Ficha 28

Estructuras Lineales: Pilas y Colas

1.] Tipos nativos y tipos abstractos.

En distintas Fichas de estudio anteriores hemos introducido el concepto de *estructura de datos* como una colección de muchos valores almacenados al mismo tiempo en una misma variable. Hemos visto que los distintos lenguajes de programación proveen ya listos para usar distintos tipos de estructuras de datos, y que en Python algunos de esos tipos son las tuplas, los rangos, las cadenas de caracteres, las listas (a través de las que se representan arreglos) y los registros. Todos estos tipos compuestos permiten organizar datos y resultados en formas diversas, brindando además distintas maneras de acceder a los datos individuales contenidos en cada estructura creada [1].

Sin embargo, en muchos problemas un programador se enfrenta a situaciones en las que percibe que necesita usar o aplicar formas diferentes de organizar y/o acceder a sus datos. Un ejemplo que hemos analizado nos llevó a la necesidad de los arreglos bidimensionales: si cada dato del dominio del problema se identifica o define con dos números (tipo de gasto y mes, número de estudiante y número de prueba, y en definitiva, un número de fila y otro de columna) puede seguir empleando un arreglo unidimensional… pero está claro que un arreglo bidimensional permitirá aprovechar mejor la forma de identificación natural de esos datos. Si el programador desconocía la forma de trabajar con arreglos bidimensionales, una situación práctica como esta lo llevaría a explorar el lenguaje en busca de variantes y más temprano que tarde encontraría la forma de usar arreglos con un índice adicional (o más).

Si el programador descubre que necesita nuevos tipos de estructuras de datos y el lenguaje que aplica dispone de esos tipos ya listos para usar, estamos en presencia de *estructuras de datos nativas* del lenguaje. Está claro que en Python las *secuencias* de cualquier tipo (estructuras de datos subindicadas, mutables o inmutables, como las *tuplas*, los *rangos*, las *cadenas* y las *listas*), así como los *registros* y los *diccionarios*, representan *estructuras de datos nativas* [1].

Pero en muchos casos, la estructura de datos que requeriría el programador no está disponible directamente en su lenguaje de trabajo, y el programador debe entonces *implementarla* combinando elementos que sí estén disponibles en su lenguaje. Un ejemplo: los registros como concepto general de variable que puede contener campos de tipos diferentes son estructuras nativas en Python, pero si el programador necesita un registro que *específicamente* permita representar libros en un sistema de información para una biblioteca, deberá producir ese nuevo tipo *Libro*, en base al uso combinado del tipo registro nativo (empleando la construcción *class*) y declarando campos con identificadores que estrictamente describan características de un libro en ese contexto. El nuevo tipo *Libro* no existía en Python: fue introducido por el programador combinando tipos y mecanismos que ***sí*** existían.

