

RESUMEN-EDNL-TEORIA.pdf



QuesoViejo_



Estructuras de Datos no Lineales



2º Grado en Ingeniería Informática



Escuela Superior de Ingeniería
Universidad de Cádiz

**QUIERES
CONSEGUIR
15E??**

→ TRÁENOS A TU
CRUSH DE APUNTES ❤
ANTES DE QUE
LOS QUEME 🔥



WUOLAH

QUIERES
CONSEGUIR
15€??

TRÁENOS A TU
CRUSH DE APUNTES

ANTES DE QUE
LOS QUEME



TEORÍA EDNL



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Reservados todos los derechos.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

QuesoViejo_

WUOLAH

WUOLAH

Parte 1 Árboles Binarios

Definiciones generales árboles: (las mismas para binario, general...)

Colección de elementos almacenados en nodos entre los que existe una relación de paternidad (definiendo una jerarquía)

Un árbol es una estructura recursiva en sí mismo.

Grado de un nodo: Número de hijos de ese nodo

Grado de un árbol: Máximo de los grados de sus nodos.

Camino: Sucesión de nodos n_1, n_2, \dots, n_k tal que

n_i es padre de n_{i+1} . *Ojo: Solo hay caminos de padres a hijos.

Longitud del camino: N° nodos - 1. Longitud del camino de un nodo a sí mismo es cero.

Ancestros y descendientes: Si existe un camino de A a B, entonces A es ancestro/antecesor (padre) de B y B es descendiente (hijo) de A.

Un ancestro/descendiente de un nodo distinto de sí mismo se llama ancestro/descendiente propio.

Raíz: Es el único nodo de un árbol que no tiene padre.

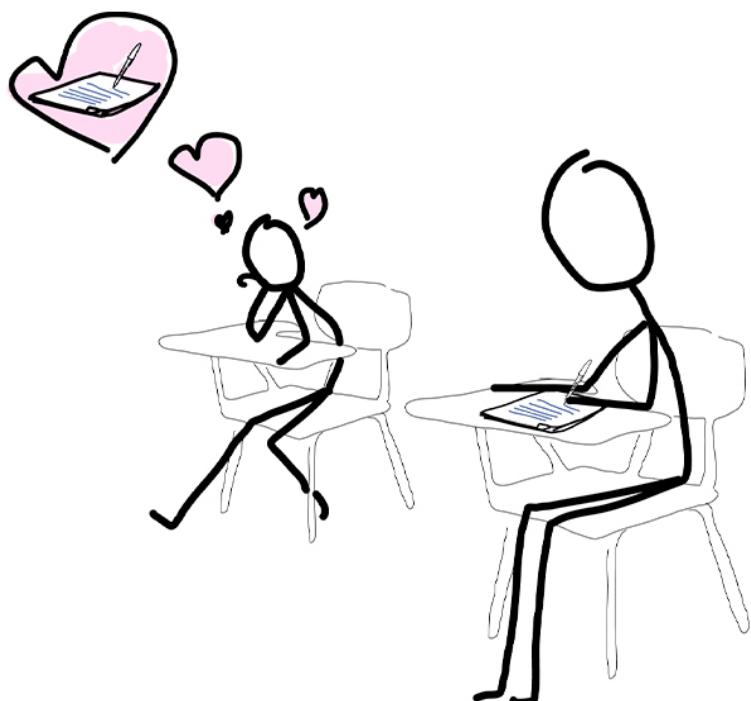
Hoja: Nodo del árbol sin descendientes propios.

Subárbol: Conjunto formado por un nodo y todos sus descendientes.

Rama: Camino que termina en un nodo hoja

QUIERES
CONSEGUIR
15€ ??

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME

si consigues que suba apuntes, te llevas 15€ + 5
Wuolah Coins para los sorteos

WUOLAH

Altura de un nodo: Longitud de la rama más larga que parte de dicho nodo.

Altura de un árbol: Altura del nodo raíz

Profundidad: Longitud del único camino que hay entre la raíz y ese nodo.

Nivel: Un árbol de altura h se divide en $h+1$ niveles. En cada nivel i se encuentran los nodos de profundidad i .

Implementación vectorial Árbol:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	d	h	j	i	k	e	z	c			
padre	0	0	1	0	3	1	5	2	2	4	*	9	*	*	
hizq	1	5	7	*	9	6	*	*	*	*	11	*	*		
hder	3	2	8	4	*	*	*	*	*	*	*	*	*		

Primer opción: Al eliminar pongo * en el padre O(1) pero al insertar tengo que buscar la primera columna libre O(n)

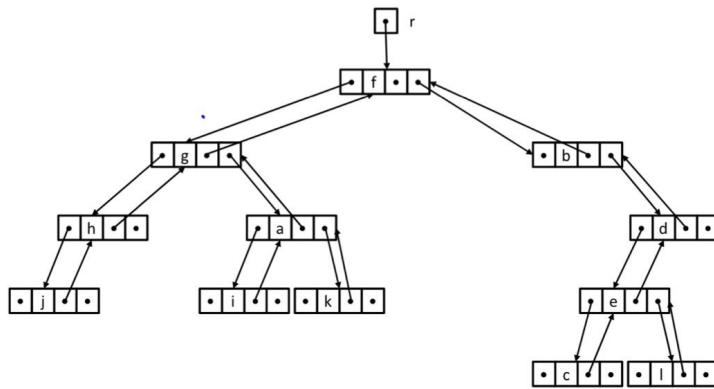
Segunda opción (al revés que la 1^a): Al eliminar, compacto de forma que no queden huecos libres (eliminar la columna 6 implica meter la 7 en la 6, la 8 en la 7...) O(n) y al insertar siempre lo hago al final O(1)

Tercera opción (mejora de la 2^a): Al eliminar, compacto moviendo únicamente el último al hueco O(1) y al insertar al final O(1). Debo llevar la cuenta de numNodos.

numNodos 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	maxNodos-1
elto	f	g	a	b	h	d	y	w							
padre	*	0	1	0	1	3	4	4							
hizq	1	4	*	*	7	*	*	*							
hder	3	2	*	5	6	*	*	*							

Implementación dinámica Aben : Muchos punteros.

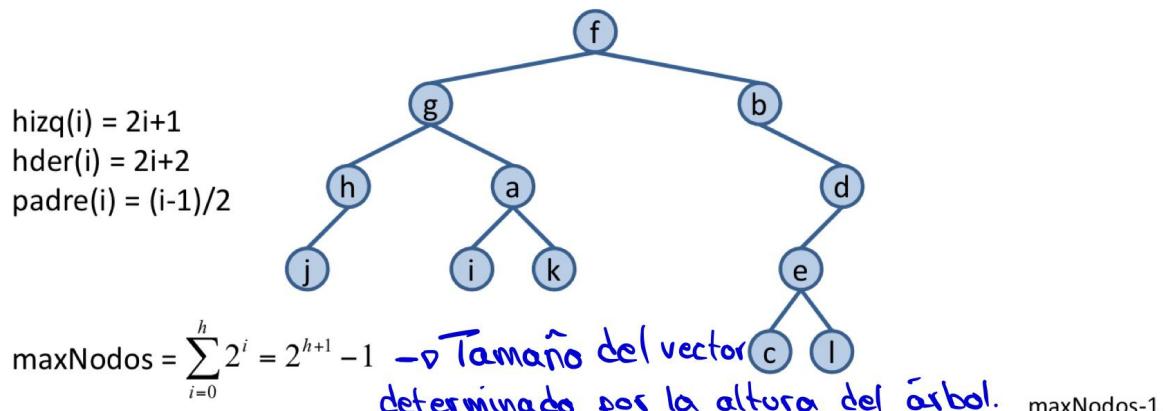


Implementación vectorial posiciones relativas :

Ventajas : No hay enlaces

Desventajas : Huecos innecesarios

Dado un nodo, sabemos exactamente dónde está su padre, sus hijos...



nodos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	25	26	27	28	29	30	maxNodos-1
	f	g	b	h	a	d	j	i	k	e						c	l						

$$h = 4 \Rightarrow \text{Maxnodos} = \sum_{i=0}^4 2^i = 2^5 - 1 = 31$$

Si juegas con fuego, te fuegas

**QUIERES
CONSEGUIR
15€??**

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Casos desfavorables: la falta de un nodo en los primeros niveles del árbol provoca muchas posiciones libres en el vector.

El más desfavorable es que el árbol sea una rama.

Será más eficiente respecto al espacio cuanto más lleno esté el árbol: Si faltan nodos son de los últimos niveles del árbol

Las posiciones libres se marcan con un valor del tipo del árbol no significativo en la aplicación (en el ejemplo @). Lo debe indicar el usuario según el tipo del árbol y el problema a resolver

nodos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	maxNodos-1	25	26	27	28	29	30
	f	g	b	h	a	@	d	j	@	i	k	@	@	e	@	@	@@	c	l	@@	@@	@@	@@

En la implementación, no confundir NODO_NULO con elto_nulo:

- **NODO_NULO**: Indica que no existe el nodo (ej: padre de raíz)
- **elto_nulo**: Indica que no existe (de momento) elemento en ese nodo.

Recuerda: Los elementos se guardan en nodos.



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Operaciones del TAD Árbol Binario :

Especificación de operaciones:

Abin ()

Post: Crea y devuelve un árbol vacío.

void insertarRaizB (const T& e)

Pre: El árbol está vacío.

Post: Inserta el nodo raíz cuyo contenido será *e*.

void insertarHijoIzqdoB (nodo n, const T& e)

Pre: *n* es un nodo del árbol que no tiene hijo izquierdo.

Post: Inserta el elemento *e* como hijo izquierdo del nodo *n*.

void insertarHijoDrchoB (nodo n, const T& e)

Pre: *n* es un nodo del árbol que no tiene hijo derecho.

Post: Inserta el elemento *e* como hijo derecho del nodo *n*.

void eliminarHijoIzqdoB (nodo n)

Pre: *n* es un nodo del árbol.

Existe *hijoIzqdoB(n)* y es una hoja.

Post: Destruye el hijo izquierdo del nodo *n*.

void eliminarHijoDrchoB (nodo n)

Pre: *n* es un nodo del árbol.

Existe *hijoDrchoB(n)* y es una hoja.

Post: Destruye el hijo derecho del nodo *n*.

void eliminarRaizB ()

Pre: El árbol no está vacío y *raizB()* es una hoja.

Post: Destruye el nodo raíz. El árbol queda vacío

bool arbolVacioB () const

Post: Devuelve *true* si el árbol está vacío y *false* en caso contrario.

const T& elemento(nodo n) const

T& elemento(nodo n)

Pre: *n* es un nodo del árbol.

Post: Devuelve el elemento del nodo *n*.

nodo raizB () const

Post: Devuelve el nodo raíz del árbol. Si el árbol está vacío, devuelve *NODO_NULO*.

nodo padreB (nodo n) const

Pre: *n* es un nodo del árbol.

Post: Devuelve el padre del nodo *n*. Si *n* es el nodo raíz, devuelve *NODO_NULO*.

nodo hijoIzqdoB (nodo n) const

Pre: *n* es un nodo del árbol.

