www.uniremington.edu.co

UNIREMINGTON®
CORPORACIÓN UNIVERSITARIA REMINGTON
RES. 2661 MEN JUNIO 21 DE 1996

ALGORITMOS II

INGENIERIA DE SISTEMAS

FACULTAD DE CIENCIAS BÁSICAS E INGENIERÍA

Vicerrectoría de Educación a Distancia y virtual

2016

AUTOR

José Antonio Polo
Ingeniero de sistemas de la Universidad de Antioquia Especialista en finanzas de la Corporación Universitaria Remington y estudiante de maestría en educación de la universidad Cesun (Tijuana México). Participación del tercer Congreso Colombiano de Computación – 3CCC de la universidad EAFIT Participación del primer simposio en Inteligencia Artificial de la Corporación Universitaria Remington Participación del IV Congreso Internacional de Software Libre GNU/Linux, Universidad de Manizales Participación del 5º Congreso Nacional de Redes y Telemática, Redes de Servicios Móviles Integrados, Centro de Construcción de Conocimiento Evento CCC Docente de cátedra del politécnico Jaime Isaza Cadavid Docente de cátedra del Tecnológico de Antioquia Participación del proyecto de la articulación de la media técnica del Tecnológico de Antioquia Docente de la Corporación Universitaria Remington.
Barra5111@yahoo.es

Nota: el autor certificó (de manera verbal o escrita) No haber incurrido en fraude científico, plagio o vicios de autoría; en caso contrario eximió de toda responsabilidad a la Corporación Universitaria Remington, y se declaró como el único responsable.

RESPONSABLES

Jorge Mauricio Sepúlveda Castaño
Decano de la Facultad de Ciencias Básicas e Ingeniería
jsepulveda@uniremington.edu.co

Eduardo Alfredo Castillo Builes
Vicerrector modalidad distancia y virtual
ecastillo@uniremington.edu.co

Francisco Javier Álvarez Gómez
Coordinador CUR-Virtual
falvarez@uniremington.edu.co

GRUPO DE APOYO

Personal de la Unidad CUR-Virtual
EDICIÓN Y MONTAJE
Primera versión. Febrero de 2011.
Segunda versión. Marzo de 2012
Tercera versión. noviembre de 2015

TABLA DE CONTENIDO

Pág.

1 MAPA DE LA ASIGNATURA

ALGORITMOS II

PROPÓSITO GENERAL DEL MÓDULO

Que el estudiante pueda identificar las diferentes estructuras de almacenamiento y aplicarlas de forma óptima en la solución de problemas.

OBJETIVO GENERAL

Brindar al estudiante nuevas herramientas de programación para para el almacenamiento de datos tanto a nivel interno con externo, para el desarrollo de algoritmos aplicables en cualquier disciplina, que permitan la generación de una capacidad analítica y creativa en la solución e implementación de problemas propuestos.

OBJETIVOS ESPECÍFICOS

* Utilizar las estructuras básicas de programación para la construcción de soluciones de problemas apoyadas en el computador, identificado las estructuras de almacenamiento tanto a nivel interno como a nivel externo y el tipo de información que puede guardar cada una de ellas, como son los arreglos de registros y los archivos, reconociendo la utilidad de manejar la eficiencia a nivel de algoritmos, para dar soluciones óptimas.
* Manejar los conceptos de manejo dinámico de memoria para optimizar el recurso de memoria a nivel de programación, representando problemas a nivel de listas simplemente ligadas, listas doblemente ligadas y listas doblemente ligadas circulares.
* Reconocer y dar solución a problemas a partir de hileras de caracteres a nivel de programación.
* Identificar las estructuras de almacenamiento de información como son las pilas y las colas y que estas se pueden representar tanto a nivel estático (vectores), y a nivel dinámico (listas ligadas) y los tipos de operaciones fundamentales que se pueden realizar con cada una de ellas.

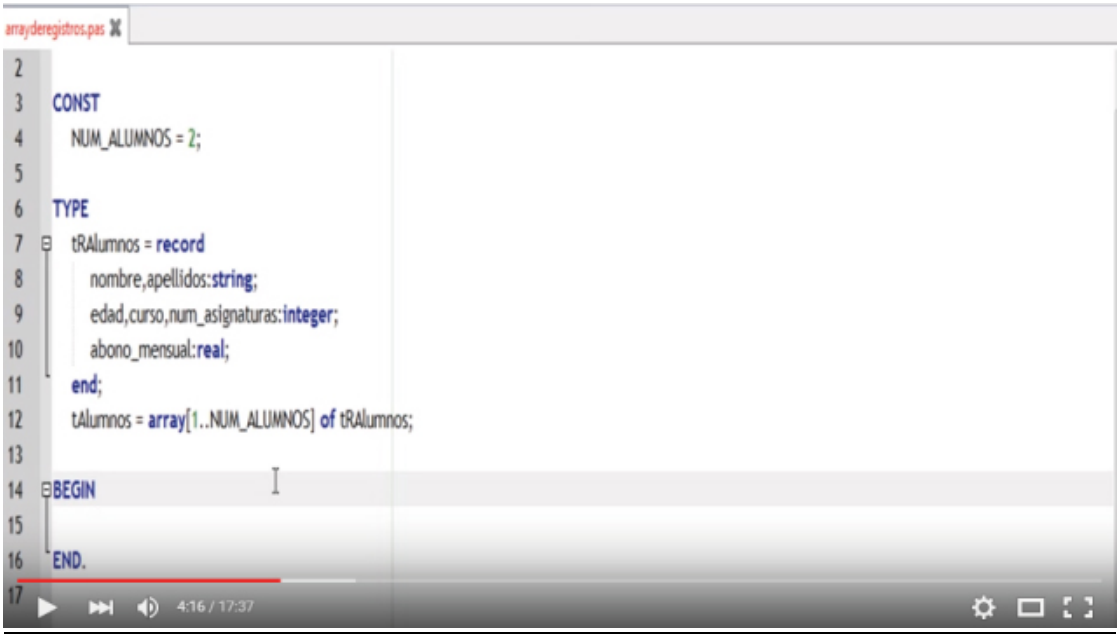
UNIDAD 1
Arreglos De Registros Y Archivos

UNIDAD 2
Manejo Dinamico De Memoria Y Punteros

UNIDAD 3
Manejo De Cadenas De Caracteres

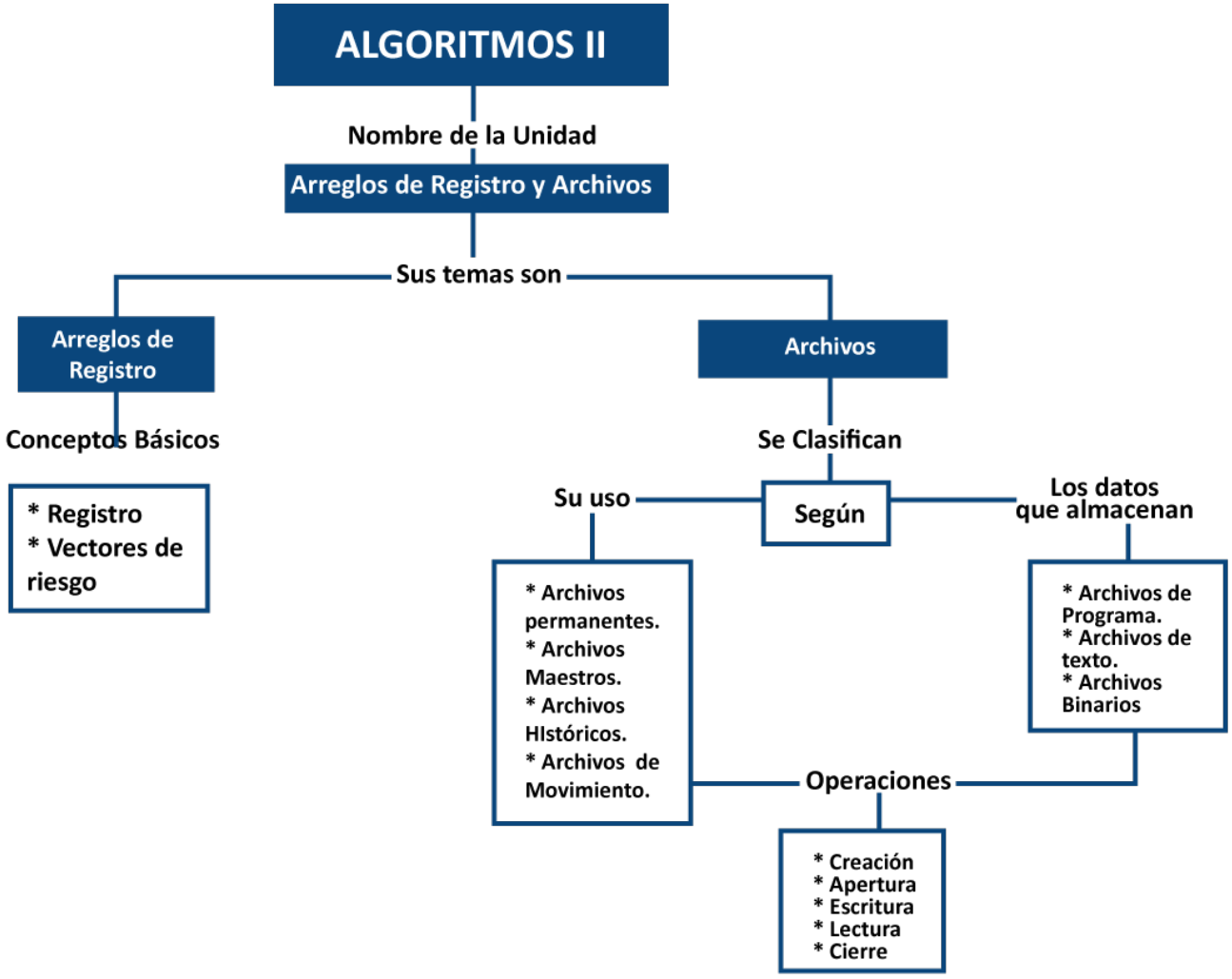
UNIDAD 4
Pilas Y Colas

2 UNIDAD 1 ARREGLOS DE REGISTROS Y ARCHIVOS



Pascal - 029 - Array de registros (Lista estática): **Enlace**

2.1 RELACIÓN DE CONCEPTOS



Arreglos de registros: Estructuras de almacenamiento de información a nivel de memoria interna, y que permite realizar diferentes operaciones a nivel de búsqueda, inserción, borrado entre otras.

Registro: Es una estructura de datos compuesta. Se puede decir que un registro es un conjunto de campos variables relacionados que, en general, puede pertenecer a tipos de datos diferentes, llamados componentes del tipo registro, donde todas las componentes pueden manipularse bajo un solo nombre de variable.

Vectores de registro: Otra importancia que tienen los arreglos de registros es que cada uno de sus elementos está formado por varios campos pertenecientes a diferentes tipos de datos, a diferencia de los otros tipos de arreglos en los que sus elementos son de un solo tipo; aunque cabe anotar que, por ejemplo, un vector puede ser de tipo registró.

Archivos: Se definen como estructuras de almacenamiento a nivel externo, permitiendo realizar diferentes operaciones sobre ellos.

Archivos permanentes: Son archivos rígidos, ya que la información que almacenan cambia muy poco.

Archivos maestros: La información almacenada en esta clase de archivos maneja el estado o situación de los registros de una determinada base de datos, y su actualización o los cambios de los datos de los registros se hacen periódicamente.

Archivos históricos: Son archivos maestros que ya no se utilizan y cuya función es solo de consulta para quienes desean obtener información sobre el pasado.

Archivos de movimiento: Son temporales y su función principal es captar información para actualizar los archivos maestros.

Archivos de programa: Son los programas fuente que se escriben en un determinado lenguaje de programación, donde cada instrucción es un registro.

Archivos de texto o ASCII (American Standard Code for Information Interchange): Estos almacenan cualquier carácter del código ASCII: palabras, frases, párrafos, y se suelen crear y mantener mediante programas que procesan o editan texto.

Archivos binarios: Estos archivos almacenan información en un lenguaje descifrable solo por el computador. La información que puedan almacenar es diversa: colores, sonidos, imágenes, órdenes o datos discretos, llamados archivos de datos.

Creación del archivo: Permite definir una variable dentro del programa que represente el archivo que se quiere manipular.

Apertura de un archivo: Permite relacionar una variable tipo archivo con el nombre que el archivo tendrá en el medio de almacenamiento, llamado nombre externo. Este es el nombre con el cual se guarda el archivo en el medio de almacenamiento, y su extensión depende de la clase de información que se va a guardar.

Escritura en el archivo: Permite almacenar la información i en el archivo que se está manipulando.

Lectura de un archivo: Permite extraer la información del archivo y almacenarla en memoria principal. La información que se transporta a la memoria puede ser cualquier tipo de variable u objetos almacenados en el archivo.

Cierre del archivo: Permite proteger el área donde reside el archivo y actualiza el directorio del medio de almacenamiento, reflejando el nuevo estado del archivo. Adicionalmente, destruye las conexiones físicas y lógicas que se construyeron cuando se abrió el archivo.

2.2 TEMA 1 DEFINICIÓN

Se define los arreglos de registros como estructuras de almacenamiento de información a nivel de memoria interna, y que permite realizar diferentes operaciones a nivel de búsqueda, inserción, borrado entre otras. Los archivos se definen como estructuras de almacenamiento a nivel externo, permitiendo realizar diferentes operaciones sobre ellos

Un registro es una estructura de datos compuesta. Se puede decir que un registro es un conjunto de campos variables relacionados que, en general, puede pertenecer a tipos de datos diferentes, llamados componentes del tipo registro, donde todas las componentes pueden manipularse bajo un solo nombre de variable. Por ejemplo, si se tiene un registro de datos compuestos por los campos: **cédula, nombre, deducción y salario**, podemos representarlo de la siguiente forma:

32506321	SANDRA FONEGRA	40000	630000

Como puede notarse, dentro del registro solo hay datos (información); es el programa quien coloca nombres a cada dato para poderlos almacenar en la memoria de la computadora.

Si la información del empleado se fuera a tratar en forma individual y los nombres de variables seleccionados para cada campo fuera: **cc, nom, deduc y sal;** al ejecutar la instrucción:

LEA: **cc, nom, deduc, sal**

Ocurrirá en memoria lo siguiente:

cc						**deduc**			
32506321						40000			
									sal
			nom						630000
			SANDRA FONEGRA						
				MEMORIA					

La información de la empleada está dispersa en la memoria. La idea de usar registro es agrupar toda la información en una misma área de memoria bajo un solo nombre. Si se toma la determinación de que **emp1** es una variable de ***tipo registro***, o sea, almacenar en el área asignada toda la información de un empleado, gráficamente se puede mirar de la siguiente manera:

emp1	MEMORIA
<Cédula>	
<Nombre>	
<Deducción>	
<Salario>	

Para que **emp1** sea tomada como un variable tipo registró, se debe definir como tal; es decir, se define a la variable como ***registro*** y se definen los demás campos que habrá en dicho registro. Para nuestro ejemplo, la variable registró quedaría así:

1. **CLASE** Ejemplo
2. METODO PRINCIPAL ()
3. empleado (Registro):
4. cc, nom (Alfanuméricos)
5. deduc, sal (Reales)
6. VARIABLES: emp1, emp2 (Empleado)
7. Ejemplo. Llenar (emp1)
8. end(Método)
9. end(Clase)

En el anterior ejemplo se ha definido un dato de tipo registro llamado **empleado** y además se define una variable que es de tipo **empleado**; esto quiere decir que cualquier variable que sea de tipo **empleado** puede almacenar los valores de las componentes del registro, es decir, que en **emp1** se almacenan: **cc, nom, deduc y sal.**

En la estructura anterior se define a **emp1** y **emp2** como variables tipo registro, por lo tanto, pueden almacenar todos los valores de los campos que conforman al tipo de dato empleado, y su representación interna seria:

emp1		emp2
<Cédula>		<Cédula>
<Nombre>		<Nombre>
<Deducción>		<Deducción>
<Salario>		<Salario>
	MEMORIA	

Las variables tipo registro **emp1** y **emp2** son variables compuestas; para referenciar a cada una de sus componentes hay que clasificarlas, o sea, decir en forma explícita a cuál de las componentes del registro nos vamos a referir. Dicha clasificación se hace así: **variable_tipo_regristro. componente**.

Veamos un método de cómo llenaríamos la información de un empleado, después de haber definido la variable de tipo registro ***empleado*** y a la variable de tipo ***empleado***:

1. PUBLICO ESTATICO VOID Llenar (Reg)
2. IMPRIMA ("Ingrese la cédula")
3. LEA (Reg.cc)
4. IMPRIMA ("Ingrese el nombre")
5. LEA (Reg. Nom)
6. IMPRIMA ("Ingrese la deducción")
7. LEA (Reg. Deduc)
8. IMPRIMA ("Ingrese el salario")
9. LEA (Reg. Sal)
10. end(método)

El parámetro **emp1** es un parámetro por referencia.

En la instrucción 3 se lee la cédula que se almacenará en el registro. Igualmente en las instrucciones 5, 7, y 9 donde se almacenará el **nombre, deducción y salario del empleado**, respectivamente.

Después de ejecutar el método anterior y llenarlo con los datos que teníamos anteriormente, nuestra memoria estaría de la siguiente forma:

emp1		emp2
32506321		<Cédula>
SANDRA FONEGRA		<Nombre>
40000		<Deducción>
630000		<Salario>
	MEMORIA	

Si hay necesidad de mover la información de una variable tipo registro a otra, se puede hacer componente a componente, o moviendo toda la información al mismo tiempo. Por ejemplo:

emp2.cc = emp1.cc
emp2.nom = emp1.nom
emp2.deduc = emp1.deduc
emp2.sal = emp1.sal
Lo cual sería lo mismo que tener: emp2 = emp1.

Un **arreglo de registro** es una estructura de datos de gran utilidad e importancia en la programación de aplicaciones, ya que muchas veces es deseable, desde el punto de vista de la lógica del programa en particular, mantener disponible en memoria principal una serie de registro con información del mismo tipo para su proceso; dichos registros no estarán dispersos; harán parte de un arreglo, lo que facilita su manejo. Un arreglo de registros es muy parecido a un archivo de datos, se diferencian en que los arreglos de registros residen en memoria principal y los archivos en memoria auxiliar, por lo tanto, estos últimos tienen existencia permanente.

2.3 TEMA 2 OPERACIONES CON ARREGLOS DE REGISTROS

Los **arreglos de registros** permiten manejar información en memoria interna, lo cual **permite realizar las operaciones de: búsqueda** que permite verificar si un dato se encuentra en la estructura de almacenamiento, **inserción** que permite ingresar un nuevo dato en el arreglo de registro, **borrado** que permite borrar un dato del arreglo de registro entre otras operaciones que se pueden realizar con estas estructuras.

Otra importancia que tienen los arreglos de registros es que cada uno de sus elementos está formado por varios campos pertenecientes a diferentes tipos de datos, a diferencia de los otros tipos de arreglos en los que sus elementos son de un solo tipo; aunque cabe anotar que, por ejemplo, **un vector puede ser de tipo registró.** Veamos un ejemplo en donde almacenamos los datos de un registro en un vector y luego imprimimos dicho vector:

1. **CLASE** RegVector
2. METODO PRINCIPAL()
3. R (REGISTRO):
4. cc, nom (Alfanuméricos)
5. deduc, sal (Reales)
6. VARIABLES: vec[], emp (R)
7. i, n (Entero)
8. IMPRIMA (Ingrese el número de registros")
9. LEA (n)
10. para (i=1, n, 1)
11. RegVector.Llenar (emp)
12. vec[i] = emp
13. endpara
14. para (i= 1, n, 1)
15. IMPRIMA (vec[i].cc)
16. IMPRIMA (vec[i].nom)
17. IMPRIMA (vec[i]. deduc)
18. IMPRIMA (vec[i]. sal)
19. endpara
20. end(Método)
21. end(Clase)

En la instrucción 3 definimos a **R** de tipo **registro** con los campos: **cc, nom, deduc y sal**; y en la instrucción 6 definimos nuestras variables, en donde utilizamos 2 variables de tipo **R**: vec [] y emp.

En la instrucción 9 leemos a **n**, para conocer la cantidad de registros que almacenaremos y, en la instrucción 10, definimos nuestro ciclo desde 1 hasta **n** para llenar el vector con los registros.

En la instrucción 11 llamamos al método Llenar para leer los datos que guardaremos en el vector. Enviamos como parámetro a la variable **emp**.

En la instrucción 12 llevamos al vector en posición **i** lo que se guardó en **emp**.

EN LAS INSTRUCCIONES 15 A 18 IMPRIMIMOS EL CONTENIDO DE LOS VECTORES, ES DECIR, IMPRIMIMOS LOS REGISTROS QUE HEMOS ALMACENADO.

2.3.1 EJERCICIOS DE ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Puedes crear una estructura tipo vector de registro, con los campos correspondientes al enunciado en el ejercicio, o puede suponer que la estructura ya está creada y puedes trabajar sobre ella, realizando las operaciones que te pide cada uno de los ejercicios mencionados.

1. Una compañía distribuye N productos a distintos comercios de la ciudad. Para ello almacena en un arreglo toda la información relacionada a su mercancía:

   - Clave
   - Descripción
   - Existencia
   - Minino a mantener de existencia
   - Precio unitario

   Escriba un programa que pueda llevar a cabo las siguientes operaciones:

   a) Venta de un producto: se deben actualizar los campos que correspondan, y verificar que la nueva existencia no esté por debajo del mínimo. (Datos: clave, cantidad vendida)

   b) Reabastecimiento de un producto: se deben actualizar los campos que correspondan. (Datos: clave, cantidad comprada.)

   c) Actualizar el precio de un producto. (Datos: clave, porcentaje de aumento.)

   d) Informar sobre un producto: se deben proporcionar todos los datos relacionados a un producto. (Dato: clave.)

2. Una inmobiliaria tiene información sobre departamentos en renta. De cada departamento se conoce:

   - Clave: es un entero que identifica al inmueble.

- Extensión: superficie del departamento, en metros cuadrados.
- Ubicación: (excelente, buena, regular, mala)
- Precio: es un real.
- Disponible: VERDADERO si está disponible para la renta y FALSO si ya está rentado.

Diariamente acuden muchos clientes a la inmobiliaria solicitando información.

Escriba un programa capaz de realizar las siguientes operaciones sobre la información disponible:

a) Liste los datos de los departamentos disponibles que tengan un precio inferior o igual a un cierto calor P.

b) Liste los datos de los departamentos que tengan una superficie mayor o igual a un cierto valor E y una ubicación excelente.

c) Liste el monto de la renta de todos los departamentos alquilados.

d) Llega un cliente solicitando rentar un departamento. Si existe un departamento con una superficie mayor o igual a la deseada, con un precio y una ubicación que se ajustan a las necesidades del cliente, el departamento se renta. Actualizar los datos que correspondan.

e) Se vence un contrato, si no se renueva, actualizar los datos que correspondan.

Se ha decidió aumentar las rentas en X%. Actualizar los precios de las rentas de los departamentos no alquilados.

2.4 TEMA 3 TIPOS DE ARCHIVOS

Existen diferentes tipos de archivos como son: **archivos de entrada, archivos de salida, archivos de situaciones, archivos históricos, archivos secuenciales, archivos de movimientos, archivos de índice, archivos maestros entre otros.**

2.4.1 ARCHIVOS

En la mayoría de aplicaciones por computador se requiere que la información que se almacene permanezca para poder usarla en proceso posteriores. Para ello necesitamos otros medios de almacenamiento como los archivos que se guardan en la memoria auxiliar **(USB, discos, DVD, memoria flash, etc.)** a través de dispositivos que no necesitan estar conectados constantemente al computador; de este modo solo se recurre a ellos cuando se necesita usar la información que tienen almacenada. El hecho de que estos aparatos residan fuera de la memoria principal y guarden la información de forma permanente permite que los datos sirvan para diferentes ejecuciones del mismo programa y sean usados por programas diferentes. Desde luego, también se puede guardar mucha más información que en la memoria principal.

Un archivo es, entonces, una estructura lógica donde se almacena en forma permanente información con las siguientes ventajas: almacenamiento de grandes volúmenes, persistencia de esta en medios diferentes a la memoria principal del computador y posibilidad de que diferentes programas puedan acceder a ella.

