Práctica 0 - Repaso

Fundamentos El objetivo de los siguientes ejercicios es repasar los fundamentos de la programación, a la vez que se aprende la sintaxis del lenguaje.

1. (⋆) Realizar un intercambio de variables `a` y `b` de tipo `int` sin utilizar variables auxiliares, y comenzando con la instrucción `a = a - b`. ¿Qué sucede en los valores límites del tipo `int`?
2. Realizar un intercambio de variables `a` y `b` de tipo `boolean` sin utilizar variables auxiliares.
3. Realizar un intercambio de variables `a` y `b` de tipo `int` sin utilizar variables auxiliares, y comenzando con la instrucción `x = x ^ y`. Esta operación *bitwise* aplica la compuerta `XOR` bit a bit.
4. Realizar un intercambio de variables `a` y `b` de tipo `String` sin utilizar variables auxiliares.
5. Realizar un intercambio de n variables para la suma de enteros.
6. Dadas tres variables $a, b, c \in \mathbb{Z}$, utilice solo sentencias *short-if* para calcular el mayor estricto.
7. Dadas tres variables $a, b, c \in \mathbb{Z}$, utilice solo sentencias *short-if* para calcular la mediana.
8. Sean las variables $a, b, c, d \in \mathbb{R}$, que representan los posibles lados de un cuadrado, crear un programa que guarde en una variable `true` solo si pueden representar los lados de un cuadrado.
9. Sean las variables $a, b, c \in \mathbb{R}$, que representan los posibles lados de un triángulo, crear una programa que guarde en una variable `true` solo si pueden representar los lados de un triángulo.
10. Sea la variable $a \in \mathbb{N}$, crear un programa que invierta sus dígitos.
11. Sea la variable $a \in \mathbb{N}$, crear un programa que invierta sus dígitos utilizando únicamente operaciones matemáticas
12. Dadas dos variables $a, b \in \mathbb{Z}$, Java calcula la división entera como `a / b` reemplazando la división usual. Escriba en una sola instrucción una división usual a partir de estas dos variables.
13. Dadas dos variables $a, b \in \mathbb{R}$, escriba en una sola instrucción una división entera a partir de estas dos variables.
14. Dado que no se puede convertir un valor de tipo `boolean` a `byte` mediante casteo, escribir en una sola instrucción la lógica necesaria para poder hacerlo.
15. La operación módulo es una operación que da un resto negativo si el divisor es un entero negativo. Escribir un ejemplo de una sola instrucción que permita obtener el resto positivo.
16. Demostrar por qué la operación `NAND` es un conjunto completo. De forma análoga, hacerlo para `NOR`.
17. Demostrar por qué las operaciones `AND` y `XOR` forman un conjunto completo.
18. `Integer.MAX_VALUE` representa el entero más grande de Java. `Integer.MIN_VALUE` representa a su vez el entero más chico. Utilizar estas expresiones para generar una resta a partir de sumas.

19. Dadas dos variables $a, b \in \mathbb{Z}$, calcular la suma de estas variables sin utilizar los operadores `+` ni `-`.
20. Calcule si dos números son iguales, sin utilizar `==`.
21. Calcule si un número es par sin usar módulo, ciclos, ni divisiones.
22. Utilizar `System.currentTimeMillis()` para verificar si sumar números grandes tarda más que sumar números chicos.
23. Sea ε un posible margen de error de comparar valores de tipo `float` que deberían ser iguales, calcular una cota para este error con una buena presición (no hace falta que sea excelente).
24. Verifique si dos números a y b son iguales sin usar `==` (y sin usar `!(a != b)`).
25. Escribir una expresión booleana que siempre de `false`, pero que si Java leyera las expresiones de derecha a izquierda, siempre de `true`.
26. Escribir una expresión de una sola instrucción, en base a dos variables a, b previamente creadas, tal que si $a = 1$ devuelva $8b$, pero si no, devuelva $7a$, sin usar `if` ni `short if`.
27. ¿Cuál es la fecha que es elemento neutro de sumar dos fechas? ¿Y para su representación en milisegundos?
28. Escribir un programa que permita sumar los dígitos de un número.
29. Escribir un programa que permita calcular el dígito del medio de un número asumiendo que tiene una cantidad impar de dígitos.
30. Calcular el bit de paridad de un valor de tipo `int` sin utilizar bitwise.

Arreglos y matrices Los arreglos poseen la propiedad de acceso aleatorio que será fundamental en el desarrollo de la materia. Por otro lado, las matrices serán importantes para detectar propiedades sobre grafos. El objetivo de estos ejercicios es el repaso de estos conceptos mientras que aprendemos su sintaxis en Java.

1. Dado un arreglo que contiene booleanos, calcular la paridad.
2. Sea A un array de $a \in \mathbb{Z}$, dado un rango $[n, m]$. Si el rango excede los índices del array, suponga que el array se extiende infinitamente con copias de el. Por ejemplo, en $[1, 2, 3, 4, 5]$, el elemento en la posición -2 es 4.
3. Cree una función que reciba un array de caracteres, y devuelva el caracter mas a la izquierda en el alfabeto.
4. Cree una función que reciba un array de caracteres, y devuelva el caracter con menos ocurrencias.