Post: Devuelve el nodo hijo izquierdo del nodo *n*. Si no existe, devuelve *NODO_NULO*.

nodo hijoDrchoB (nodo n) const

Pre: *n* es un nodo de A.

Post: Devuelve el nodo hijo derecho del nodo *n*. Si no existe, devuelve *NODO_NULO*.

Queso Viejo

Si juegas con fuego, te puedes quemar

Recorridos Árboles Binarios

Para pasar por todos los nodos, el usuario debe desarrollar una estrategia, a diferencia de otros TADs como la lista, que empezaba en la posición primera y simplemente encadenaba operaciones siguiente().

Notación Matemática

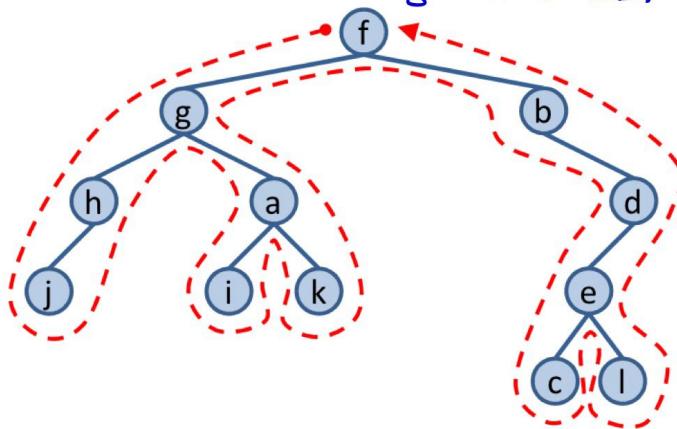
a+b	Infixo
+ab	Prefijo
ab+	Postfijo

Notación Árbol Binario

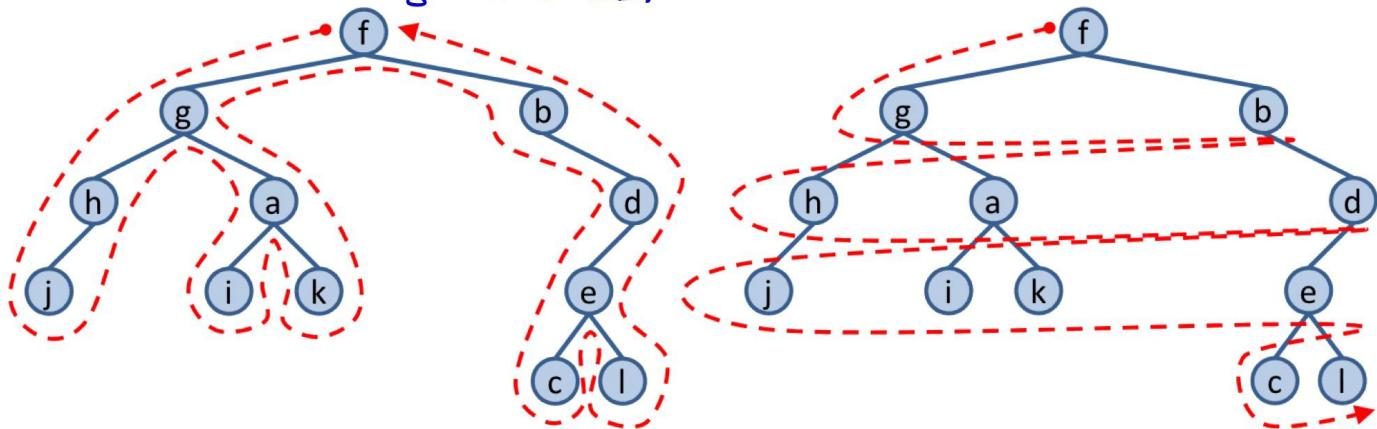
Hizq	Raiz	Hder	Inorden
Raiz	Hizq	Hder	Preorden
Hizq	Hder	Raiz	Postorden

*Nota: Al poner Hizq o Hder, se evalúa dicho hijo en Inorden, Preorden o Postorden, de la misma forma que se haya evaluado el padre. Es decir: $\text{Inorden}(n) = \begin{cases} \text{Inorden}(n.\text{hizq}) & \\ \text{Evaluar}(n) & \\ \text{Inorden}(n.\text{hder}) & \end{cases}$

En profundidad \rightarrow De cualquiera de las 3 formas vistas.

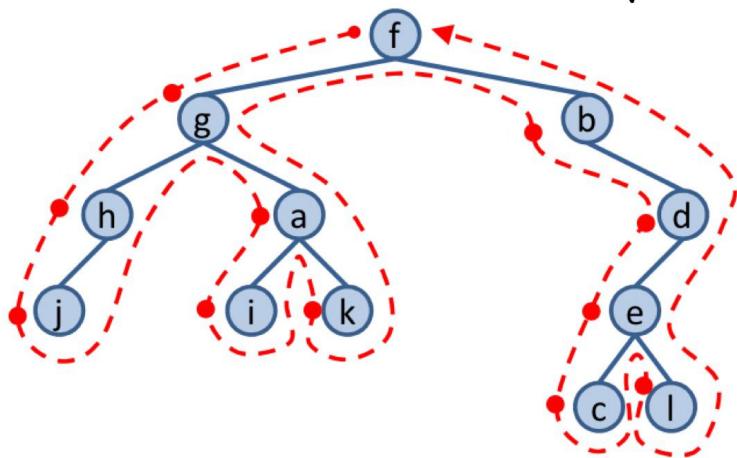


En anchura \rightarrow Solo hay 1 forma

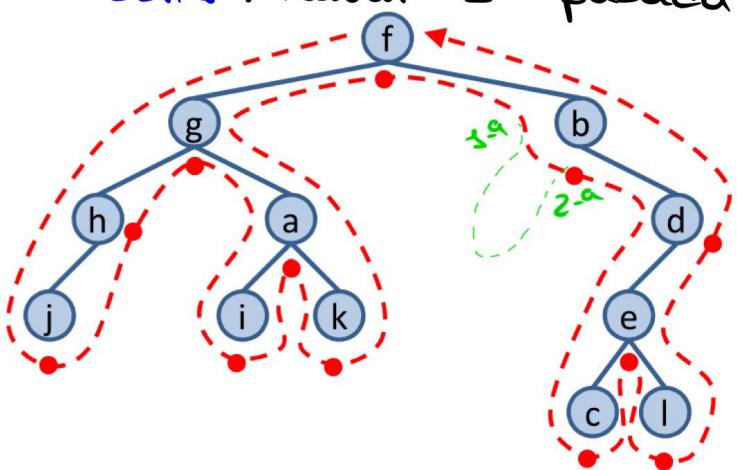


En profundidad lo único que cambia es el momento de "evaluar" el nodo según el método empleado:

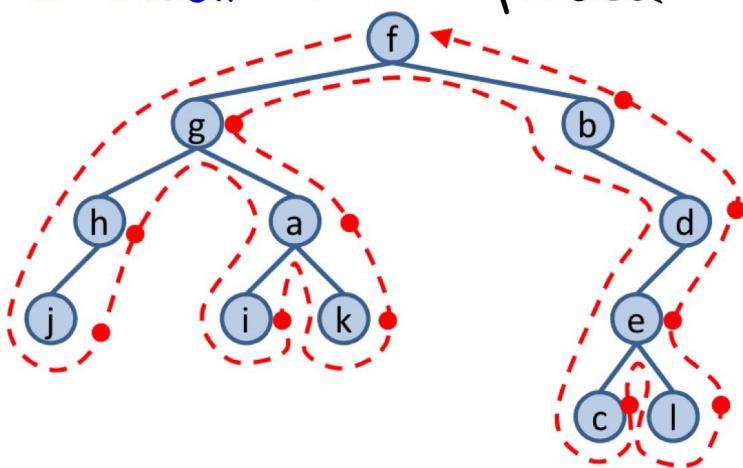
Preorden : Evaluar en la 1^a pasada



Inorden : Evaluar 2^a pasada.



Postorden : Última pasada



función típica : Destruir el árbol (hay que destruir los nodos de abajo hacia arriba)

QUIERES
CONSEGUIR
15€??

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Implementaciones :

Profundidad: Recursiva. Las 3 maneras solo cambian el orden de las llamadas recursivas y de "Evaluar"

Anchura: Con una cola donde empiezo metiendo la raíz y cada vez que "Evalve", meto los hijos.

Mientras la cola no esté vacía, saco un nodo de ahí.



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Parte 2 Árboles Generales

Árbol cuyos nodos pueden ser de cualquier grado (pueden tener cualquier número de hijos).

```
Agen(); // constructor  
void insertarRaiz(const T& e);  
void insertarHijoIzqdo(nodo n, const T& e);  
void insertarHermDrcho(nodo n, const T& e);
```

En un árbol general, los hijos de un nodo se ordenan de izquierda a derecha, siendo el primero el hijo izquierdo, el segundo el hermano derecho del primero, el tercero es hermano derecho del segundo...

En un árbol binario podía existir el hijo derecho de un nodo sin que existiera el izquierdo. En un árbol general eso no puede ocurrir.

Salvando esa diferencia, un árbol binario se podría representar como un árbol general.

Operaciones del TAD Árbol General:

Especificación de operaciones:

Agen()

Post: Construye un árbol vacío.

void insertarRaiz (const T& e)

Pre: El árbol está vacío.

Post: Inserta el nodo raíz de A cuyo contenido será e.

void insertarHijoIzqdo(nodo n, const T& e)

Pre: n es un nodo del árbol.

Post: Inserta el elemento e como hijo izquierdo del nodo n. Si ya existe hijo izquierdo, éste se convierte en el hermano derecho del nuevo nodo.

void insertarHermDrcho(nodo n, const T& e)

Pre: n es un nodo del árbol y no es el nodo raíz.

Post: Inserta el elemento e como hermano derecho del nodo n del árbol.

Si ya existe hermano derecho, éste se convierte en el hermano derecho del nuevo nodo.

void eliminarHijoIzqdo(nodo n)

Pre: n es un nodo del árbol. Existe $hijoIzqdo(n)$ y es una hoja.

Post: Destruye el hijo izquierdo del nodo n . El segundo hijo, si existe, se convierte en el nuevo hijo izquierdo de n .

void eliminarHermDrcho(nodo n)

Pre: n es un nodo del árbol. Existe $hermDrcho(n)$ y es una hoja.

Post: Destruye el hermano derecho del nodo n . El siguiente hermano se convierte en el nuevo hermano derecho de n .

void eliminarRaiz()

Pre: El árbol no está vacío y $raiz()$ es una hoja.

Post: Destruye el nodo raíz. El árbol queda vacío.

bool arbolVacio() const

Post : Devuelve `true` si el árbol está vacío y `false` en caso contrario.

const T& elemento(nodo n) const

T& elemento(nodo n)

Pre: n es un nodo del árbol.

Post: Devuelve el elemento del nodo n .

nodo raiz() const

Post: Devuelve el nodo raíz del árbol. Si el árbol está vacío, devuelve `NODO_NULO`.

nodo padre(nodo n) const

Pre: n es un nodo del árbol.

Post: Devuelve el parente del nodo n . Si n es el nodo raíz, devuelve `NODO_NULO`.

nodo hijolzqdo(nodo n) const

Pre: n es un nodo del árbol.

