

# NOTAS DE CLASE ESTRUCTURAS DE DATOS Y ALGORITMOS

{CON EL LENGUAJE DE PROGRAMACIÓN JAVA}

/\* \*\*\*\*\* Jaime E. Montoya M. \*\*\*\*\* \*/

# NOTAS DE CLASE ESTRUCTURAS DE DATOS Y ALGORITMOS

{CON EL LENGUAJE DE PROGRAMACIÓN JAVA}

```
/**  
 * Versión 5.0  
 * Año: 2026  
 * Licencia software: GNU GPL  
 * Licencia doc: GNU Free Document License (GNU FDL)  
 */  
  
public class Author {  
  
    String name = “Jaime E. Montoya M.”;  
  
    String profession = “Ingeniero Informático”;  
  
    String employment = “Docente y desarrollador”;  
  
    String city = “Medellín - Antioquia - Colombia”;  
  
    int year = 2026;  
  
}
```

# Tabla de contenido

<u>Introducción</u>	7
<u>Recursos</u>	8
<u>Lista de abreviaturas</u>	9
<u>Capítulo 0. Preliminares</u>	10
<u>Funciones incorporadas o predefinidas</u>	10
<u>Funciones matemáticas</u>	10
<u>Funciones para la manipulación de cadenas de caracteres</u>	12
<u>Funciones para la conversión de tipos de datos</u>	13
<u>Subprogramas o subalgoritmos</u>	13
<u>Procedimientos y Funciones</u>	15
<u>Funciones</u>	15
<u>Invocar (llamar) una función</u>	16
<u>Parámetros de un subprograma. Parámetros actuales y formales</u>	17
<u>Procedimientos</u>	18
<u>Invocar (llamar) un procedimiento</u>	19
<u>Ámbito de las variables. Variables globales y locales</u>	19
<u>Programación Orientada a Objetos (POO)</u>	22
<u>Clases y objetos</u>	22
<u>Declarar una clase</u>	23
<u>Instanciar una clase</u>	24
<u>Eliminar una instancia de una clase. Recolector de basura</u>	24
<u>Preguntas</u>	28
<u>Ejercicios</u>	28
<u>Capítulo 1. Estructuras de datos fundamentales</u>	30
<u>Estructuras de datos</u>	30
<u>Cadenas de caracteres</u>	31
<u>Cadenas de caracteres en Java</u>	33
<u>Definir cadenas en Java</u>	33
<u>La clase String</u>	34
<u>Longitud de una cadena de caracteres</u>	34
<u>Concatenar cadenas de caracteres</u>	34
<u>Convertir cadenas de caracteres a mayúscula/minúscula</u>	35
<u>Comparar cadenas</u>	35
<u>Conversión de datos a String</u>	36
<u>Obtener la posición de un carácter o subcadena dentro de una cadena</u>	36
<u>Eliminar espacios al inicio y fin de la cadena</u>	36
<u>Devolver un carácter de una cadena a partir de un índice</u>	36
<u>Obtener el valor ASCII de un carácter</u>	37
<u>Obtener un carácter a partir de su valor ASCII</u>	37
<u>Substraer una cadena (subcadena) de otra cadena</u>	37

<u>Arreglos</u>	39
<u>Arreglos unidimensionales: Vectores o listas</u>	39
<u>Representación en memoria</u>	40
<u>Declaración de vectores</u>	40
<u>Acceso a los elementos de un vector</u>	41
<u>Operaciones sobre un vector</u>	43
<u>Buscar un dato</u>	44
<u>Eliminar un dato</u>	45
<u>Insertar un dato</u>	46
<u>Ordenar</u>	46
<u>Arreglos bidimensionales: Matrices o tablas</u>	47
<u>Representación en memoria</u>	48
<u>Declaración de matrices</u>	48
<u>Acceso a los elementos de una matriz</u>	48
<u>Arreglos multidimensionales</u>	50
<u>Representación en memoria</u>	50
<u>Declaración de arreglos multidimensionales</u>	51
<u>Acceso a los elementos de un arreglo multidimensional</u>	51
<u>Registros</u>	54
<u>Crear un registro</u>	55
<u>Crear variables de tipo registro</u>	55
<u>Acceso a los campos de un registro</u>	55
<u>Diferencias con objetos</u>	58
<u>Diferencias con arreglos</u>	58
<u>Arreglos de registros, arreglos de objetos y arreglos paralelos</u>	59
<u>Conjuntos</u>	62
<u>Operaciones con conjuntos</u>	62
<u>Implementación de conjuntos en Python</u>	63
<u>Creación de conjuntos</u>	63
<u>Preguntas</u>	64
<u>Ejercicios</u>	64
<u>Capítulo 2. Listas Ligadas</u>	69
<u>Lista Simplemente Ligada (LSL)</u>	69
<u>Creación de una LSL</u>	70
<u>Asignar y eliminar memoria</u>	71
<u>Creación de una LSL por el inicio</u>	72
<u>Creación de una LSL por el final</u>	73
<u>Operaciones sobre listas</u>	74
<u>Recorrer la lista</u>	74
<u>Buscar un dato en la lista</u>	75
<u>Modificar un dato de la lista</u>	75
<u>Insertar un dato en la lista</u>	76
<u>Eliminar un dato de la lista</u>	77
<u>Lista Simplemente Ligada Circular (LSC)</u>	78

<u>Listas Dblemente Ligadas (LDL)</u>	80
<u>Lista Dblemente Ligada Circular (LDLC)</u>	85
<u>Preguntas</u>	85
<u>Ejercicios</u>	86
<u>Capítulo 3. Pilas y Colas</u>	88
<u>Introducción</u>	88
<u>Pilas</u>	88
<u>Operaciones con pilas</u>	89
<u>Aplicaciones con pilas</u>	89
<u>Colas</u>	91
<u>Operaciones con colas</u>	92
<u>Aplicaciones con colas</u>	92
<u>Preguntas</u>	93
<u>Ejercicios</u>	94
<u>Capítulo 4. Recursividad</u>	95
<u>Preguntas</u>	98
<u>Ejercicios</u>	98
<u>Capítulo 5. Árboles</u>	100
<u>Árboles en general</u>	100
<u>Características y propiedades de los árboles</u>	102
<u>Longitud de camino interno y externo</u>	103
<u>Longitud de Camino Interno (LCI)</u>	104
<u>Longitud de Camino Externo (LCE)</u>	104
<u>Árboles binarios</u>	106
<u>Árboles binarios distintos</u>	108
<u>Árboles binarios similares</u>	108
<u>Árboles binarios equivalentes</u>	109
<u>Árboles binarios completos</u>	109
<u>Árboles binarios degenerados o patológicos</u>	109
<u>Representación de árboles binarios en memoria</u>	111
<u>Operaciones con árboles binarios</u>	112
<u>Creación de un árbol binario</u>	112
<u>Recorrido en árboles binarios</u>	112
<u>Representación de árboles generales como árboles binarios</u>	115
<u>Bosque de árboles</u>	116
<u>Árboles Binarios de Búsqueda (ABB)</u>	116
<u>Preguntas</u>	117
<u>Ejercicios</u>	118
<u>Capítulo 6. Grafos</u>	119
<u>Definiciones fundamentales</u>	119
<u>Vértice</u>	119
<u>Arista</u>	119
<u>Grafo</u>	119
<u>Subgrafo</u>	120

<u>Grafo dirigido</u>	121
<u>Camino en un grafo</u>	122
<u>Longitud de un camino</u>	122
<u>Camino simple</u>	122
<u>Ciclo simple</u>	122
<u>Grafos no dirigidos</u>	123
<u>Estructuras de datos en la representación de grafos</u>	123
<u>Estructura de lista</u>	123
<u>Estructuras matriciales</u>	123
<u>Operaciones básicas con grafos en computación</u>	124
<u>Fuentes y referencias adicionales</u>	127
<u>Textos impresos</u>	127
<u>Páginas web</u>	127

# Introducción

Este documento es un complemento a las clases presenciales y virtuales, y está basado en la bibliografía del curso, así como de otras fuentes adicionales que se indican a lo largo del texto, además de la experiencia del autor en su función docente en las áreas de programación y como desarrollador en distintas empresas del departamento. No se pretende reemplazar los textos guías con este manual, sino servir de ayuda didáctica y apoyo académico a los estudiantes.

La guía incluye, además de los conceptos teóricos, ejemplos, gráficas, desarrollos en clase, y al final de cada capítulo, unas preguntas y ejercicios que permitan reforzar los conceptos y promover la práctica y el estudio de los conceptos vistos.

Al final de este manual, se indican fuentes y referencias adicionales que el estudiante puede consultar. Las notas al pie de página contienen enlaces a lecturas complementarias.

Los ejemplos se ilustran principalmente en pseudocódigo y en Java (o pseudo Java), y quizás ocasionalmente con el uso de otros lenguajes de programación como C/C++, PHP, Python, etc.; en algunos casos se pueden mostrar diferentes técnicas de diagramación, entre ellas, los *flujo*gramas o *diagramas libres* y los *diagramas rectangulares* o de *Nassi-Schneiderman*, aunque estos diagramas no aportan nada nuevo a la teoría, solo sirven para ilustrar algún desarrollo.

Por último, se asume que el estudiante posee conocimientos básicos en el lenguaje Java en cuanto a la sintaxis de las instrucciones comunes como entrada/salida, asignaciones y operaciones aritmético/lógicas, manejo de las estructuras de control, manipulación de cadenas de caracteres, así como la creación y gestión de clases y objetos, entre otros aspectos, lo cual implica que conozca también los principales conceptos de la POO. La unidad 0 repasa algunos de estos conceptos con el fin de nivelar al lector con algunos conocimientos básicos requeridos para el estudio del presente texto.

## Recursos

Adicional a la bibliografía disponible al final del texto y en las notas al pie de página que contienen referencias a otras lecturas, para este curso se debe contar con un equipo de cómputo que soporte los lenguajes de programación como Java, C/C++ y Python entre otros, con el fin de realizar las prácticas. Se asume que el lector ya está familiarizado con dichos lenguajes de programación (en estas notas de clase el énfasis de los desarrollos están sobre Java, pero pueden haber algunos en pseudocódigo y otros lenguajes). El sistema operativo es a gusto del estudiante, puede trabajar bajo Windows, Linux, Mac u otro.

Además de disponer de conexión a Internet, debe contar también con un IDE o editor (se sugiere Visual Studio Code, pero no es obligatorio) para la edición de los programas.

El sitio web [Programiz](#) ofrece una interfaz para el desarrollo de código con la opción de poder seleccionar entre varios lenguajes de programación, tales como Java, Python, C, C++, C#, PHP, R y Javascript, entre otros.

## Listas de abreviaturas

**ASCII:** American Standard Code for Information Interchange

**BD:** Base de Datos

**IDE:** Integrated Development Environment (Entorno de Desarrollo Integrado)

**MCD:** Máximo Común Divisor

**PHP:** PHP Hypertext Preprocessor

**POO:** Programación Orientada a Objetos

**SI:** Sistema de Información

**SO:** Sistema Operativo

# Capítulo 0. Preliminares

Este capítulo comprende un repaso de conceptos ya vistos por los estudiantes, entre los que se incluyen los subprogramas (procedimientos y funciones), así como de las funciones incorporadas en los lenguajes de programación y de tanta utilidad en distintas aplicaciones; también se tratan los conceptos más importantes del paradigma de la POO. Si el lector considera que se siente bien en dicha teoría, puede pasar sin problemas a la unidad 1.

## Funciones incorporadas o predefinidas

Los lenguajes de programación disponen de una serie de funciones para realizar tareas diversas y que tienen como objetivo, además de solucionar ciertos problemas, facilitar a los programadores el tema de construir una funcionalidad. Es así como se dispone de funciones matemáticas, para el tratamiento de cadenas de caracteres, para el manejo de fechas y para otros muchos casos.

## Funciones matemáticas

Algunas operaciones se resuelven rápidamente con los operadores aritméticos que ya conocemos, como la resta o la suma, pero hay una serie de operaciones matemáticas adicionales que se tienen que realizar usando funciones incorporadas. Dichas funciones están optimizadas y de no contar con ellas, se haría necesario construirlas para obtener los cálculos deseados, como el coseno trigonométrico, una raíz cuadrada, etc. En algoritmia se pueden definir una gran cantidad de funciones, análogamente a las existentes en los distintos lenguajes de programación y que se asemejan tanto en número como en nombre. Las más comunes se muestran a continuación.

Función	Descripción	Ejemplo
<code>abs(x)</code>	valor absoluto de x	<code>abs(-8) //devuelve 8</code>
<code>trunc(x)</code>	valor truncado de x	<code>trunc(2.6) //devuelve 2</code>
<code>redondear(x)</code>	valor redondeado de x	<code>redondear(2.6) //devuelve 3</code>
<code>rc(x), raiz(x)</code>	raíz cuadrada de x	<code>raiz(9) //devuelve 3</code>
<code>sen(x)</code>	seno trigonométrico en radianes	<code>sen(0) //devuelve 0</code>
<code>cos(x)</code>	coseno trigonométrico en radianes	<code>cos(0) //devuelve 1</code>
<code>tan(x)</code>	tangente trigonométrica en radianes	<code>tan(0) //devuelve 0</code>
<code>asen(x)</code>	arcoseno de x	
<code>acos(x)</code>	arcocoseno de x	

atan(x)	arcotangente de x	
ln(x)	logaritmo natural de x	ln(1) //devuelve 0
exp(x)	exponencial de x	exp(1) //devuelve 2.718281... (este es el número E)
aleatorio(x, y)	número aleatorio (al azar) entre x e y	aleatorio(1, 20)
pi() (como función), PI (como constante)	número pi	pi() //devuelve 3.14159... PI //devuelve 3.14159...

**Ejemplo 0.1**

Ilustración de funciones matemáticas “incorporadas” que pueden utilizarse en lógica de programación. Se muestra también el uso en PSeInt donde puede verse algunas pequeñas diferencias, dadas las convenciones que se adopten en algoritmia, así como la sintaxis misma del pseudo intérprete; es tarea del programador revisar en cada lenguaje que trabaje la implementación de estas funciones.

Pseudocódigo:

```

Inicio
Reales xx, yy, zz
Imprimir "Número 1:"
Leer xx
Imprimir "Número 2:"
Leer yy
z = abs(xx)
Imprimir "Valor absoluto de ", xx, ": ", z
z = raiz(z)
Imprimir "Raíz cuadrada (raiz) de ", abs(xx), ": ", z
z = rc(abs(xx))
Imprimir "Raíz cuadrada (rc) de ", abs(xx), ": ", z
z = aleatorio(trunc(xx), trunc(yy))
Imprimir "Aleatorio entre ", trunc(xx), " y ", trunc(yy), ": ", z
z = trunc(yy)
Imprimir "Truncar ", yy, ": ", z
z = redondear(yy)
Imprimir "Redondear ", yy, ": ", z
z = sen(xx)
Imprimir "Seno ", xx, ": ", z
z = cos(xx)
Imprimir "Coseno ", xx, ": ", z
z = tan(xx)
Imprimir "Tangente ", xx, ": ", z
z = exp(xx)
Imprimir "Exponencial ", xx, ": ", z

```

```
z = ln(xx)
Imprimir "Ln ", xx, ": ", z
Imprimir "Pi = ", pi()
Fin
```

Pseudo programa PSeInt:

```
Algoritmo FuncionesMatematicas
    Definir xx, yy, zz Como Real
    Imprimir "Número 1:" Sin Saltar
    Leer xx
    Imprimir "Número 2:" Sin Saltar
    Leer yy
    z = abs(xx)
    Imprimir "Valor absoluto de ", xx, ": ", z
    z = raiz(z)
    Imprimir "Raíz cuadrada (raiz) de ", abs(xx), ": ", z
    z = rc(abs(xx))
    Imprimir "Raíz cuadrada (rc) de ", abs(xx), ": ", z
    z = Aleatorio(trunc(xx), trunc(yy))
    Imprimir "Aleatorio entre ", trunc(xx), " y ", trunc(yy), ": ", z
    z = azar(trunc(yy))
    Imprimir "Aleatorio entre 0 y ", trunc(yy), ": ", z
    z = trunc(yy)
    Imprimir "Truncar ", yy, ": ", z
    z = redon(yy)
    Imprimir "Redondear ", yy, ": ", z
    z = sen(xx)
    Imprimir "Seno ", xx, ": ", z
    z = cos(xx)
    Imprimir "Coseno ", xx, ": ", z
    z = tan(xx)
    Imprimir "Tangente ", xx, ": ", z
    z = exp(xx)
    Imprimir "Exponencial ", xx, ": ", z
    z = ln(xx)
    Imprimir "Ln ", xx, ": ", z
    Imprimir "Pi = ", Pi
FinAlgoritmo
```

## Funciones para la manipulación de cadenas de caracteres

Así como existen funciones para el tratamiento matemático, también se dispone de una serie de funciones para la manipulación de cadenas de caracteres. En el siguiente capítulo se hablará más sobre este tema donde se tratan las cadenas de caracteres.

### Nota

En otra subsección más adelante de este mismo capítulo se retoma este tema desde el enfoque del lenguaje Java, una vez repasados algunos conceptos sobre POO para usarlos en los desarrollos.

## Funciones para la conversión de tipos de datos

Otro grupo importante de funciones predefinidas en los lenguajes son las que permiten la conversión de un tipo de dato a otro, operación también conocida como **casteo** de variables. El valor devuelto depende de las reglas establecidas en el lenguaje, así como los parámetros que recibe cada función. En nuestro caso, seguiremos las reglas indicadas como se especifica a continuación; para los casos de PSeInt y lenguajes de programación es probable que haya algunas diferencias.

Función	Descripción	Ejemplo
numero(cadena)	longitud de la cadena	numero("15.5") //devuelve 15.5 numero("hola") //devuelve 0
cadena(numero)	convierte cadena a mayúsculas	cadena(2.6) //devuelve "2.6" cadena("pc") //devuelve "pc"
logico(arg)	convierte un argumento tipo cadena o número a booleano	logico(1) //devuelve Verdadero logico("1") //devuelve Verdadero logico(0) //devuelve Falso logico("0") //devuelve Falso logico(55) //devuelve Verdadero

### Ejemplo 0.3

Pseudo programa PSeInt:

```

Inicio
Imprimir "Número a cadena: " + cadena(85)
Imprimir "Cadena a número a : ", numero("85")
Fin

```

Pseudo programa PSeInt:

```

Algoritmo sin_titulo
    Imprimir "Número a cadena: " + ConvertirATexto(85)
    Imprimir "Cadena a número a : ", ConvertirANumero("85")
FinAlgoritmo

```

## Subprogramas o subalgoritmos

Hasta el momento hemos desarrollado algoritmos en un mismo componente de código que comienza con la instrucción *Inicio* y finaliza con la sentencia *Fin*; ha dicho código se le conoce comúnmente como **programa principal**.

Hemos visto que si algunas partes del código se requieren ejecutar en varias partes, la única alternativa es duplicar el código que hace la respectiva tarea, lo que hace que la solución sea ineficiente desde varias perspectivas. Por una lado, el duplicar código, aumenta el tamaño de los archivos, por tanto la carga en los servidores, transferencias, discos, etc. Si realizamos una modificación sobre la funcionalidad, debemos buscar las partes del código donde se encuentra duplicada y realizar lo mismo, lo cual puede traer problemas de olvidos involuntarios y que no se actualicen todas las partes, o que la copia se realice incompleta, entre otros aspectos, haciendo que el mantenimiento sea una tarea complicada y poco confiable. Las aplicaciones de la vida real son desarrollos que involucran grandes cantidades de horas de trabajo y código asociado; a medida que se requieren funcionalidades, el programa va requiriendo de nuevas variables, y al tener todo un código en un mismo componente, su revisión y manejo de los datos se hace más complejo, pudiendo llevar a errores en cálculos, operaciones, asignaciones, etc. La escalabilidad de los programas es otro problema al que se ve enfrentado un desarrollo de software de este tipo, ya que al tener un mantenimiento complejo, poca fiabilidad en las actualizaciones y cambios, a medida que el código crezca, el problema va a aumentar hasta hacerse inmanejable.

Una solución que propone la algoritmia a los problemas mencionados, es mediante el paradigma de la **programación modular**, el cual consiste en *dividir*<sup>1</sup> un programa en partes más pequeñas llamadas **módulos (subprogramas)**. Este paradigma introduce conceptos como la *legibilidad* y *manejabilidad* del software, esto es, programas más legibles y manejables para los programadores, lo que a su vez permite mejorar la eficiencia de los algoritmos. Cada parte en que se divide el programa se encarga de una tarea específica que puede ser utilizada por otras partes del programa.

Un **subprograma** o **subalgoritmo** es un programa “más pequeño” que se encarga de una tarea específica y que está en el contexto de un programa principal. Esto significa que, aunque un subprograma es también es un programa, tiene varias características que los diferencian:

- Generalmente dependen de un programa principal, aunque también pueden hacer parte de librerías y ser utilizados por diversos “programas principales”. Sin embargo, esto no quita su dependencia para ser utilizado.
- No se ejecutan por sí solos, deben ser invocados o llamados desde un programa principal que lo inicia.
- Generalmente se escriben antes del programa principal, aunque también pueden estar en archivos independientes.
- Tiene un encabezado propio indicando que es un subprograma y puede tener parámetros que ingresan o sacan información de éste.
- Pueden tener un único valor de retorno.
- Cuando el subprograma termina su ejecución, el flujo del programa regresa al punto desde donde se realizó la llamada a éste o a la instrucción siguiente.

---

<sup>1</sup> Es común que distintos autores citen la conocida frase “*divide y vencerás*” en la explicación del tema de subprogramas dadas las ventajas que trae la programación modular en la creación de aplicaciones.

Los subprogramas permiten escribir programas más legibles y manejables, permitiendo que su mantenimiento sea más fácil y que los programas sean escalables; permiten además la creación de *librerías*, que son bibliotecas enteras con distintas funcionalidades que pueden ser usadas por distintos programas. Este paradigma también busca hacer que los programas sean “más simples”, de tal forma que se puedan escribir componentes independientes que se encarguen solo de realizar lo que deben hacer y reduciendo cada funcionalidad a la más mínima expresión posible.

A continuación, vamos a tratar los dos tipos de subalgoritmos que pueden utilizarse en programación.

## Procedimientos y Funciones

Básicamente, los subprogramas se clasifican en dos tipos: **procedimientos y funciones**. Aunque su estructura y funcionamiento es muy similar, tienen algunas características que los diferencian. Aunque es de anotar que algunos lenguajes solo implementan el concepto de *función*, sin embargo, esto no entra en contradicción con el paradigma de la programación modular, ya que éstos a su vez ofrecen métodos de simular una *función* como un *procedimiento*. Veamos más en detalle cada tipo de subprograma.

### Funciones

Matemáticamente, se define una función como una relación especial entre dos conjuntos, en la que a cada elemento del primer conjunto le corresponde solo un elemento del segundo conjunto. Una función se encarga de transformar mediante una regla, un valor en otro. Hay muchas funciones usadas en matemáticas para realizar distintas operaciones, por ejemplo:  $\tan(x)$ ,  $\sen(x)$ ,  $\cos(x)$ ,  $\abs(x)$ , etc., que además se encuentran disponibles no solo en calculadoras científicas, sino también en los lenguajes de programación. Es así como  $\tan(30^\circ) = 0.5$ ,  $\sen(-30^\circ) = 0.5$ ,  $\cos(60^\circ) = 0.5$ ,  $\abs(-9.6) = 9.6$ , etc. En estas funciones podemos ver un **argumento (parámetro)** que recibe la función y que luego lo transforma, según la regla como opere, en otro valor. Por ejemplo, la función *valor absoluto* (*abs*) tiene como fin, a grandes rasgos, convertir un número a positivo, es por ello que al recibir el argumento  $-9.6$ , la función devuelve  $9.6$ .

Un subprograma tipo función, también conocida como **función definida por el usuario**, de manera análoga a las funciones matemáticas, reciben unos argumentos (aunque puede recibir cero parámetros) y devuelve un único valor, el cual es hallado de alguna forma según la tarea (regla) que realice el algoritmo.

Teniendo en cuenta lo comentado anteriormente, podemos declarar el prototipo de una función como se muestra a continuación.

### Sintaxis

```
Funcion nombre_funcion([parámetros formales])[: tipo_funcion]
    Cuerpo de la función
```

```
    Retornar valor_retorno
```

```
FinFuncion
```

Donde:

- nombre\_función: es el identificador único de la función, cuyo nombre sigue las mismas reglas que para el nombre de variables
- parámetros formales: lista de *argumentos de entrada* para la función y que hacen parte de la definición de ésta. Cada parámetro es separado por comas y especificado con su respectivo tipo de dato
- tipo\_función: opcional; indica tipo del valor devuelto por la función
- cuerpo de la función: instrucciones válidas algorítmicas que requiera ejecutar la función
- valor\_retorno: variable o expresión a retornar por la función. Su tipo de dato debe coincidir con el tipo de la función

#### **Nota: Instrucción Retornar**

Como vimos arriba, la sentencia **Retornar** es una instrucción de *bifurcación de control* presente en todos los lenguajes, muy usada en subprogramas, particularmente en funciones, y que se encarga de devolver el control del flujo del programa al punto donde se hizo la llamada a la función. La sentencia Retornar no necesariamente se ubica al final de la función, puede estar en otros puntos que se consideren apropiados. Una vez la sentencia se ejecute, el control retorna al programa que realizó la llamada y el código que haya por debajo de dicha sentencia no se ejecutará.

#### Invocar (llamar) una función

Una función devuelve un único valor, por tanto puede ser llamada asignándola a una variable o incluyéndola en una operación o salida de datos.

La invocación de una función se realiza generalmente por fuera de ella, ya sea desde un “programa principal” u otro subprograma. Si dicho llamado se hace desde la misma función, se dice que es una **función recursiva**. Diversos casos matemáticos se dan como procesos recursivos que pueden ser representados algorítmicamente y llevados a un lenguaje de programación; sin embargo, debe tenerse especial cuidado en la implementación de funciones recursivas, ya que implica una carga mayor en la *pila* de la memoria que puede agotarla, además que conlleva mayor complejidad de desarrollo dada la simplicidad de la solución que ofrece. En este texto no se aborda la *recursividad*, tema que se desarrolla en Estructuras de Datos, donde se tratan problemas que implican obligatoriamente el uso de esta técnica, como en el caso de los algoritmos para *árboles*.

#### **Sintaxis**

```
variable = nombre_funcion([parámetros actuales])
```

#### **Ejemplo 0.4**

Crear una función para calcular el módulo de una división entera. La función recibe dos parámetros y devuelve el residuo de la división entre estos.

Pseudocódigo:

```
// Definición del prototipo de la función módulo()
// para devolver el residuo de una división entera
// con parámetros formales x, y
Funcion modulo(Enteros x, Enteros y): Enteros
    Enteros: z
    z <- x % y
    Retornar z
FinFuncion

// Programa principal
Inicio
Enteros: a, b, c
Leer a, b
Si y <> 0 Entonces
    //Invocación de función con parámetros actuales a, b
    c <- modulo(a, b)
    Imprimir "Módulo: ", c
SiNo
    Imprimir "No se puede dividir por 0"
FinSi
Fin
```

Parámetros de un subprograma. Parámetros actuales y formales

Cuando hablamos de argumentos o parámetros en un subprograma, debemos tener en cuenta las siguientes cuestiones:

- Al invocar un subprograma, hablamos de sus argumentos como **parámetros actuales**, mientras que en el prototipo de la función los llamamos **parámetros formales**.
- Los parámetros actuales y formales de una función deben coincidir en tipo, número, pero no necesariamente en nombre
- Por definición, los parámetros de una función son de **entrada** o pasados por **valor**.
- En un procedimiento los parámetros pueden ser pasados por **valor (entrada)** o por **referencia o dirección**, lo cual significa que también pueden ser de **salida** o **entrada/salida**, esto es, el subprograma puede cambiar los valores originales de los argumentos
- Un **parámetro opcional o por defecto**, es aquel que puede o no, usarse desde la invocación, esto es, especificarse de manera opcional, por lo que en el prototipo del subprograma debe inicializarse con un valor por defecto; se recomienda que estos argumentos estén al final de la lista de parámetros

### Ejemplo 0.5

Crear un programa que reciba dos edades y dos salarios. Se pide calcular el promedio de edades y salarios. Para esto, debe crear una función que calcule los promedios.

