Data Structures and Algorithms

The List Abstract Data Type - Singly-Linked List

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

October 29, 2018

Learning outcomes

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

- understand the operations of the list and position abstract data types
- be able to implement a singly-linked list
- know the complexity of all of the operations of the singly-linked list implementation

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 int 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,
      int d);
7    public Position insertAfter(Position p, int
      d);
8    public Position insertFirst(int d);
9    public Position insertLast(int d);
10   public int remove(Position p);
11   public int size();
12   public boolean isEmpty();
13 }
```

The List Abstract Data Type

Implementation Strategies

- Array based implementation
- 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

Singly-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

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

Singly-Linked List Implementation

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

Variables:

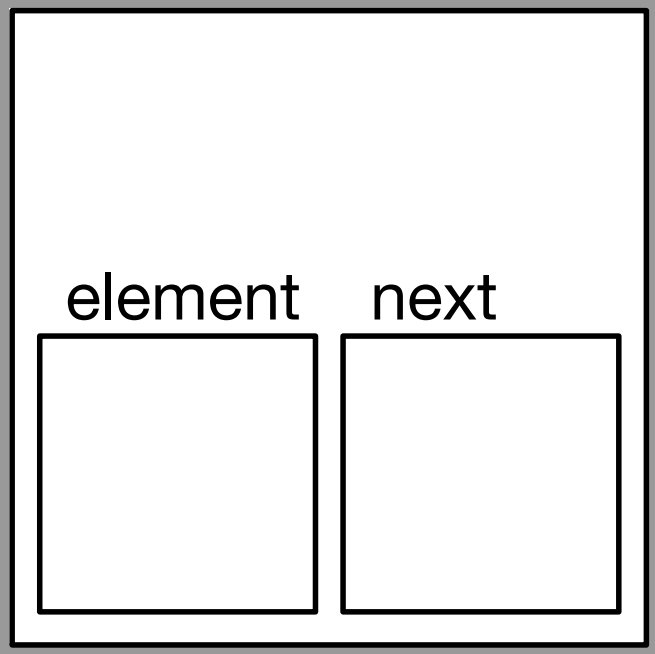
- A reference to the first position in the list

```
private Node first;
```

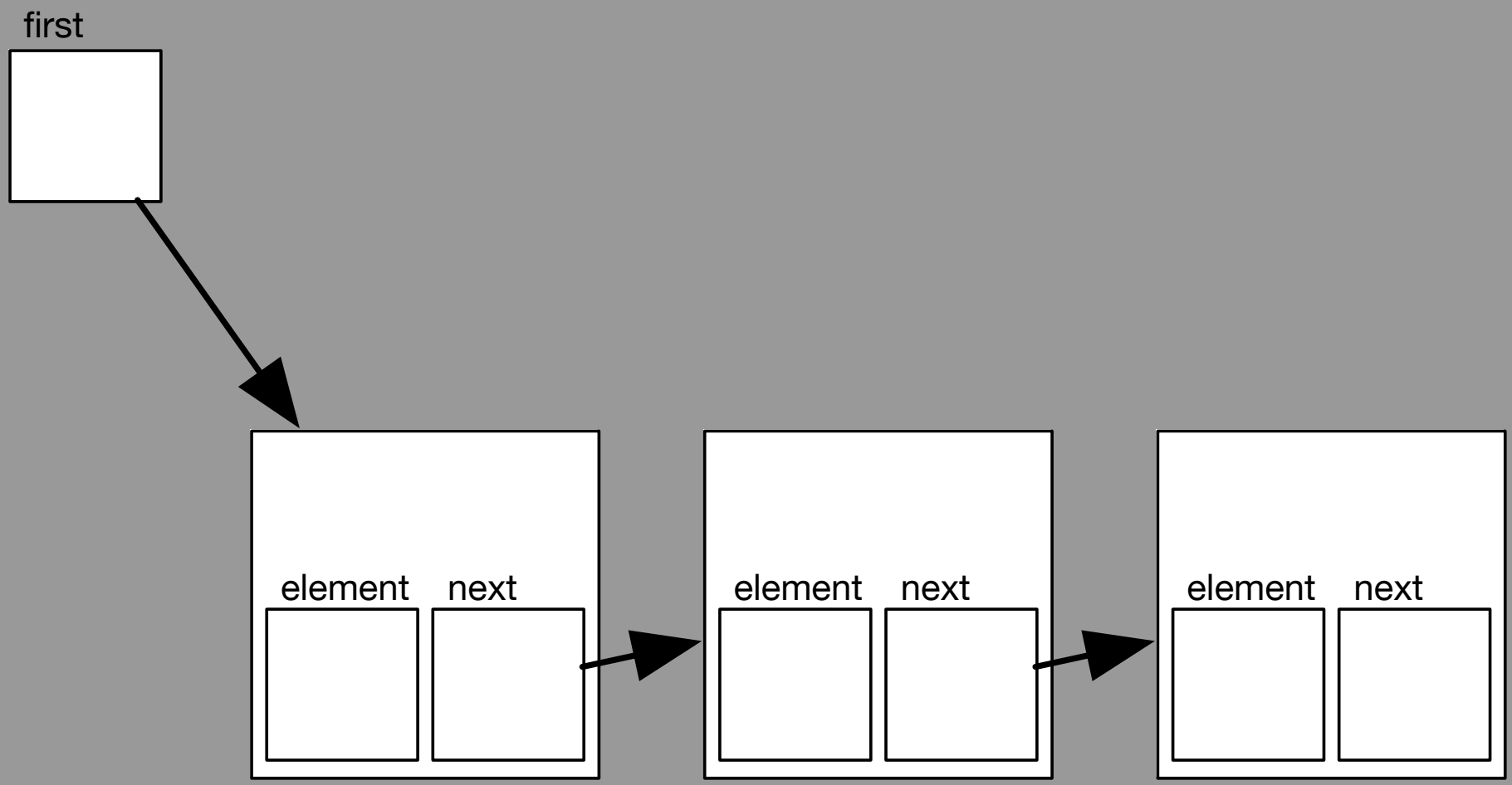
- A number to keep track of the size

```
private int size;
```

Representation of a Node Object



Representation of a List



Singly-Linked List Operations

- first()
  - Return the reference that is stored in the variable first
- size()
  - Return the value of the size variable
- isEmpty()
  - Return the result of the expression `size == 0`
- after(p)
  - Return the next reference of p
- last()
  - We only have a reference to the first position in the list
  - Each position only knows about the next element in the list
  - We must follow the list from position to position until we reach the end of the list
  - The end of the list is when there is no reference stored in the variable `next` of a position

Singly-Linked List Operations

Last Algorithm Pseudocode

```
1 Algorithm last():
2   Input: None
3   Output: The last position in the list
4 if isEmpty() then
5     return null
6 N ← first
7 while(N.next != null) do
8     N ← N.next
9 return N
```

Singly-Linked List Operations

before(p)

- Should get the position before p
- But we do not have a reference to the Node before us in the list
- We have to search through the list until we find p and return the position before it

