Apunte de Cátedra

ESTRUCTURAS DE DATOS

Analista de Sistemas y Licenciatura en Sistemas

Lic. Verónica L. Vanoli
Dra. Sandra I. Casas

Universidad Nacional de la Patagonia Austral
Unidad Académica Río Gallegos

Indice

Indice

Indice

ANALISIS DE ALGORITMOS

1.- INTRODUCCION

Un algoritmo es un conjunto de instrucciones sencillas, claramente especificado, que se debe seguir para resolver un problema. Una vez que se da un algoritmo para un problema y se decide que es correcto, un paso importante es determinar la cantidad de recursos, como tiempo y espacio, que requerirá. Un algoritmo que resuelve un problema, pero tarde un año en hacerlo, difícilmente será de utilidad. De manera similar, un algoritmo que necesita un gigabyte de memoria principal no es útil.
Es necesario estudiar:

- Como calcular el tiempo que emplea un programa.
- Como reducir el tiempo de ejecución de un programa.
- Los resultados del uso indiscriminado de la recursión.

2.- SOPORTE MATEMATICO

En general, el análisis requerido para estimar el uso de recursos de un algoritmo es una cuestión teórica, y por lo tanto, necesita un marco formal. Comenzamos con algunas definiciones matemáticas:
Las siguientes cuatro definiciones serán válidas en todo nuestro estudio:

Definición: $T(n) = O(f(n))$ si existen constantes c y n_0 tales que $T(n) \leq c\ f(n)$ cuando $n \geq n_0$
Definición: $T(n) = \Omega(g(n))$ si existen constantes c y n_0 tales que $T(n) \leq c\ g(n)$ cuando $n \geq n_0$
Definición: $T(n) = \Theta(h(n))$ sí y solo sí $T(n) = O(h(n))$ y $T(n) = \Omega(h(n))$.
Definición: $T(n) = O(p(n))$ sí $T(n) = O(p(n))$ y $T(n) \neq \Theta(p(n))$

El objetivo de estas definiciones es establecer un orden relativo entre funciones. Dadas dos funciones, por lo general hay puntos donde una función es menor que la otra, de modo que no tienen sentido afirmar que, por ejemplo, $f(n) < g(n)$. Así, se comparan sus *tasas de crecimiento relativas*. Cuando esto se aplica al análisis de algoritmos, se verá por qué ésta es la medida importante.
Aunque 1000n es mayor que n^2 para valores pequeños de n, n^2 crece con una tasa mayor, y así, n^2 finalmente será la función mayor. El punto de cambio es, en este caso, n = 1000. La primera definición dice que finalmente existe un punto n_0 pasado el actual $c \cdot f(n)$ es siempre al menos tan grande como T(n), de tal modo que se ignoran los factores constantes, $f(n)$ es al menos tan grande como T(n). En este caso, se tiene $T(n) = 1000n$, $f(n) = n^2$, $n_0 = 1000$ y c = 1. Se podría usar $n_0 = 10$ y c = 100. Así se puede decir que $1000n = O(n^2)$ (orden n cuadrada). Esta notación se conoce como *O grande*. Con frecuencia, en vez de decir “orden ...”, suele decirse “O grande..”.
Si se usan los operadores tradicionales de desigualdad para comparar las tasas de crecimiento, entonces la primera definición dice que la tasa de crecimiento de T(n) es menor o igual (≤) que la de $f(n)$. La segunda definición, $T(n) = \Omega(g(n))$ (dígase “omega”), dice que la tasa de crecimiento de T(n) es mayor o igual (≥) que la de g(n). La tercera definición, $T(n) = \Theta(h(n))$ (dígase “theta”), dice que la tasa de crecimiento de T(n) es igual (=) a la de h(n). La ultima definición, $T(n) = o(p(n))$ (dígase “o pequeña”), dice que la tasa de crecimiento de T(n) es menor (<) que la tasa de crecimiento de p(n). Esta es diferente de la O grande, porque O grande permite que las tasas de crecimiento se igualen.
Para demostrar que alguna función $T(n) = O(f(n))$, no se suelen aplicar estas definiciones formalmente, sino que se usa un repertorio de resultados conocidos. En general, esto significa que una demostración es un proceso muy sencillo y no debe implicar cálculo, excepto en circunstancias extraordinarias.
Cuando se dice que $T(n) = O(f(n))$, se esta garantizando que la función T(n) crece a una velocidad no mayor que $f(n)$; así $f(n)$ es una *cota superior* de T(n). Como esto implica que $f(n) = \Omega(T(n))$, se dice que T(n) es una *cota inferior* de $f(n)$.
Por ejemplo, n^3 crece más rápido que n^2, así se puede decir que $n^2 = O(n^2)$ o $n^3 = \Omega(n^2)$. $f(n) = n^2$ y $g(n) = 2n^2$ crecen a la misma velocidad, así que ambas, $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$, se cumplen. Cuando dos funciones crecen a la misma velocidad, la decisión de representar esto o no con $\Theta()$ puede depender del contexto particular. Intuitivamente, si g(n) = 2n, entonces $g(n) = O(n^4)$, $g(n) = O(n^3)$ y $g(n) = O(n^2)$ son

técnicamente correctas, pero es obvio que la ultima opción es la mejor respuesta. Escribir $g(n) = \Theta(n^2)$ dice no solo que $g(n) = O(n^2)$, sino también que el resultado es tan bueno (exacto) como es posible.
Las cosas importantes a saber son:

Regla 1:
Si $T_1(n) = O(f(n))$ y $T_2(n) = O(g(n))$, entonces
(a) $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$
(b) $T_1(n) * T_2(n) = O(f(n) * g(n))$

Regla 2:
Si T(x) es un polinomio de grado n, entonces $T(x) = \Theta(x^n)$

Regla 3:
$\log^k n = O(n)$ para cualquier k constante. Esto indica que los logaritmos crecen muy lentamente.

Para ver que la regla 1(a) es correcta, nótese que por definición existen cuatro constantes c_1, c_2, n_1 y n_2, tales que $T_1(n) \leq c_1 f(n)$ para $n \geq n_1$ y $T_2(n) \leq c_2\ g(n)$ para $n \geq n_2$.Sea $n_0 = \max(n_1, n_2)$. Entonces, para $n \geq n_0$, $T_1(n) \leq c_1 f(n)$ y $T_2(n) \leq c_2\ g(n)$, de modo que $T_1(n) + T_2(n) \leq c_1 f(n) + c_2\ g(n)$. Sea $c_3 = \max(c_1, c_2)$. Entonces,

$$\begin{aligned} \mathbf{T_1(n) + T_2(n)} &\leq \mathbf{c_3 f(n) + c_3\ g(n)} \\ &\leq \mathbf{c_3(f(n) + g(n))} \\ &\leq \mathbf{2c_3\ \max(f(n), g(n))} \\ &\leq \mathbf{c\ \max(f(n), g(n))} \end{aligned}$$

para $c = 2c_3$ y $n \geq n_0$

Y así podrían demostrarse las otras relaciones dadas anteriormente. Esta información es suficiente para ordenar por tasas de crecimiento la mayoría de las funciones comunes:

Función	***Nombre***
c	constante
log n	logarítmica
$\log^2 n$	logarítmica cuadrada
n	lineal
n log n	
n^2	cuadrática
n^3	cúbica
2^n	exponencial

Hay varios puntos que destacar. Primero, es muy mal estilo incluir constantes o términos de orden menor en una O grande. No se debe decir $T(n) = O(2n^2)$ o $T(n) = O(n^2 + n)$. En ambos casos, la forma correcta es $T(n) = O(n^2)$. Esto significa que en cualquier análisis que requiera una respuesta O grande, todos los tipos de simplificaciones son posibles. Por lo regular pueden ignorarse los términos de menor orden y desecharse las constantes. La precisión que se requiere en estos casos es considerablemente menor.
En segundo lugar, siempre se pueden determinar las tasas de crecimiento relativo de dos funciones $f(n)$ y g(n) mediante el cálculo $\lim_{n\to\infty} f(n) / g(n)$, usando la regla de L'Hôpital[1] si es necesario.
El límite puede tener uno de cuatro valores:

- El límite es 0: esto significa que $f(n) = o(g(n))$.
- El límite es $c \neq 0$: esto significa que $f(n) = \Theta(g(n))$.
- El límite es ∞: esto significa que $g(n) = o(f(n))$.
- El límite oscila: no hay ninguna relación.

[1] La regla de L'Hôpital establece que si $\lim_{n\to\infty} f(n) = \infty$ y $\lim_{n\to\infty} g(n) = \infty$, entonces $\lim_{n\to\infty} f(n) / g(n) = \lim_{n\to\infty} f'(n) / g'(n)$ donde f'(n) y g'(n) son las derivadas de f(n) y g(n), respectivamente.

Casi nunca es necesaria la utilización de este método. En general la relación entre $f(n)$ y $g(n)$ se puede obtener por simple álgebra. Por ejemplo, si $f(n) = n \log n$ y $g(n) = n^{1.5}$, entonces para decidir cual de las funciones $f(n)$ y $g(n)$ crece con mayor rapidez, lo que realmente se necesita es determinar que crece más rápidamente, $\log n$ o $n^{1.5}$. Esto es como determinar que crece con mayor rapidez, $\log^2 n$ o n. Este problema es sencillo, porque es bien sabido que n crece más rápidamente que cualquier potencia de un logaritmo. Así, $g(n)$ crece con mayor rapidez que $f(n)$.
Una nota de estilo: es incorrecto decir $f(n) \leq O(f(n))$ porque la desigualdad está implícita en la definición. Es erróneo escribir $f(n) \geq O(g(n))$, pues no tiene sentido.

3.- MODELO

Para analizar algoritmos en un marco formal, se necesita un modelo de computación. Nuestro modelo es básicamente una computadora normal, en la cual las instrucciones se ejecutan de modo secuencial. El modelo tiene el repertorio estándar de instrucciones sencillas, como adición, multiplicación, comparación y asignación, pero a diferencia de las computadoras reales, ésta tarda exactamente una unidad de tiempo en hacer cualquier cosa (sencilla). Para ser razonable, se supondrá que, como una computadora moderna, este modelo tiene enteros de tamaño fijo (digamos 32 bits) y que no tiene instrucciones refinadas, como la inversión de matrices o la clasificación, que claramente no se pueden hacer en una unidad de tiempo. También suponemos una memoria infinita.

Es obvio que este modelo tiene algunas debilidades. En la vida real, por supuesto, no todas las operaciones tardan exactamente el mismo tiempo. En particular, en este modelo una lectura de disco cuenta igual que una adición, aun cuando la adición suele ser varios órdenes de magnitud más rápida. También, al suponer memoria infinita, nunca hay que preocuparse por faltas de página, lo cual puede ser un problema real, en especial en algoritmos eficientes. Este puede ser un grave problema en muchas aplicaciones.

4.- ¿QUÉ ANALIZAR?

En general, el recurso más importante a analizar es el tiempo de ejecución. Varios factores afectan el tiempo de ejecución de un programa. Algunos, como el compilador y la computadora usada, están mas allá del alcance de cualquier modelo teórico, así que, aunque son importantes, no hemos de tratarlos. Los otros factores relevantes son el algoritmo usado y su entrada.
El tamaño de la entrada suele ser la consideración principal. Se definen dos funciones, $T_{prom}(n)$ y $T_{peor}(n)$, como el tiempo de ejecución promedio y el del peor caso, respectivamente, empleados por un algoritmo para una entrada de tamaño n. Claro esta, $T_{prom}(n) \leq T_{peor}(n)$. Si hay más de una entrada, esas funciones pueden tener más de un argumento.
Cabe señalar que en general la cantidad requerida es el tiempo del peor caso, a menos que se especifique otra cosa. Una razón para esto es que da una cota para todas las entradas, incluyendo entradas particularmente malas, que un análisis del caso promedio no puede ofrecer. La otra razón es que la cota del caso promedio suele ser mucho más difícil de calcular. En algunos casos, la definición de "promedio" puede afectar el resultado. (Por ejemplo, ¿qué es una entrada promedio para el problema siguiente?).
Como ejemplo, se considerará el siguiente problema:

PROBLEMA DE LA SUMA DE LA SUBSECUENCIA MÁXIMA

Dados enteros (posiblemente negativos) $a_1, a_2, ..., a_n$, encontrar el valor máximo de $\Sigma^{j}_{k=i} a_k$ (Por conveniencia, la suma de la subsecuencia máxima es 0 si todos los enteros son negativos).

Ejemplo: Para la entrada –2, 11, -4, 13, -5, -2, la respuesta es 20 (de a_2 hasta a_4).

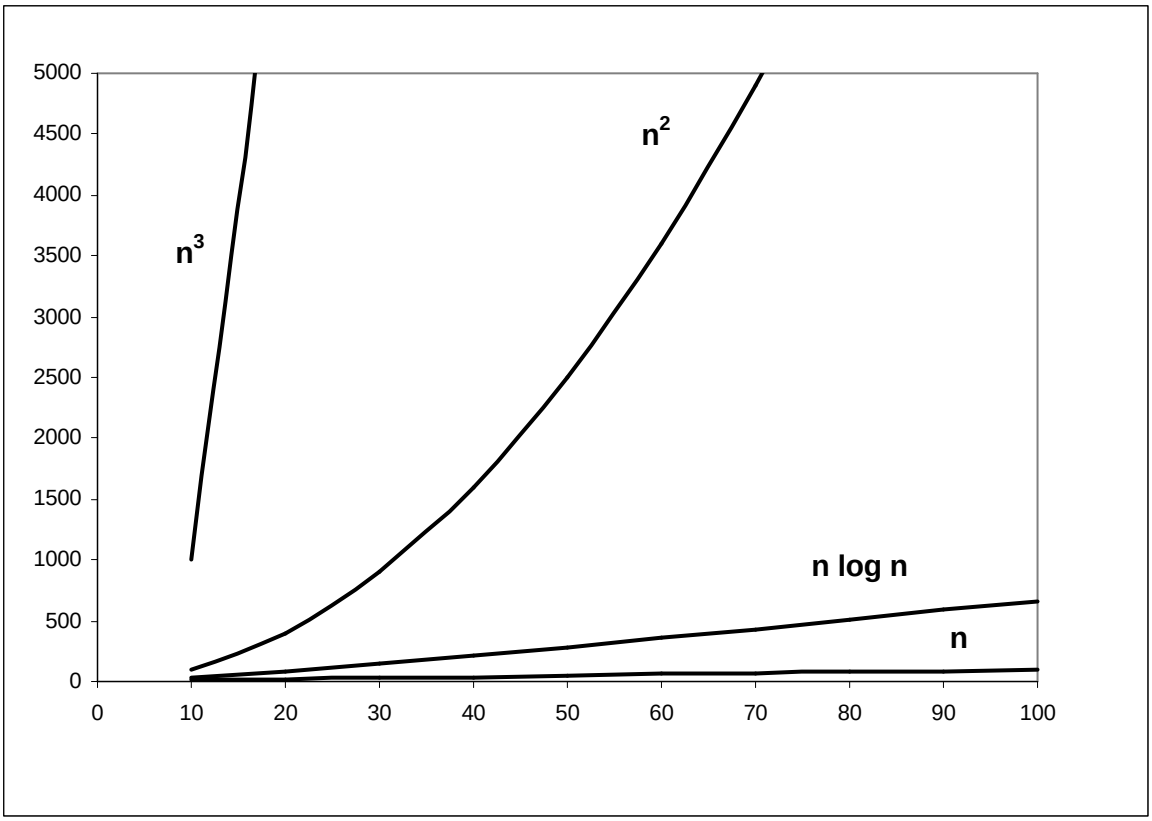
Este problema es interesante sobre todo porque existen muchos algoritmos para resolverlo, y su rendimiento varía drásticamente. Estudiaremos cuatro algoritmos para resolver el problema. En la siguiente tabla se muestra el tiempo de ejecución en cierta computadora (el modelo exacto carece de importancia) para estos algoritmos.

Algoritmo		1	2	3	4
Tiempo		$O(n^3)$	$O(n^2)$	O(n log n)	O(n)
Tamaño de la Entrada	n = 10	0.00103	0.00045	0.00066	0.00034
	n = 100	0.47015	0.01112	0.00486	0.00063
	n = 1000	448.77	1.1233	0.05843	0.00333
	n = 10000	NA	111.13	0.68631	0.03042
	n = 100000	NA	NA	8.0113	0.29832

Hay varias cosas importantes que observar en esta tabla. Para una entrada pequeña, todos los algoritmos se ejecutan en un instante, así que si solo se espera una pequeña cantidad de datos, no valdría la pena esforzarse por tratar de encontrar un algoritmo muy ingenioso. Por otro lado, existe un gran mercado hoy en día de reescritura de programas que fueron creados cinco años atrás o más, basados en la suposición, invalida hoy, de que la entrada es pequeña. Esos programas son demasiados lentos ahora porque usaron algoritmos deficientes. Para entradas grandes, el algoritmo 4 es claramente la mejor opción (aunque el 3 también es utilizable)

Segundo, los tiempos dados no incluyen el tiempo requerido para leer la entrada. Para el algoritmo 4, el tiempo de lectura de la entrada desde un disco es probablemente de un orden de magnitud mayor que el requerido para resolver el problema. Esto es característico en muchos algoritmos eficientes. La lectura de datos suele ser el cuello de botella; una vez que se leen los datos, el problema se puede resolver rápidamente. Para algoritmos ineficientes esto no es cierto, y hay que valerse de recursos de cómputo significativo. Así, es importante que, siempre que sea posible, los algoritmos sean suficientemente eficientes para no volverse el cuello de botella del problema.

La siguiente figura muestra las tasas de crecimiento de los tiempos de ejecución de los cuatro algoritmos. Aun cuando esta gráfica solo presenta valores para n entre 10 y 100, las tasas de crecimiento relativas son evidentes. Aunque el gráfico del algoritmo 3 parece lineal, es fácil verificar que no lo es usando una regla (o una hoja de papel).



Si representáramos gráficamente el rendimiento para valores mayores, las diferencias serían realmente significativas. Lo cual sería una ilustración espectacular de cuan inútiles son los algoritmos ineficientes, incluso para cantidades moderadamente grandes de datos.

5.- CALCULO DEL TIEMPO DE EJECUCION

Hay varias formas de calcular el tiempo de ejecución de un programa. La tabla anterior se obtuvo empíricamente. Si se espera que dos programas tomen tiempos parecidos, es probable que la mejor forma de decidir cual es más rápido sea codificarlos y ejecutarlos.
En general existen varias ideas algorítmicas, y es deseable la eliminación rápida de las ineficaces, por lo que se suele requerir un análisis. Más aún, la habilidad de hacer un análisis permite aprender a diseñar algoritmos eficientes. El análisis también suele detectar cuellos de botella, que vale la pena codificar con gran cuidado.
Para simplificar el análisis, adoptaremos el convenio de que no hay unidades de tiempo particulares. Así, se desechan las constantes iniciales. Tampoco se toman en cuenta los términos de orden menor. Como O grande es una cota superior, nunca se debe subestimar el tiempo de ejecución de un programa. En efecto, la respuesta obtenida es una garantía de que el programa termina en cierto lapso; el programa puede terminar antes de este, pero nunca después.

5.1.- Un ejemplo sencillo

Aquí esta un fragmento de programa sencillo para calcular $\Sigma^{n}_{i=1} i^3$

```
        int suma(int n)
             {
             int i, sumaParcial;
{1}          sumaParcial=0;
{2}          for(i=1; i<=n; i++)
{3}              sumaParcial=sumaParcial + i * i * i;
{4}          return sumaParcial;
             }
```

El análisis de este algoritmo es sencillo. Las declaraciones no cuentan en el tiempo. Las líneas {1} y {4} cuentan por una unidad de tiempo cada una. La línea {3} cuenta por cuatro unidades cada vez que se ejecuta (dos multiplicaciones, una adición y una asignación) y se ejecuta n veces, para un total de 4n unidades. La línea {2} tiene el costo oculto de la inicialización, n + 1 para todas las comprobaciones y n para todos los incrementos, lo cual da 2n + 2. Se ignoran los costos de llamar y retornar de la función, para un total de 6n + 4. Así, se dice que la función es de orden O(n).
Si tuviéramos que efectuar este trabajo cada vez que necesitemos analizar un programa, pronto la tarea se haría irrealizable. Por fortuna, como estamos dando la respuesta en términos de O grande, hay muchos medios de abreviar que se pueden seguir sin afectar la respuesta final. Por ejemplo, la línea {3} obviamente es un enunciado O(1) (por ejecución), así que es inútil contar precisamente si lleva dos, tres o cuatro unidades; esto no importa. La línea {1} es obviamente insignificante comparada con el ciclo for, así que es inútil consumir tiempo aquí. Esto lleva a varias reglas generales obvias.

5.2.- Reglas Generales

Regla 1 - Ciclos *FOR*
El tiempo de ejecución de un ciclo for es a lo sumo el tiempo de ejecución de las instrucciones que están en el interior del ciclo for (incluyendo las condiciones) por el número de iteraciones.

Regla 2 - Ciclos *FOR ANIDADOS*
Analizarlos de adentro hacia fuera. El tiempo de ejecución total de una proposición dentro del grupo for anidados es el tiempo de ejecución de la proposición multiplicado por el producto de los tamaños de todos los ciclos for.
Como ejemplo, el siguiente fragmento de programa es $O(n^2)$:

```
for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
                k=k + 1;
```

Regla 3 - Proposiciones consecutivas
Simplemente se suman (lo cual significa que el máximo es el único que cuenta).
Como ejemplo, el siguiente fragmento de programa, que tiene trabajo O(n) seguido de trabajo $O(n^2)$:

```
for(i=0; i<n; i++)
        a[i]=0;
for(i=0; i<n; i++)
        for(j=1; j<=n; j++)
                a[i]=a[i] + a[i] + i + j;
```

Regla 4 - *IF/ELSE*
Para el fragmento

```
if(cond)
        s1
else
        s2
```

El tiempo de ejecución de una proposición if/else nunca es más grande que el tiempo de ejecución de la condición más el mayor de los tiempos de ejecución de S1 y S2.

Claramente, esto puede ser un tiempo sobrevalorado en algunos casos, pero nunca es una subestimación.

Otras reglas son obvias, pero una estrategia básica de análisis que funciona es ir del interior (o parte más profunda) hacia fuera. Si hay llamadas a funciones, obviamente éstas deben ser analizadas primero. Si hay procedimientos recursivos, hay varias opciones. Si la recursión es un ciclo for ligeramente disfrazado, el análisis suele ser trivial. Por ejemplo, la siguiente función es en realidad solo un ciclo sencillo y obviamente es O(n):

```
int factorial(int n)
{
if(n= =0 || n= =1)
        return 1;
else
        return n * factorial(n-1);
}
```

Este ejemplo es realmente un uso ineficiente de la recursión. Cuando la recursión se usa adecuadamente, es difícil convertirla en una estructura iterativa sencilla. En este caso, el análisis implicará una relación de recurrencia que hay que resolver. Para ver que puede suceder, considérese el siguiente programa, que resulta ser un uso horrible de la recursión:

```
{Cálculo de los números de Fibonacci}
{Supóngase n >= 0 }

                int fibonacci(int n)
                    {
{1}                 if(n= =0 || n= =1)
{2}                     return n;
                    else
{3}                     return fibonacci(n-1) + fibonacci(n-2);
                    }
```

A primera vista, éste parece ser un uso muy inteligente de la recursión. Sin embargo, si el programa se codifica para valores de n alrededor de 30, se hace evidente que el programa es terriblemente ineficiente. El análisis es bastante sencillo. Sea T(n) el tiempo de ejecución de la función fibonacci(n). Si n = 0 o n = 1, entonces el tiempo de ejecución es algún valor constante, que es el tiempo requerido para evaluar la condición de la línea {1} y regresar. Se puede decir que T(0) = T(1) = 1, ya que las constantes no importan. El tiempo de ejecución para otros valores de n se mide entonces en relación con el tiempo de ejecución del caso base. Para n > 2, el tiempo de ejecución de la función es el trabajo constante de la línea {1} más el trabajo de la línea {3}. La línea {3} consta de una adición más dos llamadas a función. Puesto que las llamadas a función no son operaciones simples, se debe analizar por separado. La primera llamada a función es fibonacci(n-1) y, por lo tanto, por la definición de T, requiere T(n-1) unidades de tiempo. Un razonamiento similar muestra que la segunda llamada requiere T(n-2) unidades de tiempo. Entonces el tiempo total requerido es T(n-1) + T(n-2) + 2, donde 2 cuenta por el trabajo de la línea {1} mas la adición de la línea {3}. Así, para n ≥ 2, se tiene la siguiente fórmula del tiempo de ejecución de fibonacci(n):

$$T(n) = T(n-1) + T(n-2) + 2$$

Como fibonacci(n) es igual a fibonacci(n-1) + fibonacci(n-2), es fácil demostrar por inducción que T(n) ≥ fibonacci(n) y así el tiempo de ejecución de este programa crece *exponencialmente*. Esto es lo peor posible. Con un arreglo simple y un ciclo for, se obtiene una reducción sustancial del tiempo de ejecución. Este programa es lento porque se efectúa una cantidad enorme de trabajo redundante, violándose la cuarta regla de la recursión[2]. Nótese que la primer llamada de la línea {3}, fibonacci(n-1), realmente calcula fibonacci(n-2) en algún momento. Esta información es desechada y se compone recursivamente y hace que el tiempo de ejecución sea enorme. Este es tal vez el mejor ejemplar de la máxima “no calcular nada mas de una vez” y no debe ahuyentar al programador del uso de la recursión. Existen usos sobresalientes de la recursión.

5.3.- Soluciones al problema de la suma de la subsecuencia máxima

Ahora se presentan los cuatro algoritmos que resuelven el problema de la suma de la subsecuencia máxima antes planteado. (punto 4)

```
            int sumaSubsecuenciaMaxima(int[] a, int n)
                {
                int estaSuma, sumaMax, mejorI, mejorJ, i, j, k;
{1}             sumaMax=0; mejorI=0; mejorJ=0;
{2}             for(i=0; i<n; i++)
{3}                 for(j=0; j<n; j++)
                        {
{4}                     estaSuma=0;
```

[2] Regla del interés compuesto. El trabajo nunca se debe duplicar resolviendo el mismo ejemplar de un problema en llamadas recursivas separadas.

```
{5}                 for(k=i; k<=j; k++)
{6}                     estaSuma=estaSuma + a[k];
{7}                 if(estaSuma > sumaMax)
                        {//actualiza sumaMax, mejorI, mejorJ
{8}                     sumaMax=estaSuma;
{9}                     mejorI=i;
{10}                    mejorJ=j;
                        }
                    }
{11}            return sumaMax;
                }
```

Algoritmo 1.

Este algoritmo funciona. El tiempo de ejecución es $O(n^3)$ y se debe por completo a las líneas {5} y {6}, las cuales consisten en una proposición O(1) inmersa en tres ciclos for anidados. El ciclo de la línea {2} es de tamaño n. El segundo ciclo tiene un tamaño n - i + 1, el cual podría ser pequeño, pero también puede ser de tamaño n. Se puede suponer lo peor, con el entendimiento de que esta cota final puede ser un poco alta. El tercer ciclo tiene un tamaño j - i + 1, que de nuevo, se debe suponer de tamaño n. El total es $O(1 \cdot n \cdot n \cdot n) = O(n^3)$. La proposición {1} solo toma O(1) en total, y las proposiciones {7} a {10} toman sólo $O(n^2)$ en total, puesto que son proposiciones sencillas dentro de dos iteraciones.

Se puede evitar el tiempo de ejecución cúbico eliminando un ciclo for. Es obvio que esto no siempre es posible, pero en este caso hay una gran cantidad de cálculos innecesarios en el algoritmo. La ineficiencia que el algoritmo mejorado corrige sobre los cálculos de las líneas {5} y {6} del algoritmo 2 es indebidamente costosa. A continuación se muestra el algoritmo mejorado. Claramente este algoritmo es $O(n^2)$; el análisis es aun más sencillo.

```
        int sumaSubsecuenciaMaxima(int[] a, int n)
            {
            int estaSuma, sumaMax, mejorI, mejorJ, i, j;
{1}         sumaMax=0; mejorI=0; mejorJ=0;
{2}         for(i=0; i<n; i++)
                {
{3}             estaSuma=0;
{4}             for(j=0; j<n; j++)
                    {
{5}                 estaSuma=estaSuma + a[j];
{6}                 if(estaSuma > sumaMax)
                        {//actualiza sumaMax, mejorI, mejorJ
                        sumaMax=estaSuma;
                        mejorI=i;
                        mejorJ=j;
                        }
                    }
                }
{7}         return sumaMax;
            }
```

Algoritmo 2.

Este problema tiene solución O(n log n) recursiva y relativamente compleja, que ahora describiremos. Si no fuera porque existe una solución (lineal) O(n), este sería un excelente ejemplo del poder de la recursión. El algoritmo se vale de la estrategia “divide y vencerás”. La idea es partir el problema en dos subproblemas

más o menos iguales, cada uno de los cuales es de tamaño igual a la mitad del original. Entonces los subproblemas se resuelven recursivamente. Esta es la parte de "dividir". La etapa de "vencer" consiste en unir las dos soluciones de los subproblemas, y posiblemente en hacer un poco de trabajo adicional, para llegar a una solución del problema global.
En este caso, la suma de la subsecuencia máxima puede estar en uno de tres lugares. O esta entera en la primera mitad de la entrada, o en la segunda mitad, o bien pasa por el punto medio y se encuentra en ambas mitades. Los primeros dos casos se pueden resolver recursivamente. El último caso se obtiene, encontrando la suma mayor en la primera mitad que incluya al último elemento de esa primera mitad y la suma más grande de la segunda mitad que incluya al primer elemento de la segunda mitad. Estas dos sumas se pueden sumar. Como ejemplo, considérese la siguiente entrada:

Primera mitad				Segunda mitad			
4	-3	5	-2	-1	2	6	-2

La suma de la subsecuencia máxima de la primera mitad es 6 (los elementos entre a_1 y a_3), y para la segunda mitad es 8 (los elementos entre a_6 y a_7).
La suma máxima de la primera mitad incluye al último elemento de la primera mitad es 4 (elementos entre a_1 y a_4), y la suma máxima de la segunda mitad que incluye al primer elemento de la segunda mitad es 7 (elementos entre a_5 y a_7). Así, la suma máxima que abarca ambas mitades y pasa por el medio es 4 + 7 = 11 (elementos entre a_1 y a_7)
Se ve, entonces, que de las tres formas de encontrar la subsecuencia de longitud máxima, para el ejemplo, la mejor forma es incluir elementos de ambas mitades. Así la respuesta es 11.

El código del algoritmo 3 merece algún comentario. La forma general de la llamada al procedimiento recursivo es pasar el arreglo de entrada junto con los límites izquierdo y derecho, que delimitan la porción del arreglo de entrada sobre la que se va a operar. Un programa manejador de una línea hace esto pasando los bordes 1 y n para evitar hacer una copia. Si se olvida hacer esto el tiempo de ejecución de la rutina puede variar drásticamente, como se verá después.
Las líneas {1} a {4} manejan el caso base. Si izq == der, entonces hay un elemento, y este es la subsecuencia máxima si el elemento es no negativo. El caso izq > der no es posible a menos que n sea negativo (aunque perturbaciones menores en el código podrían echar a perder esto). Las líneas {6} y {7} hacen dos llamadas recursivas. Se puede ver que las llamadas recursivas son siempre sobre problemas más pequeños que el original, aunque, de nuevo, perturbaciones menores en el código pueden destruir esta propiedad. Las líneas {8} a la {12} y después de la {13} a la {17} calculan las dos sumas máximas que alcanzan el centro.
Claro está, el algoritmo 3 requiere más esfuerzo de codificación que cualquiera de los dos anteriores. Sin embargo, la brevedad de un código no siempre implica que este sea el mejor. Como se vio en la tabla anterior, donde se muestran los tiempos de ejecución de los algoritmos, este algoritmo es considerablemente más rápido que los otros dos, excepto con entradas de tamaño reducido.

```
        int sumaSubsecuenciaMaxima(int[] a, int n)
                {
                return sumaSubsecuenciaMaxima(a, 0, n-1);
                }

        int sumaSubsecuenciaMaxima(int[] a, int izq, int der)
                {
                int sumaMaxIzq, sumaMaxDer, sumaMaxIzqBorde, sumaMaxDerBorde;
                int sumaBordeIzq=0, sumaBordeDer=0, centro, i;
{1}             if(izq = = der) //caso base
```

```
{2}             if(a[izq] > 0)
{3}                 return a[izq];
                else
{4}                 return 0;
            else
                {
{5}             centro=(izq + der) / 2;
{6}             sumaMaxIzq= sumaSubsecuenciaMaxima(a, izq, centro);
{7}             sumaMaxDer= sumaSubsecuenciaMaxima(a, centro + 1, der);
{8}             sumaMaxIzqBorde=0;
{9}             for(i=centro; i>=izq; i--)
                    {
{10}                sumaBordeIzq=sumaBordeIzq + a[i];
{11}                if(sumaBordeIzq > sumaMaxIzqBorde)
{12}                    sumaMaxIzqBorde=sumaBordeIzq;
                    }
{13}            sumaMaxDerBorde=0;
{14}            for(i=centro+1; i<=der; i++)
                    {
{15}                sumaBordeDer=sumaBordeDer + a[i];
{16}                if(sumaBordeDer > sumaMaxDerBorde)
{17}                    sumaMaxDerBorde=sumaBordeDer;
                    }
{18}            return Math.max(Math.max(sumaMaxIzq, sumaMaxDer),
                        sumaMaxIzqBorde +  sumaMaxDerBorde);
                }
            }
```

Algoritmo 3.

