ESTRUCTURAS DE DATOS

Fundamentos y Aplicaciones

MARTÍN GONZÁLEZ-RODRÍGUEZ

www.martin-gonzalez.es

Estructuras de Datos

Dr. Martin Gonzalez-Rodriguez

ISBN: 978-1-4467-9145-5

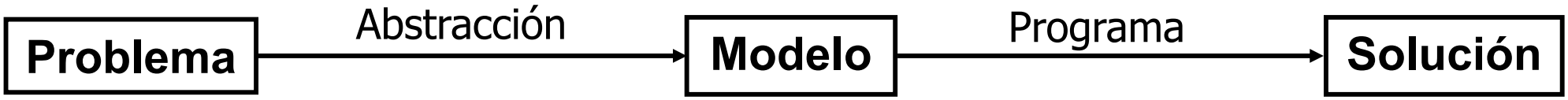
Diseño y Algoritmia

Dr. Martin Gonzalez-Rodriguez

Resolución de Problemas en Ingeniería

Estrategia

- Conocer y acotar el problema (análisis).
- Encontrar un modelo que represente el problema (abstracción).
- Formular el algoritmo sobre el modelo.



Programas

La Frase
`Programas = Estructuras de Datos + Algoritmos`

- Identificar medios para **almacenar datos** y diseñar **algoritmos que resuelvan la tarea** asignada a los procesos.

Niclaus Wirth (Wikipedia)

- Acuñada por Niclaus Wirth en 1976
  - Premio Turing 1984.
    - Diseñador de los lenguajes de programación Euler, Algol, Pascal, Modula, Modula-2 y Oberon.

Tipo de Dato

Definición

- Conjunto de valores que puede asumir una propiedad de una clase.
  - **TDP** (Tipos de Datos Predefinidos) son los Tipos de Datos **por defecto** de un lenguaje de programación.
    - Número Entero.
    - Número Real.
    - Carácter.
    - Booleano.
    - Referencia.

Estructura de Datos

Definición

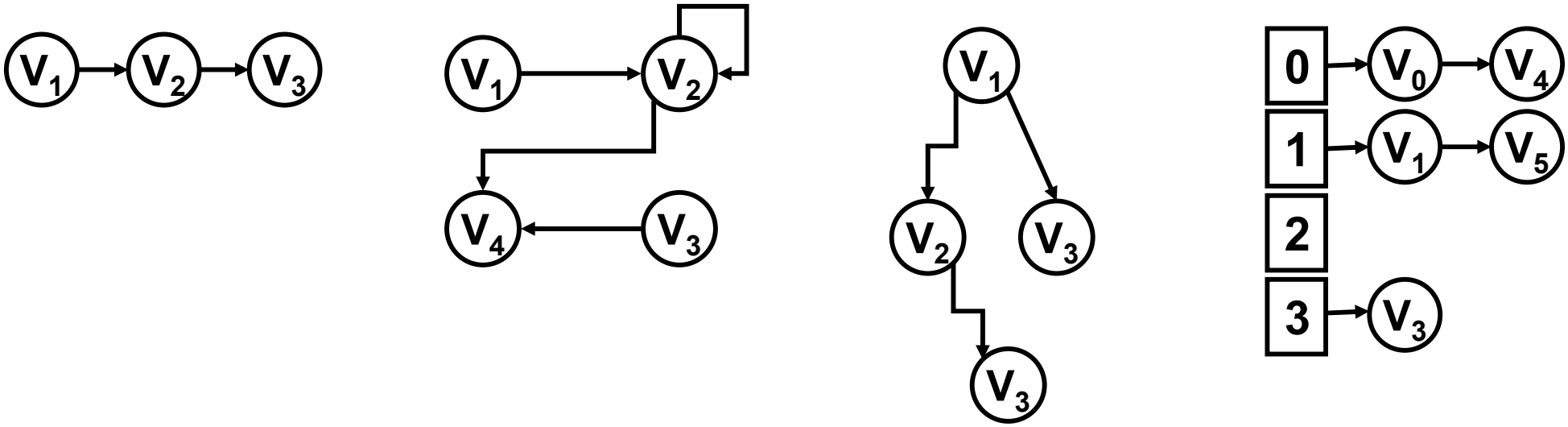
- Conjunto de datos relacionados de una forma determinada[1].
  - Los **TDE** (Tipos de Datos Estructurados) de un lenguaje de programación son colecciones de Tipos de datos almacenados de forma secuencial.
    - Arrays.
    - Cadenas de caracteres.
    - Clases y objetos.
  - Existen otras estructuras de datos básicas *por defecto* implementadas por medio de clases.
    - ArrayList.
    - List.
    - HashMap.
    - Stack.
    - …

[1] *Weiss, Mark Allen; (2000) Estructuras de Datos En Java 2. Addison-Wesley Iberoamericana.*

Estructura de Datos

Clasificación

- Principales familias de estructuras de datos
  - Lineales (listas, pilas y colas).
  - En red (grafos).
  - Jerárquicas (árboles).
  - Diccionario (tablas hash).



- Se pueden realizar combinaciones infinitas de estructuras.

Estructura de Datos

¿Qué estructura elegir?

- La selección de la estructura adecuada para un problema determinado depende de…
  1. Adecuación de la estructura a la representación del modelo.
  2. Eficiencia de la estructura.
     - Temporal (velocidad de los algoritmos asociados) → $O_T(n)$.
     - Espacial (ocupación en memoria de la estructura) → $O_M(n)$.

Algoritmia (muy) Básica

¿Cuántas veces se ejecuta *test()*?

Algoritmo A — $T_A = 3$

```
{
 test();
 test();

 int i=3;
 return (i*test());
}
```

Algoritmo B — $T_B = 2$

```
{
 test();
 test();
 if (5%2 == 0) {
  test();
  return (test()%2);
 }
 return (0);
}
```

Algoritmia (muy) Básica

¿Cuántas veces se ejecuta *test()*?

Algoritmo C $T_C(n) = 4n + 6$

```
{
 test();
 test();
 test();

 for (int i=0; i<n; i++) {
  test();
  test();
  test();
  test();
 }

 test();
 test();
 test();
}
```

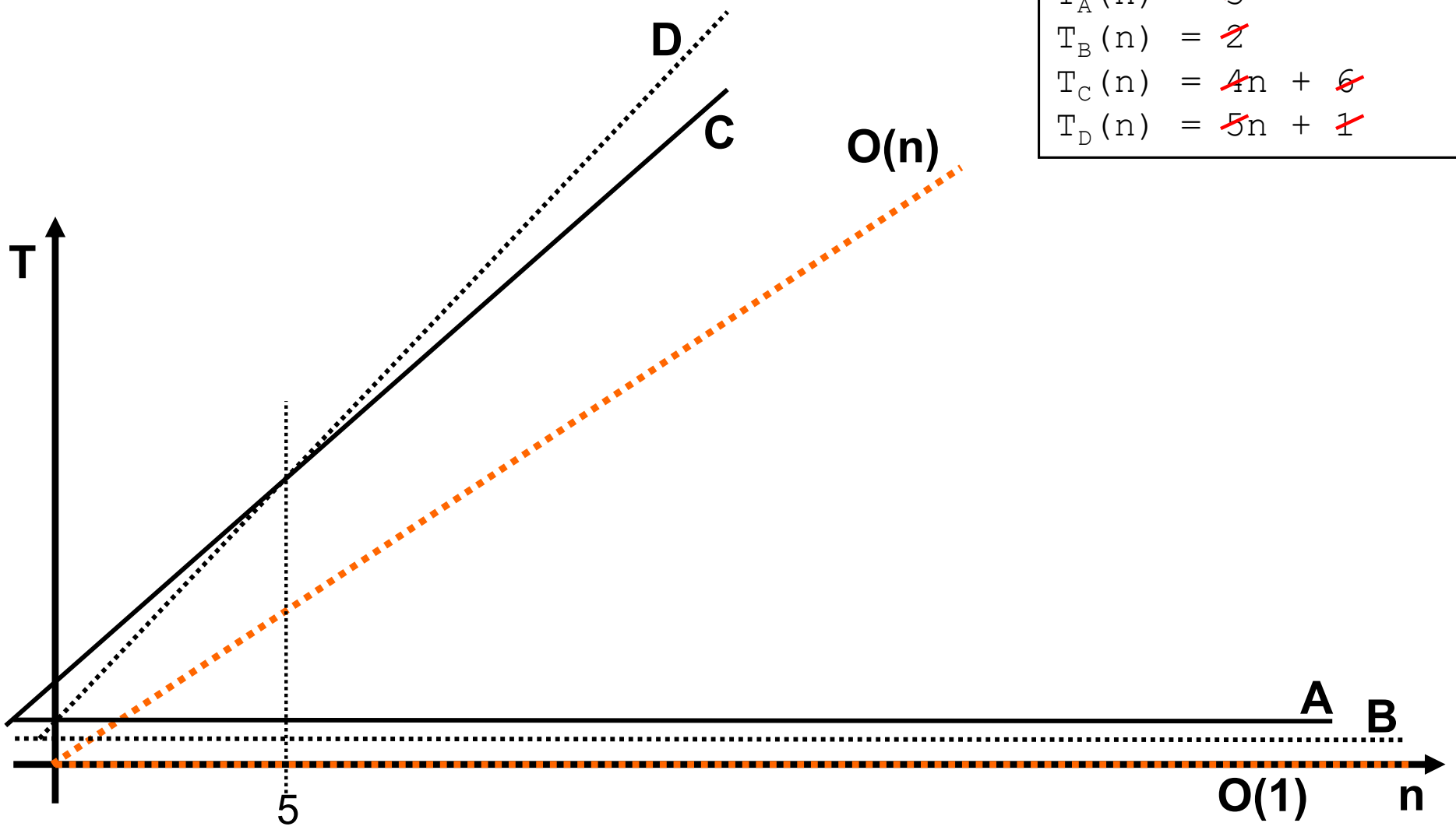
Algoritmo D $T_D(n) = 5n + 1$

```
{
 for (int i=0; i<n; i++) {
  test();
  test();
  test();
  test();
  test();
 }

 test();
}
```

Algoritmia (muy) Básica

¿Qué algoritmo es más rápido?

Tiempos de Ejecución
$T_A(n) = \cancel{3}$
$T_B(n) = \cancel{2}$
$T_C(n) = \cancel{4}n + \cancel{6}$
$T_D(n) = \cancel{5}n + \cancel{1}$



Algoritmia (muy) Básica

¿Cuántas veces se ejecuta *test()*?

Algoritmo E $T_E(n) = 2n^2 + 1$

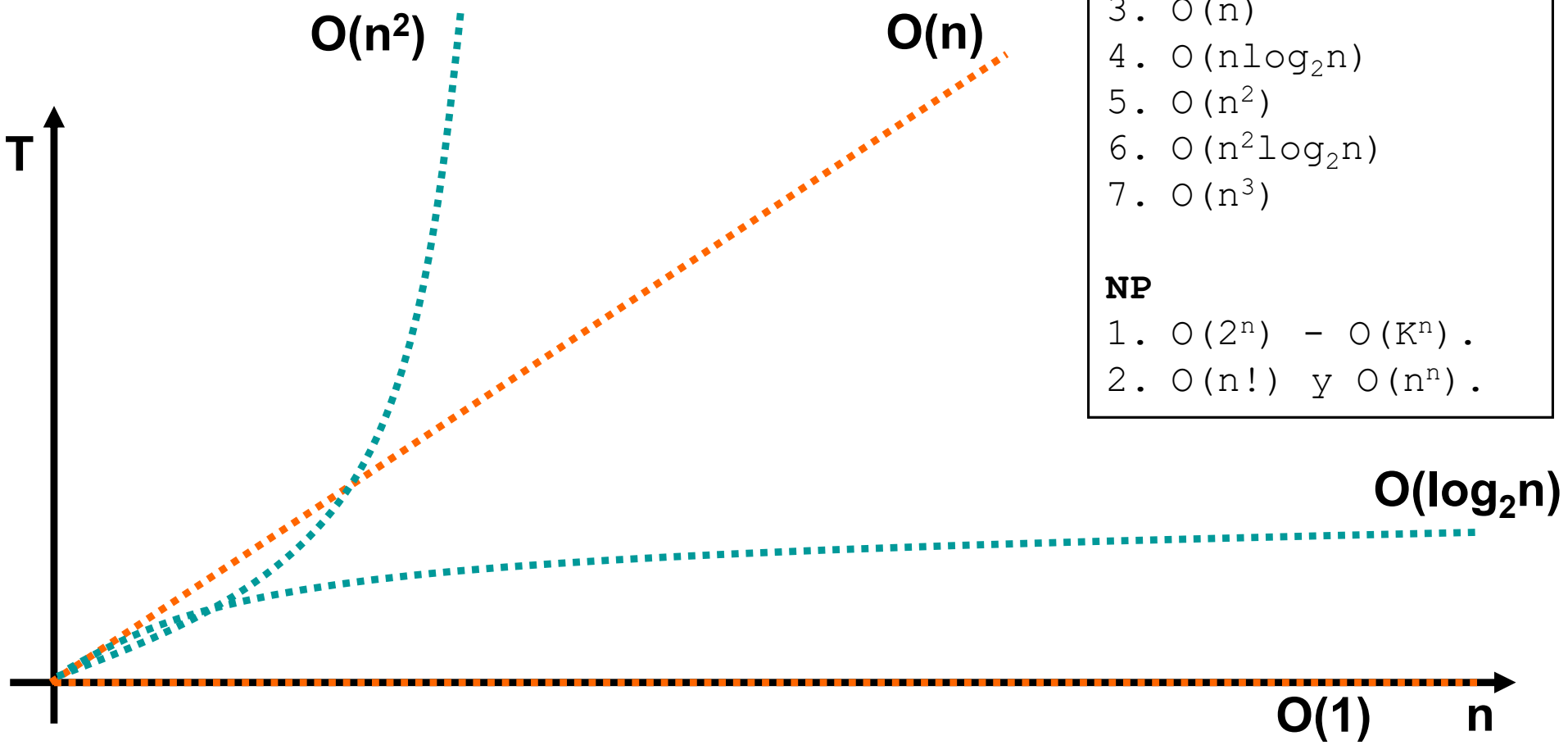
```
{
 for (int i=0; i<n; i++)
  for (int j=0; j<n; j++) {
   test();
   test();
 }
 test();
}
```

Algoritmo F $T_F(n) = 2([\log_2 n] + 1) + 1$

```
{
 while (n>0) {
  test();
  test();
  n = n/2;
 }

 test();
}
```

Algoritmia (muy) Básica

Complejidad Temporal



Ranking de Eficiencia

P

1. $O(1)$
2. $O(\log_2 n)$
3. $O(n)$
4. $O(n\log_2 n)$
5. $O(n^2)$
6. $O(n^2\log_2 n)$
7. $O(n^3)$

NP

1. $O(2^n)$ - $O(K^n)$.
2. $O(n!)$ y $O(n^n)$.

Algoritmia (muy) Básica

Importancia de la Eficiencia Temporal

N	$T_A(n) = 2^n$	$T_B(n) = n^3$
10	0,1 segundos	10 segundos
15	3,27 segundos	33,7 segundos
20	1,75 minutos	1,3 minutos
25	0,93 horas	2,5 minutos
30	29,8 horas	4,5 minutos
35	39,7 días	7,14 minutos
40	3,4 años	10,66 minutos
45	1,08 siglos	15,18 minutos

C58 Series

Estructuras en Red

Dr. Martin Gonzalez-Rodriguez

Estructuras de Datos en Red

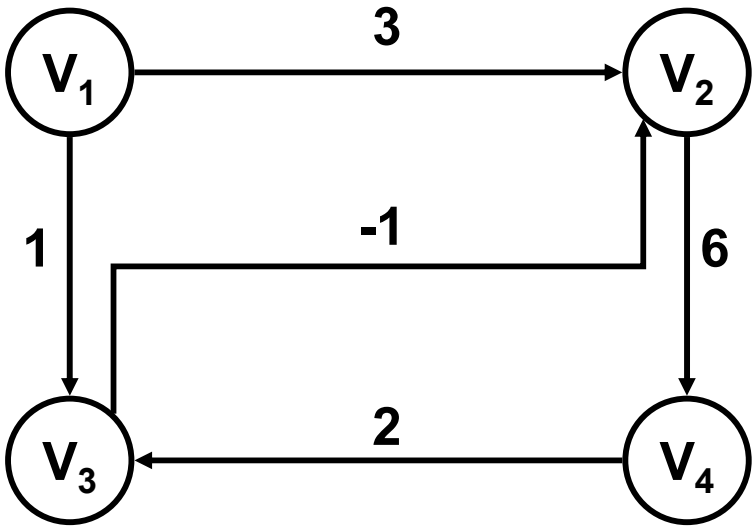
Objetivo

- Modelar relaciones conceptuales complejas entre objetos.
  - Redes de transporte (carreteras, ferrocarril, metro, electricidad, gas, petróleo, etc.).
  - Redes de comunicaciones (Internet, telefonía, correos, etc.)
  - Redes Sociales (Facebook, Instagram, préstamo, deuda, etc.).
  - Estructuras (moleculares, neuronales, genéticas, etc.).

Definición

¿Qué es un Grafo?

- Un grafo es un **modelo matemático** que permite representar *relaciones arbitrarias* entre objetos.



Definición

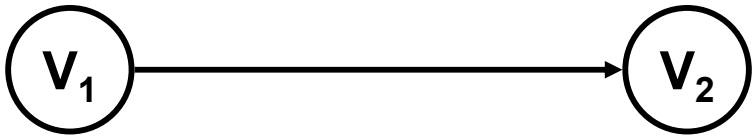
Definición Formal

- Un Grafo es un par (V, E) denotado por G(V, E) donde:
  - **V** es un conjunto finito de **Vértices** (también llamados **Nodos)**.

$$V = \{V_1, V_2, ...\}$$

  - **E** es una familia de **pares de elementos** (v, w) pertenecientes a **V** llamados Aristas (*Edges*).
    - Representan relaciones entre el vértice v y el vértice w.

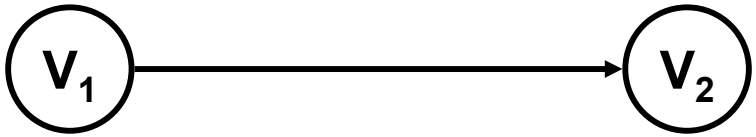
$$E = \{(V_1, V_2),...\}$$



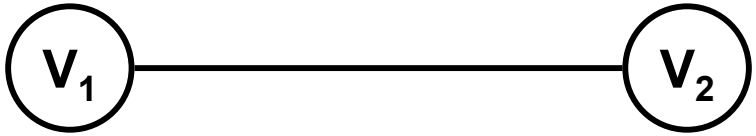
Tipología

Tipos de Grafos

- Si los pares {v,w} son ordenados
  - Éstos se conocen como **Arcos** y se dice que el grafo es ***dirigido*** (AKA *Grafo Orientado* o *Digrafo*).



- Si los pares {v, w} **no son** ordenados
  - Éstos se conocen como **Aristas** y se dice que el grafo es ***no dirigido***.



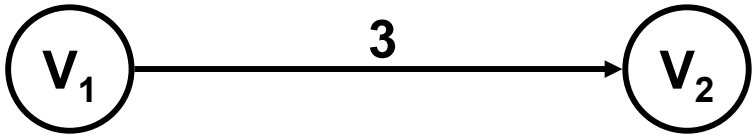
Tipología

Tipos de Grafos

- Un **Grafo Etiquetado** es un trío (V, E, W) denotado por G(V, E, W) donde
  - W es un **conjunto finito** de etiquetas en el que **cada arco u arista** dispone de su propia etiqueta.

$$W = \{W_1, W_2, ...\}$$

  - Las etiquetas pueden ser:
    - **Números**. Las etiquetas de llaman **pesos** y pueden representar costes o beneficios.



    - **Caracteres** o cadenas de caracteres.

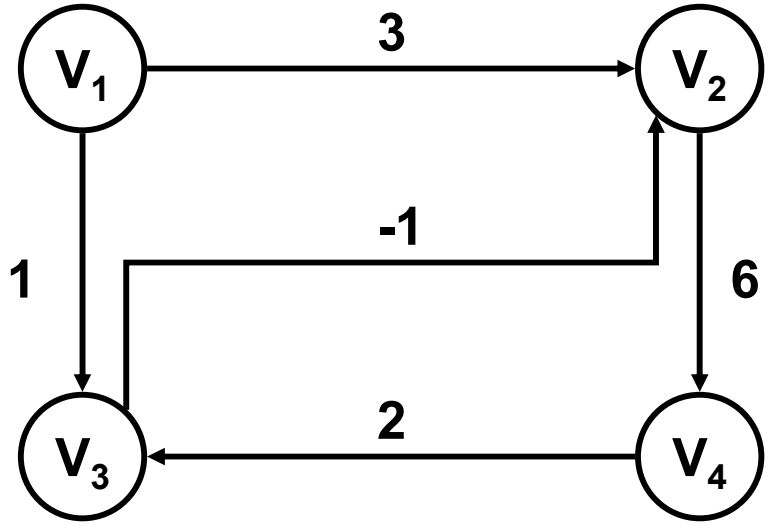
Putting it all Together

Definición Formal Completa

$$V = \{V_1, V_2, V_3, V_4\}$$

$$E = \{(V_1, V_2), (V_1, V_3), (V_2, V_4), (V_3, V_2), (V_4, V_3)\}$$

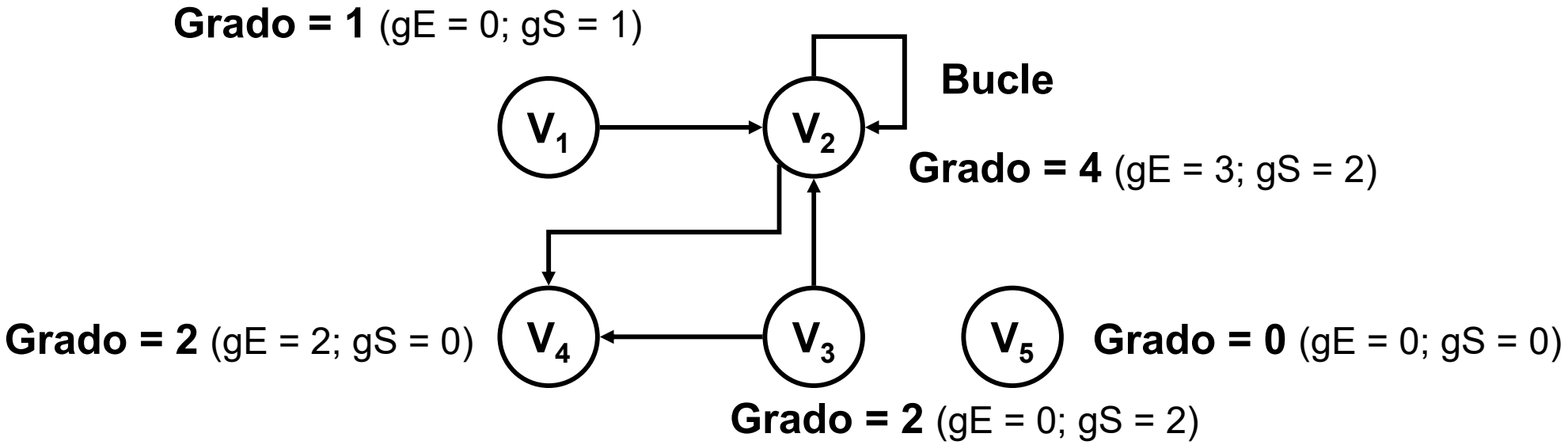
$$W = \{\ 3,\ 1,\ 6,\ -1,\ 2\}$$



Conceptos Básicos

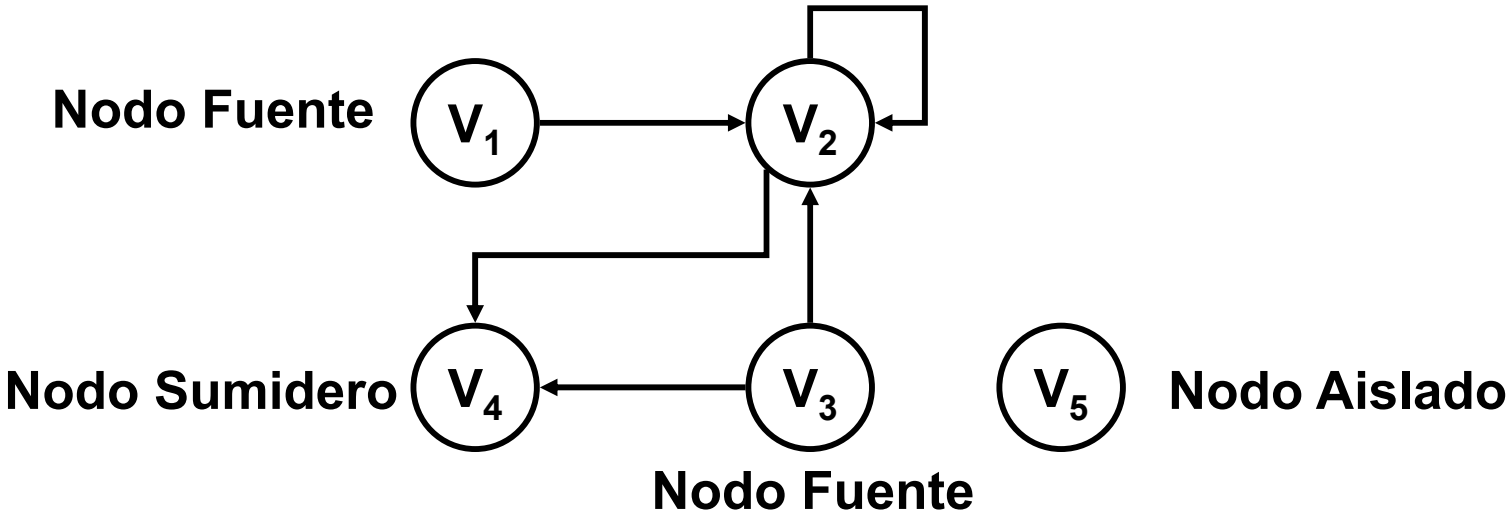
- **Bucle**
  - Arco u arista con igual origen que destino.

- **Grado** de un nodo
  - Número de arcos u aristas conectados al nodo.
    - **Grado de Entrada (gE)** de un nodo:
      - » Número de arcos o aristas que tienen al nodo como destino.
    - **Grado de Salida (gS)** de un nodo:
      - » Número de arcos o aristas que tienen al nodo como origen.



Conceptos Básicos

- Nodo Fuente
  - Si cumple que **GradoSalida > 0** y **GradoEntrada = 0**.
- Nodo Sumidero
  - Si cumple que **GradoSalida = 0** y **GradoEntrada > 0**.
- Nodo Aislado
  - Si cumple que **GradoSalida = 0** y **GradoEntrada = 0**.



Capacidad de un Grafo

n = número de nodos de un grafo

- n = Cardinalidad del conjunto V.

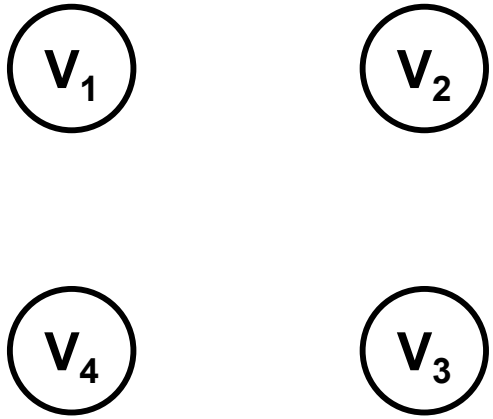
$$V = \{V_1, V_2, \ldots, V_{n-1}, V_n\}$$

- El valor de **n** se utiliza como parámetro para medir la eficiencia de las operaciones sobre grafos.

Capacidad de un Grafo

Cálculo del número de arcos en base a n

- $A_{min}(n)$: Número **mínimo** de arcos

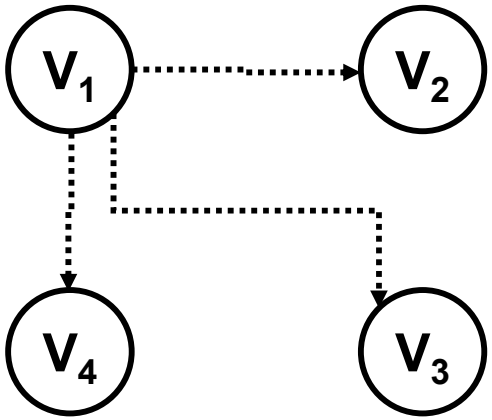


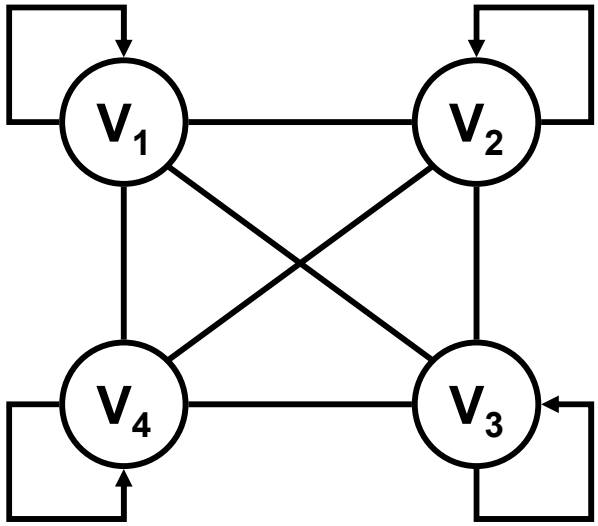
$A_{min}(n) = 0$

Capacidad de un Grafo

Cálculo del número de arcos en base a n

- $A_{max}(n)$: Número **máximo** de arcos (**Grafo Completo**)





$A_{max}(n) = n(n - 1) = n^2 - n$ (sin bucles)

$A_{max}(n) = n^2 - n + n = n^2$ (con bucles)

Representación en Memoria

Densidad de un grafo

- **Grafos densos**: $A(n) \rightarrow n^2$.
  - Número de arcos similar a la de un grafo completo.
  - Eficiencia máxima sobre memoria estática (matrices, arrays).

- **Grafos ligeros**: $A(n) \rightarrow n$.
  - Promedio de un arco por nodo.
  - Eficiencia máxima sobre memoria dinámica (listas) al requerir de pocos enlaces.

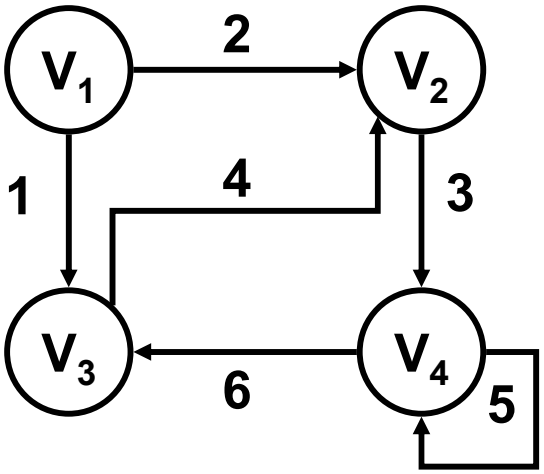
Graph Class – Matrix

Adjacency Matrix

```
ArrayList<GraphNode<T>> nodes;;
private boolean[][] edges;
private double[][] weight;
int size; // número de nodos en la estructura (nodes.size)
```

- **nodes**: Almacena los objetos de las clases que representan a cada nodo.
- El elemento **edges[i,j]** será *true* **si y solo si** existe un arco que tiene su origen en i y su destino en j.
- El elemento **weight[i, j]** almacena el peso (coste) del arco con origen en i y destino en j.
  - El peso *puede* ser cero (0,0).
  - Si ese arco no existe, el valor *será* cero (0,0).

Graph Class – Matrix



nodes

0	1	2	3	4
V_1	V_2	V_3	V_4	

size = 4

edges

	0	1	2	3	4
0	F	T	T	F	
1	F	F	F	T	
2	F	T	F	F	
3	F	F	T	T	
4					

size = 4

weight

	0	1	2	3	4
0	0	2	1	0	
1	0	0	0	3	
2	0	4	0	0	
3	0	0	6	5	
4					

size = 4

Análisis de Eficiencia

Rendimiento de las Matrices de Adyacencia

- Ventajas
  - Acceso instantáneo a la información de cualquier elemento de las matrices.
    - Acceso O(1).

- Desventajas
  - Dificultad para determinar el tamaño inicial de la matriz.
    - Debería ser lo más cercano posible a n.
  - Desaprovechamiento de memoria en grafos ligeros (matrices casi vacías).
    - Memoria consumida: $O_M(n^2)$.

- Escenario de uso
  - Grafos densos.

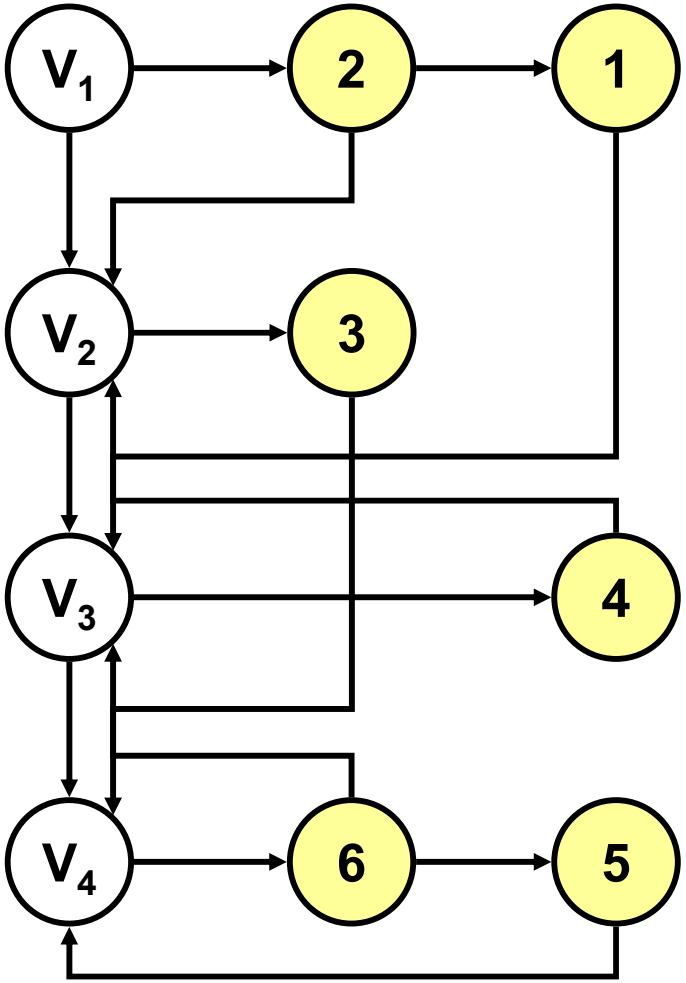
Graph Class – List

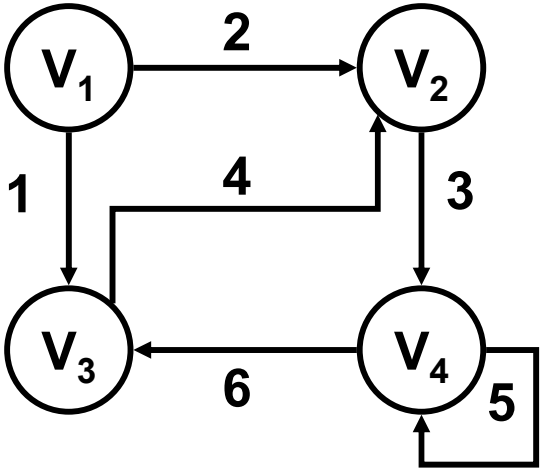
Adjacency List

```
class Edge{
        private double weight;
        private Node target;
}

class Node <T>{
        private T node;
        private LinkedList<Edge> edges;
}

private LinkedList<Node> nodes;
```

- Lista de listas
  - La lista principal (*nodes*) contiene la colección V de nodos.
  - Cada nodo de esta lista contiene a su vez una lista con información sobre sus nodos adyacentes (colección *edges*).

Graph Class – List





Análisis de Eficiencia

Rendimiento de las Listas de Adyacencia

- Ventajas
  - Memoria consumida en función del número de nodos y del número de aristas reales.
    - Memoria consumida: $O_M(K_1 n + K_2 a)$, donde K_1 = #bytes por nodo y K_2 = #bytes por arco.

- Desventajas
  - Es necesario realizar complejas búsquedas secuenciales en las listas.
    - Acceso O(n).
  - Si el grafo es denso se desaprovecha gran cantidad de memoria en las referencias necesarias para mantener las listas.
    - El grado máximo de desaprovechamiento de memoria se alcanza con el **grafo completo**.

- Escenario de Uso
  - Grafos ligeros.

Graph Class – Métodos Básicos

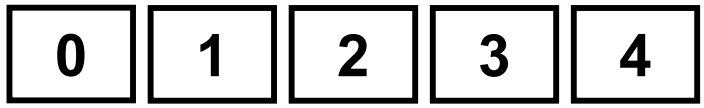
Adjacency Matrix

Método	Complejidad
graph (constructor)	O(1)
getNode	O(n)
addNode	O(n)
removeNode	O(n)
existEdge ?	O(n)
addEdge	O(n)
removeEdge	O(n)
print	$O(n^2)$

Graph Class – Métodos Básicos

graph (fragment)	O(1)
`size = 0;`	

nodes

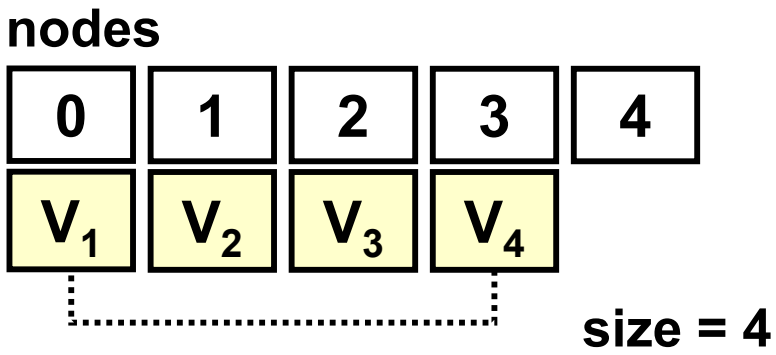


size = 0

Graph Class – Métodos Básicos

getNode (Pseudocode)	O(n)

```
public int getNode (T node)
{
 for (int i=0; i<size; i++)
  if (nodes[i].equals(node))
   return (i); // returns the node's position

 return (-1); // search fails, node does not exist
}
```



Graph Class – Métodos Básicos

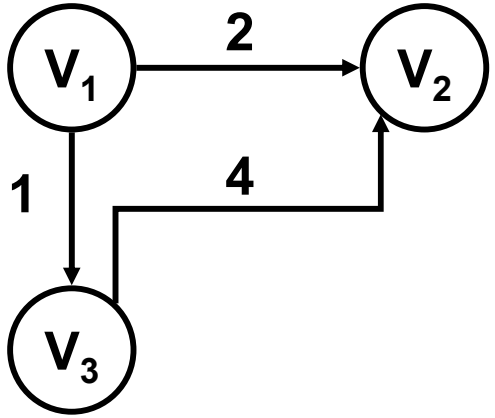
addNode (Pseudocode) **O(n)**

```
public void addNode (T node)
{
 // precondition: node does not exits and there is
 // available space for the node.

 if (getNode(node)== -1 && size<nodes.length)
 {
  nodes[size] = node;

  //inserts void edges
  for (int i=0; i<=size; i++)
  {
   edges[size][i]=false;
   edges[i][size]=false;
   weight[size][i]=0;
   weight[i][size]=0;
  }
  ++size;
 }
}
```

DEBATE: ¿Cuál es la complejidad temporal del algoritmo?

Graph Class – Métodos Básicos

Antes de insertar V_4



nodes

0	1	2	3	4
V_1	V_2	V_3		

size = 3

edges

	0	1	2	3	4
0	F	T	T		
1	F	F	F		
2	F	T	F		
3					
4					

size = 3

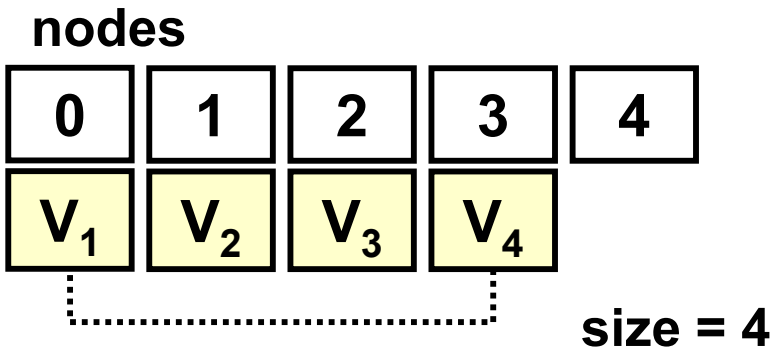
weight

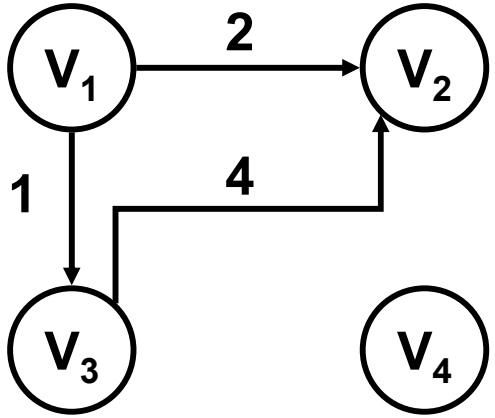
	0	1	2	3	4
0	0	2	1		
1	0	0	0		
2	0	4	0		
3					
4					

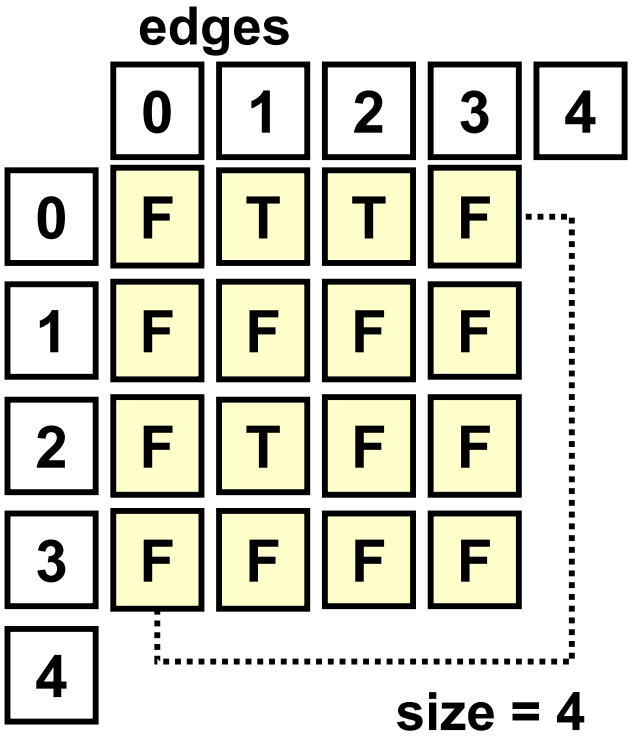
size = 3

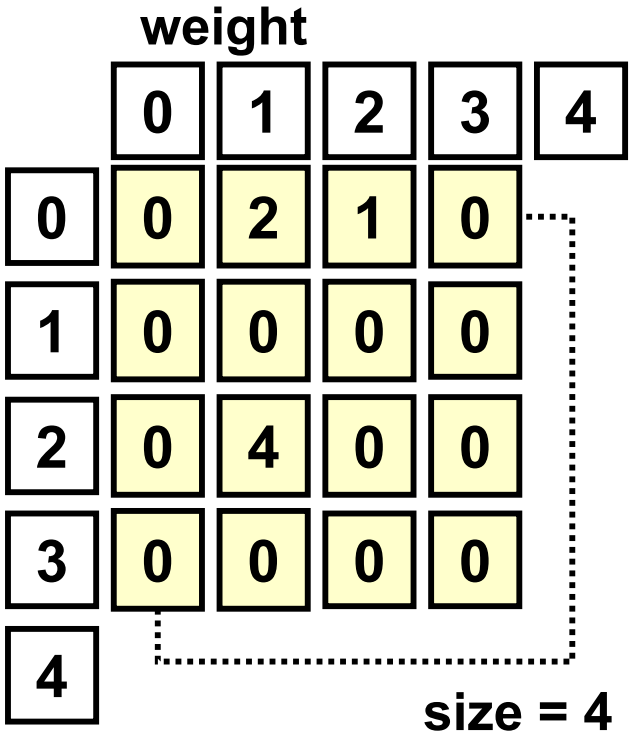
Graph Class – Métodos Básicos

Después de insertar V_4





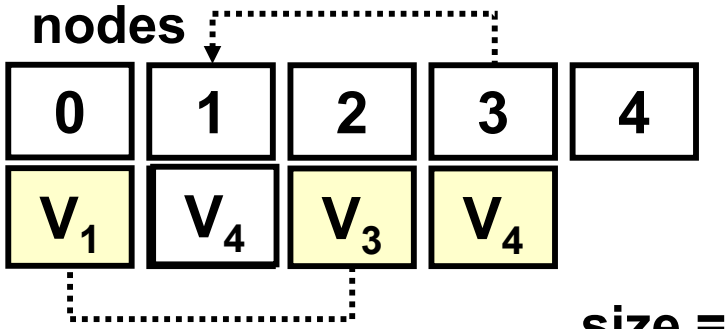




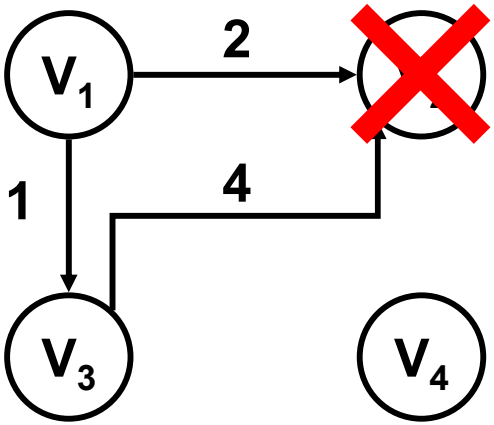
DEBATE: ¿Cómo borrar el nodo V_2?