```
1 Algorithm before(p):
2   Input: The position p we want to find the
     position before
3   Output: The position before p
4 if isEmpty() OR p = first then
5     return null
6 I ← first
7 while(I.next != null AND p != I.next) do
8     I ← I.next
9 return I
```

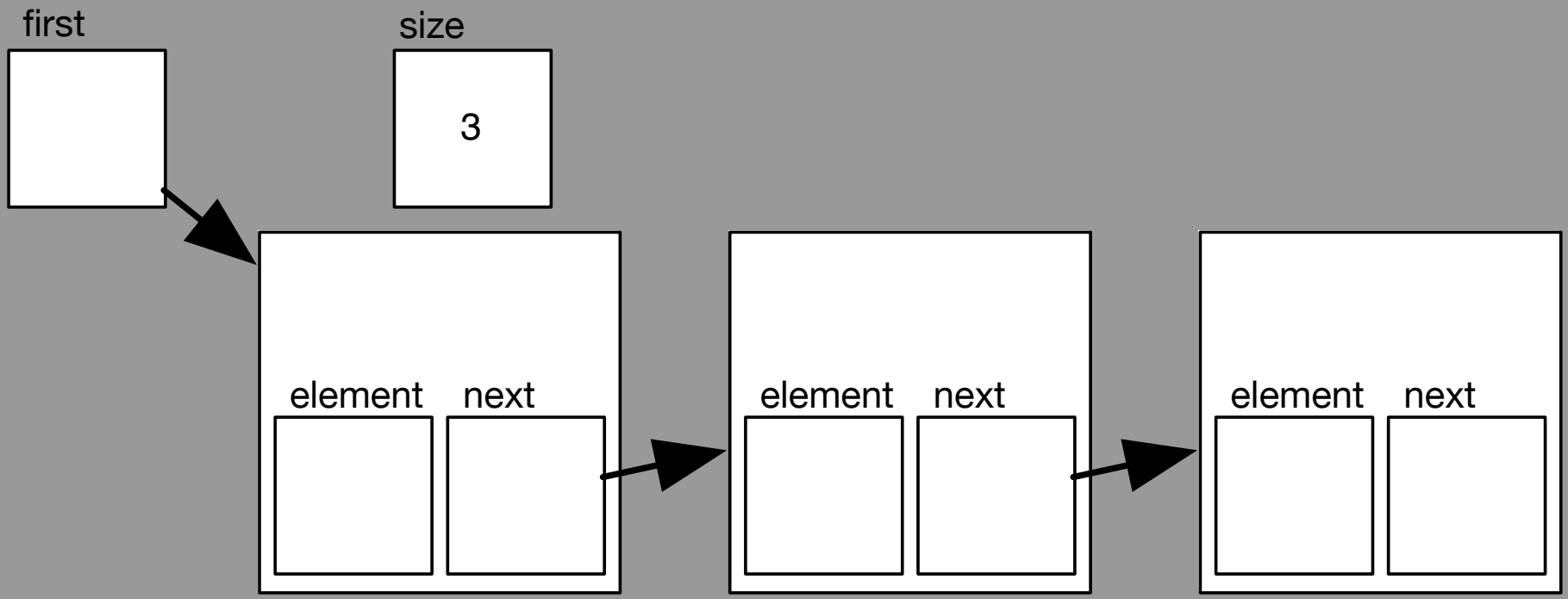
Singly-Linked List Operations

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 first reference so that it points to n
  - Increment the size

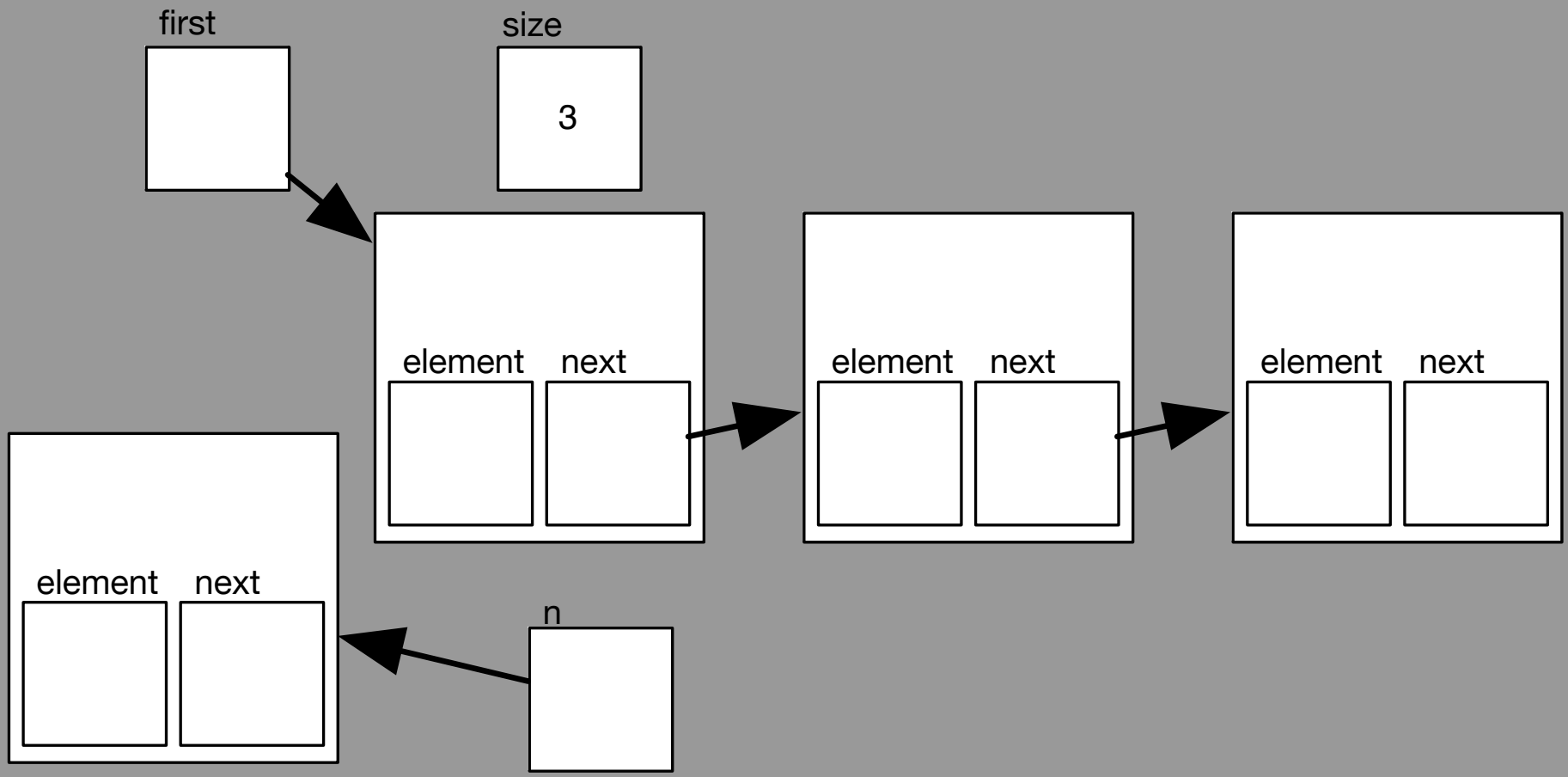
Singly-Linked List Operations

insertFirst(d)



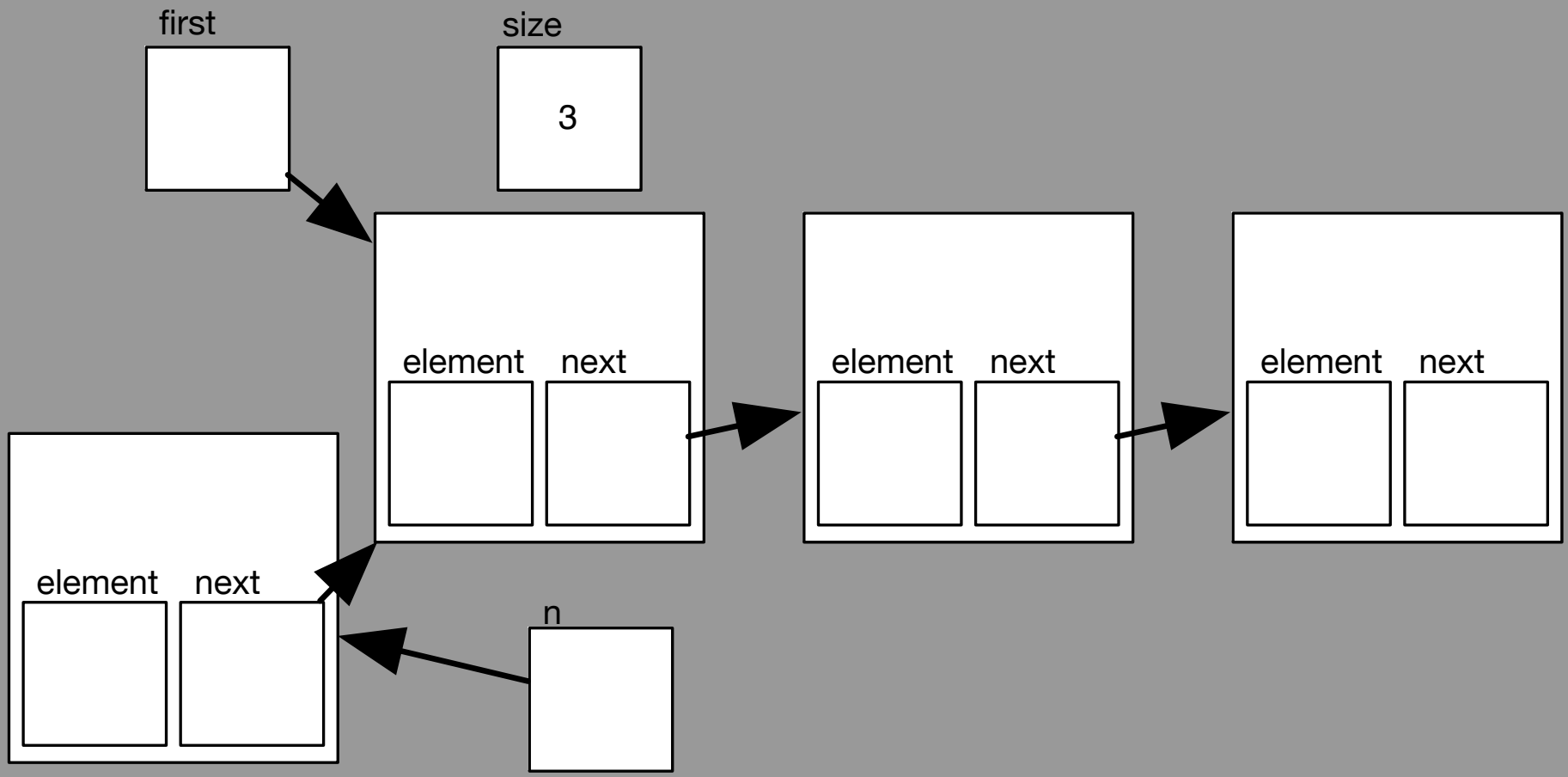
Singly-Linked List Operations

insertFirst(d)



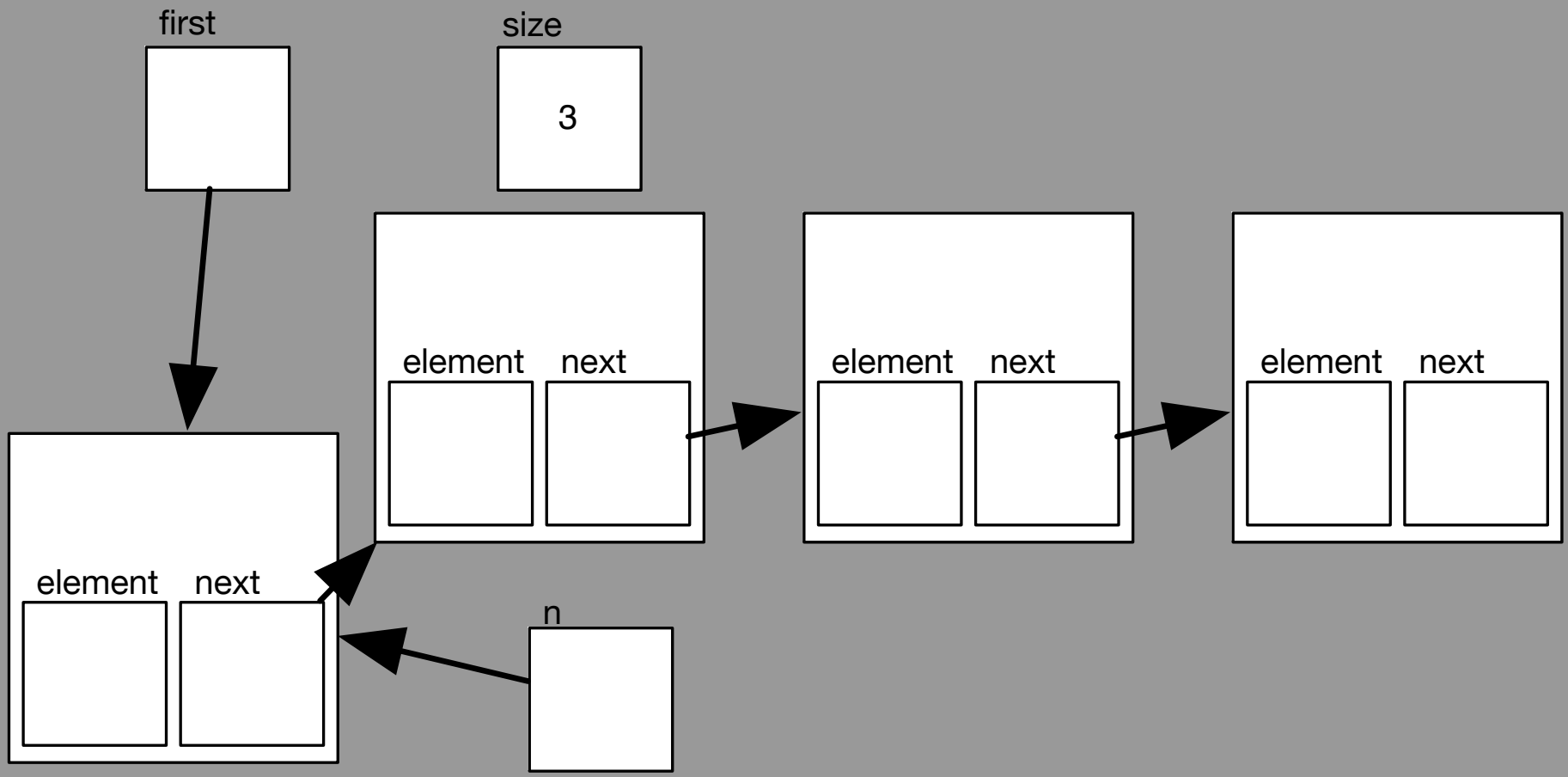
Singly-Linked List Operations

insertFirst(d)



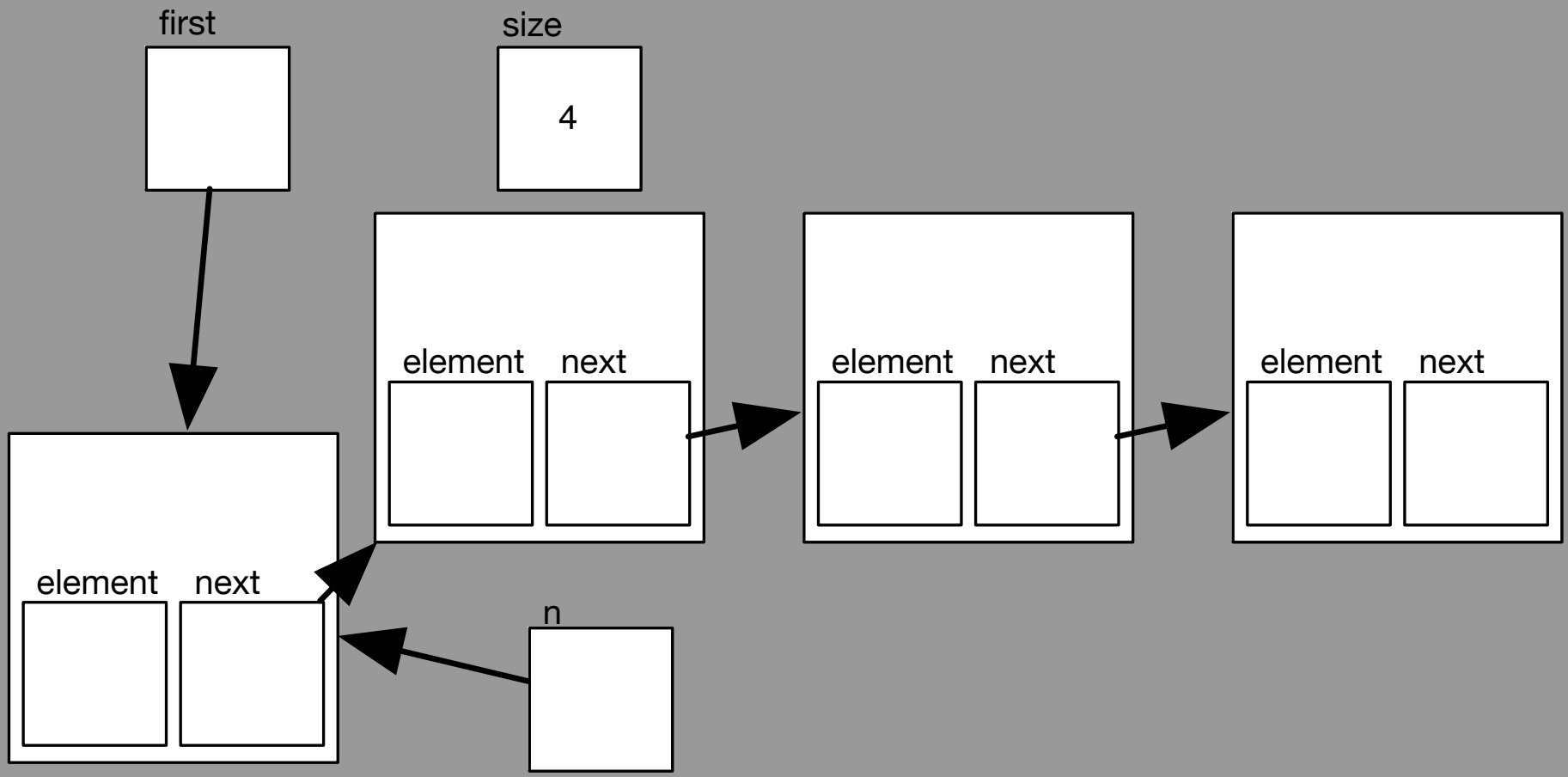
Singly-Linked List Operations

insertFirst(d)



Singly-Linked List Operations

insertFirst(d)



Singly-Linked List Operations

insertFirst(d)

