UT3 Listas, Pilas y Colas e Introducción a Colecciones

Introducción al Análisis de Algoritmos

- Algoritmo: Conjunto claramente especificado de instrucciones que la computador seguira para resolver un problema.

Qué es el análisis de algoritmos

- Analisis de algoritmos: Determinar la cantidad de recursos (tiempo de ejecucion y espacio de memoria). Depende principalmente del tamaño de los datos que procesa.

> ⚠️ Por ejemplo, es lógico pensar que ordenar una lista de 10,000 elementos tomará más tiempo que ordenar solo 10 elementos.

- La notación "O grande" (notación de Landau) nos ayuda a identificar el término más influyente en la función que describe el tiempo de ejecución de un algoritmo.

> ⚠️ Por ejemplo, un algoritmo cuadrático se representa como O(N^2), lo que significa que su tiempo de ejecución crece cuadráticamente con el tamaño de la entrada.

- Para tamaños pequeños de entrada, las diferencias entre algoritmos pueden no ser

Funcion	Nombre
C	Constante

significativas, por lo que es común utilizar el algoritmo más simple. Sin embargo, para tamaños grandes de entrada, las diferencias se vuelven más evidentes. Los algoritmos con complejidad lineal tienden a ser más eficientes que aquellos con complejidad O(N log N), pero esta relación no siempre se cumple. Los algoritmos cuadráticos y cúbicos son poco prácticos para tamaños de entrada grandes debido a su lenta tasa de crecimiento.

Funcion	Nombre
log N	Logaritmica
log^2 N	Logaritmica al cuaadrado
N	Lineal
N log N	N log N
N^2	Cuadratica
N^3	Cubica
2^N	Exponencial

Ejemplos de tiempos de ejecución de diversos algoritmos

Problema #1

Elemento mínimo de una matriz
Dada una matriz de N elementos, determinar el menor de ellos.

- La solución propuesta consiste en mantener una variable que almacena el valor mínimo e inicializarla con el valor del primer elemento. Luego, se realiza un barrido secuencial de la matriz y se actualiza el valor mínimo de acuerdo con el valor actual del elemento en el recorrido. Este algoritmo tiene un tiempo de ejecución lineal, O(N), ya que cada elemento de la matriz debe ser examinado para determinar el mínimo.

Problema #2

Puntos más próximos en el plano
Dados N puntos en un plano (es decir, en un sistema de coordenadas x-y), encontrar la pareja de puntos más próximos.

- Una solución directa implicaría calcular la distancia entre cada par de puntos y luego encontrar el par con la distancia mínima. Sin embargo, este enfoque

sería muy costoso, ya que habría N(N-1)/2 pares de puntos a considerar. Esto resultaría en un tiempo de ejecución cuadrático, O(N^2). Sin embargo, existen algoritmos más eficientes, como el algoritmo divide y vencerás que tiene un tiempo de ejecución O(N log N), que evita calcular todas las distancias combinando estratégicamente los puntos.

Problema #3

Puntos colineales en el plano
Dados N puntos en un plano (es decir, en un sistema de coordenadas x-y), determinar si cualesquiera tres puntos forman una línea recta.

- Una solución directa implicaría enumerar todos los grupos de tres puntos y verificar si forman una línea recta. Sin embargo, esto sería computacionalmente costoso ya que habría N(N-1)(N-2)/6 combinaciones de tres puntos a considerar, lo que resultaría en un tiempo de ejecución cúbico, O(N^3). Sin embargo, existen estrategias más inteligentes que permiten resolver el problema en tiempo cuadrático, O(N^2).

Reglas generales para el cálculo de cuotas o mayúscula.

Definiciones de notaciones algoritmicas

1. **Notación O Mayúscula T(N) = O(F(N)**: Indica una cota superior en el tiempo de ejecución del algoritmo. (Menor o igual que).

