

En el momento de calcular $\theta(\text{Tri})$, se toman
en cuenta las constantes que aparece Tupper.

1

Términos matemáticos

Exponencial:

$$a) X^A \cdot X^B = X^{A+B}$$

$$b) X^A / X^B = X^{A-B}$$

$$c) (X^A)^B = X^{AB}$$

$$d) X^n + X^n = 2X^n$$

$$e) 2^n + 2^n = 2 \cdot 2^n = 2^{n+1}$$

Logarítmicas (generalmente $\log_A B = \log_2 A$):

Definición: Sea $X^A = B \rightarrow \log_X B = A$

Teoremas:

$$a) \log_A B = \frac{\log_C B}{\log_C A} ; A, B, C > 0 \text{ con } A \neq 1$$

$$b) \log AB = \log A + \log B ; A, B > 0$$

$$c) \log A/B = \log A - \log B$$

$$d) \log(A^B) = B \log A$$

$$e) \log X < X , \forall X > 0$$

Series: Geométricas:

Aritméticas:

$$a) \sum_{i=0}^N 2^i = 2^{N+1} - 1 \quad a) \sum_{i=1}^N i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$b) \sum_{i=0}^N A^i = \frac{A^{N+1} - 1}{A - 1} \quad b) \sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6} \approx N^3$$

$$c) \sum_{i=0}^N A^i \leq \frac{1}{1-A} \quad c) \sum_{i=1}^N i^K \approx \frac{N^{K+1}}{(K+1)} , K \neq -1$$

Si $0 < A < 1$

IMPORTANTE

$$\sum_{k=i}^j k = j-i+1$$

Recursión:

Función recursiva: una función definida en términos de ella misma.

Formada por:

① Caso base: valor para el cual la función es conocida sin recurrir a la llamada recursiva.

② Llamada recursiva: instancia en una recursión

Reglas: a) Siempre debe existir un caso base (que se resuelve sin recursión)
b) Siempre se debe garantizar que se llegue al caso base.
c) Asumir que todas las llamadas recursivas funcionen.
d) Nunca duplicar trabajo mediante la resolución de la misma instancia de un problema en llamadas recursivas separadas.

```
Ejemplo: public boolean paridad( int n ) {
    if (n=1)                                ①
        return false;                         ②
    else :
        if (n=0)                                ③
            return true;                         ④
        else
            return paridad( n div 2 );          ⑤
}
```

Casos base: líneas ② y ③

Llamada recursiva: línea ⑤

Para emplear se aplica la función recursiva:

$$T(n) = \begin{cases} C & n=1 \\ C + T(n/2) & n>1 \end{cases}$$

Funcióñ Recursiva

Todos los $i = 1$

Bucles: $\text{while } (i < N) \{$
 $i += h$
 $\}$

$\text{while } (i > N) \{$
 $i -= h$
 $\}$

$T(n)$

$\text{while } (i < N) \{$
 $i = i * 2$
 $\}$

$\text{while } (i \geq N) \{$
 $i = i / 2$
 $\} \rightarrow \log_2(N) + 1$

$\text{while } (i \leq N) \{$
 $i = i * h$
 $\}$

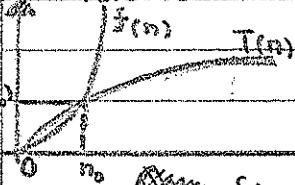
$\text{while } (i \geq N) \{ \rightarrow \log_h(N) + 1$
 $i = i / h$
 $\}$

En éstos bucles hay que prestar atención, específicamente, a la manipulación y transformación del índice ("i" en éstas casas) que aparece en la verificación. El desarrollo de éste índice nos indicará las repeticiones del bucle.

Definición Big-Oh: Sean las constantes positivas c y n_0 tales que
 $T(n) \leq c \cdot f(n)$ cuando $N \geq n_0 \rightarrow T(n) = O(f(n))$

Basicamente, lo que se trata de demostrar con el Big-Oh es que para cualquier función (en este caso nuestro tiempo de ejecución $T(n)$) siempre va a existir otra función que la acote. Esto significa que

(1) tratamos de encontrar una garantía del "máximo tiempo de ejecución del algoritmo".



$$T(n) = O(f(n)) \Rightarrow T(n) \text{ "es de orden" } f(n) \quad [\text{COTA SUPERIOR}]$$

• Sin embargo, hay momentos en los que $f(n) < T(n)$ para algunos valores de N . Así, estamos comparando sus radios de crecimiento relativo.

Por lo tanto, al definir un punto constante (n_0) se garantiza que la función $c \cdot f(n)$ siempre será más grande (o, como mínimo, igual) que nuestro tiempo de ejecución $T(n)$.

• Otra forma de considerar los radios de crecimientos relativos es viendo el valor de $\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)}$:

- el límite es 0 \rightarrow significa que $T(n) = o(f(n))$
- el límite es $c \neq 0 \rightarrow$ significa que $T(n) = \Theta(f(n))$
- el límite es $\infty \rightarrow f(n) = o(T(n))$

(little-oh $\rightarrow T(n) = o(f(n))$ significa "el radio de crecimiento de $T(n)$ es menor (\leq) que el de $f(n)$ ".)
(Theta $\rightarrow T(n) = \Theta(f(n))$ significa "el radio de crecimiento de $T(n)$ es igual ($=$) que el de $f(n)$ ".)

Radios típicos de crecimiento:

Reglas:

$f(n)$	nombre	1) Sean $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$ entonces: a) $T_1(N) + T_2(N) = O(f(N) + g(N))$. Menos formal serie $T_1(N) + T_2(N) = \max(O(f), O(g))$ b) $T_1(N) * T_2(N) = O(f(N) * g(N))$ c) \vdots
C	constante	
$\log(N)$	Logarítmico	
$\log^2(N)$	Log-cuadrática	
$N \cdot \log(N)$		
N^2	cuadrática	
N^3	cúbica	2) Sea $T(n)$ un polinomio de grado $k \rightarrow T(n) = \Theta(N^k)$
2^n	exponencial	3) Sea $\log \log^k N = O(N)$ p/ cualquier constante k

La habilidad de hacer un análisis genere una comprensión del diseño eficiente de algoritmos. 3

Análisis: al analizar un algoritmo, el recurso más importante es el tiempo de ejecución.

Para ésto consideremos:

• el tamaño de la entrada, N .

• $T_{\text{prom}}(N)$; $T_{\text{peor}}(N)$, con significado "el caso promedio de tiempo" y "el peor tiempo" respectivamente.

* N siempre debe ser mayor a 1.

* $T_{\text{prom}}(N) \leq T_{\text{peor}}(N)$ (\downarrow tiempo depende del tamaño de la entrada).

* El tiempo buscado en el análisis es el $T_{\text{peor}}(N)$, ya que se establece un límite para todas las tipos de entrada (inclusive una mala entrada).

* En algoritmos eficientes, el tiempo en orden de leer los datos de entrada es menor que el tiempo de resolución del algoritmo.

* Nunca se debe sub-estimar el $T(n)$ de un programa.

④ Cálculo $T(n)$: en un algoritmo se debe discernir entre


```

public static int maxSubSum1( int [ ] a ) {
    int maxSum = 0; // c1
    for ( int i=0 ; i < a.length ; i++ )           // indice crece N veces.
        for ( int j=i ; j < a.length ; j++ )
            { int thisSum=0; // c2
                for ( int k=i ; k < j ; k++ )
                    thisSum += a[ k ];
                if ( thisSum > maxSum )
                    maxSum = thisSum; // c3
            }
    return maxSum;
}

```

$$T(n) = \sum_{i=0}^{n-1} \left(\sum_{j=i}^{n-1} \left(c_1 + \sum_{k=i}^{j-1} c_2 + c_3 \right) \right) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i+1) = \\ = \sum_{i=0}^{n-1} (n-i)(n-i-i)$$

$$\sum_{j=i+1}^n j = \sum_{j=i}^{n-1} j = \frac{(n-i)(n-i-1)}{2} = \frac{1}{2} (n(n-i) - i(n-i)) \\ n^2 + n - in + i^2 - in - i \\ \frac{n^2 + i^2 + n - i - 2in}{2}$$

$$\frac{n(n+1) - i(i+1)}{2}$$

```

    public static void dos ( int N ) {
        int i, j, x, y;
        x = 0;
        y = 0;
        for ( i = 1; i <= n - 1; i++ ) {
            if ( ( n % 2 ) == 1 ) { // n es impar
                for ( j = 1; j <= n; j++ ) {
                    x = x + 1;
                    for ( j = 1; j <= i; j++ ) {
                        y = y + 1;
                    }
                }
            }
        }
    }

```

(1) (2)
 (3) (4)
 (5) (6) // Si el condic.ⁿ se cumple desde 0,
 (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20)

$\sum_{i=0}^{n-1} \sum_{j=0}^{n-i}$

- El peor de los casos sería que la entrada N sea impar, por la linea (6), ya que siempre se ejecutaría las bucles paternas (7) y (9).

$$\begin{aligned}
 T(n) &= \sum_{i=1}^{n-1} \left(\sum_{j=1}^i c + \sum_{j=1}^i c \right) = c \sum_{i=1}^{n-1} \sum_{j=1}^i 1 + c \sum_{i=1}^{n-1} \sum_{j=1}^i 1 = \\
 &= c \sum_{i=1}^{n-1} n + c \sum_{i=1}^{n-1} i = c[(n-1)n] + c \frac{(n-1)n}{2} = \\
 &\quad \text{no depende} \quad \text{depende} \\
 &= c[n^2 - n] + \frac{c}{2}[n^2 - n] = (n^2 - n)\left(c + \frac{c}{2}\right) = cn^2 - cn + \frac{cn^2 - cn}{2} = \\
 &= cn^2 \left(1 + \frac{1}{2}\right) = cn \left(1 + \frac{1}{2}\right) = \frac{3}{2}cn^2 - \frac{3}{2}cn
 \end{aligned}$$

Sean constantes $c = 2$ y $n_0 = 2$

$$C \cdot n^2 \geq \frac{3}{2}n^2 - \frac{3}{2}n \quad \forall n \geq n_0 \Rightarrow T(n) = O(n^2)$$

$$\begin{aligned}
 \text{Resolución de } \sum_{i=1}^{n-1} \sum_{j=1}^i 1 &= \sum_{i=1}^{n-1} (n-i+n) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \\
 &= (n-1)n - \frac{(n-1)n}{2} + (n-1) = n^2 - n - \frac{n^2 - n}{2} + n - 1 = \\
 &= \frac{1}{2}n^2 + \frac{3}{2}n - 1
 \end{aligned}$$

2

- public static void tres (int n) { (1)

 int i, j, k, sum; (2)

 sum = 0; (3)

 for (i = 1; i <= n; i ++) (4)

 for (j = i; j <= i * i; j ++) (5)

 for (k = j; k <= j; k ++) (6)

 sum = sum + j; (7)

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n \sum_{j=i}^{i^2} \sum_{k=j}^j c = c \sum_{i=1}^n \sum_{j=i}^{i^2} \sum_{k=j}^j 1 = c \sum_{i=1}^n \sum_{j=i}^{i^2} j = c \sum_{i=1}^n \frac{i^2(i^2+1)}{2} \\
 &= \frac{c}{2} \sum_{i=1}^n (i^4 + i^2) = \frac{c}{2} \left(\sum_{i=1}^n i^4 + \sum_{i=1}^n i^2 \right) = \frac{c}{2} \left[\frac{(n(n+1)(6n^2+9n^2+n-1))}{30} + \right. \\
 &\quad \left. + \frac{(n(n+1)(2n+1))}{6} \right] = \frac{c}{2} \left[\frac{1}{30} (n^2+n)(6n^3+9n^2+n-1) + \frac{1}{6} (6n^2+2n+1) \right] = \\
 &= \frac{c}{2} \left[\frac{1}{30} (6n^5+15n^4+15n^3-n) + \frac{1}{6} (2n^3+3n^2+n) \right] = \\
 &= \frac{c}{60} (6n^5+15n^4+15n^3-n) + \frac{c}{12} (2n^3+3n^2+n) = \text{Distribuimos las constantes...}
 \end{aligned}$$

$T(n) = O(n^5)$ ya que es un polinomio de grado 5. (REGLA DEL POLINOMIO)

```

    int c = 1;           (1)
    while (c < n) {      (2)
        algo de O(1);   (3)
        c = 2 * c;       (4)
    }

```

con K = cantidad de ejecuciones
while.

