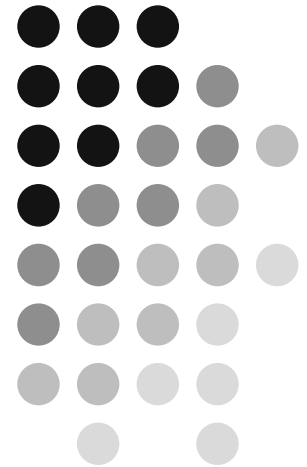


Grafos: Árboles abarcadores mínimos II

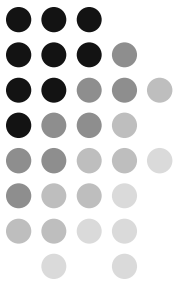


UNIVERSIDAD
DE LOS ANDES

Diseño y Análisis de Algoritmos
Cátedra de Programación
Carrera de Ingeniería de Sistemas
Prof. Isabel Besembel Carrera



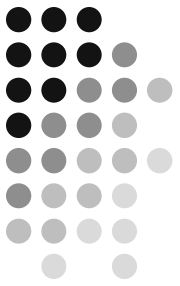
Colas por prioridad



➤ Características:

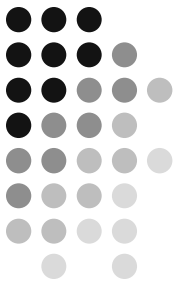
- ❖ Su nombre viene de una aplicación de Sistemas Operativos: el mantenimiento de las colas internas de procesos, donde esos procesos son manejados según su prioridad asignada.
- ❖ Es un conjunto de entradas
- ❖ Cada entrada en la cola es un par [clave, valor]
- ❖ Clave: es un campo especial para reconocer la entrada y puede tener un valor repetido en el conjunto de entradas (clave secundaria)
- ❖ Las claves están siempre ordenadas obedeciendo a un orden total
- ❖ Los valores asociados a las claves se pueden actualizar, pero no las claves
- ❖ Normalmente se implementan con montículos

Clase ColasxPrioridad



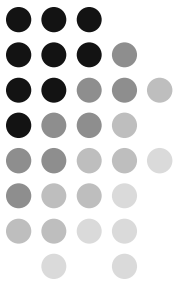
- Las operaciones más importantes en un TDA de colas por prioridad se refieren a aquellas que permiten repetidamente seleccionar el elemento de la cola de prioridad que tiene como clave el valor mínimo (máximo).
- Esto conlleva a que una cola por prioridad P debe soportar las siguientes operaciones:
 - ❖ inserta(ent)
 - ❖ min()
 - ❖ extMin()
 - crea()
 - union()
 - destruye()

Montículos binarios

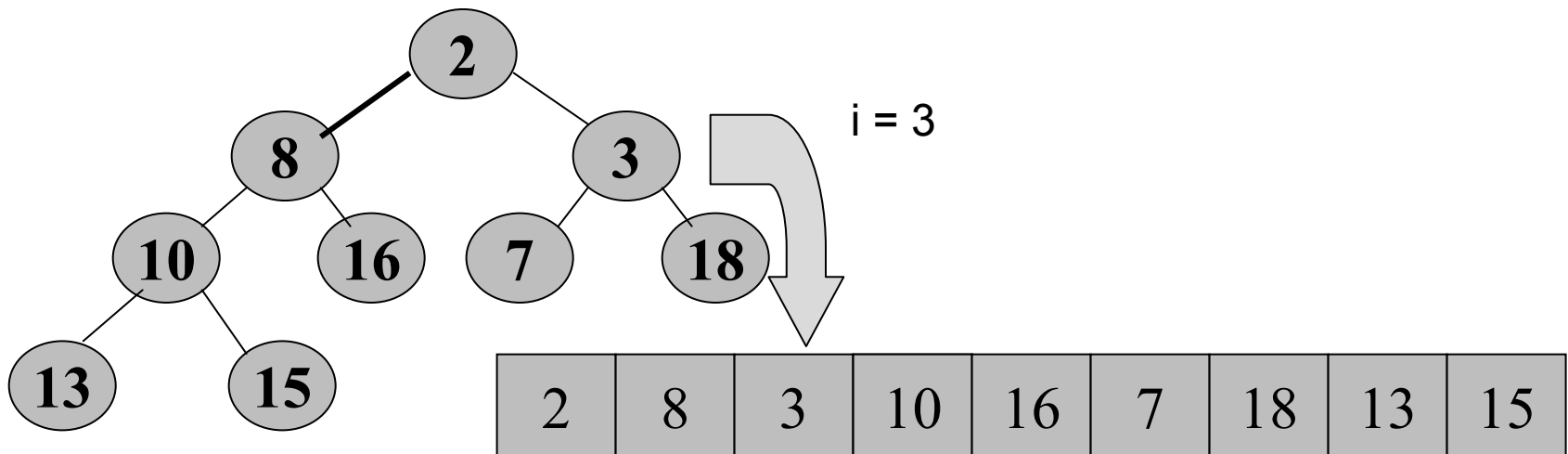


- Implementaciones de un TDA de Colad de Prioridad
 - ❖ Árboles equilibrados (AVL, ROJO y NEGRO)
 - ❖ Montículos Binarios
 - ❖ Montículos a la izquierda
 - ❖ Montículos oblicuos
 - ❖ Colas binomiales, colas binomiales perezosas
 - ❖ Colas de Fibonacci
- Un montículo binario (o simplemente montículo) es un árbol binario semicompleto en el que el valor de la clave almacenada en cualquier nodo es menor o igual que los valores claves de sus hijos
- Propiedad de ordenamiento parcial: la clave almacenada en cualquier nodo es menor o igual que los valores claves de sus hijos.

