Estructuras de Datos. ATD y estructuras lineales

Profesores Estructuras de Datos, 2024.

Dpto. Lenguajes y Ciencias de la Computación.

University of Málaga

Tipos de datos abstractos (ADT): definición

- **Definición**: Un modelo para tipos de datos definidos por *comportamiento* e *interacción*, no por implementación.

Aspectos clave de los ADT

- **Valores**: ¿Cuál es el rango de valores que pueden contener los datos?
- **Operaciones**: ¿Qué operaciones se pueden realizar sobre los datos?
- **Comportamiento**: ¿Cómo interactúan estas operaciones con los datos?

ADT frente a estructura de datos

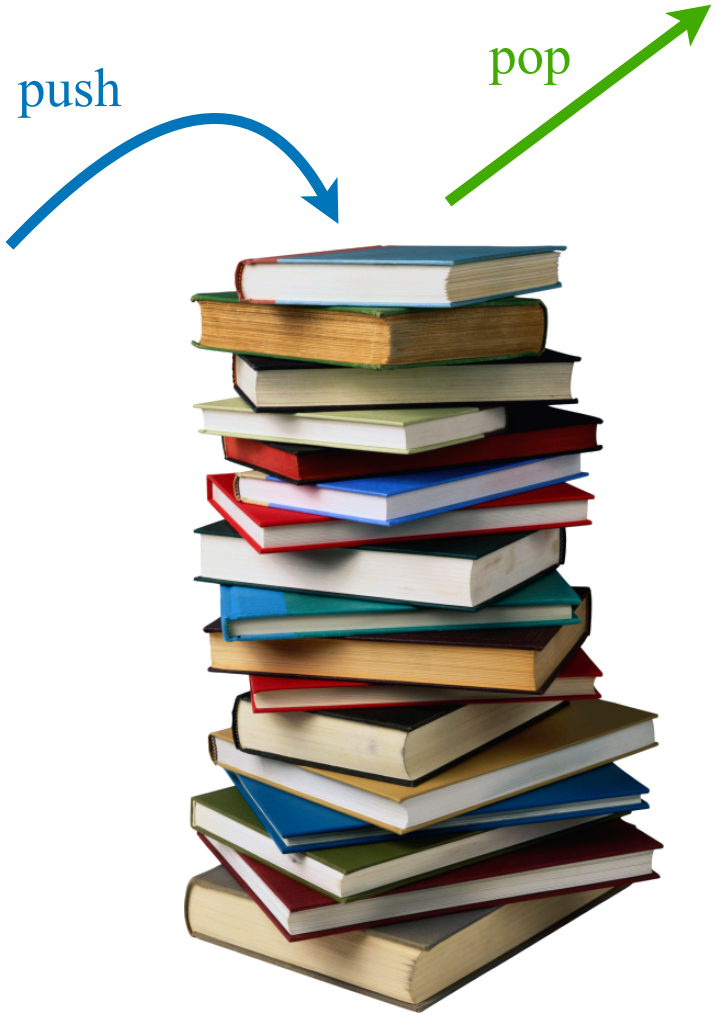
- **ADT**: Concepto teórico, centrado en el '*qué*'.
- **Estructura de datos**: Implementación práctica, centrándose en el '*cómo*'.

Beneficios de los tipos de datos abstractos (ADT)

- **Encapsulación**: Los ADT encapsulan los datos y proporcionan una *interfaz* a las operaciones.
- **Abstracción**: Nos permiten centrarnos en las operaciones sin considerar los detalles de implementación.
- **Reutilización**: los ADT se pueden implementar de múltiples maneras, lo que proporciona flexibilidad para elegir la mejor *implementación* para un problema determinado.

El TAD Pila (1/2)

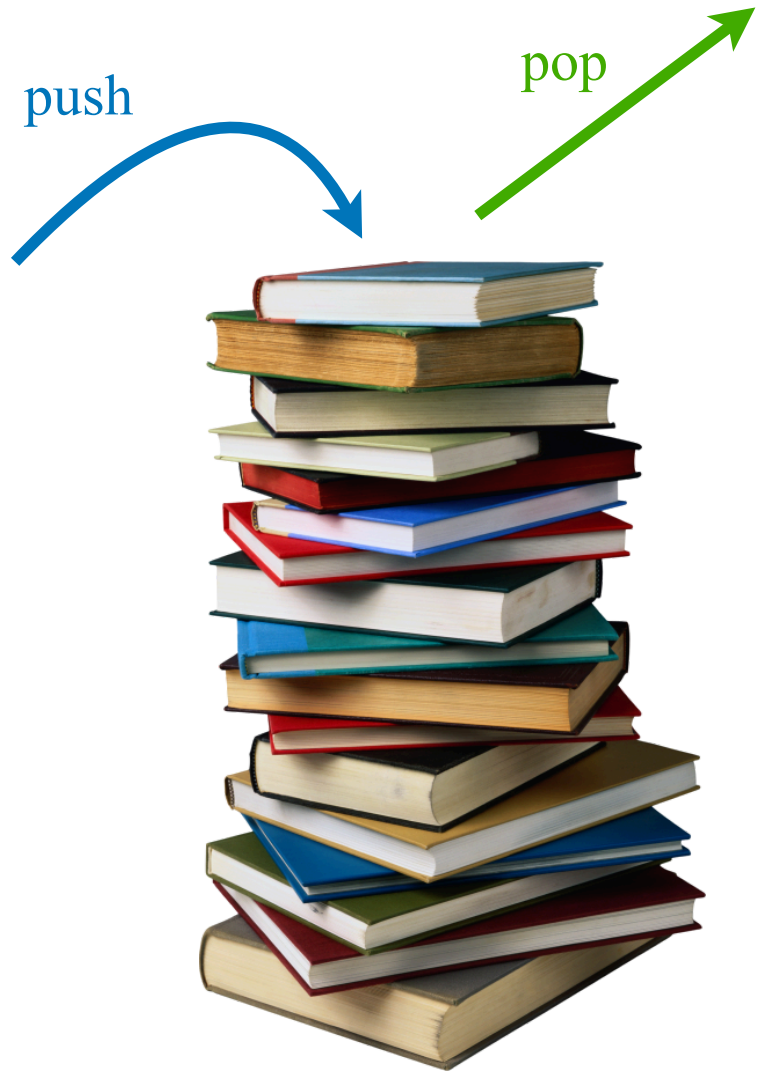
- Una *pila* es una colección que almacena elementos en un orden de último en entrar, primero en salir (*LIFO*).
- **Operaciones**:
  - `push` : agrega un elemento a la parte superior de la pila.
  - `pop` : elimina el elemento superior de la pila.
  - `top` : Devuelve el elemento superior de la pila sin eliminarlo.



Stack of books

El TAD Pila (2/2)

- **Operaciones**:
  - `isEmpty` : Comprueba si la pila está vacía.
  - `size` : Devuelve el número de elementos en la pila.
  - `clear` : elimina todos los elementos de la pila.



Stack of books

El TAD de pila en Java (1/2)

- En Java, los ADT normalmente se especifican mediante *interfaces*.
- La interfaz `Stack<T>` define una pila con elementos de tipo `T`.

```java
package dataStructures.stack;

public interface Stack<T> {
  void push(T element);
  T top();
  void pop();
  boolean isEmpty();
  int size();
  void clear();
}
```

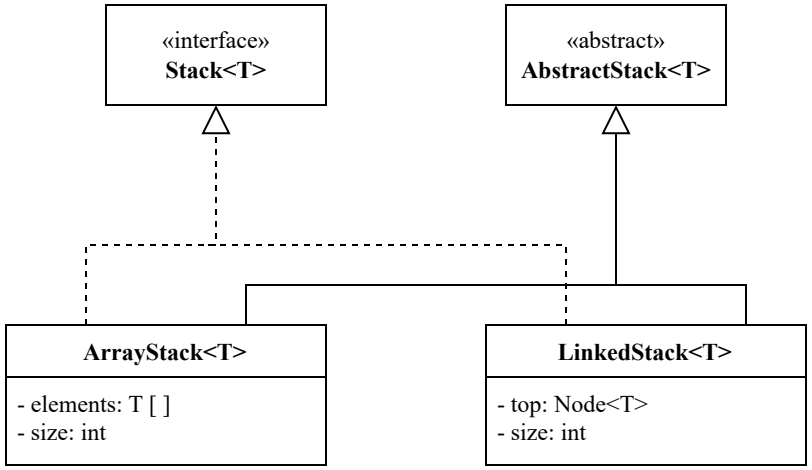
El TAD de pila en Java (2/2)

