

Arboles

- ▶ Materia Algoritmos y Estructuras de Datos - CB100
- ▶ Cátedra Schmidt
- ▶ Estructuras dinámicas de Arboles

Estructuras de arboles

- 1) Conjunto
- 2) Arboles
- 3) Arboles binarios
- 4) Arboles binarios de búsqueda
- 5) Arboles AVL
- 6) Árbol B
- 7) Árbol B+

TDA Conjunto

■ ¿Qué es un TDA Conjunto?

Un **Conjunto** (o *Set* en inglés) es una estructura de datos que:

- Almacena elementos **sin duplicados**.
- No guarda un orden específico.
- Permite operaciones clásicas de teoría de conjuntos:
 - **Unión** (todos los elementos de A y B, sin duplicados).
 - **Intersección** (elementos comunes entre A y B).
 - **Diferencia** (elementos en A que no están en B).
 - **Pertenencia** (saber si un elemento está en el conjunto).

Este TDA es útil para representar colecciones de elementos únicos como:

- Conjuntos de palabras.
- Alumnos inscriptos a materias.
- Estados alcanzados en un algoritmo.

```
11 public class Conjunto<T> implements Set<T> {  
12     private List<T> elementos;  
13  
14     public Conjunto() {  
15         elementos = new ArrayList<>();  
16     }  
17  
18     @Override  
19     public int size() {  
20         return elementos.size();  
21     }  
22  
23     @Override  
24     public boolean isEmpty() {  
25         return elementos.isEmpty();  
26     }  
27  
28     @Override  
29     public boolean contains(Object o) {  
30         return elementos.contains(o);  
31     }  
32  
33     @Override  
34     public Iterator<T> iterator() {  
35         return elementos.iterator();  
36     }  
37  
48     @Override  
49     public boolean add(T t) {  
50         if (!elementos.contains(t)) {  
51             elementos.add(t);  
52             return true;  
53         }  
54         return false;  
55     }  
56  
57     @Override  
58     public boolean remove(Object o) {  
59         return elementos.remove(o);  
60     }  
61  
62     @Override  
63     public boolean containsAll(Collection<?> c) {  
64         return elementos.containsAll(c);  
65     }  
66  
67     @Override  
68     public boolean addAll(Collection<? extends T> c) {  
69         boolean changed = false;  
70         for (T item : c) {  
71             if (add(item)) {  
72                 changed = true;  
73             }  
74         }  
75         return changed;  
76     }  
77 }
```

```
77  
78     @Override  
79     public boolean retainAll(Collection<?> c) {  
80         return elementos.retainAll(c);  
81     }  
82  
83     @Override  
84     public boolean removeAll(Collection<?> c) {  
85         return elementos.removeAll(c);  
86     }  
87  
88     @Override  
89     public void clear() {  
90         elementos.clear();  
91     }  
92  
93     public Conjunto<T> union(Conjunto<T> otro) {  
94         Conjunto<T> resultado = new Conjunto<>();  
95         resultado.addAll(this);  
96         resultado.addAll(otro);  
97         return resultado;  
98     }  
99  
100    public Conjunto<T> interseccion(Conjunto<T> otro) {  
101        Conjunto<T> resultado = new Conjunto<>();  
102        for (T elemento : this) {  
103            if (otro.contains(elemento)) {  
104                resultado.add(elemento);  
105            }  
106        }  
107        return resultado;  
108    }  
109  
110    public Conjunto<T> diferencia(Conjunto<T> otro) {  
111        Conjunto<T> resultado = new Conjunto<>();  
112        for (T elemento : this) {  
113            if (!otro.contains(elemento)) {  
114                resultado.add(elemento);  
115            }  
116        }  
117        return resultado;  
118    }  
119  
120    @Override  
121    public String toString() {  
122        return elementos.toString();  
123    }  
124 }
```

Definición

Un árbol es una estructura de datos que posee ramificaciones; no sigue con la estructura lineal de las listas, en esta estructura es posible poseer más de una posición siguiente.

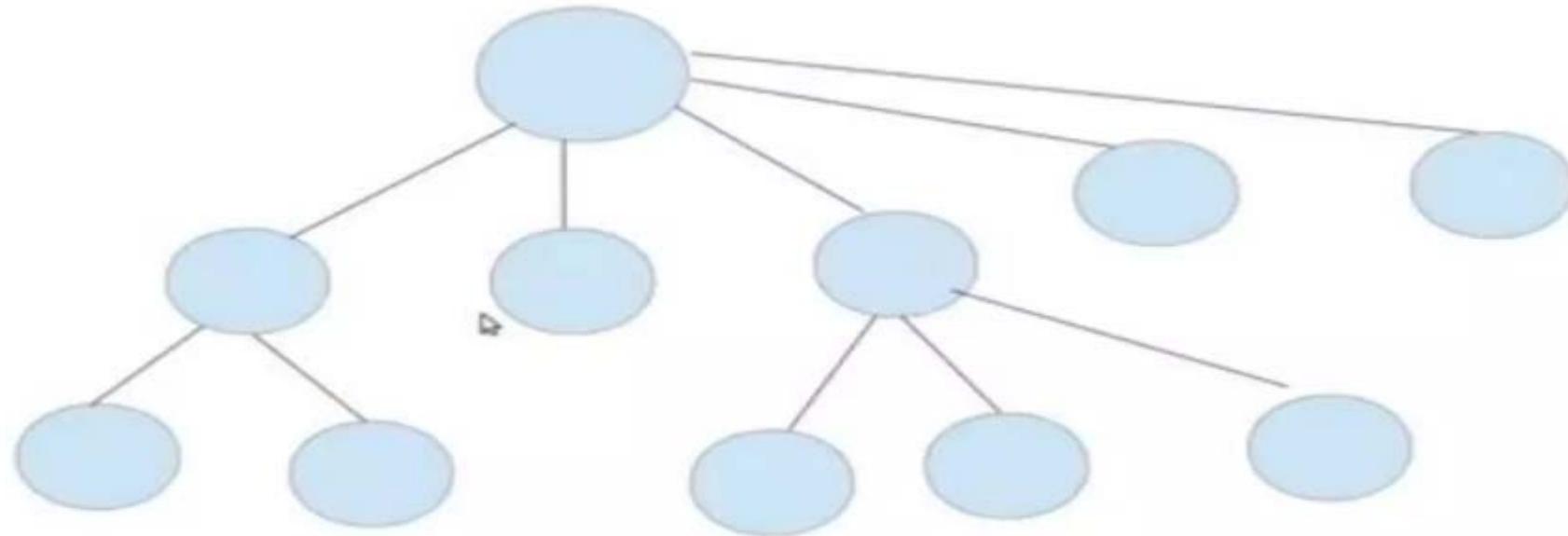
En ella es necesario recorrer cada posición al menos una vez; además los árboles poseen nodos y tienen etiqueta en cada nodo; al igual que las listas posee posiciones y referencias.

En la estructura existe un nodo especial llamado raíz del árbol, el cual proporciona un punto de entrada a la estructura.

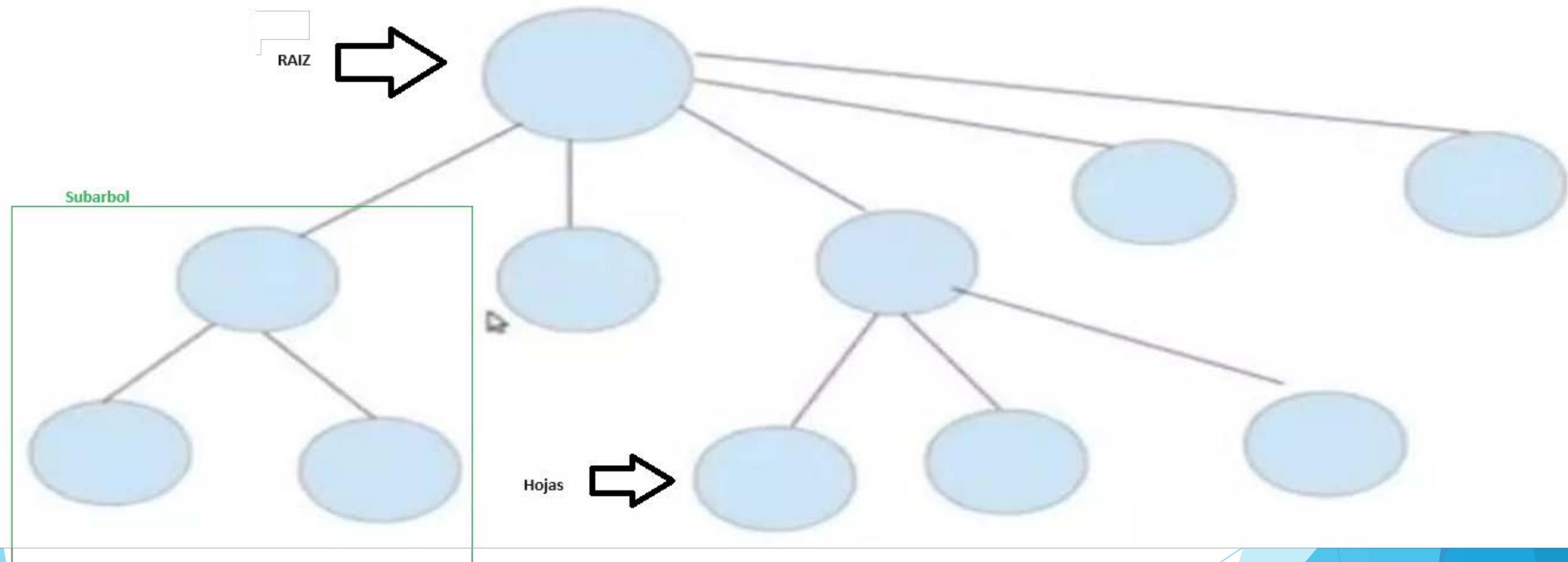
Componentes

- **Raíz:** El nodo (único) que no tiene ancestros propios.
- **Hoja:** Nodo sin descendientes propios.
- **Subárbol:** Un nodo, junto con todos sus descendientes.
- **Grado:** número de hijos que tiene un árbol.
- **Altura de un nodo:** Longitud del camino más largo desde el nodo a una hoja.
- **Ancestros y descendientes:** Si existe un camino, del nodo a al nodo b, entonces a es un ancestro de b y b es un descendiente de a.

Arbol



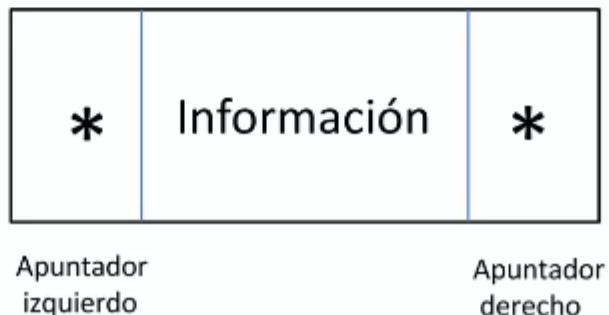
Terminos

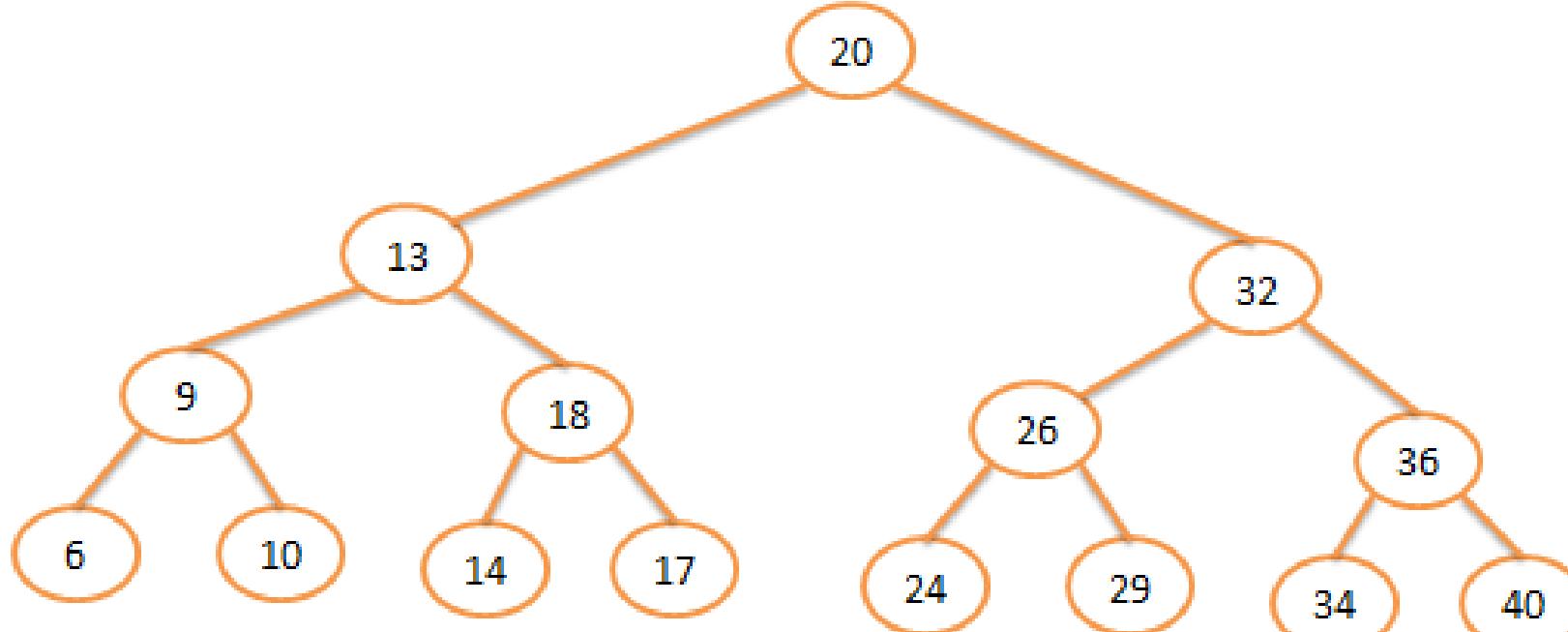


Árbol de búsqueda binario

ABB

- Un árbol de búsqueda binario ABB, es aquel que en cada nodo puede tener como mucho grado 2, es decir, máximo 2 hijos.
- Los hijos suelen denominarse hijo a la izquierda e hijo a la derecha, estableciéndose de esta forma un orden en el posicionamiento de los mismos.
- Estructura del nodo

