# ESTRUCTURA DE DATOS

***UNIDAD 1:*** *Tipos Algebraicos.*
***UNIDAD 2:*** *Tipos Abstractos y Eficiencia Operacional.*
***UNIDAD 3:*** *Modelo Destructivo y Estructuras de Bajo Nivel.*
***Temas Adicionales:*** *Hashing, Sorting, Grafos.*

**En la primera parte de la materia** se usa un lenguaje funcional (Haskell) y se trabaja sobre el modelo funcional (o denotacional). Se trabajan más que nada técnicas como pattern matching, recursión, etc; para implementar funciones.

**Después se trabajan Estructuras de Datos**, las cuales las implementan armando Tipos Abstractos de Datos (o TADs), que son tipos de datos que el usuario va a utilizar sin conocer su implementación. Solo va a utilizar las funciones que le ofrece ese TAD. Uno como implementador, implementa las funciones del TAD. Uno como usuario, usa las funciones del TAD sin saber como funcionan para implementar otras funciones.

**En esta parte, solo se fijan en la eficiencia en cuanto a tiempo** porque el concepto de memoria no existe en el modelo denotacional, ya que está abstraído de esa cuestión, es algo de lo que no te preocupás. Obviamente existe la memoria al ejecutar un programa Haskell, pero no interesa como funciona.

**En la segunda parte de la materia** se trabaja con C++ y sobre el modelo imperativo (o destructivo). Esto significa que los programas de C++ van a causar un cambio de estado. Acá también, se trabaja sobre el manejo de memoria, punteros y recorridos usando punteros.

Se van a implementar también TADs. Algunos de estos solo en C++ porque tienen sentido en este modelo. Los TADS los van a implementar usando punteros y tratando de que las funciones sean eficientes en cuanto a tiempo y a Memoria RAM ocupada durante la ejecución (tanto memoria stack como memoria heap, **algo de lo que son muy exigentes). Sobre el parcial que suelen tomar en esta segunda parte:**

- Se tienen que hacer funciones en C++ que tengan que tener una eficiencia en tiempo que ellos te dan. Además, tenes que justificar que la eficiencia de tu implementación para la función era igual a la que ellos te dan, obvio que también tenés que tratar de ocupar la menor cantidad de memoria heap/estática posible, porque si gastás de más innecesariamente va a ser ineficiente.

- Una es revertir una Linked List que te dan (te dan un puntero a una Linked List), pero no tenes que crear una nueva al revés, sino revertir que te daban, o sea que no tenes que devolver nada.

- La otra es que te dan una Linked List (puntero a Linked List) y tenes que crear un Array en memoria estática con todos los elementos de la Linked List en orden y devolver el puntero al Array. No olvidar que se tiene que devolver el puntero al Array y no el Array como tal.

- El otro es que te dan un Array (puntero a un Array) y vos tenes que crear y devolver una Linked List con todos los elementos del Array.

- El otro es que te dan un Array y vos tenes que crear y devolver un Array con los valores de este pero ordenados de menor a mayor, y te piden (o sugieren) que para lograr eso uses una Linked List que después tenes que descartar.

- Lo más importante es lo que se escribe de eficiencia. Se fijan mucho en eso.

# CLASE 1 - Presentación y Tipos Algebraicos

**¿Qué es el modelo destructivo (o imperativo)?**

- Es uno de los modelos de la computación. En el modelo destructivo (o imperativo), uno ejecuta comandos y esos comandos producen efectos (como pasaba en Gobstones, que transformaba el estado del tablero), básicamente transformación de estado: de un estado inicial a uno final.

**¿Qué es el modelo denotacional (o funcional)?**

- Es otro de los modelos de la computación. En el modelo denotacional (o funcional), no se hablan de comandos, sino que se evalúan expresiones que describen valores. Por ende, el programa es una expresión y su resultado es el valor de esa expresión.

**¿Qué es un dato?**

- Es una unidad de información. Es alguna forma de representar la información de forma tal que pueda distinguirla y manipularla. Se piensa en la idea del dato, y se escriben mediante una expresión.

**¿Qué es una expresión?**

- Una expresión según Introducción a la Programación, *es una descripción de información*, descrita mediante combinación de símbolos que describe a un dato. Una expresión se forma a partir de constantes, variables y funciones; y las cuales pueden evaluarse para obtener su valor, tal como en Introducción a la Programación.

**¿Qué es un tipo de dato?**

- Es un atributo para clasificar expresiones. Dicha clasificación es en término de qué puedo hacer con dichos valores. La utilidad real de que existan tipos de datos es imponer restricciones a las combinaciones posibles (si yo quiero sumar, sumo dos números).

**¿Qué es un sistema de tipos?**

- Son un conjunto de reglas para asignar (o decidir) tipo a las expresiones. La idea en la materia no es estudiar un sistema de tipos, pero se van a ver varias cosas a la hora de trabajar con tipos. Los sistemas de tipos en los lenguajes actuales, usualmente se ejecutan durante la compilación.

