Unidad 1.- Análisis de algoritmos

El Análisis de algoritmos es una parte importante de una Teoría de complejidad computacional más amplia, que provee estimaciones teóricas para los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Estas estimaciones resultan ser bastante útiles en la búsqueda de algoritmos eficientes.

A la hora de realizar un análisis teórico de algoritmos es corriente calcular su complejidad en un **sentido asintótico**, es decir, para una tamaño de entrada suficientemente grande. La cota superior asintótica, y las notaciones omega y theta se usan con esa finalidad. Por ejemplo, la búsqueda binaria decimos que se ejecuta en una cantidad de pasos proporcional a un logaritmo, en O(log(n)), coloquialmente "en tiempo logarítmico". Normalmente las estimaciones asintóticas se utilizan porque diferentes implementaciones del mismo algoritmo no tienen porque tener la misma eficiencia. No obstante la eficiencia de dos implementaciones "razonables" cualesquiera de un algoritmo dado están relacionadas por una constante multiplicativa llamada **constante oculta**.

La medida exacta (no asintótica) de la eficiencia a veces puede ser computada pero para ello suele hacer falta aceptar supuestos acerca de la implementación concreta del algoritmo, llamada modelo de computación. Un modelo de computación puede definirse en términos de una ordenador abstracto, como la Máquina de Turing, y/o postulando que ciertas operaciones se ejecutan en una unidad de tiempo. Por ejemplo, si al conjunto ordenado al que aplicamos una búsqueda binaria tiene n elementos, y podemos garantizar que una única búsqueda binaria puede realizarse en un tiempo unitario, entonces se requieren como mucho $\log_2 N + 1$ unidades de tiempo para devolver una respuesta.

Las medidas exactas de eficiencia son útiles para quienes verdaderamente implementan y usan algoritmos, porque tienen más precisión y así les permite saber cuanto tiempo pueden suponer que tomará la ejecución. Para algunas personas, como los desarrolladores de videojuegos, una constante oculta puede significar la diferencia entre éxito y fracaso.

Las estimaciones de tiempo dependen de cómo definamos un paso. Para que el análisis tenga sentido, debemos garantizar que el tiempo requerido para realizar un paso esté acotado superiormente por una constante. Hay que mantenerse precavido en este terreno; por ejemplo, algunos análisis cuentan con que la suma de dos números se hace en un paso. Este supuesto puede no estar garantizado en ciertos contextos. Si por ejemplo los números involucrados en la computación puedes ser arbitrariamente largos, dejamos de poder sumir que la adición requiere un tiempo contante (usando papel y lápiz, compara el tiemo que necesitas para sumar dos enteros de 2 dígitos cada uno y el necesario para hacerlo con enteros de 1000 dígitos).

1.1 Concepto de Complejidad de algoritmos.

El análisis de algoritmo es una parte muy importante de la ciencia de la computación, de modo que la medida de la eficiencia de u algoritmo será uno de los factores fundamentales. Por consiguiente es importante poder analizar los requisitos de tiempo y espacio de u algoritmo para ver si existe dentro de limites aceptables.
Es difícil realizar un análisis simple de un algoritmo que determine la cantidad exacta de tiempo requerida para ejecutarlo. La primera complicación es que la cantidad exacta de tiempo dependerá de la implementación del algoritmo y de la maquina en que se ejecuta. El análisis normalmente debe ser independiente del lenguaje o maquina que se utilice para implementar el algoritmo. El análisis del algoritmo tratará de obtener el orden de magnitud de tiempo requerido para la ejecución del mismo y cada algoritmo tendrá un coste computacional diferente.
La eficiencia es un criterio que se debe utilizar cuando se selecciona un algoritmo y su implementación. Existe al menos tres dificultades fundamentales que son los siguientes:

- ¿Cómo se codifican los algoritmos?
- ¿Que computadoras utilizara?
- ¿Qué datos debe utilizar el programa?

El análisis del a eficiencia debe ser independiente de las implementación especificas de la computadora y de los datos específicos que se manipulan. Pero las consideraciones de eficiencia fundamentales son: el tiempo y el espacio. La complejidad del espacio de un programa es la cantidad de memoria que se necesita para ejecutar hasta la compleción (terminación).
La complejidad de el tiempo de un programa es la cantidad de tiempo de computadora que se necesita para ejecutar hasta la compleción al considerar estos dos aspectos podremos tener la medida exacta de tiempo y espacio del cual necesitamos para realizar u buen análisis de algoritmos.

Un algoritmo será mas eficiente comparado con otro, siempre que consuma menos recursos, como el tiempo y espacio de memoria necesarios para ejecutarlo.

1.2 Aritmética de la notación O.

La notación O (también llamada O mayúscula), se utiliza para comparar la eficiencia de los algoritmos.

Tipos de análisis de la Notacion O

<u>Peor caso (usualmente)</u>
T(n) = Tiempo máximo necesario para un problema de tamaño n.

<u>Caso medio (a veces)</u>
T(n) = Tiempo esperado para un problema cualquiera de tamaño n.

- Requiere establecer una distribución estadística

<u>Mejor caso (engañoso)</u>

Análisis del peor caso

¿Cuál es el tiempo que necesitaría un algoritmo concreto?

-
    - Varía en función del ordenador que utilicemos.

- Varía en función del compilador que seleccionemos.
- Puede variar en función de nuestra habilidad como programadores.

~~IDEA:~~ Ignorar las constantes dependientes del contexto.

~~SOLUCIÓN:~~ Fijarse en el crecimiento de T(n) cuando n ->∞

NOTACION "O"

O(g(n)) = { f(n) | Ec,n0 constantes positivas tales que f (n) = c g(n) A n = n0 }

En la práctica, se ignoran las constantes y los términos de menor peso:
3n3 + 90n2 – 5n + 6046 = O (n3)

<u>Eficiencia asintótica</u>
Cuando n es lo suficientemente grande…
Un algoritmo O (1), es más eficiente que un algoritmo O (log n),
Un algoritmo O (log n), es más eficiente que un algoritmo O(n),
Un algoritmo O(n), es más eficiente que un algoritmo O(n log n),
Un algoritmo O(n log n), es mas eficiente que un algoritmo O (n2),
Un algoritmo O (n2), es más eficiente que un algoritmo O(n3),
Un algoritmo O (n3), es más eficiente que un algoritmo O (2n).

~~NOTA:~~ En ocasiones, un algoritmo más ineficiente puede resultar másadecuado para resolver un problema real ya que, en a práctica, hay que tener en cuenta otros aspectos además de la eficiencia.

<u>Propiedades de la notación O</u>
c O(f(n)) = O(f(n))
O (f(n)+g(n)) = max {O (f(n)), O (g(n))}
O (f(n)+g(n)) = O (f(n)+g(n))
O (f(n)) O (g(n)) = O (f(n) g(n))
O(O (f(n))) = O (f(n))

1.3 Complejidad
El análisis de complejidad se basa en la comparación del tipo de ejecución de los algoritmos desarrollados para resolver un problema.
A partir del análisis de complejidad se han definido varias clases de problemas: los problemas que se resuelven en tiempo polinomial por una máquina determinista forman la clase P y los problemas que se resuelven en tiempo polinomial por una máquina no determinista forman la clase NP.
En computación, al hablar de complejidad, no se está refiriendo a la dificultad que se tendría para diseñar un programa, o a lo rebuscado de un algoritmo. La teoría de complejidad tiene que ver con dos medidas de desempeño: tiempo y espacio.
La complejidad computacional de un problema es una medida de los recursos computacionales (generalmente el tiempo) requeridos para resolver el problema.
La complejidad temporal tiene que ver con el tiempo que tarda un programa para ejecutarse. La complejidad espacial estudia la cantidad de espacio de almacenamiento que es necesario para una operación.

1.3.1 Tiempo de Ejecución de un Algorítmo

-
  - El tiempo de ejecución de un algoritmo, es prioritario cuando este es analizado.
  - El tiempo de ejecución de un algoritmo o estructura de datos depende de varios factores relativos al hardware (procesador, reloj, memoria, disco, etc) y el software (sistema operativo, lenguaje, compilador, etc.).
  - Interesa hallar la dependencia del tiempo de ejecución en función del tamaño de la entrada.
  - Un método para estudiar el tiempo de ejecución es la experimentación, que tiene limitaciones:

    1.-Los experimentos se pueden hacer sobre un conjunto limitado de entradas de prueba.
    2.-Es necesario realizar los experimentos con el mismo hardware y software.
    3.-Es necesario implementar y ejecutar el algoritmo.

Adicionalmente a la experimentación conviene disponer de un enfoque analítico que:
Tome en consideración todas las posibles entradas.
Permita evaluar la eficiencia de dos algoritmos de forma independiente del hardware y software.
Se pueda realizar estudiando una representación de alto nivel del algoritmo sin necesidad de implementarlo.

1.3.2 Complejidad en Espacio
Memoria que utiliza un programa para su ejecución, La eficiencia en memoria de un algoritmo indica la cantidad de espacio requerido para ejecutar el algoritmo; es decir, el espacio en memoria que ocupan todas las variables propias al algoritmo. Para calcular la memoria estática de un algoritmo se suma la memoria que ocupan las variables declaradas en el algoritmo. Para el caso de la memoria dinámica, el cálculo no es tan simple ya que, este depende de cada ejecución del algoritmo.

1.4 Selección de un Algorítmo
La selección del mejor algoritmo para el problema dado; en la práctica las cosas no son tan fáciles y en ocasiones no es tan obvio poder escoger un algoritmo entre un grupo, pues la cantidad de operaciones no siempre es el único criterio a tener en cuenta. También debe considerarse, por ejemplo, el tamaño del algoritmo, la claridad con que está expresado, etc.
<u>- Finalidad.-</u>Todo algoritmo tiene el objetivo o una finalidad de resolver un tipo de problema y se ejecuta para obtener un resultado que es la solución de un caso particular de ese problema.

<u>- Orden.-</u>Los pasos del algoritmo tienen que ejecutarse en un orden preciso e indicado en el algoritmo. Si este orden se altera, generalmente no se obtiene el resultado deseado.

<u>- Finitud.-</u>El algoritmo tiene que ser finito en tres aspectos:

a. El algoritmo es una secuencia finita de pasos (no tiene sentido ninguno un algoritmo cuya descripción sea infinitamente larga)
b. La ejecución del algoritmo termina después de concluir un número finito de pasos Nótese que este aspecto es diferente al anterior, pues podría tenerse

una cantidad pequeña de pasos en la definición que se ejecutara repetidamente un número infinito de veces. Por ejemplo: “dividir un número racional mayor que 0 por dos repetidamente hasta que el cociente sea cero”.

c. La ejecución de un paso cualquiera del algoritmo tiene que poder ejecutarse en un tiempo finito.

1. Introducción

La resolución práctica de un problema exige por una parte un algoritmo o método de resolución y por otra un programa o codificación de aquel en un ordenador real. Ambos componentes tienen su importancia; pero la del algoritmo es absolutamente esencial, mientras que la codificación puede muchas veces pasar a nivel de anécdota.

A efectos prácticos o ingenieriles, nos deben preocupar los recursos físicos necesarios para que un programa se ejecute. Aunque puede haber muchos parametros, los mas usuales son el tiempo de ejecución y la cantidad de memoria (espacio). Ocurre con frecuencia que ambos parametros están fijados por otras razones y se plantea la pregunta inversa: ¿cual es el tamano del mayor problema que puedo resolver en T segundos y/o con M bytes de memoria? En lo que sigue nos centraremos casi siempre en el parametro tiempo de ejecución, si bien las ideas desarrolladas son fácilmente aplicables a otro tipo de recursos.

Para cada problema determinaremos un medida N de su tamaño (por número de datos) e intentaremos hallar respuestas en función de dicho N. El concepto exacto que mide N depende de la naturaleza del problema. Así, para un vector se suele utizar como N su longitud; para una matriz, el número de elementos que la componen; para un grafo, puede ser el número de nodos (a veces es mas importante considerar el número de arcos, dependiendo del tipo de problema a resolver); en un fichero se suele usar el número de registros, etc. Es imposible dar una regla general, pues cada problema tiene su propia lógica de coste.

2. Tiempo de Ejecución

Una medida que suele ser útil conocer es el tiempo de ejecución de un programa en función de N, lo que denominaremos T(N). Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción. Así, un trozo sencillo de programa como

```
S1; for (int i= 0; i < N; i++) S2;
```

requiere

```
T(N)= t1 + t2*N
```

siendo t1 el tiempo que lleve ejecutar la serie "S1" de sentencias, y t2 el que lleve la serie "S2".

Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten. Esto hace que mas que un valor T(N) debamos hablar de un rango de valores

```
Tmin(N)  <=  T(N)  <=  Tmax(N)
```

los extremos son habitualmente conocidos como "caso peor" y "caso mejor". Entre ambos se hallara algun "caso promedio" o más frecuente.

Cualquier fórmula T(N) incluye referencias al parámetro N y a una serie de constantes "Ti" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del ordenador que lo ejecuta. Dado que es fácil cambiar de compilador y que la potencia de los ordenadores crece a un ritmo vertiginoso (en la actualidad, se duplica anualmente), intentaremos analizar los algoritmos con

algun nivel de independencia de estos factores; es decir, buscaremos estimaciones generales ampliamente válidas.

3. Asintotas

Por una parte necesitamos analizar la potencia de los algoritmos independientemente de la potencia de la máquina que los ejecute e incluso de la habilidad del programador que los codifique. Por otra, este análisis nos interesa especialmente cuando el algoritmo se aplica a problema grandes. Casi siempre los problemas pequeños se pueden resolver de cualquier forma, apareciendo las limitaciones al atacar problemas grandes. No debe olvidarse que cualquier técnica de ingeniería, si funciona, acaba aplicándose al problema más grande que sea posible: las tecnologias de éxito, antes o después, acaban llevándose al límite de sus posibilidades.

Las consideraciones anteriores nos llevan a estudiar el comportamiento de un algoritmo cuando se fuerza el tamaño del problema al que se aplica. Matemáticamente hablando, cuando N tiende a infinito. Es decir, su comportamiento asintótico.

Sean "g(n)" diferentes funciones que determinan el uso de recursos. Habra funciones "g" de todos los colores. Lo que vamos a intentar es identificar "familias" de funciones, usando como criterio de agrupación su comportamiento asintótico.

A un conjunto de funciones que comparten un mismo comportamiento asintótico le denominaremos un órden de complejidad'. Habitualmente estos conjuntos se denominan O, existiendo una infinidad de ellos.

Para cada uno de estos conjuntos se suele identificar un miembro f(n) que se utiliza como representante de la clase, hablándose del conjunto de funciones "g" que son del orden de "f(n)", denotándose como

```
        g IN O(f(n))
```

Con frecuencia nos encontraremos con que no es necesario conocer el comportamiento exacto, sino que basta conocer una cota superior, es decir, alguna función que se comporte "aún peor".

La definición matemática de estos conjuntos debe ser muy cuidadosa para involucrar ambos aspectos: identificación de una familia y posible utilización como cota superior de otras funciones menos malas:

Dícese que el conjunto O(f(n)) es el de las funciones de orden de f(n), que se define como

```
  O(f(n))= {g: INTEGER -> REAL+  tales que
            existen las constantes  k  y  N0 tales que
            para todo  N > N0,  g(N) <= k*f(N) }
```

en palabras, O(f(n)) esta formado por aquellas funciones g(n) que crecen a un ritmo menor o igual que el de f(n).

De las funciones "g" que forman este conjunto O(f(n)) se dice que **"están dominadas asintóticamente"** por "f", en el sentido de que para N suficientemente grande, y salvo una constante multiplicativa "k", f(n) es una cota superior de g(n).

3.1. Órdenes de Complejidad

Se dice que O(f(n)) define un **"orden de complejidad"**. Escogeremos como representante de este orden a la función f(n) más sencilla del mismo. Así tendremos

O(1)	orden constante
O(log n)	orden logarítmico
O(n)	orden lineal
O(n log n)	
$O(n^2)$	orden cuadrático
$O(n^a)$	orden polinomial (a > 2)
$O(a^n)$	orden exponencial (a > 2)
O(n!)	orden factorial

Es más, se puede identificar una jerarquía de órdenes de complejidad que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo A se puede demostrar de un cierto orden O_1, es cierto que tambien pertenece a todos los órdenes superiores (la relación de orden çota superior de' es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo cubra.

3.1.1. Impacto Práctico

Para captar la importancia relativa de los órdenes de complejidad conviene echar algunas cuentas.

Sea un problema que sabemos resolver con algoritmos de diferentes complejidades. Para compararlos entre si, supongamos que todos ellos requieren 1 hora de ordenador para resolver un problema de tamaño N=100.

¿Qué ocurre si disponemos del doble de tiempo? Notese que esto es lo mismo que disponer del mismo tiempo en un odenador el doble de potente, y que el ritmo actual de progreso del hardware es exactamente ese:

"duplicación anual del número de instrucciones por segundo".

¿Qué ocurre si queremos resolver un problema de tamaño 2n?

O(f(n))	N=100	t=2h	N=200
log n	1 h	10000	1.15 h
n	1 h	200	2 h
n log n	1 h	199	2.30 h
$\mathbf{n^2}$	1 h	141	4 h
$\mathbf{n^3}$	1 h	126	8 h
$\mathbf{2^n}$	1 h	101	10^{30} h

Los algoritmos de complejidad O(n) y O(n log n) son los que muestran un comportamiento más "natural": prácticamente a doble de tiempo, doble de datos procesables.

Los algoritmos de complejidad logarítmica son un descubrimiento fenomenal, pues en el doble de tiempo permiten atacar problemas notablemente mayores, y para resolver un problema el doble de grande sólo hace falta un poco más de tiempo (ni mucho menos el doble).

Los algoritmos de tipo polinómico no son una maravilla, y se enfrentan con dificultad a problemas de tamaño creciente. La práctica viene a decirnos que son el límite de lo "tratable".

Sobre la tratabilidad de los algoritmos de complejidad polinómica habria mucho que hablar, y a veces semejante calificativo es puro eufemismo. Mientras complejidades del orden $O(n^2)$ y $O(n^3)$ suelen ser efectivamente abordables, prácticamente nadie acepta algoritmos de orden $O(n^{100})$, por muy polinómicos que sean. La frontera es imprecisa.

Cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas ridiculamente pequeños.

A la vista de lo anterior se comprende que los programadores busquen algoritmos de complejidad lineal. Es un golpe de suerte encontrar algo de complejidad logarítmica. Si se encuentran soluciones polinomiales, se puede vivir con ellas; pero ante soluciones de complejidad exponencial, más vale seguir buscando.

No obstante lo anterior ...

- ... si un programa se va a ejecutar muy pocas veces, los costes de codificación y depuración son los que más importan, relegando la complejidad a un papel secundario.
- ... si a un programa se le prevé larga vida, hay que pensar que le tocará mantenerlo a otra persona y, por tanto, conviene tener en cuenta su legibilidad, incluso a costa de la complejidad de los algoritmos empleados.
- ... si podemos garantizar que un programa sólo va a trabajar sobre datos pequeños (valores bajos de N), el orden de complejidad del algoritmo que usemos suele ser irrelevante, pudiendo llegar a ser incluso contraproducente.

  Por ejemplo, si disponemos de dos algoritmos para el mismo problema, con tiempos de ejecución respectivos:

algoritmo	**tiempo**	**complejidad**
f	100 n	O(n)
g	n^2	$O(n^2)$

  asintóticamente, "f" es mejor algoritmo que "g"; pero esto es cierto a partir de N > 100. Si nuestro problema no va a tratar jamás problemas de tamaño mayor que 100, es mejor solución usar el algoritmo "g".

El ejemplo anterior muestra que las constantes que aparecen en las fórmulas para T(n), y que desaparecen al calcular las funciones de complejidad, pueden ser decisivas desde el punto de vista de ingeniería. Pueden darse incluso ejemplos más dramaticos:

algoritmo	tiempo	complejidad
f	n	O(n)
g	100 n	O(n)

aún siendo dos algoritmos con idéntico comportamiento asintótico, es obvio que el algoritmo "f" es siempre 100 veces más rápido que el "g" y candidato primero a ser utilizado.

- ... usualmente un programa de baja complejidad en cuanto a tiempo de ejecución, suele conllevar un alto consumo de memoria; y viceversa. A veces hay que sopesar ambos factores, quedándonos en algún punto de compromiso.
- ... en problemas de cálculo numérico hay que tener en cuenta más factores que su complejidad pura y dura, o incluso que su tiempo de ejecución: queda por considerar la precisión del cálculo, el máximo error introducido en cálculos intermedios, la estabilidad del algoritmo, etc. etc.

3.2. Propiedades de los Conjuntos O(f)

No entraremos en muchas profundidades, ni en demostraciones, que se pueden hallar en los libros especializados. No obstante, algo hay que saber de cómo se trabaja con los conjuntos O() para poder evaluar los algoritmos con los que nos encontremos.

Para simplificar la notación, usaremos O(f) para decir O(f(n))

Las primeras reglas sólo expresan matemáticamente el concepto de jerarquía de órdenes de complejidad:

A. La relación de orden definida por

f < g <=> f(n) IN O(g)
es reflexiva: f(n) IN O(f)
y transitiva: f(n) IN O(g) y g(n) IN O(h) => f(n) IN O(h)

B. f IN O(g) y g IN O(f) <=> O(f) = O(g)

Las siguientes propiedades se pueden utilizar como reglas para el cálculo de órdenes de complejidad. Toda la maquinaria matemática para el cálculo de límites se puede aplicar directamente:

```
C.  Lim(n->inf)f(n)/g(n) = 0  =>  f IN O(g)
                               =>  g NOT_IN O(f)
                               =>  O(f) es subconjunto de O(g)
D.  Lim(n->inf)f(n)/g(n) = k  =>  f IN O(g)
                               =>  g IN O(f)
```

=> O(f) = O(g)

E. $Lim_{(n->inf)}$ f(n)/g(n)= INF => f NOT_IN O(g)
=> g IN O(f)
=> O(f) es superconjunto de O(g)

Las que siguen son reglas habituales en el cálculo de límites:

F. Si f, g IN O(h) => f+g IN O(h)
G. Sea k una constante, f(n) IN O(g) => k*f(n) IN O(g)
H. Si f IN O(h1) y g IN O(h2) => f+g IN O(h1+h2)
I. Si f IN O(h1) y g IN O(h2) => f*g IN O(h1*h2)
J. Sean los reales 0 < a < b => O(n^a) es subconjunto de O(n^b)
K. Sea P(n) un polinomio de grado k => P(n) IN O(n^k)
L. Sean los reales a, b > 1 => O(log_a) = O(log_b)

La regla [L] nos permite olvidar la base en la que se calculan los logaritmos en expresiones de complejidad.

La combinación de las reglas [K, G] es probablemente la más usada, permitiendo de un plumazo olvidar todos los componentes de un polinomio, menos su grado.

Por último, la regla [H] es la basica para analizar el concepto de secuencia en un programa: la composición secuencial de dos trozos de programa es de orden de complejidad el de la suma de sus partes.

4. Reglas Prácticas

Aunque no existe una receta que siempre funcione para calcular la complejidad de un algoritmo, si es posible tratar sistematicamente una gran cantidad de ellos, basandonos en que suelen estar bien estructurados y siguen pautas uniformes.

Loa algoritmos bien estructurados combinan las sentencias de alguna de las formas siguientes

1. sentencias sencillas
2. secuencia (;)
3. decisión (if)
4. bucles
5. llamadas a procedimientos

4.0. Sentencias sencillas

Nos referimos a las sentencias de asignación, entrada/salida, etc. siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño este relacionado con el tamaño N del problema. La inmensa mayoría de las sentencias de un algoritmo requieren un tiempo constante de ejecución, siendo su complejidad O(1).

4.1. Secuencia (;)

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales, aplicándose las operaciones arriba expuestas.

4.2. Decisión (if)

La condición suele ser de O(1), complejidad a sumar con la peor posible, bien en la rama THEN, o bien en la rama ELSE. En decisiones multiples (ELSE IF, SWITCH CASE), se tomara la peor de las ramas.

4.3. Bucles

En los bucles con contador explícito, podemos distinguir dos casos, que el tamaño N forme parte de los límites o que no. Si el bucle se realiza un número fijo de veces, independiente de N, entonces la repetición sólo introduce una constante multiplicativa que puede absorberse.

```
Ej.- for (int i= 0; i < K; i++) { algo_de_O(1) }       =>  K*O(1) = O(1)
```

Si el tamaño N aparece como límite de iteraciones ...

```
Ej.- for (int i= 0; i < N; i++) { algo_de_O(1) }  =>  N * O(1) = O(n)

Ej.- for (int i= 0; i < N; i++) {
       for (int j= 0; j < N; j++) {
         algo_de_O(1)
       }
     }
```

tendremos N * N * O(1) = $O(n^2)$

```
Ej.- for (int i= 0; i < N; i++) {
       for (int j= 0; j < i; j++) {
         algo_de_O(1)
       }
     }
```

el bucle exterior se realiza N veces, mientras que el interior se realiza `1, 2, 3, ... N` veces respectivamente. En total,

$$1 + 2 + 3 + ... + N = N*(1+N)/2 \quad -> O(n^2)$$

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
Ej.- c= 1;
     while (c < N) {
       algo_de_O(1)
       c= 2*c;
     }
```

El valor incial de "c" es 1, siendo "2^k" al cabo de "k" iteraciones. El número de iteraciones es tal que

2^k >= N => k= eis ($\log_2$ (N)) [el entero inmediato superior]

y, por tanto, la complejidad del bucle es O(log n).

```
Ej.- c= N;
     while (c > 1) {
       algo_de_O(1)
       c= c / 2;
     }
```

Un razonamiento análogo nos lleva a $\log_2(N)$ iteraciones y, por tanto, a un orden O(log n) de complejidad.

```
Ej.- for (int i= 0; i < N; i++) {
       c= i;
```

```
        while (c > 0) {
          algo_de_O(1)
          c= c/2;
        }
      }
```

tenemos un bucle interno de orden O(log n) que se ejecuta N veces, luego el conjunto es de orden O(n log n)

4.4. Llamadas a procedimientos

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí. El coste de llamar no es sino una constante que podemos obviar inmediatamente dentro de nuestros análisis asintóticos.

El cálculo de la complejidad asociada a un procedimiento puede complicarse notáblemente si se trata de procedimientos recursivos. Es fácil que tengamos que aplicar técnicas propias de la matemática discreta, tema que queda fuera de los límites de esta nota técnica.

4.5. Ejemplo: evaluación de un polinomio

Vamos a aplicar lo explicado hasta ahora a un problema de fácil especificación: diseñar un programa para evaluar un polinomio P(x) de grado N;

```
class Polinomio {
  private double[] coeficientes;

  Polinomio (double[] coeficientes) {
    this.coeficientes= new double[coeficientes.length];
    System.arraycopy(coeficientes, 0, this.coeficientes, 0,
                     coeficientes.length);
  }

  double evalua_1 (double x) {
    double resultado= 0.0;
    for (int termino= 0; termino < coeficientes.length; termino++) {
      double xn= 1.0;
      for (int j= 0; j < termino; j++)
        xn*= x;                 // x elevado a n
      resultado+= coeficientes[termino] * xn;
    }
    return resultado;
  }
}
```

Como medida del tamaño tomaremos para N el grado del polinomio, que es el número de coeficientes en C. Así pues, el bucle más exterior (1) se ejecuta N veces. El bucle interior (2) se ejecuta, respectivamente

$$1 + 2 + 3 + ... + N \text{ veces} = N*(1+N)/2 \Rightarrow O(n^2)$$

Intuitivamente, sin embargo, este problema debería ser menos complejo, pues repugna al sentido común que sea de una complejidad tan elevada. Se puede ser más inteligente a la hora de evaluar la potencia x^n:

```
  double evalua_2 (double x) {
```

```
    double resultado= 0.0;
    for (int termino= 0; termino < coeficientes.length; termino++) {
      resultado+= coeficientes[termino] * potencia(x, termino);
    }
    return resultado;
  }

  private double potencia (double x, int n) {
    if (n == 0)
      return 1.0;
    // si es potencia impar ...
    if (n%2 == 1)
      return x * potencia(x, n-1);
    // si es potencia par ...
    double t= potencia(x, n/2);
    return t*t;
  }
```

El análisis de la función Potencia es delicado, pues si el exponente es par, el problema tiene una evolución logarítmica; mientras que si es impar, su evolución es lineal. No obstante, como si "j" es impar entonces "j-1" es par, el caso peor es que en la mitad de los casos tengamos "j" impar y en la otra mitad sea par. El caso mejor, por contra, es que siempre sea "j" par.

Un ejemplo de caso peor seria x^{31}, que implica la siguiente serie para j:

```
31  30  15  14  7  6  3  2  1
```

cuyo número de terminos podemos acotar superiormente por

$2 * eis(\log_2(j))$,

donde eis(r) es el entero inmediatamente superior (este cálculo responde al razonamiento de que en el caso mejor visitaremos $eis(\log_2(j))$ valores pares de "j"; y en el caso peor podemos encontrarnos con otros tantos números impares entremezclados).

Por tanto, la complejidad de Potencia es de orden O(log n).

Insertada la función Potencia en la función EvaluaPolinomio, la complejidad compuesta es del orden O(n log n), al multiplicarse por N un subalgoritmo de O(log n).

Así y todo, esto sigue resultando estravagante y excesivamente costoso. En efecto, basta reconsiderar el algoritmo almacenando las potencias de "X" ya calculadas para mejorarlo sensiblemente:

```
  double evalua_3 (double x) {
    double xn= 1.0;
    double resultado= coeficientes[0];
    for (int termino= 1; termino < coeficientes.length; termino++) {
      xn*= x;
      resultado+= coeficientes[termino] * xn;
    }
    return resultado;
  }
```

que queda en un algoritmo de O(n).

Habiendo N coeficientes C distintos, es imposible encontrar ningun algoritmo de un orden inferior de complejidad.

En cambio, si es posible encontrar otros algoritmos de idéntica complejidad:

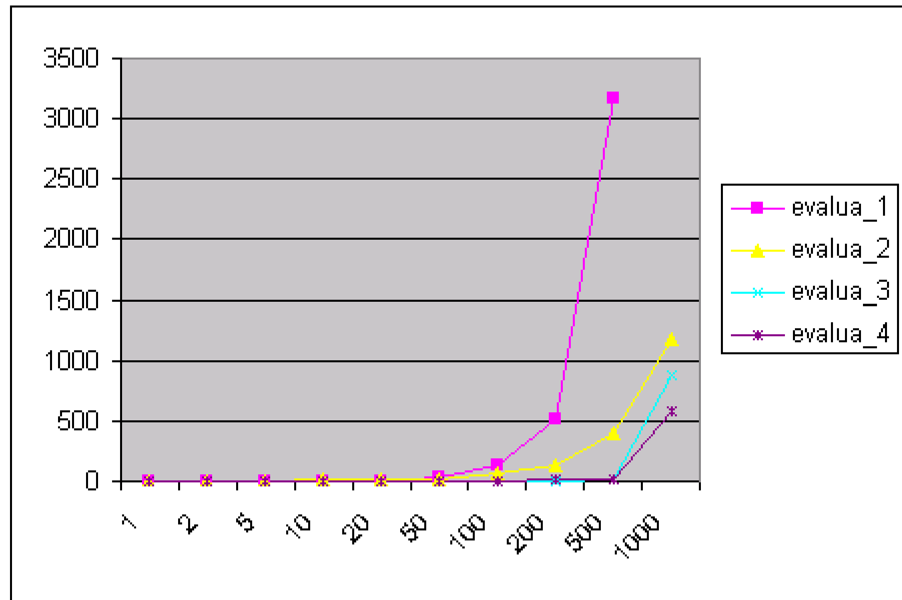
```
double evalua_4 (double x) {
  double resultado= 0.0;
  for (int termino= coeficientes.length-1; termino >= 0; termino--) {
    resultado= resultado * x +
      coeficientes[termino];
  }
  return resultado;
}
```

No obstante ser ambos algoritmos de idéntico orden de complejidad, cabe resaltar que sus tiempos de ejecución serán notablemente distintos. En efecto, mientras el último algoritmo ejecuta N multiplicaciones y N sumas, el penúltimo requiere 2N multiplicaciones y N sumas. Si, como es frecuente, el tiempo de ejecución es notablemente superior para realizar una multiplicación, cabe razonar que el último algoritmo ejecutará en la mitad de tiempo que el anterior.

4.5.1. Medidas de laboratorio

La siguiente tabla muestra algunas medidas de la eficacia de nuestros algoritmos sobre una implementación en Java:

grado	evalua_1	evalua_2	evalua_3	evalua_4
1	0	10	0	0
2	10	0	0	0
5	0	0	0	0
10	0	10	0	0
20	0	10	0	0
50	40	20	0	10
100	130	60	0	0
200	521	140	0	10
500	3175	400	10	10
1000	63632	1171	872	580



5. Problemas P, NP y NP-completos

Hasta aquí hemos venido hablando de algoritmos. Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables. Se suele decir que el orden de complejidad de un problema es el del mejor algoritmo que se conozca para resolverlo. Así se clasifican los problemas, y los estudios sobre algoritmos se aplican a la realidad.

Estos estudios han llevado a la constatación de que existen problemas muy difíciles, problemas que desafian la utilización de los ordenadores para resolverlos. En lo que sigue esbozaremos las clases de problemas que hoy por hoy se escapan a un tratamiento informático.

Clase P.-

Los algoritmos de complejidad polinómica se dice que son tratables en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P. Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables. Seria muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

Clase NP.-

Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico. Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

Clase NP-completos.-

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente podemos decir que algunos problemas se hayan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución

sería fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al Nobel reservado para el que descubra semejante solución ... ¡y se duda seriamente de que alguien lo consiga!
Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

6. Conclusiones

Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera. Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. se suele trabajar con un cálculo asintótico que indica como se comporta el algoritmo para datos muy grandes y salvo algun coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

7. Bibliografia

Es difícil encontrar libros que traten este tema a un nivel introductorio sin caer en amplios desarrollos matemáticos, aunque tambien es cierto que casi todos los libros que se precien dedican alguna breve sección al tema. Probablemente uno de los libros que sólo dedican un capítulo; pero es extremadamente claro es
L. Goldschlager and A. Lister.
Computer Science, A Modern Introduction
Series in Computer Science. Prentice-Hall Intl., London (UK), 1982.
Siempre hay algún clásico con una presentación excelente, pero entrando en mayores honduras matemáticas como
A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
Data Structures and Algorithms
Addison-Wesley, Massachusetts, 1983.
Recientemente ha aparecido un libro en castellano con una presentación muy buena, si bien esta escrito en plan matemático y, por tanto, repleto de demostraciones (un poco duro)
Carmen Torres
Diseño y Análisis de Algoritmos
Paraninfo, 1992

Para un estudio serio hay que irse a libros de matemática discreta, lo que es toda una asignatura en sí misma; pero se pueden recomendar un par de libros modernos, prácticos y especialmente claros:

R. Skvarcius and W. B. Robinson.
Discrete Mathematics with Computer Science Applications
Benjamin/Cummings, Menlo Park, California, 1986.

R. L. Graham, D. E. Knuth, and O. Patashnik.
Concrete Mathematics
Addison-Wesley, 1990.

Para saber más de problemas NP y NP-completos, hay que acudir a la "biblia", que en este tema se denomina

M. R. Garey and D. S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness
Freeman, 1979.

Sólo a efectos documentales, permítasenos citar al inventor de las nociones de complejidad y de la notación O():

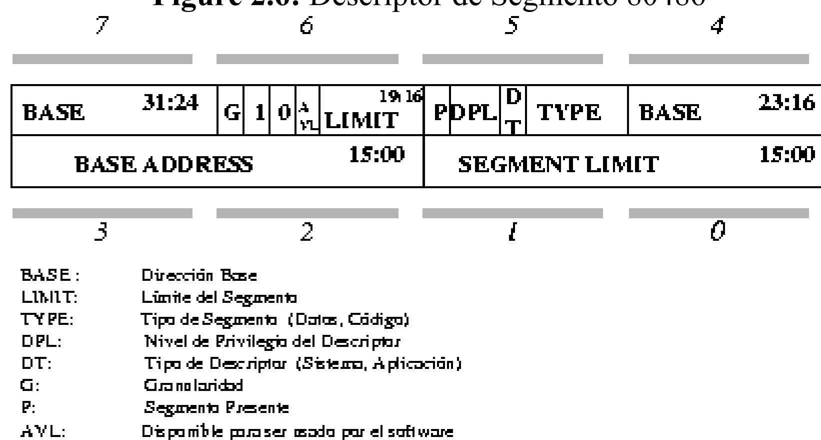
P. Bachmann
Analytische Zahlen Theorie
1894

Unidad 2.- Manejo de memoria.

Manejo de memoria

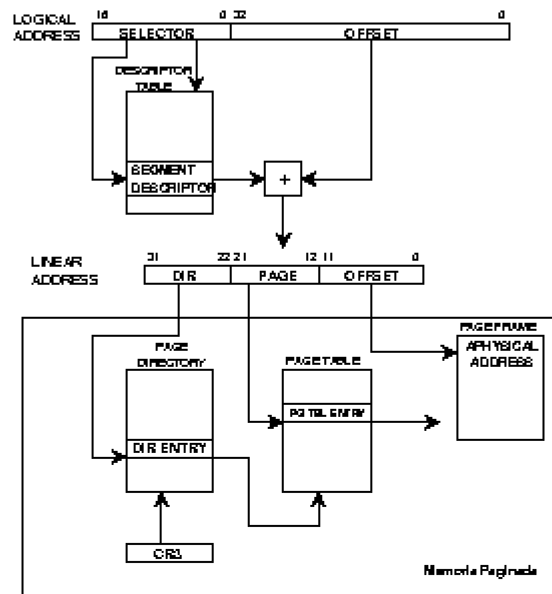
La memoria en el bus del i486 se denomina *memoria física*, es decir, cada byte en la memoria tiene asociada una única *dirección física*, la cual tiene un rango de cero hasta 2^{32}-1. Los programas direccionan el modelo de memoria denominado *memoria virtual*. El procesador i486 trabaja con el modelo de paginación y segmentación de memoria. Cualquiera o ambos de estos mecanismos puede ser utilizado. Una dirección utilizada por un programa se llama *dirección lógica*. El hardware de segmentación translada una dirección segmentada (lógica) a una dirección de 32 bits en un espacio de direccionamiento continuo y no segmentado llamada dirección lineal. El hardware de paginación translada una dirección lineal a una dirección física.

Figure 2.6: Descriptor de Segmento 80486



Las direcciones lógicas se trasladan a direcciones lineales debido a que tienen una dirección como un *offset* (registro de 32 bits) y un *selector* (registro de 16 bits). Cada segmento tiene un *descriptor de segmento*, el cual contiene su dirección base y su límite (ver figura 2.6). Si el *offset* no sobrepasa el límite, y no existe otra condición que prevenga la lectura del segmento, entonces se suman el segmento y la base para formar una dirección lineal.Si no se utiliza el modo de paginación (cuando el bit 32 del registro CR0 tiene el valor 0) entonces la dirección lineal se utiliza directamente como la dirección física. Si el mecanismo de paginación está activo, se utiliza la dirección lineal para formar la dirección física correspondiente.

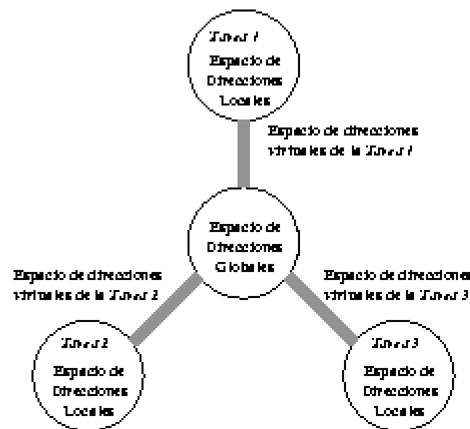
Figure 2.7: Manejo de memoria del 80486



El hardware de paginación divide el espacio de direcciones en bloques fijos de 4K bytes, llamados páginas. Entonces, el espacio de direcciones lógicas se mapea a un espacio de direcciones lógicas y éste, a su vez, se mapea a un número de páginas. Una página puede estar en memoria o en disco. Cuando se obtiene una dirección lógica, se translada a una direccón para una página en memoria, o se obtiene una excepción. La excepción le da oportunidad al sistema operativo de leer una página del disco y actualizar el mapeo de páginas. Este proceso se aprecia en la figura 2.7.

Todas las tarea que se ejecutan en modo protegido puede accesar los segmentos de los espacios de direcciones locales y de direcciones globales (ver figura 2.8).

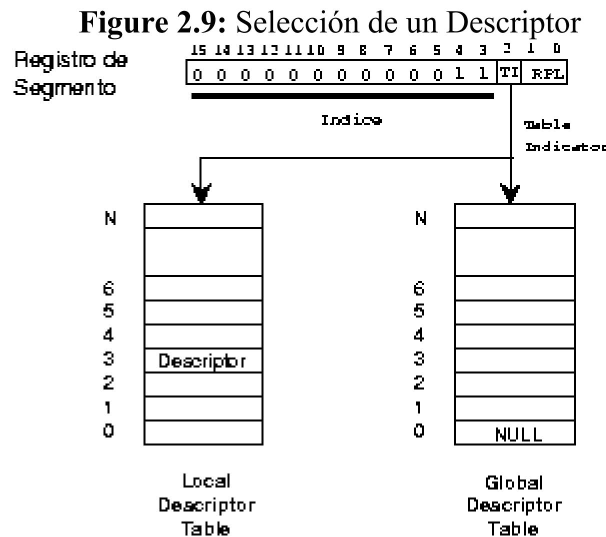
Figure 2.8: Espacios de direcciones virtuales y aislamiento de tareas.



En el espacio de direcciones globales se colocan los datos y código que se necesiten en cualquier tarea --ejemplo de estos objetos son algunos servicios del núcleo--. Mientras que en el espacio de direcciones locales se colocan elementos utilizados en alguna tarea simple. El espacio de

direcciones es privado, a menos que un segmento sea mapeado, en dos o más, intencionalmente, ya que una tarea no puede accesar una localidad de memoria si no tiene su selector respectivo. La figura 2.9 muestra la forma en que se elige un descriptor de segmento. Los 13 bits superiores del selector proporcionan el índice del descriptor en la tabla de descriptores --global (GDT) o local (LDT)--. Sus 3 bits inferiores son bits de control. El bit 2 se denomina *indicador de tabla*, si está encendido se busca en la tabla local de descriptores, si no, se busca en la tabla global. Los bits 0 y 1 son el RPL[2.5], estos bits describen el nivel de privilegio del segmento.

Figure 2.9: Selección de un Descriptor



2.1 Manejo de memoria estática

La forma más fácil de almacenar el contenido de una variable en memoria en tiempo de ejecución es en memoria estática o permanente a lo largo de toda la ejecución del programa. No todos los objetos (variables) pueden ser almacenados estáticamente. Para que un objeto pueda ser almacenado en memoria estática su tamaño (número de bytes necesarios para su almacenamiento) ha de ser conocido en tiempo de compilación. Como consecuencia de esta condición no podrán almacenarse en memoria estática:

* Los objetos correspondientes a procedimientos o funciones recursivas, ya que en tiempo de compilación no se sabe el número de variables que serán necesarias.

- Las estructuras dinámicas de datos tales como listas, árboles, etc. ya que el número de elementos que las forman no es conocido hasta que el programa se ejecuta.

Figura 3. Alojamiento en memoria de un objeto X.

Las técnicas de asignación de memoria estática son sencillas. A partir de una posición señalada por un puntero de referencia se aloja el objeto X, y se avanza el puntero tantos

bytes como sean necesarios para almacenar el objeto X. La asignación de memoria puede hacerse en tiempo de compilación y los objetos están vigentes desde que comienza la ejecución del programa hasta que termina.
En los lenguajes que permiten la existencia de subprogramas, y siempre que todos los objetos de estos subprogramas puedan almacenarse estáticamente -por ejemplo en FORTRAN-IV, como se puede ver en la figura 4a- se aloja en la memoria estática un registro de activación correspondiente a cada uno de los subprogramas.
Figura 4a.Estructura de la memoria estática en FORTRAN-IV
Estos registros de activación contendrán las variables locales, parámetros formales y valor devuelto por la función, tal como indica la fig. 4b.
Figura 4b. Registro de activavión.
Dentro de cada registro de activación las variables locales se organizan secuencialmente. Existe un solo registro de activación para cada procedimiento y por tanto no están permitidas las llamadas recursivas. El proceso que se sigue cuando un procedimiento p llama a otro q es el siguiente:

1. p evalúa los parámetros de llamada, en caso de que se trate de expresiones complejas, usando para ello una zona de memoria temporal para el almacenamiento intermedio. Por ejemplos, sí la llamada a q es q((3*5)+(2*2),7) las operaciones previas a la llamada propiamente dicha en código máquina han de realizarse sobre alguna zona de memoria temporal. (En algún momento debe haber una zona de memoria que contenga el valor intermedio 15, y el valor intermedio 4 para sumarlos a continuación). En caso de utilización de memoria estática ésta zona de temporales puede ser común a todo el programa, ya que su tamaño puede deducirse en tiempo de compilación.

2. q inicializa sus variables y comienza su ejecución.

Dado que las variables están permanentemente en memoria es fácil implementar la propiedad de que conserven o no su contenido para cada nueva llamada

2.2 Manejo de memoria dinámica.

Sobre el tratamiento de memoria, GLib™ dispone de una serie de instrucciones que sustituyen a las ya conocidas por todos `malloc`, `free`, etc. y, siguiendo con el modo de llamar a las funciones en GLib™, las funciones que sustituyen a las ya mencionadas son `g_malloc` y `g_free`.

Reserva de memoria.

La función `g_malloc` posibilita la reserva de una zona de memoria, con un número de bytes que le pasemos como parámetro. Además, también existe una función similar llamada `g_malloc0` que, no sólo reserva una zona de memoria, sino que, además, llena esa zona de memoria con ceros, lo cual nos puede beneficiar si se necesita un zona de memoria totalmente limpia.

gpointer **g_malloc** (*numero_de_bytes*);

gulong *numero_de_bytes* ;
gpointer **g_malloc0** (*numero_de_bytes*);

gulong *numero_de_bytes* ;

Existe otro conjunto de funciones que nos permiten reservar memoria de una forma parecida a cómo se hace en los lenguajes orientados a objetos. Esto se realiza mediante las siguientes macros definidas en GLib™ `/gmem.h`:

```
	/* Convenience memory allocators
	*/
	#define g_new(struct_type, n_structs)		\
		((struct_type *) g_malloc (((gsize) sizeof (struct_type)) *
((gsize) (n_structs))))
	#define g_new0(struct_type, n_structs)		\
		 ((struct_type *) g_malloc0 (((gsize) sizeof (struct_type)) *
((gsize) (n_structs))))
	#define g_renew(struct_type, mem, n_structs)	\
		 ((struct_type *) g_realloc ((mem), ((gsize) sizeof
(struct_type)) * ((gsize) (n_structs))))
```

Como se puede apreciar, no son más que macros basadas en `g_malloc`, `g_malloc0` y `g_realloc`. La forma de funcionamiento de `g_new` y `g_new0` es mediante el nombre de un tipo de datos y un número de elementos de ese tipo de datos, de forma que se puede hacer:

```
	GString *str = g_new (GString,1);
	GString *arr_str = g_new (GString, 5);
```

En estas dos líneas de código, se asigna memoria para un elemento de tipo GString, que queda almacenado en la variable `str`, y para un array de cinco elementos de tipo GString, que queda almacenado en la variable `arr_str`).

`g_new0` funciona de la misma forma que `g_new`, con la única diferencia de que inicializa a 0 toda la memoria asignada. En cuanto a `g_renew`, ésta funciona de la misma forma que `g_realloc`, es decir, reasigna la memoria asignada anteriormente.

Liberación de memoria.

Cuando se hace una reserva de memoria con `g_malloc` y, en un momento dado, el uso de esa memoria no tiene sentido, es el momento de liberar esa memoria. Y el sustituto de `free` es `g_free` que, básicamente, funciona igual que la anteriormente mencionada.

void **g_free** (*memoria_reservada*);

gpointer *memoria_reservada* ;

Realojamiento de memoria.

En determinadas ocasiones, sobre todo cuando se utilizan estructuras de datos dinámicas, es necesario ajustar el tamaño de una zona de memoria (ya sea para hacerla más grande o más pequeña). Para eso, GLib™ ofrece la función `g_realloc`, que recibe un puntero a memoria que apunta a una región que es la que será acomodada al nuevo tamaño y devuelve el puntero a la nueva zona de memoria. El anterior puntero es liberado y no se debería utilizar más:

gpointer **g_realloc** (*memoria_reservada* ,
numero_de_bytes);

gpointer *memoria_reservada* ;
gulong *numero_de_bytes* ;

UNIDAD 2: Manejo de Memoria

Memoria Estática

La forma más fácil de almacenar el contenido de una variable en memoria en tiempo de ejecución es en memoria estática o permanente a lo largo de toda la ejecución del programa.

No todos los objetos (variables) pueden ser almacenados estáticamente.

Para que un objeto pueda ser almacenado en memoria estática su tamaño (número de bytes necesarios para su almacenamiento) ha de ser conocido en tiempo de compilación, como consecuencia de esta condición no podrán almacenarse en memoria estática:

* Los objetos correspondientes a procedimientos o funciones recursivas, ya que en tiempo de compilación no se sabe el número de variables que serán necesarias.

* Las estructuras dinámicas de datos tales como listas, árboles, etc. ya que el número de elementos que las forman no es conocido hasta que el programa se ejecuta.

Las técnicas de asignación de memoria estática son sencillas.

A partir de una posición señalada por un puntero de referencia se aloja el objeto X, y se avanza el puntero tantos bytes como sean necesarios para almacenar el objeto X.

La asignación de memoria puede hacerse en tiempo de compilación y los objetos están vigentes desde que comienza la ejecución del programa hasta que termina.

En los lenguajes que permiten la existencia de subprogramas, y siempre que todos los objetos de estos subprogramas puedan almacenarse estáticamente se aloja en la memoria estática un registro de activación correspondiente a cada uno de los subprogramas.

Estos registros de activación contendrán las variables locales, parámetros formales y valor devuelto por la función.

Dentro de cada registro de activación las variables locales se organizan secuencialmente. Existe un solo registro de activación para cada procedimiento y por tanto no están permitidas las llamadas recursivas. El proceso que se sigue cuando un procedimiento p llama a otro q es el siguiente:

1. p evalúa los parámetros de llamada, en caso de que se trate de expresiones complejas, usando para ello una zona de memoria temporal para el almacenamiento intermedio. Por ejemplos, sí la llamada a q es q((3*5)+(2*2),7) las operaciones previas a la llamada propiamente dicha en código máquina han de realizarse sobre alguna zona de memoria temporal. (En algún momento debe haber una zona de memoria que contenga el valor intermedio 15, y el valor intermedio 4 para sumarlos a continuación). En caso de utilización de memoria estática ésta zona de temporales puede ser común a todo el programa, ya que su tamaño puede deducirse en tiempo de compilación.

2. q inicializa sus variables y comienza su ejecución.

Dado que las variables están permanentemente en memoria es fácil implementar la propiedad de que conserven o no su contenido para cada nueva llamada

Memoria Dinámica

¿Qué es la memoria dinámica?

Supongamos que nuestro programa debe manipular estructuras de datos de longitud desconocida. Un ejemplo simple podría ser el de un programa que lee las líneas de un archivo y las ordena. Por tanto, deberemos leer un número indeterminado de líneas, y tras leer la última, ordenarlas. Una manera de manejar ese ``número indeterminado'', sería declarar una constante MAX_LINEAS, darle un valor vergonzosamente grande, y declarar un array de tamaño MAX_LINEAS. Esto, obviamente, es muy ineficiente (y feo). Nuestro programa no sólo quedaría limitado por ese valor máximo, sino que además gastaría esa enorme cantidad de memoria para procesar hasta el más pequeño de los ficheros.

La solución consiste en utilizar memoria dinámica. La memoria dinámica es un espacio de almacenamiento que se solicita en tiempo de ejecución. De esa manera, a medida que el proceso va necesitando espacio para más líneas, va solicitando más memoria al sistema operativo para guardarlas. El medio para manejar la memoria que otorga el sistema operativo, es el puntero, puesto que no podemos saber en tiempo de compilación dónde nos dará huecos el sistema operativo (en la memoria de nuestro PC).

Memoria Dinámica.

Sobre el tratamiento de memoria, GLib dispone de una serie de instrucciones que sustituyen a las ya conocidas por todos malloc, free, etc. y, siguiendo con el modo de llamar a las funciones en GLib, las funciones que sustituyen a las ya mencionadas son g_malloc y g_free.

Reserva de memoria.

La función g_malloc posibilita la reserva de una zona de memoria, con un número de bytes que le pasemos como parámetro. Además, también existe una función similar llamada g_malloc0 que, no sólo reserva una zona de memoria, sino que, además, llena esa zona de memoria con ceros, lo cual nos puede beneficiar si se necesita un zona de memoria totalmente limpia.

gpointer g_malloc (gulong numero_de_bytes);

gpointer g_malloc0 (gulong numero_de_bytes);

Existe otro conjunto de funciones que nos permiten reservar memoria de una forma parecida a cómo se hace en los lenguajes orientados a objetos.

Liberación de memoria.

Cuando se hace una reserva de memoria con g_malloc y, en un momento dado, el uso de esa memoria no tiene sentido, es el momento de liberar esa memoria. Y el sustituto de free es g_free que, básicamente, funciona igual que la anteriormente mencionada.

```
void g_free (gpointer memoria_reservada );
```

Realojamiento de memoria

En determinadas ocasiones, sobre todo cuando se utilizan estructuras de datos dinámicas, es necesario ajustar el tamaño de una zona de memoria (ya sea para hacerla más grande o más pequeña). Para eso, GLib ofrece la función g_realloc, que recibe un puntero a memoria que apunta a una región que es la que será acomodada al nuevo tamaño y devuelve el puntero a la nueva zona de memoria. El anterior puntero es liberado y no se debería utilizar más:

```
gpointer g_realloc (gpointer memoria_reservada , gulong numero_de_bytes );
```

Asignación dinámica

El proceso de compactación del punto anterior es una instancia particular del problema de asignación de memoria dinámica, el cual es el cómo satisfacer una necesidad de tamaño n con una lista de huecos libres. Existen muchas soluciones para el problema. El conjunto de huecos es analizado para determinar cuál hueco es el más indicado para asignarse. Las estrategias más comunes para asignar algún hueco de la tabla son:

Primer ajuste: Consiste en asignar el primer hueco con capacidad suficiente. La búsqueda puede iniciar ya sea al inicio o al final del conjunto de huecos o en donde terminó la última búsqueda. La búsqueda termina al encontrar un hueco lo suficientemente grande.

Mejor ajuste: Busca asignar el espacio más pequeño de los espacios con capacidad suficiente. La búsqueda se debe de realizar en toda la tabla, a menos que la tabla esté ordenada por tamaño. Esta estrategia produce el menor desperdicio de memoria posible.

Peor ajuste: Asigna el hueco más grande. Una vez más, se debe de buscar en toda la tabla de huecos a menos que esté organizada por tamaño. Esta estrategia produce los huecos de sobra más grandes, los cuales pudieran ser de más uso si llegan procesos de tamaño mediano que quepan en ellos.

Se ha demostrado mediante simulacros que tanto el primer y el mejor ajuste son mejores que el peor ajuste en cuanto a minimizar tanto el tiempo del almacenamiento. Ni el primer o el mejor ajuste es claramente el mejor en términos de uso de espacio, pero por lo general el primer ajuste es más rápido.

Unidad 3 Estructuras lineales estática y dinámicas.

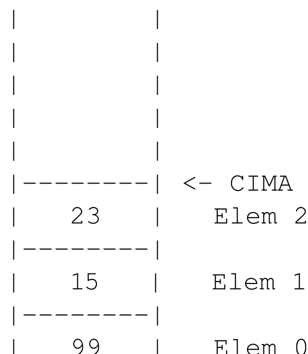
Introducción

♦ Las estructuras lineales son importantes porque aparecen con mucha frecuencia en situaciones de la vida: Una cola de clientes de un banco, las instrucciones de un programa, los caracteres de una cadena o las páginas de un libro
♦ Características: existe un único elemento, llamado primero, existe
un único elemento, llamado último, cada elemento, excepto el primero, tiene un único predecesor y cada elemento, excepto el último, tiene un único sucesor.
♦ Operaciones: crear la estructura vacía, insertar un elemento, borrar y obtener un elemento. Para definir claramente el comportamiento de la estructura es necesario determinar en qué posición se inserta un elemento nuevo y qué elemento se borra o se obtiene.
♦ Principales estructuras lineales: pilas, colas y listas.

3.1 Pilas

PILAS (STACKS)

Es un TAD que se caracteriza por el modo de acceso a sus elementos. En estas pilas, el último elemento en añadirse es el primero en liberarse. No existe un método de acceso directo a cualquier elemento de la pila, sino que para acceder a uno de ellos es necesario desapilar los anteriores (los que estén "por encima" de éste). Las pilas son estructuras de tipo LIFO (Last Input First Output). Llevan asociados una variable llamada *cima* que indica la posición del último elemento apilado.



Ej: un montón de platos.

Implementación de una pila con vectores:

```
/*--- algunas definiciones ---*/
#define TipoDatoPila  int
#define MAX_PILA 100

/*--- estructura utilizada ---*/
typedef struct
{
   TipoDatoPila ElementoPila[MAX_PILA];
   int cima;
}  Pila;


/*--- inicialización: poner la cima a -1 (no hay datos) ---*/
void InicializaPila( Pila *pila )
{    pila->cima = -1;  }


/*--- pila vacia: chequea si la pila está vacía mirando la cima ---*/
int PilaVacia( Pila *pila )
{
    if ( pila->cima == -1 )  return(1);
   return(0);
}


/*--- Apilar: si no hemos llegado al límite, aumentar la cima y dejar allí el
elemeno nuevo ---*/
int Apilar( Pila *pila, TipoDatoPila elemento )
{
   if ( pila->cima == MAX_PILA-1 )  return(0);
   pila->cima = pila->cima+1;
   pila->ElementoPila[pila->cima] = elemento;
   return(1);
}


/*--- Desapilar: si la pila no está vacía, tomar el elemento de la cima y
decrementar la cima ---*/
TipoDatoPila Desapilar( Pila *pila )
{
   TipoDatoPila x;

   if( PilaVacia( pila ) == 1 )  return(0);
   x = pila->ElementoPila[pila->cima]
   pila->cima = pila->cima-1;
   return( x );
}
```

Implementación de una pila enlazada

```
/*--- algunas definiciones ---*/
#define TipoDatoPila  int

typedef struct DatoPila
{
   TipoDatoPila dato;
   struct DatoPila *anterior;
}  ElementoPila;

typedef struct
{
   ElementoPila *cima;
} Pila;

/*--- inicialización: poner la cima a NULL (no hay datos) ---*/
void InicializaPila( Pila *pila )
{    pila->cima = NULL;  }

/*--- pila vacia: chequea si la pila está vacía mirando la cima ---*/
int PilaVacia( Pila *pila )
{
    if ( pila->cima == NULL )  return(1);
   return(0);
}

/*--- Apilar: pedir memoria para un nuevo elemento y ponerlo en la cima ---*/
int Apilar( Pila *pila, TipoDatoPila elemento )
{
   ElementoPila *nuevo;
   nuevo = (ElementoPila *) malloc(sizeof(ElementoPila));
   if( nuevo == NULL ) Error();

   nuevo->dato = elemento;
   nuevo->anterior = pila->cima;
   pila->cima = nuevo;
}

/*--- Desapilar: si la pila no está vacía, tomar el elemento,
     enlazar la cima con el anterior y liberar mem ---*/
TipoDatoPila Desapilar( Pila *pila )
{
   ElementoPila *borrar;
   TipoDatoPila x;

   if( PilaVacia( pila ) == 1 )  return(0);
   borrar = cima->pila;
   x =pila->cima->dato;
   pila->cima = pila->cima->anterior;
   free( borrar );
   return( x );
}
```

USO DE LAS PILAS

a). El gestor de programas del S.O. utiliza la pila para guardar momentáneamente los parámetros y dirección de retorno de la función que se está procesando actualmente.

b). La pila también puede utilizarse para la resolución de expresiones algebráicas, pasando las expresiones a notación infija y evaluándolas de este modo, mucho más sencillo.

Operaciones sobre pila

Dada una pila de enteros llamada pila, si se realizan las siguientes operaciones:

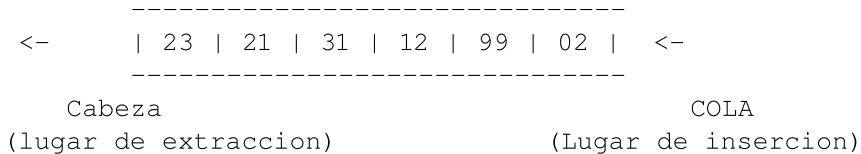
```
pila.Anadir(34);
pila.Anadir(12);
pila.Anadir(2);
pila.Anadir(35);
pila.Anadir(18);
pila.Borrar();
pila.Borrar();
pila.Anadir(15);
pila.Anadir(11);
pila.Borrar();
pila.Anadir(17);
```

¿Qué devolvería la llamada pila.Consultar()? ¿Cuál sería el contenido de la pila en ese momento?

3.2 Colas

COLAS (QUEUES)

Son un TAD formados por una sencuencia de elementos, caracterizados porque sus elementos se insertan por un extremo y se extraen por el extremo opuesto (el primer elemento en insertarse es el primero en extraerse, o FIFO).



Implementación de una cola con vectores:

```
#define TipoDatoCola  int
#define MAX_COLA 100

typedef struct
{
   TipoDatoCola ElementoCola[MAX_COLA];
   int inicio, final;
}  Cola;

void InicializaCola( Cola *cola )
{    cola->inicio = cola->final = -1;  }

int ColaVacia( Cola *cola )
{
    if ( cola->inicio == cola->final)  return(1);
   return(0);
}

int Encolar( Cola *cola, TipoDatoCola elemento )
{
   if ( cola->final == MAX_COLA-1 )  ErrorColaLlena();
   cola->final=cola->final+1;
   cola->ElementoCola[cola->final] = elemento;
}

TipoDatoCola Desencolar( Cola *cola )
{
   TipoDatoCola x;
   if( ColaVacia( cola ) == 1 )  ErrorColaVacia();
   cola->inicio = cola->inicio+1;
   x = cola->ElementoCola[cola->inicio]
   return( x );
}
```

Tras varias operaciones de inserción sobre la cola llegará un punto en que final llegará a MAX_COLA-1 y obtendremos mensajes de cola llena, así como cuando eliminemos elementos (donde aumentaremos el inicio de la cola), no pudiendo reaprovechar el espacio libre. Para evitar esto se usan las **COLAS CIRCULARES**, un tipo especial de cola que actúa como si el vector fuese una estructura circular (es decir, cuando no existen más casillas libres del vector a la derecha, sigue insertando elementos al principio). Las funciones Encolar y Desencolar quedan:

```
int Encolar( Cola *cola, TipoDatoCola elemento )
{
  int sigfinal;
   sigfinal = (cola->final+1) % MAX_COLA;
   if ( sigfinal == cola->inicio  )  ErrorColaLlena();
   cola->final=sigfinal;
   cola->ElementoCola[sigfinal] = elemento;
}

TipoDatoCola Desencolar( Cola *cola )
{
   TipoDatoCola x;
   if( ColaVacia( cola ) == 1 )  ErrorColaVacia();
   cola->inicio = (cola->inicio+1) % MAX_COLA;
   x = cola->ElementoCola[cola->inicio]
   return( x );
}
```

Mediante el operador módulo (%), obtenemos los equivalentes en el inicio y final de la cola.

<u>Implementación de una cola enlazada:</u>

```
#define TipoDatoCola int

typedef struct DatoCola
{
   TipoDatoCola dato;
   struct DatoCola *siguiente;
}  ElementoCola;

typedef struct
{
   ElementoCola *inicio, *final;
} Cola;

/*--- inicialización de la cola poniendo inicio y final a NULL
           y chequeo de si está vacia o no--- ---------------------*/
void InicializaCola( Cola *cola )
{   cola->inicio = cola->final = NULL; }

int ColaVacia( Cola *cola )
{    if ( cola->inicio == NULL )  return(1);  else   return(0);   }


/*--- Encolar: consiste en pedir memoria para un nuevo elemento, copiando
          en él los datos, y poniendo siguiente a NULL (es el último de la
cola).
```

```
         Después se hace que el último elemento de la cola apunte al nuevo,
           así como el final de la cola - --------------------------*/
int Encolar( Cola *cola, TipoDatoCola elemento )
{
   ElementoCola *nuevo;

   nuevo = (ElementoCola *) malloc( sizeof(ElementoCola ));
   if( nuevo == (ElementoCola *) NULL ) ErrorObtMemoria();

   nuevo->dato = elemento;
   nuevo->siguiente = NULL;

   if( ColaVacia(cola) )   cola->inicio = cola->final = nuevo;
   else
   {
       (cola->final)->siguiente = nuevo;
       cola->final = nuevo;
   }
}

/*--- Desencolar: Coger el inicio de la cola, hacer que su siguiente
           sea el nuevo inicio, y liberar su mem asignada--------------*/
TipoDatoCola Desencolar( Cola *cola )
{
   ElementoCola *borrar;
   TipoDatoCola elemento;
   if( ColaVacia( cola ) == 1 )  return(0);

   borrar = cola->inicio;
   elemento = borrar->dato;

   cola->inicio = (cola->inicio)->siguiente;
   free(borrar);
   return(elemento);
}
```

USO DE LAS COLAS

Las colas se suelen utilizar en los sistemas operativos para controlar las Prioridades de los Procesos (Colas de...), o en las impresoras (cola de impresión donde se almacenan las peticiones de impresión de documentos que van llegando).

Operaciones sobre cola

Dada una cola de enteros llamada cola, si se realizan las siguientes operaciones:

```
cola.Anadir(56);
cola.Anadir(12);
cola.Anadir(7);
cola.Anadir(42);
cola.Anadir(31);
cola.Borrar();
cola.Borrar();
cola.Anadir(27);
cola.Anadir(49);
cola.Borrar();
```

```
cola.Anadir(16);
cola.Borrar();
```

¿Qué devolvería una llamada `cola.Consultar()`? Y ¿cuál sería el contenido de la cola?

3.3 Listas

Listas Simples

Una lista simple es una secuencia de 0 o más elementos de un tipo de datos determinado. Este tipo de datos se llama el tipo de elemento:

$$a_1, a_2, a_3, ..., a_{n-1}, a_n$$

El valor de n es mayor o igual a 0 y cada elemento a_i es del tipo de elemento. El número n es la cantidad de elementos de la lista y se denomina longitud de la lista. Si $n \geq 1$ se denomina a_1 el primer elemento y a_n el último elemento de la lista. Si $n = 0$, se tiene una lista vacía, en este caso la longitud es igual a 0.
Una propiedad importante de una lista es que sus elementos pueden estar ordenados según algún criterio. En estos casos a_i precede a a_{i+1}; $n = 1, 2, ..., n-1$ y a_i sucede a a_{i-1}; $i = 2, 3, ..., n$.
Cada elemento de una lista tiene una posición: el elemento a_i está en la posición i. De esta forma, cada elemento tiene una distancia del primer elemento de una lista. El elemento a_i tiene la distancia i-1 del primer elemento de una lista. La posición fin(lista) es la posición que sucede a la posición del último elemento de la lista. La posición fin(lista) tiene una distancia variable con respecto al principio de la lista. Esto se debe a que la lista es una estructura de datos en la cual se agregan y se eliminan elementos.
Operaciones representativas de listas son:

- crear una lista
- insertar un elemento
- localizar un elemento
- recuperar la información de un elemento
- eliminar un elemento
- determinar el siguiente elemento o elemento anterior

Detalles sobre el TDA de listas se puede encontrar en la sección 2.1 del texto guía.
Listas se programan con arreglos, con cursores o con punteros. Cada una de estas formas tiene ventajas y desventajas. Es la aplicación específica que permite tomar la decisión sobre como programar una lista.
La utilización de arreglos y cursores se presenta muy bien la sección 2.2 del texto guía.
A continuación se presentan 10 funciones que implementan una lista enlazada por punteros.
Primero se declara una estructura que contiene dos campos: dato y puntero. El campo dato permite almacenar un número entero en cada elemento de la lista y el campo puntero permite enlazar los elementos de la lista.

```
struct nodo
{   typedef int tipo;
```

```
    tipo dato;
    nodo *puntero;
};
```

La definición del tipo *tipo* a través de typedef permite programar listas con diferentes tipos de datos para los elementos. En este caso los elementos son del tipo *int*. Si se requiere una lista con elementos de otro tipo, hay que modificar esta declaración de tipo.
En seguida se presentan 10 funciones que permiten procesar una lista:

1. listaLargo
2. listaInsertaPrimero
3. listaInserta
4. listaBusca
5. listaLocaliza
6. listaPrimeroRemover
7. listaRemover
8. listaBorrar
9. listaCopiar
10. listaCopiarParte

```
int listaLargo(nodo* ptrCabecera)
// Precondición:      ptrCabecera es el puntero al primer elemento de una lista
// Postcondición:     Se retorna el valor de la cantidad de elementos de la lista
//                    La lista no es modificada
{   nodo *ptr;
    int largo;
    largo = 0;
    for (ptr = ptrCabecera; ptr != NULL; ptr = ptr->puntero)
            largo++;
    return largo;
}
```

```
void listaInsertarPrimero(nodo*& ptrCabecera, const nodo::tipo& valor)
// Precondición:      ptrCabecera es el puntero al primer elemento de una lista
// Postcondición:     Un nuevo nodo con el valor dado ha sido agregado al comienzo de la
lista
//                    ptrCabecera apunta ahora al nuevo primer elemento de la lista
{   nodo *ptrNuevo;
    ptrNuevo = new nodo;
    ptrNuevo->dato = valor;
    ptrNuevo->puntero = ptrCabecera;
    ptrCabecera = ptrNuevo;
}
```

```
void listaInsertar(nodo* ptrPrevio, const nodo::tipo& valor)
// Precondición:      ptrPrevio es el puntero a un elemento de una lista
```

```
// Postcondición:      Un nuevo nodo con el valor dado ha sido agregado en la posicion que
//                     es posterior al elemento a cual apunta ptrPrevio
{   nodo *ptrNuevo;
    ptrNuevo = new nodo;
    ptrNuevo->dato = valor;
    ptrNuevo->puntero = ptrPrevio->puntero;
    ptrPrevio->puntero = ptrNuevo;
}
```

nodo* listaBuscar(nodo* ptrCabecera, const nodo::tipo& valor)

```
// Precondición:       ptrCabecera es el puntero al primer elemento de una lista
// Postcondición:      Se retorna el puntero al elemento que contiene el valor dado
//                     Si no hay tal elemento se retorna NULL
{   nodo *ptr;
    for (ptr = ptrCabecera; ptr != NULL; ptr = ptr->puntero)
            if (valor == ptr->dato) return ptr;
    return NULL;
}
```

nodo* listaLocalizar(nodo* ptrCabecera, int posicion)

```
// Precondición:       ptrCabecera es el puntero al primer elemento de una lista
//                     posicion > 0
// Postcondición:      (ptrCabecera apunta al elemento en la posicion 1)
//                     Se retorna el puntero al elemento en la posicion dada
//                     Si no hay tal posicion se retorna NULL
//                     assert() detiene la ejecución
{   nodo *ptr;
    int i;
    assert (0 < posicion);
    ptr = ptrCabecera;
    for (i = 1; (i < posicion) && (ptr != NULL); i++)
            ptr = ptr->puntero;
    return ptr;
}
```

void listaPrimeroRemover(nodo*& ptrCabecera)

```
// Precondición:       ptrCabecera es el puntero al primer elemento de una lista con por lo
menos //                       un elemento
// Postcondición:      El primer elemento está removido y devuelto al heap
//                     ptrCabecera es ahora el puntero al primer elemento de una lista más
corta
{   nodo *ptrRemover;
    ptrRemover = ptrCabecera;
    ptrCabecera = ptrCabecera->puntero;
    delete ptrRemover;
}
```

void listaRemover(nodo* ptrPrevio)

```
// Precondición:       ptrPrevio es el puntero a un elemento de una lista
//                     Este elemento no es el último
// Postcondición:      El elemento posterior al que es apuntado por ptrPrevio está removido y
//                     devuelto al heap
{   nodo *ptrRemover;
```

```
    ptrRemover = ptrPrevio->puntero;
    ptrPrevio->puntero = ptrRemover->puntero;
    delete ptrRemover;
}
```

```
void listaBorrar(nodo*& ptrCabecera)
// Precondición:     ptrCabecera es el puntero al primer elemento de una lista
// Postcondición:    Todos los elementos están removidos y devueltos al heap
//                   ptrCabecera es ahora NULL
{   while (ptrCabecera != NULL)
        listaPrimeroRemover(ptrCabecera);
}
```

```
void listaCopiar(nodo* ptrFuente, nodo*& ptrCabecera, nodo*& ptrUltimo)
{   ptrCabecera = NULL;
    ptrUltimo = NULL;
// Manejo de una lista vacía
    if (ptrFuente == NULL) return;
// Crear el nuevo nodo y poner el dato
    listaInsertarPrimero(ptrCabecera, ptrFuente->dato);
    ptrUltimo = ptrCabecera;
// Copia el resto de los nodos uno por uno agregando en la cola de la nueva lista
    for (ptrFuente = ptrFuente->puntero; ptrFuente != NULL; ptrFuente = ptrFuente->puntero)
    {       listaInsertar(ptrUltimo, ptrFuente->dato);
            ptrUltimo = ptrUltimo->puntero;
    }
}
```

```
void listaCopiarParte(nodo* ptrInicio, nodo* ptrFin, nodo*& ptrCabecera, nodo*&
ptrUltimo)
{   ptrCabecera = NULL;
    ptrUltimo = NULL;
// Manejo de una lista vacía
    if (ptrInicio == NULL) return;
// Crear el nuevo nodo y poner el dato
    listaInsertarPrimero(ptrCabecera, ptrInicio->dato);
    ptrUltimo = ptrCabecera;
    if (ptrInicio == ptrFin) return;
// Copia el resto de los nodos uno por uno agregando en la cola de la nueva lista
    for (ptrInicio = ptrInicio->puntero; ptrInicio != NULL; ptrInicio = ptrInicio->puntero)
    {       listaInsertar(ptrUltimo, ptrInicio->dato);
            ptrUltimo = ptrUltimo->puntero;
            if (ptrInicio == ptrFin) return;
    }
}
```

Operaciones sobre lista

Dibuja un diagrama de estados de UML que describa cómo funciona una lista, con todas las operaciones. Suponiendo que se tiene la lista de enteros lista, y se efectúan las siguientes operaciones:

```
lista.Anadir(1);
lista.Anadir(5);
lista.Anadir(7);
lista.Anadir(2);
lista.Anadir(3);
lista.Siguiente();
lista.Siguiente();
lista.Siguiente();
lista.Borrar();
lista.Borrar();
lista.Siguiente();
lista.Anadir(9);
lista.Borrar();
```

¿Qué devolvería la llamada lista.Consultar()? ¿Qué contenido tendría la lista? Comprueba que tus respuestas son coherentes con el comportamiento que has descrito en el diagrama

LISTAS (LISTS)

Son TAD en las que los elementos están organizados siguiendo un orden secuencial. Cada elemento tiene un anterior y un siguiente en la lista. En las listas genéricas (no así en colas o pilas, que son casos especiales de listas) no existe una restricción en la localización de los elementos insertados o extraídos. Son estructuras mucho más flexibles, donde tanto la inserción como eliminación pueden darse en cualquier posición de la lista.

```
  Inicio                                  Final (apunta a NULL)
     ------    ------    ------    --------
     | 23 |--->| 12 |--->| 11 |--->| 99 |\|
     ------    ------    ------    --------
```

Si la lista de implementara de manera estática, la eliminación o inserción de un elemento supondría la reorganización (movimiento) del resto de elementos del vector. La implementación enlazada es mucho más eficiente, y mediante ella podemos construir *Listas Simplemente Enlazadas* (cada elemento tiene acceso al siguiente), *Listas Doblemente Enlazadas* (cada elemento tiene acceso al anterior y al siguiente) o *Listas Circulares* (el último elemento tiene acceso al inicio de la lista).

Listas Simplemente Enlazadas:

```
typedef struct DatoLista
{
   TipoDatoLista dato;
   struct DatoLista *siguiente;
}  ElementoLista;

typedef struct
{
   ElementoLista *inicio;
} Lista;


void InicializaLista( Lista *lista )
{    lista->inicio = NULL;  }

int ListaVacia( Lista *lista )
{
   if ( lista->inicio == NULL )    return(1);
   return(0);
}

ElementoLista *LocalizarNodoEnLista( Lista *lista, TipoDatoLista x )
{
   ElementoLista *aux;

   aux = lista->inicio;                  /* nos ponemos al inicio de la lista
*/
   while( aux!=NULL )                    /* mientras no lleguemos al final...
*/
   {
      if( aux->dato == x )               /* comprobamos si el dato coincide
con el actual */
            return( aux );               /* si, lo devolvemos */
      aux = aux->siguiente;              /* no, vamos al siguiente elemento */
   }
   return(NULL);                         /* si llega aqui es que no lo
encontró */
}

ElementoLista* LocalizarNodoEnListaOrdenada( Lista *lista, TipoDatoLista x )
{
   ElementoLista *aux, *anterior=NULL;

   aux = lista->inicio;

   /* buscar hasta el final o hasta encontrar uno > que el buscado */

   while( aux!=NULL &amp;&amp; aux->dato<=x)
   {
      anterior = aux;
      aux = aux->siguiente;
   }

   /* si es distinto de NULL y es igual al buscado -> encontrado */

   if( aux!= NULL &amp;&amp; aux->dato == x )
     return(aux);
```

```
   /* si no, hemos encontrado el hueco donde deberia estar este dato */
  else
     return(anterior);
}


/*--------------------------------------------------------------*/
int InsertarEnLista( Lista *lista, TipoDatoLista dato,  ElementoLista *donde )
{
   ElementoLista *nuevo;

   nuevo = (ElementoLista *) malloc( sizeof(ElementoLista) );
   if( nuevo == NULL ) ErrorMemoria();

   /* copiamos los datos deseados al campo clave del nuevo nodo */
   nuevo->dato = dato;

   /* si la lista esta vacia o va al principio de la misma -> */
   if( ListaVacia(lista) || donde == NULL )
   {
      nuevo->siguiente = lista->inicio;
      lista->inicio = nuevo;
   }
   else
   {
      nuevo->siguiente = donde->siguiente;
      donde->siguiente = nuevo;
   }
   return(1);
}


/*--------------------------------------------------------------*/
ElementoLista *InsertarEnListaOrdenada( Lista *lista, TipoDatoLista dato )
{
   ElementoLista *donde;

    donde = LocalizarHuecoParaNodoEnListaOrdenada( lista, dato.nombre );
    InsertarEnLista( lista, dato, donde );
    return( donde );
}


/*--------------------------------------------------------------*/
int EliminarNodoLista( Lista *lista, ElementoLista *nodo )
{
   ElementoLista *aux;
   if( ListaVacia( lista ) || nodo == NULL )
      return(0);

  /* comprobamos si nodo es el inicio de la lista */
   if( nodo == lista->inicio )
   {

      /* si lo es, lo eliminamos y ponemos inicio=siguiente) */
      lista->inicio = (lista->inicio)->siguiente;
      free( nodo );
   }
```

```
  /* en caso normal, localizamos el elemento anterior y borramos */
  else
   {
      aux = lista->inicio;
      while( aux->siguiente != nodo  &amp;&amp;  aux->siguiente!= NULL)
         aux = aux->siguiente;

      /* si no encontramos el elemn anterior, devolver error() */
      if( aux->siguiente==NULL )
         return(2);

     /* eliminar el nodo especificado */
      aux->siguiente = nodo->siguiente;
      free(nodo);
   }
   return(1);
}
```

Unidad 4 Recursividad.

4.1 Definición.

El área de la programación es muy amplia y con muchos detalles. Los programadores necesitan ser capaces de resolver todos los problemas que se les presente a través del computador aun cuando en el lenguaje que utilizan no haya una manera directa de resolver los problemas. En el lenguaje de programación C, así como en otros lenguajes de programación, se puede aplicar una técnica que se le dio el nombre de recursividad por su funcionalidad. Esta técnica es utilizada en la programación estructurada para resolver problemas que tengan que ver con el factorial de un número, o juegos de lógica. Las asignaciones de memoria pueden ser dinámicas o estáticas y hay diferencias entre estas dos y se pueden aplicar las dos en un programa cualquiera.

La recursividad es una técnica de programación importante. Se utiliza para realizar una llamada a una función desde la misma función. Como ejemplo útil se puede presentar el cálculo de números factoriales. Él factorial de 0 es, por definición, 1. Los factoriales de números mayores se calculan mediante la multiplicación de 1 * 2 * ..., incrementando el número de 1 en 1 hasta llegar al número para el que se está calculando el factorial.

El siguiente párrafo muestra una función, expresada con palabras, que calcula un factorial.

"Si el número es menor que cero, se rechaza. Si no es un entero, se redondea al siguiente entero. Si el número es cero, su factorial es uno. Si el número es mayor que cero, se multiplica por él factorial del número menor inmediato."

Para calcular el factorial de cualquier número mayor que cero hay que calcular como mínimo el factorial de otro número. La función que se utiliza es la función en la que se encuentra en estos momentos, esta función debe llamarse a sí misma para el número menor inmediato, para poder ejecutarse en el número actual. Esto es un ejemplo de recursividad.

La recursividad y la iteración (ejecución en bucle) están muy relacionadas, cualquier acción que pueda realizarse con la recursividad puede realizarse con iteración y viceversa. Normalmente, un cálculo determinado se prestará a una técnica u otra, sólo necesita elegir el enfoque más natural o con el que se sienta más cómodo.

Claramente, esta técnica puede constituir un modo de meterse en problemas. Es fácil crear una función recursiva que no llegue a devolver nunca un resultado definitivo y no pueda llegar a un punto de finalización. Este tipo de recursividad hace que el sistema ejecute lo que se conoce como bucle "infinito".

Para entender mejor lo que en realidad es el concepto de recursión veamos un poco lo referente a la **secuencia de Fibonacci.**

Principalmente habría que aclarar que es un ejemplo menos familiar que el del factorial, que consiste en la secuencia de enteros.

0,1,1,2,3,5,8,13,21,34,...,

Cada elemento en esta secuencia es la suma de los precedentes (por ejemplo 0 + 1 = 0, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, ...) sean *fib*(0) = 0, *fib* (1) = 1 y así sucesivamente, entonces puede definirse la secuencia de Fibonacci mediante la **definición recursiva** (define un objeto en términos de un caso mas simple de si mismo):

fib (n) = n if n = = 0 or n = = 1

fib (n) = fib (n - 2) + fib (n - 1) if n >= 2

Por ejemplo, para calcular *fib* (6), puede aplicarse la definición de manera recursiva para obtener:

> Fib (6) = fib (4) + fib (5) = fib (2) + fib (3) + fib (5) = fib (0) + fib (1) + fib (3) + fib (5) = 0 + 1 fib (3) + fib (5)

1. + fib (1) + fib (2) + fib(5) =

1. + 1 + fib(0) + fib (1) + fib (5) =
2. + 0 + 1 + fib(5) = 3 + fib (3) + fib (4) =
3. + fib (1) + fib (2) + fib (4) =

   3 + 1 + fib (0) + fib (1) + fib (4) =

4. + 0 + 1 + fib (2) + fib (3) = 5 + fib (0) + fib (1) + fib (3) =
5. + 0 + 1 + fib (1) + fib (2) = 6 + 1 + fib (0) + fib (1) =
6. + 0 + 1 = 8

Obsérvese que la definición recursiva de los números de Fibonacci difiere de las definiciones recursivas de la función factorial y de la multiplicación . La definición recursiva de *fib* se refiere dos veces a sí misma . Por ejemplo, *fib* (6) = *fib* (4) + *fib* (5), de tal manera que al calcular *fib* (6), *fib* tiene que aplicarse de manera recursiva dos veces. Sin embargo calcular *fib* (5) también implica calcular *fib* (4), así que al aplicar la definición hay mucha redundancia de cálculo. En ejemplo anterior, *fib*(3) se calcula tres veces por separado. Sería mucho mas eficiente "recordar" el valor de *fib*(3) la primera vez que se calcula y volver a usarlo cada vez que se necesite. Es mucho mas eficiente un método iterativo como el que sigue parar calcular *fib* (n).

If (n < = 1)

return (n);

lofib = 0 ;

hifib = 1 ;

for (i = 2; i < = n; i ++)

```
{

x = lofib ;

lofib = hifib ;

hifib = x + lofib ;

} /* fin del for*/

return (hifib) ;
```

Compárese el numero de adiciones (sin incluir los incrementos de la variable índice, i) que se ejecutan para calcular *fib* (6) mediante este algoritmo al usar la definición recursiva. En el caso de la función factorial, tienen que ejecutarse el mismo numero de multiplicaciones para calcular *n!* Mediante ambos métodos: recursivo e iterativo. Lo mismo ocurre con el numero de sumas en los dos métodos al calcular la multiplicación. Sin embargo, en el caso de los números de Fibonacci, el método recursivo es mucho mas costoso que el iterativo.

4.2 Procedimientos recursivos.

Procedimientos recursivos

Cuando se definen grupos de uno o más procedimientos mutuamente recursivos (por ejemplo, cuando el procedimiento `p` llama al procedimiento `q` en su cuerpo y vice-versa), cada llamada a un procedimiento del grupo es considerada como una llamada recursiva. Los procedimientos recursivos deben tener asociados una función de cota. Dentro del cuerpo de los procedimientos, además del cumplimiento de la precondición, es necesario que las llamadas recursivas, hagan disminuir la cota, sin que ésta llegue a ser menor que cero.

La cota es únicamente utilizada para demostrar la terminación de las definiciones de los porcedimientos. Las llamadas a procedimientos definidos recursivamente, ubicadas fuera de las definiciones de éstos no necesitan condición alguna sobre la cota para estar definidas.

Corrección de llamadas recursivas

4. Sin parámetros formales de salida

Sean `n` tal que `(0<=n)` y `p` un procedimiento cuya definición es de la forma siguiente:

```
proc p : (in x1:t1,...,in xn:tn) {pre Pre_p} {post Post_p} {bount T_p}...
```

La corrección de una llamada a `p` en las acciones de su cuerpo viene definida por:

```
[pmdi(p(e1,...,en),Q) ==
  def(e1) /\ ... /\ def(en) /\
  Pre_p(x1,...,xn := e1,...,en) /\
  0 <= T_p(x1,...,xn := e1,...,en) < T_p /\
  (Post_p(x1,...,xn := e1,...,en) ==> Q)].
```

5. Con un parámetro `out`

Sean `m,n` tales que `(0<=m<=n)` y `p` un procedimiento cuya definición es de la forma siguiente:

```
proc p : (in x1:t1,...,in xm:tm, out o:=ini, ...,in xn:tn)
{pre Pre_p} {post Post_p} ...
```

Sea `C` el nombre de una constante que no aparece en el programa.

La corrección de una llamada a `p` en las acciones de su cuerpo viene definida por:

```
[pmdi(p(e1,...,en,a),Q) ==
  def(e1) /\ ... /\ def(en) /\
  Pre_p(x1,...,xn,o := e1,...,en,ini) /\
  0 <= T_p(x1,...,xn := e1,...,en) < T_p /\
  (Post_p(x1,...,xn,o := e1,...,en,C) ==> Q(a:=C))].
```

6. Con un parámetro `in out`

Sean `m,n` tales que `(0<=m<=n)` y `p` un procedimiento cuya definición es de la forma siguiente:

```
proc p : (in x1:t1,...,in xm:tm, in out o,...,in xn:tn)
{pre Pre_p} {post Post_p} ...
```

Sea `C` el nombre de una constante no utilizada en el programa.

La corrección de una llamada a `p` en las acciones de su cuerpo viene definida por:

```
[pmdi(p(e1,...,en,a),Q) ==
  def(e1) /\ ... /\ def(en) /\
  Pre_p(x1,...,xn,o := e1,...,en,a) /\
  0 <= T_p(x1,...,xn,o' := e1,...,en,a) < T_p /\
  (Post_p(x1,...,xn,o',o := e1,...,en,a,C) ==> Q(a:=C))].
```

4.3 Mecánica de recursión.

Recursión es la forma en la cual se especifica un proceso basado en su propia definición. Siendo un poco más precisos, y para evitar el aparente círculo sin fin en esta definición, las instancias *complejas* de un proceso se definen en términos de instancias más *simples*, estando las **finales** más simples definidas de forma explícita.

Funciones definidas de forma recursiva

Aquellas funciones cuyo dominio puede ser recursivamente definido pueden ser definidas de forma recursiva.

El ejemplo más conocido es la definición recursiva de la función factorial $f(n)$:

$f(0) = 1$
$f(n) = n \cdot f(n\text{-}1)$ para todo número natural $n > 0$

Con esta definición veamos como funciona esta función para el valor del factorial de 3:

```
f(3) = 3 · f(3-1)
     = 3 · f(2)
     = 3 · 2 · f(2-1)
     = 3 · 2 · f(1)
     = 3 · 2 · 1 · f(1-1)
     = 3 · 2 · 1 · f(0)
     = 3 · 2 · 1 · 1
     = 6
```

[editar]

Algoritmo recursivo

Un método usual de simplificación de un problema complejo es la división de este en subproblemas del mismo tipo. Esta técnica de programación se conoce como divide y vencerás y es el núcleo en el diseño de númerosos algoritmos de gran importancia, así como también es parte fundamental de la programación dinámica.

[editar]

Ejemplos

Resolución de ecuaciones homogéneas de primer grado, segundo orden:

a) Se pasan al primer miembro los términos an, an-1, an-2, los cuales también podrían figurar como an+2, an+1, an b)Se reemplaza an por r^2, an-1 por r y an-2 por 1, quedando una ecuación de segundo grado con raíces reales y distintas de r1 y r2. c)Se plantea a = u · r1n + v · r2n

e)Debemos tener como dato los valores de los dos primeros términos de la sucesión: A0 = k y A1 = k'. Utilizando estos datos ordenamos el sistema de 2x2: u + v = k y u·r1 + u·r2 = k'. La resolución de este sistema no da como resultado los valores u0 y v0, que son números reales conocidos.

e)La solución general es:

an = u0 · r1n + v0 · r2n

Algunos ejemplos de recursión:

- Factorial -- $n! = n \times (n-1)!$
- Sucesión de Fibonacci -- $f(n) = f(n-1) + f(n-2)$
- Números de Catalan -- $C(2n, n)/(n+1)$
- Las Torres de Hanoi
- Función de Ackermann

Podemos definir la recursividad como un proceso que se define en términos de sí mismo.

El concepto de recursión es difícil de precisar, pero existen ejemplos de la vida cotidiana que nos pueden servir para darnos una mejor idea acerca de lo que es recursividad. Un ejemplo de esto es cuando se toma una fotografía de una fotografía, o cuando en un programa de televisión un periodista transfiere el control a otro periodista que se encuentra en otra ciudad, y este a su vez le transfiere el control a otro.

Casos típicos de estructuras de datos definidas de manera recursiva son los árboles binarios y las listas enlazadas.

La recursión se puede dar de dos formas:

- *DIRECTA.* Este tipo de recursión se da cuando un subprograma se llama directamente a sí mismo.
- *INDIRECTA* Sucede cuando un subprograma llama a un segundo subprograma, y este a su vez llama al primero, es decir el subproceso A llama al B, y el B invoca al subproceso A.

Implementar Recursión Usando Pilas

Otra de las aplicaciones en las que podemos utilizar las pilas es en la implementación de la recursividad. A continuación se mostrarán algunos ejemplos.

