Linked Data Structures I: Singly-Linked Lists

What's Wrong With Arrays?

- A Java array is not ideally suited as a data representation of the various ***collection types***
  - Similarly for the Java ***collections framework***

What's Wrong With Arrays?

- A Java array is not ideally suited as a data representation of the various ***collection types***
  - Similarly for the Java ***collectio ramework***

Any type whose abstract mathematical model involves a ***`string`*** or ***`set`*** or ***`multiset`*** (or ***`tree`*** or ***`binary tree`***?) is a "collection" type.

What's Wrong With Arrays?

- A Java array is not ideally suited as a data representation of the various ***collection types***
  - Similarly for the Java ***collectio ramework***

Examples are `Queue`, `Stack`, `Sequence`, `Set`, `Map`, `SortingMachine`, ...

What's Wrong With Arrays?

- A Java array is not ideally suited as a data representation of the various ***collection types***
  - Similarly for the Java ***collections framework***

This is part of the package `java.util`, and includes many types similar to the OSU CSE components.

Collection Terminology

- ***Fixed size*** means the size/length of a collection is “inflexible”, i.e., it is determined at initialization of the collection and cannot be incrementally adjusted
  - A classical synonym is ***static***; this term unfortunately means other things in Java
- ***Dynamic*** means the size/length of a collection is “flexible”, i.e., it can be incrementally adjusted by “adding” and “removing” entries, even from the middle

Collection Terminology

- ***Direct access*** means the entries of a collection (typically with a **`string`** model) may be accessed by providing an **`int`** position/index of any entry in the collection
  - A classical but unfortunate synonym is ***random access***; nothing random about it!
- ***Sequential access*** means the entries of a collection (with a **`string`** model) may be accessed in increasing order of position by accessing the "next" entry in the collection

Collection Te

- ***Direct access*** means
  collection (typically wit
  may be accessed by
  position/index of a e
  - A classical but unfortunate synonym is ***random access***; nothing random about it!
- ***Sequential access*** means the entries of a collection (with a ***`string`*** model) may be accessed in increasing order of position by accessing the "next" entry in the collection

We might say any collection with an iterator allows sequential access, but this is about the *other* methods for access.

Key Pros and Cons of Arrays

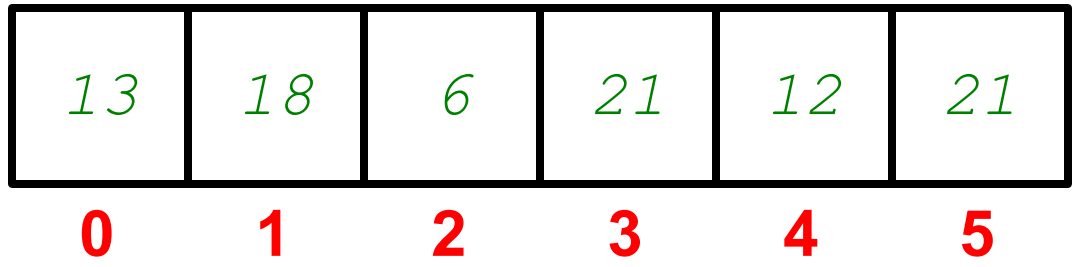
- Pros:
  - Direct access is fast, i.e., it takes ***constant time*** independent of the length of the array
- Cons:
  - Its fixed size limits array utility where dynamic size is important: it can run out of room
  - Adding and removing entries in the middle requires moving array entries, which is slow
  - Initialization may be expensive, especially if many entries are not subsequently used

Fixed Size Can Support Fast Direct Access

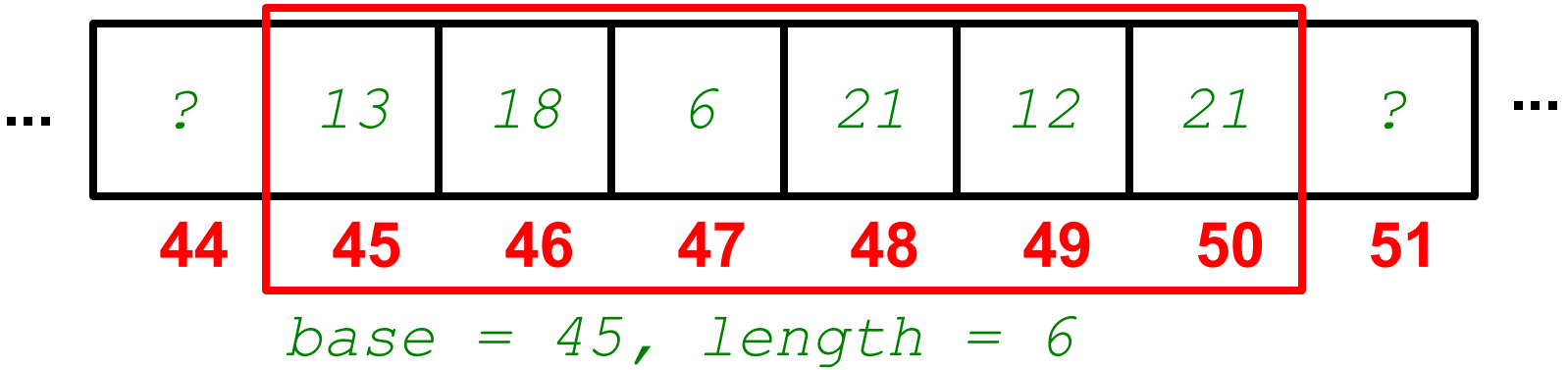
- A Java array is represented in a ***contiguous block*** of memory locations with consecutive memory addresses (IDs), so the memory address of the entry at index `i` can be ***directly calculated*** from the memory address of the first entry, by using simple arithmetic

Example

Client's view of an array (*entries = <13, 18, 6, 21, 12, 21>*):