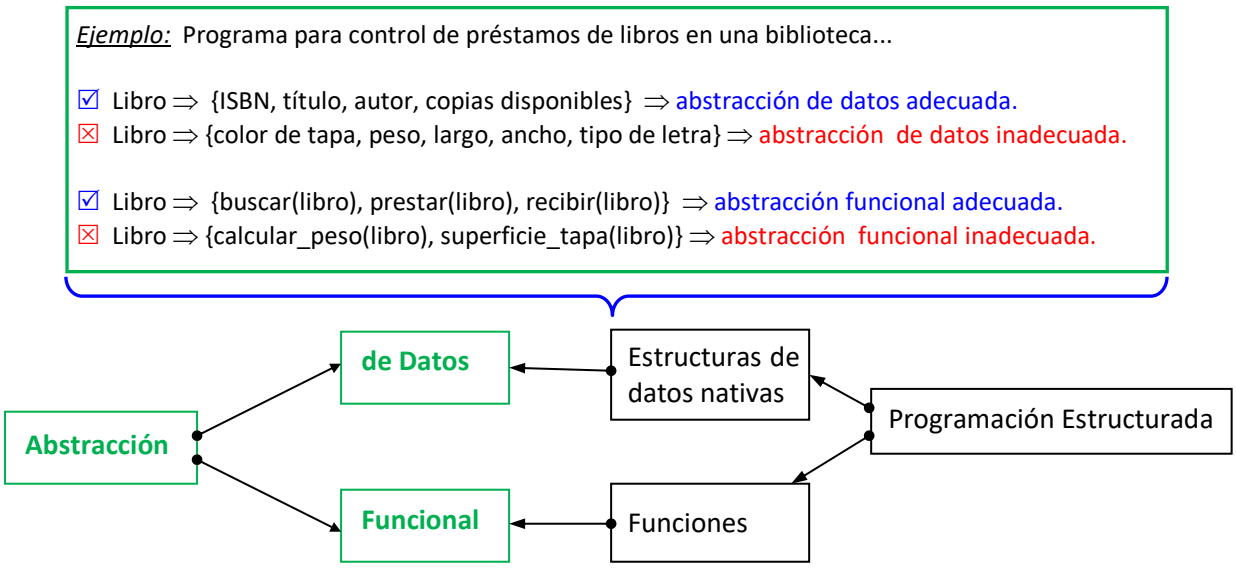
Las estructuras de datos que el programador necesita pero que el lenguaje no provee en forma nativa, se designan en general como *estructuras abstractas o tipos abstractos*. En el ejemplo anterior, *los registros como estructuras generales son estructuras nativas de Python*, pero el tipo *Libro* desarrollado por el programador para un requerimiento específico representa una *estructura abstracta*. El proceso llevado a cabo por el programador para crear el nuevo tipo abstracto (*Libro* en este ejemplo), se conoce con el nombre general de ***implementación*** del tipo abstracto [1].

El proceso de *implementación* de tipos o estructuras abstractas es una tarea muy común en programación, ya que es también muy común que se detecte la necesidad de emplear nuevos tipos que reflejen mejor el dominio de datos de un problema. Una vez que el programador descubre que debe implementar un nuevo tipo abstracto, el paso inicial es aplicar un mecanismo conceptual designado como ***mecanismo de abstracción***: es el proceso mediante el cual se intenta captar y aislar la *información* y el *comportamiento esencial* de una entidad, elemento u objeto del dominio de un problema.

La información que se usa en un programa de computadora es una *selección simplificada de datos de la realidad* de forma tal que ese conjunto de datos se considera relevante para el problema estudiado. Por lo tanto, una ***abstracción*** es también una simplificación de la realidad.

A su vez, el ***mecanismo de abstracción*** se descompone en dos aspectos: captar y aislar los ***datos relevantes*** de una entidad del dominio se conoce como ***abstracción de datos***, mientras que el mecanismo de identificar ***procesos relevantes*** se suele designar como ***abstracción funcional***. Cuando aquí se habla de *datos relevantes* y de *procesos relevantes* se quiere expresar que el programador debe hacer un análisis detallado de cuáles son los datos que ***realmente*** necesita en su programa para representar a una entidad del enunciado o dominio del problema, y cuales son los procesos o algoritmos que ***realmente*** necesita implementar en su programa para procesar esos datos. La gráfica siguiente muestra un ejemplo de un esquema de abstracción aplicado al concepto de *libro* en un programa de gestión de datos para una biblioteca:

Figura 1: Esquema de un proceso de abstracción.



En general dentro del paradigma de *programación estructurada* la ***abstracción de datos*** se realiza mediante la combinación de *estructuras de datos nativas* (comúnmente un registro agrupando distintos campos, aunque no es obligatorio el uso de registros) y la ***abstracción funcional*** se realiza mediante *funciones que luego tomen como parámetro a variables de los nuevos tipos*, o bien mediante *funciones que creen y retornen variables de esos tipos*.

