Práctica 2. Uso de TADs lineales

2.1 Pilas

Ejercicio 1: ¿Qué valores serán devueltos durante las siguientes operaciones que comienzan sobre una pila vacía? `push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop()`. Valida la respuesta implementando un programa que haga estas operaciones haciendo uso de la implementación ArrayStack.

Ejercicio 2: Modificar la implementación de ArrayStack de forma que la capacidad de la pila sea limitada a *maxlen* elementos, donde *maxlen* es un parámetro opcional del constructor (que por defecto es None). Si se llama a `push()` cuando la pila está llena, se debe lanzar una excepción *Full* (definida de forma similar a *Empty*). Prueba el correcto funcionamiento del programa con las operaciones del Ejercicio 1.

Ejercicio 3: En el ejercicio anterior, asumimos que inicialmente la pila está vacía. Rehacer el ejercicio para crear inicialmente una pila que ya tiene reservados una cantidad de espacio igual a la capacidad máxima de la pila. Prueba el correcto funcionamiento del programa con las operaciones del Ejercicio 1.

Ejercicio 4: Las pilas son usadas a menudo para proporcionar la acción "undo" en las aplicaciones como navegador web o editor de textos. Aunque esta acción se puede dar de forma ilimitada, muchas aplicaciones limitan la acción con una pila de capacidad fija. Cuando la pila es invocada y la capacidad está llena, en lugar de lanzar una excepción como en el ejercicio 2, se puede aceptar e introducir el elemento en la pila, eliminando el elemento más antiguo de la pila. Proporcionar una implementación de esta abstracción usando un array circular con la capacidad apropiada. La clase debe tener una especificación similar a la del ejercicio anterior, pero con la modificación indicada. Prueba el correcto funcionamiento del programa con las operaciones del Ejercicio 1, y añade más operaciones para chequear la sobreescritura cuando la pila se llena.

Ejercicio 5: Haz uso de la implementación básica de ArrayStack para chequear el correcto balanceo de paréntesis, simulando la evaluación de una expresión matemática. De esta forma, nuestro programa será capaz de analizar las siguientes expresiones informando del resultado:

- (a+b)+(c+d) -> correcto
- a+b)+c -> incorrecto
- ((a+b+c)+d)+1 -> correcto
- ((a+b+c)+d+1 -> incorrecto
- (a + (b*c) – (5*(a+b+c))) -> correcto
- (a + (b*c) – 5*(a+b+c))) -> incorrecto

2.2. Colas

Ejercicio 6: ¿Qué valores serán devueltos durante las siguientes operaciones sobre una cola vacía? `enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue()`. Valida la respuesta implementando un programa que haga estas operaciones haciendo uso de la implementación ArrayQueue.

Ejercicio 7: Consideremos un conjunto de niños jugando al juego de Patata Caliente. En este juego, los niños forman un círculo y se pasan un *item* de uno a otro de forma consecutiva. En un cierto momento, la actividad se para y el niño que tiene el *item* es eliminado. El juego continúa hasta que solo queda un niño. Usa una cola para implementar un programa que simule este juego. Para ello, usa un bucle para alternar el paso del *item* de un niño a otro. En cada iteración se pedirá una pulsación de teclado, haciendo uso de la función *input*. Cuando se lea la entrada 'p', el juego parará y el niño que tenga el *item* será eliminado.

Ejercicio 8: Implementar una cola usando listas enlazadas. Para ello, usa como partida la implementación ArrayQueue y sustituye la estructura de datos interna usada, basada en una lista Python, por una lista enlazada como la vista en clase. Prueba el correcto funcionamiento del programa con las operaciones del Ejercicio 6.

2.3. Listas en Python

Ejercicio 9: Escribe una función *ordenada* que tome una lista como parámetro y devuelva *True* si la lista está ordenada en orden ascendente y devuelva *False* en caso contrario.
Por ejemplo, `ordenada([1, 2, 3])` retorna *True* y `ordenada(['b', 'a'])` retorna *False*.

Ejercicio 10: Implementa una función que sea capaz de encontrar un número que se repite 5 veces de forma consecutiva en una lista de tamaño mayor o igual a 6.

Ejercicio 11: Para comprobar si una palabra está en una lista se puede utilizar el operador *in*. Escribe un pequeño programa que sea capaz de buscar un número dado dentro de una lista. Tanto el número como el conjunto de números de la lista se deben pedir por teclado.

Ejercicio 12: El operador *in* podría ser muy costoso si estuviera implementado para buscar en detalle cada elemento dentro de una lista. Si imaginamos un conjunto de palabras guardadas en una lista, éste podría realizar un barrido por cada una de las palabras, de principio a fin, para encontrar una coincidencia. Este proceso se puede acelerar si consideramos que las palabras están guardadas en orden alfabético, pudiendo realizar una búsqueda de bisección (también conocida como búsqueda binaria), que es similar a lo que haces cuando buscas una palabra en el diccionario. Comenzamos por el centro y comprobamos si la palabra que buscamos está antes o después del centro. Si está antes, se busca solo en la primera mitad, si está después se busca en la otra mitad de la lista. Con esto reduciremos el tiempo de búsqueda. Escribir una función llamada "bisect" que tome una lista ordenada y una palabra como objetivo, y nos devuelva el índice en el que se encuentra en la lista, en caso de no aparecer en la lista devuelve "No se encontró la palabra".

Ejercicio 13: Una operación útil en bases de datos es la unión natural. Si vemos una base de datos como una colección ordenada de pares de objetos, entonces la unión natural de las bases de datos *A* y *B* es la lista de las tripletas ordenadas (*x,y,z*) de forma que el par (*x,y*) está en A y el par (*y,z*) está en B. Describir un algoritmo que realice esta operación para una lista A de n pares y una lista B de m pares. Los pares de elementos se deben guardar como tuplas.