

Estructuras de Datos

Dr. Martin Gonzalez-Rodriguez

ISBN 978-1-365-00699-9

© 2012 – 2016 Martín González Rodríguez

Árboles

Dr. Martin Gonzalez-Rodriguez

Estructuras de Datos Jerárquicas

Objetivo

- ❖ Modelar relaciones de orden y/o de clasificación entre elementos.
 - Jerarquías sociales (ejército, iglesia, organigrama empresarial, etc.).
 - Modelado de gramáticas (árboles sintácticos, árboles léxicos.)
 - Modelos informáticos (jerarquía de clases, sistemas de ficheros, etc.).
 - Sistemas de clasificación (rangos taxonómicos, árboles filogenéticos, genealógicos, deportivos, etc.).

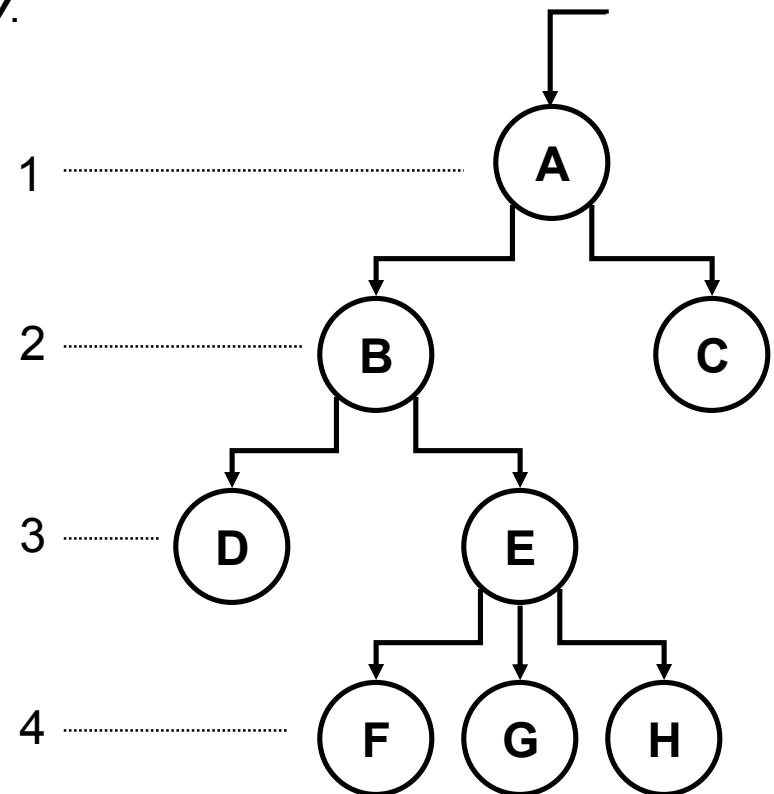
Conceptos Básicos

¿Qué es un Árbol?

- ❖ En informática¹ llamamos **árbol** a un grafo conexo sin ciclos con raíz.
 - Dado un nodo llamado **raíz** y cualquier otro vértice **v**, sólo existe un camino dirigido desde el nodo raíz al nodo **v**.

Elementos Básicos de un Árbol

1. Raíz.
2. Hijo (descendiente directo).
3. Padre (ascendiente directo).
4. Hoja (nodo terminal).
5. Nodo interior.
6. Grado de un nodo.
7. Grado de un árbol.
8. Nivel de un nodo.
9. Altura (profundidad).

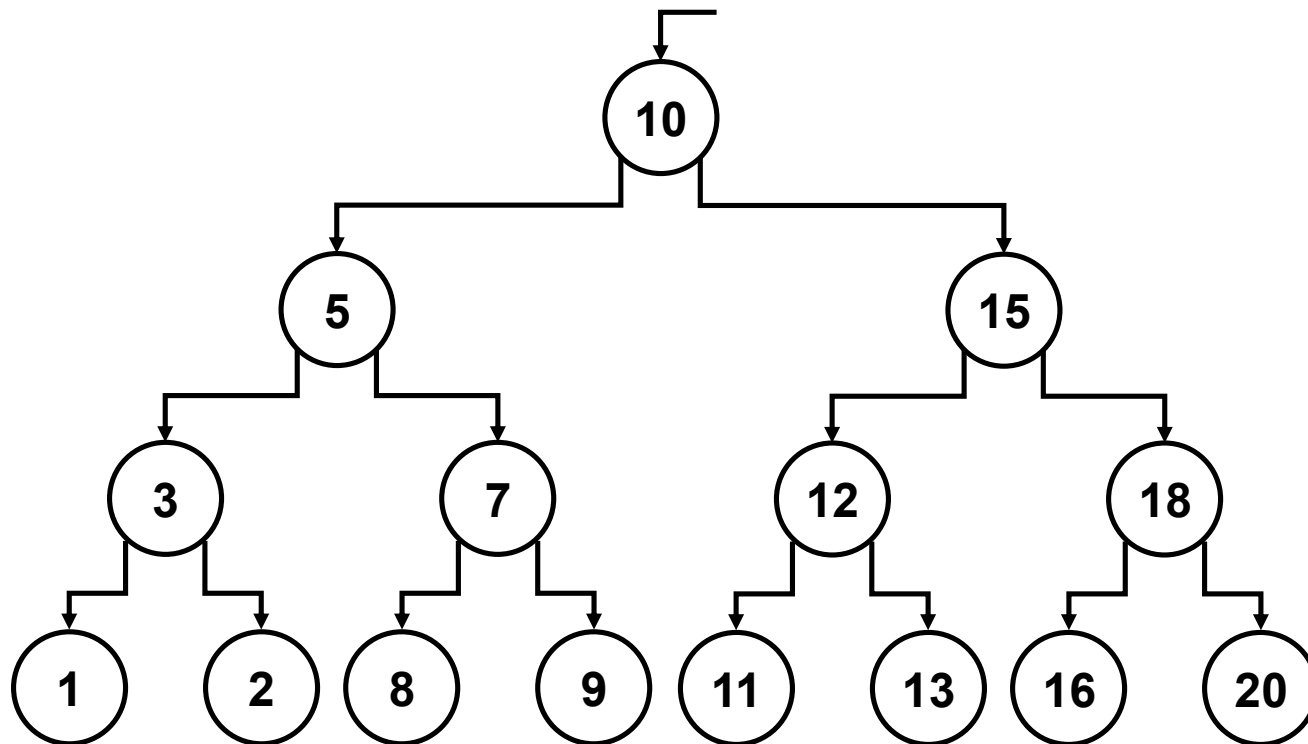


¹En matemáticas, una arborescencia es un árbol libre (grafo conexo sin ciclos) con raíz.

Conceptos Básicos

Árbol Completo

- ❖ Aquel que tiene el máximo número de nodos posible para su **altura h** y **grado g**.
 - Es aquel que tiene todos los niveles llenos.
 - Máximo nivel de eficiencia en las **búsquedas desde la raíz**.



$$n = 2^h - 1$$

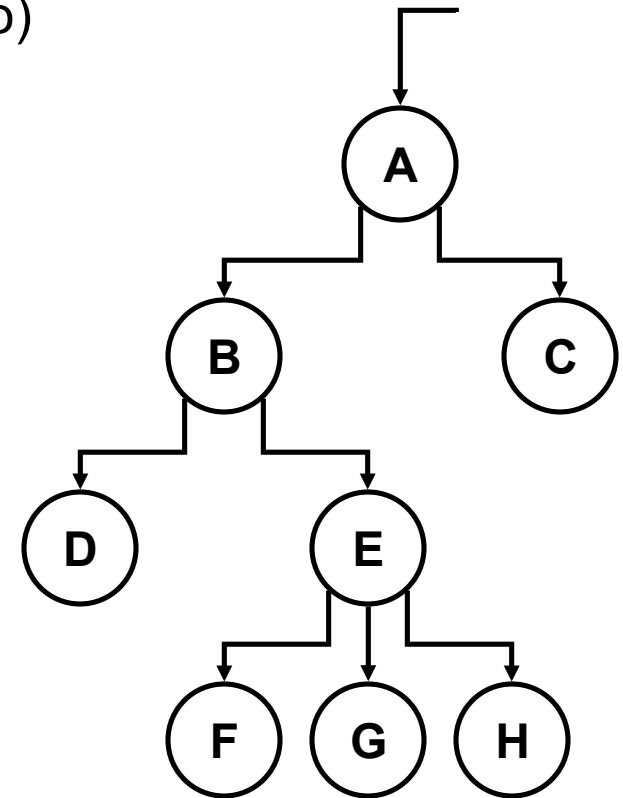
$$\log_2(n + 1) = h$$

Métricas de Eficiencia

Caminos de Búsqueda

❖ CI: Camino Interno (nodo encontrado)

- Buscar A = 1.
- Buscar B y C = 2 c/u = 4.
- Buscar D y E = 3 c/u = 6.
- Buscar F, G y H = 4 c/u = 12.
- Total = 23.
 - » Para 8 nodos = $23 / 8 = L_mCI = 2,87$

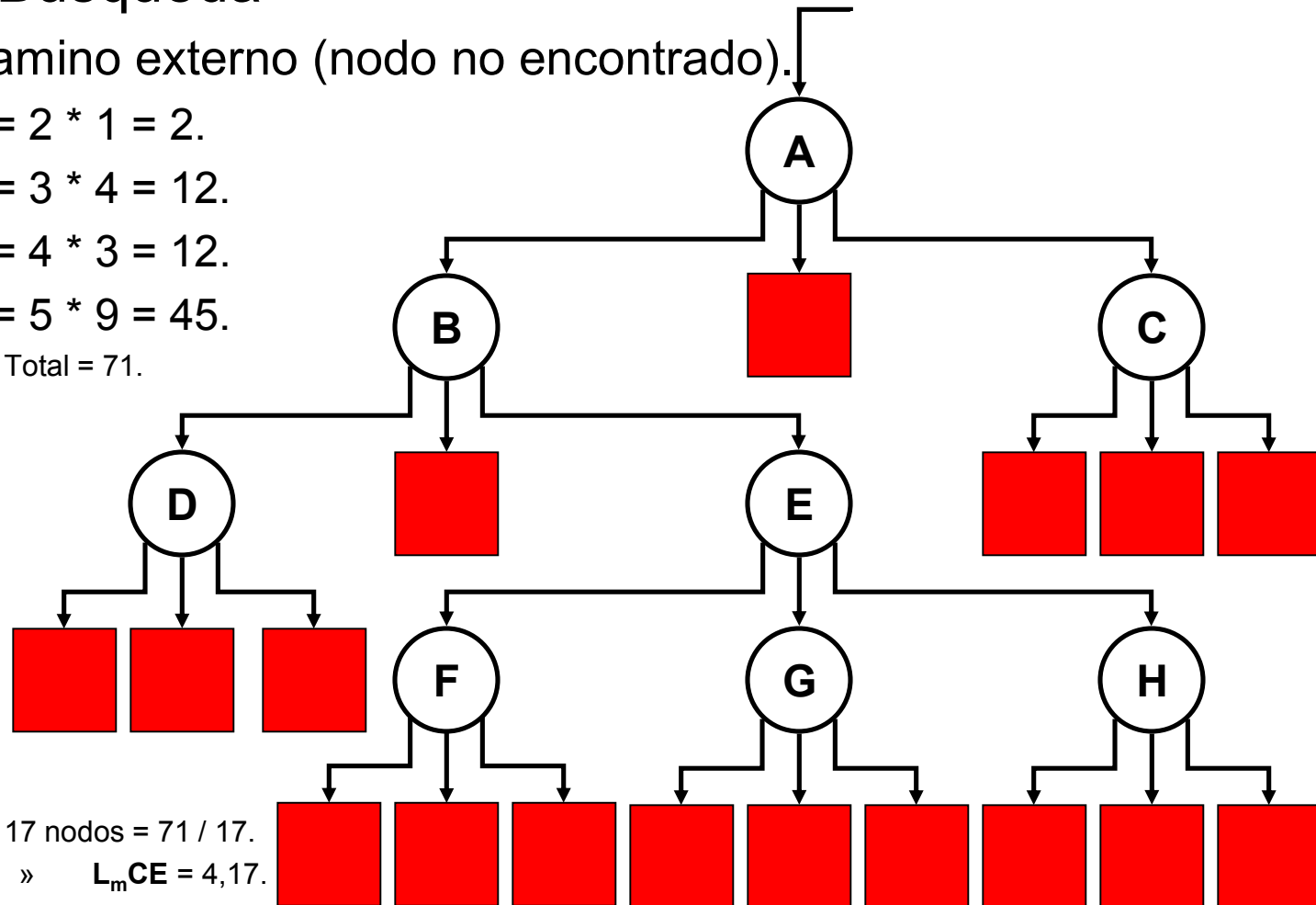


Métricas de Eficiencia

Caminos de Búsqueda

❖ CE: Camino externo (nodo no encontrado).

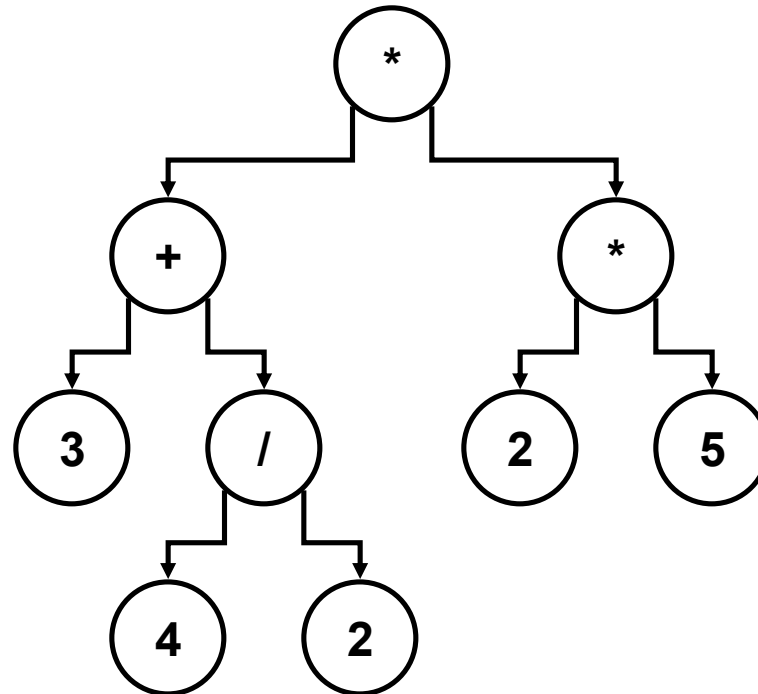
- $N_2 = 2 * 1 = 2$.
- $N_3 = 3 * 4 = 12$.
- $N_4 = 4 * 3 = 12$.
- $N_5 = 5 * 9 = 45$.
- Total = 71.



Arbol Binario

Árbol de grado 2

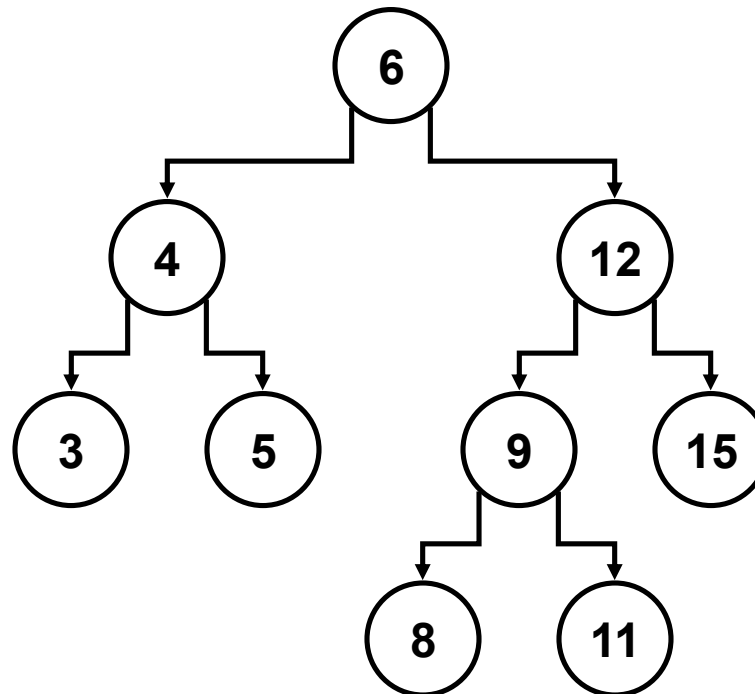
- ❖ Modela relaciones de jerarquía entre pares de elementos con respecto a un nodo superior
 - Árboles genealógicos.
 - Competiciones de copa.
 - Operadores binarios.



Árbol Binario de Búsqueda (ABB)

Árbol ordenado para facilitar operaciones de búsqueda

- ❖ Se cumple que para cada nodo...
 - **Subárbol izquierdo:** contiene elementos con **clave menor** que la del nodo padre.
 - **Subárbol derecho:** contiene elementos con **clave mayor** que la del nodo padre.



Árbol Binario de Búsqueda (ABB)

Estructura y operaciones básicas

Class BSTNode

```
public class BSTNode <T extends Comparable <T>>
{
    private T element;
    private BSTNode<T> left;
    private BSTNode<T> right;
}
```

❖ Operaciones básicas

- Add.
- Search.
- Remove.
- toString.

Árbol Binario de Búsqueda (ABB)

Insertar

❖ Procedimiento recursivo

- Caso General 1:
 - Si clave del nodo a insertar **es menor** que la clave del nodo actual, **insertar nodo por la izquierda**.
- Caso General 2:
 - Si clave del nodo a insertar **es mayor** que la clave del nodo actual, **insertar nodo por la derecha**.
- Caso de Parada 1:
 - Si clave del nodo a insertar **es igual** que la clave del nodo actual, **el nodo ya existe, lanzar excepción** pues **no se permiten claves repetidas**.
- Caso de Parada 2:
 - Si el nodo actual **es igual a null**. **Se ha alcanzado una hoja, crear nodo a insertar y devolver**.

Árbol Binario de Búsqueda (ABB)

add

```
private BSTNode<T> add (BSTNode<T> theRoot, T element){
    if (theRoot == null)
        return new BSTNode<T>(element);

    if (element.compareTo(theRoot.getElement()) == 0)
        throw new RuntimeException("element already exists!");

    if (element.compareTo(theRoot.getElement()) < 0)
        theRoot.setLeft (add(theRoot.getLeft(), element));

    if (element.compareTo(theRoot.getElement()) > 0)
        theRoot.setRight (add(theRoot.getRight(), element));
}
```

CLASSWORK

PLAYGROUND

❖ **Ejercicio ABB 1.** partiendo de un árbol binario de búsqueda vacío...

- a) Inserte la secuencia de nodos: 5, 7, 9, 3, 1, 2, 6.
 - Analice la complejidad temporal de cada inserción.

❖ **Ejercicio ABB 2.** partiendo de un árbol binario de búsqueda vacío...

- a) Inserte la secuencia de nodos: 7, 6, 5, 4, 3, 2, 1.
 - Analice la complejidad temporal de cada inserción.
- b) Inserte el nodo 8.
 - Analice la complejidad temporal de la inserción.

Complejidad caso mejor: $O(1)$

Complejidad caso peor: $O(n)$

Árbol Binario de Búsqueda

Search

```
private boolean search (BSTNode<T> theRoot, T element)
{
    if (theRoot == null)
        return false;
    else
        if (element.compareTo(theRoot.getElement()) == 0)
            return true;
        else
            if (element.compareTo(theRoot.getElement()) < 0)
                return search(theRoot.getLeft(), element);
            else
                if (element.compareTo(theRoot.getElement()) > 0)
                    return search (theRoot.getRight(), element);
}
```

Complejidad caso mejor: $O(1)$

Complejidad caso peor: $O(n)$

Árbol Binario de Búsqueda

Remove

```
private BSTNode<T> remove (BSTNode<T> theRoot, T element)
{
    if (theRoot == null)
        throw new RuntimeException("element does not exist!");
    else
        if (element.compareTo(theRoot.getElement()) < 0)
            theRoot.setLeft(remove (theRoot.getLeft(), element));
        else
            if (element.compareTo(theRoot.getElement()) > 0)
                theRoot.setRight(remove (theRoot.getRight(), element));
            else {
                // nodo encontrado
                // ¿Cómo se borra?
            }
    return theRoot;
}
```

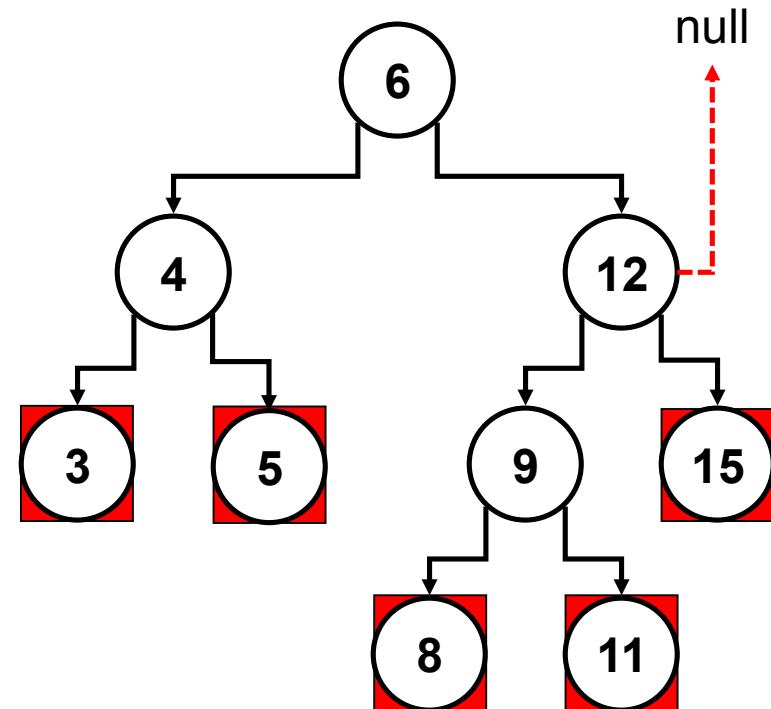

Árbol Binario de Búsqueda

Casos particulares del borrado

- ❖ Caso I: Borrar elemento sin hijos (hojas).
 - La referencia se anula (igual a *null*).

ZOOM IN

```
else {  
  
    if (theRoot.getLeft() == null &&  
        theRoot.getRight() == null)  
        return(null);  
  
}
```



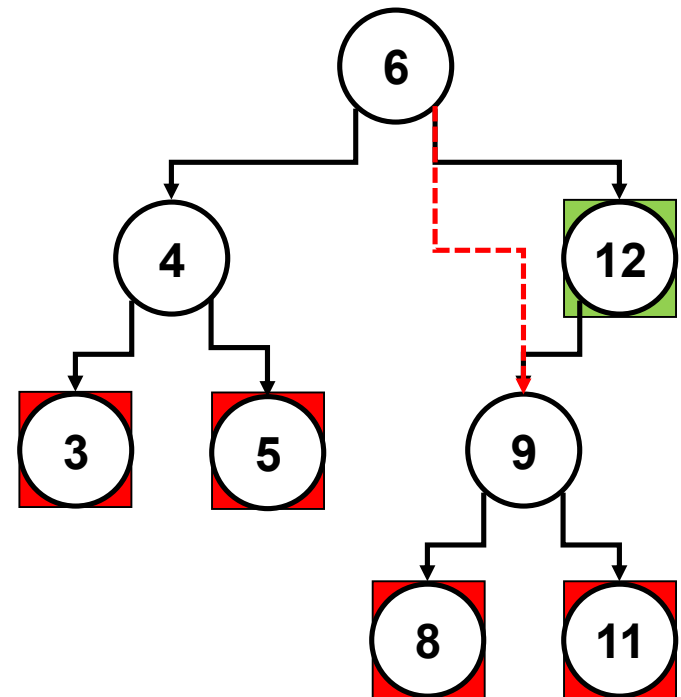
Árbol Binario de Búsqueda

Casos particulares del borrado

- ❖ Caso II: Borrar elemento con un solo hijo.
 - La referencia se reasigna al único hijo que tenga el nodo.

ZOOM IN

```
else {  
    if (theRoot.getLeft() == null)  
        return theRoot.getRight();  
    else  
        if (theRoot.getRight() ==  
null) return theRoot.getLeft();  
}
```



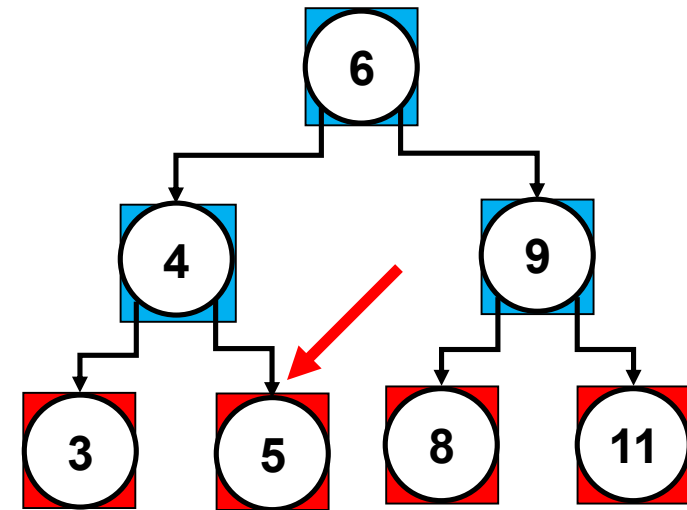
Árbol Binario de Búsqueda

Casos particulares del borrado

- ❖ Caso III: Borrar elemento con dos hijos.
 - Substituir el contenido del nodo por el del nodo mayor de su subárbol izquierdo (pivote).
 - Borrar el pivote (se presentará el caso I o caso II pero nunca el caso III).

ZOOM IN

```
else {  
    if (theRoot.getLeft() == null)  
        return theRoot.getRight();  
    else  
        if (theRoot.getRight() == null)  
            return theRoot.getLeft();  
        else {  
            theRoot.setElement(getMax(theRoot.get  
Left()));  
  
        }  
}
```



Árbol Binario de Búsqueda

getMax

```
private T getMax(BSTNode<T> theRoot)
{
    if (theRoot == null)
        return null;
    else
        if (theRoot.getRight () == null)
            return theRoot.getElement();
        else
            return getMaxRec(theRoot.getRight());
}
```

Árbol Binario de Búsqueda

getMax

```
private T getMax(BSTNode<T> theRoot)
{
    while (theRoot.getRight() != null)
        theRoot = theRoot.getRight();

    return theRoot.getElement();
}
```

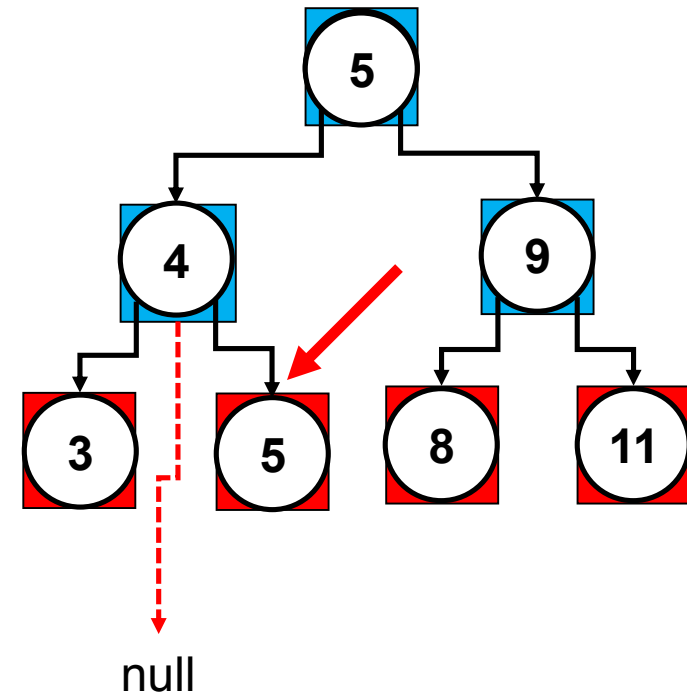
Árbol Binario de Búsqueda

Casos particulares del borrado

- ❖ Caso III: Borrar elemento con dos hijos.
 - Substituir el contenido del nodo por el del nodo mayor de su subárbol izquierdo (pivote).
 - Borrar el pivote (se presentará el caso I o caso II pero nunca el caso III).