- Como el indice "c" va creciendo de a potencias de 2 $\rightarrow c = 2^{k-1}$

$$T(n) = \sum_{c=1}^{\log(n)} 1 = \log(n) \rightarrow T(n) = O(\log(n)), \text{ ya que } \exists c = 1 \text{ y } n_0 = 2 \text{ tq } c.n \geq \frac{n}{2} \forall n \geq n_0.$$

```

    int x = 1;           (1)  ln(n) → división entera
    for (int i = 1, i < n; i = i + 4) (2)
        for (int j = 1, j < n; j = j + ln(i/4)) (3)
            for (int k = 1, K < n, K = K * 2) (4)
                x = x + 1; (5)

```

- Hay que prestar atención a como van cambiando los indices

En el bucle (2) siempre se ejecuta $\frac{n}{4} + 1$, porque el indice crece de a $\frac{1}{4}$ partes de (n),
y como la primera siempre se ejecuta \rightarrow hay $\frac{n}{4} + 1$ repeticiones

$a = b = \frac{n}{4} + 1$ y $c = \log(n) + 1$

$$T(n) \geq \sum_{i=1}^{\infty} \sum_{j=1}^b \sum_{K=1}^c c = c \sum_{i=1}^{\infty} \sum_{j=1}^b \sum_{K=1}^c 1 = c \sum_{i=1}^{\infty} \sum_{j=1}^b [\log(n) + 1] =$$

$$\begin{aligned}
&= c \sum_{i=1}^{\infty} \left(\frac{n}{4} \log(n) + \log(n) + \frac{n}{4} + 1 \right) = c \left(\frac{1}{4} \sum_{i=1}^{\infty} n \log(n) + \sum_{i=1}^{\infty} \log(n) + \frac{1}{4} \sum_{i=1}^{\infty} n + \right. \\
&\quad \left. + \sum_{i=1}^{\infty} 1 \right) = c \left(\frac{1}{4} [n \log(n)] + (n+1) \log(n) + \frac{1}{4} [n+1]n + \frac{n}{4} + 1 \right) = \\
&= c \left(\frac{1}{4} \left[-\frac{1}{4} n^2 \log(n) + n \log(n) \right] + \frac{n}{4} \log(n) + \log(n) + \frac{1}{4} \left[\frac{1}{4} n^2 + n \right] + \frac{n}{4} + 1 \right) = \\
&= c \left(\frac{1}{16} n^2 \log(n) + \frac{1}{4} n \log(n) + \frac{1}{4} n \log(n) + \log(n) + \frac{1}{16} n^2 + \frac{1}{4} n + \frac{1}{4} n + 1 \right) \\
&= c \left(\frac{1}{16} n^2 \log(n) + \frac{3}{4} n \log(n) + \log(n) + \frac{1}{16} n^2 + \frac{1}{2} n + 1 \right)
\end{aligned}$$

- static public Integer recs (Integer n) {
 - if ($n \leq 1$) (1)
 - return 1; (2) → caso base
 - else (3)
 - return (recs(n-1) + recs(n-1)); (4) → Recursión
}

$$T(n) = \begin{cases} C & n \leq 1 \\ 2T(n-1) & n > 1 \end{cases}$$

Recursiones (K)

- K=1 $T(n) = 2T(n-1)$ // param: n
- K=2 $T(n) = 2(2T(n-1-1)) = 4T(n-2)$ // param: n-1
- K=3 $T(n) = 4(2T(n-2-1)) = 8T(n-3)$ // param: n-2
- K=4 $T(n) = 8(2T(n-3-1)) = 16T(n-4)$ // param: n-3
- ...
- K=i $T(n) = 2^i T(n-i)$

Si $n-i \neq 1 \rightarrow i=n$. Luego, $T(n) = 2^n \cdot 1$.

Sean $C=1$ y $n_0=1 \rightarrow C \cdot 2^n \geq 2^n \nmid m \geq n_0 \Rightarrow \Theta(T(n)) = O(2^n)$

- static public Integer rec2 (Integer n) {
 - if ($n \leq 1$) (1)
 - return 1; (2)
 - else (3)
 - return (2 * rec2(n-1)); (4)
}

}

- En la línea (5) hay una llamada recursiva, cuyo resultado se multiplica por 2, que es la constante.

$$T(n) = \begin{cases} C & , n \leq 1 \\ T(n-1) + d, & n > 1 \end{cases}$$

Recursiones (K)

- K=1, param: n, $T(n) = T(n-1) + d$
- K=2, param: n-1, $T(n) = (T(n-2) + d) + d = T(n-2) + 2d$
- K=3, param: n-2, $T(n) = (T(n-3) + d) + 2d = T(n-3) + 3d$
- ...

K=i, param: n-i-1, $T(n) = T(n-i) + i \cdot d$

• Si $n-i = 1 \rightarrow i=n \Rightarrow$ Luego, $T(n) = n \cdot d + C$

Sean $c_1 = c$, $c_2 = \alpha$ y $n_0 = 1$ entonces $(c+\alpha)n \geq n\alpha + c$. $\therefore T(n) = O(n)$

```

static public Integer rec3 ( Integer n ) {
    if ( n == 0 )                                (1)
        return 0;                               (2)
    else
        if ( n == 1 )                                (3)
            return 1;                               (4)
        else
            return ( rec3 ( n - 2 ) * rec3 ( n - 2 ) ); (5)

```

$$T(n) = \begin{cases} c_1 & , n=0 \\ c_2 & , n=1 \\ 2T(n-2) + c_2 & , n>1 \end{cases}$$

Recurpciones:

$$K=1, \text{ param: } n, \quad T(n) = 2T(n-2) + c_2$$

$$K=2, \text{ param: } n-2, \quad T(n) = 2(2T(n-2-2) + c_2) + c_2 = 2^2 T(n-4) + 2c_2$$

$$K=3, \text{ param: } n-4, \quad T(n) = 2^2 (2T(n-4-2) + c_2) + 2c_2 = 2^3 T(n-6) + 3c_2$$

$$K=i, \quad T(n) = 2^{i-1} T(n-2i) + i \cdot c_2$$

$$\text{Cuando } n-2i = 1 \Rightarrow i = \frac{n-1}{2} \quad T(n) = 2^{\frac{(n-1)/2}{2}} + \frac{n-1}{2} \cdot c_2 =$$

Sean $C = c_2$ y $n_0 \geq 1$ entonces $(2^{\frac{(n-1)/2}{2}} + \frac{n-1}{2} \cdot c_2) \leq C \cdot 2^n + n_0$. $\therefore T(n) = O(2^n)$

```

static public int potencia_x ( Integer x, Integer n ) { (1)
    Integer potencia;                         (2)
    if ( n == 0 )                                (3)
        potencia = 1;                           (4)
    else
        if ( n == 1 )                                (5)
            potencia = x;                         (6)
        else if ( (7)
            potencia = x;                         (8)
            for ( int i = 2; i <= n; i++ ) (9)
                potencia = potencia * x;          (10)
        }
    return potencia.
}

```

- Éste no es un método Recursivo.

- El peor caso se da que se ejecuta el "else" de la linea (8).

$$T(n) = d + \sum_{i=2}^{n-1} c = d + c \sum_{i=2}^{n-1} 1 = d + c(n-2+1) = d + c(n-1) = d + cn + c.$$

Sean $c_1 = c$ y $n_0 = 1 \rightarrow c_1 n \leq c_1 n + c \quad \forall n \geq n_0 \therefore T(n) = O(n)$

```

• static public Integer potencia_rec (Integer x, Integer n) {
    if (n == 0) (1)
        return 1 (2)
    else (3)
        if (n == 1) (4)
            return x; (5)
        else (6)
            if ((n % 2) == 0) (7)
                return potencia_rec (x * x, n/2); (8)
            else (9)
                return potencia_rec (x * x, n/2) * x; (10)
}

```

- Si la cantidad de entradas (n) es par, siempre se ejecuta la linea (8). Si no, la (10).
- Los casos (4) y (10) son similares, aunque la linea (10) tiene una constante más de ejecución.

$$T(n) = \begin{cases} c_1 & , n=0 \\ c_2 & , n=1 \\ T(n/2) + d & , n>1 \end{cases}$$

$$K=1, \text{ param: } n \rightarrow T(n) = T(n/2) + d$$

$$K=2, \text{ param: } n/2 \rightarrow T(n) = [T(n/4) + d] + d = T(n/2^2) + 2d$$

$$K=3, \text{ param: } n/4 \rightarrow T(n) = [T(n/16) + d] + 2d = T(n/2^3) + 3d$$

$$\dots K=i, \rightarrow T(n) = T(n/2^i) + i.d$$

$$\text{Cuando } n/2^i = 1 \rightarrow n = 2^i \rightarrow \log n = \log 2^i \rightarrow i = \log n$$

$$\therefore T(n) = c_1 + \log(n) \cdot d$$

Sean $c = d$ y $n_0 = 1$. entonces $\log(n) \cdot d \leq \log(n) \cdot c \therefore T(n) = O(\log(n))$

```

public class Percol {
    int[][] a = new int[n][n];
    public void recu(int[][] a, int n) {
        int[][] tmp = new int[n][n];
        if (n > 1) {
            recu(tmp, n/2);
            for (int i=0; i < tmp.length; i++) {
                for (int j=0; j < tmp.length; j++)
                    tmp[i][j] = a[i][j] * tmp[i][j];
            }
            if (n > 1)
                recu(2, n/2);
        }
    }
}

```

- El caso base es implícito en este caso, ya que se terminará la recursión cuando $n \leq 1$, en líneas (5) y (10). El análisis de estas líneas es una constante c_1 .

- Mientras la entrada n sea mayor a 1 se ejecutarán 2 procesos:

- 2 recursiones (líneas 6 y 11).

- 2 bucles anidados (líneas 7 y 9).

$$T(n) = \begin{cases} c_1 & , n \leq 1 \\ 2T(n/2) + n^2 c_2 , & n > 1 \end{cases}$$

$$K=1, \text{ param: } (n), T(n) = 2T(n/2) + n^2 c_2$$

$$K=2, \text{ param: } (n/2), T(n) = 2(2T(n/2^2) + (n/2)^2 c_2) + n^2 c_2 = 2^2 T(n/2^2) + \frac{1}{2^2} n^2 c_2 + n^2 c_2$$

$$K=3, \text{ param: } (n/2^3), T(n) = 2^2 (2T(n/2^3) + (n/2^2)^2 c_2) + \frac{1}{2^2} n^2 c_2 + \frac{1}{2^2} n^2 c_2 =$$

$$= 2^3 T(n/2^3) + \frac{1}{2^3} n^2 c_2 + \frac{1}{2^2} n^2 c_2 + \frac{1}{2^2} n^2 c_2 =$$

$$K=4, \text{ param: } (n/2^4), T(n) = 2^3 (2T(n/2^4) + (\frac{1}{2^3} n)^2 c_2) + \frac{1}{2^4} n^2 c_2 + \frac{1}{2^3} n^2 c_2 + \frac{1}{2^2} n^2 c_2 =$$

$$= 2^4 T(n/2^4) + \frac{1}{2^6} n^2 c_2 + \frac{1}{2^4} n^2 c_2 + \frac{1}{2^3} n^2 c_2 + \frac{1}{2^2} n^2 c_2 =$$

$$= 2^4 T(n/2^4) + n^2 c_2 \left(\frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^6} + \frac{1}{2^8} \right) = - \frac{4}{3}$$

$$K=i, T(n) = 2^i T(n/2^i) + n^2 c_2 \left(\sum_{j=0}^i 2^{-2j} \right) = 2^i T(n/2^i) + n^2 c_2 \left(\frac{1}{1-2^{-2}} \right)$$

$$\text{(*)} \sum_{j=0}^n a^j \leq \frac{1}{1-a}, \text{ si } 0 < a < 1.$$

(*)

Cuando $n/2^i = 1 \rightarrow n = 2^i \rightarrow \log(n) = \log(2^i) \rightarrow \log(n) = i$ Entonces

$$T(n) \leq 2^{\log(n)} \cdot c_1 + \frac{4}{3} n^2 c_2 = n \cdot c_1 + \frac{4}{3} n^2 c_2$$

Scénario $C = \frac{4}{3}c_2 + c_1$ et $n_0 = 1$, Ex vold

$$C.n^2 \leq c_1 n + \frac{4}{3}c_2 n^2, \forall n \geq n_0.$$

où $T(n) = O(n^2)$

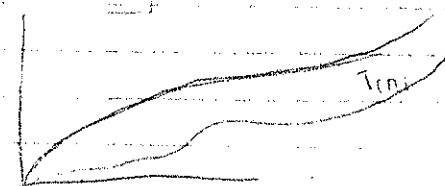
Une forme de juilletier

$$c_1 n \leq c_2 n^2 \quad \forall n \geq 1$$

$$c_1 n^2 \leq \frac{4}{3}c_2 n^2 \quad \forall n \geq 1$$

$$c_1 n^2 + \frac{4}{3}c_2 n^2 \leq \left(c_1 + \frac{4}{3}c_2\right)n^2 \quad \forall n \geq 1$$

(*)



1. Secuencias (p2)

PREFORDEN

preorden (NodoBinario) {
 Si (nodo != NULL) entonces
 cout << nodo->dato;
 preorden (NodoBinario Izq);
 preorden (NodoBinario Der);
 }

INORDEN

inorden (NodoBinario) {
 Si (nodo != NULL) entonces
 inorden (NodoBinario Izq);
 cout << nodo->dato << endl;
 inorden (NodoBinario Der);
 }

POSTORDEN

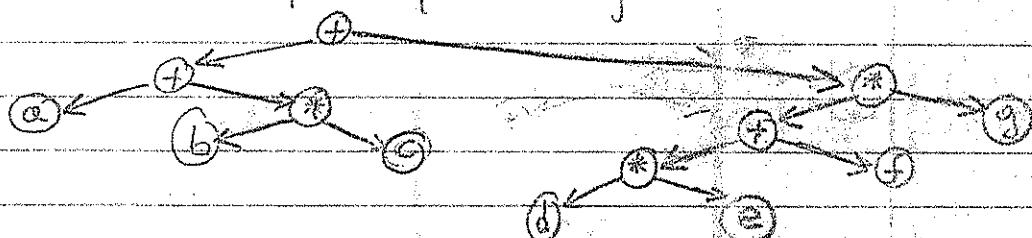
postorden (NodoBinario) {
 Si (nodo != NULL) entonces
 postorden (NodoBinario Izq);
 postorden (NodoBinario Der);
 cout << nodo->dato;
 }

- APLICACIONES

- Nodos de expresión

- Dado una expresión matemática simple, cada señal se puede valorar en un símbolo. Sus hijos son OPERANDOS y los nodos internos contienen los OPERADORES BINARIOS (+, -, *, %) & OPERADORES MATEMATICOS (max, min, ...)