```
                |
                |      1  ,                    N=0
Factorial     <
                |      N*(n-1)!,      N&gt0
                |
```

```
sp <--0
mientras n <> 1 haz
         push(pila,n)
         n<--n-1
mientras sp <> 0 haz
         factorial<--factorial*pop(pila)
```

```
                |
                |      0   ,                 si a < b
Q             <
                |   Q(a-b,b)+1,   si a<=b
                |
```

```
sp<--0
Q<--0
lee(a), lee(b)
mientras a>=b haz
        push(pila,1)
        a<--a-b
mientras sp< > 0 haz
        Q<-- Q + pop(pila)
```

En Pacal, a un procedimiento o función le es permitido no sólo invocar a otro procedimiento o función, sino también invocarse a sí mismo. Una invocación de éste tipo se dice que es *recursiva*.

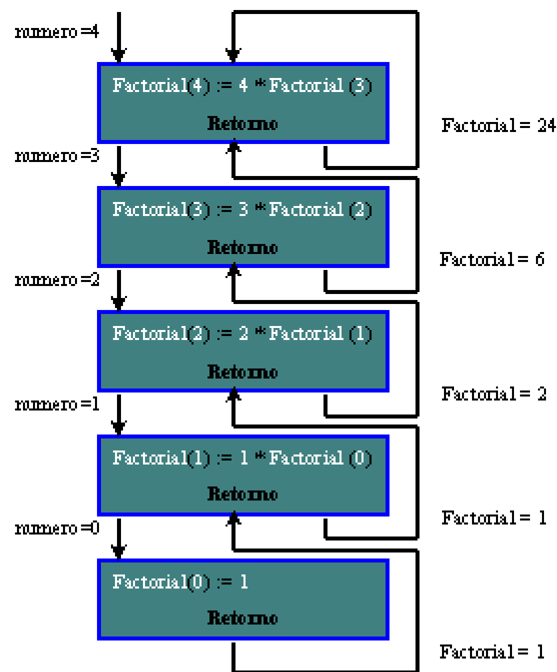
La *función recursiva* más utilizada como ejemplo es la que calcula el factorial de un número entero no negativo, partiendo de las siguientes definiciones :

```
factorial (0) = 1
factorial (n) = n*factorial(n-1), para n>0
```

La función, escrita en Pascal, queda de la siguiente manera :

```
function factorial(numero:integer):integer;
begin
  if numero = 0 then
    factorial := 1
  else
    factorial := numero * factorial(numero-1)
end;
```

Si **numero** = **4**, la función realiza los siguientes pasos :



1. **factorial(4) := 4 * factorial(3) Se invoca a si misma y crea una segunda variable cuyo nombre es numero y su valor es igual a 3.**

2. **factorial(3) := 3 * factorial(2) Se invoca a si misma y crea una tercera variable cuyo nombre es numero y su valor es igual a 2.**

3. **factorial(2) := 2 * factorial(1) Se invoca a si misma y crea una cuarta variable cuyo nombre es numero y su valor es igual a 1.**

4. **factorial(1) := 1 * factorial(0) Se invoca a si misma y crea una quinta variable cuyo nombre es numero y su valor es igual a 0.**

5. **Como factorial(0) := 1, con éste valor se regresa a completar la invocación: factorial(1) := 1 * 1 , por lo que factorial(1) := 1**

6. **Con factorial(1) := 1, se regresa a completar: factorial(2) := 2 * 1, por lo que factorial(2) := 2**

7. **Con factorial(2) := 2, se regresa a completar : factorial(3) := 3 * 2, por lo que factorial(3) := 6**

8. **Con factorial(3) := 6, se regresa a completar : factorial(4) := 4 * 6, por lo que factorial(4) := 24 y éste será el valor que la función factorial devolverá al módulo que la haya invocado con un valor de parámetro local igual a 4 .**

Un ejemplo de procedimiento recursivo es el siguiente:

Supóngase que una persona se mete a una piscina cuya profundidad es de 5 metros. Su intención es tocar el fondo de la piscina y después salir a la superficie. Tanto en el descenso como en el ascenso se le va indicando la distancia desde la superficie (a cada metro).

```
Program Piscina;
Uses Crt;
Const
  prof_max = 5;
Var
  profundidad:integer;
procedure zambullida(Var profun :integer);
begin
  WriteLn('BAJA 1 PROFUNDIDAD = ',profun);
  profun := profun + 1;
  if profun <= prof_max then
    zambullida(profun)
  else
    WriteLn;
  profun := profun - 1;
  WriteLn('SUBE 1 PROFUNDIDAD = ', profun-1)
end;
begin
  ClrScr;
  profundidad := 1;
  zambullida(profundidad)
end.
```

Algunos ejemplos simples

En este capítulo describiremos algunos de los ejemplos más simples que podamos imaginar de algoritmos recursivos.

2.1 La función factorial

Alguna vez te habrás preguntado para qué sirve la tecla **n!** de tu calculadora. Esta tecla calcula lo que se llama el *factorial de n*. Esta función se define de la forma siguiente: Si n es igual a 0, entonces n! es igual a 1. Si n es mayor que 0, entonces n! es igual al producto de (n-1)! y n. Como puedes ver, definimos n! en términos de (n-1)!, que a su vez está definido en términos de ((n-1)-1)!, que a su vez... ¿Cuándo salimos del círculo vicioso? ¡Exacto! Después de n pasos, n! está definido en términos de 0!, pero como sabemos que 0! es igual a 1, no hay necesidad de continuar. Hagamos un ejemplo:

4! = 4*3! = 4*(3*2!) = 4*(3*(2*1!)) = 4*(3*(2*(1*0!))) = 4*(3*(2*(1*1))) = 4*(3*(2*1)) = 4*(3*2) = 4*6 = 24.

Por cierto, el factorial de n cuenta el número de *permutaciones* de n objetos distintos colocados a lo largo de una línea recta.

A continuación mostramos dos funciones recursivas, una en C y la otra en Pascal, para calcular el factorial de un entero.

C	Pascal

`int factorial(int n)`
`{`
`  if (n == 0) return 1;`
`  else return n*factorial(n-1);`
`}`
	`function factorial(n : integer) : integer`
	`begin`
	`  if n = 0 then factorial := 1`
	`  else factorial := n*factorial(n-1)`
	`end`

Nota que ninguna de estas dos funciones calcula el factorial de un número entero muy grande. Usando tu compilador de C o Pascal, determina cual es el valor más grande de n para el cual `factorial(n)` calcula correctamente el valor de n! ¿Cómo podrías aumentar este valor?

2.2 Los conejos de Fibonacci

Cierto matemático italiano de nombre Leonardo de Pisa, pero mejor conocido como *Fibonacci*, propuso el siguiente problema: Suponga que acabamos de comprar una pareja de conejos adultos. Al cabo de un mes, esa pareja tiene una pareja de conejitos (un conejo y una coneja). Un mes después, nuestra primera pareja tiene otra pareja de conejitos (nuevamente, un conejo y una coneja) y, al mismo tiempo, sus primeros hijos se han vuelto adultos. Así que cada mes que pasa, cada pareja de conejos adultos tiene una pareja de conejitos, y cada pareja de conejos nacida el mes anterior se vuelve adulta. La pregunta es, ¿cuántas parejas de conejos adultos habrá al cabo de n meses? Para resolver este problema, llamemos F_n al número de parejas adultas al cabo de n meses. No es difícil convencerse de que si n es al menos 2, entonces F_n es igual a $F_{n-1} + F_{n-2}$. ¿Porqué? Así que F_n queda en términos de F_{n-1} y F_{n-2}, que a su vez quedan en términos de F_{n-2}, F_{n-3} y F_{n-4}, que a su vez... Ahora salimos del círculo vicioso recordando que al principio había una pareja de conejos adultos, la misma que había al final del primer mes, así que $F_0 = F_1 = 1$. Hagamos un ejemplo:

$F_4 = F_3 + F_2 = (F_2 + F_1) + (F_1 + F_0) = ((F_1 + F_0) + 1) + (1 + 1) = ((1 + 1) + 1) + 2 = (2 + 1) + 2 = 3 + 2 = 5$.

Por si te lo preguntabas, la sucesión de números $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$, etc. recibe el nombre de *sucesión de Fibonacci*.

A continuación mostramos dos funciones recursivas, una en C y otra en Pascal, que resuelven el problema de Fibonacci.

C	Pascal

`int fib(int n)`
`{`
`  if ((n == 0) || (n == 1)) return 1;`
`  else return fib(n-1) + fib(n-2);`
`}`
	`function fib(n : integer) : integer`
	`begin`
	`  if (n = 0) or (n = 1) then fib := 1`
	`  else fib := fib(n-1) + fib(n-2)`
	`end`

Nota de nuevo que ninguna de estas dos funciones calcula el número de parejas de conejos para

un número de meses muy grande. Usando tu compilador de C o Pascal, determina cual es el valor más grande de n para el cual `fib(n)` calcula correctamente el valor de F_n. ¿Cómo podrías aumentar este valor?

2.3 El triángulo de Pascal

En algún momento de tu vida habrás aprendido que $(x + z)^2 = x^2 + 2xz + z^2$, que $(x + z)^3 = x^3 + 3x^2z + 3xz^2 + z^3$ y, en general, para cualquier entero positivo n, a calcular los coeficientes de $(x + z)^n$, y posiblemente le diste el nombre de ${}_nC_m$ al coeficiente de $x^m z^{n-m}$. Seguramente recuerdas la siguiente tabla triangular:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
. . . . . .

Esa tabla se conoce como *triángulo de Pascal* y se construye como sigue: Al principio se coloca un 1 (que corresponde con ${}_0C_0$). Para cada renglón subsecuente, digamos para el renglón n, se coloca un 1 a la izquierda y un 1 a la derecha (que corresponden con ${}_nC_0$ y ${}_nC_n$, respectivamente) y los elementos restantes se calculan sumando los dos números que tiene justo arriba a la izquierda y arriba a la derecha, es decir, ${}_nC_m = {}_{n-1}C_{m-1} + {}_{n-1}C_m$ para toda $0 < m < n$.

Entre otras cosas, el número ${}_nC_m$ cuenta la cantidad de formas de escoger m objetos de un total de n objetos distintos. A estos números también se les llama las *combinaciones* de m objetos en n.

A continuación mostramos dos funciones recursivas, una en C y otra en Pascal, que calculan el número de combinaciones de m objetos en n.

C

```
int comb(int n, int m)
{
  if ((n == 0) || (n == m))
return 1;
  else return comb(n-1,m-1) +
comb(n-1,m);
}
```

Pascal

```
function comb(n, m : integer) :
integer
begin
  if (n = 0) or (n = m) then comb
:= 1
  else comb := comb(n-1,m-1) +
comb(n-1,m)
end
```

4.4 Transformación de algoritmos recursivos a iterativos.

4.5 Recursividad en el diseño.

Recursividad

Programación II trata del diseño y análisis formal de algoritmos abordándolo desde el punto de vista recursivo y el iterativo. La formación básica en programación (asignatura PROGRAMACIÓN I, por ejemplo) proporciona conceptos básicos de programación imperativa: secuencia de órdenes, sentencias condicionales, bucles. Usando el típico ejemplo del factorial, se propone como ejercicio el desarrollo de un procedimiento o función que calcule el factorial de un número no negativo (el factorial **!n** de un número **n** no negativo es igual a producto de todos los números no negativos iguales o inferiores que **n** teniendo en cuenta que el factorial de **0** es **1** por definición). Intuitivamente alguien con experiencia en MODULA-2 programaría la siguiente función para cálculo de factorial:

```
PROCEDURE Factorial(n : CARDINAL) : CARDINAL
BEGIN
  VAR Resultado,i : CARDINAL ;
  Resultado :=1 ;
  FOR i :=1 TO n DO
    Resultado :=Resultado*i ;
  END ;
  RETURN Resultado
END Factorial ;
```

Existe una vía alternativa para abordar el diseño de algoritmos: la recursiva. Sin embargo la mayoría de las personas que abordan la asignatura PROGRAMACION II por primera vez no han asimilado el concepto de recursividad. El objeto de este apartado es la introducción de ese concepto, dada su importancia para el dominio de la asignatura.

Una aproximación diferente a la programación de una función que calcule el factorial de un número estaría basada en el siguiente razonamiento:

- El factorial de **0** es **1**

- El factorial de un número natural n, mayor que cero, es igual al producto de n por el factorial del número **n-1**, es decir: **! n=n* !(n-1)**

Siguiendo este razonamiento se puede llegar a programar el siguiente algoritmo en MODULA-2:

```
PROCEDURE Factorial(n: CARDINAL): CARDINAL;
BEGIN
  IF n=0 THEN
     RETURN 1
  ELSE
     RETURN n* Factorial(n-1)
  END
END Factorial;
```

La primera sorpresa reside en el hecho de que en la línea remarcada se ejecute una llamada a la propia función Factorial. ¿ Es esto posible ?. Conceptualmente si. En la práctica muchos lenguajes de alto nivel permiten la recursividad: Pascal, C, MODULA-2, etc.

En el algoritmo iterativo la solución parece clara: se utiliza una variable en la que se va acumulando el producto de los **n** números. ¿ Cómo funciona el procedimiento recursivo?. Supongamos que se desea calcular el factorial del número 4. En el programa principal se ejecutará un sentencia análoga a:

```
Resultado:= Factorial(4);
```

En esta llamada, y puesto que 4 es distinto de 0, se ejecutará una segunda llamada a Factorial con parámetro 4-1=3

En la segunda llamada se efectuará una tercera llamada a Factorial con parámetro 3-1=2.

En la tercera llamada se efectuará una cuarta llamada a Factorial con parámetro 2-1=1

En la cuarta llamada se efectuará una quinta llamada a Factorial con parámetro 1-1=0

En esta llamada se cumple la condición n=0, por lo tanto se retornará a la cuarta llamada el resultado 1.

La cuarta llamada devolverá a la tercera el resultado 1*1=1

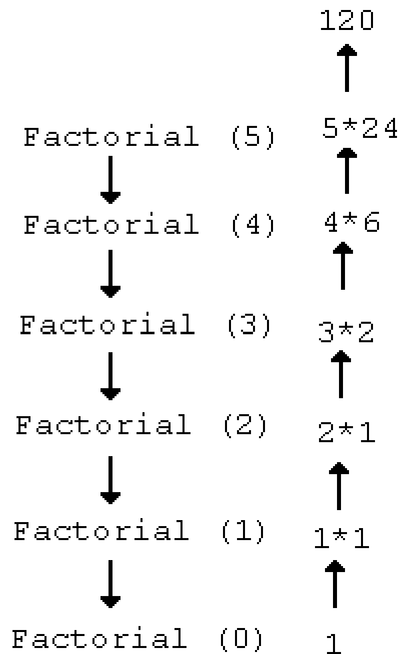
La tercera llamada devolverá a la segunda el resultado 2*1=2

La segunda llamada devolverá a la primera el resultado 3*2=6

La primera llamada devolverá al programa principal el resultado final 4*6=24

Cada llamada supone una nueva ejecución del procedimiento Factorial, en cada llamada se ha de mantener un contexto de ejecución diferente (parámetros, variables locales, punto de retorno), los programas compilados a partir de lenguajes de alto nivel que soportan la recursividad gestionan este proceso de forma transparente al programador.

Esquemáticamente la ejecución de Factorial(5) se resuelve con el siguiente proceso:



El diseño y análisis recursivo de algoritmos es una técnica muy potente una vez dominada. Muchos algoritmos tienen una solución recursiva natural. En muchos casos la programación recursiva permite resolver problemas de forma elegante, con menos código y más fácil de entender. Sin embargo la necesidad de gestionar la pila en la que se suelen mantener los contextos de ejecución independientes hace que la solución recursiva resulte algo menos eficiente, en general, en cuanto a tiempo de ejecución que la iterativa. Siempre es posible obtener la solución iterativa equivalente a una recursiva dada, aunque en muchos casos el programa resultará menos claro y compacto. Especialmente cuando el algoritmo tiene una solución recursiva natural. Se presenta un ejemplo industrial en la NOTA.

Para que una solución recursiva sea correcta se han de cumplir una serie de reglas. Estas reglas son materia importante en el estudio del Capítulo 4 de Peña. Se introducirán los criterios fundamentales utilizando el ejemplo del Factorial:

- Siempre ha de existir una (o varias) condiciones para los parámetros de entrada que permitan devolver un resultado sin necesidad de nueva llamada recursiva. Esta es la llamada ***solución trivial*** al problema. En el caso tratado cuando n=0 el resultado inmediato es 1 y no hay llamada recursiva

- Cuando los valores de los parámetros de entrada no cumplen la condición de solución trivial se volverá a llamar recursivamente el mismo procedimiento (***solución o soluciones no triviales***). En todas las llamadas recursivas que se realicen el valor del parámetro de llamada se ha de modificar de forma que se "aproxime" al valor de solución trivial. En el caso tratado en la llamada recursiva se utiliza n-1 en lugar de n, n-1 está "más próximo" al valor trivial 0 que n. (El concepto intuitivo de "proximidad" utilizado se define rigurosamente en base a la teoría de inducción noetheriana en el capítulo 4 de Peña)

- La solución retornada en el caso de condición no trivial ha de ser correcta en la hipótesis de que la llamada recursiva devuelve una solución correcta. Este concepto es la base del **principio de inducción** y la parte más importante en el diseño y verificación de un algoritmo recursivo. Para el caso tratado: en la hipótesis de que la llamada **Factorial (n-1)** devuelve una solución correcta, es decir devuelve **! (n-1)**, el resultado final es correcto puesto que se devuelve finalmente el valor :

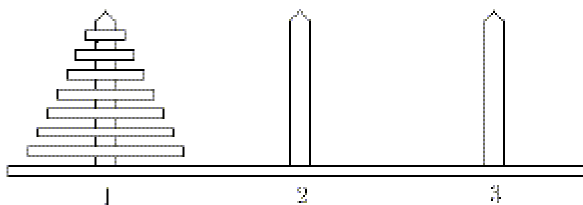
n* Factorial(n-1)= n* ! (n-1)= ! n.

Es importante empezar cuanto antes a razonar en recursivo buscando la aproximación recursiva a problemas simples. Algunos ejemplos:

- La suma de una secuencia de n números puede abordarse de forma iterativa mediante una variable de acumulación inicializada a cero en la que sucesivamente se van sumando los números. Un planteamiento recursivo sería: la suma de una secuencia de n números se puede obtener: si n es mayor que 1, como suma del primero con el resultado de la suma de la secuencia de n-1 números dada por el segundo y siguientes. Si n no es mayor que 1 la suma is igual al valor del elemento único de la secuencia.

- La búsqueda de una palabra en un diccionario se realizaría, por aproximación iterativa, mirando sucesivamente palabras desde la primera página hasta encontrar la palabra buscada u otra que sea posterior a ella en orden alfabético (en este caso la palabra no está en el diccionario). Una solución recursiva sería: abrir el diccionario por la mitad, si la primera palabra de la página es posterior a la buscada abrir el diccionario por la mitad de la mitad posterior y repetir el proceso, si no es así abrirlo por la mitad de la primera mitad y repetir el proceso .... Esta es la forma natural que usamos para buscar palabras en un diccionario.

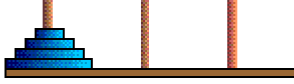
- Torres de Hanoi:



la tarea planteada a los monjes del templo de Bramah es mover n discos concéntricos de distinto diámetro desde una aguja 1 a otra aguja 3 usando como almacén temporal la aguja 2 si es necesario. No puede mover más de un disco a la vez y nunca ha de permanecer un disco de mayor diámetro sobre otro de diámetro inferior.¿ Quién encuentra una solución iterativa a este problema ?. Sin embargo la solución recursiva es más abordable: mover n discos entre la aguja 1 y la 3 es posible moviendo n-1 discos entre la 1 y la 2, moviendo el disco mayor desde la 1 a la 3 y moviendo luego los n-1 discos desde la 2 a la 3. En general mover n discos desde la aguja i a la j puede conseguirse siguiendo el siguiente procedimiento en pseudocódigo:

```
Procedimiento Hanoi(n, i, j)
  Si n=1 entonces
    mover disco de i a j
```

```
     Si no entonces
       Hanoi(n-1,i,6-i-j)
       mover disco de i a j
       Hanoi(n-1,6-i-j,j)
     Fin Si
   Fin Hanoi
Así si ejecutamos Hanoi(4,1,3) obtenemos:
```



Si quieres saber más sobre las Torres de Hanoi, ver applets demostrativos etc. una página imprescindible es:

http://www.pangea.ca/kolar/javascript/Hanoi/HTonWebE.html

y uno de los mejores applets en:

http://www.eng.auburn.edu/~fwushan/Hanoi1.html

NOTA. La transformada Rápida de Fourier

Posiblemente unos de los algoritmos con más trascendencia industrial ha sido el de Transformada Rápida de Fourier (Fast Fourier Transform o FFT). Ha permitido el cálculo rápido de transformadas de Fourier en aplicaciones relacionadas con la óptica, acústica, física cuántica, telecomunicaciones, teoría de sistemas y procesamiento de señales, reconocimiento del habla etc. Fue descubierto por Cooley y Turkey en 1965. La programación de FFTs en su forma más eficiente en cuanto a tiempo de ejecución da lugar a algoritmos iterativos más o menos complejos y muchas veces difíciles de leer. Sin embargo, cuando lo que prima es la claridad del código fuente, la codificación recursiva resulta más simple y clara. La FFT es un algoritmo naturalmente recursivo ya que el cálculo de la transformada de N puntos se basa en la combinación de las transformadas de dos secuencias de N/2 puntos. Una solución programada en C es (se presenta sólo a título de ejemplo, no es necesario comprenderla):

```
void fft(double *x, double *y, int izda, int dcha)
{
  double vr,vi,wr,wi;  /* partes real e imaginaria de v y w */
  int medio,i,j;
  int n;
  double fase, incfase;

  if(izda<dcha) {
    /* cálculo de las FFTs de las subsecuencias de N/2 puntos */
    medio=(izda+dcha)/2;
    fft(x,y,izda,medio);
    fft(x,y,medio,dcha);
    /* combinación del resultado */
    n=dcha-izda+1;
    incfase=2* M_PI/n;
    for(i=izda; i<=medio; i++) {
      j=i+n/2;
      fase=(i-izda)*incfase;
      vr=x[i];
      vi=y[i];
```

```
      wr=x[j]*cos(fase)+y[j]*sin(fase);
      wi=-x[j]*sin(fase)*y[j]*cos(fase);
      x[i]=vr+wr;
      y[i]=vi+wi;
      x[j]=vr-wr;
      y[j]=vi-wi;
    }
  }
  return;
}
```

Un programa más claro y simple todavía se obtiene en entornos que tienen definido o permiten definer el tipo ***número complejo*** (C++, Java, Fortran, Ada ...)

4.6 Complejidad de los algoritmos recursivos

Introducción

La resolución práctica de un problema exige por una parte un algoritmo o método de resolución y por otra un programa o codificación de aquel en un ordenador real. Ambos componentes tienen su importancia; pero la del algoritmo es absolutamente esencial, mientras que la codificación puede muchas veces pasar a nivel de anécdota.

A efectos prácticos o ingenieriles, nos deben preocupar los recursos físicos necesarios para que un programa se ejecute. Aunque puede haber muchos parametros, los mas usuales son el tiempo de ejecución y la cantidad de memoria (espacio). Ocurre con frecuencia que ambos parametros están fijados por otras razones y se plantea la pregunta inversa: ¿cual es el tamano del mayor problema que puedo resolver en T segundos y/o con M bytes de memoria? En lo que sigue nos centraremos casi siempre en el parametro tiempo de ejecución, si bien las ideas desarrolladas son fácilmente aplicables a otro tipo de recursos.

Para cada problema determinaremos un medida N de su tamaño (por número de datos) e intentaremos hallar respuestas en función de dicho N. El concepto exacto que mide N depende de la naturaleza del problema. Así, para un vector se suele utizar como N su longitud; para una matriz, el número de elementos que la componen; para un grafo, puede ser el número de nodos (a veces es mas importante considerar el número de arcos, dependiendo del tipo de problema a resolver); en un fichero se suele usar el número de registros, etc. Es imposible dar una regla general, pues cada problema tiene su propia lógica de coste.

2. Tiempo de Ejecución

Una medida que suele ser útil conocer es el tiempo de ejecución de un programa en función de N, lo que denominaremos T(N). Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción. Así, un trozo sencillo de programa como

```
  S1; for (int i= 0; i < N; i++) S2;
```

requiere

```
  T(N)= t1 + t2*N
```

siendo t1 el tiempo que lleve ejecutar la serie "S1" de sentencias, y t2 el que lleve la serie "S2".

Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten. Esto hace que mas que un valor T(N) debamos hablar de un rango de valores

```
Tmin(N)  <=  T(N)  <=  Tmax(N)
```

los extremos son habitualmente conocidos como "caso peor" y "caso mejor". Entre ambos se hallara algun "caso promedio" o más frecuente.

Cualquier fórmula T(N) incluye referencias al parámetro N y a una serie de constantes "Ti" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del ordenador que lo ejecuta. Dado que es fácil cambiar de compilador y que la potencia de los ordenadores crece a un ritmo vertiginoso (en la actualidad, se duplica anualmente), intentaremos analizar los algoritmos con algun nivel de independencia de estos factores; es decir, buscaremos estimaciones generales ampliamente válidas.

3. Asintotas

Por una parte necesitamos analizar la potencia de los algoritmos independientemente de la potencia de la máquina que los ejecute e incluso de la habilidad del programador que los codifique. Por otra, este análisis nos interesa especialmente cuando el algoritmo se aplica a problema grandes. Casi siempre los problemas pequeños se pueden resolver de cualquier forma, apareciendo las limitaciones al atacar problemas grandes. No debe olvidarse que cualquier técnica de ingeniería, si funciona, acaba aplicándose al problema más grande que sea posible: las tecnologias de éxito, antes o después, acaban llevándose al límite de sus posibilidades.

Las consideraciones anteriores nos llevan a estudiar el comportamiento de un algoritmo cuando se fuerza el tamaño del problema al que se aplica. Matemáticamente hablando, cuando N tiende a infinito. Es decir, su comportamiento asintótico.

Sean "g(n)" diferentes funciones que determinan el uso de recursos. Habra funciones "g" de todos los colores. Lo que vamos a intentar es identificar "familias" de funciones, usando como criterio de agrupación su comportamiento asintótico.

A un conjunto de funciones que comparten un mismo comportamiento asintótico le denominaremos un órden de complejidad'. Habitualmente estos conjuntos se denominan O, existiendo una infinidad de ellos.

Para cada uno de estos conjuntos se suele identificar un miembro f(n) que se utiliza como representante de la clase, hablándose del conjunto de funciones "g" que son del orden de "f(n)", denotándose como

```
g IN O(f(n))
```

Con frecuencia nos encontraremos con que no es necesario conocer el comportamiento exacto, sino que basta conocer una cota superior, es decir, alguna función que se comporte "aún peor".

La definición matemática de estos conjuntos debe ser muy cuidadosa para involucrar ambos aspectos: identificación de una familia y posible utilización como cota superior de otras funciones menos malas:

Dícese que el conjunto O(f(n)) es el de las funciones de orden de f(n), que se define como

```
O(f(n))= {g: INTEGER -> REAL+  tales que
          existen las constantes  k  y  N0 tales que
          para todo  N > N0,  g(N) <= k*f(N) }
```

en palabras, O(f(n)) esta formado por aquellas funciones g(n) que crecen a un ritmo menor o igual que el de f(n).

De las funciones "g" que forman este conjunto O(f(n)) se dice que **"están dominadas asintóticamente"** por "f", en el sentido de que para N suficientemente grande, y salvo una constante multiplicativa "k", f(n) es una cota superior de g(n).

3.1. Órdenes de Complejidad

Se dice que O(f(n)) define un **"orden de complejidad"**. Escogeremos como representante de este orden a la función f(n) más sencilla del mismo. Así tendremos

O(1)	orden constante
O(log n)	orden logarítmico
O(n)	orden lineal
O(n log n)	
$O(n^2)$	orden cuadrático
$O(n^a)$	orden polinomial (a > 2)
$O(a^n)$	orden exponencial (a > 2)
O(n!)	orden factorial

Es más, se puede identificar una jerarquía de órdenes de complejidad que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo A se puede demostrar de un cierto orden O_1, es cierto que tambien pertenece a todos los órdenes superiores (la relación de orden çota superior de' es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo cubra.

3.1.1. Impacto Práctico

Para captar la importancia relativa de los órdenes de complejidad conviene echar algunas cuentas.

Sea un problema que sabemos resolver con algoritmos de diferentes complejidades. Para compararlos entre si, supongamos que todos ellos requieren 1 hora de ordenador para resolver un problema de tamaño N=100.

¿Qué ocurre si disponemos del doble de tiempo? Notese que esto es lo mismo que disponer del mismo tiempo en un odenador el doble de potente, y que el ritmo actual de progreso del hardware es exactamente ese:

"duplicación anual del número de instrucciones por segundo".

¿Qué ocurre si queremos resolver un problema de tamaño 2n?

O(f(n))	N=100	t=2h	N=200
log n	1 h	10000	1.15 h
n	1 h	200	2 h
n log n	1 h	199	2.30 h
n^2	1 h	141	4 h
n^3	1 h	126	8 h
2^n	1 h	101	10^{30} h

Los algoritmos de complejidad O(n) y O(n log n) son los que muestran un comportamiento más "natural": prácticamente a doble de tiempo, doble de datos procesables.

Los algoritmos de complejidad logarítmica son un descubrimiento fenomenal, pues en el doble de tiempo permiten atacar problemas notablemente mayores, y para resolver un problema el doble de grande sólo hace falta un poco más de tiempo (ni mucho menos el doble).

Los algoritmos de tipo polinómico no son una maravilla, y se enfrentan con dificultad a problemas de tamaño creciente. La práctica viene a decirnos que son el límite de lo "tratable".

Sobre la tratabilidad de los algoritmos de complejidad polinómica habria mucho que hablar, y a veces semejante calificativo es puro eufemismo. Mientras complejidades del orden $O(n^2)$ y $O(n^3)$ suelen ser efectivamente abordables, prácticamente nadie acepta algoritmos de orden $O(n^{100})$, por muy polinómicos que sean. La frontera es imprecisa.

Cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas ridiculamente pequeños.

A la vista de lo anterior se comprende que los programadores busquen algoritmos de complejidad lineal. Es un golpe de suerte encontrar algo de complejidad logarítmica. Si se encuentran soluciones polinomiales, se puede vivir con ellas; pero ante soluciones de complejidad exponencial, más vale seguir buscando.

No obstante lo anterior ...

- ... si un programa se va a ejecutar muy pocas veces, los costes de codificación y depuración son los que más importan, relegando la complejidad a un papel secundario.
- ... si a un programa se le prevé larga vida, hay que pensar que le tocará mantenerlo a otra persona y, por tanto, conviene tener en cuenta su legibilidad, incluso a costa de la complejidad de los algoritmos empleados.
- ... si podemos garantizar que un programa sólo va a trabajar sobre datos pequeños (valores bajos de N), el orden de complejidad del algoritmo que usemos suele ser irrelevante, pudiendo llegar a ser incluso contraproducente.

  Por ejemplo, si disponemos de dos algoritmos para el mismo problema, con tiempos de ejecución respectivos:

algoritmo	tiempo	complejidad
f	100 n	O(n)
g	n^2	$O(n^2)$

asintóticamente, "f" es mejor algoritmo que "g"; pero esto es cierto a partir de N > 100. Si nuestro problema no va a tratar jamás problemas de tamaño mayor que 100, es mejor solución usar el algoritmo "g".

El ejemplo anterior muestra que las constantes que aparecen en las fórmulas para T(n), y que desaparecen al calcular las funciones de complejidad, pueden ser decisivas desde el punto de vista de ingeniería. Pueden darse incluso ejemplos más dramaticos:

algoritmo	tiempo	complejidad
f	n	O(n)
g	100 n	O(n)

aún siendo dos algoritmos con idéntico comportamiento asintótico, es obvio que el algoritmo "f" es siempre 100 veces más rápido que el "g" y candidato primero a ser utilizado.

- ... usualmente un programa de baja complejidad en cuanto a tiempo de ejecución, suele conllevar un alto consumo de memoria; y viceversa. A veces hay que sopesar ambos factores, quedándonos en algún punto de compromiso.
- ... en problemas de cálculo numérico hay que tener en cuenta más factores que su complejidad pura y dura, o incluso que su tiempo de ejecución: queda por considerar la precisión del cálculo, el máximo error introducido en cálculos intermedios, la estabilidad del algoritmo, etc. etc.

3.2. Propiedades de los Conjuntos O(f)

No entraremos en muchas profundidades, ni en demostraciones, que se pueden hallar en los libros especializados. No obstante, algo hay que saber de cómo se trabaja con los conjuntos O() para poder evaluar los algoritmos con los que nos encontremos.

Para simplificar la notación, usaremos O(f) para decir O(f(n))

Las primeras reglas sólo expresan matemáticamente el concepto de jerarquía de órdenes de complejidad:

A. La relación de orden definida por

f < g <=> f(n) IN O(g)
es reflexiva: f(n) IN O(f)
y transitiva: f(n) IN O(g) y g(n) IN O(h) => f(n) IN O(h)

B. f IN O(g) y g IN O(f) <=> O(f) = O(g)

Las siguientes propiedades se pueden utilizar como reglas para el cálculo de órdenes de complejidad. Toda la maquinaria matemática para el cálculo de límites se puede aplicar directamente:

```
C.  Lim_(n->inf) f(n)/g(n) = 0  =>  f IN O(g)
                                =>  g NOT_IN O(f)
                                =>  O(f) es subconjunto de O(g)
D.  Lim_(n->inf) f(n)/g(n) = k  =>  f IN O(g)
                                =>  g IN O(f)
                                =>  O(f) = O(g)
E.  Lim_(n->inf) f(n)/g(n)= INF =>  f NOT_IN O(g)
                                =>  g IN O(f)
                                =>  O(f) es superconjunto de O(g)
```

Las que siguen son reglas habituales en el cálculo de límites:

```
F.  Si  f, g IN O(h)  =>  f+g IN O(h)
G.  Sea k una constante, f(n) IN O(g)  =>  k*f(n) IN O(g)
H.  Si  f IN O(h1)  y  g IN O(h2)  =>  f+g IN O(h1+h2)
I.  Si  f IN O(h1)  y  g IN O(h2)  =>  f*g IN O(h1*h2)
J.  Sean los reales  0 < a < b  =>  O(n^a) es subconjunto de O(n^b)
K.  Sea P(n) un polinomio de grado k  =>  P(n) IN O(n^k)
L.  Sean los reales a, b > 1  =>  O(log_a) = O(log_b)
```

La regla [L] nos permite olvidar la base en la que se calculan los logaritmos en expresiones de complejidad.

La combinación de las reglas [K, G] es probablemente la más usada, permitiendo de un plumazo olvidar todos los componentes de un polinomio, menos su grado.

Por último, la regla [H] es la basica para analizar el concepto de secuencia en un programa: la composición secuencial de dos trozos de programa es de orden de complejidad el de la suma de sus partes.

4. Reglas Prácticas

Aunque no existe una receta que siempre funcione para calcular la complejidad de un algoritmo, si es posible tratar sistematicamente una gran cantidad de ellos, basandonos en que suelen estar bien estructurados y siguen pautas uniformes.

Loa algoritmos bien estructurados combinan las sentencias de alguna de las formas siguientes

1. sentencias sencillas
2. secuencia (;)
3. decisión (if)
4. bucles
5. llamadas a procedimientos

4.0. Sentencias sencillas

Nos referimos a las sentencias de asignación, entrada/salida, etc. siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño este relacionado con el tamaño N del problema. La inmensa mayoría de las sentencias de un algoritmo requieren un tiempo constante de ejecución, siendo su complejidad O(1).

4.1. Secuencia (;)

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales, aplicándose las operaciones arriba expuestas.

4.2. Decisión (if)

La condición suele ser de O(1), complejidad a sumar con la peor posible, bien en la rama THEN, o bien en la rama ELSE. En decisiones multiples (ELSE IF, SWITCH CASE), se tomara la peor de las ramas.

4.3. Bucles

En los bucles con contador explícito, podemos distinguir dos casos, que el tamaño N forme parte de los límites o que no. Si el bucle se realiza un número fijo de veces, independiente de N, entonces la repetición sólo introduce una constante multiplicativa que puede absorberse.

```
Ej.- for (int i= 0; i < K; i++) { algo_de_O(1) }      =>  K*O(1) = O(1)
```

Si el tamaño N aparece como límite de iteraciones ...

```
Ej.- for (int i= 0; i < N; i++) { algo_de_O(1) }  =>  N * O(1) = O(n)

Ej.- for (int i= 0; i < N; i++) {
       for (int j= 0; j < N; j++) {
         algo_de_O(1)
       }
     }
```

tendremos N * N * O(1) = $O(n^2)$

```
Ej.- for (int i= 0; i < N; i++) {
       for (int j= 0; j < i; j++) {
         algo_de_O(1)
       }
     }
```

el bucle exterior se realiza N veces, mientras que el interior se realiza `1, 2, 3, ... N` veces respectivamente. En total,

```
       1 + 2 + 3 + ... + N = N*(1+N)/2  -> O(n²)
```

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
Ej.- c= 1;
     while (c < N) {
       algo_de_O(1)
       c= 2*c;
     }
```

El valor incial de "c" es 1, siendo "2^k" al cabo de "k" iteraciones. El número de iteraciones es tal que

$2^k >= N => k= eis (\log_2 (N))$ [el entero inmediato superior]

y, por tanto, la complejidad del bucle es O(log n).

```
Ej.- c= N;
```

```
while (c > 1) {
  algo_de_O(1)
  c= c / 2;
}
```

Un razonamiento análogo nos lleva a $\log_2(N)$ iteraciones y, por tanto, a un orden O(log n) de complejidad.

```
Ej.- for (int i= 0; i < N; i++) {
       c= i;
       while (c > 0) {
         algo_de_O(1)
         c= c/2;
       }
     }
```

tenemos un bucle interno de orden O(log n) que se ejecuta N veces, luego el conjunto es de orden O(n log n)

4.4. Llamadas a procedimientos

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí. El coste de llamar no es sino una constante que podemos obviar inmediatamente dentro de nuestros análisis asintóticos.

El cálculo de la complejidad asociada a un procedimiento puede complicarse notáblemente si se trata de procedimientos recursivos. Es fácil que tengamos que aplicar técnicas propias de la matemática discreta, tema que queda fuera de los límites de esta nota técnica.

4.5. Ejemplo: evaluación de un polinomio

Vamos a aplicar lo explicado hasta ahora a un problema de fácil especificación: diseñar un programa para evaluar un polinomio P(x) de grado N;

```
class Polinomio {
  private double[] coeficientes;

  Polinomio (double[] coeficientes) {
    this.coeficientes= new double[coeficientes.length];
    System.arraycopy(coeficientes, 0, this.coeficientes, 0,
                     coeficientes.length);
  }

  double evalua_1 (double x) {
    double resultado= 0.0;
    for (int termino= 0; termino < coeficientes.length; termino++) {
      double xn= 1.0;
      for (int j= 0; j < termino; j++)
        xn*= x;                // x elevado a n
      resultado+= coeficientes[termino] * xn;
    }
    return resultado;
  }
}
```

Como medida del tamaño tomaremos para N el grado del polinomio, que es el número de coeficientes en C. Así pues, el bucle más exterior (1) se ejecuta N veces. El bucle interior (2) se ejecuta, respectivamente

```
    1 + 2 + 3 + ... + N veces  =  N*(1+N)/2  =>  O(n²)
```

Intuitivamente, sin embargo, este problema debería ser menos complejo, pues repugna al sentido común que sea de una complejidad tan elevada. Se puede ser más inteligente a la hora de evaluar la potencia x^n:

```
double evalua_2 (double x) {
  double resultado= 0.0;
  for (int termino= 0; termino < coeficientes.length; termino++) {
    resultado+= coeficientes[termino] * potencia(x, termino);
  }
  return resultado;
}

private double potencia (double x, int n) {
  if (n == 0)
    return 1.0;
  // si es potencia impar ...
  if (n%2 == 1)
    return x * potencia(x, n-1);
  // si es potencia par ...
  double t= potencia(x, n/2);
  return t*t;
}
```

El análisis de la función Potencia es delicado, pues si el exponente es par, el problema tiene una evolución logarítmica; mientras que si es impar, su evolución es lineal. No obstante, como si "j" es impar entonces "j-1" es par, el caso peor es que en la mitad de los casos tengamos "j" impar y en la otra mitad sea par. El caso mejor, por contra, es que siempre sea "j" par.

Un ejemplo de caso peor seria x^{31}, que implica la siguiente serie para j:

```
31  30  15  14  7  6  3  2  1
```

cuyo número de terminos podemos acotar superiormente por

```
  2 * eis (log₂(j)),
```

donde eis(r) es el entero inmediatamente superior (este cálculo responde al razonamiento de que en el caso mejor visitaremos $eis(log_2(j))$ valores pares de "j"; y en el caso peor podemos encontrarnos con otros tantos números impares entremezclados).

Por tanto, la complejidad de Potencia es de orden O(log n).

Insertada la función Potencia en la función EvaluaPolinomio, la complejidad compuesta es del orden O(n log n), al multiplicarse por N un subalgoritmo de O(log n).

Así y todo, esto sigue resultando estravagante y excesivamente costoso. En efecto, basta reconsiderar el algoritmo almacenando las potencias de "X" ya calculadas para mejorarlo sensiblemente:

```
double evalua_3 (double x) {
  double xn= 1.0;
```

```
    double resultado= coeficientes[0];
    for (int termino= 1; termino < coeficientes.length; termino++) {
      xn*= x;
      resultado+= coeficientes[termino] * xn;
    }
    return resultado;
  }
```

que queda en un algoritmo de O(n).

Habiendo N coeficientes C distintos, es imposible encontrar ningun algoritmo de un orden inferior de complejidad.

En cambio, si es posible encontrar otros algoritmos de idéntica complejidad:

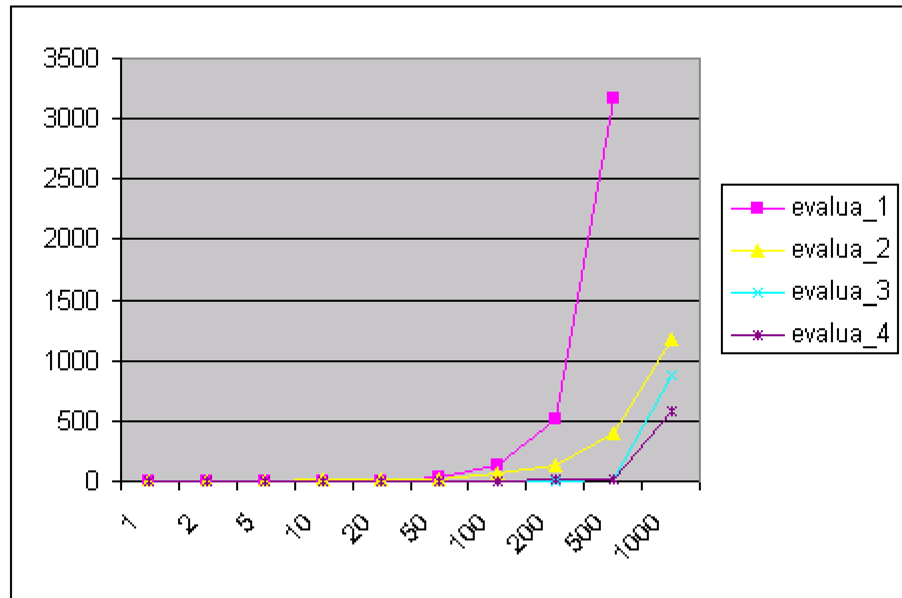
```
  double evalua_4 (double x) {
    double resultado= 0.0;
    for (int termino= coeficientes.length-1; termino >= 0; termino--) {
      resultado= resultado * x +
        coeficientes[termino];
    }
    return resultado;
  }
```

No obstante ser ambos algoritmos de idéntico orden de complejidad, cabe resaltar que sus tiempos de ejecución serán notablemente distintos. En efecto, mientras el último algoritmo ejecuta N multiplicaciones y N sumas, el penúltimo requiere 2N multiplicaciones y N sumas. Si, como es frecuente, el tiempo de ejecución es notablemente superior para realizar una multiplicación, cabe razonar que el último algoritmo ejecutará en la mitad de tiempo que el anterior.

4.5.1. Medidas de laboratorio

La siguiente tabla muestra algunas medidas de la eficacia de nuestros algoritmos sobre una implementación en Java:

grado	evalua_1	evalua_2	evalua_3	evalua_4
1	0	10	0	0
2	10	0	0	0
5	0	0	0	0
10	0	10	0	0
20	0	10	0	0
50	40	20	0	10
100	130	60	0	0
200	521	140	0	10
500	3175	400	10	10
1000	63632	1171	872	580



5. Problemas P, NP y NP-completos

Hasta aquí hemos venido hablando de algoritmos. Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables. Se suele decir que el orden de complejidad de un problema es el del mejor algoritmo que se conozca para resolverlo. Así se clasifican los problemas, y los estudios sobre algoritmos se aplican a la realidad.

Estos estudios han llevado a la constatación de que existen problemas muy difíciles, problemas que desafian la utilización de los ordenadores para resolverlos. En lo que sigue esbozaremos las clases de problemas que hoy por hoy se escapan a un tratamiento informático.

Clase P.-

Los algoritmos de complejidad polinómica se dice que son tratables en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P. Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables. Seria muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

Clase NP.-

Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico. Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

Clase NP-completos.-

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente podemos decir que algunos problemas se hayan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución

sería fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al Nobel reservado para el que descubra semejante solución ... ¡y se duda seriamente de que alguien lo consiga!
Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

6. Conclusiones

Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera. Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. se suele trabajar con un cálculo asintótico que indica como se comporta el algoritmo para datos muy grandes y salvo algun coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

7. Bibliografia

Es difícil encontrar libros que traten este tema a un nivel introductorio sin caer en amplios desarrollos matemáticos, aunque tambien es cierto que casi todos los libros que se precien dedican alguna breve sección al tema. Probablemente uno de los libros que sólo dedican un capítulo; pero es extremadamente claro es
L. Goldschlager and A. Lister.
Computer Science, A Modern Introduction
Series in Computer Science. Prentice-Hall Intl., London (UK), 1982.
Siempre hay algún clásico con una presentación excelente, pero entrando en mayores honduras matemáticas como
A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
Data Structures and Algorithms
Addison-Wesley, Massachusetts, 1983.
Recientemente ha aparecido un libro en castellano con una presentación muy buena, si bien esta escrito en plan matemático y, por tanto, repleto de demostraciones (un poco duro)
Carmen Torres
Diseño y Análisis de Algoritmos
Paraninfo, 1992

Para un estudio serio hay que irse a libros de matemática discreta, lo que es toda una asignatura en sí misma; pero se pueden recomendar un par de libros modernos, prácticos y especialmente claros:

R. Skvarcius and W. B. Robinson.
Discrete Mathematics with Computer Science Applications
Benjamin/Cummings, Menlo Park, California, 1986.

R. L. Graham, D. E. Knuth, and O. Patashnik.
Concrete Mathematics
Addison-Wesley, 1990.

Para saber más de problemas NP y NP-completos, hay que acudir a la "biblia", que en este tema se denomina

M. R. Garey and D. S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness
Freeman, 1979.

Sólo a efectos documentales, permítasenos citar al inventor de las nociones de complejidad y de la notación O():

P. Bachmann
Analytische Zahlen Theorie
1894

Unidad 5 Estructuras no lineales estáticas y dinámicas.