**¿Qué es una función?**

- Es una forma de expresión que permite transformar otras expresiones, transforma argumentos en resultados. Dichas funciones se nombran mediante identificadores y se escribe la función a continuación sus argumentos “multiplicar 2 3”.

**¿Qué es una estructura de datos?**

- Es una forma particular de organizar datos para ser accedidos o manipulados de cierta forma (usualmente en forma eficiente), por ejemplo, una lista.

**¿Qué significa el “::” de la sintaxis de Haskell?**

- Cada definición en Haskell puede indicar su tipo, y esto se puede hacer mediante el signo “::”, que significa *“tiene tipo”*.

**¿Cuáles son los diferentes tipos de comentarios en Haskell?**

- En Haskell hay dos tipos de comentario: de línea que comienzan con -- y de párrafo que van entre {- y -}.

**¿Cómo se indenta en Haskell?**

- Se usa el espacio para indicar y dejar en claro a qué pertenece cada cosa, porque al no existir las llaves como en Gobstones, el intérprete de Haskell "se guía" mediante la indentación (aunque más que nada es para comunicar mejor).

**¿Qué tipos básicos hay en Haskell?**

- En Haskell hay varios tipos básicos:
  - **Números** (Int, Integer, Float, Double (*1, 2, 3*)).
  - **Caracteres** (Char (*'a', 'b', 'c'*)).
  - **Booleanos** (Bool (*True*, *False*)).
  - **Strings** (String (*"Hola"*; siendo cada String una lista de caracteres)).

**¿Cómo se definen los tipos algebraicos?**

- Son tipos que se definen mediante constructores. Los constructores pueden llevar argumentos y cada uno representa un grupo de elementos. En general, los tipos algebraicos que se ven son tipos variantes y registros:
  - *Los tipos algebraicos variantes*, son muy parecidos a los enumerativos de Gobstones. Se dan los constructores que definen los casos, donde cada constructor (diferenciado por la primera letra en mayúscula) determina un elemento, y cada caso está separado por una barra "|".
  - *Los tipos algebraicos registros*, son muy parecidos a los registros de Gobstones. Se da un único constructor con argumentos, y dicho constructor debe tener un nombre diferente al tipo.

**¿Cuántos tipos algebraicos hay?**

- Estos son los tipos algebraicos que hay:
  - **Enumerativos:** varios constructores sin argumentos (ej, Dirección).
  - **Registros o productos:** un único constructor con varios argumentos (ej, Persona).
  - **Sumas o variantes:** varios constructores con argumentos (ej, Helado).
  - **Recursivos:** suma que usa el mismo tipo como argumento (ej, Listas).

**¿Qué es pattern matching y cómo funciona?**

- De forma simple, pattern matching es *usar los constructores* para preguntar o acceder a los datos, en lugar de usarlos para construir los datos. Su funcionamiento se basa de la siguiente forma:
  - Los constructores se pueden usar para acceder y se usan en los parámetros.
  - Si el parámetro coincide, usa dicha ecuación.
  - Si no coincide, continúa verificando la siguiente ecuación.
  - Solo se aplica pattern matching a constructores de tipos algebraicos.
  - El orden de las ecuaciones importa a la hora de utilizar pattern matching en las funciones.
- Es posible usar el comodín "_" si un parámetro no se usa, aunque no como una expresión.

**¿Qué ventaja tiene usar funciones observadoras vs pattern matching ?**

- La principal ventaja que tiene, es que puede pasar que uno no tenga siempre acceso a la estructura interna de un tipo de dato (ya sea por que la desconozca o porque por alguna razón, cambie el orden de las definiciones), en ese caso, las funciones observadoras siempre van a funcionar (si están bien aplicadas). En cambio, con pattern matching puede pasar que ante alguna modificación de la estructura interna de un tipo de dato, de error.

**¿Qué es el tipo algebraico variante: suma?**

- Son un tipo algebraico variante más complejo, ya que admite que los constructores puedan tener argumentos. Además de eso, los tipos en posición de argumento de los constructores, van SOLAMENTE en las declaraciones data, es decir, si uno lo declara va a tener que utilizar el constructor seguido de un constructor válido del tipo utilizado como argumento.

**¿Qué es una tupla?**

- Una tupla es una estructura de datos predefinida, similares a un registro pero sin nombre de campo, y se escriben entre paréntesis y con comas. Son como (aunque no exactamente) el producto cartesiano en matemáticas.
  Además, los paréntesis y la coma funcionan como un constructor, por ende, es posible hacer pattern matching sobre tuplas.

**¿Qué es polimorfismo paramétrico o funciones polimórficas?**

- Polimorfismo paramétrico es una característica que permite definir una única función para cualquier tipo de datos de los que se invocan, por ende, no es necesario redefinir los mismo una y otra vez.
  Si a una función le importa la estructura, pero no los datos específicos, puede ser polimórfica (es posible mientras no se opere con el parámetro, es decir, si se usa alguna operación no polimórfica, no es posible). Si importa el dato, la función es monomórfica.