En el ejemplo de la *Figura 1*, la entidad del dominio del problema que se quiere representar en el programa es el *libro*. Para ello, se supone que el programador definirá un tipo de registro (o una clase) llamado *Libro* y agregará campos al mismo para representar un libro (*abstracción de datos*). Sin embargo, no cualquier conjunto de campos será útil o relevante para su programa (y esto se determina en función del enunciado o requerimiento del problema): si el programa se usará para gestión de préstamos de libros, entonces un conjunto de campos de la forma {color de tapa, peso, largo, ancho, tipo de letra} no tendría ningún sentido (*abstracción de datos inadecuada*), pero el conjunto {ISBN, titulo, autor, cantidad de copias disponibles} sería relevante (abstracción de datos adecuada). Lo mismo pasa con el conjunto de posibles funciones a programar en ese proyecto (*abstracción funcional*) y en ese contexto: funciones para calcular el peso del libro o la superficie de la tapa del libro no serían en absoluto aplicables al contexto del problema (*abstracción funcional inadecuada*), pero funciones para buscar un libro, gestionar el préstamo, o recibir el libro en devolución serían no sólo adecuadas sino exigibles (*abstracción funcional adecuada*).

Como se dijo, una parte importante del trabajo de un programador es la implementación de estructuras abstractas. Esta Ficha está enfocada en el tratamiento e implementación de dos tipos abstractos: las *pilas* y las *colas*, que son quizás las estructuras de datos abstractas más sencillas de conceptualizar e implementar, sin que eso signifique que sean triviales sus aplicaciones.

2.] Estructuras de datos lineales y estructuras de datos no lineales.

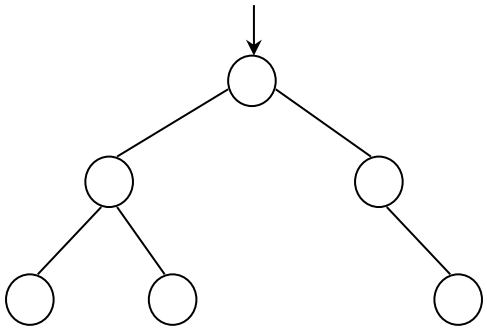
Tanto si se requiere diseñar e implementar una nueva estructura de datos abstracta, como si se quiere simplemente emplear y aplicar alguna estructura (nativa o abstracta) ya implementada, suele ser conceptualmente útil distinguir entre aquellas que son *lineales* de aquellas que son *no lineales*. La idea de linealidad en una estructura de datos se refiere a la forma en que internamente los datos son almacenados dentro de la colección, y la forma en que eso impactará luego en el recorrido de la estructura.

En ese sentido, una estructura de datos se define como *estructura lineal*, si cada elemento de esta tiene un único elemento sucesor inmediato (o ninguno si es el último) y también tiene un único elemento antecesor inmediato (o ninguno si es el primero). De acuerdo a esto, todos las estructuras de tipo secuencia en Python (*tuplas*, *listas*, *rangos*, *cadenas*) son lineales: si v es la estructura analizada, entonces el elemento v[i] tiene como único sucesor inmediato a v[i+1] (salvo que v[i] sea el último), y tiene como único antecesor inmediato a v[i-1] (salvo que v[i] sea el primero). Además de las secuencias típicas de Python que hemos nombrados, son también lineales un par de estructuras muy simples y conocidas: las *pilas* y las *colas* (que analizaremos en esta misma ficha).

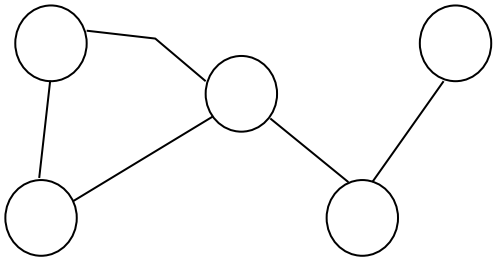
Por otra parte, una estructura de datos se dice *no lineal* si sus elementos pudiesen tener más de un sucesor o antecesor inmediato. La mayor parte de las estructuras no lineales que

suelen estudiarse en cursos introductivos de programación suelen ser abstractas (no vienen ya implementadas en forma nativa en un lenguaje, o están implementadas en librerías separadas). Algunas de las más comunes son los *árboles* (binarios o no binarios) y los *grafos* (dirigidos o no dirigidos). Ambas serán presentadas y analizadas en forma básica en fichas posteriores. Mostramos en la ***Figura 2*** un par de ejemplos esquemáticos y muy simples de ambas estructuras:

Figura 2: Ejemplos simples de estructuras no lineales típicas.