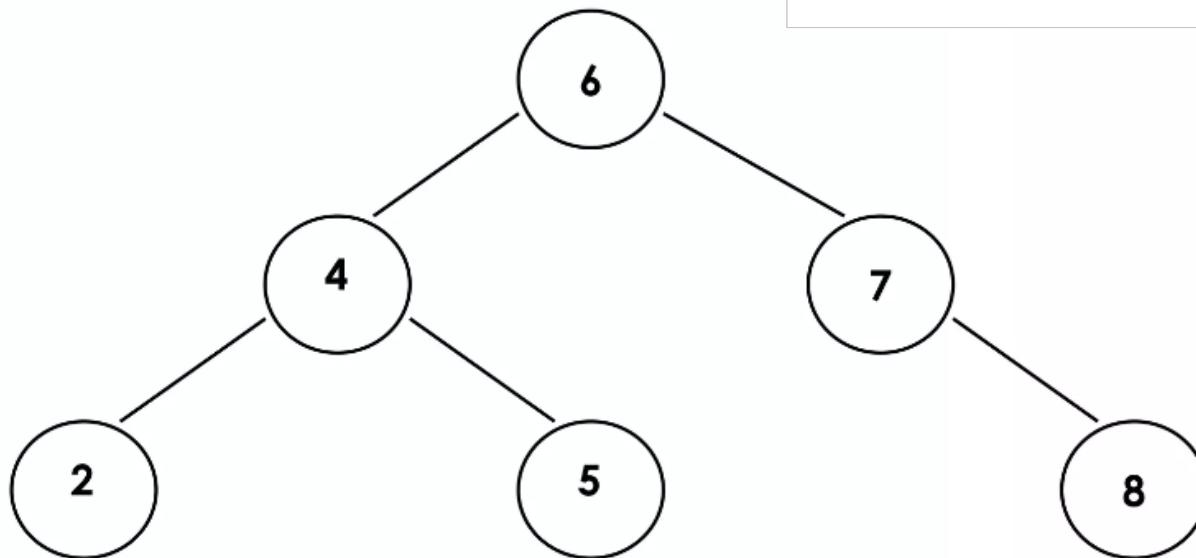




Árbol de búsqueda binaria

- Un árbol de búsqueda binaria (ABB) es un árbol binario que almacena en cada nodo una llave o valor.
- Para que un árbol sea de búsqueda binaria debe cumplir con las siguientes condiciones:
 - Debe ser un árbol binario (subárbol izquierdo y derecho).
 - El valor de la raíz es menor que los valores almacenados en el lado derecho.
 - El valor de la raíz es mayor que todo los valores almacenados del lado izquierdo del nodo.

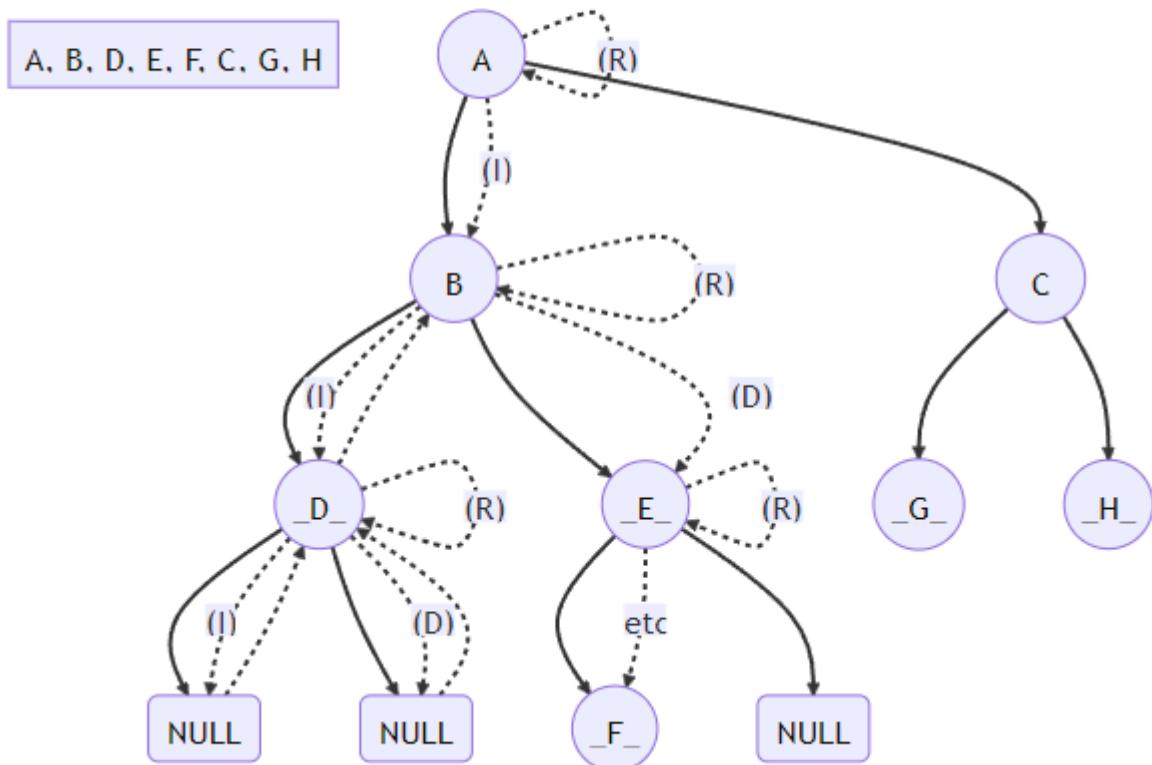
Ejemplo de ABB



- Para recorrer un ABB se tienen en cuenta las siguientes formas:
 - **Preorden:** raíz – izquierda - derecha.
 - **Inorden:** izquierda – raíz - derecha.
 - **Postorden:** izquierda – derecha - raíz.

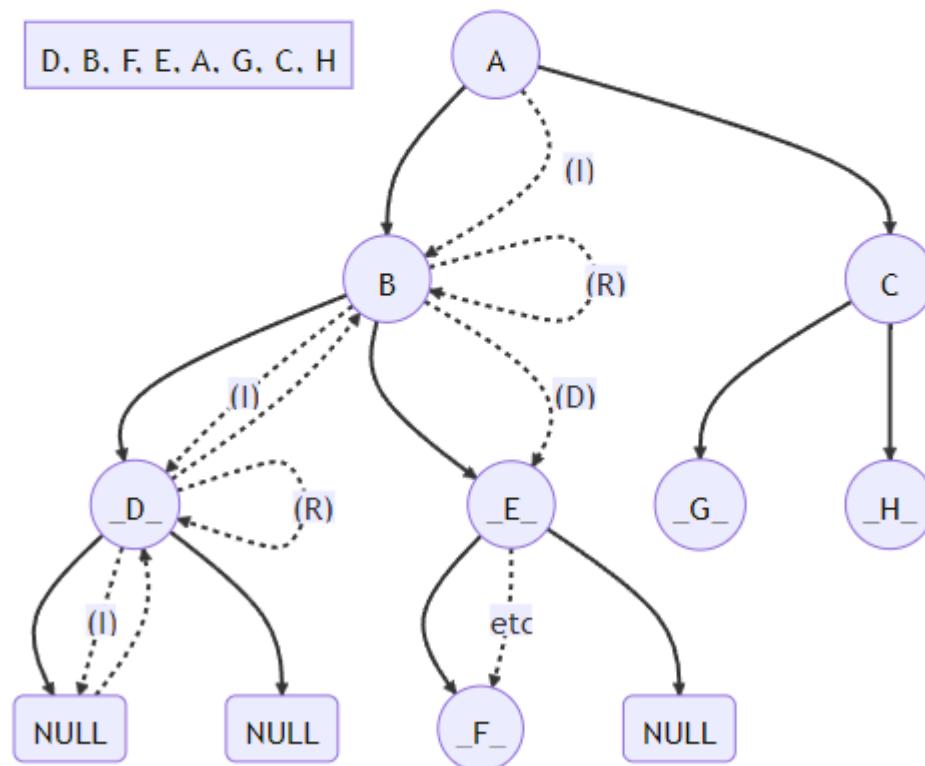
Preorden (R | ID)

- (R) Raíz
- (I) Izquierdo
- (D) Derecho



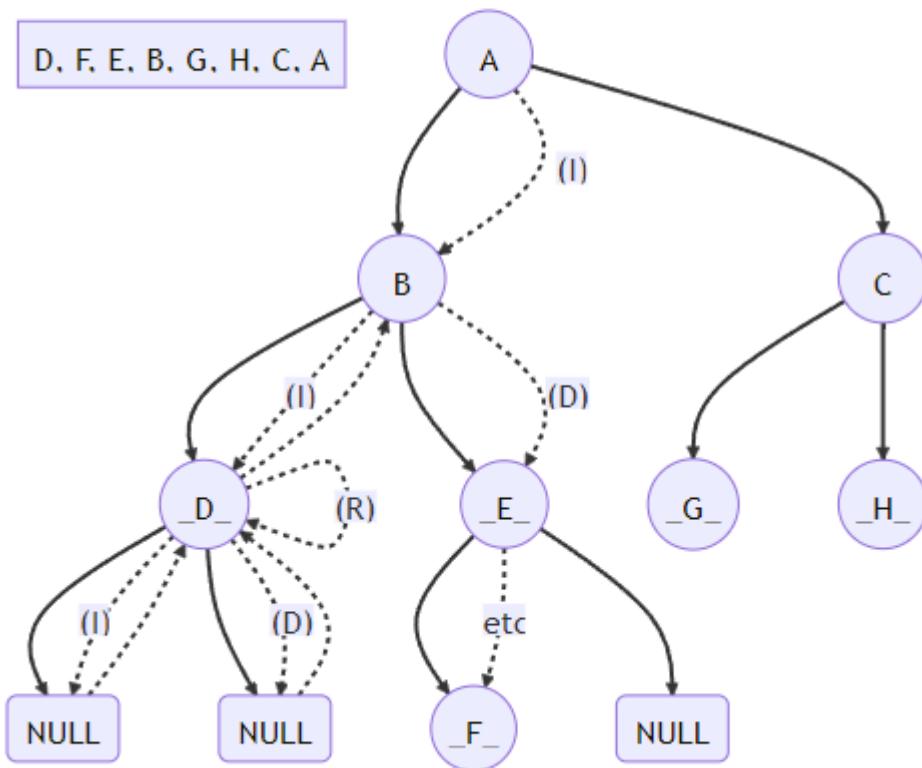
Inorden (I | R | D)