- Podemos evaluar la expresión aplicando alguno de los RECORRIDOS a la lista



④ RECORRIDO INORDEN (raiz) = Expresión Infixa = $(a + (b * c)) + ((d * e) + f) * g$

④ RECORRIDO PREFORDEN (raiz) = Expresión Prefijo = + + * * b c * + d e f

④ RECORRIDO POSTORDEN (raiz) = Expresión Postfixa = a b c * + d e * + f + g * +

- Por la definición recursiva de un AB, es común escribir rutinas recursivas.

- RECORRIDO POR NIVEL: utilizado en árboles generales, UTILIZANDO UNA COLA ADICIONAL

- Por Niveles (T árbol)

Q = cole de nodos;

encolar (Q, T);

ImprimirRaiz (T);

mientras (la cole no esté vacía) hacer

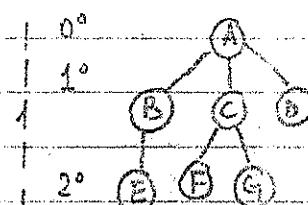
desencolar (Q, T);

L = lista de hijos Raiz (T)

por (i, dato h en L) hacer

encolar (Q, h)

Imprimir (h)



PRENIVEL (A) = A-B-C-D-E-F-G

- Este algoritmo es de $O(n)$ porque se tiene que recorrer todos los nodos sin repetir ninguno.

Árbol Binario de Búsqueda (ABB)

- La característica que lo distingue es que "para cada nodo, X , en el árbol, los valores de todos los ítems en su subárbol izquierdo son menores que el ítem en X , y los valores de todos los ítems del subárbol derecho son mayores que el ítem en X ". Esto es su **PROPIEDAD DE ORDEN**.
- Debido a la propiedad de orden, son fundamentales para realizar búsquedas, con $T(n) = O(\log n)$.
- La profundidad promedio de un ABB es de $O(\log n)$.
- Como todos los ítems de los nodos necesitan ser ordenados, es necesario que se puedan **COMPARAR** (es decir, que implementen la interfaz Comparable y sean comparados usando el método "comparar()").

IMPLEMENTACIÓN

```
class NodoBinario< T > extends Comparable< T > {
    T item;
    NodoBinario< T > Izquierda;
    NodoBinario< T > Derecha;
}
```

- Algunas operaciones:

öffentEstoContenido: busca en el árbol si un elemento X se encuentra.

```
public boolean öffEnthalten (elemento) {
    devolver öffEnthalten (elemento, raiz);
}

privado öffEnthalten (elemento, nodo) {
    si (nodo == null) entonces devolver FALSE; // no se encontró
    si (nodo.elemento < elem) entonces devolver öffEnthalten (elem, nodo.derecha);
    si (nodo.elemento > elem) entonces devolver öffEnthalten (elem, nodo.izquierdo),
    sino // si nodo.elemento == elem
        devolver TRUE; // se encontró
}
```

• # buscamin y buscamax: tienen el último nodo hacia la izquierda y derecha, respectivamente, en el ABB.

- Se puede resolver recursiva e iterativamente.

```
public int buscamin () {
    devolver buscamin (raiz);
}

privado buscamin (nodo) {
    si (nodo == null) entonces devolver null; // se fin si nro
```

Pila TAD:

TAD

- Una pila es una lista con la restricción de que las inserciones y borrados pueden ser hechas en el TOPE de la pila; es decir, el final de la lista.
- Operaciones fundamentales (interpretar en Java):
 - INPUT: $[+T] \text{ PUSH}(<T> \text{ un Item}) \rightarrow$ insertar en el final de la lista TAD;
 - OUTPUT: $[+T] \text{ POP}() \rightarrow$ convierte y devuelve el último elemento insertado.
 - $[+T] \text{ TOP}() \rightarrow$ devuelve, sin eliminar, el último elemento insertado.
- Es una estructura LIFO (último en entrar, primero en salir)
- En una lista ENLAZADA, el TOPE se encuentra en el principio de la lista.
- Estas operaciones básicas deben garantizarse de que sean $O(1)$.

Aplicaciones:

Símbolos de apertura Símbolos de clausura

- Evaluación de EXPRESIONES

- Se busca que el llave, coincide el paréntesis derecho trajo su correspondiente izquierdo

- Procedimiento:

- a) Crea una pila vacía
- b) Leer caracteres hasta el final de la linea.
 - 1) Si el carácter es un símbolo de apertura, se lo apila.
 - 2) Si es un símbolo de clausura, se extrae un elemento y se verifica:
 - si la pila está vacía \rightarrow REPORTAR ERROR.
 - si el símbolo extraído no corresponde al de clausura \rightarrow REPORTAR ERROR.
- c) Al final de la linea, si la pila no está vacía, se reporta error.

- Evaluación de expresiones POSTFIJAS, PREFIJAS o INFIXAS

- Para Nombrar o métodos recursivos:

- Cuando se llaman a métodos o procedimientos dentro de un programa, es necesario guardar los registros usados por este último para no perder sus valores.
- Cuando se realiza una LLAMADA A PROCEDIMIENTO \rightarrow se apila un conjunto de registros correspondiente al método invocado.
- Cuando se realiza un RETURN \rightarrow se desapila el último conjunto de registros y se los coloca en los registros de CPU correspondientes.
- La pila en memoria, generalmente tiene su base en la dirección de memoria más alta, creciendo hacia direcciones inferiores.

TAD (Tipo abstracto de datos): es un conjunto de objetos finitos con un conjunto de operaciones.

- Son estructuras matemáticas (como CONJUNTOS y GRAFOS)
- El conjunto de operaciones que son implementadas en el TAD (si lo se encuentra la interfaz del objeto)

General

* Lista TAD: Tiene la forma $A_0, A_1, A_2, \dots, A_N$; tiene tamaño N .

• Una lista vacía tiene tamaño 0.

• A_i es el sucesor de A_{i-1} (con $i < N$) y A_{i+1} antecede a A_i (con $i > 0$).

• Cada A_i es un objeto relacionado en el TAD con otros objetos de la misma clase.

- Implementaciones listas (ARRREGLO)

• Con un ARREGLO de tamaño redimensionable.

• $T(n)$ de operaciones:

+ Imprimir Elementos $\rightarrow T(n) = O(N)$

+ Ver Elemento En la Posición $\rightarrow T(n) = O(1)$

+ Insertar Elemento / borrar Elemento

• peor Caso = inserción en posición 0 $\rightarrow T(n) = O(N)$

• caso Promedio = inserción en mitad $\rightarrow T(n) = O(N)$

• mejor Caso = inserción / borrado al final $\rightarrow T(n) = O(1)$

• Los elementos se encuentran contiguos en memoria.

* Lista Enlazada

• los elementos son una serie de NODOS, no necesariamente adyacentes en memoria. Cada uno contiene una referencia al próximo Nodo.



• Se mantiene un puntero al NODO primero y el ÚLTIMO:

• $T(n)$ de operaciones:

+ Imprimir Elementos $\rightarrow T(n) = O(N)$

+ Ver Elemento En Posición $\rightarrow T(n) = O(N)$

+ Insertión / Borrado:

• mejor Caso = insertar Al principio/Final / Borrar Al principio $\rightarrow T(n) = O(1)$

• peor Caso = inserción o borrado en cualquier otro Posición $\rightarrow T(n) = O(N)$

• Listable = si no

• Compara (entre el nodo al PRÓXIMO y el PREDECESOR)

• Crear / Borrar en cualquier Posición $\rightarrow T(n) =$

Arboles Binarios de Búsqueda

Son una estructura de datos donde la mayoría de las operaciones es realizada en $O(\log N)$.

DEFINICIÓN: Es un árbol binario de nodos.

- Existe un NODO RAÍZ, llamado n_0 .

- Es compuesto de 0 o más hijos: s_1, s_2, s_3, \dots

- Cada hijo del nodo anterior con un VÉRTICE al nodo n_1 .

- Si s_1 es hijo de n_0 , y n_1 es padre de s_1 .

- Aquellos hijos que no son padres (es decir, que no tienen hijos), se llaman HOJAS.

- Los hijos de un mismo padre son hermanos.

DEFINICIÓN RECURSIVA:

- Existe (el) NODO RAÍZ con el que comienza los vértices.

Cada nodo, excepto la raíz, está asociado con un PADRE y un Hijo, excepto las hojas.

Características:

- Camino: es desde $n_1 \rightarrow n_k$, en la secuencia de nodos $n_1, n_2, n_3, \dots, n_k$, conectados mediante K-1 aristas. n_i es padre de n_{i+1} , con $1 \leq i \leq k-1$.

- LARGITUD del camino $(n_1 - n_k)$, es la cantidad de nodos en el camino, es decir $n_k = n_1$ tiene longitud K-1.

• La longitud del camino $n_1 - n_k$ (desde la raíz hasta el nodo) es 0.

• Existe solo un camino desde la raíz a cualquier otro nodo.

- Profundidad (o altura) de un Nodo (n_i) es la longitud del camino $n_0 - n_i$ (camino raíz).

• La profundidad del árbol es igual a la profundidad de la hoja más profunda.

• La profundidad de la raíz es 0.

- Altura (o profundidad) de un Nodo (n_i), es la longitud del camino más largo $n_i - n_j$, donde (n_j) representa una hoja del árbol.

• La altura del árbol es igual a la altura de la raíz.

• Todas las hojas tienen altura 0.

Ver ANCESTRO PROPIO y DESCENDIENTE PROPIO,

• •

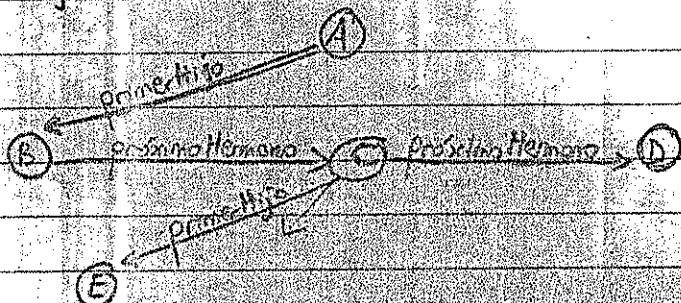
- Grafo del Árbol: es la cantidad máxima de hijos que puede tener un nodo del árbol.

- Nivel de un nodo: longitud del más largo camino desde la raíz a un nodo.

- Un árbol general es aquel en el cual no se sabe con exactitud cuantas hijos podrá tener un nodo, es decir, que es un árbol de grado N.

IMPLEMENTACIÓN (en Java)

```
class NodoGeneral < T >
{
    < T > elementoData;
    NodoGeneral < T > primerHijo;
    NodoGeneral < T > proximoHermano;
}
```



- APLICACIONES

- Esta estructura se para almacenar la jerarquía de directorios de varios sistemas operativos (UNIX, DOS)
 - En UNIX, cada directorio es un nodo con una lista de todos sus subdirectorios y ficheros (bins). Los ficheros son archivos.

Árboles binarios (AB)

- Son árboles de grado 2
- Consiste en una raíz y 2 subnodos (S_1 y S_2) (donde cualquier nodo puede ser raíz).
- En promedio, la profundidad de un árbol binario es de $O(\sqrt{N})$
- En el peor de los casos, el árbol puede convertirse en una lista, es una profundidad de $N-1$.
- Un árbol binario tiene reportes de sus N nodos, N+1 links en NULL.

IMPLEMENTACIÓN

```
class NodoBinario < T >
{
    < T > elementoData;
    NodoBinario < T > hijoIzquierdo;
    NodoBinario < T > hijoDerecho;
}
```

- Recorridos en la ejecución de los elementos de un árbol binario.

↳ EN PROFUNDIDAD



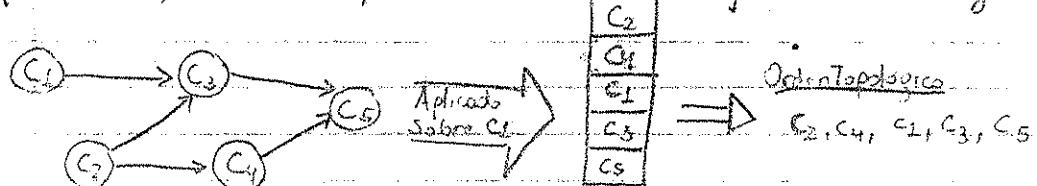
- PREORDEN: el trabajo sobre un nodo es realizado antes que sus hijos.
↳ ejemplo: PREORDEN (a) = a - b - c

- INORDEN: el trabajo se realiza en el orden: hijo izquierdo - raíz - derecho.
↳ ejemplo: INORDEN (a) = b - a - c

3) DORT con RECORRIDO EN PROFUNDIDAD (DFS) →

- Necesitamos una estructura de Pila \mathbb{P} y un array Booleano de Visitados \mathbb{A} .
- Con el DFS busco aquellos vértices que no poseen adyacentes $\mathbb{A}^{\text{procesado}}$ las voy contando.
- El orden topológico puede determinarse por el orden en que voy apilando.
- El último elemento en la pila es el que más grados de entrada posee.
- Apilando cuando, dentro del DFS, se acabe el bucle que se ejecuta sobre los adyacentes.

Ejemplo:



4) Algoritmos de caminos de costo mínimo

- Dado un grafo pesado, donde $c: (v_i, v_j) \in |E|$ tiene un peso definido como c_{ij} :

$$\rightarrow \text{el costo del camino } v_1, v_2, \dots, v_N \text{ es } \sum_{i=1}^{N-1} (c_{i,i+1})$$

- Para grafos sin peso, el costo de un camino es la cantidad de aristas.

