Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Industrial

Estructuras de Datos Abstractas en Lenguaje Java

Listas Enlazadas, Colas, Pilas y Árboles Binarios

Creado por Carlo Casorzo G. para el curso Fundamentos de Informática Industrial

Índice

Consideraciones previas

1) Sobre la estructura de programación usada.

La forma en que se muestran las estructuras de datos Lista Enlazada, Cola y Pila es sólo una manera alternativa de implementación. La idea de este tutorial es complementar los ejemplos de implementación propuestos en el laboratorio y cátedra, para que el alumno se de cuenta que puede diseñar estas estructuras de la manera que estime más conveniente o que más se ajuste al problema a solucionar, mientras entienda el concepto fundamental de la implementación de ellas.

2) Pre requisitos para entender este tutorial

Para la total comprensión de este tutorial, es necesario que el alumno domine los temas propuestos en el libro Sam's Teach Yourself Java in 21 Days, desde el capítulo 1 hasta el 7, en especial la sección "Referencia a objetos", capitulo 4, página 99. También es importante echar un vistazo al capítulo 16, "Circunstancias excepcionales, manejo de errores y seguridad".

3) Sobre los códigos y la aplicación ilustrativa

Los códigos de lista enlazada, cola y pila usados en este tutorial están incluidos en las carpetas **"Lista Enlazada Simple", "Cola"** y **"Pila".** La aplicación ilustrativa está en la carpeta **"Aplicación Ilustrativa"**. Para ejecutarla, solo se debe hacer doble clic en **AplicaciónIlustrativa.exe.**

1. Almacenamiento de datos

1.1 El problema de los arreglos

Los arreglos y las estructuras de datos (listas, colas, pilas, etc.) son entes informáticos abstractos que nos permiten almacenar datos, es decir, en un lenguaje de programación como Java, objetos y/o tipos primitivos (**int**, **double**, **char**, **boolean**, etc...).

Los arreglos son, probablemente, la estructura más usada para almacenar y ordenar datos dentro de la programación, debido a que la sintaxis para acceder a sus datos es muy amigable para el programador. Por ejemplo, si tenemos un arreglo de **int**'s, y queremos sumar el segundo número de un arreglo con el cuarto, simplemente usamos la notación **[ ]** para acceder a ellos:

```
int suma= numeros[1] + numeros[3];
```

O, por ejemplo, si queremos llenar un arreglo con todas las letras minúsculas, desde la a la z, simplemente hacemos:

```
for(int j=65;j<=90;j++)
    minusculas[j-65]=(char)j;
```

(*)

(*) ver "Conversión por cast", capitulo 4, página 100

Y así, **`minúsculas`** sería igual a **`{'a', 'b', 'c', 'd',..., 'z'}.`**
Si tratamos de representar gráficamente lo que un arreglo significa en la memoria, podríamos analizar el siguiente esquema:

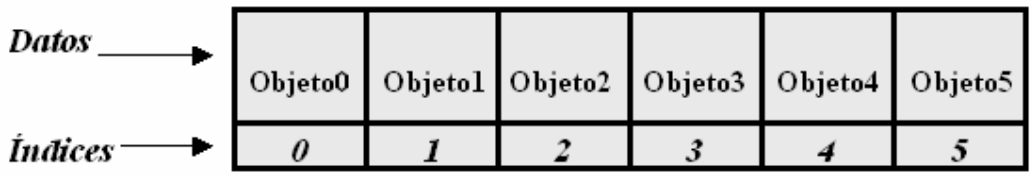

Fig. 1.1.1 Arreglos

Y para acceder a un objeto en el arreglo, simplemente hacemos referencia a su índice y usamos la notación **[ ]**.

Pero esta forma de almacenamiento tiene ciertas desventajas, por ejemplo:

1) Los arreglos tienen una capacidad de almacenamiento determinada y no modificable, lo que se transforma realmente en un problema si queremos almacenar datos sin saber cuantos almacenaremos en total, o si la cantidad de datos es variable.

2) Por (1), los programadores utilizan arreglos "lo suficientemente grandes", desperdiciando espacio de memoria que nunca se utiliza. Por ejemplo, si queremos registrar a los clientes que compraron en nuestra tienda cada día, podríamos crear un arreglo de una capacidad estimada de 100 espacios para cada día. Pero en el caso que en una ocasión sólo entraran a comprar 22 clientes, 78 espacios del arreglo son desperdiciados, y, por lo tanto, espacio en la memoria es desperdiciado. O peor aún, que entraran 150 personas a comprar y que nuestro arreglo no fuera capaz de almacenar a todos los clientes. En estos casos, los arreglos no son la herramienta indicada.

3) La inserción de un objeto al principio del arreglo, sin sobrescribir el primer espacio, se torna mas complicada, pues debemos correr en un espacio a la derecha a todo el resto de los datos. Es decir, los arreglos no manejan los datos de forma dinámica.

Para arreglar este problema, haremos uso de una nueva forma de almacenamientos: las estructuras datos basadas en punteros, con su principal exponente, la Lista Enlazada.

1.2 Una solución: la Lista Enlazada y estructuras de datos en base a punteros

Para entender este tipo de estructuras es necesario tener claro algunos conceptos:

puntero/apuntado: un *puntero* es un espacio en la memoria que almacena una referencia o dirección de memoria de otra variable, conocida como su *apuntado*. El valor de un *puntero* puede ser **null,** lo que significa actualmente no se refiere a ningún *apuntado*.

referencia: la operación de referencia en un puntero sirve para tener acceso a su apuntado.

Asignación de puntero: una operación de asignación entre dos punteros, como `p=q`, hace que los dos punteros se refieran (o apunten) al mismo apuntado. No se copia dos veces en la memoria al apuntado, sino que los dos punteros almacenan la dirección de memoria del apuntado.

Ahora bien, para solucionar el problema encontrado al usar arreglos en ese tipo de situaciones, introduciremos un nuevo tipo de estructura de almacenamiento: Las **Estructuras de Datos en Base a Punteros.**

En este apunte, abarcaremos tres de estas estructuras: Lista Enlazada Simple, Cola y Pila. Éstas dos últimas se basan en la idea de la Lista Enlazada, así que las dejaremos para el final.

2. Lista Enlazada Simple

2.1 Qué es una lista enlazada

La Lista Enlazada Simple es la más fundamental estructura de datos basada en punteros, y del concepto fundamental de ésta derivan las otras estructuras de datos.

Para solucionar un problema como el presentado anteriormente, necesitamos una estructura que, al contrario de los arreglos, sea capaz de modificar su capacidad, es decir, que maneje los datos de forma dinámica. Para lograr esto, nace la idea de lista enlazada.

Un arreglo asigna memoria para todos sus elementos ordenados como un sólo bloque. En cambio, la lista enlazada asigna espacio para cada elemento por separado, en su propio bloque de memoria, llamado **nodo**. La lista conecta estos nodos usando punteros, formando una estructura parecida a la de una cadena.

Un nodo es un objeto como cualquier otro, y sus atributos serán los encargados de hacer el trabajo de almacenar y apuntar a otro nodo. Cada nodo tiene dos atributos: un atributo **"contenido"**, usado para almacenar un objeto; y otro atributo **"siguiente"**, usado para hacer referencia al siguiente nodo de la lista.

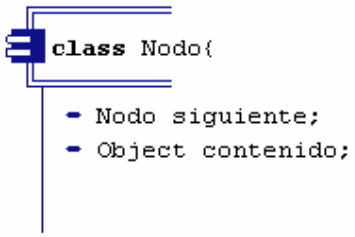

Fig. 2.1.1 Clase Nodo

Los nodos serán representados por los siguientes símbolos:

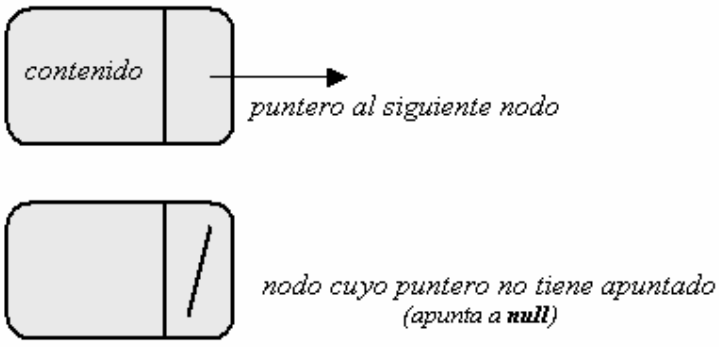

Fig. 2.1.2 Simbología para el nodo

La parte frontal de la lista es representada por un puntero al primer nodo. Es decir, un atributo de la lista enlazada es un nodo ***primero***.

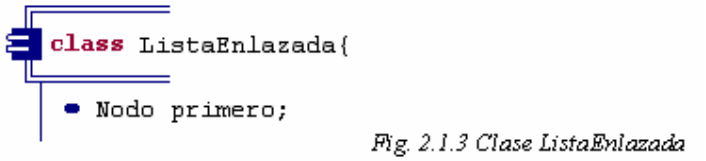

Fig. 2.1.3 Clase ListaEnlazada

Y podríamos representar a una lista enlazada que almacena **int**'s por medio del siguiente diagrama:

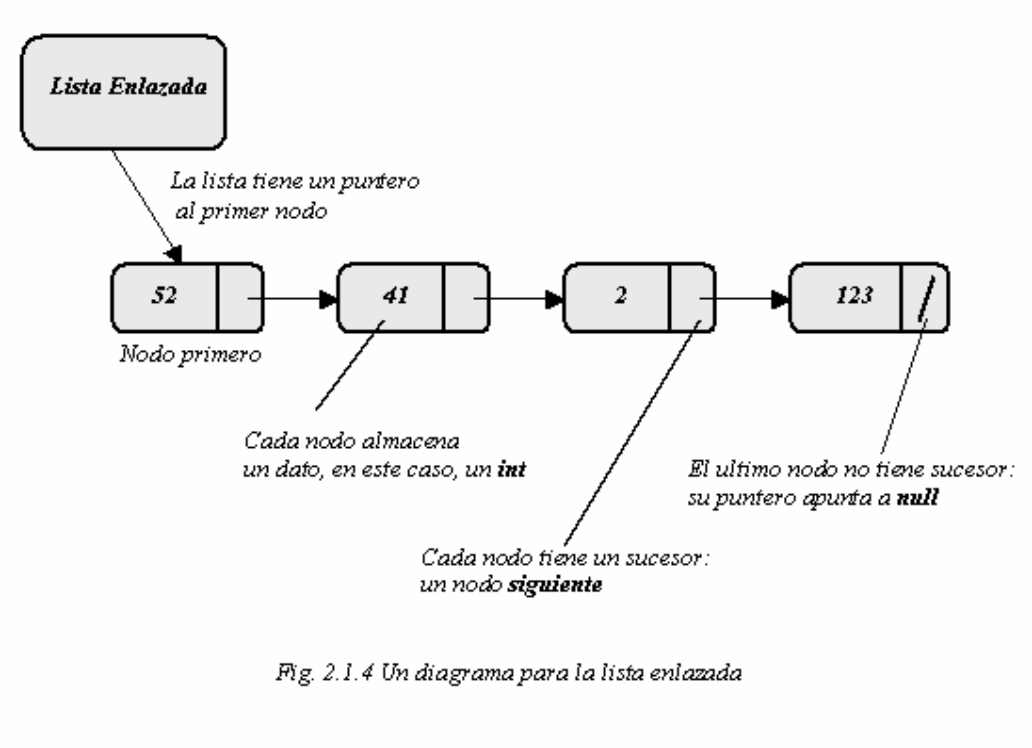

Fig. 2.1.4 Un diagrama para la lista enlazada

La lista también tendrá un atributo ***largo***, del tipo **long** (o **int**, si se quiere)**,** que representará la cantidad de elementos en ella.