ZOOM IN

```
else {  
    if (theRoot.getLeft() == null)  
        return theRoot.getRight();  
    else  
        if (theRoot.getRight() == null)  
            return theRoot.getLeft();  
        else {  
            theRoot.setElement(getMax(theRoot.getLeft())  
);  
            theRoot.setLeft(remove (theRoot.getLeft(),  
theRoot.getElement()));  
        }  
}
```



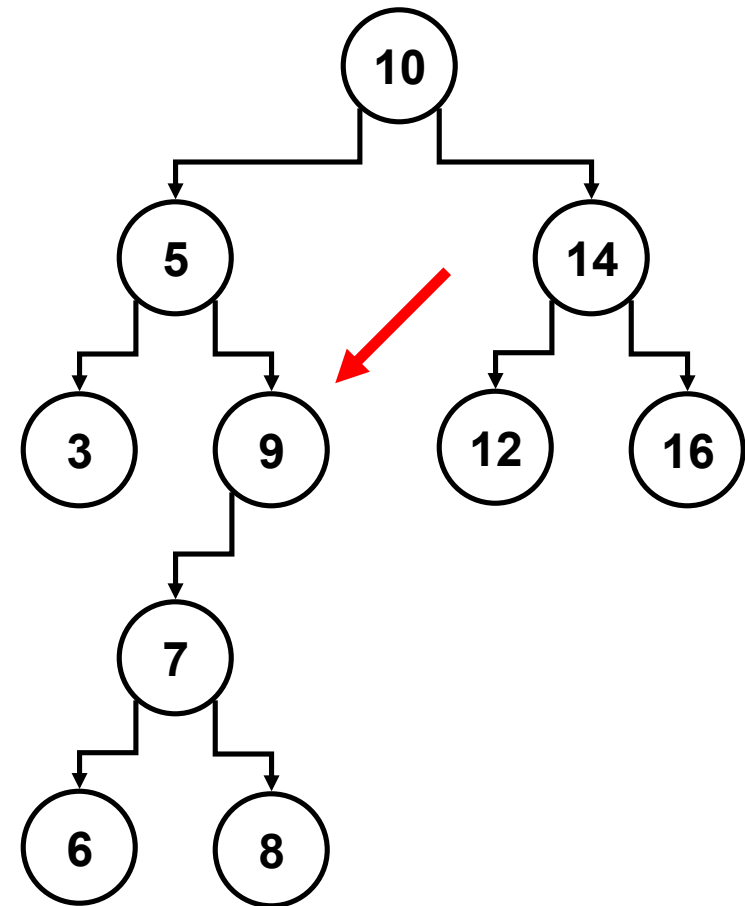
Árbol Binario de Búsqueda

Casos particulares del borrado

❖ Caso III: Borrar elemento con dos hijos.

ZOOM IN

```
else {  
    if (theRoot.getLeft() == null) return  
    theRoot.getRight();  
    else  
        if (theRoot.getRight() == null)  
            return theRoot.getLeft();  
        else {  
theRoot.setElement(getMax(theRoot.getLeft()  
t()));  
    theRoot.setLeft(remove  
(theRoot.getLeft(), theRoot.getElement()  
));}}
```



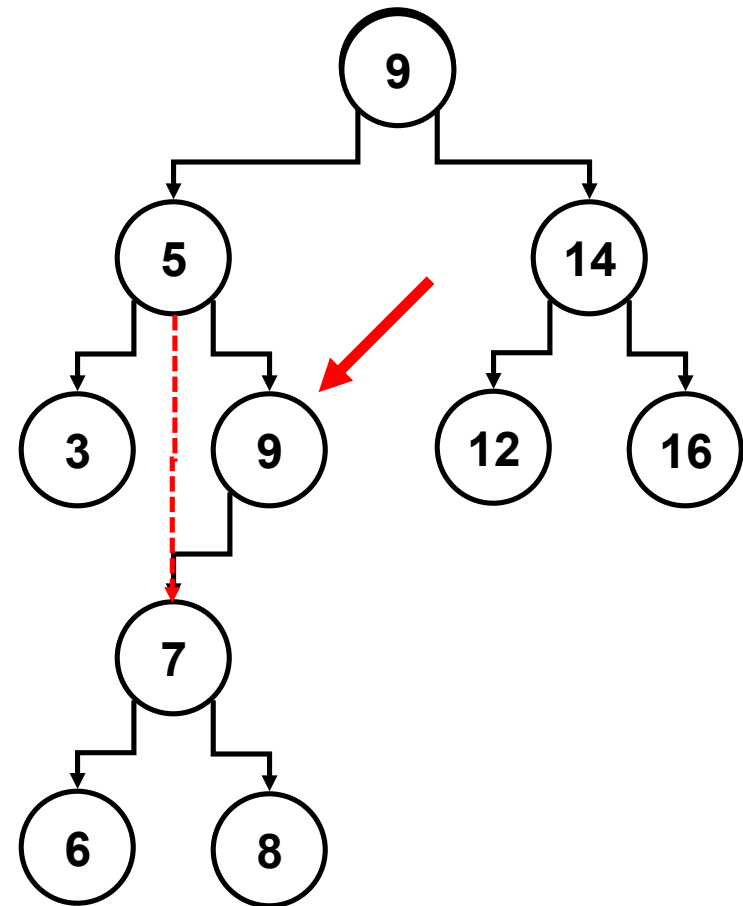
Árbol Binario de Búsqueda

Casos particulares del borrado

❖ Caso III: Borrar elemento con dos hijos.

ZOOM IN

```
else {  
    if (theRoot.getLeft() == null) return  
    theRoot.getRight();  
    else  
        if (theRoot.getRight() == null)  
            return theRoot.getLeft();  
        else {  
theRoot.setElement(getMax(theRoot.getLeft  
()));  
    theRoot.setLeft(remove  
(theRoot.getLeft(), theRoot.getElement()))  
;}}
```



Árbol Binario de Búsqueda

Remove (Pseudocode)

```
private BSTNode<T> remove (BSTNode<T> theRoot, T element){
    if (theRoot == null)
        throw new RuntimeException("element does not exist!");
    else
        if (element.compareTo(theRoot.getElement()) < 0)
            theRoot.setLeft(remove (theRoot.getLeft(), element));
        else
            if (element.compareTo(theRoot.getElement()) > 0)
                theRoot.setRight(remove (theRoot.getRight(), element));
            else {
                if (theRoot.getLeft() == null) return theRoot.getRight();
                else
                    if (theRoot.getRight() == null) return theRoot.getLeft();
                else {
                    theRoot.setElement(getMax(theRoot.getLeft()));
                    theRoot.setLeft(remove (theRoot.getLeft(),
theRoot.getElement()));
                }}
    return theRoot; }
```

Complejidad caso mejor: $O(1)$

Complejidad caso peor: $O(n)$

Árbol Binario de Búsqueda

Recorrido de un Árbol Binario

❖ preorden

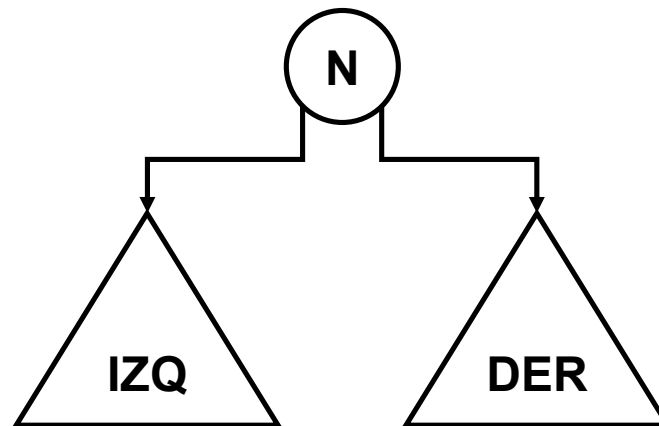
- **Primero** se analiza el nodo, luego subárboles.
 - N-IZQ-DER ó N-DER-IZQ.

❖ inorden

- El nodo se analiza **entre** los dos subárboles.
 - IZQ-N-DER ó DER-N-IZQ.

❖ postorden

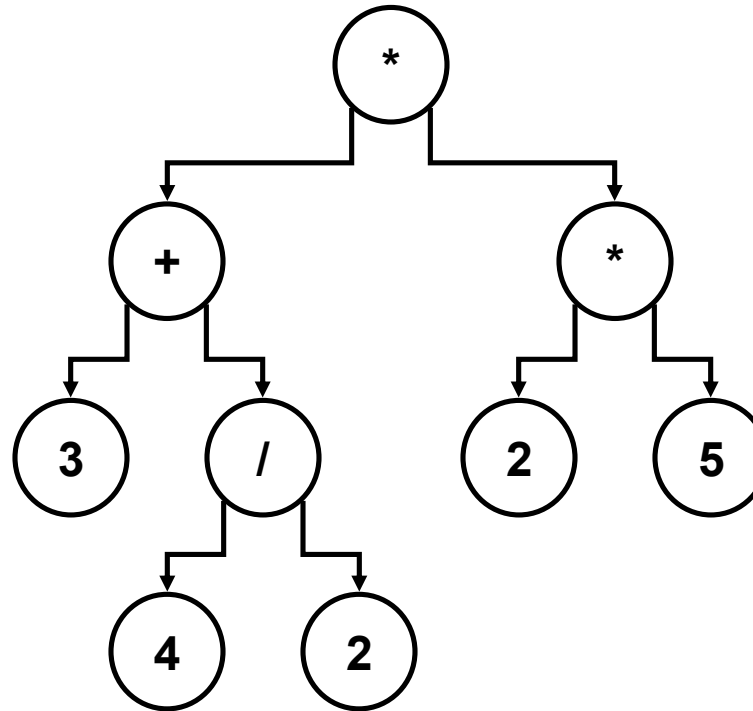
- El nodo se analiza **después** de los subárboles.
 - IZQ-DER-N ó DER-IZQ-N.



Árbol Binario de Búsqueda

Ejercicio

- ❖ Recorrer el árbol

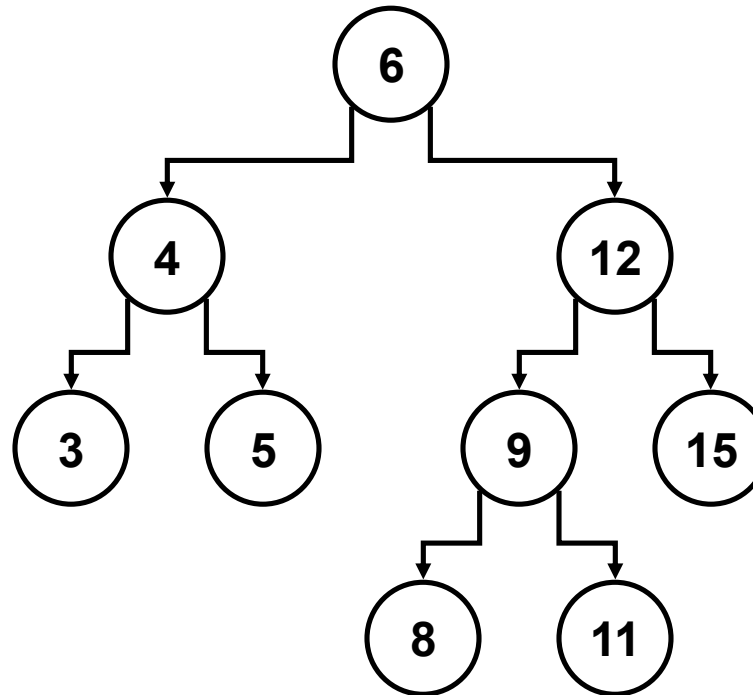


- **preorden:** * + 3 / 4 2 * 2 5 (prefija).
- **inorden:** 3 + 4 / 2 * 2 * 5 (infija).
- **postorden:** 3 4 2 / + 2 5 * * (polaca inversa)

Árbol Binario de Búsqueda

PLAYGROUND

❖ Recorrer el árbol



- **preorden:** 6, 4, 3, 5, 12, 9, 8, 11, 15 (prefija).
- **inorden:** 3, 4, 5, 6, 8, 9, 11, 12, 15 (infija).
- **postorden:** 3, 5, 4, 8, 11, 9, 15, 12, 6 (polaca inversa).

Árbol Binario de Búsqueda

toString (recorrido preorden)

```
private String toString (BSTNode<T> theRoot)
{
    if (theRoot != null)
        return (theRoot.toString()
                + toString(theRoot.getLeft())
                + toString(theRoot.getRight()));
    else
        return ("-");
}
```

Complejidad caso mejor: $O(n)$

Complejidad caso peor: $O(n)$

Árbol Binario de Búsqueda

Eficiencia de un árbol binario de búsqueda

- ❖ La eficiencia de este tipo de árboles depende ampliamente de su altura
 - Rango de h : $[\log_2 n, n]$

Método	Complejidad caso mejor	Complejidad caso peor
Insertar	$O(1)$	$O(n)$
Buscar	$O(1)$	$O(n)$
Borrar	$O(1)$	$O(n)$
print	$O(n)$	$O(n)$

- ❖ Objetivo
 - Minimizar la altura del árbol, evitando árboles degenerados.

HOMEWORK

PLAYGROUND

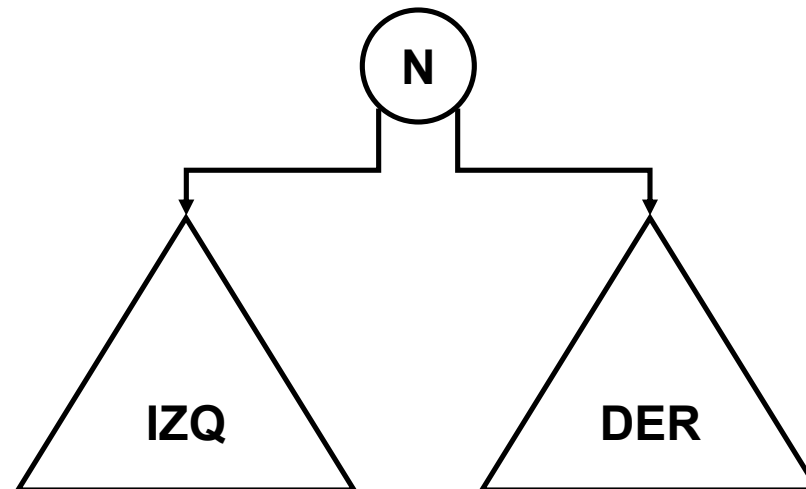
- ❖ Consulte la entrada para los **Árboles Binarios de Búsqueda** en la Wikipedia
 - Estudie **cuidadosamente** todo el contenido de la entrada poniendo especial atención a la **Búsqueda del Árbol Óptimo**.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Árboles Perfectamente Equilibrados (APE)

❖ Garantiza **altura** mínima en un árbol binario

- **Condición:** Para todo nodo n , $|\#_{\text{izq}} - \#_{\text{der}}| \leq 1$.
 - $\#_{\text{izq}}$ = número de nodos del subárbol izquierdo.
 - $\#_{\text{der}}$ = número de nodos del subárbol derecho.



❖ Todo árbol APE es de altura mínima, pero...

- ¿Todo árbol de altura mínima es APE?

ABB vs APE

Inserción y Borrado de elevada Complejidad Temporal

- ❖ Es necesario borrar y reconstruir el árbol después de cada operación.

Método	ABB (caso peor)	APE (cualquier caso)
Insertar	$O(n)$	$O(n)$
Buscar	$O(n)$	$O(\log_2 n)$
Borrar	$O(n)$	$O(n)$

- ❖ El uso de un APE tiene sentido **sólo cuando** el número de búsquedas es **infinitamente superior** al resto de operaciones.

Árboles AVL

Problema a Resolver

- ❖ Diseñar un árbol capaz de tener complejidad temporal $\log_2(n)$ en el peor de los casos para las tres operaciones básicas
 - Insertar, Buscar y Borrar.

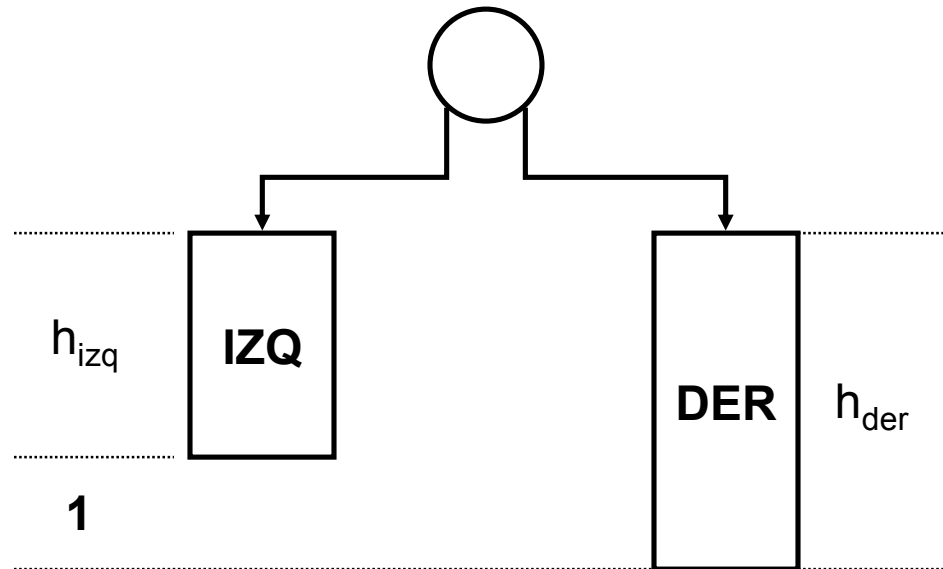
- ❖ Desarrollado por los soviéticos Georgii Adelson-Velskii y Yevgeni Landis en 1962.

Árboles AVL

Árboles de Adelson-Velskii y Landis

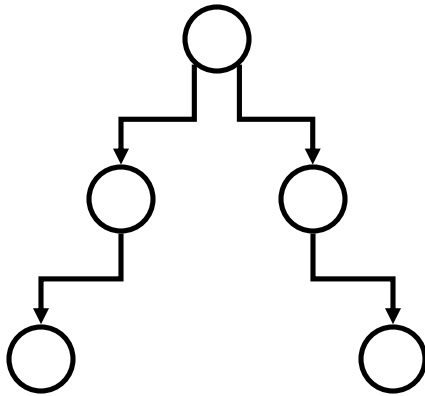
❖ AKA Árboles Simplemente Equilibrados

- **Condición:** Para todo nodo n , $|h_{\text{izq}} - h_{\text{der}}| \leq 1$.
 - h_{izq} = altura del subárbol izquierdo.
 - h_{der} = altura del subárbol derecho.

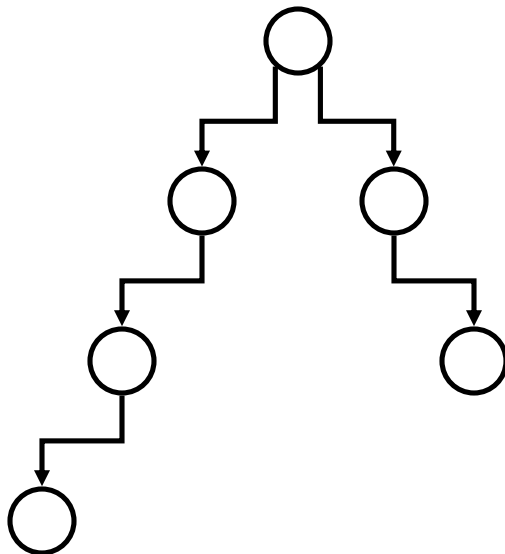


Árboles AVL

Ejemplos



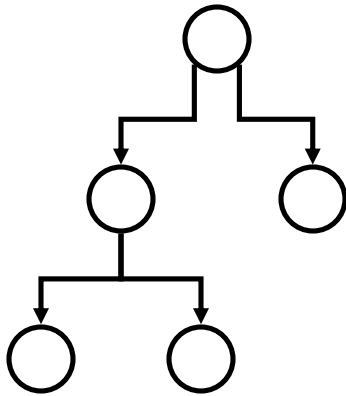
¿AVL? Sí
¿APE? Sí
¿Altura Mínima? Sí



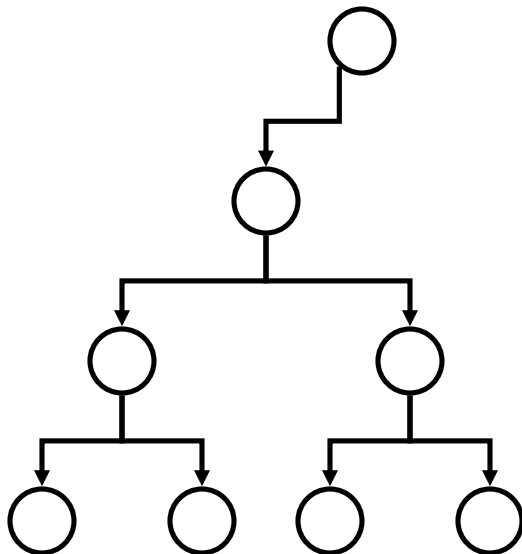
¿AVL? No
¿APE? No
¿Altura Mínima? No

Árboles AVL

Ejemplos



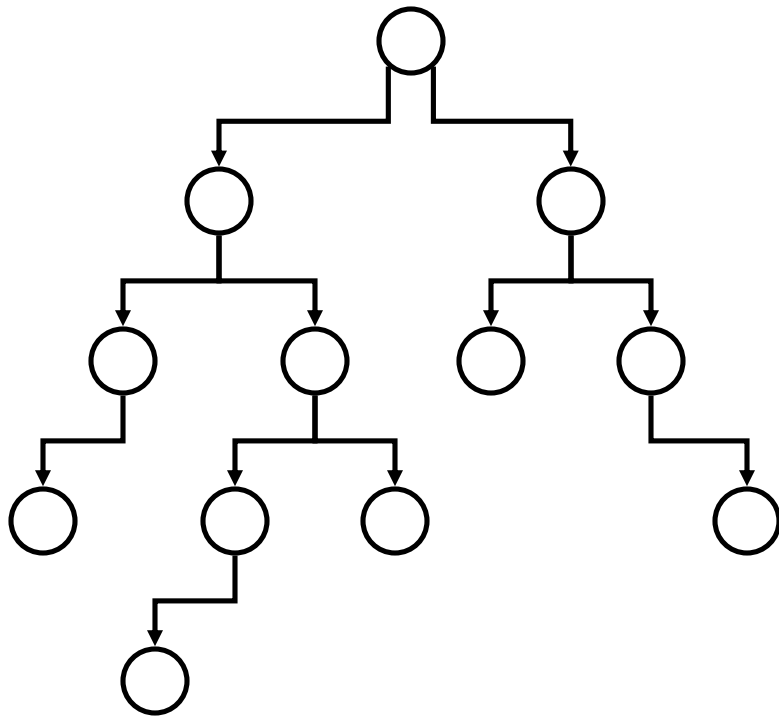
¿AVL? Sí
¿APE? No
¿Altura Mínima? Sí



¿AVL? No
¿APE? No
¿Altura Mínima? Sí

Árboles AVL

Ejemplos

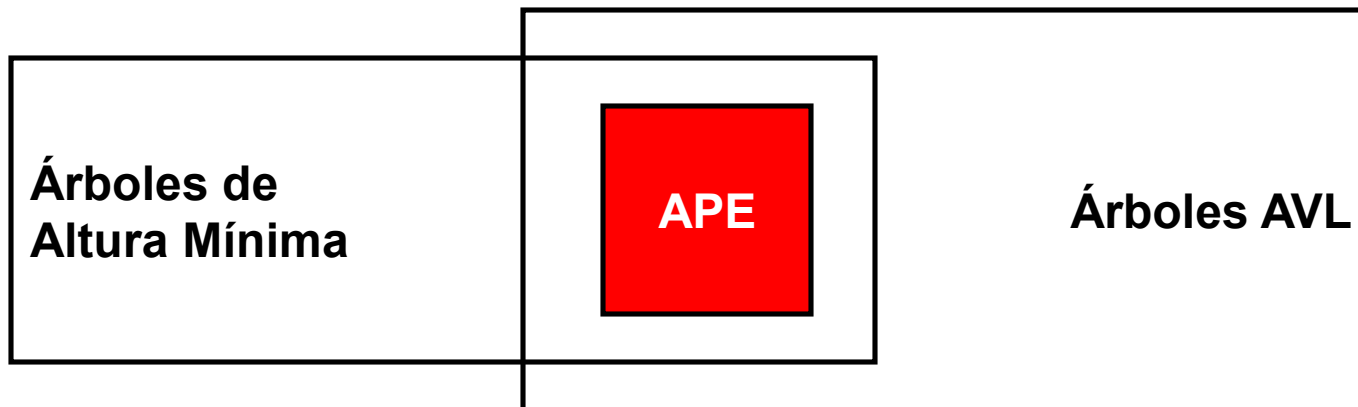


¿AVL? Sí
¿APE? No
¿Altura Mínima? No

Árboles AVL

Propiedades

- ❖ Todo APE es AVL
 - No todo AVL es APE.
 - No todo AVL es Árbol de Altura Mínima.
- ❖ No todo Árbol de Altura Mínima es AVL
 - Visto en los ejemplos.



Árboles AVL

Si los AVL no son árboles de altura mínima...

- ❖ ¿Cuál es su altura máxima?
 - ¿Es esta lo suficientemente pequeña como para resultar eficiente en las operaciones básicas?
 - ¿Cuánto difiere esta altura máxima de la altura mínima?
- ❖ Adelson-Velskii y Landis construyeron una serie de AVL de altura máxima y estimaron estadísticamente su altura
 - Para ello, utilizaron árboles de Fibonacci.
 - Son AVL contruidos de la peor manera posible para alcanzar una altura máxima respetando la condición de AVL.

Árboles AVL

Árboles de Fibonacci

- ❖ Se decide la altura (a) de antemano.
 - Para $h = 0$, usar árbol vacío (T_0).
 - Para $h = 1$, usar (T_1), o árbol de un solo nodo.
 - Para $h > 1$, usar $T_h = (T_{h-1}, x, T_{h-2})$.

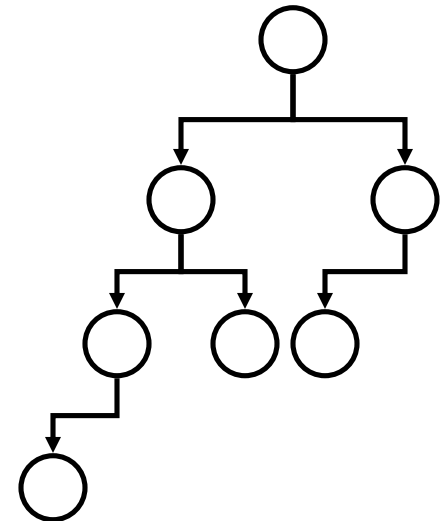
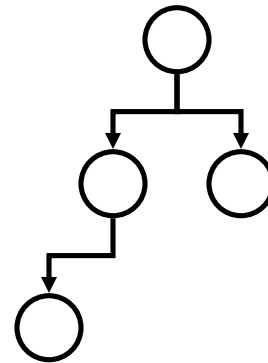
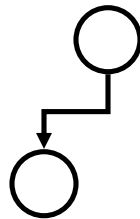
$a = 0$

$a = 1$

$a = 2$

$a = 3$

$a = 4$



Árboles AVL

Adelson-Velskii y Landis demostraron...

