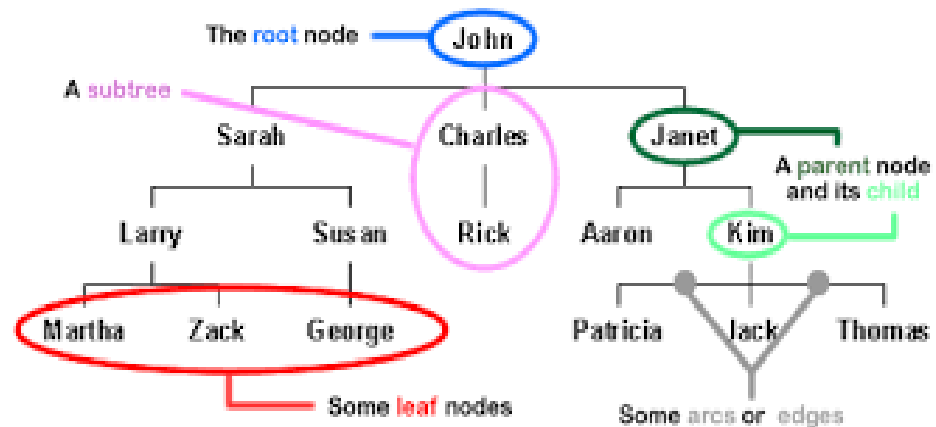


# Tipos Abstratos de Datos (TADs)

Árbol

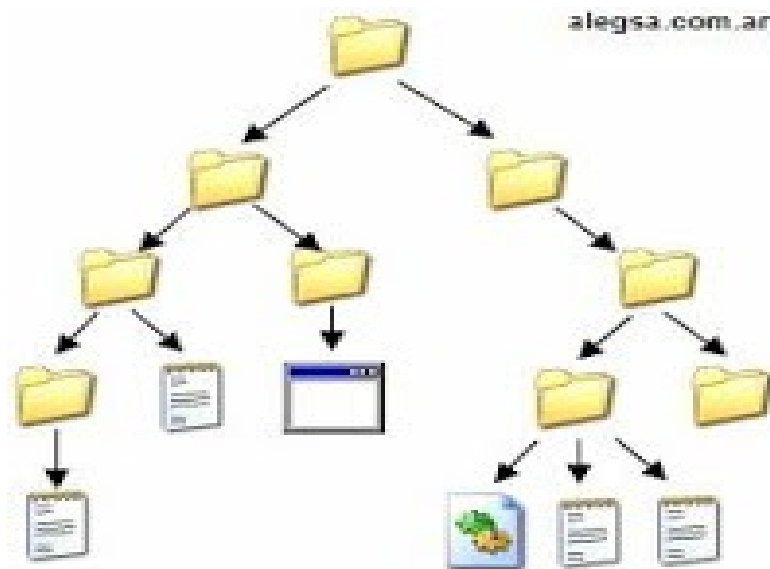
# Árbol

- Un árbol es una estructura de datos no lineal, que representa relaciones de jerarquía, con niveles.



# Árbol

- Un árbol es una estructura de datos no lineal, que representa relaciones de jerarquía, con niveles.



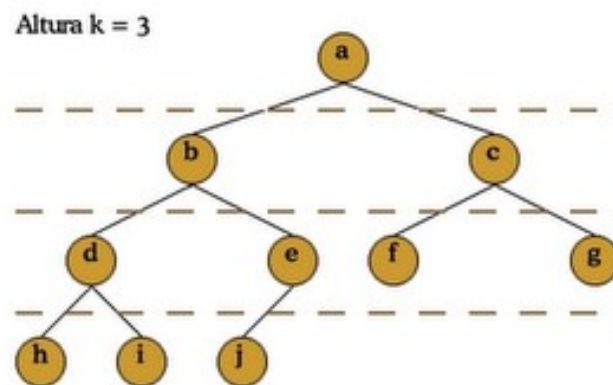
# Árbol

- Un árbol es una estructura de datos no lineal, que representa relaciones de jerarquía, con niveles.

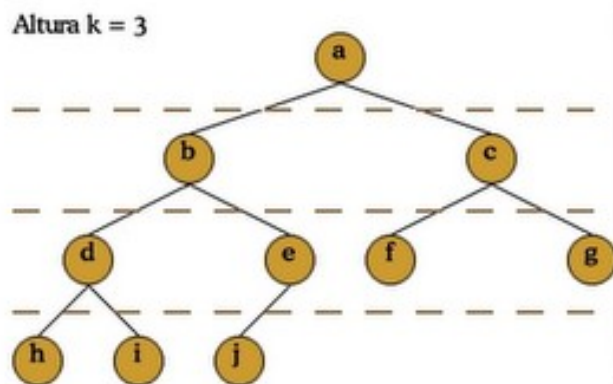
```
Command
1 /sbin/init splash
5 | /sbin/mount.ntfs /dev/sdc1 /media/monica/Elements -o rw,nodev,nosuid,uid=1001,gid=1001,uhelper=udisks2
2 | /usr/sbin/cups-browsed
9 |   | /usr/sbin/cups-browsed
9 |   | /usr/sbin/cups-browsed
3 | /usr/sbin/cupsd -l
4 | /usr/lib/fwupd/fwupd
9 |   | /usr/lib/fwupd/fwupd
9 |   | /usr/lib/fwupd/fwupd
9 |   | /usr/lib/fwupd/fwupd
8 |   | /usr/lib/fwupd/fwupd
8 | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
7 |   | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
2 |   | /opt/google/chrome/chrome
5 |   | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
6 |   | /opt/google/chrome/chrome
9 |   | /opt/google/chrome/chrome
4 |   | /opt/google/chrome/chrome --type=utility --field-trial-handle=13348535474094317396,10417735273720030990,131072 --lang=es-419 --service-
9 |   | /opt/google/chrome/chrome --type=utility --field-trial-handle=13348535474094317396,10417735273720030990,131072 --lang=es-419 --servi
9 |   | /opt/google/chrome/chrome --type=utility --field-trial-handle=13348535474094317396,10417735273720030990,131072 --lang=es-419 --servi
9 |   | /opt/google/chrome/chrome --type=utility --field-trial-handle=13348535474094317396,10417735273720030990,131072 --lang=es-419 --servi
9 |   | /opt/google/chrome/chrome --type=utility --field-trial-handle=13348535474094317396,10417735273720030990,131072 --lang=es-419 --servi
```

# Árbol

- Un árbol es una estructura de datos no lineal, que representa relaciones de jerarquía, con niveles.

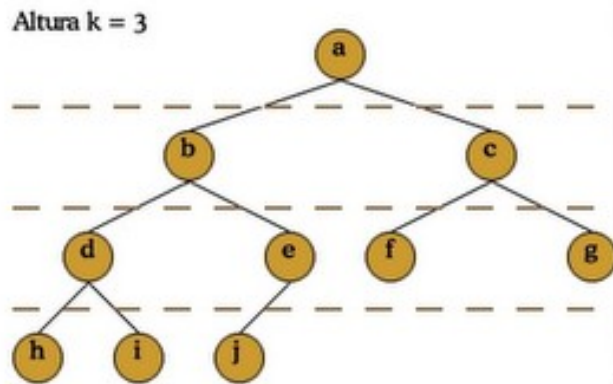


# Árbol



- Está compuesto por nodos y enlaces (dirigidos).
- Dichos nodos tienen relación: padres, hijos, nietos, hermanos.
- Todo árbol tiene un nodo raíz.
- Todo árbol tiene nodos hoja.
- La cantidad de hijos de un nodo es 0 o más de 0.
- Los árboles binarios tienen cada nodo, 0, 1 o 2 hijos.
- Un hijo es un subárbol a la vez (estructura recursiva).
- Árboles genéricos.
- Árboles binarios.
- Profundidad, nivel, altura.

