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

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

August 22, 2025

Descripción de la unidad

En este capítulo se estudian algunas de las estructuras básicas, que a su vez son ampliamente utilizadas en la construcción de estructuras más complejas y en la implementación de muchos algoritmos.

Contents

1 Estructuras de datos básicas **2**
- 1.1 Bolsa (Bag) . . . 2
- 1.2 Pila (Stack) . . . 2
- 1.3 Cola (Queue) . . . 3
- 1.4 Ejemplo de implementación de la Pila basada en un arreglo . . . 3
- 1.5 Tipos genéricos . . . 4
- 1.6 Pila con arreglo de tamaño dinámico (Resizing Array) . . . 6
- 1.7 Tipos wrapper . . . 7
- 1.8 Iteradores . . . 8
  - 1.8.1 Iteradores en Java . . . 8
  - 1.8.2 Iteradores en Python . . . 10
  - 1.8.3 Iteradores en C# . . . 11
- 1.9 Implementación de la Cola con Arreglo Circular . . . 13
- 1.10 Implementación de la Bolsa con Arreglo . . . 14

2 Listas Simplemente Enlazadas **16**
- 2.1 Definición recursiva de lista . . . 16
- 2.2 Implementación de la lista simplemente enlazada . . . 16
  - 2.2.1 Definición de la clase Nodo . . . 16
  - 2.2.2 Agregar elementos a la lista (proceso paso a paso manual) . . . 17
  - 2.2.3 Implementación de la lista enlazada como ADT . . . 18
- 2.3 Otras operaciones con listas simplemente enlazadas . . . 20
  - 2.3.1 Eliminar el primer elemento de la lista . . . 20
  - 2.3.2 Agregar un elemento al final de la lista . . . 20
  - 2.3.3 Eliminar el último elemento de la lista . . . 20
  - 2.3.4 Otras operaciones con listas simples . . . 22
- 2.4 Iteradores sobre listas simplemente enlazadas . . . 22
  - 2.4.1 Proceso de recorrido de una lista . . . 22
  - 2.4.2 Operaciones que dependen de un recorrido de la lista . . . 22

3 Listas Doblemente Enlazadas **25**
- 3.1 ADT e Implementación de la lista doblemente enlazada . . . 25
- 3.2 La cola de doble terminación (DEQUE) . . . 28
- 3.3 Colas circulares . . . 28

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.

```
1  public class StackArray {
2      private String[] stack;
3      private int top;
4      private int capacity;
5
6      public StackArray(int capacity) {
7          this.capacity = capacity;
8          this.stack = new String[capacity];
9          this.top = -1;
10     }
11
12     public void push(String element) {
13         if (top == capacity - 1) {
14             throw new IllegalStateException("Stack is full");
15         }
16         stack[++top] = element;
17     }
18
19     public String pop() {
20         if (isEmpty()) {
21             throw new IllegalStateException("Stack is empty");
22         }
23         return stack[top--];
24     }
25
26     public String peek() {
27          if (isEmpty()) {
28             throw new IllegalStateException("Stack is empty");
29         }
30         return stack[top];
31     }
32
33     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

Nota: Para que las aserciones (`assert`) funcionen, es necesario habilitarlas al ejecutar el programa desde la línea de comandos, usando el flag `-ea` (enable assertions). Ejemplo: `java -ea StackArrayTest`. Sin este flag, las líneas con `assert` serán ignoradas.

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:

```java
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 Pila con arreglo de tamaño dinámico (Resizing Array)

La implementación anterior tiene una limitación importante: un tamaño fijo. Para solucionar esto, podemos usar una estrategia de redimensionamiento dinámico. La idea es empezar con un arreglo de tamaño pequeño y, cuando se llene, crear un nuevo arreglo más grande (normalmente el doble de tamaño) y copiar los elementos. De manera similar, para no desperdiciar memoria, si al eliminar elementos el arreglo queda muy vacío (por ejemplo, a un cuarto de su capacidad), lo redimensionamos a un tamaño más pequeño (la mitad).