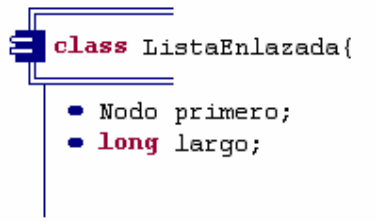

Fig. 2.1.5 El atributo "largo"

2.2 Cómo acceder a un elemento de la lista

Para tener acceso a un elemento de la lista, hay que hacer uso de los punteros. Se crea un puntero al primer nodo de la lista y se avanza por medio del atributo "siguiente". Por ejemplo, imaginemos que tenemos una lista enlazada llamada *lista*, que tiene almacenado nombres de personas por medio de String's. Si quisiéramos tener acceso al cuarto nombre almacenado, tendríamos que:

1- Crear un puntero al primero de la lista
2- Avanzar con el puntero hasta el cuarto nodo
3- Utilizar el contenido del nodo

El código sería algo como esto:

```
// creamos un puntero al primer nodo de la lista
Nodo puntero=lista.primero;

// lo hacemos avanzar hasta el 4 nodo
int i=1;
while(i<4){
    puntero=puntero.siguiente;
    i++;
}

// usamos el contenido del nodo para algo útil
System.out.println("El 4to nombre almacenado es "+puntero.contenido);
```

Fig. 2.2.1 Acceso a los elementos de la lista

Ahora, veamos el diagrama:

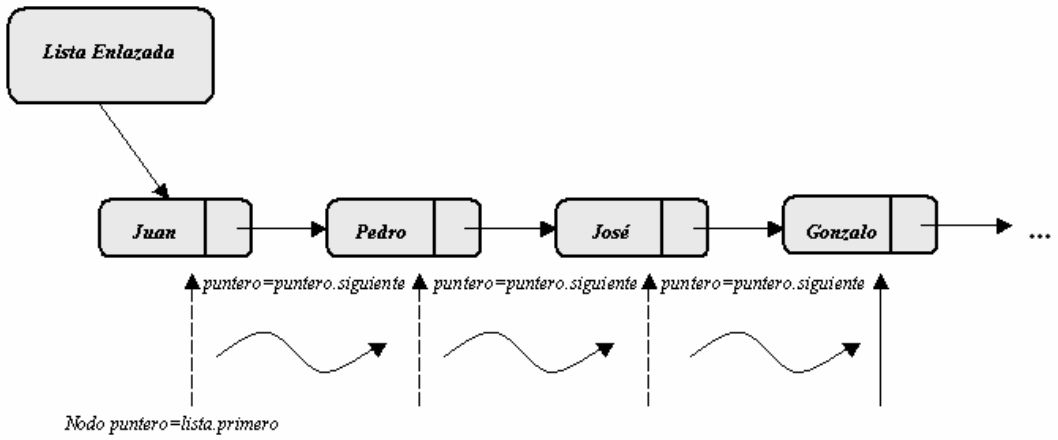

Fig. 2.2.2 Acceso a los elementos de la lista

Usando este concepto, definiremos todos los métodos de la lista enlazada.

2.3 Diseño de la clase ListaEnlazada

Para mostrar cómo definir los métodos de la lista enlazada, crearemos una clase de ejemplo. Ahora que conocemos la idea detrás de la definición de los atributos de la lista enlazada y del nodo, crearemos la clase de ejemplo. El código del ejemplo está en la carpeta **"Lista Enlazada Simple".**

El nombre de nuestra clase será **ListaEnlazada**. Esta lista almacenará sólo un tipo de datos, que serán objetos de la clase **Cliente**, definida por nosotros mismos. Para que ésto sea posible, es necesario que los Clientes tengan un atributo único para cada uno, de tal forma que podamos diferenciarlos. Este atributo será el RUT.

Así definiremos la clase Cliente:

```
public class
Cliente{

- String rut;
- String nombre;
- String apellido;
- boolean esClienteFrecuente;

public
Cliente(String rut, String nombre, String apellido, boolean esClienteFrecuente){

    this.rut=rut;
    this.nombre=nombre;
    this.apellido=apellido;
    this.esClienteFrecuente=esClienteFrecuente;
}

}
```

Fig. 2.3.1 La clase Cliente

Nota: Para poder continuar, vea detalles sobre los atributos y el método constructor en los comentarios del archivo Cliente.java

Ahora tenemos todo para empezar a construir nuestra clase ListaEnlazada. En esta implementación de lista enlazada, definiremos los siguientes métodos:

NOMBRE DEL METODO	VALOR DE RETORNO	TIPOS DE ARGUMENTO	UTILIDAD
estaVacia()	boolean	Ninguno	Retorna true si la lista está vacía, sino, false
vaciar()	void	Ninguno	Remueve todo el contenido de la lista
largo()	long	Ninguno	Retorna el largo (numero de elementos) de la lista
tieneElRut(rut)	boolean	String	Retorna true si la lista tiene el rut especificado, sino, false
tengaMasElementos(puntero)	boolean	Nodo	Retorna true si puntero tiene sucesor, sino, false
añadir(nuevoCliente)	void	String	Añade a nuevoCliente al final de la lista
buscar(rut)	Cliente	String	Retorna el Cliente con el rut especificado
remover(rut)	void	String	Remueve al Cliente con el rut especificado. Devuelve true si elimina a alguno, sino, false

Ahora revisaremos como funcionan, en términos de punteros, los métodos principales: **añadir()**, **buscar()** y **remover()**. Para detalles sobre el funcionamiento de los demás métodos, variables, etc., ver comentarios del archivo **ListaEnlazada.java**.

2.3.1 Método añadir()

Para añadir un nodo a la lista, primero creamos un nodo, al que llamaremos *nuevoNodo*, que contenga al Cliente entregado por el argumento, al que llamaremos *nuevoCliente.*

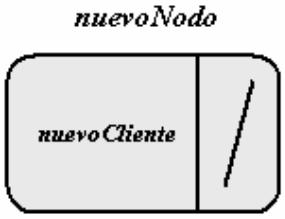

Fig. 2.3.1.1 nuevoNodo

Luego conectamos a *nuevoNodo* con el último nodo de la lista. Para referirnos a éste, creamos un puntero, al que llamaremos *puntero*, al primer nodo y avanzamos hasta el último por medio de la instrucción *puntero = puntero.siguiente* como fue explicado anteriormente.

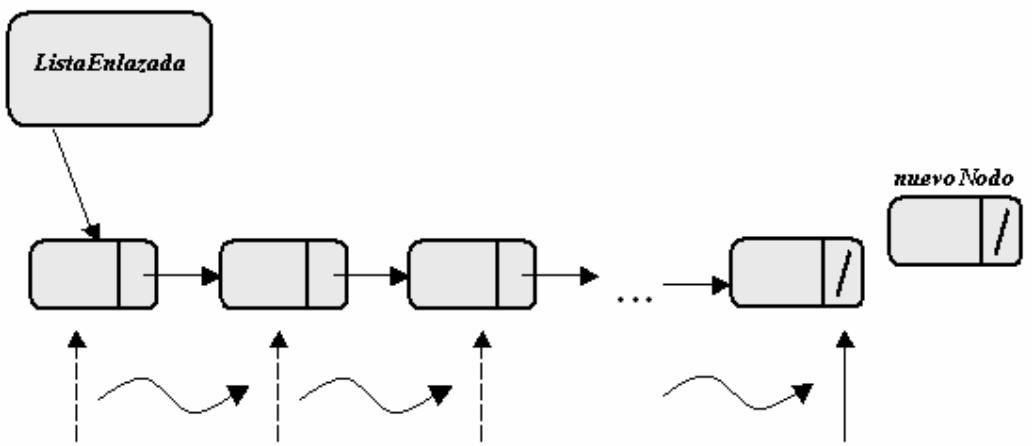

Fig. 2.3.1.2 Se avanza hasta el final

Ahora que *puntero* apunta al último nodo de la lista, y, además, ahora *puntero.siguiente* tiene un valor igual a **null,** haremos que *puntero.siguiente* tome el valor *nuevoNodo,* es decir*, puntero.siguiente = nuevoNodo* . Gráficamente:

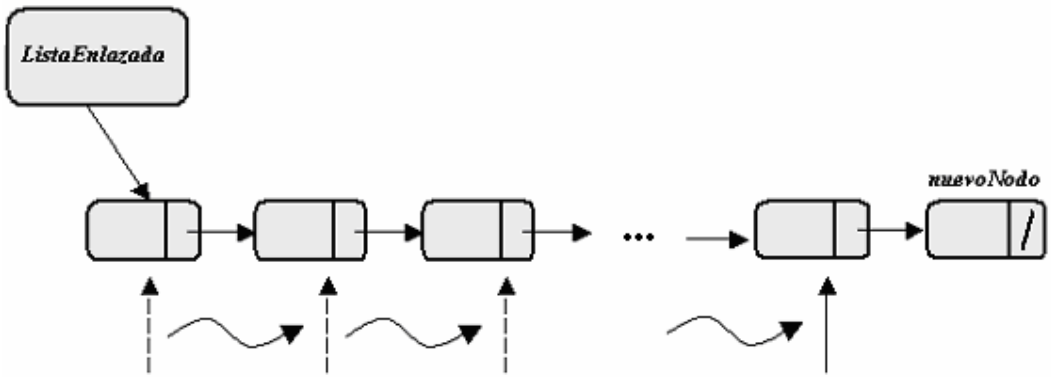

Fig. 2.3.1.3 nuevoNodo ingresa a la lista

Para ver detalles del código, ver el archivo ListaEnlazada.java

2.3.2 Método buscar()

Para encontrar un Cliente en la lista, debemos buscarlo por medio de su atributo único: el RUT. Es decir, crearemos un puntero *puntero* con el que recorreremos la lista, comparando el RUT especificado con el RUT de cada Client, hasta encontrar una coincidencia. Para ésto, tendremos que usar una instrucción como ésta:

```
while(puntero!=null){ // mientras el puntero no apunte al vacio

    if (puntero.contenido.rut.compareTo(rut)==0) // si los RUTs son iguales...
        return puntero.contenido; // ...entonces retorna el contenido del nodo apuntado

    puntero=puntero.siguiente; // en otro caso, se avanza al siguiente hasta encontrar
                               // una coincidencia
}
```