Problema: Dado un V inicial \mathbb{S} , se busca el camino menos costoso desde \mathbb{S} al resto de los nodos.

→ Algoritmo de costo mínimo No PESADO $\boxed{O(V^2)}$

- Desde el vértice inicial \mathbb{S} , haremos un BFS, de forma tal que el costo de un camino hasta el nivel N , sea de $\boxed{O(V^2)}$ aristas.
- Cada vértice mantiene info en su estructura:
 - d_v : es el costo del camino mínimo (el comienzo desconocido, ∞).
 - p_v : indica el nodo previo del camino mínimo $S \rightarrow v$ (el comienza marcado como 0).
 - conocido: indice si ese nodo fue procesado previamente (comienza en FALSE).
- Cuando un vértice está "conocido", se garantiza de que \mathbb{P} camino menor costoso.
- Al procesar un vértice v , cada adyacente w se actualiza:
 - Si $(d_w > d_v + c_{v,w}) \rightarrow d_w = d_v + c_{v,w}$
 - Si se cumple lo anterior $\rightarrow p_w = v$.

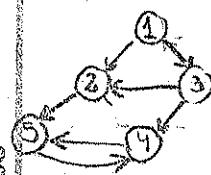
- El camino se construye haciendo el camino inverso a través de p_v .

Ejemplo:

Dado el grafo G , sea \mathbb{S} el vértice 1:

v	Con	d_v	p_v	v	Con	d_v	p_v	v	Con	d_v	p_v	v	Con	d_v	p_v
1	F	0	0	1	T	0	0	2	T	0	0	2	T	0	0
2	F	∞	0	2	F	1	1	2	T	1	1	2	T	1	1
3	F	∞	0	3	F	1	1	3	F	1	1	3	T	1	1
4	F	∞	0	4	F	∞	0	4	F	∞	0	4	F	2	3
5	F	∞	0	5	F	0	0	5	F	2	2	5	F	2	2

núsculos



agregar A lista de salida (v)

para c/w adyacentes a v

$$\text{GRADO_IN}(w) = \text{GRADO_IN}(w) + 1$$

CONTADOR incrementa en 1

//Comprobar si el condición salió y:

// a) devolver la lista de salida con el orden topológico

// b) lanza una excepción de tipo Encuentro

(EI)

$$- T(V, IA) = \sum_{i=1}^{|V|} (|V| + c + a_{Vi}) = |V|^2 + |V|c + |V|a/2;$$

Luego $T(V, IA) = O(|V|^2)$

[a_v es la cantidad de adyacentes de $v \in V$]

2) Recorrido EN AMPLITUD con una estrategia adicional cabos ($O(|V| + |E|)$)

- Se utiliza una cola que pila P para EVITAR el $O(|V|^2)$

- En $Q \neq P$ se van guardando ayudas vértices \checkmark con grado $\text{IN}=0$ o menor que se va trabajando

- Cuando procesamos \checkmark , decrementamos los GRADES_IN de sus adyacentes (w) y verificamos si alguno quedó con $\text{GRADE_IN} = 0$.

- operación SORT_TOPOLOGICA 2 (Arreglo GradosEntrada GI)

crearCola (Q)

Iniciar Q tipo GRADE_IN=0, sin \checkmark ; largo excepción y castea // $O(|V|)$

encolar (Q, v)

mientras (colaNoVacia) hacer

desencolar (Q, v)

marcar \checkmark como visitado con $\text{GRADE_IN}(v) = -1$

agregar A lista de salida (v) =

para c/w adyacente a \checkmark hacer

GRADO_IN(w) redimensionar

si ($\text{GRADE_IN}(w) = 0$) entonces

encolar (Q, w)

//Comprobar si la cola está vacía y:

// a) devolver la lista de salida si lo está.

// b) lanza excepción Decola si no Este Vacío.

|V|

$$- T(V, IA) = \sum_{i=1}^{|V|} (c + a_{Vi}) + |V| = |V| + |E| + |V| = 2|V| + |E|$$

$$\therefore T(V, IA) = O(|V| + |E|)$$

} Sino si (nodo.izquierdo es null) entonces devolver nodo;
 } devolver buscarMin (nodo.izquierdo);

• #insertar : se inserta un nuevo elemento X en el ABB, manteniendo el orden.

• Hay que evitar duplicados, por lo que el detector éste caso, se tendrá que ignorar la inserción ó actualizarse contador de frecuencia.
 • Los duplicados hacen que el árbol sea muy profundo, si no se los evite.

 Publico INSERTAR (elemInsertar) {

 } INSETAR (elemInsertar, raiz);

 PRIVATE INSERTAR (elemInsertar, nodo) {

 Si (nodo = null) {

 devolver nuevaRaiz (elemInsertar);

 Si (nodo.elem < elemInsertar)

 nodo.derecho = INSERTAR (elemInsertar, nodo.derecho),

 Si (nodo.elem > elemInsertar)

 nodo.izquierdo = INSERTAR (elemInsertar, nodo.izquierdo).

 Sino

 se activa el contador; // de duplicados.

 devolver nodo;

 }

• #REMOVER : Al eliminar un nodo, se debe activar el detector previamente. El árbol apuntará a un nuevo nodo en el lugar del que se borró. Este nuevo nodo es el más izquierdo del subárbol derecho correspondiente al nodo a eliminar.

 PRIVADA REMOVER (elemBorrar, nodo)

 Si (nodo = null) devolver nodo; // no se encontró el elemento..

 Si (nodo.elemento < elemBorrar) entonces

 nodo.derecho = REMOVER (elemBorrar, nodo.derecho);

 Si (nodo.elemento > elemBorrar)

 nodo.izquierdo = Remover (elemBorrar, nodo.izquierdo);

 Sino si (nodo tiene 2 hijos) // además es el nodo que se busca ..

 buscar menor Nodo derecho y reemplazar los valores del

 nodo a borrar por el buscado, luego, borrar el nodo buscado.

 // También verificar en el caso que sólo tiene 1 hijo el nodo borrar.

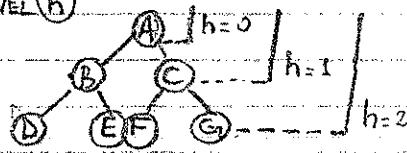
 devolver el nodo con los valores reemplazados.

Siempre se remueve
de las hojas.

Propiedades de algunos Árboles:

- Árbol Lleno: un árbol T de altura $= h$, y grado $= k$, es LLENO si: c/u nodo tiene K hijos y TODAS las hojas están en un NIVEL h .

→ Ejemplo: árbol binario LLENO de $h=2$.



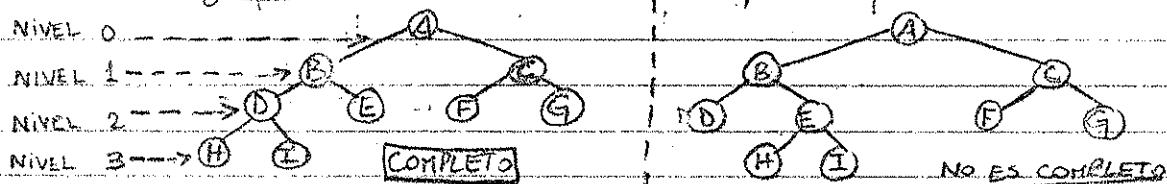
• Un árbol LLENO tiene SIEMPRE:

$$\sum_{i=0}^h K^i = \frac{K^{h+1} - 1}{K - 1} \quad \text{NODOS}$$

Serie geométrica de razón $= k$.

- Árbol Completo: c/u nodo tiene K hijos, en un árbol de altura h , se considera COMPLETO si los nodos en el último nivel h se van insertando de izquierda a derecha.

→ Ejemplo: sea un árbol binario de $h=3$, es completo si:



• Cantidad de nodos N de un árbol completo de grado K y altura h es:

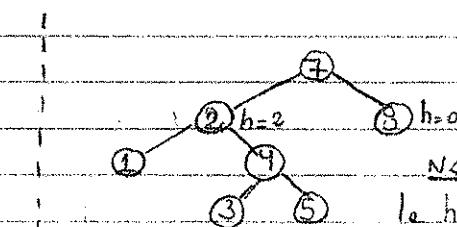
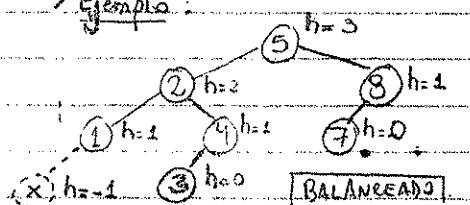
$$\frac{K^h + (K-1)}{K-1} < N < \frac{K^{h+1} - 1}{K-1} \quad \text{N está acotado por las series geométricas de razón } h-1 \text{ y } razon } h.$$

cantidad árbol hasta nivel $h-1$ ≤ cantidad árbol
hijo hasta nivel h

- Árbol Balanceado: un árbol es BALANCEADO si para c/u nodo N se cumple que "las diferencias de altura de cada subárbol de N , difieren en 1 como máximo".

• Una hoja, tiene subárboles de altura $h=1$.

→ Ejemplo:



No es balanceada, ya que $h(2) = 2$ y $h(3) = 0$.
 $\therefore h(2) - h(3) = 2 > 1$.

Árbol AVL

- Es un ABB que cumple con las propiedades de un árbol balanceado.
- Al insertar o remover un nodo, se debe garantizar:
 - 1º Mantener la propiedad de orden (menores a izquierdo, mayores a derecho).
 - 2º Mantener el balance (REBALANCEANDO si es necesario).
 - 3º Modificar la altura de un nodo cuando se es necesario.
- Cada nodo AVL mantiene información sobre su altura.
- La altura de un árbol AVL con N nodos, es de $\text{aprox. } \log N$.
- Todas las operaciones pueden ser llevadas en $O(\log N)$, a excepción de la inserción y la eliminación (no lograble).

OPERACIÓN INSERCIÓN:

- Al insertar se debe actualizar todo la información de balance (altura) del camino desde la raíz hasta el nodo insertado, ya que sólo los nodos de este camino tienen sus subárbolos alterados.
- La inserción puede violar la propiedad de balance en el AVL. Para arreglarlo se debe subir hacia arriba por el camino, encontrar el nodo desbalanceado.

cuando la diferencia de alturas entre los subárboles es igual a 2, se detecta un desbalance.

a) INserción con desbalanceo externo:

* inserción izquierda - izquierda

* inserción derecha - derecha

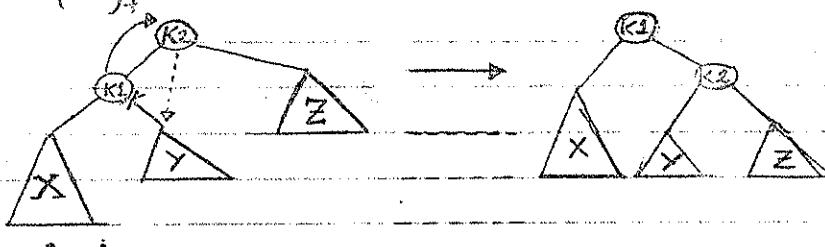
b) INserción con desbalanceo interno:

* inserción izquierda - derecha

* inserción derecha - izquierda

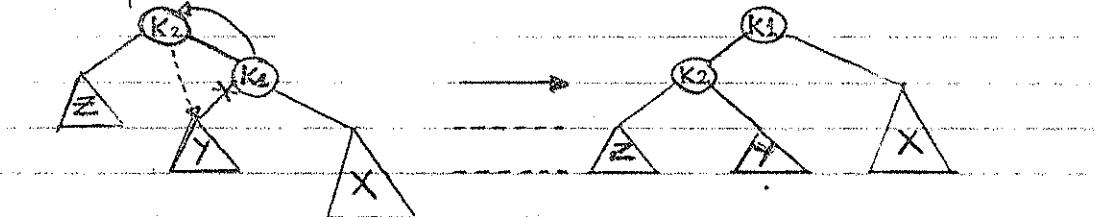
- ROTACIONES: al realizarlas sobre el nodo desbalanceado, se asegura que el AVL seguirá balanceado.
- rotación simple:

- Simple Izquierda



- * K₂ se encuentra desbalanceado, ya que la diferencia de los subárboles X y Z es 2.
- * X sube un nivel, Z desciende otro, mientras que Y se mantiene en el suyo. Además, K₂ se convierte en la nueva raíz, e Y pasa a ser subárbol izquierdo de K₂; mientras que éste, pasa a ser hijo derecho de K₁.

- Simple Derecha



- PSEUDOCÓDIGO:

rotación Simple Izquierdo (i. Nodo AVL K2)

 | Nuevo Nodoo AVL : K1 = K2. izquierda;

 | K2. izquierdo = K1. derecho;

 | K1. derecho = K2;

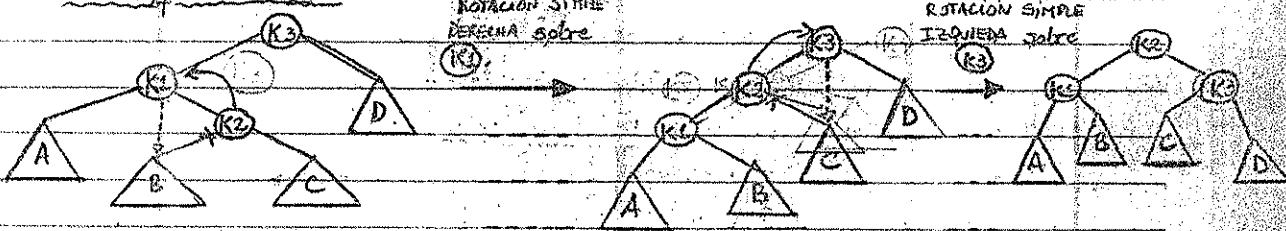
 | Altura (K2) = max (Altura (K2. izquierda), Altura (K2. derecho))

 || Idem con Altura (K1);

 | devolver K1; // K1 es la nueva raíz del subárbol balanceado.

