6 | TDA Pila

Definición

Una *Pila* es una colección de datos caracterizada por:

1. Ser una estructura de tipo LIFO[1], esto es que el último elemento que entra a la pila es el primer elemento que sale.
2. Tener un tamaño dinámico.
3. Ser lineal.

A continuación se define el tipo de dato abstracto *Pila*.

Definición 6.1: Pila

Una *Pila* es una colección de datos tal que:

1. Tiene un número variable de elementos de tipo T.
2. Mantiene el orden de los datos ingresados, permitiendo únicamente el acceso al último (tope).
3. Cuando se agrega un elemento, éste se coloca en el tope de la pila.
4. Sólo se puede extraer al elemento en el tope de la pila.

Nombre: $\mathtt{Pila}$.
Valores: T, con $\mathtt{null} \notin$ T y $\mathbb{B}$.
Operaciones: Sea $\mathtt{this}$ la pila sobre la cual se está operando.

Constructores:

Pila(): $\emptyset \to \mathtt{Pila}$
Precondiciones: $\emptyset$
Postcondiciones:
- Se tiene una $\mathtt{Pila}$ vacía.

Métodos de acceso:

[1]Por sus siglas en inglés: *Last In First Out*.

vacía?(this) → b: $\mathrm{Pila} \to \mathbb{B}$
Precondiciones: $\emptyset$
Postcondiciones:
- $b \in \mathbb{B}$, $b = true$ si no hay elementos almacenados en la pila, $b = false$ en caso contrario.

mira(this) → e: $\mathrm{Pila} \to T$
Precondiciones: $\emptyset$
Postcondiciones:
- $e = null$ si la pila está vacía.
- $e \in T$, e es el elemento almacenado en el tope de la pila si esta no está vacía.

Métodos de manipulación:

empuja(this, e): $\mathrm{Pila} \times T \xrightarrow{?} \emptyset$
Precondiciones: $e \neq null$
Postcondiciones: Sea $\hat{e}$ el elemento que estaba en el tope de la pila.
- El elemento e es asignado al tope de la Pila.
- $\hat{e}$ queda almacenado debajo de e.

expulsa(this) → e: $\mathrm{Pila} \to (T \cup null)$
Precondiciones: $\emptyset$
Postcondiciones:
- Si la pila está vacía devuelve $null$.
- Si la pila no está vacía, sea e el elemento en el tope de la pila:
  - ⋆ Si e tenía un elemento $\hat{e}$ debajo de él, $\hat{e}$ queda en el tope de la pila, si no la pila queda vacía.
  - ⋆ e ya no está almacenado en la pila.
  - ⋆ Devuelve el elemento e.

Actividad 6.1

Revisa la documentación de la clase Stack de `Java`. ¿Cuáles serían los métodos equivalentes a los definidos aquí? ¿En qué difieren?

7 | Pila con referencias

Meta

Que el alumno domine el manejo de información almacenada en una *Pila*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Implementar el tipo de dato abstracto *Pila* utilizando nodos y referencias.

Antecedentes

Nodos

Como se ilustra en la Figura 7.1, cuando se implementa una pila utilizando referencias[1], los datos se guardan dentro de objetos llamados *nodos*. Cada nodo contiene dos piezas de información:

- El dato[2] que guarda y
- la dirección del nodo con el siguiente dato.

Una clase, a la cual nosotros llamaremos `PilaLigada<E>`, tiene un atributo escencial:

- La dirección del primer nodo, es decir, del nodo con el último dato que fue agregado a la pila. A la dirección de este nodo la llamaremos *cabeza*.

Cada vez que se quiera empujar un dato a la pila, se creará un nodo nuevo para guardar ese dato. El nuevo nodo también almacenará la dirección del nodo que solía estar a la

[1] A las referencias frecuentemente también se les llama *ligas*, por el nombre usado en inglés: *link*.
[2] Que puede consistir en el valor de un tipo primitivo o la dirección un objeto.

Figura 7.1 Representación en memoria de una pila utilizando nodos y referencias.

cabeza de la estructura, y la variable `cabeza` ahora tendrá la dirección de este nuevo nodo. Imaginemos que el nuevo dato acaba de *sumergir* un poco más a los datos anteriores (los empujó más lejos). Esos datos no volverán a ser visibles hasta que el dato en la cabeza haya sido expulsado.

Para expulsar un dato se realiza el procedimiento inverso: la cabeza volverá a guardar la dirección del nodo siguiente y se devolverá el valor que estaba guardado en el nodo *de hasta arriba*, a la vez que se descarta el nodo que contenía al dato. Cuidado: al realizar estas operaciones en código es importante cuidar el orden en que se realizan, para no perder datos o direcciones en el proceso. A menudo requerirás del uso de variables temporales, para almacenar un dato que usarás después. Pero recuerda: las variables temporales deben desaparecer cuando se termina la ejecución de un método, es decir, deben ser variables locales. Asegúrate de guardar todo lo que deba permanecer en la pila en atributos de objetos, ya sea en la `PilaLigada<E>` o algún `Nodo` adecuado.

Iterador

Para recorrer uno por uno los elementos almacenados en la pila se utiliza un objeto *iterador*. Este objeto es instancia de una clase que implementa la interfaz `Iterator<E>` y que tiene acceso a los componentes de la pila. La forma más natural de obtener esto es implementar la interfaz en una clase interna a la pila.

Actividad 7.1

Revisa la documentación de la interfaz Iterator<E>. Sólo hay dos métodos cuya implementación es obligatoria: `hasNext` y `next`; el método `remove` tiene una implementación por defecto que lanza una excepción. Implementar o no este método depende de cada estructura. Asegúrate de que te queda claro qué hace cada uno de estos métodos.

