Programación II

Tecnicatura Universitaria en Inteligencia Artificial

2022

Tipos Abstractos de Datos

1. Introducción a los Tipos Abstractos de Datos

Los tipos de datos definidos en unidades anteriores son **tipos de datos concretos**. Por ejemplo, un punto se definió como un par ordenado de números, un rectángulo se definió como un punto y dos números, etc. Otra forma de definir tipos de datos es *mediante sus operaciones*: enumerándolas e indicando su comportamiento (es decir, cuál es su resultado esperado). Esta manera de definir datos se conoce como **Tipos Abstractos de Datos** o **TADs** [1]. Lo novedoso de este enfoque respecto del anterior es que en general se puede encontrar más de una representación mediante tipos concretos para representar el mismo TAD, y que se puede elegir la representación más conveniente en cada caso, según el contexto de uso. Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación. Dentro del ciclo de vida de un TAD hay dos fases: la programación del TAD y la construcción de los programas que lo usan. Durante la fase de programación del TAD, habrá que elegir una representación, y luego programar cada uno de los métodos sobre esa representación. Cuando la representación involucra un conjunto de valores, se debe elegir una forma de organizarlos en la memoria de la computadora. Para ello se diseña una **estructura de datos** que permita que los datos puedan ser accedidos y manipulados en forma eficiente. En una estructura de datos se define la colección de valores que se almacenan, cómo están agrupados y cómo se relacionan entre ellos. Durante la fase de construcción de los programas, no será relevante para el programador que utiliza el TAD cómo está implementado, sino únicamente los métodos que posee. En programación orientada a objetos, una **interfaz** (también llamada **protocolo**) es un medio común para que los objetos no relacionados se comuniquen entre sí. Estas son definiciones de métodos y valores sobre los cuales los objetos están de acuerdo para cooperar. Utilizando el concepto de interfaz, podemos decir que a quien utilice el TAD, sólo le interesará la interfaz que éste ofrezca. Veamos un ejemplo con un tipo de dato que ya conocemos. El **TAD Número Entero** está definido por el conjunto de los números enteros y todas las operaciones que podemos hacer con ellos: sumar, restar, multiplicar, etc. ¿Como representamos un número entero en memoria?

- En Python, el tipo de dato `int` es una implementación concreta del TAD Número Entero, que soporta números de precisión arbitraria: puede almacenar números arbitrariamente grandes mientras haya memoria suficiente para hacerlo. Para lograr esto, internamente el valor numérico se guarda en una estructura de datos que, simplificando para no entrar en detalles, es una secuencia de dígitos.

- En otros lenguajes como C, la representación interna del tipo de dato entero es de tamaño fijo, y solo

[1] También conocida como Tipos de Datos Abstractos (TDAs). En general, las expresiones *tipo abstracto de datos* y *tipo de datos abstracto* son intercambiables, ambas provienen de la traducción del término en inglés *abstract data type*.

puede almacenar números en un rango limitado. Por ejemplo, el tipo de dato `uint8_t` internamente utiliza 8 bits para codificar un número entero no negativo. Con 8 bits tenemos $2^8 = 256$ combinaciones posibles; por lo tanto el tipo de dato `uint8_t` puede representar números entre 0 y 255. Como ventaja, la manipulación de números de tamaño fijo es más eficiente que los de tamaño arbitrario como en Python.

2. TAD - Lista (List)

El **TAD Lista** representa un conjunto de valores ordenados y numerables, con las siguientes operaciones:

- `__str__` para obtener una representación de la lista como cadena de texto.
- `__len__` para calcular la longitud de la lista.
- `append(x)` para agregar un elemento al final de la lista.
- `insert(i, x)` para agregar el elemento x en la posición i.
- `remove(x)` para eliminar la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y detendrá la ejecución inmediatamente.)
- `pop(i)` para eliminar el elemento que está en la posición i y devolver su valor. Si no se especifica el valor de i, pop() elimina y devuelve el elemento que está en el último lugar de la lista (si el indice es inválido, imprimirá un error y detendrá la ejecución inmediatamente.)
- `index(x)` devuelve la posición de la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y detendrá la ejecución inmediatamente.)

2.1. Implementación: Listas nativas

Las listas de Python son una implementación del TAD Lista. Esto significa que el tipo de datos nativo `list` en Python provee una implementación para (al menos) las operaciones mencionadas en la sección anterior [2].

2.1.1. Arreglos y Performance

La representación interna de las listas nativas de Python utiliza una estructura de datos llamada **arreglo**. Un arreglo es la forma más simple de almacenar una secuencia de datos en la memoria, ya que los elementos están contiguos. Acceder a un elemento cualquiera de un arreglo dada su posición `i` (es decir, lo que hacemos en Python con la operación `lista[i]`) es una operación de **tiempo constante**, por lo cual, lo que tarda la operación no depende de la cantidad de elementos que contiene el arreglo:

```
>>> # Creamos una lista con 100000 elementos:
>>> numeros = list(range(100000))
>>> numeros[3692] # tarda lo mismo sin importar el argumento
3691
```

Esta es una cualidad muy útil de los arreglos (y en particular de las listas de Python), que nos permite tener listas de millones de elementos y acceder a cualquiera de ellos casi instantáneamente (siempre que sepamos su posición). En cambio, dado que los elementos deben estar siempre contiguos en la memoria, agregar o quitar elementos en una posición dada del arreglo es una operación de **tiempo lineal**: esta operación va a tardar más o menos dependiendo del tamaño de la lista, y la posición donde queramos agregar o quitar elementos.