- Las partes comunes a toda implementación, se pueden elevar a una clase abstracta.
- La clase `AbstractStack<T>` define una pila con elementos de tipo `T`.

```
public abstract class AbstractStack<T> {
  protected abstract Iterable<T> elements(); //Abstracto
  public abstract int size(); //Abstracto
  public boolean equals(Object obj);
  public int hashCode();
  public String toString();
```

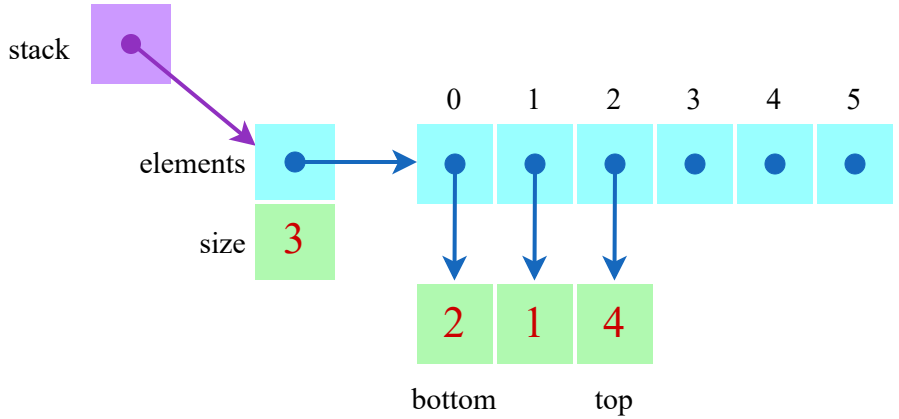
Implementaciones del TAD de pila

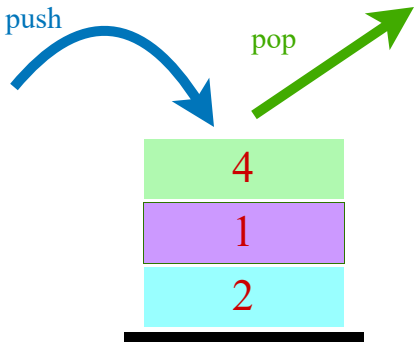
- Una pila se puede implementar utilizando diferentes *estructuras de datos*.
- Diferentes *clases* pueden implementar la interfaz `Stack<T>`:
  - `ArrayStack<T>`: utiliza un array para almacenar elementos.
  - `LinkedStack<T>`: utiliza una estructura vinculada para almacenar elementos.



La clase `ArrayStack`

- `ArrayStack<T>` implementa la interfaz `Stack<T>` utilizando un array.
- Inicialmente tiene un tamaño fijo (*capacidad* de la pila), pero puede crecer dinámicamente cuando sea necesario.
- Los nuevos elementos en la pila se almacenan en orden *de izquierda a derecha*.
- La clase también mantiene una variable entera `size` para:
  - Conocer la cantidad de elementos en la pila.
  - Saber la *primera posición libre* en el array.





Especificación de un `ArrayStack` (1/2)

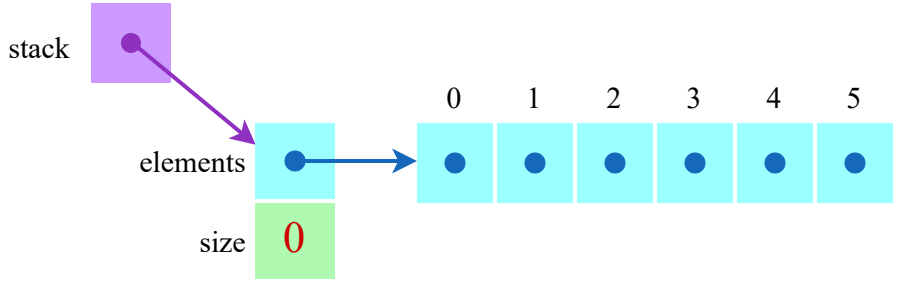
```
public class ArrayStack<T> extends AbstractStack<T> implements Stack<T> {

  private static final int DEFAULT_INITIAL_CAPACITY = 16;
  private T[] elements;
  private int size;

  public ArrayStack(int initialCapacity) {
    if (initialCapacity <= 0) throw new IllegalArgumentException("initial capacity must be greater than 0");
    elements = (T[]) new Object[initialCapacity];
    size = 0;
  }
  public ArrayStack() {
    this(DEFAULT_INITIAL_CAPACITY);
  }
  public static <T> ArrayStack<T> empty(){
    return new ArrayStack<>();
  }
  public static <T> ArrayStack<T> withCapacity(int initialCapacity){
    return new ArrayStack<>(initialCapacity);
  }
  ...
}
```

Especificación de un `ArrayStack` (2/2)

```
public class ArrayStack<T> extends AbstractStack<T> implements Stack<T> {
  ...
  //Factory methods
  @SafeVarargs
  public static <T> ArrayStack<T> of(T... elements); //elements es una colección
  public static <T> ArrayStack<T> from(Iterable<T> iterable);
  public static <T> ArrayStack<T> copyOf(ArrayStack<T> that);
  public static <T> ArrayStack<T> copyOf(Stack<T> that);

  protected Iterable<T> elements(); //Abstract
  public int size(); //Abstract && Interface
  public void push(T element); //Interface
  public T top(); //Interface
  public void pop(); //Interface
  public boolean isEmpty(); //Interface
  public void clear(); //Interface
}
```

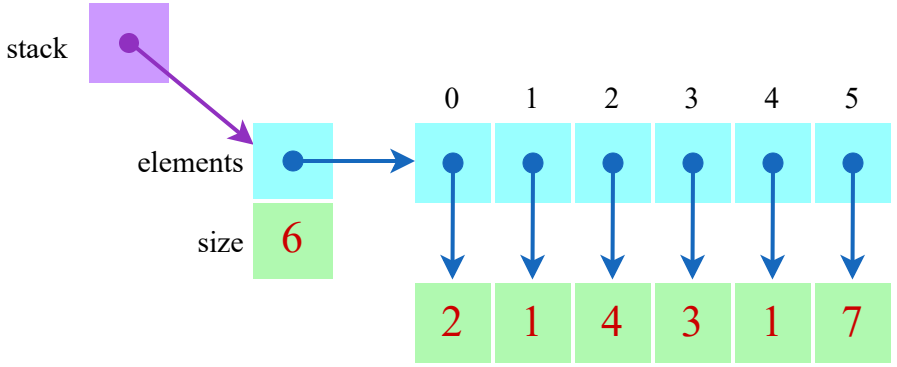
Inicialización de `ArrayStack`

- Suponiendo que la *capacidad inicial* es 6, se crea el array. Cuando se construye un array de tipos de referencia con `new`, la máquina virtual Java (JVM) inicializa automáticamente todos los elementos con `null`.

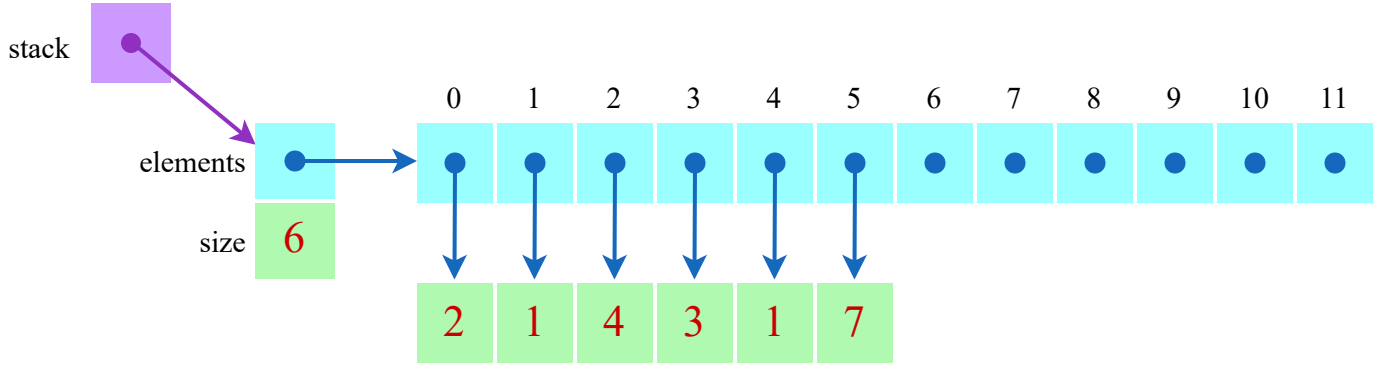


Garantizar la capacidad en `ArrayStack` (`ensureCapacity`)

- Al alcanzar su capacidad máxima, el conjunto necesita ser ampliado. Se construye un nuevo conjunto con *el doble* de capacidad.



- El método `Arrays.copyOf(oldArray, newLength)` asigna una nueva matriz que conserva todos los elementos de `oldArray` y expande su capacidad a `newLength`.



