Filas

Es una lista ordenada en la cual las operaciones de inserción se efectúa en un extremo llamado último y las operaciones de borrado se realizan el otro extremo llamado primero.

Trabajan según el método **FIFO** (Primero en entrar, primero en salir).

Arboles binarios

Un **árbol binario** es un conjunto finito de elementos, de nombre, Nodos de forma que:

- El árbol binario **está vacío** si no tiene **ningún elemento**.
- Se denomina árbol binario al tipo de árbol que **contiene un Nodo raíz y dos Nodos** que parten de él, llamados **Nodo Izquierdo** y **Nodo Derecho** (que también son árboles).
- Un árbol binario es un árbol en el que **ningún nodo puede tener más de dos subárboles.**
- En un árbol binario. Cada nodo puede tener **0, 1 o 2 hijos (subárboles)**.
- El árbol binario **completo**, es un árbol binario **lleno, completo y balanceado**.

Características de un árbol

- **Nodo Raíz**: Es aquel que tiene hijos pero **no tiene padre**.
- **Nodo Rama**: Es aquel que **tiene padre e hijos**, también se le llama nodo padre o hijo.
- **Nodo hoja**: Es el que tiene padre pero **no tiene hijos**.
- **Nivel**: Aumenta a medida que se aleja de la raíz. El nivel de la raíz es 0.
- **Altura**: Parte de la raíz y según su descendencia así es su altura. Es el nivel máximo que puedo encontrar.
- **Grado**: Es la descendencia de cada Nodo (cantidad de hijos que tiene un nodo).

Preorden, inorden y postorden

En general, la diferencia entre preorden, inorden y postorden es cuando se recorre la raíz. En los tres, se recorre primero el subárbol izquierdo y luego el derecho.

Preorden: R - I - D
Inorden: I - R - D
Preorden: I - D - R

- En preorden, la raíz se recorre antes que los recorridos de los subárboles izquierdo y derecho.
- En inorden, la raíz se recorre entre los recorridos de los árboles izquierdo y derecho
- En postorden, la raíz se recorre después de los recorridos por el subárbol izquierdo y el derecho.

Preorden (antes), inorden (en medio), postorden (después).

Borrar un nodo en un árbol binario

Si A != NULL	Si dato == A->r	Si A->i != NULL	nodoArbol * masDer = NMD(A->i); A->r = masDer->r; A->i = borrar(A->i, masDer->r);	Al encontrar el nodo con el dato buscado, sobreescribo este dato con el dato del "nodo más derecho" del subárbol izquierdo (si existe nodo izquierdo)
		Sino si A-d != NULL	nodoArbol * masIzq =NMI(A->d); A->r = masIzq->r; A->d = borrar(A->d,masIzq->r);	Similar al caso anterior, pero considerando el otro subárbol.
		Sino (cuando A es hoja)	Free (A); A = NULL;	Cuando A es hoja, simplemente se libera la memoria, se la establece en NULL y se la retorna.
	Si dato > A->r		A ->der = borrar (A->d, dato);	Se busca el dato recursivamente en el subarbol derecho
	Si dato < A->r		A ->izq = borrar (A->i, dato);	Se busca el dato recursivamente en el subarbol izquierdo.
Si A == NULL				El elemento buscado no está en el árbol. No hacer nada, no hay else
		Al final de la función	return A	Siempre retornar A;

Al encontrar el elemento, si es hoja, simplemente se lo elimina; si no es hoja, debemos reemplazar el dato del nodo encontrado, con el dato siguiente en orden (el menor o el mayor). Este dato lo encontramos en el NMI(D) (nodo más izquierdo del subárbol derecho) o en el NMD(I) (nodo más derecho del subárbol izquierdo.

Casos a contemplar

- Borrar un Nodo sin hijos.
- Borrar un Nodo con un subárbol hijo.
- Borrar un Nodo con dos subárboles hijos.

Borrar un Nodo con dos subárboles hijos

Tenemos que tomar el hijo derecho del Nodo que queremos eliminar y recorrer hasta el hijo más a la izquierda (hijo izquierdo y si este tiene hijo izquierdo repetir hasta llegar al último nodo a la izquierda), reemplazamos el valor del nodo que queremos eliminar por el nodo que encontramos (el hijo más a la izquierda), el nodo que encontramos por ser el más a la izquierda es imposible que tenga nodos a su izquierda pero sí que es posible que tenga un subárbol a la derecha, para terminar solo nos queda proceder a eliminar este nodo de las formas que conocemos y tendremos la eliminación completa.

Estructuras Compuestas

Son estructuras que permiten representar datos que resultan de la composición de estructuras más simples. Son de gran utilidad cuando se deben resolver problemas complejos que involucran la utilización de una gran variedad de tipos.

Las **composiciones más frecuentes** son:
1. Arreglo de Listas.
2. Arreglo de Árboles.
3. Listas de Listas.
4. Listas de Árboles.
5. Árboles de Listas.