```
1 Algorithm insertFirst (d):
2   Input: The value to be inserted
3   Output: The position it was inserted in
4
5 Create node n containing d
6 n.next ← first
7 first ← n
8 size ← size + 1
9 return n
```

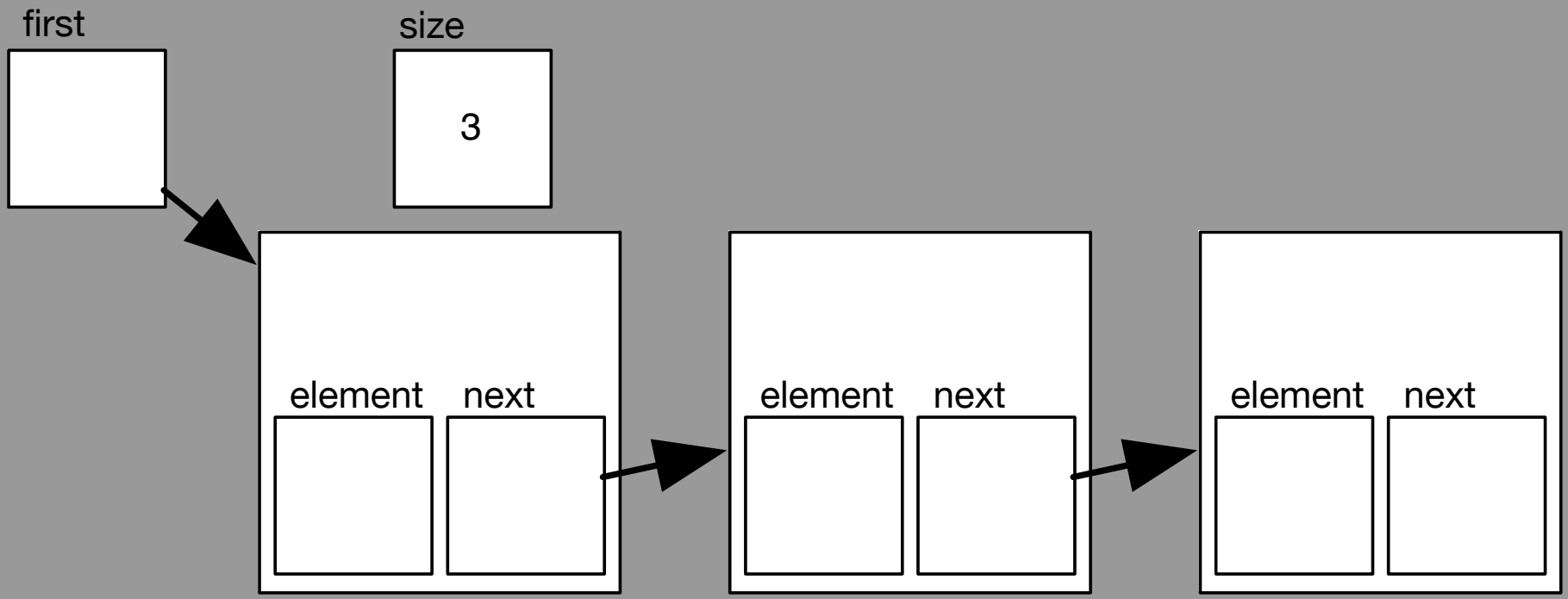
Singly-Linked List Operations

insertLast(d)

- Insert the object into the last position
  - Create a new Node n containing the object
  - Find the last position, called l, in the list using the last operation
  - Change the next reference of l so it points to n
  - Increment the size
- What happens if the list is empty?

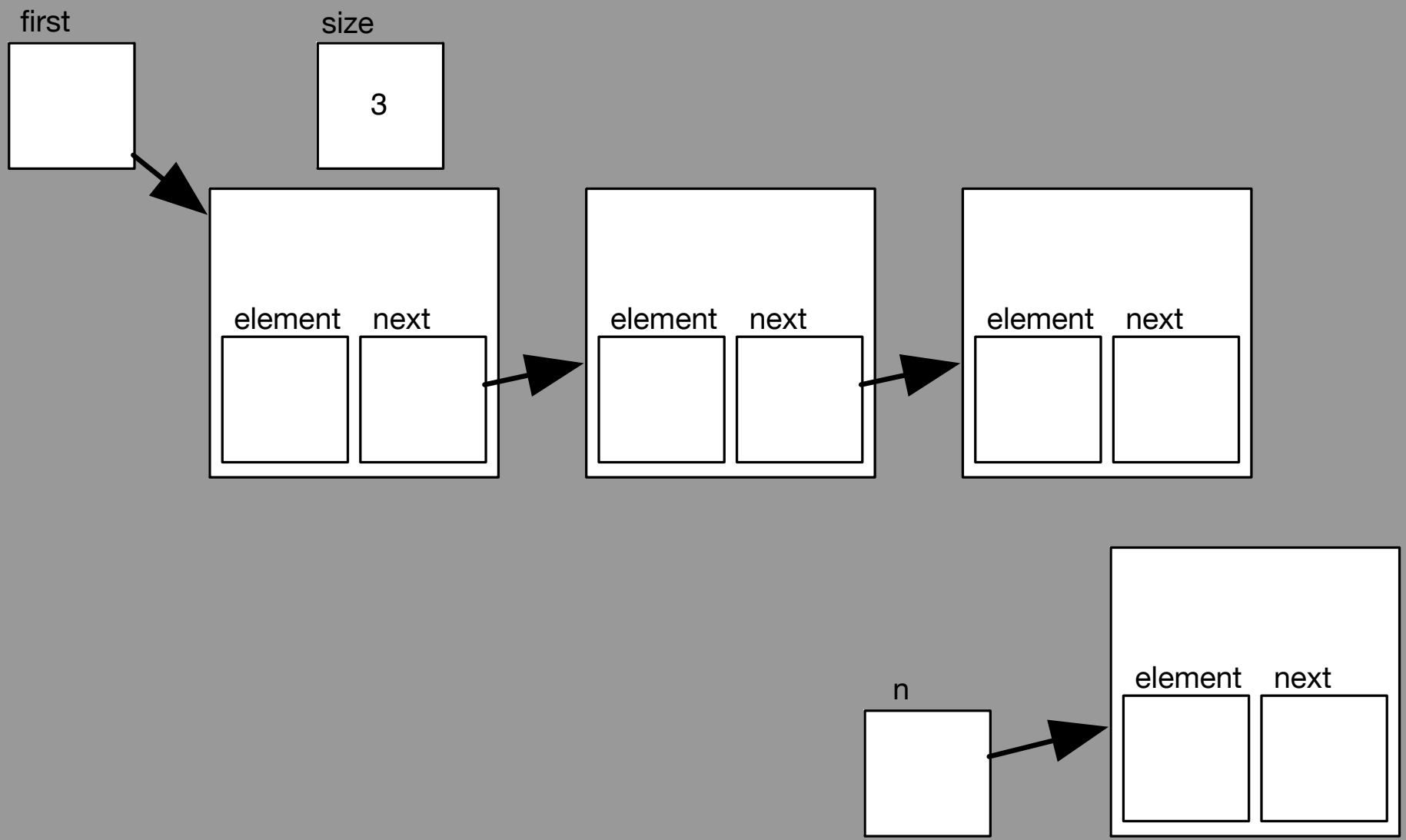
Singly-Linked List Operations

insertLast(d)



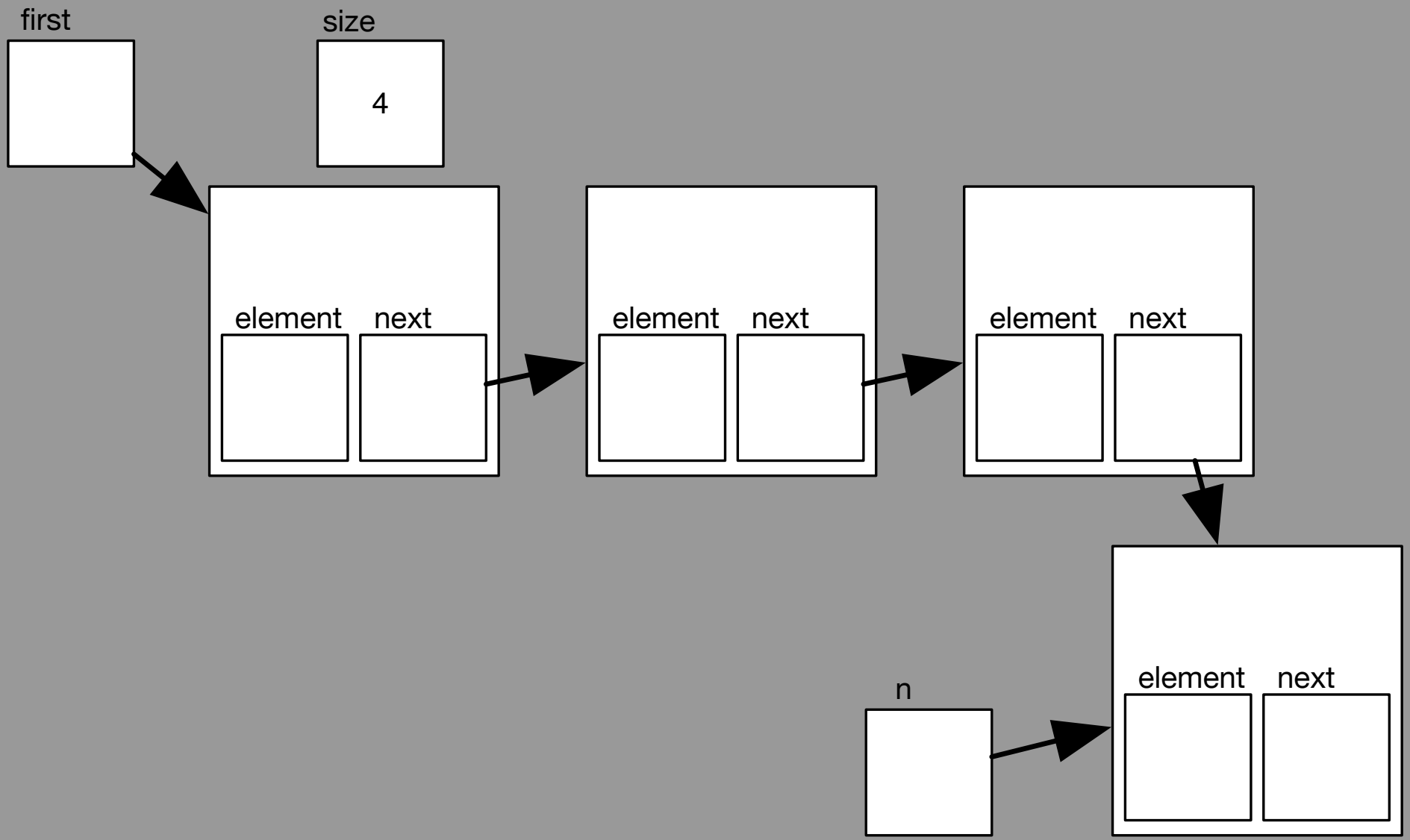
Singly-Linked List Operations

insertLast(d)



Singly-Linked List Operations

insertLast(d)



Singly-Linked List Operations

insertLast(d)

