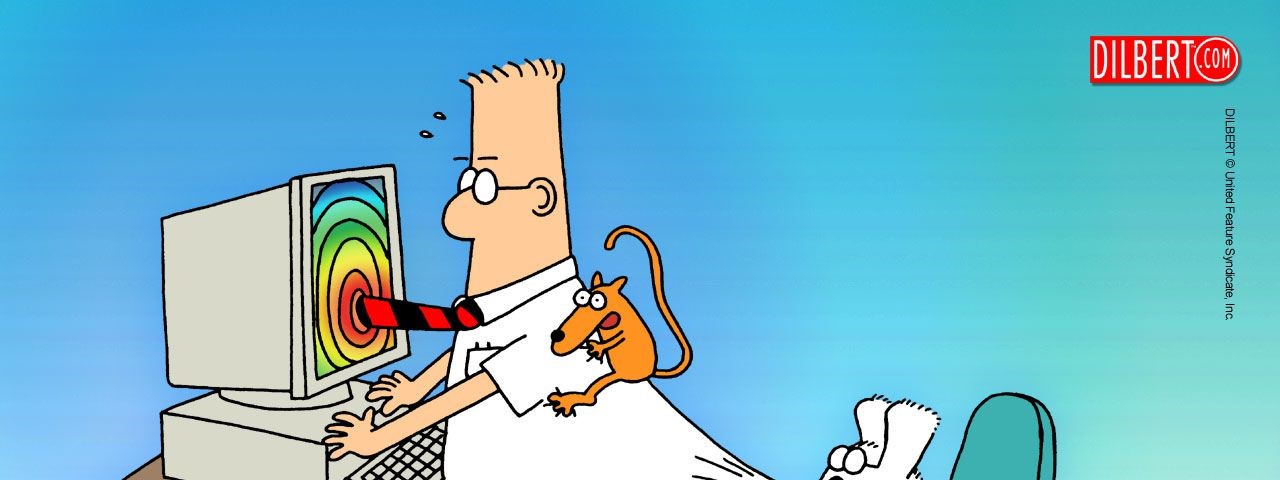


# Apunte del Módulo

Técnicas de Programación



###### Rutinas

Las rutinas son uno de los recursos más valiosos cuando se trabaja en programación ya que permiten

que los programas sean más simples, debido a que el programa principal se compone de diferentes rutinas donde cada una de ellas realiza una tarea determinada.

Una rutina se define como un bloque, formado por un conjunto de instrucciones que realizan una tarea específica y a la cual se la puede llamar desde cualquier parte del programa principal. Además, una rutina puede opcionalmente tener un **valor de retorno** y **parámetros**. El valor de retorno puede entenderse como el resultado de las instrucciones llevadas a cabo por la rutina, por ejemplo si para una rutina llamada *sumar(a, b)* podríamos esperar que su valor de retorno sea la suma de los números a y b. En el caso anterior, a y b son los datos de entrada de la rutina necesarios para realizar los cálculos correspondientes. A estos datos de entrada los denominamos **parámetros** y a las rutinas que reciben parámetros las denominamos **funciones, procedimientos o métodos**, dependiendo del lenguaje de programación.

Ejemplos de rutinas:

**SumarPrecioProductos(precioProducto1, precioProducto2)**

Rutina que realiza la suma de los precios de los productos comprados por un cliente y devuelve el monto total conseguido.

**AplicarDescuento(montoTotal)**

Rutina que a partir de un monto total aplica un descuento de 10% y devuelve el monto total con el descuento aplicado.

Entonces nuestro programa puede hacer uso de dichas rutinas cuando lo necesite. Por ejemplo cuando un cliente realice una compra determinada, podemos llamar a la rutina *sumarPrecioProductos* y cobrarle el monto devuelto por la misma. En el caso que el cliente abonara con un cupón de descuento,

## Desarrollo de programas

### Estructuras de programación

Un programa puede ser escrito utilizando tres tipos de estructuras de control:

1. secuenciales
2. selectivas o de decisión
3. repetitivas

Las Estructuras de Control determinan el orden en que deben ejecutarse las instrucciones de un algoritmo: si serán recorridas una luego de la otra, si habrá que tomar decisiones sobre si ejecutar o no alguna acción o si habrá repeticiones.

#### Estructura secuencial

Es la estructura en donde una acción (instrucción) sigue a otra de manera secuencial. Las tareas se dan de tal forma que la salida de una es la entrada de la que sigue y así en lo sucesivo hasta cumplir con todo el proceso. Esta estructura de control es la más simple, permite que las instrucciones que la constituyen se ejecuten una tras otra en el orden en que se listan. Por ejemplo, considérese el siguiente fragmento de un algoritmo:

En este fragmento se indica que se ejecute la operación 1 y a continuación la operación 2.



*Figura 15: Diagrama de Flujo Secuencial*

#### Estructura alternativa

Estas estructuras de control son de gran utilidad para cuando el algoritmo a desarrollar requiera una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existe un número de posibles alternativas que resultan de la evaluación de una determinada condición.

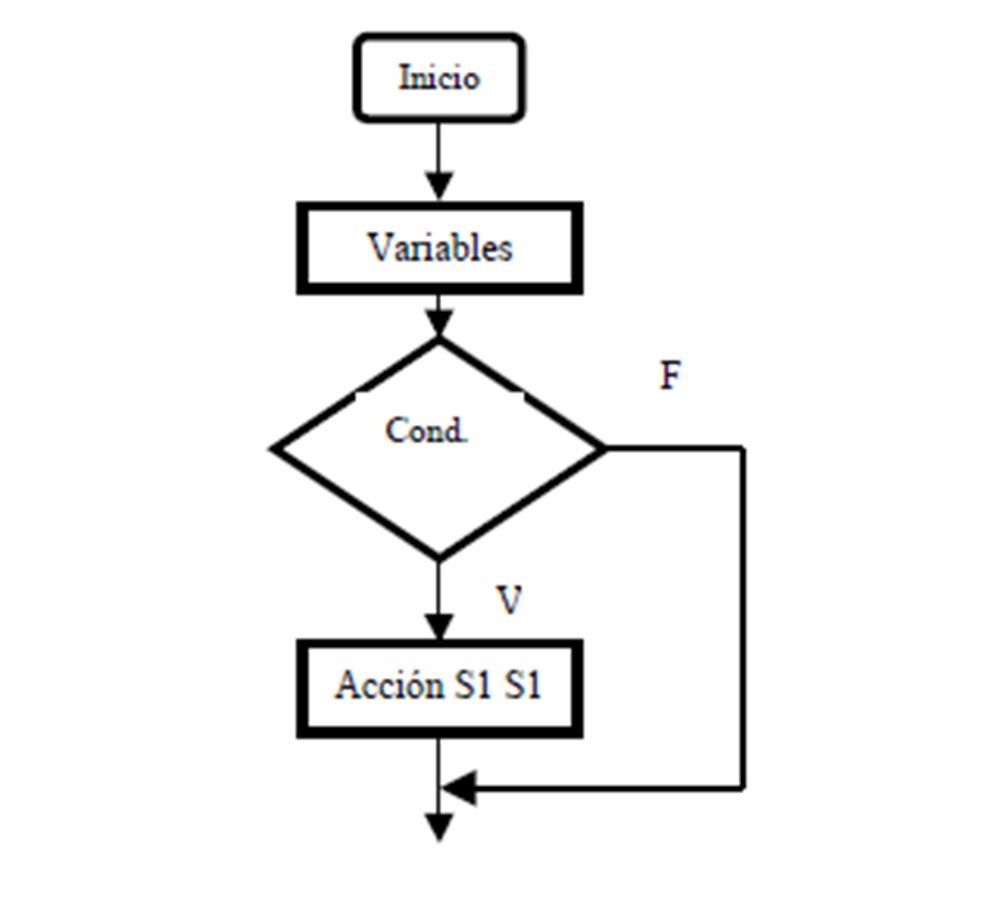
Este tipo de estructuras son utilizadas para tomar decisiones lógicas, es por esto que también se denominan estructuras de decisión o selectivas.

