UNIVERSIDAD AUTONOMA GABRIEL RENE MORENO

FACULTAD DE CIENCIAS DE LA COMPUTACION Y TELECOMUNICACIONES

MANUAL GUIA DE ESTRUCTRUCTURA DE DATOS I

AUTOR MSC. ING. MARIO CAMPOS B.

Santa Cruz - Bolivia

Prólogo:

Inmersión en las Estructuras de Datos

En el vasto paisaje de la informática, las estructuras de datos son los pilares que sustentan la eficiencia y la elegancia en el manejo de información. Este libro es una invitación a explorar y comprender el fascinante universo de las estructuras de datos, un componente esencial en la vida de todo desarrollador.

El conocimiento profundo de las estructuras de datos es un requisito indispensable para crear software eficiente y escalable en un mundo donde la cantidad de datos crece a un ritmo exponencial. Desde la más simple hasta la más compleja, estas estructuras ofrecen una variedad de formas de organizar y manipular datos, cada una con sus propias ventajas y aplicaciones.

En estas páginas, nos embarcaremos en un viaje desde los conceptos más elementales hasta las complejidades más avanzadas de las estructuras de datos. Exploraremos listas, pilas, colas, árboles, grafos y más, desglosando cada una en su esencia y desentrañando los algoritmos que les dan vida.

Nuestro propósito es proporcionar una comprensión clara y accesible de estos conceptos, acompañada de ejemplos prácticos y aplicaciones del mundo real. Queremos que te sumerjas en este viaje de aprendizaje con entusiasmo y que al final, no solo comprendas las estructuras de datos, sino que las incorpores como herramientas poderosas en tu caja de herramientas de desarrollo.

Descubrirás cómo elegir la estructura de datos adecuada para cada situación, cómo optimizar su rendimiento y cómo evitar trampas comunes que pueden afectar negativamente la eficiencia de tus aplicaciones. Además, a lo largo del libro, te alentaremos a desarrollar tu pensamiento analítico y habilidades para resolver problemas, habilidades que son cruciales en el mundo de la programación.
En este libro aprenderemos todo lo relacionado a los fundamentos de Estructuras de Datos como así su definición e importancia además de conceptos básicos y una herramienta fundamental para el modelado como son los Tipos de datos abstractos (TDA).
Seguidamente veremos el análisis e implementación de estructuras Lineales como son los arreglos unidimensionales, listas enlazadas listas circulares, Pilas y Colas que nos permitirán poder manejar y tratar cualquier tipo de información que sea requerida de forma óptima y eficiente.

Así que, con mente abierta y curiosidad en el corazón, te invitamos a comenzar este viaje educativo hacia las profundidades de las estructuras de datos. Esperamos que esta experiencia enriquecedora te empodere y te inspire a crear soluciones informáticas cada vez más innovadoras y efectivas. ¡Que el viaje comience!

Capítulo 1

Introducción a las Estructuras de Datos

INTRODUCCIÓN

La representación de la información es fundamental en ciencias de la computación y en informática, el propósito principal de la mayoría de los programas de computadoras es almacenar y recuperar información, además de realizar cálculos. De modo práctico, los requisitos de almacenamiento y tiempo de ejecución exigen que tales programas deban organizar su información de un modo que soporte procesamiento eficiente. Por estas razones, el estudio de estructuras de datos y de los algoritmos que las manipulan constituye el núcleo central de la *informática* y de la *computación*.

Se revisan en este capítulo los conceptos básicos de *dato, abstracción, algoritmos* y *programas*, así como los criterios relativos a *análisis y eficiencia de algoritmos.*

1.1. TIPOS DE DATOS

Los lenguajes de programación tradicionales, como Pascal y C, proporcionan *tipos de datos* para clasificar diversas clases de datos. Las ventajas de utilizar tipos en el desarrollo de software son:

- Apoyo y ayuda en la prevención y en la detección de errores.
- Apoyo y ayuda a los desarrolladores de software, y a la comprensión y organización de ideas acerca de sus objetos.
- Ayuda en la identificación y descripción de propiedades únicas de ciertos tipos.

Los tipos son un enlace importante entre el mundo exterior y los elementos datos que manipulan los programas. Su uso permite a los desarrolladores limitar su atención a tipos específicos de datos, que tienen propiedades únicas. Por ejemplo, el tamaño es una propiedad determinante en los arrays y en las cadenas; sin embargo, no es una propiedad esencial en los valores lógicos, verdadero (true) y falso (false).

Definición 1: Un *tipo de dato* es un conjunto de valores y operaciones asociadas a esos valores.

Definición 2: Un *tipo de dato* consta de dos partes: un conjunto de datos y las operaciones que se pueden realizar sobre esos datos.

Una estructura de datos al ser una colección de elementos organizados con un propósito específico para almacenar, organizar y procesarlas.
Las estructuras de datos podemos clasificar de acuerdo a su composición en:

a) Simples o Primitivos
b) Compuestas o Agregados

En los lenguajes de programación hay disponible un gran número de tipos de datos. Entre ellos se pueden destacar los tipos primitivos de datos, los tipos compuestos y los tipos agregados.

1.1.1. Tipos primitivos de datos

Los tipos de datos más simples son los tipos de datos primitivos, también denominados datos atómicos porque no se construyen a partir de otros tipos y son entidades únicas no descomponibles en otros.

Un tipo de dato atómico es un conjunto de datos atómicos con propiedades idénticas. Estas propiedades diferencian un tipo de dato atómico de otro. Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.

Tipo de dato atómico

1. Un conjunto de valores.

2. Un conjunto de operaciones sobre esos valores.

Ejemplo 1.1

Diferentes tipos de datos atómicos

Enteros
valores -∞ , …, -3, -2, -1, 0, 1, 2, 3, …, +∞
operaciones *, +, -, /, %, ++, --, …
Coma flotante
valores - , …, 0.0, …,
operaciones *, +, -, %, /, …
Carácter
valores \0, …, 'A', 'B', …, 'a', 'b', …
operaciones <, >, …
Lógico
valores verdadero, falso
operaciones and, or, not, …

1.1.2. Tipos de datos compuestos y agregados

Los datos compuestos son el tipo opuesto a los tipos de datos atómicos. Los datos compuestos se pueden romper en subcampos que tengan significado. Un ejemplo sencillo es el número de su teléfono celular 51199110101. Realmente, este número consta de varios campos, el código del país (51, Perú), el código del área (1, Lima) y el número propiamente dicho, que corresponde a un celular porque empieza con 9.

En algunas ocasiones los datos compuestos se conocen también como datos o tipos agregados.

Los tipos agregados son tipos de datos cuyos valores constan de colecciones de elementos de datos. Un tipo agregado se compone de tipos de datos previamente definitivos. Existen tres tipos agregados básicos: arrays (arreglos), secuencias y registros.

Un array o arreglo es, normalmente, una colección de datos de tamaño o longitud fija, cada

uno de cuyos datos es accesible en tiempo de ejecución mediante la evaluación de las expresiones que representan a los subíndices o índices correspondientes. Todos los elementos de un array deben ser del mismo tipo.

array de enteros: [4, 6, 8, 35, 46, 0810]

Una secuencia o cadena es, en esencia, un array cuyo tamaño puede variar en tiempo de ejecución. Por consiguiente, las secuencias son similares a arrays dinámicos o flexibles.

Cadena = "Aceite picual de Carchelejo"

Un *registro* puede contener elementos datos agregados y primitivos. Cada elemento agregado, eventualmente, se descompone en campos formados por elementos primitivos. Un registro se puede considerar como un tipo o colección de datos de tamaño fijo. Al contrario que en los arrays, en los que todos sus elementos deben ser del mismo tipo de datos, los campos de los registros pueden ser de diferentes tipos de datos. A los campos de los registros se accede mediante identificadores.

El registro es el tipo de dato más próximo a la idea de objeto. En realidad, el concepto de objeto en un desarrollo orientado a objetos es una generalización del tipo registro.

Registro {
Dato1
Dato2
Dato3
...
}

1.2. LA NECESIDAD DE LAS ESTRUCTUR AS DE DATOS

A pesar de la gran potencia de las computadoras actuales, la eficiencia de los programas sigue siendo una de las características más importantes a considerar. Los problemas complejos que procesan las computadoras cada vez más obligan, sobre todo, a pensar en su eficiencia dado el elevado tamaño que suelen alcanzar. Hoy, más que nunca, los profesionales deben formarse en técnicas de construcción de programas eficientes.

En sentido general, una estructura de datos es cualquier representación de datos y sus operaciones asociadas. Bajo esta óptica, cualquier representación de datos, incluso un número entero o un número de coma flotante almacenado en la computadora, es una sencilla estructura de datos.

En un sentido más específico, una estructura de datos es una organización o estructuración de una colección de elementos dato. Así, una lista ordenada de enteros almacenados en un array es un ejemplo de dicha estructuración.

Una estructura de datos es una agregación de tipos de datos compuestos y atómicos en un conjunto con relaciones bien definidas. Una estructura significa un conjunto de reglas que contienen los datos juntos.

Las estructuras de datos pueden estar anidadas: se puede tener una estructura de datos que conste de otras.

Estructura de datos

1. Una combinación de elementos en la que cada uno es o bien un tipo de dato u otra estructura de datos.

2. Un conjunto de asociaciones o relaciones (estructura) que implica a los elementos combinados.

Etapas en la selección de una estructura de datos

Los pasos a seguir para seleccionar una estructura de datos que resuelva un problema son

1. Analizar el problema para determinar las restricciones de recursos que debe cumplir cada posible solución.

2. Determinar las operaciones básicas que se deben soportar y cuantificar las restricciones de recursos para cada una. Ejemplos de operaciones básicas son la inserción de un dato en la estructura de datos, suprimir un dato de la estructura o encontrar un dato determinado en dicha estructura.

3. Seleccionar la estructura de datos que cumple mejor los requisitos o requerimientos.

Este método de tres etapas para la selección de una estructura de datos es una aproximación centrada en los datos. Primero se diseñan los datos y las operaciones que se realizan sobre ellos, a continuación viene la representación de esos datos y, por último, viene la implementación de esa representación.

Las restricciones de recursos sobre ciertas operaciones clave, como búsqueda, inserción de registros de datos y eliminación de registros de datos, normalmente conducen el proceso de selección de las estructuras de datos.

1.3 ATRIBUTOS PARA LA ESPECIFICACIÓN DE TIPOS DE ESTRUCTURAS DE DATOS

Los atributos principales para especificar estructuras de datos son:

1.3.1 NÚMERO DE COMPONENTES.

Una estructura de datos puede ser de tamaño

Fijo. - Si el número de componentes no es invariable durante el tiempo de vida (arreglos y registros)

Variable. - Si el número de componentes cambia en forma dinámica (listas, conjuntos, tablas y archivos). Los objetos de datos de tamaño variable suelen emplear un tipo de dato apuntador, además de definir operaciones que inserten y eliminen componentes de la estructura.

1.3.2. TIPO DE CADA COMPONENTE.

Estructuras homogéneas. Una estructura de datos es homogénea si todos sus componentes son del mismo tipo,

Estructuras heterogéneas. Una estructura de datos es Heterogénea si alguno de sus componentes no es del mismo tipo que los Demás.

1.3.3. NOMBRES QUE SE DEBEN USAR PARA SELECCIONAR COMPONENTES.

Un tipo de estructura de datos necesita un mecanismo de selección para identificar componentes individuales. En arreglos puede ser un subíndice y en registros el nombre propio de cada campo.

1.3.4. NÚMERO MÁXIMO DE COMPONENTES.

Determina el tamaño de la estructura en base al número de componentes.

1.3.5. ORGANIZACIÓN DE LOS COMPONENTES.

La organización más común es una serie lineal sencilla de componentes. Sin embargo, hay estructuras que abarcan formas multidimensionales que pueden tratarse como el tipo secuencial básico en el cual los componentes son estructuras de datos de tipo similar.

1.4. ESPECIFICACIÓN DE LAS ESTRUCTURA DE DATOS POR EL NÚMERO DE COMPONENTES.

Según las definiciones señaladas anteriormente estas estructuras se las clasifica por el espacio físico de memoria en el que son representadas y se las clasifican como:

- Estructuras Estáticas(Fijas)
- Estructuras Dinámicas(Variable)

1.4.1. ESTRUCTURAS DE DATOS ESTÁTICAS

Estas estructuras representan un espacio físico fijo definido por el programador ,es decir cuyo tamaño queda determinado en tiempo de compilación. Este tipo de estructuras y esta permanece mientras el proceso que la maneja este activo.

1.4.1.1. TIPOS DE ESTRUCTURAS ESTATICAS

Este tipo de estructura de datos permite mantener fija una parte de la memoria que se define para almacenar a sus elementos.

Entre este tipos de datos tenemos a :

- String o Cadenas
- Arreglos
- Registros

STRINGS.- Una cadena de caracteres es un objeto de datos compuesto de una serie de caracteres.
Es posible identificar al menos tres tratamientos distintos de los tipos de este tipo de datos:
Longitud Fija declarada. Es un objeto de datos de cadena de caracteres con una longitud fija que se declara en el programa.

Longitud variable hasta un Límite Declarado. Es un objeto de datos de cadena de caracteres con una longitud máxima que se declara en el programa como en el caso anterior, pero el valor real que se guarda en el objeto de datos puede ser una cadena de longitud más corta, posiblemente incluso la cadena vacía.

Longitud Ilimitada. Es un objeto de datos de cadena de caracteres que puede tener un valor de cadena de cualquier longitud, y la longitud puede variar en forma dinámica en tiempo de Ejecución la ejecución sin límite alguno.

ARREGLOS. - Son estructuras de datos de memoria secuencial en el que sus elementos pueden ser simples o compuestos referenciados mediante índices de acuerdo a la definición de las rectas.
Son considerados también estructuras homogéneas compuesta por varios componentes todas del mismo tipoy almacenarlas consecutivamente en memoria. Los arreglos se caracterizan por :

1. Por tener un solo nombre
2. Sus elementos se distingue por subíndices que indica la posición en la memoria de la computadora.
3. Se almacena en una memoria fija y ordenada
4. Permite acceso aleatorio o secuencial en la lectura o escritura de sus elementos.

VECTORES. - También llamado arreglo unidimensional o lineal, es una estructura de datos integrada por un número fijo de componentes del mismo tipo organizados como una serie lineal simple. Un componente de un vector se selecciona dado su subíndice, que indica la posición del componente en la serie.

IMPLANTACIÓN DE LOS VECTORES

En el Siguiente Figura 1 se realiza una representación gráfica de la implantación de los Vectores en forma Genérica

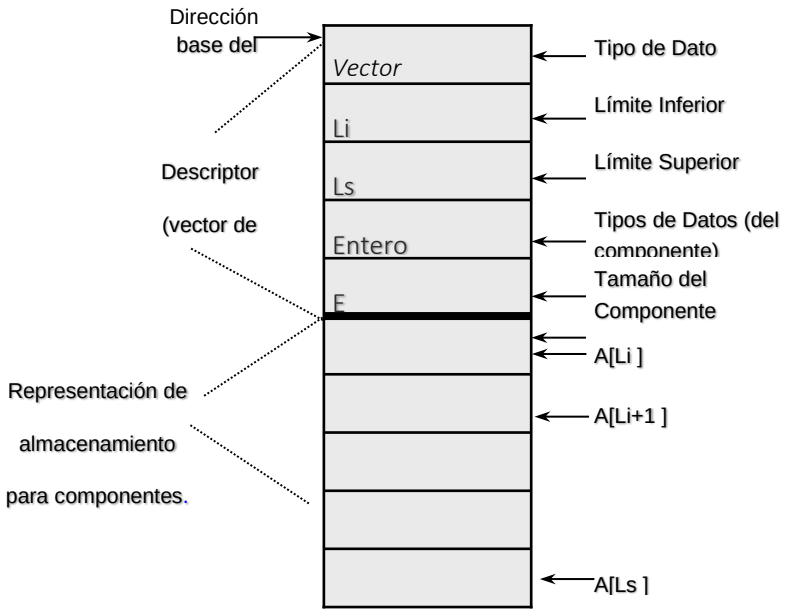


Fig.1 Implantación de Vectores

MATRICES. - Un arreglo multidimensional se construye a partir de arreglos de dimensiones menores, por ejemplo, un arreglo bidimensional que es una matriz compuesta de filas y columnas, se construye como un arreglo de arreglos; un arreglo tridimensional se compone de planos de filas y columnas, es decir como un arreglo de matrices.

Este tipo de estructuras se desplazan por dos criterios por ancho o columna y por alto o fila.

IMPLANTACIÓN DE LOS ARREGLOS MULTIDIMINCIONALES

Una Matriz, se implanta considerándola un vector de vectores, siguiendo este análisis en forma sucesiva, se puede concluir que un arreglo n_dimensional se implanta en base a arreglos de dimensiones menores Ver. Fig. 2.

En ciertos contextos es importante si una matriz se considera como una columna de filas o una fila de columnas. La estructura más común es la de columna de filas, donde la matriz se considera un vector cuyo elemento es un sub_vector que representa una fila de la matriz. Esta representación se conoce como orden por filas, El orden por columnas es la representación en la cual la matriz se trata como una sola fila de columnas, y de forma semejante, al orden por filas, se puede generalizar el concepto de orden por columna para arreglos n_dimensionales.

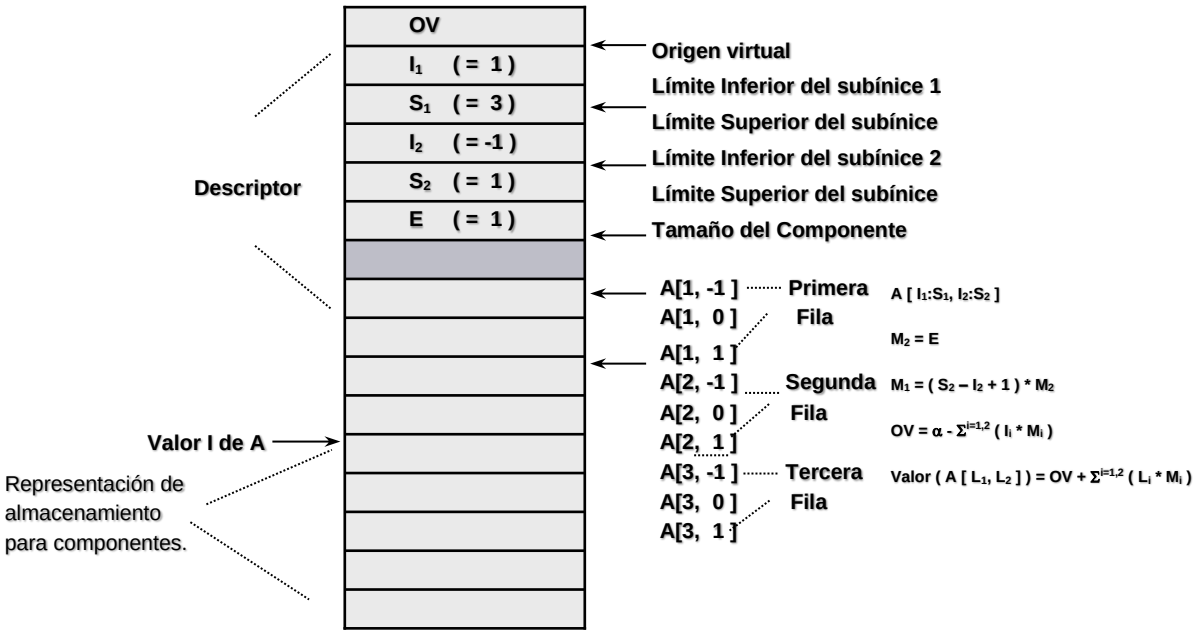


Fig. 2. Implantación de Matrices Arreglos

REGISTROS.- Un registro es una estructura de datos compuesta de un número fijo de componentes de distintos tipos, sus componentes se llaman campos. También se conocen simplemente como estructuras. Son estructuras lineales y de longitud fija, al igual que los vectores, pero difieren de estos últimos en que sus componentes pueden ser heterogéneos y se designan con nombres simbólicos.

CARACTERÍSTICAS DE LOS REGISTROS

Son estructuras para definir una entidad, un objeto expresado en conjunto de atributos implementados como campos y sub bloques.

Para referenciar a los elementos de un registro se debe de especificar el nombre de la estructura y una secuencia de desplazamientos identificados por sus bloques hasta encontrar la identificación de un elemento.

ESQUEMA LÓGICO DE LOS REGISTROS

En la Fig. 3. se representa gráficamente un Esquema lógico de los registro

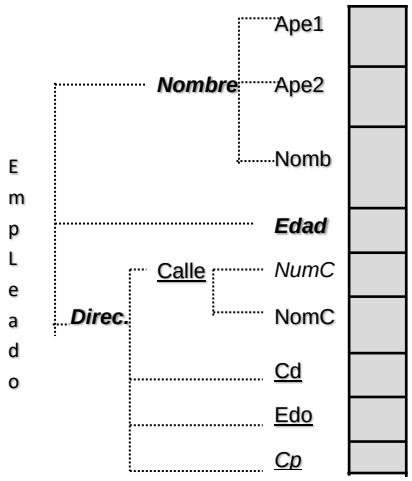


Fig. 3. Esquema Lógico de los Registros

1.4.2. ESTRUCTURAS DE DATOS DINAMICAS

LISTAS. - Una lista es una estructura de datos compuesta de una serie ordenada de estructuras de datos.

Las listas son similares a los arreglos, en cuanto se componen de una serie ordenada de objetos, así se puede hacer referencia al primer elemento de la lista, al segundo y así sucesivamente hasta llegar al último elemento de la lista. Al primer elemento de la lista se le llama comúnmente cabeza y al último elemento cola. Sin embargo, las listas difieren de los arreglos en los siguientes aspectos:

Las lista son de longitud variable, es decir, aumentan y se reducen durante la ejecución del programa, y se usan para representar estructuras de datos arbitrarias.
Sus componentes pueden ser heterogéneos, es decir, el tipo de dato de cada miembro de una lista puede diferir de su "vecino".

Los lenguajes que manejan listas, declaran estos datos de manera implícita, es decir sin atributos explícitos para los miembros de la lista.

primer elemento

$$L=<a_1, a_2,\ldots,a_n>$$

ultimo elemento

Donde.

$a_i \in T$, **i=1,…,n (n** es la longitud de la lista**)**

n=0 $\Rightarrow$ lista vacía

Una manera de clasificarlas es por la forma de acceder al siguiente elemento:

- Lista densa: la propia estructura determina cuál es el siguiente elemento de la lista. Ejemplo: un Array.
- Lista enlazada: La posición del siguiente elemento de la estructura la determina el elemento actual.

Es necesario almacenar al menos la posición de memoria del primer elemento. Además es dinámica, es decir, su tamaño cambia durante la ejecución del programa.
Una lista enlazada se puede definir recursivamente de la siguiente manera:

- Una lista enlazada es una estructura vacía o
- Un elemento de información y un enlace hacia una lista (un nodo).

REPRESENTACIÓN GRAFICA

El siguiente Grafico Fig. 4.se representa un Lista Enlazada

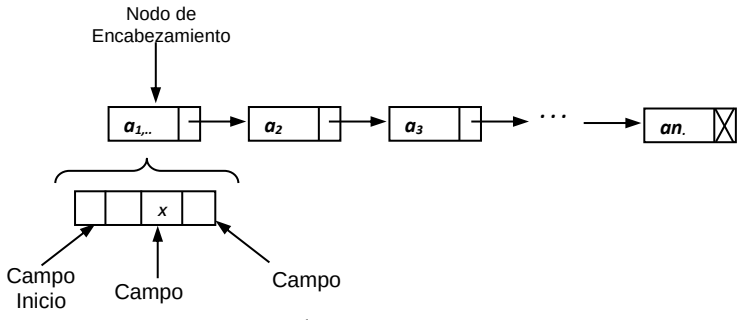


Fig.4. Representación de una Lista

Como se ha dicho anteriormente, pueden cambiar de tamaño, pero su ventaja fundamental es que son flexibles a la hora de reorganizar sus elementos; a cambio se ha de pagar una mayor lentitud a la hora de acceder a cualquier elemento.

En la lista de la fig. 4. se puede observar que hay dos elementos de información, x e y. Supongamos que queremos añadir un nuevo nodo, con la información p, al comienzo de la lista. Para hacerlo basta con crear ese nodo, introducir la información p, y hacer un enlace hacia el siguiente nodo, que en este caso contiene la información x.

¿Qué ocurre si quisiéramos hacer lo mismo sobre un Array?. En ese caso sería necesario desplazar todos los elementos de información "hacia la derecha", para poder introducir el nuevo elemento, una operación muy engorrosa.

OPERACIONES BASICAS DE UNA LISTA

OPERACIONES DE CONSTRUCCIÓN
- CREA

OPERACIONES DE POSICIONAMIENTO
- FIN
- PRIMERO
- SIGUIENTE
- ANTERIOR

OPERACIONES DE CONSULTA
- VACIA
- RECUPERA
- LONGITUD

OPERACIONES DE MODIFICACIÓN
- INSERTA
- SUPRIME

- MODIFICA

ANALISIS DE PROCESOS VECTOR VS LISTA ENLAZADA

Operación	Representación Contigua	Representación Enlazada
Fin	*O(1)*	*O(1)*
Primero	*O(1)*	*O(1)*
Sgte	*O(1)*	*O(1)*
Ante	*O(1)*	***O(n)***
Vacia	*O(1)*	*O(1)*
Recupera	*O(1)*	*O(1)*
Longitud	*O(1)*	*O(1)*
Inserta	***O(n)***	*O(1)*
Suprime	***O(n)***	*O(1)*
Modifica	*O(1)*	*O(1)*

VARIACIONES SOBRE LISTAS

En ciertos lenguajes se presentan variaciones sobre la estructura típica de las listas, entre las cuales tenemos: las pilas, las colas, los árboles, las gráficas dirigidas y las listas de propiedades.

PILA.- Una pila es una lista donde la selección, inserción y eliminación de componentes están restringidas a un extremo. En tiempo de ejecución es un objeto de datos medular definido por el sistema.

COLA .- Una cola es una lista en la cual la selección y eliminación de componentes están restringidas a un extremo y la inserción esta restringida al otro extremo. Se usan en la organización y sincronización de subprogramas concurrentes.

Son comunes las representaciones secuenciales tanto para pilas como para colas

ARBOLES.- Un árbol es una lista donde los componentes pueden ser listas y objetos de datos elementales, siempre y cuando cada lista sea un componente de cuando mucho otra lista. Se suelen usar para representar tablas de símbolos en un compilador.

GRAFOS.- Una gráfica dirigida es una estructura de datos en la cual los componentes se pueden vincular entre sí usando patrones de vinculación arbitrarios (en vez de series lineales de componentes).

LISTA DE PROPIEDADES.- Una lista de propiedades, también conocida como lista de valores de atributos, o lista de descripción, o tabla, es un registro con un número variable de componentes. En una lista de componentes de debe guardar tanto los nombres de los componentes como sus valores. Cada nombre de campo se conoce como nombre de propiedad y el valor correspondiente se conoce como valor de propiedad.

MULTILISTAS.- Un Multilista es un Conjunto de nodos en la que algunos tienen más de una referencia a otro nodo(Lista de Listas) y pueden estar en más de una lista simultáneamente, ver Fig.5.

Para cada tipo de nodo es importante distinguir los distintos campos puntero para realizar los recorridos adecuados y evitar confusiones.

Es la Estructura básica para Sistemas de Bases de Datos en Red. basado en la teoría de grafos

REPRESENTACION GRAFICA:

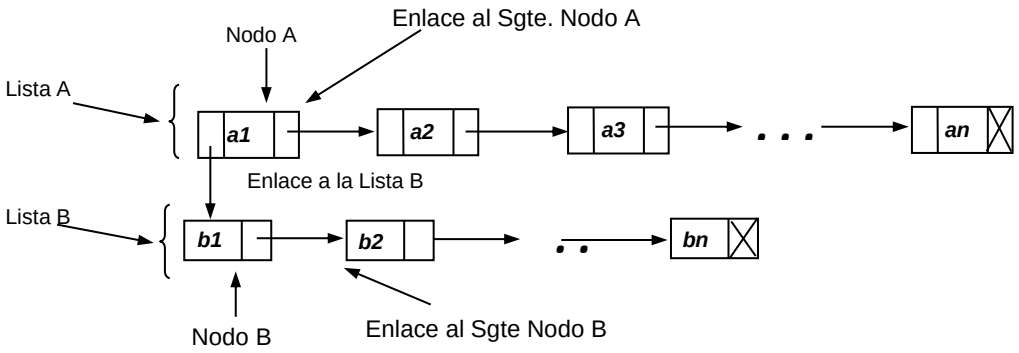


Fig. 5. Representación Grafica de una Multilista

En la lista de la figura anterior se puede observar que hay n elementos de información, a1,a2,a3...an que tiene dos campos con referencias a otros elementos el primero contiene una referencia a otro elemento del tipo Ai y el segundo contiene una referencia o un puntero a otra lista de datos de Tipo B en cual esta secuencialmente enlazado de la forma b1,b2,b3.....bn.

ARCHIVOS.- Un archivo es una estructura de datos con dos propiedades especiales:

1.- Se representa ordinariamente en un dispositivo de almacenamiento secundario, como un disco o una cinta, y por tanto puede ser mucho más grande, y por tanto puede ser mucho más grande que la mayoría de las estructuras de datos de otros tipos.

2.- Su tiempo de vida puede abarcar un intervalo de tiempo mayor que el del programa que lo crea.

TIPOS DE ARCHIVOS

- Archivos Texto
- Archivos con Tipo

ARCHIVOS TEXTO.- Son aquellos archivos en los cuales cuyo contenido son exclusivamente Texto y están formados por líneas de diferente longitud donde la información es un texto, cada línea termina con un retorno de carro y avance de línea(LR/LF).

Los textfiles son una forma de archivo secuencial ordinario y se pueden manipular en las mismas formas. Sin embargo, se suelen suministrar operaciones especiales para textfiles que permiten la conversión automática de datos numéricos (y a otros tipos de datos) a representaciones de almacenamiento interno. Permiten el formateo de salida, como parte importante de la implementación de operaciones de salida

ARCHIVOS CON TIPO- Llamados también archivos aleatorios o directos, son aquellos archivos que están organizados secuencialmente con componentes del mismo tipo, Es de longitud variable y carece de límite máximo fijo (más Por ejemplo, en Pascal, un archivo se declara dando su nombre y el tipo de componente que contiene:

```
Maestro: file of RegEmpleado;
```

Define un archivo llamado Maestro cuyos componentes son del tipo RegEmpleado. El tipo de componentes puede ser un tipo elemental o un tipo de estructura de datos de tamaño fijo, como un arreglo o un registro. Cuando los datos del archivo se leen más tarde, las ubicaciones de almacenamiento a las que hacen referencia los valores de apuntador pueden estar en uso para otro propósito.

Capítulo 2

Tipos de Datos Abstractos : Clases y Objetos

INTRODUCCIÓN

En este capítulo se examinan los conceptos de *modularidad* y *abstracción de datos*. La modularidades la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. La *abstracción de datos* es la técnica para inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, faciliten la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que, cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como int, char y float en Java, C y C++. Lenguajes de programación como Java tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato, **TAD,***

*(abstract data type, **ADT**)*. El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de tipo abstracto de dato (TAD) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado TAD.

Una ***clase*** es un tipo de dato que contiene código (***métodos***) y ***datos***. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En este capítulo se aprenderá a crear (definir y especificar) y a utilizar clases individuales.

2.1. ABSTRACCION EN LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: *abstracción de datos* (perteneciente a los datos) y *abstracción de control* (perteneciente a las estructuras de control).
Desde comienzos de la década de los sesenta, cuando se desarrollaron los primeros lenguajes de programación de alto nivel, ha sido posible utilizar las abstracciones más primitivas de ambas categorías (variables, tipos de datos, procedimientos, control de bucles, etc.).

2.1.1. Abstracciones de control

Los microprocesadores ofrecen directamente sólo dos mecanismos para controlar el flujo y ejecución de las instrucciones: *secuencia* y *salto*. Los primeros lenguajes de programación de alto nivel introdujeron las estructuras de control: sentencias de bifurcación (if) y bucles (for, while, do-loop, etc.).

Las estructuras de control describen el orden en el que se ejecutan las sentencias o grupos de sentencia *(unidades de programa)*. Las unidades de programa se utilizan como bloques básicos de la clásica descomposición "descendente". En todos los casos, los subprogramas constituyen una herramienta potente de abstracción ya que, durante su implementación, el programador describe en detalle cómo funcionan. Cuando el subprograma se llama, basta con conocer lo que hace y no cómo lo hace. De este modo, se convierten en cajas negras que amplían el lenguaje de programación a utilizar. En general, los subprogramas son los mecanismos más ampliamente utilizados para reutilizar código, a través de colecciones de subprogramas en bibliotecas.

Las abstracciones y las estructuras de control se clasifican en estructuras de control a nivel de sentencia y a nivel de unidades. Las abstracciones de control a nivel de unidad se conocen como *abstracciones procedimentales*.

Abstracción *procedimental* (por procedimientos)

Es esencial para diseñar software modular y fiable la abstracción procedimental que se basa en la utilización de procedimientos o funciones sin preocuparse de cómo se implementan. Esto es posible sólo si conocemos qué hace el procedimiento; esto es, conocemos la sintaxis y la semántica que utiliza el procedimiento o función. La abstracción aparece en los subprogramas debido a las siguientes causas:

- Con el nombre de los subprogramas, un programador puede asignar una descripción abstracta que captura el significado global del subprograma. Utilizando el nombre en lugar de escribir el código, permite al programador aplicar la acción en términos de su descripción de alto nivel en lugar de sus detalles de bajo nivel.
- Los subprogramas proporcionan ocultación de la información. Las variables locales y cualquier otra definición local se encapsulan en el subprograma, ocultándose de forma que no pueden utilizarse fuera del subprograma. Por consiguiente, el programador no tiene que preocuparse sobre las definiciones locales.
- Los parámetros de los subprogramas, junto con la ocultación de la información anterior, permiten crear subprogramas que constituyen entidades de *software* propias. Los detalles locales de la implementación pueden estar ocultos, mientras que los parámetros se pueden utilizar para establecer la interfaz *pública*.

