Data Structures and Algorithms

The List Abstract Data Type - Doubly-Linked List

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

November 5, 2018

Learning outcomes

After this lecture and the related practical students should...

- be able to implement an doubly linked list
- understand the use of polymorphism to implement a data structure that can store any type of data

Table of Contents

The List Abstract Data Type

Concept

- The List ADT models a sequence of positions
- Each position store a piece of data
- There is a before/after relation between positions
- This allows for efficient insertion into the middle of a list

The Position Abstract Data Type

Concept

Before we can fully understand the list abstract data type we need to look at the position abstract data type

- The Position abstract data type models the idea of a place within a data structure where a single piece of data is stored
- Positions provide a simple view of different ways of storing data
  - An element in an array
  - A Node in a linked list

The Position Abstract Data Type

Specification

- Operation:
  - element(): This returns the piece of data that is stored in this position

The Position Abstract Data Type

Interface

```
1 public interface Position{
2   public Object element();
3 }
```

The List Abstract Data Type

Specification

Operations:

- first(): returns the first position in the list
- last(): return the last position in the list
- before(p): returns the position in the list before p
- after(p) : returns the position in the list after p
- insertBefore(p, d): inserts the value d into the position in the list before p
- insertAfter(p, d): inserts the value d into the position in the list after p
- insertFirst(d): inserts the value d into the first position in the list
- insertLast(d): inserts the value d into the last position in the list
- remove(p): removes the position p from the list
- size(): returns the number of elements stored in the list
- isEmpty(): is the list empty?

The List Abstract Data Type

Interface

```
1  public interface List {
2    public Position first();
3    public Position last();
4    public Position before(Position p);
5    public Position after(Position p);
6    public Position insertBefore(Position p,
      Object d);
7    public Position insertAfter(Position p,
      Object d);
8    public Position insertFirst(Object d);
9    public Position insertLast(Object d);
10   public Object remove(Position p);
11   public int size();
12   public boolean isEmpty();
13 }
```

The List Abstract Data Type

Implementation Strategies

- Array based implementation
  - An array of `Position` objects
  - Search through the array to find the correct position
  - Not very efficient
- Link based implementations
- There are two versions
  - Singly-Linked List
    - ⋆ Each `Position` object keeps a reference to the next `Position` in the sequence
  - Doubly-Linked List
    - ⋆ Each `Position` object keeps a reference to the next and previous `Positions` in the sequence

Table of Contents

Position Abstract Data Type

Doubly-Linked Implementation

- We create a `Node` class that implements the `Position` interface
- We add functionality to the class to store the next `Node` in the sequence
- We also add functionality to the class to store the previous `Node` in the sequence

Position Abstract Data Type

Doubly-Linked Implementation

```
1  public class Node implements Position {
2    private Object element;
3    Node next;
4    Node previous;
5
6    public Node(int e) {
7      this.element = e;
8    }
9
10   public Object element() {
11     return element;
12   }
13 }
```

Doubly-Linked List Implementation

- We keep a reference to the first position in the list
- We keep a reference to the last position in the list
- We update the references when necessary
- We keep count of the number of positions in the list

Variables:

- A reference to the first position in the list

```
private Node first;
```

- A reference to the last position in the list

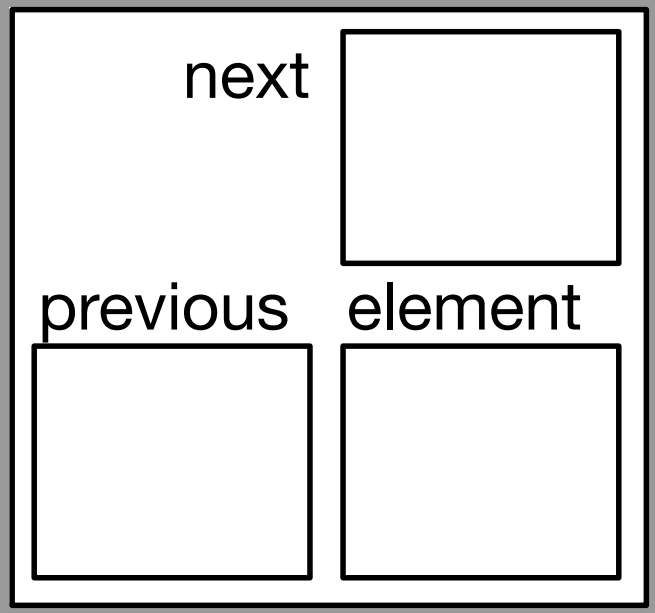
```
private Node last;
```

- A number to keep track of the size

```
private int size;
```

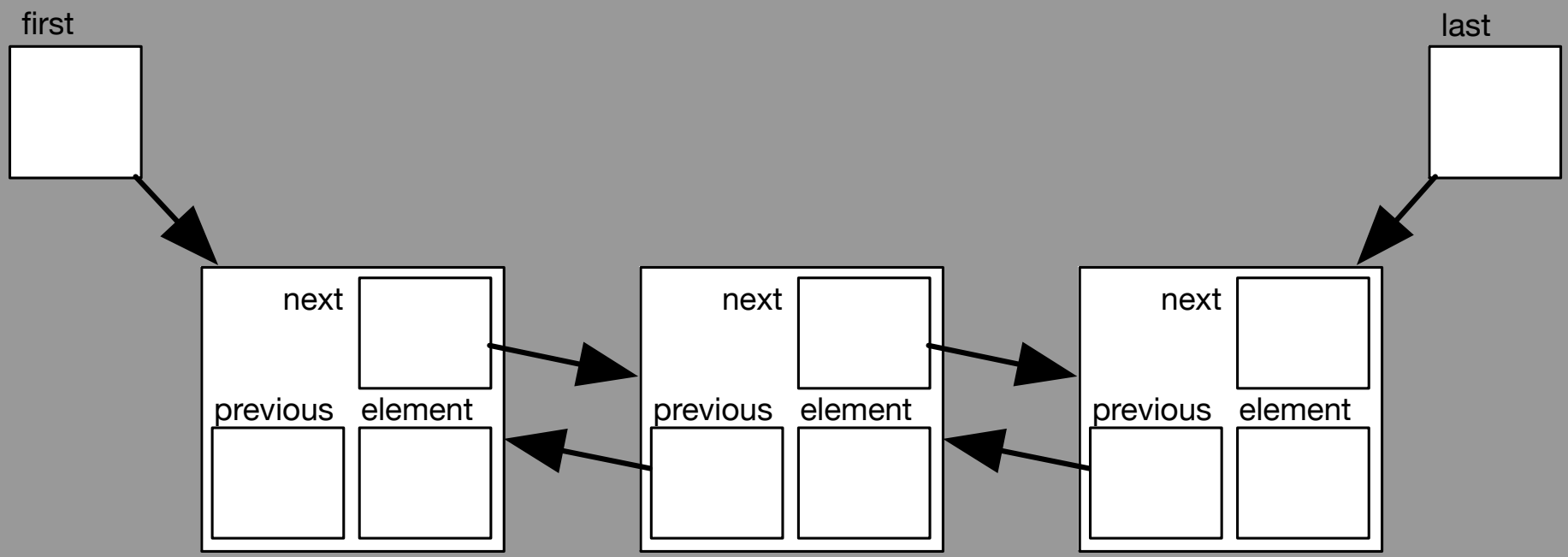
Representation of a Node Object

Doubly-Linked List Implementation



Representation of a List

Doubly-Linked List Implementation



Doubly-Linked List Operations

- first()
  - Return the reference that is stored in the variable `first`
- last()
  - Return the reference that is stored in the variable `last`
- size()
  - Return the value of the `size` variable
- isEmpty()
  - Return the result of the expression `size == 0`
- after(p)
  - Convert p to a node
  - Return the `next` reference of p
- before(p)
  - Convert p to a node
  - Return the `previous` reference of p

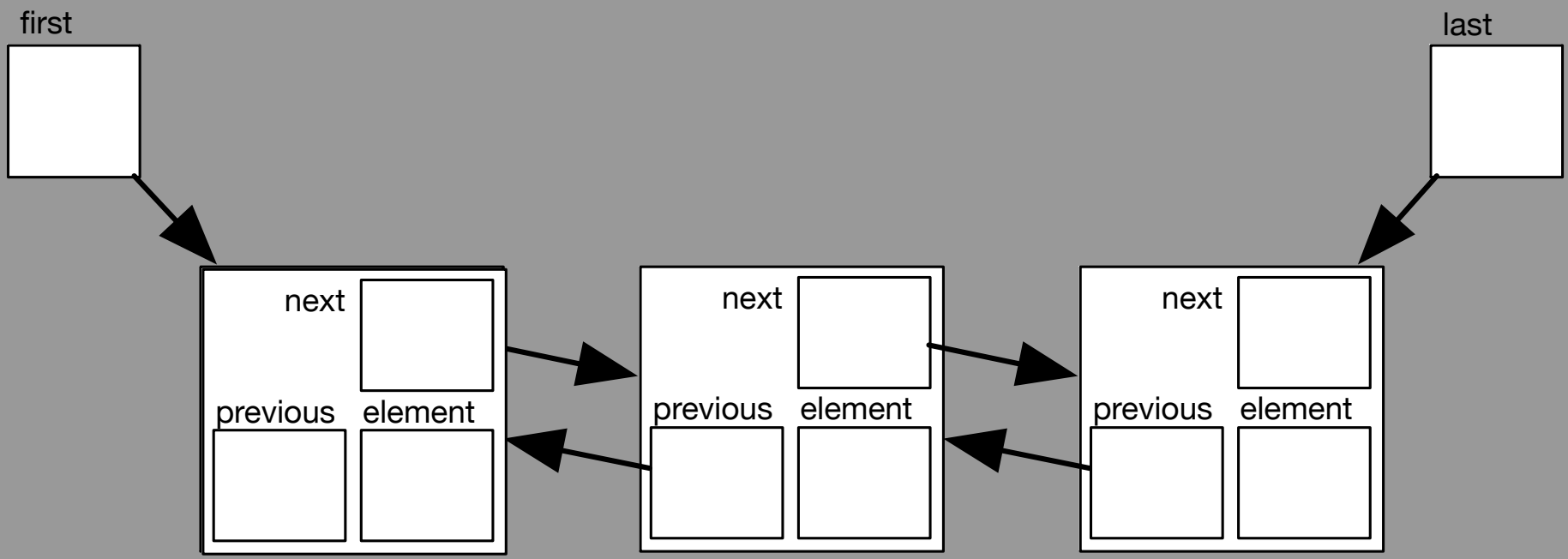
Doubly-Linked List Operations

insertFirst(d)

- Insert the value into the first position
  - Construct a new `Node` object, called `n`, containing the value
  - Change the `next` reference in `n` so that it points to `first`
  - Change the `previous` reverence in `first` so that it points to `n`
  - Change the `first` reference so that it points to `n`
  - Increment the size
- What happens if the list is empty?