# Árbol binario



- Un árbol binario es un árbol donde cada nodo puede tener a lo sumo 2 hijos
- O lo que es lo mismo, cada nodo tiene a lo sumo 2 subárboles.
- Cada hijo es un subárbol.
- Hijo izquierdo.
- Hijo derecho.
- Cuando un hijo no existe, es NULL.



# Árbol binario: árbol de expresión

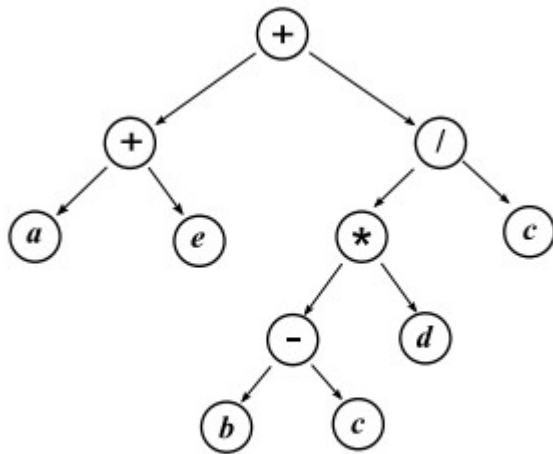


Figura 3:  $(a + e + (b - c) * d / c)$

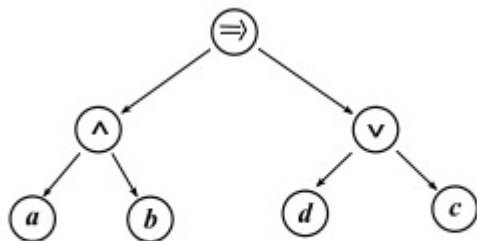


Figura 1: Árbol de expresión para  $(a \wedge b) \Rightarrow (d \vee c)$

- Un árbol de expresión es un árbol binario usado para representar y evaluar expresiones algebraicas o lógicas formadas por operadores unarios o binarios.
- Un árbol de expresión se construye a partir de los operadores simples y los operandos de alguna expresión poniendo los operandos en las hojas del árbol binario, y los operadores en los nodos internos.

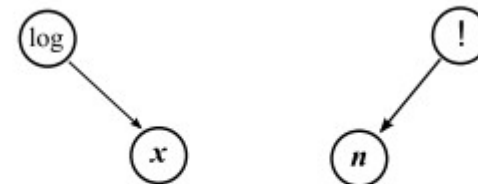
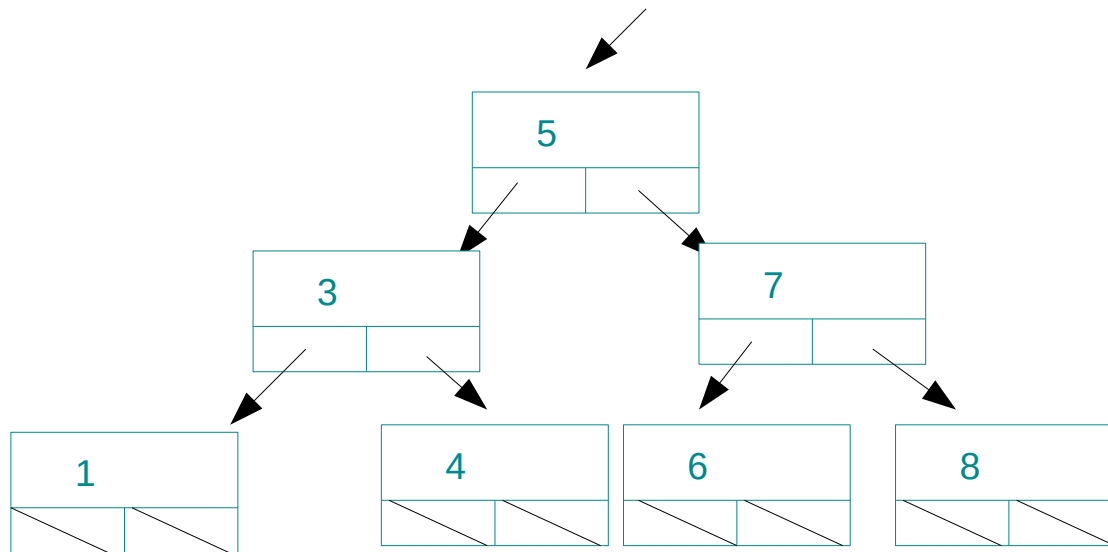
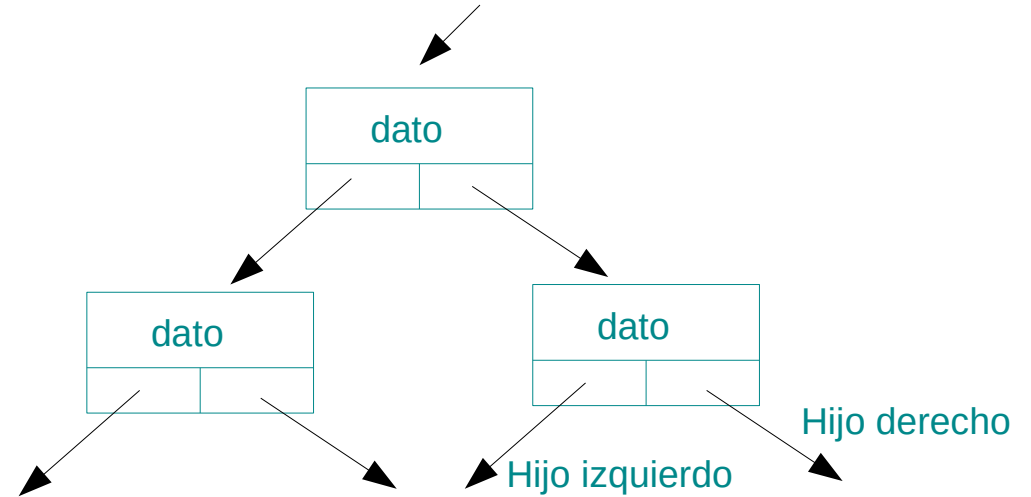
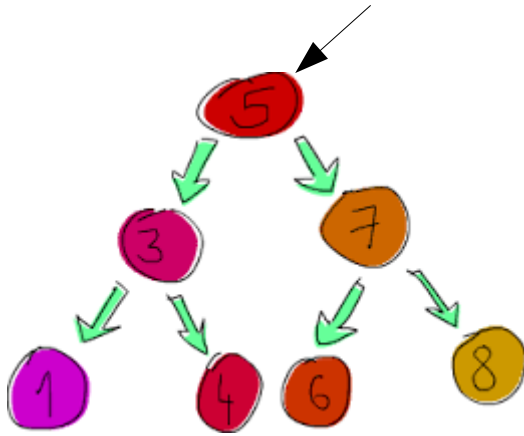
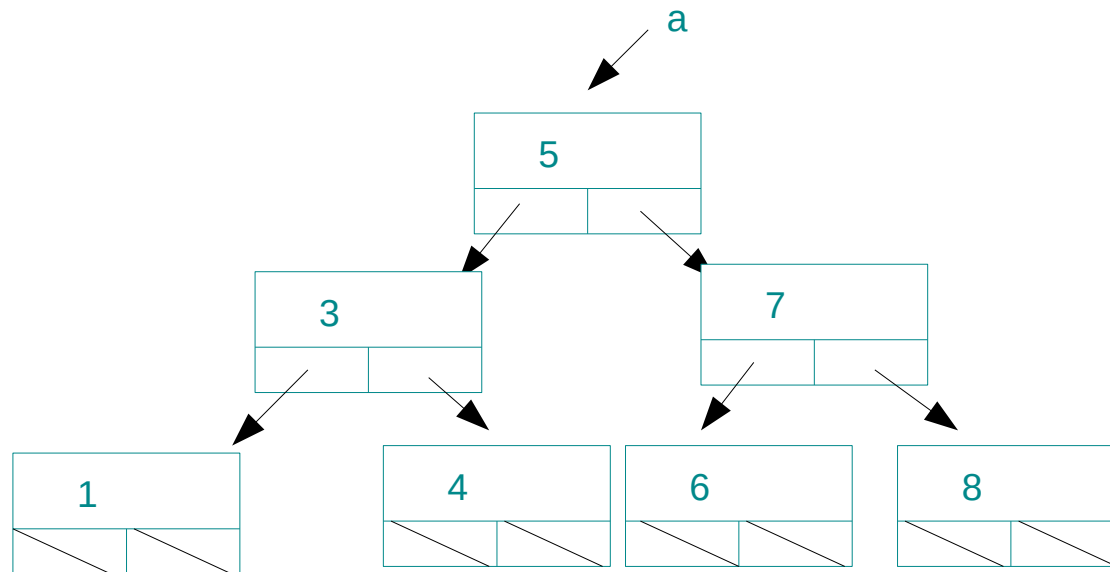