**¿Qué es una lista?**

- Una lista es una estructura de datos predefinida, son similares a las listas de Gobstones, pero hay algunas diferencias:
  - [1, 2, 3] es la abreviatura de 1:2:3:[].
  - Las operaciones “:” (se lee cons, puede utilizarse con la palabra Cons seguido de dos argumentos) y “[]” (se lee lista vacía) son constructores. El “:” toma un elemento y una lista y devuelve una lista del mismo tipo.
  - La operación de agregar “++” no es predefinida y además, tampoco es constructora. El “++” toma dos listas y devuelve una lista del mismo tipo de las listas originales.
  - Así, son equivalentes: [1, 2, 3] ó 1:2:3:[] ó 1:[2, 3] ó [1] ++ [2, 3].

# CLASE 2 - Listas y Recursión Estructurada

**¿Qué es un constructor?**

- Un constructor es importante en la definición de un tipo algebraico, ya que actúa como una función que crea valores de ese tipo y, opcionalmente, almacena otros valores dentro de ellos. Además, los constructores son la base del pattern matching, que permite inspeccionar y manipular los valores de los tipos algebraicos de manera estructurada y segura. Es decir, a los constructores se los puede usar para construir pero también se los puede utilizar para preguntar si algo está construido de una cierta manera. En listas hay solo dos constructores: cons ":", y nil "[]".

**¿Qué es recursión estructural?**

- Se le llama recursión estructural a la forma de poder "hacer recorridos" en Haskell, y se le llama así ya que sigue el mismo esquema que la estructura de la lista (las listas están hechas con cons y nil, y los constructores definen su estructura). A la hora de pensar en recursión estructural siempre vienen a la idea dos casos:
  - **Caso base:** se plantea un caso inicial (o default).
  - **Caso recursivo:** se utiliza la misma función que se define sobre una parte de la lista original.

  Todas las funciones recursivas contienen una ecuación por cada constructor, y además, se utiliza la misma función en las partes recursivas.
- La razón por la que se realizan recorridos de esta forma en Haskell es porque no hay comandos imperativos para bucles, es puramente funcional.

**¿Por qué se utiliza recursión estructural y no otra cosa?**

- Se utiliza recursión estructural para recorrer listas en Haskell ya que no hay comandos imperativos para bucles, es puramente funcional. Por ende, es una posible solución para este contexto.

**¿Siempre se utiliza recursión estructural?**

- No, solo cuando la situación me lo permita, ya que no es posible utilizar recursión estructural cuando la función es parcial.

**¿Por qué es recomendable realizar primero el caso recursivo antes del caso base?**

- Es buena practica realizar primeramente el caso recursivo antes del caso base, ya que el caso base es simplemente el elemento neutro de la operación que se realiza en el caso recursivo. Por ende, si uno desconoce lo que se realiza en el caso recursivo le va a ser muy difícil determinar el elemento neutro de dicha operación.

**¿Se puede realizar recursión estructural sobre números?**

- Sí, sigue el mismo principio de la recursión "normal" aunque con algunas diferencias:
  - La parte recursiva de un número es su anterior.
  - El caso base es el 0.
  - No se aplica a números negativos.
  - Continua teniendo dos casos: caso base y caso recursivo.

# CLASE 3 y 4 - Tipos Algebraicos Recursivos

Primeramente, recordemos un par de cosas...

**¿Qué es un constructor?**

- Un constructor es importante en la definición de un tipo algebraico, ya que actúa como una función que crea valores de ese tipo y, opcionalmente, almacena otros valores dentro de ellos. Además, los constructores son la base del pattern matching, que permite inspeccionar y manipular los valores de los tipos algebraicos de manera estructurada y segura. Es decir, a los constructores se los puede usar para construir pero también se los puede utilizar *para preguntar si algo está construido de una cierta manera.* En listas hay solo dos constructores: cons ":", y nil "[]".

**¿Cuántos tipos algebraicos hay?**

- Podemos determinar que los tipos algebraicos vistos hasta ahora son:
  - **Enumerativos:** varios constructores sin argumentos (ej, Dirección).
  - **Registros o productos:** un único constructor con varios argumentos (ej, Persona).
  - **Sumas o variantes:** varios constructores con argumentos (ej, Helado).
  - **Recursivos:** suma que usa el mismo tipo como argumento (ej, Listas).

Entonces...

**¿Qué es un tipo algebraico recursivo lineal?**

- Un tipo algebraico recursivo es un tipo nuevo que se define a través de constructores, donde al menos uno de los constructores utiliza el mismo tipo que se está definiendo como argumento. Estos tipos son un tipo de suma (o variante). Dentro de estos tipos, están los tipos algebraicos recursivos lineales, los cuales se definen de manera similar a otros tipos algebraicos recursivos, utilizando el mismo tipo como argumento en algunos de sus constructores.
  La característica "lineal" se refiere a que solamente contiene una parte recursiva dentro de su definición.

**¿Qué es un tipo algebraico recursivo árbol?**

- Es una estructura que tiene mas de una parte recursiva en su definición, ya que la lineal solamente contiene una parte recursiva.