Métodos de fábrica para `ArrayStack` (1/2)

- Los métodos de fábrica ofrecen una forma conveniente de crear instancias de objetos `ArrayStack<T>` sin invocar directamente constructores.
  Estos métodos incluyen:
  - `empty()`: construye una *pila vacía* con una capacidad inicial *predeterminada*, adecuada para cuando se desconoce el número esperado de elementos.
  - `withCapacity(int initialCapacity)`: construye una *pila vacía* con una capacidad inicial especificada, optimizando la asignación de memoria para un número conocido de elementos.
  - `of(T... elementos)`: construye una pila *previamente rellena* con los elementos proporcionados, lo que permite una configuración de pila rápida y sencilla.

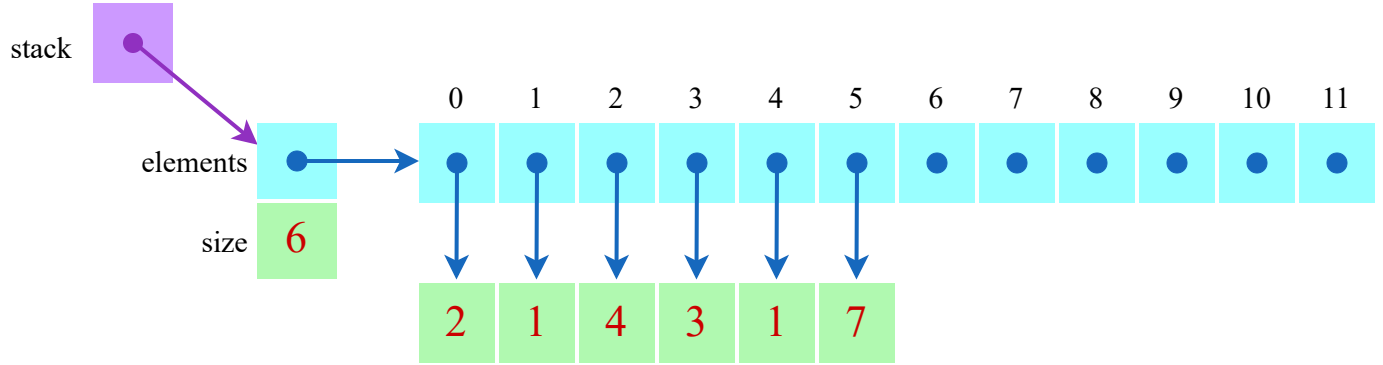
Métodos de fábrica para `ArrayStack` (2/2)

- `copyOf(Stack<T> stack)`: construye una nueva pila que es un *duplicado* de la `pila` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva pila que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

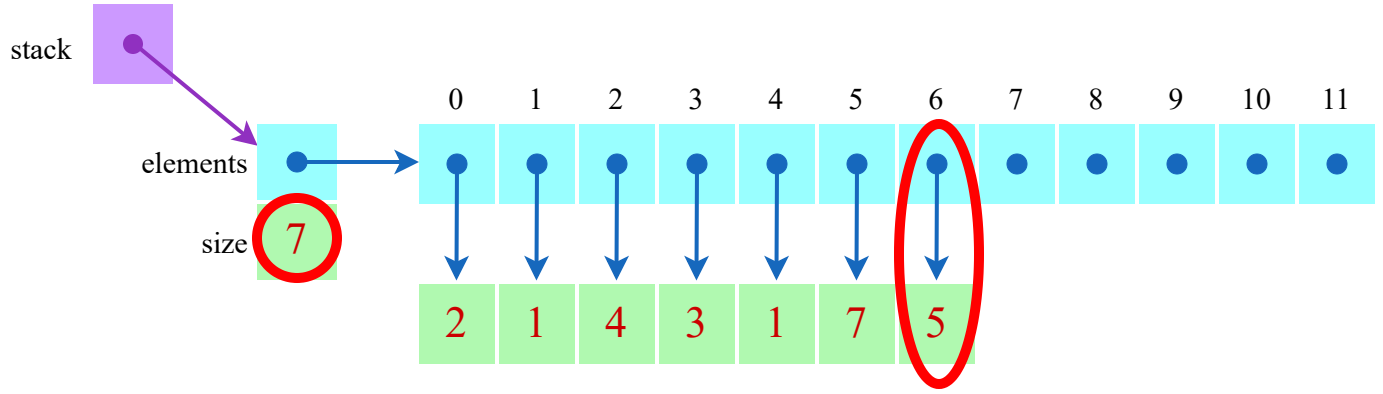
```
Stack<Integer> stack1 = ArrayStack.empty(); // Crea una pila vacía con capacidad inicial predeterminada
Stack<Integer> stack2 = ArrayStack.of(1, 2, 3); // Crea una pila que contiene los elementos 1, 2 y 3
Stack<Integer> stack3 = ArrayStack.copyOf(stack2); // Crea una copia de stack2 y coloca el elemento 4 sobre ella
pila3.push(4);
// Crea una pila a partir de una lista de elementos y calcula su suma
Pila<Entero> pila4 = ArrayStack.from(LinkedList.of(5, 6, 7));
int suma = 0;
while (!stack4.isEmpty()) {
  suma += pila4.top();
  pila4.pop();
}
```

Insertar un elemento en `ArrayStack`

- Para insertar un elemento en la pila, se realizan los siguientes pasos:
  - **Asegurar la capacidad**: Verifique que la matriz tenga suficiente capacidad para alojar el nuevo elemento. Si no, duplica el tamaño de la matriz.
  - **Almacenar elemento**: coloca el nuevo elemento en el índice especificado por el `tamaño` actual.
  - **Incrementar tamaño**: aumenta el `tamaño` en uno para reflejar el nuevo recuento de elementos en la pila.
- Partiendo de esta configuración vamos a hacer `push` 5 en la pila:

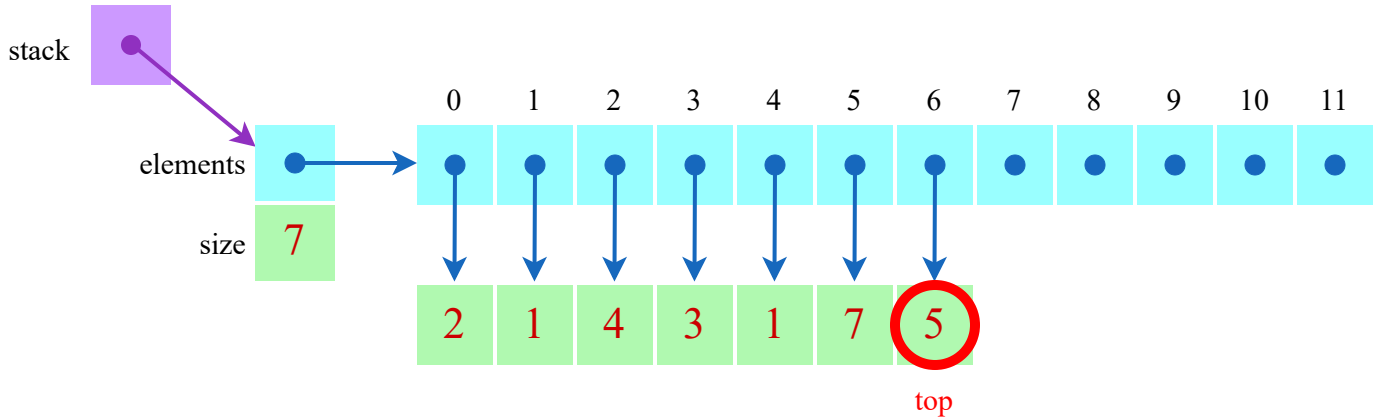


- Después de insertar 5:



Accediendo al elemento superior en `ArrayStack`

- Si la pila está vacía, se debe lanzar una excepción **no comprobada** `EmptyStackException`.
- Si la pila no está vacía:
  - El elemento superior es el que se ha introducido más recientemente en el pila.
  - Este elemento se almacena en la posición `size - 1` y debe ser devuelto.
- Aquí hay una representación visual del acceso al elemento superior de la pila:

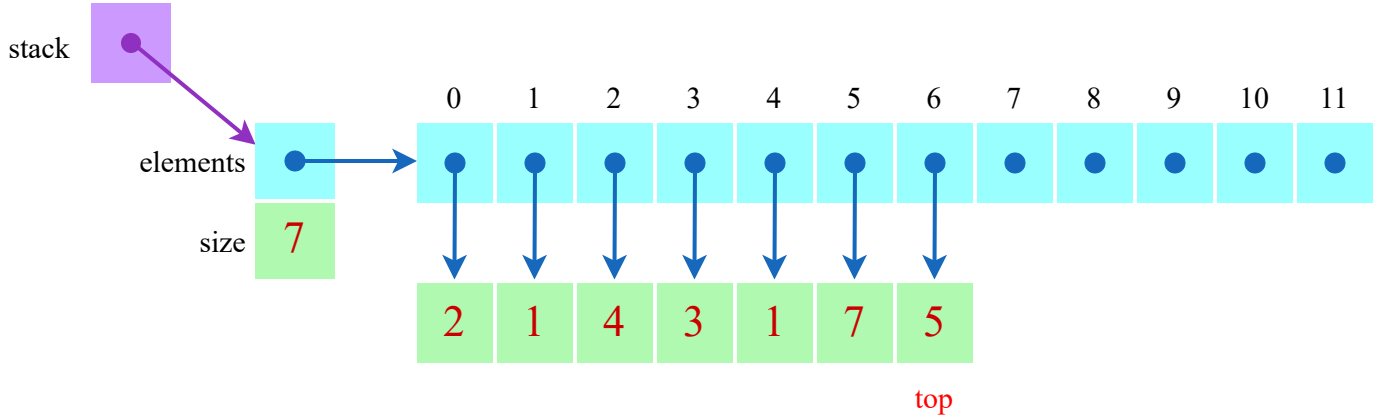


Eliminando el elemento superior en `ArrayStack`

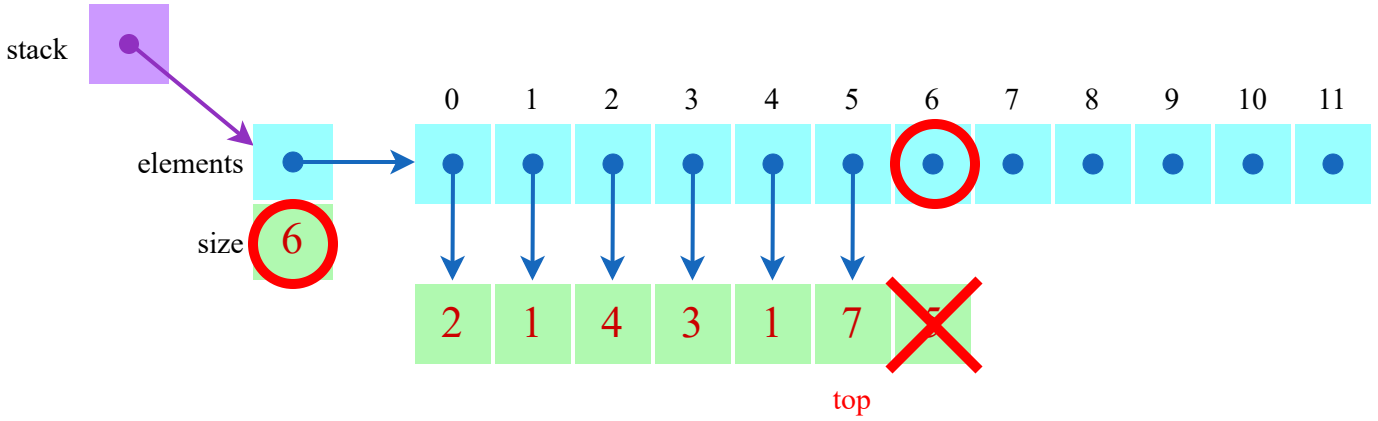
Para implementar correctamente la operación `pop`, podemos seguir estos pasos:

- **Verificar si la pila está vacía**: si la pila está vacía, lanza una `EmptyStackException`.
- **Quitar elemento superior**.
- **Disminuir** `tamaño`.
- **Borrar referencia**: El índice `size` se convierte en `null` permitiendo que el *recolector de basura* recupere la memoria utilizada.

Partiendo de esta configuración, vamos a hacer `pop` en el elemento superior de la pila:



- Después de extraer:



Arrays. Complejidad computacional en Java

- Para evaluar la complejidad computacional de las estructuras de datos que utilizan arrays, como la clase `ArrayStack`, debemos considerar el coste de las operaciones de en un array:
    - **Acceso a un elemento por índice**: O(1)
    - **Establecer un elemento por índice**: O(1)
    - **Asignación de una matriz**: O(n)
    - **Copiar una matriz**: O(n)

Complejidad computacional de las operaciones de `ArrayStack`

Operation	Cost
`empty`	O(1) †
`push`	O(1), O(n) §
`pop`	O(1)
`top`	O(1)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n) *

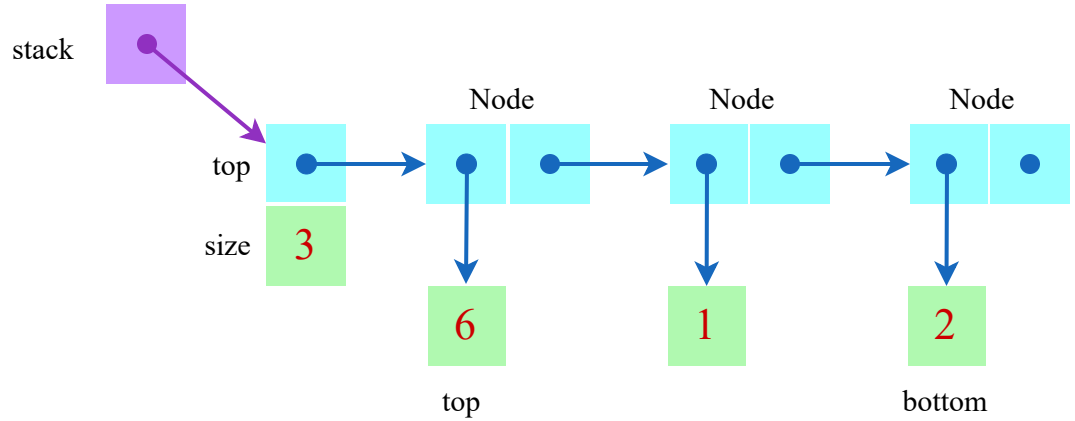
† In `empty` the size of the created array is a constant.

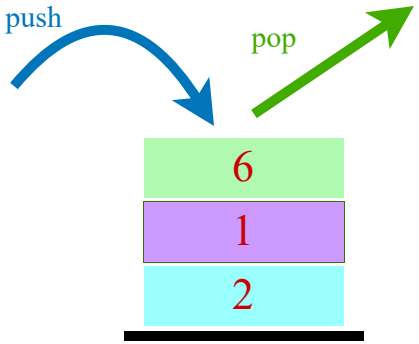
§ `push` will need to copy n elements when the array has to be enlarged.

* `clear` will need to set n references to `null`.

La clase `LinkedStack`

- `LinkedStack<T>` implementa la interfaz `Stack<T>` utilizando una estructura enlazada de nodos. Los nuevos elementos se insertan al **principio** de la estructura vinculada.
- La clase tiene en `top` la parte superior de la pila.
- Cada nodo contiene un elemento y una referencia (`next`)
- El último nodo corresponde a la *parte inferior* de la pila y su `siguiente` vale `null`.
- La variable `size` contiene la cantidad de elementos en la pila.





Especificación de un `LinkedStack` (1/2)

```
public class LinkedStack<T> extends AbstractStack<T> implements Stack<T>{

  private static final class Node<E> {
    E element;
    Node<E> next;

    Node(E element, Node<E> next) {
      this.element = element;
      this.next = next;
    }
  }
  //Atributos
  private Node<T> top;
  private int size;

  public LinkedStack() {
    top = null;
    size = 0;
  }

  public static <T> LinkedStack<T> empty() {
    return new LinkedStack<>();
  }
  ...
}
```

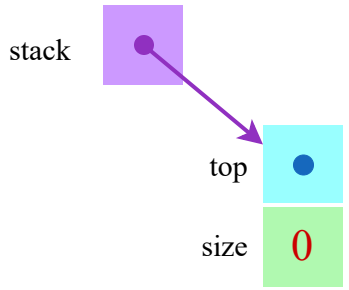
Static Nested Classes

Especificación de un `LinkedStack` (2/2)

```
  ...
  //Factory methods para crear objetos
  @SafeVarargs
  public static <T> LinkedStack<T> of(T... elements);
  public static <T> LinkedStack<T> from(Iterable<T> iterable);
  public static <T> LinkedStack<T> copyOf(LinkedStack<T> that);
  public static <T> LinkedStack<T> copyOf(Stack<T> that);

  protected Iterable<T> elements(); //Abstract
  public int size(); //Abstract && Interface
  public void push(T element); //Interface
  public T top(); //Interface
  public void pop(); //Interface
  public boolean isEmpty(); //Interface
  public void clear(); //Interface


}
```

Inicialización de `LinkedStack`:

- Cuando la pila está vacía, `top` es `null` y `size` es 0.