2.4.2 CLASIFICACIÓN DE LOS ARCHIVOS SEGÚN SU USO

CLASIFICACIÓN SEGÚN SU USO	CARACTERÍSTICAS
ARCHIVOS PERMANENTES	Son **archivos rígidos**, ya que la información que almacenan cambia muy poco. **Se usan para extraer información** que se utiliza en otros **archivos o procesos**: por ejemplo, el archivo que contiene toda la información sobre los salarios de los empleados de una empresa en años posteriores.
ARCHIVOS MAESTROS	**La información almacenada en esta clase de archivos maneja el estado o situación de los registros de una determinada base de datos, y su actualización o los cambios de los datos de los registros se hacen periódicamente**. Por ejemplo, un archivo que almacene el inventario de los bienes de una empresa.
ARCHIVOS HISTÓRICOS	Son **archivos maestros** que ya no se utilizan y cuya **función es solo de consulta** para quienes desean obtener información sobre el pasado. Por ejemplo, la información contenida en las hojas de vida académica de los estudiantes que terminaron su carrera en la década pasada.
ARCHIVOS DE MOVIMIENTO	Son **temporales y su función principal es captar información para actualizar los archivos maestros.** Sus registros muestran las **transacciones o movimientos** que se producen durante un determinado periodo. Cuando el **archivo maestro** se actualiza, usando los **archivos de movimiento**, estos últimos pierden su validez y se pueden destruir. Por ejemplo: los cambios de aumento de salarios, deducciones y sobresueldos producidos durante un determinado periodo.

2.4.3 CLASIFICACIÓN DE LOS ARCHIVOS SEGÚN LOS DATOS QUE ALMACENAN

CLASIFICACIÓN DE LOS ARCHIVOS SEGÚN LOS DATOS QUE ALMACENAN	CARACTERÍSTICAS

ARCHIVOS DE PROGRAMA	Son los **programas fuente** que se escriben en un **determinado lenguaje de programación**, donde cada instrucción es un registro. **Estos archivos se pueden cargar del medio magnético a memoria, compilar, ejecutar, imprimir y volver a guardar** donde estaban o en otro medio de almacenamiento distinto.
ARCHIVOS DE TEXTO O ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE)	Estos **almacenan cualquier carácter del código ASCII: palabras, frases, párrafos,** y se suelen crear y mantener mediante programas que procesan o editan texto.
ARCHIVOS BINARIOS	Estos archivos **almacenan información en un lenguaje descifrable solo por el computador**. La información que puedan almacenar es diversa: ***colores, sonidos, imágenes, órdenes o datos discretos, llamados archivos de datos.***

Antes de hablar sobre los archivos de texto, definiremos algunas operaciones básicas que se deben tener en cuenta cuando se trabaja con archivos.

2.5 TEMA 4 OPERACIONES CON ARCHIVOS

Los archivos son estructuras de almacenamiento de datos a nivel externo, lo cual permite manejar información para procesos posteriores, donde se puede manejar las diferentes operaciones sobre su conjunto de datos, estas operaciones son: **insertar, ordenar, buscar, actualizar, borrar, sobre uno o más datos.**

2.5.1 OPERACIONES BÁSICAS CON ARCHIVOS

A continuación, se definen las **acciones básicas con archivos:**

- **Creación del archivo:** Permite definir una variable dentro del programa que represente el archivo que se quiere manipular. Esta variable es conocida como nombre interno del archivo y se define con la siguiente instrucción:

ARCHIVO a

- **Apertura de un archivo:** Permite relacionar una variable tipo archivo con el nombre que el archivo tendrá en el medio de almacenamiento, llamado nombre externo. Este es el nombre con el cual se guarda el archivo en

el medio de almacenamiento, y su extensión depende de la clase de información que se va a guardar. También es posible darle una ruta que indique donde quedara guardado el archivo; si no se especifica la ruta, el archivo se guarda dónde está el programa que lo crea. La sintaxis de apertura es:

ABRIR a ("nombreExterno.extensión")

Con esta instrucción se genera una conexión entre el nombre interno a y el nombre externo del archivo, donde a es la variable tipo archivo con la que se manejara el nombre externo del archivo en el programa, nombre-externo es el nombre que se le dará al archivo en el medio de almacenamiento y la extensión depende de la información que se quiere almacenar.

Esta instrucción crea la variable a como variable tipo archivo y hace las conexiones físicas y lógicas con el nombre externo. Al abrir el archivo la variable tipo archivo a toma el valor verdadero si el archivo se pudo abrir o falso si el archivo no se pudo abrir o no existe. Esto nos permite preguntar si el archivo se abrió o no. Al abrir un archivo hay que especificar el modo de apertura: de salida, de entrada, entrada y salida, adicionar más información, etcétera.

- **Escritura en el archivo:** Permite almacenar la información i en el archivo que se está manipulando. La instrucción es:

ESCRIBA a (i)

Esta instrucción pasa la información i que reside en memoria al archivo cuyo nombre externo está relacionado con a.

- **Lectura de un archivo:** Permite extraer la información del archivo y almacenarla en memoria principal. La información que se transporta a la memoria puede ser cualquier tipo de variable u objetos almacenados en el archivo. La sintaxis es:

LEA a (var)

La variable var puede representar cualquier tipo de dato: las más comunes son variables de texto, estructuras y objetos.

- **Cierre del archivo:** Permite proteger el área donde reside el archivo y actualiza el directorio del medio de almacenamiento, reflejando el nuevo estado del archivo. Adicionalmente, destruye las conexiones físicas y lógicas que se construyeron cuando se abrió el archivo. La sintaxis es:

CERRAR a

Cuando se ejecuta la instrucción se adiciona un registro después del último grabado, donde coloca una marca de fin de archivo (EOF – END OF FILE).

2.5.2 ARCHIVO DE TEXTO

También **se les conoce como archivos llanos o planos por el hecho de almacenar solo texto**, es decir que **almacenan caracteres, palabras, frases o párrafos** y no tienen formato: contienen solo texto. Los caracteres que se almacenan pueden ser cualquiera del conjunto de caracteres de **ASCII.** Cada línea de texto es un registro, y la variable tipo archivo asignada en la definición del archivo se usa para fluir información desde un medio de entrada hacia la memoria auxiliar donde este reside y para extraer información desde el archivo hacia un medio externo de salida. Un ejemplo de un registro (línea de texto) seria:

```
Este es un archivo de texto que almacena un flujo de caracteres.
```

2.5.3 EJERCICIO DE APRENDIZAJE

Elaborar una clase que genere un archivo te texto con los campos cedula y nombres, imprima el contenido del archivo y adicione más registros al final del archivo.

1. Identificación de datos, acciones y limitaciones

- Datos de entrada: Cédula, nombre.
- Datos de salida: La información almacenada en el archivo.
- Acciones: crear Archivo, imprimir Archivo y adicionarRegistroAlFinal.

2. Definición de clases

- Clase seleccionada: Archivo.
- Definición del constructor de la clase: Archivo (): no recibe argumentos.
- Diagramación de la clase:

ARCHIVO
+Archivo
+Crear Archivo
+Imprimir Archivo

+Adicionar Registro Al Final

- Explicación de los métodos: crear Archivo: se crea un archivo de texto que solicite al usuario la información que se desea almacenar. El archivo se guarda en la carpeta donde está el programa, pero se puede establecer una ruta para que se almacene en cualquier otro lugar del computador. Se le pregunta al usuario si el archivo esta creado o no. En caso que el archivo este creado, lo redirige a la opción ADICIONAR REGISTRO AL FINAL. El archivo tiene como nombre interno "archivo" y como nombre externo "nombres.txt". la información que se guarda por línea es:

CÉDULA: 1158377694	NOMBRE: MAXIMILIANO DEL TORO

El usuario decide cuanta información desea almacenar. Cuando este termine de entrar la información, se cierra el archivo.

imprimir Archivo: haciendo uso de un ciclo MIENTRAS, se recorre e imprime el archivo línea por línea, hasta cuando se llegue a la marca de fin de archivo (OEF).

adicionarRegistroAlFinal: se abre el archivo en modo escritura, se le solicitan los datos por ingresar al usuario, se escriben al final del archivo y posteriormente se cierra.

3. definición del método principal: se lee la opción seleccionada por el usuario y se invoca el método correspondiente.

Definición de variables:

archivo: objeto de la clase Archivo para invocar los métodos.

opción: opción del menú seleccionada por el usuario.

2.5.4 EJERCICIO DE APRENDIZAJE

1. CLASE Archivo
2. METODO PRINCIPAL ()
3. PUBLICO VOID Crear Archivo ()
4. ENTERO cedula
5. CARÁCTER nombre
6. CARÁCTER sw
7. ESCRIBA: "¡SI EL ARCHIVO ESTA CREADO SERA DESTRUIDO!
8. ¿ESTA CREADO? (S/N):"
9. LEA: sw

```
si (sw==" N") ENTONCES
CREAR archivo ("nombres.txt")
ESCRIBA: "DIGITE LA CEDULA:"
LEA: cedula
mientras (¡cedula! =0) HAGA
ESCRIBA: "DIGITE EL NOMBRE:"
LEA: nombre
ESCRIBA archivo ("CEDULA:", cedula, "\t", "NOMBRE:",
nombre)
ESCRIBA: "DIGITE NUEVA CEDULA O CERO PARA TERMINAR:"
LEA: cedula
end_mientras
CERRAR archivo
end_si
end_crearArchivo
PUBLICO VOID imprimirArchivo ()
CADENA línea
ABRIR archivo ("nombres.txt")
si (archivo) ENTONCES
mientras (archivo<>EOF) HAGA
LEA archivo(línea)
ESCRIBA: línea
end_mientras
CERRAR archivo
si_no
ESCRIBA: "EL ARCHIVO NOMBRES.TXT NO EXISTE O NO SE PUDO
ABRIR"
end_si
end_imprimirArchivo
PUBLICO VOID adicionarRegistroAlFinal ()
ENTERO cedula
CARÁCTER nombre
ABRIR archivo ("nombres.txt")
si (archivo)
ESCRIBA:" DIGITE LA CEDULA:"
LEA: cedula
ESCRIBA: "DIGITE EL NOMBRE:"
LEA: nombre
ESCRIBA archivo (cedula+" \t" + nombre)
CERRAR archivo
si_no
ESCRIBA: "EL ARCHIVO NOMBRES.TXT NO EXISTE O NO SE PUDO ABRIR"
```

```
52.     end_si
53.     end_adicionarRegistroAlFinal
54.     METODO PRINCIPAL ()
55.     ENTERO opción
56.     Archivo archivo
57.     HACER
58.     ESCRIBA: ”1: CREAR EL ARCHIVO”
59.     ESCRIBA: ”2: IMPRIMIR CONTENIDO DEL ARCHIVO”
60.     ESCRIBA: ”3: ADICIONAR REGISTRO AL FINAL”
61.     ESCRIBA: ”4: SALIR DEL MENU”
62.     ESCRIBA:” DIGITE OPCION:”
63.     LEA: opción
64.     CASOS DE (opción)
65.     CASO 1: archivo. crear Archivo ()
66.     CASO 2: archivo. ImprimirArchivo ()
67.     CASO 3: archivo. AdicionarRegistroAlFinal ()
68.     end_CASOS
69.     MIENTRAS (¡opción! =4)
70.     end_PRINCIPAL
71.     end(Método)
72.     end(Clase)
```

2.5.5 EJERCICIOS DE ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Para desarrollar estos ejercicios, debes de tener en cuenta, declarar una clase y dentro de esta clase declarar las variables o atributos que va a tener el archivo que se desea crear.

1. Elaborar una clase que forme un archivo de datos que contenga en cada registro: cédula, nombre, horas trabajadas, valor de hora trabajada y porcentaje de deducción. La clase debe producir un informe (archivo de texto) con la cedula, el nombre y el salario devengado (ordenado ascendentemente por el campo cédula).

2. Elaborar una clase que cree un archivo de datos con la siguiente información por registro: cedula, apellidos y nombre y cuatro notas con sus respectivos porcentajes. Luego de creado el archivo se pide mostrar un listado ordenado por apellido, insertar un nuevo estudiante, la calificación definida de cada estudiante y visualizar el contenido del archivo.

2.6 TEMA 5 EFICIENCIA DE ALGORITMOS

Cuando se trabaja con procesos en línea y los algoritmos son utilizados con un grado alto de frecuencia, **se deben manejar algoritmos muy eficientes**, para ello hay que medir la eficiencia del algoritmo con su orden de magnitud, para tener certeza de que los algoritmos si son eficientes.

2.6.1 EVALUACIÓN DE ALGORITMOS

La evaluación de algoritmo consiste en medir la eficiencia de un algoritmo en cuanto a consumo de memoria y tiempo de ejecución.

Anteriormente, cuando existían grandes restricciones tecnológicas de memoria, **evaluar un algoritmo en cuanto a consumo de recursos de memoria era bastante importante**, ya que dependiendo de ella los esfuerzos de programación y de ejecución sería grande. **En la actualidad,** con el gran desarrollo tecnológico del hardware, **las instrucciones de consumo de memoria han pasado a un segundo plano**, por lo tanto, el aspecto de evaluar un algoritmo en cuanto a su consumo de memoria no es relevante.

En cuanto al **tiempo de ejecución**, a pesar de las altas velocidades de procesamiento que en la actualidad existen, **sigue siendo un factor fundamental**, especialmente en algoritmos que tienen que ver con **el control de procesos en tiempo real y en aquellos cuya frecuencia de ejecución es alta.**

Procedamos entonces a ver como **determinar el tiempo de ejecución de un algoritmo**. Básicamente **existen dos formas de hacer esta medición:** una que llamamos **a posteriori** y otra que llamamos a **priori**.

Consideremos la primera forma, a posteriori: el proceso de elaboración y ejecución de un algoritmo consta de los siguientes pasos:

- **Elaboración del algoritmo.**
- **Codificación en algún lenguaje de programación.**
- **Compilación del programa.**
- **Ejecución del programa en una maquina determinada.**

Cuando se ejecuta el programa, los sistemas operativos proporcionan la herramienta de informar cuánto tiempo consumió la ejecución del algoritmo.

Esta forma de medición tiene algunos inconvenientes: al codificar el algoritmo en algún lenguaje de programación estamos realmente midiendo la eficiencia del lenguaje de programación, ya **que no es lo mismo un programa codificado en FORTRAN, que en PASCAL o que en C**; al ejecutar un programa en una máquina determinada estamos introduciendo otro parámetro, el cual es la eficiencia de la máquina, ya que no es lo mismo ejecutar el programa en un XT, en un 386, en un 486, en un Pentium, es un AS400, es un RISC, etc.

En otras palabras, con esta forma de medición estamos es evaluando la calidad del compilador y de la máquina, más no la del algoritmo, que es nuestro interés.

Lo que nos interesa es medir la eficiencia del algoritmo por el hecho de ser ese algoritmo, independiente del lenguaje en el que se haya codificado y de la máquina en la cual se ejecute.

Veamos entonces la forma de medir la ejecución de un **algoritmo a priori.** Para ello necesitamos definir dos conceptos básicos que son: **Contador de frecuencias y Orden de magnitud.**

2.6.2 CONTADOR DE FRECUENCIAS

El contador de frecuencias **es una expresión algebraic**a que identifica el número de veces que se ejecutan las instrucciones de un método. Para ilustrar como determinar esta expresión consideramos los siguientes ejemplos:

MÉTODO PRINCIPAL1()		
1. LEA (a, b, c)	1	
2. x = a + b	1	
3. y = a + c	1	
4. z = b * c	1	
5. w = x / y – z	1	
6. IMPRIMA (a, b, c, w)	1	
Contador de frecuencias		6

MÉTODO PRINCIPAL1()		
1. LEA (n)	1	
2. s = 0	1	
3. i = 1	1	
4. MQ (i <= n)	n+1	
5. s = s + 1	n	
6. i = i + 1	n	
7. endMQ		n
8. IMPRIMA (n, s)	1	
Contador de frecuencias		4n+5

En todos los métodos las instrucciones están numeradas secuencialmente. Al frente de cada instrucción aparece un número o una letra que indica el número de veces que se ejecuta esa instrucción en el método.

En el **método 1** cada instrucción se ejecuta solo una vez. El total de veces que se ejecutan todas las instrucciones es 6. Este valor es el contador de frecuencias para el método 1.

En el **método 2** las instrucciones 1, 2, 3 y 8 se ejecutan solo una vez, mientras que las instrucciones 5, 6 y 7 se ejecutan **n** veces cada una, ya que pertenece a un ciclo, cuya variable controladora del ciclo se inicia en uno, tiene

un valor final de **n** y se incrementa de a uno, la instrucción 4 se ejecuta una vez más ya que cuando **i** sea mayor que **n** de todas formas hace la pregunta para luego salir del ciclo. Por tanto, el número de veces que se ejecutan las instrucciones del método es **4n + 5**. Esta expresión es el contador de frecuencias para el método 2.
Veamos otros ejemplos de métodos con su contador de frecuencias respectivo

METODO PRINCIPAL3()

1.	LEA (m, n)	1
2.	s = 0	1
3.	i = 1	1
4.	MQ (i <= n)	n+1
5.	t = 0	n
6.	j = 1	n
7.	MQ (j <= m)	n*m+n
8.	t = t + 1	n*m
9.	j = j + 1	n*m
10.	endMQ	n*m
11.	s = s + t	n
12.	i = i + 1	n
13.	endMQ	n
14.	IMPRIMA (n, m, s)	1
	Contador de frecuencias	4(n*m)+7n+5

MÉTODO PRINCIPAL4()

1.	LEA (n, m)	1
2.	s = 0	1
3.	i = 1	1
4.	MQ (i <= n)	n+1
5.	t = 0	n
6.	j = 1	n
7.	MQ (j <= n)	n2+n
8.	t = t + 1	n2
9.	j = j + 1	n2
10.	endMQ	n2
11.	s = s + t	n
12.	i = i + 1	n

13. endMQ		n
14. IMPRIMA (n, m, s)	1	
Contador de frecuencias		4n2+7n+5

MÉTODO PRINCIPAL5()		
1. LEA (n, m)		1
2. s = 0		1
3. i = 1		1
4. MQ (i <= n)		n+1
5. s = s + 1		n
6. i = i + 1		n
7. endMQ		n
8. IMPRIMA (n, s)		1
9. t = 0		1
10. i = 1	1	
11. MQ (i <= m)		m+1
12. t = t + 1	m	
13. i = i + 1	m	
14. endMQ		m
15. IMPRIMA (m, t)		1
Contador de frecuencias		4m+4n+9

De una manera análoga se obtuvieron los contadores de frecuencias para los métodos 3, 4 y 5.

2.6.3 EJERCICIOS DE ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Las pautas que debes utilizar para hallar el contador de frecuencia, es mirar el número de veces que se ejecuta cada instrucción en el algoritmo, recuerda que el contador de frecuencia es una expresión algebraica, que sale del número de veces que se ejecuta cada instrucción.

Calcule el contador de frecuencias de los siguientes métodos:

1. PUBLICO ESTATICO VOID Programa1 (n)

```
2.  s = 0
3.  PARA (i= 1, n, 1)
4.  PARA (j= 1, i, 1)
5.  PARA (k= 1, j, 1)
6.  s= s + 1
7.  endPARA
8.  endPARA
9.  endPARA
10. end(Método)
```

```
1.  CLASE Programa2
2.  METODO PRINCIPAL ()
3.  LEA (n)
4.  s = 0
5.  i = 2
6.  MQ (i <= n)
7.  s = s + 1
8.  i = i * i
9.  endMQ
10. IMPRIMA (n, s)
11. end(Método)
12. end(Clase)
```

```
1.  CLASE Programa3
2.  MÉTODO PRINCIPAL ()
3.  LEA (n)
4.  s = 0
5.  PARA (i= 1, n, 1
6.  t = 0
7.  PARA (j= 1, i, 1)
8.  t = t + 1
```

```
9.    endPARA
10.   s = s + t
11.   endPARA
12.   IMPRIMA (n, s)
13.    end(Método)
14.   end(Clase)
```

2.6.4 ORDEN DE MAGNITUD

EL ORDEN DE MAGNITUD ES EL CONCEPTO QUE DEFINE LA EFICIENCIA DEL MÉTODO EN CUANTO A TIEMPO DE EJECUCIÓN. SE OBTIENE A PARTIR DEL CONTADOR DE FRECUENCIAS.

Para obtener el orden de magnitud seguimos los siguientes pasos: Se eliminan los coeficientes, las constantes y los términos negativos de los términos resultantes; si son dependientes entre sí, se elige el mayor de ellos y este será el orden de magnitud de dicho método, de lo contrario el orden de magnitud será la suma de los términos que quedaron. Si el contador de frecuencias es una constante, como en el caso del método 1, el orden de magnitud es 1 y lo representamos como O (1).
Teniendo presente esto, los órdenes de magnitud de los métodos que vimos anteriormente son:

Método 1: O (1)
Método 2: O(n)
Método 3: O(n*m)
Método 4: $O(n^2)$
Método 5: O(n+m)

> **ES BUENO ACLARAR LO SIGUIENTE: en los ejemplos, el valor de n es un dato que se lee dentro del programa. En el caso más amplio, n podrá ser el número de registros que haya que procesar en un archivo, el número de datos que haya que producir como salida, o quizá otro parámetro diferente; es decir, hay que poner atención en el análisis que se haga del método, de cuál es el parámetro que tomamos como n.**

En el ambiente de sistemas, **los órdenes de magnitud más frecuentes** para los métodos que se desarrollan son:

ORDEN DE MAGNITUD	REPRESENTACION
Constante	O(1)
Lineal	O(n)
Cuadrático	$O(n^2)$
Cubico	$O(n^3)$
Logarítmico	$O(\log_2(n))$
Semilogarítmico	$O(n \log_2(n))$
Exponencial	$O(2^n)$

Si quisiéramos clasificar estas órdenes de magnitud **desde la más eficiente hasta la menos eficiente**, tendríamos el siguiente orden:

1.	Constante	O(1)
2.	Logarítmico	$O(\log_2(n))$
3.	Lineal	O(n)
4.	Semilogarítmico	$O(n \log_2(n))$
5.	Cuadrático	$O(n^2)$
6.	Cúbico	$O(n^3)$
7.	Exponencial	$O(2^n)$

Hasta aquí hemos visto **métodos cuyo orden de magnitud es constante, lineal y cuadrático**. De igual manera que se hizo el método con orden **de magnitud cuadrático** se puede hacer uno **cúbico**; en el cuadrático se necesitaron 2 ciclos anidados, controlados por la misma variable, para el cúbico se necesitarían 3 ciclos con las mismas condiciones.

Pasemos a tratar **métodos con orden de magnitud logarítmicos**. Empecemos definiendo lo que es un logaritmo.

La **definición clásica del logaritmo** de un número **x** es: es el exponente al cual hay que elevar un número llamado base para obtener **x.**

Con fines didácticos, presentamos otra definición de logaritmo: logaritmo de un numero **x** es el número de veces que hay que dividir un número, por otro llamado base, para obtener como cociente uno (1).

Por ejemplo, si tenemos el número 100 y queremos hallar su logaritmo en base 10, habrá que dividirlo por 10 sucesivamente hasta obtener como cociente uno (1).

100	10	
0	10	10
	0	1

Hubo que dividir el 100 dos veces por 10 para obtener 1 en el cociente, por tanto, el logaritmo en base 10 de 100 es 2 (**$\log_{10}(100) = 2$**).

Si queremos hallar el logaritmo en base 2 de 16 habrá que dividir 16 sucesivamente por 2 hasta obtener un cociente de 1. Veamos:

16	2			
0	8	2		
	0	4	2	
		0	2	2
			0	1

Es decir, hubo que dividir el 16, cuatro (4) veces por dos (2) para obtener un cociente de 1, por tanto, el logaritmo en base 2 de 16 es 4 (**$\log_2(16) = 4$**).

Planteemos un **ejemplo práctico de algoritmo** en el cual se presenta esta situación.
Consideremos el caso de un vector en el cual tenemos los datos ordenados ascendentemente:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	2	4	7	9	11	15	17	28	31	36	43	50	58	62	69

Si nos interesa buscar el dato **x = 85** y decir en cual posición del vector se encuentra hay dos formas de hacerlo:

Una, es **hacer una búsqueda secuencial** a partir del primer elemento e ir comparando el contenido de esa posición

con el dato **x** (85), el proceso terminará cuando lo encontremos o cuando hayamos recorrido todo el vector y no lo encontremos. Un pequeño método que haga ese proceso es el siguiente:
Sea **n** el número de elementos del vector y **x** el dato a buscar.

```
1. i = 1
2. MQ ((i <= n) & (V[i] != x))
3.         i = i + 1
4. endMQ
5. SI (i <= n)
6.         Pos = i
7. SINO
8.         IMPRIMA ("el dato no existe")
9. EndSI
```

En dicho método tenemos planteado un ciclo, el cual, en el peor de los casos, se ejecuta **n** veces, es decir, cuando el dato **x** no esté en el vector. Por tanto, el orden de magnitud de ese método es **O(n).**
En general, cuando se efectúa una búsqueda, **el peor caso es no encontrar lo que se está buscando**.

En nuestro ejemplo hubo que hacer 15 comparaciones para poder decir que el 85 no está en el vector.

Si el vector hubiera tenido quinientos mil de datos y hubiéramos tenido que buscar un dato que no estaba, hubiéramos tenido que haber hecho quinientas mil comparaciones para poder decir que el dato buscado no se encontró.