Este programa aprovecha una función que evita duplicar código para realizar una operación común: calcular el promedio de dos valores numéricos.

Pseudocódigo:

```
// Definición del prototipo de la función promedio()
// para devolver el promedio de dos valores numéricos
Funcion promedio(Reales x, Reales y): Reales
    Reales: prom
    prom = (x + y) / 2
    Retornar prom
FinFuncion

// Programa principal
Inicio
Reales: ed1, ed2, sal1, sal2, prom
Leer ed1, ed2, sal1, sal2
prom = promedio(ed1, ed2)
Imprimir "Promedio edad: ", prom
prom = promedio(sal1, sal2)
Imprimir "Promedio salario: ", prom
Fin
```

## Procedimientos

En muchos casos necesitamos escribir subprogramas que no devuelvan ningún valor, o que por el contrario, puedan devolver muchos. Para este tipo de situaciones, las funciones se quedan cortas y no permiten dar una solución al problema. Para ello, existen los **procedimientos** (aunque no presentes en todos los lenguajes), otra clase de subprograma que podría verse como el tipo general, siendo las funciones el caso especial de subprogramas, esto es, una función es un tipo especial de procedimiento en el sentido que toda función puede ser escrita con un procedimiento, pero al contrario no es posible. Como se mencionó antes, la estructura de las funciones y procedimientos es muy similar, pero estos últimos no tendrán una sentencia *Retornar*.

### Sintaxis

```
Procedimiento nombre_procedimiento([parámetros formales])
    Cuerpo del procedimiento
FinProcedimiento
```

### Ejemplo 0.6

Parámetros formales por valor y referencia en un prototipo de procedimiento

Pseudocódigo:

```
//Procedimiento Proc1 con 4 parámetros
Procedimiento Proc1(Caracter dato, Enteros a, Enteros Val b, Reales Ref x)
    //Parámetros dato, a, b pasados por valor (de entrada)
```

```
//Si no se indica Val, por defecto el parámetro es por valor
//Parámetro x pasado por referencia (entrada/salida)
Imprimir dato, a, b
a = a + b //valor de a cambia solo localmente
Imprimir a
//x puede tener un valor de entrada y el subprograma cambia su valor
//El programa que llama puede usar el valor de x
x = a * b / 5
Imprimir x
FinProcedimiento

//Programa principal que llama al procedimiento Proc1()
Inicio
Caracter: nombre
Enteros: num1, num2
Reales: numero
Leer nombre, num1, num2
Imprimir nombre, num1, num2
Proc1(nombre, num1, num2, numero)
Imprimir num1, numero
Fin
```

### Invocar (llamar) un procedimiento

Estrictamente hablando, un procedimiento no devuelve valores, aunque puede modificar los parámetros si éstos son pasados por referencia o dirección, lo que nos lleva a decir de manera informal que un procedimiento puede devolver cero, uno o más valores. Como no sabemos si el procedimiento devuelve valores o no, no lo podemos invocar como una función, pero la forma es un poco similar, solo que no lo asignamos a una variable o la incluimos dentro de una expresión.

### Sintaxis

*nombre\_procedimiento([parámetros actuales])*

### Ámbito de las variables. Variables globales y locales

Decimos que las variables que están declaradas en un subprograma, incluyendo sus parámetros, tienen un **ámbito local**; esto quiere decir que dichas variables sólo existen dentro del subprograma y mientras éste se esté ejecutando, una vez termine de correr, estas variables desaparecerán de la memoria. Esto significa que podemos tener nombres de variables con el mismo nombre definidas en subprogramas diferentes sin que entren en conflicto, ya que los lenguajes serán capaces de reconocer el contexto del nombre del campo que se esté utilizando. Las variables definidas en los subprogramas, incluyendo los argumentos formales, son **variables locales** y solo pueden ser usadas dentro de éstos.

Una **variable global** es aquella que se define dentro del programa principal y que por tanto podrá ser accedida desde cualquier parte o subprograma perteneciente al programa. Decimos por tanto que estas variables tienen un **ámbito global**.

### Notas

- En el paradigma de la Programación Modular es común hablar del ámbito de las variables, en particular de las variables globales, sin embargo, esto pierde peso al carecer de sentido en el paradigma de la Programación Orientada a Objetos (POO) con respecto al ámbito global, quién aprovecha la mayoría de conceptos de la programación modular, añadiendo avances significativos a la teoría algorítmica.
- Un parámetro pasado por valor puede ser modificado dentro de un subprograma, pero el cambio no se verá reflejado a nivel global
- Se debe tener especial cuidado con las variables globales, ya que pueden ser modificadas en cualquier parte del programa incluyendo los subprogramas y se pueden dar cambios accidentales
- Un nombre de variable local igual al de una variable global, puede traer resultados inesperados. A medida que los programas crecen, también el número de variables, por lo que hay que tener cuidado en el uso de las variables globales en los lenguajes que las permiten
- Las variables globales se mantienen durante toda la ejecución del programa, por lo que estarán consumiendo recursos, otro aspecto a tener presente a la hora de definir variables de ámbito global

### Ejemplo 0.7

Crear subprogramas para:

- Sumar dos enteros
- Duplicar dos números en procedimientos con argumentos pasados por valor y referencia
- Imprimir información

Pseudocódigo:

```
// Función sumaNumeros para sumar dos números enteros
Funcion sumaNumeros(Enteros x, Enteros z)
    Enteros: sum
    sum <- x + z
    Retornar sum
FinFuncion

// Procedimiento imprimirInfo para mostrar un dato. No devuelve valores
Procedimiento imprimirInfo(Caracter dato)
    Imprimir dato
FinProcedimiento

// Procedimiento duplicar; recibe un parámetro por valor
// y otro por referencia
Procedimiento duplicar(Enteros num1, Enteros Ref num2)
    num1 <- num1 * 2
```

```

num2 <- num2 * 2
Imprimir "Número 1 duplicado: ", num1
Imprimir "Número 2 duplicado: ", num2
FinProcedimiento

// Programa principal desde donde se llaman los subprogramas
Inicio
Enteros: n1, n2, s
Imprimir "Ingrese número 1: "
Leer n1
Imprimir "Ingrese número 2: "
Leer n2
imprimirInfo("Número 1: " + ConvertirATexto(n1))
imprimirInfo("Número 2: " + ConvertirATexto(n2))
s <- sumaNumeros(n1, n2)
imprimirInfo("Resultado suma: " + ConvertirATexto(s))
duplicar(n1, n2)
imprimirInfo("Número 1: " + ConvertirATexto(n1))
imprimirInfo("Número 2: " + ConvertirATexto(n2))
Fin

```

Pseudo Programa PSeInt:

```

// Función sumaNumeros para sumar dos números enteros
Funcion sum <- sumaNumeros(Enteros x, Enteros z)
    sum <- x + z
Fin Funcion

// Procedimiento imprimirInfo para mostrar un dato. No devuelve valores
SubAlgoritmo imprimirInfo(Carácter dato)
    Imprimir dato
FinSubAlgoritmo

// Procedimiento duplicar; recibe un parámetro por valor y otro por
referencia
SubAlgoritmo duplicar(Enteros num1 Por Valor, Enteros num2 Por Referencia)
    num1 <- num1 * 2
    num2 <- num2 * 2
    Imprimir "Número 1 duplicado: ", num1
    Imprimir "Número 2 duplicado: ", num2
FinSubAlgoritmo

// Programa principal desde donde se llaman los subprogramas
Algoritmo ProgramaPrincipal
    Definir a, b Como Caracter
    Definir n1, n2, s Como Entero
    Imprimir "Ingrese a: " Sin Saltar
    Leer a

```

```
    si a no es numero Entonces
        a <- "0"
    FinSi
    n1 <- ConvertirANumero(a)
    Imprimir "Ingrese b: " Sin Saltar
    Leer b
    si b no es numero Entonces
        b <- "0"
    FinSi
    n2 <- ConvertirANumero(b)
    s <- sumaNumeros(n1, n2)
    imprimirInfo("Resultado suma: " + ConvertirATexto(s))
    duplicar(n1, n2)
    imprimirInfo("Valor de a: " + ConvertirATexto(n1))
    imprimirInfo("Valor de b: " + ConvertirATexto(n2))
FinAlgoritmo
```

## Programación Orientada a Objetos (POO)

La **POO** es otro paradigma de programación que surge esencialmente por las limitaciones de otras técnicas de programación; aprovecha la mayoría de conceptos del paradigma de la programación modular para mejorar las técnicas de desarrollo de software, en particular para la construcción de proyectos grandes.

A diferencia de la programación modular que hace énfasis en los algoritmos, la POO se enfoca en los datos, permitiendo modelar un *mundo basado en objetos*, teniendo como referencia los objetos del mundo real: personas, computadores, casas, vehículos, etc; así como objetos no tangibles: música, deporte, educación, etc., esto es, para la POO cualquier cosa es un objeto.

### Clases y objetos

Un **objeto** es un elemento de una **clase**, conocido como **instancia** de la clase. Por ejemplo, podemos tener animales como gatos, aves, peces, etc., los cuales podemos ver como objetos de una clase *animal*; esto porque todos los animales comparten características comunes de todos los seres vivos.

Una clase puede verse como una plantilla o un tipo estructurado de datos que permite crear objetos de dicho tipo. A manera de analogía podemos decir que una variable es a un tipo de dato como un objeto es a una clase.

Una clase define en una misma unidad **propiedades** o **atributos** (variables) que describen a cualquier objeto de la clase, y **métodos** (subprogramas) que manipulan estos atributos (datos de la clase).

## Declarar una clase

Hay una serie de nuevos términos que entran en juego al declarar una clase que permiten especificar cómo se podrá acceder a ésta y como podrá interactuar con otros objetos.

### Sintaxis

```
Clase NombreClase
    Privado|Publico|Protegido tipo_dato propiedad1
    Privado|Publico|Protegido tipo_dato propiedad2
    ...
    Privado|Publico|Protegido tipo_dato propiedadN

    [Publico Metodo constructor([argumentos])
        //Carga por defecto para las instancias que se creen
    FinMetodo]

    [Publico Metodo destructor([argumentos])
        //Acciones a ejecutar al eliminar la instancia
    FinMetodo]

    Publico Metodo asignarPropiedad1(tipo_dato: valor)
        propiedad1 = valor
    FinMetodo

    Publico Metodo asignarPropiedad2(tipo_dato: valor)
        propiedad2 = valor
    FinMetodo

    ...
    Publico Metodo asignarPropiedadN(tipo_dato: valor)
        propiedadN = valor
    FinMetodo

    Publico Metodo obtenerPropiedad1(): [tipo_dato]
        Retornar propiedad1
    FinMetodo

    Publico Metodo obtenerPropiedad2(): [tipo_dato]
        Retornar propiedad2
    FinMetodo

    ...
    Publico Metodo obtenerPropiedadN(): [tipo_dato]
        Retornar propiedadN
```

**FinMetodo**

```
Publico|Privado|Protegido Metodo otroMetodo1([argumentos])[:tipo_dato]
    // Cuerpo del método
    [Retornar valor]
FinMetodo
```

...

```
Publico|Privado|Protegido Metodo otroMetodoN([argumentos])[:tipo_dato]
    // Cuerpo del método
    [Retornar valor]
FinMetodo
```

**FinClase**

## Instanciar una clase

Instanciar una clase consiste en declarar un **objeto** del tipo de dicha clase.

### Sintaxis

```
NombreClase nombreObjeto = Nuevo NombreClase
```

## Eliminar una instancia de una clase. Recolector de basura

Esta operación consiste en eliminar (destruir) el objeto y las referencias a éste, ahorrando con esto recursos de la máquina y posibles errores debido a las referencias a la memoria que aún quedan presentes. Los lenguajes modernos eliminan automáticamente los objetos que dejan de ser usados y son enviados a **recolectores de basura**, por lo que esta operación es opcional y haciendo que el programador no se tenga que preocupar por la gestión de la memoria.

Un **recolector de basura (garbage collector)** es un mecanismo implícito de gestión de la memoria implementado en la mayoría de lenguajes de programación orientados a objetos, ya sean puros o híbridos. Sin que se requiera eliminar de forma explícita un objeto de la memoria, el recolector de basura se encarga de “limpiar la basura”, o en otras palabras. borrar de la memoria aquellos objetos en desuso dentro del programa.

Algunos lenguajes que disponen del recolector de basura son: Python, PHP, Java, Javascript, Ruby, C# y Visual Basic .NET, entre otros. En C++ (la “versión” de C orientada a objetos) no existe este mecanismo, por lo que los objetos deben ser destruidos de forma explícita.

En caso de querer eliminar el objeto de forma explícita o en aquellos lenguajes que no disponen de un recolector de basura, existe una sentencia que permite borrarlos (destruirlos) de la memoria. A continuación se muestra la forma general en algoritmia.

## Sintaxis

```
nombreObjeto = Nulo
```

### Ejemplo 0.8

Crear una clase (super clase) llamada *Persona* con las propiedades *nombre* y *edad*. La clase debe contener los métodos para agregar datos, devolverlos y otro que determine si la persona es mayor de edad. Realizar además lo siguiente:

- Crear dos objetos de tipo Persona e ingresar la información respectiva
- Mostrar los datos de las personas
- Determinar cuál de éstas personas es mayor
- A partir de la clase Persona, crear otra clase para gestionar empleados llamada *Empleado* con los atributos *salario mínimo* y *salario*. Además de los métodos de asignación y devolución, crear otro para determinar si un empleado gana el salario mínimo. El constructor de la clase debe permitir cargar opcionalmente el salario mínimo.

Pseudocódigo:

```
// Super clase Persona
Clase Persona
    // Atributos de clase
    Privado Caracter nombre
    Privado Entero edad

    Publico Metodo constructor()
        //Carga por defecto
        FinMetodo

    Publico Metodo destructor()
        //Acciones al destruir el objeto
        FinMetodo

    Publico Metodo asignarNombre(Caracter: nom): Ninguno
        nombre = nom
        FinMetodo

    Publico Metodo obtenerNombre(): Caracter
        Retornar nombre
        FinMetodo

    Publico Metodo asignarEdad(Enteros: ed): Ninguno
        edad = ed
        FinMetodo

    Publico Metodo obtenerEdad(): Entero
        Retornar edad
```

```
FinMetodo

Publico Metodo mayorEdad(): Logico
    Si edad >= 18 Entonces
        Retornar Verdadero
    SiNo
        Retornar Falso
    FinSi
FinMetodo
FinClase

// Clase derivada o heredada: la clase Empleado hereda de la clase Persona
Clase Empleado HeredaDe Persona
    Privado Real salarioMinimo
    Privado Real salario

    Publico Metodo constructor(salMin = 0): Ninguno
        salarioMinimo = salMin
    FinMetodo

    Publico Metodo destructor(): Ninguno
        //Acciones al destruir el objeto
    FinMetodo

    Publico Metodo asignarSalarioMinimo(Real: salMin): Ninguno
        salarioMinimo = salMin
    FinMetodo

    Publico Metodo obtenerSalarioMinimo(): Real
        Retornar salarioMinimo
    FinMetodo

    Publico Metodo asignarSalario(Real: sal): Ninguno
        salario = sal
    FinMetodo

    Publico Metodo obtenerSalario(): Real
        Retornar salario
    FinMetodo

    Publico Metodo ganaSalarioMinimo(): Logico
        Si salarioMinimo = salario Entonces
            Retornar Verdadero
        SiNo
            Retornar Falso
        FinSi
```

```
FinMetodo
FinClase

Inicio
Entero: edad
Caracter: nombre
Real: salario

Persona per1 = Nuevo Persona()
Persona per2 = Nuevo Persona()
Empleado emp = Nuevo Empleado(1000000)

Leer nombre, edad
per1.asignarNombre(nombre)
per1.asignarEdad(edad)

Leer nombre, edad
per2.asignarNombre(nombre)
per2.asignarEdad(edad)

Si per1.mayorEdad() Entonces
    Imprimir per1.obtenerNombre(), " es mayor de edad"
SiNo
    Imprimir per1.obtenerNombre(), " es menor de edad"
FinSi
Si per2.mayorEdad() Entonces
    Imprimir per2.obtenerNombre(), " es mayor de edad"
SiNo
    Imprimir per2.obtenerNombre(), " es menor de edad"
FinSi
Si per1.obtenerEdad() > per2.obtenerEdad() Entonces
    Imprimir per1.obtenerNombre(), " es mayor que ", per2.obtenerNombre()
SiNo
    Imprimir per2.obtenerNombre(), " es mayor que ", per1.obtenerNombre()
FinSi

Leer nombre, edad, salario
emp.asignarNombre(nombre)
emp.asignarEdad(edad)
emp.asignarSalario(salario)
Imprimir "Nombre empleado: ", emp.obtenerNombre()
Imprimir "Edad empleado: ", emp.obtenerEdad()
Imprimir "Salario empleado: ", emp.obtenerSalario()
Si emp.obtenerSalarioMinimo() > 0 Entonces
    Imprimir "Salario mínimo actual: ", emp.obtenerSalarioMinimo()
    Imprimir "Salario empleado actual: ", emp.obtenerSalario()
    Si emp.ganaSalarioMinimo() Entonces
```

```

        Imprimir "Salario igual al mínimo"
SiNo
        Imprimir "Salario diferente al mínimo"
FinSi
SiNo
        Imprimir "No ha indicado el salario mínimo"
FinSi
Fin

```

## Preguntas

## Ejercicios

1. Cree un procedimiento para intercambiar el valor de dos variables. Las variables deben ser ingresadas desde un procedimiento. Mostrar los valores antes y después del intercambio en otro procedimiento.
2. Se tiene un grupo de estudiantes del TdeA, de los cuales se lee la nota del parcial y del final, así como el código de éstos. El último código que se lee es un registro centinela con código igual a “\*”. Cree un subprograma para hallar el promedio de notas obtenido en el parcial, en el final y general.
3. Ingrese un número entero por teclado. Cree una función que determine si dicho número pertenece a la serie de Fibonacci
4. Cree un programa que solicite cuatro datos numéricos para armar una dato para fechas: día de la semana entre 1 y 7 (el domingo es el día 1, el lunes el 2, etc.), día del mes entre 1 y 31, número del mes entre 1 y 12 (el 1 es el mes enero, el 2 febrero, etc.) y el año. Cree una función que devuelva una fecha en un formato similar a este: Jueves, 26 de Octubre de 2023, si los datos son: 5, 26, 10, 2023
5. Dado un valor real entre  $0$  y  $2\pi$ , cree dos funciones para calcular el seno, el coseno y tangente trigonométricos en radianes usando la serie de Taylor para 100 términos y compare los resultados con una calculadora y aumente el valor de  $n$  para que observe qué pasa cuando  $n$  se hace grande, es decir, cuando  $n \rightarrow \infty$ : Permitir al usuario poder ingresar ángulos en grados:

$$\text{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

$$\tan(x) = \text{sen}(x)/\cos(x)$$

6. Dado un valor real  $x$ , cree una función para calcular su exponencial usando la serie de Taylor para 100 o más términos:  $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$  Compare los resultados con una calculadora y aumente el valor de  $n$  para que observe qué pasa cuando  $n$  se hace grande, es decir, cuando  $n \rightarrow \infty$ .

7. La función lineal está expresada por  $y = mx + b$ , con  $m$ ,  $b$  números reales. Cree un programa usando funciones para generar una tabla de valores. Grafique esta función por pantalla de acuerdo a los valores suministrados por el usuario.
8. Crear un programa para conversión de coordenadas. Debe tener un menú para especificar el tipo de conversión y los datos a ingresar, así como un procedimiento que convierta coordenadas polares ( $r$ ,  $\theta$ ) a coordenadas cartesianas:  $x = r\cos(\theta)$ ,  $y = r\sin(\theta)$ , y otro procedimiento que convierta de coordenadas cartesianas a polares:  $r = \sqrt{x^2 + y^2}$ ;  $\theta = \tan^{-1} \frac{y}{x}$
9. Leer el nombre y salario básico de  $n$  empleados. Cree una función que calcule el aporte a salud (4%) y a pensión (4%) y un procedimiento para mostrar el salario neto
10. Dados dos números enteros positivos; cree una función para multiplicarlos como una suma sucesiva. Ejemplo:  $2 * 3 = 2 + 2 + 2 = 3 + 3$ .
11. Dados dos números enteros positivos; cree una función para elevar el primero al segundo como una multiplicación sucesiva. Ejemplo:  $2 ^ 3 = 2 * 2 * 2$ .
12. Se tiene un grupo de empleados en una empresa, de los cuales se lee el salario y el código. Cree un subprograma para incrementar el salario de cada empleado en un 10%. El último código que se lee es un registro centinela con código igual a “\*\*”.

# Capítulo 1. Estructuras de datos fundamentales

Este capítulo comprende un repaso de conceptos que ya han sido vistos por los estudiantes en asignaturas como Lógica de Programación y Programación Orientada a Objetos, por lo que también puede considerarse un repaso y/o profundización en algunos aspectos relacionados con arreglos. Si el lector considera que se siente bien en dicha teoría, puede pasar sin problemas al capítulo 2, no sin antes dar una revisión general al material del capítulo con el fin de que esté seguro que ha estudiado anteriormente los temas aquí tratados.

## Estructuras de datos

Son colecciones de datos que pueden ser categorizadas por su organización y las operaciones que se pueden hacer sobre ellas. Podemos decir que una **estructura de datos** es un **tipo de dato complejo** capaz de almacenar más de un dato al tiempo, a diferencia de los tipos datos simples o primitivos, los cuales soportan un solo dato a la vez.

Ya vimos los **tipos de datos primitivos**: numéricos (reales, enteros), lógicos y caracteres. En los **tipos estructurados**, tenemos varios tipos en dos categorías:

### Estáticas

- Cadenas de caracteres
- Arreglos: vectores, matrices, arreglos multidimensionales
- Registros
- Conjuntos
- Archivos
- Pilas (tratadas como vectores)
- Colas (tratadas como vectores)

### Dinámicas

- Pilas (tratadas como listas ligadas)
- Colas (tratadas como listas ligadas)
- Listas ligadas
- Árboles y grafos

Un tipo de dato simple o primitivo significa que no está compuesto de otros tipos o estructuras de datos, mientras que un tipo de dato compuesto está basado en los tipos de datos primitivos, esto es, se forman a partir de los datos simples, así como posiblemente de otros datos compuestos.

Una **estructura de datos estática** es aquella que reserva un espacio fijo de memoria al declararse; sin embargo, los lenguajes modernos permiten un manejo dinámico de la memoria para los arreglos y otras estructuras que se han considerado estáticas.

Una **estructura de datos dinámica** no reserva un espacio determinado, sino que se basa en solicitar memoria a medida que lo requiera; para ello se apoya en el uso de **punteros**, un

tipo especial de tipo de dato presente en algunos lenguajes con los cuales se puede acceder a las direcciones de memoria.

## Cadenas de caracteres

Una **cadena de caracteres** es un conjunto de símbolos disponibles en un idioma, lengua, medio/dispositivo (como un computador), etc. y que se compone de letras (a - z; A - Z y posiblemente los caracteres de otros alfabetos) , caracteres numéricos (0 - 9) y símbolos especiales (+, -, ¿, }, #, ", ...). A nivel de programación, una cadena se considera una **estructura de datos lineal estática** que almacena una secuencia de caracteres y se puede considerar como una **matriz** de caracteres; de ahí que lenguajes como C/C++ las traten como **vectores** (arreglos unidimensionales) en donde cada carácter ocupa una posición en el vector<sup>2</sup>.

Cada lenguaje de programación ofrece un conjunto de funciones para tal fin, ya que la manipulación de texto es algo común y recurrente en las aplicaciones, y que en los inicios de la informática y hasta muchos años después, el tratamiento de las cadenas de caracteres fue un asunto que traía bastantes dificultades a los programadores. Las funcionalidades de los lenguajes actuales facilitan la labor del desarrollador permitiendo la correcta manipulación de las cadenas de texto y concentrándose en la lógica del negocio que requiere solucionar y no en temas que deben resolver los lenguajes de programación.

Algorítmicamente, se definen las funciones más relevantes para ilustrar el tratamiento de cadenas, aunque los lenguajes disponen de una lista extensa para la manipulación de texto, cada una con sus propias especificaciones de implementación, pero en general, realizando las mismas funciones. A nivel de la lógica de programación, podemos definir las siguientes funciones que se muestran en la siguiente tabla y que son de común implementación.

### Notas

- Cada carácter de una cadena tiene una posición asociada dentro de ella. Muchos lenguajes acostumbran a tomar la primera posición, esto es, el lugar donde está el primer carácter, como cero, pero también se puede tomar desde uno. En este curso en algunos casos algorítmicos, las cadenas se toman desde la posición uno.
- La mayoría de lenguajes son sensibles a los caracteres, esto es, distinguen entre mayúsculas y minúsculas, por lo que la variable “a” será diferente de la variable “A”. En este curso siempre consideraremos la sensibilidad de los caracteres.

Función	Descripción	Ejemplo
longitud(cadena)	longitud de la cadena	longitud("hola") //devuelve 4
mayusculas(cadena )	convierte cadena a mayúsculas	mayusculas("abc") //devuelve "ABC"

---

<sup>2</sup> El tema de vectores, matrices y arreglos en general se tratará más adelante en este capítulo.

minusculas(cadena )	convierte cadena a minúsculas	minusculas("ABC") //devuelve "abc"
subCadena(cadena, inicio, num_caract)	extrae una cadena de otra comenzando desde <i>inicio</i> , extrayendo <i>num_caract</i> caracteres	subCadena("Hoy es martes", 5, 2) //devuelve "es" (comienza en 1)
concatenar(cad1, cad2, ,,, cadN)	concatena (une) cad1 con cad2, ..., cadN	concatenar("Pedro, ", " ", "Gil") // devuelve "Pedro Gil" (hace lo mismo que "Pedro" + " " + "Gil")
caracterEn(cad, índice)	devuelve el carácter en posición índice. de la cadena cad. Si el índice es inválido, devuelve Nulo.	caracterEn("Hola", 2) //Devuelve o
caracterASCII(car )	Devuelve el valor ASCII del carácter car.	caracterASCII("a") //Devuelve 97
ASCIIICaracter(valor_ASCII)	Devuelve el carácter correspondiente al valor ASCII especificado	ASCIIICaracter(97) //Devuelve "a"
indiceEn(cad, sub_cad[, pos_ini[, pos_fin]])	Devuelve la posición de sub_cad dentro de cad; la búsqueda inicia en pos_ini y finaliza en pos_fin. Si se omiten los límites de búsqueda, comienza desde la primera posición hasta la última. Si la subcadena no encuentra, la función devuelve 0 (o -1 dependiendo en qué posición comienzan las cadenas)	indiceEn("Hola", "la", 1) //Devuelve 3
eliminaEspacios(cad)	Elimina los espacios al inicio y fin de la cadena cad	cad " Hoy es un día " eliminaEspacios(cad) //devuelve "Hoy es un día"

**Nota**

La comparación de cadenas se realiza con los operadores relacionales conocidos (= o ==, <> o !=, >, <, >=, <=), teniendo en cuenta que dicha comparación es alfabética en donde se tiene presente el valor ASCII de los caracteres.

**Ejemplo 1.1**

Uso de las funciones para la manipulación de cadenas de caracteres. También se presenta el pseudo programa en PSeInt y la forma cómo esta herramienta implementa las funciones.