> ⚠️ Imagina que estás buscando un libro en una biblioteca muy grande. La notación O mayúscula te dice cuánto tiempo te llevaría encontrar ese libro en la biblioteca en función del número de libros que tiene la biblioteca. Por ejemplo, si el tiempo de búsqueda es O(N), significa que, en el peor de los casos, necesitarías revisar cada libro de la biblioteca para encontrar el que buscas. Es una forma de medir el tiempo de ejecución de un algoritmo en función del tamaño de los datos de entrada.

2. **Omega Mayúscula T(N) = Q(F(N))**: Indica una cota inferior en el tiempo de ejecución del algoritmo. (Mayor o igual que).

> ⚠️ La notación Omega mayúscula es similar, pero te dice cuánto tiempo te tomaría encontrar el libro en la biblioteca si ya supieras en qué sección está. Es decir, te da un límite inferior en el tiempo de ejecución del algoritmo. Por ejemplo, si el tiempo de búsqueda es Ω(N), significa que tomará al menos tanto tiempo como revisar cada libro, pero podría tomar más tiempo si tienes que hacer otras cosas además de revisar los libros.

3. **Theta Mayúscula T(N) = (F(N))**: Indica que el tiempo de ejecución del algoritmo está dentro de un rango específico. (Igual).

> ⚠️ La notación Theta mayúscula combina las dos anteriores. Te dice que el tiempo de búsqueda en la biblioteca está entre un límite superior e inferior. Es decir, te da un rango en el que se espera que esté el tiempo de ejecución del algoritmo. Por ejemplo, si el tiempo de búsqueda es Θ(*N*), significa que tomará alrededor del mismo tiempo que revisar cada libro, ni más ni menos.

4. **O Minúscula T(N) = o(F(N))**: Indica que el tiempo de ejecución del algoritmo es más rápido de lo esperado. (Menor que).

> ⚠️ Esta notación es un poco diferente. Te dice que el tiempo de búsqueda en la biblioteca es menos de lo que se espera en función del tamaño de la biblioteca. Es una forma de decir que el tiempo de ejecución del algoritmo es más rápido de lo que podrías pensar. Por ejemplo, si el tiempo de búsqueda es *o*(*N*), significa que tomará menos tiempo que revisar cada libro en la biblioteca

Relacion con el analisis de algoritmos

1. **Regla básica para el tiempo de ejecución de un bucle:**
    - El tiempo de ejecución de un bucle está determinado principalmente por las instrucciones dentro del bucle, multiplicadas por el número de iteraciones del bucle.
2. **Expresiones matemáticas para el crecimiento:**
    - Se presentan diferentes expresiones matemáticas para describir el crecimiento de las funciones, como $O()$, $\Omega()$, $\Theta()$, $o()$, donde $T(N)$ representa el crecimiento de una función $F(N)$.
3. **Análisis del tiempo de ejecución para bucles anidados y secuenciales:**
    - El tiempo de ejecución de bucles anidados se calcula multiplicando el tiempo de ejecución del bucle más interno por el número de iteraciones de todos los bucles.
    - Para bucles secuenciales, el tiempo de ejecución es igual al tiempo del bucle dominante.
    - Se compara el tiempo de ejecución entre bucles anidados y bucles secuenciales, destacando que los primeros pueden ser cuadráticos mientras que los segundos son lineales.
4. **Comparación entre análisis de caso peor y promedio:**
    - Se discute la diferencia entre el análisis del caso peor y del caso promedio.
    - Se enfatiza que tener una mejor cota de caso peor no garantiza una mejor cota de caso promedio, aunque a menudo están relacionadas.
5. **Crecimiento del tiempo de ejecución:**
    - Se analiza cómo crece el tiempo de ejecución para diferentes tipos de algoritmos (cúbicos, cuadráticos, lineales, etc.) a medida que aumenta el tamaño de la entrada.
    - Se proporcionan ejemplos de cómo el tiempo de ejecución aumenta cuando se incrementa el tamaño de la entrada para algoritmos cúbicos, cuadráticos y lineales.
6. **Valor de los algoritmos cuadráticos y cúbicos:**

- Se concluye que aunque los algoritmos cuadráticos y cúbicos pueden parecer ineficientes para grandes conjuntos de datos, pueden ser útiles en ciertos contextos.
- Se destaca que la eficiencia de un algoritmo depende de varios factores, como el tamaño de la entrada y la facilidad de implementación.