Métodos de fábrica para `LinkedStack` (1/2)

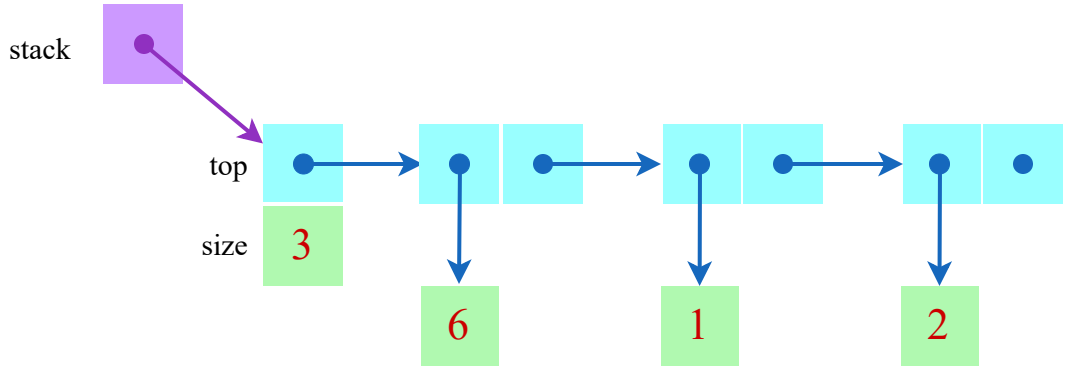
- `empty()`: construye una *pila vacía*.
- `of(T... elementos)`: construye una pila *previamente rellena* con los elementos proporcionados, lo que permite una configuración de pila rápida y sencilla.
- `copyOf(Stack<T> stack)`: construye una nueva pila que es una *duplicado* de la `pila` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva pila que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

Métodos de fábrica para `LinkedStack` (1/2)

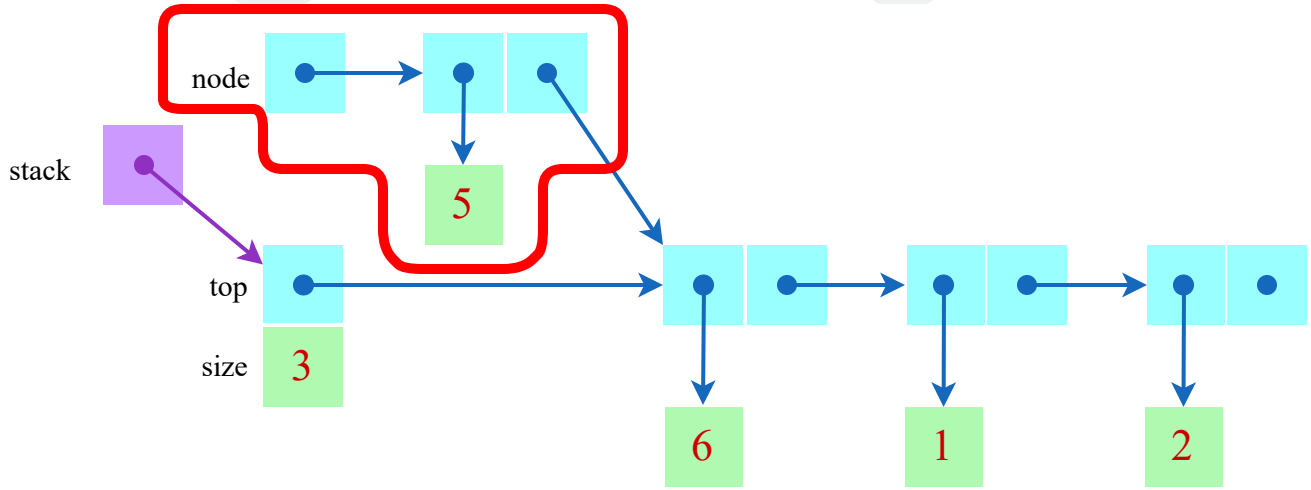
```
Stack<Integer> stack1 = LinkedStack.empty();

Stack<Integer> stack2 = LinkedStack.of(1, 2, 3);

Stack<Integer> stack3 = LinkedStack.copyOf(stack2);
stack3.push(4);

Stack<Integer> stack4 = LinkedStack.from(LinkedList.of(5, 6, 7));
int sum = 0;
while (!stack4.isEmpty()) {
  sum += stack4.top();
  stack4.pop();
}
```

Insertar un elemento en la sección `LinkedStack`

- Partiendo de esta configuración, vamos a `push` el elemento 5:

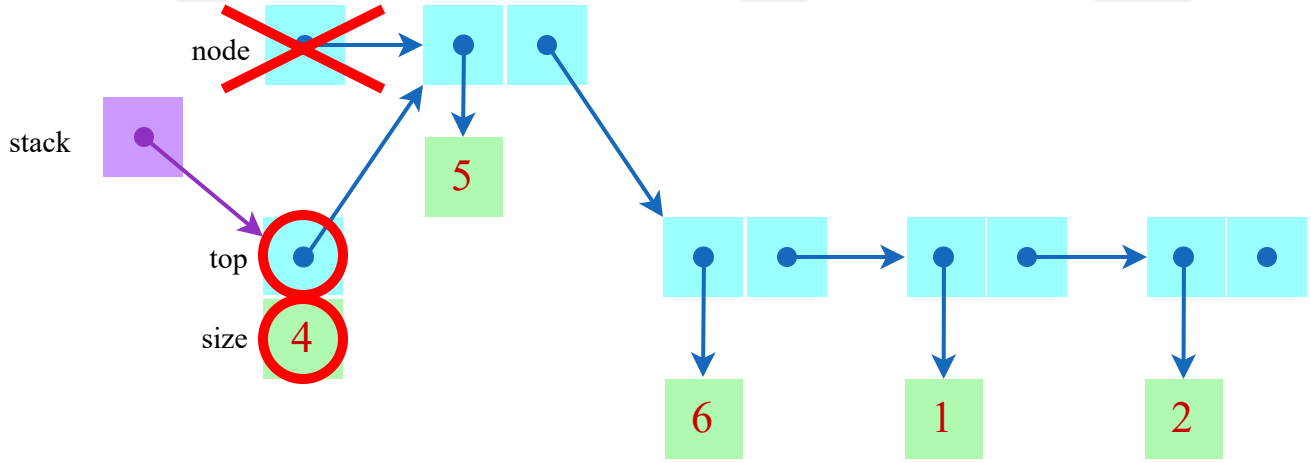


- Se crea un nuevo `nodo` con el elemento (5) y una referencia al `top` actual:



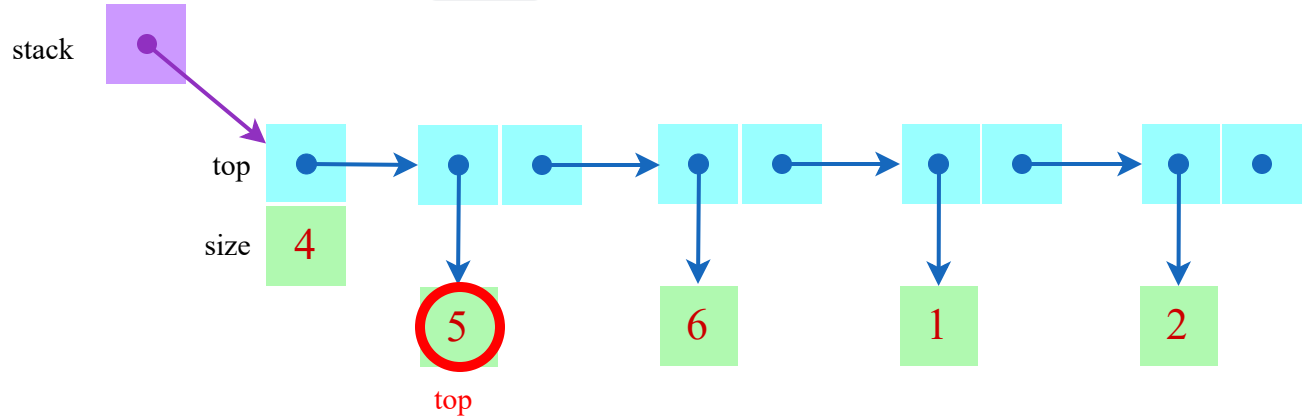
Insertar un elemento en la sección `LinkedStack`

- La referencia `top` se actualiza para apuntar al nuevo `nodo` y se incrementa el `tamaño`:



Acceso al elemento superior en `LinkedStack`:

- El elemento superior es el último elemento insertado en la pila.
- Si la pila está vacía, se debe lanzar una `EmptyStackException`.
- Si la pila no está vacía, el elemento superior es el que se almacena en el nodo referenciado por `top`.