Esta estrategia asegura que el costo de cada operación (push y pop), en promedio, sea constante (costo amortizado O(1)).

```
public class StackResizingArray<T> {
    private T[] stack;
    private int n; // Numero de elementos en la pila

    public StackResizingArray() {
        stack = (T[]) new Object[2]; // Empezar con capacidad 2
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
```

```
    }

    public int size() {
        return n;
    }

    private void resize(int capacity) {
        assert capacity >= n;
        T[] copy = (T[]) new Object[capacity];
        for (int i = 0; i < n; i++) {
            copy[i] = stack[i];
        }
        stack = copy;
    }

    public void push(T element) {
        if (n == stack.length) {
            resize(2 * stack.length); // Doblar el tamano si esta lleno
        }
        stack[n++] = element;
    }

    public T pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        T element = stack[--n];
        stack[n] = null; // Evitar "loitering" (mantener referencias a objetos no
    usados)
        if (n > 0 && n == stack.length / 4) {
            resize(stack.length / 2); // Reducir a la mitad si esta a 1/4 de
    capacidad
        }
        return element;
    }

    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return stack[n - 1];
    }
}
```

Listing 5: Implementación de la Pila con arreglo dinámico en Java

1.7 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.

1.8 Iteradores

Un **iterador** es un objeto que permite recorrer los elementos de una colección, uno a la vez, sin necesidad de conocer la estructura interna de la colección. Los iteradores proporcionan una forma estándar de acceder a los elementos de una colección, independientemente de cómo estén almacenados. Esto permite que el código que usa la colección sea más genérico y reutilizable.

1.8.1 Iteradores en Java

En Java, los iteradores se implementan utilizando las interfaces `Iterable` e `Iterator`. La interfaz `Iterable` define el método `iterator()`, que retorna un objeto de tipo `Iterator`. La interfaz `Iterator` define los métodos `hasNext()`, que verifica si hay más elementos en la colección, y `next()`, que retorna el siguiente elemento de la colección.

A continuación, se presenta un ejemplo de implementación del iterador para la clase `StackGeneric` en Java:

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4  public class StackGeneric<T> implements Iterable<T> {
5      private T[] stack;
6      private int top;
7      private int capacity;
8
9      public StackGeneric(int capacity) {
10         this.capacity = capacity;
11         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;
    }

    @Override
    public Iterator<T> iterator() {
        return new StackIterator();
    }

    private class StackIterator implements Iterator<T> {
        private int current = top;

        @Override
        public boolean hasNext() {
            return current >= 0;
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return stack[current--];
        }
    }
}
```

Listing 6: Implementación del iterador para StackGeneric en Java

```
public class StackGenericTest {
    public static void main(String[] args) {
```

```
        StackGeneric<Integer> stack = new StackGeneric<>(5);
        stack.push(1);
        stack.push(2);
        stack.push(3);

        for (Integer element : stack) {
            System.out.println(element);
        }
    }
}
```

Listing 7: Ejemplo de uso del iterador para StackGeneric en Java

1.8.2 Iteradores en Python

En Python, los iteradores se implementan utilizando el método `__iter__()` y el protocolo de iteración. El método `__iter__()` debe retornar un objeto iterador, que a su vez debe implementar los métodos `__next__()` y `__iter__()`. El método `__next__()` retorna el siguiente elemento de la colección, y el método `__iter__()` retorna el propio objeto iterador.

A continuación, se presenta una implementación más idiomática en Python, utilizando un generador (`yield`) para la iteración. Esto elimina la necesidad de una clase `StackIterator` explícita, resultando en un código más limpio y conciso.

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)

    def __iter__(self):
        """
        Iterador que recorre la pila desde la cima (LIFO).
        """
        for i in range(len(self.items) - 1, -1, -1):
            yield self.items[i]
```

Listing 8: Implementación del iterador para Stack en Python con yield

```
stack = Stack()
stack.push(1)
```

```
stack.push(2)
stack.push(3)

for item in stack:
    print(item)
```

Listing 9: Ejemplo de uso del iterador para Stack en Python

1.8.3 Iteradores en C#

En C#, los iteradores se implementan utilizando la interfaz `IEnumerable` y `IEnumerator`. La interfaz `IEnumerable` define el método `GetEnumerator()`, que retorna un objeto de tipo `IEnumerator`. La interfaz `IEnumerator` define los métodos `MoveNext()`, que avanza al siguiente elemento de la colección, `Current`, que retorna el elemento actual de la colección, y `Reset()`, que reinicia el iterador al inicio de la colección.