Post: Devuelve el hijo izquierdo del nodo n . Si no existe, devuelve `NODO_NULO`.

nodo hermDrcho(nodo n) const

Pre: n es un nodo del árbol.

Post: Devuelve el hermano derecho del nodo n . Si no existe, devuelve `NODO_NULO`.

Implementación vectorial mediante listas de hijos.

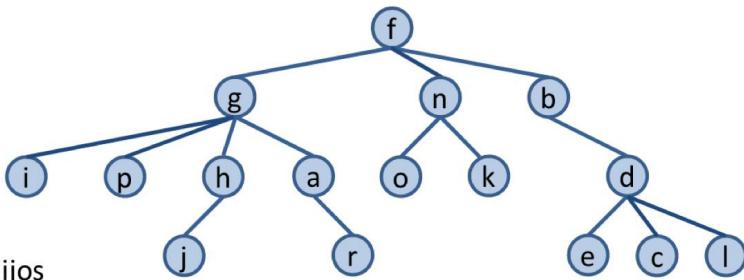
Al constructor se le pasa por parámetros el número máximo de nodos que puede almacenar.

Ojo: Ese nº de nodos se puede distribuir de cualquier manera.

QuesoViejo_

Si juegas con fuego, te fuegas

wuolah



elto padre hijos

0	f	-1	(1,3,5) ↗
1	g	0	(7,14,2,8)
2	h	1	(9)
3	n	0	(4,10)
4	o	3	()
5	b	0	(6)
6	d	5	(11,12,13)
..			
99	-	-1	-

Vector: **nodos**
numNodos
maxNodos

Parte privada del TAD

elto padre hijos

f	-1	•	•
1	•	•	•
3	•	•	•
5	•	•	•

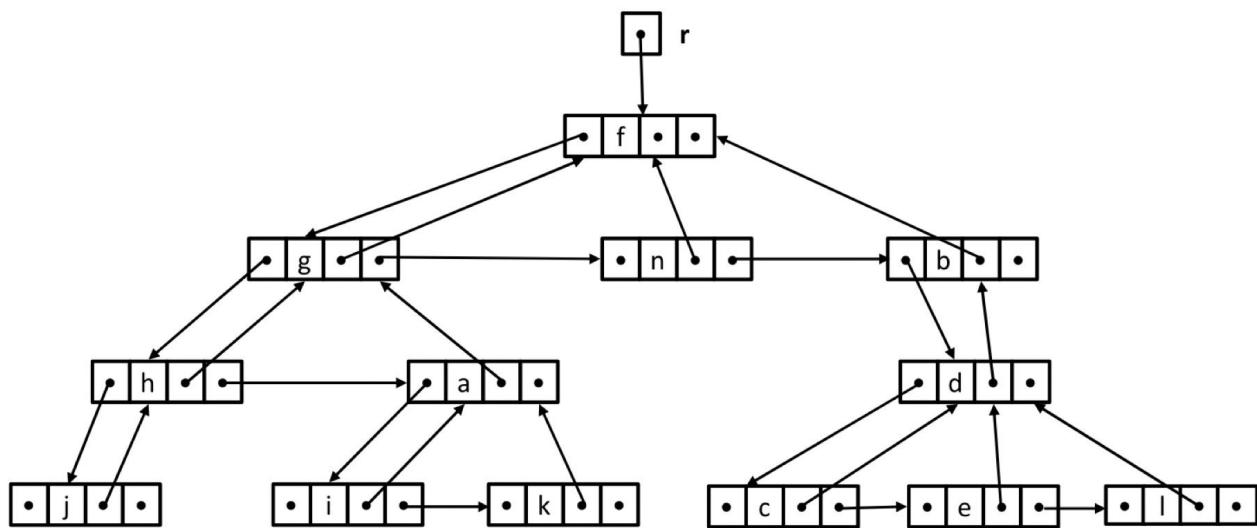
Lista Simplemente enlazada con cabecera.

Lista de nodos

En las listas de hijos se guardan las posiciones del vector donde se encuentran dichos hijos.

Implementación mediante celdas enlazadas:

En el árbol binario había punteros a: padre, hijo_{izq}, hijo_{derecho}
En árbol general: padre, hijo_{izq}, hermano_{derecho}.



QUIERES
CONSEGUIR
15€??

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Recorridos Árboles Generales

Igual que en Árboles Binarios, pero ahora hay más hijos.

Profundidad :

Preorden : Raíz - Hijos (todos)

Inorden : Hijo 1 - Raíz - Hijos (el resto)

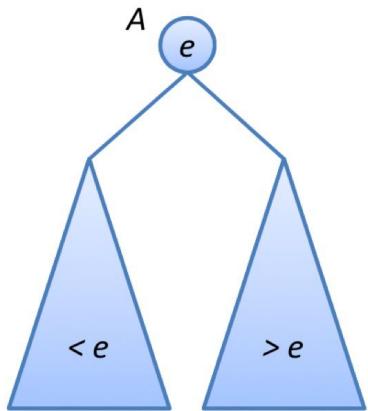
Postorden : Hijos (todos) - Raíz

Anchura : Igual, implementación con una cola.



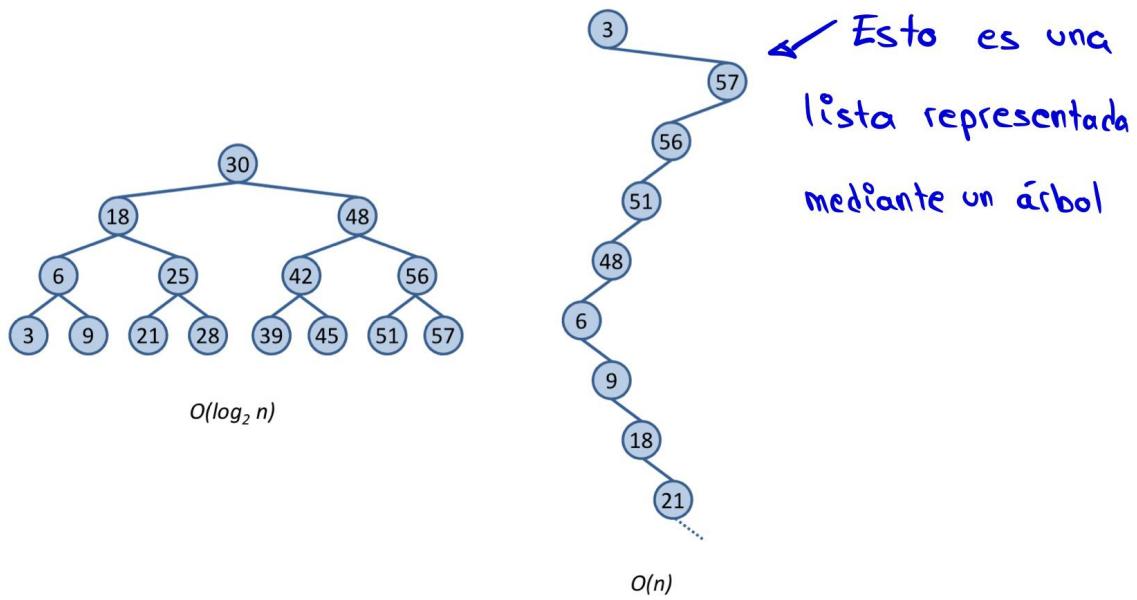
si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Parte 3 Árbol Binario de Búsqueda (ABB)



Esta propiedad de orden se cumple para todos los nodos del árbol.

El orden de la operación búsqueda depende de la estructura de ramificación del árbol:



Definición ABB: Árbol sin elementos repetidos con un orden lineal definido sobre el tipo de los elementos dado por el operador $<$ (sin dicho operador no funcionaría).

La propiedad de estos árboles es que para todo nodo n , los elementos del subárbol izquierdo son menores que el elemento

de n y los elementos del subárbol derecho son mayores que el elemento de n .

Operaciones del TAD Árbol Binario de Búsqueda:

Operaciones:

Abb()

Post: Construye un árbol binario de búsqueda vacío.

const Abb& buscar(const T& e) const

Post: Si el elemento e pertenece al árbol, devuelve el subárbol en cuya raíz se encuentra e ; en caso contrario, devuelve un árbol vacío.

Devuelve

subárbol

void insertar(const T& e)

Post: Si e no pertenece al árbol, lo inserta; en caso contrario, el árbol no se modifica.

void eliminar(const T& e)

Post: Elimina el elemento e del árbol. Si e no se encuentra, el árbol no se modifica.

bool vacio() const

Post: Devuelve `true` si el árbol está vacío y `false` en caso contrario.

Sirve para comprobar el resultado de `buscar(e)`

const T& elemento() const

Pre: Árbol no vacío.

Post: Devuelve el elemento de la raíz de un árbol binario de búsqueda.

→ No permite modificar el elemento libremente para que se siga cumpliendo la propiedad de orden.

const Abb& izqdo() const

Pre: Árbol no vacío.

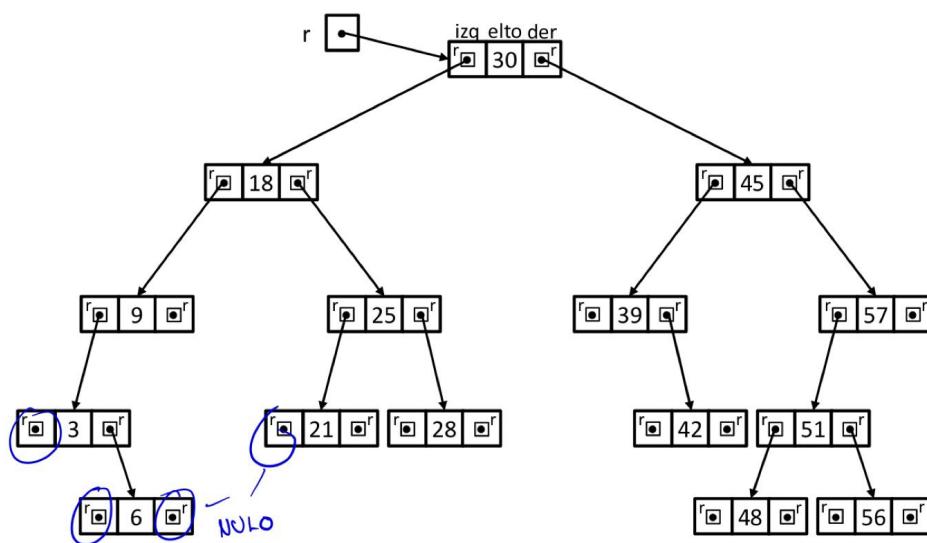
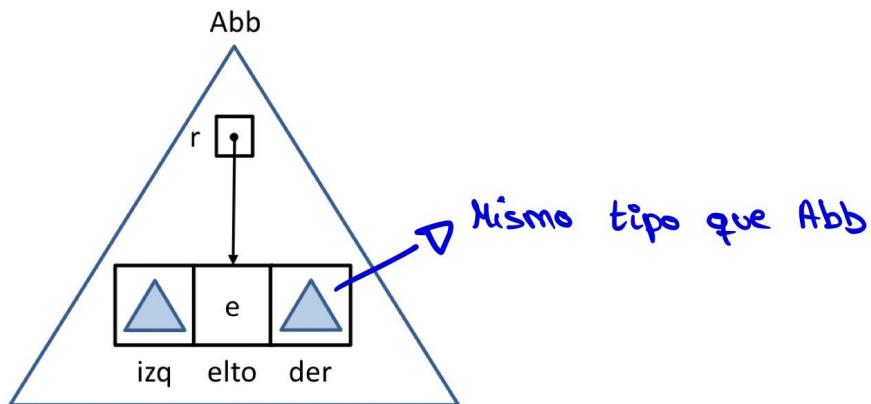
Post: Devuelve el subárbol izquierdo.

const Abb& drcho() const

Pre: Árbol no vacío.