Ahora, si aprovechamos la característica de que los datos en el vector están ordenados podemos plantear **otro método de búsqueda**:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	2	4	7	9	11	15	17	28	31	36	43	50	58	62	69
								M							

Comparamos el dato **x** con el dato del elemento de la mitad del vector, (llamémoslo **M**). Pueden suceder tres casos: **V(M) == x; V(M) > x; V(M) < x**

Si **V(M) == x** se ha terminado la búsqueda.

Si **V(M) > x** significa que el dato **x**, si se encuentra en el vector, estará en la primera mitad del vector.

Si V(M) < x significa que, si el dato x se encuentra en el vector, estará en la segunda mitad del vector.
En nuestro ejemplo, para seguir buscando el dato x (85) lo haremos desde la posición 9 hasta la posición 15.

Es decir, con unas comparaciones hemos eliminado la mitad de los datos sobre los cuales hay que efectuar la búsqueda. En otras palabras, hemos dividido la muestra por dos (2).

El conjunto de datos sobre el cual hay que efectuar la búsqueda queda así:

9	10	11	12	13	14	15
31	36	43	50	58	62	69
			M			

De la mitad que quedo escogemos nuevamente el elemento de la mitad y repetimos la comparación planteada anteriormente. Con esta comparación eliminamos nuevamente la mitad de los datos que nos quedan, es decir, dividimos nuevamente por dos.

Si continuamos este proceso, con cuatro comparaciones podremos afirmar que el dato x no está en el vector.

Hemos reducido el número de comparaciones de 15 a 4.

Cuatro (4) es el logaritmo en base 2 de 15, aproximándolo por encima.

Si N fuera 65000 posiciones en el vector, con 16 comparaciones podremos decir que un dato no se encuentra en un conjunto de 65000 datos.
Como puede observarse, la ganancia en cuanto al número de comparaciones es grande.

Un algoritmo de búsqueda que funcione con esta técnica tiene orden de magnitud logarítmico en base dos, ya que la muestra de datos se divide sucesivamente por dos.

Consideremos ahora, métodos sencillos cuyo orden de magnitud es logarítmico:

```
1. n = 32
2. s = 0
3. i = 32
4. MQ (i > 1)
5.          s = s + 1
6.          i = i / 2
7. endMQ
```

```
8.  IMPRIMA (n, s)
```

En el método anterior, la variable controladora del ciclo es **i**, y dentro del ciclo, **i** se divide por dos (2). Si hacemos un seguimiento detallado tendremos:

- La primera vez que se ejecuta el ciclo, la variable i sale valiendo 16.
- La segunda vez que se ejecuta el ciclo, la variable i sale valiendo 8.
- La tercera vez que se ejecuta el ciclo, la variable i sale valiendo 4.
- La cuarta vez que se ejecuta el ciclo, la variable i sale valiendo 2.
- La quinta vez que se ejecuta el ciclo, la variable i sale valiendo 1 y termina el ciclo.

Es decir, las instrucciones del ciclo se ejecutan 5 veces.
Cinco (5) es el logaritmo en base dos (2) de **n** (32), por consiguiente, cada instrucción del ciclo se ejecuta un número de veces igual al logaritmo en base dos de **n**. El contador de frecuencias y el orden de magnitud de dicho método se presentan a continuación:

```
1.  n = 32                          1
2.  s = 0                           1
3.  i = 32                          1
4.  MQ (i > 1)                      log2 n + 1
5.        s = s + 1                 log2 n
6.        i = i / 2                 log2 n
7.  endMQ                           log2 n
8.  IMPRIMA (n, s)                  1

Contador de frecuencias             4 log2 n + 5

Orden de magnitud                   O(log2 (n))
```

Eliminando constantes y coeficientes el orden de magnitud es $O(\log_2 n)$.

Consideremos este nuevo método:

```
1.  n = 32                          1
2.  s = 0                           1
3.  i = 1                           1
4.  MQ (i < n)                          log2 n + 1
5.        s = s + 1                 log2 n
6.        i = i * 2                 log2 n
7.  endMQ                           log2 n
8.  IMPRIMA (n, s)                      1
```

En este método la variable controladora del ciclo **i** se multiplica por dos dentro del ciclo.

Haciendo un seguimiento tendremos:

- En la primera pasada, la variable i sale valiendo 2.
- En la segunda pasada, la variable i sale valiendo 4.
- En la tercera pasada, la variable i sale valiendo 8.
- En la cuarta pasada, la variable i sale valiendo 16.
- En la quinta pasada, la variable i sale valiendo 32 y el ciclo se termina.

Las instrucciones del ciclo se ejecutan 5 veces, por tanto, **el orden de magnitud del método es logarítmico en base 2 (O ($\log_2$ n)).**

En general, **para determinar si un ciclo tiene orden de magnitud lineal o logarítmica, debemos fijarnos cómo se está modificando la variable controladora del ciclo**: Si esta se modifica mediante sumas o restas el orden de magnitud del ciclo es lineal, si se modifica con multiplicaciones o divisiones el orden de magnitud del ciclo es logarítmico.

Consideremos ahora el siguiente método:

```
1. n = 81
2. s = 0
3. i = 81
4. MQ (i > 1)
5.         s = s + 1
6.         i = i / 3
7. endMQ
8. IMPRIMA (n, s)
```

Aquí, la variable controladora del ciclo **i** se divide por tres dentro del ciclo. Haciéndole el seguimiento tendremos:

- En la primera pasada, la variable i sale valiendo 27.
- En la segunda pasada, la variable i sale valiendo 9.
- En la tercera pasada, la variable i sale valiendo 3.
- En la cuarta pasada, la variable i sale valiendo 1, y termina el ciclo.

Las instrucciones del ciclo se ejecutaron 4 veces. Cuatro es el logaritmo en base 3 de 81. Por consiguiente, el orden de magnitud de dicho método es logarítmico en base 3 (O ($\log_3$ n)).

En general, si tenemos un método:

```
1. n = 81
2. s = 0
3. i = 81
4. MQ (i > 1)
5.       s = s + 1
6.       i = i / X
7. endMQ
8. IMPRIMA (n, s)
```

Aquí, la variable controladora del ciclo se divide por **x.** El número de veces que se ejecutan las instrucciones del ciclo es logaritmo en base **x** de **n** (O ($\log_x$ n)), por tanto, el orden de magnitud es logaritmo en base **x.**

Para obtener métodos con orden de **magnitud Semilogarítmico** basta con que un ciclo logarítmico se ubique dentro de un ciclo con orden de magnitud **n** (O(n)) o viceversa. Por ejemplo:

```
1.  LEA (n)                          1
2.  s = 0                            1
3.  i = 1                            1
4.  MQ (i <= n)                      n+1
5.       t = 0                       n
6.       j = n                       n
7.      MQ (j > 1)                   n*log2n +n
8.          t = t + 1                n*log2n
9.          j = j / 2                n*log2n
10.      endMQ                       n*log2n
11.      IMPRIMA (t)                 n
12.      s = s + t                   n
13.      i = i + 1                   n
14. endMQ                            n
15. IMPRIMA (n, s)                   1

Contador de frecuencias      4nlog2n+8n+5

Orden de magnitud            O(n log2n)
```

En el método anterior el orden de magnitud es **Semilogarítmico.**

Digamos que la idea es elaborar métodos lo más eficientes posible. Consideremos el siguiente problema:

Elaborar un método que lea un entero positivo **n** y que determine la suma de los enteros desde 1 hasta **n.**

Una solución es la siguiente:

```
1. LEA (n)
2. s=0
3. i = 1
4. MQ (i <= n)
5.       s = s + i
6.       i = i + 1
7. endMQ
8. IMPRIMA (n, s)
9.
```

El anterior método lee un dato entero **n** (**n>0**), calcula e imprime la suma de los enteros desde 1 hasta **n,** y el orden de magnitud de éste es **O(n)**.

Matemáticamente hablando, **la sumatoria de los enteros desde 1 hasta n, se representa así:**

$$S = \sum_{i=1}^{i=n} i$$

Esta representación, convertida en **fórmula matemática**, quedaría de la siguiente forma:

$$\frac{n * (n + 1)}{2}$$

Conociendo esta fórmula, podemos elaborar otro método que ejecute exactamente la misma tarea que el anterior. Veamos cómo sería:

```
1. LEA (n)
2. s = n * (n + 1) / 2
3. IMPRIMA (n, s)
```

Éste método tiene orden de **magnitud lineal (O(1))**. Es decir, tenemos dos métodos completamente diferentes

que ejecutan exactamente la misma tarea, uno con orden de magnitud **O(n)** y el otro con orden de magnitud **O (1).**

Lo anterior no significa que toda tarea podrá ser escrita con **métodos O(1),** no, sino que de acuerdo a los conocimientos que se tengan o a la creatividad de cada cual, se podrá elaborar métodos más eficientes.

Este ejemplo, el cual es válido, desde el punto de vista de desarrollo de métodos, **es aplicable también a situaciones de la vida real**. Seguro que alguna vez usted tuvo algún problema, y encontró una solución, y salió del problema, aplicando la solución que encontró, y sin embargo, **al tiempo**, dos o tres meses después, pesando en lo que había hecho, encontró que pudo haber tomado **una mejor determinación** y que le habría ido mejor.

La enseñanza es que cuando se nos presente un problema debemos encontrar como mínimo dos soluciones y luego evaluar cuál es más apropiada.

2.6.5 EJERCICIOS DE ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Recuerda que, primero se debe hallar el contador de frecuencia que es una expresión algebraica, luego de obtener el contador de frecuencia se pasa a hallar el orden de magnitud, que es el que define la eficiencia del algoritmo. Recuerda que, para hallar el orden de magnitud se eliminan los coeficientes, las constantes y los términos negativos de los términos resultantes

Determine el orden de magnitud de los siguientes métodos, a los cuales en un ejercicio pasado les calculamos el contador de frecuencias:

```
1.  PUBLICO ESTATICO VOID Programa1 (n)
2.      s = 0
3.      PARA (i= 1, n, 1)
4.          PARA (j= 1, i, 1)
5.              PARA (k= 1, j, 1)
6.                  s = s + 1
7.              endPARA
8.          endPARA
9.      endPARA
10. end(Método)
```

```
1.  CLASE Programa2
2.    MÉTODO PRINCIPAL ()
3.      LEA (n)
4.      s = 0
5.      i = 2
6.      MQ (i <= n)
7.          s = s + 1
8.          i = i * i
```

```
9.              endMQ
10.             IMPRIMA (n, s)
11.           end(Método)
12.       end(Clase)
```

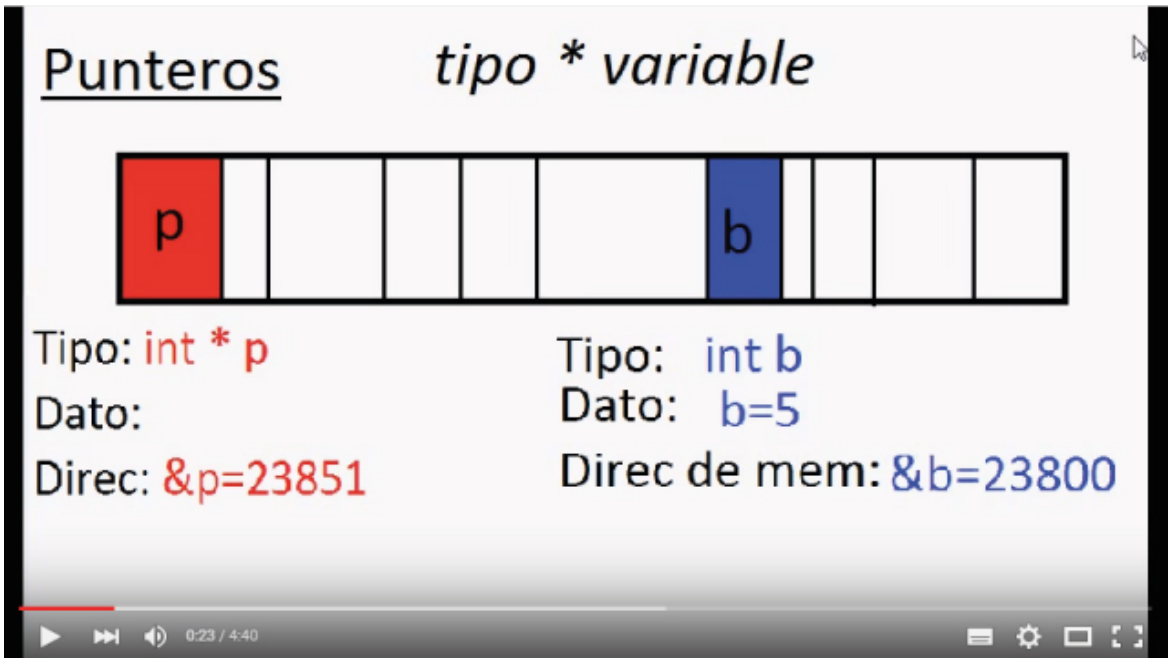
```
1.        CLASE Programa3
2.          MÉTODO PRINCIPAL ()
3.            LEA (n)
4.            s = 0
5.            PARA (i= 1, n, 1
6.                    t = 0
7.                    PARA (j= 1, i, 1)
8.                            t = t + 1
9.                    endPARA
10.                   s = s + t
11.           endPARA
12.           IMPRIMA (n, s)
13.         end(Método)
14.       end(Clase)
```

4. Elabore los siguientes métodos y determine el contador de frecuencias y el orden de magnitud dichos métodos:

a) Realizar un método que calcule la suma de la siguiente serie aritmética para n términos:
2, 5, 8, 11, 14, n

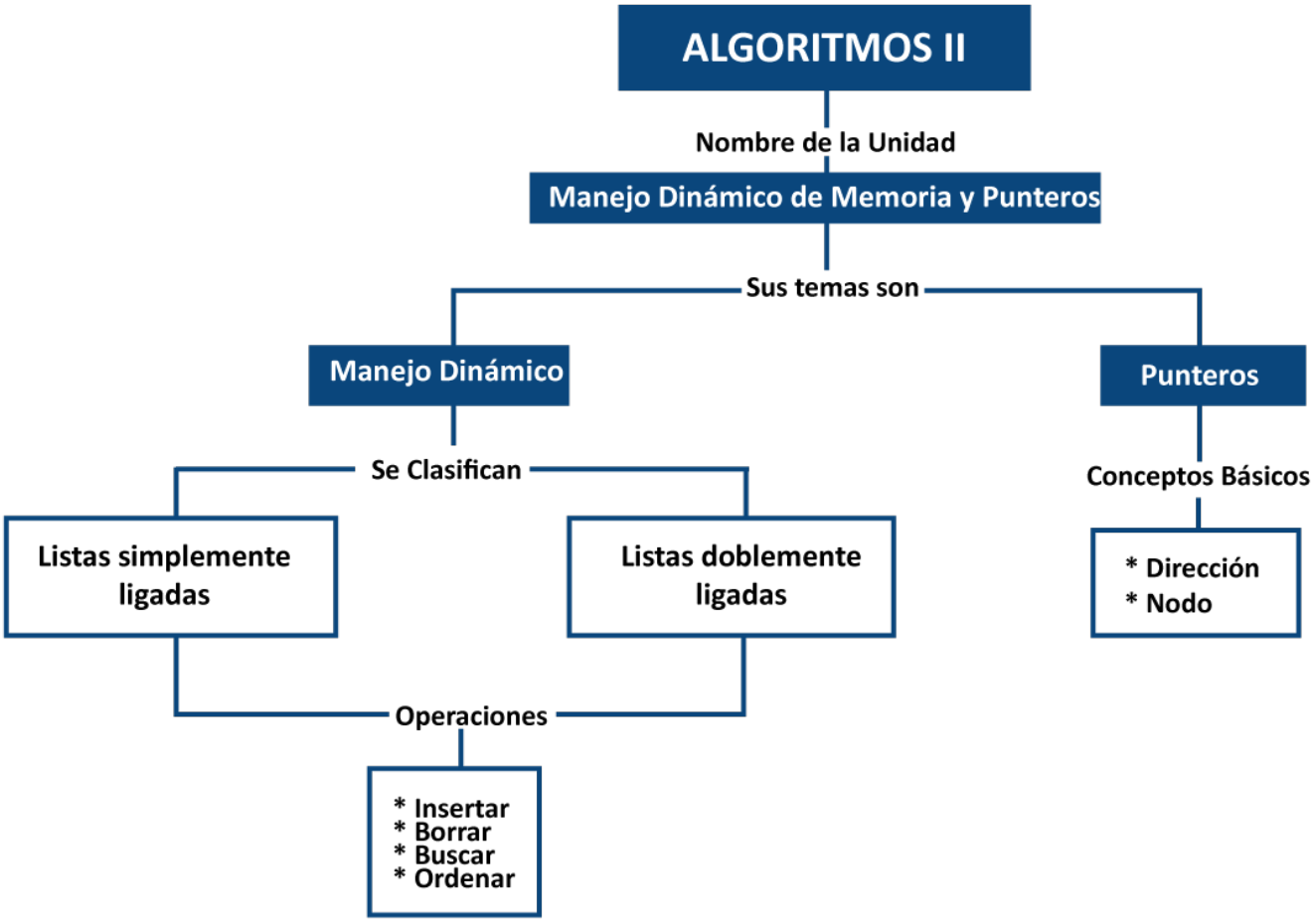
b) Realizar un método que calcule la suma de la siguiente serie aritmética para n términos:
2, 4, 8, 16, 32, 64, n

3 UNIDAD 2 MANEJO DINAMICO DE MEMORIA Y PUNTEROS



Introducción a Punteros: Enlace

3.1.1 RELACIÓN DE CONCEPTOS



Punteros: Direcciones de memoria que indica una posición de memoria dentro de la RAM.

Dirección: Es una dirección física de memoria que se representa por medio del sistema numérico hexadecimal dentro de la RAM.

Nodo: Es un espacio físico de memoria que tiene una dirección y se encuentra compuesto por una parte de datos y una parte de liga (dirección).

Manejo dinámico de memoria: Para optimizar el recurso de memoria a nivel de programación.

Las listas simplemente ligadas: Son estructuras donde se maneja la información de forma dinámica, donde el espacio donde se la guarda la información es un nodo, donde este se subdivide en dos espacios un espacio para guardar la información y otro para guardar la dirección de memoria al siguiente nodo.

Las listas doblemente ligadas: Son estructuras que permiten manejar la información de forma dinámica, donde el espacio donde se guarda la información es un nodo, que es una dirección de memoria dividida en tres partes, una parte para guardar la información, otra para la dirección al siguiente nodo y la otra para apuntar al nodo anterior.

Insertar: Consiste en ingresar un nuevo dato a la lista, de manera que la lista continúe ordenada.

Borrar: Consiste en borrar un dato un dato de la lista.

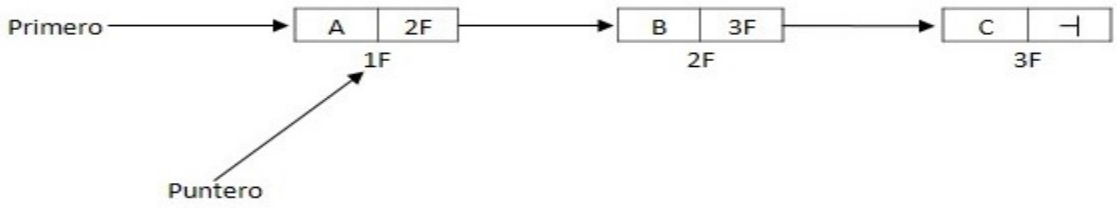
Buscar: Consiste en buscar un dato, si se encuentra o no en una lista.

Ordenar: Consiste en ordenar los datos de la lista, ya sea, en orden ascendente o descendente.

3.2 TEMA 1 PUNTEROS

Los punteros son direcciones de memoria que indica una posición de memoria dentro de la RAM, son muy utilizados con cuando se trabaja con memoria dinámica.

Para representar el concepto de puntero se mostrará una lista.



Donde cada cuadro contiene **dos campos**, uno para el dato y otro para la liga. Técnicamente, **cada cuadro se denomina nodo**, donde cada nodo posee una dirección de memoria.

Ejemplo: el primer cuadro tiene la dirección 1F, el segundo 2F y el tercero 3F.

3.2.1 CLASE NODOSIMPLE

Como **nuestro objetivo** es trabajar **programación orientada a objetos**, y para ello usamos clases con sus respectivos métodos, comencemos definiendo la clase en la cual manipulamos el nodo que acabamos de definir. A dicha clase la denominaremos **nodoSimple**:

1. CLASE **nodoSimple**
2. Privado
3. Objeto dato

```
4.          nodoSimple liga
5.      publico
6.          nodoSimple ()          // constructor
7.          objeto retornaDato ()
8.          nodoSimple retorna Liga ()
9.          void asigna Dato (Objeto d)
10.         void asigna Liga (nodoSimple x)
11. end (Clase nodoSimple)
```

Los algoritmos correspondientes a los métodos definidos son:

```
1.  nodoSimple (objeto d)      // constructor
2.      dato = d
3.      liga = null
4.  end(nodoSimple)
```

```
1.  objeto retornaDato ()
2.      return dato
3.  end(retornaDato)
```

```
1.  nodoSimple retorna Liga ()
2.      return liga
3.  end (retorna Liga)
1.  void asigna Dato (objeto d)
2.      dato = d
3.  end (asigna Dato)
```

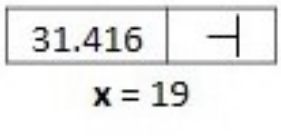
```
1.  void asigna Liga (nodoSimple x)
2.      liga = x
3.  end (asigna Liga)
```

Para usar esta clase consideremos el siguiente ejemplo:

```
1.  nodoSimple x
```

2. d = 3.1416
3. x = new nodoSimple(d)

Al ejecutar la instrucción 3 el programa se comunica con el sistema operativo y este le asigna una posición de memoria a nuestro programa, el cual identifica esta posición de memoria con la variable **x.** el resultado obtenido es:



En el cual **19 es la dirección del nodo en la que se almacenan los campos de dato y liga**. Dicha dirección la suministra el sistema operativo de una forma transparente para el usuario.

Si queremos acceder los campos del nodo **x**, lo haremos así:

d = x. retornaDato () **// d queda valiendo 3.1416**

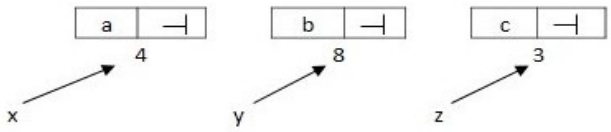
z = x. retorna Liga () **// z queda valiendo nulo**

A continuación, presentamos un ejemplo más completo de la utilización de la clase nodoSimple.

Supongamos que tenemos el conjunto de datos **a, b, c, f** y que se procesan con el siguiente algoritmo:

1. nodoSimple x, y, z, p
2. read(d) // lee la letra “a”
3. x = new nodoSimple(d) // digamos que envió el nodo 4
4. read(d) // lee la letra “b”
5. y = new nodoSimple(d) // digamos que envió el nodo 8
6. read(d) // lee la letra “c”
7. z = new nodoSimple(d) // digamos que envió el nodo 3

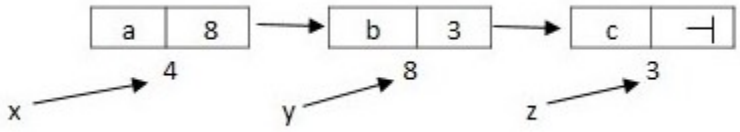
El **resultado grafico** al ejecutar estas instrucciones es:



Ahora, si ejecutamos las instrucciones:

8. x.asignaLiga(y)
9. y. asigna Liga(z)

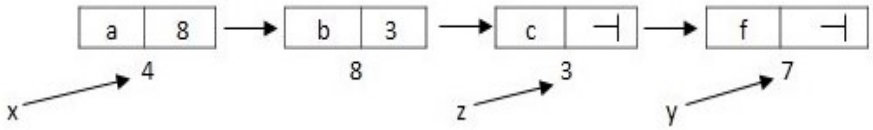
Los **nodos** quedaran conectados así:



Si ejecutamos las siguientes instrucciones:

10. read(d) // lee la letra "f"
11. y = new nodoSimple(d) // digamos que envió el nodo 7

Nuestros **nodos** quedan así:



Ahora, ejecutemos estas instrucciones:

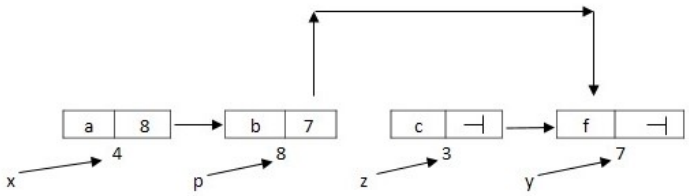
12. p= x. retorna Liga ()
13. write (p. retornaDato ())

La variable **p** queda valiendo 8 y escribe la letra "b".