[2] Puede verse documentación sobre las operaciones en la documentación oficial. Sin embargo, por ser un tipo de datos nativo de Python, la implementación de las listas se encuentra en el código fuente del lenguaje.

```
>>> numeros.pop(3692) # esta operación es de tiempo lineal
```

Esta es una desventaja importante de los arreglos. Si para resolver un problema determinado tenemos una lista de millones de elementos y tenemos que insertar o eliminar elementos frecuentemente, la implementación resultará poco eficiente. El arreglo no es la única representación posible del TAD Lista. A continuación veremos una estructura de datos muy diferente al arreglo, con sus propias ventajas y desventajas.

2.2. Implementación: Lista Enlazada

Una lista enlazada está formada por nodos, y cada nodo guarda un elemento y una referencia a otro nodo, como si fueran vagones en un tren. Vamos a explorar este concepto en Python. En primer lugar, definiremos una clase muy simple, Nodo, que tendrá dos atributos: `dato`, que servirá para almacenar cualquier información, y `proximo`, que servirá para poner una referencia al siguiente nodo. Además, como siempre, implementaremos el constructor y el método `__str__` para poder obtener una representación en cadena de texto.

```
from typing import Any

class Nodo:
    def __init__(self, dato: Any =None, prox: "Nodo" | None =None):
        self.dato = dato
        self.prox = prox
    def __str__(self):
        return str(self.dato)
```

Si ejecutamos este código:

```
>>> n3 = Nodo("Bananas")
>>> n2 = Nodo("Peras", n3)
>>> n1 = Nodo("Manzanas", n2)
>>> str(n1)
'Manzanas'
>>> str(n2)
'Peras'
>>> str(n3)
'Bananas'
```

Este código crea tres nodos y los enlaza en un orden particular, como se ve en la figura 1.

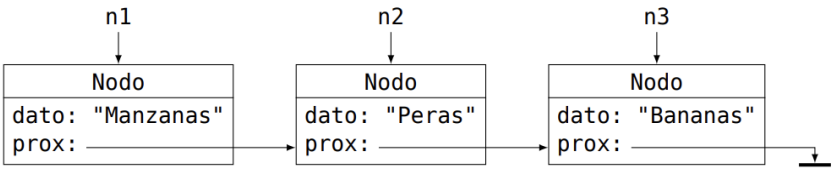

Figura 1: Nodos enlazados.

El atributo `proximo` de `n3` tiene una referencia nula, lo que indica que `n3` es el último nodo de nuestra estructura. Hemos creado una lista en forma manual. Si nos interesa recorrerla, podemos hacer lo siguiente:

```
def ver_lista(nodo: Nodo | None) -> None:
    """Recorre todos los nodos a través de sus enlaces,
    mostrando sus contenidos."""
    while nodo is not None:
        print(nodo)
```

```
        nodo = nodo.prox
```

Podemos utilizar esta función de la siguiente forma:

```
>>> ver_lista(n1)
Manzanas
Peras
Bananas
```

Es interesante notar que la estructura del recorrido de la lista es el siguiente:

- Se le pasa a la función sólo la referencia al primer nodo.
- El resto del recorrido se consigue siguiendo la cadena de referencias dentro de los nodos.

Si se desea «desenganchar» un nodo del medio de la lista, alcanza con cambiar la referencia `proximo`, por ejemplo:

```
>>> n1.prox = n3
>>> ver_lista(n1)
Manzanas
Bananas
>>> n1.prox = None
>>> ver_lista(n1)
Manzanas
```

2.2.1. Referenciando el inicio de la lista

Una cuestión no contemplada hasta el momento es la de mantener una referencia a la lista completa. Por ahora para nosotros la lista es la colección de nodos que se enlazan a partir de n1. Sin embargo puede suceder que queramos quitar a n1 y continuar con el resto de la lista como la colección de nodos a tratar. Una solución muy simple es asociar una referencia al principio de la lista, que llamaremos lista, y que mantendremos independientemente de cuál sea el nodo que está al principio de la lista, por ejemplo:

```
>>> n3 = Nodo("Bananas")
>>> n2 = Nodo("Peras", n3)
>>> n1 = Nodo("Manzanas", n2)
>>> lista = n1
>>> ver_lista(lista)
Manzanas
Peras
Bananas
```

Ahora sí estamos en condiciones de eliminar el primer elemento de la lista sin perder la identidad de la misma:

```
>>> lista = lista.prox
>>> ver_lista(lista)
Peras
Bananas
```

Ya podemos ver una ventaja importante de las listas enlazadas: eliminar el primer elemento es una operación de tiempo constante, es decir que no depende de la longitud de la lista. En las listas de Python, como ya vimos, esta operación requiere un tiempo lineal. Sin embargo no todo es tan positivo: el acceso a la posición i se realiza en un tiempo proporcional a i, mientras que en las listas de Python esta operación se realiza en tiempo constante.