En estas estructuras, se realiza una evaluación de una condición y de acuerdo al resultado, el algoritmo realiza una determinada acción. Las condiciones son especificadas utilizando expresiones lógicas.

Las estructuras selectivas/alternativas pueden ser:

* Simples
* Dobles
* Múltiples

##### ***Alternativa simple (si-entonces/if-then)***



*Figura 16: Diagrama de Flujo de alternativa simple*

La estructura alternativa simple si-entonces (en inglés if-then) lleva a cabo una acción al cumplirse una determinada condición. La selección si-entonces evalúa la condición y:

* Si la condición es verdadera, ejecuta la acción S1
* Si la condición es falsa, no ejecuta nada.

|  |  |
| --- | --- |
| En español:  Si <condición>  Entonces <acción S1> Fin\_si | En Inglés:  If <condición>  Then <acción S1>  End\_if |

Ejemplo:

INICIO

ENTERO edad = 18 SI (edad > 18)

ENTONCES: puede manejar un auto

FIN\_SI

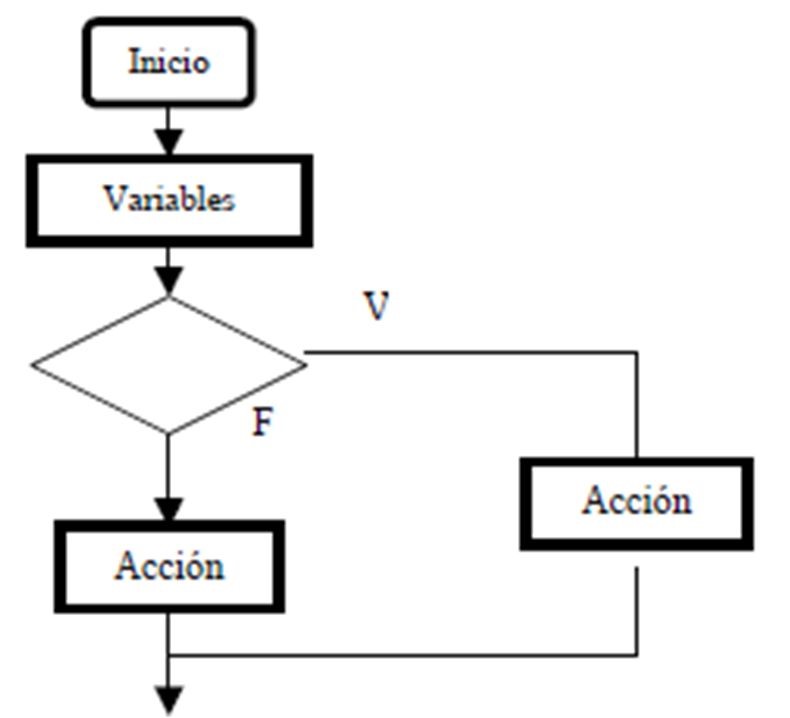
FIN

##### ***Alternativa Doble (si-entonces-sino/if-then-else)***

Existen limitaciones en la estructura anterior, y se necesitará normalmente una estructura que permita elegir dos opciones o alternativas posibles, de acuerdo al cumplimiento o no de una determinada condición:

* Si la condición es verdadera, se ejecuta la acción S1
* Si la condición es falsa, se ejecuta la acción S2

En español:

Si <condición> entonces <acción S1> sino <acción S2>

Fin\_Si

En inglés:

If <condición> then<acción> else<acción S2>

End\_if

*Figura 17: Diagrama de Flujo de*

*alternativa doble*

Ejemplo:

INICIO

BOOLEANO afueraLlueve = verdadero SI (afueraLlueve es verdadero)

ENTONCES:

me quedo viendo películas

SINO: salgo al parque a tomar mates

FIN\_SI

FIN

##### ***Alternativa de Decisión múltiple (según\_sea, caso de/case)***

Se utiliza cuando existen más de dos alternativas para elegir. Esto podría solucionarse por medio de estructuras alternativas simples o dobles, anidadas o en cascada. Sin embargo, se pueden plantear serios problemas de escritura del algoritmo, de comprensión y de legibilidad, si el número de alternativas es grande.

En esta estructura, se evalúa una condición o expresión que puede tomar n valores. Según el valor que la expresión tenga en cada momento se ejecutan las acciones correspondientes al valor.

PSEUDOCÓDIGO:

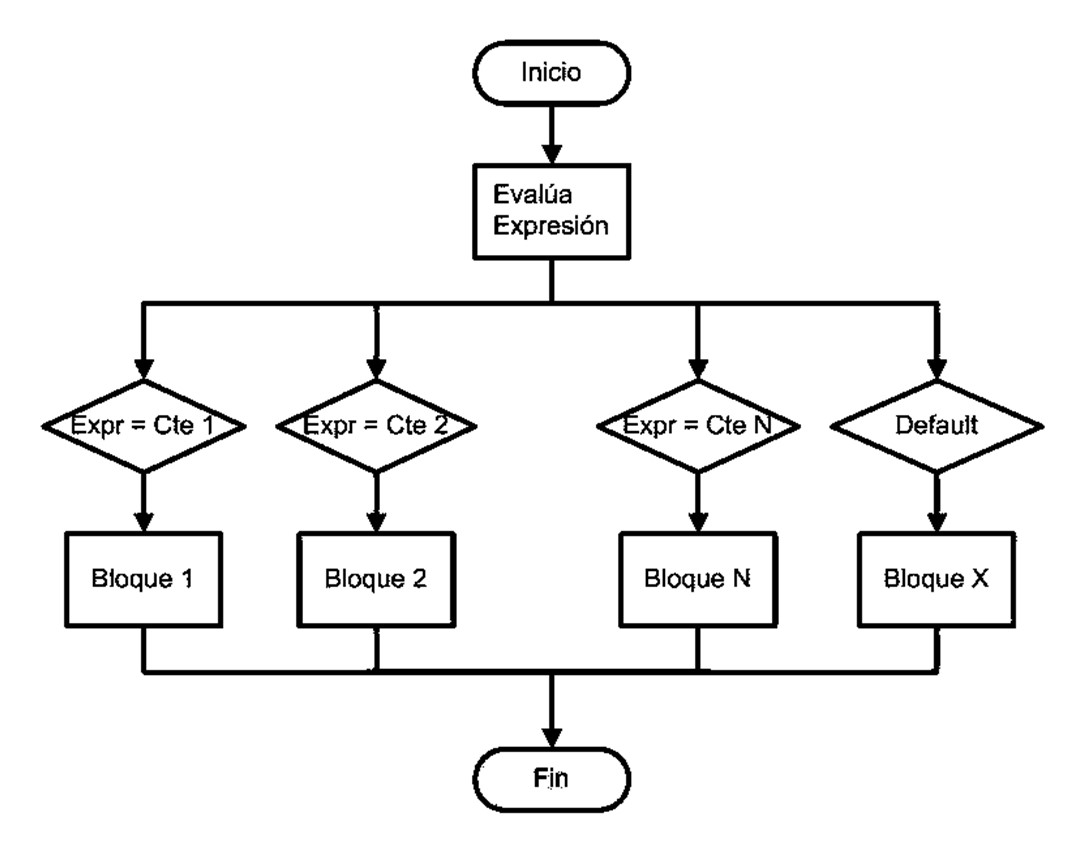
Según sea <expresión>

<Valor1>: <acción1>

<valor2>: <acción2>

.................................

