Programación II

Estructuras de Datos y Abstracción de Datos

Universidad Nacional de Rosario.
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Resumen sobre Tipo Abtracto de Datos (TAD)

Un tipo abstracto de datos es una forma de describir el comportamiento y las propiedades de un tipo de datos, sin especificar como esta implementado.
Un tipo de datos concreto es una implementación particular utilizando una **estructura de datos** y algoritmos especifico.
Ventajas:

Resumen sobre Tipo Abtracto de Datos (TAD)

Un tipo abstracto de datos es una forma de describir el comportamiento y las propiedades de un tipo de datos, sin especificar como esta implementado.
Un tipo de datos concreto es una implementación particular utilizando una **estructura de datos** y algoritmos especifico.
Ventajas:

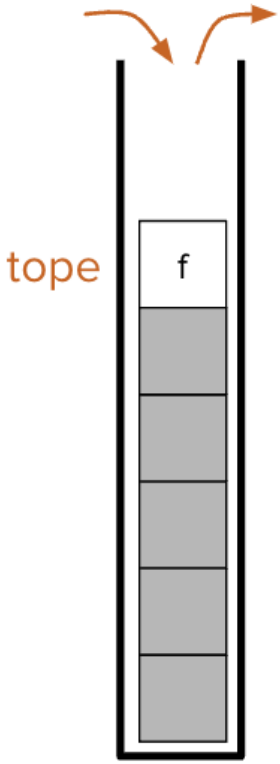
- Podemos **modularizar**: separar la lógica de implementación de un TAD de su uso.
- Podemos cambiar como esta implementado el TAD y las otras unidades del programa no deberían cambiar su funcionamiento.
- Distintas implementaciones del mismo TAD pueden tener características de performance distintas: podemos elegir la representación que más nos convenga.

Ejercicio

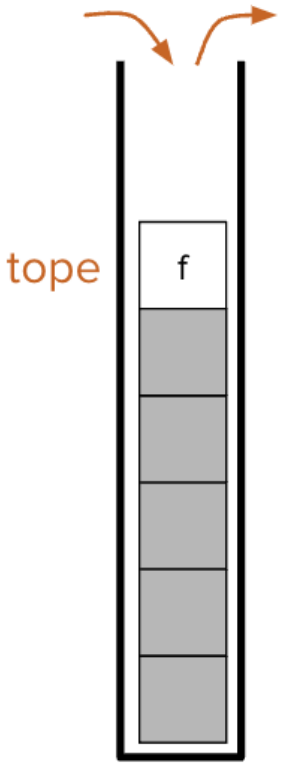
Completar el cuadro indicando la complejidad temporal de cada método en cada implementación.

Método	Lista (arreglo)	Lista (enlazada)
`insert`		
`remove`		
`append`		
`pop`		
`index`		

TAD Pila (Stack)

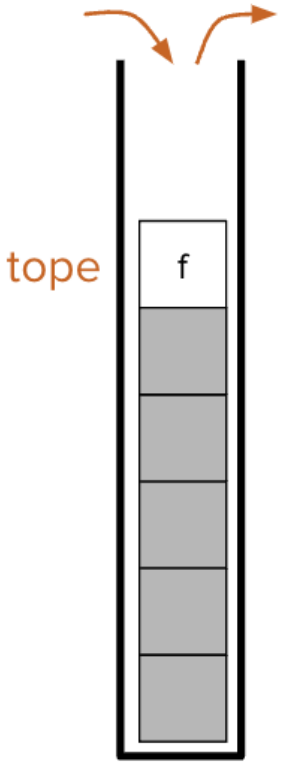


TAD Pila (Stack)