Un árbol binario (esquema)



Un grafo no dirigido (esquema)

Es intuitivamente claro que una *estructura lineal* se presenta como más simple de recorrer y procesar: en cualquiera de ellas, un simple ciclo puede avanzar hacia el extremo final de la colección sin ninguna dificultad. Si la estructura está basada en índices, es solo cuestión de sumar uno al índice actual y se tendrá el índice del sucesor, sin más trámite. Pero si la estructura es *no lineal*, entonces el avance hacia el sucesor implicará tomar alguna decisión: si hay más de un sucesor posible para cada elemento, el programador tendrá que diseñar algoritmos que prioricen avanzar hacia uno u otro camino, y de alguna manera recordar cuáles son los caminos que todavía no se recorrieron para luego poder volver atrás y completar esos caminos (y evitar recorrer de nuevo a los que ya fueron recorridos antes).

Cada estructura de datos posible tiene sus aplicaciones específicas que justifican que alguien se haya tomado el trabajo de diseñarlas e implementarlas. En ese sentido, existen numerosas situaciones y problemas que requieren que los datos sean tratados en *orden inverso al de su entrada*, y en estos casos son especialmente útiles las *pilas*. Otros problemas requieren que los datos sean procesados en el *mismo orden en que fueron cargados*, y en casos así las *colas* son estructuras de datos muy recomendables. Ambas estructuras pueden ser implementadas con relativamente poco trabajo (si es que no vienen nativas en el lenguaje utilizado), y esto es particularmente cierto en Python, como veremos en las secciones que siguen.

3.] Pilas.

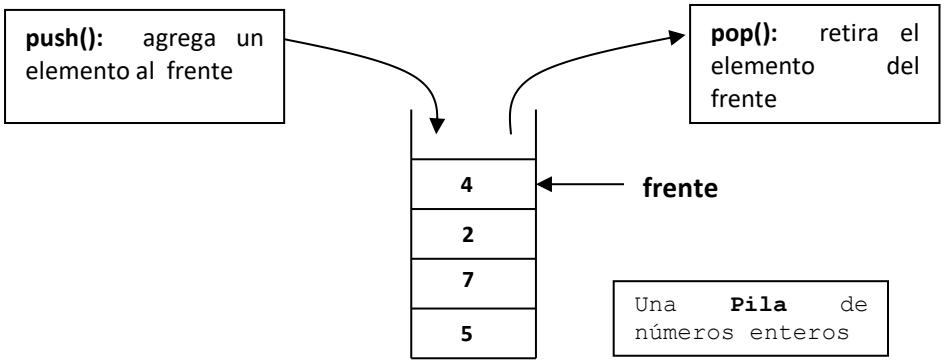
Una *pila* es una *estructura lineal* en la cual los elementos se organizan de forma tal que uno de ellos se ubica *al principio* (o *al frente* o *en el tope*) de la *pila* y los demás se *enciman* o

apilan uno sobre el otro[1] a partir del primero en ser insertado (y este queda ubicado al fondo de la pila) [1].

Si un elemento se inserta en una pila, lo hace de forma tal que queda ubicado como el elemento del frente. El único elemento que puede retirarse directamente con una sola operación es el elemento del frente (ver *Figura 3*). Como la operación de inserción puede entenderse como una operación de *empujar hacia abajo los datos* y dejar encima al último que ingresa, se suele designar como *push()* (en inglés: *empujar*) a la función o método que la implementa. Y como la operación de extraer un elemento se puede entender como la *expulsión hacia arriba* del elemento que está en el tope, esa función o método suele designarse como *pop()* (que en inglés hace referencia a la *acción de expulsar o saltar hacia arriba*).