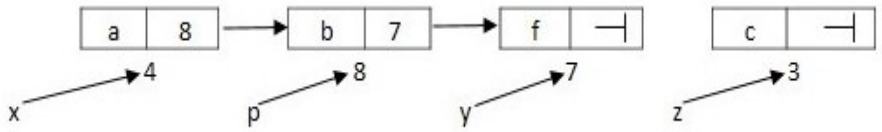
Y si ejecutamos esta otra instrucción:

14. x.retornaLiga (). asigna Liga(y)

Nuestros **nodos** quedan así:



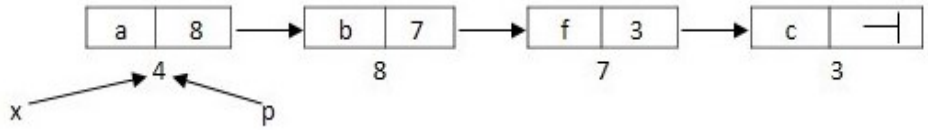
Observe que cuando se ejecuta x. retorna Liga () se está obteniendo el contenido del campo de liga del nodo **x**, el cual es el nodo 8 en nuestro ejemplo. Entonces. El nodo 8 invoca el método asigna Liga () y le estamos enviando como parámetro el nodo **y,** que en este momento vale 7. Podemos reorganizar el dibujo de nuestros nodos así:



Ahora, si ejecutamos las siguientes instrucciones:

15. y. asigna Liga(z)
16. y = null
17. z = null
18. p = x

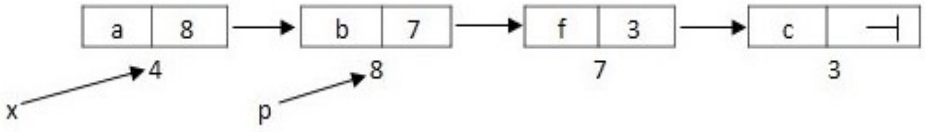
Nuestra lista queda así:



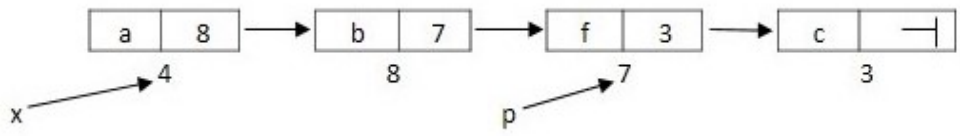
Y ahora ejecutemos estas instrucciones:

19. while (¡p! = null) do
20. write (p. retornaDato ())
21. p = p. retorna Liga ()
22. end(while)

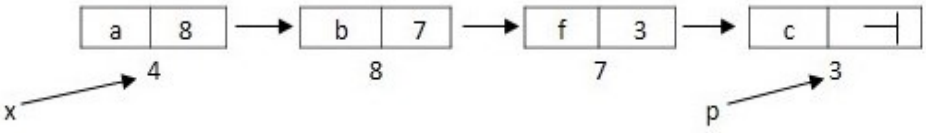
En la instrucción 20, la primera vez que entra al ciclo escribe la letra “a”, puesto que **p** vale 4. En la instrucción 21 modifica el valor de p: a **p** le asigna lo que hay en **el campo de liga del nodo 4, es decir, 8**. Nuestra lista está así:



La segunda vez que entra al ciclo escribe la letra “b” y la p queda valiendo 7. Nuestra lista está así:



La tercera vez que entra al ciclo escribe la letra “f” y la p queda valiendo 3:

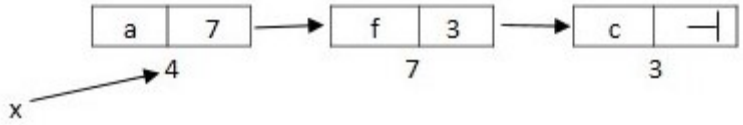


La cuarta vez que entra al ciclo escribe la letra "c" y la **p** queda valiendo null; por consiguiente, termina el ciclo. Este ciclo, instrucciones 19 a 22, **tiene orden de magnitud O(n),** siendo n el número de nodos de la lista.

Ahora, si por ultimo ejecutamos estas instrucciones:

23. y. asigna Liga (x. retorna Liga (). retorna Liga ())

La lista queda así:



Observe que no hemos dibujado el nodo 8 ya que no está conectado con los otros, pues se ha desconectado de la lista con la instrucción 23. Lo que hace la instrucción 23 se puede escribir también con las siguientes instrucciones:

23a. y = x. retorna Liga ()

23b. y. asigna Liga (y. retorna Liga ())

o

23a. y = x. retorna Liga ()

23b. y = y. retorna. Liga ()

23c. y. asigna Liga(y)

Es supremamente importante que entienda bien este ejemplo de aplicación de la clase nodoSimple.

3.3 TEMA 2 LISTAS SIMPLEMENTE LIGADAS

Las listas simplemente ligadas **son estructuras donde se maneja la información de forma dinámica**, donde el espacio donde se la guarda la información es un nodo, donde este se **subdivide en dos espacios** un espacio para guardar la información y otro para guardar la dirección de memoria al siguiente nodo, con estas listas se pueden realizar **las operaciones de insertar, ordenar, buscar, borrar y todo esto bajo el concepto de memoria dinámica.**

3.3.1 CLASE LISTAS SIMPLEMENTE LIGADA

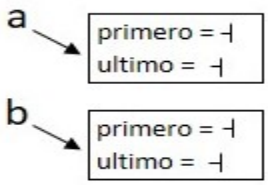
Es un **conjunto de nodos conectados** cuyo elemento básico son objetos de la clase **nodoSimple.** Con el fin de poder operar sobre este conjunto de nodos es necesario conocer el primer nodo del conjunto de nodos que están conectados, y en muchas situaciones el último nodo del conjunto. Con base en esto vamos a definir la clase llamada **LSL** (lista simplemente ligada), la cual tendrá dos datos privados de la clase **nodoSimple**, que llamaremos **primero** y **último:** primero apuntará hacia el primer nodo de la lista y último apuntará hacia el último nodo de la lista. Además, definiremos las operaciones que podremos efectuar sobre objetos de dicha clase.

```
1.  CLASE LSL
2.      Privado
3.          nodoSimple primero, ultimo
4.      Publico
5.          LSL ()              // constructor
6.          boolean es Vacía ()
7.          nodoSimple primer Nodo ()
8.          nodoSimple ultimo Nodo ()
9.          nodoSimple anterior (nodoSimple x)
10.         boolean finDeRecorrido (nodoSimple x)
11.         void recorre ()
12.         nodoSimple buscaDondeInsertar (Objeto d)
13.         void insertar (Objeto d, nodoSimple y)
14.         void conectar (nodoSimple x, nodoSimple y)
15.         nodoSimple buscar Dato (Objeto d, nodoSimple y)
16.         void borrar (nodoSimple x, nodoSimple y)
17.         void desconectar (nodoSimple x, nodoSimple y)
18.         void ordena Ascendentemente ()
19. end(Clase)
```

Expliquemos brevemente cada uno de los métodos definidos. Comencemos con el **constructo**r: cuando se ejecuta el constructor lo único que se hace es crear una nueva instancia de la clase **LSL** con sus correspondientes datos privados en null. Ejemplo: si tenemos estas instrucciones:

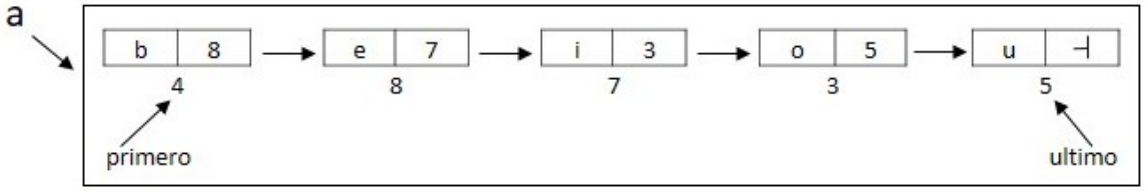
```
1. LSL a, b
2. a = new LSL ()
3. b = new LSL ()
```

Lo que se obtiene es:



Para explicar qué es lo que hace cada uno de los métodos definidos en la clase LSL consideremos el siguiente objeto, perteneciente a la clase LSL y que llamamos **a:**

La función **es Vacía ()** retorna verdadero si la lista que invoca el método está vacía; de lo contrario, retorna falso. Una lista está vacía cuando la variable primero es null.



La función **primer Nodo ()** retorna el nodo que esta de primero en la lista. Si efectuamos la instrucción **p = a. primer Nodo (),** entonces **p** quedara valiendo 4.

La función **ultimo Nodo (),** retorna el nodo que esta de ultimo en la lista. Si ejecutamos la instrucción **p = a. ultimo Nodo (),** entonces **p** quedara valiendo 5.

La función **finDeRecorrido(p)** retorna verdadero si el nodo **p** enviado como parámetro es null; de lo contrario, retorna falso.

La función **anterior(x)** retorna el nodo anterior al nodo enviado como parámetro. Si tenemos que x = 7 y ejecutamos la instrucción **p = a. anterior(x),** entonces **p** quedara valiendo 8.

El método **recorre ()**, como su nombre lo dice, simplemente recorre y escribe los datos de una lista simplemente ligada. Si ejecutamos la instrucción **a. Recorre ()**, el resultado que se obtiene es la escritura de **b, e, i, o, u.**

La función **buscaDondeInsertar(d)** retorna el nodo a continuación del cual se debe insertar un nuevo nodo con dato **d** en una lista simplemente ligada en la que los datos están ordenados ascendentemente y deben continuar cumpliendo esta característica después de insertarlo. Presentemos algunos ejemplos:

y = a. BuscaDondeInsertar("f") entonces **y** queda valiendo 8.

y = a. BuscaDondeInsertar("z") entonces **y queda** valiendo 5.

y = a. BuscaDondeInsertar("a") entonces **y queda** valiendo null.

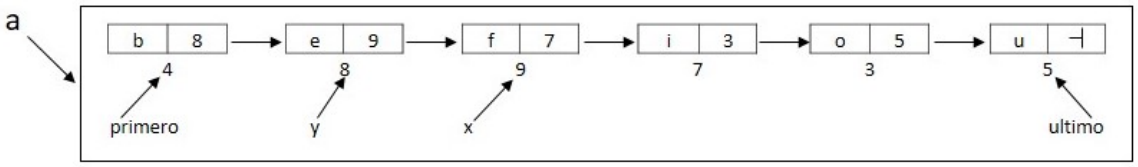
El método **insertar (d, y)** consigue un nuevo **nodoSimple**, lo carga con el dato **d** e invoca el método conectar con el fin de conectar el nuevo nodo (llamémoslo **x**) a continuación del nodo **y.** Si ejecutamos las siguientes instrucciones:

d = "f"

y = a. BuscaDondeInsertar(d)

a. Insertar (d, y)

El objeto **a,** quedara así:



El método **conectar** simplemente conecta el nodo **x** a continuación del nodo **y,** tal como se ve en la figura anterior.

La función **buscar Dato (d, y)**, como su nombre lo dice, busca el dato **d** en la lista invoca el método: si lo encuentra, retorna el nodo en el cual lo encontró; de lo contrario, retorna null. En el parámetro **y,** el cual debe ser un parámetro por referencia, retorna el nodo anterior al nodo en el cual encontró el dato **d.** algunos ejemplos son:

x = a. buscar Dato ("f", y) entonces **x** queda valiendo 9 y **y** queda valiendo 8.

x = a. buscar Dato ("u", y) entonces **x** queda valiendo 5 y **y** queda valiendo 3.

x = a. buscar Dato ("b", y) entonces **x** queda valiendo 4 y **y** queda valiendo null.

x = a. buscar Dato ("m", y) entonces **x** queda valiendo null y **y** queda valiendo último, es decir 5.

El método **borrar (x, y)** controla que el parámetro **x** sea diferente de null: si **x** es null produce el mensaje de que el dato **d** (el dato buscado con el método buscar Dato) no se halla en la lista y retorna; si **x** es diferente de null, invoca el método desconectar (x, y).

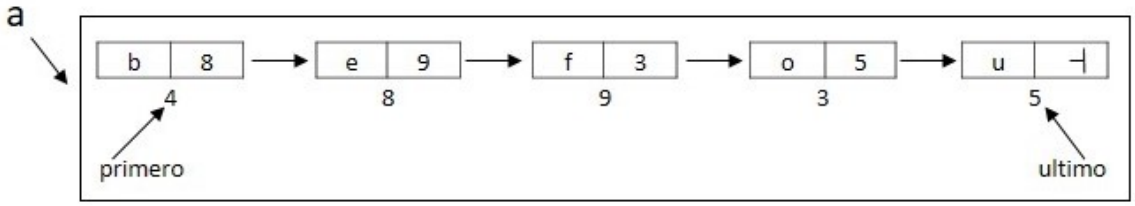
El método **desconectar (x, y)** simplemente desconecta el nodo **x** de la lista que invoca el método. Para desconectar un nodo de una lista se necesita conocer cuál es el nodo anterior. Por lo tanto, el nodo **y**, el segundo parámetro, es el nodo anterior a **x**. si ejecutamos las siguientes instrucciones:

d = "i"

x = a. buscar Dato (d, y) // **x** queda valiendo 7 y **y** queda valiendo 9

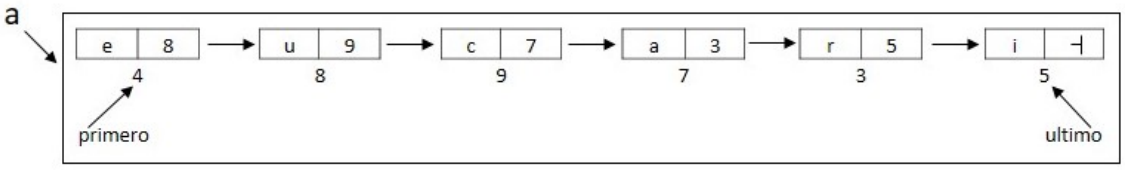
a. Borrar (x, y) // desconecta el nodo **x**

Esta queda así:

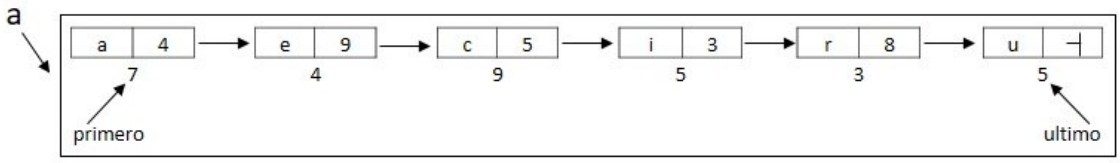


El método **ordena Ascendentemente ()**, como su nombre lo dice, **reorganiza los nodos de una lista simplemente ligada**, de tal manera que los datos en ella queden ordenados ascendentemente.

Si tenemos la siguiente lista:



Y ejecutamos la instrucción **a. ordena Ascendentemente ()**, la lista quedara así:



Observe **que los datos no se han movido**, son los nodos los que han cambiado de posición y por consiguiente primero y último han variado.

Asegúrese de entender bien que es lo que hace cada método. Si usted conoce bien lo que hace cada método, podrá usarlos apropiadamente; de lo contrario, no**. Como ingeniero de sistemas, debe saber cómo usar los métodos de una clase y además como implementarlos.**

3.3.2 IMPLEMENTACIÓN DE LOS MÉTODOS DE LA CLASE LISTA SIMPLEMENTE LIGADA

El constructor:

LSL ()

Primero = ultimo = null

```
end(LSL)
```

Simplemente inicializa los datos privados primero y último en null.

```
Boolean es Vacía ()
        Return primero == null
end (es Vacía)
```

Recuerde que tanto C como en java la instalación

```
return x == y
```

Es equivalente a:

```
if (x == y)
        return true
else
        return false
nodoSimple primerNodo()
        return primero
end(primero)
nodoSimple ultimoNodo()
        return ultimo
end(ultimo)
boolean finDeRecorrido (nodoSimple x)
        return x == null
end(finDeRecorrido)
```

```
1.  void recorre ()
2.          nodoSimple p
3.          p = primerNodo ()
4.          while no finDeRecorrido(p) do
5.                  Write (p. retornaDato ())
```

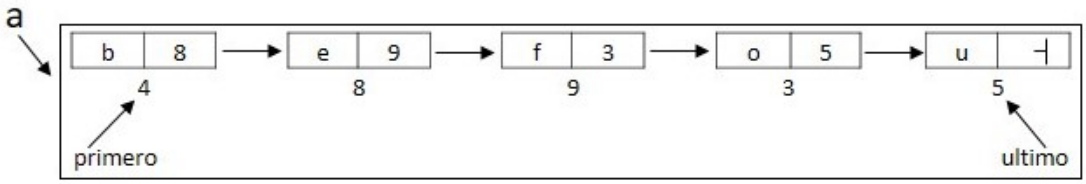
6. p = p. retorna Liga ()
7. end(while)
8. end(Método)

Para **recorrer y escribir** los datos de una lista simplemente ligada se **requiere una variable auxiliar**, la cual llamamos **p.** dicha variable se inicializa con el primer nodo de la lista simplemente ligada (instrucción 3 del método anterior) y luego planteamos un ciclo (instrucciones 4 a 7), el cual se ejecuta mientras **p** sea diferente a null (método finDeRecorrido ()). Cuando **p** sea null se sale del ciclo y termina la tarea. **Observe que, si por alguna razón el objeto que invoca este método tiene la lista vacía, nuestro método funciona correctamente**. Simplemente no escribe nada. Es decir, la instrucción 4 se controlan dos situaciones: lista vacía y fin de recorrido.

3.3.3 PROCESO DE INSERCIÓN DE UN DATO EN UNA LISTA SIMPLEMENTE LIGADA

Para insertar un dato **d** en una lista ligada hemos definido tres métodos en nuestra clase **LSL: buscarDondeInsertar(d), insertar (d, y) y conectar (x, y).** Analicemos cada uno de ellos. Es importante recordar que en este proceso de inserción se requiere que los datos de la lista se hallen ordenados en forma ascendente.

Consideremos la siguiente lista y que se desea insertar el dato “g”. Lo primero que se debe hacer es determinar en qué sitio debe quedar la letra “g” para que se cumpla que los datos continúen ordenados en forma ascendente. Por observación, nos damos cuenta de que la “g” debe quedar entre la “f” y la “o”, es decir, a continuación del nodo 9.



Nuestro primer método, buscaDondeInsertar(d), efectúa la tarea que determina a continuación de cual nodo debe quedar el dato a insertar. El método para esta tarea es:

1. **nodoSimple** buscaDondeInsertar (Objeto d)
2. nodoSimple p, y
3. p = primerNodo ()
4. y = anterior(p)
5. while (no finDeRecorrido(p) and p. retornaDato () < d) do
6. y = p
7. p = p. retorna Liga ()
8. end(while)
9. return y

10. end(buscaDondeInsertar)

En este método **se requieren dos variables auxiliares**, las cuales llamamos **p** y **y.** con la variable **p** se recorre y a medida que la vamos recorriendo se va comparando el dato de **p** (p. retornaDato ()) con el dato **d** que se desea insertar. Mientras que el dato de **p** sea menor que **d** se avanza en la lista con **p** y con **y**. la variable **y** siempre estará apuntando hacia a **p**. como inicialmente **p** es el primer nodo, inicialmente **y** es null. **Fíjese** que si el dato **d** a insertar es menor que el dato del primer nodo de la lista simplemente ligada nuestro método retorna null. El hecho de que nuestro método retorne null significa que el dato a insertar va a quedar de primero en la lista.

Veamos ahora como es el algoritmo para nuestro método **Insertar (d, y)**:

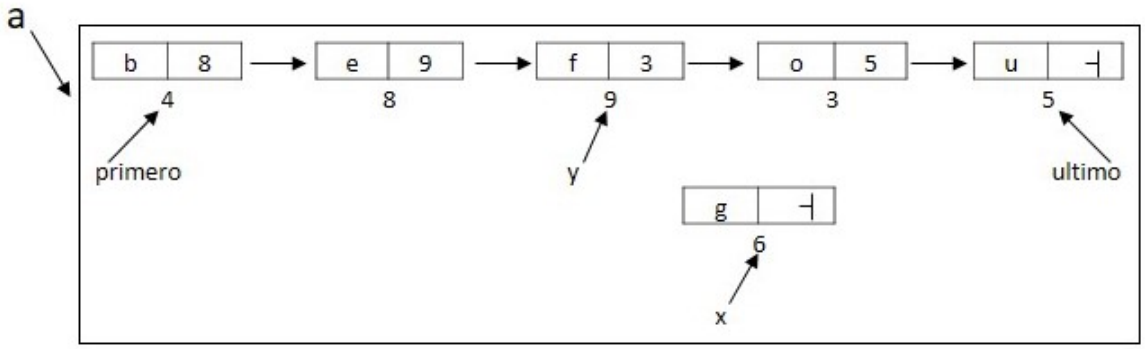
```
1. void insertar (Objeto d, nodoSimple y)
2.         nodoSimple x
3.         x = nodoSimple(d)
4.         conectar (x, y)
5. end(Método)
```

Nuestro **método insertar** consigue simplemente un nuevo nodo, lo carga con el dato **d** e invoca el método conectar. El parámetro **y** se refiere al nodo a continuación del cual habrá que conectar el nuevo **x.**

Ocupémonos del método conectar (x, y):

Caso 1:

Al ejecutar y = buscaDondeInsertar(d) con d = "g" en el objeto de la figura anterior y el método insertar (d, y), estamos en la situación mostrada en la figura siguiente:

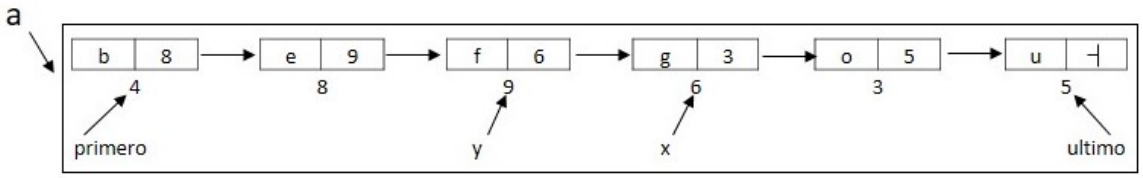


Conectar el nodo **x** a continuación del nodo **y** implica que cuando lleguemos al nodo **y** debemos trasladarlo hacia el nodo **x**, o sea que el campo de liga del nodo **y** debe quedar valiendo 6, y cuando estemos en el nodo **x** debemos trasladarnos hacia el nodo 3, es decir, que el campo liga del nodo **x** debe quedar valiendo 3. O sea que, **hay que modificar dos campos de liga**: el campo de liga del nodo **y** y el campo de liga del nodo **x**. para lograr esto debemos ejecutar las siguientes instrucciones:

y. asigna Liga (y. retorna Liga ()) // modifica el campo de liga del nodo x

y. asigna Liga(x) // modifica el campo de liga del nodo y

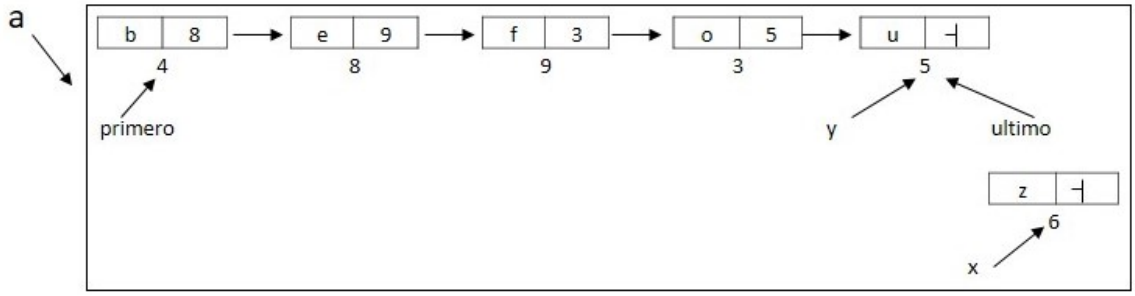
Y la lista queda así:



Y como se podrá observar, se ha insertado el nodo **x** a continuación del nodo **y**

Caso 2:

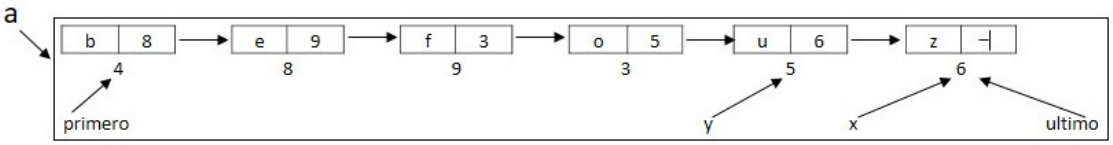
Consideremos que el dato a insertar es d = "z". Al ejecutar **y=buscaDondeInsertar(d)**, y el método **insertar (d, y)**, estamos en la siguiente situación:



Conectar **x** a continuación de **y** se logra con las mismas instrucciones del caso 1, teniendo en cuenta que como el dato que se inserta queda de ultimo hay que actualizar la variable último. **Lo anterior significa** que las instrucciones para conectar **x** a continuación de **y** serán:

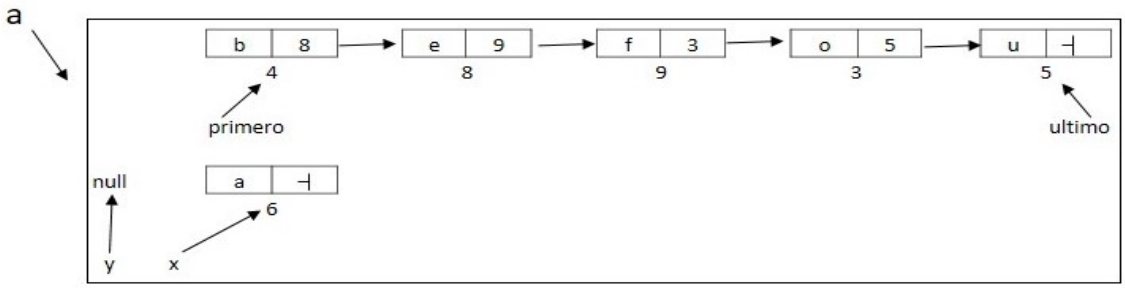
```
y. asigna Liga (y. retorna Liga ())  // modifica el campo de liga del nodo x
y. asigna Liga(x)                    // modifica el campo de liga del nodo y
if (y == ultimo) then
    ultimo = x
end(if)
```

Y la lista queda así:



Caso 3:

Consideremos que el dato a insertar es d = "a". al ejecutar **y=buscaDondeInsertar(d)** y el método **insertar (d, y)**, estamos en la situación mostrada:



En esta situación, cuando la **y** es null, es decir que **el dato a insertar quedara de primero**, hay que modificar el campo de la liga **x** y la variable primero:

y. asigna Liga(primero)

primero = x

Pero hay que tener en cuenta que la lista pudo haber estado vacía. Si esa fuera la situación habría que actualizar también la variable última. Considerando estas situaciones, las instrucciones completas para conectar un nodo **x** al principio de la lista son:

```
y. asigna Liga(primero)
if (primero == null) then
        ultimo = null
end(if)
primero = x
```

El algoritmo completo para el método conectar consiste simplemente en agrupar las instrucciones descritas para cada uno de los casos en un solo método. **Dicho método se presenta a continuación**: en las instrucciones 2 a 7 se consideran los casos 1 y 2, mientras que en las instrucciones 9 a 13 se considera el caso 3:

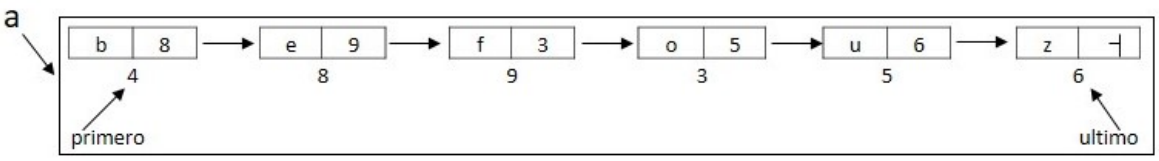
1. **void** conectar (nodoSimple x, nodoSimple y)

```
2.      if (¡y! = null) then
3.          y. asigna Liga (y. retorna Liga ())
4.          y. asigna Liga(x)
5.          if (y == ultimo) then
6.              ultimo = x
7.          end(if)
8.      Else
9.          y. asigna Liga(primero)
10.         if (primero == null) then
11.             ultimo = x
12.         end(if)
13.         primero = x
14.     end(if)
15. end(Método)
```

3.3.4 PROCESO DE BORRADO DE UN DATO EN UNA LISTA SIMPLEMENTE LIGADA

Borrar un dato de una lista simplemente ligada **consiste en ubicar el nodo en el cual se halla el dato y desconectarlo de lista**. Para ello hemos definido tres métodos que hemos llamado **buscar Dato (d, y), borrar (x, y) y desconectarlo (x, y).**

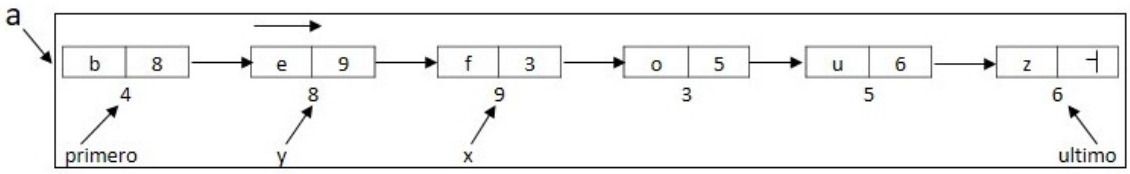
Si tenemos la siguiente lista:



Y queremos borrar el nodo que contiene la letra "f", lo primero que debemos hacer es determinar en cual nodo se halla la letra "f" y cuál es su nodo anterior. Esta tarea se efectúa con el método **buscar Dato (d, y):** ejecutamos la instrucción

x = a. buscar Dato (d, y)

Y estamos en esta situación:



El algoritmo correspondiente al método buscar Dato (d, y) es similar al método **buscarDondeInsertar(d)** y lo presentamos a continuación.

```
1.  nodoSimple buscar Dato (Objeto d, nodoSimple y)
2.          nodoSimple x
3.          x = primerNodo ()
4.          y = anterior(x)
5.          while (no finDeRecorrido(x) and x. retornaDato ()) != d do
6.                  y = x
7.                  x = x. retorna Liga ()
8.          end(while)
9.          return x
10. end (buscar Dato)
```

Utilizamos la variable **x** para recorrer la lista e ir comparando el dato del nodo **x** con **d**. de este ciclo, instrucciones 5 a 8, se sale cuando encuentre o cuando haya terminado de recorrer la lista. **Observe** que cuando el dato que se busca no se halla en la lista el parámetro **y** queda valiendo último. Además, cuando el dato que se busca es el primer dato de la lista, la variable **y** queda valiendo null.

Nuestro método borrar (x, y) consiste simplemente en controlar que el nodo **x** sea diferente de null. Si el nodo **x** es null significa que el dato no se encontró en la lista; por consiguiente, no hay ningún nodo para desconectar, entonces produce el mensaje correspondiente y retorna al programa llamante puesto que no hay nada más que hacer. Si **x** es diferente de null se invoca el método desconectar. **El algoritmo correspondiente a dicho método se presenta a continuación:**

```
void borrar (nodoSimple x, nodoSimple y)
        if x == null then
                write (“dato no existe”)
                return
        end(if)
        desconectar (x, y)
end(Método)
```

Analicemos ahora las **instrucciones correspondientes al método desconectar (x, y).**

Caso 1:

Es la situación mostrada en la figura anterior. Lo único que hay que hacer para desconectar el nodo **x** es modificar el campo de liga del nodo **y**, asignándole lo que hay en el campo liga del nodo **x**. esto se logra con la siguiente instrucción:

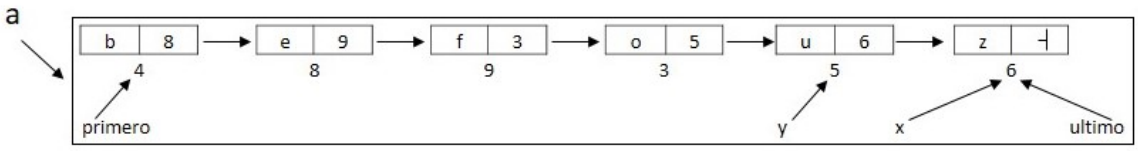
y. asigna Liga (x. retorna Liga ())

Caso 2:

Si el dato a borrar es d = "z", al ejecutar la instrucción

x = a. buscar Dato (d, y)

La situación es:



Por consiguiente, cuando se desconecte el nodo **x** la variable última debe quedar valiendo **y** y las instrucciones serán:

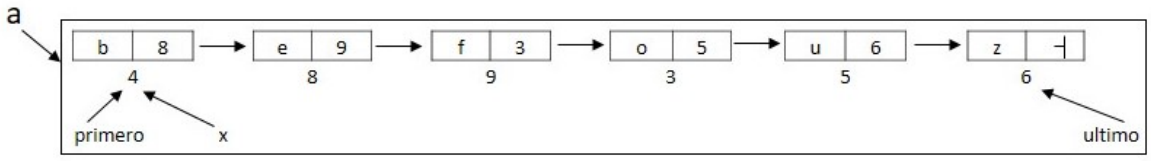
```
y. asigna Liga (x. retorna Liga ())
if (x == ultimo) then
        ultimo = y
end(if)
```

Caso 3:

Si el dato a borrar es d = "b" al ejecutar la instrucción

x = a. buscar Dato (d, y)

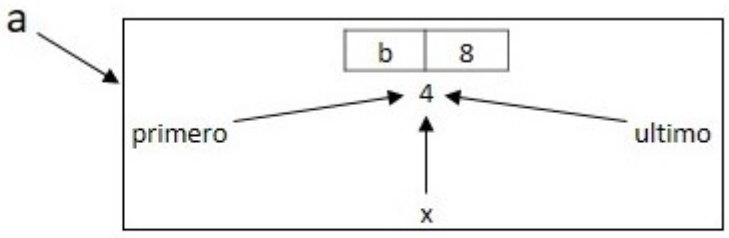
La situación es:



Con la variable **y** valiendo null.

En esta situación lo que hay que hacer es actualizar la variable primero: a **primero** se le asigna lo que haya en el campo de liga primero. Hay que tener en cuenta que puede suceder que la lista solo hubiera tenido un nodo; en este caso la variable **primero** quedará valiendo null, y por lo tanto la variable ultimo también deberá quedar valiendo null. Las instrucciones para resolver este caso son:

```
primero = primero. retorna Liga ()
if (primero == null) then
        ultimo = null
end(if)
```



Agrupando las instrucciones correspondientes a los **tres casos en un solo algoritmo** obtenemos:

```
1.  void desconectar (nodoSimple x, nodoSimple y)
2.      if (x != primero) then
3.          y. asigna Liga (x. retorna Liga ())
4.          if (x == ultimo) then
5.              ultimo = y
6.          end(if)
7.      else
8.          primero = primero. retorna Liga()
9.          if (primero == null) then
10.             ultimo = null
11.         end(if)
12.     end(if)
13. end(Método)
```

En las instrucciones 2 a 6 se resuelve para los casos 1 y 2, mientras que en las instrucciones 8 a 11 se resuelve para el caso 3.

Los métodos para conectar y desconectar son métodos con orden de magnitud 0(1), lo cual mejora sustancialmente estas operaciones con respecto a la representación de datos en un vector.

3.3.5 ORDENAMIENTO DE DATOS EN UNA LISTA SIMPLEMENTE LIGADA

Presentemos a continuación un método que utiliza el método *selección, el cual,* se enuncia así: "de los datos que faltan por ordenar, determine cuál es el menor y colocarlo de primero en ese conjunto de datos".

```
1.  void Ordena Ascendentemente ()
2.      nodoSimple p, ap., menor, a menor, q, aq
3.      p = primerNodo ()
4.      ap. = anterior(p)
5.      while (¡p! = ultimoNodo ())
6.          menor = p
7.          a menor = ap.
8.          q = p. retorna Liga
9.          aq =p
10.         while (no finDeRecorrido(q))
11.             if (q. retornaDato () < menor. RetornaDato ())
12.                 menor = q
13.                 a menor = aq
14.             end(if)
15.             aq =q
16.             q = q. retorna Liga ()
17.         end(while)
18.         if (menor == p)
19.             ap. = p
20.             p = p. retorna Liga ()
21.         else
22.             desconectar (menor, a menor)
23.             conectar (menor, ap.)
24.             ap. = menor
25.         end(if)
26.     end(while)
27. end(Método)
```

Con base en este enunciado definimos **las variables necesarias para elaborar el método**. Se requiere una variable que indique a partir de cual nodo faltan datos por ordenar. Dicha variable la llamamos **p**. inicialmente **p** apunta hacia el primer nodo ya que al principio faltan todos los datos por ordenar. Se requiere otra variable para recorrer la lista e ir comparando los datos para determinar en cual nodo se halla el menor dato. Esta variable la llamamos **q**. y por ultimo requerimos otra variable que apunte hacia el nodo que tiene el menor dato. Esta variable la llamamos **menor**.

Nuestro método consistirá entonces en determinar cuál es el nodo que tiene el menor dato entre los datos que faltan por ordenar; desconectamos el nodo de ese sitio y lo conectamos al principio de ese conjunto de datos.

Como ya hemos visto, **para efectuar las operaciones de conexión y desconexión se requiere conocer los registros anteriores a los nodos con los cuales se desean ejecutar dichas operaciones**; por lo tanto, definimos otras tres variables que llamaremos **ap., aq** y **a menor,** las cuales apuntan hacia los nodos anteriores a **p, q** y **menor** respectivamente.

En la instrucción 5 definimos el ciclo principal del ciclo. El método terminara cuando **p** este apuntando hacia el último nodo de la lista, puesto que cuando **p** este apuntando hacia ese nodo significa que solo falta un dato por ordenar, y un solo dato esta ordenado.

En las instrucciones 6 y 7 asignamos los valores iniciales de **menor** y **a menor**. Inicialmente consideramos que el menor dato es el principio del conjunto de datos que faltan por ordenar, es decir, **p**.

En las instrucciones 8 y 9 se asignan los valores iniciales de **q** y **aq.**

En las instrucciones 10 a 17 se efectúa el ciclo para determinar cuál es el nodo que contiene el menor dato. Se compara el dato de nodo **q** con el dato del nodo **menor.** Si el dato de **q** es menor que el dato de **menor**, actualizaremos menor y su interior. Cualquiera que hubiera sido el resultado avanzamos con **q** y su anterior.

En la instrucción 18 confrontamos en cual posición está el nodo de tiene el menor dato. Si el menor dato estaba de primero en ese conjunto de datos, es decir, en **p**, simplemente avanzamos con **p** y su anterior (instrucciones 19 y 20); **de lo contrario**, desconectamos el nodo **menor,** cuyo anterior es **a menor**, y lo conectamos a continuación de **ap.** (instrucciones 22 y 23). De haber ocurrido esto último, el anterior a **p**, es decir **ap.**, ya es **menor** (instrucción 24).

3.3.6 MÉTODO ANTERIOR(X)

Ya hemos definido que nuestro método denominado **anterior(x)** retornara el nodo anterior al nodo **x** enviado como parámetro. Para ello bastara con recorrer la lista simplemente ligada utilizando **dos variables auxiliares**: una (llamémosla **p**) que se va comparando con x, y otra que siempre apuntara hacia el nodo anterior a **x** (llamémosla **y**). cuando **p** sea igual a **x,** simplemente se retorna **y**. inicialmente será el primer nodo y **y** será null. Nuestro método es el siguiente:

```
nodoSimple anterior(x)

        nodoSimple p, y
```

```
        p = primerNodo ()
        y = null
        while (¡p! = x) do
            y = p
            p = p. retornaLiga ()
        end(while)
        return y
end(anterior)
```

3.3.7 CONSTRUCCIÓN DE LISTAS SIMPLEMENTE LIGADAS

Pasemos ahora a considerar como construir listas simplemente ligadas. Básicamente hay tres formas de hacerlo: una, que los datos queden ordenados ascendentemente a medida que la lista se va construyendo; otra, insertando nodos siempre al final de la lista; y una tercera, insertando los nodos siempre al principio de la lista.

Para la primera forma de construcción (que los datos queden ordenados ascendentemente a medida que se va construyendo la lista), un método general es:

```
LSL a
a = new LSL ()
Mientras haya datos por leer haga
    Lea(d)
    y = a. BuscaDondeInsertar(d)
    a. Insertar (d, y)
end(mientras)
```

Es decir, basta con plantear un ciclo para la lectura de datos e invocar los métodos para buscar donde insertar e insertar desarrollados previamente.

Para la segunda forma de construcción (insertando nodos siempre al final de la lista), un método es:

```
LSL a
nodoSimple
```

```
a = new LSL ()
mientras haya datos por leer haga
        lead(d)
        y = a. ultimoNodo ()
        a. Insertar (d, y)
end(mientras)
```

Nuevamente se plantea un ciclo para la lectura de datos y cada vez que se lea un dato se determina el último y se invoca el método para insertar.

Para la tercera forma de construcción (insertando nodos siempre al principio de la lista), nuestro método es:

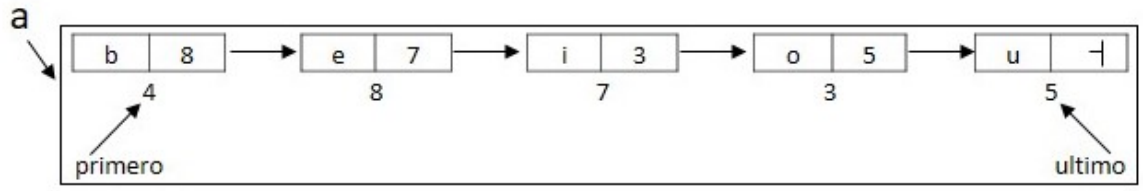
```
LSL a
a = new LSL
Mientras haya datos por leer haga
        lead(d)
        a. Insertar (d, null)
end(mientras)
```

Aquí, basta con plantear el ciclo para lectura de datos e invocar el método insertar enviado como segundo parámetro null. Recuerde que nuestro método insertar (d, y) inserta un nodo con dato **d** a continuación de **y**: si **y** es null significa que el dato **d** quedara de primero.

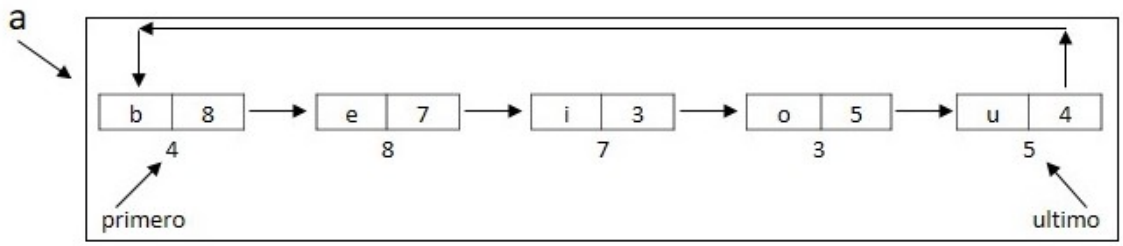
3.3.8 DIFERENTES TIPOS DE LISTAS SIMPLEMENTE LIGADAS Y SUS CARACTERÍSTICAS

Comencemos presentando gráficamente los diferentes tipos de lista que se pueden construir:

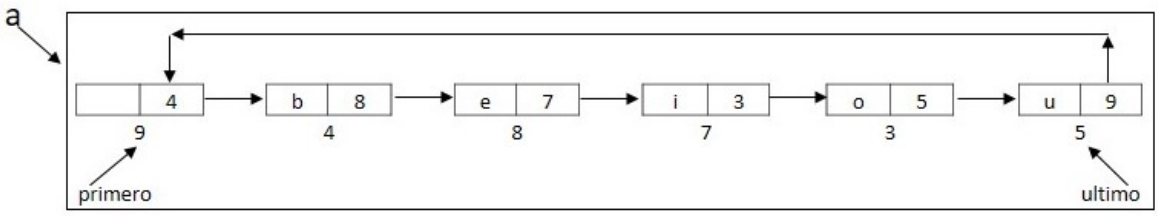
- Listas simplemente ligadas:



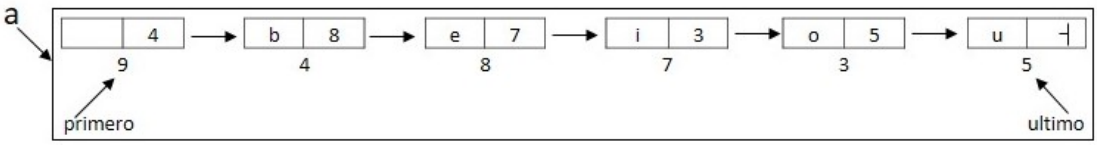
- Listas simplemente ligadas circulares:



- Listas simplemente ligadas circulares con nodo cabeza:



- Listas simplemente ligadas con nodo cabeza:



3.4 TEMA 3 LISTA DOBLEMENTE LIGADAS

Las listas doblemente ligadas **son estructuras que permiten manejar la información de forma dinámica**, donde el espacio donde se guarda la información es un nodo, que es una **dirección de memoria dividida en tres partes**, una parte para guardar la información, otra para la dirección al siguiente nodo y la otra para apuntar al nodo anterior, **con estas listas se pueden realizar las operaciones de insertar, borrar, buscar, ordenar y todo bajo el concepto de memoria dinámica.**

3.4.1 DEFINICIÓN DE CARACTERÍSTICAS

Un nodo doble es un registro que tiene dos campos de liga: uno que llamaremos **Li** (Liga izquierda) y otro que llamaremos **Ld** (Liga derecha). Un dibujo para representar dicho nodo es:

Donde:

Li: apunta hacia el nodo anterior.

Ld: apunta hacia el nodo siguiente.

Con base en esto vamos a definir una clase que llamaremos **nodo Doble:**

```
1.  CLASE nodo Doble
2.      Privado
3.          Objeto dato
4.          nodo Doble Li, Ld
5.      publico
6.      nodo Doble(objeto d)        // constructor
7.      void asigna Dato (objeto d)
8.      void asignaLd (nodoDoble x)
9.      void asignaLi (nodoDoble x)
10.     objeto retornaDato ()
11.     nodo Doble retornaLd ()
12.     nodo Doble retornaLi ()
13. end(Clase)
```

Los algoritmos correspondientes a cada uno de estos métodos son los siguientes:

```
1. nodo Doble(objeto d)          // constructor
2.     dato = d
3.     Ld = null
4.     Li = null
5. end(nodo Doble)
```

```
1. void asigna Dato (objeto d)
2.     dato = d
3. end(Método)
```

```
1. void asignaLd (nodo Doble x)
2.     Ld. = x
3. end(asignaLd)
```

```
1. void asignaLi (nodoDoble x)
2.     Li = x
3. end(asignaLi)
```

```
1. objeto retornaDato ()
2.     return dato
3. end(retornaDato)
```

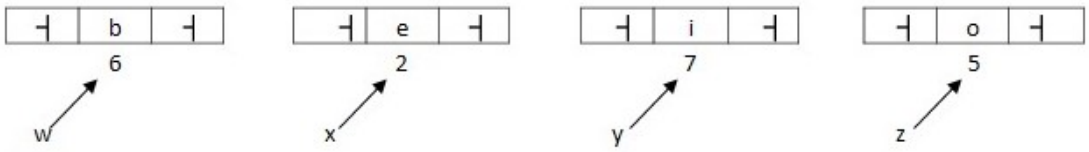
```
1. nodo Doble retorna(Li)
2.     return Li
3. end(retornaLi)
```

1. **nodo Doble** Ld ()
2. return Ld
3. end(retornaLd)

Teniendo definida esta clase, **procedemos a desarrollar un método en el cual hagamos uso de ella y de sus métodos**. Consideremos que se tiene el conjunto de datos b, e, i, o, u, y que nuestro método los leerá en ese orden:

1. nodo Doble w, x, y, z
2. read(d)
3. w = new nodo Doble(d)
4. read(d)
5. x = new nodo Doble(d)
6. read(d)
7. y = new nodo Doble(d)
8. read(d)
9. z = new nodo Doble(d)

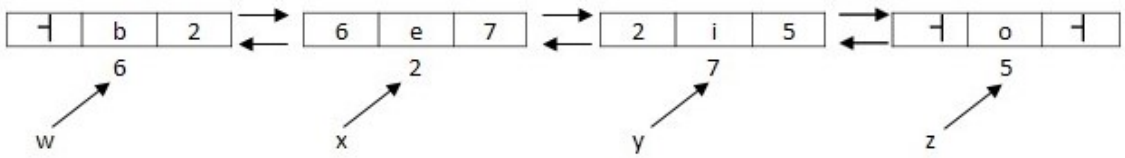
Al ejecutar estas instrucciones el escenario que se obtiene es el siguiente:



Continuemos nuestro algoritmo con estas instrucciones:

10. w. asignaLd(x)
11. x.asignaLi(w)
12. x.asignaLd(y)
13. y. asignaLi(x)
14. y. AsignaLd(z)
15. z. asignaLi(y)

Al ejecutar estas instrucciones nuestro escenario es:



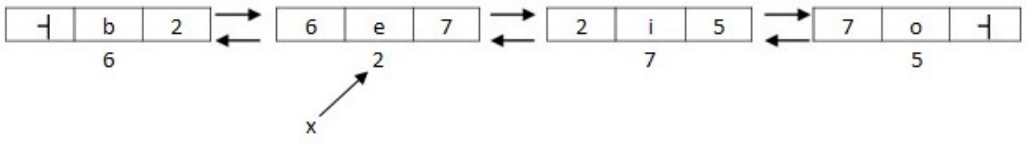
Fíjese que el campo **Ld** del nodo **w** quedo valiendo 2, en virtud de la instrucción 10, lo cual indica que el nodo siguiente a **w** es el 2; el campo **Li** de **x** (el nodo 2) quedo valiendo 6, en virtud de la instrucción 11, lo cual significa

que el nodo anterior a **x** es el 6; el campo **Ld** del nodo **x** quedo valiendo 7, en virtud de la instrucción 12, lo cual quiere decir que el nodo siguiente a **x** es el 7. De una forma similar se han actualizado los campos **Li** y **Ld** de los nodos **y** y **z**. **Asegúrese de entender bien el escenario actual.**

Continuemos ejecutando las siguientes instrucciones:

16. w = null
17. y = null
18. z = null

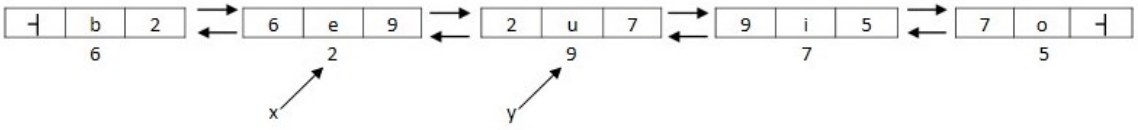
En este momento nuestra situación es:



Ahora ejecutemos estas otras instrucciones:

19. read(d)
20. y = new nodo Doble(d)
21. y. AsignaLd (x. retornaLd ())
22. y. asignaLi(x)
23. y. RetornaLd (). asignaLi(y)
24. x.asignaLd(**y**)

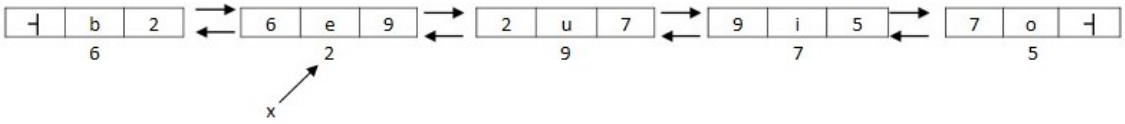
La situación queda así:



Ahora, al ejecutar esta instrucción:

25. y = null

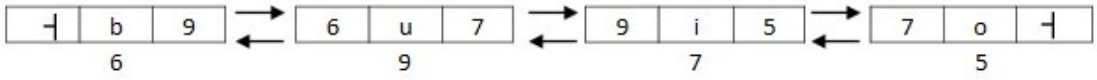
La nueva situación es:



Por último, ejecutemos estas otras instrucciones:

26. x.retornaLi (). asignaLd (x. retornaLd ())
27. x.retornaLd (). asignaLi (x. retornaLi ())

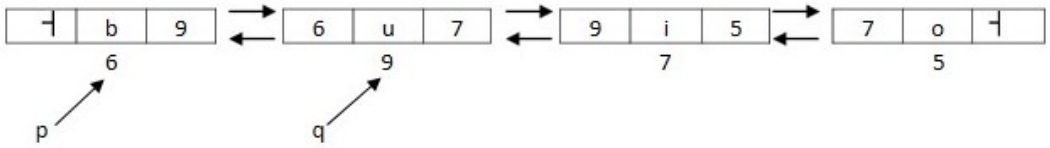
Y nuestros nodos quedan así:



Fíjese que el resultado de ejecutar las instrucciones 26 y 27 fue eliminar el nodo x. unas de instrucciones que harían la misma tarea que las instrucciones 26 y 27 son:

1. p = retornaLi ()
2. q = x. retornaLd ()
3. p. asignaLd(q)
4. q. asignaLi(p)

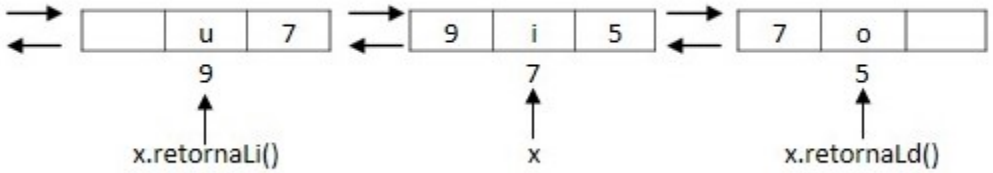
Si lo hubiéramos hecho de ese modo, la lista quedaría así:



Es muy importante entender que las instrucciones 26 y 27 ejecutan la misma tarea que las instrucciones 1, 2, 3 y 4 mostradas a continuación de ellas.

3.4.2 PROPIEDAD FUNDAMENTAL DE LAS LISTAS DOBLEMENTE LIGADAS

Es importante ver aquí lo que se conoce como la **propiedad fundamental de las listas doblemente ligadas** ya que está basada en las características de los campos de liga de los nodos dobles. Para ello, consideremos los siguientes tres registros consecutivos:



El campo de la liga izquierda del nodo **x** vale 9, lo cual indica que el nodo anterior es el nodo 9; por lo tanto, el nodo 9 es el nodo **retornaLi ().**

El campo de liga derecha del nodo **x** vale 5, lo cual indica que el nodo siguiente es el nodo 5; por consiguiente, el nodo 5 es el nodo **x. retornaLd ().**

El nodo **x. retornaLi ()** tiene un campo de liga derecha, el cual vale 7, es decir, el mismo nodo **x.**

El nodo **x. retornaLd ()** tiene un campo de liga izquierda, el cual también vale 7, es decir, el nodo **x.**

Dado lo anterior, se tiene que:

```
x. retornaLi().retornaLd() == x == x.retornaLd().retornaLi()
```

Esta característica se conoce como la "**propiedad fundamental de las listas doblemente ligadas**", y es de suprema importancia entenderla bien con el fin de manipular apropiadamente los objetos de la clase lista doblemente ligada.

3.4.3 EJERCICIOS DE ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Para desarrollar cada uno de esos ejercicios, debes tener muy claro el concepto de nodo y la forma de cómo vas a estructurar el nodo. Recuerda que, para listas simplemente ligadas el nodo se compone de dos partes, una parte de datos y una parte de liga; para las listas doblemente ligadas el nodo se conforma de tres partes, una parte de dato, y dos partes de ligas, una liga que apunta al nodo anterior y una liga que apunta al nodo siguiente.

1. Elabore un método que lea un entero n y que construya una lista simplemente ligada, de a digito por nodo.

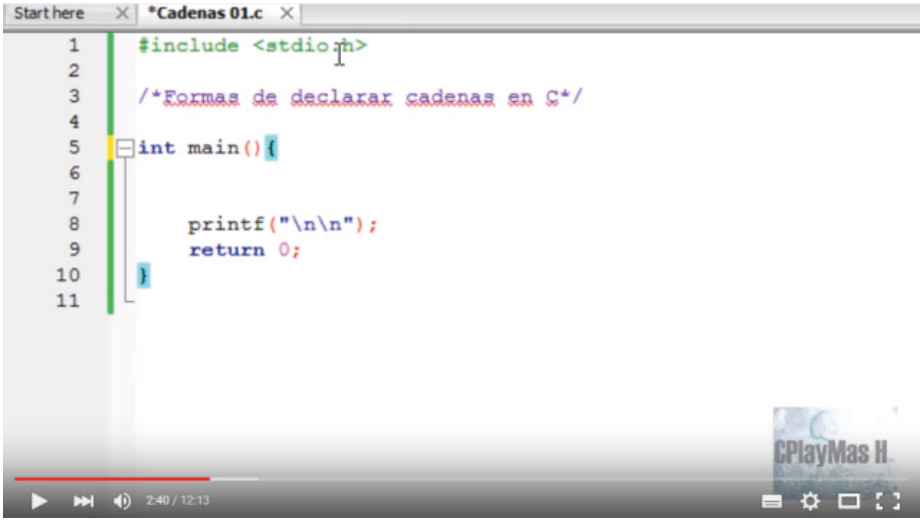
2. Elabore un método que borre de una lista simplemente ligada un dato dado todas las veces que lo encuentre.

3. Se tiene una lista simplemente ligada, con un dato numérico en cada nodo. Elabore un método que determine e imprima el promedio de datos de la lista.

4. Elabore un método que intercambie los registros de una lista doblemente ligada así: el primero con el último, el segundo con el penúltimo, el tercero con el antepenúltimo, y así sucesivamente.

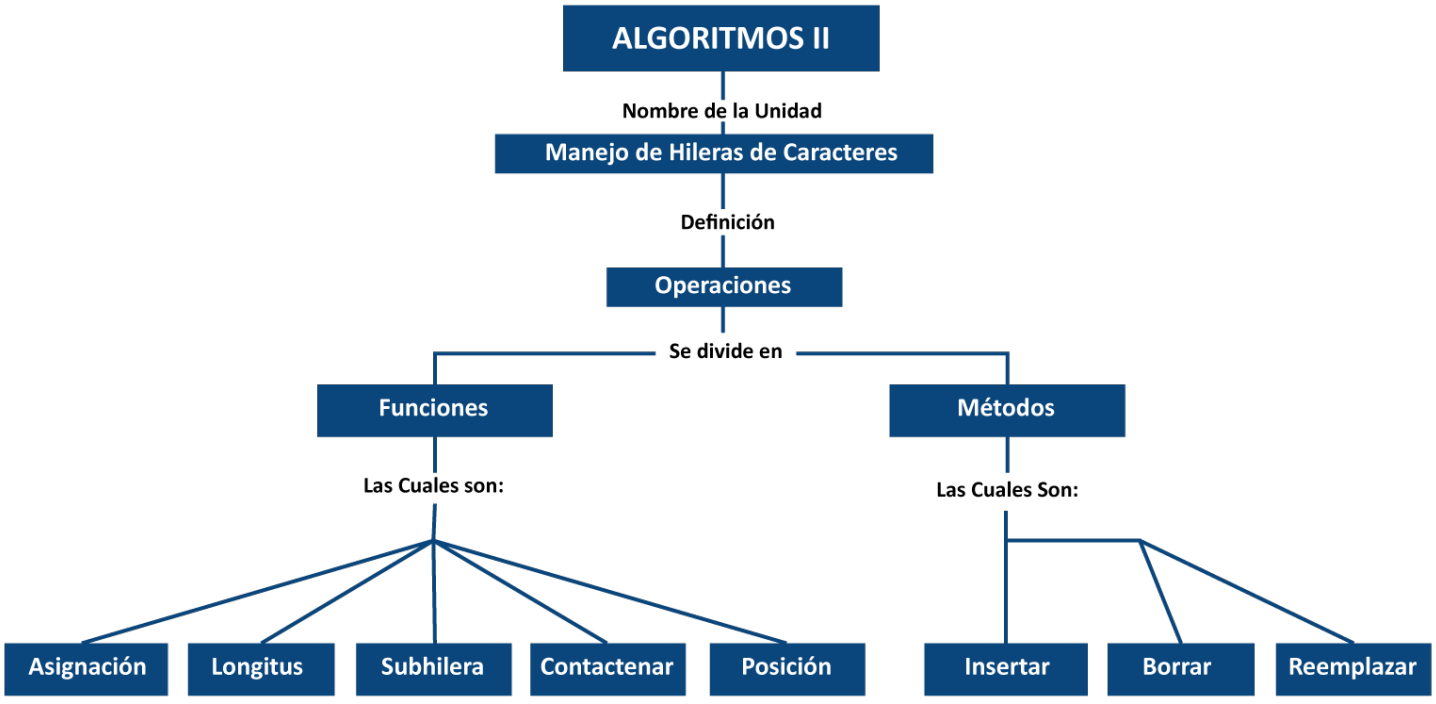
5. Elabore un método que intercambie los registros de una lista doblemente ligada así: el primero con el tercero, el segundo con el cuarto, el quinto con el séptimo, el sexto con el octavo, y así sucesivamente.

4 UNIDAD 3 MANEJO DE HILERAS DE CARACTERES



Programación en C - MANEJO DE CADENAS - Parte 1: Enlace

4.1.1 RELACIÓN DE CONCEPTOS



Hileras: Es un tipo de dato construido con base al tipo primitivo char.

Asignación: Consiste en asignarle a una variable una hilera de caracteres.

Longitud: Consiste en verificar la cantidad de caracteres de la cual se compone la hilera.

Subhilera: Parte que se toma de una hilera.

Concatenar: Consiste en unir una hilera a partir de otra.

Posición: Consiste en determinar, a partir de cual posición de una hilera **x,** se encuentra una hilera **y;** si es que se encuentra.

Insertar: Este método consiste en insertar una hilera **x**, a partir del carácter **i,** de la hilera **y.**

Borrar: Este método consiste, a partir de la posición **i** de la hilera **x,** borra **j** caracteres.

Reemplazar: Este método consiste que a partir de la posición **i** de la hilera **x,** reemplaza **j** caracteres por una copia de la hilera **y.**

4.2 TEMA 1 DEFINICIÓN

Las hileras es un objeto de datos compuestos, **construidos con base en el tipo primitivo char**, su mayor aparición fue cuando empezaron a aparecer los procesadores de textos.

4.3 TEMA 2 OPERACIONES CON HILERAS

Las operaciones que podemos hacer con cadenas son las operaciones de **asignación, longitud, subhilera, concatenar, insertar, borrar, reemplazar y posición en una cadena.** Consideremos ahora las **operaciones básicas** que se pueden ejecutar con hileras. **Definamos una clase hilera con sus operaciones**:

```
1.    CLASE Hilera
2.        Publico
3.            entero longitud ()
4.            hilera subHilera (entero i, entero j)
5.            hilera concat (hilera s)
6.            void inserte (hileras, entero j)
7.            void borre (entero i, entero j)
8.            void replace (entero i, entero j, hilera s)
9.            entero posición (hilera s)
```

10. end(Clase)

4.3.1 ASIGNACIÓN

En el lado izquierdo tendremos el nombre de la variable a la cual se le asigna una hilera dada. En el lado derecho se tendrá una hilera o una variable tipo hilera. Por ejemplo, si **S, T, U y V** son variables tipo hilera podemos tener las siguientes instrucciones:

S = "abc"

T = "def."

U = T

V = ""

En el primer ejemplo la variable **S** contendrá la hilera "abc".

En el segundo ejemplo la variable **T** contendrá la hilera "def".

En el tercer ejemplo la variable **U** contendrá la hilera "def".

En el cuarto ejemplo la variable **V** contendrá la hilera vacía.

4.3.2 FUNCIÓN LONGITUD

Forma general: longitud(S)

Esta función retorna el número de caracteres que tiene la hilera **S**. por ejemplo, si tenemos S = "mazamorra" y ejecutamos la instrucción:

n = S. longitud ()

La variable **n** quedara valiendo 9.

4.3.3 FUNCIÓN SUBHILERA

Forma general: T = S. subHilera (i, j), con $1 \leq i \leq i + j - 1 \leq n$

Siendo n la longitud de la hilera S.

Esta función, a partir de la posición i de la hilera S, extrae j caracteres creando una nueva hilera y dejando intacta la hilera original S. por ejemplo, si tenemos S = "mazamorra" y ejecutamos la instrucción:

T = S. subHilera (4, 5)

La variable T quedara valiendo "amorr".

4.3.4 FUNCIÓN CONCATENAR

Forma general: U = S. Concat(T)

Esta función crea una copia de la hilera S y a continuación agrega una copia de la hilera T. las hileras S y T permanecen intactas. Por ejemplo, si S = "nacio" y T "nal" y ejecutamos la instrucción:

U = S. concat(T)

La variable U quedara conteniendo la hilera "nacional".

4.3.5 MÉTODO INSERTAR

Forma general: S. inserte (T, i)

Este método inserta la hilera T a partir del carácter de la posición **i** de la hilera S. por ejemplo, si tenemos S = "nal" y T = "ciona" y ejecutamos la instrucción:

S. inserte (T, 3)

La hilera S quedara valiendo "nacional".

Es necesario resaltar que en este caso la hilera S queda modificada mientras que la hilera T permanece intacta.

4.3.6 MÉTODO BORRAR

Forma general: S. borre (i, j)

Este método, a partir de la posición **i** de la hilera S borra **j** caracteres. Por ejemplo, si S = "amotinar" y ejecutamos la instrucción:

S. borre (4, 4)

La hilera S quedara valiendo “amor”.

Es bueno anotar también que la hilera S queda modificada.

4.3.7 MÉTODO REEMPLAZAR

Forma general: S. replace (i, j, T)

Este método, a partir de la posición **i** de la hilera S reemplaza **j** caracteres por una copia de la hilera T. por ejemplo, si la hilera S = “abcdf” y la hilera T = “xyz” y ejecutamos la instrucción:

S. replace (3, 2, T)

La hilera S quedara valiendo “abxyzef”.

Este método modifica la hilera S, mientras que la hilera T permanece intacta.

4.3.8 FUNCIÓN POSICIÓN

Forma general: m = S. Posición(T)

Esta función determina a partir de cual posición de la hilera S se encuentra la hilera T, si es que se encuentra la hilera T en la hilera S. en caso de no encontrar la hilera T en la hilera S retornara cero. Por ejemplo, si S = “mazamorra” y T = “amor” y ejecutamos la instrucción:

m = posición(T)

La variable **m** quedara valiendo 4.

Las operaciones aquí descritas nos proporcionan las facilidades para manipular hileras en cualquier lenguaje de programación. **Es importante agregar que lenguajes de programación como C++ y java traen definida la clase hilera (String) en la cual se implementa una gran cantidad de variaciones correspondientes a estas operaciones.**

4.4 TEMA 3 EJERCICIOS DE APLICACIÓN

Con las operaciones mencionadas en el numeral anterior, se pueden realizar muchos ejemplos de aplicación, por ejemplo: cuando se tiene un texto y se solicita determinar cuál es la frecuencia de cada una de las letras del alfabeto español, es decir, cuantas veces aparece la letra “p”, cuantas veces aparece la letra “o” y así sucesivamente.

4.4.1 APLICACIÓN DE LA CLASE HILERA

Pasemos a considerar ejemplos de aplicación con las funciones anteriormente definidas.

Empecemos considerando un ejercicio en el cual nos solicitan determinar cuál es la frecuencia de cada una de las letras del alfabeto español en un determinado texto dado, es decir, cuantas veces se aparece la "a" cuantas veces aparece la "b" y así sucesivamente.

Llamemos **texto** la variable en la cual hay que hacer dicha evaluación. Para desarrollar dicho método utilizaremos una variable llamada **alfabeto**, que es una variable tipo hilera y que definiremos así:

alfabeto = "abcdefghijklmnñopqrstuvwxyz"

Utilizaremos un vector de 27 posiciones, que llamaremos **frecuencia**, en cada una de las cuales guardaremos el número de veces que se encuentre alguna letra del alfabeto definido. Es decir, a la letra "a" le corresponde la posición 1 del vector de frecuencias, a la letra "b" la posición 2, a la letra "c" la posición 3 y así sucesivamente.

Nuestro método consistirá en recorrer los caracteres de la variable **texto**, buscando cada carácter en la variable **alfabeto** utilizando la función **posición**; cuando la encuentre sumaremos 1 a la posición correspondiente a esa letra en el vector de **frecuencia**.

Nuestro método es:

```
1.  void frecuencia Letras (hilera alfabeto)
2.      entero m, i, j, n
3.      hilera car
4.      m = alfabeto. longitude ()
5.      for (i = 1; i <= m; i++) do
6.          frecuencia[i] =0
7.      end(for)
8.      n = longitud ()
9.      for (i = 1; 1 <= n; i++) do
10.         car = subHilera (i, 1)
11.         j = alfabeto. posicion(car)
12.         if (j!= 0) then
13.             frecuencia[j] = frecuencia[j] + 1
14.         end(if)
15.     end(for)
16.     for (i = 1; i <= m; i++) do
17.         letra = alfabeto. SubHilera (i, 1)
```

```
18.      write (letra, frecuencia[i])
19.    end(for)
20. end(Método)
```

Consideremos ahora un método en el cual nos interesa determinar la frecuencia de cada palabra que aparezca en un texto. Para desarrollar dicho método debemos, **primero que todo**, identificar cada una de las palabras del texto. Para lograr esta identificación hay que definir cuáles caracteres se utilizan como separadores de palabras. Estos caracteres pueden ser cualquier símbolo diferente de letra, es decir, la coma, el punto, el punto y coma, los dos puntos, el espacio en blanco, etc.

Nuestro método manejara las siguientes variables:

VARIABLES	CARACTERÍSTICAS
TEXTO	Variable en la cual **se halla almacenado** el texto a procesar.
PALABRAS	Variables **tipo vector** en la cual almacenaremos cada palabra que se identifique en el texto.
FRECUENCIA	Variable **tipo vector** en la cual almacenaremos **el número de veces** que se encuentre cada palabra del texto. Una palabra que se encuentre en la posición **i** del vector **palabra**, en la posición **i** del vector **frecuencia** se hallara el número de veces que se ha encontrado.
PALABRAHALLADA	Variable en la cual almacenaremos cada palabra que se **identifique** en el texto.
K	Variable para contar **cuantas palabras diferentes** hay en el texto. La inicializamos en 1. El total de las palabras diferentes encontradas será **k – 1.**
N	Variable que **guarda el número de caracteres** en el texto.
ALFABETO	Es una **hilera enviada como parámetros**, la cual contiene los **símbolos** con los cuales se construyen las palabras.

método es el siguiente:

```
1.  void frecuencia Palabras (hilera alfabeto)
2.      entero i, k, j, p, n, frecuencia [1000]
3.      hilera car, palabras [1000]
4.      k = 1
5.      for (i = 1; i <= 1000; i++) do
6.          frecuencia[i] = 0
7.      end(for)
8.      n = longitud ()
9.      i = 1
10.     while i <= n do
11.         car = subHilera (i, 1)
12.         p = alfabeto. posicion(car)
13.         while (i < n and p == 0) do
14.             i = i + 1
15.             car = subHilera (i, 1)
16.             p = alfabeto. posicion(car)
17.         end(while)
18.         j = i
19.         while (i < n and p != 0) do
20.             i = i + 1
21.             car = subHilera (i, 1)
22.             p = alfabeto. posicion(car)
23.         end(while)
24.         palabra Hallada = subHilera (j, i - j)
25.         palabras[k] = palabra Hallada
26.         j = 1
27.         while (palabras[j] != palabra Hallada) do
28.             j = j + 1
29.         end(while)
30.         if (j == k) then
31.             frecuencia[k] = 1
32.             k = k + 1
```

```
33.         else
34.             frecuencia[j] = frecuencia[j] + 1
35.         end(if)
36.     end(while)
37.     k = k – 1
38.     for (i = 1; i <= k; i++) do
39.         write(palabras[i], frecuencia[i])
40.     end(for)
41. end(Método)
```

En la instrucción 2 y 3 **se definen las variables** con las cuales se va a trabajar.

En las instrucciones 5 a 7 se **inicializan los contadores** de palabras en 0.

En las instrucciones 8 se determina la **longitud del texto** a analizar **(el que invoco el método).**

En la instrucción 9 se **inicializa la variable i en 1**. Utilizamos la variable i para recorrer el texto, carácter por carácter, e ir identificando las diferentes palabras en él.

Las instrucciones 10 a 36 conforman el **ciclo principal del método**.

En las instrucciones 11 a 17 se **omiten** los caracteres que no conforman una palabra valida. Los caracteres que conforman una palabra valida son los que pertenecen al alfabeto. Cada carácter del texto se busca en **la hilera alfabeto** (instrucciones 12 y 16): si no lo encuentra significa que no conforma palabra y por lo tanto se desecha. Del ciclo de las instrucciones 13 a 17 se sale cuando encuentre un carácter que pertenezca al alfabeto. Esto significa que a partir de esa posición comienza una palabra. Dicha posición la guardaremos en una variable que llamamos **j** (instrucción 18).

En el ciclo de las instrucciones 19 a 23 se determinan hasta cual posición llega una palabra de este ciclo se sale cuando encuentre un carácter que no pertenezca al alfabeto; por consiguiente, la palabra será **la subhilera desde la posición j hasta la posición i – 1**, la cual extrae con la instrucción 24.

En la instrucción 25 **se almacena la palabra hallada** en la posición **k** del vector palabras. Con el fin de determinar si la palabra almacenada en la posición **k** del vector es primera vez que aparece o ya estaba, la buscamos en el vector de palabras (instrucciones 26 a 29) con la certeza de que la encontraremos: si se encuentra en la posición **k** significa que es la primera vez que aparece y por lo tanto le asignamos 1 al contador de esa palabra (instrucción 31) e incrementamos la variable **k** en 1; si encontró la palabra en una posición diferente a la posición **k** significa que dicha palabra ya se había encontrado y por lo tanto incrementamos su respectivo contador en 1 (instrucciones 34).

Por último, en las instrucciones 37 a 40 se describen las diferentes palabras encontradas con su respectivo contador.