Estructuras dinámicas de datos

Indice

[1. Punteros](#)
[2. Estructuras dinámicas](#)
[3. Listas](#)

1. Punteros

Hemos visto ya cómo las variables son las células de memoria a las que podemos tener acceso por un identificador. Pero estas variables se guardan en lugares concretos de la memoria de la computadora. Para nuestros programas, la memoria de la computadora es solamente una sucesión de las células de 1 octeto (la talla mínima para un dato), cada una con una dirección única.}
La memoria de computadora puede ser comparada con una calle en una ciudad. En una calle todas las casas se numeran consecutivamente con un identificador único tal que si hablamos del número 27 de la calle Córdova, podremos encontrar el lugar sin pérdida, puesto que debe haber solamente una casa con ese número y, además, nosotros sabemos que la casa estará entre las casas 26 y 28.
Una declaración de puntero consiste en un tipo base, un * y el nombre de la variable.
La forma general de declaración de una variable puntero es:
Tipo *nomb_var;
Donde:
Tipo: cualquier tipo valido ,ya sea primitivo o definido por el usuario
nomb_var: es el nombre de la variable de tipo apuntador.
Los operadores de punteros
Existen dos operadores especiales de punteros: & y *.
El & devuelve la dirección de memoria de su operando. Por ejemplo:
m=&cuenta;
pone en m la dirección de memoria de la variable cuenta. Esta dirección es la posición interna de la variable en la computadora. La dirección no tiene nada que ver con el valor de cuenta. Se puede pensar en el operador & como devolviendo "la dirección de".
El segundo operador de punteros, *, es el complemento de &. Devuelve el valor de la variable localizada en la dirección que sigue. Por ejemplo, si m contiene la dirección de memoria de la variable cuenta, entonces:
q=*m;
pone el valor de cuenta en q. Se puede pensar en * como "en la dirección".
El siguiente programa ilustra un ejemplo:

```
#include <stdio.h>
main()
{int cuenta, q;
int *m;
cuenta=100;
```

m=&cuenta; //m recibe la dirección de cuenta
q=*m; //a q se le asigna el valor de cuenta
indirectamente a través de m
print("%d,q") //imprime 100
}

Punteros estáticos
Definamos un puntero a un entero y una variable entera como sigue:
Int *p1;
Int valor1;
Con estas definiciones es posible hacer las siguientes asignaciones estáticas:
p1= *valor1;
*p1=25;

El apuntador p1 se define como un apuntador a un entero. La variable valor2 se define como una variable entera. La primera asignación hace que p1 apunte a la variable valor1, la segunda asignación almacena en memoria el valor 25 en donde p1 está apuntando.
Se dice que este tipo de inicialización es de tipo estática porque la asignación de la memoria que se utiliza para almacenar es fija. Una vez definida la variable, el compilador establece suficiente memoria para almacenar un valor de un tipo dado. Esta memoria permanece reservada para esta variable y no es posible usarla para nada más hasta que se termine la función.

Punteros Dinámicos
La segunda forma para inicializar un puntero es por medio de la asignación dinámica de memoria. Por asignación dinámica de la memoria se entiende que se reserva memoria cuando se necesite para almacenar un valor de un tipo dado. Después, una vez que no se necesite el valor, es posible liberar la memoria y hacerla disponible para otro uso por el sistema .
De nuevo definamos a p1 como un valor entero como sigue:
Int *p1;
Ahora es posible inicializar a p1 en forma dinámica para apuntar a un valor de la siguiente manera:
p1=new int;
*p1=25;
Esta vez no es necesario inicializar primero p1 a la dirección de un a variable estática. En cambio el operador new crea suficiente memoria para contener un valor entero apuntado por p1. Después se almacena en esta área de memoria el valor 25. Una vez que se asigna dinámicamente la memoria como ésta, es posible liberar la misma área de memoria usando el operador delete, como sigue:
Delete p1;
Esta operación liberará la memoria apuntada por p1.
Es importante señalar que el operador delete no elimina el apuntador, simplemente libera el área de memoria al cual se dirige el puntero. Por tanto luego que se ejecute el enunciado anterior, p1 todavía existe como un puntero que no apunta a nada, pero que es posible inicializarlo de nuevo para apuntar a otro entero utilizando el operador new.

2. Estructuras dinámicas

Las estructuras dinámicas de datos son estructuras que cuya dimensión puede crecer o disminuir durante la ejecución del programa. Una estructura dinámica de datos es una colección de elementos llamados nodos. Al contrario que un array, que contiene espacio para almacenar un número fijo de elementos, una estructura dinámica de datos se amplía y contrae durante la ejecución del programa.

Las estructuras dinámicas de datos se pueden dividir en dos grandes grupos:
Lineales: listas enlazadas, pilas, colas
No lineales: árboles , grafos
Las estructuras dinámicas de datos son de gran utilidad para almacenar datos del mundo real, que están cambiando constantemente. Por ejemplo si tenemos almacenados en un array los datos de los alumnos de un curso, los cuales estan ordenados de acuerdo al promedio, para insertar un nuevo alumno seria necesario correr cada elemento un espacio: Si en su lugar se utilizara una estructura dinámica de datos, los nuevos datos del alumno se pueden insertar fácilmente.

3. Listas

Listas Enlazadas
Una lista enlazada es un conjunto de elementos llamados nodos en los que cada uno de ellos contiene un dato y también la dirección del siguiente nodo.
El primer elemento de la lista es la cabecera, que sólo contiene un puntero que señala el primer elemento de la lista.
El último nodo de la lista apunta a NULL (nulo) porque no hay más nodos en la lista. Se usará el término NULL para designar el final de la lista.

La lista enlazada se muestra en la siguiente figura:

Cabecera
Las operaciones que normalmente se ejecutan con listas incluyen:

1. Recuperar información de un nodo específico.
2. Encontrar el nodo que contiene una información específica.
3. Insertar un nuevo nodo en un lugar específico.
4. Insertar un nuevo nodo en relación a una información particular.
5. Borrar un nodo existente.

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
class nodo
{int num;
nodo *sgte;
public:
nodo(){sgte=NULL;}
void apns(nodo *n);
```

```
nodo *rpns();
void ingresar();
int mostrar();
};
void nodo::apns(nodo *n)
{sgte=n;}
nodo *nodo::rpns()
{return sgte;}
void nodo::ingresar()
{cin>>num;}
int nodo::mostrar()
{return num;}
////////////////////////////////////////
class lista
{nodo *cab;
public:
lista(){cab=NULL;}
void insertar_i();
void insertar_f();
void eliminar_nodo();
void mostrar_lista();
void eliminar_i();
void eliminar_f();
void eliminar_lista();
};
void lista::insertar_f()
{nodo *a;
a=new nodo;
a->ingresar();
if(cab==NULL)
{cab=a;}
else
{nodo *temp,*temp1,*temp2;
temp=cab;
while(temp2!=NULL)
{temp1=temp->rpns();
temp2=temp1->rpns();}
temp->rpns();
temp->apns(a);
}
}
void lista::insertar_i()
{nodo *a;
a=new nodo;
a->ingresar();
if(cab==NULL)
cab=a;
else
{nodo *temp;
```

```
temp=cab;
cab=a;
cab->apns(temp);
}
}
void lista::mostrar_lista()
{nodo *t;
if (cab==NULL)
cout<<"La lista esta vacia";
else
{t=cab;
while(t!=NULL)
{//t=t->rpns();
cout<<t->mostrar();
t=t->rpns();
}
}
}
void lista::eliminar_nodo()
{nodo *temp,*temp1,*temp2;
temp1=cab;
int numero;
cout<<"ingrese numero a borrar";
cin>>numero;
if(temp1->rpns()==NULL)
cout<<"no hay elementos";
else
{do
{temp=temp1;
temp1=temp1->rpns();
if(temp1->mostrar()==numero)
{temp2=temp1->rpns();
temp->apns(temp2);
delete temp1;
}
}while(temp!=NULL); /////
}}
void lista::eliminar_i()
{if(cab==NULL)
cout<<"\nno existen nodos";
else
{nodo *temp;
temp=cab;
cab=cab->rpns();
delete temp;
}
}
void lista::eliminar_f()
{if(cab==NULL)
```

```
cout<<"\nno existen nodos";
else
{nodo *ptr1, *ptr2;
ptr2=cab;
ptr1=NULL;
while(ptr2->rpns()!=NULL)
{ptr1=ptr2;
ptr2=ptr2->rpns();
}
if(ptr1==NULL)
{delete cab;
cab=NULL;
}
else
{delete ptr2;
ptr1->apns(NULL);
}
}
}}
void lista::eliminar_lista()
{nodo *temp;
while(cab!=NULL)
{temp=cab;
cab=cab->rpns();
delete temp;
}
}
void main()
{int op;
lista b;
clrscr();
for(;;)
{
cout<<"\n\n<1> insertar al inicio";
cout<<"\n<2> insertar al final";
cout<<"\n<3> eliminar nodo";
cout<<"\n<4> mostrar lista";
cout<<"\n<5> eliminar inicio";
cout<<"\n<6> eliminar final";
cout<<"\n<7> eliminar lista";
cout<<"\n<8> salir\n";
op=getch();
switch(op)
{case '1': b.insertar_i();break;
case '2': b.insertar_f();break;
case '3': b.eliminar_nodo(); break;
case '4': b.mostrar_lista();break;
case '5': b.eliminar_i();break;
case '7': b.eliminar_lista();break;
```

```
case '8': exit(0);
}
}
}
```

Listas Doblemente Enlazadas

Hasta ahora se han manejado listas que se recorren en un asola dirección. En algunas aplicaciones es práctico o hasta indispensable poder recorrer una lista en ambas direcciones. Para estos casos se tienen las lista doblemente enlazadas. Esta propiedad implica que cada nodo debe tener dos apuntadores, uno al nodo predecesor y otro al nodo sucesor.

Cabecera

Listas Circulares

Una lista circular es una lista en la cual el último nodo es enlazado al primer elemento de la lista. La ventaja de este tipo de estructura es que siempre se puede llegar a cualquier nodo siguiendo los enlaces. La desventaja es que si no se tiene cuidado una búsqueda puede resultar en un bucle infinito. Esto se puede evitar al determinar a un nodo como nodo-cabeza o nodo inicial.

Pilas

Una pila es un tipo especial de lista lineal en la cual un elemento sólo puede ser añadido o eliminado por un extremo llamado cima. Esto significa que los elementos se sacan de la pila en orden inverso al que se pusieron en ella.

Las dos operaciones básicas asociadas a las pilas son:

-Poner: es añadir un elemento a la pila.

-Sacar: es extraer un elemento de la pila.

La pila es una estructura con numerosas analogías en la vida real: una pila de platos, una pila de monedas, una pila de bandejas, etc.
La representación consiste en un vector con espacio para un máximo de elementos y un contador que indica el número de elementos válidos almacenados en el vector.

Colas

Una cola es una lista en las que las supresiones se realizan solamente solamente al principio de la lista y las inserciones al final de la misma. Al igual que en el caso de las pilas, hay que prever un vector para almacenar el máximo número de elementos que puedan presentarse en el programa. A diferencia de las pilas, no basta con añadir un simple contador, tal que indique el número de elementos válidos; sino hay que prever dos índices que indiquen la posición del comienzo y del final de la cola. Si la cola no está vacía, en CABEZA está el primer elemento, y si la cola no está llena, en FIN es el lugar donde se copia el siguiente elemento que se incorpora a la misma.
Las colas se usan para almacenar datos que necesitan ser procesados según el orden de llegada. En la vida real se tienen ejemplos numerosos de colas: la cola de un cine, la cola de un banco, etc; en todas ellas el primer elemento que llega es el primero que sale.

Unidad 7. Estructuras dinámicas de datos

Los tipos de datos simples pueden ser organizados en diferentes estructuras de datos: estáticas y dinámicas.

7.1. Tipos de estructuras dinámicas.

Son estructuras que crecen a medida que se ejecuta un programa. Es una colección de elementos (nodos) que son normalmente registros. Se pueden dividir en dos grandes grupos:
Lineales y no lineales.

7.1.1. Estructuras lineales.

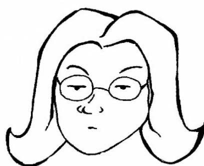
Son pilas, colas y listas enlazadas.

7.1.2. Estructuras no lineales.

Árboles y Grafos.

arriba

7.2. Listas enlazadas.



Es un conjunto de elementos en los que cada elemento contiene la posición (dirección) del siguiente elemento de la lista. Cada elemento debe tener al menos dos campos. Un campo que tiene el valor del elemento y un campo de enlace (link) que contiene la posición del siguiente elemento.

7.2.1. Creación.

Los valores se almacenan en un nodo, cada nodo contiene al menos un campo dato o valor y un enlace (indicador o puntero) con el siguiente campo. El campo enlace apunta al siguiente nodo de la lista. El último nodo de la lista suele terminarse con un nulo para indicar el fin de la lista. No es necesario que los elementos de la lista sean almacenados en posiciones físicas adyacentes, ya que el puntero indica donde se encuentra el siguiente elemento.

7.2.2. Recorrido y búsqueda.

Las operaciones que normalmente se ejecutan son:

1. Recuperar información de un nodo específico
2. Encontrar el nodo que contiene la información específica.
3. Borrar un nodo existente que contiene información específica.

7.2.3. Inserción.

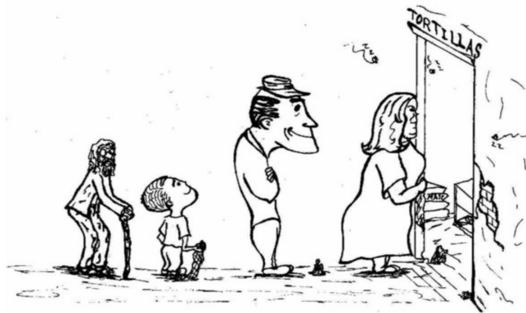
1. Insertar un nuevo nodo en un lugar específico.
2. Insertar un nuevo nodo en relación a una información particular.

7.2.4. Listas doblemente enlazadas.

Pueden recorrerse en ambas direcciones. En estas listas cada nodo consta de un campo de información de datos y dos campos de enlace al nodo anterior y al nodo posterior. Ocupa más espacio en memoria.

arriba

7.3. Colas.



Las colas se utilizan para almacenar datos que necesitan ser procesados según el orden de llegada. Es una estructura lineal de datos. La eliminación se realiza al principio de la lista (Frente/Front). El primer elemento que entró es el primero que saldrá (FIFO, First In First Out).

7.3.1. Creación.

Se pueden representar por listas enlazadas o por arreglos. Se necesitan dos punteros Frente y Final, y la lista o arreglo de n elementos. También puede crearse una lista enlazada circular que sólo necesitará un puntero.

7.3.2. Inserción.

La inserción de un elemento se realiza en el último extremo (Final/Rear).

arriba

7.4. Pilas.

Una Pila (Stack) es una lista lineal en la que la inserción y borrado de elementos se realiza sólo por un extremo denominado Cima o Top (Top)

7.4.1. Creación.

Se necesita un puntero que apunte a la Cima o Top. Para crearse, es posible utilizar un arreglo unidimensional de tipo FIFO.

7.4.2. Inserción.

Dado que las operaciones de insertar y eliminar se realizan por un solo extremo, los elementos solo pueden eliminarse en orden inverso al que se insertan en la pila. El último elemento que se pone en la pila es el primero que se puede sacar. Estas estructuras se conocen con el nombre de LIFO (Last In First Out). Las operaciones más usuales son Push (insertar un elemento en la pila) y Pop (eliminar un elemento).

arriba

7.5. Árboles.

Se usan para representar datos con una relación jerárquica entre sus elementos (Árboles Genealógicos, Tablas, etc.). Existen 2 tipos de árboles: Binario, Binario completo y No binario . Es una estructura recursiva que contiene un nodo llamado Padre o Raíz y subnodos llamados Hijos u Hojas.

7.5.1. Representación de árboles.

- Raíz.- Todos los árboles que no estén vacíos tienen un nodo llamado Raíz. Todos los demás elementos o nodos se derivan o descienden de él. Este nodo no tiene Padre, es decir, no es hijo de ningún elemento.
- Nodo.- Son los vértices o elementos del árbol.
- Nodo Terminal u Hoja (Leaf Node).- Es aquel nodo que no contiene ningún subárbol, o sea que no tiene hijos.
- Hijo.- A cada nodo que no es Hoja se le asocia uno o vario subárboles llamados descendiente o Hijos. De igual forma, cada nodo tiene asociado un antecesor o ascendiente llamado Padre.

  Los nodos de un mismo Padre se llaman Hermanos.

7.5.2. Recorrido de árboles binarios.

- Prefijo. Se compone de tres pasos: Raíz, Hijo Izquierdo, Hijo Derecho.
- Infijo: Se compone de tres pasos: Hijo Izquierdo, Raíz, Hijo Derecho.
- Postfijo: Se compone de tres pasos: Hijo Izquierdo, Hijo Derecho, Raíz.

7.5.3. Búsqueda e inserción.

El árbol de búsqueda se construirá teniendo en cuenta las siguientes premisas:

1. El primer elemento se utiliza para crear el nodo Raíz.
2. Los valores del árbol deben ser tales que pueda existir un orden (entero, real, lógico o carácter e incluso definido por el usuario).
3. En cualquier nodo todos los valores del subárbol izquierdo del nodo son menor o igual al valor del nodo. De modo similar todos los valores del subárbol derecho deben ser mayores que los valores del nodo.

arriba

7.6. Grafos.

Es una estructura de datos no lineal. Los árboles representan estructuras jerárquicas con limitaciones. Si se eliminan las restricciones de que cada nodo puede apuntar a otros nodos y que cada nodo puede estar apuntado por otro nodo nos encontraremos con un grafo.
Es un conjunto de puntos y líneas cada uno de los cuales une a un punto con otro.

7.6.1. Representación de grafos en la computadora.

Existen dos técnicas para representar un grafo:

1. Matriz de Adyacencia.- Es una arreglo de dos dimensiones que representan las conexiones entre pares de verticales.
2. Lista de Adyacencia.- Se utiliza cuando un grafo tiene muchos vértices y pocas aristas. Esta representación se lleva a cabo con una lista enlazada por cada vértice del grafo que contenga vértices adyacentes desde él.

5.1 Concepto de árbol.

Concepto y definiciones

En ciencias de la computación, un **árbol** es una estructura de datos ampliamente usada que emula la forma de un árbol (un conjunto de nodos conectados). Un **nodo** es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él. Se dice que un nodo *a* es **padre** de un nodo *b* si existe un enlace desde *a* hasta *b* (en ese caso, también decimos que *b* es hijo de *a*). Sólo puede haber un único nodo sin padres, que llamaremos **raíz**. Un nodo que no tiene hijos se conoce como **hoja**.

Formalmente, podemos definir un árbol de la siguiente forma recursiva:

- Caso base: un árbol con sólo un nodo (es a la vez raíz del árbol y hoja).

- Un nuevo árbol a partir de un nodo n_r y k árboles $A_1, A_2 \ldots A_k$ de raíces $n_1, n_2, \ldots, n_k$ con $N_1, N_2, \ldots, N_k$ elementos cada uno, puede construirse estableciendo una relación padre-hijo entre n_r y cada una de las raíces de los k árboles. El árbol resultante de $N = 1 + N_1 + \ldots + N_k$ nodos tiene como raíz el nodo n_r, los nodos $n_1, n_2, \ldots, n_k$ son los hijos de n_r y el conjunto de nodos hoja está formado por la unión de los k conjuntos hojas iniciales. A cada uno de los árboles A_i se les denota ahora **subárboles** de la raíz.

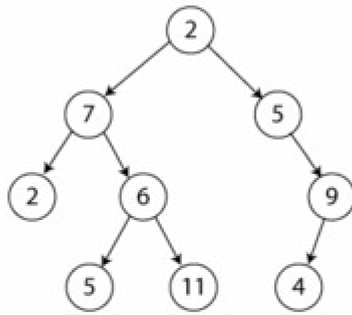
Una sucesión de nodos del árbol, de forma que entre cada dos nodos consecutivos de la sucesión haya una relación de parentesco, decimos que es un **recorrido** árbol. Existen dos recorridos típicos para listar los nodos de un árbol: **primero en profundidad** y **primero en anchura**. En el primer caso, se listan los nodos expandiendo el hijo actual de cada nodo hasta llegar a una hoja, donde se vuelve al nodo anterior probando por el siguiente hijo y así sucesivamente. En el segundo, por su parte, antes de listar los nodos de nivel $n + 1$ (a distancia $n + 1$ aristas de la raíz), se deben haber listado todos los de nivel n. Otros recorridos típicos del árbol son **preorden**, **postorden** e **inorden**:

- El recorrido en **preorden**, también llamado **orden previo** consiste en recorrer en primer lugar la raíz y luego cada uno de los hijos $A_1, A_2 \ldots A_k$ en orden previo.
- El recorrido en **inorden**, también llamado **orden simétrico** (aunque este nombre sólo cobra significado en los árboles binarios) consiste en recorrer en primer lugar A_1, luego la raíz y luego cada uno de los hijos $A_2 \ldots A_k$ en orden simétrico.
- El recorrido en **postorden**, también llamado **orden posterior** consiste en recorrer en primer cada uno de los hijos $A_1, A_2 \ldots A_k$ en orden posterior y por último la raíz.

Finalmente, puede decirse que esta estructura es una representación del concepto de árbol en teoría de grafos. Un árbol es un grafo **conexo** y **acíclico** (ver también teoría de grafos y Glosario en teoría de grafos).

[editar]

Tipos de árboles



Ejemplo de árbol (binario).

- Árboles Binarios
- Árbol de búsqueda binario auto-balanceable
  - Árboles Rojo-Negro
  - Árboles AVL
- Árboles B
- Árboles Multicamino

[editar]

Operaciones de árboles. Representación

Las operaciones comunes en árboles son:

- Enumerar todos los elementos.
- Buscar un elemento.
- Dado un nodo, listar los hijos (si los hay).
- Borrar un elemento.
- Eliminar un subárbol (algunas veces llamada **podar**).
- Añadir un subárbol (algunas veces llamada **injertar**).
- Encontrar la raíz de cualquier nodo.

Por su parte, la representación puede realizarse de diferentes formas. Las más utilizadas son:

- Representar cada nodo como una variable en el heap, con punteros a sus hijos y a su padre.
- Representar el árbol con un array donde cada elemento es un nodo y las relaciones padre-hijo vienen dadas por la posición del nodo en el array.

[editar]

Uso de los árboles

Usos comunes de los árboles son:

- Representación de datos jerárquicos.
- Como ayuda para realizar búsquedas en conjuntos de datos (ver también: algoritmos de búsqueda en Árboles)

5.1.1 Clasificación de árboles.

Árbol binario

De Wikipedia

Saltar a navegación, búsqueda

Para otros usos de este término, véase Árbol binario (desambiguación).

En ciencias de la computación, un **árbol binario.** es una estructura de datos en el cual cada nodo tiene como máximo dos nodos hijos. Típicamente los nodos hijos son llamados **izquierdo** y **derecho**. Usos comunes de los árboles binarios son los árboles binarios de búsqueda y los montículos binarios.

Tabla de contenidos

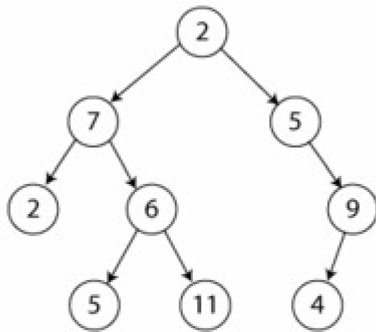
[ocultar]

- 1 Definición de teoría de grafos
- 2 Tipos de árboles binarios
- 3 Implementación en C
- 4 Recorridos sobre árboles binarios
  - 4.1 Recorridos en profundidad
    - 4.1.1 Recorrido en preorden
    - 4.1.2 Recorrido en postorden
    - 4.1.3 Recorrido en inorden
  - 4.2 Recorridos en amplitud (o por niveles)
- 5 Métodos para almacenar árboles binarios

- 6 Véase también

[editar]

Definición de teoría de grafos



Un árbol binario sencillo de tamaño 9 y altura 3, con un nodo raíz cuyo valor es 2

En teoría de grafos, se usa la siguiente definición: «Un árbol binario es un grafo conexo, acíclico y no dirigido tal que el grado de cada vértice no es mayor a 3». De esta forma sólo existe un camino entre un par de nodos.

Un árbol binario con enraizado es como un grafo que tiene uno de sus vértices, llamado *raíz*, de grado no mayor a 2. Con la raíz escogida, cada vértice tendrá un único padre, y nunca más de dos hijos. Si reusamos el requerimiento de la conectividad, permitiendo múltiples componentes conectados en el grafo, llamaremos a esta última estructura un *bosque*.

[editar]

Tipos de árboles binarios

- Un **árbol binario** es un árbol **con raíz** en el que cada nodo tiene como máximo dos hijos.
- Un **árbol binario lleno** es un árbol en el que cada nodo tiene cero o dos hijos.
- Un **árbol binario perfecto** es un árbol binario lleno en el que todas las **hojas** (vértices con cero hijos) están a la misma profundidad (distancia desde la **raíz**, también llamada **altura**)
- A veces un árbol binario perfecto es denominado **árbol binario completo**. Otros definen un árbol **binario completo** como un árbol binario lleno en el que todas las hojas están a profundidad *n* o *n-1*, para alguna *n*.
- Un **árbol casi-completo** es un árbol en el que cada nodo que tiene un hijo derecho también tiene un hijo izquierdo. Tener un hijo izquierdo no requiere que un nodo tenga un hijo derecho. Dicho de otra forma, un **árbol casi completo** es un árbol donde para un hijo derecho, hay siempre un hijo izquierdo, pero para un hijo izquierdo puede no haber un hijo derecho.

[editar]

Implementación en C

Un árbol binario puede declararse de varias maneras. Algunas de ellas son:

Estructura con manejo de memoria dinámica:

```
typedef struct tArbol
{
  int clave;
  struct tArbol *hIzquierdo, *hDerecho;
} tArbol;
```

Estructura con arreglo indexado:

```
typedef struct tArbol
{
  int clave;
  int hIzquierdo, hDerecho;
};
tArbol arbol[NUMERO_DE_NODOS];
```

En el caso de un árbol binario casi-completo (o un árbol completo), puede utilizarse un sencillo arreglo de enteros con tantas posiciones como nodos deba tener el árbol. La información de la ubicación del nodo en el árbol es implícita a cada posición del arreglo. Así, si un nodo está en la posición i, sus hijos se encuentran en las posiciones $2i+1$ y $2i+2$, mientras que su padre (si tiene), se encuentra en la posición truncamiento($(i-1)/2$) (suponiendo que la raíz está en la posición cero). Este método se beneficia de un almacenamiento más compacto y una mejor localidad de referencia, particularmente durante un recorrido en preorden. La estructura para este caso sería por tanto:

```
int arbol[NUMERO_DE_NODOS];
```

[editar]

Recorridos sobre árboles binarios

[editar]

Recorridos en profundidad

[editar]

Recorrido en preorden

En este tipo de recorrido se realiza cierta acción (quizás simplemente imprimir por pantalla el valor de la clave de ese nodo) sobre el nodo actual y posteriormente se trata el subárbol izquierdo y cuando se haya concluido, el subárbol derecho. En el árbol de la figura el recorrido en preorden sería: 2, 7, 2, 6, 5, 11, 5, 9 y 4.

```
void preorden(tArbol *a)
{
```

```
  if (a != NULL) {
    tratar(a);
    preorden(a->hIzquierdo);
    preorden(a->hDerecho);
  }
}
```

[editar]

Recorrido en postorden

En este caso se trata primero el subárbol izquierdo, después el derecho y por último el nodo actual. En el árbol de la figura el recorrido en postorden sería: 2, 5, 11, 6, 7, 4, 9, 5 y 2.

```
void postorden(tArbol *a)
{
  if (a != NULL) {
    postorden(a->hIzquierdo);
    postorden(a->hDerecho);
    tratar(a);
  }
}
```

[editar]

Recorrido en inorden

En este caso se trata primero el subárbol izquierdo, después el nodo actual y por último el subárbol derecho. En un ABB este recorrido daría los valores de clave ordenados de menor a mayor. En el árbol de la figura el recorrido en inorden sería: 2, 7, 5, 6, 11, 2, 5, 4 y 9.

Pseudocódigo:

```
funcion inorden(nodo)
inicio
    si(existe(nodo))
        inicio
            inorden(hijo_izquierdo(nodo));
            visitar(nodo);
            inorden(hijo_derecho(nodo));
        fin;
fin;
```

Implementación en C:

```
void inorden(tArbol *a)
{
  if (a != NULL) {
    inorden(a->hIzquierdo);
    tratar(a);
    inorden(a->hDerecho);
  }
}
```

[editar]

Recorridos en amplitud (o por niveles)

En este caso el recorrido se realiza en orden por los distintos niveles del árbol. Así, se comenzaría tratando el nivel 1, que sólo contiene el nodo raíz, seguidamente el nivel 2, el 3 y así sucesivamente. En el árbol de la figura el recorrido en amplitud sería: 2, 7, 5, 2, 6, 9, 5, 11 y 4.

Al contrario que en los métodos de recorrido en profundidad, el recorrido por niveles no es de naturaleza recursiva. Por ello, se debe utilizar una cola para recordar los subárboles izquierdos y derecho de cada nodo.

Pseudocódigo:

```
    encolar(raiz);
    mientras(cola_no_vacia())
        inicio
            nodo=desencolar();              //Saca un nodo de la cola
            visitar(nodo);                  //Realiza una operación en nodo
            encolar_nodos_hijos(nodo);      //Mete en la cola los hijos del nodo
actual
        fin;
```

Implementación en C:

```
void amplitud(tArbol *a)
{
  tCola cola;
  tArbol *aux;

  if (a != NULL) {
    crearCola(cola);
    encolar(cola, a);
    while (!colavacia(cola)) {
      desencolar(cola, aux);
      visitar(aux);
      if (aux->hIzquierdo != NULL) encolar(cola, aux->hIzquierdo );
      if (aux->hDerecho!= NULL) encolar(cola, aux->hDerecho);
    }
  }
}
```

[editar]

Métodos para almacenar árboles binarios

Los árboles binarios pueden ser construidos desde lenguajes de programación primitivos en muchas formas. En un lenguaje con estructuras y referencias, los árboles binarios son típicamente implementados construyendo con una estructura de tres nodos que contiene algunos datos y referencias a su hijo izquierdo y a su hijo derecho.

Árbol binario de búsqueda auto-balanceable

De Wikipedia

(Redirigido desde Árbol de búsqueda binario auto-balanceable)
Saltar a navegación, búsqueda

En ciencias de la computación, un **árbol binario de búsqueda auto-balanceable** o **equilibrado** es un árbol binario de búsqueda que intenta mantener su *altura*, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente. Esto es importante, ya que muchas operaciones en un árbol de búsqueda binaria tardan un tiempo proporcional a la altura del árbol, y los árboles binarios de búsqueda ordinarios pueden tomar alturas muy grandes en situaciones normales, como cuando las claves son insertadas en orden. Mantener baja la altura se consigue habitualmente realizando transformaciones en el árbol, como la rotación de árboles, en momentos clave.

Tiempos para varias operaciones en términos del número de nodos en el árbol n:

Operación	Tiempo en cota superior asintótica
Búsqueda	O(log n)
Inserción	O(log n)
Eliminación	O(log n)
Iteración en orden	O(n)

Para algunas implementaciones estos tiempos son el peor caso, mientras que para otras están amortizados.

Estructuras de datos populares que implementan este tipo de árbol

- Árbol AVL
- Árbol rojo-negro

Árbol rojo-negro

De Wikipedia

Saltar a navegación, búsqueda

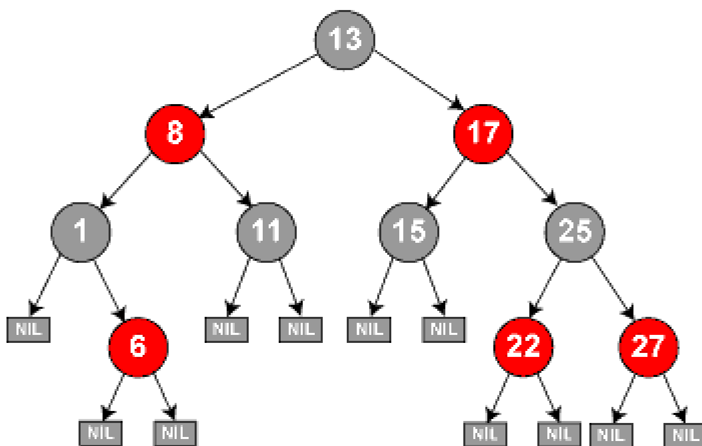
Un **Árbol rojo-negro** es un tipo de Árbol de búsqueda binario auto-balanceable, una estructura de datos usada en ciencias de la computación. Fue inventada en 1972 por Rudolf Bayer quien los llamó "B-Árboles binarios simétricos"

Un Árbol rojo-negro es un árbol binario en el que cada nodo tiene un color como atributo extra, ya sea rojo o negro. El color de los nodos asegura que la trayectoria más larga de la raíz a una hoja no es más larga que el doble del largo de la más corta. Esto significa que el árbol está fuertemente balanceado. Esto asegura que las operaciones de inserción, eliminación y búsqueda tomen un tiempo $O(\log n)$.

Un Árbol rojo-negro debe satisfacer estas propiedades:

- La raíz es negra
- Todas las hojas son negras
- Los nodos rojos sólo pueden tener hijos negros
- Todos los caminos de un nodo a sus hojas contienen el mismo número de nodos negros

Una fuente común de confusión con estas propiedades es que se asume que todos los hijos en el árbol son *hojas nulas*, que no contienen datos y sólo sirven para indicar donde termina el árbol. Estos nodos son a menudo omitidos en los dibujos, resultando en árboles que parecen contradictorios a los anteriores principios, pero que realmente no lo son.



Cuando los nodos son removidos o borrados, el árbol debe ser transformado para mantener estas propiedades. Esto se hace repintando o rotando los nodos.

[editar]

Inserción

En el caso de la inserción, a lo más se realiza una rotación, en contraste con otros árboles balanceados, como por ejemplo, el árbol AVL.

Los nuevos nodos se insertan normalmente con el color rojo. Entonces puede ocurrir que:

- Si el nodo padre es negro, el árbol todavía es válido.
- Si el nodo padre es rojo y existe un nodo tío rojo, entonces ellos deben ser repintados de negro. y el nodo abuelo debe ser repintado de rojo (Puede ser necesario continuar repintando hacia arriba hasta llegar a la raíz).
- Cuando queda un nodo rojo con padre rojo se efectua una rotación. Si la raíz termina roja, ésta debe ser repintada de negro.

Cabe notar que al hacer una inserción a lo más se va a realizar una rotación simple.

Árbol AVL

De Wikipedia

Saltar a navegación, búsqueda

Árbol AVL es un término usado en computación para referirse a un tipo especial de árbol binario ideado por los matemáticos rusos **A**delson-**V**elskii y **L**andis. Fue el primer árbol de búsqueda binario auto-balanceable que se ideó.

Tabla de contenidos

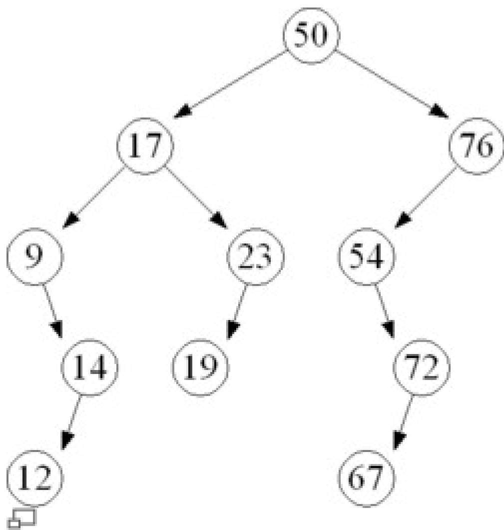
[ocultar]

- 1 Descripción
  - 1.1 Definición formal
    - 1.1.1 Definición del altura de un árbol
    - 1.1.2 Definición de árbol AVL
- 2 Factor de equilibrio
- 3 Operaciones
  - 3.1 Inserción
  - 3.2 Extracción
  - 3.3 Búsqueda
- 4 Véase también

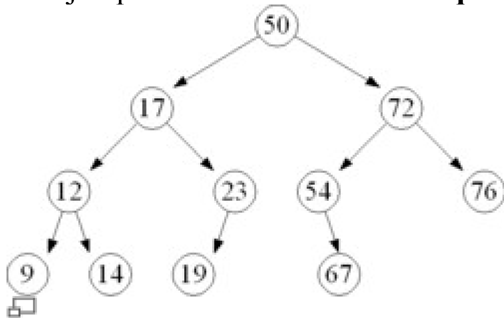
- 5 Enlaces externos

[editar]

Descripción



Un ejemplo de **árbol binario no equilibrado** (no es **AVL**)



Un ejemplo de **árbol binario equilibrado** (sí es **AVL**)

El **árbol AVL** toma su nombre de las iniciales de los apellidos de sus inventores, **A**delson-**V**elskii y **L**andis. Lo dieron a conocer en la publicación de un artículo en 1962: "An algorithm for the organization of information" (*"Un algoritmo para la organización de la información*").

Los **árboles AVL** están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad O(log n). El **factor de equilibrio** puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los subárboles.

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones de los nodos.

Los **árboles AVL** más profundos son los árboles de Fibonacci.

[editar]

Definición formal

[editar]

Definición del altura de un árbol

Sea T un árbol binario de búsqueda y sean T_i y T_d sus subárboles, su altura $H(T)$, es:

- 0 si el árbol T está vacío
- $1 + \max(H(T_i),H(T_d))$ si no lo está

[editar]

Definición de árbol AVL

Sea T un árbol binario de búsqueda con T_i y Td siendo sus subárboles izquierdo y derecho respectivamente, tenemos que:

- Si T es vacío, es un árbol AVL
- Si T es un ABB no vacío, es AVL sii (si y sólo si):
  - $T_i < math > y < math > T_d$ son AVL y
  - $H(T_i) - H(T_d) = -1, 0$ ó $+1$ (factor de equilibrio)

Por esta definición tenemos que el árbol de la figura de arriba no es AVL, mientras que el de abajo sí lo es. Véase también que se trata de un árbol ordenado, en el cual para cada nodo todos los nodos de su subárbol izquierdo tienen un valor de clave menor y todos los nodos de su subárbol derecho tienen un valor de clave mayor que el suyo, cumpliendo así la propiedad de los ABB.

[editar]

Factor de equilibrio

Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los árboles binarios de búsqueda (ABB), y además el dato que controla el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

> FE = altura subárbol derecho - altura subárbol izquierdo;
> Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

[editar]

Operaciones

Las operaciones básicas de un árbol AVL implican generalmente el realizar los mismos algoritmos que serían realizados en un árbol binario de búsqueda desequilibrado, pero precedido o seguido por una o más de las llamadas "rotaciones AVL".

[editar]

Inserción

La inserción en un árbol de AVL puede ser realizada insertando el valor dado en el árbol como si fuera un árbol de búsqueda binario desequilibrado y después retrocediendo hacia la raíz, rotando sobre cualquier nodo que pueda haberse desequilibrado durante la inserción.

Dado que como mucho un nodo es rotado 1,5 veces $\log n$ en la vuelta hacia la raíz, y cada rotación AVL tarda el mismo tiempo, el proceso de inserción tarda un tiempo total de $O(\log n)$.

[editar]

Extracción

El problema de la extracción puede resolverse en $O(\log n)$ pasos. Una extracción trae consigo una disminución de la altura de la rama donde se extrajo y tendrá como efecto un cambio en el factor de equilibrio del nodo padre de la rama en cuestión, pudiendo necesitarse una rotación.

Esta disminución de la altura y la corrección de los factores de equilibrio con sus posibles rotaciones asociadas pueden propagarse hasta la raíz.

[editar]

Búsqueda

Las búsquedas se realizan de la misma manera que en los ABB, pero al estar el árbol equilibrado la complejidad de la búsqueda nunca excederá de $O(\log n)$.

Árbol-B

De Wikipedia

(Redirigido desde B-Árbol)
Saltar a navegación, búsqueda

Los **Árboles-B** ó **B-Árboles** son estructuras de datos de árbol que se encuentran comúnmente en las implementaciones de bases de datos y sistemas de archivos. Fueron desarrollados en 1970 por Rudolf Bayer y Edward McCreight.

Mientras los árboles binarios pueden desbalancearse fácilmente a menos que algún proceso los vigile, los árboles-b garantizan balanceo de datos y un número de niveles (altura) menor. Esto permite que las inserciones, extracciones, búsqueda y modificaciones sobre grandes cantidades de datos se realicen de una manera eficiente. Las inserciones y extracciones se realizan en tiempo logarítmico amortizado.

Tabla de contenidos

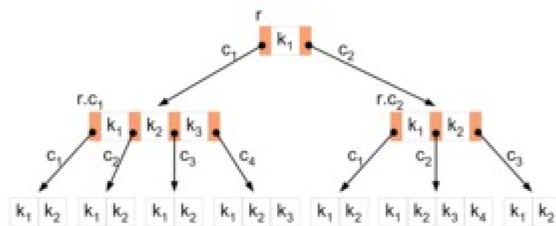
[ocultar]

- 1 Definición
- 2 Algoritmos
  - 2.1 Búsqueda
  - 2.2 Inserción
  - 2.3 Borrado
- 3 Véase también

- 4 Enlaces externos

[editar]

Definición



Definición de Árbol B

Un árbol-B de grado *m* (donde m es el número de claves máximas permitidas en un nodo) cumple las siguientes propiedades:

- Cada nodo tiene *m* o menos hijos.
- Cada nodo, excepto la raíz, tiene al menos sup(m/2) hijos.
- La raíz tiene al menos 2 hijos (a menos que sea hoja).
- Todos los nodos hoja, es decir nodos con enlaces nulos, aparecen al mismo nivel del árbol.
- Si tengo 'n' hojas o tallos en un nodo, ese nodo tendrá n-1 claves.

La idea tras los árboles-B es que los nodos internos deben tener un número variable de nodos hijo dentro de un rango predefinido. Esto causa que los árboles-B no necesiten rebalancearse tan frecuentemente como los árboles AVL. Los límites superior e inferior en el número de nodos hijo son definidos al hacer una implementación en particular. Por ejemplo, en un **2-3 B-Árbol** (A

menudo simplemente llamado **2-3 árbol**), cada nodo sólo puede tener 2 o 3 nodos hijos. Se considera que un nodo se encuentra en un estado ilegal si tiene un número inválido de nodos hijo.

[editar]

Algoritmos

[editar]

Búsqueda

La búsqueda es similar a la de los árboles binarios. Se empieza en la raíz, y se recorre el árbol hacia abajo, escogiendo el sub-nodo de acuerdo a la posición relativa del valor buscado respecto a los valores de cada nodo. Típicamente se utiliza la búsqueda binaria para determinar esta posición relativa.

[editar]

Inserción

1. Primero se inserta el nuevo valor en el nodo que le corresponde y después se corrigen todos los nodos ilegales. **Se dice que un nodo está en estado "ilegal" si tiene más elementos de los permitidos.**
2. Si un nodo tiene demasiados elementos, se separa en dos nodos. Si se separa el nodo raíz, entonces se crea un nuevo nodo raíz.

[editar]

Borrado

1. Primero se busca el valor a borrar y se remueve del nodo que lo contiene.
2. Si ningún nodo está en estado ilegal, entonces el proceso está terminado.
3. Si algún nodo está en estado ilegal, hay dos opciones:
    1. Un nodo hermano puede transferir uno o más hijos al nodo ilegal para corregirlo.
    2. Si el hermano no tiene hijos extra, entonces los hermanos se fusionan en un solo nodo.

Árbol multicamino

De Wikipedia

Saltar a navegación, búsqueda

Los **árboles multicamino** o **árboles multirrama** son estructuras de datos de tipo árbol usadas en computación.

Tabla de contenidos

[ocultar]

- 1 Definición
- 2 Ventajas e inconvenientes
    - 2.1 Nota
- 3 Véase también

[editar]

Definición

Un **árbol multicamino** posee un grado *g* mayor a dos, donde cada nodo de información del árbol tiene un máximo de *g* hijos.

Sea un árbol de m-caminos *A*, es un árbol m-caminos si y solo si:

- *A* está vacío
- Cada nodo de A muestra la siguiente estructura:
  $[nClaves, Enlace_0, Clave_1, ..., Clave_{nClaves}, Enlace_{nClaves}]$

  nClaves es el número de valores de clave de un nodo, pudiendo ser: $0 <= nClaves <= g-1$
  $Enlace_i$, son los enlaces a los subárboles de *A*, pudiendo ser: $0 <= i <= nClaves$
  $Clave_i$, son los valores de clave, pudiendo ser: $1 <= i <= nClaves$

- $Clave_i < Clave_{i+1}$
- Cada valor de clave en el subárbol $Enlace_i$ es menor que el valor de $Clave_{i+1}$
- Los subárboles $Enlace_i$, donde $0 <= i <= nClaves$, son también árboles m-caminos.

Existen muchas aplicaciones en las que el volumen de la información es tal, que los datos no caben en la memoria principal y es necesario almacenarlos, organizados en archivos, en dispositivos de almacenaminento secundario. Esta organización de archivos debe ser suficientemente adecuada como para recuperar los datos del mismo en forma eficiente.

[editar]

Ventajas e inconvenientes

La principal ventaja de este tipo de árboles consiste en que existen más nodos en un mismo nivel que en los árboles binarios con lo que se consigue que, si el árbol es de búsqueda, los accesos a los nodos sean más rápidos.

El inconveniente más importante que tienen es la mayor ocupación de memoria, pudiendo ocurrir que en ocasiones la mayoría de los nodos no tengan descendientes o al menos no todos los que podrían tener desaprovechándose por tanto gran cantidad de memoria. Cuando esto ocurre lo más frecuente es transformar el árbol multicamino en su binario de búsqueda equivalente.

[editar]

Nota

Un tipo especial de árboles multicamino utilizado para solucionar el problema de la ocupación de memoria son los árboles B o árboles Bayer

5.2 Operaciones Básicas sobre árboles binarios.

Operaciones básicas de los árboles binarios de búsqueda

Como en toda estructura de datos hay dos operaciones básicas, inserción y eliminación.

Inserción

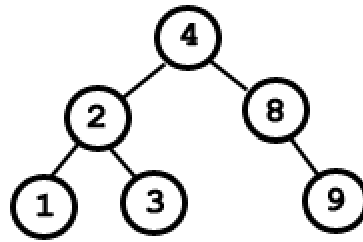
El procedimiento de inserción en un árbol binario de búsqueda es muy sencillo, únicamente hay que tener cuidado de no romper la estructura ni el orden del árbol.

Cuando se inserta un nuevo nodo en el árbol hay que tener en cuenta que cada nodo no puede tener más de dos hijos, por esta razón si un nodo ya tiene 2 hijos, el nuevo nodo nunca se podrá insertar como su hijo. Con esta restricción nos aseguramos mantener la estructura del árbol, pero aún nos falta mantener el orden.

Para localizar el lugar adecuado del árbol donde insertar el nuevo nodo se realizan comparaciones entre los nodos del árbol y el elemento a insertar. El primer nodo que se compara es la raíz, si el nuevo nodo es menor que la raíz, la búsqueda prosigue por el nodo izquierdo de éste. Si el nuevo nodo fuese mayor, la búsqueda seguiría por el hijo derecho de la raíz.

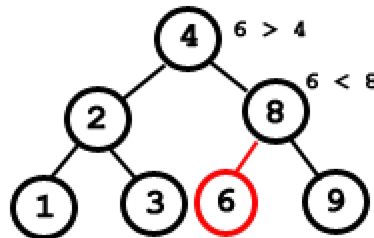
Este procedimiento es recursivo, y su condición de parada es llegar a un nodo que no tenga hijo en la rama por la que la búsqueda debería seguir. En este caso el nuevo nodo se inserta en ese hueco, como su nuevo hijo.

Vamos a verlo con un ejemplo sobre el siguiente árbol:



Se quiere insertar el elemento 6.

Lo primero es comparar el nuevo elemento con la raíz. Como 6 > 4, entonces la búsqueda prosigue por el lado derecho. Ahora el nuevo nodo se compara con el elemento 8. En este caso 6 < 8, por lo que hay que continuar la búsqueda por la rama izquierda. Como la rama izquierda de 8 no tiene ningún nodo, se cumple la condición de parada de la recursividad y se inserta en ese lugar el nuevo nodo.

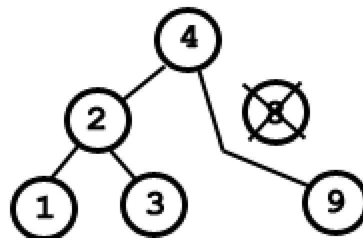


Borrar

El borrado en árboles binarios de búsqueda es otra operación bastante sencilla excepto en un caso. Vamos a ir estudiando los distintos casos.

Tras realizar la búsqueda del nodo a eliminar observamos que el nodo no tiene hijos. Este es el caso más sencillo, únicamente habrá que borrar el elemento y ya habremos concuído la operación.

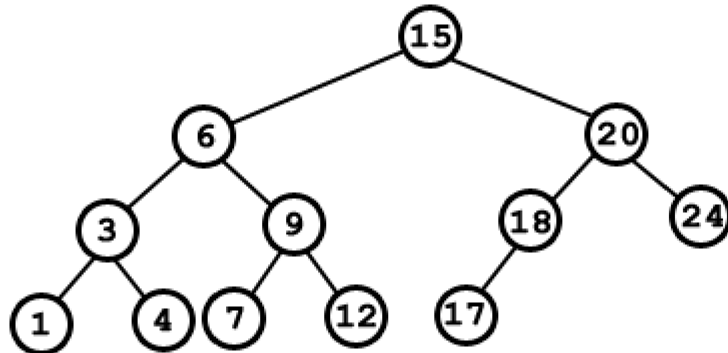
Si tras realizar la búsqueda nos encontramos con que tiene un sólo hijo. Este caso también es sencillo, para borrar el nodo deseado, hacemos una especie de *puente*, el padre del nodo a borrar pasa a apuntar al hijo del nodo borrado.



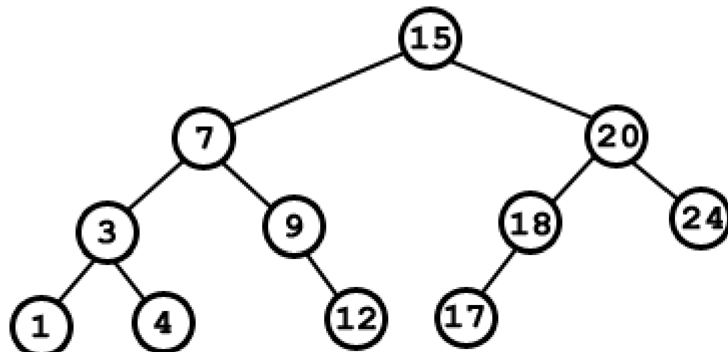
Por último, el caso más complejo, si el nodo a borrar tiene dos hijos. En este caso se debe sustitui el nodo a borrar por mayor de los nomdos menores del nodo borrado, o por el menor de los nodos mayores de dicho nodo. Una vez realizada esta sustitución se borrra el nodo que sustituyó al nodo eliminado (operación sencilla ya que este nodo tendrá un hijo a lo sumo).

Sobre el siguiente árbol queremos eliminar el elemento 6. Tenemos dos opciones para sustituirlo:

- El menor de sus mayores: 7.
- El mayor de sus menores: 4.



Vamos a sustituirlo por el 7 (por ejemplo). El árbol resultante sería el siguiente, tras eliminar también el elemento 7 de su ubicación original.



Otras operaciones

En los árboles de búsqueda la operación buscar es muy eficiente. El algoritmo compara el elemento a buscar con la raíz, si es menor continua la búsqueda por la rama izquierda, si es mayor continua por la izquierda. Este procedimiento se realiza recursivamente hasta que se encuentra el nodo o hasta que se llega al final del árbol.

Otra operación importante en el árbol es el recorridod el mismo. El recorrido se puede realizar de tres formas diferentes:

- Preorden: Primero el nodo raíz, luego el subárbol izquierdo y a continuación el subárbol derecho.
- Inorden: Primero el subárbol izquierdo, luego la raíz y a continuación el subárbol derecho.
- Postorden: Primero el subárbol izquierdo, luego el subárbol derecho y a continuación la raíz.

5.2.1 Creación.

Estructuras de datos avanzadas.

Tablas de dispersión.

Qué es una tabla de dispersión.

Las tablas de dispersión, más conocidas como tablas *hash*, son unas de las estructuras de datos más frecuentemente usadas. Para tener una idea inicial, las tablas de dispersión posibilitan tener una estructura que relaciona una clave con un valor, como un diccionario. Internamente, las tablas de dispersión son un array. Cada una de las posiciones del array puede contener ninguna, una o varias entradas del diccionario. Normalmente contendrá una como máximo, lo que permite un acceso rápido a los elementos, evitando realizar una búsqueda en la mayoría de los casos. Para saber en qué posición del array se debe buscar o insertar una clave, se utiliza una función de dispersión. Una función de dispersión relaciona a cada clave con un valor entero. Dos claves iguales deben tener el mismo valor de dispersión, también llamado *hash value*, pero dos claves distintas pueden tener el mismo valor de dispersión, lo cual provocaría una colisión.

El valor de dispersión es un entero sin signo entre 0 y el máximo entero de la plataforma, por lo que la tabla de dispersión usa el resto de dividir el valor de dispersión entre el tamaño del array para encontrar la posición. Cuando dos claves tienen que ser almacenadas en la misma posición de la tabla se produce una colisión. Esto puede ser debido a que la función de dispersión no distribuye las claves lo suficiente, o a que hay más claves en la tabla *hash* que el tamaño del array. En el segundo caso, GLib se encargará de redimensionar el array de forma automática.

Para aclarar los conceptos, se examinará el siguiente ejemplo a lo largo de todo el capítulo con el fin de facilitar la comprensión del mismo. Póngase que se desea tener una relación de la información de los activistas de GNOME. Para ello se usará una tabla de dispersión usando como clave el apodo que usa cada desarrollador dentro del proyecto. Y como valor una relación de sus datos personales. Esta relación de datos personales vendría dada por la siguiente estructura.

