REPÚBLICA BOLIVARIANA DE VENEZUELA

UNIVERSIDAD JOSÉ ANTONIO PÁEZ

FACULTAD DE INGENIERÍA

ESCUELA DE

CARRERA:

INGENIERIA EN PROGRAMACION

# Documentacion de sistema para la optimización de colas y el efecto de cuello de botella

Estudiantes:

Jesus Ceballos

C.I: V-29.834.617

Jose Bastidas

C.I: V-27.894.111

Alejandro Leon

C.I: V-28.301.668

Jhorver Chirino

C.I: V-28.359.939

Prof:

Prof. Maria Garcia

# 

San Diego, agosto de 2023

**Descripción de código**

Esta documentación describe un programa de optimiza el procesos que utiliza una cola de prioridad para gestionar los procesos en función de su prioridad. El programa está implementado en lenguaje Java y se usaron arboles para la genereacion del código

Los árboles se utilizan para analizar algoritmos y estimar su tiempo de ejecución. Por ejemplo, se puede usar el árbol de decisión para analizar la eficiencia de un algoritmo de búsqueda.

**Fases del código ( arboles )**

Este código define una clase llamada `Arbol` en JavaScript que representa una estructura de árbol utilizada para almacenar procesos del sistema. Aquí tienes una descripción y explicación del código:

1. \*Importación de Módulo:\*

- La clase `Arbol` importa una clase llamada `Proceso` desde el archivo `'./proceso.mjs'`.

2. \*Constructor:\*

- El constructor de la clase `Arbol` inicializa una raíz (`root`) como un objeto `Proceso` con nombre `'inicio'` y tiempo máximo `0`.

- También crea un objeto `datos` que se utiliza para almacenar información sobre los procesos y sus tiempos máximos.

3. \*Método `empty()`:\*

- Este método verifica si el árbol está vacío. Si el siguiente proceso de la raíz es `undefined`, devuelve `true`, lo que indica que el árbol está vacío; de lo contrario, devuelve `false`.

4. \*Método `agregarNodo(proceso, destino, nodo)`:\*

- Este método agrega un nodo (proceso) al árbol. Si el árbol está vacío, simplemente agrega el nodo como siguiente de la raíz.

- Si el nodo actual (`nodo`) tiene el mismo nombre que el nodo de destino (`destino`), agrega el nodo al siguiente del nodo actual y devuelve `true`.

- Luego, realiza un recorrido recursivo por los nodos para encontrar el nodo de destino. Si lo encuentra, agrega el nodo como siguiente y devuelve `true`.

5. \*Método `insertItem(item)`:\*

- Este método agrega un elemento (`item`) al nodo raíz utilizando el método `agregarItem()` del objeto `Proceso` asociado a la raíz.

6. \*Método `getData(nodo, data)`:\*

- Este método recopila información sobre los nodos y sus tiempos máximos en el objeto `datos`.

- Comienza actualizando el objeto `datos` con el nombre del nodo actual y su tiempo máximo.

- Luego, realiza un bucle en los nodos siguientes y llama recursivamente a `getData()` en cada uno.

7. \*Método `printData()`:\*

- Este método busca el proceso con el mayor tiempo máximo en el objeto `datos` y muestra un mensaje indicando que ese proceso es el cuello de botella del sistema.

En resumen, esta clase `Arbol` parece estar diseñada para almacenar procesos en una estructura de árbol y proporciona métodos para agregar nodos, recopilar y mostrar información sobre los procesos. La lógica parece estar orientada a identificar el proceso con el tiempo máximo más alto y emitir un mensaje de recomendación. Sin embargo, el código podría requerir una revisión adicional para asegurarse de que todas las partes estén funcionando correctamente y que las propiedades y métodos utilizados sean coherentes con la definición de la clase `Proceso`.

**Fases del código ( arboles )**

Este código define una clase llamada `Cola` en JavaScript que representa una estructura de datos de cola (FIFO - "First-In-First-Out"). Aquí tienes una descripción y explicación del código:

1. \*Constructor:\*

- La clase `Cola` tiene un constructor que inicializa la cola como una lista vacía. La propiedad `list` almacena los elementos de la cola.

2. \*Método `enqueue(item)`:\*

- Este método agrega un elemento (`item`) a la cola.

- Agrega el elemento al final de la lista utilizando el método `push()`.

3. \*Método `dequeue()`:\*

- Este método elimina y devuelve el primer elemento de la cola.

- Almacena el primer elemento en una variable llamada `item`.

- Utiliza el método `shift()` para eliminar el primer elemento de la lista.

- Retorna el valor almacenado en `item`.

4. \*Método `empty()`:\*

- Este método verifica si la cola está vacía.

- Si la longitud de la lista es mayor que 0, devuelve `false`, lo que significa que la cola no está vacía; de lo contrario, devuelve `true`.