El logaritmo.

Definicion y propiedades del logaritmo

- El logaritmo es una función matemática que nos dice a qué potencia necesitamos elevar un número base para obtener otro número específico.

> ⚠️ Imagina que tienes un número, digamos 1000, y quieres saber a qué potencia necesitas elevar otro número, por ejemplo 10, para obtener 1000. Esa potencia es el logaritmo de 1000 en base 10.

- En informática, a menudo usamos el logaritmo en base 2, lo que significa que nos preguntamos a qué potencia necesitamos elevar 2 para obtener un número dado.

Caracteristicas del logaritmo

- Una cosa importante sobre el logaritmo es que crece muy lentamente. Por ejemplo, el logaritmo de 1000 es solo alrededor de 3, y el logaritmo de 1 millón es aproximadamente 20. Esto significa que el logaritmo no aumenta tan rápido como los números que estamos acostumbrados a ver.
- Esta lentitud en el crecimiento del logaritmo es útil en informática porque algunos algoritmos se vuelven más eficientes cuando tienen logaritmos en su tiempo de ejecución.

Aplicaciones practicas del logaritmo

- En computación, a menudo necesitamos representar números utilizando la menor cantidad de espacio posible. El logaritmo nos ayuda a calcular cuántos bits necesitamos para representar un rango dado de números.

- Además, el logaritmo se usa en problemas como la duplicación repetida (por ejemplo, si duplicas una cantidad de dinero cada año) o la división repetida (si divides una cantidad de dinero a la mitad cada año).
- Estos conceptos son importantes en algoritmos que dividen o duplican datos repetidamente, ya que nos ayudan a entender cuánto tiempo llevará completar estas operaciones.

Relacion con el analisis de algoritmos

- En el análisis de algoritmos, el logaritmo a menudo aparece en el tiempo de ejecución de ciertos algoritmos. Por ejemplo, algunos algoritmos de búsqueda o de ordenamiento tienen tiempos de ejecución que involucran logaritmos.
- Esto es útil porque significa que estos algoritmos son más eficientes y pueden manejar tamaños de datos más grandes sin volverse demasiado lentos.
- Entender cómo el logaritmo afecta el rendimiento de un algoritmo nos ayuda a elegir el algoritmo más adecuado para una tarea dada en informática.

Listas enlazadas

Referencias en objetos

- Las variables de instancia de un objeto pueden ser arreglos o referencias a otros objetos.
- Las listas aprovechan esta característica al estar compuestas por nodos, donde cada nodo contiene una referencia al siguiente nodo y una carga de datos.

La clase nodo

- La clase `Nodo` define nodos individuales que pueden ser utilizados para construir listas enlazadas.
- **Creación y enlace de nodos**:

```
public class Nodo {
  int carga;
  Nodo prox;

  public Nodo() {
    carga = 0;
    prox = null;
  }

  public Nodo(int carga, Nodo prox) {
    this.carga = carga;
    this.prox = prox;
  }

  public String toString() {
    return carga + "";
  }
}
```

- Se crean varios nodos (`nodo1`, `nodo2`, `nodo3`) con diferentes valores de carga pero con la referencia `prox` inicializada a `null`.

```
Nodo nodo1 = new Nodo(1, null);
Nodo nodo2 = new Nodo(2, null);
Nodo nodo3 = new Nodo(3, null);
```

- **Vinculación de nodos para formar una lista**:
  - Los nodos se enlazan estableciendo las referencias `prox` de cada nodo al siguiente nodo.

```
nodo1.prox = nodo2;
nodo2.prox = nodo3;
nodo3.prox = null;
```