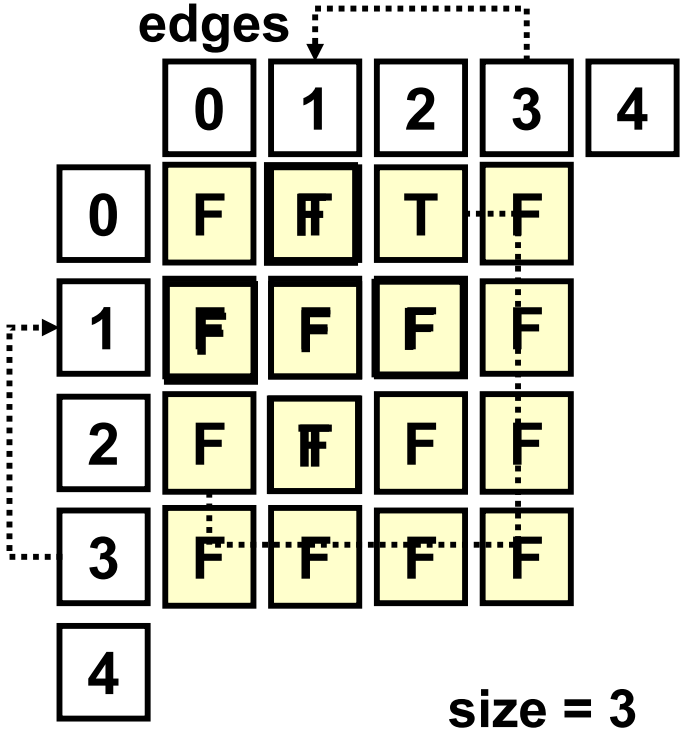
Graph Class – Métodos Básicos

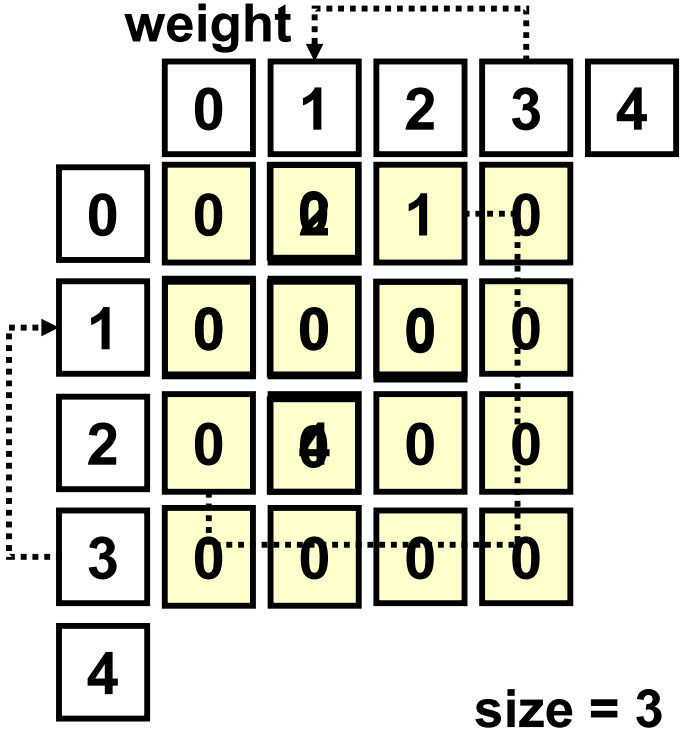
Después de borrar V_2



size = 3







Graph Class – Métodos Básicos

removeNode (Pseudocode) **O(n)**

```
public void removeNode (T node){
   int i = getNode(node);

   if (i>=0) {
    --size;
    if (i != size+1) { // it is not the last node
     nodes[i] = nodes[size]; //replaces by the last node

     //replace elements in the vectors edges and weights
     for (int j=0; j<=size; j++) {
      edges[j][i]=edges[j][size];
      edges[i][j]=edges[size][j];
      weight[i][j]=weight[size][j];
      weight[j][i]=weight[j][size];
      }
      // loop (diagonal)
      edges[i][i] = edges[size][size];
      weight[i][i] = weight[size][size];
    }
}
```

DEBATE: ¿Cómo implementar el método *existsEdge*?

Graph Class – Métodos Básicos

existsEdge (Pseudocode) **O(n)**

```
public boolean existsEdge (T origin, T destination)
{
 int i=getNode(origin);
 int j=getNode(destination);

 // precondition: both nodes must exist.
 // if don't… should we throw an exception?

 if (i>=0 && j>=0)
  return(edges[i][j]);
 else
  return (false);
}
```

Graph Class – Métodos Básicos

addEdge (Pseudocode) **O(n)**

```
public void addEdge (T origin, T destination, double
edgeWeight)
{

 // precondition: the edge must not already exist.
 if (!existEdge(origin, destination))
 {
  int i=getNode(origin);
  int j=getNode(destination);

  edges[i][j]=true;
  weight[i][j]=edgeWeight;
 }
 else
  ; // what about throwing an exception here?
}
```

Graph Class – Métodos Básicos

removeEdge (Pseudocode) **O(n)**

```
public void removeEdge (T origin, T destination){

 // precondition: the edge must exist.
 if (existsEdge(origin, destination)) {
  int i=getNode(origin);
  int j=getNode(destination);

  edges[i][j]=false;
  weight[i][j]=0.0;
 }
 else
  ; // what about throwing an exception?
}
```

Graph Class – Métodos Básicos

print (Pseudocode) **$O(n^2)$**

```
public void print(){

 for (int k=0; k<size; k++)
  nodes[k].print();

 for (int i=0; i<size; i++) {
  for (int j=0; j<size; j++) {
   System.out.print(edges[i][j] + "(");
   System.out.print(weight[i][j] + ") ");
  }
  System.out.println();
}
```

Graph Class – Métodos Avanzados

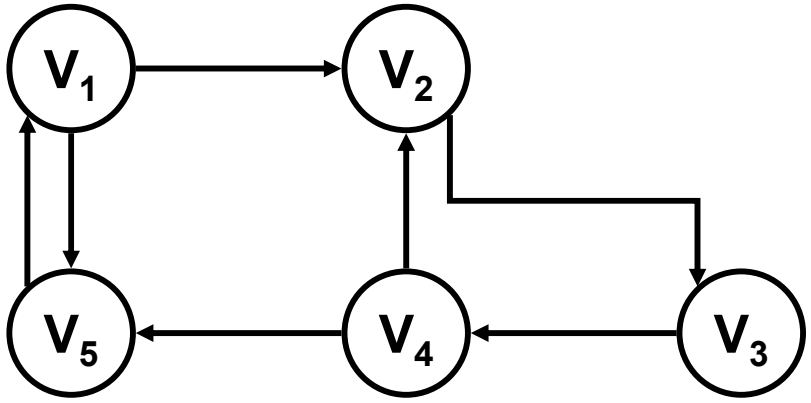
Adjacency Matrix

Método	Complejidad
Dijkstra	$O(n^2)$
Floyd	$O(n^3)$
Recorrido en Profundidad	$O(n^2)$
Prim / Warshall	$O(n^2)$

Más Conceptos Básicos

- **Camino** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Secuencia de nodos (con sus respectivas aristas) que permiten acceder al nodo V_j desde el nodo V_i.
    - Caminos entre V_1 y V_5
      - » $C_A = V_1, V_5$.
      - » $C_B = V_1, V_2, V_3, V_4, V_5$.
      - » $C_C = V_1, V_2, V_3, V_4, V_2, V_3, V_4, V_5$.
      - » …

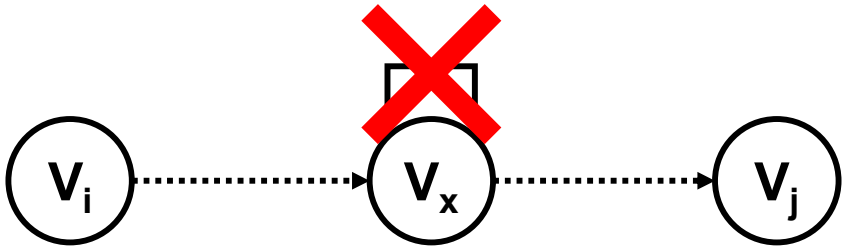
- **Longitud** de un camino entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Número de aristas empleadas para llegar al nodo V_j.
  - Equivale al número de nodos del camino **menos uno**.
    - Longitud de caminos entre V_1 y V_5
      - » $L(C_A) = 1$.
      - » $L(C_B) = 4$.
      - » $L(C_C) = 7$.



Más Conceptos Básicos

- **Camino Simple** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Es aquel camino en el que **no se repite** ningún nodo.

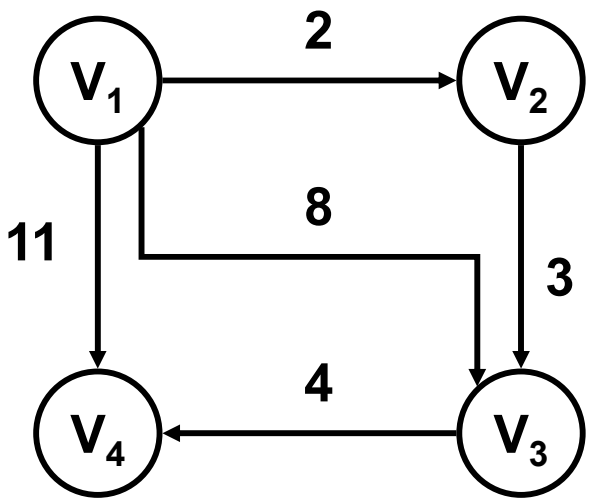
Teorema del Camino Simple
Si existe algún camino entre un par de nodos V_i (origen) y V_j (destino), entonces **existe al menos un camino simple** entre V_i y V_j.



- Es **posible** eliminar los ciclos de un camino para **convertirlo en un camino simple**.

Más Conceptos Básicos

- **Camino de Longitud Mínima** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Es aquel camino que implique pasar por el menor número de arcos.
    - El Camino de Longitud Mínima **es simple**.
    - Camino de Longitud Mínima entre V_1 y V_4
      - » $C_A = V_1, V_4$ (Longitud 1).



- **Camino de Coste Mínimo** entre dos nodos V_i, V_j ($V_i \neq V_j$)
  - Aquel que implica pasar por arcos cuya suma de pesos es mínima.
    - Camino de Coste mínimo entre V_1 y V_4
      - » $C_A = V_1, V_2, V_3, V_4$ (Coste 9).

Algoritmo de Dijkstra

Problema a Resolver

- ¿Cuál es el camino de coste mínimo para acceder a cada uno de los nodos de un grafo desde un nodo ***v*** dado?
  - ¿Cuál es la ruta más barata para llegar a Barcelona **desde Oviedo**?
  - ¿Cuál es la ruta más corta para llegar a Madrid **partiendo de Oviedo**?
    - ¿Y la ruta a Valencia? ¿Y el camino a Sevilla? ¿Y a Bilbao?... **desde Oviedo**.

Edger Dijkstra (Wikipedia)

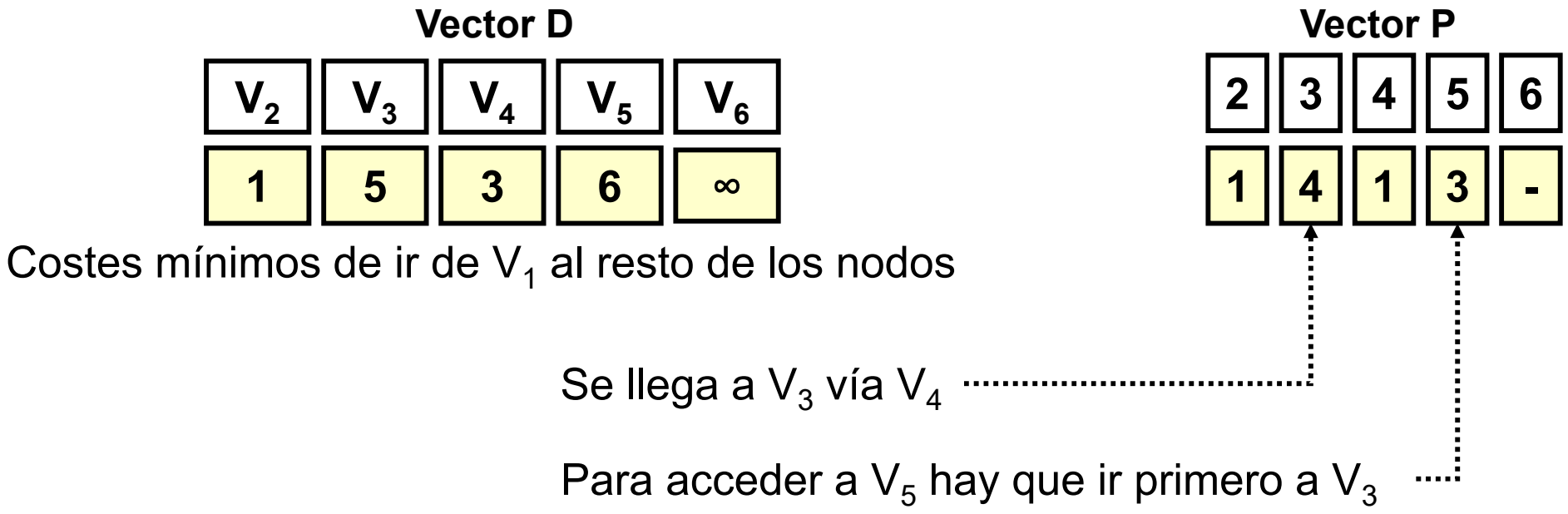
- Desarrollado por el holandés Edger Dijkstra en 1956
  - Premio Turing 1972.

Algoritmo de Dijkstra

Productos Obtenidos

- **Vector D** (unidimensional) o de Costes Mínimos
  - Guarda el coste mínimo desde ***v*** a cada uno de los nodos del grafo.

- **Vector P** (unidimensional) o de Rutas de Coste Mínimo
  - Almacena la ruta de coste mínimo desde ***v*** a cada uno de los nodos del grafo.



Algoritmo de Dijkstra

Ejemplo

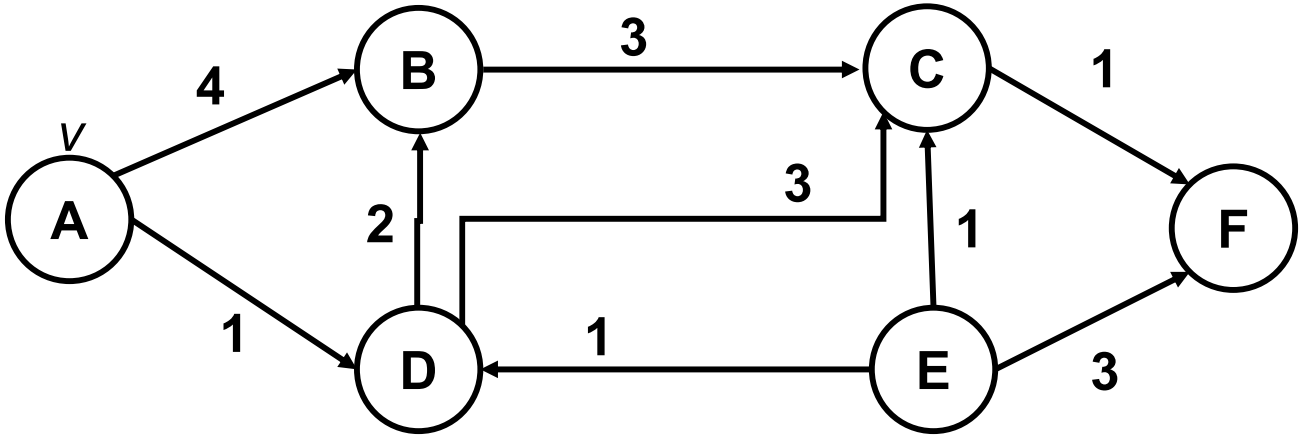
S = {A}

Vector D

V_2	V_3	V_4	V_5	V_6
1	5	3	6	∞

Vector P

2	3	4	5	6
1	4	1	3	-



Algoritmo de Dijkstra

Inicialización

- Iniciar **Conjunto S**
  - Elementos para los cuales ya se conoce el coste mínimo de ir desde ***v***.
  - Se inicializa con el propio ***v*** ya que al principio solo se conoce el coste mínimo de ir desde ***v*** a ***v*** (es decir, cero).
    - S = {v}.

- Iniciar **Vector D** de Coste Mínimo
  - Copia la fila correspondiente al elemento ***v*** de una matriz **weight** modificada…
    - **…sustituyendo** los valores de coste 0 por ∞.
    - El **coste** de moverse desde un nodo a otro **a través de un camino** (directo) **que no existe es infinito**.
    - En la primera iteración solo se conocen los costes de moverse de ***v*** a todos los demás nodos **a través de un camino directo** (longitud uno).

Algoritmo de Dijkstra

Ejemplo

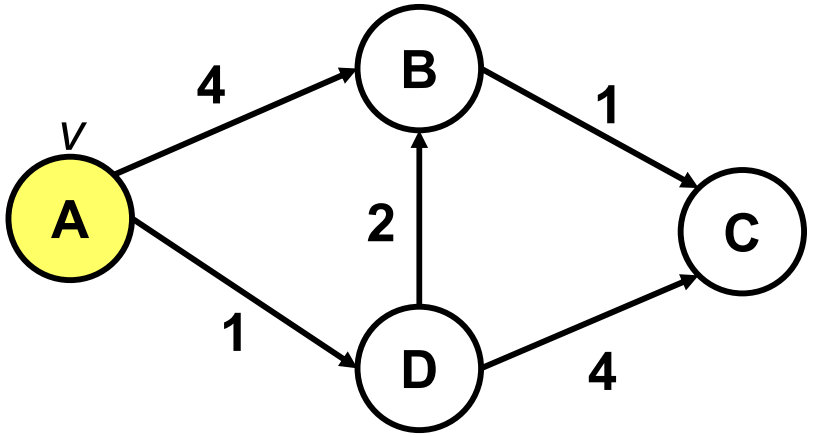
S = {A}

Vector D

B	C	D
4	∞	1

Vector P

B	C	D
-	-	-



Algoritmo de Dijkstra

Ejemplo

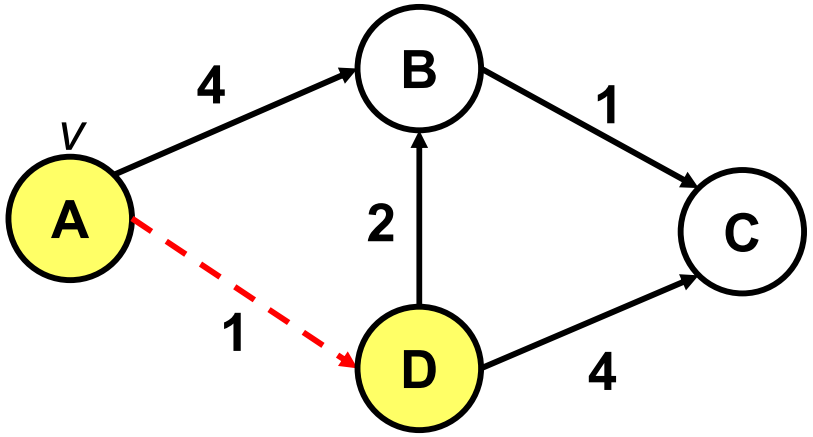
S = {A, D}

Vector D

B	C	D
3	5	1

Vector P

B	C	D
D	D	-



Algoritmo de Dijkstra

Ejemplo

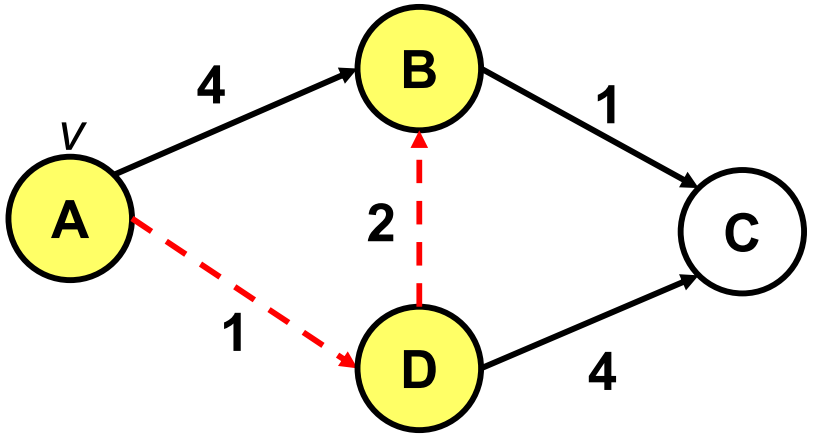
S = {A, D, B}

Vector D

B	C	D
3	4	1

Vector P

B	C	D
D	B	-



Algoritmo de Dijkstra

Ejemplo

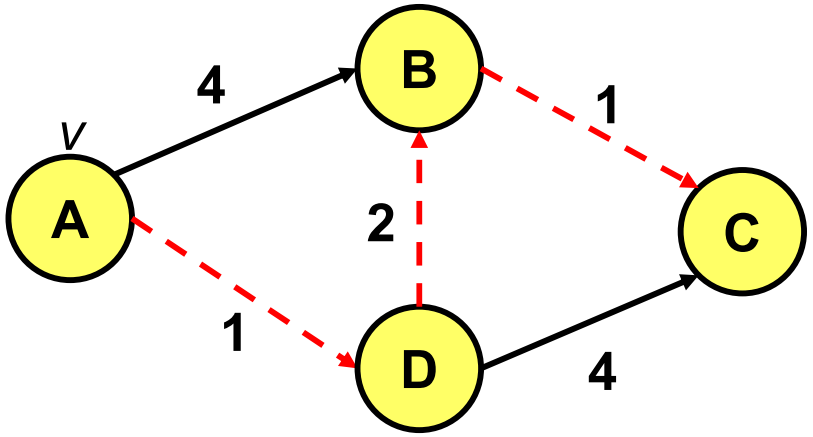
S = {A, D, B, C}

Vector D

B	C	D
3	4	1

Vector P

B	C	D
D	B	-



Algoritmo de Dijkstra

El Algoritmo

En cada Iteración…

1. Evaluar el coste de todos los arcos {k, w} en donde ***k*** pertenece al **conjunto S** y ***w*** al **conjunto V-S.**
2. Seleccionar aquel de coste mínimo, añadiendo ***w*** al **conjunto S.**
   a. w es el nodo con el **menor coste en D!**
3. **Para todo** nodo ***m*** de **V-S** hacer:

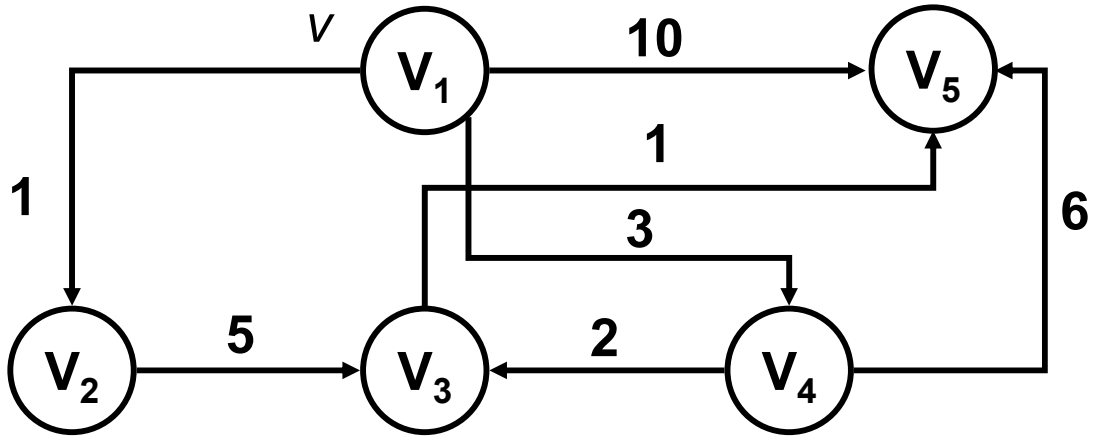
```
if (D[w] + weight[w][m] < D[m]) {
 D[m] = D[w] + weight[w][m];
 P[m] = w;
}
```

- **Condición de Parada**
  - **Conjunto S = Conjunto V** (se han explorado todos los nodos del grafo).
    - Realizadas **n – 1 iteraciones**.

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.

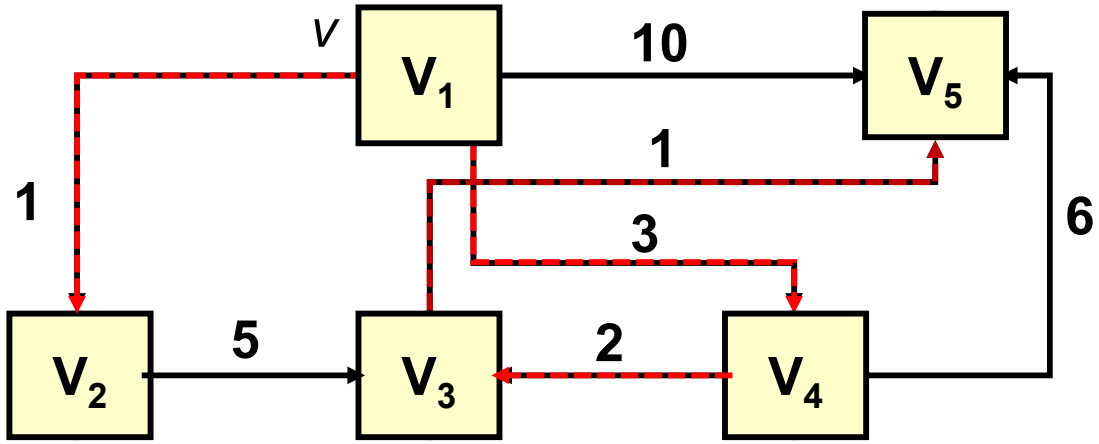


it	S	w	Vector D					Vector P				
			V_2	V_3	V_4	V_5	V_6	2	3	4	5	6
1	1		1	∞	3	10	--	1	-	1	1	-

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.



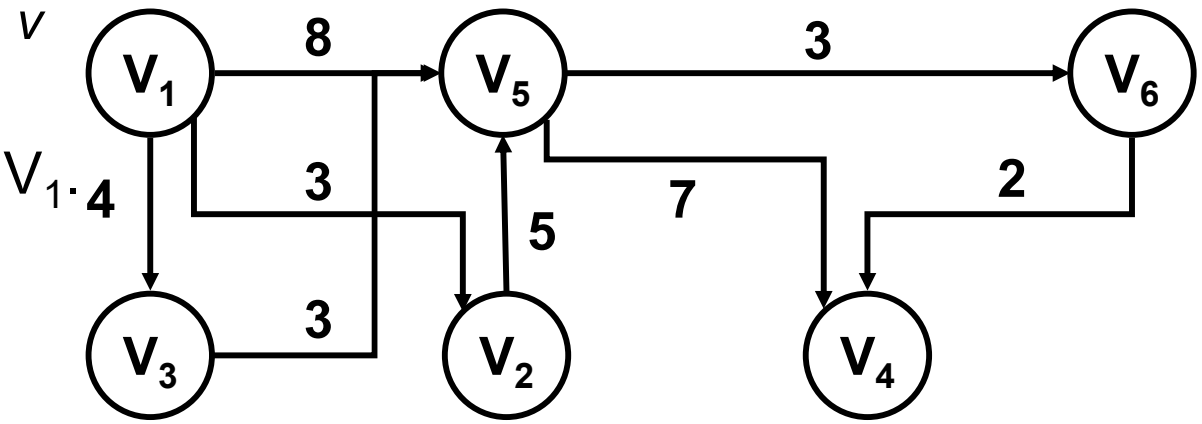
it	S	w
1	1	
2	1, 2	2
3	1, 2, 4	4
4	1, 2, 3, 4	3
5	1, 2, 3, 4, 5	5

Vector D					Vector P				
V_2	V_3	V_4	V_5	V_6	2	3	4	5	6
1	∞	3	10	--	1	-	1	1	-
1	6	3	10	--	1	2	1	1	-
1	5	3	9	--	1	4	1	4	-
1	5	3	6	--	1	4	1	3	-
1	5	3	6	--	1	4	1	3	-

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.

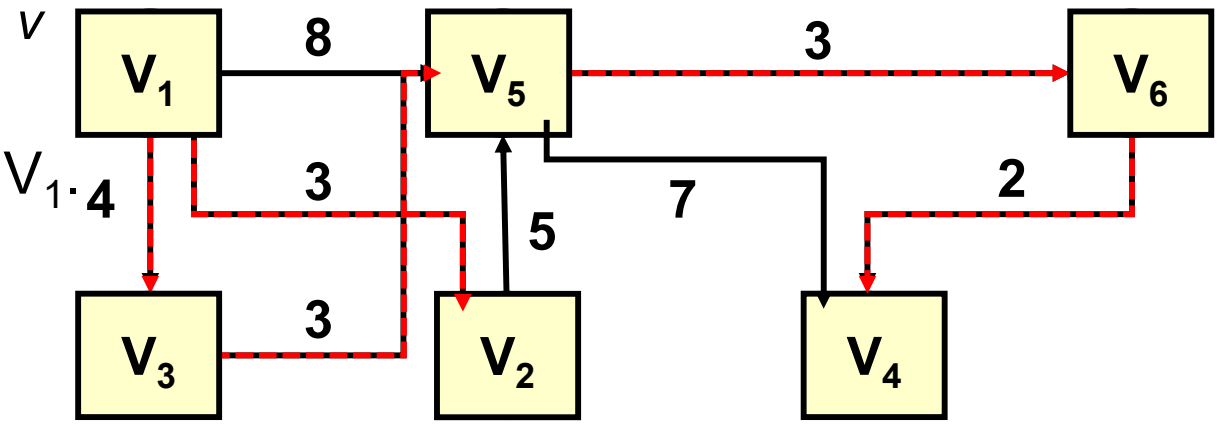


it	S	W	Vector D					Vector P				
			V_2	V_3	V_4	V_5	V_6	2	3	4	5	6
1	1		3	4	∞	8	∞	1	1	-	1	-

Algoritmo de Dijkstra

Ejercicio

- Costes partiendo de V_1.



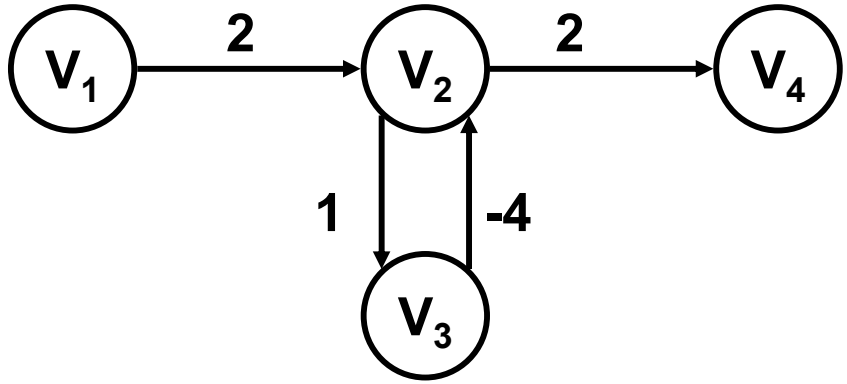
it	S	W	Vector D					Vector P				
			V_2	V_3	V_4	V_5	V_6	2	3	4	5	6
1	1		3	4	∞	8	∞	1	1	-	1	-
2	1, 2	2	3	4	∞	8	∞	1	1	-	1	-
3	1, 2, 3	3	3	4	∞	7	∞	1	1	-	3	-
4	1, 2, 3, 5	5	3	4	14	7	10	1	1	5	3	5
5	1, 2, 3, 5, 6	6	3	4	12	7	10	1	1	6	3	5
6	1, 2, 3, 4, 5, 6	4	3	4	12	7	10	1	1	6	3	5

Algoritmo de Dijkstra

Conclusiones

- Dijkstra supone el coste de ir de un nodo a si mismo como 0
  - Por ello no calcula D[v].

- El algoritmo no funciona con costes negativos (bonificaciones)
  - ¡El camino de coste mínimo no tiene porqué ser simple!



Coste mínimo entre V_1 y V_4 implicaría viajes infinitos entre V_2 y V_3

- Puede calcular el **Camino de Longitud Mínima**
  - Basta con **sustituir costes por 1 en *weight*.**

Algoritmo de Dijkstra

Complejidad Temporal

En cada Iteración…	**n – 1 iteraciones**

1. Evaluar el coste de todos los arcos {k, w} en donde ***k*** pertenece al **conjunto S** y ***w*** al **conjunto V-S**.
2. Seleccionar aquel de coste mínimo, añadiendo ***w*** al **conjunto S**.
   a. ***w*** es el nodo con el **menor coste en D!**
3. **Para todo** nodo ***m*** de **V-S** hacer:

```
if (D[w] + weight[w][m] < D[m]) {
 D[m] = D[w] + weight[w][m];
 P[m] = w;
}
```

O(n)

O(n)

$O(n^2)$

Algoritmo de Floyd-Warshall

Problema a Resolver

- Obtener caminos de coste mínimo entre ***cualquier*** par de nodos del grafo
  - ¿Cuál es la ruta más barata para llegar a Barcelona desde Oviedo, Sevilla o Burgos?
    - ¿Aplicar Dijkstra n veces? (una por cada nodo de partida).

Robert Floyd (Wikipedia)

Stephen Warshall (Wikipedia)

- Desarrollado por los estadounidenses Robert Floyd y Stephen Warshall en 1962.

Algoritmo de Floyd-Warshall

Productos Obtenidos (1/2)

- **Matriz A** (AKA matriz de Costes Mínimos)
  - Guarda el coste mínimo de ir **desde cualquier nodo a cada uno de los restantes** nodos del grafo.

Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	12	7	10
V_2	∞	0	∞	10	5	8
V_3	∞	∞	0	8	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	5	0	3
V_6	∞	∞	∞	2	∞	0

Algoritmo de Floyd-Warshall

Productos Obtenidos (2/2)

- **Matriz P** o de Rutas de Coste Mínimo
  - Almacena la secuencia de nodos que forman **todos los caminos** de coste mínimo.

printPath (fragmento)

```
private void printPath(int i, int j)
{
 int k = P[i][j];
 if (k>0) {
  printPath (i, k);
  System.out.print ('-' + k);
  printPath (k, j);
 }
}

System.out.print (departure);
printPath (departure, arrival);
System.out.println ('-' + arrival);
```

Matriz P

	1	2	3	4	5	6
V_1	-	-	-	6	3	5
V_2	-	-	-	6	-	5
V_3	-	-	-	6	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	6	-	-
V_6	-	-	-	-	-	-

Algoritmo de Floyd-Warshall

Inicialización

- Iniciar **Matriz A** de Coste Mínimo
  - Copia de todos los valores de una matriz **weight** modificada de forma idéntica al algoritmo de Dijkstra
    - **Sustituyendo** los valores de coste 0 por ∞.
    - **Pero**… **utilizando valores 0 en la diagonal principal** (el coste de ir de un nodo a si mismo se considera nulo).

Algoritmo de Floyd-Warshall

El Algoritmo

Floyd (fragmento) — **$O(n^3)$**

```
for (int k=0; k<size; k++)                          O(n)
 for (int i=0; i<size; i++)                         O(n)
  for (int j=0; j<size; j++)                        O(n)
    if (A[i][k] + A[k][j] < A[i][j])
    {
        A[i][j] = A[i][k] + A[k][j];
        P[i][j] = k;
    }
```

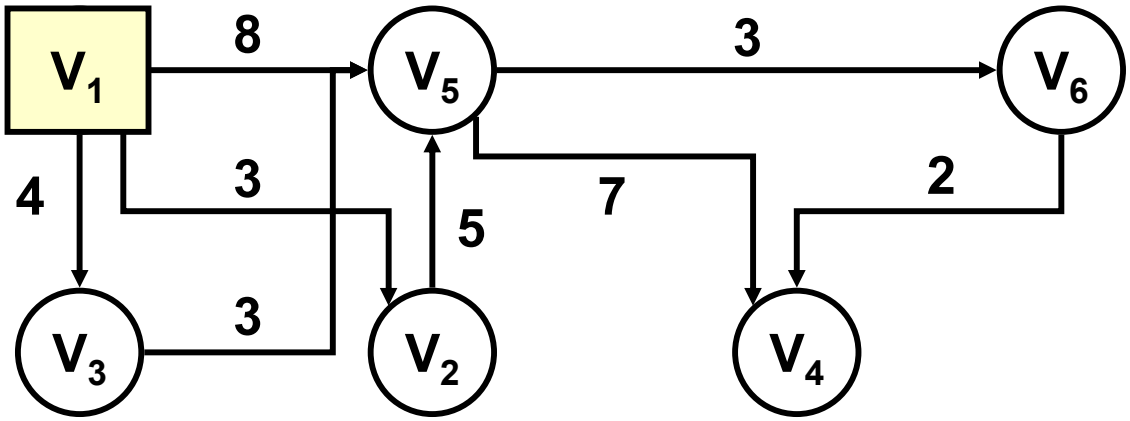
- En cada iteración se considera un nodo k **por el que hay** que pasar obligatoriamente
  - Se ejecutan **n iteraciones**
    - Equivalente de ir añadiendo uno a uno todos los nodos al conjunto S utilizado por Dijkstra.
  - En cada iteración se evalúa el coste de ir de **cualquier nodo i** a **cualquier nodo j** pasando por k.
    - **Si el coste es menor** que el registrado hasta entonces en A, se actualiza el valor de A[i,j] y de P[i,j] indicando que el camino de coste mínimo pasa por k.