• Rotación Doble: Se basa en realizar dos rotaciones simples en los nodos que participan.

- Doble Izquierda-Derecha:

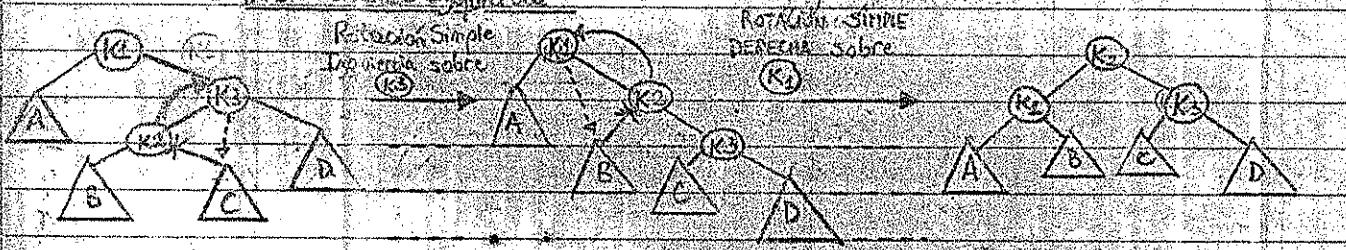


* K3 se encuentra desbalanceado, ya que la altura de K3 = $1\frac{1}{2}$ > altura de D = 1

* Para solucionarla, hay que tomar a que K1 sea hijo izquierdo de K2,

y luego, que K3 sea hijo derecho de K2, cual pasará a ser la nueva raíz.

- Doble Derecha-Izquierda



- PSEUDOCÓDIGO:

rotaciónDoble Izquierdo (Nodo AVL K3)

 | K3. Izquierdo = rotaciónSimple Derecha (K3. Izquierdo);

 | devolver rotación Simple Izquierda (K3);

- IMPLEMENTACIÓN del NODO AVL: similar al NODO BINARIO, a excepción de que mantiene un campo correspondiente a la ALTURA del nodo.

- INSERTAR: "X" es el elemento a insertar.

```

privado INSERTAR (X, nodo) {
    si (el "nodo" es nulo) devolver un NuevoNodoAVL (con X), // lugar correcto
    para insertar NUEVA hoja.
    si (X < nodo.elemento) ...
        nodo.Izquierdo = INSERTAR (X, nodo.Izquierdo);
        si (altura (nodo.Izquierdo) - altura (nodo.Derecho) = 2)
            si (X < nodo.Izquierdo.elemento) // se verifica que rotación hacer,
                nodo = ROTACIÓN SIMPLE Izquierdo (nodo);
            SINO
                nodo = ROTACIÓN Doble Izquierda (nodo);

    SINO. si (X > nodo.elemento)
        nodo.Derecho = INSERTAR (X, nodo.Derecho);
        si (altura (nodo.Derecho) - altura (nodo.Izquierdo) = 2)
            si (X > nodo.derecho.elemento) // se verifica que rotación hacer
                nodo = ROTACIÓN SIMPLE Derecho (nodo);
            SINO
                nodo = ROTACIÓN Doble Derecho (nodo);

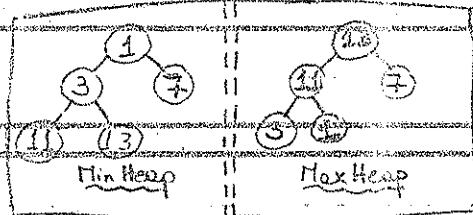
    SINO;
    ; // se evitan los duplicados y no se hace nada.

    ACTUALIZAR ALTURA (nodo) // ...
    devolver nodo;
}

```

- Bonjour :

húsares



- HEAP (colección de prioridades):

* Si una estructura de datos que cumple con 2 propiedades:

- Propiedad ESTRUCTURAL: Una HEAP es un ÁRBOL BINARIO COMPLETO

- Propiedad de ORDEN HEAP:

- MinHeap: el padre de cada nodo (excepto la raíz) tiene una clave menor que sus hijos.

- MaxHeap: el padre de cada nodo (excepto la raíz) tiene una clave mayor que sus hijos.

Con estas propiedades, siempre se asegura que el menor/mayor nodo SIEMPRE esté en la raíz.

Por su simplicidad, una HEAP se puede implementar con un vector (de tamaño variable), donde cada elemento se ubica en (suponiendo el nodo x en la posición " i "):

- el padre (de x) en posición $\text{mod}(i/2)$ en el vector.

- el hijo Izquierdo (de x) en pos. $(2*i)$ en el vector.

- el hijo Derecho (de x) en pos. $(2*i)+1$ en el vector.

Ejemplo:

	X_p	X		X_{izq}	X_{der}
1					
2					
3					
4	X_p	X			
5					
6					
7					
8					
9					
10					

$$\text{pos}(X) = 4 \rightarrow \text{pos}(X_{\text{padre}}) = \lfloor 4/2 \rfloor = 2 \quad | \quad \text{pos}(X_{\text{izq}}) = 4 * 2 = 8$$

$$\text{pos}(X_{\text{der}}) = (4 * 2) + 1 = 9$$

* Operaciones Básicas:

- INSERTAR: Siempre se inserta en el hijo izquierdo de una hoja en el último nivel (si el árbol está lleno) ó en el próximo hijo libre de un nodo (si el árbol no está lleno). Al insertar se debe garantizar que se mantenga la propiedad de orden, mediante el proceso de: PERCOLATE UP (filtrada hacia arriba).

- Máx/Min Remove: El elemento mayor/menor siempre se encuentra en la raíz, por lo que para eliminar el máx/min tenemos que BORRAR LA RAÍZ y REEMPLAZARLO por el último nodo ubicado en el último nivel. Al hacer éste, se debe seguir garantizando la propiedad de orden, por lo que se debe hacer un PERCOLATE DOWN (filtrada hacia abajo).

- Es una operación de $O(n)$

• FILTRADOS

- PERCOLATE UP: se trata de llevar un nodo, de niveles superiores, hacia algún lugar en un nivel inferior de la estructura, hasta que se cumpla la propiedad de orden.

- En este método siempre se ve Comparando y reemplazando un nodo con el padre.

- PERCOLATE DOWN: se trata de llevar un nodo en niveles inferiores (más bajas)

a su lugar correcto en algún nivel superior (más alto) hasta cumplir la propiedad de orden.
- En este método se va reemplazando en cada padre por el menor / mayor de sus hijos.

PSEUDOCÓDIGO: (para una MinHeap)

- INSERTAR:

publico INSERTAR (elem)

 verificar que el arreglo tenga capacidad, sino REDIMENSIONAR;

 posNuevo = arreglo[Tamaño-1];

 while (posNuevo >= 1 AND elem > arreglo [posNuevo / 2]) do { PEROZER_UP
 arreglo [posNuevo] = arreglo [posNuevo / 2],
 posNuevo = posNuevo / 2 ; }

 arreglo [posNuevo] = elem

- MINREMOVE:

publico MinRemove () :

 Verificar si es un árbol vacío;

 elementoMinimo = arreglo [1]

 arreglo [1] = arreglo [Tamaño - 1] // Intercambiar el último elemento con el 1º.

 percolateDown (1); // Filtra el primer elemento hacia abajo.

 devolver elementoMinimo;

procedura percolateDown (pos)

 temp = arreglo [pos]; // Si el (tmp) se encuentra el elemento a FILTRAR

 while (pos * 2 <= tamañoMinHeap) do

 posHijo = pos * 2 ;

 si (posHijo != tamañoMinHeap y arreglo [posHijo + 1] < arreglo [posHijo])

 posHijo = posHijo + 1 ; // me queda con la posición del Hijo con menor valor

 si (arreglo [posHijo] < temp)

 arreglo [pos] = arreglo [posHijo]; // ... e intercambiarlo con el padre si el hijo es menor

 salir

 • Se agrega Salida Bucle;

 pos = posHijo; // Si el padre sigue siendo menor que el hijo, lo sigue FILTRANDO ABAJO.

 arreglo [pos] = temp; // Aquí ya encontré la posición correcta en la HEAP p/ el elem.

- Algunos $T(n)$:

- En Listas Ligadas: • **INSERTAR** en el lugar correcto es de $O(N)$.
 - **BORRARMINIMO** (el principio o el final, manteniendo punteros) es de $O(1)$.
- En ABE: • **INSERTAR** es de $O(\log N)$
 - **BORRARMINIMO** en el extremo Izquierdo. El borrado va a ser proporcional a la altura del árbol. En promedio, es de $O(\log N)$.
- En AVL: • **INSERTAR** es de $O(\log N)$
 - **BORRARMINIMO** es de $O(\log N)$
- En HEAP: • **INSERTAR** en el peor de los casos (insertar un *NewMinimoKey*) es de $O(\log N)$.
 - **ENCONTRARMINIMO** es de $O(1)$, ya que siempre se encuentra en el topo en una *MinHeap*.

- Otros Operaciones HEAP

- **DECREMENTAR CLAVE**: decrementarClave (p, Δ) reduce el valor del ítem en la posición " p " con una cantidad Δ . Para aceptar la propiedad de orden se realiza "percolate-up" (p). Es de $O(\log N)$.
 - **Uso**: en S.O., los administradores pueden substraer prioridad de ejemplos de un proceso.
- **INCREMENTAR CLAVE**: incrementarClave (p, Δ) incrementa el valor de un ítem en la posición " p " con una cantidad positiva Δ . Para aceptar la propiedad de orden se realiza "PERCOLATE-DOWN" (p). Es de $O(\log N)$.
 - **Uso**: en muchos S.O., los planificadores reducen la prioridad del proceso que está consumiendo tiempo excesivo de uso en la CPU.
- **DELETE** - delete (p) remueve el nodo en la posición " p ". Esto es hecho mediante un **DECREMENTAR CLAVE** (p, ∞) (intentando bajar el ítem a la raíz...) y mediante un **MinRemove()**. Es de $O(\log N)$.
 - **Caso**: Cuando un proceso s_1 fuese dejado a finalizar bruscamente éste debe ser removido de la colección de prioridad.
- **CONSTRUIRHEAP**: dado una colección " C " de elementos, se construye a partir de éste una cole de prioridad HEAP.
 - Tras insertando todos los \geq elementos de la C en la HEAP, sería de orden:
 - $O(N)$ en el mejor de los casos, donde $c/$ inserción es de $O(1)$.
 - $O(N \log N)$ en el peor de los casos, donde $c/$ inserción es de $O(\log N)$.
 - Adoptando el siguiente método para construir HEAP, se garantiza un $O(N)$ con los operadores:
 - 1º) **Volcar los elementos en la HEAP sin respetar la propiedad de ORDEN.**
 - 2º) **Comenzando por el antepenúltimo nivel (nodo en la posición $T/2$, donde T es el tamaño de la HEAP), hacer PERCOLATE-DOWN($T/2 - i$), con $i = 0, 1, \dots, T/2$.**

* Mediante este método, se deben hacer 2 comparaciones por nudo:

1) Comparar los hijos y ver el menor de ellos → $\frac{1}{2}$ pesote - down.

2) Sumarlos al nudo con el menor ANTERIOR. \rightarrow si por un nudo que se ingresa

* Para saber cuánto podría llegar la cantidad de comparaciones en el peor de los casos, se debe calcular la suma de altura de todos los nodos, ya que el peor de los casos sería que un nudo INTERNO descendiera una hoja por el filtrado.

Entonces la SUMA DE LAS ALTURAS en un árbol binario completo sería:

$$\sum_{i=0}^{h-1} \sum_{j=0}^{2^i} 1 = \sum_{i=0}^{h-1} 2^i + 1 = 2^{h+1} - 1 - (h+1)$$

Prueba: Un árbol consta en 1 nudo de altura h , 2 nudos de altura $h-1$, 4 nudos de altura $h-2$, ..., 2^i nudos de altura $h-i$. Luego, la suma de las alturas es:

$$S = \sum_{i=0}^{h-1} 2^i (h-i)$$

$$S = h + 2(h-1) + 4(h-2) + \dots + 2^{h-1}(1) + 2^h 0$$

Multiplicando x 2 obtenemos:

$$2S = 2h + 4(h-1) + 8(h-2) + \dots + 2^h(1)$$

$$S+S = 2h + 4(h-1) + 8(h-2) + \dots + 2^h(1)$$

$$S = (2h + 4(h-1) + 8(h-2) + \dots + 2^h(1)) - S$$

Restando los términos correspondientes (p.e., $2h - 2(h-1) = 2$, $4(h-1) - 4(h-2) = 4$, ...):

$$S = -h + 2 + 4 + 8 + \dots + 2^{h-2} + 2^{h-1} + 2^h + 1 - 1 = 0$$

$$\sum_{i=0}^{h-1} 2^i (h-i) = (2^{h+1} - 1) - (h+1)$$

Será grandeza de orden 2.

Como un árbol completo tiene entre 2^h y 2^{h+1} nudos, la suma se demuestra que es de $O(N)$, con $N = \text{número de nudos}$. \therefore el construir HEAP es lineal.

» HEAPSORT: dado un conjunto de elementos sin orden, ésto se puede ordenar (sort) de forma creciente o decreciente - utilizando en una HEAP, siendo éste operación de $O(N \log N)$.

