Estructuras de Datos en Java

1. Antecedentes

Introducción a la Orientación a Objetos

La programación orientada a objetos (POO) es una nueva manera de enfocar la programación. Desde sus comienzos, la programación ha estado gobernada por varias metodologías. En cada punto crítico de la evolución de la programación se creaba un nuevo enfoque para ayudar al programador a manejar programas cada vez más complejos. Los primeros programas se crearon mediante un proceso de cambio de los conmutadores del panel frontal de la computadora. Obviamente, este enfoque solo es adecuado para programas pequeños. A continuación se invento el lenguaje ensamblador que permitió escribir programas más largos. El siguiente avance ocurrió en los años 50 cuando se invento el primer lenguaje de alto nivel (FORTRAN).

Mediante un lenguaje de alto nivel, un programador estaba capacitado para escribir programas que tuvieran una longitud de varios miles de líneas. Sin embargo, el método de programación usado en el comienzo era un enfoque *adhoc* que no solucionaba mucho. Mientras que esto esta bien para programas relativamente cortos, se convierte en "código espagueti" ilegible y difícil de tratar cuando se aplica a programas más largos. La eliminación del código espagueti se consiguió con la creación de los *lenguajes de programación estructurados* en los años sesenta. Estos lenguajes incluyen ALGOL y PASCAL. En definitiva, C es un lenguaje estructurado, y casi todos los tipos de programas que se han estado haciendo se podrían llamar programas estructurados.

Los programas estructurados se basan en estructuras de control bien definidas, bloques de código, la ausencia del GOTO, y subrutinas independientes que soportan recursividad y variables locales. La esencia de la programación estructurada es la reducción de un programa a sus elementos constitutivos. Mediante la programación estructurada un programador medio puede crear y mantener programas de una longitud superior a 50,000 líneas.

Aunque la programación estructurada nos ha llevado a excelentes resultados cuando se ha aplicado a programas moderadamente complejos, llega a fallar en algún punto cuando el programa alcanza un cierto tamaño. Para poder escribir programas de mayor complejidad se necesitaba de un nuevo enfoque en la tarea de programación. A partir de este punto nace la programación orientada a objetos (POO). La POO toma las mejores ideas incorporadas en la programación estructurada y las combina con nuevos y potentes conceptos que permiten organizar los programas de forma más efectiva. La POO permite descomponer un problema en subgrupos relacionados. Cada subgrupo pasa a ser un objeto autocontenido que contiene sus propias instrucciones y datos que le relacionan con ese objeto. De esta

manera, la complejidad se reduce y el programador puede tratar programas más largos.

Todos los lenguajes de POO comparten tres características: encapsulación, polimorfismo y herencia.

Encapsulación.

La encapsulación es el mecanismo que agrupa el código y los datos que maneja y los mantiene protegidos frente a cualquier interferencia y mal uso. En un lenguaje orientado a objetos, el código y los datos suelen empaquetarse de la misma forma en que se crea una "caja negra" autocontenida. Dentro de la caja son necesarios tanto el código como los datos. Cuando el código y los datos están enlazados de esta manera, se ha creado un objeto. En otras palabras, un objeto es el dispositivo que soporta encapsulación.

En un objeto, los datos y el código, o ambos, pueden ser privados para ese objeto o públicos. Los datos o el código privado solo los conoce o son accesibles por otra parte del objeto. Es decir, una parte del programa que esta fuera del objeto no puede acceder al código o a los datos privados. Cuando los datos o el código son públicos, otras partes del programa pueden acceder a ellos, incluso aunque este definido dentro de un objeto. Normalmente, las partes públicas de un objeto se utilizan para proporcionar una interfaz controlada a las partes privadas del objeto.

Para todos los propósitos, un objeto es una variable de un tipo definido por el usuario. Puede parecer extraño que un objeto que enlaza código y datos se pueda contemplar como una variable. Sin embargo, en programación orientada a objetos, este es precisamente el caso. Cada vez que se define un nuevo objeto, se esta creando un nuevo tipo de dato. Cada instancia específica de este tipo de dato es una variable compuesta.

Polimorfismo.

Polimorfismo (del Griego, cuyo significado es "muchas formas") es la cualidad que permite que un nombre se utilice para dos o más propósitos relacionados pero técnicamente diferentes. El propósito del polimorfismo aplicado a la POO es permitir poder usar un nombre para especificar una clase general de acciones. Dentro de una clase general de acciones, la acción específica a aplicar está determinada por el tipo de dato. Por ejemplo, en C, que no se basa significativamente en el polimorfismo, la acción de valor absoluto requiere tres funciones distintas: **abs(), labs() y fabs()**. Estas tres funciones calculan y devuelven el valor absoluto de un entero, un entero largo y un valor real, respectivamente. Sin embargo, en C++, que incorpora polimorfismo, a cada función se puede llamar **abs()**.

El tipo de datos utilizado para llamar a la función determina que versión específica de la función se esta usando, es decir, es posible usar un

nombre de función para propósitos muy diferentes. Esto se llama *sobrecarga de funciones*.

De manera general, el concepto de polimorfismo es la idea de "una interfaz, múltiples métodos". Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades relacionadas. Sin embargo, la acción específica ejecutada depende de los datos. La ventaja del polimorfismo es que ayuda a reducir la complejidad permitiendo que la misma interfaz se utilice para especificar una *clase general de acción*. Es trabajo del compilador seleccionar la *acción específica* que se aplica a cada situación. El programador no necesita hacer esta selección manualmente, solo necesita recordar y utilizar la interfaz general.

El polimorfismo se puede aplicar tanto a funciones como a operadores, prácticamente todos los lenguajes de programación contienen una aplicación limitada de polimorfismo cuando se relaciona con los operadores aritméticos, por ejemplo, en C, el signo + se utiliza par añadir enteros, enteros largos, caracteres y valores reales. En estos casos, el compilador automáticamente sabe que tipo de aritmética debe aplicar, en C++, se puede ampliar este concepto a otros tipos de datos que se definan, este tipo de polimorfismo se llama *sobrecarga de operadores*.

Herencia.

La *herencia* es el proceso mediante el cual un objeto puede adquirir las propiedades de otro. Mas en concreto, un objeto puede heredar un conjunto general de propiedades a alas que puede añadir aquellas características que son específicamente suyas. La herencia es importante porque permite que un objeto soporte el concepto de *clasificación jerárquica*. Mucha información se hace manejable gracias a esta clasificación, por ejemplo, la descripción de una casa. Una **casa** es parte de una clase general llamada **edificio**, a su vez, **edificio** es una parte de la clase mas general **estructura**, que es parte de la clase aun más general de objetos que se puede llamar **obra-hombre**.

En cualquier caso, la clase hija hereda todas las cualidades asociadas con la clase padre y le añade sus propias características definitorias. Sin el uso de clasificaciones ordenadas, cada objeto tendría que definir todas las características que se relacionan con él explícitamente.

Sin embargo, mediante el uso de la herencia, es posible describir un objeto estableciendo la clase general (o clases) a las que pertenece, junto con aquellas características específicas que le hacen único.

Tipos de Datos Abstractos

Abstracción: consiste en ignorar los detalles de la manera particular en que está hecha una cosa, quedándonos solamente con su visión general. Tipo de Dato Abstracto (TDA) se define como un conjunto de valores que pueden tomar los datos de ese tipo, junto a las operaciones que los manipulan.

Un TDA es un modelo matemático de estructuras de datos que especifican los tipos de datos almacenados, las operaciones definidas sobre esos datos y los tipos de parámetros de esas operaciones.

TAD = Valores (tipo de dato) + operaciones

Un TDA define lo **que** cada operación debe hacer, más no **como** la debe hacer. En un lenguaje de programación como Java un TDA puede ser expresado como una interface, que es una simple lista de declaraciones de métodos.

Un TDA es materializado por una estructura de datos concreta, en Java, es modelada por una clase. Una clase define los datos que serán almacenados y las operaciones soportadas por los objetos que son instancia de la clase. Al contrario de las interfaces, las clases especifican **como** las operaciones son ejecutadas (implementación).

Ejemplos de tipos de datos abstractos son las Listas, Pilas, Colas, etc., que se discutirán más adelante.

Estructuras de Datos

En programación, una ***estructura de datos*** es una forma de organizar un conjunto de datos elementales (un dato elemental es la mínima información que se tiene en el sistema) con el objetivo de facilitar la manipulación de estos datos como un todo y/o individualmente.

Una estructura de datos define la organización e interrelacionamiento de estos, y un conjunto de operaciones que se pueden realizar sobre él. Las operaciones básicas son:

- Alta, adicionar un nuevo valor a la estructura.
- Baja, borrar un valor de la estructura.
- Búsqueda, encontrar un determinado valor en la estructura para se realizar una operación con este valor, en forma SECUENCIAL o BINARIO (siempre y cuando los datos estén ordenados).