- (I) Izquierdo
- (R) Raíz
- (D) Derecho

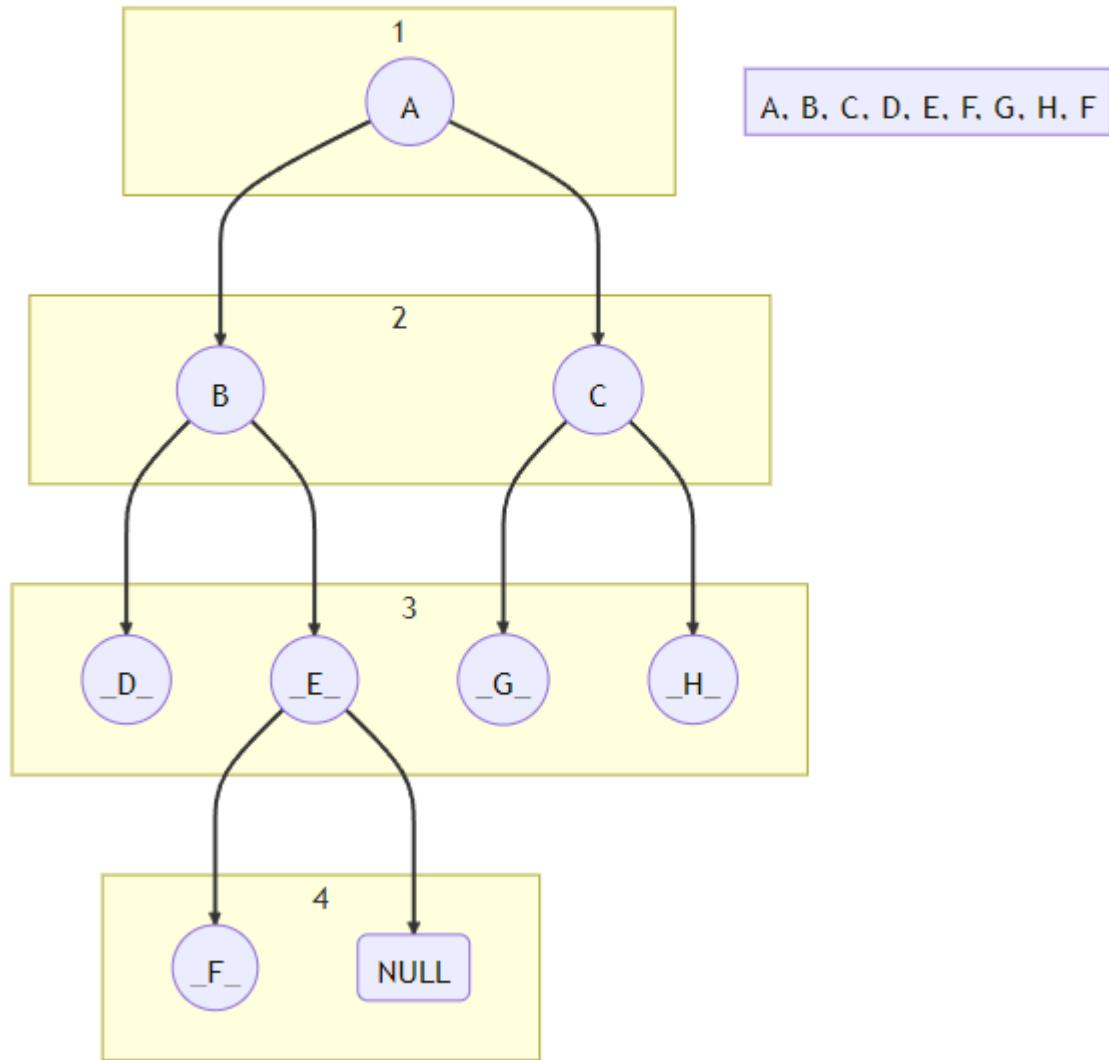


Posorden (ID | R)

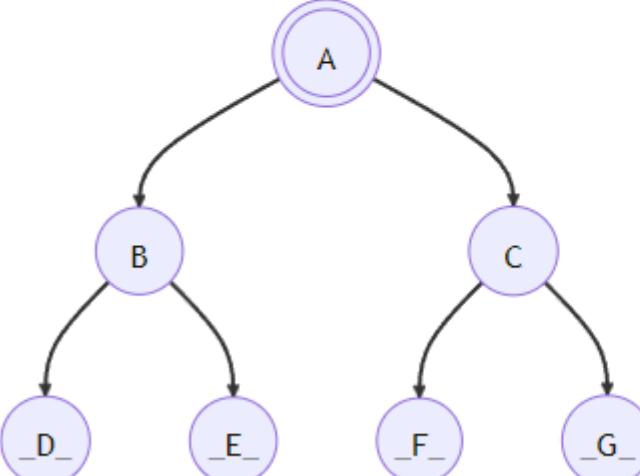
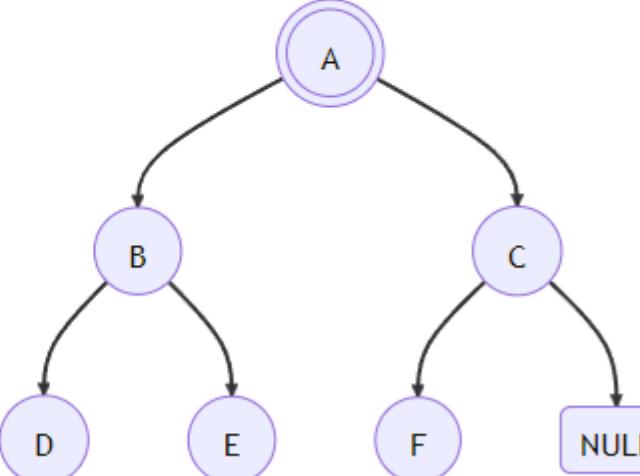
- (I) Izquierdo
- (D) Derecho
- (R) Raíz



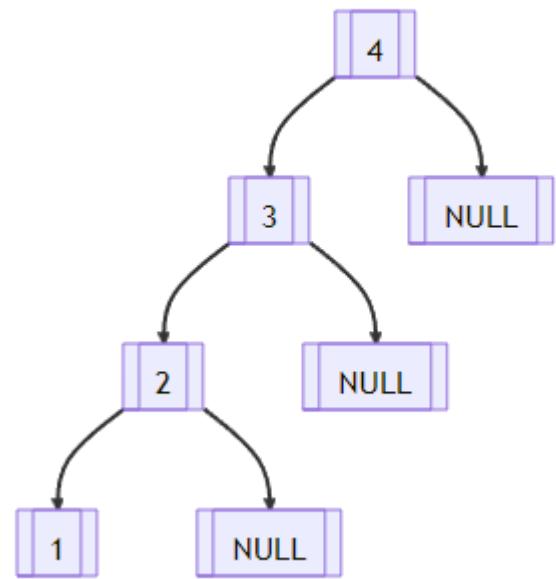
Recorrido en anchura



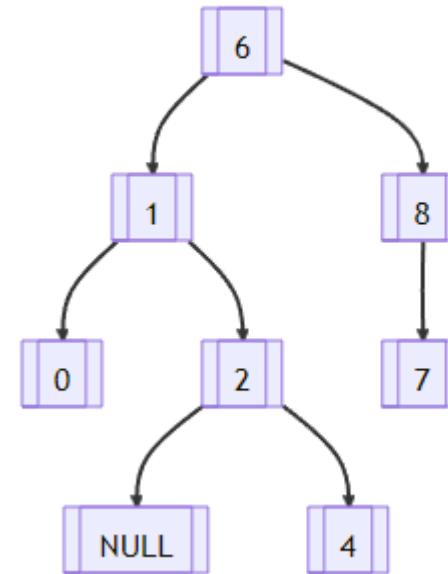
Definiciones

Árbol Lleno	Árbol Completo
<p>Todas sus hojas están al mismo nivel h y todos los nodos anteriores tienen el número máximo de hijos (en un árbol binario, 2).</p>  <pre>graph TD; A((A)) --> B((B)); A --> C((C)); B --> D((D)); B --> E((E)); C --> F((F)); C --> G((G));</pre>	<p>Todas sus hojas llenas hasta $h-1$ y todos los nodos del nivel h están lo más a la izquierda posible.</p>  <pre>graph TD; A((A)) --> B((B)); A --> C((C)); B --> D((D)); B --> E((E)); C --> F((F)); C --> NULL[NULL];</pre>

Árbol degenerado e



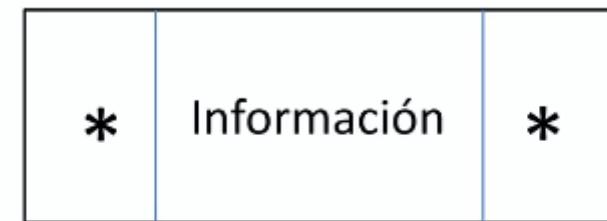
Árbol sin equilib



Implementación

```
4  
3 public class ArbolBinarioDeBusqueda<T extends Comparable<T>> {  
4  
5     private Nodo<T> raiz = null;  
6
```

```
4  
3 class NodoDeArbol<T> {  
4     private T valor;  
5     private NodoDeArbol<T> izquierdo = null;  
6     private NodoDeArbol<T> derecho = null;  
7
```



Apuntador
izquierdo

Apuntador
derecho

Operaciones

- **Inserción:** Cuando se inserta un nuevo nodo en el árbol hay que tener en cuenta que cada nodo NO puede tener más de dos hijos.
- **Búsqueda:** El algoritmo compara el elemento a buscar con la raíz, si es menor continua la búsqueda por la rama izquierda, si es mayor continua por la derecha. Este procedimiento se realiza recursivamente hasta que se encuentra el nodo o hasta que se llega al final del árbol.

Borrar nodo

- Tras realizar la búsqueda del nodo a eliminar observamos que:
 - **El nodo no tiene hijos:** se borra el elemento y se concluye la operación.
 - **El nodo tiene un sólo hijo:** para borrar el nodo se hace una especie de puente, el padre del nodo a borrar pasa a apuntar al hijo del nodo borrado.

Algoritmo de búsqueda

Se parte del nodo raíz, hay que descender a lo largo del árbol a izquierda o derecho dependiendo del elemento que se busca.

1. Si el árbol está vacío, se finaliza la búsqueda: el elemento no está en el árbol.
2. Si el valor del nodo raíz es igual que el del elemento que se busca, se finaliza la búsqueda con éxito.
3. Si el valor del nodo raíz es mayor que el elemento que se busca, la búsqueda continúa en el árbol izquierdo.
4. Si el valor del nodo raíz es menor que el elemento que se busca, la búsqueda continúa en el árbol derecho.

El valor de retorno de una función de búsqueda en un ABB puede ser un puntero al nodo encontrado, o NULL, si no se ha encontrado.

Algoritmo de búsqueda

```
int Buscar(Arbol a, int dat) {  
    pNodo actual = a;  
  
    while(!Vacio(actual)) {  
        if(dat == actual->dato)  
            return 1; /* encontrado (2) */  
        else  
            if(dat < actual->dato)  
                actual = actual->izquierdo; /*(3)*/  
            else  
                if(dat > actual->dato)  
                    actual = actual->derecho;  
                /*(4)*/  
    }  
    return 0; /* No está en árbol (1) */  
}
```

Para comprobar si el árbol está vacío solo se implementa la función:

```
int Vacio(Arbol r) {  
    return r == NULL;  
}
```

Insertar un elemento

- El algoritmo de insertar un elemento se basa en el algoritmo de búsqueda. Si el elemento está en el árbol no se inserta. Si no está, se inserta a continuación del último nodo visitado.
- Se necesita un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

Algoritmo de inserción

- Padre = NULL
- nodo = Raíz
- Ciclo: mientras actual no sea un árbol vacío o hasta que se encuentre el elemento.
 - Si el valor del nodo raíz es mayor que el elemento que se busca, continua la búsqueda en el árbol izquierdo:
 - Padre = nodo, nodo = nodo->izquierdo.
 - Si el valor del nodo raíz es menor que el elemento que se busca, la búsqueda continúa en el árbol derecho:
 - Padre = nodo, nodo = nodo->derecho.
- Si nodo no es NULL, el elemento está en el árbol.
- Si padre es NULL, el árbol está vacío, por lo tanto, el nuevo árbol sólo contendrá el nuevo elemento, que será la raíz del árbol.
- Si el elemento es menor que el padre, entonces se inserta el nuevo elemento como un nuevo árbol izquierdo del padre.
- Si el elemento es mayor que el padre, entonces se inserta el nuevo elemento como un nuevo árbol derecho de padre.

Algoritmo de inserción

```
void Insertar(Arbol *a, int dat) {
    pNodo padre = NULL; /* (1) */
    pNodo actual = *a; /* (2) */

    while(!Vacio(actual) && dat != actual->dato) { /* (3) */
        padre = actual;
        if(dat < actual->dato)
            actual = actual->izquierdo; /* (3-a) */
        else
            if(dat > actual->dato)
                actual = actual->derecho; /* (3-b) */
    }
}
```

```
if(!Vacio(actual)) return; /* (4) */  
  
if(Vacio(padre)) { /* (5) */  
    *a = (Arbol) new (sizeof(tipoNodo));  
    (*a)->dato = dat;  
    (*a)->izquierdo = (*a)->derecho = NULL; }  
else  
    if(dat < padre->dato) { /* (6) */  
        actual = (Arbol) new (sizeof(tipoNodo));  
        padre->izquierdo = actual;  
        actual->dato = dat;  
        actual->izquierdo = actual->derecho = NULL; }  
    else  
        if(dat > padre->dato) { /* (7) */  
            actual = (Arbol)malloc(sizeof(tipoNodo));  
            padre->derecho = actual;  
            actual->dato = dat;  
            actual->izquierdo = actual->derecho = NULL; }  
} /* Fin de la función inserción */
```

Borrar un elemento

El algoritmo de borrar un elemento también se basa en el algoritmo de búsqueda.

Si el elemento no está en el árbol no se puede borrar.

- Padre = NULL
- Si el árbol está vacío: el elemento no está en el árbol, por lo tanto se retorna sin eliminar ningún elemento.

Si el valor del nodo raíz es igual que el del elemento que se busca, se analizan los siguientes casos:

```
actual->izquierdo = actual->derecho = NULL; }  
} /* Fin de la función inserción */
```

(1) El nodo raíz es un nodo hoja:

- Si padre es NULL, el nodo raíz es el único del árbol, por lo tanto el puntero al árbol debe ser NULL.
- Si raíz es la rama derecha de padre, esa rama apunta a NULL.
- Si raíz es la rama izquierda de padre, esa rama apunta a NULL.
- Se elimina el nodo, y se retorna.