Fig.2.3.3.1 Instrucción de Búsqueda

Gráficamente:

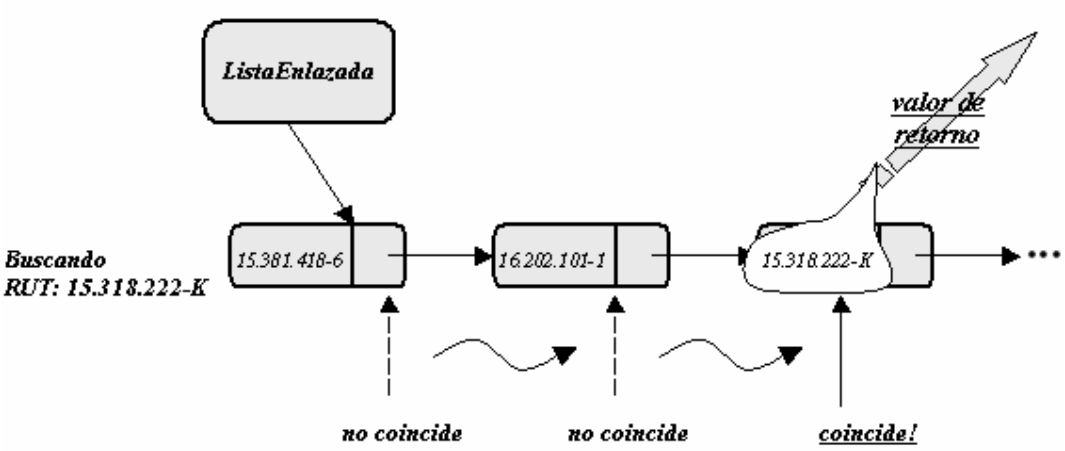

Fig. 2.3.3.2 Buscando un RUT

Para ver detalles del código, ver el archivo ListaEnlazada.java.

2.3.3 Método remover()

Para eliminar un nodo de la lista, primero debemos ubicar el nodo anterior éste, es decir, *puntero.siguiente* será el nodo a eliminar. Ahora, simplemente dejamos sin puntero al nodo diciendo: *puntero.siguiente = puntero.siguiente.siguiente,* para que el Recolector de Basura de Java lo recoja y sea eliminado. Gráficamente:

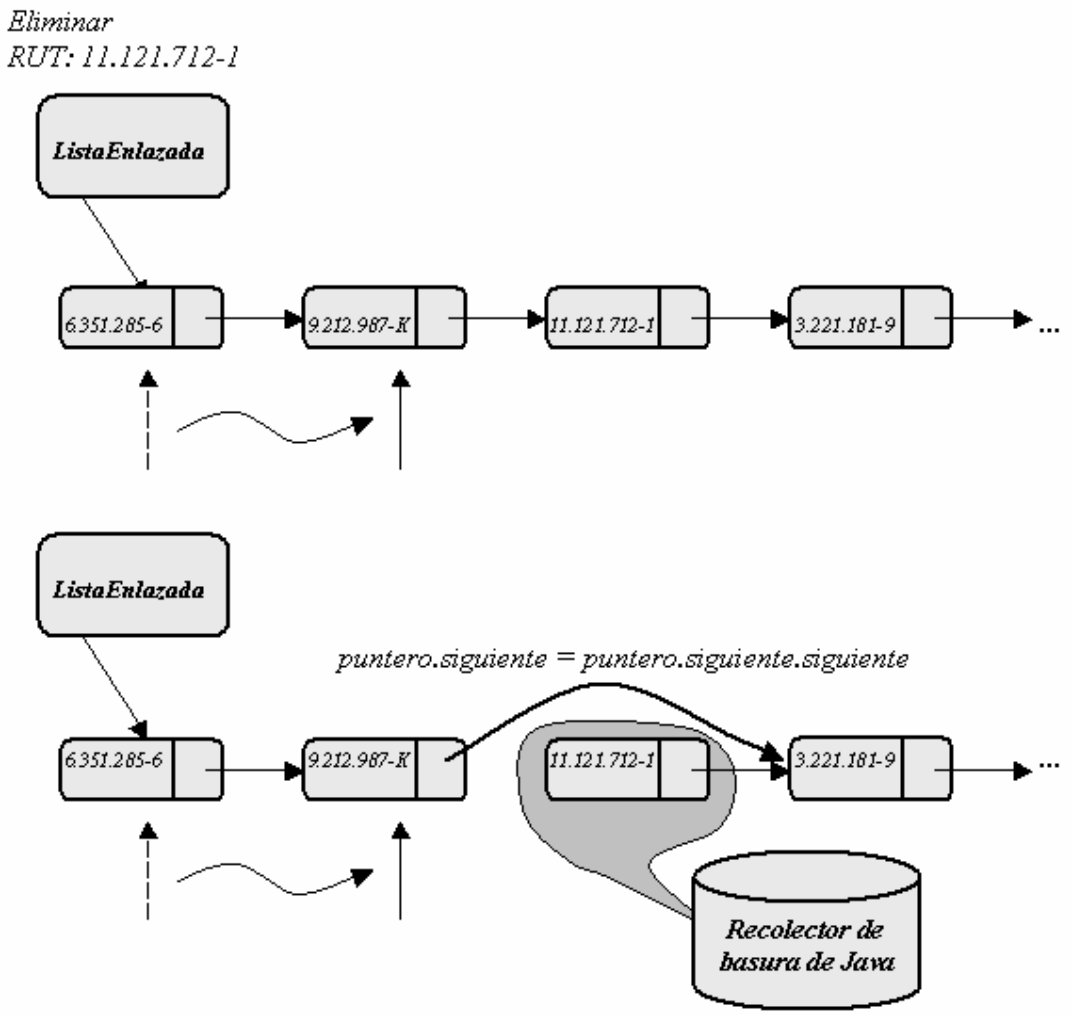

Fig.2.3.3.1 Instrucción de Búsqueda

Para ver detalles del código, ver el archivo ListaEnlazada.java.
El resto de los métodos creados en esta implementación están explicados claramente en los comentarios de el archivo ListaEnlazada.java.

Estos tres métodos son los mas básicos que podemos crear a partir de la *"lógica del puntero"*, y la implementación de otros métodos basados en la misma idea queda a gusto (o necesidad) del programador. Por ejemplo, podríamos crear métodos que filtren la lista según una característica de los clientes (como discriminar entre los clientes frecuentes y los no frecuentes), o un método que traspase los datos de los clientes a un archivo de texto para guardarlos, y otro que sea capaz de leer este archivo y llenar una lista con los datos guardados. En fin, hay infinitas posibilidades.

2.3.4 Desafíos para el alumno

Ahora que conoces la forma fundamental de la estructura de la lista enlazada, trata de implementar lo siguiente:

- Un método filtrador que reciba como argumento un valor booleano y que retorne una lista filtrada. La idea es que, cuando el argumento sea **true**, devuelva una lista sólo con los clientes frecuentes de la lista original, es decir, clientes que tengan su atributo *esClienteFrecuente* con valor **true,** y que cuando sea **false,** devuelva una lista con sólo los clientes no frecuentes.

- Un método filtrador que reciba como argumento un **char** y que retorne una lista con todos los clientes que tengan apellido que empiece con esa letra, es decir, que la primera letra de *apellido* sea la especificada en el argumento.

- Una lista enlazada que guarde objetos de cualquier tipo. Entonces, para poder diferenciarlos, hacer que cada nodo tenga un índice (algo parecido a un arreglo), y crear un método para ésta lista que retorne el objeto n-ésimo de la lista, dado el valor de n.

A continuación revisaremos otras dos estructuras de datos basadas en el "juego de punteros": la Cola y la Pila (Queue y Stack respectivamente).

3. Pila

3.1 Qué es una pila

La Pila o Stack es una estructura de datos muy simple. Imagina que tienes una pistola de juguete que dispara pelotas, y, para cargarla, tienes que introducir las pelotas una a una por la parte frontal del cañón de la pistola, una tras otra. La primera pelota que dispararás será la última que introdujiste, y la última que dispararás será la primera que introdujiste. Eso es una pila.

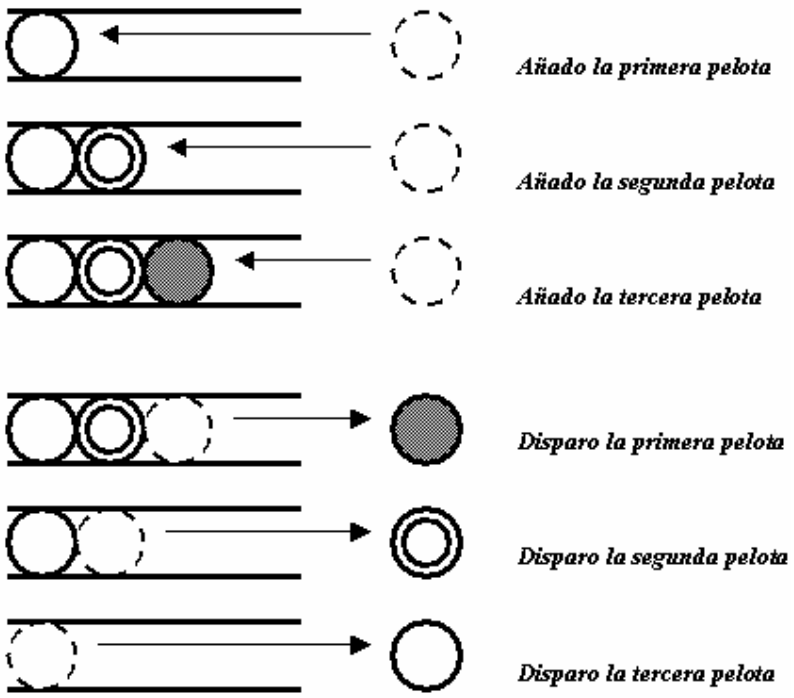

Fig. 3.1.1 Un Stack: la pistola de juguete

Otro ejemplo muy común es la pila de platos que formas para lavarlos: el último plato que apilaste será el primero que lavarás.

