TAREA\_52

**Estructura-de-datos | array | lista-enlazada | grafo | colas | pilas | árboles**

**Estructura de datos:** En programación, se llama estructura de datos a la forma particular en que se almacenan, organizan y gestionan datos o información. De manera que el acceso a los mismos o su modificación sea mucho más fácil.

Existen muchas clasificaciones de estructura de datos. Atendiendo a cómo los datos se representan en la memoria, una de ellas podría ser la siguiente:

* Las estructuras **contiguamente asignadas** están compuestas de bloques de memoria únicos, e incluyen a los arrays, matrices, heaps, y hash tables.
* Las **estructuras enlazadas** están compuestas de distintos fragmentos de memoria unidos por pointers ó punteros, e incluyen a los listas, árboles, y grafos.
* Los **contenedores** son estructuras que permiten almacenar y recuperar datos en un orden determinado sin importar su contenido, en esta se incluyen los stacks y queues.

**ESTRUCTURAS CONTIGUAMENTE ASIGNADAS**

**Array**: Esta estructura es “la” fundamental de las estructuras contiguamente asignadas. Arrays ó arreglos son estructuras de datos de tamaño fijo de modo que cada elemento puede ser eficientemente ubicado por su index (índice) ó dirección.

Ventajas:

Eficiencia de espacio: Los arrays son puramente datos, no se desperdicia espacio de memoria guardando información extra que ayude a la localización, como pasa con las listas enlazadas.

Acceso de tiempo constante: al estar la información almacenada de forma contigua, cada index apunta directamente a una dirección de memoria, el acceso a la información en cualquier posición es inmediato.

Localización de memoria: a la hora de iterar sobre ellos como tienen una excelente localización de memoria, permiten aprovechar una mayor velocidad de cache.

*Desventajas*:

Su tamaño no se puede modificar a mitad de la programación. Cómo solución, se puede crear crear una array lo suficientemente grande para almacenar nuestros datos, pero esto deriva en un desperdicio de memoria totalmente innecesario, ó podemos crear un nuevo array, doblar el tamaño de éste cada vez que se necesite crecer y copiar los datos del array anterior al nuevo array, hacer esto tiene el mismo nivel de complejidad que si tuviéramos un array único suficientemente grande, pero con la ventaja de que sólo va a crecer cuándo sea necesario, evitando así desperdiciar memoria. Este tipo de arrays se conocen con el nombre de arrays dinámicos o *dynamic arrays*.

Otra estructura muy común que nos ayuda a representar datos reales son las llamadas **matrices o tablas** que no son más que arrays de múltiples dimensiones.

**ESTRUCTURAS ENLAZADAS**

La magia de las estructuras enlazadas es dada por los *pointers* o punteros, que como su nombre indica apuntan a una dirección de memoria donde se encuentra ubicado un valor. Por medio de los *pointers* tenemos una secuencia de valores enlazados, que no necesariamente están ubicados en un espacio contiguo de memoria.

Ventajas:

Es posible cambiar su tamaño en cualquier momento.

Desventajas:

No sólo guardan los valores, si no la información necesaria para enlazar un valor con otro (un nodo con otro), es decir, la dirección en memoria del siguiente valor.

**Listas enlazadas**: Una lista es aquella estructura que representa un número contable de valores ordenados donde un mismo valor puede repetirse y considerarse un valor distinto a otro ya existente. Las listas son consideradas secuencias de valores y cómo ya veíamos en la explicación anterior sobre pointers estos se pueden aplicar para implementar una lista de tal forma que las características principales de una lista serían:

* Cada nodo en nuestra lista contiene uno o más campos que contienen el valor que deseamos almacenar.
* Cada nodo contiene al menos un campo pointer apuntando a otro nodo, lo que significa que podemos tener 2 pointers, uno que apunta a un nodo consecuente y otro que apunta a un nodo previo formando así una lista doblemente enlazada (double linked list).
* Es necesario tener un pointer que apunte a la cabeza de la estructura para así saber por dónde comenzar.

**Array vs Linked list**

* Una diferencia grande entre los *arrays* vs *linked lists* es que insertar o eliminar de una *linked list* es más fácil ya que no tiene tamaño fijo y lo único que debemos hacer para insertar o eliminar un valor es simplemente apuntar al nuevo nodo creado, ó apuntar al siguiente nodo de la lista si un nodo fue eliminado. En un *array* no existe esta flexibilidad.
* Si tenemos una gran cantidad de valores siempre será más fácil mover *pointers* de una valor a otro que mover los valores en sí.
* En un *array* podemos provocar un desbordamiento de memoria (*memory overflow*) si queremos insertar un valor extra y ya hemos excedido el tamaño del *array*, lo cual no sucedería en una *linked list.*
* Por otro lado en una *linked list* necesitamos más espacio en memoria para almacenar los *pointers.*
* Los *arrays* tienen un mejor manejo del acceso aleatorio a los valores y son mucho mejores en la ubicación de los datos en memoria y aprovechamiento de la caché.