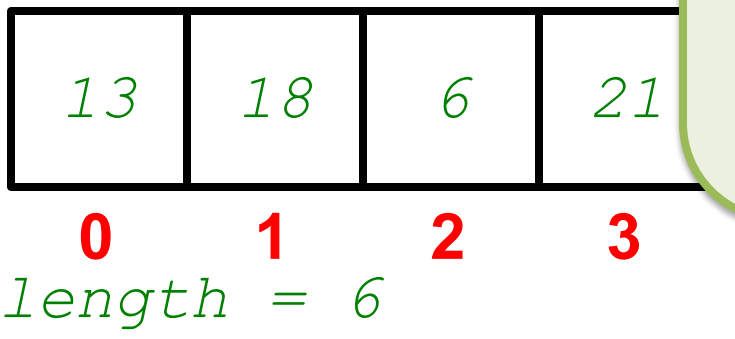
Implementer's view of an array in memory:



Exampl

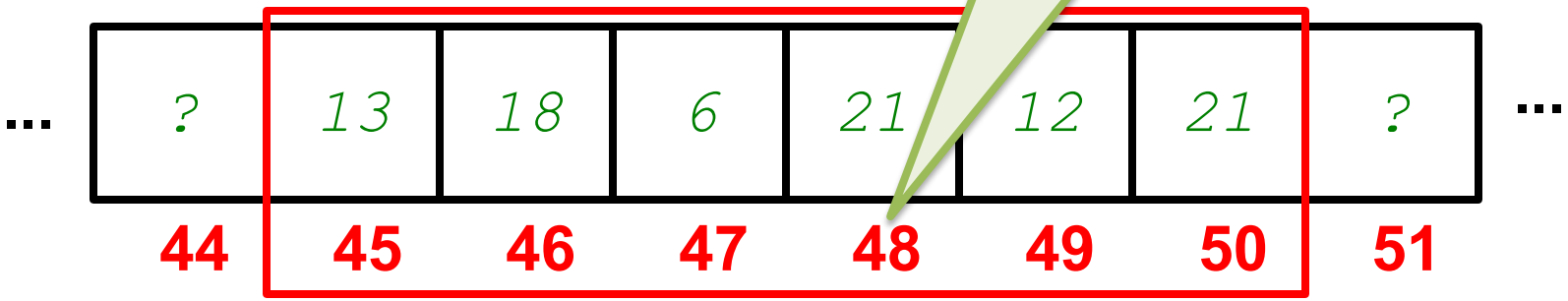
Client's view of an array

(*entries = <13, 18,*



If client wants to access the entry at position *3* of the array, how does implementer compute its memory address/ID?

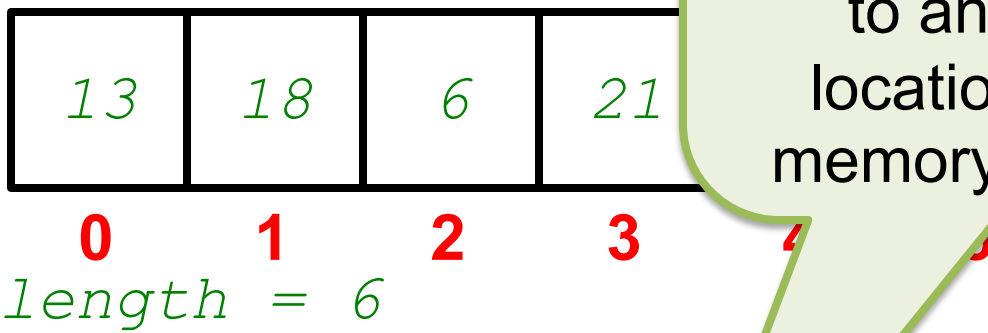
Implementer's view of an array in memory:



Exampl

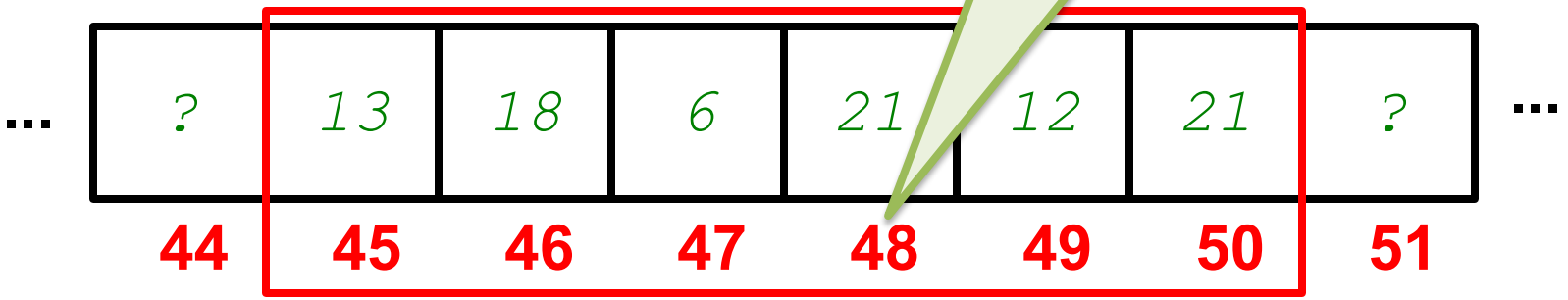
Client's view of an array

(*entries = <13, 18,*



Every modern computer the JVM runs on provides constant-time access to any memory location given the memory address/ID.

Implementer's view of an array[ ] memory:



Notice the Array Mismatches

Collection	Fixed Size?	Dynamic?	Direct Access?	Seq Access?
array	✔		✔	
`Queue`		✔		✔
`Stack`		✔		✔
`Sequence`		✔	✔	
`Set`		✔		
`Map`		✔		
`Sorting-Machine`		✔		

What Can Be Done?

