Estructuras dinámicas

Licenciatura en Gestión de Tecnología de la Información
Tecnicatura Universitaria en Desarrollo de Software

Ing. Mariano Martínez

UADE

Contenido

- Listas vinculadas

Limitaciones de los arreglos

```
int[] a = new int[100];
```

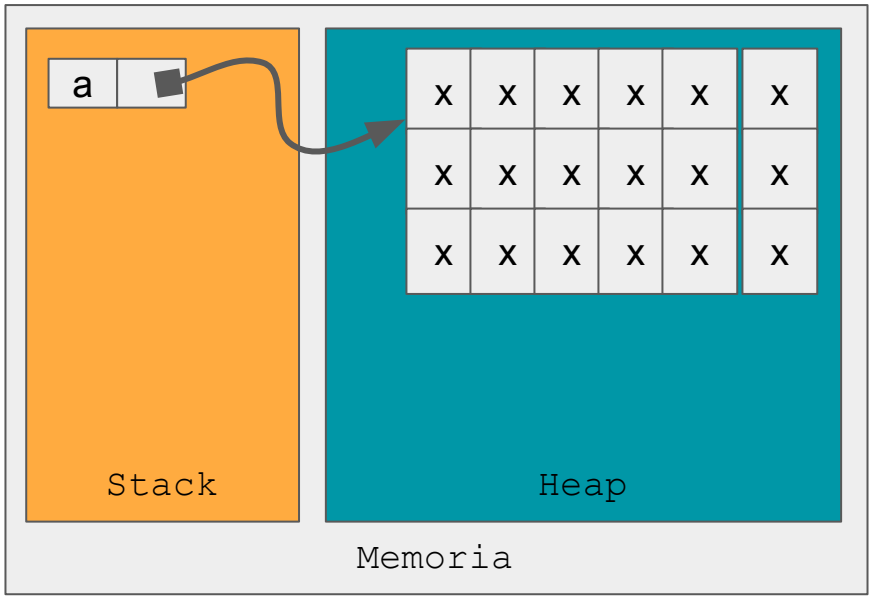
Que pasa si solo utilizo 10?

Desperdicio 90%

```
int[] a = new int[10];
```

Que pasa si necesito 11?

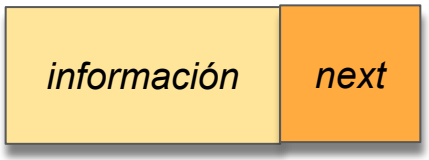
No me alcanza



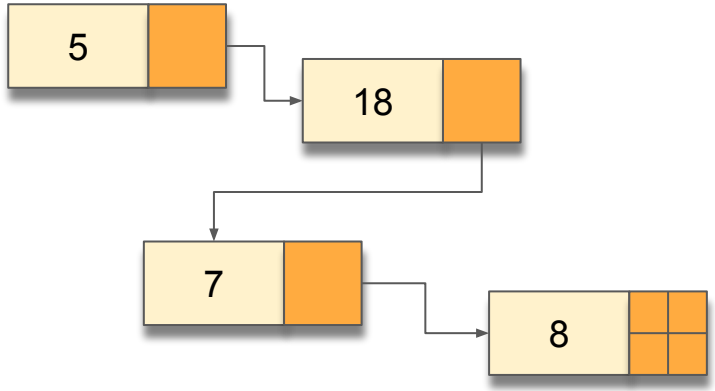
Listas vinculadas

Óptimas para cantidades variables de elementos.

Sobrecarga de información, por lo tanto no son óptimas para pocos elementos



nodo



Nodo

```
public class Nodo {
    private int info;
    private Nodo next;

    public Nodo() {
        this.next = null;
    }

    public void setInfo(int value) {
        this.info = value;
    }

    public int getInfo() {
        return this.info;
    }

    public void setNext(Nodo next) {
        this.next = next;
    }

    public Nodo getNext() {
        return this.next;
    }
}
```

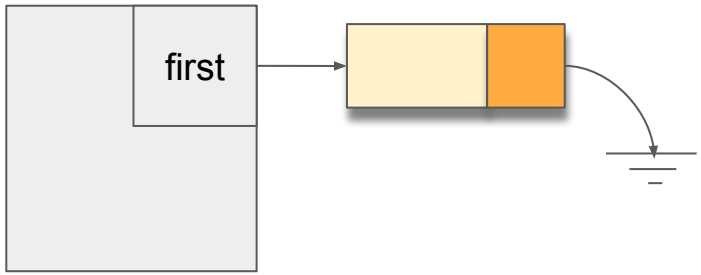
Lista

```
public class Lista {
   Nodo primero;
   Nodo ultimo;

   public Lista() {
       this.primero = new Nodo();
   }

      public void add(int x) {
       Nodo nuevo = new Nodo();
       nuevo.setInfo(x);

       Nodo pivote = new Nodo();
       pivote = this.primero;
       while (pivote.getNext() != null) {
           pivote = pivote.getNext();
       }
       pivote.setNext(nuevo);
   }
```

```
    public String toString() {
    Nodo pivote;
    String out = "";
    pivote = primero.getNext();
    while (pivote != null) {
        out = out + " " + pivote.getInfo();
        pivote = pivote.getNext();
    }
    return out;
    }
}
```