En Java, la abstracción procedimental se establece con los métodos o funciones miembros de clases.

2.2. T IPOS ABSTR ACTOS DE DATOS

Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina tipo abstracto de datos (TAD) para diferenciarlo del tipo fundamental (predefinido) de datos. Por ejemplo, en Java, el tipo Punto, que representa las coordenadas x e y de un sistema de coordenadas rectangulares, no existe. Sin embargo, es posible implementar el tipo abstracto de datos, considerando los valores que se almacenan en las variables y qué operaciones están disponibles para manipular estas variables. En esencia, un tipo abstracto es un tipo de dato que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre ellos. Un TAD se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

Para recordar

Un tipo abstracto de datos puede definirse mediante la ecuación:
TAD = Representación (datos) + Operaciones (funciones y procedimientos)

La estructura de un tipo abstracto de dato *(clase)*, desde un punto de vista global, se compone de la interfaz y de la implementación (Figura 6).
Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los TAD están encapsuladas dentro de los propios TAD. La característica de ocultamiento de la información significa que los objetos tienen *interfaces públicas*. Sin embargo, las representaciones e implementaciones de esas interfaces son *privadas*.

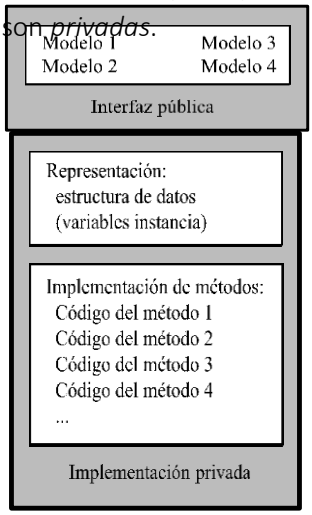


Figura6 . Estructura de un tipo abstracto de datos (TAD)

2.3 VENTAJAS DE LOS TIPOS ABSTRACTOS DE DATOS

Un tipo abstracto de datos es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permiten una mejor conceptualización y modelización del mundo real. Mejoran la representación y la comprensibilidad. Clarifican los objetos basados en estructuras y comportamientos comunes.
2. Mejoran la robustez del sistema. Si hay características subyacentes en los lenguajes, permiten la especificación del tipo de cada variable. Los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.
3. Mejoran el rendimiento (prestaciones). Para sistemas tipeados (tipificados), el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separan la implementación de la especificación. Permiten la modificación y la mejora de la implementación sin afectar la interfaz pública del tipo abstracto de dato.
5. Permiten la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.

6. Recogen mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un TAD lo hace teniendo en cuenta las operaciones o la funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los usuarios de un TAD se comunican con éste a partir de la interfaz que ofrece el TAD mediante funciones de acceso. Podría cambiarse la implementación del tipo de dato sin afectar al programa que usa el TAD ya que para el programa está oculta.

2.4 IMPLEMENTACIÓN DE LOS TAD

Las unidades de programación de lenguajes que pueden implementar un TAD reciben distintos nombres:

Modula-2	Modulo
Ada	Paquete
C++	Clase
Java	Clase

En estos lenguajes se definen la especificación del TAD, que declara las operaciones y los datos, y la implementación, que muestra el código fuente de las operaciones, que permanece oculto al exterior del módulo.

2.5. ESPECIFICACIÓN DE LOS TAD

El objetivo de la especificación es describir el comportamiento del TAD; consta de dos partes, la descripción matemática del conjunto de datos y la de las operaciones definidas en ciertos elementos de ese conjunto de datos.

La especificación del TAD puede tener un **enfoque informal**, que describe los datos y las operaciones relacionadas en lenguaje natural. Otro enfoque mas riguroso, la **especificación formal**, supone suministrar un conjunto de axiomas que describen las operaciones en su as-pecto sintáctico y semántico.

2.5.1 Especificación informal de un TAD

Consta de dos partes:

- Detallar en los datos del tipo los valores que pueden tomar.
- Describir las operaciones relacionándolas con los datos.

El formato que generalmente se emplea, primero especifica el nombre del TAD y los datos:

TAD nombre del tipo (valores y su descripción)

A continuación, cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural, con este formato:

- *Operación(argumentos)*
- *Descripción funcional*

Como ejemplo, se va a especificar el tipo abstracto de datos Conjunto:

TAD Conjunto(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).
Operaciones, se ponen las operaciones básicas sobre conjuntos:
Conjuntovacio.
Crea un conjunto sin elementos.
Añadir(Conjunto, elemento).
Comprueba si el elemento forma parte del conjunto; en caso negativo, es añadido. La operación modifica al conjunto.
*Retirar(*Conjunto, elemento).
Si el elemento pertenece al conjunto, es eliminado de éste. La operación modifica al conjunto.
Pertenece(Conjunto, elemento).
Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve cierto.
Esvacio(Conjunto).
Verifica si el conjunto no tiene elementos, en cuyo caso devuelve cierto.
Cardinal(Conjunto).
Devuelve el número de elementos del conjunto.
Union(Conjunto, Conjunto).
Realiza la operación matemática de la unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes a los dos conjuntos.
Se pueden especificar más operaciones sobre conjuntos, todo dependerá de la aplicación que se quiera dar al TAD.

A tener en cuenta
La especificación informal de un TAD tiene como objetivo describir los datos del tipo y las operaciones según la funcionalidad que tienen. No sigue normas rígidas al hacer la especificación, simplemente indica, de forma comprensible, la acción que realiza cada operación.

2.5.2. Especificación formal de un TAD

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir una parte de sintaxis, en cuanto a los tipos de los argumentos y al tipo del resultado, y una parte de semántica, donde se detalla la expresión del resultado que se obtiene para unos valores particulares de los argumentos. La especificación formal ha de ser lo bastante potente para que cumpla el objetivo de verificar la corrección de la implementación del TAD.
El esquema que sigue consta de una cabecera con el nombre del TAD y los datos:

TAD nombre del tipo (valores que toma los datos del tipo)

Le sigue la sintaxis de las operaciones, que lista las operaciones mostrando los tipos de los argumentos y el tipo del resultado:

Sintaxis

Operación(Tipo argumento, ...) -> Tipo resultado

A continuación se explica la semántica de las operaciones. Ésta se construye dando unos valores particulares a los argumentos de las operaciones, a partir de los que se obtiene una expresión resultado. Éste puede tener referencias a tipos ya definidos, valores de tipo lógico o referencias a otras operaciones del propio TAD.

Semántica

Operación(valores particulares argumentos) ⇒ expresión resultado

Al hacer una especificación formal, siempre hay operaciones definidas por sí mismas que se consideran constructores del TAD. Se puede decir que mediante estos constructores se generan todos los posibles valores del TAD. Normalmente, se elige como constructor la operación que inicializa (por ejemplo, Conjuntovacío en el TAD Conjunto) y la operación que añade un dato o elemento (esta operación es común a la mayoría de los tipos abstractos de datos). Se acostumbra a marcar con un asterisco las operaciones que son constructores.

A continuación se especifica formalmente el TAD Conjunto; para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa si-entonces-sino.

TAD Conjunto(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).

Sintaxis

- Conjuntovacio -> Conjunto
- Añadir(Conjunto, Elemento) -> Conjunto
- Retirar(Conjunto, Elemento) -> Conjunto
- Pertenece(Conjunto, Elemento) -> Boolean
- Esvacio(Conjunto) -> Boolean
- Cardinal(Conjunto) -> Entero
- Union(Conjunto, Conjunto) -> Conjunto

Semántica $\forall$ e1,e2 $\in$ Elemento y $\forall$ C,D $\in$ Conjunto

Añadir(Añadir(C, e1), e1) $\Rightarrow$ Añadir(C, e1)

Añadir(Añadir(C, e1), e2) $\Rightarrow$ Añadir(Añadir(C, e2), e1)

Retirar(Conjuntovacio, e1) $\Rightarrow$ Conjuntovacio

Retirar(Añadir(C, e1), e2) $\Rightarrow$ si e1 = e2 entonces Retirar(C,e2) sino Añadir(Retirar(C,e2),e1)

Pertenece(Conjuntovacio, e1) $\Rightarrow$ falso

Pertenece(Añadir(C, e2), e1) $\Rightarrow$ si e1 = e2 entonces cierto sino Pertenece(C, e1)

Esvacio(Conjuntovacio) $\Rightarrow$ cierto

Esvacio(Añadir(C, e1)) $\Rightarrow$ falso

Cardinal(Conjuntovacio) $\Rightarrow$ Cero

Cardinal(Añadir(C, e1)) $\Rightarrow$ si Pertenece(C,e1) entonces Cardinal(C) sino 1 + Cardinal(C)

Union(Conjuntovacio, Conjuntovacio) ⇒ Conjuntovacio
Union(Conjuntovacio, Añadir(C, e1)) ⇒ Añadir(C, e1)
Union(Añadir(C, e1), D) ⇒ Añadir(Union(C, D), e1)

2.6 CLASES Y OBJETOS

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard, que al intentar escribir un programa de computadora que describiera el movimiento de los barcos a través de un fiordo, descubrió que era muy difícil simular las mareas, los movimientos de los barcos y las formas de la línea de la costa con los métodos de programación existentes en ese momento. Descubrió que los elementos del entorno que trataba de modelar —barcos, mareas y línea de la costa de los fiordos— y las acciones que cada elemento podía ejecutar mantenían unas relaciones que eran más fáciles de manejar.
Las tecnologías orientadas a objetos han evolucionado mucho, pero mantienen la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos, como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, como posición, tamaño, color, forma, textura, etc. que definen su estado. Los objetos también poseen ciertos comportamientos que los hacen diferentes de otros objetos.
Booch1 define un objeto como “algo que tiene un estado, un comportamiento y una identidad”. Imaginemos una máquina de una fábrica. El estado de la máquina puede estar funcionan-do/parando (“on/off”), hay que tener en cuenta su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su comportamiento puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, conocer las condiciones de señal de error o cambiar la velocidad. Su identidad se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basada en las propiedades y en el comportamiento que son útiles en el tiempo.

Un mensaje es una instrucción que se envía a un objeto y que, cuando se recibe, ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.
El usuario de un objeto se comunica con el objeto mediante su interfaz, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico como una máquina de fax tiene una interfaz de usuario bien definida; por ejemplo, esa interfaz incluye el mecanismo de avance del papel, botones de marcado, el receptor y el botón “enviar”. El usuario no tiene

que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el periodo de garantía puede anularla.

¿Qué son clases?

En términos prácticos, una clase es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch define una clase como "un conjunto de objetos que comparten una estructura y un comportamiento comunes".

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como servicios o métodos. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como atributos, variables o variables instancia. El término atributo se utiliza en análisis y diseño orientado a objetos, y el término variable instancia se suele utilizar en programas orientados a objetos.

2.7. DECLARACIÓN DE UNA CLASE

Antes de que un programa pueda crear objetos de cualquier clase, ésta debe ser definida. La definición de una clase significa que se debe dar a la misma un nombre, dar nombre también a los elementos que almacenan sus datos y describir los métodos que realizarán las acciones consideradas en los objetos.

Las definiciones o especificaciones no son un código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer los métodos que utilizará el programa. Normalmente, se sitúan en archivos formando los denominados packages, se utiliza un archivo para varias clases que están relacionadas.

Formato

```
class NombreClase {
Lista_de_miembros }
```

NombreClase	Nombre definido por el usuario que identifica la clase (puede incluir letras, números y subrayados).
Lista_de_miembros	métodos y datos miembros de la clase.

Ejemplo 1 .

Definición de una clase llamada Puntoque contiene las coordenadas x e y de un punto en un plano.

```
class Punto {
private int x;    // coordenada x
 private int y;    // coordenada y

public Punto(int x_, int y_) // constructor
{
x = x_; y = y_;
}

public Punto() // constructor sin argumentos
 {
```

```
x = y = 0; }
public int leerX() // devuelve el valor de x
 {
return x; }

public int leerY() // devuelve el valor de y
{
return y; }

void fijarX(int valorX) // establece el valor de x
{
x = valorX; }

void fijarY(int valorY) // establece el valor de y
 {
y = valorY; }
}
```

Ejemplo 2 .

Declaración de la clase Edad.

```
class Edad {
private int edadHijo, edadMadre, edadPadre;- > datos
public Edad(){...}                          - > método especial: constructor
public void iniciar(int h,int e,int p){...} - > métodos
public int leerHijo(){...}
public int leerMadre(){...}
public int leerPadre(){...}
}
```

2.8.- OBJETOS

Una vez que una *clase* ha sido definida, un programa puede contener una *instancia* de la clase, denominada *objeto de la clase*. Un *objeto* se crea con el operador *new* aplicado a un *constructor* de la *clase*.

Un *objeto* de la *clase* Puntoinicializado a las coordenadas (2,1) sería:

new *Punto*(2,1);

El operador *new* crea el objeto y devuelve una referencia al *objeto* creado. Esta referencia se asigna a una *variable* del tipo de la clase. El *objeto* permanecerá vivo siempre que esté referenciado por una variable de la *clase* que es instancia.

Formato para definir una referencia

```
NombreClase varReferencia;
```

Formato para crear un objeto

```
varReferencia = new NombreClase(argmntos_constructor);
```

Toda *clase* tiene uno o mas métodos, denominados constructores, para inicializar el objeto cuando es creado; tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados. En la clase Edad del Ejemplo 2.2, el constructor no tiene argumentos, se puede crear un objeto:

```
Edad f = new Edad();
```

El operador de acceso a un miembro ( ) selecciona un miembro individual de un objeto de la clase. Por ejemplo:

```
Punto p;
p = new Punto(); p.fijarX (100);
System.out.println(" Coordenada x es " + p.leerX());
```

El operador punto (.) se utiliza con los nombres de los métodos y variables instancia para especificar que son miembros de un objeto.
Ejemplo: Clase DiaSemana, contiene el método visualizar()

```
DiaSemana hoy;   // hoy es una referencia hoy = new DiaSemana();// se ha creado un objeto
hoy.visualizar();    // llama al método visualizar()
```

2.8.1. Visibilidad de los Miembros de la Clase

Un principio fundamental en programación orientada a objetos es la ocultación de la información, que significa que a determinados datos del interior de una clase no se puede acceder por métodos externos a ella. El mecanismo principal para ocultar datos es ponerlos en una *clase* y hacerlos privados. A los datos o métodos privados sólo se puede acceder desde dentro de la *clase*. Por el contrario, los *datos* o métodos públicos son accesibles desde el exterior de la *clase*.

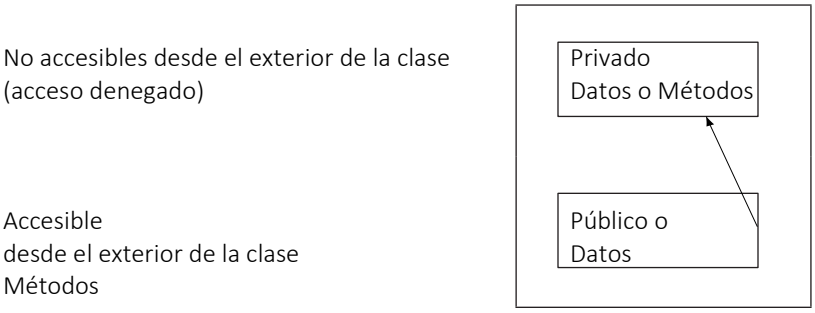


Figura 7 . Secciones pública y privada de una clase

Para controlar el acceso a los miembros de la clase se utilizan tres especificadores de acceso: *public*, *private y protected*. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

Formato

```
class NombreClase {
private declaración miembro privado; // miembros privados
protected declaración miembro protegido; // miembros protegidos
public declaración miembro público; // miembros públicos }
```

El especificador *public* define miembros públicos, que son aquellos a los que se puede ac-ceder por cualquier método desde fuera de la clase. A los miembros que siguen al especificador *private* sólo se puede acceder por métodos miembros de la misma clase. A los miembros que siguen al especificador *protected* se puede acceder por métodos miembro de la misma clase o de clases derivadas, así como por métodos de otras clases que se encuentran en el mismo paquete. Los especificadores *public*, *protected* y *private* pueden aparecer en cualquier orden. Sino se especifica acceso (acceso por defecto) a un miembro de una *clase*, a éste se puede acceder desde los *métodos* de la clase y desde cualquier método de las clases del *paquete* en el que se encuentra.

Ejemplo .

Declaración de la clase Foto y Marco con miembros declarados con distinta visibilidad. Ambas clases forman parte del paquete soporte.

```
package soporte;
class Foto {
private int nt; private char opd;
String q;
public Foto(String r) // constructor
{
nt = 0; opd = 'S';
q = new String(r); }
public double mtd(){...} }

class Marco {
private double p; String t;
public Marco() {...}
public void poner() {
Foto u = new Foto("Paloma"); p = u.mtd();
t = "**" + u.q + "**"; }
}
```

tabla . Visibilidad, “x” indica que el acceso está permitido

tipo de miembro	Miembro de la misma clase	Miembro de una clase derivada	Miembro de clase del paquete	Miembro de clase de otro paquete
Private	x			
En blanco	x		x	
Protected	x	x	x	
Public	x	x	x	x

Aunque las especificaciones *públicas, privadas y protegidas* pueden aparecer en cualquier orden, en Java los programadores suelen seguir ciertas reglas en el diseño que citamos a continuación, para que usted pueda elegir la que considere más eficiente.

1. Poner los miembros privados primero, debido a que contienen los atributos (datos).
2. Colocar los miembros públicos primero, debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permita ignorar los detalles de implementación de los objetos. Por consiguiente, los datos y los *métodos públicos* forman la interfaz externa del objeto, mientras que los elementos privados son los aspectos internos que no necesitan ser accesibles para usar el objeto. Los elementos de una clase sin especificador y los *protected* tienen las mismas propiedades que los públicos respecto a las clases del paquete.

El principio de encapsulamiento significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

Métodos de una clase

Las *métodos* en Java siempre son miembros de clases, no hay métodos o funciones fuera de las clases. La implementación de los métodos se incluye dentro del cuerpo de la clase. La Figura 8 muestra la declaración completa de una *clase*.

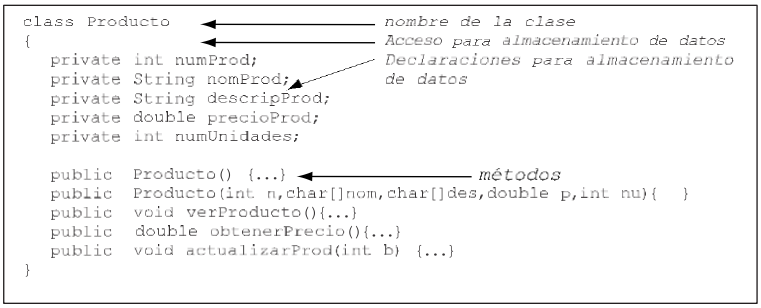


Figura 8 . Definición típica de una clase

Ejemplo.

La *clase* Racional define el *numerador* y el *denominador* característicos de un número racional. Por cada dato se proporciona un método miembro que devuelve su valor y un método que asigna numerador y denominador. Tiene un constructor que inicializa un objeto a 0/1.

```
class Racional {
private int numerador;
private int denominador;

public Racional() {
numerador = 0; denominador = 1;
}

public int leerN() { return numerador; }
public int leerD() { return denominador; }
public void fijar (int n, int d)
{
numerador = n; denominador = d;
} }
```

Ejercicio .

Definir una *clase DiaAnyoque* contenga los atributos mes y día, el método igual()y el método visualizar(. El mes se registra como un valor entero (1, Enero; 2, Febrero; etc.). El día del mes se registra en la variable entera día. Escribir un programa que compruebe si una fecha es la de su cumpleaños.

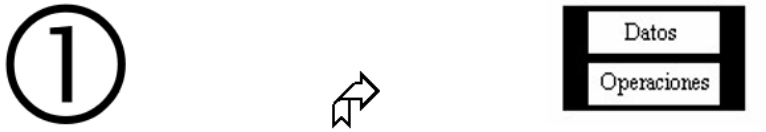
El método *main*()de la clase principal, Cumple, crea un *objeto* DiaAnyoy llama al método igual()para determinar si coincide la fecha del objeto con la fecha de su cumpleaños, que se ha leído del dispositivo de entrada.

```
import java.io.*;
class DiaAnyo {
private int mes; private int dia;

public DiaAnyo(int d, int m) {
dia = d; mes = m;
}
public boolean igual(DiaAnyo d) {
if ((dia == d.dia) && (mes == d.mes)) return
true;
else
return false; }
// muestra en pantalla el mes y el día public
void visualizar()
{
System.out.println("mes = " + mes + " , dia =
" + dia); }
}
// clase principal, con método main public
class Cumple
{
public static void main(String[] ar)throws
IOException {
DiaAnyo hoy;
DiaAnyo cumpleanyos; int d, m;
BufferedReader entrada = new
BufferedReader(
new InputStreamReader(System.in));
System.out.print("Introduzca fecha de hoy,
dia: ");
d = Integer.parseInt(entrada.readLine());
System.out.print("Introduzca el número de
mes: ");
m = Integer.parseInt(entrada.readLine());
hoy = new DiaAnyo(d,m);
System.out.print("Introduzca su fecha de
nacimiento, dia: ");
d = Integer.parseInt(entrada.readLine());
System.out.print("Introduzca el número de
mes: ");
m = Integer.parseInt(entrada.readLine());
cumpleanyos = new DiaAnyo(d,m);
System.out.print( " La fecha de hoy es ");
hoy.visualizar();
System.out.print( " Su fecha de nacimiento
es ");
cumpleanyos.visualizar();
if (hoy.igual(cumpleanyos))
        System.out.println( "¡Feliz
cumpleaños ! ");
else
        System.out.println( "¡Feliz dia ! ");
}
```

Capitulo 3

Tipos Abstractos de Datos en Java

3.1. Definición y propiedades de los TADs.



Realidad — Abstracción de la Realidad — Tipo abstracto de Datos

ABSTRACCIÓN DE DATOS = { datos, operaciones }

- Tipos de datos: Especificación del TAD
- Operaciones: Implementación del TAD

TAD = ED + Operaciones

Características de los TADs:

- ENCAPSULAMIENTO: Se desconoce la implementación de la Declaración y de las Operaciones del TAD.
- PROTECCIÓN: Sólo es posible acceder al TAD a través de las Operaciones del mismo.
- Representa una ABSTRACCIÓN: Se seleccionan ciertos datos que interesan del mundo real, ignorando el resto.
- Deben poder COMPILARSE POR SEPARADO.

Especificación del TAD: Permite al usuario la utilización del TAD.
Implementación del TAD: Sólo la conoce el programador del TAD, no el usuario.
Especificaciones
Sintácticas: Hacen referencia a aspectos Sintácticos:
Nombre de la operación, Parámetros de entrada, Parámetros de salida, Tipo de dichos parámetros, Resultado devuelto por la operación, Tipo del resultado, etc.
Semánticas: Indican el efecto producido por la operación.
Para CADA OPERACIÓN del TAD es necesario dar sus especificaciones Sintácticas y Semánticas.

La implementación de un TAD en Java se realiza deforma natural con una clase. Dentro de la clase va a residir la representación de los datos junto a las operaciones (métodos de la clase). La interfaz del tipo abstracto queda perfectamente determinada con la etiqueta *public*, que se aplicará a los métodos de la clase que representen operaciones.

Por ejemplo, si se ha especificado el TAD Punto para representar la abstracción punto en el espacio tridimensional, la siguiente clase implementa el tipo:

```
class Punto {  // representación de los datos
               private double x, y, z;
               // operaciones
```

```
public double distancia(Punto p);
public double modulo();
public double anguloZeta(); ...
};
```

3.2. Implementación del TAD Conjunto

La implementación de un TAD se realiza según la especificación realizada del tipo. La clase Conjunto implementa el TAD Conjunto, cuya especificación se encuentra en el apartado anterior.
La clase representa los datos enteros, utiliza un array para almacenar los elementos, de tipo Int.

```
package Negocio;
public class Conjunto { int C[];
                int dim;

  public Conjunto() {
     this.C = new int [10];
     this.dim = -1;
  }

 public boolean vacia(){
 return(dim==-1);
 }

 public void insertar(int ele){
 if (vacia()){
  dim++;
  C[dim]=ele;
 }else{
    if (!pertenece(ele)){
       redimencionar();
       dim++;
       C[dim]=ele;
      }
   }
 }

 public void eliminar(int ele){
 if (!vacia()){
  int i=0;
    while ((i<=dim)&&(C[i]!=ele)){
    i++;
   }
    if (i<=dim){
       for (int j = i; j < dim; j++) {
          C[j]=C[j+1];
       }
       dim--;
    }
 }
}
```

```
public boolean pertenece(int ele){
 int i=0;
 while ((i<=dim)&&(C[i]!=ele)){
   i++;
}
return(i<=dim);
}

private void redimencionar(){
if (dim==C.length-1){
 int C1[] = new int [C.length];
 System.arraycopy(C, 0, C1, 0, C.length);
 C=new int[C1.length+10];
 System.arraycopy(C1, 0, C, 0, C1.length);
}
}

public void Union(Conjunto A, Conjunto B){
   for (int i = 0; i <= A.dim; i++) {
      insertar(A.C[i]);
   }
   for (int j = 0; j <= B.dim ; j++) {
      insertar(B.C[j]);
   }
}
public void Interseccion(Conjunto A, Conjunto B){
   for (int i = 0; i <= A.dim; i++) {
      if (B.pertenece(A.C[i]))
           insertar(A.C[i]);
   }
}

   @Override
   public String toString() {
      String S="C={ ";
      for (int i = 0; i <=dim; i++) {
         S=S+C[i]+" , ";
      }
      S=S+"}";
      return S;
   }
```

```
}

public class TDAConjunto {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // TODO code application logic here
     Conjunto A=new Conjunto();
    A.insertar(2);
    A.insertar(8);
    A.insertar(2);
    A.insertar(7);
    System.out.println(A);
    A.eliminar(2);
    A.eliminar(5);
    A.insertar(3);
    System.out.println(A);
    Conjunto B=new Conjunto();
    B.insertar(2);
    B.insertar(8);
    B.insertar(21);
    B.insertar(35);
    B.insertar(3);
    System.out.println(B);
    Conjunto C=new Conjunto();
   // C.Union(A, B);+
    C.Interseccion(A, B);
    System.out.println(C);
  }
}
```

Ahora consideremos que se quiere almacenar o manipular cualquier tipo de datos en nuestro TDA conjunto para lo cual se tendría que hacer uso de algunos métodos y herramientas y declarar los tipos de datos como *Object* que nos ofrece el lenguaje para realizar esta implementación por lo que dicha implementación quedaría asi

```
package conjunto;
public class Conjunto {  static int M = 20; // aumento
de la capacidad
        private Object [] cto;
        private int cardinal;
        private int capacidad;
// operaciones

public Conjunto() {
cto = new Object[M];
cardinal = 0;
capacidad = M;
}
// determina si el conjunto está vacío
public boolean esVacio(){
return (cardinal == 0); }

// añade un elemento si no está en el conjunto
public void annadir(Object elemento){
if (!pertenece(elemento)) {
/* verifica si hay posiciones libres,en caso contrario
amplia el conjunto */
 if (cardinal == capacidad){
Object [] nuevoCto;
nuevoCto = new Object[capacidad + M];
for (int k = 0; k < capacidad; k++)
nuevoCto[k] = cto[k]; capacidad += M;
cto = nuevoCto;
System.gc();        // devuelve la memoria no
referenciada
}
cto[cardinal++] = elemento; }
}

// quita elemento del conjunto
public void retirar(Object elemento) {
if (pertenece(elemento)) {
int k = 0;
while (!cto[k].equals(elemento))
k++;
/* desde el elemento k hasta la última posición
mueve los elementos una posición a la izquierda */
for (; k < cardinal ; k++)
cto[k] = cto[k+1];
cardinal--; }
}

//busca si un elemento pertenece al conjunto
public boolean pertenece(Object elemento){
int k = 0;
boolean encontrado = false;
while (k < cardinal && !encontrado) {
encontrado = cto[k].equals(elemento);
k++;
}
return encontrado; }
```

```
//devuelve el número de elementos
 public int cardinal(){
return this.cardinal; }

//operación unión de dos conjuntos
public Conjunto union(Conjunto c2) {
Conjunto u = new Conjunto();
// primero copia el primer operando de la unión
for (int k = 0; k < cardinal; k++)
u.cto[k] = cto[k]; u.cardinal = cardinal;
// añade los elementos de c2 no incluidos for (int k =
0; k < c2.cardinal; k++)
u.annadir(c2.cto[k]); return u;
}

public Object elemento(int n) throws Exception {
if (n <= cardinal) return cto[--n];
else
throw new Exception("Fuera de rango"); }
}
```

3.3.IMPLEMENTACION DE TDAS BASICAS

3.3.1. **TDA FRACCION**

Una fracción es una expresión algebraica compuesta de un Signo, Numerador y Denominador

$$Signo\ \frac{Numerador}{Denominador}$$

Ejemplo $A=-\frac{3.2}{5}$ $B=+\frac{1}{5.2}$

Modelado Lógico del TDA Fraccion

1.- ***Nombre.****- Racional*
2.- ***Universo de datos*** *<Numerador->Reales ; Denominador->reales>*
3.- ***Relación****.-Sin relación*
4.- ***Metodos****.-*

Constructores
Nombre.- **Fraccion** ()
Descripción.-Construye una instancia de un racional
Datos Entrada.-//
Datos de salida.-//
Pre Condicion.-//denominador<>0
Post Condicion//la fracción se creo;

Nombre.- **Fraccion**
Descripción.-Construye una instancia sobrecargado de un racional con parámetro de entrada
Datos Entrada.-char signo, numerador entero, denominador entero
Datos de salida.-//
Pre Condicion.-//denominador<>0
Post Condicion//la fracción se creo;

Setters
Nombre.- **Setnumerador**
Descripción.-Modifica el Numerador del racional
Datos Entrada.- numerador entero
Datos de salida.-//
Pre Condición.-//
Post Condición.-El numerador se modifica;

Nombre.- **Setdenominador**
Descripción.-Modifica el denominador del racional
Datos Entrada.- denominador entero
Datos de salida.-//
Pre Condición.-denominador <>0
Post Condición//El denominador se modifica;

Nombre.- **Setsigno**
Descripción.-Modifica el signo del racional
Datos Entrada.- signo char
Datos de salida.-//
Pre Condición.-//
Post Condición//El signo se modifica;

Getters

Nombre.- **Getnumerador**
Descripción.-Devuelve el numerador del racional
Datos Entrada.- //
Datos de salida.- numerador Entero
Pre Condición.-//

Post Condición//

Nombre.- **Getdenominador**
Descripción.-Devuelve el denominador del racional
Datos Entrada.- //
Datos de salida.- denominador entero
Pre Condición.-//
Post Condición//

Nombre.- **Getsigno**
Descripción.-Devuelve el signo del racional
Datos Entrada.- //
Datos de salida.- signo char
Pre Condición.-//
Post Condición//

Operaciónes Auxiliares

Nombre.- **SUMA**
Descripción.-Realiza la suma de 2 racionales en el self
Datos Entrada.- Racional A, Racional B
Datos de salida.-Racional C(self)
Pre Condición.-//
Post Condición.- Se modifican los atributos del self

Nombre.- **RESTA**
Descripción.-Realiza la Resta 2 racionales en el self
Datos Entrada.- Racional A, Racional B
Datos de salida.-Racional C(self)
Pre Condición.-//
Post Condición.- Se modifican los atributos del self

Nombre.- **MULTIPLICACION**
Descripción.-Realiza la Multiplicación de 2 racionales en el self
Datos Entrada.- Racional A, Racional B
Datos de salida.-Racional C(self)
Pre Condición.-//
Post Condición.- Se modifican los atributos del self

Nombre.- **DIVISION**
Descripción.-Realiza la division de 2 racionales en el self
Datos Entrada.- Racional A, Racional B
Datos de salida.-Racional C(self)
Pre Condición.-//
Post Condición.- Se modifican los atributos del self

Nombre.- **SIMPLIFICAR**
Descripción.-Realiza la simplificación algebraica del numerador Y denominador del self
Datos Entrada.- //
Datos de salida.-//
Pre Condición.-//
Post Condición.- Se modifican los atributos del self

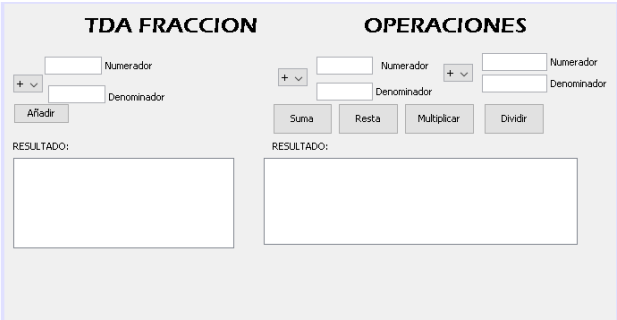
Nombre.- **GETREAL**
Descripción.-Realiza la división algebraica del racional
Datos Entrada.- //
Datos de salida.- valor Real
Pre Condición.-//
Post Condición.- //

Modelado Físico e implementación del TDA Fracción

```
package Negocios;
public class fraccion {
  int nume;
  int deno;

public fraccion(){
   this.nume=0;
   this.deno=1;
}
public fraccion(int nume,int deno,char signo){
   this.deno=Math.abs(deno);
   if (signo=='+')
      this.nume=+Math.abs(nume);
   else
      this.nume=-Math.abs(nume);
}
public int getNume(){
   return Math.abs(nume);
}
public int getDeno(){
   return deno;
}
public char getSigno(){
   if (nume>=0) {
      return '+';
   } else {
      return '-';
   }
}
public void setNume(int nume){
   if (this.nume>=0)
      this.nume=nume;
   else
      this.nume=-nume;
```

```
}
public void setDeno(int deno){
  this.deno=deno;
}
public void setSigno(char signo){
  if (signo=='+')
    nume=Math.abs(nume);
  else
    nume=-Math.abs(nume);
}
public void Suma(fraccion A,fraccion B){
  deno=A.getDeno()*B.getDeno();

nume=(A.nume*B.getDeno())+(A.getDeno()*B.nume
);
  Simplificar();
}
public void Resta(fraccion A,fraccion B){
  deno=A.getDeno()*B.getDeno();
  nume=(A.nume*B.getDeno())-
(A.getDeno()*B.nume);
  Simplificar();
}
public void Multiplicar(fraccion A,fraccion B){
  deno=A.getDeno()*B.getDeno();
  nume=A.nume*B.nume;
  Simplificar();
}
public void Dividir(fraccion A,fraccion B){
  nume=A.nume*B.deno;
  deno=A.deno*B.nume;
  if (deno<0){
    nume=nume*-1;
    deno=Math.abs(deno);}
  Simplificar();
}
public void Simplificar(){
  int x=MCD();
  nume=nume/x;
  deno=deno/x;
}
public int MCD(){
  int u=Math.abs(nume);
  int v=Math.abs(deno);
  if (v==0)
   return u;
  else{
    int z;
    while(v!=0){
    z=u%v;
    u=v;
    v=z;
    }
    return (u);
  }
}
public String ToString(){
  String c=" C= ";
  c="     "+getNume()+"\n"+c+getSigno()+" ---"+"\n
"+getDeno();
  return c;
}
public String AString(){
  String c="C= "+getSigno()+"
"+getNume()+"/"+getDeno();
  return c;
}

public static void main(String[] args) {
  fraccion A=new fraccion(1,4,'+');
   System.out.println(A.ToString());
  fraccion B=new fraccion(1,2,'-');
   System.out.println(B.ToString());
  fraccion C=new fraccion();
  C.Dividir(A, B);
  System.out.println(C.ToString());
  fraccion D=new fraccion();
  D.Multiplicar(A, B);
  System.out.println(D.ToString());

}
}
```

DESARROLLO DE LA INTERFAZ



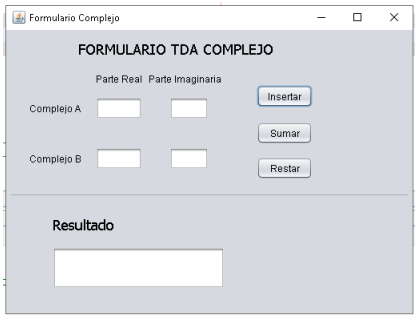
3.3.2. TDA COMPLEJO

Diseñar e implementar el TAD Complejo para representar los números complejos.
Las operaciones que se deben definir son SetPReal(asigna un valor a la parte real),SetPImag (asigna un valor a la parte imaginaria), GetParteReal (devuelve la parte real de un complejo),GetParteImaginaria (devuelve la parte imaginaria de un complejo), Modulo de un complejo y Suma, resta dedos números complejos. Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

```
package Negocio;
public class Complejo { float Preal;
                        int Pimag;
  // Construye una instacia de la class complejo
   public Complejo() {
      Preal=0;
      Pimag=0;
   }
 //Construye una instancia de la Class complejo
sobre cargado con parametos de entrada
   public Complejo(float Preal, int Pimag) {
      this.Preal = Preal;
      this.Pimag = Pimag;
   }
    // Intruduce la parte Real del complejo
   public void setPreal(float Preal) {
      this.Preal = Preal;
   }
  // Intruduce la parte Imaginaria del complejo
   public void setPimag(int Pimag) {
      this.Pimag = Pimag;
   }
   // Devuelve la parte Real de un complejo
   public float getPreal() {
      return Preal;
   }
  // Devuelve la parte Imaginaria de un complejo
   public int getPimag() {
      return Pimag;
   }
   @Override
   public String toString() {
      String S="C{";
      if (Preal>0)
         S=S+'+';
      S=S+Preal;
      if (Pimag>0)
         S=S+'+';
      S=S+Pimag+"i}";
       return S;
   }
  public void Suma(Complejo A,Complejo B){
  this.Preal=A.Preal+B.Preal;
  this.Pimag=A.getPimag()+B.getPimag();
  }
  public void Resta(Complejo A,Complejo B) {
  this.Preal=A.Preal-B.Preal;
  this.Pimag=A.Pimag-B.Pimag;
```

```
  }
}

public class Tcomplejo {
  public static void main(String[] args) {
    // TODO code application logic here
    Complejo A=new Complejo();
    A.setPreal(2);
    A.setPimag(5);
    Complejo B=new Complejo(4,-2);
    Complejo C=new Complejo();
    System.out.println(A);
    System.out.println(B);
    C.Suma(A, B);
    System.out.println(C);
    Complejo D=new Complejo();
    D.Resta(A, B);
    System.out.println(D);
  }
}
```

IMPLEMENTACION DE LA INTERFAZ



3.3.3. TDA POLINOMIO

Implementación del TDA Polinomio el cual es una expresión algebraica y colección de N Monomios para lo cual se debe realizar las operaciones básicas como insertar un monomio, eliminar un monomio, evaluar el polinomio, Suma, resta Multiplicación de polinomios etc.