La pila es una estructura que se basa en el concepto **LIFO: Last In First Out,** es decir, el último elemento que entra es el primero que sale.

La implementación de una pila es muy parecida a la de lista enlazada, y sólo difiere en la forma que gestionamos los elementos almacenados. En una pila, crearemos métodos que cumplan las funciones expuestas anteriormente y aclaradas con la *Fig. 3.1.1*, es decir, un método que agregue un nodo al principio de la pila y otro que elimine el primer nodo de la pila. Para esta implementación, los nodos serán instancias de la clase Nodo, definida de la misma forma que la definimos para la lista enlazada. Para poder continuar, es necesario que revises detalladamente la definición de esta clase. La explicación de su funcionamiento fue agregada en el mismo código en forma de comentario.

3.2 Diseño de la clase Pila

Seguiremos la misma idea de lista enlazada, pero ahora, al primer nodo de la pila lo llamaremos *primeroDeLaPila*.

En esta implementación, definiremos los siguientes métodos:

NOMBRE DEL METODO	VALOR DE RETORNO	TIPOS DE ARGUMENTO	UTILIDAD
estaVacia()	boolean	Ninguno	Retorna true si la pila está vacía, sino, false
vaciar()	void	Ninguno	Remueve todo el contenido de la pila
largo()	long	Ninguno	Retorna el largo (numero de elementos) de la pila
verPrimero()	Object	Ninguno	Retorna el valor del contenido de *primeroDeLaPila*
sacar()	void	Ninguno	Elimina al primer nodo de la pila
sacarYverPrimero ()	Object	Ninguno	Elimina al primer nodo de la pila, pero lo retorna
apilar(nuevoObjeto)	void	Object	Añade un nodo que contiene a *nuevoObjeto* a la pila, "empujando" al resto de los nodos

Ahora revisaremos como funcionan, en términos de punteros, los métodos principales: **apilar()** y **sacar()**. Para detalles sobre el funcionamiento de los demás métodos, variables, etc., ver comentarios del archivo **Pila.java**.

3.2.1 Método verPrimero()

La idea del método verPrimero() es, simplemente, decirnos quién es el objeto que está al principio de la pila, es decir, el último objeto ingresado y, potencialmente, el primero a eliminar. Para tener acceso a este objeto, hacemos uso del atributo *primeroDeLaPila,* que no es más que un puntero al primer nodo de la pila. Así que este método sólo retorna el contenido de *primeroDeLaPila*, es decir, *primeroDeLaPila.contenido.*

Para ver detalles del código, ver el archivo Pila.java.

3.2.2 Método sacar()

El objetivo de este método es eliminar el nodo más externo, es decir, el último ingresado. Para esto, simplemente hacemos que *primeroDeLaPila* apunte a su sucesor, es decir *primeroDeLaPila = primeroDeLaPila.siguientet*. Gráficamente:

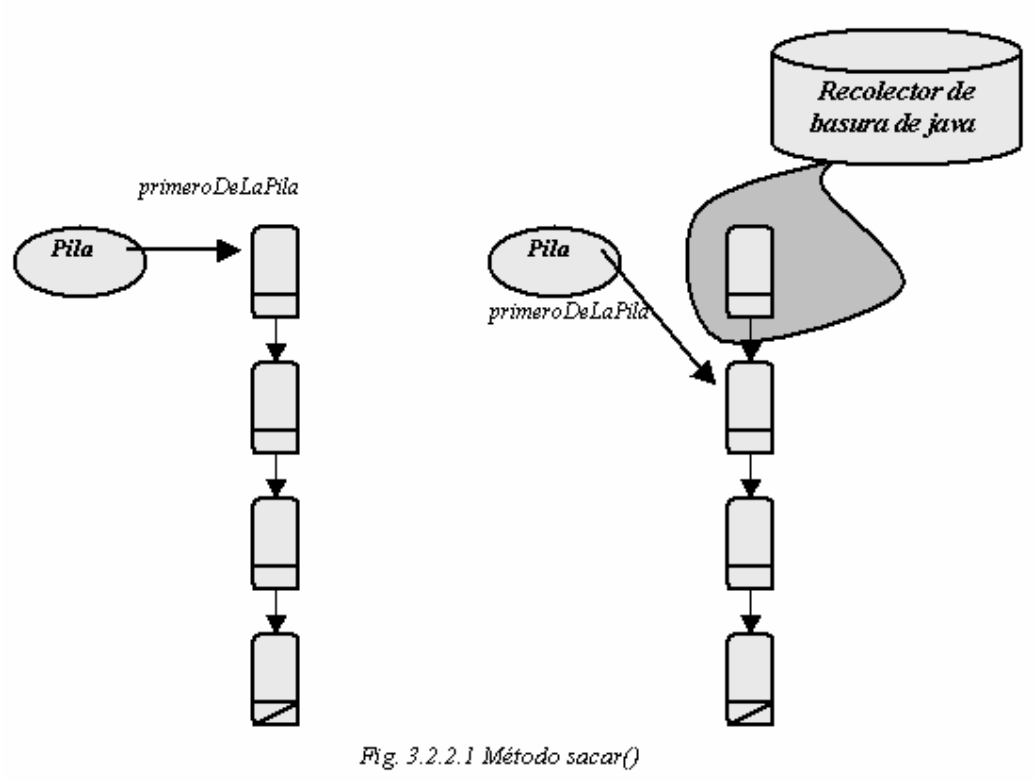

Fig. 3.2.2.1 Método sacar()

3.2.3 Método apilar()

Este método es el encargado de añadir nodos a la pila. Para esto, es necesario que el nodo sea añadido al principio de la pila. Gráficamente:

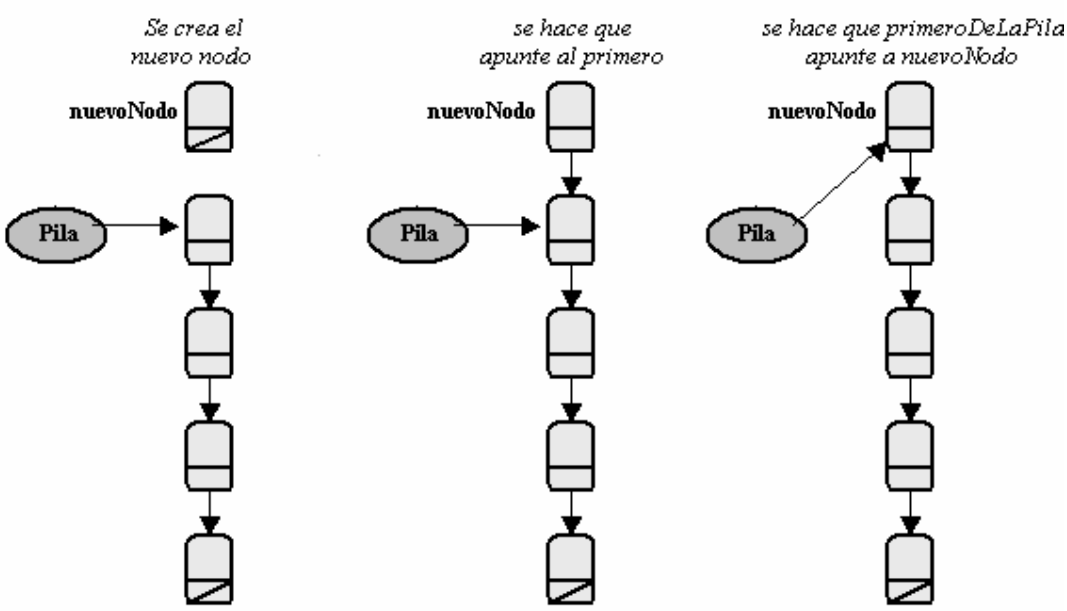

Fig. 3.2.3.1 Método apilar()

Otra forma de visualizar a **apilar()** y **sacar()**:

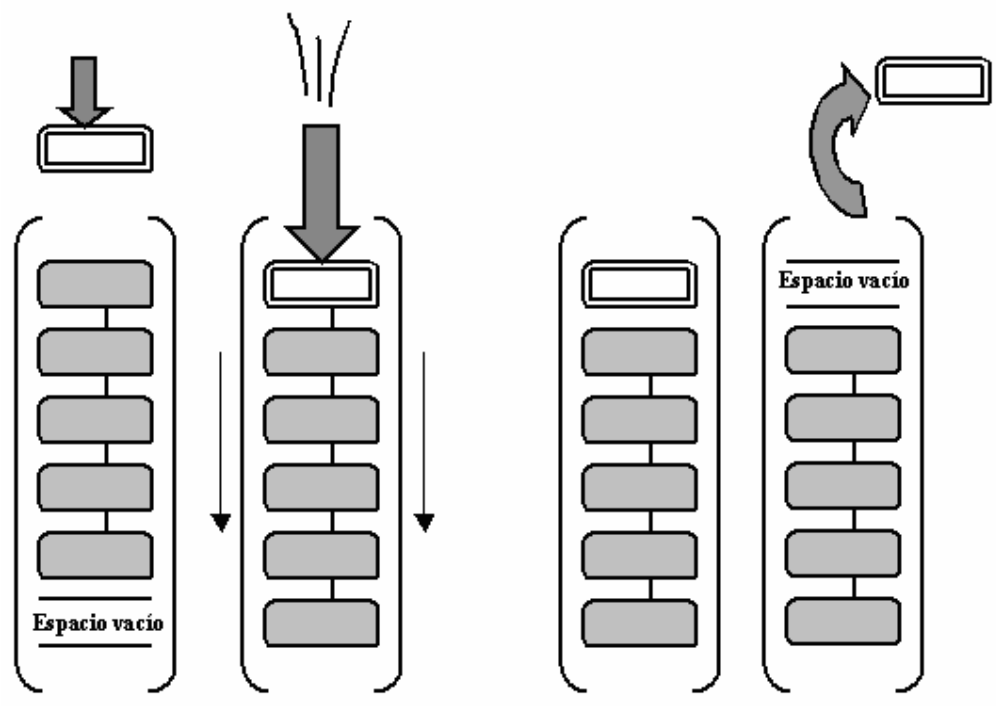

Fig. 3.2.3.1 apilar() y sacar()

Para ver detalles del código, ver el archivo Pila.java.

