Programación orientada a objetos - Informática II

Clase 6 – Estructuras avanzadas de datos

¡Hola! ¿Cómo están? Continuamos bajo la modalidad en línea de Informática II, en la clase de hoy nos centraremos en lo que comenzamos a ver en Informática I y profundizaremos en Informática II, el contenido de esta clase serán las estructuras avanzadas de datos.

Estructuras avanzadas de datos

En esta clase veremos las estructuras conocidas como cola, pila y listas. Recordemos que en Informática I cuando vimos estos temas ya trabajábamos con memoria dinámica, no solo para hacer un uso más óptimo de la memoria sino porque no conocíamos la cantidad de elementos o recursos a utilizar.

Para comenzar con este tema abordaremos 3 ejes:

- La definición de cada tipo de estructura
- Las operaciones que podemos realizar con cada estructura
- Las aplicaciones que pueden tener

Recordando Punteros

- Un puntero es una variable cuyo valor es la dirección de memoria de otra variable.
- Se utiliza como método de acceso indirecto.
- Son fundamentales para el manejo de memoria dinámica y pasaje por referencia.

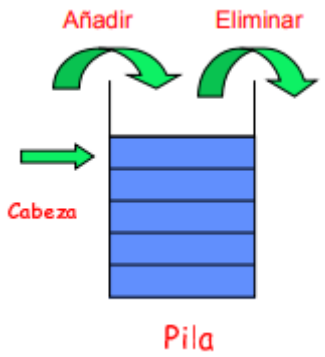
PILA

Como hemos visto en Informática I, la forma de organizar elementos es a través de un array o vector. La característica es que **todos** los elementos que componen a un array deben ser iguales, es decir, que un array es una *entidad coherente y lógica con elementos homogéneos* o comunes.

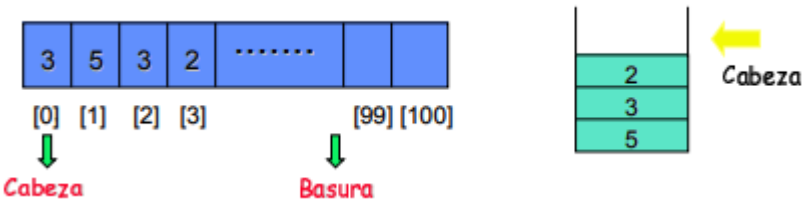
¿Qué característica tiene la PILA?

Sobre este tipo de estructuras podemos agregar/añadir o eliminar datos siempre a partir de una cabeza o header. Esta estructura se la conoce como LIFO, por sus siglas en ingles, Last input, First Output, es decir, el último que ingresa es el primero que sale.

Podemos pensarlo como la **pila** de la ropa, a medida que vamos sumando ropa para ordernar, la prenda más vieja será la que está debajo de toda la ropa, la última prenda corresponderá a lo último que hayamos colocado antes de llevarlo a planchar, lavar o que corresponda☺.



En Informática I, pedíamos memoria dinámica con *malloc*, y luego a medida que trabajamos íbamos haciendo *realloc* a partir de los diferentes requerimientos. En este caso, todas las posiciones de memoria eran consecutivas y podíamos trabajar con un índice.



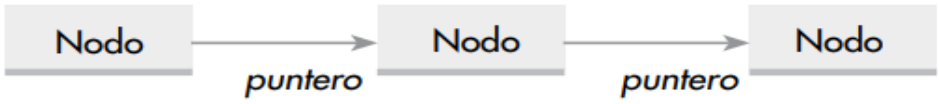
Tal como sabemos no hace falta que todos los elementos sean valores enteros sino que podemos complejizar nuestros tipos de datos a partir de una necesidad específica. Por ejemplo, podemos contar con una estructura del tipo **alumno** que contenga: nombre, apellido y edad, por lo tanto en cada **posición** de nuestro array contaremos con la información correspondiente a cada **alumno** y acceder a sus datos.

¿Qué sucede si mis datos continúan creciendo? ¿Puedo contar siempre con memoria suficiente para almacenar toda la información que necesite?

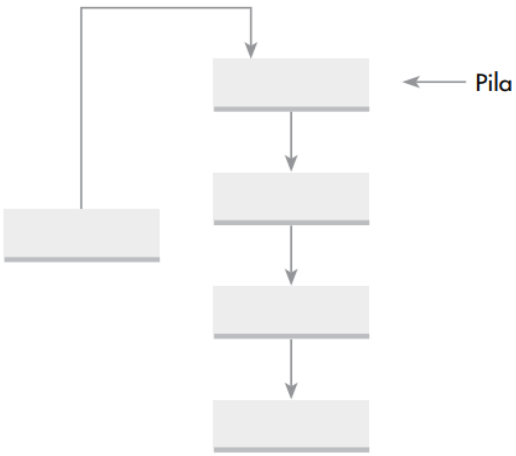
Desde ya, la respuesta es NO! Entonces, que podemos hacer para solucionar este problema. Dentro de las estructuras de datos, contamos con las estructuras auto-referenciadas. Partamos este concepto, por un lado **referencia**, nos habla de *punteros* y **auto** nos habla de "nosotros", es decir, que es "puntero que apunta hacia nosotros", no queremos ponernos muy reflexivos ☺ , pero lo que necesitamos saber es que contaremos con un puntero que apunta a una estructura del mismo tipo. De esta forma, ya no tendremos un índice por el cual podemos movernos dentro de un array sino que trabajaremos con el contenido de lo apuntado por una serie de estructuras ***enlazadas*** por el valor de un puntero.

¿Cómo denominamos formalmente entonces a esta estrategia?