Para ello necesitamos implementar el elemento mas básico de un polinomio el cual es un Monomio que esta compuesto de un Signo, Coeficiente entero y exponente entero de la siguiente Forma

$$P(x) = 7x^1 - 8x^0 + x^2$$

```
public class Monomio {
        int coef;  //El coeficiente incluye el signo
        int exp;

 public Monomio(){
   coef=0;
   exp=1;
 }

 public Monomio( char signo, int coef,int exp){
   this.exp=exp;
  if( signo=='+'){
   this.coef=Math.abs(coef);
  }else{
       if( signo=='-'){
         this.coef=-Math.abs(coef);
       }else{

         System.out.println("Error de Signo");
       }
```

```
    }
  }

public void SetCoeficiente(int  coef){
    if(coef!=0){
      if(this.coef>=0)
        this.coef=Math.abs(coef);
      else
        this.coef=-Math.abs(coef);
    }else{
      System.out.println("El coeficiente no debe ser
cero");
      System.exit(1);
    }
  }
public void SetCoeficiente1(int coe) {
   this.coef = coe;
  }

public void SetExponente( int Exp){
    if(Exp>=0){
      this.exp=Exp;
    }else{
      System.out.println("Error Exponente
Negativo");
      System.exit(1);
    }
  }
  public void setSigno( char signo){
    if (signo =='+'){
      coef=Math.abs(coef);
    }else{

      coef=-Math.abs(coef);

    }
  }
  //Getters
  public int GetCoef(){
    return (Math.abs(coef));
  }

  public int GetExpo(){
    return exp;
  }
  public char GetSigno(){
    char r;
    if(coef>0){
      r='+';
    }else{
      r='-';
    }
    return r;
  }

  public  int Evaluar(int x){
    int valor;
   //  Math.pow(base,potencia);
    if(x!=0){
      valor=(int) (coef*Math.pow(x,exp));
    }else{
      valor=0;
    }
    return valor;
  }

public void Sumar(Monomio A,Monomio B){
    if (A.GetExpo()==B.GetExpo()){
      coef=A.GetCoef()+B.GetCoef();
      exp=A.GetExpo();
    }

  }
public void Restar(Monomio A,Monomio B){
    if (A.GetExpo()==B.GetExpo()){
      coef=A.GetCoef()-B.GetCoef();
      exp=A.GetExpo();
    }

  }
  ///  5X^5+4X^2=20X^7
 public void Multiplicar(Monomio A,Monomio B){
    coef=A.GetCoef()*B.GetCoef();
    exp=A.exp+B.exp;

  }
  //  8X^7 / 2X^5 = 4X^2
 public void Division(Monomio A,Monomio B){
    if(A.exp>=B.exp){
      coef=A.coef/B.coef;
      exp=A.exp-B.exp;
    }
    System.out.println("nose puede ejecutar la
division");

  }

 public void Derivar(){
    if(exp==0){
      coef=0;
      exp=1;
    }else{
      coef=coef*exp;
      exp=exp-1;
    }
  }
```

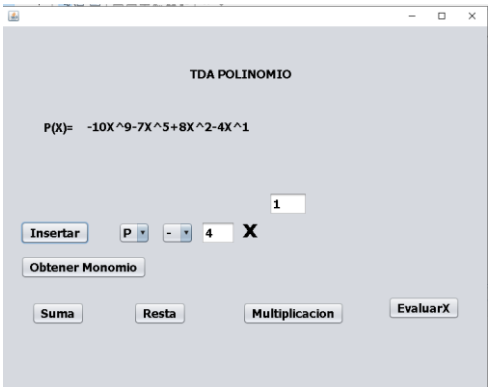
```
 public void Integral(){
    if(exp==0){
       coef=0;
       exp=1;
    }else{
       exp=exp+1;
       coef=coef/exp;

    }
  }

  public String toStringS(){
     String S=" ";
      S=S+GetSigno()+Math.abs(coef)+"X^"+exp;
          return S;
  }
  //   @Override
  public String toStringss() {
     String S=" "+GetSigno();
     if (GetCoef()!=1)
        S=S+" "+GetCoef();
     if (GetExpo()!=0)
        S=S+" X^"+GetExpo();
     return S;
  }

@Override
  public String toString(){
     String S="";
     if(coef==0 && exp==0){
        S="";
        } if(exp==0){
       S=S+GetSigno()+Math.abs(coef);
     }else{
       S=S+GetSigno()+Math.abs(coef)+"X^"+exp;
     }
     return S;
   }

   public static void main(String[] args) {
     Monomio M=new Monomio();
     M.SetCoeficiente(10);
     M.setSigno('+');
     M.SetExponente(5);
     System.out.println(M.toString());
    //    Salida  +3X^5,-3X^5
    // M.Derivar();
    //  System.out.println(M.toString());

      //System.out.println(M.Evaluar(3));
      Monomio B=new Monomio();
      B.SetCoeficiente(5);
      B.setSigno('+');
      B.SetExponente(2);
     System.out.println(B.toString());
      Monomio H=new Monomio();
     //M.Sumar(M, B);
     H.Multiplicar(M, B);
      System.out.println(H.toString());

   }
}
```

```
public class Polinomio {
     Monomio P[];
     int dim;

 public Polinomio() {
     this.P = new Monomio[10];
     this.dim = -1;
  }

public Polinomio(int cant) { // constructor
parametrizado
     dim = -1;
     P = new Monomio[cant]; // se define la dimension
del arreglo
     for (int i = 0; i < cant; i++) { // se inicializa cada
casilla del arreglo por ser un TDA compuesto
        P[i] = new Monomio('+', 0, 0);
     }
   }

public void Insertar(Monomio A){
     if (vacia()){
        dim++;
     P[dim]=A;
     }else{
        int aux=0;
     while
(aux<=dim)&&(A.GetExpo()<P[aux].GetExpo())){
           aux++;
         }
     if (aux<=dim){
          if (P[aux].GetExpo()!=A.GetExpo()){
             Redimencionar();
             for (int i = dim; i >= aux; i--){
                P[i+1] = P[i];
             }
```

```
                P[aux]=A;
                dim++;
          }else{
             int suma=(int) (A.coef+P[aux].coef);
               if (suma!=0){
                 P[aux].coef=suma;
               }else{
                    for (int j = aux; j < dim ; j++){
                      P[j]=P[j+1];
                    }
                    dim--;
                  }
        }
             } else{
                  Redimencionar();
                   dim++;
                   P[dim]=A;
             }
       }
    }

private boolean vacia(){
return (dim==-1);
}
public int cantidad(){
   int cant=0;
   for (int i = 0; i < P.length; i++) {
      cant++;
   }
   return cant;
}

  private void Redimencionar(){
 if (P.length-1==dim){
    Monomio Copia[]=new Monomio[P.length];
    System.arraycopy(P, 0, Copia, 0, P.length);
    P=new Monomio[P.length+10];
    System.arraycopy(Copia, 0, P, 0, Copia.length);}

 }

 public Monomio Getmonomio(int Exp){
 int i=0;
 while (P[i].GetExpo()!=Exp)
    i++;
 return P[i];
 }

 private int posGrado(int x){
     int i=0;
     while(P[i].exp!=x && i<dim){
        i++;
     }
     if(P[i].exp==x){
        return i;
     }else{
        return -1;
     }
  }
 public void eliminar(int grado){
  if (existeGrado(grado)){
    int p=posGrado(grado);
    if (p==dim){
       dim--;
    }else{
    for (int i=p;i<=dim;i++){
      P[i]=P[i+1];
    }
    dim--;
    }
  }

}

public int valorX(int x){
 int aux=0;
for(int i=0;i<dim;i++){
   aux=(int)
(aux+P[i].coef*(int)Math.pow(x,P[i].GetExpo()));
    }
    return aux;
 }

public void Sumar(Polinomio A, Polinomio B) {
      for (int i = 0; i <= A.dim; i++) {
         Insertar(A.obtenerElemento(i));
      }
      for (int j = 0; j <= B.dim; j++) {
         Insertar(B.obtenerElemento(j));
      }
   }

   public void Restar(Polinomio A, Polinomio B) {
      for (int i = 0; i <= A.dim; i++) {
         Insertar(A.obtenerElemento(i));
      }
      for (int j = 0; j <= B.dim; j++) {
         if(B.obtenerElemento(j).GetSigno()=='+'){
           B.obtenerElemento(j).setSigno('-');
             Insertar(B.obtenerElemento(j));
         }else{
           B.obtenerElemento(j).setSigno('+');
             Insertar(B.obtenerElemento(j));
         }
      }
   }
```

```
  @Override
  public String toString() {
     String S=" ";
     for (int i = 0; i <=dim; i++) {
        S=S+P[i];
     }
     return S;
  }

  public String toString2() {
     String S="Q(x)= ";
     for (int i = 0; i <=dim; i++) {
        S=S+P[i];
     }
     return S;
  }
 public static void main(String[] args) {
     Polinomio P=new Polinomio(4);
      Monomio B=new Monomio('-',5,8);
      P.Insertar(B);
      Monomio C=new Monomio('+',1,4);
      P.Insertar(C);
     Monomio D=new Monomio('+',3,0);
     P.Insertar(D);
      System.out.println(P.toString());
      System.out.println(p.valorX(3));
       Polinomio Q=new Polinomio(4);
       Monomio B1=new Monomio('+',5,2);
      Q.Insertar(B1);
      Monomio C1=new Monomio('+',1,4);
      Q.Insertar(C1);
      Monomio D1=new Monomio('+',3,0);
      Q.Insertar(D1);
      System.out.println(Q.toString2());
      Polinomio R=new Polinomio(4);
      R.Sumar(P, Q);
      System.out.println(R.toString());
    }

}
```

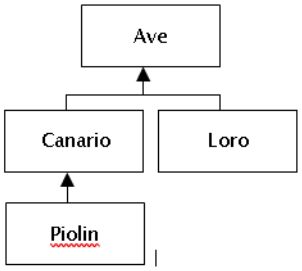
Implementación de la Interfaz



Practica 1

Realizar la modelación e implementación de los siguientes TDAs

1.- Implementar la siguiente jerarquía de clases



La clase AVE:

- Tiene como atributos de instancia el sexo (M/H), la edad (entero) y como atributo de clase el número de aves creadas, que se incrementa cada vez que se crea un nuevo Ave
- Tiene como método el constructor que inicializa, con los parámetros que recibe, los dos atributos de la clase.
- Además existe un método de clase que indica el número de Aves creadas
- Además tiene un método QuienSoy que saca los valores de los dos atributos por pantalla

La clase CANARIO:

- Tiene como atributos el sexo (M/H), la edad (entero) y el tamaño (real).
- La clase canario tiene dos constructores, uno inicializa el sexo y la edad según dos parámetros que le llegan y otro también inicializa el tamaño.
- Tiene un método Altura que dependiendo del tamaño del canario saca un mensaje por pantalla ("Alto" si tamaño > 30, "Mediano" si tamaño [15..30] y "Bajo" si tamaño < 15)

La clase LORO:

- Tiene como atributos el sexo (M/H), la edad (entero), la región (N/S/E/O) y el color (String).
- Tiene un constructor que inicializa todos los atributos con parámetros que le llegan.
- Tiene un método DeDondeEres que saca el mensaje "Norte", "Sur", "Este" u "Oeste" dependiendo de la región de donde sea el loro.

La clase PIOLIN:

- Tiene como atributos el sexo (M/H), la edad (entero) y el número de películas (entero).
- Su constructor inicializa todos los atributos con valores que le llegan como parámetros.
- En esta clase se encuentra el programa principal en el cual se crea una instancia de la clase Piolin y otro de la clase loro.
    - Llamar al método QuienSoy de las dos instancias
    - Llamar al método Altura con la instancia del Piolín
    - Invocar al método DeDondeEres con la instancia del Loro
    - Modificar el tamaño del Piolín y volver a invocar al método Altura
    - Modificar la región del Loro
    - Mostrar el número de Aves creadas.

2.- Realizar un TDA que represente la idea de "texto" que soporte 128 caracteres. Asumamos que en java no existe este tipo básico, con lo cual nuestras aplicaciones siempre quedan acopladas a la idea de que un string es una cadena de caracteres, por lo que se manejan con punteros y posiciones.
A continuación una definición básica del TDA (pueden ustedes definir más operaciones si quisieran).

tipo abstracto Texto
operacion crear() -> Texto
operacion destruir(Texto t)
operacion tamanio(Texto t) -> int
operacion caracterEn(Texto texto, int posicion) -> char
ETC.
Completar y modelar el TDA string

3.- Realizar un TDA para el tipo booleanoX en Java . asumamos que en java no existe este tipo como tipo básico, por lo cual muchas veces nuestro código pierde legibilidad y a veces el uso de números lleva a errores. En C se asume que el número 1 es verdadero y el 0 es falso.
Describimos el TDA como:
tipo abstracto boolean
operacion create(int condicionComoInt) -> boolean
operacion and(boolean b1, boolean b2) -> boolean
Pueden agregar más operaciones

4.- Realizar un TDA Fecha, abstracción de las fechas, que consisten en un año, un mes y un día y permiten abstraer de la representación interna y operar fechas:
tipo abstracto Fecha
operacion crearDesdeTextoAnioDosDigitos(char[6] fechaEnTexto) -> Fecha
operacion crearDesdeTextoAnioCompleto(char[8] fechaEnTexto) -> Fecha
operacion crear(int dia, int mes, int anio) -> Fecha
// devuelven los componentes de la fecha como números
operacion dia(Fecha f) -> int
Etc. Completar los demás Datos

5.- Implementar el TDA PERIODO cuya representación contiene el almacenamiento de un periodo en día, meses y años según sea el requerimiento.
Modelar los atributos y métodos necesarios para que este objeto sea operacional
Ejm.
// Crea un periodo especificado todo el períod en días. Ej: trabajé en la empresa 66 días
operacion crear(int dias) -> Periodo
// Crea un periodo especificado en meses + dias. Ej: trabajé en la empresa 2 meses y 6 días
operacion crear(int meses, int dias) -> Periodo
// Crea un periodo especificado en anios + meses + dias. Ej: trabajé en la empresa 2 años, 3 meses y 7 días
operacion crear(int anios, int meses, dias) -> Periodo
// Retorno la representacion del Periodo en dias: Ej: 2 meses y 3 dias -> 63 dias
operacion traducirADias(Periodo p) -> int
Nota.- Completar este TDA.

6.- Implementar el TDA Punto para representar un punto en el monitos con coordenadas x, y además de cole, tamaño, visible, etc. Y además modelar los métodos respectivos que hagan operacional es objeto como ser crear punto(), moverPunto(), ponercolor(), etc.

Nota.- Terminar el TDA completo

7.- Codificar el TDA para representar una "dupla". Una dupla es un conjunto ordenado de tamaño fijo de 2 elementos. Para acotar la implementación podrían realizar una Dupla de dos enteros (int), a fin de simplificarlo. Con lo cual no sería una Dupla genérica que podrá tener cualquier tipo de elemento.
Las operaciones definidas sobre la dupla deberán ser:
crear(int primero, int segundo) -> Dupla
primero(Dupla) -> int
segundo(Dupla) -> int
multiplicar(Dupla d, int multiplo) -> Dupla : Genera una nueva Dupla que es el resultado de multiplicar ambos elementos por el número multiplo que se pasa por parámetro.
adicionar(Dupla dupla, int adicion) -> Dupla: genera una nueva Dupla resultado de sumarle el numero dado a ambos elementos.
sumar(Dupla a, Dupla b) -> Dupla: genera una nueva dupla resultado de la suma de las dos.
restar(Dupla a, Dupla b) -> Dupla: idem anterior, pero restando de a, los valores de la dupla b
etc... (pueden acá definir más operaciones que les parezca).
Nota:
Estas definición de las operaciones, definen al tipo de dato Dupla como inmutable, es decir, cuando invocamos la función multiplicar, por ejemplo, no va a alterar la dupla original que pasamos por parámetro, si no que va a devolver una nueva. La original nunca se modifica. Ésto es una decisión, no es necesario. Podríamos haber definido operaciones que modifican la Dupla, en cuyo caso no haría falta que devuelvan nada. Eso va a "gusto" de ustedes. Hasta podrían ejercitar hacer ambas cosas.

8.- Realizar una especificación informal del TAD Conjunto rango cuyo contenido depende de un rango inicial y un rango final para determinar si los elementos son insertados en la estructura, las operaciones deberán ser mas mismas que el conjunto normal implementado anteriormente.

9.- Realizar una implementación de un TDA conjunto dinámico(simulado) que permita incrementar o decrementar la dimensión del Conjunto según vayan llenándose el vector de bits internamente al insertar elementos en dicho vector de bits del Conjunto, es decir:
El constructor del conjunto no debe tener numero de elementos
Public ConjuntoD(){

10. Construir el TAD Natural para representar los números naturales, con las operaciones: Cero, Sucesor, EsCero, Igual, Suma, Antecesor, Diferencia y Menor.
Realizar la especificación informal y formal considerando como constructores las operaciones Cero y Sucesor.

11.- Diseñar el TAD Bolsa como una colección de elementos no ordenados y que pueden estar repetidos. Las operaciones del tipo abstracto son CrearBolsa, Añadir un elemento, BolsaVacia (verifica si tiene elemento), Dentro (verifica si un elementos pertenece a la bolsa), Cuantos (determina el número de veces que se encuentra un elemento), Union y Total.
Realizar la especificación informal y formal considerando como constructores las operaciones CrearBolsa y Añadir.

12.- Diseñar el tipo abstracto de datos Matriz con la finalidad de representar matrices matemáticas. Las operaciones a definir son CrearMatriz (crea una matriz, sin elemen-tos, de mfilas por ncolumnas), Asignar (asigna un elemento en la fila i, columna j), ObtenerElemento (obtiene el elemento de la fila i, y columna j), Sumar (realiza la suma de dos matrices cuando tienen las mismas dimensiones), ProductoEscalar (obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor).

Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

13.- Implementar la clase Hora. Cada objeto de esta clase representa una hora específica del día, almacenando las horas, minutos y segundos como enteros. Se ha de incluir un constructor, métodos de acceso, una método adelantar(int h, int m, int s)para adelantar la hora actual de un objeto existente, un método reiniciar(int h, int m, int s)que reinicializa la hora actual de un objeto existente y un método imprimir()

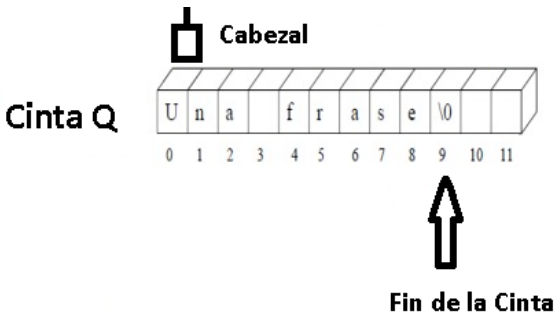
14.- Realizar la clase ComplejoReal que permita la gestión de números complejos (un número complejo = dos números reales double: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- establecer()permite inicializar un objeto de tipo Complejo a partir de dos componentes double.
- imprimir()realiza la visualización formateada de un Complejo.
- agregar()(sobrecargado) para añadir, respectivamente, un Complejo a otro y añadir dos componentes double a un Complejo.

15.- Defina un TDA fecha con miembros dato para el mes, día y año como enteros , que permita operar con fechas. Entre las operaciones características, no debería faltar una función para crear una fecha, una para devolver la próxima fecha de una fecha dada, una para devolver una fecha como un string dd/mm/aaaa, una para devolver la fecha como un string "dd de nombre del mes de aaaa", una para cambiar el día, otra para cambiar el mes y otra para cambiar el año de una fecha dada; una para recuperar el día, otra para recuperar el mes y otra para recuperar el año de una fecha dada, además un método reiniciar (int d, int m, int a)para reiniciar la fecha de un objeto existente, un método adelantar(int d, int m, int a) para avanzar a una fecha existente (dia, d; mes, m; y año a) y otro método para imprimir

Escribir un método de utilidad, normalizar(),que asegure que los miembros dato están en el rango correcto 1 ≤ año, 1 ≤ mes ≤ 12, día ≤ días (mes)donde días(Mes)es otro método que devuelve el número de días de cada mes.
Tomar en cuenta de modo que pueda aceptar años bisiestos. (Definir año bisiesto)

16.- Implementar un TDA cinta de caracteres que me permita manipular una cinta como si se tratase de un String de caracteres simulando a un reproductor de música, se deberá implementar un **constructor** que inicialice el cabezal en la posición inicial, un método **avanzar** que mueva el cabezal a la siguiente posición, un método retroceder que mueva el cabezal a una posición anterior, un método carácter Corriente **CC** que me permita obtener el carácter al cual el cabezal apunta, el método **REG** que me permita registra un carácter corriente donde esta actualmente el cabezal, un método **siguiente** palabra el cual posicione el cabezal a la siguiente posición donde encuentre un carácter en blanco o espacio, un método **retroceder** el cual posicione el cabezal a la posición anterior en blanco encontrado, un método **pausa** el cual pause el avance del cabezal además del método **stop** el cual reinicie el cabezal a la primera posición.



Capítulo 4

Bitwise: Manejo de Datos bit a bit

4.1. Operador a nivel de bits

Una **operación bit a bit** o **bitwise** opera sobre números binarios a nivel de sus bits individuales. Es una acción primitiva rápida, soportada directamente por los procesadores. En procesadores simples de bajo costo, las operaciones de bit a bit, junto con los de adición y sustracción, son típicamente sustancialmente más rápidas que la multiplicación y la división, mientras que en los modernos procesadores de alto rendimiento usualmente las operaciones se realizan a la misma velocidad.

Tipos de operaciones

Operaciones bit a bit: Ejecutan las operaciones lógicas AND, OR, XOR, NOT, etc, sobre los bits individuales de los operandos.

Operaciones de desplazamiento: Desplazan los bits de los operandos hacia la derecha o hacia la izquierda una o más posiciones.

Operaciones de rotación: Rotan los bits del operando hacia la derecha o hacia la izquierda una o más posiciones. Pueden usar o no el flag del acarreo como un bit adicional en la rotación.

4.2. Operadores bit a bit

En las explicaciones de abajo, cualquier indicación de una posición de un bit es contada de derecha a izquierda a partir del bit menos significativo. Por ejemplo, el valor binario 0001 (el decimal 1) tiene ceros en cada posición excepto en la primera.

NOT

El NOT bit a bit, o bitwise, o complemento, es una operación unaria que realiza la negación lógica en cada bit, invirtiendo los bits del número, de tal manera que los ceros se convierten en 1 y viceversa. Por ejemplo:

A	NOT *A*
0	1
1	0

```
NOT 10011
  = 01100
```

- El NOT forma el complemento a uno de un valor binario dado.
- En un número entero con signo en complemento a dos, el NOT da como resultado el inverso aditivo del número menos 1, es decir `NOT x = -x - 1`. Para obtener el complemento a dos de un número, se debe sumar 1 al resultado, dando el negativo del número. Esto equivale a un cambio de signo del número: +5 se convierte en -5, y -5 se convierte en +5.
- Para los enteros sin signo, el complemento bit a bit es la “reflexión de espejo” del número a través del punto medio del rango del entero. Por ejemplo, para los enteros sin signo de 8 bits, `NOT x =`

`255 - x`, para los enteros sin signo de 16 bits, `NOT x = 65535 - x`, y en general, para los enteros sin signo de n bits, `NOT x = (2^n - 1) - x`.

AND

El AND bit a bit, o bitwise, toma dos números enteros y realiza la operación AND lógica en cada par correspondiente de bits. El resultado en cada posición es 1 si el bit correspondiente de los dos operandos es 1, y 0 de lo contrario, por ejemplo:

A	*B*	*A* AND *B*
0	0	0
0	1	0
1	0	0
1	1	1

```
  0101
AND 0011
  = 0001
```

El AND puede ser usado para filtrar determinados bits, permitiendo que unos bits pasen y los otros no. También puede usarse en sistemas de mayor fiabilidad.

Determinando el estado de bits

El AND puede ser usado para determinar si un bit particular está encendido (1) o apagado (0). Por ejemplo, dado un patrón de bits 0011, para determinar si el segundo bit está encendido se usa una operación AND con una máscara que contiene encendido solo el segundo bit, que es el que se quiere determinar:

```
  0011
AND 0010  (máscara)
  = 0010
```

Puesto que el resultado 0010 es diferente de cero, se sabe que el segundo bit en el patrón original está encendido. Esto es a menudo llamado enmascaramiento del bit (bit masking). (Por analogía, al uso de las cintas de enmascarar, que cubren o enmascaran porciones que no deben ser alteradas o porciones que no son de interés. En este caso, los valores 0 enmascaran los bits que no son de interés).

Extrayendo bits

El AND se puede usar para extraer determinados bits de un valor. Si en un byte, por ejemplo, tenemos representados dos dígitos hexadecimales empaquetados, (uno en los 4 bits superiores y el otro en los 4 bits inferiores), podemos extraer cada dígito hexadecimal usando el AND con las máscaras adecuadas:

```
  0011 0101                    0011 0101
AND 1111 0000  (máscara)     AND 0000 1111  (máscara)
  = 0011 0000                  = 0000 0101
  Hex. superior                Hex. Inferior
```

Apagando bits

El AND también se puede usar para apagar determinados bits. Solo hay que poner una máscara con bits en cero en las posiciones de los bits que se quieren apagar y 1 en los demás bits. Todos los demás bits con la máscara 1 pasarán inalterados, y los que tienen la máscara 0 se apagarán. Dado el ejemplo 0111, el segundo bit puede ser apagado usando un AND con el patrón que tiene un cero en el segundo bit y un 1 en el resto de los bits:

```
  0111
AND 1101  (máscara)
= 0101
```

OR

Una operación OR de bit a bit, o bitwise, toma dos números enteros y realiza la operación OR inclusivo en cada par correspondiente de bits. El resultado en cada posición es 1 si el bit correspondiente de cualquiera de los dos operandos es 1, y 0 si ambos bits son 0, por ejemplo:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

```
  0101
OR 0011
= 0111
```

Encendiendo bits

El OR bit a bit, o bitwise, puede ser usado para encender un bit individual o un conjunto de bits. Para ello se usa una máscara OR con los bits que se quieren encender en 1 y el resto de los bits en cero. El resultado será que todos los bits originales quedarán como estaban excepto los bits en donde la máscara tenga 1, que resultarán encendidos. Por ejemplo, si en el patrón de bits 0101 se quiere encender el segundo bit se hará de la manera siguiente:

```
  0101
OR 0010  (máscara)
= 0111
```

Copiando bits

El OR, y el desplazamiento lógico (explicado más adelante), puede ser usado para copiar un grupo de bits a una posición determinada.

Supongamos que tenemos el signo, el exponente, y la parte significativa de un número, en diferentes registros de 32 bits, y queremos empaquetarlos para formar un número en representación de punto flotante de simple precisión de 32 bits:

```
Signo: 00000000000000000000000000000001
Exponente: 00000000000000000000000010000011

Parte significativa: 00000000111000001111000000001110
```

Todos ellos tienen los valores correctos y tenemos que mover cada uno de ellos a su posición para poder armar el punto flotante.

Se debe mover el signo 31 posiciones hacia la izquierda, el exponente 23 posiciones hacia la izquierda, y la parte significativa no es necesaria moverla porque ya está en la posición correcta. Estos desplazamientos se hacen con la operación de desplazamiento hacia la izquierda descrito más adelante:

```
      Signo: 10000000000000000000000000000000  <-- Se desplaza el signo 31 posiciones hacia la izquierda
Exponente: 010000001000000000000000000000000  <-- Se desplaza el exponente 23 posiciones hacia la izquierda
Parte significat: 00000000111000001111000000001110  <-- La parte significativa no se mueve, ya está en su lugar
```

Ahora que tenemos cada parte del número en su lugar, las combinamos para empaquetarlas y formar el número en su representación de punto flotante de 32 bits. Para ello usamos el OR: (Resultado final) = (Signo) OR (Exponente) OR (Parte significativa):

```
                Signo: 10000000000000000000000000000000
            Exponente: 010000001000000000000000000000000
Parte significativa: 00000000111000001111000000001110
  Resultado final: 11000001111100000111000000001110
```

Ya tenemos el número en su representación de punto flotante definitiva.

Procedimiento genérico para copiar un grupo de bits

Para copiar una serie de bits en un lugar determinado usando OR, se necesita que ese lugar donde se van a copiar tenga sus bits en cero (para hacer un espacio libre para poder copiar los bits). También se necesita que el registro donde se encuentran los bits que se quieren copiar tenga los demás bits (los que no se quieren copiar) apagados. Ambas operaciones, aclarar los bits en el lugar del destino, y aclarar los bits que no se quieren copiar se hacen con AND:

Tenemos dos registros de 16 bits:

```
Registro A: 1011 1100 0110 1100
Registro B: 1001 0001 1111 1010
```

Queremos copiar los cuatro bits menos significativos del registro A en el registro B.