3.2.4 Desafíos para el alumno

En esta ocasión, propongo un desafío muy simple:

- Basándose en la implementación de Pila anteriormente mostrada, crear una pila que tenga una capacidad máxima de elementos, es decir, que pueda "estar lleno".

Ahora, abordemos la siguiente estructura de datos a estudiar: la Cola.

4. Cola

4.1 Qué es una cola

La Cola o Queue es otra de las estructuras de datos que revisaremos en este apunte. Como las pilas, las colas son muy simples de entender e implementar. La idea de la cola es simular el funcionamiento de una cola de la vida real, por ejemplo, la cola de un banco. Imagina que tienes una pistola de juguete parecida a la del ejemplo de pila, pero ahora, para cargarla, debes introducir las pelotas por la parte trasera del cañón. Eso es una Cola.

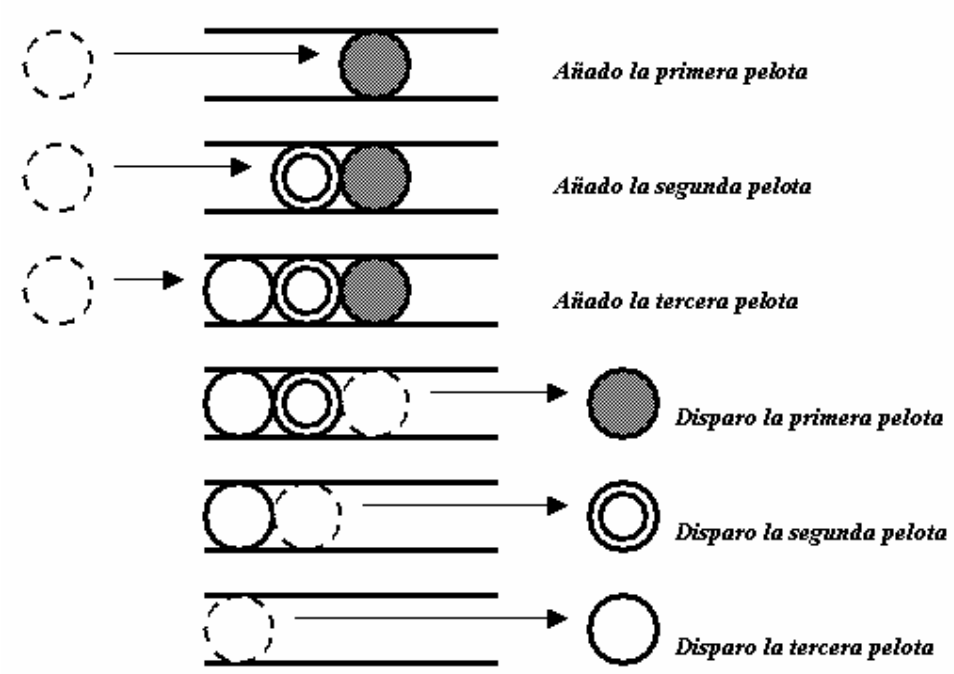

Fig. 4.1.1 Un Queue: la pistola de juguete

La cola es una estructura que se basa en el concepto **FIFO: First In First Out,** es decir, el primer elemento que entra es el primero que sale.

Tal como las pilas, la implementación de la pila es muy parecida a la de lista enlazada, y sólo difiere en la forma que gestionamos los elementos almacenados. En una cola, crearemos métodos que cumplan las funciones expuestas anteriormente y aclaradas con la *Fig. 4.1.1*, es decir, un método que agregue un nodo al principio de la cola y otro que elimine el último nodo de ésta. Para esta implementación, los nodos serán instancias de la clase **Nodo,** definida anteriormente en el archivo **Nodo.java**. Para poder continuar, es necesario que revises detalladamente la definición de esta clase. La explicación de su funcionamiento fue agregada en el mismo código en forma de comentario.

4.2 Diseño de la clase Cola

Seguiremos la misma idea de lista enlazada, pero ahora, al primer nodo de la cola lo llamaremos *primeroEnLaCola* y al último lo llamaremos *ultimoEnLaCola*.

En esta implementación, definiremos los siguientes métodos:

NOMBRE DEL METODO	VALOR DE RETORNO	TIPOS DE ARGUMENTO	UTILIDAD
estaVacia()	boolean	Ninguno	Retorna true si la cola está vacía, sino, false
vaciar()	void	Ninguno	Remueve todo el contenido de la cola
largo()	long	Ninguno	Retorna el largo (numero de elementos) de la cola
verPrimero()	Object	Ninguno	Retorna el valor del contenido de *primeroEnLaCola*
verUltimo()	Object	Ninguno	Retorna el valor del contenido de *ultimoEnLaCola*
enfilar(*nuevoObjeto*)	void	Object	Añade un nodo que contiene a *nuevoObjeto* al final de la cola
sacar()	void	Ninguno	Saca el primer nodo de la cola y retorna su contenido

Ahora revisaremos como funcionan, en términos de punteros, los métodos principales: **enfilar()** y **sacar()**. Para detalles sobre el funcionamiento de los demás métodos, variables, etc., ver comentarios del archivo **Cola.java**.

4.2.1 Método enfilar()

Este método tiene como objetivo poner en la fila al objeto entregado en el argumento, al que llamaremos *nuevoObjeto*. Para hacer esto, se debe agregar a la cola un nodo, al que llamaremos *nuevoNodo*, que contenga a *nuevoObjeto*. Es decir *ultimoEnLaCola.siguiente* debe apuntar a *nuevoNodo***,** y luego *ultimoEnLaCola* debe apuntar a *nuevoNodo*. Veámoslo gráficamente:

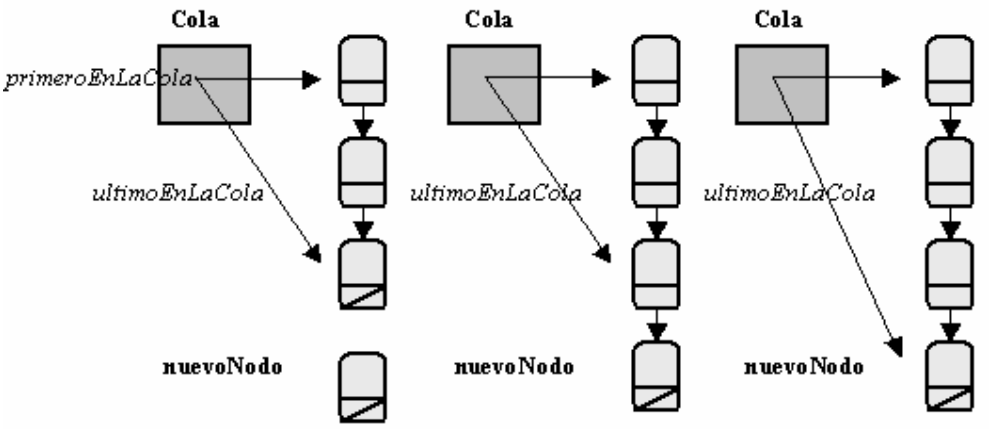

Fig.4.2.1.1 Método enfilar()

Para ver detalles del código, ver el archivo Cola.java

4.2.2 Método sacar()

Este método es el encargado de "atender" al primero de la fila, es decir, de eliminarlo de la cola y retornarlo. Para ésto, simplemente basta hacer que *primeroEnLaCola* apunte a su sucesor, o sea, *primeroEnLaCola = primeroEnLaCola.siguiente.* Gráficamente:

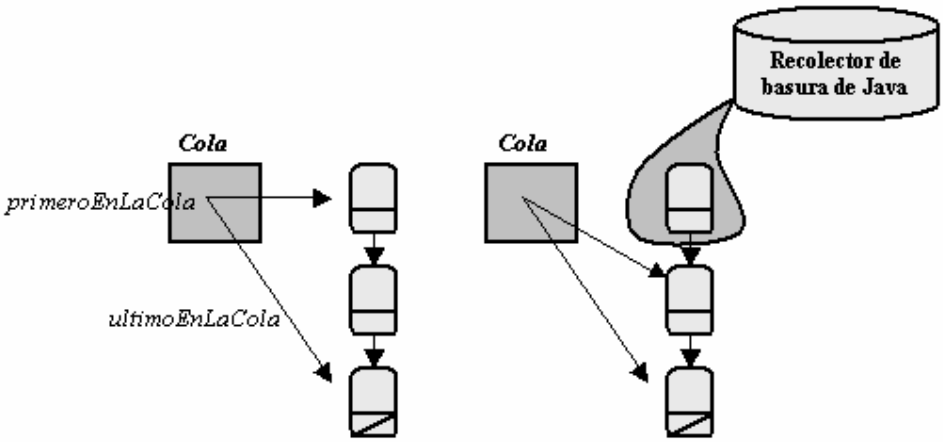

Fig. 4.2.2.1 Método sacar()

Para ver detalles del código, ver el archivo Cola.java.

