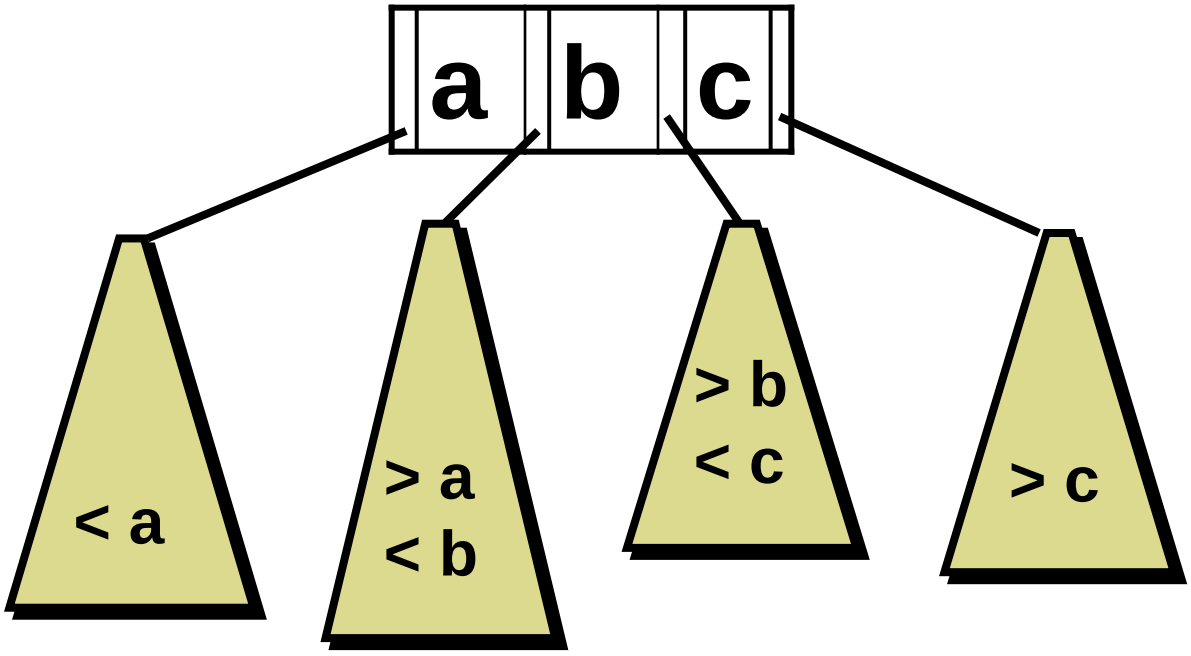
Posible Implementación => array en cada nodo

Otra? Una lista vinculada de hijos en cada nodo

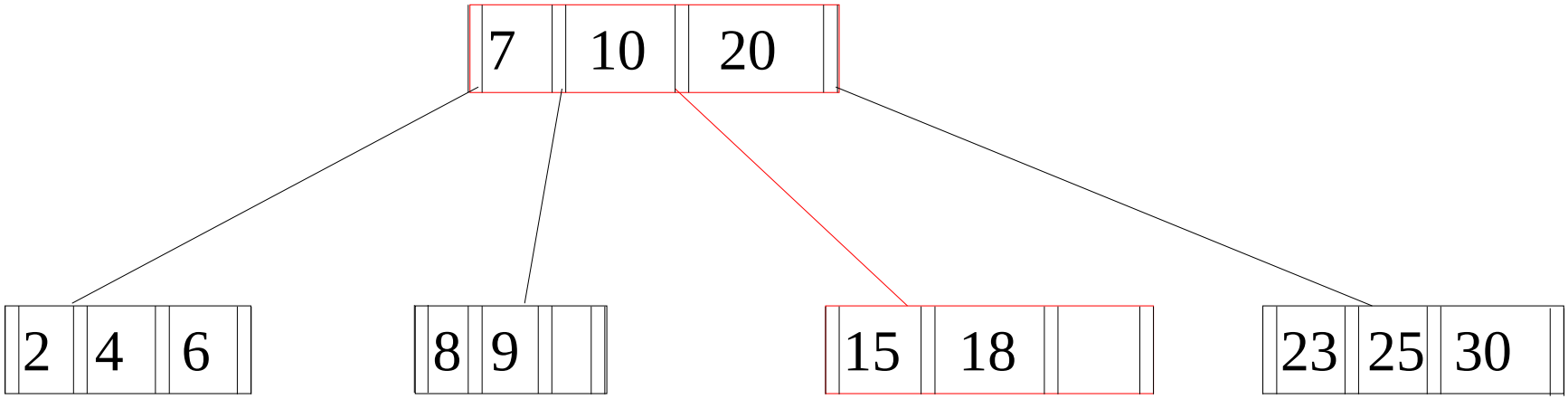
Árbol n-ario de Búsqueda

En cada nodo hay **n-1** claves y **n** punteros a nodos hijos.

Por ejemplo para n=4:



Árbol n-ario de Búsqueda



Qué se busca con esto?
Menos niveles (menos altura h para la misma cantidad de elementos N) → Menos accesos → Más eficiente ya que en el peor caso las operaciones eran O(h)

Por ejemplo familia de árboles B: son árboles n-arios de búsqueda y balanceados.