(2) El nodo no es un nodo hoja:

- Se busca el nodo más a la izquierda del árbol derecho de raíz o el más a la derecha del árbol izquierdo. Hay que tener en cuenta que puede que sólo exista uno de esos árboles. Al mismo tiempo, se actualiza el padre para que apunte al parente de nodo.
- Se intercambian los elementos de los nodos raíz y nodo.
- Se borra el nodo 'nodo'. Esto significa volver a (1), ya que puede suceder que 'nodo' no sea un nodo hoja.

Si el valor del nodo raíz es mayor que el elemento que se busca, la búsqueda continúa en el árbol izquierdo.

Si el valor del nodo raíz es menor que el elemento que se busca, la búsqueda continúa en el árbol derecho.

Algoritmo de borrado

```
void Borrar (Arbol *a, int dat) {  
    pNodo padre = NULL; /* (1) */  
    pNodo actual; pNodo nodo;  
    int aux; actual = *a;  
    while(!Vacio(actual)) {  
        /* Búsqueda (2) else implícito */  
        if(dat == actual->dato) { /* (3) */  
            if (EsHoja (actual)) { /* (3-a) */  
                if(padre) /* (3-a-i caso else implícito) */  
                    if (padre->derecho == actual)  
                        /* (3-a-ii) */  
                            padre->derecho = NULL;  
                else if (padre->izquierdo == actual)  
                    /* (3-a-iii) */  
                            padre->izquierdo = NULL;  
                delete (actual); /* (3-a-iv) */  
                actual = NULL;  
            }  
            return;  
        }  
    }  
}
```

```
else /* (3-b) */ /* Buscar nodo */
    padre = actual; /* (3-b-i) */
    if (actual->derecho) {
        nodo = actual->derecho;
        while(nodo->izquierdo) {
            padre = nodo;
            nodo = nodo->izquierdo; } //While
    } //if
    else {
        nodo = actual->izquierdo;
        while(nodo->derecho) {
            padre = nodo;
            nodo = nodo->derecho; } //While
    }
    /* Intercambio */
    aux = actual->dato; /* (3-b-ii) */
    actual->dato = nodo->dato;
    nodo->dato = aux; actual = nodo;
}
```

```
 } //primer if
else {
    padre = actual;
    if (dat > actual->dato)
        actual = actual->derecho; /* (4) */
    else if (dat < actual->dato)
        actual = actual->izquierdo; /* (5) */
    }
} //primer while
} // fin función borrar
```

//Función es hoja

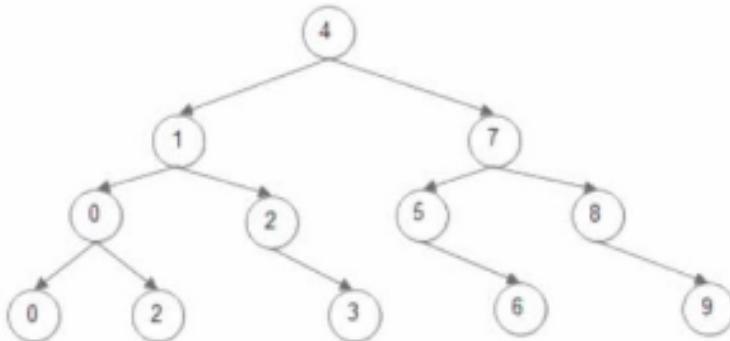
```
int EsHoja (pNodo r) {
    return !r->derecho && !r->izquierdo;
}
```

nodo->dato = aux; actual = nodo;

```
}
```

¿Qué es un árbol AVL?

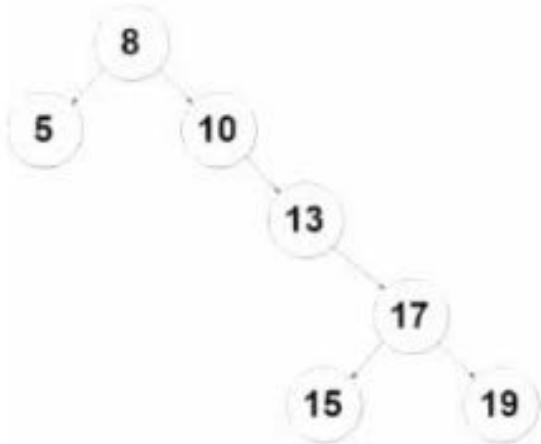
- ▶ Su nombre deriva de los creadores de este algoritmo, los matemáticos Adelson-Velski y Landis (1962)
- ▶ Es un árbol binario de búsqueda (ABB) que tiene como característica que siempre está balanceado.



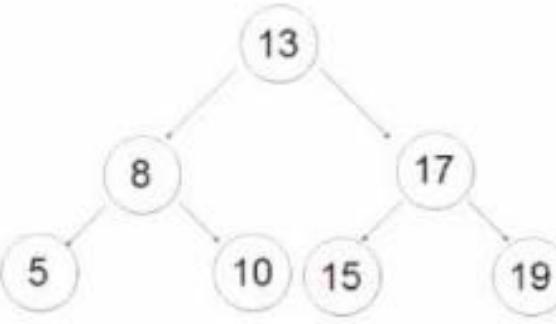
Operaciones en Árboles AVL

- ▶ Las operaciones de:
 - Insertar
 - Eliminar
 - Buscar
- ▶ Cuando se inserta o elimina un elemento, se comprueba si el árbol está equilibrado, en caso de no estarlo se realiza el balanceo

Árbol degenerado y árbol balanceado



Árbol Degenerado



Árbol Balanceado

Rotaciones

- ▶ Para balancear un árbol AVL necesariamente se tienen que realizar rotaciones.
- ▶ Existen 4 variaciones:
- ▶ Rotación simple a la derecha (RSD)
- ▶ Rotación simple a la izquierda (RSI)
- ▶ Rotación doble a la derecha (RD)
- ▶ Rotación doble a la derecha (RDI)

Factor de Equilibrio

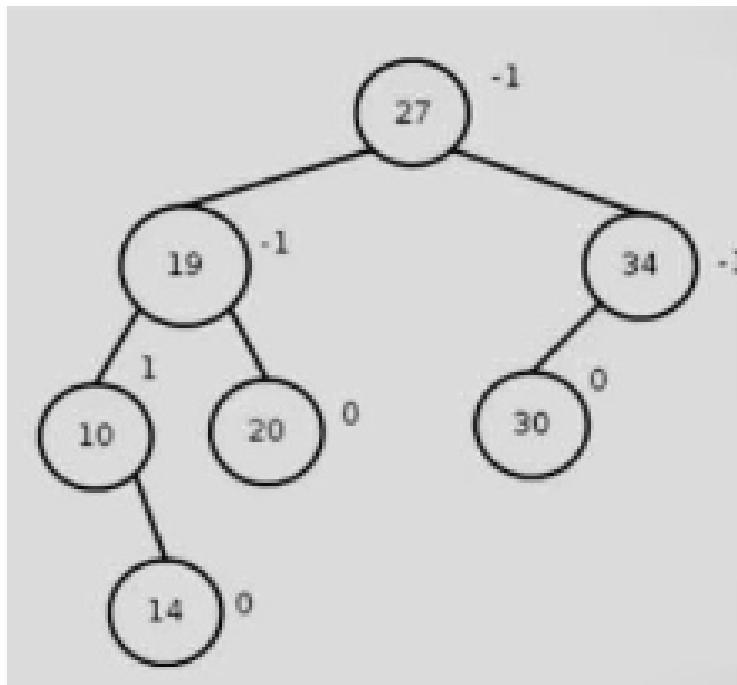
- ▶ Para lograr que el árbol esté balanceado, se debe reestructurar el árbol rotando los nodos del mismo.
- ▶ Se requiere del Factor de Equilibrio o balanceo, el cual se define como “*la diferencia entre las alturas del árbol izquierdo y el derecho*”

$$FE = \text{altura subárbol derecho} - \text{altura subárbol izquierdo}$$

- ▶ Para que sea un árbol AVL el valor de FE deben ser 1,0,-1
 - -1 = cargado a la izquierda
 - 0 = equilibrado
 - 1 = cargado a la derecha

Factor de Equilibrio

- ▶ El factor de equilibrio significa un cambio en la estructura de los árboles ABB.
- ▶ En la estructura del nodo de un árbol de agrega un nuevo miembro, el cual indica su factor de equilibrio.



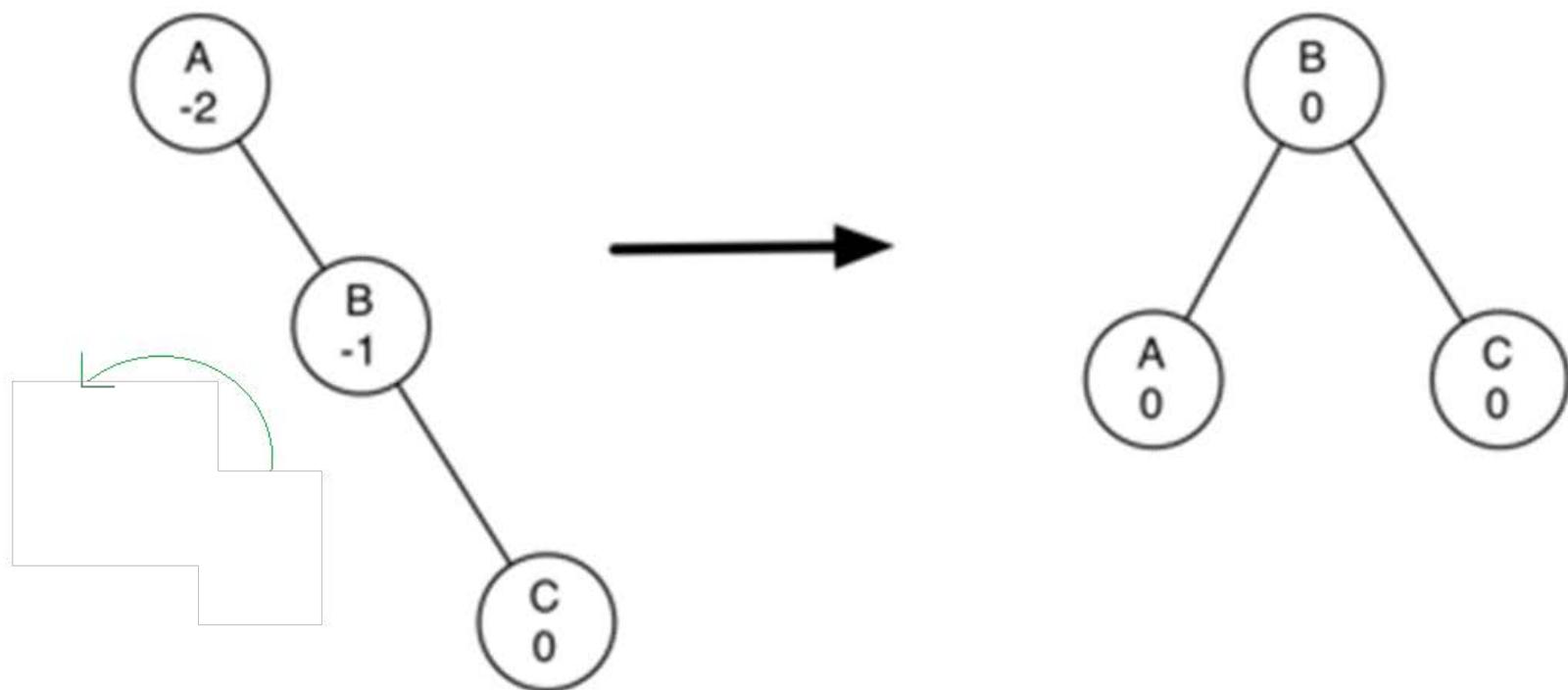
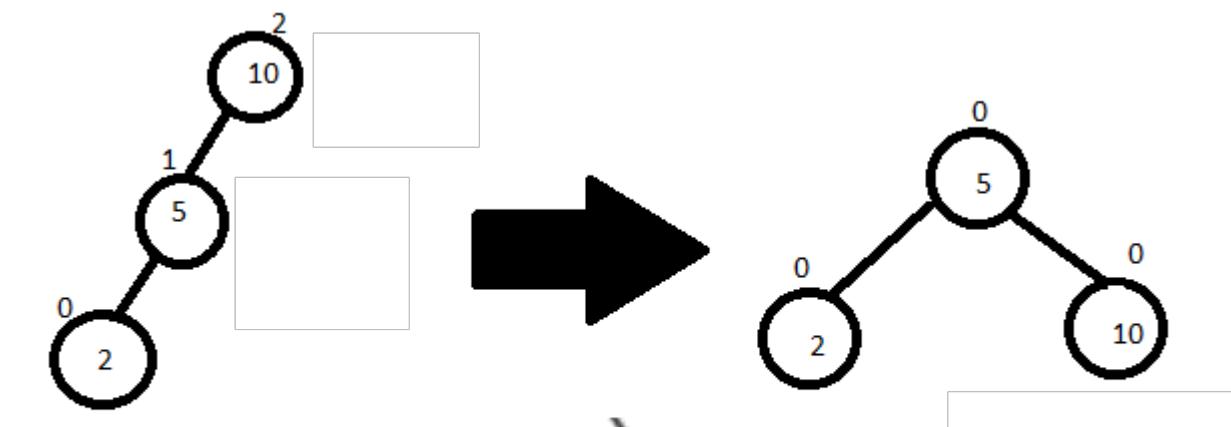