Otra forma de visualizar a **enfilar()** y **sacar()**:

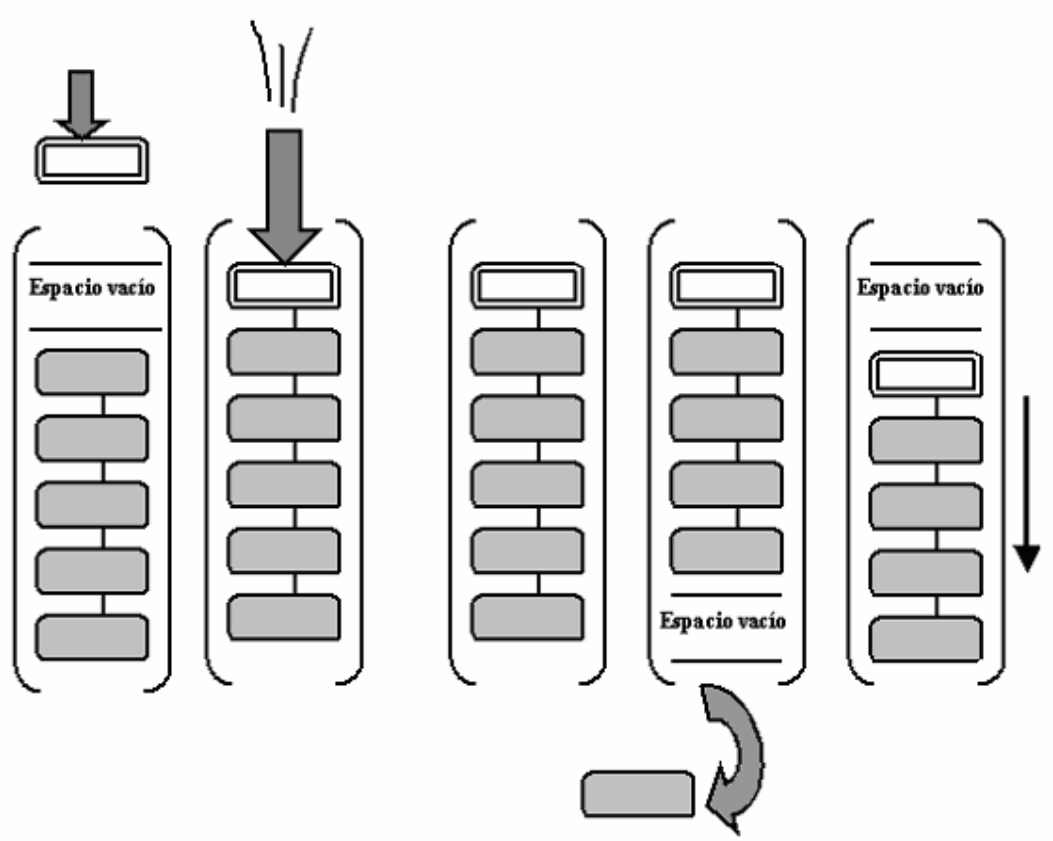

Fig. 4.2.2.2 enfilar() y sacar()

4.2.3 Desafíos para el alumno

En esta ocasión también propondré un muy simple desafío:

- Basándose en la implementación de Cola anteriormente mostrada, crear una cola que su método sacar() también reciba como a argumento a un **int** *n,* y que, en vez de eliminar un sólo nodo, elimine *n* nodos.

Finalmente, revisaremos la última estructura de datos abstracta de este apunte: Los Árboles Binarios.

5. Árbol Binario

5.1 Conceptos básicos

Los árboles son una de las estructuras de datos no lineales más utilizada. Sirve para representar estructuras de información jerárquicas y direcciones o etiquetas de una manera organizada.

Dentro de la ciencia de la computación, los árboles tienen muchas aplicaciones, como, por ejemplo:

- organizar tablas de símbolos en compiladores
- representar tablas de decisión
- asignar bloques de memoria de tamaño variable
- ordenar
- buscar
- solucionar juegos
- probar teoremas

Como objetos matemáticos los árboles también han sido estudiados ampliamente.

Los árboles permiten representar situaciones de la vida diaria como son:

- organización de una empresa
- árbol genealógico de una persona
- organización de torneos deportivos

Para dar una definición clara de árbol, podríamos decir que un árbol es un conjunto finito T de uno o más nodos tal que:

a) Existe un nodo especial llamado la raíz del árbol
b) Los nodos restantes están particionados en $m \geq 0$ conjuntos disjuntos $T_1,...,T_m$ y cada uno de estos conjuntos es a su vez un árbol. Los árboles $T_1,...,T_m$ son llamados subárboles de la raíz

Cualquier nodo es la raíz de un subárbol que consiste de él y los nodos debajo de él. Esto se deriva de la definición recursiva de árbol presentada.

Un árbol puede representarse gráficamente de muchas formas. Algunas de ellas son por medio de:

- conjuntos anidados:

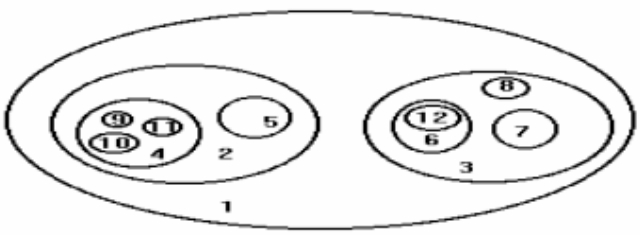

Fig. 5.1.1 Conjuntos anidados

- paréntesis anidados:

(1(2(4(9,10,11),5),3(6(12),7,8)))

Fig. 5.1.2Paréntesis anidados

- indentación:

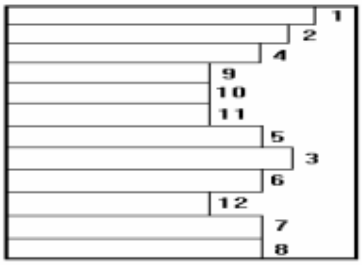

Fig. 5.1.3 Identación

- grafos:

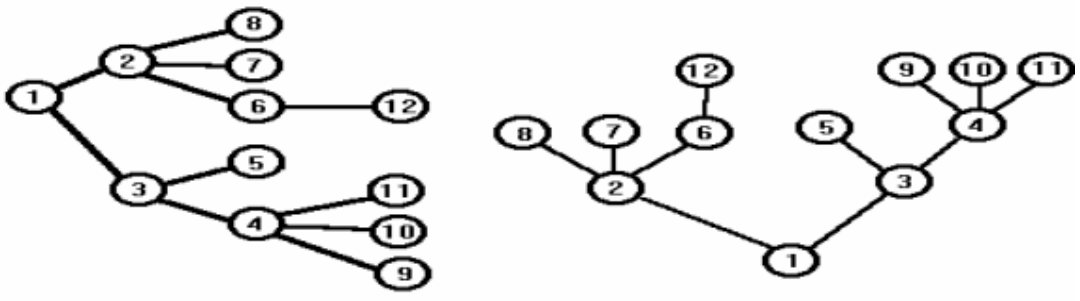

Fig. 5.1.4 Grafos

La forma más común de representar a los árboles es por medio de un grafo con la raíz hacia arriba:

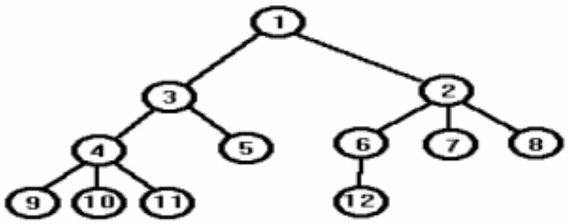

Fig. 5.1.5 Forma más común de representación: Grafo hacia arriba

Cada vértice o nodo del árbol puede tener un nombre y puede tener una información asociada; un arco es una conexión entre dos vértices.

Un camino en un árbol es una lista de vértices diferentes en los que vértices sucesivos están conectados por arcos en el árbol. Una propiedad que define a los árboles es que existe exactamente un camino entre la raíz y cada uno de los otros nodos en el árbol.

La longitud de un camino es el número de nodos del camino menos uno. Por tanto, hay un camino de longitud cero de cualquier nodo a si mismo.

Se dice que un nodo Y está abajo de un nodo X, si X está en el camino de Y a la raíz. Además, cada nodo, excepto la raíz, tiene exactamente un nodo arriba de él, que es llamado su padre; los nodos que están exactamente abajo de él son llamados sus hijos.

El número de hijos que cuelgan de un nodo es llamado el grado del nodo. El grado de un árbol es el grado máximo de los nodos del árbol. Un nodo de grado cero es llamado hoja, es decir, no tiene hijos.

Por ejemplo, considera el siguiente árbol:

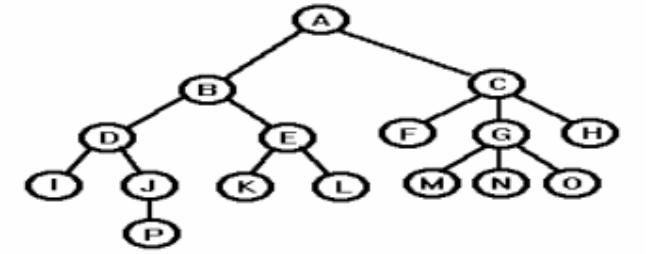

Fig. 5.1.6 Árbol de ejemplo

El camino del nodo *A* al nodo *P* es (*A, B, D, J, P*), cuya longitud de camino es 4. *E* es el padre de *K* y *L*.

Los hijos de *G* son *M, N* y *O*.

Los nodos de grado 0 son: *I, P, K, L, F, M, N, O,* y *H* es decir son las hojas del árbol.
El único nodo de grado 1 es *J*.

Los nodos de grado 2 son *A, B, D* y *E*.

Los nodos de grado 3 son *C* y *G*.

No existen nodos de mayor grado, por tanto, el grado del árbol es 3.

Los nodos de un árbol pueden dividirse en niveles; el nivel de un nodo es el número de nodos en el camino de él a la raíz. La altura de un árbol es el nivel máximo de todos los nodos en el árbol, es decir, la distancia máxima de la raíz a cualquier nodo.

Por ejemplo, a continuación se muestra el nivel de cada nodo del árbol de la figura 5.1.7:

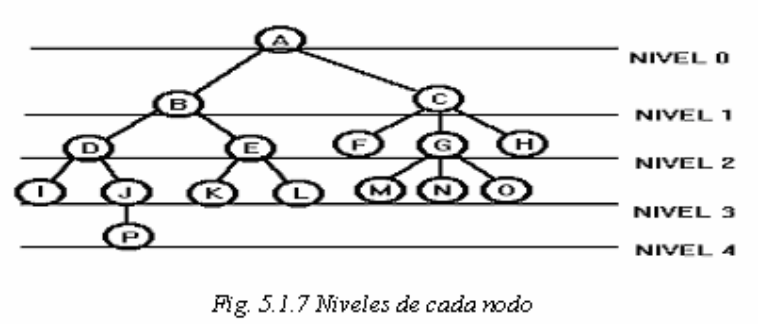

Fig. 5.1.7 Niveles de cada nodo

De lo anterior, se ve que el árbol es de altura 4.

En muchas ocasiones los árboles se estudian por su grado. En este caso, estudiaremos los árboles de grado 2, es decir, los **árboles binarios**.

5.2 Qué es un Árbol Binario

Los árboles binarios son el caso particular más simple. Son usados para representar ciertos tipos de expresiones algebraicas, algoritmos, búsquedas y ordenamientos. Para definirlos formalmente, podríamos decir que un conjunto T de elementos (nodos) es un árbol binario si:

1. T es vacío, o
2. T está particionado en 3 conjuntos disjuntos

    a) un solo elemento R, llamado la raíz
    b) 2 conjuntos que son árboles binarios, llamados los subárboles izquierdo y derecho de R

De esta forma, T es un árbol binario si:

1. T no tiene nodos, o
2. T es de la forma:

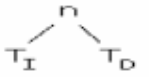

Fig. 5.2.1 Forma de árbol binario

Donde n es un nodo y T_I y T_D son árboles binarios.

