# GUIA DE ESTRUCTURA DE DATOS

Nicklaus Wirth dijo que:

Algoritmo + Estructuras de datos = Programas

## Tipos de estructuras de datos

Por Asignación de Memoria:

* **Estáticas:** La asignación de memoria siempre es la misma aun en ejecución.
* **Dinámicas:** Asignación de memoria cambiante durante ejecución.

Por Estructura de Memoria:

* **Lineales:** Tienen un enlace simple y son accesibles por su posición.
* **No Lineales:** Contienen elementos con múltiples enlaces y pueden accederse a ellos por múltiples caminos.

## Tipos de datos

Básicos: Son los datos que aparecen en todos los Lenguajes de Programación.

* **Primitivos:** Int, Float, Char, Bolean, Cadenas de Caracteres, Punteros, Enumeraciones, Etc.
* **Compuestos:** Arreglos (Unidimensionales y Multidimensionales).

Abstractos (ADT): Es una clase en POO creada por el programador (Datos Encapsulados + Métodos que los manipulan = ADT).

### ADT Arrray

Es un arreglo de una sola dimensión donde los elementos pueden ser accesibles por medio de un índice.

Métodos:

* Array( tamanio ) 🡪 Constructor donde recibirá el tamaño lineal del arreglo.
* getItem( ubicación ) 🡪 Retorna el elemento en la ubicación ingresada.
* setItem( elemento, ubicación ) 🡪 Guarda el elemento en la ubicación dada.
* getLenght() 🡪 Retorna el tamaño del arreglo.
* clear( elemento ) 🡪 Hace todos los elementos del arreglo, el elemento dado por default 0.
* getIterator() 🡪 Retorna el recuento de todos los elementos en el arreglo(Parecido o igual a toString()).

### ADT Array2D

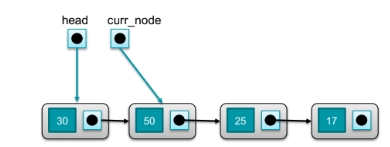
Es un arreglo de dos dimensiones (comúnmente llamados renglón y columna) de igual forma que el arreglo de una dimensión los elementos son accesibles por medio de índices.

Métodos:

* Array2D( renglón,columna) 🡪 Constructor del arreglo.
* clear( elemento ) 🡪 Hace todos los elementos del arreglo, el elemento dado por default 0.
* get\_row\_size() 🡪 Retorna el numero de renglones que contiene el arreglo.
* get\_col\_size() 🡪 Retorna el numero de columnas que contiene el arreglo.
* set\_item(renglón, columna, dato) 🡪 Guarda el elemento en la ubicación dada.
* get\_item(renglón, columna) 🡪 Retorna el elemento en la ubicación ingresada.
* to\_string() 🡪 Retorna el estado actual del arreglo mostrando los elementos que contiene.

### ADT Listas Simplemente Ligadas

Es una estructura de datos ligada el cual consiste ene el uso de nodos que contienen un enlace de una sola dirección; funciona con un apuntador el cual funciona como entrada a la estructura (head). Para el recorrido se hace uso de otro apuntador que nos servirá como ayuda (curr\_node) el cual estará apuntando cada nodo de nuestra estructura uno a la vez.



next

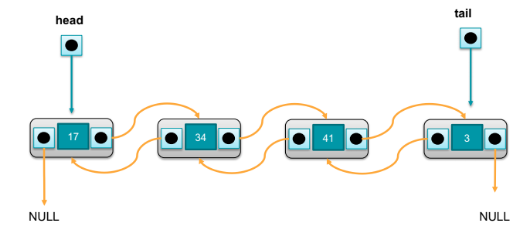
Métodos:

* LinkedList() 🡪 Constructor de la lista.
* Is\_empty() 🡪 Retorna True o False dependiendo si la lista esta vacía o no.
* tail() 🡪 Retorna el ultimo nodo de la lista.
* append( elemento ) 🡪 Agrega el nodo con el elemento al final de la lista entrando por head.
* prepend( elemento ) 🡪 Agrega el nodo con el elemento al inicio de la lista renombrando el apuntador head a este nodo.
* transversal() 🡪 Hace el recorrido transversal de la lista empezando con head, mostrando todos sus elementos.
* remove( elemento ) 🡪 Elimina el primer elemento que coincida con el ingresado de la lista.

### ADT Listas Doblemente Ligadas

Es una estructura de datos ligada que funciona con dos nodos apuntando uno al siguiente y otro al anterior, de igual forma que la simplemente ligada cuenta con un apuntador a la entrada (head) pero cuenta con otro apuntador final (tail) el cual puede ser también otra entrada de la lista.