Las *pilas* son estructuras de datos muy útiles en situaciones donde se debe invertir una secuencia de entrada: como el último elemento en insertarse se ubica arriba de todos y solo puede retirarse en forma directa el de arriba, entonces el último en ser insertado es el primero en ser retirado, y por lo tanto el primero en ser procesado. De este modo, si una secuencia de valores es almacenada en una *pila*, quedarán ubicados de forma que cuando comiencen a ser retirados para su procesamiento, *lo harán en orden inverso al que tenían cuando entraron*. Esta forma de procesamiento con inversión de entradas se conoce como ***LIFO*** (iniciales de *last in – first out*: último en entrar – primero en salir), y por ello mismo las *pilas* suelen designarse como *estructuras tipo LIFO*.

Figura 3: Esquema conceptual de una pila de números enteros.



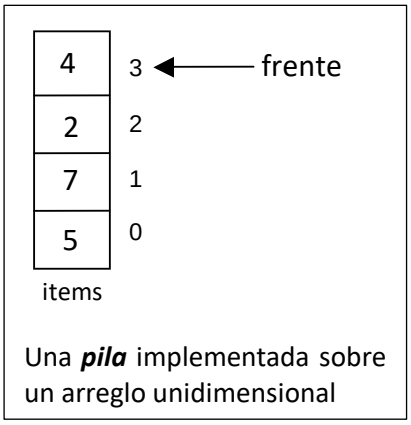
Observemos (por ejemplo) que el segmento de memoria que se conoce como *stack segment* se comporta como una *pila de bloques de memoria* para soportar el esquema de llamadas de funciones que pudiera disparar un programa. A pesar de su apariencia sencilla, como vemos las *pilas* tienen una fuerte participación en el software de base de una computadora. A lo largo de esta ficha, analizaremos algunas aplicaciones sencillas de *pilas* para resolver problemas típicos.

[1] Una curiosa (y morbosa…) situación de *apilamiento de cosas* se puede ver en la película *World War Z* (o *Guerra Mundial Z*) de 2013, dirigida por *Marc Forster* y protagonizada por *Brad Pitt*: en cierto momento, miles y miles de *zombies* (sí… *zombies*… como los de *The Walking Dead*) inician un ataque sobre las murallas de la ciudad de Jerusalén, y para poder escalar esas murallas simplemente se trepan unos sobre otros, formando descomunales *pilas de zombies*, hasta que logran llegar a la parte alta de la muralla y pasar al otro lado para atacar a los desprevenidos habitantes que su vez serán convertidos en *zombies*. No se puede negar que se trata de una espeluznante y original aplicación práctica para las pilas… ☺.

La forma más simple de implementar una *pila* en un programa es usar un *arreglo unidimensional como soporte*. La idea es que los elementos de la *pila* se guarden en los casilleros de un arreglo (que por ejemplo puede llamarse *items*), comenzando desde la posición cero. El primer elemento que se inserta en la *pila* se guarda en el casillero cero del arreglo, y es hasta allí el elemento del frente de la *pila*. La *Figura 4* muestra la misma pila de la *Figura 3*, suponiendo que se implementa sobre un arreglo unidimensional en Python.

Si se inserta otro elemento en la *pila* (operación *push()*), se guarda en el casillero número uno del arreglo, y este pasa a ser el del frente. Si se desea obtener y eliminar el elemento del frente (operación *pop()*), sólo se debe remover el elemento que se encuentra en la última posición del arreglo de soporte y retornar ese valor (recuerde que en Python, la última casilla también se identifica con el índice *-1*). Todo esto es particularmente simple de hacer en Python, ya que a diferencia de otros lenguajes, los arreglos unidimensionales en Python son de naturaleza dinámica y pueden aumentar o disminuir su tamaño en forma directa, mediante una llamada a *append()* para aumentarlo o al operador *del* para disminuirlo [2] [3]. Si el programador decidiera que el frente de la pila estuviese en la casilla 0 en lugar de en la última, entonces puede agregar un nuevo elemento con un corte de índices (*pila[0:0]*) en lugar de *append()*.

Figura 4: Esquema de una pila implementada sobre un arreglo unidimensional.