**¿Cómo pensar los árboles?**

- Inicialmente se piensan igual que cualquier tipo que sea recursivo, pensando en el caso (o casos) base y recursivo (o recursivos). Después, se plantea en el caso recursivo del árbol, generalmente "un algo" y dos caminos posibles, que idealmente se piensan como "camino de la izquierda" y "camino de la derecha", en donde ambas dan un resultado y deben ser combinadas a conveniencia.

**¿Qué es un árbol binario?**

- Es un tipo algebraico recursivo árbol, en el que solamente se conserva la estructura más básica para poder operar.

**¿Entonces cómo es recomendable plantear una recursión estructural en general?**

- Se recomienda seguir la siguiente metodología: Decidir si usar recursión sobre qué estructura → Plantear el esquema → Resolver los casos recursivos → Completar los casos base.

# CLASE 5 - TADs (Stacks, Queues y Sets)

Es recomendable antes de ver Tipos Abstractos de Datos, repasar toda la Unidad 1...

**¿Por qué tiene sentido aprender TADs si los Tipos Algebraicos eran útiles?**

- El problema es que los Tipos Algebraicos presentan una gran dependencia de la estructura, y eso lo hace complicado cambiarla, además de que no es posible controlar la forma en la que se usan los datos (se pueden construir cosas inválidas). Todo esto, hace que tenga sentido plantear la idea de los Tipos Abstractos de Datos (o TADs).

**¿Qué es un Tipo Abstracto de Dato o TAD?**

- Son una forma de definir tipos, pero la manera es diferente a la definir tipos algebraicos. En lugar de decirte “estos son los constructores de tipos”, te voy a decir “estas son las funciones básicas que el tipo maneja”. Para eso es necesario dar: nombre y tipo de cada una de las operaciones, y el comportamiento esperado de cada una esas operaciones.
  Algo relevante es que no va a ser necesario saber la implementación de las operaciones, no interesa, solo saber qué hacen las operaciones.

**¿Cómo se usa un Tipo Abstracto de Dato en Haskell?**

- En Haskell se utiliza un mecanismo que se llama “mecanismo de módulos”. Un modulo en Haskell es un grupo de definiciones que tiene: nombre y una lista de operaciones que son visibles, implementación de cada una de las operaciones y otras auxiliares necesarias, y opcionalmente, indica qué otros módulos utiliza.

```
module <nombre> (<interfaz>) where
        <implementación>
```

La interfaz es la única manera de usar un elemento, de esta manera, la interfaz define todas las maneras posibles de uso.

**¿Se puede usar Pattern Matching sobre TADs?**

- NO se puede hacer pattern matching sobre TADs porque lo que te da una interfaz NO son constructores. Pattern matching solamente es para tipos algebraicos.

**¿Cuál es el rol entonces que tenemos que tener en los TADs?**

- La idea es que uno puede ser usuario o implementador de TADs, pero no diseñarlas porque no tiene que ver con esta materia. Ser usuario implica usar lo que está definido en la interfaz, en cambio, ser implementador implica decidir la representación interna, fija el invariante de representación, y provee la implementación de las operaciones. Lo importante es saber cuando uno es usuario, y cuando uno debe ser implementador (la parte mas difícil).

**¿Qué es un invariante de representación?**

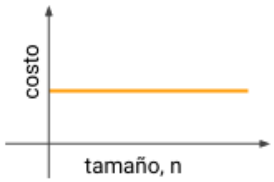
- Los invariantes de representación son fundamentales desde el lado del implementador, aunque carecen de sentido para el usuario del TAD ya que no los conoce. Para un implementador de un TAD, los invariantes son una guía fundamental para el desarrollo interno del mismo, ya que son restricciones que hacen que el TAD sea consistente, tenga sentido y además lo más eficiente posible.

**¿Qué cosas debe tener en cuenta un implementador de TADs?**

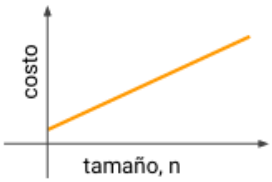
- Como implementador de TADs se debe tener en cuenta un par de cosas. Primeramente que si uno es usuario a la vez de otros TADs, solamente utilizar su interfaz y tratar de no saber su implementación.
  Segundo, sería ideal que se planteen Invariantes de Representaciones que tengan sentido con el TAD que se está implementando y que además, lo haga más eficiente.
  Tercero, implementar internamente cada operación de la interfaz planteada del TAD que se está implementando.

**¿Qué es eficiencia, para qué sirve y cómo se mide la eficiencia?**

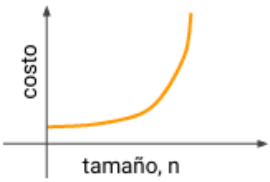
- La eficiencia tiene que ver con el costo que puede llegar a tener en el peor caso una función u operación. Esto sirve para tener una idea de qué tan útil puede ser usar una implementación u otra de una función, operación o TAD.
- Para medir la eficiencia, es necesario ver como se comporta la función a medir en el peor caso posible y en función a la cantidad de datos de la estructura. En base al comportamiento general en peor caso, usando un indicador grueso de eficiencia, el cual está basado en el siguiente orden:
  - **Costo constante:** la función siempre tarda lo mismo, no depende de la cantidad de elementos.
  - **Costo logarítmico:** se recorre una rama una rama de un árbol balanceado.
  - **Costo lineal:** por cada elemento de la estructura, solamente se hacen operaciones de costo constante.
  - **Costo lineal logarítmico "eneloguene":** hasta operaciones logarítmicos por cada elemento.
  - **Costo cuadrático:** por cada elemento de la estructura, se hacen operaciones de costo lineal (en peor caso).