Post: Devuelve el subárbol derecho.

Implementación mediante estructura dinámica recursiva:



Inserción: Hay que recorrer el ABB hasta encontrar el nodo que será padre del nuevo nodo que vamos a insertar

Suprimir

- Una hoja: La eliminamos y listo

**QUIERES
CONSEGUIR
15€??**

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



- **Un nodo n con solo un hijo:** El padre de n ahora tiene por hijo en vez de n , al hijo de n y eliminamos n

- **Un nodo n con 2 hijos:** Buscamos el menor de los mayores (el nodo de menor elemento en el subárbol derecho)

En el nodo n , metemos el elemento menor de los mayores y eliminamos el nodo que contenía al elemento menor de los mayores.

(El nodo que tiene al menor de los mayores o es hoja o tiene solo hijo derecho. No puedo tener hijo izquierdo porque si no significaría que hay un elemento menor que el que ese nodo contiene)

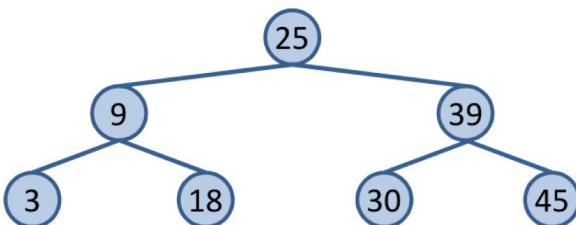
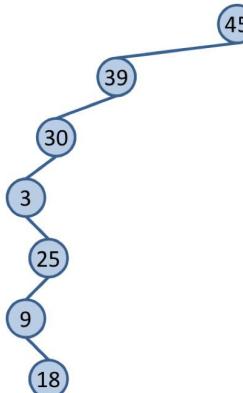


ÁRBOL Equilibrado (AVL)

El orden de inserción de los elementos en el árbol determina el grado de equilibrio del ABB.

Ej: 45, 39, 30, 3, 25, 9, 18

25, 9, 18, 39, 30, 3, 45



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Sucesivas inserciones y eliminaciones pueden alterar el grado de equilibrio del ABB.

Ya vimos que el tiempo de las operaciones insertar, eliminar y buscar dependen del grado de equilibrio del ABB porque en todas hay que recorrer el árbol.

El caso más favorable es $O(\log n)$ cuando el árbol está equilibrado y el más desfavorable es $O(n)$ cuando el árbol está degenerado en una lista.

Para garantizar que el tiempo es proporcional a la mínima altura posible del árbol, es decir, $O(\log n)$ es necesario que el árbol se quede lo más equilibrado posible tras cada operación modificadora.

Definiciones AVL:

Factor de equilibrio de un nodo: Altura del subárbol derecho - Altura del subárbol izquierdo.

ABB equilibrado: Árbol Binario en el que el factor de equilibrio de cada nodo es $-1, 0 \text{ o } 1$.

Árbol AVL (Adelson Velskii & Landis): ABB equilibrado. Los algoritmos de inserción y eliminación lo mantienen equilibrado en $O(\log n)$, reorganizando los nodos mediante una rotación en caso de que se fuera a perder la condición de equilibrio al insertar/eliminar.

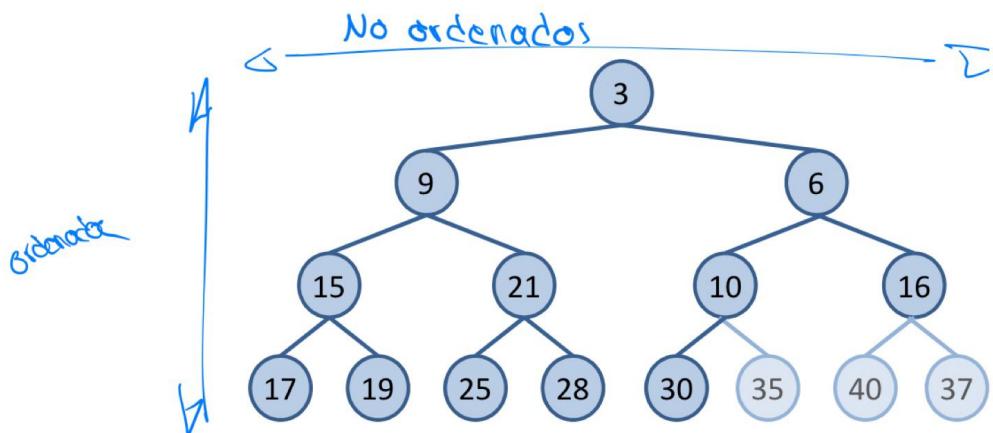
Parte 4 Árbol Parcialmente Ordenado (APO)

Definiciones:

Árbol completo: Que tiene todos sus niveles llenos, a excepción del nivel de más abajo al que le pueden faltar nodos por la derecha.

Completo \Rightarrow Equilibrado pero Equilibrado \neq Completo

Un árbol binario completo de n nodos tiene una altura $h = \log_2 n$



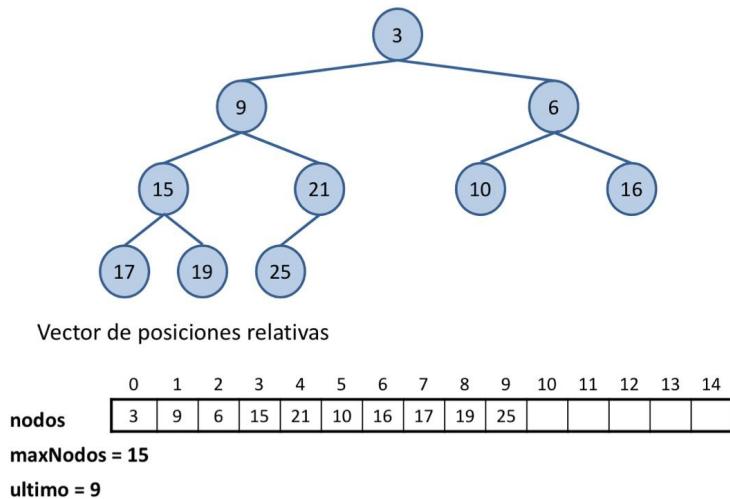
APO o Montículos: Árbol completo en el que el valor de cualquier nodo es menor que el de todos sus descendientes
 \Rightarrow El menor elemento está en la raíz.

//Nota: Se puede hacer un montículo de máximos.

La propiedad de completitud implica que la altura menor posible de un APO de n nodos es $h = \log_2 n$. Por lo tanto, inserciones y eliminaciones $O(\log_2 n)$

Se utiliza para representar colas con prioridad: Menor valor \Rightarrow Mayor prioridad.

Implementación vectorial posiciones relativas: Por la propiedad de completitud (garantiza que si hay huecos son en el último nivel a la derecha), no tenemos el problema de desperdicio de espacio.



Inserción : Meter en la última posición e ir subiendo (flotando) hasta que cumpla la propiedad de orden.

Eliminar : Solo se puede eliminar la raíz

Apo de 1 nodo: Se elimina el nodo y lista. El Apo queda vacío

Apo de 2 nodos: El hijo de raíz pasa a ser raíz y se elimina la antigua raíz

Apo de más de 2 nodos: Metemos el elemento "último" en raíz y eliminamos el anterior último.

Para que la actual raíz respete la propiedad de orden, vamos hundiéndola (mientras no cumpla la propiedad de orden, intercambiaremos el nodo n (initialmente raíz) con el

QUIERES
CONSEGUIR
15€??

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



menor de sus hijos (para que el que hemos subido cumpla propiedad de orden) y ahora el nodo n será el que hemos bajado.

APO min - max

Árbol completo en el que el elemento de un nodo de nivel par (recuerda que los niveles empiezan en cero) es menor que los elementos de todos sus descendientes y el elemento de un nodo de nivel impar es mayor que los elementos de sus descendientes.

Así, el elemento de menor valor está en la raíz y el de mayor valor en uno de sus hijos

* Ojo: El segundo elemento de mayor valor es también hijo de raíz? No necesariamente, puede ser descendiente del nodo con el elemento mayor.

El orden se define de padres a hijos, nietos... pero no entre hermanos

Insertar: Como es un árbol completo, se inserta en el último y se va flotando reordenando con el padre y el abuelo.



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Suprimir Min o Max: poner el último en la posición del min o max y reordenar de esta forma:

- Si eliminamos min, buscar el nieto menor y reordenar el nieto, su padre y su abuelo
- Si eliminamos max, lo mismo pero buscando el nieto mayor.

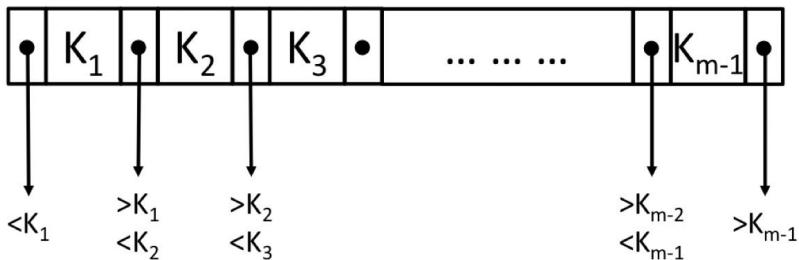
Si no tiene nietos, solo con los hijos y termina

Si ya es nodo hoja, termina.

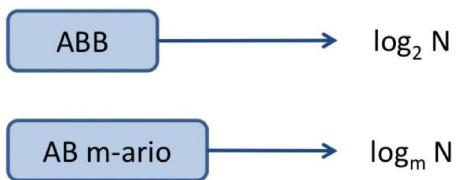
Parte 5 Árboles B

Si tenemos enormes (muy enormes) colecciones de elementos almacenados en ABBs, por cada paso del algoritmo de búsqueda tendríamos que acceder a memoria secundaria a por el nodo en concreto (y el tiempo acceso memoria secundaria es mucho mayor que en memoria principal)

Por ello surgen los Árboles B (varios elementos por nodo). El tamaño del nodo no superará al tamaño del bloque de memoria. Así por cada nodo, se lee un bloque de memoria y obtenemos varios elementos.



Localización de un registro en un fichero (accesos a bloques)



Dentro de un nodo, los elementos (llamados claves) se ordenan de menor (izquierda) a mayor (derecha).

Estos puntos a continuación son los que rigen el comportamiento de un árbol B en cuanto a inserciones, eliminaciones...