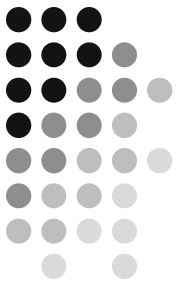
Montículos binario



- Ventajas: el hecho de ser semicompleta hace que sea posible una representación secuencial
- Si un nodo esta almacenado en la posición i .
 - ❖ Su hijo izquierdo si existe, se encuentra en la posición $2i$.
 - ❖ Su hijo derecho si existe, se encuentra en la posición $2i+1$.
 - ❖ El padre en la posición $i/2$

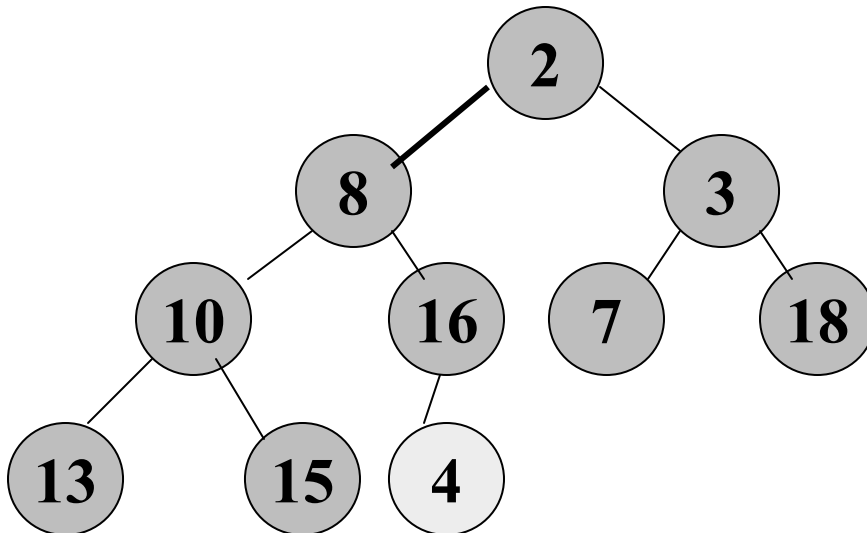


Montículos binarios



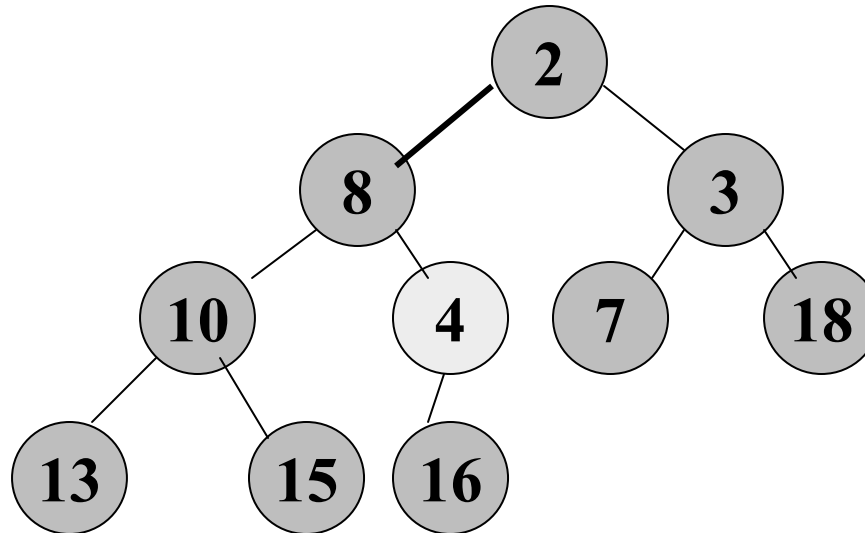
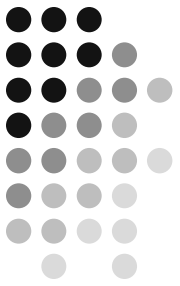
Inserta(ent):

- El nodo se añade como una hoja extrema creciendo de izquierda a derecha ($n+1$). (garantiza la propiedad de forma)
- Se garantiza la propiedad de ordenamiento parcial