## CONTENEDORES

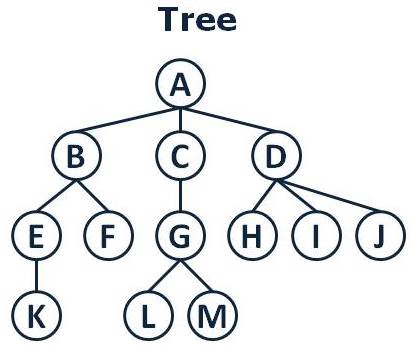
Las estructuras de tipo contenedor se caracterizan principalmente por la forma particular de recuperación ordenada de datos que soportan, y en los dos tipos principales de contenedores (stack y queue) el orden de recuperación depende del orden de inserción

## Stack

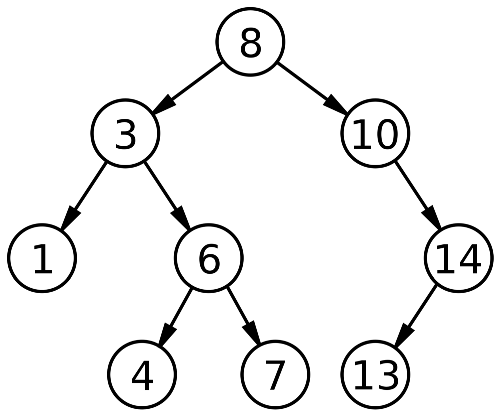
**Un stack o pila**, soporta la recuperación ordenada de datos last-in, first-out (LIFO) o bien: el último dato en entrar, el primer dato en salir. Se usan cuando **el orden** de la recuperación de datos **no nos importa tanto**, simplemente queremos apilarlos y des-apilarlos, por lo que las operaciones fundamentales en un stack son **push** y **pop** para poner y obtener datos de la pila

## Queue

Una **queue o cola**, soporta la recuperación ordenada de datos first-in, first-out (FIFO) o bien: el primer dato en entrar, es el primer dato en salir. En este tipo de estructura **el orden sí importa**, pues siempre el primero de la cola debe ser el primero en ser atendido y el resto de forma ordenada esperan su turno. Las operaciones de esta estructura son: **enqueue y dequeue** para encolar y des-encolar (poner y obtener de la cola).

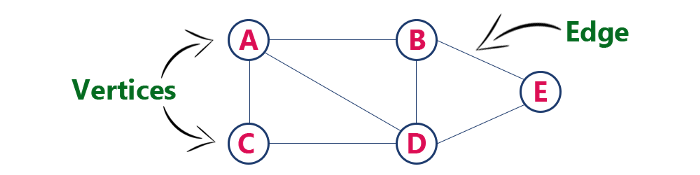
**Árbol**: Un árbol es una estructura que consta de nodos y hojas. Cada nodo está conectado a otro nodo de forma jerárquica existiendo nodos “padre” y nodos “hijo” en distintos niveles, de modo que podamos crear una jerarquía desde un nodo raíz (root) hasta un último nivel de nodos hijo. Un tree, en realidad es un caso específico de un grafo  

**Árbol binario**: En un árbol binario, a la izquierda se encuentran los nodos cuyo valor es menor al del nodo raíz, y a la derecha los de mayor valor, de forma que: un árbol binario sólo acepta tener como máximo 2 nodos hijo (izquierda y derecha).



Grafo: Son importantes porque se pueden usar para representar esencialmente **cualquier** relación, por ejemplo: las calles en una ciudad, las computadoras conectadas en un red local, las conexiones de vuelos nacionales o internacionales de un aeropuerto a otros, un circuito eléctrico, etc.

En un grafo podemos representar la conexión de muchos nodos, los vértices o vertex y bordes o edges son el conjunto de elementos dentro de un grafo y no existe una representación única.



<https://medium.com/techwomenc/estructuras-de-datos-a29062de5483>

<https://medium.com/techwomenc/estructuras-de-datos-e15b419e9e8e>

https://www.youtube.com/watch?v=wv90D4dZUA0