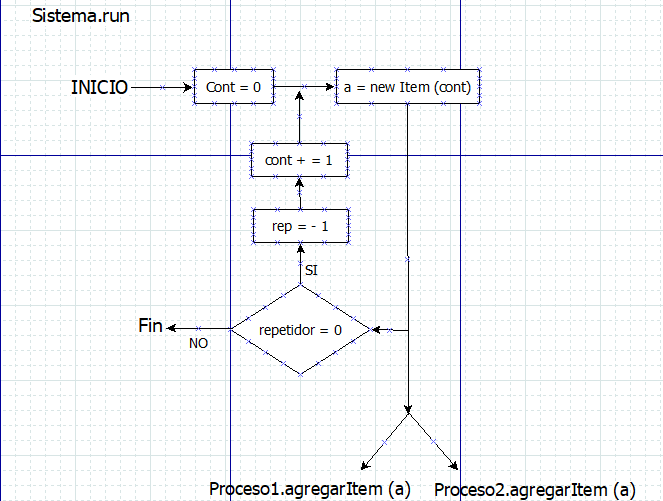
5. \*Método `getSize()`:\*

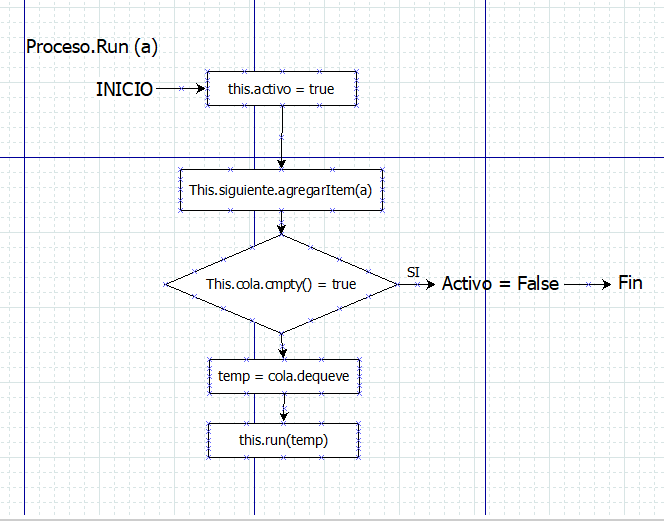
- Este método devuelve la cantidad de elementos en la cola.

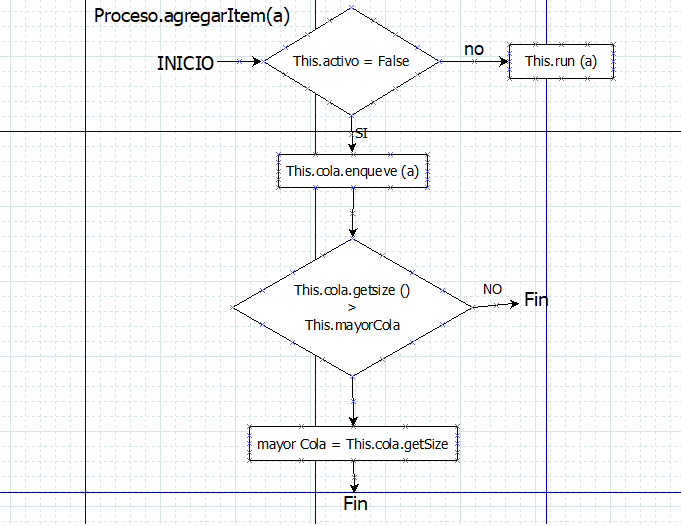
- Retorna la longitud de la lista.

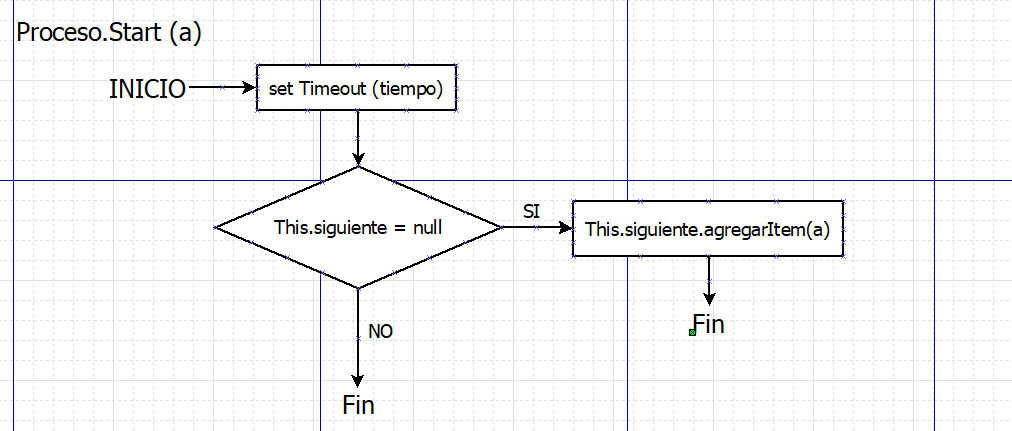
En resumen, la clase `Cola` está diseñada para implementar la funcionalidad de una cola básica. Los métodos `enqueue()` y `dequeue()` permiten agregar y eliminar elementos respetando el principio de "primero en entrar, primero en salir". Los métodos `empty()` y `getSize()` permiten verificar si la cola está vacía y obtener el número de elementos en la cola, respectivamente. Esta estructura de datos de cola es útil para manejar elementos en el orden en que fueron agregados, como podría ser necesario en situaciones como planificación de tareas o procesos en un sistema.

**Diagrama de flujo**









**Diagrama de flujo**