A continuación, se presenta un ejemplo de implementación del iterador para la clase `StackGeneric` en C#:

```
using System;
using System.Collections;
using System.Collections.Generic;

public class StackGeneric<T> : IEnumerable<T>
{
    private T[] stack;
    private int top;
    private int capacity;

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

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

    public T Pop()
    {
        if (IsEmpty())
        {
            throw new InvalidOperationException("Stack is empty");
        }
        return stack[top--];
    }

    public T Peek()
    {
        if (IsEmpty())
        {
```

```
            throw new InvalidOperationException("Stack is empty");
        }
        return stack[top];
    }

    public bool IsEmpty()
    {
        return top == -1;
    }

    public int Size()
    {
        return top + 1;
    }

    public IEnumerator<T> GetEnumerator()
    {
        return new StackEnumerator(this);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    private class StackEnumerator : IEnumerator<T>
    {
        private StackGeneric<T> stack;
        private int current;

        public StackEnumerator(StackGeneric<T> stack)
        {
            this.stack = stack;
            // El iterador se posiciona antes del primer elemento.
            this.current = stack.top + 1;
        }

        public bool MoveNext()
        {
            // Avanza al siguiente elemento y retorna true si hay mas elementos.
            current--;
            return current >= 0;
        }

        public void Reset()
        {
            current = stack.top + 1;
        }

        public T Current
        {
            get
            {
                if (current < 0 || current > stack.top)
                {
                    throw new InvalidOperationException();
                }
                return stack.stack[current];
            }
```

```
        }

        object IEnumerator.Current
        {
            get { return Current; }
        }

        public void Dispose()
        {
            // No resources to dispose
        }
    }
}
```

Listing 10: Implementación del iterador para StackGeneric en C#

```
public class StackGenericTest
{
    public static void Main(string[] args)
    {
        StackGeneric<int> stack = new StackGeneric<int>(5);
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);

        foreach (int element in stack)
        {
            Console.WriteLine(element);
        }
    }
}
```

Listing 11: Ejemplo de uso del iterador para StackGeneric en C#

Porque se usan dos interfaces en C#?

- La versión pública y fuertemente tipada (`IEnumerator<T>`) se usa cuando el consumidor trabaja con la clase a través de `IEnumerable<T>`, obteniendo elementos del tipo correcto (T) sin necesidad de hacer casting. Esta versión es la forma moderna de usar enumeradores.
- La versión explícita no genérica (IEnumerator) existe para cumplir con el contrato de la vieja interfaz IEnumerable y garantizar compatibilidad con código que no usa genéricos (por ejemplo, foreach sobre una referencia de tipo IEnumerable en lugar de `IEnumerable<T>`). Si no se incluye, genera un error de compilación.

1.9 Implementación de la Cola con Arreglo Circular

Una implementación ingenua de una cola (Queue) usando un arreglo implicaría que al hacer `dequeue`, todos los elementos restantes deberían moverse una posición hacia adelante, lo cual es una operación costosa de O(n). Para evitar esto, se utiliza una técnica llamada **arreglo circular**.

En un arreglo circular, mantenemos dos punteros: `head` (cabeza) y `tail` (cola). `head` apunta al primer elemento y `tail` apunta a la siguiente posición libre. Cuando un puntero llega al final del arreglo, vuelve al principio (de ahí el nombre "circular"). Esto se logra fácilmente con el operador módulo (

```
public class QueueCircularArray<T> {
    private T[] queue;
```

```
    private int head; // Puntero al primer elemento
    private int tail; // Puntero a la proxima posicion libre
    private int n;    // Numero de elementos en la cola

    public QueueCircularArray(int capacity) {
        queue = (T[]) new Object[capacity];
        head = 0;
        tail = 0;
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void enqueue(T element) {
        if (n == queue.length) {
            throw new IllegalStateException("Queue is full");
        }
        queue[tail] = element;
        tail = (tail + 1) % queue.length; // Avanzar y dar la vuelta si es
    necesario
        n++;
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        T element = queue[head];
        queue[head] = null; // Evitar loitering
        head = (head + 1) % queue.length; // Avanzar y dar la vuelta si es
    necesario
        n--;
        return element;
    }

    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return queue[head];
    }
}
```

Listing 12: Implementación de una Cola con arreglo circular en Java

1.10 Implementación de la Bolsa con Arreglo

La bolsa (Bag) es la más simple de las tres estructuras en términos de requisitos de orden. Como no hay un orden que preservar, la operación `add` simplemente puede agregar el elemento al final del arreglo, usando una estrategia de redimensionamiento dinámico igual que la pila.