Figura 3: Transformación de un árbol desequilibrado usando una rotación a la izquierda

Figura 3: Transformación de un árbol desequilibrado usando una rotación a la izquierda

Rotación simple a derecha



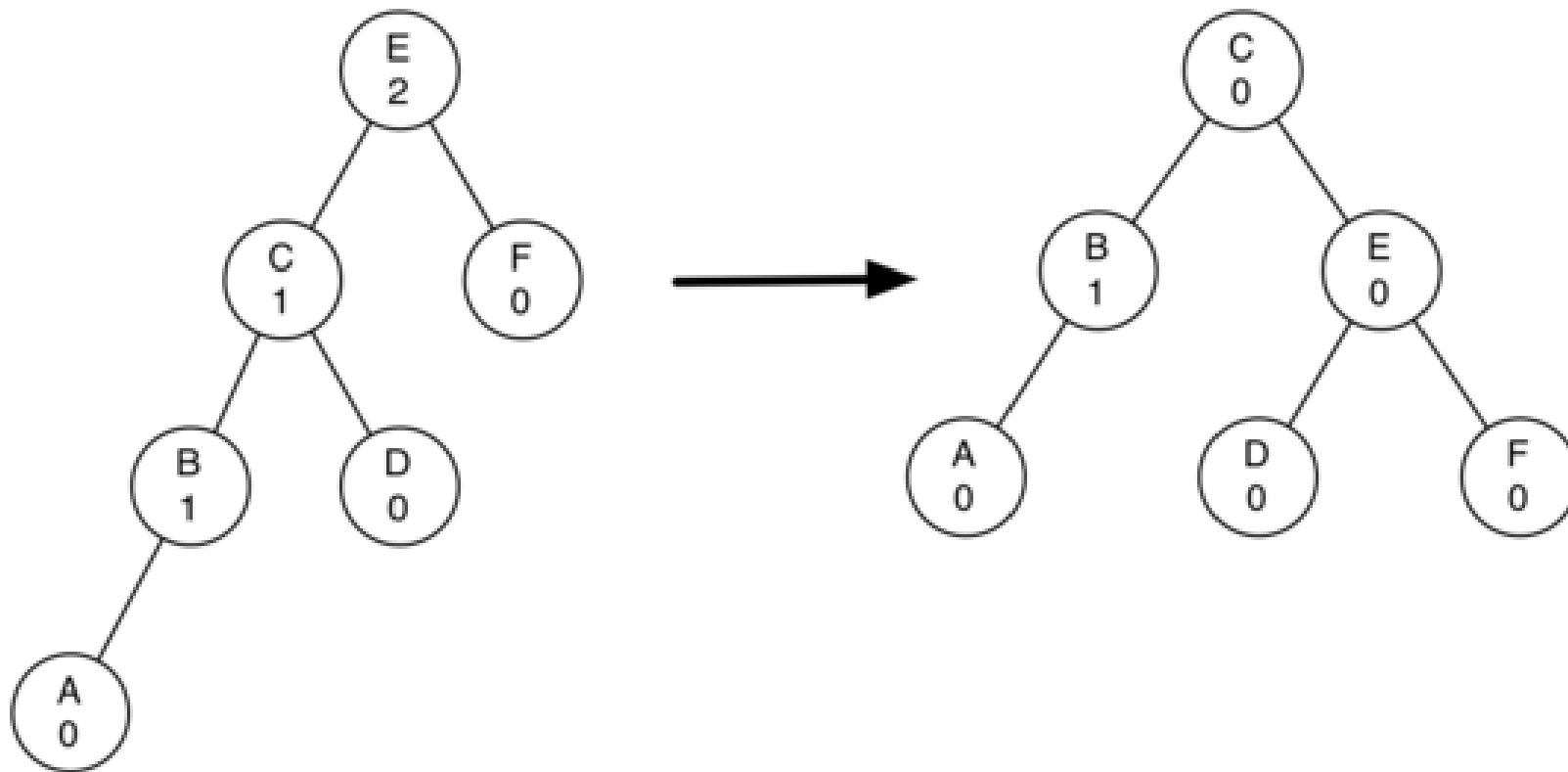


Figura 4: Transformación de un árbol desequilibrado usando una rotación a la derecha

Figura 4: Transformación de un árbol desequilibrado usando una rotación a la derecha

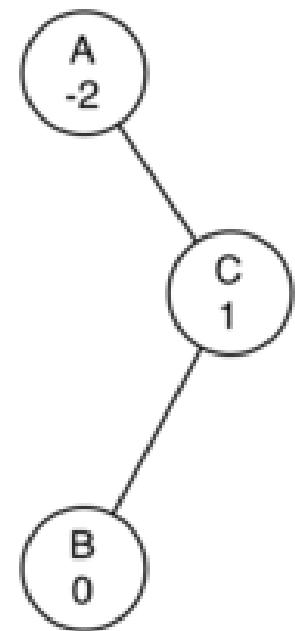


Figura 6: Un árbol desequilibrado que es más difícil de equilibrar

Figura 6: Un árbol desequilibrado que es más difícil de equilibrar

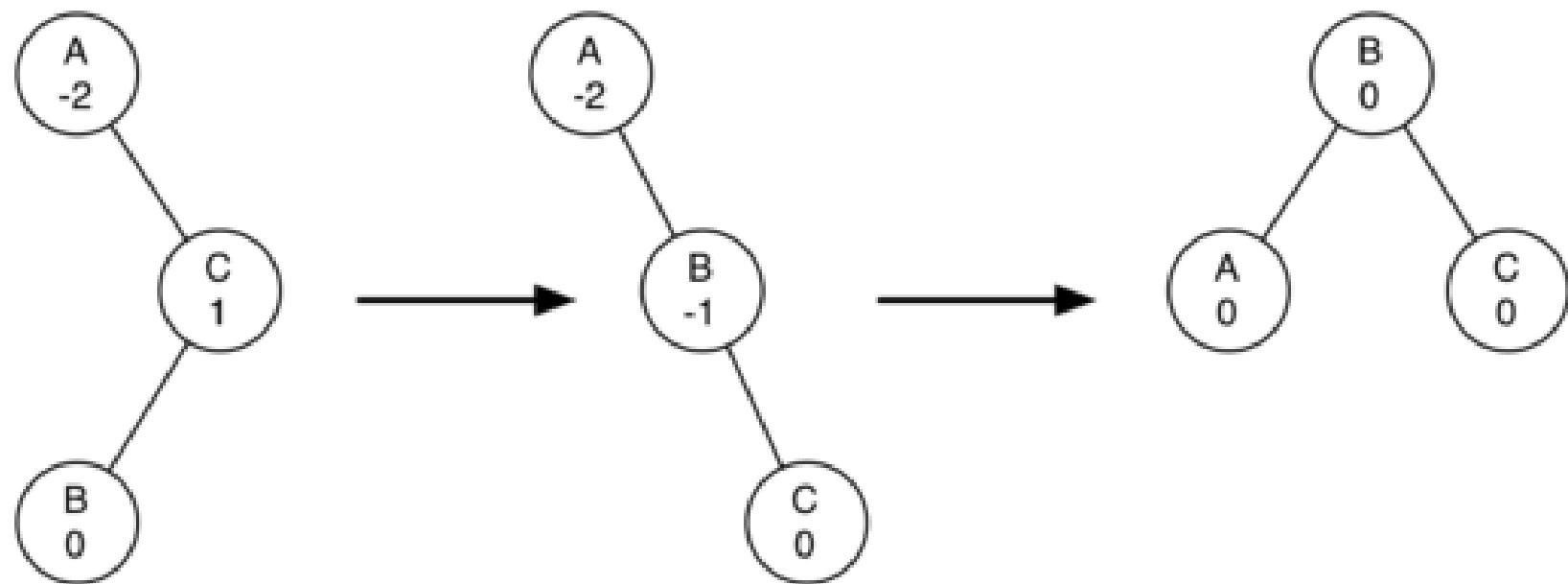


Figura 8: Una rotación a la derecha seguida de una rotación a la izquierda

Figura 8: Una rotación a la derecha seguida de una rotación a la izquierda

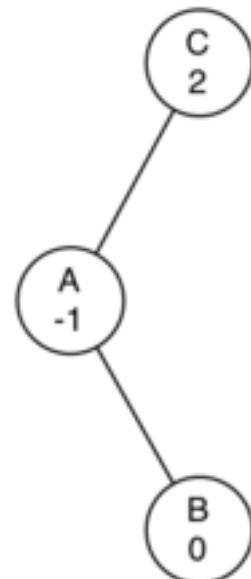
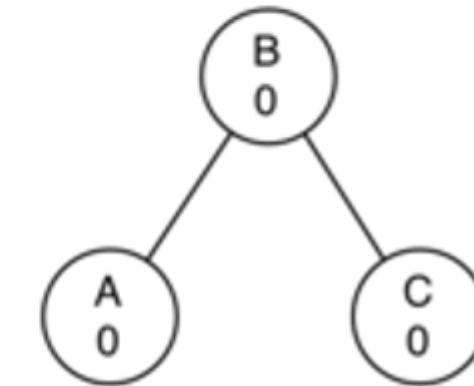
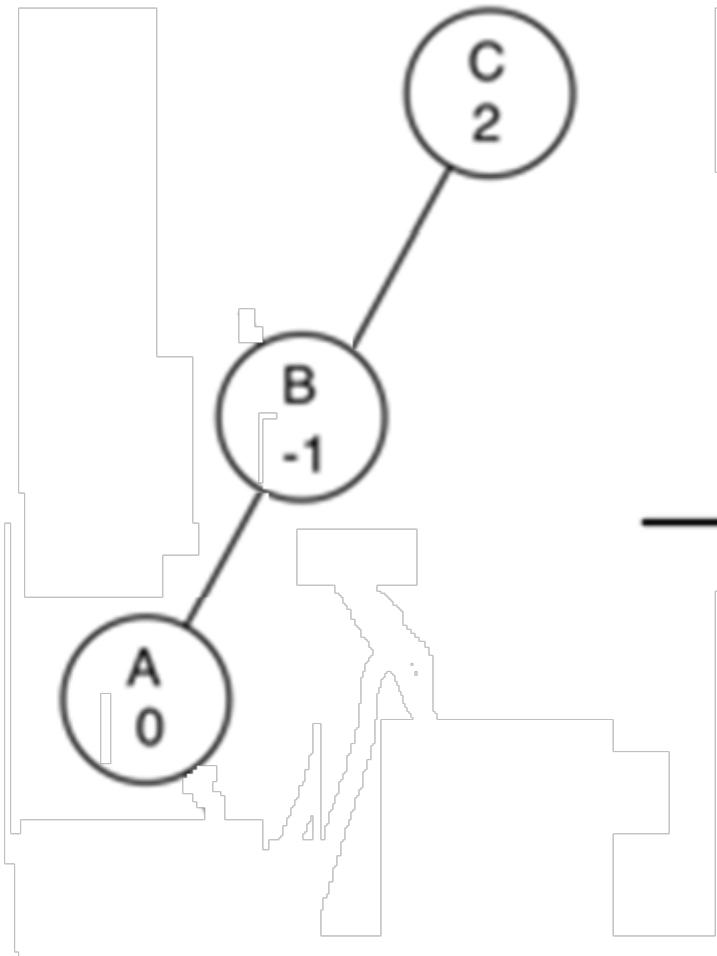
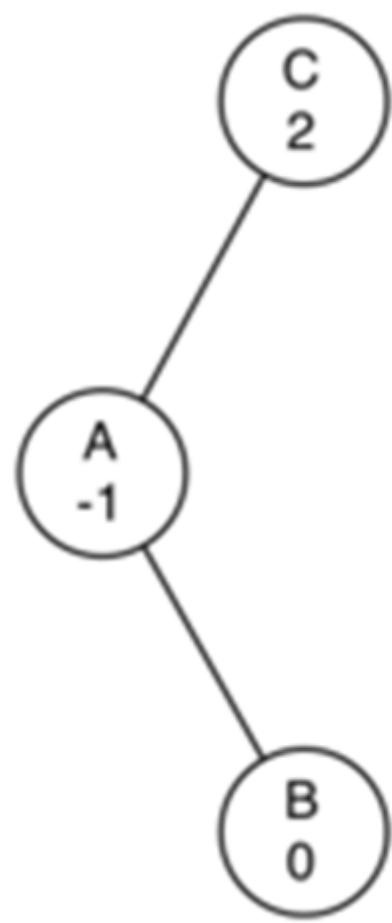


Figura 7: Despues de una rotación a la izquierda el árbol está desequilibrado en la otra dirección

Figura 7: Despues de una rotación a la izquierda el árbol está desequilibrado en la otra dirección



Rotaciones dependiendo el caso

- ▶ **Rotación Simple a la Derecha (RSD):** Se tiene un factor de equilibrio de -2 y su nodo izquierdo no tiene un factor de equilibrio de 1.
- ▶ **Rotación Simple a la Izquierda (RSI):** Se tiene un factor de equilibrio de 2 y su nodo derecho no tiene un factor de equilibrio de -1.
- ▶ **Rotación Doble a la Izquierda (RDI):** Se tiene un factor de equilibrio de 2 y su nodo derecho tiene un factor de equilibrio de -1.
- ▶ **Rotación Doble a la Derecha (RDD):** Se tiene un factor de equilibrio de -2 y su nodo izquierdo tiene un factor de equilibrio de 1.

