Listas enlazadas

Jorge Mario Londoño Peláez & Varias AI

February 10, 2025

1 Listas Simplemente Enlazadas

1.1 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.

1.2 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.

1.2.1 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 1: Clase Nodo en Java

1.2.2 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;
```

1.2.3 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 2: 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

```

```python
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

    def is_empty(self):
        return self.cabeza is None

    def size(self):
        return self.size
```

Listing 3: Lista Enlazada como ADT en Python

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

```csharp
public class Node<T>
{
    public T Data { get; set; }
    public Node<T> Next { get; set; }

    public Node(T data)
    {
        Data = data;
        Next = null;
    }
}

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

    public LinkedList()
    {
        head = null;
        size = 0;
    }

    public void AddFirst(T data)
    {
        Node<T> newNode = new Node<T>(data);
        newNode.Next = head;
        head = newNode;
        size++;
    }

    public bool IsEmpty()
    {
        return head == null;
    }

    public int Size()
    {
```

```
39          return size;
40      }
41  }
```

Listing 4: Lista Enlazada como ADT en C#

1.3 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.

1.3.1 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.

1.3.2 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.

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

1.3.3 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`.
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--;
```

```
35          }
36      }
37
38      public int size() {
39          return size;
40      }
41
42      public boolean isEmpty() {
43          return cabeza == null;
44      }
45  }
```

Listing 5: Operaciones en Lista Enlazada en Java

1.3.4 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.

1.4 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.

1.4.1 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`).

1.4.2 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 6: 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 7: Búsqueda Secuencial en Java

Ejemplos Python:

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

```
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 8: Iterador de Listas en 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 9: Búsqueda Secuencial en Python

Ejemplos C#:

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

```
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 10: 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 11: Búsqueda Secuencial en C#

2 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.

2.1 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++;
    }
```

```
36
37      public void addLast(T dato) {
38          NodoDoble<T> nuevoNodo = new NodoDoble<>(dato);
39          if (isEmpty()) {
40              cabeza = nuevoNodo;
41              cola = nuevoNodo;
42          } else {
43              nuevoNodo.anterior = cola;
44              cola.siguiente = nuevoNodo;
45              cola = nuevoNodo;
46          }
47          size++;
48      }
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 12: Lista Doblemente Enlazada en Java

2.2 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 (Last In, First Out (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.

2.3 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.