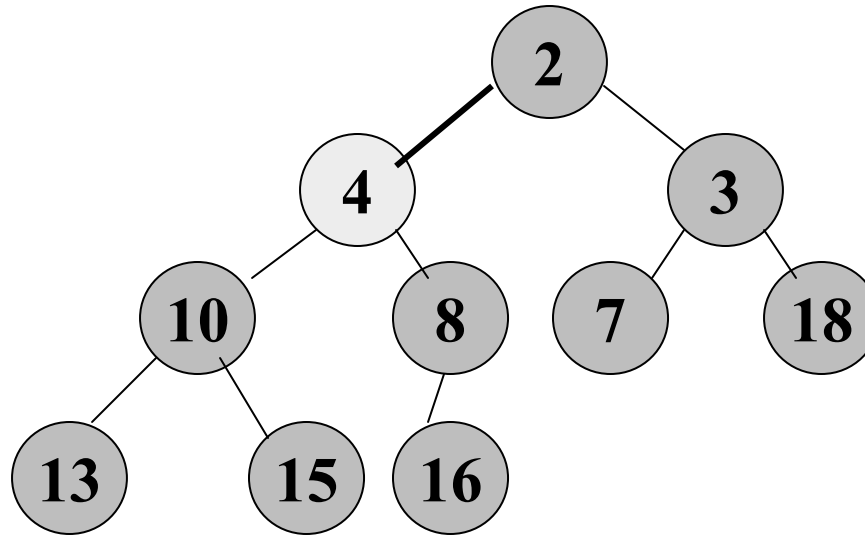
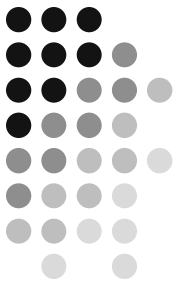
2	8	3	10	16	7	18	13	15	4
---	---	---	----	----	---	----	----	----	---

Montículos binarios



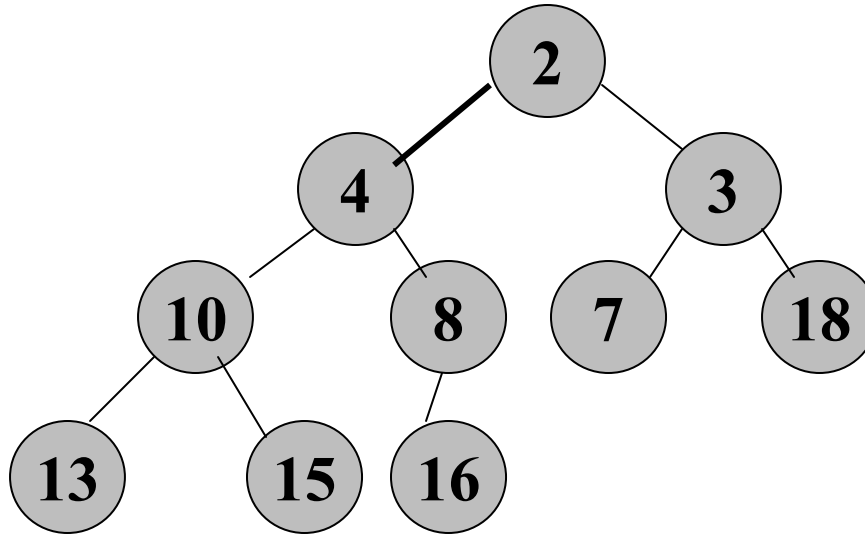
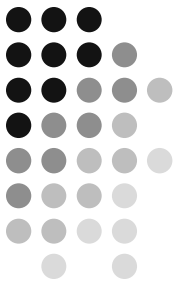
2	8	3	10	4	7	18	13	15	16
---	---	---	----	---	---	----	----	----	----