[<otro>: <acciones>] fin según



*Figura 18: Diagrama de Flujo de Decisión Múltiple*

Ejemplo en pseudocódigo:

INICIO

ENTERO posicionDeLlegada = 3

SEGUN SEA posicionDeLlegada

1: entregar medalla de oro

2: entregar medalla de plata 3: entregar medalla de bronce otro: entregar mención especial

FIN

Es importante mencionar que la estructura anterior puede ser escrita usando los condicionales vistos anteriormente de la siguiente forma:

INICIO

ENTERO posicionDeLlegada = 3 SI (posicionDeLlegada = 1)

ENTONCES: entregar medalla de oro

SINO:

SI (posicionDeLlegada = 2)

ENTONCES: entregar medalla de plata

SINO:

SI (posicionDeLlegada = 3)

ENTONCES: entregar medalla de bronce

SINO: entregar mención especial

FIN\_SI

FIN\_SI

FIN\_SI

FIN

Podemos ver que usar condiciones anidadas podemos resolver el mismo problema, pero la estructura resultante es mucho más compleja y difícil de modificar.

#### Estructura repetitiva o iterativa

Durante el proceso de creación de programas, es muy común, encontrarse con que una operación o conjunto de operaciones deben repetirse muchas veces. Para ello es importante conocer las estructuras de algoritmos que permiten repetir una o varias acciones, un número determinado de veces.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan BUCLES. Y cada repetición del bucle se llama iteración.

Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuándo se repite el bucle y cuando deja de repetirse.

Un bucle se denomina también lazo o loop. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalización del bucle no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores principiantes.

Hay distintos tipos de bucles:

* Mientras, en inglés: While
* Hacer Mientras, en inglés: Do While.
* Para, en inglés: For

##### ***Estructura mientras (while, en inglés)***

Esta estructura repetitiva “mientras”, es en la que el cuerpo del bucle se repite siempre que se cumpla una determinada condición.

|  |  |
| --- | --- |
| Diagrama de Flujo    *Figura 19: Diagrama de Flujo de la estructura “mientras”* | Pseudocódigo:    Mientras <cond> hacer <acciones> fin mientras |

Ejemplo:

INICIO

BOOLEANO tanqueLleno = falso MIENTRAS (tanqueLleno == falso)

HACER: llenar tanque

FIN\_MIENTRAS

// el tanque ya está lleno :)

FIN

##### ***Estructura hacer-mientras (do while, en inglés)***

Esta estructura es muy similar a la anterior, sólo que a diferencia del while el contenido del bucle se ejecuta siempre al menos una vez, ya que la evaluación de la condición se encuentra al final. De esta forma garantizamos que las acciones dentro de este bucle sean llevadas a cabo, aunque sea una vez independientemente del valor de la condición.

|  |  |
| --- | --- |
| Diagrama de Flujo    *Figura 20: Diagrama de flujo de “hacer-mientras”* | Pseudocódigo:    Hacer  <acciones>  Mientras <cond>  Fin mientras |

Ejemplo:

INICIO

BOOLEANO llegadaColectivo=false;

HACER: esperar en la parada

MIENTRAS (llegadaColectivo == false) FIN\_MIENTRAS

FIN

##### ***Estructura para (for, en inglés)***

La estructura for es un poco más compleja que las anteriores y nos permite ejecutar un conjunto de acciones para cada elemento de una lista, o para cada paso de un conjunto de elementos. Su implementación depende del lenguaje de programación, pero en términos generales podemos identificar tres componentes: la inicialización, la condición de corte y el incremento.

|  |  |
| --- | --- |
| Diagrama de Flujo    *Figura 21: Diagrama de Flujo de “para”* | Pseudocódigo:    Para (inicialización; condición de corte; incremento)  <acciones>  Fin para |

Ejemplo:

INICIO

PARA (ENTERO RUEDA = 1; RUEDA <= 4; RUEDA++)

inflar\_rueda (RUEDA)

FIN\_PARA

FIN

La ejecución del pseudocódigo anterior dará como resultado las siguientes llamadas a la función inflar():

1. inflar\_rueda (1)
2. inflar\_rueda (2)
3. inflar\_rueda (3)
4. inflar\_rueda (4)

Luego de esto podríamos suponer que hemos inflado las 4 cubiertas del auto y estamos listos para seguir viaje.

#### Recursividad

La recursividad es un elemento muy importante en la solución de algunos problemas de computación. Por definición, un algoritmo recursivo es aquel que utiliza una parte de sí mismo como solución al problema. La otra parte generalmente es la solución trivial, es decir, aquella cuya solución será siempre conocida, es muy fácil de calcular, o es parte de la definición del problema a resolver. Dicha solución sirve como referencia y además permite que el algoritmo tenga una cantidad finita de pasos.

La implementación de estos algoritmos se realiza generalmente en conjunto con una estructura de datos, la pila, en la cual se van almacenando los resultados parciales de cada recursión.

Ventajas

* Algunos problemas son esencialmente recursivos, por lo cual su implementación se facilita mediante un algoritmo de naturaleza recursiva, sin tener que cambiarlo a un método iterativo, por ejemplo.
* En algunas ocasiones el código de un algoritmo recursivo es muy pequeño

Desventajas

* Puede llegar a utilizar grandes cantidades de memoria en un instante, pues implementa una pila cuyo tamaño crece linealmente con el número de recursiones necesarias en el algoritmo. Si los datos en cada paso son muy grandes, podemos requerir grandes cantidades de memoria lo que a veces puede agotar la memoria de la computadora donde está corriendo el programa.

### Estructuras de Datos: Pilas, Colas y Listas

En esta sección veremos estructuras de datos que nos permiten coleccionar elementos. Para poder agregar u obtener elementos de estas estructuras de datos tenemos dos operaciones básicas:

 COLOCAR  OBTENER.

Dependiendo de cómo se realicen las operaciones sobre los elementos de la colección es que definimos pilas, colas y listas.

#### Listas

Una lista (en inglés array) es una secuencia de datos. Los datos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3, etc. En una lista los datos deben ser del mismo tipo. Es importante destacar que la mayoría de las estructuras de datos en los lenguajes de programación son zero-based, es decir que el primer elemento siempre tendrá asignado el número de orden 0, lo que significa que la cantidad de elementos total será igual al número del último elemento más 1. El tipo de elementos almacenados en la lista puede ser cualquier tipo de dato. Normalmente la lista se utiliza para almacenar tipos de datos, tales como cadenas de texto, números enteros o decimales.

Una lista puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de asientos que tiene un colectivo de larga distancia. Cada ítem de una lista se denomina elemento. Una lista tiene definido una longitud, que indica la cantidad de elementos que contiene la misma. Por ejemplo, si tenemos una lista que contiene los meses del año, entonces dicha lista tiene en total 12 elementos, donde el primer elemento “Enero” tiene el número de orden 0 y “Diciembre” el número 11.

Los elementos de una lista se enumeran consecutivamente 0, 1, 2, 3, 4, 5, etc. Estos números se denominan valores índices o subíndice de la lista.

Los índices o subíndices de una lista son números que sirven para identificar unívocamente la posición de cada elemento dentro de la lista. Entonces, si uno quiere acceder a un elemento determinado de la lista, conociendo su posición, es decir su índice, se puede obtener el elemento deseado fácilmente.

Podemos representar una lista de la siguiente forma:

Si consideramos una lista de longitud 6:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| elemento 1 | elemento 2 | elemento 3 | elemento 4 | elemento 5 | elemento 6 |

índice:0 índice:1 índice:2 índice:3 índice:4 índice:5

Ejemplo, volviendo al ejemplo planteado en el párrafo superior, se muestra a continuación una lista que contiene todos los meses del año:

Nombre de la Lista: **mesesDelAño**

Longitud de la Lista: **12** Tipo de Datos: **String**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Enero | Febrero | Marzo | Abril | Mayo | Junio | Julio | Agosto | Septiembre | Octubre | Noviembre | Diciembre |

0 1 2 3 4 5 6 7 8 9 10 11

Si quisiéramos acceder a un elemento determinado de la lista, simplemente debemos conocer cuál es la posición del elemento deseado.