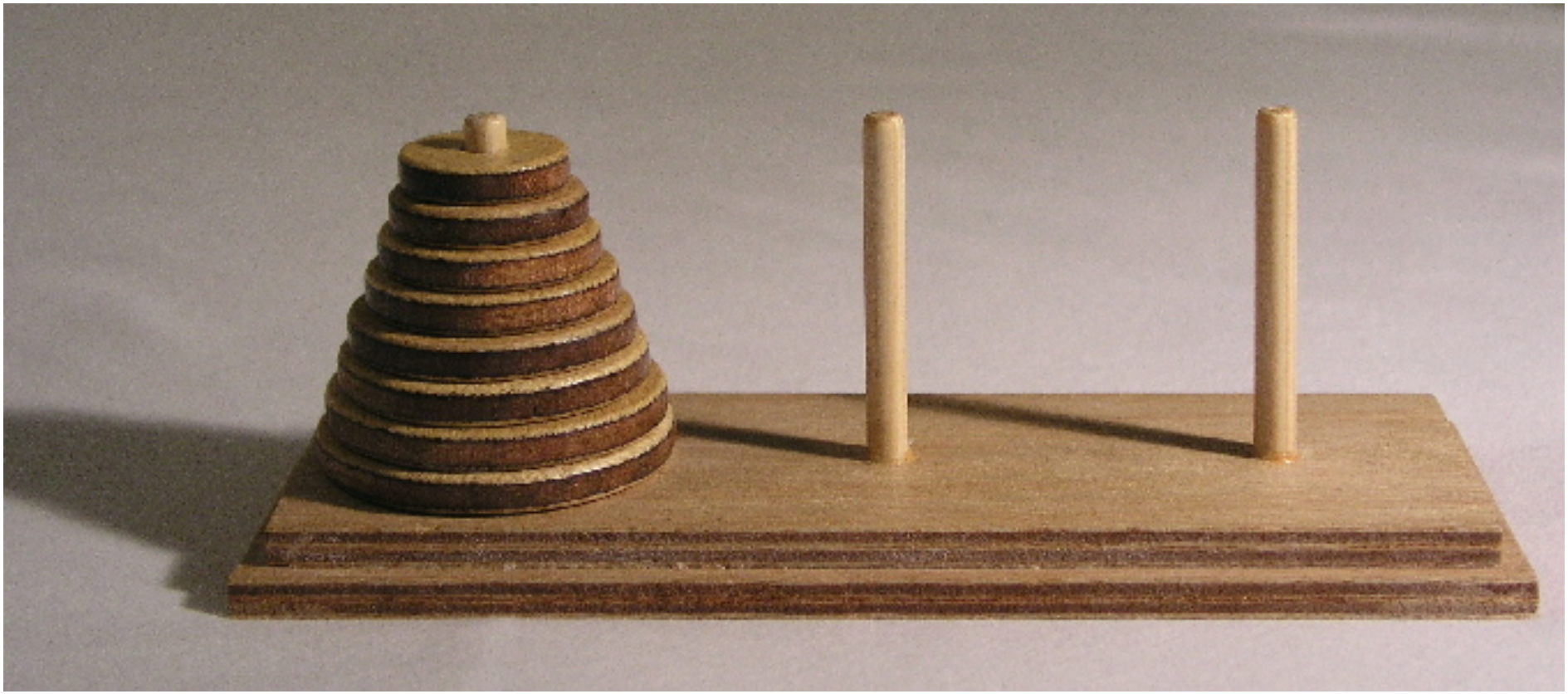
El comportamiento de una pila se puede describir mediante la frase *"Lo último que se apiló es lo primero que se usa"*, que es exactamente lo que uno hace con una pila (de platos por ejemplo): en una pila de platos uno sólo puede ver la apariencia completa del plato de arriba, y sólo puede tomar el plato 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).

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.



TAD Pila (Stack)

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

`__init__()`: inicializa una pila vacía

`push(x)`: también llamado apilar, agrega un nuevo elemento x 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.

Al crear un tipo abstracto de datos, es importante decidir cuál será la representación a utilizar.