- To represent collections that are ***dynamic*** with ***sequential access***, a different approach is needed: not arrays
  - Note: It is an open problem to represent a `Sequence`, which is ***dynamic*** and offers ***direct access***, in a way that is efficient in both execution time of access operations (i.e., constant time) and memory "footprint" (i.e., constant factor overhead)

Dynamic Can Support Fast Sequential Access

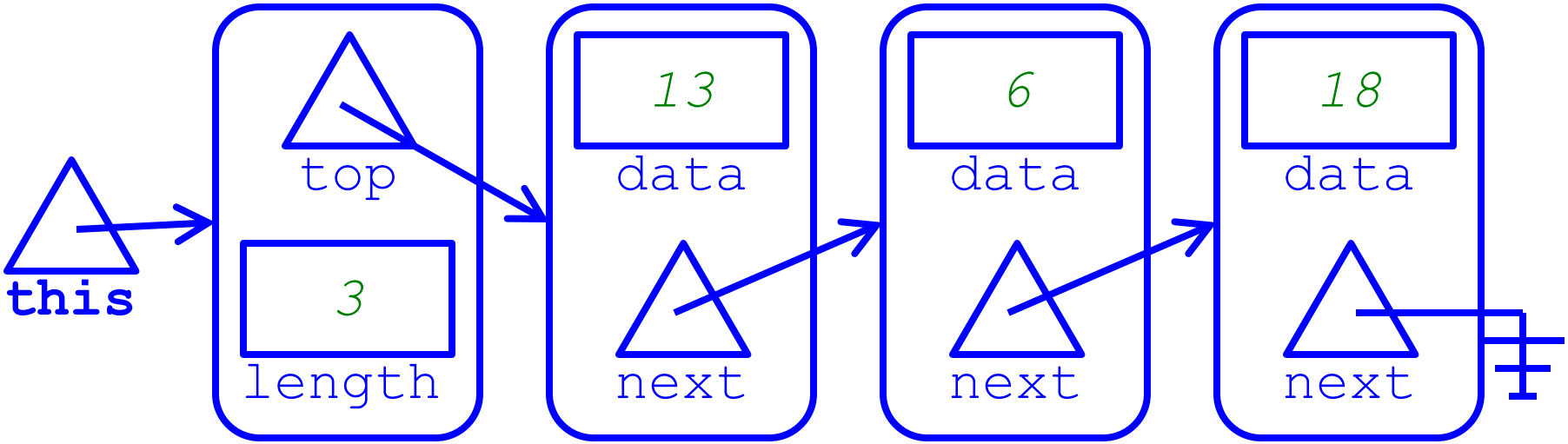
- If we want a ***dynamic*** collection, then we should give up on storing all entries of the collection in contiguous memory locations
- If we want fast ***sequential access***, then we should give up on fast direct access
  - Instead, for every entry in the collection, wherever it is in memory, simply keep a ***reference*** to (i.e., memory location of) the "next" entry

Example: `Stack2`

Client's view of a `Stack`:

this = *<13, 6, 18>*

Implementer's view of a `Stack2`:



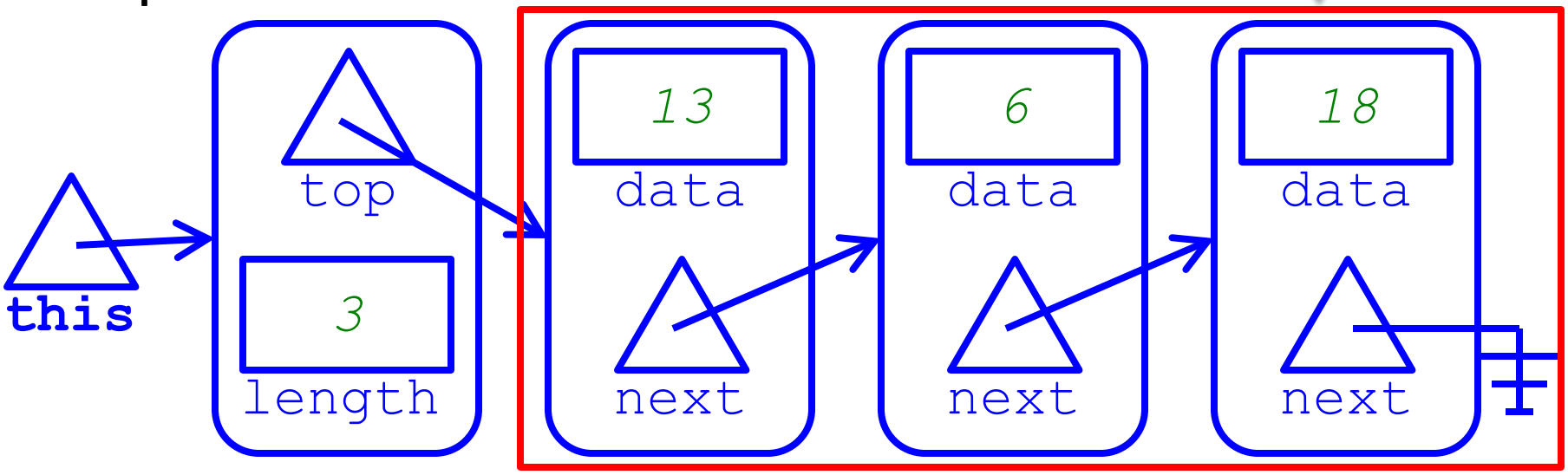
Example:

Client's view of a `Stack`:

this = <13, 6, 18>

This is called a ***singly-linked-list*** data structure.