Ejemplo:

INICIO

LISTA mesesDelAño

OBTENER(mesesDelAño, 11) // esto nos devuelve “diciembre”

OBTENER(mesesDelAño, 0) // esto nos devuelve “enero”

OBTENER(mesesDelAño, 7) // esto nos devuelve “agosto”

OBTENER(mesesDelAño, 12) // error: no existe el elemento 12 FIN

Si quisiéramos asignar un valor a una posición determinada de la lista, necesitamos conocer por un lado la posición que queremos asignar, y el elemento que vamos a asignar a dicha posición:

INICIO

LISTA mesesDelAño

COLOCAR(mesesDelAño, 10, “noviembre”) // “noviembre”, en la posición 10

COLOCAR(mesesDelAño, 0, “enero”) // esto asigna “enero”, en la posición 0 COLOCAR(mesesDelAño, 3, “abril”) // esto asigna “abril”, en la posición 3 FIN

#### Pilas

Una pila puede imaginarse como un conjunto de platos colocados uno sobre otro, o como un tubo de papas fritas, o en donde al colocar elementos dentro de él, sólo podemos sacar los últimos colocados. Este tipo de estructuras de datos se denominan LIFO (Last In First Out, del inglés Primero en entrar último en salir). De esta forma los últimos elementos en ser extraídos serán los que estén ubicados en la parte superior de la estrucutra.

Las operaciones básicas de las pilas son dos:

* APILAR (PUSH, colocar un elemento al principio)
* DESAPILAR (POP, obtener el último elemento colocado)

Un ejemplo de pseudocódigo del uso de una pila sería:

INICIO

PILA sillaDelDormitorio

APILAR (sillaDelDormitorio, “buzo”)

APILAR (sillaDelDormitorio, “jeans”)

APILAR (sillaDelDormitorio, “remera”)

DESAPILAR(sillaDelDormitorio) // esto nos devuelve “remera” DESAPILAR(sillaDelDormitorio) // esto nos devuelve “jeans”

DESAPILAR(sillaDelDormitorio) // esto nos devuelve “buzo”

DESAPILAR(sillaDelDormitorio) // error: la pila está vacía FIN

#### Colas

Esta estructura de datos se caracterizada por ser una secuencia de elementos en la que la operación de inserción:

 ENCOLAR (PUSH, agregar) se realiza por un extremo y,  DESENCOLAR (POP, extraer) por el otro.

También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.



*Figura 22: Ejemplo de una cola de espera*

Esto representa la idea que tenemos de cola en la vida real. La cola para subir al colectivo está compuesta de elementos (personas), que dispone de dos extremos comienzo y fin. Por el comienzo se extraerá un elemento cuando haya comprado el pasaje para su viaje, y si llega una nueva persona con intención de viajar, tendrá que colocarse al final y esperar que todos los elementos situados antes que él abandonen la cola.

Existen otros tipos de colas más sofisticados como colas con prioridad en las que algunos elementos podrán abandonar la cola antes que otros independientemente de que hayan sido agregados luego, usando como factor de decisión cierto valor de prioridad. Un ejemplo puede darse en las colas para pagar los impuestos, en las que personas mayores pueden ser atendidos antes (abandonar la cola antes) debido a su condición prioritaria.

Un ejemplo de pseudocódigo del uso de una cola sin prioridad sería:

INICIO

COLA cajaDelSupermercado

ENCOLAR (cajaDelSupermercado, “Judith”)

ENCOLAR (cajaDelSupermercado, “Candelaria”)

ENCOLAR (cajaDelSupermercado, “Joaquin”)

DESENCOLAR(cajaDelSupermercado) // esto nos devuelve “Judith”

DESENCOLAR(cajaDelSupermercado) // esto nos devuelve “Candelaria”

DESENCOLAR(cajaDelSupermercado) // esto nos devuelve “Joaquin”

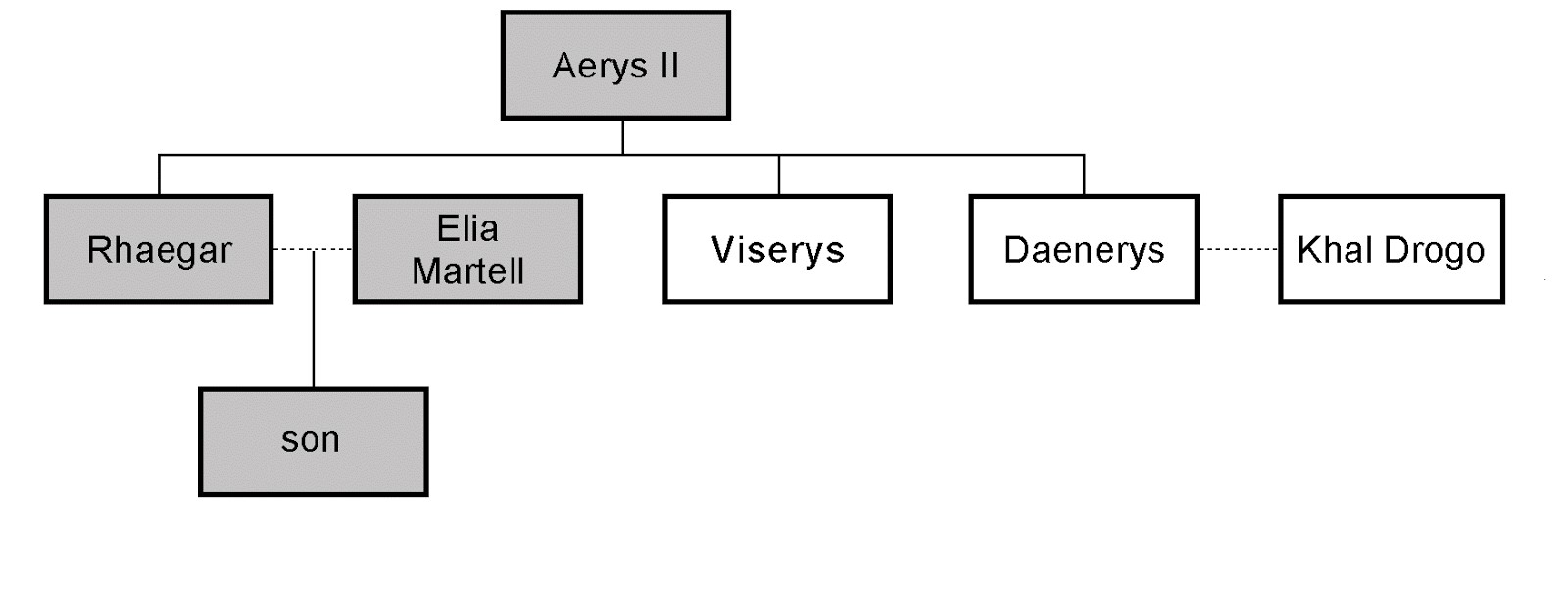
DESENCOLAR(cajaDelSupermercado) // error: la cola está vacía FIN

### Árboles

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras **no lineales** a diferencia de las listas, colas y pilas vistas anteriormente, que constituyen estructuras lineales.

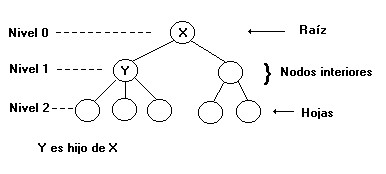
Los árboles son muy utilizados en informática como un método eficiente para búsquedas grandes y complejas, listas dinámicas y aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de estas aplicaciones, los árboles se utilizan en diseño de compiladores, proceso de texto y algoritmos de búsqueda.