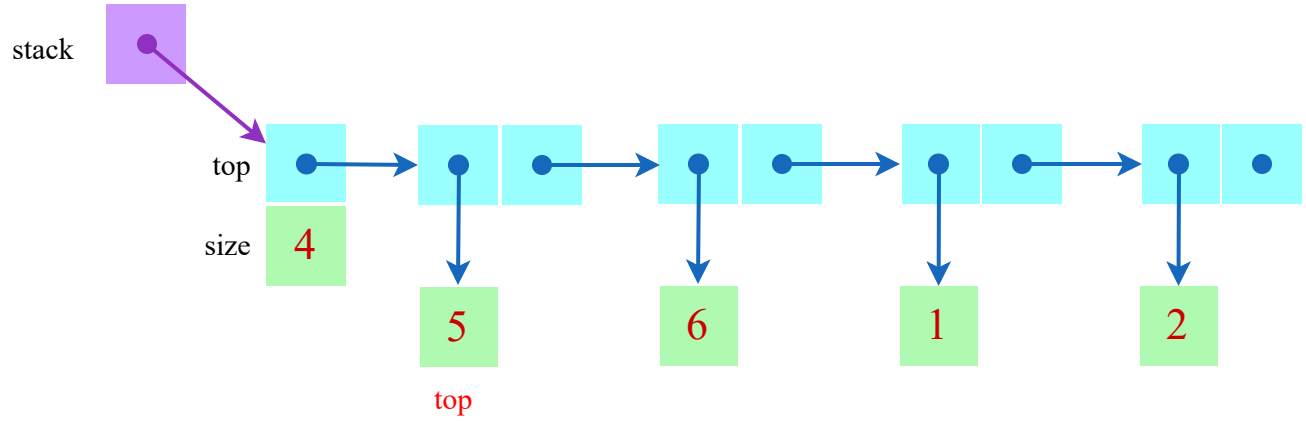


Cómo eliminar el elemento superior en `LinkedStack`

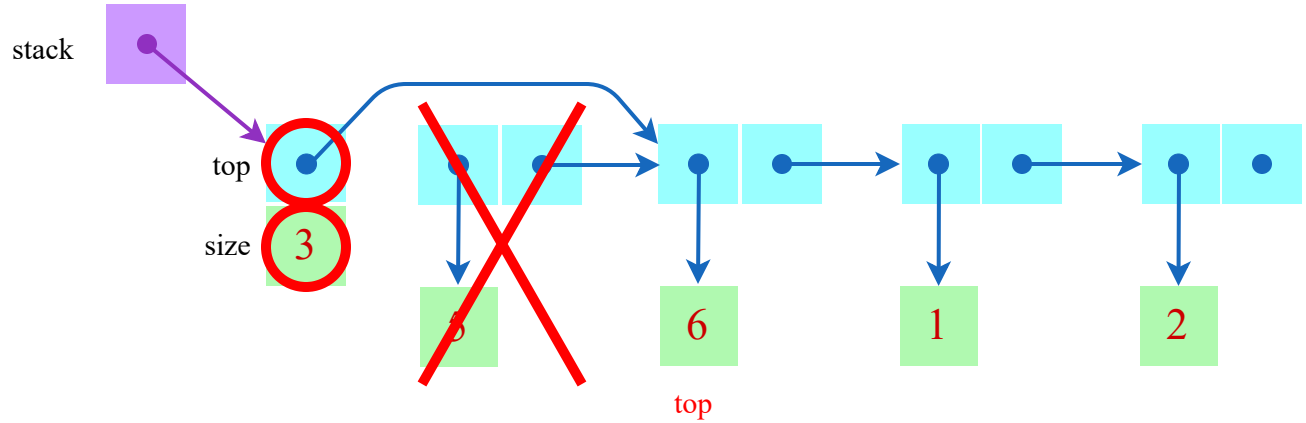
- `pop` elimina el elemento `top` de la pila.
- Si la pila está vacía, se debe lanzar una `EmptyStackException`.
- Si la pila no está vacía:
  - El elemento superior está en el nodo referenciado por `top`.
  - La referencia `top` debe actualizarse para apuntar al siguiente nodo.
  - El *recolector de basura* recuperará la memoria utilizada por el nodo eliminado
  - `size` debe reducirse para reflejar el nuevo recuento de elementos en la pila.

Cómo eliminar el elemento superior en `LinkedStack`

- Partiendo de esta configuración, vamos a hacer "pop" del elemento superior de la pila:



- Después de hacer estallar el elemento superior:



Complejidad computacional de las operaciones de «LinkedStack»

Operation	Cost
`empty`	O(1)
`push`	O(1)
`pop`	O(1)
`top`	O(1)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)

Comparación experimental entre `ArrayStack` y `LinkedStack`

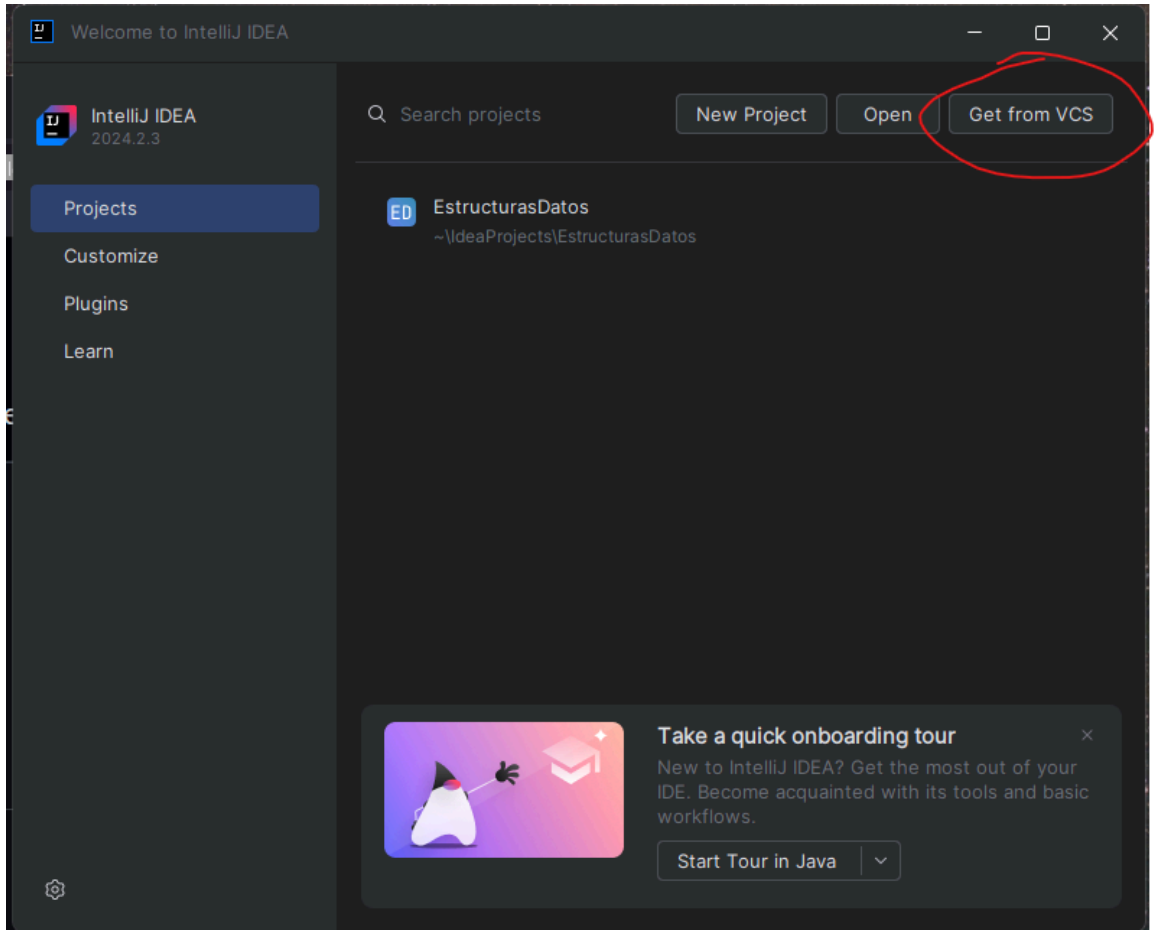
- Medimos el tiempo de ejecución al realizar 10 millones de operaciones aleatorias (`push` o `pop`) en una pila inicialmente vacía.
- Usando una CPU Intel i7 860 y JDK 22:
  - `ArrayStack` fue aproximadamente 1,50 veces más rápido que `LinkedStack`.