Árbol B

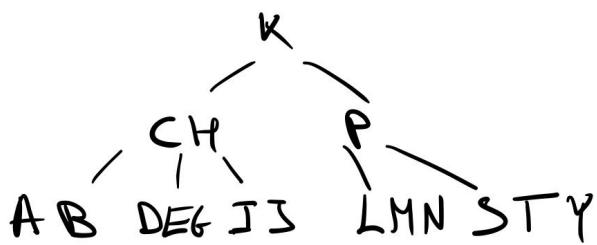
1. Un nodo que no es hoja con k claves tiene $k+1$ hijos.
2. Cada nodo tiene como máximo m hijos y m es el orden del árbol.
3. Cada nodo, excepto la raíz, contiene por lo menos $\lfloor (m-1)/2 \rfloor$ claves y, por supuesto, no más de $m-1$ claves.
4. La raíz tiene como mínimo una clave y dos hijos (al menos que sea hoja).
5. Todas las hojas aparecen en el mismo nivel.

Insertar la siguiente secuencia de valores en un árbol B de orden 4.

C, E, H, S, B, A, P, D, J, L, K, I, G, M, T, N, Y

$$m = k + 1$$

$$m = 4 \Rightarrow k = 3 \text{ (máximo de claves)}$$



Sí al insertar n° claves $> k \Rightarrow$ División y promoción

**QUIERES
CONSEGUIR
15€??**

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Eliminación en un árbol B

1. Si la clave que se va a eliminar no está en una hoja, intercambiarla con su sucesor más cercano según el orden lineal de las claves, el cuál se encontrará siempre en una hoja.
2. Eliminar la clave.
3. Si el nodo del que se ha suprimido la clave contiene por lo menos el número mínimo de claves, se termina.
4. Si este nodo contiene una menos de las claves mínimas, comprobar los hermanos izquierdo y derecho.
 - a. Si existe un hermano que tenga más del número mínimo de claves, redistribuir las claves con este hermano.
 - b. Si no existe un hermano que tenga más del número mínimo de claves, se concatenan los dos nodos junto con la clave del nodo padre que separa ambos.
5. Si se han concatenado dos nodos, volver a aplicar al nodo padre desde el paso 3, excepto cuando el padre sea el nodo raíz, en cuyo caso, el árbol queda con un nivel menos.

Resumen

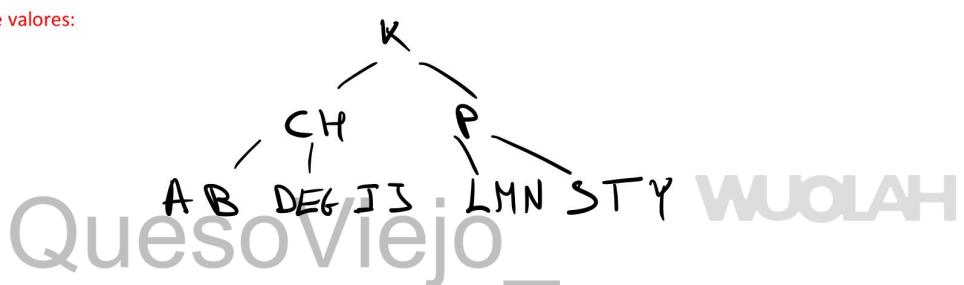
(menor de los mayores)

- 1) Intercambiar valor con el siguiente valor (una hoja)
- 2) Eliminar la clave
- 3) Si n° claves $< \lfloor (m-3)/2 \rfloor$
 - Pedir a hermano que redistribuya (de hermo a padre y de padre a nodo) si tiene más de $\lfloor (m-3)/2 \rfloor$
 - Si no, concatenar 2 hermanos y la clave del padre que los separaba (ojo, puede que dejemos al padre con menos de $\lfloor (m-3)/2 \rfloor$. Comprobar).

si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Eliminar la siguiente secuencia de valores:

A, H, B, J, G, I, E



CD — KP
 |
 LMN STY

$$m=4 \Rightarrow \min = \lfloor \frac{3}{2} \rfloor = 1$$

Parte 6 Grafos

Grafo: está formado por un conjunto de vértices y un conjunto de aristas. Cada arista es un par de vértices

$$G = (V, A)$$

↑ Grafo ↑ Vértices
↓ Aristas

$$A \subseteq (V \times V) \quad \begin{matrix} v \in V; w \in V \\ \therefore (v, w) \in A \end{matrix}$$

Grafo dirigido: $(v, w) \in A \neq (w, v) \in A$

La arista (v, w) se representa con una flecha de v a w . v es vértice incidente a w y w adyacente a v .

Grafo no dirigido: $(v, w) \in A = (w, v) \in A$

v y w son adyacentes. La arista (v, w) es incidente sobre los dos

Grafo ponderado: Sus aristas tienen asociados valores (pesos)
Representan tiempos, distancias, costes... según el contexto

Grado de un vértice:

- No dirigido: N° de aristas

- Dirigido:

- Grado de Entrada: Aristas incidentes (que llegan)

- Grado de Salida: Aristas adyacentes (que salen)

Camino: sucesión de vértices n_1, n_2, \dots, n_k tal que

$(n_i, n_{i+1}) \in A$, es una arista. Su longitud es el número

de aristas ($k-s$) si es no ponderado y será la suma de los pesos de las aristas si es ponderado

Camino simple: Todas las aristas son distintas

Camino elemental: Camino simple con todos los vértices distintos.

Ciclo: Camino en el que coinciden los vértices inicial y final.

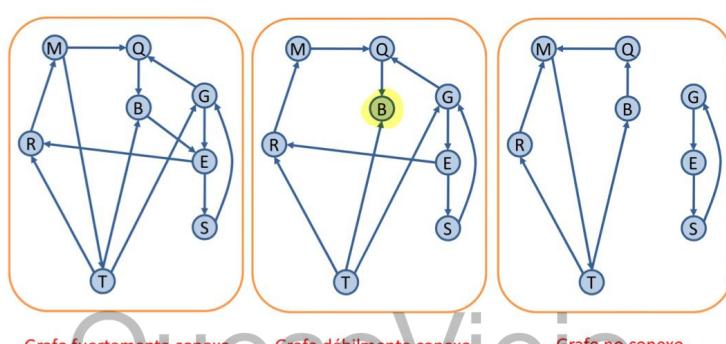
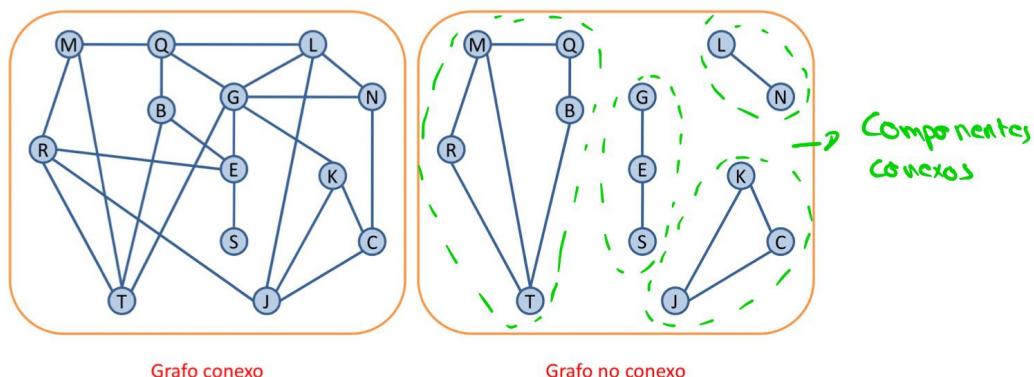
Es simple/elemental si el camino es simple/elemental

Ciclo elemental: Solo coinciden los vértices inicial y final.

Grafo conexo: Grafo no dirigido en el que hay al menos un camino entre cada par de vértices.

Si es dirigido y hay caminos entre cada par de vértices

\Rightarrow **Grafo fuertemente conexo.** Si no es fuertemente conexo, pero el grafo no dirigido subyacente (eliminando las direcciones) es conexo \Rightarrow **Grafo débilmente conexo.**



QuesoViejo_

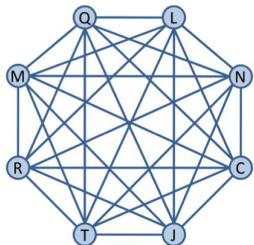
Si juegas con fuego, te fuegas

QUIERES
CONSEGUIR
15€??

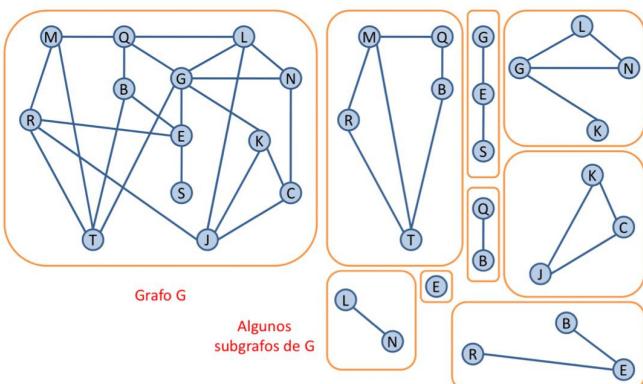
TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Grafo completo: Hay una arista entre cada par de vértices (en ambos sentidos si es dirigido).



Subgrafo: Formado por un subconjunto de vértices del grafo y todas las aristas entre ellos.



Componente conexo: Subgrafo conexo maximal (que no es subgrafo de ningún otro subgrafo conexo del grafo original)

Si hablamos de un grafo dirigido \Rightarrow **Componente fuertemente conexo:** Subgrafo maximal fuertemente conexo.

si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Implementaciones: Matriz donde los índices son los vértices y los elementos las aristas

Matriz de adyacencia: Los valores son 1,0 según si existe la arista o no.

Matriz de costes: Los valores son el peso de las aristas

Listas de adyacencia: Vector de listas que guarda las aristas que salen de cada vértice. Los elementos del vector son todas las aristas que salen del vértice [índice del vector].

- **No ponderado:** Solo se guarda vértice destino
- **Ponderado:** Guarda un struct {vértice_destino, coste}

Ventajas e inconvenientes:

- Las matrices de adyacencia y costes son muy eficientes para comprobar si existe una arista entre un vértice y otro. \Rightarrow Acceso con operador [] aplicado 2 veces
- Pueden desaprovechar gran cantidad de memoria si el grafo no es completo.
- La representación mediante listas de adyacencia aprovecha mejor el espacio de memoria, pues sólo se representan los arcos existentes en el grafo.
- Las listas de adyacencia son poco eficientes para determinar si existe una arista entre dos vértices del grafo. \Rightarrow Recorrer la lista
- Todas estas estructuras de datos no admiten la adición y eliminación de vértices, si no se utilizan matrices y vectores dinámicos.
- Una estructura alternativa es una lista de listas de adyacencia, en la que es posible añadir y suprimir vértices.

Parte 7 Algoritmos de Caminos de Coste Mínimo

Algoritmo de Dijkstra

```
template <typename tCoste>
vector<tCoste> Dijkstra(const GrafoP<tCoste>& G,
                         typename GrafoP<tCoste>::vertice origen,
                         vector<typename GrafoP<tCoste>::vertice>& P);
```

Calcula los caminos de coste mínimo entre **origen** y todos los vértices del grafo **G**.

