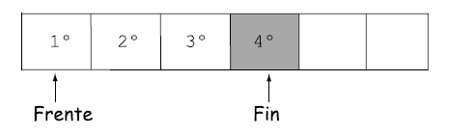
**Colas​:**

* Las colas se conocen como estructuras FIFO ( *first-in, first-out,*primero en entrar-primero en salir o bien primero en llegar-primero en ser servido (FCFS)), debido a la forma y orden de inserción y de extracción de elementos. ​
* Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades, etc.​

Una cola es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista). ​

Un elemento se inserta en la cola (parte final) de la lista y se suprime o elimina por el frente (parte inicial, frente) de la lista. ​

Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.​

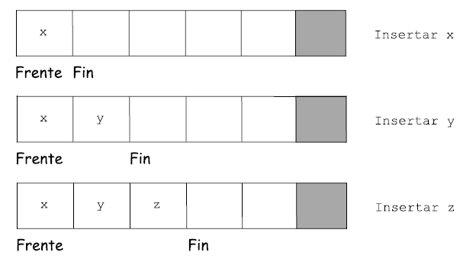


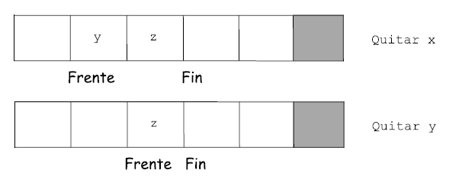
En una cola, al igual que en una pila, los datos se almacenan de un modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola.​

La forma que los lenguajes tienen para representar el TAD Cola depende de donde se almacenen los elementos: en un array, en una estructura dinámica como puede ser un Vector (contenedor de Java) o en una lista enlazada. ​

La utilización de arrays tiene el problema de que la cola no puede crecer indefinidamente, está limitada por el tamaño del array; como contrapartida, el acceso a los extremos es muy eficiente. ​

Utilizar una lista enlazada permite que el número de nodos se ajuste al de elementos de la cola; por el contrario, cada nodo necesita memoria extra para el enlace, y también hay que tener en cuenta el límite de memoria de la pila del computador.​





**COLAS IMPLEMENTADAS CON ARRAYS​:**

* Al igual que las pilas, las colas se implementan utilizando una estructura estática *(arrays)*o una estructura dinámica (listas enlazadas, Vector...). ​
* En esta sección se considera la implementación con *arrays.*​
* La declaración de una Cola ha de contener un *array*para almacenar los elementos de la cola y dos marcadores o apuntadores para mantener las posiciones frente y fin de la cola, es decir, un marcador apuntando a la posición de la cabeza de la cola y el otro al primer espacio vacío que sigue al final de la cola. ​
* Cuando un elemento se añade a la cola, se verifica si el marcador fin apunta a una posición válida, y entonces se añade el elemento a la cola y se incrementa el marcador fin en 1. ​
* Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador de cabeza, y éste se incrementa en 1.​
* La operación de poner un elemento en la cola comienza en la posición fin 0, y cada vez que se añade un nuevo elemento se incrementa fin en 1. ​
* La extracción de un elemento se hace por el extremo contrario, frente, y cada vez que se extrae un elemento avanza frente una posición.​
* El avance lineal de frente y fin tiene un grave problema, deja *huecos*por la *izquierda del array*. ​
* Llega a ocurrir que fin alcanza el índice más alto del *array,*sin que puedan añadirse nuevos elementos, y, sin embargo, hay posiciones libres a la izquierda de frente.​
* Una alternativa consiste en mantener fijo el frente de la cola al comienzo del *array*y mover todos los elementos de la cola una posición cada vez que se retira un elemento. ​
* Estos problemas quedan resueltos considerando el *array*como *circular*.​