El tiempo de ejecución se analiza casi en la misma forma que el programa que calcula los números de Fibonacci. Sea T(n) el tiempo que lleva resolver un problema de suma de la subsecuencia máxima de tamaño n. Si n = 1, entonces el programa usa una cantidad de tiempo constante para ejecutar la línea{1} a {4}, a la cual se tomará como la unidad. Así T(1) = 1. De otro modo, el programa debe realizar dos llamadas recursivas, los dos ciclos que están entre las líneas {5} y {18}. Los dos ciclos for se combinan para alcanzar todo elemento entre a_1 y a_n, con un trabajo constante en el interior de los ciclos, así que el tiempo consumido en las líneas {9} a {17} es O(n). El código de las líneas {1} a {5}, {8} y {18} es constante en trabajo y se puede ignorar cuando se compara con O(n). El resto del trabajo se realiza en las líneas {6} y {7}. Esas líneas resuelven dos problemas de suma de la subsecuencia máxima con tamaño n/2 (suponiendo n par). Así, estas líneas utilizan T(n/2) unidades de tiempo cada una, para un total de 2T(n/2). El tiempo total consumido por el algoritmo, por lo tanto, es 2T(n/2) + O(n). Esto da las ecuaciones

$$T(1) = 1$$
$$T(n) = 2T(n/2) + O(n)$$

Para simplificar los cálculos, se puede sustituir por n el término O(n) de a ecuación anterior; como de todos modos T(n) se expresará en notación O grande, esto no afectará la respuesta. Si T(n) = 2T(n/2) + n, y T(1) = 1, entonces T(2) = 4 = 2 * 2, T(4) = 12 = 4 * 3, T(8) = 32 = 8 * 4, T(16) = 80 = 16 * 5. El patrón que es evidente, y que se puede obtener, es que si $n = 2^k$, entonces T(n) = n * (k+1) = n log n + n = O(n log n).
Este análisis supone que n es par, ya que de otra forma n/2 no esta definido. Por la naturaleza recursiva del análisis, esto es realmente válido solo cuando n es una potencia de 2, pues de otra forma tarde o temprano se

llega a un subproblema que no es de tamaño par, y la ecuación es inválida. Cuando n no es una potencia de 2, se requiere un análisis un poco mas complejo, pero el resultado O grande permanece sin cambio.
Se calcula el número total de llamadas recursivas que efectúa el algoritmo. Sea R(n) el número de llamadas recursivas hechas para resolver un problema de subsecuencia de tamaño n. Entonces R(1) = 0, ya que éste es el caso base. El número de llamadas recursivas de la línea {6} es igual al número de llamadas hechas cuando se resuelve el primer subproblema más, por supuesto, la llamada recursiva real. La misma lógica se aplica a la línea {7}. Esta dice que R(n) = 2R(n/2). Esto se puede resolver (cuando n es una potencia de 2) para llegar a R(n) = 2n – 2. El cuarto algoritmo para encontrar la suma de la subsecuencia máxima, es más simple de implantar que el algoritmo recursivo y también es más eficiente.

```
int sumaSubsecuenciaMaxima(int[] a, int n)
        {
        int estaSuma, sumaMax, mejorI, mejorJ, i, j;
        i=0; estaSuma=0; sumaMax=0; mejorI=0; mejorJ=0;
        for(j=0; j<n; j++)
            {
            estaSuma=estaSuma + a[j];
            if(estaSuma > sumaMax)
                {//actualiza sumaMax, mejorI, mejorJ
                sumaMax=estaSuma;
                mejorI=i;
                mejorJ=j;
                }
            else
                if(estaSuma <= 0)
                    {
                    i=j+1;
                    estaSuma=0;
                    }
            }
        return sumaMax;
        }
```

Algoritmo 4.

Debe quedar claro por que la cota del tiempo es correcta, pero lleva más tiempo entender por que de hecho funciona. Esto se deja al estudiante. Una ventaja adicional del algoritmo es que solo recorre una vez los datos, y una vez que se lee y procesa a[i], no necesita recordarse esto. Así, si el arreglo esta en disco o cinta, se puede leer secuencialmente, sin necesidad de almacenar parte alguna en la memoria principal. Más aún, en cualquier momento, el algoritmo puede dar correctamente una respuesta al problema de la subsecuencia para los datos que ya ha leído (los otros algoritmos no comparten esta propiedad). Los algoritmos que pueden hacer esto se conocen como *algoritmos en línea*. Un algoritmo en línea que requiere solo espacio constante y se ejecuta en tiempo lineal es el mejor posible.

5.4.- Logaritmos en el tiempo de ejecución

Es posible que el aspecto más confuso del análisis de algoritmos se centre en el logaritmo. Ya se ha visto que algunos algoritmos “divide y vencerás” se ejecutarán en un tiempo O(n log n). Además de los algoritmos de “divide y vencerás”, la aparición más frecuente de los logaritmos esta alrededor de la siguiente regla general: *un algoritmo es O(log n) si usa un tiempo constante (O(1)) en dividir el problema en partes (normalmente en ½).* Por otro lado, si se requiere un tiempo constante simplemente para reducir el problema en una cantidad constante (como hacer el problema más pequeño en 1), el algoritmo es O(n).

Algo que debe ser obvio es que solo esa clase especial de problemas puede ser O(log n). Por ejemplo si la entrada es una lista de n números, un algoritmo solo debe tardar Ω(n) en leer los datos. Así, cuando se habla de algoritmos O(log n) para esa clase de problemas, por lo regular se supone que la entrada fue leída antes. A continuación se dan tres ejemplos de comportamiento logarítmico.

BUSQUEDA BINARIA

El primer ejemplo se conoce como búsqueda binaria. Dado un entero x y enteros a_1, a_2, ... a_n, los cuales están ordenados y en memoria, encontrar i tal que $a_i = x$, o devolver i = 0 si x no esta en la entrada.
La solución obvia consiste en rastrear la lista de izquierda a derecha y se ejecuta en tiempo lineal. No obstante, este algoritmo no aprovecha el hecho que la lista esta ordenada, y por lo tanto, probablemente no sea la mejor solución. La mejor estrategia es probar si x esta en la mitad de la lista, de ser así, la respuesta esta a la mano. Si x es menor que el elemento del centro, se puede aplicar la misma estrategia al subarreglo ordenado a la izquierda del elemento central; si x es mayor que el elemento del centro se buscará en la mitad derecha. (También esta el caso de cuando parar). A continuación se muestra el código para la búsqueda binaria (la respuesta es el contenido de medio, en este caso la variable pos).

```
        int busquedaBinaria(int[] a, int x, int n)
                {
                int primero, medio, ultimo, pos=-1;
{1}             primero=0;
{2}             ultimo=n-1;
{3}             while(primero<=ultimo & pos= = -1)
                    {
{3}                 medio=(primero + ultimo) / 2;
{5}                 if(a[medio] = = x)
{6}                     pos=medio;
                    else
{7}                     if(a[medio] < x)
{8}                         primero=medio + 1;
                        else
{9}                         ultimo=medio - 1;
                    }
{10}            return pos;
                }
```

Por supuesto todo el trabajo realizado por iteración dentro del ciclo es O(1), así que el análisis requiere determinar el número de veces que se itera. El ciclo empieza con ultimo – primero = n – 1 y termina cuando ultimo – primero ≤ 0. Cada vez que se itera el valor ultimo – primero debe reducirse al menos a la mitad de su valor original; así, el número de veces que se itera es a lo sumo [log(n-1)+2]. (Por ejemplo, si ultimo – primero = 128, entonces los valores máximos de ultimo – primero después de cada iteración son 64, 32, 16, 8, 4, 2, 1, 0. Así, el tiempo de ejecución es O(log n). En forma equivalente, se podría escribir una fórmula recursiva para el tiempo de ejecución, pero esta clase de enfoque por fuerza bruta es innecesario cuando se entiende que esta ocurriendo realmente y por qué.
La búsqueda binaria se puede ver como una estructura de datos. Permite la operación buscar en tiempo O(log n), pero todas las demás operaciones (en particular insertar) requieren tiempo O(n). En aplicaciones donde los datos son estáticos (esto es, no se permiten inserciones y eliminaciones), puede ser una estructura de datos muy útil. La entrada podría ser ordenada solo una vez, y después los accesos serían muy rápidos. Un ejemplo podría ser un programa que necesita mantener información acerca de la tabla periódica de los elementos (problema típico en física y química). Esta tabla es relativamente estable, dado que es poco frecuente que se agreguen nuevos elementos. Los nombres de los elementos se podrían almacenar

ordenados. Como solo hay unos 110 elementos, a lo sumo se requerirían ocho accesos para encontrar un elemento. Realizar la búsqueda secuencial requeriría muchos más accesos.

ALGORITMO DE EUCLIDES

Un segundo ejemplo es el algoritmo de Euclides para calcular el máximo común divisor. El máximo común divisor de dos enteros es el mayor entero que divide a ambos. Así, *maximoComunDivisor*(50,15) = 5. El siguiente algoritmo calcula *maximoComunDivisor(m, n)*, suponiendo que $m \geq n$. (Si m < n, la primera iteración del ciclo los intercambia)

```
int maximoComunDivisor(int m, int n)
    {
    int resto;
    while(n > 0)
        {
        resto=m % n;
        m=n;
        n=resto;
        }
    return m;
    }
```

El algoritmo funciona a base de calcular continuamente los restos hasta llegar a 0. La respuesta es el último distinto de cero. Así, si m = 1989 y n = 159, entonces la secuencia de restos es 399, 393, 6, 3, 0. Por lo tanto, *maximoComunDivisor* (1989,1590) = 3. Como lo muestra el ejemplo, este es un algoritmo rápido.
Como antes, el tiempo de ejecución total del algoritmo depende de lo grande que sea la secuencia de residuos. Aunque log n parece ser una buena respuesta, no es en absoluto obvio que el valor del resto tenga que descender en un factor constante, pues se ve que el resto va de 399 a solo 393 en el ejemplo. En efecto, el resto no disminuye en un factor constante en una iteración. Sin embargo, se puede demostrar que, después de dos iteraciones, el resto es a lo sumo la mitad de su valor original. Esto podría demostrar que el número de iteraciones es a lo sumo 2 log n = O(log n) y establecer el tiempo de ejecución. Esta demostración es fácil, por lo que se incluye aquí. Se infiere directamente del siguiente teorema.

Teorema: Si m > n, entonces m % n < m / 2.

Demostración: Hay dos casos. Si $n \leq m / 2$, entonces obviamente, como el resto es menor que n, el teorema se cumple para este caso. El otro caso es n > m / 2. Pero entonces n cabe en m una vez con un resto m – n < m / 2, demostrando el teorema.

Uno puede preguntarse si esta es la mejor cota posible, ya que 2 log n es cerca de 20 para este ejemplo, y solo se realizaron siete operaciones. Resulta que la constante se puede mejorar ligeramente, para llegar a 1.44 log n, en el peor caso (el cual es alcanzable si m y n son números de Fibonacci consecutivos). El rendimiento en el caso promedio del algoritmo de Euclides requiere páginas y páginas de análisis matemático altamente complicado, y resulta finalmente que el número medio de iteraciones es de cerca de $(12 \ln 2 \ln) / \pi^2 + 1.47$

EXPONENCIACION

El último ejemplo trata de la elevación de un entero a una potencia (que también es entera). Los números que resultan de la exponenciación suelen ser bastantes grandes, así que un análisis sólo sirve si se supone

que se tienen una máquina con capacidad para almacenar tales enteros grandes (o un compilador que los pueda simular). Contaremos el número de multiplicaciones como la medida del tiempo de ejecución.
El algoritmo obvio para calcular x^n usa n - 1 multiplicaciones. El algoritmo recursivo siguiente lo hace mejor. Las líneas {1} a la {4} manejan el caso base de la recursión. De otra forma, si n es par, se tienen $x^n = x^{n/2} \cdot x^{n/2}$, y si n es impar, $x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$.

```
            int potencia(int x, int n)
                {
{1}             if(n = = 0)
{2}                 return 1;
                else
{3}                 if(n = = 1)
{4}                     return x;
                    else
{5}                     if(n % 2 = = 0)
{6}                         return potencia(x * x, n / 2);
                        else
{7}                         return potencia(x * x, n / 2) * x;
                }
```

Por ejemplo, para calcular x^{62} el algoritmo hace los siguientes cálculos, en los que intervienen solo nueve multiplicaciones:

$x^3 = (x^2)\, x,\ \ x^7 = (x^3)^2\, x,\ \ x^{15} = (x^7)^2\, x,\ \ x^{31} = (x^{15})^2\, x,\ \ x^{62} = (x^{31})^2$

Claramente, el número de multiplicaciones requeridas es a lo sumo 2 log n, porque a lo sumo se necesitan dos multiplicaciones (n es impar) para partir en dos, el problema. De nuevo, se puede escribir y resolver una fórmula recursiva. La simple intuición hace obvia la necesidad de un enfoque por la fuerza bruta.
A veces es interesante ver cuánto código se puede abreviar sin afectar la corrección. En el código escrito, de hecho, {3} a {4} son innecesarias ya que si n = 1, entonces la línea {7} resuelve ese caso. También la línea {7} se puede escribirse como:

```
{7}     return potencia(x, n - 1) * x;
```

sin afectar la corrección del programa. En efecto, el programa seguirá ejecutándose en O(log n), porque la secuencia de multiplicaciones es la misma que antes. Sin embargo, todas las alternativas a la línea {6} que se encuentran a continuación no sirven, aunque parezcan correctas:

```
{6a}    return potencia(potencia(x, 2), n / 2);
{6b}    return potencia(potencia(x, n / 2), 2);
{6c}    return potencia(x, n / 2) * potencia(x, n / 2);
```

Las líneas {6a} y {6b} son incorrectas porque cuando n = 2, una de las llamadas recursivas a potencia tiene 2 como segundo argumento. Así, no se obtiene progreso, y resulta un ciclo infinito.
Usar la línea {6c} afecta la eficiencia porque hay dos llamadas recursivas de tamaño n / 2 en vez de una. Un análisis demostrará que ese tiempo de ejecución ya no será O(log n).

5.5.- Verificación del análisis

Una vez que se ha realizado un análisis, es deseable ver si la respuesta es correcta y es la mejor posible. Una forma de hacer esto es codificar el programa y ver si los tiempos de ejecución observados empíricamente concuerdan con los predichos por el análisis. Cuando n se duplica, el tiempo de ejecución crece en un factor

de 2 para los programas lineales, 4 para programas cuadráticos y 8 para programas cúbicos. Los programas que se ejecutan en tiempo logarítmicos sólo se incrementan en una cantidad constante cuando n se duplica, y los programas que se ejecutaban en O(n log n) tardan ligeramente más del doble bajo las mismas circunstancias. Estos incrementos pueden ser difíciles de determinar si los términos de orden menor tienen coeficientes relativamente grandes y n no es bastante grande. Un ejemplo es el saldo de n = 10 a n = 100 en el tiempo de ejecución de las diferentes implementaciones del problema de la suma de la subsecuencia máxima. También puede ser muy difícil diferenciar los programas lineales de los programas O(n log n) a partir de una simple comprobación empírica.
Otro truco que se usa mucho para verificar que algún programa es O(f(n)) consiste en calcular los valores de T(n) / f(n) para un intervalo de n (por lo regular espaciado en factores de 2), donde T(n) es el tiempo de ejecución empírica observado. Si f(n) es una respuesta cercana al tiempo de ejecución, entonces los valores calculados convergen a una constante positiva. Si f(n) es una sobreestimación, los valores convergen a cero. Si f(n) esta subestimada y, por lo tanto, es incorrecta, los valores divergen.
Por ejemplo, el fragmento de programa de la siguiente figura calcula la probabilidad de que dos enteros positivos distintos, menores o iguales que n y escogidos al azar, sean primos relativos. (Cuando n aumente, la respuesta se acerca a $6/\pi^2$)

```
rel=0;
tot=0;
for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
            {
            tot=tot + 1;
            if(maximoComunDivisor(i, j) = = 1)
                rel=rel + 1;
            }
System.out.println("El porcentaje de pares primos relativos es "+ (double)rel / (double)tot);
```

n	Tiempo de CPU (T)	T/n^2	T/n^3	$T/n^2 \log n$
100	022	.002200	.000022000	.0004777
200	056	.001400	.000007.000	.0002642
300	118	.001311	.000004370	.0002299
400	207	.001294	.000003234	.0002159
500	318	.001272	.000002544	.0002047
600	466	.001294	.000002157	.0002024
700	644	.001314	.000001877	.0002006
800	846	.001322	.000001652	.0001977
900	1086	.001341	.000001490	.0001971
1000	1362	.001362	.000001362	.0001972
1500	3240	.001440	.000000960	.0001969
2000	5949	.001482	.000000740	.0001947
4000	25720	.001608	.000000402	.0001938

La tabla muestra el tiempo de ejecución real observado para esta rutina en una computadora real. La última columna es más probable, y así el análisis obtenido debió ser correcto. Nótese que no hay gran diferencia entre $O(n^2)$ y $O(n^2 \log n)$, ya que los logaritmos crecen muy lentamente.

ARBOLES

1.- INTRODUCCION

Hasta el momento sólo se han estudiado estructuras de datos lineales estáticas y dinámicas: a un elemento sólo le sigue otro elemento. Al analizar la estructura árbol se introduce el concepto de estructura no-lineal y dinámica de datos más importante en computación. Dinámica, puesto que puede cambiar durante la ejecución de un programa. No lineal, puesto que a cada elemento del árbol pueden seguirle varios elementos.

2.- TERMINOLOGIA FUNDAMENTAL

Un árbol es una estructura jerárquica de una colección de objetos. Es decir, un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación (de "paternidad") que impone una estructura jerárquica sobre los nodos. Los árboles genealógicos y los organigramas son ejemplos comunes de árboles.

Formalmente, un árbol se puede definir de manera recursiva (se utiliza la recursión para definir un árbol porque es una característica inherente a los mismos) como:

1. Un solo nodo es, por sí mismo, un árbol. Ese nodo es también la raíz de dicho árbol.
2. Supóngase que n es un nodo y que A^1, A^2, ..., A^K son árboles con raíces n^1, n^2, ..., n^K, respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el padre de los nodos n^1, n^2, ..., n^K. En dicho árbol, n es la raíz y A^1, A^2, ..., A^K son los subárboles (o árboles hijos) de la raíz. Los nodos n^1, n^2, ..., n^K reciben el nombre de hijos del nodo n y el nodo n recibe el nombre de padre de dichos nodos.

Una forma particular de árbol es el árbol nulo o vacío, que es un "árbol" sin nodos.

Gráficamente la estructura árbol se puede representar de diferentes formas:
a) Diagramas de Venn.
b) Anidación de paréntesis.
c) Notación decimal de Dewey.
d) Notación identada.
e) Grafos jerárquicos, que es la que más se utiliza.

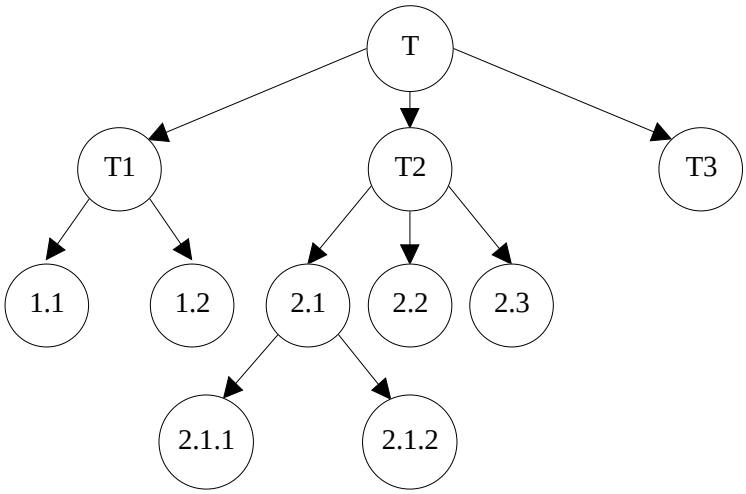
Como ejemplo se puede considerar el índice de un libro.

T1 (Tema 1)
- 1.1.- (Pregunta 1 del Tema 1)
- 1.2.- (Pregunta 2 del Tema 1)

T2 (Tema 2)
- 2.1.- (Pregunta 1 del Tema 2)
  - 2.1.1.-(Pregunta 1 de la pregunta 1 del Tema 2)
  - 2.1.2.-(Pregunta 2 de la pregunta 1 del Tema 2)
- 2.2.- (Pregunta 2 del Tema 2)
- 2.3.- (Pregunta 3 del Tema 2)

T3 (Tema 3)

Tal índice es el árbol que se muestra en la siguiente figura. La relación padre-hijo entre dos nodos se representa por una línea descendente que los une. Normalmente, los árboles se dibujan de arriba hacia abajo, con el padre encima de los hijos. En el ejemplo, la relación de paternidad representa inclusión: el libro está compuesto por los temas 1, 2 y 3.



Se llaman hermanos a los árboles hijos del mismo padre. Ejemplo: 2.1, 2.2 y 2.3 son hermanos.

Si n^1, n^2, ..., n^K, es una sucesión de nodos de un árbol tal que n^i es el padre de n^{i+1} para i = 1, 2, .., K-1, entonces a la sucesión se denomina camino del nodo n^1 al nodo n^K. Ejemplo: T2, 2.1, 2.1.1 es un camino.

La longitud de un camino es el número de nodos del camino menos 1. Por lo tanto, existen caminos de longitud cero, que son aquellos que van de cualquier nodo a sí mismo. Ejemplo: el camino T2, 2.1, 2.1.1 tiene longitud 2.

Si existe un camino de un nodo *a* a otro *b*, entonces a es un antecesor de b, y b es un descendiente de a. Cada nodo es a la vez un antecesor y un descendiente de sí mismo. En nuestro ejemplo, los antecesores de 2.1. son el mismo, T2 y Libro. Un antecesor o un descendiente de un nodo que no sea el mismo recibe el nombre de antecesor propio o descendiente propio, respectivamente. En un árbol, la raíz es el único nodo que no tiene antecesores propios. Un nodo sin descendientes propios se denomina hoja o nodo terminal.

Un subárbol de un árbol es un nodo junto con todos sus descendientes. Ejemplo:

Grado de un nodo es el número de subárboles que tiene. En nuestro ejemplo el grado de T2 es 3. Por lo tanto son nodos terminales u hojas los nodos de grado 0.

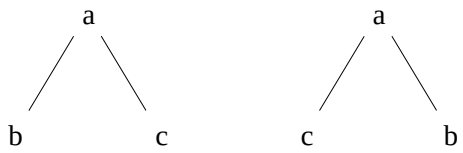
La altura de un nodo es la longitud del camino más largo de ese nodo a una hoja. En nuestro ejemplo, el nodo T1 tiene altura 1, T2 altura 2 y el nodo T3 altura 0. La altura del árbol es la altura de la raíz.

Nivel o profundidad de un nodo es la longitud del único camino desde la raíz a ese nodo. Por definición la raíz tiene nivel 0. En nuestro ejemplo, el nodo T1 tiene nivel 1, 2.1 nivel 2. La profundidad de un árbol se define como el máximo de los niveles de los nodos del árbol. En el ejemplo la profundidad del árbol es 3.

2.1.- Orden de los nodos

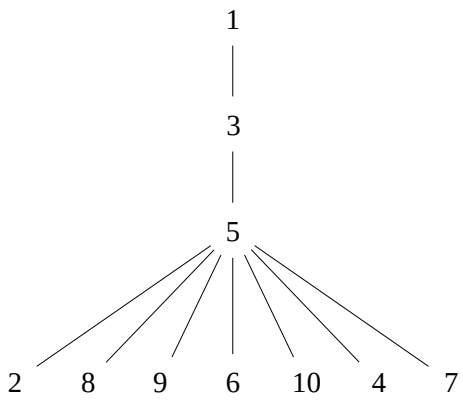
Se habla de un árbol no ordenado cuando explícitamente se ignora el orden de los hijos.

A menudo los hijos de un nodo se ordenan de izquierda a derecha. Así, los dos árboles siguientes son diferentes porque los dos hijos (**b** y **c**) del nodo **a** aparecen en distintos órdenes en los dos árboles.



El orden de izquierda a derecha de los hermanos se puede extender para comparar dos nodos cualesquiera entre los cuales no exista la relación antecesor-descendiente. La regla es que si a y b son hermanos y a está a la izquierda de b, entonces todos los descendientes de a están a la izquierda de todos los descendientes de b.

Ejemplo 1:
El nodo 8 está a la derecha del nodo 2 y a la izquierda de los nodos 9, 6, 10, 4, 7 y no está a la izquierda ni a la derecha de sus antecesores 1, 3 y 5.



Dado un nodo n, una regla sencilla para determinar qué nodos están a su izquierda y cuáles a su derecha, consiste en dibujar el camino de la raíz a n. Todos los nodos que salen a la izquierda de este camino, y todos sus descendientes, están a la izquierda de n. Los nodos, y sus descendientes, que salen a la derecha, están a la derecha de n.

2.2.- Recorridos de un árbol

Hay varias maneras de recorrer los nodos de un árbol para ser procesados con una cierta operación. Vamos a estudiar dos estrategias para recorrer un árbol y procesar sus nodos:

- Recorrido en profundidad.
- Recorrido en anchura.

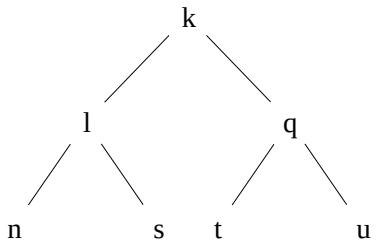
2.2.1.- Recorrido en profundidad

Existen tres formas de recorrer un árbol en profundidad:
A) Preorden u orden previo.
B) Inorden u orden simétrico.
C) Postorden u orden posterior.

Estas tres formas de ordenar un árbol se definen de forma recursiva como sigue:

1. Si el árbol A es nulo, entonces la lista vacía, es el listado de los nodos del árbol A en los órdenes preorden, inorden y postorden.

2. Si el árbol A tiene un solo nodo, entonces ese nodo constituye el listado del árbol A en los tres órdenes (preorden, inorden y postorden).
3. Si el árbol A tiene más de un nodo, es decir, tiene como raíz el nodo n y los subárboles A^1, A^2, ..., A^K entonces:



A) El listado en preorden de los nodos del árbol A está formado por la raíz del árbol A, seguida de los nodos del árbol A^1 en preorden, luego por los nodos de A^2 en preorden y así sucesivamente hasta los nodos de A^K en preorden.
El listado en preorden es: k - l - n - s - q - t - u

B) El listado en inorden de los nodos del árbol A está constituido por los nodos del árbol A^1 en inorden, seguidos de la raíz n y luego por los nodos de A^2, ..., A^K en inorden.
El listado en inorden es: n - l - s - k - t - q - u
Siempre se debe llegar al último nivel y luego se sube hacia arriba.

C) El listado en postorden de los nodos del árbol A está formado por los nodos del árbol A^1 en postorden, luego los nodos de A^2 en postorden y así sucesivamente hasta los nodos de A^K en postorden y por último la raíz n.
El listado en postorden es: n - s - l - t - u - q - k

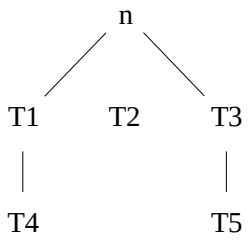
En nuestro ejemplo 1 se lista 1, y luego se llama recursivamente a Preorden del primer subárbol de 1, o sea el subárbol con raíz 2. Este subárbol tiene un solo nodo de manera que simplemente se lista. Luego se sigue con el segundo subárbol de 1, el que tiene raíz 3. Se lista 3 y luego se vuelve a llamar a Preorden con el primer subárbol de 3. Esta llamada origina que se listen 5, 8 y 9, en ese orden. Continuando de esta forma, se obtiene el recorrido en preorden: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.

En nuestro ejemplo 1 obtendremos el siguiente recorrido en postorden: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1.

En nuestro ejemplo 1 el recorrido en inorden es: 2, 1, 8, 5, 9, 3, 10, 6, 7, 4.

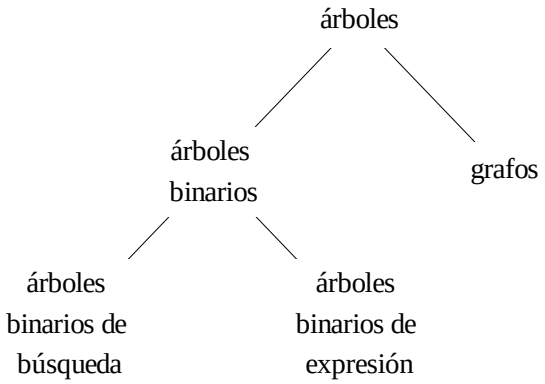
2.2.2.- Recorrido en anchura
Otro posible recorrido de un árbol es un recorrido en anchura, es decir explorando el árbol por niveles, y listando los nodos de izquierda a derecha empezando por el nivel 0, luego el nivel 1, etc.

El recorrido en anchura para el árbol: daría la secuencia de nodos: n T^1 T^2 T^3 T^4 T5



2.3.- Arboles etiquetados

Cuando se asocia una etiqueta, o valor, a cada nodo del árbol, a éste se le denomina árbol etiquetado.
La etiqueta de un nodo no es el nombre del nodo, sino que es información que está incluida en el nodo. Es posible cambiar la etiqueta del nodo sin modificar su nombre.

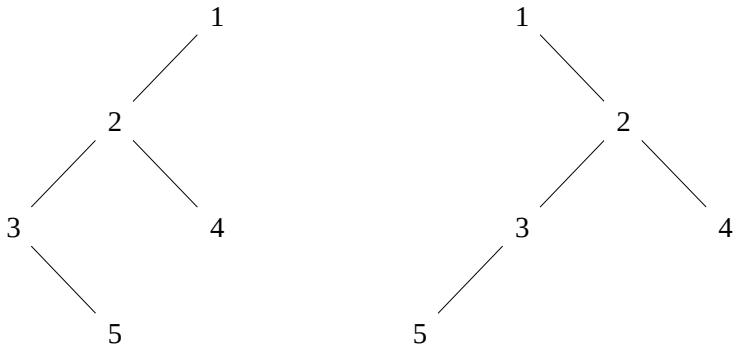


3.- ARBOLES BINARIOS

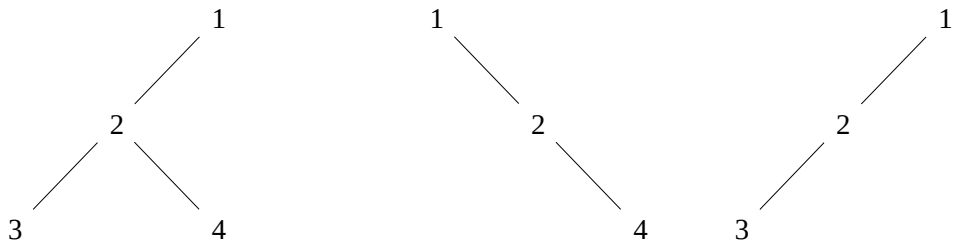
3.1.- TDA Arbol Binario. Definición

Los árboles binarios son un caso particular de árboles. Un árbol binario es un árbol donde cada nodo tiene como máximo grado 2. Es decir, un árbol binario es ó un árbol vacío, ó un árbol con un solo nodo sin hijos, ó un árbol en que sus nodos tienen un sólo hijo izquierdo, ó un árbol en que sus nodos tienen un sólo hijo derecho, ó un árbol en que sus nodos tienen un hijo izquierdo y un hijo derecho.
En los árboles, vistos anteriormente, los hijos de un nodo se encuentran ordenados de izquierda a derecha. En los árboles binarios esta ordenación es más fuerte, ya que se distingue los hijos de un nodo como izquierdo y derecho y deben estar etiquetados por ello.
Si se adopta el que los hijos izquierdos se dibujan hacia la izquierda de su padre y los hijos derechos hacia la derecha, los siguientes árboles binarios son distintos.

Ejemplo 2:
En el primer árbol, 2 es el hijo izquierdo de 1 mientras que en el segundo árbol, 2 es el hijo derecho de 1. Los árboles binarios no son directamente comparables con los árboles ordinarios, ya que en éstos no se indica si un hijo es izquierdo o derecho.



En un árbol binario, el borrar un nodo puede dar lugar a dos representaciones distintas, ya que es diferente borrar el hijo derecho o borrar el hijo izquierdo.



Los listados en preorden y postorden de un árbol binario son similares a los de un árbol general, es decir no influye si el hijo es izquierdo o derecho.

Para realizar el listado en inorden sí afecta que el hijo sea izquierdo o derecho. En el caso de que el hijo sea izquierdo, entonces primero se lista dicho hijo y luego el padre. Si el hijo es derecho, entonces primero se lista al padre y luego a este hijo derecho. El listado en inorden de los dos árboles binarios anteriores del ejemplo 2 son: 3, 5, 2, 4, 1 para el primero y 1, 5, 3, 2, 4 para el segundo.

3.2.- Implementaciones de Arboles Binarios

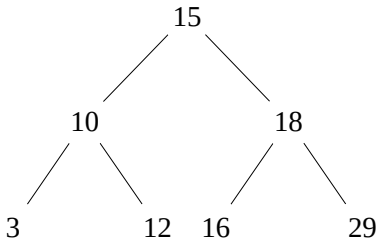
Existen dos formas tradicionales para representar un árbol binario en memoria:

- Mediante arreglos
- Mediante referencias

3.2.1.- Mediante arreglos

Hay una forma de almacenar un árbol binario en un arreglo en la que las relaciones del árbol no están representadas físicamente por campos enlaces sino que están implícitas en los algoritmos que manipulan el árbol almacenado.

Por ejemplo:
El árbol está almacenado en el arreglo por niveles, de izquierda a derecha. El árbol está almacenado con la raíz en Arbol [1] y el último nodo en Arbol [7].



15	10	18	3	12	16	29
[1]	[2]	[3]	[4]	[5]	[6]	[7]

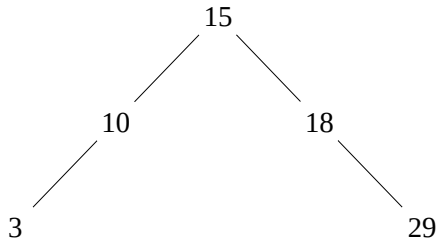
En esta implementación se cumple que para cualquier nodo Arbol[i]:

El hijo izquierdo de Arbol [i] está en Arbol [i*2].
El hijo derecho de Arbol [i] está en Arbol [i*2+1].

Para usar esta representación el árbol ha de estar completamente lleno, es decir, todos los nodos han de tener sus dos hijos. Entonces podemos escribir algoritmos para manipular el árbol de esta forma.

También podemos representar las relaciones padre-hijo mediante campos de enlace. Para ello los nombres de los nodos han de poder utilizarse como índices de un arreglo. El arreglo guarda para cada nodo sus hijos izquierdo y derecho. Ahora un árbol no es más que un índice dentro de un arreglo, que indica cual es el nodo que tiene su raíz. El acceso a los subárboles será mediante los índices que marque el nodo. El árbol vacío se representa mediante un índice imposible del arreglo.