Salida:

- Un vector de costes mínimos de tamaño **G.numVert()**.
- **P**, un vector de vértices de tamaño **G.numVert()**, tal que **P[i]** es el vértice anterior a **i** en el camino de coste mínimo desde **origen** hasta **i**.

Código:

```
// Suma de costes
template <typename tCoste>
tCoste suma(tCoste x, tCoste y)
{
    const tCoste INFINITO = GrafoP<tCoste>::INFINITO;

    if (x == INFINITO || y == INFINITO)
        return INFINITO;
    else
        return x + y;
} // Ojo, la suma de costes de Infinito + Infinito
   // debe ser Infinito y no desbordar.

template <typename tCoste>
vector<tCoste> Dijkstra(const GrafoP<tCoste>& G,
                         typename GrafoP<tCoste>::vertice origen,
                         vector<typename GrafoP<tCoste>::vertice>& P)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    vertice v, w;
    const size_t n = G.numVert();
    vector<bool> S(n, false); // Conjunto de vértices vacío.
    vector<tCoste> D; // Costes mínimos desde origen.

    // Iniciar D y P con caminos directos desde el vértice origen.
    D = G[origen]; // Ojo, en la matriz de costes ponía infinito
    D[origen] = 0; // Coste origen-origen es 0.
    P = vector<vertice>(n, origen); // Que el anterior a cada uno sea origen (de momento)
```

```

    // Calcular caminos de coste mínimo hasta cada vértice.
    S[origen] = true; // Ya sabemos que origen - orig es coste mínimo 0.
    for (size_t i = 1; i <= n-2; i++)
    {
        Para Mvert-S(origen) // Localizar vértice w no incluido en S con menor coste desde origen
        tCoste costeMin = GrafoP<tCoste>::INFINITO;
        for (v = 0; v <= n-1; v++)
            if (!S[v] && D[v] <= costeMin)
            {
                costeMin = D[v];
                w = v;
            }
        S[w] = true; // Incluir vértice w en S.
        // Recalcular coste hasta cada v no incluido en S, a través de w.
        for (v = 0; v <= n-1; v++)
            if (!S[v])
            {
                tCoste Owv = suma(D[w], G[w][v]);
                if (Owv < D[v])
                {
                    D[v] = Owv;
                    P[v] = w;
                }
            }
        return D;
    }
}

```

Porque al aplicar el algoritmo a los otros, este último vértice ya queda con el coste mínimo

Para Mvert-S(origen)

->

Función Camino:

```

#include "listaenla.h"
template <typename T> class GrafoP {
public:
    typedef Lista<vertice> tCamino;
    // ...
};

template <typename tCoste> typename GrafoP<tCoste>::tCamino
camino(typename GrafoP<tCoste>::vertice orig,
         typename GrafoP<tCoste>::vertice v, (destino)
         const vector<typename GrafoP<tCoste>::vertice>& P) Vector de
// Devuelve el camino de orig a v a partir de un vector
// P obtenido mediante la función Dijkstra().
{
    typename GrafoP<tCoste>::tCamino C;
    C.insertar(v, C.primera());
    do {
        C.insertar(P[v], C.primera());
        v = P[v];
    } while (v != orig);
    return C;
}

```

QuesoViejo_

wuolah

Si juegas con fuego, te fuegas

QUIERES
CONSEGUIR
15€??

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Algoritmo de Floyd

```
template <typename tCoste>
matriz<tCoste> Floyd(const GrafoP<tCoste>& G,
                      matriz<typename GrafoP<tCoste>::vertice>& P)
```

Calcula los caminos de coste mínimo entre cada par de vértices del grafo **G**.

Salida:

- Una matriz de costes mínimos de tamaño $n \times n$, con $n = G.\text{numVert}()$
- **P**, una matriz de vértices de tamaño $n \times n$, tal que **P[i][j]** es un vértice intermedio por el que pasa el camino de coste mínimo desde **i** a **j**.

Código:

Clase matriz:

```
/*-----*/
/* matriz.h */
/*-----*/
#ifndef MATRIZ_H
#define MATRIZ_H

#include <vector>

using std::vector;

// matriz cuadrada
template <typename T> class matriz {
public:
    matriz() {}
    explicit matriz(size_t n, const T& x = T())
        : m(n, vector<T>(n, x)) {}
    size_t dimension() const { return m.size(); }
    const vector<T>& operator [] (size_t i) const { return m[i]; }
    vector<T>& operator [] (size_t i) { return m[i]; }
private:
    vector< vector<T> > m;
};

#endif // MATRIZ_H
```



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Algoritmo de Floyd:

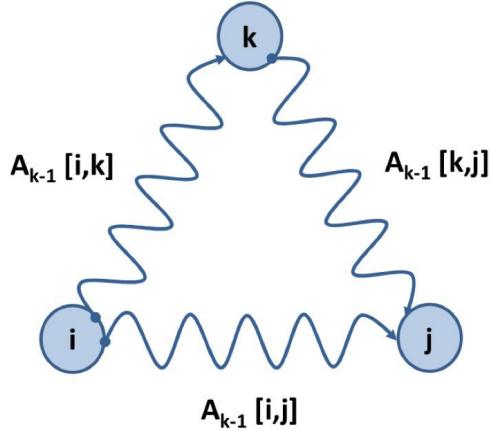
```
#include "matriz.h"

template <typename tCoste>
matriz<tCoste> Floyd(const GrafoP<tCoste>& G,
                      matriz<typename GrafoP<tCoste>::vertice>& P)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    const size_t n = G.numVert();
    matriz<tCoste> A(n); // matriz de costes mínimos
    // Iniciar A y P con caminos directos entre cada par de vértices.
    P = matriz<vertice>(n);
    for (vertice i = 0; i <= n-1; i++) {
        A[i] = G[i]; // copia costes del grafo
        A[i][i] = 0; // diagonal a 0
        P[i] = vector<vertice>(n, i); // caminos directos (Ir de 0 a j será 0)
    }
    // Calcular costes mínimos y caminos correspondientes
    // entre cualquier par de vértices i, j
    for (vertice k = 0; k <= n-1; k++) -> K, busca si pasar por K mejora los costes que
        for (vertice i = 0; i <= n-1; i++) ya teníamos (al principio directo)
            for (vertice j = 0; j <= n-1; j++) {
                tCoste ikj = suma(A[i][k], A[k][j]);
                if (ikj < A[i][j]) {
                    A[i][j] = ikj;
                    P[i][j] = k; -> Para ir de i a j, hay que pasar por k antes
                }
            }
    return A;
}
```

*Lai del bucle
pone la fila i a i*

*Ojo entre i-k y
k-j pueden haber otros
nodos intermedios.*

*(de i a k y de k a j se quedan (de momento) como
estaban al principio de esta iteración)*



$$A_k[i,j] = \min \{A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]\}$$

```
#include "listaenla.h"

template <typename T> class GrafoP {
public:
    typedef Lista<vertice> tCamino;
    // ...
};

caminoAux: Función recursiva interna. No incluye u ni w porque si no en
template <typename tCoste> typename GrafoP<tCoste>::tCamino caminoAux(typename GrafoP<tCoste>::vertice v,
    typename GrafoP<tCoste>::vertice w,
    const matriz<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo entre v y w, excluidos estos,
// a partir de una matriz P obtenida mediante la función Floyd().
{
    typename GrafoP<tCoste>::tCamino C1, C2;
    typename GrafoP<tCoste>::vertice u;

    u = P[v][w]; -> v a w hay que pasar por u
    if (u != v) {
        C1 = caminoAux<tCoste>(v, u, P); Llam recursiva de va u
        C1.insertar(u, C1.fin()); Acción de esta llamada, pasar por u
        C2 = caminoAux<tCoste>(u, w, P); Llam recursiva de u a w
        C1 += C2; // Lista<vertice>::operator +=(), concatena C1 y C2
    }
    return C1;
}
return C1;      Lista: Dev a rec + u + Dev u w
}
```

Llamada que hace el usuario.

```
template <typename tCoste> typename GrafoP<tCoste>::tCamino
camino(typename GrafoP<tCoste>::vertice v,
        typename GrafoP<tCoste>::vertice w,
        const matriz<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo desde v hasta w a partir
// de una matriz P obtenida mediante la función Floyd().
{
    typename GrafoP<tCoste>::tCamino C;

    C = caminoAux<tCoste>(v, w, P);
    C.insertar(v, C.primera());
    C.insertar(w, C.fin());
    return C;
}
```

Algoritmo de Warshall

```
matriz<bool> Warshall(const Grafo& G)
```

Determina si hay un camino entre cada par de vértices del grafo no ponderado G.

Salida:

- Una matriz booleana cuadrada de tamaño **G.numVert()**, tal que una posición **[i][j]** es **true** si existe al menos un camino entre el vértice **i** y el vértice **j**, y **false** si no existe ningún camino entre estos vértices.

Este algoritmo lo usamos con la implementación con listas de adyacencia.

Con los grafos ponderados, esa información la ya obtengo con Floyd:

Si el coste $A[i][j] \neq G::\text{INFINITO}$ es que hay camino y si no lo hay pues $A[i][j] == G::\text{INFINITO}$ siendo A la matriz de Floyd.

El procedimiento es el de Floyd pero "pa tantos" 😊😊 ya que aquí solo comprueba que existe camino y guarda que existe camino.

```
#include "matriz.h"
matriz<bool> Warshall(const Grafo& G)
{
    typedef Grafo::vertice vertice;
    const size_t n = G.numVert();
    matriz<bool> A(n);

    // Inicializar A con la matriz de adyacencia de G
    for (vertice i = 0; i <= n-1; i++) {
        A[i] = G[i];
        A[i][i] = true;
    }
    // Comprobar camino entre cada par de vértices i, j
    // a través de cada vértice k
    for (vertice k = 0; k <= n-1; k++)
        for (vertice i = 0; i <= n-1; i++)
            for (vertice j = 0; j <= n-1; j++)
                if (!A[i][j])
                    A[i][j] = A[i][k] && A[k][j];
    return A;
}
```

QUIERES
CONSEGUIR
15€??

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Parte 8 Recorridos Grafos

Profundidad

Aprender la versión Profundidad porque es igualita a anchura

```
Listado<Grafo::vertice> Profundidad(const Grafo& G, Grafo::vertice v)
```

```
{
    typedef Grafo::vertice vertice;
    const size_t n = G.numVert();
    vector<visitadas> marcas(n, NO_VISITADO);
    Lista<vertice> Lv;
    vertice i = v;
    do {
        if (marcas[i] == NO_VISITADO)
            Lv += Profun(G, i, marcas);
        i = (i+1) % n;
    } while (i != v);
    return Lv;
}
```

-> Devuelve el recorrido en profundidad de ese componente conexo (si existe un camino desde i). Además actualiza el vector de marcas

```
static Lista<Grafo::vertice> Profun(const Grafo& G, vertice v,
                                         vector<visitadas>& marcas)
```

```
{
    typedef Grafo::vertice vertice;
    const size_t n = G.numVert();
    Lista<vertice> Lv;

    marcas[v] = VISITADO;
    Lv.insertar(v, Lv.fin());
    for (vertice w = 0; w < n; w++)
        if (G[v][w] && marcas[w] == NO_VISITADO)
            Lv += Profun(G, w, marcas);
    return Lv;
}
```

{ En profundidad
según el orden de
los nodos.