Otras operaciones que se pueden realizar son:

- Ordenamiento, de los elementos pertenecientes a la estructura.
- Apareo, dadas dos estructuras originar una nueva ordenada y que contenga a las apareadas.

Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada para cada problema depende de factores como las frecuencias y el orden en que se realiza cada operación sobre los datos.

Algunas estructuras de datos utilizadas en programación son:

- Arrays (Arreglos)
    - Vectores
    - Matrices
- Listas Enlazadas
    - Listas simples
    - Listas dobles
    - Listas Circulares
- Pilas
- Colas
- Árboles
    - Árboles binarios
    - Árboles Multicamino
- Conjuntos
- Grafos
- Montículos

Acceso directo y secuencial a los datos

- Secuencial. Para acceder a un objeto se debe acceder a los objetos almacenados previamente en el archivo. El acceso secuencial exige elemento a elemento, es necesario una exploración secuencial comenzando desde el primer elemento.

- Directo o Aleatorio. Se accede directamente al objeto, sin recorrer los anteriores. El acceso directo permite procesar o acceder a un elemento determinado haciendo una referencia directamente por su posición en el soporte de almacenamiento.

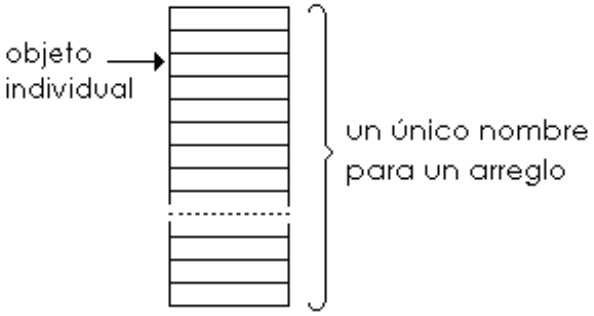
Resumen de Java

2. Arreglos

Arreglos lineales

Un arreglo se usa para agrupar, almacenar y organizar datos de un mismo tipo. En un arreglo cada valor se almacena en una posición numerada específica dentro del arreglo. El número correspondiente a cada posición se conoce como índice.

La estructura de un arreglo es:



Normalmente el primer objeto del arreglo tiene el índice 0, aunque esto varía de lenguaje en lenguaje.

Declaración:

Como se sabe, hay 2 tipos de datos en Java: primitivos (como int y double) y objetos. En muchos lenguajes de programación (aún en Orientados a Objetos como C++) los arreglos son tipos primitivos, pero en Java son tratados como objetos. Por consiguiente, se debe utilizar el operador new para crear un arreglo:

```
int miArreglo[];                    // define la referencia a un arreglo
miArreglo = new int[100];           // crea el arreglo y establece miArreglo
                                    // como referencia a él
```

O el equivalente a hacerlo dentro de una misma sentencia

```
int miArreglo[] = new int[100];
```

El operador [] es la señal para el compilador que estamos declarando un arreglo (objeto) y no una variable ordinaria. Dado que un arreglo es un objeto, el nombre –miArreglo-, en el ejemplo anterior- es una referencia a un arreglo y no el nombre del arreglo como tal. El arreglo es almacenado en una dirección de memoria x, y miArreglo almacena solo esa dirección de memoria.

Los arreglos tienen el método length, el cual se utiliza para encontrar el tamaño de un arreglo:

```
int tamaño = miArreglo.length;  // arroja el tamaño del arreglo
```

Para acceder a los elementos de un arreglo se debe utilizar los corchetes, muy similar a otros lenguajes de programación:

```
temp =  miArreglo[4];           // guarda en temp el valor del cajón 4 del arreglo
miArreglo[8] = 50;              // almacena en el cajon 8 del arreglo el valor 50
```

Es preciso recordar que en Java, como en C y C++, el primer elemento del arreglo es el 0. Luego entonces, los índices de un arreglo de 10 elementos van del 0 al 9.

Ejemplo:

```
// array.java
// demonstrates Java arrays
////////////////////////////////////////////////////////////////
class ArrayApp
{
public static void main(String[] args) {
	int[] arr; // reference
	arr = new int[100]; // make array
	int nElems = 0; // number of items
	int j; // loop counter
	int searchKey; // key of item to search for
//-----------------------------------------------------------
	arr[0] = 77; // insert 10 items
	arr[1] = 99;
	arr[2] = 44;
	arr[3] = 55;
	arr[4] = 22;
	arr[5] = 88;
	arr[6] = 11;
	arr[7] = 00;
	arr[8] = 66;
	arr[9] = 33;
	nElems = arr.length; // now 10 items in array
//-----------------------------------------------------------
	for (j=0; j<nElems-1; j++) // display items {
		System.out.print(arr[j] + " ");
		System.out.println("");
	}
//-----------------------------------------------------------
	searchKey = 66; // find item with key 66
	for (j=0; j<nElems; j++) // for each element,
		if(arr[j] == searchKey) // found item?
			break; // yes, exit before end
	if(j == nElems) // at the end?
		System.out.println("Can't find " + searchKey); // yes
	else
		System.out.println("Found " + searchKey); // no
//-----------------------------------------------------------
	searchKey = 55; // delete item with key 55
	for (j=0; j<nElems; j++) // look for it
		if(arr[j] == searchKey)
			break;
	for (int k=j; k<nElems; k++) // move higher ones down
		arr[k] = arr[k+1];
	nElems--; // decrement size
//-----------------------------------------------------------
	for (j=0; j<nElems; j++) { // display items
		System.out.print( arr[j] + " ");
		System.out.println("");
	}
} // end main()
} // end class ArrayApp
```

En el anterior programa, creamos un arreglo llamado arr, almacena 10 elementos dentro de él, después busca por el elemento 66, despliega todos los elementos, elimina el elemento con valor 55, y finalmente despliega los 9 elementos restantes. La salida el programa sería similar a la siguiente:

```
77 99 44 55 22 88 11 0 66 33
```

Found 66
77 99 44 22 88 11 0 66 33

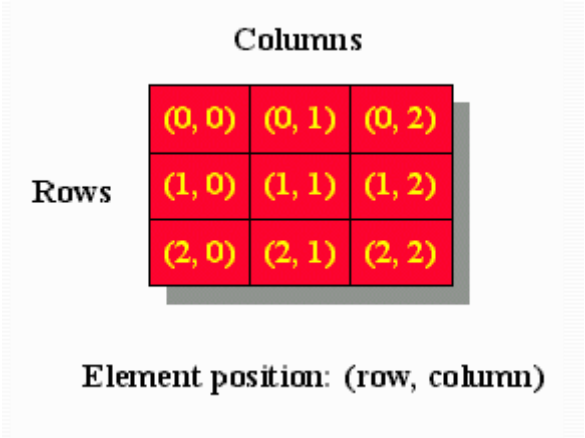
Operaciones sobre arreglos

Se pueden implementar una serie de operaciones básicas sobre los arreglos, a saber:

- Insertar elementos
- Búsqueda de elementos
- Eliminar elementos
- Mostrar los elementos
- Ordenar los elementos
- Modificar algún elemento

Arreglos bidimensionales

Un arreglo de dos dimensiones, también llamado tabla o matriz, donde cada elemento se asocia con una pajera de índices, es otro arreglo simple. Conceptualizamos un arreglo bidimensional como una cuadrícula rectangular de elementos divididos en filas y columnas, y utilizamos la notación (fila, columna) para identificar un elemento específico. La siguiente figura muestra la visión conceptual y la notación específica de los elementos:



El trabajo con los arreglos bidimensionales es muy parecido a los arreglos lineales, la declaración de uno de ellos se da de la siguiente manera:

```
double [][] temperatures = new double [2][2]; // Dos filas y dos columnas .
```

La asignación de datos, se hace de una forma similar a los arreglos lineales:

```
temperatures [0][0] = 20.5; // Populate row 0
temperatures [0][1] = 30.6;
temperatures [0][2] = 28.3;

temperatures [1][0] = -38.7; // Populate row 1
temperatures [1][1] = -18.3;
temperatures [1][2] = -16.2;
```

Las operaciones con las matrices son muy variadas, desde las básicas como insertar y eliminar elementos hasta multiplicaciones de matrices, entre otras.

3. Listas ligadas

Definición

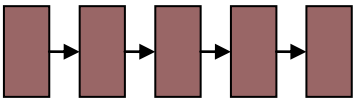
En el capítulo anterior, vimos a la estructura llamada “arreglos” o vectores. Esta estructura presenta ciertas desventajas:

- en un arreglo desordenado, la búsqueda es lenta
- en un arreglo ordenado, la inserción es lenta
- en ambos casos, la eliminación es lenta
- el tamaño de un arreglo no puede cambiar después que se creó

Ahora veremos una estructura de datos que resuelve los problemas anteriores, llamada lista ligada. Las listas ligadas, son probablemente, la segunda estructura de almacenamiento de propósito general más comúnmente utilizadas, después de los arreglos.