Supongamos el siguiente árbol. Su representación podría ser la siguiente:



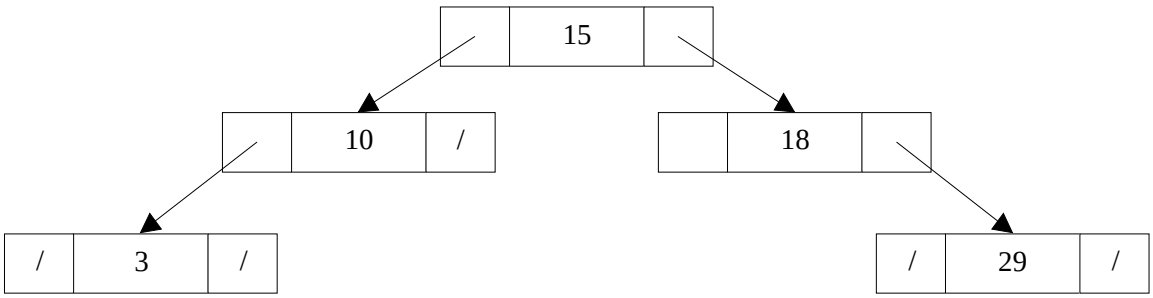
	Hijo Izquierdo	Hijo Derecho	elemento
[1]	0	18	18
[2]	4	1	15
[3]			
[4]	17	0	10
--			
--			
[17]	0	0	3
[18]	0	0	29

Para conocer el padre de cada nodo se le puede añadir a esta estructura un campo más que contenga el nombre del nodo padre del nodo i.

3.2.2.- Mediante referencias

Supone que el árbol se representa por una referencia a un nodo. En él, los hijos son de nuevo árboles, por lo que usaremos la posibilidad de definiciones de estructuras recursivas mediante variables dinámicas. Las declaraciones necesarias serían:

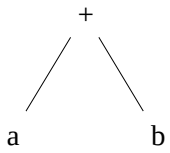
Y el árbol de la figura se representaría:



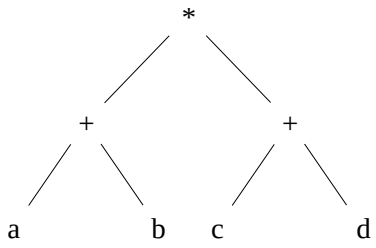
3.3.- Arboles Binarios de Expresión

Un caso particular de los árboles etiquetados lo constituyen los árboles de expresión, utilizados para la representación de expresiones aritméticas. Las reglas para representar una expresión mediante un árbol etiquetado son:
1.- Cada hoja está etiquetada con un operando y sólo consta de ese operando.
2.- Cada nodo interior está etiquetado con un operador.

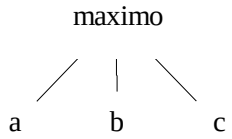
Ejemplo 1: la expresión a+b se representaría:
n, n1, n2 son los nombres de los nodos cuyas etiquetas se muestran al lado de los nodos correspondientes. La operación a realizar se pone en el nodo raíz y los operandos en los descendientes de éste.



Ejemplo 2: Si la expresión es: (a + b) * (c + d) tendremos:



Si en la operación intervienen más de dos operandos, entonces se tendrá más de dos descendientes. Por ejemplo, si la expresión es: máximo (a, b, c), el árbol será:



A menudo, cuando se recorre un árbol en preorden, inorden o postorden, se listan las etiquetas de los nodos en vez de sus nombres.

- Cuando un árbol representa una expresión, el listado en preorden de sus etiquetas da lugar a la forma prefija o polaca de la expresión, en la cual el operador precede a sus operandos izquierdo y derecho. La expresión prefija correspondiente a (E1) q (E2), donde q es un operador binario, es q P1 P2, donde P1 y P2 son las expresiones prefijas correspondientes a E1 y E2.

Se puede observar que en una expresión prefija no se necesitan paréntesis para determinar cuáles son los operandos de cada operación, porque siempre se busca la expresión más corta que se pueda obtener en preorden. Es decir, no hay dudas en la interpretación de lo que se indica.

Por ejemplo, el listado en preorden de las etiquetas de nuestro ejemplo 2 anterior es: * + a b + c d.

- El listado en postorden de las etiquetas de un árbol da lugar a la representación postfija o polaca inversa. La expresión (E1) q (E2) se representa con la expresión postfija como P1 P2 q, donde P1 y P2 son las representaciones postfijas de E1 y E2 respectivamente. Los paréntesis tampoco son necesarios, porque se puede identificar a P1 buscando la expresión más corta de P1 P2 en notación postfija.

En nuestro ejemplo 2, la expresión postfija resultante es: a b + c d + * (P1 será "a b +" porque es la expresión más corta en notación polaca).

- Si se realiza el recorrido de un árbol de expresión en inorden se obtiene la expresión infija, pero sin paréntesis.

Con nuestro ejemplo 2, el listado en inorden resultante sería: a + b * c + d. Se puede observar que es necesario los paréntesis para identificar los operandos de cada operación. Esto se puede realizar añadiendo un paréntesis abierto antes de listar la primera hoja de un subárbol y cerrando el paréntesis después de listar la última hoja del subárbol.

3.4.- Arboles Binarios de Búsqueda

El árbol binario de búsqueda es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación.

Comparando esta estructura con otra pueden observarse ciertas ventajas:

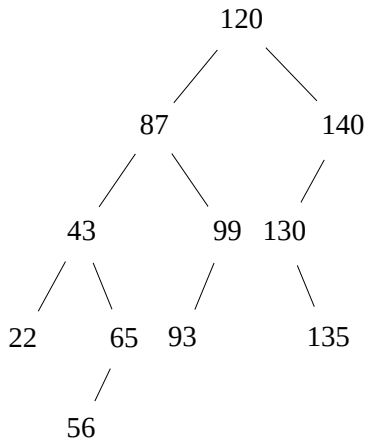
- en las matrices, si los posibles elementos son muchos, no es práctico emplear los propios elementos del árbol como índices de matrices.
- en las listas, las operaciones de inserción y eliminación se pueden llevar a cabo con facilidad, sin embargo la búsqueda es bastante costosa llevando incluso a recorrer en ocasiones todos los elementos de ella para localizar uno particular.

El árbol binario de búsqueda es una estructura de datos básica para almacenar elementos que están clasificados de acuerdo con algún orden lineal.

Se define de la siguiente manera: *Para todo nodo A del árbol debe cumplirse que todos los valores de los nodos del subárbol izquierdo de A deben ser menores al valor del nodo A. De forma similar, todos los valores de los nodos del subárbol derecho de A deben ser mayores al valor del nodo A.*

Esta condición, conocida como propiedad del árbol binario de búsqueda, se cumple para todo nodo de un árbol binario de búsqueda, incluyendo la raíz. Esta definición supone que no hay elementos duplicados.

La siguiente figura contiene un árbol binario de búsqueda, correspondiente a la entrada de datos: 120, 87, 43, 99, 140, 22, 65, 56, 93, 130 y 135.



Obsérvese la interesante propiedad de que si se listan los nodos del árbol en Orden Simétrico (InOrden), los elementos quedan clasificados en orden ascendente.

La propiedad del árbol binario de búsqueda hace que sea fácil la localización de un elemento en el árbol. Para determinar si x está en el árbol, primero se compara con el dato r que se encuentre en la raíz. Si x = r, el elemento ya está localizado. Si x < r, entonces si existe x, sólo puede estar en el subárbol izquierdo de la raíz. De igual modo, si x > r, x sólo puede estar en el subárbol derecho de la raíz. El proceso es análogo a una búsqueda binaria en un arreglo.

La inserción es una operación que se puede realizar eficientemente en un árbol binario de búsqueda. La estructura crece conforme se inserten elementos en el árbol. Los pasos que deben realizarse para insertar un elemento son los siguientes:

1. Debe compararse la clave a insertar con la raíz del árbol. Si es mayor, debe avanzarse hacia el subárbol derecho. Si es menor, debe avanzarse hacia el subárbol izquierdo.
2. Repetir sucesivamente el paso 1 hasta que se cumpla alguna de las siguientes condiciones:
   2.1 El subárbol derecho es igual a vacío, o el subárbol izquierdo es igual a vacío; en cuyo caso se procederá a insertar el elemento en el lugar que le corresponde.
   2.2 La clave que quiere insertarse es igual a la raíz del árbol; en cuyo caso no se realiza la inserción.

La operación de borrado es un poco más complicada que la de inserción. Esta consiste en eliminar un nodo del árbol sin violar los principios que definen justamente un árbol binario de búsqueda. Se deben distinguir los siguientes casos:

1. Si el elemento a borrar es terminal u hoja, simplemente se suprime.
2. Si el elemento a borrar tiene un solo descendiente, entonces tiene que sustituirse por ese descendiente.

3. Si el elemento a borrar tiene los dos descendientes, entonces se tiene que sustituir por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

3.5.- Arboles Binarios Balanceados

Como se puede ver los árboles binarios de búsqueda es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación. Sin embargo, si el árbol crece o decrece descontroladamente, el rendimiento puede disminuir considerablemente. El caso más desfavorable se produce cuando se inserta un conjunto de claves ordenadas en forma ascendente o descendente.

Con el objeto de mejorar el rendimiento en la búsqueda surgen los árboles balanceados. La idea central de éstos es la de realizar reacomodos o balanceos, después de inserciones o eliminaciones de elementos.

3.5.1.- Arboles Binarios Balanceados AVL

El nombre de AVL es en honor a sus inventores, dos matemáticos rusos, G.M. Adelson_Velskii y E.M. Landis, en el año 1962.

Formalmente se define un árbol balanceado como un árbol binario de búsqueda en el cual se debe cumplir la siguiente condición: Para todo nodo A del árbol, la altura de los subárboles izquierdo y derecho no debe diferir en más de una unidad.

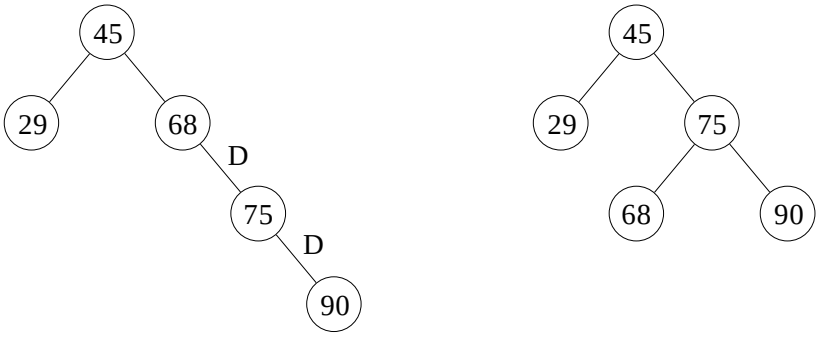
Surge el concepto llamado factor de equilibrio (FE) de un nodo como la altura del subárbol derecho menos la altura del subárbol izquierdo. Los valores que puede tomar FE son -1, 0, 1. Si FE llegara a tomar los valores -2 o 2 entonces debería reestructurarse el árbol.

La reestructuración se efectúa cuando al regresar por el camino de búsqueda después de una inserción o una supresión se comprueba que la condición del FE de algún nodo se ha violado. El proceso termina al llegar a la raíz del árbol.

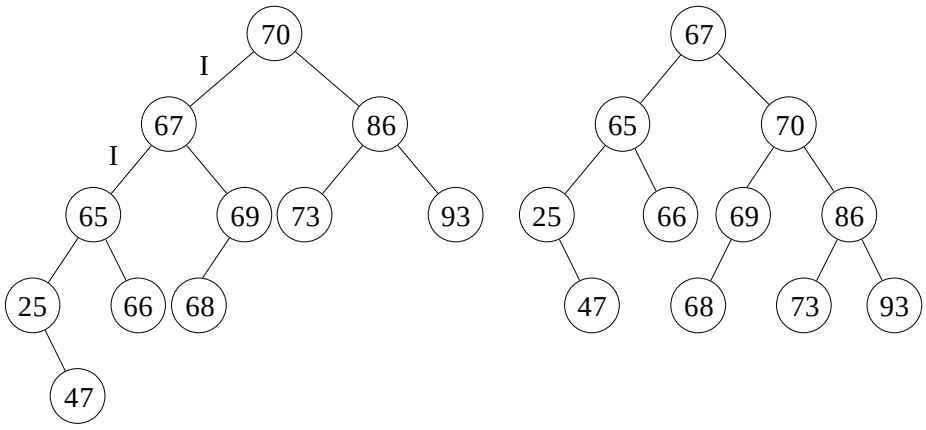
Reestructurar el árbol significa rotar los nodos del mismo. La rotación puede ser simple o compuesta. El primer caso involucra dos nodos y el segundo caso afecta a tres. Si la rotación es simple puede realizarse por las ramas derechas o por las ramas izquierdas.
Si la rotación es compuesta puede realizarse por las ramas derecha e izquierda o por las ramas izquierda y derecha.

3.5.1.1.- Inserción en Arboles Binarios Balanceados AVL

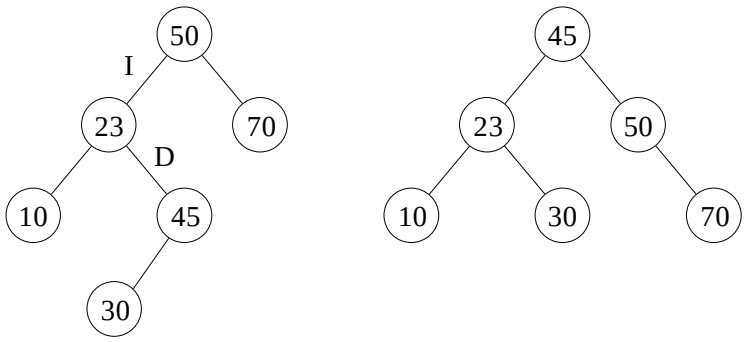
Si la rotación es DD los nodos rotan en sentido contrario a las agujas del reloj pasando el nodo central como raíz y el movimiento de referencias es el siguiente:



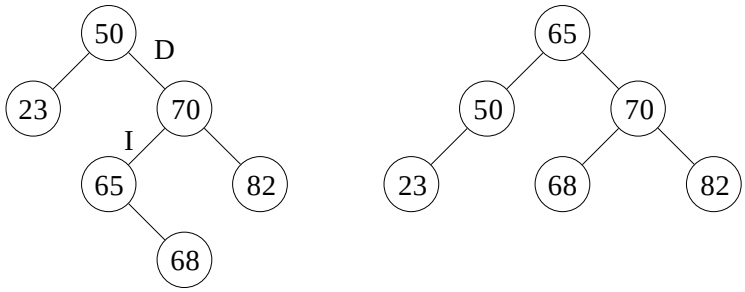
Si la rotación es II los nodos rotan en el sentido de las agujas del reloj pasando el nodo central a nodo raíz y el movimiento de referencias es el siguiente:



Si la rotación es ID el movimiento de referencias es el siguiente:



Si la rotación es DI el movimiento de referencias es el siguiente:



3.5.1.2.- Borrado en Arboles Binarios Balanceados AVL

La operación de borrado es un poco más compleja que la de inserción. Utiliza el mismo algoritmo de borrado que en los árboles binarios de búsqueda y las mismas operaciones de reacomodo que se utilizan en el algoritmo de inserción en árboles balanceados.

Para eliminar un nodo en un árbol balanceado lo primero que debe hacerse es localizar su posición en el árbol. Se elimina siguiendo los criterios establecidos para los árboles binarios de búsqueda y se regresa por el camino de búsqueda calculando el FE de los nodos visitados. Si en alguno de los nodos se viola el criterio de equilibrio, entonces debe reestructurarse el árbol. El proceso termina cuando se llega a la raíz del árbol.

Cabe aclarar que, mientras en el algoritmo de inserción una vez que era efectuada una rotación podía detenerse el proceso, en este algoritmo debe continuarse puesto que se puede producir más de una rotación en el camino hacia atrás.

3.5.1.3.- Implementación de Arboles Binarios Balanceados AVL

```
class AVLTree
        {
        AVLNode root;                   //raíz del árbol
        boolean grown, shrunk, found;   //variables artificiales para los métodos

        AVLTree()
                {//constructor para un árbol vacío
                root=null;
                grown=false;
                shrunk=false;
                found=false;
                }

        boolean search(int c)
                {//busca el contenido c en el árbol
                AVLNode n=root;
                while(n!=null)
                        {
                        if(c==n.content)
                                return true;
                        if(c<n.content)
                                n=n.left;
                        else
                                n=n.right;
                        }
                return false;
                }

        void balanceError(AVLNode n)
                {
                System.out.println("Error en el valor de equilibrio: "+ n.balance+" y contenido: "+n.content);
                }

        boolean insert(int c)
                {//inserta el contenido c en el árbol
                found=false;
                grown=true;             //por defecto el TB más bajo crecerá
                root=insert(root, c);
                return !found;
                }

        AVLNode insert(AVLNode n, int c)
                {
                if(n==null)
                        return new AVLNode(c);
                if(c==n.content)
                        {
                        found=true;
                        grown=false;
                        }
                else
                        {
                        if(c<n.content)
                                {
                                n.left=insert(n.left, c);
                                if(grown)
                                        n.balance--;            //el balance se inclina hacia la izquierda
                                }
                        else
                                {
                                n.right=insert(n.right, c);
```

```
                        if(grown)
                                n.balance++;            //el balance se inclina hacia la derecha
                        }
                switch(n.balance)
                        {
                        case -2:    {
                                    if(n.left.balance==+1)
                                            rotateLeft(n.left);
                                    rotateRight(n);
                                    grown=false;
                                    break;
                                    }
                        case -1:    break;
                        case  0:    {
                                    grown=false;
                                    break;
                                    }
                        case +1:    break;
                        case +2:    {
                                    if(n.right.balance==-1)
                                            rotateRight(n.right);
                                    rotateLeft(n);
                                    grown=false;
                                    break;
                                    }
                        default:    {
                                    balanceError(n);
                                    break;
                                    }
                        }
                }
        return n;
        }

void rotateRight(AVLNode n)
        {//rotación simple a la derecha
        AVLNode m=n.left;
        int cc=n.content;
        n.content=m.content;
        m.content=cc;
        n.left=m.left;
        m.left=m.right;
        m.right=n.right;
        n.right=m;
        int bm=1 + Math.max(-m.balance, 0) + n.balance;
        int bn=1 + m.balance + Math.max(0, bm);
        n.balance=(byte)bn;
        m.balance=(byte)bm;
        }

void rotateLeft(AVLNode n)
        {//rotación simple a la izquierda
        AVLNode m=n.right;
        int cc=n.content;
        n.content=m.content;
        m.content=cc;
        n.right=m.right;
        m.right=m.left;
        m.left=n.left;
        n.left=m;
        int bm=-(1 + Math.max(+m.balance, 0) - n.balance);
        int bn=-(1 - m.balance + Math.max(0, -bm));
        n.balance=(byte)bn;
        m.balance=(byte)bm;
        }

int height()
        {//devuelve la altura
        int h=0;
        AVLNode n=root;
```

```
		while(n!=null)
			{
			h++;
			if(n.balance>0)
				{
				h+=n.balance;
				n=n.left;
				}
			else
				{
				h-=n.balance;
				n=n.right;
				}
			}
		return h;
		}

int internal()
		{//trayectoria interna que alarga
		return internal(root, 0);
		}

int internal(AVLNode n, int h)
		{
		if(n==null)
			return 0;
		else
			{
			h++;
			return h + internal(n.left, h) + internal(n.right, h);
			}
		}

boolean delete(int c)
		{//borra o elimina el contenido c del árbol
		found=true;
		shrunk=true;
		root=delete(root, c);
		return found;
		}

AVLNode delete(AVLNode n, int c)
		{
		if(n==null)
			{
			found=false;
			shrunk=false;
			return n;
			}
		if(c==n.content)
			{
			if(n.left==null)
				return n.right;
			if(n.right==null)
				return n.left;
			n.content=c=minValue(n.right);
			}
		if(c<n.content)
			{
			n.left=delete(n.left, c);
			if(shrunk)
				n.balance++;
			}
		else
			{
			n.right=delete(n.right, c);
			if(shrunk)
				n.balance--;
			}
```

```
                switch(n.balance)
                        {
                        case -2:    {
                                    switch(n.left.balance)
                                            {
                                            case +1:    {
                                                        rotateLeft(n.left);
                                                        break;
                                                        }
                                            case  0:    {
                                                        shrunk=false;
                                                        break;
                                                        }
                                            case -1:    break;
                                            default:    {
                                                        balanceError(n.left);
                                                        break;
                                                        }
                                            }
                                    rotateRight(n);
                                    break;
                                    }
                        case -1:    {
                                    shrunk=false;
                                    break;
                                    }
                        case  0:    break;
                        case +1:    {
                                    shrunk=false;
                                    break;
                                    }
                        case +2:    {
                                    switch(n.right.balance)
                                            {
                                            case -1:    {
                                                        rotateRight(n.right);
                                                        break;
                                                        }
                                            case  0:    {
                                                        shrunk=false;
                                                        break;
                                                        }
                                            case +1:    break;
                                            default:    {
                                                        balanceError(n.right);
                                                        break;
                                                        }
                                            }
                                    rotateLeft(n);
                                    break;
                                    }
                        default:    {
                                    balanceError(n);
                                    break;
                                    }
                        }
                return n;
                }

int minValue(AVLNode n)
        {//determina el valor más pequeño en el subárbol n
        while(n.left!=null)
                n=n.left;
        return n.content;
        }
}
```

```
class AVLNode
        {
        int content;        //el contenido aquí es entero
        byte balance;       //para los valores -2, -1, 0, 1, +2
        AVLNode left;       //sucesor izquierdo
        AVLNode right;      //sucesor derecho

        AVLNode(int c)
                {//constructor para los nuevos nodos
                content=c;
                balance=0;
                left=null;
                right=null;
                }
        }
```

4.- ARBOLES MULTICAMINO

Hasta ahora el análisis se ha limitado a los árboles en que cada nodo tienen como máximo dos descendientes o hijos, es decir, a los árboles binarios. Esto resulta perfectamente apropiado si, por ejemplo, se quieren representar relaciones familiares en las que cada persona se encuentre relacionada con sus padres. Pero si la relación es a la inversa necesitamos una estructura que asocie a cada padre un número arbitrario de hijos. A estas estructuras se les llama árboles multicamino o n-arios. Este último término viene del inglés n-ary, dónde n es el grado máximo de los nodos del árbol, por ejemplo, si n es igual a 2 al árbol se le llama binario, para n igual a 3 se le llama terciario o ternario, para n igual a 4 se le llama cuaternario, y así sucesivamente.

4.1.- TDA Arbol Multicamino. Definición

Son los árboles definidos al principio del capítulo, es decir, cada nodo tienen un número arbitrario de hijos y se sigue manteniendo el orden entre los nodos.

4.2.- Implementaciones de Arboles Multicamino

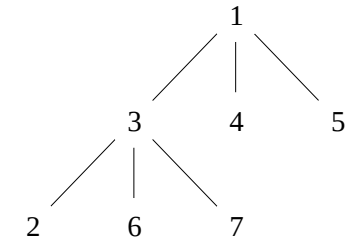
Las siguientes son formas tradicionales para representar un árbol multicamino en memoria:

- Mediante arreglos
- Mediante lista de hijos
- Basada en árboles binarios

4.2.1.- Mediante arreglos

a) Sea A un árbol cuyos nodos tienen como nombres 1, 2, ..., n. Tal vez la representación más sencilla para manejar la operación Padre, sin utilizar enlaces entre nodos, sea un arreglo unidimensional, indexada por los nombres de los nodos, y cuyo contenido A[i] sea el nombre del padre del nodo i.

Sea el siguiente árbol A: El arreglo con que representamos este árbol sería:



0	3	1	1	1	3	3
[1]	[2]	[3]	[4]	[5]	[6]	[7]

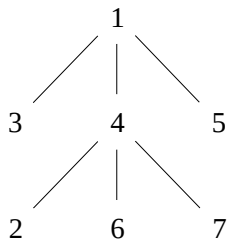
El padre del nodo raíz lo representamos con el valor 0 que es un valor nulo.

Esta representación se puede realizar porque en un árbol cada nodo tiene un padre único, por lo tanto esta implementación permite hallar el padre de un nodo en un tiempo constante. Esto da lugar a que se pueda obtener un camino ascendente en el árbol, es decir de un nodo a su padre, de éste a su padre y así sucesivamente, en un tiempo proporcional al número de nodos del camino.

Esta representación no facilita las operaciones que requieren información de los hijos. Dado un nodo n, resulta difícil determinar sus hijos o su altura. Además, esta representación no especifica el orden de los hijos de un nodo. Se puede imponer un orden artificial, por ejemplo, numerando los hijos de cada nodo después de numerar al padre, y numerando los hijos en orden ascendente, de izquierda a derecha.

b) Otra realización sería aquella en la que cada nodo se compone de la raíz y un arreglo de árboles con tantas componentes como el máximo de hijos de los nodos del árbol.

Supongamos el siguiente árbol: Su representación sería:



elemento	Hijos						
1	3	4	5				
4	2	6	7				

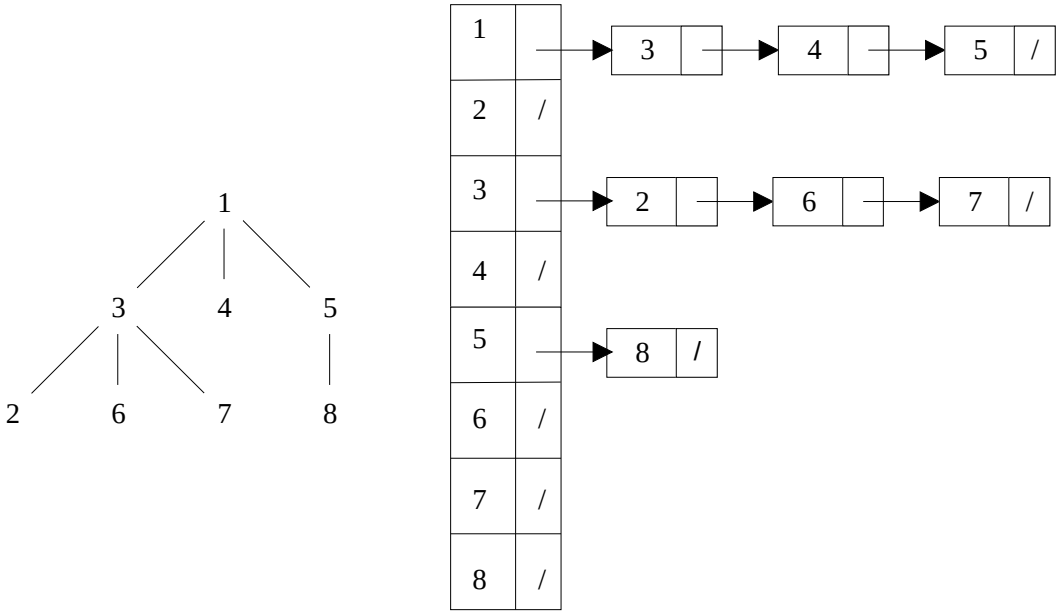
Esta implementación requiere una estimación del número de hijos y si fuese muy variable desperdiciaría mucha memoria.

4.2.2.- Mediante listas de hijos

c) Otra manera importante y útil de representar árboles consiste en formar una lista de los hijos de cada nodo. Las listas se pueden implementar con cualquiera de los métodos vistos, pero como el número de hijos que cada nodo puede tener es variable, es más apropiada la implementación de éstas mediante referencias.

En definitiva, lo que se va a tener, es un arreglo indexado por los nombres de los nodos, y donde el contenido de cada casilla va a ser la etiqueta del nodo y una lista enlazada de árboles hijos de izquierda a derecha.

Supongamos el árbol A: Su representación mediante listas de hijos es la siguiente:



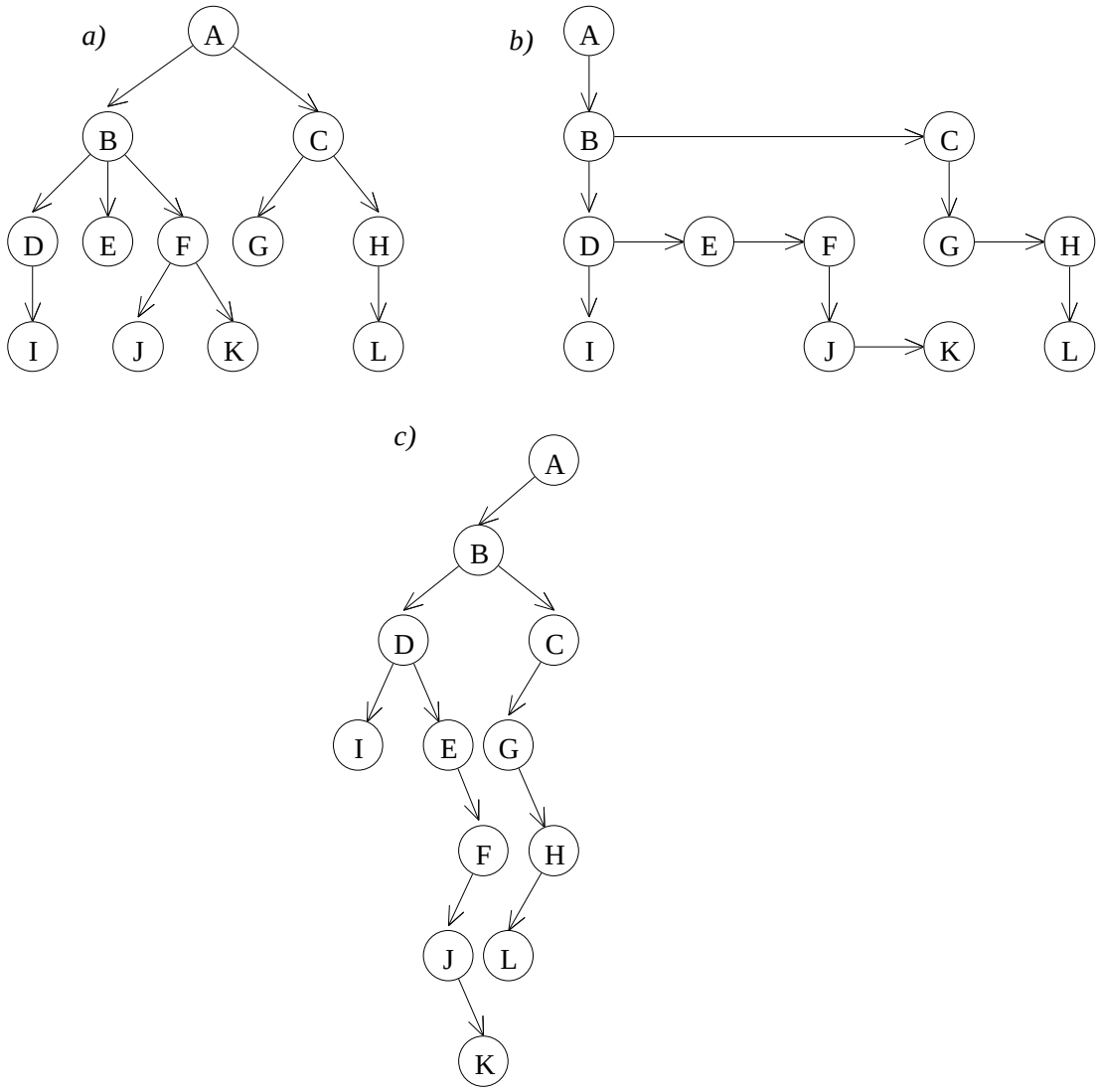
Al tener el arreglo tantas casillas como nodos tiene el árbol, se produce un desperdicio de espacio ya que los nodos hojas no tienen hijos y por lo tanto sus casillas correspondientes, dentro de este arreglo, no se usan. Pero este problema no se puede solucionar ya que en principio no se sabe qué nodos son interiores y cuáles hojas. Pero se puede observar que con esta implementación es fácil conocer el número de hijos de cada nodo.
Para conocer el padre de un nodo no resulta tan fácil ya que es necesario recorrer todas las listas hasta encontrar dicho nodo, en cuyo caso su padre será el nodo correspondiente a esa casilla de encabezamiento, suponiendo que hay un sólo árbol; en otro caso la búsqueda tendría que hacerse recursivamente. En resumen, el recorrido en profundidad (hacia arriba) se complica.

d) Otra alternativa, es utilizar una lista de listas. Hacerlo totalmente dinámico. En vez de representar los nodos de origen en un vector, se crea un nodo de una lista por cada uno, siempre que tenga hijos. De esta manera, no se desperdicia memoria en aquellos casos donde un nodo no posee hijos.

4.2.3.- Basada en Arboles Binarios
Los árboles generales quedan representados por árboles binarios de manera que el hijo izquierdo es el hijo más a la izquierda y el hijo derecho es el siguiente hermano hacia la derecha.
La principal ventaja de esta representación es su simplicidad. De hecho, esta es la representación más usada para árboles generales.

Aquí se establecerán los mecanismos necesarios para convertir un árbol general en un árbol binario. Los pasos que se deben aplicar para lograr la conversión del árbol general al árbol binario son los siguientes:
1. Deben enlazarse los hijos de cada nodo en forma horizontal (los hermanos).
2. Debe enlazarse en forma vertical el nodo padre con el hijo que se encuentra más a la izquierda. Además, debe eliminarse el vínculo de ese padre con el resto de sus hijos.
3. Debe rotarse el diagrama resultante, aproximadamente 45 grados hacia la izquierda, y así se obtendrá el árbol binario correspondiente.

Ejemplo:



Para todo nodo de un árbol binario, obtenido a partir de un árbol general, debe cumplirse lo siguiente:

1. En la rama derecha de cada nodo, excepto el nodo raíz, si ésta es distinta del vacío se encuentra un nodo que era hermano de éste en el árbol general.
2. En la rama izquierda de cada nodo, si ésta es distinta del vacío, se encuentra un nodo que era hijo de éste en el árbol general.

4.3.- Arboles Multicamino B

Un área muy práctica de aplicación de árboles multicamino es la construcción y mantenimiento de árboles de búsqueda a gran escala, donde es preciso realizar inserciones y supresiones de elementos y en situaciones en que la memoria principal es demasiado costosa o no suficientemente grande como para ser utilizada como almacenamiento permanente.