Implementer's view of a `Stack2`:



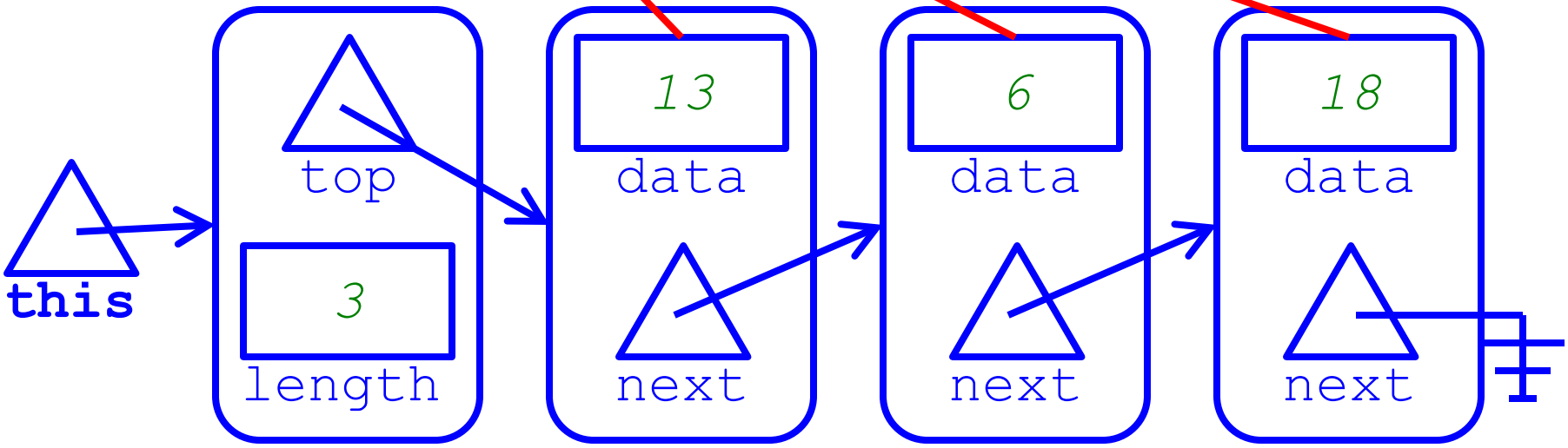
Example:

The abstraction function (correspondence) ...

Client's view of a `Stack`:

this = <13, 6, 18>

Implementer's view of a `Stack2`:



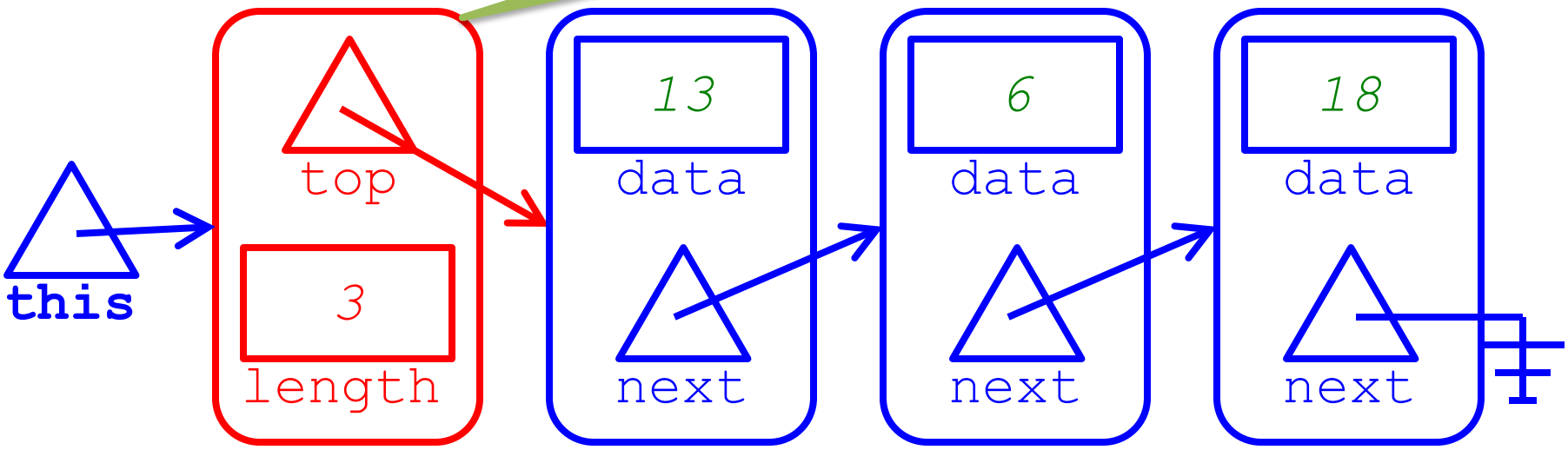
Example

Client's view of a `Stack`

this = <13, 6, ...

The ***instance variables (fields)*** of the data representation for `Stack2` are shown here.

Implementer's view of a `Stack2`:



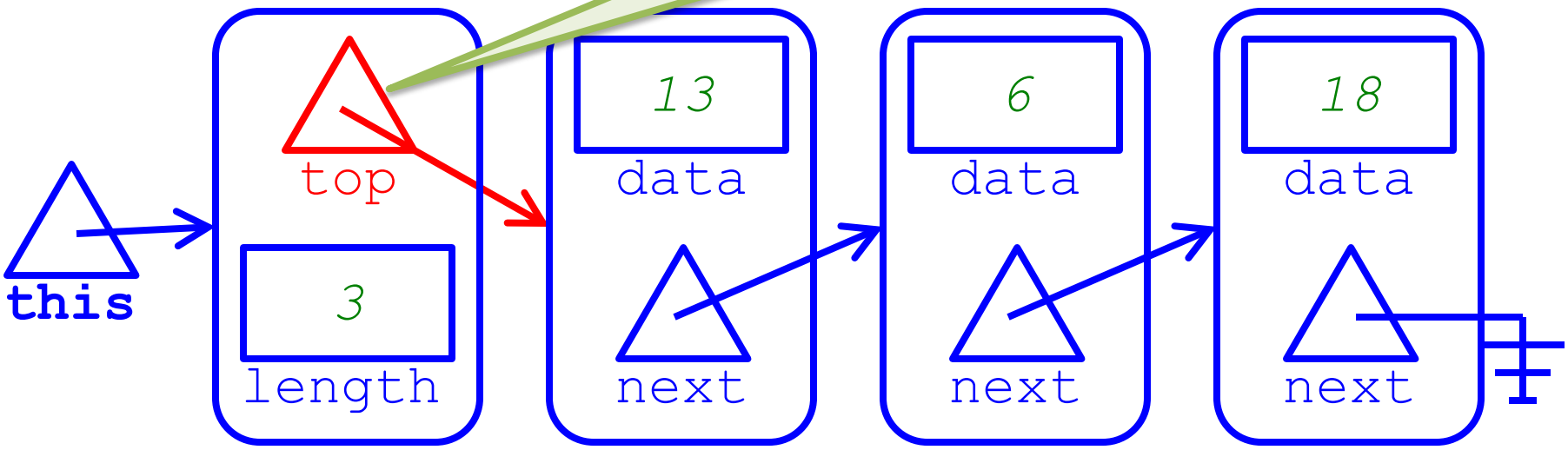
Example

Client's view of a `Stack`

this = <13, 6, 18>

The `Stack` methods only require access to the first entry, i.e., the top, so we keep a reference to its node.

Implementer's view of `LinkedStack2`:



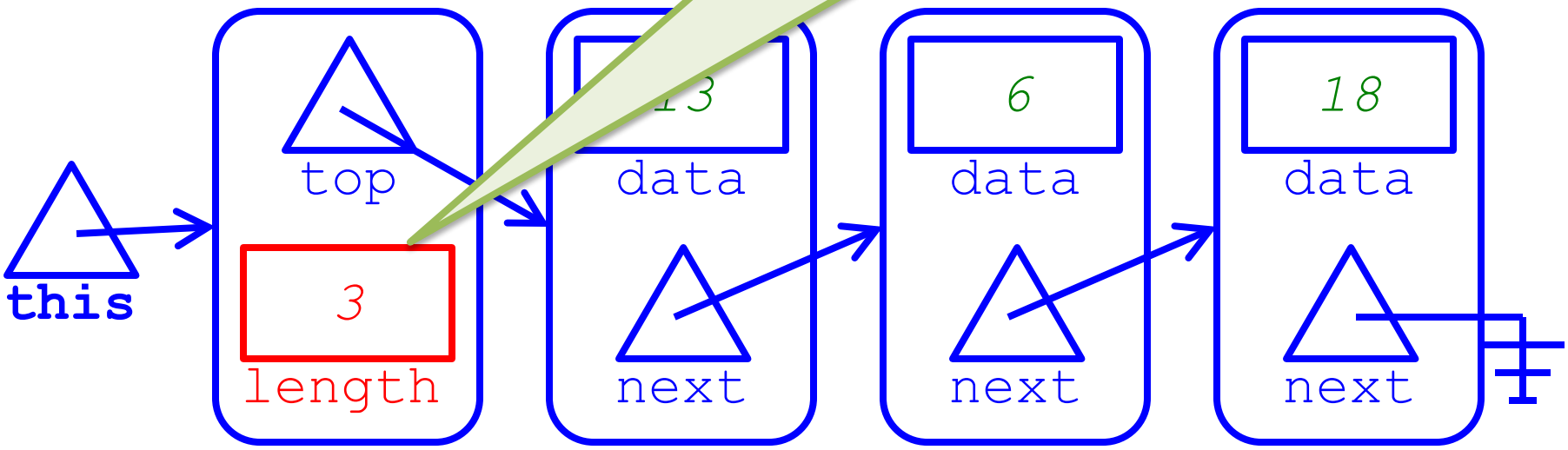
Exampl

Client's view of a Sta

this = <13, 6,

Implementer's view of a ck2:

The Stack methods include length, so we keep this direct count of the number of nodes in the linked list.



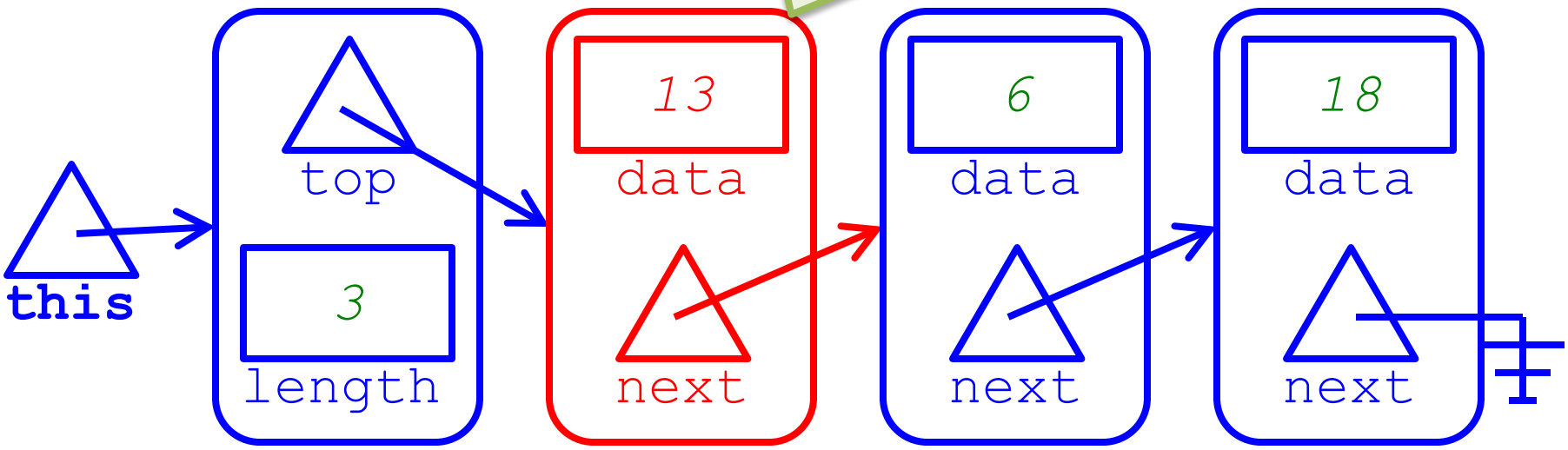
Exam

Client's view of a S

this = <13,

Each of these objects (a pair of variables) is called a ***node*** in this ***singly-linked-list data structure***: a variable of type `T`, and a reference to the "next" node.