La operación `remove(element)` es más compleja, ya que requiere encontrar el elemento en el

arreglo. Una vez encontrado, no necesitamos mantener el orden, por lo que la forma más eficiente de eliminarlo es intercambiarlo con el último elemento del arreglo y luego eliminar el último elemento. Esto evita el costo de O(n) de desplazar todos los elementos subsiguientes.

```java
public class BagArray<T> {
    private T[] bag;
    private int n; // Numero de elementos

    public BagArray() {
        bag = (T[]) new Object[2];
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    private void resize(int capacity) {
        T[] copy = (T[]) new Object[capacity];
        for (int i = 0; i < n; i++) {
            copy[i] = bag[i];
        }
        bag = copy;
    }

    public void add(T element) {
        if (n == bag.length) {
            resize(2 * bag.length);
        }
        bag[n++] = element;
    }

    public boolean remove(T element) {
        for (int i = 0; i < n; i++) {
            if (element.equals(bag[i])) {
                // Encontrado, intercambiar con el ultimo elemento
                bag[i] = bag[n - 1];
                bag[n - 1] = null; // loitering
                n--;
                return true;
            }
        }
        return false; // No encontrado
    }
}
```

Listing 13: Implementación de una Bolsa con arreglo dinámico en Java

2 Listas Simplemente Enlazadas

2.1 Definición recursiva de lista

Una lista enlazada puede definirse recursivamente de la siguiente manera:

- Caso base: Una lista vacía es una referencia nula (`null`).
- Caso recursivo: Una lista es un `Nodo` que contiene un dato y una referencia a una lista, la cual a su vez puede ser el siguiente `Nodo` o la lista vacía.

En otras palabras, una lista enlazada es una secuencia de nodos, donde cada nodo contiene un elemento de datos y un enlace (o referencia) al siguiente nodo en la secuencia. El último nodo en la lista tiene un enlace nulo, indicando el final de la lista.

Ejemplo: Proceso de construcción de una lista añadiendo nodos a la cabeza de la lista.

Supongamos que queremos construir una lista con los elementos 1, 2 y 3. Inicialmente, la lista está vacía (referencia nula).

1. Añadimos el nodo con el valor 1. Este nodo se convierte en la cabeza de la lista, y su referencia siguiente es nula.
2. Añadimos el nodo con el valor 2. Este nuevo nodo se convierte en la nueva cabeza de la lista, y su referencia siguiente apunta al nodo que contiene el valor 1.
3. Añadimos el nodo con el valor 3. Este nodo se convierte en la nueva cabeza de la lista, y su referencia siguiente apunta al nodo que contiene el valor 2.

El resultado final es una lista enlazada donde el primer nodo contiene el valor 3, el segundo nodo contiene el valor 2, y el tercer nodo contiene el valor 1.

2.2 Implementación de la lista simplemente enlazada

Una lista simplemente enlazada se compone de nodos, donde cada nodo contiene un dato y una referencia al siguiente nodo en la lista.

2.2.1 Definición de la clase Nodo

A continuación, se muestra un ejemplo de la definición de la clase `Nodo` en Java:

```
class Nodo<T> {
    T dato;
    Nodo siguiente;

    Nodo(T dato) {
        this.dato = dato;
        this.siguiente = null;
    }
}
```

Listing 14: Clase Nodo en Java

2.2.2 Agregar elementos a la lista (proceso paso a paso manual)

Para agregar un elemento a la lista, se crea un nuevo nodo con el dato deseado y se enlaza al principio de la lista. Esta forma de proceder siempre agrega nuevos nodos al principio de la lista.