La idea básica para programar al iterador en el caso de una pila con referencias se muestra en la Figura 7.2. El objeto iterador guarda la dirección del nodo que sería devuelto por `next` y la actualiza para apuntar al nodo siguiente cada vez que se manda llamar este método. Cuando ya no hay nodo siguiente esta referencia vale `null` y por ello `hasNext` debe devolver `false`. El constructor del iterador inicializa la referencia con la dirección de la `cabeza`.

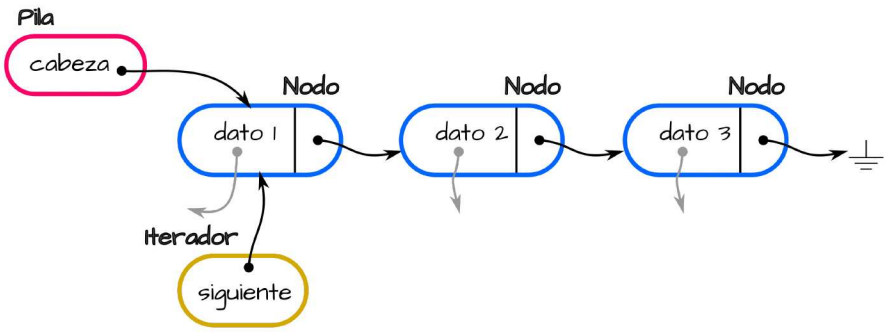

Figura 7.2 Objeto iterador apuntando al elemento que sería devuelto por una llamada a `next`.

Desarrollo

Se implementará el TDA `Pila` utilizando nodos y referencias. Para esto se deberá implementar la intefaz `Pila<E>` y extender `ColecciónAbstracta<E>`. Asegúrate de que tu implementación cumpla con las condiciones indicadas en la documentación de la intefaz. Por razones didácticas, no se permite el uso de ninguna clase que se encuentre en la API de `Java`, por ejemplo clases como `Vector<E>`, `LinkedList<E>` o cualquier otra estructura del paquete `java.util`.

1. Programa la clase `Nodo<E>`.

   Puedes crear esta clase dentro del paquete `ed.estructuras.lineales`. Esto te permitirá reutilizarlo cuando programes la siguiente estructura: la cola. Si eliges esta opción, dale acceso de paquete (es decir, la declaración de la clase omite el acceso e inicia con `class Nodo<E>...` en lugar de `public class Nodo...`). Esto es para no confundir este nodo con otros nodos que utilizarán futuras estructuras y que tienen características diferentes. Otra opción es programarlo como una clase estática interna de `PilaLigada<E>`, pero en ese caso, sólo la pila podrá usarlo.

   Como esta pila no almacena `null`, `Nodo<E>` deberá lanzar `NullPointerException` cada vez que se intente almacenar este valor, ya sea mediante el constructor o un método de escritura.

2. Programa la clase `PilaLigada<E>`.

   - Implementa la interfaz `Pila<E>` y
   - extiende `ColecciónAbstracta<E>`. Necesitarás agregar tu implementación de `ColecciónAbstracta<E>` en el paquete `ed.estructuras` e importarla en `PilaLigada.java`. Observa que sólo falta implementar:
     - ⋆ `public Iterator<E> iterator()`
     - ⋆ `public boolean add(E e)` Se convierte en sinónimo de `empuja` y siempre devuelve `true` pues siempre debe poder agregar un elemento no nulo.

- ⋆ `public void clear()` Es necesario sobreescribir este método, porque tu iterador no tendrá operación `remove()`, así que la implementación de la superclase no funcionará.
- ⋆ `public boolean remove(Object o)` Compara `o` sólo con el objeto devuelto por `mira()`, si son iguales lo remueve, si no devuelve `false`. Si `o` es `null` lanza `NullPointerException`.

TIP: No olvides actualizar la variable `tam`, que heredas de `ColecciónAbstracta`.

Sin embargo, dado que en una pila no operaremos sobre elementos que no están en el tope, los métodos siguientes deberán ser sobreescritos para que lancen `UnsupportedOperationException`:

- ⋆ `public boolean retainAll(Collection<?> c)`
- ⋆ `public boolean removeAll(Collection<?> c)`

Inicia con los métodos básicos. Más aún, crea todos los métodos necesarios para que tu clase compile y se puedan ejecutar las pruebas, devuelve valores por defecto, luego irás programando su contenido.

3. Agrega el iterador que requiere `Collection<E>`. Puedes programar al iterador como una clase interna, en este caso debes implementar la interfaz `Iterator<E>` pero no es necesario que tu clase declare una nueva variable de tipo, utiliza el valor de `<E>` en la clase `PilaLigada`. Esto se vería así:

```
import java.util.Iterator;
...
public class PilaLigada<E> ... {
  private class Iterador implements Iterator<E> {
    ...
  }
}
```

Aunque en una pila sólo se pueden agregar y remover elementos en un extremo, necesitaremos un iterador que permitar ver todos los elementos en la pila, desde el último insertado hasta el primero. Para esta práctica sólo programarás un constructor, que inicialice el iterador en el tope de la pila (`cabeza`), y los métodos `next` y `hasNext` del iterador.

Preguntas

1. Explica, para esta implementación, cómo funciona el método `empuja`.
2. ¿Cuál es la complejidad, en el peor caso, de los métodos `mira`, `expulsa` y `empuja`?