Intuitivamente el concepto de aŕ bol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sı ́a través de ramas. El árbol genealógico es el ejemplo tıṕ ico más representativo del concepto de árbol general.



*Figura 23: Un árbol genealógico como ejemplo de la estructura de datos*

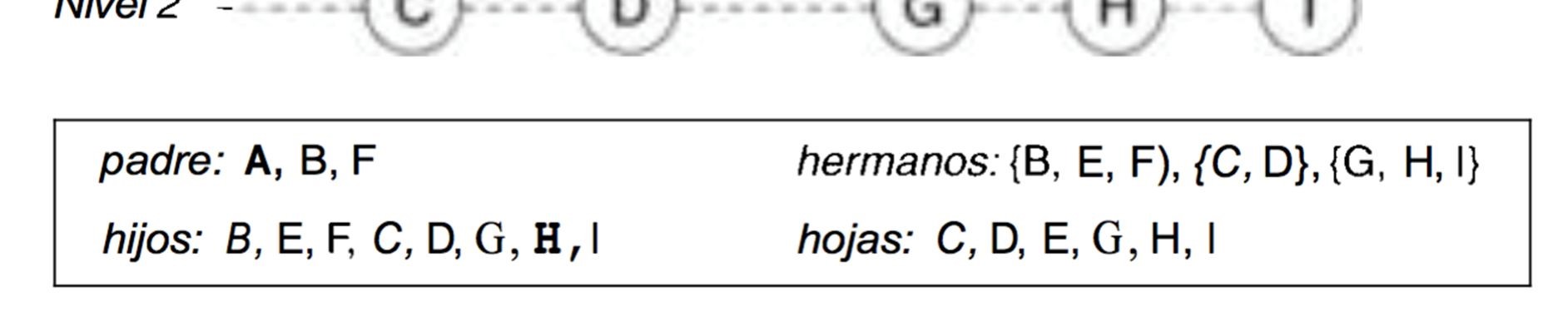
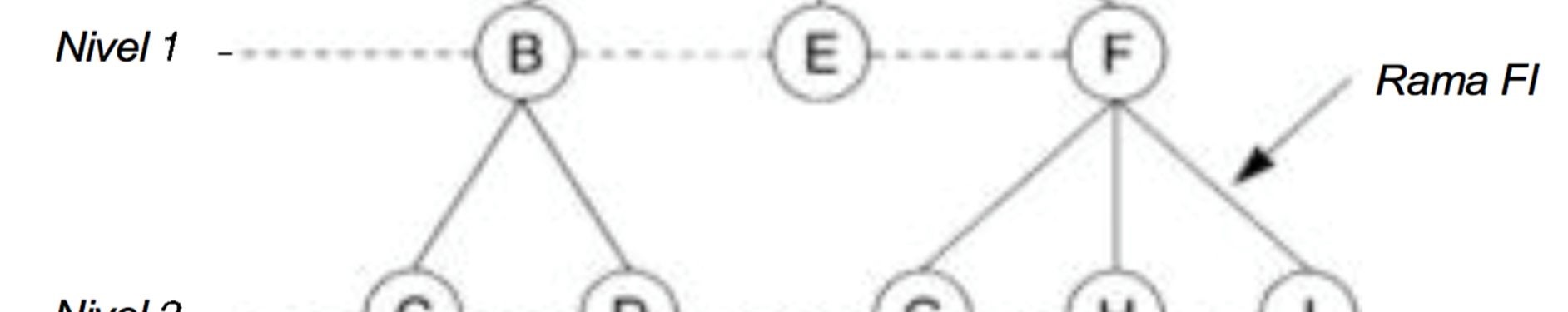
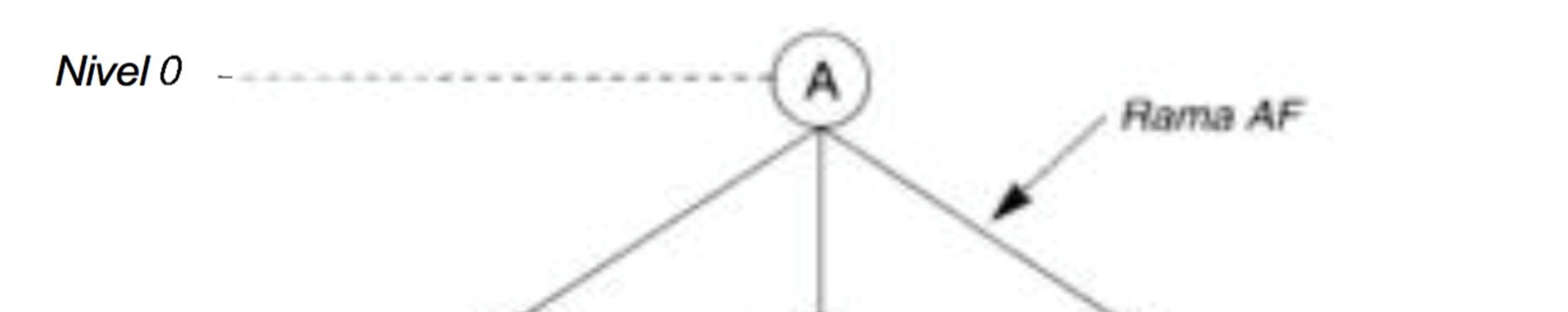
Un árbol consta de un conjunto finito de elementos, denominados **nodos** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo. Si un árbol no está́ vacío, entonces el primer nodo se llama **raíz**.



*Figura 24: Componentes de un árbol*

Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores, los cuales se llaman **hijos**. Dos o más nodos con el mismo padre se llaman **hermanos**. Los nodos sin hijos se denominan nodos **hoja**.

El **nivel** de un nodo es su distancia al raıź . El raıź tiene una distancia cero de sı ́misma, por lo que se dice que el raıź está en el nivel 0. Los hijos del raıź estań en el nivel 1, sus hijos están en el nivel 2 y así ́ sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y hermanos. Los hermanos estań siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos.

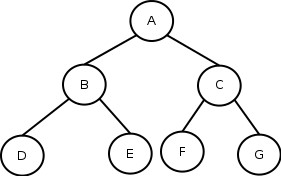


*Figura 25: Niveles de profundidad de los árboles*

Un árbol se divide en **subárboles**. Un subárbol es cualquier estructura conectada por debajo del raıź . Cada nodo de un árbol es la raıź de un subárbol que se define por el nodo y todos los descendientes de ese nodo. El primer nodo de un subaŕ bol se conoce como el **raız del sub́ árbol** y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subaŕ boles.

#### Árboles binarios

Un **árbol binario** es un tipo particular de aŕ bol en el que ningún nodo puede tener más de dos subárboles. En un aŕ bol binario, cada nodo puede tener, cero, uno o dos hijos (subárboles). Se conoce al nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho.



*Figura 26: Un árbol binario como tipo particular de árbol*

### Algoritmos Fundamentales

#### Algoritmos de Ordenación

Los algoritmos de ordenación sirven para dar un orden determinado a los elementos de una lista. Este procedimiento de ordenación, mediante el cual se disponen los elementos del array en un orden especificado, tal como orden alfabético u orden numérico, es una tarea muy usual en la mayoría de los programas.

Un diccionario es un ejemplo de una lista ordenada alfabéticamente, y una agenda telefónica o lista de cuentas de un banco es un ejemplo de una lista ordenada numéricamente.