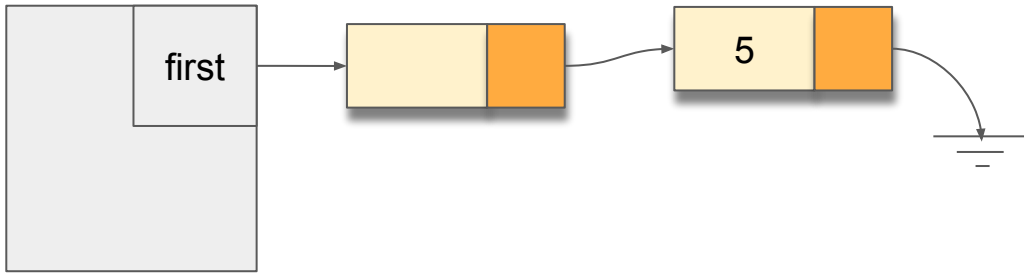
UADE

Lista



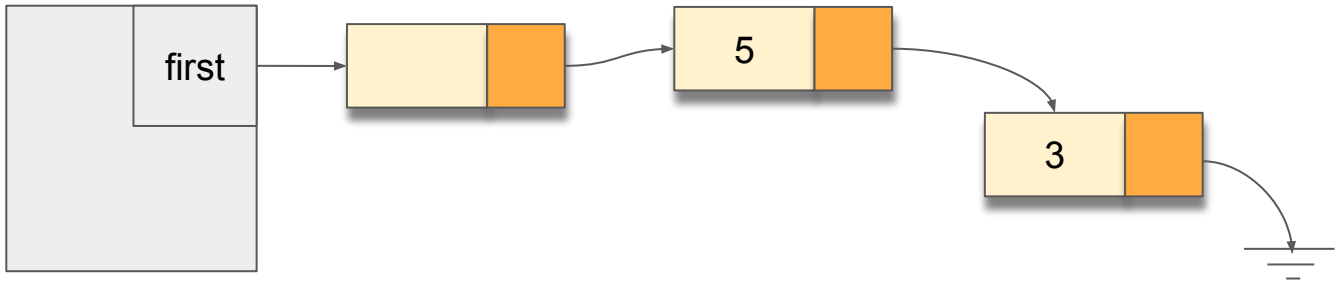

```
Lista l = new Lista();
```

Lista::Add



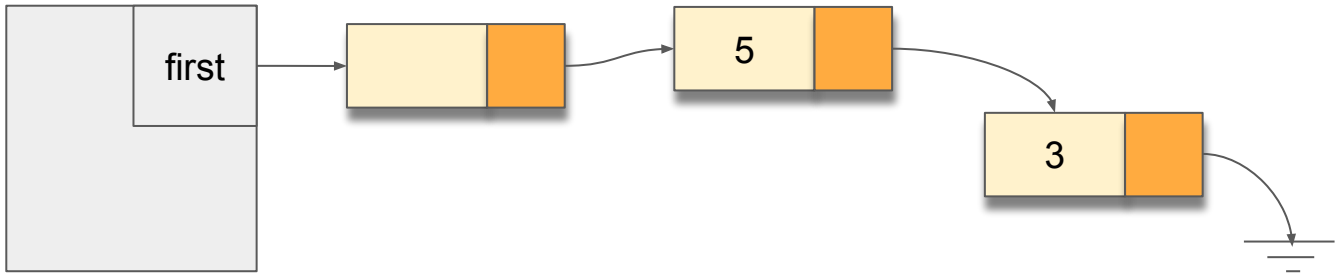

```
l.add(5);
```

Lista::Add



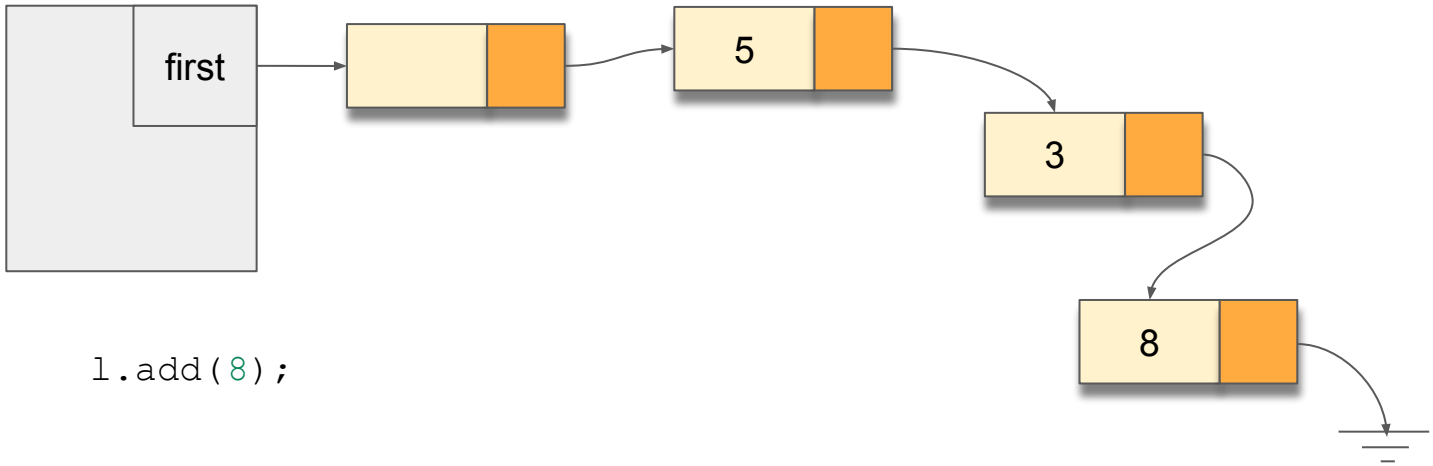

```
l.add(3);
```

Lista::Add



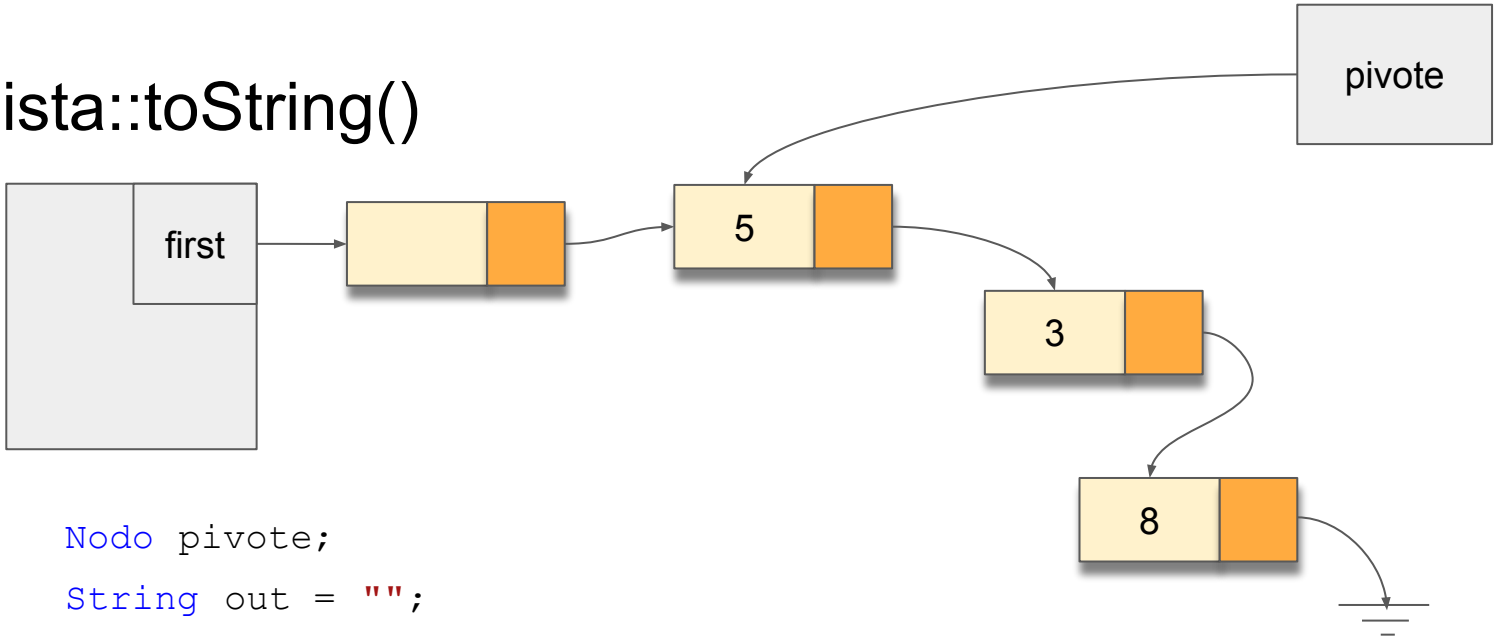

```
l.add(3);
```

Lista::Add




```
l.add(8);
```

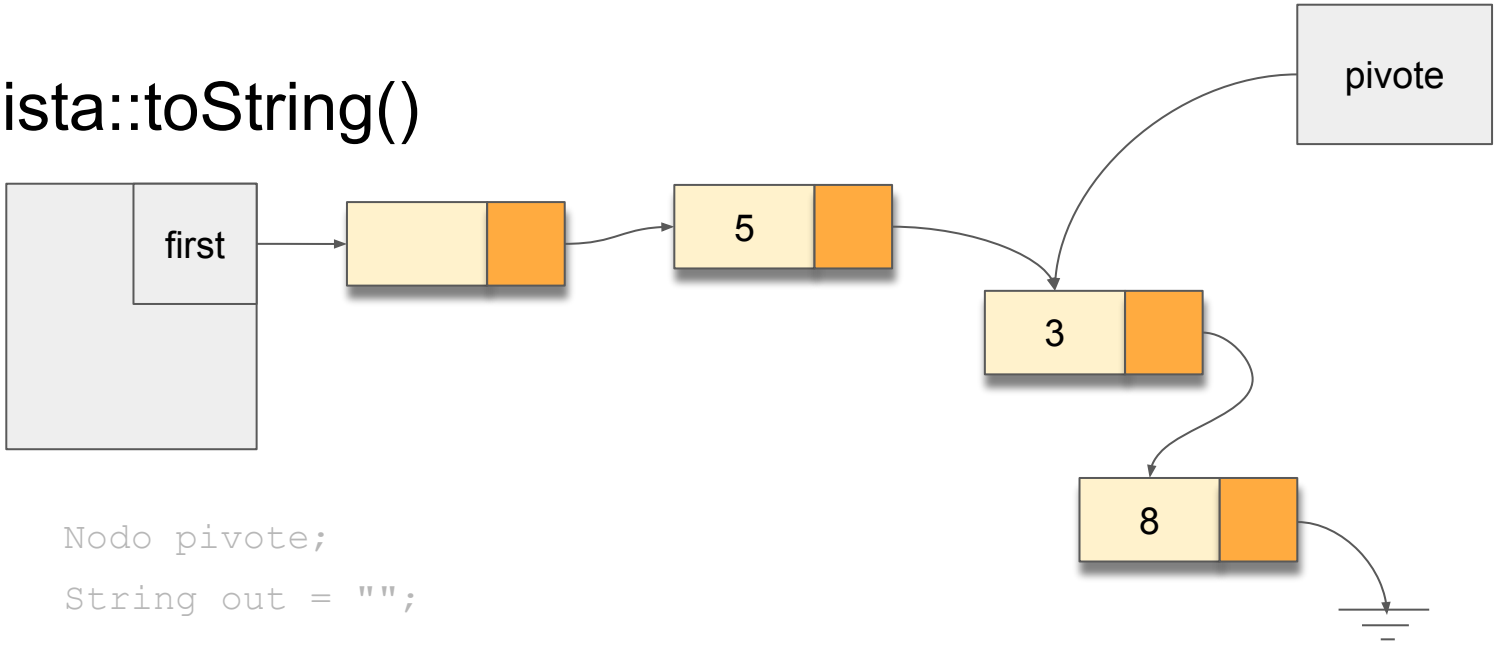