```
struct
DatosDesarrollador {
   gchar *apodo;
   gchar *nombre;
   gchar *proyecto;
   gchar *telefono;
};
```

Ahora, si se hace un recuento de datos, se obtiene una curiosa información. Por ejemplo, si tenemos en cuenta que cada apodo tiene como mucho diez caracteres y que el alfabeto tiene veintiseis letras, el resultado es que tendremos 26^10 posibles llaves, lo que supera con creces el número de claves que vamos a utilizar. Con ésto se quiere hacer hincapié en que el uso de esta estructura es útil cuando el número de llaves libres excede con mucho a las llaves que van a ser ocupadas.

Cómo se crea una tabla de dispersión.

Para crear una tabla de dispersión se tiene que indicar la función de dispersión y la función que se utilizará para comparar dos claves. Estas dos funcione se deben pasar como parámetros a la función `g_hash_table_new`

```
GHashTable* g_hash_table_new (GHashFunc funcion_de_dispersion , GEqualFunc
funcion_de_comparación );
```

GLib tiene implementadas una serie de funciones de dispersión y comparación. De este modo, el desarrollador no ha de reinventar la rueda. Las funciones de dispersión trabajan siempre de la misma manera. A este tipo de funciones se les pasa un valor y devuelven la clave. En cuanto a las funciones de comparación, se les pasa como parámetros dos valores y devuelve *`TRUE`* si son los mismos valores y *`FALSE`* si no son iguales.

Estas funciones son implementadas basándose en el siguiente esquema. Si se diesa el caso de que las existentes en GLib no satisfacen sus necesidades, lo único que tendrá que hacer será implementar un par de funciones que siguieran estos esquemas.

```
guint (*GHashFunc) (gconstpointer clave );

gboolean (*GEqualFunc) (gconstpointer clave_A , gconstpointer clave_B );
```

Una vez se tiene presente este esquema, se puede comprender mejor el funcionamiento de las funciones que se describirán a continuación. La primera pareja de funciones son `g_str_hash` y `g_str_equal`, que son de dispersión y de comparación respectivamente. Con la función `g_str_hash` podremos convertir una clave en forma de cadena en un valor de dispersión. Lo cual quiere decir que el desarrollador podrá usar, con estas funciones, una clave en forma de cadena para la tabla *hash*.

Otras funciones de carácter similar son `g_int_hash`, `g_int_equal`, `g_direct_hash` o `g_direct_equal`, que posibilitan usar como claves para la tabla *hash* números enteros y punteros genéricos respectivamente.

Ahora, siguiendo con el ejemplo que se comenzó al principio del capitulo, se pasará a ver cómo se creará la tabla de dispersión que contendrá la información de los desarrolladores de GNOME. Recuérdese que se iba a usar, como clave, el apodo del desarrollador. Así que, como función de dispersión y de comparación, se usaran `g_str_hash` y `g_str_equal`, respectivamente. Ya que el apodo es una cadena y estas funciones satisfacen perfectamente el fin que se persigue.

Cómo manipular un tabla de dispersión.

Ahora se pasará a describir las funciones de manipulación de este TAD. Para añadir y eliminar elementos de una tabla se dispone de estas dos funciones: `g_hash_table_insert` y `g_hash_table_remove`. La primera insertará un valor en la tabla *hash* y le indicará la clave con

la que después podrá ser referenciado. En cuanto a la segunda, borrará el valor que corresponda a la clave que se le pase como parámetro a la función.

```
void g_hash_table_insert (GHashTable tabla_hash , gpointer clave , gpointer
valor );
```

```
gboolean g_hash_table_remove (GHashTable tabla_hash , gpointer clave );
```

En caso que se desee modificar el valor asociado a una clave, hay dos opciones a seguir en forma de funciones: `g_hash_table_insert` (vista anteriormente) o `g_hash_table_replace`. La única diferencia entre ambas es qué pasa cuando la clave ya existe. En el caso de la primera, se conservará la clave antigua mientras que, en el caso de la segunda, se usará la nueva clave. Obsérvese que dos claves se consideran la misma si la función de comparación devuelve *TRUE*, aunque sean diferentes objetos.

```
void g_hash_table_replace (GHashTable tabla_hash , gpointer clave , gpointer
valor );
```

Búsqueda de información dentro de la tabla.

Llegados a este punto, en el que se ha explicado como insertar, borrar y modificar elementos dentro de una tabla de este tipo, parece obvio que el siguiente paso es cómo conseguir encontrar información dentro de una tabla de dispersión. Para ello se dispone de la función `g_hash_table_lookup`. Esta función tiene un funcionamiento muy simple al igual que las funciones anteriormente explicadas. Lo único que necesita es que se le pase como parámetro la tabla de dispersión y la clave que desea buscar en la misma y la función devolverá un puntero al valor asociado a la clave en caso de existir o el valor *NULL* en caso de no existir.

```
gpointer g_hash_table_lookup (GHashTable tabla_hash , gpointer clave );
```

También se dispone de otra función para realizar las búsquedas de datos dentro de una tabla de dispersión. Esta función es más compleja en su uso pero provee de un sistema de búsqueda más eficiente que el anterior que ayudará a suplir las necesidades más complejas de un desarrollador.

```
gboolean g_hash_table_lookup_extended (GHashTable tabla_hash , gconstpointer
clave_a_buscar , gpointer* clave_original , gpointer* valor );
```

La función anterior devolverá *TRUE* si encuentra el valor buscado, o *FALSE* en caso contrario. En el supuesto de encontrar el valor, el puntero *clave_original* apuntará a la clave con la que fue almacenado en la tabla, y el puntero *valor* apuntará al valor almacenado. Téngase en cuenta que la clave usada para la realizar la búsqueda, *clave_a_buscar*, no tiene por qué ser la misma que la encontrada, *clave_original*, aunque ambas serán iguales según la función de comparación que hayamos indicado al crear la tabla de dispersión.

Manipulación avanzada de tablas de dispersión.

Arboles binarios balanceados

Los árboles binarios balanceados son estructuras de datos que almacenan pares clave - dato, de manera ordenada según las claves, y posibilitan el acceso rápido a los datos, dada una clave particular, y recorrer todos los elementos en orden.

Son apropiados para grandes cantidades de información y tienen menor *overhead* que una tabla de dispersión, aunque el tiempo de acceso es de mayor complejidad computacional.

Si bien internamente la forma de almacenamiento tiene estructura de árbol, esto no se exterioriza en la API, haciendo que el manejo de los datos resulte transparente. Se denominan balanceados porque, cada vez que se modifican, las ramas se rebalancean de tal forma que la altura del árbol sea la mínima posible, acortando de esta manera el tiempo promedio de acceso a los datos.

Creación de un árbol binario.

Para crear un árbol binario balanceado es necesaria una función de comparación que pueda ordenar un conjunto de claves. Para ello se adopta la convención utilizada por `strcmp`. Esto es, dados dos valores, la función devuelve 0 si son iguales, un valor negativo si el primer parámetro es anterior al segundo, y un valor positivo si el primero es posterior al segundo. La utilización de esta función permite flexibilidad en cuanto a claves a utilizar y en como se ordenan. El prototipo es el siguiente:

```
gint (*GCompareFunc)(gconstpointer a, gconstpointer b);
```

o bien, para el caso en que sea necesario proveer algún dato adicional a la función:

```
gint (*GCompareDataFunc)(gconstpointer a, gconstpointer b, gpointer
user_data);
```

Una vez definida dicha función el árbol se crea con alguna de las siguientes formas:

```
GTree *g_tree_new(GCompareFunc key_compare_func);

GTree *g_tree_new_with_data(GCompareDataFunc key_compare_func, gpointer
key_compare_data);

GTree *g_tree_new_full(GCompareDataFunc key_compare_func, gpointer
key_compare_data, GDestroyNotify key_destroy_func, GDestroyNotify
value_destroy_func);
```

donde `key_compare_func` es la función previamente mencionada, `key_compare_data` es un dato arbitrario a enviar a la función de comparación y `key_destroy_func` y `value_destroy_func` funciones de retrollamada cuando alguna clave o dato se eliminan del árbol.

En caso de no utilizar la última función, es tarea del usuario liberar la memoria utilizada por claves y/o datos cuando se destruye el árbol o cuando se elimina algún dato del mismo utilizando `g_tree_remove`.

Para destruir un árbol se utiliza la función `g_tree_destroy`.

```
void g_tree_destroy(GTree *tree);
```

Agregar y eliminar elementos a un árbol binario.

Para insertar un nuevo elemento en el árbol se utiliza `g_tree_insert` o `g_tree_replace`. La diferencia entre ambas vale sólo en el caso de que en el árbol ya exista una clave igual a la que se intenta agregar y sólo cuando, en la creación del árbol, se especificó una función de destrucción de claves. Para `g_tree_insert` la clave que se pasó como parámetro se destruye, llamando a `key_destroy_func` y la del árbol permanece inalterada. Para el caso de `g_tree_replace`, la clave ya existente se destruye y se reemplaza por la nueva. En ambos casos, de existir el elemento, el dato anterior se destruye llamando a `value_destroy_func`, siempre que se haya especificado en la creación del árbol.

```
void g_tree_insert(GTree *tree, gpointer key, gpointer value);

void g_tree_replace(GTree *tree, gpointer key, gpointer value);
```

`key` es la clave con la cual se recuperará el dato luego y *`value`* el dato en sí. Para eliminar un elemento del árbol se pueden utilizar `g_tree_remove` o `g_tree_steal`. La diferencia entre ambas es que, si se invoca `g_tree_steal` no se destruyen la clave y el valor aunque se hayan especificado funciones para tal fin en la creación del árbol.

```
void g_tree_remove(GTree *tree, gconstpointer key);

void g_tree_steal(GTree *tree, gconstpointer key);
```

Búsqueda y recorrida en un árbol binario.

Existen dos funciones para buscar en un árbol binario, similares a las utilizadas en tablas de *hash*: `g_tree_lookup` y `g_tree_lookup_extended`.

```
gpointer g_tree_lookup(GTree *tree, gconstpointer key);
```

En este caso, se busca un elemento del árbol cuya clave sea igual (en los términos de la función especificada al momento de la creación) a *`key`*. Devuelve el dato del elemento encontrado. Si la búsqueda fue infructuosa devuelve `NULL`.

```
gboolean g_tree_lookup_extended(GTree *tree, gconstpointer lookup_key,
gpointer *orig_key, gpointer *value);
```

Esta función no sólo busca el elemento cuya clave coincida con la especificada, sino que además devuelve la clave y dato del elemento encontrado en los parámetros de salida *`orig_key`* y *`value`*. A diferencia del caso anterior, la función devuelve un booleano indicando si la búsqueda fue exitosa o no.

La versión extendida de la búsqueda es particularmente útil cuando se quiere eliminar un elemento del árbol sin destruirlo, utilizando la función `g_tree_steal` (p.e. para ser agregado a otro árbol).

Por último para recorrer todos los elementos de un árbol se utiliza `g_tree_foreach`. Los elementos son tomados en orden y pasados como parámetro a la función de iteración especificada.

```
void g_tree_foreach(GTree *tree, GTraverseFunc func, gpointer user_data);
```

Es importante decir que durante el recorrido, no se puede modificar el árbol (agregar y/o eliminar elementos). Si se quieren eliminar algunos elementos, se deben agregar las claves a una lista (o cualquier estructura auxiliar) y finalizada la iteración proceder a eliminar uno por uno los elementos seleccionados. La función de iteración debe tener la siguiente forma:

```
gboolean (*GTraverseFunc)(gpointer key, gpointer value, gpointer data);
```

Si dicha función devuelve `TRUE`, la iteración se interrumpe. *`data`* es el valor que se especificó como *`user_data`* en la función `g_tree_foreach`.

GNode : Arboles de orden n.

Para representar datos de manera jerárquica, GLib ofrece el tipo de dato GNode que permite representar árboles de cualquier orden.

Al igual que con las listas y a diferencia de árboles binarios balanceados o tablas *hash*, los GNode son elementos independientes: sólo hay conexiones entre ellos, pero nada en la estructura dice, por ejemplo, cuál es la raíz del árbol. `NULL` es el árbol vacío. La estructura GNode tiene la siguiente forma:

```
struct GNode {
   gpointer  data;
   GNode    *next;
   GNode    *prev;
   GNode    *parent;
   GNode    *children;
};
```

data contiene el dato en sí del nodo. *parent* apunta al nodo padre: éste es `NULL` para el nodo raíz. *children* apunta al primer hijo del nodo, accediéndose a los demás hijos mediante los campos *next* y *prev*.

Agregar nodos a un árbol.

GLib tiene una serie de funciones para agregar nodos a un árbol. Se puede crear el nodo aislado primero e insertarlo en una posición dada, pero también hay macros definidas que crean el nodo y lo insertan directamente. No hay diferencia en el resultado final, simplemente una línea menos de código.

Para crear un nodo se utiliza `g_node_new`:

```
GNode *g_node_new(gpointer data);
```

Una vez obtenido el GNode, se inserta a un árbol especificando el nodo padre (*parent*) ya existente y la posición relativa entre los hijos de dicho padre. Hay cinco funciones diferentes en función de la posición y las condiciones en que se quiera insertar. En todos los casos el valor devuelto es el mismo nodo *node* insertado.

```
GNode *g_node_insert(GNode *parent, gint position, GNode *node);

GNode *g_node_insert_before(GNode *parent, GNode *sibling, GNode *node);

GNode *g_node_insert_after(GNode *parent, GNode *sibling, GNode *node);

GNode *g_node_append(GNode *parent, GNode *node);

GNode *g_node_prepend(GNode *parent, GNode *node);
```

sibling es un nodo hijo de *parent* que se toma como referencia para insertar el nuevo nodo antes o después de él mismo. En ambos casos, `g_node_insert_after` y `g_node_insert_before`, si *sibling* es `NULL`, el nuevo nodo se inserta al final de la lista de hijos.

Con `g_node_n_children` se obtiene la cantidad de hijos de un nodo y, a partir de este dato, se puede especificar una posición relativa dentro de la lista de hijos *posittion*. Las formas `g_node_append` y `g_node_prepend` agregan el nuevo nodo al final o al principio de la lista de hijos.

Cuatro de estas cinco funciones tienen su equivalente de inserción de nodo directa a partir de el dato. Como se dijo antes, la diferencia es que en estos casos se especifica el dato y la creación del nodo es transparente al usuario aunque, de ser requerido, se retorna en el valor de la función. Estas cuatro nuevas formas son macros que utilizan las funciones anteriores. En todos los casos, *data* es el dato que contendrá el nuevo nodo.

```
GNode *g_node_insert_data(GNode *parent, gint position, gpointer data);

GNode *g_node_insert_data_before(GNode *parent, GNode *sibling, gpointer
data);

GNode *g_node_append_data(GNode *parent, gpointer data);

GNode *g_node_prepend_data(GNode *parent, gpointer data);
```

Eliminar Vs. desvincular.

Al igual que con las listas, hay dos formas de quitar un dato de un árbol: `g_tree_unlink` y `g_tree_destroy`.

```
void g_node_unlink(GNode *node);
```

```
void g_node_destroy(GNode *node);
```

`g_node_unlink` quita el nodo especificado del árbol, resultando en un nuevo árbol que lo tiene como raíz. `g_node_destroy` no sólo elimina el nodo del árbol, sino que, además, libera toda la memoria utilizada por el nodo y por sus hijos. GLib, sin embargo, no tiene forma de liberar la memoria de los datos que contenían los nodos: eso es responsabilidad del usuario.

Información sobre los nodos.

`G_NODE_IS_LEAF` devuelve `TRUE` si el nodo es un terminal u hoja del árbol. En otras palabras si el campo *children* es `NULL`. Análogamente, `G_NODE_IS_ROOT` devuelve `TRUE` si el nodo especificado es la raíz del árbol, o bien, si el campo *parent* es `NULL`.

`g_node_depth` devuelve la profundidad de un nodo (cuantos niveles hay hasta subir a la raíz); `g_node_n_children` la cantidad de nodos hijos inmediatos y `g_node_max_height`, la cantidad máxima de niveles inferiores (es decir, iterando hacia los hijos). `g_node_depth` y `g_node_max_height` miden alturas. Un árbol vacío tiene altura 0, un único nodo 1 y así sucesivamente.

```
guint g_node_n_nodes(GNode *node, GTraverseFlags flags);
```

`g_node_n_nodes` recorre el árbol desde el nodo especificado y cuenta los nodos. El parámetro *`flags`* indica qué nodos debe contar y puede tener como valores:

`G_TRAVERSE_LEAFS`: Sólo contar los nodos terminales
`G_TRAVERSE_NON_LEAFS`: Sólo contar los nodos intermedios
`G_TRAVERSE_ALL`: Contar todos los nodos

```
gboolean g_node_is_ancestor(GNode *node, GNode *descendant);
```

`g_node_is_ancestor` devolverá `TRUE` si *`node`* es un ancestro (padre, padre del padre, etc.) de *`descendant`*.

`g_node_get_root` devuelve la raíz del árbol que contiene al nodo especificado.

```
gint g_node_child_index(GNode *node, gpointer data);
```

```
gint g_node_child_position(GNode *node, GNode *child);
```

`g_node_child_index` y `g_node_child_position` son funciones similares que devuelven la posición de un hijo en la lista de hijos de un nodo (la primera busca el dato en lugar del nodo en sí). En caso de que el nodo no sea hijo del padre especificado, ambas funciones devolverán -1.

Para acceder a los hijos de un nodo dado se pueden utilizar los campos de la estructura o funciones que provee GLib: `g_node_first_child`, `g_node_last_child` y `g_node_nth_child`.

En forma similar, se puede recorrer la lista de hijos usando los campos *prev* y *next*, o mediante `g_node_first_sibling`, `g_node_prev_sibling`, `g_node_next_sibling` y `g_node_last_sibling`.

Buscar en el árbol y recorrerlo.

Para buscar un nodo determinado, GLib tiene dos funciones:

```
GNode *g_node_find(GNode *node, GTraverseType order, GTraverseFlags flags,
gpointer data);

GNode *g_node_find_child(GNode *node, GTraverseFlags flags, gpointer data);
```

`g_node_find` y `g_node_find_child` buscan a partir de *`node`* el nodo del árbol que contenga el *`data`* especificado. `g_node_find` busca en todo el árbol, mientras que `g_node_find_child` se limita a los hijos inmediatos del nodo.

En ambos casos *`flags`* especifica que clase de nodos se buscarán, y en `g_node_find`, *`order`* indica el orden de búsqueda. Este último puede tener uno de estos cuatro valores:

- `G_IN_ORDER`: para cada nodo, visitar el primer hijo, luego el nodo mismo y por último el resto de los hijos.
- `G_PRE_ORDER`: para cada nodo, visitar primero el nodo y luego los hijos.
- `G_POST_ORDER`: para cada nodo, visitar primero los hijos y luego el nodo.
- `G_LEVEL_ORDER`: visitar los nodos por niveles, desde la raíz (es decir la raíz, luego los hijos de la raíz, luego los hijos de los hijos de la raíz, etc.)

Al igual que con otros tipos de dato, es posible recorrer el árbol con dos funciones de GLib, que son análogas a las de búsqueda:

```
void g_node_traverse(GNode *root, GTraverseType order, GTraverseFlags flags,
gint max_depth, GNodeTraverseFunc func, gpointer data);

void g_node_children_foreach(GNode *node, GTraverseFlags flags,
GNodeForeachFunc func, gpointer data);
```

`max_depth` especifica la profundidad máxima de iteración. Si este valor es -1 se recorre todo el árbol; si es 1 sólo *`root`*; y así sucesivamente. *`order`* indica cómo recorrer los nodos y *`flags`* qué clase de nodos visitar. Notar que al igual que `g_node_find_child`, `g_node_children_foreach` se limita a los hijos directos del nodo.

La forma de las funciones de iteración es la siguiente:

```
gboolean (*GNodeTraverseFunc)(GNode *node, gpointer data);

gboolean (*GNodeForeachFunc)(GNode *node, gpointer data);
```

Al igual que con las funciones de iteración de los árboles binarios, en caso de necesitar interrumpir la iteración, la función debe devolver `TRUE`.

Caches

Los cachés son algo que, tarde o temprano, cualquier programador acaba implementando, pues permiten el almacenamiento de datos costosos de conseguir (un fichero a través de la red, un informe generado a partir de datos en una base de datos, etc) para su posterior consulta, de forma que dichos datos sean obtenidos una única vez, y leídos muchas.

Como librería de propósito general que es, GLib incluye, no un sistema de caché superfuncional, si no una infraestructura básica para que se puedan desarrollar cachés de todo tipo. Para ello, GLib incluye `GCache`, que, básicamente, permite la compartición de estructuras complejas de datos. Esto se usa, principalmente, para el ahorro de recursos en las aplicaciones, de forma que, si se tienen estructuras de datos complejas usadas en distintas partes de la aplicación, se pueda compartir una sola copia de dicha estructura compleja de datos entre todas esas partes de la aplicación.

Creación del caché.

`GCache` funciona de forma muy parecida a las tablas de claves (`GHashTable`), es decir, mediante el uso de claves para identificar cada objeto, y valores asociados con esas claves.

Pero el primer paso, como siempre, es crear el caché en cuestión. Para ello se usa la función `g_cache_new`, que, a primera vista, parece un tanto compleja:

```
GCache g_cache_new (GCacheNewFunc value_new_func, GCacheDestroyFunc
value_destroy_func, GCacheDupFunc key_dup_func, GCacheDestroyFunc
key_destroy_func, GHashFunc hash_key_func, GHashFunc hash_value_func,
GEqualFunc key_equal_func);
```

Todos los parámetros que recibe esta función son parámetros a funciones y, para cada uno de ellos, se debe especificar la función que realizará cada una de las tareas, que son:

- Creación de nuevas entradas en el caché (*`value_new_func`*). Esta función será llamada por `g_cache_insert` (ver más adelante) cuando se solicite la creación de un elemento cuya clave no existe en el caché.
- Destrucción de entradas en el caché (*`value_destroy_func`*). En esta función, que es llamada cuando se destruyen entradas en el caché, se debe liberar toda la memoria alojada para la entrada del caché que va a ser destruida. Normalmente, esto significa liberar toda la memoria alojada en `value_new_func`.
- Duplicación de claves. Puesto que las claves son asignadas fuera del interfaz de `GCache` (es decir, por la aplicación que haga uso de `GCache`), es tarea de la aplicación la creación o duplicación de claves. Así, esta función es llamada cada vez que `GCache` necesita duplicar una clave.
- Destrucción de claves. Al igual que en el caso de las entradas, las claves también deben ser destruidas, liberando toda la memoria que estén ocupando.
- Gestión de la tabla de claves incorporada en `GCache`. Al igual que cuando se usan los `GHashTable`, con `GCache` tambien es necesario especificar las funciones de manejo de la tabla de claves asociada, y para ello se usan los tres últimos parámetros de `g_cache_new`.

Como se puede observar, `GCache` está pensado para ser extendido, no para ser usado directamente. Esto es a lo que se hacía referencia en la introducción, de que es una infraestructura para la implementación de cachés. `GCache` implementa la lógica del caché, mientras que el manejo de los datos de ese caché se dejan de la mano de la aplicación, lo cual se obtiene mediante la implementación de todas estas funciones explicadas en este punto.

Es necesario en este punto hacer un pequeño alto y ver con detalle cómo debe ser la implementación de estas funciones.

Gestión de los datos del caché.

Una vez que el manejo de los datos del caché están claramente definidos en las funciones comentadas, llega el momento de usar realmente el caché. Para ello, se dispone de un amplio conjunto de funciones.

La primera operación que viene a la mente es la inserción y obtención de datos de ese caché. Esto se realiza mediante el uso de una única función: `g_cache_insert`.

```
gpointer g_cache_insert (GCache *cache, gpointer key);
```

Esta función busca el objeto con la clave *key* especificada. Si lo encuentra ya disponible en el caché, devuelve el valor asociado a esa clave. Si no lo encuentra, lo crea, para lo cual llama a la función especificada en el parámetro *value_new_func* en la llamada a `g_cache_new`. Así mismo, si el objeto no existe, la clave es duplicada, para lo cual, como no podía ser de otra forma, se llama a la función especificada en el parámetro `key_dup_func` de `g_cache_new`.

Podría ser una buena idea usar cadenas como claves para especificar determinados recursos y, mediante esos identificadores, crear la estructura compleja de datos asociada en la función de creación de entradas del caché. De esta forma, por ejemplo, se podría implementar fácilmente un caché de ficheros obtenidos a través de diferentes protocolos: las claves usadas serían los identificadores únicos de cada fichero (URLs), mientras que la entrada en el caché contendría el contenido mismo del fichero.

Pero, puesto que no siempre se accede al contenido del caché conociendo de antemano las claves, GLib incluye funciones que permiten acceder secuencialmente a todos los contenidos del caché. Esto se puede realizar de dos formas, y, por tanto, con dos funciones distintas:

```
void g_cache_key_foreach(GCache *cache, GHFunc func, gpointer user_data);

void g_cache_value_foreach(GCache *cache, GHFunc func, gpointer user_data);
```

Ambas funciones operan de la misma forma, que es llamar retroactivamente a la función especificada en el parámetro *func* por cada una de las entradas en el caché. La diferencia entre ellas es que `g_cache_key_foreach` opera sobre las claves y `g_cache_value_foreach` opera sobre los valores.

En cualquiera de los dos casos, cuando se realice una llamada a una de estas funciones, se deberá definir una función con la siguiente forma:

```
        void foreach_func (gpointer key, gpointer value, gpointer user_data);
```

En *key* se recibe la clave de cada una de las entradas (una cada vez), mientras que *value* recibe el valor de dicha entrada. Por su parte, *user_data* es el mismo dato que el que se especifica en la llamada a g_cache_*_foreach, y que permite pasar datos de contexto a la función de retrollamada.

Otra operación importante que se puede realizar en un caché es la eliminación de entradas. Para ello, se usa la función g_cache_remove.

```
void g_cache_remove(GCache *cache, gconstpointer value);
```

Esta función marca la entrada identificada por *value* para ser borrada. No la borra inmediatamente, sino que decrementa un contador interno asociado a cada entrada, que especifica el número de referencias que dicha entrada tiene. Dichas referencias se asignan a 1 cuando se crea la entrada, y se incrementan cada vez que g_cache_insert recibe una petición de creación de esa misma entrada. Así, cuando dicho contador llega a 0, significa que no hay ninguna referencia a la entrada del caché, y por tanto, puede ser eliminada. Cuando la entrada es eliminada, se realizan llamadas a las funciones *value_destroy_func* para la destrucción de la entrada y *key_destroy_func* para la destrucción de la clave, tal y como se especificó en la explicación de la función g_cache_new.

Destrucción del caché.

Una vez que no sea necesario por más tiempo el caché, hay que destruirlo, como con cualquier otra estructura de GLib, de forma que se libere toda la memoria ocupada. Esto se realiza con la función g_cache_destroy, cuyo único parámetro es el caché que se quiere destruir.

5.2.2 Inserción.

Definición de árbol

Un árbol es una estructura de datos, que puede definirse de forma recursiva como:

- Una estructura vacía o
- Un elemento o clave de información (nodo) más un número finito de estructuras tipo árbol, disjuntos, llamados subárboles. Si dicho número de estructuras es inferior o igual a 2, se tiene un árbol binario.

Es, por tanto, una estructura no secuencial.

Otra definición nos da el árbol como un tipo de grafo (ver grafos): un árbol es un grafo acíclico, conexo y no dirigido. Es decir, es un grafo no dirigido en el que existe exactamente un camino entre todo par de nodos. Esta definición permite implementar un árbol y sus operaciones empleando las representaciones que se utilizan para los grafos. Sin embargo, en esta sección no se tratará esta implementación.

Formas de representación

- Mediante un grafo:

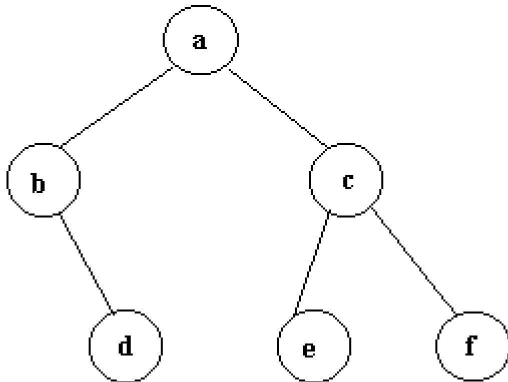


Figura 1

- Mediante un diagrama encolumnado:

a
  b
    d
  c
    e
    f

En la computación se utiliza mucho una estructura de datos, que son los árboles binarios. Estos árboles tienen 0, 1 ó 2 descendientes como máximo. El árbol de la figura anterior es un ejemplo válido de árbol binario.

Nomenclatura sobre árboles

- Raíz: es aquel elemento que no tiene antecesor; ejemplo: *a*.
- Rama: arista entre dos nodos.
- Antecesor: un nodo X es es antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.
- Sucesor: un nodo X es sucesor de un nodo Y si por alguna de las ramas de Y se puede llegar a X.
- Grado de un nodo: el número de descendientes directos que tiene. Ejemplo: *c* tiene grado 2, *d* tiene grado 0, *a* tiene grado 2.
- Hoja: nodo que no tiene descendientes: grado 0. Ejemplo: *d*
- Nodo interno: aquel que tiene al menos un descendiente.
- Nivel: número de ramas que hay que recorrer para llegar de la raíz a un nodo. Ejemplo: el nivel del nodo *a* es 1 (es un convenio), el nivel del nodo *e* es 3.

- Altura: el nivel más alto del árbol. En el ejemplo de la figura 1 la altura es 3.
- Anchura: es el mayor valor del número de nodos que hay en un nivel. En la figura, la anchura es 3.

Aclaraciones: se ha denominado *a* a la raíz, pero se puede observar según la figura que cualquier nodo podría ser considerado raíz, basta con *girar* el árbol. Podría determinarse por ejemplo que *b* fuera la raíz, y *a* y *d* los sucesores inmediatos de la raíz *b*. Sin embargo, en las implementaciones sobre un computador que se realizan a continuación es necesaria una jerarquía, es decir, que haya una única raíz.

Declaración de árbol binario

Se definirá el árbol con una clave de tipo entero (puede ser cualquier otra tipo de datos) y dos hijos: izquierdo (izq) y derecho (der). Para representar los enlaces con los hijos se utilizan punteros. El árbol vacío se representará con un puntero nulo.

Un árbol binario puede declararse de la siguiente manera:

```
typedef struct tarbol
{
  int clave;
  struct tarbol *izq,*der;
} tarbol;
```

Otras declaraciones también añaden un enlace al nodo padre, pero no se estudiarán aquí.

Recorridos sobre árboles binarios

Se consideran dos tipos de recorrido: recorrido en profundidad y recorrido en anchura o a nivel. Puesto que los árboles no son secuenciales como las listas, hay que buscar estrategias alternativas para visitar todos los nodos.

- **Recorridos en profundidad**:

* Recorrido en **preorden**: consiste en visitar el nodo actual (visitar puede ser simplemente mostrar la clave del nodo por pantalla), y después visitar el subárbol izquierdo y una vez visitado, visitar el subárbol derecho. Es un proceso recursivo por naturaleza.
Si se hace el recorrido en preorden del árbol de la figura 1 las visitas serían en el orden siguiente: a,b,d,c,e,f.

```
void preorden(tarbol *a)
{
  if (a != NULL) {
    visitar(a);
    preorden(a->izq);
    preorden(a->der);
  }
}
```

* Recorrido en **inorden** u orden central: se visita el subárbol izquierdo, el nodo actual, y después se visita el subárbol derecho. En el ejemplo de la figura 1 las visitas serían en este orden: b,d,a,e,c,f.

```
void inorden(tarbol *a)
{
  if (a != NULL) {
    inorden(a->izq);
    visitar(a);
    inorden(a->der);
  }
}
```

* Recorrido en **postorden**: se visitan primero el subárbol izquierdo, después el subárbol derecho, y por último el nodo actual. En el ejemplo de la figura 1 el recorrido quedaría así: d,b,e,f,c,a.

```
void postorden(arbol *a)
{
  if (a != NULL) {
    postorden(a->izq);
    postorden(a->der);
    visitar(a);
  }
}
```

La ventaja del recorrido en postorden es que permite borrar el árbol de forma consistente. Es decir, si visitar se traduce por borrar el nodo actual, al ejecutar este recorrido se borrará el árbol o subárbol que se pasa como parámetro. La razón para hacer esto es que no se debe borrar un nodo y después sus subárboles, porque al borrarlo se pueden perder los enlaces, y aunque no se perdieran se rompe con la regla de manipular una estructura de datos inexistente. Una alternativa es utilizar una variable auxiliar, pero es innecesario aplicando este recorrido.

- **Recorrido en amplitud**:

Consiste en ir visitando el árbol por niveles. Primero se visitan los nodos de nivel 1 (como mucho hay uno, la raíz), después los nodos de nivel 2, así hasta que ya no queden más.
Si se hace el recorrido en amplitud del árbol de la figura una visitaría los nodos en este orden: a,b,c,d,e,f
En este caso el recorrido no se realizará de forma recursiva sino iterativa, utilizando una cola (ver Colas) como estructura de datos auxiliar. El procedimiento consiste en encolar (si no están vacíos) los subárboles izquierdo y derecho del nodo extraido de la cola, y seguir desencolando y encolando hasta que la cola esté vacía.
En la codificación que viene a continuación no se implementan las operaciones sobre colas.

```
void amplitud(tarbol *a)
{
  tCola cola;   /* las claves de la cola serán de tipo árbol binario */
  arbol *aux;

  if (a != NULL) {
    CrearCola(cola);
```

```
    encolar(cola, a);
    while (!colavacia(cola)) {
      desencolar(cola, aux);
      visitar(aux);
      if (aux->izq != NULL) encolar(cola, aux->izq);
      if (aux->der != NULL) encolar(cola, aux->der);
    }
  }
}
```

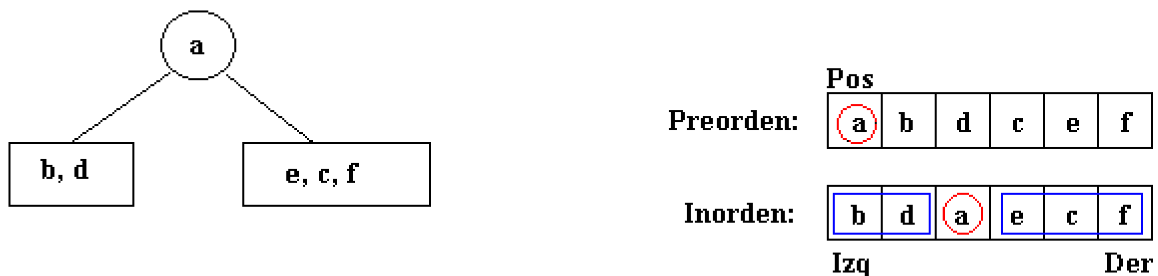
Por último, considérese la sustitución de la cola por una pila en el recorrido en amplitud. ¿Qué tipo de recorrido se obtiene?

Construcción de un árbol binario

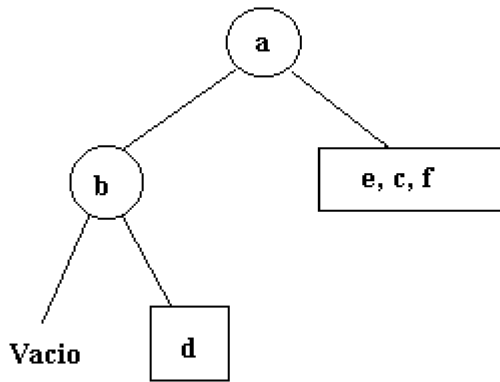
Hasta el momento se ha visto la declaración y recorrido de un árbol binario. Sin embargo no se ha estudiado ningún método para crearlos. A continuación se estudia un método para crear un árbol binario que no tenga claves repetidas partiendo de su recorrido en preorden e inorden, almacenados en sendos arrays.

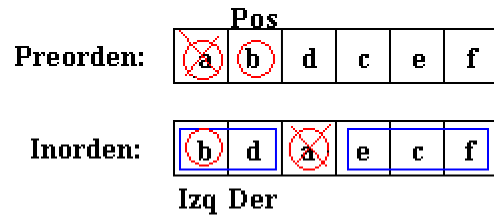
Antes de explicarlo se recomienda al lector que lo intente hacer por su cuenta, es sencillo cuando uno es capaz de construir el árbol viendo sus recorridos pero sin haber visto el árbol terminado.

Partiendo de los recorridos preorden e inorden del árbol de la figura 1 puede determinarse que la raíz es el primer elemento del recorrido en preorden. Ese elemento se busca en el array inorden. Los elementos en el array inorden entre **izq** y la raíz forman el subárbol izquierdo. Asimismo los elementos entre **der** y la raíz forman el subárbol derecho. Por tanto se tiene este árbol:

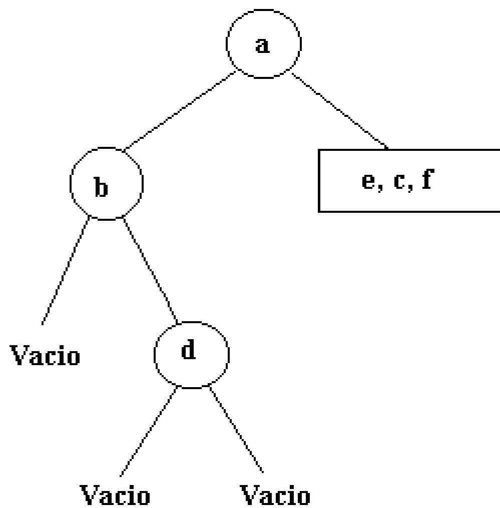


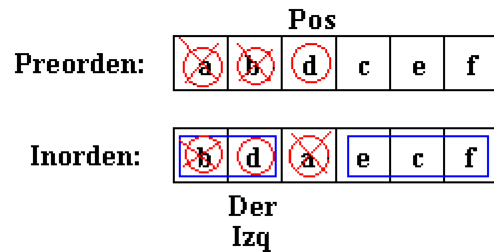
A continuación comienza un proceso recursivo. Se procede a crear el subárbol izquierdo, cuyo tamaño está limitado por los índices **izq** y **der**. La siguiente posición en el recorrido en preorden es la raíz de este subárbol. Queda esto:





El subárbol *b* tiene un subárbol derecho, que no tiene ningún descendiente, tal y como indican los índices **izq** y **der**. Se ha obtenido el subárbol izquierdo completo de la raíz *a*, puesto que *b* no tiene subárbol izquierdo:





Después seguirá construyéndose el subárbol derecho a partir de la raíz *a*.

La implementación de la construcción de un árbol partiendo de los recorridos en preorden y en inorden puede consultarse aquí (en C).

Árbol binario de búsqueda

Un árbol binario de búsqueda es aquel que es:

- Una estructura vacía o
- Un elemento o clave de información (nodo) más un número finito -a lo sumo dos- de estructuras tipo árbol, disjuntos, llamados subárboles y además cumplen lo siguiente:

* Todas las claves del subárbol izquierdo al nodo son menores que la clave del nodo.
* Todas las claves del subárbol derecho al nodo son mayores que la clave del nodo.
* Ambos subárboles son árboles binarios de búsqueda.

Un ejemplo de árbol binario de búsqueda:

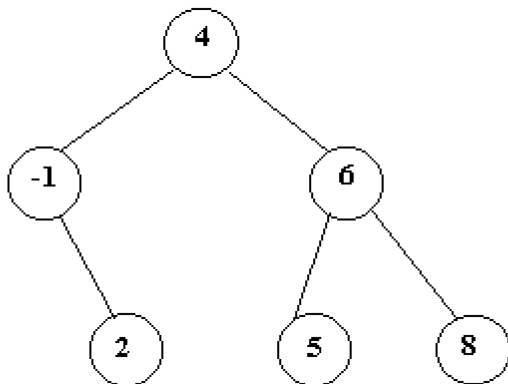


Figura 5

Al definir el tipo de datos que representa la clave de un nodo dentro de un árbol binario de búsqueda es necesario que en dicho tipo se pueda establecer una relación de orden. Por ejemplo, suponer que el tipo de datos de la clave es un puntero (da igual a lo que apunte). Si se codifica el árbol en Pascal no se puede establecer una relación de orden para las claves, puesto que Pascal no admite determinar si un puntero es mayor o menor que otro.

En el ejemplo de la figura 5 las claves son números enteros. Dada la raíz *4*, las claves del subárbol izquierdo son menores que 4, y las claves del subárbol derecho son mayores que 4. Esto se cumple también para todos los subárboles. Si se hace el recorrido de este árbol en orden central se obtiene una lista de los números ordenada de menor a mayor.
Cuestión: ¿Qué hay que hacer para obtener una lista de los números ordenada de mayor a menor?

Una ventaja fundamental de los árboles de búsqueda es que son en general mucho más rápidos para localizar un elemento que una lista enlazada. Por tanto, son más rápidos para insertar y borrar elementos. Si el árbol está **perfectamente equilibrado** -esto es, la diferencia entre el número de nodos del subárbol izquierdo y el número de nodos del subárbol derecho es a lo sumo 1, para todos los nodos- entonces el número de comparaciones necesarias para localizar una clave es aproximadamente de logN en el peor caso. Además, el algoritmo de inserción en un árbol binario de búsqueda tiene la ventaja -sobre los arrays ordenados, donde se emplearía búsqueda dicotómica para localizar un elemento- de que no necesita hacer una reubicación de los elementos de la estructura para que esta siga ordenada después de la inserción. Dicho algoritmo funciona avanzando por el árbol escogiendo la rama izquierda o derecha en función de la clave que se inserta y la clave del nodo actual, hasta encontrar su ubicación; por ejemplo, insertar la clave 7 en el árbol de la figura 5 requiere avanzar por el árbol hasta llegar a la clave 8, e introducir la nueva clave en el subárbol izquierdo a *8*.
El algoritmo de borrado en árboles es algo más complejo, pero más eficiente que el de borrado en un array ordenado.

Ahora bien, suponer que se tiene un árbol vacío, que admite claves de tipo entero. Suponer que se van a ir introduciendo las claves de forma ascendente. Ejemplo: 1,2,3,4,5,6
Se crea un árbol cuya raíz tiene la clave 1. Se inserta la clave 2 en el subárbol derecho de *1*. A continuación se inserta la clave 3 en el subárbol derecho de *2*.
Continuando las inserciones se ve que el árbol degenera en una lista secuencial, reduciendo drásticamente su eficacia para localizar un elemento. De todas formas es poco probable que se de un caso de este tipo en la práctica. Si las claves a introducir llegan de forma más o menos aleatoria entonces la implementación de operaciones sobre un árbol binario de búsqueda que vienen a continuación son en general suficientes.

Existen variaciones sobre estos árboles, como los AVL o Red-Black (no se tratan aquí), que sin llegar a cumplir al 100% el criterio de árbol perfectamente equilibrado, evitan problemas como el de obtener una lista degenerada.

Operaciones básicas sobre árboles binarios de búsqueda

- Búsqueda

Si el árbol no es de búsqueda, es necesario emplear uno de los recorridos anteriores sobre el árbol para localizarlo. El resultado es idéntico al de una búsqueda secuencial. Aprovechando las propiedades del árbol de búsqueda se puede acelerar la localización. Simplemente hay que descender a lo largo del árbol a izquierda o derecha dependiendo del elemento que se busca.