Montículos binarios

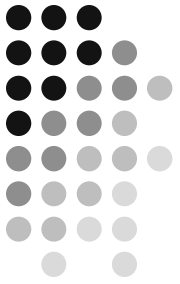


2	4	3	10	8	7	18	13	15	16
---	---	---	----	---	---	----	----	----	----

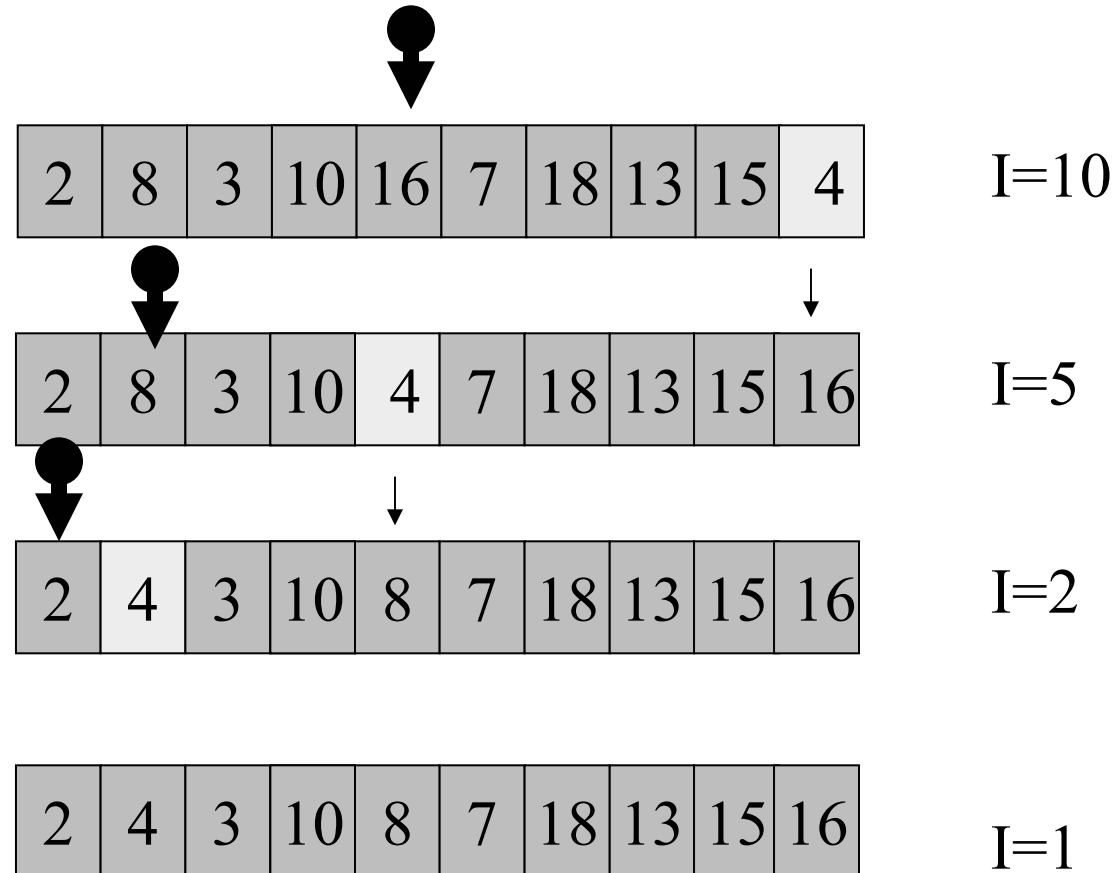
Montículos binarios



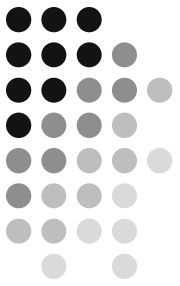
2	4	3	10	8	7	18	13	15	16
---	---	---	----	---	---	----	----	----	----