Lista::toString()




```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out:

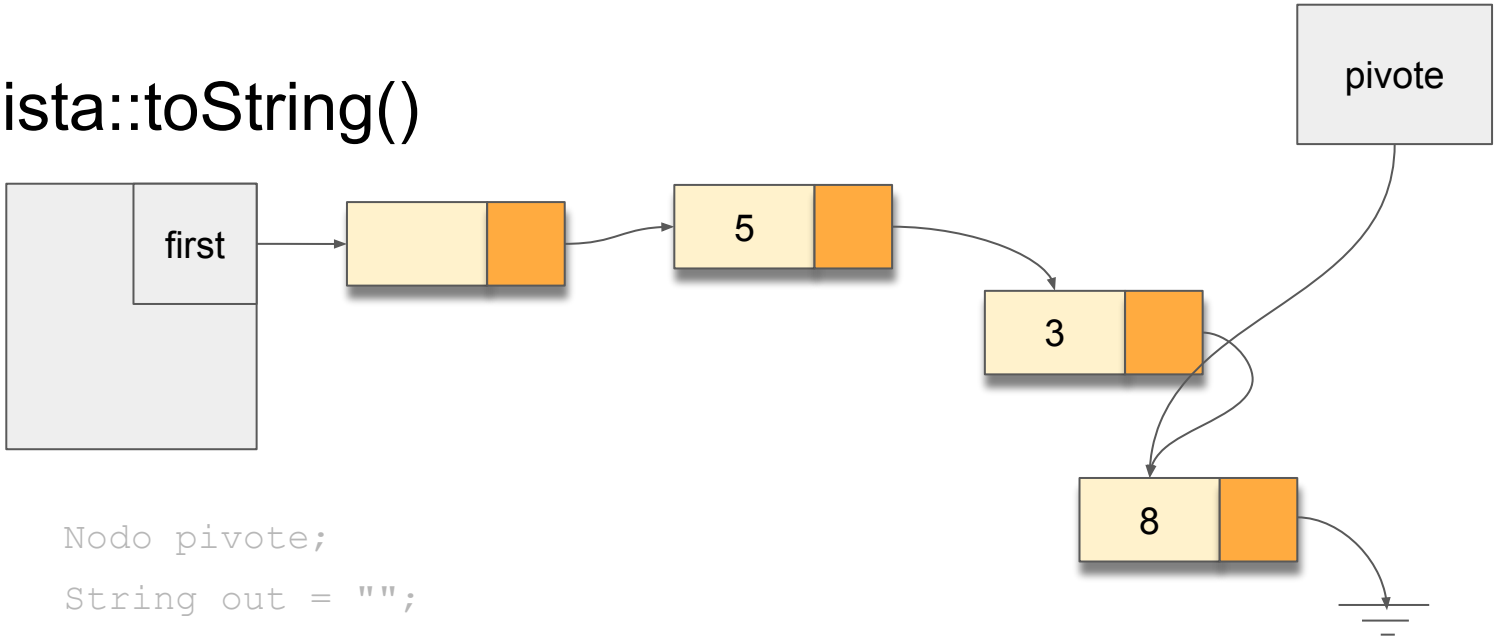
Lista::toString()




```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out: 5

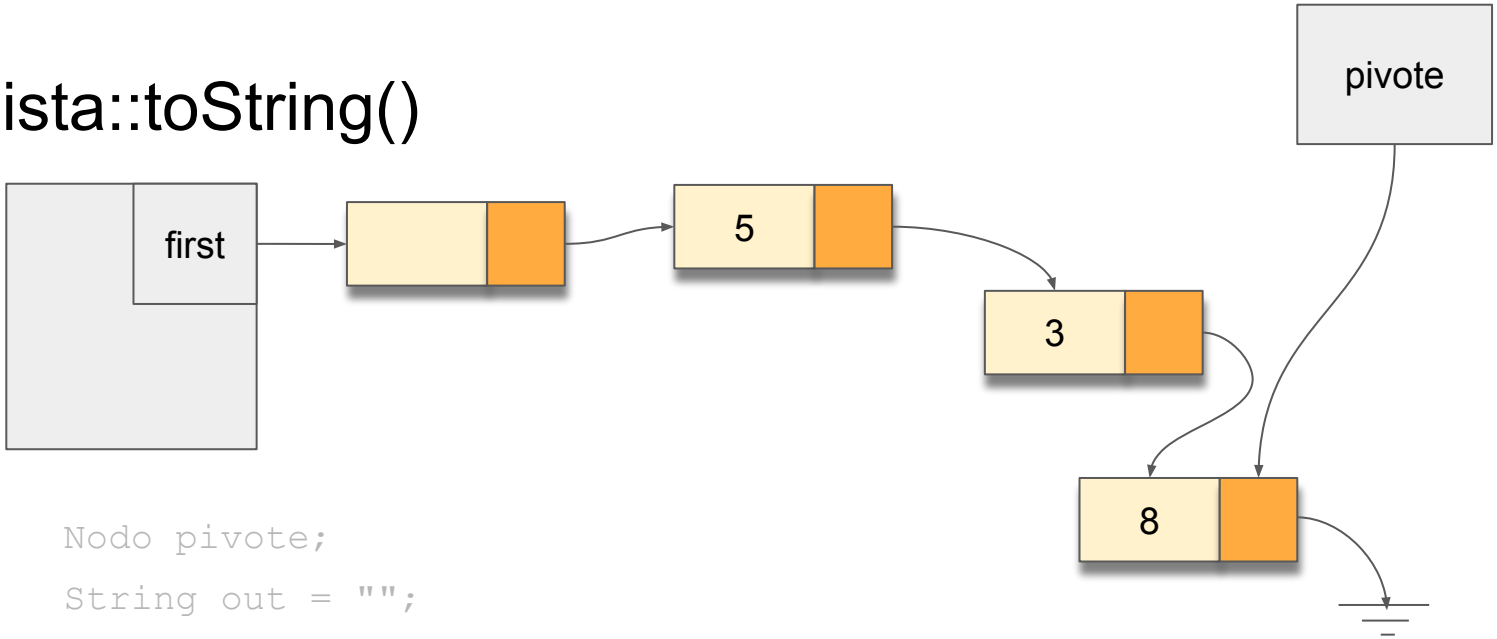
Lista::toString()




```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out: 5 3

Lista::toString()



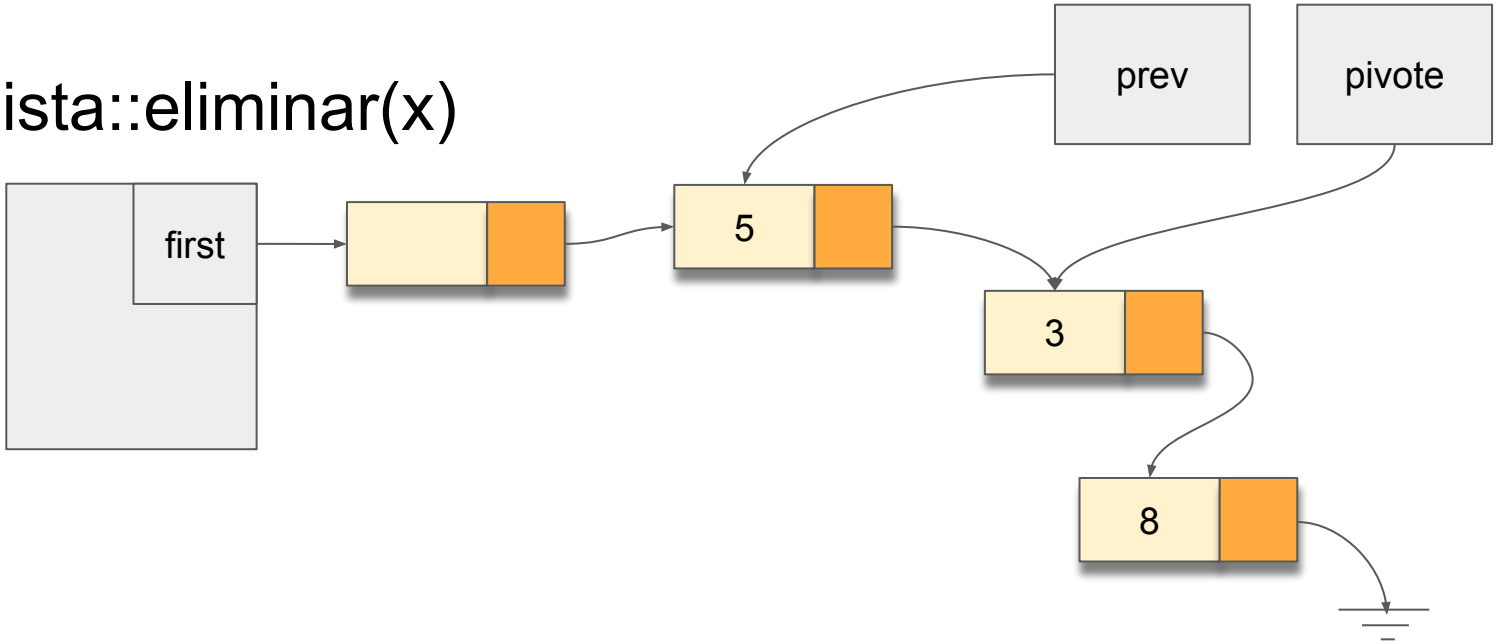