Constante, O(1)



Lineal, O(n)



Cuadrática, O(n^2)

**¿Qué es un Stack?**

- Un Stack es un TAD clásico, en el mismo se pueden ingresar, consultar y eliminar elementos. La propiedad que cumple es *"el último que entra es el primero que sale"* (en inglés, Last In - First Out o LIFO). Tiene aplicaciones en ejecución de lenguajes, en problemas de balanceo, entre otras más.

**¿Qué es una Queue?**

- Una Queue es un TAD clásico, en el mismo se pueden ingresar, consultar y eliminar elementos. La propiedad que cumple es *"el primero que entra es el primero que sale"* (en inglés, First In - First Out, o FIFO). Tiene aplicaciones en sistemas operativos, atención de cajas, entre otras más.

**¿Qué es un Set?**

- Un Set es un TAD clásico, en el que se pueden ingresar elementos y consultar existencia. La propiedad que cumple es *"no hay elementos repetidos en un conjunto"*. Su aplicaciones tiene sentido en donde se deben mantener elementos sin repetición.

# CLASE 6 - TADs: (Multisets, Maps y Priority Queues)

Es recomendable antes de ver Tipos Abstractos de Datos, repasar toda la Unidad 1...

**¿Qué es un Multiset?**

- Un Multiset es un TAD que modela colecciones con repeticiones, diferente de un Set. El funcionamiento es que indique la cantidad de veces que se agregó un elemento.

**¿Qué es un Map?**

- No confundir con mapas, un Map es un diccionario. La idea del Map es que es un TAD que permite asociar una clave a un valor, y además cada clave es única en el Map (no admite claves repetidas). Otro detalle interesante, es que cada clave tiene un valor asociado específico, y cada valor tiene una clave asociada que puede no ser la misma.
- Las implementaciones eficientes (como las basadas en árboles) logran costos logarítmicos O(log K) para las operaciones de búsqueda, inserción y borrado, lo cual es crucial para aplicaciones con muchos datos.

**¿Qué es una Priority Queue?**

- Una Priority Queue es un TAD similar a una cola, pero donde el orden de salida de los elementos no es el de llegada, sino que siempre sale el elemento de mayor (o menor) prioridad. Básicamente el próximo que sale es el mínimo (de máxima prioridad).
- Las implementaciones eficientes se basan en la estructura de Heap, logrando un costo constante O(1) para consultar el elemento de mayor/menor prioridad y un costo logarítmico O(log M) para insertar y eliminar, lo cual termina siendo mucho más eficiente que las implementaciones lineales O(n).

**El origen del por qué algo puede ser de costo log n, o por qué se le llama así.**

- El costo *log n* se encuentra entre el costo constante O(1) y el costo lineal O(n), y suele ser preferible frente a O(n) por ser mucho más eficiente. En cuanto a su origen, en un árbol binario completo (donde no falta ningún nodo), la cantidad de nodos *n* y la altura *h* están relacionadas. Cada nivel duplica la cantidad de nodos del anterior, por lo que se cumple que $n = 2^h$. Si despejamos *h*, obtenemos que $h = \log n$. Esto significa que el logaritmo indica cuántas veces hay que multiplicar por 2 para llegar a *n*.
- En el peor caso de una búsqueda en un BST (árbol binario de búsqueda), se descarta la mitad del árbol en cada paso (se busca solo en una de las dos subramas), y si el elemento está en el último nivel, la cantidad de operaciones realizadas es igual a la altura del árbol, es decir, *log n*.

**¿Pero de qué me sirven todas estos TADs?**

- Porque básicamente son herramientas para organizar y manipular datos de formas específicas (colecciones con repeticiones (como Multiset), asociaciones clave-valor (como Map), elementos ordenados por prioridad (como Priority Queue)).
- Su eficiencia operacional es ideal para obtener costos idealmente logarítmicos o constantes, por eso requieren implementaciones basadas en estructuras de datos eficientes como árboles balanceados, que se basan en invariantes específicos para garantizar que las operaciones fundamentales sean baratos para operar.

# CLASE 7 y 8 - TADs: (Heaps, BST y AVL's)

Es recomendable antes de ver Tipos Abstractos de Datos, repasar toda la Unidad 1...