Planteemos ahora un logaritmo que considero interesante: **reemplazar una palabra por otra en un texto dado.**

Utilizaremos tres variables:

VARIABLES	CARACTERÍSTICAS
TEXTO	Variable que contiene el **texto** donde hay que hacer el **reemplazo**.
VIEJA	Variable que contiene **la hilera** que hay que **reemplazar.**
NUEVA	Variable que contiene la **hilera** que **reemplazara la hilera vieja**.

Para entender el desarrollo de este método debemos entender lo siguiente: si tenemos un texto S = "aabcdabcde" y T = "abc" y ejecutamos la instrucción

m = S. Posición(T)

La variable m queda valiendo 2. Esto significa que a partir de la posición 2 de la hilera S se encontró la hilera T, es decir, nuestro método retorna el sitio donde encuentra la primera ocurrencia de la hilera que se está buscando. Fíjese que en nuestra hilera S la hilera "abc" se halla en la posición 2 y en la 6.

Si ejecutamos la siguiente instrucción (fíjese que se está buscando la hilera T a partir de la posición 5 de S):

m = S. subHilera (5, 6). posición(T) (formula 1)

La variable m también queda valiendo 2. ¿Por qué? La razón es que la hilera T la buscara en la hilera S. subHilera (5, 6), la cual es: "babcde".

La pregunta es: ¿está realmente la segunda ocurrencia de la hilera T en la posición 2 de S? la respuesta es no.

Si planteamos una utilización de la función posición como en la fórmula 1 y queremos determinar en cual posición de S comienza la subhilera T, al resultado obtenido habrá que sumarle i – 1, siendo i el primer parámetro de la función subHilera. En nuestro caso i = 5.

Llamemos posini la variable a partir de la cual se inicia la búsqueda de una hilera en una subhilera obtenida con la función subhilera.

Nuestro método es:

```
1. void reemplazarViejaPorNueva(hilera vieja, hilera nueva)
2.     v = vieja. Longitud()
```

```
3.      n = nueva. Longitud ()
4.      t = longitud ()
5.      posini = 1
6.      sw = 1
7.      while (sw == 1) do
8.          p = subHilera (posini, t – posini + 1)
9.          posvieja = p. posición(vieja)
10.         if (posvieja > 0) then
11.             posreal = posvieja + posini – 1
12.             replace (posreal, v, nueva)
13.             posini = posreal + n + 1
14.             t = longitud ()
15.             if (posini > t + v 1) then
16.                 sw = 0
17.             end(if)
18.         else
19.             sw = 0
20.         end(if)
21.     end(while)
22. end(Método)
```

4.4.2 EJERCICIOS ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Para desarrollar estos ejercicios debe tener en cuenta, todas las funciones y métodos vistos en el tema. **Recuerda que**, puedes utilizar una estructura tipo vector o una estructura lista ligada; si lo requiere, para dar solución a los problemas.

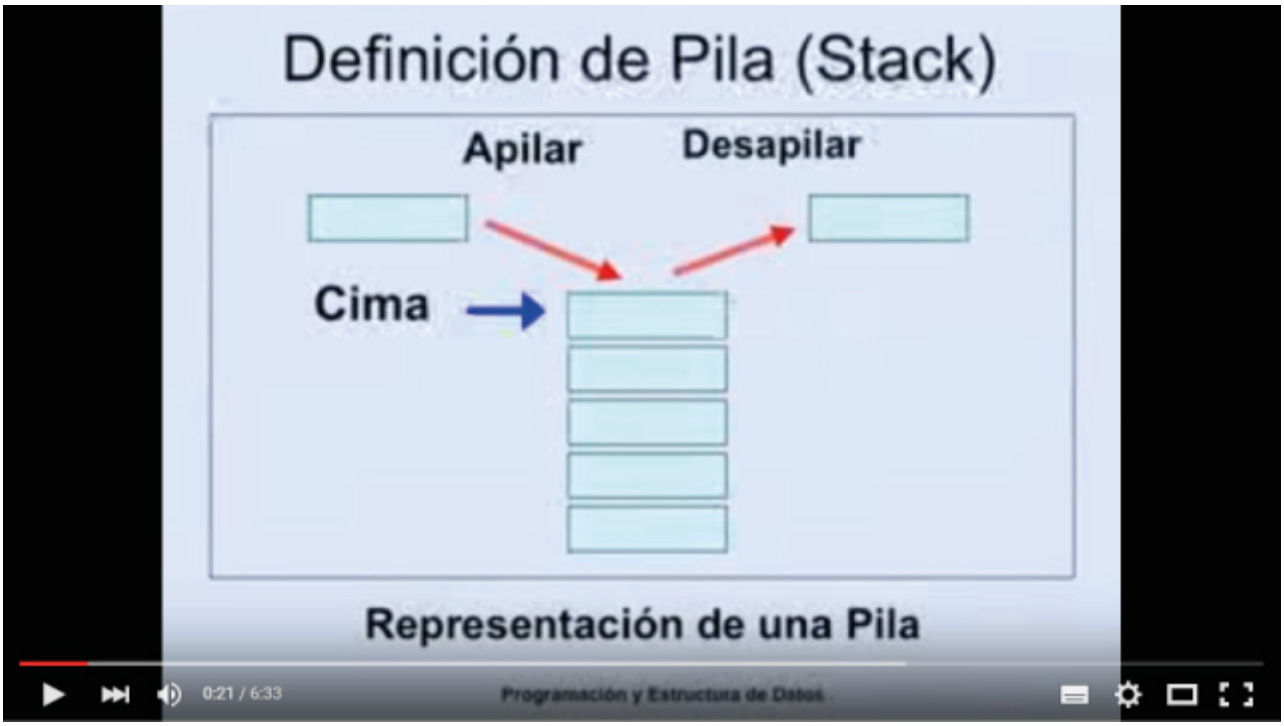
1. Elabore un método para determinar si una hilera dada constituye un palíndromo o no. Un palíndromo es una hilera que se lee igual de izquierda a derecha que de derecha a izquierda; por ejemplo: radar, reconocer, Abba. Su método debe retornar verdadero si cumple la condición, falso de lo contrario.

2. Elabore un método que determine si una palabra tiene más vocales que consonantes o no. Su método debe retornar verdadero si cumple la condición; de lo contrario, debe retornar falso.

3. Elabore un método que invierta todos los caracteres de una hilera dada. Por ejemplo, si la hilera es “amor”, al ejecutar su método debe quedar la hilera “roma”.

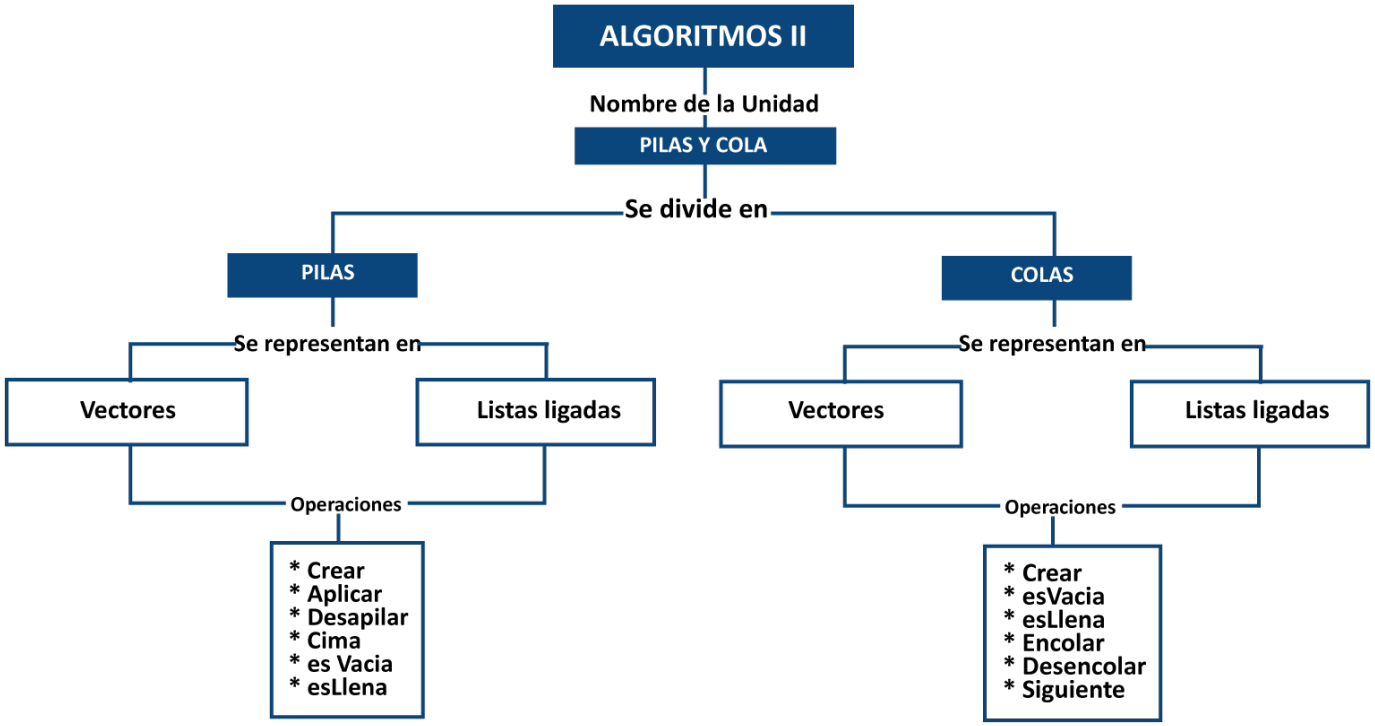
4. Elabore un método que determine si una palabra tiene las cincos vocales o no. Su método debe retornar verdadero si cumple la condición; de lo contrario, debe retornar falso.

5 UNIDAD 4 PILAS Y COLAS



Manipulación de información. Pilas y colas: Enlace

5.1.1 RELACIÓN DE CONCEPTOS



Pilas: Son estructuras de datos que se manejan de forma estática, donde se realiza las operaciones de apilar y desapilar, manejando el concepto de último en llegar primero en salir.

Vector: En programación se denomina matriz, vector o formación (en inglés *Array*) a una zona de almacenamiento continuo que contiene una serie de elementos del mismo tipo, los elementos de la matriz. Desde el punto de vista lógico una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).
Tomado de: https://es.wikipedia.org/wiki/Vector_(inform%C3%A1tica)

Listas ligadas: Las listas ligadas son una mejor alternativa para añadir información a una base de datos; en lugar de utilizar arreglos que son estáticos pues hay que reservar memoria a la hora de programarlos, contrario a las listas ligadas que en un principio no ocupan memoria y se va reservando (creando) conforme lo va requiriendo nuestro programa. La memoria que ya no se utiliza se puede destruir y así ahorramos memoria en nuestros programas.
Tomado de: http://sergioaj.blogspot.com.co/2011/01/listas-ligadas.html

Crear: Crea una pila vacía.

Apilar: Incluye el dato d en la pila.

Desapilar: Elimina el último elemento de la pila y deja una nueva pila, la cual queda con un elemento menos.

Cima: Retorna el dato que esta de último en la pila, sin eliminarlo.

es Vacía: Retorna verdadero si la pila está vacía: de lo contrario; retorna falso.

es Llena: Retorna verdadero si la pila está llena; de lo contrario; retorna falso.

Cola: Son estructuras de almacenamiento de datos, donde se maneja de forma estática, donde la operación de inserción se hace al final y las de borrado se hace al principio, en otras palabras, el primero que llega es el primero en salir, y el último en llegar será el último en salir

Crear: Crea una cola vacía.

es Vacía: Retorna verdadero si la cola está vacía; de lo contrario, retorna falso.

es Llena: Retorna verdadero si la cola está llena; de lo contrario, retorna falso.

Encolar: Inserta un dato d al final de la cola.

Desencolar: Remueve el primer elemento de la cola.

Siguiente: Retorna el dato que se halla de primero en la cola.

5.2 TEMA 1 DEFINICIÓN

Son **estructuras de datos** que **no tienen representación propia** a nivel de programación, para ello se apoyan en los **vectores y en listas ligadas**, permitiendo así manejar operaciones sobre ellas.

5.3 TEMA 2 PILAS CON VECTORES Y LISTAS

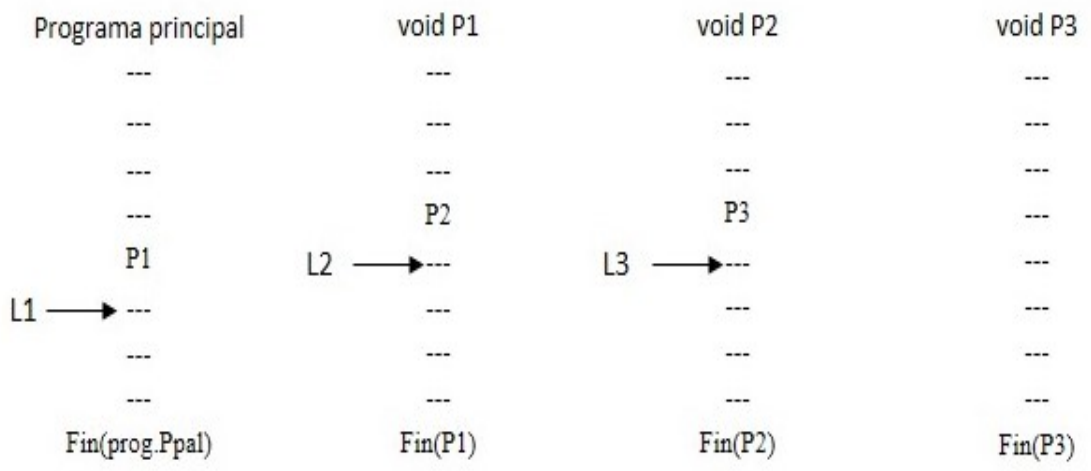
Las pilas como vectores **son estructuras de datos que se manejan de forma estática**, donde se realiza las operaciones de **apilar y desapilar**, manejando el concepto de último en llegar primero en salir. Las pilas como lista son estructuras de datos, pero se maneja de forma **dinámica en la memoria.**

5.3.1 DEFINICIÓN

Una pila **es una lista ordenada** en la cual todas las operaciones **(inserción y borrado)** se efectúan en un solo extremo llamado **tope.** Es una estructura **LIFO** (**L**ast **I**nput **F**irst **O**utput), que son las iniciales de las palabras en inglés **"ultimo en entrar primero en salir"**, debido a que los datos almacenados en ella se retiran en orden inverso al que fueron entrados.

Un ejemplo clásico de aplicación de pilas en computadores se representa en el proceso de llamadas a métodos y sus retornos.

Supongamos que tenemos un programa principal y tres métodos, así:



Cuando se ejecuta el **programa principal**, se hace una llamada al **método P1**, es decir, ocurre una interrupción en la ejecución del **programa principal**. Antes de iniciar la ejecución de este método se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del **programa principal** cuando termine de ejecutar el método; llamamos **L1** esta dirección. Cuando ejecuta el método **P1** existe una llamada al **método P2**, hay una nueva interrupción, pero antes de ejecutar **P2** se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del **método P1**, cuando termine de ejecutar el **método P2**; llamamos **L2** esta dirección.

Hasta el momento hay guardados **dos direcciones de retorno:**

L1, L2

Cuando ejecuta el **método P2** hay llamada a un **método P3**, lo cual implica una nueva interrupción y, por ende, guardar una dirección de retorno al **método P2**, la cual llamamos **L3**.

Tenemos entonces **tres direcciones** guardadas, así:

L1, L2, L3

Al terminar la ejecución del **método P3** retorna a continuar ejecutando en la última dirección que guardo, es decir, extrae la **dirección L3** y regresa a continuar ejecutando el **método P2** en dicha instrucción. Los datos guardados ya son:

L1, L2

Al terminar el **método P2** extrae la última dirección que tiene guardada, en este caso **L2**, y retorna a esta dirección a continuar la ejecución del método **P1.**

En este momento los datos guardados son:

L1

Al terminar la ejecución del **método P1** retornara a la dirección que tiene guardada, o sea a **L1.**

Obsérvese que los datos fueron procesados en orden inverso al que fueron almacenados, es decir, último en entrar primero en salir. En esta forma de procesamiento la que define **una estructura PILA.**

Veamos cuales son las operaciones que definiremos para la clase pila.

OPERACIONES	CARACTERÍSTICAS
CREAR	Crea una **pila vacía.**
APILAR	**Incluye** el dato **d** en la pila.
DESAPILAR	**Elimina el último elemento de la pila** y deja una nueva pila, la cual queda con un elemento menos.
CIMA	**Retorna el dato que esta de último en la pila, sin eliminarlo.**
ESVACIA	Retorna verdadero **si la pila está vacía**: de lo contrario; **retorna falso.**
ESLLENA	Retorna verdadero **si la pila está llena**; de lo contrario; **retorna falso.**

5.3.2 REPRESENTACIÓN DE PILAS EN UN VECTOR

La forma más simple es utilizar **un arreglo de una dimensión y una variable**, que llamaremos **tope,** que indique la posición del arreglo en la cual se halla el último elemento de la pila. Por consiguiente, vamos a definir la clase pila derivada de la clase vector. La variable **m** de nuestra clase vector funciona coma la variable **tope.** Definamos entonces la clase pila.

```
1.  CLASE Pila
2.      Privado:
3.          Object V []
4.          Entero tope, n
5.      Publico:
6.          pila (entero m)          //constructor
7.          boolean es Vacía ()
8.          boolean es Llena ()
9.          void apilar (objeto d)
10.         objet desapilar ()
11.         void desapilar (entero i)
12.         objeto tope ()
13. end(Clase)
```

Como hemos definido **la clase pila derivada de la clase vector,** veamos como son los algoritmos correspondientes a estos métodos:

```
1.  pila (entero m)          //constructor
2.      V = new Array[m]
3.      n = m
4.      Tope = 0
5.  end(pila)
```

```
1. boolean es Vacía ()
2.       return tope == 0
3. end (es Vacía)
```

```
1. boolean es Llena ()
2.       return tope == n
3. end (es Llena)
```

```
1. void Apilar (objeto d)
2.      if (es Llena ()) then
3.         write (“pila llena”)
4.         return
5.      end(if)
6.      tope = tope + 1
7.      V[tope] = d
8. end(Método)
```

Apilar consiste simplemente en sumarle 1 a tope y llevar el dato a esa posición del vector:

```
1. objeto desapilar()
2.        if (es Vacía ()) then
3.           write (“pila vacía”)
4.           return null
5.        end(if)
6.        d = V[tope]
7.        tope = tope -1
8.        return d
9. end(desapilar)
```

Estrictamente hablando, **desapilar consiste simplemente en eliminar el dato que se halla en el tope de la pila.** Sin embargo, es usual eliminarlo y retornarlo. Eso es lo que hace nuestro anterior método para desapilar. **El proceso de eliminación consiste simplemente en restarle 1 a tope.** Definamos, en forma polimórfica, otro método para

desapilar. Este nuevo método tendrá un parámetro **i,** el cual indicara cuantos elementos se deben eliminar de la pila:

```
1. void Desapilar (entero i)
2.       if ((tope – 1) >= 0) then
3.             tope = tope – i
4.       else
5.             write (“no se pueden eliminar tantos elementos”)
6.       end(if)
7. end(Método)
```

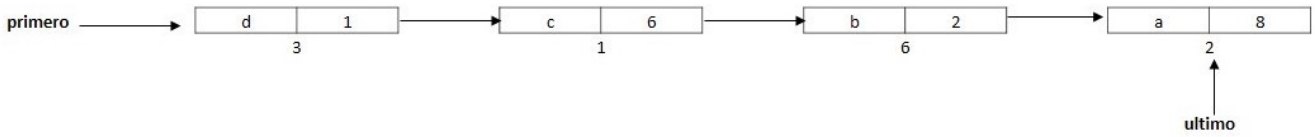
```
1. Objeto cima ()
2.        if (es Vacía ()) then
3.              write (“pila vacía”)
4.              return null
5.        end(if)
6.        return V[tope]
7. end(cima)
```

5.3.3 REPRESENTACIÓN DE PILAS COMO LISTAS LIGADAS

Para representar una pila como listas ligadas basta definir la clase pila derivada de la clase **LSL**. Nuevamente, como estamos definiendo la clase pila derivada de la clase. **LSL**, podremos hacer uso de todos los métodos que hemos definido para esta clase **LSL.** A modo de ejemplo, representemos como listas ligadas la siguiente pila:

Pila

Dibujémosla de la forma como solemos hacerlo con las listas ligadas:



Como hemos dicho **las operaciones sobre una pila son apilar y desapilar**:

- **apilar:** consiste **en insertar un registro al principio de una lista ligada**
- **desapilar:** consiste **en eliminar el primer nodo de la lista ligada y retornar el dato que se hallaba en ese nodo.**

Al definir la clase pila derivada de la clase LSL el método para controlar pila llena ya no se utiliza. **El método para la clase pila son:**

```
1. void Apilar (objeto d)
2.         insertar (d, null)
3. end(Método)
```

```
1. Objeto desapilar ()
2.         If (es Vacía ()) then
3.                 Write("pila vacia, no se puede desapilar")
4.                 Return null
5.         End(if)
6.         nodoSimple p
7.         p = primerNodo ()
8.         d = p. retornaDato ()
9.         borrar (p, null)
10.        return d
11. end(desapilar)
```

Fíjese que en los métodos **apilar y desapilar** hemos usado **los métodos insertar y borrar,** los cuales fueron definidos para la clase LSL:

```
1. Objeto tope ()
2.         if (es Vacía ()) then
```

```
3.            write ("pila vacía, no hay elemento para mostrar")
4.            return null
5.        end(if)
6.        nodoSimple p
7.        p = primerNodo ()
8.        return p. retornaDato
9.  end(tope)
```

5.3.4 MANEJO DE DOS PILAS EN UN VECTOR

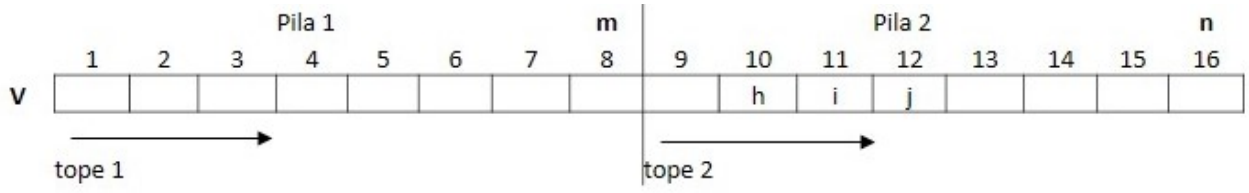
Puesto que es muy frecuente tener que manejar más de una pila en muchas situaciones, veremos **cómo manejar dos pilas en un solo vector.** Si **n** es el número de elementos del vector, dividimos inicialmente el vector en dos partes iguales. Llamamos **m** la variable que apunta hacia el elemento de la mitad.

La primera mitad del vector será para manejar la pila 1, y **la segunda mitad** para manejar la pila 2.

Cada pila requiere una variable tope para conocer en qué posición está el ultimo dado de la pila. Llamemos estas variables **topes 1** y **tope 2** para manejar las pilas 1 y 2, respectivamente.

Consideremos el siguiente ejemplo:

Sea **V** el vector en el cual manejaremos las dos pilas. El valor de **n** es 16. Inicialmente, el valor de **m** es 8 (la mitad de **n**):



La variable **tope1** variara de 1 hasta **m**

La variable **tope2** variara desde **m + 1** hasta **n**

La pila 1 estará vacía cuando **tope1** sea igual a 0

La pila 1 estará llena cuando **tope1** sea igual a **m**

La pila 2 estará vacía cuando **tope2** sea igual a **m**

La pila 2 estará llena cuando **tope2** sea igual a **n**

Si definimos una clase denominada **dos Pilas** cuyos datos privados son el vector **V, m, n**, **tope1** y **tope2**, veamos cómo serán los métodos para manipular dicha clase.