```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out: 5 3 8

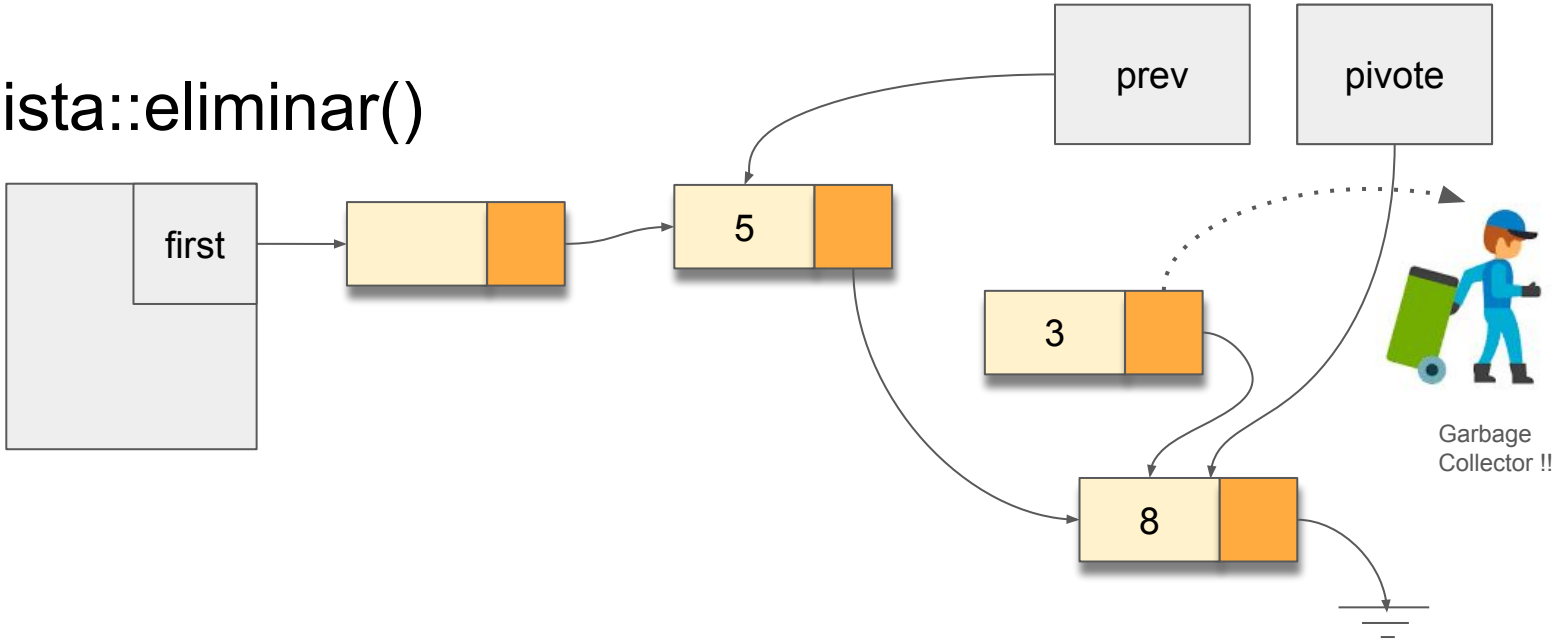
Lista::existe()

```
public Nodo existe(int value) {
    Nodo pivote;
    String out = "";

    pivote = primero.getNext();
    while (pivote != null) {
        if (pivote.getInfo() == value) {
            return pivote;
        }
        pivote = pivote.getNext();
    }

    return null;
}
```

Lista::eliminar(x)



Lista::eliminar()

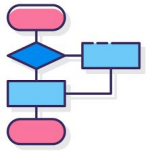


GC: Cada tanto recopila todos los nodos perdidos
Pro: Se responsabiliza de los memory leaks
Contra: Impredecible

Lista::eliminar()

```
public void eliminar(int value) {
        Nodo prev;
        Nodo pivote;

        prev = this.primero;
        pivote = prev.getNext();
        while ((pivote != null) && (pivote.getInfo() != value)) {
            prev = pivote;
            pivote = prev.getNext();
        }
        if (pivote != null) {
            prev.setNext(pivote.getNext());

        }
    }
```

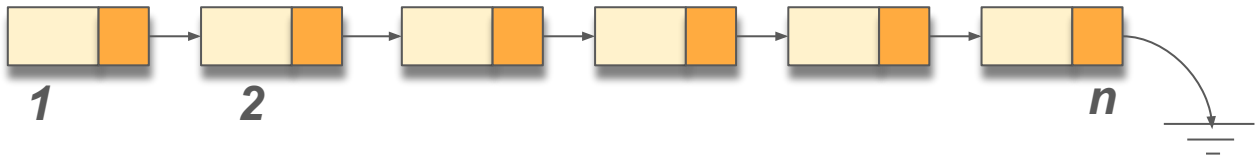
Práctica



1. Implemente una lista vinculada
    a. Método para determinar la lista vacía
    b. Método para agregar un elemento
    c. Método para eliminar la primer ocurrencia de un valor
    d. Método para buscar la primer ocurrencia de un valor
    e. Método para eliminar TODAS las ocurrencias de un valor
    f. Método para convertir a un solo String

Análisis de costos

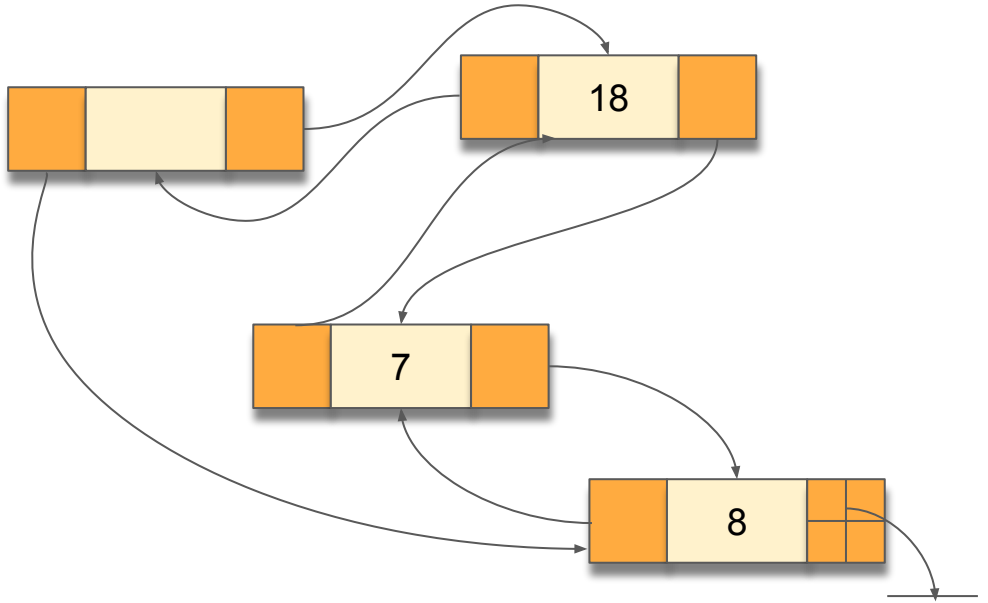
- Determinar vacía: O(1)
- Agregar: O(n)
- Buscar: O(n)
- Eliminar: O(n)
- Listar: O(n)