El orden de clasificación u ordenación puede ser ascendente (de menor a mayor) o descendente (de mayor a menor), por lo tanto, debe existir una función o característica de los elementos que determine su precedencia. Los ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

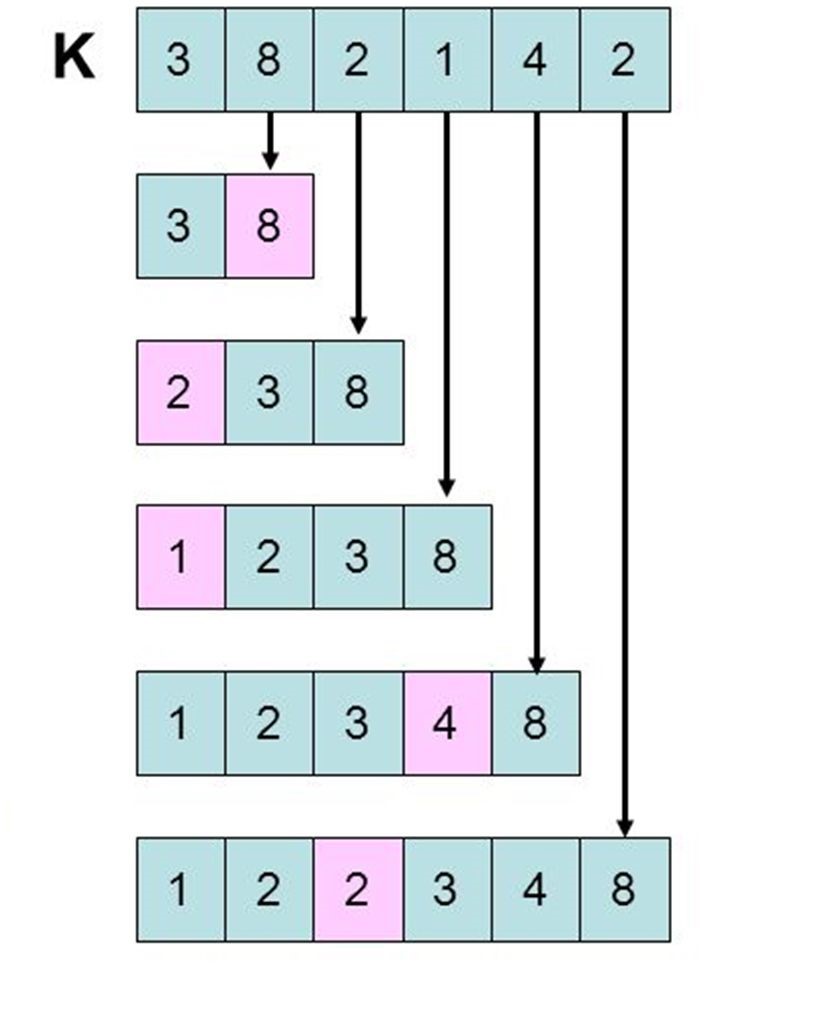
Existen numerosos algoritmos de ordenación de listas: inserción, burbuja, selección, rápido (quick sort), fusión (merge), montículo (heap), shell, etc. Las diferencias entre estos algoritmos se basan en su eficiencia y en su **orden de complejidad** (es una medida de la dificultad computacional de resolver un problema). A continuación, describiremos algunos de ellos.

#### Ordenamiento por inserción

Este algoritmo es el más sencillo de comprender ya que es una representación natural de cómo aplicaríamos el orden a un conjunto de elementos. Supongamos que tenemos un mazo de cartas desordenadas, este algoritmo propone ir tomando las cartas de a una y luego ir colocandolas en la posición correcta con respecto a las anteriores ya ordenadas.

En términos generales, inicialmente se tiene un solo elemento, que por defecto es un conjunto ordenado. Después, cuando hay

k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.



*Figura 27: Ejemplo gráfico del algoritmo de ordenamiento por inserción*

El pseudocódigo para este algoritmo es el siguiente:

INICIO insercion (A: lista de elementos)

PARA (ENTERO i = 1; i < longitud(A); i++) :

ENTERO valor = A[i]

ENTERO j = i-1

MIENTRAS (j >= 0 && A[j] > valor)

HACER:

A[j+1] = A[j] j--

FIN\_MIENTRAS

A[j+1] = valor

FIN\_PARA

FIN

#### Algoritmo de la burbuja

La ordenación por burbuja es uno de los métodos más fáciles de ordenación, ya que el algoritmo de ordenación utilizado es muy simple.

Este algoritmo consiste en comparar cada elemento de la lista con el siguiente (por parejas), si no están en el orden correcto, se intercambian entre sí sus valores. El valor más pequeño flota hasta el principio de la lista como si fuera una burbuja en un vaso de gaseosa.

A continuación, se muestra un ejemplo gráfico de este algoritmo, considerando la siguiente lista inicial:

|  |  |
| --- | --- |
|  | **No ordenado**     1. < 3, ok 2. < 4, ok 3. < 5, ok 4. > 1, intercambiar      1. < 3, ok 2. < 4, ok 3. > 1, intercambiar      1. < 3, ok 2. > 1, intercambiar     2 > 1, intercambiar    **Ordenado** |

*Figura 28: Pasos para ordenar una lista con el método de la burbuja*

El pseudocódigo para este algoritmo es el siguiente:

INICIO burbuja (A: lista de elementos) n = longitud(A)

HACER: intercambiado = falso

PARA (ENTERO i = 1; i < n; i++)

// si este par no está ordenado

SI (A[i-1] > A[i]) ENTONCES:

// los intercambiamos y recordamos que algo ha cambiado

ENTERO aux = A[i-1] A[i-1] = A[i]

A[i] = aux

intercambiado = verdadero

FIN\_SI

FIN\_PARA

MIENTRAS (intercambiado == verdadero)

FIN

Es importante notar que la recorrida completa de la lista (determinada por la sentencia PARA del pseudocódigo) será ejecutada hasta que intercambiado deje de ser verdadero, es decir que seguiremos recorriendo la lista e intercambiando elementos desordenados hasta que no encontremos ninguno más fuera de orden.

##### Ordenamiento por selección

El algoritmo de ordenamiento por selección es similar al método de la burbuja y funciona de la siguiente manera: inicialmente se recorre toda la lista buscando el menor de todos los elementos, una vez terminada la recorrida el menor elemento se coloca al inicio de la lista recorrida. En la siguiente iteración se recorre nuevamente la lista pero comenzando en el segundo elemento (ya que al haber insertado el menor encontrado al inicio ya lo consideramos ordenado). El procedimiento continúa hasta que el último elemento recorrido es el menor de su subconjunto.

Una desventaja de este algoritmo con respecto a los anteriores mencionados es que no mejora su rendimiento cuando los datos ya están ordenados o parcialmente ordenados debido a que necesariamente recorre la lista en busca del menor de los datos aun cuando el primero de ellos ya es el menor a encontrar.

El pseudocódigo de este algoritmo es muy similar al de la burbuja:

INICIO seleccion (A : lista de elementos ) n = longitud(A)

PARA (ENTERO i = 1; i < n - 1; i++)

ENTERO minimo = i

PARA (ENTERO j = i+1; j < n; j++)

// si este par no está ordenado SI (A[j] < A[minimo]) ENTONCES: // encontramos un nuevo mínimo minimo = j

FIN\_SI

// intercambiamos el actual con el mínimo encontrado

ENTERO aux = A[minimo] A[minimo] = A[j]

A[j] = aux

FIN\_PARA

FIN\_PARA

FIN

##### Algoritmo quick-sort

Esta es la técnica de ordenamiento más rápida conocida, 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). Tiene la propiedad de trabajar mejor para elementos de entrada desordenados completamente que para elementos semiordenados. Esta situación es precisamente la opuesta al ordenamiento de burbuja o al de selección antes mencionados.