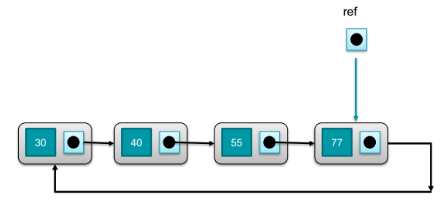
Métodos:

* DoubleLinkedList() 🡪 Constructor de la lista doblemente ligada.
* get\_size() 🡪 Retorna el numero de elementos que contiene la lista.
* is\_empty() 🡪 Retorna True o False dependiendo si la lista esta vacía o no.
* append( elemento ) 🡪 Agrega el nodo con el elemento al final de la lista entrando por head.
* find\_from\_tail( elemento ) 🡪 Busca el elemento ingresado iniciando por tail y retorna la posición.
* find\_from\_head( elemento ) 🡪 Busca el elemento ingresado iniciando por head y retorna la posición.
* remove\_from\_tail( elemento ) 🡪 Elimina el elemento ingresado buscandolo inciando por tail.
* remove\_from\_head( elemento ) 🡪 Elimina el elemento ingresado buscandolo inciando por head.
* transversal() 🡪 Hace el recorrido transversal de la lista empezando con head, mostrando todos sus elementos.
* reverse\_transversal() 🡪 Hace el recorrido transversal de la lista empezando con tail, mostrando todos sus elementos.

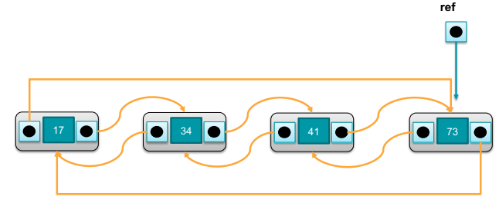
### ADT Listas Circulares

Es una estructura de datos ligada linealmente en donde el ultimo elemento de la lista apunta al primero, esta estructura solo cuenta con un apuntador el cual indica la entrada a la lista, el cual por conveniencia esta posicionado en el ultimo elemento de la lista.

#### Lista circular simplemente ligada



#### Lista circular doblemente ligada

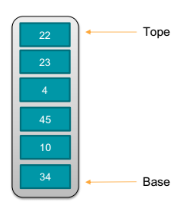


Métodos:

* CircularList() 🡪 Constructor de la lista circular.
* is\_empty() 🡪 Retorna True o False dependiendo si la lista esta vacía o no.
* insert( elemento ) 🡪 Agrega el elemento de forma ordenada desde menor hasta mayor.
* transversal() 🡪 Hace el recorrido transversal de la lista, mostrando todos sus elementos.
* search( elemento ) 🡪 Busca el elemento ingresado y dependiendo si lo encuentra o no retornara un true o un false.
* remove( elemento ) 🡪 Elimina de la lista el elemento ingresado.

### ADT Pilas

Es una estructura de datos lineal el cual implementa el protocolo de acceso LIFO(ultimo en entrar es el primero en salir), donde los elementos son metidos y sacados desde el mismo extremo llamado tope, el opuesto del tope es la base.



Aplicaciones:

* Verificar balanceo de paréntesis, llaves, etc., en expresiones y programas.
* Administración de llamada a funciones y excepciones.
* Recorrido en árboles y grafos.
* Evaluar expresiones aritméticas.
* Cambiar de notación infija a posfijas expresiones matemáticas.
* Estrategia de diseño de algoritmos Backtracking.

Métodos:

* Stack() 🡪 Constructor de la pila.
* is\_empty() 🡪 Retorna True o False dependiendo si la lista está vacía o no.
* lenght() 🡪 Regresa el numero de elementos en la pila.
* pop() 🡪 Saca el elemento del tope y lo retorna. Cambiando el puntero tope al anterior elemento.
* peek() 🡪 Retorna el elemento en el tope sin sacarlo de la pila.
* push( elemento ) 🡪 Agrega el elemento ingresado al tope de la pila.

### ADT Colas

Es una estructura de datos lineal que implementa el protocolo de acceso FIFO (el primero en entrar es el primero en salir) donde los elementos son agregados por un extremo y son sacados en el extremo contrario.



Salida

Entrada

Métodos:

* Queue() 🡪 Constructor de la pila.
* is\_empty() 🡪 Retorna True o False dependiendo si la cola está vacía o no.
* lenght() 🡪 Retorna el número de elementos existentes en la cola.
* enqueue( elemento ) 🡪 Ingresa el elemento ingresado al final de la cola.
* dequeue() 🡪 Saca y retorna ele elemento que se encuentra la frente o a la salida de la cola.
* to\_string() 🡪 Muestra el estado actual de la cola, mostrando todos sus elementos.

### ADT Colas de Prioridad

Es una estructura de datos lineal o de dos dimensiones donde los elementos son agregados por un extremo teniendo asignada una prioridad, para poder sacar un elemento se sacara primero el de mayor prioridad aplicando para elementos de la misma prioridad el protocolo FIFO de igual manera se aplicara el protocolo para insertar elementos.

#### Cola con Prioridad No Acotada

No tiene limites en su rango de valores de prioridad ya que los niveles de prioridad no están preestablecidos teniendo una mayor flexibilidad de crecimiento.