Por ahora, representaremos una **pila** mediante una **lista** de Python.

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.

Implementación: Pilas con Listas en Python

Implementación: Pilas con Listas en Python

Las operaciones de **lista** que proporciona Python son similares a las operaciones que definen a una **pila**.

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**.

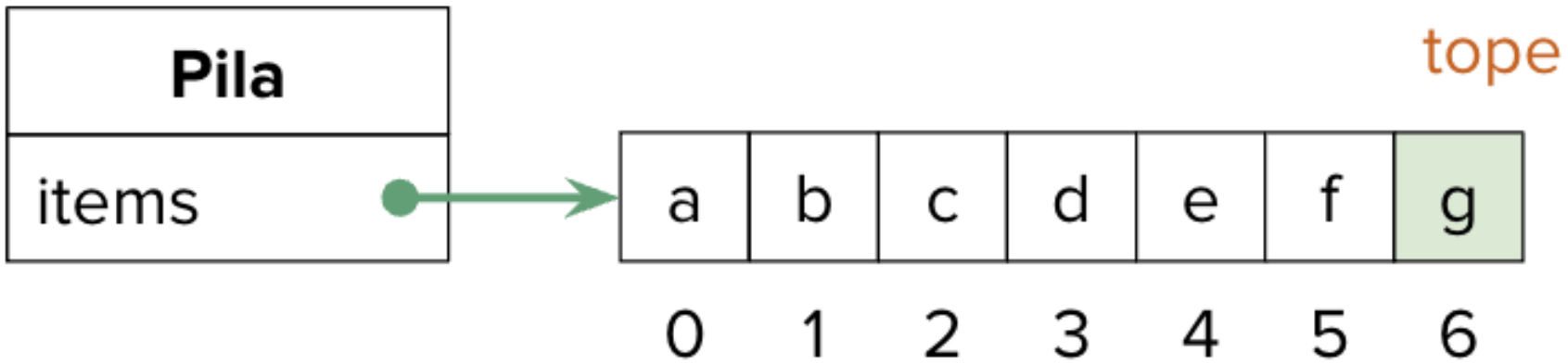
En general, una implementación es un conjunto de métodos que satisfacen los requisitos sintácticos y semánticos de una interfaz.

Implementación: Pilas con Listas en `Python`

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.

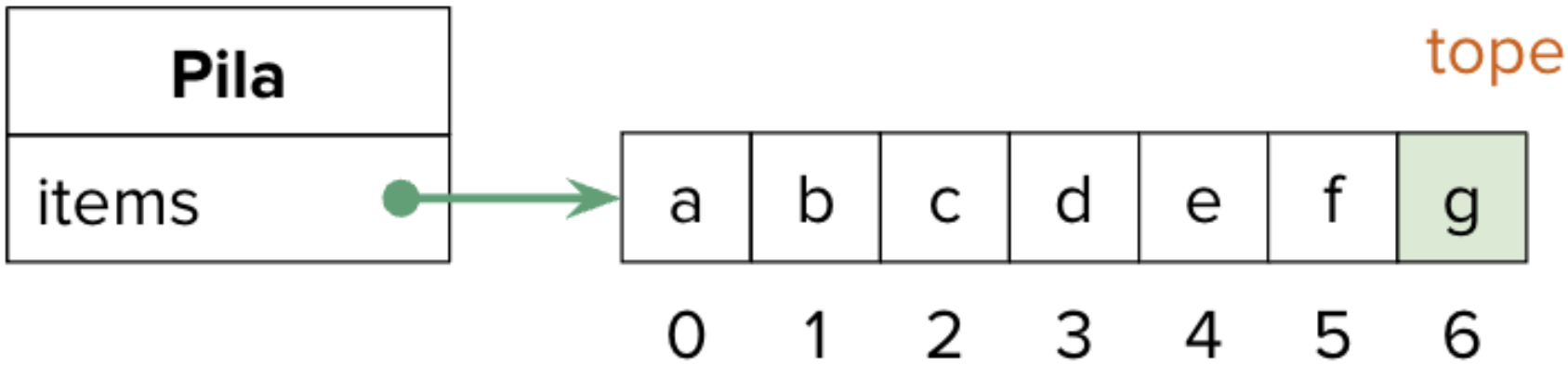
Implementación: Pilas con Listas en Python

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.