Montículos binarios

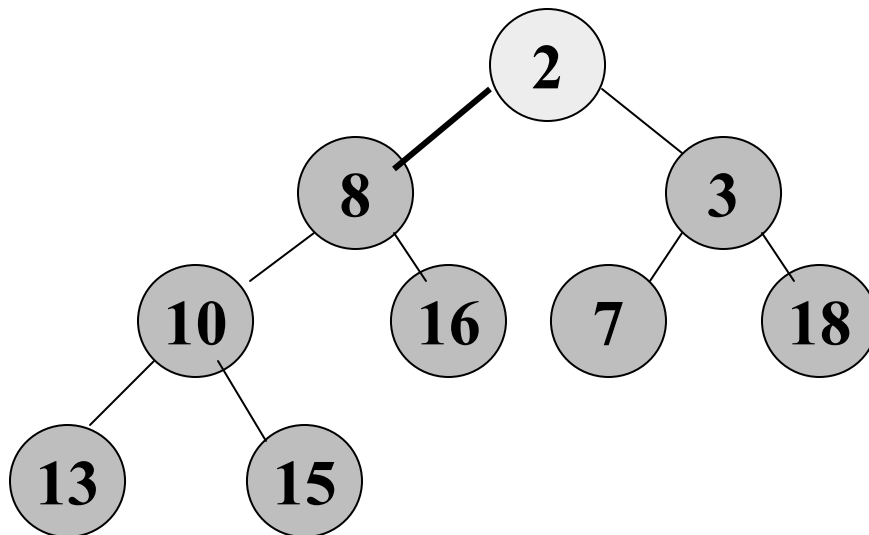


Montículos binarios



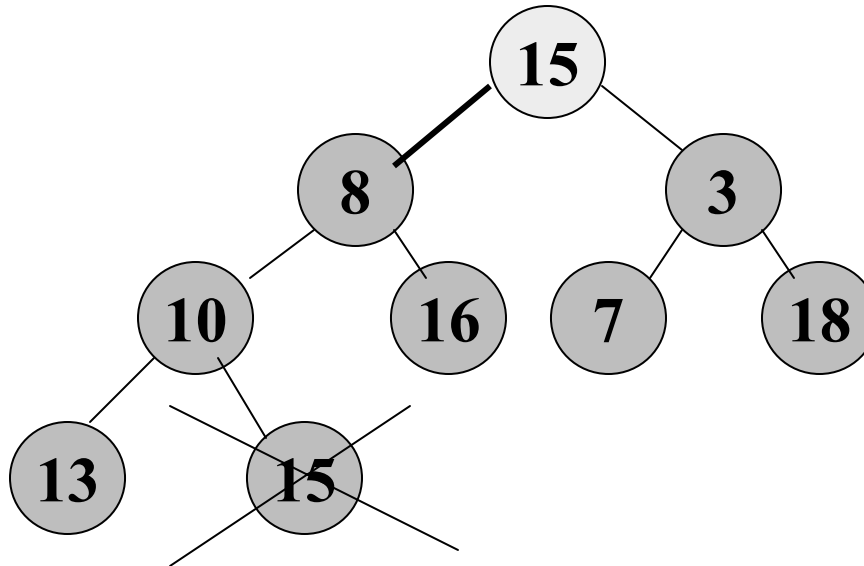
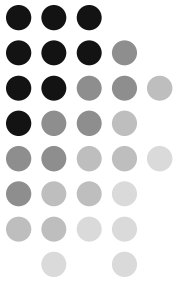
extMin(): se elimina el elemento de clave mínima, es decir el elemento de la raíz.

Se debe garantizar la propiedad de ordenamiento parcial



2	8	3	10	16	7	18	13	15
---	---	---	----	----	---	----	----	----

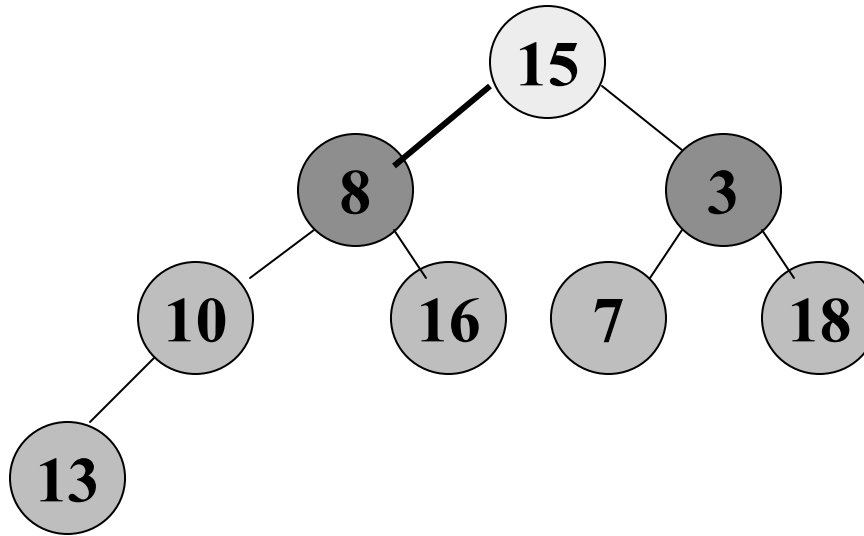
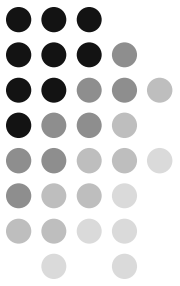
Montículos binarios



15	8	3	10	16	7	18	13
----	---	---	----	----	---	----	----

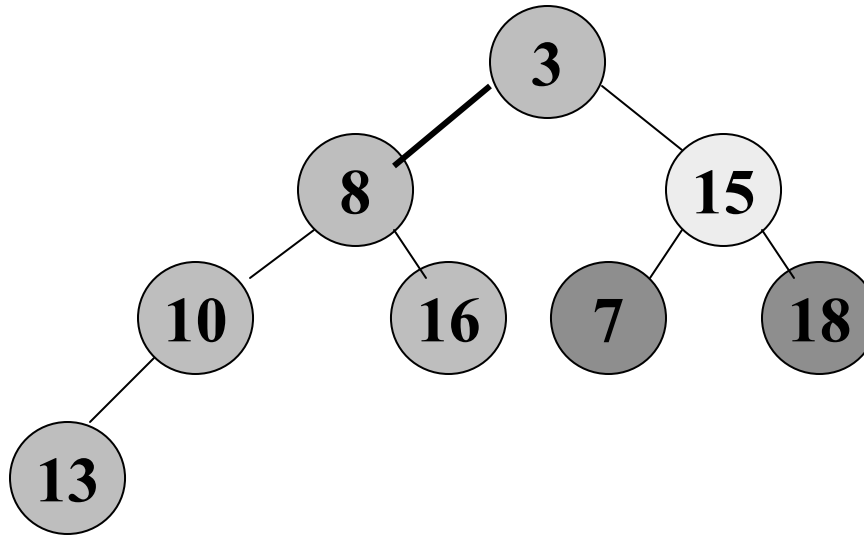
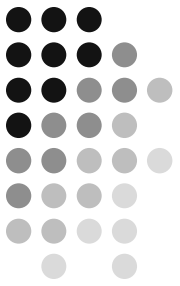
$n=n-1$