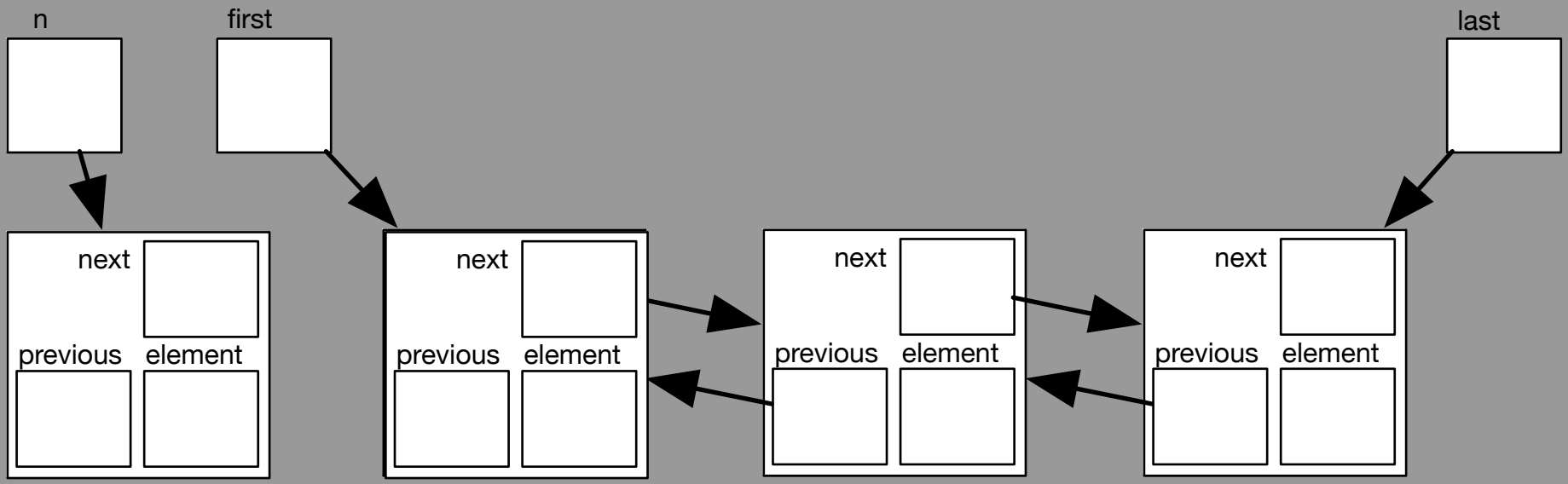
Doubly-Linked List Operations

insertFirst(d)



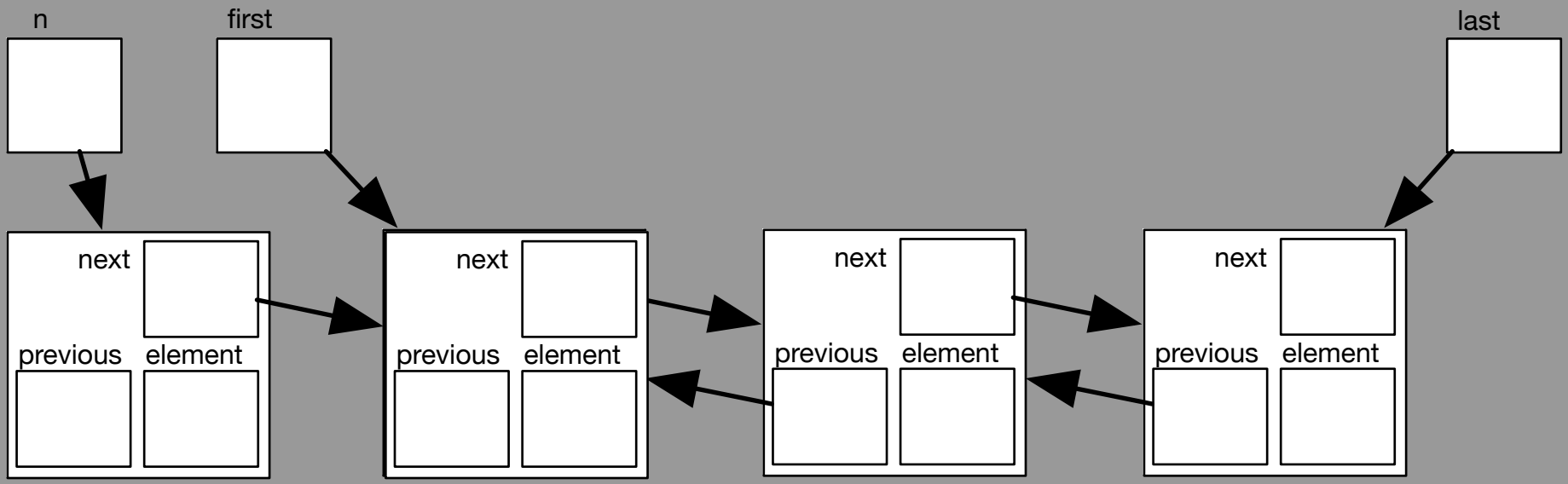
Doubly-Linked List Operations

insertFirst(d)



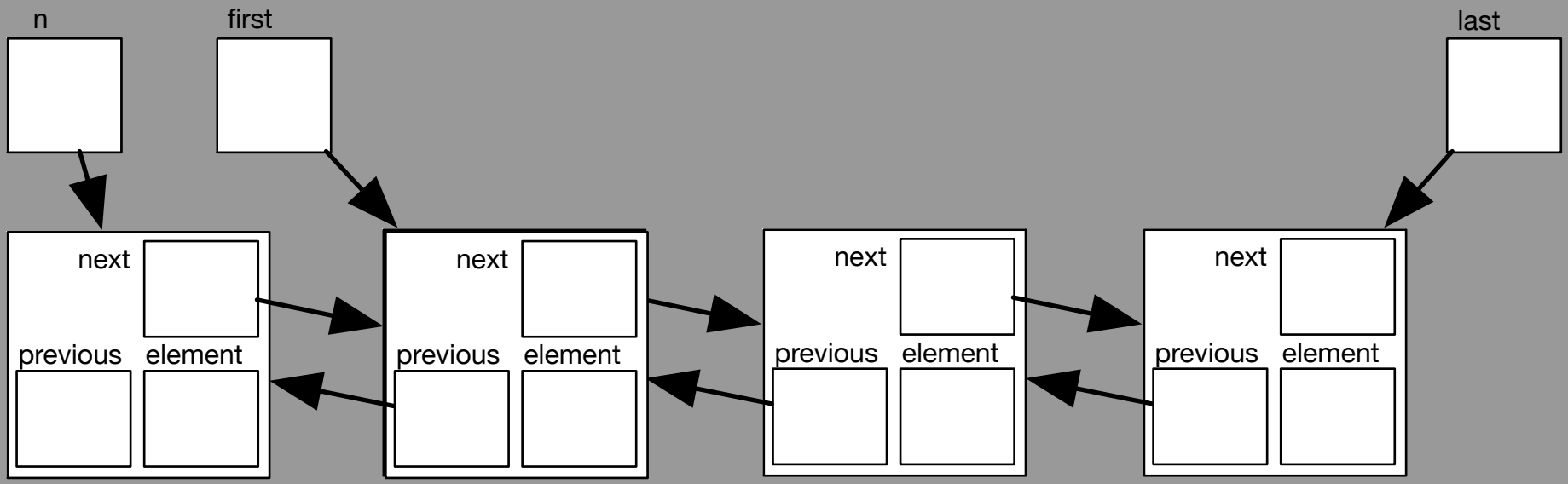
Doubly-Linked List Operations

insertFirst(d)



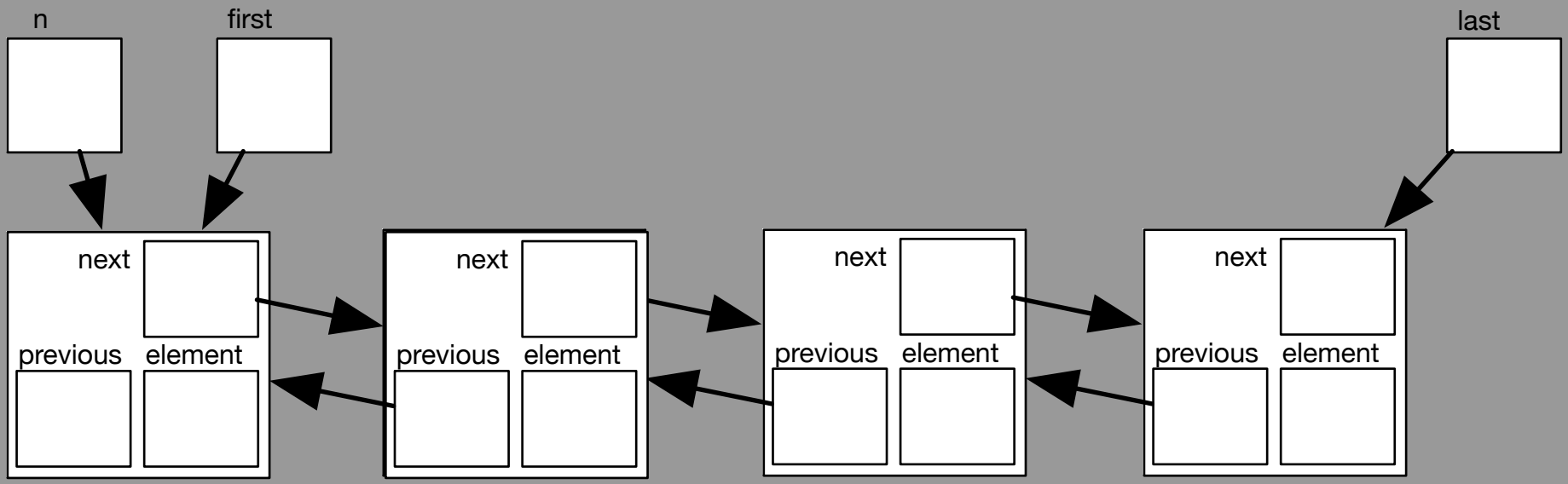
Doubly-Linked List Operations

insertFirst(d)



Doubly-Linked List Operations

insertFirst(d)