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_0 (V_1)



Matriz A

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	8	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

Matriz P

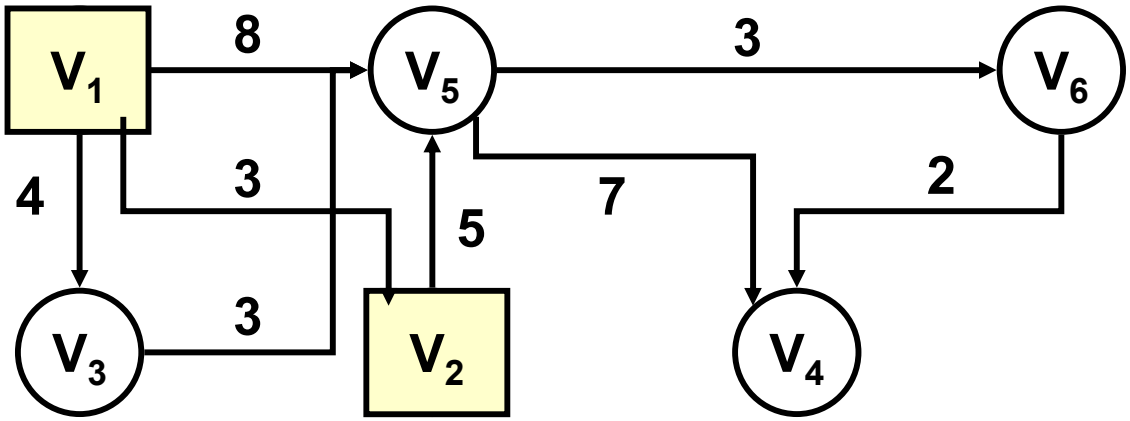
	1	2	3	4	5	6
V_1	-	-	-	-	-	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_2 a V_3 vía V_1 (coste ∞ + 4 = ∞) es más barato que ir directamente (coste ∞)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_1 (V_2)



Matriz A

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	8	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

Matriz P

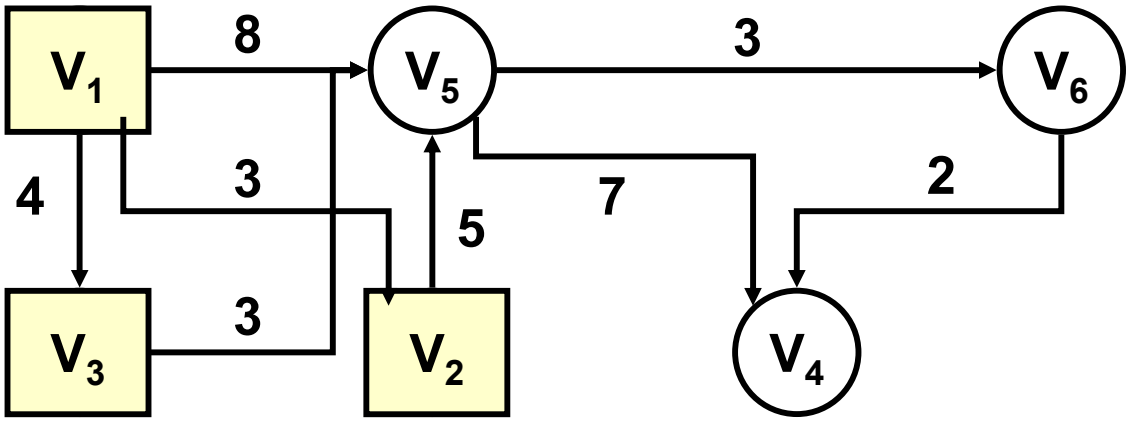
	1	2	3	4	5	6
V_1	-	-	-	-	-	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_5 vía V_2 (coste 3 + 5 = 8) es más barato que ir con coste A_0 (8)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_2 (V_3)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	7	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

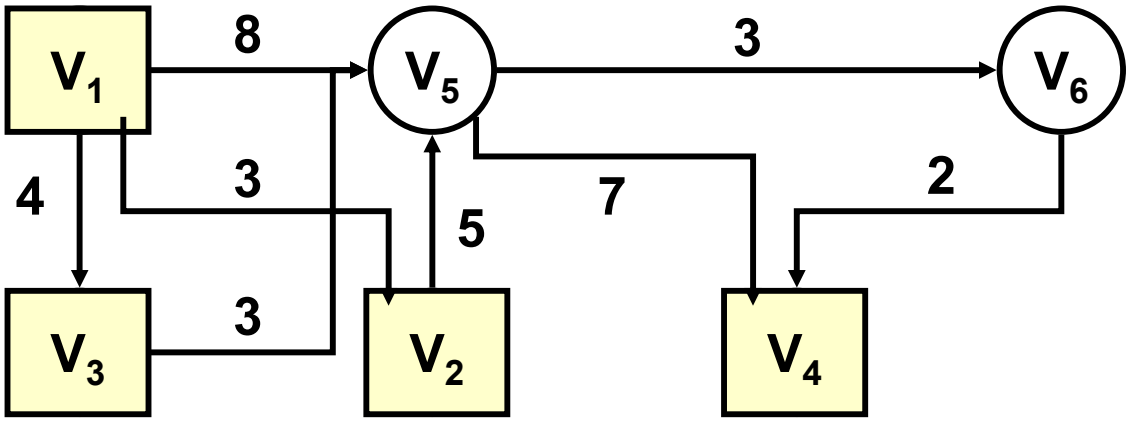
Matriz P	1	2	3	4	5	6
V_1	-	-	-	-	3	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_5 vía V_3 (coste 4 + 3 = 7) es más barato que ir con coste A_1 (8)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_3 (V_4)



Matriz A

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	7	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

Matriz P

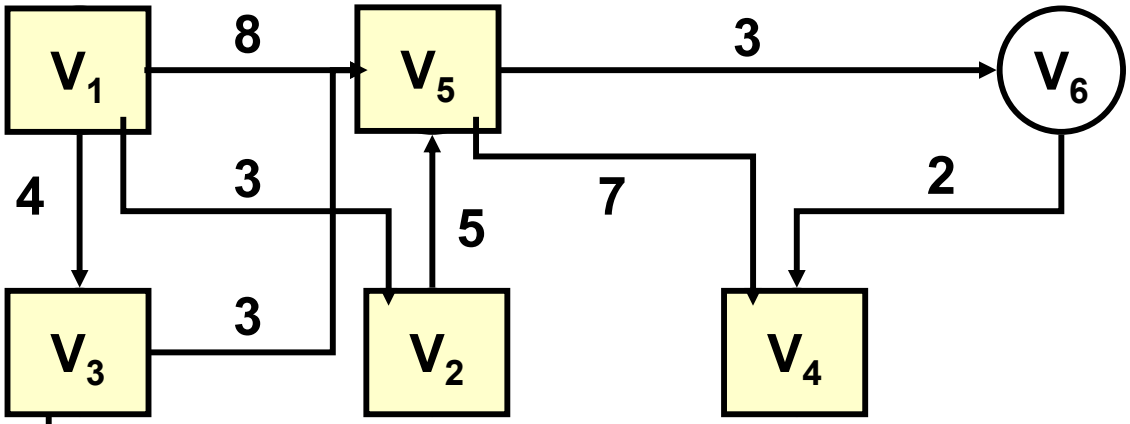
	1	2	3	4	5	6
V_1	-	-	-	-	3	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_5 a V_6 vía V_4 (coste 7 + ∞ = ∞) es más barato que ir con coste A_2 (3)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_4 (V_5)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	14	7	10
V_2	∞	0	∞	12	5	8
V_3	∞	∞	0	10	3	6
V_4	∞	∞	∞	0	∞	∞
→ V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

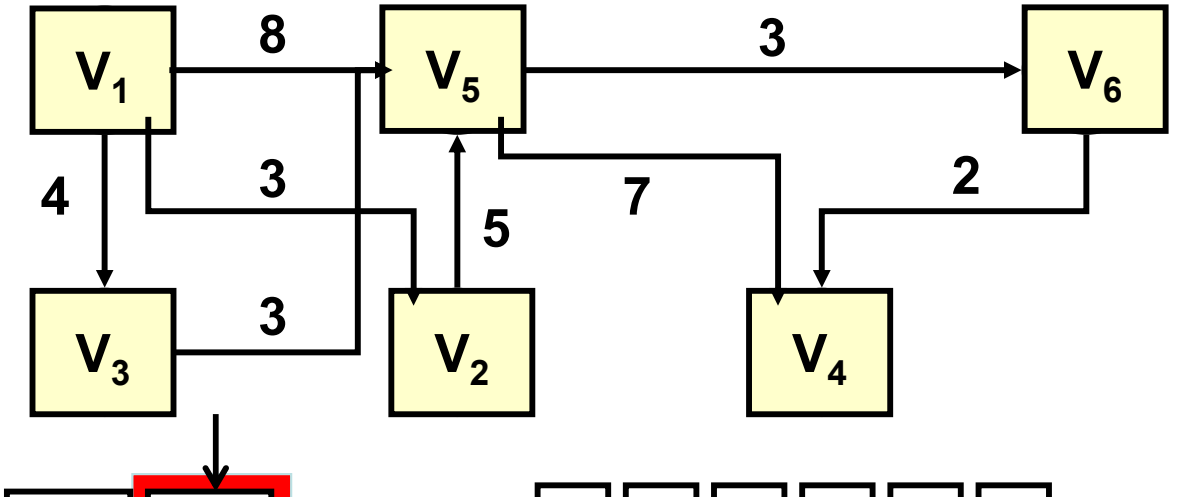
Matriz P	1	2	3	4	5	6
V_1	-	-	-	5	3	5
V_2	-	-	-	5	-	5
V_3	-	-	-	5	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_4 vía V_5 (coste 7 + 7 = 14) es más barato que ir con coste A_3 (∞)?

Algoritmo de Floyd-Warshall

Ejercicio 1

- Matriz A_5 (V_6)



Matriz A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	12	7	10
V_2	∞	0	∞	10	5	8
V_3	∞	∞	0	8	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	5	0	3
→ V_6	∞	∞	∞	2	∞	0

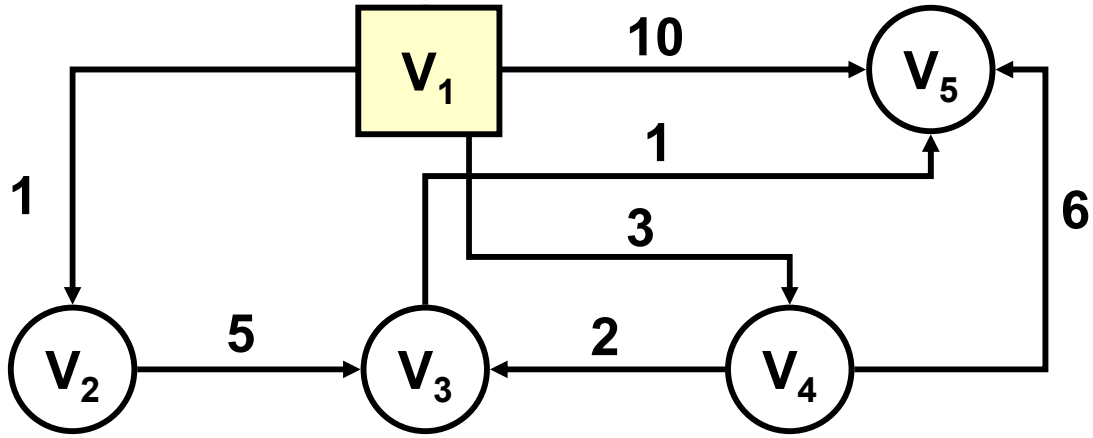
Matriz P	1	2	3	4	5	6
V_1	-	-	-	6	3	5
V_2	-	-	-	6	-	5
V_3	-	-	-	6	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	6	-	-
V_6	-	-	-	-	-	-

¿Ir de V_1 a V_4 vía V_6 (coste 10 + 2 = 10) es más barato que ir con coste A_4 (14)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_0 (V_1)



Matriz A

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	∞	3	10
V_2	∞	0	5	∞	∞
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	6
V_5	∞	∞	∞	∞	0

Matriz P

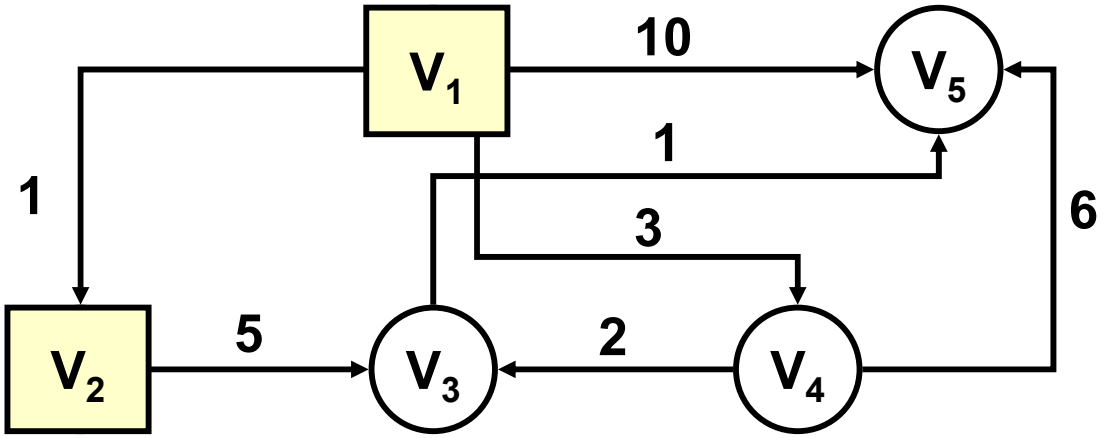
	1	2	3	4	5
V_1	-	-	-	-	-
V_2	-	-	-	-	-
V_3	-	-	-	-	-
V_4	-	-	-	-	-
V_5	-	-	-	-	-

¿Ir de V_4 a V_5 vía V_1 (coste ∞ + 10 = ∞) es más barato que ir directamente (6)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_1 (V_2)



Matriz A

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	6	3	10
V_2	∞	0	5	∞	∞
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	6
V_5	∞	∞	∞	∞	0

Matriz P

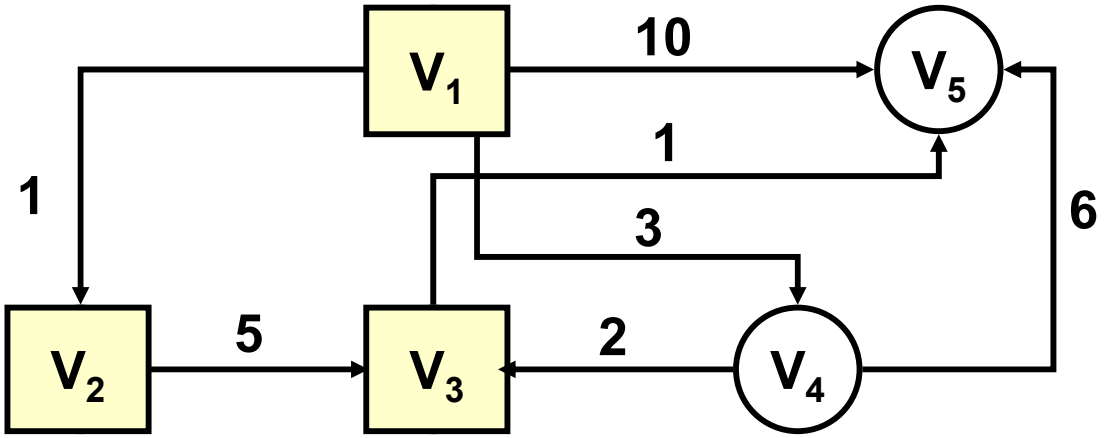
	1	2	3	4	5
V_1	-	-	2	-	-
V_2	-	-	-	-	-
V_3	-	-	-	-	-
V_4	-	-	-	-	-
V_5	-	-	-	-	-

¿Ir de V_1 a V_3 vía V_2 (coste 1 + 5 = 6) es más barato que ir con coste A_0 (∞)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_2 (V_3)



Matriz A

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	6	3	7
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

Matriz P

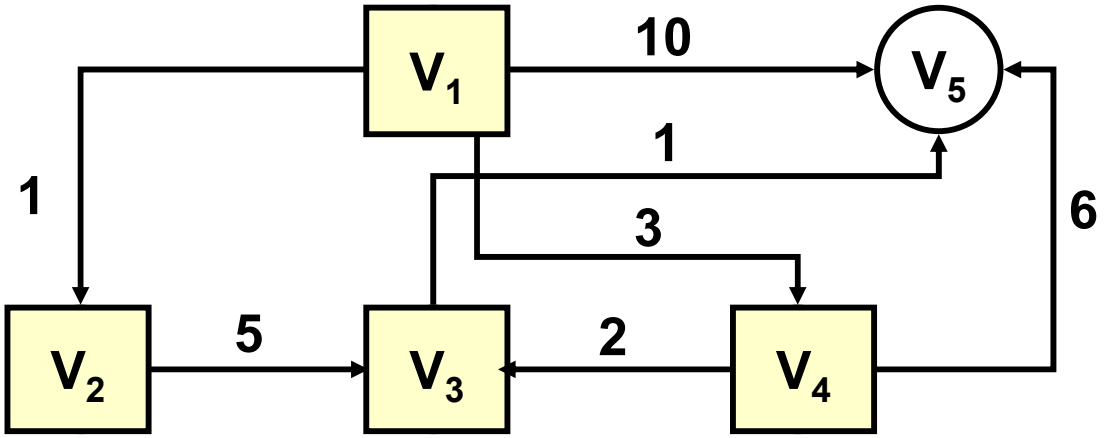
	1	2	3	4	5
V_1	-	-	2	-	3
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

¿Ir de V_1 a V_5 vía V_3 (coste 6 + 1 = 7) es más barato que ir con coste A_1 (10)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_3 (V_4)



Matriz A

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	5	3	6
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

Matriz P

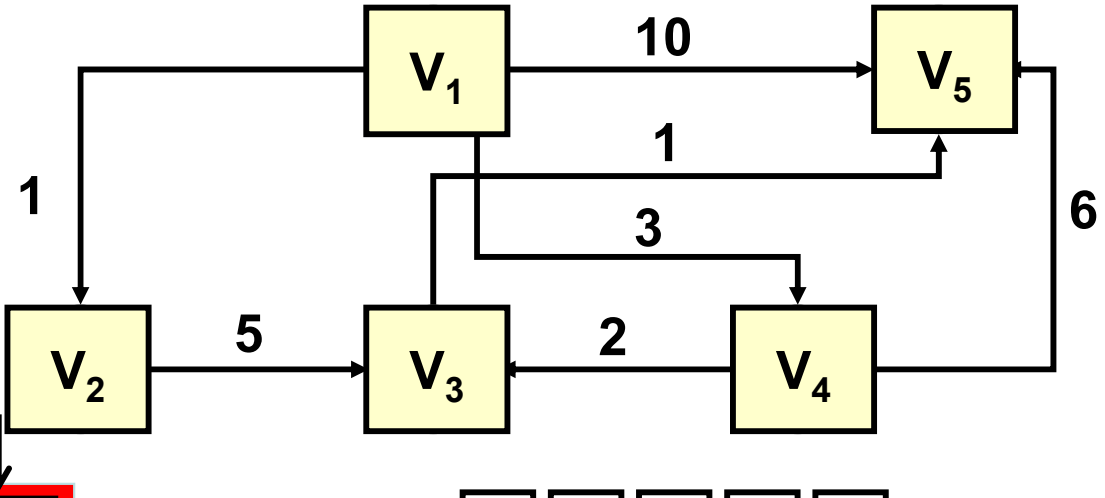
	1	2	3	4	5
V_1	-	-	4	-	4
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

¿Ir de V_1 a V_3 vía V_4 (coste 3 + 3 = 5) es más barato que ir con coste A_2 (6)?

Algoritmo de Floyd-Warshall

Ejercicio 2

- Matriz A_4 (V_5)



Matriz A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	5	3	6
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
→ V_5	∞	∞	∞	∞	0

Matriz P	1	2	3	4	5
V_1	-	-	4	-	4
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

¿Ir de V_1 a V_2 vía V_5 (coste 6 + ∞ = ∞) es más barato que ir con coste A_3 (1)?

Algoritmo de Floyd-Warshall

Floyd para rutas especiales

- Es posible mejorar el algoritmo para calcular caminos de coste mínimo que pasen por un **conjunto L de nodos**.

Floyd (fragmento)

```
    for (int k=0; k<size; k++)
     if (k in L)
      for (int i=0; i<size; i++)
       for (int j=0; j<size; j++)
        if (A[i][k] + A[k][j] < A[i][j])
        {
            A[i][j] = A[i][k] + A[k][j];
            P[i][j] := k;
        }
```

Algoritmo de Floyd-Warshall

Centro de un Grafo Dirigido

- Es centro de un grafo es aquel nodo ***v* más cercano al nodo más distante**.
  - ¿Dónde ubicar un centro de distribución en una región?
  - ¿Dónde colocar el hospital o la estación central en una ciudad?

- Excentricidad
  - La excentricidad de un nodo ***v*** es el **máximo de los costes de todos los caminos** de coste mínimo con destino ***v***.
  - El centro de un grafo se encuentra en aquel nodo de **mínima** excentricidad.

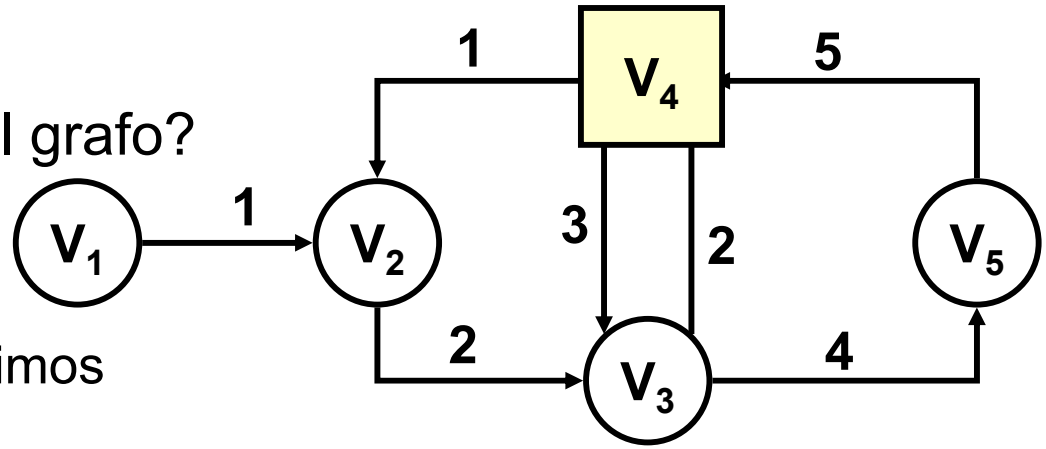
Algoritmo de búsqueda del centro de un grafo

1. Aplicar Floyd para **obtener matriz de costes mínimos.**
2. Buscar el **coste mayor en cada columna** (excentricidad de cada nodo destino).
3. Elegir aquel nodo con **la menor excentricidad** como centro del grafo.

Algoritmo de Floyd-Warshall

Ejercicio

- ¿Qué nodo es el centro del grafo?



Obtener mínimo de los máximos

Matriz A Original

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	∞	∞	∞
V_2	∞	0	2	∞	∞
V_3	∞	∞	0	2	4
V_4	∞	1	3	0	∞
V_5	∞	∞	∞	5	0

Matriz A Final

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	3	5	7
V_2	∞	0	2	4	6
V_3	∞	3	0	2	4
V_4	∞	1	3	0	7
V_5	∞	6	8	5	0

Buscar máximo en cada columna

Recorrido en Profundidad (DFPrint)

Problema a Resolver

- Recorrer todos los nodos de un grafo a partir de un nodo inicial, siguiendo el camino señalado por sus sus arcos.
  - Emplea la estrategia *visitar primero a los hijos (depth-first) y luego a los hermanos*.
  - Es necesario llevar un control de los nodos visitados.

resetVisited	**O(n)**

```
public void resetVisited ()
{
 for (int i=0; i<size; i++)
  nodes[i].setVisited(false);
}
```

Recorrido en Profundidad (DFPrint)

Problema a Resolver

- Recorrer todos los nodos de un grafo a partir de un nodo inicial, siguiendo el camino señalado por sus sus arcos.
  - Emplea la estrategia *visitar primero a los hijos (depth-first) y luego a los hermanos*.
  - Es necesario llevar un control de los nodos visitados.

Deep-first print (pseudocode)

```
public void DFPrint(int v){
  nodes[v].setVisited(true);
  nodes[v].print();

  for each node w accessible from v do
   if (!nodes[w].getVisited())
    DFPrint(w);
}
```

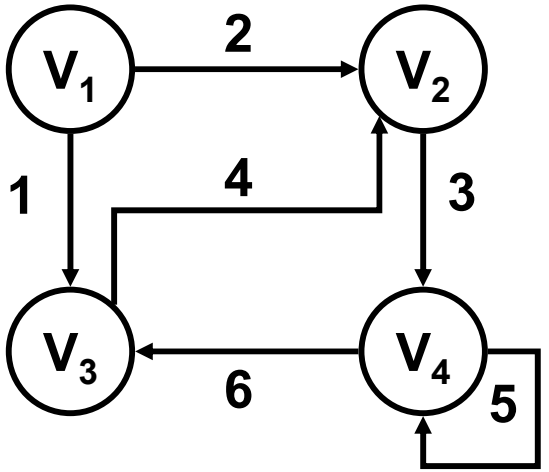
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_1)*

- Antes de visitar V_1

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	F	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



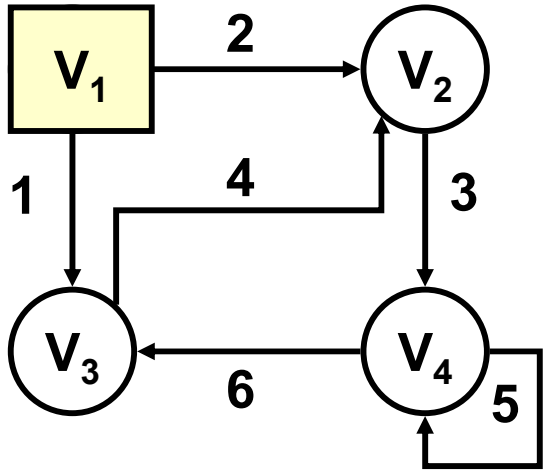
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_1

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	F	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



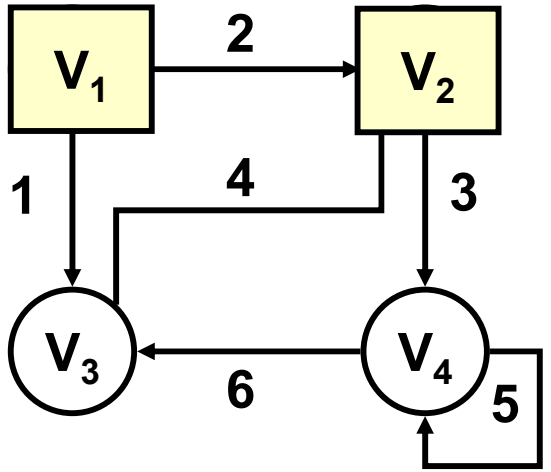
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_2

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



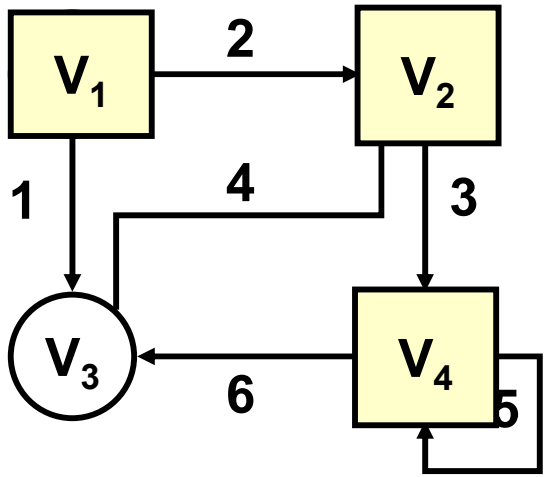
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_4

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	F	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	

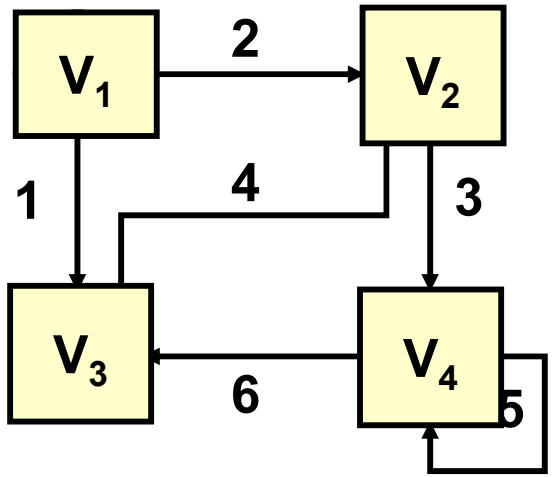


Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_1)

- Visitando V_3

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	T	T	

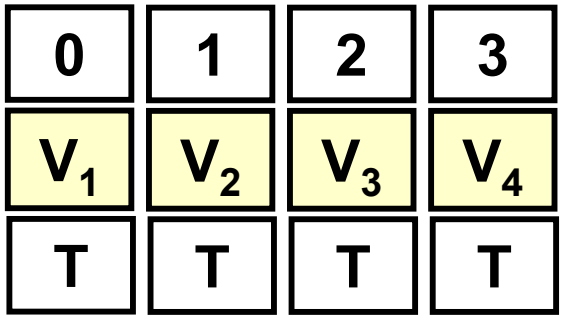


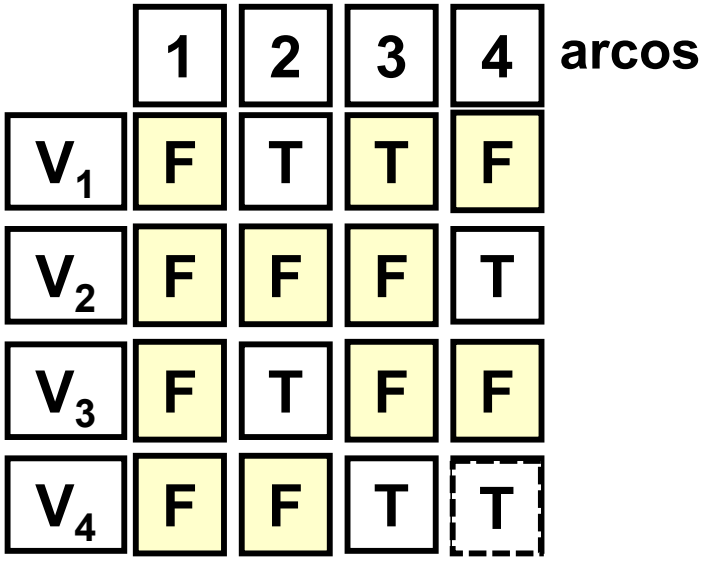
	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	

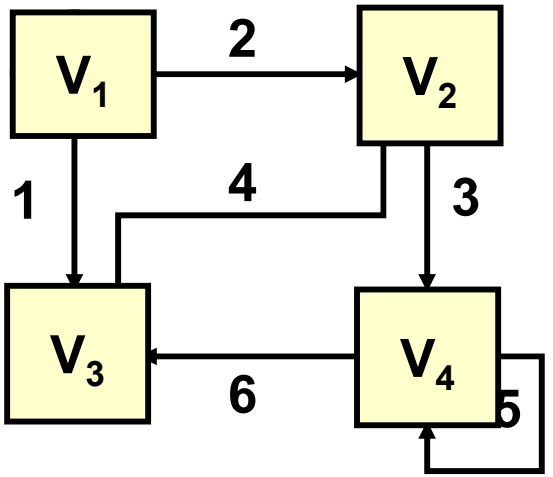
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_1)*

- Continuamos visita de V_4





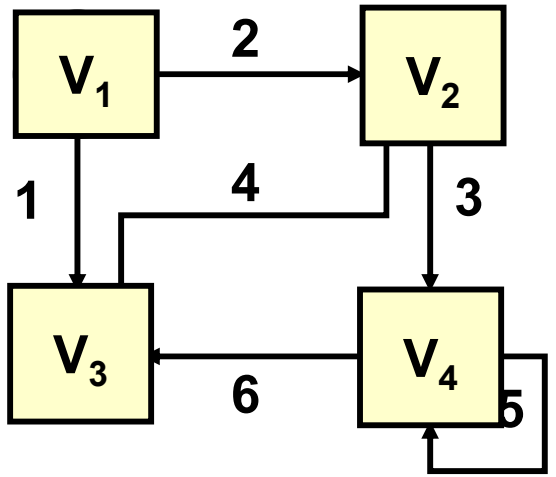


Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint (V_1)*

- Continuamos visita de V_1

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
T	T	T	T	



	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	n
V_3	F	T	F	F	
V_4	F	F	T	T	
		n			

$O(n^2)$

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

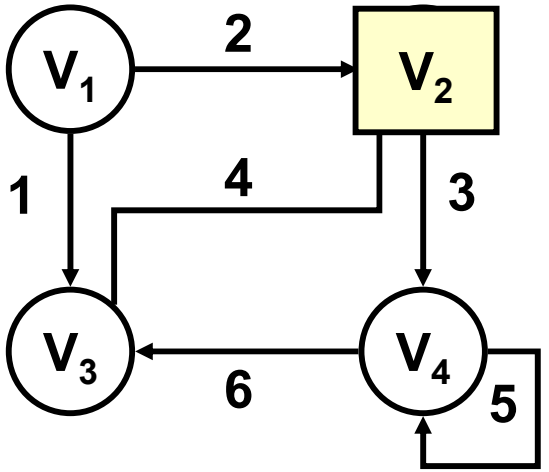
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_2)

- Visitando V_2

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	F	F	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



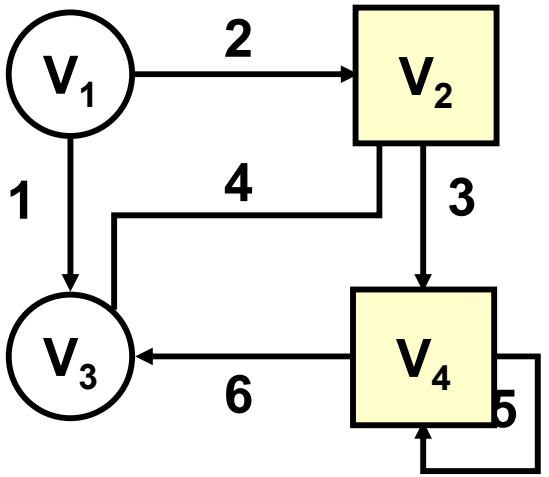
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_2)

- Visitando V_4

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	F	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



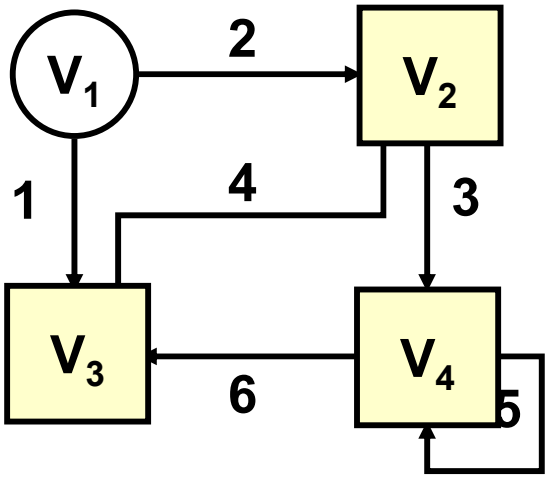
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_2)

- Visitando V_3

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



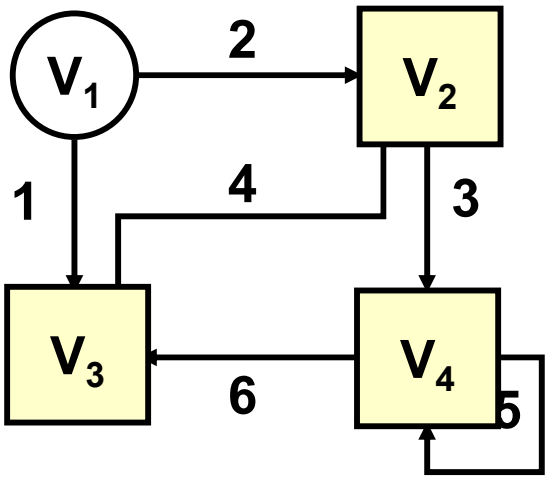
Recorrido en Profundidad (DFPrint)

Ejercicio *DFPrint* (V_2)

- Continuamos visita de V_4

0	1	2	3	nodos
V_1	V_2	V_3	V_4	
F	T	T	T	

	1	2	3	4	arcos
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



Recorrido en Profundidad (DFPrint)

Para garantizar el recorrido completo del grafo

Invocación especial a DFPrint

```
resetVisited();

For (int i=0; i<size; i++)
 if (!nodes[i].getVisited())
   DFPrint (i);
```

Recorrido en Profundidad (DFPrint)

Búsqueda **primero** en profundidad

- Modificación de *DFPrint* para detener el recorrido una vez cumplida una determinada condición sobre un nodo concreto.

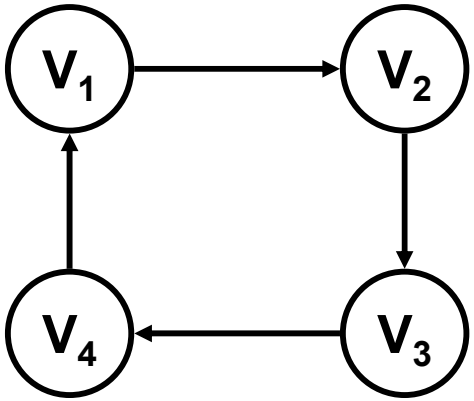
DFSearch (pseudocode)

```
public boolean DFPrint(int v){
  nodes[v].setVisited(true);
  nodes[v].print();

  if (boolean_condition(v))
   return (true);

   for each node w accessible from v do
    if (!nodes[w].getVisited())
     DFPrint(w);

  return (false);
}
```

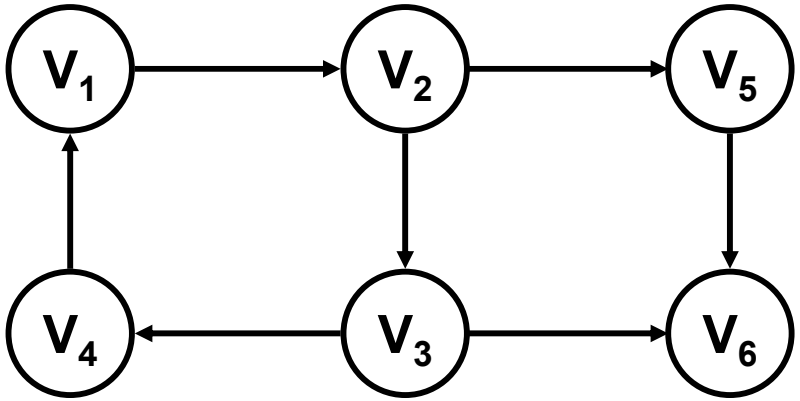
Más Conceptos Básicos

- **Nodo Fuertemente Conexo**
  - Si desde el nodo se puede acceder a todos los demás nodos del grafo **Y** viceversa.

- **Grafo Fuertemente Conexo**
  - Si **todos** los nodos del grafo son fuertemente conexos.
    - Si existe un nodo fuertemente conexo, todos los demás también lo serán y por ende, también el grafo.



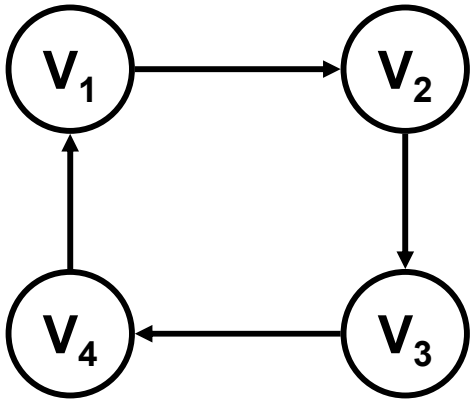
Grafo fuertemente conexo



Grafo no conexo (ver V_6)

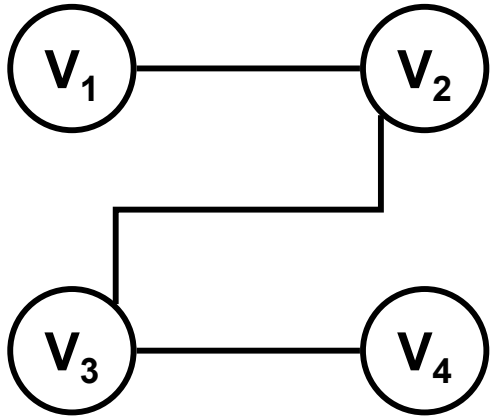
Más Conceptos Básicos

- **Ciclo sobre un nodo**
  - Camino desde un nodo hasta si mismo.
    - Ciclo para V_1
      - » $C = V_1, V_2, V_3, V_4$ (longitud 4).



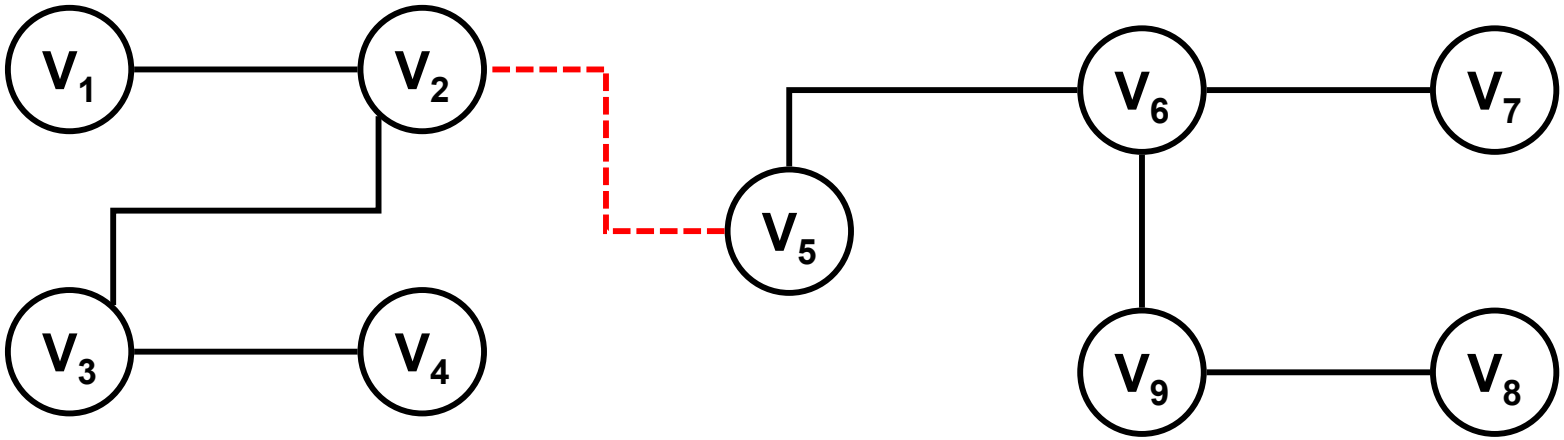
Más Conceptos Básicos

- **Árbol**
  - Grafo Conexo sin Ciclos
    - Todo árbol libre con n > 0 nodos, tiene n - 1 aristas.
    - Si se agrega una arista, ésta formará parte de un ciclo (el grafo deja de ser árbol libre).
    - Para cualquier par de nodos, sólo hay un camino simple.



Más Conceptos Básicos

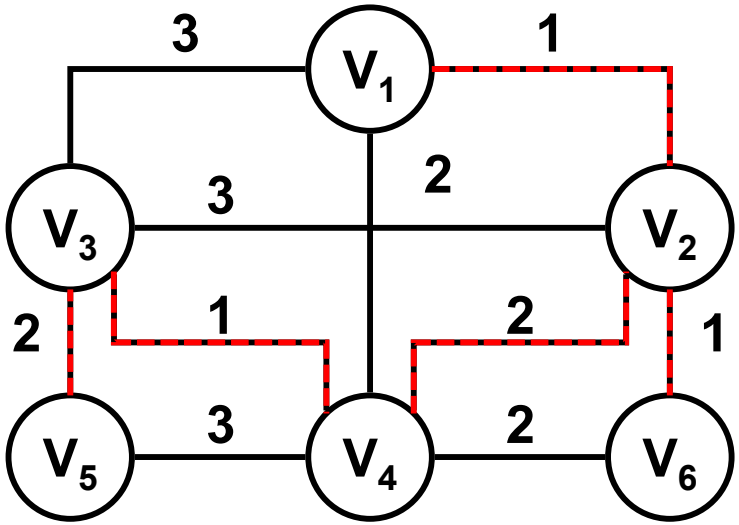
- **Árbol Abarcador**
  - Árbol que conecta todos los **nodos** del grafo.
    - El árbol forma **una única componente conexa.**



Las dos componentes conexas necesitan contactarse entre ellas para formar un árbol libre abarcador

Más Conceptos Básicos

- **Árbol Libre Abarcador de Coste Mínimo**
  - Aquel en el que la suma de los pesos de sus aristas es la mínima posible.
    - Permite conectar todos los componentes de una red al menor coste.



Algoritmo de Prim

Problema a Resolver

- Dado un árbol libre abarcador, devolver el equivalente de coste mínimo
  - ¿Qué carreteras se deben construir para conectar todas las ciudades de Europa de la forma más barata?
  - ¿Cómo se pueden conectar todos los ordenadores de una red con la menor longitud de cable?

Robert C. Prim (Wikipedia)

- Desarrollado por el estadounidense Robert C. Prim en 1957

Algoritmo de Prim

Inicialización

- **Conjunto T** (vacío)
  - En donde se irán almacenando las aristas que formarán parte del Árbol Libre Abarcador de coste mínimo.

- **Conjunto U** (se inicia con un nodo cualquiera del grafo)
  - Similar al conjunto S del algoritmo de Dijkstra, almacena los nodos que se van evaluando en cada iteración.

En cada iteracción (mientras que U != V)

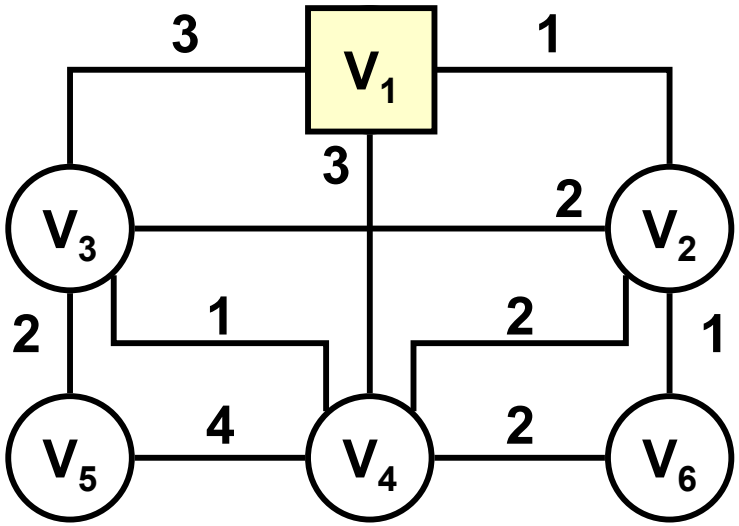
```
1. Evaluar todas las aristas {u, v} en las que u
   pertenezca a U y v pertenezca a V – U y quedarse con la
   de menor coste
2. T = T + {u, v}
3. U = U + {v}
```

- **Condición de Parada**
  - **Conjunto U != Conjunto V** (se han explorado todos los nodos del grafo).
    - Realizadas **n – 1 iteraciones**.

Algoritmo de Prim

Ejercicio 1

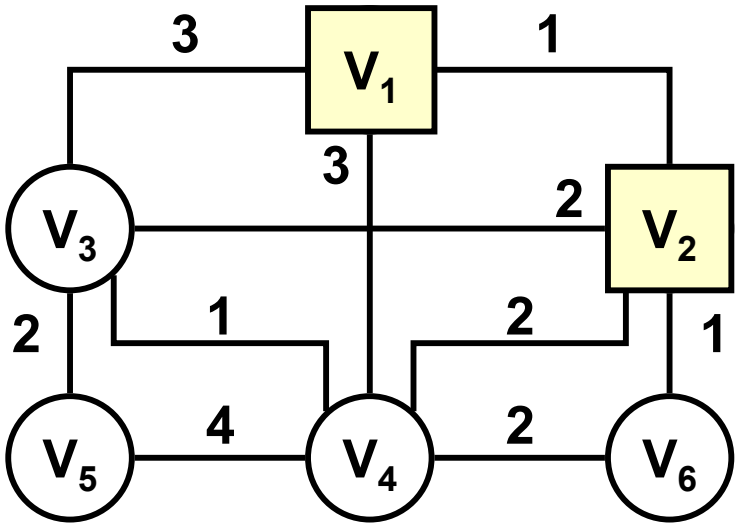
- Empezando con V_1.