Montículos binarios



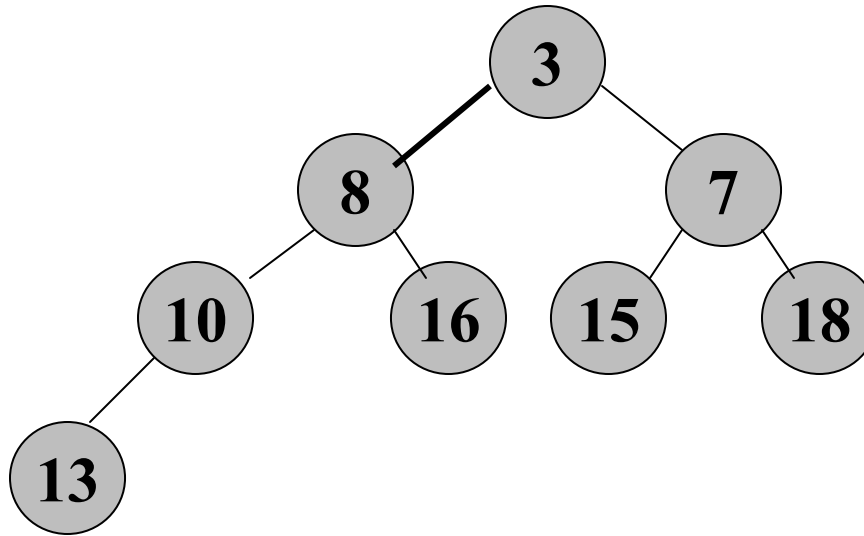
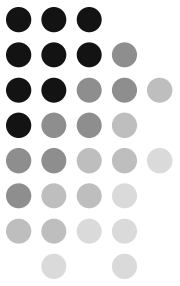
15	8	3	10	16	7	18	13
----	---	---	----	----	---	----	----

Montículos binarios

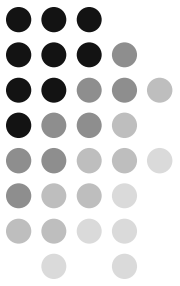


3	8	15	10	16	7	18	13
---	---	----	----	----	---	----	----

Montículos binarios



3	8	7	10	16	15	18	13
---	---	---	----	----	----	----	----



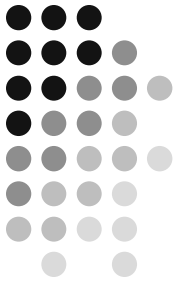
Montículos binarios

- En la operación de inserción es necesario realizar un **flotar** (filtrado ascendente) del nodo insertador para asegurar la propiedad de forma.
- En la operación de eliminación es necesario realizar un **hundir** (filtrado descendente) del nodo insertador para asegurar la propiedad de forma.

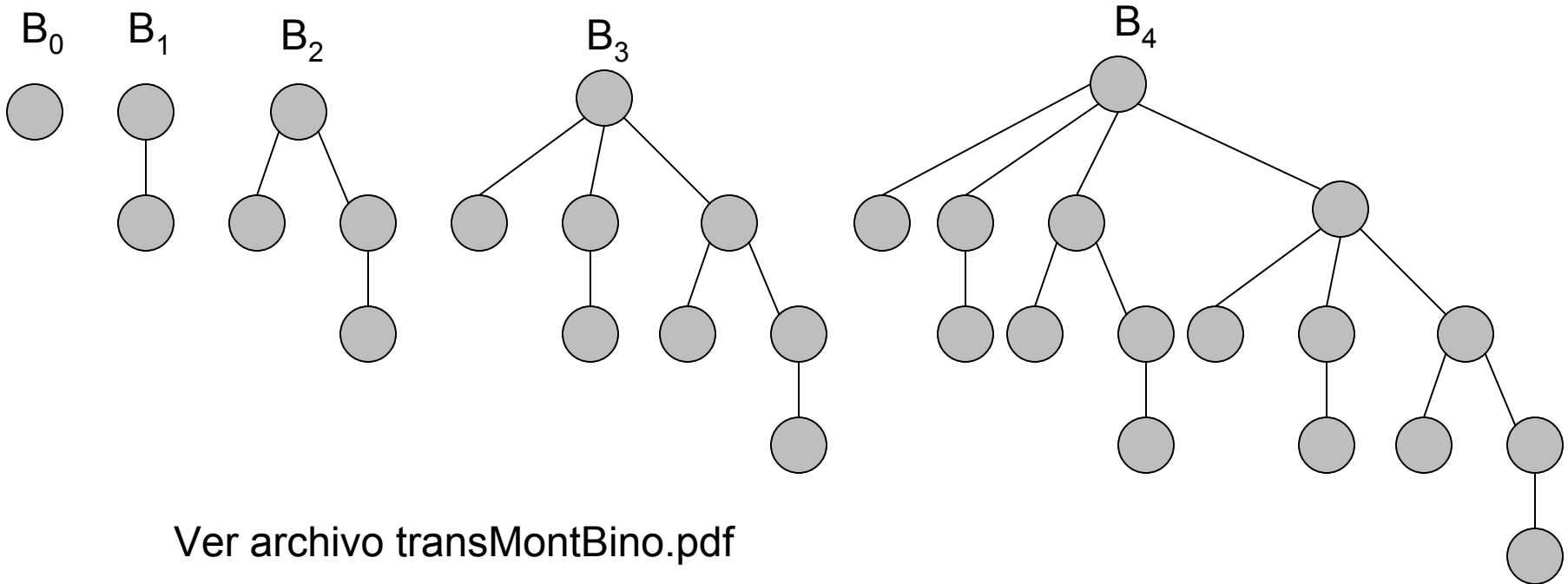
Operación	Montículo binario $W(n)$	Montículo binomial $W(n)$	Montículo Fibonacci Amort.
crear()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
inserta()	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
min()	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
extMin()	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
union()	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
decreClave()	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
elimina()	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$