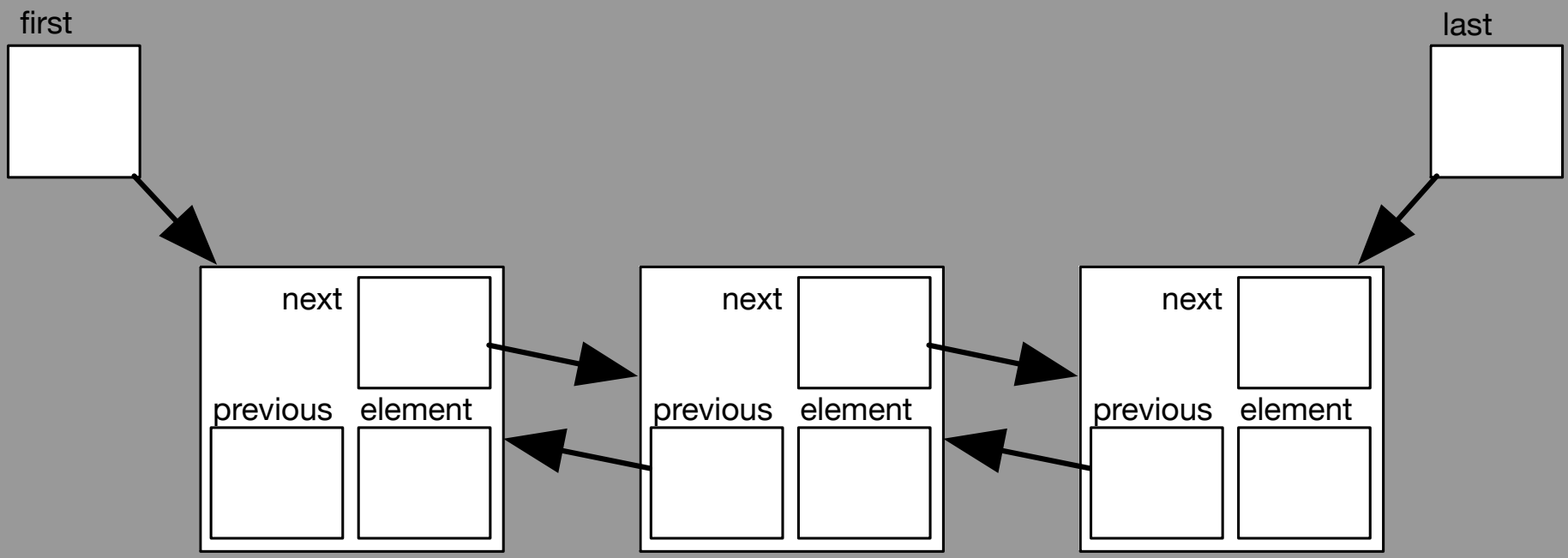
Doubly-Linked List Operations

insertLast(d)

- Insert the value into the last position
  - Construct a new `Node` object, called `n`, containing the value
  - Change the `previous` reference of `n` so it points to `last`
  - Change the the `next` reference of `last` so that it points to `n`
  - Change the `last` reference so that it points to `n`
  - Increment the size
- What happens if the list is empty?

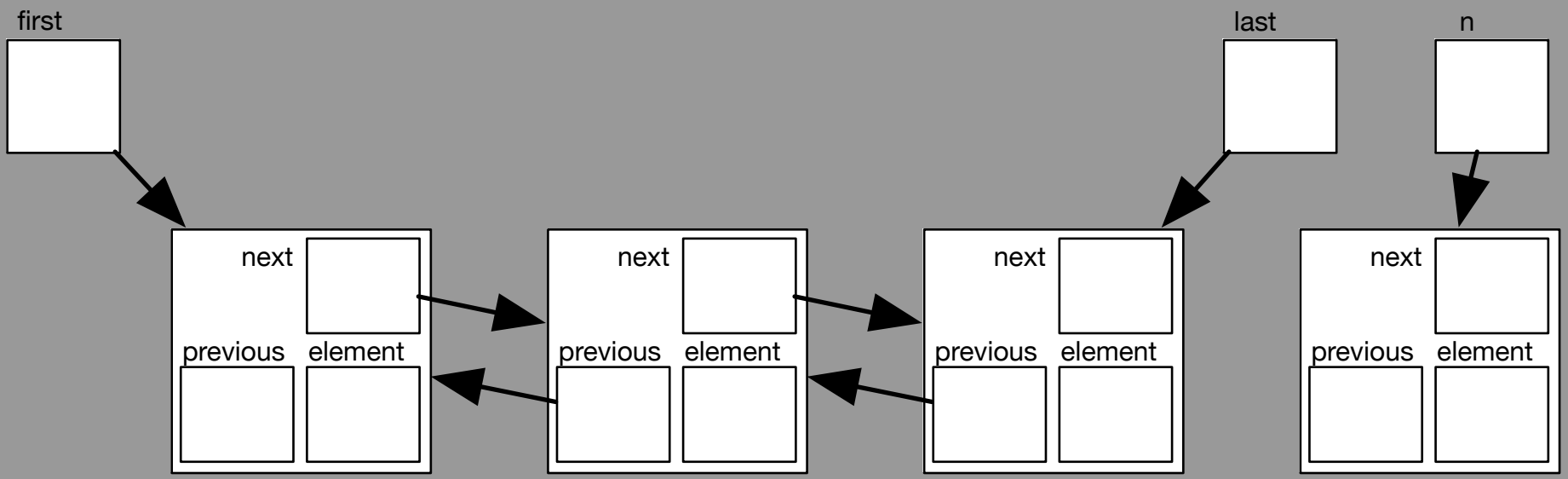
Doubly-Linked List Operations

insertLast(d)



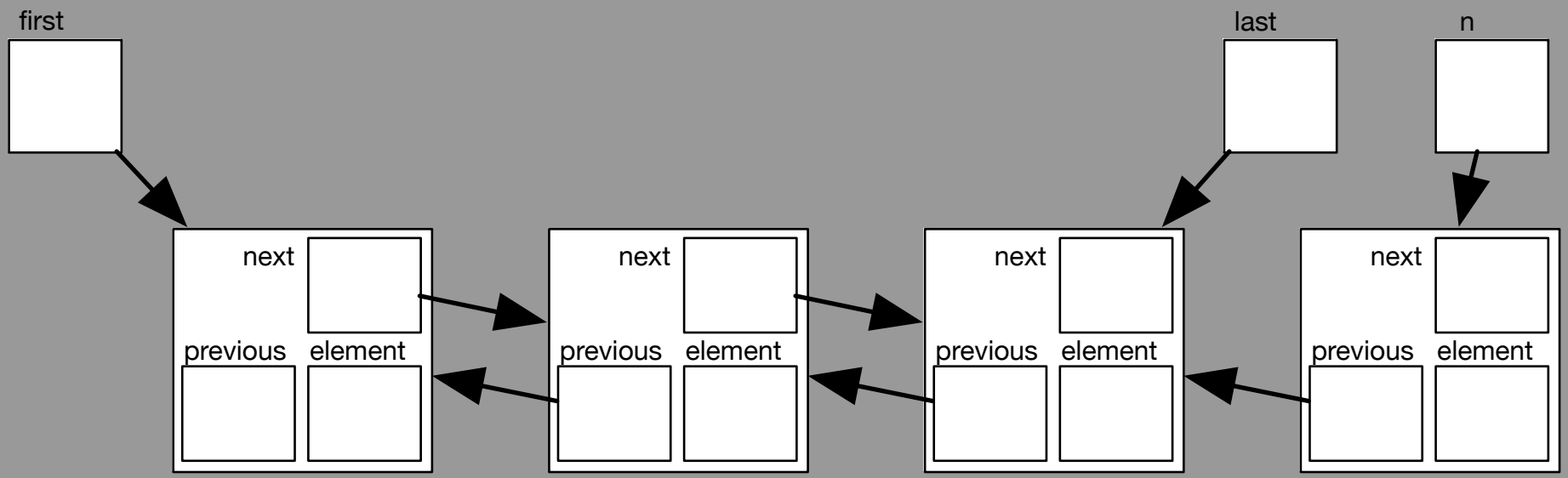
Doubly-Linked List Operations

insertLast(d)



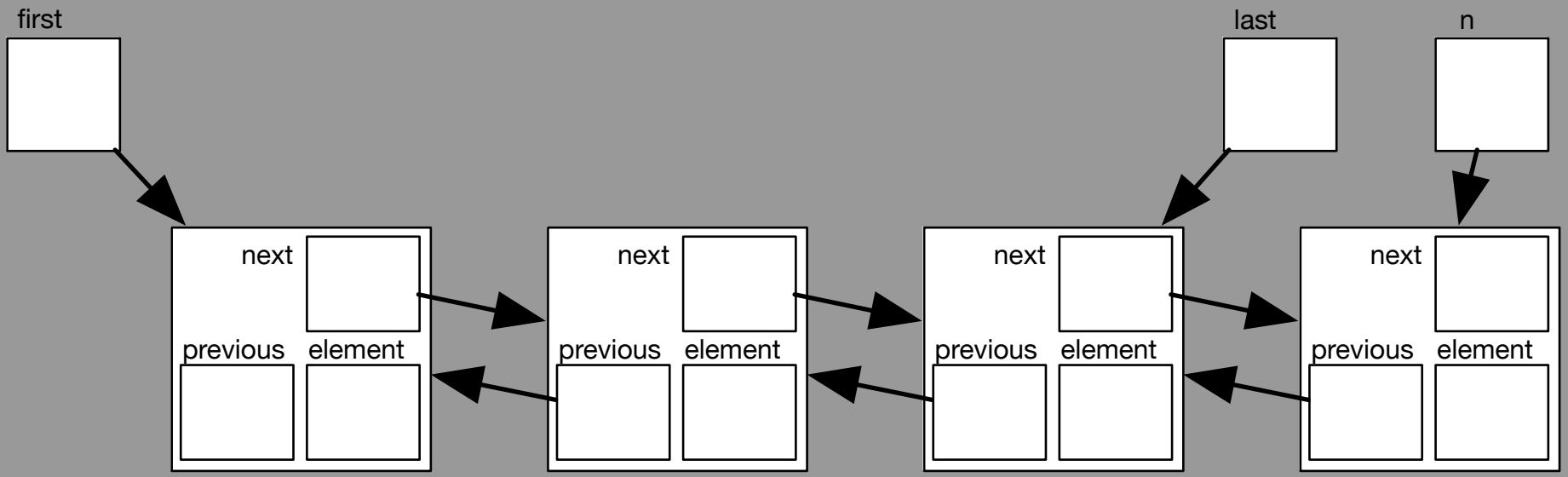
Doubly-Linked List Operations

insertLast(d)