- Con este método se evita la duplicación del espacio causado por el uso de una 2da estructura en donde se encuentren los elementos ordenados.

Método:

1º Dado un vector con los elementos, éstos se convertirán en una MaxHeap o una MinHeap, que empezará por la posición 0.

2º Recorrerá el Max/Min Remove por cada elemento.

3º) Cade X que voy borriendo, en vez de ponerlo en otro vector, lo coloco en la última posición y el último lo coloco en la primera posición, disminuyendo en 1 el tamaño de la HEAP.

4º) Realizar un `perc-rotate-down()` sobre el primer elemento.

- Con una MaxHeap → la ordenación será de menor a mayor.

- MinHeap → la ordenación será de mayor a menor.

Código en Java

`heapSort Descendente (T<extends Comparable<T>> a) {`

`< T > tmp;`

`for (int i = a.length / 2; i >= 1; i--)`

`percDownMax(a, i, a.length); } Construir Heap.`

Este bucle maneja el $T(n)$ [`for (int i = a.length; i >= 1; i--)`

que es de $O(n \log n)$.] `tmp = a[i]; a[i] = a[1]; a[1] = tmp; } deleteMax.`

`percDownMax(a, 1, i - 1); }`

`}`

`// percDownMax (heap, posición, tamañoHeap); jjj; //`

- El problema de la selección del K-ésimo elemento MAYOR

1ºº Solución:

a) Construyo una MaxHeap con los n elementos de entrada.

b) Voy, haciendo sucesivos borrarlos.

c) Al K-ésimo borrado, obtengo el K-ésimo máximo.

• Es de $T(n) = K \cdot \log n \rightarrow O(N \cdot \log N)$

FEDERICO:

Selección K-Máxima (maxHeap H)

```
i = 0; // mantiene el n° de borrados
while (i < k) do
    | deleteMax (H, X);
    | i++;
return X;
```

2ºº Solución:

a) Creo una MinHeap \textcircled{S} con K elementos de la entrada (N) donde.

(Como el K-ésimo elemento es el menor en \textcircled{S} - llamado S_K - se encuentra en la Raiz.)

b) Los elementos x de los $(N-K)$ restantes, serán comparados con la raiz S_K .

- c) Si $x > S_K$, entra en el heap deleteMin() para borrar S_K .
- d) Luego, inserta x y lo filtra hacia arriba.
- Al finalizar de procesar los $(N-K)$ elementos restantes, tendrá el K -ésimo en la raíz.

• T(n)

- + crear Heap con K elementos $\rightarrow O(K)$
- + procesar $(N-K)$ elementos restantes $\rightarrow O(1) \text{ c/v } \left\{ (N-K) \log K \right.$
 $\quad \downarrow \text{percolate up de c/v restante} \rightarrow O(\log K) \quad \left. \right\}$

$$T(n) = O(K + (N-K) \log K) = O(N \cdot \log N)$$

PS: IDEAS:

seleccion (MinHeapDeK(H, (N-K) restantes R))

por c/v ② en R hace

si ($\text{②} > S_K$) then

deleteMin(H);

insertar(H, x); // dentro se hace un percolate up(x)

return (S_K, \dots) // es la raíz de la HEAP.

GRAFO

* Definiciones :

- GRÁFICO : consiste en un conjunto de vértices (V) y un conjunto de aristas (E).

$$G = (V, E)$$

(notación)

- VÉRTICES : son los elementos en donde se encuentran los datos consistentes de la estructura.

- ARISTA : cada una es un par (v, w) , donde $v, w \in V$.

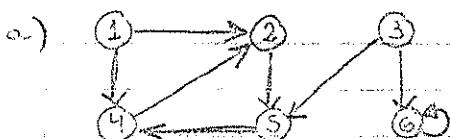
- CAMINO : es una secuencia de vértices $w_1, w_2, w_3, \dots, w_N$ tq $(w_i, w_{i+1}) \in E$ para $1 \leq i < N$.

- LONGITUD (de camino) : número de aristas en un camino, igual a $N-1$.
 El camino $w_1, w_2, w_3, \dots, w_N$.

• La longitud de un camino de un vértice w_i a sí mismo es de 0, si existe $(w_i, w_i) \in E$. En caso contrario, la longitud es 0.

- Camino Simple : camino en el que todos sus vértices son distintos, excepto las vertices el primera y el último.

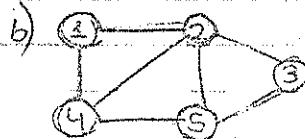
EJEMPLOS



Grafo dirigido $G(V, E)$

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1,2), (1,4), (2,5), (3,5), (3,6), (4,2), (5,4), (6,6)\}$$



Grafo No dirigido $G(V, E)$

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{\{1,2\}, \{1,4\}, \{1,5\}, \{2,4\}, \{2,5\}, \{2,6\}, \{3,4\}, \{3,5\}, \{3,6\}, \{4,5\}, \{5,6\}\}$$

- GRÁFICO DIRIGIDO : es un grafo cuyos vértices son parcialmente ordenados, es decir, que tienen una dirección explícita. Son conocidos también como DIGRAFOS.

• Arista \vec{E} parcialmente ordenada (u, v) . $\vec{U} \rightarrow \vec{V}$

- GRÁFICO NO DIRIGIDO : es un grafo cuyos vértices son no ordenados (simétricos).

• Arista \vec{E} parcialmente ordenada $\{u, v\}$ con $u \neq v$. $\vec{U} - \vec{V}$

- En realidad subyace un grafo con aristas bidireccionalmente.



- CICLO : camino simple desde v_1, v_2, \dots, v_k tq $v_1 = v_k$

Ej: $\langle 2, 5, 4, 2 \rangle$ es un ciclo de longitud 3 en el ejemplo (2).

→ Ciclo : es un ciclo de longitud 1. Ej: $\langle 6, 6 \rangle$ es ciclo de long. 1 en (2).

- GRÁFICO ACÍCLICO : grafo sin ciclos, es decir, grafo en que todos caminos v_1, v_2, \dots, v_k se cumple que $v_i \neq v_k$ para $i, k \in \mathbb{Z}$. Llamado DAG (Directed Acyclic Graph)

- GRAFOS PESADOS: son aquellos en los que cada arista tiene asociado un peso o valor.



Términos

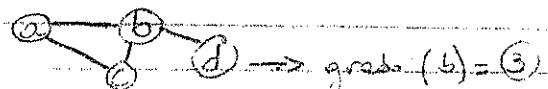
- adyacente: v_i es adyacente a v_j si $\exists (v_i, v_j) \in E$.

• en grafos no dirigidos, $(v_i, v_j) \in E$ incide en los nodos v_i, v_j .

• en dirigidos, $(v_i, v_j) \in E$ incide en v_j y le incidencia proviene de v_i .

- grado:

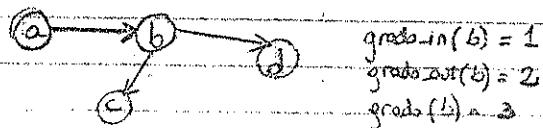
• en No dirigidos, el grado de un vértice es el número de aristas que inciden en él.



• en dirigidos, el grado del vértice será la suma de los grados de entrada y de salida.

- grado de entrada: n.º de aristas que inciden en él.

- grado de salida: n.º de aristas que parten desde él.

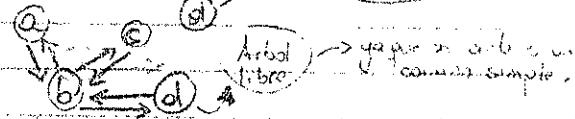


- grado de un grafo: MAXIMO grado de sus vértices.

- BOSQUE: es un grafo sin ciclos.



- ÁRBOL LIBRE: es un bosque conexo.



- Un árbol es un árbol libre en el que un nodo se ha designado como raíz.

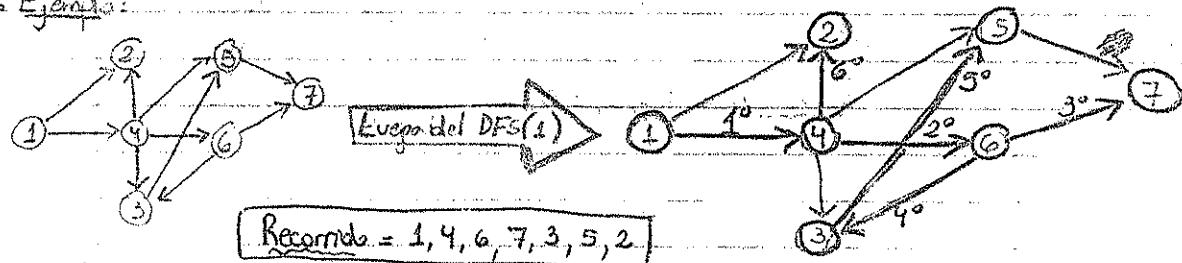
RECORRIDOS sobre el grafo:

- Recorrido en profundidad (DFS) \rightarrow [para dirigidos y grafos no dirigidos]
 - Debemos disponer de un arreglo booleano adicional, llamado ARREGLO DE VISITADOS, y nos permite evitar los ciclos.
- MÉTODO:
 - > Partir de un vértice elegido (V)
 - > Por el nodo nuevo visitado, EXPLORAR cada camino que salga de él.
 - > Hasta que no se haya finalizado de explorar una de las caminos no se comienza con el siguiente.
 - > Un camino deje de explorarse cuando se llegue a un vértice ya visitado.
 - > Si quedan vértices No visitados (por ej.: Y, X, \dots) luego de hacer el recorrido sobre (V), repetir el proceso sobre los faltantes. (ESTO SUCEDE EN GRAFOS DENSOS)
 - (CONEXOS O NO CONEXOS)

ESQUEMA RECURSIVO para $G = (V, E)$

- 1 - Marcar en el arreglo de visitados, todos los vértices como NO visitados (on FALSE).
- 2 - Elección vértice (v) como punto de partida
- 3 - Marcar (v) como visitado
- 4 - $\forall v \in \text{adyacente a } v, (v, u) \in E$, si (v) no ha sido visitado, repetir recursivamente (3) y (4), para (v).
- 5 - Verificar si faltar vértices que procesar

Ejemplos:



PSEUDOCÓDIGO

operación DFS (v : Vértice)

 merca [v] := visitado

 para cada vértice (w) adyacente a (v) hacer

 si merca [w] = noVisitado entonces

 DFS (w)

 finPara

operación main_dfs ()

 var merca : array Boolean de Visitados

 para i/p posición (i) hacer

 merca [i] := noVisitado

 para i/p posición (v) hacer

 si merca [v] = noVisitado entonces

 DFS (modo v)

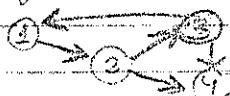
REPRESENTACIONES

- Matriz Adyacencia: Tenemos un arreglo bidimensional $a[|V|][|V|]$, en donde queremos informar sobre las adyacencias entre vértices.
Para un DIGRAFO: para $c/(v, v) \in E$, ponemos $a[v][v]$ en true si $v \sim v$.
son adyacentes. Ponemos false en caso contrario.
Para un digráfico pesado: para $c/(v, v) \in E$, ponemos $a[v][v]$ con el valor del peso correspondiente si es que existe adyacencia. Caso contrario, usamos un cero (0).
(0 por ejemplo) para indicar la No adyacencia.
- Costo Espacial: $O(|V|^2)$
- Pros: usar este representación es apropiada para grafos denses ($|E| \approx |V| \times |V|$) donde la mayoría de los cellos en la matriz contienen información relevante.
- Consultar si (v, v) pertenece a E es constante $\rightarrow T(|V|/|E|) = O(1)$
- Contro: Para grafos que no son denses, se pierde desfondizar mucho espacio ya que no hay demasiadas adyacencias.

• Lista de adyacencias

- Para c/v vértice (v) incluimos de todos los vértices adyacentes.
 - El espacio requerido es lineal o depende: $O(|E| + |V|)$.
 - Si $(v, v) \in E$, entonces se entiende:
 - en DIGRAFOS: poner a v en la lista de adyacencias de v .
 - en NO DIGRAFOS: lo mismo que lo anterior, además de poner a v en la lista de adyacencias de v .
 - En grafos no dirigidos, las adyacencias tienen doble espacia.
- Por lo tanto:
- Si G es dirigido \rightarrow la suma de las longitudes de las listas de adyacencias será $|E|$.
 - Si G es no dirigido \rightarrow la suma será $2|E|$.
 - La búsqueda de adyacencias será lineal $\rightarrow O(|E| + |V|)$.

Dado $G = \{1, 2, 3, 4\}$



Matriz de Adyacencias

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

Lista de adyacencias

1	$\rightarrow [2]$
2	$\rightarrow [3] \rightarrow [4]$
3	$\rightarrow [1] \rightarrow [4]$
4	\emptyset

para cada posición (j) hacer
 si $\text{mire}[j] = \text{NoVisitado}$ entonces
 L BFS (nodo, j)

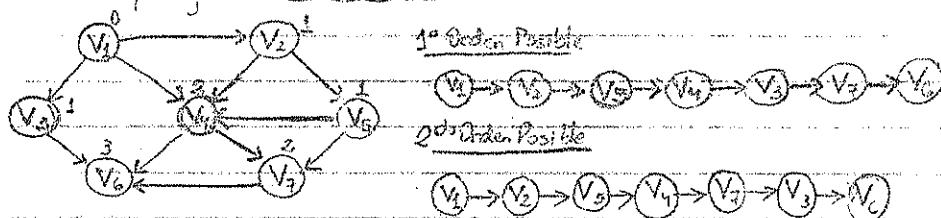
$$\cdot T(VI, IE) = O(VI + IE)$$

SORT TOPOLOGICO:

- La ordenación topológica es una permutación $V_1, V_2, V_3, \dots, V_{VI}$ de los vértices, tal que si $(V_i, V_j) \in E \wedge V_i \neq V_j$, entonces V_i precede a V_j en la permutación.
- Como $V_i \neq V_j$, entonces una ordenación topológica evita los ciclos.
- El SORT TOPOLOGICO se realiza sobre GRAFOS ACICLICOS.