- **Resultado del enlace**:
  - Después de vincular los nodos, se forma una lista enlazada donde cada nodo apunta al siguiente nodo en la secuencia, con el último nodo apuntando a `null`.

```
plaintextCopy code
nodo1 --> nodo2 --> nodo3 --> null
```

Listas como coleciones

- Las listas son útiles para ensamblar muchos objetos en una sola entidad, a menudo llamada colección.
- **Pasando listas como parámetros**:
  - Para pasar una lista como parámetro a un método, solo se necesita pasar una referencia al primer nodo.
  - Por ejemplo, el método `imprimirLista` toma un solo nodo como argumento y lo imprime hasta que llega al final de la lista (indicado por la referencia a `null`).
- **Implementación de `imprimirLista()`**:
  - El método `imprimirLista()` recorre la lista empezando desde el primer nodo y avanzando a través de la referencia `prox` de cada nodo.
  - Utiliza un bucle `while` para imprimir cada nodo hasta llegar al final de la lista.
- **Recorrido de la lista**:
  - Moverse a través de una lista se llama recorrido, similar al patrón de moverse a través de los elementos de un arreglo.
  - Es común usar un iterador, como `nodo`, para referenciar cada nodo de la lista sucesivamente durante el recorrido.
- **Convención de impresión de listas**:

- Por convención, las listas se imprimen entre paréntesis con comas entre los elementos, como en (1, 2, 3).

Listas y recursion

- La recursión y las listas están estrechamente relacionadas, ya que se pueden utilizar algoritmos recursivos para manipular listas.Un ejemplo de algoritmo recursivo es imprimir el reverso de una lista.
- **Algoritmo para imprimir el reverso de una lista**:
  1. Separar la lista en dos partes: la cabeza (primer nodo) y la cola (resto de la lista).
  2. Imprimir el reverso de la cola.
  3. Imprimir la cabeza.
- **Implementación del método** `imprimirInverso()`:
  - Se define un método recursivo llamado `imprimirInverso()` que imprime el reverso de una lista.
  - Se utiliza un caso base donde si la lista es `null`, el método termina la recursión.
  - Se divide la lista en la cabeza y la cola, y luego se llama recursivamente para imprimir el reverso de la cola.
  - Finalmente, se imprime la cabeza.
- **Llamada al método** `imprimirInverso()`:
  - El método se llama de manera similar al método `imprimirLista()`, pasando el nodo inicial como argumento.
- **Demostración de terminación del método**:
  - Se plantea la pregunta de si el método `imprimirInverso()` siempre terminará (siempre alcanzará el caso base).
  - Se señala que existen algunas listas que podrían hacer que este método falle, lo que sugiere que no siempre terminará.

Pilas

Tipos de datos abstractos

- **Concepto de TAD**: Los TAD son tipos de datos que especifican un conjunto de operaciones y su semántica, pero no la implementación concreta de esas operaciones. Esto los hace abstractos, ya que no están ligados a una implementación específica.
- **Utilidad de los TAD**:
  - Simplifican la tarea de especificar algoritmos al permitirnos enfocarnos en las operaciones necesarias sin preocuparnos por cómo se implementan.
  - Dado que puede haber múltiples formas de implementar un TAD, son útiles para escribir algoritmos que puedan ser utilizados con cualquier implementación.
  - Los TADs comúnmente reconocidos, como el TAD Pila, son a menudo implementados en bibliotecas estándar, lo que permite a los programadores escribir código una vez y reutilizarlo muchas veces.
- **Distinción entre código cliente y código proveedor**:
  - Codigo cliente: Programa que usa un TAD (o persona que escribió el programa).
  - Codigo proveedor: Código que implementa un TAD (o persona que lo escribió).

El TAD Pila

- Una pila es una coleccion.
- Operaciones de las pilas:
  - **constructor:** Crea una nueva pila vacía.
  - **apilar:** Agregar un nuevo elemento al final de la pila.
  - **desapilar:** Quitar y devolver un ítem de la pila. El ítem devuelto es siempre el último que fue agregado.