1. Crear un nuevo nodo con el dato a insertar.

```
Nodo<Integer> primero = new Nodo<>(1);
```

2. Hacer que la referencia siguiente del nuevo nodo apunte a la cabeza actual de la lista.

```
primero.siguiente = null;
```

3. Actualizar la cabeza de la lista para que apunte al nuevo nodo.

Supongamos que la lista está vacía (referencia nula).

```
Nodo<Integer> primero = null;
```

1. Añadimos el nodo con el valor 1. Este nodo se convierte en la cabeza de la lista, y su referencia siguiente es nula.

```
Nodo<Integer> primero = new Nodo<>(1);
primero.siguiente = null;
```

2. Añadimos el nodo con el valor 2. Este nuevo nodo se convierte en la nueva cabeza de la lista, y su referencia siguiente apunta al nodo que contiene el valor 1.

```
Nodo<Integer> segundo = new Nodo<>(2);
segundo.siguiente = primero;
primero = segundo;
```

3. Añadimos el nodo con el valor 3. Este nodo se convierte en la nueva cabeza de la lista, y su referencia siguiente apunta al nodo que contiene el valor 2.

```
Nodo<Integer> tercero = new Nodo<>(3);
tercero.siguiente = primero;
primero = tercero;
```

2.2.3 Implementación de la lista enlazada como ADT

A continuación, se muestra un ejemplo de la implementación de la lista enlazada como un Abstract Data Type (ADT) en Java, con los métodos `addFirst`, `isEmpty`, y `size`:

```java
public class ListaEnlazada<T> {
    private Nodo<T> cabeza;
    private int size;

    private class Nodo {
        T dato;
        Nodo siguiente;

        Nodo(T dato) {
            this.dato = dato;
            this.siguiente = null;
        }
    }

    public ListaEnlazada() {
        this.cabeza = null;
        this.size = 0;
    }

    public void addFirst(T dato) {
        Nodo<T> nuevoNodo = new Nodo<>(dato);
        nuevoNodo.siguiente = cabeza;
        cabeza = nuevoNodo;
        size++;
    }

    public boolean isEmpty() {
        return cabeza == null;
    }

    public int size() {
        return size;
    }
}
```

Listing 15: Lista Enlazada como ADT en Java

Ejemplo en Python: Implementación de la lista enlazada como ADT, con métodos: `add_first`, `is_empty`, `size`.

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None

class ListaEnlazada:
    def __init__(self):
        self.cabeza = None
        self.size = 0

    def add_first(self, dato):
        nuevo_nodo = Nodo(dato)
        nuevo_nodo.siguiente = self.cabeza
        self.cabeza = nuevo_nodo
        self.size += 1
```

```
16
17      def is_empty(self):
18          return self.cabeza is None
19
20      def size(self):
21          return self.size
```

Listing 16: Lista Enlazada como ADT en Python

Ejemplo en C#: Implementación de la lista enlazada como ADT, con métodos: `AddFirst`, `IsEmpty`, `Size`.

```
1  public class Node<T>
2  {
3      public T Data { get; set; }
4      public Node<T> Next { get; set; }
5
6      public Node(T data)
7      {
8          Data = data;
9          Next = null;
10     }
11 }
12
13 public class LinkedList<T>
14 {
15     private Node<T> head;
16     private int size;
17
18     public LinkedList()
19     {
20         head = null;
21         size = 0;
22     }
23
24     public void AddFirst(T data)
25     {
26         Node<T> newNode = new Node<T>(data);
27         newNode.Next = head;
28         head = newNode;
29         size++;
30     }
31
32     public bool IsEmpty()
33     {
34         return head == null;
35     }
36
37     public int Size()
38     {
39         return size;
40     }
41 }
```

Listing 17: Lista Enlazada como ADT en C#

2.3 Otras operaciones con listas simplemente enlazadas

Además de las operaciones básicas de agregar elementos y verificar si la lista está vacía, se pueden implementar otras operaciones útiles en listas simplemente enlazadas. A continuación, se describen algunas de estas operaciones, junto con su firma y una explicación conceptual de su implementación.

2.3.1 Eliminar el primer elemento de la lista

Lenguaje	Firma del método removeFirst()
Java	`public void removeFirst()`
Python	`def remove_first(self):`
C#	`public void RemoveFirst()`

Table 1: Firma del método removeFirst() en diferentes lenguajes

Implementación conceptual:

1. Verificar si la lista está vacía. Si lo está, no se puede eliminar ningún elemento.
2. Si la lista no está vacía, actualizar la cabeza de la lista para que apunte al segundo nodo.
3. Disminuir el tamaño de la lista en 1.

2.3.2 Agregar un elemento al final de la lista

Lenguaje	Firma del método addLast()
Java	`public void addLast()`
Python	`def add_last(self):`
C#	`public void AddLast()`

Table 2: Firma del método addLast en diferentes lenguajes

Implementación conceptual:

1. Crear un nuevo nodo con el dato a insertar.
2. Verificar si la lista está vacía. Si lo está, el nuevo nodo se convierte en la cabeza de la lista.
3. Si la lista no está vacía, recorrer la lista hasta llegar al último nodo.
4. Hacer que la referencia siguiente del último nodo apunte al nuevo nodo.
5. Aumentar el tamaño de la lista en 1.

2.3.3 Eliminar el último elemento de la lista

Implementación conceptual:

1. Verificar si la lista está vacía. Si lo está, no se puede eliminar ningún elemento.
2. Si la lista contiene un solo elemento, establecer la cabeza de la lista a `null`.

Lenguaje	Firma del método removeLast()
Java	`public void removeLast()`
Python	`def remove_last(self):`
C#	`public void RemoveLast()`

Table 3: Firma del método removeLast en diferentes lenguajes

3. Si la lista contiene más de un elemento, recorrer la lista hasta llegar al penúltimo nodo.
4. Hacer que la referencia siguiente del penúltimo nodo sea `null`.
5. Disminuir el tamaño de la lista en 1.

Ejemplo Java: Implementación de las operaciones eliminar primer elemento y agregar al final.