La ordenación topológica no es única:

Ejemplo:



- Para realizar el sort topológico, debes ir tratando progresivamente los vértices INDEPENDIENTES, es decir, aquellos cuyos grados in son 0.
- Por cf vértice trabajada debe reducir los grados in de sus adyacentes.
- Para realizar el orden, NECESITAMOS una estructura arregla con los GRADOS IN de cf vértice (GRADO-IN = grado de entrada).
- Se llama GRAFO DE PREREQUISITOS a un grafo que cumple:
 "Dados, $(V, V) \in E$, no puedo trabajar con (V) si antes no lo hice con (U) .

IMPLEMENTACIONES SORT:

1) Con un RECORRIDO EN AMPLITUD ($O(VI^2)$)

- Para encontrar a los $(V), V \in V$, cuyo GRADO-IN = 0, debes revisar lógicamente el arreglo que contiene los grados de entrada.

- operación SORT-1: $\text{sort-1}(V, V, 1)$ (parametros: gradoEntrada, gradoIN)

ciclo = OnFalso

contador = 1.

mientras (contador $\leq VI$ y ... not(ciclo)) . hace,

 tomar un (V) tal que $\text{GRADO-IN}(V) = 0$. // $O(VI)$

 si $\neq (V)$ entonces ciclo = entruc

 Sino

 marcar a (V) como visitado con $\text{GRADO-IN}(V) = -1$.

$$T(E, V) = O(|V| + |E|)$$

- Recorrido en amplitud o por niveles (BFS)

- para realizar este recorrido

- 1 cole permanentemente las nodos adyacentes
- 1 array booleano de visitados

- MÉTODO :

> Partir de algún vértice (v_0) , visitar v_0 y después, visitar c/u de los vértices adyacentes a v_0 .

> Repetir el proceso para c/u vértice adyacente a v_0 , siguiendo el orden en que fueron visitados.

- ESQUEMA ITERATIVO

1º) Marcar en el array de visitados todos los vértices como NO visitados.

2º) Escojermos un vértice (v_0) , determinante.

3º) Marcar a v_0 como visitado.

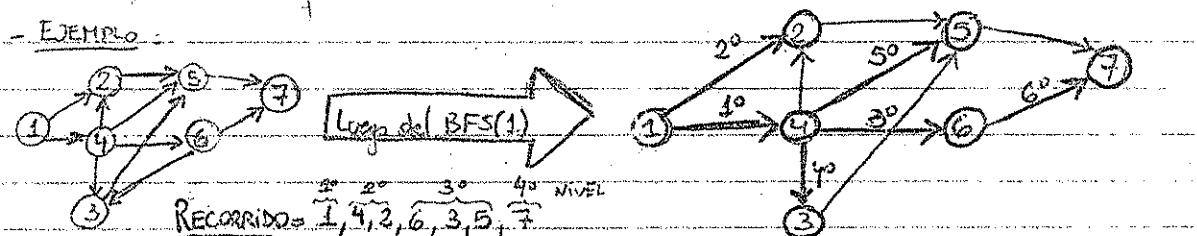
4º) desencolaemos v_0 y lo tratamos.

5º) Encolumnar los adyacentes de v_0 que no hayan sido visitados y la marquemos como visitados.

6º) Seguimos con c/u de v_0 en la lista aplicando (4) y (5). Hasta que ya no se vea.

7º) Verificarse si quedan vértices NO visitados.

- EJEMPLO



- PSEUDOCÓDIGO

operación BFS (G, V)

marca[v] = Visitado

encolar(Q, v)

mientras (coleNoVecia(Q)) hacer

desencolar(Q, v)

tratemos (v)

para cada (w) adyacente a v hacer

Si marca[w] = NoVisitado entonces

marca[w] = Visitado

encolar(Q, w)

main operación_bfs ()

ver marca, = array booleano de visitados .

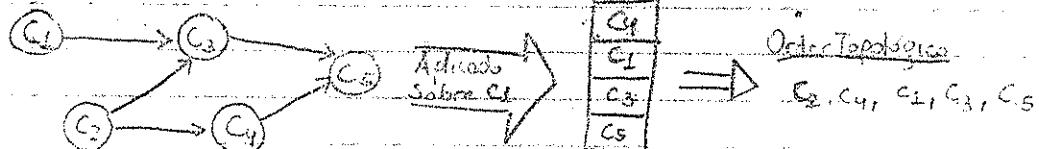
para cada posición(i) hacer

marca[i] = NoVisitado

3) SORT CON RECORRIDO EN PROFUNDIDAD (DFS) →

- Necesitamos una estrucutra de pila $\stackrel{\wedge}{\leftarrow}$ excepto la lista de visitados $\{ \}$.
- Con el DFS busco aquellos vértices que no poseen adyacentes $\stackrel{\wedge}{\leftarrow}$ los voy añadiendo.
- El orden topológico queda determinado por el orden en que voy apilando.
- El último elemento en la pila es el que más grados de entrada posee.
- Apila cuando, dentro del DFS, se acaba el bucle que se ejecuta sobre los adyacentes.

Ejemplo:



4) Algoritmos de caminos de costo mínimo

- Dado un grafo pesado, donde $c_f(v_i, v_j) \in |E|$ tiene un peso definido como c_{ij} :

• el costo del camino v_1, v_2, \dots, v_n es $\sum_{i=1}^{n-1} (v_i, v_{i+1})$.

- Para grafos sin peso, el costo de un camino es la cantidad de aristas.

PROBLEMA: Dado un V inicial \odot , se busca el camino menor costoso desde \odot al resto de los nodos.

→ Algoritmo de costomínimo NO PESADO $[O(V^2)]$

- Desde el vértice inicial \odot , haremos un BFS, de forma tal que el costo de un camino hasta el nivel N , sea de N aristas.

- Cada vértice mantiene cierta info en su estructura:

- d_v : es el costo del camino mínimo (el comienzo desconocido, ∞).
- p_v : indica el nodo previo del camino mínimo $S \rightarrow v$ (el comienzo marcado como 0).
- conocido: índice si ese nodo fue procesado previamente (comienza en FALSE).

- Cuando un vértice está "conocido", se garantiza de que es camino menor costoso.

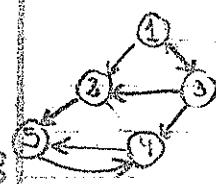
- Al procesar un vértice v , cada adyacente w se actualiza:

- Si $(d_w > d_v + c_{v,w}) \rightarrow d_w = d_v + c_{v,w}$
- Si se cumple lo anterior $\rightarrow p_w = v$.

• El camino se construye haciendo el camino inverso a través de p_v .

Ejemplo:

Dado el grafo G , sea \odot el vértice 1:



V	Con	d_v	p_v
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0

V	con	d_v	p_v
1	T	0	0
2	F	1	1
3	F	1	1
4	F	0	0
5	F	0	0

V	con	d_v	p_v
1	T	0	0
2	T	0	0
3	T	1	1
4	F	2	3
5	F	2	3

- Procesar el vértice 4 y 5 no cambia la última tabla.
- El camino mínimo desde 2 a 4 (usando el p.) es: $1 \rightarrow 3 \rightarrow 4$, siendo $p_1=3$ y $p_2=1$. Se puede hacer mediante un método recursivo.

*PSEUDOCÓDIGO:

```

PROCEDURE sinPeso (Vértice s) {
    para c/ vértice  $\forall$  hacer
        v.dist =  $\infty$ ;
        v.conocido = falso;
        s.dist = 0  $\rightarrow$  distActual
    bucle① ← DESDE ④ HASTA lo CANT.VERTICES hacer (distActual++)
    bucle② ← por c/ vértice  $\forall$  hacer
        si ( !v.conocido & v.dist = distActual ) entonces
            v.conocido = TRUE;
            para c/ adyacente  $\forall$  a  $\forall$  hacer
                si (w.dist =  $\infty$ ) entonces
                    w.dist = distActual;
                    w.previo = v;
}

```

- El tiempo de ejecución es de $O(M^2)$ por los bucles anidados.

MEJORAMIENTO

- Se puede usar una cola para guardar los adyacentes.

- El campo "conocido" no se usa, ya que una vez procesado no puede entrar en la cola de vuelta.

*Pseudocódigo

```

PROCEDURE conCola (vertice S) {
    cola Q;
    vértices v, w;
    encolar (S, Q);
    bucle1 → |v| veces ← mientras no esté ( $\emptyset$ ) hacer
        desencolar (v, Q);
        para c/  $\forall$  adyacente a  $\forall$  hacer  $\rightarrow$  bucle2 → |E| veces
            si (w.dist =  $\infty$ ) entonces
                w.previo = v;
                w.dist = v.dist + 1;
                encolar (w, Q);
}

```

- El tiempo de ejecución es $O(|V| + |E|)$.

Algoritmos de costo mínimo PESADO

DÍJKSTRA

- Cada vértice mantiene los mismos datos que el grado sin peso.
- En i^{a} etapa del algoritmo, se selecciona el vértice $\circled{1}$ de menor distancia d_v entre los vértices DESCONOCIDOS. Luego se actualizan los datos de los adyacentes.
- || Dado un vértice S , elegir vértice \circled{V} que esté a la menor distancia de S .
- ACTUALIZACIÓN DE ADYACENTES: Sea \circled{W} un adyacente de \circled{V}

$$\text{si } (d_w > d_v + c_{vw}) \rightarrow d_w = d_v + c_{vw}$$

- Se prueban todas las caminos posibles.

• Dijkstra funciona correctamente si no hay aristas negativas.

• El costo de un camino hacia sí misma es 0.

EJEMPLO:



1^a

V	con	d_v	p_v																
1	F	0	0	1	T	0	0	1	T	0	0	1	T	0	0	1	T	0	0
2	F	oo	0	2	F	2	1	2	F	2	1	2	T	2	1	2	T	2	1
3	F	oo	0	3	F	oo	0	3	F	3	4	3	F	3	4	3	T	3	4
4	F	oo	0	4	F	1	1	4	T	1	1	4	T	1	1	4	T	1	1
5	F	oo	0	5	F	oo	0	5	F	3	4	5	F	3	4	5	T	3	4
6	F	oo	0	6	F	oo	0	6	F	9	4	6	F	9	4	6	F	8	3
7	F	oo	0	7	F	oo	0	7	F	5	4	7	F	5	4	7	F	5	4

2^a

3^a

4^a

5^a

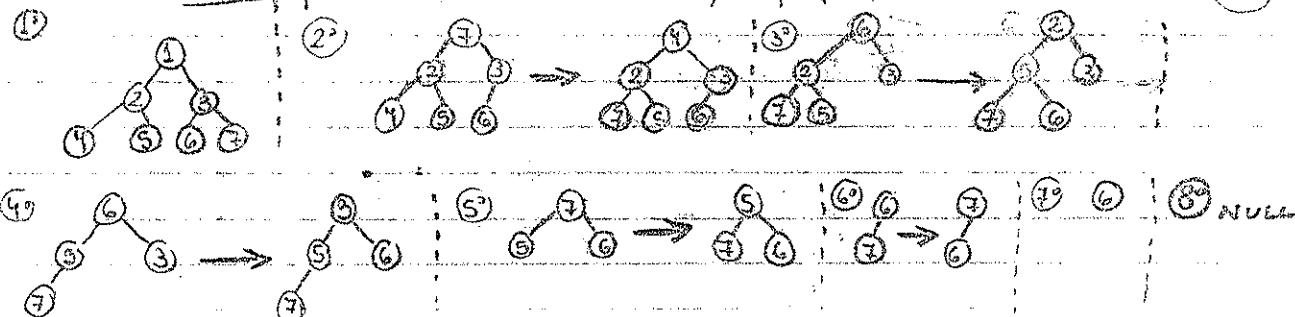
6^a

7^a

8^a

V	con	d_v	p_v	V	con	d_v	p_v
1	T	0	0	1	T	0	0
2	T	2	1	2	T	0	1
3	T	3	4	3	T	3	4
4	T	1	1	4	T	1	1
5	T	3	4	5	T	3	4
6	F	6	7	6	T	6	7
7	T	5	4	7	T	5	4

* Usando minHeap para calcular el mínimo en i^{a} etapa (cada nodo usa como clave el d_v)



* Cada etapa se divide en 2: la primera es con el mínimo eliminado sin realizar percolate down (raíz). La segunda se realiza el percolate down (raíz) una vez actualizadas las distancias de los adyacentes al mínimo sacado.