Pseudocódigo:

```
Inicio
    Caracter texto, z
    Imprimir "Ingrese un texto: "
    Leer texto
    Imprimir "Texto ingresado: ", texto
    Imprimir "Cantidad de caracteres: ", longitud(texto)
    Imprimir "Cadena en mayúscula: ", mayusculas(texto)
    Imprimir "Cadena en mayúscula: ", minusculas(texto)
    z = subCadena(texto, 1, 3)
    Imprimir "Cadena extraída: ", z
    Imprimir "Concatenar cadenas: ", concatenar(texto, z)
Fin
```

Pseudo programa PSeInt:

```
Algoritmo FuncionesCadenas
    Definir texto, z Como Caracter
    Imprimir "Ingrese un texto: " Sin Saltar
    Leer texto
    Imprimir "Texto ingresado: ", texto
    Imprimir "Cantidad de caracteres: ", Longitud(texto)
    Imprimir "Cadena en mayúscula: ", Mayusculas(texto)
    Imprimir "Cadena en mayúscula: ", Minusculas(texto)
    z = Subcadena(texto, 1, 3)
    Imprimir "Cadena extraída: ", z
    Imprimir "Concatenar cadenas", Concatenar(texto, z) //solo 2 cadenas
FinAlgoritmo
```

## Cadenas de caracteres en Java

Java trata las cadenas de caracteres como **objetos** de la clase **String**, por lo que es posible acceder a sus métodos para manipular éstas<sup>3</sup>.

Definir cadenas en Java

Se puede utilizar el constructor de la clase String o usar directamente un literal de cadena. Una cadena también puede ser leída desde el teclado usando el escáner, por ejemplo.

### **Ejemplo 1.2**

Ilustración en la creación de cadenas en Java

Pseudo programa Java:

---

<sup>3</sup> La clase String de Java es una clase inmutable y no posee atributos, sólo dispone de métodos.

```
String literal = "Esto es un literal de cadena";  
  
String keyboardInput = input.next();  
  
String object = new String("Invocando al constructor de la clase String");
```

## La clase String

La clase String de Java forma parte del paquete `java.lang` y es una clase inmutable que no posee atributos, sólo dispone de métodos; esto significa que una vez creado un objeto `String`, su valor no puede modificarse.

Algunos de los métodos más comunes se ilustran a continuación.

### Longitud de una cadena de caracteres

La longitud de una cadena especifica el número de caracteres de ésta. El método `length()` devuelve un entero indicando la longitud de la cadena.

#### **Ejemplo 1.3**

Uso del método `length`.

Pseudo programa Java:

```
String firstName = "Flavio";  
int length = firstName.length();  
System.out.print(length); //Muestra: 6
```

### Concatenar cadenas de caracteres

Hay dos formas de realizar esta operación consistente en unir cadenas en Java:

#### **Con el operador + (recomendado):**

#### **Ejemplo 1.4**

Concatenación de cadenas

Pseudo programa Java:

```
String firstName = "Flavio ";  
String lastName = "Gil";  
String fullName = firstName + lastName;  
System.out.print(fullName); //Muestra: Flavio Gil
```

#### **Con el método concat:**

Pseudo programa Java:

```
String firstName = "Flavio ";
```

```
String lastName = "Gil";
String fullName = firstName.concat(lastName);
System.out.print(fullName); //Muestra: Flavio Gil
```

### **Nota**

Algunos desarrollos (códigos) se presentan de forma abreviada obviando todo el código asociado al presentado con el fin de hacer énfasis en la lógica e implementación; en estos casos podemos considerar que estamos trabajando con código *pseudo Java*, que no afectará la lógica del problema.

Convertir cadenas de caracteres a mayúscula/minúscula

Utilizamos los métodos `toUpperCase()` para convertir a mayúsculas y `toLowerCase()` para convertir a minúsculas, respectivamente:

### **Ejemplo 1.5**

Convertir a mayúscula y minúscula.

Pseudo programa Java:

```
String firstName = "Flavio ";
String lastName = "Gil";
String fullName = firstName.concat(lastName).toUpperCase();
System.out.print(fullName); //Muestra: FLAVIO GIL
fullName = firstName.concat(lastName).toLowerCase();
System.out.print(fullName); //Muestra: flavio gil
```

Comparar cadenas

Java dispone de varios métodos para comparar dos cadenas; estos son:

### **Método equals**

El método permite comparar dos cadenas devolviendo `true` si son iguales o `false` en caso contrario: `cadena1.equals(cadena2)`.

### **Método equalsIgnoreCase**

Método similar a `equals`, pero sin tener en cuenta las mayúsculas y minúsculas: `cadena1.equalsIgnoreCase(cadena2)`.

### **Método compareTo**

Devuelve cero (0) si las dos cadenas son iguales. Devuelve un valor menor a cero (< 0) si la primera cadena es alfabéticamente menor que la segunda ó un valor mayor a cero (> 0) si la primera cadena es alfabéticamente mayor que la segunda: `cadena1.compareTo(cadena2)`.

### **Método compareToIgnoreCase**

Similar a `compareTo`, pero sin tener en cuenta las mayúsculas y minúsculas: `cadena1.compareToIgnoreCase(cadena2)`.

### Conversión de datos a String

El método `valueOf()` es un método estático que convierte un argumento de cualquier tipo de dato a String: `String.valueOf(cadena)`.

### Obtener la posición de un carácter o subcadena dentro de una cadena

En muchas ocasiones necesitamos saber si un carácter o una subcadena se encuentran en una cadena; el método `indexOf()` permite encontrar la posición de la primera ocurrencia del carácter o subcadena dentro de la cadena dada. Si la búsqueda es infructuosa, el método devuelve -1.

#### Sintaxis

`indexOf(subCadena[, posiciónInicial])`

Donde:

**subCadena**: carácter o subcadena a buscar en la cadena dada

**posiciónInicial**: argumento opcional que permite especificar en dónde inicia la búsqueda dentro de la cadena

#### Ejemplo 1.6

Aplicación del método `indexOf`.

Pseudo programa Java:

```
String str1 = "Feria de las flores Medellín";
str2 = "Medellín";
int pos = str1.indexOf(str2, 0);
System.out.print(pos); //Muestra: 20
```

### Eliminar espacios al inicio y fin de la cadena

Java dispone de la función `trim()` para eliminar los espacios en blanco que se encuentre tanto al inicio como al final de una cadena de caracteres:

```
System.out.println("Uso de trim():" + " Hola ".trim() + " a todos");
// Imprime: Uso de trim():Holaa todos
```

### Devolver un carácter de una cadena a partir de un índice

El método `charAt` permite obtener el carácter que se especifique según su posición dentro de la cadena mediante un índice; recordemos que las cadenas en Java inician en la posición 0 y finalizan en la posición `cadena.length() - 1`; así, para obtener un carácter de una cadena, podemos escribir: `cadena.charAt(índice)`.

### Obtener el valor ASCII de un carácter

Java no dispone de una función directa para obtener el valor ASCII de un carácter, pero basta con convertir éste a entero, por lo que para realizar esta operación es necesario que el tipo de dato de la variable sea `char` para que pueda ser convertida al tipo `int`. Recuerde que un dato tipo `char` se encierra entre apóstrofos:

```
System.out.println("Ascii '/'：" + (int) '/');
System.out.println("Ascii：" + (int)"hola".charAt(3));
```

### Obtener un carácter a partir de su valor ASCII

Java no dispone de una función directa para obtener el carácter a partir de su valor ASCII, pero basta con convertir éste a carácter (`char`), por lo que para realizar esta operación es necesario que el tipo de dato de la variable sea `int` para que pueda ser convertida al tipo `char`. Recuerde que un dato tipo `char` se encierra entre apóstrofos:

```
System.out.println("Carácter ASCII(97)：" + (char)97); // ASCII(97)：a
```

### Substraer una cadena (subcadena) de otra cadena

Otra operación común es la sustracción de cadenas de otra cadena. Java dispone del método `substring()` para substraer subcadenas a partir de una cadena dada.

#### Sintaxis

```
substring(inicio[, fin])
```

Donde:

**inicio**: posición a partir de la cual se sustraerá la subcadena de la cadena dada

**fin**: opcional. Argumento que permite especificar hasta dónde se sustrae la cadena. Si no se especifica, se sustrae hasta el final de la cadena. Este índice debe indicar uno más después del último carácter de la subcadena.

#### Ejemplo 1.7

Substraer una subcadena de una cadena dada

Pseudo programa Java:

```
String str1 = "Feria de las flores Medellín";
String str2 = str1.substring(13, 19);
System.out.print(str2); //Muestra: flores
```

#### Ejemplo 1.8

Determinar si una frase es un palíndromo<sup>4</sup>. Aplicar los métodos de cadenas de caracteres para solucionar este problema.

Programa Java:

```
package com.packages.strings;

public class Strings
{
    private String text;

    public Strings()
    {
        this.text = "";
    }

    public void setText(String str)
    {
        this.text = str;
    }

    public String getText()
    {
        return text;
    }

    public String palindrome(String text)
    {
        String message = "";
        text = text.toLowerCase();
        text = this.deleteSpaces(text);
        message = this.compareCharacters(text) ?
            " es palíndromo" : " no es palíndromo";
        return message;
    }

    public String deleteSpaces(String text)
    {
        int i = 0;
        text = text.trim();
        while (i < text.length()) {
            if (text.substring(i, i + 1).equals(" "))
                text = text.substring(0, i) +
                    text.substring(i + 1, text.length());
    }
}
```

---

<sup>4</sup> Un palíndromo es una frase que puede leerse igual de izquierda a derecha y de derecha a izquierda; por ejemplo: amad a la dama, dábale arroz a la zorra el abad, anita lava la tina y anilina, entre otras.

```

        } else {
            i++;
        }
    }
    return text;
}

public boolean compareCharacters(String text)
{
    boolean sw = true; //Supuesto: text es palíndromo
    int i = 0;
    while (i < text.length() / 2 && sw) {
        if (text.substring(i, i + 1).equals(
            text.substring(text.length() - i - 1, text.length() - i)
        )) {
            i++;
        } else {
            sw = false;
        }
    }
    return sw;
}
}

```

## Arreglos

Un **arreglo (array)** es un conjunto finito y ordenado de elementos homogéneos, esto significa que cada elemento puede ser identificado y que la información es del mismo tipo, aunque algunos de los nuevos lenguajes permiten tener arreglos con información heterogénea.

En los lenguajes de programación existen **estructuras de datos** especiales que nos sirven para guardar información más compleja que simples variables. Una estructura típica en todos los lenguajes son los **arreglos**, y que es generalmente la primera estructura de datos en materia de estudio. Existen varios tipos de arreglos: **unidimensionales** (vectores), **bidimensionales** (matrices) y **n-dimensionales** ( $n > 2$ ). Sin embargo, su implementación depende del lenguaje, pues aunque teóricamente ya hay un amplio desarrollo al respecto, no todas estas “máquinas” implementan un uso extendido en su manejo.

### Arreglos unidimensionales: Vectores o listas

Los **vectores** permiten almacenar varios valores del mismo tipo (*información homogénea*, aunque los lenguajes modernos también permiten tener vectores que guardan *datos heterogéneos*) utilizando un mismo **nombre de variable** e identificando cada elemento con un **índice** que representa la **posición** de éste en el *vector*, el cual va desde **uno** hasta el

**total de elementos -tamaño-** (algunos lenguajes toman la primera posición como cero y la última como el total de elementos del arreglo menos uno).

Matemáticamente, un vector se representa con sus elementos separados por comas, ya sea entre corchetes o entre paréntesis:

```
vec1 = [2, 3, 4, -5, 0]
```

```
vec2 = (b, a, d, z)
```

Un vector está compuesto de una serie de espacios consecutivos de memoria a los que se accede por medio de un nombre y un índice entero y que puede representarse gráficamente.

### Representación en memoria

A nivel informático, los arreglos de una, dos y tres dimensiones se pueden representar gráficamente. Un vector se representa como una secuencia de cajones consecutivos, cada uno asociado a un índice y al nombre del arreglo.

vec1 (n = 5)

2	3	4	-	0
5				

vec2 (n = 4)

"b"
"a"
"d"
"z"

Figura 1.1. Representación gráfica de arreglos unidimensionales: vector fila y vector columna, respectivamente

### Declaración de vectores

Los vectores son arreglos de una dimensión, por tanto usamos un valor para especificar el tamaño de éste, es decir, la cantidad de espacios de memoria que contendrá. Para esto indicamos el tipo de dato seguido del nombre del vector, y seguido de éste y entre corchetes, el tamaño que tendrá, esto es, la longitud o número de posiciones.

### Sintaxis

En pseudocódigo:

```
tipo_de_dato: nombre_vector[tamaño]
```

En Java:

```
tipo_de_dato nombre_vector[] = new tipo_de_dato[tamaño]
```

Donde *tamaño* es un valor entero que define la longitud del vector. Debe tenerse en cuenta las limitantes de cada lenguaje de programación a la hora de asignar dicho valor.

Acceso a los elementos de un vector

Los arreglos se acceden elemento por elemento, esto significa que debemos especificar el nombre de éste y las dimensiones respectivas. Para el caso de un vector, indicamos el nombre del vector y seguido de éste y entre corchetes, la posición (índice) a la que queremos acceder, ya sea para guardar un dato allí, mostrarlo o usarlo en una operación.

### Sintaxis

```
nombre_vector[posición]
```

Dónde *posición* es un número entero entre 1 y el total de elementos del vector. En el lenguaje Java la primera posición está determinada por el índice 0.

### Ejemplo 1.9

Crear un vector de 5 posiciones. Agregar dos elementos en las dos primeras posiciones y luego sume estos valores y guárdalos en la tercera posición. Imprima estas posiciones del vector.

Pseudocódigo:

```
Inicio
    Enteros: vector[5]
    Leer vector[1]
    vector[2] = 4
    vector[3] = vector[1] + vector[2]
    Imprimir vector[1], vector[2], vector[3]
Fin
```

### Ejemplo 1.10

Crear un vector de *t* elementos de máximo de 30 posiciones con datos aleatorios entre 1 y 50, mostrar sus datos, hallar la suma y el mayor de éstos.

Para hallar el mayor o el menor en un vector, partimos del supuesto que el primer elemento es el que cumple la condición y a partir del segundo comenzamos a comparar.

Se presenta la solución en pseudocódigo y PSeInt.

Pseudocódigo:

```
Funcion mayorDatoVector(Enteros: vec, Enteros: n): Enteros
    Enteros: i, mayor
    mayor = vec[1] // Supuesto: el mayor dato está en la posición 1
```

```

Para i = 2 Hasta n Con Paso 1 Hacer
    Si vec[i] > mayor Entonces
        mayor = vec[i]
    FinSi
FinPara
Retornar mayor
FinFuncion

Funcion sumaVector(vec, n)
    Enteros: i, s
    s = 0
    Para i = 1, n, 1
        s = s + vec[i]
    FinPara
    Retornar s
FinFuncion

Procedimiento llenarVector(vec, n)
    Para i = 1 Hasta n Hacer
        vec[i] = Aleatorio(1, 50)
    FinPara
FinProcedimiento

Procedimiento mostrarVector(vec, n)
    Enteros: i
    Para i = 1 Hasta n Hacer
        Imprimir vec[i]
    FinPara
FinProcedimiento

Inicio
Enteros: V[30], t, i
Leer t
llenarVector(V, t)
mostrarVector(V, t)
Imprimir "Suma vector: ", sumaVector(V, t)
Imprimir "Mayor dato vector: ", mayorDatosVector(V, t)
Fin

```

#### Pseudo Programa PSeInt:

```

Funcion mayor = mayorDatosVector(vec, n)
    Definir i, mayor Como Entero
    mayor = vec[1] // Supuesto: el mayor dato está en la posición 1
    Para i = 2 Hasta n Con Paso 1 Hacer
        Si vec[i] > mayor Entonces
            mayor = vec[i]
        FinSi

```

```
FinPara
FinFuncion

Funcion s = sumaVector(vec, n)
    Definir i, s Como Entero
    s = 0
    Para i = 1 Hasta n Con Paso 1 Hacer
        s = s + vec[i]
    FinPara
FinFuncion

SubAlgoritmo llenarVector(vec, n)
    Para i = 1 Hasta n Hacer
        vec[i] = Aleatorio(1, 50)
    FinPara
FinSubAlgoritmo

SubAlgoritmo mostrarVector(vec, n)
    Definir i Como Entero
    Para i = 1 Hasta n Hacer
        Imprimir vec[i], "    " Sin Saltar
    FinPara
FinSubAlgoritmo

Algoritmo Vectores
    Dimension V[30]
    Definir V, t, i Como Entero
    Imprimir "Total elementos vector: " Sin Saltar
    Leer t
    llenarVector(V, t)
    mostrarVector(V, t)
    Imprimir ""
    Imprimir "Suma vector: ", sumaVector(V, t)
    Imprimir "Mayor dato vector: ", mayorDatoVector(V, t)
FinAlgoritmo
```

## Operaciones sobre un vector

Ya hemos visto algunas operaciones que pueden aplicarse sobre un vector, tales como el acceso a los elementos de este, ya sea para leerlos o mostrarlos. Las operaciones típicas sobre arreglos unidimensionales son:

1. Llenar el vector
2. Recorrer el vector
3. Buscar un dato en el vector

4. Insertar un dato en el vector (en una posición dada, o antes o después de una referencia)
5. Eliminar un dato (de una posición dada)
6. Ordenar el vector

Las dos primeras operaciones se realizaron en el ejemplo 1.2; veamos las demás.

### Buscar un dato

La búsqueda es una de las operaciones más importantes y comunes en vectores y otras estructuras de datos, por lo que se han desarrollado distintas técnicas, unas más complejas que otras, aprovechando la capacidad de las máquinas para ejecutar instrucciones a altas velocidades, lo cual es aprovechado para buscar información en listas de datos mediante la creación de algoritmos para ello. La **búsqueda secuencial** es la más sencilla de todas y de las más utilizadas; los tipos de búsqueda sobre vectores y otras estructuras de datos, son:

- Búsqueda secuencial
- Búsqueda binaria
- Búsqueda por transformación de claves
- Árboles Binarios de Búsqueda (ABB)

En cursos posteriores se analizan los algoritmos que permiten realizar dichas búsquedas midiendo y comparando su eficiencia; veamos la búsqueda secuencial y binaria.

#### Ejemplo 1.11

**Búsqueda secuencial** de un dato en un vector.

Este tipo de búsqueda consiste en tomar un dato y compararlo elemento por elemento del vector hasta encontrarlo o hasta terminar de recorrer el vector. El tipo de dato que devuelve el método (función) puede ser un valor *lógico* (*booleano*) para indicar que el dato está o no en el arreglo; también puede devolver un valor *entero* con la posición donde el dato se encuentra, en cuyo caso se inicializa en un valor por fuera de los posibles valores que tome el índice del vector para indicar que el dato no se encuentra.

Programa Java:

```
public int secuencialSearchVector(int datum)
{
    int i, pos;
    i = 0;
    pos = -1;
    while (i < this.n && pos == -1) {
        if (this.vec[i] == datum) {
            pos = i;
        } else {
            i++;
        }
    }
}
```

```
    return pos;  
}
```

### Ejemplo 1.12

**Búsqueda binaria** de un dato en un vector.

Para implementar este tipo de búsqueda, **el vector debe estar ordenado**. Consiste definir el **límite inferior** y un **límite superior**, y a partir de estos datos, se toma el elemento central calculando la posición “**promedio**”; si el dato se encuentra en dicha posición, la búsqueda finaliza, de lo contrario, se evalúa si el dato es mayor o menor que el elemento en la posición y se redefinen los límites. El proceso finaliza si el dato es encontrado o si no hay un intervalo de búsqueda. Esta búsqueda reduce significativamente el número de comparaciones, lo cual puede verse en vectores grandes

Programa Java:

```
public int binarySearchVector(int datum)  
{  
    int lowerLimit, upperLimit, pos, centralPos;  
    lowerLimit = 0;  
    upperLimit = this.n;  
    pos = -1;  
    while (lowerLimit <= upperLimit && pos == -1) {  
        centralPos = (upperLimit + lowerLimit) / 2;  
        if (this.vec[centralPos] == datum) {  
            pos = centralPos;  
        } else {  
            if (this.vec[centralPos] > datum) {  
                upperLimit = centralPos - 1;  
            } else {  
                lowerLimit = centralPos + 1;  
            }  
        }  
    }  
    return pos;  
}
```

Eliminar un dato

Esta operación consiste en “pararse” en la posición del elemento a eliminar y desplazar los siguientes  $n - \text{posición} - 1$  elementos hacia la “izquierda” una posición. Una vez se muevan los elementos (se creen las copias), se reduce el tamaño del vector en uno. Debe comprobarse que hayan elementos para poder eliminar.

### Ejemplo 1.13

**Eliminar** un dato de un vector.

Programa Java:

```
public void deleteVector(int pos)
{
    int i;
    for (i = pos; i < n - 1; i++) {
        this.vec[i] = this.vec[i + 1];
    }
    this.n--;
}
```

## Insertar un dato

Esta operación consiste en “pararse” en la última posición e ir moviendo (copiando) los anteriores  $n - \text{posición} - 1$  elementos hacia la “derecha” una posición para “abrir el espacio”. Una vez se muevan los elementos (se creen las copias), se guarda el nuevo dato en la posición y se aumenta el tamaño del vector en uno. Antes de realizar la inserción, debe verificarse que haya espacio disponible en el vector para evitar un error.

### Ejemplo 1.14

**Insertar** un dato en un vector.

Programa Java:

```
public void insertVector(int pos, int datum)
{
    int i;
    for (i = this.n; i > pos; i--) {
        this.vec[i] = this.vec[i - 1];
    }
    this.vec[pos] = datum;
    n++;
}
```

## Ordenar el vector

La ordenación de datos es una operación importante y requerida en distintas situaciones al operar con vectores y otras estructuras de datos, tanto a nivel interno como externo; consiste en clasificar los elementos en un orden determinado, facilitando así las tareas de búsqueda. Al igual que con la búsqueda de datos, con la ordenación se han desarrollado distintas técnicas, unas más complejas que otras que permiten la clasificación de la información. La ordenación por el **método de intercambio directo o burbuja** es la más sencilla de todas, de las más utilizadas, pero también la más ineficiente; los tipos de ordenación sobre vectores y otras estructuras de datos, son:

- Ordenación directa (burbuja)
- Ordenación por inserción directa (baraja)
- Ordenación por selección directa
- Ordenación por el método de Shell
- Ordenación por el método de fusión (merge sort)

- Ordenación por el método rápido (quicksort)
- Ordenación por el método del montículo (heapsort)

En cursos posteriores se analizan los algoritmos que permiten realizar distintas formas de ordenaciones midiendo y comparando su eficiencia.

### Ejemplo 1.15

Ordenación de un vector por el método de **intercambio directo o burbuja**.

Este método consiste en tomar cada elemento del vector y compararlo con los siguientes  $n - i - 1$  elementos; si alguno es mayor (o menor), se realiza un *intercambio directo* y se continúa comparando. Este método es lento comparado con otros, pero garantiza que deja ordenado cada elemento antes de pasar al siguiente.

Programa Java:

```
public void sortBubbleVector()
{
    int i, j, aux;

    for (i = 0; i < this.n - 1; i++) {
        for (j = i + 1; j < this.n; j++) {
            if (this.vec[i] > this.vec[j]) {
                aux = this.vec[i];
                this.vec[i] = this.vec[j];
                this.vec[j] = aux;
            }
        }
    }
}
```

## Arreglos bidimensionales: Matrices o tablas

Una matriz es una colección de elementos dispuestos en filas (horizontales) y columnas (verticales), cada una etiquetada con un número entero que indica su número, lo cual significa que para hacer referencia a un elemento, debemos especificar dos índices. Tanto los vectores como las matrices son materia de estudio muy utilizados en matemáticas en distintas líneas como el Álgebra Lineal y en otras áreas de las ciencias naturales y aplicadas.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Figura 1.2. Representación gráfica de un arreglo bidimensional: matriz  $A_{m \times n}$ .<sup>5</sup>

### Representación en memoria

La representación en memoria de una matriz es a manera de tabla. La siguiente figura muestra un ejemplo de representación de matriz a nivel informático.

mat <sub>5x3</sub>		
2	1	0
99	10	-1
6	-8	11
22	16	2
65	-3	47
	1	

Figura 1.3. Representación gráfica de un arreglo bidimensional en memoria

### Nota

Observe que un vector es una matriz de  $1 \times n$  ó  $n \times 1$  elementos, es por ello que también se habla de **vector fila** o **vector columna** para referirse a estos tipos especiales de matrices.

### Declaración de matrices

Las matrices son arreglos de dos dimensiones compuestos por filas y columnas, por tanto usamos dos valores para especificar el total de filas y de columnas, respectivamente, separados por comas.

### Sintaxis

#### Pseudocódigo

```
tipo_de_dato: nombre_matriz[total_filas, total_columnas]
```

#### Java

```
tipo_de_dato nom_matriz[][] = new tipo_de_dato[tot_filas][tot_columnas]
```

### Nota

Observe que el tamaño de una matriz es igual al número de filas multiplicado por el número de columnas.

### Acceso a los elementos de una matriz

Al tener dos dimensiones, utilizamos dos índices (posiciones) para acceder a un elemento de una matriz. El primer índice hace referencia a la fila, y el segundo a la columna.

---

<sup>5</sup> Imagen tomada de: [Matriz \(matemática\) - Wikipedia, la enciclopedia libre](#)

## Sintaxis

Pseudocódigo

```
nombre_matriz[número_fila, número_columna]
```

Java

```
nombre_matriz[número_fila][número_columna]
```

### Nota

En Java, tanto la primera posición para las filas como para las columnas, es la cero (0).

### Ejemplo 1.16

Crear una matriz de orden 3x3. Agregar algunos elementos y realizar algunas operaciones.

Pseudocódigo:

```
Inicio
    Enteros: matriz[3, 3]
    Leer matriz[1, 1]
    matriz[1, 3] = 4
    matriz[2, 3] = matriz[1, 1] + matriz[1, 3]
    Imprimir matriz[1, 1], matriz[1, 3], matriz[2, 3]
```

### Ejemplo 1.17

Crear una matriz con tamaño máximo de 30 filas y 30 columnas con datos aleatorios entre 1 y 50 y mostrarla en forma de tabla.

Se presenta la solución en pseudocódigo y PSeInt.

Pseudocódigo:

```
Procedimiento mostrarMatriz(mat, m, n)
    Entero: i, j
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Imprimir mat[i, j]
        FinPara
    FinPara
FinProcedimiento

Inicio
Enteros: M[30, 30], i, j
Para i = 1 Hasta 3 Hacer
    Para j = 1 Hasta 3 Hacer
        M[i, j] = Aleatorio(1, 50)
    FinPara
FinPara
```

```
mostrarMatriz(M, 3, 3)
Fin
```

Pseudo Programa PSeInt:

```
SubAlgoritmo mostrarMatriz(mat, m, n)
    Definir i, j Como Entero
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Imprimir ConvertirATexto(mat[i, j]) + "    " Sin Saltar
        FinPara
        Imprimir ""
    FinPara
FinSubAlgoritmo

Algoritmo Matrices
    Dimension M[30, 30]
    Definir M Como Entero
    Definir i, j Como Entero
    Para i = 1 Hasta 3 Hacer
        Para j = 1 Hasta 3 Hacer
            M[i, j] = Aleatorio(1, 50)
        FinPara
    FinPara

    mostrarMatriz(M, 3, 3)
FinAlgoritmo
```

## Arreglos multidimensionales

Son arreglos de más de dos dimensiones. Hasta tres dimensiones, pueden ser representados gráficamente (un cubo o caja); más allá de ahí, es imposible, pero se pueden implementar tanto matemática como algorítmicamente. Sin embargo, no todos los lenguajes implementan arreglos multidimensionales; dentro de los que permiten crear este tipo de arreglos se encuentran C/C++, PHP y Java.

### Representación en memoria

Arreglos de más de tres dimensiones no pueden representarse gráficamente; un arreglo tridimensional se representa como un cubo o caja, como se muestra en las siguientes figuras.