Figura 2: Árboles de expresión de  $\log x$  (izquierda) y  $n!$  (derecha)

# Árbol binario



# Árbol binario



```
5  struct Nodo
6  {
7      struct Nodo *izq;
8      Tipo_Dato dato;
9      struct Nodo *der;
10 };
11
12 typedef struct Nodo *Arbol_T;
13
```

# Árbol binario de búsqueda

## Árbol binario de búsqueda

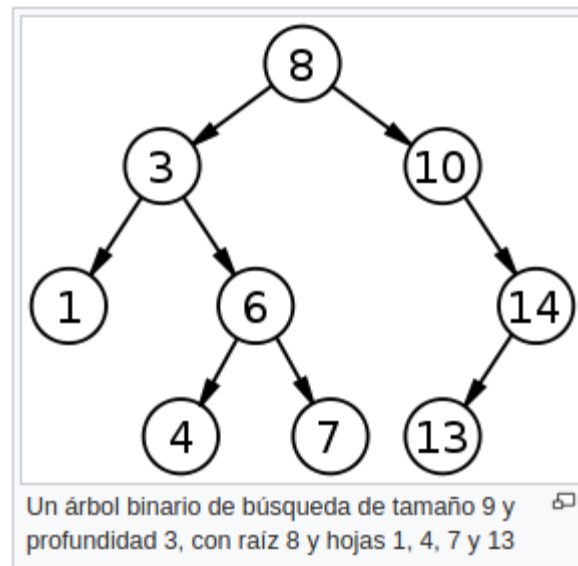
Sea  $A$  un árbol binario de raíz  $R$  e hijos izquierdo y derecho (posiblemente nulos)  $H_I$  y  $H_D$ , respectivamente.

Decimos que  $A$  es un árbol binario de búsqueda (ABB) si y solo si se satisfacen las dos condiciones al mismo tiempo:

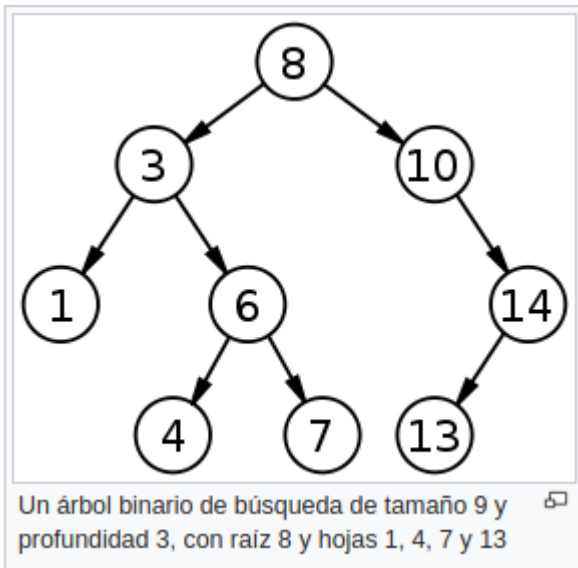
- " $H_I$  es vacío"  $\vee$  (" $R$  es mayor que todo elemento de  $H_I$ "  $\wedge$  " $H_I$  es un ABB").
- " $H_D$  es vacío"  $\vee$  (" $R$  es menor que todo elemento de  $H_D$ "  $\wedge$  " $H_D$  es un ABB").

Donde " $\wedge$ " es la conjunción lógica "y", y " $\vee$ " es la disyunción lógica "o".

Fuente: Wikipedia (se toma esta fuente dado que la definición es correcta y sencilla).

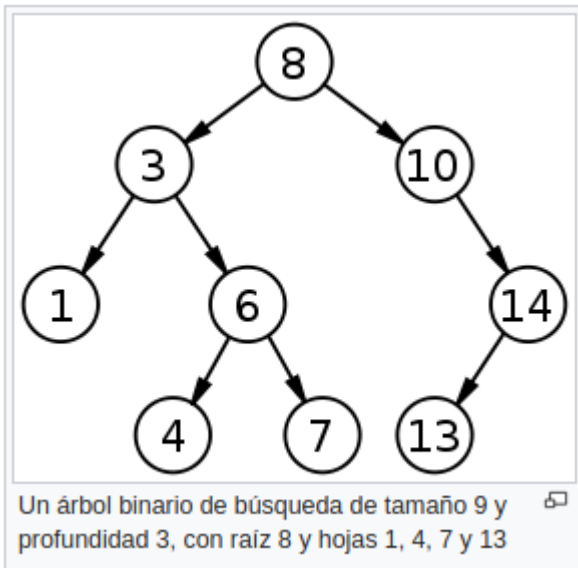


# Árbol binario de búsqueda



- ¿Por qué se llama árbol binario de búsqueda?
- ¿Para qué sirve un ABB?
- ¿Para qué son eficientes?

