Práctica 5
Pila con referencias

Estructuras de datos

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

Una `Pila` es una estructura de datos caracterizada por:

1. El último elemento que entra a la `Pila` es el primer elemento que sale.
2. Tiene un tamaño dinámico.

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

Definición 1: Pila

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

1. Tiene un número variable de elementos de tipo T.
2. Cuando se agrega un elemento, éste se coloca en el tope de la `Pila`.
3. Sólo se puede extraer al elemento en el tope de la `Pila`.

Nombre: $\mathtt{Pila}$.
Valores: $\mathbb{N}$, T, con $\mathtt{null} \in$ T.
Operaciones: sea `this` la pila sobre la cual se está operando.
Constructores :
Pila(): $\emptyset \rightarrow \mathtt{Pila}$
Precondiciones: $\emptyset$
Postcondiciones :
- Una $\mathtt{Pila}$ vacía.

Métodos de acceso :
mira(this) $\rightarrow$ **e:** $\mathtt{Pila} \times \mathbb{N} \rightarrow$ T

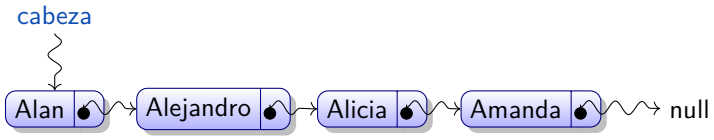

Figura 1 Representación en memoria de una pila, utilizando nodos y referencias.

Precondiciones: $\emptyset$
Postcondiciones :
- $e \in \mathrm{T}$, e es el elemento almacenado en el tope de la Pila.

Métodos de manipulación :

expulsa(this) $\rightarrow$ **e:** $\mathrm{Pila} \rightarrow \mathrm{T}$
Precondiciones :$\emptyset$
Postcondiciones :
- Elimina y devuelve el elemento e que se encuentra en el tope de la Pila.

empuja(this, e): $\mathrm{Pila} \times \mathrm{T} \xrightarrow{?} \emptyset$
Precondiciones: $\emptyset$
Postcondiciones :
- El elemento e es asignado al tope de la Pila.

Actividad 1

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

Como se ilustra en la Figura 1, en esta implementación los datos se guardan dentro de objetos llamados *nodos*. Cada nodo contiene dos piezas de información:

- El dato[1] 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.

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 depués. 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.

[1] O la dirección del dato, si se trata de un objeto.

DESARROLLO

Se implementará el TDA `Pila` utilizando nodos y referencias. Para esto se deberá implementar la intefaz `IPila<E>` y extender `ColeccionAbstracta<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 el api de Java, por ejemplo clases como `Vector<E>`, `LinkedList<E>` o cualquiera otra estructura del paquete `java.util`.

1. Programa la clase `Nodo`..

   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.

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

   - Implementa la interfaz `IPila<E>` y
   - extiende `ColeccionAbstracta<E>`.

   Inicia con los métodos básicos.

3. Luego 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, la `<E>`. Esto se vería así:

```
1 import java.util.Iterator;
2 ...
3 public class PilaLigada<E> ... {
4   private class Iterador implements Iterator<E> {
5     ...
6   }
7 }
```

   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, 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`?

EXTRA

Una aplicación de las pilas es el algoritmo conocido como *backtracking*. Se utiliza para buscar soluciones a problemas en forma sistemática. En esta sección se utilzará una pila para resolver el problema de las *n-reinas* utilizando backtracking (Main 2003).

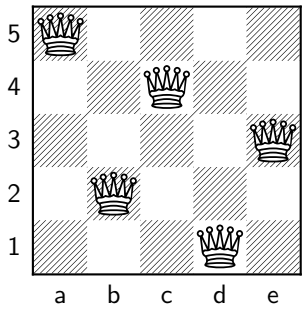

Figura 2 Solución al problema para 5-reinas.

Problema de las n-reinas

Dado un tablero de ajedrez de $n \times n$ casillas, se desea colocar n reinas sin que se coman las unas a las otras. [2] En la Figura 2 se muestra la solución para un tablero 5×5.

Ejercicio

Harás un programa que, dado el número n de reinas, determine si existe una solución para el problema en un tablero $n \times n$. En caso de existir imprimirá la columna y renglón donde se debe colocar cada reina, de lo contrario imprimirá un aviso indicando que no existe solución para ese tamaño del tablero.

El algoritmo funciona de la siguiente manera: para que las reinas no se coman, deben estar en renglones distintos. Por lo tanto, ya sabes que debes colocar una reina por renglón y falta averiguar en qué columna. Irás probando a colocar las reinas una por una de izquierda a derecha incrementando renglón por renglón. A continuación se incluye el pseudocódigo correspondiente. Tu labor es implementarlo en `Java` utilizando la pila que acabas de programar, con su iterador. Nota que los renglones se cuentan a partir de 1 y las columnas se indican con letras, puedes representar internamente a las columnas con números, si así lo prefieres, pero el programa debe imprimir los resultados usando la notación con caracteres.