Listas doblemente vinculadas

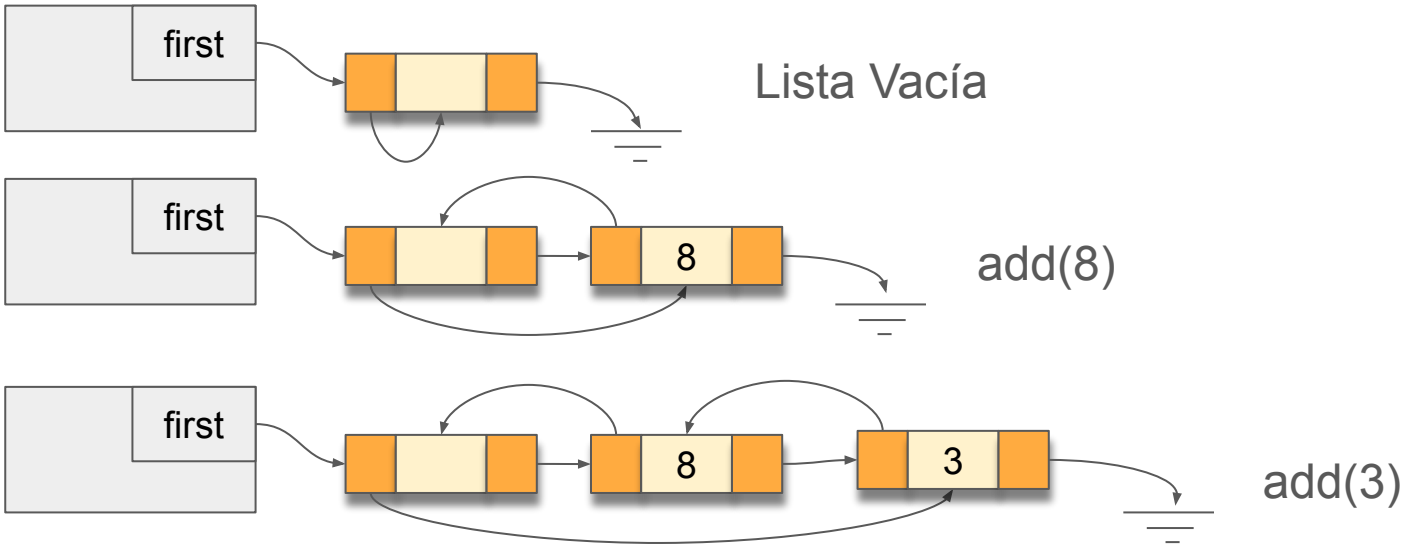
Paradójicamente más sencillas que las simples