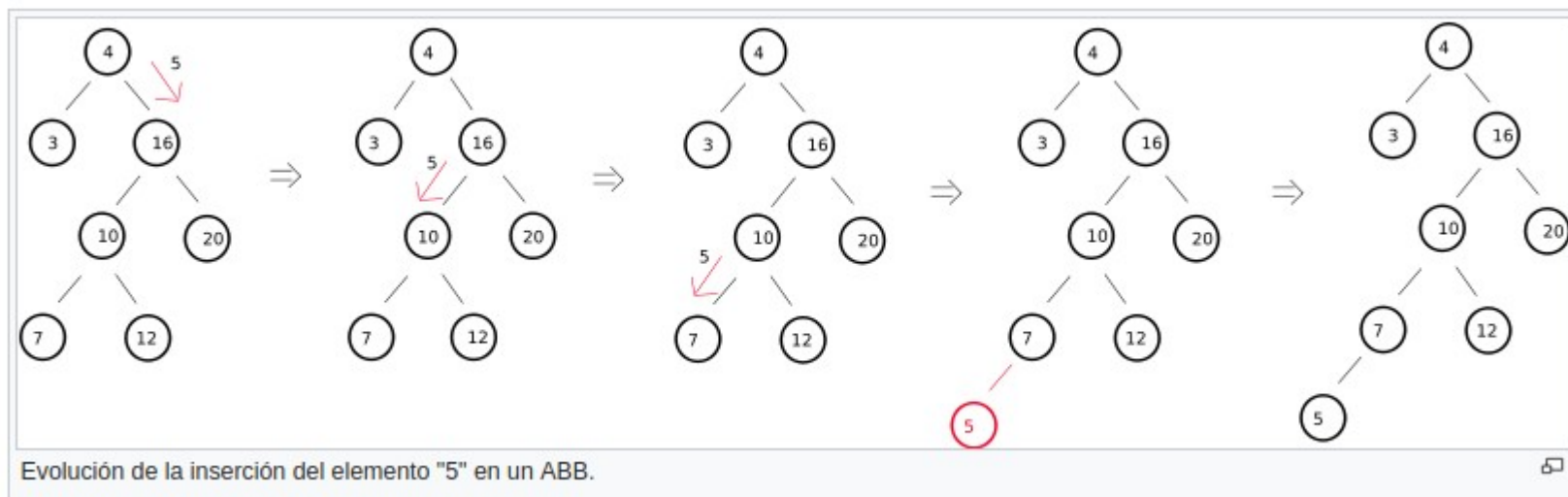
# Árbol binario de búsqueda



- ¿Por qué se llama árbol binario de búsqueda?
- TAD Árbol binario de búsqueda: operaciones:
  - Crear árbol.
  - Insertar elemento.
  - Buscar elemento.
  - Eliminar elemento.
  - Recorridos:
    - En orden.
    - Pre orden.
    - Post orden.
  - Máximo, mínimo.
  - Etc, etc.

# Árbol binario de búsqueda

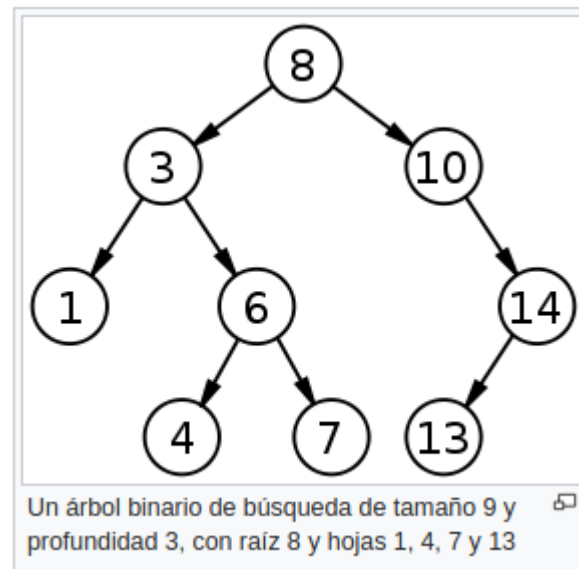
- Cada operación que modifique los datos de la colección debe garantizar que el orden del ABB se garantiza luego de ejecutar la operación.
- Esto condiciona las altas y bajas (insertar o eliminar) elementos de estos árboles.
- Por ejemplo, dado el siguiente ABB, se quiere insertar el valor 5



Fuente: Wikipedia

# Árbol binario de búsqueda

- ¿Cómo se realiza una búsqueda de un valor en un ABB?



**Se recomienda ver estos videos:**

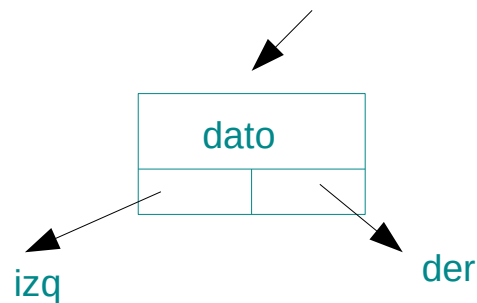
Introducción ABB: <https://www.youtube.com/watch?v=Bh61AvHAF90>

Insercion y búsqueda: <https://www.youtube.com/watch?v=DVKDQcJOqy8>



# Árbol binario de búsqueda

- Una posible implementación de la estructura es:



```
6 typedef <tipo> Tipo_Dato;  
7 struct Nodo  
8 {  
9     struct Nodo *izq;  
10    Tipo_Dato dato;  
11    struct Nodo *der;  
12 };  
13  
14 typedef struct Nodo *Arbol_T;  
15
```

Arbol\_T a;

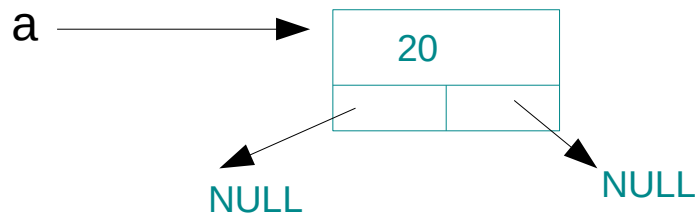
La variable a es un puntero a struct Nodo.  
Otra posibilidad es no tener el tipo Arbol\_T sino manejar directamente variables de tipo struct Nodo \*

# Árbol binario de búsqueda

- CrearArbol: crea un nodo y retorna su dirección:

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17  
18 // InsertarElemento(&a, 20);  
19 InsertarElemento(&a, 10);  
20  
21 InsertarElemento(&a, 30);  
22 InsertarElemento(&a, 50);  
23 InsertarElemento(&a, 55);  
24 InsertarElemento(&a, 75);  
25 InsertarElemento(&a, 85);  
26
```

```
6 Arbol_T CrearArbol(Tipo_Dato x)  
7 {  
8     Arbol_T a;  
9     a = (Arbol_T)malloc(sizeof(struct Nodo));  
10    a->izq = NULL;  
11    a->der = NULL;  
12    a->dato = x;  
13    return(a);  
14 }  
15
```



# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

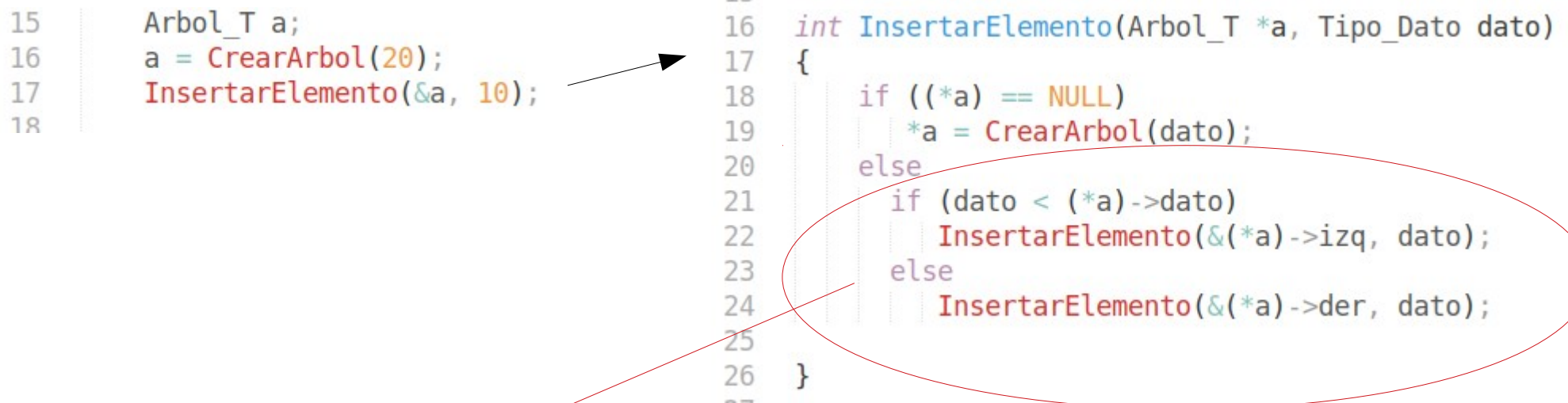
```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25  
26 }
```

# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25 }  
26
```