Para ello, primero aclaramos los 4 bits menos significativos de B con una operación AND, y así tener un espacio libre:

```
  1001 0001 1111 1010  <-- Valor original del registro B
AND 1111 1111 1111 0000  <-- Máscara para aclarar los bits de B donde se van a copiar los que vienen de A
  = 1001 0001 1111 0000  <-- Registro B preparado para recibir los 4 bits menos significativos de A
```

Luego, aclaramos los bits de A que no queremos copiar, dejando solo los bits que queremos copiar:

```
  1011 1100 0110 1100  <-- Valor original del registro A
AND 0000 0000 0000 1111  <-- Máscara para dejar solo los bits de A que se quieren copiar
```

```
= 0000 0000 0000 1100   <-- Registro A con solo los bits que se desean copiar
```

Ahora estamos listos para hacer el OR de A sobre B y combinar los 4 bits menos significativos de A sobre B:

```
  0000 0000 0000 1100   <-- Registro A con los 4 bits que se desean copiar
OR 1001 0001 1111 0000   <-- Registro B con un espacio para los 4 bits que desean copiar
= 1001 0001 1111 1100   <-- Registro B con los 4 bits menos significativos de A copiados sobre él
```

Ahora, el registro B tiene copiado los 4 bits menos significativos de A. El resto de los bits de B quedaron intactos.

XOR

El XOR bit a bit, o bitwise, toma dos números enteros y realiza la operación OR exclusivo en cada par correspondiente de bits. El resultado en cada posición es 1 si el par de bits son diferentes y cero si el par de bits son iguales. Por ejemplo:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

```
  0101
XOR 0011
= 0110
```

Invirtiendo bits selectivamente

A diferencia del NOT, que invierte todos los bits de un operando, el XOR bit a bit, o bitwise, puede ser usado para invertir selectivamente uno o más bits en un registro. Dado el patrón de bits 0011, el segundo y el cuarto bit pueden ser invertidos por XOR con una máscara con un patrón de bits conteniendo 1 en las posiciones que se quieren invertir, la segunda y cuarta, y 0 en las demás. Los bits de las posiciones con cero de la máscara resultarán inalterados:

```
  0011
XOR 1010  (máscara)
= 1001
```

Igualdad y desigualdad de bits

XOR es equivalente y tiene la misma tabla de verdad que la desigualdad, XOR y desigualdad son sinónimos:

A	B	A XOR B	A <> B
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

El XOR puede usarse para saber si los bits correspondientes de dos operandos son iguales o diferentes. Por ejemplo, si tenemos dos operandos, 1000 y 0010 y queremos saber si los bits más significativos de ambos son iguales procedemos como sigue:

```
    1000
XOR 0010
  = 1010
```

Ahora, cada bit del resultado estará en 0 si el bit correspondiente de los dos operandos son iguales, y en 1 si son diferentes. El bit más significativo del resultado está en 1 indicando que son diferentes, pero tenemos que aislarlo de los demás con un AND para poder usarlo o tomar una decisión:

```
    1010  (resultado anterior)
AND 1000  (máscara para aislar el bit más significativo)
  = 1000
```

Ahora lo tenemos aislado en el resultado final, que es diferente de cero indicando que los bits más significativo de los operandos son diferentes.

Asignar cero a un registro

Los programadores avanzados de lenguaje ensamblador usan XOR como una manera eficiente y rápida de asignar cero a un registro. Realizar XOR de un valor contra sí mismo siempre resulta en cero (`A XOR A` siempre es cero), y en muchas arquitecturas esta operación requiere menos ciclos de reloj y/o memoria que cargar un valor cero a un registro (`A = 0`).

EXOR

En resumen

Las operaciones bit a bit, o bitwise, pueden encender, apagar, dejar pasar, eliminar, o invertir, bits individualmente o en conjunto, usando la máscara adecuada con un OR, AND, o XOR:

```
    0011                1011                10101               10101               1010
OR 1000 (máscara)   AND 1110 (máscara)   AND 00111 (máscara)   AND 11000 (máscara)   XOR 1001 (máscara)
  = 1011              = 1010              = 00101               = 10000               = 0011
Enciende el         Apaga el            Deja pasar los 3      Elimina los 3         Invierte los bits
bit superior        bit inferior        bits inferiores       bits inferiores       inferior y superior
```

NOT invierte los bits y XOR junto con AND permiten determinar si dos operandos tienen los bits de una determinada posición iguales o diferentes:

```
NOT 1011              11010
  = 0100          XOR 10100
invierte todos      = 01110  (0 = bit iguales, 1 = bits diferentes)
los bits          AND 00010  (se filtra el segundo bits, que es el que interesa)
                    = 00010  Determina si los bits de la segunda posición
                de los dos operandos son iguales o diferentes
                  0 = iguales
                  1 = diferentes
```

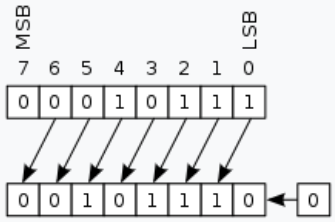
3.4. Operaciones de desplazamiento y rotación

- Las operaciones de desplazamiento y rotación son:
- Desplazamiento lógico
- Desplazamiento aritmético
- Rotación
- Rotación a través del bit de acarreo

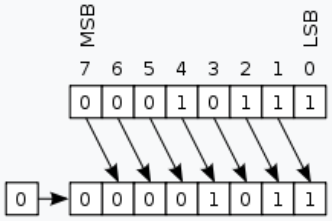
Desplazamientos de bits

Los desplazamientos de bit (bit shifts) son a veces considerados operaciones bit a bit, porque operan en la representación binaria de un número entero en vez de sobre su valor numérico; sin embargo, los desplazamientos de bits no operan en pares de bits correspondientes, y por lo tanto no pueden ser llamados propiamente como "bit a bit" (bit-wise). En estas operaciones los dígitos (bits) son movidos, o desplazados, hacia la izquierda o hacia la derecha. Los registros en un procesador de computador tienen un ancho fijo, así que algunos bits “serán desplazados hacia fuera” ("shifted out"), es decir, "salen" del registro por un extremo, mientras que el mismo número de bits son “desplazados hacia adentro” ("shifted in"), es decir, "entran" por el otro extremo; las diferencias entre los operadores de desplazamiento de bits están en cómo éstos determinan los valores de los bits que entran al registro (desplazamiento hacia adentro) (shifted-in).

Desplazamiento lógico



Desplazamiento lógico hacia la izquierda



Desplazamiento lógico hacia la derecha

Hay dos desplazamientos lógicos (logical shifts). El desplazamiento lógico hacia la izquierda (left shift) y el desplazamiento lógico hacia la derecha (right shift). En el desplazamiento lógico los bits de un registro son desplazados (movidos) una o más posiciones hacia la derecha o hacia la izquierda. Los bit que salen del registro por un extremo se pierden y en el otro extremo del registro se rellena con un bit cero por cada bit desplazado.

Por ejemplo. Si se tiene en un registro de 8 bits el valor 10110011, y se hace un desplazamiento hacia la izquierda de un bit, todos los bits se mueven una posición hacia la izquierda, el bit de la izquierda se pierde y entra un bit cero de relleno por el lado derecho. En un desplazamiento de un bit hacia la derecha ocurre algo análogo, el bit de la derecha se pierde y el de la izquierda se rellena con un cero:

```
    10110011              10110011        <-- Bits antes del desplazamiento
1 <-- 0110011 <-- 0   0 --> 1011001 --> 1   <-- Desplazamiento
   01100110              01011001          <-- Bits después del desplazamiento
  Desplazamiento        Desplazamiento
 hacia la izquierda                  hacia la derecha
```

En determinados procesadores, queda almacenado el último bit que salió con el desplazamiento del registro. En la serie de los procesadores x86 dicho bit queda almacenado en el flag del acarreo.

Moviendo bits

El desplazamiento lógico se usa para mover bits hacia la izquierda o hacia la derecha para colocarlos en la posición adecuada.

Por ejemplo, supongamos que tenemos, en dos registros del tamaño de un byte, a dos dígitos hexadecimales (en representación binaria de 4 bits cada uno), y se quiere empaquetarlos en un solo byte, donde los 4 bits superiores es el hexadecimal más significativo y los 4 bits inferiores es el hexadecimal menos significativo:

```
0000 1001  <-- Dígito hexadecimal más significativo   (hexadecimal 9)
0000 1010  <-- Dígito hexadecimal menos significativo (hexadecimal A)
```

Para empaquetarlos en un solo byte, primero hay que desplazar el hexadecimal más significativo 4 posiciones hacia la izquierda. (Esto se hace con el desplazamiento lógico hacia la izquierda):

```
1001 0000  <-- hexadecimal 9, desplazado 4 bits hacia la izquierda para colocarlo en la posición correcta dentro del byte
```

Luego, se hace un OR de los dos valores que contienen los dígitos hexadecimales para que queden combinados en un solo byte:

```
     0000 1010  <-- Hexadecimal menos significativo A
OR   1001 0000  <-- OR con el hexadecimal más significativo 9, el cual ya está en su posición
     1001 1010  <-- Byte con los dos hexadecimales empaquetados (hexadecimal 9A)
```

Ahora tenemos un byte con el valor de 1001 1010, el cual tiene los dos dígitos hexadecimales empaquetados.

Multiplicación y división por 2^n, de enteros sin signo

En números enteros sin signo, el desplazamiento lógico hacia la izquierda equivale a una multiplicación por 2 y el desplazamiento lógico hacia la derecha equivale a una división por 2. En la división (desplazamiento hacia la derecha), se pierde el bit menos significativo, dando como resultado un truncamiento del resultado (redondeo hacia abajo, hacia menos infinito). Así, 6 / 2 es igual a 3, pero 7 / 2 es igual a 3,5, pero el 0,5 se pierde quedando el resultado en 3.

Los programadores de lenguaje ensamblador usan esta propiedad para hacer multiplicaciones y divisiones rápidas, de enteros sin signo, por una potencia de 2, en donde n desplazamientos equivalen a multiplicar o dividir por 2^n. También, si el procesador no tiene operaciones de multiplicación y división de enteros, o si éstas son muy lentas, se puede multiplicar o dividir usando desplazamientos y

sumas para multiplicar y desplazamientos y restas para dividir. Por ejemplo, para multiplicar un entero por 10, se procede como sigue (en el lenguaje ensamblador del x86):

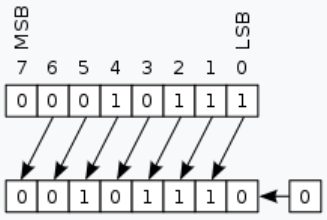
Se quiere multiplicar el contenido del registro EAX por 10:

En las instrucciones de abajo, EAX y EBX son registros del procesador, SHL (shift left), desplaza el registro indicado una posición (un bit) hacia la izquierda (que equivale a multiplicar por 2), MOV copia el registro de la derecha sobre el registro de la izquierda, y ADD suma el registro de la derecha al registro de la izquierda.

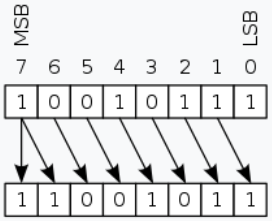
```
SHL EAX, 1     ; EAX = EAX * 2     EAX = 2n   ; desplaza a la izquierda el contenido del registro EAX una posición,
                                  ; (multiplica EAX por 2)
MOV EBX, EAX   ; EBX = EAX         EBX = 2n   ; copia el registro EAX en EBX, ahora los dos registros tienen 2n
SHL EBX, 1     ; EBX = EBX * 2     EBX = 4n   ; multiplica EBX por 2, obteniendo 4n
SHL EBX, 1     ; EBX = EBX * 2     EBX = 8n   ; vuelve a multiplicar EBX por 2, obteniendo 8n
ADD EAX, EBX   ; EAX = EAX + EBX   EAX = 2n + 8n = 10n  ; suma EBX (8n) a EAX (2n),
                                   ; (ahora EAX tiene el valor original multiplicado por 10)
```

3.5. Desplazamiento aritmético

Desplazamiento aritmético



Desplazamiento aritmético hacia la izquierda



Desplazamiento aritmético hacia la derecha

Los desplazamientos aritméticos son similares a los desplazamientos lógicos, solo que los aritméticos están pensados para trabajar sobre números enteros con signo en representación de complemento a dos en lugar de enteros sin signo. Los desplazamientos aritméticos permiten la multiplicación y la división por dos, de números enteros con signo, por una potencia de dos. Desplazar n bits hacia la izquierda o a la derecha equivale a multiplicar o dividir por 2^n, (asumiendo que el valor no hace desbordamiento (overflow o underflow)).

El desplazamiento aritmético hacia la izquierda es exactamente igual al desplazamiento lógico hacia la izquierda. De hecho son dos nombres diferentes para exactamente la misma operación. Al desplazar los bits una posición hacia la izquierda es equivalente a una multiplicación por 2 independientemente de si es un número entero con signo o sin signo. En los procesadores x86, el ensamblador tiene dos pnemónicos para el desplazamiento lógico y el aritmético hacia la izquierda, pero cuando el programa es ensamblado, solo hay un opcode para ambos en la instrucción en lenguaje de máquina.

El desplazamiento aritmético hacia la derecha es diferente al desplazamiento lógico hacia la derecha. En los enteros sin signo, para dividir por 2, se debe usar el desplazamiento lógico, el cual siempre agrega un 0 en el extremo izquierdo por cada desplazamiento de un bit hacia la derecha. En cambio, en los enteros con signo, se debe usar el desplazamiento aritmético hacia la derecha, el cual copia el bit del signo (el bit más significativo (MSB)) en el espacio vacío que queda en el extremo izquierdo cada vez que se hace un desplazamiento de un bit hacia la derecha. De esta manera, se divide efectivamente por 2 al entero con signo.

Si el entero con signo es positivo, (con el bit del signo igual a 0), se insertará el bit 0 del signo en el extremo izquierdo al desplazar un bit hacia la derecha (igual que el desplazamiento lógico hacia la derecha), pero si es un entero negativo, (con el bit del signo igual a 1), se insertará el bit 1 del bit del signo en el extremo izquierdo. De esta manera, el signo del número se preserva con la división por 2 y el número resultante tiene sentido. Si se insertara un 0 a la izquierda a un número negativo (como lo haría el desplazamiento lógico hacia la derecha), en primer lugar, este número negativo cambiaría de signo a positivo, y en segundo lugar, la interpretación de los bits restantes no tendrían sentido.

Estos ejemplos utilizan un registro de 8 bits:

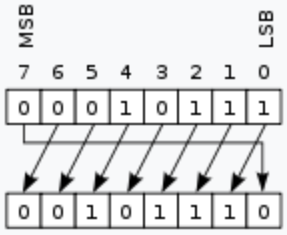
```
  00010111 (Decimal 23) (Desplazamiento aritmético hacia la izquierda de un número positivo)
= 00101110 (Decimal 46) (El bit de la izquierda se pierde y un bit 0 se añade a la derecha)

  11010111 (Decimal -41) (Desplazamiento aritmético hacia la izquierda de un número negativo)
= 10101110 (Decimal -82) (El bit de la izquierda se pierde y un bit 0 se añade a la derecha)

  00010111 (Decimal 23) (Desplazamiento aritmético hacia la derecha de un número positivo)
= 00001011 (Decimal 11) (El bit de la derecha se pierde y el bit del signo anterior se conserva en el resultado)

  11010111 (Decimal -41) (Desplazamiento aritmético hacia la derecha de un número negativo)
= 11101011 (Decimal -21) (El bit de la derecha se pierde y el bit del signo anterior se conserva en el resultado)
```

Si el número binario es tratado como complemento a 1, entonces la misma operación de desplazamiento hacia la derecha resulta en una división por 2^n redondeando hacia el cero.

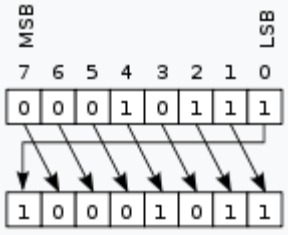
Rotación de bits

Rotación

Desplazamiento circular



Desplazamiento o rotación circular hacia la izquierda

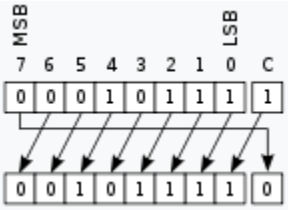


Desplazamiento o rotación circular hacia la derecha

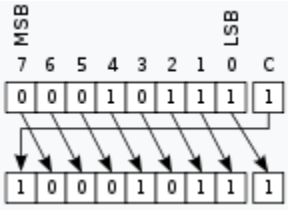
Otra forma de desplazamiento es el desplazamiento circular o rotación de bits. En esta operación, los bits de un registro son “rotados” de una manera circular como si los extremos izquierdo y derecho del registro estuvieran conectados. En la rotación hacia la izquierda, el bit que sale por el extremo izquierdo entrará por el extremo derecho, y viceversa con la rotación hacia la derecha. Esta operación

es útil si es necesario conservar todos los bits existentes, y es frecuentemente usada en criptografía digital.

Rotación a través del bit del acarreo



Rotación hacia la izquierda a través del bit del acarreo



Rotación hacia la derecha a través del bit del acarreo

Rotar a través del bit del acarreo es similar a la operación de rotar anterior (rotación sin acarreo). La diferencia está en que los dos extremos del registro están unidos entre sí a través del flag del acarreo, el cual queda en medio de ellos. El bit que sale por un extremo va al flag del acarreo, y el bit original que estaba en el flag del acarreo entra al registro por el extremo opuesto.

Si se fija el flag del acarreo de antemano, una rotación simple a través del acarreo puede simular un desplazamiento lógico o aritmético de una posición. Por ejemplo, si el flag del acarreo contiene 0, después de una rotación hacia la derecha a través del flag del acarreo, equivale a un desplazamiento lógico hacia la derecha, y si el flag del acarreo contiene una copia del bit del signo, equivale a un desplazamiento aritmético hacia la derecha. Por esta razón, algunos microcontroladores tales como los PIC solo tienen las funciones de rotar y rotar a través del acarreo, y no se preocupan de tener instrucciones de desplazamiento aritmético o lógico.

Rotar a través del acarreo es especialmente útil cuando se hacen desplazamientos en números más grandes que el tamaño nativo de la palabra del procesador, porque si, por ejemplo, un número grande es almacenado en dos registros y se quiere desplazar hacia la derecha un bit, el bit que sale del extremo derecho del registro de la izquierda debe entrar por el extremo izquierdo del registro de la derecha. Con rotación a través del acarreo, ese bit es “almacenado” en el flag del acarreo durante el primer desplazamiento hacia la derecha sobre el registro de la izquierda, listo para ser desplazado al registro de la derecha usando una simple rotación con acarreo hacia la derecha y sin usar ninguna preparación extra.

3.6. REPRESENTACION DE BITWISE SEGÚN LAS NECESIDADES

Si bien la implementación de bitwise nos ayudara a mejorar la representación de diversas Estructuras de datos tanto para el ahorro en memoria como asi también para mejorar el proceso podemos clasificar 2 tipos de representaciones para implementar Estructuras de Datos a nivel de Bits según sean las necesidades las cuales clasificamos en representación basada en posición y representación basada en Valor.

3.6.1. REPRESENTACION BASADA EN POSICION

Esta representación se basa fundamentalmente en representar los datos mediante el encendido o apagado de un bit en una determinada posición de una Estructura que maneja bits en la cual dicha posición indicara la presencia o ausencia del dato expresado en la estructura de datos a nivel de bits.

Esta representación es útil cuando los datos que se van a representar con únicos y carecen de duplicidad ya que la posición a la cual representara es única por lo que solo sirve parara representaciones con datos Únicos y de Dominio conocido, Ejm. La representación de una agenda de días, una estructura que represente un cartón de Loto, bingo, un conjunto de enteros o chars.

A continuación implementaremos el TDA bitwise la cual es la base fundamental para las posteriores estructuras de Datos

TDA BITWISE

Se desea implementar un TDA bidwise que me permita manipular un entero a nivel de bits, donde pueda encender o apagar un determinado numero de bit además de devolver el estado en el que se encuentra un determinado de bit, como asi también implementar el método toString el cual me mostrara los bits de manera visual

```
package Negocio;
public class Bitwise {  int x;

public Bitwise() {
     this.x = 0;
  }

public void Setbit1(int pos){
     if (pos<=32){
     int mask=1;
     mask=mask<<pos-1;
     x=x|mask;}
}

public void Setbit0(int pos){
     if (pos<=32){
     int mask=1;
     mask=mask<<pos-1;
     mask=~mask;
     x=x&mask;
     }
}

public int Getbit(int pos){
     int mask=1;
       mask=mask<<pos-1;
       mask=mask&x;
       mask=mask>>>pos-1;
       return(mask);
}

    @Override
    public String toString() {
      String S="X= ";
       for (int i = 32; i >=1; i--) {
          S=S+" "+Getbit(i);
       }
     return(S);
    }
}
public static void main(String[] args){
     Bitwise P=new Bitwise();
     System.out.println(P);
     P.Setbit1(1);
     P.Setbit1(2);
     P.Setbit1(3);
     P.Setbit1(4);
     System.out.println(P);
    P.Setbit0(2);
    P.Setbit0(1);
```

```
  P.S etbit0 (32);
  P.S etbit0 (31);
  P.Setbit1(32);
   System.out.println(P);
  }
```

TDA Conjunto Bitwise

En base a los requerimientos del capitulo anterior, implementar un TDA conjunto Bits para manipular los datos a Nivel de Bits

```
public class Conjuntob { Bitwise C[];
                         int cant;

  public Conjuntob(int NEle) {
    int Nbw=NEle/32;
    if(NEle%32!=0){
    Nbw++;
    }
    this.C=new Bitwise[Nbw];
    cant=NEle;
    for (int i = 0; i <= Nbw-1; i++) {
      C[i] = new Bitwise();

    }
  }

  public void Insertar(int ele){
      if(ele<=cant){
      int NBW=CalcNBW(ele);
      int Nbit=CalcNbit(ele);
      C[NBW].Setbit1(Nbit);
  }
  else {
System.out.println("Error::ConjuntoBit:Eliminar:Ele
mento fuera de Rango");
  System.exit(1);
  }
  }

  public void Eliminar(int ele){
      if(ele<=cant){
      int NBW=CalcNBW(ele);
      int Nbit=CalcNbit(ele);
      C[NBW].Setbit0(Nbit);
  }
  else {
System.out.println("Error::ConjuntoBit:Eliminar:Ele
mento fuera de Rango");
  System.exit(1);
  }
  }


private int CalcNBW(int ele){
  return((ele-1)/32);
}

private int CalcNbit(int ele){
return(((ele-1)%32)+1);
}

public boolean Pertenece(int ele){
  int NBW=CalcNBW(ele);
  int Nbit=CalcNbit(ele);
return(C[NBW].Getbit(Nbit)==1);
}

  @Override
  public String toString() {
    String S="C={";
    for (int i = 1; i <=cant; i++) {
      if (Pertenece(i))
        S=S+i+" , ";
    }
    S=S+"}";
     return S;
  }
}
```

3.6.2.REPRESENTACION BASADA EN VALOR

Esta se basa en representar los datos a nivel de bit según su valor en una determinada subposicion del vector de bits, bajo determinada cantidad de bits necesarios para representar un rango de datos a nivel de bits, esta representación a diferencia de la anterior si es posible expresar datos repetidos almacenados en una estructura a nivel de bits ya que lo que se representa es el valor y no así la posición.

Bajo este esquema será necesario implementar un vector de bits para almacenar los datos a nivel de bits con una cierta cantidad de bits según sean los requisitos del dato por lo que podemos afirmar que este vector de bits será tratado como un array de enteros el cual se insertara el dato en juna cierta posición pero que en la practica ese dato estará expresado a nivel de bits.

TDA VECTOS DE BITS

```
package Negocio;
public class VectorNbits {
                            int v[];
                              int cant;
                            int CantidadBits;

  public VectorNbits(int NumElementos, int
CantBits) {
    int NumBits = NumElementos * CantBits;
    int NumEnteros = NumBits / 32;
    if ((NumBits % 32) != 0) {
      NumEnteros++;
    }
    v = new int[NumEnteros];
    cant = NumElementos;
    CantidadBits = CantBits;
  }

  public void insertar(int ele, int pos) {
    if (pos <= cant) {
      int ele1 = ele;
      int mask = (int) Math.pow(2, CantidadBits) - 1;
      int NumBits = CalcularBits(pos);
      int NumEntero = CalcularEntero(pos);
      mask = mask << NumBits;
      mask = ~mask;
      v[NumEntero] = v[NumEntero] & mask;
      ele = ele << NumBits;
      v[NumEntero] = v[NumEntero] | ele;
      if ((NumBits + CantidadBits) > 32) {
        int mask1 = (int) Math.pow(2, CantidadBits) -
1;
        mask1 = mask1 >>> (32 - NumBits);
        mask1 = ~mask1;
        v[NumEntero + 1] = v[NumEntero + 1] &
mask1;
        ele1 = ele1 >>> (32 - NumBits);
        v[NumEntero + 1] = v[NumEntero + 1] | ele1;
      }
    }
  }

  public int sacar(int pos) {
    int mask = (int) ((Math.pow(2, CantidadBits) - 1));
    int NumBits = CalcularBits(pos);
    int NumEntero = CalcularEntero(pos);
    mask = mask << NumBits;
    mask = mask & v[NumEntero];
    mask = mask >>> NumBits;
    if ((NumBits + CantidadBits) > 32) {
      int mask1 = (int) ((Math.pow(2, CantidadBits) -
1));
      mask1 = mask1 >>> (32 - NumBits);
      mask1 = mask1 & v[NumEntero + 1];
      mask1 = mask1 << (32 - NumBits);
      mask = mask | mask1;
    }
    return (mask);
  }

  private int CalcularBits(int pos) {
    return (((pos - 1) * CantidadBits % 32));
  }

  private int CalcularEntero(int pos) {
    return ((pos - 1) * CantidadBits / 32);
  }

  @Override
  public String toString() {
    String S = "V=[ ";

    for (int i = 1; i <= cant; i++) {

      S = S + sacar(i) + " , ";
```

```
    }

    S = S.replaceAll(", $", "");

    S = S + "]";

    return (S);

  }

}
```

```
public static void main(String[] args) {
    // TODO code application logic here
    VectorNbits x=new VectorNbits(10,6);
    x.insertar(33,1);
    x.insertar(22, 3);
    x.insertar(58, 7);
    x.insertar(35, 10);
    x.insertar(8, 6);
    x.insertar(34, 5);
    x.insertar(18, 4);
    System.out.println(x);
    x.insertar(55,4);
    System.out.println(x);
}
```

En Base a este Vector de Bits podemos implementar un TDA baso a nivel de bits para representar cualquier tipo de Estructura como es el caso de un TDA Ficha covid que me permita almacenar y registrar Datos acerca de una persona que se esta vacunando del Covid con atributos como ser Nombre, C.I., fecha de nacimiento, edad, sexo, Departamento, municipio, Lugar de vacunación, Tipo de vacuna, Provedor, Lote, Fecha de Vacunacion, etc

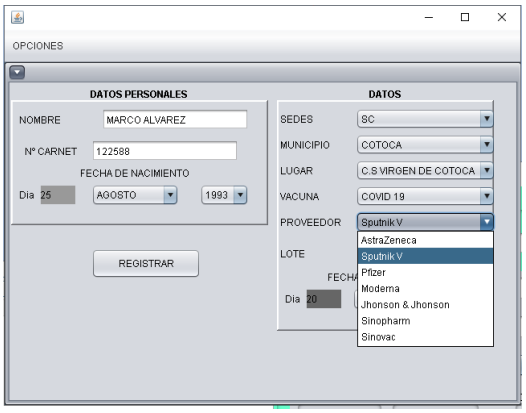
A continuación se detalla el TDA Certificado de ficha Covid

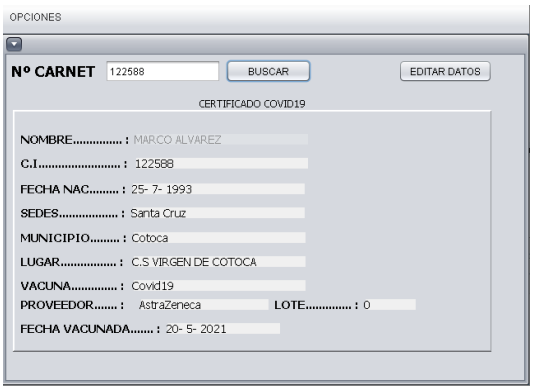
```
public class certificado {
VectorNbits c[];
int i;
public int bus;
String vnombre[];

public certificado(int ele){
c=new VectorNbits [3];
c[0]=new VectorNbits (ele,24);
c[1]=new VectorNbits (ele,31);
c[2]=new VectorNbits (ele,15);
vnombre=new String[ele];
i=0;
}

public void insertar(int ci,int d,int m,int a,int s,int
mun,int lug,int vac,int p,int dv,int mv,int lot,int
av,String nombre){
  i++;
  c[0].insertar(ci, i);
  vac<<=3;
  vac|=p;
  lug<<=6;
  lug|=vac;
  mun<<=9;
  lug|=mun;
  s<<=11;
  lug|=s;
  d<<=4;
  d|=m;
  a<<=9;
  d|=a;
  d<<=15;
  lug|=d;
  c[1].insertar(lug,i);
  lot<<=3;
  lot|=av;
  mv<<=6;
  lot|=mv;
  dv<<=10;
  lot|=dv;
  c[2].insertar(lot, i);
  vnombre[i-1]=nombre;
}

public int getCi   (int p){
return c[0].sacar(p);
}



public int getDia(int n){
int mask=((int)Math.pow(2, 5))-1;
```

```
  mask<<=19;
   int x=c[1].sacar(n);
  mask&=x;
  mask>>>=19;
  return mask;
  }

 public int getMes(int n){
  int mask=((int)Math.pow(2, 4))-1;
  mask<<=15;
   int x=c[1].sacar(n);
  mask&=x;
  mask>>>=15;
  return mask;
  }

 public String getNombre(int pos){
      String nam=vnombre[pos-1];
 return nam;
 }

  public int getAño(int n){
  int x=c[1].sacar(n);
  x=x>>>24;
  return (2003-x);
  }

public String getSedes(int n){
  int mask=((int)Math.pow(2, 4))-1;
  mask<<=11;
  int x=c[1].sacar(n);
  mask&=x;
  mask>>>=11;String t="";
  switch(mask){
     case 0: t="Santa Cruz";break;
     case 1: t="Beni";break;
     case 2: t="Pando";break;
     case 3: t="Tarija";break;
     case 4: t="Potosi";break;
     case 5: t="Oruro";break;
     case 6: t="Cochabamba";break;
     case 7: t="La Paz";break;
     case 8: t="Chuquisaca";break;
 }
  return t;
  }



 public String getMuni(int n){
  int mask=((int)Math.pow(2, 2))-1;
  mask<<=9;
  int x=c[1].sacar(n);
  mask&=x;
  mask>>>=9;
  String t="";
  switch(mask){
     case 0: t="Santa Cruz";break;
     case 1: t="Cotoca";break;
     case 2: t="La guardia";break;
 }
  return t;
  }

public String getLugar(int n){
  int mask=((int)Math.pow(2, 3))-1;
  mask<<=6;
  int x=c[1].sacar(n);
  mask&=x;
  mask>>>=6;String t="";
  switch(mask){
     case 0: t="C.S LAS AMERICAS";break;
     case 1: t="UPDS";break;
     case 2: t="EMI";break;
     case 3: t="C.S VIRGEN DE COTOCA";break;
     case 4: t="C.S PLAN 4000";break;
     case 5: t="FEXPOCRUZ";break;
     case 6: t="C.S EL RECREO";break;
 }
  return t;
  }

public String getVacuna(int n){
  int mask=((int)Math.pow(2, 3))-1;
  mask<<=3;
  int x=c[1].sacar(n);
  mask&=x;
  mask>>>=3;String t="";
  switch(mask){
     case 0: t="Covid19";break;
     case 1: t="Fiebre Amarilla";break;
     case 2: t="Sarampion";break;
 }
  return t;
  }

public String getProveedor(int n){
  int mask=((int)Math.pow(2, 3))-1;
  int x=c[1].sacar(n);
  mask&=x;
  String t="";
  switch(mask){
     case 0: t="AstraZeneca";break;
     case 1: t="Sputnik V";break;
     case 2: t="Pfizer";break;
     case 3: t="Moderna";break;
```

```
      case 4: t="Jhonson & Jhonson";break;
      case 5: t="Sinopharm";break;
      case 6: t="Sinovac";break;
  }
    return t;
    }

  public int getdiaVacu(int n){
    int x=c[2].sacar(n);
    x=x>>>10;
    return x;
    }

  public int getMesVacu(int n){
    int mask=((int)Math.pow(2, 4))-1;
    mask<<=6;
     int x=c[2].sacar(n);
    mask&=x;
    mask>>>=6;
    return mask;
    }

  public int getLote(int n){
    int mask=((int)Math.pow(2, 3))-1;
    mask<<=3;
     int x=c[2].sacar(n);
    mask&=x;
    mask>>>=3;
    return mask;
    }

  public int getAñoVacu(int n){
    int mask=((int)Math.pow(2, 3))-1;
     int x=c[1].sacar(n);
    mask&=x;
    return mask+2021;
    }

    public boolean pertenece(int ci){
    boolean x=false;int j=1;
          while (x==false && j<=this.i) {
            int sa=c[0].sacar(j);
          int mask=((int)Math.pow(2, 24))-1;
          mask&=sa;
          if (ci==mask){x=true;bus=j;}
          j++;
          }
      return x;
    }
  }
```

La Interfaz Respectiva Seria Así





Practica 2

1.- Realizar una implementación de un TDA conjunto dinámico(simulado) que permita incrementar o decrementar la dimensión del Conjunto según vayan llenándose el vector de bits internamente al insertar elementos en dicho vector de bits del Conjunto, es decir:
El constructor del conjunto no debe tener numero de elementos
Public ConjuntoD(){
}

2.- Realizar una implementación de un TDA Conjunto Rango a nivel de bits cuya particularidad es que solo es capaz de insertar un número determinado de elementos desde un rango inicial hasta un rango final, es decir:
Si Rango Inicial =20
Rango Final=80
Se debería insertar elementos comprendidos entre 20 y 80
Ejemplo
Insertar (30) si
Insertar (55) si
Insertar (45) si
Insertar (84) no
Insertar (70) si
Insertar (15) no
El conjunto resultante será
C=[30,55,45,70]

3.- Se desea implementar una ED de la mejor manera representada que almacene y procese los sgtes datos

FORMULARIO DE ANALISIS COVID

Nro:_____
Nombre:_____
Edad:_____ (de 1 a 99 años)
Sexo:_____M F
Peso:__90.5_____(de 1 hasta 120 kilos) con 1 decimal

SINTOMAS
Seleccione si tiene uno de los sgtes síntomas

FIEBRE	☐	TOS	☐	DOLOR DE GARGANTA	☐
DOLOR DE CABEZA	☐	FALTA DE RESPIRACION	☐	DIARREA	☐
DOLOR MUSCULAR	☐	PERDIDA DE OLFATO	☐	PERDIDA DEL GUSTO	☐

Nota.- Se desea almacenar estos datos de para 100 personas

4.- Realizar la Implementación del TDA Polinomio del anterior capitulo a nivel de bits

5.- Realizar la implementación de TDA fracción a nivel de Bits3

6.- Realizar la Implementación del TDA Fecha a nivel de Bits

7.- Realizar la Implementación del TDA Hora a nivel de Bits

8.- Se desea implementar una TDA polinomio Real a nivel de bits la cual contenga la siguiente información

$$P(X)= -\frac{3}{2}x^{\frac{1}{3}} + \frac{5}{8}x^{\frac{3}{4}} \ldots\ldots\ldots$$

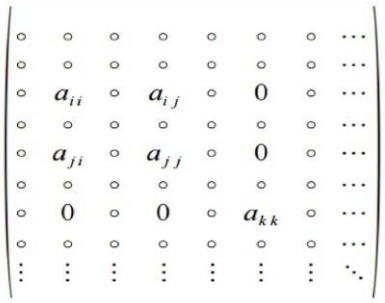
Donde el Coeficiente y exponente serán expresados en fracción

Capitulo 5

MATRIZ ESPARCIDA(Sparse)

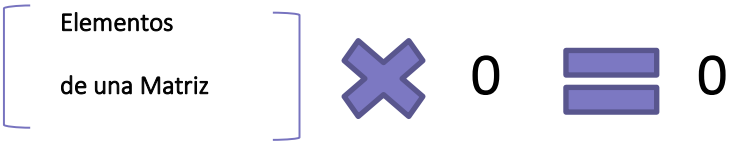
5.1. Definición

Se dice que una matriz es dispersa cuando se puede hacer uso de técnicas especiales para sacar ventaja del gran número de elementos ceros o predominantes que posee.