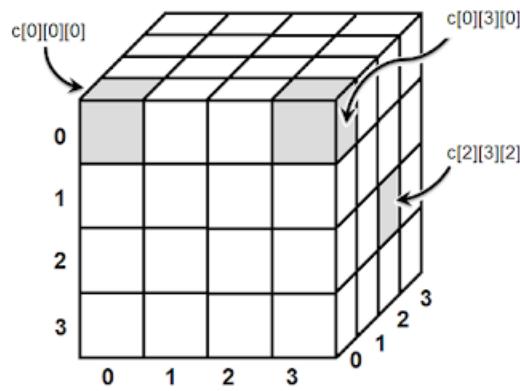


Figura 1.4. Representación gráfica de un arreglo tridimensional en forma de cubo. Aquí la primera posición para cada dimensión es la cero (0)<sup>6</sup>

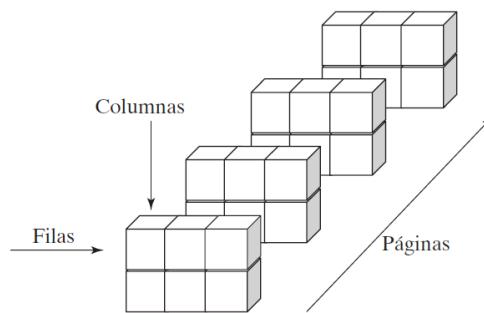


Figura 1.5. Representación gráfica de un arreglo tridimensional<sup>7</sup>

## Declaración de arreglos multidimensionales

Para declarar arreglos multidimensionales, simplemente separamos por comas tantas dimensiones como necesitemos, teniendo presente las limitaciones de los lenguajes de programación.

### Sintaxis

Pseudocódigo

```
tipo_de_dato: nombre_arreglo[dimension1, dimension2, ..., dimensionN]
```

Java

```
tipo_de_dato nom_arreglo[][]...[] = new tipo_de_dato[dim1][dim2]...[dimN]
```

## Acceso a los elementos de un arreglo multidimensional

Especificamos un índice por cada dimensión del arreglo separados por comas.

### Sintaxis

---

<sup>6</sup> Imagen tomada de [Arreglos de n dimensiones](#).

<sup>7</sup> Imagen tomada de [Tipos de Arreglos en Matlab - \[octubre, 2023\]](#)

Pseudocódigo

```
nombre_arreglo[indice1, indice2, ..., indiceN]
```

Java

```
nombre_arreglo[indice1][indice2]...[indiceN]
```

**Nota**

A medida que se aumentan las dimensiones de un arreglo, aumenta tanto la complejidad de los algoritmos, como el consumo de recursos de la máquina, por lo que se debe tener precaución al usar arreglos de varias dimensiones. Incluso un problema resuelto mediante vectores y que podría ser solucionado sin ellos, consumirá más recursos de máquina que otro que no los use.

A nivel matemático y algorítmico se puede teorizar a un nivel arbitrario finito de dimensiones, sin embargo, los lenguajes de programación imponen restricciones en cuanto a la cantidad que pueden usarse en éstos, por lo que es necesario revisar la documentación de cada uno antes de intentar realizar algún desarrollo.

Veamos una aplicación del uso de arreglos tridimensionales (tres dimensiones).

**Ejemplo 1.18**

Una empresa lleva registro de la producción de 10 artículos que elaboran 5 empleados en la semana. Se requiere crear un arreglo que almacene la cantidad de unidades que cada empleado produce al día para sacar diversos reportes.

Necesitamos crear un arreglo de tres dimensiones de orden  $5 \times 10 \times 7$ . La primera dimensión (*filas*) representa los empleados, y el índice representa su código; la segunda dimensión (*columnas*) representa los productos que igualmente se identifican con un código asociado al índice; y la tercera dimensión (*profundidad, páginas*) representa los días de la semana. Así, si el arreglo se llama *produccion*:

```
produccion[2, 7, 3] = 250
```

Significa que el empleado de código 2 produjo el martes 250 unidades del artículo con código 7.

Tres vectores de tamaños iguales a cada dimensión, almacenan el nombre de los empleados, los artículos y los días respectivamente.

Crear subprogramas que resuelvan además:

- Dado un empleado y un artículo, sume las unidades producidas
- Encuentre el día en que el empleado obtiene mejor rendimiento de un artículo dado
- Promedio de producción de un día determinado

Pseudocódigo:

```
Procedimiento mostrarArreglo(arreglo, m, n, p)
```

```
    Entero: i, j, k
```

```
Para i = 1 Hasta m Hacer
    Para j = 1 Hasta n Hacer
        Para k = 1 Hasta p Hacer
            Imprimir arreglo[i, j, k]
        FinPara
    FinPara
FinPara
FinProcedimiento

Procedimiento llenarArreglo(arreglo, m, n, p)
    Entero: i, j, k
    Para i = 1 Hasta m Hacer
        Para j = 1 Hasta n Hacer
            Para k = 1 Hasta p Hacer
                arreglo[i, j, k] = Aleatorio(1, 100)
                // Otra opción: Leer arreglo[i, j, k]
            FinPara
        FinPara
    FinPara
FinProcedimiento

Funcion sumarProduccionDiasEmpleado(prod, codEmp, codArt, dias): Entero
    Entero suma, i
    suma = 0
    Para i = 1, dias, 1
        suma = suma + prod[codEmp, codArt, i]
    FinPara
    Retornar suma
FinFuncion

Funcion mejorDiaEmpleado(prod, codEmp, codArt, dias): Entero
    Entero mayor, i
    mayor = prod[codEmp, codArt, 1] //Supuesto: domingo mejor producción
    Para i = 2, dias, 1
        Si prod[codEmp, codArt, i] > mayor Entonces
            mayor = prod[codEmp, codArt, i]
        FinSi
    FinPara
    Retornar mayor
FinFuncion

Funcion promedioDia(prod, m, n, dia): Real
    Entero: i, j, suma
    Real: promedio
    suma = 0
    Para i = 1, m, 1
        Para j = 1, n, 1
```

```

        suma = suma + prod[i, j, dia]
    FinPara
    FinPara
    promedio = suma / (m * n)
    Retornar promedio
FinFuncion

Inicio
Entero: produccion[5, 10, 7], s
llenarArreglo(produccion, 5, 10, 7])
mostrarArreglo(produccion, 5, 10, 7])
// Empleado 3, artículo 2, para sumar todos los días de la semana
s = sumarProduccionDiasEmpleado(produccion, 3, 2, 7)
Imprimir "Producción empleado 3 artículo 2: ", s, " unidades"
Imprimir "Promedio de unidades por día: ", promedioDia(prod, 5, 10, 4)
Fin

```

## Registros

Un **registro** es un tipo estructurado de datos (compuesto), esto significa que se compone de datos primitivos y posiblemente otros tipos estructurados, los cuales se conocen como **campos** del registro y que deben tener un identificador (nombre) único. Algunos lenguajes estructurados (procedimentales) e híbridos soportan la creación de registros (como en C/C++ y en PHP); los lenguajes orientados a objetos no implementan un tipo especial para crear registros, pero esto se soluciona fácilmente creando objetos sin métodos, solo especificando propiedades (atributos) que representan los campos del registro.

Al poder definir registros en un lenguaje de programación, tenemos la posibilidad de crear nuevos tipos de datos, esta vez definidos por el usuario a partir de los datos primitivos y posiblemente otros datos estructurados de acuerdo a las necesidades que indique un problema.

La siguiente figura muestra la estructura de un registro de clientes con algunos campos definidos para éste.

Registro Clientes						
Nombre del campo	Identificación	Nombre	Ciudad	Dirección		Teléfono
Tipo de dato	(carácter)	(carácter)	(carácter)	Barrio	Tipo vía	Número
				(carácter)	(carácter)	(entero)



Figura 1.6. Representación gráfica de un registro de clientes con algunos campos

## Crear un registro

Algorítmicamente o en pseudocódigo, podemos definir un registro así:

### Sintaxis

```
Registro nombre_registro
    tipo_dato campo1 [, 
    tipo_dato campo2 [, 
    ...]]
FinRegistro
```

Análogamente, en lógica orientada a objetos realizamos lo siguiente:

### Sintaxis

```
Clase nombre_clase
    publico|privado|protegido tipo_dato atributo1 [, 
    publico|privado|protegido tipo_dato atributo2 [, 
    ...]]
FinClase
```

## Crear variables de tipo registro

Para definir una variable de tipo registro realizamos lo siguiente.

### Sintaxis (crear una variable de tipo registro)

```
Registro nombre_registro variable
```

Análogamente, en lógica orientada a objetos creamos un objeto, es decir, instanciamos una clase:

### Sintaxis (instanciar una clase)

```
NombreClase nombre_objeto = Nuevo NombreClase
```

## Acceso a los campos de un registro

Acceder a una variable de tipo registro consiste en acceder a sus campos. Dado que el lenguaje algorítmico permite flexibilidad, o más bien, definir un propio pseudolenguaje

independiente de los lenguajes existentes, pero “universal”, podemos establecer en este texto el operador **punto** (.) para acceder a los campos.

### **Sintaxis (variable de tipo registro)**

***variable.campo***

Análogamente, en lógica orientada a objetos accedemos a las propiedades de un objeto de esta forma:

### **Sintaxis (objeto representando un registro)**

***objeto.campo***

#### **Ejemplo 1.19**

Crear un registro de estudiantes con los campos identificación, nombre, edad y si trabaja o no. Leer los datos de un estudiante y mostrar su información. Presentar la solución en pseudocódigo, C/C++ y Java.

Pseudocódigo:

```
Inicio
  Registro Estudiante
    Caracter: identificacion,
    Caracter: nombre,
    Enteros: edad,
    logico: trabaja
  FinRegistro

  Estudiante: est1
  Leer est1.identificacion
  Leer est1.nombre
  Leer est1.edad
  Leer est1.trabaja
  Imprimir est1.identificacion
  Imprimir est1.nombre
  Imprimir est1.edad
  Imprimir est1.trabaja
Fin
```

Programa C++:

```
#include <iostream>

struct Estudiante
{
    char identificacion[20];
    char nombre[50];
    int edad;
```

```
    bool trabaja;
};

int main(int argc, char** argv)
{
    struct Estudiante est1;
    printf("Identificación: ");
    cin.getline(est1.identificacion);
    printf("Nombre: ");
    cin.getline(est1.nombre);
    printf("Edad: ");
    cin >> est1.edad;
    printf("Trabaja [1->Sí | 2->No: ");
    cin >> est1.trabaja;
    printf(est1.identificacion);
    printf(est1.nombre);
    printf(est1.edad);
    printf(est1.trabaja);
}
```

La solución en Java requiere de un poco más de trabajo. Crearemos un paquete con dos clases, una contendrá la estructura del registro y la otra la interfaz de la aplicación

La estructura de carpetas es la siguiente para un SO Windows:

carpeta\_principal\Registros.java

carpeta\_principal\com\packages\Estudiante.java

Programa Java: Estudiante.java

```
public class Estudiante
{
    String identificacion;
    String nombre;
    int edad;
    boolean trabaja;
}
```

Programa Java: Registros.java

```
import java.util.Scanner;
import com.packages.Estudante;

public class Registros
{
    public static Scanner input = new Scanner(System.in);

    public static void main(String[] args)
```

```

{
    Estudiante est1 = new Estudiante();
    System.out.println("Identificación: ");
    est1.identificacion = input.nextInt();
    System.out.printls("Nombre: ");
    est1.nombre = input.next();
    System.out.println("Edad: ");
    est1.edad = input.nextInt();
    System.out.println("Trabaja [true | false: ");
    est1.trabaja = input.nextBoolean();
    System.out.println(est1.identificacion);
    System.out.println(est1.nombre);
    System.out.println(est1.edad);
    System.out.println(est1.trabaja);
}
}

```

## Diferencias con objetos

Observe la similitud entre un registro y un objeto cuando se define su estructura básica, es decir, un objeto sin métodos; sin embargo, son elementos muy diferentes; un objeto generalmente contiene métodos y hace parte de un paradigma que va más allá de solo especificar atributos (campos) a una entidad. Los registros son más simples y anteriores a la POO, en la cual se trata el registro como un conjunto de atributos junto a un conjunto de métodos que permiten la manipulación de dichos atributos (propiedades).

Realmente, y de acuerdo a lo que en informática entendemos por registro, el conjunto de atributos o propiedades de un objeto ¡es un **registro!** y esta puede ser una de las razones por las que en los lenguajes orientados a objetos no existe una definición para el tipo struct como en C/C++ o PHP para evitar ser redundantes en cuestiones que son inherentes a los objetos. C es un lenguaje estructurado que solo soporta la creación de registros; C++ y PHP son lenguajes híbridos y soportan tanto la creación de registros como de objetos; Java y Python al ser lenguajes orientados a objetos, todo lo consideran como un objeto, por lo que un registro es simplemente otro objeto.

## Diferencias con arreglos

Hay básicamente dos diferencias entre registros y arreglos:

- La información almacenada en arreglos es homogénea, esto es, del mismo tipo; mientras que en los registros se guarda información heterogénea, es decir, de distintos tipos de datos, tanto primitivos como estructurados.
- Para acceder a un elemento de un arreglo, se utiliza uno o varios índices representados por valores enteros, mientras que para acceder a los elementos de un registro se hace mención al nombre único o identificador del campo usando el operador punto (.).

## Arreglos de registros, arreglos de objetos y arreglos paralelos

Los lenguajes permiten un nivel de abstracción para combinar arreglos con registros y objetos, lo que da la posibilidad de crear “arreglos heterogéneos” en el sentido que permitirán manipular información de distinto tipo, pero sin salir de la regla de ser homogéneos, ya que en realidad son definidos y están almacenando información de un mismo tipo estructurado. Un arreglo de registros, así como un arreglo de objetos no es más que un arreglo que contendrá en cada posición un registro u objeto

En cuanto a los objetos, es de tener cuidado en cómo se realiza la implementación, ya que cada objeto es una instancia única de la clase, por lo que debe instanciarse un objeto por cada posición del arreglo donde deseemos almacenarlo.

Cuando un lenguaje no implementa registros ni el paradigma de la POO, es posible usar “arreglos paralelos” para solucionar problemas que involucren tratar varios datos de una misma entidad. Como en el ejemplo anterior, en lugar de usar registros u objetos, se pueden crear los arreglos `identificacion[]`, `nombre[]`, `edad[]`, `trabaja[]`, todos del mismo tamaño y en donde una posición (índice) corresponde a la misma entidad, en este caso a un estudiante determinado.

### Ejemplo 1.20

Teniendo presente el registro y la clase de Estudiantes del ejemplo anterior, crear un arreglo de registros en pseudocódigo y un arreglo de objetos en Java de tipo Estudiantes leyendo varias entradas y mostrando dicha información.

Pseudocódigo:

```
Inicio
Registro Estudiante
    caracter identificacion,
    caracter nombre,
    entero edad,
    logico trabaja
FinRegistro

Caracter: resp
Enteros: te = 0
Estudiante: est[]
Repetir
    te = te + 1
    Leer est[te].identificacion
    Leer est[te].nombre
    Leer est[te].edad
    Leer est[te].trabaja
    Leer resp
Hasta Que resp == minusculas("no")
```

```
Para i = 1, te, 1
    Imprimir est[i].identificacion
    Imprimir est[i].nombre
    Imprimir est[i].edad
    Imprimir est[i].trabaja
FinPara
Fin
```

Así como vimos anteriormente, la solución en Java requiere un poco más de trabajo. Crearemos un paquete con dos clases, una contendrá la estructura del registro y la otra métodos para manipularlo.

La estructura de carpetas y archivos es la siguiente para un SO Windows:

main\_folder\MainMenu.java

main\_folder\com\packages\records\Student.java  
main\_folder\com\packages\records\Records.java

Programa Java: Student.java

```
package com.packages.records;

public class Student
{
    public String id;
    public String name;
    public int age;
    public boolean work;
}
```

Programa Java: Records.java

```
package com.packages.records;

import java.util.Scanner;

public class Records
{
    private static Scanner input = new Scanner(System.in);
    private Student est[] = new Student[30];
    private int ts;

    public Records()
    {
        ts = 0;
    }
```

```
public int getTs()
{
    return ts;
}

public void setTs(int ts)
{
    this.ts = ts;
}

public void createRecords()
{
    String resp = "";
    do {
        est[ts] = new Student();
        System.out.println("Identificación: ");
        est[ts].id = input.nextInt();
        System.out.println("Nombre: ");
        est[ts].name = input.next();
        System.out.println("Edad: ");
        est[ts].age = input.nextInt();
        System.out.println("Trabaja [true | false]: ");
        est[ts].work = input.nextBoolean();
        ts++;
        System.out.println("Continuar? [s/?]");
        resp = input.next();
    } while (resp.toLowerCase().equals("s"));
}

public void showRecords()
{
    System.out.println("-----");
    for (int i = 0; i < ts; i++) {
        System.out.println("Id: " + est[i].id);
        System.out.println("Nombre: " + est[i].name);
        System.out.println("Edad: " + est[i].age);
        System.out.println("Trabaja: " + est[i].work);
        System.out.println("-----");
    }
}
```

## Conjuntos

Un conjunto es un elemento similar a un vector, pero que tiene la misma propiedad de los conjuntos matemáticos, y es que cada elemento que lo compone es único. Los conjuntos por su parte, admiten información heterogénea.

Muy pocos lenguajes tienen una estructura de datos explícita para su tratamiento, como el lenguaje Python, por ejemplo. En Java si se desea implementar una aplicación relacionada con conjuntos, se pueden usar vectores o listas ligadas (aunque es más cómodo trabajar estos elementos con arreglos) e implementar las reglas de conjuntos respectivas.

A nivel de lógica de programación, podemos definir un conjunto así:

### Sintaxis

1.

```
variableConjunto = {elementos}
```

2.

```
variableConjunto = Conjunto(iterable)
```

En Python se hace de esta forma:

### Sintaxis

1.

```
variableConjunto = {elementos}
```

2.

```
variableConjunto = set(iterable)
```

Donde:

*elementos*: es la secuencia o conjunto de elementos separados por comas. Si no se indica, el conjunto se crea vacío

*iterable*: objeto iterable reconocible por Python (o en lógica), tal como listas (vectores), tuplas o cadenas. Si no se especifica, el conjunto se inicializará vacío

Con los conjuntos de Python es posible realizar algunas de las operaciones matemáticas entre ellos.

## Operaciones con conjuntos

Las operaciones fundamentales entre conjuntos son: Unión, Intersección y Diferencia; otras operaciones se pueden expresar en términos de estas operaciones y otras reglas. Estas operaciones crean nuevos conjuntos.

Operación	Símbolo operador	Prioridad	Uso
-----------	------------------	-----------	-----

Unión	$+$	Union	Baja	$A + B$
Intersección	$*$	& Interseccion	Alta	$\text{Interseccion}(A, B)$
Diferencia	-	Diferencia	Baja	$A - B$

## Implementación de conjuntos en Python

El tratamiento de un conjunto en Python es similar al tratamiento de listas (vectores), aunque difiere en algunos métodos y también dispone de otros para las operaciones entre conjuntos.

### Creación de conjuntos

#### Ejemplo 1.21

Creación de los conjuntos a y b:

```
>>> a = {1, 2, 3}          # Tener
>>> a                      # Recibir
{1, 2, 3}

>>> b = set([0, 5, 1])
>>> b
{0, 1, 5}
```

#### Ejemplo 1.21

A partir de los conjuntos a y b realizar las operaciones de Unión, intersección y diferencia de los conjuntos utilizando los métodos y operadores disponibles en Python:

```
>>> c = a.union(b)
>>> c
{0, 1, 2, 3, 5}
>>> c = a.intersection(b)
>>> c
{1}
```

```
>>> a
{0, 1, 3}
>>> b
{0, 1, 5}
>>> c = a | b
>>> c
{0, 1, 3, 5}
>>> c = a & b
>>> c
{0, 1}
>>> c = a - b
>>> c
{3}
```

Sean:

a, b, c: conjuntos

dato: una variable

posicion: variable que guarda un índice de la tupla

Se pueden aplicar los siguientes métodos a un conjunto en Python:

- Obtener el total de elementos con la función len: len(a).
- Agregar un elemento: a.add(dato).
- Buscar con el operador in: sw = dato in a # True: si se encuentra.
- Eliminar un elemento: a.discard(dato).

### **Ejemplo 1.22**

A partir de los conjuntos a y b dados anteriormente, realizar lo siguiente:

- Encontrar la cardinalidad del conjunto
- Adicionar un nuevo elemento al conjunto
- Eliminar un elemento del conjunto

```
>>> len(a)
3
>>> a.add(2)
>>> a
{1, 2, 3}
>>> a.add(0)
>>> a
{0, 1, 2, 3}
>>> a.discard(2)
>>> a
{0, 1, 3}
```

### **Nota**

En los ejercicios se propone desarrollar una aplicación que permita trabajar con conjuntos en Java.

## Preguntas

## Ejercicios

1. Ingrese un texto por teclado. Cree un subprograma para determinar cuántas vocales se encuentran en éste
2. Ingrese un texto y un carácter. Cree un subprograma para encontrar cuántas veces está dicho carácter en el texto
3. Ingrese un texto. Cree un subprograma que indique cuántas palabras conforman el texto

4. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena separada cada carácter por un espacio en blanco. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “h o y e s j u e v e s”
5. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena sin espacios en blanco. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “hoyesjueves”
6. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena con los caracteres intercambiados entre minúscula y mayúscula. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “HoY Es jUeVeS”
7. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena en letra capital. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “Hoy Es Jueves”
8. Dada una cadena de caracteres, use un subalgoritmo para obtener la misma cadena en orden inverso. Ejemplo: si la cadena es “hoy es jueves”, al final debe entregar “seveuj se yoh” (esta podría considerarse una forma básica de cifrar un mensaje)
9. Ingrese el nombre de una persona. Utilice una función para validar que no ingresen números
10. Dadas dos cadenas de caracteres, use un subalgoritmo que realice lo siguiente: si la primera cadena tiene un número impar de caracteres, concatene la segunda cadena al inicio de la primera; en caso contrario, concaténela al final.
11. Dada una cantidad numérica, use funciones para convertirla a letras
12. Dado un número entero positivo, use funciones para convertirlo a número romano
13. Dado un número, utilice subprogramas para determinar si es un número capicúa sin convertirlo a cadena (los números capicúas son aquellos que se leen igual de ambos sentidos, por ejemplo: 12321).
14. Dada una frase, cree subprogramas para determinar si ésta es un palíndromo o no (los palíndromos son aquellas frases que se leen igual de ambos sentidos, por ejemplo: “amad a la dama”)
15. Leer N números y guardar en una vector los números positivos. Crear subprogramas para mostrar el arreglo y mostrar el mayor número almacenado.
16. Teniendo presente el ejemplo 1.2, agregue métodos a la clase Vector desarrollada en la aplicación de ejemplo:

- a. Sumar los datos del vector (sumatoria)  $s = \sum_{i=1}^n V_i$
- b. Multiplicar los datos del vector (productoria)  $p = \prod_{i=1}^n V_i$
- c. Hallar el promedio  $\bar{x} = \frac{\sum_{i=1}^n V_i}{n}$
- d. Encontrar el mayor dato  $\max(V_i)$
- e. Encontrar el menor dato  $\min(V_i)$
- f. Encontrar la moda si existe (la moda es el dato que más se repite)
- g. Encontrar la multimoda
- h. La mediana (dato que con la lista ordenada, se encuentra en el centro de la muestra).  $\hat{x} = x_{(n+1)/2} \text{ si } n \text{ es impar}; \hat{x} = \frac{x_{n/2} + x_{n/2+1}}{2} \text{ si } n \text{ es par}$

$$i. \text{ Varianza muestral } \sigma^2 = \sum_{i=1}^n \frac{(x_i - \bar{x})^2}{n-1}$$

$$j. \text{ Desviación estándar } \sigma = \sqrt{\sigma^2} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}$$

17. La universidad está procesando la información por grupos de sus estudiantes en arreglos, para lo cual requiere un programa para llevar registro de su documento de identidad, el nombre y la nota. La solución debe implementar programación modular para los siguientes requerimientos utilizando un menú de opciones:

- a. Pedir los datos de un estudiante y agregarlos a las listas. Debe validar que no se ingrese el mismo documento de identidad más de una vez y que la nota esté entre 0 y 5
- b. Listar todos los estudiantes en forma tabulada y añadir una columna adicional que muestre si ganó o perdió la asignatura
- c. Mostrar los datos de un estudiante seleccionado, para lo cual debe hacer una búsqueda por documento de identidad luego de solicitar éste
- d. Mostrar el promedio de notas del grupo
- e. Mostrar la mayor nota del grupo y a quien pertenece ésta
- f. Mostrar la menor nota del grupo y a quien pertenece ésta
- g. Permitir la eliminación de estudiantes
- h. Permitir actualizar la nota de un estudiante
- i. Informar si hay un estudiante de nombre Pedro Zapata o Ana Zapata

18. La universidad está procesando la información de sus estudiantes en arreglos, para lo cual requiere un programa para llevar registro de su documento de identidad, el nombre y la nota de cinco asignaturas. La solución debe implementar programación modular para los siguientes requerimientos utilizando un menú de opciones:

- a. Pedir los datos de un estudiante y agregarlos a las listas. Debe validar que no se ingrese el mismo documento de identidad más de una vez y que las notas estén entre 0 y 5
- b. Listar todos los estudiantes en forma tabulada junto con su promedio de notas
- c. Mostrar los datos de un estudiante seleccionado, para lo cual debe hacer una búsqueda por documento de identidad luego de solicitar éste
- d. Mostrar el promedio general de notas
- e. Mostrar la mayor nota de un estudiante cualquiera
- f. Mostrar la menor nota de un estudiante cualquiera
- g. Mostrar la mayor nota de todos los estudiantes y a quien pertenece ésta
- h. Mostrar la menor nota de todos los estudiantes y a quien pertenece ésta
- i. Permitir la eliminación de estudiantes
- j. Permitir actualizar la nota de un estudiante

19. Una tienda procesa la siguiente información de sus productos: nombre, código, precio y cantidad. La tienda maneja un dato general para controlar el stock mínimo de cada producto, esto es, la cantidad mínima de productos para que se lance una alerta y se deba pedir mercancía. La tienda requiere una aplicación modular que mediante un menú de opciones, permita realizar las siguientes operaciones sobre su información que se guarda en arreglos unidimensionales:

- a. Agregar productos de forma ordenada por código, validando que éstos no se repitan
  - b. Mostrar el producto más costoso
  - c. Mostrar el producto más barato
  - d. Listar el inventario de forma tabulada y añadir una columna adicional que muestre un IVA del 19% para todos los artículos
  - e. Permitir buscar un producto por código
  - f. Listar los productos cuya cantidad sea inferior al *stock* mínimo y alertar en caso de encontrarlos
  - g. Permitir actualizar cualquier dato del producto, excepto el código, validando además que la cantidad no sea negativa y el precio mayor a cero
  - h. Permitir eliminar productos
  - i. Informar si hay un producto que tenga una existencia de 300 o de 400 unidades
20. Crear una matriz  $M$  con  $m \times n$  valores numéricos aleatorios entre 1 y 50. Usando subprogramas, realizar las siguientes operaciones sobre ella:
- a. Mostrar el arreglo
  - b.  $\sum_{i=1}^n M_{i,j}$  (sumatoria)
  - c.  $\prod_{i=1}^n M_{i,j}$  (productoria)
  - d.  $\bar{M}$  (media o promedio)
  - e. Encontrar el mayor dato
  - f. Encontrar el mayor dato de una fila o columna dada
  - g. Encontrar el menor dato
  - h. Encontrar el menor dato de una fila o columna dada
  - i. Buscar un elemento
  - j. Eliminar una fila o columna
  - k. Insertar una fila o columna dada una fila y/o columna de referencia
  - l. Ordenar una fila o columna dada
  - m. Ordenar la matriz
21. Un almacén registra las ventas de la semana de  $n$  productos que manejan en una matriz. Cada fila de la matriz representa un producto que está codificado de acuerdo al número de ésta, así, la primera fila significa que el producto tiene código 1, la fila 2 es para el producto con código 2 y así sucesivamente; cada columna representa un día de la semana: la primera equivale al domingo, la segunda al lunes, etc. Usando programación modular y mediante un menú de opciones, se pide:
- a. Ingresar las ventas de totales de cada producto por día
  - b. Mostrar la matriz completa en forma de tabla
  - c. Buscar un producto y mostrar las ventas de la semana
  - d. Dado un día, mostrar las ventas de cada producto en éste
  - e. Mostrar el total de ventas por día
  - f. Mostrar el promedio de ventas por día
  - g. Mostrar el total de ventas por producto
  - h. Mostrar el promedio de ventas por producto
  - i. Mostrar el total de ventas de la semana
  - j. Mostrar el promedio de ventas de la semana

- k. Mostrar la mejor venta del día
  - l. Mostrar el mejor día en ventas de un producto
  - m. Mostrar la peor venta del día
  - n. Mostrar el peor día en ventas de un producto
  - o. Mostrar la mejor venta e indicar a qué producto corresponde y en qué día se realizó
  - p. Mostrar la peor venta e indicar a qué producto corresponde y en qué día se realizó
  - q. Permitir actualizar los valores de las ventas
22. Utilice el lenguaje Java para codificar el ejemplo 1.7 relacionado con arreglos multidimensionales
23. Aplicaciones con conjuntos. Sean A y B dos conjuntos. Cree dos vectores y realice las operaciones básicas sobre conjuntos (recuerde que un conjunto no contiene elementos repetidos)
- a. La cardinalidad de los conjuntos
  - b. Determine si A y B son iguales:  $A = B$
  - c. Unión:  $A \cup B$
  - d. Intersección:  $A \cap B$
  - e. Diferencia:  $A - B$
  - f. Diferencia simétrica:  $A \Delta B$
  - g. Complemento de un conjunto cualquiera X ( $X = A$  o  $X = B$ ):  $X'$
  - h. Conjunto potencia de un conjunto cualquiera X ( $X = A$  o  $X = B$ ):  $P(X)$
  - i. Producto cartesiano:  $A \times B$  y  $B \times A$
24. Dada una matriz cuadrada de orden n, imprima las áreas y/o elementos
- a. De la diagonal principal y secundaria
  - b. Que están por debajo de la diagonal principal
  - c. Que están por encima de la diagonal principal
  - d. Que están por debajo de la diagonal secundaria
  - e. Que están por encima de la diagonal secundaria
  - f. Que forman un triángulo isósceles por cualquier lado
  - g. Que forman una onda sinusoidal
  - h. Que siguen la forma de un caracol

## Capítulo 2. Listas Ligadas

Las **listas ligadas** son **estructuras de datos lineales dinámicas** conformadas por elementos llamados **nodos**. Un nodo es un *registro* formado básicamente con dos campos: uno para almacenar la **información** y otro para guardar una **referencia (liga)** a otro nodo.