Búsqueda del sitio donde insertar

# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

Insertión del elemento

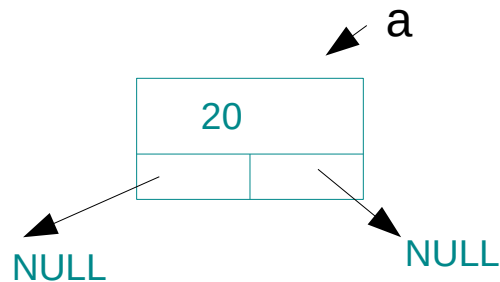
```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25  
26 }
```

# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25 }  
--
```

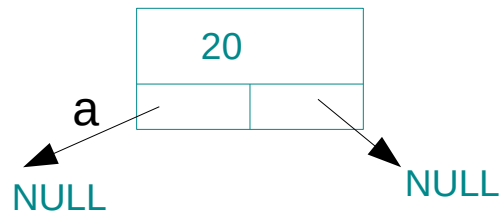


# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25  
26 }
```

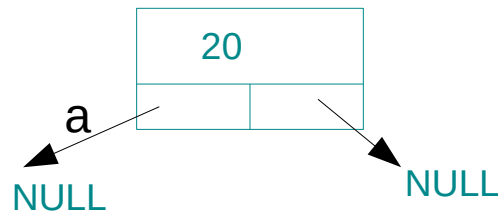


# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25 }  
--
```



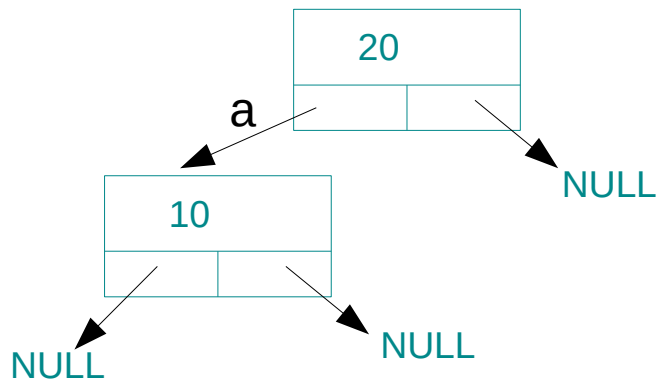


# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25 }  
--
```

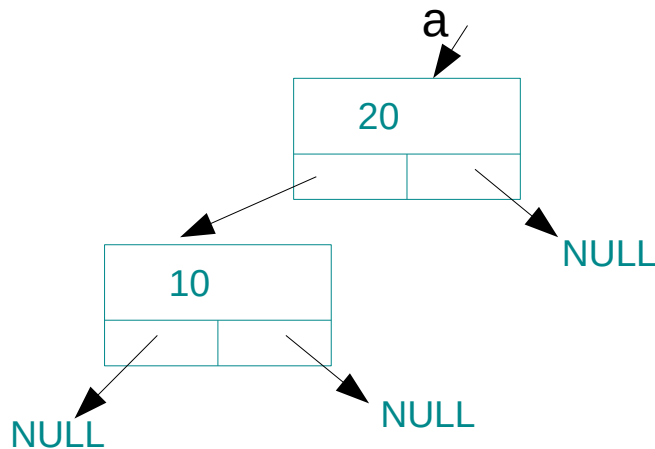


# Árbol binario de búsqueda

- InsertarElemento: 1) Busca el sitio donde insertar (ABB). 2) Inserta.

```
15 Arbol_T a;  
16 a = CrearArbol(20);  
17 InsertarElemento(&a, 10);  
18
```

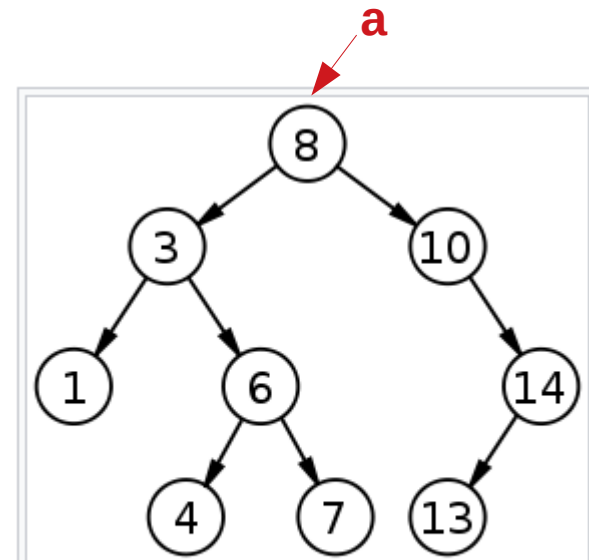
```
--  
16 int InsertarElemento(Arbol_T *a, Tipo_Dato dato)  
17 {  
18     if ((*a) == NULL)  
19         *a = CrearArbol(dato);  
20     else  
21         if (dato < (*a)->dato)  
22             InsertarElemento(&(*a)->izq, dato);  
23         else  
24             InsertarElemento(&(*a)->der, dato);  
25 }  
--
```



# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

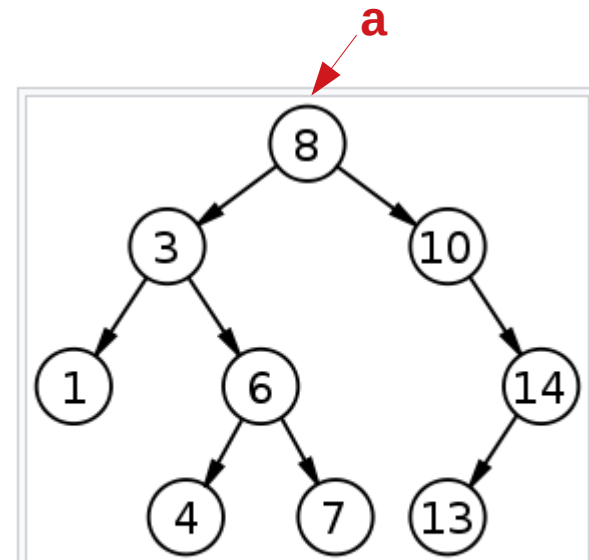


Buscar el elemento 6 en a

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```



La búsqueda implica:

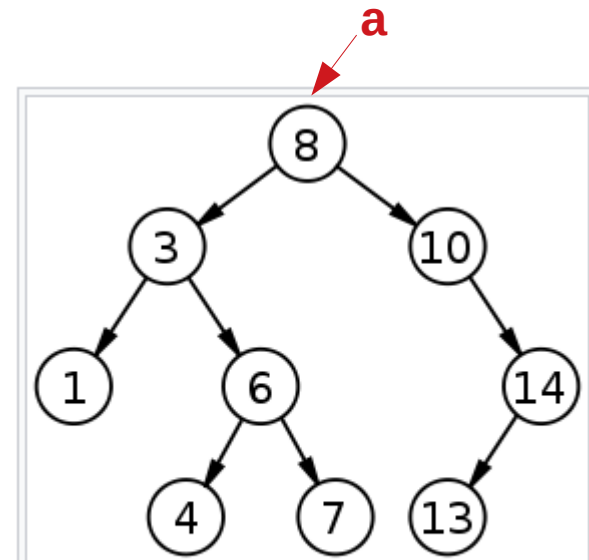
- 1) Verificar si  $a == \text{NULL}$  o encuentro el dato.
- 2) Llamada recursiva con subárbol derecho o izquierdo dependiendo del valor del nodo y del valor buscado

6 ??