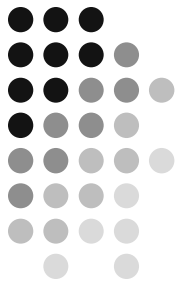
Árboles binomiales



- El i -ésimo árbol binomial B_i , con $i \geq 0$, es aquel que consta de un nodo raíz con i hijos, donde el j -ésimo hijo con $1 \leq j \leq i$, es a su vez un árbol binomial B_{j-1} .



Ver archivo [transMontBino.pdf](#)



Montículos de Fibonacci

Fredman
y Tarjan,
1987

Es un montículo mezclable conformado por una colección de árboles, pero sin relación de orden entre ellos.

Se recomiendan cuando el número de `extMin()` y `elimina()` es menor en relación al número de ocurrencias de las otras operaciones.

Diferencia con los binomiales: los de fibonacci tienen una estructura menos rígida cuyo mantenimiento se retrasa hasta que sea conveniente realizarlo, permitiendo una menor complejidad en tiempo.

NodoMont:

p clave grado marca hijo izq der

p: Apuntador al nodo padre

clave: Clave del nodo

grado: Número de hijos del nodo

marca: Es verdadero si el nodo ha perdido un hijo durante el periodo en que él fue hecho hijo de otro nodo. Los nodos recién creados tienen marca Falso.

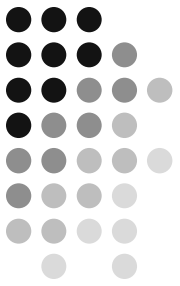
hijo: Apuntador a un nodo hijo

izq: Apuntador al nodo hermano a la izquierda

der: Apuntador al nodo hermano a la derecha

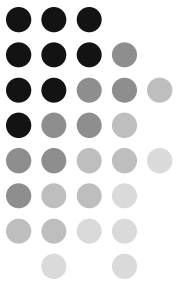
Ver archivo
[transMontFibo.pdf](#)

Algoritmo de Prim



- Es un algoritmo incremental de tipo II.
- El árbol formado es un árbol simple, que se comienza a formar con un nodo arbitrario y crece hasta tener todos los nodos en X .
- Todas las aristas sumadas a X son seguras para X .
- El árbol se aumenta en cada etapa con una arista que contribuye con la mínima cantidad posible a los pesos del árbol.
- Se implementa con una cola por prioridad para seleccionar fácilmente la nueva arista a ser incluida en X .

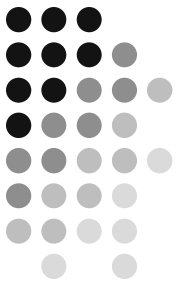
Algoritmo de Prim



➤ Características:

- ❖ El árbol abarcador mínimo crece en forma natural, desde un nodo seleccionado como raíz
- ❖ En cada fase se añade una arista hasta que se alcanzan todos los nodos
- ❖ Sea B el conjunto de nodos y T el conjunto de aristas
- ❖ Inicialmente B tiene un único nodo arbitrario y T está vacío
- ❖ En cada paso, busca la arista más corta posible (u, v) tal que $u \in B$ y $v \in N$, se añade v a B y (u, v) a T
- ❖ Se continúa mientras $B \neq N$, las aristas en T siempre forman un árbol abarcador mínimo

Algoritmo de Prim



➤ Algoritmo genérico

Prim(Grafo: g):Conjunto

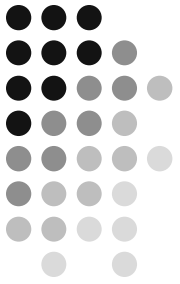
1 $T = \{\}$, $B = \{\text{un nodo cualquiera de } g\}$