Una lista ligada es un mecanismo versátil conveniente para su uso en muchos tipos de bases de datos de propósito general. También puede reemplazar a los arreglos como base para otras estructuras de almacenamiento como pilas y colas.

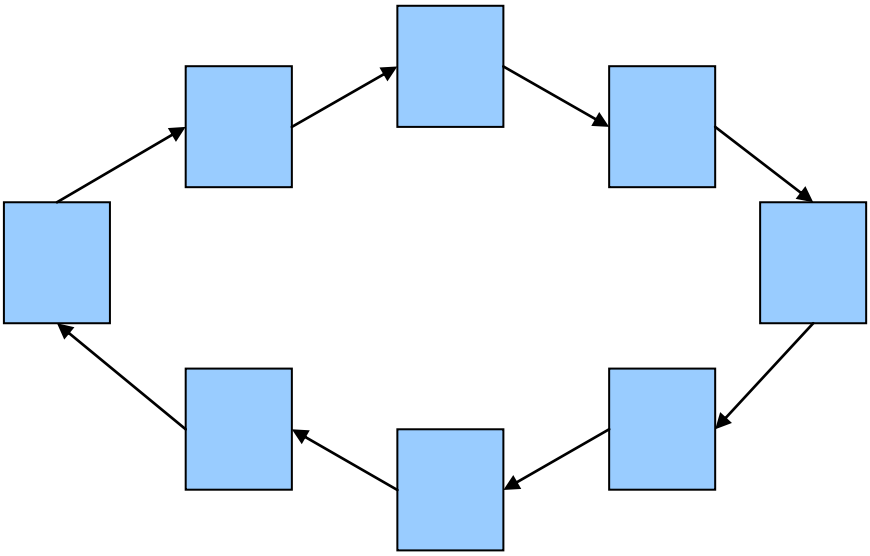
La *ventaja* más evidente de utilizar estructuras ligadas, es que permite optimizar el uso de la memoria, pues no desperdiciamos el espacio de localidades vacías:



La *desventaja* más grande de las estructuras ligadas es que deben ser recorridas desde su inicio para localizar un dato particular. Es decir, no hay forma de acceder al i-ésimo dato de la lista, como lo haríamos en un arreglo.

Algunas listas más complejas son las listas doblemente ligadas o las listas circulares, por nombrar algunas.

Lista doblemente ligada

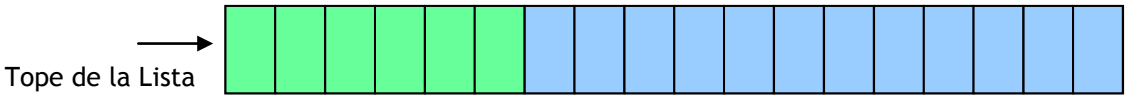


Lista circular

Listas desordenadas

Desde el punto de vista de programación, una lista es una estructura de datos dinámica que contiene una colección de elementos homogéneos, con una relación lineal entre ellos. Una relación lineal significa que cada elemento (a excepción del primero) tiene un precedente y cada elemento (a excepción del último) tiene un sucesor.

Gráficamente, una lista se puede conceptualizar de la siguiente forma:



Lógicamente, una lista es tan solo un objeto de una clase *Lista* que se conforma de un tope (posición de inicio) de lista y un conjunto de métodos para operarla.

Se pueden implementar listas estáticas utilizando arreglos de N elementos donde se almacenen los datos. Recordemos la desventaja de esto.

Listas ordenadas

Una lista ordenada, es una extensión de las listas normales con la cualidad de que todos sus datos se encuentran en orden. Como al igual que las listas se debe poder insertar datos, hay dos formas de garantizar el orden de los mismos:

- crear un método de inserción ordenada
- crear un método de ordenamiento de la lista

Operaciones sobre listas

Los distintos métodos de la clase Lista deben asegurarnos poder insertar, eliminar u obtener un dato en cualquier posición de la lista.

Como se mencionó, una lista se compone de

- un tope de la lista, que es un objeto de la clase NodoDeLista
- un conjunto de instrucciones que controlen su estructura

La clase NodoDeLista debe contener al menos un espacio de almacenamiento de datos y un objeto de la clase NodoDeLista no necesariamente instanciado. La clase NodoDeLista debe ser nuestra clase base, que incorporará los constructores necesarios para inicializar nuestros datos. El objetivo es una estructura de clase que nos represente:

Ejemplo clase NodoDeLista

```
public class NodoDeLista{
        String dato;
        NodoDeLista siguiente;
        public NodoDeLista(){
                dato = “”;
                siguiente = null;
        }
        public NodoDeLista(String d){
                dato = d;
                siguiente = null;
        }
}
```

La clase Lista

La clase lista contiene:

- un objeto de tipo NodoDeLista que será la primera referencia de nuestra lista

- un grupo de constructores para inicializarla
- un grupo de operaciones para el control de la lista, tales como:
  - insertar un nuevo NodoDeLista
  - buscar un NodoDeLista
  - eliminar un NodoDeLista
  - obtener un NodoDeLista en alguna posición
  - mostrar la lista

Se crearán los métodos uno a uno. La primera aproximación quedaría así:

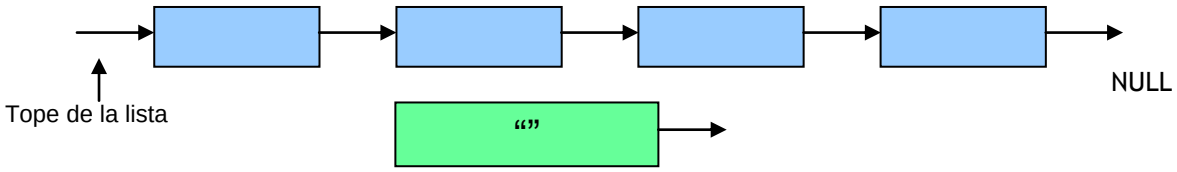
```
public class Lista{
        NodoDeLista tope;
        public Lista(){}
        void insertar(int){}
        void eliminar(int){}
        int buscar(int){}
        boolean vacia();
        void mostrar(){}
}
```

En el ejemplo, se incluye un constructor que crea una lista vacía.

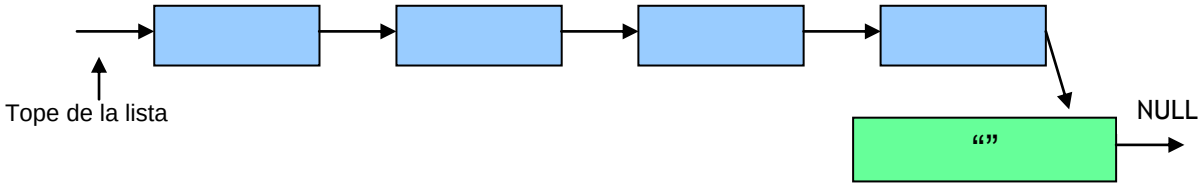
Insertar en una lista

Insertar un dato en una lista se refiere a recorrerla para agregar un dato al final de la lista (listas desordenadas) o en una posición determinada (listas ordenadas). En este caso, insertamos el NodoDeLista al final de la lista. El recorrido se inicia haciendo actual = tope:

1. Crear un nuevo NodoDeLista



2. Recorrer la lista hasta el final, es decir, cuando el valor del campo siguiente del nodo es NULL



3. El nuevo NodoDeLista ahora es parte de la lista

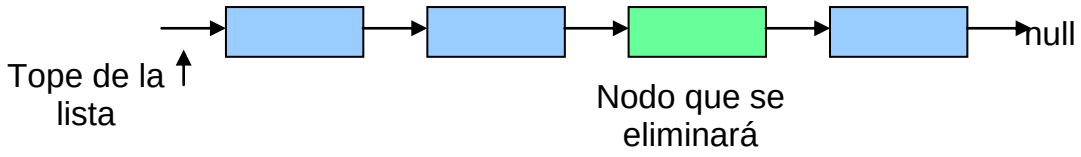
Consideraciones:

a) ¿qué pasa si la lista esta vacía?