Implementación: Pilas con Listas en Python

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):

Implementación: Pilas con Listas en Python

Implementación: Pilas con Listas en 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)
    ...
```

Implementación: Pilas con Listas en 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)
    ...
```

Para agregar un nuevo elemento a la pila, push lo agrega a items.

Implementación: Pilas con Listas en 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)
    ...
```

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.

Implementación: Pilas con Listas en Python

```
class Pila:
    ...
    def pop(self) -> Any:
        """ Des-apila un elemento y lo devuelve.
        Si la pila esta vacía, imprime un mensaje
        de error y termina la ejecucion
        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 == [])
```

Implementación: Pilas con Listas en Python

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.

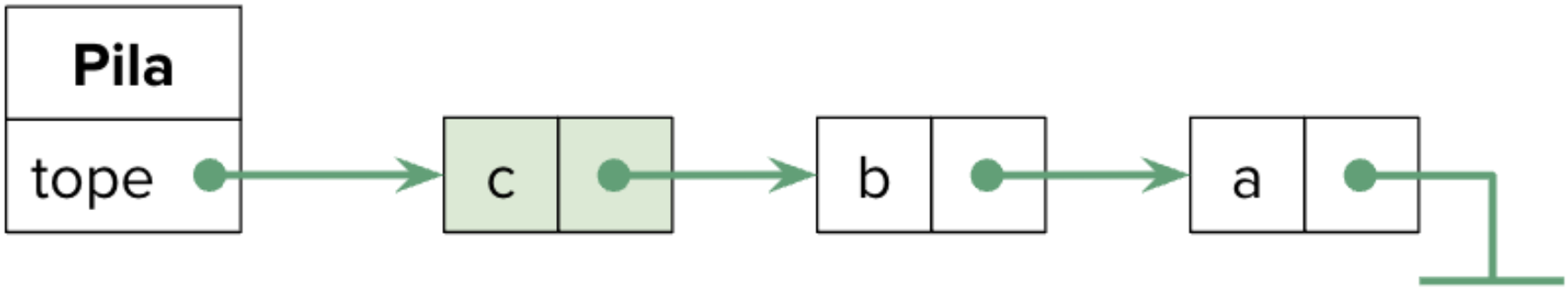
Implementación: Pilas con Listas en Python

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
```

Implementación: Pilas con Listas en Enlazadas

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



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.

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:

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.

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(x)`: o encolar, agrega un nuevo elemento x a la cola.

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(x)`: o encolar, agrega un nuevo elemento x 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.

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(x)`: o encolar, agrega un nuevo elemento x 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.