Simulador: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

Arboles de Fibonacci - análisis de eficiencia de un AVL

# niveles	1	2	3	4	5	6
Mínima cantidad de nodos	1	2	4	7	12	20
	•	• •	• • •	• • • •	• • • • •	• • • • • •

“Peores AVL”: los que tienen la menor cantidad de nodos para un determinado número de niveles.

Se observa una relación entre la cantidad minima de nodos para un árbol AVL de determinada altura y las cantidades correspondientes a los precedentes en altura.

Llamando $F(h)$ a la menor cantidad de nodos para un árbol AVL de altura h , se observa:

$$F(0)=0, \quad F(1)=1, \quad F(h) = F(h-1) + F(h-2)+1$$

Por la similitud con la sucesión de Fibonacci, a estos árboles se los llama árboles de Fibonacci.

Se demuestra que, si $F(h)=N$, entonces h es aproximadamente $1.44 * (\log N)$

Árbol B

Introducción:

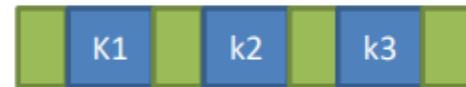
Los B-árboles surgieron en 1972 creados por R.Bayer y E.McCreight. El problema original comienza con la necesidad de mantener índices en almacenamiento externo para acceso a bases de datos, i.e., con el grave problema de la lentitud de estos dispositivos se pretende aprovechar la gran capacidad de almacenamiento para mantener una cantidad de información muy alta organizada de forma que el acceso a una clave sea lo más rápido posible.

Como se ha visto anteriormente existen métodos y estructuras de datos que permiten realizar una búsqueda dentro de un conjunto alto de datos en un tiempo de orden $O(\log_2 n)$.

Mientras que en memoria interna el tiempo de acceso a n datos situados en distintas partes de la memoria es independiente de las direcciones que estos ocupen ($n \cdot \text{cte}$ donde cte es el tiempo de acceso a 1 dato), en memoria externa es fundamental el acceder a datos situados en el mismo bloque para hacer que el tiempo de ejecución disminuya debido a que el tiempo depende fuertemente del tiempo de acceso del dispositivo externo, si disminuimos el número de accesos a disco lógicamente el tiempo resultante de ejecución de nuestra búsqueda se ve fuertemente recortado.

En los árboles B

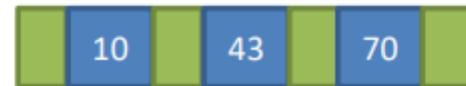
Se puede observar un nodo del árbol B, en un nodo hay un número mayor de claves (k) y de hijos (c).



Donde debe de cumplir:

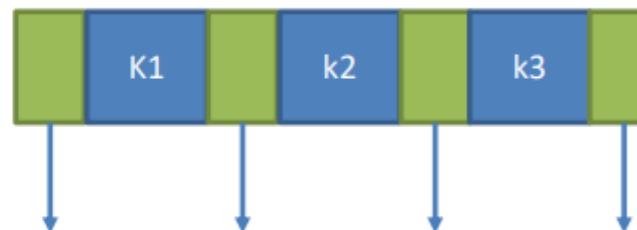
$$v(k_1) < v(k_2) < v(k_3)$$

Ejemplo:



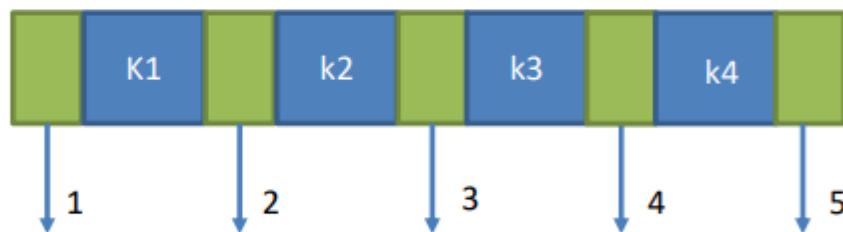
El cual saldrán hijos donde:

- Sean menor que k_1
- Sean valores mayores k_1 y menores que k_2
- Sean valores mayores a k_2 y menores que k_3
- Sean mayores a k_3



Árbol B

- Un árbol B se dice que es de orden m si sus nodos pueden contener hasta un máximo de m hijos.



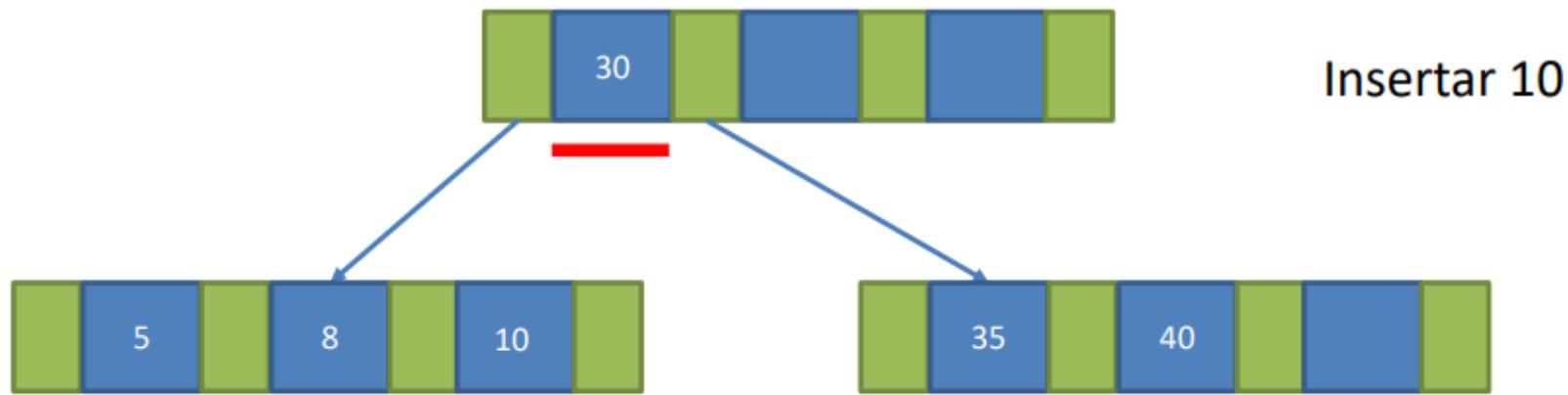
- Es decir que:
 - Un árbol B de orden M tiene M hijos (c).
 - Un árbol B de orden M puede tener $M-1$ claves (k).

Deberá cumplir:

- Todos los nodos excepto la raíz tienen al menos $E(\frac{(m-1)}{2})$ claves.
- Para los nodos interiores eso implica que tienen al menos $E(\frac{(m+1)}{2})$ hijos.
- Todas las hojas están en el mismo nivel.

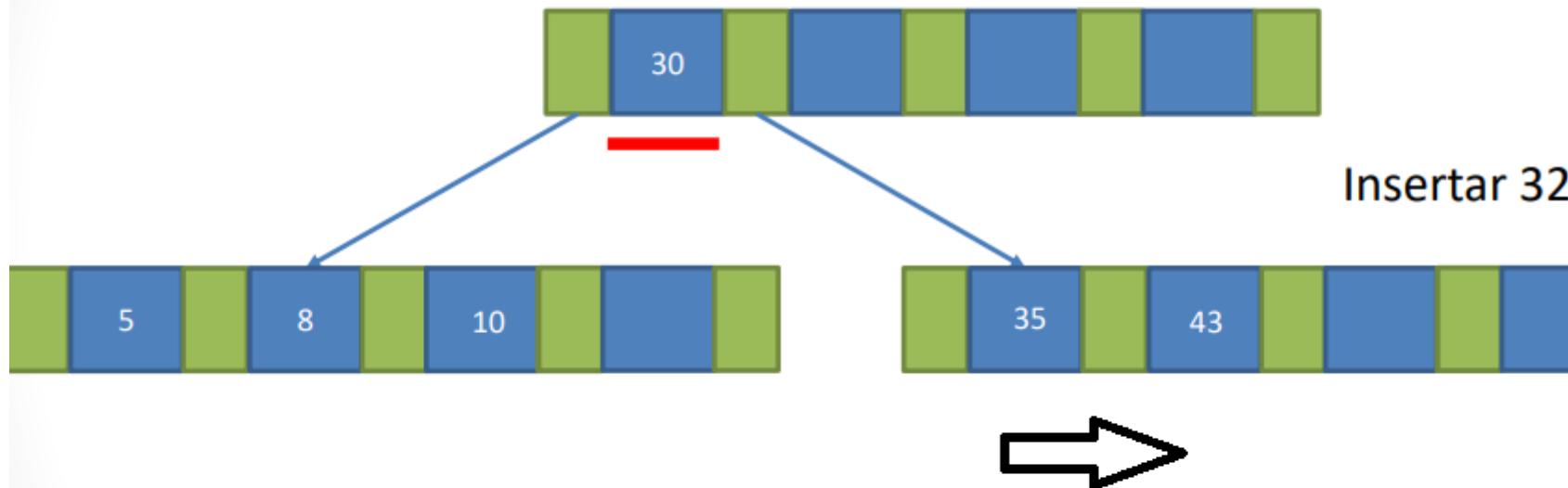
Inserción :

- Se debe de realizar una búsqueda para ver el lugar destino del nuevo elemento.



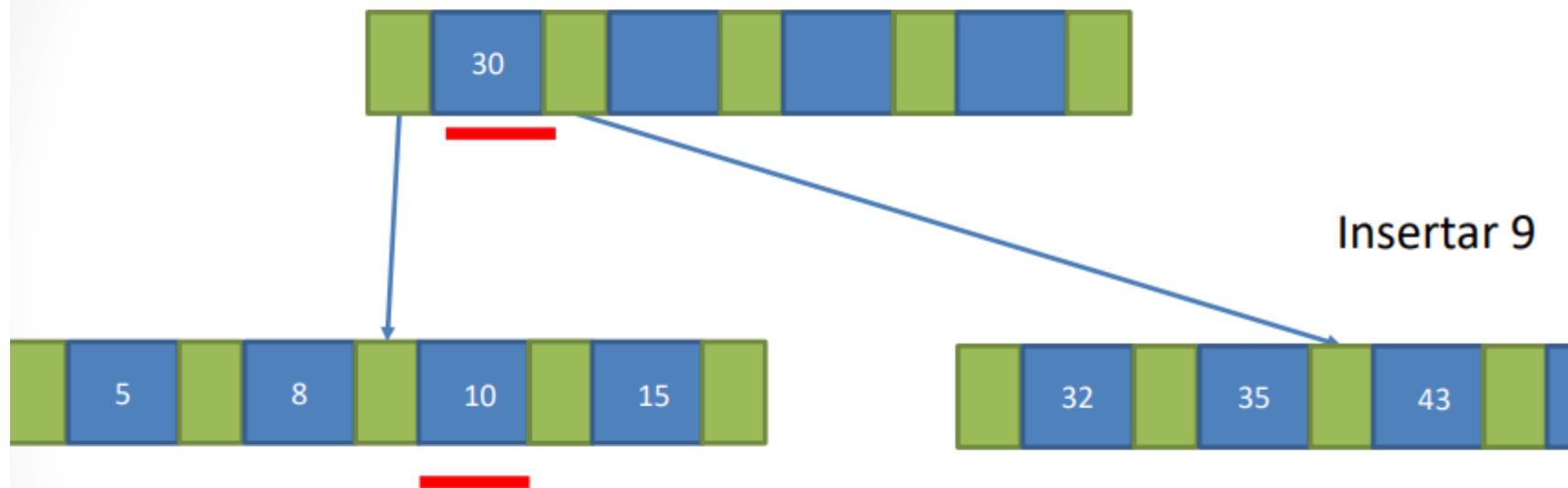
- Se va comparando hasta que se encuentre su ubicación

Inserción :



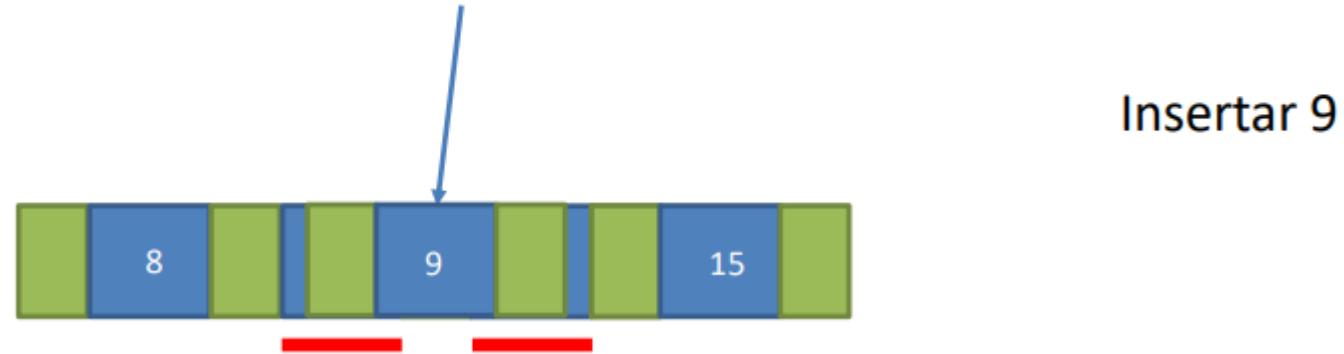
- Se debe reacomodara el nodo.