**¿Qué son los BST?**

- Un BST es un Binary Search Tree (árbol binario de búsqueda), donde en (NodeT x ti td): todos los elementos de ti son menores que x, todos los elementos de td son mayores que x, ti y td también cumplen con este invariante. Básicamente la funcionalidad fundamental es entorno a la búsqueda dentro de un árbol dado, ya que resulta muy barato en cuestiones de eficiencia si el árbol es BST. Además, BST es una implementación de un Set que busca costos logarítmicos.

**¿Cómo se puede mantener el balance en un BST?**

- Para mantener el balance en un BST es necesario un invariante, para esto hay varias opciones: AVLs (Adelson-Veslky & Lands), Red-Black Trees, AA Trees, Tango Trees, y muchas otras variantes.

**¿Qué son los AVLs?**

- Un AVL es un tipo de BST que también se utiliza para implementar Set, pero es mucho mejor que un BST ya que termina logrando garantizar costos logarítmicos incluso en el peor caso. Además del invariante de un BST (en relación al orden), AVL contiene otro invariante y es que debe estar balanceado (la diferencia entre el subárbol izquierdo con el derecho es como máximo de 1). Lo que termina implicando todo esto, es que la inserción como el borrado requieren "rebalanceos" en el árbol, lo cuál no es la idea que sepamos hacer pero al menos entender la idea del rebalanceo (para eso son útiles las páginas para visualizarlo).

**¿Qué son las Heaps?**

- Un Heap se utiliza para implementar Priority Queue de forma eficiente, y existen dos tipos: MaxHeap y MinHeap. En un MinHeap, la raíz del árbol es el elemento mínimo de todos, y sus subárboles también cumplen con el invariante del Heap. En un MaxHeap, la raíz del árbol es el elemento máximo de todos, y sus subárboles también cumplen con el invariante del Heap. Para los Heaps Binarios, hay una invariante importante adicional que se llama "árbol lleno", en donde todos los niveles excepto quizás el último, están completos, y el último nivel se llena de izquierda a derecha sin "agujeros". *Esto permite ubicar el lugar de inserción/borrado de forma única.* Esto termina resultando que para encontrar el máximo en un MaxHeap, la operación sea de costo constante, ya que es la raíz del árbol. Es por eso que esta implementación sirve para encontrar el máximo o el mínimo de algo (esto en el caso de MinHeap).
- Otro detalle a destacar, es que en las Heaps se admiten repetidos.

**Páginas para visualizar mejor el tema AVLs y Heaps:**

- *AVLs:* https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- *Heaps:* https://gallery.selfboot.cn/en/algorithms/heap

**Recomendaciones para el parcial:**

- Fijarse bien la consigna para poder fijar buenos invariantes de representación.
- Revisar bien las interfaces dadas, suelen dar bastante información de cómo está hecho internamente ese TAD, lo cual es útil (más cuando la consigna no es clara).
- No dar más invariantes de representación de las que son necesarias, es penalizado fuertemente.

# CLASE 9 - Modelo Imperativo

**¿Qué es el Modelo Funcional?**

- El modelo denotacional (o funcional) se basa en la evaluación de expresiones que describen valores. El programa es una expresión y su resultado es el de esa expresión. Las funciones se utilizan para abstraer combinaciones de expresiones y transforman valores.

**¿Qué es el Modelo Imperativo?**

- El modelo destructivo (o imperativo) se centra en la ejecución de comandos que producen efectos o modifican el estado. El programa transita de un estado inicial a un estado final, siendo la memoria el estado general, similar al tablero en Gobstones. Los procedimientos son usados para abstraer secuencias de comandos y alteran el estado.

**¿Por qué usar el Modelo Imperativo?**

- La razón es porque se pueden lograr implementaciones más eficientes, además que refleja como funcionan las computadoras a bajo nivel.

**¿Qué es la memoria?**

- En el modelo destructivo, el estado que se modifica es usualmente la memoria.
- A muy bajo nivel, la memoria es un arreglo de celdas (bytes), cuyo contenido se interpreta en binario. En C/C++, cuando se ejecuta un programa cada variable se asigna a celdas de memoria y su contenido se codifica en binario.
- A un nivel abstracto, es posible pensar a la memoria como un grupo de espacios de memoria, y que además las variables locales se organizan en frames. Esta abstracción permite no preocuparse por los detalles de bajo nivel, como la cantidad de celdas o el código binario.

**¿Qué son los frames?**

- Los frames (marcos) son espacios de memoria donde se ubican las variables locales de un procedimiento. Las funciones en C/C++ tienen efectos permanentes; y las que no devuelven nada (void) son procedimientos.
- Cada función tiene su propio frame. Al anidar funciones (incluyendo llamadas recursivas), los frames se apilan, formando lo que se conoce como Stack (pila), y cada frame es un stack frame. Los parámetros de las funciones también ocupan espacio de memoria y se comportan como variables.
- La memoria gestionada de esta forma se denomina memoria estática.

**Entonces, ¿en qué se diferencian la recursión y la iteración en relación a memoria?**

- Las funciones recursivas en C/C++ consumen una cantidad lineal de memoria (O(n)) debido a la anidación de frames.
- Es por eso, que se prefieren las iteraciones, que ocupan memoria constante, ya que son más eficientes al evitar la constante asignación y liberación de memoria. Aunque en relación a tiempo siga siendo lineal, termina resultando más conveniente que la recursión en la mayoría de los casos.
- El único caso en donde es conveniente la recursión, es en los árboles.