5.2.1 Recorrido en árboles binarios

Una vez construido un árbol binario, surge la necesidad de recorrerlo, es decir, una manera sistemática de visitar cada nodo del árbol. La forma en la cual una lista lineal se recorre es trivial (del primero al último, o viceversa). Sin embargo, para recorrer un árbol, esta forma natural no puede aplicarse.

Para recorrer un árbol, existen varias formas de lograrlo, las 3 más comunes son **pre-orden**, **in-orden** y **post-orden**. Estos métodos difieren en el orden en el cual los nodos son visitados. Siguiendo la costumbre de empezar a visitar antes lo que se encuentra a la izquierda que lo de la derecha, se tienen 3 posibilidades de visitar a la raíz: antes, en medio o después.

Para recorrer un árbol binario no vacío T en **pre-orden** o **previo**:

1. visitar la raíz
2. recorrer recursivamente el subárbol izquierdo
3. recorrer recursivamente el subárbol derecho

Para recorrer un árbol binario no vacío T en **in-orden** o **simétrico**:

1. recorrer recursivamente el subárbol izquierdo
2. visitar la raíz
3. recorrer recursivamente el subárbol derecho

Para recorrer un árbol binario no vacío T en **post-orden** o **posterior**:

1. recorrer recursivamente el subárbol izquierdo
2. recorrer recursivamente el subárbol derecho
3. visitar la raíz

Por ejemplo, considere el siguiente árbol:

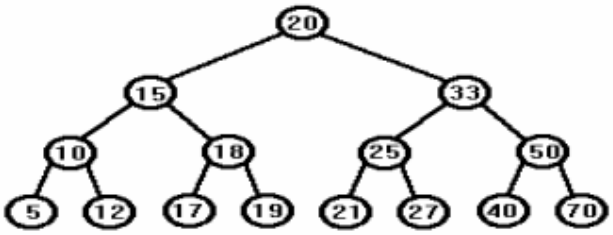

Fig. 5.2.1.1 Recorrido del árbol binario

el resultado que se obtiene al recorrer el árbol en

pre-orden: 20, 15 10, 5, 12, 18, 17, 19, 33, 25, 21, 27, 50, 40, 70

in-orden: 5, 10, 12, 15, 17, 18, 19, 20, 21, 25, 27, 33, 40, 50, 70

post-orden: 5, 12, 10, 17, 19, 18, 15, 21, 27, 25, 40, 70, 50, 33, 20

5.2.2 Árboles binarios de búsqueda

Dado un conjunto de nodos (con un campo asignado como llave de búsqueda) T es un árbol binario de búsqueda si:

1. T es vacío, o
2. T es de la forma

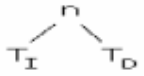

Fig. 5.2.2.1 Forma de árbol binario

y

a) la llave de búsqueda de n es mayor que todas las llaves de búsqueda de T_I
b) la llave de búsqueda de n es menor que todas las llaves de búsqueda de T_D
c) T_I y T_D son árboles binarios de búsqueda

Buscar:

Una ventaja de los árboles binarios de búsqueda es que el algoritmo de búsqueda surge de una manera natural. Si *x* es la llave de la raíz de un árbol binario de búsqueda y se está buscando a *x*, se ha terminado la búsqueda; esto es cierto suponiendo que las llaves de búsqueda son únicas. Si se busca una llave menor que *x*, entonces debe encontrarse en el subárbol izquierdo. Similarmente, si se busca una llave mayor que *x*, debe buscarse en el subárbol derecho.

Así, puede procederse recursivamente hasta encontrar la llave buscada. Por ejemplo, considere el siguiente árbol binario de búsqueda:

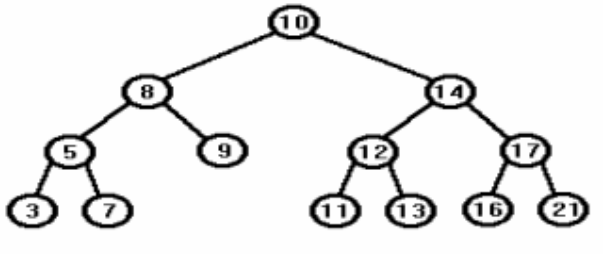

Fig. 5.2.2.2 Búsqueda en un árbol binario

Si se busca la llave 7 en el árbol se compara 7 con 10 y se desciende por la izquierda, porque $7 < 10$; se compara 7 con 8 y se desciende por la izquierda, porque $7 < 8$; al comparar 7 con 5 se desciende por la derecha, porque $7 > 5$, y se encuentra la llave.

Insertar:

La inserción en un árbol binario de búsqueda es similar a la búsqueda. Suponga que se desea insertar una nueva llave con valor *x*, lo primero que debe hacerse es una búsqueda, si *x* se encuentra en el árbol no se requiere insertarla de nuevo (si se desea que las llaves del árbol sean únicas); en otro caso la búsqueda termina no exitosamente en una hoja o en un nodo con un solo subárbol; y así un nuevo nodo que contenga la llave *x* debe ser insertado debajo de esa hoja, como hijo derecho o hijo izquierdo, dependiendo del valor de *x*.

Por ejemplo, mirando la figura 5.2.2.2, si se desea insertar la llave 6, se compara 6 con 10 y se desciende por la izquierda; se compara 6 con 8 y se desciende por la izquierda; se compara 6 con 5 y se desciende por la derecha; se compara 6 con 7 y se inserta la llave 6 como hijo izquierdo del nodo con llave 7.

5.3 Diseño de la clase ArbolBinario

Ahora crearemos nuestro primer árbol binario. Para este efecto, la complejidad de éste podría ser muy grande, así que en esta ocasión haremos un árbol relativamente simplificado. Cada árbol tendrá cuatro atributos: un número, que representará la llave del árbol, al cual llamaremos *raíz*; un String, que representará el contenido, al cual llamaremos *contenido*; y dos ArbolBinario, *hijoDerecho* e *hijoIzquierdo*, que representarán a los hijos del árbol.

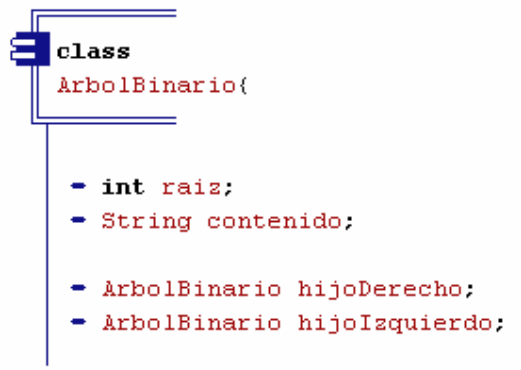

Fig. 5.3.1 Atributos

El método constructor recibirá dos argumentos: la llave y el contenido, y creará un árbol sin hijos.

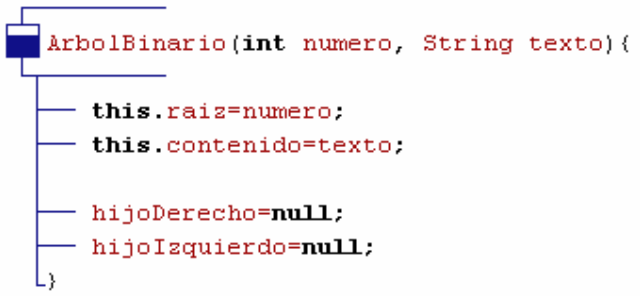

Fig. 5.3.2 Método constructor

En esta implementación, definiremos los siguientes métodos:

NOMBRE DEL METODO	VALOR DE RETORNO	TIPOS DE ARGUMENTO	UTILIDAD
insertar(*árbol, número, texto*)	ArbolBinario	ArbolBinario, int, String	Inserta un nuevo árbol con la llave *número* y el contenido *texto* en el árbol *árbol*
buscar(*numero*)	String	int	Busca el árbol con la raíz *número* y retorna su contenido
mostrarPreOrden()	void	ninguno	Recorre el árbol en pre-orden
mostrarInOrden()	void	ninguno	Recorre el árbol en in-orden
mostrarPosOrden()	void	ninguno	Recorre el árbol en pos-orden

5.3.1 Método buscar()

Tal como fue explicado anteriormente, usaremos el siguiente algoritmo de búsqueda:

```
public String
buscar(int numero){
    if(numero==raiz)
        return contenido;
    else{
        if(numero > raiz)
            if(hijoDerecho!=null)
                return hijoDerecho.buscar(numero);
            else
                return ("No existe el elemento con el numero "+numero);
        else{
            if(hijoIzquierdo!=null)
                return hijoIzquierdo.buscar(numero);
            else
                return ("No existe el elemento con el numero "+numero);
        }
    }
}
```

Fig. 5.3.1.1 Método buscar()

5.3.2 Método insertar()

Tal como fue explicado anteriormente, usaremos el siguiente algoritmo de inserción:

```
public static ArbolBinario
insertar(ArbolBinario arbol, int numero, String texto){
    if(arbol==null){
        ArbolBinario nuevoArbol=new ArbolBinario(numero,texto);
        return nuevoArbol;
    }
    else{
        if(arbol.raiz < numero){
            ArbolBinario nuevoDerecho=insertar(arbol.hijoDerecho,numero,texto);
            arbol.hijoDerecho=nuevoDerecho;
        }
        else{
            if(arbol.raiz > numero){
                ArbolBinario nuevoIzquierdo=insertar(arbol.hijoIzquierdo,numero,texto);
                arbol.hijoIzquierdo=nuevoIzquierdo;
            }
        }
        return arbol;
    }
}
```

Fig. 5.3.2.1 Método insertar()

5.3.3 Método mostrarPreOrden()

Recordemos que para recorrer un árbol binario no vacío en pre-orden se debe:

4. visitar la raíz
5. recorrer recursivamente el subárbol izquierdo
6. recorrer recursivamente el subárbol derecho

Así que definiremos el método de esta forma:

```
public void mostrarPreOrden(){
    // visitar la raiz
    System.out.println("numero : "+raiz+ " Contenido: "+contenido);

    // recorrer recursivamente el subárbol izquierdo
    if (hijoIzquierdo!=null)
        hijoIzquierdo.mostrarPreOrden();

    // recorrer recursivamente el subárbol izquierdo
    if (hijoDerecho!=null)
        hijoDerecho.mostrarPreOrden();
}
```

Fig. 5.3.2.1 Método insertar()