2.2.2. La clase `ListaEnlazada`

Basándonos en los nodos implementados anteriormente, pero buscando desligar al programador que desea usar la lista de la responsabilidad de manipular las referencias, definiremos ahora la clase ListaEnlazada, de modo tal que no haya que operar mediante las referencias internas de los nodos, sino que se lo pueda hacer a través de operaciones de lista. Se podrá notar que implementaremos los métodos mencionados en el TAD Lista. De este modo, más allá de las diferencias en el funcionamiento interno, tanto la clase ListaEnlazada como las listas nativas de Python serán implementaciones del TAD Lista. Recordamos a continuación las operaciones que inicialmente deberá cumplir la clase ListaEnlazada.

- `__str__` para obtener una representación como cadena de texto.
- `__len__` para calcular la longitud de la lista.
- `append(x)` para agregar un elemento al final de la lista.
- `insert(i, x)` para agregar el elemento x en la posición i.
- `remove(x)` para eliminar la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y terminará su ejecución inmediatamente.)
- `pop(i)` para eliminar el elemento que está en la posición i y devolver su valor. Si no se especifica el valor de i, pop() elimina y devuelve el elemento que está en el último lugar de la lista (si el indice es inválido, imprimirá un error y terminará su ejecución inmediatamente.)
- `index(x)` devuelve la posición de la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y terminará su ejecución inmediatamente.)

Más adelante podrá agregarse a la lista otros métodos que también están implementados por las listas de Python. Valen ahora algunas consideraciones más antes de empezar a implementar la clase:

- Por lo dicho anteriormente, es claro que la lista deberá tener como atributo la referencia al primer nodo que la compone.
- Una implementación trivial del método `__len__` es recorrer todos los nodos de la lista y contar la cantidad de elementos, pero si la lista tiene muchos elementos esto podría ser poco eficiente. Para mejorar la eficiencia alcanza con agregar un atributo numérico que contenga la cantidad de nodos. Así, este atributo que llamaremos `len` se inicializará en 0 cuando se cree la lista vacía, se incrementará en 1 cada vez que se agregue un elemento y se decrementará en 1 cada vez que se elimine un elemento.

  ☞ Al agregar el atributo `len` estamos duplicando información, ya que habrá dos formas de obtener la longitud de la lista:

  1. contar los nodos.
  2. obtener el valor de `len`.

  Como consecuencia, vamos a tener que prestar especial atención para que el atributo `len` siempre contenga un valor consistente; es decir que su valor sea siempre igual a la cantidad de nodos que contiene la lista. Mas adelante explicamos mas formalmente este concepto.

- Por otro lado, como vamos a incluir todas las operaciones de listas que sean necesarias para operar con ellas, no es necesario que la clase `Nodo` esté disponible para que otros programadores puedan modificar (y romper) las listas a voluntad usando operaciones de nodos. Para eso incluiremos la clase `Nodo` de manera privada (es decir oculta), de modo que la podamos usar nosotros como dueños (fabricantes)

de la clase, pero no cualquier programador que utilice la lista. Python tiene una convención para hacer que atributos, métodos o clases dentro de una clase dada no puedan ser usados por los usuarios, y sólo tengan acceso a ellos quienes programan la clase: su nombre tiene que empezar con un guión bajo y terminar sin guión bajo. Así que para hacer que los nodos sean privados, cambiaremos el nombre de la clase a `_Nodo`. [3].

2.2.3. Construcción de la lista enlazada

Empezamos escribiendo la clase con su constructor.

```
class ListaEnlazada:
    """Modela una lista enlazada."""
    def __init__(self) -> None:
        """Crea una lista enlazada vacía."""
        # referencia al primer nodo (None si la lista está vacía)
        self.prim = None
        # cantidad de elementos de la lista
        self.len = 0
```

Si la lista contiene dos elementos, su estructura será como la representada por la figura 2 .

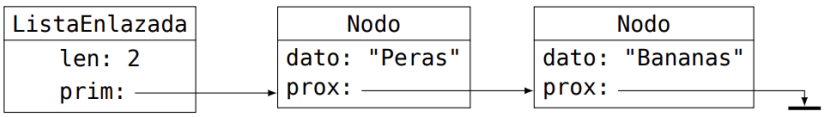

Figura 2: Lista enlazada.

Ejercicio Escribir los métodos `__str__` y `__len__`[4] para la lista enlazada.

2.2.4. Insertar elementos

Debemos programar ahora `insert(i, x)`, que debe agregar el elemento x en la posición i (e imprimir un mensaje de error y retornar inmediatamente si la posición i es inválida). Veamos qué debemos tener en cuenta para programar esta función.