Cota para la altura máxima de un árbol de Fibonacci

$$h_{\text{MaxFib}}(n) \leq 1,44 \log_2 n$$

Rango de altura para un AVL

$$h_{\text{APE}}(n) \leq h_{\text{AVL}}(n) \leq h_{\text{MaxFib}}(n)$$

$$\log_2 n \leq h_{\text{AVL}}(n) \leq 1,44 \log_2 n$$

❖ En el caso peor, la altura de un AVL supera a la de un APE en un máximo de un 44% (caso peor)

Complejidad en el caso peor para las tres operaciones básicas

$$O(\log_2 n) \leq O(h_{\text{AVL}}(n)) \leq O(1,44 \log_2 n)$$

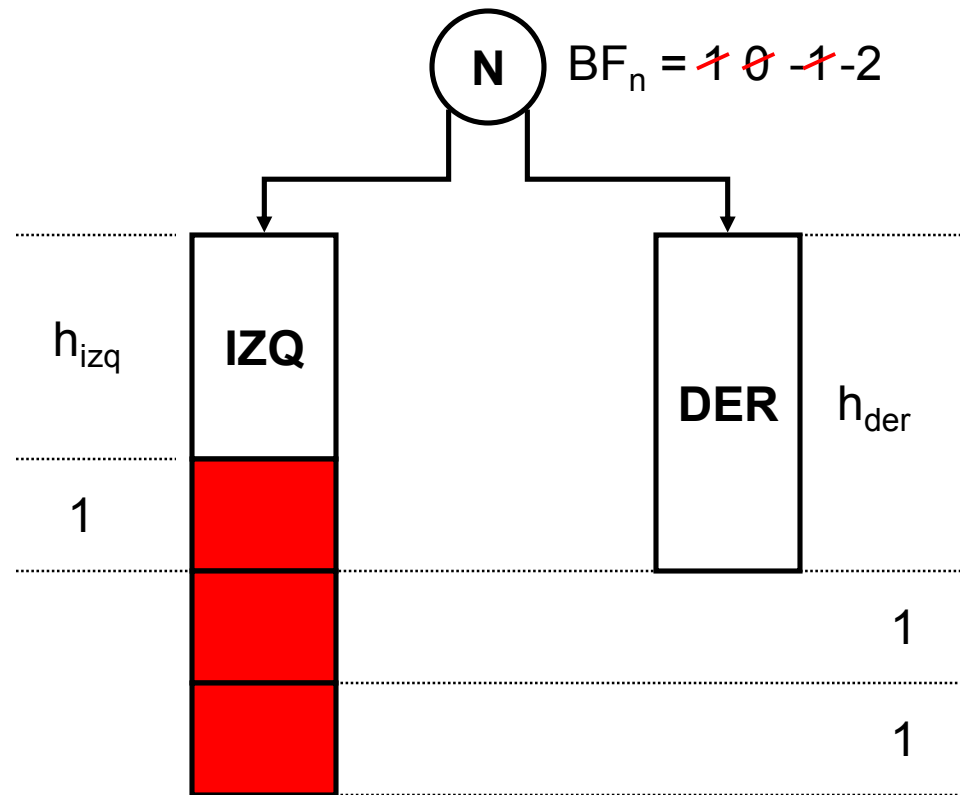
$$O(1,44 \log_2 n)$$

$$O(\log_2 n)$$

Árboles AVL

Balance Factor (BF)

- ❖ $BF_n = h_{\text{der}} - h_{\text{izq}}$
- ❖ Situaciones de Partida:
 - $h_{\text{izq}} > h_{\text{der}}$ ($BF_n = -1$).
 - $h_{\text{izq}} = h_{\text{der}}$ ($BF_n = 0$).
 - $h_{\text{izq}} < h_{\text{der}}$ ($BF_n = 1$).
- ❖ Desequilibrio si
 - $|BF_n| > 1$.



Árboles AVL

Inserción

- ❖ Insertar nodo normalmente y **si ha cambiado la altura** del árbol...
 - Recalcular BF al regresar de la recursividad (actualizando los BF de los nodos que forman parte del camino de búsqueda).
 - Si $|BF_n| > 1$ para algún n entonces reequilibrar (detectando caso).

Class AVLTreeNode

```
public class AVLNode <T extends Comparable <T>>{  
    private T element;  
    public  AVLNode<T> left;  
    private AVLNode<T> right;  
    int BF; // int height;  
}
```

Árboles AVL

Add (Pseudocode)

```
private AVLNode<T> add (AVLNode<T> theRoot, T element)
{
    if (theRoot == null)
        return new AVLNode<T>(element);

    if (element.compareTo(theRoot.getElement()) == 0)
        throw new RuntimeException("the element already exist!");

    if (element.compareTo(theRoot.getElement()) < 0)
        theRoot.setLeft(add(theRoot.getLeft(), element));
    else
        theRoot.setRight(add(theRoot.getRight(), element));

    return(updateBF (theRoot));
}
```

Árboles AVL

UpdateBF (pseudocódigo)

```
private AVLNode<T> updateBF (AVLNode<T> theRoot) {
    theRoot.updateHeight();

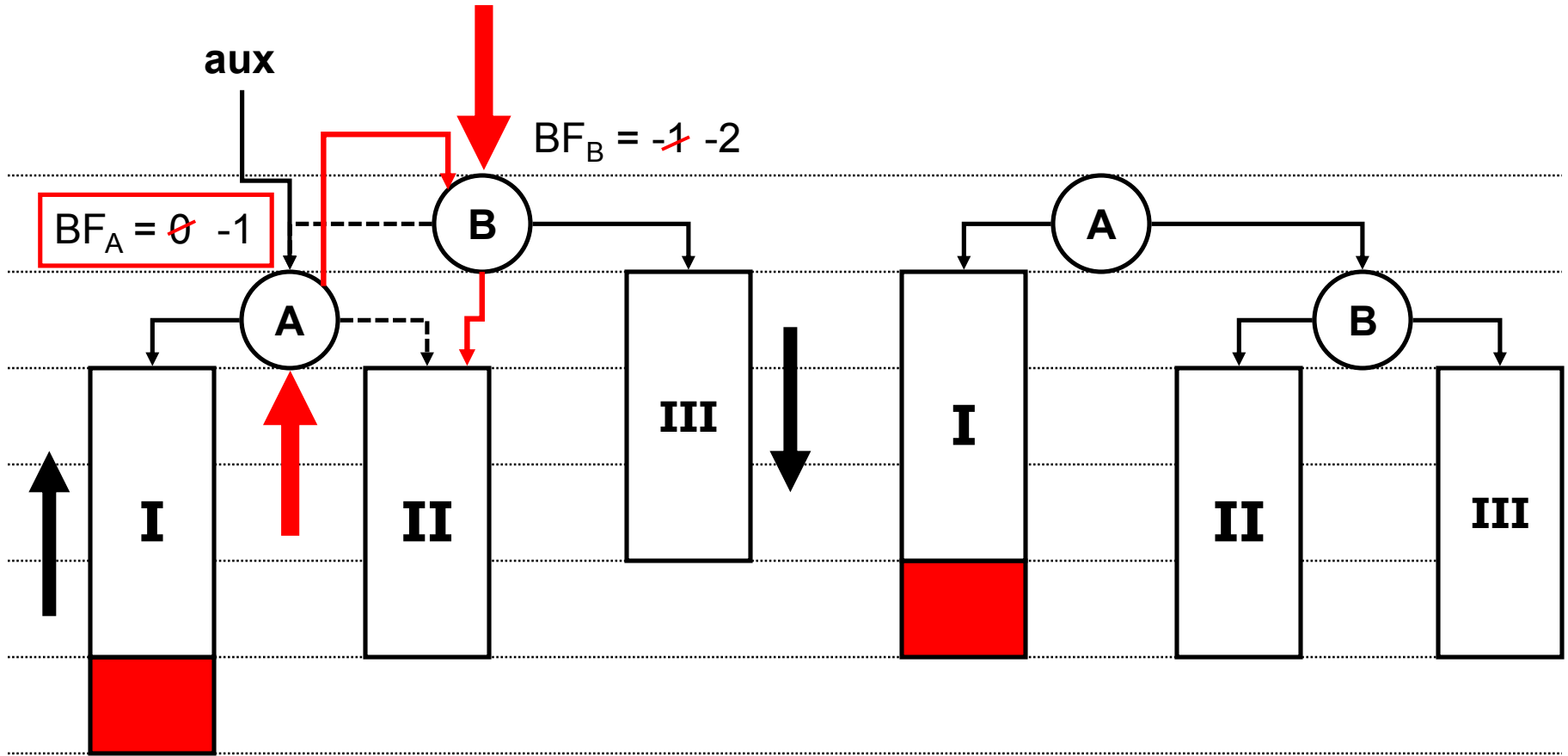
    if (theRoot.getBF() == -2)
    {
        if (theRoot.getLeft().getBF() == -1)
            theRoot = singleLeftRotation (theRoot);
        else
            theRoot = doubleLeftRotation (theRoot);
    }
    else if (theRoot.getBF() == 2)
    {
        if (theRoot.getRight().getBF() == 1)
            theRoot = (singleRightRotation (theRoot));
        else
            theRoot = (doubleRightRotation (theRoot));
    }

    return (theRoot);
}
```

Árboles AVL

Caso la

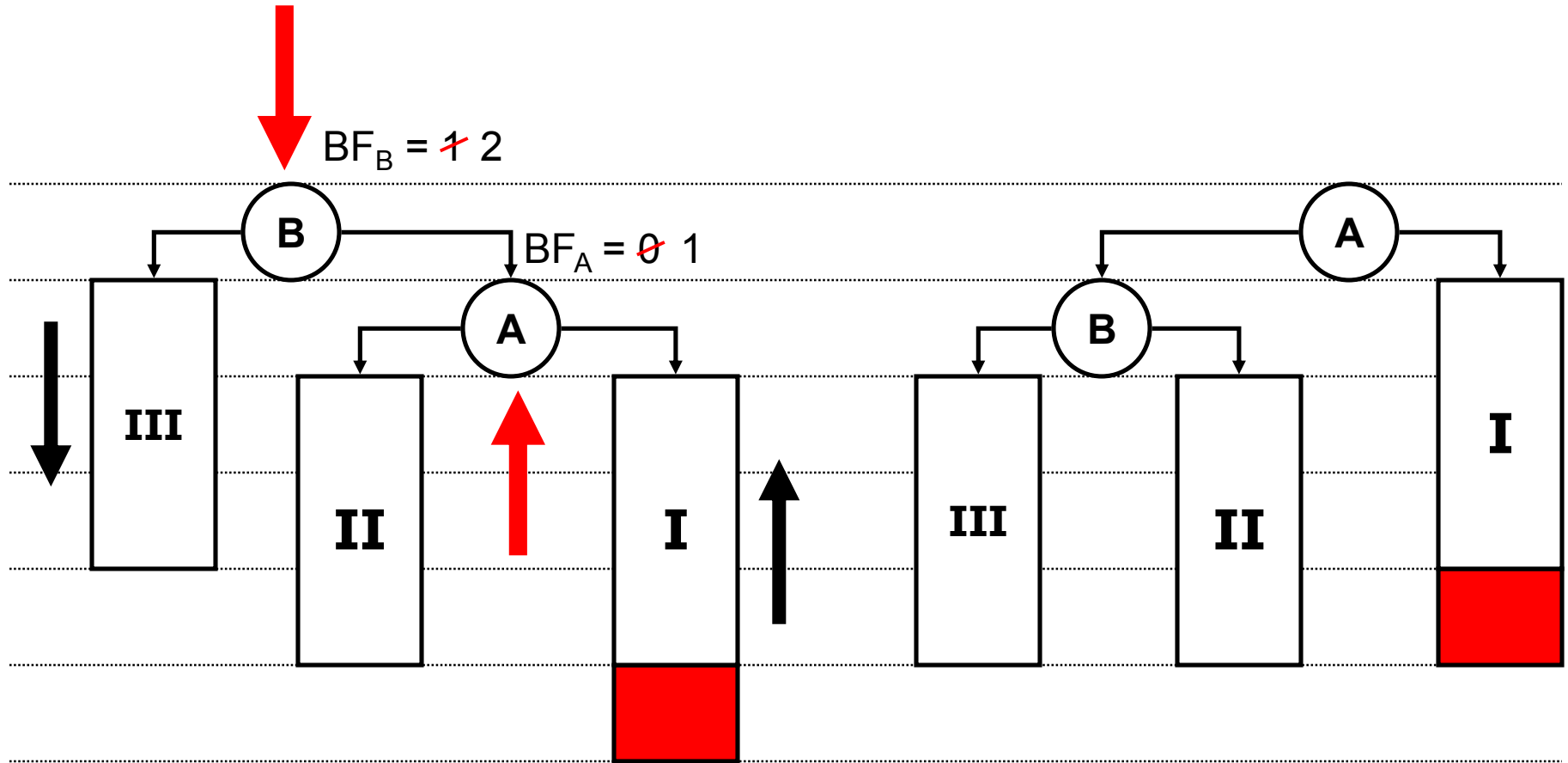
❖ Rotación Simple (izquierdo)



Árboles AVL

Caso Ib

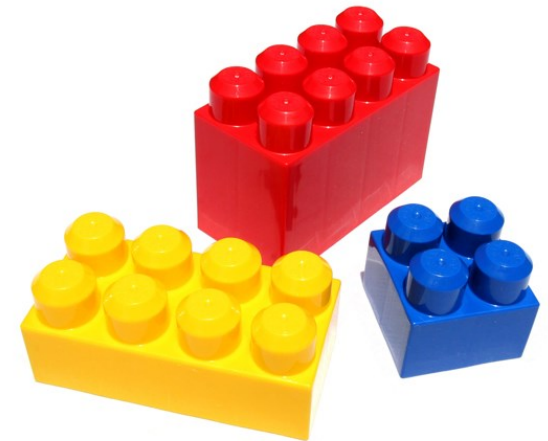
❖ Rotación Simple (derecho)



PLAYGROUND

❖ Ejercicio AVL 1. Partiendo de un árbol AVL vacío...

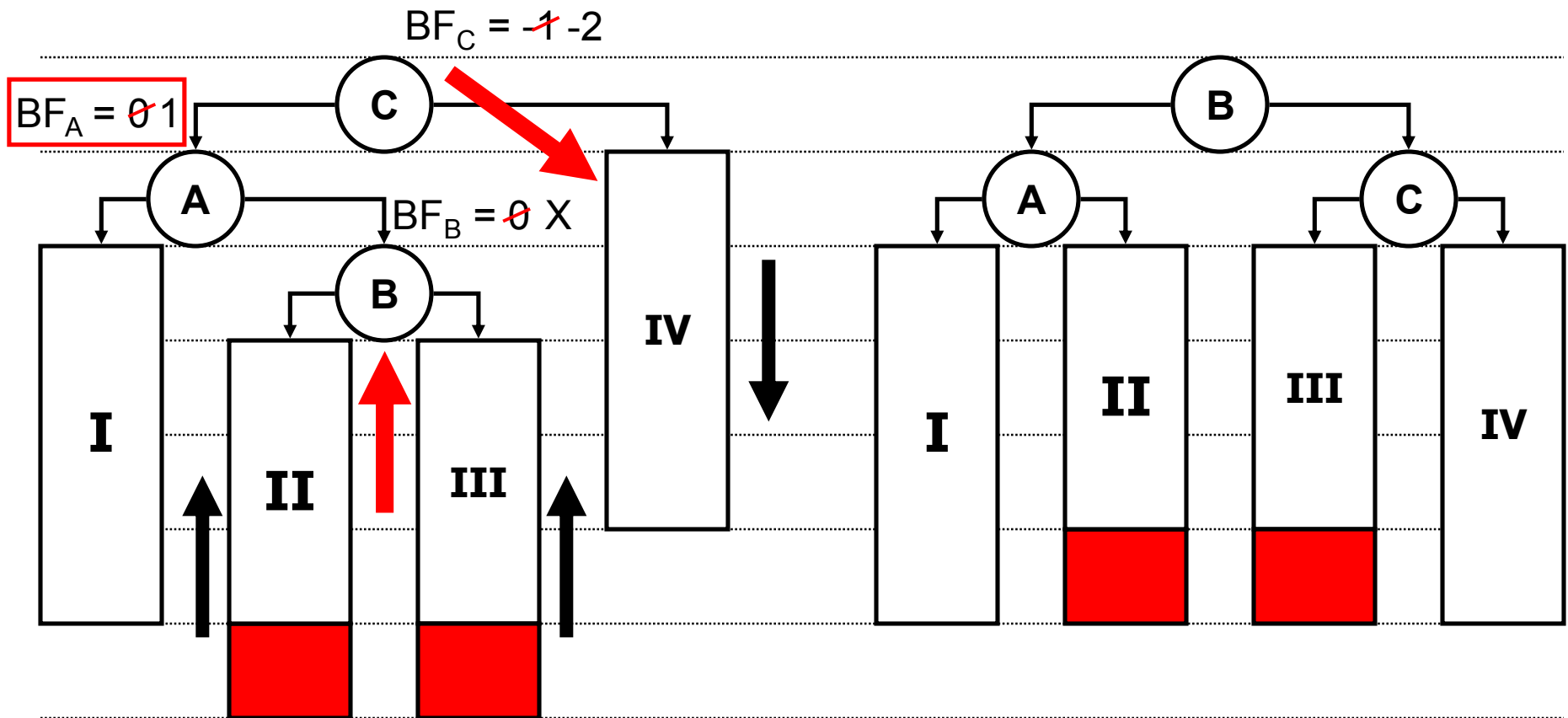
- a) Inserte la secuencia de elementos 7, 6, 5, 4, 3, 2, 1.
 - Analice la complejidad temporal de cada inserción.
- b) Inserte la secuencia de elementos 8, 9, 10.



Árboles AVL

Caso IIa

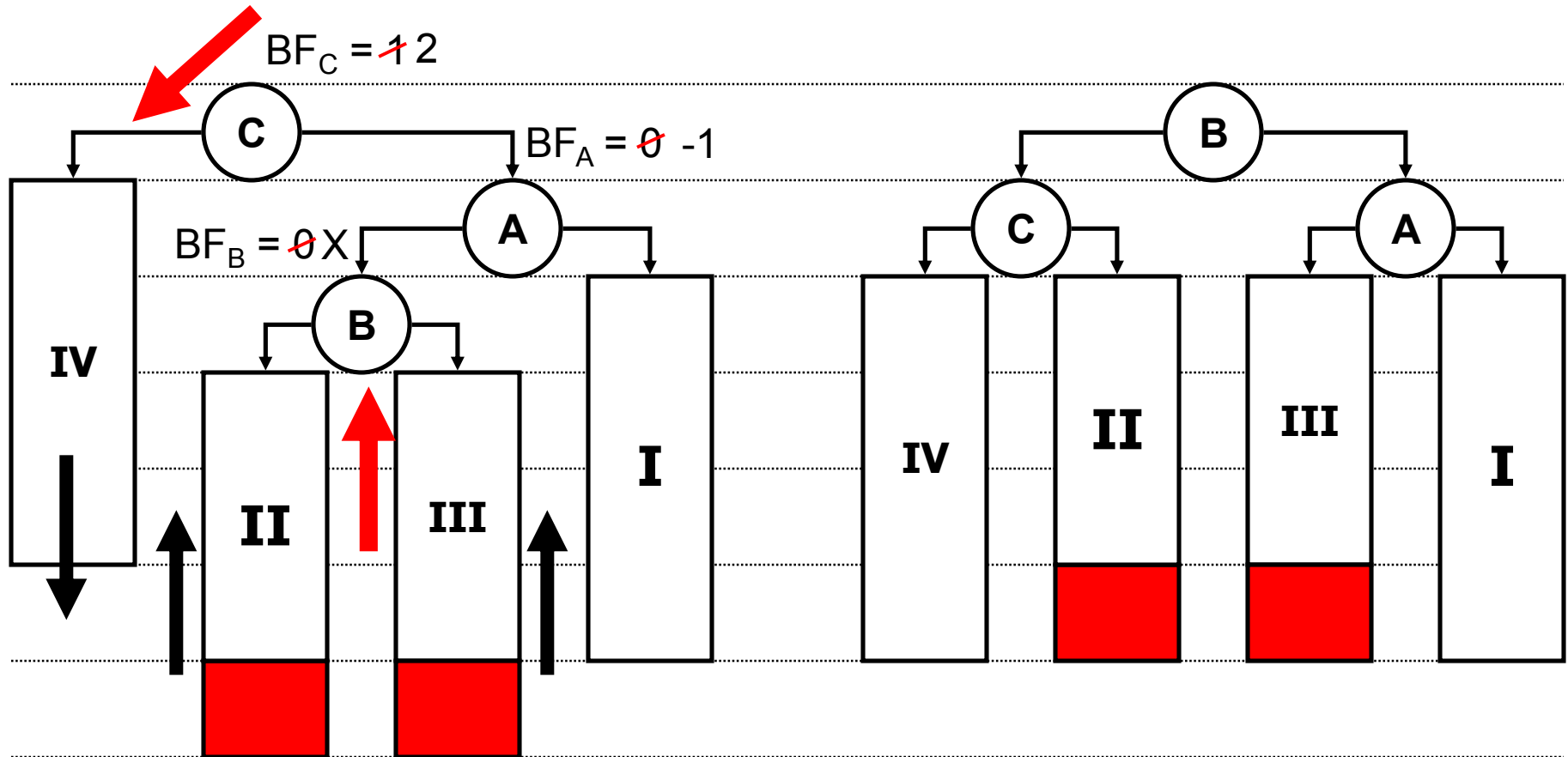
❖ Rotación Doble (izquierdo)



Árboles AVL

Caso IIb

❖ Rotación Doble (derecho)



PLAYGROUND

- ❖ **Ejercicio AVL 2.** Partiendo de un árbol AVL vacío...
 - Inserte la secuencia de elementos 1, 2, 3, 4, 5, 6, 10, 11, 8, 7.
 - Analice la complejidad temporal de cada inserción.

- ❖ **Ejercicio AVL 3.** Partiendo de un árbol AVL vacío...
 - Inserte la secuencia de elementos 5, 2, 10, 15, 12, 9, 7, 8, 6.

Árboles AVL

Borrado

- ❖ Borrar nodo normalmente y **si ha cambiado la altura** del árbol
 - Recalcular BF al regresar de la recursividad (actualizando los BF de los nodos que forman parte del camino de búsqueda).
 - En términos de variación de altura, borrar un nodo del subárbol izquierdo equivale a haber insertado por la derecha.
 - Si $|BF_n| > 1$ para algún n entonces reequilibrar (detectando caso).
- ❖ ¡En el borrado el reequilibrado no es puntual!
 - El reequilibrado de un subárbol **no garantiza** el equilibrio del árbol.
 - A diferencia de la inserción, en el borrado es preciso **continuar reequilibrando siempre** hasta la raíz del árbol.

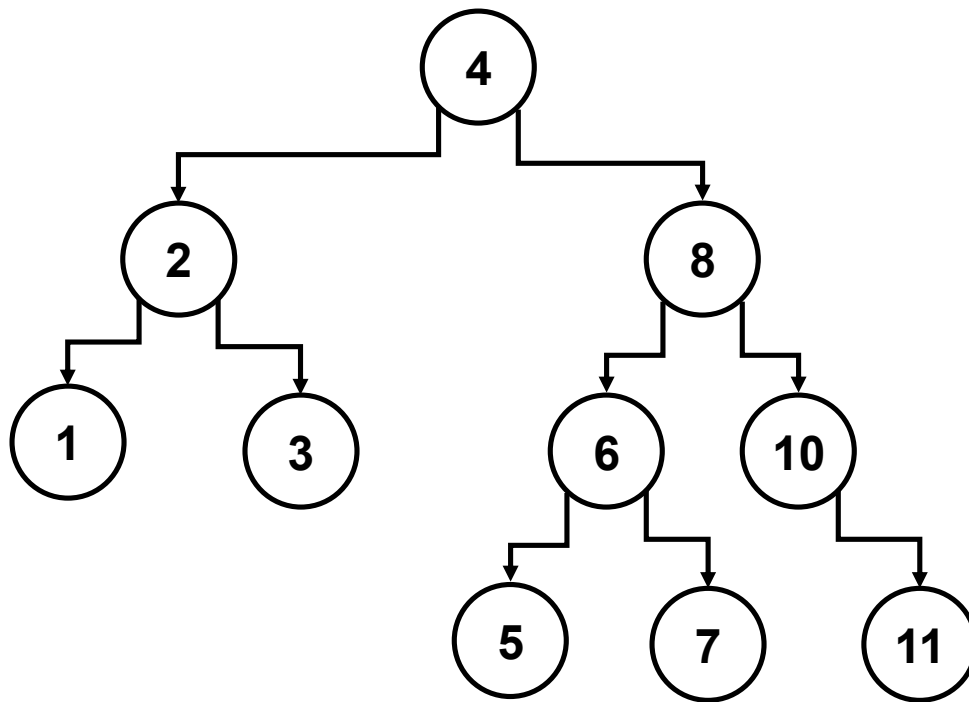
Árboles AVL

Remove (Pseudocode)

```
private AVLNode<T> remove (AVLNode<T> theRoot, T element)
{
    if (theRoot == null) throw new RuntimeException("element does
not exist!");
    else
        if (element.compareTo(theRoot.getElement()) < 0)
            theRoot.setLeft(remove (theRoot.getLeft(), element));
        else
            if (element.compareTo(theRoot.getElement()) > 0)
                theRoot.setRight(remove (theRoot.getRight(), element));
            else {
                if (theRoot.getLeft() == null) return theRoot.getRight();
                else {
                    if (theRoot.getRight() == null) return theRoot.getLeft();
                    else // copies the max value from the left subtree...
                        theRoot.setElement(getMax(theRoot.getLeft()));
                }
            }
    theRoot.setLeft(remove (theRoot.getLeft(), theRoot.getElement()));
    return (updateBF (theRoot));
}
```

PLAYGROUND

- ❖ **Ejercicio AVL 4.** Partiendo del árbol AVL resultante del ejercicio AVL 2...
 - Borre la siguiente secuencia de elementos: 1, 3, 4, 7, 11, 10.
 - Analice la complejidad temporal de cada borrado.

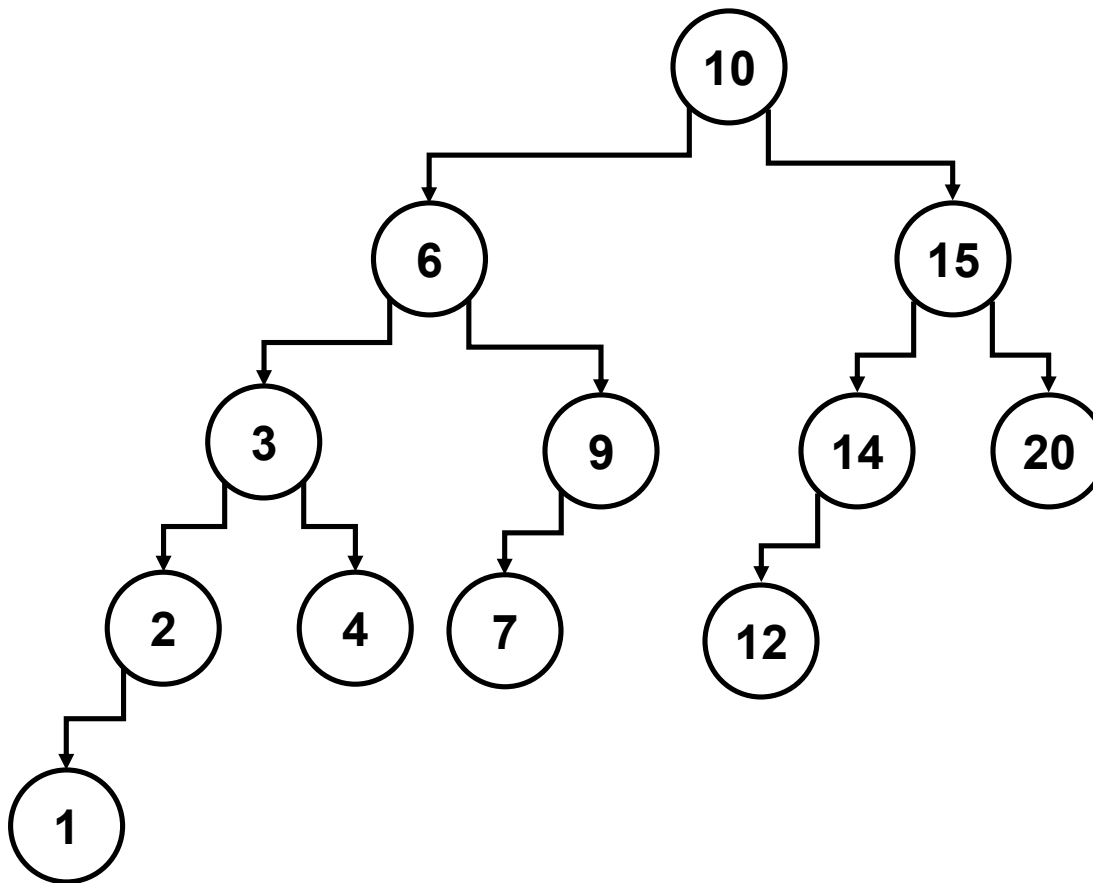


CLASSWORK

PLAYGROUND

❖ **Ejercicio AVL 5.** Partiendo del siguiente árbol AVL...