Implementer's view of a k2:



Declaration of `Node` Class

- A `Node` class is declared as a ***nested class*** inside the kernel implementation that uses it in a data representation (e.g., `Stack2`, `Queue2`, and similar classes)

```
private final class Node {
  private T data;
  private Node next;
}
```

Declaratio

This declaration is recursive, and may seem circular!
One of the instance variables of a `Node` is of type `Node`.
How can this work?

- A `Node` class is ***class*** inside the … that uses it in a data … sentation (e.g., `Stack2`, `Queue2`, an… similar classes)

```
private final class Node {
   private T data;
   private Node next;
}
```

ode Class

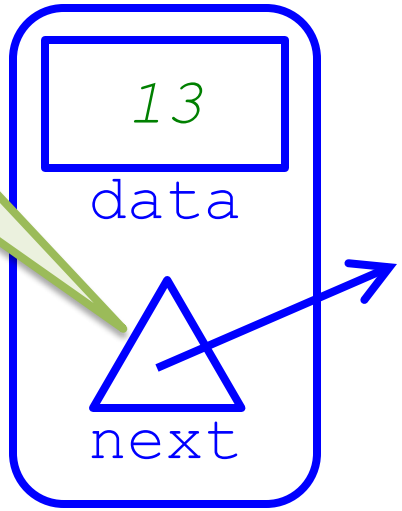
as a ***nested***

nplementation

that uses it in a epresentation (e.g.,

Stack2, Queue2, a imilar classes)

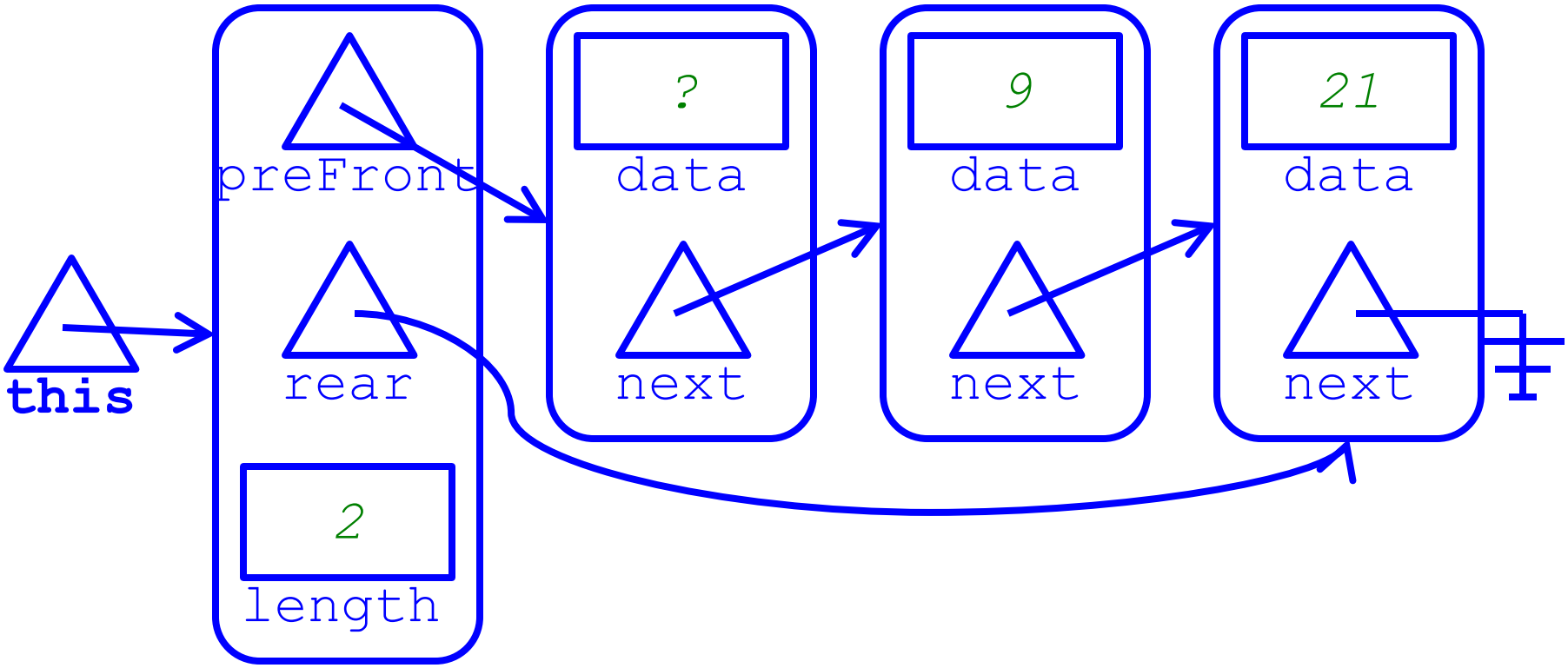
```
private final class Nod
   private T data;
   private Node next;
}
```