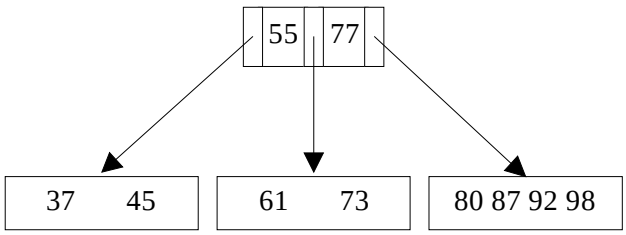
Supóngase que el volumen de información de un árbol es tal que no cabe en memoria principal y debe ser guardado en un dispositivo de almacenamiento secundario, como por ejemplo, un disco. Si se utilizase un

árbol binario como estructura para almacenar los elementos, la búsqueda de un elemento requerirá muchos pasos (en el peor de los casos, tantos como altura tenga el árbol), cada uno de los cuales necesita un acceso al disco. Una forma de reducir el número de accesos sería almacenar los elementos en bloques, llamados páginas, de manera que el acceso a una página representa ahora un acceso al disco, ahorrando así un tiempo considerable. Este es el objetivo de los árboles multicamino. Una página es un nodo del tipo $p_0(k_1p_1)(k_2p_2)...(k_np_n)$ donde k_i son claves únicas tales que $k_i < k_{i+1}$ y p_i son referencias a subárboles. Todas las claves del subárbol p_i son mayores que k_i y menores que k_{i+1}.

Debemos tener presente que un acceso a disco es extremadamente lento si lo comparamos con un acceso a memoria; es por lo tanto de primordial interés la reducción del número de accesos a disco. Si se accede a un elemento que está almacenado en memoria secundaria, también se puede acceder a un grupo completo de elementos sin que sea preciso mucho esfuerzo adicional. Esto es, cuesta lo mismo leer 4 bytes que 512 bytes, pues la unidad mínima de transferencia (bloque) suele ser de 512 bytes. Esto sugiere agrupar la información del árbol en bloques o páginas.

Un ejemplo de árbol multicamino en donde además se controla el crecimiento desequilibrado es el árbol-B. Por lo tanto, su principal característica es su buen rendimiento en cuanto a operaciones de búsqueda, aunque veremos que esto supone un coste en las operaciones de inserción y borrado.

Los árboles-B fueron propuestos por Bayer y McCreight, en el año 1970. Propusieron que:
1. todas las páginas, excepto una (la raíz), contengan entre n y 2n claves, siendo n una constante dada que llamamos aridad (orden del árbol). Con esto se garantiza que cada página esté llena como mínimo hasta la mitad.
2. Respecto al número de descendientes, cada página de un árbol-B de orden n tiene 2n+1 hijos como máximo y n+1 hijos como mínimo, excepto la página raíz que puede tener como mínimo un nodo y por consiguiente solamente dos hijos.
3. Las páginas hojas están todas al mismo nivel y el camino de la raíz a las hojas tiene la misma longitud.
4. Dentro de cada hoja las claves se encuentran ordenadas en forma creciente.



4.3.1.- Búsqueda en Arboles Multicamino B

Esta organización supone una extensión natural de los árboles binarios de búsqueda. El proceso de búsqueda en árboles-B es una generalización del proceso de búsqueda en árboles binarios de búsqueda.

1. Debe tenerse en memoria principal la página sobre la cual vamos a buscar.

Considérese la página de la figura, en la cual cada k_i representa una clave y cada p_i un apuntador, y el elemento a buscar x. Si m es suficientemente grande, se puede utilizar la búsqueda binaria. Si m es bastante pequeña, una búsqueda secuencial será suficiente.

2. Si la búsqueda es infructuosa se estará en una de las siguientes situaciones:
   2.1. $k < x < k_{i+1}$ para $1 <= i < m$. La búsqueda continúa en la página p_i.
   2.2. $k_m < x$. La búsqueda continúa en la página p_m.
   2.3. $x < k_1$. La búsqueda continúa en la página p_0.

Si en algún caso la referencias es null, es decir, si no hay página descendiente, entonces no hay ningún elemento x en todo el árbol y se acaba la búsqueda.

4.3.2.- Inserción en Arboles Multicamino B

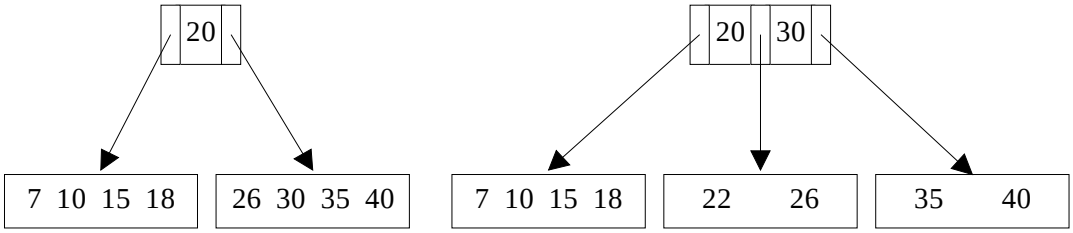
Los árboles-B tienen un comportamiento particular, diferente al resto de los árboles estudiados anteriormente. Todas las hojas están al mismo nivel y por lo tanto cualquier camino desde la raíz hasta alguna de las hojas tiene la misma longitud. Por otra parte, los árboles-B crecen de abajo hacia arriba, desde las hojas hasta la raíz.

Para insertar un elemento en un árbol-B, primero debe localizarse la página donde corresponde insertar el elemento. Una vez encontrada, pueden darse dos casos: que la página tenga espacio para más elementos, o que esté llena.

Si el número de elementos de la página es menor a 2n (m<2n), el elemento se inserta en el lugar que le corresponde.

Si el número de elementos de la página es igual a 2n (página llena), el proceso tendrá consecuencias directas en la estructura del árbol. La página afectada se divide en 2, distribuyéndose los m+1 elementos equitativamente entre las dos páginas. El elemento del medio sube a la página antecesora. Si la página antecesora se desborda nuevamente, entonces se tendrá que repetir el proceso correspondiente al caso anterior. El proceso de propagación puede llegar incluso hasta la raíz, en dicho caso la altura del árbol puede incrementarse en una unidad.

Inserción de la clave 22 en un árbol-B



4.3.3.- Borrado en Arboles Multicamino B

Consiste en quitar un elemento del árbol sin que se violen los principios que definen a un árbol-B. Se distinguen dos casos:

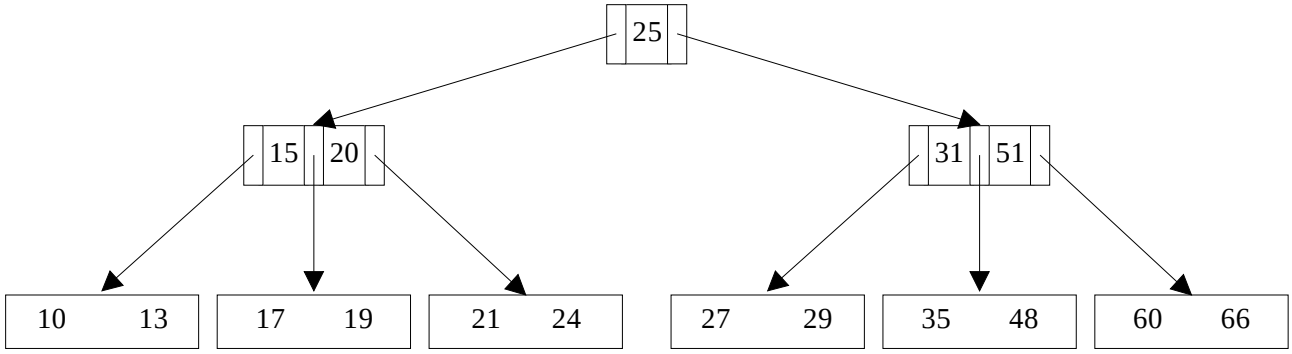
1. El elemento a borrar se encuentra en una página hoja, entonces simplemente se suprime.
2. La clave a borrar no se encuentra en una página hoja, entonces debe sustituirse por la clave que se encuentra más a la izquierda en el subárbol derecho o por la clave que se encuentra más a la derecha en el subárbol izquierdo.

En cualquiera de los dos casos verificamos m:

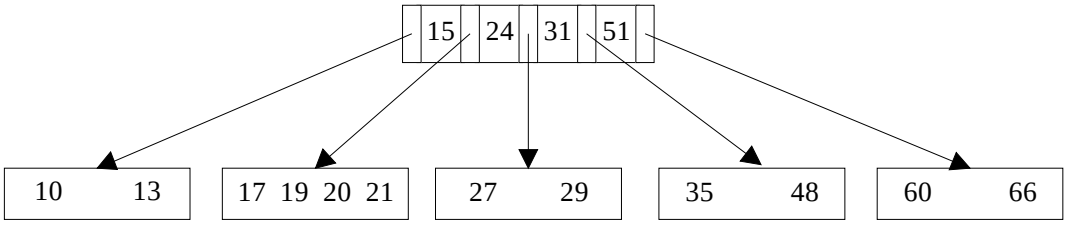
- Si m>=n entonces termina la operación de borrado.
- Si no, debe bajarse la clave lexicográficamente adyacente de la página antecesora y fusionar las páginas que son descendientes directas de dicha clave.

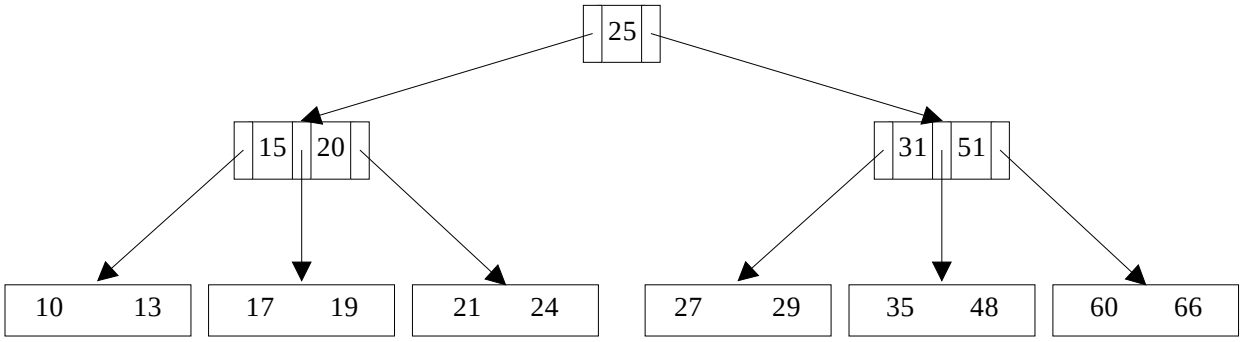
Cabe aclarar que el proceso de fusión de páginas puede propagarse incluso hasta la raíz, en cuyo caso la altura del árbol disminuye en una unidad.

Ejemplos:

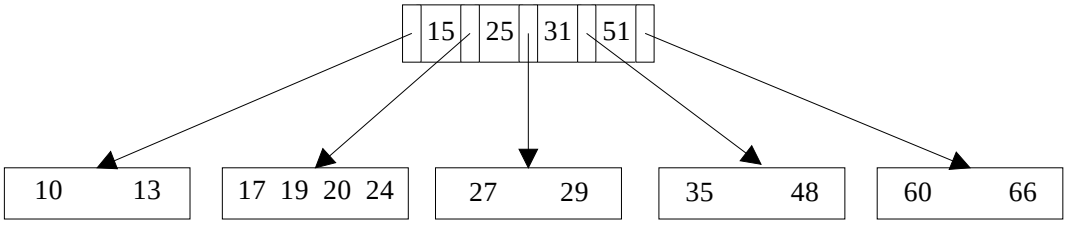


Eliminación de la clave 25





Eliminación de la clave 21



4.3.4.- Implementación de Arboles Multicamino B

```
import java.util.*;
class BTree
	{
	BNode root;
	int arity;

	BTree(int arity)
		{
		this.arity=arity;
		root=null;
		}

	void addKey(Object key)
		{
		if(root==null)
			{
			BNode newNode=new BNode(arity + 1);
			Keys keys=new Keys(arity);
			int aux=keys.size;
			keys.addKey(key);
			newNode.setData(keys);
			root=newNode;
			return;
			}
		//find a leaf to insert key. The leaf will be referenced by currentNode
		BNode currentNode=root;
		Keys currentKeys=root.getData();
		int index;
		Stack nodes=new Stack();
		while(currentNode.degree()>0)
			{
			index=currentKeys.search(key);
			if(index==-1)
				return;
			nodes.push(currentNode);
			currentNode=currentNode.getChild(index);
			currentKeys=currentNode.getData();
			}
		index=currentKeys.search(key);
		if(index==-1)
			return;
		//insert key into currentNode
		int aux=currentKeys.size;
		currentKeys.addKey(key);
		//if a node contains arity number of keys, split the node to two nodes.
		//Repeat the step for its parent and other ancestors if necessary
		while (currentKeys.size==arity)
			{
			int mid=(arity + 1) / 2 - 1;
			key=currentKeys.keyAt(mid);
			currentKeys.deleteKey(key);
			BNode newNode=new BNode(arity + 1);
			Keys newKeys=new Keys(arity);
			newNode.setData(newKeys);
			//move keys from currentNode to newNode
			for(int i=0; i<(arity + 1) / 2 - 1; i++)
				{
				Object tempKey=currentKeys.keyAt(0);
				newKeys.addKey(tempKey);
				currentKeys.deleteKey(tempKey);
				}
			//move children
			if(currentNode.degree()>0)
				for(int i=0; i<=(arity + 1) / 2 - 1; i++)
					{
					BNode child=currentNode.getChild(0);
					newNode.addChild(child,i);
```

```
                                        currentNode.deleteChild(0);
                                        }
                        if(currentNode==root)
                                {
                                root=new BNode(arity + 1);
                                Keys tempKeys=new Keys(arity);
                                tempKeys.addKey(key);
                                root.setData(tempKeys);
                                root.addChild(newNode,0);
                                root.addChild(currentNode,1);
                                return;
                                }
                        //add key and newNode into parent
                        currentNode=(BNode)nodes.peek();
                        nodes.pop();
                        currentKeys=currentNode.getData();
                        index=currentKeys.addKey(key);
                        currentNode.addChild(newNode,index);
                        }
                }

boolean searchKey(Object key)
                {
                if(root==null)
                        return false;
                BNode currentNode=root;
                Keys currentKeys=root.getData();
                int index=currentKeys.search(key);
                while(currentNode.degree()>0)
                        {
                        if(index==-1)
                                return true;
                        currentNode=currentNode.getChild(index);
                        currentKeys=currentNode.getData();
                        index=currentKeys.search(key);
                        }
                if(index!=-1)
                        return false;
                else
                        return true;
                }

boolean deleteKey(Object key)
                {
                if(root==null)
                        return false;
                BNode currentNode=root;
                Keys currentKeys=root.getData();
                int index=currentKeys.search(key);
                Stack nodes=new Stack();
                //searches for key in the BTree
                while(currentNode.degree()>0)
                        {
                        if(index==-1)
                                return true;
                        nodes.push(currentNode);
                        currentNode=currentNode.getChild(index);
                        currentKeys=currentNode.getData();
                        index=currentKeys.search(key);
                        }
                if(index==-1 && currentNode.degree()>0)
                        {//key is in a non-leaf node
                        currentKeys.deleteKey(key);
                        index=currentKeys.search(key);
                        nodes.push(currentNode);
                        Keys tempKeys;
                        if(index<currentNode.degree() - 1)
                                {//finds inorder successor
                                currentNode=currentNode.getChild(index + 1);
```

```
                        while(currentNode.degree()>0)
                                {
                                nodes.push(currentNode);
                                currentNode.getChild(0);
                                }
                        tempKeys=currentNode.getData();
                        key=tempKeys.keyAt(0);
                        }
                else
                        {//finds inorder predecessor
                        currentNode=currentNode.getChild(index);
                        while(currentNode.degree()>0)
                                {
                                nodes.push(currentNode);
                                currentNode=currentNode.getChild(currentNode.degree() - 1);
                                }
                        tempKeys=currentNode.getData();
                        key=tempKeys.keyAt(tempKeys.size - 1);
                        }
                currentKeys.addKey(key);
                tempKeys.deleteKey(key);
                currentKeys=tempKeys;
                }
else
                {
                if(index==-1)
                        currentKeys.deleteKey(key);
                else
                        return false; //key is not in BTree
                }
while(currentNode!=root && currentKeys.size<(arity + 1) / 2 - 1)
                {
                BNode parent=(BNode)nodes.peek();
                nodes.pop();
                Keys parentKeys=parent.getData();
                //find the index of currentNode in the children array of parentNode
                for(int i=0; i<parent.degree(); i++)
                        {
                        if(currentNode==parent.getChild(i))
                                {
                                index=i;
                                break;
                                }
                        }
                if(index>0)
                        {// currentNode has left sibling
                        BNode leftSibling=parent.getChild(index - 1);
                        Keys leftKeys=leftSibling.getData();
                        Object tempKey=parentKeys.keyAt(index - 1);
                        parentKeys.deleteKey(tempKey);
                        if(leftKeys.size>=(arity + 1) / 2)
                                {//right rotation
                                Object movedKey=leftKeys.keyAt(leftKeys.size - 1);
                                leftKeys.deleteKey(movedKey);
                                parentKeys.addKey(movedKey);
                                currentKeys.addKey(tempKey);
                                if(currentNode.degree()>0)
                                        {
                                        BNode movedNode=leftSibling.getChild(leftSibling.degree() - 1);
                                        leftSibling.removeChild(movedNode);
                                        currentNode.addChild(movedNode,0);
                                        }
                                return true;
                                }
                        else
                                {//node merge
                                leftKeys.addKey(tempKey);
                                for(int i=0; i<currentKeys.size; i++)
                                        {
                                        leftKeys.addKey(currentKeys.keyAt(i));
```

```
                                        if(currentNode.degree()>0)
                                                {
                                                BNode tempNode=currentNode.getChild(i);
                                                leftSibling.addChild(tempNode,leftSibling.degree());
                                                }
                                        }
                                if(currentNode.degree()>0)
                                     leftSibling.addChild(currentNode.getChild(currentNode.degree() - 1));
                                parent.deleteChild(index);
                                currentNode=parent;
                                currentKeys=parentKeys;
                                continue;
                                }
                        }
                else
                        {//currentNode has right sibling
                        BNode rightSibling=parent.getChild(index - 1);
                        Keys rightKeys=rightSibling.getData();
                        Object tempKey=parentKeys.keyAt(index - 1);
                        parentKeys.deleteKey(tempKey);
                        if(rightKeys.size>=(arity + 1) / 2)
                                {//left rotation
                                Object movedKey=rightKeys.keyAt(0);
                                rightKeys.deleteKey(movedKey);
                                parentKeys.addKey(movedKey);
                                currentKeys.addKey(tempKey);
                                if(currentNode.degree()>0)
                                        {
                                        BNode movedNode=rightSibling.getChild(0);
                                        rightSibling.removeChild(movedNode);
                                        currentNode.addChild(movedNode,currentNode.degree());
                                        }
                                return true;
                                }
                        else
                                {
                                currentKeys.addKey(tempKey);
                                for(int i=0; i<currentKeys.size; i++)
                                        {
                                        currentKeys.addKey(rightKeys.keyAt(i));
                                        if(currentNode.degree()>0)
                                                {
                                                BNode tempNode=rightSibling.getChild(i);
                                                currentNode.addChild(tempNode);
                                                }
                                        }
                                if(currentNode.degree()>0)
                                     currentNode.addChild(rightSibling.getChild(rightSibling.degree() - 1));
                                parent.deleteChild(index + 1);
                                currentNode=parent;
                                currentKeys=parentKeys;
                                continue;
                                }
                        }
                }//while
        if(currentNode==root)
                if(root.getData().size==0)
                        if(root.degree()>0)
                                root=root.getChild(0);
                        else
                                root=null;
        return true;
        }
}

class BNode
        {
        Keys data;
        BNode[] children;
        int arity, size;
```

```
BNode(int arity)
        {
        this.arity=arity;
        size=0;
        children=new BNode[arity];
        }

BNode(int arity, Keys data)
        {
        this.arity=arity;
        size=0;
        children=new BNode[arity];
        this.data=data;
        }

void setData(Keys data)
        {
        this.data=data;
        }

Keys getData()
        {
        return data;
        }

BNode getChild(int index)
        {
        if(index>=size || index<0)
                return null;
        return (BNode)children[index];
        }

BNode addChild(BNode data)
        {
        if(size==arity)
                return null;
        else
                {
                Keys temp=data.getData();
                BNode tempNode=new BNode(arity, temp);
                children[size++]=tempNode;
                return tempNode;
                }
        }

BNode addChild(BNode data, int index)
        {
        Keys temp=data.getData();
        if(index<0 || index>=size)
                return addChild(data);
        for(int i=size; i>index; i--)
                children[i]=children[i - 1];
        BNode tempNode=new BNode(arity, temp);
        children[index]=tempNode;
        size++;
        return tempNode;
        }

BNode deleteChild(int index)
        {
        if(index<0 || index>=size)
                return null;
        BNode tempNode=(BNode)children[index];
        for(int i=index; i<size - 1; i++)
                children[i]=children[i + 1];
        size--;
        return tempNode;
        }
```

```
        void removeChild(BNode data)
                {
                if(size>0)
                        {
                        int j;
                        for(int i=0; i<size; i++)
                                if((BNode)children[i]==data)
                                        deleteChild(i);
                        }
                }

        int degree()
                {
                return size;
                }
        }
```

class Keys

```
        {
        int arity, size;
        Object[] keys;

        /*Keys()
                {
                this(3);
                }*/

        Keys(int arity)
                {
                this.arity=arity;
                size=0;
                keys=new Object[arity];
                }

        Object keyAt(int index)
                {
                if(index<0 || index>=size)
                        return null;
                return keys[index];
                }

        int search(Object key)
                {
                for(int i=0; i<size; i++)
                        {
                        int c=((String)key).compareTo((String)keys[i]);
                        if(c<0)
                                return i;
                        if(c==0)
                                return -1;
                        }
                return size;
                }

        int addKey(Object key)
                {
                int index=search(key);
                if(index==-1)
                        return -1;
                for(int k=size; k>index; k--)
                        keys[k]=keys[k - 1];
                keys[index]=key;
                size++;
                return index;
                }

        boolean deleteKey(Object key)
                {
                int index=-1;
```

```
                for(int i=0; i<size; i++)
                        {
                        int c=((String)key).compareTo((String)keys[i]);
                        if(c<0)
                                return false;
                        if(c==0)
                                {
                                index=i;
                                break;
                                }
                        }
                if(index==-1)
                        return false;
                for(int j=index + 1; j<size; j++)
                        keys[j - 1]=keys[j];
                size--;
                return true;
                }
        }

import java.util.*;
class Transversal implements Enumeration
        {
        private Vector nodes;

        public Transversal(BTree AB)
                {
                nodes=new Vector();
                if(AB.root!=null)
                        nodes.addElement(AB.root);
                }

        public boolean hasMoreElements()
                {
                return (nodes.size()!=0);
                }

        public Object nextElement()
                {
                BNode tempNode=(BNode)nodes.elementAt(0);
                nodes.removeElementAt(0);
                for(int i=0; i<tempNode.degree(); i++)
                        nodes.addElement(tempNode.getChild(i));
                return tempNode;
                }
        }
```

4.4.- Arboles Multicamino B^+

Los árboles-B^+ se han convertido en la técnica más utilizada para la organización de archivos indexados. La principal característica de estos árboles es que todas las claves se encuentran en las hojas (a diferencia de los árboles-B, en que las claves podían estar en las páginas intermedias) y por lo tanto cualquier camino desde la raíz hasta alguna de las claves tienen la misma longitud.

Formalmente se define un árbol-B^+ de la siguiente manera:

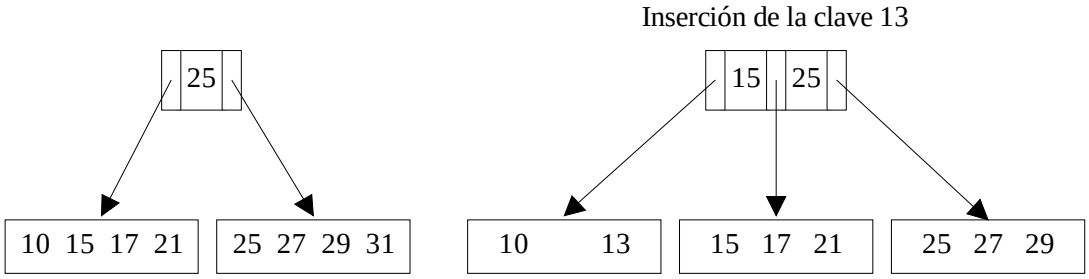
1. Cada página, excepto la raíz , contiene entre n y 2n elementos.
2. Cada página, excepto la raíz, tiene entre n+1 y 2n+1 descendientes. Se utiliza m para expresar el número de elementos por página.
3. La página raíz tiene al menos dos descendientes.
4. Las páginas hojas están todas al mismo nivel.
5. Todas las claves se encuentran en las páginas hojas.
6. Las claves de las páginas raíz e interiores se utilizan como índices.

4.4.1.- Búsqueda en Arboles Multicamino B+

La operación de búsqueda es similar a la operación de búsqueda en árboles-B. Sin embargo puede suceder que al buscar un elemento éste se encuentre en una página raíz o interior, en cuyo caso debe continuarse la búsqueda por la rama derecha de dicha clave.

4.4.2.- Inserción en Arboles Multicamino B+

La dificultad en el proceso de inserción se presenta cuando se desea insertar una clave en una página que se encuentra llena (m=2n). En este caso, la página afectada se divide en 2, distribuyéndose las m+1 claves de la siguiente forma: las n primeras claves en la página de la izquierda y las n+1 restantes claves en la página de la derecha. Una copia de la clave del medio sube a la página antecesora. Puede suceder que la página antecesora se desborde nuevamente, entonces tendrá que repetirse el proceso anterior.



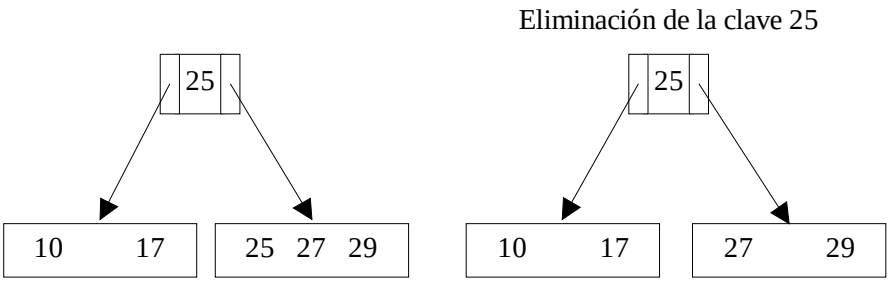
Es importante notar que el desbordamiento en una página que no es hoja no produce duplicidad de claves. El proceso de propagación puede llegar hasta la raíz, en cuyo caso la altura del árbol puede incrementarse en una unidad.

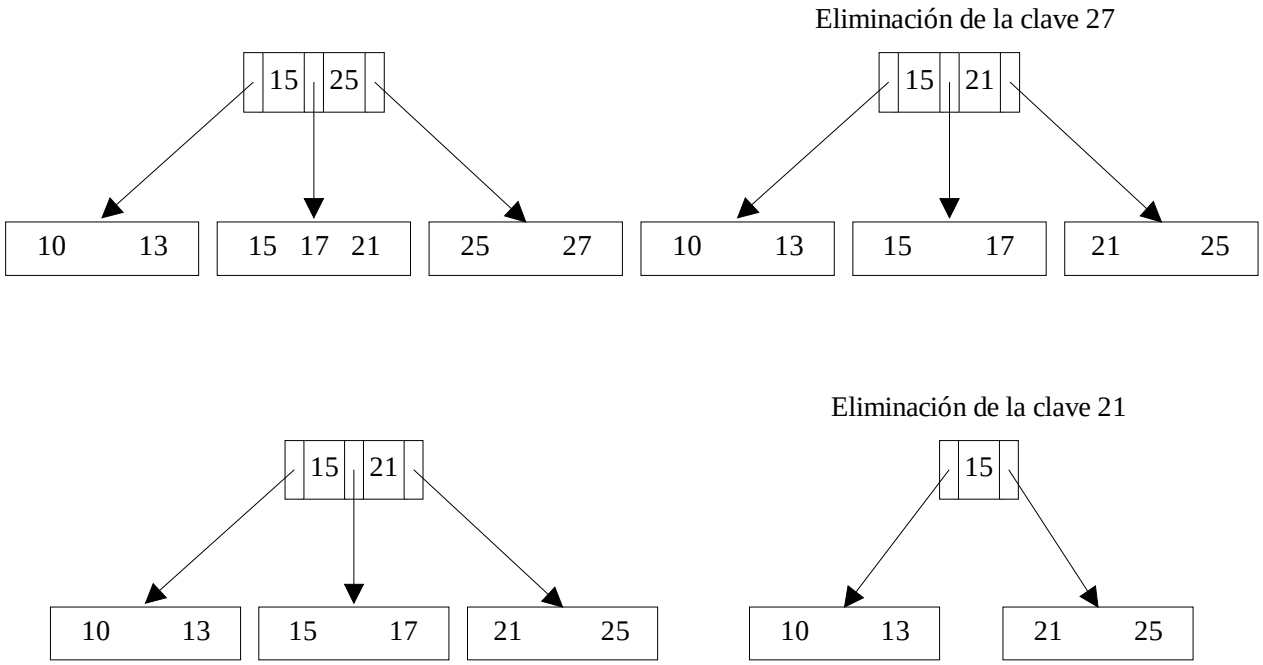
4.4.3.- Borrado en Arboles Multicamino B+

En este caso las claves a eliminar siempre se encuentran en las páginas hojas. Se deben distinguir dos casos:

1. Si al eliminar una clave, m queda mayor o igual a n entonces termina la operación de borrado. Las claves de las páginas raíz o internas no se modifican por más que sean una copia de la clave eliminada en las hojas.
2. Si al eliminar una clave, m queda menor a n entonces debe realizarse una redistribución de claves, tanto en el índice como en las páginas hojas. Puede suceder que al eliminar una clave y al realizar una redistribución de las mismas, la altura del árbol disminuya en una unidad.

Ejemplos:





4.5.- Arboles Multicamino Trie

Una estructura trie es esencialmente un árbol n-ario o multicamino. El término trie viene de la palabra "re**trie**val" que significa recuperar o recuperación. Dicha estructura se introdujo en el año 1960 por Fredkin.

A diferencia de las estructuras vistas anteriormente que basaban las búsquedas en la comparación entre claves, los trie (conocidos también como árboles digitales) utilizan la representación de las claves como secuencias de dígitos o caracteres alfabéticos.

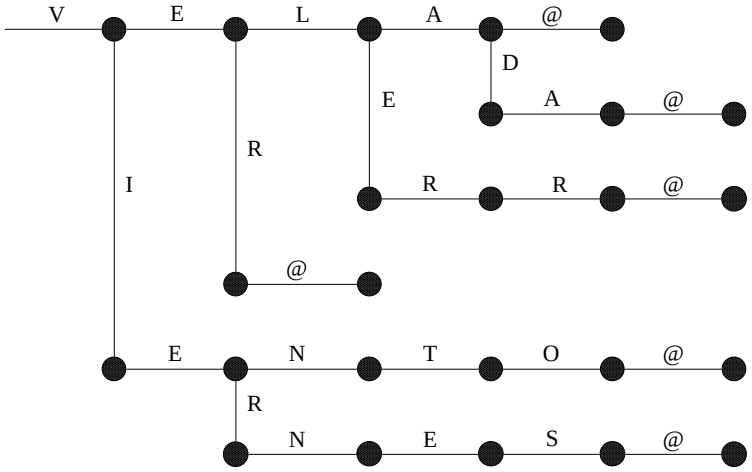
Los Trie son parte de los árboles multicamino y son útiles cuando la búsqueda de alguna clave se necesita hacer caracter por caracter. No son más que árboles de prefijos.

Cada nodo en un trie es un estado al que se llega recorriendo los arcos correspondientes a la secuencia de caracteres que este estado representa. Para evitar confusiones entre claves similares como "si" y "sin" usaremos un caracter especial '@' para marcar el final de una cadena de caracteres, de manera que ningún prefijo de una palabra puede ser en sí otra palabra. En otras palabras, en un trie, un camino completo desde la raíz hasta una hoja descendiente corresponde a una clave dentro del conjunto de claves posibles.

La altura del árbol está relacionada directamente a la cantidad de caracteres de las palabras ingresadas, así la palabra con mayor cantidad de caracteres será quién determine cuál es la altura correspondiente.

Veamos un ejemplo de Arbol Multicamino Trie:

Estructura Trie para las claves vela, velada, velero, ver, viento y viernes



4.5.1.- Representaciones de Arboles Multicamino Trie

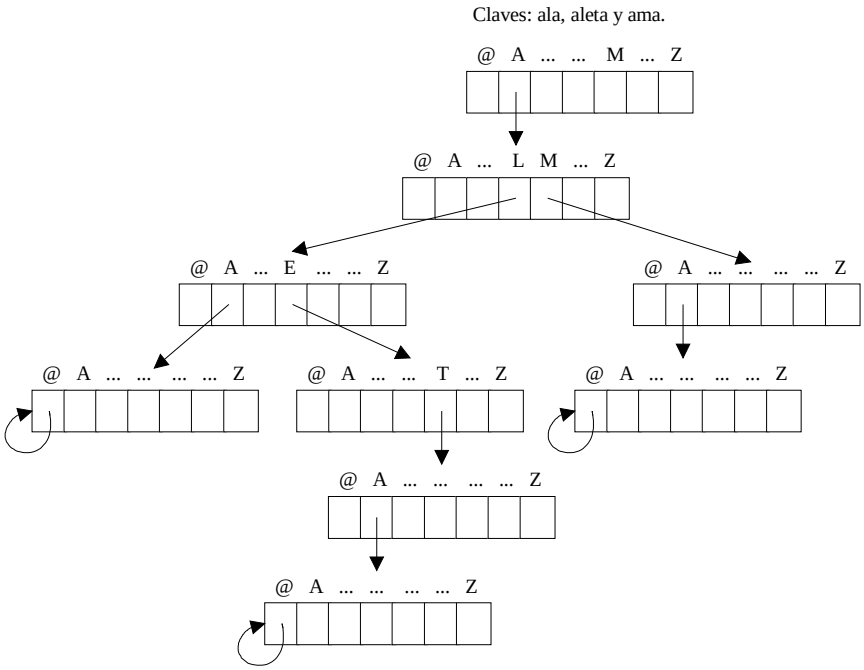
Existen varias formas de representar un Arbol Trie, entre las más conocidas tenemos:

- Mediante matrices

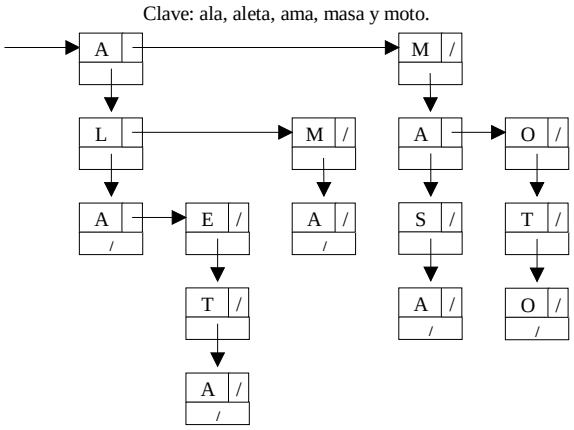
Claves: hija, hijo, hoja, lata, latita, leo y liz.