- Si se intenta insertar en una posición menor que cero o mayor que la longitud de la lista debe imprimirse un mensaje de error y detener la ejecución del método inmediatamente:

```
    if i < 0 or i > self.len:
        print("Posición inválida")
        return
```

- Para los demás casos hay que crear un nodo, que será el que se insertará en la posición que corresponda. Construimos un nodo nuevo cuyo dato será x.

```
    nuevo = _Nodo(x)
```

- Si se quiere insertar en la posición 0, hay que cambiar la referencia de `self.prim`.

```
    if i == 0:
        nuevo.prox = self.prim.prox
        self.prim = nuevo
```

[3] Esto es sólo una convención entre programadores, nada impide que utilicemos la clase `_Nodo` desde fuera

[4] El método `__len__` **sobrecarga** la función `len` incluida en Python, por lo que podríamos obtener la longitud de la lista simplemente utilizando esta función, tal cuál lo hacemos con las listas nativas de Python

- Para los demás casos, obtenemos el nodo anterior a la posición en la que queremos insertar. Será necesario implementar una **máquina de parejas**. Por ejemplo, si nuestra secuencia tiene la forma ABCDE se itera sobre ella de modo de tener las parejas AB, BC, CD, DE. En la pareja XY llamaremos a X el elemento anterior y a Y el elemento actual. En general estos ciclos terminan o bien cuando no hay más parejas que formar, o bien cuando el elemento actual cumple con una determinada condición. En nuestro problema, tenemos la siguiente situación:

  - Las parejas son parejas de nodos.
  - Para avanzar en la secuencia se usa la referencia al próximo nodo de la lista.
  - En este caso particular no debemos preocuparnos por la terminación de la lista porque la validez del índice buscado ya fue verificada más arriba.

  Utilizando esto, nos queda un bucle como el siguiente:

```
    n_ant = self.prim
    for pos in range(1, i):
        n_ant = n_ant.prox
    nuevo.prox = n_ant.prox
    n_ant.prox = nuevo
```

- En todos los casos, se debe incrementar en un 1 el contador de elementos de la lista.

Juntando todo, el método `insert(i, x)` nos queda como sigue:

```
def insert(self, i: int, x: Any) -> None:
    """Inserta el elemento x en la posición i.
    Si la posición es inválida, imprime un error y retorna inmediatamente"""
    if i < 0 or i > self.len:
        print("Posición inválida")
        return

    nuevo = _Nodo(x)
    if i == 0:
        # Caso particular: insertar al principio
        nuevo.prox = self.prim
        self.prim = nuevo
    else:
        # Buscar el nodo anterior a la posición deseada
        n_ant =self.prim
        for pos in range(1, i):
            n_ant = n_ant.prox

        # Intercalar el nuevo nodo
        nuevo.prox = n_ant.prox
        n_ant.prox = nuevo

    self.len += 1
```

2.2.5. Eliminar un elemento por posición

Analizaremos a continuación `pop([i])`, que elimina el elemento que está en la posición i y devuelve su valor. Si no se especifica el valor de i, `pop()` elimina y devuelve el elemento que está en el último lugar de la lista. Por otro lado, si se hace referencia a una posición no válida de la lista, imprime un mensaje de error y

retorna inmediatamente. Dado que se trata de una función con cierta complejidad, separaremos el código en las diversas consideraciones a tener en cuenta.

- Si no se indica la posición, i toma la última posición de la lista:

```
    if i is None:
        i = self.len - 1
```

- Si la posición es inválida (i menor que 0 o mayor o igual a la longitud de la lista), se considera error, debemos informarlo y detener inmediatamente la ejecución del método. Esto se resuelve con este fragmento de código:

```
    if i < 0 or i > self.len:
        print("Posición inválida")
        return
```

- Cuando la posición es 0 se trata de un caso particular, ya que en ese caso hay que cambiar la referencia de `self.prim` para que apunte al nodo siguiente. Es decir, pasar de `self.prim` → nodo0 → nodo1 a `self.prim` → nodo1).

```
    if i == 0:
        dato = self.prim.dato
        self.prim = self.prim.prox
```

  (Guardamos el dato del nodo descartado para poder devolverlo al finalizar la función).

- Vemos ahora el caso general: Mediante un ciclo, se deben ubicar los nodos n_{i-1} y n_i que están en las posiciones i - 1 e i de la lista, respectivamente, de modo de poder ubicar no sólo el nodo que se descartará, sino también estar en condiciones de saltear el nodo descartado en los enlaces de la lista. La lista debe pasar de contener el camino $n_{i-1} \rightarrow n_i \rightarrow n_{i+1}$ a contener el camino $n_{i-1} \rightarrow n_{i+1}$. Nuevamente usaremos el patrón de máquina de parejas, utilizando como condición de terminación que la posición del nodo en la lista sea igual al valor buscado. En este caso particular no debemos preocuparnos por la terminación de la lista porque la validez del índice buscado ya fue verificada más arriba. Esta es la porción de código correspondiente a la búsqueda. Llamamos `n_ant` y `n_act` a los elementos anterior y actual de la pareja de nodos:

```
    n_ant = self.prim
    n_act = n_ant.prox
    for pos in range(1, i):
        n_ant = n_act
        n_act = n_ant.prox
```

  Al finalizar el ciclo, `n_ant` será una referencia al nodo $i-1$ y `n_act` una referencia al nodo i. Una vez obtenidas las referencias, se obtiene el dato y se cambia el camino para descartar el nodo `n_act`:

```
    dato = n_act.dato
    n_ant.prox = n_act.prox
```

- Finalmente, en todos los casos, actualizamos la longitud de la lista y devolvemos el dato:

```
    self.len -= 1
    return dato
```