Consideremos primero **un método para apilar un dato en alguna de las dos pilas.** Habrá que especificar en cual pila es que se desea apilar. Para ello utilizaremos una variable llamada **pila**, la cual enviamos como parámetro del método. Si pila es igual a 1 hay que apilar en **pila 1,** y si pila es igual a 2, hay que apilar en **pila 2. Nuestro método será:**

```
1.  void Apilar (entero pila, objeto d)
2.      If (pila == 1) then
3.          If (tope == m) then
4.              pilaLlena(pila)
5.          end(if)
6.          tope1 = tope1 + 1
7.          V[tope] = d
8.      else
9.          If (tope2 == n) then
10.             pilaLlena(pila)
11.         end(if)
12.         tope2 = tope2 + 1
13.         V[tope2] = d
14.     end(if)
15. end(Método)
```

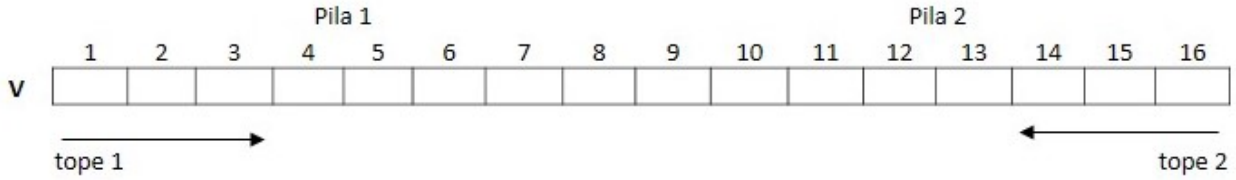
El método **pilaLlena** recibe como parámetro el valor **pila.**

La tarea de **pilaLlena** es: si el parámetro pila es 1 significa que la **pila 1** es la que está llena, y por lo tanto buscara espacio en la **pila 2**; en caso de que esta no esté llena, se moverán los datos de la **pila 2** una posición hacia la derecha, se actualizara **m** y se regresara al método apilar para apilar en la **pila 1.** Si es la **pila 2** la que está llena buscara si hay espacio en la **pila 1,** y en caso de haberlo moverá los datos de la **pila 2** una posición hacia la izquierda, actualizara **m** y regresara a apilar en la **pila 2.** Un método que efectué esta tarea es:

```
1.  void PilaLlena (entero pila)
2.      entero i
3.      if (pila == 1) then
4.          if (tope2 < n) then                  //hay espacio en la pila 2
5.              for (i = tope2; i > m; i--) do
6.                  V [i + 1] =V[i]
7.              end(for)
```

```
8.                tope2 = tope2 + 1
9.                m = m + 1
10.           end(if)
11.      else
12.           if (tope1 < m) then
13.               for (i = m; i < tope2; i++) do
14.                    V [i - 1] = V[i]
15.               end(for)
16.               tope2 = tope2 – 1
17.               m = m – 1
18.           end(if)
19.       end(if)
20.       write (“pilas llenas”)
21.       stop
22. end(Método)
```

Como se podrá observar, la operación de **apilar implica mover datos en el vector** debido a que una pila puede crecer más rápido que la otra. En otras palabras, **este método tiene orden de magnitud lineal,** el cual se considera ineficiente. Una mejor alternativa de diseño es la siguiente:



La pila 1 se llenará de izquierda a derecha y **la pila 2 de** derecha a izquierda. Con este diseño no hay que preocuparse por cual pila crezca más rápido. Las condiciones a controlar son:

La pila 1 estará vacía cuanto **tope1** sea igual a cero

La pila 2 estará vacía cuando **tope2** sea igual a **n + 1** (n =16)

Las pilas estarán llenas cuando **tope1 + 1** sea igual a **tope2**

El **método apilar** para este segundo diseño es:

```
1.  void Apilar (entero pila, objeto d)
2.      if (tope1 + 1 == tope2) then
3.          pilaLlena
4.      end(if)
```

```
5.      if (pila == 1) then
6.          tope1 = tope1 + 1
7.          V[tope] = d
8.      end(if)
9.  end(Método)
```

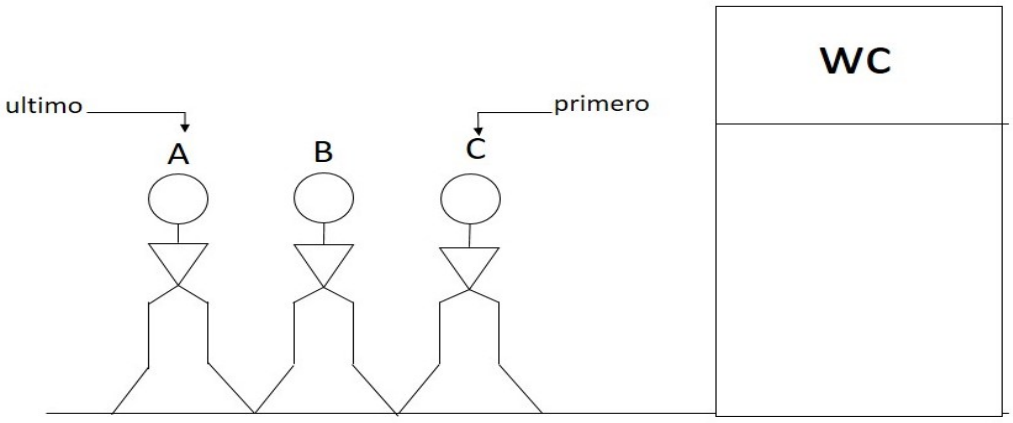
Como podrá observar, con este diseño el proceso para pila llena no aparece; por lo tanto, no hay que mover los datos del vector y nuestro método para apilar tiene orden de magnitud constante

5.4 TEMA 3 COLAS CON VECTORES Y LISTAS

Las colas como vectores son estructuras de almacenamiento de datos, donde se maneja de **forma estática**, donde las operaciones de inserción se hace al final y las de borrado se hace al principio, en otras palabras el primero que llega es el primero en salir, y el último en llegar será el último en salir, de igual forma **se maneja con las listas pero de forma dinámica.**

5.4.1 DEFINICIÓN

Una cola es una lista ordenada en la cual las operaciones de inserción se efectúan en un extremo llamado **último** y las operaciones de borrado se efectúan en el otro extremo llamado **primero**. Es una estructura **FIFO** (**F**irst **I**nput **F**irst **O**utput).



En términos prácticos, es lo que supuestamente se debe hacer para tomar un bus, para comprar las boletas y entrar a cine o para hacer uso de un servicio público. **Las operaciones sobre una cola son:**

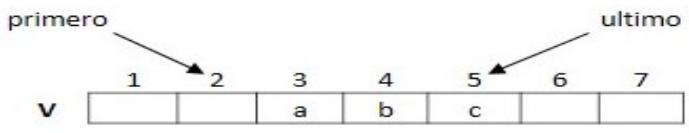
OPERACIONES	CARACTERÍSTICAS
Crear	Crea una **cola vacía.**
es Vacía()	retorna **verdadero si la cola está vacía**; de lo contrario, **retorna falso**
es Llena()	Retorna **verdadero si la cola está llena**; de lo contrario, **retorna falso.**
Encolar(d)	**Inserta** un dato **d** al final de la cola.
Desencolar()	**remueve** el primer elemento de la cola
Siguiente()	**Retorna** el dato que se halla de primero en la cola.

5.4.2 REPRESENTACIÓN DE COLAS EN UN VECTOR, EN FORMA NO CIRCULAR

Para representar colas en esta forma se requiere un vector, que llamaremos **V**, y **dos variables:** una que llamaremos **primero**, la cual indica la posición del vector en la que se halla el primer dato de la cola, y otra, que llamaremos **último**, que **indica la posición en la cual se halla el último dato de la cola.**

En el vector **V** se guardan los datos; primero apuntara hacia la posición anterior en la cual se halla realmente el primer dato de la cola, y **ultimo** apuntara hacia la posición en la que realmente se halla el último dato de la cola.

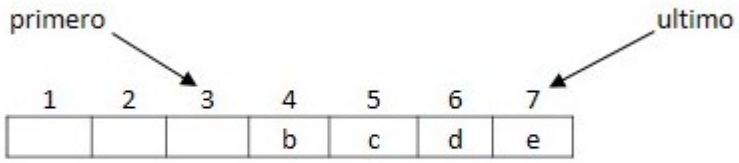
Si tenemos un vector con la siguiente configuración:



La variable **primero** valdrá 2 indicando que el primer elemento de la cola se halla en la posición 3 del vector; la variable **ultimo** valdrá 5 indicando que el último elemento de la cola se halla en la posición 5 del vector.

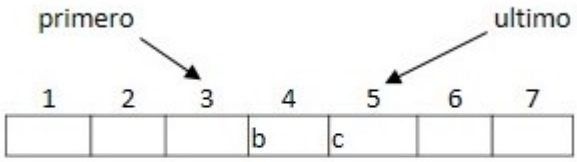
Las operaciones sobre la cola, que son encolar y desencolar, funcionan de la siguiente forma: si se desea encolar basta con incrementar la variable **última** en 1 y llevar a la posición **ultimo** del vector el dato a encolar; si se desea desencolar basta con incrementar en uno la variable **primero** y retornar el dato que se halla en esa posición.

En general, si el vector **V** se ha definido con **n** elementos, cuando la variable **última** sean **n** se podría pensar que la cola está llena:



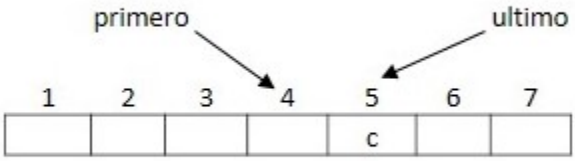
Pero, como se observa en la figura, **el vector tiene espacio al principio**; por consiguiente, podemos pensar en mover los datos hacia el extremo izquierdo y actualizar **primero** y **ultimo** para luego encolar. Es decir, el hecho de que el **ultimo** sea igual a **n** no es suficiente para determinar que la cola está llena. Para que la cola está llena se deben cumplir con dos condiciones: que **primero** sea igual a 0 y **ultimo** sea igual a **n.**

Consideremos ahora operaciones sucesivas de desencole para el ejemplo dado: después de ejecutar la primera operación de desencole los valores de las variables **ultimo** y **primero** y del vector **V** serán los siguientes:



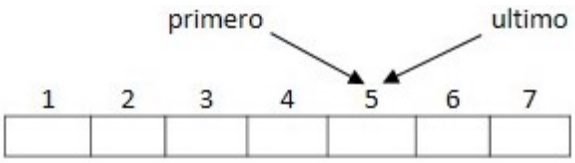
Donde la variable **primero** valdrá 3 y la variable **ultimo** 5

Al desencolar nuevamente, la configuración del vector será:



Donde la variable **primero** vale 4 y la variable **ultimo** 5.

Si desencolamos de nuevo, la configuración del vector será:



Y la variable **primero** vale 5 y la variable **ultimo** 5.

En otras palabras; **primero** es igual a **ultimo** y la cola está vacía, o sea que la condición de la cola vacía será **primero == ultimo**. Teniendo definidas **las condiciones de la cola llena y cola vacía** procedamos a definir la clase cola:

```
1.  CLASE Cola
2.          Privado
3.                  Entero primero, n
4.                  Objeto V[ ]
5.          Publico
6.                  Cola(n)                 //constructor
7.                  boolean es Vacía ()
8.                  boolean es Llena ()
9.                  void encolar (objeto d)
10.                 Objeto desencolar ()
11.                 Objeto siguiente ()
12. end(Clase)
```

```
1.  Cola (entero m)
2.      n = m
3.      primero = ultimo = 0
4.      V = new objeto[n]
5.  end(cola)
```

```
1.  Boolean es Vacía ()
2.       Return primero == ultimo
3.  end (es Vacía)
```

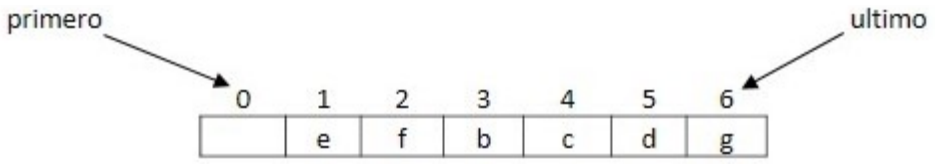
```
1.  Boolean es Llena ()
2.          return primero == 0 and ultimo == n
3.  end (es Llena)
```

```
1.  void encolar (objeto d)
2.          if (es Llena ()) then
3.              write (“cola llena”)
4.              return
5.          end(if)
6.          If (ultimo == n) then
7.              for (i = primero + 1; i <=n; i ++)
```

```
8.                  V [I - primero] = V[i]
9.              end(for)
10.             ultimo = ultimo – primero
11.             primero = 0
12.         end(if)
13.         ultimo = ultimo + 1
14.         V[ultimo] = d
15. end(Método)
```

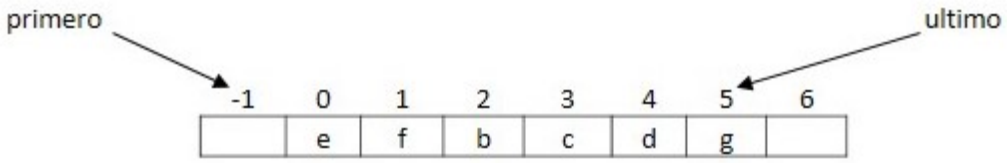
```
1.  objeto desencolar ()
2.          If (es Vacía ()) then
3.              write (“cola vacía”)
4.              return null
5.          end(if)
6.          primero = primero + 1
7.          return V[primero]
8.  end(desencolar)
```

Si consideramos una situación como la siguiente:



n vale 7, **último** vale 6 y **primero** vale 1.

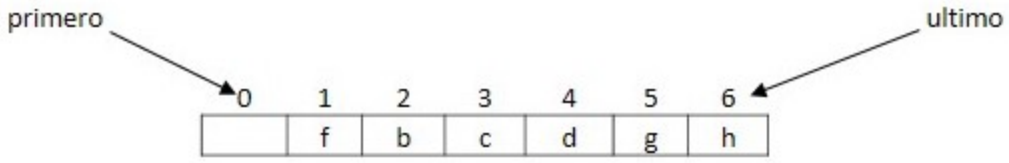
Si se desea encolar el dato **h** y aplicamos el método **encolar** definido, la acción que efectúa dicho método es mover los elementos desde la posición 1 hasta la 6, una posición hacia la izquierda de tal forma que quede espacio en el vector para incluir la **h** en la cola. El vector quedara así:



Con **primer**o valiendo -1 y **ultimo** valiendo 5. De esta forma continua la ejecución del método **encolar** y se incluye el dato **h** en la posición 6 del vector. El vector queda así:

Con **primero** valiendo -1 y **ultimo** valiendo 6.

Si en este momento se desencola, el vector queda así:



Con **primero** valiendo 0 y **ultimo** valiendo 6.

Si la situación anterior se repite sucesivas veces, que sería el peor de los casos, cada vez que se vaya a encolar se tendría que mover **n-1** elementos del vector, lo cual haría ineficiente el manejo de la cola ya que el proceso de encolar tendría orden de magnitud **O(n).** Para obviar este problema y poder efectuar los métodos de **encolar y desencolar** en un tiempo **O (1)** manejaremos el vector circularmente.

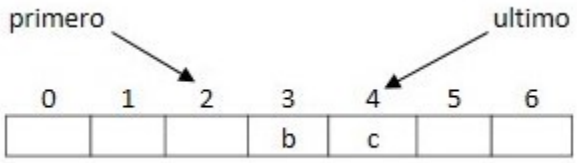
5.4.3 REPRESENTACIÓN DE COLAS CIRCULARES EN UN VECTOR

Para manejar una cola circularmente en un vector se requiere definir un vector de **n** elementos con los subíndices en el rango desde 0 hasta **n-1,** es decir, si el vector tiene 10 elementos los subíndices variaran desde 0 hasta 9, y el incremento de las variables **primero** y **último** se hace utilizando la operación **modulo** (%) de la siguiente manera:

Primero = (primero + 1) % n

Ultimo = (ultimo + 1) % n

Recuerde que la operación **modulo (%)** retorna el residuo de una división entera. Si se tiene una cola en un vector, con la siguiente configuración:



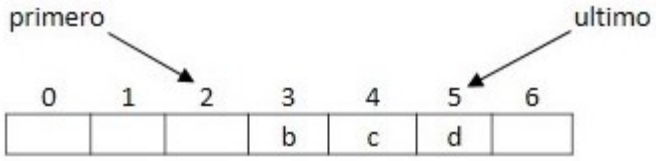
n vale 7, los subíndices varían desde 0 hasta 6**, primero** vale 2 y **ultimo** vale 4.

Si se desea encolar el dato **d** incrementamos la variable **última** utilizando la operación modulo, así:

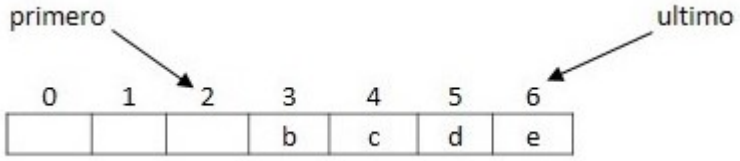
Ultimo = (ultimo + 1) % n

O sea, **ultimo = (4 + 1) % 7**, la cual asignara a **ultimo** el residuo de dividir 5 por 7, que es 5.

El vector queda así:



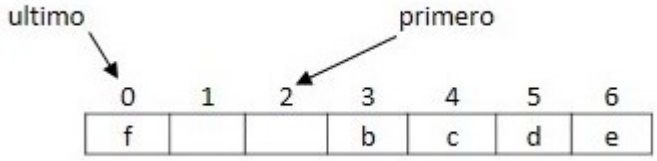
Al encolar el dato e el vector queda así:



Y las variables: **primera** y **última** valdrán 2 y 6 respectivamente. Al encolar de nuevo el dato **f**, aplicamos el operador % y obtenemos el siguiente resultado:

Ultimo = (6 + 1) % 7

El cual es 0, ya que el residuo de dividir 7 por 7 es 0. Por consiguiente, la posición del vector a la cual se llevará el dato **f** es la posición 0. El vector quedara con la siguiente conformación:



Con ultimo valiendo 0 y primero 2. De esta forma hemos podido encolar en el vector sin necesidad de mover elementos en él.

Los métodos para encolar y desencolar quedan así:

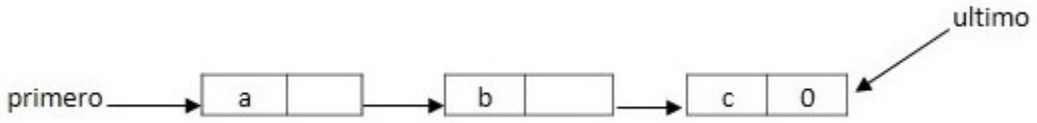
```
1. void encolar (objeto d)
2.     Ultimo = (ultimo + 1) % n
3.     if (ultimo == primero) then
4.         cola Llena ()
5.     end(if)
6.     V[ultimo] = d
7. end(Método)
```

```
1. objeto desencolar ()
2.         if (primero == ultimo) then
3.                 cola Vacía
4.         end(if)
5.         primero = (primero + 1) % n
6.         d = V[primero]
7. end(desencolar)
```

Es importante notar que **la condición de cola llena y cola vacía es la misma**, con la diferencia de que en el método **encolar** se chequea la condición después de incrementar **último**. Si la condición resulta verdadera invoca el método **cola Llena**, cuya función será dejar **último** con el valor que tenía antes de incrementarla y detener el proceso, ya que no se puede encolar. Por consiguiente, habrá una posición del vector que no se utilizará, pero que facilita el manejo de las condiciones de **cola vacía y cola llena.**

5.4.4 REPRESENTACIÓN DE COLAS COMO LISTAS LIGADAS

Definimos la cola como derivada de la clase LSL:



Encolar: consiste **en insertar un registro al final de una lista ligada**:


```
1. void encolar (objeto d)
2.         Insertar (d, ultimoNodo ())
3. end(Método)
```

Desencolar: consiste en **borrar el primer registro de una lista ligada** (exacto a desapilar). Habrá que controlar si la cola está o no vacía

```
1.  Objeto desencolar (primero, ultimo, d)
2.      if (es Vacía ()) then
3.          write ("cola vacia, no se puede desencolar")
4.          Return null
5.      end(if)
6.      nodoSimple p
7.      p = primerNodo ()
8.      d = p. retornaDato ()
9.      borrar (p, null)
10.  return d
11. end(desencolar)
```

5.5 EJERCICIOS DE ENTRENAMIENTO

Pautas para desarrollar los siguientes ejercicios: Para desarrollar estos ejercicios, **debes tener claro** las operaciones que se pueden realizar en una pila, y las operaciones que se pueden realizar en una cola.

1. Escriba un método que inserte un elemento en una pila. Considere todos los casos que puedan presentarse.
2. Escriba un método que elimine un elemento de una pila. Considere todos los casos que puedan presentarse.
3. Escriba un método que inserte un elemento en una cola circular. Considere todos los casos que puedan presentarse.
4. Escriba un método que elimine un elemento de una cola circular. Considere todos los casos que puedan presentarse.

Utilice una estructura de cola para simular el movimiento de clientes en una cola de espera de un banco (puede auxiliarse con los métodos utilizados que están planteados en los ejercicios 3 y 4).

6 PISTAS DE APRENDIZAJE

Recuerde que, a la hora de intercambiar datos dentro de un vector de registro, primero debe guardar el contenido en una celda a dentro de una variable auxiliar, para no perder lo que hay en a; luego a la celda a le lleva lo que tiene la celda b; y por último, a la celda b le lleva lo que guardó en la variable auxiliar. Así, por ejemplo:

aux = vec[i] vec[i] = vec[k] vec[k] = aux

Recuerde que los parámetros por referencia son aquellas variables que se modificarán dentro de los métodos y volverán al programa principal con su valor modificado.

Recuerde que la información que se guarda en archivos, es una información que permanece en el tiempo.

No olvide que los archivos se deben de abrir y también se deben cerrar.

Tenga en cuenta los siguientes puntos en el manejo dinámico de memoria:

El puntero es una dirección de memoria en la RAM.

Un nodo es un espacio de memoria.

Un nodo simple, está compuesto por una parte de dato y otra parte de liga.

Recuerde que al pedir un nodo se hace con la instrucción new(x) y para liberar el espacio de memoria se hace con free(x).

No olvide que las pilas y las colas, no tienen representación propia a nivel de programación, que para hacerlo se apoyan en vectores y listas ligadas.

Tenga en cuenta que cuando se busca el contador de frecuencias de un algoritmo, y en éste hay una instrucción que se ejecuta n*n veces, lo representamos como n2; y si existe una instrucción que se ejecuta n*n*n veces, ésta instrucción la representamos como n3.

Recuerde que, en la multiplicación, tener 2*n es lo mismo que tener 2n.

Igualmente, con los logaritmos: n*log2n es lo mismo que nlog2n.

7 GLOSARIO

Factorial: Se llama factorial de n al producto de todos los números naturales desde 1 hasta n.

Frecuencia: Repetición mayor o menor de un acto o de un suceso.

Parámetro: Variable que, en una familia de elementos, sirve para identificar cada uno de ellos mediante su valor numérico

Void: Significa vacío, esto indica que la función no devuelve nada si no que realiza ciertas acciones.

Pila: conjunto de uno o más elementos que se apilan.

Nodo simple: espacio de memoria que tiene una dirección y que se encuentra dividido en dos partes, una parte de dato y una parte de liga.

Nodo doble: espacio de memoria que tiene una dirección y que se encuentra dividido en tres partes una parte de liga izquierda, una parte de liga derecha y una parte de datos.

Evaluación a priori: consiste en medir la eficiencia de un algoritmo antes de decodificarse en algún lenguaje

8 BIBLIOGRAFÍA

Joyanes Aguilar, L. (1996). Fundamentos de programación: Algoritmos y estructura de datos. McGraw Hill/Interamericana de España S.A.

Oviedo Regino, E. M. (2015). Lógica de programación orientada a objeto, primera edición. ECOE ediciones: Bogotá.

Villalobos S., J. A., & Casallas G., R. (2006). Fundamentos de programación, aprendizaje activo basado en casos primera edición. Bogotá: Pearson educación de México S.A. de C.V.

Cairo, O., & Guardati Buemo, S. (1998). Estructuras de datos. McGraw-Hill Interamericano de México S.A.

Roberto Flórez Rueda. (2010). Algoritmia II, primera edición. Medellín Colombia.

Programación en Castellano. (s.f.). Recuperado el 9 de Junio de 2011, de http://www.programacion.com/articulo/introduccion_a_la_programacion_205/7

Universidad Nacional de Colombia sede Bogotá. (s.f.). Recuperado el 8 de Abril de 2011, de http://www.virtual.unal.edu.co/cursos/ingenieria/2001839/docs_curso/contenido.html

Canaria, D. d. (s.f.). Grupo de Estructuras de Datos y Lingüística Computacional. Recuperado el 6 de Junio de 2011, de http://www.gedlc.ulpgc.es/docencia/NGA/subprogramas.html

Granada, U. d. (s.f.). www.ugr.es. Recuperado el 9 de Junio de 2011, de http://elvex.ugr.es/decsai/c/apuntes/vectores.pdf

TECNOLÓGICO. (s.f.). Obtenido de http://www.mitecnologico.com/: http://www.mitecnologico.com/Main/ArreglosVectoresYMatrices

Universidad Tecnológica de Pereira. (s.f.). Recuperado el 6 de Junio de 2011, de http://www.utp.edu.co/~chami17/sn.htm

Vieyra, G. E. (31 de Agosto de 2000). Facultad de Ciencias Físico-Matemáticas UMSNH. Recuperado el 8 de Junio de 2011, de http://www.fismat.umich.mx/~elizalde/curso/node110.html

Wikipedia. (s.f.). Recuperado el 8 de Junio de 2011, de http://es.wikipedia.org/wiki/Sistema_de_numeraci%C3%B3n