Inserción :

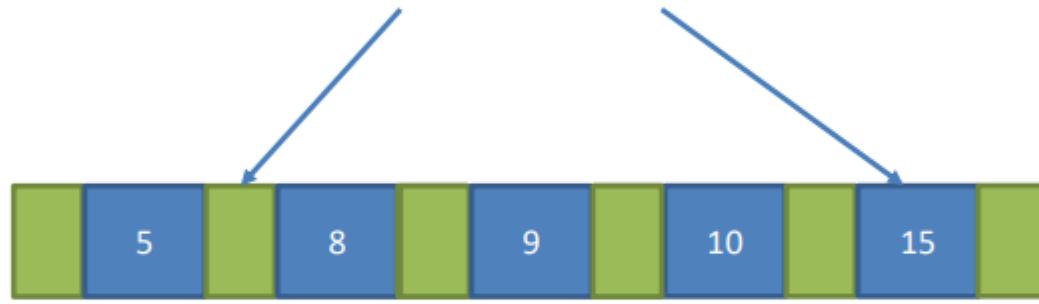


- Se debe reacomodara el nodo.

Inserción

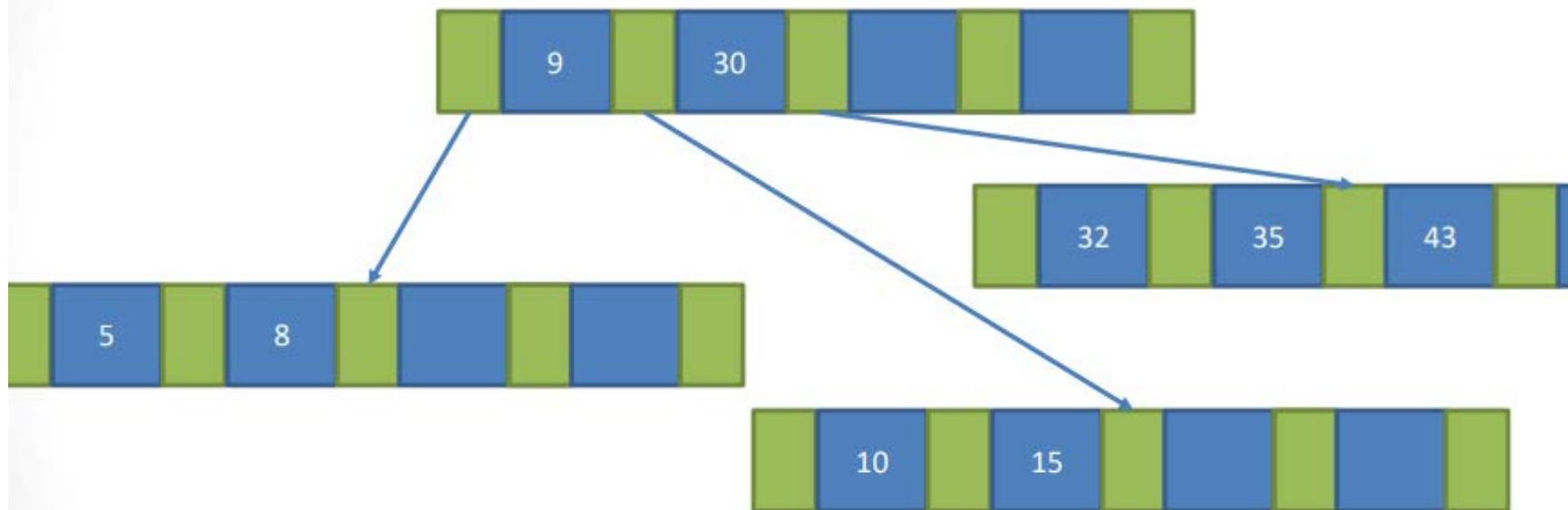


- Se en este momento es un nodo invalido, por lo cual se tiene que crear división

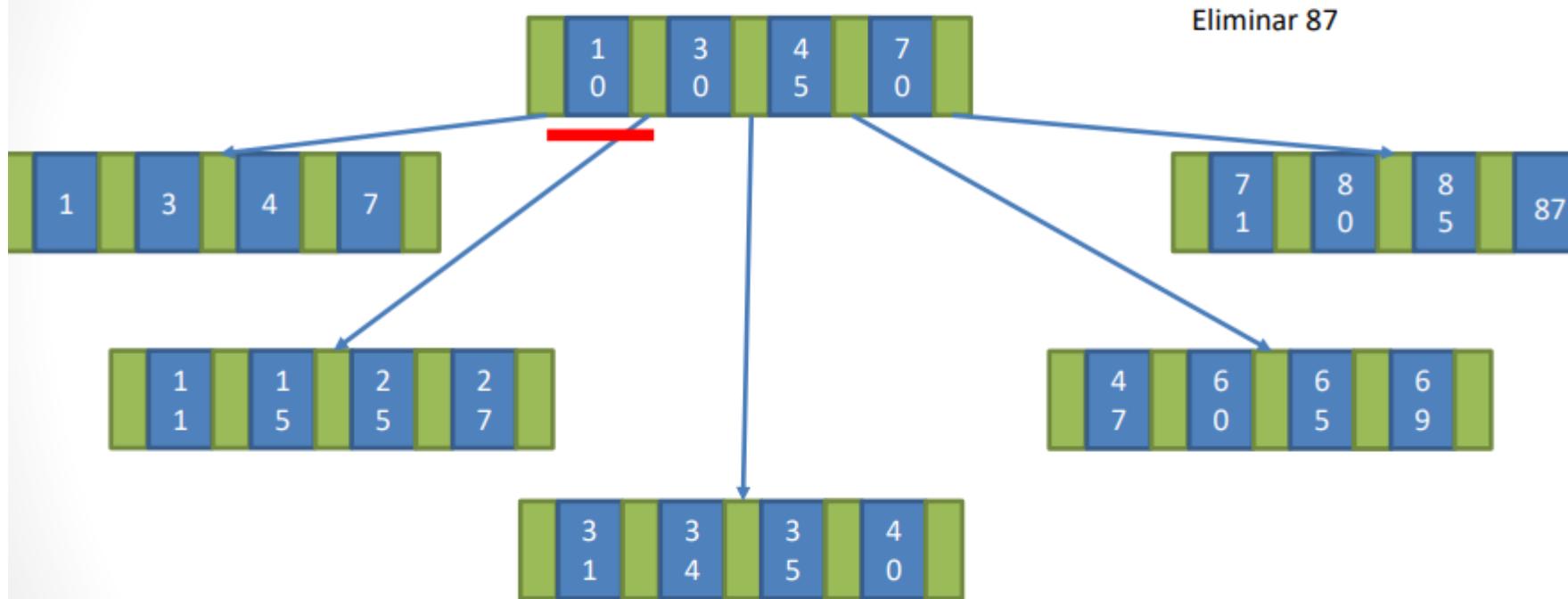


- Haciendo **división**.

Resultado de Insertar 9

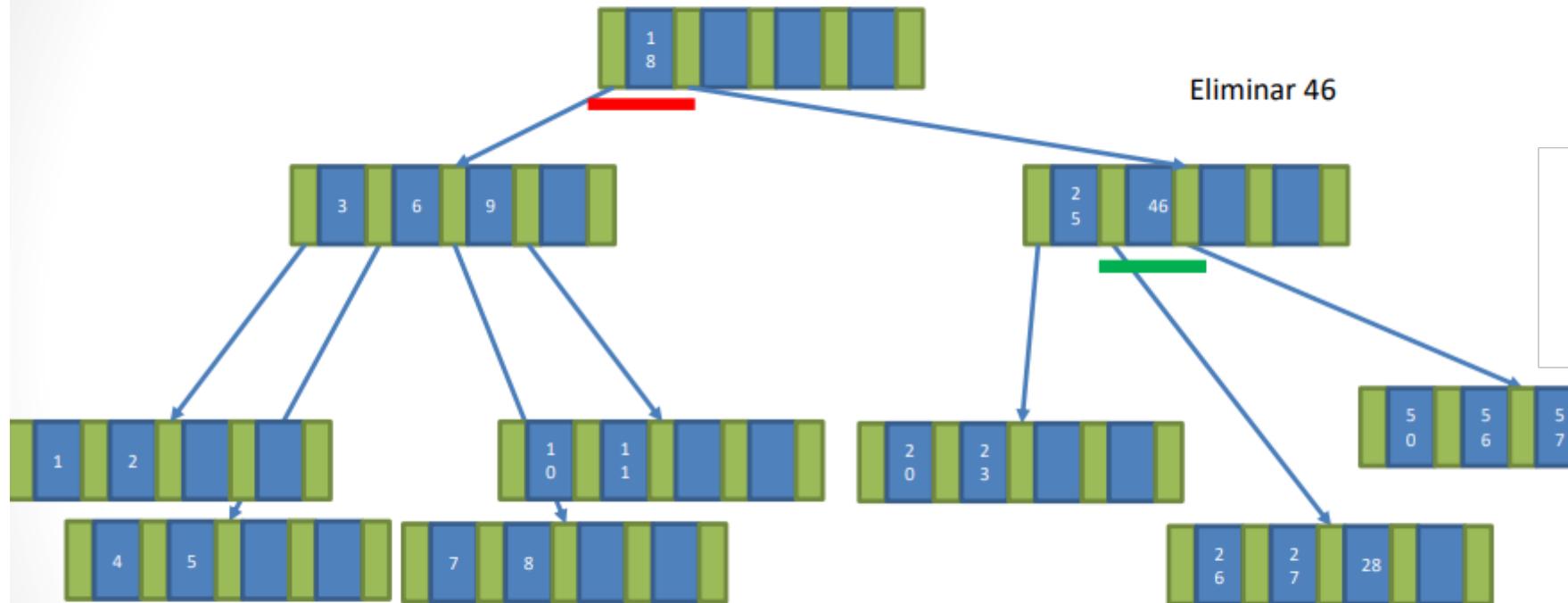


Eliminar



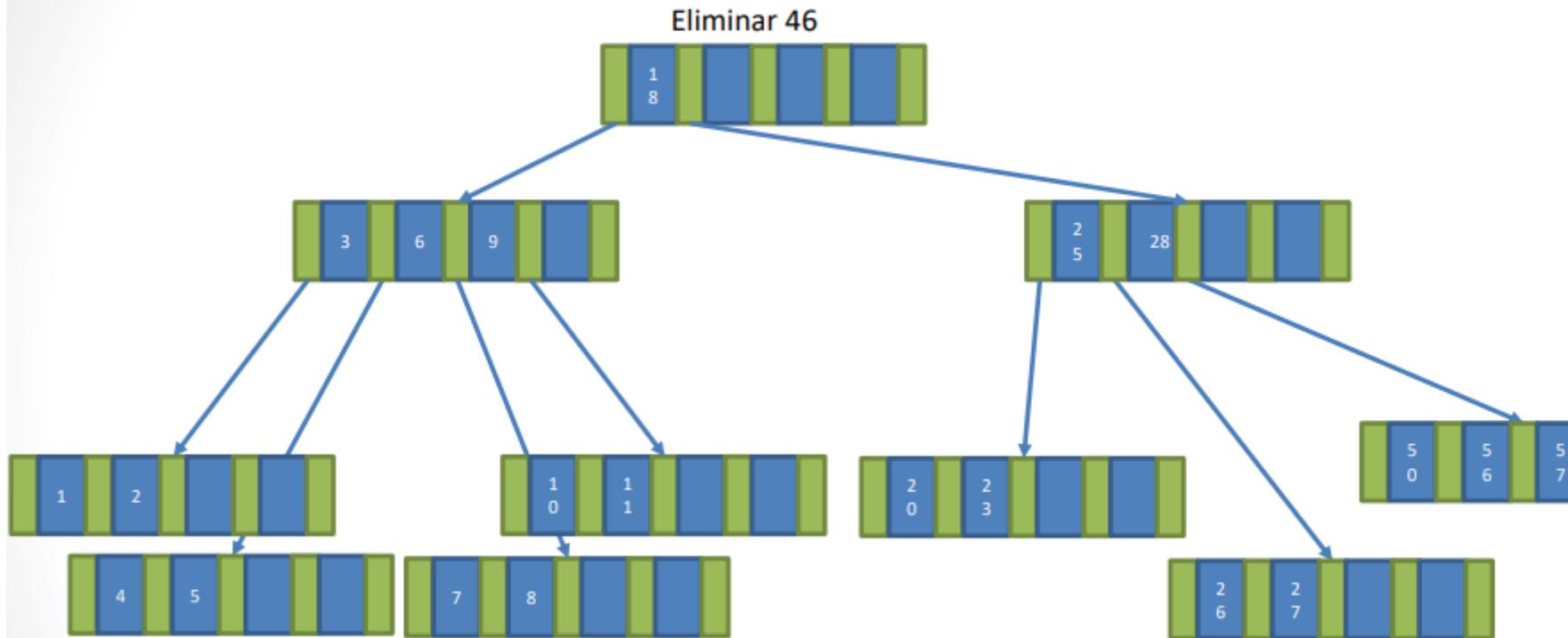
- Se inicia por la raíz
- El nodo sigue siendo valido