```
boolean buscar(tarbol *a, int elem)
{
  if (a == NULL) return FALSE;
  else if (a->clave < elem) return buscar(a->der, elem);
  else if (a->clave > elem) return buscar(a->izq, elem);
  else return TRUE;
}
```

- Inserción

La inserción tampoco es complicada. Es más, resulta practicamente idéntica a la búsqueda. Cuando se llega a un árbol vacío se crea el nodo en el puntero que se pasa como parámetro por referencia, de esta manera los nuevos enlaces mantienen la coherencia. Si el elemento a insertar ya existe entonces no se hace nada.

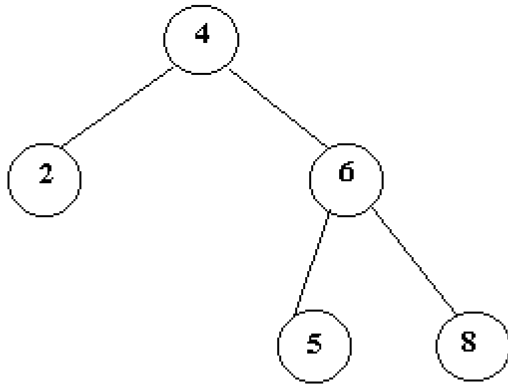
```
void insertar(tarbol **a, int elem)
{
  if (*a == NULL) {
    *a = (arbol *) malloc(sizeof(arbol));
    (*a)->clave = elem;
    (*a)->izq = (*a)->der = NULL;
  }
  else if ((*a)->clave < elem) insertar(&(*a)->der, elem);
  else if ((*a)->clave > elem) insertar(&(*a)->izq, elem);
}
```

- Borrado

La operación de borrado si resulta ser algo más complicada. Se recuerda que el árbol debe seguir siendo de búsqueda tras el borrado. Pueden darse tres casos, una vez encontrado el nodo a borrar:

1) El nodo no tiene descendientes. Simplemente se borra.

2) El nodo tiene al menos un descendiente por una sola rama. Se borra dicho nodo, y su primer descendiente se asigna como hijo del padre del nodo borrado. Ejemplo: en el árbol de la figura 5 se borra el nodo cuya clave es -1. El árbol resultante es:

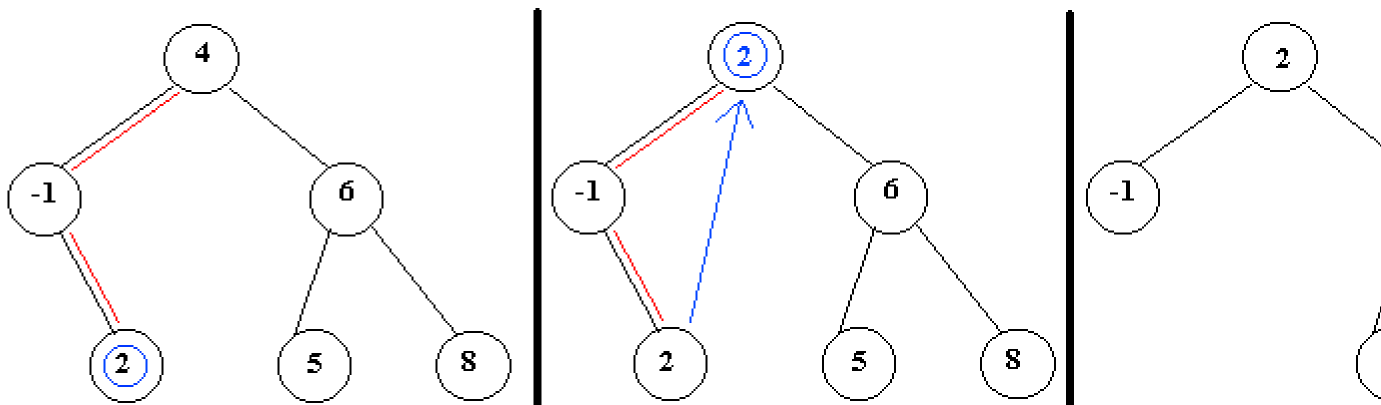


3) El nodo tiene al menos un descendiente por cada rama. Al borrar dicho nodo es necesario mantener la coherencia de los enlaces, además de seguir manteniendo la estructura como un árbol binario de búsqueda. La solución consiste en sustituir la información del nodo que se borra por el de una de las hojas, y borrar a continuación dicha hoja. ¿Puede ser cualquier hoja? No, debe ser la que contenga una de estas dos claves:

· la **mayor** de las claves **menores** al nodo que se borra. Suponer que se quiere borrar el nodo *4* del árbol de la figura 5. Se sustituirá la clave 4 por la clave 2.

· la **menor** de las claves **mayores** al nodo que se borra. Suponer que se quiere borrar el nodo *4* del árbol de la figura 5. Se sustituirá la clave 4 por la clave 5.

El algoritmo de borrado que se implementa a continuación realiza la sustitución por la mayor de las claves menores, (aunque se puede escoger la otra opción sin pérdida de generalidad). Para lograr esto es necesario descender primero a la izquierda del nodo que se va a borrar, y después avanzar siempre a la derecha hasta encontrar un nodo hoja. A continuación se muestra gráficamente el proceso de borrar el nodo de clave 4:



Codificación: el procedimiento sustituir es el que desciende por el árbol cuando se da el caso del nodo con descencientes por ambas ramas.

```
void borrar(tarbol **a, int elem)
{
  void sustituir(tarbol **a, tarbol **aux);
  tarbol *aux;

  if (*a == NULL) /* no existe la clave */
    return;

  if ((*a)->clave < elem) borrar(&(*a)->der, elem);
  else if ((*a)->clave > elem) borrar(&(*a)->izq, elem);
  else if ((*a)->clave == elem) {
    aux = *a;
    if ((*a)->izq == NULL) *a = (*a)->der;
    else if ((*a)->der == NULL) *a = (*a)->izq;
    else sustituir(&(*a)->izq, &aux); /* se sustituye por
      la mayor de las menores */

    free(aux);
  }
}
```

Ficheros relacionados

Implementación de algunas de las operaciones sobre árboles binarios.

Ejercicio resuelto

Escribir una función que devuelva el numero de nodos de un árbol binario. Una solución recursiva puede ser la siguiente:

funcion nodos(arbol : tipoArbol) : devuelve entero;
inicio
 si arbol = vacio entonces devolver 0;
 en otro caso devolver (1 + nodos(subarbol_izq) + nodos(subarbol_der));
fin

Adaptarlo para que detecte si un árbol es perfectamente equilibrado o no.

Problemas propuestos

Árboles binarios: OIE 98. (Enunciado)

Aplicación práctica de un árbol

Se tiene un fichero de texto ASCII. Para este propósito puede servir cualquier libro electrónico de la librería Gutenberg o Cervantes, que suelen tener varios cientos de miles de palabras. El objetivo es clasificar todas las palabras, es decir, determinar que palabras aparecen, y cuantas veces aparece cada una. Palabras como 'niño'-'niña', 'vengo'-'vienes' etc, se consideran diferentes por simplificar el problema.

Escribir un programa, que recibiendo como entrada un texto, realice la clasificación descrita anteriormente.
Ejemplo:
Texto: "a b'a c. hola, adios, hola"

La salida que produce es la siguiente:
a 2
adios 1
b 1
c 1
hola 2

Nótese que el empleo de una lista enlazada ordenada no es una buena solución. Si se obtienen hasta 20.000 palabras diferentes, por decir un número, localizar una palabra cualquiera puede ser, y en general lo será, muy costoso en tiempo. Se puede hacer una implementación por pura curiosidad para evaluar el tiempo de ejecución, pero no merece la pena.

La solución pasa por emplear un árbol binario de búsqueda para insertar las claves. El valor de log(20.000) es aproximadamente de 14. Eso quiere decir que localizar una palabra entre 20.000 llevaría en el peor caso unos 14 accesos. El contraste con el empleo de una lista es simplemente abismal. Por supuesto, como se ha comentado anteriormente el árbol no va a estar perfectamente equilibrado, pero nadie escribe novelas manteniendo el orden lexicográfico (como un diccionario) entre las palabras, asi que no se obtendrá nunca un árbol muy degenerado. Lo que está claro es que cualquier evolución del árbol siempre será mejor que el empleo de una lista.

Por último, una vez realizada la lectura de los datos, sólo queda hacer un recorrido en orden central del árbol y se obtendrá la solución pedida en cuestión de segundos.

Una posible definición de la estructura árbol es la siguiente:

```
typedef struct tarbol
{
  char clave[MAXPALABRA];
  int contador; /* numero de apariciones. Iniciar a 0 */
  struct tarbol *izq,
                *der;
} tarbol;
```

Inserción

Como dijimos en the Section called *Balance del árbol*, implementaremos la inserción de elementos en un árbol AVL de forma análoga a cómo lo haríamos para árboles binarios de búsqueda salvo que en cada recursión del algoritmo verificaremos y corregiremos el equilibrio

del árbol. También es importante ir actualizando las alturas de cada nodo en cada recursión dado que las rotaciones, insercióñes y eliminaciones pueden modificarlas.

Dado que ya vimos funciones tanto para balancear un árbol y para actualizar la altura de un nodo (ambas de tiempo de ejecución constante), estamos listos para implementar el algoritmo de inserción. Esperamos que sea intuitivo.

```
void insertar (AVLTree ** t, int x);
/* inserta x en el árbol en un tiempo O(log(n)) peor caso. */


void
insertar (AVLTree ** t, int x)
{
  if (es_vacio (*t))
    *t = hacer (x, vacio (), vacio ());       /* altura actualizada
                                          automáticamente */
  else
    {
      if (x < raiz (*t))
        insertar (&(*t)->izq, x);

      else
        insertar (&(*t)->der, x);

      balancear (t);
      actualizar_altura (*t);
    }
}
```

5.2.3 Eliminación.

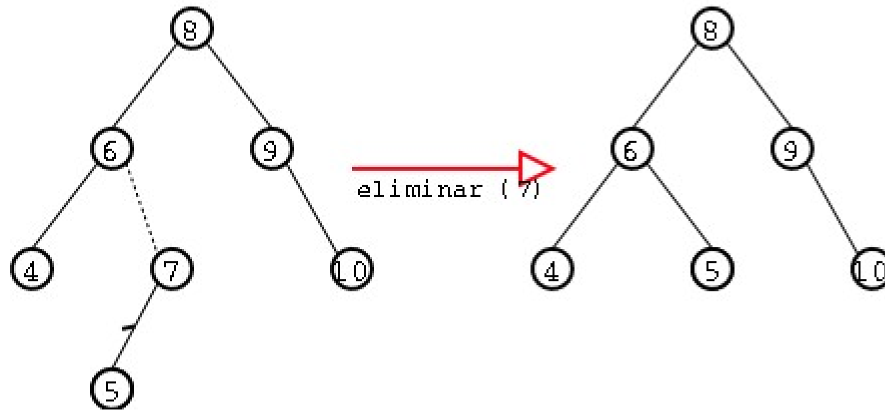
Eliminación

La estrategia para diseñar el algoritmo de eliminación sobre árboles AVL es la misma que para la inserción: Se utiliza el mismo algoritmo que sobre árboles binarios de búsqueda, pero en cada recursión se detectan y corrijen errores por medio de `balancear()` y se actualiza la altura del nodo actual.

Recordamos un poco la idea del algoritmo de eliminación sobre árboles binarios de búsqueda. Primero se recorre el árbol para detectar el nodo a eliminar. Una vez hecho esto hay tres casos a diferenciar por su complejidad:

- Si dicho nodo es una hoja procedemos a eliminarlos de inmediato, sin más.
- Si dicho nodo tiene un sólo hijo, el nodo puede eliminarse después de ajustar un apuntador del padre para saltar el nodo. Esto se muestra en Figure 13.

Figure 13. Eliminación de un nodo (7) con un sólo hijo.



- Si dicho nodo tiene dos hijos el caso es un poco más complicado. Lo que se estila hacer (y que de hecho se hace en el algoritmo gracias a la función auxiliar `eliminar_min()`) reemplazar el nodo actual por el menor nodo de su subárbol derecho (y luego eliminar éste).

```
void eliminar (AVLTree ** t, int x);
/* elimina x del árbol en un tiempo O(log(n)) peor caso.
   Precondición: existe un nodo con valor x en el árbol
   t. */


int eliminar_min (AVLTree ** t);
/* Función auxiliar a eliminar(). Elimina el menor nodo del árbol
   *t devolviendo su contenido (el cual no se libera de
   memoria). Se actualizan las alturas de los nodos.
   Precondición: !es_vacio(*t) */



void
eliminar (AVLTree ** t, int x)
{
  AVLTree *aux;

  if (x < raiz (*t))
    eliminar (&(*t)->izq, x);

  else if (x > raiz (*t))
    eliminar (&(*t)->der, x);

  else              /* coincidencia! */
    {
      if (es_vacio (izquierdo (*t)) && es_vacio (derecho (*t)))
        {/* es una hoja */
          free (*t);
          (*t) = vacio();
        }
      else if (es_vacio (izquierdo (*t)))
        {/* subárbol izquierdo vacio */
          aux = (*t);
          (*t) = (*t)->der;
          free (aux);
        }
```

```
      else if (es_vacio (derecho (*t)))
        {/* subárbol derecho vacio */
          aux = (*t);
          (*t) = (*t)->izq;
          free (aux);
        }
      else     /* caso más complicado */
        {
          (*t)->dato = eliminar_min (&(*t)->der);
        }
    }

  balancear (t);
  actualizar_altura (*t);
}




int
eliminar_min (AVLTree ** t)
{
  if (es_vacio (*t))
    {
      fprintf (stderr,
               "No se respeta precondición de eliminar_min()\n");
      exit(0);
    }
  else
    {
      if (!es_vacio (izquierdo (*t)))
        {
          int x = eliminar_min (&(*t)->izq);
          balancear (t);
          actualizar_altura (*t);
          return x;
        }
      else
        {
          AVLTree *aux = (*t);
          int x = raiz (aux);
          *t = derecho (*t);
          free (aux);
          balancear (t);
          actualizar_altura (*t);
          return x;
        }
    }
}
```

5.2.4 Recorridos sistemáticos.

5.2.5 Balanceo.

Balance del árbol

Como se mostró anteriormente, cda vez que se modifique el árbol (i.e. agreguen o eliminen elementos) corremos el riesgo de que pierda su propiedad de equilibrio en alguno de sus nodos, la cual debe conservarse si queremos obtener tiempos de ejecución de orden O(log(n)) en el peor de los casos.

La idea general que se utiliza en esta implementación de árboles AVL para implementar los algoritmos de inserción y de eliminación de nodos sobre un AVL es la siguiente:

- Efectuar los algoritmos de igual forma que en los árboles binarios de búsqueda pero
- en cada recursión ir actualizando las alturas y rebalanceando el árbol en caso de que sea necesario.

En the Section called *Consideraciones sobre la altura de los nodos* se implementó una función de tiempo de ejecución O(log(n)), peor caso, para actualizar la altura de un nodo. Así, lo que nos falta es una función que detecte un "desequilibrio" en un nodo dado del árbol y por medio de un número finito de rotaciones lo equilibre.

Important: No se demostrará aquí, pero cabe señalar la existencia de un teorema que asegura que el número máximo de rotaciones para equilibrar un árbol AVL luego de una inserción es 2 y luego de una eliminación es log(n) dónde n es el número de nodos.

En las secciones anteriores hemos ya descripto a grandes razgos cuál rotación usar en cada caso de desequilibrio. Esperamos que en el código siguiente el lector pueda formalizar tales ideas.

```
void balancear (AVLTree ** t);
/* Detecta y corrige por medio de un número finito de rotaciones
   un desequilibrio en el árbol *t. Dicho desequilibrio no debe
   tener una diferencia de alturas de más de 2. */


void
balancear (AVLTree ** t)
{
  if(!es_vacio(*t))
  {

    if (altura (izquierdo (*t)) - altura (derecho (*t)) == 2)
      {                /* desequilibrio hacia la izquierda! */

        if (altura ((*t)->izq->izq) >= altura ((*t)->izq->der))
          /* desequilibrio simple hacia la izquierda */
          rotar_s (t, true);
        else
          /* desequilibrio doble hacia la izquierda */
          rotar_d (t, true);
      }
```

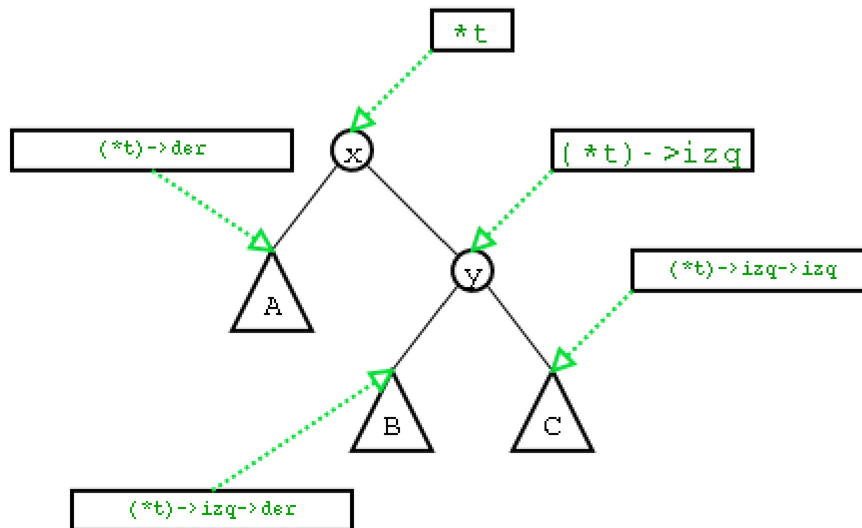
```
    else if (altura (derecho (*t)) - altura (izquierdo (*t)) == 2)
      {                 /* desequilibrio hacia la derecha! */
        if (altura ((*t)->der->der) >= altura ((*t)->der->izq))
          /* desequilibrio simple hacia la izquierda */
          rotar_s (t, false);
        else
          /* desequilibrio doble hacia la izquierda */
          rotar_d (t, false);
      }
  }
}
```

Declaración de la función `balancear()`. Esta declaración junto con el comentario que le sigue deberían estar en un archivo de cabecera usado para la interfaz del tipo abstracto de dato árbol avl con el usuario-programador.

Como dice en el comentario de la función, sólo se contemplarán aquellos desequilibrios cuya diferencia entre alturas es hasta 2.

Sabiendo que en el nodo al que apunta *t hay un desequilibrio hacia la izquierda (de diferencia de alturas 2), debemos averiguar qué clase de rotación aplicar. En Figure 12 se explica gráficamente a dónde apuntan las variables de la función en un árbol genérico.

Figure 12. Decidiendo qué clase de rotación aplicar para solucionar desequilibrio en el nodo.



Como puede verse en el código, nos decidimos por una rotación simple izquierda si el subárbol más pesado de `(*t)->izq` es el izquierdo o por una rotación doble izquierda si el subárbol más pesado de `(*t)->izq` es el derecho.

Si detectamos un desequilibrio hacia la derecha, la toma de deciciones son análogas a las de un desequilibrio hacia la izquierda, las cuales ya explicamos.

Unidad 6 Ordenación interna.

Cuestiones generales

Su finalidad es organizar ciertos datos (normalmente arrays o ficheros) en un orden creciente o decreciente mediante una regla prefijada (numérica, alfabética...). Atendiendo al tipo de elemento que se quiera ordenar puede ser:

- Ordenación interna: Los datos se encuentran en memoria (ya sean arrays, listas, etc) y son de acceso aleatorio o directo (se puede acceder a un determinado campo sin pasar por los anteriores).

- Ordenación externa: Los datos están en un dispositivo de almacenamiento externo (ficheros) y su ordenación es más lenta que la interna.

Ordenación interna

Los métodos de ordenación interna se aplican principalmente a arrays unidimensionales. Los principales algoritmos de ordenación interna son:

Selección: Este método consiste en buscar el elemento más pequeño del array y ponerlo en primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segudo lugar, y así sucesivamente hasta colocar el último elemento. Por ejemplo, si tenemos el array {40,21,4,9,10,35}, los pasos a seguir son:

```
{4,21,40,9,10,35} <-- Se coloca el 4, el más pequeño, en primera posición : se
cambia el 4 por el 40.
{4,9,40,21,10,35} <-- Se coloca el 9, en segunda posición: se cambia el 9 por
el 21.
{4,9,10,21,40,35} <-- Se coloca el 10, en tercera posición: se cambia el 10
por el 40.
{4,9,10,21,40,35} <-- Se coloca el 21, en tercera posición: ya está colocado.
{4,9,10,21,35,40} <-- Se coloca el 35, en tercera posición: se cambia el 35
por el 40.
```

Si el array tiene N elementos, el número de comprobaciones que hay que hacer es de N*(N-1)/2, luego el tiempo de ejecución está en $O(n^2)$

```
var
   lista: array[1..N] of byte;
   aux, i, j, menor: byte;

BEGIN
 for i:= 1 to N-1 do
  begin
   menor:= i;
```

```
   for j:= i+1 to N do
    if (lista [j+1]<lista[j]) then
     begin
      if (lista[j]<lista[menor]) then  {si el elemento j es menor que el
menor}
       menor:=j;                       {pasa a ser el menor}
      aux:= lista[i];                  {se intercambian los elemento}
      lista[i]:=lista[menor];          {de las posiciones i y menor}
      lista[menor]:=aux;               {usando la variable auxiliar}
     end;
  end;
END.
```

Burbuja: Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, {40,21,4,9,10,35}:

```
Primera pasada:
{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.
{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

Segunda pasada:
{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.
{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.
{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.
```

Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

Si el array tiene N elementos, para estar seguro de que el array está ordenado, hay que hacer N-1 pasadas, por lo que habría que hacer (N-1)*(N-i-1) comparaciones, para cada i desde 1 hasta N-1. El número de comparaciones es, por tanto, N(N-1)/2, lo que nos deja un tiempo de ejecución, al igual que en la selección, en $O(n^2)$.

```
var
   lista: array[1..N] of byte;
   aux, i, j: byte;

BEGIN

{Dar valores a los elementos del array}

 for i:= 1 to N do
  for j:= 1 to N-i do
   if (lista [j+1]<lista[j]) then
    begin
     aux:= lista[j+1];
     lista[j+1]:=lista[j];
     lista[j]:=aux;
    end;
END.
```

Inserción directa: En este método lo que se hace es tener una sublista ordenada de elementos del array e ir insertando el resto en el lugar adecuado para que la sublista no

pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada. Para el ejemplo {40,21,4,9,10,35}, se tiene:

```
{40,21,4,9,10,35} <-- La primera sublista ordenada es {40}.

Insertamos el 21:
{40,40,4,9,10,35} <-- aux=21;
{21,40,4,9,10,35} <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:
{21,40,40,9,10,35} <-- aux=4;
{21,21,40,9,10,35} <-- aux=4;
{4,21,40,9,10,35} <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:
{4,21,40,40,10,35} <-- aux=9;
{4,21,21,40,10,35} <-- aux=9;
{4,9,21,40,10,35} <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:
{4,9,21,40,40,35} <-- aux=10;
{4,9,21,21,40,35} <-- aux=10;
{4,9,10,21,40,35} <-- Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:
{4,9,10,21,40,40} <-- aux=35;
{4,9,10,21,35,40} <-- El array está ordenado.
```

En el peor de los casos, el número de comparaciones que hay que realizar es de N*(N+1)/2-1, lo que nos deja un tiempo de ejecución en $O(n^2)$. En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(n).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Vemos que cuanto más ordenada esté inicialmente más se acerca a O(n) y cuanto más desordenada, más se acerca a $O(n^2)$.

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

```
var
 lista: array[1..N] of byte;
 i, j, aux: byte;
 encontrado: boolean;

BEGIN

{Dar valores a los elementos del array}

 for i:= 2 to N do
  begin
   aux:= lista[i];                     {se intenta añadir el elemento i}
   encontrado:=false;
   j:=i-1;
   while not(encontrado) and (j>=1) do {se recorre la sublista de atras a
adelante
```

```
                                          para buscar la nueva posicion del
elemento i}
    begin
     if (aux>lista[j]) then               {si se encuentra la posición}
      begin
       lista[j+1]:=aux;                   {se coloca}
       encontrado:=true;                  {y se sale del bucle para colocar el
siguiente número}
      end
     else                                 {si no, se sigue buscando}
      begin
       lista[j+1]:=lista[j];
       dec(j);
      end;
    end;
   if (j=0) then lista[1]:= aux;          {si se han mirado todas las posiciones
y no se ha
                                          encontrado, es que es la primera
posicion}
  end;
END.
```

Inserción binaria: Es el mismo método que la inserción directa, excepto que la búsqueda del orden de un elemento en la sublista ordenada se realiza mediante una búsqueda binaria (ver algoritmos de búsqueda), lo que en principio supone un ahorro de tiempo. No obstante, dado que para la inserción sigue siendo necesario un desplazamiento de los elementos, el ahorro, en la mayoría de los casos, no se produce, si bien hay compiladores que realizan optimizaciones que lo hacen ligeramente más rápido.

Shell: Es una mejora del método de inserción directa, utilizado cuando el array tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (llamado salto) a su izquierda. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, lo pasos para ordenar el array {40,21,4,9,10,35} mediante el método de Shell serían:

```
Salto=3:
   Primera pasada:
   {9,21,4,40,10,35} <-- se intercambian el 40 y el 9.
   {9,10,4,40,21,35} <-- se intercambian el 21 y el 10.
Salto=1:
   Primera pasada:
   {9,4,10,40,21,35} <-- se intercambian el 10 y el 4.
   {9,4,10,21,40,35} <-- se intercambian el 40 y el 21.
   {9,4,10,21,35,40} <-- se intercambian el 35 y el 40.
   Segunda pasada:
   {4,9,10,21,35,40} <-- se intercambian el 4 y el 9.
```

Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

```
var
```

```
 lista: array[1..N] of byte;
 aux, i, salto, cambios: byte;


BEGIN
 salto:= N div 2;
 while not(salto=0) do                  {el salto va desde N/2 hasta 1}
  begin
   cambios:=1;
   while not(cambios=0) do              {mientras se intercambien elementos}
    begin
     cambios:=0;
     for i:= (salto+1) to N do          {se pasa una vez}
      begin
       if (lista[i-salto]>lista[i]) then {y se reordenan si no lo estan}
        begin
         aux:=lista[i];
         lista[i]:=lista[i-salto];
         lista[i-salto]:=aux;
         inc(cambios);
        end;
      end;
    end;
   salto:= salto div 2;
  end;
END.
```

Ordenación rápida (quicksort): Este método se basa en la táctica "divide y vencerás" (ver sección divide y vencerás), que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Normalmente se toma como pivote el primer elemento de array, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.Por ejemplo, para dividir el array {21,40,4,9,10,35}, los pasos serían:

```
{21,40,4,9,10,35} <-- se toma como pivote el 21. La búsqueda de izquierda a
derecha encuantra el valor 40, mayor que pivote, y la búsqueda de derecha a
izquierda encuentra el valor 10, menor que el pivote. Se intercambian:
{21,10,4,9,40,35} <-- Si seguimos la búsqueda, la primera encuentra el valor
40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para
terminar la división, se coloca el pivote en su lugar (en el número encontrado
por la segunda búsqueda, el 9, quedando:
{9,10,4,21,40,35} <-- Ahora tenemos dividido el array en dos arrays más
pequeños: el {9,10,4} y el {40,35}, y se repetiría el mismo proceso.
```

Intercalación: no es propiamente un método de ordenación, consiste en la unión de dos arrays ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arrays de izquierda a derecha e ir cogiendo el menor de los dos elementos, de forma que sólo aumenta el contador del array del que sale el elemento siguiente para el array-suma. Si quisiéramos sumar los arrays {1,2,4} y {3,5,6}, los pasos serían:

```
Inicialmente: i1=0, i2=0, is=0.
Primer elemento: mínimo entre 1 y 3 = 1. Suma={1}. i1=1, i2=0, is=1.
Segundo elemento: mínimo entre 2 y 3 = 2. Suma={1,2}. i1=2, i2=0, is=2.
Tercer elemento: mínimo entre 4 y 3 = 3. Suma={1,2,3}. i1=2, i2=1, is=3.
Cuarto elemento: mínimo entre 4 y 5 = 4. Suma={1,2,3,4}. i1=3, i2=1, is=4.
Como no quedan elementos del primer array, basta con poner los elementos que
quedan del segundo array en la suma:
Suma={1,2,3,4}+{5,6}={1,2,3,4,5,6}

var
 i1,i2,is: byte;
 lista1: array[1..N1] of byte;
 lista2: array[1..N2] of byte;
 suma: array[1..N1+N2] of byte;

BEGIN
 i1:=1;
 i2:=1;
 is:=1;
 while (i1<=N1) and (i2<=N2) do
  begin
   if (lista1[i1]<lista2[i2]) then
    begin
     suma[is]:=lista1[i1];
     inc(i1);
    end
   else
    begin
     suma[is]:=lista2[i2];
     inc(i2);
    end;
   inc(is);
  end;

 if (i1<N1) then          {se termina de añadir el array que no haya llegado
al final}
  for i:=i1 to N1 do
   begin
    suma[is]:=lista1[i];
    inc(is);
   end
  else
   begin
    for i:=i2 to N2 do
     begin
      suma[is]:=lista2[i];
      inc(is);
     end;
   end;
END.
```

Fusión: Consta de dos partes, una parte de intercalación de listas y otra de divide y vencerás.

- Primera parte: ¿Cómo intercalar dos listas <u>ordenadas</u> en una sola lista ordenada de forma eficiente?

Suponemos que se tienen estas dos listas de enteros ordenadas ascendentemente:

lista *1*: 1 -> 3 -> 5 -> 6 -> 8 -> 9
lista *2*: 0 -> 2 -> 6 -> 7 -> 10

Tras mezclarlas queda:

lista: 0 -> 1 -> 2 -> 3 -> 5 -> 6 -> 6 -> 7 -> 8 -> 9 -> 10

Esto se puede realizar mediante un único recorrido de cada lista, mediante dos punteros que recorren cada una. En el ejemplo anterior se insertan en este orden -salvo los dos 6 que puede variar según la implementación-: 0 (lista 2), el 1 (lista 1), el 2 (lista 2), el 3, 5 y 6 (lista 1), el 6 y 7 (lista 2), el 8 y 9 (lista 1), y por llegar al final de la lista 1, se introduce directamente todo lo que quede de la lista 2, que es el 10.

En la siguiente implementación no se crea una nueva lista realmente, sólo se *modifican* los enlaces destruyendo las dos listas y fusionándolas en una sola. Se emplea un centinela que apunta a sí mismo y que contiene como clave el valor más grande posible. El último elemento de cada lista apuntará al centinela, incluso si la lista está vacía.

```
struct lista
{
  int clave;
  struct lista *sig;
};


struct lista *centinela;
centinela = (struct lista *) malloc(sizeof(struct lista));
centinela->sig = centinela;
centinela->clave = INT_MAX;
...

struct lista *fusion(struct lista *l1, struct lista *l2)
{
  struct lista *inic, *c;

  if (l1->clave < l2->clave) { inic = l1; l1 = l1->sig; }
  else { inic = l2; l2 = l2->sig; }
  c = inic;
  while (l1 != centinela && l2 != centinela) {
    if (l1->clave < l2->clave) {
      c->sig = l1; l1 = l1->sig;
    }
    else {
      c->sig = l2; l2 = l2->sig;
    }
    c = c->sig;
  }
  if (l1 != centinela) c->sig = l1;
  else if (l2 != centinela) c->sig = l2;
  return inic;
}
```

- Segunda parte: divide y vencerás. Se separa la lista original en dos trozos del mismo tamaño (salvo listas de longitud impar) que se ordenan recursivamente, y una vez ordenados se fusionan obteniendo una lista ordenada. Como todo algoritmo basado en divide y vencerás tiene un caso base y un caso recursivo.

* *Caso base*: cuando la lista tiene 1 ó 0 elementos (0 se da si se trata de ordenar una lista vacía). Se devuelve la lista tal cual está.

* *Caso recursivo*: cuando la longitud de la lista es de al menos 2 elementos. Se **divide** la lista en dos trozos del mismo tamaño que se ordenan recursivamente. Una vez ordenado cada trozo, se fusionan y se devuelve la lista resultante.

El esquema es el siguiente:

```
Ordenar(lista L)
inicio
  si tamaño de L es 1 o 0 entonces
    devolver L
  si tamaño de L es >= 2 entonces
    separar L en dos trozos: L1 y L2.
    L1 = Ordenar(L1)
    L2 = Ordenar(L2)
    L = Fusionar(L1, L2)
    devolver L
fin
```

El algoritmo funciona y termina porque llega un momento en el que se obtienen listas de 2 ó 3 elementos que se dividen en dos listas de un elemento (1+1=2) y en dos listas de uno y dos elementos (1+2=3, la lista de 2 elementos se volverá a dividir), respectivamente. Por tanto se vuelve siempre de la recursión con listas ordenadas (pues tienen a lo sumo un elemento) que hacen que el algoritmo de fusión reciba siempre listas ordenadas.

Se incluye un ejemplo explicativo donde cada sublista lleva una etiqueta identificativa.

Dada: 3 -> 2 -> 1 -> 6 -> 9 -> 0 -> 7 -> 4 -> 3 -> 8 (lista original)

se **divide** en:
3 -> 2 -> 1 -> 6 -> 9 (lista 1)
0 -> 7 -> 4 -> 3 -> 8 (lista 2)
· se ordena recursivamente cada lista:
· 3 -> 2 -> 1 -> 6 -> 9 (lista 1)
· · se **divide** en:
· · 3 -> 2 -> 1 (lista 11)
· · 6 -> 9 (lista 12)
· · se ordena recursivamente cada lista:
· · 3 -> 2 -> 1 (lista 11)
· · · se **divide** en:
· · · 3 -> 2 (lista 111)
· · · 1 (lista 112)
· · · se ordena recursivamente cada lista:
· · · 3 -> 2 (lista 111)
· · · se **divide** en:
· · · 3 (lista 1111, que no se divide, *caso base*). Se devuelve 3
· · · 2 (lista 1112, que no se divide, *caso base*). Se devuelve 2
· · · se fusionan 1111-1112 y queda:
· · · 2 -> 3. Se devuelve 2 -> 3
· · · 1 (lista 112)
· · · 1 (lista 1121, que no se divide, *caso base*). Se devuelve 1
· · · se fusionan 111-112 y queda:

· · · 1 -> 2 -> 3 (lista 11). Se devuelve 1 -> 2 -> 3
· · 6 -> 9 (lista 12)
· · · se **divide** en:
· · · 6 (lista 121, que no se divide, *caso base*). Se devuelve 6
· · · 9 (lista 122, que no se divide, *caso base*). Se devuelve 9
· · · se fusionan 121-122 y queda:
· · · 6 -> 9 (lista 12). Se devuelve 6 -> 9
· · se fusionan 11-12 y queda:
· · 1 -> 2 -> 3 -> 6 -> 9. Se devuelve 1 -> 2 -> 3 -> 6 -> 9
· 0 -> 7 -> 4 -> 3 -> 8 (lista 2)
· ... tras repetir el mismo procedimiento se devuelve 0 -> 3 -> 4 -> 7 -> 8
· se fusionan 1-2 y queda:
· 0 -> 1 -> 2 -> 3 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9, que se devuelve y se termina.

La implementación propuesta emplea un centinela sobre la lista inicial que apunte hacia sí mismo y que además contiene el máximo valor de un entero. La lista dispone de cabecera y centinela, pero obsérvese como se elimina durante la ordenación.

