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.

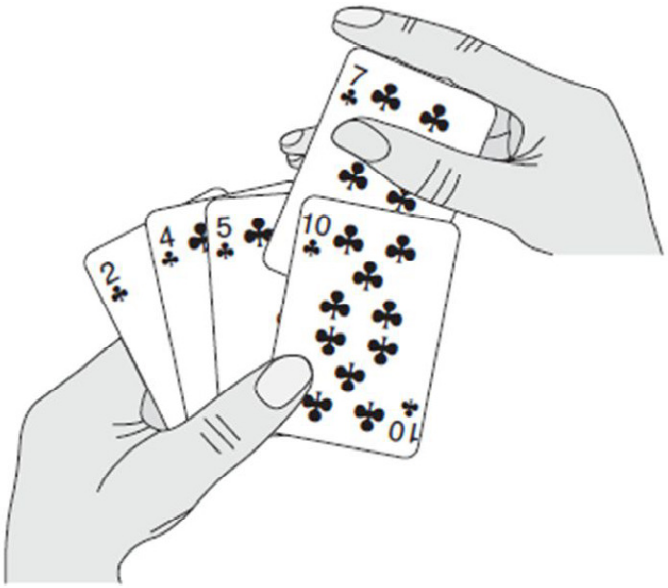
Un diccionario es un ejemplo de una lista ordenada alfabéticamente, y una agenda telefónica o lista de cuentas de un banco es un ejemplo de una lista ordenada numéricamente.

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.

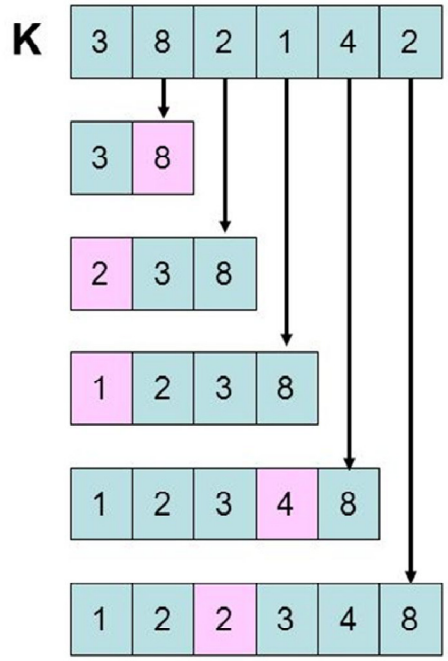
Existen numerosos algoritmos de ordenación de listas: inserción, burbuja, selección, rápido (quick sort), fusión (merge), montículo (heap), shell, etc. Las diferencias entre estos algoritmos se basan en su eficiencia y en su **orden de complejidad** (es una medida de la dificultad computacional de resolver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



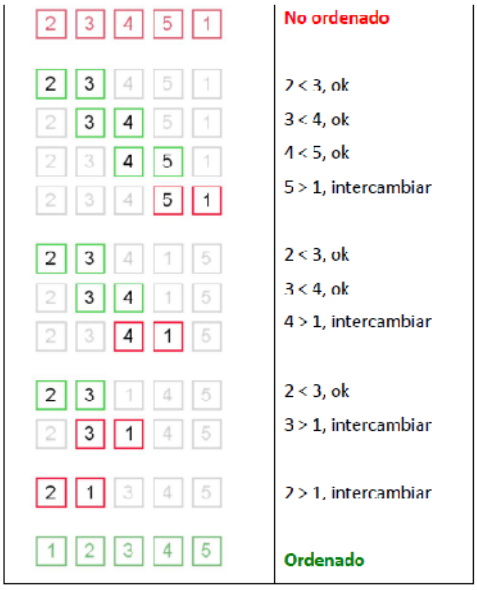
Ejemplo gráfico del algoritmo de ordenamiento por inserción

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:



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).