nodo

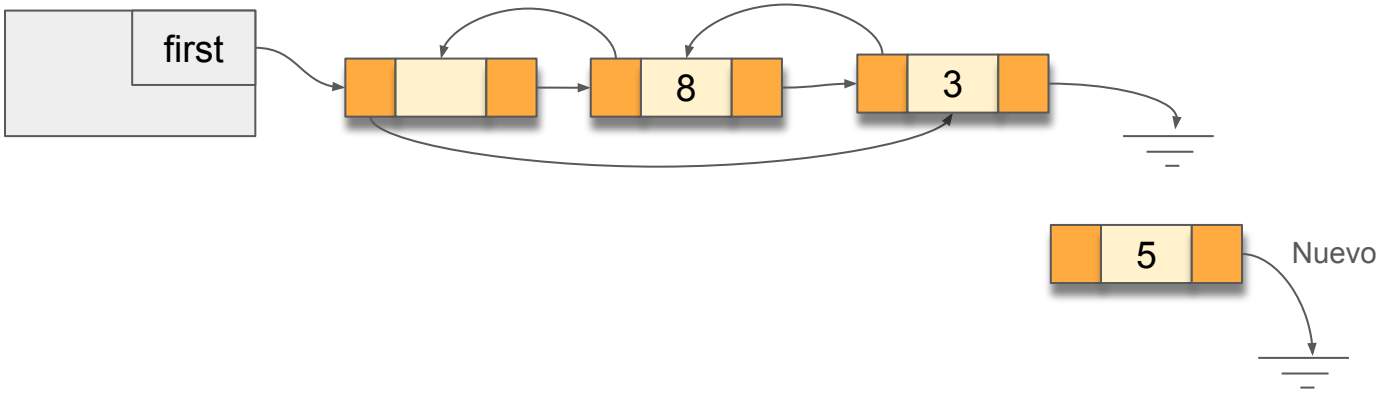


Listas doblemente vinculadas

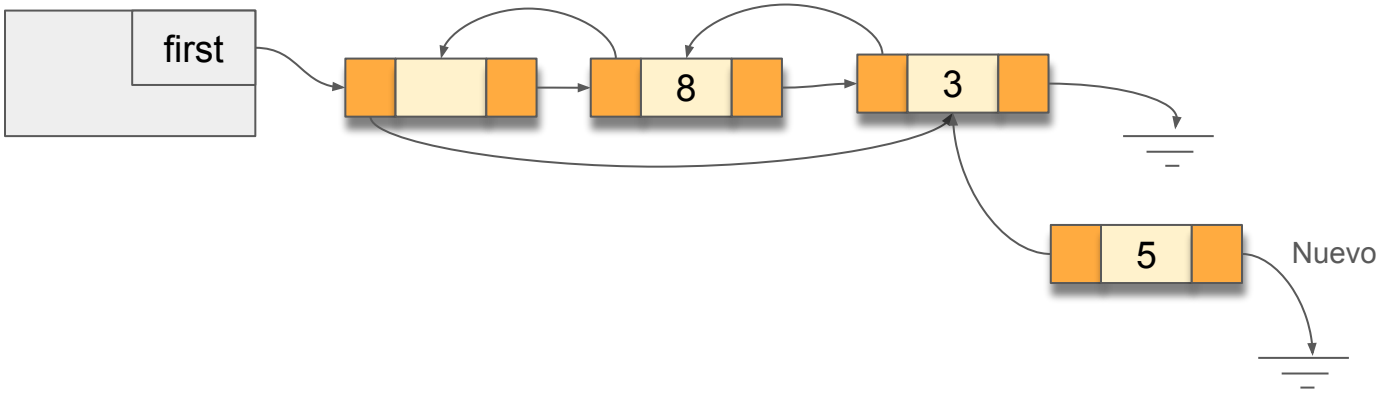
Hay un nodo ficticio, el inicial



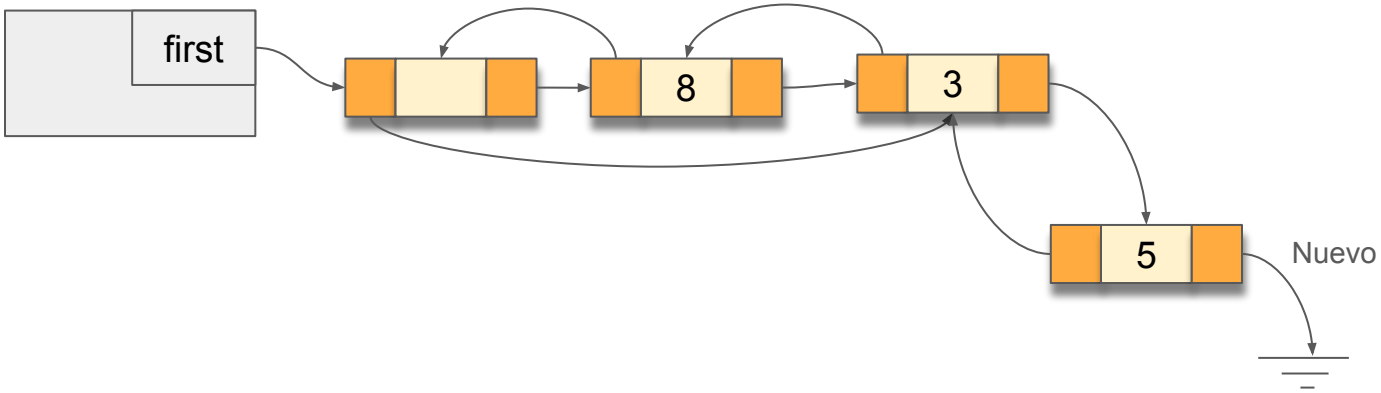
Lista doble: Agregar



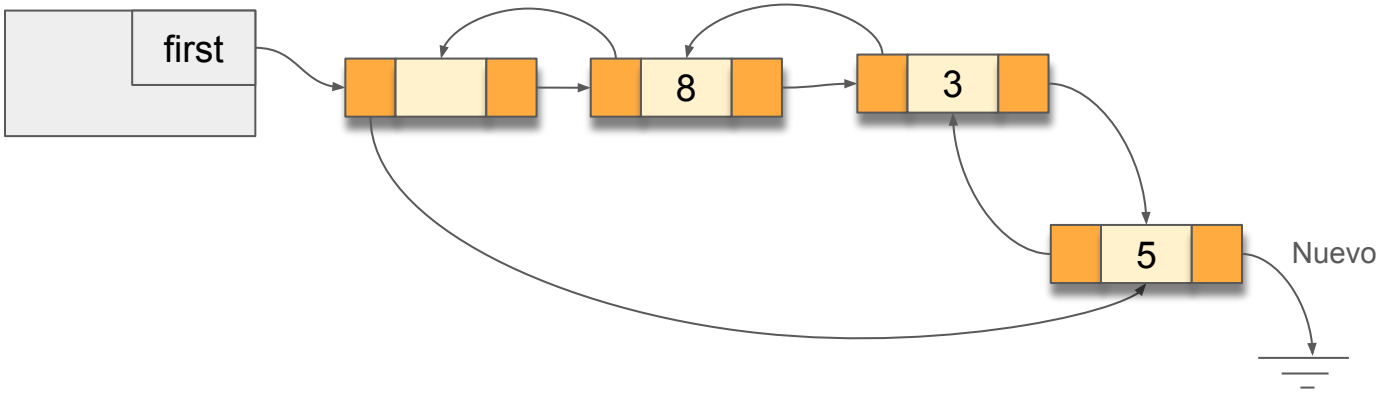
Lista doble: Agregar



Lista doble: Agregar



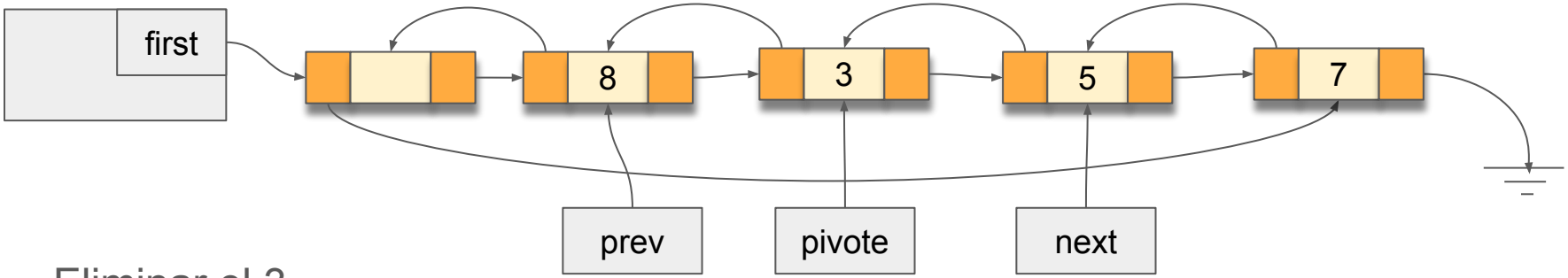
Lista doble: Agregar



Lista doble: Agregar

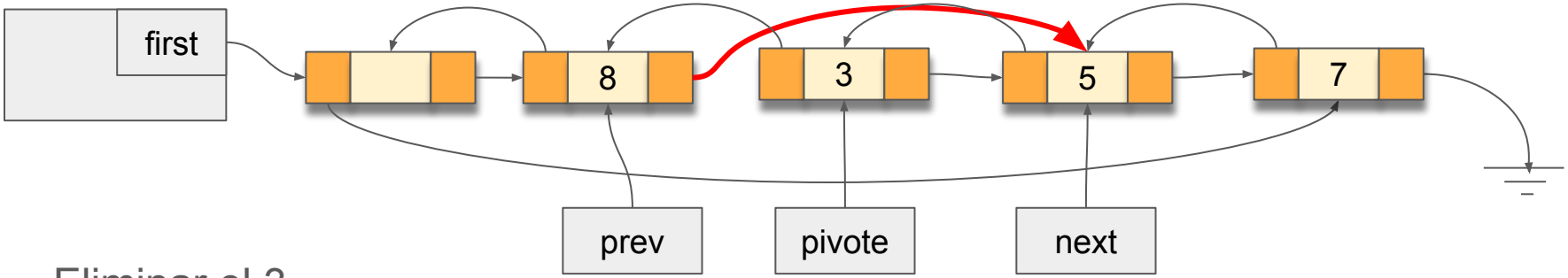
```
public void add(int x) {
    // Creo el nuevo
    NodoDoble nuevo = new NodoDoble();
    nuevo.setInfo(x);
    // Modificaciones sobre el nuevo
    NodoDoble ultimo = this.primero.getPrev();
    nuevo.setPrev(ultimo);

    // Modificaciones sobre el ultimo
    ultimo.setNext(nuevo);
    this.primero.setPrev(nuevo);
}
```