```
#include <stdlib.h>
#include <limits.h>

struct lista
{
  int clave;
  struct lista *sig;
};
/* lista con cabecera y centinela */
struct listacc
{
  struct lista *cabecera,
               *centinela;
};

/* centinela declarado como variable global */
struct lista *centinela;

  /* fusiona dos listas */
struct lista *fusion(struct lista *l1, struct lista *l2)
{
  struct lista *inic, *c;

  if (l1->clave < l2->clave) { inic = l1; l1 = l1->sig; }
  else { inic = l2; l2 = l2->sig; }
  c = inic;
  while (l1 != centinela && l2 != centinela) {
    if (l1->clave < l2->clave) {
      c->sig = l1; l1 = l1->sig;
    }
    else {
      c->sig = l2; l2 = l2->sig;
    }
    c = c->sig;
  }
  if (l1 != centinela) c->sig = l1;
  else if (l2 != centinela) c->sig = l2;
```

```
  return inic;
}

  /* algoritmo de ordenación por fusión mediante divide y vencerás */
struct lista *ordenfusion(struct lista *l)
{
  struct lista *l1, *l2, *parte1, *parte2;

    /* caso base: 1 ó 0 elementos */
  if (l->sig == centinela) return l;


    /* caso recursivo */
  /* avanza hasta la mitad de la lista */
  l1 = l; l2 = l1->sig->sig;
  while (l2 != centinela) {
    l1 = l1->sig;
    l2 = l2->sig->sig;
  }
  /* la parte en dos */
  l2 = l1->sig;
  l1->sig = centinela;

  /* ordena recursivamente cada parte */
  parte1 = ordenfusion(l);
  parte2 = ordenfusion(l2);

  /* mezcla y devuelve la lista mezclada */
  l = fusion(parte1, parte2);
  return l;
}


  /* operaciones de lista */
void crearLCC(struct listacc *LCC)
{
  LCC->cabecera = (struct lista *) malloc(sizeof(struct lista));
  LCC->centinela = (struct lista *) malloc(sizeof(struct lista));
  LCC->cabecera->sig = LCC->centinela;
  LCC->centinela->sig = LCC->centinela;
}

  /* inserta un elemento al comienzo de la lista */
void insertarPrimero(struct listacc LCC, int elem)
{
  struct lista *nuevo;

  nuevo = (struct lista *) malloc(sizeof(struct lista));
  nuevo->clave = elem;
  nuevo->sig = LCC.cabecera->sig;
  LCC.cabecera->sig = nuevo;
}

int main(void)
{
  struct listacc LCC;
  crearLCC(&LCC);
  centinela = LCC.centinela;
  centinela->clave = INT_MAX;
```

```
  insertarPrimero(LCC, 8);
  insertarPrimero(LCC, 3);
  insertarPrimero(LCC, 4);
  insertarPrimero(LCC, 7);
  insertarPrimero(LCC, 0);
  insertarPrimero(LCC, 9);
  insertarPrimero(LCC, 6);
  insertarPrimero(LCC, 1);
  insertarPrimero(LCC, 2);
  insertarPrimero(LCC, 3);
  LCC.cabecera = ordenfusion(LCC.cabecera->sig);

  return 0;
}
```

Este es un buen algoritmo de ordenación, pues no requiere espacio para una nueva lista y sólo las operaciones recursivas consumen algo de memoria. Es por tanto un **algoritmo ideal** para ordenar listas.

La complejidad es la misma en todos los casos, ya que no influye cómo esté ordenada la lista inicial -esto es, no existe ni mejor ni peor caso-, puesto que la intercalación de dos listas ordenadas siempre se realiza de una única pasada. La complejidad es proporcional a N·logN, característica de los algoritmos "Divide y Vencerás". Para hacer más eficiente el algoritmo es mejor realizar un primer recorrido sobre toda la lista para contar el número de elementos y añadir como parámetro a la función dicho número.

6.1 Algoritmos de Ordenamiento por Intercambio.

Algoritmos de Ordenamiento

¿Qué es ordenamiento?

Es la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento.

El ordenamiento se efectúa con base en el valor de algún campo en un registro.

El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado.

Ej. de ordenamientos:

Dir. telefónico, tablas de contenido, bibliotecas y diccionarios, etc.

El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.

¿Cuándo conviene usar un método de ordenamiento?

Cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.

Tipos de ordenamientos:

Los 2 tipos de ordenamientos que se pueden realizar son: los internos y los externos.

Los internos:

Son aquellos en los que los valores a ordenar están en memoria principal, por lo que se asume que el tiempo que se requiere para acceder cualquier elemento sea el mismo (a[1], a[500], etc).

Los externos:

Son aquellos en los que los valores a ordenar están en memoria secundaria (disco, cinta, cilindro magnético, etc), por lo que se asume que el tiempo que se requiere para acceder a cualquier elemento depende de la última posición accesada (posición 1, posición 500, etc).

Eficiencia en tiempo de ejecución:

Una medida de eficiencia es:

Contar el # de comparaciones (C)

Contar el # de movimientos de items (M)

Estos están en función de el #(n) de items a ser ordenados.

Un "buen algoritmo" de ordenamiento requiere de un orden *nlogn* comparaciones.

La eficiencia de los algoritmos se mide por el número de comparaciones e intercambios que tienen que hacer, es decir, se toma n como el número de elementos que tiene el arreglo o vector a ordenar y se dice que un algoritmo realiza O(n2) comparaciones cuando compara n veces los n elementos, n x n = n2

Algoritmos de ordenamiento:

Internos:

1. Inserción directa.
    1. Inserción directa.
    2. Inserción binaria.
2. Selección directa.
    1. Selección directa.
3. Intercambio directo.
    1. Burbuja.
    2. Shake.
4. Inserción disminución incremental.
    1. Shell.
5. Ordenamiento de árbol.
    1. Heap.
    2. Tournament.
6. Sort particionado.
    1. Quick sort.
7. Merge sort.
8. Radix sort.
9. Cálculo de dirección.

Externos:

1. Straight merging.
2. Natural merging.
3. Balanced multiway merging.
4. Polyphase sort.
5. Distribution of initial runs.

Clasificación de los algoritmos de ordenamiento de información:

El hecho de que la información está ordenada, nos sirve para poder encontrarla y accesarla de manera más eficiente ya que de lo contrario se tendría que hacer de manera secuencial.

A continuación se describirán 4 grupos de algoritmos para ordenar información:

Algoritmos de inserción:

En este tipo de algoritmo los elementos que van a ser ordenados son considerados uno a la vez. Cada elemento es INSERTADO en la posición apropiada con respecto al resto de los elementos ya ordenados.

Entre estos algoritmos se encuentran el de INSERCION DIRECTA, SHELL SORT, INSERCION BINARIA y HASHING.

Algoritmos de intercambio:

En este tipo de algoritmos se toman los elementos de dos en dos, se comparan y se INTERCAMBIAN si no están en el orden adecuado. Este proceso se repite hasta que se ha analizado todo el conjunto de elementos y ya no hay intercambios.

Entre estos algoritmos se encuentran el BURBUJA y QUICK SORT.

Algoritmos de selección:

En este tipo de algoritmos se SELECCIONA o se busca el elemento más pequeño (o más grande) de todo el conjunto de elementos y se coloca en su posición adecuada. Este proceso se repite para el resto de los elementos hasta que todos son analizados. Entre estos algoritmos se encuentra el de SELECCION DIRECTA.

Algoritmos de enumeración:

En este tipo de algoritmos cada elemento es comparado contra los demás. En la comparación se cuenta cuántos elementos son más pequeños que el elemento que se está analizando, generando así una ENUMERACION. El número generado para cada elemento indicará su posición.

Los métodos simples son: Inserción (o por inserción directa), selección, burbuja y shell, en dónde el último es una extensión al método de inserción, siendo más rápido. Los métodos más complejos son el quick-sort (ordenación rápida) y el heap sort.

A continuación se mostrarán los métodos de ordenamiento más simples.

METODO DE INSERCIÓN.

Este método toma cada elemento del arreglo para ser ordenado y lo compara con los que se encuentran en posiciones anteriores a la de él dentro del arreglo. Si resulta que el elemento con el que se está comparando es mayor que el elemento a ordenar, se recorre hacia la siguiente posición superior. Si por el contrario, resulta que el elemento con el que se está comparando es menor que el elemento a ordenar, se detiene el proceso de comparación pues se encontró que el elemento ya está ordenado y se coloca en su posición (que es la siguiente a la del último número con el que se comparó).

Procedimiento ***Insertion Sort***

Este procedimiento recibe el arreglo de datos a ordenar **a[]** y altera las posiciones de sus elementos hasta dejarlos ordenados de menor a mayor. **N** representa el número de elementos que contiene a[].

paso 1: [Para cada pos. del arreglo] For i <- 2 to N do

paso 2: [Inicializa v y j] v <- a[i]

j <- i.

paso 3: [Compara v con los anteriores] While a[j-1] > v AND j>1 do

paso 4: [Recorre los datos mayores] Set a[j] <- a[j-1],

paso 5: [Decrementa j] set j <- j-1.

paso 5: [Inserta v en su posición] Set a[j] <- v.

paso 6: [Fin] End.

Ejemplo:

Si el arreglo a ordenar es a = ['a','s','o','r','t','i','n','g','e','x','a','m','p','l','e'], el algoritmo va a recorrer el arreglo de izquierda a derecha. Primero toma el segundo dato 's' y lo asigna a *v* y *i* toma el valor de la posición actual de *v*.

Luego compara esta 's' con lo que hay en la posición j-1, es decir, con 'a'. Debido a que 's' no es menor que 'a' no sucede nada y avanza i.

Ahora v toma el valor 'o' y lo compara con 's', como es menor recorre a la 's' a la posición de la 'o'; decrementa j, la cual ahora tiene la posición en dónde estaba la 's'; compara a 'o' con a[j-1] , es decir, con 'a'. Como no es menor que la 'a' sale del for y pone la 'o' en la posición a[j]. El resultado hasta este punto es el arreglo siguiente: a = ['a','o','s','r',....]

Así se continúa y el resultado final es el arreglo ordenado :

a = ['a','a','e','e','g','i','l','m','n','o','p','r','s','t','x']

MÉTODO DE SELECCIÓN.

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenar todo el arreglo.

Procedimiento ***Selection Sort***

paso 1: [Para cada pos. del arreglo] For i <- 1 to N do

paso 2: [Inicializa la pos. del menor] menor <- i

paso 3: [Recorre todo el arreglo] For j <- i+1 to N do

paso 4: [Si a[j] es menor] If a[j] < a[menor] then

paso 5: [Reasigna el apuntador al menor] min = j

paso 6: [Intercambia los datos de la pos.

min y posición i] Swap(a, min, j).

paso 7: [Fin] End.

Ejemplo:

El arreglo a ordenar es a = ['a','s','o','r','t','i','n','g','e','x','a','m','p','l','e'].

Se empieza por recorrer el arreglo hasta encontrar el menor elemento. En este caso el menor elemento es la primera 'a'. De manera que no ocurre ningún cambio. Luego se procede a buscar el siguiente elemento y se encuentra la segunda 'a'.

Esta se intercambia con el dato que está en la segunda posición, la 's', quedando el arreglo así después de dos recorridos: a = ['a','a','o','r','t','i','n','g','e','x','s','m','p','l','e'].

El siguiente elemento, el tercero en orden de menor mayor es la primera 'e', la cual se intercambia con lo que está en la tercera posición, o sea, la 'o'. Le sigue la segunda 's', la cual es intercambiada con la 'r'.

El arreglo ahora se ve de la siguiente manera: a = ['a','a','e','e','t','i','n','g','o','x','s','m','p','l','r'].

De esta manera se va buscando el elemento que debe ir en la siguiente posición hasta ordenar todo el arreglo.

El número de comparaciones que realiza este algoritmo es :

Para el primer elemento se comparan n-1 datos, en general para el elemento i-ésimo se hacen n-i comparaciones, por lo tanto, el total de comparaciones es:

la sumatoria para i de 1 a n-1 (n-i) = 1/2 n (n-1).

MÉTODO BURBUJA.

El bubble sort, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Se recorre el arreglo tantas veces hasta que ya no haya cambios. Prácticamente lo que hace es tomar el elemento mayor y lo va recorriendo de posición en posición hasta ponerlo en su lugar.

Procedimiento ***Bubble Sort***

paso 1: [Inicializa i al final de arreglo] For i <- N down to 1 do

paso 2: [Inicia desde la segunda pos.] For j <- 2 to i do

paso 4: [Si a[j-1] es mayor que el que le sigue] If a[j-1] < a[j] then

paso 5: [Los intercambia] Swap(a, j-1, j).

paso 7: [Fin] End.

Tiempo de ejecución del algoritmo burbuja:

1. Para el mejor caso (un paso) O(n)
2. Peor caso n(n-1)/2
3. Promedio $O(n^2)$

MÉTODO DE SHELL.

Ordenamiento de disminución incremental.

Nombrado así debido a su inventor Donald Shell.

Ordena subgrupos de elementos separados K unidades (respecto de su posición en el arreglo) del arreglo original. El valor K es llamado incremento.

Después de que los primeros K subgrupos han sido ordenados (generalmente utilizando INSERCION DIRECTA), se escoge un nuevo valor de K más pequeño, y el arreglo es de nuevo partido entre el nuevo conjunto de subgrupos. Cada uno de los subgrupos mayores es ordenado y el proceso se repite de nuevo con un valor más pequeño de K.

Eventualmente el valor de K llega a ser 1, de tal manera que el subgrupo consiste de todo el arreglo ya casi ordenado.

Al principio del proceso se escoge la secuencia de decrecimiento de incrementos; el último valor debe ser 1.

"Es como hacer un ordenamiento de burbuja pero comparando e intercambiando elementos."

Cuando el incremento toma un valor de 1, todos los elementos pasan a formar parte del subgrupo y se aplica inserción directa.

El método se basa en tomar como salto N/2 (siendo N el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia vale 1.

Procedimiento **Shell Sort**;

```
const

MAXINC = _____;

incrementos = array[1..MAXINC] of integer;

var

j,p,num,incre,k:integer;

begin

for incre := 1 to MAXINC do begin /* para cada uno de los incrementos */

k := inc[incre]; /* k recibe un tipo de incremento */

for p := k+1 to MAXREG do begin /* inserción directa para el grupo que se encuentra
cada K posiciones */

num := reg[p];

j := p-k;

while (j>0) AND (num < reg[j]) begin

reg[j+k] := reg[j];

j := j - k;

end;

reg[j+k] := num;

end

end

end;
```

Ejemplo:

Para el arreglo a = [6, 1, 5, 2, 3, 4, 0]

Tenemos el siguiente recorrido:

Recorrido	Salto	Lista Ordenada	Intercambio
1	3	2,1,4,0,3,5,6	(6,2), (5,4), (6,0)
2	3	0,1,4,2,3,5,6	(2,0)
3	3	0,1,4,2,3,5,6	Ninguno
4	1	0,1,2,3,4,5,6	(4,2), (4,3)
5	1	0,1,2,3,4,5,6	Ninguno

6.1.1 Burbuja.

Ordenamiento de burbuja

De Wikipedia

Saltar a navegación, búsqueda

El **Bubble sort** es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas".

A continuación se muestra el pseudocódigo, donde `Vector(posicion)` es una función que devuelve el elemento situdado en una determinada posición de la lista, y `NO_ORDENADOS(a, b)` devuelve VERDADERO si no están en orden los elementos a y b:

```
Mientras LISTA no ordenada, hacer
    Para iCasilla desde INICIO hasta FINAL-1
        Si NO_ORDENADOS(Vector(iCasilla), Vector(iCasilla+1)) entonces
           Variable_Auxiliar=Vector(iCasilla)
           Vector(iCasilla)=Vector(iCasilla+1)
           Vector(iCasilla+1)=Variable_Auxiliar
        Fin Si
   Siguiente iCasilla
Fin Mientras
```

Tabla de contenidos

[ocultar]

- 1 Implementación en C

- 2 Implementación en VB
- 3 Características
    - 3.1 Enlaces externos

[editar]

Implementación en C

A continuación viene una implementación en lenguaje C del algoritmo básico **bubblesort** para ordenar arrays de enteros de menor a mayor.

```
#define intercambia(x, y) register int temp=(x);(x)=(y);(y)=temp;

void bubble(int *start, int *end) {
  short fin;
  do {
      fin=0;
      for (int *i=start;i!=*end;i++) {
          if (*i>*(i+1)) {
              intercambia(i, i+1);
              fin=1;
          }
      }
  } while (fin==1);
} /* fin del método bubble */
```

[editar]

Implementación en VB

```
  Dim i As Integer
  Dim j As Integer
  Dim temp As Integer
  Dim swapped As Boolean

  'do the bubble sort
  For i = (num - 1) To 0 Step -1
      swapped = False
      For j = 1 To i
          If (numbers(j - 1) > numbers(j)) Then
              swapped = True
              temp = numbers(j - 1)
              numbers(j - 1) = numbers(j)
              numbers(j) = temp
          End If
      Next j
      If Not swapped Then
          i = -1
      End If
  Next i
```

[editar]

Características

Ventajas:

- Es bastante sencillo
- En un código reducido se realiza el ordenamiento
- Eficaz

Desventajas:

- Consume bastante tiempo de computadora
- Requiere muchas lecturas/escrituras en memoria
- Siempre hace la misma cantidad de comparaciones, incluso cuando el arreglo ya está ordenado.

6.1.2 Quicksort.

Esta es probablemente la técnica más rápida conocida. Fue desarrollada por C.A.R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). El algoritmo fundamental es el siguiente:

- Eliges un elemento de la lista. Puede ser cualquiera (en Optimizando veremos una forma más efectiva). Lo llamaremos **elemento de división**.
- Buscas la posición que le corresponde en la lista ordenada (explicado más abajo).
- Acomodas los elementos de la lista a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores (explicado más abajo también). En este momento el elemento de división separa la lista en dos sublistas (de ahí su nombre).
- Realizas esto de forma recursiva para cada sublista mientras éstas tengan un largo mayor que 1. Una vez terminado este proceso todos los elementos estarán ordenados.

Una idea preliminar para ubicar el elemento de división en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos contador por la izquierda, y j, al que llamaremos contador por la derecha. El algoritmo es éste:

- Recorres la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
- Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias.
- Repites esto hasta que se crucen los índices.
- El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
inf	Entero	Elemento inferior de la lista
sup	Entero	Elemento superior de la lista
elem_div	El mismo que los elementos de la lista	El elemento divisor
temp	El mismo que los elementos de la lista	Para realizar los intercambios
i	Entero	Contador por la izquierda
j	Entero	Contador por la derecha
cont	Entero	El ciclo continua mientras cont tenga el valor 1

```
Nombre Procedimiento: OrdRap
Parámetros:
   lista a ordenar (lista)
   índice inferior (inf)
   índice superior (sup)

   // Inicialización de variables
   1. elem_div = lista[sup];
   2. i = inf - 1;
   3. j = sup;
   4. cont = 1;

   // Verificamos que no se crucen los límites
   5. if (inf >= sup)
   6.      retornar;

   //  Clasificamos la sublista
   7. while (cont)
   8.      while (lista[++i] < elem_div);
   9.      while (lista[--j] > elem_div);
  10.      if (i < j)
  11.           temp = lista[i];
  12.           lista[i] = lista[j];
  13.           lista[j] = temp;
  14.      else
  15.           cont = 0;

  // Copiamos el elemento de división
  // en su posición final
  16. temp = lista[i];
  17. lista[i] = lista[sup];
  18. lista[sup] = temp;

  // Aplicamos el procedimiento
  // recursivamente a cada sublista
  19. OrdRap (lista, inf, i - 1);
  20. OrdRap (lista, i + 1, sup);
```

Nota:

- La primera llamada debería ser con la lista, cero (0) y el tamaño de la lista menos 1 como parámetros.

3. Un ejemplo

Esta vez voy a cambiar de lista ;-D

5 - 3 - 7 - 6 - 2 - 1 - 4

Comenzamos con la lista completa. El elemento divisor será el 4:

5 - 3 - 7 - 6 - 2 - 1 - **4**

Comparamos con el 5 por la izquierda y el 1 por la derecha.

5 - 3 - 7 - 6 - 2 - **1** - **4**

5 es mayor que cuatro y 1 es menor. Intercambiamos:

1 - 3 - 7 - 6 - 2 - **5** - **4**

Avanzamos por la izquierda y la derecha:

1 - **3** - 7 - 6 - **2** - 5 - **4**

3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.

1 - 3 - **7** - 6 - **2** - 5 - **4**

7 es mayor que 4 y 2 es menor: intercambiamos.

1 - 3 - **2** - 6 - **7** - 5 - **4**

Avanzamos por ambos lados:

1 - 3 - 2 - **6** - 7 - 5 - **4**

En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[sup] (pasos 16-18):

1 - 3 - 2 - **4** - 7 - 5 - **6**

Aplicamos recursivamente a la sublista de la izquierda (índices 0 - 2). Tenemos lo siguiente:

1 - **3** - **2**

1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha. Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[sup]:

1 - **2** - **3**

Al llamar recursivamente para cada nueva sublista (lista[0]-lista[0] y lista[2]-lista[2]) se retorna sin hacer cambios (condición 5.).Para resumir te muestro cómo va quedando la lista:

Segunda sublista: lista[4]-lista[6]

7 - **5** - **6**

5 - **7** - **6**

5 - **6** - **7**

Para cada nueva sublista se retorna sin hacer cambios (se cruzan los índices).

Finalmente, al retornar de la primera llamada se tiene el arreglo ordenado:

1 - 2 - 3 - 4 - 5 - 6 - 7

Eso es todo. Bastante largo ¿verdad?

4. Optimizando.

Sólo voy a mencionar algunas optimizaciones que pueden mejorar bastante el rendimiento de quicksort:

- Hacer una versión iterativa: Para ello se utiliza una pila en que se van guardando los límites superior e inferior de cada sublista.
- No clasificar todas las sublistas: Cuando el largo de las sublistas va disminuyendo, el proceso se va *encareciendo*. Para solucionarlo sólo se clasifican las listas que tengan un largo menor que **n**. Al terminar la clasificación se llama a otro algoritmo de ordenamiento que termine la labor. El indicado es uno que se comporte bien con listas casi ordenadas, como el ordenamiento por inserción por ejemplo. La elección de **n** depende de varios factores, pero un valor entre 10 y 25 es adecuado.
- Elección del elemento de división: Se elige desde un conjunto de tres elementos: lista[inferior], lista[mitad] y lista[superior]. El elemento elegido es el que tenga el valor medio según el criterio de comparación. Esto evita el comportamiento degenerado cuando la lista está prácticamente ordenada.

5. Análisis del algoritmo.

- Estabilidad: No es *estable*.
- Requerimientos de Memoria: No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila.

- Tiempo de Ejecución:
  - Caso promedio. La complejidad para dividir una lista de **n** es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:

    $f(1) = 1$

    $f(n) = n + 2 f(n/2)$

    La forma cerrada de esta expresión es:

    $f(n) = n \log_2 n$

    Es decir, la complejidad es $\mathbf{O(n \log_2 n)}$.

  - El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a $O(n^2)$. Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.

Ventajas:

- Muy rápido
- No requiere memoria adicional.

Desventajas:

- Implementación un poco más complicada.
- Recursividad (utiliza muchos recursos).
- Mucha diferencia entre el peor y el mejor caso.

La mayoría de los problemas de rendimiento se pueden solucionar con las optimizaciones mencionadas arriba (al costo de complicar mucho más la implementación). Este es un algoritmo que puedes utilizar en la vida real. Es muy eficiente. En general será la mejor opción. Intenta programarlo. Mira el código si tienes dudas.

Quicksort

De Wikipedia

Saltar a navegación, búsqueda

El **ordenamiento rápido** (**quicksort** en inglés) es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar *n* elementos en un tiempo proporcional a *n* log *n*. Esta es probablemente la técnica de ordenamiento más rápida conocida. Fue desarrollada por C. Antony R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos).

Tabla de contenidos

[ocultar]

- 1 Descripción del algoritmo
- 2 Optimización del algoritmo
  - 2.1 Técnicas de elección del pivote
  - 2.2 Técnicas de reposicionamiento
  - 2.3 Transición a otro algoritmo
- 3 Pseudocódigo con sintaxis de C
- 4 Ejemplo

- 5 Referencias

[editar]

Descripción del algoritmo

El algoritmo fundamental es el siguiente:

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es **$O(n \cdot \log n)$**.
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de **$0(n^2)$**. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas.
- En el caso promedio, el orden es **$O(n \cdot \log n)$**.

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del **pivote**.

[editar]

Optimización del algoritmo

[editar]

Técnicas de elección del pivote

El algoritmo básico Quicksort permite tomar cualquier elemento de la lista como pivote, aunque ya hemos visto que dependiendo del pivote que se elija, el algoritmo será más o menos eficiente.

- Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección «a ciegas» siempre provoca que el algoritmo tenga un orden de $O(n^2)$ para ciertas permutaciones de los elementos en la lista.
- Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en **O(n)** y asegura que hasta en el caso peor el algoritmo sea **O(n·log n)**. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.
- La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor de enmedio como pivote.

[editar]

Técnicas de reposicionamiento

Una idea preliminar para ubicar el **pivote** en su posición final sería contar la cantidad de elementos menores que él, y colocarlo un lugar más arriba, moviendo luego todos esos elementos menores que él a su izquierda, para que pueda aplicarse la recursividad.

Existe, no obstante, un procedimiento mucho más efectivo. Se utilizan dos índices: **i**, al que llamaremos índice izquierdo, y **j**, al que llamaremos índice derecho. El algoritmo es el siguiente:

- Recorrer la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
- Cuando lista[i] sea mayor que el pivote y lista[j] sea menor, se intercambian los elementos en esas posiciones.
- Repetir esto hasta que se crucen los índices.
- El punto en que se cruzan los índices es la posición adecuada para colocar el pivote, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

[editar]

Transición a otro algoritmo

Como se comentó antes, el algoritmo quicksort ofrece un orden de ejecución **$O(n^2)$** para ciertas permutaciones "críticas" de los elementos de la lista, que siempre surgen cuando se elige el pivote «a ciegas». La permutación concreta depende del pivote elegido, pero suele corresponder a secuencias ordenadas. Se tiene que la probabilidad de encontrarse con una de estas secuencias es inversamente proporcional a su tamaño.

- Los últimos pases de quicksort son numerosos y ordenan cantidades pequeña de elementos. Un porcentaje medianamente alto de ellos estarán dispuestos de una manera similar al peor caso del algoritmo, volviendo a éste ineficiente. Una solución a este problema consiste en ordenar las secuencias pequeñas usando otro algoritmo. Habitualmente se aplica el algoritmo de inserción para secuencias de tamaño menores de 8-15 elementos.
- Pese a que en secuencias largas de elementos la probabilidad de hallarse con una configuración de elementos "crítica" es muy baja, esto no evita que sigan apareciendo (a veces, de manera intencionada). El algoritmo introsort es una extensión del algoritmo quicksort que resuelve este problema utilizando heapsort en vez de quicksort cuando el número de recursiones excede al esperado.

[editar]

Pseudocódigo con sintaxis de C

Una idea preliminar para ubicar el pivote en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos contador por la izquierda, y j, al que llamaremos contador por la derecha. El algoritmo es éste:

```
Procedimiento:
   QuickSort

Parámetros:
   *lista a ordenar (lista)
   *índice inferior (inf)
   *índice superior (sup)
   // Inicialización de variables
   1. pivote = lista[sup];
   2. i = inf;
   3. j = sup - 1;
   4. cont = 1;

   // Verificamos que no se crucen los límites
   5. if (inf >= sup)
   6.      retornar;

   //  Clasificamos la sublista
   7. while (cont)
   8.      while (lista[++i] < pivote);
   9.      while (lista[--j] > pivote);
  10.      if (i < j)
  11.           temp = lista[i];
  12.           lista[i] = lista[j];
  13.           lista[j] = temp;
  14.      else
  15.           cont = 0;

  // Copiamos el pivote en su posición final
  16. temp = lista[i];
  17. lista[i] = lista[sup];
```

```
18. lista[sup] = temp;

// Aplicamos el procedimiento recursivamente a cada sublista
19. QuickSort (lista, inf, i - 1);
20. QuickSort (lista, i + 1, sup);
```

Nota: La primera llamada debería tomar los parámetros la lista, 0 (cero) y el tamaño de la lista menos 1.

[editar]

Ejemplo

En el siguiente ejemplo se marcan el pivote y los índices i y j con las letras p, i y j respectivamente.

Comenzamos con la lista completa. El elemento divisor será el 4:

```
5 - 3 - 7 - 6 - 2 - 1 - 4
                        p
```

Comparamos con el 5 por la izquierda y el 1 por la derecha.

```
5 - 3 - 7 - 6 - 2 - 1 - 4
i                   j   p
```

5 es mayor que cuatro y 1 es menor. Intercambiamos:

```
1 - 3 - 7 - 6 - 2 - 5 - 4
i                   j   p
```

Avanzamos por la izquierda y la derecha:

```
1 - 3 - 7 - 6 - 2 - 5 - 4
    i           j       p
```

3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.

```
1 - 3 - 7 - 6 - 2 - 5 - 4
        i       j       p
```

7 es mayor que 4 y 2 es menor: intercambiamos.

```
1 - 3 - 2 - 6 - 7 - 5 - 4
        i       j       p
```

Avanzamos por ambos lados:

```
1 - 3 - 2 - 6 - 7 - 5 - 4
           iyj          p
```

En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[sup] (pasos 16-18):

```
1 - 3 - 2 - 4 - 7 - 5 - 6
            p
```

Aplicamos recursivamente a la sublista de la izquierda (índices 0 - 2). Tenemos lo siguiente:

```
1 - 3 - 2
```

1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha. Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[sup]:

```
1 - 2 - 3
```

El mismo procedimiento se aplicará a la otra sublista. Al finalizar y unir todas las sublistas queda la lista inicial ordenada en forma ascendente.

```
1 - 2 - 3 - 4 - 5 - 6 - 7
```

6.1.3 ShellSort.

Ordenación Shell Sort

De Wikipedia

Saltar a navegación, búsqueda

El **Shell Sort** es un algoritmo de ordenación de disminución incremental, nombrado así debido a su inventor Donald Shell. El rendimiento de este algorítmo depende del orden de la tabla inicial y va de n^2 en el caso peor a $n^{4/3}$ *y se puede mejorar.*

Tabla de contenidos

[ocultar]

- 1 Descripción del algoritmo
- 2 Algoritmo Shell en C++
- 3 Pseudocódigo en C del Shell Sort
- 4 Pseudocódigo de Shell Sort en Pascal

- 5 Ejemplo

[editar]

Descripción del algoritmo

El algoritmo ordena subgrupos de elementos separados K unidades (respecto de su posición en la tabla) en la tabla original. El valor K es llamado incremento. Después de que los primeros K

subgrupos han sido ordenados (generalmente utilizando [[Ordenamiento por inserción]), se escoge un nuevo valor de K más pequeño, y la tabla es de nuevo partida entre el nuevo conjunto de subgrupos. Cada uno de los subgrupos mayores es ordenado y el proceso se repite de nuevo con un valor más pequeño de K hasta que en algún momento k llega a valer 1.

[editar]

Algoritmo Shell en C++

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define n 100


void shell_sort(int A[], int size)
{
  int i, j, incrmnt, temp;


  incrmnt = 3;
  while (incrmnt > 0)
  {
    for (i=0; i < size; i++)
    {
      j = i;
      temp = A[i];
      while ((j >= incrmnt) && (A[j-incrmnt] > temp))
      {
        A[j] = A[j - incrmnt];
        j = j - incrmnt;
      }
      A[j] = temp;
    }
    if (incrmnt/2 != 0)
      incrmnt = incrmnt/2;
    else if (incrmnt == 1)
      incrmnt = 0;
    else
      incrmnt = 1;
  }
//imprime arreglo
 for(i=0;i<n;i++)
        printf("%d ",A[i]);


}


main()
{
 int A[n],a,i;

  clrscr();
  randomize();

  for (i=0;i<n;i++)
```

```
        A[i]=random(5);

 shell_sort(A,n);
 getch();
 return 0;
}
```
[editar]

Pseudocódigo en C del Shell Sort

```
 /* Ordenación por ShellSort
 tabla T: tabla a ordenar
 índice P = principio de la tabla
 índice U = final de la tabla
 tabla Inc = tabla con los incrementos de mayor a menor. Un ejemplo podría ser
4,3,2,1
 entero nInc = número de incrementos
 */
 ShellSort (tabla T, indice P, indice U)

 tabla Inc = [4,3,2,1]
 entero nInc = 4

 for (i=1; 1<nInc; i++)
   for (j=0; j<Inc[i]; j++)
     InsertSortconIncrementos (T, P+j-1, Inc[i])

 /* Es el InsertSort pero con subtablas */
 InsertSortconIncrementos (tabla T, indice P, indice U, entero k)
     i = P + k;
     while (i <= U)
       A = T[i]
       j = i - k
       while (j >= P ) y (T[j] > A)
         T[j+k] = T[j]
         j = j - k
       T[j+k] = A
     i = i + k
```
[editar]

Pseudocódigo de Shell Sort en Pascal

```
Procedimiento Shell Sort;

const MAXINC = _____;
incrementos = array[1..MAXINC] of integer;
var j,p,num,incre,k:integer;
begin
    for incre := 1 to MAXINC do begin
        k := inc[incre]; /* k recibe un tipo de incremento */
        for p := k+1 to MAXREG do begin /* inserción directa para el grupo que
se encuentra
                                                   cada k posiciones */
            num := reg[p];
            j := p-k;
            while (j>0) AND (num < reg[j]) begin
```

```
                reg[j+k] := reg[j];
                j := j - k;
            end;
            reg[j+k] := num;
        end
    end
end;
```
[editar]

Ejemplo

Para el arreglo a = [6, 1, 5, 2, 3, 4, 0] Tenemos el siguiente recorrido:

```
Recorrido   Salto    Lista Ordenada   Intercambio
1           3        2,1,4,0,3,5,6    (6,2), (5,4), (6,0)
2           3        0,1,4,2,3,5,6    (2,0)
3           3        0,1,4,2,3,5,6    Ninguno
4           1        0,1,2,3,4,5,6    (4,2), (4,3)
5           1        0,1,2,3,4,5,6    Ninguno
```

6.2 Algoritmos de ordenamiento por Distribución.

6.2.1 Radix.

Radix sort

De Wikipedia

Saltar a navegación, búsqueda

Radix Sort (o ordenamiento Radix) es un algoritmo de ordenamiento estable que puede ser usado para ordenar items identificados por llaves (o claves) únicas. Cada llave debe ser una cadena o un número capaz de ser ordenada alfanuméricamente.

Tabla de contenidos

[ocultar]

- 1 Reglas para ordenar
- 2 Un ejemplo escrito en Lenguaje C
  - 2.1 Características
  - 2.2 Ventajas

[editar]

Reglas para ordenar

Empezar en el dígito más significativo y avanzar por los dígitos menos significativos mientras coinciden los dígitos correspondientes en los dos números. El número con el dígito más grande

en la primera posición en la cual los dígitos de los dos números no coinciden es el mayor de los dos (por supuesto sí coinciden todos los dígitos de ambos números, son iguales). Este mismo principio se toma para Radix Sort, para visualizar esto mejor tenemos el siguiente ejemplo. En el ejemplo anterior se ordeno de izquierda a derecha. Ahora vamos a ordenar de derecha a izquierda.

Archivo original.

25 57 48 37 12 92 86 33

Asignamos colas basadas en el dígito menos significativo.

Parte delantera Parte posterior

0

1

2 12 92

3 33

4

5 25

6 86

7 57 37

8 48

9

10

Después de la primera pasada:

12 92 33 25 86 57 37 48

Colas basadas en el dígito más significativo.

Parte delantera Parte posterior

0

1 12

2 25

3 33 37

4 48

5 57

6

7

8 86

9 92

10

Archivo ordenado: 12 25 33 37 48 57 86 92

A S O R T I N G E X A M P L E

A E O L M I N G E A X T P R S

A E A E G I N M L O

A A E E G

A A

A A

E E G

E E

I N M L O

L M N O

L M

N O

S T P R X

S R P T

P R S

R S

[editar]

Un ejemplo escrito en Lenguaje C

```
#include <stdio.h>
#include <stdlib.h>
#define NUMELTS 20

void radixsort(int x[], int n)
{
  int front[10], rear[10];
  struct {
    int info;
    int next;
  } node[NUMELTS];
  int exp, first, i, j, k, p, q, y;

  /* Inicializar una lista viculada */
  for (i = 0; i < n-1; i++) {
    node[i].info = x[i];
    node[i].next = i+1;
  } /* fin del for */
  node[n-1].info = x[n-1];
  node[n-1].next = -1;
  first = 0; /*first es la cabeza de la lista vinculada */
  for (k = 1; k < 5; k++) {
    /* Suponer que tenemos n£meros de cuatro dÁgitos */
    for (i = 0; i < 10; i++) {
      /*Inicializar colas */
      rear[i] = -1;
      front[i] = -1;
    } /*fin del for */
    /* Procesar cada elemento en la lista */
    while (first != -1) {
      p = first;
      first = node[first].next;
      y = node[p].info;
      /* Extraer el kâsimo dÁgito */
      exp = pow(10, k-1);       /* elevar 10 a la (k-1)âsima potencia */
      j = (y/exp) % 10;
      /* Insertar y en queue[j] */
      q = rear[j];
      if (q == -1)
        front[j] = p;
      else
        node[q].next = p;
```

```
      rear[j] = p;
    } /*fin del while */

    /* En este punto, cada registro est  en su cola bas ndose en el dÁgito k
       Ahora formar una lista £nica de todos los elementos de la cola.
Encontrar
       el primer elemento. */
    for (j = 0; j < 10 && front[j] == -1; j++);
      ;
    first = front[j];

    /* Vincular las colas restantes */
    while (j <= 9) {         /*Verificar si se ha terminado */
      /*Encontrar el elemento siguiente */
      for (i = j+1; i < 10 && front[i] == -1; i++);
        ;
      if (i <= 9) {
        p = i;
        node[rear[j]].next = front[i];
      } /* fin del if */
      j = i;
    } /* fin del while */
    node[rear[p]].next = -1;
  } /* fin del for */

  /* Copiar de regreso al archivo original */
  for (i = 0; i < n; i++) {
    x[i] = node[first].info;
    first = node[first].next;
  } /*fin del for */
} /* fin de radixsort*/


int main(void)
{
  int x[50] = {NULL}, i;
  static int n;

  printf("\nCadena de n£meros enteros: \n");
  for (n = 0;; n++)
    if (!scanf("%d", &x[n])) break;
  if (n)
    radixsort (x, n);
  for (i = 0; i < n; i++)
    printf("%d ", x[i]);
  return 0;
}
```

Estado de la lista

i

```
Node[i].info
Node[i].next
```

Inicialización K = 1 K = 2 K = 3

0

```
65
1
3
1
2
```

1

```
789
2
-1
-1
-1
```

2

```
123
3
0
3
3
```

3

```
457
-1
1
0
1
```

rear = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

2 0 3 1

front = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

2 0 3 1

k = 1

p = 0 p = 1 p = 2 p = 3

first = 1 first = 2 first = 3 first = -1

y = 65 y = 789 y = 123 y = 457

exp = 1 exp = 1 exp = 1 exp = 1

j = 5 j = 9 j = 3 j = 7

q = -1 q = -1 q = -1 q = -1

si q == -1 si q == -1 si q == -1 si q == -1

front[5] = 0 front[9] = 1 front[3] = 2 front[7] = 3

rear[5] = 0 rear[9] = 1 rear[3] = 2 rear[7] = 3

j = 3 first = 2 while (j <= 9) i = 5 si i <= 9

```
                  p = 5
                        node[2].next = 0
                       j = 5
      i = 7
      si i <= 9
                       p = 7
                       node[0].next = 3
                       j = 5
      i = 9
      si i <= 9
                       p = 9
                        node[2].next = 1
                       j = 9
```

fin del while p = 9 node[1].next = -1

[editar]

Características

Debido a que el ciclo for (k = 1; k <= m; k++) externo se recorre m veces (una para cada dígito) y el ciclo interior n veces (una para cada elemento en el archivo) el ordenamiento es de aproximadamente (m*n). Si las llaves son complejas (es decir, si casi cada número que puede ser una llave lo es en realidad) m se aproxima a log n, por lo que (m*n) se aproxima a (n log n). Si la cantidad de dígitos es grande, en ocasiones es más eficiente ordenar el archivo aplicando primero el ordenamiento de raíz a los dígitos más significativos y después utilizando inserción directa sobre el archivo ordenado.

Ventajas

El ordenamiento es razonablemente eficiente si el número de dígitos en las llaves no es demasiado grande. Si las máquinas tienen la ventaja de ordenar los dígitos (sobre todo si están en binario) lo ejecutarían con mucho mayor rapidez de lo que ejecutan una comparación de dos llaves completas y desiguales..

Unidad 7 Ordenación externa.

ORDENACIÓN EXTERNA

...o como perder menos tiempo, y hacer menor número de accesos a la memoria.

INDICE TEMA 4.

4.1 Justificación

4.2 Ordenación por mezcla

Ordenación externa

4.3.1. Mezcla directa

4.3.2. Mezcla natural

4.4. Intercalación múltiple

1. JUSTIFICACIÓN

Volumen de datos amplio

TamaÒo de memoria limitado

Objetivo: minimizar el número de accesos a los ficheros

2. ORDENACIÓN POR MEZCLA.

- N ficheros `ordenados' de unen para formar un único archivo ordenado:

A	B	Operación C
Vacio	Vacio	nada
Vacio	No vacio	copiar todos los registros de **B** en **C**
No vacio	Vacio	copiar todos los registros de **A** en **C**
No vacio	No vacio	bucle a repetir hasta que **A** o **B** estén vacios (*)

(*) Operaciones a realizar en este caso:

Leer (A, ElemA)

Leer (B, ElemB)

Bucle hasta llegar al final de A o B

Comparar (ElemA, ElemB)

if ElemA > ElemB

then Escribir ElemB en C

Leer B

else Escribir ElemA en C

Leer A

Introducir el resto de A o B en C

3. ORDENACIÓN EXTERNA.

- **MEZCLA DIRECTA**

Se ordenan las secuencias de un archivo en grupos de tamaÒo creciente = 2n (1, 2, 4, 8, …):

Sea la secuencia:

22 6 25 48 32 5 12 20 31 35

Para grupos de tamaÒo = 1, tenemos:

22 25 32 12 31

6 48 5 20 35

Ordenando estas secuencias:

6 , 22 25 , 48 5 , 32 12 , 20 31 , 35

Agrupando en grupos de tamaÒo = 2:

6 , 22 5 , 32 31 , 35

25 , 48 12 , 20

Ordenando estos grupos:

6 , 22 , 25 , 48 5 , 12 , 30 , 32 , 31 , 35

Agrupando en grupos de tamaÒo = 4:

6 , 22 , 25 , 48 31 , 35

5 , 12 , 20 , 32

Ordenando:

5 , 6 , 12 , 20 , 22 , 25 , 32 , 48 31, 35

La ultima agrupación (tamaÒo = 8):

5 , 6 , 12 , 20 , 22 , 25 , 32 , 48

31 , 35

Ordenando:

5 , 6 , 12 , 20 , 22 , 25 , 31 , 32 , 35 , 48

El procedimiento a seguir es el siguiente:

Dividir la secuencia a ordenar en 2 subsecuencias de menor tamaÒo, cada una la mitad de la secuencia original.

Mezclar las dos secuencias de forma ordenada, para generar otra mayor, formada por cinjuntos ordenados de valores con 20, 21, 22, ... elementos.

Iterar los pasos 1 y 2 hasta que el tamaÒo del conjunto ordenado (2n) sea mayor que el número de elemtentos a ordenar.

- **MEZCLA NATURAL**

Las secuencias intermedias no tienen tamaÒo prefijado ni longitud constante. Estas se generan con sus elementos ordenados, separando un elemento nuevo a otra secuencia si no se respeta esta condición.

Se incluyen separadores de secuencia.

Sea la cadena de numeros:

F1:

3 15 7 17 9 8 14 2 41 24 36 1 13 42 26 35 5 33

Aplicando el método de mezcla natural con tres ficheros, uno original y dos auxiliares (frecuententemente llamados `**cintas**'), el resultado es:

F2:

3 15 9 2 41 1 13 42 5 33

7 17 8 14 24 36 36 35

7.1 Algoritmos de ordenación externa.

7.1.1 Intercalación directa.

Mezcla Directa

Este algoritmo consiste en tener un archivo A desordenado. Este archivo se ordenara haciendo los siguientes pasos:

1.- Definir tamaño de tramo igual a 1

2.- Repartir el archivo A en dos archivos B y C escribiendo alternadamente un tramo en cada uno
3.- Aplicar el algoritmo de mezcla simple a cada par de tramos correspondiente de los archivos B y C guardando el resultado en el archivo A
4.- Duplicar el tamaño del tramo
6.- regresar al paso 2 si el tamaño del tramo es menor que la cantidad de elementos a ordenar.

```
void mezcla_directa(char *nom)
  {
    int t=1;
   while(t<n)
     {
        partir(nom,t);
        mezclar(nom,t);
        t* = 2;
      }
}

void partir(char *nom, int t)
{
  FILE *a,*b,*c;
  int reg, cont, sw=1;
  a=fopen(nom,"rb");
  b=fopen("m1","wb");
  c=fopen("m2","wb");
  if(a&&b&&c)
    {
      fread(&reg,sizeof(reg),1,a);
      while(!feof(a))
          {
            for(cont=0;cont<t && !feof(a);cont++)
              {
                if(sw)
                    fwrite(&reg,sizeof(reg),1,b);
                else
                   fwrite(&reg,sizeof(reg),1,c);
                   fread(&reg,sizeof(reg),1,a);
              }
            sw=!sw ;
           }
      }
  fclose(a);
  fclose(b);
  fclose(c);
}

void mezclar(char *nom, int t)
```

```
{
  FILE *a,*b,*c;
  int rb,rc,ctb,ctc;
  a= fopen(nom,"wb");
  b= fopen("m1","rb");
  c= fopen("m2","rb");
   if(a&&b&&c)
     {
       fread(&rb,sizeof(rb),1,b);
       fread(&rc,sizeof(rb),1,c);
      while(!feof(b) && !feof(c))
        {
          ctb=ctc=t;
          while(ctb && ctc && !feof(b) && !feof(c))
            {
              if(rb<rc)
               {
                 fwrite(&rb,sizeof(rb),1,a);
                 fread(&rb,sizeof(rb),1,b);
                 ctb--;
                }
              else
                 {
                   fwrite(&rc,sizeof(rc),1,a);
                   fread(&rc,sizeof(rc),1,c);
                   ctc--;
                  }
              }
            while(ctb && !feof(b))
                {
                  fwrite(&rb,sizeof(rb),1,a);
                  fread(&rb,sizeof(rb),1,b);
                  ctb--;
                 }
                  while(ctc && !feof(c))
                      {
                        write(&rc,sizeof(rc),1,a);
                        fread(&rc,sizeof(rc),1,c);
                        ctc--;
                       }
        }
            while(!feof(b))
                {
                  fwrite(&rb,sizeof(rb),1,a);
                  fread(&rb,sizeof(rb),1,b);
                 }
                  while(!feof(c))
                     {
                         write(&rc,sizeof(rc),1,a);
```

```
                                fread(&rc,sizeof(rc),1,c);
                            }
    }
fclose(a);
fclose(b);
fclose(c);
remove("m1");
remove("m2");
}
```

Intercalación simple

Supone dos archivos ordenados y obtiene al final un solo archivo ordenado que contiene los elementos de los dos archivos iniciales.

```
Void mezcla_simple(char *A,char *B,char *C)
{
    FILE a*,b*,c*;
    int ra,rb:
    a=fopen(A,"rb");
    b=fopen(B,"rb");
    c=fopen(C,"rb");
      if(a&&b&&c)
        {
          fread(&ra,sizeof(ra),1,a);
          fread(&rb,sizeof(rb),1,b);
          while(!feof(a)&&!feof(b))
            {
              if(ra<=rb)
                {
                    fwrite(&ra,sizeof(ra),1,c);
                    fread(&ra,sizeof(ra),1,a);
                }
             else
                {
                   fwrite(&rb,sizeof(rb),1,c);
                   fread(&ra,sizeof(ra),1,b);
                }
           }
              while(!feof(a))
                  {
                    fwrite(&ra,sizeof(ra),1,c);
                    fread(&ra,sizeof(ra),1,a);
                  }
            while(!feof(a))
               {
                 fwrite(&rb,sizeof(rb),1,c);
                 fread(&rb,sizeof(rb),1,b);
               }
```

```
  fclose(a);
  fclose(b);
  fclose(c);
  }

}
```

7.1.2 Mezcla natural.

Es una mejora del algoritmo de mezcla directa puesto que en vez de considerar tramos de tamaño fijo se toman en cuenta para la ordenación en todo momento tramos de longitud máxima.

Al igual que el mezcla directa se debe hacer un proceso de partir el archivo original para mezclarlo, posteriormente mientras en el archivo C haya elementos a mezclar.

```
FILE *a,*b,*c;
void mezcla_natural( char nom[])
  {
    unsigned long int tc;
    tc =partir(nom);
     while(tc)
        {
         mezclar(nom);
         tc=partir(nom);
        }
        remove("m1");
        remove("m2");
    }

unsigned long int partir(char *nom)
  {
     unsigned long int tamc =0;
     int sw=1;
     nodonumeros ant,ra;
     a=fopen(nom,"rb");
     b=fopen("m1","wb");
     c=fopen("m2","wb");
      if(a&&b&&c)
        {
         fread(&ra,sizeof(ra),1,a);
         ant=ra;
        while(!feof(a))
          {
            if(ant.numero>ra.numero)
              sw=!sw;
            if(sw)
              fwrite(&ra,sizeof(ra),1,b);
           else
```

```
              fwrite(&ra,sizeof(ra),1,c);
              ant=ra;
              fread(&ra,sizeof(ra),1,a);
             }
             tamc =ftell(c);
       }
fclose(a);
fclose(b);
fclose(c);
return tamc;
}


void mezclar(char *nom)
{
  int ftb,ftc;
  a=fopen(nom,"wb");
  b=fopen("m1","rb");
  c=fopen("m2","rb");
    if(a&&b&&c)
      {
        fread(&rb,sizeof(rb),1,b);
        fread(&rc,sizeof(rc),1,c);
        antb=rb;
        antc=rc;
        while(!feof(b) && !feof(c))
          {
            ftb=ftc=0;
           while(!ftb && !ftc && !feof(b) && !feof(c))
              {
                  if(rb.numero<=rc.numero)
                     {
                      fwrite(&rb,sizeof(rb),1,a);
                      antb=rb;
                      fread(&rb,sizeof(rb),1,b);
                      if(antb.numero>rb.numero)
                      ftb=1;
                     }
                  else
                    {
                       fwrite(&rc,sizeof(rc),1,a);
                       antc=rc;
                       fread(&rc,sizeof(rc),1,c);
                       if(antc.numero>rc.numero)
                         ftc=1;
                    }
              }

       while(!ftb && !feof(b))
```

```
          {
            fwrite(&rb,sizeof(rb),1,a);
            antb=rb;
            fread(&rb,sizeof(rb),1,b);
            if(antb.numero>rb.numero)
               ftb=1;
           }
            while(!ftc && !feof(c))
               {
                 fwrite(&rc,sizeof(rc),1,a);
                 antc=rc;
                 fread(&rc,sizeof(rc),1,c);
                if(antc.numero>rc.numero)
                     ftc=1;
                }

}

          while(!feof(b))
               {
                 fwrite(&rb,sizeof(rb),1,a);
                 fread(&rb,sizeof(rb),1,b);
                }
           while(!feof(c))
               {
                 fwrite(&rc,sizeof(rc),1,a);
                 fread(&rc,sizeof(rc),1,c);
               }
          }

fclose(a);
fclose(b);
fclose(c);
}
```

Unidad 8 Métodos de búsqueda.

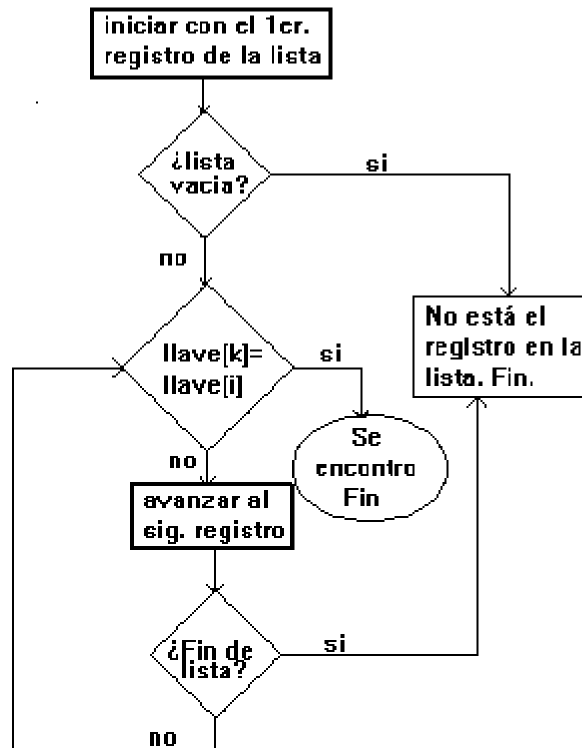
8.1 Algoritmos de ordenación externa.