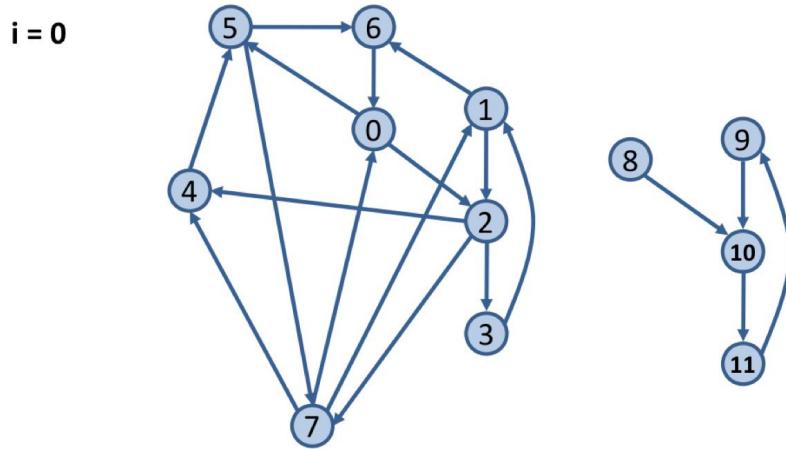
si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Otra versión de profundidad, con una Pila

```

#include "listaela.h"
#include "pilaela.h"
Lista<Grafo::vertice> Profundidad2(const Grafo& G, Grafo::vertice u)
{
    typedef Grafo::vertice vertice;
    const size_t n = G.numVert();
    vector<visitadas> marcas(n, NO_VISITADO);
    Pila<vertice> P;
    Lista<vertice> Lv;
    vertice i = u;
    do {
        if (marcas[i] == NO_VISITADO) {
            P.push(i);
            do {visita todos los nodos a los que haya un camino desde i
                vertice v = P.tope(); P.pop();
                if (marcas[v] == NO_VISITADO) {
                    marcas[v] = VISITADO;
                    Lv.insertar(v, Lv.fin());Lo mete en la lista que guarda el recorrido.
                    // Meter en la pila los adyacentes no visitados
                    for (vertice w = n; w > 0; w--)*Importante para que la primera arista que meta sea al menor nodo posible, hay que insertar en orden inverso al ser una pila.
                        if (G[v][w-1] && marcas[w-1] == NO_VISITADO)
                            P.push(w-1);
                }
            } while (!P.vacia());
        }
        i = (i+1) % n;
    } while (i != u);
    return Lv;
}

```



$$P = \{0\}$$

$$Lv = \{ \}$$

P: ...

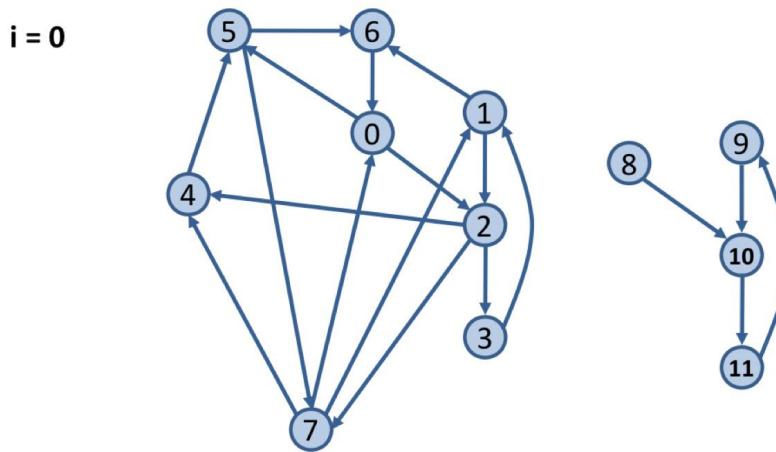
Lv: 0, 2, 3, 1, 6, 4, 5, 7 | Iter hasta $i=8$ | 8, 10, 11, 9 | hasta $i=n$ | $i=32$

Queso Viejo

wuolah

Anchura: En vez de insertar en una pila, en una cola (en P no meto los nuevos al principio sino al final)

```
#include "listaela.h"
#include "colaela.h"
Lista<Grafo::vertice> Anchura(const Grafo& G, Grafo::vertice u)
{
    typedef Grafo::vertice vertice;
    const size_t n = G.numVert();
    vector<visitadas> marcas(n, NO_VISITADO);
    Cola<vertice> C;
    Lista<vertice> Lv;
    vertice i = u;
    do {
        if (marcas[i] == NO_VISITADO) {
            C.push(i);
            do {
                vertice v = C.frente(); C.pop();
                if (marcas[v] == NO_VISITADO) {
                    marcas[v] = VISITADO;
                    Lv.insertar(v, Lv.fin());
                    // Meter en la cola los adyacentes no visitados
                    for (vertice w = 0; w < n; ++w) Ahora bucle al derecho.
                    if (G[v][w] && marcas[w] == NO_VISITADO)
                        C.push(w);
                }
            } while (!C.vacia());
        }
        i = (i+1) % n;
    } while (i != u);
    return Lv;
}
```



$$P = \{0\}$$

$$Lv = \{ \}$$

$$P = \emptyset, 2, 5, 3, 4, 7, 6, 7, 5, 3 \quad | \quad 8, 10, 11, 9$$

$$Lv = 0, 2, 5, 3, 4, 7, 6, 5 \quad | \quad 8, 10, 11, 9$$

QuesoViejo_

Si juegas con fuego, te puedes quemar.

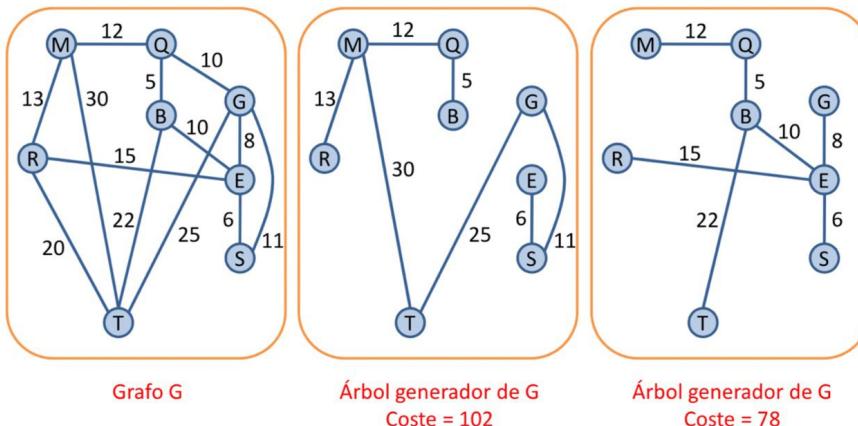
wuolah

Parte 9 Árboles Generadores de Coste Mínimo

Árbol Generador de un Grafo: Aquel que conecta todos los vértices del grafo. Su coste es la suma de los costes de las aristas que se hayan empleado.

No hay árbol generador de un grafo no conexo.

Un árbol es un grafo conexo acíclico



Pues vamos a buscar de todos los posibles árboles generadores de un grafo, el de coste mínimo.

El tipo arista debe tener definido el operador <

Hay 2 algoritmos para calcularlo :

- Kruskal : Utiliza el TAD partición y un APO
- Prim : Utiliza un APO

*Importante: Pueden haber varios árboles generadores de coste mínimo (varios con ese coste) y los 2 algoritmos devolver uno distinto. Lo importante es que sean mínimos. En Prim según por qué vértice empezaramos podría dar uno distinto.

**QUIERES
CONSEGUIR
15€??**

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



TAD Partición

Representa un conjunto de vértices de un grafo no dirigido y conexo.

La partición de un conjunto C es un conjunto de subconjuntos. La unión de todos los subconjuntos es el conjunto original C .

Cada subconjunto tendrá un representante canónico que es uno de sus elementos. Como los subconjuntos serán de nodos, no hay nodos repetidos por lo que es un buen representante canónico.

Veamos los contratos de las operaciones:

Operaciones: n subconjuntos de 1 elemento. Cada elemento es representante
Particion(int n); canónico de su subconjunto

Post: Construye una partición de subconjuntos unitarios del intervalo de enteros $[0, n-1]$.

void unir (int a, int b);

Pre: $0 \leq a, b \leq n-1$, a y b son los representantes de sus clases y $a \neq b$.

Post: Une el subconjunto del elemento a y el del elemento b en uno de los dos subconjuntos arbitrariamente. La partición queda con un miembro menos.

int encontrar(int x) const;

Pre: $0 \leq x \leq n-1$.

Post: Devuelve el representante del subconjunto al que pertenece el elemento x.



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Implementaciones del TAD Partición : Hay varios, unas más eficientes que otras.

Implementación del TAD Partición

1. Vector de pertenencia

$$P = \{\{2,3,7\} \{0,4\} \{5\} \{1,6,8,9\}\}$$

Representantes canónicos

$$P = \{\{2,3,7\} \{0,1,4,6,8,9\} \{5\}\}$$

Nuevo representante canónico

P	0	1	2	3	4	5	6	7	8	9
	4	1	7	7	4	5	1	7	1	1

P.unir(4,1)

P	0	1	2	3	4	5	6	7	8	9
	4	4	7	7	4	5	4	7	4	4

"El 9 está en el subconjunto del 1"

Particion() $\in O(n)$
encontrar() $\in O(1)$
unir() $\in O(n)$

Implementación del TAD Partición

2.1. Listas de elementos

$$P = \{\{2,3,7\} \{0,4\} \{5\} \{1,6,8,9\}\}$$

P	0	1	2	3	4	5	6	7	8	9
	4	1	7	7	4	5	1	7	1	1
	-1	6	3	-1	0	-1	8	2	9	-1

(El primero él mismo), después el que le indique la 2º fila.

P.unir(4,1)

$$P = \{\{2,3,7\} \{0,1,4,6,8,9\} \{5\}\}$$

P	0	1	2	3	4	5	6	7	8	9
	4	4	7	7	4	5	4	7	4	4
	-1	6	3	-1	1	-1	8	2	9	0

encontrar() $\in O(1)$

unir() $\in O(n)$, pero el tiempo de ejecución es menor. En vez de recorrer el vector entero, voy accediendo solo a las posiciones siguientes

QuesoViejo_

Si juegas con fuego, te fuegas

Implementación del TAD Partición

2.2. Listas de elementos (con longitud)

$$P = \{\{2,3,7\} \{0,4\} \{5\} \{1,6,8,9\}\}$$

P	0	1	2	3	4	5	6	7	8	9
	4	1	7	7	4	5	1	7	1	1
	-1	6	3	-1	0	-1	8	2	9	-1
	■	4	■	■	2	1	■	3	■	■

de la lista cuyo representante es 5,4,5,7 → O(n)

Se elige el de menor longitud (O(n)), pero hay menos elementos ⇒ constante multiplicativa menor

$$P = \{\{2,3,7\} \{0,1,4,6,8,9\} \{5\}\}$$

P	0	1	2	3	4	5	6	7	8	9
	1	1	7	7	1	5	1	7	1	1
	6	4	3	-1	0	-1	8	2	9	-1
	■	6	■	■	■	1	■	3	■	■

encontrar() ∈ O(1)

unir() ∈ O(n), pero el tiempo de ejecución se reduce por lo menos a la mitad.