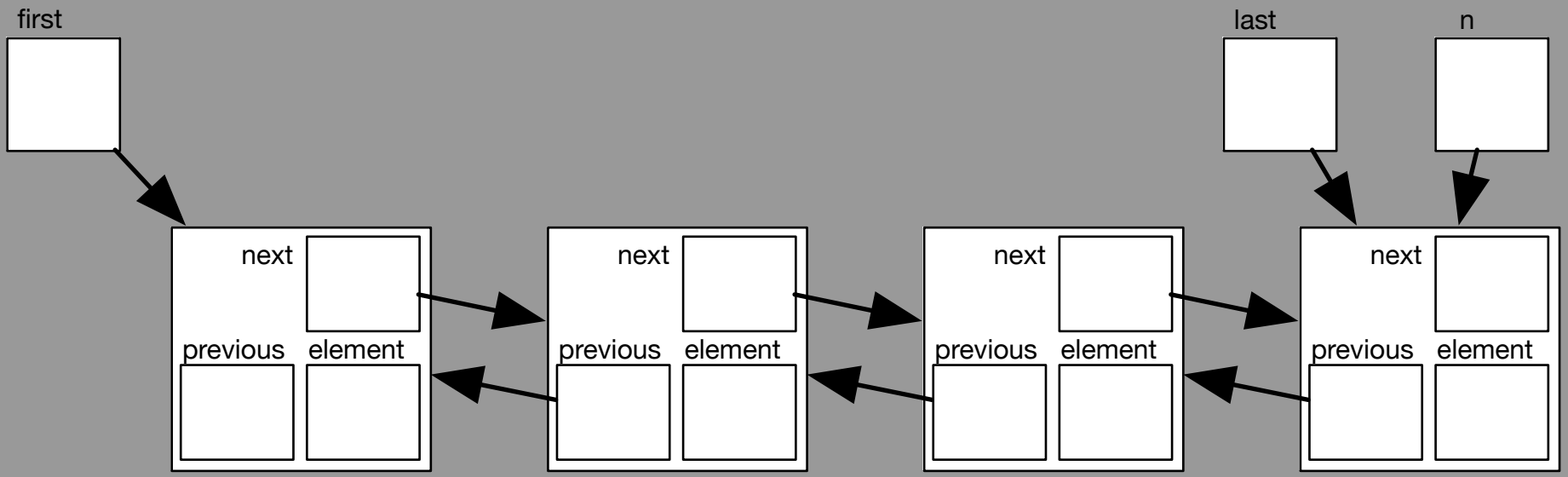
Doubly-Linked List Operations

insertLast(d)



Doubly-Linked List Operations

insertLast(d)



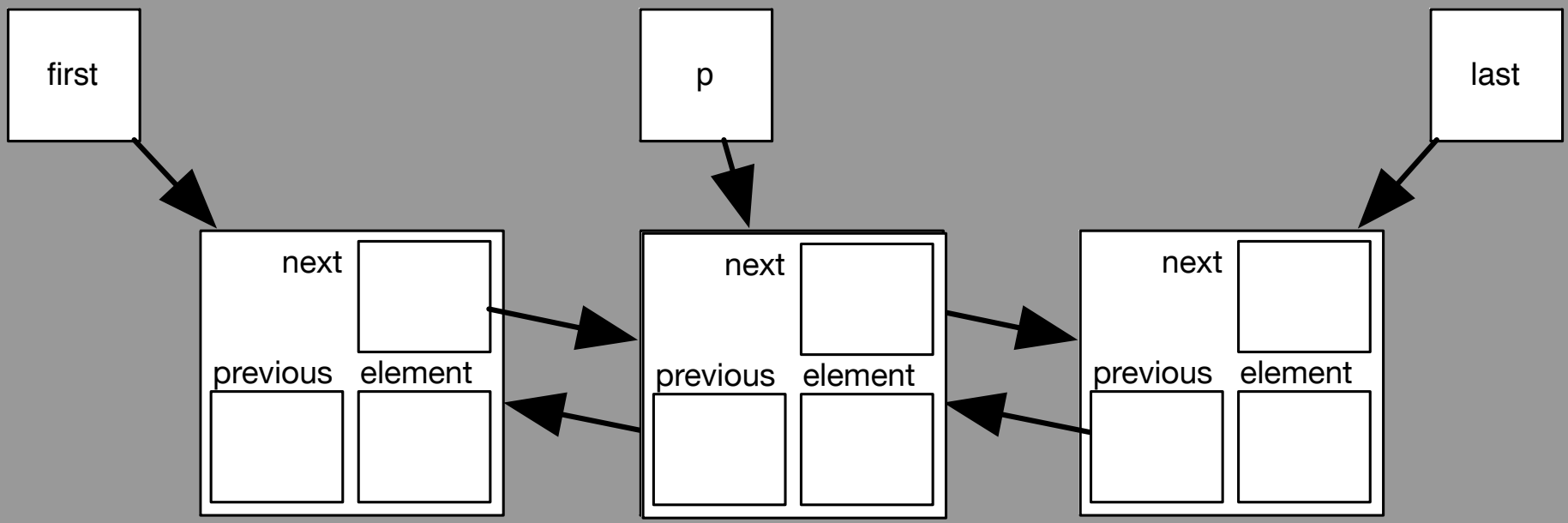
Doubly-Linked List Operations

insertAfter(p, d)

- Insert the value into the position after `p`
  - Construct a new `Node` object, called `n`, containing the value
  - Convert `p` to a `Node`
  - Change `next` reference in `n` so it points to the `next` of `p`
  - Change `previous` reference in `n` so it points to `p`
  - Change `previous` reference in the `next` of `p` so it points to `n`
  - Change the `next` of `p` so it points to `n`
  - Increment the size
- What happens if `p` is in the last position?

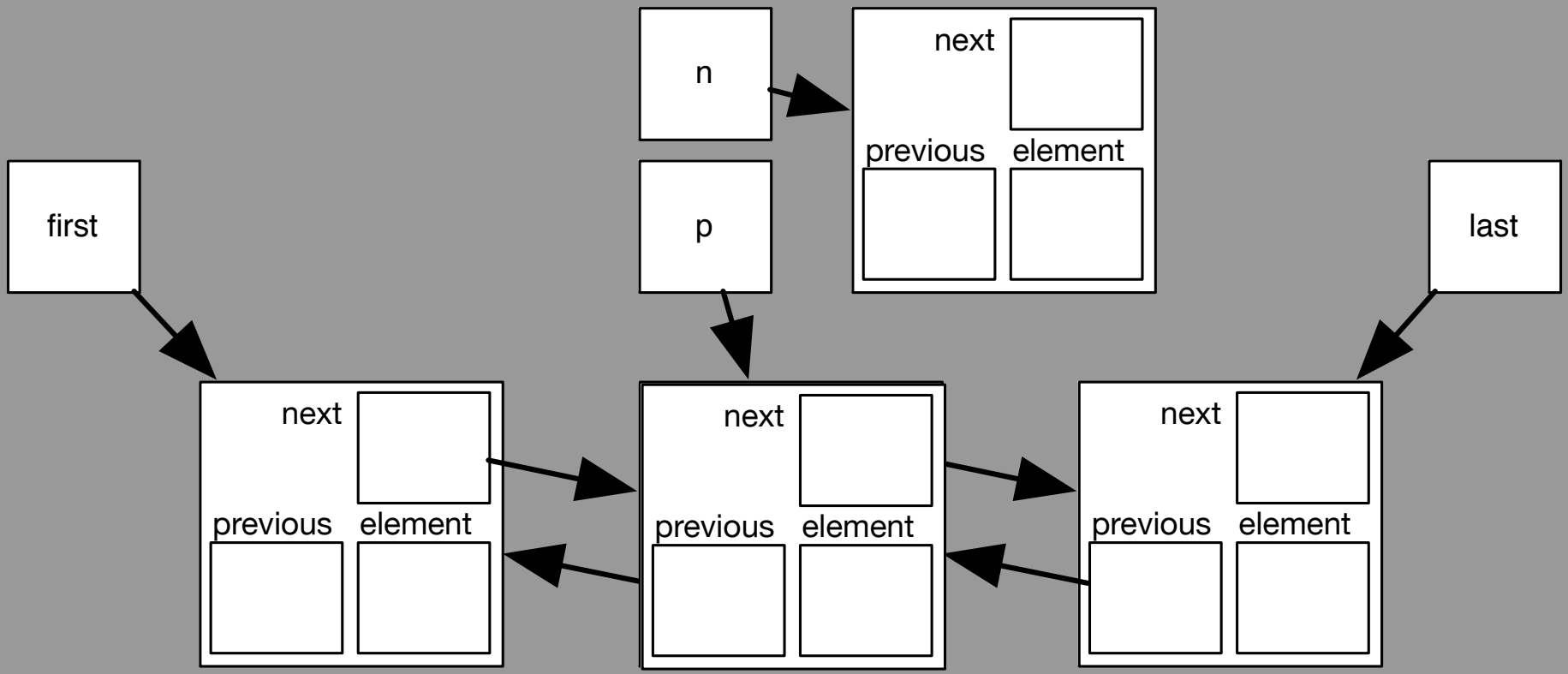
Doubly-Linked List Operations

insertAfter(p, d)