```java
public class ListaEnlazada<T> {
    private Nodo<T> cabeza;
    private int size;

    public ListaEnlazada() {
        this.cabeza = null;
        this.size = 0;
    }

    public void addFirst(T dato) {
        Nodo<T> nuevoNodo = new Nodo<>(dato);
        nuevoNodo.siguiente = cabeza;
        cabeza = nuevoNodo;
        size++;
    }

    public void addLast(T dato) {
        Nodo<T> nuevoNodo = new Nodo<>(dato);
        if (isEmpty()) {
            cabeza = nuevoNodo;
        } else {
            Nodo<T> current = cabeza;
            while (current.siguiente != null) {
                current = current.siguiente;
            }
            current.siguiente = nuevoNodo;
        }
        size++;
    }

    public void removeFirst() {
        if (!isEmpty()) {
            cabeza = cabeza.siguiente;
            size--;
        }
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
```

```
43          return cabeza == null;
44      }
45  }
```

Listing 18: Operaciones en Lista Enlazada en Java

2.3.4 Otras operaciones con listas simples

La siguiente tabla describe algunas operaciones adicionales que frecuentemente se implementan en listas simples:

Operación	Firma de la operación
Obtener dato en posición	`T get(int index)`
Asignar data en posición	`void set(int index, T data)`
Remover el datos en posición	`void remove(int index)`
Está un datos en la lista	`boolean contains(T data)`
Limpiar toda la lista	`void clear()`

Table 4: Otras operaciones con listas

Ejercicio: Dar una implementación de estas operaciones.

2.4 Iteradores sobre listas simplemente enlazadas

El recorrido de una lista enlazada es un proceso fundamental para realizar diversas operaciones sobre sus elementos. Un iterador es un objeto que permite recorrer una lista (u otra estructura de datos) y acceder a sus elementos de manera secuencial, sin exponer la estructura interna de la lista.

2.4.1 Proceso de recorrido de una lista

Para recorrer una lista enlazada, se utiliza un puntero (o referencia) que inicialmente apunta a la cabeza de la lista. Luego, se itera sobre la lista, moviendo el puntero al siguiente nodo en cada paso, hasta que el puntero llegue al final de la lista (es decir, apunte a `null`).

2.4.2 Operaciones que dependen de un recorrido de la lista

Muchas operaciones comunes en listas enlazadas requieren un recorrido de la lista. Algunas de estas operaciones son:

- **Recorrer una lista:** Visitar cada nodo de la lista y realizar alguna acción sobre su dato (por ejemplo, imprimirlo).
- **Buscar un elemento en una lista:** Recorrer la lista hasta encontrar un nodo cuyo dato coincida con el valor buscado.
- **Eliminar un elemento arbitrario de una lista:** Recorrer la lista hasta encontrar el nodo que se desea eliminar, y luego actualizar las referencias de los nodos adyacentes para eliminar el nodo de la lista.
- **Insertar un elemento en una posición arbitraria de una lista:** Recorrer la lista hasta encontrar la posición donde se desea insertar el nuevo nodo, y luego actualizar las referencias de los nodos adyacentes para insertar el nuevo nodo en la lista.

Ejemplos Java:

- Implementación del iterador de listas.
- Implementación de la búsqueda secuencial en la lista.