Pilas. Aplicaciones

- **Llamadas a funciones**: se utilizan para almacenar información sobre llamadas a funciones (parámetros, resultados, direcciones de retorno, etc.).
- **Evaluación de expresiones**: se utiliza para evaluar expresiones aritméticas (algoritmo de dos pilas de Dijkstra)
- **Búsqueda en profundidad**: se utiliza para almacenar los nodos que se visitarán en un algoritmo de búsqueda en profundidad.
- **Mecanismo de deshacer**: Se utiliza para almacenar el historial de operaciones para permitir deshacerlas.
- **Verificación de sintaxis**: se utiliza para comprobar la corrección de paréntesis, corchetes y llaves en un programa.
- **Expresar recursión**: se utiliza para simular recursión cuando el lenguaje de programación no la admite.

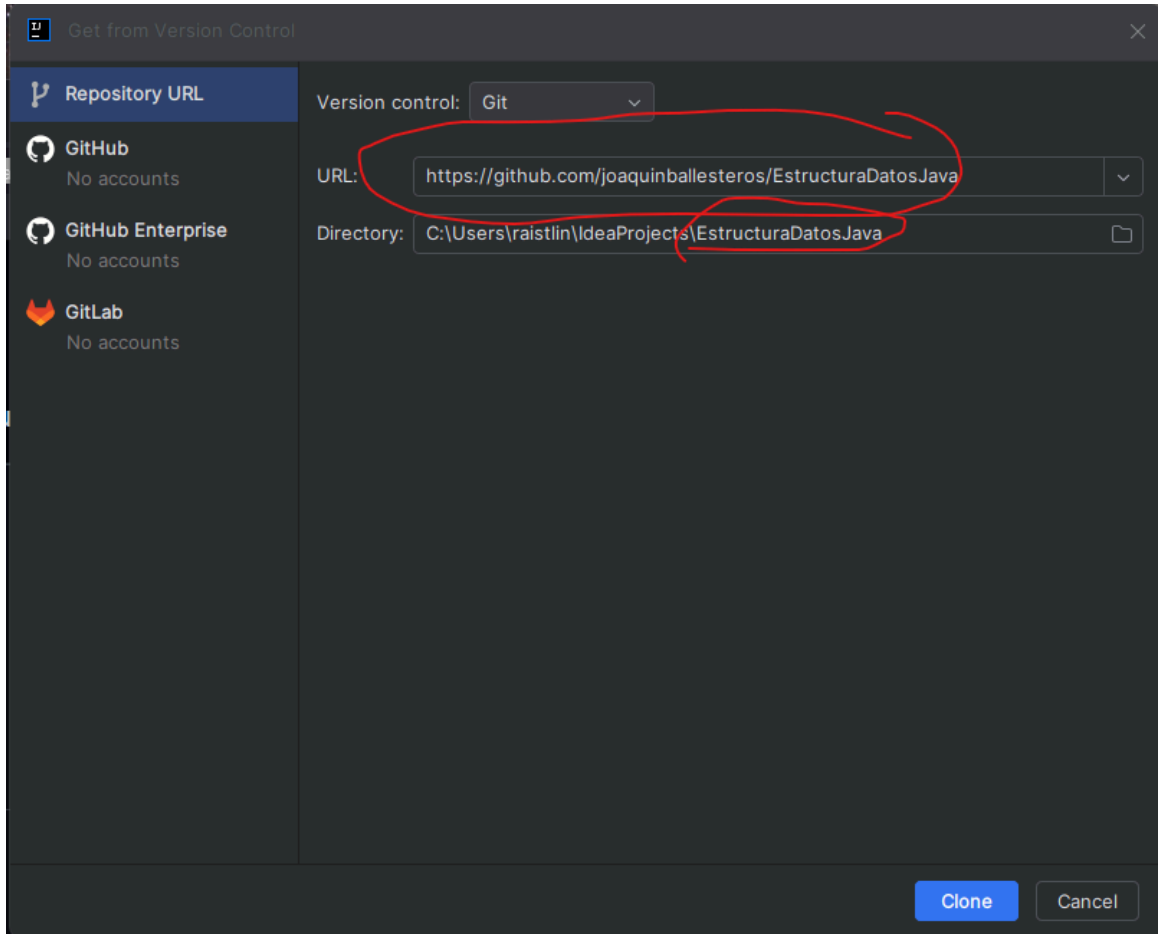
Arrancando (1/3)

1. Selecciona que vas a crear un proyecto desde un repositorio remoto.



Arrancando (2/3)

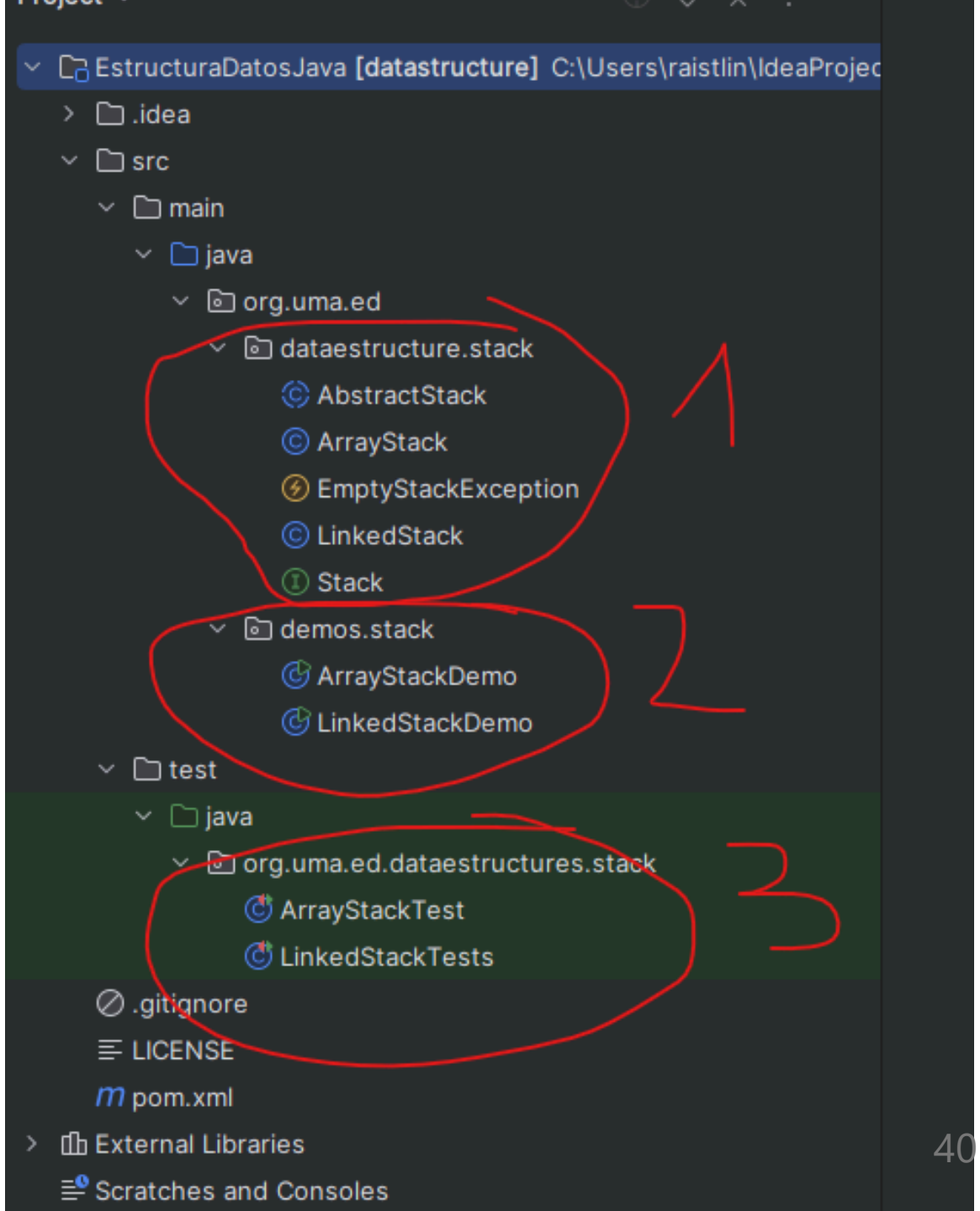
2. Pega el siguiente https://github.com/joaquinballesteros/EstructuraDatosJava
3. Elige el nombre que quieres (*EstructuraDatosJava*)



Arrancando (3/3)

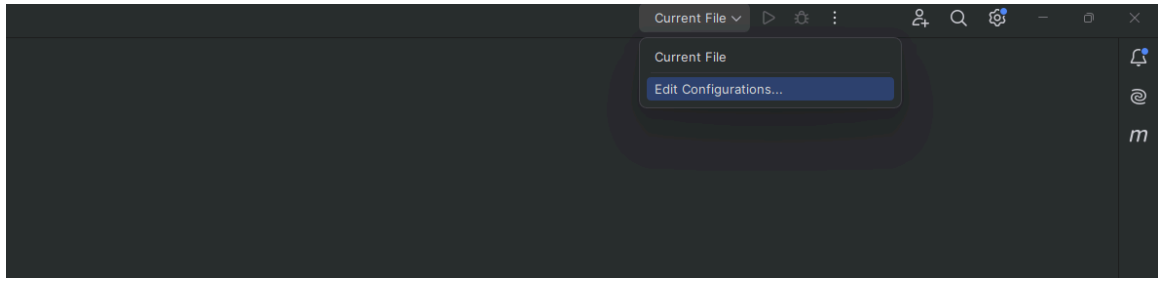
Tienes tres zonas:

1. Liberías. Zona de desarrollo de nuevas estructuras de datos.
2. Principales para probar rápido (incluyen main).
3. Tests con JUnit.