Figura 2.1. Estructura básica de un nodo

Este registro puede crearse fácilmente como una clase en Java; en el lenguaje C++ puede crearse como registro o bien como una clase, ya que éste soporta la creación de la estructura de datos **struct** para la creación de registros, mientras que en la versión del lenguaje C sólo puede crearse usando la estructura de datos **struct**. El campo **Información** puede a su vez estar compuesto de otro registro, arreglos, etc., mientras el campo **Liga** es una **referencia (dirección)** a otro nodo; si no hay una referencia a ningún otro nodo, dicho campo almacenará una marca de **NULO** (vacío) o **NIL** (**null** en Java).

### Nota

La estructura del nodo que utilizaremos en los ejemplos que se presentan más adelante, tendrán una estructura muy similar a la mostrada en la figura 2.1 con los campos **info** de tipo **entero** y **link** de tipo **nodo**. Para acceder a un nodo vamos a requerir de una variable de tipo **apuntador** que lo pueda referenciar. Las variables tipo **puntero**, apuntan a la memoria y sus valores son **números hexadecimales** que especifican una **dirección** cualquiera de la memoria o un valor **nulo (null)** o vacío.

A diferencia de los arreglos, las listas ligadas son estructuras de datos **dinámicas**, ya que no se reserva un espacio determinado de memoria antes de iniciar el programa, sino que ésta se va solicitando al sistema de acuerdo a la necesidad del usuario y/o aplicación, así como de la disponibilidad que tenga de ésta en la máquina. El hardware actual supone que no hay problemas para la asignación dinámica de la memoria, sin embargo, es importante tener las precauciones correspondientes y no abusar de los recursos disponibles, buscando siempre el mejor uso posible de éstos.

Al igual que sucede con los vectores, las listas ligadas son estructuras de datos **lineales**, ya que después de un elemento encontramos a lo sumo otro elemento.

## Lista Simplemente Ligada (LSL)

Una LSL es el tipo de lista ligada más sencilla, en el sentido que ésta se recorre en un solo sentido y no contiene referencias circulares. La figura 2.2 muestra la representación de una LSL.

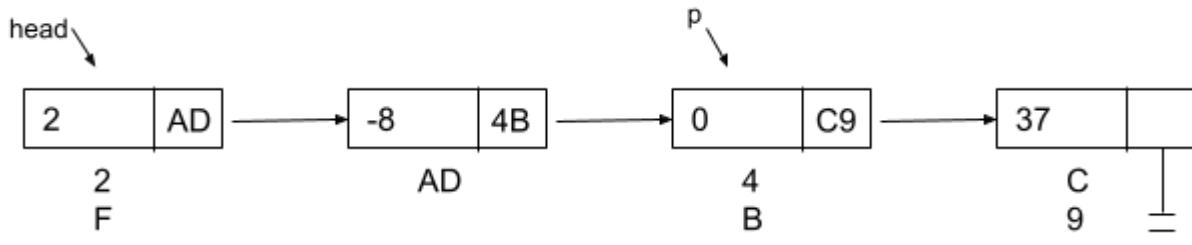


Figura 2.2. Representación de una LSL que contiene en su campo **info** números enteros y su primer registro es apuntado por la variable apuntador **head**. Un puntero **p** apunta a un nodo arbitrario de la lista. El campo **link** contiene la **dirección (referencia)** al siguiente nodo; el último nodo que no apunta a ningún otro, tiene una “marca” de **nulo** en este campo. Las direcciones de memoria están dadas en hexadecimal.

Una lista ligada, por regla general, debe contener un apuntador al primer nodo, con lo que garantizamos que la lista **no se pierda** y podamos acceder a su contenido. Dicho apuntador se suele conocer como **nodo (registro) cabeza**; en la gráfica aparece como **head**. En algunas aplicaciones, el registro cabeza se define como un registro independiente a los demás nodos de la lista, con el fin de establecer el inicio de ésta y la diferencia con los demás nodos; aquí no se considerarán nodos (registros) adicionales, ya que con el apuntador al primer nodo (cabeza) se puede implementar cualquier operación y no perder la información de la lista, basta con especificar correctamente las reglas en cada algoritmo de implementación sobre listas.

## Creación de una LSL

Una lista ligada puede creerse por el **inicio** o por el **final** de ésta. Antes de realizar esto, se debe crear primero el **registro** correspondiente para cada **nodo** de la lista. La representación de los nodos depende de las necesidades de la aplicación. Para los ejemplos que se describen a continuación para los subtemas de LSL y LSCL se utilizará la representación dada en la nota anterior y que se muestra en el ejemplo 2.1, que describe la creación de una clase para crear nodos.

### Ejemplo 2.1

Clase **Node (Nodo)** para crear nodos en una LSL; la clase básicamente es un registro que contiene los campos **info** de tipo **int (enteros)** y **link (liga)** de tipo **Node (Nodo)**.

Pseudocódigo:

```
Clase Nodo
    publico enteros info;
    publico Nodo liga;
FinClase
```

Programa Java:

```
public class Node
{
    public int info;
    public Node link;
}
```

## Asignar y eliminar memoria

Los lenguajes de programación disponen de mecanismos para traer y liberar la memoria. En pseudocódigo se acostumbra a usar una funciones como **TRAER(x)** y **LIBERAR(x)** para asignar memoria dinámicamente y eliminarla cuando ésta ya no se requiera.

### **Ejemplo 2.2**

Asignar memoria dinámicamente y eliminarla posteriormente y acceder a los campos del nodo.

Pseudocódigo:

```
...
punteros p, q // p, q: variables de tipo apuntador

// La función TRAER asigna memoria a las variables tipo puntero p y q
TRAER(p) // En forma de procedimiento o función sin valor de retorno
q = TRAER() // En forma de función

// Acceder a los campos del nodo
Leer p->info // Forma de acceder al campo info del nodo con el puntero p
p->liga = nulo // Guarda un nulo en el campo liga (link) del nodo

...

// La función LIBERAR(x) elimina de la memoria la referencia x
LIBERAR(p)
LIBERAR(q)
...
```

En el lenguaje C/C++ se utiliza la función **malloc(x)** para la asignación dinámica de memoria; en Java, al usar clases para definir el nodo, simplemente instanciamos la clase que los crea con la sentencia **new()**.

La función utilizada en C/C++ para liberar la memoria es **free(x)**; en Java no se requiere de una función para liberar memoria, ya que al trabajar con objetos, estos son eliminados automáticamente por el **garbage collector (recolector de basura)** cuando se pierden las referencias o ya no son usados, facilitando aún más el trabajo con este tipo de estructura de datos y con la gestión de la memoria.

Las referencias a los nodos están dadas por direcciones a la memoria expresadas en **números en hexadecimal**; al trabajar con objetos, estamos trabajando con referencias, por lo que el trabajo con listas ligadas en Java se hace relativamente cómodo comparado con la complejidad que ofrece al respecto un lenguaje como C/C++.

### **Ejemplo 2.3**

Crear la clase para gestionar la LSL. La clase contendrá inicialmente las propiedades **head** y **n**; **head** será el puntero al nodo cabeza (primer nodo de la lista) y también permitirá establecer si la lista está vacía (**head = null**); **n** será usado para tener el total de nodos en la lista. Los métodos de esta clase se especifican en desarrollos más adelante, sólo se indica el constructor.

Pseudocódigo:

```
Clase ListaSimplementeLigada
    publico Nodo cab; // O también: punteros p
    publico enteros n;

    Constructor ListaSimplementeLigada()
        cab = nulo;
        n = 0;
    FinConstructor
FinClase
```

Programa Java:

```
package com.packages.linked_list;

import java.util.Scanner;

public class LinkedList
{
    public static Scanner input = new Scanner(System.in);
    public Node head;
    public int n;

    public LinkedList()
    {
        head = null;
        n = 0;
    }
}
```

Creación de una LSL por el inicio

En este método, los nodos quedan almacenados en **orden inverso** al orden en que se ingresan. En términos gráficos, podemos decir que la lista se crea de “**derecha a izquierda**”.

#### Ejemplo 2.4

Crear una LSL por el inicio

Pseudocódigo:

```
publico Metodo crearInicioLSL()
    punteros p; // O también: Nodo p
```

```
cadenas resp;
Repetir
    p = TRAER(); // O también: p = nuevo Nodo()
    Imprimir "Ingrese un número a la LSL: ";
    Leer p->info;
    p->liga = cab;
    cab = p;
    n++;
    Imprimir "¿Agregar más nodos?: ";
    Leer resp;
    Hasta Que resp <> "s";
FinMetodo
```

Programa Java:

```
public void createStartLSL()
{
    Node p;
    String resp;
    do {
        p = new Node();
        System.out.print("Ingrese un número a la LSL: ");
        p.info = input.nextInt();
        p.link = head;
        this.head = p;
        this.n++;
        System.out.print("¿Agregar más nodos?: ");
        resp = input.next();
    } while (resp.equals("s"));
}
```

Creación de una LSL por el final

En este método, los nodos quedan almacenados en el **orden natural** en que se ingresan.  
En términos gráficos, podemos decir que la lista se crea de “**izquierda a derecha**”.

### Ejemplo 2.5

Crear una LSL por el final

Programa Java:

```
public void createEndLSL()
{
    Node p;
    String resp;
    do {
        p = new Node();
        System.out.print("Ingrese un número a la LSL: ");
        p.info = input.nextInt();
```

```

        p.link = null;
        this.n++;
        if (this.head == null) {
            this.head = p;
        } else {
            this.last.link = p;
        }
        this.last = p;
        System.out.print("¿Añadir más nodos?: ");
        resp = input.next();
    } while (resp.equals("s"));
}

```

## Operaciones sobre listas

Son varias las operaciones que pueden realizarse sobre listas ligadas en general, independientemente del tipo de información que éstas contengan y del tipo de lista ligada que se esté tratando. Además de las operaciones de creación por el inicio y final de la lista vistas en los ejemplos anteriores, existen otras operaciones fundamentales sobre ellas:

- Recorrer la lista
- Buscar un dato
- Modificar un dato
- Eliminar un nodo
- Insertar un nodo antes de una referencia dada
- Insertar un nodo después de una referencia dada

### Recorrer la lista

Esta operación, similar a la usada en arreglos unidimensionales, consiste en visitar cada elemento de la lista, en otras palabras, desplazarnos sobre cada nodo de la lista. Al recorrerla, podemos realizar distintas operaciones sobre la información contenida en los nodos, como imprimirla por ejemplo.

### Ejemplo 2.6

Recorrer la lista para mostrar sus elementos (información contenida en cada nodo) e informar cuantos nodos hay en ella.

Pseudocódigo:

```

publico Metodo recorrerLSL()
    punteros p = cab; // O también: Nodo p
    Mientras p <> nulo
        Imprimir p->info;
        p = p->liga;
    FinMientras
    Imprimir "Total nodos: ", n;

```

### FinMetodo

Programa Java:

```
public void scrollLSL()
{
    Node p = this.head;
    while (p != null) {
        System.out.print(p.info + "\t");
        p = p.link;
    }
    System.out.println("\nTotal nodos: " + this.n);
}
```

Buscar un dato en la lista

La función en este caso puede devolver la referencia (puntero) al nodo o un valor booleano para indicar que el dato fue o no encontrado. En el ejemplo a continuación, implementaremos la primera opción.

De igual forma que con los arreglos (vectores), esta búsqueda será útil tanto para la modificación de información como la inserción y eliminación de nodos.

### Ejemplo 2.7

Buscar un dato en la lista

Programa Java:

```
public Node searchLSL(int datum)
{
    Node p = this.head;
    Node pSearch = null;
    while (p != null && pSearch == null) {
        if (p.info == datum) {
            pSearch = p;
        } else {
            p = p.link;
        }
    }
    return pSearch;
}
```

Modificar un dato de la lista

En esta operación se busca el dato a modificar, se solicita el nuevo dato y se procede al cambio. Para ello, utilizaremos la búsqueda sobre la LSL.

### **Ejemplo 2.8**

Modificar un dato de la lista

Programa Java:

```
public void updateNodeLSL(Node p, int datum)
{
    p.info = datum;
}
```

Insertar un dato en la lista

En esta operación se busca un dato de referencia para insertar antes o después de él, el nuevo dato; se debe solicitar también el nuevo dato para ser agregado a la lista si la referencia fue encontrada. Veamos los dos casos.

### **Ejemplo 2.9**

Insertar un dato en la lista después de otro dato dado como referencia

Programa Java:

```
public boolean insertAfterNode(int dr, int di)
{
    boolean sw = false;
    Node p, z;
    p = this.head;
    while (p != null && !sw) {
        if (p.info == dr) {
            z = new Node();
            z.info = di;
            z.link = p.link;
            p.link = z;
            this.n++;
            sw = true;
        } else {
            p = p.link;
        }
    }
    return sw;
}
```

### **Ejemplo 2.10**

Insertar un dato en la lista antes de otro dato dado como referencia

Programa Java:

```
public boolean insertBeforeNode(int dr, int di)
{
    boolean sw = false;
    Node z;
```

```
if (this.head.info == dr) {
    this.n++;
    sw = true;
    z = new Node();
    z.info = di;
    z.link = this.head;
    this.head = z;
} else {
    Node p = this.head.link;
    Node q = this.head;
    while (p != null && !sw) {
        if ( p.info == dr) {
            this.n++;
            sw = true;
            z = new Node();
            z.info = di;
            z.link = p;
            q.link = z;
        } else {
            p = p.link;
            q = q.link;
        }
    }
}
return sw;
}
```

### Eliminar un dato de la lista

En esta operación se busca el dato a eliminar; si se encuentra, se procede a eliminarlo.

#### **Ejemplo 2.11**

Eliminar un dato de la lista

Programa Java:

```
public boolean deleteNode(int datum)
{
    boolean sw = false;
    if (this.head.info == datum) {
        this.n--;
        sw = true;
        this.head = this.head.link;
    } else {
        Node p = this.head.link;
        Node q = this.head;
        while (p != null && !sw) {
            if ( p.info == datum) {
```

```

        this.n--;
        sw = true;
        q.link = p.link;
    } else {
        p = p.link;
        q = q.link;
    }
}
return sw;
}

```

## Lista Simplemente Ligada Circular (LSCC)

Una LSLC es muy similar a una LSL, la única diferencia es que el nodo que sigue al último, es el primero, esto es, el último nodo de la lista apunta al registro **cabeza**.

La gráfica 2.3 muestra una LSLC que contiene en su campo *info* números enteros y su primer registro (registro cabeza) es apuntado por la variable apuntador *head*. Un puntero *p* apunta a un nodo arbitrario de la lista. El campo *link* contiene la *dirección (referencia)* al siguiente nodo; al ser circular la lista, el siguiente nodo al último, es el primero.

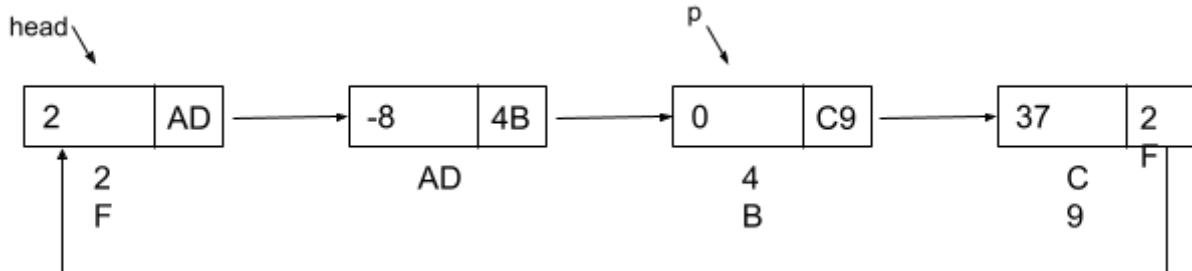


Figura 2.3. Representación de una LSLC que contiene en su campo *info* números enteros y su primer registro es apuntado por la variable apuntador *head*. Un puntero *p* apunta a un nodo arbitrario de la lista. El campo *link* contiene la *dirección (referencia)* al siguiente nodo; al ser circular la lista, el siguiente nodo al último, es el primero.

Para el caso de una LSLC con un solo nodo, éste se apunta a sí mismo, tal y como se muestra en la figura 2.4.

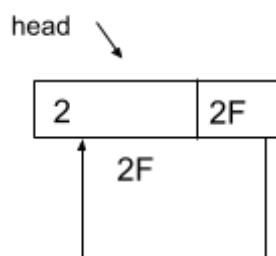


Figura 2.4. Representación de una LSLC con un solo nodo que apunta a sí mismo.

Las operaciones sobre una LSLC son las mismas a las vistas para una LSL, pero deben realizarse algunos ajustes en la implementación de los algoritmos, teniendo en cuenta que ya se está tratando con una LSLC y deben tenerse presente todos los casos posibles. Se ilustrarán algunas operaciones y las demás se dejan como ejercicio a los estudiantes.

### **Ejemplo 2.12**

Crear una clase para gestionar LSLC. La clase es similar a la creada para LSL; se ilustra aquí la creación de la lista por el final y el desplazamiento por cada nodo de la lista para mostrar su contenido; las demás operaciones se dejan como ejercicio para el estudiante. Se mantendrá un puntero al último registro para facilitar la creación de la lista.

Programa Java:

```
package com.packages.linked_list;

import java.util.Scanner;

public class LinkedListCircular
{
    public static Scanner input = new Scanner(System.in);
    public Node head, last;
    public int n;

    public LinkedListCircular()
    {
        this.head = null;
        this.last = null;
        this.n = 0;
    }

    public void createEndLSLC()
    {
        Node p;
        String resp;
        do {
            p = new Node();
            System.out.print("Ingrese un dato en la LSLC: ");
            p.info = input.nextInt();
            if (this.head == null) {
                this.head = p;
            } else {
                last.link = p;
            }
            p.link = this.head;
            last = p;
            this.n++;
        }
    }
}
```

```

        System.out.print("¿Desea agregar más nodos? [s/?]: ");
        resp = input.next();
    } while (resp.equals("s"));
}

public void scrollLSLC()
{
    Node p;
    System.out.print(this.head.info + "\t");
    p = this.head.link;
    while (p != this.head) {
        System.out.print(p.info + "\t");
        p = p.link;
    }
}
}

```

## Listas Dblemente Ligadas (LDL)

Un nodo en una LDL está formado al menos por tres campos:

1. **Liga Izquierda LI (Left Link LL)**: contiene la referencia al nodo anterior o un valor nulo en su defecto
2. **Liga Derecha LD (Right Link RL)**: contiene la referencia al nodo siguiente o un valor nulo en su defecto
3. **Información (Information)**: contiene los datos que se almacenan en el nodo. Al igual que con las LSL y LSCL, este “campo” puede ser en realidad un registro tan complejo como se quiera; para los ejemplos a tratar, continuaremos con un campo de tipo entero

La figura 2.5 muestra la estructura de un nodo de una LDL

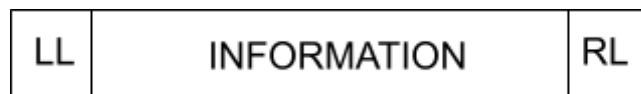


Figura 2.5. Estructura de un nodo en una LDL

Al tener dos punteros en cada nodo, se puede avanzar o retroceder sobre la lista, esto es, ir hacia adelante o hacia atrás de ésta.

Las operaciones sobre LDL son las mismas que las implementadas en LSL, pero al poder avanzar hacia atrás o adelante de la lista, éstas operaciones se hacen más sencillas. Veremos algunos ejemplos de éstas en los ejemplos presentados más adelante.

Primero vamos a definir la estructura del nodo para una LDL.

### **Ejemplo 2.13**

Clase **NodeLDL** (**NodoLDL**) para crear nodos en una LDL; la clase básicamente es un registro que contiene los campos **info** de tipo **int** (**enteros**), **ll** (**left link - liga izquierda li**) y **rl** (**right link - liga derecha ld**) de tipo **Node** (**Nodo**).

Pseudocódigo:

```
Clase NodoLDL
    publico enteros info
    publico NodoLDL ll
    publico NodoLDL ld
FinClase
```

Programa Java:

```
package com.packages.linked_list;

public class NodeLDL
{
    public int info;
    public NodeLDL ll;
    public NodeLDL rl;
}
```

### **Ejemplo 2.14**

Crear una clase para gestionar LDL. La clase es similar a la creada para LSL; se ilustran varios métodos para procesar la información de la lista, las demás operaciones se dejan como ejercicio para el estudiante. Se mantendrá un puntero al último registro para facilitar la creación y procesamiento de datos en la lista.

Pseudocódigo:

```
Clase ListaDoblementeLigada
    publico NodoLDL cab, ult // O también: publico punteros cab, ult
    publico enteros n

    Constructor ListaDoblementeLigada()
        cab = nulo
        ult = nulo
        n = 0
    FinConstructor

    publico Metodo crearFinalLDL()
        NodoLDL p // O también: punteros p
        cadenas resp
        Repetir
            p = nuevo NodoLDL(); // O también: TRAER(p); p=TRAER()
            Imprimir "Ingrese un dato en la LDL: "
            Leer p->info
            p->ld = nulo
```

```

        p->li = ult
        Si cab == nulo Entonces
            cab = p
        SiNo
            ult->ld = p
        }
        ult = p
        n++
        Imprimir "¿Desea agregar más nodos? [s/?]: "
        Leer resp
        Hasta Que resp <> "s"
FinMetodo

publico Metodo recorrerLDL()
    NodoLDL p
    p = cab
    Mientras p <> null
        Imprimir p->info
        p = p->ld
    FinMientras
FinMetodo

publico Metodo buscarLDL(Enteros d)
    NodoLDL p = cab
    Logicos sw = Falso
    Mientras p <> null && !sw
        Si p->info == d
            sw = Verdadero
        SiNo
            p = p->ld
        FinSi
    FinMientras
    Retornar p
FinMetodo

publico Metodo eliminarLDL(NodoLDL p)
    Si p == cab
        cab = cab->ld
    Si cab <> nulo Entonces
        cab->li = nulo
    FinSi
    SiNo
        Si p == ult
            ult = ult->li
            ult->ld = nulo
        SiNo
            (p->li)->ld = p->ld;
    
```

```
        (p->ld)->li = p->li;
    FinSi
FinSi
n--;
FinMetodo

publico Metodo insertAntesLDL(NodoLDL p, Enteros d)
    NodoLDL q = nuevo NodoLDL()
    q->info = d
    q->ld = p;
    Si p == cab
        q->li = nulo
        cab = q;
    SiNo
        (p->li)->ld = q
        q->li = p->li
    FinSi
    p->li = q
    n++
FinMetodo
FinClase
```

Programa Java:

```
package com.packages.linked_list;

import java.util.Scanner;

public class DoubleLinkedList
{
    public static Scanner input = new Scanner(System.in);
    public NodeLDL head, last;
    public int n;

    public DoubleLinkedList()
    {
        this.head = null;
        this.last = null;
        this.n = 0;
    }

    public void createEndLDL()
    {
        NodeLDL p;
        String resp;
        do {
            p = new NodeLDL();
```

```
        System.out.print("Ingrese un dato en la LDL: ");
        p.info = input.nextInt();
        p.rl = null;
        p.ll = last;
        if (this.head == null) {
            this.head = p;
        } else {
            last.rl = p;
        }
        last = p;
        this.n++;
        System.out.print("¿Desea agregar más nodos? [s/?]: ");
        resp = input.next();
    } while (resp.equals("s"));
}

public void scrollLDL()
{
    NodeLDL p;
    p = this.head;
    while (p != null) {
        System.out.print(p.info + "\t");
        p = p.rl;
    }
}

/**
 * @param d
 * @return p
 */
public NodeLDL searchLDL(int d)
{
    NodeLDL p = this.head;
    boolean sw = false;
    while (p != null && !sw) {
        if (p.info == d) {
            sw = true;
        } else {
            p = p.rl;
        }
    }
    return p;
}

/**
 * @param p
 */
```

```

public void deleteLDL(NodeLDL p)
{
    if (p == this.head) {
        this.head = this.head.rl;
        if (this.head != null) {
            this.head.ll = null;
        }
    } else if (p == this.last) {
        this.last = this.last.ll;
        this.last.rl = null;
    } else {
        (p.ll).rl = p.rl;
        (p.rl).ll = p.ll;
    }
    this.n--;
}

public void insertBeforeLDL(NodeLDL p, int d)
{
    NodeLDL q = new NodeLDL();
    q.info = d;
    q.rl = p;
    if (p == this.head) {
        q.ll = null;
        this.head = q;
    } else {
        (p.ll).rl = q;
        q.ll = p.ll;
    }
    p.ll = q;
    this.n++;
}
}

```

## Lista Dblemente Ligada Circular (LDLC)

Una LDLC es muy similar a una LDL, la única diferencia es que el nodo que sigue al último, es el primero, esto es, el último nodo de la lista apunta al registro **cabeza**, además, ésta también apunta al último registro, es decir, el nodo anterior al nodo cabeza es el último.

## Preguntas

1. ¿Qué es una lista ligada y pará qué sirve?
2. ¿Qué es un nodo cabeza y cuál es su importancia?
3. ¿Son mejores las listas ligadas que los vectores?