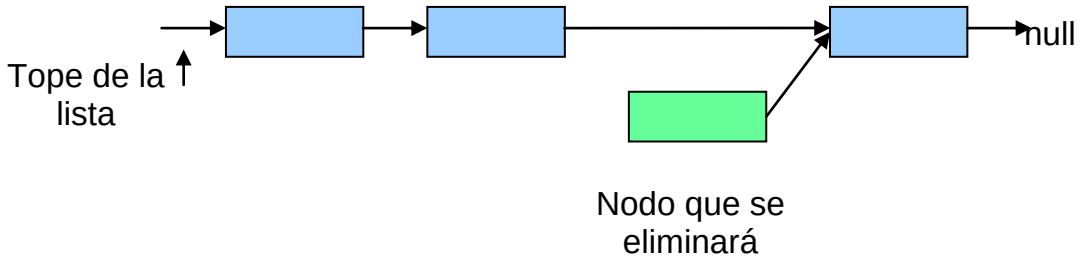
Eliminar un nodo de la lista

Eliminar un nodo significa quitarlo, no importando su posición, pero cuidando que no se pierdan los datos. Para eliminar un nodo, primero debemos encontrarlo. El recorrido se inicia haciendo igual = tope:

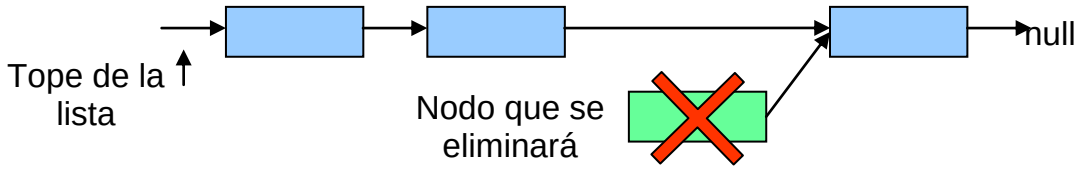
1. Localizar el nodo que queremos eliminar



2. Localizando el nodo de la lista, cambiar el campo siguiente del nodo anterior hacia el siguiente del nodo a eliminar.



3. Como ningún objeto hace referencia al nodo, este se eliminará al terminar de ejecutarse el método.



Mostrar la lista

El método mostrar escribe a pantalla el contenido de la lista, de manera que podamos confirmar su contenido. El método es tan sencillo como recorrer la lista elemento a elemento y escribir su contenido en la pantalla, hasta que encontremos el final de la lista. El recorrido, como todos se inicia haciendo actual = tope.

Funciones adicionales

Se pueden construir un grupo de funciones adicionales que nos proporcionen información sobre la lista, tales como:

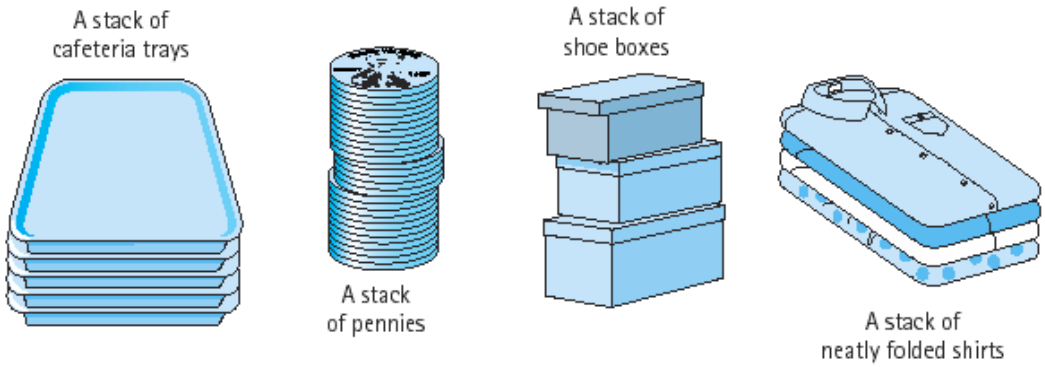
a) tamaño de la lista
b) lista vacía

c) fin de la lista
d) insertar en una lista ordenada
e) ordenar una lista desordenada
f) buscar un elemento de la lista

4. Pilas

Definición

Considere los elementos de la figura siguiente:



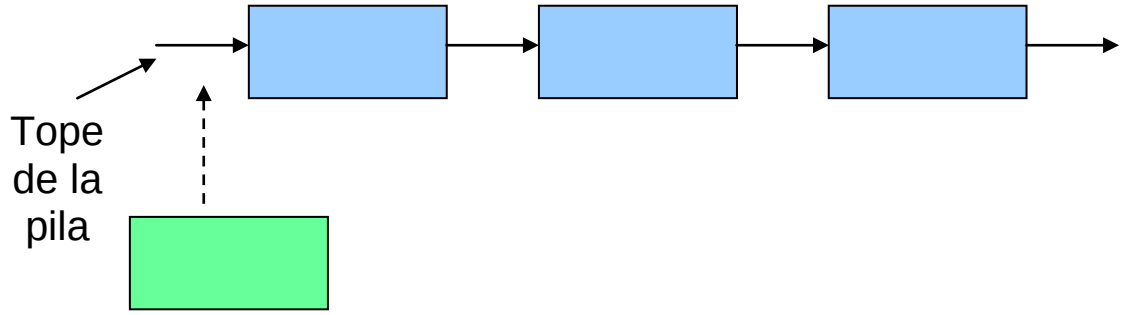
Aunque los objetos son todos diferentes, todos ilustran el concepto de pila (stack). Una pila, es una particularización de las listas y se puede definir como una estructura en la cual los elementos son agregados y eliminados en el tope de la lista. Es una estructura LIFO (Last In First Out – el primero que llega es el último que sale).

Por ejemplo, si nuestra camisa favorita azul se encuentra debajo de la vieja camisa roja que se encuentra en el tope de las camisas, primero debemos sacar a la camisa roja. Solamente así podremos sacar a la camisa azul, la cual ahora esta en el tope de la pila de camisas. Una ves sacada la camisa azul, podemos regresar a la vieja camisa roja al tope de la pila.

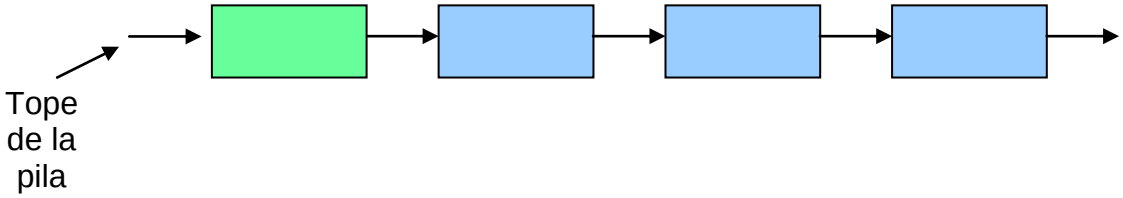
Las pilas cuentan solo con 2 operaciones, conocidas como PUSH, insertar un elemento en el tope de la pila, y POP, leer el elemento del tope de la pila. Veamos como funcionan estos métodos:

PUSH

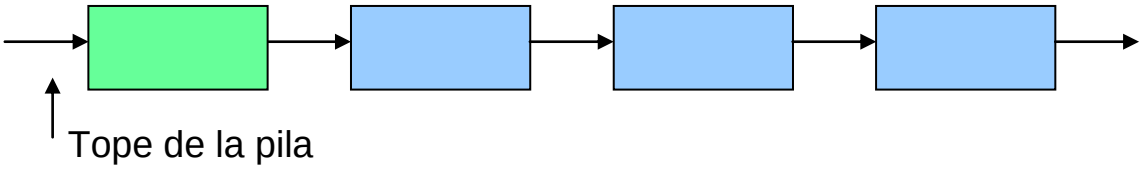
1.- Crear un NodoDeLista nuevo



2.- Hacer el siguiente de nuevo igual al tope



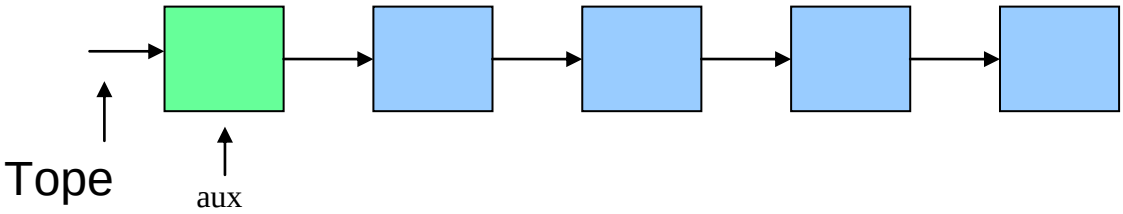
3.- Hacer el tope de la pila igual al nuevo nodo



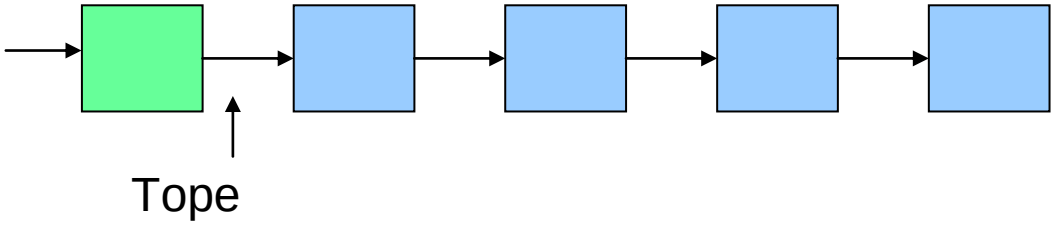
Se ha agregado el dato a la pila!!