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL) Falso
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```



6 ??

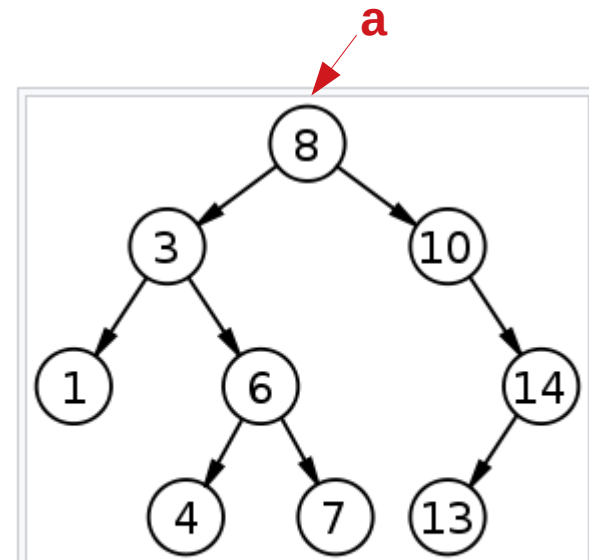
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

**Falso**



**6 ??**

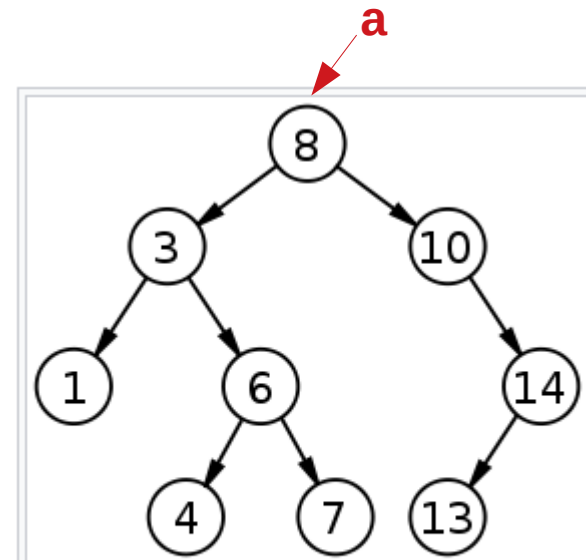
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111     else
112         if (dato < a->dato)
113             return Existe(a->izq, dato);
114         else
115             return Existe(a->der, dato);
116 }
117 }
```

Verdadero



6 ??

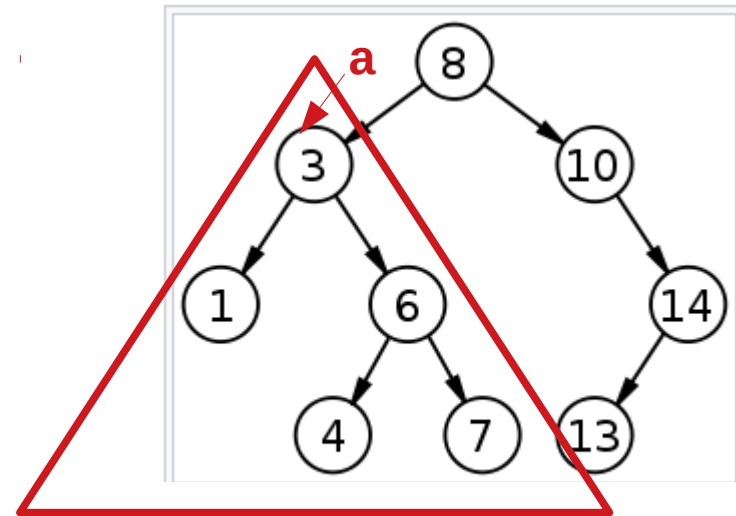
Parametro: 8 retorno: -



# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```



Existe con el subarbol izquierdo!!

6 ??

Parametro: 3 retorno: -  
Parametro: 8 retorno: -

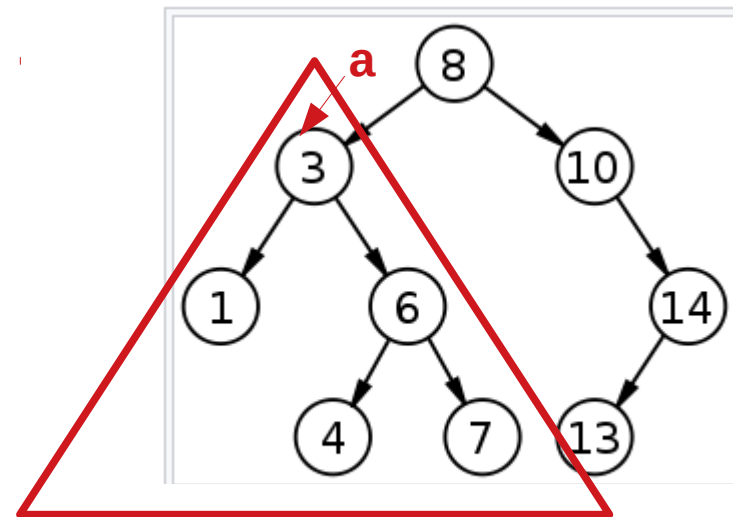


# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Falso



6 ??

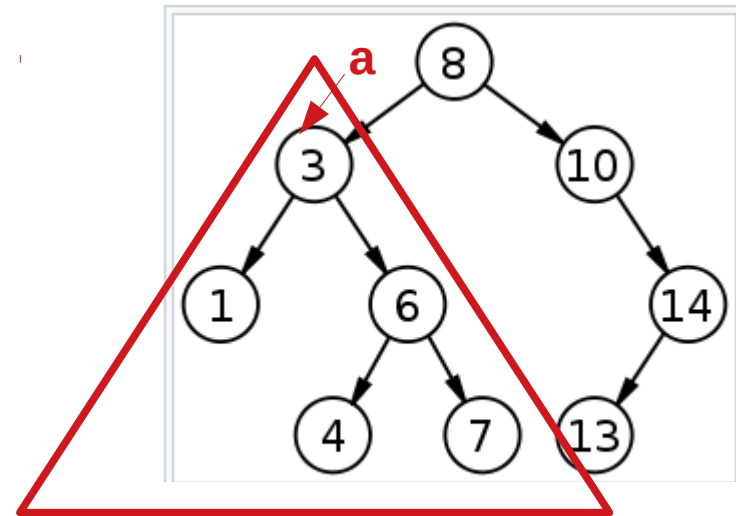
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

**Falso**



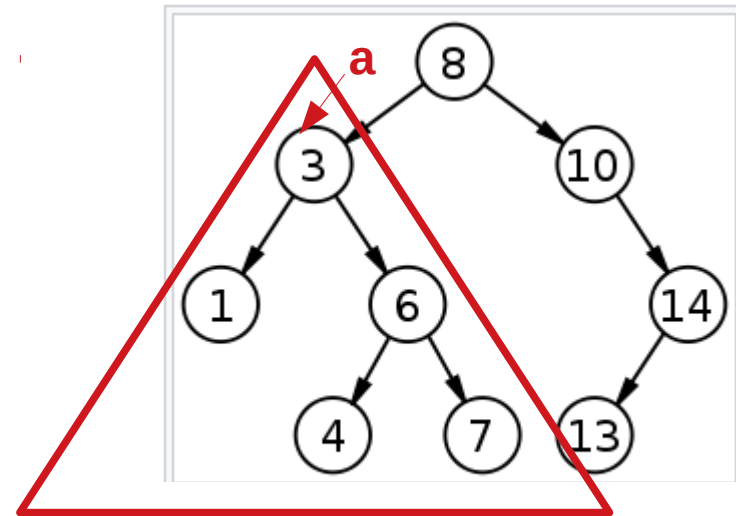
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111     else
112         if (dato < a->dato)
113             return Existe(a->izq, dato);
114         else
115             return Existe(a->der, dato);
116 }
117 }
```