Juntando todo, el método `pop` nos queda del siguiente modo:

```
def pop(self, i: int | None = None) -> Any:
    """Elimina el nodo de la posición i, y devuelve el dato contenido.
    Si i está fuera de rango, se muestra un mensaje de error y se
    retorna inmediatamente. Si no se recibe la posición, devuelve el
    último elemento."""
    if i is None:
        i = self.len - 1
    if i < 0 or i >= self.len:
        print("Posición inválida")
        return
    if i == 0:
        # Caso particular: saltear la cabecera de la lista
        dato = self.prim.dato
        self.prim = self.prim.prox
    else:
        # Buscar los nodos en las posiciones (i-1) e (i)
        n_ant = self.prim
        n_act = n_ant.prox
        for pos in range(1, i):
            n_ant = n_act
            n_act = n_ant.prox
        # Guardar el dato y descartar el nodo
        dato = n_act.dato
        n_ant.prox = n_act.prox
    self.len -= 1
    return dato
```

2.2.6. Eliminar un elemento por valor

Análogamente se resuelve `remove(self, x)`, que debe eliminar la primera aparición de x en la lista, o bien imprimir un mensaje de error y retornar inmediatamente si x no se encuentra en la lista. Nuevamente, dado que se trata de un método de cierta complejidad, lo resolveremos por partes, teniendo en cuenta los casos particulares y el caso general.

- Si la lista está vacía, mostramos un mensaje de error y detenemos la ejecución del método inmediatamente. Tenemos que tratarlo como un caso particular, ya que en todos los siguientes casos necesitamos que haya al menos un nodo.

```
    if self.prim is None:
        print("La lista esta vacía")
        return
```

- El caso en el que x está en el primer nodo también es particular, ya que modificar la referencia `self.prim`:

```
    if self.prim.dato == x:
        self.prim = self.prim.prox
```

- El caso general también implica un recorrido con máquina de parejas, sólo que esta vez la condición de terminación es: o bien la lista se terminó o bien encontramos un nodo con el valor x buscado.

```
    n_ant = self.prim
    n_act = n_ant.prox
    while n_act is not None and n_act.dato != x:
        n_ant = n_act
        n_act = n_ant.prox
```

En este caso, al terminarse el ciclo será necesario corroborar si se terminó porque llegó al final de la lista, y de ser así mostrar el mensaje de error correspondiente y retornar inmediatamente; o si se terminó porque encontró el dato, y de ser así eliminarlo.

```
        if n_act is None:
            print("El valor no está en la lista.")
            return
        n_ant.prox = n_act.prox
```

- Finalmente, en todos los casos de éxito debemos decrementar en 1 el valor de `self.len`.

Juntando todo, así nos queda el código completo del método `remove`

```
def remove(self, x: Any) -> None:
    """Borra la primera aparición del valor x en la lista.
    Si x no está en la lista, imprime un mensaje de error y retorna
    inmediatamente."""
    if self.len == 0:
        print("La lista esta vacía")
        return
    if self.prim.dato == x:
        # Caso particular: saltear la cabecera de la lista
        self.prim = self.prim.prox
    else:
        # Buscar el nodo anterior al que contiene a x (n_ant)
        n_ant = self.prim
        n_act = n_ant.prox
        while n_act is not None and n_act.dato != x:
            n_ant = n_act
            n_act = n_ant.prox
        if n_act == None:
            print("El valor no está en la lista.")
            return

        # Descartar el nodo
        n_ant.prox = n_act.prox
        self.len -= 1
```

Ejercicio Completar la implementación de lista enlazada con los métodos que faltan: `append` e `index`.

2.3. Invariantes de objeto

Los **invariantes** son condiciones que deben ser siempre ciertas. Las **invariantes de objetos**, que son condiciones que deben ser ciertas a lo largo de toda la existencia de un objeto. La clase `ListaEnlazada` presentada en la sección anterior, cuenta con dos invariantes que siempre debemos mantener. Por un lado, el atributo `len` debe contener siempre la cantidad de nodos de la lista. Es decir, siempre que se modifique la lista, agregando o quitando un nodo, se debe actualizar `len` como corresponda. Por otro lado, el atributo `prim` referencia siempre al primer nodo de la lista. Si se agrega o elimina este primer nodo, es necesario actualizar esta referencia.

Cuando se desarrolla una estructura de datos como la lista enlazada, es importante destacar cuáles serán sus invariantes, ya que en cada método habrá que tener especial cuidado de que los invariantes permanezcan siempre ciertos.

3. TAD - Pila (Stack)

El comportamiento de una pila se puede describir mediante la frase "Lo último que se apiló es lo primero que se usa", que es exactamente lo que uno hace con una pila (de platos por ejemplo): en una pila de platos uno sólo puede ver la apariencia completa del plato de arriba, y sólo puede tomar el plato de arriba (si se intenta tomar un plato del medio de la pila lo más probable es que alguno de sus vecinos, o él mismo, se arruine).

Es por eso que las pilas también se llaman estructuras **LIFO** (del inglés *Last In First Out*), debido a que el último elemento en entrar será el primero en salir.