	@	A	B	C	D	E	F	G	H	I	J	K	L	...	O	...	T	...	Z
0									1				12						
1										2					8				
2											3								
3		4													6				
4	5																		
5																			
6	7																		
7																			
8											9								
9		10																	
10	11																		
11																			
12		13				21				24									
13																	14		
14		15								17									
15	16																		
16																			
17																	18		
18		19																	
19	20																		
20																			
21															22				
22	23																		
23																			
24																			25
25	26																		
26																			

- Mediante arreglos enlazados



- Mediante listas enlazadas



4.5.2.- Búsqueda, Inserción y Borrado en Arboles Multicamino Trie

En representaciones de arreglos enlazados cualquier operación es más rápida que listas enlazadas, pero consume mucho espacio (proporcional al número de estados multiplicado por el número de símbolos).
Cuando la representación es por medio de listas enlazadas, puede ser ineficiente en el tiempo de búsqueda de una clave cuando hay muchos arcos que parten del mismo nodo.

4.5.3.- Implementación de Arboles Multicamino Trie

Adaptación de los algoritmos de TRIE propuestos en el libro Estructuras de Datos y Algoritmos de Aho, Hopcroft y Ullman.

```
class Trie
	{
	TrieNode root;
	final int end=26;
	final int begin=0;

	Trie()
		{
		root=null;
		}

	void insertWord(String word)
		{
		TrieNode t;
		int i, pos;
		if(root==null)
			root=new TrieNode(end);
		t=root;
		i=0;
		//pasar word a minúscula o mayúscula
		while (i<word.length())
			{
			pos=getPosition(word.charAt(i));
			//falta validar posición pos
			if(t.getValueAt(pos) == null)
				t.setValueAt(pos,new TrieNode(end));
			t=t.getValueAt(pos);
			i++;
			}
		t.setValueAt(begin, t);
		}

	boolean searchWord(String word)
		{
		TrieNode t=root;
		int i=0;
		if(t==null)
			return false;
		int pos;
		//ver el tema de minúsculas y mayúsculas
		while (i<word.length())
			{
			pos=getPosition(word.charAt(i));
			//falta validar posición pos
			if(t.getValueAt(pos)!=null)
				{
				t=t.getValueAt(pos);
				i++;
				}
			else
				return false;
			}
		if(t.getValueAt(begin)==t)
			return true;
		else
			return false;
		}

	//dos formas de obtener la posición de una letra (elegir una)
	//primera implementación de getPosition
	static int getPosition(char c)
		{
		switch(c)
			{
			case '@': return 0;
```

```
                        case 'a': return 1;
                        case 'b': return 2;
                        case 'c': return 3;
                        case 'd': return 4;
                        case 'e': return 5;
                        case 'f': return 6;
                        case 'g': return 7;
                        case 'h': return 8;
                        case 'i': return 9;
                        case 'j': return 10;
                        case 'k': return 11;
                        case 'l': return 12;
                        case 'm': return 13;
                        case 'n': return 14;
                        case 'o': return 15;
                        case 'p': return 16;
                        case 'q': return 17;
                        case 'r': return 18;
                        case 's': return 19;
                        case 't': return 20;
                        case 'u': return 21;
                        case 'v': return 22;
                        case 'w': return 23;
                        case 'x': return 24;
                        case 'y': return 25;
                        case 'z': return 26;
                        }
                return -1;
                }

        //segunda implementación de getPosition
        static int getPosition(char c)
                {
                Character caux=new Character(c);
                if(c=='@')
                        return 0;
                else
                        if(c>='a' & c<='z')
                                return caux.hashCode()-96;
                        else
                                return -1;
                }

//Mediante arreglos enlazados
class TrieNode
        {
        TrieNode[] chars;

        TrieNode(int end)
                {
                chars=new TrieNode[end+1];
                }

        TrieNode getValueAt(int pos)
                {
                //falta validar posición
                return chars[pos];
                }

        void setValueAt(int pos, TrieNode newNode)
                {
                //falta validar posición
                chars[pos]=newNode;
                }
        }

//Mediante listas enlazadas
class TrieNode
        {
        char character;
```

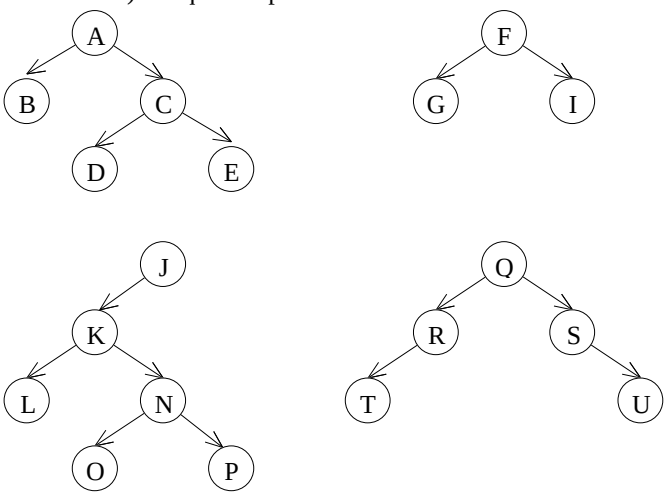
```
TrieNode right; //abecedario
TrieNode down; //palabra
.
.
.
}
```

5.- BOSQUES

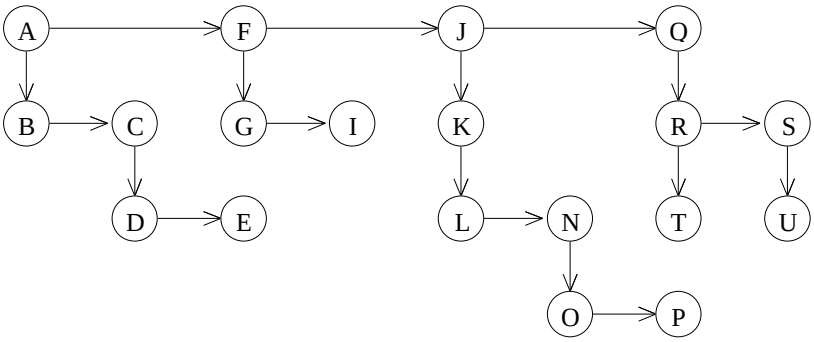
Un bosque representa un conjunto normalmente ordenado de uno o más árboles generales, que normalmente tienen algo en común. Es posible representar un bosque mediante un árbol binario. Los pasos que deben aplicarse para lograr la conversión son:

1. Deben enlazarse en forma horizontal las raíces de los distintos árboles generales.
2. Deben enlazarse los hijos de cada nodo en forma horizontal (los hermanos).
3. Deben enlazarse en forma vertical el nodo padre con el hijo que se encuentra más a la izquierda. Además, debe eliminarse el vínculo de ese padre con el resto de sus hijos.
4. Debe rotarse el diagrama resultante, aproximadamente 45 grados hacia la izquierda y así se obtendrá el árbol binario correspondiente.

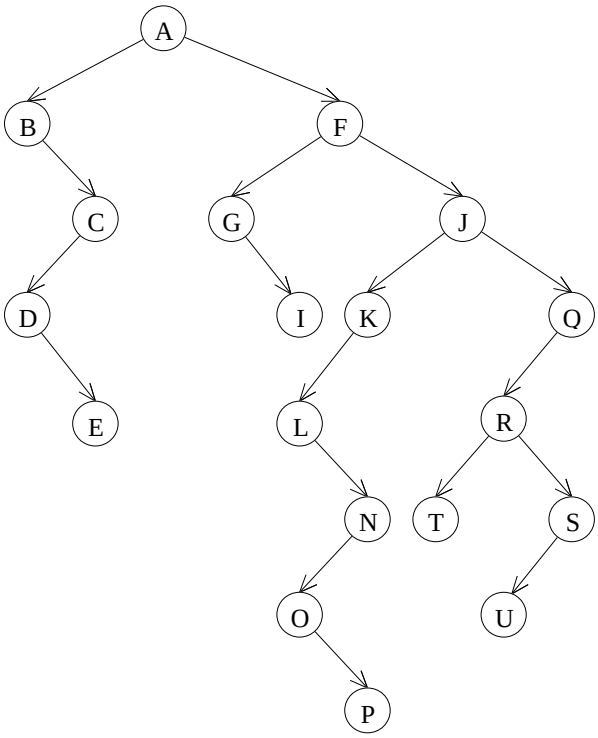
a) Bosque compuesto de cuatro árboles



b) Arbol luego de aplicar el primer, segundo y tercer paso



c) Arbol binario luego de aplicar el cuarto paso



GRAFOS

1.- INTRODUCCION

Hemos hablado de tipos de datos lineales como una colección de elementos donde cada uno de ellos tiene un elemento siguiente y uno anterior. Los árboles son tipos de datos no lineales, en donde la restricción anterior se relaja y cada elemento puede tener más de un elemento siguiente (hijos), pero solo tienen un elemento anterior (padre del nodo). Ahora vamos a generalizar más la estructura de árbol para permitir que un elemento pueda tener más de un elemento anterior. Un grafo es una estructura donde cada elemento pueda tener cero, uno o muchos elementos anteriores y siguientes. Esta generalización permite utilizar esta estructura para reflejar muchas situaciones del mundo real.

2.- TERMINOLOGIA FUNDAMENTAL

Existen dos teorías paralelas, la de grafos dirigidos y la de grafos no dirigidos.
Un grafo dirigido o digrafo G se define como un par (V, A) donde V es un conjunto de elementos, y A es una relación binaria definida sobre V. Los elementos de V se denominan nodos o vértices, y los elementos de A arcos. Dado un arco (x, y) de A, se dice que x es el origen e y el destino del arco.

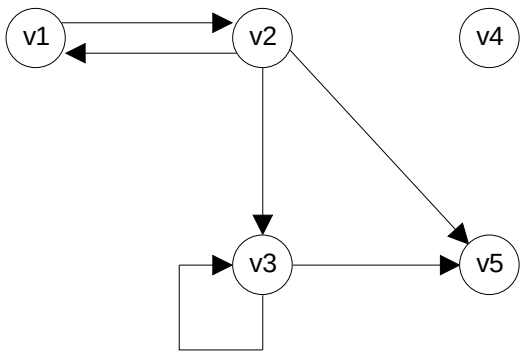
Ejemplo:
$V = \{v_1, v_2, v_3, v_4\}$
$A = \{(v_1, v_2), (v_2, v_1), (v_3, v_2), (v_3, v_3)\}$

Cada par (x, y) de A es un par ordenado (por ser A una relación) y es esto lo que determina que el grafo sea dirigido (cada arco tiene un sentido). Si el par (x, y) no fuera considerado ordenado el grafo sería un grafo no dirigido. En este último caso el par (x, y) es igual al par (y, x). Si G es un grafo no dirigido los elementos de A se denominan aristas.

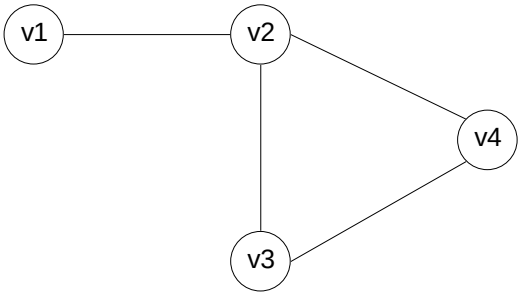
2.1.- Representación gráfica de grafos

Para representar gráficamente un digrafo generalmente se utilizan círculos para los vértices y flechas para los arcos. En un grafo no dirigido los vértices también se representan con círculos, pero las aristas se representan con segmentos.

Ejemplos:
G1 = (V, A) grafo dirigido
$V = \{v_1, v_2, v_3, v_4, v_5\}$
$A = \{(v_1, v_2), (v_2, v_1), (v_2, v_3), (v_2, v_5), (v_3, v_3), (v_3, v_5)\}$

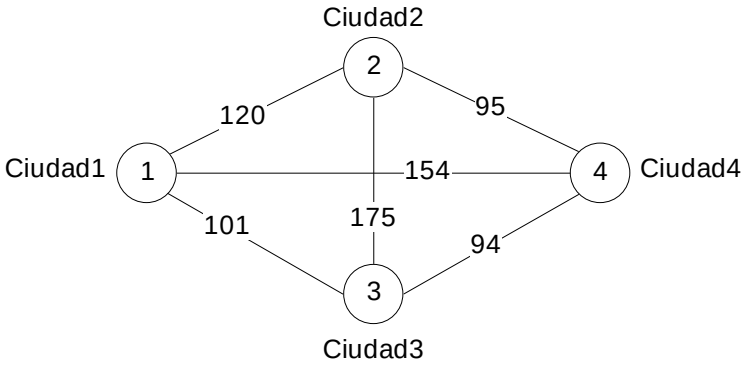


G2 = (V, A) grafo no dirigido
V = {v_1, v_2, v_3, v_4}
A = {(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4)}

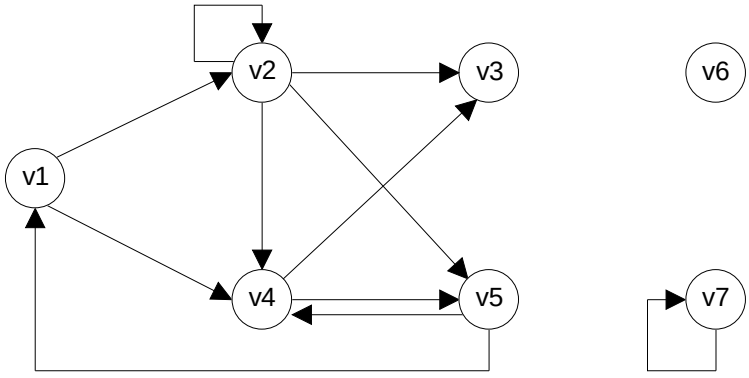


En G2 no podemos tener como elemento de A a la arista (v_2, v_1) dado que es igual a la arista (v_1, v_2). Si la agregamos, A tendría elementos repetidos y esto no está permitido porque A es un conjunto.

Los vértices y arcos (o aristas) de un grafo pueden ser elementos con información adicional unida a ellos. Tales grafos se llaman grafos etiquetados o ponderados. Por ejemplo:



2.2.- Definiciones básicas en grafos dirigidos
Se dan a continuación una serie de conceptos definidos sobre digrafos. Para ejemplificar los mismos usaremos el siguiente grafo:



Bucle

Un bucle es un arco de la forma (x, x), es decir es un arco que tiene por origen y destino al mismo vértice.

Ejemplo:
En el grafo dado existen dos bucles: el (v_2, v_2) y el (v_7, v_7).

Sucesor

Se dice que el nodo y es sucesor o adyacente del nodo x si existe un arco que tenga por origen a x y por destino a y, es decir sí el arco (x, y) $\in$ A.
Al conjunto de nodos sucesores de x lo denotaremos con S(x).

Ejemplo:
Calculemos el conjunto de sucesores de v_4. Existen dos arcos que tienen como origen a v_4: (v_4, v_3) y (v_4, v_5), por lo tanto el conjunto de sucesores de v_4 lo forman los vértices v_3 y v_5.

$S(v_4) = \{v_3, v_5\}$

Si hacemos el mismo análisis para v_2, existen cuatro arcos que tienen como origen a v_2: (v_2, v_2), (v_2, v_3), (v_2, v_4) y (v_2, v_5), por lo tanto el conjunto de sucesores de v_2 lo forman los vértices v_2, v_3, v_4 y v_5.

$S(v_2) = \{v_2, v_3, v_4, v_5\}$

Sin embargo no existe ningún arco que tenga como origen a v_3, por lo que el conjunto de sucesores de v_3 es vacío. Algo similar ocurre con v_6.

$S(v_3) = S(v_6) = \varnothing$

Predecesor

Se dice que el nodo y es predecesor del nodo x si existe un arco que tenga por origen a y y por destino a x, es decir sí el arco (y, x) $\in$ A.
Al conjunto de nodos predecesores de x lo denotaremos con P(x).

Ejemplo:
El vértice v_4 es destino de tres arcos (v_1, v_4), (v_2, v_4) y (v_5, v_4), por lo que el conjunto de predecesores de v_4 lo forman v_1, v_2 y v_5.

$P(v_4) = \{v_1, v_2, v_5\}$

El vértice v_2 es destino de dos arcos (v_1, v_2) y (v_2, v_2), por lo tanto el conjunto de predecesores de v_2 lo forman v_1 y v_2.

$P(v_2) = \{v_1, v_2\}$

Notar que en el caso v_2, el mismo v_2 forma parte de su conjunto de predecesores y de su conjunto de sucesores. Esto se debe a que existe un bucle en v_2.

Grado

Grado de entrada del nodo x:

$g^-(x) = |P(x)|$

Grado de salida del nodo x:

$g^+(x) = |S(X)|$

Grado del nodo x:

$g(x) = |P(x) \cup S(X)|$

Ejemplo:
Si hacemos el cálculo de grados para v_4:
$g^-(v_4) = |\{v_1, v_2, v_5\}| = 3$
$g^+(v_4) = |\{v_3, v_5\}| = 2$
$g(v_4) = |\{v_1, v_2, v_5\} \cup \{v_3, v_5\}| = 5$

Hacemos el mismo cálculo para v_2:
$g^-(v_2) = |\{v_1, v_2\}| = 2$
$g^+(v_2) = |\{v_2, v_3, v_4, v_5\}| = 4$
$g(v_2) = |\{v_1, v_2\} \cup \{v_2, v_3, v_4, v_5\}| = 5$

Notar que si el vértice no tiene un bucle, el grado del mismo es igual a la suma de los grados de entrada y de salida.

Vértice aislado
Se dice que un vértice x es aislado sí g(x)=0.

Ejemplo:
En el grafo anterior el único vértice aislado es v_6:
$g(v_6) = |P(v_6) \cup S(v_6)| = |\varnothing| = 0$

El vértice v_7 no puede ser considerado como aislado, dado que al tener un bucle su grado es 1.
En general, con la definición de grado que hemos visto, para que un vértice se considere aislado no tiene que ser origen ni destino de ningún arco, ni siquiera un bucle.

Camino
Informalmente decimos que existe un camino del vértice x al vértice y, si podemos llegar de x a y siguiendo los arcos y en el sentido en que estos están.
En el grafo del ejemplo existe un camino de v_1 a v_5 siguiendo los arcos (v_1, v_2), (v_2, v_4) y (v_4, v_5). También existe otro camino entre los mismos vértices siguiendo los arcos (v_1, v_4) y (v_4, v_5).
Formalmente, dado un grafo G = (V, A) decimos que existe un camino del vértice x al vértice y, si existe una secuencia $<v_0, v_1, ..., v_{n-1}>$ tal que:

a) $v_0 = x \wedge v_{n-1} = y$
b) $(v_{i-1}, v_i) \in A$ con i=1...n-1

La longitud del camino es igual a la cantidad de arcos que lo forman; en la definición anterior sería n-1.
Como caso especial la secuencia <v> es un camino de longitud cero. Esto nos dice que siempre existe un camino de longitud cero entre un nodo y sí mismo.
Un camino se dice simple o elemental si no pasa dos veces por un mismo vértice.
Un ciclo es un camino simple de longitud mayor que 1, en donde coinciden los vértices primero y último.

Ejemplo:
Existe un camino de v_1 a v_5 dado que la secuencia $<v_1, v_2, v_4, v_5>$ cumple con la definición. La longitud de este camino es 3.
No existe ningún camino con origen en v_3 (de v_3 no se puede llegar a ningún otro vértice del grafo).
La secuencia $<v_2, v_2>$ también cumple con la definición de camino; en general un bucle es un camino de longitud uno. Pero este camino no puede ser considerado un ciclo (por su longitud).
El camino $<v_1, v_2, v_4, v_5, v_4, v_5, v_1>$ tampoco es un ciclo dado que no es simple.
La secuencia $<v_1, v_2, v_4, v_5, v_1>$ si es un ciclo y su longitud es 4.

Cadena

Un concepto similar al de camino pero menos restrictivo es el de cadena.
Informalmente decimos que existe una cadena del vértice x al vértice y, si podemos llegar de x a y siguiendo los arcos pero sin tener en cuenta el sentido de los mismos.
Habíamos visto que no existía ningún camino con origen en v_3. Pero si existen cadenas con inicio en ese vértice, por ejemplo v_3, v_4, v_5.
Formalmente, dado un grafo G = (V, A) decimos que existe una cadena del vértice x al vértice y, si existe una secuencia v_0, v_1, ..., v_{n-1} tal que:

a) $v_0 = x \wedge v_{n-1} = y$
b) $(v_{i-1}, v_i) \in A$ con i=1...n-1

La longitud de la cadena es igual a la cantidad de arcos que la forman; en la definición anterior sería n-1.
Una cadena se dice simple si no pasa dos veces por un mismo vértice.
Un circuito es una cadena simple de longitud mayor que 1, en donde coinciden los vértices primero y último.
De las definiciones dadas se deduce que:

a) Todo camino es una cadena, pero no toda cadena es un camino.
b) Todo ciclo es un circuito, pero no todo circuito es un ciclo.
c) Existe una cadena de x a y si y sólo si existe una cadena de y a x.

Ejemplo:
Las siguientes secuencias son ejemplos de cadena: $<v_1, v_2, v_4, v_5>$, $<v_2, v_1, v_4>$, $<v_3, v_2, v_4, v_5>$, $<v_2, v_2>$.
Las siguientes secuencias además de cadenas son circuitos: $<v_5, v_2, v_4, v_5>$, $<v_1, v_2, v_4, v_5, v_1>$, $<v_3, v_4, v_2, v_3>$.
No existe ninguna cadena con origen en v_6. Esto se debe a que v_6 es un vértice aislado.
Existe una sola cadena con inicio en v_7: $<v_7, v_7>$ (el único arco con origen en v_7 es el bucle).

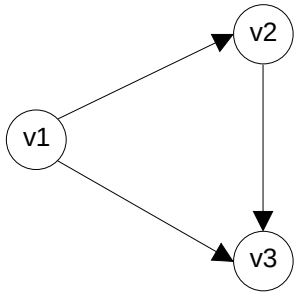
Conectividad

Un grafo se dice conectado o conexo si existe una cadena entre cualquier par de vértices del grafo.
Un grafo se dice fuertemente conectado o fuertemente conexo si existe un camino entre cualquier par de vértices del grafo.
Nuevamente a partir de estas definiciones podemos sacar algunas conclusiones:

a) Si G es un grafo fuertemente conectado entonces G es un grafo conectado (el recíproco no se da).
b) Si G no es un grafo conectado entonces G no es un grafo fuertemente conectado.
c) Si G tiene vértices aislados entonces G no es conectado.

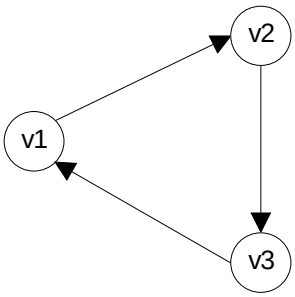
Ejemplos:
G1: el grafo que hemos usado en los ejemplos anteriores no es un grafo conectado, y por lo tanto tampoco es fuertemente conectado

G2:



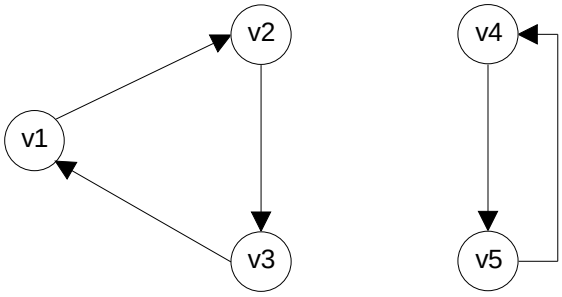
Es un grafo conectado, pero no es fuertemente conectado: (no existe camino entre v3 y v2)

G3:



Es un grafo conectado y también es fuertemente conectado (ciclo)

G4:



No es conectado, porque por ejemplo no existe un camino entre v5 y v3, por consiguiente tampoco es fuertemente conectado

3.- TDA GRAFO

Como tipo de dato abstracto un grafo G = (V, A) consta de un conjunto de vértices V, de un conjunto de arcos A, y operaciones actuando sobre estos dos conjuntos.
Esto implica la necesidad de definir dos nuevas clases llamadas Vertice y Arco. Un vértice es un objeto compuesto de una etiqueta. Un arco es un objeto compuesto de dos vértices y opcionalmente de una etiqueta.

4.- IMPLEMENTACIONES DE GRAFOS

Para representar un grafo se pueden emplear varias estructuras de datos. La selección apropiada depende de las operaciones que se realizarán sobre los vértices y sobre los arcos del grafo.

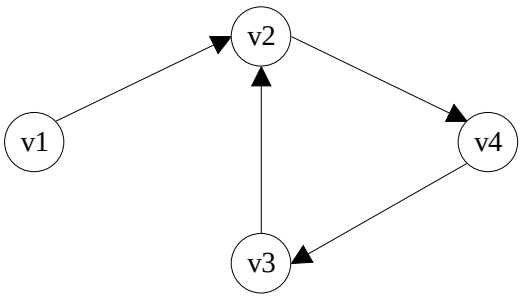
Vamos a ver dos posibles implementaciones:

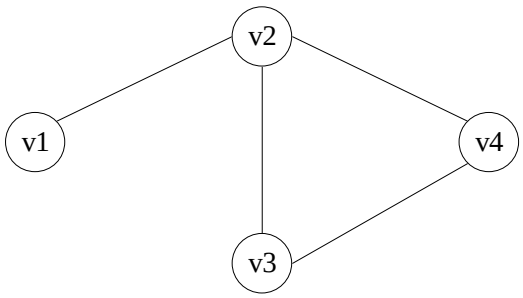
- Mediante matrices de adyacencia.
- Mediante listas de adyacencia.

4.1.- Mediante matrices de adyacencia

Esta representación utiliza, una secuencia de vértices, y una matriz (array bidimensional) que permite determinar la existencia de adyacencias entre pares de vértices en un tiempo constante. Esta mejora se ve contrarrestada por el incremento en el espacio utilizado, que es del $O(n^2)$.
Cada vértice tiene asociado un índice.
Para un grafo con n vértices, dispondremos de una matriz M de NxN, tal que M[i , j] contiene la referencia al arco A que tiene como origen el vértice i-ésimo y como destino el vértice j-ésimo. Si no hay un arco entre los vértices i y j , entonces M[i , j] almacena una referencia *null*.

Si el grafo es no dirigido, entonces la referencia a la arista a se almacena en M[i , j] y M[j , i]. Es decir, se trata de una matriz simétrica.
La representación gráfica es de la siguiente forma:





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

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

La implementación básica para un grafo dirigido es la siguiente:

```
public class Graph
	{
	private Vertex[] vertices;
	private int vertexPosition;
	private boolean[][] edges;
	private int vertexQuantity;

	Graph(int quantity)
		{
		vertexQuantity=quantity;
		vertices=new Vertex[vertexQuantity];
		vertexPosition=0;
		edges=new boolean[vertexQuantity][vertexQuantity];
		}

	public void insertVertex(Object element)
		{
		vertices[vertexPosition]=new Vertex();
		vertices[vertexPosition].setElement(element);
		vertexPosition++;
		}

	public void insertEdge(Object originElement, Object finishElement)
		{
		int originPosition=getVertexOrder(originElement);
		int finishPosition=getVertexOrder(finishElement);
		edges[originPosition][finishPosition]=true;
		}

	private int getVertexOrder(Object element)
		{
		int position=0, order=-1;
		boolean found=false;

		while(position<vertexQuantity & found==false)
			{
			if(vertices[position].getElement().equals(element))
				{
```

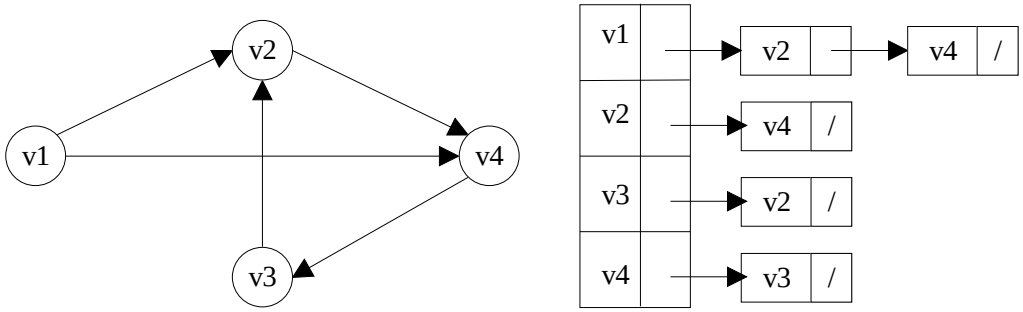
```
                                found=true;
                                order=position;
                                }
                        position++;
                        }

                return order;
                }
        }

public class Vertex
        {
        private Object element;

        Vertex()
                {
                element=null;
                }

        public Object getElement()
                {
                return element;
                }

        public void setElement(Object element)
                {
                this.element=element;
                }
        }
```

Cuando un grafo es esparcido o disperso, es decir, la mayoría de los términos de la matriz de adyacencia son *null*, esta implementación no es eficiente. Es adecuada cuando el grafo es denso (existen muchos arcos o aristas)

4.2.- Mediante listas de adyacencia

En esta representación, se mantiene una secuencia con los vértices del grafo. Además para cada vértice del grafo se mantiene una secuencia con todos sus vértices adyacentes. En un grafo con n vértices habrá n secuencias de vértices adyacentes. Las referencias a las secuencias se almacenarán en un vector de tamaño n. Los vértices suelen identificarse por un nombre o etiqueta. En tal caso, se necesita asociar un índice a cada uno de ellos que nos permita acceder a la secuencia de vértices adyacentes.

La representación gráfica es de la siguiente forma:



La implementación básica para un grafo dirigido es la siguiente:

```
public class Graph
        {
        private Vertex[] vertices;
        private int vertexPosition;
```

```
	private int vertexQuantity;

	Graph(int quantity)
		{
		vertexQuantity=quantity;
		vertices=new Vertex[vertexQuantity];
		vertexPosition=0;
		}

	public void insertVertex(Object element)
		{
		vertices[vertexPosition]=new Vertex();
		vertices[vertexPosition].setElement(element);
		vertexPosition++;
		}

	public void insertEdge(Object originElement, Object finishElement)
		{
		int originPosition=getVertexOrder(originElement);
		Edge edge=vertices[originPosition].getEdge();

		Edge newEdge=new Edge();
		newEdge.setPosition(getVertexOrder(finishElement));

		if(edge==null)
			{
			vertices[originPosition].setEdge(newEdge);
			}
		else
			{
			while(edge.getEdge()!=null)
				edge=edge.getEdge();
			edge.setEdge(newEdge);
			}
		}

	private int getVertexOrder(Object element)
		{
		int position=0, order=-1;
		boolean found=false;

		while(position<vertexQuantity & found==false)
			{
			if(vertices[position].getElement().equals(element))
				{
				found=true;
				order=position;
				}
			position++;
			}

		return order;
		}

public class Vertex
	{
	private Object element;
	private Edge edge;

	Vertex()
		{
		element=null;
		edge=null;
		}

	public Object getElement()
		{
		return element;
		}
```

```
        public Edge getEdge()
                {
                return edge;
                }

        public void setElement(Object element)
                {
                this.element=element;
                }

        public void setEdge(Edge edge)
                {
                this.edge=edge;
                }
        }

public class Edge
        {
        private int position;
        private Edge edge;

        Edge()
                {
                position=0;
                edge=null;
                }

        public int getPosition()
                {
                return position;
                }

        public Edge getEdge()
                {
                return edge;
                }

        public void setPosition(int position)
                {
                this.position=position;
                }

        public void setEdge(Edge edge)
                {
                This.edge=edge;
                }
        }
```

Faltarían las implementaciones correspondientes a los grafos no dirigidos, cuyo código es muy similar a los anteriormente citados, solamente que se tendrá en cuenta el hecho de que cada vez que se crea un arco de v_i, a v_j, también se debe agregar el arco de v_j a v_i.

5.- OPERACIONES SOBRE GRAFOS

5.1.- Recorridos

La operación de recorrer un grafo consiste en partir de un vértice determinado y visitar todos aquellos vértices que son accesibles desde él en un determinado orden. Para realizar la operación pueden seguirse dos estrategias: el recorrido en profundidad (DFS: Depth First Search) o recorrido en anchura (BFS: Breadh First Search).

5.1.1.- Recorrido en profundidad

Es una generalización del recorrido en preorden de un árbol: se comienza visitando un vértice cualquiera y, a continuación, se recorre en profundidad el componente conexo que “cuelga” de cada sucesor.
Este método de recorrido se realiza de acuerdo a los siguientes pasos:

1. Se visita el vértice del que se parte, v.
2. Se selecciona un vértice w, adyacente a v y que todavía no se haya visitado.
3. Se realiza el recorrido en profundidad partiendo del vértice w.
4. Cuando se encuentra un vértice cuyo conjunto de adyacentes han sido visitados en su totalidad se retrocede hasta el último vértice visitado que tenga vértices adyacentes no visitados y se ejecuta el paso 2.