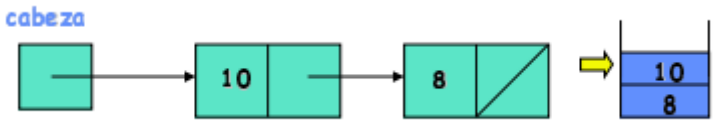
Las estructura auto-referenciadas son una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un “enlace” o “referencia”.



Veamos la estructura vinculada a una pila:



A partir del ejemplo de elementos enteros, intentaremos graficar lo dicho y obviamente aclarar lo máximo posible en la video-clase correspondiente:



COLA

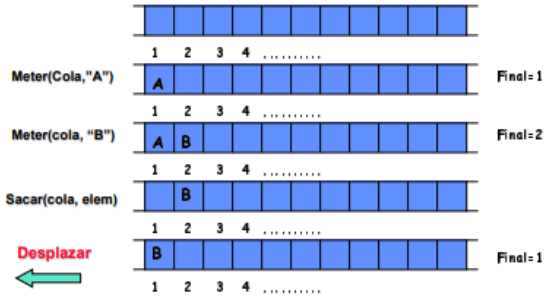
Veamos esta estructura de la misma forma que hicimos con la PILA. Primero pensando en una estructura basada como array y luego pensando en su implementación con **nodos**.

Una **cola** es un grupo de datos homogéneos, en la cual los elementos se agregan al final y se retiran desde el comienzo. Obviamente para ilustrarlo podemos pensar en una cola del supermercado, donde nos colocamos al final y el cajero o cajera nos va llamando por el orden en el que llegamos y nosotros vamos saliendo del supermercado (en este momento, con 2 metros de distancia entre cada elemento ☺).

¿Qué acciones podemos realizar?

- Meter / Poner en la cola
- Sacar / Quitar de la cola
- Saber si la cola está vacía

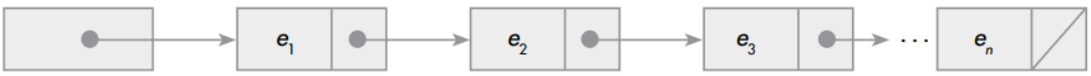
Veamos un ejemplo:



Las colas se conocen como estructuras **FIFO** (First-in, First-out, primero en entrar-primero en salir) debido a la forma y orden de inserción y de extracción de elementos de la cola.

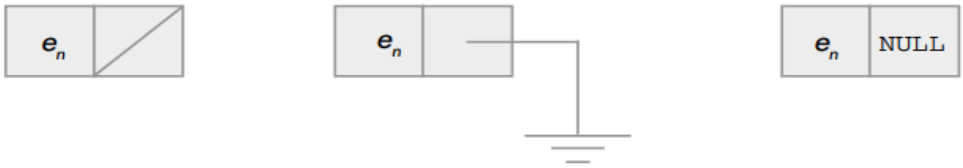
Las colas tienen numerosas aplicaciones en nuestro "mundo": colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

Si lo pensamos en una estructura apoyada en **nodos**, partiendo de una cabecera, cada una de las estructuras se iran *encolando,* a través de cada uno de los punteros, sabiendo que cada **nodo** posee un puntero que nos *enlaza* con el siguiente:



¿Cómo sabemos cuál es el último elemento?

Como trabajamos con punteros y no podía ser de otra, el último **nodo** contendrá un puntero cuyo valor será NULL. De esta forma podremos ir barriendo cada uno de los **nodos** que compone a la **cola** y determinar si llegamos al final o no. Existen diferentes formas de graficar esta situación, podemos adoptar la que no siente mejor, en particular, solemos adoptar el puntero a "masa".

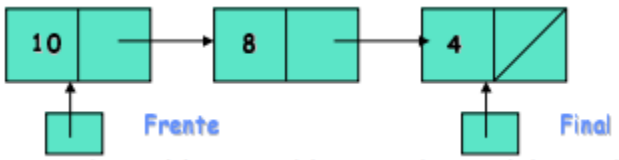


<u>Otras estructuras de datos</u>

Existen otras estructuras de datos avanzadas, como son las *listas* con algunas características. La diferencia principal en las *listas*, es que los datos no son agregados al principio o al final sino que son insertados con un **determinado criterio**.

- Lista simplemente enlazada
- Lista circular simplemente enlazada
- Lista circular doblemente enlazada
- Lista doblemente enlazada

Si seguimos trabajando con enteros, podríamos tener todos los elementos insertados de **mayor a menor**, de esta forma no existiría el ordenamiento posterior sino que cada elemento es insertado cumpliendo con el criterio explicitado.



Si pensamos en las acciones que podemos realizar sobre una lista, el repertorio se acciones se amplía:

- Crear
- ¿Está vacía?
- Insertar
- Eliminar
- Listar / Imprimir
- Destruir

Nota importante:

Cuando liberamos memoria desde una cola, pila o lista implementada con **nodos** deberemos reescribir nuestra función tal como la conocemos, no es posible realizar un *free* o un *delete* de la memoria pedida ya que NO es un array, por lo tanto debemos liberar cada **nodo** solicitado en forma individual.

Otras implementaciones en estructuras avanzadas:

Ejemplo de lista circular

Ejemplo de lista doblemente enlazada

Final de esta clase

Bueno, este es el final de la clase de C++ sobre el uso de estructuras avanzadas de datos.

Los y las invitamos a ver la video-clase que complementa este material de lectura inicial.

Si surgen dudas o consultas, pueden escribirnos en el foro correspondiente.

En la próxima clase, comenzaremos a trabajar sobre uno de los pilares de la programación orientada a objetos que es la **herencia.**

¡Nos leemos!