Veamos primero la interfaz del **TAD Pila**:

- `__init__`: inicializa una pila vacía
- `push`: también llamado apilar, agrega un nuevo elemento a la pila
- `pop`: también llamado des-apilar, elimina y devuelve un elemento de la pila. El elemento devuelto es siempre el último agregado.
- `isEmpty`: chequea si la pila está vacía o no.

Como ya se dijo, al crear un tipo abstracto de datos, es importante decidir cuál será la representación a utilizar. En el caso de la pila, si bien puede haber más de una representación, por ahora veremos la más sencilla: representaremos una pila mediante una lista de Python. Sin embargo, para los que construyen programas que usan un TAD vale el siguiente llamado de atención:

> ☞ Al usar esa pila dentro de un programa, deberemos ignorar que se está trabajando sobre una lista: solamente podremos usar los métodos de la interfaz pila.

3.1. Implementación: Pilas con Listas en `Python`

Las operaciones de **lista** que proporciona `Python` son similares a las operaciones que definen a una **pila**. La interfaz no es exactamente la misma pero podemos escribir código para traducir del TAD **pila** a las operaciones integradas (built-in) de **listas** en `Python`. Este código se denomina implementación del TAD **pila**. En general, una implementación es un conjunto de métodos que satisfacen los requisitos sintácticos y semánticos de una interfaz.

Definiremos una clase `Pila` con un atributo `items`, de tipo lista, que contendrá los elementos de la pila. El tope de la pila se encontrará en la última posición de la lista, y cada vez que se apile un nuevo elemento, se lo agregará al final.

El método constructor no recibirá parámetros adicionales, ya que deberá crear una pila vacía (que representaremos por una lista vacía):

Aquí hay una implementación del TAD **pila** que usa una **lista** de `Python`:

```
class Pila:
    """ Representa una pila con operaciones de apilar, desapilar y
    verificar si está vacía. """
    def __init__(self) -> None:
        """ Crea una pila vacia """
        self.items = []

    def push(self, item: Any) -> None:
        """ Apila un elemento a la pila"""
```

```
        self.items.append(item)

    def pop(self) -> Any:
        """ Des-apila un elemento y lo devuelve.
        Si la pila esta vacía, imprime un mensaje de error y retorna
        inmediatamente """
        if self.isEmpty():
            print("La pila esta vacía")
            return
        return self.items.pop()

    def isEmpty(self) -> bool:
        """ Devuelve True si la pila esta vacía, y False si no """
        return (self.items == [])
```

Para agregar un nuevo elemento a la pila, `push` lo agrega a `items`. Para eliminar un elemento de la pila, `pop` utiliza el método homónimo del tipo de dato lista para eliminar y devolver el último elemento de la lista.

Finalmente, para comprobar si la pila está vacía, `isEmpty` compara `items` con la lista vacía.

> ☞ Utilizamos los métodos `append` y `pop` de las listas de Python, porque sabemos que estos métodos se ejecutan en tiempo constante. Queremos que el tiempo de apilar o des-apilar de la pila no dependa de la cantidad de elementos contenidos.

El uso básico de la pila se puede ejemplificar como sigue:

```
>>> p = Pila()
>>> p.isEmpty()
True
>>> p.push(1)
>>> p.isEmpty()
False
>>> p.push(5)
>>> p.push("+")
>>> p.push(22)
>>> p.pop()
22
>>> q = Pila()
>>> q.pop()
La pila esta vacía
```

Ejercicio Implementar pilas mediante listas enlazadas. Analizar el costo de los métodos a utilizar.

4. TAD - Colas (Queues)

El **TAD Cola** modela el comportamiento: “el primero que llega es el primero en ser atendido”; los demás elementos se van encolando hasta que les toque su turno.

El TAD Cola se define mediante las siguientes operaciones:

`__init__`: inicializa una nueva cola vacía.

`insert`: o encolar, agrega un nuevo elemento a la cola.

`remove`: o des-encolar, eliminar y devolver un elemento de la cola. El artículo que se devuelve es el primero que se agregó.

`isEmpty`: Comprueba si la cola está vacía.

4.1. Implementación: Colas con Listas en `Python`

Definiremos una clase `Cola` con un atributo, `items`, de tipo lista, que contendrá los elementos de la cola. El primero de la cola se encontrará en la primera posición de la lista, y cada vez que encole un nuevo elemento, se lo agregará al final. El método constructor no recibirá parámetros adicionales, ya que deberá crear una cola vacía (que representaremos por una lista vacía):

```
class Cola:
    """Representa a una cola, con operaciones de encolar y
    desencolar. El primero en ser encolado es también el primero
    en ser desencolado."""
    def __init__(self) -> None:
        """Crea una cola vacía."""
        self.items = []
```

El método `insert` se implementará agregando el nuevo elemento al final de la lista:

```
    def insert(self, x: Any) -> None:
        """Encola el elemento x."""
        self.items.append(x)
```

Para implementar `remove`, se eliminará el primer elemento de la lista y se devolverá el valor del elemento eliminado, utilizaremos nuevamente el método pop, pero en este caso le pasaremos la posición 0, para que elimine el primer elemento, no el último. Si la cola está vacía imprimimos un error y retornamos inmediatamente.

```
    def remove(self) -> Any:
        """Elimina el primer elemento de la cola y devuelve su
        valor. Si la cola está vacía, imprime un error y retorna
        inmediatamente."""
        if self.isEmpty():
            print("La cola está vacía")
            return
        return self.items.pop(0)
```