Práctica 1 - Pilas

Definición e implementación En los siguientes ejercicios, se espera la creación de la interfaz (si es necesaria) y la clase correspondiente para la estructura modificada. Las implementaciones pueden ser estáticas o dinámicas, según se considere más apropiado. En la definición, se espera que se detallen las *precondiciones*, *postcondiciones* y la *estrategia* utilizada. En la implementación, se espera un manejo adecuado de errores. Además, se recomienda incluir casos de prueba que demuestren el correcto funcionamiento de la estructura modificada.

1. (⋆) Modificar el TDA Stack para que tenga una capacidad máxima.
2. (⋆) Modificar el TDA Stack para que cada elemento que llega se apile 2 veces, manteniendo el comportamiento usual para el desapilar.
3. (⋆⋆) Modificar el TDA Stack para que cada elemento que llega se apile n veces, y cada vez que se llame al método que desapila, se desapilen m elementos. ¿Cuáles serían las precondiciones? ¿Cuál sería la restricción necesaria para que nunca intentar desapilar elementos de más?
4. (⋆⋆⋆) Modificar el TDA Stack para lograr una implementación estática utilizando nodos.
5. (⋆⋆⋆) Modificar el TDA Stack para lograr una implementación dinámica utilizando arreglos.
6. (⋆⋆⋆⋆) † Modificar el TDA Stack para poder almacenar elementos de tipo *char*. Luego, crear un método por fuera de la estructura que reciba un texto como argumento y chequee que el texto respeta el uso adecuado de paréntesis. Luego, crear un segundo método análogo para chequear paréntesis, corchetes y llaves de forma simultánea.

Algoritmos Crear métodos que reciban una instancia de Stack por parámetro, tales que podamos:

7. (⋆) Contar la cantidad de elementos.
8. (⋆) Imprimir en pantalla los elementos que son pares.
9. (⋆⋆) † Invertir la instancia.
10. (⋆⋆) † Copiar la instancia.
11. (⋆⋆) Intercambiar el elemento del tope con el elemento del fondo.
12. (⋆⋆) Decidir si tenemos una cantidad par de elementos sin usar variables auxiliares.
13. (⋆⋆) † Sacar el elemento del medio.
14. (⋆⋆⋆) † Dividir la instancia en tercios (usando únicamente otras instancias de Stack), e intercambiar el tercio superior con el tercio inferior manteniendo el orden.
15. (⋆⋆) † Eliminar elementos repetidos.
16. (⋆⋆) Ordenar por inserción.

17. (★★) Ordenar por selección.
18. (★★★) Ordenar por burbujeo.
19. (★) Buscar un elemento si la instancia se encuentra desordenada.
20. (★) Buscar un elemento si la instancia se encuentra ordenada.
21. (★★) Verificar si es capicúa.
22. (★★★★) Invertir la instancia sin usar nada auxiliar.

    Hint

    Utilizar recursividad.

Complejidad computacional

23. (★) Calcular la complejidad computacional del ordenamiento por inserción, luego del ordenamiento por selección y compararlos.
24. (★★) † Analizar si tiene sentido utilizar búsqueda binaria sobre la estructura.

Práctica 2 - Colas

Definición e implementación En los siguientes ejercicios, se espera la creación de la interfaz (si es necesaria) y la clase correspondiente para la estructura modificada. Las implementaciones pueden ser estáticas o dinámicas, según se considere más apropiado. En la definición, se espera que se detallen las *precondiciones*, *postcondiciones* y la *estrategia* utilizada. En la implementación, se espera un manejo adecuado de errores. Además, se recomienda incluir casos de prueba que demuestren el correcto funcionamiento de la estructura modificada.

1. (★) Modificar el TDA Queue para que la implementación dinámica tenga como primer nodo el elemento del final.
2. (★★) Modificar el TDA Queue para que los elementos se acolen de forma ordenada.
3. (★★) Modificar el TDA Queue para que no admita un elemento múltiplo de otro elemento ya presente en la estructura.

Algoritmos Crear métodos que reciban una instancia de Queue por parámetro, tales que podamos:

4. (★) Sumar los elementos.
5. (★) Imprimir en pantalla los elementos que son primos.
6. (★★) † Invertir la instancia.
7. (★★) † Copiar la instancia.
8. (★★) † Decidir si tenemos una cantidad múltiplo de n sin usar variables auxiliares para contar los elementos.
9. (★★) Ordenar la estructura para conseguir la mediana.
10. (★★★) † Encontrar el elemento que más se repite.
11. (★★) † Eliminar elementos repetidos.
12. (★★) Ordenar por inserción.
13. (★★) Ordenar por selección.
14. (★★★) Ordenar por burbujeo.
15. (★★) Verificar si es capicúa sin usar instancias de Stack auxiliares.
16. (★) Buscar un elemento si la instancia se encuentra desordenada.
17. (★) Buscar un elemento si la instancia se encuentra ordenada.
18. (★★★) Invertir la instancia sin utilizar una instancia de Stack auxiliar.

Combinación de estructuras

19. (⋆) Crear un *mapa* que transforme una instancia de Queue en una instancia de Stack.
20. (⋆) Crear un *mapa* que transforme una instancia de Stack en una instancia de Queue.
21. (⋆) Invertir una instancia de Queue, utilizando una instancia de Stack auxiliar.
22. (⋆) Verificar si una instancia de Stack es capicúa, utilizando una instancia de Queue auxiliar.

Complejidad computacional

23. (⋆) Calcular la complejidad computacional del método que encuentra el elemento más repetido.
24. (⋆⋆) † Analizar si tiene sentido utilizar el algoritmo *merge sort* por fuera de la estructura.