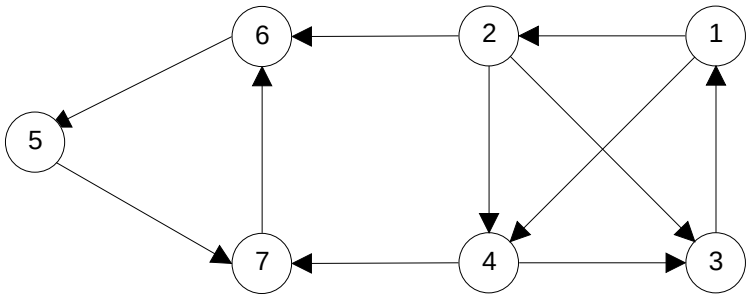
Suponemos la existencia de un conjunto (visitados) para ir almacenando los vértices del grafo por los que se va pasando. En principio el conjunto de vértices visitados estará vacío.
Es posible que algunos vértices del grafo no se hayan visitado, en cuyo caso se selecciona alguno de ellos como nuevo vértice de partida y se repite el proceso hasta que todos los vértices de G se hayan visitado.

La implementación del algoritmo es la siguiente:

```
public void depthFirstSearch(Object element)
        {
        Vector visited=new Vector(vertexQuantity);
        depthFirst(getVertexOrder(element), visited);
        }

private void depthFirst(int element, Vector visited)
        {
        System.out.print(vertices[element].getElement()+" ");
        visited.addElement(new Integer(element));
        Enumeration adjs=adjacents(new Integer(element));
        while(adjs.hasMoreElements())
                {
                Integer adjsOther=(Integer)adjs.nextElement();
                if(!visited.contains(adjsOther))
                        depthFirst(adjsOther.intValue(), visited);
                }
        }
```

Supóngase el siguiente grafo dirigido:

Vamos a realizar su recorrido en profundidad tomando como vértice de partida 1.



El vértice 1 pasa al conjunto de visitados (Visitados = {1}) y obtiene el conjunto de adyacentes de 1 que son {2, 4}.
Se recorre en profundidad el vértice 2 pasando al conjunto de visitados (Visitados = {1, 2}) y se obtienen sus adyacentes; este caso {3, 4, 6}.
Como el vértice 3 no se ha visitado se recorre en profundidad. Se inserta en el conjunto de visitados (Visitados = {1, 2, 3}) y se obtienen su adyacentes; en este caso el vértice {1}. Como el vértice 1 ya está en el conjunto de visitados terminó el recorrido en profundidad del vértice 3.
Pasamos entonces al otro adyacente del 2 que era el vértice 4. Se recorre en profundidad el vértice 4. Se inserta en el conjunto de visitados (Visitados = {1, 2, 3, 4}) y se obtienen su adyacentes; en este caso los vértices {3, 7}. Como el vértice 3 ya está visitado se recorre en profundidad el vértice 7.

Se inserta el 7 en el conjunto de visitados (Visitados = {1, 2, 3, 4, 7}) y se obtienen su adyacentes; en este caso el vértice {6}.
Como el vértice 6 no se ha visitado se recorre en profundidad. Se inserta en el conjunto de visitados (Visitados = {1, 2, 3, 4, 7, 6}) y se obtienen su adyacentes; en este caso el vértice {5}.
Como el vértice 5 tampoco se ha visitado se recorre en profundidad el vértice 5. Se inserta en el conjunto de visitados (Visitados = {1, 2, 3, 4, 7, 6, 5}) y se obtienen sus adyacentes; en este caso el vértice {7}. Como ya se ha visitado termina el recorrido en profundidad del vértice 4 y con el también termina el recorrido de los vértices 5, 6, y 7.
Pasamos al último adyacente del 2 que era el 6 que ya está visitado terminando el recorrido en profundidad del vértice 2.
Pasamos ahora al otro adyacente del vértice 1 que es el 4. Como ya está en el conjunto de visitados se termina el recorrido en profundidad del vértice 1 y con él el recorrido 5 6 7 2 4 1 3 en profundidad del grafo.
El listado de los vértices en profundidad es: **1 2 3 4 7 6 5**.

5.1.2.- Recorrido en anchura

Generaliza el recorrido en anchura (por niveles) de un árbol: después de visitar un vértice se visitan los sucesores, después los sucesores de los sucesores, y así reiteradamente.
Este método de recorrido consiste en los siguientes pasos:
1. Se visita el vértice de partida v.
2. Se visitan todos sus adyacentes que no estuvieran ya visitados y así sucesivamente. Esto es, se visitan todos los vértices adyacentes antes de pasar a otro vértice.

Utilizamos un conjunto (visitados) para ir almacenando los vértices del grafo por los que se va pasando. En una cola se mantienen los vértices adyacentes que se han obtenido a partir de los vértices visitados y cuyos adyacentes aún restan por explorar.
Inicialmente, tanto el conjunto de vértices visitados como la cola de vértices por visitar estarán vacíos.

La implementación del algoritmo es la siguiente:

```
public void breadhFirstSearch(Object element)
        {
        breadhFirst(getVertexOrder(element));
        }

private void breadhFirst(int element)
        {
        Vector visited=new Vector(vertexQuantity);
        Queue explore=new Queue();
        explore.insert(new Integer(element));
        visited.addElement(new Integer(element));
        do
                {
                Integer vertexOther=(Integer)explore.delete();
                System.out.print(vertices[vertexOther.intValue()].getElement()+" ");
                Enumeration adjs=adjacents(vertexOther);
                while(adjs.hasMoreElements())
                        {
                        Integer adjsOther=(Integer)adjs.nextElement();
                        if(!visited.contains(adjsOther))
                                {
                                explore.insert(adjsOther);
                                visited.addElement(adjsOther);
                                }
                        }
                }
        while(!explore.isEmpty());
        }
```

Supóngase el mismo grafo dirigido que antes. Hacemos un recorrido en anchura tomando como vértice de partida el vértice 1. Se inserta en la cola (Cola: <1>) y se inserta en el conjunto de visitados (Visitados = {1}).
Sacar de la cola el vértice 1. Obtener todos sus adyacentes, en este caso los vértices {2, 4}. Como no pertenecen al conjunto de visitados se insertan en ese conjunto (Visitados = {1, 2, 4}) y se insertan en la cola (Cola: <2, 4>).
Sacamos de la cola el vértice 2. Obtenemos todos sus adyacentes, en este caso los vértices {3, 4, 6}. Como 3 y 6 no pertenecen al conjunto de visitados se insertan en ese conjunto (Visitados = {1, 2, 4, 3, 6}) y se insertan en la cola (Cola: <4, 3, 6>).
Sacamos de la cola el vértice 4. Obtenemos todos sus adyacentes, en este caso los vértices 3 y 7. Como el vértice 7 no pertenecen al conjunto de visitados se inserta en dicho conjunto (Visitados = {12, 4, 3, 6, 7}) y se inserta en la cola (Cola: <3, 6, 7>).
Sacamos de la cola el vértice 3. Obtenemos todos sus adyacentes, en este caso el vértice {1}. Como ya pertenece al conjunto de visitados no hacemos nada (Cola: <6, 7>).
Sacamos de la cola el vértice 6. Obtenemos todos sus adyacentes, en este caso el vértice {5}. Como no pertenecen al conjunto de visitados se inserta en ese conjunto (Visitados = {1, 2, 4, 3, 6, 7, 5}) y se inserta en la cola (Cola: <7, 5>).
Sacamos de la cola el vértice 7. Obtenemos todos sus adyacentes, en este caso el vértice {6}. Como ya pertenece al conjunto de visitados no hacemos nada (Cola: <5>).
Sacamos de la cola el vértice 5. Obtenemos todos sus adyacentes, en este caso el vértice {7}. Como ya pertenece al conjunto de visitados no se procesa.
La cola ya se vació terminando el recorrido en anchura y produciendo el listado: **1 2 4 3 6 7 5**.

Para ambos recorridos será necesario el método adyacentes (para lista de adyacencia).

```
private Enumeration adjacents(Integer element)
        {
        Vector vertices=new Vector();
        Edge position=vertices[element.intValue()].getEdge();

        while(position != null)
                {
                vertices.addElement(new Integer(position.getPosition()));
                position=position.getEdge();
                }

        return vertices.elements();
        }
```

5.2.- Algoritmos de caminos mínimos

El algoritmo de recorrido en anchura puede utilizarse para buscar el camino más corto desde un vértice a otro cualquiera en un grafo conectado, suponiendo que todas las aristas son igualmente buenas.
En muchas situaciones los arcos tienen asociados costes distintos.

Ejemplos:

- Mapa de carreteras: Los vértices son las ciudades y el coste de cada arco se representa por la distancia en Km. entre las ciudades.
- Red de computadoras: Las conexiones entre los nodos que forman la red se representan por aristas cuyos costes estarán en función de las velocidades de dichas conexiones (líneas telefónicas vs. fibra óptica).

Sea G un grafo etiquetado. La **longitud** o **coste** de un camino P es la suma del coste de los arcos de P. Esto es, si P = $((v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k))$, la longitud de P, denotada por w(P), se define como

$$w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1}))$$

La distancia de un vértice *v* a otro *u* en G, denotada por *d(v, u)*, es la longitud del *camino de longitud mínima* (o, simplemente, **camino mínimo**) de *v* a *u*, si existe dicho camino.

Se utiliza la convención de que $d(v, u) = \infty$ si no hay un camino entre *v* y *u*.
Sea un grafo etiquetado G = (V, A), interesa encontrar el camino mínimo de un vértice *v* al resto de vértices del grafo. El algoritmo de Dijkstra calcula dichos caminos cuando los arcos tienen costes no negativos, es decir, $w(a) \geq 0 \ \forall a \in A$.

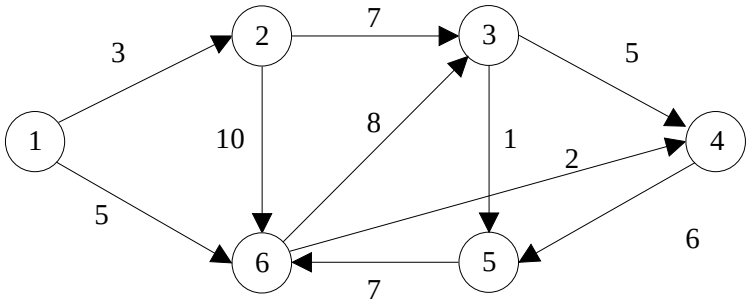
5.2.1.- Algoritmo de Dijkstra (árbol mínimo/máximo - camino mínimo/máximo - camino crítico)
Algoritmo voraz (greedy): en cada iteración se selecciona la mejor opción entre las disponibles. Este tipo de algoritmos se utiliza a menudo en situaciones donde se trata de optimizar una función de coste sobre una colección de objetos. También conocido como algoritmos ávidos.
Adaptación del esquema voraz al problema de caminos mínimos con origen en v: algoritmo que en cada iteración añade un vértice al conjunto de vértices *visitados*. El vértice seleccionado es aquel de los vértices por *visitar* más próximo a *v*. El algoritmo finaliza cuando no quedan vértices por visitar.
Para mantener en cada paso la distancia mínima desde el origen podemos usar un array llamado *distancia*, que mantiene en cada paso la distancia desde el origen a cada vértice del grafo.
En cada iteración, una vez seleccionado, de entre los vértices por *visitar*, el vértice *u* más próximo a *v* se actualiza el array *distancia* para aquellos vértices *w* que son adyacentes de *u* y que cumplen que *distancia[u] + w((u, w)) < distancia[w]*.

La implementación del algoritmo es la siguiente:

```
public Vector dijkstraAlgorithm(Object vertex)
        {
        return dijkstra(getVertexOrder(vertex));
        }

private Vector dijkstra(int vertex)
        {
        int vs;
        Vector distance=new Vector(vertexQuantity);
        Vector toVisit=new Vector(vertexQuantity);
        for(vs=0; vs<vertexQuantity; vs++)
                {
                Integer duv=null;
                if(vs==vertex)
                        duv=new Integer(0);
                else
                        duv=new Integer(INFINITO);
                distance.insertElementAt(duv, vs);
                toVisit.addElement(new Integer(vs));
                }
        while(!toVisit.isEmpty())
                {
                Integer u=minimum(distance, toVisit.iterator());
                toVisit.removeElement(u);
                int du=((Integer)distance.get(u.intValue())).intValue();
                if(du != INFINITO)
                        {
                        Enumeration adjs=adjacents(u);
```

```
                        while(adjs.hasMoreElements())
                                {
                                Integer w=(Integer)adjs.nextElement();
                                if(toVisit.contains(w))
                                        {
                                        int cuw=getEdge(u, w).getCoste();
                                        if(du + cuw < ((Integer)distance.get(w.intValue())).intValue())
                                                distance.set(w.intValue(), new Integer(du + cuw));
                                        }
                                }
                        }
                }
        return distance;
        }

private Integer minimum(Vector distance, Iterator toVisitI)
        {
        Integer vertex, vertexMinimum=(Integer)toVisitI.next();
        int distanceValue, distanceMinimum=((Integer)distance.get(vertexMinimum.intValue())).intValue();
        while(toVisitI.hasNext())
                {
                vertex=(Integer)toVisitI.next();
                distanceValue=((Integer)distance.get(vertex.intValue())).intValue();
                if(distanceValue < distanceMinimum)
                        {
                        vertexMinimum=vertex;
                        distanceMinimum=distanceValue;
                        }
                }
        return vertexMinimum;
        }
```

Ejemplo de caminos mínimos a partir del vértice 1:



		distancia						
visitados	**porVisitar**	**1**	**2**	**3**	**4**	**5**	**6**	**v mín.**
∅	{1, 2, 3, 4, 5, 6}	0	∞	∞	∞	∞	∞	1
{1}	{2, 3, 4, 5, 6}	0	3	∞	∞	∞	5	2
{1, 2}	{3, 4, 5, 6}	0	3	10	∞	∞	5	6
{1, 2, 6}	{3, 4, 5}	0	3	10	7	∞	5	4
{1, 2, 6, 4}	{3, 5}	0	3	10	7	13	5	3
{1, 2, 6, 4, 3}	{5}	0	3	10	7	11	5	5
{1, 2, 6, 4, 3, 5}	∅	0	3	10	7	11	5	

El coste total del algoritmo es $O(n^2)$.

Para reconstruir el camino mas corto a cada vértice, se agrega otro arreglo *Camino* de vértices, tal que *caminos[v]* contenga el vértice inmediato anterior a *v* en el camino más corto.
Se asigna un valor inicial a *camino[v]* igual a 0 para todo $v \neq 0$. El arreglo Camino puede actualizarse después de la actualización del arreglo *distancia* en el método de *dijkstra(int).*

```
if(du + cuw < ((Integer)distance.get(w.intValue())).intValue())
        {
        distance.set(w.intValue(), new Integer(du + cuw));
        way[w.intValue()]=u.intValue();
        }
```

Al término de Dijkstra, el camino a cada vértice puede encontrarse regresando por los vértices predecesores del arreglo *camino*. Por ejemplo, para reconstruir el camino más corto del vértice 1 (origen) al vértice 4, sería:

```
v ← 4
hacer
        {
        Escribir v
        v ← way[v]
        }
mientras(v != 1)
```

5.2.2.- Algoritmo de Floyd-Warshall (camino mínimo entre todos los pares de nodos)

El problema que intenta resolver este algoritmo es el de encontrar el camino más corto entre todos los pares de nodos o vértices de un grafo. Esto es semejante a construir una tabla con todas las distancias mínimas entre pares de ciudades de un mapa, indicando además la ruta a seguir para ir de la primera ciudad a la segunda. Este es uno de los problemas más interesantes que se pueden resolver con algoritmos de grafos.
Existen varias soluciones a este problema y los algoritmos a aplicar dependen también de la existencia de arcos con pesos o costes negativos en el grafo. En el caso de no existir pesos negativos, sería posible ejecutar n veces el algoritmo de *Dijkstra* para el cálculo del camino mínimo, donde n es el número de vértices o nodos del grafo. Esto conllevaría un tiempo de ejecución de $O(n^3)$ (aunque se puede reducir). Si existen arcos con pesos negativos, se puede ejecutar también n veces el algoritmo de Bellman-Ford (que se verá mas adelante), una vez para cada nodo del grafo. Para grafos densos (con muchas conexiones o arcos) esto conllevaría un tiempo de ejecución de $O(n^4)$.
El algoritmo de Floyd-Warshall ("All-Pairs-Shortest-Path" - Todos los caminos mínimos) ideado por Floyd en 1962 basándose en un teorema de Warshall también de 1962, usa la metodología de Programación Dinámica para resolver el problema. Éste puede resolver el problema con pesos negativos y tiempos de ejecución iguales a $O(n^3)$; sin embargo, para ciclos de peso negativo el algoritmo tiene problemas.
Este algoritmo se puede aplicar a multitud de problemas, incluyendo el diseño de circuitos, el diseño de rutas de transporte, aproximaciones al problema del viajante de comercio, o como base de otros algoritmos más complejos.

La implementación del algoritmo es la siguiente:

```
public int[][] floyd()
        {
        int u, v, w;
        int[][] floydMatrix=//asignarle la matriz de adyacencia
        //no se aceptan bucles, ni ciclos
        for(u=0; u<vertexQuantity; u++)
                floydMatrix[u][u]=0;
        for(u=0; u<vertexQuantity; u++)
                for(v=0; v<vertexQuantity; v++)
                        for(w=0; w<vertexQuantity; w++)
                                if((v!=u) & (u!=w) & floydMatrix[v][u] < INFINITO & floydMatrix[u][w] < INFINITO)
                                        if(floydMatrix[v][w] > (floydMatrix[v][u] + floydMatrix[u][w]))
                                                floydMatrix[v][w]=floydMatrix[v][u]+floydMatrix[u][w];
        return floydMatrix;
        }
```

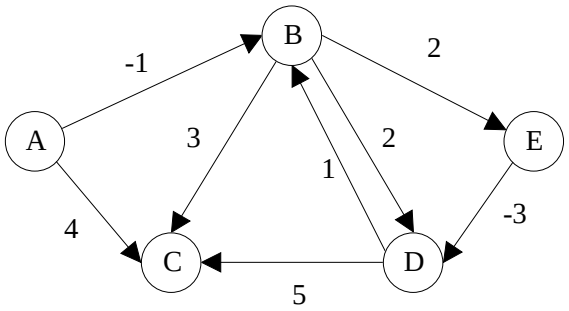
5.2.3.- Algoritmo de Bellman-Ford (camino mínimo/máximo)

Soluciona el problema de la ruta más corta o camino mínimo desde un nodo origen, de un modo más general que el algoritmo de *Dijkstra*, ya que permite valores negativos en los arcos.
El algoritmo devuelve un valor booleano si encuentra un circuito o lazo de peso negativo. En caso contrario calcula y devuelve el camino mínimo con su coste.
Para cada vértice *v* perteneciente a V, se mantiene el atributo *distancia[v]* como cota superior o coste del camino mínimo desde el origen *o* al vértice *v*.

El pseudocódigo del algoritmo es:

```
booleano Bellman-Ford(o)
        {
        distancia[o]=0
        predecesor[o] = 0
        para(cada v perteneciente a V[G]-{v})
                {
                distancia[v]=infinito
                predecesor[v]=nulo
                }
        para(i de 1 a V[G]-1)
                para(cada arco (u ,v) perteneciente a A[G])
                        si(distancia[v] > distancia[u] + matrizAdya(u, v))
                                {
                                distancia[v] = distancia[u] + matrizAdya(u, v)
                                predecesor[v] = u
                                }
        para (cada arco (u, v) chequea lazo de peso negativo)
                si (distancia[v] > distancia[u] + matrizAdya(u, v))
                        retornar falso //el algoritmo no converge
        retornar verdadero
        }
```

Ejemplo de caminos mínimos con pesos negativos a partir del vértice A:



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

El problema de la ruta más larga puede ser transformado en el de ruta más corta cambiando el signo de los costes de los arcos.
En este caso el problema es inconsistente para circuitos de peso positivo.

5.2.4.- Algoritmo de Ford-Fulkerson (flujo máximo)

Se puede considerar un grafo como una red de flujo. Donde un nodo fuente produce o introduce en la red cierta cantidad de algún tipo de material, y un nodo sumidero lo consume. Cada arco, por lo tanto, puede considerarse como un conducto que tiene cierta capacidad de flujo. De igual modo que en redes eléctricas (Ley de Kirchhoff), la suma de flujos entrantes a un nodo, debe ser igual a la suma de los salientes (principio de conservación de energía), excepto para el nodo fuente y el nodo sumidero.
Por lo tanto, el problema de flujo máximo se enuncia como: ¿cuál es la tasa a la cual se puede transportar el material desde el nodo fuente al nodo sumidero, sin violar las restricciones de capacidad?. Este algoritmo se puede usar para resolver modelos de: transporte de mercancías (logística de aprovisionamiento y distribución), flujo de gases y líquidos por tuberías, componentes o piezas en líneas de montaje, corriente en redes eléctricas, paquetes de información en redes de comunicaciones, tráfico ferroviario, sistema de regadíos, etc.
Una red de flujo es un grafo dirigido G=(V, E) donde cada arco (u, v) perteneciente a E tiene una capacidad no negativa. Se distinguen dos nodos: la fuente o nodo s, y el sumidero o nodo t. Si existen múltiples fuentes y sumideros, el problema se puede simplificar añadiendo una fuente común y un sumidero común.

Este algoritmo depende de tres conceptos principales:

- Un **camino de aumento**, es una trayectoria desde el nodo fuente s al nodo sumidero t que puede conducir más flujo.
- La **capacidad residual** es la capacidad adicional de flujo que un arco puede llevar $c_f(u, v) = c(u, v) - f(u, v)$.
- **Teorema de Ford-Fulkerson (1962):** En cualquier red, el flujo máximo que fluye de la fuente al destino es igual a la capacidad del corte mínimo que separa a la fuente del destino.

Una variación del algoritmo de Ford-Fulkerson es el algoritmo de **Edmonds-Karp** (J. Edmonds; R.M. Karp - 1972). En éste, el 'camino de aumento' es elegido usando una búsqueda por niveles o en anchura (breadth-first search). El algoritmo de Edmonds-Karp requiere $O(n^2)$ tiempo de computación, donde n es el número de nodos o vértices, y el número de arcos del grafo.

5.2.5.- Algoritmo de Kruskal y Prim (árbol de coste total mínimo/máximo)

El objetivo del algoritmo de Kruskal es construir un árbol (subgrafo sin ciclos) formado por arcos sucesivamente seleccionados de mínimo peso a partir de un grafo con pesos en los arcos.
El algoritmo de Prim encuentra un árbol de peso total mínimo conectando nodos o vértices con arcos de peso mínimo del grafo sin formar ciclos.

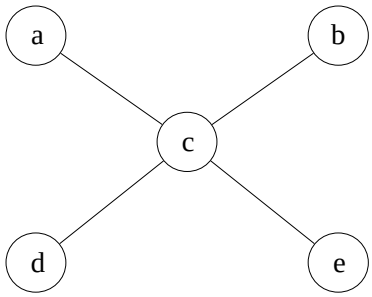
5.3.- Algoritmo de Fleury (recorridos eulerianos)

El algoritmo de Fleury es un algoritmo de búsqueda de caminos eulerianos en grafos. El algoritmo garantiza que si existe un camino euleriano en el grafo lo encuentra. Basa su ejecución en cuatro pasos, partiendo del requisito de que el grafo sea euleriano.

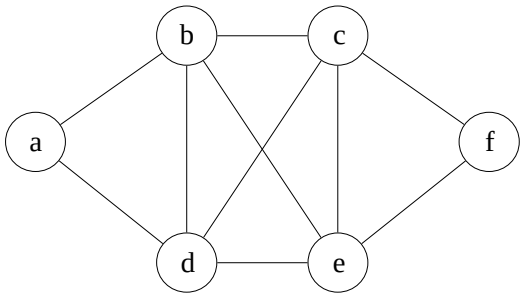
Grafos Eurelianos

Definimos un grafo euleriano G como un grafo conexo que posee un camino cerrado que incluye todas las aristas de G, a la que llamaremos camino euleriano. Cada arista se recorre una vez y sólo una vez. Definimos que G es semi-euleriano si levantamos la restricción de que el camino euleriano deba ser cerrado.

En la figura siguiente se puede observar un grafo no euleriano y otro grafo euleriano y su correspondiente camino:



Grafo no Euleriano

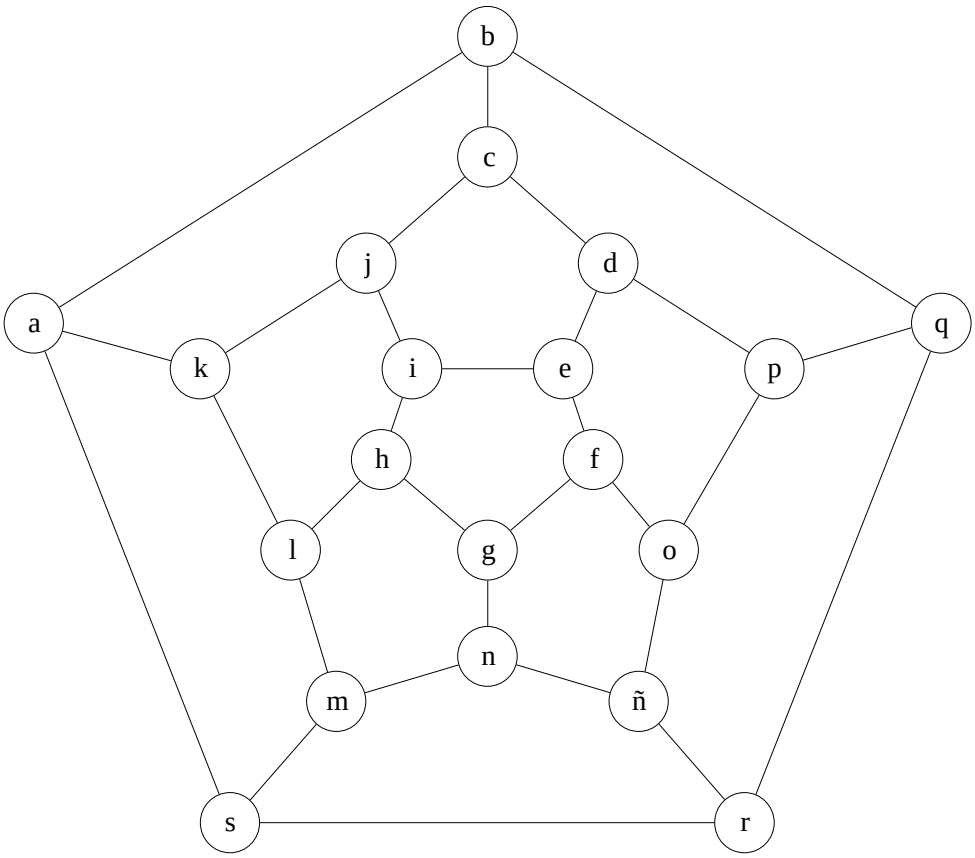


Grafo Euleriano de camino: a d e f c e b c d b a

Grafos Hamiltonianos

En los grafos euleriano se examinaba la posibilidad de crear un camino cerrado que incluyese todas las aristas de un grafo conexo G. Un problema similar es el de sí existe un camino cerrado que pasa exactamente una vez a través de cada vértice de G. Si este existe se dice que G es un grafo hamiltoniano.

A continuación se muestra el grafo hamiltoniano dodecaedrico:



ALGORITMOS DE ORDENACION

1.- INTRODUCCION

Dado un conjunto de *n* elementos a_1, a_2,..., a_n y una relación de orden total (≤) sobre ellos, el problema de la ordenación consiste en encontrar una permutación de esos elementos ordenada de forma creciente.
Aunque tanto el tipo y tamaño de los elementos como el dispositivo en donde se encuentran almacenados pueden influir en el método que utilicemos para ordenarlos, en este tema vamos a solucionar el caso en que los elementos son números enteros y se encuentran almacenados en un arreglo.
Si bien existen distintos criterios para clasificar a los algoritmos de ordenación, una posibilidad es atendiendo a su eficiencia. De esta forma, en función de la complejidad que presentan en el caso medio, podemos establecer la siguiente clasificación:

- $\Theta(n^2)$: Burbuja, Inserción, Selección, ShakerSort.
- $\Theta(n\log n)$: MergeSort, HeapSort, QuickSort.
- Otros: ShellSort $\Theta(n^{1.25})$, BinSort $\Theta(n)$, RadixSort $\Theta(n)$.

Otra clasificación que podemos hacer según su tipo de ordenación, es la siguiente:

- Ordenaciones por intercambio: Burbuja, ShakerSort, QuickSort.
- Ordenaciones por selección: Selección, HeapSort.
- Ordenaciones por inserción: Inserción, ShellSort.
- Ordenaciones por distribución: BinSort, RadixSort.
- Ordenaciones por intercalación: MergeSort.

A continuación, desarrollaremos todos ellos con detenimiento, prestando especial atención a su complejidad, no sólo en el caso medio sino también en los mejores y peores casos, pues para algunos existen diferencias significativas.
Como hemos mencionado anteriormente, nos centraremos en la ordenación de enteros, muchos de los problemas de ordenación que nos encontramos en la práctica son de ordenación de datos más complejos. Sin embargo este problema puede ser fácilmente reducido al de ordenación de números enteros utilizando las claves o bien índices. Por otro lado, puede que los datos a ordenar excedan la capacidad de memoria del ordenador, y por lo tanto, deban residir en dispositivos externos. Aunque este problema, denominado *ordenación externa*, presenta ciertas dificultades específicas, los métodos utilizados para resolverlo se basan fundamentalmente en los algoritmos que aquí presentamos.

Antes de pasar a desarrollar los principales algoritmos, es necesario aclarar que todos los métodos presentados van incluidos en la siguiente clase:

```
public final class Sorts
        {
        }
```

2.- ALGORITMOS BASICOS DE ORDENACION

2.1.- Ordenación por Inserción

El método de Inserción realiza n–1 iteraciones sobre el arreglo, dejando en la i-ésima etapa ($2 \le i \le n$) ordenado el arreglo vec[1...i]. La forma de hacerlo es colocando en cada iteración el elemento vec[i] en su sitio correcto, aprovechando el hecho de que el arreglo vec[1...i–1] ya ha sido previamente ordenado. Este método puede ser implementado de forma iterativa como sigue:

```
public static void insertionSort(int[] vec)
        {
        for(int i=1; i<vec.length; i++)
                {
                int aux=vec[i];
                int j=i-1;
                while(j>=0 && vec[j]>aux)
                        {
                        vec[j+1]=vec[j];
                        j-=1;
                        }
                vec[j+1]=aux;
                }
        }
```

Para estudiar su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al método *insertionSort(vec)*.

- En el mejor caso el bucle interno no se realiza nunca, y por lo tanto:

$$T(n) = \left(\sum_{i=1}^{n-1} (3 + 4 + 4 + 3) \right) + 3 = 14n - 11.$$

- En el peor caso hay que llevar cada elemento hasta su posición final, con lo que el bucle interno se realiza siempre de i–1 veces. Así, en este caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 4 + \left(\sum_{j=0}^{i-1} (4 + 5) \right) + 1 + 3 \right) \right) + 3 = \frac{9}{2} n^2 + \frac{13}{2} n - 10.$$

- En el caso medio, supondremos equiprobable la posición de cada elemento dentro del arreglo. Por lo tanto, para cada valor de i, la probabilidad de que el elemento se sitúe en alguna posición k de las i primeras será de 1/i. El número de veces que se repetirá el bucle while en este caso es (i–k), con lo cual el número medio de operaciones que se realizan en el bucle es:

$$\left(\frac{1}{i} \sum_{k=1}^{i} 9(i - k) \right) + 4 = \frac{9}{2} i - \frac{1}{2}.$$

Por lo tanto, el tiempo de ejecución en el caso medio es:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + 4 + \left(\frac{9}{2} i - \frac{1}{2} \right) + 3 \right) \right) + 3 = \frac{9}{2} n^2 + \frac{47}{4} n - 11.$$

Por el modo en que funciona el algoritmo, tales casos van a corresponder a cuando el arreglo se encuentra ordenado de forma creciente, decreciente o aleatoria.
Como podemos ver, en este método los órdenes de complejidad de los casos peor, mejor y medio difieren bastante. Así, en el mejor caso el orden de complejidad resulta ser lineal, mientras que en los casos peor y medio su complejidad es cuadrática.
Este método se muestra muy adecuado para aquellas situaciones en donde necesitamos ordenar un arreglo del que ya conocemos que está casi ordenado, como suele suceder en aquellas aplicaciones de inserción de elementos en bancos de datos previamente ordenados cuya ordenación total se realiza periódicamente.

2.2.- Ordenación por Selección

En cada paso (i=1...n–1) este método busca el mínimo elemento del arreglo vec[i...n] y lo intercambia con el elemento en la posición i:

```
public static void selectionSort(int[] vec)
        {
        for(int i=0; i<vec.length - 1; i++)
                {
                int pos=i;
                for(int j=i + 1; j<vec.length; j++)
                        if(vec[j]<vec[pos])
                                pos=j;
                interChange(vec, pos, i);
                }
        }
```

En cuanto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al método *selectionSort(vec)*.

- En el mejor caso:

$$T(n) = \left(\sum_{i=0}^{n-2} (3 + 1 + (5 + 6(n - i)) + 1 + 7) \right) + 3 = 3n^2 + 14n - 14.$$

- En el peor caso:

$$T(n) = \left(\sum_{i=0}^{n-2} (3 + 1 + (5 + 7(n - i)) + 1 + 7) \right) + 3 = \frac{7}{2} n^2 + \frac{27}{2} n - 14.$$

- En el caso medio:

$$T(n) = \left(\sum_{i=0}^{n-2} \left(3 + 1 + \left(5 + \frac{13}{2} (n - i) \right) + 1 + 7 \right) \right) + 3 = \frac{13}{4} n^2 + \frac{55}{4} n - 14.$$

En consecuencia, el algoritmo es de complejidad cuadrática.
Este método, por el número de operaciones de comparación e intercambio que realiza, es el más adecuado para ordenar pocos registros de gran tamaño. Si el tipo base del arreglo a ordenar no es entero, sino un tipo

más complejo (guías telefónicas, índices de libros, historiales hospitalarios, etc.) deberemos darle mayor importancia al intercambio de valores que a la comparación entre ellos en la valoración del algoritmo por el coste que suponen. En este sentido, analizando el número de intercambios que realiza el método de Selección vemos que es de orden O(n), frente al orden $O(n^2)$ de intercambios que presentan los métodos de Inserción o Intercambio (Burbuja).

2.3.- Ordenación por Intercambio (Burbuja)

Este método de ordenación consiste en recorrer los elementos siempre en la misma dirección, intercambiando elementos adyacentes si fuera necesario:

```
public static void bubbleSort(int[] vec)
        {
        for(int i=vec.length - 1; i>=1; i--)
                for(int j=0; j<=i - 1; j++)
                        if(vec[j]>vec[j + 1])
                                interChange(vec, j, j + 1);
        }
```

El nombre de este algoritmo trata de reflejar cómo el elemento mínimo "sube", a modo de burbuja, hasta el principio del arreglo.
Respecto a su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al método *bubbleSort(vec)*.