5.2. Objetivos

- Reducir requerimientos de memoria:
- En una matriz común, las casillas que no están siendo ocupadas por un elemento contienen un null, pero aun así se está reservando memoria para alojar este dato. Una matriz dispersa soluciona este problema usando encadenamientos que no necesitan reservar memoria para las posiciones de la matriz que son vacías.
- Podemos utilizar las matrices dispersas, que contienen tanta información como las matrices de adyacencia, pero, en principio, no ocupan tanta memoria como las matrices, ya que al igual que en las listas de adyacencia, sólo representaremos aquellos enlaces que existen en el grafo.
- Reducir el coste aritmético de las operaciones.
  (no hay que multiplicarlos)



5.3. Representaciones

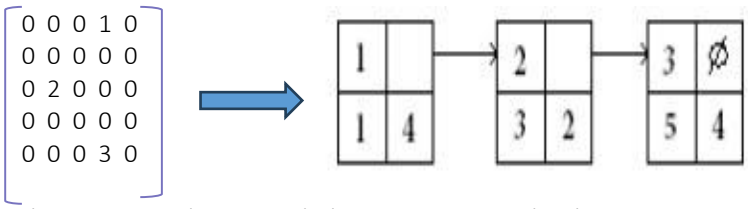
Para representar una matriz esparcida existen diferentes representación según sea la necesidad y las características de la Estructura a Representar, dentro de estas podemos nombrar a

5.3.1. Representación Mediante Lista enlazada

Con matrices de gran tamaño los métodos tradicionales para almacenar la matriz en la memoria de una computadora o para la resolución de sistemas de ecuaciones lineales necesitan una gran cantidad de memoria y de tiempo de proceso. Se han diseñado algoritmos específicos para estos fines cuando las matrices son dispersas, uno de ellos es la lista enlazada.

Lista enlazada

En la Figura se observa una matriz de 5x5 en donde solo 3 de sus elementos son diferentes de cero

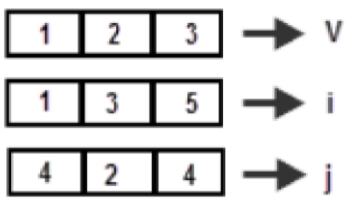


Cada nodo almacena un elemento de la matriz, se puede observar que se tienen punteros de un nodo al siguiente, además de la información de la fila y la columna en la que se encuentra en dicha matriz.

5.3.2. Representación Mediante Arrays o vectores

El consiste en almacenar la misma información que se guardaba en el método de listas enlazadas, pero esta vez con 3 arreglos estáticos.

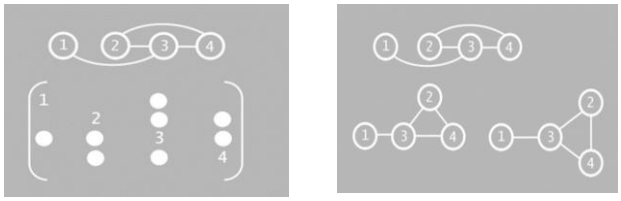
Para la anterior matriz se tendría entonces los siguientes vectores



En el primer arreglo se almacenarán todos los datos no nulos de la matriz dispersa, por lo tanto el tamaño del vector dependerá de la cantidad de valores no nulos que tenga la matriz dispersa. En el segundo vector se almacenara la información pertinente para la fila que contiene dicho dato, y por último en el tercer vector estará almacenada la información que tiene que ver con el valor de la columna del dato.

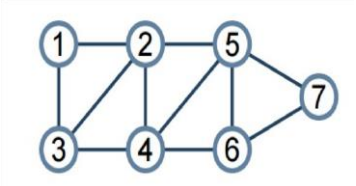
5.3.3. Representación Mediante Grafos

Una matriz dispersa con estructura simétrica se puede representar como un grafo no dirigido. En el grafo no están representados los valores de la matriz, sólo la estructura de las entradas distintas de cero .



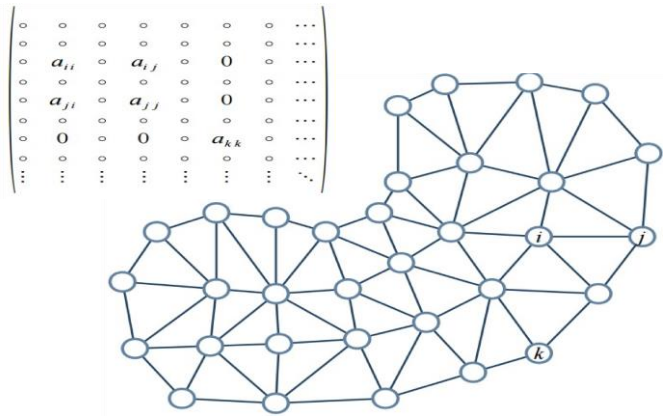
Para una matriz cuadrada A, se asocia un nodo con cada fila. Si aij es un elemento no nulo (entrada) de una matriz dispersa, hay una arista dirigida del nodo i al j.

- Matrices dispersas de mallas de elemento finito



En el método de elemento finito, se trabaja de forma contraria, se parte de la malla (grafo) y se genera la matriz dispersa.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & 0 \\ 0 & a_{52} & 0 & a_{54} & a_{55} & a_{56} & a_{57} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} & a_{67} \\ 0 & 0 & 0 & 0 & a_{75} & a_{76} & a_{77} \end{pmatrix}$$



Dado que un nodo de la malla se conecta sólo con pocos nodos, tendremos una matriz muy dispersa. En general, el tamaño del número de entradas distintas de cero será O(n).

REPRESENTACIÓN GRAFICA

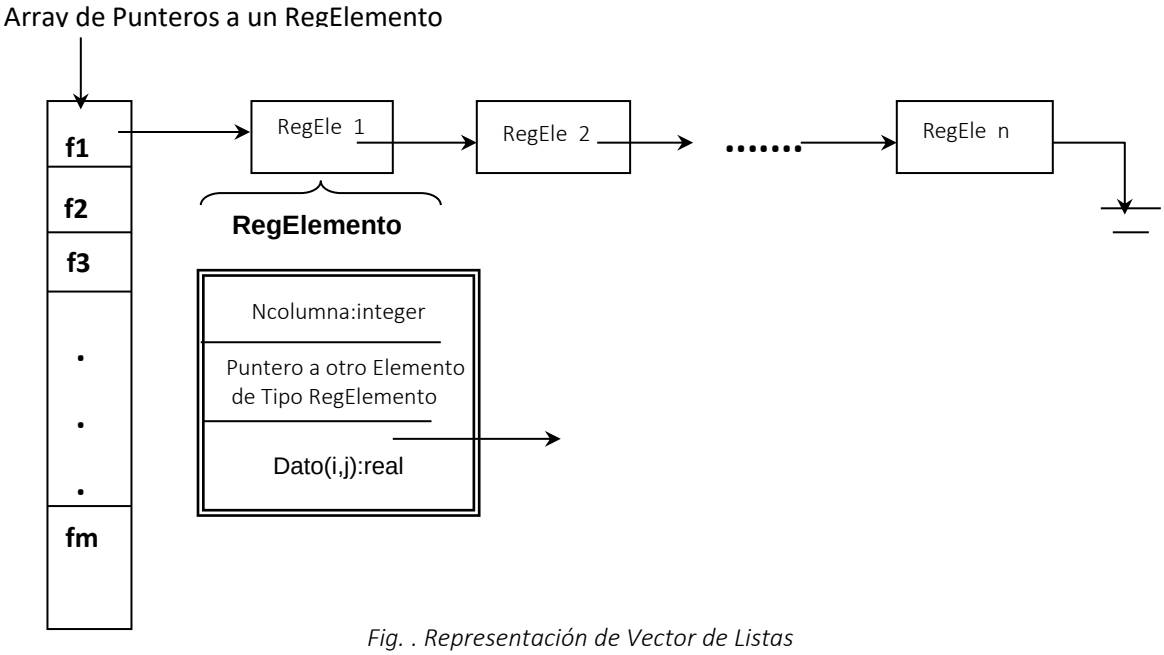


Fig. . Representación de Vector de Listas

5.3.4. Representación Optimizada mediante Arrays

Al ser una **matriz esparcida** una estructura de datos que representa matrices de cualquier orden. La ventaja es que esta matriz ocupa menor espacio en memoria de lo que una matriz normal pueda pesar, la matriz en si, tiene también todas la propiedades que una matriz normal tiene, y se maneja de la misma manera que una matriz normal.

Esta es una imagen de una matriz:

2	2	8	5
5	8	4	6
1	3	6	2

Como atributos tiene: filas, columnas, elementos.

Para hacer el diseño de la matriz esparcida

1. Se analiza el elemento que mas veces se repite en una matriz, para el caso de la matriz anterior es el 2, este elemento se lo considera como elemento dominante.
2. Luego se ve el dominio de los elementos de la matriz, para el caso de la matriz anterior el dominio son los numeros naturales.

La matriz esparcida evita guardar el elemento dominante, y solo guarda todos los elementos que no sean dominantes, para esto necesitamos guardar cada elemento no dominante, y su posicion dentro de la matriz, entonces, se puede deducir una formula para mantener la posicion de cada elemento.

2	2	8	5
5	8	4	6
1	3	6	2

Observando las posicion de los elementos no dominantes:

- El elemento 8 se encuentra en la fila 1 y columna 3 y tambien en la fila 2 columna 2.
- El elemento 5 se encuentra en la fila 1 y columna 4 y tambien en la fila 2 columna 1.
- El elemento 4 se encuentra en la fila 2 y columna 3.
- El elemento 6 se encuentra en la fila 2 y columna 4 y tambien en la fila 3 columna 3.
- El elemento 1 se encuentra en la fila 3 y columna 1.
- El elemento 3 se encuentra en la fila 3 y columna 2.

Una ves conseguido los datos necesarios la matriz esparcida se realiza de la siguiente manera:

Realización de la matriz esparcida

Se utiliza un vector de datos para almacenar todos los elementos de la matriz, cada elemento tiene almacenado su posicion en un vector de posiciones.

El vector de posiciones almacena la posicion de cada elemento obedeciendo a la siguiente formula:

$$(F - 1) * N + C$$

Donde: F -> fila del elemento.

C -> columna del elemento.

N - > numero de columnas que tiene la matriz normal.

Deduccion de la formula (F – 1) * N + C:

Si contamos los elementos de la matriz como si fuera un vector de izquierda a derecha y cuando llegamos a la ultima columna volvemos a empezar en la siguiente fila se tiene la siguiente enumeracion.

2 **1**	2 **2**	8 **3**	5 **4**
5 **5**	8 **6**	4 **7**	6 **8**
1 **9**	3 **10**	6 **11**	2 **12**

Entonces si lo tomamos como si fuese un vector normal, decimos que el elemento 4 esta en la posicion 7, el elemento 5 esta en la posicion 4 y 5.

El usuario ve la enumeracion de una matriz desde la posicion 1,1 , pero un programador sabe que en la computadora las posiciones de una matriz comienzan en 0,0.

Tomando en cuenta esta consideracion entonces formamos la formula.

(F – 1) * N + C

F-1 , porque la fila que introduce el usuario comienza su conteo desde 1, pero en programacion las filas comienzan en 0, entonces le restamos 1 a la fila introducida.

(F – 1) *** N** , de esta manera sabemos cuantas posiciones recorrimos desde la fila 0 hasta la fila donde esta nuestros elemento al que nos estamos refiriendo.

(F – 1) * N **+ C,** al sumar C, obtenemos exactamente la posicion donde se encuentra nuestro elemento.

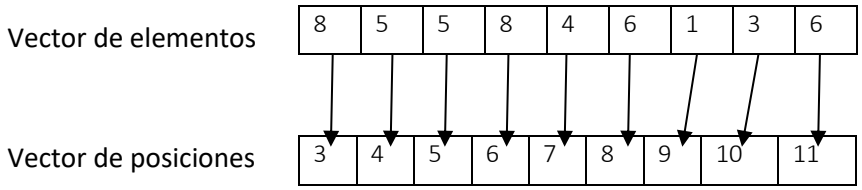
Ejemplos.

Tenemos el elemento 4 en la fila 2 y columna 3, veamos si funciona la formula.

(F – 1) * N + C
F = 2
N = Numero de columnas de la matriz normal = 4
C = 3.
(2 – 1) * 4 + 3 = 7.
Entonces el elemento cuatro esta en la posicion 7.

Siguiendo con la realización

Hasta ahora tenemos dos vectores, donde en un vector almacenamos los elementos que no son dominantes y en el otro vector guardamos las posiciones respectivas para cada elemento:



En la grafica podemos ver que cada elemento tiene su correspondiente posicion almacenada en el vector de posiciones, el orden si importa.

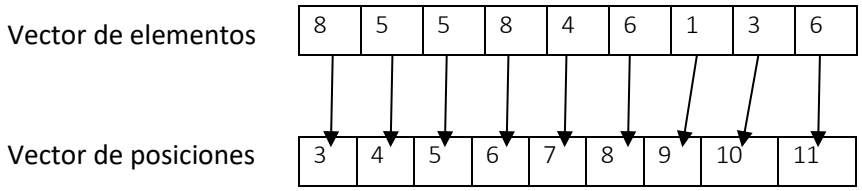
Analizando un poco podemos ver que en el vector de posiciones su dominio siempre seran numeros naturales, sin embargo, para el vector de elementos podemos tener diferentes dominios, por ejemplos, caracteres, cadenas, objetos de cualquier tipo.

Siguiendo con la implementacion de una matriz esparcida, hasta ahora tenemos dos vectores; necesitamos ademas de los vector almacenar el elemento dominante, para poder salvar de esta manera el valor que corresponde al elemento dominante.

Finalizando tenemos estos atributos:

- Vector de elementos.
- Vector de posiciones.
- Elemento dominante.

Veamos ahora como trabajamos con la matriz esparcida:



Elemento dominante -> 2

Ejemplos

- ¿Que elemento esta en la fila 3 y columna 2?

  Reemplazando estos valores en la formula:

  (F – 1) * N + C
  F = 3
  N = Numero de columnas de la matriz normal = 4
  C = 2.
  (3 – 1) * 4 + 2 = 10.
  Entonces buscamos el elemento 10 en nuestro vector de posiciones.

3	4	5	6	7	8	9	**10**	11

  El 10 esta en el vector, y esta en la posicion 8 del vector, entonces accedemos a la posición 8 del vector de elementos.

8	5	5	8	4	6	1	**3**	6

  En la posición 8 esta el elemento **3**, entonces el elemento que se encuentra en la fila 3, columna 2 es el elemento **3**.

- ¿Que elemento esta en la fila 1 y columna 2?

  Reemplazando estos valores en la formula:

  (F – 1) * N + C
  F = 1
  N = Numero de columnas de la matriz normal = 4
  C = 2.
  (1 – 1) * 4 + 2 = 2.
  Entonces buscamos el elemento 2 en nuestro vector de posiciones.

3	4	5	6	7	8	9	10	11

  El 2 no esta en el vector de posiciones, entonces quiere decir que en esa fila y columna se encuentra un elemento dominante.
  Entonces el elemento que se encuentra en la fila 1, columna 2 es el elemento dominante, ósea el **2**.

Modificando los elementos de la matriz esparcida

Cuando el usuario de la matriz esparcida decide cambiar los elementos que hay en dicha matriz, osea quiere poner otro valor en una fila y columna dada, se pueden dar 4 casos:

1. Que el usuario quiera meter un elemento dominante a una celda donde hay un elemento que no sea dominante.
2. Que el usuario quiera meter un elemento dominante a una celda donde hay un elemento dominante.
3. Que el usuario quiera meter un elemento no dominante a una celda donde hay un elemento que no sea dominante.

4. Que el usuario quiera meter un elemento no dominante a una celda donde hay un elemento que dominante.

Entonces analizamos que tenemos que hacer internamente en nuestros vectores para que la matriz funcione de manera correcta.

Veamos para cada caso correspondientemente:

1. Dominante → No Dominante.

   Se debe eliminar el elemento que se encuentre en esa posicion, de esta manera cuando el usuario acceda a dicha posicion, el elemento no se encontrara en los vectores, y como no se encuentra, el resultado sera el elemento dominante.

2. Dominante → Dominante.
   No se hace nada.

3. No dominante → No dominante.
   Modificar en el vector de elementos el valor que se encuentra en esa posicion, de esta manera cuando el usuario quiera volver a acceder a dicha posicion obtendra el nuevo valor que a ingresado.

4. No dominante → dominante.
   Adicionar en los dos vectores el valor que el usuario esta ingresando, de esta manera, cuando el usuario quiera acceder a dicha posicion, el elemento podra ser encontrado en los vectores, y se devolverá el valor del vector de elementos.

Implementacion de una matriz esparcida en java.

```
class Sparce {
    float VD[]; //Vector de datos o elementos, para
este caso son numeros reales.
    int VFC[]; //Vector de posiciones
    int dim;
    int m;
    int n;
    int e; //Elemento dominante

  public Sparce(int nf, int nc, int ele){ //Constructor
de la matriz esparcida
    VFC = new int[10];
    VD = new float[10];
    n = nc;
    m = nf;
    dim = -1;
    e = ele;
  }
```

```
public void setDato(int f, int c, float dato)
 {
   if ((f > 0)&&(c > 0)&&(f <= m)&&(c <= n))
   {
     int fc = (f-1)*n + c;
     int pos = buscar(fc);
     if (pos != -1)
     {
       if (dato == e)
       {
         dim--;
         for (int i = pos; i < dim; i++) {
           VFC[i] = VFC[i+1];
           VD[i] = VD[i+1];
         }
       }else
         VD[pos] = dato;
     }else{
       if (dato != e)
       {
```

```
            Redimencionar();
            dim++;
            VD[dim] = dato;
            VFC[dim] = (f-1)*n + c;
          }
        }
    }else
        System.out.println("Error: la Fila o Columna
esta fuera de rango");
  }

  private int buscar(int fc)
  {
    int i = 0;
    while ((i <= dim) && (VFC[i] != fc ))
      i++;
    if (i > dim)
      return -1;
    else
      return i;
  }

Private void Redimencionar(){
if ((dim % 9 == 0)&&(dim!=0)){
 float vaux[] = new float[dim+10];
System.arraycopy(VD, 0, vaux, 0, dim+1);
 VD = new float[dim+10];       System.arraycopy(vaux,
0, VD, 0, dim+1);
 int vfc2[] = new int[dim+10];
  System.arraycopy(VFC, 0, vfc2, 0, dim+1);
  VFC = new int[dim+10];
System.arraycopy(vfc2, 0, VFC, 0, dim+1);
}

public float getDato(int f, int c)
  {
    float valor = e;
    if ((f > 0)&&(c > 0)&&(f <= m)&&(c <= n))
    {
      int pos = buscar((f-1)*n + c);
      if (pos != -1)
      valor = VD[ pos ];
    }else
      System.out.println("Error: Fila o Columna fuera
de rango");
    return valor;
  }

public String toStringFila(int fila)
  {
    String s = "";
    if ((fila > 0)&&(fila <=m))
    for (int i = 0; i < n; i++) {
      s = s + "  " + getDato(fila, i+1);
    }
    return s;
  }

  public static void main(String[] args) {
    Sparce a = new Sparce(3,4, 2);
    a.setDato(1, 3, 8);
    a.setDato(1, 4, 5);
    a.setDato(2, 1, 5);
    a.setDato(2, 2, 8);
    a.setDato(2, 2, 8);
    a.setDato(2, 3, 4);
    a.setDato(2, 4, 6);
    a.setDato(3, 1, 1);
    a.setDato(3, 2, 3);
    a.setDato(3, 3, 6);
    for (int i = 0; i < 3; i++) {
      System.out.println(a.toStringFila(i+1));
    }
  }
}
```

Observaciones de la implementación en java

- Podemos observar que en el constructor tenemos los vectores inicializados en 10, debido a que un vector siempre se inicializa con un tamaño especifico.
- Para que el tamaño de 10 no sea definitivo para la matriz, al momento de insertar siempre se comprueba si ya hemos llegado a 10 mas, si es que ya se ha llegado a este rango, entonces se amplian los vectores en 10, de esta manera se tiene un tipo de matriz mas o menos genérica, es por eso que hay el codigo arraycopy en el método setDato.
- El arraycopy lo unico que hace es copiar los valores entre dos vectores definiendo la posicion inicial para cada vector, tanto como para el vector de donde se va a copiar, como para el vector donde se esta copiando, y una cantidad que se va a copiar.
- El método toStringFila devuelve el valor de toda una fila de la matriz esparcida en una cadena, acorde como uno quiera, para eso se programa dentro del metodo, para el caso de esta implementación, tenemos que devuelve cada elemento que se encuentra en la fila a la que corresponde.

Practica Matrices Sparse

1.- Realizar la Implementación de una matriz Sparse a nivel de Bits, donde el vector Fila y columna se representen a nivel de Bits, el vector de datos se lo haga a nivel de Floats

2.- Del ejercicio anterior modificar la estructura para que los datos de la Matriz se representen a nivel de bits con las siguientes características

0	0	$+0.2$	0	0
0	0	-2.5	0	0
0	0	$+1.9$	0	0
-8.1	$+2.9$	-3.8	-8.4	$+3.4$
0	0	-9.9	0	0
0	0	0	0	0
0	0	$+0.5$	0	0

Nota.- La parte entera y decimal soportan 1 sola cifra comprendida entre 0…9

4.- Dada una matriz Sparse MxN de dimensiones impares
Implementar esta matriz que almacene elementos fraccionarios es decir $-4/5$ $+\frac{1}{2}$ donde el elemento predominante es el 0;

0	0	$+\frac{7}{5}$	0	0
0	0	$-\frac{1}{2}$	0	0
0	0	$+\frac{5}{8}$	0	0
$-\frac{2}{7}$	$+\frac{3}{8}$	$-1/4$	$-\frac{14}{5}$	$+\frac{5}{2}$
0	0	$-\frac{4}{5}$	0	0
0	0	0	0	0
0	0	$+\frac{1}{2}$	0	0

5.- Dada una matriz Sparse MxN de dimensiones impares
Implementar esta matriz que almacene elementos fraccionarios es decir $-4/5$ $+\frac{1}{2}$ en forma de cruz donde el elemento predominante es el 0;

0	0	$+\frac{7}{5}$	0	0
0	0	$-\frac{1}{2}$	0	0
0	0	$+\frac{5}{8}$	0	0
$-\frac{2}{7}$	$+\frac{3}{8}$	$-1/4$	$-\frac{14}{5}$	$+\frac{5}{2}$
0	0	$-\frac{4}{5}$	0	0
0	0	0	0	0
0	0	$+\frac{1}{2}$	0	0

Implementar el método set y mostrar para esta matriz
Set(1, 1, 3/5) no inserta
Set(3,1, 4/5) NO inserta
Set (1,3 7/5)SI inserta
Set (3,3,0) Si inserta
Set (3,3, 5/8) Si inserta
Set(7,3, 1/2)Si inserta
Set(6,3, 0)Si inserta
Set(6,4, 4/3)NO inserta

6.- Dada una matriz Sparse MxN de dimensiones impares
Implementar esta matriz que caracteres es decir A,B.C.....Z
donde el elemento predominante es el espacio

H	O	L	A	
E	S	T	O	
E	S	U	N	A
P	R	U	E	B
A		F	E	L
I	C	I	D	A
D	E	S		

Capitulo 6

TDA Listas

6.1. Definición

Es una colección de elementos homogéneos (del mismo tipo: TipoElemento) con una relación LINEAL establecida entre ellos. Pueden estar ordenadas o no con respecto a algún valor de los elementos y se puede acceder a cualquier elemento de la lista.

Las listas son similares a los arreglos, en cuanto se componen de una serie ordenada de objetos, así se puede hacer referencia al primer elemento de la lista, al segundo y así sucesivamente hasta llegar al último elemento de la lista. Al primer elemento de la lista se le llama comúnmente cabeza y al último elemento cola. Sin embargo, las listas difieren de los arreglos en los siguientes aspectos:

Las lista son de longitud variable, es decir, aumentan y se reducen durante la ejecución del programa, y se usan para representar estructuras de datos arbitrarias.
Sus componentes pueden ser heterogéneos, es decir, el tipo de dato de cada miembro de una lista puede diferir de su "vecino".

Los lenguajes que manejan listas, declaran estos datos de manera implícita, es decir sin atributos explícitos para los miembros de la lista.

primer elemento

$L=<a_1, a_2,...,a_n>$

ultimo elemento

Donde.

$a_i \in T$, i=1,...,n (**n** es la longitud de la lista**)**

n=0 $\Rightarrow$ lista vacía

Una manera de clasificarlas es por la forma de acceder al siguiente elemento:

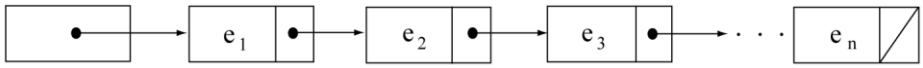
- Lista densa: la propia estructura determina cuál es el siguiente elemento de la lista. Ejemplo: un Array.
- Lista enlazada: La posición del siguiente elemento de la estructura la determina el elemento actual.
  Es necesario almacenar al menos la posición de memoria del primer elemento. Además es dinámica, es decir, su tamaño cambia durante la ejecución del programa.

Una lista enlazada se puede definir recursivamente de la siguiente manera:

- Una lista enlazada es una estructura vacía o
- Un elemento de información y un enlace hacia una lista (un nodo).

6.2. REPRESENTACIÓN GRAFICA

El siguiente Grafico se representa una lista Enlazada

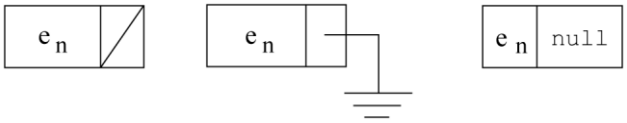


$e_1, e_2, ...e_n$, en son valores del tipo `TipoElemento`

Fig 10 Representación de una Lista

6.3. Representación del Ultimo Elemento

Como vimos en las figuras anteriores el ultimo nodo puede ser representado como el iesimo elemento o con el puntero a null que determina que ese nodo no apunta a nadie mas por lo que es el ultimo elemento de la lista



Diferentes representaciones gráficas del nodo último

Como se ha dicho anteriormente, pueden cambiar de tamaño, pero su ventaja fundamental es que son flexibles a la hora de reorganizar sus elementos; a cambio se ha de pagar una mayor lentitud a la hora de acceder a cualquier elemento.

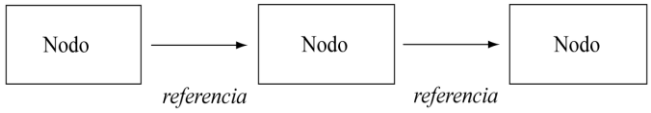
En la lista de la fig. 10. se puede observar que hay dos elementos de información, e1 y e2. Supongamos que queremos añadir un nuevo nodo entre esos 2 elementos con la información x, casi al comienzo de la lista. Para hacerlo basta con crear ese nodo, introducir la información x, y hacer un enlace hacia el siguiente nodo, que en este caso contiene la información x.

¿Qué ocurre si quisiéramos hacer lo mismo sobre un Array?. En ese caso sería necesario desplazar todos los elementos de información "hacia la derecha", para poder introducir el nuevo elemento, una operación muy engorrosa.

6.4. Análisis de Procesos Vector Vs Lista Enlazada

Operación	Representación Contigua	Representación Enlazada
Fin	*O(1)*	*O(1)*
Primero	*O(1)*	*O(1)*
Sgte	*O(1)*	*O(1)*
Ante	*O(1)*	***O(n)***
Vacia	*O(1)*	*O(1)*
Recupera	*O(1)*	*O(1)*
Longitud	*O(1)*	*O(1)*
Inserta	***O(n)***	*O(1)*
Suprime	***O(n)***	*O(1)*
Modifica	*O(1)*	*O(1)*

6.5. Clasificación de Listas Enlazadas



Las listas se pueden dividir en cuatro categorías :

- Listas simplemente enlazadas. Cada nodo (elemento) contiene un único enlace que lo conecta al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos ("adelante").
- Listas doblemente enlazadas. Cada nodo contiene dos enlaces, uno a su nodo predecesor y otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo ("adelante") como en recorrido inverso ("atrás").
- Lista circular simplemente enlazada. Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular ("en anillo").
- Lista circular doblemente enlazada. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular ("en anillo") tanto en dirección directa ("adelante") como inversa ("atrás").

6.5.1. Listas Simplemente enlazadas

Operaciones Básicas

TipoLista Crear()
void Imprimir(TipoLista lista)

Operaciones de Modificacion

void Insertar(TipoLista lista, TipoElemento elem)
void Eliminar(TipoLista lista, TipoElemento elem)
void Modificar(TipoLista lista, Int Posicion)

Operaciones de consulta

int ListaVacía(TipoLista lista)
int Longitud(TipoLista lista)
int Recupera(TipoLista lista, Int posicion)
Opcionalmente, si la implementación lo requiere, puede definirse:
int ListaLlena(TipoLista lista)

- Hay que tener en cuenta que la posición de la inserción no se especifica, por lo que dependerá de la implementación. No obstante, cuando la posición de inserción es la misma que la de eliminación, tenemos una subclase de lista denominada **pila**, y cuando insertamos siempre por un extremo de la lista y eliminamos por el otro tenemos otra subclase de lista denominada **cola**.

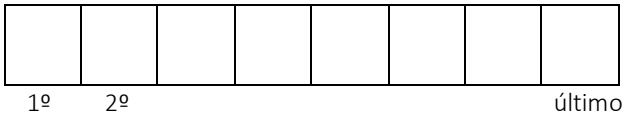
- En el procedimiento Eliminar el argumento elem podría ser una clave, un campo de un registro TipoElemento.

Implementación

La implementación o representación física puede variar:

- Representación **secuencial**: el orden físico coincide con el orden lógico de la lista.

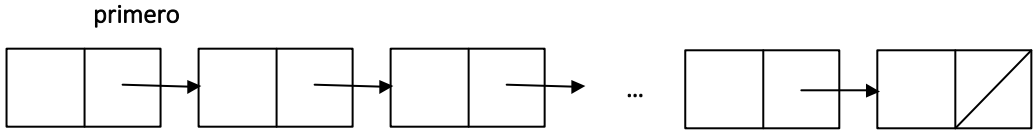
Ejemplo: arrays. Para insertar o eliminar elementos podría exigirse que se desplazaran los elementos de modo que no existieran huecos. Por otra parte, tiene la desventaja de tener que dimensionar la estructura global de antemano, problema propio de las estructuras estáticas.



- Representación **enlazada**: da lugar a la lista enlazada ya estudiada, en la que el orden físico no es necesariamente equivalente al orden lógico, el cual viene determinado por un campo de enlace explícito en cada elemento de la lista. **Ventaja:** se evitan movimientos de datos al insertar y eliminar elementos de la lista.

Podemos implementar una lista enlazada de elementos mediante variables dinámicas o estáticas (el campo enlace será un puntero o un índice de array):

1. **Variables de tipo puntero:** Esta representación permite dimensionar en tiempo de ejecución (cuando se puede conocer las necesidades de memoria). La implementación de las operaciones son similares a las ya vista sobre la lista enlazada de punteros.



2. **Variables estáticas**: se utilizan arrays de registros con dos campos: elemento de la lista y campo enlace (de tipo índice del array) que determina el orden lógico de los elementos, indicando la posición del siguiente elemento de la lista (simulan un puntero). Esta representación es útil cuando el lenguaje de programación no dispone de punteros o cuando tenemos una estimación buena del tamaño total del array.