```java
import java.util.Iterator;

public class ListaEnlazada<T> implements Iterable<T> {
    private Nodo<T> cabeza;
    private int size;

    // Constructor, addFirst, addLast, removeFirst, size, isEmpty (como antes)

    @Override
    public Iterator<T> iterator() {
        return new IteradorListaEnlazada();
    }

    private class IteradorListaEnlazada implements Iterator<T> {
        private Nodo<T> current = cabeza;

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException();
            }
            T dato = current.dato;
            current = current.siguiente;
            return dato;
        }
    }
}
```

Listing 19: Iterador de Listas en Java

```java
public class ListaEnlazada<T> {
    // ... (codigo anterior)

    public boolean buscar(T valor) {
        Nodo<T> current = cabeza;
        while (current != null) {
            if (current.dato.equals(valor)) {
                return true;
            }
            current = current.siguiente;
        }
        return false;
    }
}
```

Listing 20: Búsqueda Secuencial en Java

Ejemplos Python:

- Implementación del iterador de listas.
- Implementación de la búsqueda secuencial en la lista.

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None

class ListaEnlazada:
    def __init__(self):
        self.cabeza = None
        self.size = 0

    # add, isEmpty, size (como antes)

    def __iter__(self):
        self.current = self.cabeza
        return self

    def __next__(self):
        if self.current is None:
            raise StopIteration
        dato = self.current.dato
        self.current = self.current.siguiente
        return dato
```

Listing 21: Iterador de Listas en Python

```python
class ListaEnlazada:
    # ... (codigo anterior)

    def buscar(self, valor):
        current = self.cabeza
        while current is not None:
            if current.dato == valor:
                return True
            current = current.siguiente
        return False
```

Listing 22: Búsqueda Secuencial en Python

Ejemplos C#:

- Implementación del iterador de listas.
- Implementación de la búsqueda secuencial en la lista.

```csharp
using System.Collections;
using System.Collections.Generic;

public class LinkedList<T> : IEnumerable<T>
{
    private Node<T> head;
    private int size;

    // Constructor, AddFirst, AddLast, RemoveFirst, Size, IsEmpty (como antes)

```

```
    public IEnumerator<T> GetEnumerator()
    {
        Node<T> current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Listing 23: Iterador de Listas en C#

```
public class LinkedList<T>
{
    // ... (codigo anterior)

    public bool Buscar(T valor)
    {
        Node<T> current = head;
        while (current != null)
        {
            if (current.Data.Equals(valor))
            {
                return true;
            }
            current = current.Next;
        }
        return false;
    }
}
```

Listing 24: Búsqueda Secuencial en C#

3 Listas Doblemente Enlazadas

Una lista doblemente enlazada es una estructura de datos en la que cada nodo tiene dos enlaces: uno al nodo siguiente y otro al nodo anterior. Esto permite recorrer la lista en ambas direcciones, lo que facilita ciertas operaciones en comparación con las listas simplemente enlazadas.

3.1 ADT e Implementación de la lista doblemente enlazada

El ADT de una lista doblemente enlazada incluye operaciones como:

- `addFirst(T data)`: Agrega un elemento al principio de la lista.
- `addLast(T data)`: Agrega un elemento al final de la lista.
- `removeFirst()`: Elimina el primer elemento de la lista.
- `removeLast()`: Elimina el último elemento de la lista.

- `getFirst()`: Obtiene el primer elemento de la lista.
- `getLast()`: Obtiene el último elemento de la lista.
- `size()`: Devuelve el número de elementos en la lista.
- `isEmpty()`: Indica si la lista está vacía.

A continuación, se muestra un ejemplo de implementación básica de una lista doblemente enlazada en Java:

```java
public class ListaDoblementeEnlazada<T> {
    private NodoDoble<T> cabeza;
    private NodoDoble<T> cola;
    private int size;

    private class NodoDoble<T> {
        T dato;
        NodoDoble<T> siguiente;
        NodoDoble<T> anterior;

        NodoDoble(T dato) {
            this.dato = dato;
            this.siguiente = null;
            this.anterior = null;
        }
    }

    public ListaDoblementeEnlazada() {
        this.cabeza = null;
        this.cola = null;
        this.size = 0;
    }

    public void addFirst(T dato) {
        NodoDoble<T> nuevoNodo = new NodoDoble<>(dato);
        if (isEmpty()) {
            cabeza = nuevoNodo;
            cola = nuevoNodo;
        } else {
            nuevoNodo.siguiente = cabeza;
            cabeza.anterior = nuevoNodo;
            cabeza = nuevoNodo;
        }
        size++;
    }

    public void addLast(T dato) {
        NodoDoble<T> nuevoNodo = new NodoDoble<>(dato);
        if (isEmpty()) {
            cabeza = nuevoNodo;
            cola = nuevoNodo;
        } else {
            nuevoNodo.anterior = cola;
            cola.siguiente = nuevoNodo;
            cola = nuevoNodo;
        }
        size++;
    }
```

```
49
50      public T getFirst() {
51          if (isEmpty()) {
52              throw new NoSuchElementException("La lista esta vacia");
53          }
54          return cabeza.dato;
55      }
56
57      public T getLast() {
58          if (isEmpty()) {
59              throw new NoSuchElementException("La lista esta vacia");
60          }
61          return cola.dato;
62      }
63
64
65      public void removeFirst() {
66          if (isEmpty()) {
67              return;
68          }
69          if (cabeza == cola) {
70              cabeza = null;
71              cola = null;
72          } else {
73              cabeza = cabeza.siguiente;
74              cabeza.anterior = null;
75          }
76          size--;
77      }
78
79     public void removeLast() {
80          if (isEmpty()) {
81              return;
82          }
83          if (cabeza == cola) {
84              cabeza = null;
85              cola = null;
86          } else {
87              cola = cola.anterior;
88              cola.siguiente = null;
89          }
90          size--;
91      }
92
93
94      public int size() {
95          return size;
96      }
97
98      public boolean isEmpty() {
99          return size == 0;
100     }
101 }
```

Listing 25: Lista Doblemente Enlazada en Java

3.2 La cola de doble terminación (DEQUE)

Una cola de doble terminación (Doubly Ended Queue (DEQUE)) es una generalización de una cola en la que se pueden insertar y eliminar elementos tanto al principio como al final de la estructura. Es decir, un DEQUE puede funcionar como una cola (First In, First Out (FIFO)) o como una pila (LIFO).

El ADT de un DEQUE incluye operaciones como:

- `addFirst(T data)`: Agrega un elemento al principio del DEQUE.
- `addLast(T data)`: Agrega un elemento al final del DEQUE.
- `removeFirst()`: Elimina el primer elemento del DEQUE.
- `removeLast()`: Elimina el último elemento del DEQUE.
- `getFirst()`: Obtiene el primer elemento del DEQUE.
- `getLast()`: Obtiene el último elemento del DEQUE.
- `size()`: Devuelve el número de elementos en el DEQUE.
- `isEmpty()`: Indica si el DEQUE está vacío.

Ejercicio: Implementar el DEQUE utilizando una lista doblemente enlazada.

3.3 Colas circulares

Una cola circular es una estructura de datos que utiliza un buffer de tamaño fijo y trata el principio y el final del buffer como si estuvieran conectados. Esto permite reutilizar el espacio de memoria de los elementos que se han eliminado de la cola.

El ADT de una cola circular incluye operaciones como:

- `enqueue(T data)`: Agrega un elemento al final de la cola.
- `dequeue()`: Elimina el elemento del principio de la cola.
- `peek()`: Obtiene el elemento del principio de la cola sin eliminarlo.
- `isFull()`: Indica si la cola está llena.
- `isEmpty()`: Indica si la cola está vacía.
- `size()`: Devuelve el número de elementos en la cola.

Ejercicio: Implementar una cola circular.

Bibliografía

[1] Aditya Bhargava. *Grokking Algorithms*. Manning Publications, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy*. CareerMonk Publications, 2011.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2005.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.