- Borre la secuencia de elementos 20, 4, 10, 9, 6, 3.



Árboles AVL

Eficiencia de un AVL

❖ Caso Peor

- El reequilibrado en un AVL solo afecta al camino de búsqueda
 - Su longitud es del orden de $\log_2 n$

$$\log_2 n \leq \text{Longitud del Camino de Búsqueda} \leq 1,44 \log_2 n$$

Método	APE	AVL
Insertar	$O(n)$	$O(\log_2 n)$
Buscar	$O(\log_2 n)$	$O(\log_2 n)$
Borrar	$O(n)$	$O(\log_2 n)$

HOMEWORK

PLAYGROUND

- ❖ Consulte la entrada para los **Árboles AVL** en la Wikipedia
 - Estudie **con detalle** todo el contenido de la entrada poniendo especial atención a los pseudocódigos y al código en Java.
 - ¿Qué quiere decir la expresión *if ((altura(insertar(R1,l)) – altura(D)) < 2)* en el pseudocódigo referente a la inserción?
 - ¿Cómo se controla el BF en el método de inserción del árbol AVL en el código Java?

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Árboles B (Bayer & McCreight)

Problema a Resolver

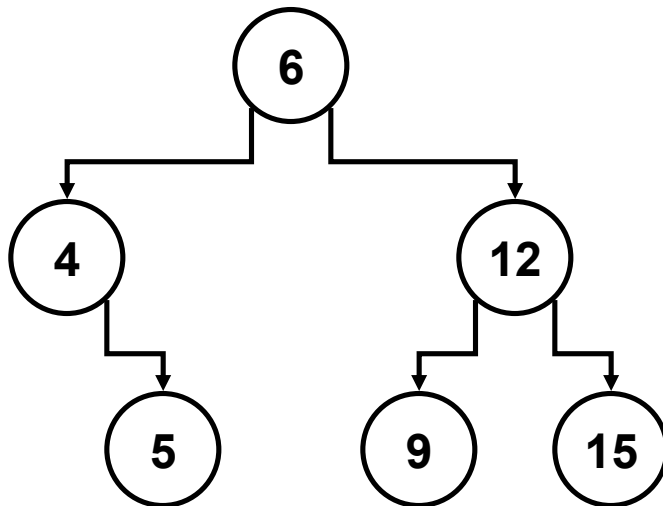
- ❖ Modelar árboles sobre memoria secundaria (disco) capaces de almacenar **cantidades masivas con acceso logarítmico**
 - **Reducen la altura del árbol** a costa de almacenar múltiples elementos por nivel.

- ❖ Desarrollado por el alemán Rudolf Bayer y el suizo Edward M. McCreight en 1972.

Árboles B (Bayer & McCreight)

Procesamiento de árboles en disco

- ❖ Es más barato procesar múltiples elementos en RAM que acceder a ellos en disco de forma individual.



	data	left	right
0	6	2	1
1	12	3	4
2	4	-1	5
3	9	-1	-1
4	15	-1	-1
5	5	-1	-1

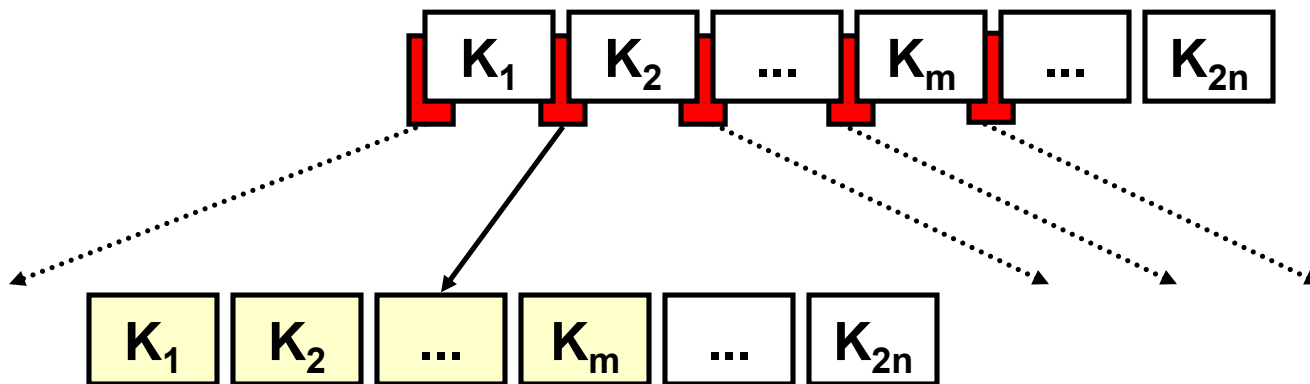
No encontrar un elemento en un árbol AVL de 1.000.000 elementos requiere...

... entre 20 y 28 accesos a disco

Árboles B (Bayer & McCreight)

Definición

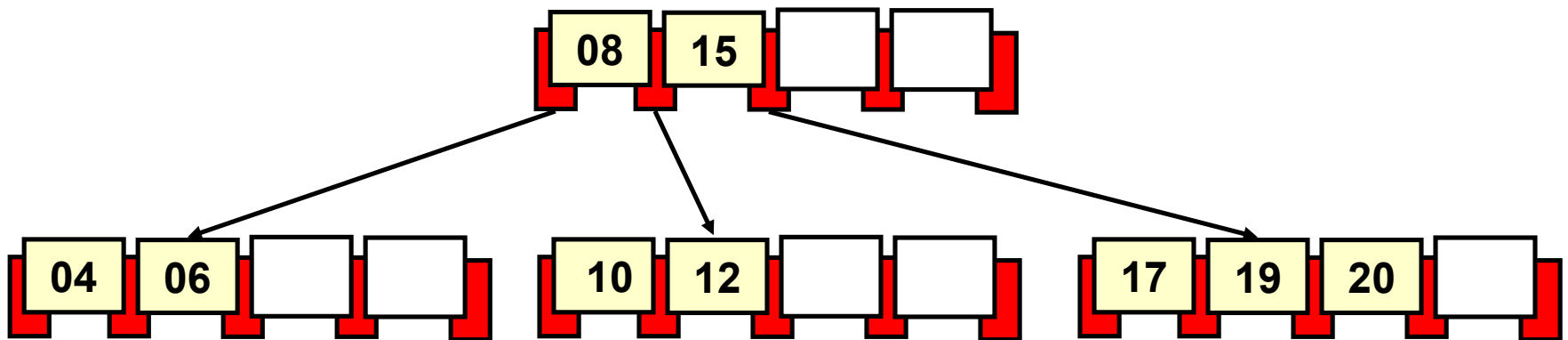
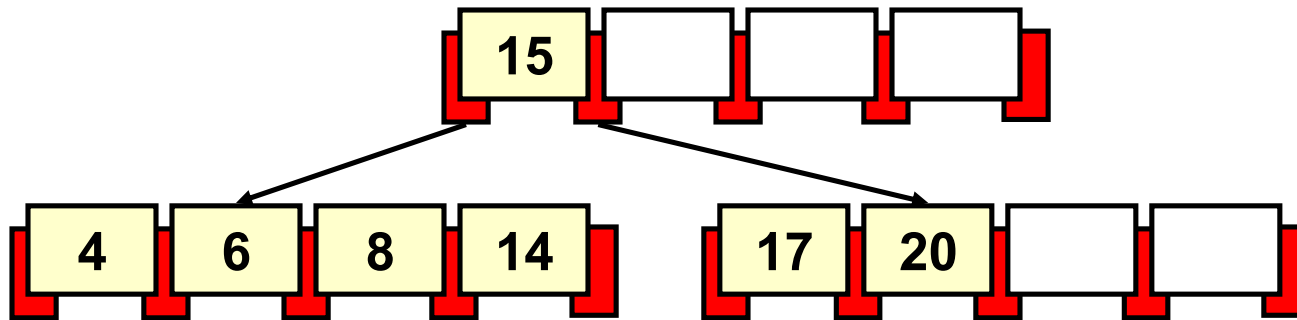
- ❖ Se define un **árbol B de orden n** (B-n) como un árbol donde
 - Todas las hojas se encuentran en el mismo nivel.
 - Todo nodo (llamado página) contiene m elementos (claves) **almacenados de forma ordenada**.
 - La página raíz contiene $1 \leq m \leq 2n$ claves.
 - Toda página no raíz contiene $n \leq m \leq 2n$ claves.
 - Toda página **no hoja** tiene m + 1 páginas hijas.



Árboles B (Bayer & McCreight)

Ejemplos

❖ Árboles B-2



Árboles B (Bayer & McCreight)

Bnode (Pseudocode)

```
class BPage <T extends Comparable <T>> {  
    private final static int n=...;  
    private final static int 2n = 2*n;  
  
    T elements[1..2n];  
    Bpage<T> links [0..2n];  
    int m;  
}
```

Or...

```
class BPage <T extends Comparable <T>> {  
    private final static int n=...;  
    private final static int 2n = 2*n;  
  
    LinkedList<T> elements;  
    LinkedList<Bpage> links;  
    int m; // can be substituted by elements.size();  
}
```

Árboles B (Bayer & McCreight)

Capacidad de un Árbol de orden n

- ❖ Dado un árbol B-n de altura **h**, el número **mínimo** de claves (N_{Min}) que éste puede almacenar se corresponde con
 - Capacidad de un árbol B-n degenerado (estirado al máximo).

Nivel	Pag. x Nivel	Valor mínimo de m	Total
1	1	1	1
2	2	n	2 n
3	2(n + 1)	n	2 n * (n + 1)
4	2(n + 1) ²	n	2 n * (n + 1) ²
...			
h	2(n + 1) ^{h - 2}	n	2 n * (n + 1) ^{h - 2}

$$N_{\text{Min}} = 1 + 2 n * \sum_{i=2}^h (n + 1)^{i - 2}$$

Árboles B (Bayer & McCreight)

Altura **Máxima** de un árbol B-n

- ❖ h_{\max} se obtiene de
 - $N = 1 + 2n * \sum_{i=2}^h (n+1)^{i-2}$
 - N es el número de claves del árbol.
- ❖ $h_{\max} \approx 1 + \text{Log}_{n+1}(N+1)/2$
 - Si la constante **n** es lo suficientemente grande, h_{\max} se aproxima a:
 - $h_{\max} \approx \text{Log}_n N$.

Rango de altura árbol B-n

$$h < \approx 1 + \text{Log}_{n+1}(N+1)/2$$

$$O(h) < \approx O(\text{Log}_n N)$$

- ❖ Cuanto mayor sea el orden del árbol (**n**) menor será la altura del árbol.

Árboles B (Bayer & McCreight)

Capacidad de un árbol de orden n

- ❖ Dado un árbol B- n de altura h , el número **máximo** de claves (N_{Max}) que éste puede almacenar se corresponde con
 - Capacidad de un árbol B- n compacto (completo).

Nivel	Pag. x Nivel	Valor máximo de m	Total
1	1	$2n$	$2n$
2	$(2n + 1)$	$2n$	$2n * (2n + 1)$
3	$(2n + 1)^2$	$2n$	$2n * (2n + 1)^2$
4	$(2n + 1)^3$	$2n$	$2n * (2n + 1)^3$
...			
h	$(2n + 1)^{h-1}$	$2n$	$2n * (2n + 1)^{h-1}$

$$N_{\text{Max}} = 2n * \sum_{i=1}^h (2n + 1)^{i-1}$$

Árboles B (Bayer & McCreight)

Altura **Mínima** de un árbol B-n

❖ h_{Min} se obtiene de...

- $N = 2n * \sum_{i=1}^h (2n + 1)^{i-1}$.
 - N es el número de claves del árbol.

❖ $h_{\text{min}} \approx \text{Log}_{2n+1}(N+1)$.

- Si la constante **n** es lo suficientemente grande, h_{min} se aproxima a:
 - $h_{\text{min}} \approx \text{Log}_{2n}N$.

Rango de altura árbol B-n

$$\text{Log}_{2n+1}(N+1) < \approx h < \approx 1 + \text{Log}_{n+1}(N+1) / 2$$

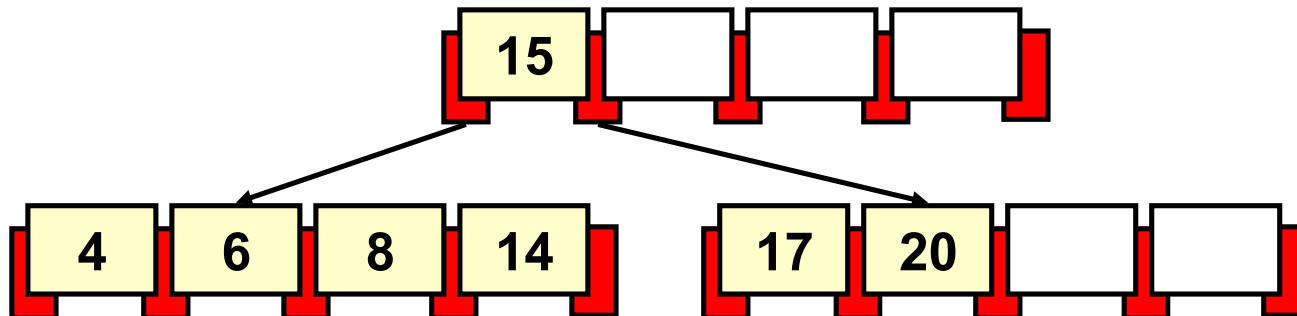
$$O(\text{Log}_{2n}N) \leq O(h) \leq O(\text{Log}_nN)$$

❖ Cuanto mayor sea el orden del árbol (**n**) menor será la altura del árbol.

Árboles B (Bayer & McCreight)

Búsqueda

- ❖ Buscar elemento X dentro de los *elements* de la página
 - Búsqueda secuencial.
 - Búsqueda binaria.
- ❖ Si la búsqueda falla, ésta se detendrá en una posición j ($\text{elements}[j]$) de la página tal que $0 \leq j \leq m$
 - Cargar la página $\text{links}[j]$ y repetir búsqueda.
 - El proceso recursivo se repite hasta encontrar X o hasta llegar a un enlace null, en cuyo caso el elemento no existe.



Árboles B (Bayer & McCreight)

Complejidad Temporal

❖ Caso Mejor

- El elemento buscado se encuentra en la raíz
 - $O(m) = O(1)$.
 - Dado que $1 \leq m \leq 2n$, m se puede considerar constante.

❖ Caso Peor

- Se busca en un árbol degenerado y el elemento no se encuentra
 - $O(h) * O(m)$.
 - $O(\log_n N) * O(1) = O(\log_n N)$.

No encontrar un elemento en un árbol B-10 de 1.000.000 elementos requiere...

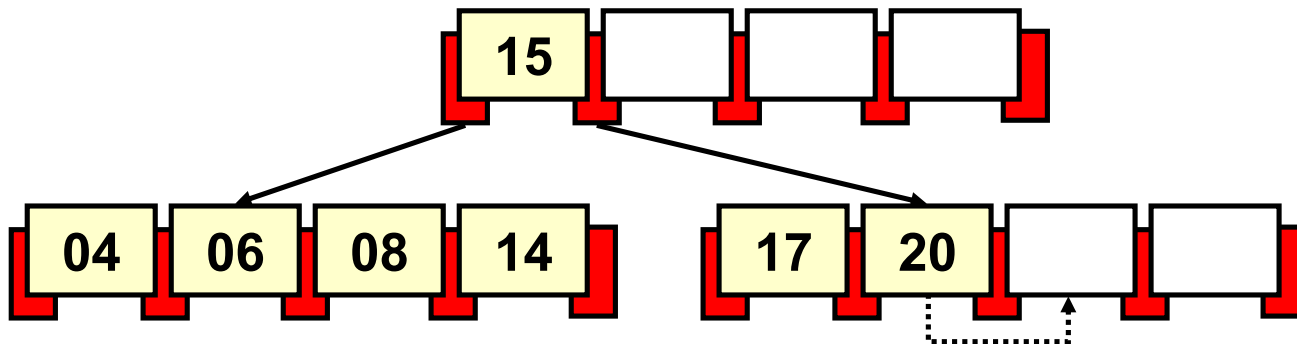
... entre 5 y 6 accesos a disco

... un AVL requiere entre 20 y 28 accesos

Árboles B (Bayer & McCreight)

Inserción

- ❖ **Caso I:** Pagina hoja tiene $m < 2 \cdot n$ claves.
 - Se abre hueco para el elemento a insertar desplazando los elementos de clave mayor una posición hacia la derecha.

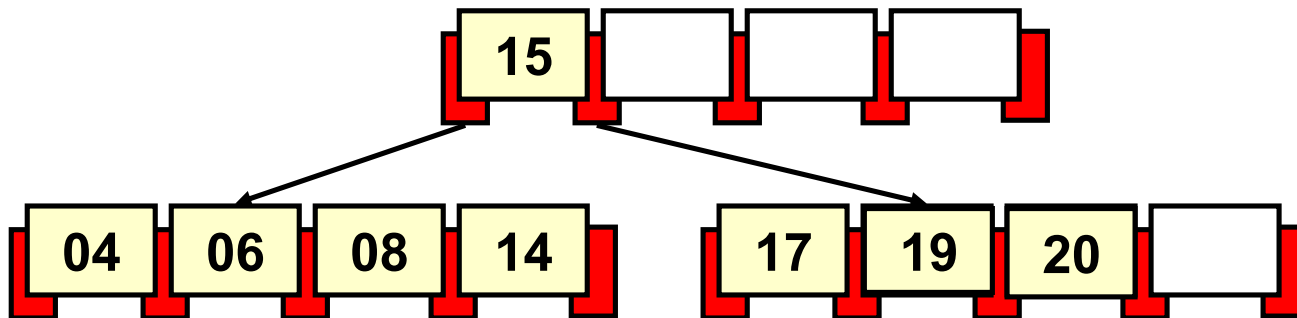


- ❖ La inserción siempre se produce en las hojas y tras una búsqueda infructuosa.

Árboles B (Bayer & McCreight)

Inserción

- ❖ **Caso I:** Pagina hoja tiene $m < 2 \cdot n$ claves.
 - Se abre hueco para el elemento a insertar desplazando los elementos de clave mayor una posición hacia la derecha.

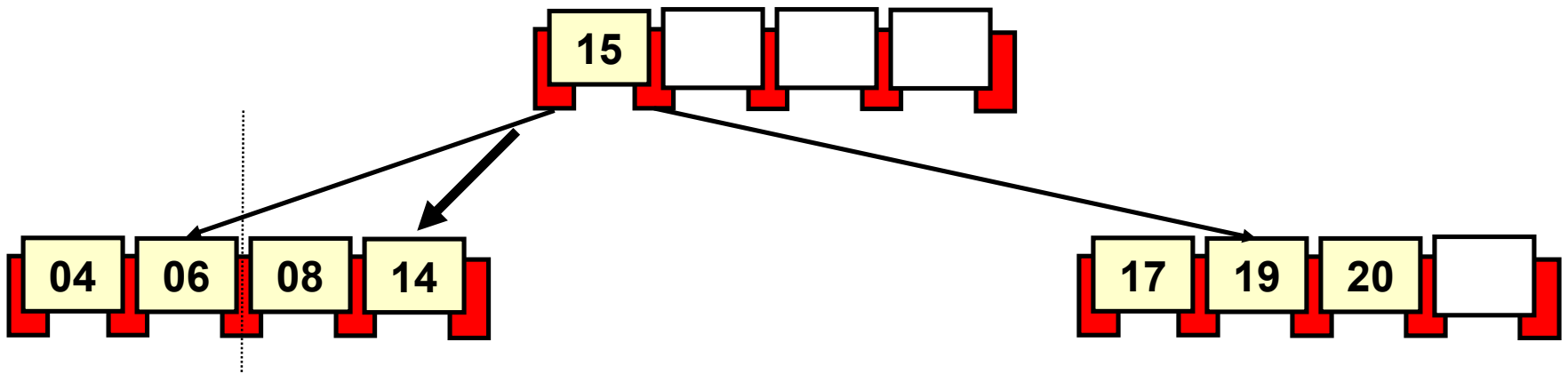


- ❖ La inserción siempre se produce en las hojas y tras una búsqueda infructuosa.

Árboles B (Bayer & McCreight)

Inserción

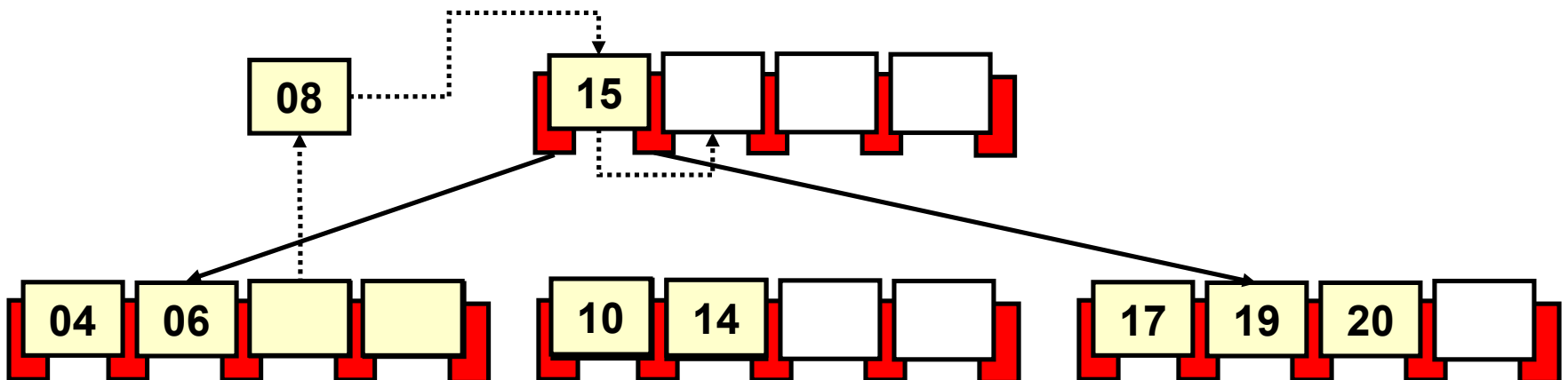
- ❖ **Caso 2:** Pagina hoja tiene $m = 2 \cdot n$ claves (**Overflow**).
 - División de la hoja en dos, repartiendo claves en ellas
 - Últimas $(m+1)/2$ claves en hoja nueva.
 - Primeras $(m+1)/2$ claves permanecen en hoja original.
 - Promociona el elemento central (mediana) a la página padre actuando de separador.
 - El objeto promocionado se inserta en la página padre repitiendo el proceso de forma recursiva, pudiendo afectar a la raíz.
 - Desdoblar la raíz es la única forma de aumentar la altura de un árbol B.



Árboles B (Bayer & McCreight)

Inserción

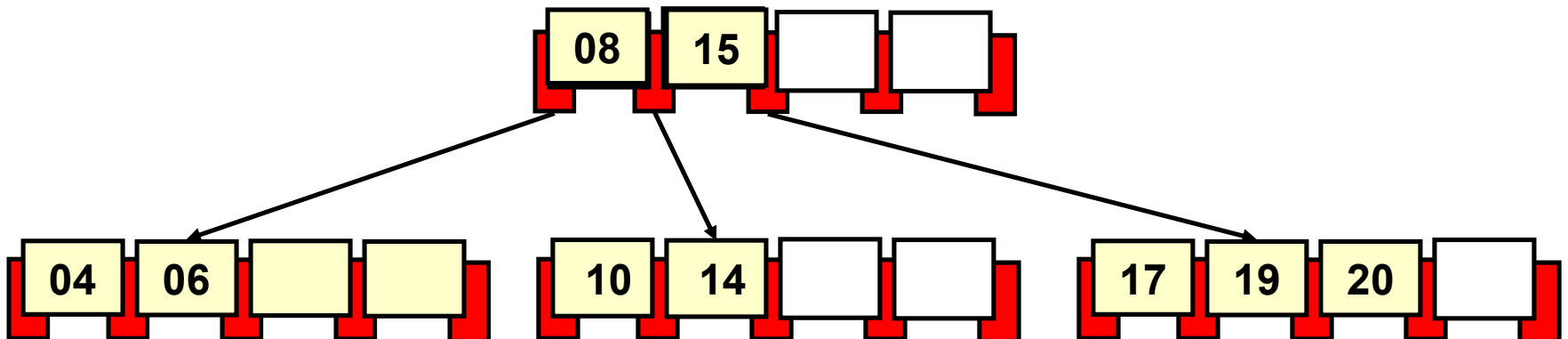
- ❖ **Caso 2:** Pagina hoja tiene $m = 2 \cdot n$ claves (**Overflow**).
 - División de la hoja en dos, repartiendo claves en ellas
 - Últimas $(m+1)/2$ claves en hoja nueva.
 - Primeras $(m+1)/2$ claves permanecen en hoja original.
 - Promociona el elemento central (mediana) a la página padre actuando de separador.
 - El objeto promocionado se inserta en la página padre repitiendo el proceso de forma recursiva, pudiendo afectar a la raíz.
 - Desdoblar la raíz es la única forma de aumentar la altura de un árbol B.



Árboles B (Bayer & McCreight)

Inserción

- ❖ **Caso 2:** Pagina hoja tiene $m = 2 \cdot n$ claves (**Overflow**).
 - División de la hoja en dos, repartiendo claves en ellas
 - Últimas $(m+1)/2$ claves en hoja nueva.
 - Primeras $(m+1)/2$ claves permanecen en hoja original.
 - Promociona el elemento central (mediana) a la página padre actuando de separador.
 - El objeto promocionado se inserta en la página padre repitiendo el proceso de forma recursiva, pudiendo afectar a la raíz.
 - Desdoblar la raíz es la única forma de aumentar la altura de un árbol B.



Árboles B (Bayer & McCreight)

Complejidad Temporal Inserción

❖ Caso Mejor

- El elemento se inserta directamente en una hoja en la que hay espacio dentro de un árbol de altura mínima.
 - $O(\log_{2n}(N)) + O(m) = O(\log_{2n}(N))$.

❖ Caso Peor

- Se inserta en un árbol degenerado y se desdoblan todas las páginas desde las hojas hasta la raíz.
 - $O(\log_n(N)) * O(n) = O(\log_n(N))$.

PLAYGROUND

❖ **Ejercicio Árbol B (Inserción).** Partiendo de un árbol B-2 vacío...