Las **listas de listas** pueden usarse cuando la estructura principal requiere la flexibilidad de una **estructura dinámica** (poder crecer, inserciones en orden y borrados rápidos, etc.). Un uso posible es para la representación de grafos o estructuras de datos con dos jerarquías claramente definidas (Categoría y Empleados).

Las **estructuras que combinan árboles binarios** son aquellas en dónde se requiere una **búsqueda** eficiente de los elementos por una clave en particular, en especial si él o los árboles se mantienen balanceados.

En todos los casos **cada TDA debe proveer todas las funciones necesarias con sus respectivas validaciones**. Por ejemplo, proveer la inserción de un Empleado en la categoría correcta.

La elección de una estructura u otra dependerá de los requerimientos particulares del problema a resolver. En cada caso se deberá analizar los mismos y elegir la mejor opción en base a, por ejemplo: tamaño a ocupar en memoria (datos fijos cuyo tamaño se conoce de antemano o dinámicos), eficiencia requerida para las búsquedas, cantidad altas y bajas esperadas, etc. Todos estos criterios podrán lograrse en menor o mayor medida con cada una de las combinaciones planteadas.

Programación orientada a objetos

La POO, es una nueva manera de pensar y resolver un sistema. Ya no se subdividen tareas complejas en un conjunto de tareas simples.

Un programa no es otra cosa que un conjunto de entidades y objetos que se comunican y colaboran entre sí para lograr un objetivo común.

La POO basa su funcionamiento en 5 pilares:

- **Abstracción**: Generalización no conceptual de un determinado conjunto de objetos y de sus atributos y propiedades.
  Deja en segundo término los detalles concretos de cada objeto.

- **Polimorfismo** (un mismo objeto se puede comportar de diferentes formas ante un mismo mensaje): Acceso a un grupo de funciones a través de la misma interfaz.
  Un identificador puede tener distintas formas (distintos cuerpos de función y comportamientos)

- **Modularidad**: Propiedad de un sistema descompuesto en un conjunto de módulos cohesivos débilmente acoplados.

- **Encapsulamiento** (pretende proteger al objeto del entorno): Capacidad de agrupar y condensar en un entorno con límites bien definidos distintos elementos.
  Se hace referencia a encapsulamiento abstracto.

- **Herencia**: Es un tipo de jerarquía aplicada sobre las clases.
  Las clases tienen descendencia y heredan atributos de clases “padres” (superclases).
  Existe la herencia simple y la múltiple.

Objeto

- Una entidad (tangible o intangible) que posee **propiedades** y **acciones** que realiza por sí solo o interactuando con otro objetos.
- Esta entidad encapsula datos (**atributos**) y funciones que las manejan (**métodos**).
- Un objeto se define como una instancia o particularización de una clase.

Cada objeto cumple una **función** y nos permiten interactuar con ellos por medio de formas conocidas (los métodos) modificando su estado (sus atributos).
Si llevamos esto a la programación, **nuestros sistemas se parecen más al mundo real**, posibilitando un mejor entendimiento de los mismos.
Un objeto representa una entidad del mundo real.
Entidades físicas: Vehículo, casa, producto.
Entidades conceptuales: Proceso químico, transacción bancaria.
Entidades de software: Lista enlazada, interfaz gráfica.

Características generales de Objetos

- Se identifica por **nombre único**.
- Posee **estados**. (cuenta de banco activa/inactiva, saldo positivo/negativo)
- Tiene **métodos**. (agregar plata a la cuenta bancaria “depositar()”, extraer())
- Posee un conjunto de **atributos** (tipo de cuenta, id usuario, saldo, características)
- Soportan **encapsulamiento**.
- Tienen un **tiempo de vida** dentro del programa.

Clase

La clase es una colección de objetos que comparten una **estructura** y **comportamiento** común.
Todo objeto o instancia, pertenece a alguna clase.
Una clase representa sólo **una abstracción** (buscar forma más general de identificar un tipo de objeto).

Sintaxis algorítmica:

Class <Identificador de la Clase>
{
private: “Atributos”
……..
public: “metodos”
……..
};

Características de las clases

- Una clase es un **nivel de abstracción alto**. (fijarse en las particularidades que son comunes a todos los objetos de ese conjunto).
- Un objeto es una instancia de la clase.
- Las clases se relacionan entre sí mediante una **jerarquía**, la herencia, en las cuales **la clase hija (subclase) hereda los atributos y métodos de la clase padre (superclase).**
- Los identificadores de las clases deben colocarse en **singular** (clase Animal, Clase Auto, Clase Alumno). (Las clases trabajan con Mayúscula y Los objetos con camelCase).

Codigo

```
class Auto
{
private:
        Int patente;
        char marca[20];     } Atributos
        char color[20];
public:
        Auto();                                 } Constructor, miembro especial de la clase
        Auto(int p, char m[], char c[]...)      } Constructores de la clase.

        Void setPatente(int p);
        int getPatente();

        Void setMarca(char m);      } Metodos
        char * getMarca();

        Void setColor(char c[]);
        char * getColor();

        Los métodos set asignan valores a las propiedades.
        Los métodos get retornan los valores de las propiedades.
}
```

Atributos

Son los datos o variables que caracterizan al objeto y cuyos valores en un momento dado indican su estado.