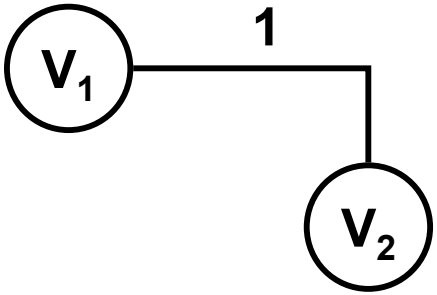
it	U	w
1	1	

Algoritmo de Prim

Ejercicio 1

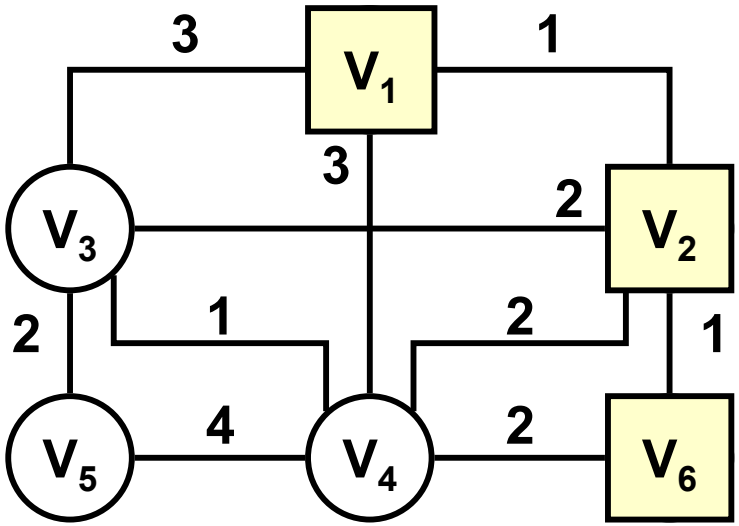


it	U	w
1	1	
2	1, 2	2

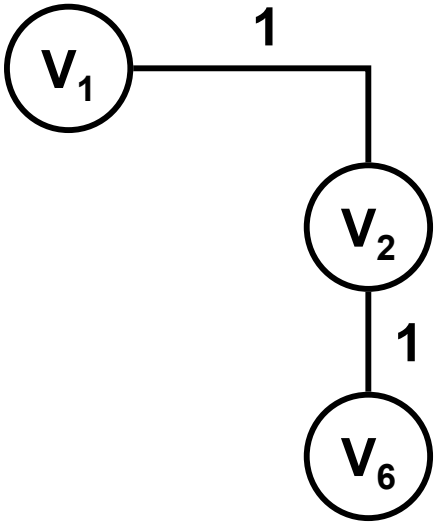


Algoritmo de Prim

Ejercicio 1



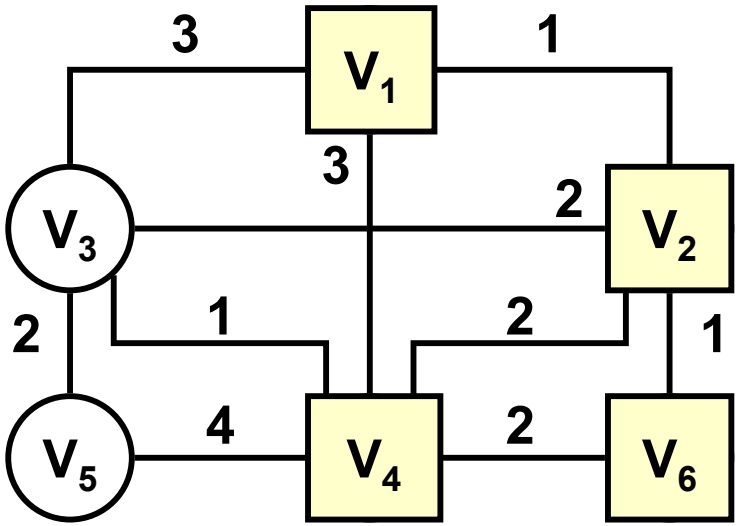
it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6



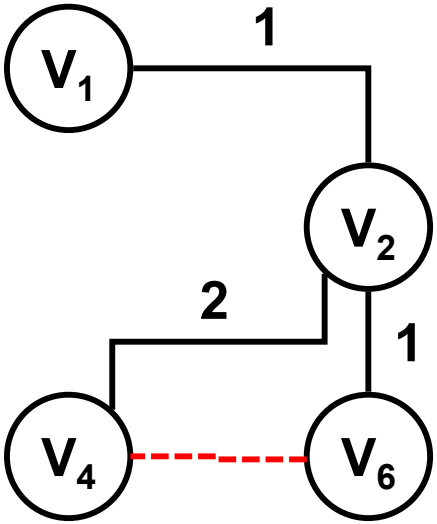
Algoritmo de Prim

Ejercicio 1

- También se podría escoger V_3

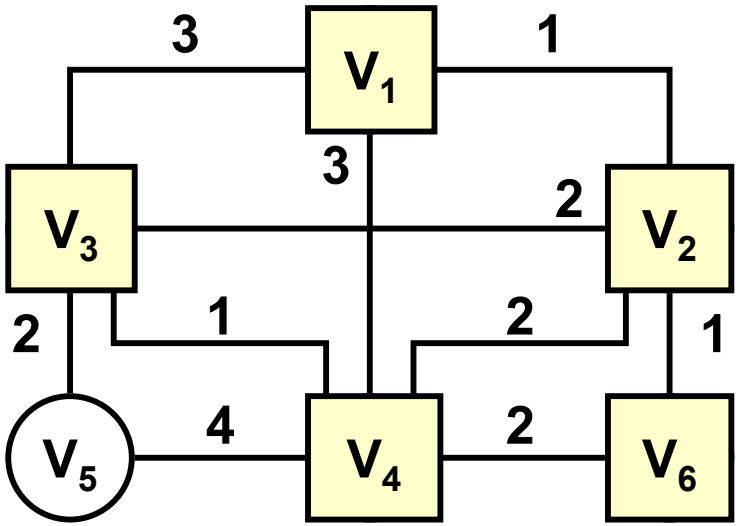


it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4

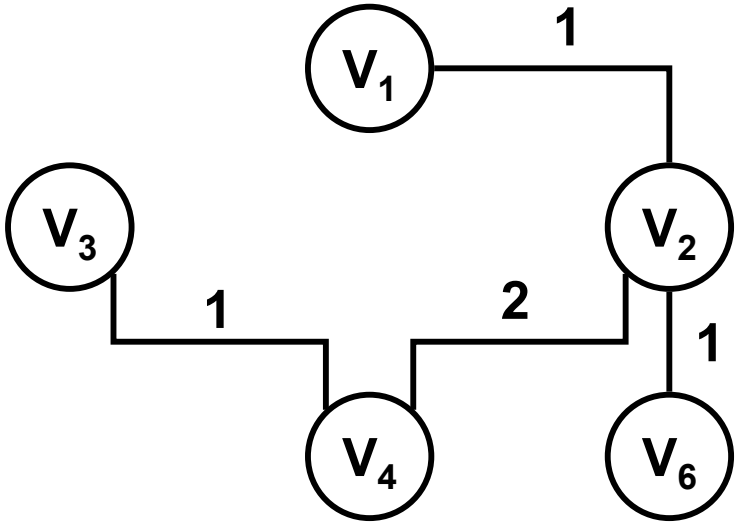


Algoritmo de Prim

Ejercicio 1

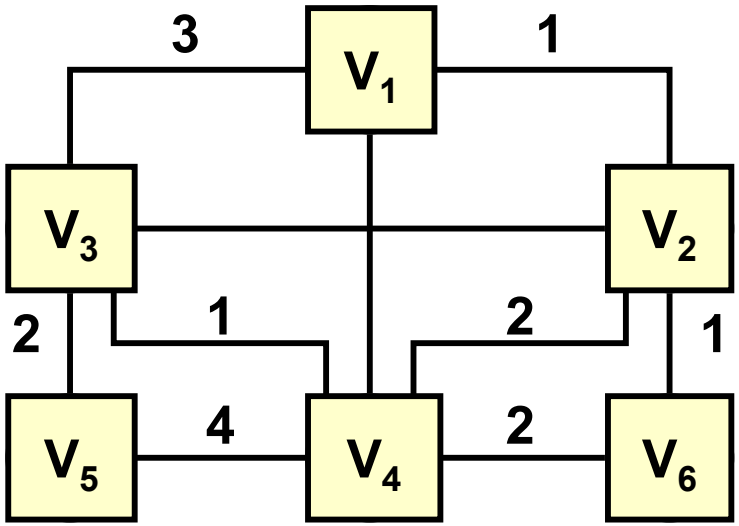


it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4
5	1, 2, 3, 4, 6	3

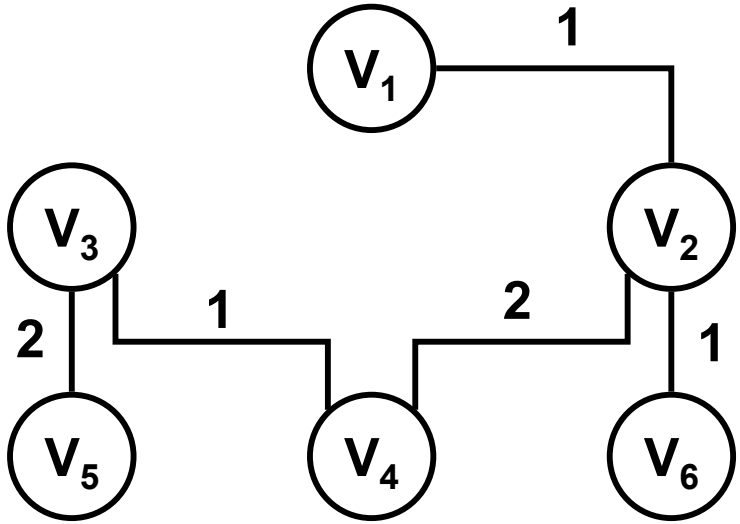


Algoritmo de Prim

Ejercicio 1



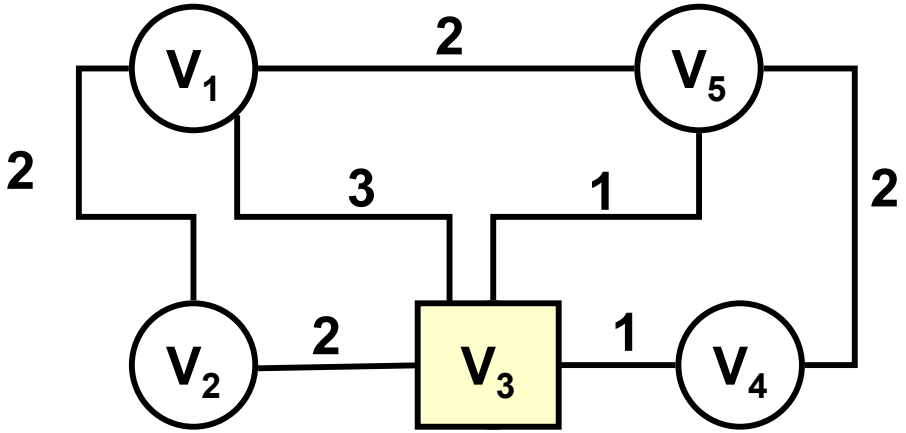
it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4
5	1, 2, 3, 4, 6	3
6	1, 2, 3, 4, 5, 6	5



Algoritmo de Prim

Ejercicio 2

- Empezando con V_3.

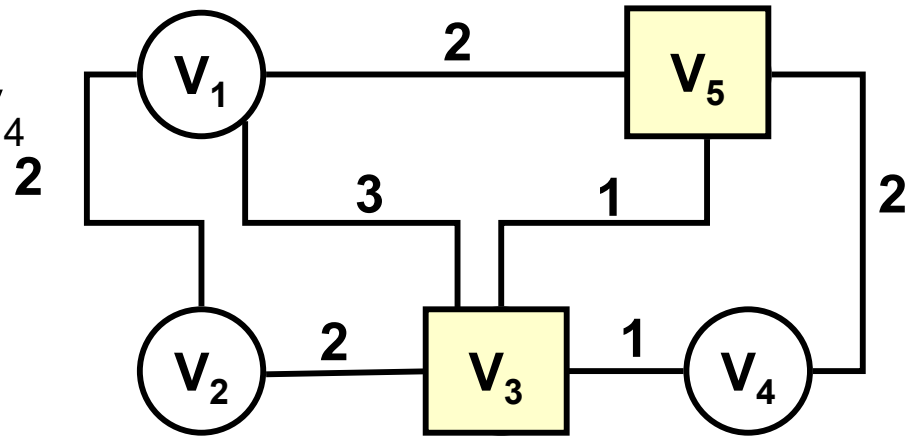


it	U	w
1	3	
2		
3		
4		
5		

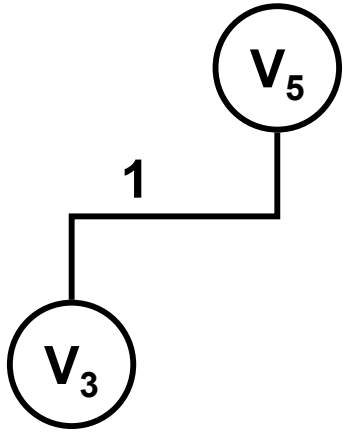
Algoritmo de Prim

Ejercicio 2

- También se podría escoger V_4

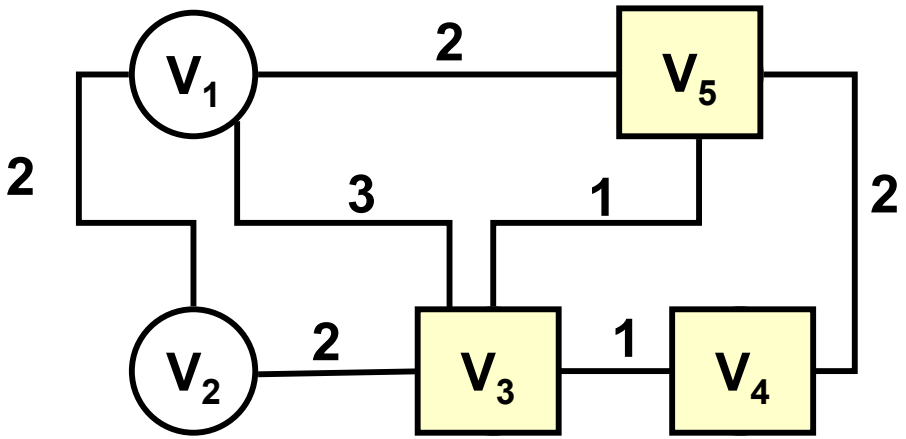


it	U	w
1	3	
2	3, 5	5
3		
4		
5		

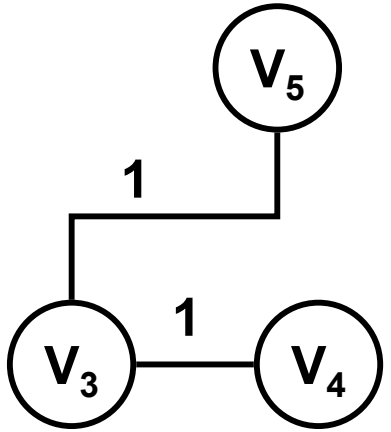


Algoritmo de Prim

Ejercicio 2



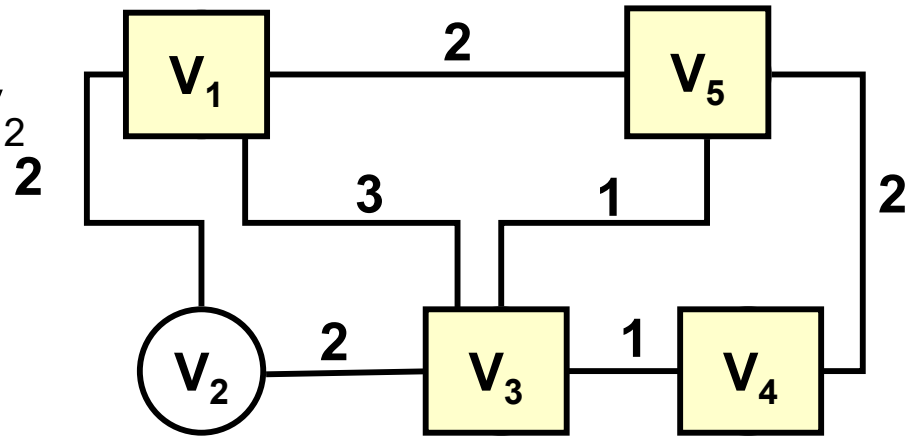
it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4		
5		



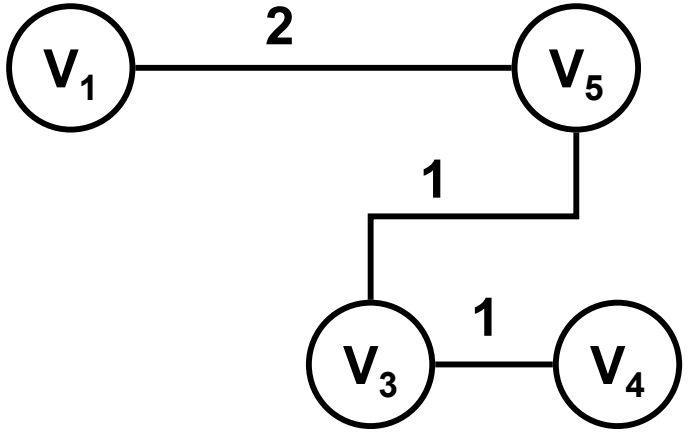
Algoritmo de Prim

Ejercicio 2

- También se podría escoger V_2



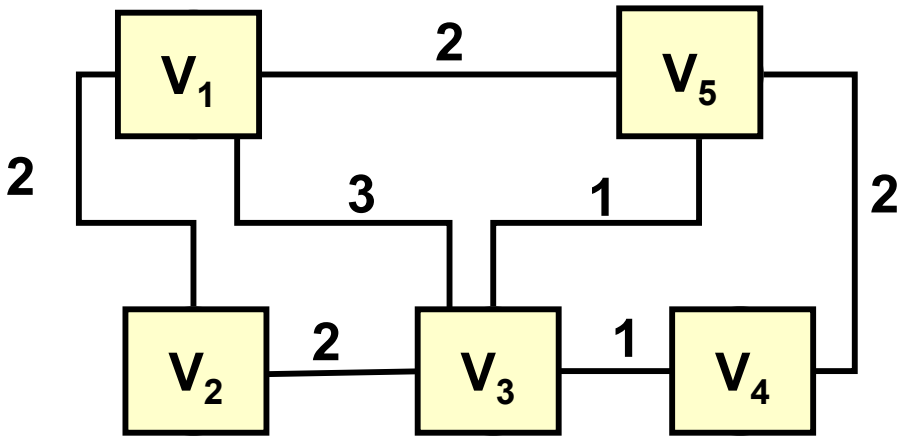
it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4	1, 3, 4, 5	1
5		



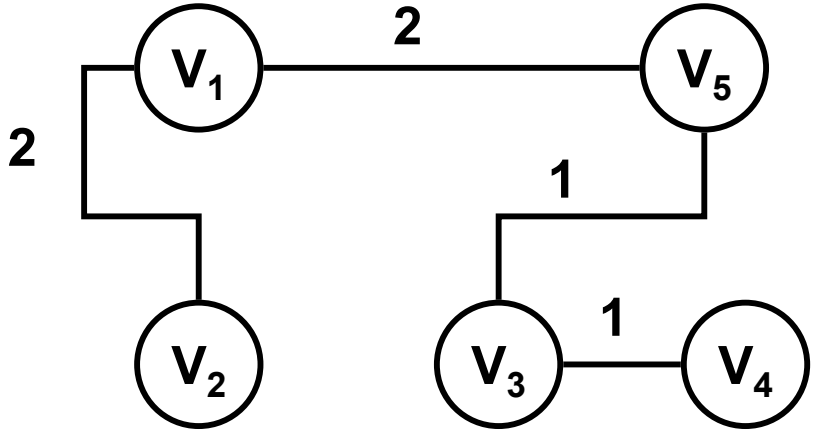
Algoritmo de Prim

Ejercicio 2

- Alternativa: $\{V_2, V_3\}$



it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4	1, 3, 4, 5	1
5	1, 2, 3, 4, 5	2



Algoritmo de Prim

Conclusiones

- El árbol resultante depende de…
  - Nodo de partida.
  - Selección de la arista de coste mínimo en cada iteración.
    - Puede existir más de una con el coste más pequeño.

En cada iteracción (hasta que U == V)	n
1. Evaluar todas las aristas {u, v} en las que u pertenezca a U y v pertenezca a V – U y quedarse con la de **menor** coste 2. T = T + {u,v} 3. U = U + {v}	n^2

$O(n^3)$

Algoritmo de Prim

- Optimización
  - Utilizar vectores auxiliares **ordenados** para elegir la arista de menor coste, reduciendo la complejidad a O(n).
    - Mayor velocidad a costa de mayor consumo de memoria.

En cada iteracción (hasta que U == V)	**n**
1. Evaluar todas las aristas {u, v} en las que u pertenezca a U y v pertenezca a V – U y quedarse con la de **menor** coste 2. T = T + {u,v} 3. U = U + {v}	**n**

$O(n^2)$

C59 Series

Estructuras Jerárquicas

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.).

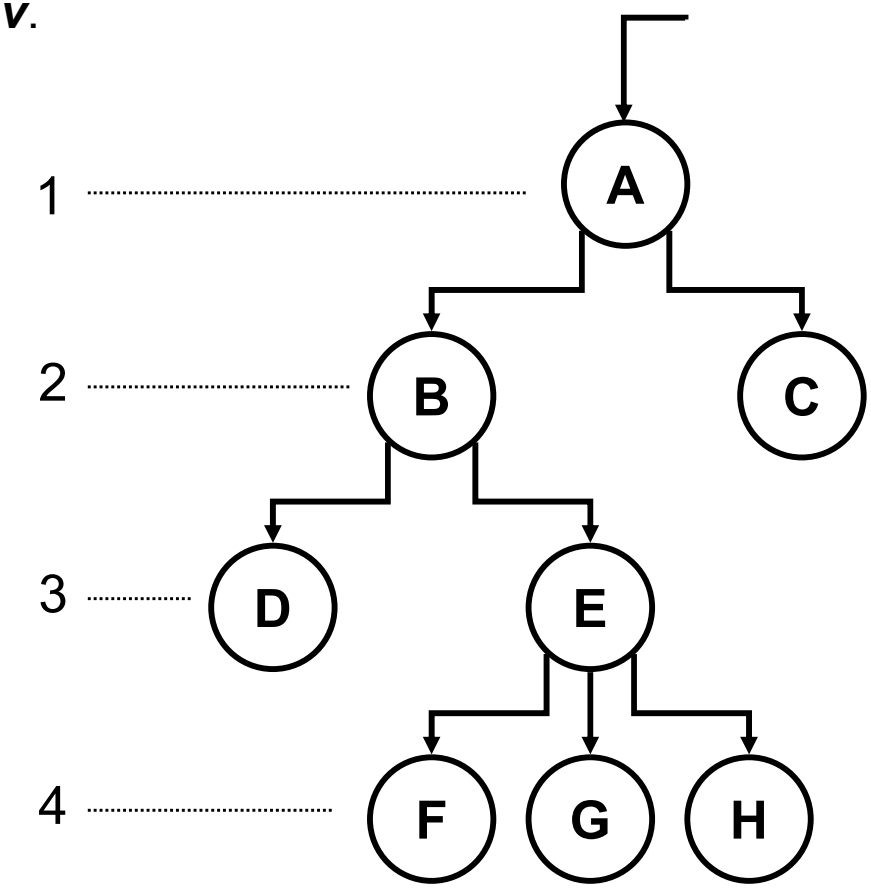
Conceptos Básicos

¿Qué es un Árbol?

- En informática[1] 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).

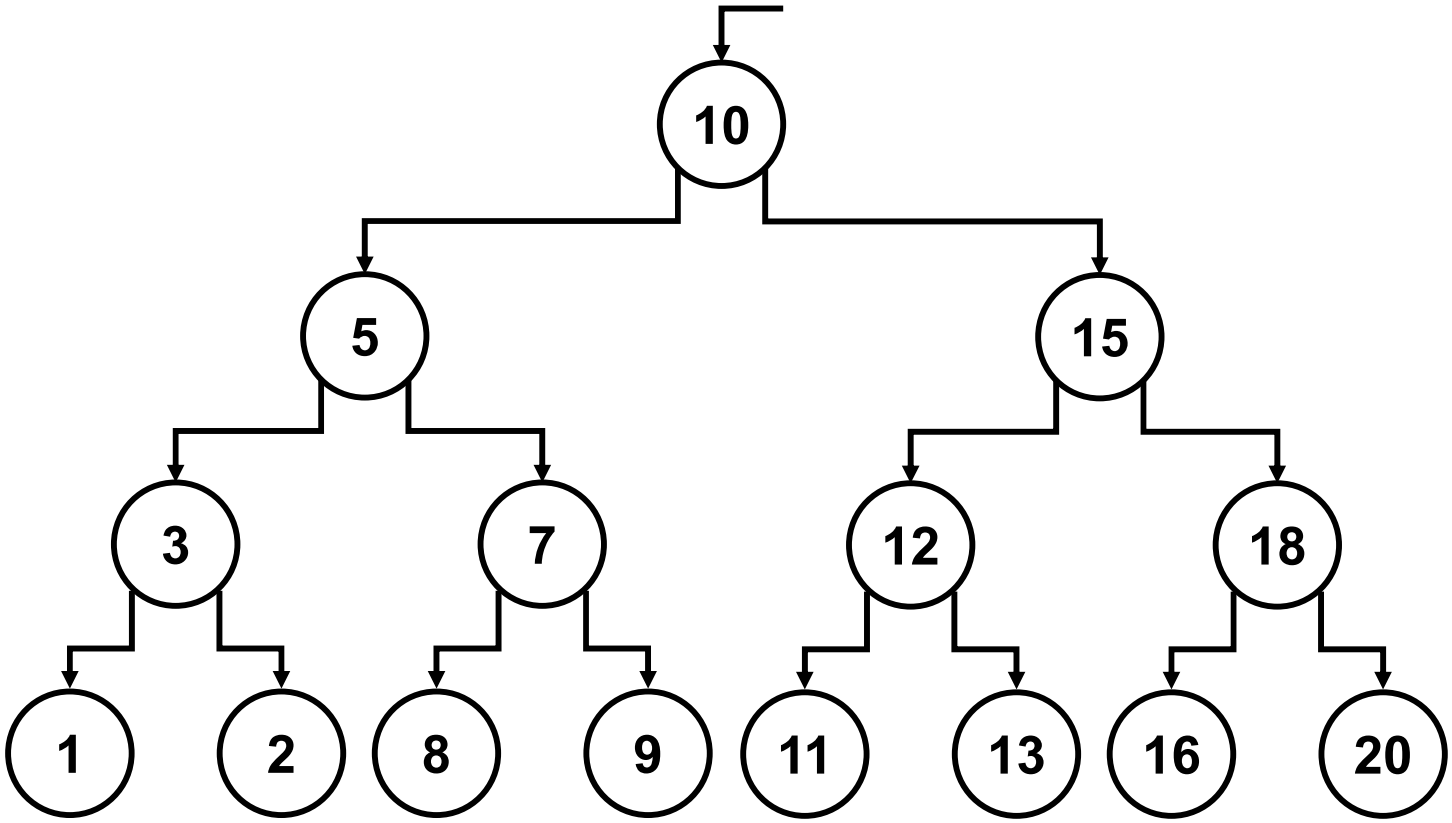


[1] *En matemáticas, una arborescencia es un árbol libre (grafo conexo sin ciclos) con raíz.*

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**.



$n = 2^h - 1$

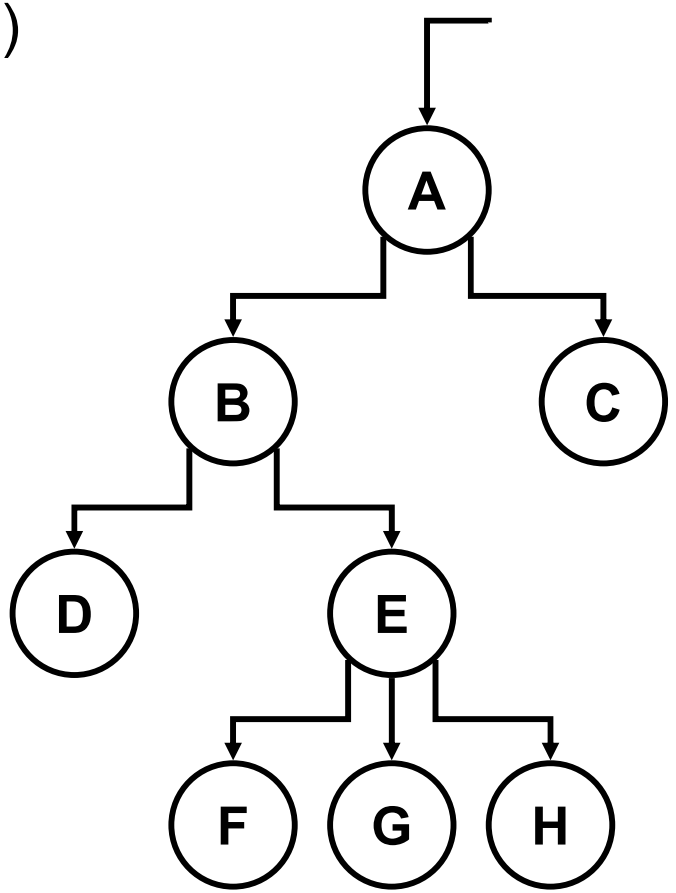
$\log_2(n + 1) = h$

DEBATE: ¿Cuál es la longitud máxima de un camino de búsqueda para n?

Métricas de Eficiencia

Caminos de Búsqueda

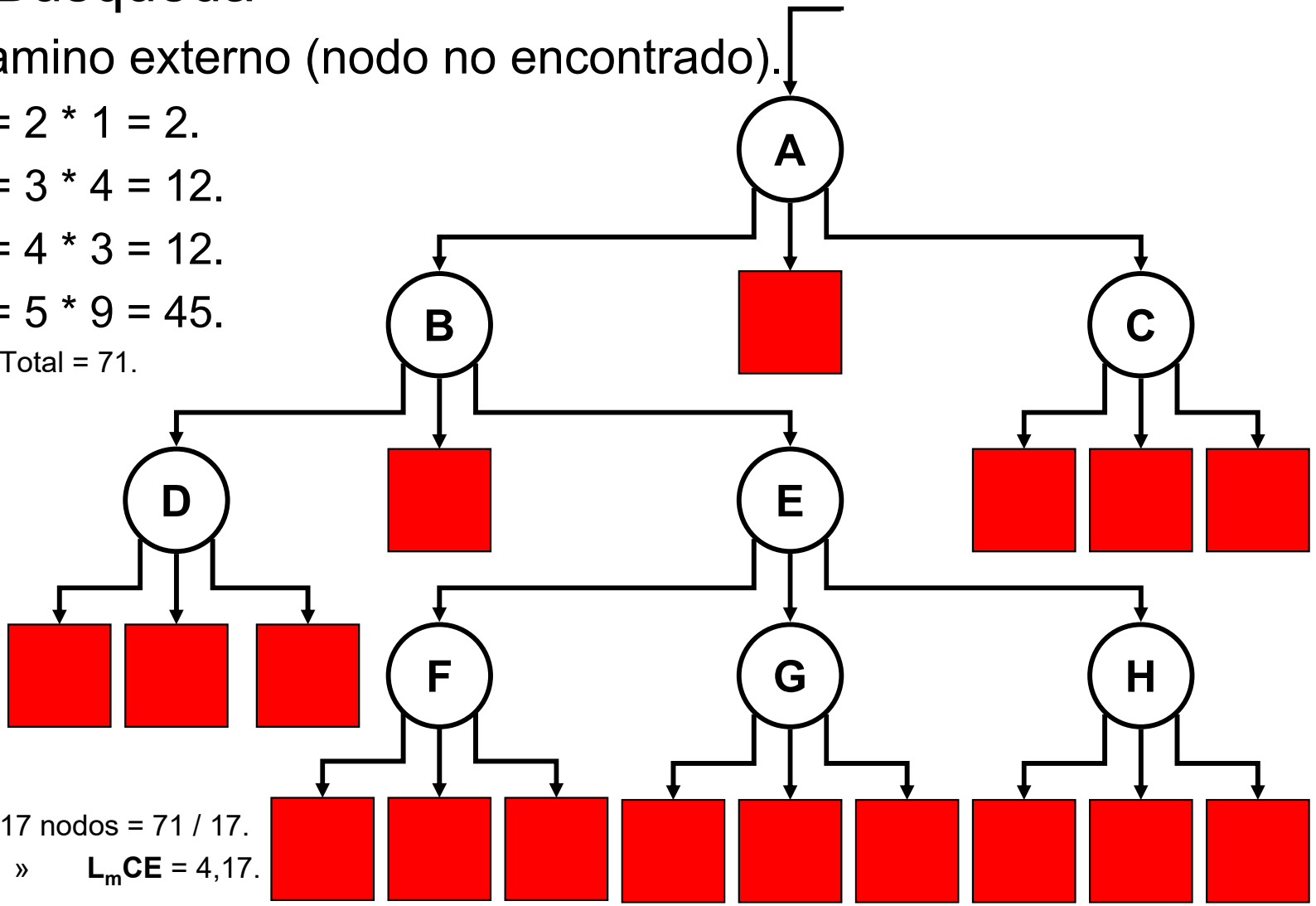
- CI: Camino Interno (nodo encontrado)
  - Buscar A = 1.
  - Buscar B y C = 2 c/u = 4.
  - Buscar D y E = 3 c/u = 6.
  - Buscar F, G y H = 4 c/u = 12.
    - Total = 23.
      - » Para 8 nodos = 23 / 8 = **L_mCI = 2,87**



Métricas de Eficiencia

Caminos de Búsqueda

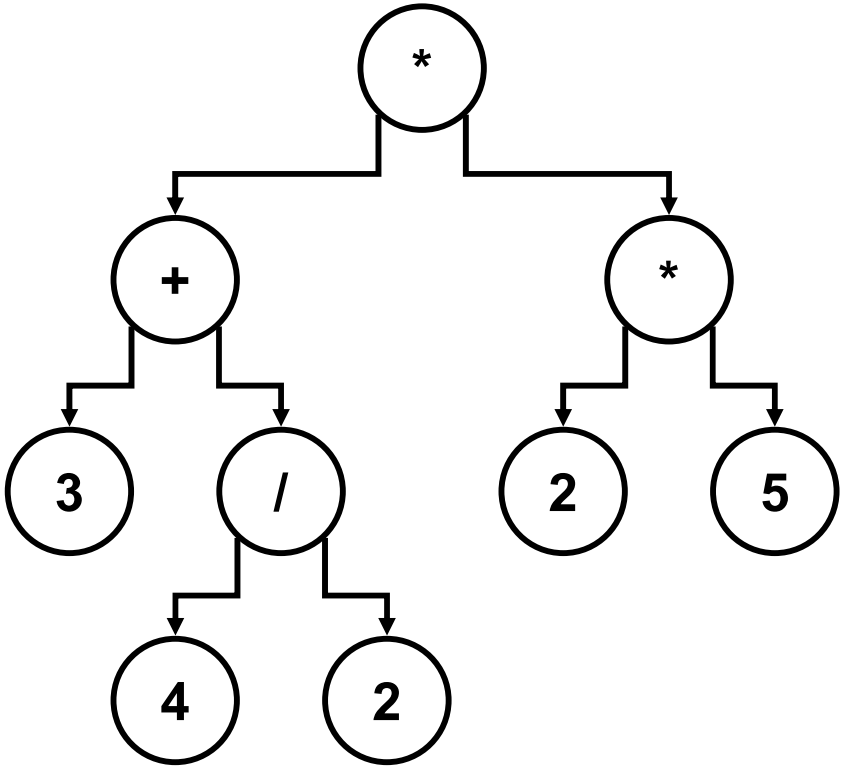
- CE: Camino externo (nodo no encontrado).
  - $N_2 = 2 * 1 = 2$.
  - $N_3 = 3 * 4 = 12$.
  - $N_4 = 4 * 3 = 12$.
  - $N_5 = 5 * 9 = 45$.
    - Total = 71.
    - 17 nodos = 71 / 17.
      - » **L_mCE** = 4,17.



Arbol Binario

Árbol de grado 2

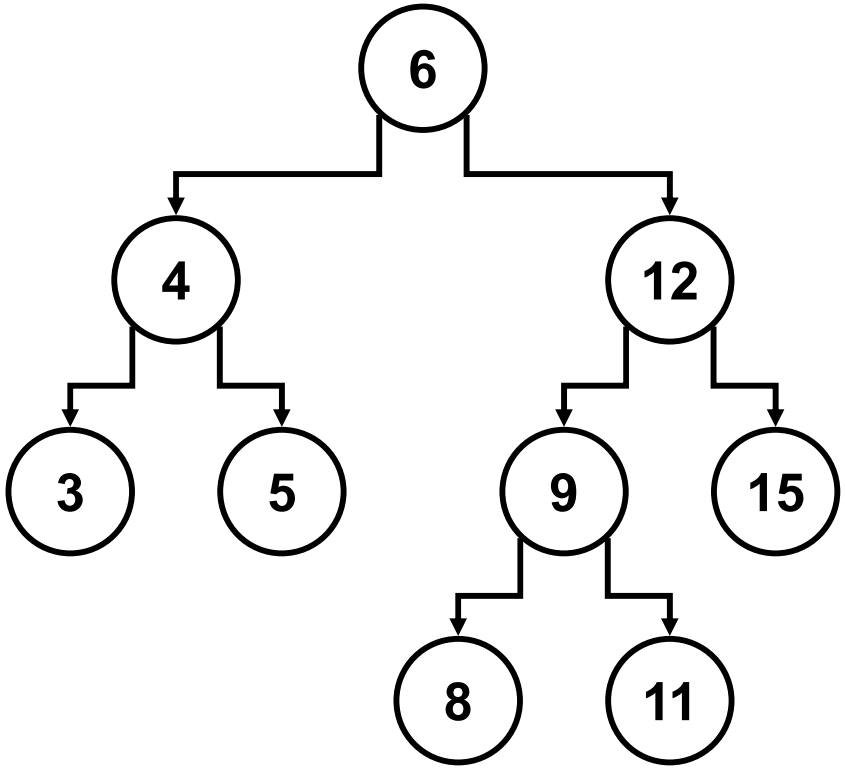
- Modela relaciones de jerarquía entre pares de elementos con respecto a un nodo superior
  - Árboles genealógicos.
  - Competiciones de copa.
  - Operadores binarios.