    - **estaVacia:** Responder si la pila está vacía.
- Una pila es a veces llamada una estructura LIFO, del inglés “last in, first out,”, es decir “último en entrar, primero en salir”, porque el último ítem agregado es el primero en ser removido.

El objeto Stack Java

- Operaciones del TAD en la clase Stack de Java:
    - **apilar:** push
    - **desapilar:** pop
    - **estaVacia:** isEmpt
- Construir una Stack:

```
Stack pila = new Stack ();
// Inicialmente esta vacia, como podemos confirmar con el método
// un boolean:
System.out.println (pila.isEmpty ());
```

> Sólo podemos agregar objetos y no valores de tipos nativos.

Ejemplo #1:

```
// Empecemos creando e imprimiendo una lista corta.
ListaInt lista = new ListaInt();
lista.agregarAdelante (3);
lista.agregarAdelante (2);
lista.agregarAdelante (1);
lista.imprimir ();
// La salida es (1, 2, 3). Para poner un objeto de tipo Nodo en
// método push:
pila.push (lista.cabeza);
// El siguiente ciclo recorre la lista y apila todos los nodos e
for (Nodo nodo = lista.cabeza; nodo != null; nodo = nodo.prox) {
```

```
    pila.push (nodo);
}
// Podemos remover un elemento de la pila con el método pop.
Object obj = pila.pop ();
// El tipo de retorno de pop es Object! Esto es porque la imple
// sabe exactamente de qué tipo son los objetos que contiene. Cu
// objetos de tipo Nodo, son convertidos automáticamente en Obje
// obtenemos de nuevo desde el stack tenemos que castearlos de n
Nodo nodo = (Nodo) obj;
System.out.println (nodo);
// El siguiente ciclo es la forma usual de recorrer la pila, des
// elementos y frenando cuando queda vacía:
while (!pila.isEmpty ()) {
    Node nodo = (Nodo) pila.pop ();
    System.out.print (nodo + " ");
}
// La salida es 3 2 1. En otras palabras, acabamos de usar una p
// elementos de una lista de atrás para adelante.
```

Clases adaptadoras

- Para cada tipo primitivo en Java, hay un tipo de objeto preincorporado llamado una clase adaptadora.
- **Clase adaptadora:** Una de las clases de Java, como Double e Integer que provee objetos para contener tipos primitivos y métodos que operan en ellos.
- Utilidad:
  - Es posible instanciar una clase adaptadora y crear objetos que contengan valores primitivos.
  - Cada clase adaptadora contiene valores especiales (como el mínimo y el máximo valor para ese tipo), y métodos que son útiles para convertir entre tipos.

Implementando TADs

- Un objetivo fundamental de un TAD es separar los intereses del proveedor, quien implementa el TAD, del cliente, quien lo utiliza.
- **Roles del proveedor y del cliente**:
  - El proveedor se centra en garantizar que la implementación del TAD sea correcta según la especificación.
  - El cliente confía en la implementación del TAD y no se preocupa por los detalles internos.
- Cuando se utilizan clases preincorporadas de Java, los programadores pueden pensar exclusivamente como clientes, confiando en que la implementación proporcionada es correcta y eficiente.
- Al implementar un TAD, es necesario escribir código cliente para probarlo. En este caso, los programadores deben cambiar de rol entre proveedor y cliente según sea necesario, lo que requiere pensar cuidadosamente en el contexto actual.

Implementacion del TAD Pila usando arreglos

- **Variables de instancia**:
  - Se utiliza un arreglo de objetos (`Object[]`) para contener los elementos de la pila.
  - Se emplea un índice entero para llevar la cuenta del siguiente espacio disponible en el arreglo. Inicialmente, el arreglo está vacío y el índice es 0.
- **Constructor**:
  - Se define un constructor que inicializa el arreglo con un tamaño predeterminado de 128 ítems y establece el índice en 0.
- **Método `isEmpty()`**:
  - Se proporciona un método para verificar si la pila está vacía, verificando si el índice es igual a 0.
- **Métodos `push()` y `pop()`**:
  - `push`: Agrega un elemento a la pila copiando una referencia a él en el arreglo y luego incrementando el índice.