Falso



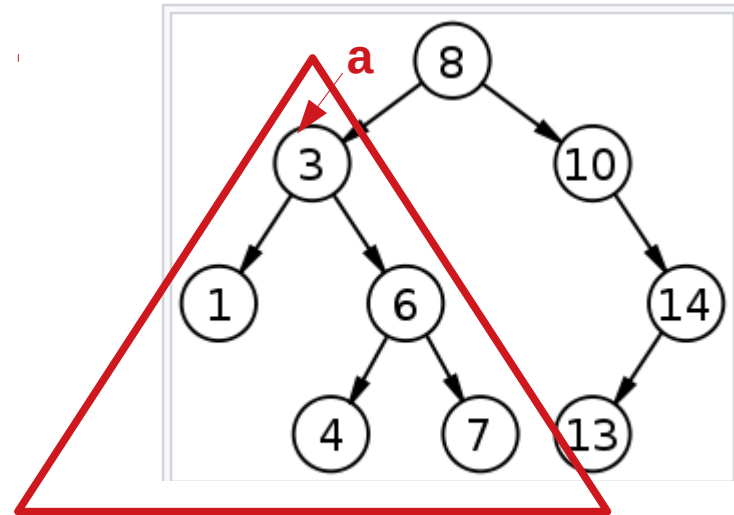
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Llamada  
recursiva



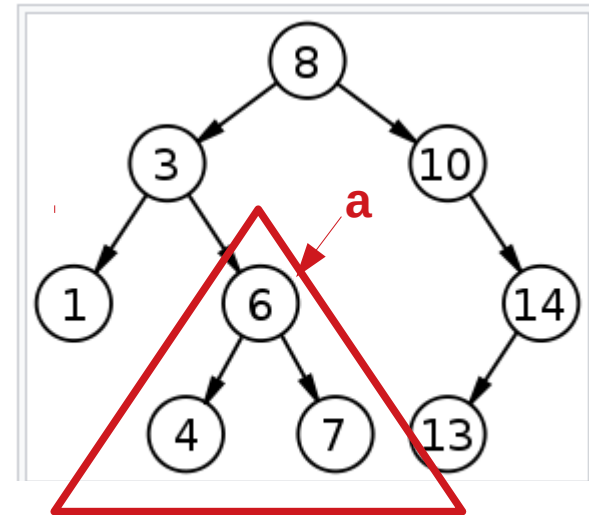
6 ??

Parametro: 3 retorno: -  
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```



Existe con el  
subarbol derecho

6 ??

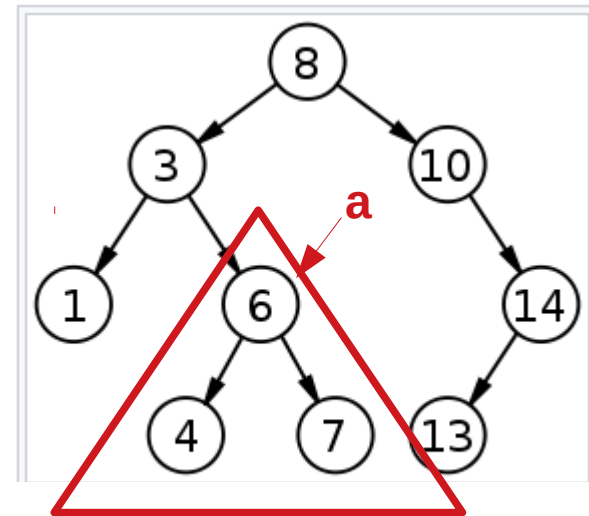
Parametro: 6 retorno: -  
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

**Falso**



**6 ??**

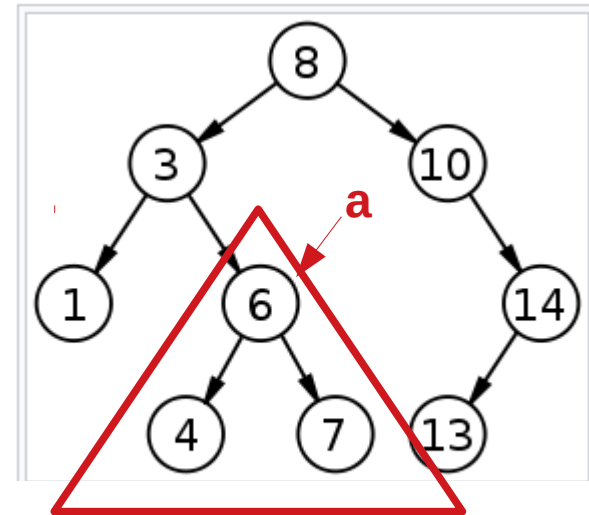
Parametro: 6 retorno: -  
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Verdadero



6 ??

Parametro: 6 retorno: -  
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

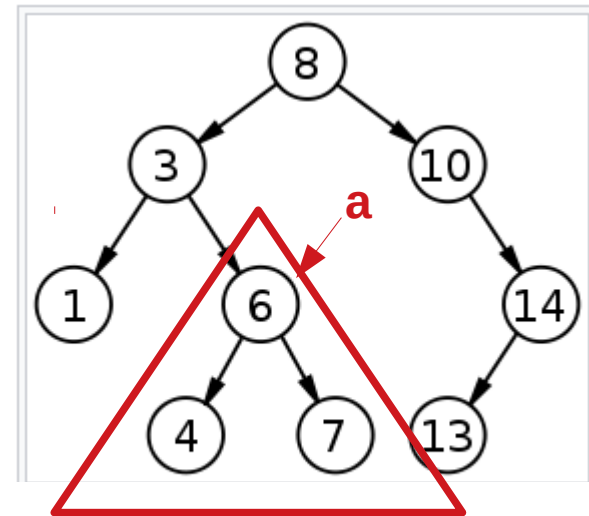


# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Verdadero



6 ??