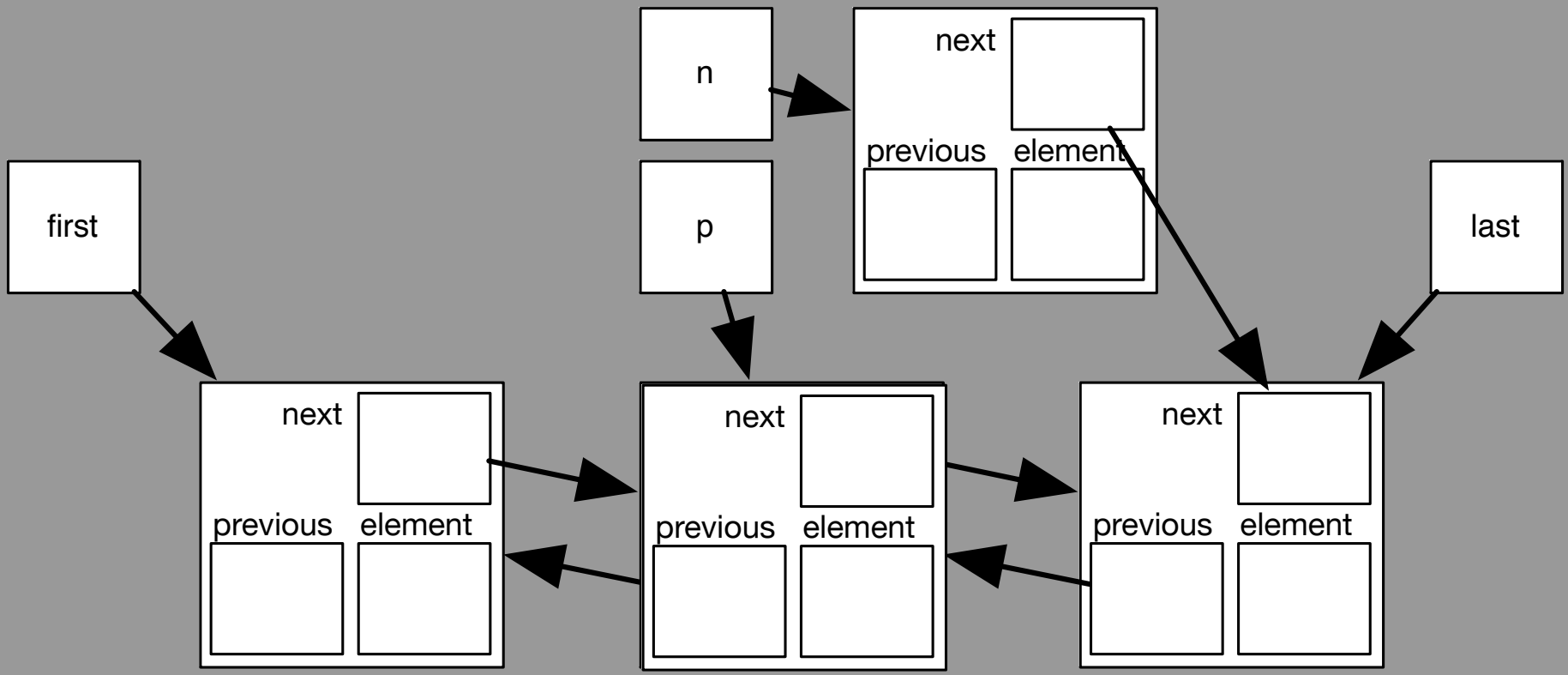
Doubly-Linked List Operations

insertAfter(p, d)



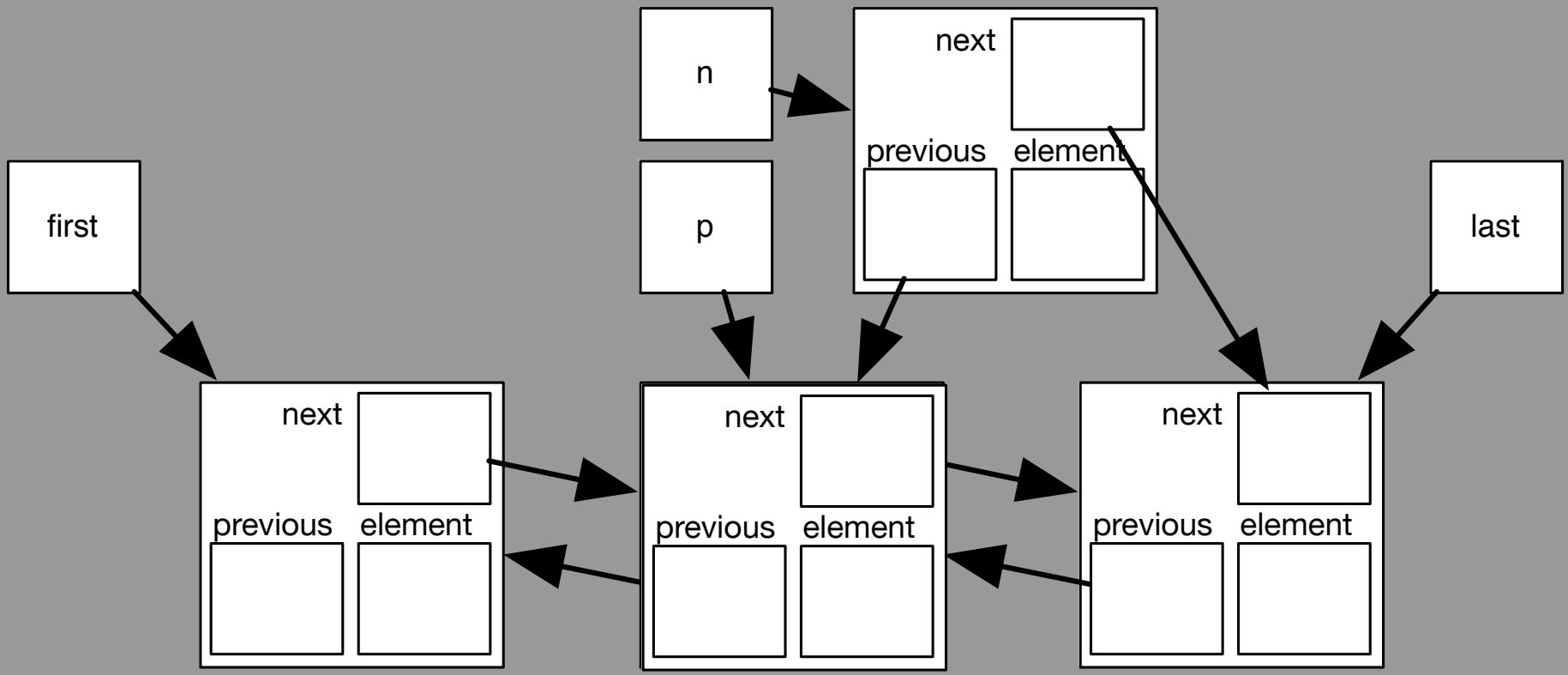
Doubly-Linked List Operations

insertAfter(p, d)



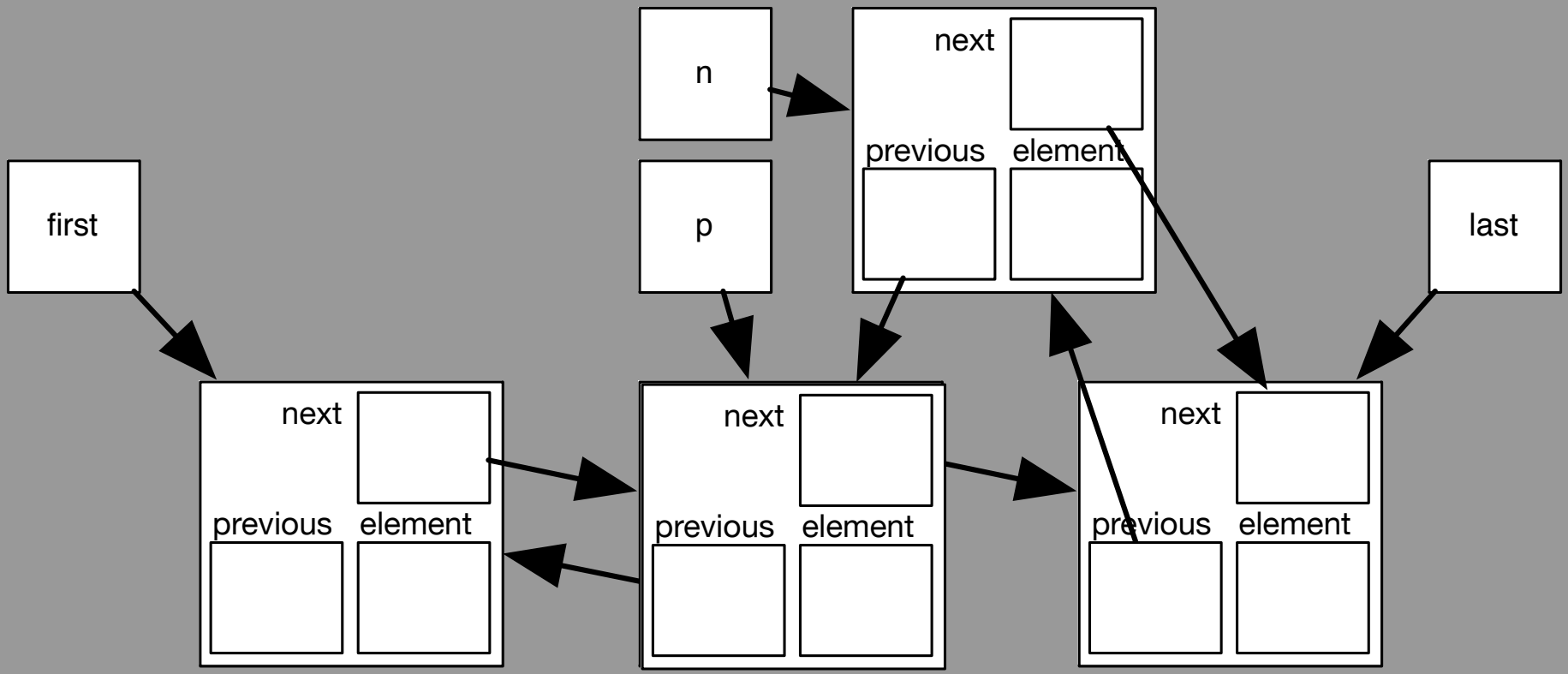
Doubly-Linked List Operations

insertAfter(p, d)