POP

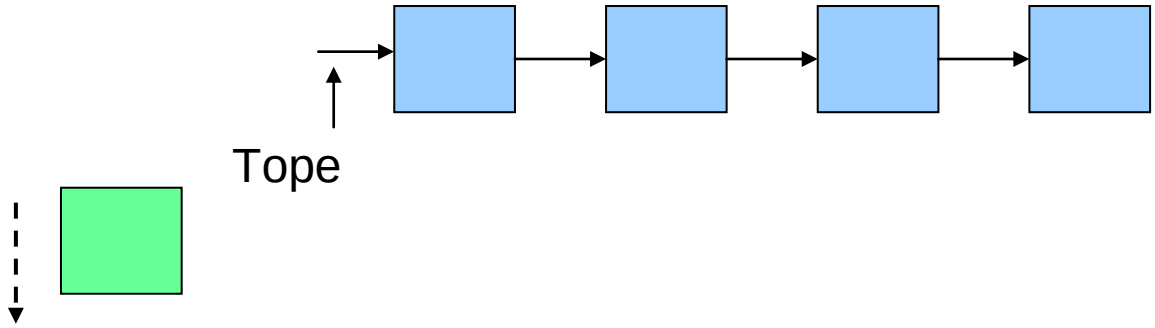
1.- Crear un nodo temporal que se instancia al tope de la pila



2.- Hacer el tope de la pila igual al siguiente del temporal



3.- Regresar el valor del nodo temporal antes que desaparezca



Implementación de pilas

Para trabajar con pilas, utilizaremos la misma base con la que trabajamos para las listas. Dispondremos de 2 clases: Pila y NodoDePila. Con la primera crearemos a la pila y las operaciones sobre ella y con la segunda instanciaremos a cada uno de los nodos de la pila.

```
public class Pila{
        NodoDePila tope;
        public Pila(){}
        void Push(){}
        int Pop(){}
        void mostrar(){}
}
```

```
class NodoDePila{
    int num;
    NodoDeLista sig;
    public NodoDePila(){
        num = 0;
        sig = null;
     }
     public NodoDePila(int n){
        num = n;
        sig = null;
     }
     public NodoDePila(int n,NodoDePila s){
                num = n;
                sig = s;
    }
}
```

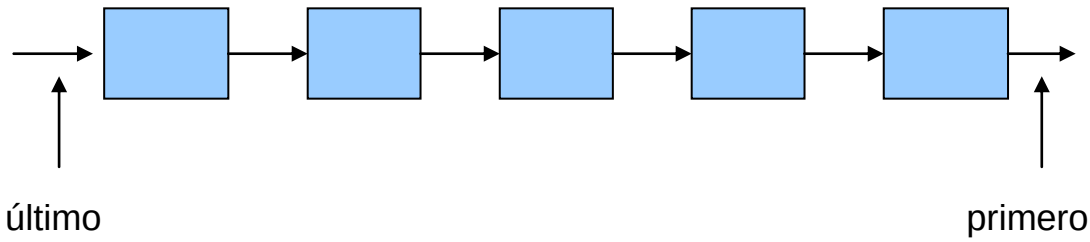
5. Colas

Definición

Una cola es una estructura de datos similar a una lista con restricciones especiales. Los elementos son agregados en la parte posterior de la cola y son eliminados por el frente. Esta estructura es llamada FIFO (First In, First Out – el primero que llega, es el primero en salir). Considere la siguiente figura:

Si una persona ha elegido sus libros dentro de la librería y desea pasar a pagarlos, debe hacer "cola" para poder efectuar el pago. La cajera atenderá a cada persona en el orden en que llegaron.

A nivel de lógico, podemos representar a una cola de la siguiente manera:



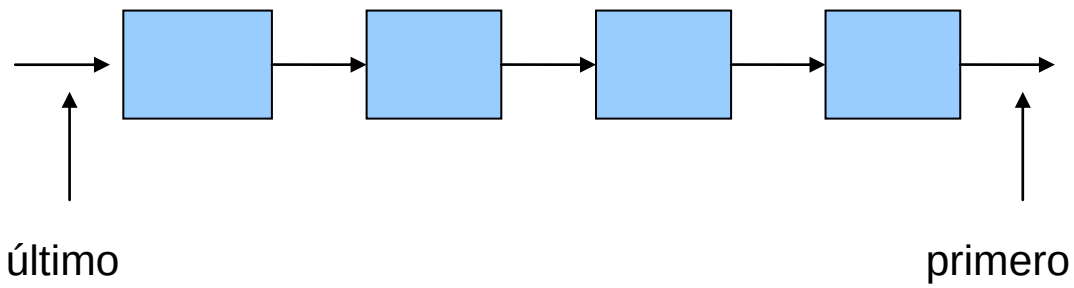
Construcción de colas

Implementar colas involucra el uso clases similares a una lista. Las operaciones sobre esta estructura son idénticas: insertar y eliminar, con las consideraciones pertinentes.

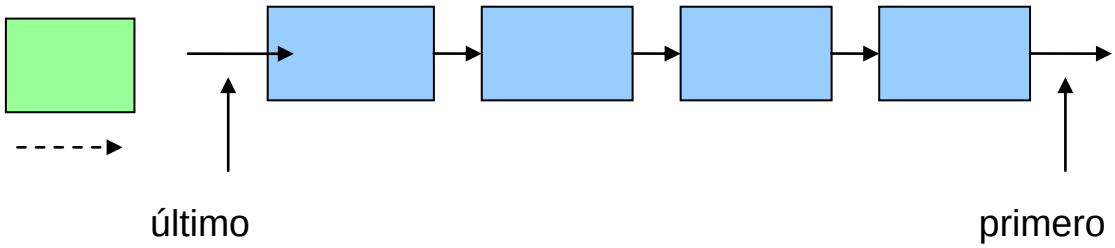
Insertar

Insertar un dato en una cola es muy similar a hacerlo en una pila o una lista, la diferencia es que tendremos que hacerlo por el final de la cola. A este proceso se le llama encolar.

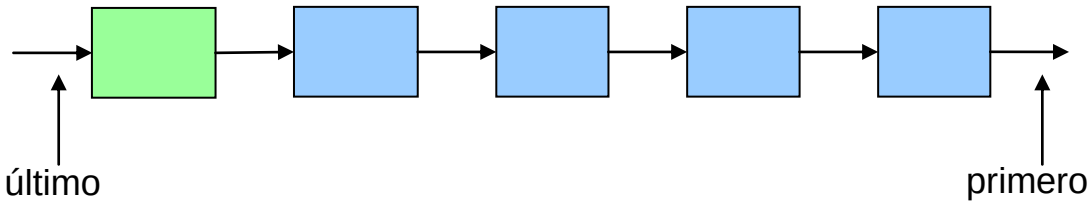
1.- Cola inicial



2.- Creamos el nuevo nodo



3.- Hacemos que último apunte al nuevo nodo

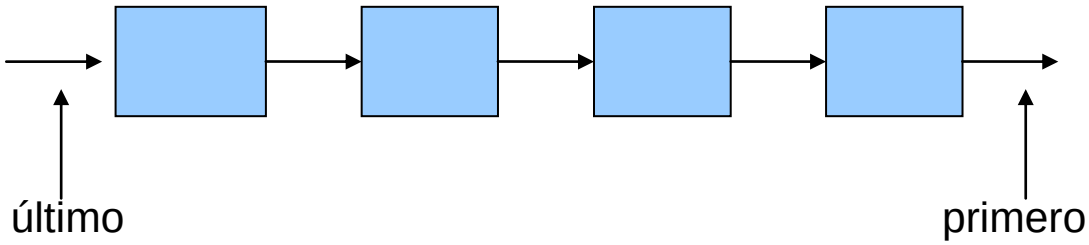


Eliminar

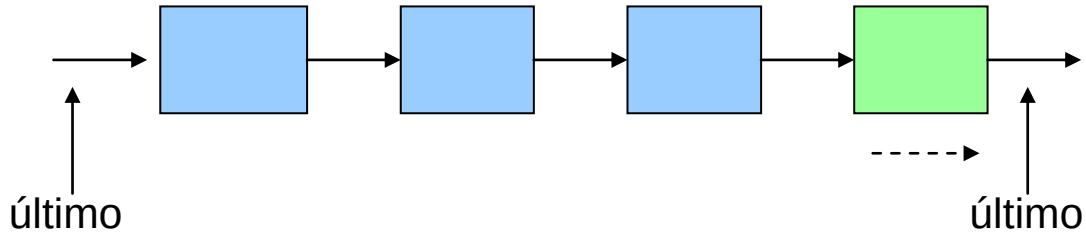
Para extraer un dato de la cola, es necesario modificar el valor del nodo primero, de manera que deje de apuntar al primer elemento de la cola. De la misma manera que con las pilas, se extrae el valor antes de que se pierda. Se

hace que el nodo primero ahora apunte al nuevo elemento (anterior de la cola).