Ejemplo: Una lista formada por A,B,C,D,E (en este orden) puede representarse mediante el siguiente array de registros:

	0	1	2	3	4	5	6	7	8	9
datos	A		D		B		E	C		
enlace	4		6		7		-1	4		

Con esta implementación debemos escribir nuestros propios algoritmos de manejo de memoria (Asignar y free). Nuestra memoria para almacenar la lista de elementos será el array de registros. De él tomaremos memoria para almacenar un nuevo elemento, y en él liberaremos el espacio cuando un elemento se elimina de la lista.

Por tanto, en realidad coexisten dos listas enlazadas dentro del array de registros, una de memoria ocupada y otra de memoria libre, ambas terminadas con el enlace nulo -1.

datos	A		D		B		E	C		
enlace	4	5	6	8	7	3	-1	4	9	-1
	0	1	2	3	4	5	6	7	8	9

De esta forma, tendremos dos variables: lista, que contiene el índice del array donde comienza la lista enlazada de elementos (lista = 0) y libre, que contendrá el índice del array donde comienza la lista de posiciones de memoria libres (libre = 1).

Cuando queramos **añadir** un nuevo elemento a la lista, se tomará un hueco de la lista libre, disminuyendo ésta y aumentando la primera. Si queremos **eliminar** un elemento, realizaremos la operación contraria.

Cuando no existe ningún elemento en la lista, sólo existirá la lista libre, que enlaza todas las posiciones del array, mientras que si la lista se llena, no tendremos lista libre.

Las siguientes declaraciones nos servirían para construir esta representación, la cual se basa en un array y simula una lista enlazada:

Implementación Basado en Arrays

```
import java.util.ArrayList;
class Nodo {
  int valor;
  int siguiente;

  public Nodo(int valor, int siguiente) {
    this.valor = valor;
    this.siguiente = siguiente;
  }
}

public class ListaEnlazadaBasadaEnVectores {
  private ArrayList<Nodo> nodos;
  private int primerElemento;
  private int ultimoElemento;

  public ListaEnlazadaBasadaEnVectores() {
    nodos = new ArrayList<>();
    nodos.add(new Nodo(0, -1)); // Nodo inicial
    primerElemento = -1;
    ultimoElemento = -1;
  }

  public void agregarElemento(int valor) {
    int nuevoIndice = nodos.size();
    nodos.add(new Nodo(valor, -1));
    if (ultimoElemento == -1) {
      primerElemento = nuevoIndice;
      ultimoElemento = nuevoIndice;
    } else {
      nodos.get(ultimoElemento).siguiente =
nuevoIndice;
      ultimoElemento = nuevoIndice;
    }
  }

  public void imprimirLista() {
    int actual = primerElemento;
    while (actual != -1) {
      System.out.print(nodos.get(actual).valor + " ->
");
      actual = nodos.get(actual).siguiente;
    }
    System.out.println("null");
  }

  public static void main(String[] args) {
    ListaEnlazadaBasadaEnVectores lista = new
ListaEnlazadaBasadaEnVectores();
    lista.agregarElemento(10);
    lista.agregarElemento(20);
    lista.agregarElemento(30);
    lista.imprimirLista();
  }
}
```

Implementación basada en punteros

TDA NODO

```
package Negocio;
public class Nodo {   int Dato;
                Nodo Enlace;

public Nodo() {
      this.Enlace=null;
   }

public Nodo(int Dato, Nodo Enlace) {
      this.Dato = Dato;
      this.Enlace = Enlace;
   }

public void setDato(int Dato) {
      this.Dato = Dato;
   }

public void setEnlace(Nodo Enlace) {
      this.Enlace = Enlace;
   }

public int getDato() {
      return Dato;
   }

public Nodo getEnlace() {
      return Enlace;
   }

   @Override
public String toString() {
      return (Dato+ "->");
   }
}
```

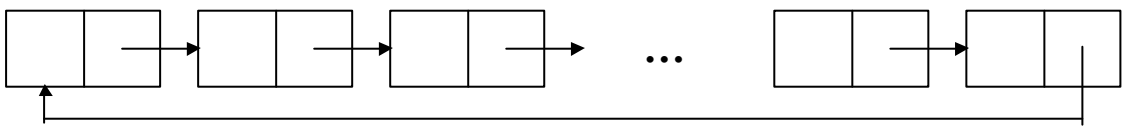
Tda Lista Enlazada

```
package Negocio;
public class Lista { Nodo L;
                     int cant;

 public Lista() {
      this.L = null;
      this.cant =0;
   }
  public boolean vacia(){
  return(L==null);
  }

 public void insertar(int ele){
     if (vacia()){
      Nodo p=new Nodo();
      p.setDato(ele);
      L=p;
      cant++;
     }
     else{ // hay mas elementos
       Nodo Aux=L;
       Nodo Ant=null;
       while ((Aux!=null)&&(Aux.getDato()<=ele)){
          Ant=Aux;
          Aux=Aux.getEnlace();
       }
       if (Ant==null)// Insertamos en la Cabeza
       {
         Nodo p=new Nodo();
         p.setDato(ele);
         p.setEnlace(L);
         L=p;
         cant++;

       }else {  if (Ant.getDato()!=ele){
             Nodo p=new Nodo();
             p.setDato(ele);
             Ant.setEnlace(p);
             p.setEnlace(Aux);
             cant++;
           }
       }
     }
  }

public void Eliminar(int ele){
  if (!vacia()){
   Nodo Aux=L;
   Nodo Ant=null;
      while ((Aux!=null)&&(Aux.getDato()!=ele)){
         Ant=Aux;
         Aux=Aux.getEnlace();
      }
    if (Aux!=null)
       if (Ant==null){
         L=L.getEnlace();
         cant--;

       }
       else{ Ant.setEnlace(Aux.getEnlace());
           cant--;
       }
    }
  }
```

```
public boolean Existe(int ele){
  boolean x=false;
  if (!vacia()){
   Nodo Aux=L;
   while ((Aux!=null)&&(Aux.getDato()!=ele)){
    Aux=Aux.getEnlace();
    }
   if (Aux!=null){
    x=true;
   }
  }
  return x;
  }

public int getDato (int pos){//pre la posicion debe
estar en rango
  Nodo Aux=L;
     for (int i = 1; i < pos; i++) {
        Aux=Aux.getEnlace();
     }
   return(Aux.getDato());
  }

  @Override
public String toString() {
      String S="L->  < ";
      Nodo Aux=L;
      while  (Aux!=null){
        S=S+Aux.getDato()+" , ";
        Aux=Aux.getEnlace();
       }
      S=S+" > ";
      return S;
   }

public void Invertir(){
  InvertirR(L);
}
private void InvertirR(Nodo p){
 if (p==null){
  //nada
  }
 else {
      if (p.getEnlace()==null){
      //nada
         L=p;
       }
      else{// caso general
          InvertirR(p.getEnlace());
          p.getEnlace().setEnlace(p);
          p.setEnlace(null);
      }
    }
}

public static void main(String[] args) {
      // TODO code application logic here
      Lista LI=new Lista();
      LI.insertar(10);
      LI.insertar(2);
      LI.insertar(1);
      LI.insertar(10);
      LI.insertar(5);
      LI.insertar(8);
      LI.insertar(3);
      LI.insertar(2);
      System.out.println(LI);
      LI.Invertir();
      System.out.println(LI);
      LI.Eliminar(8);
      LI.Eliminar(1);
      System.out.println(LI);
}
```

6.52. Listas Circulares

- Son listas en las que el último elemento está enlazado con el primero, en lugar de contener el valor NULL o NULO. Evidentemente se implementarán con una representación enlazada (con punteros o variables estáticas). Con punteros sería:



- Una lista vacía lo representaremos por el valor NULL o NULO. Es conveniente guardar en algún sitio el primer o último elemento de la lista circular para detectar el principio o el final de la lista, ya que NULL o NULO ahora sólo significa que la lista está vacía. Así, las comparaciones para detectar final de lista se harán con el propio puntero de la misma.

A continuación realizaremos la implementación de una Lista Circular que almacena Objetos para alma cenar de forma genérica cualquier elemento pero en esta ampliación lo haremos con enteros

TDA LISTA CIRCULAR SIMPLE GENÉRICA

Tda Nodo Simple

```
public class CNodoS {
 private Object Elem; // el dato Pude ser cualquier
tipo
 private CNodoS RS;
 public CNodoS()
 {
 Elem=null;
 RS=null;
 }
 void setElem(Object ele){
  Elem = ele;
 }
 Object getElem(){
  return Elem;
 }

void setRS(CNodoS Raux ){
  RS=Raux;
 }
  CNodoS getRS(){
  return RS;
 }
}
```

Tda Lista Circular

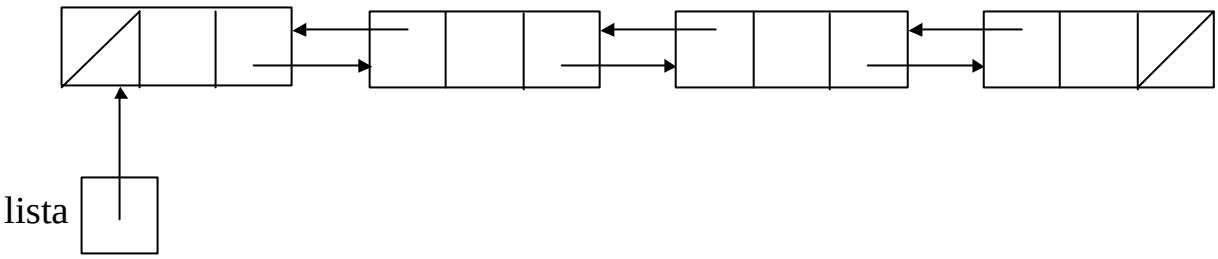
```
public class CLCircularS {
  CNodoS Primero;
  int N;
  public CLCircularS() {
  Primero = new CNodoS();
  Primero=null;
  N=0;
  }

public void insertar(Object e){
  CNodoS nuevo= new CNodoS();
  nuevo.setElem(e);
  if (Primero == null){
    Primero=nuevo;
    Primero.setRS(Primero);
  }else{
    CNodoS aux= new CNodoS();
    aux=Primero;
    while(aux.getRS()!=Primero){
    aux=aux.getRS();
    }
    aux.setRS(nuevo);
    nuevo.setRS(Primero);
   }
  N++;
  }

 public void insertarInicio(Object e){
  CNodoS nuevo = new CNodoS();
  nuevo.setElem(e);
  if (Primero == null){
    Primero = nuevo;
    Primero.setRS(Primero);
  }else{
  CNodoS aux = new CNodoS();
  aux= Primero;
  while (aux.getRS()!=Primero)
  aux=aux.getRS();
  aux=nuevo;
  nuevo.setRS(Primero);
  Primero=nuevo;
  }
  N++;
  }

 public void insertar(Object e, int pos ){
  CNodoS nuevo= new CNodoS();
  nuevo.setElem(e);
  if ((Primero == null)||(pos<=0))
  insertarInicio(e);
  else{
  if (pos>=N-1)
  insertar(e);
  else{
  CNodoS aux = new CNodoS();
  aux=Primero;
  pos--;
  while(pos>0){
  aux=aux.getRS();
  pos--;
  }
  nuevo.setRS(aux.getRS());
  aux.setRS(nuevo);
  }
  }
  N++;
  }
```

```
 public Object consultarInicio(){
   return Primero.getElem();
  }

  public Object consultar(){
  CNodoS aux = new CNodoS();
  aux=Primero;
  while (aux.getRS()!= Primero){
  aux= aux.getRS();
  }
    return aux.getElem();
  }

  public Object consultar(int pos){
  CNodoS aux = new CNodoS();
  aux = Primero;
  if(pos>=N-1)
  return consultar();
  else{
  if (pos<=0)
  return  consultarInicio();
  else{
  while (pos>0){
    aux=aux.getRS();
     pos--;   }
    }  }
  return aux.getElem();
  }

public void eliminarInicio(){
  if(N==1)
  Primero= null;
  else{
  CNodoS aux = new CNodoS();
  aux=Primero;
  while (aux.getRS()!= Primero)
  aux=aux.getRS();
  aux.setRS(Primero.getRS());
  Primero=aux.getRS();   }
  N--;   }

  public void eliminar(){
  if(N==1)
  Primero= null;
  else{
  CNodoS aux = new CNodoS();
  aux=Primero;
  int i=N-2;
  while (i>0){
  aux=aux.getRS();
  i--;
  }
  aux.setRS(aux.getRS().getRS());
  }
  N--;    }

  public void eliminar(int pos){
  if((pos<=0)||(N==1))
  eliminarInicio();
  else{
  if(pos>=N-1)
  eliminar();
  else{
  CNodoS aux =  new CNodoS();
  aux=Primero;
  pos--;
  while (pos>0){
  aux=aux.getRS();
  pos--;
  }
  aux.setRS(aux.getRS().getRS());
  }
  }
  N--;
  }

  public int getLongitud(){
  return N;
  }

  @Override
  public String toString() {
   if (Primero == null) {
     System.out.println("La lista está vacía.");
     return "C->null";
    }
   else{
    String S="C->";
    CNodoS p=Primero;
    do {
      S=S+p.getElem()+ " -> ";
      p=p.getRS();
    } while (p != Primero); // Se imprime hasta volver
al inicio

    return S;
    }
  }

  public static void main(String Args[]){
  CLCircularS C= new CLCircularS();
  C.insertarInicio(5);
  C.insertar(7);
  C.insertar(8);
  C.insertar(6,1);
  C.eliminar(2);
  System.out.println(C);

  }
}
```

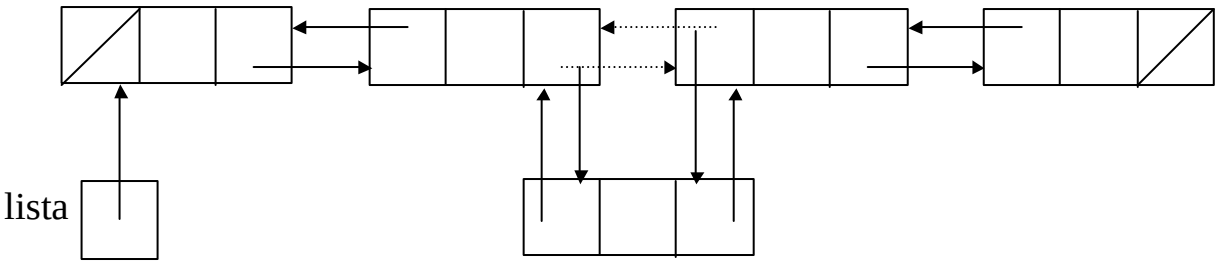
TDA

6.5.3. Listas doblemente enlazadas

- Son listas en las que cada nodo, además de contener los datos información propios del nodo, contiene un enlace al nodo anterior y otro al nodo siguiente:

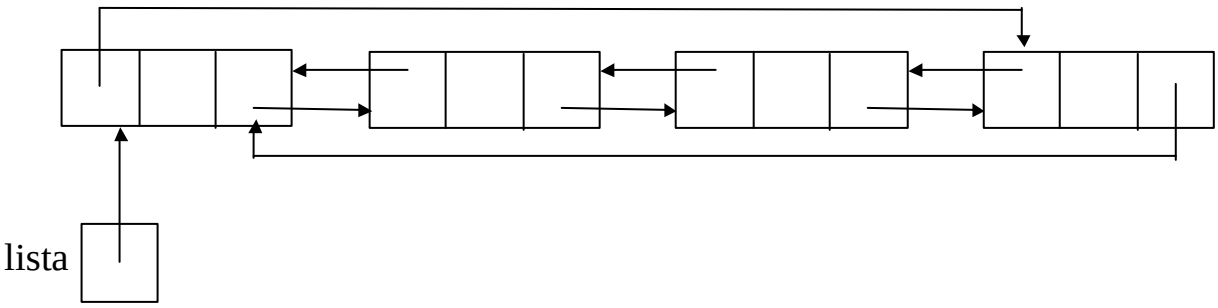


- Aparte de que ocupan más memoria (el tamaño de un puntero por cada nodo más), los algoritmos para implementar operaciones para listas dobles son más complicados que para las listas simples porque requieren manejar más punteros. Para insertar un nodo, por ejemplo, habría que cambiar cuatro punteros (o índices si se simulan con un array):



- La ventaja es que pueden ser recorridas en ambos sentidos, con la consiguiente eficiencia para determinados procesamientos.

- Una lista doble también puede ser circular:



Implementación del TDA Lista Doble con manejo de Enteros

TDA Nodo Doble

```
public class NodoD {
  int Data;
  NodoD EnlaceD;
  NodoD EnlaceI;

public NodoD(int Data) {
    this.Data = Data;
    this.EnlaceD = null;
    this.EnlaceI = null;
  }

public NodoD() {
    this.EnlaceD=null;
    this.EnlaceI=null;
  }
public void setData(int Data) {
    this.Data = Data;
  }
public int getData() {
    return (this.Data);
  }
public void setEnlaceD(NodoD EnlaceD) {
    this.EnlaceD = EnlaceD;
  }
public void setEnlaceI(NodoD EnlaceI) {
    this.EnlaceI = EnlaceI;
  }
public NodoD getEnlaceD() {
    return EnlaceD;
  }
public NodoD getEnlaceI() {
    return EnlaceI;
  }
@Override
public String toString() {
    return "NodoD{" + "Data=" + Data +
        ", EnlaceD=" + EnlaceD +
        ", EnlaceI=" + EnlaceI + '}';
  }
}
```

TDA Lista Doble

```
public class Listas_Dobles {
 NodoD Primero;
          NodoD Ultimo;
         int Cantidad;

  public Listas_Dobles() {
    this.Primero = null;
    this.Ultimo = null;
    this.Cantidad = 0;
  }

  private boolean vacia() {
    return (Cantidad == 0);
  }

  public void insertar(int ele) {
    NodoD N = new NodoD(ele);
    if (vacia()) {
      this.Primero = N;
      this.Ultimo = N;
      this.Cantidad++;
    } else {
      NodoD aux = this.Primero;
      while ((aux != null) && (aux.getData() < ele)) {
        aux = aux.getEnlaceD();
      }
      if (aux == this.Primero) {//Insertar en el inicio
        if (ele != aux.getData()) {
          N.setEnlaceD(this.Primero);
          this.Primero.setEnlaceI(N);
          this.Primero = N;
          this.Cantidad++;
        }
      } else {
        if (aux != null) {//Insertar en el cuerpo
          if (aux.getData() != ele) {
            N.setEnlaceD(aux);
            N.setEnlaceI(aux.getEnlaceI());
            aux.getEnlaceI().setEnlaceD(N);
            aux.setEnlaceI(N);
            this.Cantidad++;
          }
        } else {
          this.Ultimo.setEnlaceD(N);
          N.setEnlaceI(this.Ultimo);
          this.Ultimo = N;
          this.Cantidad++;
        }
      }
    }
  }

  public void eliminar(int ele) {
    if (vacia()) {
      System.out.println("Error :: Eliminar : La lista
esta vacia");
      System.exit(1);
    } else {
```

```
        NodoD aux = this.Primero;
        while ((aux != null) && (aux.getData() != ele)) {
          aux = aux.getEnlaceD();
        }
        if (aux != null) {
          if (this.Primero == this.Ultimo) {
            this.Primero = null;
            this.Ultimo = null;
          } else {
            if (aux == this.Primero) {
              this.Primero = this.Primero.getEnlaceD();
              this.Primero.setEnlaceI(null);
              this.Cantidad--;
            } else {
              if (aux == this.Ultimo) {
                this.Ultimo = this.Ultimo.getEnlaceI();
                this.Ultimo.setEnlaceD(null);
                this.Cantidad--;
              } else {//Aqui Modificamos el cuerpo

aux.getEnlaceD().setEnlaceI(aux.getEnlaceI());

aux.getEnlaceI().setEnlaceD(aux.getEnlaceD());
                aux.setEnlaceD(null);
                aux.setEnlaceI(null);
                this.Cantidad--;
              }
            }
          }
        }
      }
    }

    public int buscar(int pos) {
      NodoD N = this.Primero;
      if ((pos <= this.Cantidad) && (pos > 0)) {
        for (int i = 1; i < pos; i++) {
          N = N.getEnlaceD();
        }
        return N.getData();
      } else {
        return 0;
      }
    }


    /*function que devuelva
    el numero en dicha posicion*/
    public int buscarPos(int ele) {
      int p = 1;
      NodoD N = this.Primero;
      if (!vacia()) {
        while ((ele != N.getData()) && (N !=
this.Ultimo)) {
          N = N.getEnlaceD();
          p++;
        }
        if((N==Ultimo)&&(ele != N.getData())){
          p=0;
        }
      }
      return p;
    }

    @Override
    public String toString() {
      String cad = "P <=> ";
      NodoD N = this.Primero;
      for (int i = 1; i <= this.Cantidad; i++) {
        cad = cad + "[" + N.getData() + "] <=> ";
        N = N.getEnlaceD();
      }
      return cad + " U";
    }
  }
```

6.5.4. Listas Genéricas

Las listas genéricas en programación ofrecen flexibilidad y reutilización del código al permitirte crear estructuras de datos que pueden trabajar con cualquier tipo de datos. Estas listas se definen utilizando un tipo de dato genérico que se determina en tiempo de compilación. Aquí te describo algunas de las utilidades y ventajas de las listas genéricas:

a. **Flexibilidad de datos:** Las listas genéricas permiten trabajar con cualquier tipo de datos, incluidos tipos primitivos y objetos. Esto es especialmente útil cuando necesitas almacenar elementos de diferentes tipos en una misma lista.

b. **Reutilización de código:** Al usar listas genéricas, puedes escribir una implementación que funcione para cualquier tipo de datos. No es necesario volver a escribir el código de la lista para diferentes tipos de elementos.

c. **Seguridad de tipos en tiempo de compilación:** La verificación de tipos en tiempo de compilación asegura que solo se puedan agregar y obtener elementos del tipo especificado, evitando errores de tipo en tiempo de ejecución.

d. **Mejora de la legibilidad del código:** Al usar listas genéricas, el código es más claro y legible, ya que no necesitas realizar conversiones de tipos explícitas, lo que facilita la comprensión y el mantenimiento del código.

e. **Encapsulación y abstracción**: Las listas genéricas permiten encapsular la lógica de almacenamiento y manipulación de datos, lo que facilita la abstracción del funcionamiento interno de la lista.

f. **Facilita la interoperabilidad y adaptabilidad:** Al ser genérica, la lista puede adaptarse a diferentes contextos y necesidades de una aplicación, mejorando la interoperabilidad con otros componentes y módulos.

g. **Eficiencia y rendimiento:** Al trabajar con tipos de datos específicos, las listas genéricas pueden ser más eficientes en términos de rendimiento al evitar conversiones y optimizar el almacenamiento y manipulación de datos.

En resumen, las listas genéricas son fundamentales en la programación moderna ya que brindan flexibilidad, reutilización de código, seguridad de tipos y mejoran la legibilidad y eficiencia en el desarrollo de aplicaciones. Su uso es esencial en muchos lenguajes de programación, incluido Java, C#, Python y muchos otros.

Implementación del TDA Lista Genérica

Esta implementación que permitirá manipular y procesar cualquier tipo de estructuras de datos, Objetos simples o definidos por el Usuario la cual esta basado en parámetros de tipo de variable que permitirá enviar el tipo de variable al momento de hacer uso de la Lista.

TDA Nodo Genérico

```
import java.io.Serializable;

 public class NodoGP <Tipovar> implements
Serializable
   {  Tipovar Dato;
      NodoGP<Tipovar> Enlace;

   public NodoGP(){
    Enlace=null;
   }

   public NodoGP (Tipovar Dato){
     this.Dato=Dato;
    Enlace=null;
   }

   public void SetDato(Tipovar Dato){
     this.Dato=Dato;
   }

    public Tipovar GetDato(){
      return Dato;
    }

    public void SetEnlace(NodoGP<Tipovar> P){
     Enlace=P;
    }
    public  NodoGP<Tipovar> GetEnlace(){
      return Enlace;
    }

    @Override
    public String toString() {
      return "NodoGP{" + "Dato=" + Dato + ", Enlace="
 + Enlace + '}';
    }
    }
```

TDA Lista Genérico

```
import java.io.Serializable;
public class ListaG <Tipovar extends Comparable>
implements Serializable{    NodoGP<Tipovar> L;
                                int Cant;
public  ListaG(){
     L=null;
     Cant=0;   }

public int cantidad(){
  return Cant;
  }

 public void Insertar(Tipovar Dato){
     if (L==null) { //esta vacia
        NodoGP P=new NodoGP<Tipovar>();
        P.SetDato(Dato);
        L=P;
        Cant++;
     }else {
        NodoGP<Tipovar> Ant = null;
        NodoGP<Tipovar> Aux = L;
          while
((Aux!=null)&&(Aux.GetDato().compareTo(Dato)<=0
)) {
             Ant=Aux;
             Aux=Aux.GetEnlace();
           }
           NodoGP P= new NodoGP<Tipovar>(Dato);
          if (Ant==null) {//inserta en el primero
           P.SetEnlace(L);
           L=P;
           Cant++;
          }else {
              if (Ant.GetDato().compareTo(Dato)!=0) {
               Ant.SetEnlace(P);
               P.SetEnlace(Aux);
               Cant++;
              }
          }
     }
  }

public String Mostrar(){
     String S="Vacia !!!";
     if (L!=null) {
        S="";
        NodoGP P=L;
        while (P!=null) {
          if (P.GetEnlace()==null) S=S+P.GetDato();
else S=S+P.GetDato()+"->";
          P=P.GetEnlace();
        }
     }
     return S;
  }

public Boolean Buscar(Tipovar Dato){
     if (L==null) {
       return (false);
     } else {
        NodoGP<Tipovar> Aux = L;
          while
((Aux!=null)&&(Aux.GetDato().compareTo(Dato)!=0))
            Aux=Aux.GetEnlace();
            if (Aux!=null)
             return true;
            else return false;
     }
  }

public Tipovar Getdato(int pos){
    NodoGP<Tipovar> Aux=L;
    for (int i = 1; i <pos; i++) {
      Aux=Aux.GetEnlace();
    }
 return(Aux.GetDato());
 }

public Tipovar BuscarObj(Tipovar Dato){
     if (L==null) {
       return Dato;
     } else {
        NodoGP<Tipovar> Aux = L;
          while
((Aux!=null)&&(Aux.GetDato().compareTo(Dato)!=0)
)
            Aux=Aux.GetEnlace();
          if (Aux!=null)
            Dato=Aux.GetDato();
          return Dato;
     }    }

  public boolean Modificar(Tipovar Dato){
     if (L==null) {
       return false;
     } else {
        NodoGP<Tipovar> Aux = L;
          while
((Aux!=null)&&(Aux.GetDato().compareTo(Dato)!=0))
            Aux=Aux.GetEnlace();
          if (Aux!=null) {
            Aux.Dato=Dato;
            return true;
          }
          return false;
     }
  }
```

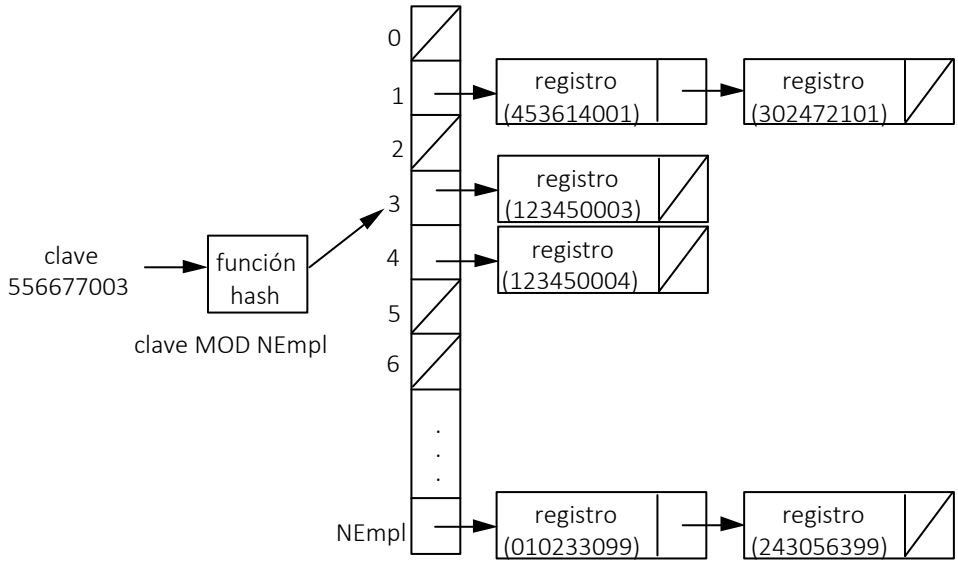
```
public void Eliminar(Tipovar ele){
 if (L!=null){
  NodoGP<Tipovar> Aux=L;
  NodoGP<Tipovar> Ant=null;
     while
((Aux!=null)&&(Aux.GetDato().compareTo(ele)!=0)){
       Ant=Aux;
       Aux=Aux.GetEnlace();  }
   if (Aux!=null)
      if (Ant==null){
       L=L.GetEnlace();
       Cant--;    }
      else{ Ant.SetEnlace(Aux.GetEnlace());
          Cant--;
      } } }

  @Override
  public String toString() {
    String S=" ";
    NodoGP<Tipovar> Aux=L;
    while (Aux!=null)
    { S=S+Aux.toString()+'\n';
     Aux=Aux.GetEnlace();   }
    return S;
  } }
```

```
public static void main(String[] args) {
    // TODO code application logic here
    ListaG LE=new ListaG<Integer>();
    LE.Insertar(1);
    LE.Insertar(20);
    LE.Insertar(5);
    LE.Insertar(3);
    System.out.println(LE.Mostrar());
    ListaG LF=new ListaG<Float>();
    LF.Insertar(10.2);
    LF.Insertar(1.1);
    LF.Insertar(5.1);
    LF.Insertar(8.41);
    System.out.println(LF.Mostrar());
    ListaG LS=new ListaG<String>();
    LS.Insertar("zalberto");
    LS.Insertar("carlos");
    LS.Insertar("raul");
    LS.Insertar("manuel");
    LS.Insertar("abigail");
    System.out.println(LS.Mostrar());
    System.out.println('\n');
  }
```

6.5.5. Otras Aplicaciones de listas

Las listas se utilizan en casi todo tipo de software, especialmente su representación enlazada con punteros. Un ejemplo de aplicación del TAD lista son las tablas hash con encadenamiento de sinónimos. La tabla sería un array de listas de longitud variable: un nodo por cada colisión, lo que evitaría sobredimensionar la tabla y además su posible desborde:

```
/* TIPOS */  typedef ¿? TipoElemento /* un registro p. ej. /
 TipoLista TablaHash[NmEmpl];
```



- Un elemento de la tabla, TablaHash[índice], sería un TAD lista, por lo que podríamos utilizar sus operaciones para almacenarlo:

  Insertar(TablaHash[fhash(clave)], elemento);

- para buscarlo:

  Buscar(TablaHash[fhash(clave)],clave,elemento,es tá)

- o para eliminarlo:

Eliminar(TablaHash[fhash(clave)], elemento)

PRACTICA DE LISTAS

1.- Añadir a la clase ListaDoble un método que devuelva el número de nodos de una lista doble.

2.- En una lista enlazada de números enteros, se desea añadir un nodo entre dos nodos consecutivos cuyos datos tienen distinto signo; el dato del nuevo nodo debe ser la diferencia, en valor absoluto, de los dos nodos.

3- Añadir a la clase Lista el método eliminaraPosicion()para que elimine el nodo que ocupa la posición i, siendo el nodo cabecera el que ocupa la posición 0.

4.- Se tiene una lista simplemente enlazada de números reales. Escribir un método para obtener una lista doble ordenada respecto al campo dato con los valores reales de la lista simple.

5.- Escribir un método para crear una lista doblemente enlazada de palabras introducidas por teclado. La referencia de acceso a la lista debe ser el nodo que está en la posición intermedia.

6.-La clase Lista Circular Cadena dispone de los métodos que implementan las operaciones de una lista circular de palabras. Escribir un método que cuente el número de veces que una palabra dada se encuentra en la lista.

7.-Dada una lista circular de números enteros, escribir un método que devuelva el mayor entero de la lista.

8.- Se tiene una lista enlazada donde el campo dato es objeto alumno con las variables: nombre, edad, sexo. Escribir un método para transformar la lista de tal forma que si el primer nodo es un alumno de sexo masculino, el siguiente sea de sexo femenino, y así alternativamente, siempre que sea posible.

9.-Una lista circular de cadenas está ordenada alfabéticamente. El puntero de acceso a la lista tiene la dirección del nodo alfabéticamente mayor. Escribir un método para añadir una nueva palabra, en el orden que le corresponda, a la lista.

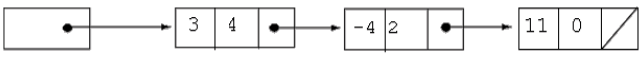
10.-Dada la lista del Ejercicio 9 escribir un método que elimine una palabra dada.

11.- Crear una lista enlazada de números enteros positivos al azar, donde la inserción se realiza por el último nodo, además implementar un método para recorrer la lista para mostrar los elementos por pantalla y también implementar el método para Eliminar todos los nodos que superen un valor dado.

12.- Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista, se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa, escribir las palabras de la lista en el archivo.

13.- Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y el exponente.

Por ejemplo, el polinomio $3x^4 – 4x^2 + 11$ se representa así:



14.- Escribir un programa que permita dar entrada a polinomios en x, representándolos con una lista enlazada simple. A continuación, obtener una tabla de valores del polinomio para valores de x = 0.0, 0.5, 1.0, 1.5, … , 5.0.

15.- Teniendo en cuenta la representación de un polinomio propuesta en el Problema 8.3, hacer los cambios necesarios para que la lista enlazada sea circular. La referencia de acceso debe tener la dirección del último término del polinomio, el cual apuntará al primer término.

16.- Según la representación de un polinomio propuesta en el Problema 14, escribir un programa que realice las siguientes operaciones:

- Obtener la lista circular suma de dos polinomios.
- Obtener el polinomio derivada.
- Obtener una lista circular que sea el producto de dos polinomios.}

17.- Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter. Una vez que se haya creado la lista, ordenarla alfabéticamente y escribirla en pantalla.