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)

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

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;
}
```

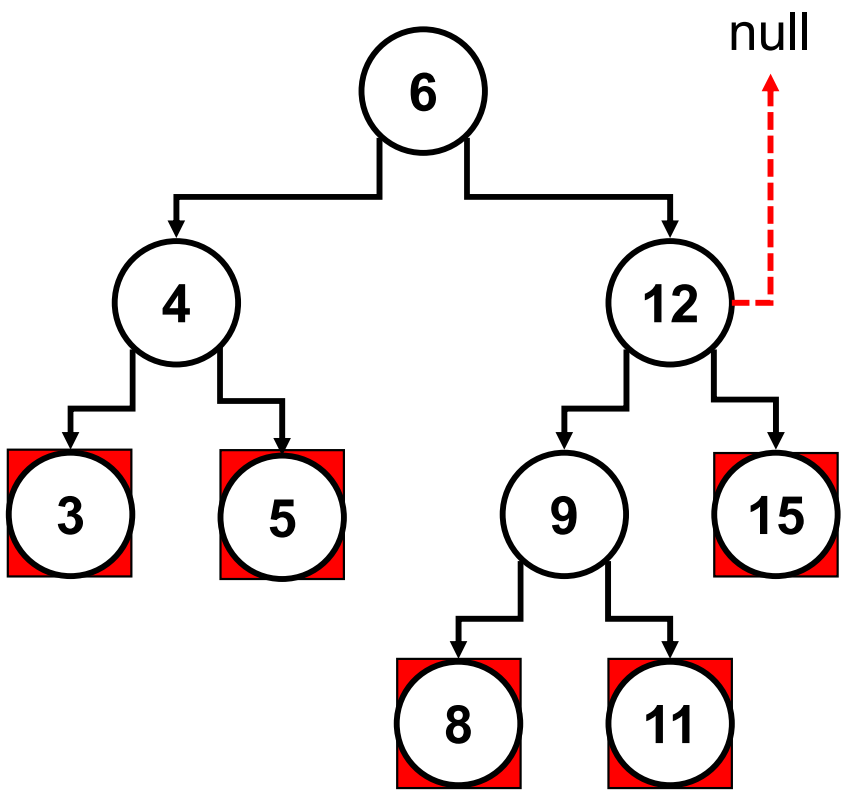
Árbol Binario de Búsqueda

Casos particulares del borrado

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

ZOOM IN

```
else {

 if (theRoot.getLeft() == null &&
  theRoot.getRight() == null)
   return(null);




}
```



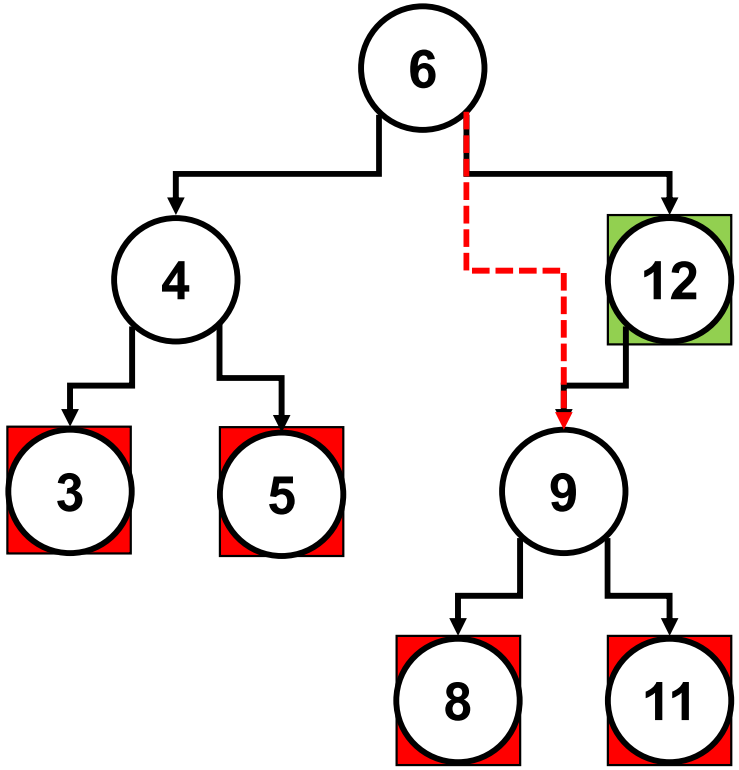
Árbol Binario de Búsqueda

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();



}
```



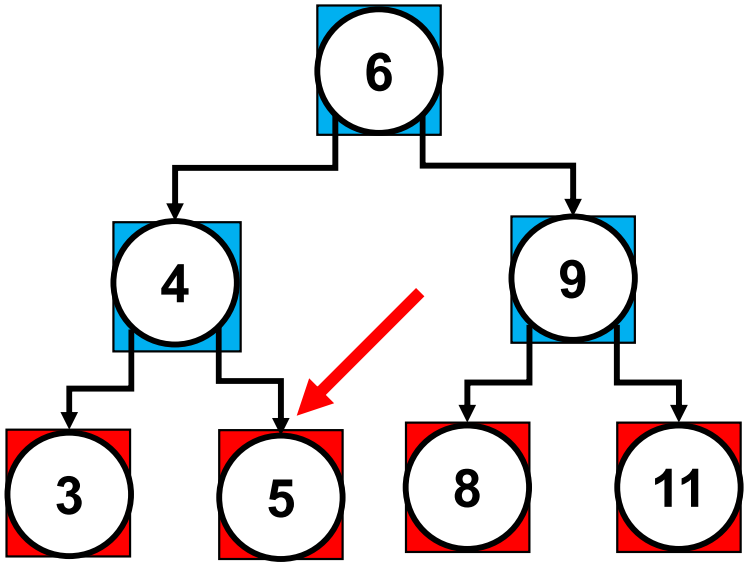
Árbol Binario de Búsqueda

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

```
public T getMax(BSTNode<T> theRoot)
{
 if (theRoot == null)
  return null;
 else
  return getMaxRec(theRoot);
}


private T getMaxRec(BSTNode<T> theRoot)
{
  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();
}
```

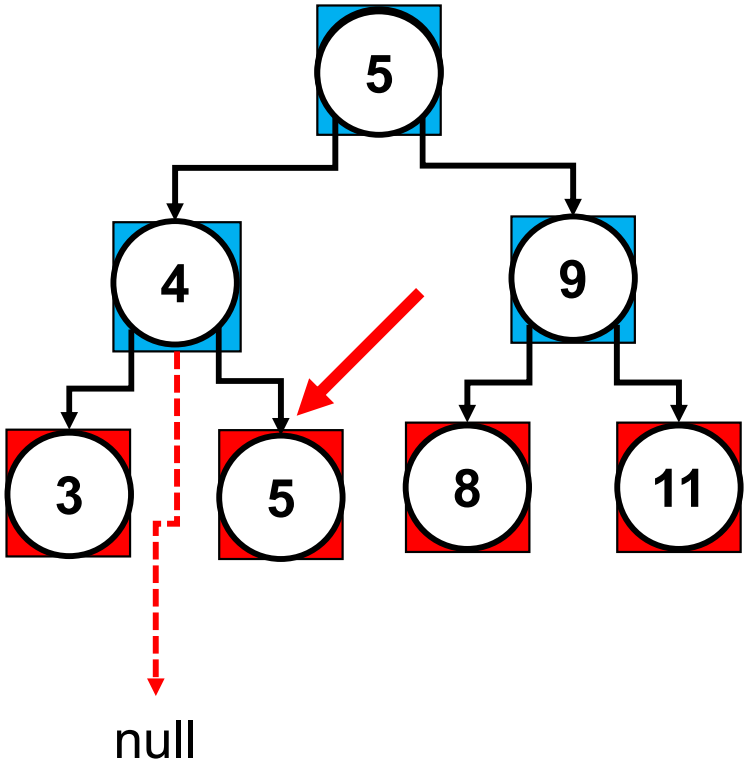
Árbol Binario de Búsqueda

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()));
  }
}
```



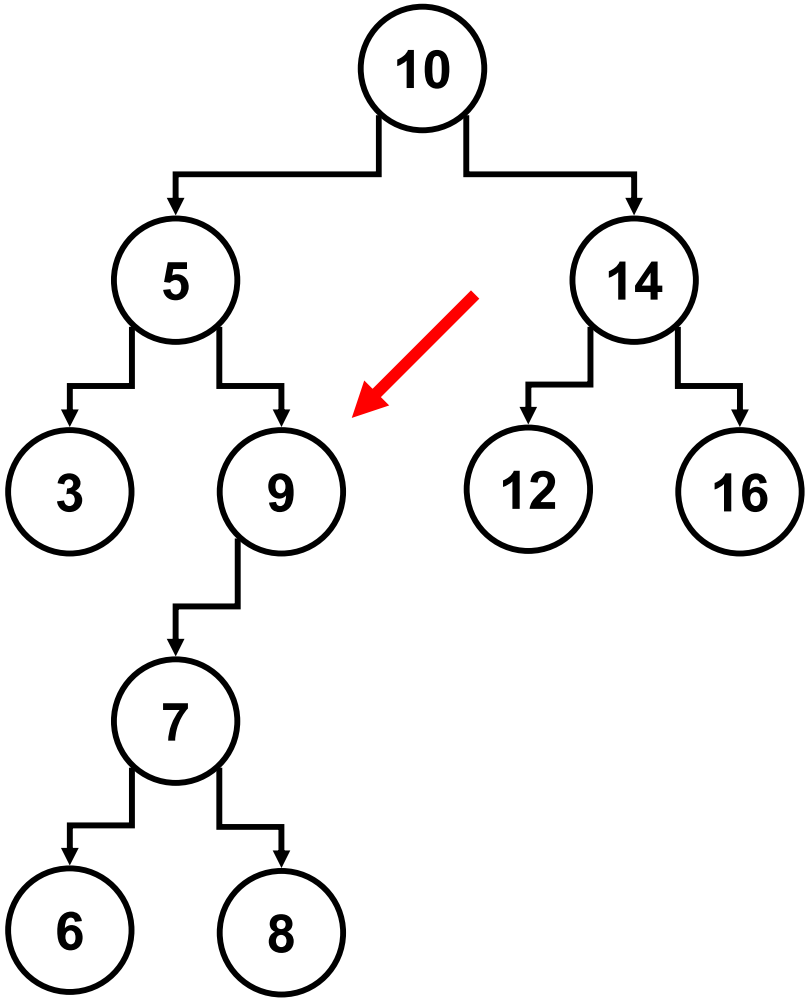
Árbol Binario de Búsqueda

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.getLef
t()));
 theRoot.setLeft(remove
(theRoot.getLeft(),theRoot.getElement())
);}}
```



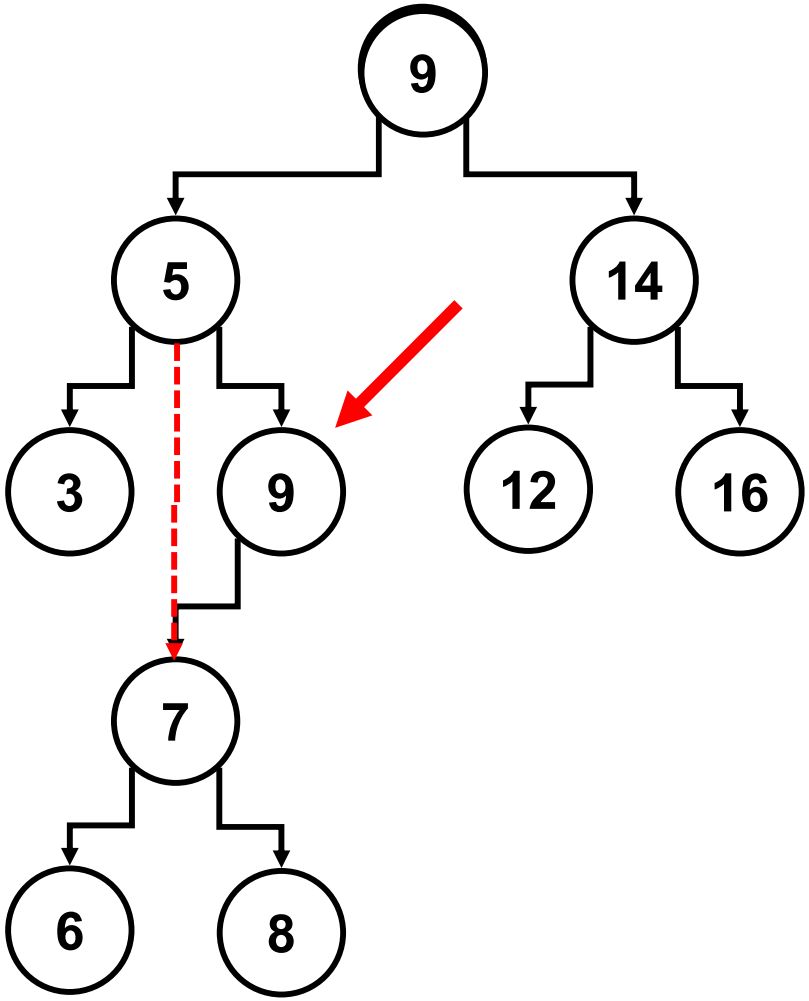
Árbol Binario de Búsqueda

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)

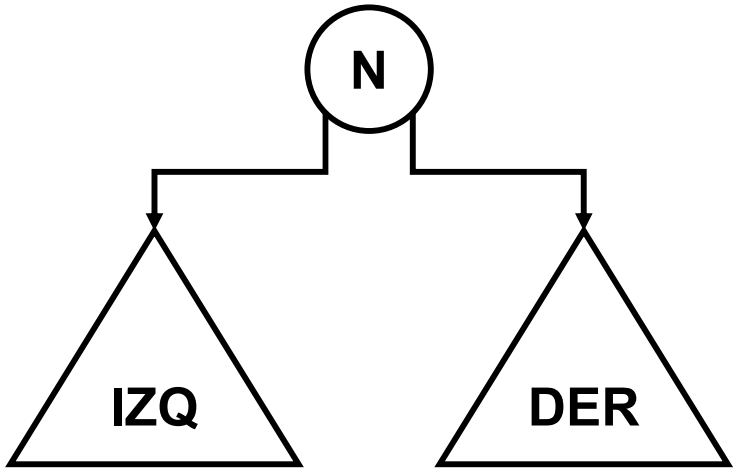
Complejidad caso peor: O(n)

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Árbol Binario de Búsqueda

Recorrido de un Árbol Binario

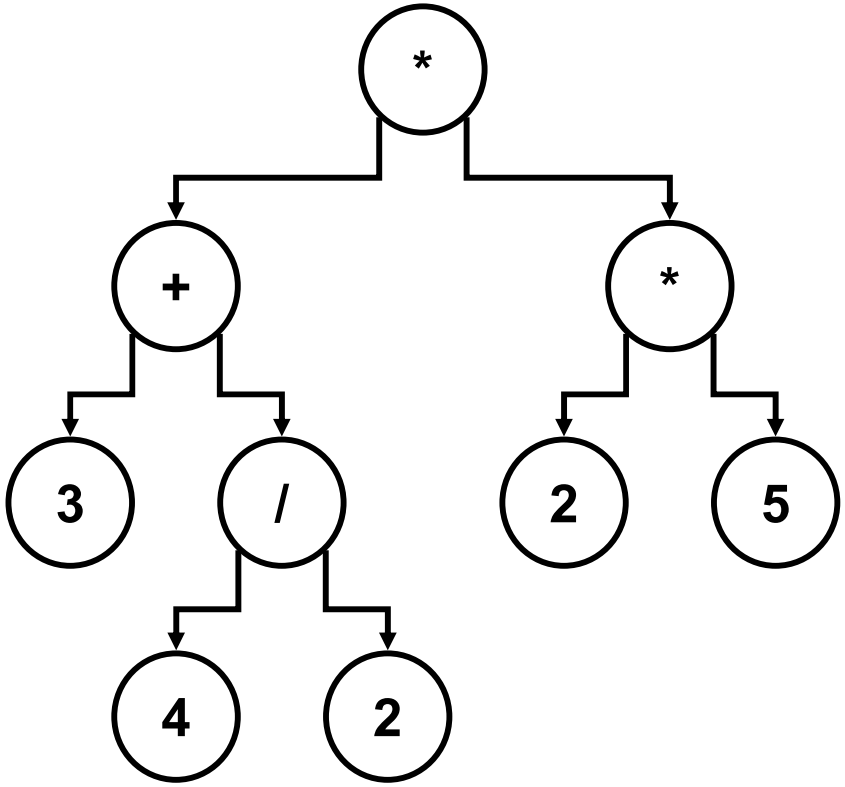
- **preorden**
  - **Primero** se analiza el nodo, luego subárboles.
    - N-IZQ-DER ó N-DER-IZQ.
- **inorden**
  - El nodo se analiza **entre** los dos subárboles.
    - IZQ-N-DER ó DER-N-IZQ.
- **postorden**
  - El nodo se analiza **después** de los subárboles.
    - IZQ-DER-N ó DER-IZQ-N.



Árbol Binario de Búsqueda

Ejercicio

- Recorrer el árbol

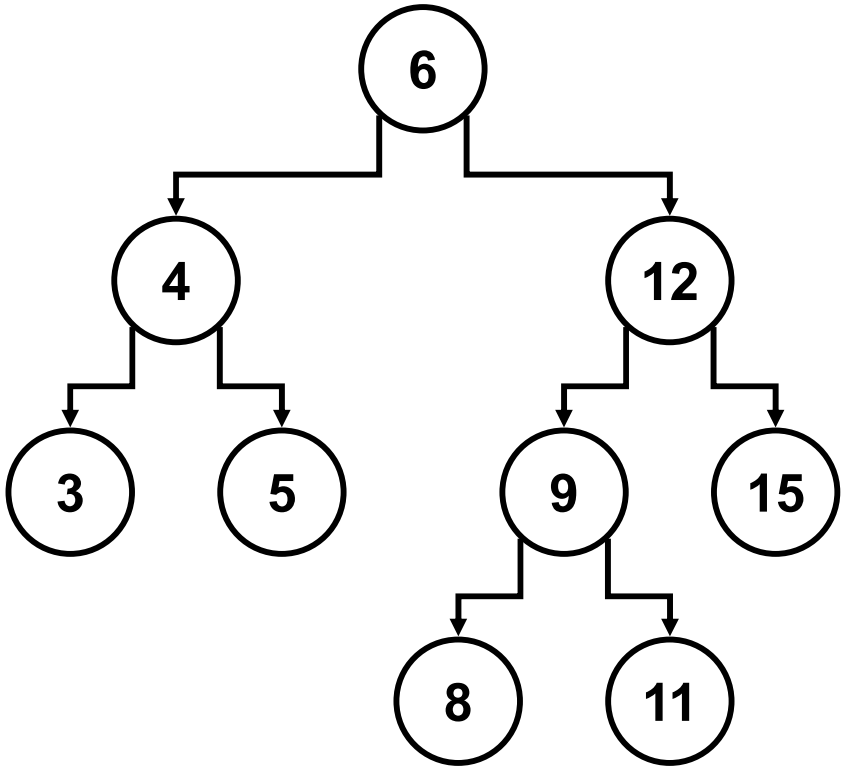


- **preorden**: * + 3 / 4 2 * 2 5 (prefija).
- **inorden**: 3 + 4 / 2 * 2 * 5 (infija).
- **postorden**: 3 4 2 / + 2 5 * * (polaca inversa)

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 (postfija).

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)

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Árbol Binario de Búsqueda

Eficiencia de un árbol binario de búsqueda

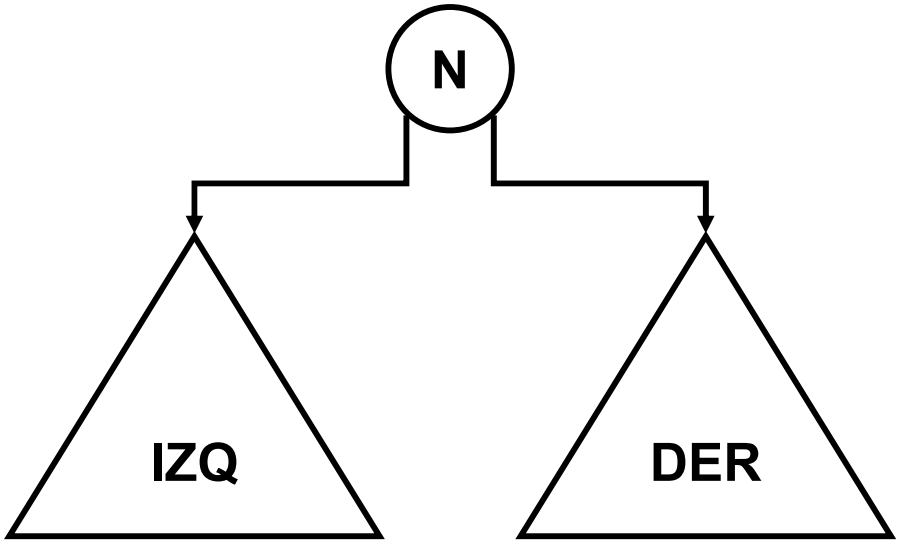
- La eficiencia de este tipo de árboles depende ampliamente de su altura
  - Rango de h: $[\log_2 n, n]$

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.

Árboles Perfectamente Equilibrados (APE)

- Garantiza **altura** mínima en un árbol binario
  - **Condición**: Para todo nodo n, $|\#_{izq} - \#_{der}| <= 1$.
    - $\#_{izq}$ = número de nodos del subárbol izquierdo.
    - $\#_{der}$ = 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.

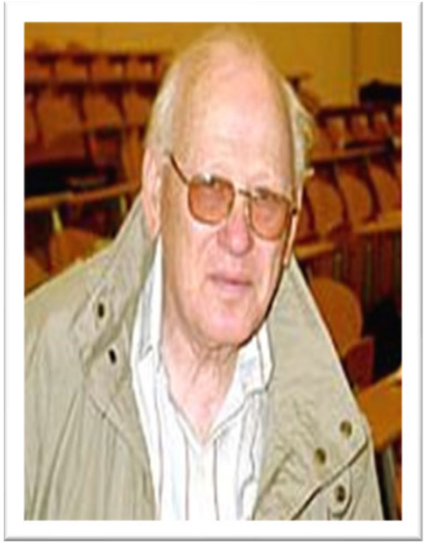
Método	ABB (caso peor)	APE (cualquier caso)
Insertar	O(n)	O(n)
Buscar	O(n)	$O(\log_2 n)$
Borrar	O(n)	O(n)

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

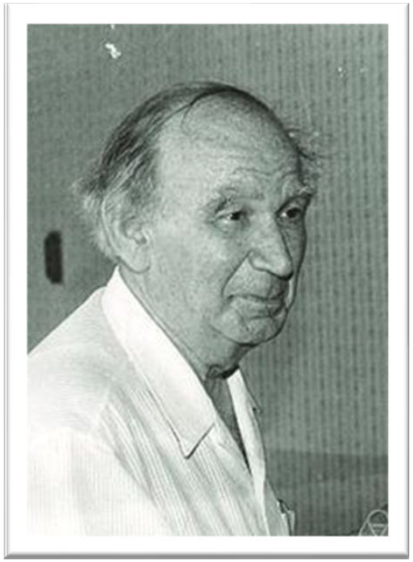
Árboles AVL

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.



Georgii Adelson-Velskii (Wikipedia)



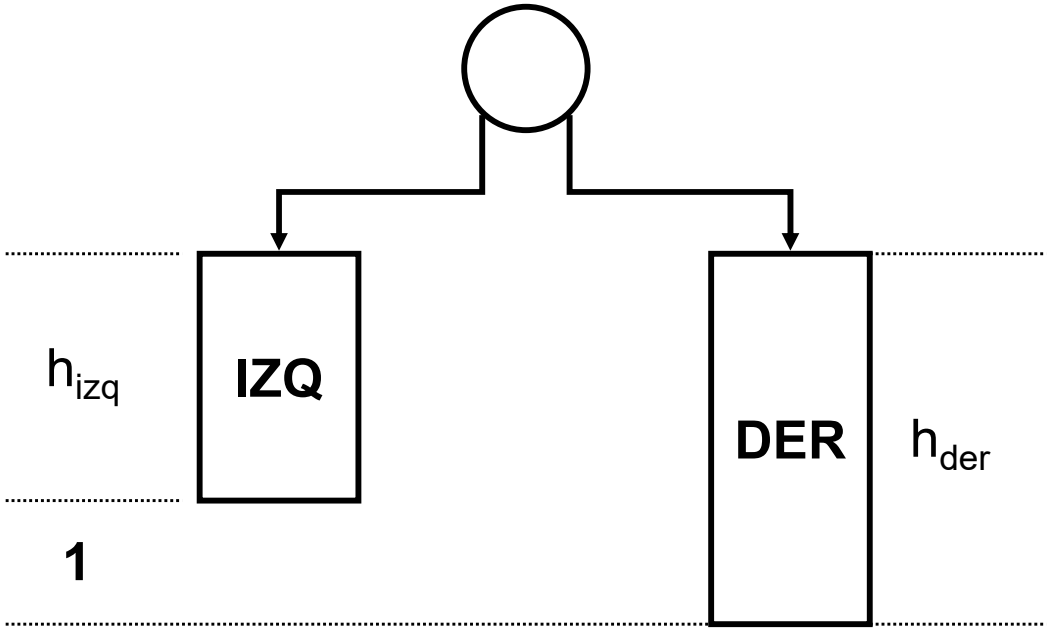
Yevgeni Landis (Wikipedia)

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

Árboles AVL

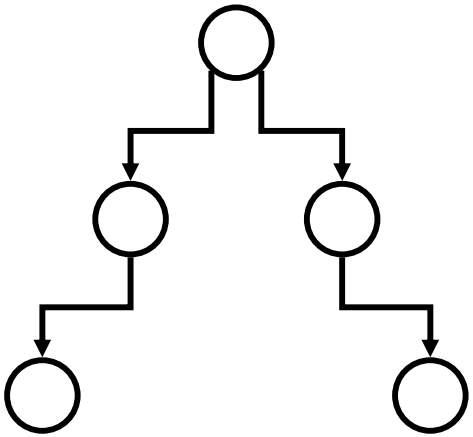
Árboles de Adelson-Velskii y Landis

- AKA Árboles Simplemente Equilibrados
  - **Condición**: Para todo nodo n, $|h_{izq} - h_{der}| <= 1$.
    - h_{izq} = altura del subárbol izquierdo.
    - h_{der}= altura del subárbol derecho.

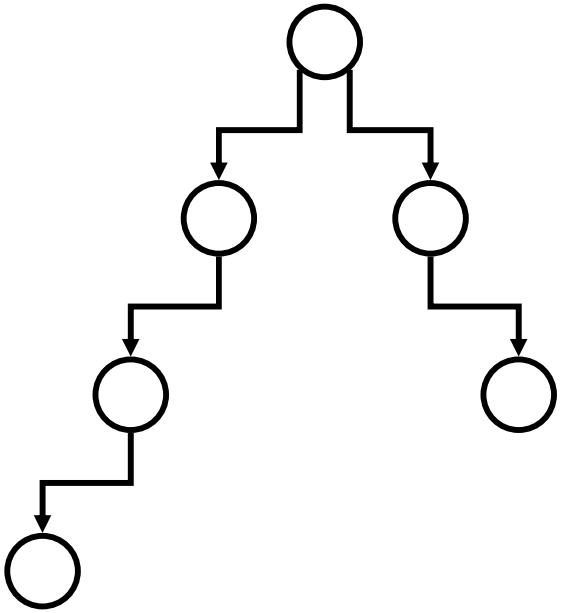


Árboles AVL

Ejemplos



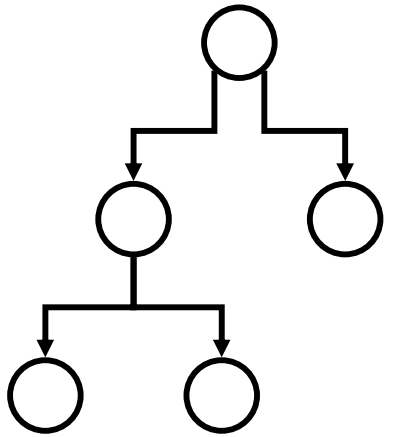
¿AVL? ..... Sí
¿APE? ..... Sí
¿Altura Mínima? ..... Sí



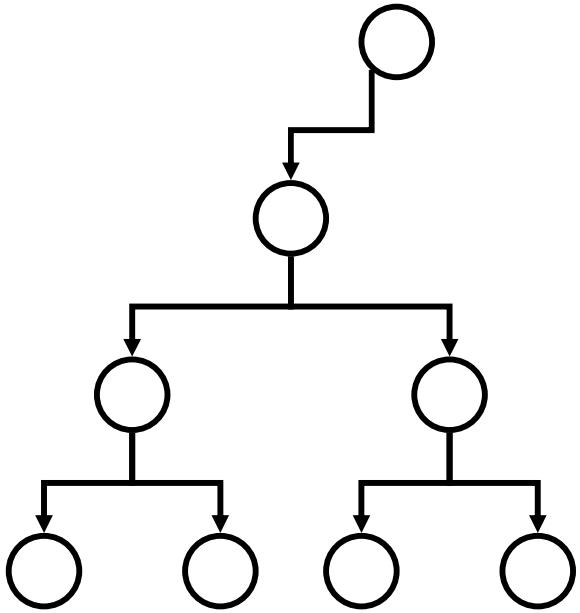
¿AVL? ..... No
¿APE? ..... No
¿Altura Mínima? ..... No

Árboles AVL

Ejemplos



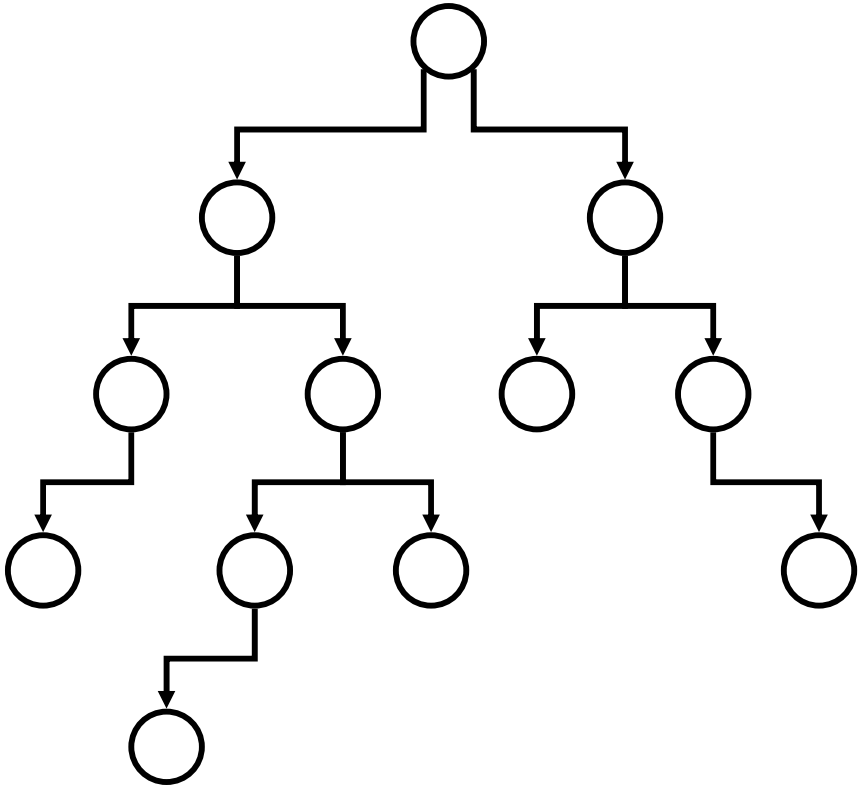
¿AVL? ..... Sí
¿APE? ..... No
¿Altura Mínima? ..... Sí



¿AVL? ..... No
¿APE? ..... No
¿Altura Mínima? ..... Sí

Árboles AVL

Ejemplos



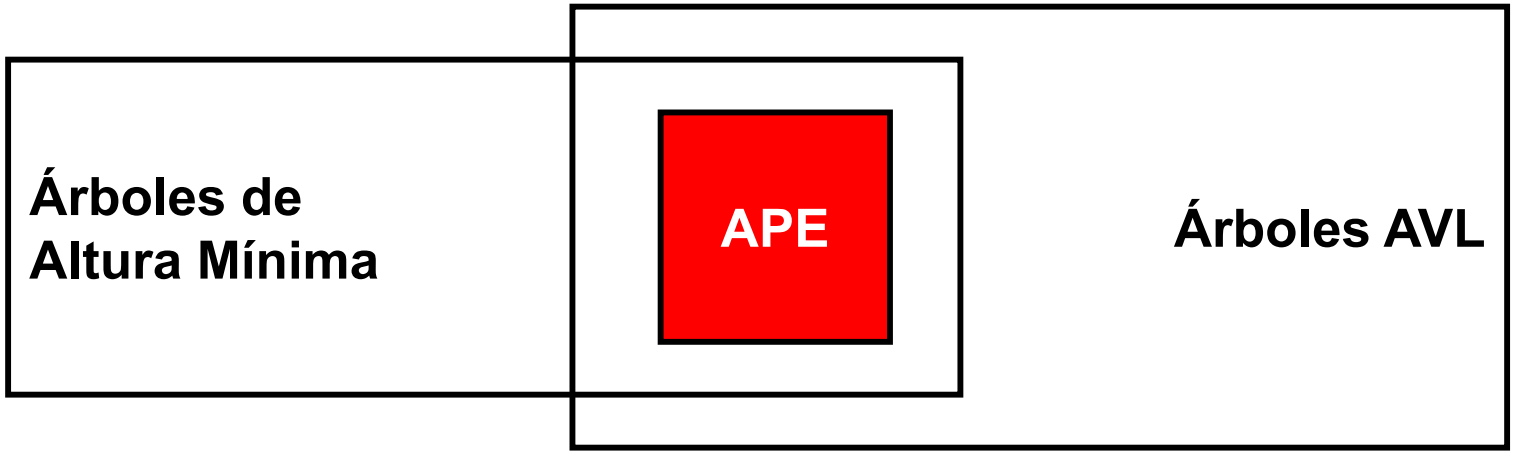
¿AVL? ..... Sí
¿APE? ..... No
¿Altura Mínima? ..... No

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 construidos de la peor manera posible para alcanzar una altura máxima respetando la condición de AVL.

Árboles AVL

Árboles de Fibonacci

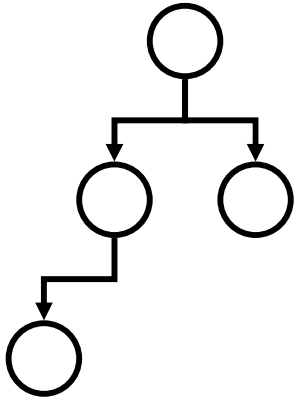
- Se decide la altura (a) de antemano.
  - Para **h = 0**, usar árbol vacío (T_0).
  - Para **h = 1**, usar (T_1), o árbol de un solo nodo.
  - Para **h > 1**, usar $T_h = (T_{h-1}, x, T_{h-2})$.

a = 0

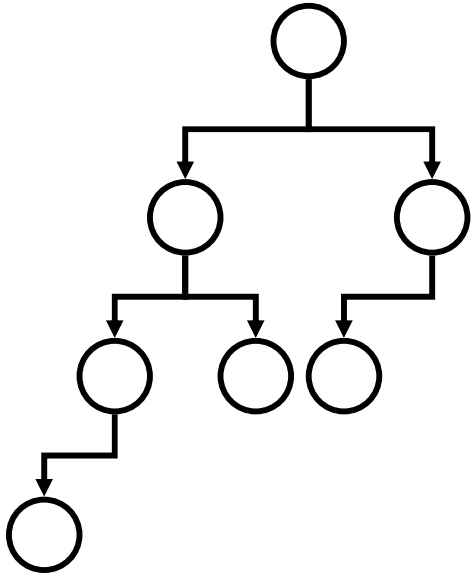
a = 1

a = 2

a = 3



a = 4



Árboles AVL

Adelson-Velskii y Landis demostraron…

Cota para la altura máxima de un árbol de Fibonacci

$h_{MaxFib}(n) <= 1,44 Log_2 n$

Rango de altura para un AVL

$h_{APE}(n) <= h_{AVL}(n) <= h_{MaxFib}(n)$

$Log_2 n <= h_{AVL}(n) <= 1,44 Log_2 n$

- 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

$O(Log_2 n) <= O(h_{AVL}(n)) <= O(1,44 Log_2 n)$

$O(1,44 Log_2 n)$

$O(Log_2 n)$

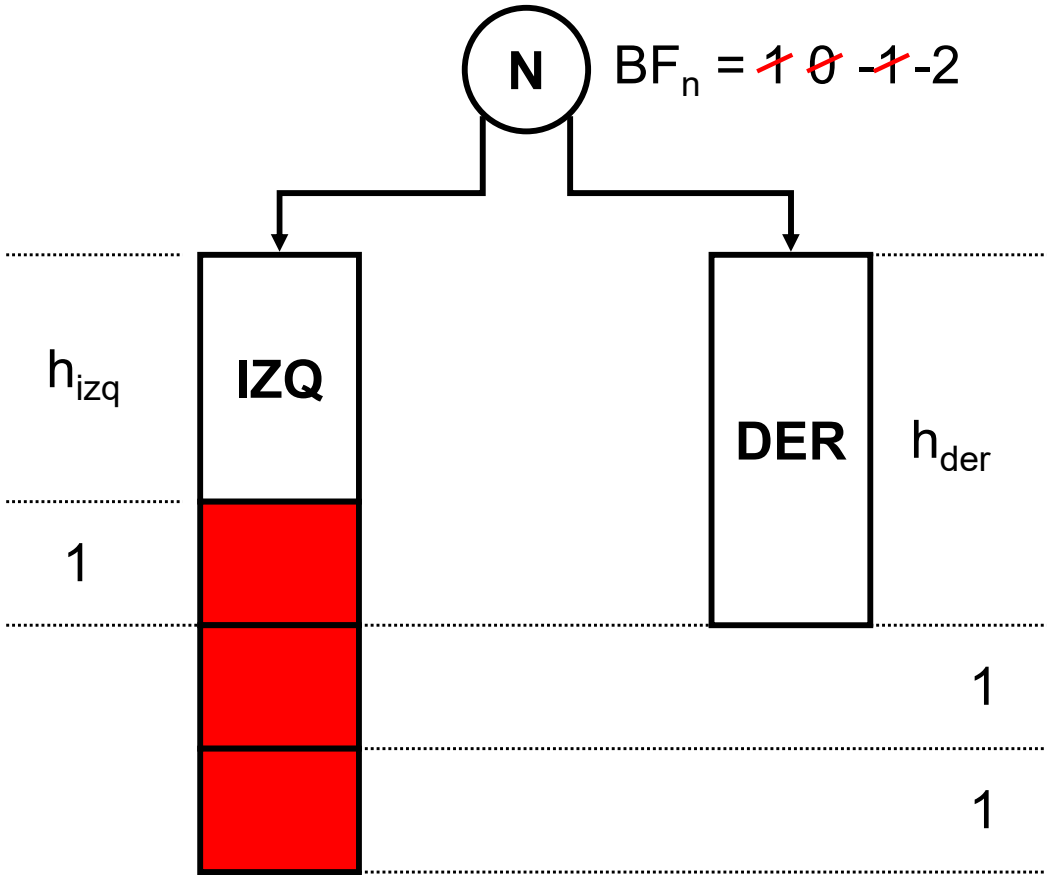
Árboles AVL

Balance Factor (BF)

- $BF_n = h_{der} - h_{izq}$

- Situaciones de Partida:
  - $h_{izq} > h_{der}$ ($BF_n = -1$).
  - $h_{izq} = h_{der}$ ($BF_n = 0$).
  - $h_{izq} < h_{der}$ ($BF_n = 1$).

- 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

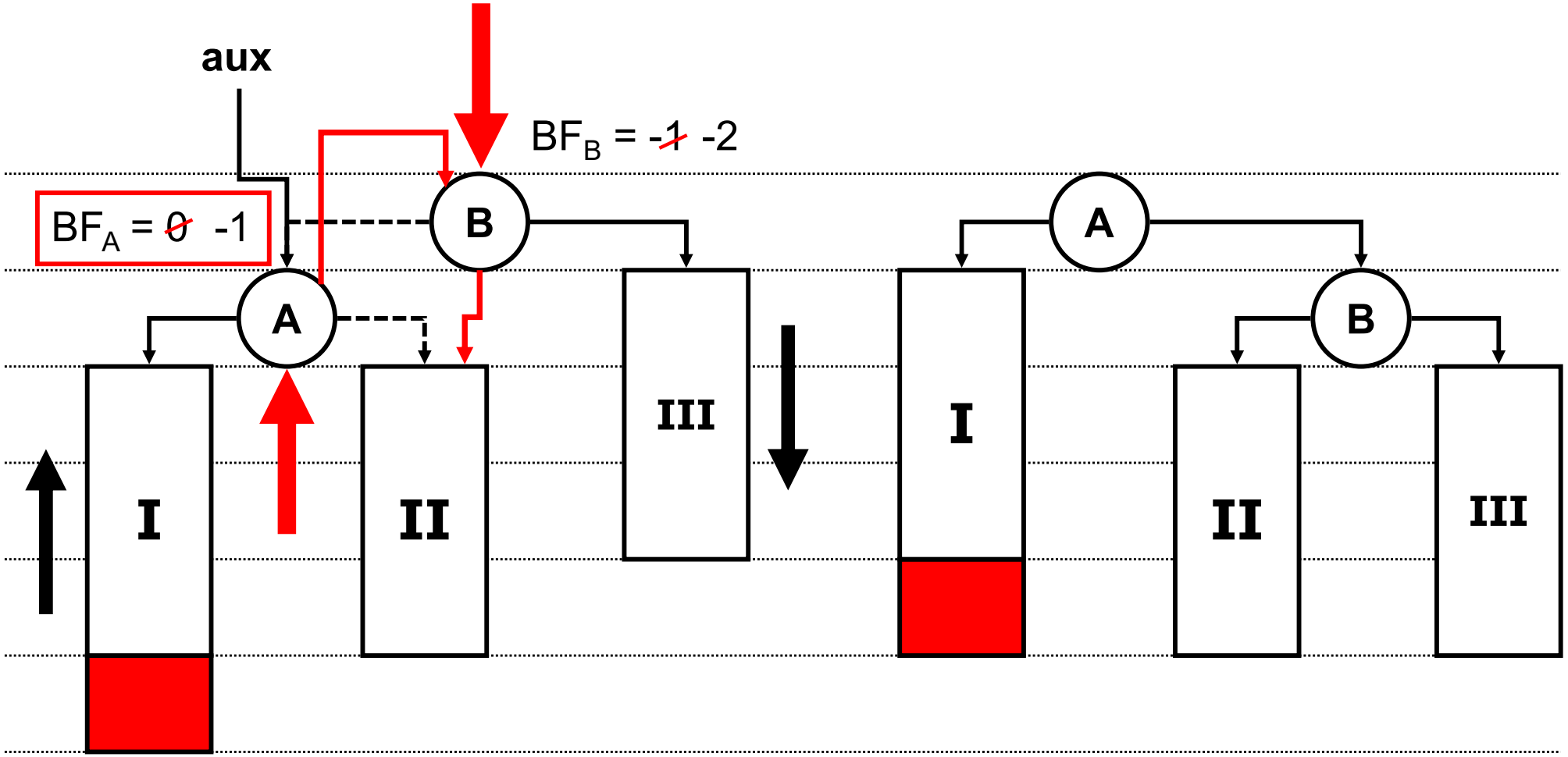
UpdateBF (pseudocódigo)

```
private AVLNode<T> updateBF (AVLNode<T> theRoot){

 if (theRoot.getBF() == -2)
 {
  if (theRoot.getLeft().getBF() <=0)
   theRoot = singleLeftRotation (theRoot);
  else
   theRoot = doubleLeftRotation (theRoot);
 }
 else if (theRoot.getBF() == 2)
 {
  if (theRoot.getRight().getBF() >= 0)
   theRoot = (singleRightRotation (theRoot));
  else
   theRoot = (doubleRightRotation (theRoot));
  }

  theRoot.updateHeight();
  return (theRoot);
}
```

Árboles AVL

Caso Ia

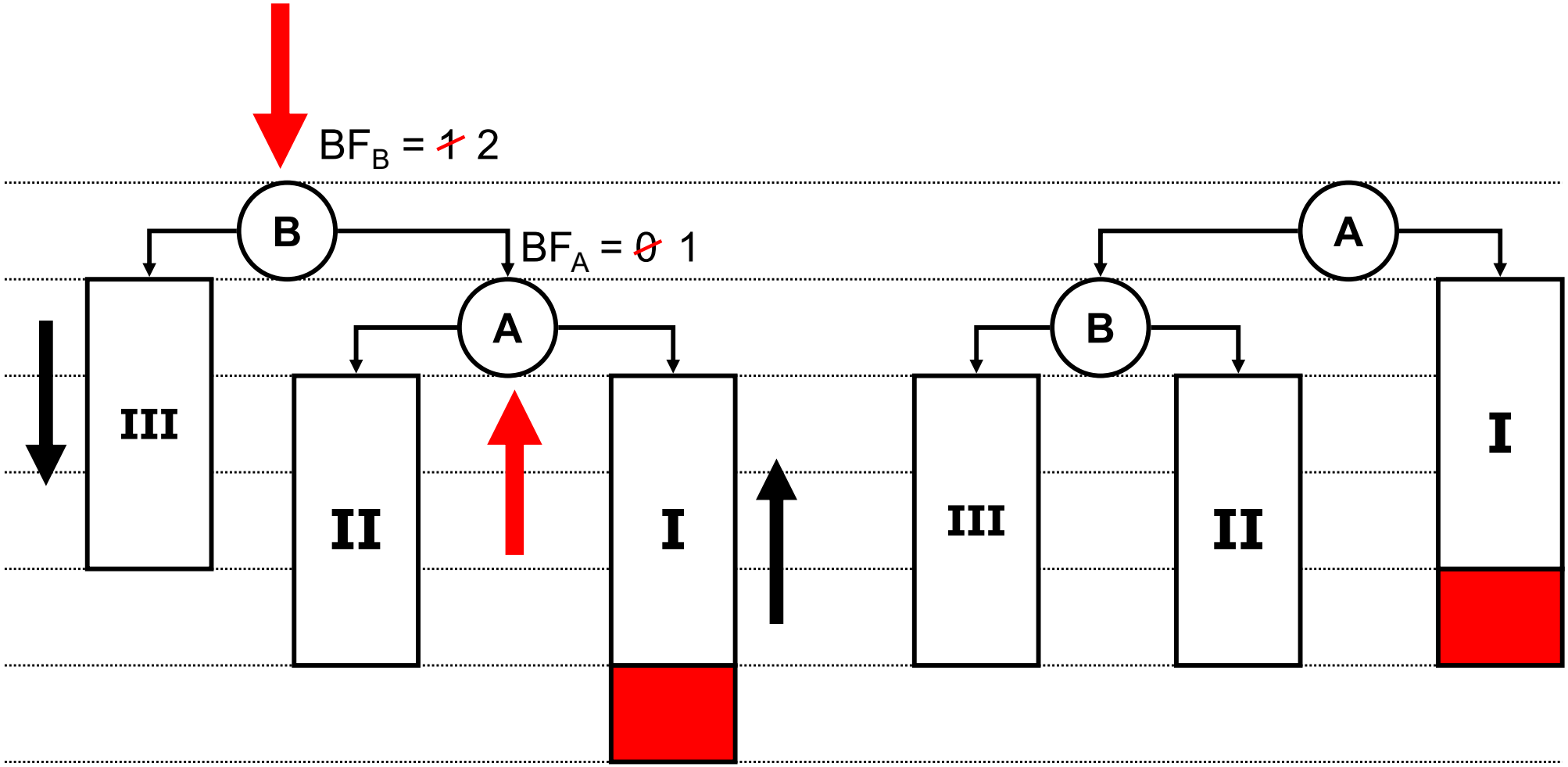
- **Rotación Simple** (izquierdo)



Árboles AVL

Caso Ib

- **Rotación Simple** (derecho)



