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 C#. ¿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.

Código Auxiliar 7.1: Pila ligada

https://github.com/computacion-ciencias/ed-pila-ligada-cs

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<T>`, tiene un atributo escencial:

[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.

- 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 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<T>` 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 `IEnumerator<T>` y que tiene acceso a los componentes de la pila.

Implementación directa

La idea básica para programar al iterador en el caso de una pila con referencias se muestra en la Figura 7.2. Se programa un método con el encabezado siguiente:

```
1 public IEnumerator<T> GetEnumerator()
```

el cual se encarga de crear un objeto de tipo `Enumerator<T>`. Para ello debes crear una clase que implemente dicha interfaz, e indique el nodo siguiente al que se moverá el

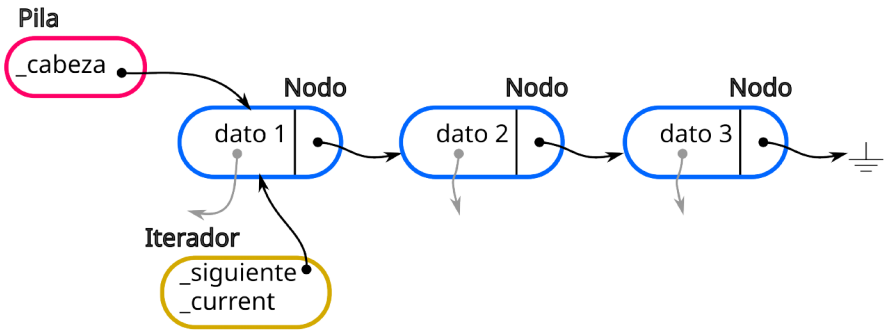

Figura 7.2 Variable temporal apuntando al elemento que será devuelto en la siguiente llamada del `foreach`.

iterador y el valor a devolver en su estado actual.

NOTA: La interfaz `IEnumerable` también exige que se implemente la versión no genérica:

```
IEnumerator IEnumerable.GetEnumerator()
```

aunque el funcionamiento es el mismo.

Produce

Dentro del método `GetEnumerator` se utiliza una variable temporal de tipo `Nodo` que irá guardando la dirección del nodo que contiene al siguiente dato a devolver. El valor inicial de esta variable debe ser la dirección guardada en la variable `_cabeza`. Luego se utiliza un ciclo para ir recorriendo los nodos. En el cuerpo del ciclo se utiliza `yield return` para devolver el valor del nodo al que apunta `temp` y luego se actualiza esta variable para que almacene la dirección del nodo siguiente. El ciclo termina cuando `temp` valga `null`.

La forma más natural de obtener esto es hacer uso de la instrucción `yield return` (produce y devuelve). El código siguiente muestra cómo funciona esta construcción para generar un iterador que vaya devolviendo los números en una secuencia:

Listado 7.1: Creación de un iterador con yield return

```
public IEnumerator<int> Secuencia(int fin)
{
  if(fin < 0) throw new ArgumentException();
  for(int i = 0; i < fin; i++)
  {
    yield return i;
  }
}
```

Luego se puede utilizar la estructura de control `foreach` para obtener los números uno por uno:

```
public static void Main()
{
  foreach(int num in Secuencia(5))
  {
    Console.WriteLine($"Número␣{num}");
  }
}
```


Desarrollo

Se implementará el TDA Pila utilizando nodos y referencias. Para esto se deberá implementar la intefaz `IPila<T>`, que extiende `IColección<T>`, la cual extiende a su vez a `ICollection<T>`, que extiende `IEnumerable<T>`, que extiende a `IEnumerable`. 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 C#o `.NET`.

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

   Puedes crear esta clase dentro del proyecto `Pila`, en el espacio de nombres

   `ED.Estructuras.Lineales`.

   Esto te permitirá reutilizarlo cuando programes la siguiente estructura: la cola. Si eliges esta opción, dale acceso de `internal` y hazla genérica `internal class Nodo<T>...` 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<T>`, pero en ese caso, sólo la pila podrá usarlo..

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

   - Implementa la interfaz `IPila<T>`.

     TIP: Tendrás que implementar una propiedad `Count`, no olvides actualizarla cada vez que modifiques la estructura.

     También te pedirá una propiedad `IsReadOnly`, haz que siempre valga `false` puesto que esta estructura es para ser modificada.

   - Observa que también hay que implementar los métodos que se heredan de `IColección`:

     ⋆ `public void Add(T item)` Se convierte en sinónimo de `Empuja`.

- `public void Clear()` Vacía la pila.
- `public bool Remove(T item)` Compara `item` sólo con el objeto devuelto por `Mira()`, si son iguales lo remueve, si no devuelve `false`. Cuidado si `item` es `null`.
- `public IEnumerator<T> GetEnumerator()` y
- `IEnumerator IEnumerable.GetEnumerator()`. Puedes programar sólo la versión genérica y en la otra sólo mandarla llamar con:

```
1       IEnumerator IEnumerable.GetEnumerator()
2       {
3           return GetEnumerator();
4       }
```

  Implementa tu iterador **creando una clase que implemente** `IEnumerator<T>` 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. Inicializa el iterador en el tope de la pila (`cabeza`) y recórrela hasta el final.
- `public bool Contains(T item)`. Puedes utilizar el iterador para ver si el elemento está en algún lugar de la pila.
- `public void CopyTo(T[] array, int arrayIndex)`. Copia los elementos de la pila en el arreglo indicado.

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.

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`?