18-. Un conjunto es una secuencia de elementos, todos ellos del mismo tipo sin duplicidades. Escribir un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las siguientes operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.
- Escribir en pantalla los elementos del conjunto.

19.- Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final) y eliminar un pasajero de la lista. A la lista se accede por dos variables, una referencia al primer nodo y la otra al último nodo.

20.- Un vector disperso es aquél que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las siguientes operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.

21.- En base a la lista Simple implementada en la anterior Unidad implementar los siguientes Métodos propios de la class Lista

a. Public void Invertir()// invierte la Lista modificando enlaces no contenido
b. Public void intercambiar2()// intercambia los datos de la lista de 2 en 2 desde la cabeza de la lista, es decir
   Si L<1,2,3,4,5,6,7 > el resultado será L<2,1,4,3,6,5,7 >
c. Public void intercambiar2()// intercambia los datos de la lista de 2 en 2 desde la cola de la lista, es decir
   Si L<1,2,3,4,5,6,7 > el resultado será L<1,3,2,5,4,7,6 >

d. Public void intercambiar3()// intercambia los datos de la lista de 3 en 3, es decir
   Si L<1,2,3,4,5,6 > el resultado será L<3,2,1,6,5,4 >
e. Public void intercambiarmedios()// intercambia los datos de la lista de 3 con la característica que el primer numero de cada 3 lo envia a la ultima posición de los 3, es decir
   Si L<1,2,3,4,5,6 > el resultado será L<2,3,1,5,6,4 >
f. Public void intercambiarextremos ()// intercambia los datos extremos de toda la lista de la lista, es decir
   Si L<1,2,3,4,5,6 > el resultado será L<6,2,3,4,5,1 >
g. Public void EliminarRepetidos(int dato)// Elimina todos los elementos repetidos que se encuentran en la lista según el dato ingresado, es decir
   Si L<1,1,1,2,3,3,3,,4,5,6 > se quiere eliminar el elemento 3 el resultado será L<1,1,1,2,3,4,5,6 >
h. Public void EliminarRepetidos()// Elimina todos los elementos repetidos que se encuentran en la lista, es decir
   Si L<1,1,1,2,3,3,3,,4,5,6 > el resultado será L<1,2,3,4,5,6 >

i. Public void EliminarRepysimismo()// Elimina todos los elementos repetidos y asi mismo que se encuentran en la lista, es decir
   Si L<1,1,1,2,3,3,3,,4,5,6 > el resultado será L<2,4,5,6 >

22.- Implementar una lista para contener los siguientes datos ordenados por Código de libro bajo el siguiente criterio

Lista Libros
CodigoLibro=1002
Titulo=Programación I
Autor=Luis Joyanes Aguilar
Editorial= Mc Graw Hill
Edicion=3°
Año de publicación=Marzo del 2010
Nro Paginas=550
Volumen=1

23.- Implementar una lista para contener los datos personales de 10000 alumnos de la Facultad de Ciencias de la Computación de la UAGRM que contenga la siguiente información

Lista Alumnos

Registro=20202547
Nombre=Raul
Apellido Paterno=Gomez
Apellido Materno= Terrazas
Fecha Nacimiento=15/03/1980
Sexo=Masculino
Lugar de Nacimiento=Santa Cruz de la Sierra
Carrera=Ingeniería Informática
Año ingreso=2020
PPA=88
Activo=Si

24.- En base a la Class Lista Doble implementar la class Domino que simula el juego del Domino donde se tiene una ficha con 2 datos numéricos es decir

Se desea implementar los siguientes métodos 

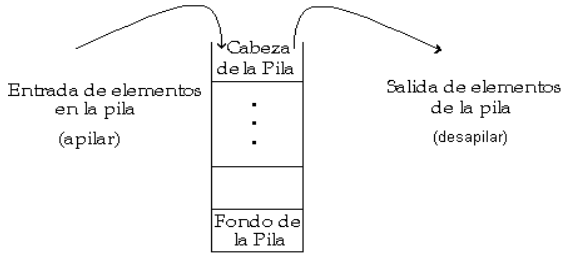
- Crear el Domino donde no hay ni una sola ficha
- Insertar Ficha la cual inserta en la lista de fichas una ficha que tenga unos de los números iguales al primer elemento o al ultimo elemento de la lista de fichas
- Mostar la secuencia de la lista de fichas en formato texto

CAPITULO 7

PILAS

7.1. Definición.-

Una Pila es una ED que almacena elementos de un tipo determinado homogéneos, que sigue la política de que el primer elemento que se extrae (se 'desapila') es el último que se introdujo (se 'apiló') -primero en salir, ultimo en entrar LIFO *(Last In First Out):*, es decir *que sólo se pueden añadir y extraer elementos por el mismo extremo, la cabeza.*



TAD Pila

Todos los procesadores tienen un mecanismo de este tipo conocido por el nombre 'Pila del Sistema', el cual se utiliza para guardar los parámetros de las subrutinas, y otros valores.

7.2. Especificación Del Tad Pila

Operación: Inicializar una pila.

Especificación Sintáctica:

Nombre de la operación: INIPILA.
Parámetros de entrada, y tipos: no tiene
Parámetros de salida: una pila, de tipo TPila.
Resultado: ninguno.
Declaración: **Procedure** IniPila (**var** Pila: TPila);

Especificación Semántica:

Crea una pila, que inicialmente está vacía.

Operación: Apilar un elemento en la pila

Especificación Sintáctica:

Nombre de la operación: APILAR
Parámetros de entrada, y tipos:
Pila que recibe al elemento, de tipo TPila.
Elemento que se introduce en la pila, de tipo Entero.
Parámetros de salida: la propia pila, de tipo TPila, con el elemento introducido.
Declaración: **Procedure** Apilar (Elemento: **Integer**; **var** Pila: TPila);

Especificación Semántica:

Introduce en la pila el elemento pasado como parámetro. La próxima vez que se ejecute la operación 'DESAPILAR', el elemento extraido será el ahora introducido.

Operación: Desapilar un elemento de la pila

Especificación Sintáctica:

Nombre de la operación: DESAPILAR

Parámetros de entrada, y tipos:

Pila que contiene el elemento, de tipo TPila.

Parámetros de salida: la propia pila, con el elemento introducido.

Pila de la que se ha extraido al elemento, de tipo TPila.

Elemento que se extrae de la pila, de tipo Entero.

Declaración: Procedure Desapilar (var Pila: TPila; var Elemento: Integer);

Especificación Semántica:

Extrae de la pila un elemento. El elemento extraido es justamente el último que se introdujo.

Operación: Determinar si la pila está vacía.

Especificación Sintáctica:

Nombre de la operación: PILAVACIA

Parámetros de entrada, y tipos:

Pila que se quiere comprobar, de tipo TPila.

Parámetros de salida: ninguno.

Resultado: un valor booleano.

Declaración: Function PilaVacia (Pila: TPila): Boolean;

Especificación Semántica:

Devuelve el valor 'cierto' si la pila no contiene ningún elemento.

7.3. Representaciones del TAD Pila

7.3.1. Implementación Con un array

- Podemos poner los elementos de forma secuencial en el array, colocando el primer elemento en la primera posición, el segundo en la segunda posición, y así sucesivamente. La última posición utilizada será la cabeza, que será la única posición a través de la cual se podrá acceder al array (aunque el array permita el acceso a cualquiera de sus elementos). Para almacenar el valor de la cabeza utilizamos una variable del tipo índice del array (normalmente entera):
- Si el tipo base de los elementos de la pila coincide con el tipo índice o es compatible, podemos utilizar una posición del propio array (pila[0] por ejemplo), para guardar el valor de la cabeza:
- ¡Ojo con las precondiciones de las operaciones Meter y Sacar!:
  a) No se puede meter más elementos en una pila llena.
  b) No se puede sacar elementos de un pila vacía.

  Pueden tenerse en cuenta ANTES de cada llamada a Meter y Sacar, o preferiblemente en los propios procedimientos Meter y Sacar.

```
public class Pila_De_Vector {
                        int P[];
                       int cima;

  public Pila_De_Vector(int cant) {
    P = new int[cant];
    cima = -1;
  }

  public boolean vacia() {
    return (cima == -1);   }

  public boolean llena() {
    return (cima == P.length - 1);
  }
  public int get() {
    return (P[cima]);
  }
  public void push(int ele) {
    if (!llena()) {
      cima++;
      P[cima] = ele;
```

```
    } else {//Pienso que es mejor quitar la parte de
else
        System.out.println("Error : Pila Llena");
        System.exit(1);
    }
  }

  public int pop() {
    cima--;
    return (P[cima + 1]);
  }

  public int getCima() {
    return (this.cima);
  }

  public void eliminar() {
    if (this.cima > -1) {
      cima--;
    }
  }
```

```
@Override
  public String toString() {
    String cad = "[ cima ] \n";
    if (!vacia()) {
      for (int i = cima; i >= 0; i--) {
        cad = cad + "[ " + P[i] + " ] \n";
      }
    }
    return cad;
  }

public static void main(String[] args) {
    Pila_De_Vector v = new Pila_De_Vector(5);
    v.push(4);
    v.push(5);
    v.push(2);
    v.push(4);
    v.push(2);
    System.out.println("Primera Pila" + v.toString());
}
```

7.3.2. Representación a nivel de bits

Así también para poder optimizar esta representación, es posible poder implementar esta estructura Pila a nivel de bits donde el array estará subdividido en subvectores para almacenar los datos requeridos según sea necesario en base a la cantidad de bits que se requieran para cada dato.

Implementación a Nivel de Bits

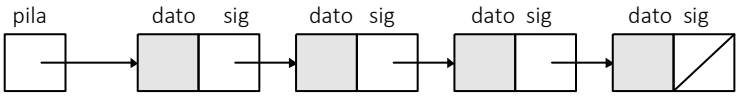
```
public class PilaB { VectorbitsGe P;
                    int cima;
 public PilaB(int cant, int nbits) {
     P=new VectorbitsGe(cant,nbits);
     cima=0;
  }
  public boolean Vacia(){
  return cima==0;
  }
  public boolean llena(){
  return P.dim==cima;
  }
   public void Push(int ele){

    if (llena()){
    System.out.println("Error::Push: La pila Bits esta
Llena");
    System.exit(1);
  }
  else{
    cima++;
    P.Insertar(ele, cima);
  } }
```

```
  public int Pop(){
   cima--;
   return(P.Get(cima+1));

  }

   public int Get(){
   return P.Get(cima);
   }

  @Override
  public String toString() {
    String S="";
    for (int i = cima; i >=1; i--) {
      S=S+P.Get(i)+'\n';

    }
    S=S+"P";
    return S;
  }
}
```

Nota.- Se debe importar la Class VectorBitsG que fue implementado en la Sección Anterior

7.3.3. Representación Basada en Lista Enlazada

Los nodos de la pila están enlazados con punteros, de forma que se va reservando/liberando memoria según se necesita (por lo que no es necesario implementar la operación PilaLlena). El comienzo de la lista enlazada se trata como si fuera la cabeza de la pila: todas las operaciones se realizan al principio de la lista.



- Los tipos necesarios serían en base a un Nodo definido por el usuario o haciendo uso del TDA Nodo:
- La operación Meter se corresponderá con la de insertar un nodo al principio de la lista enlazada y la de Sacar con la de recuperar el primer nodo de la lista enlazada y eliminarlo después.
- A este nivel se dispone de la idea abstracta de la estructura de datos Tipo Pila y de las operaciones que se pueden utilizar para manejarla, pero sin necesidad de saber cómo están implementadas ni cuál es la estructura con que se ha implementado la pila.

TDA Pila basado en Listas

```
public class PilaL { Nodo Cima;
            int cant;

  public PilaL() {
    this.Cima = null;
    this.cant = 0;
   }

public boolean Vacia(){
return Cima==null;
}

public void Push(int ele){
Nodo p=new Nodo();
p.setDato(ele);
p.setEnlace(Cima);
Cima=p;
cant++;
}

public int Pop(){
int x=Cima.getDato();
Cima=Cima.getEnlace();
cant--;
return x;
}

public int Get(){
return Cima.getDato();
}

@Override
public String toString() {
    String S="";
    Nodo Aux=Cima;
    while (Aux!=null) {
      S=S+Aux.getDato()+'\n';
      Aux=Aux.getEnlace();
    }
    S=S+"P";
    return S;
  };
}
```

Nota.- Se debe racializar la importación del TDA Nodo implementado anteriormente

7.3.4. Representación de Una Pila genérica

Esta representación permitirá manipular y manejar diferentes tipos de Objetos aplicados a la vida real como es el caso de un mazo de cartas, la implementación de una calculadora aritmética donde se usa varios tipos de pilas de números y operadores que trabajan en conjunto con la notación polaca inversa para poder procesar un conjunto de operaciones aritméticas que devuelve un resultados buscado.

TDA Pila Genérica

```
import java.util.EmptyStackException;
public class PilaArray<T> {
  private T[] elementos;
  private int capacidad;
  private int tamaño;

  public PilaArray(int capacidad) {
    this.capacidad = capacidad;
    this.elementos = (T[]) new Object[capacidad];
    this.tamaño = 0;
  }

  public void apilar(T elemento) {
    if (tamaño == capacidad) {
      throw new StackOverflowError("La pila está
llena");
    }
    elementos[tamaño] = elemento;
    tamaño++;
  }

  public T desapilar() {
    if (estaVacia()) {
      throw new EmptyStackException();
    }
    T elementoDesapilado = elementos[tamaño - 1];
    elementos[tamaño - 1] = null;  // Liberar la
referencia al objeto
    tamaño--;
    return elementoDesapilado;
  }

  public T cima() {
    if (estaVacia()) {
      throw new EmptyStackException();
    }

    return elementos[tamaño - 1];
  }

  public boolean estaVacia() {
    return tamaño == 0;
  }

  public int tamaño() {
    return tamaño;
  }

  public static void main(String[] args) {
    PilaArray<Integer> pila = new PilaArray<>(5);
    pila.apilar(10);
    pila.apilar(20);
    pila.apilar(30);
    System.out.println("Cima de la pila: " +
pila.cima());
    System.out.println("Tamaño de la pila: " +
pila.tamaño());

    while (!pila.estaVacia()) {
      System.out.println("Desapilando: " +
pila.desapilar());
    }
  }
}
```

Esta implementación utiliza un array para almacenar los elementos de la Pila. Utiliza un tipo genérico T que se especifica cuando se crea una instancia de la Pila. Las operaciones de Apilar, Desapilar y acceso al cima se realizan de manera similar a la implementación para una Pila de enteros.

Practico Pilas

Implementar las operaciones que se realizan sobre un array de enteros, utilizando exclusivamente el TAD Pila. Se pide:

1. Implementar un procedimiento que efectué la operación de reservar espacio para un array de N elementos.
2. Realizar una función que recupere el contenido de una determinada posición del array.
3. Escribir un procedimiento que simule la operación de escribir un dato en una determinada posición del array.

IMPORTANTE: Al utilizar un TAD, para acceder a la estructura
de datos oculta en él, sólo es posible a través de los procedimientos definidos en dicho TAD.

4.- ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es int?
Pila p = new Pila(); int x = 4, y;

p.insertar(x);
System.out.println("\n " + p.cimaPila()); y = p.quitar();
p.insertar(32); p.insertar(p.quitar()); do {
System.out.println("\n " + p.quitar()); }while (!p.pilaVacia());

5.- Escribir el método mostarPila()para escribir los elementos de una pila de cadenas de caracteres, utilizando sólo las operaciones básicas y una pila auxiliar.

6.- Utilizando una pila de caracteres, transformar la siguiente expresión a su equivalente expresión en postfija.
(x-y)/(z+w) – (z+y)^x

7.- Transformar la expresión aritmética del Ejercicio 9.3 en su expresión equivalente en notación prefija.

8.- Dada la expresión aritmética r = x*y - (z+w)/(z+y)^x, transformar la expresión a notación postfija y, a continuación, evaluar la expresión para los valores: x = 2, y = -1, z = 3, w = 1. Para obtener el resultado de la evaluación seguir los pasos del algoritmo descrito en el Apartado 9.5.3.

9.- Se tiene una lista enlazada a la que se accede por el primer nodo. Escribir un método que imprima los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar, utilizar una pila y sus operaciones.

10.- La implementación del TAD Pila con arrays establece un tamaño máximo de la pila que se controla con el método pilaLlena(. Modificar este método de tal forma que, cuando se llene la pila, se amplíe el tamaño del array a justamente el doble de la capacidad actual
Escribir un método, copiarPila(, que copie el contenido de una pila en otra. El método tendrá dos argumentos de tipo pila, uno para la pila fuente y otro para la pila destino. Utilizar las operaciones definidas sobre el TAD Pila.

11.- Escribir un método para determinar si una secuencia de caracteres de entrada es de la forma:
X & Y siendo X una cadena de caracteres e Y la cadena inversa. El carácter & es el separador.

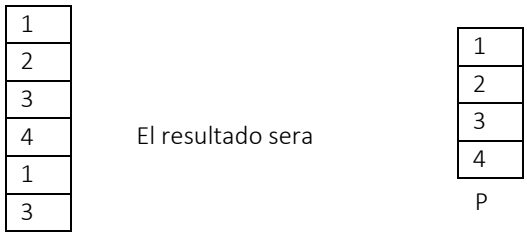
12.- Escribir un programa que, haciendo uso de una Pila, procese cada uno de los caracteres de una expresión que viene dada en una línea. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:
((a+b)*5) - 7
A esta otra expresión le falta un corchete: 2*[(a+b)/2.5 + x - 7*y

13.- Se quieren determinar las frases que son palíndromo, para lo cual se ha de seguir la siguiente estrategia: considerar cada línea de una frase; añadir cada carácter de la frase a una pila y, a la vez, a lista enlazada circular por el final; extraer carácter a carácter, simultáneamente de la pila y de la lista circular el primero, su comparación determina si es palíndromo o no. Escribir un programa que lea líneas y determine si son palíndromo.

Operaciones del TDA Pila

a. Dada una pila P realizar un método para eliminar los elementos repetidos de esta

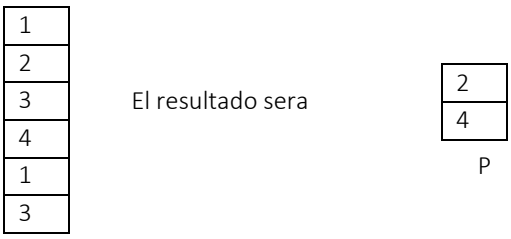
Es decir



P

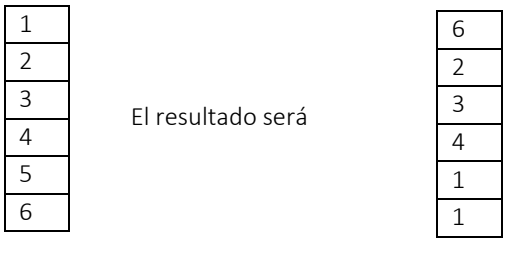
b) Dada una pila P realizar un método para eliminar los elementos repetidos inclusive el mismo

Es decir



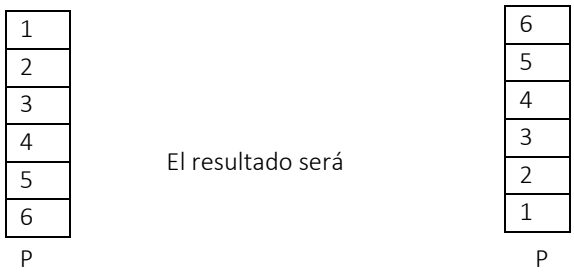
P

c) Dada una pila P realizar un método que me permita intercambiar el primero con el ultimo elemento de la pila, es decir

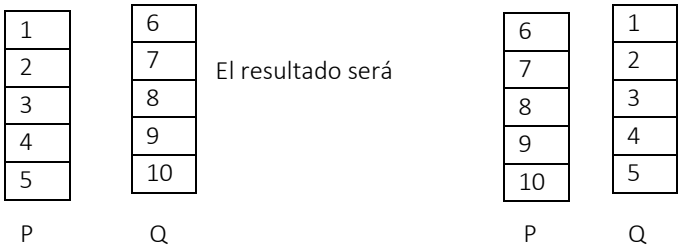


P P

d) Dada una pila P realizar un método que me permita intercambiar el primero con el ultimo elemento de la pila, es decir



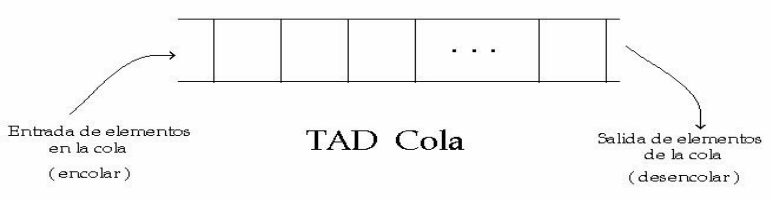
e) Dada 2 pilas P y Q realizar un método que me permita intercambiar los elementos de ambas pilas haciendo uso de una pila auxiliar como máximo requisito, es decir



CAPITULO 8

TDA COLAS

El concepto de cola es ampliamente utilizado en la vida real. Cuando nos situamos ante la taquilla del cine para obtener nuestra entrada, o cuando esperamos en el autoservicio de un restaurante solemos hacerlo en una cola. Esto significa que formamos una fila en la que el primero que llega es el primero en obtener el servicio y salir de la misma. Esta política de funcionamiento se denomina **FIFO** (First In First Out), es decir, el primer elemento en entrar es el primer elemento en salir.



En la vida real puede perfectamente ocurrir que alguien pretenda saltarse su turno en una cola, o incluso que abandone la misma antes de que le toque el turno. Sin embargo, en ambos casos se está incumpliendo la política de funcionamiento de la cola y, estrictamente hablando, ésta deja de serlo.

En el ámbito de las estructuras de datos definiremos una cola del siguiente modo:

8.1. Definición

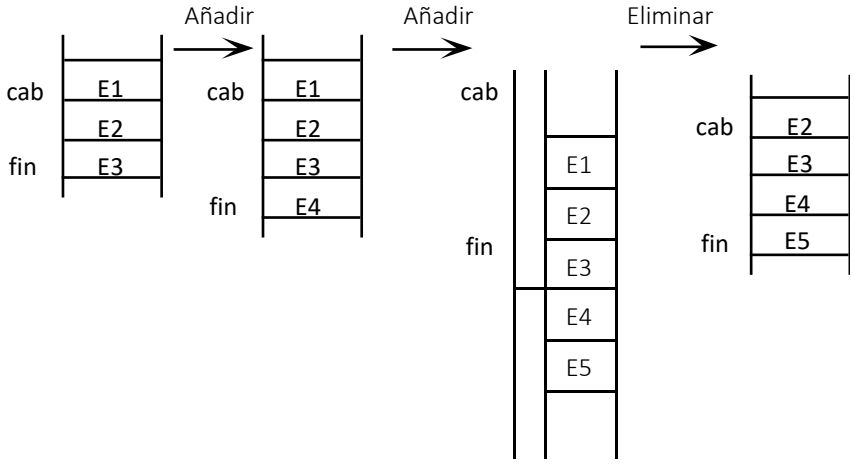
Una cola es un conjunto ordenado de elementos homogéneos, en el cual los elementos se eliminan por uno de sus extremos, denominado **cabeza**, y se añaden por el otro extremo, denominado **final**. Las eliminaciones y añadidos se realizan siguiendo una política FIFO.

Cuando hablamos de un conjunto ordenado, al igual que ocurre con las pilas, nos referimos a la disposición de sus elementos y no a su valor. Esto es, los elementos no tienen porque estar ordenados según su valor, sino que cada uno de ellos, salvo el primero y el último, tiene un anterior y un siguiente. Por otro lado, al decir que los elementos de la cola son homogéneos, queremos decir que son del mismo tipo base, aunque sin establecer ninguna limitación sobre este tipo.

Finalmente decir que en la literatura sobre estructuras de datos la cabeza de una cola suele denominarse también principio o frente de la misma, y el final suele denominarse fondo.

Al igual que las pilas, las colas se gestionan añadiendo y borrando elementos de las mismas. En este caso en particular las dos operaciones básicas de manipulación funcionan del siguiente modo:

Añadir: Añade un elemento al final de la cola.
Eliminar: Elimina un elemento de la cabeza de la cola.



La figura anterior muestra la evolución de una cola al aplicar sucesivamente las operaciones Añadir y Eliminar. Podemos ver que se trata de una estructura dinámica, ya que su tamaño cambia a medida que se añaden y eliminan elementos de la misma.
En el ámbito de la informática las colas son ampliamente utilizadas. Por ejemplo, los Sistemas Operativos suelen utilizar esta estructura de datos para gestionar los recursos que pueden ser compartidos por varios procesos (gestión de memoria, tiempo de procesador, etc.). En general, las colas se aplicarán cuando los objetos manejados sigan una política FIFO, tal y como la hemos descrito anteriormente.

8.2. Especificación del TAD Cola

Operación: Inicializar una cola

Especificación Sintáctica:
Nombre de la operación: INICOLA.
Parámetros de entrada, y tipos: no tiene
Parámetros de salida: una cola, de tipo TCola.
Declaración: Procedure IniCola (var Cola: TCola);
Especificación Semántica:
Crea una cola, que inicialmente está vacía.

Operación: Cola Vacía

Especificación Sintáctica:
Nombre de la operación: COLAVACIA.
Parámetros de entrada, y tipos: una cola perteneciente al tipo TCola.
Parámetros de salida: ninguno.
Declaración: Function ColaVacia (Cola: TCola): boolean;
Especificación Semántica:
Devuelve 'verdad' si la cola pasada como parámetro está vacía; 'falso' en caso contrario.

Operación: Encolar un elemento en la cola

Especificación Sintáctica:
Nombre de la operación: ENCOLAR
Parámetros de entrada, y tipos:
Cola que recibe al elemento, de tipo TCola.
Elemento que se introduce en la pila, de tipo Entero.
Parámetros de salida: la propia cola, de tipo TCola, con el elemento incluido.
Declaración: Procedure Encolar (Elemento: Integer; var Cola: TCola);
Especificación Semántica:

Introduce en la cola el elemento pasado como parámetro. Este elemento será el último en salir, de todos los elementos presentes en este momento.

Operación: Desencolar un elemento de la cola.
Especificación Sintáctica:
Nombre de la operación: DESENCOLAR.
Parámetros de entrada, y tipos:
Cola que contiene el elemento, de tipo TCola.
Parámetros de salida: la propia cola, con el elemento incluido.
Cola de la que se ha incluido al elemento, de tipo TCola.
Elemento que se extrae de la cola, de tipo Entero.
Declaración: Procedure Desencolar (var Cola: TPila; var Elemento: Integer);
Especificación Semántica:
Extrae de la cola un elemento. Dicho elemento es el primero que se introdujo, entre los que se encuentran actualmente en la cola.

8.3. Representaciones del TDA Cola

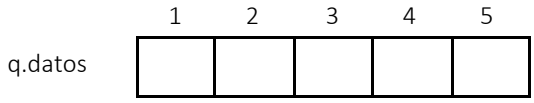
Al igual que ocurre con el tipo Pila, es posible realizar diversas implementaciones del TAD Cola. En primer lugar veremos como implementar de forma eficiente una cola utilizando vectores, y a continuación describiremos una implementación realizada mediante el uso de punteros y variables dinámicas. En ambos casos estudiaremos las ventajas e inconvenientes de utilizar cada implementación.

8.3.1. Implementación Mediante Vectores

Para realizar esta implementación, se toma como modelo la implementación del tipo Pila mediante vectores. Como en aquel caso, es más conveniente utilizar una sola entidad para contener la estructura de datos, y por tanto utilizaremos un registro e integraremos el vector como uno de sus campos. Además del vector, se necesitan unos índices que nos indiquen en cada momento qué elemento de dicho vector se está comportando como elemento cabeza y qué elemento se comporta como elemento final. Por lo tanto, el registro estará constituido por tres campos: un vector conteniendo los elementos de la cola, y dos valores, cabeza y final, que definen las posiciones del vector asociadas a ambos elementos de la cola.

Antes de ver la implementación concreta de las operaciones, es conveniente realizar una pequeña discusión sobre su funcionamiento. Se parte, como en el caso de la Pila, de que al realizar la operación CrearCola, tanto cabeza como final indicarán la posición 0 :

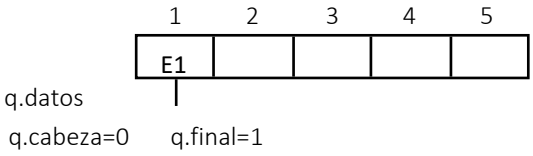
CrearCola(q):



q.cabeza=0 q.final=0

Añadir(q, E1):

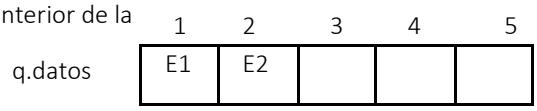
Como final indica la última posición ocupada, una política sencilla es añadir en la siguiente (final+1); por lo tanto, la operación Añadir supondrá incrementar final y añadir en la posición indicada: q.datos[q.final]



q.cabeza=0 q.final=1

Añadir(q, E2):

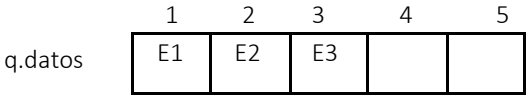
Siguiendo la definición anterior de la operación:



q.cabeza=0 q.final=2

Añadir(q, E3):

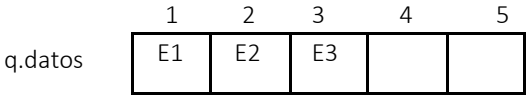
Siguiendo la definición anterior de la operación:



q.cabeza=0 q.final=3

Cabeza(q):

El elemento más antiguo en la cola es E1, que ocupa la posición 1. El valor de cabeza es 0, luego esta operación devolverá el elemento situado en la siguiente posición a la indicada por la cabeza: q.datos[q.cabeza+1], que en este caso es E1.

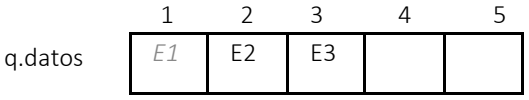


q.cabeza=0 q.final=3

Démonos cuenta que esta operación no modifica ninguno de los componentes de la cola; ni siquiera el valor de los campos cabeza y final.

Eliminar(q):

Dado que, como hemos dicho anteriormente, el campo cabeza indica la posición anterior al elemento más antiguo de la cola, la operación Eliminar supondrá eliminar de la cola el elemento situado en la posición q.cabeza+1. Para ello bastará con incrementar el valor del campo cabeza para que "apunte" al siguiente elemento de la cola. El elemento eliminado en este caso es el E1, y el más antiguo en la cola pasa a ser E2.

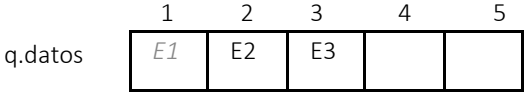


q.cabeza=1 q.final=3

Debemos darnos cuenta de que no hemos sobreescrito el valor de E1 y que este sigue en la componente 1 del vector. Sin embargo, desde un punto de vista lógico, hemos eliminado este elemento de la cola. En este momento no existe ninguna operación valida del TAD Cola que nos permita acceder al mismo. En la figura representamos los elementos eliminados en cursiva. Estos elemento ya no forman parte de la cola.

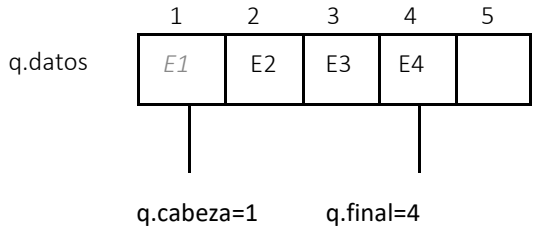
Cabeza(q):

Al devolver q.datos[q.cabeza+1], el resultado es E2.

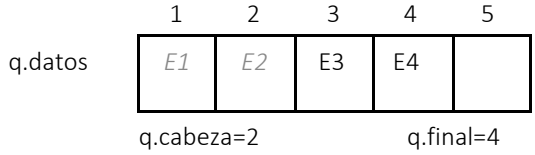


q.cabeza=1 q.final=3

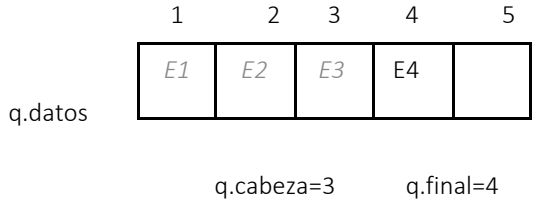
Añadir(q, E4):



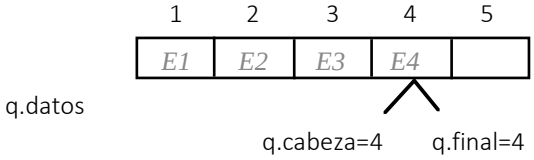
Eliminar(q):



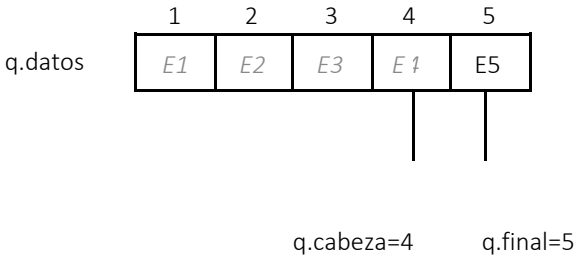
Eliminar(q):



Eliminar(q):



Añadir(q, E5):



En el ejemplo anterior hemos concretado varios aspectos relacionados con una posible implementación del tipo Cola mediante vectores:

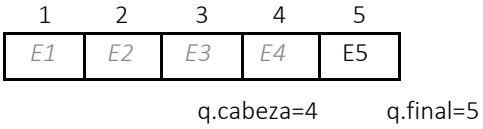
- En primer lugar utilizamos la siguiente definición de cabeza y final de la cola:

- **Cabeza(Frente)**: contiene el índice de la componente del vector inmediatamente anterior a la que contiene el elemento más antiguo de la cola. Así pues, no apunta al elemento más antiguo, sino a la posición anterior al mismo.
- **Final(Atras)**: contiene el índice de la última componente añadida a la cola. Así pues, apunta al elemento más reciente.