    - `pop`: Elimina y devuelve el elemento superior de la pila decrementando primero el índice y luego devolviendo el elemento correspondiente en el arreglo.
- **Pruebas de los métodos**:
    - Se menciona que se pueden probar estos métodos utilizando un código cliente que se usó previamente con la Stack preincorporada de Java, simplemente cambiando la implementación utilizada.
- **Fortalezas de utilizar un TAD**:
    - Se destaca que una de las fortalezas de utilizar un TAD es que es posible cambiar las implementaciones sin modificar el código cliente.

Redimencionando el arreglo

- La implementación inicial de la pila usando un arreglo tiene una debilidad: el tamaño del arreglo se elige arbitrariamente y puede causar excepciones si se supera.
- Una solución es permitir que el arreglo se redimensione dinámicamente cuando sea necesario.
- **Método de redimensionamiento**:
    - Se propone duplicar el tamaño del arreglo cada vez que se llene.
    - El método `push` primero verifica si el arreglo está lleno; si lo está, llama al método `redimensionar()` antes de agregar un nuevo elemento.
- **Implementación de `lleno()`**:
    - El método `lleno()` simplemente verifica si el índice ha alcanzado el tamaño del arreglo.
- **Implementación de `redimensionar()`**:
    - Crea un nuevo arreglo con el doble de tamaño.
    - Copia los elementos del arreglo original al nuevo arreglo.
    - Reemplaza el arreglo original con el nuevo.
- **Visibilidad de los métodos**:

    - Ambos métodos `lleno()` y `redimensionar()` son privados, lo que significa que solo pueden ser llamados desde dentro de la clase.
    - Esto refuerza la separación de intereses entre el código proveedor y el cliente.
- **Precondiciones y postcondiciones**:
    - Precondiciones (condiciones que deben ser verdaderas antes de que un método se ejecute).
    - Postcondiciones (condiciones que deben ser verdaderas después de que un método se complete) en la correctitud del programa.

Colas y colas de prioridad

- **Disciplinas de la cola**:
    - La regla que determina quién es atendido a continuación se conoce como disciplina de la cola.
    - La disciplina más simple es FIFO, "first-in-first-out" en inglés, donde el primero en llegar es el primero en ser atendido.
    - La disciplina más general es la de encolado con prioridad, donde a cada cliente se le asigna una prioridad y el cliente con la prioridad más alta se atiende primero, sin importar el orden de llegada.
- **TAD Cola y TAD Cola de Prioridad**:
    - Tanto el TAD Cola como el TAD Cola de Prioridad tienen el mismo conjunto de operaciones y la misma interfaz.
    - La diferencia radica en la semántica de las operaciones: una cola sigue una política FIFO, mientras que la Cola de Prioridad utiliza una política de encolado con prioridades.
- **Implementación de colas**:
    - Existen muchas formas de implementar colas, ya que son colecciones de elementos.

- Se pueden usar diversos mecanismos básicos para almacenar elementos, como arreglos y listas.
- La elección de la implementación se basa en la eficiencia y en la facilidad de implementación.

El TAD Cola

- El TAD Cola se define por las operaciones básicas: constructor, agregar, quitar y estaVacia.
- **Implementación**:
  - La implementación utiliza la clase predefinida `java.util.LinkedList` para gestionar la estructura de datos subyacente.
  - Se define una clase `Cola` que contiene una única variable de instancia, que es un objeto de tipo `LinkedList`.
  - El constructor inicializa la cola creando una nueva instancia de `LinkedList`.
  - El método `estaVacia()` verifica si la cola está vacía, utilizando el método `isEmpty()` de `LinkedList`.
  - El método `agregar(Object obj)` añade un elemento al final de la cola utilizando el método `addLast(Object obj)` de `LinkedList`.
  - El método `quitar()` elimina y devuelve el primer elemento de la cola utilizando el método `removeFirst()` de `LinkedList`.
- **Clase Cola**:

```
public class Cola {
    private LinkedList lista;

    public Cola() {
        lista = new LinkedList();
    }

    public boolean estaVacia() {
        return lista.isEmpty();
```

```
    }

    public void agregar(Object obj) {
        lista.addLast(obj);
    }

    public Object quitar() {
        return lista.removeFirst();
    }
}
```

- **Facilidad de implementación**:
  - La implementación es simple y eficiente, ya que utiliza los métodos proporcionados por `LinkedList`.