**Declaración de la clase Cola​:**

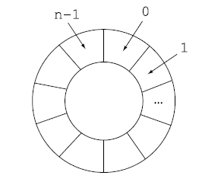
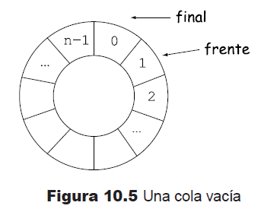
* Una cola debe tratar elementos de diferentes tipos de datos: enteros, cadenas, objetos... ​
* Por esta circunstancia, los elementos han de ser de un tipo genérico. ​
* Declarando el tipo del elemento Object y utilizando objetos y referencias se puede conseguir el mayor grado de generalidad.​
* La clase ColaLineal contiene un *array*(listaCola) cuyo máximo tamaño se determina por la constante MAXTAMQ. ​
* El tipo de los elementos queda sin especificar (TipoDeDato), para que pueda sustituirse por un tipo simple, o bien por Object. ​
* Los atributos frente y fin son los punteros de cabecera y cola (fin), respectivamente. ​
* El constructor de la clase inicializa una cola vacía y define el *array:*new *TipoDeDato*[MAXTAMQ].​
* La operación insertar toma un elemento y lo añade al final de la cola. ​
* Quitar suprime y devuelve el elemento de la cabeza de la cola. ​
* La operación frente devuelve el elemento que está en la primera posición *(frente)*de la cola, sin eliminar el elemento.​
* La operación de control colaVacia comprueba si la cola tiene elementos, ya que es necesaria esta comprobación antes de eliminar un elemento. ​
* La operación colaLlena comprueba si la cola está llena, esta comprobación se realiza antes de insertar un nuevo miembro. ​
* Si las precondiciones para insertar y quitar se violan, el programa debe generar una excepción o error.​

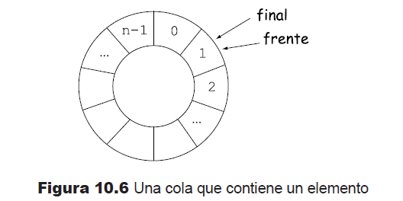
package tipoCola;  
  
public class ColaLineal {  
 private static final int MAXTAMQ = 39;  
 protected int frente;  
 protected int fin;  
 protected Object[] listaCola;  
  
 public ColaLineal() {  
 frente = 0;  
 fin = -1;  
 listaCola = new Object[MAXTAMQ];  
 }  
  
 // operaciones de modificación de la cola  
 public void insertar(Object elemento) throws Exception {  
 if (!colaLlena()) {  
 listaCola[++fin] = elemento;  
 } else  
 throw new Exception("Overflow en la cola");  
 }  
  
 public Object quitar() throws Exception {  
 if (!colaVacia()) {  
 return listaCola[frente++];  
 } else  
 throw new Exception("Cola vacia ");  
 }  
  
 // vacía la cola  
 public void borrarCola() {  
 frente = 0;  
 fin = -1;  
 }  
  
 // acceso a la cola  
 public Object frenteCola() throws Exception {  
 if (!colaVacia()) {  
 return listaCola[frente];  
 } else  
 throw new Exception("Cola vacia ");  
 }  
  
 // métodos de verificación del estado de la cola  
 public boolean colaVacia() {  
 return frente > fin;  
 }  
  
 public boolean colaLlena() {  
 return fin == MAXTAMQ - 1;  
 }  
}

Esta implementación de una cola es notablemente ineficiente, ya que se puede alcanzar la condición de cola llena existiendo elementos del array sin ocupar. ​

Esto se debe a que, al realizar la operación de quitar un elemento, avanza el frente y, por consiguiente, las posiciones anteriores quedan desocupadas, no accesibles.​

**COLA CON UN ARRAY CIRCULAR​:**

** **

****

• Creación de una cola vacía, de tal forma que fin apunte a una posición inmediatamente anterior a frente:​

frente = 0; fin = MAXTAMQ-1.​

• Comprobar si una cola está vacía:​

frente == siguiente(fin)​

• Comprobar si una cola está llena. Para diferenciar la condición de cola llena de cola vacía se sacrifica una posición del array, de tal forma que la capacidad de la cola va a ser MAXTAMQ-1. La condición de cola llena es:​

frente == siguiente(siguiente(fin))​

• Poner un elemento a la cola: si la cola no está llena, avanzar fin a la siguiente posición,​

fin = (fin + 1) % MAXTAMQ, y asignar el elemento.​

• Retirar un elemento de la cola: si la cola no está vacía, quitarlo de la posición frente y avanzar frente a la siguiente posición:(frente + 1) % MAXTAMQ.​