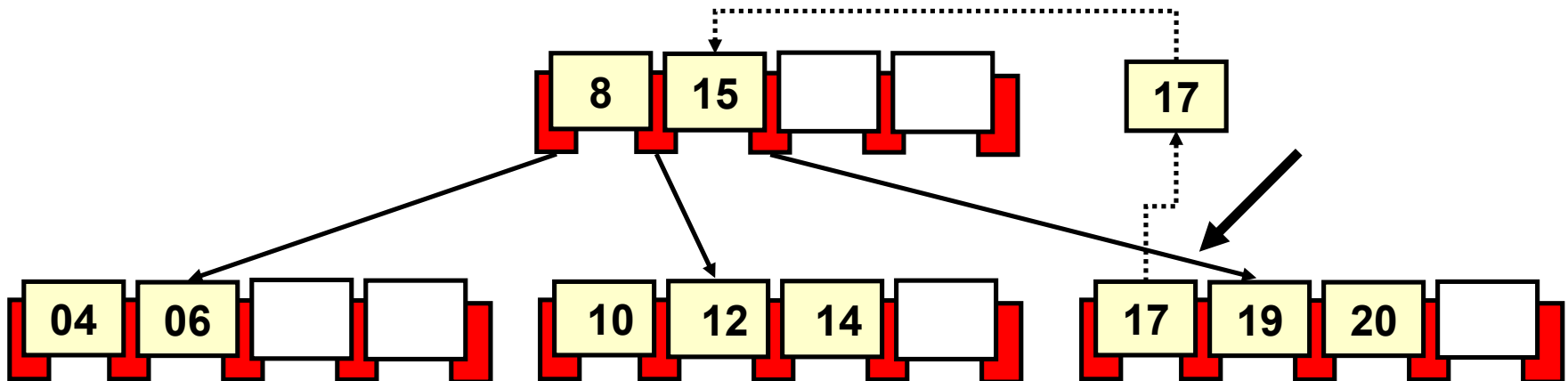
- a) Insertar la secuencia de claves 6, 11, 5, 4, 8, 9, 12.
- b) Insertar la secuencia de claves 21.
- c) Insertar la secuencia de claves 14, 10, 19, 28.
- d) Insertar la secuencia de claves 3, 17, 32, 15, 16.
- e) Insertar la secuencia de claves 26, 27.

Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento interior

- Sustituir elemento por su sucesor
 - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
- Borrar elemento **de página donante**.

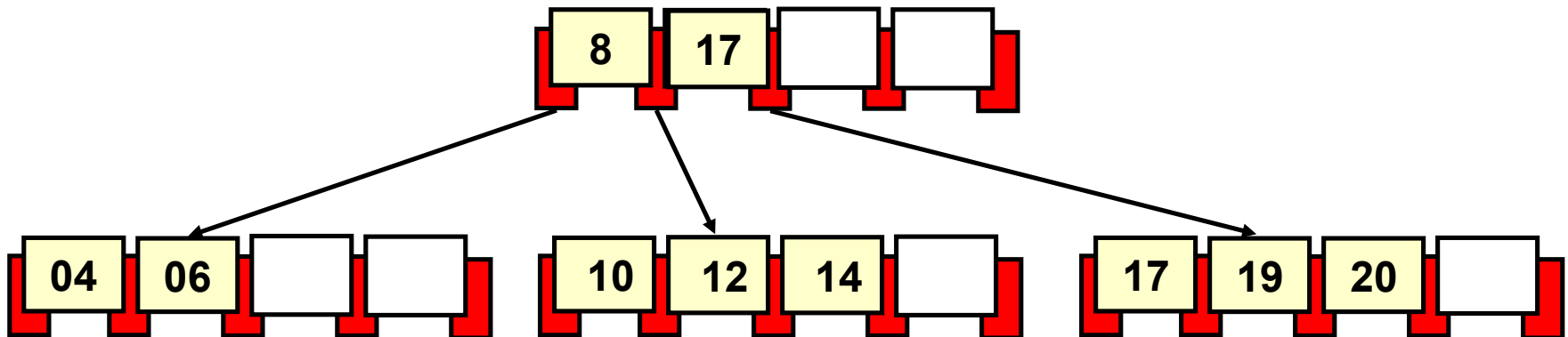


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento interior

- Sustituir elemento por su sucesor
 - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
- Borrar elemento **de página donante**.

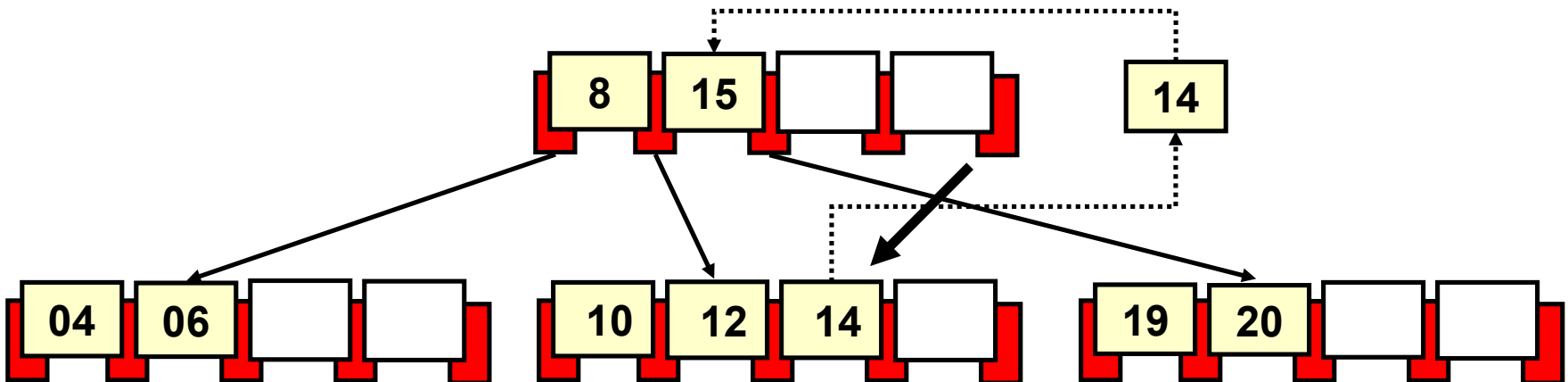


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento interior

- Sustituir elemento por su sucesor
 - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
- Si la página donante se encuentra en situación crítica,
 - Intentar sustituir por antecesor (elemento extremo derecho subárbol izquierdo).
 - La página se encuentra en situación crítica si $m=n$ antes de la sustitución.
- Borrar elemento **de página donante**.

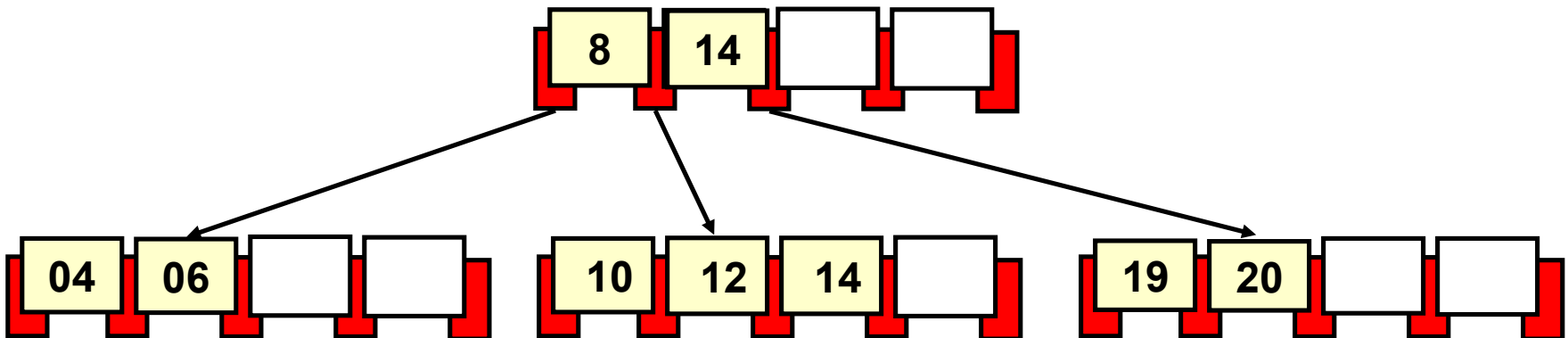


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento interior

- Sustituir elemento por su sucesor
 - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
- Si la página donante se encuentra en situación crítica,
 - Intentar sustituir por antecesor (elemento extremo derecho subárbol izquierdo).
 - La página se encuentra en situación crítica si $m=n$ antes de la sustitución.
- Borrar elemento **de página donante**.

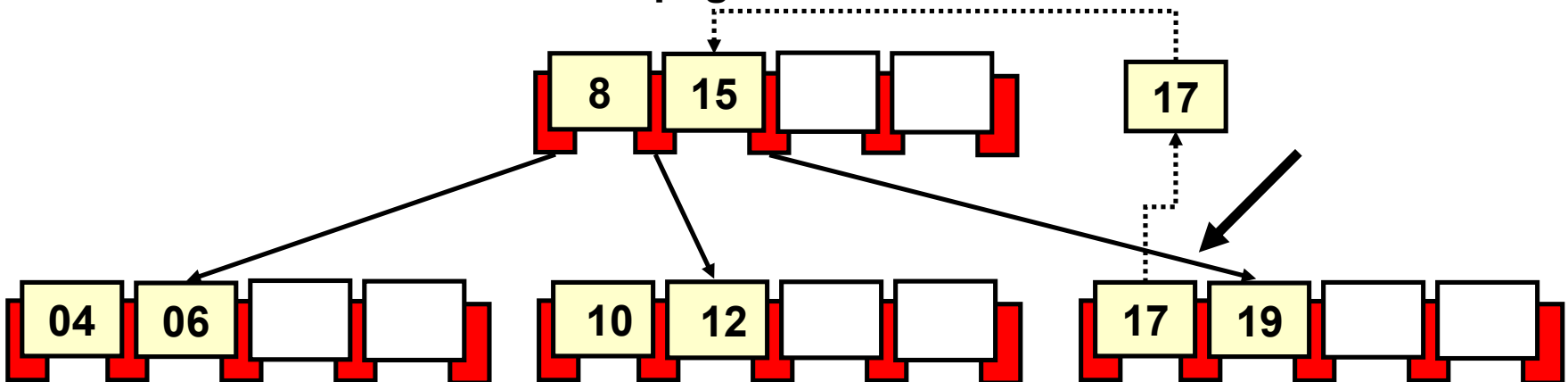


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento interior

- Sustituir elemento por su sucesor
 - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
- Si la página donante se encuentra en situación crítica,
 - Intentar sustituir por antecesor (elemento extremo derecho subárbol izquierdo).
 - La página se encuentra en situación crítica si $m=n$ antes de la sustitución.
- Si la página donante se encuentra en situación crítica **sustituir por el sucesor.**
- Borrar elemento **de página donante.**

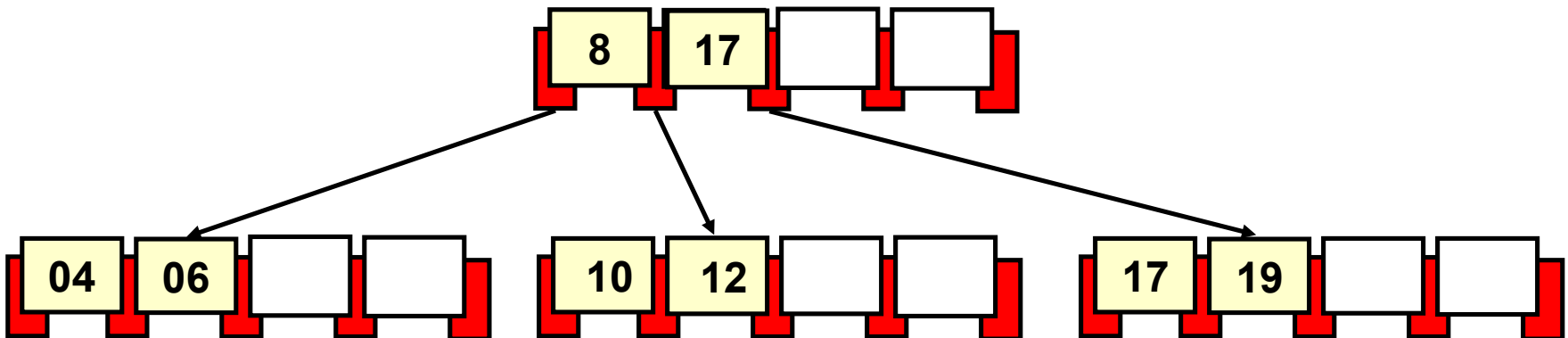


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento interior

- Sustituir elemento por su sucesor
 - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
- Si la página donante se encuentra en situación crítica,
 - Intentar sustituir por antecesor (elemento extremo derecho subárbol izquierdo).
 - La página se encuentra en situación crítica si $m=n$ antes de la sustitución.
- Si la página donante se encuentra en situación crítica **sustituir por el sucesor**.
- Borrar elemento de página donante.

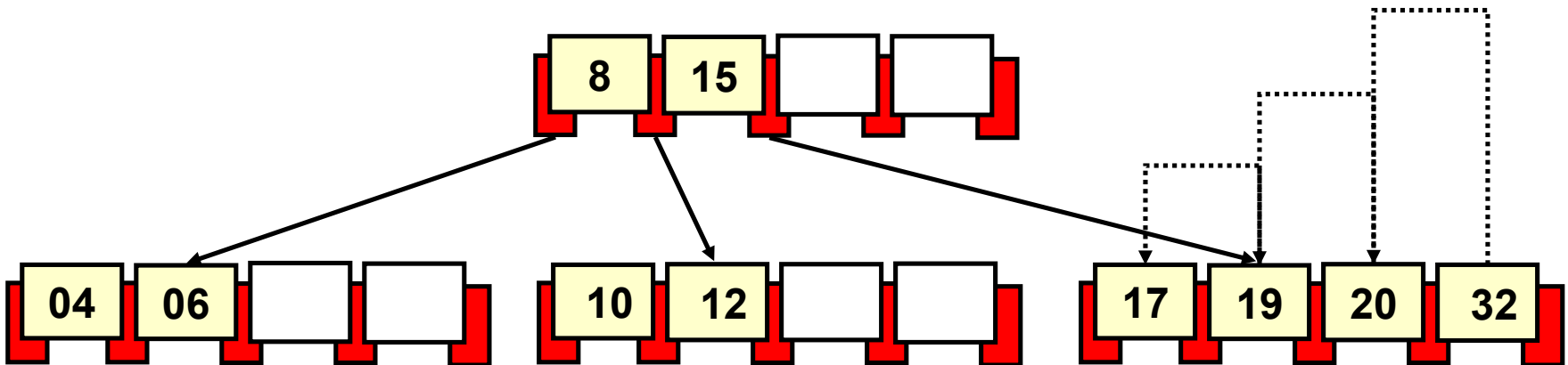


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 1:** Página tiene $n < m$ claves.
 - Se desplazan los elementos de clave mayor una posición hacia la izquierda.

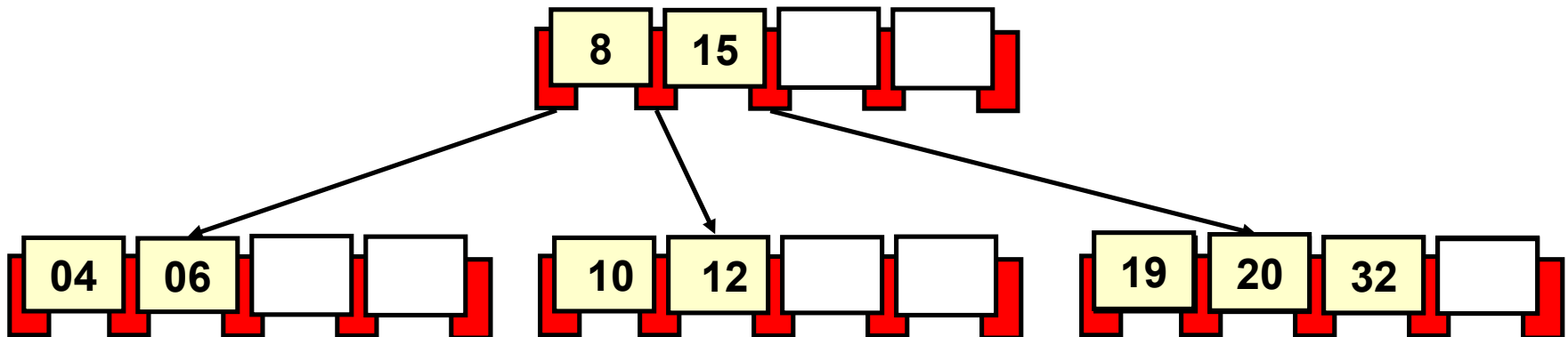


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 1:** Página tiene $n < m$ claves.
 - Se desplazan los elementos de clave mayor una posición hacia la izquierda.

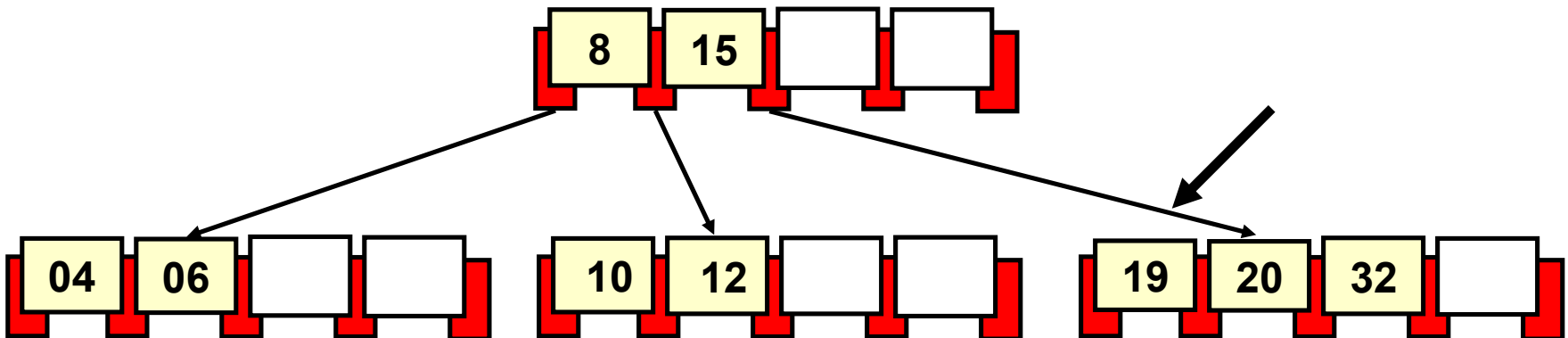


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 2:** Página tiene $n = m$ claves (*underflow*).
 - Se busca entre las hojas adyacentes alguna que tenga $n < m$ claves y se le pide la cesión de una de ellas.
 - » Se consulta **primero con la vecina derecha** (si existe) y si ésta no puede ceder claves **se intenta el proceso con la vecina izquierda**.
 - » Una hoja no puede ceder claves cuando se encuentra en situación crítica ($n = m$).

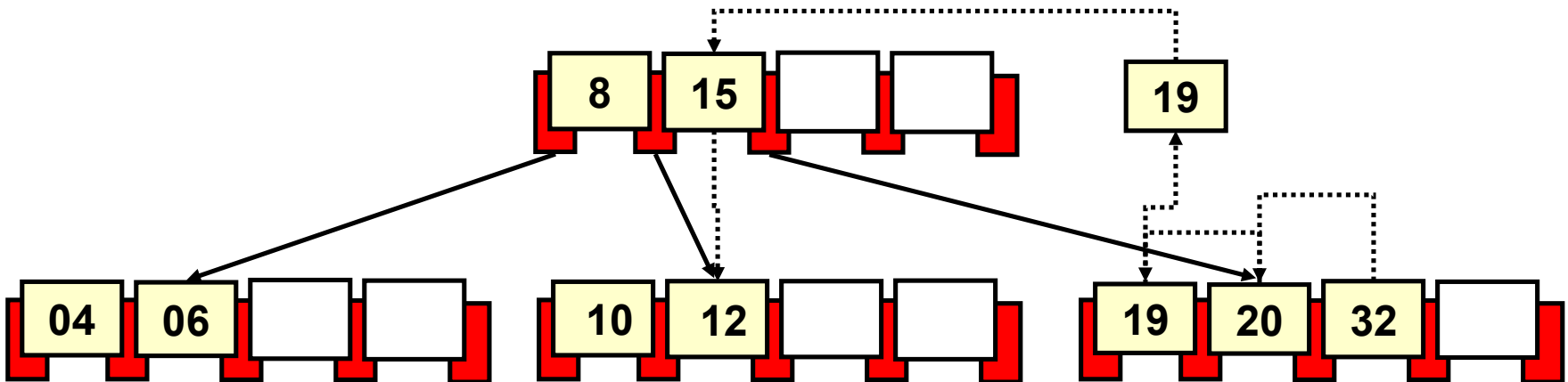


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 2:** Página tiene $n = m$ claves (*underflow*).
 - El préstamo se realiza **siempre a través de la página padre** de las dos implicadas.
 - El elemento cedido por la página donante se envía a la página padre para sustituir al separador, quien baja a la página que recibe la cesión
 - El **separador sustituye** al elemento borrado.

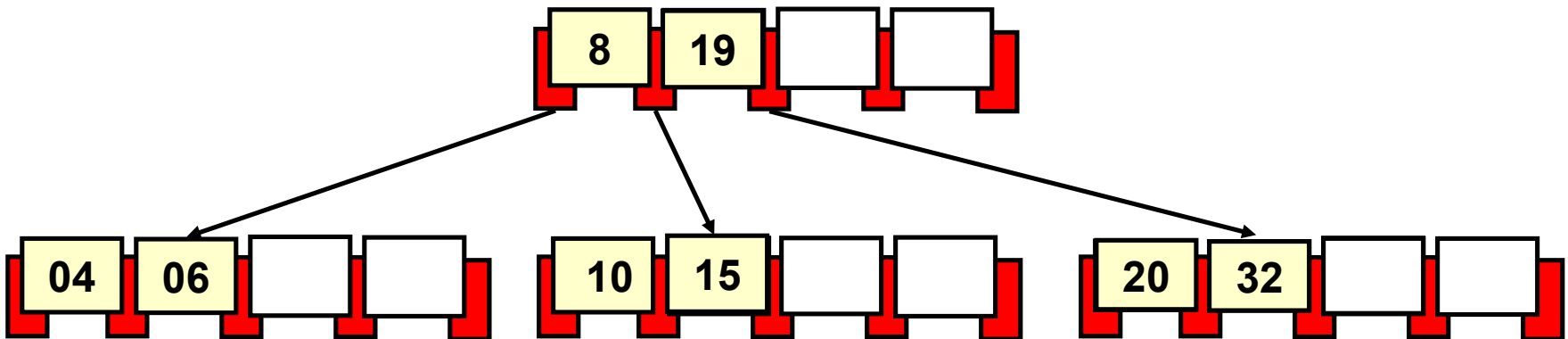


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 2:** Página tiene $n = m$ claves (*underflow*).
 - El préstamo se realiza **siempre a través de la página padre** de las dos implicadas.
 - El elemento cedido por la página donante se envía a la página padre para sustituir al separador, quien baja a la página que recibe la cesión.
 - El **separador sustituye** al elemento borrado.

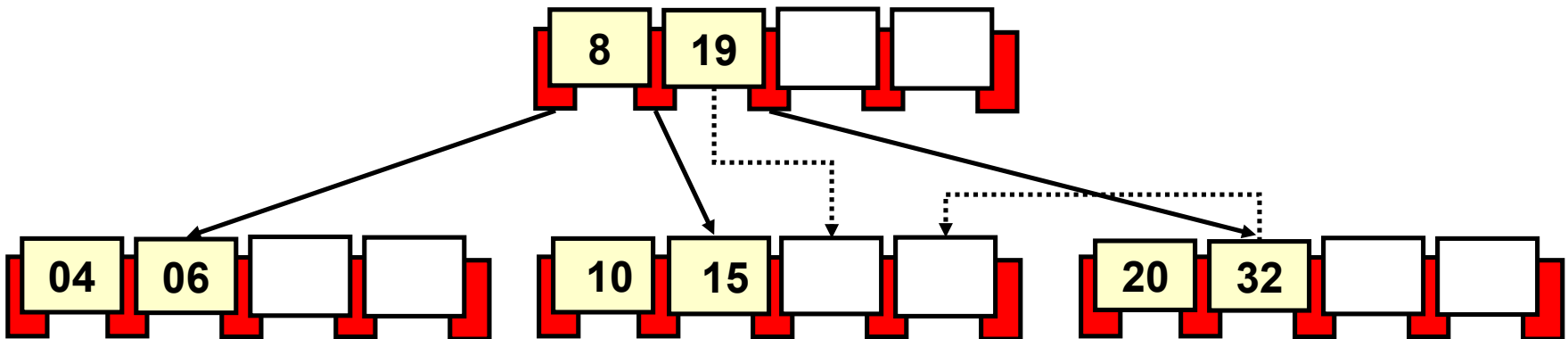


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 2b:** Página tiene $n = m$ claves (*underflow*) y ninguna página vecina puede ceder claves.
 - Las dos páginas adyacentes (izquierda y derecha) se encuentran en situación crítica.
 - La página se fusiona con la que vecina derecha (de no tenerla, se fusiona con la vecina izquierda).
 - » En la página resultante **se incluyen todos elementos de las dos páginas más el separador** que se elimina de la página padre.
 - » El **borrado del separador** fuerza un **borrado recursivo ascendente** que puede alcanzar a la raíz, en cuyo caso **disminuye la altura del árbol**.

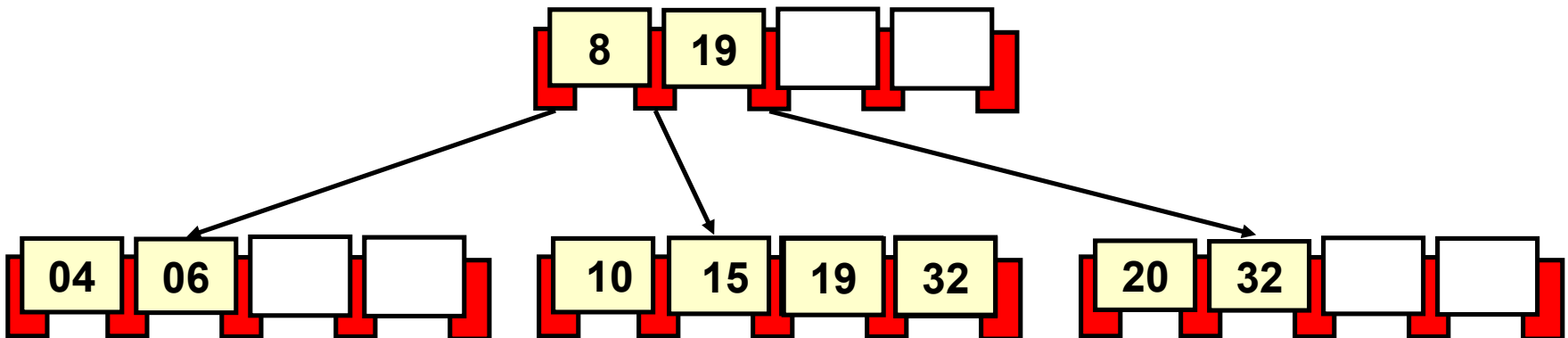


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 2b:** Página tiene $n = m$ claves (*underflow*) y ninguna página vecina puede ceder claves.
 - Las dos páginas adyacentes (izquierda y derecha) se encuentran en situación crítica.
 - La página se fusiona con la que vecina derecha (de no tenerla, se fusiona con la vecina izquierda).
 - » En la página resultante **se incluyen todos elementos de las dos páginas más el separador** que se elimina de la página padre.
 - » El **borrado del separador** fuerza un **borrado recursivo ascendente** que puede alcanzar a la raíz, en cuyo caso **disminuye la altura del árbol**.