Lista doble: Eliminar



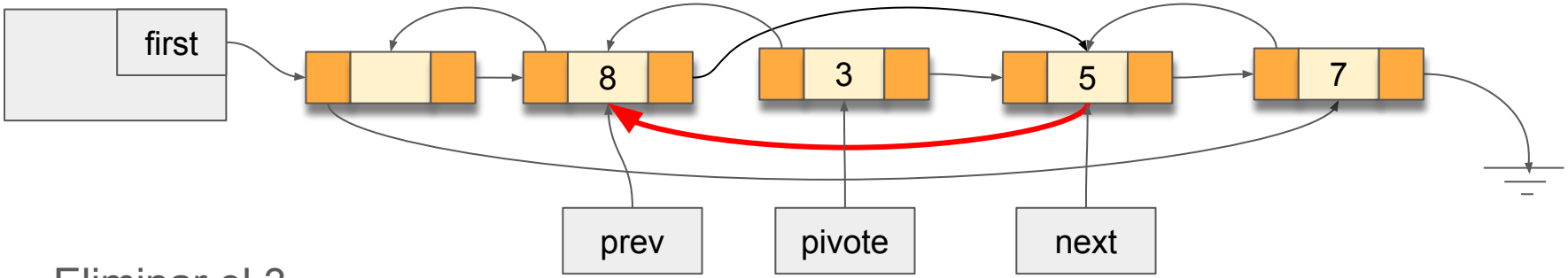
Eliminar el 3

Lista doble: Eliminar



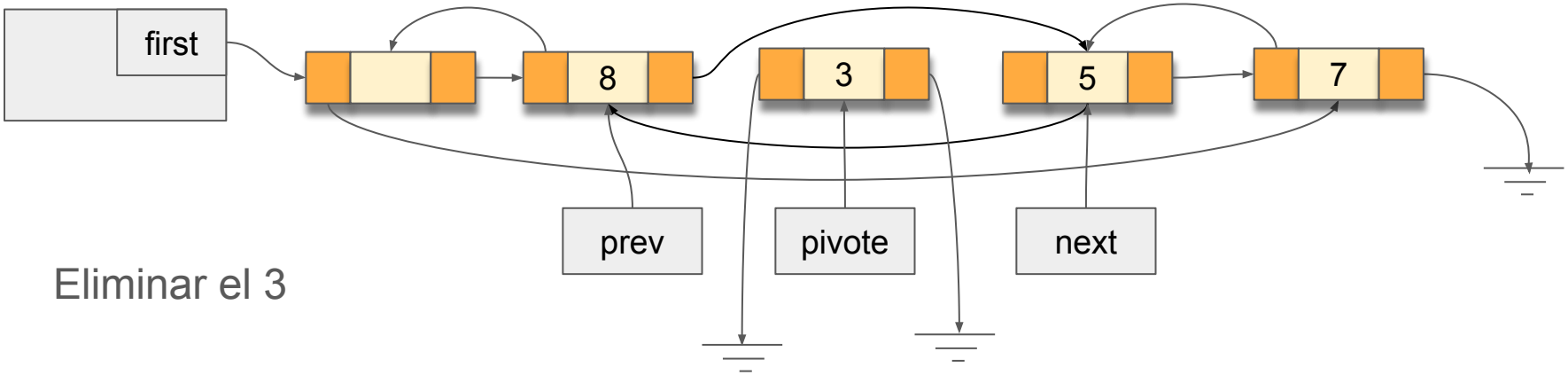
Eliminar el 3

Lista doble: Eliminar



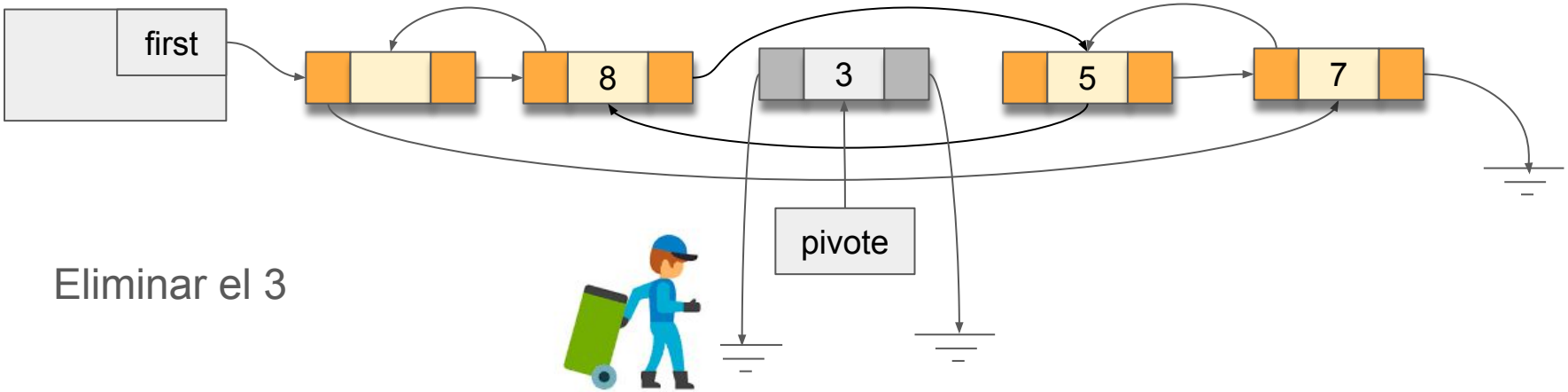
Eliminar el 3

Lista doble: Eliminar



Eliminar el 3

Lista doble: Eliminar

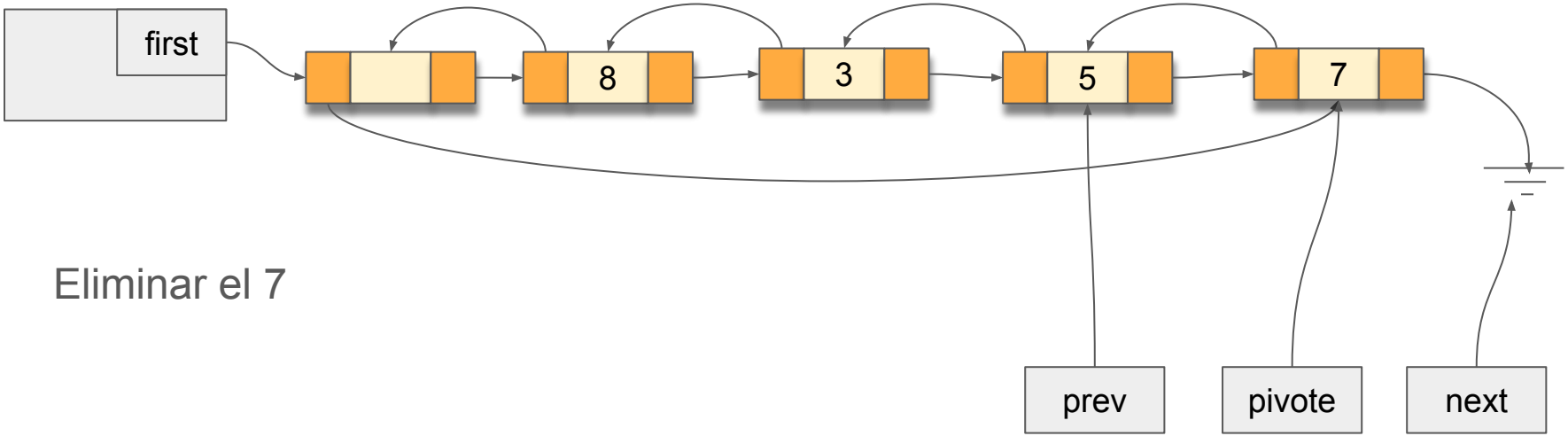


Eliminar el 3

Lista doble: Eliminar

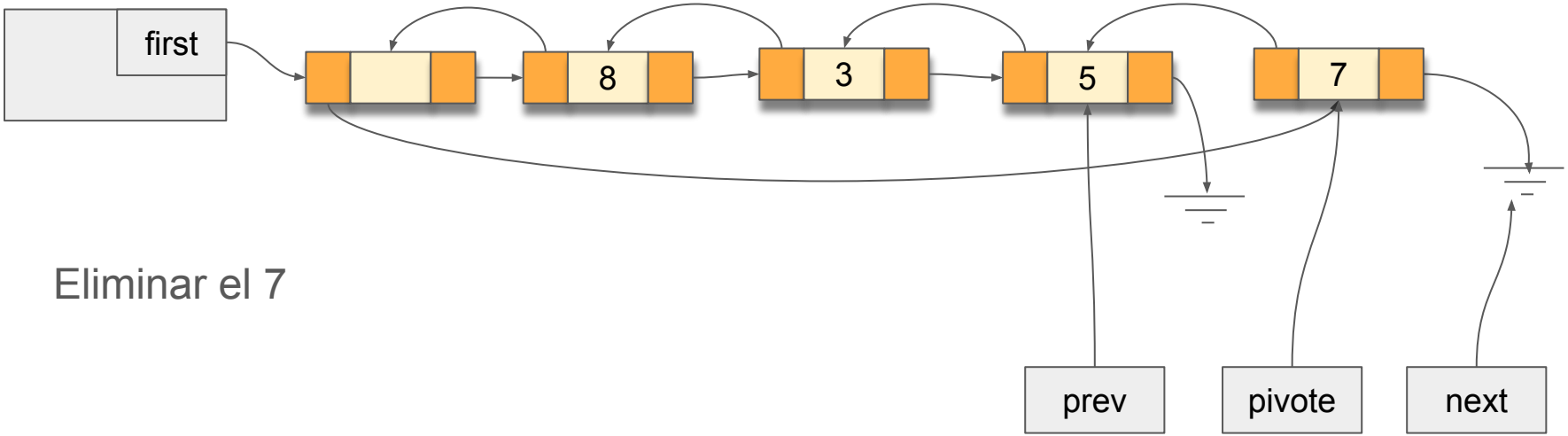
```
public void eliminar(int value) {
    NodoDoble pivote;

    pivote =  this.existe(value);

    if (pivote != null) {
        NodoDoble prev = pivote.getPrev();
        NodoDoble next = pivote.getNext();

        prev.setNext(next);
        next.setPrev(prev);

        pivote.setNext( null);
        pivote.setNext( null);
    }
}
```