CLASSWORK

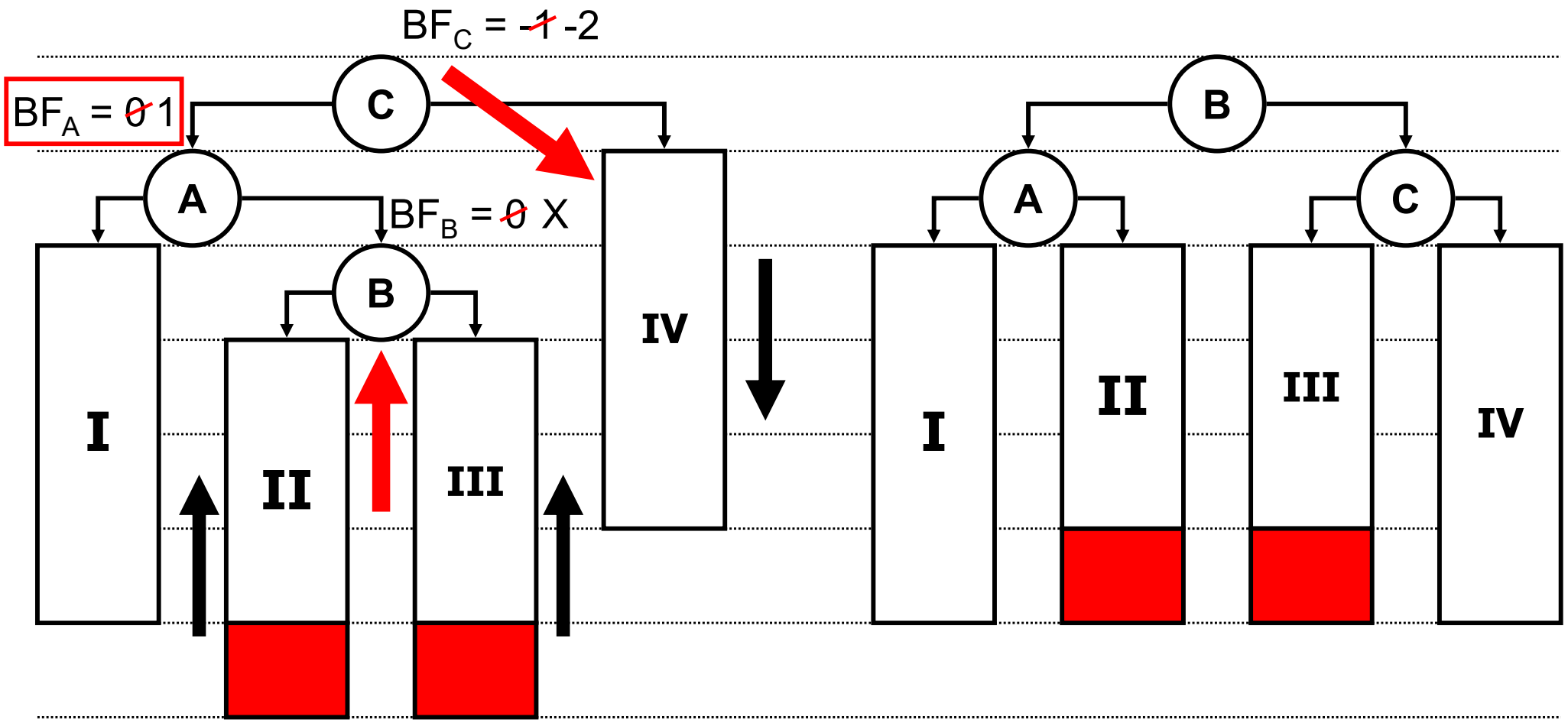
PLAYGROUND

- **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

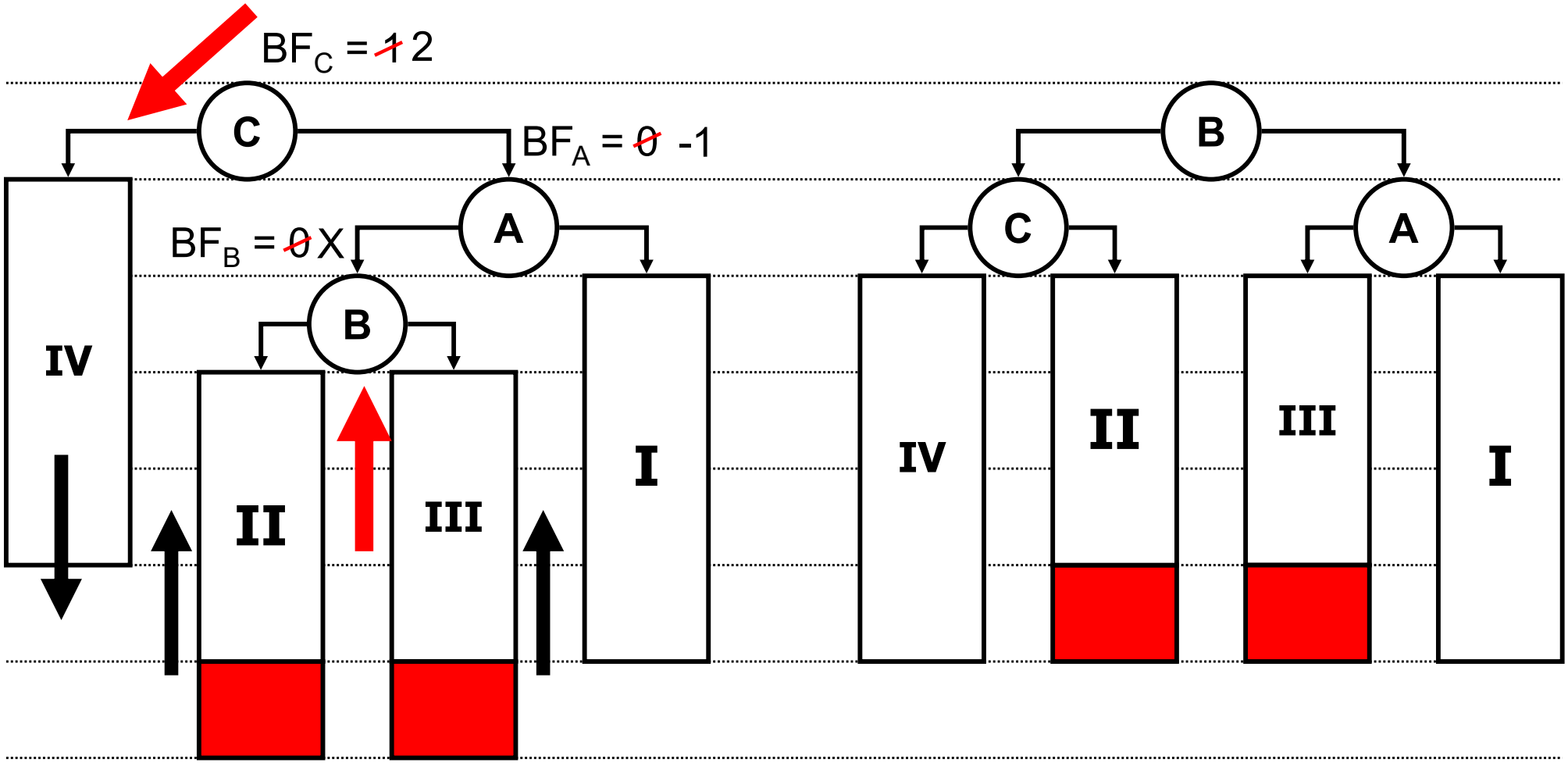
- **Rotación Doble** (izquierdo)



Árboles AVL

Caso IIb

- **Rotación Doble** (derecho)



CLASSWORK

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.