5.3.4 Método mostrarInOrden()

Recordemos que para recorrer un árbol binario no vacío en in-orden se debe:

1. recorrer recursivamente el subárbol izquierdo
2. visitar la raíz
3. recorrer recursivamente el subárbol derecho

Así que definiremos el método de esta forma:

```
public void mostrarInOrden(){
    // recorrer recursivamente el subárbol izquierdo
    if (hijoIzquierdo!=null)
        hijoIzquierdo.mostrarInOrden();

    // visitar la raiz
    System.out.println("numero : "+raiz+ " Contenido: "+contenido);

    // recorrer recursivamente el subárbol derecho
    if (hijoDerecho!=null)
        hijoDerecho.mostrarInOrden();
}
```

Fig. 5.3.4.1 Método mostrarInOrden()

5.3.5 Método mostrarPosOrden()

Recordemos que para recorrer un árbol binario no vacío en pos-orden se debe:

4. recorrer recursivamente el subárbol izquierdo
5. recorrer recursivamente el subárbol derecho
6. visitar la raíz

Así que definiremos el método de esta forma:

```
public void mostrarPosOrden(){

    // recorrer recursivamente el subárbol izquierdo
    if (hijoIzquierdo!=null)
        hijoIzquierdo.mostrarPosOrden();

    // recorrer recursivamente el subárbol derecho
    if (hijoDerecho != null)
        hijoDerecho.mostrarPosOrden();

    // visitar la raíz
    System.out.println("numero : "+raiz+ " Contenido: "+contenido);
}
```

Fig. 5.3.5.1 Método mostrarPosOrden()

5.3.6 Desafíos para el alumno

Agregue a la clase ArbolBinario anteriormente presentada un método igual al de inserción, pero que, en vez de recibir un **int** y un String, reciba un **int**[] y un String[] e inserte todos esos datos al árbol en el orden que indiquen los índices de los arreglos.

6. La aplicación ilustrativa

En la carpeta *"Aplicación Ilustrativa"* se encuentra el archivo *AplicacionIlustrativa.java*, el cual tiene por objetivo, tal como su nombre lo dice, ilustrar gráficamente el funcionamiento de las estructuras de datos anteriormente presentadas. Puedes ejecutarla con el pcGrasp o jGrasp, o bien, simplemente hacer doble click sobre el archivo *AplicacionIlustrativa.exe*, dentro de la misma carpeta. Es de muy fácil uso y muestra como funcionan las listas enlazadas, pilas y colas. En el caso que tuvieras alguna duda de su funcionamiento, solo haz click en el botón *Ayuda*. Esta aplicación te servirá también para aprender y practicar el uso del paquete de clases *swing*, así que deberías mirar el código y tratar de entenderlo. Como desafío, te propongo que hagas una aplicación parecida para el árbol binario, en la cual se puedan ingresar y buscar datos, como también recorrer el árbol en pre, in y pos-orden.

Recuerda que para que la aplicación ilustrativa funcione en tu computador, debes tener instalada, al menos, la versión 1.4.2 de java. Si no, deja como comentario las líneas 514 y 515 y vuelve a compilar.

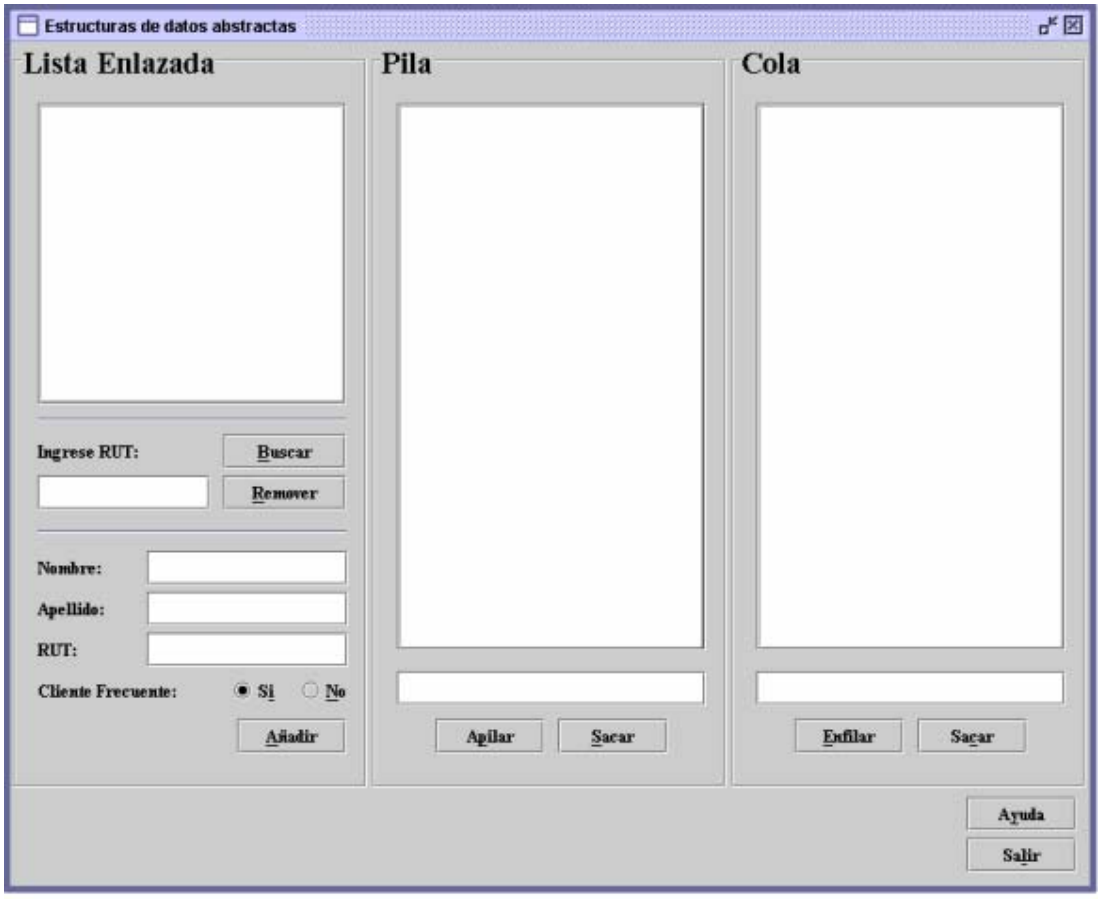

Fig. 6.1 Aplicación ilustrativa

En la carpeta *"ListaEnlazada"* también incluí un ejemplo de uso de las listas, llamado *AplicacionAdministrativa.java*, pero que corre en la consola del sistema y usa flujos de datos para la interacción con en usuario. El código está detalladamente explicado con comentarios dentro del mismo. Revisen detalladamente el funcionamiento de esta clase para comprender mejor el uso de las estructuras de datos abstractas. Como desafío, creen una aplicación parecida a ésta, pero que en vez de usar listas enlazadas, use colas y/o pilas.

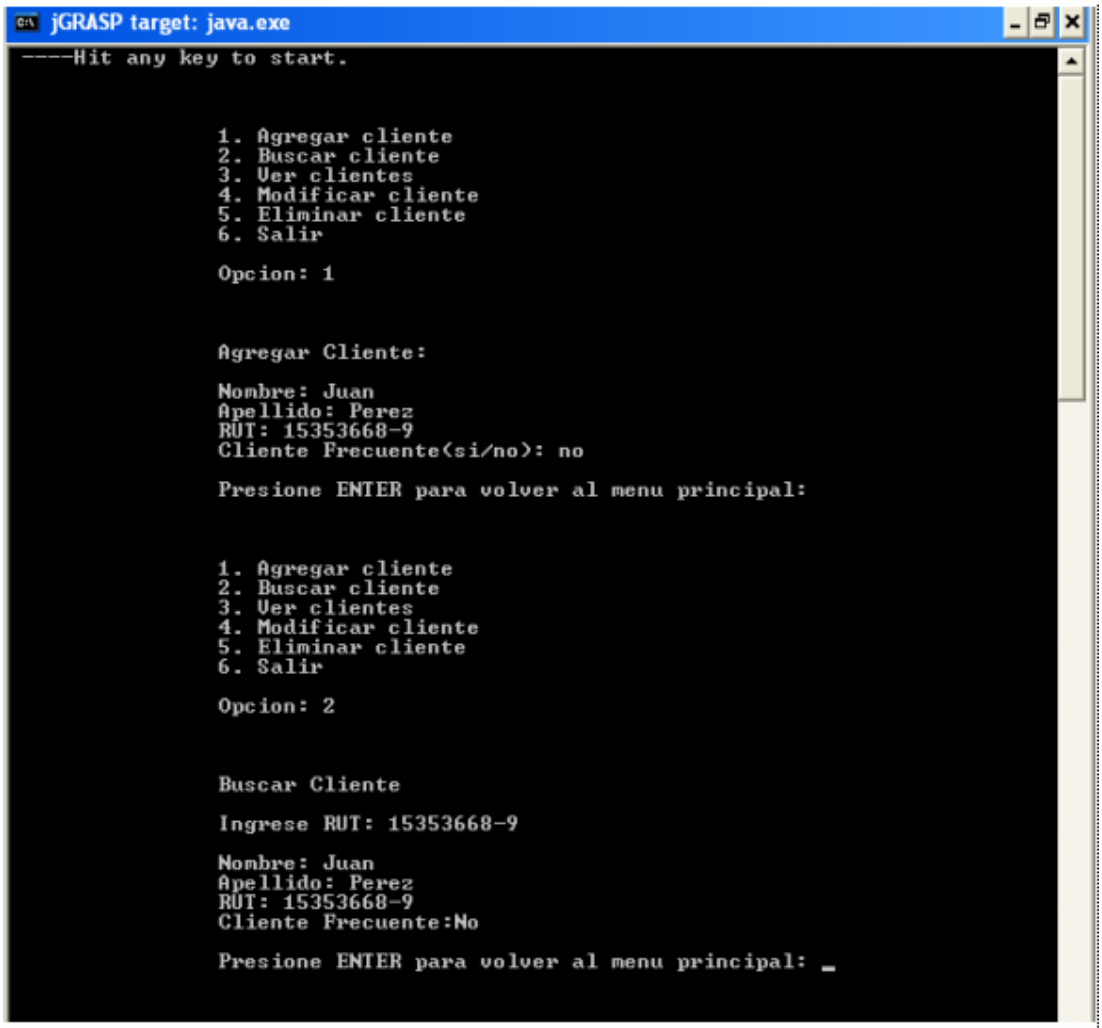

Fig. 6.2 Aplicación administrativa