**¿Qué tipos de datos ya existentes se ven en C/C++ por ahora?**

- **Strings:** Se representan con una celda por carácter y terminan con un carácter nulo (NULL).
- **Structs**: Son similares a los registros de Gobstones, donde el espacio de memoria de un struct se compone de los espacios de todos sus campos.

**¿Cuándo es necesario mirar el bajo nivel en C/C++?**

- Es necesario considerar el bajo nivel cuando no se respeta el tipo de dato (ej. operar un char como un número) o para acceder a partes específicas de un string usando índices (ej. *msg[i]*).

**¿Se pueden implementar TADs en C/C++?**

- Sí, se puede intentar implementar TADs en C/C++ usando memoria estática, con la interfaz (.h) y la implementación (.cpp) separadas.
- Sin embargo, esto conlleva la copia de variables y resultados *(Persona p; p.nombre = n; return(p))*, lo que es ineficiente y puede llevar a duplicaciones de datos no deseadas. *Este modelo de memoria estática no es suficiente para la gestión de TADs complejos.*

# CLASE 10 - Punteros, Memoria Dinámica y Arrays

**¿Qué es la memoria estática (o Stack)?**

- Es una memoria cuyo comportamiento no depende de valores de ejecución. Los frames de las funciones se abren al invocar y se eliminan al terminar, comportándose como una pila.

**¿Qué problemas conlleva utilizar la memoria estática (o stack)?**

- Uno de los problemas más comunes a la hora de utilizar memoria estática, es la copia de ida y vuelta de parámetros y resultados. Además de la posibilidad de "clonar" elementos de forma no deseada.
- Es por eso que es conveniente utilizar la memoria dinámica (o heap).
- Además, el hecho de poder compartir memoria estática trae muchísimos problemas, como dejar punteros apuntando al vacío.

**¿Qué es la memoria dinámica (o heap)?**

- Es una memoria formada por espacios reservados cuyo comportamiento e*s controlado explícitamente por el programador.*
- Nace como una posible solución ante los problemas que existen con utilizar la memoria estática con TADs.

**¿Qué es un Puntero?**

- En otros lenguajes, un puntero se lo conoce como referencia.
- Un puntero es una dirección de memoria (un número) que indica dónde se encuentra un dato. Se usa con el operador * para declararlos y acceder a la memoria la que apuntan. Para acceder a campos de un registro a través de un puntero, se usa *(*p).campo* o la abreviatura *p->campo*.
- De manera simple, un puntero es como una llave que me da acceso a la memoria heap, me dice en qué lugar de la memoria está y cuando doy la llave estoy dando permiso de acceder a esa parte de la memoria (es como el modo indirecto de Organización de Computadoras).

**¿Cómo se utiliza la Memoria Dinámica en C++?**

- En C++, primeramente se pide lugar en esta memoria con la operación "*new*", y esta operación termina devolviendo un puntero al espacio reservado.

**¿Cómo se puede gestionar la memoria dinámica (o heap)?**

- La operación "*new*" se utiliza para pedir espacio en la heap y devuelve un puntero al espacio reservado. La operación "*delete*" libera la reserva de memoria en la heap; después de "*delete*", el puntero no debe usarse hasta que sea reasignado.

**¿Qué problemas conlleva utilizar la memoria dinámica (o heap)?**

- El principal problema es la *memory leak* (pérdida de memoria), que ocurre cuando la memoria solicitada ya no puede ser accedida porque se pierde la referencia al puntero. La solución es usar "*delete*" cuando la memoria ya no es necesaria.

**Entonces, ¿se puede crear un TAD con memoria dinámica (o heap)?**

- Si, un ejemplo claro es el tipo Persona de las diapositivas, el cual se define como un puntero a una estructura *(typedef PersonaSt* Persona)*.
- Las operaciones como nacer asignan la memoria para la persona en la heap y devuelven un puntero, copiando solo el puntero, no toda la estructura. Operaciones como CumplirAnios pueden ser procedimientos (void) que modifican la memoria compartida a través del puntero, sin necesidad de devolver un nuevo valor.

**¿Qué ventajas/desventajas tiene utilizar la memoria dinámica en vez de la estática?**

- **Como ventaja**, se evitan las copias innecesarias de datos (solo se copian punteros) y se elimina la posibilidad de "clonar" elementos de forma no deseada.
- **Como desventaja**, la gestión manual de la memoria puede provocar memory leaks si no se libera correctamente.

**¿Qué son los Arrays?**

- Los Arrays son estructuras de datos que almacenan muchos datos del mismo tipo en celdas de memoria contiguas. Se usan corchetes [] para reservarlos y acceder a sus celdas. Pueden ser estáticos (en el stack, desaparecen al terminar la función y no pueden ser retornados) o dinámicos (en la heap, se crean con new y pueden ser retornados).