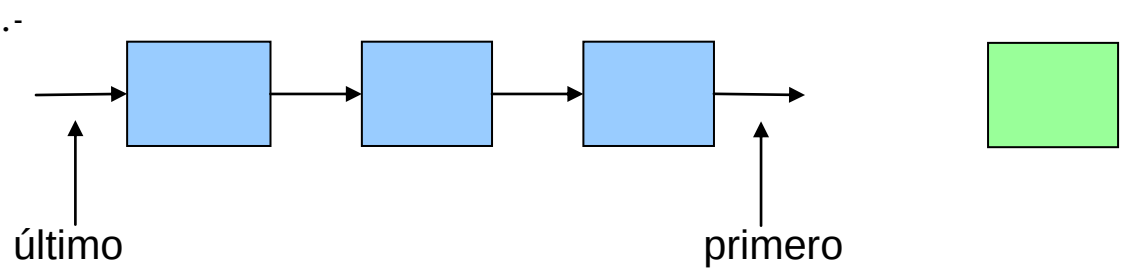
1.-



2.-



3.-



6. Árboles

Recursión

Alguna vez hemos visto ese conjunto de muñecos de madera de origen ruso (matrioshkas) en la que uno se encuentra dentro de otro. Dentro del primer muñeco, se encuentra un muñeco menor, y dentro de ese muñeco, hay otro muñeco de menor tamaño y así sucesivamente. Un método recursivo es como los muñecos rusos: se reproducen a sí mismo con versiones más y más pequeñas.

Recursión es la capacidad de un método o función de invocarse a sí mismo para resolver un problema [1]. Cada vez que un método se llama a sí mismo, se envía una parte más pequeña del problema, a fin de encontrar una solución a la misma.

En Java, un método puede llamar a otros métodos, pero además, ¡se puede llamar a sí mismo! Haciendo una llamada recursiva.

La recursión se forma de dos elementos:

- Caso base. El momento en que se detiene la recursión
- Llamada recursiva. Invocación del método a sí mismo

Cuando se alcanza el caso base, la recursión comienza un proceso llamado "backtracking" (retroceso), resolviendo las llamadas anteriores de sí mismo.

Ejemplo clásico de recursión

El calculo del factorial de un número, es el método más recurrido para explicar la recursión. Supongamos que solamente sabemos la *definició*n de factorial:

- El factorial de 0 (cero), es igual a 1 (caso base)
- El factorial de N, es igual a N multiplicado por el factorial de N-1 (llamada recursiva)

Calculemos el factorial de 3

Factorial(3) = 3 x Factorial(3-1) // por definición
Factorial(3) = 3 x Factorial(2)

Pero ahora nos damos cuenta que desconocemos el factorial de 2, así que procedemos a calcularlo

Factorial(2) = 2 x Factorial(2-1) // por definición
Factorial(2) = 2 x Factorial(1)

Seguimos sin conocer el factorial de 3, de 2, y ahora de 1. Así que pasamos a calcular el factorial de 1

Factorial(1) = 1 x Factorial(1-1) // por definición
Factorial(1) = 1 x Factorial(0)

Conocemos el factorial de 0 (cero) que es 1 (nuestro caso base), por lo que comenzamos a resolver e iniciar el *backtracking*.

Factorial(1)	=	1 x Factorial(0)	=	1 x 1 = 1
Factorial(2)	=	2 x Factorial(1)	=	2 x 1 = 2
Factorial(3)	=	3 x Factorial(2)	=	3 x 2 = 6

El factorial de 3, es igual a 6 (3 x 2 x 1).

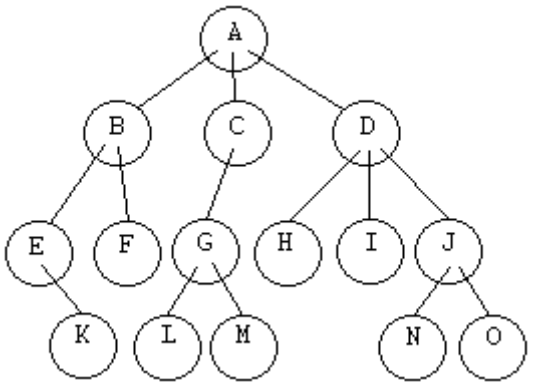
Para programar recursión, debemos tener en cuenta que siempre debe existir una forma de llegar al caso base, de lo contrario, las llamadas recursivas serían infinitas. La recursión se utiliza siempre que deseemos resolver un problema a través de él mismo, mediante fracciones más sencillas de él.

Existen diversas aplicaciones para la recursión, tales como cálculos matemáticos, trabajo con listas o árboles, etc. Ejemplos:

- Sucesión de Fibonacci
- Encontrar un nodo hoja en árboles
- Recorrer una lista
- Etc

Definición

Un árbol es una estructura no lineal en la que cada nodo puede apuntar a uno o varios nodos. También se suele dar una definición recursiva: un árbol es una estructura compuesta por un dato y varios árboles. La forma gráfica se puede apreciar como sigue:



En relación con los nodos, debemos definir algunos conceptos como:

- ***Nodo hijo***: cualquiera de los nodos apuntados por uno de los nodos del árbol. En el ejemplo, 'L' y 'M' son hijos de 'G'.
- ***Nodo padre***: nodo que contiene un puntero al nodo actual. En el ejemplo, 'A' es padre de 'B', 'C' y 'D'.

En cuanto a la posición dentro del árbol, tenemos:

- ***Nodo raíz***: nodo que no tiene padre. Este es el nodo que usaremos para acceder al árbol. En el ejemplo, ese nodo es 'A'.
- ***Nodo hoja***: nodo que no tiene hijos. En el ejemplo, hay varios: 'K', 'M', 'O', etc.
- ***Nodo rama***: aunque esta definición apenas la usaremos, estos son los nodos que no pertenecen a ninguna de las dos categorías anteriores. En el ejemplo 'B', 'G', 'D', 'J', etc.

Existen otros conceptos que definen las características del árbol, en relación a su tamaño:

- ***Orden***: es el número potencial de hijos que puede tener cada elemento del árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos, es de orden dos, si puede apuntar a tres, es de orden tres, etc.
- ***Grado***: es el número de hijos que tiene el elemento con más hijos dentro del árbol. En el árbol de ejemplo, el grado es tres, ya que tanto 'A' como 'D' tienen tres hijos y no existen elementos con más de tres hijos.
- ***Nivel***: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero, y el de sus hijos uno, así sucesivamente. En el ejemplo, el nodo 'D' tiene nivel uno, el nodo 'G' tiene nivel dos, el nodo 'N nivel tres.

- ***Altura***: la altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas. El árbol de ejemplo, tiene altura tres, la rama 'B' tiene altura dos, la rama 'G' tiene altura uno, la 'H' cero, etc.

Árboles binarios

Los árboles de orden dos son bastante especiales, de hecho les dedicaremos varios capítulos. Estos árboles se conocen también como **árboles binarios**.

Un árbol binario, requiere de una estructura de NODO que permita almacenar el dato correspondiente, además de una *referencia* al hijo izquierdo y una más al hijo derecho. El árbol será una liga al nodo raíz, a partir del cual, se podrá acceder al resto de la estructura.

```
public class NodoDeArbol  {
        int dato;
        NodoDeArbol izq;
        NodoDeArbol der;
}
```

```
public class Arbol {
        NodoDeArbol raiz;
        public Arbol() {
                raiz = null;
        }
        public Arbol(int d) {
                raiz = new NodoDeArbol(d);
        }
        public Arbol(NodoDeArbol n) {
                raiz = n;
        }
}
```

Operaciones básicas con árboles

Salvo que trabajemos con árboles especiales, como los que veremos más adelante, las inserciones serán siempre en punteros de nodos hoja o en punteros libres de nodos rama. Con estas estructuras no es tan fácil generalizar, ya que existen muchas variedades de árboles.

De nuevo tenemos casi el mismo repertorio de operaciones de las que disponíamos con las listas:

- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través del árbol.
- Recorrer el árbol completo.

Los algoritmos de inserción y borrado dependen en gran medida del tipo de árbol que estemos implementando, de modo que por ahora los pasaremos por alto y nos centraremos más en el modo de recorrer árboles.