Por último, el método `isEmpty`, que indicará si la cola está o no vacía.

```
    def isEmpty(self) -> bool:
        """Devuelve True si la cola esta vacía, False si no."""
        return len(self.items) == 0
```

Veamos una ejecución de este código:

```
>>> q = Cola()
>>> q.isEmpty()
True
>>> q.insert(1)
>>> q.insert(2)
>>> q.insert(5)
>>> q.isEmpty()
False
```

```
>>> q.remove()
1
>>> q.remove()
2
>>> q.insert(8)
>>> q.remove()
5
>>> q.remove()
8
>>> q.isEmpty()
True
>>> q.remove()
La cola está vacía
```

¿Cuánto cuesta esta implementación? Dijimos en la sección anterior que usar listas comunes para borrar elementos al principio da muy malos resultados. Como en este caso necesitamos agregar elementos por un extremo y quitar por el otro extremo, esta implementación será una buena alternativa sólo si nuestras listas son pequeñas, ya que e medida que la cola crece, el método para des-encolar tardará cada vez más. Pero si queremos hacer que tanto el encolar como el desencolar se ejecuten en tiempo constante, debemos apelar a otra implementación.

4.2. Implementación: Cola Enlazada

Las colas enlazadas están formadas por nodos, donde cada nodo contiene un enlace al siguiente nodo de la cola. Además, cada nodo contiene una unidad de datos denominada carga.

Una cola enlazada es:

- la cola vacía, representada por ningún nodo (`None`), o
- un nodo que contiene un objeto de carga y una referencia a un enlace de la cola.

Aquí está la definición de clase:

```
class Queue:
    def __init__(self) -> None:
        self.length = 0
        self.head = None

    def isEmpty(self) -> bool:
        return (self.length == 0)

    def insert(self, valor: Any) -> None:
        node = Nodo(valor)
        node.prox = None
        if self.head == None:
            # si la lista está vacía, el nuevo nodo se ubica al principio
            self.head = node
        else:
            # localizar el último nodo de la lista
            last = self.head
            while last.next:
                last = last.prox
            # agregar al final el nuevo nodo
            last.prox = node
```

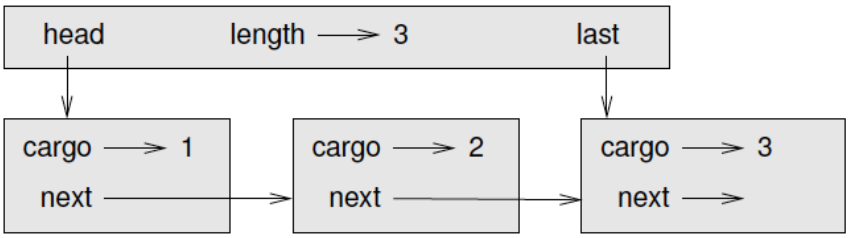

Figura 3: Cola Enlazada Mejorada

```
        self.length = self.length + 1

    def remove(self) -> Any:
        dato = self.head.dato
        self.head = self.head.prox
        self.length = self.length - 1
        return dato
```

4.2.1. Performance

Normalmente, cuando invocamos un método, no nos preocupan los detalles de su implementación. Pero hay un "detalle" que podríamos querer saber: la "performance" del método. ¿Cuánto tiempo lleva y cómo funciona? ¿Cambia el tiempo de ejecución a medida que aumenta el número de elementos de la colección?

Primero analicemos `remove`. Aquí no hay bucles ni llamadas a funciones, lo que sugiere que el tiempo de ejecución de este método es el mismo cada vez. Esta operación se ejecuta en un **tiempo constante**.

La "performance" del `insert` es muy diferente. En el caso general, tenemos que recorrer la lista para encontrar el último elemento. Este recorrido lleva un tiempo proporcional a la longitud de la cola. Dado que el tiempo de ejecución resulta una función lineal de la longitud, este método se dice que se ejecuta en un **tiempo lineal**. Comparado a un tiempo constante, esto no es muy bueno.

4.3. Implementación: Cola Enlazada Mejorada

Nos gustaría una implementación del TAD Cola que pueda realizar todas las operaciones en tiempo constante. Una forma de hacerlo es modificar la clase `Queue` para que mantenga una referencia tanto al primer como al último nodo, como se muestra en la figura 3.

La implementación de Cola Enlazada Mejorada (`ImprovedQueue`) se ve así:

```
class ImprovedQueue:
    def __init__(self) -> None:
        self.length = 0
        self.head = None
        self.last = None

    def isEmpty(self) -> bool:
        return (self.length == 0)
```

Hasta ahora, el único cambio es el atributo `last`. Se utiliza en los métodos `insert` y `remove`.

```
class ImprovedQueue:
    ...
    def insert(self, valor: Any) -> Nonee:
```

```
        node = Node(valor)
        node.next = None
        if self.length == 0:
            # si la lista está vacía, el nuevo nodo es tanto head como last
            self.head = self.last = node
        else:
            # localizar el último nodo
            last = self.last
            # agregar al final el nuevo nodo
            last.next = node
            self.last = node
        self.length = self.length + 1
```

Dado que `last` apunta siempre al último nodo, no tenemos que buscarlo. Como resultado, este método es **tiempo constante**.