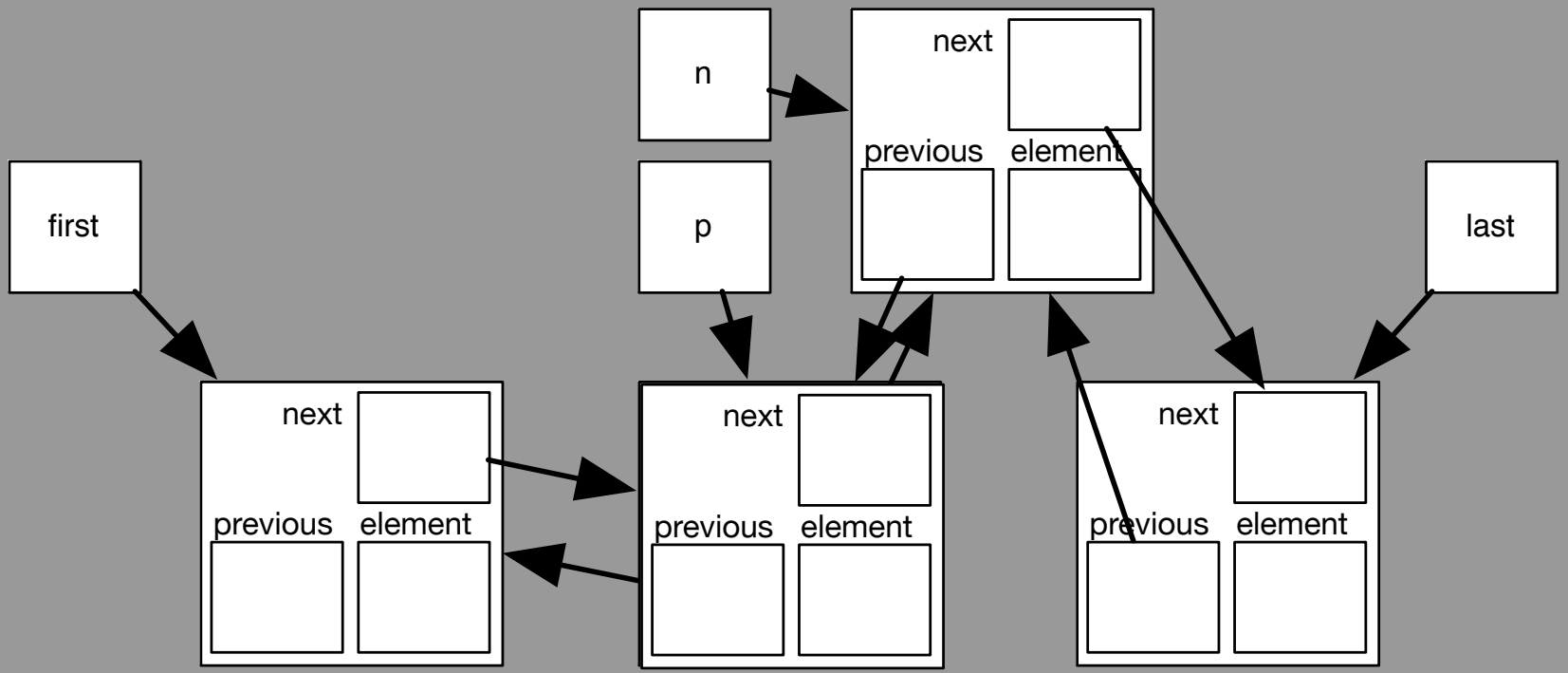
Doubly-Linked List Operations

insertAfter(p, d)



Doubly-Linked List Operations

insertAfter(p, d)



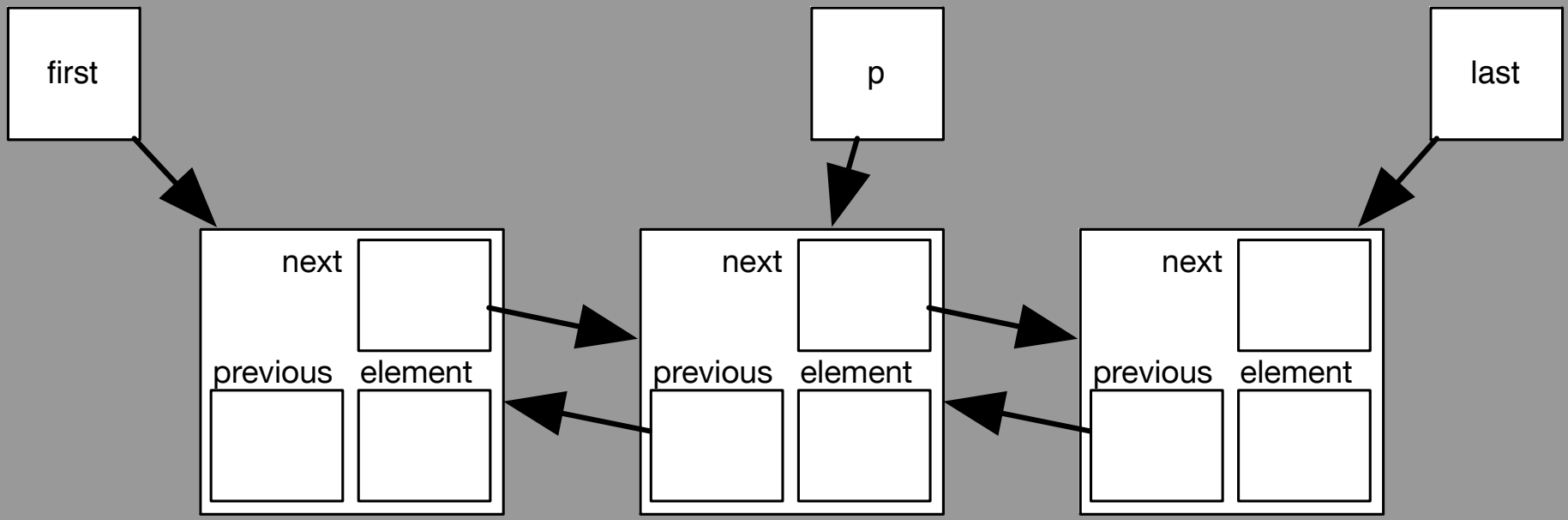
Doubly-Linked List Operations

insertBefore(p, d)

- Insert the value into the position before `p`
  - Construct a new `Node` object, called `n`, containing the value
  - Convert `p` to a `Node`
  - Change `previous` reference in `n` so it points to the `previous` of `p`
  - Change `next` reference in `n` so it points to `p`
  - Change `next` reference in the `previous` of `p` so it points to `n`
  - Change the `previous` of `p` so it points to `n`
  - Increment the size
- What happens if `p` is the fist position?

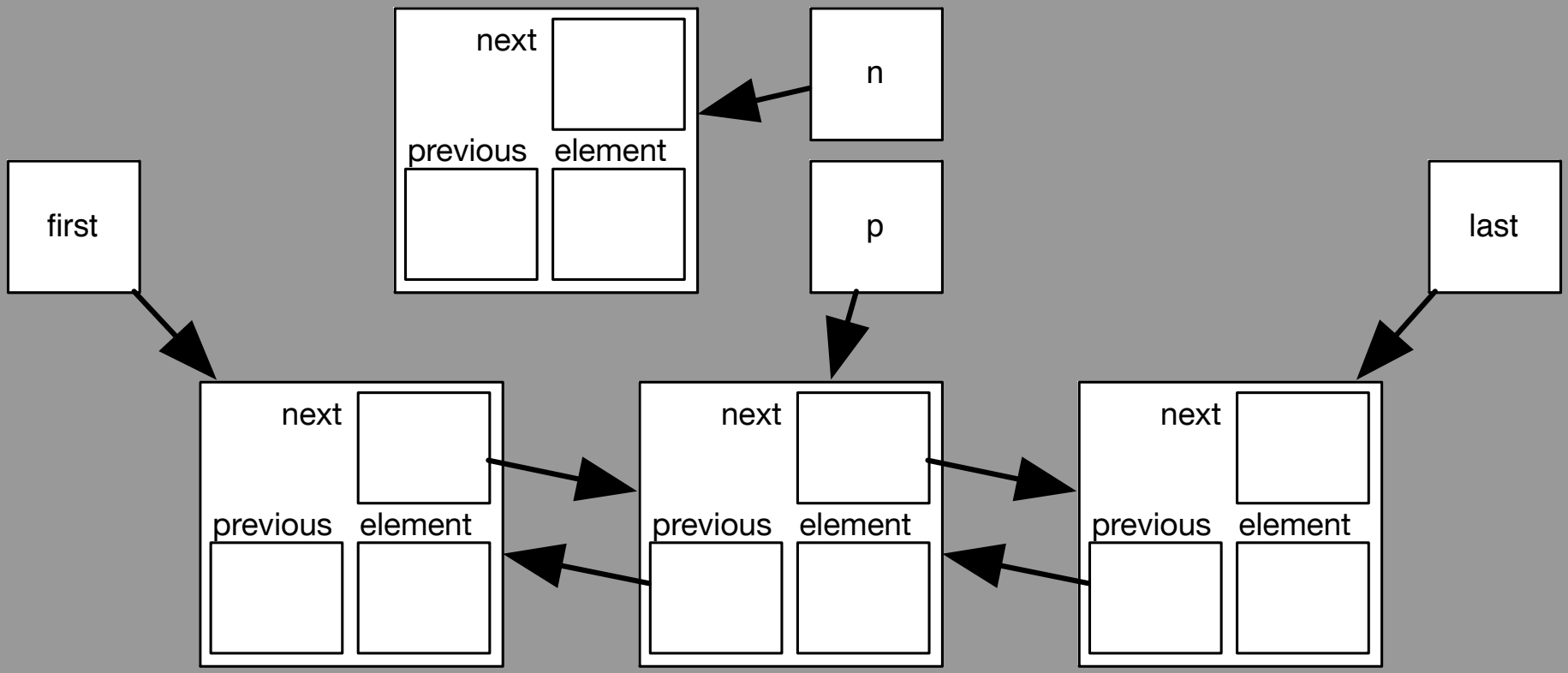
Doubly-Linked List Operations

insertBefore(p, d)



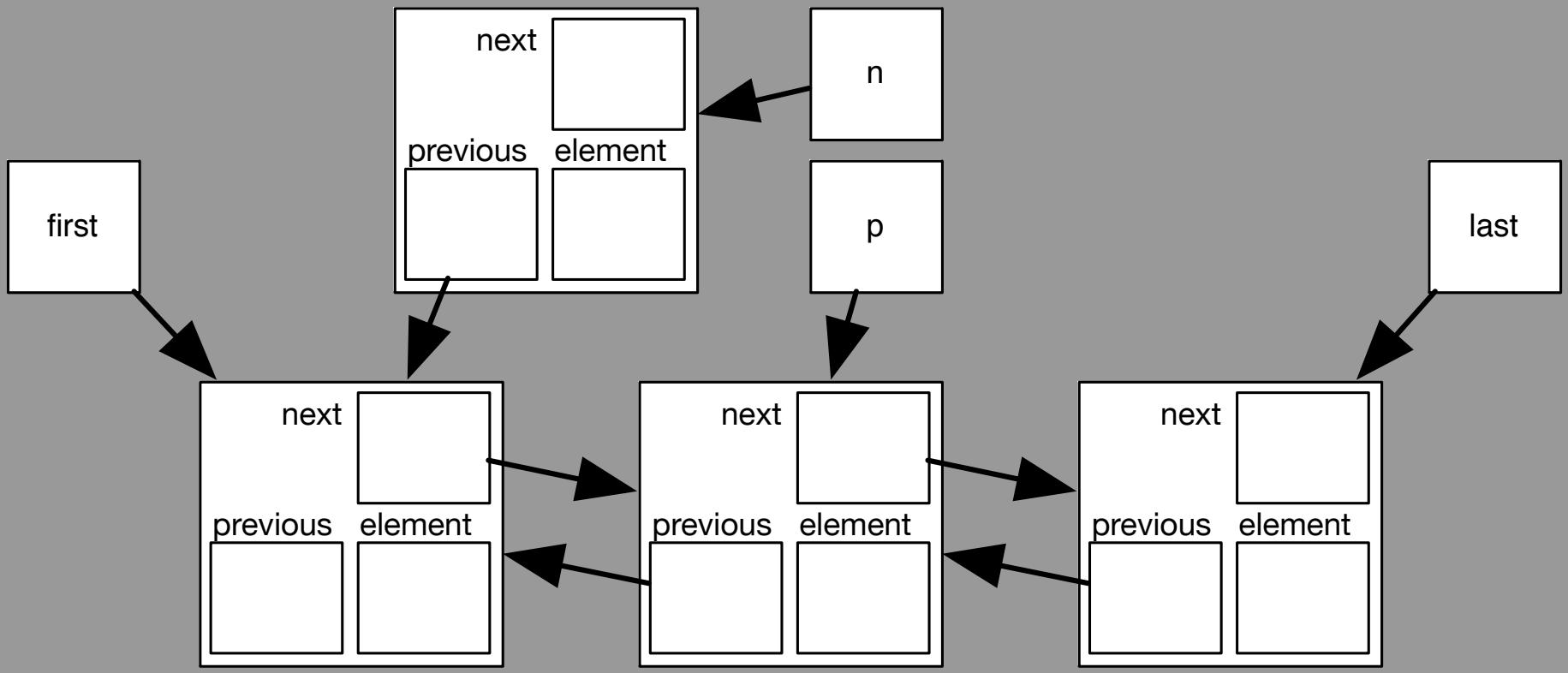
Doubly-Linked List Operations

insertBefore(p, d)