Con estas pautas, se puede definir un módulo *stack.py* en Python que contenga la implementación de una *pila* soportada en un arreglo. Lo típico es que un módulo que contenga la implementación de una estructura abstracta declare una clase para ello. La *abstracción de datos* se hará representando completamente a la pila dentro de la clase por el arreglo que se define como único atributo, y los métodos para gestionar ese arreglo se encargarán de la *abstracción funcional*.

El módulo *stack.py* está incluido en el proyecto *[F28] Pilas y Colas* que acompaña a esta Ficha. Este módulo implementa varias operaciones básicas para manejo de una pila representada internamente como un arreglo unidimensional, que luego analizaremos:

```
class Stack:
    def __init__(self):
        """Crea una pila vacía.
        """
        self.items = []

    def __str__(self):
        """Retorna una cadena simple con el contenido de la pila.
```

```
        """
        return str(self.items)

    def is_empty(self):
        """Chequea si la pila está vacía.
        :return: True si pila está vacía - False en caso contrario.
        """
        n = len(self.items)
        return n == 0

    def peek(self):
        """Retorna el elemento del frente de la pila, sin eliminarlo.
        :return: el valor que del frente, o None si la pila estaba vacía.
        """
        x = None
        if not self.is_empty():
            frente = -1
            x = self.items[frente]
        return x

    def push(self, x):
        """Inserta un elemento x al frente de la pila.
        :param x: el elemento a insertar.
        """
        self.items.append(x)

    def pop(self):
        """Elimina y retorna el elemento del frente de la pila.
        :return: el valor que del frente, o None si la pila estaba vacía.
        """
        x = None
        if not self.is_empty():
            frente = -1
            x = self.items[frente]
            del self.items[frente]
        return x
```

El método *__init__(self)* realiza una tarea simple y directa: solo debe iniciar como arreglo vacío al único atributo *self.items* que representa a la pila que se quiere crear, también vacía. El método *__str__(self)* simplemente convierte el arreglo self.items es una cadena, y retorna esa cadena (la conversión a cadena y eventual visualización de la pila será simplemente la conversión a cadena del arreglo de soporte).

La operación de insertar un elemento en la *pila* es realizada por el método *push(self, x)*, el cual también es simple: si se consideró que el frente o tope está en la última casilla, entonces el nuevo elemento *x* a ser insertado debe ubicarse al final del arreglo de soporte. Esto en Python es directo: solo se debe invocar a *append()*.

La operación de eliminar el elemento del frente (*pop(self)*) hace exactamente lo contrario de la operación de insertar: elimina el elemento que en ese momento se encuentre en la última casilla del arreglo, y retorna ese valor para su eventual uso posterior. Simplemente se controla si el arreglo está vacío (en cuyo caso no puede removerse nada de él y se retorna *None*), y en caso de contener al menos un elemento, se copia el valor de la última casilla en *x* (con la instrucción *x = self.items[-1]*) y luego se elimina ese casillero del arreglo con la instrucción *del self.items[-1]*) [2]. Recuerde que en Python, la última casilla de una secuencia puede accederse con el índice -1.

Se incluyó además un método *peek(self)* que retorna el valor del elemento del frente, pero sin removerlo de la *pila*, más un método *is_empty(self)* para determinar si la *pila* está vacía o no. Ambos son muy sencillos y se deja su análisis para el alumno.

El modelo *test01.py* incluido en el mismo proyecto *[F28].Pilas y Colas* contiene un pequeño programa para probar todos estos métodos. De nuevo, dejamos al alumno la tarea de analizar su funcionamiento:

```
import stack


def main():
    p1 = stack.Stack()
    p1.push(5)
    p1.push(7)
    p1.push(2)
    p1.push(4)
    print('Estado actual de la pila:', p1)

    y = p1.pop()
    print('Elemento removido del frente:', y)

    x = p1.peek()
    print('Elemento actual al frente:', x)

    print('Estado actual de la pila:', p1)
    if p1.is_empty():
        print('La pila está vacía...')
    else:
        print('La pila no está vacía')

    while not p1.is_empty():
        print('Elemento removido:', p1.pop())

    print('Estado actual de la pila:', p1)
    if p1.is_empty():
        print('La pila está vacía...')
    else:
        print('La pila no está vacía')


if __name__ == '__main__':
    main()
```

4.] Colas.