- En el mejor caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=0}^{i-1} (3 + 4) + 3 \right) \right) + 3 = \frac{7}{2} n^2 + \frac{5}{2} n - 3.$$

- En el peor caso:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=0}^{i-1} (3 + 4 + 2 + 7) + 3 \right) \right) + 3 = 8n^2 - 2n - 1.$$

- En el caso medio:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(3 + \sum_{j=0}^{i-1} \left(3 + 4 + \frac{2 + 7}{2} \right) + 3 \right) \right) + 3 = \frac{23}{4} n^2 + \frac{1}{4} n - 1.$$

En consecuencia, el algoritmo es de complejidad cuadrática.
Este algoritmo funciona de forma parecida al de Selección, pero haciendo más trabajo para llevar cada elemento a su posición. De hecho es el peor de los tres vistos hasta ahora, no sólo en cuanto al tiempo de ejecución, sino también respecto al número de comparaciones y de intercambios que realiza.
Una posible mejora que puede admitir este algoritmo es el control de la existencia de una pasada sin intercambios; en ese momento el arreglo estará ordenado.

2.4.- Ordenación por Intercambio (Burbuja Mejorado)

```
public static void bubbleImprovedSort(int[] vec)
        {
        int i=vec.length - 1;
        boolean ordered;
        do
                {
                ordered=true;
                for(int j=0; j<=i - 1; j++)
                        if(vec[j]>vec[j+1])
                                {
                                interChange(vec, j, j + 1);
                                ordered=false;
                                }
                i--;
                }
        while(i>=1 & !ordered);
        }
```

Dejamos el análisis del algoritmo para que se investigue.

3.- ALGORITMOS AVANZADOS DE ORDENACION

3.1.- Ordenación por Mezcla (MERGESORT)

Este método utiliza la técnica de "Divide y Vencerás" para realizar la ordenación del arreglo. Su estrategia consiste en dividir el arreglo en dos arreglos, ordenarlos mediante llamadas recursivas, y finalmente combinar los dos arreglos ya ordenados. Esta idea da lugar a la siguiente implementación:

```
public static void mergeSort(int[] vec)
        {
        int[] tmp=new int[vec.length];
        mergeSort(vec, tmp, 0, vec.length-1);
        }

private static void mergeSort(int[] vec, int[] tmp, int left, int right)
        {
        if(left<right)
                {
                int center=(right + left) / 2;
                mergeSort(vec, tmp, left, center);
                mergeSort(vec, tmp, center+1, right);
                merge(vec, tmp, left, center, right);
                }
        }

private static void merge(int[] vec, int[] tmp, int left, int center, int right)
        {
        int aptr=left;
        int bptr=center+1;
        int cptr=left;

        while(aptr<=center & bptr<=right)
                if(vec[aptr]<vec[bptr])
                        tmp[cptr++]=vec[aptr++];
                else
                        tmp[cptr++]=vec[bptr++];

        while(aptr<=center)
                tmp[cptr++]=vec[aptr++];

        while(bptr<=right)
                tmp[cptr++]=vec[bptr++];

        for(int i=left; i<=right; i++)
                vec[i]=tmp[i];
        }
```

Una posible implementación del método que lleva a cabo el proceso de mezcla vuelca primero los elementos a ordenar en el arreglo auxiliar para después, utilizando dos índices, uno para cada arreglo, rellenar el arreglo ordenadamente. Nótese que el algoritmo *merge* utiliza el hecho de que los dos arreglos están ya ordenados y que son además consecutivos.
En cuanto al estudio de su complejidad, siguiendo la metodología aplicada anteriormente, se llega a que el tiempo de ejecución de *mergeSort(vec)* puede expresarse mediante una ecuación en recurrencia:

$$T(n) = 2T(n/2) + 16n + 17$$

con la condición inicial T(1) = 1. Ésta es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x–2)^2 (x–1) = 0$, lo que permite expresar T(n) como:

$$T(n) = c_1 n + c_2 n\log n + c_3.$$

El cálculo de las constantes puede hacerse en base a la condición inicial, lo que nos lleva a la expresión final:

$$T(n) = 16n\log n + 18n – 17 \in \Theta(n\log n).$$

Obsérvese que este método ordena *n* elementos en tiempo Θ(nlogn) en cualquiera de los casos (peor, mejor o medio). Sin embargo tiene una complejidad espacial, en cuanto a memoria, mayor que los demás (del orden de n).

Otras versiones de este algoritmo no utilizan el arreglo auxiliar, sino que trabajan sobre el propio arreglo a ordenar, combinando sobre él los arreglos obtenidos de las etapas anteriores. Si bien es cierto que esto consigue ahorrar espacio (un arreglo auxiliar), también complica el código del algoritmo resultante.
El método de ordenación por Mezcla se adapta muy bien a distintas circunstancias, por lo que es comúnmente utilizado no sólo para la ordenación de arreglos. Por ejemplo, el método puede ser también implementado de forma que el acceso a los datos se realice de forma secuencial, por lo que hay diversas estructuras (como las listas enlazadas) para las que es especialmente apropiado. También se utiliza para realizar ordenación externa, en donde el arreglo a ordenar reside en dispositivos externos de acceso secuencial (ficheros).

3.2.- Ordenación mediante Montículos (HEAPSORT)

La filosofía de este método de ordenación consiste en aprovechar la estructura particular de los montículos (heaps), que son árboles binarios completos (todos sus niveles están llenos salvo a lo sumo el último, que se rellena de izquierda a derecha) y cuyos nodos verifican la propiedad del montículo: todo nodo es mayor o igual que cualquiera de sus hijos. En consecuencia, en la raíz se encuentra siempre el elemento mayor.
Estas estructuras admiten una representación muy sencilla, compacta y eficiente mediante arreglos (por ser árboles completos). Así, en un arreglo que represente una implementación de un montículo se cumple que el "padre" del i-ésimo elemento del arreglo se encuentra en la posición i÷2 (menos la raíz, claro) y sus "hijos", si es que los tiene, estarán en las posiciones 2i y 2i+1 respectivamente.
La idea es construir, con los elementos a ordenar, un montículo sobre el propio arreglo. Una vez construido el montículo, su elemento mayor se encuentra en la primera posición del arreglo. Se intercambia entonces con el último y se repite el proceso para el arreglo con la primera posición + 1 y la última posición - 1. Así sucesivamente hasta recorrer el arreglo completo. Esto nos lleva a un algoritmo de orden de complejidad O(nlogn) cuya implementación puede ser la siguiente:

```
public static void heapSort(int[] vec)
        {
        int index;
```

```
        for(index=vec.length - 1; index>=0; index--)
                reHeapDown(vec, index, vec.length);
        for(index=vec.length - 1; index>0; index--)
                {
                interChange(vec, 0, index);
                reHeapDown(vec, 0, index);
                }
        }

private static void reHeapDown(int vec[], int length, int index)
        {
        boolean done=false;
        int aux=vec[length];
        int parent=length;
        int child=2 * (length + 1) - 1;

        while(child<index & !done)
                {
                if(child<index - 1)
                        if(vec[child]<vec[child + 1])
                                child++;
                if(aux>=vec[child])
                        done=true;
                else
                        {
                        vec[parent]=vec[child];
                        parent=child;
                        child=2 * (parent + 1) - 1;
                        }
                }
        vec[parent]=aux;
        }
```

Los métodos *heapSort* y *reHeapDown* son, respectivamente, el que construye un montículo a partir del arreglo dado, y el que "empuja" un elemento hasta su posición definitiva en el montículo, reconstruyendo la estructura de montículo en el arreglo.
Para estudiar la complejidad del algoritmo hemos de considerar dos partes. La primera es la que construye inicialmente el montículo a partir de los elementos a ordenar y la segunda va recorriendo en cada iteración un arreglo más pequeño, colocando el elemento raíz en su posición correcta dentro del montículo. En ambos casos nos basamos en el método que "empuja" elementos en el montículo. Observando el comportamiento del algoritmo, la diferencia básica entre el caso peor y el mejor está en la profundidad que hay que recorrer cada vez que necesitamos "empujar" un elemento. Podríamos llegar a tener que recorrer todo el árbol (profundidad: logn) dependiendo del elemento a "empujar", respecto al resto.
El método *heapSort* es de complejidad O(n) en el peor caso, puesto que si k es la altura del montículo (k = logn), el algoritmo transforma primero cada uno de los dos subárboles que cuelgan de la raíz en montículos de altura a lo sumo k–1 (el subárbol derecho puede tener altura k–2), y después empuja la raíz hacia abajo, por un camino que a lo sumo es de longitud k. Esto lleva a lo sumo un tiempo t(k) de orden de complejidad O(k) con lo cual

$$T(k) = 2T(k-1) + t(k),$$

ecuación en recurrencia cuya solución verifica que $T(k) \in (2^k)$. Como k = logn, la complejidad de *heapSort* es lineal en el peor caso. Este caso ocurre cuando hay que recorrer siempre la máxima profundidad al empujar a cada elemento, lo que sucede si el arreglo está originalmente ordenado de forma creciente.
Respecto al mejor caso de *heapSort*, éste se presenta cuando la profundidad a la que hay que empujar cada elemento es cero. Esto se da, por ejemplo, si todos los elementos del arreglo son iguales. En esta situación la complejidad del algoritmo es O(1).
Estudiemos ahora los casos mejor y peor del resto del algoritmo *heapSort*. En esta parte hay un bucle que se ejecuta siempre n–1 veces, y la complejidad del método que intercambia dos elementos es O(1). Todo va a depender del método *reHeapDown*, es decir, de la profundidad a la que haya que empujar la raíz del

montículo en cada iteración, sabiendo que cada montículo tiene n–i elementos, y por lo tanto, una altura de log(n–i), siendo i el número de la iteración.
En el peor caso, la profundidad a la que hay que empujar las raíces respectivas es la máxima, y por lo tanto, la complejidad de esta segunda parte del algoritmo es O(nlogn). ¿Cuándo ocurre esto?... cuando el elemento es menor que todos los demás. Pero esto sucede siempre que los elementos a ordenar sean distintos, por la forma en la que se van escogiendo las nuevas raíces.
En el mejor caso, aunque el bucle se sigue repitiendo n–1 veces, las raíces no descienden, por ser mayores o iguales que el resto de los elementos del montículo. Así, la complejidad de esta parte del algoritmo es de orden O(n). Pero este caso sólo se dará si los elementos del arreglo son iguales, por la forma en la que originariamente se construyó el montículo y por cómo se escoge la nueva raíz en cada iteración (el último de los elementos, que en un montículo ha de ser de los menores).

3.3.- Ordenación Rápida de Hoare (QUICKSORT)

Este método es probablemente el algoritmo de ordenación más utilizado, pues es muy fácil de implementar, trabaja bien en casi todas las situaciones y consume en general menos recursos (memoria y tiempo) que otros métodos.
Su diseño está basado en la técnica de “Divide y Vencerás”, que estudiaremos a continuación, y consta de dos partes:

a) En primer lugar el arreglo a ordenar es dividido en dos arreglos no vacíos, tal que todos los elementos del primero son menores que los del segundo. Forma parte de esto, un pivote (elemento dentro del conjunto) como protagonista esencial en el método de partición.
b) A continuación, los dos arreglos son ordenados mediante llamadas recursivas al método quickSort. Como los arreglos se ordenan sobre ellos mismos, no es necesario realizar ninguna operación de combinación.

El siguiente método constituye la versión clásica del algoritmo de ordenación rápida de Hoare:

```
public static void quickSort(int[] vec)
        {
        quickSort(vec, 0, vec.length-1);
        }

private static void quickSort(int[] vec, int first, int last)
        {
        int center=(first + last) / 2;
        if(first<last)
                {
                center=division(vec, first, last, center);
                quickSort(vec, first, center);
                if(center= =first)
                        quickSort(vec, center + 1, last);
                else
                        quickSort(vec, center, last);
                }
        }

private static int division(int[] vec, int first, int last, int center)
        {
        int left, right, data;
        data=vec[center];
        left=first;
        right=last;
        do
                {
                while(left<right && vec[left]<data)
                        left++;
                while(left<right && vec[right]>data)
                        right--;
```

```
            if(left<right)
                    {
                    interChange(vec, left, right);
                    left++;
                    right--;
                    }
            }
while(left<right);
if(left<right)
            {
            int pos=right;
            right=left;
            left=pos;
            }

return left;
}
```

El método *division* parte del elemento pivote y permuta los elementos del arreglo de forma que al finalizar el método, todos los elementos menores o iguales que el pivote estén a su izquierda, y los elementos mayores que él a su derecha. Devuelve la posición en la que ha quedado situado el pivote.
Este método es de orden de complejidad $\Theta(n^2)$ en el peor caso y $\Theta(n\log n)$ en los casos mejor y medio. Obtenemos la siguiente ecuación en recurrencia:

$$T(n) = 8 + T(a) + T(b) + T_{Pivote}(n)$$

donde a y b son los tamaños en los que el método *division* divide al arreglo (por lo tanto, podemos tomar que a + b = n), y $T_{Pivote}(n)$ es el método que define el tiempo de ejecución del método *division*.
El método *quickSort* "rompe" la filosofía de mejor caso, peor y medio de los algoritmos clásicos de ordenación, pues aquí tales casos no dependen de la ordenación inicial del arreglo, sino de la elección del pivote.
Así, el mejor caso ocurre cuando a = b = n/2 en todas las invocaciones recursivas del método, pues en este caso obtenemos $T_{Pivote}(n) = 13 + 4n$, y por lo tanto:

$$T(n) = 21 + 4n + 2T(n/2).$$

Resolviendo esta ecuación en recurrencia y teniendo en cuenta las condiciones iniciales T(0) = 1 y T(1) = 27 se obtiene la expresión final de T(n), en este caso:

$$T(n) = 15n\log n + 26n + 1.$$

Ahora bien, si a = 0 y b = n–1 (o viceversa) en todas las invocaciones recursivas del método, $T_{Pivote}(n) = 11 + 39/8n$, obteniendo:

$$T(n) = 19 + 39/8n + T(n–1).$$

Resolviendo la ecuación para las mismas condiciones iniciales, nos encontramos con una desagradable sorpresa:

$$T(n) = \frac{3}{8} n^2 + \frac{213}{8} n + 1 \in \Theta(n^2).$$

En consecuencia, la elección idónea para el pivote es la mediana del arreglo en cada etapa, lo que ocurre es que encontrarla requiere un tiempo extra que hace que el algoritmo se vuelva más ineficiente en la mayoría de los casos. Por esa razón como pivote suele escogerse un elemento cualquiera, a menos que se conozca la

naturaleza de los elementos a ordenar. En nuestro caso, como a priori suponemos equiprobable cualquier ordenación inicial del arreglo, hemos escogido el elemento central del arreglo.
Esta elección lleva a tres casos desfavorables para el algoritmo: cuando los elementos son todos iguales y cuando el arreglo está inicialmente ordenado en orden creciente o decreciente. En estos casos la complejidad es cuadrática puesto que la partición se realiza de forma totalmente descompensada.
A pesar de ello suele ser el algoritmo más utilizado, y se demuestra que su tiempo promedio es menor, en una cantidad constante, al de todos los algoritmos de ordenación de complejidad O(nlogn). En todo esto es importante hacer notar, como hemos indicado antes, la relevancia que toma una buena elección del pivote, pues de su elección depende considerablemente el tiempo de ejecución del algoritmo.
Sobre el algoritmo expuesto anteriormente pueden realizarse varias mejoras:
1. Respecto a la elección del pivote. En vez de tomar como pivote el elemento central, puede seguirse alguna estrategia del tipo:

- Tomar al azar tres elementos seguidos del arreglo y escoger como pivote el elemento medio de los tres.
- Tomar k elementos al azar, clasificarlos por cualquier método, y elegir el elemento medio como pivote.

2. Con respecto al tamaño de los arreglos a ordenar. Cuando el tamaño de éstos sea pequeño (menor que una cota dada), es posible utilizar otro algoritmo de ordenación en vez de invocar recursivamente a *quickSort*. Esta idea utiliza el hecho de que algunos métodos, como Selección o Inserción, se comportan muy bien cuando el número de datos a ordenar son pocos, por disponer de constantes multiplicativas pequeñas. Aun siendo de orden de complejidad cuadrática, son más eficientes que los de complejidad nlogn para valores pequeños de n.

3.4.- Ordenación por Incrementos (SHELLSORT)

Conocida como disminución incremental y nombrada así debido a su inventor Donald Shell.
La ordenación por intercambio puede resultar lenta pues sólo intercambia elementos adyacentes. Así, si por ejemplo el elemento menor está al final del arreglo, hacen falta n pasos para colocarlo donde corresponde. El método de incrementos es una extensión muy simple y eficiente del método de burbuja en el que cada elemento se coloca casi en su posición definitiva en la primera pasada.
El algoritmo consiste básicamente en dividir el arreglo en h arreglos:

vec[k], vec[k + h], vec[k + 2h], vec[k + 3h], ...

y ordenar cada uno de esos arreglos (k=1,2,...,h–1).
Un arreglo de esta forma, es decir, compuesto por h arreglos ordenados intercalados, se denomina h-ordenado. Haciendo h-ordenaciones de vec para valores grandes de h permitimos que los elementos puedan moverse grandes distancias dentro del arreglo, facilitando así las h-ordenaciones para valores más pequeños de h. A h se le denomina incremento.
Con esto, el método de ordenación por incrementos consiste en hacer h-ordenaciones de vec para valores de h decreciendo hasta llegar a uno.
El número de comparaciones que se realizan en este algoritmo va a depender de la secuencia de incrementos h, y será mayor que en el método clásico de burbuja (que se ejecuta finalmente para h = 1), pero la potencia de este método consiste en conseguir un número de intercambios mucho menor que con la inserción clásica.
El método presentado a continuación utiliza la secuencia de incrementos h = ..., 1093, 364, 121, 40, 13, 1. Otras secuencias pueden ser utilizadas, pero la elección ha de hacerse con cuidado. Por ejemplo la secuencia ..., 64, 32, 16, 8, 4, 2, 1 es muy ineficiente pues los elementos en posiciones pares e impares no son comparados hasta el último momento.

```
public static void shellSort(int[] vec)
        {
        int interval=vec.length / 2;
```

```
while(interval>0)
        {
        for(int i=interval; i<vec.length; i++)
                {
                int j=i - interval;
                while(j>=0)
                        {
                        if(vec[j]<=vec[j + interval])
                                j=0;
                        else
                                interChange(vec, j, j + interval);
                        j-=interval;
                        }
                }
        interval/=2;
        }
}
```

En cuanto al estudio de su complejidad, este método es diferente al resto de los métodos vistos en este apunte. Su complejidad es difícil de calcular y depende mucho de la secuencia de incrementos que utilice. Por ejemplo, para la secuencia dada existen dos conjeturas en cuanto a su orden de complejidad: $nlog^2n$ y $n^{1.25}$. En general este método es el escogido para muchas aplicaciones reales por ser muy simple teniendo un tiempo de ejecución aceptable incluso para grandes valores de n.

3.5.- Ordenación por Sacudida o Vibración (SHAKERSORT)

La idea básica de este algoritmo consiste en mezclar las dos formas en que se puede realizar el método de la burbuja. En este algoritmo cada pasada tiene dos etapas. En la primera etapa “de derecha a izquierda” se trasladan los elementos más pequeños hacia la parte izquierda del arreglo, almacenando en una variable la posición del último elemento intercambiado. En la segunda etapa “de izquierda a derecha” se trasladan los elementos más grandes hacia la parte derecha del arreglo, almacenando en otra variable la posición del último elemento intercambiado. Las sucesivas pasadas trabajan con los componentes del arreglo comprendidos entre las posiciones almacenadas en las variables. El algoritmo termina cuando en una etapa no se producen intercambios, o bien cuando el contenido de la variable que almacena el extremo izquierdo del arreglo es mayor que el contenido de la variable que almacena el extremo derecho.
Este algoritmo tiene la ventaja que reduce considerablemente el número de comparaciones cuando los elementos están casi ordenados (se va directamente al elemento que falta ordenar).

```
public static void shakerSort(int[] vec)
        {
        int left=1;
        int right=vec.length - 1;
        int aux=vec.length - 1;

        do
                {
                for(int i=right; i>=left; i--)
                        if(vec[i - 1]>vec[i])
                                {
                                interChange(vec, i - 1, i);
                                aux=i;
                                }
                left=aux + 1;
                for(int i=left; i<=right; i++)
                        if(vec[i - 1]>vec[i])
                                {
                                interChange(vec, i - 1, i);
                                aux=i;
                                }
                right=aux - 1;
                }
        while(left<right);
        }
```

El análisis del método por sacudida y en general el de los métodos mejorados y logarítmicos es muy complejo. Para el análisis de este método es necesario tener en cuenta tres factores que afectan directamente al tiempo de ejecución del algoritmo: las comparaciones entre los elementos, los intercambios entre los mismos y las pasadas que se realizan. Encontrar fórmulas que permitan calcular cada uno de estos factores es una tarea muy difícil de realizar.
Los estudios que se han efectuado sobre el método por sacudida demuestra que en el mismo, sólo pueden reducirse las dobles comparaciones entre elementos; pero debe recordarse que la operación de intercambio es una tarea más complicada y costosa que la de comparación. Por lo tanto, es posible afirmar que las hábiles mejoras realizadas sobre el método de intercambio sólo producen resultados apreciables si el arreglo está parcialmente ordenado (lo cual resulta difícil saber de antemano), pero si el arreglo está desordenado el método se comporta, incluso, peor que otros métodos directos como los de inserción y selección.
Algo muy interesante para tener en cuenta es que recientemente Brona Brejova (especialista en Ciencias de la Computación de Canadá) probó que shakersort tiene un orden de complejidad O(nlogn) en el peor caso para ciertas secuencias de incrementos (el primer salto más alto no cuadrático del peor caso). Usando un método de incomprensibilidad, también probó el salto más chico sobre tiempos de ejecución en el caso medio.

4.- OTROS ALGORITMOS AVANZADOS DE ORDENACION

Los algoritmos vistos hasta ahora se basan en la ordenación de arreglos de números enteros, sin ningún tipo de restricción. A continuación veremos cómo pueden encontrarse algoritmos de orden O(n) cuando dispongamos de información adicional sobre los valores a ordenar. Además, como lograr ordenar sin la necesidad de comparaciones e intercambios.

4.1.- Ordenación por Urnas (BINSORT)

Suponemos que los datos a ordenar son números naturales, todos distintos y comprendidos en el intervalo [1, n]. Es decir, nuestro problema es ordenar un arreglo con los n primeros números naturales. Bajo esas circunstancias es posible implementar un algoritmo de complejidad temporal O(n). Es el método de ordenación por Urnas, en donde en cada iteración se sitúa un elemento en su posición definitiva:

```
public static void binSort(int[] vec)
        {
        //crea la urna
        Queue[] urn=new Queue[/*elemento más grande en el arreglo más 1*/];
        for(int i=0; i<urn.length; i++)
                urn[i]=new Queue();

        //guarda cada elemento del arreglo en su correspondiente urna
        for(int i=0; i<vec.length; i++)
                urn[getKey(vec[i])].insert(new Integer(vec[i]));

        //recupera cada elemento de las distintas urnas y lo guarda en el arreglo
        int j=0;
        for(int i=0; i<urn.length; i++)
                while(!urn[i].isEmpty())
                        {
                        vec[j]=((Integer)urn[i].delete()).intValue();
                        j++;
                        }
        }
```

4.2.- Ordenación por Residuos (RADIXSORT)

Este método puede utilizarse cuando los valores a ordenar están compuestos por secuencias de letras o dígitos que admiten un orden lexicográfico. Éste es el caso de palabras, números (cuyos dígitos admiten este orden) o bien fechas.

El método consiste en definir k colas (numeradas de 0 a k–1) siendo k los posibles valores que puede tomar cada uno de los dígitos que componen la secuencia. Una vez que tengamos las colas habría que repetir, para i a partir de 0 y hasta llegar al número máximo de dígitos o letras de nuestras cadenas:
1. Distribuir los elementos en las colas en función del dígito i.
2. Extraer ordenada y consecutivamente los elementos de las colas, introduciéndolos de nuevo en el arreglo.
Veamos un ejemplo de este método. Supongamos el arreglo:

[0, 1, 81, 64, 23, 27, 4, 25, 36, 16, 9, 49].

En este caso se trata de números naturales en base 10, que no son sino secuencias de dígitos. Como cada uno de los dígitos puede tomar 10 valores (del 0 al 9), necesitaremos 10 colas. En la primera pasada introducimos los elementos en las colas de acuerdo a su dígito menos significativo:

	0	81 1		23	4 64	25	16 36	27		49 9
Colas	0	1	2	3	4	5	6	7	8	9

y ahora extraemos ordenada y sucesivamente los valores, obteniendo el arreglo:

[0, 1, 81, 23, 64, 4, 25, 36, 16, 27, 9, 49].

Volvemos a realizar otra pasada, esta vez fijándonos en el segundo dígito menos significativo:

	0	16	23 25 27	36	49 4 9	5	64 6	7	81 8	
Colas	0	1	2	3	4	5	6	7	8	9

Volviendo a extraer ordenada y sucesivamente los valores obtenemos el arreglo:

[0, 1, 4, 9, 16, 23, 25, 27, 36, 49, 64, 81].

Como el máximo de dígitos de los números a ordenar era de dos, con dos pasadas hemos tenido suficiente.
La implementación de este método es el siguiente:

```
//Parte del código de este algoritmo es similar al de binSort.
//Por lo tanto, sería óptimo que se reutilice código.
public static void radixSort(int vec[])
	{
	for(int d=1; d<=/*cantidad de dígitos del máximo elemento en el arreglo*/; d++)
		{
		//crea la urna
		Queue[] urn=new Queue[10];
		for(int i=0; i<urn.length; i++)
			urn[i]=new Queue();

		//guarda cada elemento del arreglo en su correspondiente urna
		for(int i=0; i<vec.length; i++)
			urn[getKey(vec[i], d)].insert(new Integer(vec[i]));

		//recupera cada elemento de las distintas urnas y lo guarda en el arreglo
		int j=0;
		for(int i=0; i<urn.length; i++)
			while(!urn[i].isEmpty())
				{
				vec[j]=((Integer)urn[i].delete()).intValue();
				j++;
				}
		}
	}
```

ALGORITMOS DE BUSQUEDA

1.- INTRODUCCION

Buscar en un arreglo un determinado valor significa localizar un elemento del arreglo cuyo contenido coincida con él. Una vez finalizada la búsqueda puede suceder:

- que la búsqueda haya tenido éxito, habiendo localizado la posición donde estaba almacenado el elemento buscado, o
- que la búsqueda no haya tenido éxito, concluyéndose que no existía ningún elemento a buscar.

Las búsquedas son independientes del estado del arreglo. Es evidente que se puede buscar en arreglos ordenados como en arreglos desordenados, utilizando diferentes algoritmos de búsqueda.

Antes de pasar a desarrollar los principales algoritmos, es necesario aclarar que todos los métodos presentados van incluidos en la siguiente clase:

```
public final class Searchs
        {
        }
```

2.- ALGORITMOS BASICOS DE BUSQUEDA

2.1.- Búsqueda Secuencial

Es la forma más simple de buscar un elemento y consiste en examinar secuencialmente uno a uno hasta encontrar el elemento buscado o haber revisado todos los elementos sin éxito. Si tenemos un arreglo de enteros y un elemento que tratamos de localizar, un algoritmo simple que devuelve la posición del elemento en el arreglo, es éste se encuentra en el mismo, o -1 si no se encuentra, es el siguiente:

```
public static int sequentialSearch(int[] vec, int search)
        {
        int pos=0;
        while(pos<vec.length)
                {
                if(vec[pos]= =search)
                        return pos;
                pos++;
                }
        return -1;
        }
```

En el caso de que pudiese haber 2 o más ocurrencias del mismo valor, se encuentra la primera de ellas. Sin embargo, es posible modificar el algoritmo para obtener todas las ocurrencias del dato buscado.

Para estudiar su complejidad, vamos a estudiar los casos mejor, peor y medio de la llamada al método *sequentialSearch(int[], int).*

- En el mejor caso el algoritmo de búsqueda Secuencial termina tan pronto como encuentra el elemento buscado en el arreglo. Si tenemos mucha suerte, puede ser que la primera posición examinada contenga el elemento que buscamos, en cuyo caso el algoritmo informará que tuvo éxito después de una sola comparación. Por lo tanto, su complejidad será O(1).

- En el peor caso sucede cuando encontramos el elemento a buscar en la última posición del arreglo o cuando no se encuentra el elemento en el arreglo. Como se requieren n ejecuciones del bucle mientras, la cantidad de tiempo es proporcional a la longitud del arreglo n, más un cierto tiempo para realizar las condiciones del bucle mientras y para la llamada al método. Por lo tanto, la cantidad de tiempo es de la forma an + b para ciertas constantes a y b. En notación O, O(an + b) = O(an) = O(n).

- En el caso medio, supongamos que cada elemento almacenado en el arreglo es igualmente probable que sea buscado. La media se puede calcular tomando el tiempo total de encontrar todos los elementos y dividiéndolo por n:

$$Total = a (1 + 2 + ...+ n) + bn = a (n(n + 1) / 2) + bn$$
$$Media = (Total / n) = a((n + 1) / 2) + b \text{ que es } O(n).$$

2.2.- Búsqueda Binaria

Si los elementos sobre los que se realiza la búsqueda están ordenados, entonces podemos utilizar un algoritmo de búsqueda mucho más rápido que el Secuencial, la búsqueda Binaria. Consiste en reducir paulatinamente el ámbito de búsqueda a la mitad de los elementos, basándose en comparar el elemento a buscar con el elemento que se encuentra en la mitad del intervalo y en base a esta comparación:

1. Si el elemento buscado es **menor** que el elemento medio, entonces sabemos que el elemento está en la mitad inferior del arreglo.
2. Si es **mayor** es porque el elemento está en la mitad superior.
3. Si es **igual** se finaliza con éxito la búsqueda ya que se ha encontrado el elemento.

La implementación de este algoritmo, se muestra a continuación:

```
public static int binarySearch(int[] vec, int search)
        {
        int first=0;
        int mid;
        int last=vec.length - 1;
        while(first<=last)
                {
                mid=(first + last) / 2;
                if(vec[mid]= =search)
                        return mid;
                else
                        if(vec[mid]>search)
                                last=mid - 1;
                        else
                                first=mid + 1;
                }
        return -1;
        }
```

Para poder medir la velocidad de cálculo del algoritmo de búsqueda Binaria, se deberán obtener el número de comparaciones que realiza el algoritmo, es decir, el número de vueltas del ciclo o el número de recursiones. Aunque en principio puede parecer que ambas versiones invierten el mismo tiempo, la recursiva es más lenta a medida que se incrementa el número de elementos, ya que existirán más llamadas a la función por resolver, con el consiguiente gasto de tiempo de guardar y restaurar parámetros.
En el mejor caso, la búsqueda Binaria podría toparse con el elemento buscado en el primer punto medio, requiriéndose sólo una comparación de elementos. Esto equivale al caso óptimo durante una búsqueda Secuencial, pero en el peor de los casos la búsqueda Binaria es mucho más rápida cuando N es grande.
El algoritmo de búsqueda Binaria progresivamente va disminuyendo el número de elementos sobre el que realizar la búsqueda a la mitad: n, n/2, n/4, ... Así, tras logn divisiones se habrá localizado el elemento o se tendrá la seguridad de que no estaba.

- En el mejor caso, en sus casos óptimos, tanto la búsqueda Secuencial como la Binaria requieren sólo una comparación; esto significa que sus tiempos de ejecución óptimos no dependen de la cantidad de datos: son constantes y por lo tanto, proporcionales a 1, es decir, son de O(1).
- En el peor caso, la búsqueda Secuencial y la Binaria sí dependen de N. La primera, recorre todo el arreglo, requiriendo un tiempo de O(n); la Binaria divide el arreglo, requiriendo sólo un tiempo O(logn). Lo mismo ocurre para el caso medio.

3.- ALGORITMOS AVANZADOS DE BUSQUEDA

3.1.- Búsqueda por Interpolación

Este algoritmo procede igual que la búsqueda Binaria sólo que el arreglo se va dividiendo acorde a nuestra estimación en dónde el elemento a buscar se encuentra ubicado. Dada una distribución uniforme de elementos, la búsqueda por Interpolación tiene un caso medio de complejidad O(loglogn).

```
public static int interpolationSearch(int[] vec, int search)
        {
        int first=0;
        int mid;
        int last=vec.length - 1;
        while(search>=vec[first] & search<=vec[last])
                {
                mid=first + (int)Math.abs(Math.floor((search - vec[first]) * (last - first) / (vec[last] - vec[first])));
                if(search= =vec[mid])
                        return mid;
                else
                        if(search<vec[mid])
                                last=mid - 1;
                        else
                                first=mid + 1;
                }
        return -1;
        }
```

Si las siguientes condiciones son verdaderas, entonces la búsqueda por Interpolación puede ser mejor que la búsqueda Binaria:

- Cada acceso es muy costoso comparado a la instrucción típica, por ejemplo, el arreglo se encuentra almacenado en el disco y cada comparación requiere un acceso a disco.
- Los datos no sólo se encuentran ordenados sino que también se encuentran medianamente distribuidos uniformemente, por ejemplo, una guía telefónica cumple esas características, en cambio una entrada [1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 355, 1000, 12300...] no.

