Estructuras de datos básicas: Bolsas, pilas, colas y listas

Jorge Mario Londoño Peláez & Varias AI

January 31, 2025

1 Estructuras de datos básicas

En esta sección, exploraremos las estructuras de datos básicas, diferenciándolas de estructuras más simples como arreglos y matrices. Estas estructuras, aunque fundamentales, ofrecen una mayor flexibilidad y adaptabilidad en la gestión de datos. Las estructuras de datos que veremos deben ser dinámicas, es decir, que puedan crecer o decrecer según la necesidad, deben garantizar la consistencia de tipos (usando tipos genéricos) y deben implementar funcionalidades de iteración (interface *Iterable*).

1.1 Bolsa (Bag)

Una **Bolsa** es una colección no ordenada de elementos, donde se permite la duplicación. Es útil cuando se necesita almacenar elementos sin importar el orden y sin necesidad de acceder a ellos por una posición específica.

API de la Bolsa:

- `add(element)`: Agrega un elemento a la bolsa.
- `remove(element)`: Elimina una instancia del elemento de la bolsa (si existe).
- `isEmpty()`: Verifica si la bolsa está vacía.
- `size()`: Retorna el número de elementos en la bolsa.
- `contains(element)`: Verifica si la bolsa contiene el elemento.
- `iterator()`: Retorna un iterador para recorrer los elementos de la bolsa.

Ejemplo de aplicación: Una bolsa puede ser usada para almacenar los productos en un carrito de compras, donde el orden de los productos no es relevante y puede haber duplicados.

1.2 Pila (Stack)

Una **Pila** es una colección ordenada de elementos que sigue el principio Last In, First Out (LIFO), es decir, el último elemento en entrar es el primero en salir. Es útil para gestionar el orden de ejecución de funciones, el historial de navegación, etc.

API de la Pila:

- `push(element)`: Agrega un elemento a la cima de la pila.
- `pop()`: Elimina y retorna el elemento de la cima de la pila.

- `peek()`: Retorna el elemento de la cima de la pila sin eliminarlo.
- `isEmpty()`: Verifica si la pila está vacía.
- `size()`: Retorna el número de elementos en la pila.

Ejemplo de aplicación: Una pila puede ser usada para implementar la funcionalidad de "deshacer" en un editor de texto, donde la última acción realizada es la primera en ser deshecha.

1.3 Cola (Queue)

Una **Cola** es una colección ordenada de elementos que sigue el principio FIFO (First In, First Out), es decir, el primer elemento en entrar es el primero en salir. Es útil para gestionar tareas en espera, simular filas de atención, etc.

API de la Cola:

- `enqueue(element)`: Agrega un elemento al final de la cola.
- `dequeue()`: Elimina y retorna el elemento del frente de la cola.
- `peek()`: Retorna el elemento del frente de la cola sin eliminarlo.
- `isEmpty()`: Verifica si la cola está vacía.
- `size()`: Retorna el número de elementos en la cola.

Ejemplo de aplicación: Una cola puede ser usada para gestionar la cola de impresión de documentos, donde el primer documento enviado a imprimir es el primero en ser impreso.

1.4 Ejemplo de implementación de la Pila basada en un arreglo

A continuación, se presenta una implementación de la Pila basada en un arreglo en Java. Esta implementación utiliza un arreglo de `String` para almacenar los elementos de la pila.

```java
public class StackArray {
    private String[] stack;
    private int top;
    private int capacity;

    public StackArray(int capacity) {
        this.capacity = capacity;
        this.stack = new String[capacity];
        this.top = -1;
    }

    public void push(String element) {
        if (top == capacity - 1) {
            throw new IllegalStateException("Stack is full");
        }
        stack[++top] = element;
    }

    public String pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return stack[top--];
```

```
    }

    public String peek() {
         if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return stack[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public int size() {
        return top + 1;
    }
}
```

Listing 1: Implementación de la Pila basada en un arreglo en Java

Ejemplo de pruebas unitarias para los métodos de la Pila:

```
public class StackArrayTest {
    public static void main(String[] args) {
        StackArray stack = new StackArray(5);
        stack.push("A");
        stack.push("B");
        stack.push("C");

        assert stack.size() == 3;
        assert stack.peek().equals("C");
        assert stack.pop().equals("C");
        assert stack.size() == 2;
        assert !stack.isEmpty();

        assert stack.pop().equals("B");
        assert stack.pop().equals("A");
        assert stack.isEmpty();
    }
}
```

Listing 2: Ejemplo de pruebas unitarias para la Pila

Discusión: Para implementar la Bolsa o la Cola, sería necesario cambiar la forma en que se insertan y eliminan los elementos. En la Bolsa, la inserción se haría al final del arreglo y la eliminación requeriría buscar el elemento. En la Cola, la inserción se haría al final y la eliminación al inicio del arreglo.

Limitaciones de esta implementación: Esta implementación tiene las siguientes limitaciones:

- **Tamaño fijo:** El tamaño de la pila se define al momento de la creación y no puede cambiar.
- **Solo soporta un tipo:** La pila solo puede almacenar elementos de tipo `String`.