Pior caso: 2 subconjuntos de igual longitud. Si solo hay 2, $O(\frac{n}{2}) = O(n)$

Implementación del TAD Partición

3.1. Bosque de árboles

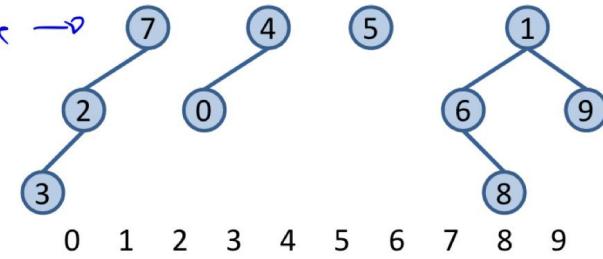
$$P = \{\{2,3,7\} \{0,4\} \{5\} \{1,6,8,9\}\}$$

P	0	1	2	3	4	5	6	7	8	9
	4	-1	7	2	-1	-1	1	-1	6	1

Padre de 9

P.unir(4,1)

Representante →

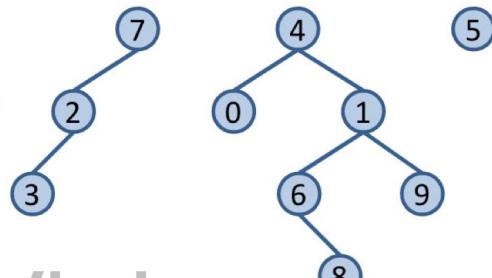


Poner los del subconjunto 4 como hijos de 1 o los del subconjunto 3 como hijos de 4

$$P = \{\{2,3,7\} \{0,1,4,6,8,9\} \{5\}\}$$

P	0	1	2	3	4	5	6	7	8	9
	4	4	7	2	-1	-1	1	-1	6	1

Hace padre de x hasta llegar a uno sin padre (el representante canónico)
encontrar() ∈ O(n) unir() ∈ O(1)



Pior caso de encontrar(x):

QuesoViejo_

Si juegas con fuego, te puedes quemar

Implementación del TAD Partición

3.2. Bosque de árboles (con control de altura)

a) Unión por tamaño

El árbol con menos nodos se convierte en subárbol del que tiene mayor número de nodos.

b) Unión por altura

El árbol menos alto se convierte en subárbol del otro.

} Usamos esta

$\text{unir}() \in O(1)$

$\text{encontrar}() \in O(\log n)$

encontrar(x) : Además de encontrar el representante canónico de x , reduce la altura del árbol para que no crezca indefinidamente

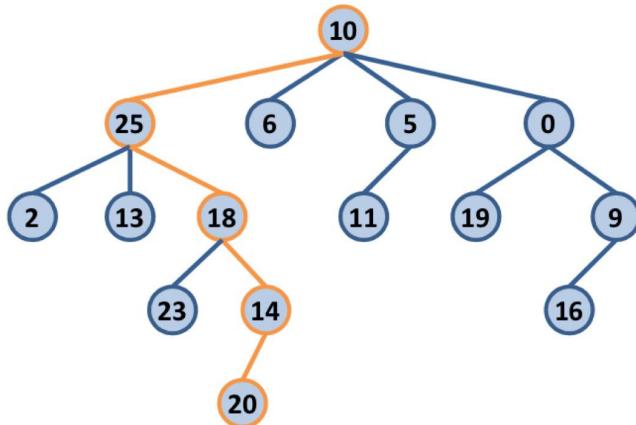
Como tiene que ir haciendo "padre de x " hasta llegar a raíz, moverá cada uno de estos elementos para que sean hijos de raíz: comprime la altura del árbol.

QUIERES
CONSEGUIR
15€??

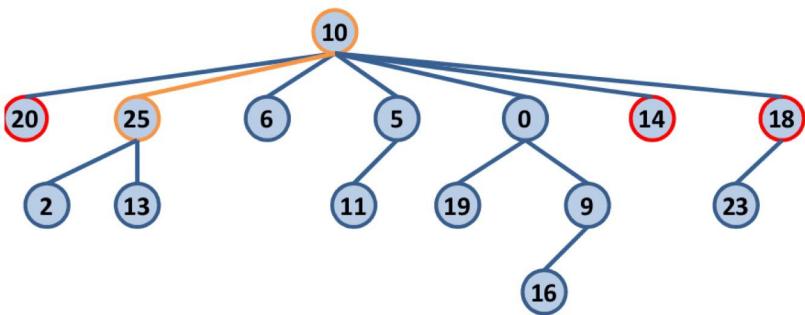
TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



Ejemplo: encontrar(20)



Además de devolver 10, pone 20, 14, 18 y 25 (ya estaba) como hijos de 10:



Cómo controla la altura para elegir la menor en unir()

En el vector "padre" se guardaba el padre de cada nodo y en raíz poníamos -1. En vez de -1, guardaremos -Altura-del-Árbol.

La operación encontrar(), al comprimir el árbol no actualiza su altura porque sería muy costoso, pero eso no importa porque el valor que tenga es cota superior de la altura real y va a hacer que los árboles no crezcan indefinidamente.



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

Tanto para Kruskal como para Prim:

```
template <typename T> class GrafoP {
public:
    typedef T tCoste;
    typedef size_t vertice;
    struct arista {
        vertice orig, dest;
        tCoste coste;
        explicit arista(vertice v=vertice(), vertice w=vertice(),
                        tCoste c=tCoste()): orig(v), dest(w), coste(c) {}
        // Orden de aristas para Prim y Kruskall
        bool operator <(const arista& a) const { return coste < a.coste; }
    };
    // resto de miembros de la clase GrafoP<T> ...
};
```

} Importante el operador < en el tipo arista.

Algoritmo de Kruskal

```
#include "particion.h"
#include "apo.h"

template <typename tCoste>
GrafoP<tCoste> Kruskall(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste mínimo
// de un grafo no dirigido ponderado y conexo G.
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    typedef typename GrafoP<tCoste>::arista arista;
    const tCoste INFINITO = GrafoP<tCoste>::INFINITO;

    const size_t n = G.numVert();
    GrafoP<tCoste> g(n);      // Árbol generador de coste mínimo.
    Particion P(n);          // Partición inicial de los vértices de G.
    Apo<arista> A(n*(n-1)/2); // Aristas de G ordenadas por coste.
```

Quitando autoaristas Porque es dirigido \Rightarrow matriz simétrica.

* Se podría haber hecho un apo de n^2 pero perdemos en eficiencia espacial.

QuesoViejo_

Si juegas con fuego, te fuegas

```

// Copiar aristas del grafo G en el APO A.
for (vertice u = 0; u <= n-2; u++)
    for (vertice v = u+1; v <= n-1; v++)
        if (G[u][v] != INFINITO)
            A.insertar(arista(u, v, G[u][v]));

size_t i = 1;
while (i <= n-1) {                                // Seleccionar n-1 aristas.
    arista a = A.cima();                          // arista de menor coste
    A.suprimir();
    vertice u = P.encontrar(a.orig); } Árbol del TAD Partición
    vertice v = P.encontrar(a.dest); }             a. coste;
    if (u != v) { // extremos de a pertenecen a distintos componentes
        P.unir(u, v);
        // Incluir la arista a en el árbol g.
        g[a.orig][a.dest] = g[a.dest][a.orig] = a.coste; → Guardo la
        i++;                                         arista en el GrafoP
    } else { } no se ha sumado a i, y hemos suprimido la
} arista del APO que simboliza el árbol.
return g;
}

```

Vemos que el TAD Partición solo nos sirve para comprobar que la arista elegida no une 2 vértices que ya formaran parte del árbol.

Deberemos hacer esa comprobación porque el algoritmo de Kruskal selecciona la arista de menor coste (no "visitada" todavía) entre **TODAS** las del grafo, por lo que puede sacar una arista que une 2 nodos que ya forman parte del árbol generador que estamos creando \Rightarrow la descartamos.

Algoritmo de Prim

En el algoritmo de Prim vamos a tener que hacer la misma comprobación pero la haremos con un vector de bool en vez de usar el TAD Partición

En este algoritmo, la arista seleccionada será una de entre todas en las que participa algún nodo del árbol generador que estamos creando (y que no esté ya incluida en el árbol generador).

El tipo arista debe tener el operador <, exactamente igual que para el algoritmo de Kruskal.

```
#include "apo.h"

template <typename tCoste>
GrafoP<tCoste> Prim(const GrafoP<tCoste>& G)
// Devuelve un árbol generador de coste mínimo
// de un grafo no dirigido ponderado y conexo G.
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    typedef typename GrafoP<tCoste>::arista arista;
    const tCoste INFINITO = GrafoP<tCoste>::INFINITO;

    arista a;
    const size_t n = G.numVert();
    GrafoP<tCoste> g(n); // Árbol generador de coste mínimo.
    vector<bool> U(n, false); // Conjunto de vértices incluidos en g.
    Apo<arista> A(n*(n-1)/2-n+2); // Aristas adyacentes al árbol g
    
    // ordenadas por costes.
```

Tampoco hay que ajustar tanto, con $n*(n-1)/2$ vale
($n * n$ queda geo, es muy poco eficiente)

-n+2 porque no se meten las aristas que formen ciclos.

QuesoViejo_

wuolah

**QUIERES
CONSEGUIR
15€??**

TRÁENOS A TU
CRUSH DE APUNTES
ANTES DE QUE
LOS QUEME



```
U[0] = true; // Incluir el primer vértice en U. Se podría meter cualquiera
// Introducir en el APO las aristas adyacentes al primer vértice.
for (vertice v = 1; v < n; v++)
    if (G[0][v] != INFINITO)
        A.insertar(arista(0, v, G[0][v])); Inserta en APO las aristas que salen
for (size_t i = 1; i <= n-1; i++) { // Seleccionar n-1 aristas. del origen (0)
    // Buscar una arista a de coste mínimo que no forme un ciclo.
    // Nota: Las aristas en A tienen sus orígenes en el árbol g.
    do {
        a = A.cima();
        A.suprimir();
    } while (U[a.dest]); // a forma un ciclo (a.orig y a.dest
                        // están en U y en g).
    // Incluir la arista a en el árbol g y el nuevo vértice u en U.
    g[a.orig][a.dest] = g[a.dest][a.orig] = a.coste;
    vertice u = a.dest;
    U[u] = true;
    // Introducir en el APO las aristas adyacentes al vértice u
    // que no formen ciclos. → para hacerlo más eficiente y que no haga muchas veces la
    // comprobación de arriba.
    for (vertice v = 0; v < n; v++)
        if (!U[v] && G[u][v] != INFINITO)
            A.insertar(arista(u, v, G[u][v]));
    }
return g;
}
```



si
consigues
que suba
apuntes, te
llevas 15€ +
5 Wuolah
Coins para
los sorteos

QuesoViejo_

Si juegas con fuego, te fuegas

wuolah