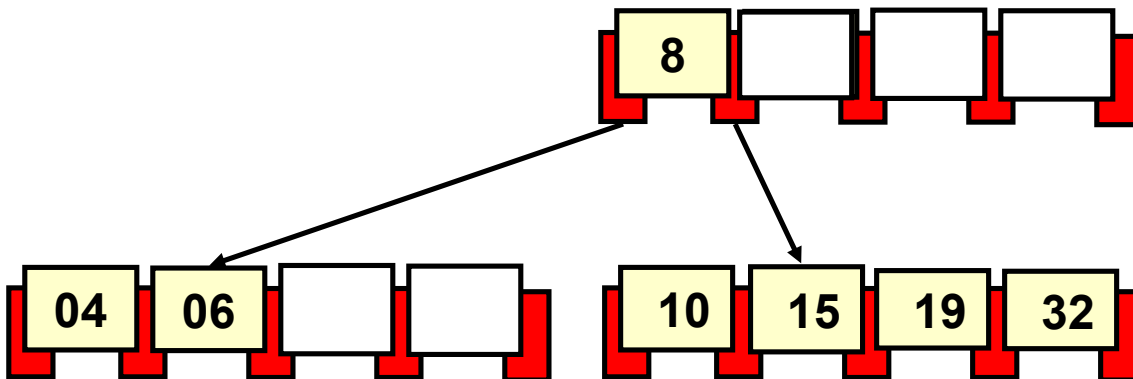


Árboles B (Bayer & McCreight)

Borrado

❖ Borrado de elemento en página hoja

- **Caso 2b:** Página tiene $n = m$ claves (*underflow*) y ninguna página vecina puede ceder claves.
 - Las dos páginas adyacentes (izquierda y derecha) se encuentran en situación crítica.
 - La página se fusiona con la que vecina derecha (de no tenerla, se fusiona con la vecina izquierda).
 - » En la página resultante **se incluyen todos elementos de las dos páginas más el separador** que se elimina de la página padre.
 - » El **borrado del separador** fuerza un **borrado recursivo ascendente** que puede alcanzar a la raíz, en cuyo caso **disminuye la altura del árbol**.



Árboles B (Bayer & McCreight)

Complejidad Temporal Borrado

❖ Caso Mejor

- Borrado caso 1 sobre un árbol B de altura mínima: $O(\log_{2n}(N)) + O(m) = O(\log_{2n}(N))$.

❖ Caso Peor

- El elemento se borra en un árbol de altura máxima sobre caso 2b, produciendo compactación desde las hojas hasta la raíz
 - $O(\log_n(N)) * O(n) = O(\log_n(N))$.

PLAYGROUND

❖ **Ejercicio Árbol B (borrado).** Partiendo del B-2 creado en el ejercicio anterior...

- a) Borrar la clave 11.
- b) Borrar la clave 15.
- c) Borrar la clave 6.
- d) Borrar la clave 16.
- e) Borrar la clave 10.
- f) Borrar la clave 12.
- g) Borrar la clave 28.
- h) Borrar la clave 27.

HOMEWORK

PLAYGROUND

- ❖ Consulte la entrada para el **Árbol-B** en la Wikipedia
 - Estudie **con detalle** todo el contenido de la entrada poniendo especial atención a:
 - Multi-modo: Combinar y Dividir.
 - Posibilidad de acceder concurrentemente a los árboles B en sistemas de bases de datos.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Colas de Prioridad

Objetivo

- ❖ Modelar estructuras lineales en las que los elementos se atienden en el **orden indicado por una prioridad** asociada.
 - Colas de impresión de documentos.
 - Gestión de tráfico aéreo.
 - Planificación de procesos en CPU.
 - Supervisión planes estratégicos y de emergencia.
 - Colas de Espera para Servicios Médicos.

Colas de Prioridad

Problema a Resolver

- ❖ Optimizar dos operaciones...
 1. Insertar elemento (con su prioridad asociada).
 2. Sacar elemento de máxima prioridad.

- ❖ Las colas de prioridad se suelen implementar mediante **Montículos Binarios**
 - Proporcionan **complejidad $\log_2(n)$** para las dos operaciones.
 - Se pueden construir con **vectores** (no son necesarias referencias).

Montículos Binarios

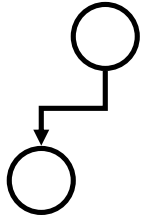
¿Qué es un Montículo Binario?

- ❖ Es un árbol binario completo con la excepción del nivel inferior.
 - Este nivel se rellena de izquierda a derecha.

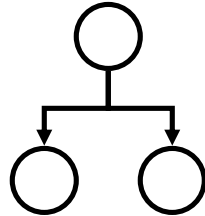
n = 1



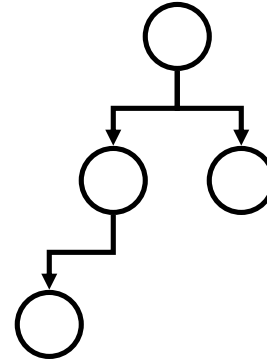
n = 2



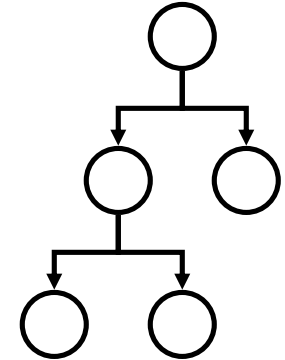
n = 3



n = 4



n = 5



Rango de altura en un Montículo Binario

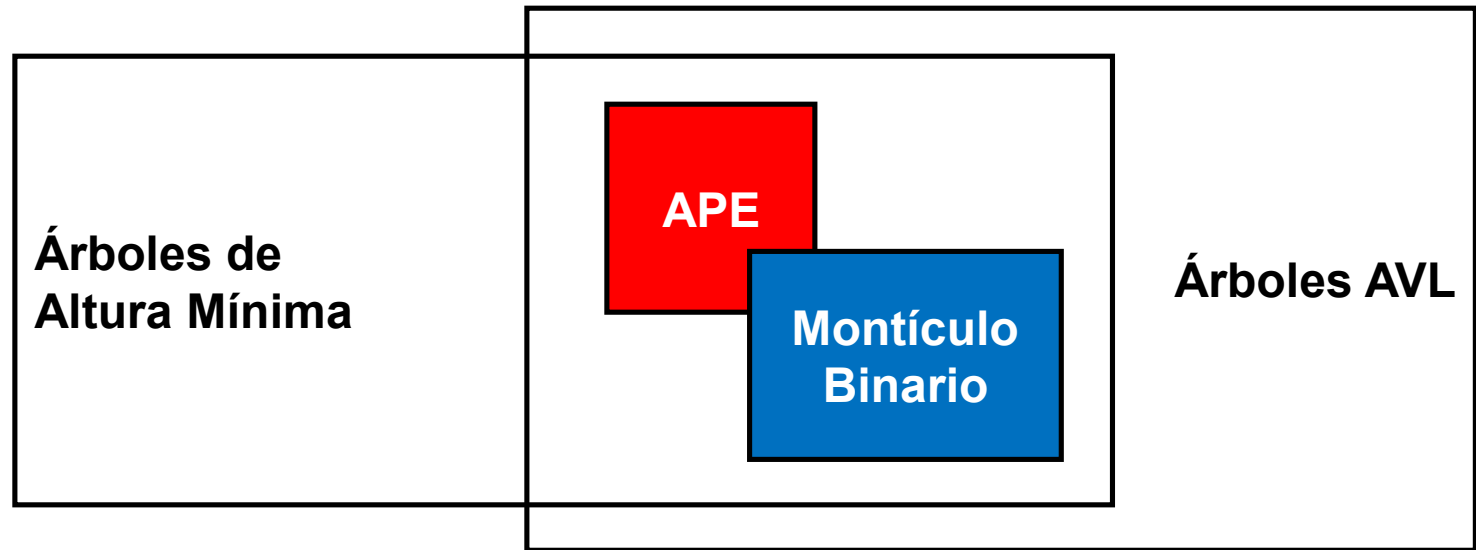
$$h = \lceil \log_2 n \rceil + 1$$

$$O(h) \leq O(\log_2 n)$$

Montículos Binarios

Propiedades

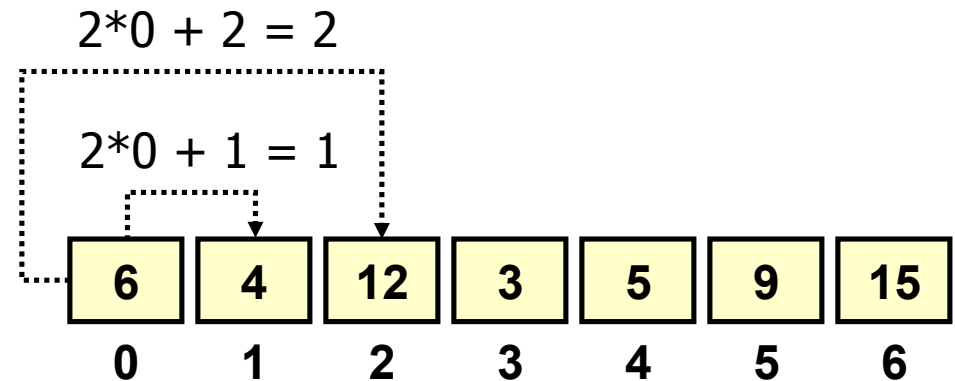
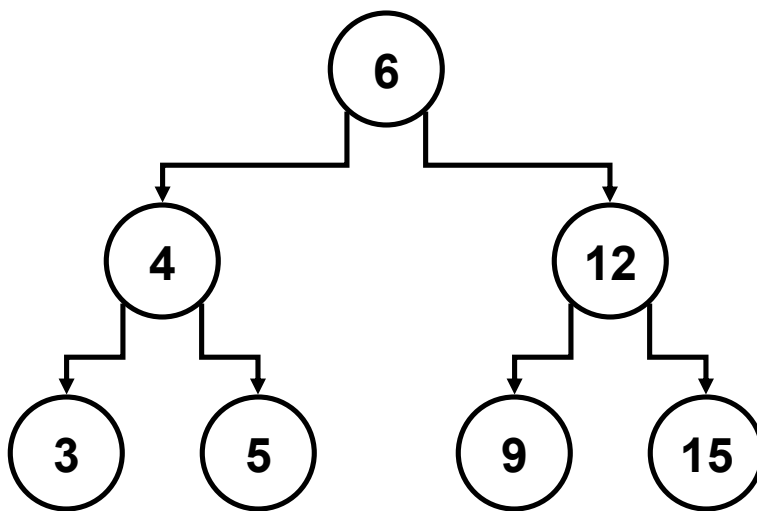
- ❖ Todo Montículo Binario es un Árbol de Altura Mínima



Montículos Binarios

Propiedades

- ❖ Dada la estructura fija del árbol binario, éste puede representarse en un vector sin necesidad de referencias
 - La raíz del árbol se almacena en la primera celda del vector.
 - Dado un nodo situado en la posición i del vector:
 - Su hijo **izquierdo** se almacenará en la posición $2i + 1$.
 - Su hijo **derecho** se almacenará en la posición $2i + 2$.



Montículos Binarios

Relación de orden (suponiendo claves no repetidas)

❖ Montículo de Mínimos

- Todo nodo tiene una clave **menor** que la de sus hijos.
- El **menor elemento** se encuentra en la raíz (posición 0 del vector).
 - Optimiza las operaciones *Add* y *getMin*.

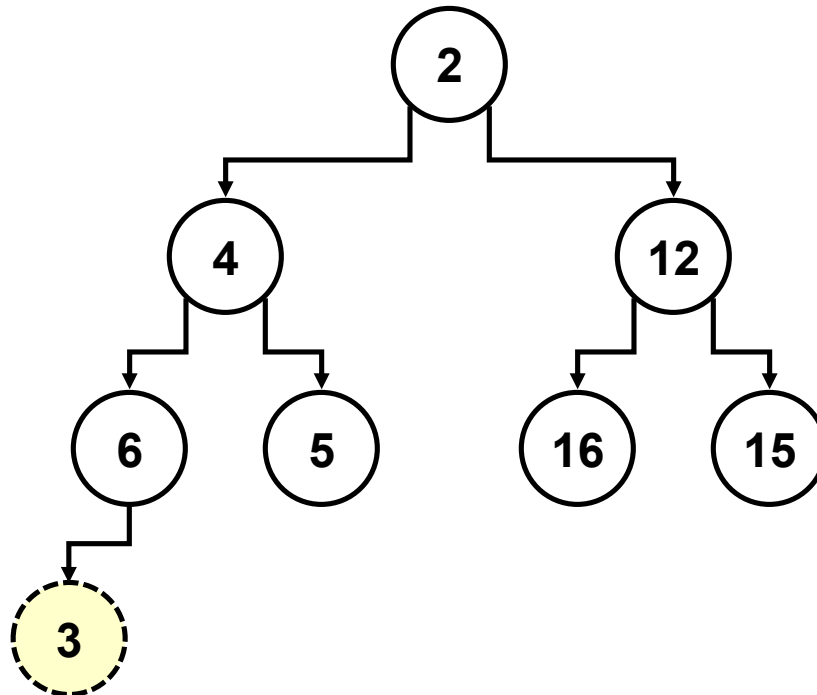
❖ Montículo de Máximos

- Todo nodo tiene una clave **mayor** que la de sus hijos.
- El **mayor elemento** se encuentra en la raíz (posición 0 del vector).
 - Optimiza *Add* y *getMax*.

Montículos Binarios

Inserción (mediante Filtrado Ascendente)

1. Colocar el elemento a insertar en la última posición del vector.
2. Repetir hasta que el elemento llegue a la raíz (posición 0 del vector) o sea mayor que su padre.
 - Si elemento es menor que el ubicado en la posición $E[(i-1)/2]$ (su padre actual), intercambiarlos.



Complejidad caso mejor: $O(1)$

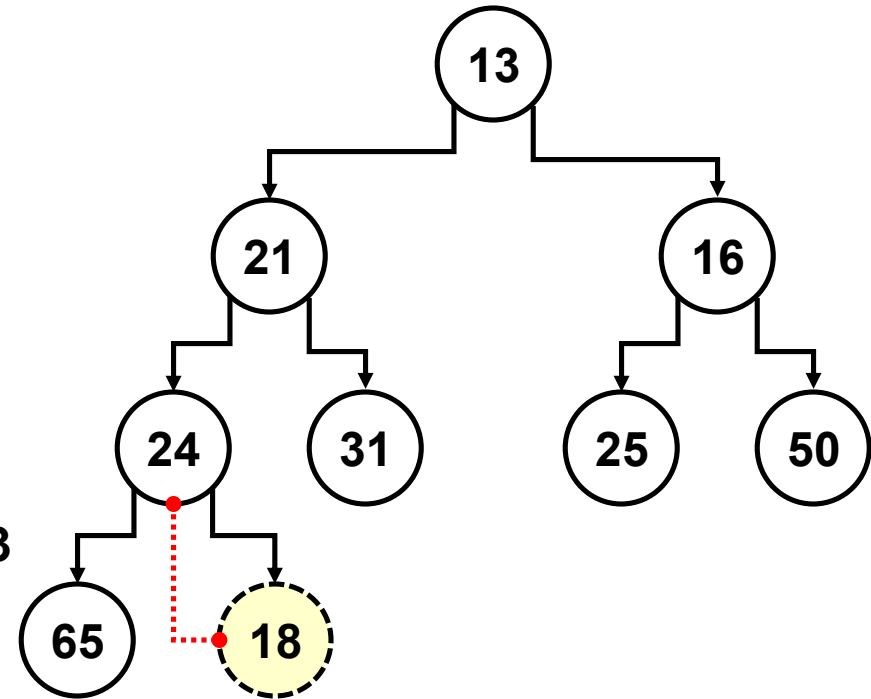
Complejidad caso peor: $O(\log_2 n)$

Montículos Binarios

Ejercicio

Comparar posición $E[(i-1)/2]$

Comparar posición $E[(8-1)/2] = E[7/2] = 3$



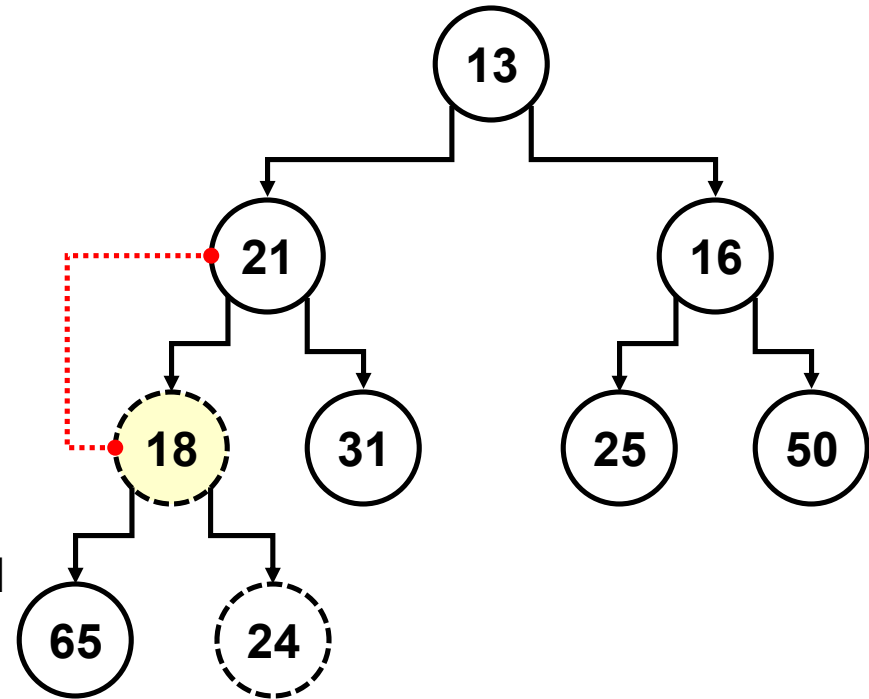
it	0	1	2	3	4	5	6	7	8
1	13	21	16	24	31	25	50	65	18

Montículos Binarios

Ejercicio

Comparar posición $E[(i-1)/2]$

Comparar posición $E[(3-1)/2] = E[2/2] = 1$



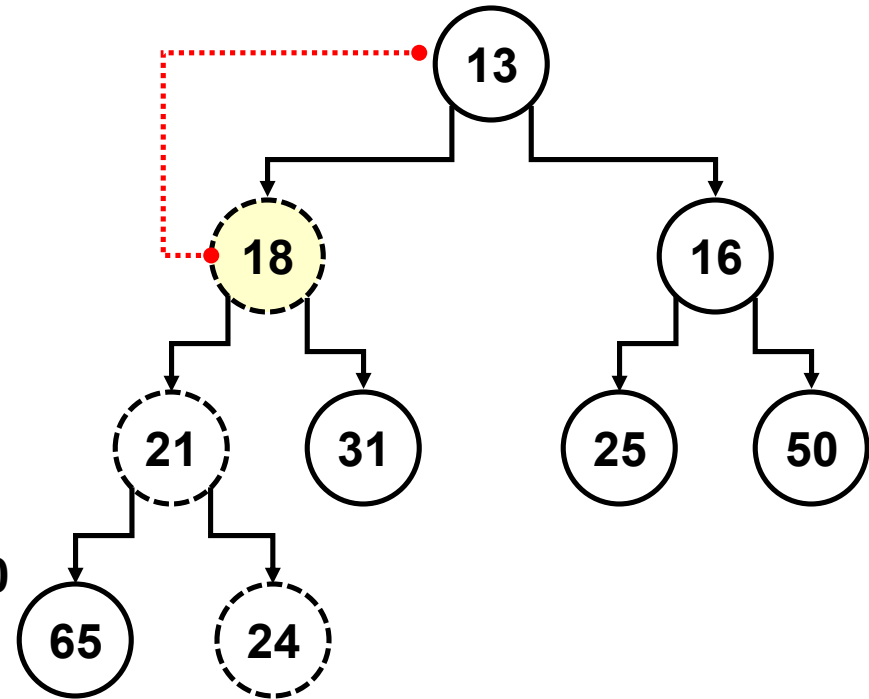
it	0	1	2	3	4	5	6	7	8
1	13	21	16	24	31	25	50	65	18
2	13	21	16	18	31	25	50	65	24

Montículos Binarios

Ejercicio

Comparar posición $E[(i-1)/2]$

Comparar posición $E[(1-1)/2] = E[0/2] = 0$



it	0	1	2	3	4	5	6	7	8
1	13	21	16	24	31	25	50	65	18
2	13	21	16	18	31	25	50	65	24
3	13	18	16	21	31	25	50	65	24

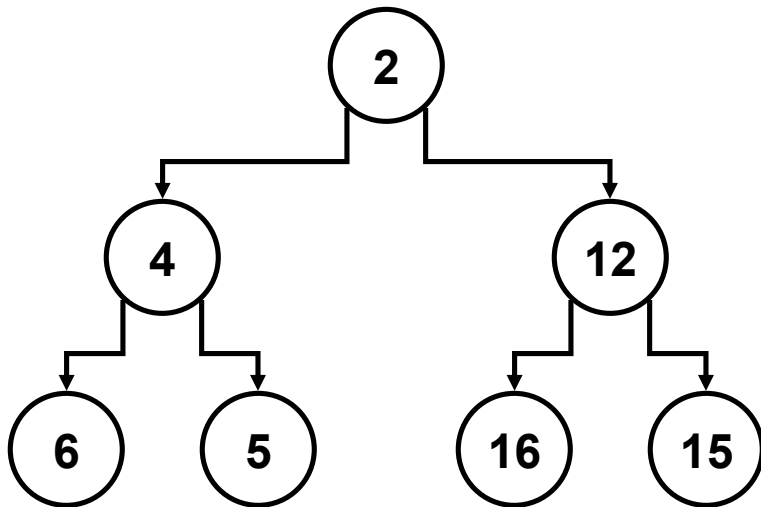
Montículos Binarios

Sacar (mediante Filtrado Descendente)

1. Devolver el elemento situado en la raíz (mínimo).
2. Colocar el último elemento del vector en la raíz y aplicar filtrado descendente usándolo como pivote.
3. Repetir hasta que el pivote llegue a ser hoja o sea **menor que sus dos hijos**.
 - Intercambiar la posición del pivote con la de aquel de sus dos hijos **que sea menor**.

$O(1)$

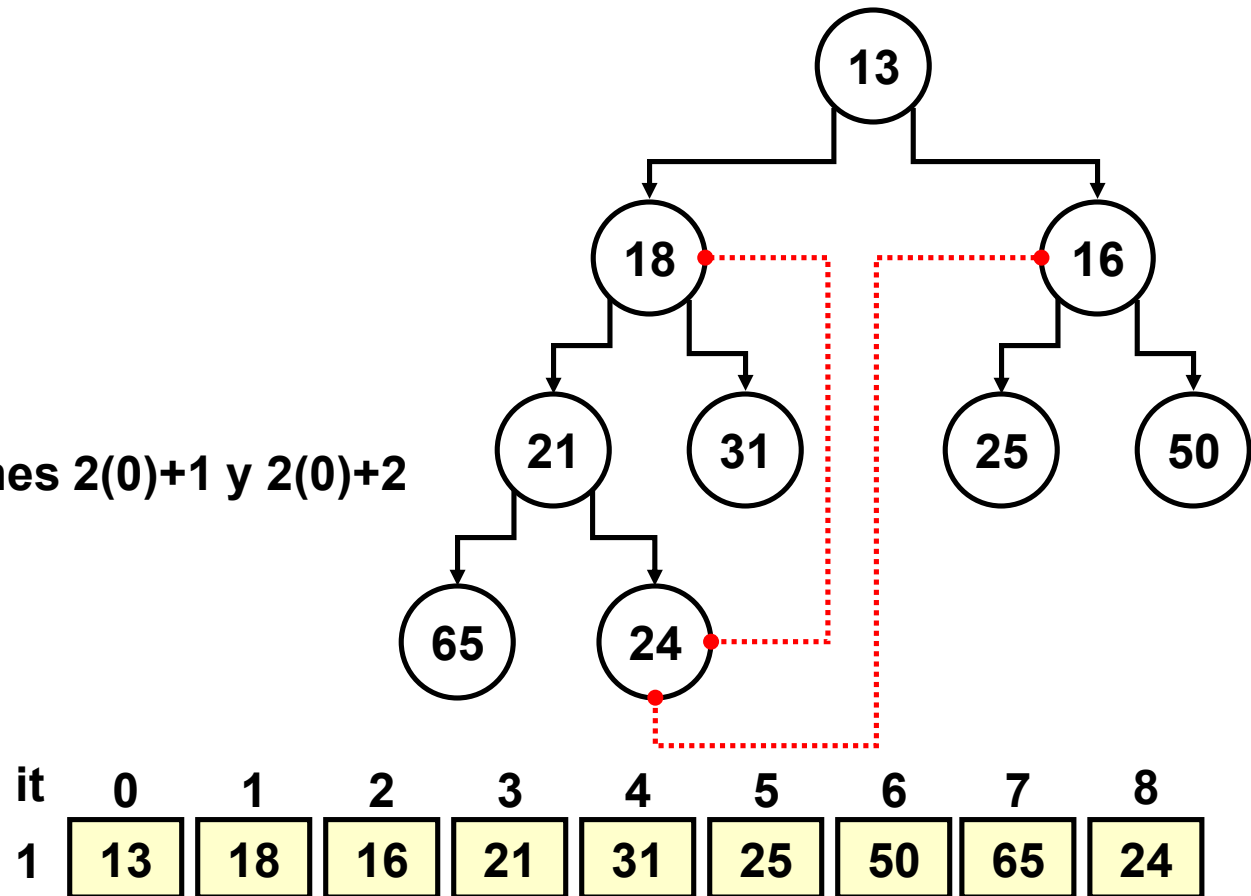
$O(\log_2 n)$



Montículos Binarios

Ejercicio 1

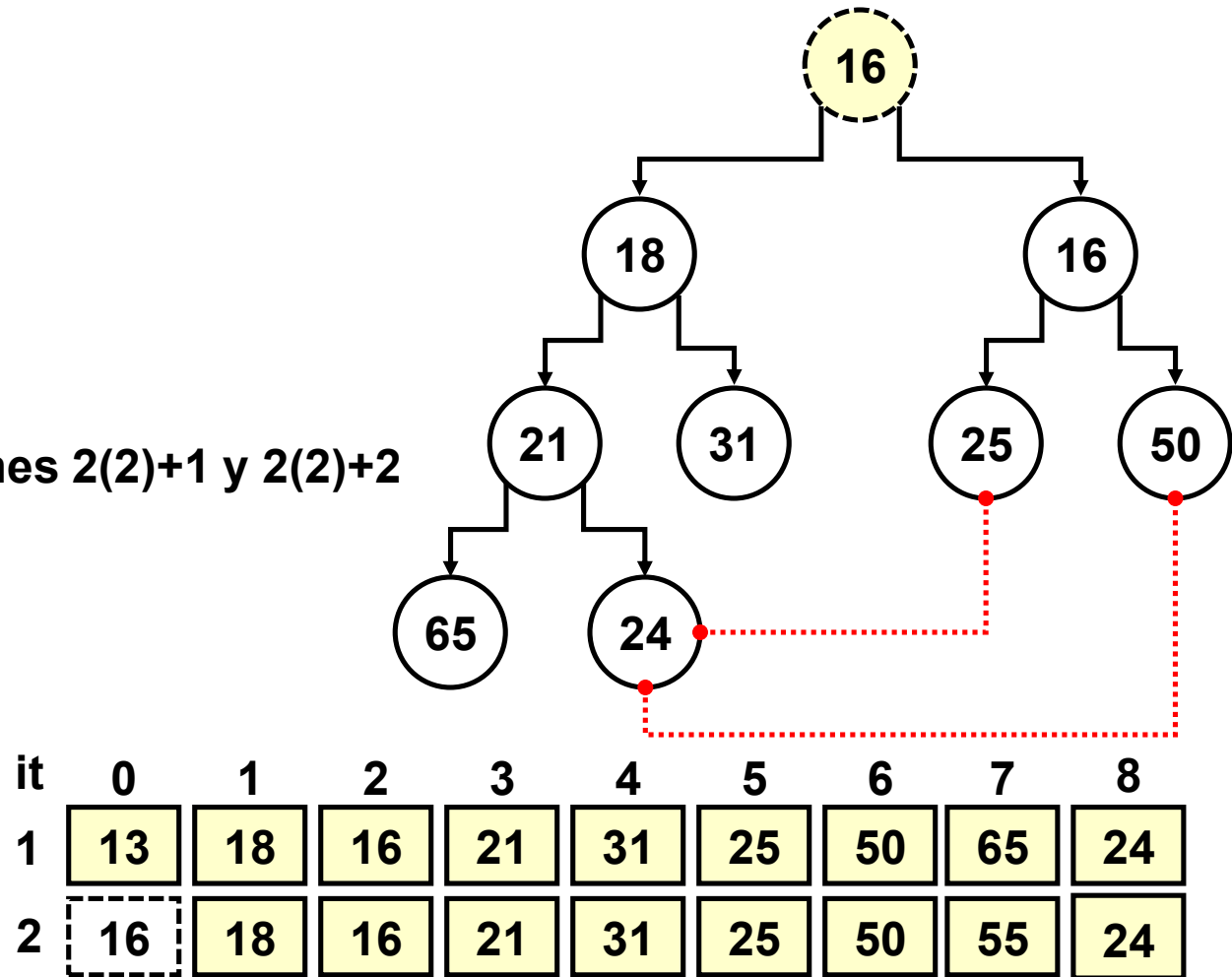
Comparar con posiciones $2(0)+1$ y $2(0)+2$