```
1  Algorithm insertLast(d):
2    Input: The value to be inserted
3    Output: The position it was inserted in
4
5  Create node n containing d
6  if isEmpty() then
7      return insertFirst(d)
8  Create node n containing d
9  l ← last()
10 l.next ← n
11 size ← size + 1
12 return n
```

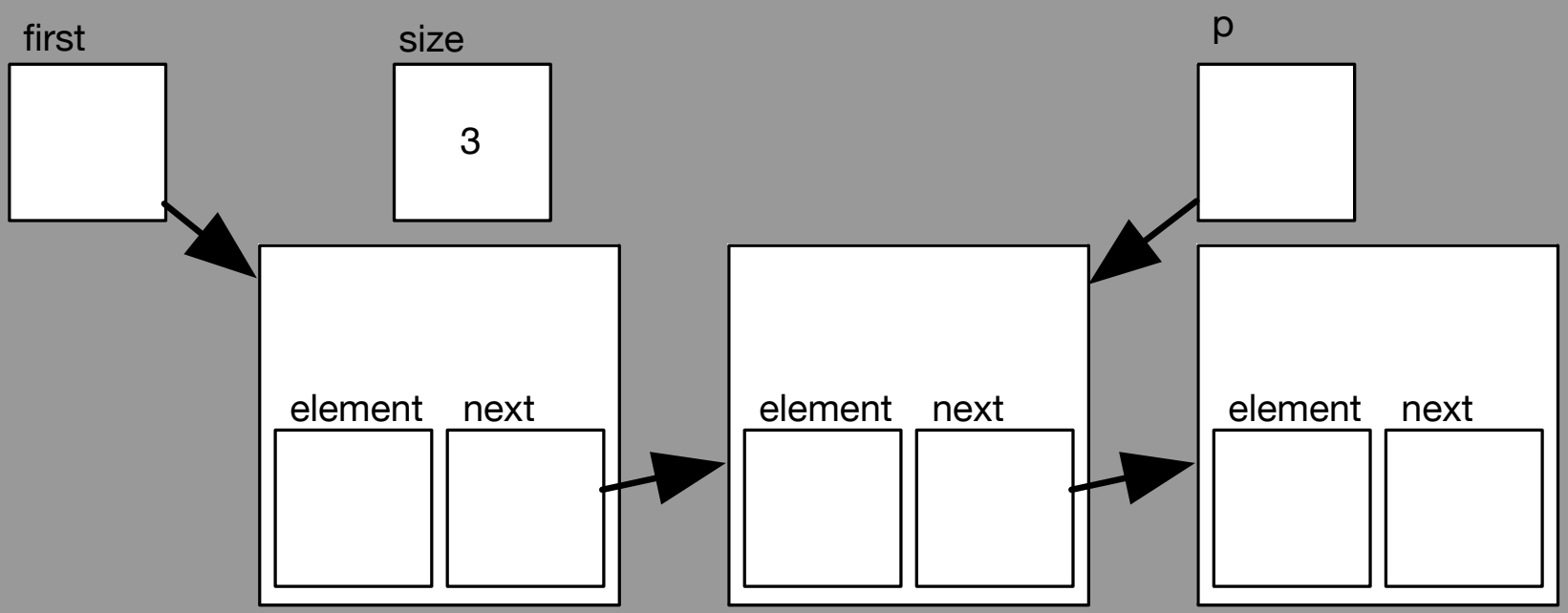
Singly-Linked List Operations

insertAfter(p, d)

- Insert the object into the position after p
  - Create a new Node n containing the object
  - Change next of n so it points to the next of p
  - Change the next of p so it points to n
  - Increment the size

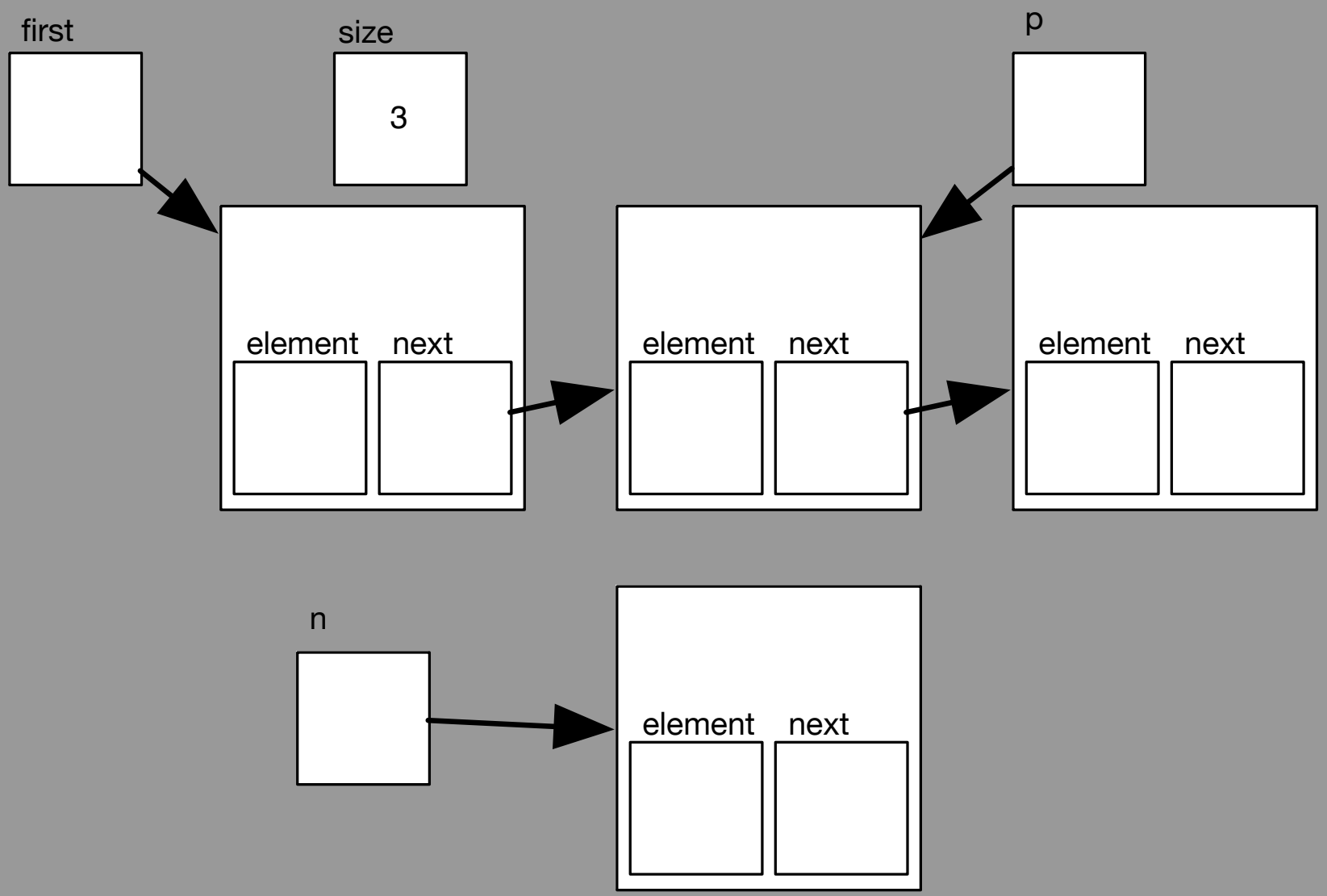
Singly-Linked List Operations

insertAfter(p, d)



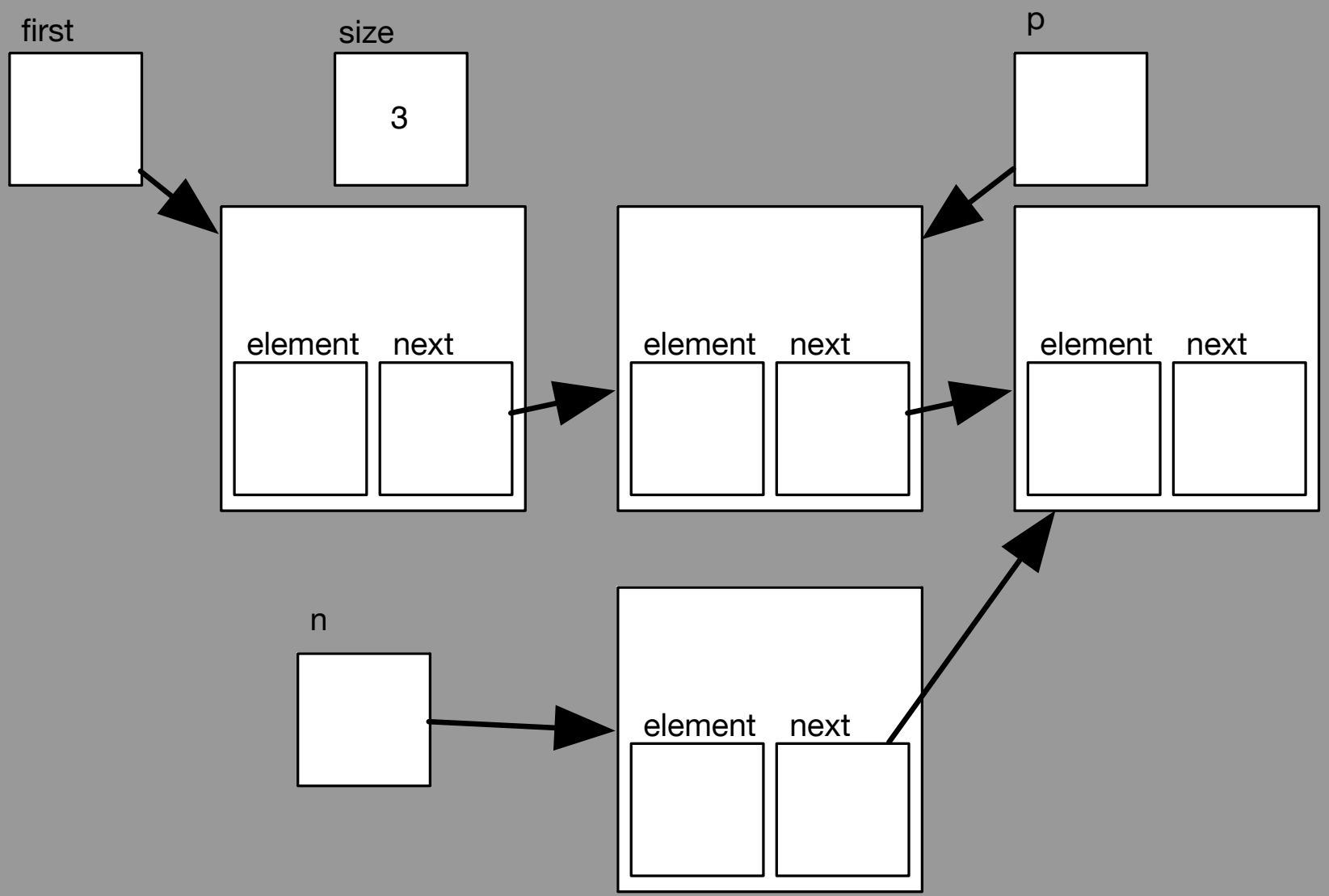
Singly-Linked List Operations

insertAfter(p, d)



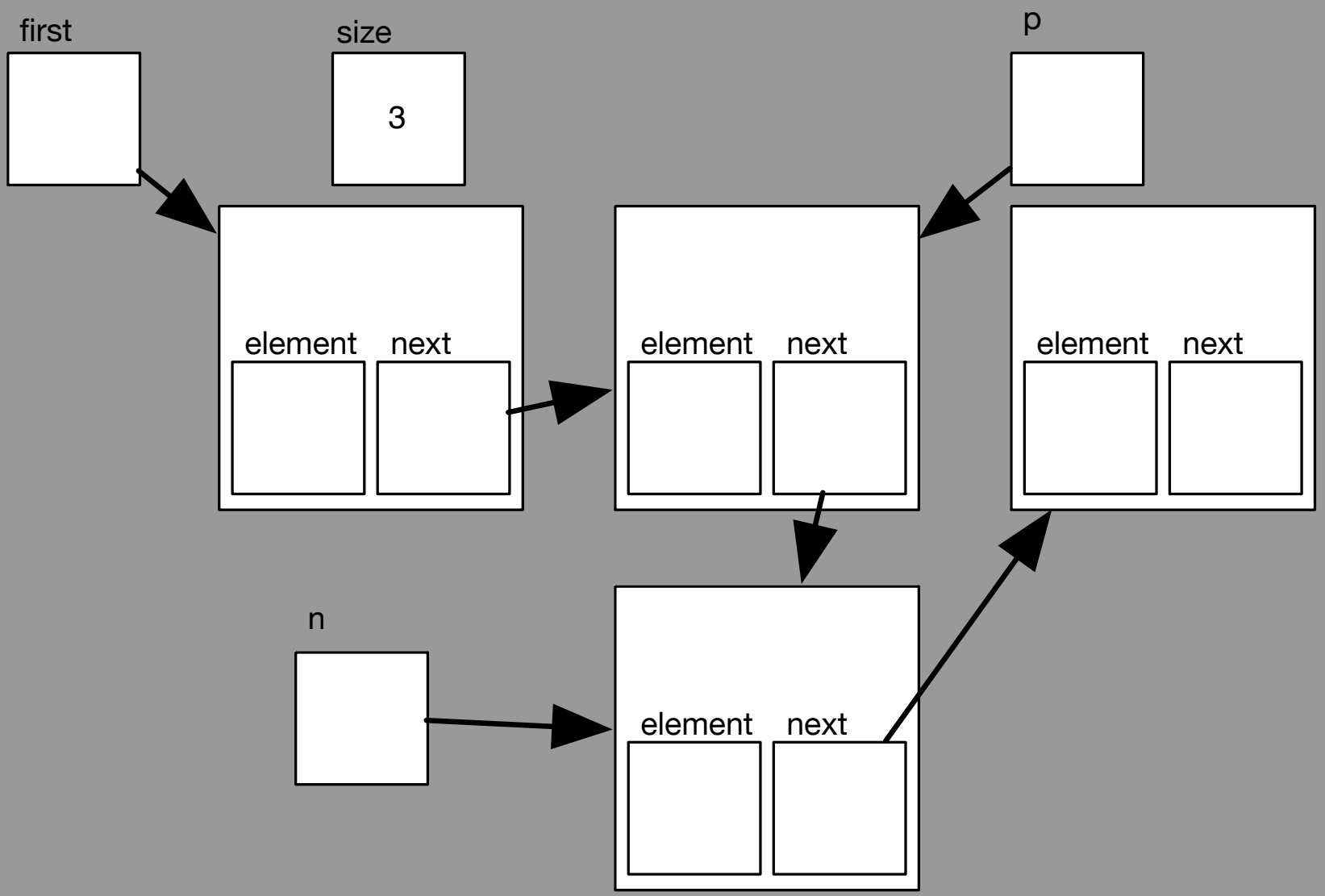
Singly-Linked List Operations

insertAfter(p, d)



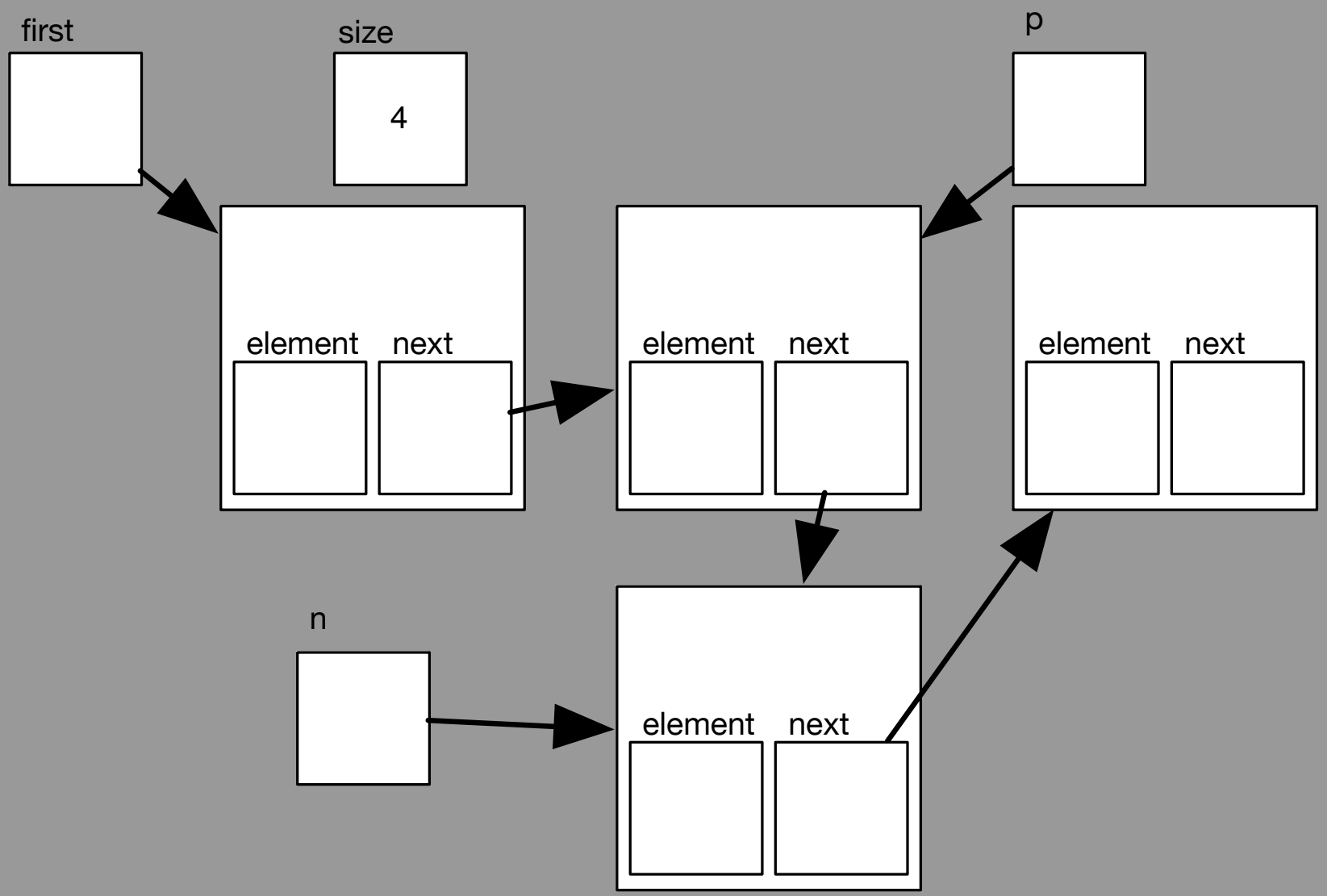
Singly-Linked List Operations

insertAfter(p, d)



Singly-Linked List Operations

insertAfter(p, d)



Singly-Linked List Operations

insertAfter(p, d)