Lista doble: Eliminar



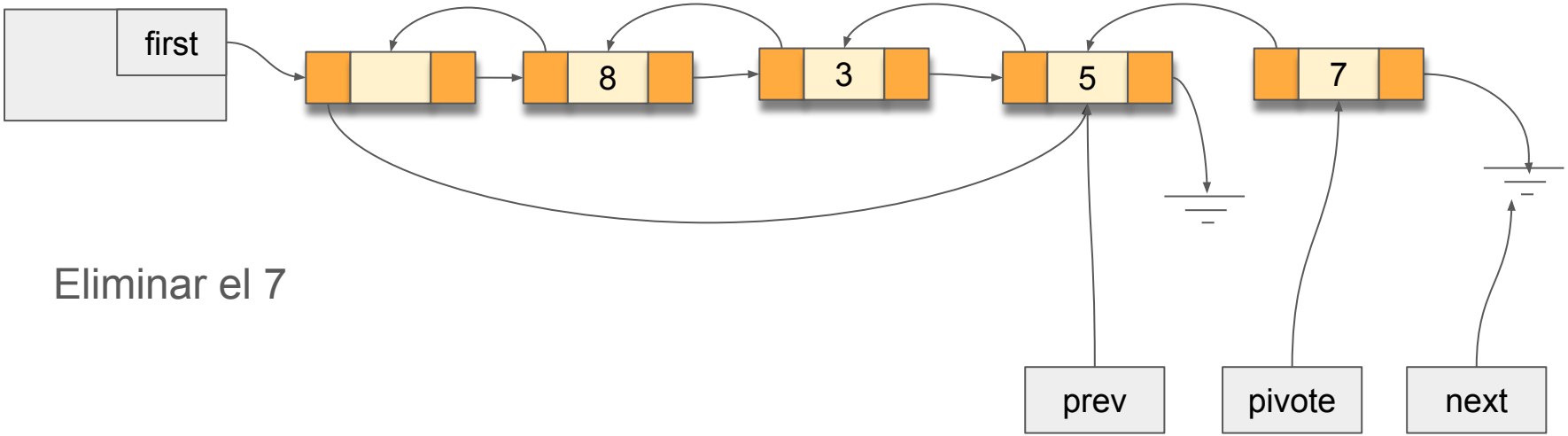
Eliminar el 7

Lista doble: Eliminar



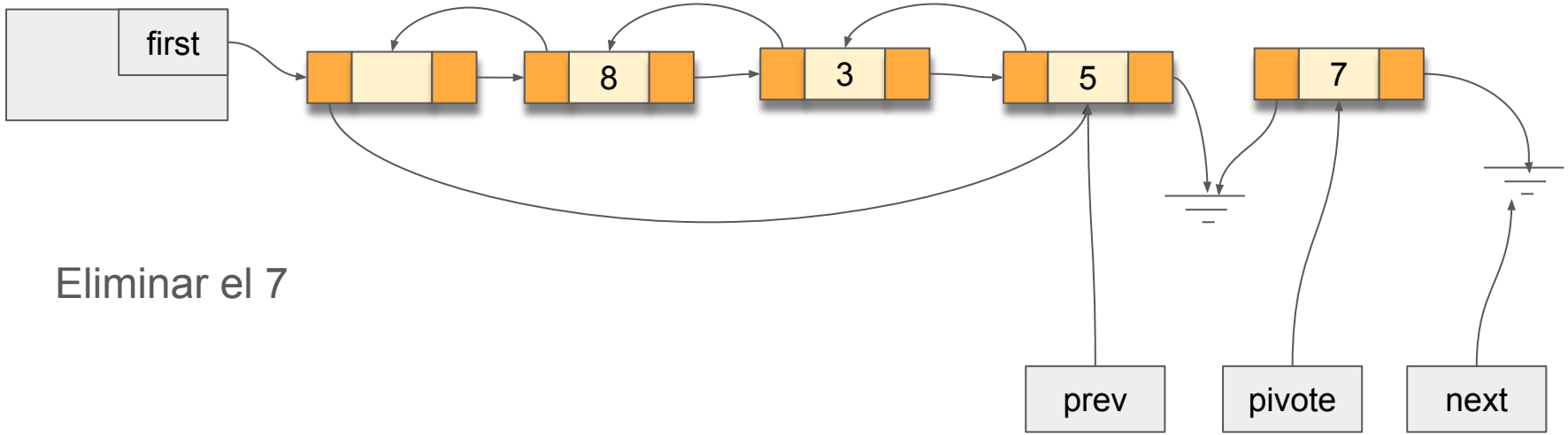
Eliminar el 7

Lista doble: Eliminar



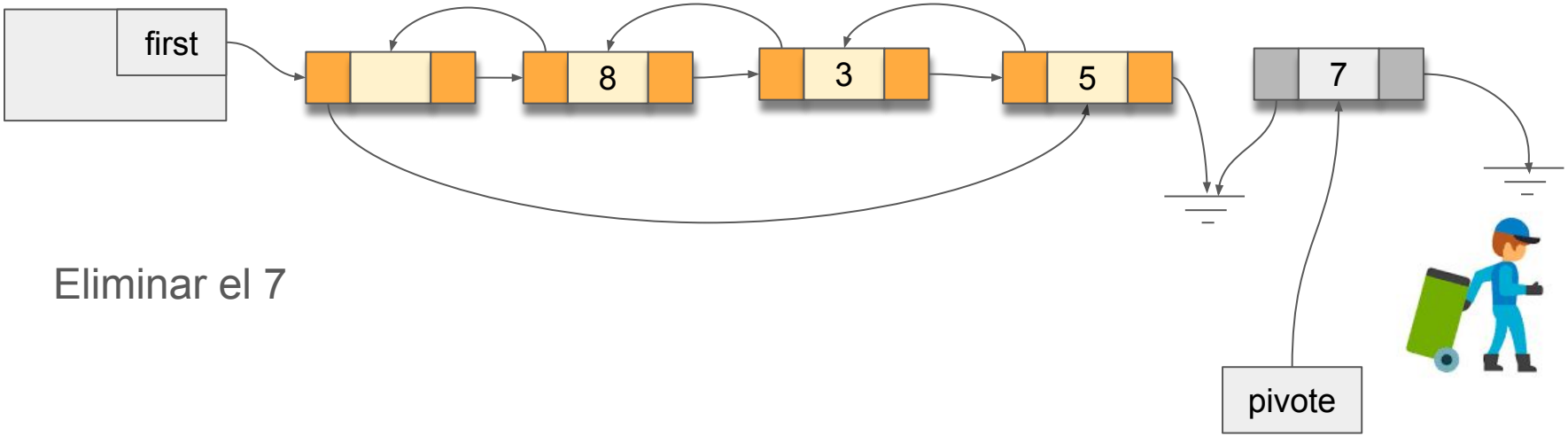
Eliminar el 7

Lista doble: Eliminar



Eliminar el 7

Lista doble: Eliminar



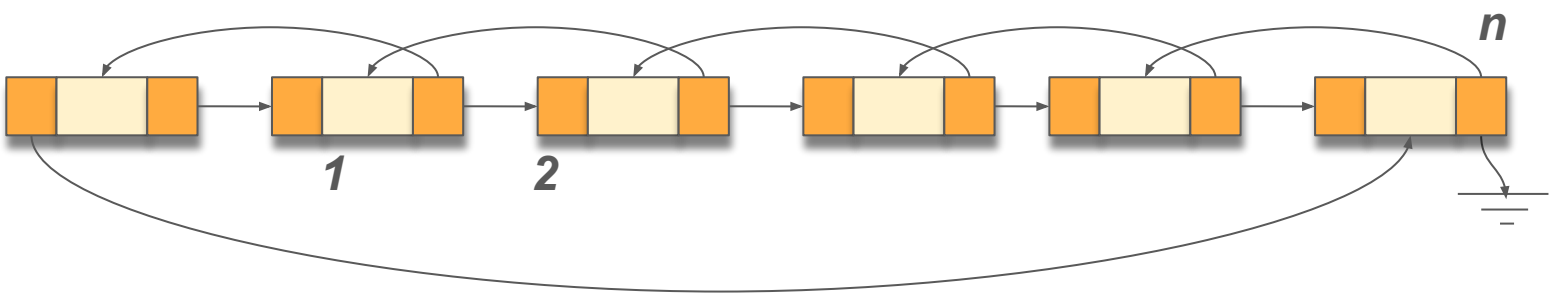
Eliminar el 7

Lista doble: Eliminar

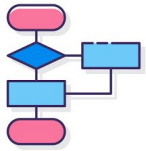
```
public void eliminar(int value) {
    NodoDoble pivote;

    pivote = this.existe(value);

    if (pivote != null) {
        NodoDoble prev = pivote.getPrev();
        NodoDoble next = pivote.getNext();

        this.primero.setPrev(pivote.getPrev());

        prev.setNext(next);
        if (next != null) {
            next.setPrev(prev);
        }

        pivote.setNext(null);
        pivote.setNext(null);
    }
}
```

Análisis de costos

- Determinar vacía: O(1)
- Agregar: O(1)
- Buscar: O(n)
- Eliminar: O(n)
- Listar: O(n)

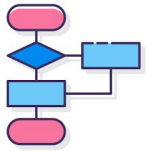


Práctica



1. Implemente una lista doblemente vinculada
    a. Método para determinar la lista vacía
    b. Método para agregar un elemento
    c. Método para eliminar la primer ocurrencia de un valor
    d. Método para buscar la primer ocurrencia de un valor
    e. Método para eliminar TODAS las ocurrencias de un valor
    f. Método para convertir a un solo String

Práctica

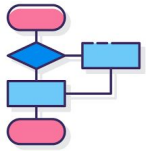


1. Implemente el TDA Pila, con una estructura dinámica
2. Implemente el TDA Cola, con una estructura dinámica
3. Implemente el TDA Conjunto, con una estructura dinámica

En ambos casos puede elegir entre Lista Simple o Doble

Probar agregados y eliminaciones, asi como el mostrar.

Conclusiones



Hemos visto dos formas de listas vinculadas, que permiten almacenamiento dinámico.

El análisis de costo, nos muestra que aunque se necesitan más operaciones en la lista doble, es menos costoso en términos computacionales.

Los TDA pueden implementarse con estas nuevas estructuras, ocultando el comportamiento interno.


consultas
dudas
dudas
dudas
consultas
consultas
dudas
dudas
consultas
dudas
dudas
dudas
dudas
consultas
dudas
consultas
dudas