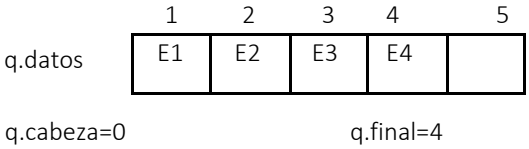
• La condición de ColaVacia es que (q.cabeza=q.final). Está condición se cumple inicialmente, cuando tanto cabeza como final valen 0 y también se cumple cuando se han eliminado todos los elementos añadidos.

Pero, ¿cuál es la condición de ColaLlena? Podemos utilizar la condición (q.final=MAX) para afirmar

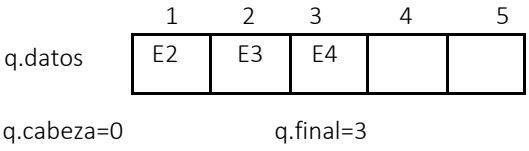
que la cola está llena, tal y como ocurre en la siguiente figura: q.datos



Sin embargo, si utilizamos la condición anterior, se da la paradoja de que hay sitio en el vector, pero no se puede añadir ningún elemento más. Una posible solución a este problema sería desplazar todos los elementos de la cola a la posición anterior del vector cada vez que eliminamos un elemento. De este modo, los elementos de la cola siempre estarían situados sobre las componentes iniciales del vector y no desaprovecharíamos espacio del mismo. Veámoslo con un ejemplo: Si partimos de una cola como la siguiente:



la operación Eliminar(q) dará como resultado:



Esta implementación alternativa nos permite mantener la cabeza de la cola siempre fija, pero supone mover múltiples elementos cada vez que queramos eliminar uno de ellos. Esto puede suponer un incremento sustancial del coste, por lo que optamos por otra solución al problema de llenado de la cola.

Implementación cola lineal basado en arrays

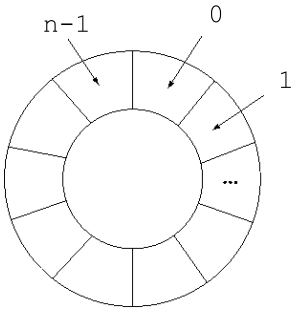
```
public class ColaLineal {
        private static fin int MAXTAMQ = 40;
        protected int frente;
        protected int fin;
        protected Int [] listaCola;

public ColaLineal() {
frente = 0; fin = -1;
listaCola = new Int [MAXTAMQ]; }
// operaciones de modificación de la cola

public void insertar(Int elemento) throws Exception {
if (!colaLlena()) {
listaCola[++fin] = elemento; }
else
throw new Exception("Overflow en la cola"); }

public Int quitar() throws Exception {
if (!colaVacia()) {
}
return listaCola[frente++]; }
else
throw new Exception("Cola vacia "); }
//vacía la cola public void borrarCola() {
frente = 0; fin = -1;
}
// acceso a la cola
public Int frenteCola() throws Exception {
if (!colaVacia()) {
return listaCola[frente]; }
else
throw new Exception("Cola vacia "); }
// métodos de verificación del estado de la cola
public boolean colaVacia()
{
return frente > fin; }
public boolean colaLlena() {
return fin == MAXTAMQ-1; }
```

La solución que vamos a describir implica reutilizar las componentes del vector que contenían elementos ya eliminados. Esto es, cuando durante el proceso de añadido lleguemos al final del vector, comenzaremos a llenar de nuevo las componentes iniciales del mismo si se encuentran vacías. Para logra esto, manejaremos el vector como si fuese un "vector circular". Esto significa que no consideraremos la componente MAX del vector como la última del mismo, sino que consideraremos que la siguiente componente a está es otra vez la primera del vector

8.3.2.Cola basado en Array Circular

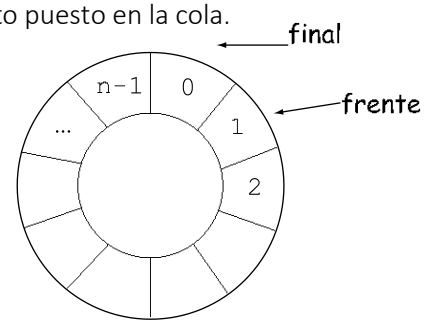
La alternativa, sugerida en la operación de quitar un elemento, de desplazar los restantes elementos del array de modo que la cabeza de la cola vuelva al principio del array, es costosa en términos de tiempo de computadora, especialmente si los datos almacenados en el array son estructuras de datos grandes.

La forma más eficiente de almacenar una cola en un array es modelarlo de tal forma que se una el extremo final con el extremo cabeza. Tal array se denomina array circular y permite que la totalidad de sus posiciones se utilicen para almacenar elementos de la cola sin necesidad de desplazar elementos. La siguiente Figura muestra un array circular de n elementos.



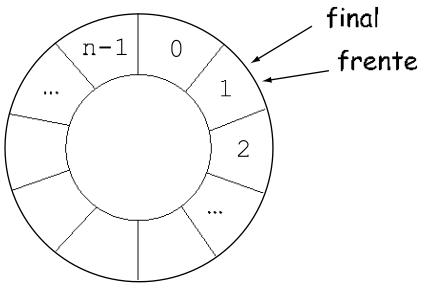
Un array circular

El array se almacena de modo natural en la memoria como un bloque lineal de n elementos. Se necesitan dos marcadores (apuntadores) frente y fin para indicar, respectivamente, la posición del elemento cabeza y del último elemento puesto en la cola.



Una cola vacía

El *frente* siempre contiene la posición del primer elemento de la cola y avanza en el sentido de las agujas del reloj; *fin* contiene el aposición donde se puso el último elemento y también avanza en el sentido del reloj (circularmente a la derecha). La implementación del movimiento circular se realiza según la *teoría de los restos,* de tal forma que se generen índices de 0a MAXTAMQ-1



Una cola que contiene un elemento

La implementación de la estructura de datos es la misma que hemos utilizado anteriormente. Así, se utiliza un registro con tres campos: uno para guardar los elementos de la cola, y dos para guardar la posición de la *cabeza(Frente)* y el *final(Atras*) de la misma. La diferencia estriba en cómo se manejan estos campos en las operaciones de manipulación del TAD.

Al utilizar un vector circular para la implementación sera la siguiente}

TDA Cola Circular Basada en Arrays

```
public class ColaV { int C[];
                     int Frente;   // Indice a la cabeza de
la Cola
                     int Atras;    //Indice al final de la cola

  public ColaV(int cant) {
    this.C = new int[cant];
    this.Frente = -1;
    this.Atras = -1;
  }

  public boolean vacia(){
  return (Frente==-1);
  }
```

```
public boolean Llena(){
return((Atras+1)%C.length==Frente);
}

public void Encolar(int ele){
 if (vacia()){
    Frente=0;
    Atras=0;
    C[Atras]=ele;
  }else
              { if (Llena()){
              System.out.println("Error::Encolar:Cola
          Llena");
              System.exit(1);
              }else{
               Atras=(Atras+1)%C.length;
```

```
                C[Atras]=ele;
               }
      }
  }

  public int Decolar(){
  int x=C[Frente];
  if (Frente==Atras){
  Frente=-1;
  Atras=-1;
  }else
    Frente=(Frente+1)%C.length;
  return x;
  }

  public int Get(){
  return(C[Frente]);
  }

  @Override
  public String toString() {
    String S="Q=[";
    int Aux=Frente;
    while (Aux!=Atras){
      S=S+C[Aux]+" , ";
      Aux=(Aux+1)%C.length;
      }
      S=S+C[Aux]+"]";
      return S;
    }
  }

  public static void main(String[] args) {
      // TODO code application logic here
      ColaV Q=new ColaV(10);
      Q.Encolar(9);
      Q.Encolar(1);
      Q.Encolar(2);
      Q.Encolar(8);
      Q.Encolar(3);
      Q.Encolar(0);
      Q.Encolar(1);
      Q.Encolar(18);
      Q.Encolar(31);
      Q.Encolar(7);
      System.out.println(Q);
  }
```

8.3.3. Representación del TDA Cola a Nivel de Bits

Así también para poder optimizar esta representación, es posible poder implementar esta estructura Cola a nivel de bits donde el array estará subdividido en subvectores para almacenar los datos requeridos según sea necesario en base a la cantidad de bits que se requieran para cada dato.

Representación de una Cola a Nivel de Bits

```
public class ColaB { VectorbitsGe C;
              int Frente;
              int Atras;

  public ColaB(int cant, int nbits) {
    this.C = new VectorbitsGe(cant,nbits);
    this.Frente = 0;
    this.Atras = 0;
  }

public boolean vacia(){
  return (Frente==0);
  }

public boolean Llena(){
  return((Atras+1)%C.dim==Frente);

  }

public void Encolar(int ele){
   if (vacia()){
     Frente=1;
     Atras=1;
     C.Insertar(ele,Atras);

   }else
   { if (Llena()){
   System.out.println("Error::Encolar:Cola Llena");
   System.exit(1);

   }else{
    Atras=(Atras+1)%C.dim;
    C.Insertar(ele,Atras);
    }
   }
  }

  public int Decolar(){
  int x=C.Get(Frente);
  if (Frente==Atras){
  Frente=0;
  Atras=0;
  }else
    Frente=(Frente+1)%C.dim;
```

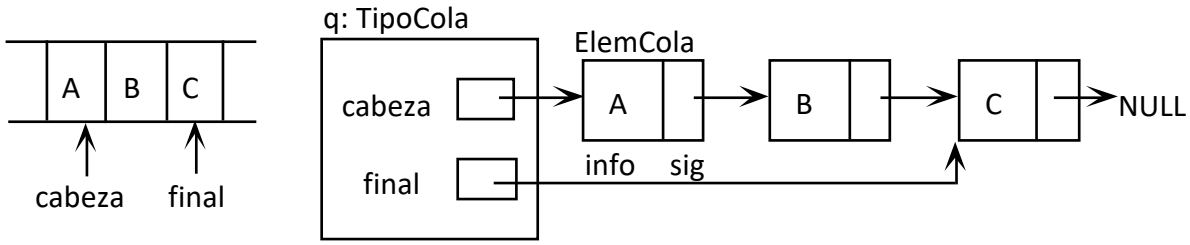
```
  return x;
  }

 public int Get(){
  return(C.Get(Frente));
  }

  @Override
  public String toString() {
     String S="Q=[";
        int Aux=Frente;
        while (Aux!=Atras){
        S=S+C.Get(Aux)+" , ";
        Aux=(Aux+1)%C.dim;
        }
        S=S+C.Get(Aux)+"]";
        return S;
     }
  }
```

8.3.4. Representación del TDA Cola mediante Listas

En este apartado estudiaremos la implementación del TAD Cola mediante el uso de variables dinámicas y punteros. Al igual que en la definición de la Pila, al utilizar punteros para definir una Cola, se llega a una definición recursiva:

```
public class ColaL { Nodo Cabeza;
                     Nodo Final;
                     int cant;
```

Gráficamente:

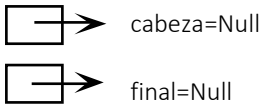


Implementación mediante punteros

Con esta definición se obtiene la siguiente implementación para las operaciones:

```
public ColaL() {
   this.Cabeza = null;
   this.Final = null;
   this.cant = 0;
}
```

Al crear la cola tanto la cabeza como el final de la misma están indefinidos. Así pues, se asigna el valor NIL a los punteros asociados.

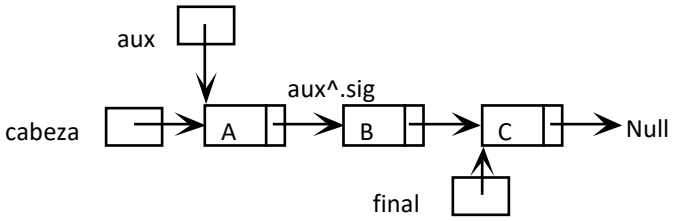


Consideraremos que la cola está vacía cuando tanto la cabeza como el final apunten a Null. Este caso se dará nada más ser creada y cuando se acabe de eliminar su último elemento.

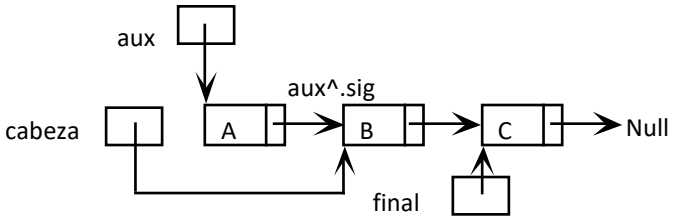
Para conocer la cabeza de la cola basta con acceder al nodo apuntado por q.cabeza. El campo info de este nodo contendrá el elemento de la cola deseado. Esta operación debe devolver un error cuando la cola se encuentre vacía.

Para eliminar un elemento hay que hacer que la cabeza de la cola apunte al siguiente elemento de la misma. A continuación se borra el elemento que ocupa la cabeza en la actualidad. Veamos los pasos seguidos:

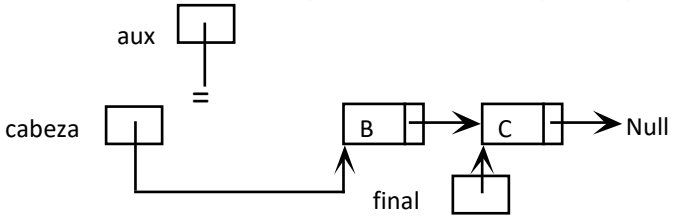
- En primer lugar se usa una variable auxiliar de tipo TipoPuntero para acceder al elemento de la cola situado en la cabeza:



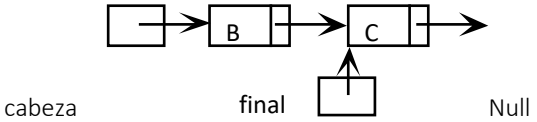
- A continuación se hace que la cabeza pase a apuntar al siguiente elemento, dado por aux^.sig.



- Finalmente se libera la memoria correspondiente al nodo que ocupa la cabeza.

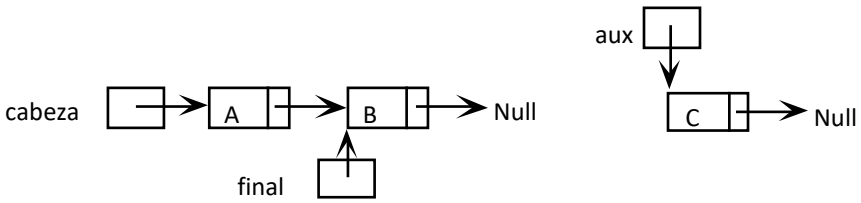


- Al salir del procedimiento se libera la memoria de la variable local aux.

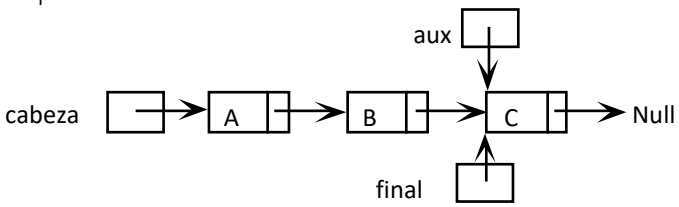


Al implementar la rutina de eliminado hemos de tener en cuenta dos casos particulares.

- Si la cola estaba vacía, el procedimiento devolverá un error
- Si la cola contenía un solo elemento, tras eliminarlo quedará vacía. Cuando nos encontremos en este caso, el procesamiento normal nos llevará a que q.cabeza valga Null. Cuando se produzca este hecho debemos hacer que también q.final valga Null.
- Para añadir un nuevo elemento, las tres primeras instrucciones del procedimiento crean y rellenan un nuevo nodo apuntado por aux:



- A continuación, si la cola estaba vacía, se hace que la cabeza apunte al nuevo nodo. Si la cola no estaba vacía, el último nodo de la cola debe apuntar al que se va a añadir. En ambos casos, se hace apuntar el final de la cola al nuevo nodo.



Con esta implementación mediante variables dinámicas, no es necesario tener en cuenta el caso de la cola llena, ya que el número de elementos a almacenar no está limitado en principio.

Implementación basada en Listas

```
public class ColaL { Nodo Frente;
            Nodo Atras;
            int cant;

  public ColaL() {
    this.Frente = null;
    this.Atras = null;
    this.cant = 0;
  }

  public boolean Vacia(){
  return Frente==null;
  }
  public void Encolar(int ele){
  Nodo p=new Nodo();
  p.setDato(ele);
  if (Vacia()){
   Frente=p;
   Atras=p;
   cant++;

  }else{    Atras.setEnlace(p);
          Atras=p;
          cant++;
  }
  }
```

```
  public int Decolar(){
    int x=Atras.getDato();
    if(Frente==Atras){
       Frente=null;
       Atras=null;
       cant--;
  }else{ Frente=Frente.getEnlace();
       cant--;
    }
   return(x);

  }
  public int Get(){
  return Atras.getDato();
  }

  @Override
  public String toString() {
    String S="C->";
    Nodo aux=Frente;
    while (aux!=null){
    S=S+aux.getDato()+" -> ";
    aux=aux.getEnlace();
      }
    return S;
  }
}
```

8.3.5. Representación de una TDA Cola Genérica

Esta representación nos permitirá no solo almacenar elementos enteros o algún tipo definido en tiempo de compilación, sino que nos permitirá almacenar cualquier tipo de datos según sea necesario como ser un mazo de cartas, una cola de procesos, una cola de impresión, un programa de emisión de fichas y diversas utilidades que en la vida real se requieren

Implementación de una Cola Genérica

```
public class ColaGenerica<T> {
  private T[] elementos;
  private int capacidad;
  private int frente;
  private int fin;
  private int tamaño;

  @SuppressWarnings("unchecked")
  public ColaGenerica(int capacidad) {
    this.capacidad = capacidad;
    this.elementos = (T[]) new Object[capacidad];
    this.frente = 0;
    this.fin = -1;
    this.tamaño = 0;
  }

  public void encolar(T elemento) {
    if (tamaño == capacidad) {
      throw new IllegalStateException("La cola está
llena");
    }

    fin = (fin + 1) % capacidad;
    elementos[fin] = elemento;
    tamaño++;
  }

  public T desencolar() {
    if (estaVacia()) {
      throw new IllegalStateException("La cola está
vacía");
    }

    T elementoDesencolado = elementos[frente];
    frente = (frente + 1) % capacidad;
    tamaño--;
    return elementoDesencolado;
  }

  public T frente() {
    if (estaVacia()) {
      throw new IllegalStateException("La cola está
vacía");
    }

    return elementos[frente];
  }

  public boolean estaVacia() {
    return tamaño == 0;
  }

  public int tamaño() {
    return tamaño;
  }

 @Override
  public String toString() {
    return "ColaGenerica{" +
        "elementos=" + Arrays.toString(elementos) +
        ", capacidad=" + capacidad +
        ", frente=" + frente +
        ", fin=" + fin +
        ", tamaño=" + tamaño +
        '}';
  }

  public static void main(String[] args) {
    ColaGenerica<String> cola = new
ColaGenerica<>(5);
    cola.encolar("A");
    cola.encolar("B");
    cola.encolar("C");
    System.out.println("Frente de la cola: " +
    cola.frente());
    System.out.println("Tamaño de la cola: " +
    cola.tamaño());
    while (!cola.estaVacia()) {
    System.out.println("Desencolando: " +
    cola.desencolar());
    }
  }
}
```

Esta implementación utiliza un array para almacenar los elementos de la cola. Utiliza un tipo genérico T que se especifica cuando se crea una instancia de la cola. Las operaciones de encolar, desencolar y acceso al frente se realizan de manera similar a la implementación para una cola de enteros.

Practica de Colas

1. Un concesionario de coches tiene un número limitado m de modelos, todos en un número limitado c de colores distintos. Cuando un cliente quiere comprar un coche, pide un coche de un modelo y color determinados. Si el coche de ese modelo y color no está disponible en el concesionario, se toman los datos del cliente (nombre y dirección), que verá atendida su petición cuando el coche esté disponible. Si hay más de una petición de un coche de las mismas características, se atienden las peticiones por orden cronológico. Se pide:

a) Definir la estructura de datos más adecuada capaz de contener las peticiones de un modelo y color de coche.

b) Definir la estructura de datos global más adecuada, capaz de contener las peticiones para todos los modelos y colores de coches del concesionario.

c) Definir una operación que, dado un cliente (nombre y dirección) que desea comprar un coche de un modelo y color determinado, coloque sus datos como última petición de ese modelo y color.

d) Definir una operación que, dado un modelo del que se han recibido k coches de determinado color, elimine los k primeros clientes de la lista de peticiones de ese coche y los devuelva en un vector, sabiendo que $k \leq 20$.

3. Se desea tener una estructura de datos cola que tenga disponible en todo momento qué cantidad de elementos contiene. Se pide:

a) Definición de tipos necesarios, sabiendo que se trata de una estructura dinámica.

b) Definición de las operaciones de cola, y además de una operación longitud, que devuelva el número de elementos de la cola.

4. Una librería registra las peticiones de cualquier libro que no tiene en ese momento. La información de cada libro consiste en el título del libro, el precio (en pesetas), el número de libros en stock, y las peticiones del libro en estricto orden de llegada. Cada petición consiste en el nombre de una persona y su dirección.

a) Define los tipos de datos capaces de describir toda esta información para un total de 2000 libros. Justifica la elección de estructuras para las peticiones y para la estructura general que incluye los datos de todos los libros.

b) Implementa una operación que dado un cliente que pide un libro, vea si hay en stock, y si quedan, actualice el stock con la venta de ese libro, y si no, guarde los datos del cliente como última petición de ese libro.

c) Implementa otra operación que dado un libro del que se reciben k ejemplares vea si hay peticiones y si las hay escriba en pantalla los k primeros clientes y elimine sus peticiones. Si hay más de k peticiones el stock seguirá a 0. Si hay menos de k peticiones, habrá que actualizar el stock con los libros que queden.

5. Una agencia de viajes ofrece n destinos; para cada destino se puede optar por 5 clases de viaje, super, luxe, normal, turista y estudiante, y además se ofrecen tres tipos de alojamiento: AD (alojamiento y desayuno), MP (media pensión) y PC (pensión completa). Cada programa de viaje se caracteriza por la

información (destino, clase, alojamiento). Por cada programa de viaje se quiere saber el número de plazas disponibles, de manera que cuando un cliente contrata un determinado programa, el número de plazas disponibles se decrementa. Cuando un programa no dispone de plazas, entonces la información del cliente (nombre, dirección y NIF) se almacena en orden cronológico. Así, cuando se disponga de nuevas plazas en ese programa se atenderán las peticiones en orden.

Se desea informatizar la gestión de esta agencia. Entre otras cosas, es preciso:

a) La definición de la estructura de datos que soporte la información descrita (es decir, la estructura que permita almacenar todos los programas de viaje de la agencia).

b) Escribir un algoritmo que, dado un cliente y un determinado programa de viajes, compruebe si es posible o no que el cliente lo contrate. En cualquier caso, habrá que realizar las acciones oportunas para que la estructura se actualice de forma conveniente.

c) Escribir un algoritmo que indique todos los destinos con alguna plaza disponible. La solución debe incluir la definición de la estructura de datos más idónea para devolver la información, sabiendo que n < 100.

6. Un restaurante dispone de m mesas. Por cada mesa se sabe su código de identificación y su capacidad en comensales.

Hay una cola de espera para ir ocupando las mesas, de forma que, para cada elemento, se sabe el nombre de la persona que ha hecho la reserva y el número de comensales. Se pide:

a) Definir las estructuras de datos restaurante y espera.

b) Definir la operación Maître que, dado un identificador de mesa, devuelve el nombre de una persona que haya hecho una reserva y que ya puede pasar al comedor junto con sus acompañantes. La estructura espera debe quedar convenientemente actualizada. Se deben hacer dos versiones de la operación:

b.1) La elección se hace buscando la primera reserva que cabe en la mesa.

b.2) La elección se hace optimizando la ocupación de las mesas.

7. Se desea realizar un enriquecimiento del TAD cola que llamaremos cola con prioridad. En la cola con prioridad habrá dos tipos de elementos, los que tienen prioridad y los que no la tienen. Además tendremos dos operaciones para insertar elementos, la operación Añadir, que inserta elementos sin prioridad (hace lo mismo que en la cola normal), y la operación Añadir_con_prioridad, la cual inserta un elemento de modo que queda colocado antes de todos aquellos que no tiene prioridad y después de aquellos con prioridad que ya estuvieran en la cola. Se pide:

a) Definir la estructura de datos cola con prioridad utilizando variables dinámicas.

b) Modificar las operaciones del TAD Cola que sean necesarias para que se adecúen al enriquecimiento.

c) Implementar la operación Añadir_con_prioridad mediante el procedimiento que debe tener el siguiente perfil:

procedure Añadir_con_prioridad(var q:Cola_prioridad; e:TB);

8.- En un supermercado hay 20 cajas registradoras, en cada una de las cuales se colocan los clientes con sus carros de la compra en orden de llegada. Por cada caja registradora queremos guardar el nombre de la cajera, la recaudación acumulada y los carros en espera.

Por otro lado, en cada carro se amontonan los distintos productos, de modo que tan sólo puede añadirse o extraerse el situado en la parte superior. Por cada producto guardamos su nombre y precio.

a) Estructuras de datos. Utilizar en cada subapartado las estructuras de los anteriores.

a.1) Definir la estructura más adecuada para guardar un carro de la compra.

a.2) Definir la estructura más adecuada para guardar una caja registradora.

a.3) Definir la estructura más adecuada para guardar el supermercado.

b) Implementar un algoritmo denominado atender_cliente que dado un carro de la compra, pase los productos que contiene por caja y calcule el precio total.

c) Implementar un algoritmo que calcule la recaudación de las cajas después de pasar los primeros x carros por cada una de ellas. x será un argumento de entrada que puede ser mayor que el número de carros en espera en algunas de las cajas.

Utilizar la estructura de datos más adecuada para devolver el resultado pedido.
Nota. Utilizar de modo adecuado el procedimiento implementado en el apartado b)

9. Una Bicola es una estructura de datos similar a una Cola, pero en la que pueden insertarse y eliminarse elementos en los dos extremos (y en ninguna otra posición).

Supongamos la siguiente especificación en lenguaje natural de las operaciones de manejo de una bicola.

Crearbicola(bicola):
Crea una bicola vacía.
Bicolavacia(bicola):
Devuelve verdadero si la bicola está vacía y falso en caso contrario.
Elimizq(bicola):
Elimina el elemento situado en el extremo izquierdo de la bicola.
Elimder(bicola):
Elimina el elemento situado en el extremo derecho de la bicola.
Añadizq(bicola, x):
Añade el elemento x en el extremo izquierdo de la bicola.
Añadder(bicola,x):
Añade el elemento x en el extremo derecho de la bicola.
Cabizq(bicola, x):
Devuelve en x el elemento situado en el extremo izquierdo de la bicola sin eliminarlo.
Cabder(bicola, x):
Devuelve en x el elemento situado en el extremo derecho de la bicola sin eliminarlo.

10.- Definir la estructura de datos dinámica más adecuada para almacenar una bicola.
a) Justificar la respuesta.

b) Implementar la operación Elimder(bicola);

c) Dada una bicola de caracteres, implementar un algoritmo que nos diga si la cadena de caractéres que contiene es un palindromo (capicúa).

Ejemplos de cadenas capicúa son: a, aa, aba, abba.

11.- Una bicola es una estructura de datos lineal en la que la inserción y el borrado se pueden hacer tanto por el extremo frentecomo por el extremo fin. Suponer que se ha elegido una representación de los elementos con una lista doblemente enlazada y que los extremos de la lista se denominan frentey fin. Escribir la clase Bicolacon la representación de los datos y la implementación de las operaciones del TAD Bicola.

12. Suponga que tiene ya codificados los métodos que implementan las operaciones del TAD Cola. Escribir un método para crear una copia de una cola determinada. Las operaciones que se han de utilizar serán únicamente las del TAD Cola.

13. Considere una bicola de caracteres, representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y los elementos de la bicola son:

frente = 5 fin = 7 Bicola: A,C,E

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos F y K por el final de la bicola.
- Añadir los elementos R, W y V por el frente de la bicola.
- Añadir el elemento M por el final de la bicola.
- Eliminar dos caracteres por el frente.
- Añadir los elementos K y L por el final de la bicola.
- Añadir el elemento S por el frente de la bicola.

14. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos que son par de la pila en la cola.

15. Implementar el TAD Cola utilizando una lista enlazada circular. Por conveniencia, establecer el acceso a la lista, lc, por el último nodo (elemento) insertado y considerar al nodo siguiente de lc el primero o el que mas tarde se insertó.

16.- Escribir un método que tenga como argumentos dos colas del mismo tipo y devuelva cierto si las dos colas son idénticas.

17.- Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en que un cliente paga en la caja, su carrito de la compra queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

17. En un archivo Están almacenados números enteros arbitrariamente grandes. La disposición es tal que hay un número entero por cada línea de . Escribir un programa que muestre por pantalla la suma de todos los números enteros. Al resolver el problema habrá que tener en cuenta que, al ser enteros grandes, no pueden almacenarse en variables numéricas.

Utilizar dos pilas para guardar los dos primeros números enteros, almacenándose dígito a dígito. Al extraer los elementos de la pila, salen en orden inverso y, por tanto, de menor peso a mayor peso; se suman dígito con dígito y el resultado se guarda en una cola, también dígito a dígito. A partir de este primer paso se obtiene el siguiente número del archivo, se guarda en una pila y, a continuación, se suma

dígito a dígito con el número que se encuentra en la cola; el resultado se guarda en otra cola. El proceso se repite, nuevo número del archivo se mete en la pila, que se suma con el número actual de la cola.

18.- Una empresa de reparto de propaganda contrata a sus trabajadores por días. Cada repartidor puede trabajar varios días continuados o alternos. Los datos de los repartidores se almacenan en una lista enlazada. El programa a desarrollar contempla los siguientes puntos:

- Crear una cola que guarde el número de la seguridad social de cada repartidor y la entidad anunciada en la propaganda para un único día de trabajo.
- Actualizar la lista citada anteriormente (que ya existe con contenido) a partir de los datos de la cola.

La información de la lista es la siguiente: número de seguridad social, nombre y to-tal de días trabajados. Además, está ordenada por el número de la seguridad social. Si el trabajador no está incluido en la lista, debe añadirse a la misma de tal manera que siga ordenada

19.- El supermercado Esperanza quiere simular los tiempos de atención al cliente a la hora de pasar por la caja. Los supuestos de los que se parte para la simulación son los siguientes:

- Los clientes forman una única fila. Si alguna caja está libre, el primer cliente de la fila es atendido. En el caso de que haya mas de un caja libre, la elección del número de caja por parte del cliente es aleatoria.
- El número de cajas del que se dispone para atención a los clientes es de tres, salvo que haya mas de 20 personas esperando en la fila; entonces se habilita una cuarta caja, que se cierra cuando no quedan clientes esperando. El tiempo de atención de cada una de las cajas está distribuido uniformemente: la caja 1 entre 1,5 y 2,5 minutos; la caja 2 entre 2 y 5 minutos, la caja 3 entre 2 y 4 minutos. La caja 4, cuando está abierta, tiene un tiempo de atención entre 2 y 4,5 minutos.
- Los clientes llegan a la salida en intervalos de tiempo distribuidos uniformemente, con un tiempo medio de 1 minuto.

El programa de simulación se debe realizar para 7 horas de trabajo. Se desea obtener una estadística con los siguientes datos:

- Clientes atendidos durante la simulación.
- Tamaño medio de la fila de clientes.
- Tamaño máximo de la fila de clientes.
- Tiempo máximo de espera de los clientes.
- Tiempo en que está abierto la cuarta caja.

Operaciones entre Pilas y Colas

20.- Dada una Pila P y Una Cola Q, implementar un método de la clase Cola que me permita intercambiar el contenido de ambas estructuras, es decir el contenido de la Pila se ira a la Cola y viceversa

21.- Dada una cola Q [1,2,3,4,5]. Implementar un método que me permita intercambiar el primer y ultimo elemento de la cola sin uso de ninguna estructura Auxiliar siempre tomando en cuenta el funcionamiento de la cola que es FIFO(el primero en entrar es el primero en salir), el resultado será Q [5,2,3,4,1].

22.- Dada una cola Q [1,2,3,4,5]. Implementar un método que me permita Invertir la cola en su contenido es decir el resultado sera Q [5,4,3,2,1]. sin uso de ninguna estructura Auxiliar diferente que no sea una cola siempre tomando en cuenta el funcionamiento de la cola que es FIFO(el primero en entrar es el primero en salir)

23.- Dada 2 colas Q1 [1,2,3,4,5]. Y Q2 [6,7,8,9,10,11]. No necesariamente de la misma dimensión, Implementar un método que me permita intercambiar el contenido entre ambas colas, es decir el contenido mde Q1 se ira a Q2 y viceversa el resultado será Q1 [6,7,8,9,10,11] y Q2 [1,2,3,4,5] sin uso de ninguna estructura Auxiliar diferente que no sea una cola siempre tomando en cuenta el funcionamiento de la cola que es FIFO(el primero en entrar es el primero en salir)

24.- Dada una cola Q [1,2,3,4,5] y una pila P [6,7,8,9,10] donde 10 es el ultimo elemento ingresado a la pila, No necesariamente de la misma dimensión, Implementar un método que me permita intercambiar los extremos de ambas estructuras , es decir intercambiar el primer elemento de la cola Q con el ultimo elemento ingresado de la pila P y también intercambiar el Ultimo elemento de la Cola Q con el Primer Elemento Ingresado de la Cola Q, el resultado será Q [10,2,3,4,6] y una pila P [1,7,8,9,5].sin uso de ninguna estructura Auxiliar diferente que no sea una cola siempre tomando en cuenta el funcionamiento de la cola que es FIFO(el primero en entrar es el primero en salir)

25.- Dada una cola Q [1,2,2,3,4,4,5], implementar un método de la clase cola para eliminar todos los elementos repetidos de la cola sin el uso de ninguna estructura Auxiliar, es decir el resultado será Q [1,,2,3,4,5],