```
1 Algorithm insertAfter(p, d):
2   Input: The value to be inserted and the
     position it should be inserted after
3   Output: The position it was inserted in
4
5 Create node n containing d
6 n.next ← p.next
7 p.next ← n
8 size ← size + 1
9 return n
```

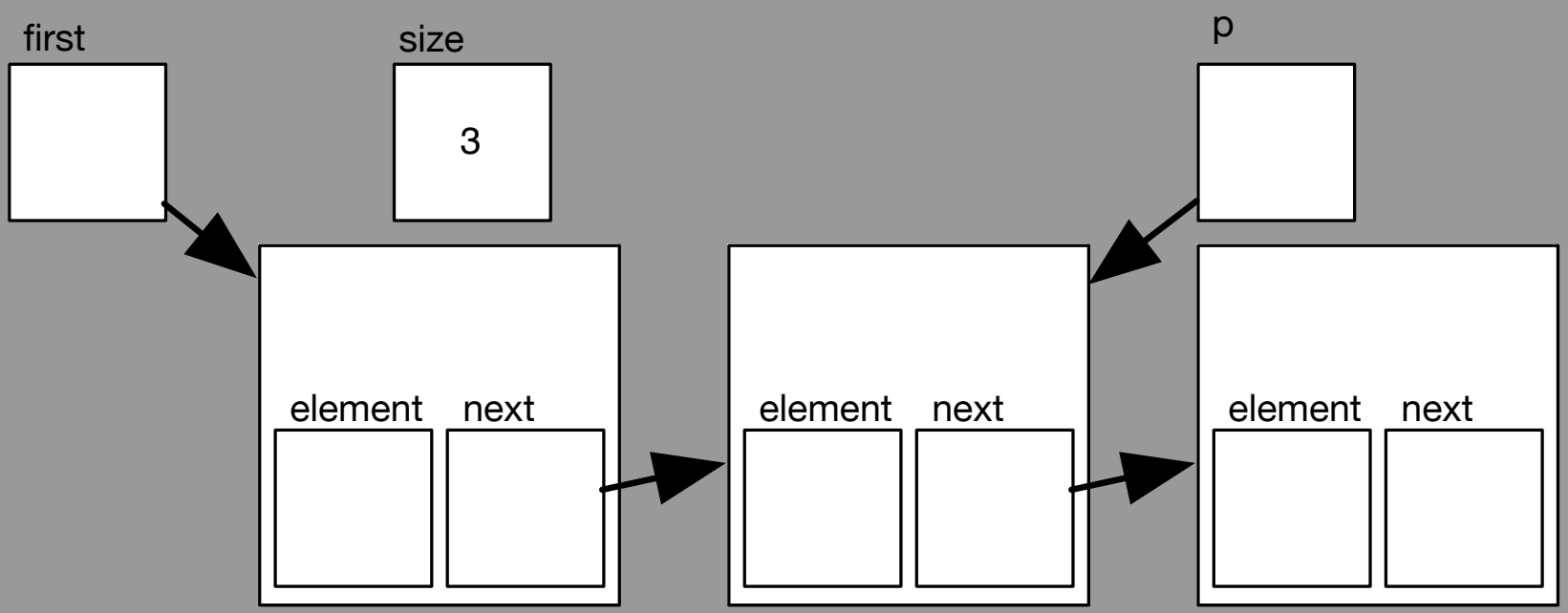
Singly-Linked List Operations

insertBefore(p, d)

- Insert the object into the position before p
  - Create a new Node n containing the object
  - Use the before operation to get the position before p, called bp
  - Change next of n so it points to p
  - Change the next of bp so it points to n
  - Increment the size