4. ¿Cuáles son las diferencias entre LSL, LSLC, LDL y LDLC?
5. ¿En qué casos debemos utilizar listas ligadas o arreglos unidimensionales?
6. ¿En qué otros lenguajes, además de Java, se pueden implementar listas ligadas?
7. ¿Por qué se dice que las listas ligadas son estructuras de datos dinámicas lineales y cuál es su diferencia con las llamadas estáticas?
8. ¿Qué otras estructuras de datos dinámicas existen, además de las listas ligadas?
9. ¿Qué otras estructuras de datos lineales existen, además de las listas ligadas?
10. ¿Qué tan compleja puede ser la definición de un nodo?
11. ¿Por qué en Java no liberamos memoria luego de utilizar variables de tipo apuntador?
12. ¿Qué es un puntero y para qué se utilizan?

## Ejercicios

1. Leer una cantidad indeterminada de números y guardarlos en una LSL los números positivos y en otra los negativos. Encuentre en ambas listas:
  - a. Tamaño de cada lista
  - b. Cuál lista es la más extensa
  - c. Suma de datos (sumatoria)
  - d. Producto de datos (productoria)
  - e. Promedio de datos
  - f. Mayor dato
  - g. Menor dato
  - h. Moda
  - i. Mediana
2. Implemente el algoritmo de la burbuja para ordenar los datos en una lista ligada. Considere todos los tipos de listas
3. Implemente el uso de LSL para representar polinomios y realizar operaciones sobre ellos. Un polinomio es una expresión algebraica de la forma:  $a_nx^n + a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0$ . con  $a_i \in \mathbb{R}$ . El nodo debe estar configurado para permitir agregar el coeficiente  $a_i$  ( $i = 0, \dots, n$ ) y la potencia  $i$ . El siguiente es un ejemplo de un polinomio con los coeficientes 10, 5, -7, 0 y 2 y las potencias 0, 1, 2, 3, y 4 respectivamente:  $2x^4 - 7x^2 + 5x + 10$ . Realice las siguientes operaciones:
  - a. Sumar dos polinomios
  - b. Restar dos polinomios
  - c. Multiplicar dos polinomios
4. Implemente las operaciones realizadas en LSL sobre LSLC, LDL y LDLC para:
  - a. Recorrer la lista
  - b. Ordenar la lista
  - c. Buscar un dato
  - d. Modificar un dato
  - e. Eliminar un nodo
  - f. Insertar un nodo antes de una referencia dada
  - g. Insertar un nodo después de una referencia dada

5. Inserte en un dato en una lista ligada que se encuentra ordenada. Considere todos los tipos de listas.
6. Aplicaciones con conjuntos. Sean A y B dos conjuntos. Cree dos listas ligadas y realice las operaciones básicas sobre conjuntos (recuerde que un conjunto no contiene elementos repetidos)
  - a. La cardinalidad de los conjuntos
  - b. Determine si A y B son iguales:  $A = B$
  - c. Unión:  $A \cup B$
  - d. Intersección:  $A \cap B$
  - e. Diferencia:  $A - B$
  - f. Diferencia simétrica:  $A \Delta B$
  - g. Complemento de un conjunto cualquiera X ( $X = A$  o  $X = B$ ):  $X'$
  - h. Conjunto potencia de un conjunto cualquiera X ( $X = A$  o  $X = B$ ):  $P(X)$
  - i. Producto cartesiano:  $A \times B$  y  $B \times A$
7. Elimine todas las ocurrencias de un dato dado en una lista ligada. Considere todas las variantes de listas
8. Elimine los datos repetidos en una lista ligada. Considere todos los tipos de listas
9. En un arreglo se almacena el nombre de distintas recetas de comidas. Cada posición del arreglo contiene dos campos: un campo para guardar el nombre de la receta y otro con una dirección a una lista ligada, la cual contiene los ingredientes de la receta. Cree un programa que permita modelar esta situación y además pueda realizar las siguientes operaciones:
  - a. Crear las recetas
  - b. Mostrar los ingredientes de una receta dada
  - c. Agregar ingredientes a una receta dada
  - d. Quitar ingredientes a una receta dada
  - e. Agregar nuevas recetas
  - f. Eliminar recetas
  - g. ¿Cuál receta contiene más ingredientes?
10. En una LDL se almacena el nombre y edad de un grupo de personas. Realice lo siguiente:
  - a. Cree un menú de opciones para gestionar la información de las personas, según los siguientes literales. Adicione una opción para finalizar el programa.
  - b. Registrar personas
  - c. Listado general de personas con todos sus datos
  - d. Total de personas y promedio de edades
  - e. Convertir la LDL en LDLC. Si la lista ya fue convertida a LDLC, permitir convertirla de nuevo en LDL. Muestre el tipo de lista actual

# Capítulo 3. Pilas y Colas

## Introducción

Las **pilas** y **colas** son estructuras de datos abstractas, en el sentido que no existe como tal un tipo de representación directa en los lenguajes de programación; ambas estructuras de datos se representan entonces usando arreglos (vectores) o listas ligadas (en cualquiera de sus tipos), ya que también son estructuras de datos lineales, pues sus elementos ocupan lugares sucesivos en la estructura.

Tanto en el uso de arreglos unidimensionales como listas ligadas, vimos que se pueden agregar o quitar elementos que se encuentren en cualquier parte de estas estructuras; para el caso de las pilas y colas, la inserción y la eliminación deben realizarse por alguno de sus extremos.

## Pilas

Una **pila** es una estructura de datos en donde los elementos se insertan y eliminan por uno de los extremos. Los elementos se eliminan en orden inverso al ingresado, esto es, el último elemento en agregarse, será el primero en eliminarse. Debido a esto, una pila es conocida como una estructura **LIFO** (**Last In First Out -Último en Entrar Primero en Salir-**).

Ejemplos de pilas en la vida real hay muchos: una pila de platos, las cajas arrumadas en una bodega, productos acomodados en estanterías uno debajo del otro en estanterías, etc. En informática las pilas se aplican en el manejo de la memoria RAM, el tratamiento de expresiones aritméticas y la llamada a subprogramas, entre otras aplicaciones.

Si tratamos de ingresar elementos a una pila llena, se obtendrá un error de **desbordamiento (overflow)**. En caso de querer eliminar un elemento de una pila vacía, se obtiene un error de **subdesbordamiento (underflow)**.

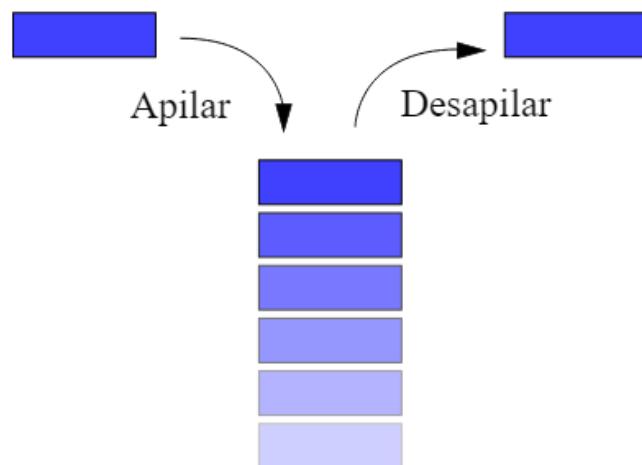


Figura 3.1. Representación de una pila<sup>8</sup>

## Operaciones con pilas

Las dos operaciones fundamentales en pilas son **apilar** (agregar un elemento) y **desapilar** (quitar un elemento) siguiendo la metodología LIFO. En los ejemplos presentados más adelante se ilustran estas sencillas operaciones.

## Aplicaciones con pilas

Son varios los casos donde las pilas tienen uso a nivel informático, algunos de estos son:

1. **Llamadas a subprogramas**: los lenguajes guardan en una pila las llamadas a subprogramas, esto con el fin de poder controlar el **punto de retorno** de éstos una vez finalizan. Los algoritmos recursivos son un ejemplo claro del uso de pilas cuando éstos se ejecutan; estos algoritmos tienen la particularidad de invocarse a sí mismos.
2. **Manejo de la memoria principal (RAM)**: a medida que se abren aplicaciones en un computador, el sistema operativo los guarda en una pila. Si excedemos la capacidad de la pila, esto es, agotamos la memoria, la máquina se bloqueará
3. **Tratamiento de expresiones aritméticas y lógicas**: una operación común en computación, es convertir expresiones aritméticas y lógicas en notación **infija** a notación **prefija** o **postfija**. Los lenguajes de programación traducen las expresiones en notación infija (las que escribimos los humanos) a notación prefija o postfija; con esto, se eliminan los signos de agrupación (paréntesis), ya que el orden de los operadores y operandos indica cómo deben realizarse las operaciones. La notación prefija y postfija se conoce también como **notación polaca**. Veamos algunos ejemplos, teniendo presente la prioridad natural de cada operador usado (aritmético o lógico):

### Ejemplo 3.1

---

<sup>8</sup> Imagen tomada de [Pila \(informática\) - Wikipedia, la enciclopedia libre](#)

Convertir a notación polaca prefija y postfija las siguientes expresiones escritas en notación prefija.

- a. Si  $a + b$  es una expresión algebraica cualquiera:  
Expresión infija:  $a + b$   
Expresión prefija:  $+ab$   
Expresión postfija:  $ab+$
- b. Si  $a + bc = a + b * c$  es una expresión algebraica cualquiera:  
Expresión infija:  $a + b * c$   
Expresión prefija:  $a + *bc = +a*bc$   
Expresión postfija:  $a + bc* = abc*+$
- c. Si  $a * (b + c) - a / d$  es una expresión algebraica cualquiera:  
Expresión infija:  $a * (b + c) - a / d$   
Expresión prefija:  $a * +bc - a / d = *a+bc = *a+bc - /ad = -/ad*a+bc$   
Expresión postfija:  $a * bc+ - a / d = abc+* - a / d = abc+* - ad/ = abc+*ad/-$

### Ejemplo 3.2

Crear una clase para implementar el manejo de pilas

Programa Java:

```
package com.packages.stacks_tails;

public class Stacks
{
    public int stack[];
    public int top;
    public final int MAX = 50;

    public Stacks()
    {
        this.stack = new int[MAX];
        this.top = 0;
    }

    public void stacking(int datum)
    {
        if (this.top < this.MAX) {
            this.stack[this.top] = datum;
            this.top++;
        } else {
            System.err.println("Error de desbordamiento: pila llena
(overflow)");
        }
    }

    public void unStacking()
```

```

{
    if (this.top > 0) {
        this.stack[this.top] = 0;
        this.top--;
    } else {
        System.err.println("Error de subdesbordamiento: pila vacía
(underflow)");
    }
}

public void showStack()
{
    int i;
    for (i = this.top - 1; i >= 0; i--) {
        System.out.print("\n__\n" + this.stack[i]);
        // System.out.print(this.stack[i] + " | ");
    }
}
}

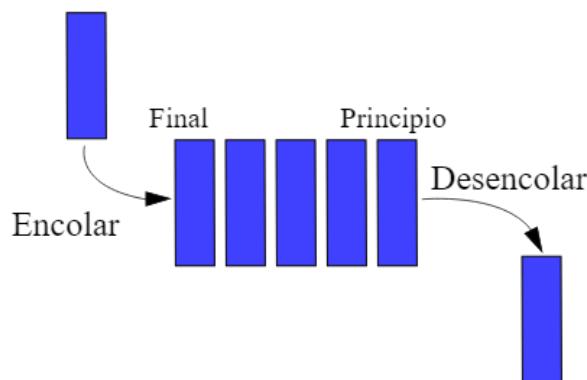
```

## Colas

Una **cola** es una estructura de datos en donde los elementos se insertan por un extremo y se eliminan por el otro. Los elementos se eliminan en el mismo orden al ingresado, esto es, el primer elemento en agregarse, será el primero en eliminarse. Debido a esto, una cola es conocida como una estructura **FIFO (First In First Out -Primero en Entrar Primero en Salir-)**.

Ejemplos de colas en la vida real hay muchos: una cola para ingresar al cine, una fila para retirar dinero en un cajero electrónico, la fila para ingresar a un parqueadero, etc. En informática las colas se aplican cuando se comparten recursos, como por ejemplo en una red se cuenta solo con una impresora y se envían varios trabajos a imprimir; de acuerdo a los tiempos de llegada de cada trabajo, éstos entran en la *cola de impresión* y serán atendidos en orden de llegada. Otro caso es cuando se comparten recursos como memoria, procesador u otros, donde se asignarán a medida que vayan entrando solicitudes, si suponemos que todos tienen la misma prioridad.

Análogamente al caso de las pilas, si tratamos de ingresar elementos a una cola llena, se obtendrá un error de **desbordamiento (overflow)**. En caso de querer eliminar un elemento de una pila vacía, se obtiene un error de **subdesbordamiento (underflow)**.

Figura 3.2. Representación de una cola<sup>9</sup>

## Operaciones con colas

Las dos operaciones fundamentales en colas son **encolar** (agregar un elemento) y **desencolar** (quitar un elemento) siguiendo la metodología FIFO. En los ejemplos presentados más adelante se ilustran estas sencillas operaciones.

## Aplicaciones con colas

Son varios los casos donde las colas tienen aplicaciones en informática. El caso más conocido son las **colas de impresión**, en donde varias estaciones de trabajo comparten una sola impresora; cuando se envían trabajos para imprimir, éstos entran en cola y son atendidos una vez sale el que ya fue atendido (impreso). Otro caso se da cuando se comparten recursos de un servidor, tales como la memoria, el procesador u otros, en donde se atienden de acuerdo al orden de llegada, suponiendo que todas las solicitudes tienen la misma prioridad.

### Ejemplo 3.3

Crear una clase para implementar el manejo de colas

Programa Java:

```
package com.packages.stacks_tails;

public class Tails
{
    public int tail[];
    public int end;
    public final int MAX = 50;

    public Tails()
    {
        this.tail = new int[MAX];
        this.end = 0;
    }
}
```

<sup>9</sup> Imagen tomada de [Cola \(informática\) - Wikipedia, la enciclopedia libre](#)

```
}

public void pushTail(int datum)
{
    if (this.end < this.MAX) {
        this.tail[this.end] = datum;
        this.end++;
    } else {
        System.err.println("Error de desbordamiento: cola llena
(overflow)");
    }
}

public void popTail()
{
    if (this.end > 0) {
        this.deleteElementTail(0);
        this.end--;
    } else {
        System.err.println("Error de subdesbordamiento: cola vacía
(underflow)");
    }
}

public void deleteElementTail(int pos)
{
    int i;
    for (i = pos; i < this.end - 1; i++) {
        this.tail[i] = this.tail[i + 1];
    }
}

public void showTail()
{
    int i;
    for (i = 0; i < this.end; i++) {
        System.out.print(this.tail[i] + " | ");
    }
}
```

## Preguntas

## Ejercicios

4. Convertir a notación polaca prefija y postfija las siguientes expresiones escritas en notación infija.
  - a.  $x^y + z^w$
  - b.  $x - (w / z)^y$
  - c.  $a / (a - b) * (c + b)$
  - d.  $m / (n - q)^p$
  - e.  $a * (b + (x * y))^c$
  - f.  $z^{(x + y + z)} / w$
  - g.  $(a + b) - c * (d + e^x)$
  - h.  $a + b + c + d + e$
  - i.  $(x + y + z)^e / a + (b * c)$
  - j.  $((m - n)^p + q)^r / s$
5. Escriba funciones para convertir expresiones escritas en notación infija a notación polaca prefija y postfija, respectivamente.
6. Problema sobre Colas. Una institución educativa universitaria requiere para el departamento de Admisiones y Registro un control básico de las personas que atiende en el día por ventanilla. Las personas se atienden una a una a medida que van llegando, y se guarda de ellas el documento, el nombre y por defecto se establece que no ha sido atendida. A medida que llegan personas, se ubican en la fila y van saliendo de ésta una vez son atendidas.
  - a. Cree un menú de opciones para gestionar la atención de personas de acuerdo con los siguientes literales. Adicione una opción para finalizar.
  - b. Registrar el ingreso a la fila de una persona
  - c. Mostrar la fila de personas a atender
  - d. Cuántas personas hay en espera
  - e. Atender personas
7. Problema sobre Pilas. Una tienda vende dos tipos de productos (ratón -mouse- y teclados) que llegan en cajas y los organizan en estanterías; de cada producto se conoce su código, nombre y precio, y ambos tipos de productos se organizan de forma independiente de acuerdo con su tipo.
  - a. Cree un menú de opciones para gestionar el movimiento del inventario de acuerdo con los siguientes literales. Adicione una opción para finalizar.
  - b. Acomode productos en las estanterías de acuerdo con su tipo registrando su ingreso al inventario
  - c. Liste los productos disponibles de acuerdo con el tipo seleccionado con todos sus datos e indique cuántos hay en dicha estantería
  - d. Muestre el producto de mayor valor de acuerdo a su tipo
  - e. Aliste productos para la venta. Los productos que se alistan para la venta salen del inventario; el usuario debe tener la posibilidad de indicar el tipo de producto a alistar y cuántos productos va a alistar

## Capítulo 4. Recursividad

La **recursión** o **recursividad** la encontramos en distintas situaciones de la vida cotidiana, por ejemplo cuando se hace una llamada internacional a través de un operador de llamadas y éste transfiere la llamada a otro país. Tomar una foto a otra foto también es un caso de recursión. Un ejemplo de recursión infinita se da cuando se sobreponen dos espejos, donde vemos que la imagen se extiende indefinidamente sobre un pasillo. A nivel matemático también se encuentran muchos ejemplos, muchos de ellos llevados a la aplicación computacional.

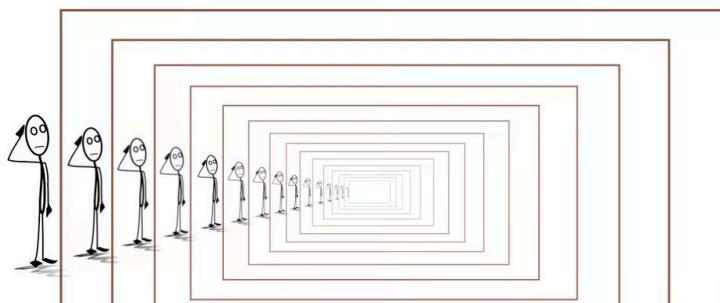


Figura 4.1. Representación de la recursión<sup>10</sup>

Matemáticamente hablando, y por tanto aplicado a la computación, la recursión es una característica de ciertas funciones capaces de definirse en términos de sí mismas. Una función recursiva debe tener o establecer un **estado básico** que no se define de manera recursiva y al cual se van acercando las distintas entradas recursivas de la función; si esto no sucede, se entraría en un ciclo infinito. Por tanto, si un problema cumple con estas características, entonces puede definirse de manera recursiva. Algunos casos de este tipo son el factorial de un número entero, el término  $n$ -ésimo de la serie de Fibonacci, el capital acumulado en  $n$  años en un banco, etc.

Las listas ligadas pueden ser tratadas de forma recursiva, ya que es posible establecer un estado básico y acercarse a él; otro caso son los árboles, una estructura de datos no lineal que trataremos más adelante.

A pesar de la simpleza con que se pueden resolver algunos problemas de forma recursiva, es claro que su entendimiento requiere un mayor esfuerzo para comprender cómo funciona dicho mecanismo. Uno de los aspectos que ayudan a entender el funcionamiento de la recursión, es realizar la prueba de escritorio en donde se utilizan pilas para guardar los estados de las variables y los puntos de retorno.

Es importante tener presente que el uso de la recursión en computación puede conllevar a altos consumos de memoria, por lo que es importante determinar si de verdad se requiere de su implementación en la solución de problemas; por otro lado, algunos problemas que pueden ser tratados de esta forma, es preferible hacerlos en su correspondiente forma iterativa, dado la complejidad que encierra la recursividad. Sin embargo, y como se verá en el tema de **árboles**, la recursión es imprescindible.

<sup>10</sup> Imagen tomada de [Reasons To Use Recursion and How It Works - DEV Community](#)

### **Ejemplo 4.1**

Crear una clase para implementar algunos ejemplos de funciones recursivas en distintos campos, así como casos conocidos ya resueltos de forma iterativa o secuencial.

1. Mostrar los primeros n números naturales
2. Sumar los primeros n números naturales
3. Factorial de un número n
4. Término n-ésimo de la serie de Fibonacci
5. Capital acumulado en n años en un banco.

Pseudocódigo:

```
Clase Recursion
{
    Publico Entero mostrarNumerosNaturales(Entero n)
        Imprimir n
        Si n == 1 Entonces
            Retornar 1;
        SiNo
            Retornar mostrarNumerosNaturales(n - 1);
        FinSi
    FinMetodo

    Publico Entero sumaNumerosNaturales(Entero n)
        Entero sum = n;
        Si n == 1 Entonces
            Retornar 1;
        SiNo
            Retornar sum + sumaNumerosNaturales(n - 1);
        FinSi
    FinMetodo

    Publico Entero factorial(Entero n)
        Si n == 0 Entonces
            Retornar 1
        SiNo
            Retornar n * factorial(n - 1)
        FinSi
    FinMetodo

    Publico Entero fibonacci(Entero n)
        Enteros f
        Si n < 2 Entonces
            f = n
        SiNo
            f = fibonacci(n - 1) + fibonacci(n - 2)
        FinSi
```

```
    Retornar f
FinMetodo

Publico Real capital(Real cant, Entero numA, Real ptje)
    Real: capitalActual
    Si  numA == 0 Entonces
        capitalActual = cant
    SiNo
        capitalActual = (1 + ptje) * capital(cant, numA - 1, ptje)
    FinSi
    Retornar capitalActual
FinMetodo
FinClase
```

Programa Java:

```
package com.packages.recursivity;

public class Recursion
{
    public int showNaturalNumbers(int n)
    {
        System.out.println(n);
        if (n == 1) {
            return 1;
        } else {
            return showNaturalNumbers(n - 1);
        }
    }

    public int sumNaturalNumbers(int n)
    {
        int sum = n;
        if (n == 1) {
            return 1;
        } else {
            return sum + sumNaturalNumbers(n - 1);
        }
    }

    public int factorial(int n)
    {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial(n - 1);
        }
    }
}
```

```

        }
    }

    public int fibonacci(int n)
    {
        int f;
        if (n < 2) {
            f = n;
        } else {
            f = fibonacci(n - 1) + fibonacci(n - 2);
        }
        return f;
    }

    public double capital(double amount, int numYear, double percentage)
    {
        double currentCapital;
        if (numYear == 0) {
            currentCapital = amount;
        } else {
            currentCapital = (1 + percentage) * capital(amount, numYear -
1, percentage);
        }
        return currentCapital;
    }
}

```

## Preguntas

## Ejercicios

1. Recorrer listas LSL, LSLC, LDL y LDLC de forma recursiva
2. La *función de Ackermann*<sup>11</sup> se define como:

$$A(m, n) = n + 1, \text{ si } m = 0$$

$$A(m, n) = A(m - 1, 1), \text{ si } n = 0$$

$$A(m, n) = A(m - 1, A(m, n - 1)), \text{ en otro caso}$$

Esta función toma dos números enteros positivos como argumentos y devuelve un único entero positivo. Escriba una función recursiva para calcular la función de Ackermann  $A(M, N)$  para  $M, N \geq 0$

3. El *algoritmo de Euclides* para el cálculo del *Máximo Común Divisor (MCD)* se define de la siguiente forma:

---

<sup>11</sup> Esta función se debe al matemático alemán Wilhelm Ackermann y tiene interés en las ciencias de la computación. En [Función de Ackermann - Wikipedia, la enciclopedia libre](#) se habla más sobre esta función, así como en otras fuentes de literatura escrita y digital.

$$MCD(m, n) = m, \text{ si } n = 0$$

$$MCD(m, n) = MCD(n, m \bmod n), \text{ si } n > 0$$

Escriba una función recursiva para calcular  $MCD(m, n)$  para  $m, n \geq 0$

4. Invierta una palabra de forma recursiva
5. Escriba funciones no recursivas para solucionar los problemas 2, 3 y 4
6. Muestre el cuadrado de los primeros  $n$  números naturales
7. Muestre la suma de los cuadrados de los primeros  $n$  números naturales de forma recursiva
8. Escriba un programa para simular el problema de *Las Torres de Hanoi*<sup>12</sup>. Resuelva de forma iterativa y recursiva. Este problema fue ideado por el matemático francés Edouard Lucas en el año de 1883 inspirado en una leyenda hindú. El problema consiste en resolver lo siguiente: se tienen tres torres a las que llamamos *origen*, *destino* y *auxiliar*; en la torre de origen se encuentran  $n$  discos todos de distintos tamaños, ordenados de mayor a menor tamaño desde la base de la torre; se deben pasar los discos a la torre de destino usando como apoyo la torre auxiliar con las siguientes condiciones: 1) solo puede pasarse un disco a la vez; 2) no puede quedar un disco de mayor tamaño sobre uno de menor tamaño. Después de muchas pruebas y de analizar este problema, se ha encontrado que el número de movimientos a efectuar es  $2^n - 1$ .

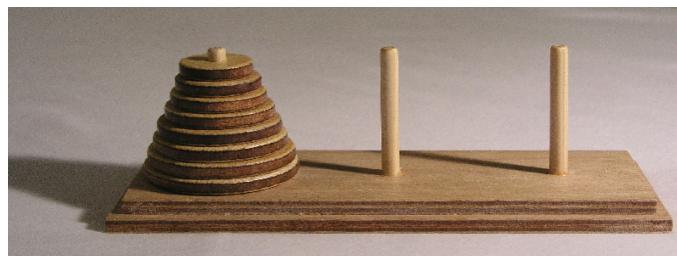


Figura 3.4. Las Torres de Hanoi como juego infantil<sup>13</sup>

---

<sup>12</sup> Puede encontrar más detalles sobre el curioso problema de Las torres de Hanoi en distinta literatura, tanto de matemáticas como de computación. La siguiente fuente habla un poco acerca de la leyenda hindú y una solución usando Python

[4.10. Las torres de Hanoi — Solución de problemas con algoritmos y estructuras de datos](#)

<sup>13</sup> Imagen tomada de: [Torres de Hanói - Wikipedia, la enciclopedia libre](#)

# Capítulo 5. Árboles

Las estructuras de datos estudiadas hasta ahora son **estructuras lineales**, tanto estáticas como dinámicas. Se dice que éstas son lineales, porque a un elemento solo le puede anteceder o suceder otro elementos, esto es, antes o después de un elemento, solo se encuentra, posiblemente, otro elemento.

Un **árbol** es una estructura de datos **no lineal** y **dinámica** compuesta de elementos que pueden ramificarse, esto es, después de un elemento pueden haber otros elementos. Los árboles mantienen una estructura jerárquica sobre un conjunto de elementos conocidos como **nodos**. El nodo principal, y de cual se genera todo el árbol, se conoce como **nodo raíz**. Esta estructura jerárquica permite usar los conceptos de antecesor, sucesor, padre, hijo, hermano, entre otros.

La siguiente tabla muestra las estructuras de datos lineales y no lineales disponibles en computación, algunas ya conocidas:

Estáticas	Dinámicas	Característica	Observación
Arreglos	Listas ligadas	Lineal	
Conjuntos (tratados con vectores)	Pilas	Lineal	Como lista
Registros (tratados como clases sin métodos)	Colas	Lineal	Como lista
Pilas		Lineal	Como vector
Colas		Lineal	Como vector
	Árboles	No Lineal	Es un tipo especial de grafo
	Grafos	No Lineal	

El nombre de árbol a este tipo de estructura de datos, se debe a su similitud con los árboles naturales, como se verá más adelante en las representaciones gráficas. Matemáticamente, un árbol es un tipo especial de **grafo**, y ambos objetos son tema de estudio permanente en este campo, además del computacional.

## Árboles en general

Sea el árbol A de cualquier tipo. La estructura vacía es considerada un árbol, esto es **A = nulo** es un árbol. En general, un árbol A es una estructura homogénea conformada por la unión finita de otros elementos de tipo A, llamados subárboles y que se encuentran disjuntos.

La recursión es una característica inherente a los árboles, por lo que se utilizará esta técnica algorítmica para el tratamiento de éstos.

Los árboles tienen diversas aplicaciones en el mundo real, convirtiendo a este tipo de estructura de datos en una de las más utilizadas para resolver problemas. Un caso muy actual, es el uso de algoritmos basados en árboles para la IA (Inteligencia Artificial); también están las aplicaciones en circuitos, árboles genealógicos, fórmulas matemáticas, indexación de información (como la utilizada en bases de datos -BD-) entre otras.