Una *cola* es una estructura lineal en la cual los elementos se organizan uno detrás del otro, quedando uno de ellos al principio (o *al frente*) de la *cola* y otro en el último lugar de esta (o *al fondo*). Para insertar un elemento (operación *add()*) se hace de forma tal que el nuevo componente queda último en la *cola*. Para eliminar un elemento (operación *remove()*) sólo puede eliminarse aquél que está primero (ver *Figura 5* más abajo). Entonces, como cada elemento que se inserta queda último en la fila y por ese motivo será también el último en ser retirado, las *colas* suelen designarse también genéricamente como estructuras tipo ***FIFO*** (por *First In – First Out: primero en entrar, primero en salir*) [1].

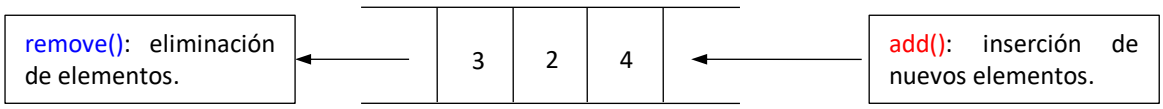
Las *colas* son estructuras de datos muy útiles en programas que requieren efectuar lo que se denomina una *simulación de situaciones de espera frente a un puesto de servicio*. En estos

programas se supone la existencia de un *proveedor de servicios* (por ejemplo, una ventanilla de un cajero de un banco, un puesto de peaje en una ruta, un programa controlador de entrada a una impresora en un sistema de red, etc.), y se busca simular la llegada de *clientes* para ese servicio, los cuales *forman una cola* (o *fila*) de espera hasta que les toca ser atendidos.

Los programas de simulación de este tipo buscan estudiar el comportamiento de estas colas en cuanto a diversos resultados. Por ejemplo, puede pedirse determinar cual es el tiempo promedio de espera de un auto en una cola hasta que es atendido en la ventanilla de peaje, o puede pedirse determinar si hace falta abrir más ventanillas de cobro a una hora determinada en un banco de acuerdo a la cantidad de clientes que hay en las colas a esa hora.

La realización de programas de simulación como los descriptos escapa a los alcances de este curso. Nos limitaremos sólo a mostrar la forma básica de implementar una cola en el lenguaje Python, y su aplicación en programas simples.

Figura 5: Esquema conceptual de una cola de números enteros.



Al igual que las pilas, una *cola* también puede implementarse usando un arreglo como soporte, y otra vez esto es directo y relativamente sencillo en Python, favorecido por el hecho de gestionar arreglos como variables de tipo *list*, y tener con ellos la propiedad del crecimiento y del decrecimiento en forma dinámica. La idea es exactamente la misma que hemos sugerido para las pilas, pero ahora simplemente cambiando la forma de trabajo de la función de inserción de un elemento para que lo haga en el extremo opuesto al que se use para retirar un elemento: lo común es que la inserciones se hagan a partir de la última casilla de la derecha (usando *append()*) y las eliminaciones se hagan a partir de la primera casilla de la izquierda (usando *del*) [2] [3].