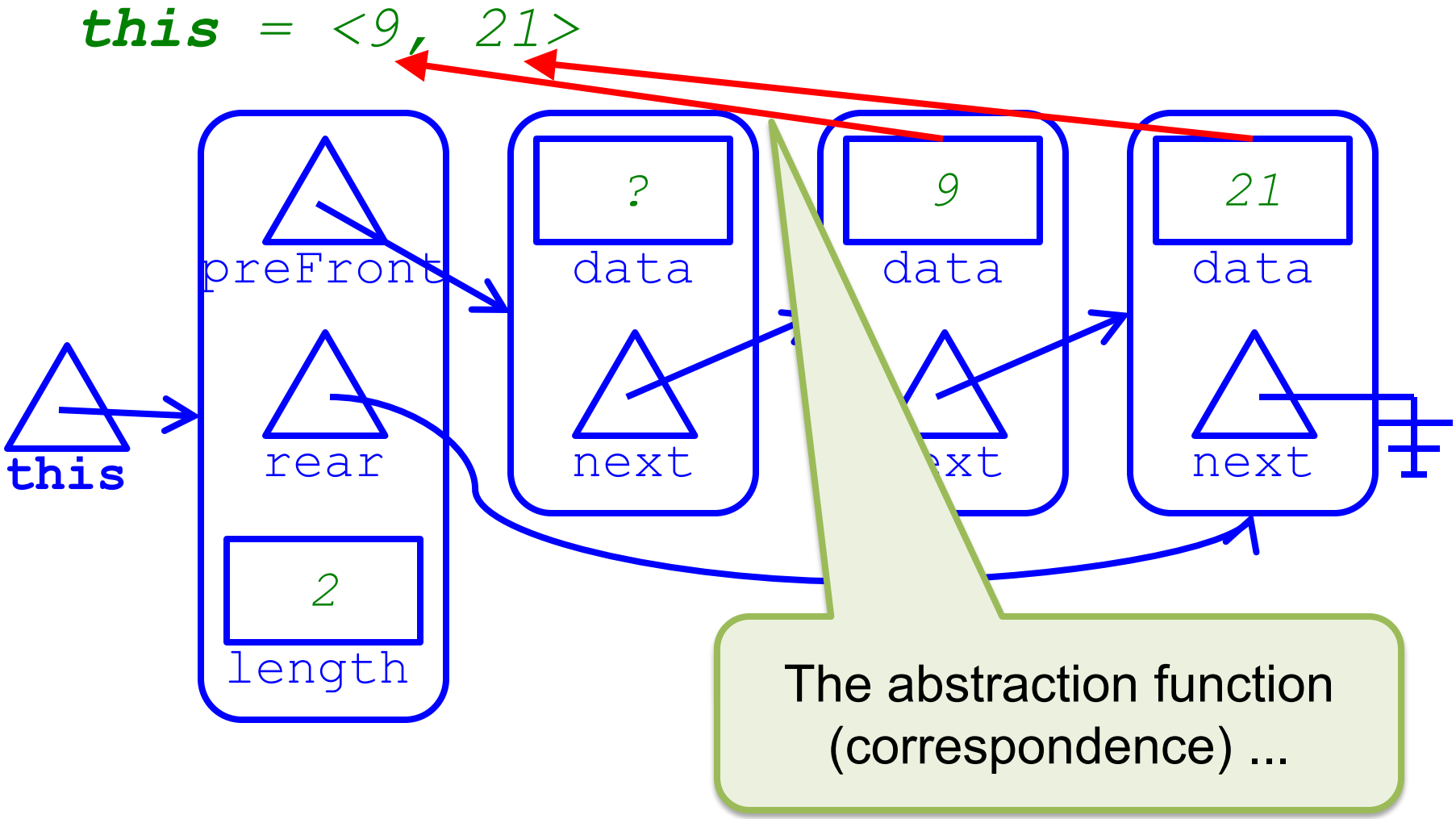
It works because the instance variable `next` is a ***reference variable***, hence is a ***reference*** to a `Node` object rather than a "nested" object.