- What happens if p is the fist position?

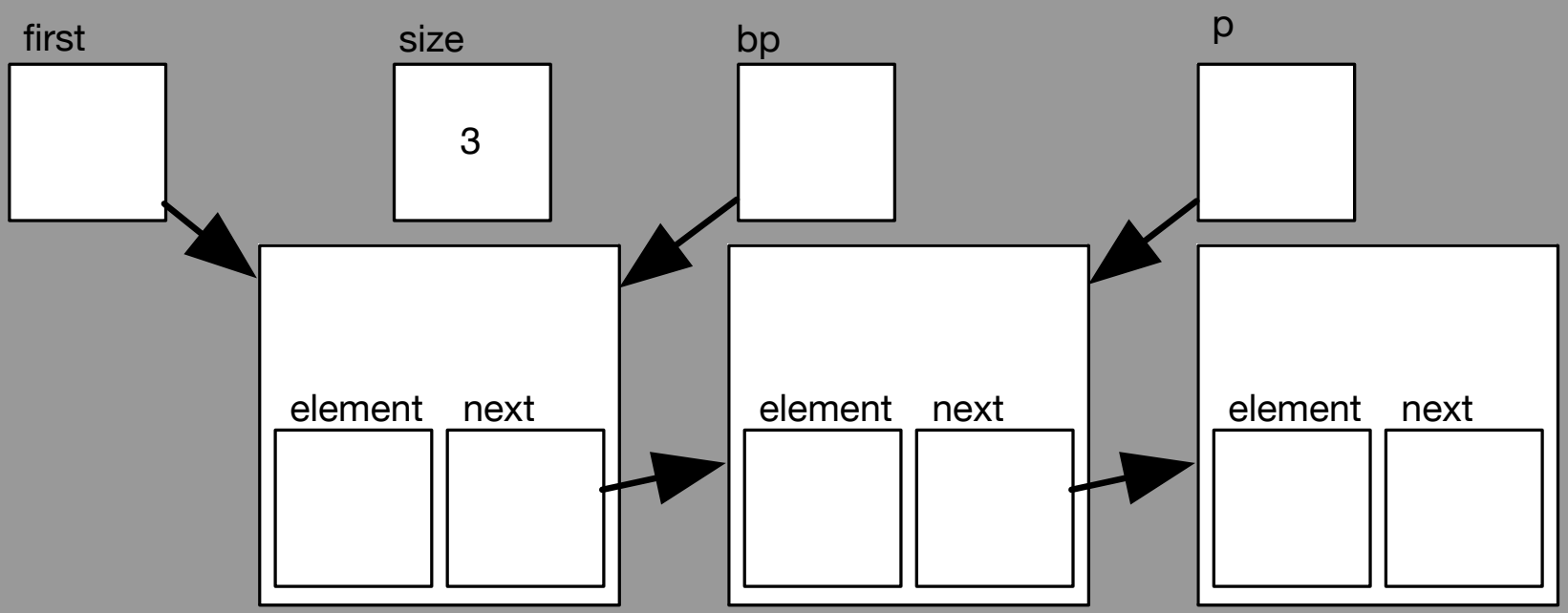
Singly-Linked List Operations

insertBefore(p, d)



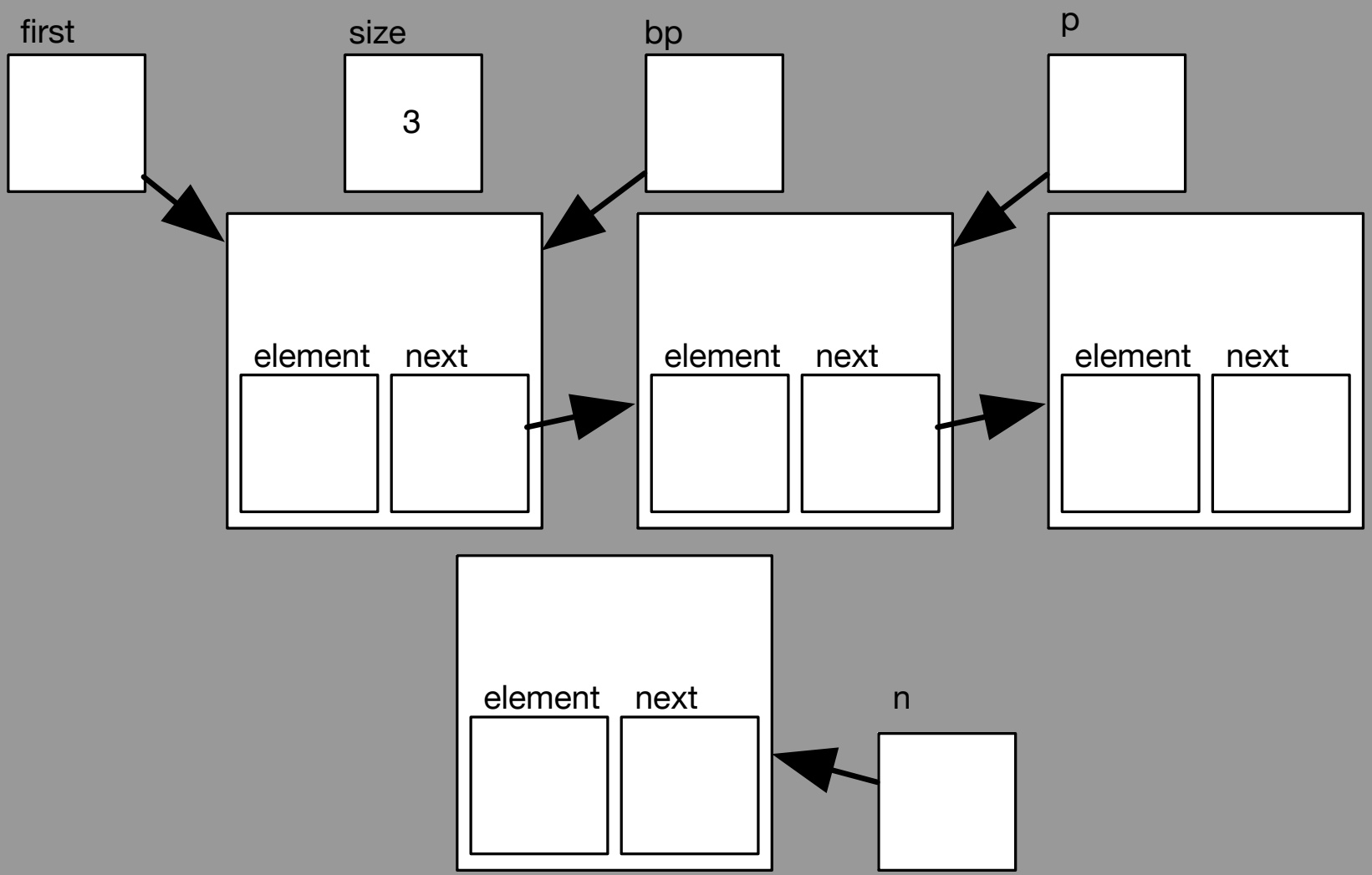
Singly-Linked List Operations

insertBefore(p, d)



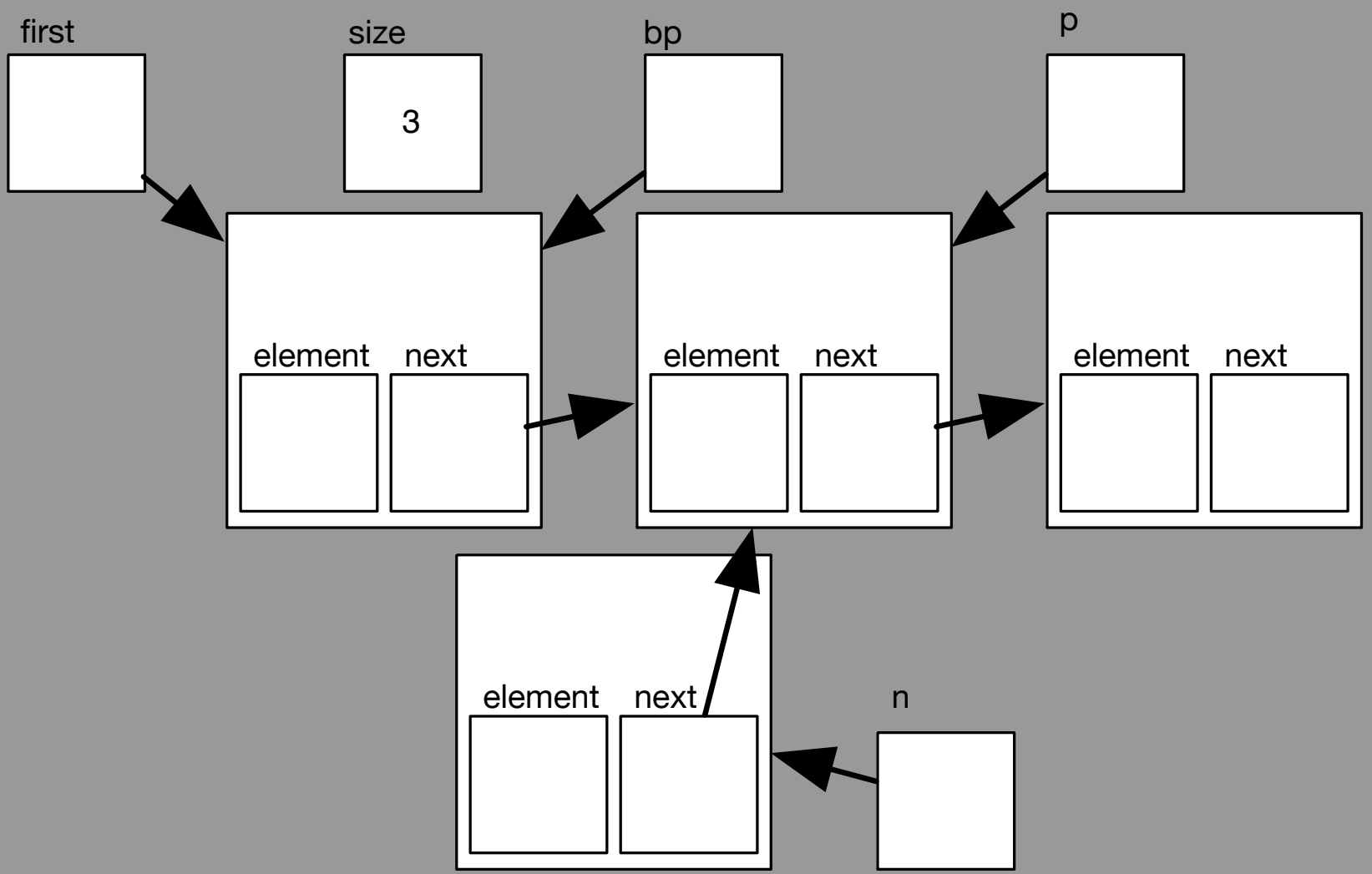
Singly-Linked List Operations

insertBefore(p, d)



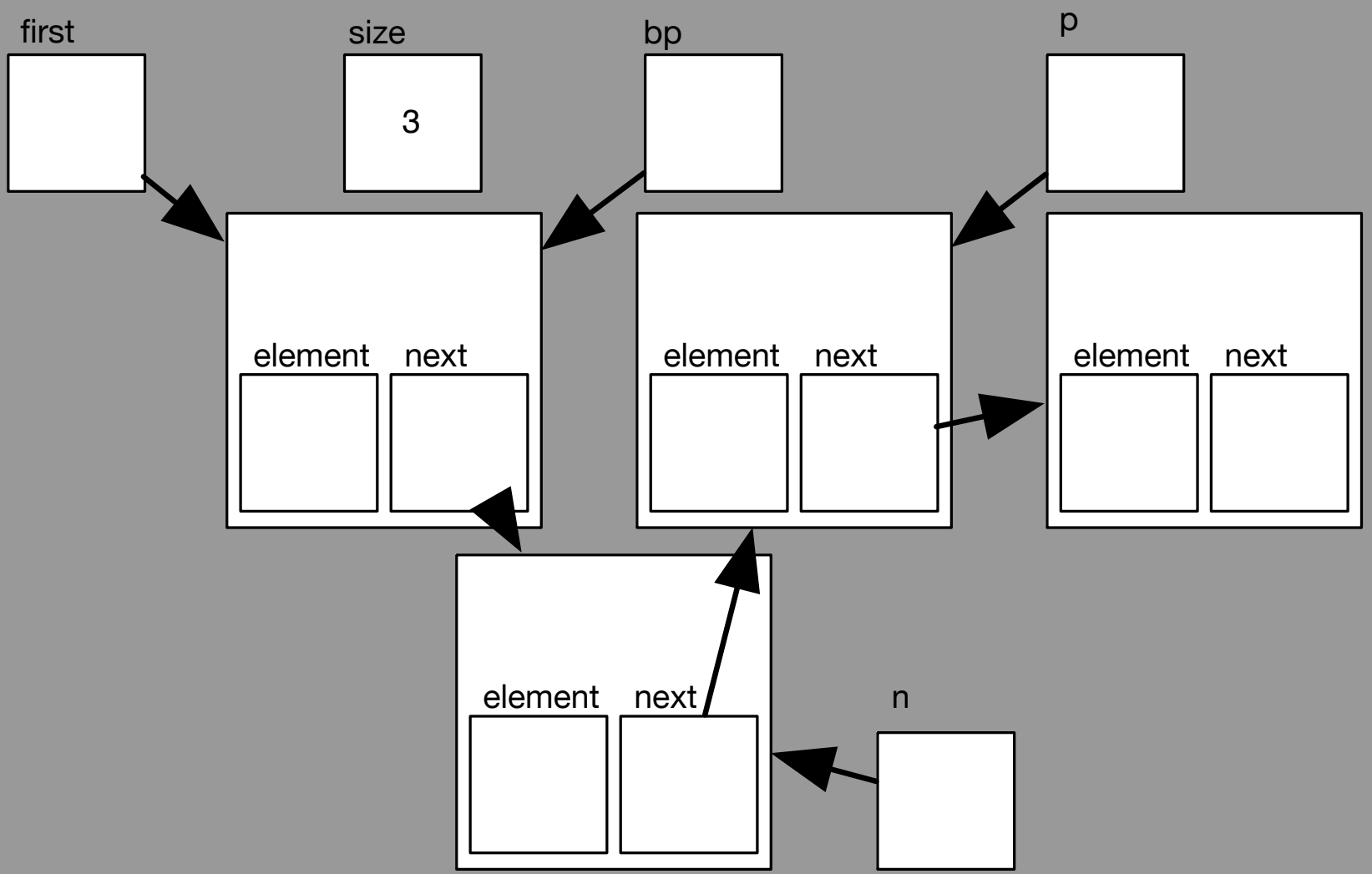
Singly-Linked List Operations

insertBefore(p, d)



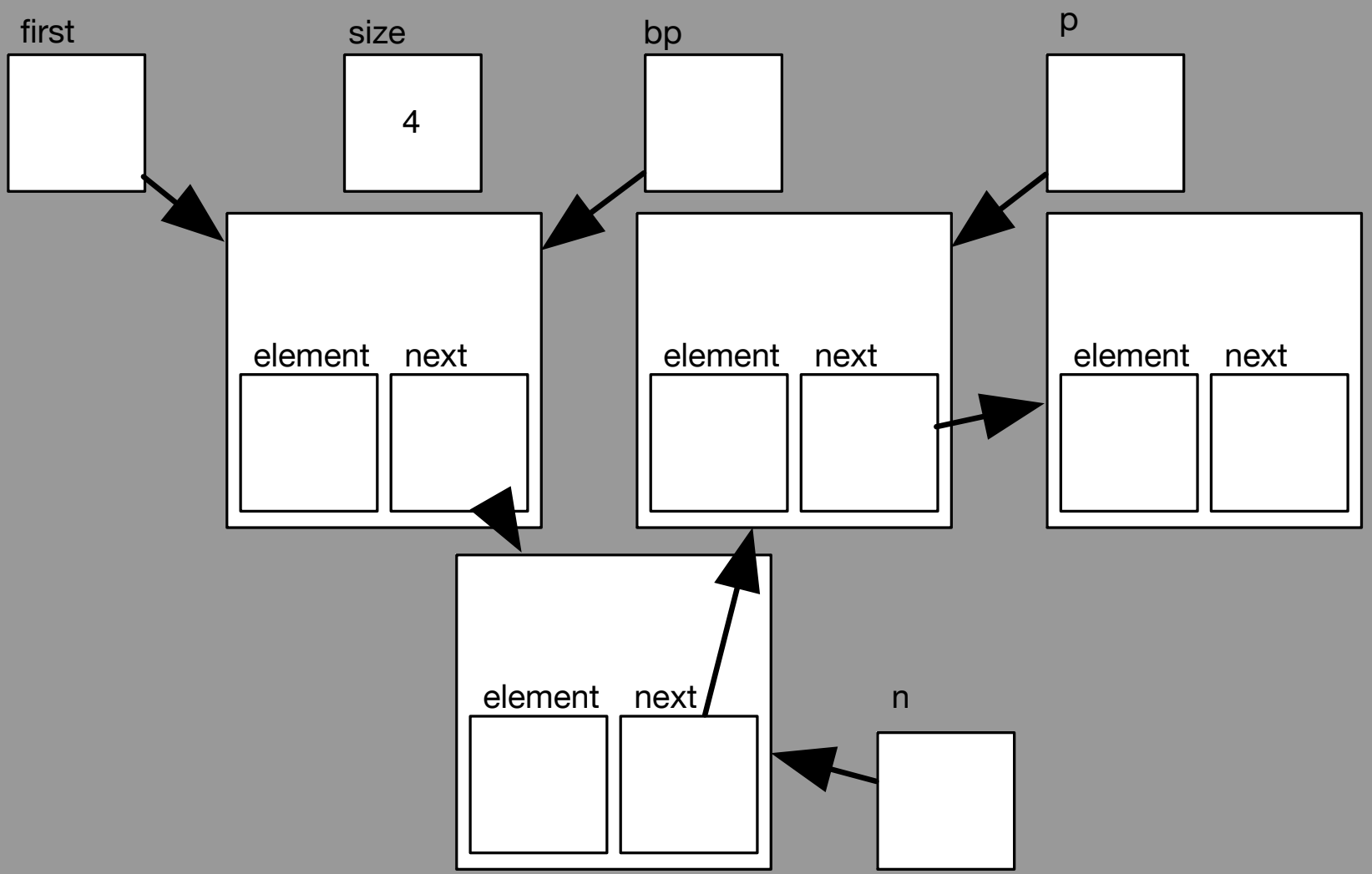
Singly-Linked List Operations

insertBefore(p, d)



Singly-Linked List Operations

insertBefore(p, d)



Singly-Linked List Operations

insertBefore(p, d)