• Obtener el elemento primero de la cola, si la cola no está vacía, sin suprimirlo de la cola.​

package tipoCola;  
  
public class ColaCircular {  
 private static final int MAXTAMQ = 99;  
 protected int frente;  
 protected int fin;  
 protected Object[] listaCola;  
  
 // avanza una posición  
 private int siguiente(int r) {  
 return (r + 1) % MAXTAMQ;  
 }  
  
 // inicializa la cola vacía  
 public ColaCircular() {  
 frente = 0;  
 fin = MAXTAMQ - 1;  
 listaCola = new Object[MAXTAMQ];  
 }  
  
 // operaciones de modificación de la cola  
 public void insertar(Object elemento) throws Exception {  
 if (!colaLlena()) {  
 fin = siguiente(fin);  
 listaCola[fin] = elemento;  
 } else  
 throw new Exception("Overflow en la cola");  
 }  
  
 public Object quitar() throws Exception {  
 if (!colaVacia()) {  
 Object tm = listaCola[frente];  
 frente = siguiente(frente);  
 return tm;  
 } else  
 throw new Exception("Cola vacia ");  
 }  
  
 public void borrarCola() {  
 frente = 0;  
 fin = MAXTAMQ - 1;  
 }  
  
 // acceso a la cola  
 public Object frenteCola() throws Exception {  
 if (!colaVacia()) {  
 return listaCola[frente];  
 } else  
 throw new Exception("Cola vacia ");  
 }  
  
 // métodos de verificación del estado de la cola  
 public boolean colaVacia() {  
 return frente == siguiente(fin);  
 }  
  
 // comprueba si está llena  
 public boolean colaLlena() {  
 return frente == siguiente(siguiente(fin));  
 }  
}

Encontrar un numero capicúa leído del dispositivo estándar de entrada.​

El algoritmo para encontrar un número capicúa utiliza conjuntamente una Cola y una Pila.​

package capicua;  
  
import java.io.\*;  
import pilaVector.PilaVector;  
import tipoCola.ColaCircular;  
  
public class Capicua {  
  
 public static void main(String[] args) {  
 // TODO Auto-generated method stub  
 boolean capicua;  
 BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));  
 String numero;  
 PilaVector pila = new PilaVector();  
 ColaCircular q = new ColaCircular();  
 try {  
 capicua = false;  
 while (!capicua) {  
 do {  
 System.out.print("\nTeclea el número: ");  
 numero = entrada.readLine();  
 } while (!valido(numero)); // todos dígitos  
 // pone en la cola y en la pila cada dígito  
 for (int i = 0; i < numero.length(); i++) {  
 Character c;  
 c = numero.charAt(i);  
 q.insertar(c);  
 pila.insertar(c);  
 }  
 // se retira de la cola y la pila para comparar  
 do {  
 Character d;  
 d = (Character) q.quitar();  
 capicua = d.equals(pila.quitar());// compara por igualdad  
 } while (capicua && !q.colaVacia());  
 if (capicua)  
 System.out.println(numero + " es capicúa. ");  
 else {  
 System.out.print(numero + " no es capicúa, ");  
 System.out.println(" intente otro. ");  
 // se vacía la cola y la pila  
 q.borrarCola();  
 pila.limpiarPila();  
 }  
 }  
 } catch (Exception er) {  
 System.err.println("Error (excepcion) en el proceso: " + er);  
 }  
 }  
  
 private static boolean valido(String numero) {  
 boolean sw = true;  
 int i = 0;  
 while (sw && (i < numero.length())) {  
 char c;  
 c = numero.charAt(i++);  
 sw = (c >= '0' && c <= '9');  
 }  
 return sw;  
 }  
}

**Ejemplo extra de Pila:**

Escribir un programa en el que se manejen un total de n = 5 pilas: P1, P2, P3, P4 y P5.

La entrada de datos serán pares de enteros (i, j) tal que 1 ≤ aves(i) ≤ n.

De tal forma que el criterio de selección de pila será:

• Si i es positivo, debe insertarse el elemento j en la pila Pi.

• Si i es negativo, debe eliminarse el elemento j de la pila Pi.

Los datos de prueba se introducen directamente en el programa principal mediante una secuencia de llamadas a un procedimiento entrada(i, j).

Cuando termina el proceso, el programa debe escribir el contenido de las n Pilas en pantalla, identificando cada Pila.