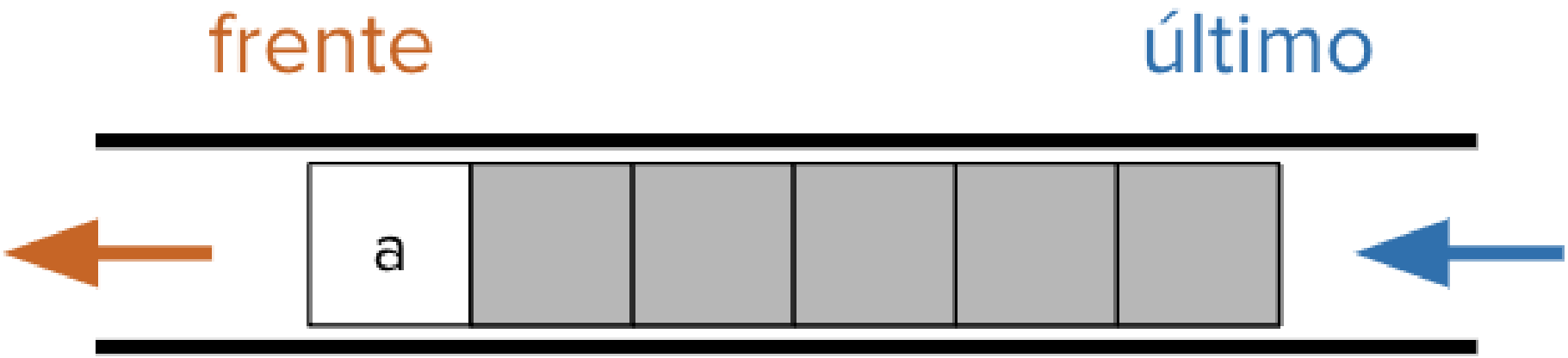
`isEmpty()`: Comprueba si la cola está vacía.

Implementación: Colas con Listas en `Python`

Definiremos una clase `Cola` con un atributo, `items`, de tipo lista, que contendrá los elementos de la cola.

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):

Implementación: Colas con Listas en Python

```
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 = []
    ...
```

Implementación: Colas con Listas en Python

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)
```

Implementación: Colas con Listas en Python

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 termina la
    ejecucion inmediatamente."""
    if self.isEmpty():
        print("La cola está vacía")
        return
    return self.items.pop(0)
```

Implementación: Colas con Listas en Python

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
```

Implementación: Colas con Listas en Python

Veamos una ejecución de este código:

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

Implementación: Colas con Listas en Python

Veamos una ejecución de este código:

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

Implementación: Colas con Listas en Python

¿Cuánto cuesta esta implementación?

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.

Implementación: Colas con Listas Enlazadas

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.

Implementación: Colas con Listas Enlazadas

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)
    ...
```

Implementación: Colas con Listas Enlazadas

```
class Queue:
    ...
    def insert(self, valor: Any) -> None:
        node = _Nodo(valor) # node.prox = None
        if self.head == None:
            self.head = node
        else:
            last = self.head
            while last.prox:
                last = last.prox
            last.prox = node
        self.length = self.length + 1
    ...
```

Implementación: Colas con Listas Enlazadas

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

Implementación: Colas con Listas Enlazadas

Performance

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.

Implementación: Colas con Listas Enlazadas Mejoradas

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:

Implementación: Colas con Listas Enlazadas Mejoradas

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`.

Implementación: Colas con Listas Enlazadas Mejoradas

```
class ImprovedQueue:
    ...
    def insert(self, valor: Any) -> Nonee:
        node = _Node(valor) #node.prox = None
        if self.length == 0:
            self.head = self.last = node
        else:
            last = self.last
            last.prox = 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**.

Implementación: Colas con Listas Enlazadas Mejoradas

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
```

TAD - Cola de Prioridad

El TAD **Cola de Prioridad** (**Priority Queue**) se utiliza para gestionar un conjunto de elementos con ciertas prioridades asociadas. Tiene la misma interfaz que el TAD **Cola** (**Queue**), pero diferente semántica.
En una cola de prioridad, los elementos se insertan en cualquier orden, pero se eliminan según su nivel de prioridad.

`__init__()`: inicializa una nueva cola vacía.

`insert(x, p)`: agrega un nuevo elemento x a la cola con la prioridad p.

`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.

Ejemplo de aplicación: Calculadora científica

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.

Resumen

- 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.

Resumen

- 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.

¿PREGUNTAS?

Referencias

Apunte de Cátedra

Elaborados por el staff docente.

Será subido al campus virtual de la materia.

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

A. Downey et al, 2002.

How to Think Like a Computer Scientist. Learning with Python.

Capítulos 12 a 16

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