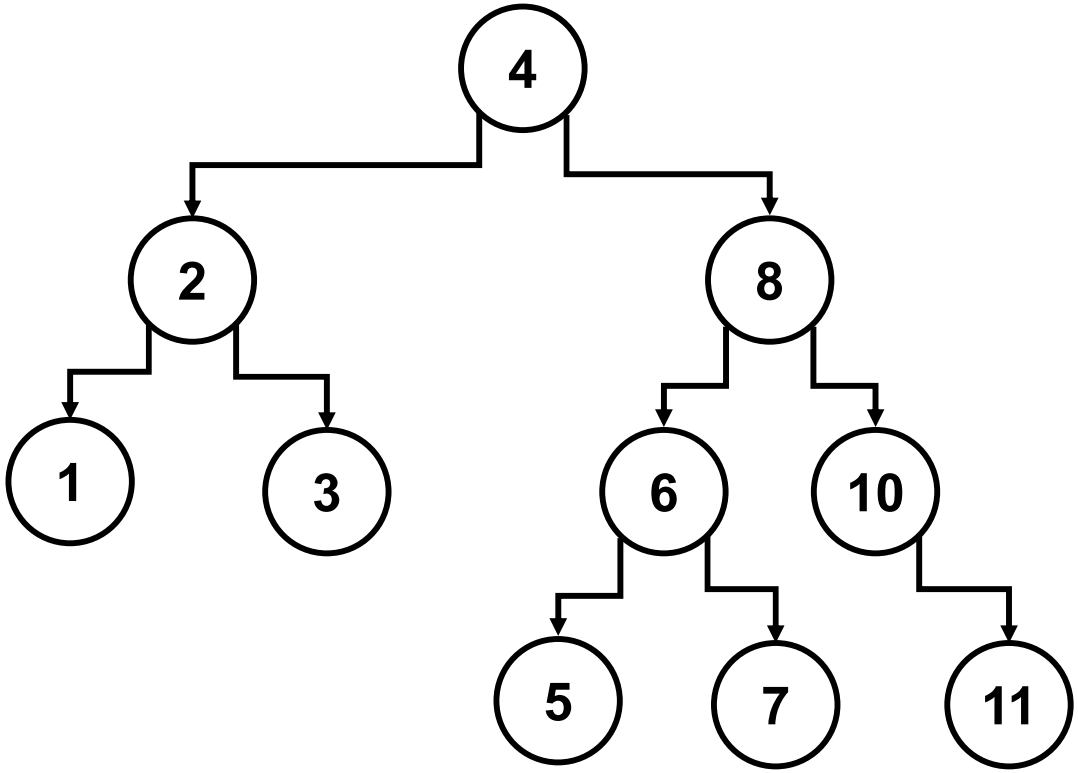
Árboles AVL

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));
}
```

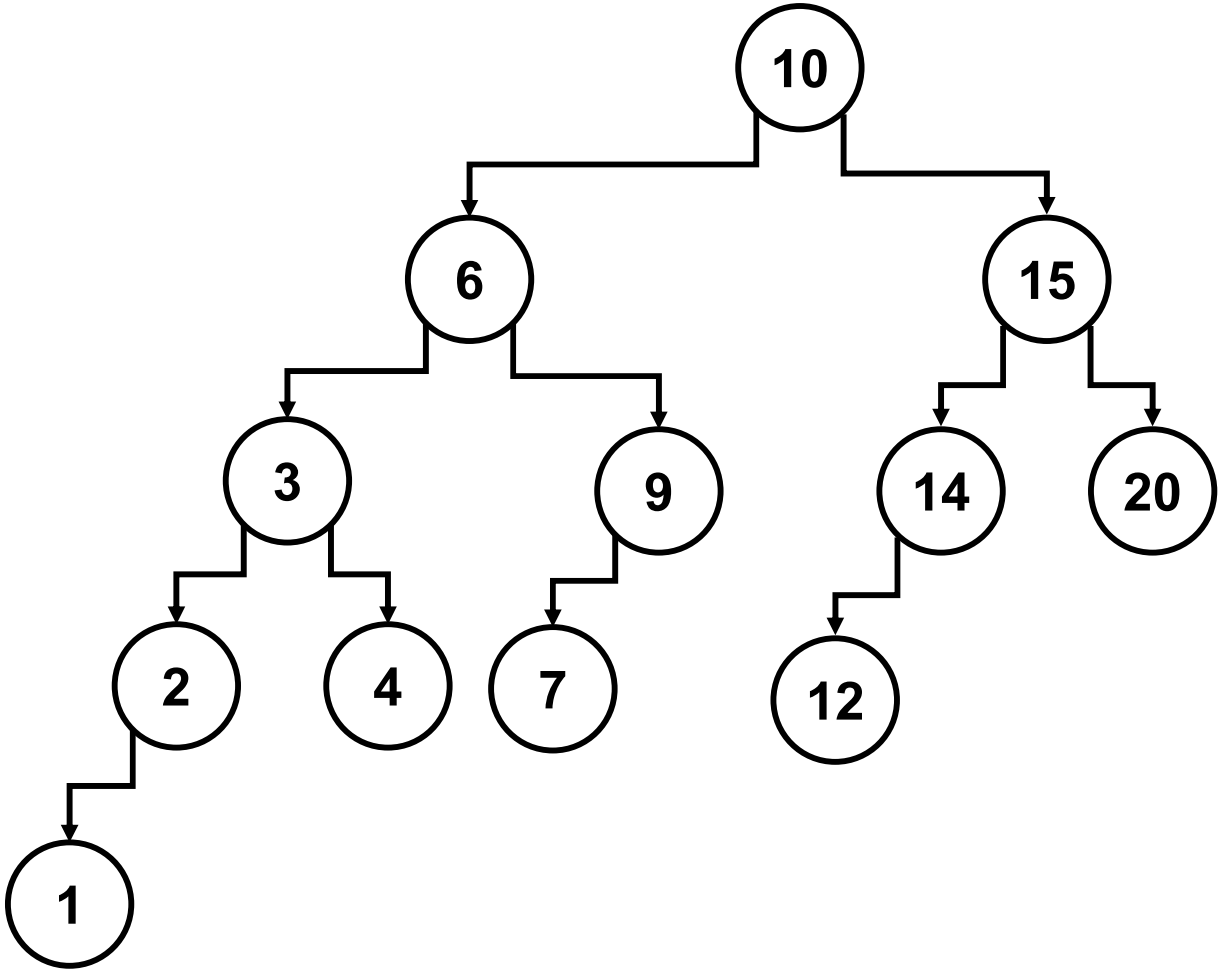
PLAYGROUND

- **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.



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 $\log_2 n$

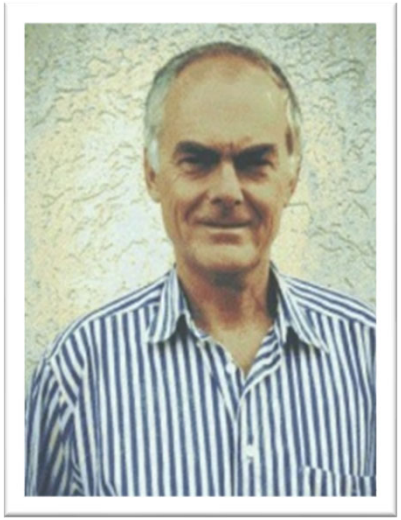
$\text{Log}_2 n <= \text{Longitud del Camino de Búsqueda} <= 1{,}44\text{Log}_2 n$

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

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.



Rudolf Bayer (Wikipedia)

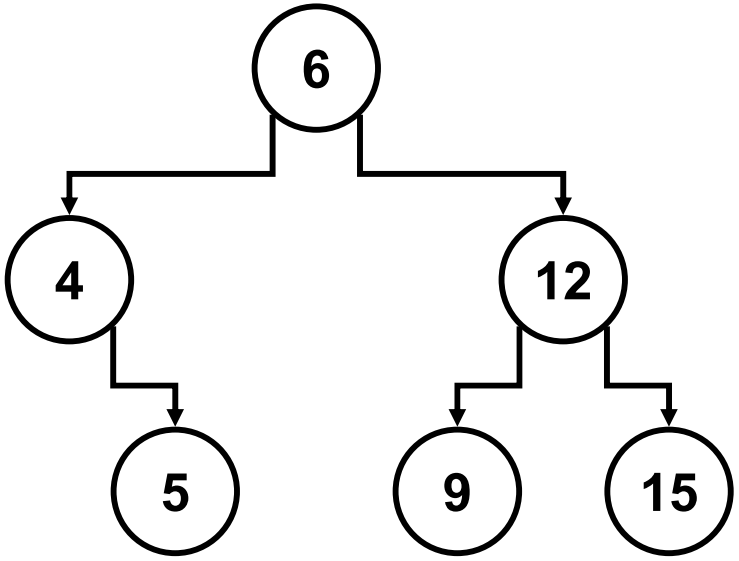
Edward M. McCreight (Wikipedia)

- 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

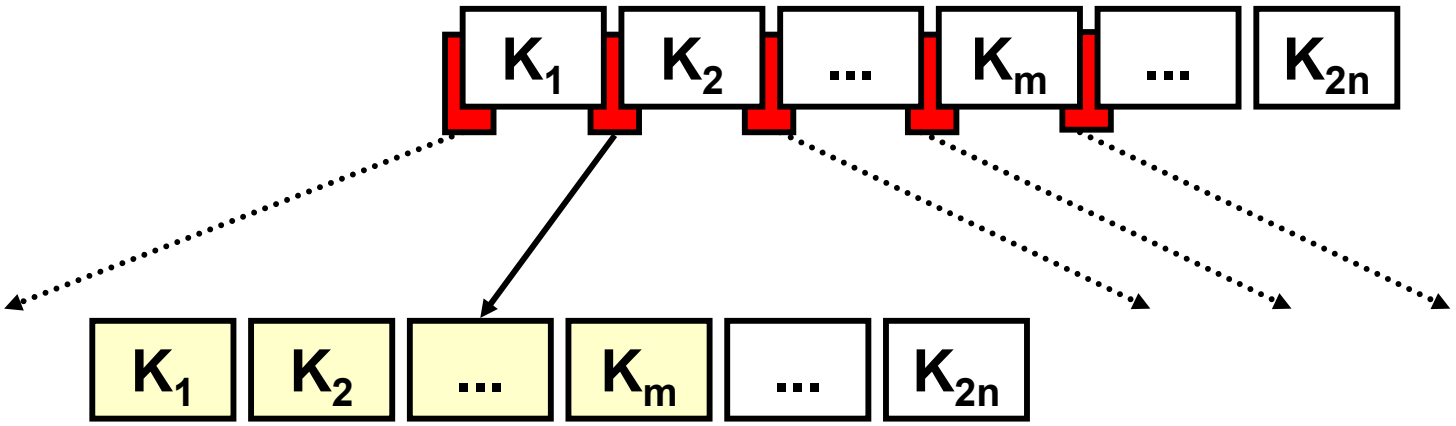
No encontrar un elemento en un árbol AVL de 1.000.000 elementos requiere…

… entre 20 y 28 accesos a disco

Árboles B (Bayer & McCreight)

Definición

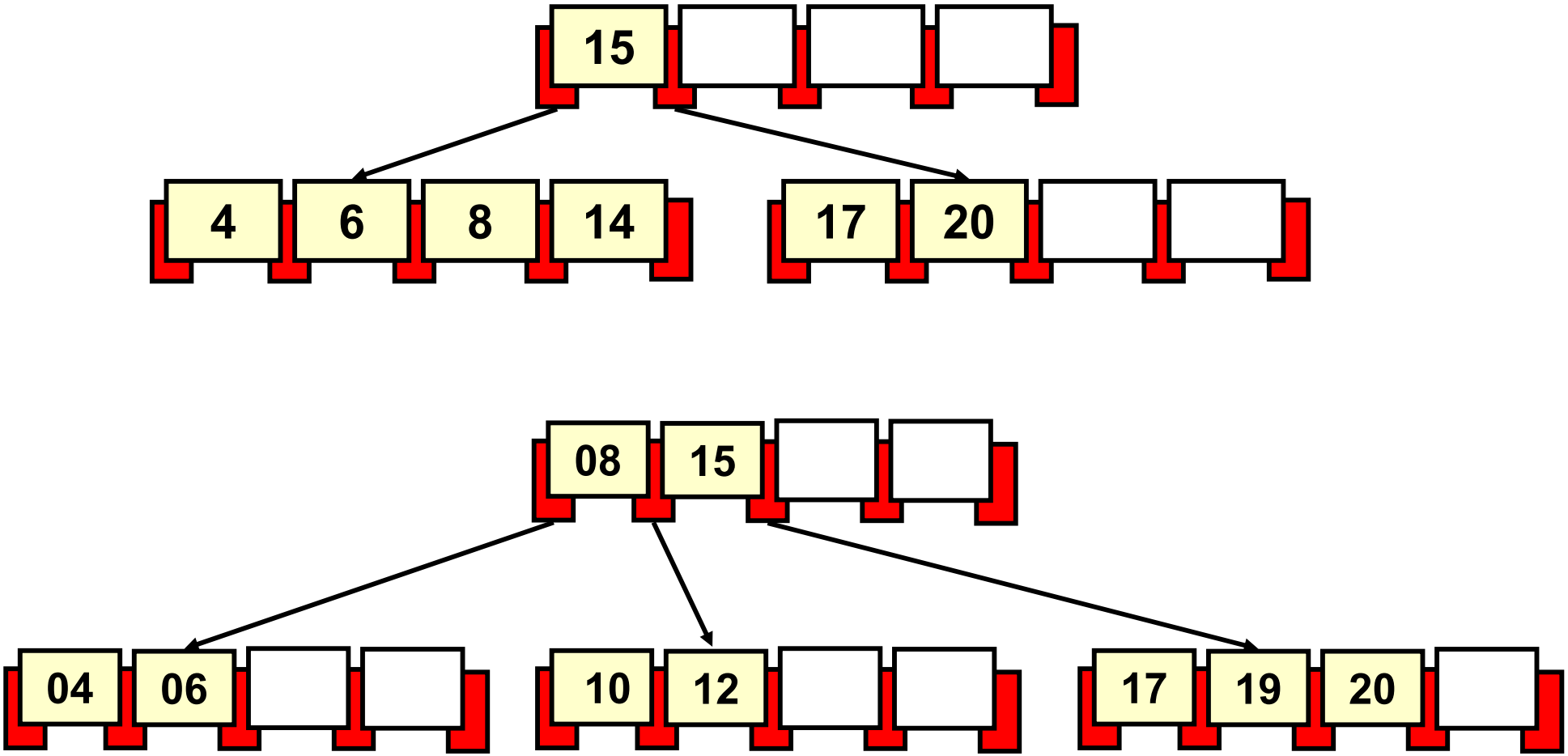
- Se define un **árbol B de orden n** (B-n) como un árbol donde
  - Todas las hojas se encuentran en el mismo nivel.
  - Todo nodo (llamado página) contiene m elementos (claves) **almacenados de forma ordenada**.
    - La página raíz contiene 1 <= m <= 2n claves.
    - Toda página no raíz contiene n <= m <= 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	$2 n * (n + 1)^{h - 2}$

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

Árboles B (Bayer & McCreight)

Altura **Máxima** de un árbol B-n

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

- $h_{max} \approx 1 + \log_{n+1}(N+1)/2$
  - Si la constante **n** es lo suficientemente grande, h_{max} se aproxima a:
    - $h_{max} \approx \log_n N$.

Rango de altura árbol B-n
$h <\approx 1 + \log_{n+1}(N+1)/2$ $O(h) <\approx O(\log_n N)$

- 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	2 n * (2n + 1)
3	$(2n + 1)^2$	2n	2 n * $(2n + 1)^2$
4	$(2n + 1)^3$	2n	2 n * $(2n + 1)^3$
...			
h	$(2n + 1)^{h - 1}$	2n	2 n * $(2n + 1)^{h - 1}$

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

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.

- $h_{min} \approx Log_{2n+1}(N+1)$.
  - Si la constante **n** es lo suficientemente grande, h_{min} se aproxima a:
    - $h_{min} \approx Log_{2n}N$.

Rango de altura árbol B-n
$Log_{2n+1}(N+1) <\approx h <\approx 1 + Log_{n+1}(N+1)/2$
$O(Log_{2n}N) <= O(h) <= O(Log_{n}N)$

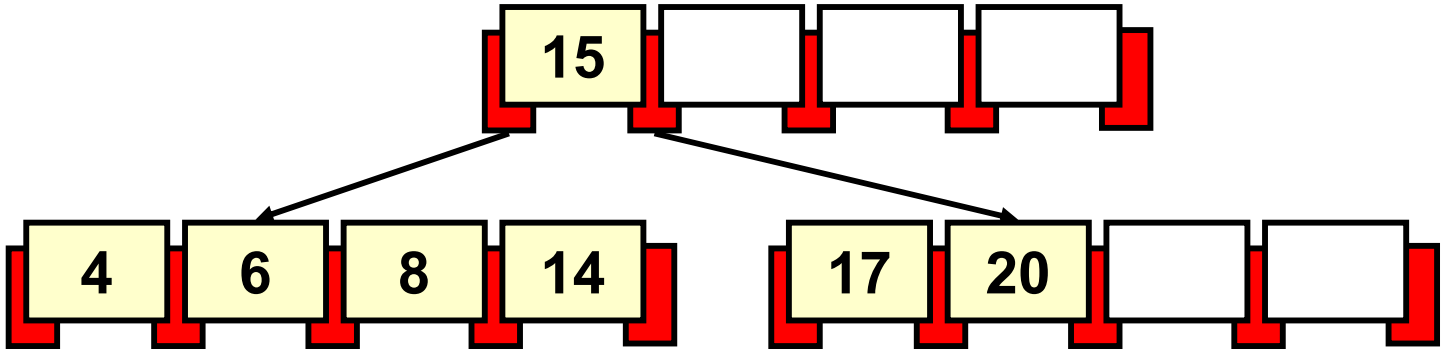
- Cuanto mayor sea el orden del árbol (**n**) menor será la altura del árbol.

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 (elements[j]) de la página tal que 0 <= j <=m
  - Cargar la página *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 <= m <= 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…

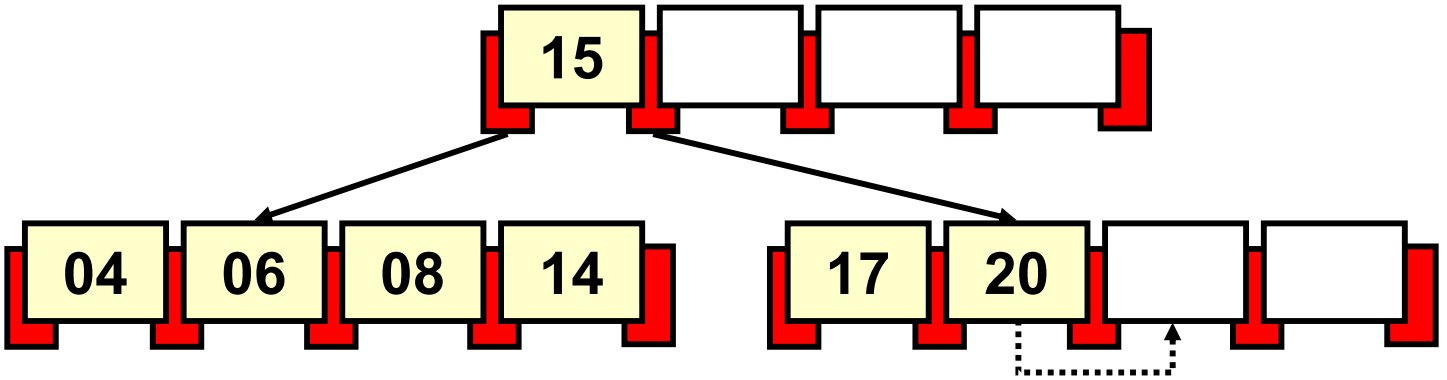
… entre 5 y 6 accesos a disco

… un AVL requiere entre 20 y 28 accesos

Árboles B (Bayer & McCreight)

Inserción

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

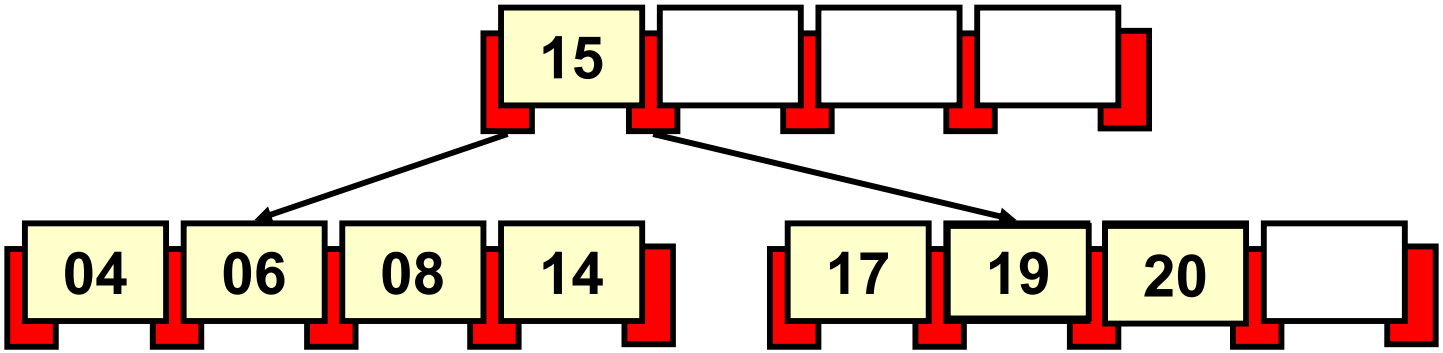


- La inserción siempre se produce en las hojas y tras una búsqueda infructuosa.

Árboles B (Bayer & McCreight)

Inserción

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

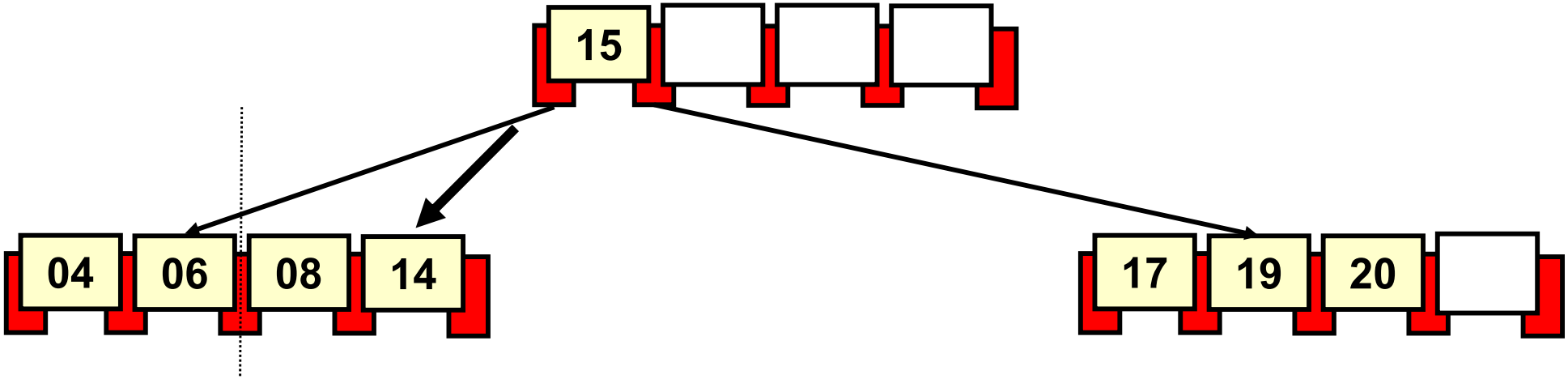


- La inserción siempre se produce en las hojas y tras una búsqueda infructuosa.

Árboles B (Bayer & McCreight)

Inserción

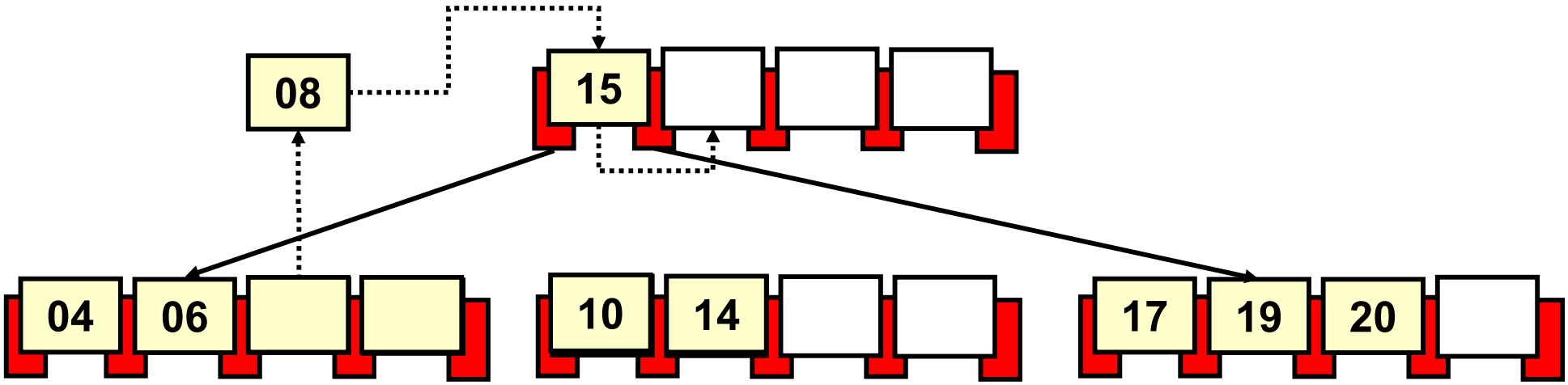
- **Caso 2**: Pagina hoja tiene m = 2*n claves (**Overflow**).
  - División de la hoja en dos, repartiendo claves en ellas
    - Últimas (m+1)/2 claves en hoja nueva.
    - Primeras (m+1)/2 claves permanecen en hoja original.
    - Promociona el elemento central (mediana) a la página padre actuando de 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

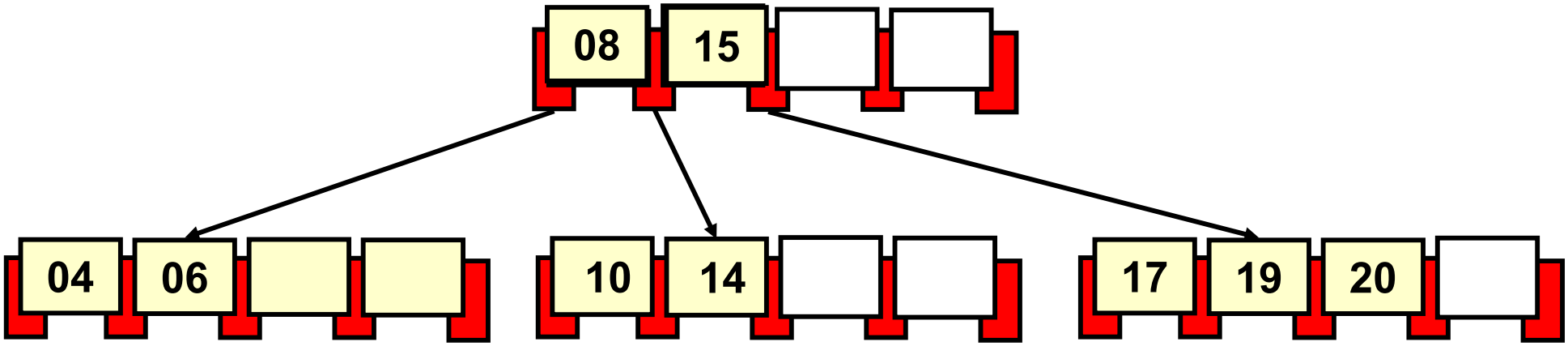
- **Caso 2**: Pagina hoja tiene m = 2*n claves (**Overflow**).
  - División de la hoja en dos, repartiendo claves en ellas
    - Últimas (m+1)/2 claves en hoja nueva.
    - Primeras (m+1)/2 claves permanecen en hoja original.
    - Promociona el elemento central (mediana) a la página padre actuando de 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*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.
    - $O(\log_{n}(N)) * O(n) = O(\log_{n}(N))$.

CLASSWORK

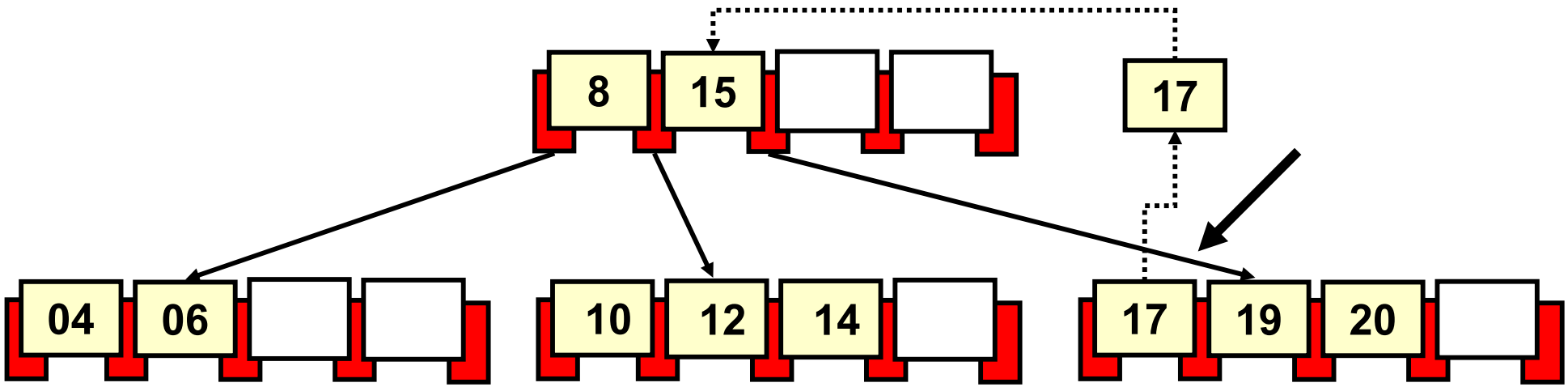
PLAYGROUND

- **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

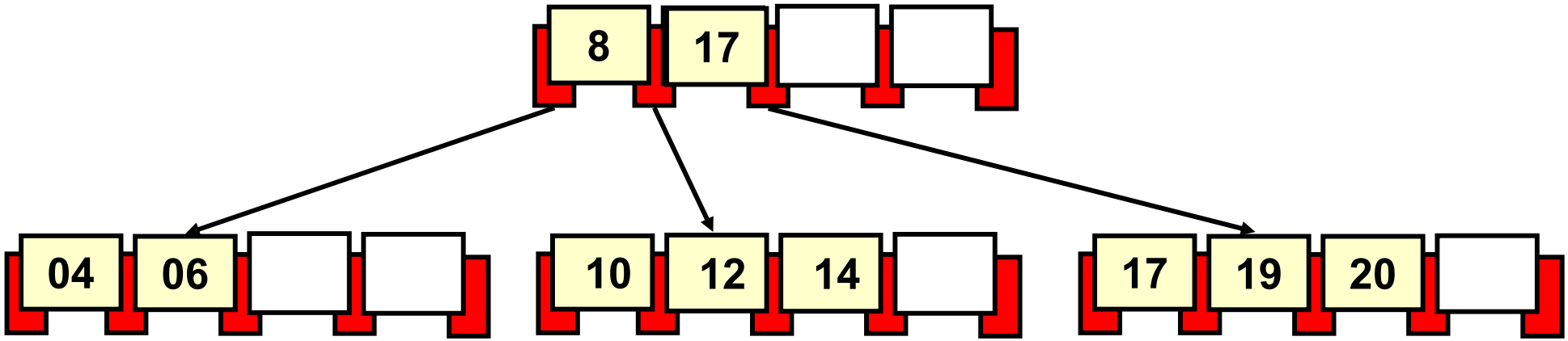
- **Borrado de elemento interior**
  - Sustituir elemento por su sucesor
    - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
  - Borrar elemento **de página donante**.



Árboles B (Bayer & McCreight)

Borrado

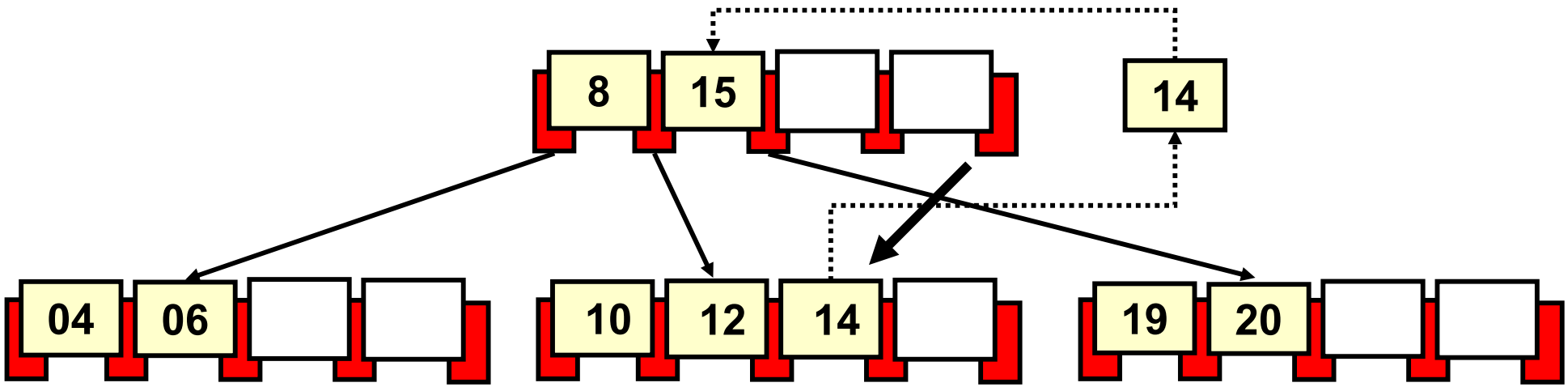
- **Borrado de elemento interior**
  - Sustituir elemento por su sucesor
    - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
  - Borrar elemento **de página donante**.



Árboles B (Bayer & McCreight)

Borrado

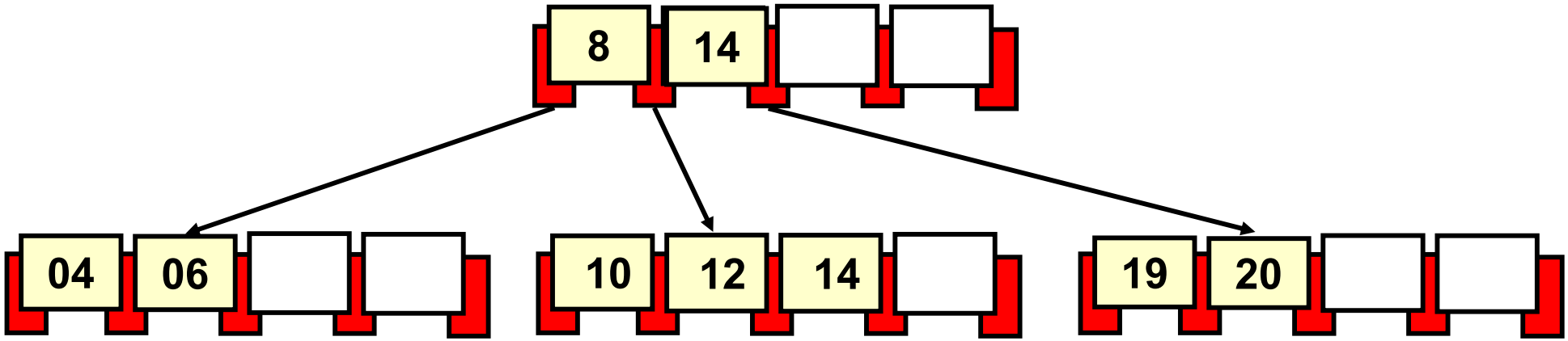
- **Borrado de elemento interior**
  - Sustituir elemento por su sucesor
    - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
  - Si la página donante se encuentra en situación crítica,
    - Intentar sustituir por antecesor (elemento extremo 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

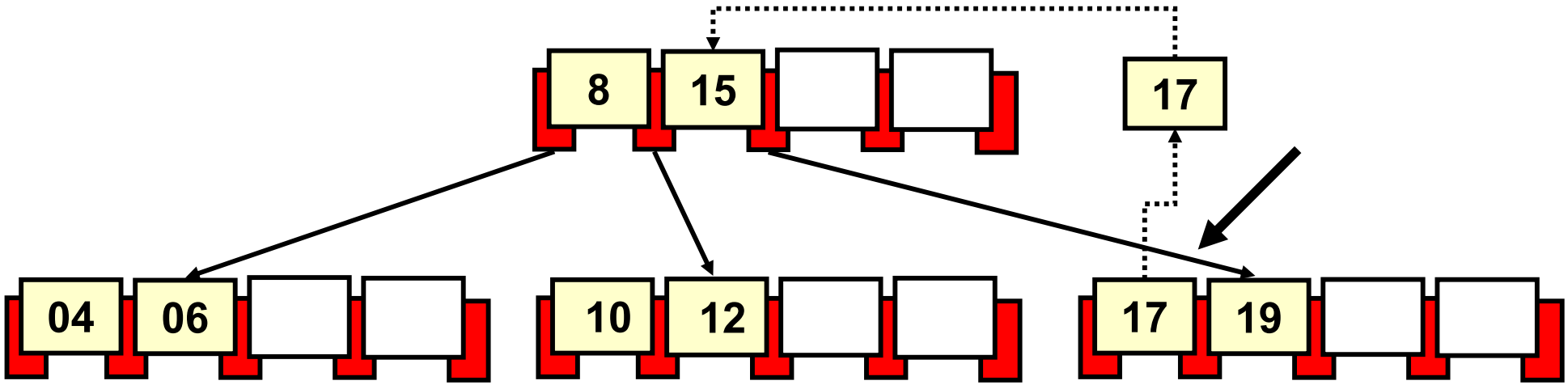
- **Borrado de elemento interior**
  - Sustituir elemento por su sucesor
    - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
  - Si la página donante se encuentra en situación crítica,
    - Intentar sustituir por antecesor (elemento extremo 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

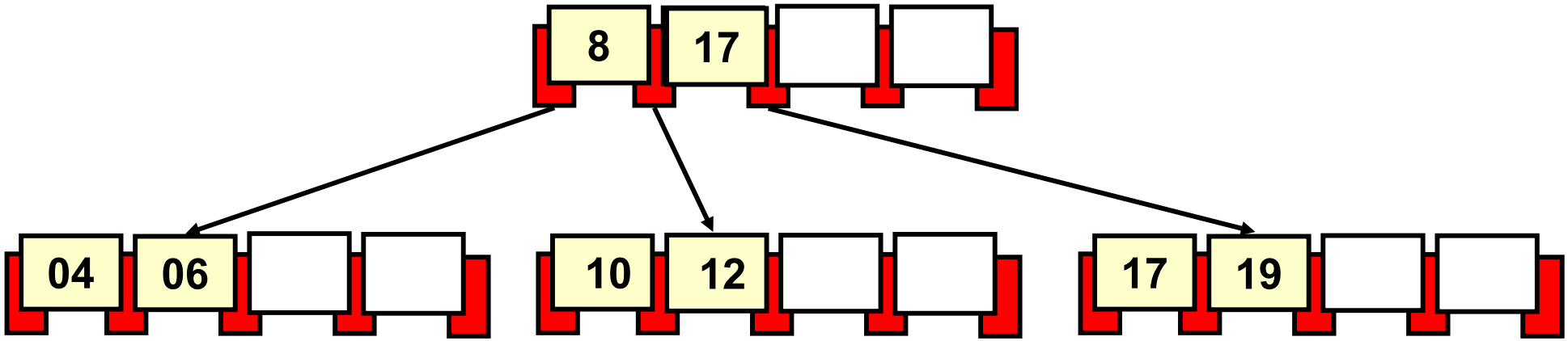
- **Borrado de elemento interior**
  - Sustituir elemento por su sucesor
    - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
  - Si la página donante se encuentra en situación crítica,
    - Intentar sustituir por antecesor (elemento extremo 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

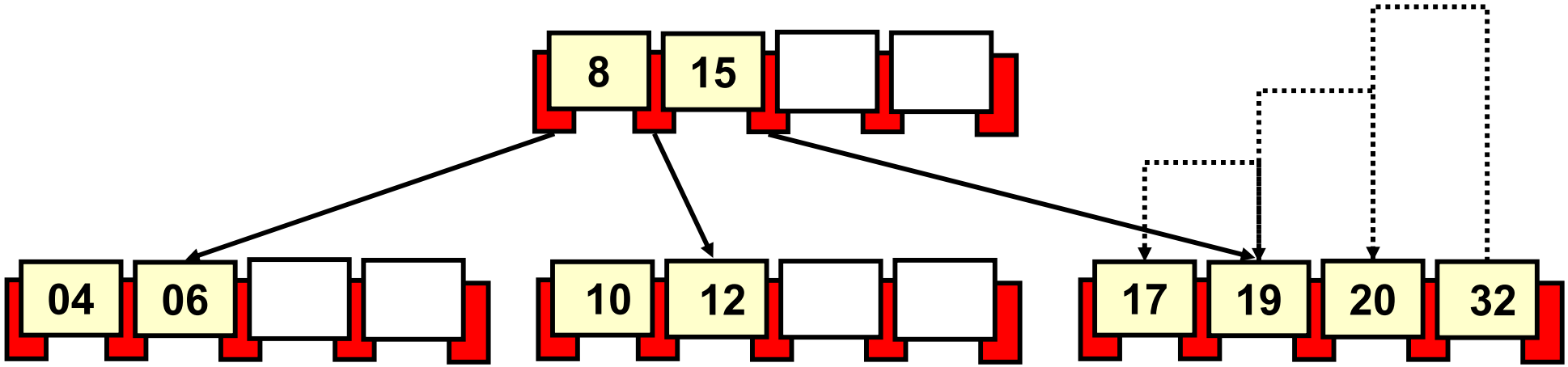
- **Borrado de elemento interior**
  - Sustituir elemento por su sucesor
    - El sucesor se encuentra en página elemento extremo izquierdo de subárbol derecho (se trata de una hoja).
  - Si la página donante se encuentra en situación crítica,
    - Intentar sustituir por antecesor (elemento extremo 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

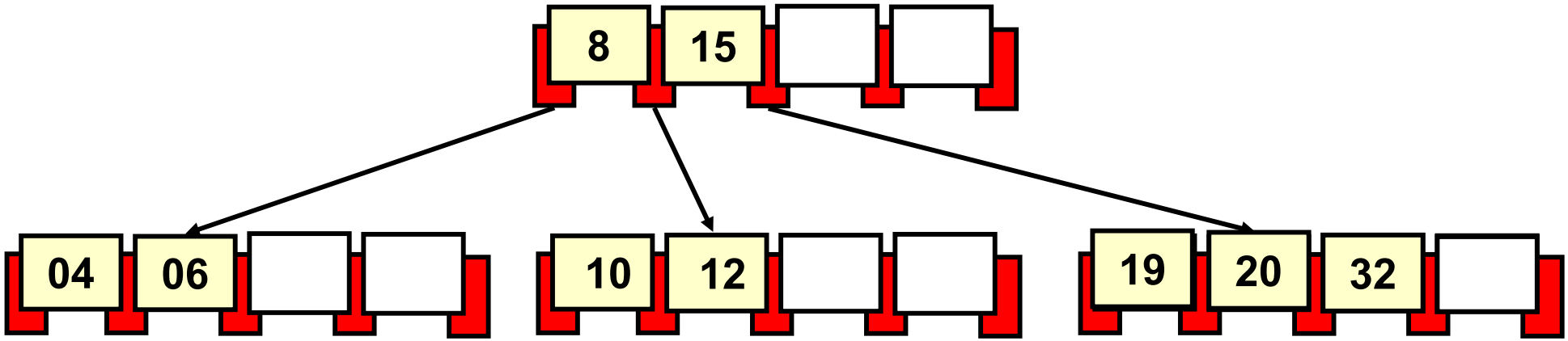
- **Borrado de elemento en página hoja**
  - **Caso 1**: Página tiene n < m claves.
    - Se desplazan los elementos de clave mayor una posición hacia la izquierda.



Árboles B (Bayer & McCreight)

Borrado

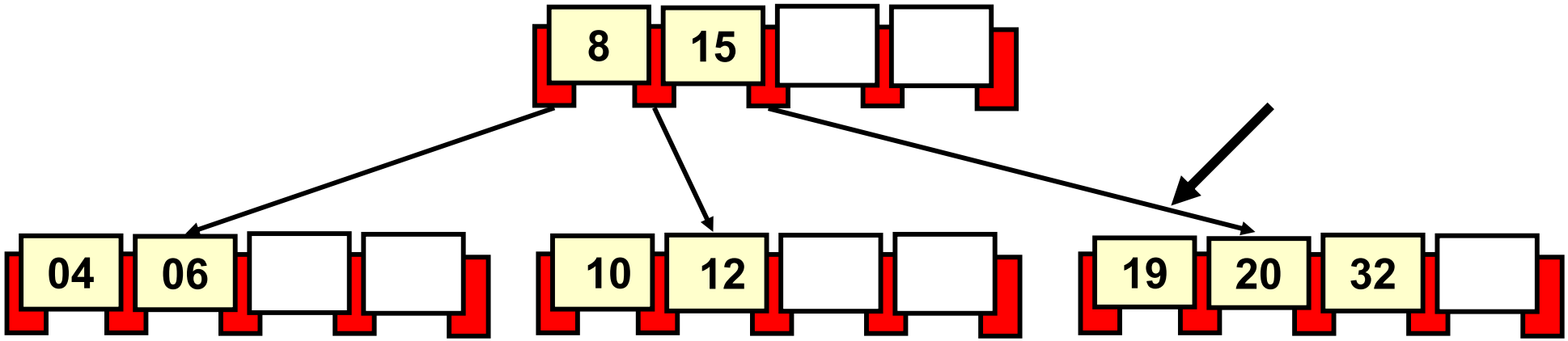
- **Borrado de elemento en página hoja**
  - **Caso 1**: Página tiene n < m claves.
    - Se desplazan los elementos de clave mayor una posición hacia la izquierda.



Árboles B (Bayer & McCreight)

Borrado

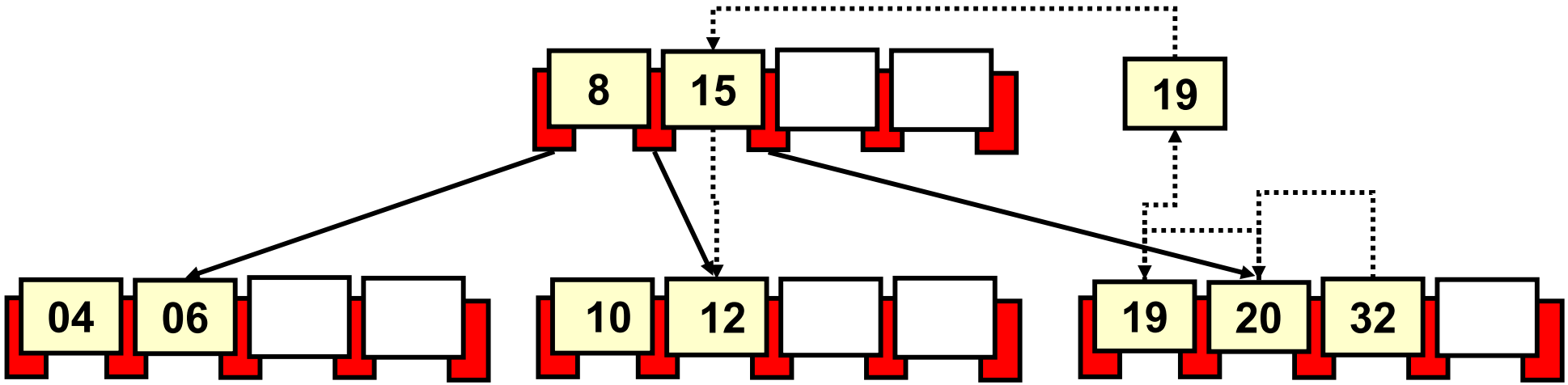
- **Borrado de elemento en página hoja**
  - **Caso 2**: Página tiene n = m claves (*underflow*).
    - Se busca entre las hojas adyacentes alguna que tenga n < m claves y se le pide la cesión de una de ellas.
      - » Se consulta **primero con la vecina derecha** (si existe) y si ésta no puede ceder 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

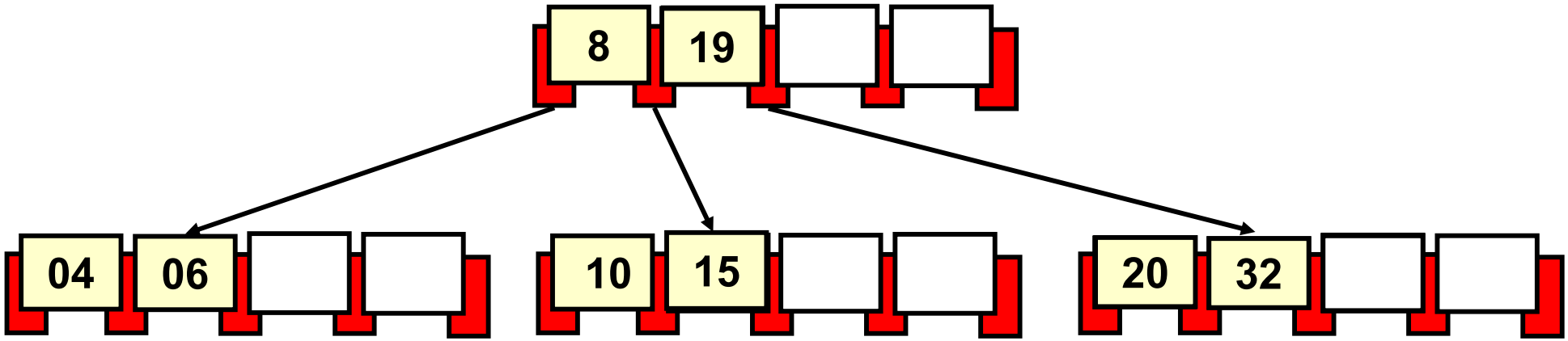
- **Borrado de elemento en página hoja**
  - **Caso 2**: Página tiene n = m claves (*underflow*).
    - El préstamo se realiza **siempre a través de la página padre** de las dos implicadas.
    - El elemento cedido por la página donante se envía a la página padre para sustituir al separador, quien baja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



Árboles B (Bayer & McCreight)

Borrado

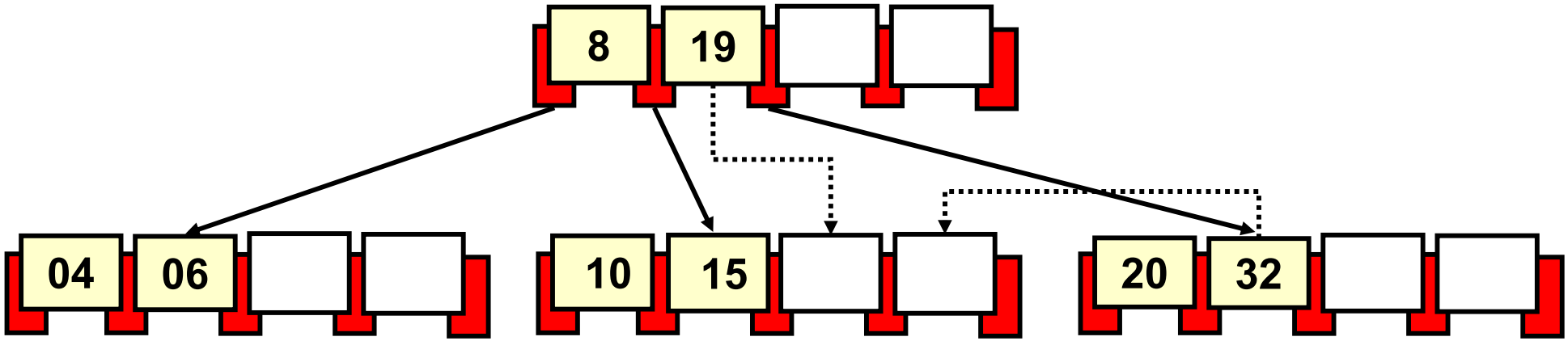
- **Borrado de elemento en página hoja**
  - **Caso 2**: Página tiene n = m claves (*underflow*).
    - El préstamo se realiza **siempre a través de la página padre** de las dos implicadas.
    - El elemento cedido por la página donante se envía a la página padre para sustituir al separador, quien baja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



Árboles B (Bayer & McCreight)

Borrado

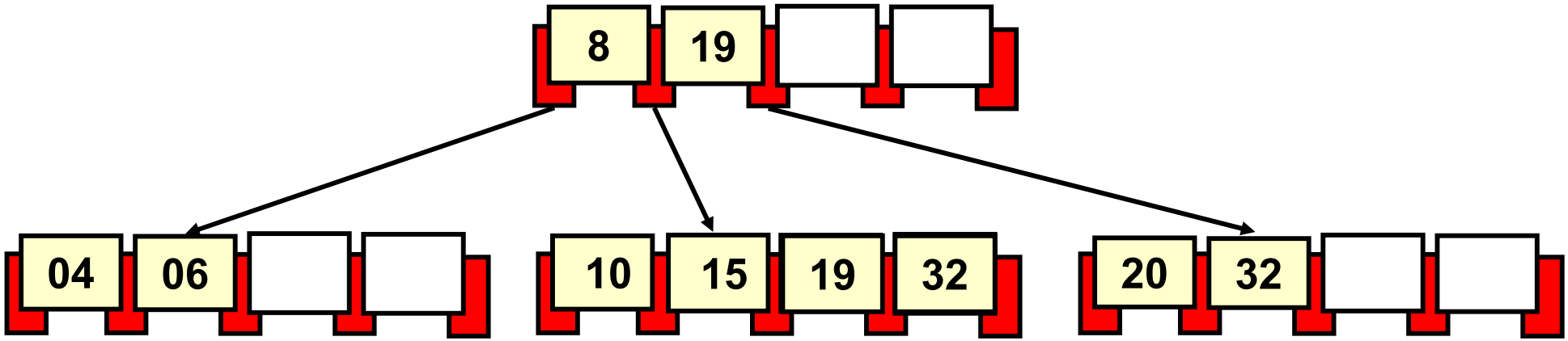
- **Borrado de elemento en página hoja**
  - **Caso 2b**: Página tiene n = m claves (*underflow*) y ninguna página vecina puede ceder claves.
    - Las dos páginas adyacentes (izquierda y derecha) se encuentran en situación crítica.
    - La página se fusiona con la que vecina derecha (de no tenerla, 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

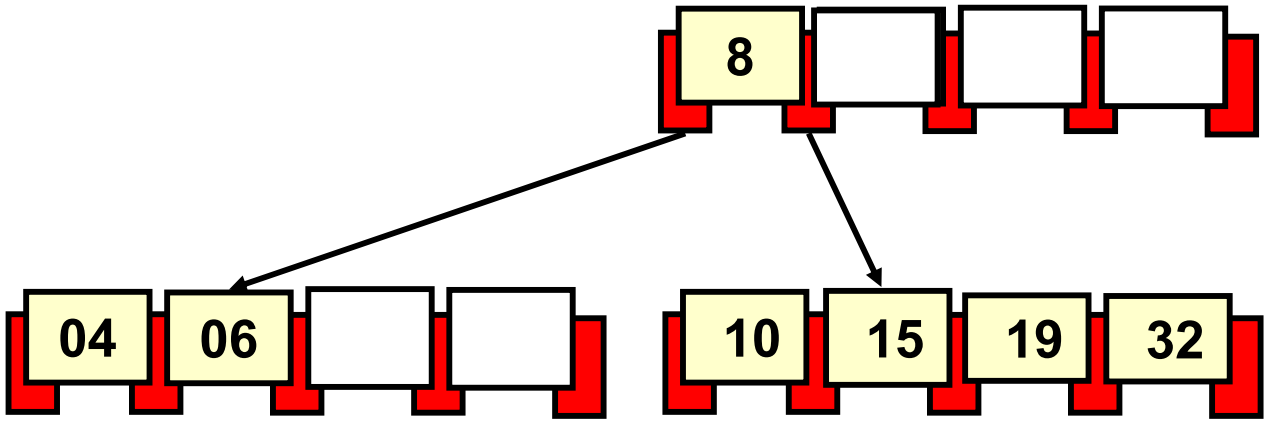
- **Borrado de elemento en página hoja**
  - **Caso 2b**: Página tiene n = m claves (*underflow*) y ninguna página vecina puede ceder claves.
    - Las dos páginas adyacentes (izquierda y derecha) se encuentran en situación crítica.
    - La página se fusiona con la que vecina derecha (de no tenerla, 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.

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

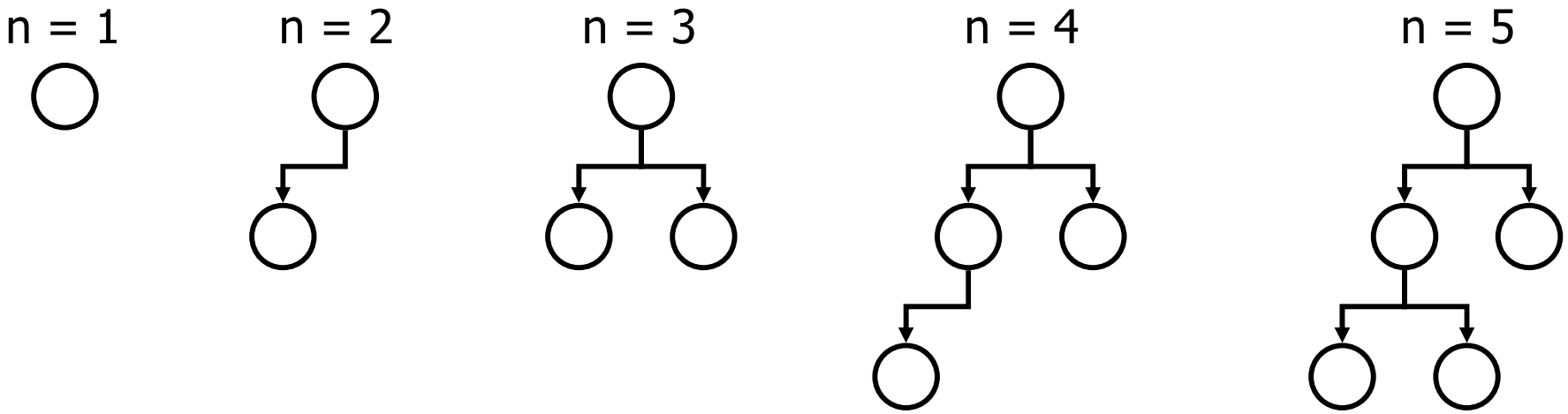
- Optimizar dos operaciones…
  1. Insertar elemento (con su prioridad asociada).
  2. Sacar elemento de máxima prioridad.

- 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.

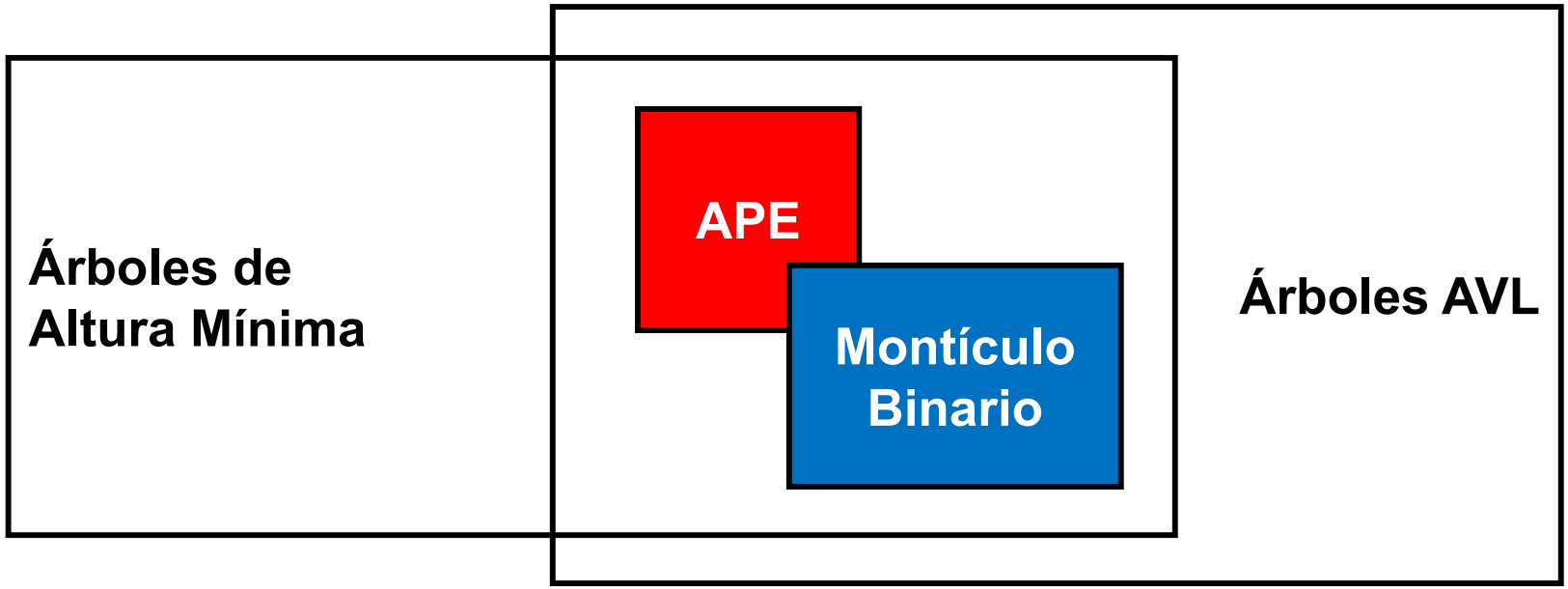


Rango de altura en un Montículo Binario
$h = E[\log_2 n] + 1$ $O(h) <= O(\text{Log}_2 n)$

Montículos Binarios

Propiedades

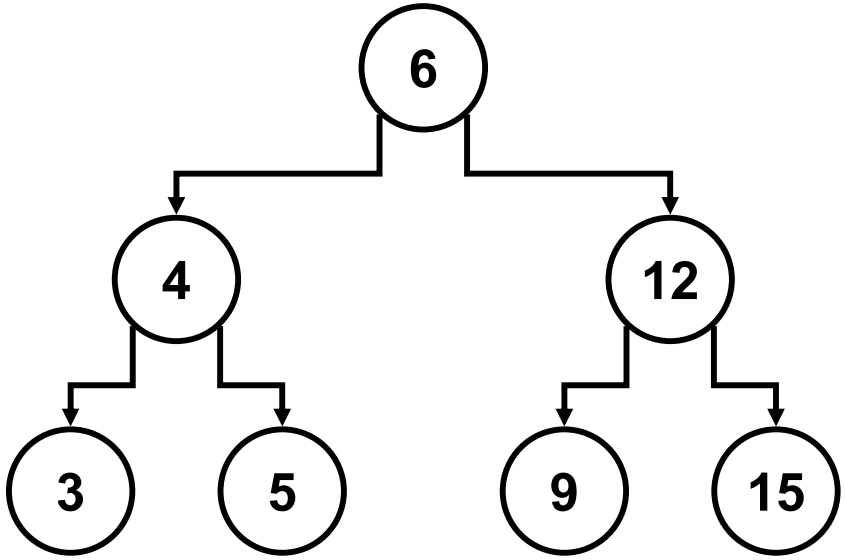
- Todo Montículo Binario es un Árbol de Altura Mínima

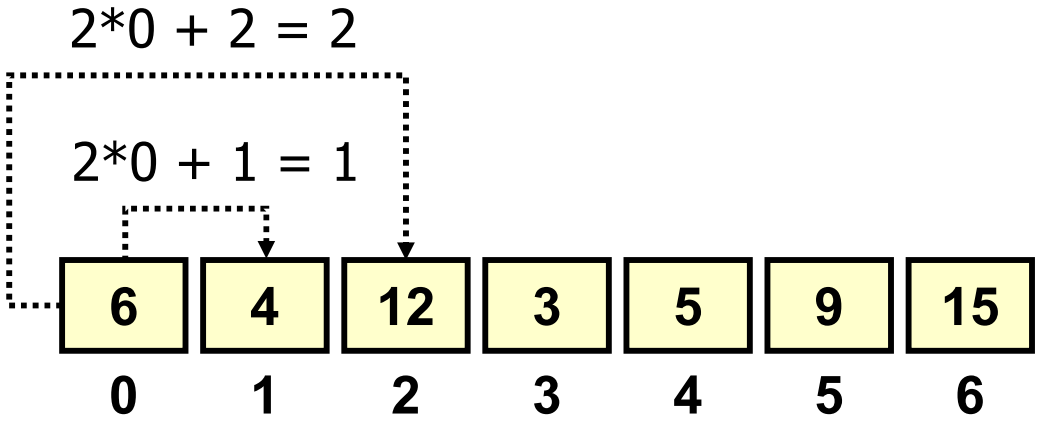


Montículos Binarios

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)

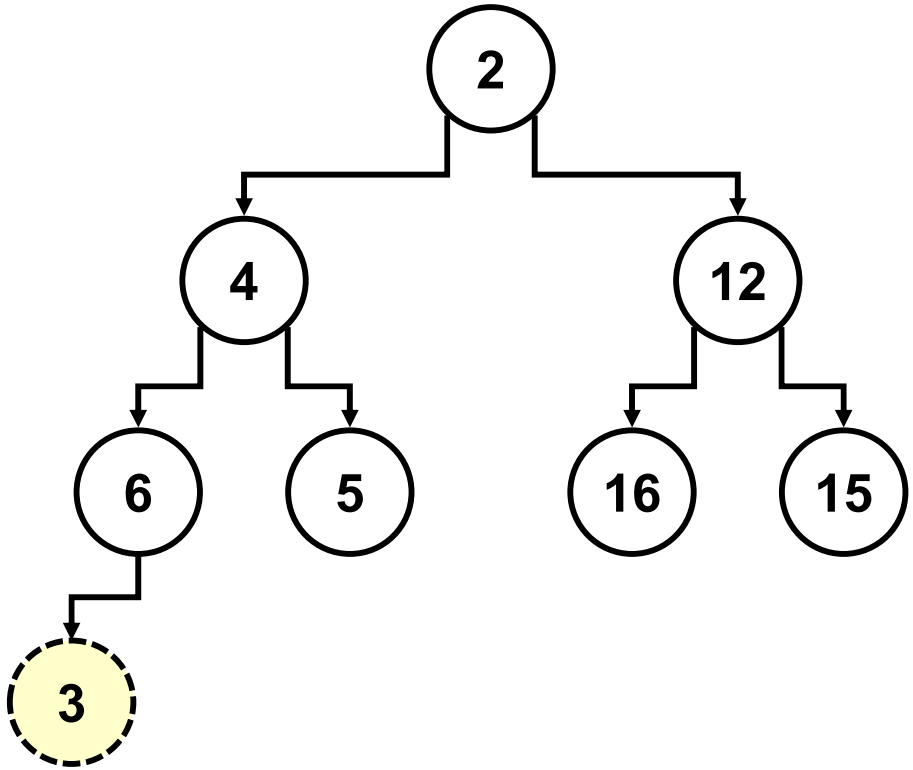
- **Montículo de Mínimos**
  - Todo nodo tiene una clave **menor** que la de sus hijos.
  - El **menor elemento** se encuentra en la raíz (posición 0 del vector).
    - Optimiza las operaciones ***Add*** y ***getMin***.

- **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: $O(\log_2 n)$

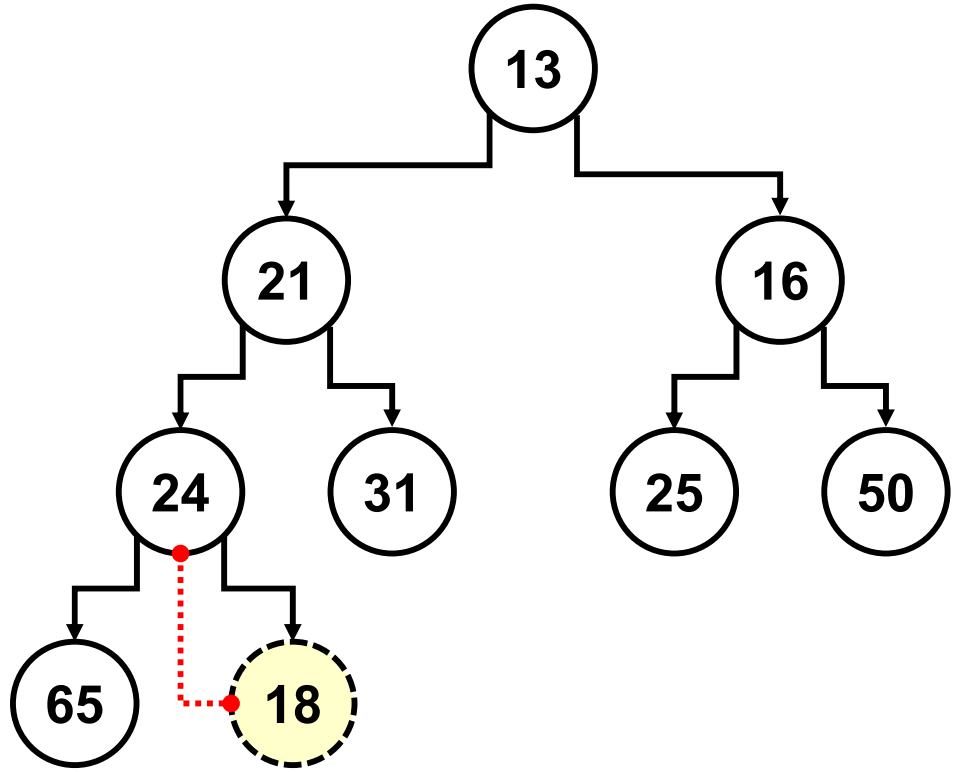
DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Montículos Binarios

Ejercicio

Comparar posición E[(i-1)/2]

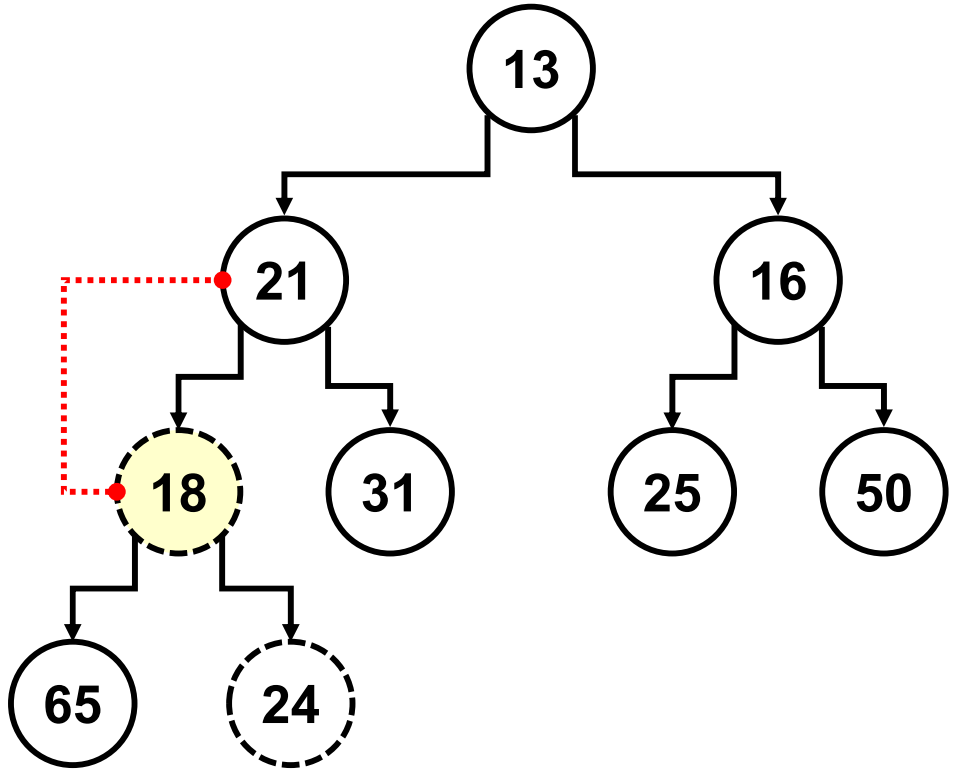
Comparar posición E[(8-1)/2] = E[7/2] = 3



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

Montículos Binarios

Ejercicio



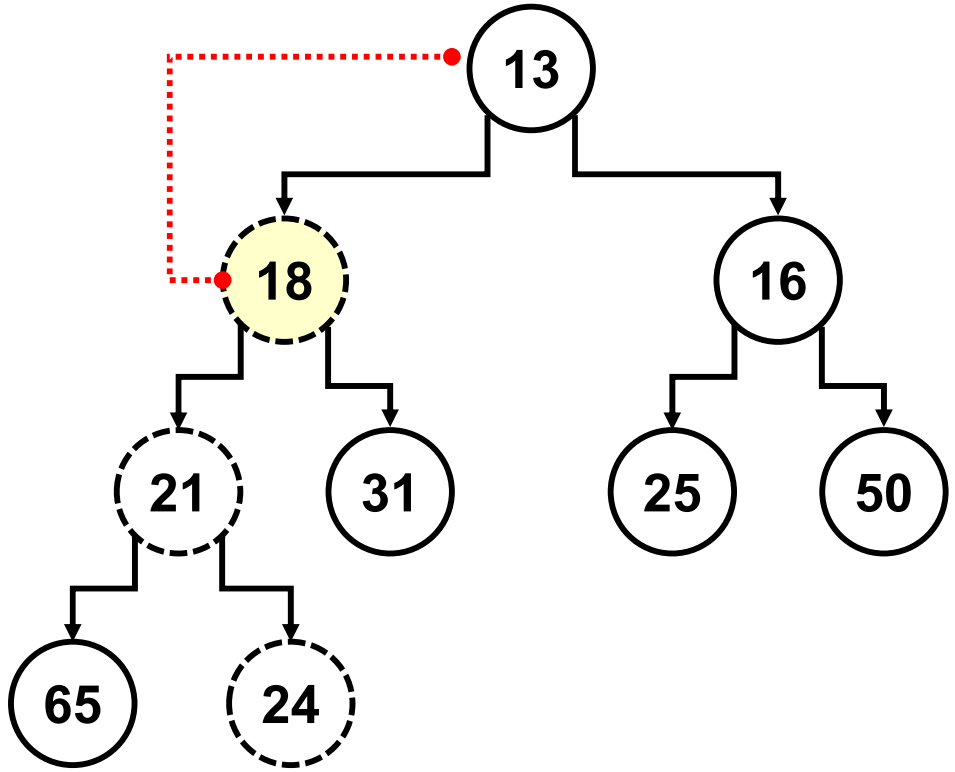
Comparar posición E[(i-1)/2]

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

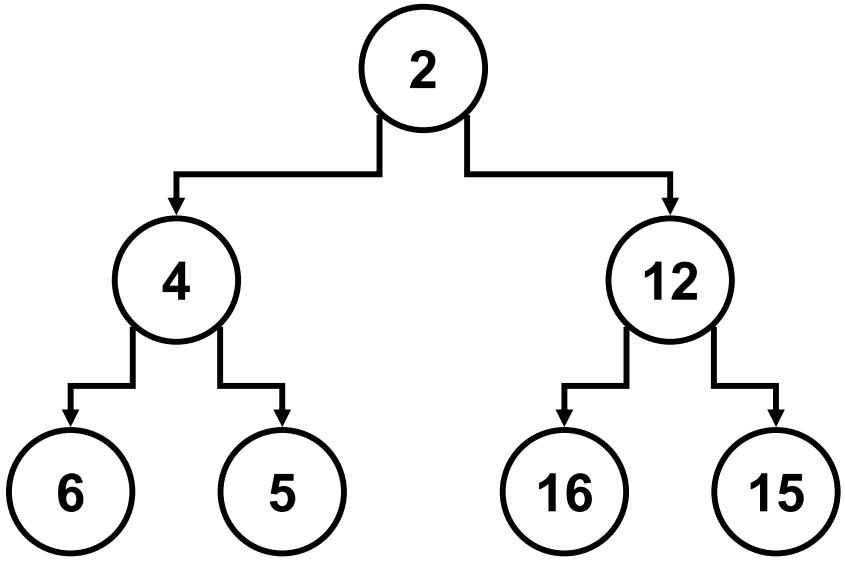
Montículos Binarios

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)

$O(\log_2 n)$

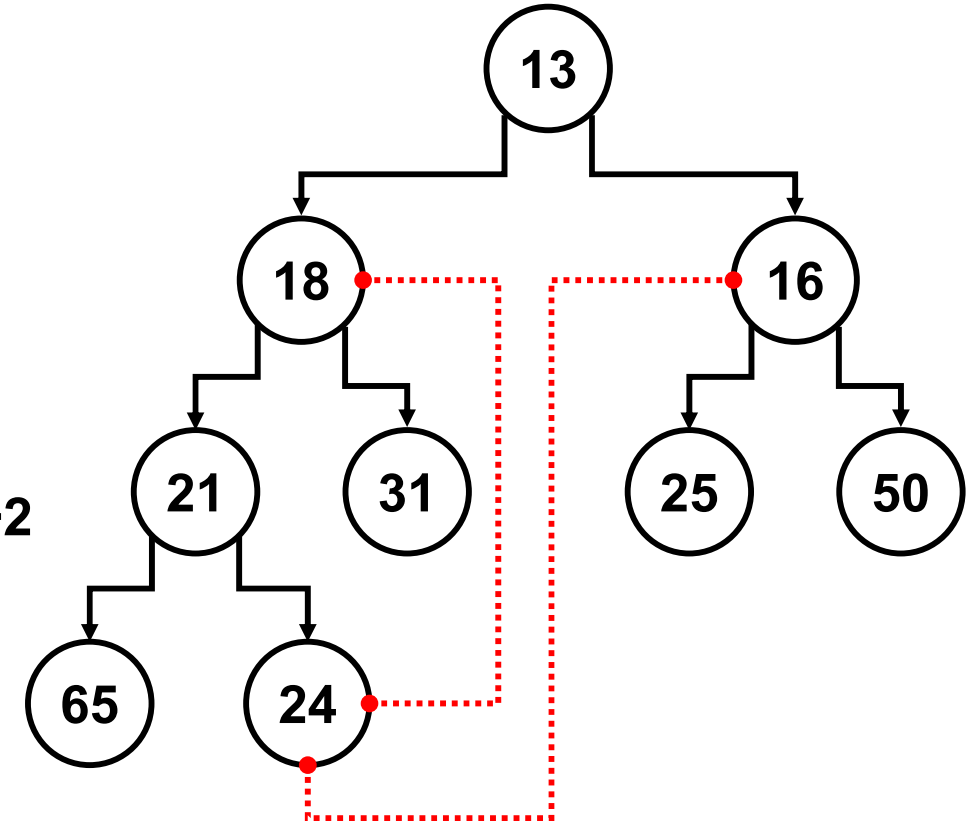


DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Montículos Binarios

Ejercicio 1

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

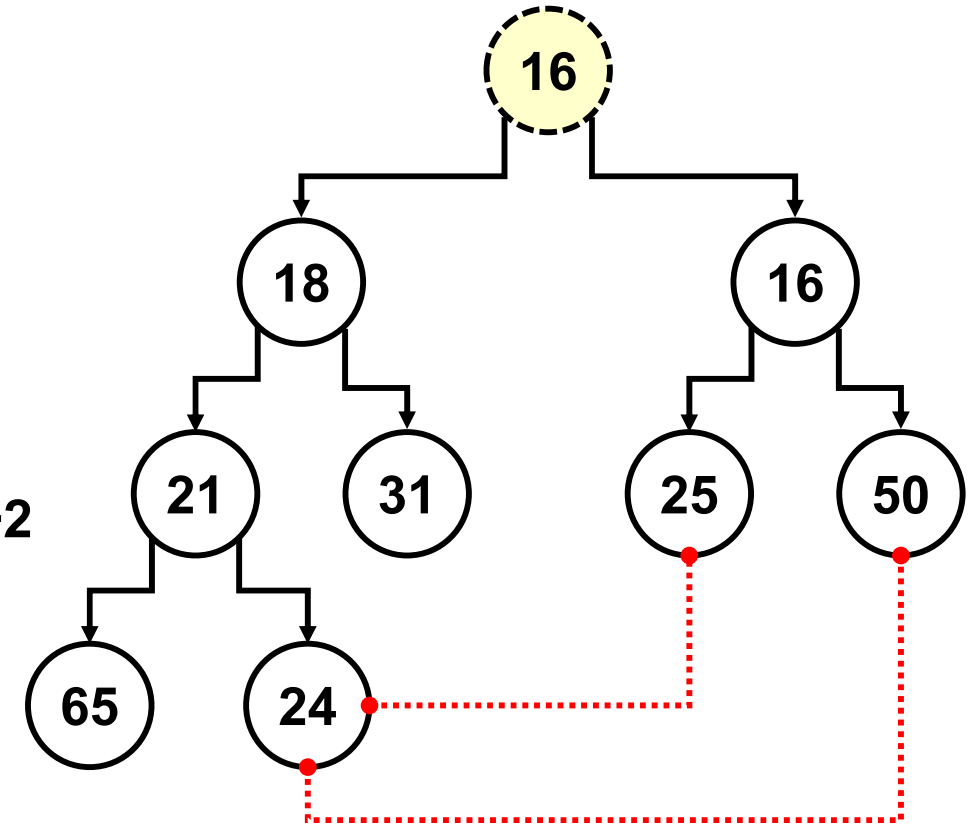


it	0	1	2	3	4	5	6	7	8
1	13	18	16	21	31	25	50	65	24

Montículos Binarios

Ejercicio 1

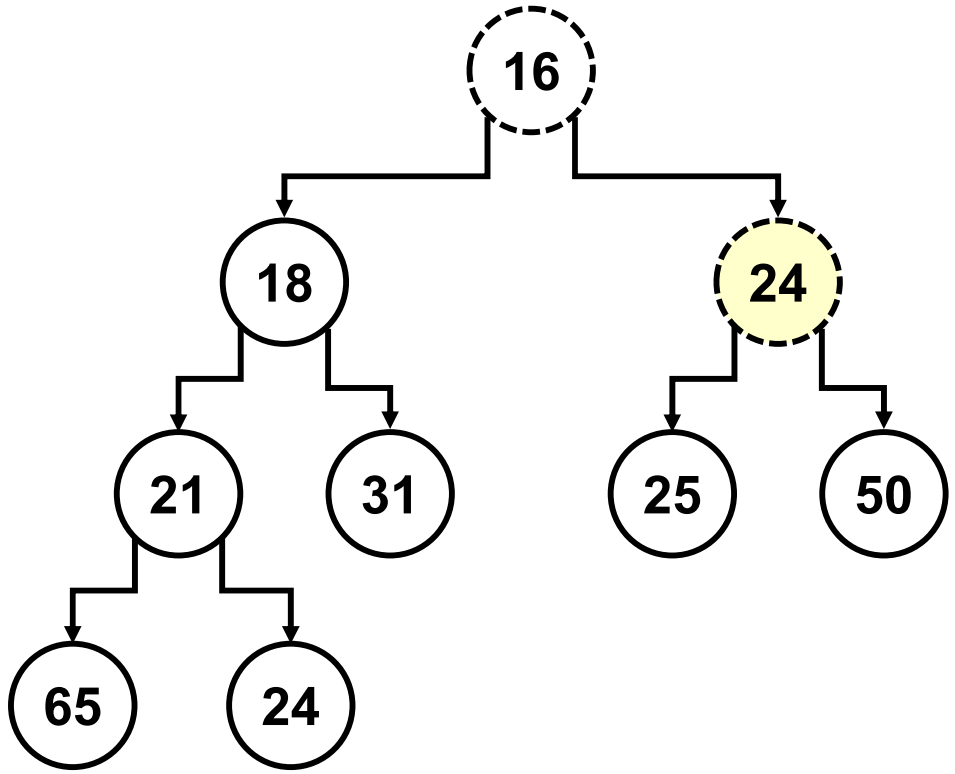
Comparar con posiciones 2(2)+1 y 2(2)+2



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	55	24

Montículos Binarios

Ejercicio 1

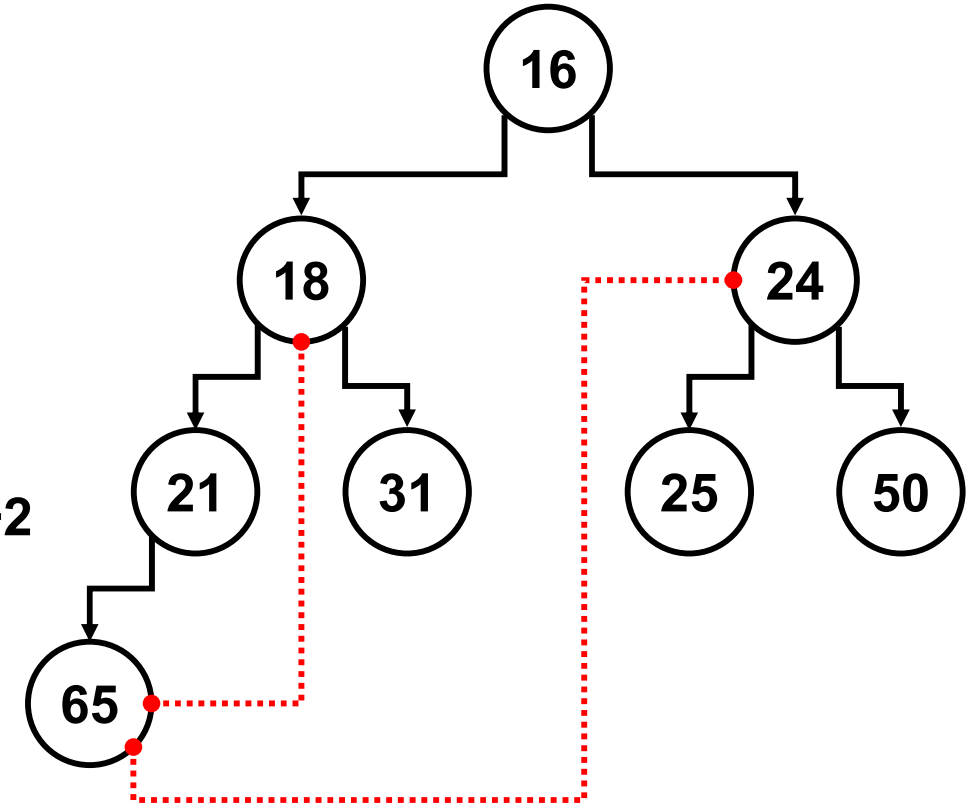


it	0	1	2	3	4	5	6	7	8
1	13	18	16	21	31	25	50	65	24
2	16	18	16	21	31	25	50	65	24
3	16	18	24	21	31	25	50	65	24

Montículos Binarios

Ejercicio 2

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

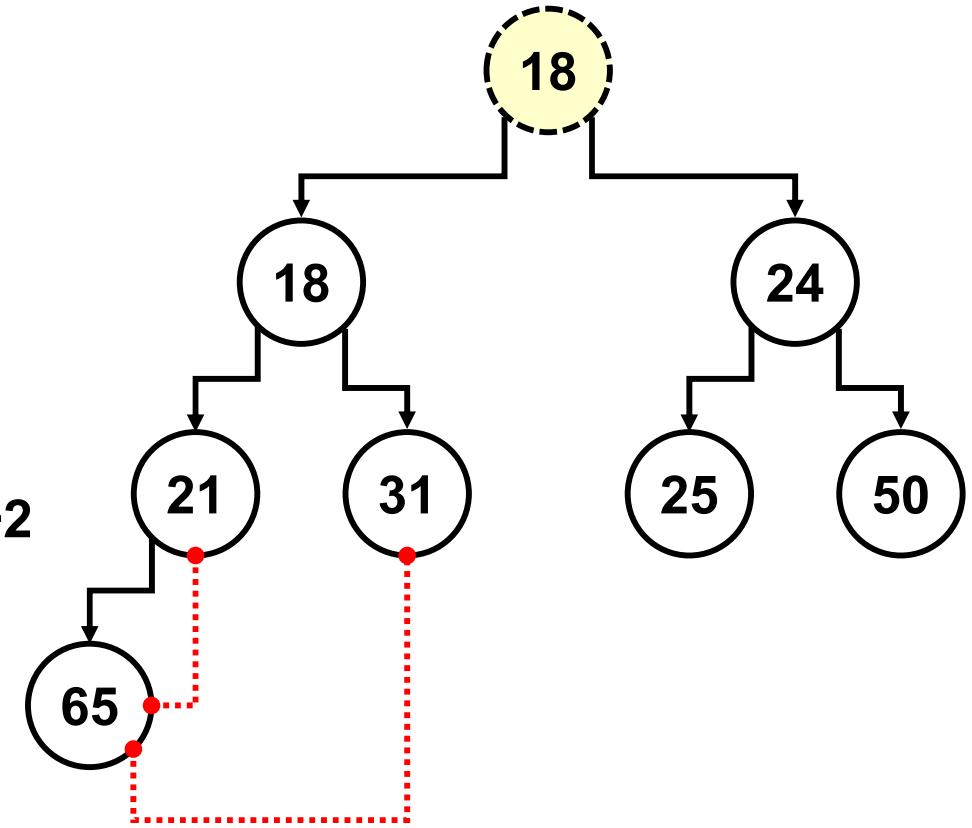


it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	

Montículos Binarios

Ejercicio 2

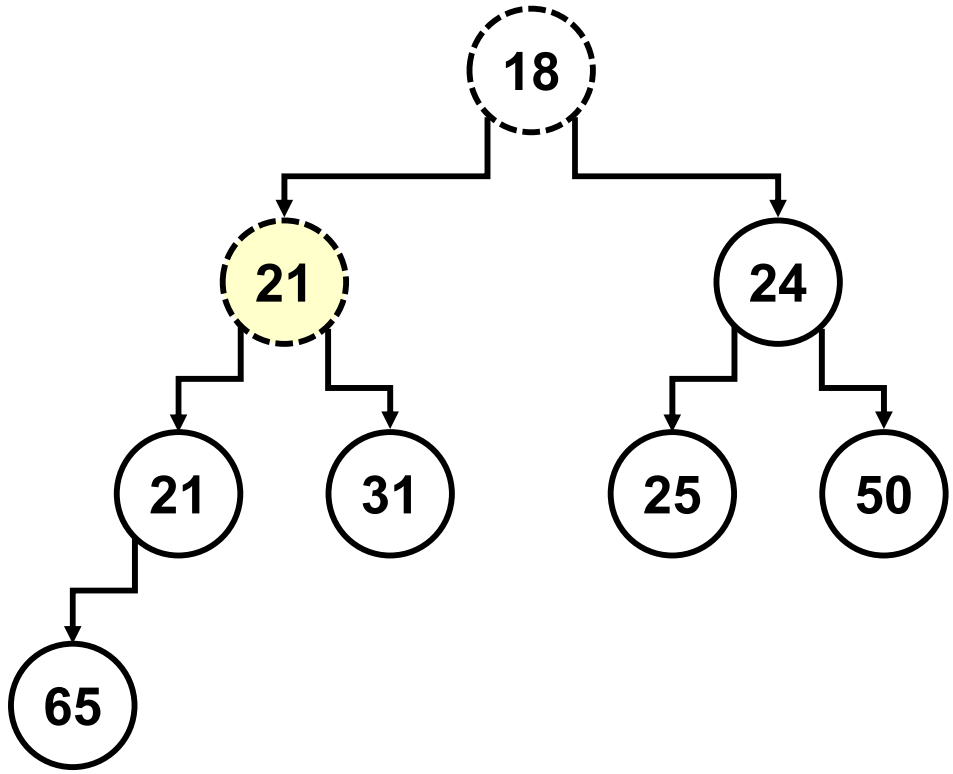
Comparar con posiciones 2(1)+1 y 2(1)+2



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	

Montículos Binarios

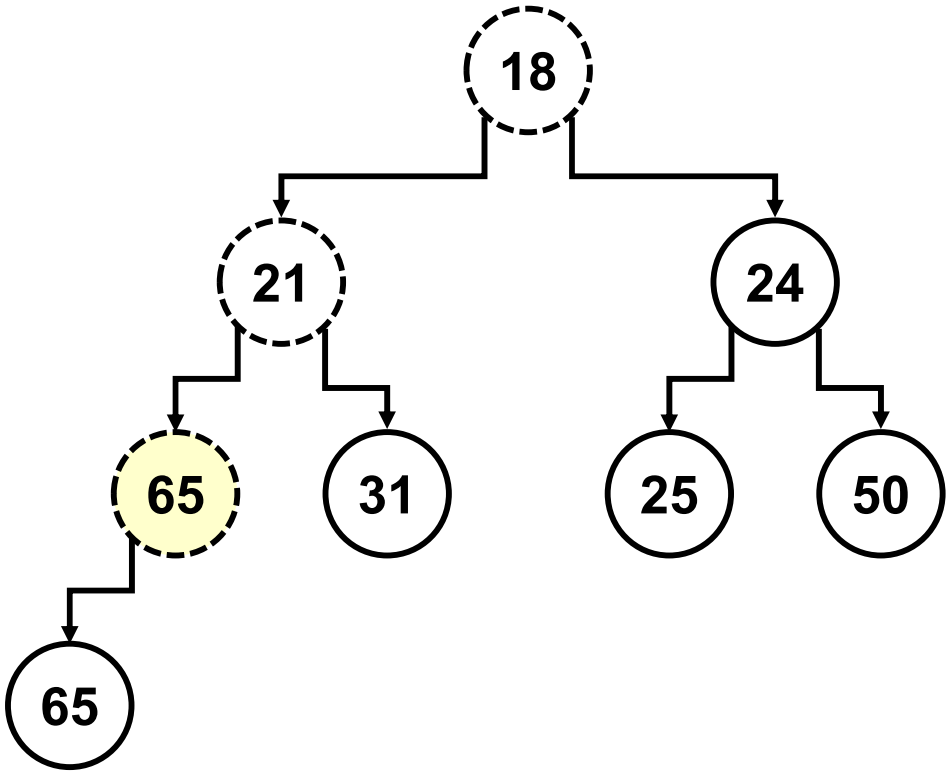
Ejercicio 2



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

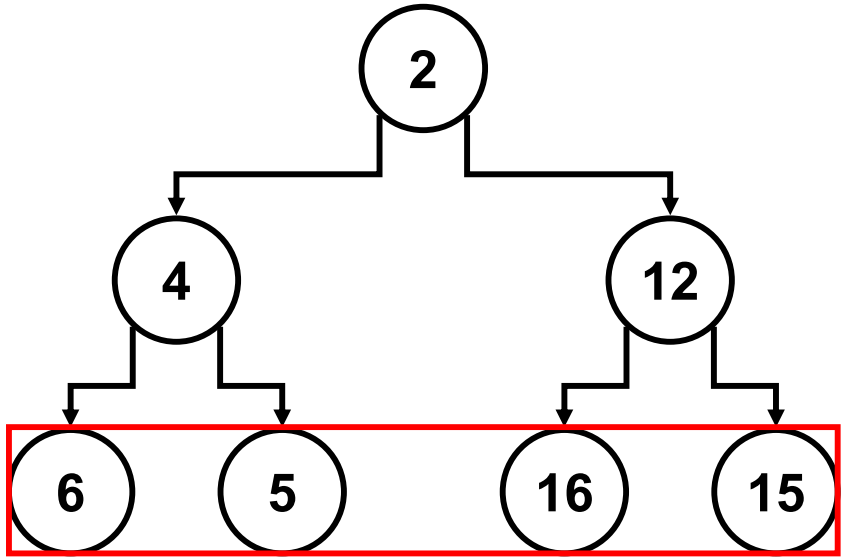


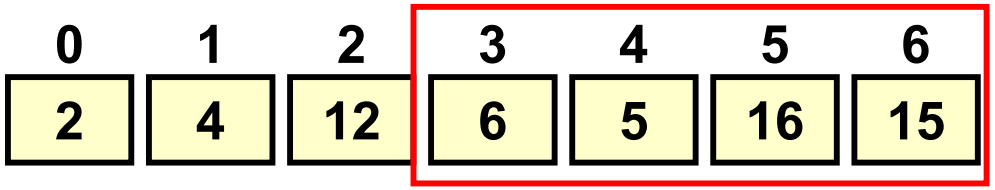
it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	
2	18	18	24	21	31	25	50	65	
3	18	21	24	21	31	25	50	65	
4	18	21	24	65	31	25	50	65	

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.



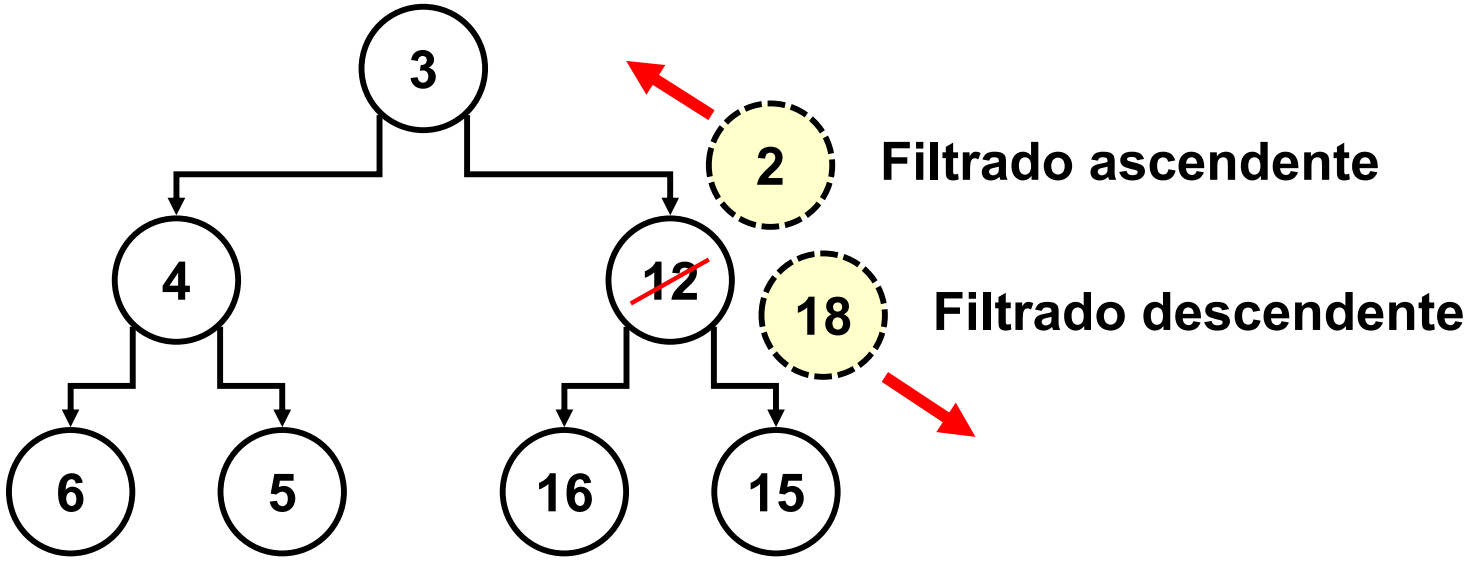


DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

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.
```