Un árbol puede representarse de distintas formas, tanto gráficas como tipo conjunto, todas equivalentes. Veamos la siguiente figura:

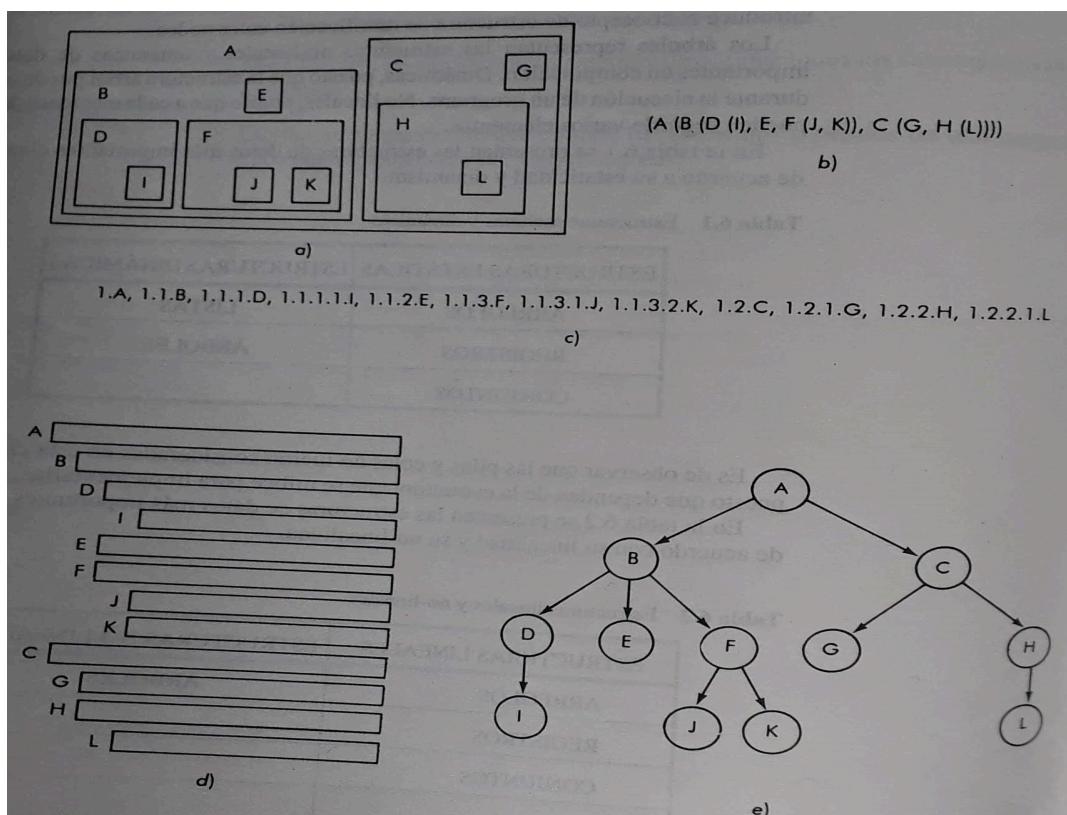


Figura 5.1. Diferentes representaciones de un mismo árbol<sup>14</sup>

- Figura 5.1a) representación por medio de **diagramas de Venn**
- Figura 5.1b) representación por medio de **anidación de paréntesis**
- Figura 5.1c) representación por medio de la **notación decimal de Dewey**<sup>15</sup>
- Figura 5.1d) representación por medio de la **notación indentada**

<sup>14</sup> Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 182.

<sup>15</sup> El Sistema de Clasificación Decimal Dewey es un sistema simple, sólido, lógico y fácil de interpretar, muy útil en bibliotecas y lugares que utilicen clasificaciones de material de manera similar (ver más en [Sistema Decimal Dewey](#)). Este nombre se debe a su creador, un bibliotecario llamado Melvil Dewey (1851-1931), que 1875-1876 creó un sistema numérico decimal para organizar los libros de la biblioteca escolar en la que trabajaba (ver más en [Sistema de Clasificación Dewey | Biblioteca Pública de Denver](#))

- Figura 5.1e) representación por medio de **grafos**

Los grafos son los más utilizados en la representación de árboles, y es debido al parecido (abstracto) con este tipo de planta, que recibe este nombre. En los grafos la raíz del árbol se encuentra en la parte superior, como tomando el árbol invertido.

## Características y propiedades de los árboles

Sea el árbol A de cualquier tipo; se cumplen las siguientes propiedades/características en el árbol:

1. Si  $A \neq$  nulo, entonces tiene un único **nodo raíz**
2. Un nodo cualquiera  $N$  es descendiente directo de un nodo  $M$ , si el nodo  $N$  es apuntado por el nodo  $M$ :  **$N$  es hijo de  $M$**
3. Un nodo cualquiera  $N$  es antecesor directo de un nodo  $M$ , si el nodo  $N$  apunta al nodo  $M$ :  **$N$  es padre de  $M$**
4. Los nodos descendientes directos del mismo nodo  $N$  son **hermanos**
5. Los nodos sin ramificaciones (sin hijos), reciben el nombre de **hojas** o nodos **terminales**
6. Los nodos que no son raíz ni hojas se conocen como nodos **interiores**
7. El número de descendientes directos de un nodo cualquiera se conoce como **grado**
8. El máximo grado de todos los nodos del árbol es el **grado del árbol**
9. Una **arista (camino)** es el enlace (liga) entre dos nodos consecutivos
10. Una **rama** está compuesta por un conjunto de aristas (caminos) que termina en una hoja
11. El número de nodos (o aristas + 1) que deben ser recorridos para llegar a un determinado nodo desde la raíz se conoce como **nivel**. Por definición, la raíz tiene nivel 1 (algunos textos toman la raíz en el nivel 0, siendo el nivel igual al número de aristas desde la raíz hasta el nodo)
12. El máximo número de niveles de todos los nodos del árbol se conoce como **altura** o **profundidad** del árbol. También se puede definir como el número máximo de nodos de una rama
13. El **peso** del árbol es el número de nodos terminales u hojas

### Ejemplo 5.1

Verificar las propiedades y características del árbol T mostrado en la siguiente figura

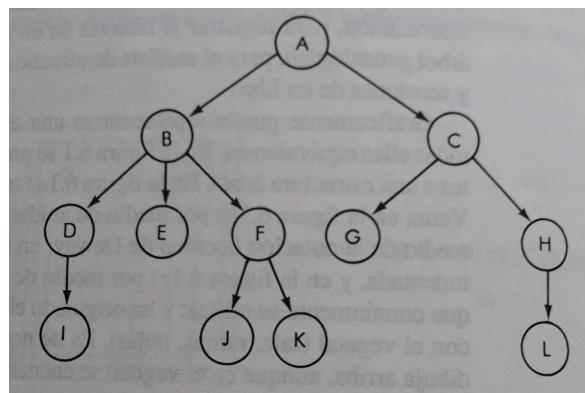


Figura 5.2. Un árbol T cualquiera<sup>16</sup>

1. Nodo raíz: A, con  $A \neq$  nulo
2. Relaciones entre nodos
  - a. B y C son hijos de A (o equivalentemente, A es el padre de B y C), por tanto B y C son hermanos
  - b. D, E y F son hijos de B y por tanto hermanos; también son *nietos* de A
  - c. F es el padre de J y K
  - d. F y G son *primos* y tienen como abuelo a A
  - e. L es bisnieto de A
3. Hojas o nodos terminales: I, E, J, K, G, L
4. Peso del árbol: 6
5. Nodos interiores: B, D, F, C, H
6. Grado de algunos nodos
  - a. Grado de A: 2
  - b. Grado de B: 3
  - c. Grado de C: 2
  - d. Grado de D: 1
  - e. Grado de E: 0
7. Grado del árbol: 3
8. Niveles de los nodos
  - a. Nivel de A: 1
  - b. Nivel de B, C: 2
  - c. Nivel de D, E, F, G, H: 3
  - d. Nivel de I, J, K, L: 4
9. Altura del árbol: 4

### Longitud de camino interno y externo

El nivel de los nodos nos ayuda a encontrar la **Longitud de Camino LC** como el número de nodos que deben ser recorridos para llegar a él desde la raíz; así, para el árbol de la figura 5.2 la raíz tiene una  $LC = 1$ , sus descendientes directos será una  $LC = 2$ , el nivel 3 tendrá una  $LC = 3$ , y análogamente con los demás niveles.

---

<sup>16</sup> Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 184.

## Longitud de Camino Interno (LCI)

Se define la **Longitud de Camino Interno (LCI)** como la suma de las longitudes de camino (LC) de todos los nodos del árbol.

$$LCI = \sum_{i=1}^h n_i * i$$

Donde:

h: altura del árbol

i: nivel actual en el árbol

n<sub>i</sub>: número de nodos en el nivel i

### Ejemplo 5.2

Encontrar la LCI del árbol de la figura 5.2

$$LCI = \sum_{i=1}^h n_i * i$$
$$h = 4; LCI = \sum_{i=1}^4 n_i * i = 1 * 1 + 2 * 2 + 5 * 3 + 4 * 4 = 36$$

Se define también la **Longitud de Camino Interno Medio (LCIM)** como

$$LCIM = LCI/n$$

Donde n es el número de nodos que hay en el árbol.

Esta medida indica la longitud media (promedio) para visitar un nodo cualquiera desde la raíz.

### Ejemplo 5.3

Encontrar la LCIM del árbol de la figura 5.2

$$n = 12; LCI = 36; LCIM = LCI/n = 36/12 = 3$$

## Longitud de Camino Externo (LCE)

Para hallar la **Longitud de Camino Externo** de un árbol, primero debemos encontrar su correspondiente **árbol extendido**, el cual se forma a partir del árbol original añadiendo una serie de **nodos especiales**.

### Árbol extendido

Es un árbol en el que cada nodo tiene un número de hijos igual al grado del árbol.

### Nodo especial

Es aquel que debe agregarse al árbol cuando algunos nodos de éste no cumplen con la condición de árbol extendido. Un nodo cualquiera puede requerir agregar varios nodos

especiales de ser necesario para que cumpla dicha condición. Estos nodos especiales no pueden tener hijos (descendientes) y su objetivo es sustituir **ramas** vacías o nulas.

#### Ejemplo 5.4

Extender el árbol de la figura 5.2.

La siguiente figura muestra el árbol extendido, donde cada nodo tiene un número de hijos igual al grado del árbol. El grado del árbol original es 3, por consiguiente tenemos:

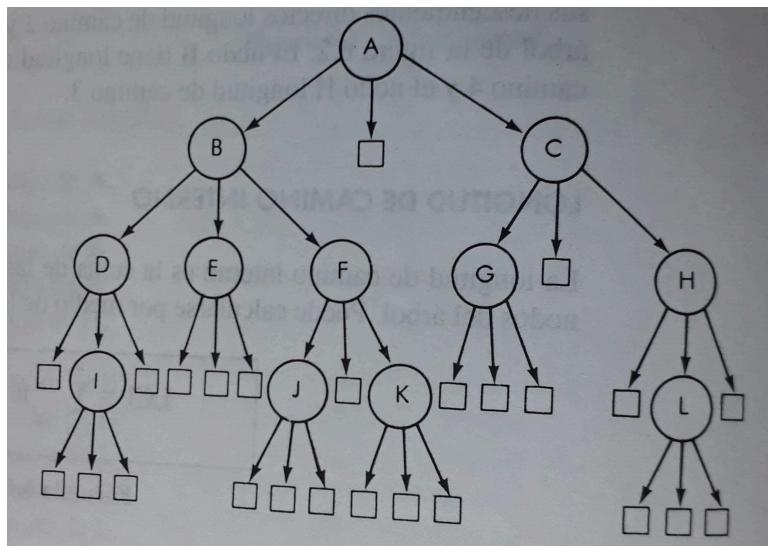


Figura 5.3. Un árbol extendido correspondiente al árbol de la figura 4.2<sup>17</sup>

Observe que estos nodos especiales se representan con cuadrados pequeños.

Para este árbol, hubo que adicionar 25 nodos especiales.

Se define la **Longitud de Camino Externo (LCE)** como la suma de las longitudes de todos los nodos especiales del árbol:

$$LCE = \sum_{i=2}^{h+1} ne_i * i$$

Donde:

*h*: altura del árbol

*i*: nivel actual en el árbol

*ne<sub>i</sub>*: número de nodos especiales en el nivel *i*

#### Ejemplo 5.5

Encontrar la LCE del árbol de la figura 5.3

<sup>17</sup> Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 186.

$$LCE = \sum_{i=2}^{h+1} ne_i * i$$

$$h = 4; LCI = \sum_{i=2}^5 ne_i * i = 1 * 2 + 1 * 3 + 11 * 4 + 12 * 5 = 2 + 3 + 44 + 60 = 109$$

Se define también la **Longitud de Camino Externo Medio (LCEM)** como

$$LCEM = LCE/ne$$

Donde ne es el número de nodos especiales que hay en el árbol.

Esta medida indica la longitud media (promedio) para visitar un nodo especial cualquiera desde la raíz.

### **Ejemplo 5.6**

Encontrar la LCEM del árbol de la figura 5.3

$$ne = 25; LCE = 109; LCEM = LCE/ne = 109/25 = 4.36$$

## Árboles binarios

Un árbol de grado 2 se conoce como **árbol binario** y goza de particular atención en las ciencias computacionales gracias a sus características y fácil tratamiento.

En un árbol binario cada nodo tiene a lo sumo dos descendientes directos, y podemos distinguirlos como la **rama (subárbol) izquierda** y la **rama (subárbol) derecha**. Con un árbol binario se pueden realizar diversas operaciones, como por ejemplo escribir operaciones algebraicas, representar árboles genealógicos, búsquedas sobre información clasificada (**árboles binarios de búsqueda**), entre otras.

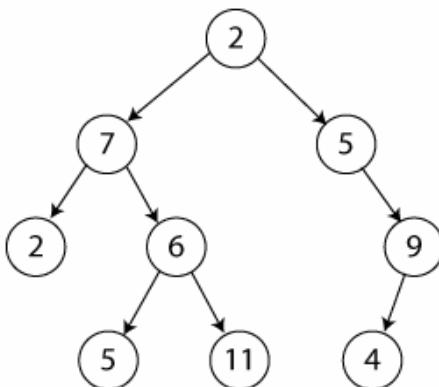


Figura 5.4. Un árbol binario<sup>18</sup>

### **Ejemplo 5.7**

---

<sup>18</sup> Imagen tomada de [Árbol binario - Wikipedia, la enciclopedia libre](#)

En la figura 5.5 se tiene un árbol binario almacenando una expresión aritmética. Escribir la expresión equivalente y mostrar el resultado.

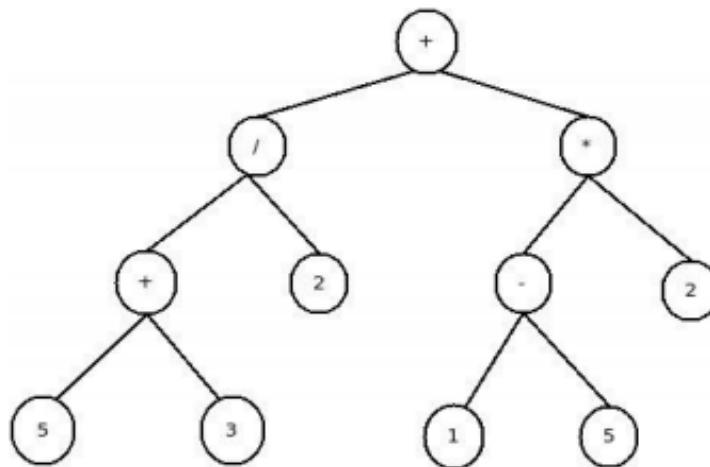


Figura 5.5. Un árbol binario para calcular una expresión aritmética<sup>19</sup>

### Solución

Las operaciones de mayor prioridad se encuentran en el nivel más profundo del árbol, dando prioridad al subárbol izquierdo y siguiendo estas mismas condiciones en cada nivel, en el orden de abajo hacia arriba.

$$\begin{aligned}
 & (((5 + 3) / 2) + ((1 - 5) * 2)) \\
 &= ((5 + 3) / 2) + ((1 - 5) * 2) \\
 &= (5 + 3) / 2 + (1 - 5) * 2 \\
 &= 4 + (-8) \\
 &= -4
 \end{aligned}$$

### Nota

Observe que los paréntesis no son requeridos en una representación por medio de árboles binarios de una expresión aritmética.

### Ejemplo 5.8

Represente mediante un árbol binario la siguiente expresión algebraica

$$(a + 3 * b * c)/(x^2 - y\%5)$$

### Solución

La figura 5.6 muestra la representación como árbol binario de la expresión en cuestión

---

<sup>19</sup> Imagen tomada de [Arbol binario para calcular operaciones simples - Stack Overflow en español](#)

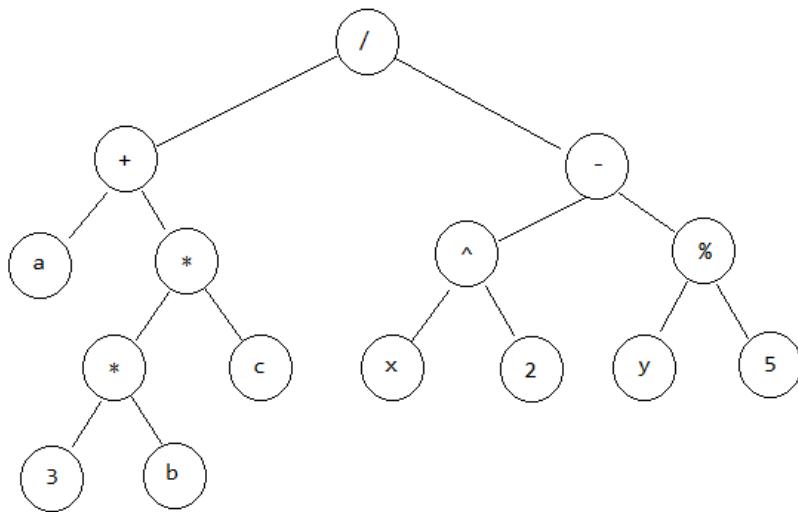


Figura 5.6. Representación de la expresión aritmética del ejemplo 4.8 como un árbol binario

## Árboles binarios distintos

Se dice que dos árboles binarios son **distintos** si difieren en su estructura. En la figura 5.7 se muestran dos casos.

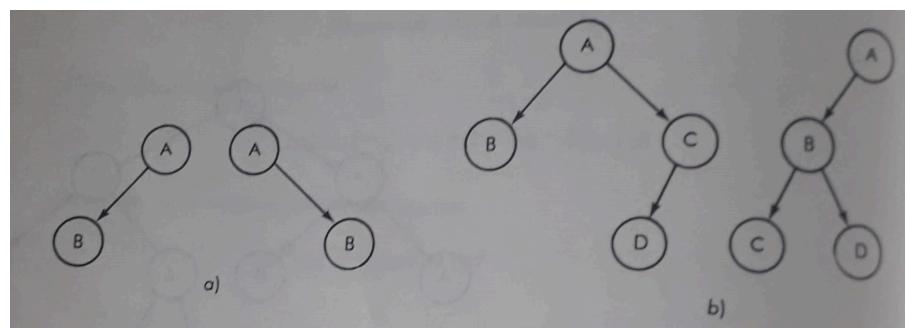


Figura 5.7. Árboles binarios distintos<sup>20</sup>

## Árboles binarios similares

Dos árboles binarios son **similares** si sus estructuras son iguales, pero la información contenida en los nodos difiere. La figura 5.8 muestra un par de casos.

---

<sup>20</sup> Imagen tomada de: CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996, pág 190.

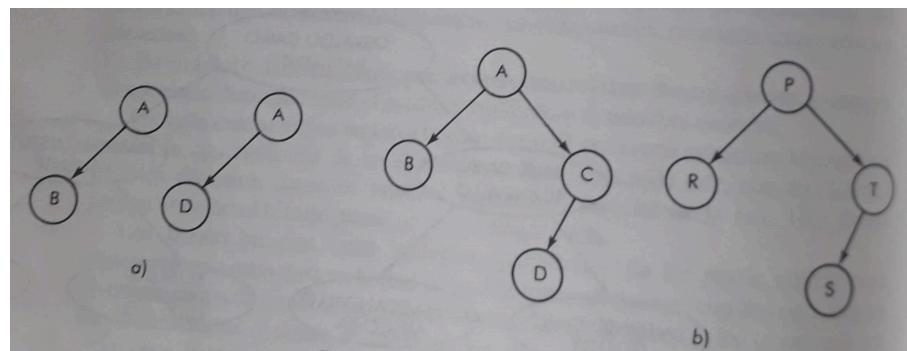


Figura 5.8. Árboles binarios similares<sup>21</sup>

## Árboles binarios equivalentes

Dos árboles binarios son **equivalentes** si son similares y además tienen la misma información en cada nodo correspondiente.

## Árboles binarios completos

Un árbol binario es **completo**, si todos los nodos, excepto los del último nivel, tienen exactamente dos hijos. Los nodos del último nivel son hojas. La figura 5.9 muestra dos árboles binarios completos.

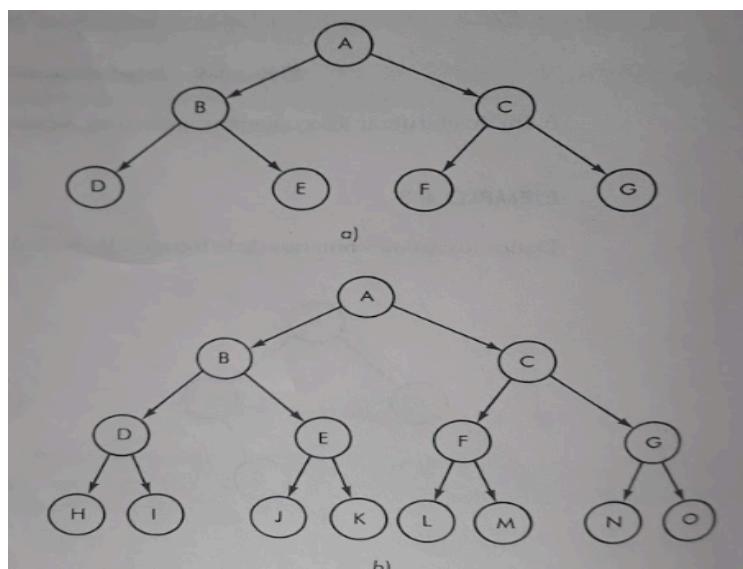


Figura 5.9. Árboles binarios completos<sup>22</sup>

## Árboles binarios degenerados o patológicos

Un árbol binario se considera **degenerado**, si todos los nodos tienen solamente un subárbol, esto es, tienen un solo hijo, excepto el último. En un árbol binario degenerado, la altura del árbol es igual al número total de nodos.

<sup>21</sup> Ibídem

<sup>22</sup> Idem, pág 192

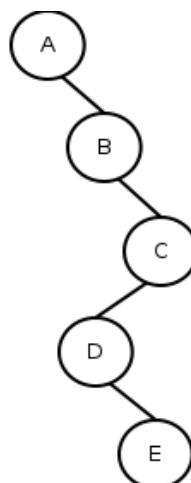


Figura 5.10. Árbol binario degenerado<sup>23</sup>

El número de nodos en un árbol binario completo de altura  $h$  se calcula mediante la expresión:

$$n_{abc} = 2^h - 1$$

Donde:

$n_{abc}$ : Número de nodos del árbol binario completo

$h$ : altura del árbol binario completo

Si conocemos el número de nodos de un árbol binario completo, podemos encontrar su altura  $h$ :

$$n_{abc} + 1 = 2^h \Rightarrow \lg(n_{abc} + 1) = \lg(2^h) \Rightarrow \lg(n_{abc} + 1) = h \lg(2) \Rightarrow \lg(n_{abc} + 1) = h$$

Donde:

$n_{abc}$ : Número de nodos del árbol binario completo

$h$ : altura del árbol binario completo

$\lg$ : Logaritmo en base 2

También podemos conocer el número de nodos de un nivel determinado mediante la expresión:

$$n_l = 2^{l-1}$$

Donde:

$n_l$ : Número de nodos en el nivel  $l$  del árbol binario completo

$l$ : nivel en el árbol binario completo

<sup>23</sup> Imagen tomada de: <https://elarbolinformatico.blogspot.com/2015/>

### **Ejemplo 5.9**

Encontrar el  $n_{abc}$  para los árboles binarios completos de la figura 5.9

Para el árbol de la figura a) se tiene

$$h = 3$$

$$n_{abc} = 2^3 - 1 = 7 \text{ nodos}$$

$$n_3 = 2^{3-1} = 2^2 = 4 \text{ nodos en el nivel 3}$$

Para el árbol de la figura b) se tiene

$$h = 4$$

$$n_{abc} = 2^4 - 1 = 15 \text{ nodos}$$

$$n_3 = 2^{3-1} = 2^2 = 4 \text{ nodos en el nivel 3}$$

$$n_4 = 2^{4-1} = 2^3 = 8 \text{ nodos en el nivel 4}$$

### **Nota**

Algunos textos pueden referirse a los árboles binarios completos como **I llenos**.

## Representación de árboles binarios en memoria

Hay dos formas de representar árboles binarios, mediante punteros o con el uso de arreglos, sin embargo, la naturaleza dinámica de los árboles hace más conveniente el uso de punteros.

Un nodo en un árbol binario es representado como un registro (o clase sin métodos en los lenguajes OO), similar al definido para una LDL, que contiene al menos tres campos (o propiedades en OO). La figura 5.11 muestra dicha representación como un registro:

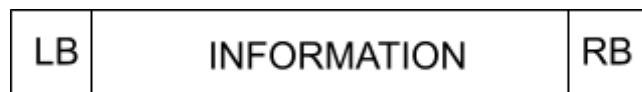


Figura 5.11. Estructura de un nodo en un árbol binario como registro

Los campos de este registro almacenan lo siguiente:

- **INFORMATION - INFORMACIÓN:** en su forma más simple, almacena un dato primitivo (entero, real, lógico, carácter), pero puede ser tan complejo como se quiera para que contenga arreglos, otros registros, objetos, etc.
- **LB (Left Branch) - IZQ (Subárbol Izquierdo):** campo de tipo puntero que guardará la referencia a un nodo apuntado en el subárbol izquierdo o nulo sino apunta a ningún subárbol (árbol vacío)
- **RB (Right Branch) - DER (Subárbol Derecho):** campo de tipo puntero que guardará la referencia a un nodo apuntado en el subárbol derecho o nulo sino apunta a ningún subárbol (árbol vacío)

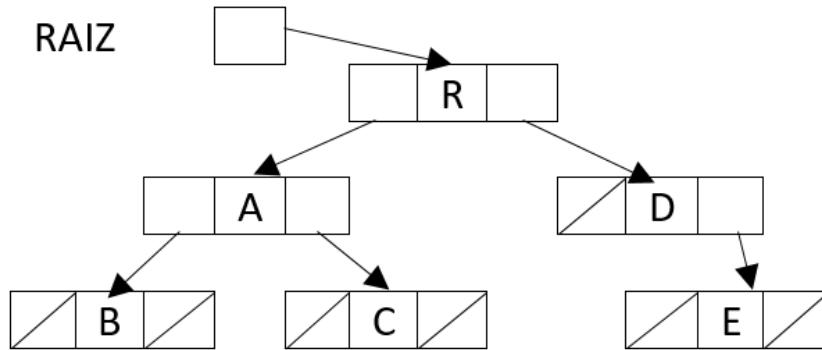


Figura 5.12. Representación de un árbol binario en memoria<sup>24</sup>

## Operaciones con árboles binarios

Las operaciones fundamentales en árboles binarios están relacionadas con su creación, inserción y borrado de nodos, así como visitar sus elementos (recorrer el árbol). Veamos la creación del árbol y como recorrerlo; las otras dos operaciones se tratarán con los árboles binarios de búsqueda.