Este tipo de algoritmos se basa en la técnica "divide y vencerás", lo que supone que es más rápido y fácil ordenar dos arreglos o listas de datos pequeños, que un arreglo o lista más grande.

El algoritmo trabaja de la siguiente forma:

* 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. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. 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, algunas alternativas son:

* Tomar un elemento cualquiera como pivote, tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido.
* Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. 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 del medio como pivote.

### Algoritmos de Búsqueda

La búsqueda de un elemento dado en una lista es una aplicación muy usual en el desarrollo de programas. Dos algoritmos típicos que realizan esta tarea son la búsqueda **secuencial** o en serie y la búsqueda **binaria** o dicotómica. La búsqueda secuencial es el método utilizado para listas no ordenadas, mientras que la búsqueda binaria se utiliza en listas que ya están ordenados.

#### Búsqueda secuencial

Este algoritmo busca el elemento dado, recorriendo secuencialmente la lista desde un elemento al siguiente, comenzando en la primera posición de la lista y se detiene cuando encuentra el elemento buscado o bien se alcanza el final de la lista sin haberlo encontrado.

Por consiguiente, el algoritmo debe comprobar primero el elemento almacenado en la primera posición de la lista, a continuación, el segundo elemento y así sucesivamente, hasta que se encuentra el elemento buscado o se termina el recorrido de la lista. Esta tarea repetitiva se realiza con bucles, en nuestro caso con el bucle Para (en inglés, for).

Consideremos que tenemos una lista de alumnos de un curso de Programación y queremos saber si el alumno ‘Pedro Lopez’, se encuentra cursando el mismo, entonces debemos recorrer toda la lista y buscar el nombre ‘Pedro Lopez’, e indicar si se encontró o no el alumno buscado.

INICIO busquedaSecuencial (L: lista de alumnos, a: alumno buscado )

ENTERO n = longitud(L)

BOOLEAN seEncontró= falso;

// recorremos la lista, revisando cada elemento de la misma, para ver // si es el alumno a.

PARA (ENTERO i = 1; i < n - 1; i++)

// comparamos el alumno de la posición actual con el alumno buscado: a

SI (L[i] == a) ENTONCES:

// encontramos el alumno buscado seEncontró = verdadero;

FIN\_SI

// si nunca se cumple L[i] == a, entonces la variable que indica si se

// encontró o no el alumno: seEncontró, quedará valiendo falso.

FIN\_PARA

FIN

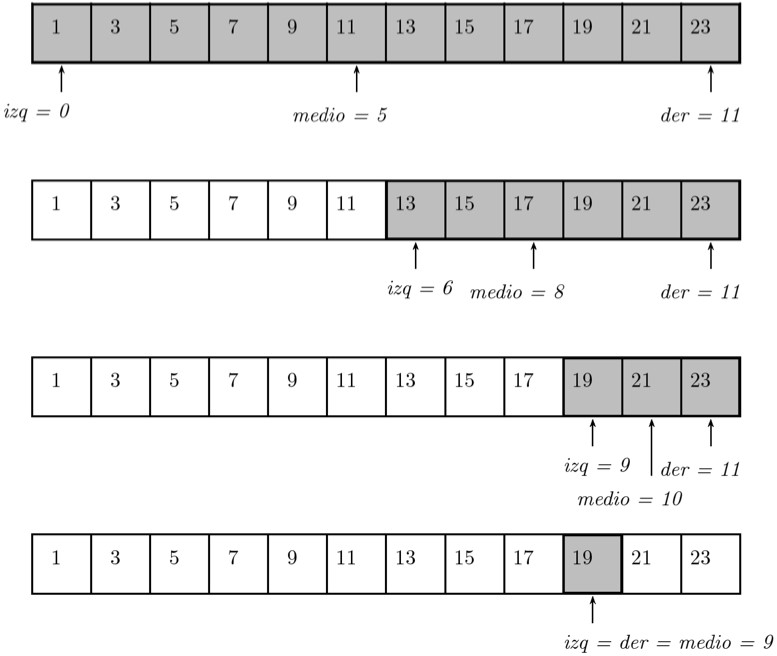
##### Búsqueda Binaria

Este algoritmo se utiliza cuando disponemos de una lista ordenada, lo que nos permite facilitar la búsqueda, ya que podemos ir disminuyendo el espacio de búsqueda a segmentos menores a la lista original y completa.

La idea es no buscar en aquellos segmentos de la lista donde sabemos que el valor seguro que no puede estar, considerando que la lista esta ordenada.

Pensemos en el ejemplo anterior de la lista de alumnos del curso de Programación, si tenemos la lista ordenada alfabéticamente por Apellido, podemos comenzar la búsqueda considerando la lista completa y evaluar un valor central de la misma, es probable que ese valor central no sea el buscado, pero podemos ver si ese valor central es mayor o menor al alumno buscado. Si nuestro alumno buscado se llama: ‘Lopez Pedro’, y el alumno que corresponde a la posición central de la lista es: ‘Martinez Sofia’, entonces sabemos que como la lista esta ordenada alfabéticamente por apellido, ‘Lopez Pedro’, efectivamente tiene que estar en el segmento de la lista que es la primera mitad de la misma, es decir que hemos reducido el espacio de búsqueda a la mitad, lo cual hace que encontremos el valor más rápido que si lo buscaríamos en toda la lista, recorriendo todos los elementos. Si proseguimos con este procedimiento y continuamos buscando el valor central de cada segmento obtenido, podemos ir reduciendo cada vez más el espacio de búsqueda hasta llegar al elemento buscado, si es que existe en la lista.

Para explicar el algoritmo de búsqueda binaria, consideremos que queremos ver si se encuentra en una lista el número 19, a partir de una lista que contiene 12 números ordenados de menor a mayor, como se muestra a continuación:



*Figura 29: Búsqueda binaria sobre una lista de enteros* Para poder realizar la búsqueda binaria debemos:

* Primero, conocer cuál es el valor del índice izquierdo, derecho y del medio, de la siguiente forma:
  + Índice izq = 0; //sabemos que las listas comienzan con índices enumerados desde 0.
  + Índice der = Longitud de la lista inicial -1; // en este caso la longitud es 12, o sea, que el índice derecho valdrá 11.
  + Índice medio = (izq + der) / 2 // en este caso, (0 + 11)/2, considerando solo la parte entera de la división valdrá 5.
* A partir de la definición de estos índices, el siguiente paso es preguntar si en la posición del medio se encuentra el elemento buscado, es decir si Lista(medio)==19
* Si Lista(medio)==19, devuelve verdadero, entonces la búsqueda finaliza rápidamente.
* Si Lista(medio)==19, devuelve falso, entonces debemos preguntar si el valor de la lista en la posición medio es mayor o menor al valor buscado, para así saber si el segmento que nos interesa es del medio hacia la izquierda o del medio hacia la derecha. En este caso: Lista(medio) es menor a 19. Entonces el segmento que nos interesa de la lista es del medio (sin incluir, porque ya evaluamos y el medio no es igual a 19) hacia la derecha.
* El siguiente paso es volver a realizar el procedimiento descripto, pero considerando sólo el segmento que comienza en el medio hacia la derecha: son los mismos pasos pero para una nueva lista que es un segmento de la lista original. Entonces:
  + Índice izq = 6; o Índice der = 11 o Índice medio = (izq + der) / 2 // en este caso, (6+ 11)/2, considerando solo la parte entera de la división valdrá 8.

Como se ve este algoritmo de búsqueda binaria, a diferencia del algoritmo de búsqueda secuencial, no recorre toda la lista, sino que acorta la lista en segmentos más pequeños sucesivamente, esto es muy ventajoso en el caso de tener listas con gran cantidad de elementos, es decir con millones de valores, en estos casos realizar una búsqueda secuencial lleva mucho tiempo y si la lista ya se encuentra ordenada es mucho más eficiente realizar una búsqueda binaria que secuencial.