Debe imprimir adicionalmente su nombre completo.

public class NodoPila {  
 int elemento;  
 NodoPila siguiente;  
  
 NodoPila(int x) {  
 elemento = x;  
 siguiente = null;  
 }  
}

public class Pila {  
  
 private NodoPila cima;  
  
 public Pila(){  
 cima = null;  
 }  
  
 public boolean pilaVacia() {  
 return cima == null;  
 }  
  
 public void insertar(int elemento) {  
 NodoPila nuevo;  
 nuevo = new NodoPila(elemento);  
 nuevo.siguiente = cima;  
 cima = nuevo;  
 }  
  
 public int quitar() throws Exception {  
 if (pilaVacia())  
 throw new Exception("Pila vacía, no se puede extraer.");  
 int aux = cima.elemento;  
 cima = cima.siguiente;  
 return (int) aux;  
 }  
  
 public int cimaPila() throws Exception {  
 if (pilaVacia())  
 throw new Exception("Pila vacía, no se puede leer cima.");  
 return (int) cima.elemento;  
 }  
  
  
 public void mostrar(){  
 // Crea una copia de la pila.  
 NodoPila aux = cima;  
 // Recorre la pila hasta el ultimo nodo.  
 while(aux != null){  
 System.*out*.println("|\t" + aux.elemento+ "\t|");  
 System.*out*.println("---------");  
 aux = aux.siguiente;  
 }  
 System.*out*.println("");  
 System.*out*.println("");  
 }  
}

Programa Principal

public class Main {  
 static Pila *p1* = new Pila();  
 static Pila *p2* = new Pila();  
 static Pila *p3* = new Pila();  
 static Pila *p4* = new Pila();  
 static Pila *p5* = new Pila();  
 static Pila *p* = new Pila();  
  
 public static void main(String[] args) throws Exception {  
  
 System.*out*.println("Alexander Ramírez Chiquito");  
  
 System.*out*.println("\nPila 1:");  
 *entrada*(1, 2);  
 *entrada*(1, 5);  
 *entrada*(1, 6);  
 *entrada*(-1, 6);  
 *p1*.mostrar();  
  
 System.*out*.println("");  
 System.*out*.println("\nPila 2:");  
 *entrada*(2,3);  
 *entrada*(2,5);  
 *entrada*(2,7);  
 *entrada*(2,8);  
 *entrada*(-2,5);  
 *p2*.mostrar();  
  
 System.*out*.println("");  
 System.*out*.println("\nPila 3:");  
 *entrada*(3,-10);  
 *entrada*(3,11);  
 *entrada*(3,14);  
 *entrada*(3,19);  
 *p3*.mostrar();  
  
 System.*out*.println("");  
 System.*out*.println("\nPila 4:");  
 *entrada*(4,20);  
 *entrada*(4,0);  
 *entrada*(4,43);  
 *entrada*(-4,43);  
 *entrada*(4,12);  
 *p4*.mostrar();  
  
 System.*out*.println("");  
 System.*out*.println("\nPila 5:");  
 *entrada*(5,25);  
 *entrada*(5,10);  
 *entrada*(5,51);  
 *entrada*(5,42);  
 *p5*.mostrar();  
 }  
  
 public static void entrada(int i, int j) throws Exception{  
 if (Math.*abs*(i) >= 1 && Math.*abs*(i) <= 5) {  
 switch (i) {  
 case 1:  
 *p* = *p1*;  
 break;  
 case 2:  
 *p* = *p2*;  
 break;  
 case 3:  
 *p* = *p3*;  
 break;  
 case 4:  
 *p* = *p4*;  
 break;  
 case 5:  
 *p* = *p5*;  
 break;  
 }  
 if(i < 0){  
 Pila pilaAux = new Pila();  
 do {  
 if (*p*.cimaPila() == j){  
 *p*.quitar();  
 break;  
 }  
 pilaAux.insertar(*p*.quitar());  
  
 } while (!*p*.pilaVacia());  
  
 if(!pilaAux.pilaVacia() ){  
  
 do {  
 *p*.insertar(pilaAux.quitar());  
 } while (!pilaAux.pilaVacia() );  
 }  
  
 }else{  
 *p*.insertar(j);  
 }  
 }  
 }  
}