```
1  Algorithm insertBefore(p, d):
2    Input: The value to be inserted and the
       position it should be inserted before
3    Output: The position it was inserted in
4
5  if p = first then
6    return insertFirst(d)
7  Create node n containing d
8  bp ← before(p)
9  n.next ← p
10 bp.next ← n
11 size ← size + 1
12 return n
```

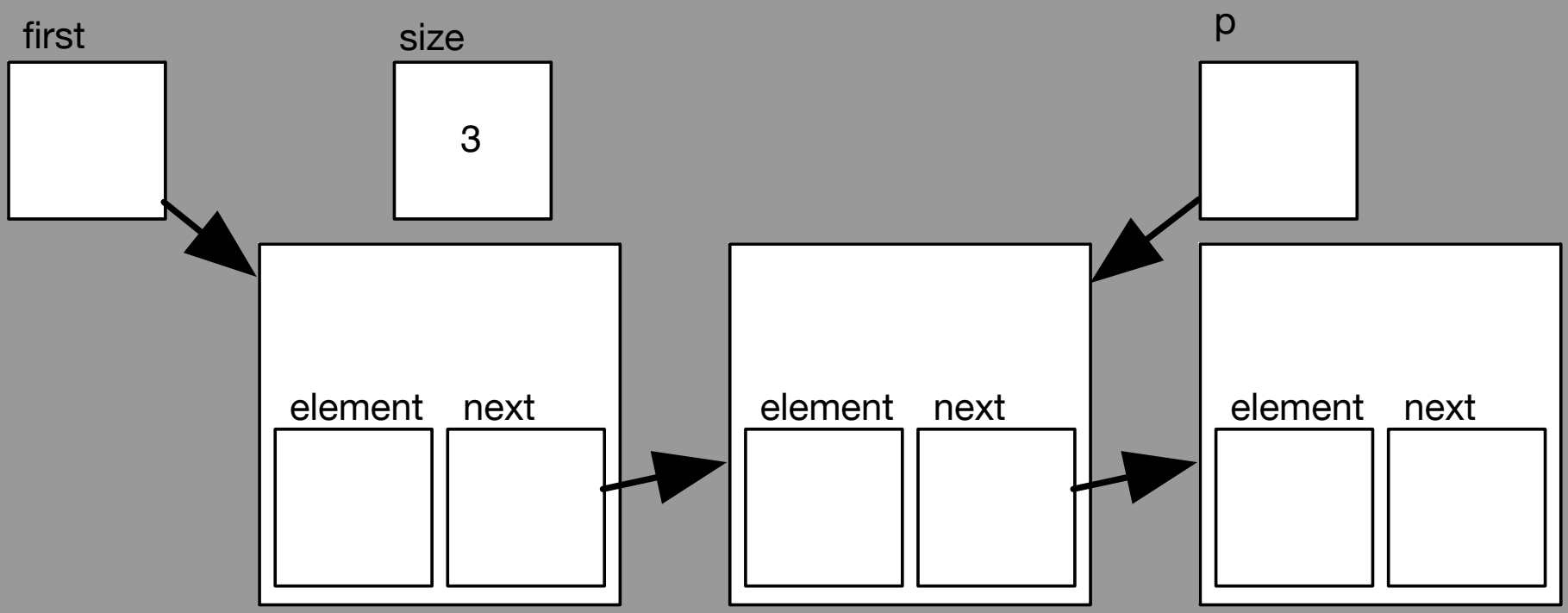
Singly-Linked List Operations

remove(p)

- Remove the object p from the list
  - Copy the data from inside the position p to variable d
  - Use the before operation to get the position before p, called bp
  - Change next of bp so it points to p.next
  - Decrement the size
  - return d
- What happens if p is the fist position?

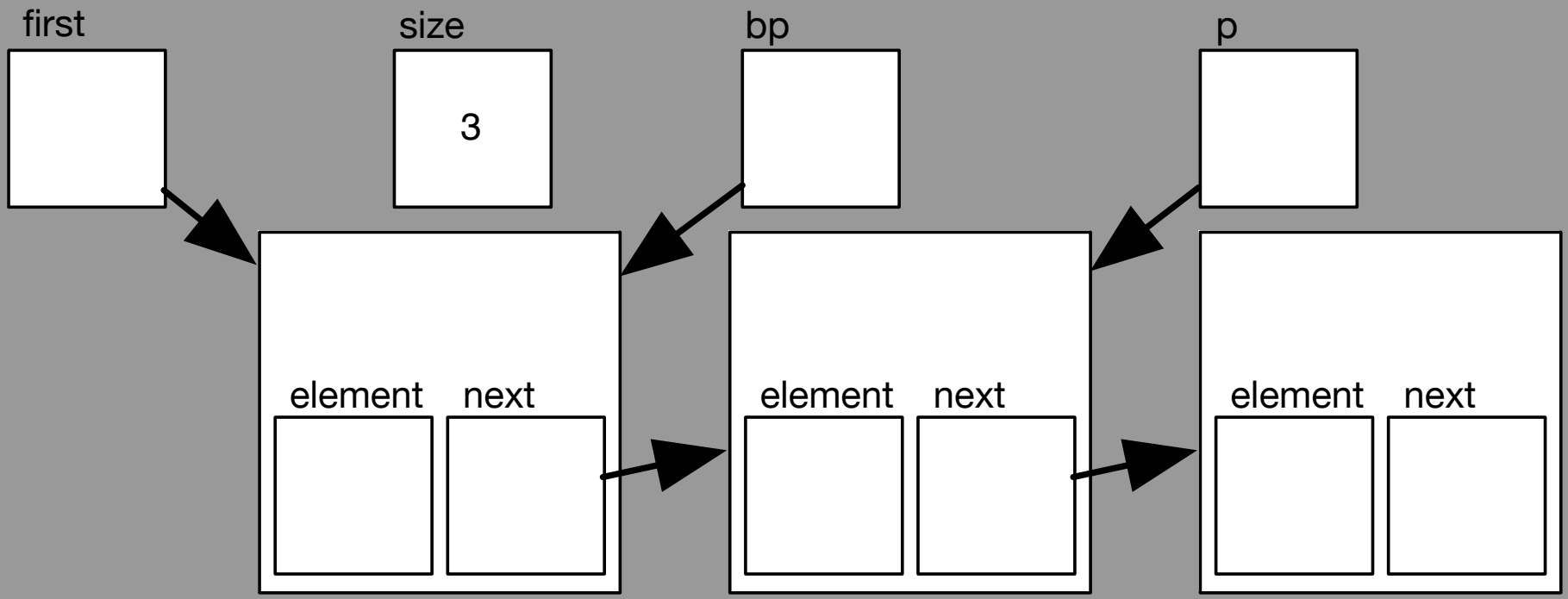
Singly-Linked List Operations

remove(p)



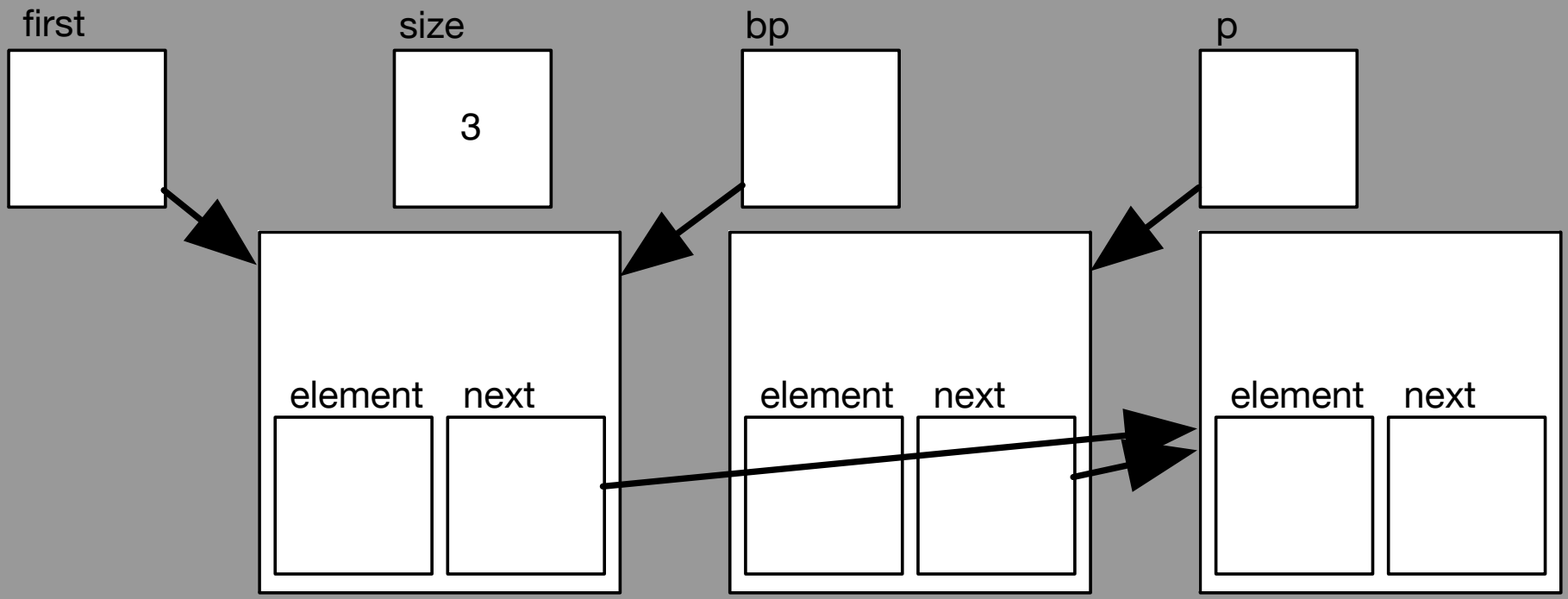
Singly-Linked List Operations

remove(p)



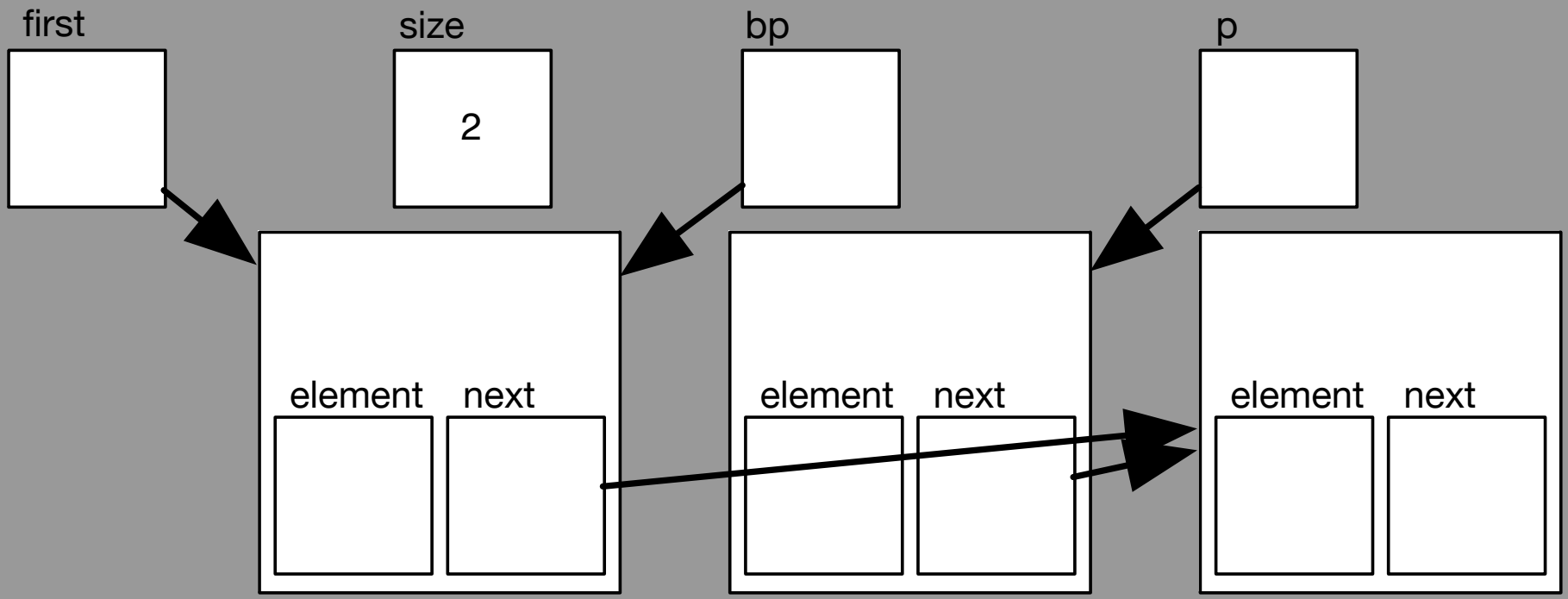
Singly-Linked List Operations

remove(p)



Singly-Linked List Operations

remove(p)



Singly-Linked List Operations

remove(p)