[2] Recuerda que una reina en el ajedrez se puede comer a las piezas que se encuentran en la misma columna, mismo renglón o sobre cualquiera de las dos diagonales.

Algoritmo 1 Backtracking N-Reinas.

```
1: function RESUELVENREINAS(n)
2:     pila ←′ a′                                   ▷ Iniciamos con (1, a)
3:     while pila ≠ ∅ do
4:         if La última reina agregada es comida por alguna de las anteriores then
5:             while pila ≠ ∅ y mira(pila) = n do   ▷ Se acabaron las opciones
6:                 expulsa(pila)                    ▷ Regresa un renglón
7:             if pila ≠ ∅ then
8:                 mira(pila) ← mira(pila) + 1      ▷ Prueba la siguiente columna
9:         else if tamaño(pila) = n then
10:            return pila                          ▷ Hay n reinas en su lugar
11:        else
12:            pila ←′ a′                           ▷ Avanza un renglón
13:    if pila = ∅ then return Fallo
14:    elsereturn pila
```

1. Agrega una clase en el paquete `ed.aplicaciones`. En esta clase implementarás el algoritmo y deberás incluir un método `main` para ejecutarlo. Eres libre para diseñar tu clase, pero procura seguir las buenas prácticas de orientación a objetos.
2. Modifica el archivo `build.xml` para que al escribir `ant run` se ejecute tu programa.
3. Prueba tu código para varios valores de n y verifica que las soluciones encontradas sean válidas, añade algunos ejemplos a tu reporte.

Este es un ejemplo de solución, que corresponde a la Figura 3:

Listing 1: Solución 5x5

```
Tablero 5x5
Renglón 5, columna e
Renglón 4, columna c
Renglón 3, columna f
Renglón 2, columna d
Renglón 1, columna b

Tablero 2x2
No hay solución
```

Un tablero 2×2 es un ejemplo de n para la cual no existe solución.

FORMA DE ENTREGA

- Asegúrense de borrar todos los archivos generados, incluyendo archivos de respaldo usando `ant clean`.
- Copien el directorio `Practica5` dentro de un directorio llamado `<apellido1_apellido2>` donde `apellido1` es el primer apellido de un miembro del equipo y `apellido2` es el primer apellido del otro miembro. Por ejemplo: `marquez_vazquez`.
- Borren el directorio libs en la copia.

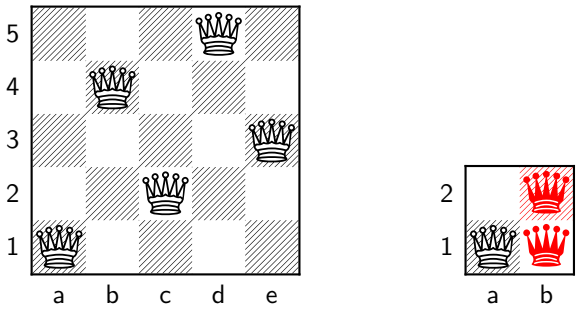

Figura 3 Otra solución al problema para 5-reinas, obtenida con *backtracking*. En el tablero 2×2 no hay solución.

- Agreguen un archivo `reporte.pdf` dentro del directorio `Practica5` de la copia, con el nombre completo de los integrantes del equipo, las repuestas a las preguntas de la sección anterior y cualquier comentario que quieran hacer sobre la práctica.
- Compriman el directorio `<apellido1_apellido2>` creando

  `<apellido1_apellido2>.tar.gz`.

  Por ejemplo: `marquez_vazquez.tar.gz`.
- Suban este archivo en la sección correspondiente del aula virtual. Basta una entrega por equipo.

Forma de evaluación

Para su calificación final se tomarán en cuenta los aspectos siguientes:

70 %	Calificación generada por las pruebas automáticas. Usaremos nuestros archivos, por lo que si realizan modificaciones a sus copias, éstas no tendrán efecto al momento de calificar.
10 %	Reporte con respuestas a las preguntas.
10 %	Documentación completa y adecuada. Entrega en el formato requerido, sin archivos `.class`, respaldos, bibliotecas de `JUnit` u otros no requeridos.
10 %	Revisión manual del código, para verificar que se cumpla con las especificaciones, que no se haya copiado, etcétera.

Referencias

Main, Michael (2003). *Data Structures & Other Objects Using Java*. 2nd. Pearson Education, Inc. 808 págs.