Cola enlazada

- Una Cola enlazada es una implementación del TAD cola que mantiene referencias tanto al primer como al último nodo de la cola.
- **Implementación**:
  - La clase `Cola` tiene dos variables de instancia, `primero` y `ultimo`, que representan el primer y último nodo de la cola respectivamente.
  - El constructor inicializa ambas variables en `null` para indicar una cola vacía.
  - El método `estaVacia()` verifica si la cola está vacía comprobando si `primero` es `null`.
- **Operación agregar**:
  - El método `agregar(Object obj)` crea un nuevo nodo con el objeto pasado como parámetro.
  - Si la cola no está vacía, se establece el siguiente nodo del último nodo de la cola como el nuevo nodo.
  - Se actualiza el puntero `ultimo` para que apunte al nuevo nodo.

    - Si la cola estaba vacía, tanto `primero` como `ultimo` se actualizan para apuntar al nuevo nodo.
- **Operación quitar**:
    - El método `quitar()` devuelve y elimina el primer elemento de la cola.
    - Se guarda una referencia al primer nodo en una variable local llamada `resultado`.
    - Si la cola no está vacía, se actualiza el puntero `primero` para que apunte al siguiente nodo en la cola.
    - Si después de quitar el primer nodo la cola queda vacía, se actualiza el puntero `ultimo` para que también sea `null`.
- **Complejidad temporal**:
    - Esta implementación garantiza que tanto la operación de agregar como la de quitar se realicen en tiempo constante, independientemente del tamaño de la cola.
- **Complejidad y demostración de corrección**:
    - Aunque esta implementación es más compleja que la de una cola básica, cumple con el objetivo de tener operaciones de tiempo constante.
    - Demostrar la corrección de esta implementación puede ser más desafiante debido a los casos especiales que deben manejarse.

Buffer circular

- Un buffer circular es una implementación de una cola que utiliza un arreglo circular para almacenar los elementos de manera eficiente.
- **Implementación**:
    - La clase `Cola` tiene tres variables de instancia: `arreglo` para almacenar los elementos, `primero` para indicar el primer elemento de la cola, y `prox` para apuntar al siguiente espacio disponible en el arreglo.
    - El constructor inicializa el arreglo con un tamaño predeterminado y establece tanto `primero` como `prox` en 0.

    - El método `estaVacia()` verifica si la cola está vacía comprobando si `primero` es igual a `prox`.
- **Operación agregar**:
    - El método `agregar(Object elemento)` inserta un nuevo elemento en la posición indicada por `prox` en el arreglo.
    - Se incrementa `prox` para señalar el próximo espacio disponible en el arreglo.
    - Si `prox` alcanza el final del arreglo, se reinicia a 0 para mantener la estructura circular.
- **Operación quitar**:
    - El método `quitar()` devuelve y elimina el primer elemento de la cola.
    - Se devuelve el elemento en la posición indicada por `primero` en el arreglo.
    - Se incrementa `primero` para señalar el siguiente elemento en la cola.
    - Si `primero` alcanza el final del arreglo, se reinicia a 0.
- **Verificación de si la cola está llena**:
    - Para evitar que la cola parezca vacía cuando está llena, se sacrifica un espacio en el arreglo.
    - Se utiliza la fórmula `((prox + 1) % arreglo.length == primero)` para verificar si la cola está llena.
- **Redimensionamiento del arreglo**:
    - Si el arreglo está lleno y se intenta agregar un nuevo elemento, se redimensiona el arreglo para evitar la pérdida de datos.