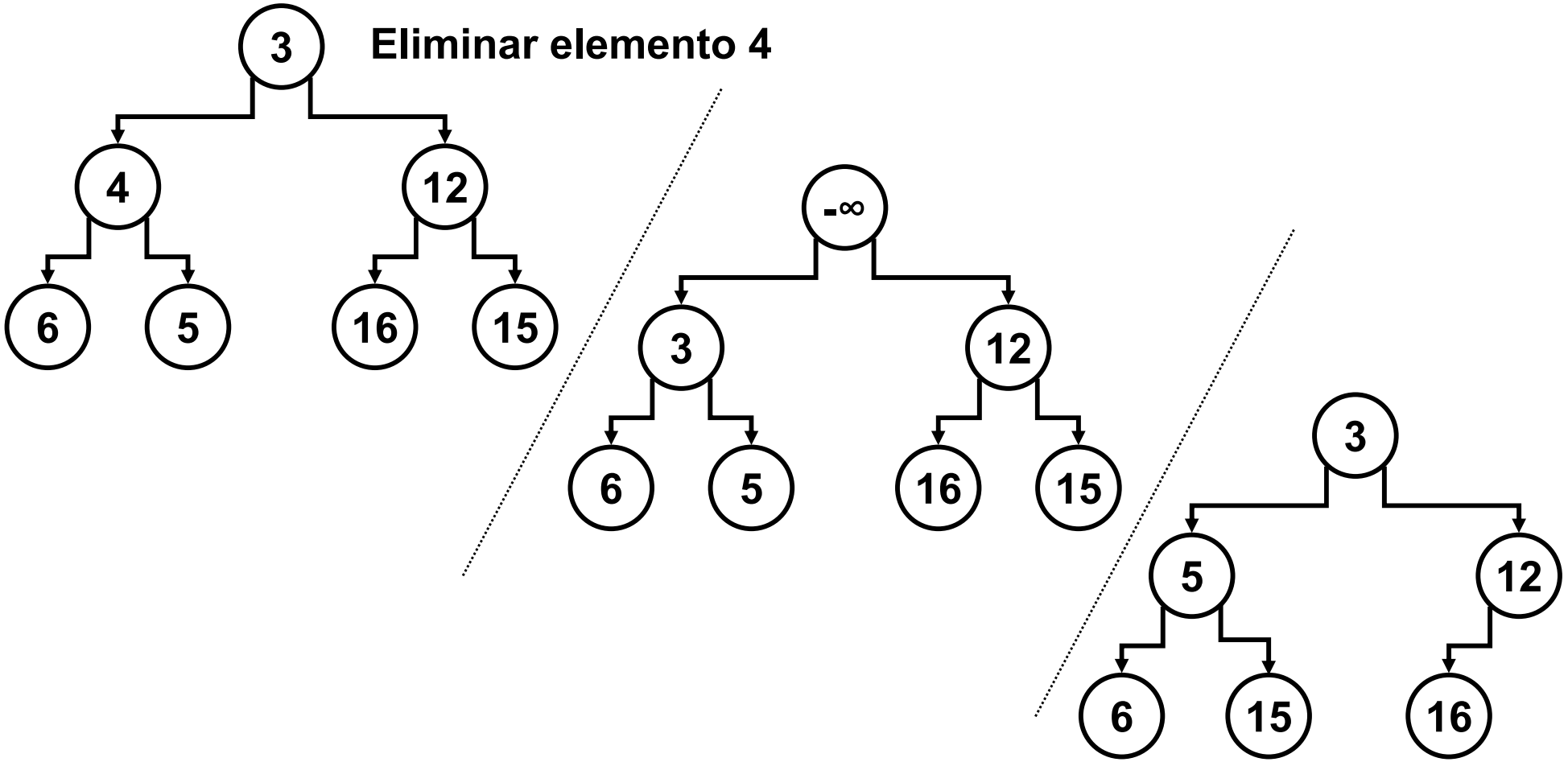


DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Operaciones Especiales con Montículos

Eliminar elemento	$O(\log_2 n)$

```
1. Cambiar su prioridad a -∞ para subirlo a la raíz.
2. Invocar al método sacar().
```



DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

C60 Series

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

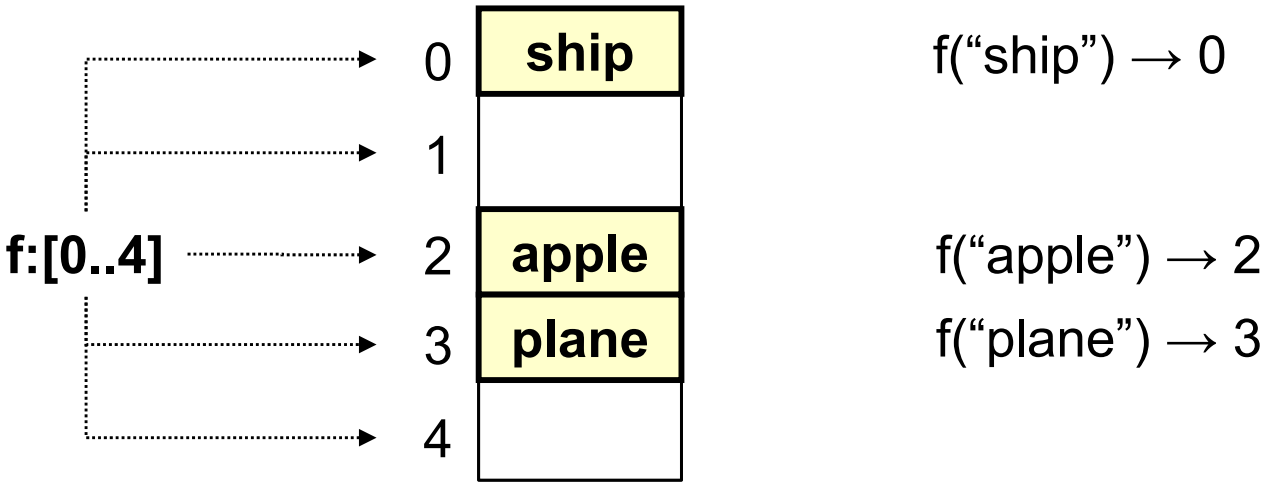
```
class HashTable<T> {
  private final static int B = 5; //size
  private ArrayList<HashNode<T>> associativeArray;

  public HashTable() {
        associativeArray = new ArrayList<HashNode<T>>(B);
  }

  int private 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].

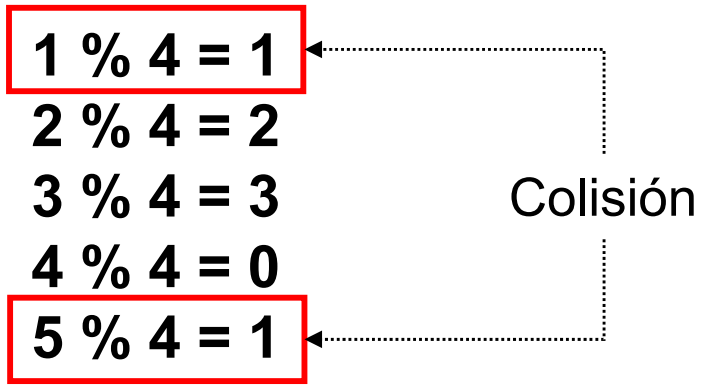


Función Hash

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

- 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);
}
```

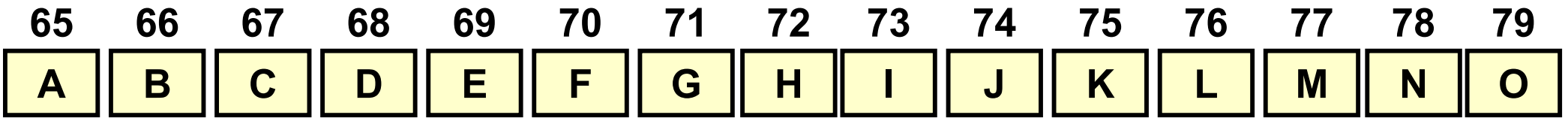
- Convierte la cadena a un entero para luego aplicar la función de dispersión.
  - La función *convert* suma los códigos de representación de cada letra de la cadena.
    - (Códigos ASCII, EBDIC, etc).

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
Total	**368**



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.

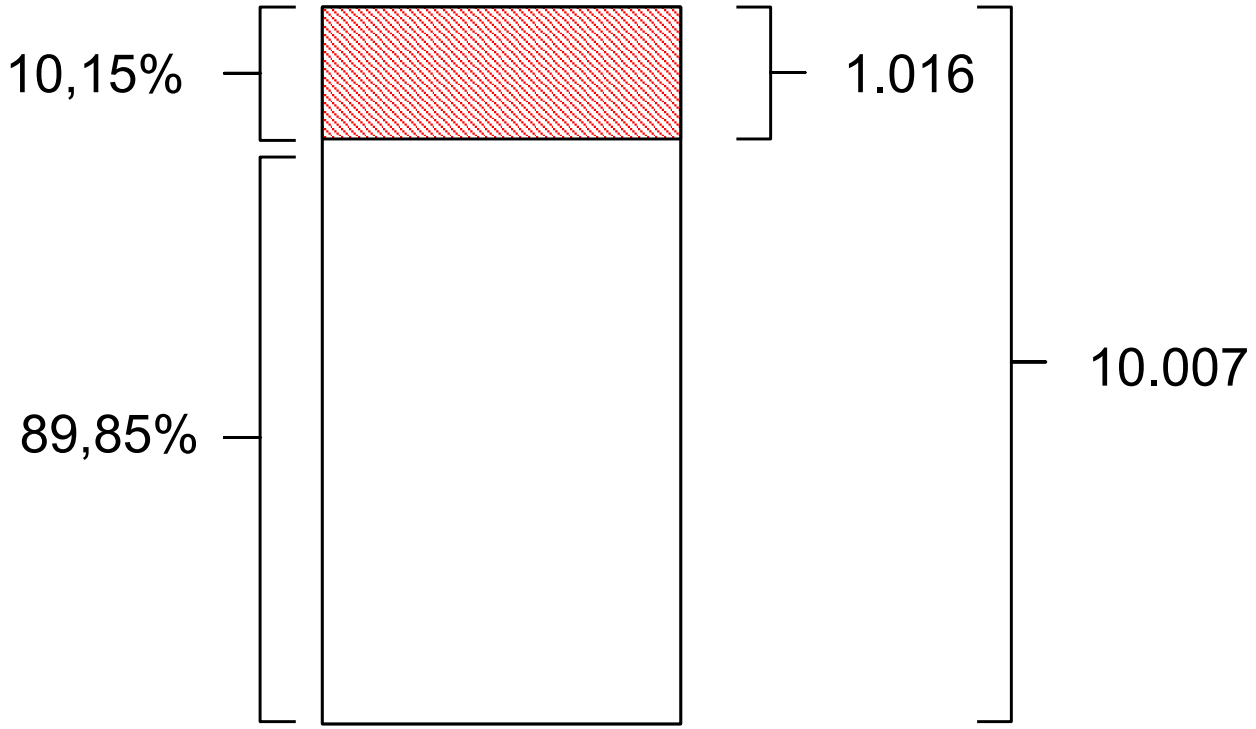
Rango *convert (String t)*
`[8*0, 8*127] = [0, 1.016]`

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 <= 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	$80*27^2$	58.320
L	$76*27^1$	2.052
A	$65*27^0$	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.
  - "**PLA**NE", "**PLA**NING", "**PLA**STIC", 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).
      - $32 = 2^5$.

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

DEBATE: ¿Cómo reducir el número de llamadas a la función potencia?

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	$80*32^4$	83.886.080
L	$76*32^3$	2.490.368
A	$65*32^2$	66.560
N	$78*32^1$	2.496
E	$69*32^0$	69
Total		**86.445.573**

86.445.573 % 10.007 = 5.107

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

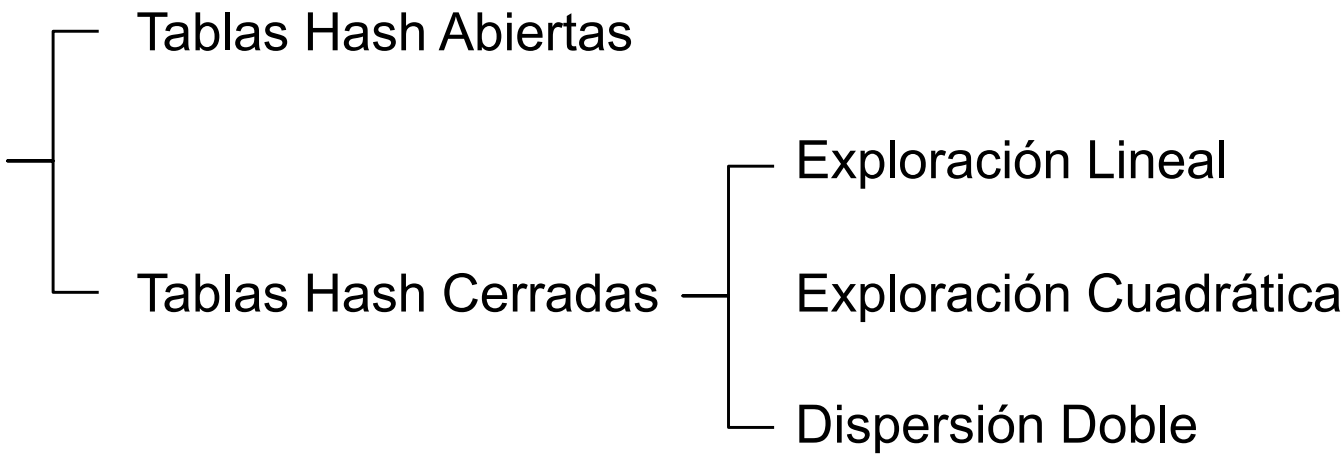
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

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)**

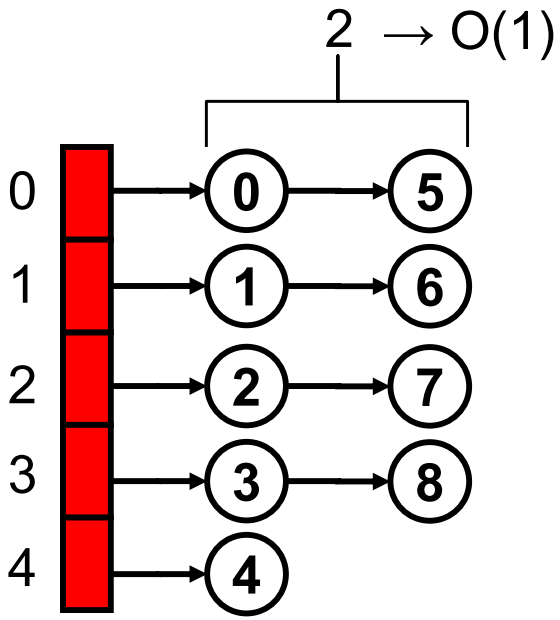
```
public class HashTable<T>
{
 private int B = 10007;
 rivate ArrayList<AVLTree<T>> associativeArray;

 public HashTable(int B) {
  this.B = B;
  associativeArray = new ArrayList<AVLTree<T>>(B);

  for (int i=0; i<associativeArray.size(); i++)
   associativeArray.add(new AVLTree<T>());
 }
}
```

Tablas Hash Abiertas

add	O(n/B) → O(1)

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

find() y remove() son análogas

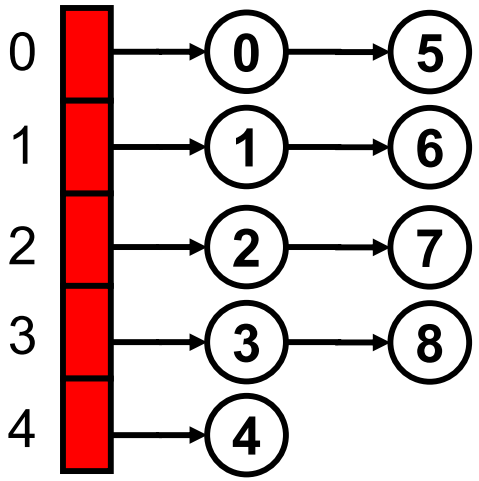



```
for (int i=0; i< 9; i++)
 table.add(new Integer(i), i);
```

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 <= 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.

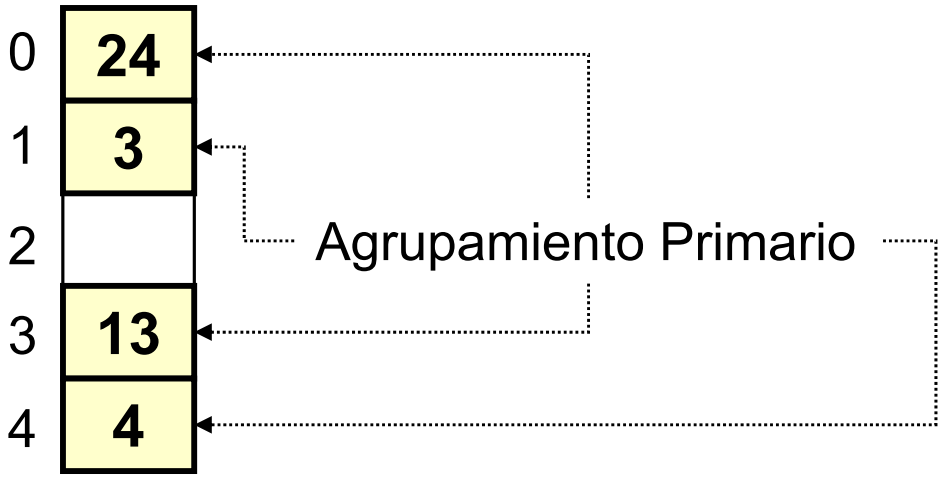
HashTable class

```
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…



add(4) → [4 + 0] % 5 = 4

add(13) → [13 + 0] % 5 = 3

add(24) → [24 + 0] % 5 = 4

add(24) → [24 + 1] % 5 = 0

add(3) → [3 + 0] % 5 = 3

add(3) → [3 + 1] % 5 = 4

add(3) → [3 + 2] % 5 = 0

add(3) → [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

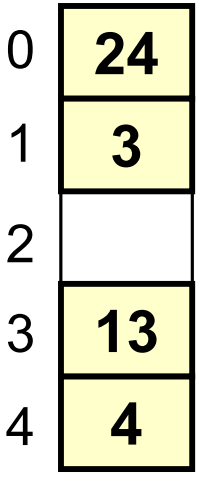
- Se recomienda usar LF <= 0,5
  - B debería ser al menos el doble de n.
  - El incremento en el consumo de memoria es notable.

En las tablas hash abiertas la recomendación es LF <= 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.



delete(24) → [24 + 0] % 5 = 4

delete(24) → [24 + 1] % 5 = 0

find(3) → [3 + 0] % 5 = 3

find(3) → [3 + 1] % 5 = 4

find(3) → [3 + 2] % 5 = 0

El acceso a la clave 3 se ha perdido

Tablas Hash Cerradas

Borrado Perezoso (Lazy deletion)

HashTable class

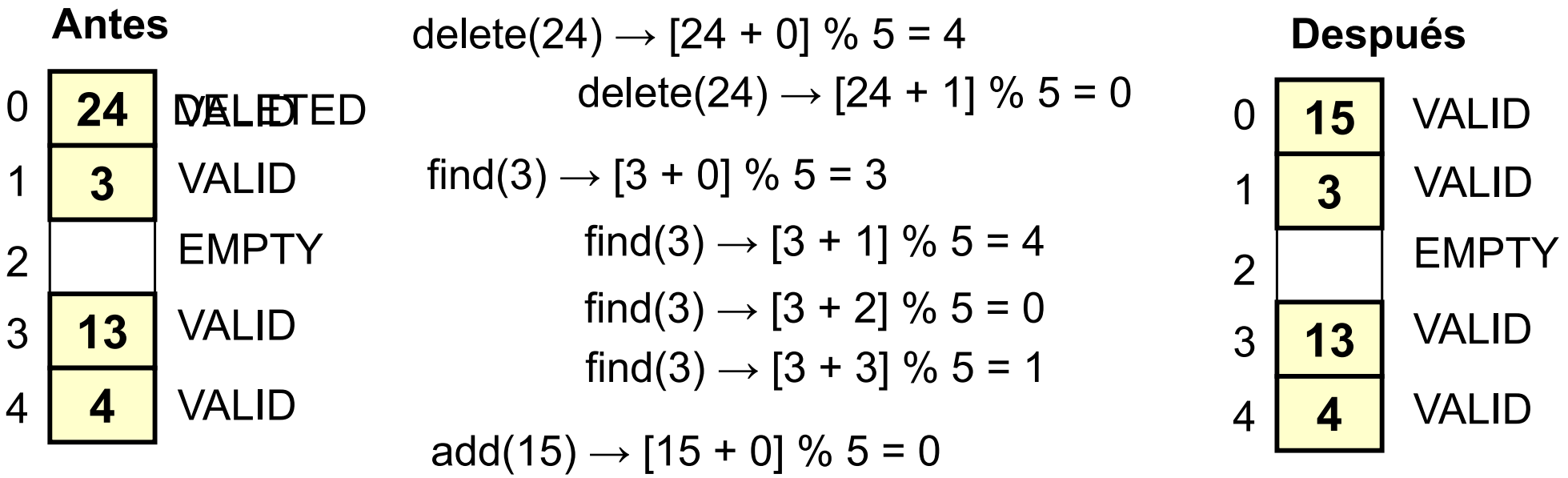
```
public class HashNode <T>
{
    public final static byte EMPTY   = 0;
    public final static byte VALID   = 1;
    public final static byte DELETED = 2;

    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;
}
```



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**

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

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

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

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

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

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

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

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 <= 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**

add(4) → [4 + 0*(3 – 4%3)] % 5 = 4

add(13) → [13 + 0*(3 – 13%3)] % 5 = 3

add(24) → [24 + 0*(3 – 24%3)] % 5 = 4

add(24) → [24 + 1*(3 – 24%3)] % 5 = 2

add(3) → [3 + 0*(3 – 3%3)] % 5 = 3

add(3) → [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 redispersión recupera un LF aceptable **trasladando** 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**

add(24) → [24 + 0^2] % 11 = 2

add(3) → [3 + 0^2] % 11 = 3

add(13) → [13 + 0^2] % 11 = 2

add(13) → [13 + 1^2] % 11 = 3

add(13) → [13 + 2^2] % 11 = 6

add(4) → [4 + 0^2] % 11 = 4

O(n)

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

DEBATE: ¿Cuál es la complejidad Temporal del algoritmo?

Redispersión

Activación de la Redispersión

- La redispersión 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 redispersión inversa
Abierta	0,33
Cerrada	0,16

Apéndice A

Para saber más

Saber más: Grafos

PLAYGROUND

- Consulte la entrada para el **Algoritmo de Dijkstra** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Ponga especial atención a como el uso de **Colas de Prioridad** puede afectar a la complejidad temporal del algoritmo.
    - La estructura de datos *Colas de Prioridad* será tratada cuando se analicen las Estructuras de Datos Jerárquicas.

Saber más: Grafos

PLAYGROUND

- Consulte la entrada para el **Algoritmo de Floyd-Warshall** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Averigüe que quiere decir que el algoritmo utilice *Memoria Cuadrática*.
    - Preste atención al tratamiento que reciben los **Ciclos Negativos** y como pueden ser detectados por el algoritmo.

Saber más: Grafos

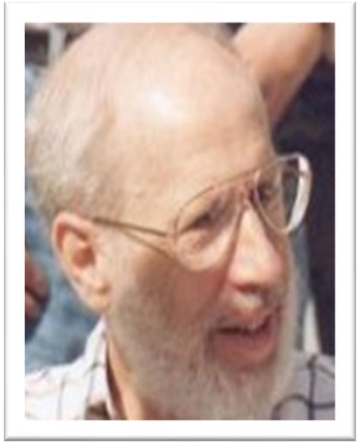
PLAYGROUND

- Consulte la entrada para el **Algoritmo de Prim** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Ponga especial atención a la demostración de por qué el algoritmo realmente funciona.

Saber más: Grafos

PLAYGROUND

- El problema del Árbol Libre Abarcador de coste mínimo también fue resuelto por el estadounidense Joseph Kruskal.
- Consulte la entrada para el **Algoritmo de Kruskal** en la Wikipedia
  - Estudie **cuidadosamente** todo el contenido de la entrada.
    - Preste especial atención a las diferencias entre el Algoritmo de Kruskal y el Algoritmo de Prim.



Joseph Kruskal (Wikipedia)

Saber más: Árboles

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**.

Saber más: Árboles

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 códijo en Java.
    - ¿Qué quiere decir la expresión *if ((altura(insertar(R1,I)) – 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?

Saber más: Árboles

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.

Saber más: Montículos Binarios

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(logn) en la inserción de elementos.

Saber más: Tablas Hash

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*.

Apéndice B

Referencias

Teoría de Grafos

AHO, A; HOPCROFT, J; ULLMAN, D; (1988) *Estructuras de Datos y Algoritmos*. Addison-Wesley Iberoamericana. México [Cap 9].

JOYANES AGUILAR, Luis; ZAHONERO MARTÍNEZ, Ignacio; (1998) *Estructura de Datos: Algoritmos, Abstracción y Objetos*. Mc Graw Hill. ISBN: 84-481-2042-6. [Cap 14.]

ORTEGA F., Maruja; (1988) *Grafos y Algoritmos*. Universidad Metropolitana, Oficina Metrópolis.

WEISS, Mark Allen; (2000*) Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN 84-7829-035-4. [Cap 14.].

WEISS, Mark Allen; (1995) *Estructuras de Datos y Algoritmos* Addison-Wesley Iberoamericana. ISBN 0-201-62571-7. [Cap 9.].

Estructuras de Datos Jerárquicas

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.

Tablas Hash

BRASSARD G.; BRATLEY, P.; (1997) *Fundamentos de Algoritmia*. Prentice Hall. ISBN: 84-89660-00-X. [Cap. 5].

COLLADO M., MORALES R. y MORENO J. (1987) Estructuras de datos. Realización en Pascal. Ed. Díaz de Santos, 1987.

WEISS, Mark Allen; (2000) *Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN84-7829-035-4. [Cap. 19].

WEISS, Mark Allen (1995) *Data Structures and Algorithm Analysis*. Addison-Wesley Iberoamericana. [Cap. 5].