Montículos Binarios

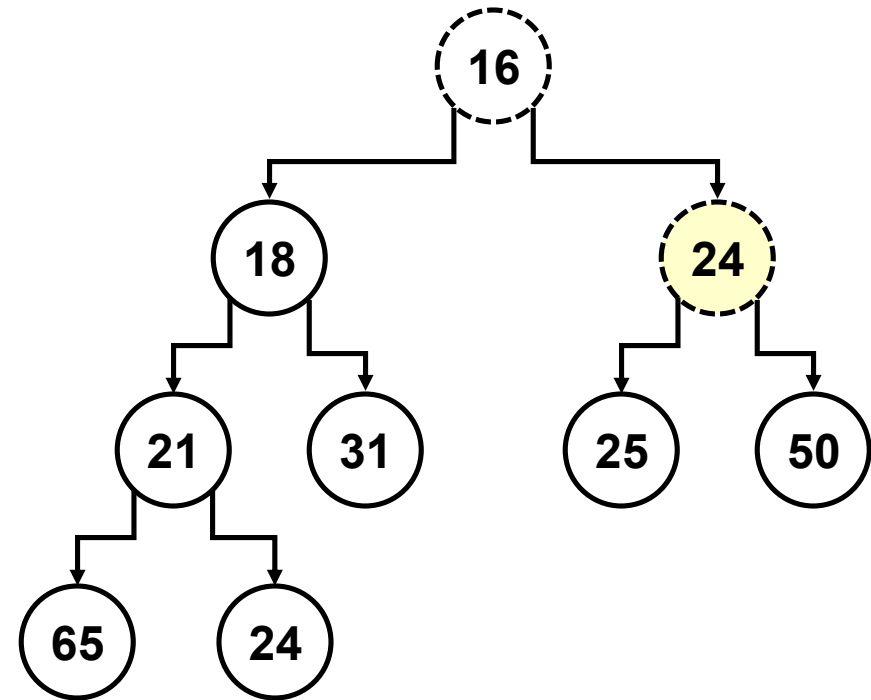
Ejercicio 1

Comparar con posiciones $2(2)+1$ y $2(2)+2$



Montículos Binarios

Ejercicio 1

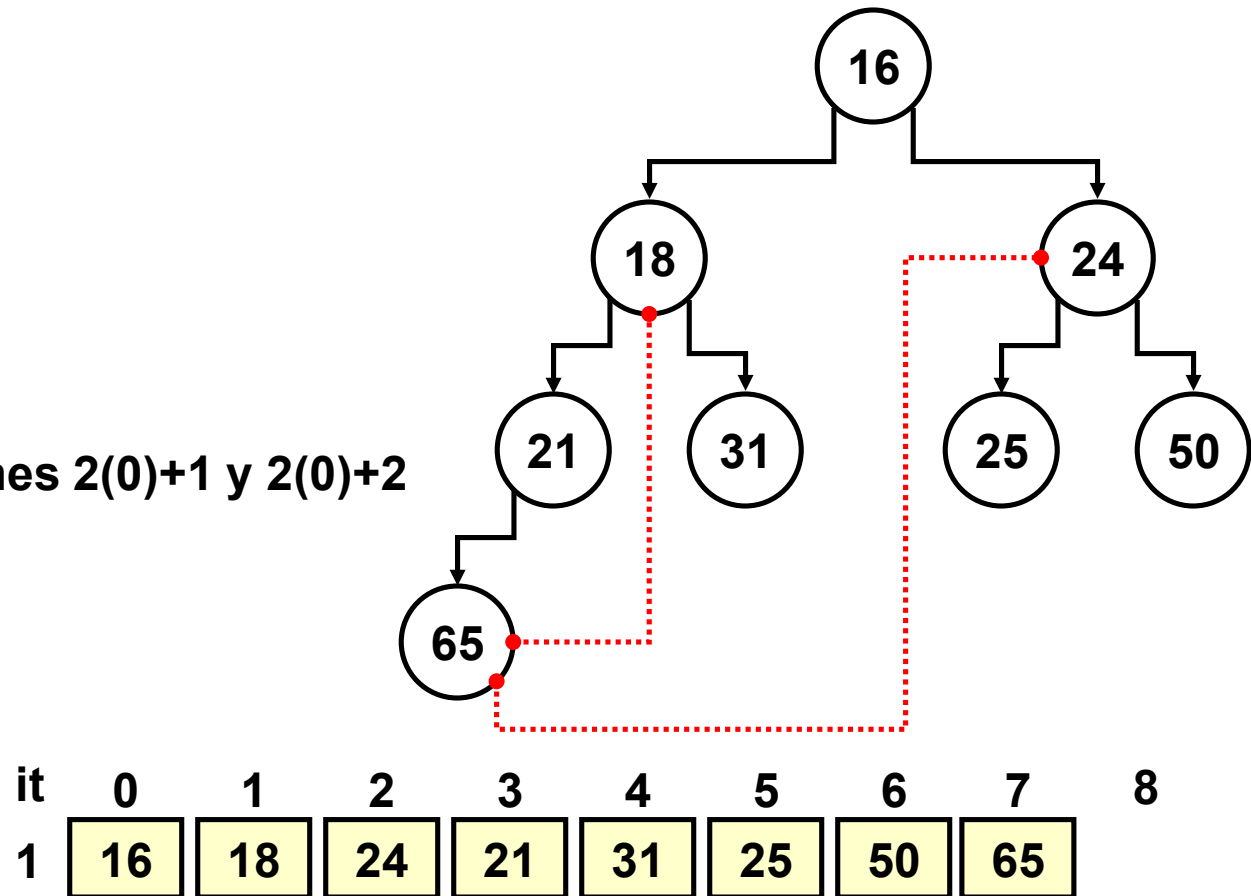


it	0	1	2	3	4	5	6	7	8
1	13	18	16	21	31	25	50	65	24
2	16	18	16	21	31	25	50	65	24
3	16	18	24	21	31	25	50	65	24

Montículos Binarios

Ejercicio 2

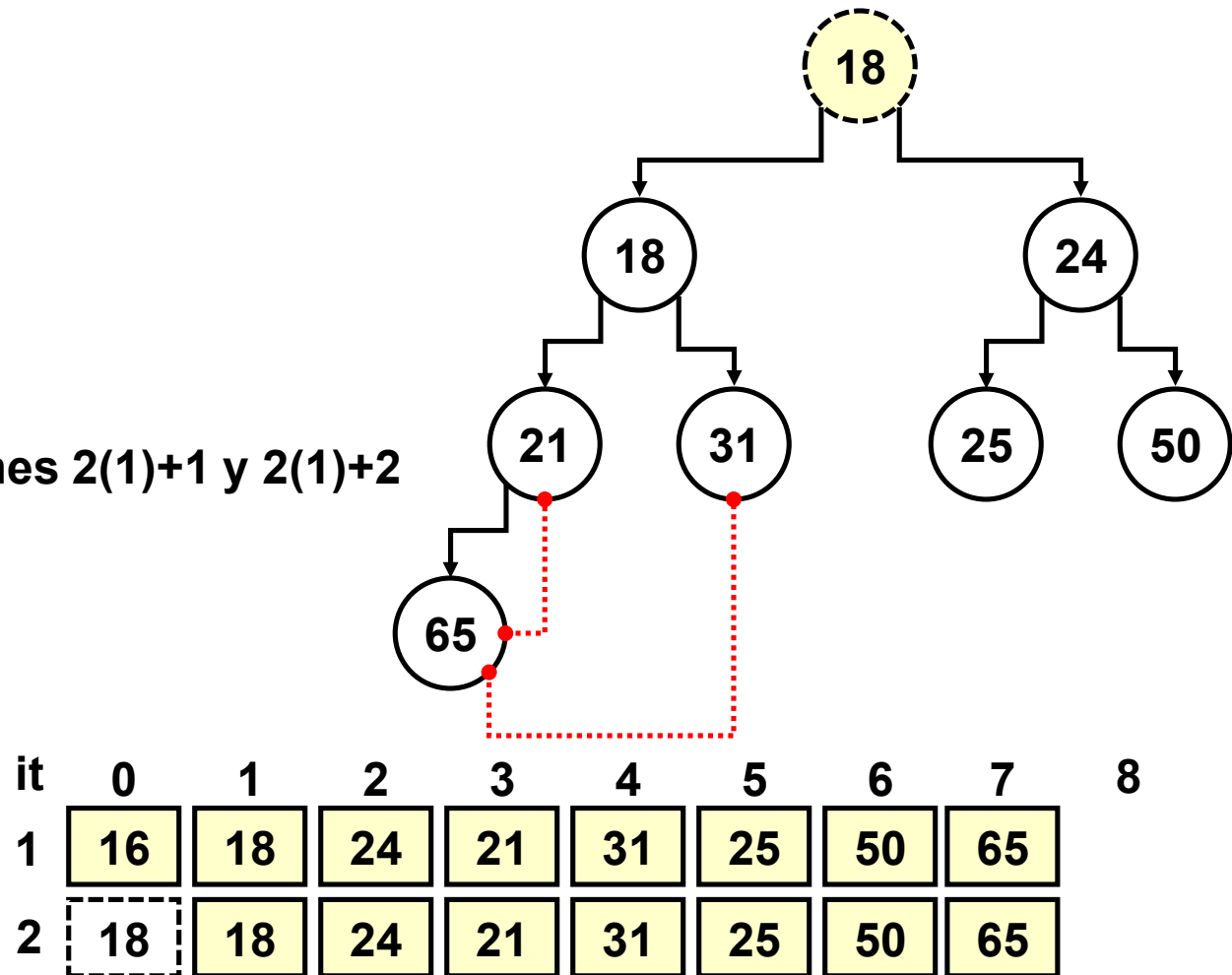
Comparar con posiciones $2(0)+1$ y $2(0)+2$



Montículos Binarios

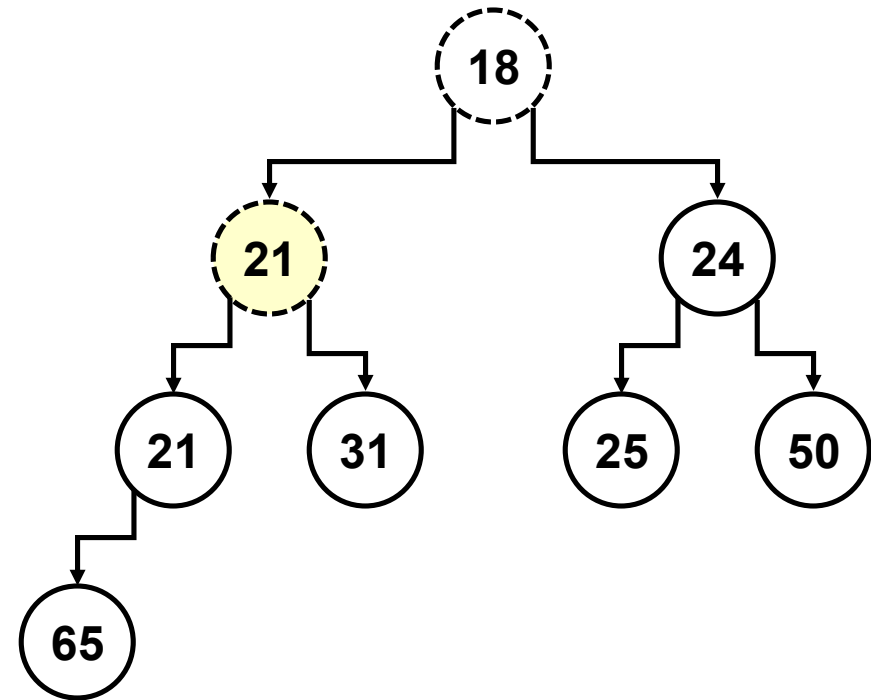
Ejercicio 2

Comparar con posiciones $2(1)+1$ y $2(1)+2$



Montículos Binarios

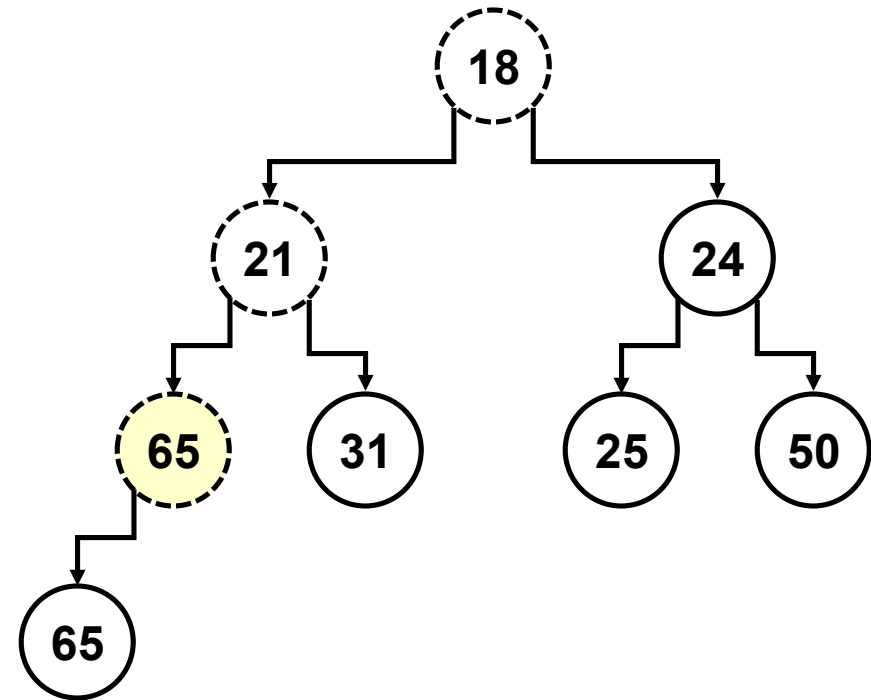
Ejercicio 2



it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	
2	18	18	24	21	31	25	50	65	
3	18	21	24	21	31	25	50	65	

Montículos Binarios

Ejercicio 2



it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	
2	18	18	24	21	31	25	50	65	
3	18	21	24	21	31	25	50	65	
4	18	21	24	65	31	25	50	65	

Operaciones Especiales con Montículos

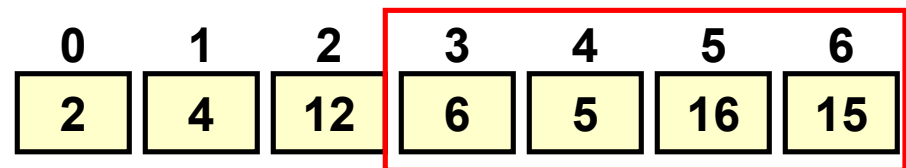
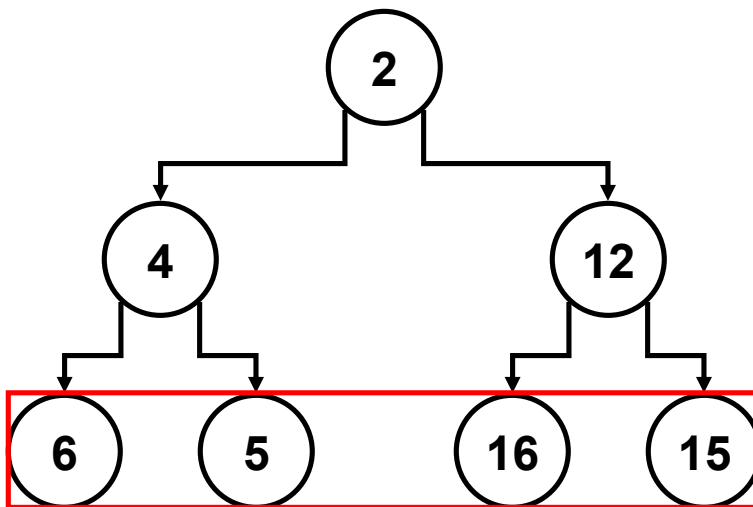
Devolver máximo elemento

$O(n)$

Búsqueda secuencial en la zona del vector comprendida en el rango:

$[size/2, size]$.

- ❖ Los elementos de mayor peso se encuentran en las hojas
 - Solo es necesario explorar la mitad del vector.

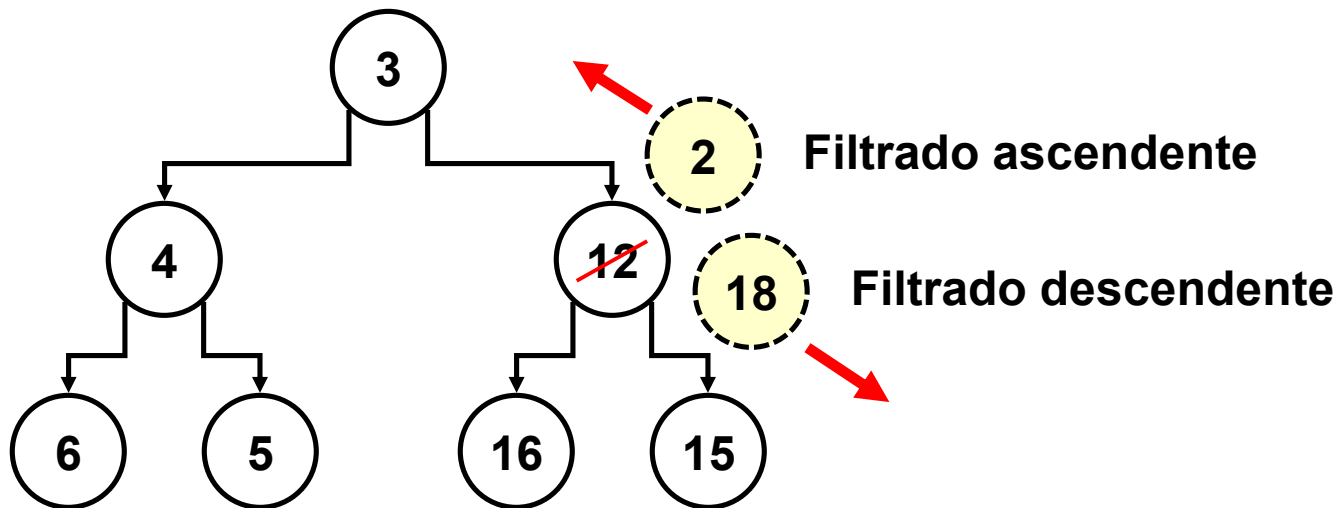


Operaciones Especiales con Montículos

Cambiar la prioridad de un elemento

$O(\log_2 n)$

1. Modificar el valor de la prioridad.
2. Si el nuevo valor es menor que el original
 - Aplicar filtrado ascendente
- else
 - Aplicar filtrado descendente.

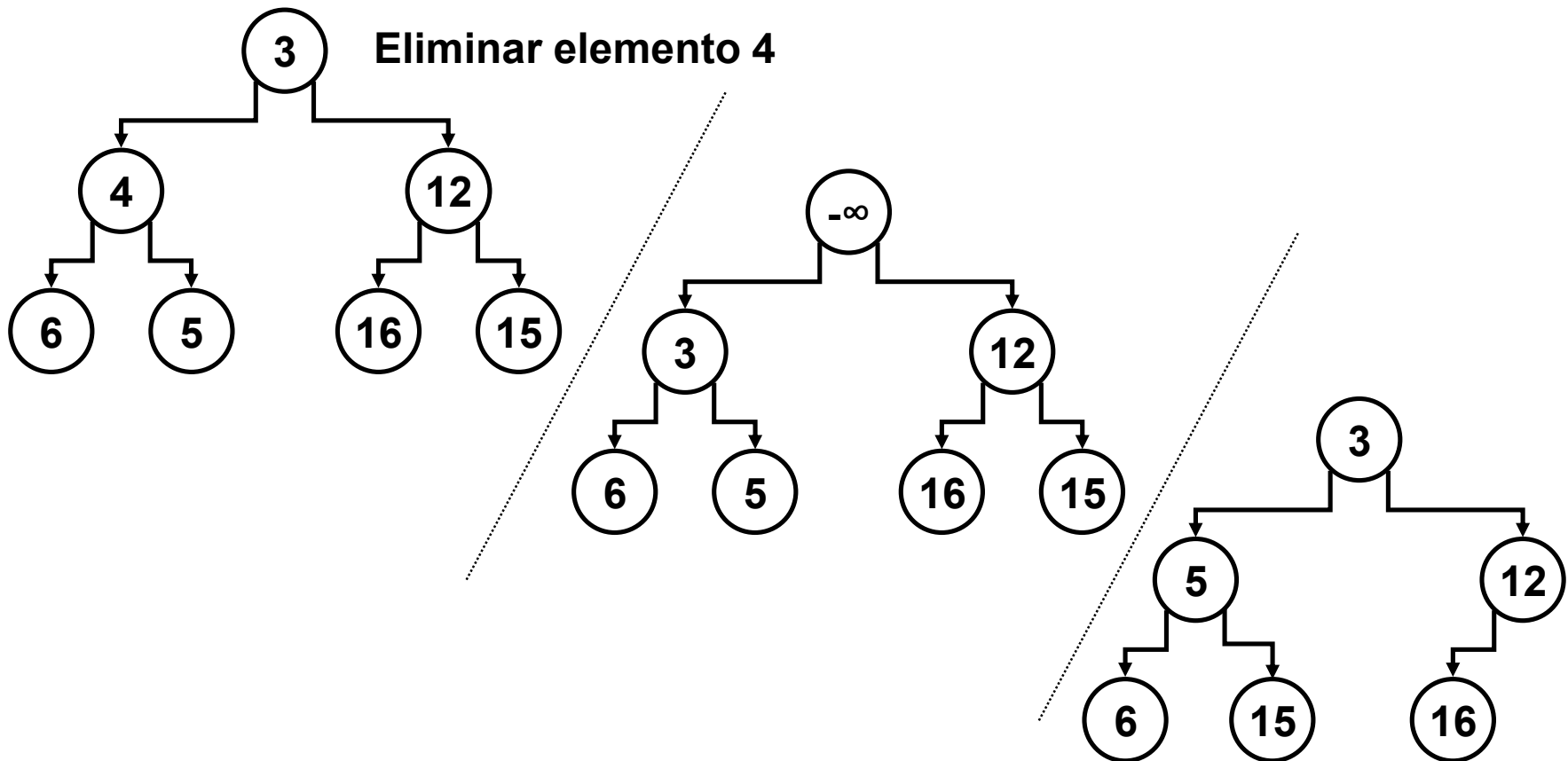


Operaciones Especiales con Montículos

Eliminar elemento

$O(\log_2 n)$

1. Cambiar su prioridad a $-\infty$ para subirlo a la raíz.
2. Invocar al método sacar().



HOMEWORK

PLAYGROUND

- ❖ Consulte la entrada para el **Montículo Binario** en la Wikipedia
 - Estudie **cuidadosamente** todo el contenido de la entrada.
 - Preste especial atención a porqué la reorganización del montículo requiere un tiempo $O(\log n)$ en la inserción de elementos.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

ISBN 978-1-300-67089-6

© 2012 – 2015 Martín González Rodríguez

Estructuras Diccionario

Dr. Martin Gonzalez-Rodriguez

Estructuras de Datos Diccionario

Objetivo

- ❖ Almacenar objetos sin relaciones entre sí para su recuperación de la manera más rápida posible.
 - Máxima velocidad de acceso.
 - Usan grandes cantidades de memoria.
 - Ampliamente utilizadas en **sistemas de caché en la web** y acceso a **bases de datos**.

Estructuras de Datos Diccionario

Objetivo

- ❖ Eficiencia temporal $O(1)$ en operaciones de acceso
 - Se resiente la eficiencia en el resto de las operaciones.

Método	Complejidad
Insertar	$O(1)$
Buscar	$O(1)$
Borrar	$O(1)$
print	$O(n)$
Obtener Máximo	$O(n)$
Obtener Mínimo	$O(n)$

Tablas Hash

Componentes Básicos

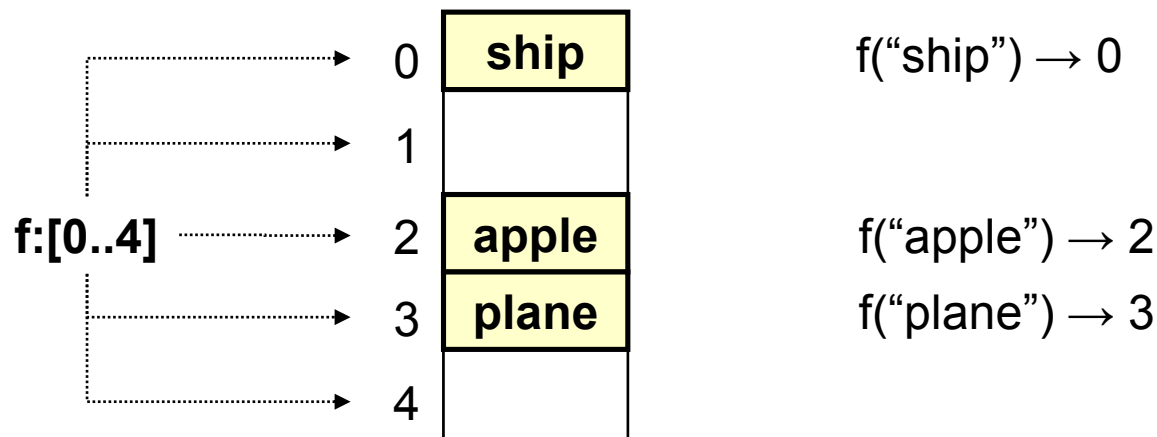
HashTable class

```
class HashTable<T> {  
    private final static int B = 5; //size  
    private ArrayList<HashNode<T>> associativeArray;  
  
    public HashTable() {  
        associativeArray = new ArrayList<HashNode<T>>(B);  
    }  
  
    private int f (T element){  
        return (...);    // converts T to an int value in the range  
                        // [0, B-1].  
    }  
}
```


Función Hash

Convierte claves en índices

- ❖ Recibe la clave de un objeto en el dominio del problema.
 - Usualmente *String* o *int*.
- ❖ Devuelve la posición en la que debería alojarse el elemento en el *associativeArray*.
 - Rango de f : $[0, B-1]$.

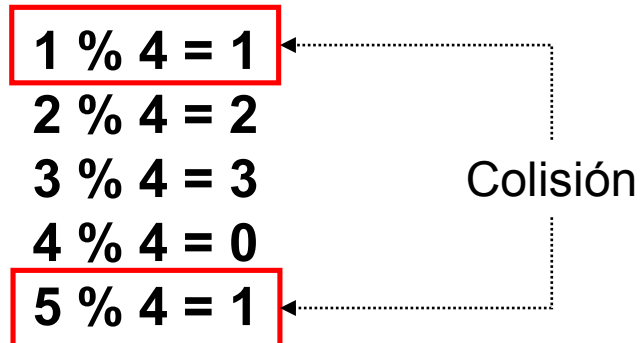


Función Hash

Función f para claves enteras

```
private int f (T element)
{
    return (element.hashCode() % B);
}
```

- ❖ Es una operación fácil y rápida de ejecutar.
 - Si las claves son aleatorias, distribuye los elementos uniformemente.



