ESTRUCTURA DE DATOS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN Y LA DECISIÓN

MONITOR: ALEJANDRO SALAZAR MEJÍA

alsalazarm@unal.edu.co

2023-2S

TALLER 8: PILAS Y COLAS

Este taller está diseñado para afianzar tu comprensión sobre las pilas y colas, entender las políticas FIFO y FILO en arreglos secuenciales, fortalecer tu habilidad para resolver algoritmos sobre estructuras de datos y usar los beneficios que traen pilas y colas para solucionar problemas algorítmicos de forma eficiente. Varios de estos ejercicios están inspirados en problemas de entrevistas técnicas y competencias de programación.

Los talleres son herramientas de estudio, y no tienen asignada una ponderación dentro de las evaluaciones del curso. Se recomienda desarrollar cada problema mediante la presentación de un seudocódigo, y en el caso de diseño de clase mediante diagramas de clase. Sin embargo, si el estudiante puede, solucionar cada problema empleando un lenguaje de programación de alto nivel.

Para resolver los ejercicios usando seudocódigo, asuma que ya tiene la implementación de la estructura de datos con los métodos y características principales que se ven en clase.

1. La pizzería Santoro de la universidad ofrece lasaña de pollo y lasaña de carne a la hora del almuerzo, denominados con los números 0 y 1 respectivamente. Todos los estudiantes hacen cola. Cada estudiante prefiere uno de los dos: prefiere lasaña de pollo o prefiere lasaña de carne.

El número de lasañas en la pizzería es igual al número de estudiantes. Las lasañas se colocan en una pila. En cada paso:

- Si el estudiante que está al principio de la cola prefiere la lasaña de la parte superior de la pila, la tomará y abandonará la cola.
- Si no, lo dejará y se irá al final de la cola.

Esto continúa hasta que ninguno de los estudiantes de la cola quiera coger la lasaña de arriba y, por tanto, no puedan comer.

Cree un algoritmo que reciba dos arreglos de enteros *'estudiantes'* y *'lasaña'* donde *lasaña[i]* es el tipo de la i-ésima lasaña en la pila (i = 0 es el principio de la pila) y estudiantes[j] es la preferencia del j-ésimo estudiante en la cola inicial (j = 0 es la cabeza de la cola). Devuelva el número de alumnos que no pueden comer.

2. Dado un *string*, determine si es **palíndromo usando pilas**.

3. Dado un *string* que contiene solo paréntesis, determine si dicha secuencia de paréntesis está bien formada (balanceada) o no. Por ejemplo, la secuencia '(( ) ( )) ( )' está bien formada, pero ') (' está mal, '( ))' está mal, etc.

4. Dada una pila, cree un algoritmo que invierta la pila **utilizando una cola**. Es decir, si por ejemplo la pila es 1 ⟶ 2 ⟶ 3 ⟶ 4, su algoritmo debe devolver la pila 4 ⟶ 3 ⟶ 2 ⟶ 1.

5. Responda: ¿por qué para hacer una implementación adecuada de las pilas es suficiente usar listas enlazadas **simples**, pero para colas es necesario listas **doblemente** enlazadas? Cuando digo "implementación adecuada" me refiero a que las operaciones características son O(1).

6. Dada una frase con varias palabras, imprima cada palabra al revés individualmente, es decir, no invierta toda la frase como si fuera un único *string*. Así, si la entrada es "Hola Mundo!" su algoritmo debe retornar "aloH !odnuM".

7. Consulte sobre la Notación Polaca Inversa. Una vez se haya familiarizado con esta, cree un algoritmo que reciba una expresión aritmética en notación polaca inversa y devuelva su resultado. Por ejemplo, si se recibe "100 200 + 2 / 5 * 7 +", al evaluar esta expresión su resultado es 757.

(***hint:*** utilice pilas!)

8. Consulte cómo es posible implementa la estructura de datos **Cola** utilizando la estructura de datos **Pila**. Con esto me refiero a que de la misma forma como se implementaron las colas usando listas doblemente enlazadas, es posible implementar las colas usando pilas. Es posible obtener las mismas operaciones características *queue* y *enqueue* utilizando *push* y *pop* de pilas.

9. Diseñe el objeto **ZigZagIterator** con las siguientes características. Para entender cómo va a funcionar, vea el ejemplo primero.

- Implemente la clase `ZigZagIterator`.
  `ZigZagIterator(Array array1, Array array2):` Inicializa el objeto con dos arreglos como parámetros.
- `int next():` regresa el siguiente elemento del ZigZagIterator basado en la política ZigZag (revisar el ejemplo).
- `boolean hasNext():` *True* si aún quedan elementos en el Iterador, *False* si está vacío.

Considere el siguiente ejemplo (pseudocódigo, no es código de Java real):

```
Int[] v1 = {1, 2, 3};
Int[] v2 = {4, 5, 6, 7, 8};

ZigZagIterator i = new ZigZagIterator(v1, v2);

while(i.hasNext()) {
      System.out.println(i.next());
}
```

El programa anterior debería imprimir

```
1
4
2
5
3
6
7
8
```

Es decir, imprime un elemento de v1, después uno de v2, después el siguiente de v1, el siguiente de v2, y así sucesivamente.

(***hint:*** utilice una **cola de arreglos**, es decir, una cola cuyos elementos son arreglos)

10. Diseñe un algoritmo que acepte una secuencia de enteros y obtenga el producto de los últimos *k* enteros de la secuencia. Más específicamente, implemente la clase ProductoDeNumeros con las siguientes características:

- `ProductoDeNumeros()`: Inicializa el objeto con una secuencia vacía.
- `void agregar(int num)`: Añade el número entero num a la secuencia.
- `int getProducto(int k)`: Devuelve el producto de los últimos k números de la lista actual. Puede asumir que siempre la lista actual tiene al menos k números.

Considere el siguiente ejemplo (pseudocódigo, no es código de Java real ProductoDeNumeros):

```
ProductoDeNumeros producto = new ProductOfNumbers();
producto.agregar(3);       // [3]
producto.agregar(0);       // [3, 0]
producto.agregar(2);       // [3, 0, 2]
producto.agregar(5);       // [3, 0, 2, 5]
producto.agregar(4);       // [3, 0, 2, 5, 4]

producto.getProducto(2);   // 20 ya que 5 * 4 = 20 son los dos
                           // últimos números
producto.getProducto(4);  // 0 ya que 0 * 2 * 5 * 4 = 0 son los
                          // dos últimos número
```