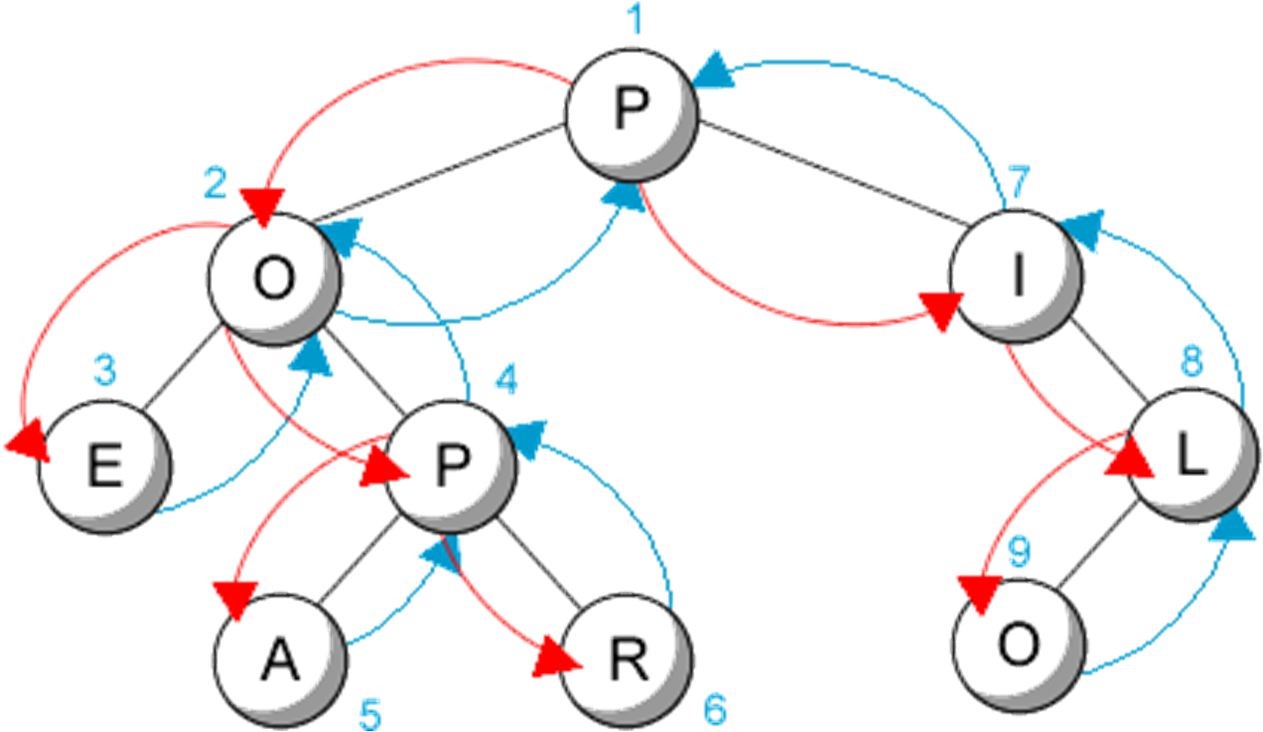
### Algoritmos de Recorrido

Para visualizar o consultar los datos almacenados en un aŕ bol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario de las listas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el raıź viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayorıá de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

Un recorrido de un árbol binario requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, **profundidad** y **anchura**.

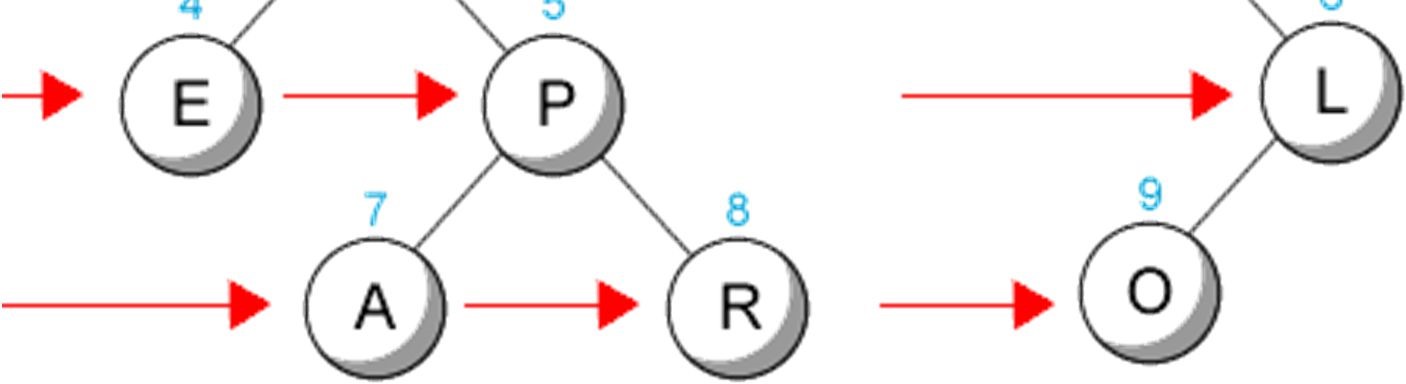
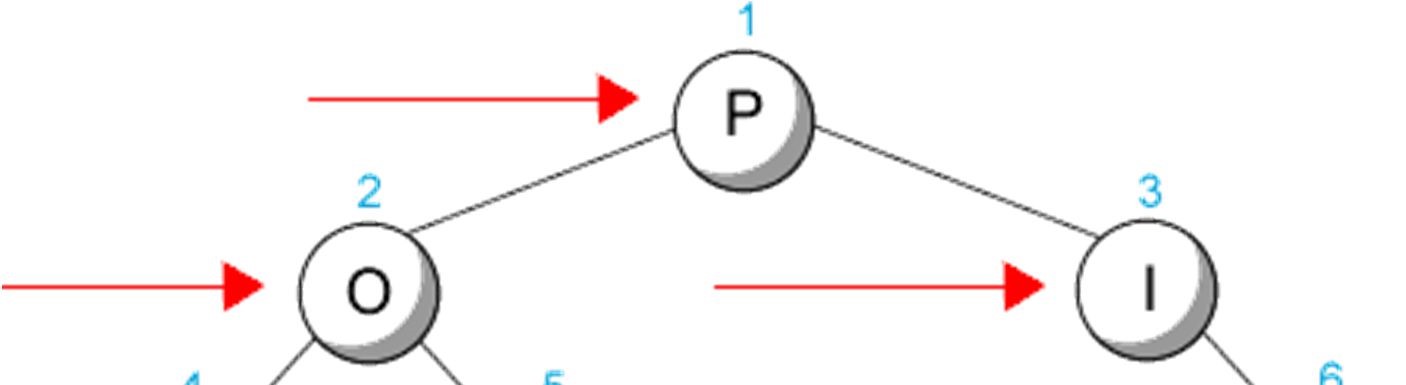
En el **recorrido en profundidad**, el proceso exige un camino desde el nodo raıź a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo. Para saber cómo regresarnos, vamos guardando los nodos visitados en una estructura de **pila**. Es por esto que se acostumbra programar esta búsqueda de forma recursiva, con lo que el manejo de la pila lo realiza el lenguaje de programación utilizado.

Haciendo un recorrido en profundidad recorreríamos los nodos en el siguiente orden:



*Figura 30: Recorrido en profundidad*

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raıź a todos sus hijos, a continuación, a los hijos de sus hijos y así ́ sucesivamente hasta que todos los nodos han sido procesados. En otras palabras, en el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel. Para poder saber qué vértices visitar, utilizamos una **cola**.



*Figura 31: Recorrido en anchura*

## 