En esta situación tenemos la intención de gastar más tiempo para hacer un cálculo acertado donde el elemento puede ser (en vez de elegir siempre el punto medio), por ejemplo:

- Un arreglo de 1000 elementos.
- El elemento más pequeño del arreglo es 1.000.
- El elemento más grande del arreglo es 1.000.000.
- Buscamos el elemento con el valor 12.000.
- Luego esperamos encontrar el elemento, más o menos, en la posición 11 (siempre asumiendo que los elementos están uniformemente distribuidos). Esto se expresa con la fórmula:

n = 1.000
search = 12.000

vec	1.000	... ...	1.000.000
	0	?	999
	⇑	⇑	⇑
	first	mid	last

Búsqueda Binaria: $mid = \frac{(first + last)}{2}$; $mid = \frac{(0 + 999)}{2} = 500$

Búsqueda por Interpolación: $mid = First + \frac{(search - vec[first]) * (last - first)}{vec[last] - vec[first]}$;

$$mid = 0 + \frac{(12.000 - 1.000) * (999 - 0)}{1.000.000 - 1.000} = \frac{10.989.000}{999.000} = 11$$

En el análisis, podemos concluir lo siguiente:

- El cálculo es más costoso que el cálculo de la búsqueda Binaria.
- Una iteración puede ser más lenta que la búsqueda Binaria completa.
- Si el costo de estos cálculos es insignificante para el costo de acceso a un elemento, sólo nos interesa el número de iteraciones.
- En el peor caso, cuando los números no se encuentran uniformemente distribuidos, el tiempo de ejecución puede ser lineal y todos los elementos podrían ser examinados.
- Si los elementos son de manera razonable uniformemente distribuidos, el tiempo de ejecución ha sido demostrado como O(loglogn) (aplicar el logaritmo dos veces en la sucesión). Por ejemplo, para n=4 billones, logn es alrededor 32 y loglogn es aproximadamente 5.

3.2.- Búsqueda Fibonacci

En vez de dividir el arreglo por la mitad (búsqueda Binaria), esta implementación divide el arreglo según los números de Fibonacci. Para substraer el número Fibonacci disminuye el tamaño del arreglo de disminuciones. Los números de Fibonacci se definen de la siguiente manera:

F(0) = 0
F(1) = 1
F(n) = F(n - 1) + F(n - 2) (para n>=2)

Veamos el código correspondiente a la búsqueda Fibonacci:

```
//el arreglo debe estar cargado con los números de Fibonacci
public static int fibonacciSearch(int[] vec, int search)
        {
        //for(j=1; fib(j)<n; j++)
        int f1=1;
        int f2=1;
        int faux;
        int mid;
        int length=vec.length - 1;
        while(f1<length)
                {//encuentra a f1 tal que f1>=length
                f1=f1 + f2; //siguiente número Fibonacci
                f2=f1 - f2; //guarda al f1 anterior
                }
        f1=f1 - f2; //encuentra el número Fibonacci más chico
        f2=f2 - f1; //f1=fib(j-2), f2=fib(j-3)
        mid=length - f1 + 1;
        while(search!=vec[mid]) //si no lo encuentra
                if(mid<0 | search>vec[mid])
                        {//busca en la mitad más baja
                        if(f1= =1) //si no lo encuentra retorna -1
                                return -1;
                        mid=mid + f2; //disminuye los números de Fibonacci
                        f1=f1 - f2;
                        f2=f2 - f1;
                        }
                else
                        {//busca en la mitad más alta
                        if(f2= =0) //si no lo encuentra retorna -1
                                return -1;
                        mid=mid - f2; //disminuye los números de Fibonacci
                        faux=f1 - f2; //esta vez, disminuye mas
                        f1=f2; //para el arreglo más chico
                        f2=faux;
                        }
        return mid;
        }
```

El algoritmo de búsqueda de Fibonacci divide el rango de búsqueda en dos partes, de acuerdo a los números de Fibonacci, luego se compara el elemento a buscar con la posición F(n - 2). Si el elemento es más chico, continuamos en la mitad más baja del arreglo y en caso contrario, con la mitad más alta del arreglo.
Al igual que la búsqueda Binaria, en el peor caso, el tiempo de ejecución de la búsqueda Fibonacci es de O(logn).

4.- HASHING

Existe otro método que puede aumentar la velocidad de búsqueda donde los datos no necesitan estar ordenados y esencialmente es independiente del número n. Este método se conoce como transformación de claves (clave-dirección) o hashing. El hashing consiste en convertir el elemento almacenado (numérico o alfanumérico) en una dirección (índice) dentro del arreglo.

El objetivo del *direccionamiento hash* se presentará mediante el siguiente ejemplo:

Supongamos una compañía que maneja un inventario consistente de 100 artículos, donde cada artículo tiene un número identificador único de dos dígitos. La forma más obvia de almacenar esta información es declarando un arreglo del tipo:

TArticulo posee los atributos:

Número articulo;
Descripción articulo;

TAticulo [] Inventario=new TArticulo [100]; //arreglo de 100 elementos.

donde Inventario[i] representa el registro cuyo número de artículo es i. Los números de los artículos son las claves utilizadas como índices del arreglo.
Incluso si la compañía almacena menos de 100 artículos, se puede utilizar la misma estructura para mantener el arreglo (siempre y cuando las claves sigan siendo de dos dígitos). Aun cuando muchas posiciones en Inventario estén vacías, esta pérdida es compensada con la ventaja de los accesos directos a cada uno de los artículos existentes en el arreglo Inventario.
Pero este sistema no es siempre práctico. Supongamos que la compañía tiene un inventario de no más de mil artículos, y que el Número de artículo es un número de siete dígitos para cada artículo. Para seguir utilizando direccionamiento directo se requeriría un arreglo de 10 millones de elementos. Esta pérdida de espacio es inaceptable, ya que es muy poco probable que la compañía almacene más de mil artículos. Haría falta algún método de convertir una clave dentro de un rango limitado, de manera que idealmente no hubiera dos claves convertidas al mismo entero.
Siguiendo con el ejemplo y suponiendo que la compañía tiene menos de mil artículos, y que existe solamente un registro por cada artículo, está claro que con un arreglo de 1000 elementos es suficiente para contener toda esa información. Si el arreglo está indexado mediante un entero entre 0 y 999 (ambos inclusive), se pueden utilizar los últimos tres dígitos del número de artículo como índice para el registro del artículo correspondiente en el arreglo.
Utilizando esta técnica, dos claves que están relativamente muy cerca por número de artículo, como 4618396 (Inventario[396]) y 4618996 (Inventario[996]), pueden estar muy separadas la una de la otra en el arreglo, comparadas con dos claves que están separadas numéricamente como 0000991 (Inventario[991]) y 9846995 (Inventario[995]). Esto es debido a que sólo los últimos tres dígitos de la clave se utilizan para determinar la posición del registro en el arreglo.

La idea general de usar la clave para determinar la dirección del registro es una excelente idea, pero se debe modificar de forma que no se desperdicie tanto espacio. Esta modificación se lleva a cabo mediante una función que transforma una clave en un índice de un arreglo y que se denomina *función de Randomización o Hash*. Si *H* es una función hash y *X* es un elemento a almacenar, entonces *H(X)* es la función hash del elemento y se corresponde con el índice donde se debe colocar *X*. En nuestro ejemplo, la función hash sería *H(X)=X % 1000* (función resto).

Los valores generados por H deben cubrir todo el conjunto de índices del arreglo. Además, el tamaño del arreglo debe ser un poco más grande que el número de elementos que han de ser insertados, aunque queden posiciones del arreglo sin uso.
El método anterior tiene una deficiencia: suponer que dos elementos X e Y son tales que H(X) = H(Y). Entonces, cuando un elemento X entra en el arreglo, éste se inserta en la posición dada por su función Hash, H(X). Pero cuando al elemento Y se le asigna su posición donde va a ser insertado mediante la función hash, resulta que la posición que se obtiene es la misma que la del elemento X. Esta situación se denomina *Randomización o Hashing con colisión o choqu*e.
En el arreglo Inventario se produciría una colisión, por ejemplo, si después de haber insertado en el arreglo el elemento con Número de articulo 4957397, al cual le corresponde la posición 397, se intenta insertar en el arreglo el elemento 0596397, pues la función hash le vuelve a asignar la misma posición 397 a este elemento, y esa posición ya está ocupada.
Una buena función Hash será aquella que:

1) minimice los choques o coincidencias, y;
2) distribuya los elementos uniformemente a través del arreglo.

Esta es la razón por la que el tamaño del arreglo debe ser un poco mayor que el número real de elementos a insertar, pues cuanto más grande sea el rango de la función de randomización, es menos probable que dos claves generen el mismo valor de asignación o hash, es decir, que se asigne una misma posición a más de un elemento.
Habrá que llegar a un compromiso entre *Eficiencia en Espacio-Tiempo*: el dejar espacios vacíos en el arreglo es una deficiencia en cuanto a espacio, mientras que reduce la necesidad de resolver los casos de choque en la asignación, y por lo tanto es más eficiente en términos de tiempo.

4.1.- Métodos de transformación de claves

Los dos criterios principales al seleccionar una función hash H son:

- H deber ser muy fácil y rápida de calcular.
- H debe, en la medida de lo posible, distribuir uniformemente las direcciones hash sobre el conjunto de direcciones del arreglo de forma que haya el menor número de colisiones posible.

Naturalmente, no existe garantía de que la segunda condición se pueda cumplir plenamente sin conocer de antemano las claves y las direcciones. Sin embargo, ciertas técnicas generales prestan una gran ayuda. Una de estas técnicas es la de “trocear” una clave X en pedazos y combinarlos de alguna forma para obtener la dirección hash H(X). (El término “hash” viene - en inglés - de esta técnica de “trocear” una clave en pedazos). A continuación exponemos algunas funciones hash de uso más extendidas.

4.1.1.- Restas sucesivas

Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:

1998-000 --> 0 = 1998000 - 1998000
1998-001 --> 1 = 1998001 - 1998000
1998-002 --> 2 = 1998002 - 1998000
...
1998-399 --> 399 = 1998399 - 1998000
1999-000 --> 400 = 1999000 - 1999000 + 400
...
yyyy-nnn --> N = yyyynnn - yyyy000 + (400 * (yyyy-1998))

4.1.2.- Método de división o resto

Se escoge un número *m* menor o igual que la capacidad *n* de elementos.

La función hash H se define por:

H(X) = X % m *o* *H(X) = (X % m) + 1*

donde X % m indica el resto de la división de X por m. La segunda fórmula se usa cuando queremos que las direcciones hash vayan de *1 a m* en vez de desde *0 hasta m-1*.
El mejor resultado del método de división se obtiene cuando m es primo (es decir, m no es divisible por ningún entero positivo distinto de 1 y m).
Si el número m es el 13, los números siguientes quedan transformados en:
13000000 --> 0 = 13000000 % 13
12345678 --> 7 = 12345678 % 13
13602499 --> 1 = 13602499 % 13
71140205 --> 6 = 71140205 % 13
73062137 --> 5 = 73062137 % 13

Este método presenta problemas de colisión que veremos más adelante.

4.1.3.- Método del medio cuadrado
La clave es multiplicada por sí misma y los dígitos del medio (el número exacto depende del rango del índice) del cuadrado son utilizados como índice.
Es importante que se usen las mismas posiciones del cuadrado para todas las claves.
123 * 123 = 15129 --> 51
136 * 136 = 18496 --> 84
730 * 730 = 532900 --> 29
301 * 301 = 90601 --> 06
625 * 625 = 390625 --> 06

Este método también presenta problemas de colisión.

4.1.4.- Truncamiento
Ignora parte de la clave y se utiliza la parte restante directamente como índice. Si las claves, por ejemplo, son enteros de 8 dígitos y el arreglo de transformación tiene 1000 posiciones, entonces el 1º, 3º y 8º dígitos pueden formar la función hash. Por ejemplo, 72588495 se convierte en 755.
El truncamiento es un método muy rápido, pero falla para distribuir las claves de modo uniforme. También presenta problemas de colisión.

4.1.5.- Método de superposición
Consiste en la división de la clave en diferentes partes y su combinación en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice. La clave X se divide en varias partes X_1, X_2, ..., X_n, donde cada una, con la única posible excepción de la última, tiene el mismo número de dígitos que la dirección especificada. A continuación, se suman todas las partes. En esta operación se desprecian los dígitos más significativos que se obtengan de arrastre o acarreo. También presenta problemas de colisión.

Hay dos formas de conseguir la función hash mediante este método:
1. *superposición por desplazamiento* donde todas las partes se suman entre sí.

13000000 --> 130 = 130 + 000 + 00
12345678 --> 657 = 123 + 456 + 78
71140205 --> 118 = 1118 = 711 + 402 + 05
13602499 --> 259 = 136 + 024 + 99
35000010 --> 360 = 350 + 000 + 10

2. *superposición por plegamiento* se hace la inversa a las partes pares X_2, X_4, ..., antes de sumarlas, con el fin de afinar más.

13000000 --> 130 = 130 + 000 + 00
12345678 --> 855 = 123 + 654 + 78
71140205 --> 920 = 711 + 204 + 05
13602499 --> 655 = 136 + 420 + 99
35000010 --> 359 = 350 + 000 + 10

Las claves no tienen por qué ser numéricas y en ellas podrán aparecer letras. En general, cuando aparecen letras en las claves se suele asociar a cada letra un entero, o se utiliza su código ASCII.
Existen otras funciones de randomización o asignación, las cuales tienen sus ventajas y desventajas dependiendo del conjunto de claves a las que se ha de aplicar la función de asignación.
Una consideración muy importante cuando se elige una función Hash es la *eficiencia de cálculo*, no es suficiente que se pueda encontrar el elemento en el primer intento si este intento toma mucho más tiempo que otros.

4.2.- Soluciones al problema de las colisiones

Supongamos que queremos añadir un nuevo elemento k, pero supongamos que la posición de memoria H(k) ya está ocupada. Esta situación se llama *colisión*.

A continuación se presentan las formas generales de resolver las colisiones:

- Rehashing o Reasignación.
- Arreglos anidados o Cubos.
- Encadenamiento o Tablas Hash Abiertas.
- Zona de Desbordamiento.

El procedimiento particular que se escoja dependerá de muchos factores. Un factor importante es la relación entre el número *n* de elementos y el número *m* de tamaño del arreglo. Esta razón, $\lambda=n/m$ se llama *factor de carga*.
Las colisiones son casi imposibles de evitar. Véase el siguiente ejemplo:

> Si una clase tiene 24 alumnos y tenemos un arreglo con espacio para 365 registros. Una función hash aleatoria es la de escoger la fecha de nacimiento como dirección hash. Aunque el factor de carga λ=24/365 (aproximadamente el 7%) es muy pequeño, se puede demostrar que existe una posibilidad mayor del 50% de que dos alumnos hayan nacido el mismo día.

La *eficiencia* de una función hash con un procedimiento de resolución de colisiones se mide por el número medio de *pruebas* (comparaciones entre elementos) necesitadas para encontrar la posición de un elemento X dado. La eficiencia depende principalmente del factor de carga λ. En concreto, interesa conocer:

$S(\lambda)$ = número medio de celdas examinadas para una búsqueda CON éxito.
$U(\lambda)$ = número medio de celdas examinadas para una búsqueda SIN éxito.

4.2.1.- Rehashing o Reasignación

Se trata de buscar (o colocar, ya que los datos que pretenden ser localizados en el arreglo mediante su dirección hash, *deberán ser previamente almacenados utilizando esa misma dirección*) el elemento cuya posición ya está ocupada, en otra posición disponible en el arreglo. Esto se hace mediante la función de *reasignación R(H(X))*, la cual acepta un índice del arreglo y produce otro.
Si la posición del arreglo dada por H(X) se encuentra ocupada (o, en caso de búsqueda, ocupada por un elemento distinto al buscado), se aplica la función R sobre H(X) para encontrar otra posición donde se pueda colocar (o localizar) el elemento. Si de nuevo R(H(X)) se encuentra también ocupada, se vuelve a aplicar

rehashing hasta encontrar una posición vacía, en cuyo caso se realizaría la inserción (o en caso de tratarse de una búsqueda, indicaría que el elemento buscado no existe).
Existen varios métodos que trabajan bajo el principio de comparación y reasignación de elementos:

- Prueba lineal o secuencial.
- Prueba cuadrática.
- Doble dirección hash.

4.2.1.1.- Prueba Lineal o Secuencial

Consiste en que una vez detectada la colisión se debe recorrer el arreglo secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía. Se trata al arreglo como a una estructura circular: el siguiente elemento después del último es el primero. La función de rehashing es, por lo tanto, de la forma:

$$R(H(X)) = H(X) + 1$$

Ejemplo:

> Si la posición 397 ya estaba ocupada, el registro con clave 0596397 es colocado en la posición 398, la cual se encuentra disponible. Una vez que el registro ha sido insertado en esta posición, otro registro que genere la posición 397 o la 398 es insertado en la posición siguiente disponible.

Una implementación en Java de búsqueda hash solucionando las colisiones por medio de la prueba lineal o secuencial se muestra a continuación:

```
public class Hashing
    {
    public int[] table;

    public Hashing(int count)
            {
            table=new int[count];
            }

    public void addSequential(int value)
            {
            int place=getHash(value);
            if(table[place]!=0)
                    {
                    do
                            place=getNextPlace(place);
                    while(table[place]!=0);
                    }
            table[place]=value;
            }

    public int searchSequential(int search)
            {
            int place=getHash(search);
            if(table[place]!=search)
                    {
                    //para que no siga en forma circular infinitamente
                    int init=place;
                    do
                            place=getNextPlace(place);
                    while(table[place]!=search & init!=place);
                    if(init= =place)
                            return -1;
                    }
            return place;
            }
```

```
private int getHash(int value)
        {
        return value % table.length;
        }

private int getNextPlace(int place)
        {
        place++;
        if(place= =table.length)
                place=0;
        return place;
        }
}
```

Utilizando este método de resolución de colisiones, el número medio de pruebas para una búsqueda CON éxito es:

$$S(\lambda) = 1/2 (1 + (1 / (1 - \lambda)))$$

y el número medio de pruebas para una búsqueda SIN éxito es:

$$U(\lambda) = 1/2 (1 + (1 / (1 - \lambda)^2))$$

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del arreglo permanezcan vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial perdiendo así las ventajas del método hash. Por lo tanto, una propiedad de una buena función de reasignación es que *para cualquier índice i, las reasignaciones sucesivas R(H(i)), R(H(R(H(i)))),... cubran la mayoría de los enteros entre 0 y la longitud del arreglo.*
Se cumple que, para cualquier función *R(H(i))=(i % m) + c*, donde *m* es el número de elementos del arreglo y *c* es una constante tal que *c* y *m* son *primos relativos* (es decir, no tienen factores en común), *genera valores sucesivos que cubren todo el arreglo*. Si m y c tienen factores en común, el número de posiciones diferentes del arreglo que se obtienen será el cociente de dividir *m* y c.
Basándonos en esta última afirmación, el algoritmo de búsqueda (y el de carga) tienen un problema: utilizando una función de rehashing de este tipo podrían salir sin inserción aun cuando exista alguna posición vacía en el arreglo.

Ejemplo:

Dada la función de reasignación:

$$RH(i) = (i \% 1000) + 200$$

Con esta función, cada clave solo puede ser colocada en cinco posiciones posibles: (1000/200= 5) si i = 215, y esta posición está ocupada, entonces se reasigna de la siguiente forma:
RH(215) = (215 % 1000) + 200 = *415*
RH(415) = (415 % 1000) + 200 = 615
RH(615) = (615 % 1000) + 200 = 815
RH(815) = (815 % 1000) + 200 = 15
RH(15) = (15 % 1000) + 200 = 215
RH(215) = (215 % 1000) + 200 = *415*
...
Si estas cinco posiciones posibles están ocupadas, saldremos sin inserción aun cuando haya posiciones libres en el arreglo.

Colisión por rehashing: agrupamiento o clustering
A pesar de la utilización de una función de rehashing donde *c* y *m* sean primos, con el método de prueba lineal o secuencial los elementos siguen tendiendo a *agruparse*, o sea, a aparecer unos junto a otros, cuando el factor de carga es mayor del 50%. Cuando el arreglo está vacío es igualmente probable que cualquier

elemento al azar sea colocado en cualquier posición libre dentro del arreglo, pero una vez que se han tenido algunas entradas y se han presentado varias colisiones o choques en la asignación, esto no es cierto.

Ejemplo:

Claves
4618396
4957397
1286399
0000990
0000991
1200992
0047993
9846995

> En este caso, es cinco veces más probable que un registro sea insertado en la posición 994 que en la posición 401. Esto se debe a que cualquier registro cuya clave genera la asignación 990, 991, 992, 993 o 994 será colocado en la 994, mientras que cualquier registro cuya clave genera la asignación 401 será colocado en su posición.

Este fenómeno en el que *dos claves que inicialmente generan una asignación en dos sitios diferentes, luego compiten entre sí en reasignaciones sucesivas*, se denomina *agrupamiento o clustering*: $H(X) \neq H(Y)$ pero $RH(X) = RH(Y)$ en reasignaciones sucesivas.
Se cumple que:

- cualquier función de reasignación que dependa únicamente del índice dará lugar a agrupamiento.
- daría agrupamiento la función de reasignación $Rh(i) = (i \% m) + c$, aunque c y m fuesen primos.

Ejemplo:

> Si m=1000, c=21 y las posiciones 10, 31, 52, 73 y 94 están todas ocupadas, cualquier elemento cuya clave sea cualquiera de estos cinco enteros sería colocado en la posición 115. Se dice entonces que 10, 31, 52, 73 y 94 forman un *cluster*.

El clustering se debe a que las reasignaciones sucesivas siguen siempre la misma secuencia. Si conseguimos que esa secuencia varíe en cada reasignación se evitaría la formación del cluster.

Las técnicas que se presentan a continuación minimizan el agrupamiento:

4.2.1.2.- Prueba Cuadrática

Este método es similar al de la prueba lineal o secuencial. La diferencia consiste en que, en lugar de buscar en las posiciones con direcciones: dirHash, dirHash + 1, dirHash + 2, dirHash + 3, ... buscamos linealmente en las posiciones con direcciones: dirHash, dirHash + 1, dirHash + 4, dirHash + 9, ..., $dirHash + i^2$.
Esta variación permite una mejor distribución de las claves colisionadas.
Si el número m de posiciones en el arreglo T es un número primo y el factor de carga no excede del 50%, sabemos que siempre insertaremos el nuevo elemento X y que ninguna celda será consultada dos veces durante un acceso.

4.2.1.3.- Doble Direccionamiento Hash

Consiste en que una vez detectada la colisión se debe generar otra dirección aplicando una función hash H_2 a la dirección previamente obtenida. Entonces buscamos linealmente en las posiciones que se encuentran a una distancia $H_2(X)$, 2 $H_2(X)$, 3 $H_2(X)$, ...

La función hash H_2 que se aplique a las sucesivas direcciones puede ser o no ser la misma que originalmente se aplicó a la clave. No existe una regla que permita decidir cuál será la mejor función a emplear en el cálculo de las sucesivas direcciones. Pero una buena elección sería $H_2(X) = R - (X \% R)$, siendo R un número primo más pequeño que el tamaño del arreglo. Y se obtienen resultados mejores cuando el tamaño del arreglo es un número primo.

Problemas de la técnica de rehashing

- Inserción: como se asume un arreglo de tamaño fijo, si el número de elementos aumenta más allá de ese tamaño es imposible insertarlo sin que sea necesario asignar un arreglo más grande y recalcular los valores de asignación de las claves de todos los elementos que ya se encuentran en el arreglo utilizando una nueva función de asignación.
- Borrado: es difícil eliminar un elemento. Por ejemplo, si el elemento *r1* está en la posición *p*, para añadir un elemento *r2* cuya clave *k2* queda asignada en p, éste debe ser insertado en la primera posición libre de las siguientes: Rh(*p*), Rh(Rh(*p*))... Si luego *r1* es eliminado y la posición *p* queda vacía, una búsqueda posterior del elemento *r2* comenzará en la posición H(*k2*) = *p*. Como esta posición está ahora vacía, el proceso de búsqueda puede erróneamente llevarnos a la conclusión de que el elemento *r2* no se encuentra en el arreglo.

Una posible solución a este problema es marcar el elemento eliminado como '*eliminado*' en vez de '*vacío*', y continuar la búsqueda cuando se encuentra una posición como '*eliminada*'. Pero esto sólo es factible para un número pequeño de eliminaciones pues, en caso contrario, una búsqueda sin éxito requerirá una búsqueda a través de todo el arreglo debido a que la mayoría de las posiciones estarán marcadas como '*eliminado*', en lugar de '*vacío*'. Por esto, el direccionamiento directo normalmente no se usará cuando el arreglo vaya a ser modificado constantemente.

4.2.2.- Arreglos anidados o Cubos

Este método consiste en que cada elemento del arreglo tenga otro arreglo en el cual se almacenen los elementos colisionados. Si bien la solución parece ser sencilla, es claro también que resulta ineficiente. Al trabajar con arreglos se depende del espacio que se le haya asignado a éste. Lo cual conduce a un nuevo problema difícil de solucionar: elegir un tamaño adecuado de arreglo que permita un equilibrio entre el costo de memoria y el número de valores colisionados que pudiera almacenar.

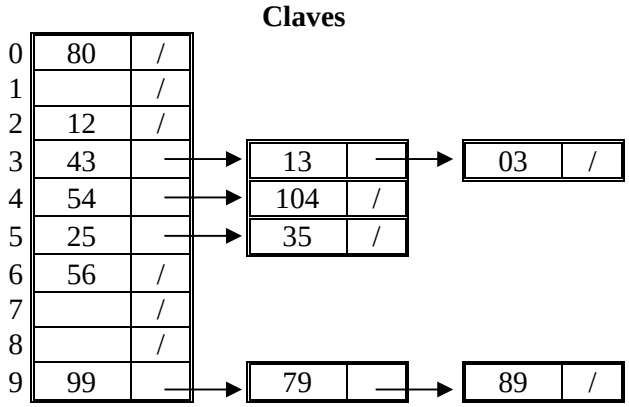
Cuando el cubo se llena, debemos tratar de nuevo con colisiones.

Existen dos maneras de implementarlo, una seria usando arreglos anidados (arreglo de arreglos) y la otra seria usando matrices (cubos). Vemos esta última implementación con un ejemplo:

	Claves			
0	80			...
1				...
2	12			...
3	43	13	03	...
4	54	104		...
5	25	35		...
6	56			...
7				...
8				...
9	99	79	89	...

4.2.3.- Encadenamiento o Tablas Hash Abiertas

Otro método para resolver las colisiones consiste en mantener una lista encadenada de todos los elementos cuyas claves generan la misma posición. Si la función hash genera valores entre 0 y (m - 1), declaramos un arreglo de nodos de encabezamiento de tamaño *m*, de manera que Tabla[i] apunta a la lista con todos los elementos cuyas claves generan posiciones en *i*.
Ejemplo:



- Inserción: al buscar un elemento cuya función hash le hace corresponder la posición i, se accede a la cabeza de la lista correspondiente a esa posición: Tabla[i], y se recorre la lista que dicha posición inicia. Si éste no se encuentra entonces se inserta al final de la lista.
- Eliminación: la eliminación de un nodo de un arreglo que ha sido construida mediante randomización y encadenamiento se reduce simplemente a remover un nodo de la lista encadenada. Un nodo eliminado no afecta a la eficiencia del algoritmo de búsqueda. El algoritmo continúa como si el nodo nunca se hubiera insertado.

Estas listas pueden reorganizarse para maximizar la eficiencia de búsqueda, utilizando diferentes métodos:
Probabilidad. Consiste en insertar los elementos dentro de la lista en su punto apropiado. Esto significa que si *prob* es la probabilidad de que un elemento sea el argumento buscado, entonces el elemento debe insertarse entre los elementos r(i) y r(i +1), donde i es tal que:

$$P(i) >= prob >= P(i + 1)$$

El problema es que P(i) pocas veces se conocen a priori, y aunque algunos elementos sean buscados más frecuentemente que otros, es casi imposible identificar dichos elementos por adelantado. Además, la probabilidad de que un elemento determinado sea recuperado puede cambiar con el tiempo.

Movimiento al frente. Cuando una búsqueda ha tenido éxito, el elemento encontrado es retirado de su posición actual en la lista y colocado en la cabeza de dicha lista.

Trasposición. Cuando una búsqueda de un elemento ha tenido éxito es intercambiado con el elemento que le precede inmediatamente, de manera que si es accedido muchas veces llegará a ocupar la primera posición de la lista.
El método de trasposición adelantando solo una posición evita el inconveniente que tiene el de movimiento al frente, que solamente porque un elemento sea recuperado una vez es movido inmediatamente al frente de la lista, reduciendo la eficiencia de la búsqueda con respecto a otros elementos que le precedían. El método de trasposición asegura que al avanzar un elemento en una posición cada vez que es obtenido, éste sólo avanzará hasta el frente de la lista si es sacado con una alta frecuencia.
Otra ventaja del método de trasposición sobre el de movimiento al frente es que se puede aplicar eficientemente tanto a arreglos de arreglos como a arreglos de listas. La trasposición de dos elementos en un arreglo es una operación relativamente eficiente, mientras que el movimiento desde el medio de un arreglo hasta el frente implica (en promedio) mover medio arreglo.

El número medio de pruebas, con encadenamiento, para una búsqueda con éxito y para una búsqueda sin éxito tiene los siguientes valores aproximados:

$$S(\lambda) = 1 + \frac{1}{2}\lambda \qquad U(\lambda) = e^{-\lambda} + \lambda$$

Ventajas y desventajas del uso del encadenamiento

- Ventajas: es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones registradas en una posición, siempre será posible tratar una más.
- Desventajas: la desventaja principal es el espacio extra adicional que se requiere para la referencia a otros elementos. Sin embargo, el arreglo inicial es generalmente pequeño en esquemas que utilizan encadenamiento comparado con aquellos que utilizan reasignación. Esto se debe a que el encadenamiento es menos catastrófico si el arreglo llega a llenarse completamente, pues siempre es posible asignar más nodos y añadirlos a varias listas.

Por supuesto, si las listas llegan a ser muy largas no se va a obtener ninguno de los beneficios de la randomización o hashing, como eran el direccionamiento directo y la eficiencia de búsqueda resultante.

4.2.4.- Zona de Desbordamiento

Se trata de mantener una zona reservada para aquellos elementos que llegan a colisionar, de manera que cuando se produzca una colisión el elemento se va a localizar en esta zona de desbordamiento.

Al realizar la búsqueda y comprobar si el elemento buscado está en la posición dada por su tabla hash, si esa posición ya está ocupada por otro elemento con el mismo valor de hashing, se seguirá buscando a partir del inicio de la zona de desbordamiento de manera secuencial, hasta encontrar el elemento o llegar al final de dicha zona de desbordamiento.

Ejemplo:

	Claves
0	80
1	
2	12
3	43
4	54
5	25
6	56
7	
8	
9	99

	Desbordamientos
0	13
1	104
2	79
3	03
4	35
5	89
.	
.	
.	
n	

REFERENCIAS

De libros...

1. "Algoritmos + Estructuras = Programas". Nicklaus Wirth. Editorial Prentice Hall. 1987. México.

2. "Cómo programar en Java". H. M. Deitel y P. J. Deitel. Editorial Prentice Hall. 1997. México.

3. "Data Structures, Algorithms and Program Style Using C". J. F. Korsh y L. J. Garrett. Editorial PWS Publishing CO. 1988. Estados Unidos.

4. "Data Structures: from Arrays to Priority Queues". Wayne Amsbury. Editorial Wadsworth Publishing. 1985. Estados Unidos.

5. "Data Structures and Algorithm Analysis in C". Mark Allen Weiss. Editorial Addison Wesley. 1996. Estados Unidos.

6. "El lenguaje de programación Java". Tercera edición. K. Arnold, J. Gosling y D. Holmes. Editorial Addison Wesley. 2001. España.

7. "Estructuras de Datos". Osvaldo Cairó y Silvia Guardati. Editorial Mc. Graw Hill. 1993. México.

8. "Estructuras de Datos en Java". Mark Allen Weiss. Editorial Addison Wesley. 2000. España.

9. "Estructuras de Datos en Pascal". A. Tenenbaum y M. Augestein. Editorial Prentice Hall. 1983. España.

10. "Estructuras de Datos y Algoritmos". Alfred Aho, John Hopcroft y Jeffrey Ullman. Editorial Addison-Wesley. 1988. Estados Unidos.

11. "Estructuras de Datos-Algoritmos, Abstracción y Objetos". L. Joyanes Aguilar y I. Zahonero Martínez. Editorial Mac. Graw Hill. 1998. España.

12. "Fundamentals of Data Structures". Ellis Horowitz y Sartaj Sahni. Editorial WH Freeman & CO. 1983. Estados Unidos.

13. "Introducción a la Programación Orientada a Objetos con Java". Primera Edición. C. Thomas Wu. Editorial Mc. Graw Hill. 2001. España.

14. "Java Data Structures and Programming". Liwu Li. Editorial Springer. 1998. Alemania.

15. "Pascal y Estructuras de Datos". Neil Dale y Susan Lily. Editorial Mc. Graw Hill. 1992. México.

16. "Reliable Data Structures in C". Thomas Plum. Editorial Plum Hall. 1985. Estados Unidos.

De páginas en Internet...

- http://babbage.clarku.edu/~achou/cs160/source/datastructures/
- http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/binsort.html
- http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/radixsort.html
- http://dis.eafit.edu.co/cursos/st030/desarrolloCursoST030_034.html

- http://doschivos.com/new/algobusque.htm
- http://guamuchil.udo.mx/~fcampos/ordenamientos.htm
- http://home.hawaii.rr.com/chodandkathy/linear/search sort_files/slide0285.htm
- http://mailweb.pue.udlap.mx/~ccastane/Syllabus_Estructura_Datos/Sy_EstructuraDatos_Java.html
- http://mailweb.udlap.mx/~ingrid/Clases/Trie.html
- http://members.tripod.com/fcc98/tutores/alg2/alg2.html
- http://rfhs8012.fh-regensburg.de/~saj39122/AD/aufgaben/aufg06/aufg06.htm
- http://trevinca.ei.uvigo.es/~pavon/transparencias/TTema4Grafos.pdf
- http://xue.unalmed.edu.co/~dosorio/linux/hash.htm
- http://www.cee.hw.ac.uk/DSA/ProgramIndex.htm
- http://www.cs.utsa.edu/~carola/teaching/cs5633/spring04/slides/Lecture-18.pdf
- http://www.dcc.uchile.cl/~rbaeza/handbook/hbook.html
- http://www.dma.fi.upm.es/fleury/definicionesGrafos.htm
- http://www.dma.fi.upm.es/gregorio/grafos/paginagrafos.html
- http://www.hut.fi/~ccandoli/botanica/algorithms/java/
- http://www.infor.uva.es/~jmrr/TAD2003/home.htm
- http://www.it.uc3m.es/~tsps/practica08/enunciado.htm
- http://www.itlp.edu.mx/publica/tutoriales/estructdatos2/
- http://www.lcc.uma.es/~av/Libro/CAP2.pdf
- http://www.javacommerce.com/tutorial/JavaData/JavaData.html
- http://www.seeingwithc.org/topic4html.html