Un atributo es una característica de un objeto.

Un atributo consta de un **nombre y un valor**. Cada atributo está asociado a un tipo de dato, que puede ser **simple** (entero, real, lógico, caracter, string, objeto) o **estructurado** (arreglo, registro, archivo, lista, etc).

Métodos de acceso

- **Público**: Atributos (o Métodos): **Son accesibles fuera de la clase**. Pueden ser llamados **por cualquier clase**, aun si no está relacionada con ella. Este modo de acceso se puede representar con el símbolo **+**.
- **Privado**: Atributos (o Métodos): **Son accesibles dentro de la implementación** de la clase. Se puede representar con el símbolo **-**.
- **Protegido**: Atributos (o Métodos): **Son accesibles para la propia clase y sus clases hijas** (subclases). Se puede representar con el símbolo **#**.

Metodo

Los **métodos** (funciones) constituyen la secuencia de acciones que implementan las operaciones sobre los objetos. La implementación de los métodos **no es visible** fuera del objeto.

Cada **método** tiene un nombre, cero o más **parámetros** (por valor o por referencia) que recibe o devuelve y un algoritmo con el desarrollo del mismo. (misma definición que una función).

En particular se destaca el método **constructor**, que no es más que el método que **se ejecuta cuando el objeto es creado**.

Este constructor suele tener el **mismo nombre que la clase/objeto**, se puede definir **más de un método constructor**, que normalmente **se diferencian entre sí por la cantidad de parámetros que reciben**.

Creación de objetos y métodos constructivos

Cada objeto o instancia de una clase debe ser creada explícitamente a través de un método u operación especial denominado **constructor**. Por convención el método constructor tiene el mismo nombre de la clase y no se le asocia un modo de acceso (**es público**).

Sintaxis:

```
Public nombreDeLaClase()
{
……..
};
```

Métodos Destructores de objetos

Los objetos que ya no son utilizados en un programa, ocupan inútilmente espacio de memoria, que es conveniente recuperar en un momento dado. Al utilizar lenguajes de programación debemos conocer los método destructores suministrados por el lenguaje y utilizarlos.

Relaciones entre clases

La idea de las clases es que se puedan relacionar entre sí de manera que puedan compartir atributos y métodos sin necesidad de escribirlos.

- **Relación por Herencia**
  Mediante la herencia las instancias de una clase hija (o subclase) pueden acceder tanto a los atributos como a los métodos públicos y protegidos de la clase padre (o superclase)

- **Relación de Agregación**
  Es una relación que representa a los objetos compuestos por otros objetos. Indica objetos que a su vez están formados por otros. El objeto en el nivel superior de la jerarquía es el todo y los que están en los niveles inferiores son sus partes o componentes.

- **Relación de Composición**
  Es la relación entre un elemento muy esencial a tal punto que si desaparece el elemento desaparece de igual forma la clase.

- **Relación por Asociación**
  Se da cuando una clase utiliza atributos o métodos de la otra, se representa una interdependencia pero no necesariamente una es padre y otra hija, ambas pueden ser del mismo tipo.

Diagramas de clase

La representación gráfica de una o varias clases se hará mediante los denominados Diagramas de clase.

Para los diagramas de clase se utilizará la notación que provee el **Lenguaje de Modelación Unificado:**

- Las clases se denotan como rectángulos divididos en tres partes. La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.
- Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y - respectivamente, al lado derecho del atributo (+ público, # protegido, - privado).

Fundamentos del enfoque orientado a objeto

La POO se apoya teóricamente en 5 principios que son la base de este paradigma. Estos principios son:

- **Fundamento 1: La abstracción**
  Expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás.
  Conocemos un objeto viéndolo, sabemos que es sin necesidad de ver su interior, su implementación o su forma de construcción. (Autobús -> placa, color, n de llantas)

- **Fundamento 2: El Encapsulamiento (ocultamiento de información)**
  Es la propiedad que permite ocultar al mundo exterior la representación interna del objeto. El objeto puede ser utilizado, pero los datos esenciales del mismo no son conocidos fuera de él.
  La idea central del encapsulamiento es esconder los detalles y mostrar lo relevante. (La especificación es visible al usuario, mientras que la implementación se le oculta).

- **Fundamento 3: Modularidad**
  La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos.

- **Fundamento 4: Herencia**
  Es el proceso mediante el cual un objeto de una clase adquiere propiedades definidas en otra clase que lo preceda en una jerarquía de clasificaciones.
  La herencia puede ser:
  Simple: cada clase tiene sólo una superclase.
  Múltiple: Cada clase puede tener asociada varias superclases.

- **Fundamento 5: Polimorfismo**
  Permite que un método tenga múltiples implementaciones.
  El polimorfismo operacional o Sobrecarga operacional permite aplicar operaciones con igual nombre a diferentes clases o están relacionados en términos de inclusión.