Función Hash

Colisiones

- ❖ Dos elementos x e y son **sinónimos** si...
 - $f(x) == f(y)$
 - Los elementos **sinónimos** producen **colisiones** sobre la misma posición del vector.

- ❖ Tratamiento de las colisiones:
 - **Protección Activa**
 - Evitar la colisión (diseño de la función hash perfecta).

 - **Protección Pasiva**
 - Dos o más elementos comparten la misma posición del vector.

 - **Redispersión**
 - Aumentar o disminuir el tamaño del vector (B) de forma dinámica en base al número de elementos que contiene.

Función Hash

Función f perfecta

$$P(f(X_1)=0) == P(f(X_2)=1) = \dots == P(f(X_m)=B-1) == 1/B$$

- ❖ Garantiza el menor número de colisiones posibles.
 - Por **cada n elementos** a insertar, tan **sólo** se producirían **n/B** colisiones.

$$10 \% 10 = 0$$

$$20 \% 10 = 0$$

$$30 \% 10 = 0$$

$$40 \% 10 = 0$$

$$50 \% 10 = 0$$

$$10 \% 7 = 3$$

$$20 \% 7 = 6$$

$$30 \% 7 = 2$$

$$40 \% 7 = 5$$

$$50 \% 7 = 1$$

- ❖ ¡B debería ser un número primo!
 - Ayuda a reducir colisiones cuando las claves **no son aleatorias**.

Función Hash

HashCode para claves String (Versión 1)

```
public int convert (String t){ // <-> t.hashCode()  
    int result = 0;  
  
    for (int i=0; i<t.length(); i++)  
        result += (int) t.charAt(i);  
  
    return (result);  
}  
  
private int f (String element)  
{  
    return (convert(element) % B);  
}
```

- ❖ Convierte la cadena a un entero para luego aplicar la función de dispersión.
 - La función *convert* suma los códigos de representación de cada letra de la cadena.
 - (Códigos ASCII, EBDIC, etc).

Función Hash

Ejercicio

- ❖ Convertir la cadena “PLANE” suponiendo que el código para la letra A es 65.

Letra	Código
P	80
L	76
A	65
N	78
E	69
<i>Total</i>	368

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Función Hash

Ejercicio

- ❖ Calcular el rango de f suponiendo...
 - Cadenas de una longitud máxima igual a 8 caracteres.
 - Rango de códigos $[0, 127]$.
 - B igual a 10.007 posiciones.

Rango $convert$ (*String* t)

$[8*0, 8*127] = [0, 1.016]$

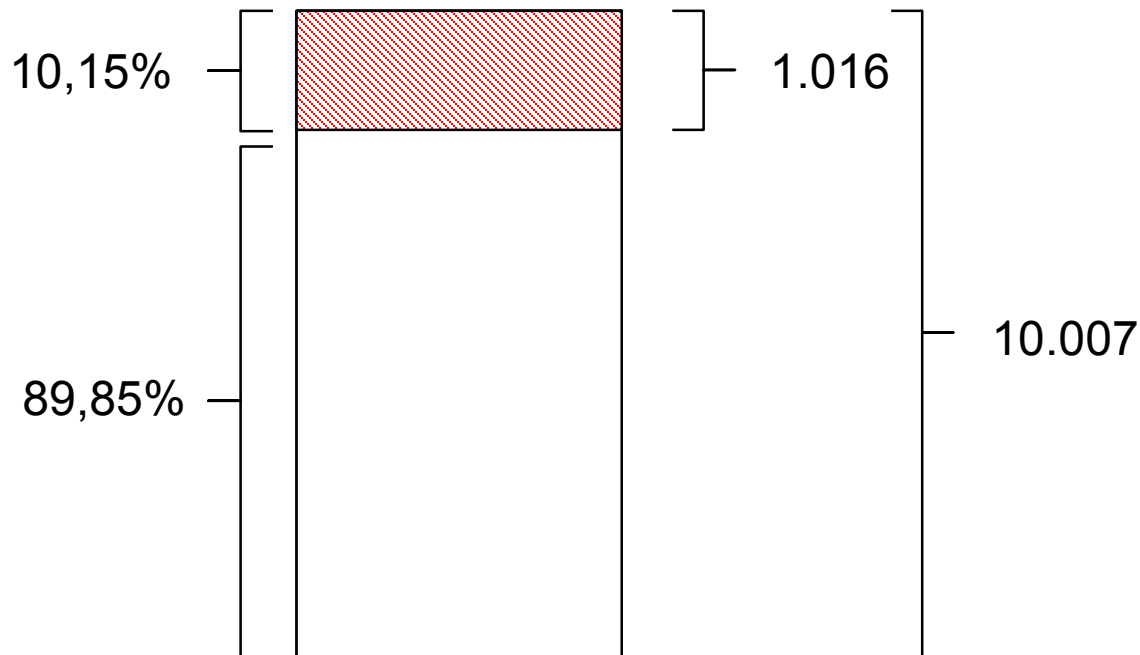
Rango f (*String* t)

$[0, 1.016] \% 10.007 = [0, 1.016]$

Función Hash

Desventajas

- ❖ Si **B es grande** y la **longitud de la clave es pequeña**, la dispersión se concentrará en la zona superior del vector.
 - Si la longitud es pequeña, la suma de los códigos también lo será.
 - Al aplicar el operador módulo (%) entre la pequeña suma y un valor de B grande, el resultado obtenido será muy pequeño.



Función Hash

HashCode para claves String (Versión 2)

```
public int convert (String t){// <-> t.hashCode()
    int result = 0;
    int k =(t.length()>3)?3:t.length();

    for (int i=0; i<k; i++)
        result += (int) Math.pow(27, 2-i) * (int) t.charAt(i);

    return (result);
}
```

- ❖ Asigna un peso a cada carácter en función de su posición.
 - El valor del peso (27) se corresponde con la longitud del alfabeto.
 - La ponderación es 27^{2-i} siendo i la posición del carácter en la cadena.
 - Se puede restringir el número de caracteres analizados a un límite máximo k por razones de eficiencia.
 - En el ejemplo, $k \leq 3$.
 - La operación de multiplicación consume mucho tiempo de CPU.

$$\text{Convert ("PLANE")} = P * 27^2 + L * 27^1 + A * 27^0$$

Función Hash

Ejemplo

- ❖ Convertir la cadena “PLANE” suponiendo que el código para la letra A es 65.

Letra	Código Ponderado	Total
P	$80 \cdot 27^2$	58.320
L	$76 \cdot 27^1$	2.052
A	$65 \cdot 27^0$	65
N	-	-
E	-	-
Total		60.437

$$60.437 \% 10.007 = 395$$

La versión 1 de *Convert*(“PLANE”) obtenía 358 ($358 \% 10.007$) = 358

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Función Hash

Desventajas

- ❖ Palabras que empiezan con la misma combinación de letras conducen a colisiones.
 - “**PLANE**”, “**PLANING**”, “**PLASTIC**”, etc.
- ❖ Suponiendo un vector de tamaño $B = 10.007...$
 - En **Teoría...**
 - Para $k=3$ existen $27*26*25$ (17.550) combinaciones distintas de inicio de palabra en el dominio de la función *convert*.
 - Dado que $17.550 > 10.007$, los elementos se distribuyen por todo el vector.
 - En la **Práctica...**
 - De las 17.550 combinaciones posibles solo tienen sentido 2.851 en lengua castellana.
 - » Por ejemplo, no existen palabras que empiecen por ZYV, ZVW, XYV, etc.
 - Con 2.851 palabras válidas **tan solo se emplea un 28,4%** de las 10.007 posiciones disponibles en el vector.

Se hace necesario explorar todos los caracteres de la cadena

Función Hash

HashCode para claves String (Versión 3)

```
public long convert (String t){ // <-> t.hashCode()
    long result = 0;

    for (int i=0; i<t.length(); i++)
        result += (int) Math.pow(32, t.length()-i-1) * (int) t.charAt(i);

    return (result);
}
```

❖ ¿Cómo optimizar la función para analizar toda la cadena?

- Se **utiliza 32 como peso**, en lugar de 27.
 - A nivel binario, multiplicar por 32 equivale a un desplazamiento de 5 bits (operación mucho más rápida que una multiplicación).
 - » $32 = 2^5$.

$$\text{Convert ("PLANE")} = P * 32^4 + L * 32^3 + A * 32^2 + N * 32^1 + E * 32^0$$

Función Hash

HashCode para claves String (Versión 4)

```
public long convert (String t){// <-> t.hashCode()  
    long result = (int) t.charAt(0);  
  
    for (int i=1; i<t.length(); i++)  
        result = (32 * result) + (int) t.charAt(i);  
  
    return (result);  
}
```

❖ Utilización de la **Regla de Horner**

- Minimiza el uso de las multiplicaciones utilizando una representación alternativa del polinomio.

$$\text{Convert ("PLANE")} = P * 32^4 + L * 32^3 + A * 32^2 + N * 32^1 + E * 32^0$$

$$\text{Convert}_{\text{Horner}} (\text{"PLANE"}) = (((((P * 32) + L) * 32) + A) * 32) + N) * 32 + E$$

Función Hash

HashCode para claves String (Versión 5)

```
public long convert (String t) { // <-> t.hashCode()
    long result = (int) t.charAt(0);

    for (int i=0; i<t.length(); i++)
        result = ((32 * result) + (int) t.charAt(i)) % B;

    return (result);
}
```

❖ Eliminación del *Overflow*

- Durante el cálculo se generan **cifras tan grandes que no puedan ser almacenadas**.
- Se debe aplicar el operador **resto (%)** en cada iteración para reducir el tamaño de las cifras parciales
 - Se elimina el *overflow* a costa de una penalización temporal.

$$f(\text{"PLANE"}) = \\ ((((((P * 32) + L) \% B * 32) + A) \% B * 32) + N) \% B * 32 + E) \% B$$

Función Hash

Ejemplo

- ❖ Convertir la cadena “PLANE” suponiendo que el código para la letra A es 65.

Letra	Código Ponderado	Total
P	$80 \cdot 32^4$	83.886.080
L	$76 \cdot 32^3$	2.490.368
A	$65 \cdot 32^2$	66.560
N	$78 \cdot 32^1$	2.496
E	$69 \cdot 32^0$	69
<i>Total</i>		86.445.573

$$86.445.573 \% 10.007 = 5.107$$

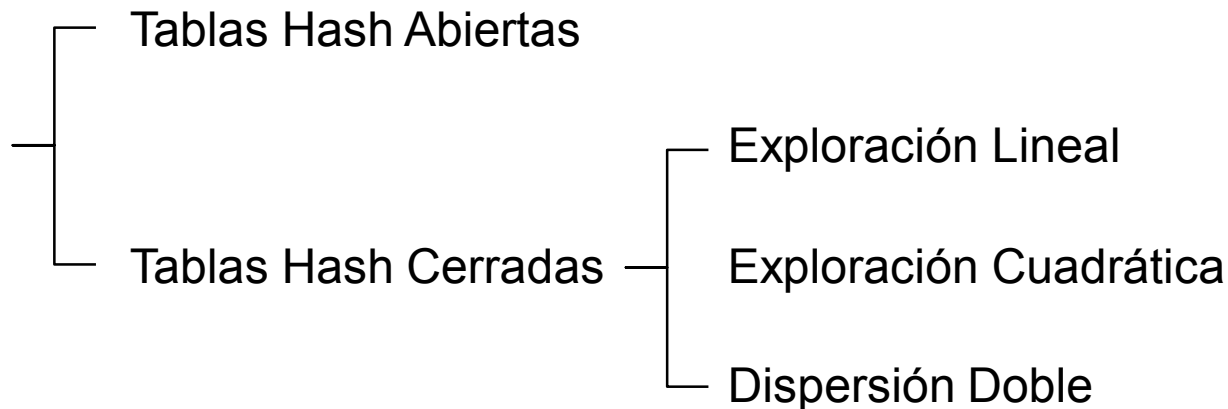
La versión 2 de *Convert*(“PLANE”) obtenía $60.437 \% 10.007 = 395$

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Protección Pasiva

Las Colisiones son inevitables a largo plazo...

- ❖ Cuanto **menor** sea B **mayor** será la probabilidad de colisión.
 - La certeza se alcanza con...
 - $B = 1$.
 - Dominios de problema en los que existen elementos de clave repetida.
- ❖ Dos o más elementos comparten la misma posición del vector.
 - Existen varias formas de gestionar elementos en colisión.



Tablas Hash Abiertas

Tablas Hash Abiertas

- ❖ Cada celda contiene una estructura de datos dinámica encargada de almacenar los sinónimos.
 - LinkedList.
 - AVLTree.

HashTable class

$O(B) = O(1)$

```
public class HashTable<T>
{
    private final static int B = 10007;
    private AVL<T> associativeArray[];

    public HashTable(int B) {
        this.B = B;
        associativeArray = new AVL<T>[B];

        for (int i=0; i<associativeArray.length; i++)
            associativeArray[i] = new AVL<T>();
    }
}
```

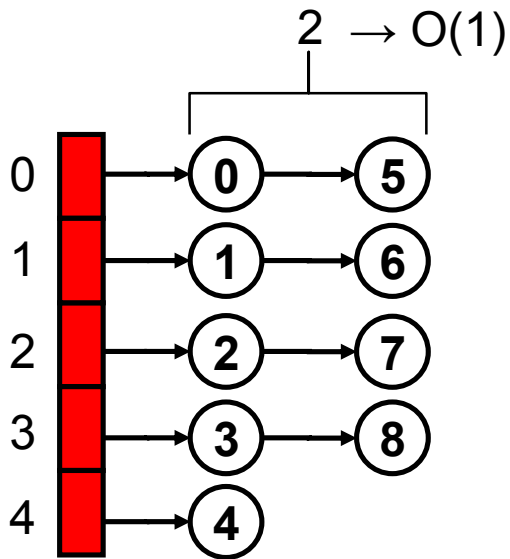
Tablas Hash Abiertas

add

$O(n/B) \rightarrow O(1)$

```
public void add (T a){  
    if (!find(a))  
        associativeArray[f(a.hashCode())].add(a);  
}
```

find() y remove() son análogas

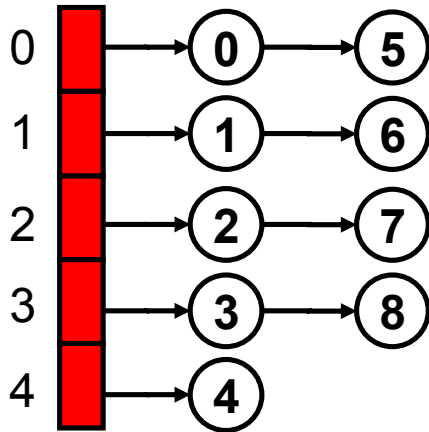


```
for (int i=0; i< 9; i++)  
    table.add(new Integer(i), i);
```

Tablas Hash Abiertas

Factor de Carga (load factor)

- ❖ Número de elementos de la tabla dividido entre el tamaño de la tabla.
 - $LF = n/B$.
 - Coincide con la longitud media de cada lista.



$$LF = 9/5 = 1,8$$

Tablas Hash Abiertas

LF Eficiente

Búsqueda	Promedio de enlaces visitados
Infructuosa	LF
Exitosa	$1 + LF/2$

- ❖ Para garantizar un alto rendimiento en tablas hash abiertas, el LF **ha de ser menor o igual que uno ($LF \leq 1$)**
 - $B = n$ (aproximadamente).
 - Longitud media de las listas = 1.

Tablas Hash Cerradas

Tablas Hash Cerradas

- ❖ Cada celda tiene capacidad para un único objeto.
 - Cuando se detecta una colisión (celda previamente ocupada), se busca el elemento en las celdas próximas.
 - Existen diversos enfoques para realizar la exploración:
 - Exploración Lineal.
 - Exploración Cuadrática.
 - Dispersión Doble.

HashTable class

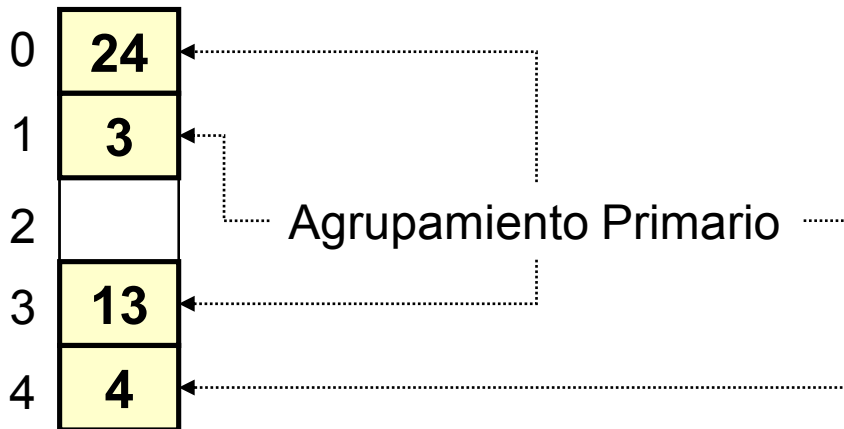
```
public class HashTable<T>
{
    private final static int B = 10007;
    private ArrayList<HashNode<T>> associativeArray;
}
```

Tablas Hash Cerradas

Exploración Lineal

❖ Búsqueda consecutiva en celdas próximas modificando la función f .

- $f(x) = [x + i] \% B$.
 - Donde i representa el número de intentos y asume valores de 0, 1, 2, 3...



$$\text{add}(4) \rightarrow [4 + 0] \% 5 = 4$$

$$\text{add}(13) \rightarrow [13 + 0] \% 5 = 3$$

$$\text{add}(24) \rightarrow [24 + 0] \% 5 = 4$$

$$\text{add}(24) \rightarrow [24 + 1] \% 5 = 0$$

$$\text{add}(3) \rightarrow [3 + 0] \% 5 = 3$$

$$\text{add}(3) \rightarrow [3 + 1] \% 5 = 4$$

$$\text{add}(3) \rightarrow [3 + 2] \% 5 = 0$$

$$\text{add}(3) \rightarrow [3 + 3] \% 5 = 1$$

Tablas Hash Cerradas

Agrupamientos

- ❖ Bloques de celdas ocupadas interrelacionadas.
 - Incluso en tablas relativamente vacías se pueden presentar agrupamientos.
 - Cualquier clave que se disperse sobre un agrupamiento **requerirá varios intentos para su encontrar su ubicación.**
 - Y lo que es peor... si se inserta **se unirá al agrupamiento.**
- ❖ Si la tabla es suficientemente grande, se podrá encontrar una ubicación para el elemento...
 - ...Pero la búsqueda puede llevar mucho tiempo.

Búsqueda	Número aproximado de intentos
Infructuosa	$(1 + 1/(1 - LF)^2)/2$
Exitosa	$(1 + 1/(1 - LF))/2$

Tablas Hash Cerradas

Estudios teóricos de velocidad de acceso

LF	Intentos por Inserción (promedio)
0,90	50
0,75	8,5
0,50	2,5

- ❖ Se recomienda usar $LF \leq 0,5$
 - B debería ser al menos el doble de n.
 - El incremento en el consumo de memoria es notable.

En las tablas hash abiertas la recomendación es $LF \leq 1$

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

- ❖ La existencia de agrupamientos impide el borrado directo de un elemento.
 - El elemento se **marca para borrar** pero no se elimina definitivamente **hasta que su espacio no sea requerido** por una operación de inserción.
 - Los elementos marcados se consideran vacíos durante las inserciones y ocupados durante las búsquedas.

0	24
1	3
2	
3	13
4	4

$\text{delete}(24) \rightarrow [24 + 0] \% 5 = 4$

$\text{delete}(24) \rightarrow [24 + 1] \% 5 = 0$

$\text{find}(3) \rightarrow [3 + 0] \% 5 = 3$

$\text{find}(3) \rightarrow [3 + 1] \% 5 = 4$

$\text{find}(3) \rightarrow [3 + 2] \% 5 = 0$

El acceso a la clave 3 se ha perdido

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

HashTable class

```
public class HashNode <T>
{
    public final static byte EMPTY    = 0;
    public final static byte VALID    = 1;
    public final static byte DELETED = 2;

    private T element;
    private byte status = EMPTY;
}
```

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

HashTable class

```
public class HashTable<T>
{
    private final static int B = 10007;
    private ArrayList<HashNode<T>> associativeArray;
}
```

Antes

0	24	VALID
1	3	VALID
2		EMPTY
3	13	VALID
4	4	VALID

$\text{delete}(24) \rightarrow [24 + 0] \% 5 = 4$

$\text{delete}(24) \rightarrow [24 + 1] \% 5 = 0$

$\text{find}(3) \rightarrow [3 + 0] \% 5 = 3$

$\text{find}(3) \rightarrow [3 + 1] \% 5 = 4$

$\text{find}(3) \rightarrow [3 + 2] \% 5 = 0$

$\text{find}(3) \rightarrow [3 + 3] \% 5 = 1$

$\text{add}(15) \rightarrow [15 + 0] \% 5 = 0$

Después

0	15	VALID
1	3	VALID
2		EMPTY
3	13	VALID
4	4	VALID

Tablas Hash Cerradas

Exploración Cuadrática

- ❖ Si se produce una colisión se exploran las celdas a una distancia cuadrática de la anteriormente consultada.
 - $f(x) = [x + i^2] \% B$.
 - Donde i representa el número de intentos y asume valores de 0, 1, 2, 3...

0	24
1	
2	3
3	13
4	4

$$\text{add}(4) \rightarrow [4 + 0^2] \% 5 = 4$$

$$\text{add}(13) \rightarrow [13 + 0^2] \% 5 = 3$$

$$\text{add}(24) \rightarrow [24 + 0^2] \% 5 = 4$$

$$\text{add}(24) \rightarrow [24 + 1^2] \% 5 = 0$$

$$\text{add}(3) \rightarrow [3 + 0^2] \% 5 = 3$$

$$\text{add}(3) \rightarrow [3 + 1^2] \% 5 = 4$$

$$\text{add}(3) \rightarrow [3 + 2^2] \% 5 = 2$$

Tablas Hash Cerradas

Exploración Cuadrática

- ❖ Puesto que la longitud de los saltos es mayor (longitud cuadrática) es posible no encontrar una posición libre.
 - ¡Aún cuando puedan existir posiciones libres, la exploración cuadrática puede saltar por encima de ellas... ignorándolas!

Teorema de la Exploración Cuadrática

Si utilizando exploración cuadrática **se cumple que** B es primo y el $LF \leq 0,5$ **siempre es posible** encontrar una posición para insertar un elemento.

- ❖ La exploración cuadrática elimina el agrupamiento primario...
 - ... aún cuando puede crear agrupamientos secundarios.
- ❖ El agrupamiento secundario podría llegar a ser asumible...
 - Estudios de simulación demuestran que ante agrupamientos secundarios **tan solo es necesario un salto** para encontrar posiciones libres.

Tablas Hash Cerradas

Dispersión Doble

❖ Utilizada una doble función de dispersión.

- $f(x) = [x + i * H_2(x)] \% B$.
 - Donde i representa el número de intentos y asume valores de 0, 1, 2, 3...
 - Donde H_2 es la función de cálculo de salto. Puede ser cualquiera. Se recomienda:
 - » $H_2(x) = R - X \% R$.
 - » Donde R es el número primo antecesor de B .

0	
1	3
2	24
3	13
4	4

$$\text{add}(4) \rightarrow [4 + 0 * (3 - 4 \% 3)] \% 5 = 4$$

$$\text{add}(13) \rightarrow [13 + 0 * (3 - 13 \% 3)] \% 5 = 3$$

$$\text{add}(24) \rightarrow [24 + 0 * (3 - 24 \% 3)] \% 5 = 4$$

$$\text{add}(24) \rightarrow [24 + 1 * (3 - 24 \% 3)] \% 5 = 2$$

$$\text{add}(3) \rightarrow [3 + 0 * (3 - 3 \% 3)] \% 5 = 3$$

$$\text{add}(3) \rightarrow [3 + 1 * (3 - 3 \% 3)] \% 5 = 1$$

Solución: posiciones 4, 1, 3 y 0

Tablas Hash Cerradas

Evaluación de la Dispersión Doble

❖ Ventajas

- Elimina el agrupamiento.
- El número esperado de intentos es bajo.

❖ Desventajas

- El uso de una segunda función aumenta el cálculo del coste de ejecución.

Redispersión

Duplicar el tamaño de la tabla dinámicamente

- ❖ Si el LF aumenta demasiado...
 - El rendimiento de la tabla decrece considerablemente.
 - $LF > 1$ en tablas hash abiertas.
 - Se paraliza el funcionamiento de tablas cerradas al no encontrar posiciones libres.
 - $LF > 0,5$ en tablas hash cerradas.

- ❖ La redistribución recupera un LF aceptable **traslizando** los elementos a una tabla de mayor tamaño.
 - Se establece un número B para la nueva tabla buscando el **número primo inmediatamente superior al doble** del original.
 - Recorre secuencialmente los elementos de la tabla original añadiéndolos a la nueva tabla.

Redispersión

Ejercicio

❖ Redispersar utilizando Exploración Cuadrática

El número primo inmediatamente superior al doble de 5 es el 11

0	24
1	
2	3
3	13
4	4

$$\text{add}(24) \rightarrow [24 + 0^2] \% 11 = 2$$

$$\text{add}(3) \rightarrow [3 + 0^2] \% 11 = 3$$

$$\text{add}(13) \rightarrow [13 + 0^2] \% 11 = 2$$

$$\text{add}(13) \rightarrow [13 + 1^2] \% 11 = 3$$

$$\text{add}(13) \rightarrow [13 + 2^2] \% 11 = 6$$

$$\text{add}(4) \rightarrow [4 + 0^2] \% 11 = 4$$

$O(n)$

0	
1	
2	24
3	3
4	4
5	
6	13
7	
8	
9	
10	

Redispersión

Activación de la Redispersión

- ❖ La redistribución se puede lanzar automáticamente cuando...
 - a) Se alcance un $LF > 0,5$.
 - b) FALLE una inserción (no hay posiciones libres).
 - c) Cuando se supere un cierto umbral de LF definido en el constructor de la tabla hash.

- ❖ Redispersión Inversa
 - Reduce el tamaño de la tabla para ahorrar memoria cuando se han realizado muchas operaciones de borrado.

Tipo de tabla	Umbral de LF para redistribución inversa
Abierta	0,33
Cerrada	0,16

HOMEWORK

PLAYGROUND

- ❖ Consulte la entrada para la **Tabla Hash** en la Wikipedia
 - Estudie **cuidadosamente** todo el contenido de la entrada.
 - Ponga especial atención a las situaciones en las que resulta interesante utilizar árboles en lugar de listas en las tablas hash abiertas.
 - Analice en detalle las funciones de dispersión *Hash de División* y *Hash de Multiplicación*.

Los conocimientos adquiridos en esta tarea **serán evaluados en el examen**

Apéndices

Referencias

Referencias

- HERNÁNDEZ, Roberto; LÁZARO, Juan Carlos; DORMIDO, Raquel, ROS, Salvador; (2001) *Estructuras de Datos y Algoritmos*. Prentice Hall. ISBN 84-205-2980-X [Cap. 5 y 6].
- JOYANES AGUILAR, Luis; ZAHONERO MARTÍNEZ, Ignacio; (1998) *Estructura de Datos: Algoritmos, Abstracción y Objetos*. Mc Graw Hill. ISBN: 84-481-2042-6 [Cap. 10, 11 y 12].
- ORTEGA F., Maruja; (1988) *Grafos y Algoritmos*. Universidad Metropolitana, Oficina Metrópolis.
- WEISS, Mark Allen; (2000) *Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN84-7829-035-4.
- WEISS, Mark Allen; (1995) *Estructuras de Datos y Algoritmos* Addison-Wesley Iberoamericana. ISBN 0-201-62571-7.