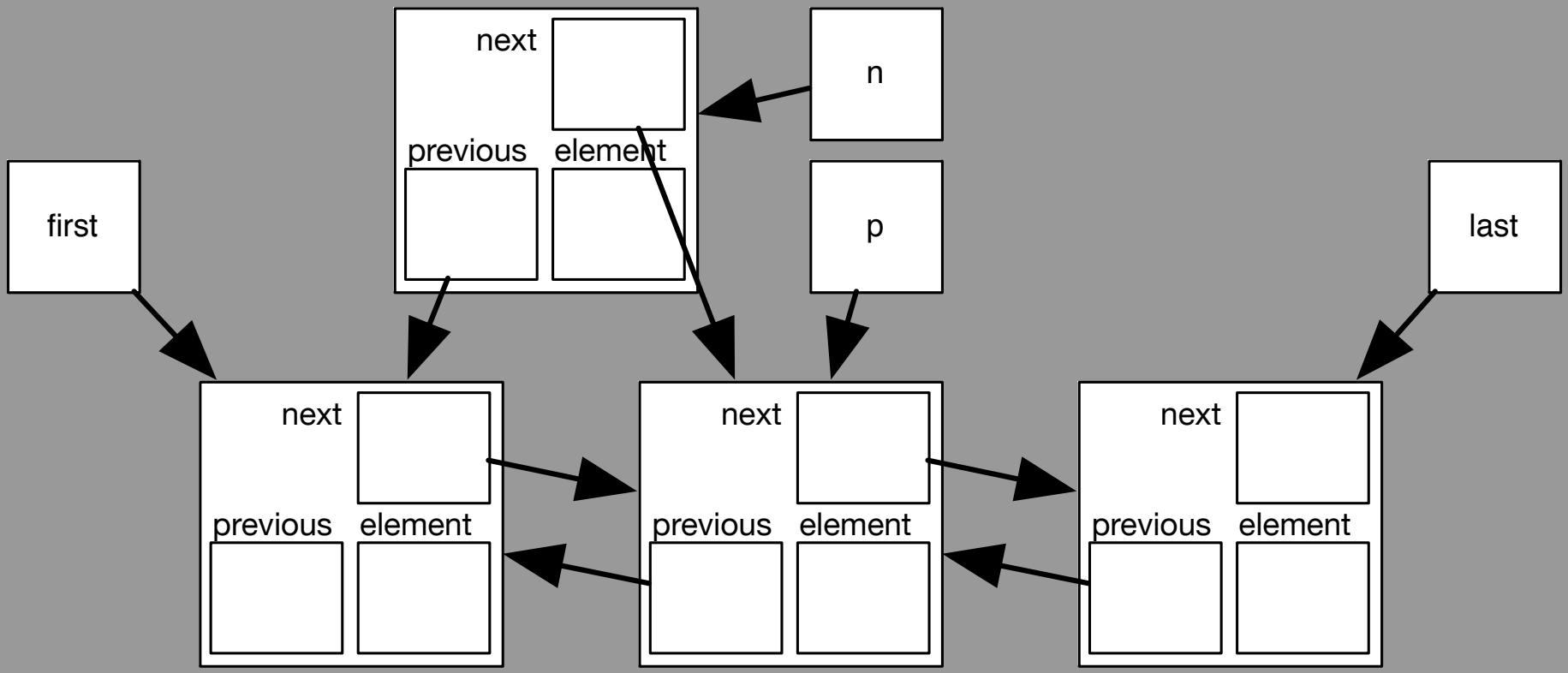
Doubly-Linked List Operations

insertBefore(p, d)



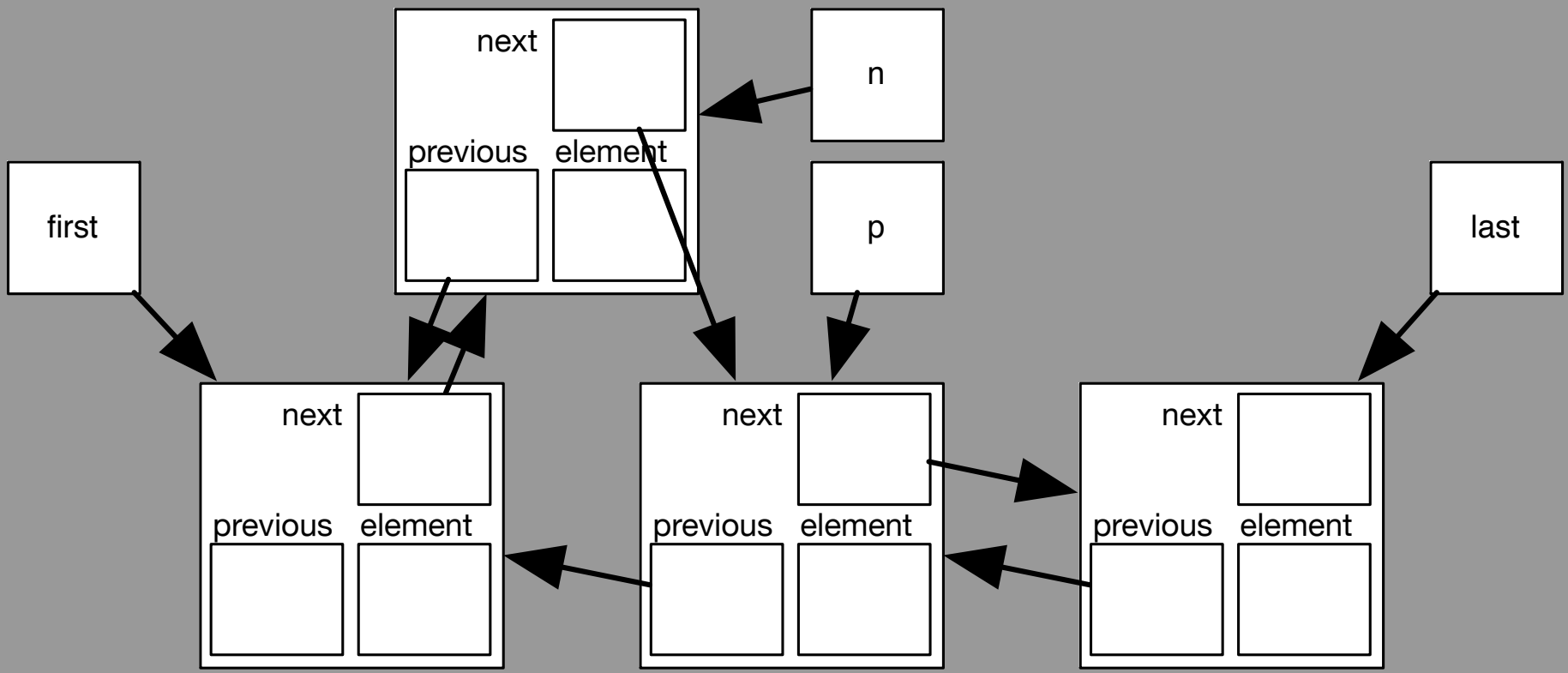
Doubly-Linked List Operations

insertBefore(p, d)



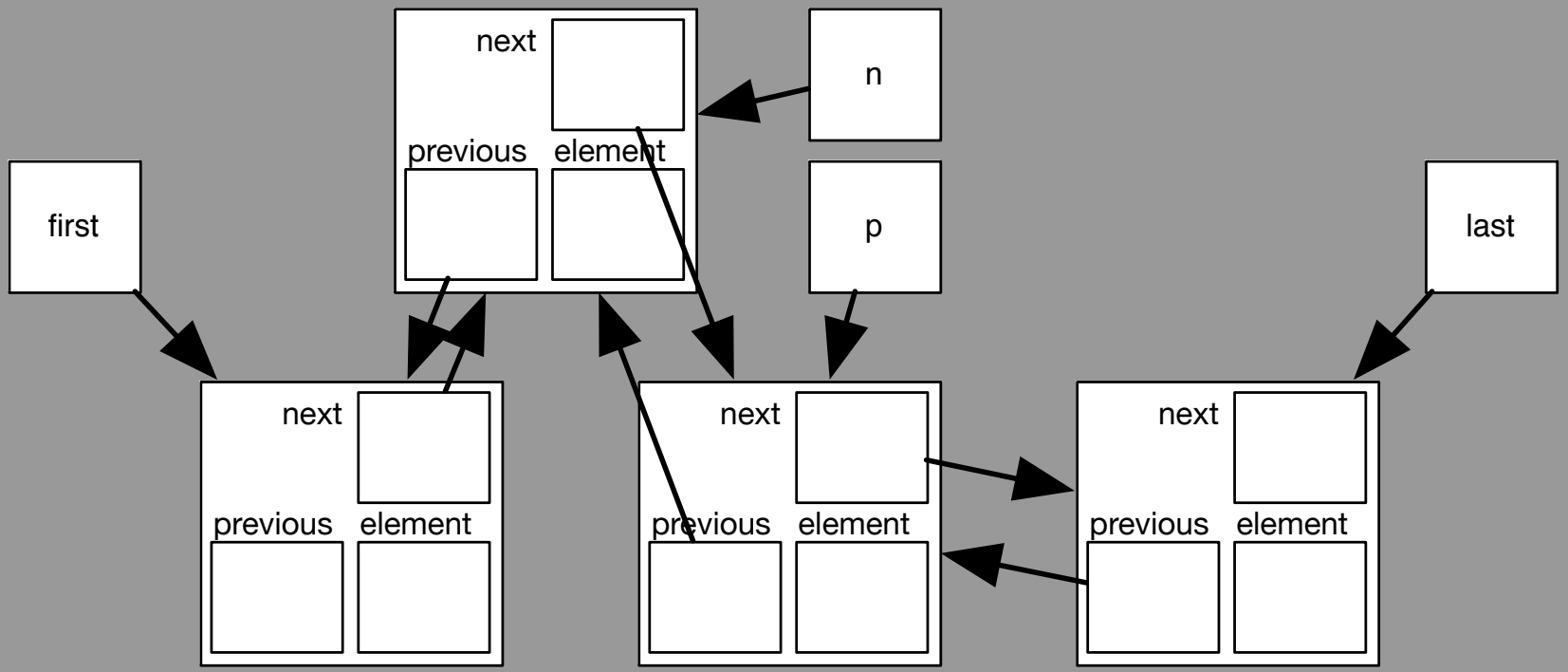
Doubly-Linked List Operations

insertBefore(p, d)