- PSEUDOCÓDIGO (sin minHeap)

```

DIJKSTRA ( $G, w, s$ ) {
    //  $w$ =adyacente,  $s$ =origen.
    (1) para c/u vértice  $v \in |V|$  hacer
        (2)    $d_v = \infty$ ;  $p_v = 0$ ;
        (3)    $d_s = 0$ ;
    (4) para c/u vértice  $v \in |V|$  hacer
        (5)   Vértice  $u = \text{vérticeDeMenorDistancia}()$ ;
        (6)   Marcar a  $u$  se como conocido;
    (7) para c/u vértice  $w \in |V|$  adyacente a  $u$  hacer
        (8)   si ( $w$  no está conocido) entonces
        (9)       si ( $d_w > d_u + C_{u,w}$ ) entonces
        (10)           $d_w = d_u + C_{u,w}$ ;
        (11)           $p_w = u$ ;
    }
    {Rearranciar HEAP, si q. p. se usa?}
}

```

$$- T(n) = O(|V|^2)$$

- el bucle (4) se ejecuta p/todos los vértices $\rightarrow |V|$ iteraciones
- La operación vérticeDeMenorDistancia (5) es $O(|V|)$ y nota que se realiza $|V|$ veces \rightarrow el costo total de la operación (5) es $O(|V|^2)$.
- El bucle (7) se ejecuta para los adyacentes de c/u vértice. El número total de iteraciones será la cantidad de aristas del grafo $\rightarrow |E|$ iteraciones

$$\cdot T(|V|, |E|) = \sum_{i=1}^{|V|} (|V| + |\alpha_{v_i}|) + |V| \approx |V|^2 + |E| + |V|; \text{ donde } \alpha_{v_i} = \text{adyacentes de } v;$$

- PSEUDOCÓDIGO (optimización usando MinHeap)

- previamente se tendría que haber pasado una heap como parámetro... mantener de los vértices ordenados por su d_v .
- Se reemplaza la linea (5) por vérticeDeMenorDistancia (H, u), el cual hará un selectMin (H) y lo guardará en u (no se realiza perc-down (relaj)).
- Como linea 12, se agrupa fuera del 2º bucle (7) el perc-down (relaj), una vez actualizadas las distancias de los adyacentes.

$$T(n) = O(|E| \cdot \log |V|)$$

- La operación vérticeDeMenorDistancia (H, u) es $O(\log |V|)$ y como se realiza $|V|$ veces \rightarrow el costo de la operación es de $O(|V| \log |V|)$.

• El bucle (7) supone modificar la prioridad (distancia) y reorganizar la heap. Cada iteración es de $O(\log |V|)$ \rightarrow realiza $|E|$ iteraciones $\rightarrow O(|E| \log |V|)$.

- El costo total del algoritmo es $(|V| \log |V| + |E| \log |V|)$ es

$$O(|E| \log |V|)$$

- PSEUDOCÓDIGO (variante de la Heap).

Dijkstra (G, w, s, H): { // $d_v = \text{dist}$, $c_{u,v} = \text{coste arista } (u, v)$.

para c/ vértice $v \in |V|$ hacer

| $d_v \infty$; $p_v = 0$;

| $d_s = 0$; insert-Heap(S, d_s);

para c/ vértice $v \in |V|$ hacer

| delete-min(H, u);

| mientras (u) esté tratado hacer

| | delete-min(H, u);

| $u.con = \text{TRUE}$;

| para c/ adyacente (u, v) al vértice (u) hacer

| | si ($v.con = \text{FALSE}$) entonces

| | | si ($w.dist > u.dist + c_{u,v}$) entonces

| | | | $w.dist = u.dist + c_{u,v}$;

| | | | insert-Heap($w, w.dist$);

}

- La variación es que se inserta c/ adyacente w y su d_w , y vez que se modifica.

$$\bullet T(n) = O(|E| \log |V|)$$

- El tamaño de la heap puede crecer hasta $|E|$.

Dado que $|E| \leq |V|^2 \rightarrow \log |E| \leq 2 \log |V|$, el costo total no varía.

$$\bullet T(|V|, |E|) = \sum_{i=1}^{|V|} (a^{in} \log |E| + c_1 + a_{uv} (c_2 + \log |E|)) =$$

ya que $\log |E| = 2 \log |V|$

$$= |E| \cdot \log |E| + |V| c_1 + |E| c_2 + |E| \log |E| = (c_1 \log |V| + c_1 |V| + c_2 |E|)$$

$$\boxed{T(|V|, |E|) = O(|E| \log |V|)}$$

Algoritmos de costo mínimo c/ aristas NEGATIVAS.

- Se lleva a cabo utilizando una cole auxiliar que vaya almacenando los adyacentes.
- Los vértices guardan la misma info que antes, excepto el campo de "conecto".

PSEUDOCÓDIGO

PSONEGATIVE (G, w, s): {

$q = \text{cole de adyacentes};$

para c/ vértice (v) hacer

| $d_v = \infty$;

$d_s = 0$;

$q.encolar(s)$;

mientras coleNoVacia(q) hacer

| Vértice $v = q.desencolar()$;

```

    para c/ vértice (w) adyacente a (v) hacer
        si ( $d_w > d_v + c_{v,w}$ ) entonces
             $d_w = d_v + c_{v,w}$ ;
             $p_w = v$ ;
            si (w no esté en q) entonces
                q. encolar(w);

```

T(n)

- Dada que c/vértice puede ser desencolado como máximo |V| veces, el

$$T(|E|, |V|) = O(|E| \cdot |V|)$$

Caminos mínimos p/ GRAFOS ACÍClicos (Optimización del algoritmo de Dijkstra)

- Mediante el orden topológico.

- La selección de cada vértice se realiza siguiendo el orden topológico.

- Esta estrategia funciona correctamente, dado que al seleccionar un vértice \textcircled{v} , no se va a encontrar una distancia d_v menor, porque ya se procesaron todos los caminos que llegan a él.

- Costo total del algoritmo: $T(|V|, |E|) = O(|E| + |V|)$

Caminos mínimos p/ todos los pares de vértices

Algoritmo de Floyd ($O(|V|^3)$)

- Se utilizan 2 matrices de dimensión $|V| \times |V|$:

- una matriz D de costos mínimos.

- una matriz P de vértices intermedios (son los que se recorren para ir desde el origen \textcircled{S} hasta el final \textcircled{T}).

- PSEUDO CÓDIGO

```

para k:=1 hasta cont.Vértices (G) hacer

```

```

    para i:=1 hasta cont.vértices (G) hacer

```

```

        para j:=1 hasta cont.vértices hacer

```

```

            si ( $D[i,j] > D[i,k] + D[k,j]$ ) entonces

```

```

                 $D[i,j] = D[i,k] + D[k,j]$ ;

```

```

                 $P[i,j] = k$ ;

```

- Ver página 426 del libro.

Arbol de expansión mínima (AEM)

- Estos algoritmos se realiza sobre GRAFOS NO DIRIGIDOS Y CONEXOS

Definición:

El árbol de expansión mínima de un grafo G no dirigido y conexo, es un árbol formado por los vértices de G que contienen todas las vértices con un costo total mínimo.

PRIM: - El árbol crece por etapas.

- En i etapa se:

- elige un vértice como raíz.

- le agrega al árbol una arista y un vértice asociado.

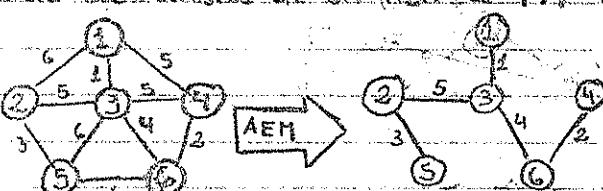
- Se seleccionan aquellas aristas (u, v) de mínimo costo que cumplen:

Mi árbol y raíz actual.

- para evitar los ciclos.

- El número de aristas de un AEM lés $|V| - 1$.

EJEMPLO



- La implementación es muy similar al algoritmo de Dijkstra. Solo cambia la forma de ACTUALIZAR LAS DISTANCIAS: se actualiza D_w con $C_{v,w}$ si $D_w > C_{v,w}$.

- Ahora d_v es el peso del arista mínima que conecta v con algún vértice conocido y p_v es el último vértice en causar el cambio en d_v .

EJEMPLO

v	con d _v p _v	v con d _v p _v					
1 F 0 0	1 T 0 0	1 T 0 0	1 T 0 0	1 T 0 0	1 T 0 0	1 T 0 0	1 T 0 0
2 F ∞ 0	2 F ∞ 0	2 F 6 1	2 F 5 3	2 F 5 3	2 F 5 3	2 F 5 3	2 F 5 3
3 F ∞ 0	3 F 1 1	3 T 1 3	3 T 1 3	3 T 1 3	3 T 1 3	3 T 1 3	3 T 1 3
4 F ∞ 0	4 F 5 1	4 F 5 1	4 F 5 1	4 T 2 6	4 T 2 6	4 T 2 6	4 T 2 6
5 T ∞ 0	5 F ∞ 0	5 T 6 3	5 T 6 3	5 F 6 3	5 F 6 3	5 F 6 3	5 T 3 2
6 F ∞ 0	6 T ∞ 0	6 T 9 3	6 T 9 3	6 T 9 3	6 T 9 3	6 T 9 3	6 T 9 3

Orden de Ejecución

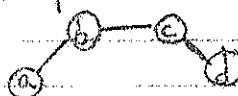
• Sin usar minHeap $\rightarrow T(M, |E|) = O(M^2)$

• usando minHeap $\rightarrow T(M, |E|) = O(|E| \log |V|)$

ALGORITMO DE KRUSKAL (O(n log(n)))

- Se van seleccionando las aristas en orden creciente según su peso y las acepte si no originan un ciclo. SIEMPRE SE EVITAN LOS CICLOS (sino no sería un árbol).
- Cada vértice se encuentra ordenado en una matriz, al recuperar el costo mínimo en el paso.
- Inicialmente, cada vértice pertenece a su propio conjunto (en total de una componente conexa) $\rightarrow \{V\}$ conjuntos con un único elemento.
- Se intenta ir agregando aristas hasta lograr que todos los vértices formen un AEM (una sola componente conexa).
- Al aceptar una arista se realiza la unión de 2 conjuntos (componentes conexas).
- Si los vértices (u) y (v) están en el mismo conjunto, la arista (u, v) es RECHAZADA porque al aceptarla formaría un ciclo.
- En el punto del proceso, 2 vértices pertenecen al mismo conjunto si están conectados.

Componente ①

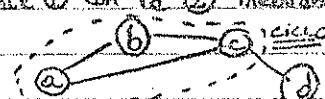


- "a está conectado con b"
- "a está conectado con c"
- "a está conectado con d"
- (etc)

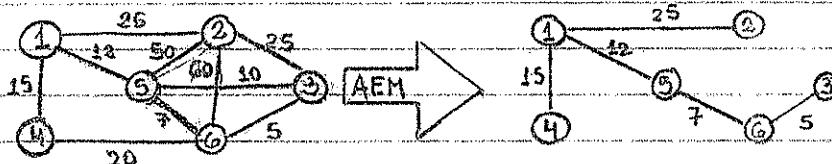
Componente ②



Unir la componente ① con la ② mediante la arista (a, c) , originaría un ciclo:



EJEMPLO

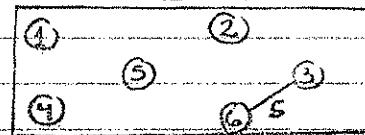


* Tenemos las aristas: $(3,6)$; $(5,6)$; $(5,3)$; $(4,5)$; $(4,1)$; $(4,6)$; $(1,2)$; $(2,3)$; $(5,2)$; $(2,6)$.

* Tenemos los conjuntos (componentes conexas): $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$

1er etapa:

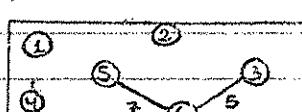
ARISTA: $(3,6)$; se unen los conjuntos $\{3\}$ y $\{6\}$.



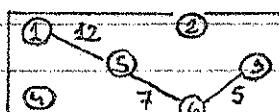
2da ETAPA:

ARISTA: $(5,6)$; se unen los conjuntos $\{5\}$ y $\{6\}$.

ya que $\{6\} \in$ Conjunto y $\{5\} \notin$ Conjunto

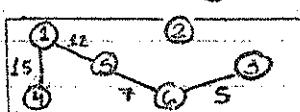


3er etapa: ARISTA: $(5,3)$; No se agrega esta arista porque causaría el ciclo $5-3-6-5$. El gráfico queda igual.
ya que $\{5\} \in$ Conjunto y $\{3\} \notin$ Conjunto.



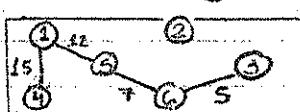
4ta etapa: ARISTA: $(1,5)$; se unen conjuntos $\{1\}$ y $\{5-6-3\}$.

ya que $\{5\} \in$ Conjunto y $\{1\} \notin$ Conjunto.



5ta etapa: ARISTA: $(1,4)$; se unen $\{1\}$ y $\{1-5-6-3\}$

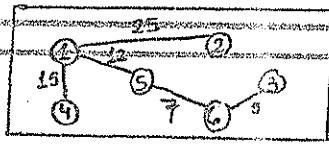
ya que $\{1\} \in$ Conjunto y $\{4\} \notin$ Conjunto.



6ta etapa: ARISTA: $(4,6)$; No se agrega porque causaría el ciclo $4-6-5-1-4$. El gráfico queda igual.

ya que $\{1\} \in$ Conjunto y $\{6\} \in$ Conjunto.

7^a etapa: ARISTA: (1,2) ; se unen ② y ④-1-5-6-3
ya que ① ∈ Conjunto y ② ∉ Conjunto.



- El procedimiento finaliza cuando se tienen $|V|-1$ aristas agregadas.

- Tiempo de Ejecución $T(|V|, |E|) = O(|E| \log |V|)$

- Se organizan las aristas en una minHeap para OPTIMIZAR la recuperación del mínimo
- El tamaño de la minHeap es $|E|$ y extraer α arista es de $O(\log |E|)$.
→ extraer todas las vértices en el peor de los casos es $O(|E| \log |E|)$.
- Como $|E| \leq |V|^2$, $\log |E| \leq \log |V|^2 = 2 \log |V|$
→ el coste Total es de $O(|E| \log |V|)$.

୩୧