8.1.1 Secuencial.

Búsqueda secuencial

La búsqueda es el proceso de localizar un registro (elemento) con un valor de llave particular. La búsqueda termina exitosamente cuando se localiza el registro que contenga la llave buscada, o termina sin éxito, cuando se determina que no aparece ningún registro con esa llave.

Búsqueda secuencial, también se le conoce como búsqueda lineal. Supongamos una colección de registros organizados como una lista lineal. El algoritmo básico de búsqueda secuencial consiste en empezar al inicio de la lista e ir a través de cada registro hasta encontrar la llave indicada (k), o hasta al final de la lista.



La situación óptima es que el registro buscado sea el primero en ser examinado. El peor caso es cuando las llaves de todos los n registros son comparados con k (lo que se busca). El caso promedio es n/2 comparaciones.

Este método de búsqueda es muy lento, pero si los datos no están en orden es el único método que puede emplearse para hacer las búsquedas. Si los valores de la llave no son únicos, para encontrar todos los registros con una llave particular, se requiere buscar en toda la lista.

Mejoras en la eficiencia de la búsqueda secuencial

1)Muestreo de acceso

Este método consiste en observar que tan frecuentemente se solicita cada registro y ordenarlos de acuerdo a las probabilidades de acceso detectadas.

2)Movimiento hacia el frente

Este esquema consiste en que la lista de registros se reorganicen dinámicamente. Con este método, cada vez que búsqueda de una llave sea exitosa, el registro correspondiente se mueve a la primera posición de la lista y se recorren una posición hacia abajo los que estaban antes que el.

3)Transposición

Este es otro esquema de reorganización dinámica que consiste en que, cada vez que se lleve a cabo una búsqueda exitosa, el registro correspondiente se intercambia con el anterior. Con este procedimiento, entre mas accesos tenga el registro, mas rápidamente avanzara hacia la primera posición. Comparado con el método de movimiento al frente, el método requiere mas tiempo de actividad para reorganizar al conjunto de registros . Una ventaja de método de transposición es que no permite que el requerimiento aislado de un registro, cambie de posición todo el conjunto de registros. De hecho, un registro debe ganar poco a poco su derecho a alcanzar el inicio de la lista.

4)Ordenamiento

Una forma de reducir el numero de comparaciones esperadas cuando hay una significativa frecuencia de búsqueda sin éxito es la de ordenar los registros en base al valor de la llave. Esta técnica es útil cuando la lista es una lista de excepciones, tales como una lista de decisiones, en cuyo caso la mayoría de las búsquedas no tendrán éxito. Con este método una búsqueda sin éxito termina cuando se encuentra el primer valor de la llave mayor que el buscado, en lugar de la final de la lista.

8.1.2 Binaria.

Se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda. Los prerrequisitos principales para la búsqueda binaria son:

- La lista debe estar ordenada en un orden especifíco de acuerdo al valor de la llave.
- Debe conocerse el número de registros.

Algoritmo

1. Se compara la llave buscada con la llave localizada al centro del arreglo.
2. Si la llave analizada corresponde a la buscada fin de búsqueda si no.
3. Si la llave buscada es menor que la analizada repetir proceso en mitad superior, sino en la mitad inferior.
4. El proceso de partir por la mitad el arreglo se repite hasta encontrar el registro o hasta que el tamaño de la lista restante sea cero , lo cual implica que el valor de la llave buscada no esta en la lista.

El esfuerzo máximo para este algoritmo es de $\log_2 n$. El mínimo de 1 y en promedio ½ $\log_2 n$.

8.1.3 Hash.

Hasta ahora las técnicas de localización de registros vistas, emplean un proceso de búsqueda que implica cierto tiempo y esfuerzo. El siguiente método nos permite encontrar directamente el registro buscado.

La idea básica de este método consiste en aplicar una función que traduce un conjunto de posibles valores llave en un rango de direcciones relativas. Un problema potencial encontrado en este proceso, es que tal función no puede ser uno a uno; las direcciones calculadas pueden no ser todas únicas, cuando $R(k_1) = R(k_2)$
Pero : K_1 diferente de K_2 decimos que hay una **colisión**. A dos llaves diferentes que les corresponda la misma dirección relativa se les llama **sinónimos**.

A las técnicas de calculo de direcciones también se les conoce como :

- Técnicas de almacenamiento disperso
- Técnicas aleatorias
- Métodos de transformación de llave - a- dirección
- Técnicas de direccionamiento directo
- Métodos de tabla Hash
- Métodos de Hashing

Pero el término mas usado es el de hashing. Al cálculo que se realiza para obtener una dirección a partir de una llave se le conoce como función hash.

Ventaja

1. Se pueden usar los valores naturales de la llave, puesto que se traducen internamente a direcciones fáciles de localizar
2. Se logra independencia lógica y física, debido a que los valores de las llaves son independientes del espacio de direcciones
3. No se requiere almacenamiento adicional para los índices.

Desventajas

1. No pueden usarse registros de longitud variable
2. El archivo no esta clasificado
3. No permite llaves repetidas
4. Solo permite acceso por una sola llave

Costos

- Tiempo de procesamiento requerido para la aplicación de la función hash
- Tiempo de procesamiento y los accesos E/S requeridos para solucionar las colisiones.

La eficiencia de una función hash depende de:

1. La distribución de los valores de llave que realmente se usan

2. El numero de valores de llave que realmente están en uso con respecto al tamaño del espacio de direcciones
3. El numero de registros que pueden almacenarse en una dirección dad sin causar una colisión
4. La técnica usada para resolver el problema de las colisiones

Las funciones hash mas comunes son:

- Residuo de la división
- Medio del cuadrado
- Pliegue

HASHING POR RESIDUO DE LA DIVISIÓN

La idea de este método es la de dividir el valor de la llave entre un numero apropiado, y después utilizar el residuo de la división como dirección relativa para el registro (dirección = llave módulo divisor).

Mientras que el valor calculado real de una dirección relativa, dados tanto un valor de llave como el divisor, es directo; la elección del divisor apropiado puede no ser tan simple. Existen varios factores que deben considerarse para seleccionar el divisor:

1. El rango de valores que resultan de la operación "llave % divisor", va desde cero hasta el divisor 1. Luego, el divisor determina el tamaño del espacio de direcciones relativas. Si se sabe que el archivo va a contener por lo menos n registros, entonces tendremos que hacer que divisor > n, suponiendo que solamente un registro puede ser almacenado en una dirección relativa dada.
2. El divisor deberá seleccionarse de tal forma que la probabilidad de colisión sea minimizada. ¿Como escoger este numero? Mediante investigaciones se ha demostrado que los divisores que son números pares tienden a comportase pobremente, especialmente con los conjuntos de valores de llave que son predominantemente impares. Algunas investigaciones sugieren que el divisor deberá ser un numero primo. Sin embargo, otras sugieren que los divisores no primos trabajan también como los divisores primos, siempre y cuando los divisores no primos no contengan ningún factor primo menor de 20. Lo mas común es elegir el número primo mas próximo al total de direcciones.

Ejemplo:

Independientemente de que tan bueno sea el divisor, cuando el espacio de direcciones de un archivo esta completamente lleno, la probabilidad de colisión crece dramáticamente. La saturación de archivo de mide mediante su factor de carga, el cual se define como la relación del numero de registros en el archivo contra el numero de registros que el archivo podría contener si estuviese completamente lleno.

$$\text{Factor de carga} = \frac{\text{número de registro en el archivo}}{\text{máximo número de registro que puede contener el archivo}}$$

Todas las funciones hash comienzan a trabajar probablemente cuando el archivo esta casi lleno. Por lo general el máximo factor de carga que puede tolerarse en un archivo para un rendimiento razonable es de entre el 70 % y 80 %.

HASHING POR MEDIO DEL CUADRADO

En esta técnica, la llave es elevada al cuadrado, después algunos dígitos específicos se extraen de la mitad del resultado para constituir la dirección relativa. Si se desea una dirección de n dígitos, entonces los dígitos se truncan en ambos extremos de la llave elevada al cuadrado, tomando n dígitos intermedios. Las mismas posiciones de n dígitos deben extraerse para cada llave.

Ejemplo:

Utilizando esta función hashing el tamaño del archivo resultante es de 10^n donde n es el numero de dígitos extraídos de los valores de la llave elevada al cuadrado.

HASHING POR PLIEGUE

En esta técnica el valor de la llave es particionada en varias partes, cada una de las cuales
(excepto la ultima) tiene el mismo numero de dígitos que tiene la dirección relativa objetivo. Estas particiones son después plegadas una sobre otra y sumadas. El resultado, es la dirección relativa. Igual que para el método del medio del cuadrado, el tamaño del espacio de direcciones relativas es una potencia de 10.

Ejemplo:

COMPARACIÓN ENTRE LAS FUNCIONES HASH

Aunque alguna otra técnica pueda desempeñarse mejor en situaciones particulares, la técnica del residuo de la división proporciona el mejor desempeño. Ninguna función hash se desempeña siempre mejor que las otras. El método del medio del cuadrado puede aplicarse en archivos con factores de cargas bastantes bajas para dar generalmente un buen desempeño. El método de pliegues puede ser la técnica mas fácil de calcular pero produce resultados bastante erráticos, a menos que la longitud de la llave se aproximadamente igual a la longitud de la dirección.

Si la distribución de los valores de llaves no es conocida, entonces el método del residuo de la división es preferible. Note que el hashing puede ser aplicado a llaves no numéricas. Las posiciones de ordenamiento de secuencia de los caracteres en un valor de llave pueden ser utilizadas como sus equivalentes "numéricos". Alternativamente, el algoritmo hash actúa sobre las representaciones binarias de los caracteres.
Todas las funciones hash presentadas tienen destinado un espacio de tamaño fijo. Aumentar el tamaño del archivo relativo creado al usar una de estas funciones, implica

cambiar la función hash, para que se refiera a un espacio mayor y volver a cargar el nuevo archivo.

MÉTODOS PARA RESOLVER EL PROBLEMA DE LAS COLISIONES

Considere las llaves K_1 y K_2 que son sinónimas para la función hash R. Si K_1 es almacenada primero en el archivo y su dirección es $R(K_1)$, entonces se dice que K_1 esta almacenado en su dirección de origen.

Existen dos métodos básicos para determinar donde debe ser alojado K_2 :

- **Direccionamiento abierto.-** Se encuentra entre dirección de origen para K_2 dentro del archivo.
- **Separación de desborde (Area de desborde).-** Se encuentra una dirección para K_2 fuera del área principal del archivo, en un área especial de desborde, que es utilizada exclusivamente para almacenar registro que no pueden ser asignados en su dirección de origen

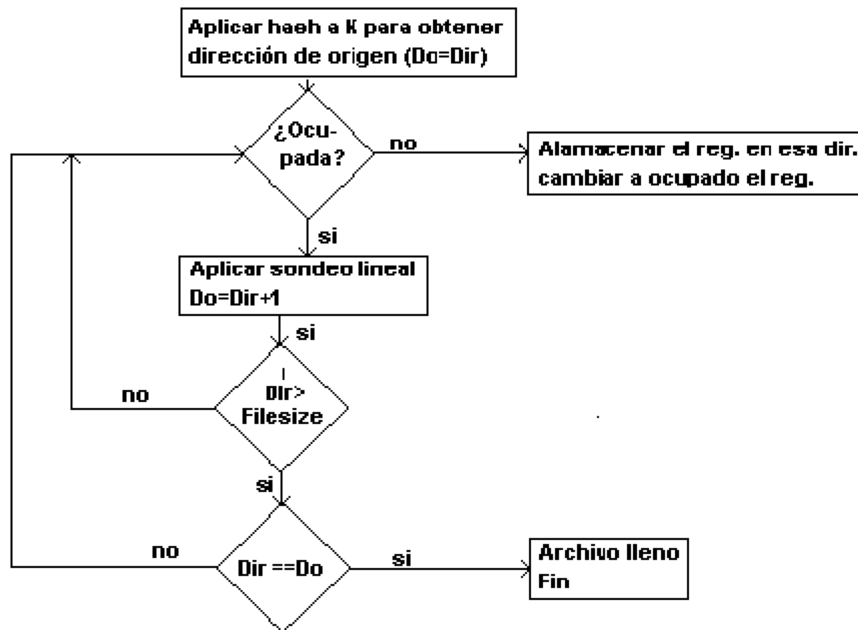
Los métodos mas conocidos para resolver colisiones son:

Sondeo lineal

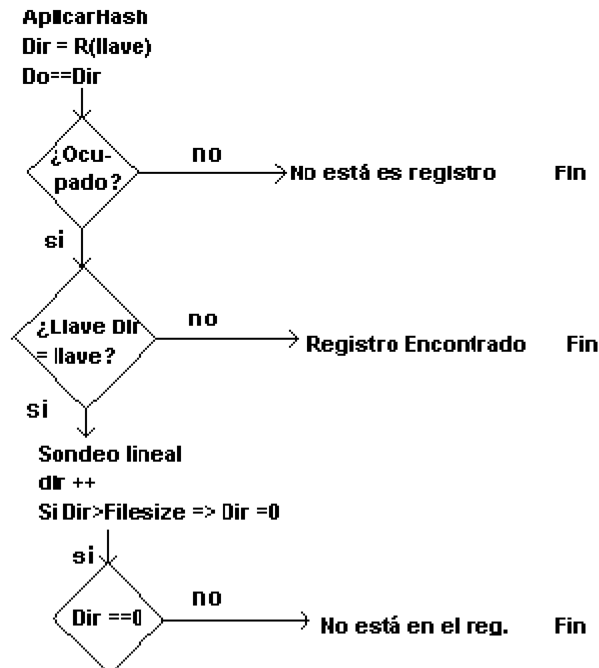
Que es una técnica de direccionamiento abierto. Este es un proceso de búsqueda secuencial desde la dirección de origen para encontrar la siguiente localidad vacía. Esta técnica es también conocida como método de desbordamiento consecutivo.

Para almacenar un registro por hashing con sondeo lineal, la dirección no debe caer fuera del limite del archivo, En lugar de terminar cuando el limite del espacio de dirección se alcanza, se regresa al inicio del espacio y sondeamos desde ahí. Por lo que debe ser posible detectar si la dirección base ha sido encontrada de nuevo, lo cual indica que el archivo esta lleno y no hay espacio para la llave.

Para la búsqueda de un registro por hashing con sondeo lineal, los valores de llave de los registros encontrados en la dirección de origen, y en las direcciones alcanzadas con el sondeo lineal, deberá compararse con el valor de la llave buscada, para determinar si el registro objetivo ha sido localizado o no.



El sondeo lineal puede usarse para cualquier técnica de hashing. Si se emplea sondeo lineal para almacenar registros, también deberá emplearse para recuperarlos.



Doble hashing

En esta técnica se aplica una segunda función hash para combinar la llave original con el resultado del primer hash. El resultado del segundo hash puede situarse dentro

del mismo archivo o en un archivo de sobreflujo independiente; de cualquier modo, será necesario algún método de solución si ocurren colisiones durante el segundo hash.

La ventaja del método de separación de desborde es que reduce la situación de una doble colisión, la cual puede ocurrir con el método de direccionamiento abierto, en el cual un registro que no esta almacenado en su dirección de origen desplazara a otro registro, el que después buscará su dirección de origen. Esto puede evitarse con direccionamiento abierto, simplemente moviendo el registro extraño a otra localidad y almacenando al nuevo registro en la dirección de origen ahora vacía.

Puede ser aplicado como cualquier direccionamiento abierto o técnica de separación de desborde.
Para ambas métodos para la solución de colisiones existen técnicas para mejorar su desempeño como:

1.- Encadenamiento de sinónimos

Una buena manera de mejorar la eficiencia de un archivo que utiliza el calculo de direcciones, sin directorio auxiliar para guiar la recuperación de registros, es el encadenamiento de sinónimos. Mantener una lista ligada de registros, con la misma dirección de origen, no reduce el numero de colisiones, pero reduce los tiempos de acceso para recuperar los registros que no se encuentran en su localidad de origen. El encadenamiento de sinónimos puede emplearse con cualquier técnica de solución de colisiones.
Cuando un registro debe ser recuperado del archivo, solo los sinónimos de la llave objetivo son accesados.

2.- Direccionamiento por cubetas

Otro enfoque para resolver el problema de las colisiones es asignar bloques de espacio (cubetas), que pueden acomodar ocurrencias múltiples de registros, en lugar de asignar celdas individuales a registros. Cuando una cubeta es desbordada, alguna nueva localización deberá ser encontrada para el registro. Los métodos para el problema de sobrecupo son básicamente los mismo que los métodos para resolver colisiones.

COMPARACIÓN ENTRE SONDEO LINEAL Y DOBLE HASHING

De ambos métodos resultan distribuciones diferentes de sinónimos en un archivo relativo. Para aquellos casos en que el factor de carga es bajo (< 0.5), el sondeo lineal tiende a agrupar los sinónimos, mientras que el doble hashing tiende a dispersar los sinónimos mas ampliamente a travéz del espacio de direcciones.

El doble hashing tiende a comportarse casi también como el sondeo lineal con factores de carga pequeños (< 0.5), pero actúa un poco mejor para factores de carga mayores. Con un factor de carga > 80 %, el sondeo lineal por lo general resulta tener un comportamiento terrible, mientras que el doble hashing es bastante tolerable para búsquedas exitosas pero no así en búsquedas no exitosas.

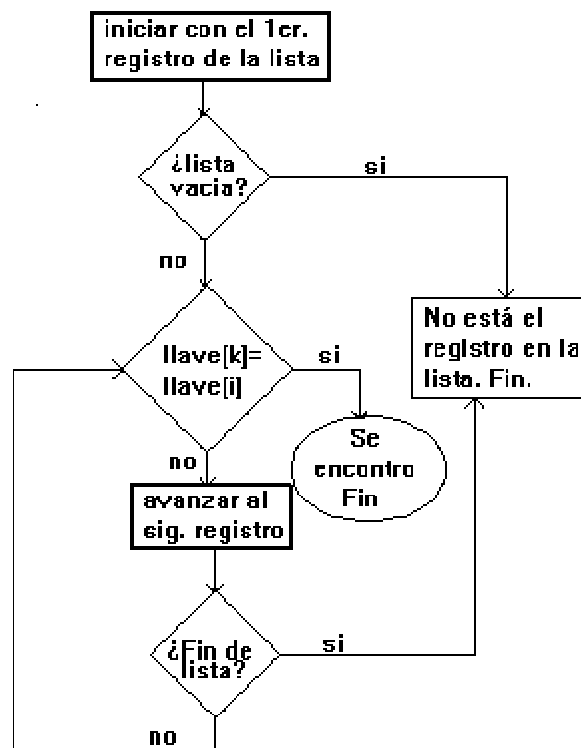
8.2 Búsqueda externa.

Busqueda Secuencial

Definicion:

La búsqueda es el proceso de localizar un registro (elemento) con un valor de llave particular. La búsqueda termina exitosamente cuando se localiza el registro que contenga la llave buscada, o termina sin éxito, cuando se determina que no aparece ningún registro con esa llave.

Búsqueda secuencial, también se le conoce como búsqueda lineal. Supongamos una colección de registros organizados como una lista lineal. El algoritmo básico de búsqueda secuencial consiste en empezar al inicio de la lista e ir a través de cada registro hasta encontrar la llave indicada (k), o hasta al final de la lista.



La situación óptima es que el registro buscado sea el primero en ser examinado. El peor caso es cuando las llaves de todos los n registros son comparados con k (lo que se busca). El caso promedio es n/2 comparaciones.

Este método de búsqueda es muy lento, pero si los datos no están en orden es el único método que puede emplearse para hacer las búsquedas. Si los valores de la llave no son únicos, para encontrar todos los registros con una llave particular, se requiere buscar en toda la lista.

Mejoras en la eficiencia de la búsqueda secuencial

1)Muestreo de acceso

Este método consiste en observar que tan frecuentemente se solicita cada registro y ordenarlos de acuerdo a las probabilidades de acceso detectadas.

2)Movimiento hacia el frente

Este esquema consiste en que la lista de registros se reorganicen dinámicamente. Con este método, cada vez que búsqueda de una llave sea exitosa, el registro correspondiente se mueve a la primera posición de la lista y se recorren una posición hacia abajo los que estaban antes que el.

3)Transposición

Este es otro esquema de reorganización dinámica que consiste en que, cada vez que se lleve a cabo una búsqueda exitosa, el registro correspondiente se intercambia con el anterior. Con este procedimiento, entre mas accesos tenga el registro, mas rápidamente avanzara hacia la primera posición. Comparado con el método de movimiento al frente, el método requiere mas tiempo de actividad para reorganizar al conjunto de registros . Una ventaja de método de transposición es que no permite que el requerimiento aislado de un registro, cambie de posición todo el conjunto de registros. De hecho, un registro debe ganar poco a poco su derecho a alcanzar el inicio de la lista.

4)Ordenamiento

Una forma de reducir el numero de comparaciones esperadas cuando hay una significativa frecuencia de búsqueda sin éxito es la de ordenar los registros en base al valor de la llave. Esta técnica es útil cuando la lista es una lista de excepciones, tales como una lista de decisiones, en cuyo caso la mayoría de las búsquedas no tendrán éxito. Con este método una búsqueda sin éxito termina cuando se encuentra el primer valor de la llave mayor que el buscado, en lugar de la final de la lista.

Programa:

```
#include <conio.h>
#include <iostream.h>
class Lista
{
private:
int Lista[10],N;
public:
Lista()
{
for(int i=0;i<10;i++)
Lista[i]=0;
N=0;
}
void Busqueda(int Elem)
{
if(N!=0)
{
for(int i=0;i<N;i++)
if(Lista[i]==Elem)
{
cout<<"El "<<Elem<<" esta en la Lista"<<endl;
return;
```

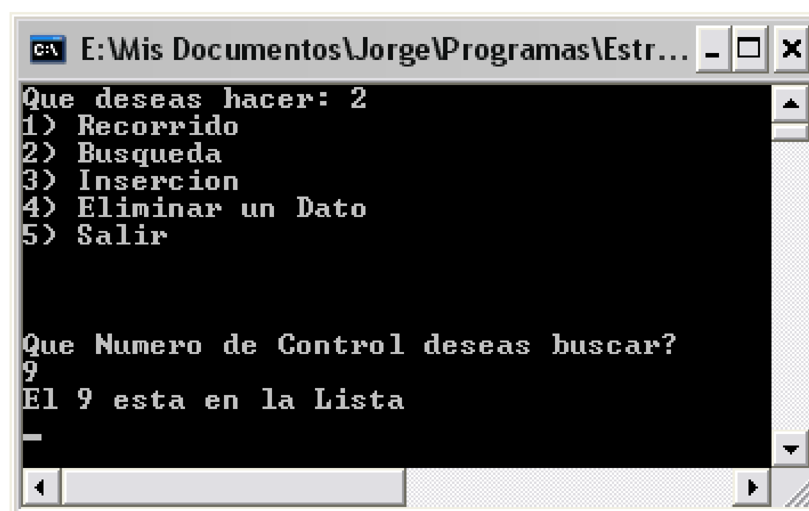
```
}
cout<<"El "<<Elem<<" no esta en la Lista"<<endl;
return;
}
cout<<"Lista Vacia..."<<endl;
return;
}
void Insertar(int Elem)
{
if(N<10)
{
Lista[N]=Elem;
N++;
cout<<"El "<<Elem<<" fue Insertado"<<endl;
return;
}
cout<<"Lista Llena... Imposible Insertar"<<endl;
return;
}
void Eliminar(int Elem)
{
if(N!=0)
{
for(int i=0;i<N;i++)
```

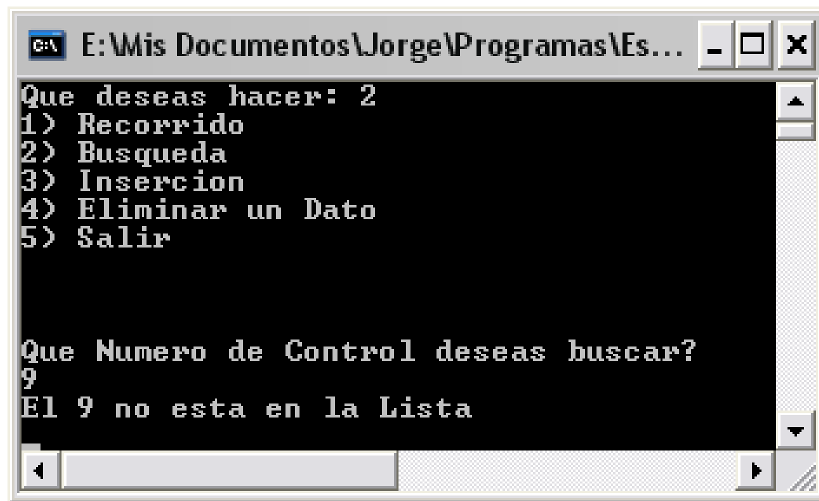
```
if(Lista[i]==Elem)
{
Lista[i]=0;
N--;
return;
}
}
cout<<"Lista Vacia... Imposible Eliminar..."<<endl;
return;
}
void Recorrido()
{
if(N!=0)
{
for(int i=0;i<N;i++)
cout<<Lista[i]<<endl;
}
cout<<"Lista Vacia..."<<endl;
}
}tec;
main()
{
int op=0,res;
while(op!=5)
```

```
{
clrscr();
cout<<"\n1) Recorrido\n2) Busqueda\n3) Insercion\n4) Eliminar un Dato\n5) Salir"<<endl;
gotoxy(1,1);
cout<<"Que deseas hacer: ";
cin>>op;
gotoxy(1,10);
switch (op)
{
case 1:
tec.Recorrido();
break;
case 2:
cout<<"Que Numero de Control deseas buscar?"<<endl;
cin>>res;
tec.Busqueda(res);
break;
case 3:
cout<<"Que Numero Deseas Insertar?"<<endl;
cin>>res;
tec.Insertar(res);
break;
case 4:
cout<<"Que Numero de Control deseas Eliminar?"<<endl;
```

```
cin>>res;

tec.Eliminar(res);

break;

case 5:

cout<<"Salida...";

break;

default:

cout<<"Opcion Erronea"<<endl;

break;

}

getch();

}

}
```

CORRIDA:





Búsqueda Binaria

Definicion:

Se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda. Los prerrequisitos principales para la búsqueda binaria son:

La lista debe estar ordenada en un orden especifíco de acuerdo al valor de la llave.

Debe conocerse el número de registros.

Algoritmo:

Se compara la llave buscada con la llave localizada al centro del arreglo.

Si la llave analizada corresponde a la buscada fin de búsqueda si no.

Si la llave buscada es menor que la analizada repetir proceso en mitad superior, sino en la mitad inferior.

El proceso de partir por la mitad el arreglo se repite hasta encontrar el registro o hasta que el tamaño de la lista restante sea cero , lo cual implica que el valor de la llave buscada no esta en la lista.

El esfuerzo máximo para este algoritmo es de log2n.

El mínimo de 1 y en promedio ½ log2 n.

Programa:

```
#include <conio.h>

#include <iostream.h>

class Lista

{

private:

int Lista[10],N;

public:

Lista()

{

for(int i=0;i<10;i++)

Lista[i]=0;

N=0;

}

void Busqueda(int Elem)

{

if(N!=0)

{

int inicio=0,medio,final=9;

while(inicio<=final)

{
```

```
medio=(inicio+final)/2;
if(Elem==Lista[medio])
{
cout<<"El "<<Elem<<" esta en la Lista"<<endl;
return;
}
else if(Elem<Lista[medio])
{
final=medio-1;
}
else
{
inicio=medio+1;
}
}
cout<<"El "<<Elem<<" no esta en la Lista"<<endl;
return;
}
cout<<"Lista Vacia..."<<endl;
return;
}
void Insertar(int Elem)
{
if(N<10)
```

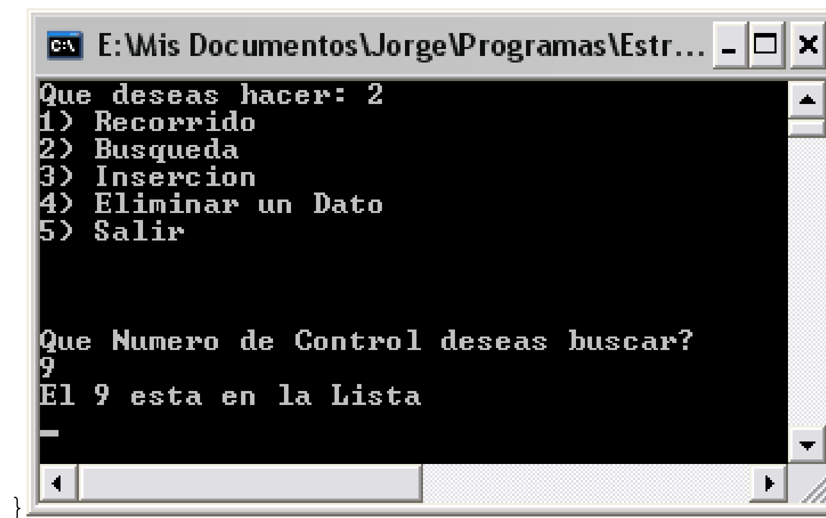
```
{
Lista[N]=Elem;
N++;
cout<<"El "<<Elem<<" fue Insertado"<<endl;
return;
}
cout<<"Lista Llena... Imposible Insertar"<<endl;
return;
}
void Eliminar(int Elem)
{
if(N!=0)
{
for(int i=0;i<N;i++)
if(Lista[i]==Elem)
{
Lista[i]=0;
N--;
return;
}
}
cout<<"Lista Vacia... Imposible Eliminar..."<<endl;
return;
}
```

```
void Recorrido()
{
if(N!=0)
{
for(int i=0;i<N;i++)
cout<<Lista[i]<<endl;
}
cout<<"Lista Vacia..."<<endl;
}
}tec;
main()
{
int op=0,res;
while(op!=5)
{
clrscr();
cout<<"\n1) Recorrido\n2) Busqueda\n3) Insercion\n4) Eliminar un Dato\n5) Salir"<<endl;
gotoxy(1,1);
cout<<"Que deseas hacer: ";
cin>>op;
gotoxy(1,10);
switch (op)
{
case 1:
```

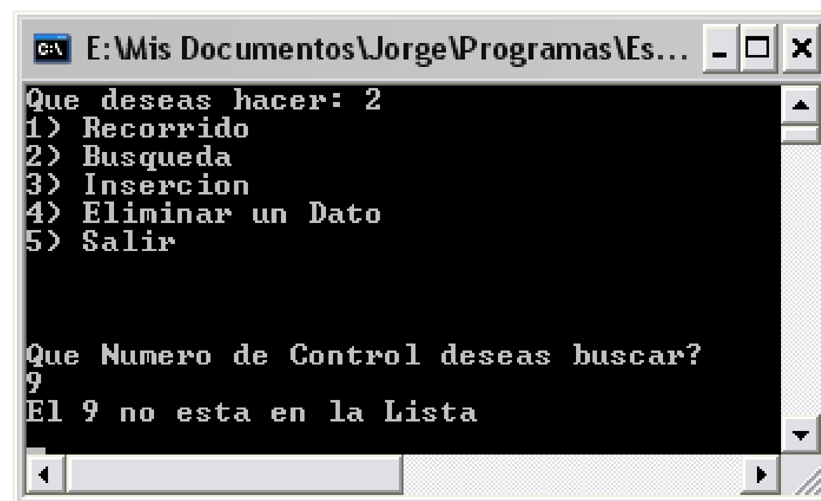
```
tec.Recorrido();
break;
case 2:
cout<<"Que Numero de Control deseas buscar?"<<endl;
cin>>res;
tec.Busqueda(res);
break;
case 3:
cout<<"Que Numero Deseas Insertar?"<<endl;
cin>>res;
tec.Insertar(res);
break;
case 4:
cout<<"Que Numero de Control deseas Eliminar?"<<endl;
cin>>res;
tec.Eliminar(res);
break;
case 5:
cout<<"Salida...";
break;
default:
cout<<"Opcion Erronea"<<endl;
break;
}
```

```
getch();

}
```

CORRIDA



}



Búsqueda por Hash

Definicion:

Hasta ahora las técnicas de localización de registros vistas, emplean un proceso de búsqueda que implica cierto tiempo y esfuerzo. El siguiente método nos permite encontrar directamente el registro buscado.

La idea básica de este método consiste en aplicar una función que traduce un conjunto de posibles valores llave en un rango de direcciones relativas. Un problema potencial encontrado en este proceso, es que tal función no puede ser uno a uno; las direcciones calculadas pueden no ser todas únicas, cuando $R(k_1) = R(k_2)$

Pero : K_1 diferente de K_2 decimos que hay una **colisión**. A dos llaves diferentes que les corresponda la misma dirección relativa se les llama **sinónimos**.

A las técnicas de calculo de direcciones también se les conoce como :

-
- Técnicas de almacenamiento disperso
-
- Técnicas aleatorias
-
- Métodos de transformación de llave - a- dirección
-
- Técnicas de direccionamiento directo
-
- Métodos de tabla Hash
-
- Métodos de Hashing

Pero el término mas usado es el de hashing. Al cálculo que se realiza para obtener una dirección a partir de una llave se le conoce como función hash.

Ventaja

1.
2. Se pueden usar los valores naturales de la llave, puesto que se traducen internamente a direcciones fáciles de localizar
3.
4. Se logra independencia lógica y física, debido a que los valores de las llaves son independientes del espacio de direcciones
5.
6. No se requiere almacenamiento adicional para los índices.

Desventajas

1.
2. No pueden usarse registros de longitud variable
3.
4. El archivo no esta clasificado
5.
6. No permite llaves repetidas
7.
8. Solo permite acceso por una sola llave

Costos

-
- Tiempo de procesamiento requerido para la aplicación de la función hash
-
- Tiempo de procesamiento y los accesos E/S requeridos para solucionar las colisiones.

La eficiencia de una función hash depende de:

1.
2. La distribución de los valores de llave que realmente se usan
3.
4. El numero de valores de llave que realmente están en uso con respecto al tamaño del espacio de direcciones
5.
6. El numero de registros que pueden almacenarse en una dirección dad sin causar una colisión
7.
8. La técnica usada para resolver el problema de las colisiones

Las funciones hash mas comunes son:

-
- Residuo de la división
-
- Medio del cuadrado
-
- Pliegue

HASHING POR RESIDUO DE LA DIVISIÓN

La idea de este método es la de dividir el valor de la llave entre un numero apropiado, y después utilizar el residuo de la división como dirección relativa para el registro (dirección = llave módulo divisor).

Mientras que el valor calculado real de una dirección relativa, dados tanto un valor de llave como el divisor, es directo; la elección del divisor apropiado puede no ser tan simple. Existen varios factores que deben considerarse para seleccionar el divisor:

1.
2. El rango de valores que resultan de la operación "llave % divisor", va desde cero hasta el divisor 1. Luego, el divisor determina el tamaño del espacio de direcciones relativas. Si se sabe que el archivo va a contener por lo menos n registros, entonces tendremos que hacer que divisor > n, suponiendo que solamente un registro puede ser almacenado en una dirección relativa dada.
3.
4. El divisor deberá seleccionarse de tal forma que la probabilidad de colisión sea minimizada. ¿Como escoger este numero? Mediante investigaciones se ha demostrado que los divisores que son números pares tienden a comportase pobremente, especialmente con los conjuntos de valores de llave que son predominantemente impares. Algunas investigaciones sugieren que el divisor deberá ser un numero primo. Sin embargo, otras sugieren que los divisores no primos trabajan también como los divisores primos, siempre y cuando los divisores no primos no contengan ningún factor primo menor de 20. Lo mas común es elegir el número primo mas próximo al total de direcciones.

Ejemplo:

Independientemente de que tan bueno sea el divisor, cuando el espacio de direcciones de un archivo esta completamente lleno, la probabilidad de colisión crece dramáticamente. La saturación de archivo de mide mediante su factor de carga, el cual se define como la relación del numero de registros en el archivo contra el numero de registros que el archivo podría contener si estuviese completamente lleno.

$$\text{Factor de carga} = \frac{\text{número de registro en el archivo}}{\text{máximo número de registro que puede contener el archivo}}$$

Todas las funciones hash comienzan a trabajar probablemente cuando el archivo esta casi lleno. Por lo general el máximo factor de carga que puede tolerarse en un archivo para un rendimiento razonable es de entre el 70 % y 80 %.

HASHING POR MEDIO DEL CUADRADO

En esta técnica, la llave es elevada al cuadrado, después algunos dígitos específicos se extraen de la mitad del resultado para constituir la dirección relativa. Si se desea una dirección de n dígitos, entonces los dígitos se truncan en ambos extremos de la llave elevada al cuadrado, tomando n dígitos intermedios. Las mismas posiciones de n dígitos deben extraerse para cada llave.

Ejemplo:

Utilizando esta función hashing el tamaño del archivo resultante es de 10^n donde n es el numero de dígitos extraídos de los valores de la llave elevada al cuadrado.

HASHING POR PLIEGUE

En esta técnica el valor de la llave es particionada en varias partes, cada una de las cuales

(excepto la ultima) tiene el mismo numero de dígitos que tiene la dirección relativa objetivo. Estas particiones son después plegadas una sobre otra y sumadas. El resultado, es la dirección relativa. Igual que para el método del medio del cuadrado, el tamaño del espacio de direcciones relativas es una potencia de 10.

Ejemplo:

COMPARACIÓN ENTRE LAS FUNCIONES HASH

Aunque alguna otra técnica pueda desempeñarse mejor en situaciones particulares, la técnica del residuo de la división proporciona el mejor desempeño. Ninguna función hash se desempeña siempre mejor que las otras. El método del medio del cuadrado puede aplicarse en archivos con factores de cargas bastantes bajas para dar generalmente un buen desempeño. El método de pliegues puede ser la técnica mas

fácil de calcular pero produce resultados bastante erráticos, a menos que la longitud de la llave se aproximadamente igual a la longitud de la dirección.

Si la distribución de los valores de llaves no es conocida, entonces el método del residuo de la división es preferible. Note que el hashing puede ser aplicado a llaves no numéricas. Las posiciones de ordenamiento de secuencia de los caracteres en un valor de llave pueden ser utilizadas como sus equivalentes "numéricos". Alternativamente, el algoritmo hash actúa sobre las representaciones binarias de los caracteres.

Todas las funciones hash presentadas tienen destinado un espacio de tamaño fijo. Aumentar el tamaño del archivo relativo creado al usar una de estas funciones, implica cambiar la función hash, para que se refiera a un espacio mayor y volver a cargar el nuevo archivo.

MÉTODOS PARA RESOLVER EL PROBLEMA DE LAS COLISIONES

Considere las llaves K_1 y K_2 que son sinónimas para la función hash R. Si K_1 es almacenada primero en el archivo y su dirección es $R(K_1)$, entonces se dice que K_1 esta almacenado en su dirección de origen.

Existen dos métodos básicos para determinar donde debe ser alojado K_2 :

- **Direccionamiento abierto**.- Se encuentra entre dirección de origen para K_2 dentro del archivo.
- **Separación de desborde (Area de desborde).-** Se encuentra una dirección para K_2 fuera del área principal del archivo, en un área especial de desborde, que es utilizada exclusivamente para almacenar registro que no pueden ser asignados en su dirección de origen

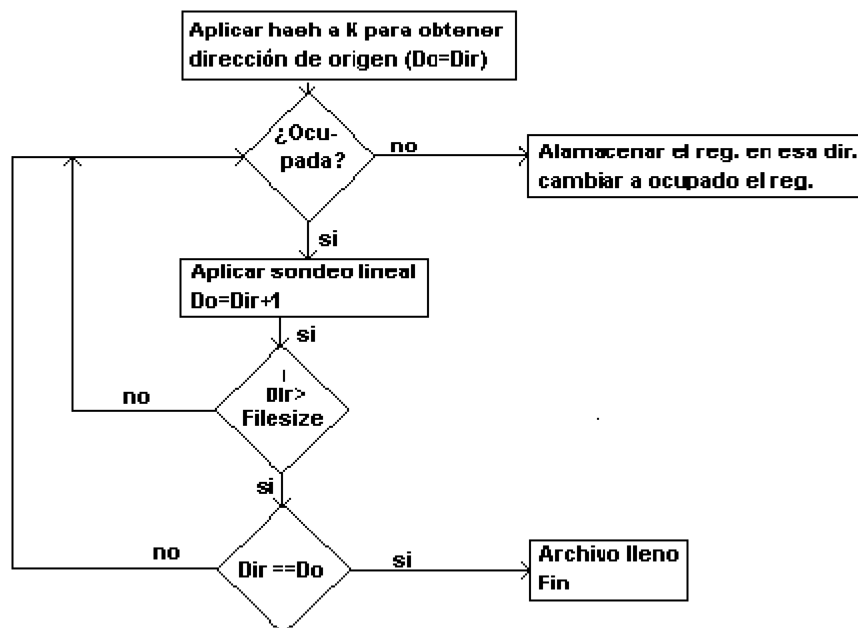
Los métodos mas conocidos para resolver colisiones son:

Sondeo lineal

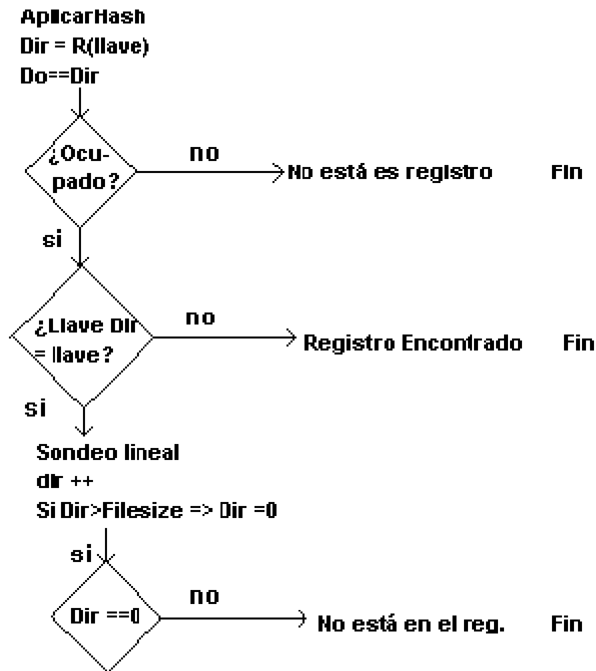
Que es una técnica de direccionamiento abierto. Este es un proceso de búsqueda secuencial desde la dirección de origen para encontrar la siguiente localidad vacía. Esta técnica es también conocida como método de desbordamiento consecutivo.

Para almacenar un registro por hashing con sondeo lineal, la dirección no debe caer fuera del limite del archivo, En lugar de terminar cuando el limite del espacio de dirección se alcanza, se regresa al inicio del espacio y sondeamos desde ahí. Por lo que debe ser posible detectar si la dirección base ha sido encontrada de nuevo, lo cual indica que el archivo esta lleno y no hay espacio para la llave.

Para la búsqueda de un registro por hashing con sondeo lineal, los valores de llave de los registros encontrados en la dirección de origen, y en las direcciones alcanzadas con el sondeo lineal, deberá compararse con el valor de la llave buscada, para determinar si el registro objetivo ha sido localizado o no.



El sondeo lineal puede usarse para cualquier técnica de hashing. Si se emplea sondeo lineal para almacenar registros, también deberá emplearse para recuperarlos.



Doble hashing

En esta técnica se aplica una segunda función hash para combinar la llave original con el resultado del primer hash. El resultado del segundo hash puede situarse dentro del mismo archivo o en un archivo de sobreflujo independiente; de cualquier modo, será necesario algún método de solución si ocurren colisiones durante el segundo hash.

La ventaja del método de separación de desborde es que reduce la situación de una doble colisión, la cual puede ocurrir con el método de direccionamiento abierto, en el cual un registro que no esta almacenado en su dirección de origen desplazara a otro registro, el que después buscará su dirección de origen. Esto puede evitarse con direccionamiento abierto, simplemente moviendo el registro extraño a otra localidad y almacenando al nuevo registro en la dirección de origen ahora vacía.

Puede ser aplicado como cualquier direccionamiento abierto o técnica de separación de desborde.

Para ambas métodos para la solución de colisiones existen técnicas para mejorar su desempeño como:

1.- Encadenamiento de sinónimos

Una buena manera de mejorar la eficiencia de un archivo que utiliza el calculo de direcciones, sin directorio auxiliar para guiar la recuperación de registros, es el encadenamiento de sinónimos. Mantener una lista ligada de registros, con la misma dirección de origen, no reduce el numero de colisiones, pero reduce los tiempos de acceso para recuperar los registros que no se encuentran en su localidad de origen. El encadenamiento de sinónimos puede emplearse con cualquier técnica de solución de colisiones.

Cuando un registro debe ser recuperado del archivo, solo los sinónimos de la llave objetivo son accesados.

2.- Direccionamiento por cubetas

Otro enfoque para resolver el problema de las colisiones es asignar bloques de espacio (cubetas), que pueden acomodar ocurrencias múltiples de registros, en lugar de asignar celdas individuales a registros. Cuando una cubeta es desbordada, alguna nueva localización deberá ser encontrada para el registro. Los métodos para el problema de sobrecupo son básicamente los mismo que los métodos para resolver colisiones.

COMPARACIÓN ENTRE SONDEO LINEAL Y DOBLE HASHING

De ambos métodos resultan distribuciones diferentes de sinónimos en un archivo relativo. Para aquellos casos en que el factor de carga es bajo (< 0.5), el sondeo lineal tiende a agrupar los sinónimos, mientras que el doble hashing tiende a dispersar los sinónimos mas ampliamente a travéz del espacio de direcciones.

El doble hashing tiende a comportarse casi también como el sondeo lineal con factores de carga pequeños (< 0.5), pero actúa un poco mejor para factores de carga mayores. Con un factor de carga > 80 %, el sondeo lineal por lo general resulta tener un comportamiento terrible, mientras que el doble hashing es bastante tolerable para búsquedas exitosas pero no así en búsquedas no exitosas.

8.2.1 Secuencial.

8.2.2 Binaria.

8.2.3 Hash.