Eliminar



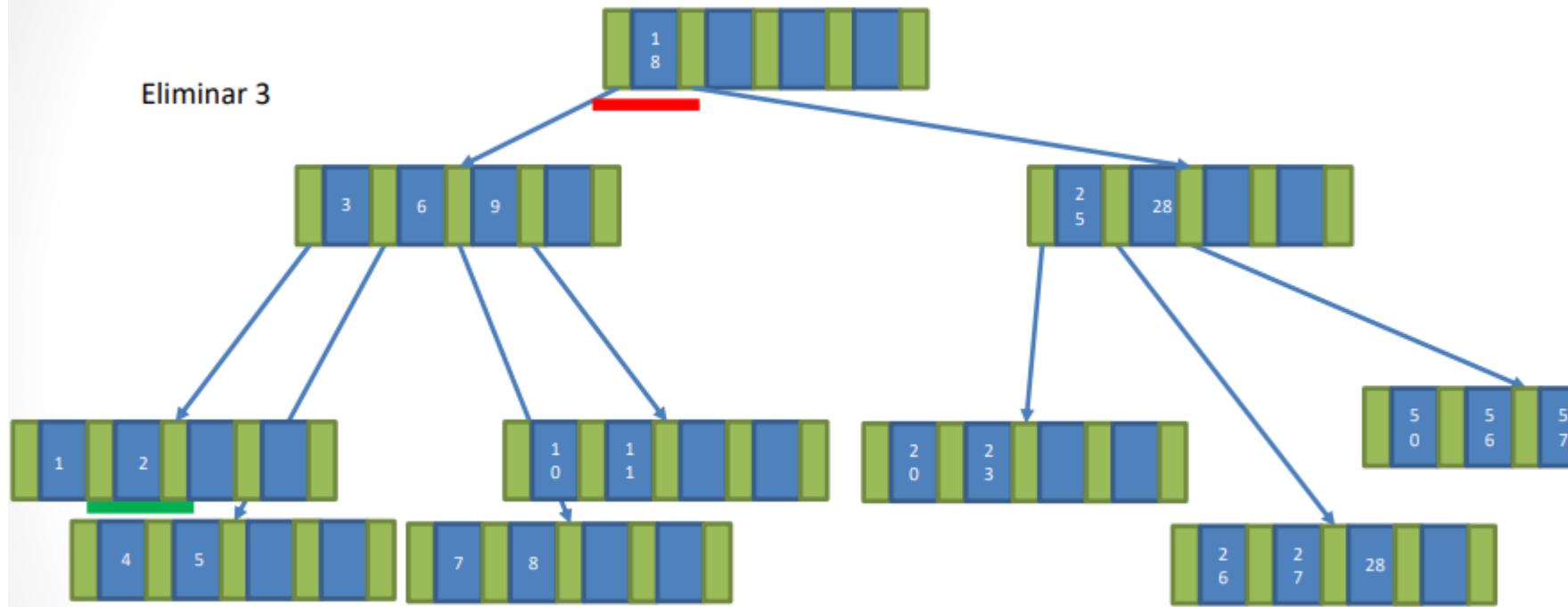
- Se inicia por la raíz
- El dato a eliminar está en un nodo interior que se sustituye por una clave que le antecede o precede en orden ascendente

Resultado



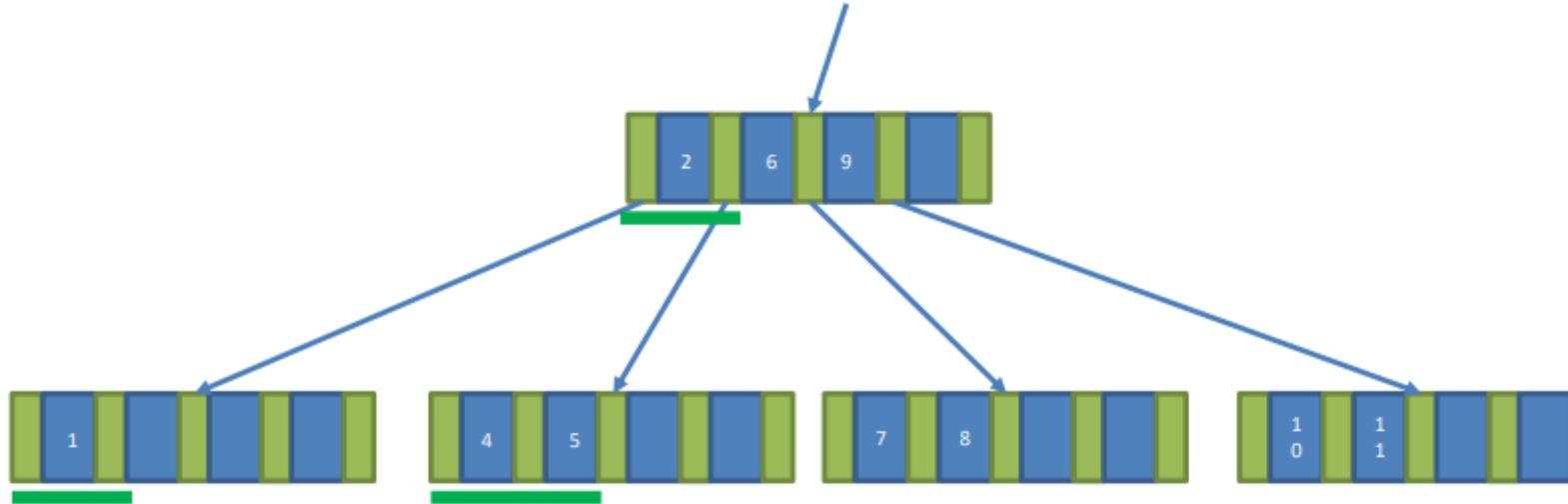
- Se busco el valor que le antecedió para este caso.
- Lo cual el 28 ocupó el lugar del numero eliminado es decir “subió”.
- Los nodos siguen siendo validos.

Eliminar



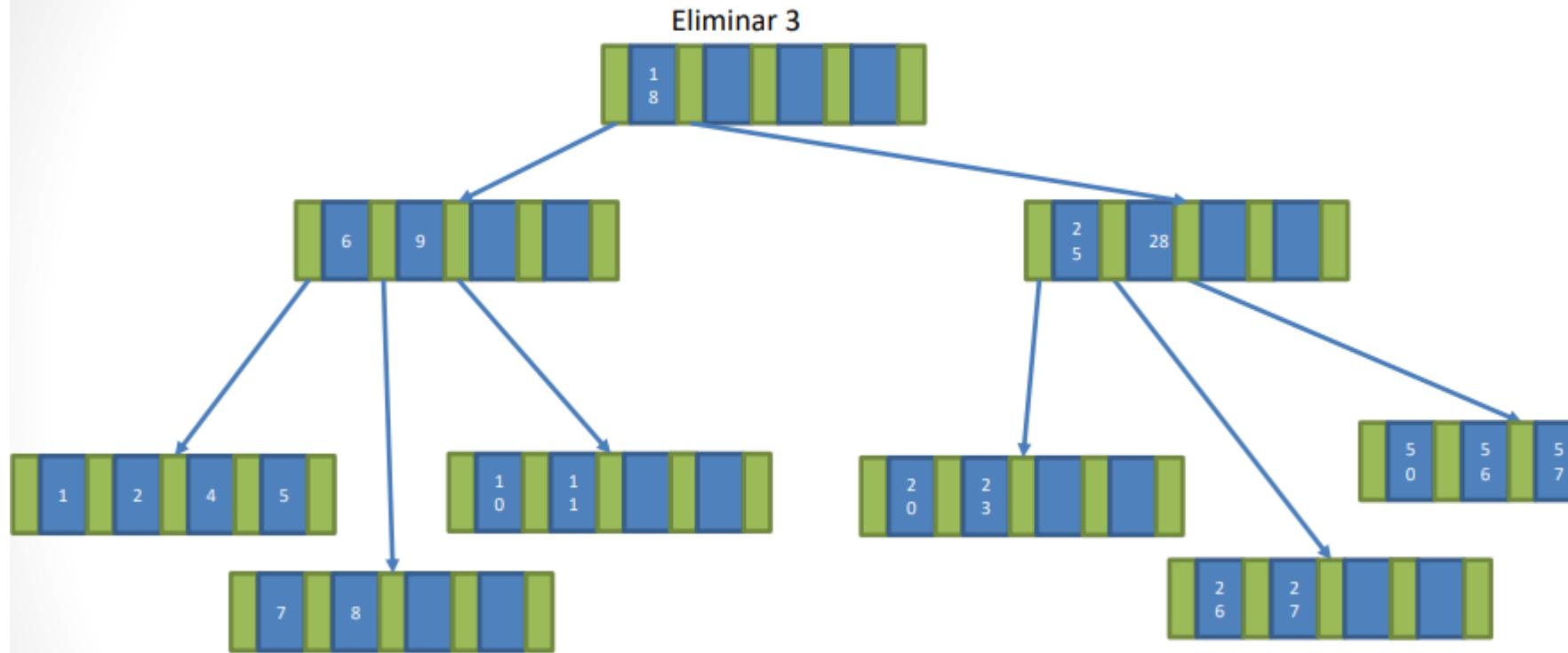
- Se sube el valor que le antecede
- Lo cual provoca que un nodo no sea valido

Fusión



- Se unirán el nodo de la derecha con la izquierda
- Se baja también el padre de los hijos.

Resultado



- Queda un nodo valido.
- Todos los demás nodos cumplen la condición.

Observaciones

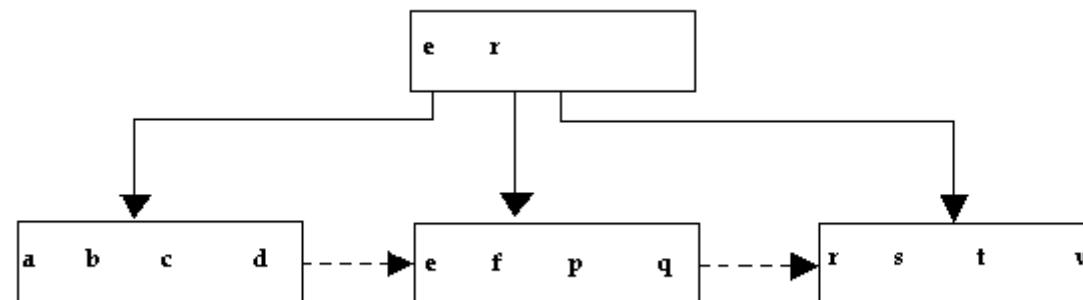
- Los arboles B tienen las mismas operaciones que las de un árbol binario en cambio se agregan, Dividir y Fusionar.
- Cuando el rendimiento en la velocidad de acceso a datos es un problema, se recomienda mas un árbol B.
 - un ejemplo pueden ser las bases de datos.
- Llega a ser mas compleja la organización de esta debido a que necesita un mínimo de hijos y claves.

Simulador: <https://www.cs.usfca.edu/~galles/visualization/BTree>

Árbol B+

Los árboles B+ constituyen otra mejora sobre los árboles B, pues conservan la propiedad de acceso aleatorio rápido y permiten además un recorrido secuencial rápido. En un árbol B+ todas las claves se encuentran en hojas, duplicándose en la raíz y nodos interiores aquellas que resulten necesarias para definir los caminos de búsqueda. Para facilitar el recorrido secuencial rápido las hojas se pueden vincular, obteniéndose, de esta forma, una trayectoria secuencial para recorrer las claves del árbol.

Su principal característica es que todas las claves se encuentran en las hojas. Los árboles B+ ocupan algo más de espacio que los árboles B, pues existe duplicidad en algunas claves. En los árboles B+ las claves de las páginas raíz e interiores se utilizan únicamente como índices.



2. BUSQUEDA EN UN ÁRBOL B+

En este caso, la búsqueda no debe detenerse cuando se encuentre la clave en la página raíz o en una página interior, si no que debe proseguir en la página apuntada por la rama derecha de dicha clave.

3. INSERCIÓN EN UN ÁRBOL B+

Su diferencia con el proceso de inserción en árboles B consiste en que cuando se inserta una nueva clave en una página llena, ésta se divide también en otras dos, pero ahora la primera contendrá con $m/2$ claves y la segunda $1+m/2$, y lo que subirá a la página antecesora será una copia de la clave central.

4. BORRADO EN UN ÁRBOL B+

La operación de borrado debe considerar:

- Si al eliminar la clave(siempre en una hoja)el número de claves es mayor o igual a $m/2$ el proceso ha terminado. Las claves de las páginas raíz o internas no se modifican aunque sean una copia de la eliminada,pues siguen constituyendo un separador válido entre las claves de las páginas descendientes.
- Si al eliminar la clave el número de ellas en la página es menor que $m/2$ será necesaria una fusión y redistribución de las mismas tanto en las páginas hojas como en el índice.

Simulador: <https://www.cs.usfca.edu/~galles/visualization/BPlusTree>

Fin