El módulo *queue.py* incluido en el proyecto *[F28] Pilas y Colas* implementa una clase con todas la funcionalidad necesaria para la gestión de una *cola* con estas ideas. El parecido con el módulo *stack.py* salta a la vista, por lo cual dejamos su análisis para el alumno:

```
class Queue:
    def __init__(self):
        """Crea una cola vacía.
        """
        self.items = []

    def __str__(self):
        """Retorna una cadena simple con el contenido de la cola.
        """
        return str(self.items)

    def is_empty(self):
        """Chequea si la cola está vacía.
        :return: True si cola está vacía - False en caso contrario.
        """
        n = len(self.items)
        return n == 0
```

```
    def peek(self):
        """Retorna el elemento del frente de la cola, sin eliminarlo.
        :return: el valor del frente, o None si la cola estaba vacía.
        """
        x = None
        if not self.is_empty():
            x = self.items[0]
        return x

    def add(self, x):
        """Inserta un elemento x al fondo de la cola.
        :param x: el elemento a insertar.
        """
        self.items.append(x)

    def remove(self):
        """Elimina y retorna el elemento del frente de la cola.
        :return: el valor del frente, o None si la cola estaba vacía.
        """
        x = None
        if not self.is_empty():
            x = self.items[0]
            del self.items[0]
        return x
```

El modelo *test03.py* incluido en el mismo proyecto, contiene un programa para aplicar cada una de las funciones vistas, pero ahora en una *cola* de números:

```
import queue


def main():
    q1 = queue.Queue()
    q1.add(5)
    q1.add(7)
    q1.add(2)
    q1.add(4)
    print('Estado actual de la cola:', q1)

    y = q1.remove()
    print('Elemento removido del frente:', y)

    x = q1.peek()
    print('Elemento actual al frente:', x)

    print('Estado actual de la cola:', q1)
    if q1.is_empty():
        print('La cola está vacía...')
    else:
        print('La cola no está vacía')

    while not q1.is_empty():
        print('Elemento removido:', q1.remove())

    print('Estado actual de la cola:', q1)
    if q1.is_empty():
        print('La cola está vacía...')
    else:
        print('La cola no está vacía')
```

```
if __name__ == '__main__':
    main()
```

5.] Uso de pilas en problemas de control de simetría.

Un interesante ejercicio de aplicación de pilas para controlar si una secuencia de datos presenta elementos de simetría, puede usarse para cerrar esta ficha. Mostramos directamente el enunciado y la solución propuesta.

Problema 61.) *Desarrollar un programa que cargue por teclado una cadena de caracteres y use una pila para determinar si esa cadena es capicúa: esto es, queremos determinar si una cadena de caracteres que se recibe para analizar puede leerse igual en dirección izquierda–derecha que en dirección derecha–izquierda.*

Discusión y solución: El proyecto *[F28] Pilas y Colas* que acompaña a esta Ficha contiene un modelo *test02.py* con el programa completo que resuelve este problema.

Usando una *pila* la solución es directa: la idea es que se guardan en la *pila* los caracteres de la mitad izquierda de la palabra, uno a uno tomados en el mismo orden en que aparecen en la cadena. Al guardarlas en la pila, el *orden se invierte*: el caracter que entró primero se va al fondo de la *pila*, y el que entró último queda al frente.

Luego se recorre la segunda mitad de la cadena, y por cada caracter que se tenga, se extrae a su vez un caracter de la *pila*. Si la cadena era capicúa, los caracteres extraídos de la *pila* deberán coincidir siempre con los caracteres que se recorren en la segunda mitad de la cadena. Si algún par de caracteres no coincide, la palabra ***no es capicúa*** y el proceso se detiene. Se deja el estudio de los detalles para el alumno, o para su discusión en clase:

```
import stack


def capicua(s):
    n = len(s)
    a = stack.Stack()

    mitad = n // 2
    if n % 2 == 1:
        d = mitad + 1
    else:
        d = mitad

    # fase 1: almacenar en la pila la mitad izquierda de la cadena...
    for i in range(mitad):
        a.push(s[i])

    # fase 2: recorrer la mitad derecha de la cadena y controlar
    # con lo que sale de la pila...
    for i in range(d, n):
        x = a.pop()
        if x != s[i]:
            return False
    return True


def main():
    s = input('Ingrese la cadena a analizar: ')
    res = capicua(s)
```

```
    if res:
        print('La cadena es capicúa...')
    else:
        print('La cadena no es capicúa...')


if __name__ == '__main__':
    main()
```

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

[3] M. Pilgrim, "Dive Into Python - Python from novice to pro" , Nueva York: Apress, 2004.