Retorna 1.  
Fin llamadas recursivas

Parametro: 6 retorno: 1  
Parametro: 3 retorno: -  
Parametro: 8 retorno: -

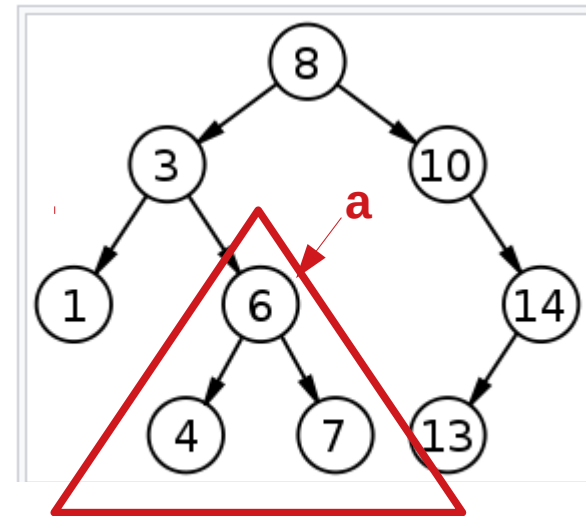


# Árbol binario de búsqueda

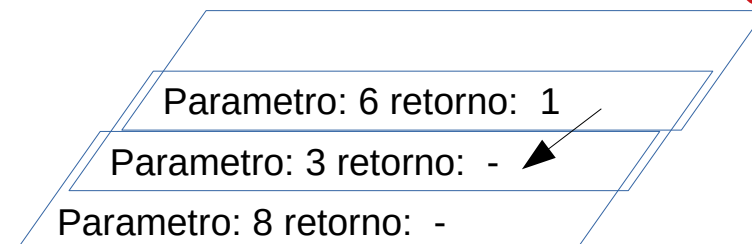
- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Retorna 1



6 ??

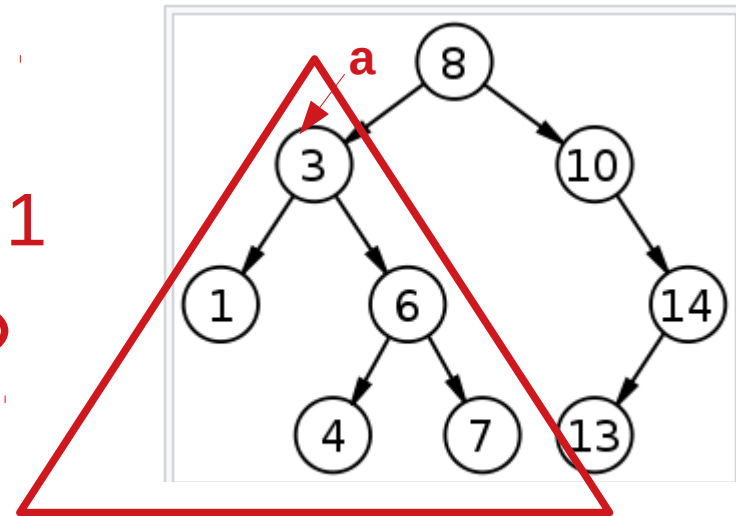


# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Retorna 1



6 ??

Parametro: 3 retorno: 1  
Parametro: 8 retorno: -

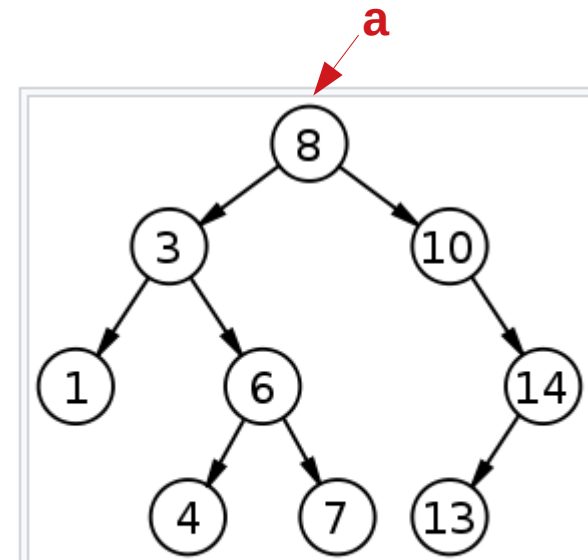
# Árbol binario de búsqueda

- ExisteElemento: 1) Recorrido del árbol según orden del ABB.

```
104 int Existe(Arbol_T a, Tipo_Dato dato)
105 {
106     if (a == NULL)
107         return 0;
108     else
109         if (a->dato == dato)
110             return 1;
111         else
112             if (dato < a->dato)
113                 return Existe(a->izq, dato);
114             else
115                 return Existe(a->der, dato);
116 }
117 }
```

Retorna 1

Retorna 1 y fin de las  
llamadas recursivas

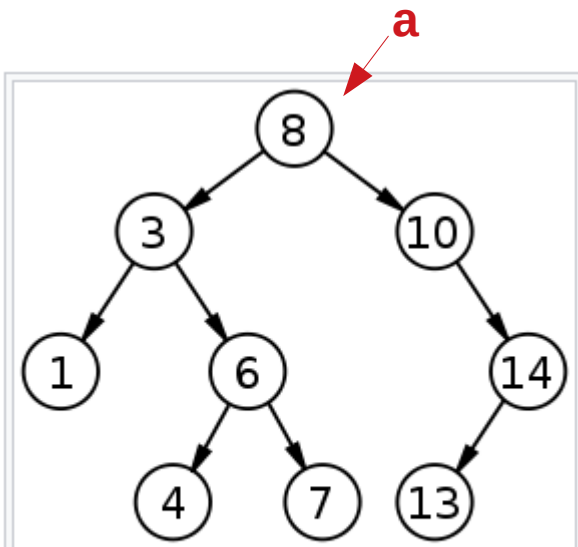


6 ??

Parametro: 8 retorno: 1

# Árbol binario de búsqueda

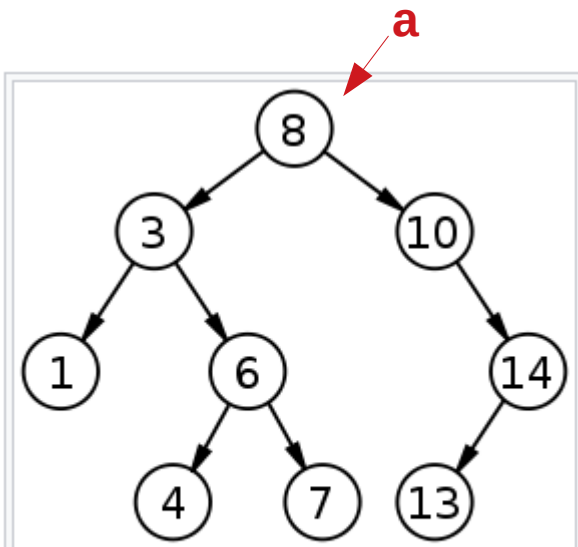
- Recorridos: en orden, post orden, pre orden.
- Estos recorridos visitan cada uno de los nodos. En esta visita realizan el procesamiento que sea necesario (imprimir, sumar, etc).



- Pre orden: VID
- En orden: IVD
- Post orden: IDV

# Árbol binario de búsqueda

- Recorridos: en orden, post orden, pre orden.
- Estos recorridos visitan cada uno de los nodos. En esta visita realizan el procesamiento que sea necesario (imprimir, sumar, etc).



- Pre orden: VID (visita, hijo izquierdo, hijo derecho)
- En orden: IVD (hijo izquierdo, visita, hijo derecho).
- Post orden: IDV (hijo izquierdo, hijo derecho, visita).
- Por ejemplo: EnOrden():

```
void EnOrden(Arbol_T a)
{
    if (a != NULL)
    {
        EnOrden(a->izq);
        Visito(a->dato);
        EnOrden(a->der);
    }
}
```