### Creación de un árbol binario

Esta operación, de manera similar al manejo de listas ligadas, consiste en cargar los nodos que se requieran en la memoria, solo que aquí se hará de forma recursiva. En los ejemplos que siguen a continuación, se ilustra ésta y otras operaciones, entre ellas el desarrollo algorítmico del problema.

### Recorrido en árboles binarios

Esta operación consiste en visitar cada nodo del árbol y efectuar alguna operación sobre la información de éste: mostrarla, usarla en cálculos, etc. Hay tres formas de naturaleza recursiva para realizar esto:

- 1. Recorrido en preorden**
  - a. Visitar la raíz
  - b. Recorrer el subárbol izquierdo
  - c. Recorrer el subárbol derecho
- 2. Recorrido en inorder**
  - a. Recorrer el subárbol izquierdo
  - b. Visitar la raíz
  - c. Recorrer el subárbol derecho
- 3. Recorrido en postorden**
  - a. Recorrer el subárbol izquierdo
  - b. Recorrer el subárbol derecho

<sup>24</sup> Imagen tomada de: [Arboles Binarios de Búsqueda \( ABB \) – DEFINICIÓN, ESTRUCTURA, REPRESENTACIÓN EN MEMORIA DINÁMICA, OPERACIONES, RECORRIDOS – – Programación C++](#)

c. Visitar la raíz

**Ejemplo 5.10**

Mostrar como son los recorridos en el árbol de la figura 5.6

Solución

Preorden:  $/+a^{**}3bc-^x2\%y5$

Inorden:  $a+3*b*c/x^2-y\%5$

Postorden:  $a3b*c+x2^y5\%-/$

**Ejemplo 5.11**

Clase **NodeTree (NodoArbol)** para crear nodos en un árbol binario; la clase, sin métodos, básicamente es un registro que contiene los campos **info** de tipo **int (enteros)**, **lb (left branch - subárbol izquierdo izq)** y **rb (right branch - subárbol derecho der)** de tipo **Node (Nodo)**.

Pseudocódigo:

```
Clase NodoArbol
    publico enteros info
    publico NodoArbol izq
    publico NodoArbol der
FinClase
```

Programa Java:

```
package com.packages.trees;

public class NodeTree
{
    public int info;
    public NodeTree left;
    public NodeTree right;
}
```

**Ejemplo 5.12**

Clase **BinaryTree (ArbolBinario)** para realizar distintas operaciones sobre árboles binarios, entre ellas las siguientes:

- Crear el árbol
- Recorrer el árbol: preorden, inorder y postorden
- Encontrar el total de nodos

Programa Java:

```
package com.packages.trees;
```

```
import java.util.Scanner;

public class BinaryTree
{
    public static Scanner input = new Scanner(System.in);

    public BinaryTree()
    {

    }

    public void loadNode(NodeTree node)
    {
        NodeTree p;
        String resp;

        System.out.print("Ingrese un dato para el árbol: ");
        node.info = input.nextInt();

        System.out.print("¿Desea agregar nodos por el subárbol izquierdo?
[s/?]: ");
        resp = input.next().toLowerCase();
        if (resp.equals("s")) {
            p = new NodeTree();
            node.left = p;
            loadNode(node.left);
        } else {
            node.left = null;
        }

        System.out.print("¿Desea agregar nodos por el subárbol derecho?
[s/?]: ");
        resp = input.next().toLowerCase();
        if (resp.equals("s")) {
            p = new NodeTree();
            node.right = p;
            loadNode(node.right);
        } else {
            node.right = null;
        }
    }

    public void traversePreorder(NodeTree node)
    {
        if (node != null) {
            System.out.println("Nodo: " + node.info);
            traversePreorder(node.left);
        }
    }
}
```

```
        traversePreorder(node.right);
    }
}

public void traverseInorder(NodeTree node)
{
    if (node != null) {
        traverseInorder(node.left);
        System.out.println("Nodo: " + node.info);
        traverseInorder(node.right);
    }
}

public void traversePostorder(NodeTree node)
{
    if (node != null) {
        traversePostorder(node.left);
        traversePostorder(node.right);
        System.out.println("Nodo: " + node.info);
    }
}

public int totalNodes(NodeTree node)
{
    if (node != null) {
        return 1 + totalNodes(node.left) + totalNodes(node.right);
    } else {
        return 0;
    }
}
```

## Representación de árboles generales como árboles binarios

Un árbol general puede convertirse en un árbol binario. El alcance del curso no cubre este aspecto, pero se deja al lector consultar el algoritmo respectivo para realizar esta operación; en los textos de Estructuras de Datos mencionados en la bibliografía se encuentra dicho desarrollo.

## Bosque de árboles

Un conjunto de 2 o más árboles se conoce como Bosque de árboles. Éstos también pueden ser representados mediante árboles binarios. Se deja como ejercicio consultar o desarrollar el algoritmo respectivo.

## Árboles Binarios de Búsqueda (ABB)

Este tipo especial de árbol facilita las operaciones de acceso a los elementos, tales como la búsqueda, inserción y eliminación de nodos. Generalmente, los árboles binarios de búsqueda no contienen información repetida.

La definición formal de un **árbol binario de búsqueda** es la siguiente:

### **Definición: árbol binario de búsqueda**

Para todo nodo N del árbol debe cumplirse que todos los valores del subárbol izquierdo deben ser menores o iguales al valor de N; análogamente, se debe cumplir que todos los valores del subárbol derecho deben ser mayores o iguales al valor de N.

La siguiente figura muestra un árbol binario de búsqueda.

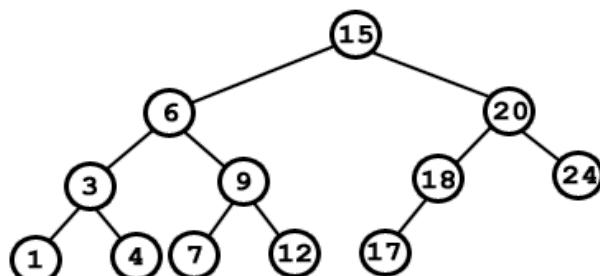


Figura 5.13. Representación de un árbol binario de búsqueda<sup>25</sup>

### **Ejemplo 5.13**

Crear un método para buscar un elemento en un árbol binario de búsqueda en la Clase **BinaryTree**.

Programa Java:

```
public void searchBinary(NodeTree node, int datum)
{
    if (node != null) {
        if (datum < node.info) {
            searchBinary(node.left, datum);
        } else if (datum > node.info) {
            searchBinary(node.right, datum);
        }
    }
}
```

<sup>25</sup> Imagen tomada de: [Operaciones básicas de los árboles binarios de búsqueda](#)

```
        searchBinary(node.right, datum);
    } else {
        System.out.println("Nodo encontrado en el árbol");
    }
} else {
    System.out.println("Nodo no encontrado en el árbol");
}
}
```

### **Ejemplo 5.14**

Crear un método para insertar un elemento en un árbol binario de búsqueda en la Clase **BinaryTree**. Los datos en el árbol se suponen únicos.

Programa Java:

```
public void insertBinary(NodeTree node, int datum)
{
    NodeTree p;
    if (datum < node.info) {
        if (node.left == null) {
            p = new NodeTree();
            p.info = datum;
            p.left = null;
            p.right = null;
            node.left = p;
        } else {
            insertBinary(node.left, datum);
        }
    } else if (datum > node.info) {
        if (node.right == null) {
            p = new NodeTree();
            p.info = datum;
            p.left = null;
            p.right = null;
            node.right = p;
        } else {
            insertBinary(node.right, datum);
        }
    } else {
        System.out.println("El nodo ya se encuentra en el árbol");
    }
}
```

## Preguntas

1. ¿Qué es un árbol?
2. ¿Qué es un árbol binario?

3. ¿Qué es un árbol completo?
4. ¿Qué es un árbol AVL?
5. ¿Qué es un grafo?

## Ejercicios

1. Realice los siguientes cálculos para encontrar el valor numérico de cada expresión si  $a = 3$ ,  $b = 4$ ,  $c = -1$ ,  $d = -5$ ,  $e = 2$ , reescriba cada expresión en notación algorítmica y muestre su representación mediante árboles binarios
  - a.  $ab^2 + 3c - \sqrt{b}$
  - b.  $5e - 2bcd + 4(d^3 - 2a + c) + a \% e$
  - c.  $(\frac{b}{e} + \frac{e}{c}) - 6(\frac{c^2}{a})$
  - d.  $\sqrt[3]{d^6} + \frac{a+b+c}{e} + e - b \% 2$
  - e.  $\sqrt{ab - a} + 5d \div c - a^4 be$
2. Muestre los árboles del punto anterior utilizando la notación de Dewey, notación indentada, notación de paréntesis y mediante diagramas de Venn
3. En un vector se almacena el número de nodos en cada nivel correspondiente a un árbol general. Cree una función que calcule la LCI
4. Si se sabe que el número de nodos de un árbol general es  $n$ , cree una función para calcular la LCIM usando el resultado del ejercicio anterior
5. Calcule el  $NN_{abc}$  para árboles binarios completos de altura 2, 3, 4, 5, 6, 7, 8, 9 y 10, respectivamente
6. Realice las siguientes operaciones sobre un árbol binario de raíz root que contiene números enteros:
  - a. Muestre los números pares e impares
  - b. Muestre el total de números pares
  - c. Muestre el total de números impares
  - d. Encuentre la suma de los números
  - e. Encuentre el promedio de los números
  - f. Encuentre el mayor de los números
  - g. Encuentre el menor de los números
  - h. Muestre los números primos
  - i. Encuentre el total de números primos
  - j. Muestre solo los nodos del subárbol izquierdo
  - k. Muestre solo los nodos del subárbol derecho
  - l. Muestre y cuente las hojas del árbol
  - m. Muestre solo los nodos que tengan exactamente dos hijos
  - n. Buscar un nodo
  - o. Insertar un nodo
  - p. Eliminar un nodo
7. Implemente un algoritmo para crear árboles generales
8. Implemente un algoritmo para convertir árboles generales en árboles binarios

## Capítulo 6. Grafos

En matemáticas y en ciencias de la computación, la teoría de grafos (también llamada teoría de las gráficas) estudia las propiedades de los **grafos** (también llamadas **gráficas**). Un grafo es una estructura discreta compuesta y no vacía, por un conjunto de objetos llamados **vértices** (o **nodos**) y una selección de pares de vértices llamados **aristas** (o **arcos**) (*edges* en inglés) que pueden ser orientados o no. Típicamente, un grafo se representa mediante una serie de puntos (los vértices) conectados por líneas (las aristas).

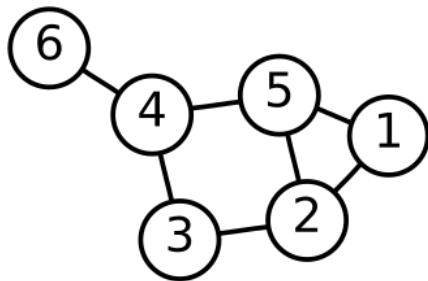


Figura 6.1. Grafo con seis vértices y siete aristas

### Definiciones fundamentales

#### Vértice

Los **vértices** constituyen uno de los dos elementos que forman un grafo. Como ocurre con el resto de las ramas de las matemáticas, a la Teoría de Grafos no le interesa saber qué son los vértices, sin embargo también los podemos llamar **nodos**, que se representan como “**puntos**” en la gráfica.

Diferentes situaciones en las que pueden identificarse objetos y relaciones que satisfagan la definición de grafo pueden verse como grafos y así aplicar la Teoría de Grafos en ellos.

#### Arista

Una **arista** es una **línea**, orientada o no, que une dos vértices. Corresponde a una **relación** entre dos vértices de un grafo. En un grafo no dirigido, se trata de relaciones simétricas sin dirección, mientras que en un grafo dirigido son relaciones direccionales, también conocidas como **arcos**.

#### Grafo

Un **grafo** es una pareja de conjuntos  $G = (V, A)$ , donde  $V$  es el conjunto de **vértices** y  $A$  es el conjunto de **aristas**; este último es un conjunto de pares de la forma  $(u, v)$  tal que  $u, v \in V$ . También se pueden usar otras notaciones equivalentes para indicar una

**arista:**  $(a, b) = ab = a \rightarrow b$ ; la última forma se aplica cuando el grafo es dirigido; las otras dos pueden usarse en grafos no dirigidos, aunque el contexto también puede indicar lo contrario.

En teoría de grafos, sólo queda lo esencial del dibujo: la forma de las aristas no son relevantes, sólo importa a qué vértices están unidas. La posición de los vértices tampoco importa, y se puede variar para obtener un dibujo más claro.

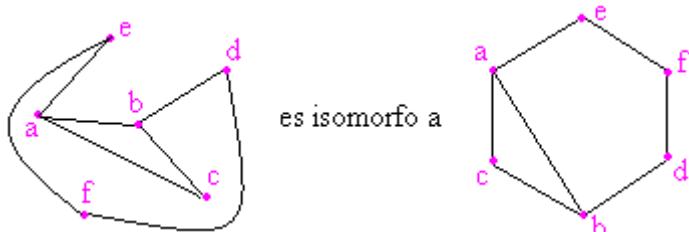


Figura 6.2. Grafos isomorfos<sup>26</sup>.  $V = \{ a, b, c, d, e, f \}$ , y  $A = \{ ab, ac, ae, bc, bd, df, ef \}$ .

Muchas situaciones pueden ser modeladas con un grafo: redes de uso cotidiano, una red de carreteras que conecta ciudades, una red eléctrica o la red de gas de una ciudad, el control de flujo de un programa (los vértices pueden ser sentencias de control y los arcos la transferencia del flujo), etc.

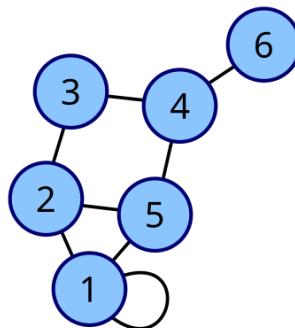


Figura 6.3. Grafo no dirigido

## Subgrafo

Un **subgrafo** de un grafo  $G$  es un grafo cuyos conjuntos de vértices y aristas son subconjuntos de los de  $G$ . Se dice que un grafo  $G$  contiene a otro grafo  $H$  si algún subgrafo de  $G$  es  $H$  o es isomorfo a  $H$  (dependiendo de las necesidades de la situación).

El **subgrafo inducido** de  $G$  es un subgrafo  $G'$  de  $G$  tal que contiene todas las aristas adyacentes al subconjunto de vértices de  $G$ .

---

<sup>26</sup> "Isomorfo" se refiere a objetos que tienen la misma forma o estructura, incluso si están compuestos de diferentes materiales o tienen propiedades distintas. En varios contextos, el concepto de "isomorfismo" describe una relación de equivalencia estructural.

### **Definición**

Sea  $G = (V, A)$ .  $G' = (V', A')$  se dice subgrafo de  $G$  si:

1.  $V' \subseteq V$
2.  $A' \subseteq A$
3.  $(V', A')$  es un grafo

Si  $G' = (V', A')$  es subgrafo de  $G$ , para todo  $v \in G$  se cumple  $\text{gr}(G', v) \leq \text{gr}(G, v)$

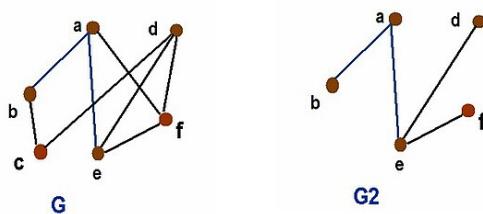


Figura 6.4.  $G_2$  es un subgrafo de  $G$ .

### **Grafo dirigido**

Un **grafo dirigido**  $G$  consiste en un conjunto de vértices  $V$  y un conjunto de arcos dirigidos  $A$ . Un arco o arista es un **par ordenado** de vértices  $(v, w)$ ;  $v$  es la *cola* y  $w$  es la *cabeza* de la arista.



Figura 6.5. Representación gráfica del arco  $(v, w) = v \rightarrow w$ .

En la figura puede verse como la “cola” está en el vértice  $v$ , mientras que la punta de la flecha indica la “cabeza” en el vértice  $w$  del par ordenado. Se dice que el arco  $v \rightarrow w$  va de  $v$  a  $w$ , y que  $w$  es adyacente a  $v$ .

Los vértices y los arcos pueden tener una **etiqueta**, la cual puede representar un nombre o un valor. En la figura 6.5 los nodos están etiquetados como  $v$  y  $w$  respectivamente.

### **Ejemplo 6.1**

Grafo dirigido con 6 vértices (nodos) y 8 aristas (arcos). Cada vértice y cada arista está etiquetada

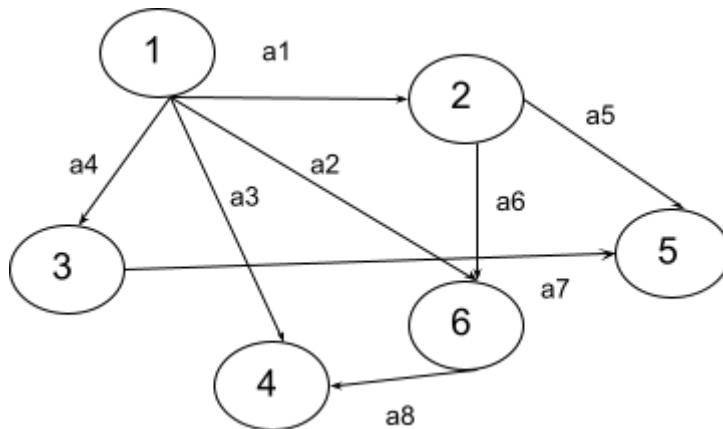


Figura 6.6. Grafo dirigido con 6 vértices (nodos) y 8 aristas (arcos)

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\} = \{(1, 2), (1, 6), (1, 4), (1, 3), (2, 5), (2, 6), (3, 5), (6, 4)\}$$

Observe cómo el orden de las parejas ordenadas en el conjunto A de las aristas especifican su dirección.

### Camino en un grafo

Un **camino en un grafo** es una secuencia de vértices  $v_1, v_2, \dots, v_n$  tal que  $v_1 \rightarrow v_2, v_2 \rightarrow v_3, \dots, v_{n-1} \rightarrow v_n$  son arcos. Este camino va del vértice  $v_1$  al vértice  $v_n$ , pasa por los vértices  $v_2, v_3, \dots, v_{n-1}$  y termina en el vértice  $v_n$ .

### Longitud de un camino

La **longitud de un camino** es el número de arcos en ese camino. Como consecuencia, se tiene que la longitud de un camino  $v \rightarrow v$  es 0.

### Camino simple

Un camino es simple si todos sus vértices, excepto quizá el primero y el último son distintos.

### Ciclo simple

Es un camino simple de longitud por lo menos uno, que empieza y termina en el mismo vértice.

### Ejemplo 6.2

En el grafo de la figura 6.6 la secuencia (camino) 1, 2, 6, 4 es tal que  $1 \rightarrow 2, 2 \rightarrow 6, 6 \rightarrow 4$  son arcos y tiene una longitud de camino igual a 3; la secuencia 1, 3, 5 con los

arcos  $1 \rightarrow 3$ ,  $3 \rightarrow 5$ , tiene una longitud igual a 2. Se observa que no hay ciclos simples en este grafo.

## Grafos no dirigidos

Un grafo no dirigido (muchas veces descrito simplemente como grafo)  $G = (V, A)$  consta de un conjunto finito de vértices  $V$  y de un conjunto de aristas  $A$ . Se diferencia de un grafo dirigido en que cada arista en  $A$  es un par no ordenado de vértices.

## Estructuras de datos en la representación de grafos

Existen diferentes formas de almacenar grafos en un computador. La estructura de datos depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

### Estructura de lista

- **Lista de incidencia:** las aristas son representadas con un vector de pares (ordenados, si el grafo es dirigido), donde cada par representa una de las aristas.
- **Lista de adyacencia:** cada vértice tiene una lista de vértices los cuales son adyacentes a él. Esto causa redundancia en un grafo no dirigido (ya que  $A$  existe en la lista de adyacencia de  $B$  y viceversa), pero las búsquedas son más rápidas.

En esta estructura de datos la idea es asociar a cada vértice  $i$  del grafo una lista que contenga todos aquellos vértices  $j$  que sean adyacentes a él. De esta forma sólo reservará memoria para los arcos adyacentes a  $i$  y no para todos los posibles arcos que pudieran tener como origen  $i$ . El grafo, por tanto, se representa por medio de un vector de  $n$  componentes ( $|V| = n$ ) donde cada componente va a ser una lista de adyacencia correspondiente a cada uno de los vértices del grafo. Cada elemento de la lista consta de un campo indicando el vértice adyacente. En caso de que el grafo sea etiquetado, habrá que añadir un segundo campo para mostrar el valor de la etiqueta.

### Estructuras matriciales

- **Matriz de incidencia:** el grafo está representado por una matriz de  $A$  (aristas) por  $V$  (vértices), donde [arista, vértice] contiene la información de la arista (1 - conectado, 0 - no conectado). También puede representarse con una matriz de  $V$  (vértices) por  $A$  (aristas), donde [vértice, arista] contiene la información de la arista (1 - conectado, 0 - no conectado).

- **Matriz de adyacencia:** el grafo está representado por una matriz cuadrada  $M$  de tamaño  $n^2$ , donde  $n$  es el número de vértices. Si hay una arista entre un vértice  $x$  y un vértice  $y$ , entonces el elemento  $m_{x,y}$  es 1, de lo contrario, es 0.

## Operaciones básicas con grafos en computación

Las operaciones básicas con grafos en computación incluyen la comprobación de la existencia de una arista entre dos vértices, la obtención de vértices adyacentes, la inserción y eliminación de vértices y aristas, y la determinación de la adyacencia de un vértice a otro. Aquí un listado de las más comunes:

- Inserción de un vértice: agrega un nuevo vértice al grafo.
- Recorrido del grafo: permite visitar todos los vértices y aristas del grafo, normalmente en un orden determinado.
- Determinación de la adyacencia: verifica si un vértice es adyacente a otro.
- Búsqueda de un vértice: permite encontrar un vértice específico en el grafo.
- Comprobación de la existencia de una arista: determina si existe una conexión (arista) entre dos vértices específicos.
- Obtención de vértices adyacentes: devuelve la lista de vértices a los que un vértice dado está conectado.
- Eliminación de un vértice: borra un vértice y todas las aristas que lo conectan a otros vértices.
- Inserción de una arista: añade una nueva conexión entre dos vértices existentes.
- Eliminación de una arista: elimina la conexión entre dos vértices.

### Ejemplo 6.3

La siguiente figura muestra un grafo no dirigido con 6 vértices y 8 arcos. Encontrar y representar:

1. Matriz de incidencia
2. Matriz de adyacencia
3. Lista de incidencia
4. Lista de adyacencia

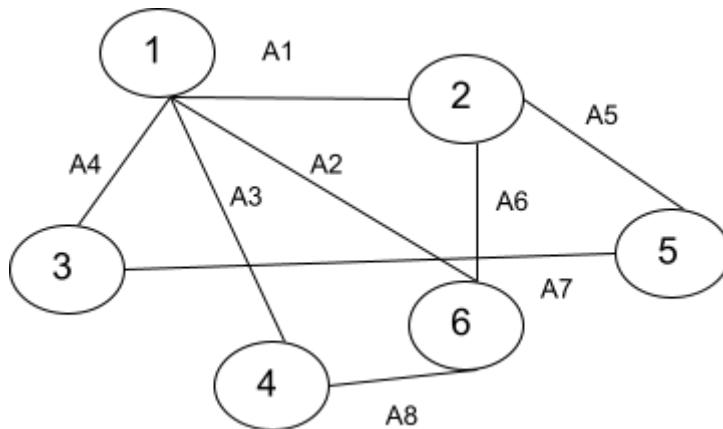


Figura. Grafo no dirigido con 6 vértices (nodos) y 8 aristas (arcos)

Matriz de incidencia

Esta matriz está conformada de  $m$  vértices y  $n$  aristas. En un vector columna almacenamos los vértices del grafo y en un vector fila almacenamos sus aristas. Si  $\text{matriz}[\text{vértice}, \text{arista}] = 1$  es por que el vértice está conectado con la arista, y es igual a cero en caso contrario.

Vertice / Arista	A1	A2	A3	A4	A5	A6	A7	A8
Vertice	1	0	1	1	0	0	0	0
Arista	0	1	0	0	1	1	0	0
Vertice	0	1	0	1	0	0	1	0
Arista	1	0	0	0	0	0	0	1
Vertice	0	0	1	0	0	1	0	0
Arista	1	0	0	0	1	0	1	0
Vertice	0	0	0	1	0	0	0	1
Arista	0	1	0	0	0	1	0	1

Matriz de adyacencia

Esta matriz está conformada de  $m$  filas por  $m$  columnas, donde  $m$  es el número de vértices del grafo, esto es, es una matriz cuadrada. Si  $\text{matriz}[i, j] = 1$  es por que el vértice  $i$  está conectado con el vértice  $j$ , y es igual a cero en caso contrario. Usamos el vector de vértices implementado para la matriz de incidencia y lo utilizamos tanto como vector fila como vector columna.

Vértice	1	2	3	4	5	6
1	0	1	1	1	0	1
2	1	0	0	0	1	1
3	1	0	0	0	1	0
4	1	0	0	0	0	1
5	0	1	0	0	1	0
6	0	0	1	0	0	1

5	0	1	1	0	0	0
6	1	1	0	1	0	0

Lista de incidencia

Creamos un vector para almacenar las aristas del grafo, donde cada arista es una pareja (ordenada si el grafo es dirigido)  $(v_i, v_j)$ ,  $0 \leq i, j < n$  ( $n$  total de vértices),  $v_i, v_j \in V$ .

(1, 2)	(1, 6)	(1, 4)	(1, 3)	(2, 5)	(2, 6)	(5, 3)	(6, 4)
--------	--------	--------	--------	--------	--------	--------	--------

Lista de adyacencia

Creamos un vector (o lista) para almacenar los vértices y de cada uno de éstos se desprende una lista con los vértices adyacentes a éstos.

1	2	3	4	5	6
2	1	1	1	2	1
3	5	5	6	3	2
4	6				4
6					

**Ejemplo 6.4**

Operaciones sobre grafos. Realizar las principales operaciones sobre grafos dirigidos y no dirigidos

# Fuentes y referencias adicionales

## Textos impresos

JOYANES AGUILAR, LUIS. Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos. McGraw-Hill/Interamericana de España. Madrid, 2004.

JOYANES AGUILAR, LUIS. Fundamentos de Programación, Algoritmos y Estructuras de Datos. Segunda edición. McGraw-Hill/Interamericana de España. Madrid, 1996.

CAIRÓ, OSVALDO; GUARDATI BUEMO, SILVIA. Estructuras de Datos. McGraw-Hill. México DF, 1996.

AHO, ALFRED V., HOPCROFT, JOHN E., HULLMAN, JEFFREY D. Estructuras de datos y algoritmos. Addison-Wesley Iberoamericana. Wilmington, Delaware E.U.A., 1988.

FLÓREZ R., ROBERTO. Algoritmos y Estructuras de Datos. Universidad de Antioquia. Medellín, 1996.

BRASSARD, G., BRATLEY, T. Fundamentos de Algoritmia. Prentice Hall. Madrid, 1996.

VILLALOBOS J., CASALLAS R. Fundamentos de programación: Aprendizaje activo basado en casos. Bogotá, Pearson - Prentice Hall. 2006

BOTERO R., CASTRO C., MAYA J., TABORDA G. y VALENCIA M.. Lógica y Programación orientada a objetos: un enfoque basado en problemas. CITIA, proyecto SISMOO, Tecnológico de Antioquia. Medellín, 2009.

## Páginas web

Árboles Balanceados AVL. Universidad Don Bosco, Facultad de Ingeniería, Escuela de Computación, Asignatura Programación con Estructuras de Datos  
En Internet (en pdf)

[Tema: “Árboles Balanceados AVL”](#)

Teoría de grafos

[Teoría de grafos](#)