2 ($B \neq N$) [buscar $e = (u, v)$ de longitud mínima / $u \in B$ y $v \in N$

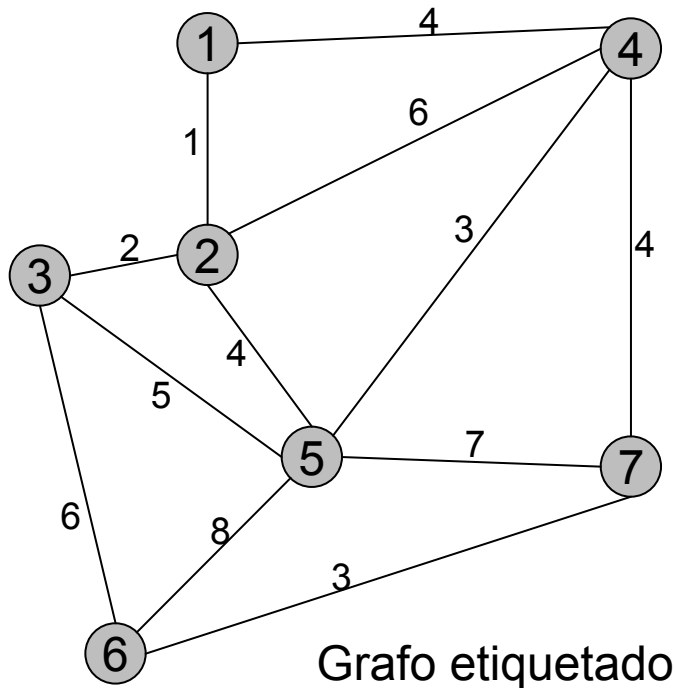
$$T = T \cup \{e\}$$

$$B = B \cup \{v\} \quad]$$

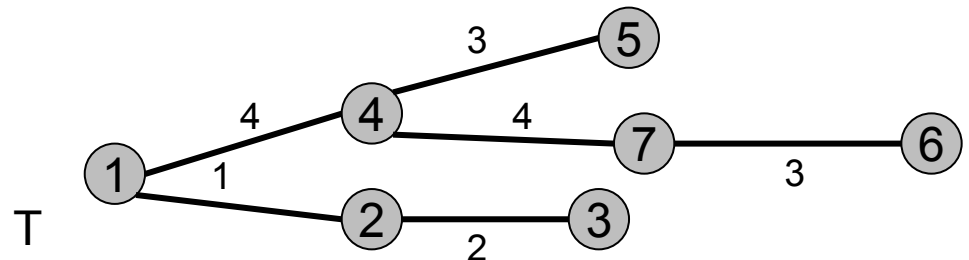
3 regrese T

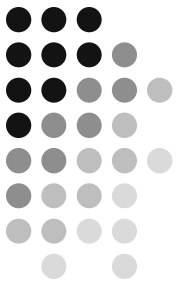


Algoritmo de Prim



Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}
3	(1, 4)	{1, 2, 3, 4}
4	(4, 5)	{1, 2, 3, 4, 5}
5	(4, 7)	{1, 2, 3, 4, 5, 7}
6	(7, 6)	{1, 2, 3, 4, 5, 7, 6}





- El algoritmo de Prim halla un árbol abarcador mínimo

Demostración: por inducción sobre el número de aristas en T . Se muestra que si T es prometedor entonces sigue siendo prometedor en cualquier paso del algoritmo cuando se le añade una arista adicional.

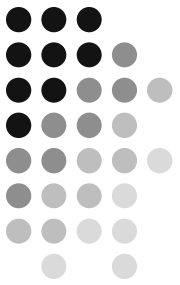
Base: T vacío es prometedor

Inductiva: suponga que T es prometedor inmediatamente antes que el algoritmo añada una nueva arista $e=(u, v)$. Entonces:

- B es un subconjunto estricto de N , pues no incluye a v
- T es un conjunto prometedor
- e es una de las aristas más cortas que salen de B

Se cumplen las condiciones del lema CP y se concluye que $T \cup \{e\}$ es también prometedor

T es prometedor en todas las fases del algoritmo, T es una solución óptima del problema.



Algoritmo de Prim

26/11/98

PrimAEM(Nodo: r, Arreglo[n]De Nodo: &clave): Arreglo[n]De Nodo

{pre: $n > 0 \wedge r \in N$ }

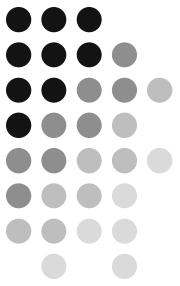
{pos: $n > 0 \wedge G' = G \wedge$ }

<p>1 [c.entrar(v, MV)] $v \in N$ 2 clave(r), padre(r) = 0, Nulo 3 (\negc.vaciaCola()) [u = c.min() [Si ($v \in c \wedge w(u, v) < clave(v)$) ent. padre(v) = u clave(v) = w(u, v) fsi] $\forall v \in u.listaAdyacencia$ 4 regrese padre</p>	<p>-c. ColaPrioridad[X]. Cola de nodos. -clave: Arreglo[n]De Nodo. Variable auxiliar con los pesos. -entrar(), vaciaCola(), min(). Definidas en la clase ColaPrioridad. -padre. Arreglo[n]De Nodo. Variable auxiliar con el árbol de expansión mínima.</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

$$T(n) = O(A + N \lg N)$$



Análisis del algoritmo de Prim



- La clase ColaPrioridad debe ser implantada con montículos de Fibonacci para poder tener el tiempo asintótico dado.
- Si se implanta con un montículo binario su complejidad es igual a la del algoritmo de Kruskal.