Recorridos

El modo evidente de moverse a través de las ramas de un árbol es siguiendo los punteros (referencias), del mismo modo en que nos movíamos a través de las listas.

Esos recorridos dependen en gran medida del tipo y propósito del árbol, pero hay ciertos recorridos que usaremos frecuentemente. Se trata de aquellos recorridos que incluyen todo el árbol.

Hay tres formas de recorrer un árbol completo, y las tres se suelen implementar mediante *recursividad*. En los tres casos se sigue siempre a partir de cada nodo todas las ramas una por una.

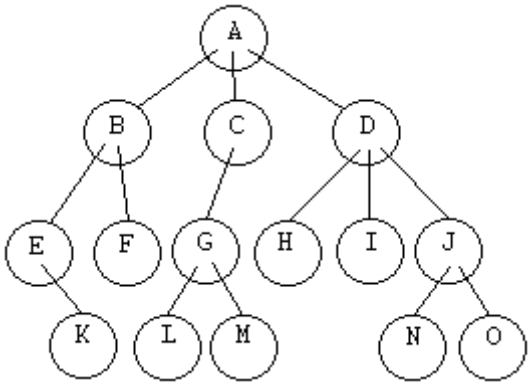
Supongamos que tenemos un árbol de orden tres, y queremos recorrerlo por completo. Partiremos del nodo raíz:

```
RecorrerArbol(raiz);
```

La función *RecorrerArbol*, aplicando recursividad, será tan sencilla como invocar de nuevo a la función *RecorrerArbol* para cada una de las ramas:

```
void RecorrerArbol (NodoDeArbol aux) {
  if (aux == NULL) return;
  RecorrerArbol(aux.der);
  RecorrerArbol(aux.izq);
}
```

Lo que diferencia los distintos métodos de recorrer el árbol no es el sistema de hacerlo, sino el momento que elegimos para procesar el valor de cada nodo con relación a los recorridos de cada una de las ramas.



Los tres tipos son:

Pre-orden:

En este tipo de recorrido, el valor del nodo se procesa antes de recorrer las ramas:

```
void PreOrden(NodoDeArbol a) {
  if (a == NULL) return;
  Procesar(dato); //método que lee el dato
  RecorrerArbol(a.izq);
  RecorrerArbol(a.centro);
  RecorrerArbol(a.der);
}
```

Si seguimos el árbol del ejemplo en pre-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

A B E K F C G L M D H I J N O

In-orden:

En este tipo de recorrido, el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última. Esto tiene más sentido en el caso de árboles binarios, y también cuando existen ORDEN-1 datos, en cuyo caso procesaremos cada dato entre el recorrido de cada dos ramas (este es el caso de los árboles-b):

```
void InOrden(NodoDeArbol a) {
  if (a == NULL) return;
  RecorrerArbol(a.izq);
  Procesar(dato); //método que lee el dato
  RecorrerArbol(a.centro);
  RecorrerArbol(a.der);
}
```

Si seguimos el árbol del ejemplo en in-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

K E B F A L G M C H D I N J O

Post-orden:

En este tipo de recorrido, el valor del nodo se procesa después de recorrer todas las ramas:

```
void PostOrden(NodoDeArbol a) {
  if (a == NULL) return;
  RecorrerArbol(a.izq);
  RecorrerArbol(a.centro);
  RecorrerArbol(a.der);
  Procesar(dato); //método que lee el dato
}
```

Si seguimos el árbol del ejemplo en post-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

K E F B L M G C H I N O J D A

Arboles ordenados

Un árbol ordenado, en general, es aquel a partir del cual se puede obtener una secuencia ordenada siguiendo uno de los recorridos posibles del árbol: inorden, preorden o postorden.

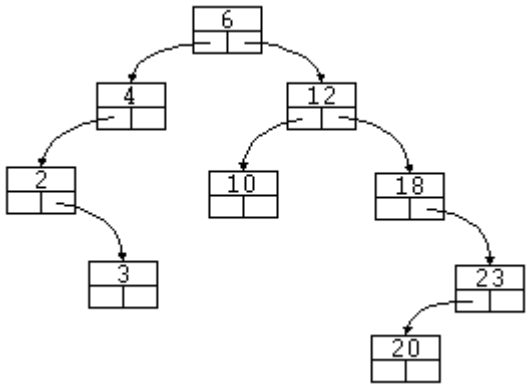
En estos árboles es importante que la secuencia se mantenga ordenada aunque se añadan o se eliminen nodos.

Existen varios tipos de árboles ordenados, que veremos a continuación:

- árboles binarios de búsqueda (ABB): son árboles de orden 2 que mantienen una secuencia ordenada si se recorren en inorden.
- árboles AVL: son árboles binarios de búsqueda equilibrados, es decir, los niveles de cada rama para cualquier nodo no difieren en más de 1.
- árboles perfectamente equilibrados: son árboles binarios de búsqueda en los que el número de nodos de cada rama para cualquier nodo no difieren en más de 1. Son por lo tanto árboles AVL también.
- árboles 2-3: son árboles de orden 3, que contienen dos claves en cada nodo y que están también equilibrados. También generan secuencias ordenadas al recorrerlos en inorden.
- árboles-B: caso general de árboles 2-3, que para un orden M, contienen M-1 claves.

Árboles binarios de búsqueda (ABB)

Se trata de árboles de orden 2 en los que se cumple que para cada nodo, el valor del nodo raíz del subárbol izquierdo es menor que el valor del nodo raíz y que el valor del nodo raíz del subárbol derecho es mayor que el valor del nodo raíz.



Operaciones en ABB.

El conjunto de operaciones que se pueden realizar sobre un ABB es similar al que se realiza sobre otras estructuras de datos, más alguna otra propia de árboles:

- Buscar un elemento.
- Insertar un elemento.
- Eliminar un elemento.
- Movimientos a través del árbol:
    - Izquierda.
    - Derecha.
    - Raíz.
- Información:
    - Comprobar si un árbol está vacío.
    - Calcular el número de nodos.
    - Comprobar si el nodo es hoja.
    - Calcular el nivel de un nodo.
    - Calcular la altura de un árbol.

Buscar un elemento.

Partiendo siempre del nodo raíz, el modo de buscar un elemento se define de forma recursiva como:

- Si el árbol está vacío, terminamos la búsqueda: el elemento no está en el árbol.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda con éxito.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.

- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

El valor de retorno de una función de búsqueda en un ABB puede ser un puntero al nodo encontrado, o NULL, si no se ha encontrado.

Insertar un elemento.

Para insertar un elemento nos basamos en el algoritmo de búsqueda. Si el elemento está en el árbol no lo insertaremos. Si no lo está, lo insertaremos a continuación del último nodo visitado. Para ello, se necesita un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

- Padre = NULL
- nodo = Raiz
- Bucle: mientras actual no sea un árbol vacío o hasta que se encuentre el elemento.
    - Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo:

        Padre=nodo, nodo=nodo->izquierdo.

    - Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho:

        Padre=nodo, nodo=nodo->derecho.

- Si nodo no es NULL, el elemento está en el árbol, por lo tanto salimos.
- Si Padre es NULL, el árbol estaba vacío, por lo tanto, el nuevo árbol sólo contendrá el nuevo elemento, que será la raíz del árbol.
- Si el elemento es menor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol izquierdo de Padre.
- Si el elemento es mayor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol derecho de Padre.

Este modo de actuar asegura que el árbol sigue siendo ABB.

Eliminar un elemento.

Para eliminar un elemento también nos basamos en el algoritmo de búsqueda. Si el elemento no está en el árbol no lo podremos borrar. Si está, hay dos casos posibles:

1. Se trata de un nodo hoja: en ese caso lo borraremos directamente.
2. Se trata de un nodo rama: en ese caso no podemos eliminarlo, puesto que perderíamos todos los elementos del árbol de que el nodo actual es padre. En su lugar buscamos el nodo más a la izquierda del subárbol derecho, o el más a la derecha del subárbol izquierdo e intercambiamos sus valores. A continuación eliminamos el nodo hoja.