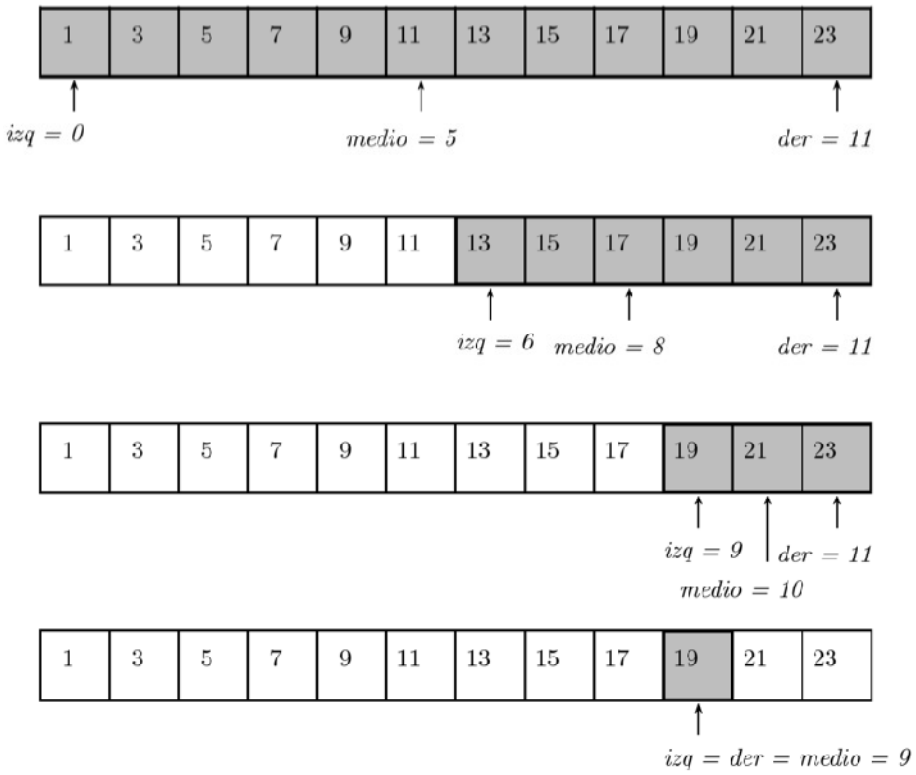
Consideremos que tenemos una lista de alumnos de un curso de Programación y queremos saber si el alumno 'Pedro Lopez', se encuentra cursando el mismo, entonces debemos recorrer toda la lista y buscar el nombre 'Pedro Lopez', e indicar si se encontró o no el alumno buscado.

```
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:



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: o 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;
Índice der = 11
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

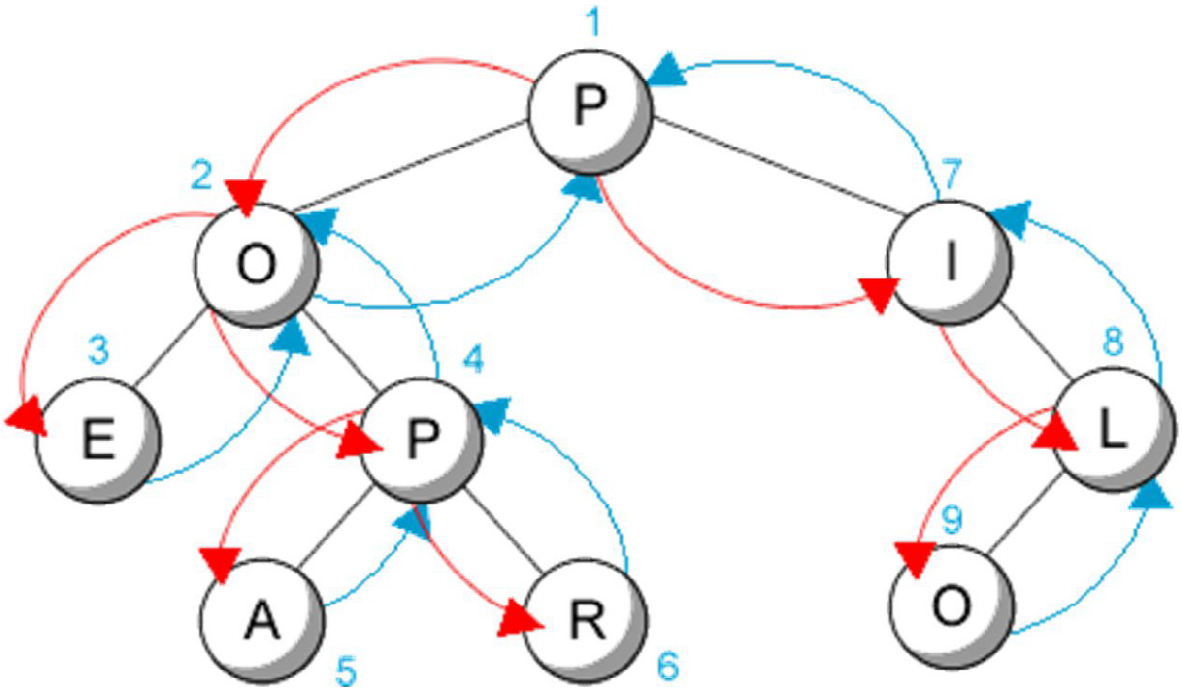
Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario de las listas, los árboles binarios no tienen realmente un primer valor, un segundo valor, tercer valor, etc. Se puede afirmar que el raıź viene el primero, pero ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoríá 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

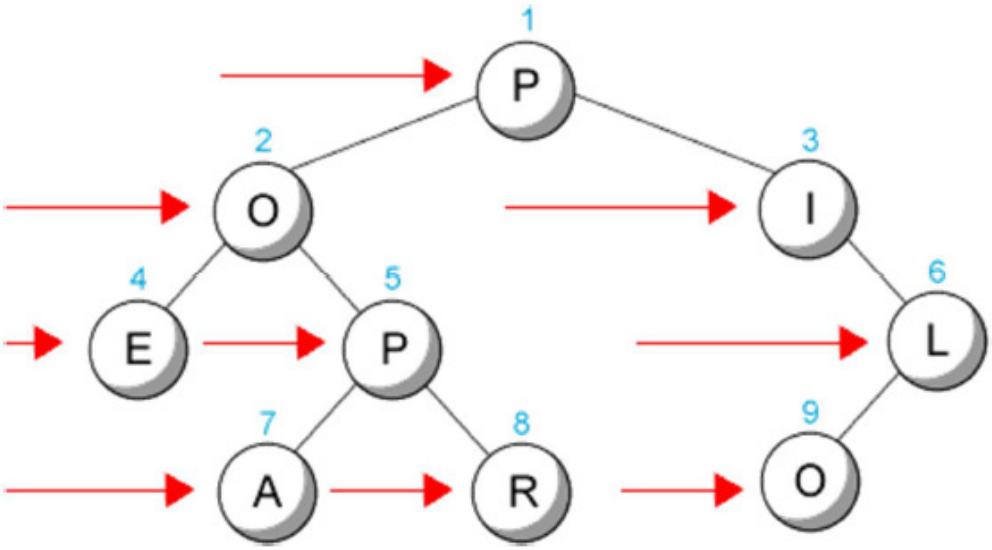
un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo. Para saber cómo regresarnos, vamos guardando los nodos visitados en una estructura de **pila**. Es por esto que se acostumbra programar esta búsqueda de forma recursiva, 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:



Recorrido en profundidad

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raiź 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**.



Recorrido en anchura