```
1  Algorithm remove(p):
2    Input: The position to be removed from the
      list
3    Output: The value that was removed
4
5  Copy data of p to variable d
6  if p = first then
7       first = p.next
8  else
9       bp ← before(p)
10      bp.next ← p.next
11 p.next ← null
12 size ← size - 1
13 return d
```

Singly-Linked List Operations

remove(d)

- Remove the object with the value d from the list.
- d is the value in the object, but not the object itself.
- First, you need to find the object(s) with a value of d. Remove it/them.

Table of Contents

Algorithmic Complexity

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

Algorithmic Complexity

before(p)

```
1 Algorithm before(p):
2   Input: The position p we want to find the
     position before
3   Output: The position before p
4 if isEmpty() OR p = first then
5     return null
6 I ← first
7 while(I.next != null AND p != I.next) do
8     I ← I.next
9 return I
```

- Complexity is O(n)

Algorithmic Complexity

insertFirst(d)

```
1 Algorithm insertFirst (d):
2   Input: The value to be inserted
3   Output: The position it was inserted in
4
5 Create node n containing d
6 n.next ← first
7 first ← n
8 size ← size + 1
9 return n
```

- Complexity is O(1)

Algorithmic Complexity

insertLast(d)

```
1  Algorithm insertLast(d):
2    Input: The value to be inserted
3    Output: The position it was inserted in
4
5  Create node n containing d
6  if isEmpty() then
7      return insertFirst(d)
8  Create node n containing d
9  l ← last()
10 l.next ← n
11 size ← size + 1
12 return n
```

- Complexity is O(n)

Algorithmic Complexity

insertAfter(p, d)

```
1 Algorithm insertAfter(p, d):
2   Input: The value to be inserted and the
     position it should be inserted after
3   Output: The position it was inserted in
4
5 Create node n containing d
6 n.next ← p.next
7 p.next ← n
8 size ← size + 1
9 return n
```

- Complexity is O(1)

Algorithmic Complexity

insertBefore(p, d)

```
1  Algorithm insertBefore(p, d):
2    Input: The value to be inserted and the
       position it should be inserted before
3    Output: The position it was inserted in
4
5  if p = first then
6    return insertFirst(d)
7  Create node n containing d
8  bp ← before(p)
9  n.next ← p
10 bp.next ← n
11 size ← size + 1
12 return n
```

- Complexity is O(n)

Algorithmic Complexity

remove(p)

```
1  Algorithm remove(p):
2    Input: The position to be removed from the
      list
3    Output: The value that was removed
4
5  Copy data of p to variable d
6  if p = first then
7       first = p.next
8  else
9       bp ← before(p)
10      bp.next ← p.next
11 p.next ← null
12 size ← size - 1
13 return d
```

- Complexity is O(n)