Necesitamos un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

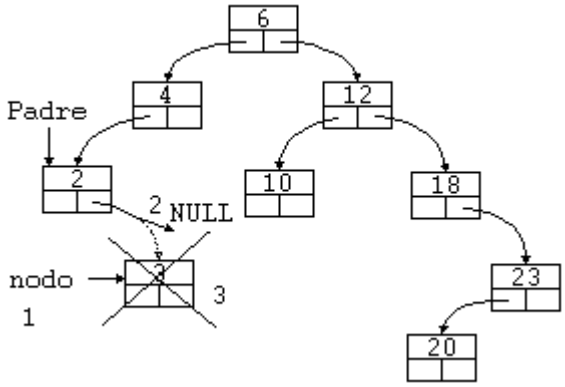
- Padre = NULL
- Si el árbol está vacío: el elemento no está en el árbol, por lo tanto salimos sin eliminar ningún elemento.
- (*) Si el valor del nodo raíz es igual que el del elemento que buscamos, estamos ante uno de los siguientes casos:
    - El nodo raíz es un nodo hoja:
        - Si 'Padre' es NULL, el nodo raíz es el único del árbol, por lo tanto el puntero al árbol debe ser NULL.
        - Si raíz es la rama derecha de 'Padre', hacemos que esa rama apunte a NULL.
        - Si raíz es la rama izquierda de 'Padre', hacemos que esa rama apunte a NULL.
        - Eliminamos el nodo, y salimos.
    - El nodo no es un nodo hoja:
        - Buscamos el 'nodo' más a la izquierda del árbol derecho de raíz o el más a la derecha del árbol izquierdo. Hay que tener en cuenta que puede que sólo exista uno de esos árboles. Al mismo tiempo, actualizamos 'Padre' para que apunte al padre de 'nodo'.
        - Intercambiamos los elementos de los nodos raíz y 'nodo'.
        - Borramos el nodo 'nodo'. Esto significa volver a (*), ya que puede suceder que 'nodo' no sea un nodo hoja.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

Ejemplos de eliminación en un ABB.

Ejemplo 1: Eliminar un nodo hoja

En el árbol de ejemplo, eliminar el nodo 3.

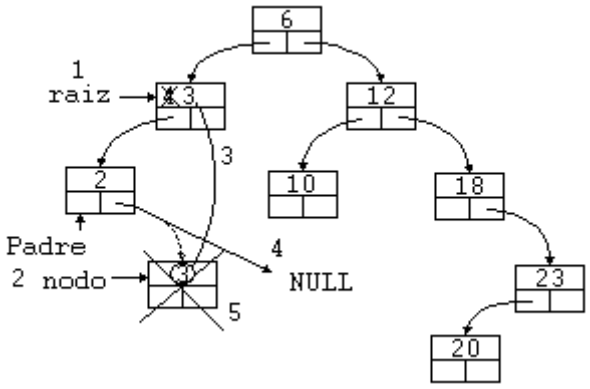
1. Localizamos el nodo a borrar, al tiempo que mantenemos un puntero a 'Padre'.
2. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
3. Borramos el 'nodo'.



Ejemplo 2: Eliminar un nodo rama con intercambio de un nodo hoja.

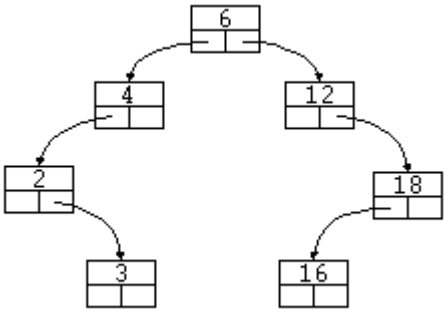
En el árbol de ejemplo, eliminar el nodo 4.

1. Localizamos el nodo a eliminar (nodo raíz).
2. Buscamos el nodo más a la derecha del árbol izquierdo de 'raíz', en este caso el 3, al tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
3. Intercambiamos los elementos 3 y 4.
4. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
5. Borramos el 'nodo'.

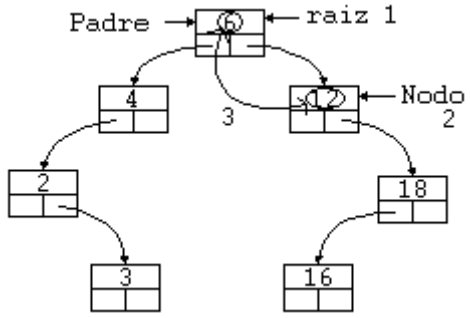


Ejemplo 3: Eliminar un nodo rama con intercambio de un nodo rama.

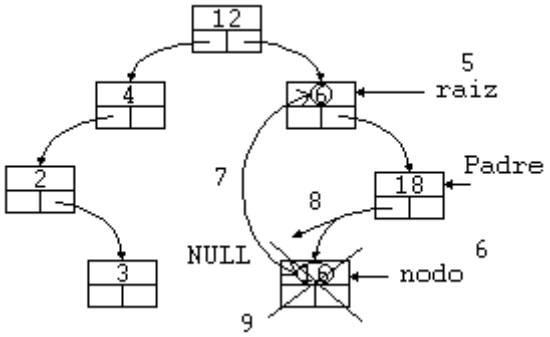
Para este ejemplo usaremos otro árbol. En éste borraremos el elemento 6.



1. Localizamos el nodo a eliminar (nodo raíz).
2. Buscamos el nodo más a la izquierda del árbol derecho de 'raíz', en este caso el 12, ya que el árbol derecho no tiene nodos a su izquierda, si optamos por la rama izquierda, estaremos en un caso análogo. Al mismo tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
3. Intercambiamos los elementos 6 y 12.
4. Ahora tenemos que repetir el bucle para el nodo 6 de nuevo, ya que no podemos eliminarlo.



5. Localizamos de nuevo el nodo a eliminar (nodo raíz).
6. Buscamos el nodo más a la izquierda del árbol derecho de 'raíz', en este caso el 16, al mismo tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
7. Intercambiamos los elementos 6 y 16.
8. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
9. Borramos el 'nodo'.



Este modo de actuar asegura que el árbol sigue siendo ABB.

Movimientos a través de un árbol

No hay mucho que contar. Nuestra estructura se referenciará siempre mediante un puntero al nodo Raíz, este puntero no debe perderse nunca.

Para movernos a través del árbol usaremos punteros auxiliares, de modo que desde cualquier puntero los movimientos posibles serán: moverse al nodo raíz

de la rama izquierda, moverse al nodo raíz de la rama derecha o moverse al nodo Raíz del árbol.

Información

Hay varios parámetros que podemos calcular o medir dentro de un árbol. Algunos de ellos nos darán idea de lo eficientemente que está organizado o el modo en que funciona.

Comprobar si un árbol está vacío.

Un árbol está vacío si su raíz es NULL.

Calcular el número de nodos.

Tenemos dos opciones para hacer esto, una es llevar siempre la cuenta de nodos en el árbol al mismo tiempo que se añaden o eliminan elementos. La otra es, sencillamente, contarlos.

Para contar los nodos podemos recurrir a cualquiera de los tres modos de recorrer el árbol: inorden, preorden o postorden, como acción sencillamente incrementamos el contador.

Comprobar si el nodo es hoja.

Esto es muy sencillo, basta con comprobar si tanto el árbol izquierdo como el derecho están vacíos. Si ambos lo están, se trata de un nodo hoja.

Calcular el nivel de un nodo.

No hay un modo directo de hacer esto, ya que no nos es posible recorrer el árbol en la dirección de la raíz. De modo que tendremos que recurrir a otra técnica para calcular la altura.

Lo que haremos es buscar el elemento del nodo de que queremos averiguar la altura. Cada vez que avancemos un nodo incrementamos la variable que contendrá la altura del nodo.

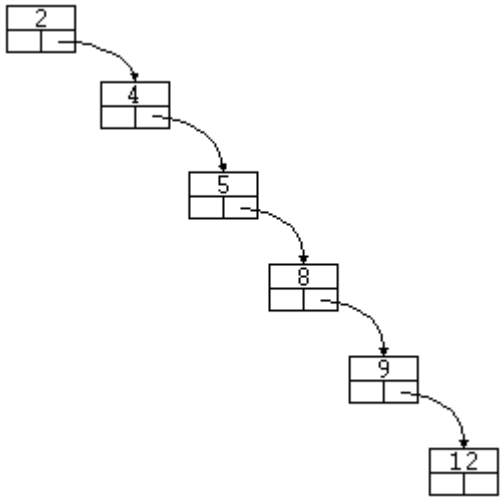
- Empezamos con el nodo raíz apuntando a Raiz, y la 'Altura' igual a cero.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda y el valor de la altura es 'Altura'.
- Incrementamos 'Altura'.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

Árboles degenerados

Los árboles binarios de búsqueda tienen un gran inconveniente. Por ejemplo, supongamos que creamos un ABB a partir de una lista de valores ordenada:

2, 4, 5, 8, 9, 12

Difícilmente podremos llamar a la estructura resultante un árbol:



Esto es lo que llamamos un árbol binario de búsqueda degenerado, y en el siguiente capítulo veremos una nueva estructura, el árbol AVL, que resuelve este problema, generando árboles de búsqueda equilibrados.

Bibliografía

[1] Nell Dale, *Object-oriented Data Structures using Java*, Jones and Bartlett Publishers, 2002.
[2] Robert Lafore, *Data Structures and Algorithms in Java*, Sams, 1998.