**¿Qué ventajas o desventajas tiene utilizar Arrays?**

- **Como ventaja**, el acceso a cualquier dato en O(1) si se conoce la posición.
- **Como desventaja**, debe conocerse la cantidad máxima de antemano (para arrays estáticos), pueden llenarse, y pueden dejar "agujeros" si se eliminan elementos del medio.
- El concepto de ArrayList surge para solucionar el problema del tamaño máximo, duplicando el espacio cuando es necesario y copiando elementos.

**¿Qué son los Arrays Multidimensionales?**

- Los Arrays Multidimensionales, se utilizan para manejar colecciones de datos con más de una dimensión (matrices, tensores). C/C++ no los maneja directamente, por lo que se pueden simular con arrays unidimensionales y cálculos de índices (row + col * maxRow) o arrays de punteros a arrays.

# CLASE 11 - Linked Lists y Arboles en Imperativo

**¿Qué son las Linked List (o listas enlazadas)?**

- Es una alternativa a los ArrayLists para tener tamaño variable.

**¿Cómo se representan las Linked List?**

- Se representan con:
  - **Nodos** (NodeL) que contienen un valor (value) y un puntero (next) al siguiente nodo. El último nodo apunta a NULL.
  - **Es común tener un encabezado** (ListHeaderSt) para la lista, que almacena el puntero al primer nodo (first), el tamaño (size) y, opcionalmente, un puntero al último nodo (last) para un acceso eficiente al final.

**¿Qué incluye la Interfaz (imperativa) Linked List?**

- Incluye operaciones como emptyList, Cons (agregar al inicio), Snoc (agregar al final), head (primer elemento), Tail (eliminar primer elemento), isEmptyList, length y Liberar (liberar toda la memoria). Cons, Snoc y Tail suelen ser O(1). Liberar es O(n).

**¿Qué son los Iteradores?**

- Para realizar recorridos no destructivos sobre la lista (sin modificarla), se utilizan *iteradores* (ListIterator), que son un tipo asociado a la lista que permite avanzar y acceder a elementos. Un *iterador* tiene un puntero al nodo actual (current) y operaciones como iniciarRecorrido, estaAlFinalDelRecorrido, elementoActual, PasarAlSiguienteElemento y LiberarIterador.

**¿Cómo se pueden implementar Árboles en Imperativo?**

- La representación más común es usando punteros para los hijos *(left, right)* y un valor *(value)* en cada nodo. Utilizar árboles en imperativo implica un manejo complejo de la abstracción e invariantes. Es por eso que normalmente se suelen manejar directamente.

**¿Qué tipos de recorridos hay para Árboles en Imperativo?**

- **La recursión** es la forma común de recorrer árboles, tanto para funciones que retornan información sin afectar el árbol (ej. sumT) como para funciones que lo modifican (ej. SuccT).
- **La liberación de memoria (LiberarTree)** debe hacerse recorriendo el árbol (ej. en post-orden).
- **Los recorridos lineales (iterativos)** son posibles si el resultado no depende de la estructura completa del árbol (ej. sumT, sizeT). Requieren el uso de estructuras auxiliares como pilas (stacks) o colas (queues) para almacenar los nodos pendientes.
- **Recorrido en Profundidad (DFS - Depth First Search):** Se comporta como una pila *(LIFO).* Se explora una rama completamente antes de retroceder. Al agregar hijos a la lista de pendientes, se utiliza una operación Cons (agregar al frente).
- **Recorrido a lo Ancho (BFS - Breadth First Search):** Se comporta como una cola (*FIFO).* Se exploran todos los nodos de un nivel antes de pasar al siguiente. Al agregar hijos a la lista de pendientes, se utiliza una operación Snoc (agregar al final).

**¿Qué es una Heap Binaria, y como se implementa en Imperativo?**

- Una Heap Binaria es un árbol que cumple el invariante de heap (la raíz es el mínimo de todos los elementos y sus subárboles también son Heaps) y el invariante de árbol lleno (todos los niveles, excepto quizás el último, están completos, y el último no tiene "agujeros" de izquierda a derecha).
- Pueden representarse eficientemente usando un Array, donde la raíz está en la posición 1 y los demás nodos se ubican por niveles.
- Las posiciones de los hijos de un nodo *i* son *2*i* (izquierdo) y *2*i+1* (derecho), y el padre es *i/2*.

**¿Qué tipos de operaciones puede realizar una Heap Binarias en Imperativo?**

- **Insertar:** El nuevo elemento se agrega en la primera posición libre del Array (para mantener el invariante de árbol lleno) y luego "flota" hacia arriba intercambiándose con su padre si es menor, hasta que se cumple el invariante de heap.
- **Borrar el Mínimo:** El mínimo (raíz) se elimina, el último elemento del Array se mueve a la raíz, y luego se "hunde" hacia abajo, intercambiándose con el menor de sus hijos, hasta que se cumple el invariante de heap.

**¿Por qué es tan eficiente la implementación de una Heap Binaria en Imperativo?**

- La eficiencia obtenida es alta gracias al uso de Arrays y trucos de implementación.