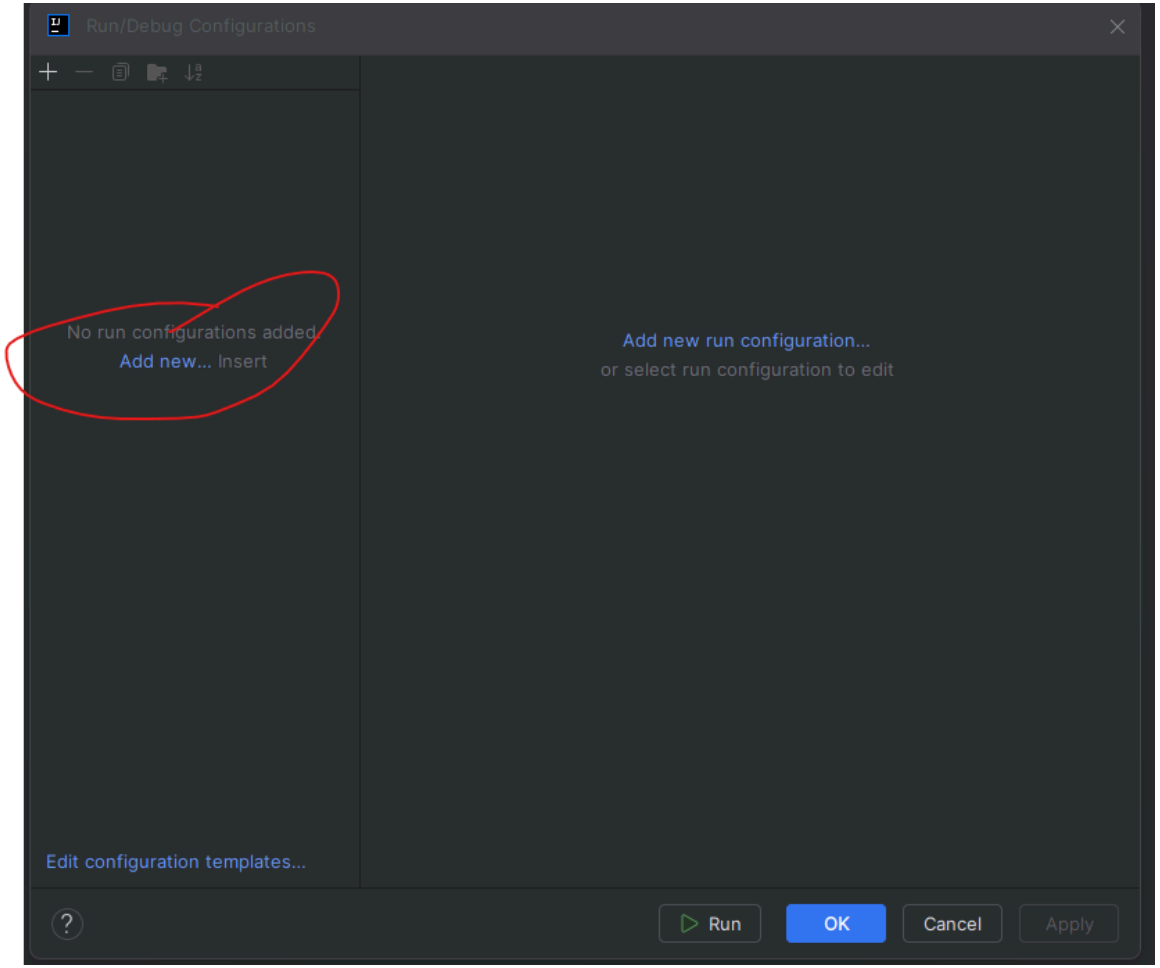
Configurando testing (1/4)

Vamos a la parte superior y seleccionamos editar configuraciones.



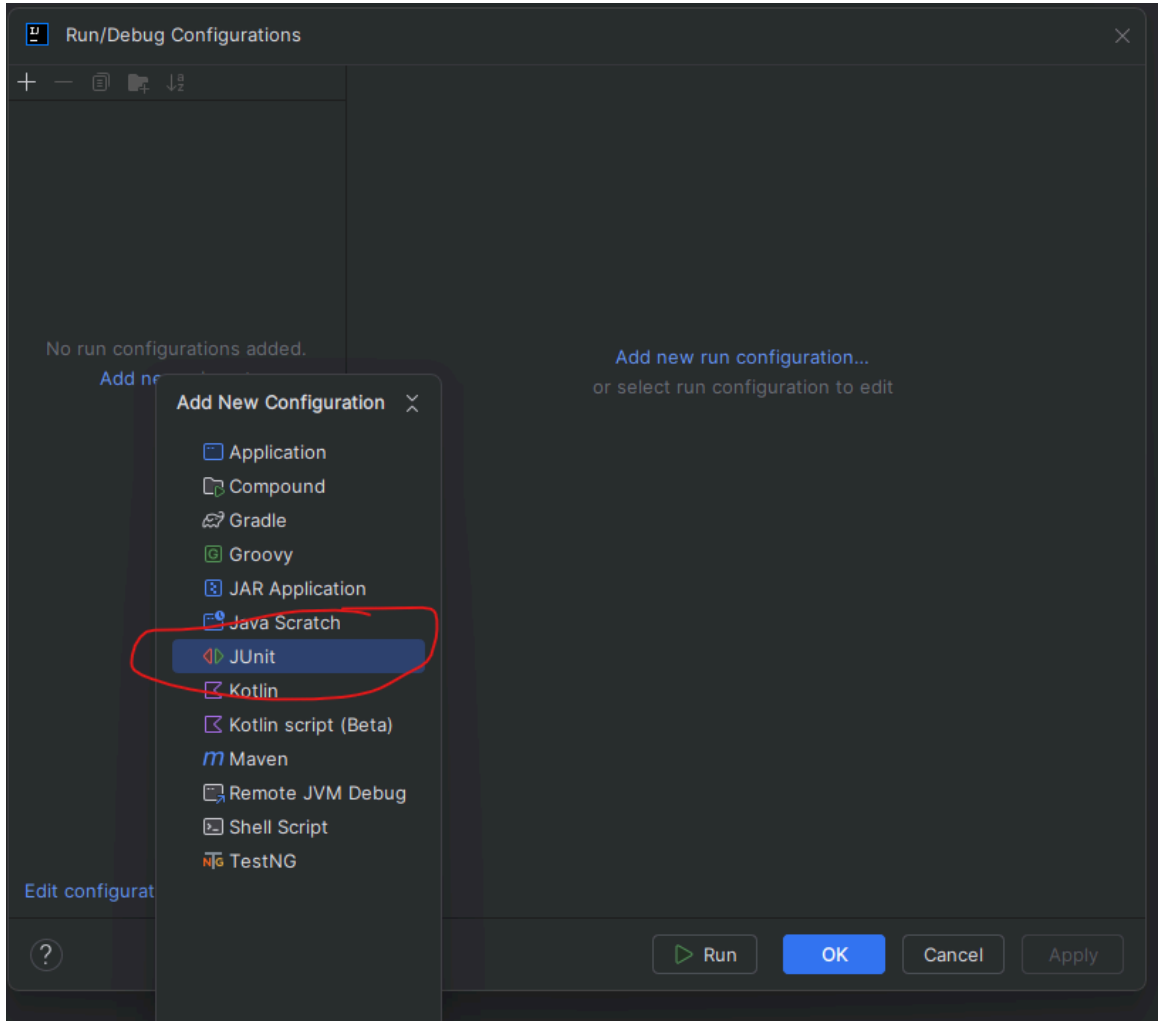
Configurando testing (2/4)

Añadimos una nueva configuración.



Configurando testing (3/4)

Seleccionamos JUnit



Configurando testing (4/4)

Añadimos el nombre que queremos asignar a esta configuración y seleccionamos el fichero de pruebas que vamos a lanzar.