Hay un precio a pagar por esa velocidad. Tenemos que agregar un caso especial en `remove` para setear a `last` en `None` cuando se elimine el último nodo:

```
class ImprovedQueue:
    ...
    def remove(self) -> Any:
        dato = self.head.dato
        self.head = self.head.next
        self.length = self.length - 1
        if self.length == 0:
            self.last = None
        return dato
```

5. TAD - Cola de Prioridad

El TAD `Cola de Prioridad` (`Priority Queue`) tiene la misma interfaz que el TAD **Cola** (`Queue`), pero diferente semántica. De nuevo, la interfaz es:

__init__: inicializa una nueva cola vacía.

insert: agrega un nuevo elemento a la cola.

remove: eliminar y devolver un elemento de la cola. El artículo que se devuelve es el que tiene mayor prioridad.

isEmpty: Comprueba si la cola está vacía.

La diferencia semántica es que el elemento que se elimina de la cola no es necesariamente el primero que se agregó. Más bien, es el elemento en la cola que tiene la máxima prioridad el cual se elimina. Cuáles son las prioridades y cómo se comparan unas con otras no está especificado por la implementación de Cola de Prioridad. Depende de cuáles sean los elementos que estén en la cola.

Por ejemplo, si los elementos de la cola tienen nombres, podríamos elegirlos según su orden alfabético. Si fuesen puntajes, podríamos ir de mayor a menor o de menor a mayor dependiendo de cómo se establezca la prioridad entre los puntajes. Mientras podamos comparar los elementos en la cola, podemos encontrar y eliminar el que tiene el mayor prioridad.

Esta implementación de `Priority Queue` tiene como atributo una lista de `Python` que contiene los elementos en la cola. Como ejemplo, utilizaremos como politica de prioridad, eliminar siempre el elemento mayor.

```
class PriorityQueue:
    def __init__(self) -> None:
        self.items = []

    def isEmpty(self) -> bool:
        return self.items == []

    def insert(self, item: Any) -> None:
        self.items.append(item)
```

Hasta ahora nada nuevo. Veamos el método `remove`:

```
class PriorityQueue:
    ...
    def remove(self) -> None:
        maxi = 0
        for i in range(1,len(self.items)):
            if self.items[i] > self.items[maxi]:
                maxi = i
        item = self.items[maxi]
        self.items[maxi:maxi+1] = []
        return item
```

Vamos a probar la implementación:

```
>>> q = PriorityQueue()
>>> q.insert(11)
>>> q.insert(12)
>>> q.insert(14)
>>> q.insert(13)
>>> while not q.isEmpty(): print(q.remove())
14
13
12
11
```

6. Resumen

- Un **Tipo de datos abstracto** o **TAD** es un tipo de datos que está definido por las operaciones que contiene y cómo se comportan (su **interfaz**), no por la forma en la que esas operaciones están implementadas.
- Una **estructura de datos** es un formato para organizar un conjunto de datos en la memoria de la computadora, de forma tal de que la información pueda ser accedida y manipulada en forma eficiente.
- Las listas de Python son una implementación del **TAD Lista**, utilizando una estructura de datos llamada arreglo. En un arreglo, es barato (tiempo constante) acceder a cualquier elemento dada su posición, y es caro (tiempo lineal) insertar o eliminar elementos.
- Una **lista enlazada** es otra implementación del TAD Lista. Se trata de una secuencia compuesta por nodos, en la que cada nodo contiene un dato y una referencia al nodo que le sigue.
- En las listas enlazadas, es barato (tiempo constante) insertar o eliminar elementos, ya que simplemente se deben alterar un par de referencias; pero es caro (tiempo lineal) acceder a un elemento en particular, ya que es necesario pasar por todos los anteriores para llegar a él.

- Una **pila** es un tipo abstracto de datos que permite agregar elementos y sacarlos en el orden inverso al que se los colocó, de la misma forma que una pila (de platos, libros, cartas, etc) en la vida real.

- Una **cola** es un tipo abstracto de datos que permite agregar elementos y sacarlos en el mismo orden en que se los colocó, como una cola de atención en la vida real.

- Las colas son útiles en las situaciones en las que se desea operar con los elementos en el orden en el que se los fue agregando, como es el caso de un cola de atención de clientes.

7. Bibliografía

Referencias

[1] Apunte de la Facultad de Ingeniería de la UBA, 2da Edición, 2016. *Algoritmos y Programación I: Aprendiendo a programar usando Python como herramienta.* Unidades 15 y 16

[2] A. Downey, J. Elkner y C. Meyers, *How to Think Like a Computer Scientist: Learning with Python*, Green Tea Press, Wellesley, Massachusetts, 2008.

[3] Alfred V. Aho, John E. Hopcroft, Jefrey D. Ullman con la colaboración de Guillermo Levine Gutiérrez; versión en español de Américo Vargas y Jorge Lozano, *Estructura de datos y algoritmos*, Addison-Wesley Iberoamericana: Sistemas Técnicos de edición, 1988.