Doubly-Linked List Operations

insertBefore(p, d)



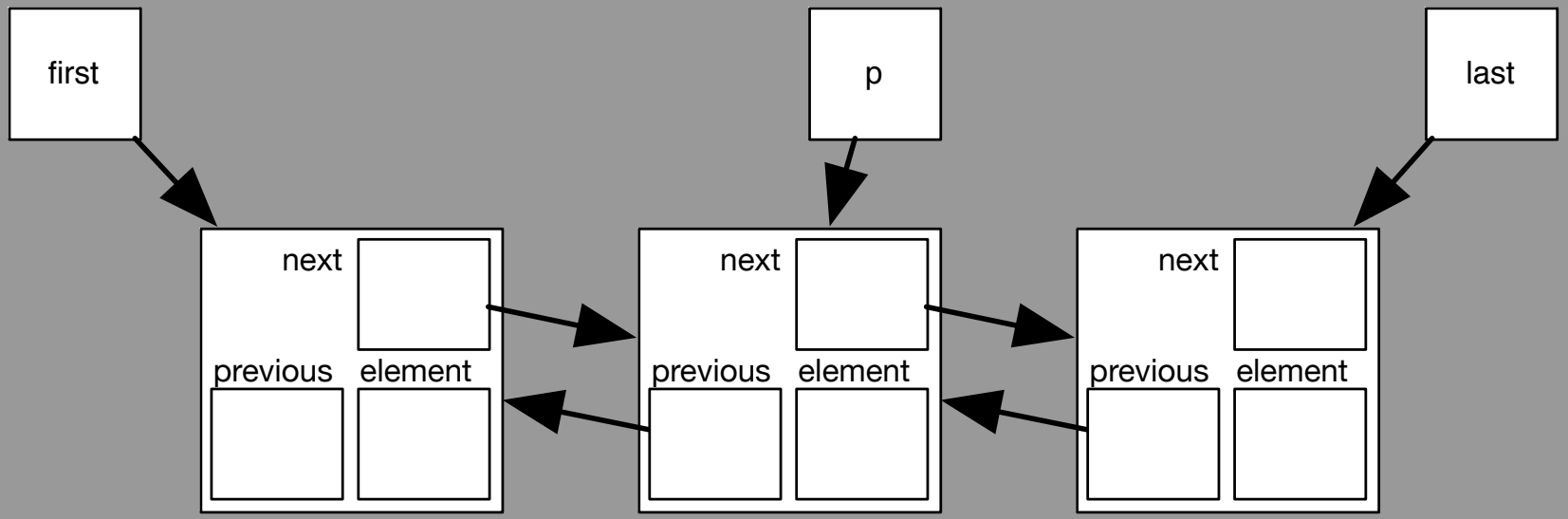
Doubly-Linked List Operations

remove(p)

- Remove the object p from the list
  - Copy the data from inside the position `p` to variable `d`
  - Convert `p` to a `Node`
  - Change `next` of `previous` of `p` so that it points to the `next` of `p`
  - Change `previous` of `next` of `p` so that it points to the `previous` of `p`
  - Decrement the size
  - return `d`
- What happens if p is the fist position?
- What happens if p is the last position?
- What happens if p is the only element?

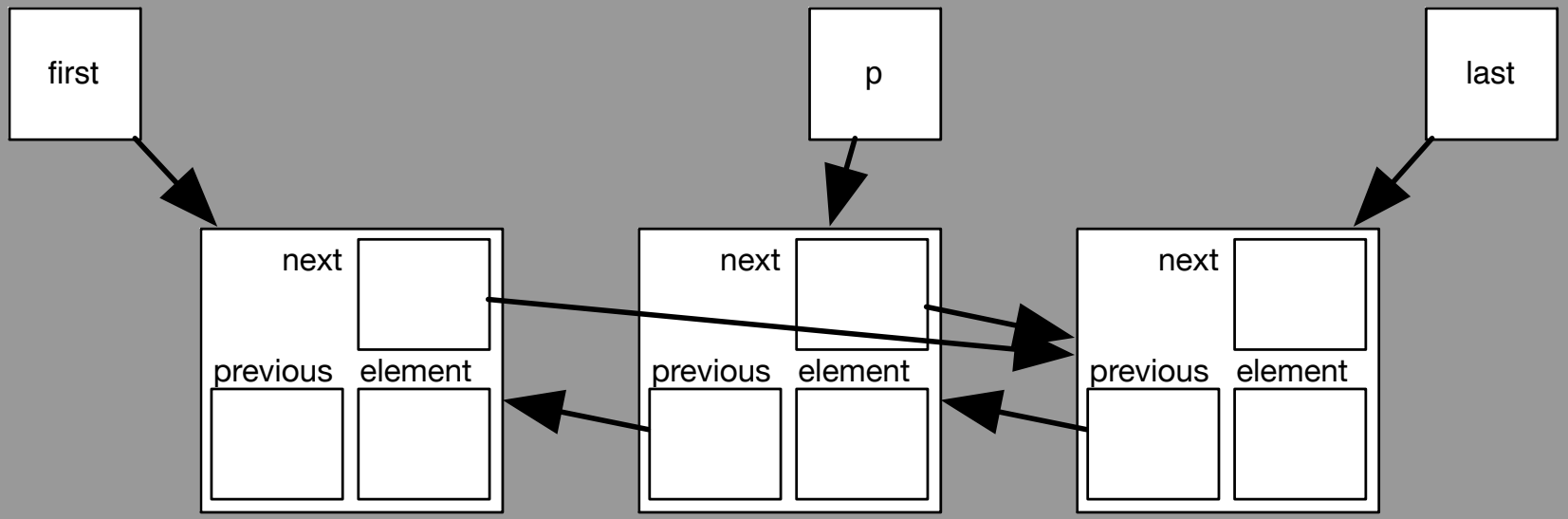
Doubly-Linked List Operations

remove(p)



Doubly-Linked List Operations

remove(p)



Doubly-Linked List Operations

remove(p)

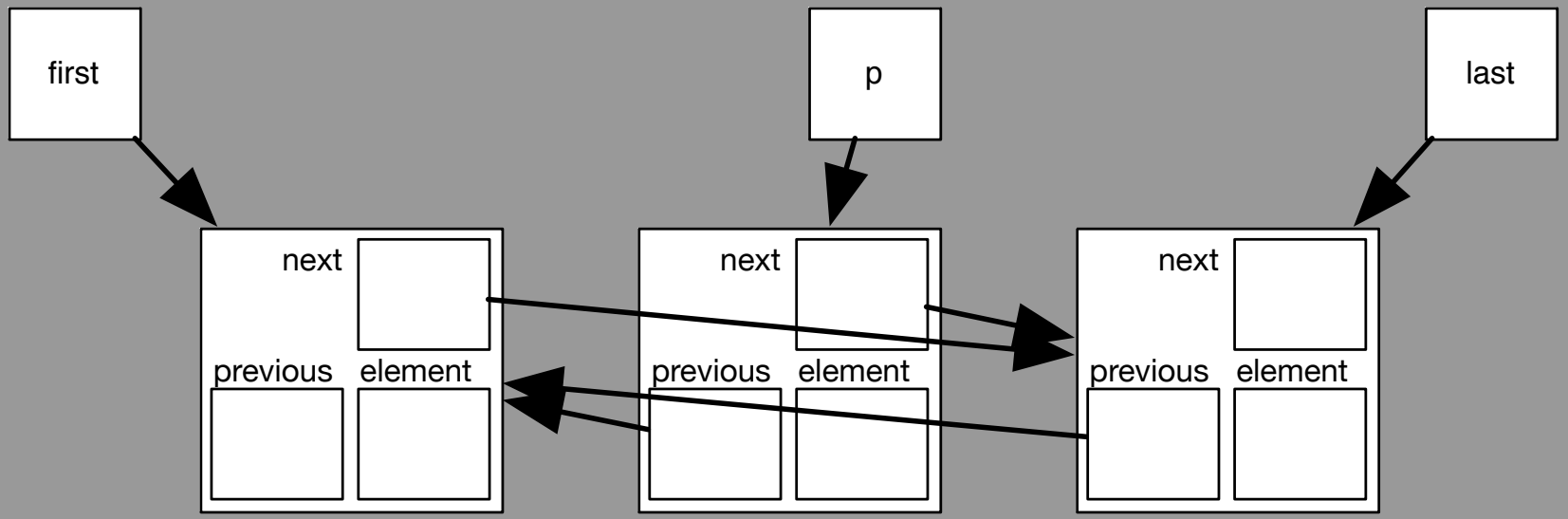


Table of Contents

Algorithmic Complexity

- first()
  - O(1)
- size()
  - O(1)
- isEmpty()
  - O(1)
- after(p)
  - O(1)
- last()
  - O(1)

Table of Contents

Comparing Complexity of Different Implementations

Operation	Array-Based	Singly-Linked	Doubly-Linked
first()	O(1)	O(1)	O(1)
last()	O(1)	O(n)	O(1)
size()	O(1)	O(1)	O(1)
isEmpty()	O(1)	O(1)	O(1)
after()	O(1)	O(1)	O(1)
before()	O(1)	O(n)	O(1)
insertFirst()	O(n)	O(1)	O(1)
insertLast()	O(1)	O(n)	O(1)
insertBefore()	O(n)	O(n)	O(1)
insertAfter()	O(n)	O(1)	O(1)
remove()	O(n)	O(n)	O(1)