Para solucionar estas limitaciones, se pueden usar estructuras de datos más avanzadas como listas enlazadas o usar tipos genéricos.

1.5 Tipos genéricos

Los **tipos genéricos** permiten definir clases e interfaces que trabajan con diferentes tipos de datos sin perder la verificación de tipos en tiempo de compilación. Esto significa que se puede escribir código que funcione con cualquier tipo de dato, sin necesidad de escribir código específico para cada tipo.

Características de los tipos genéricos:

- **Reutilización de código:** Se puede escribir una sola clase o interfaz que funcione con diferentes tipos de datos.
- **Seguridad de tipos:** El compilador verifica que los tipos de datos utilizados sean correctos, evitando errores en tiempo de ejecución.
- **Eliminación de casting:** No es necesario realizar conversiones de tipo explícitas (casting), ya que el compilador conoce el tipo de dato con el que se está trabajando.

Ejemplo de implementación de la Pila basada en arreglo con tipos genéricos en Java: A continuación, se presenta una implementación de la Pila basada en un arreglo en Java, utilizando tipos genéricos. Esta implementación puede almacenar elementos de cualquier tipo.

```java
public class StackGeneric<T> {
    private T[] stack;
    private int top;
    private int capacity;

    public StackGeneric(int capacity) {
        this.capacity = capacity;
        this.stack = (T[]) new Object[capacity];
        this.top = -1;
    }

    public void push(T element) {
        if (top == capacity - 1) {
            throw new IllegalStateException("Stack is full");
        }
        stack[++top] = element;
    }

    public T pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return stack[top--];
    }

    public T peek() {
         if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return stack[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

```

```
    public int size() {
        return top + 1;
    }
}
```

Listing 3: Implementación de la Pila basada en un arreglo con tipos genéricos en Java

Ejemplo de pruebas unitarias para la Pila genérica:

```
public class StackGenericTest {
    public static void main(String[] args) {
        StackGeneric<Integer> stack = new StackGeneric<>(5);
        stack.push(1);
        stack.push(2);
        stack.push(3);

        assert stack.size() == 3;
        assert stack.peek().equals(3);
        assert stack.pop().equals(3);
        assert stack.size() == 2;
        assert !stack.isEmpty();

        assert stack.pop().equals(2);
        assert stack.pop().equals(1);
        assert stack.isEmpty();

        StackGeneric<String> stringStack = new StackGeneric<>(5);
        stringStack.push("A");
        stringStack.push("B");
        stringStack.push("C");

        assert stringStack.size() == 3;
        assert stringStack.peek().equals("C");
        assert stringStack.pop().equals("C");
        assert stringStack.size() == 2;
        assert !stringStack.isEmpty();

        assert stringStack.pop().equals("B");
        assert stringStack.pop().equals("A");
        assert stringStack.isEmpty();
    }
}
```

Listing 4: Ejemplo de pruebas unitarias para la Pila genérica

1.6 Tipos wrapper

En Java, existen dos categorías principales de tipos de datos: **tipos primitivos** y **tipos wrapper**. Los tipos primitivos son los tipos de datos básicos, como `int`, `double`, `boolean`, etc. Los tipos wrapper, por otro lado, son clases que ”envuelven” a los tipos primitivos, como `Integer`, `Double`, `Boolean`, etc.

Diferencias entre tipos primitivos y tipos wrapper:

- **Almacenamiento:** Los tipos primitivos almacenan directamente el valor, mientras que los tipos wrapper almacenan una referencia a un objeto que contiene el valor.
- **Valores nulos:** Los tipos primitivos no pueden ser nulos, mientras que los tipos wrapper sí pueden serlo.

- **Métodos:** Los tipos wrapper ofrecen métodos útiles para trabajar con los valores que contienen, como convertir a `String`, comparar, etc.

Uso de tipos wrapper en genéricos: En Java, los tipos genéricos solo pueden trabajar con tipos de referencia, es decir, con objetos. Por lo tanto, no se pueden usar tipos primitivos directamente en estructuras genéricas como `StackGeneric<int>`. En su lugar, se deben usar los tipos wrapper correspondientes, como `StackGeneric<Integer>`.

Autoboxing y unboxing: Java ofrece la funcionalidad de **autoboxing** y **unboxing** para facilitar el trabajo con tipos primitivos y wrapper.

- **Autoboxing:** Es la conversión automática de un tipo primitivo a su tipo wrapper correspondiente. Por ejemplo, `Integer i = 5;` es un ejemplo de autoboxing, donde el valor `5` de tipo `int` es convertido automáticamente a un objeto de tipo `Integer`.

- **Unboxing:** Es la conversión automática de un tipo wrapper a su tipo primitivo correspondiente. Por ejemplo, `int j = i;` es un ejemplo de unboxing, donde el objeto `i` de tipo `Integer` es convertido automáticamente a un valor de tipo `int`.

Gracias al autoboxing y unboxing, se puede trabajar con tipos primitivos y wrapper de forma transparente al implementar estructuras genéricas. Por ejemplo, se puede usar `stack.push(5);` en una pila de tipo `StackGeneric<Integer>`, y Java se encargará de convertir el valor `5` de tipo `int` a un objeto de tipo `Integer` antes de agregarlo a la pila.