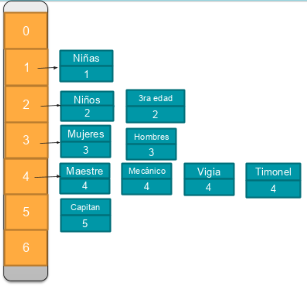


Métodos:

* PriorityQueue() 🡪 Constructor de la cola con prioridad no acotada.
* is\_empty() 🡪 Retorna True o False dependiendo si la cola está vacía o no.
* lenght() 🡪 Retorna el número de elementos existentes en la cola.
* enqueue(prioridad, elemento) 🡪 Añade el elemento ingresado al final de su prioridad.
* dequeue() 🡪 Saca y retorna el elemento que se encuentra la frente o a la salida de la cola por su prioridad.
* to\_string() 🡪 Muestra el estado actual de la cola, mostrando todos sus elementos.

#### Cola con Prioridad Acotada

Tiene un rango limitado de prioridades que van de 0 a p, donde su constructos establece el numero de niveles de prioridad; de igual manera en cada nivel implementara el protocolo FIFO el primer elemento en salir es el de más alta prioridad ( 0 ).



Métodos:

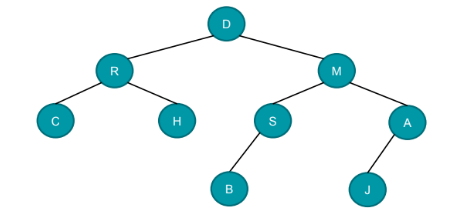
* BoundedPriorityQueue( niveles ) 🡪 Constructor de la cola con prioridad acotada.
* is\_empty() 🡪 Retorna True o False dependiendo si la cola está vacía o no.
* lenght() 🡪 Retorna el número de elementos existentes en la cola.
* enqueue( prioridad, elemnto ) 🡪 Añade el elemento ingresado al final de su prioridad.
* dequeue() 🡪 Saca y retorna el elemento que se encuentra la frente o a la salida de la cola por su prioridad.
* to\_string() 🡪 Muestra el estado actual de la cola, mostrando todos sus elementos.

### ADT Arboles Binarios

Es una estructura de datos que cuenta con un orden de , donde los hijos de un nodo son conocidos como hijo izquierdo e hijo derecho, sus elementos se pueden recorrer en in\_orden, post\_orden y pre\_orden.

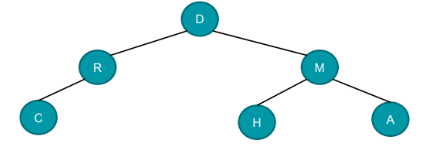
#### Árbol Binario Equilibrado

Es aquel árbol en el que las alturas de las hojas mas altas de los subárboles tienen una diferencia de máximo un nivel.



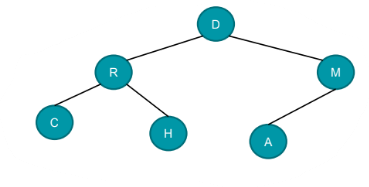
#### Árbol Binario Perfectamente Equilibrado

Es aquel árbol en el que las alturas de las hojas mas altas de los subárboles tienen la misma altura.



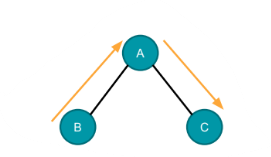
#### Árbol Binario Completo (Lleno)

Es aquel árbol binario equilibrado, ene le cual del nivel 1 al nivel n-1 tienen todos los nodos llenos. Tomar en cuenta que los nodos del nivel n se van agregando primero al subárbol izquierdo.

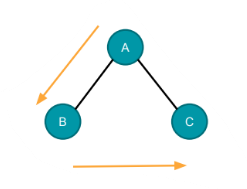


#### Recorridos de Arboles Binarios

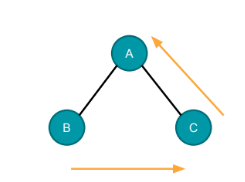
* In-Orden



* Pre-Orden



* Post-Orden



#### Árbol Binario de Búsqueda

Es un árbol lleno y cullos nodos están ordenados; los nodos son posicionados los emnores a la izquierda y los mayores a la derecha, permitiéndonos búsquedas más rápidas. Recordar los elementos repetidos no se insertan.

Métodos:

* BinarySearchTree() 🡪 Constructor del árbol binario de búsqueda.
* insert( elemento ) 🡪 Inserta el elemento en la hoja o nodo correspondiente del árbol.
* remove( elemento ) 🡪 Elimina el nodo que contenga el valor ingresado.
* search( elemento ) 🡪 Busca el elemento ingresado en el árbol retornando un true o un false según lo encuentre.
* transversal( recorrido ) 🡪 recorrerá el árbol según sea el tipo de recorrido ingresado(preorden, posorden e inorden).

## Recursividad

La recursividad es una herramienta en el proceso de diseño de algoritmos que consiste en resolver problemas subdividiendo un problema grande en pequeños casos hasta el punto en que se conviertan en un caso trivial, esta basada en la capacidad de una función de programación que se pude llamar a sí mismo.

Cada implementación recursiva requiere de un caso base, de lo contrario se haría infinita.