Example: `Queue2`

this = *<9, 21>*

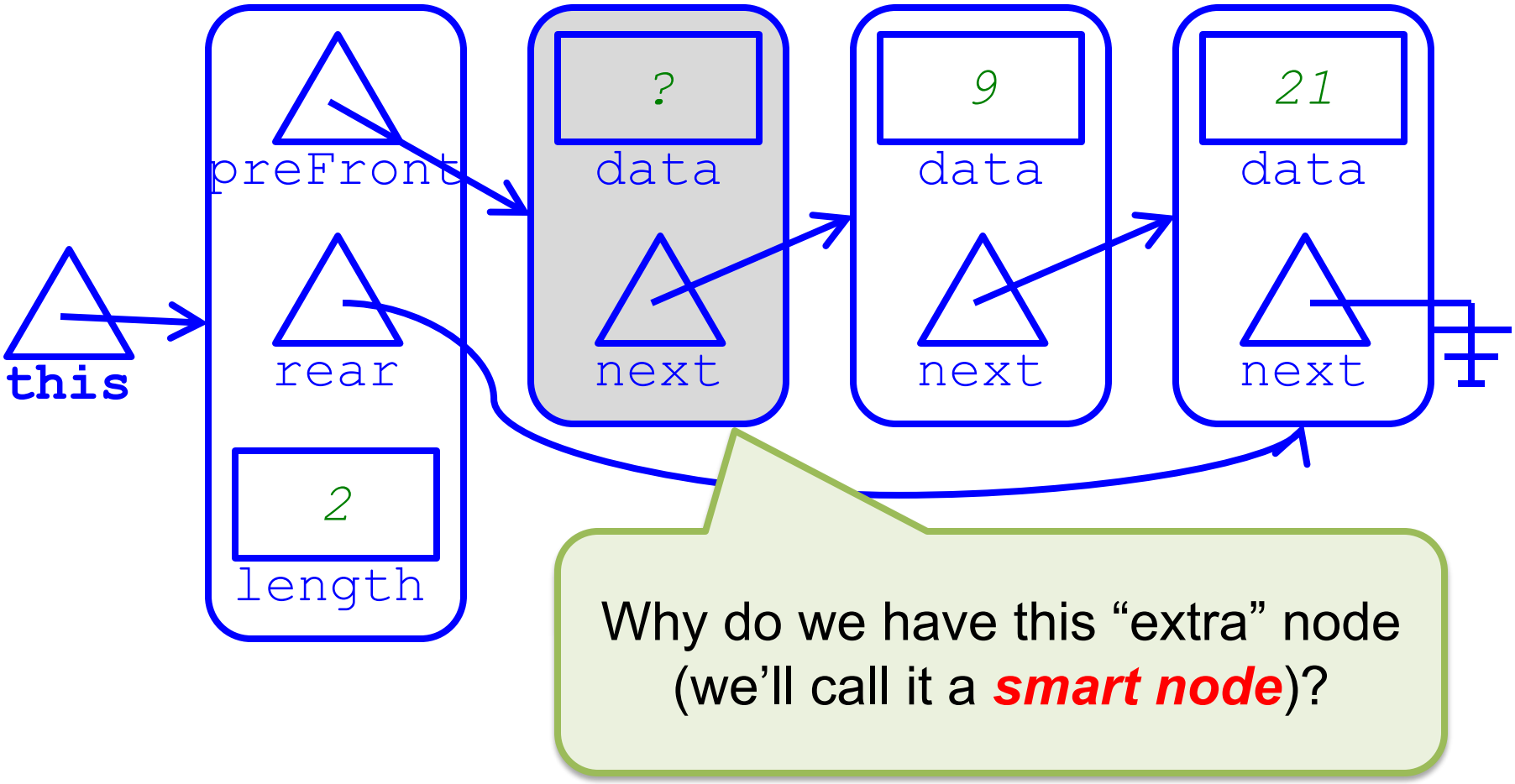


Example: Queue2



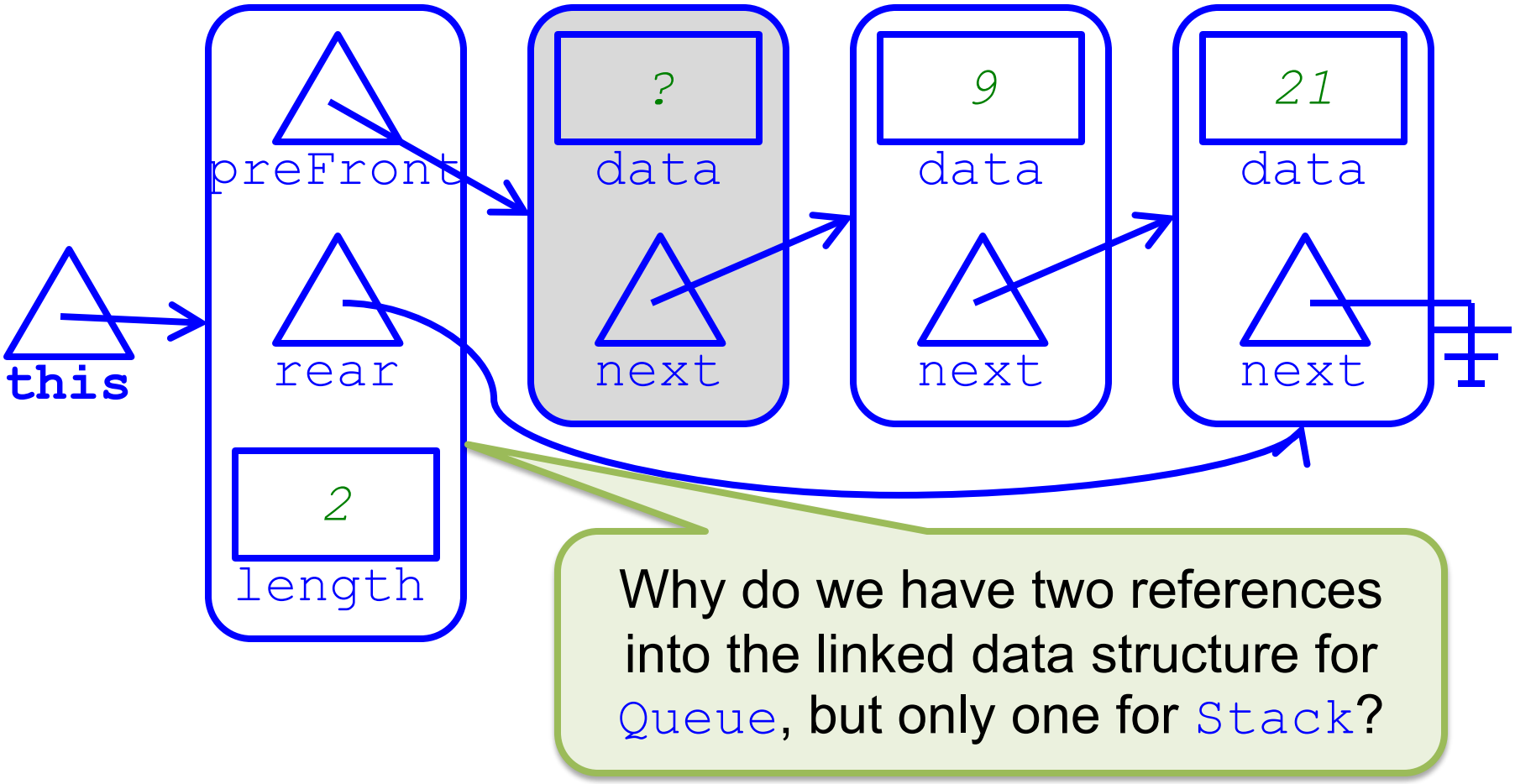
Example: `Queue2`

this = <9, 21>



Example: `Queue2`

this = <9, 21>

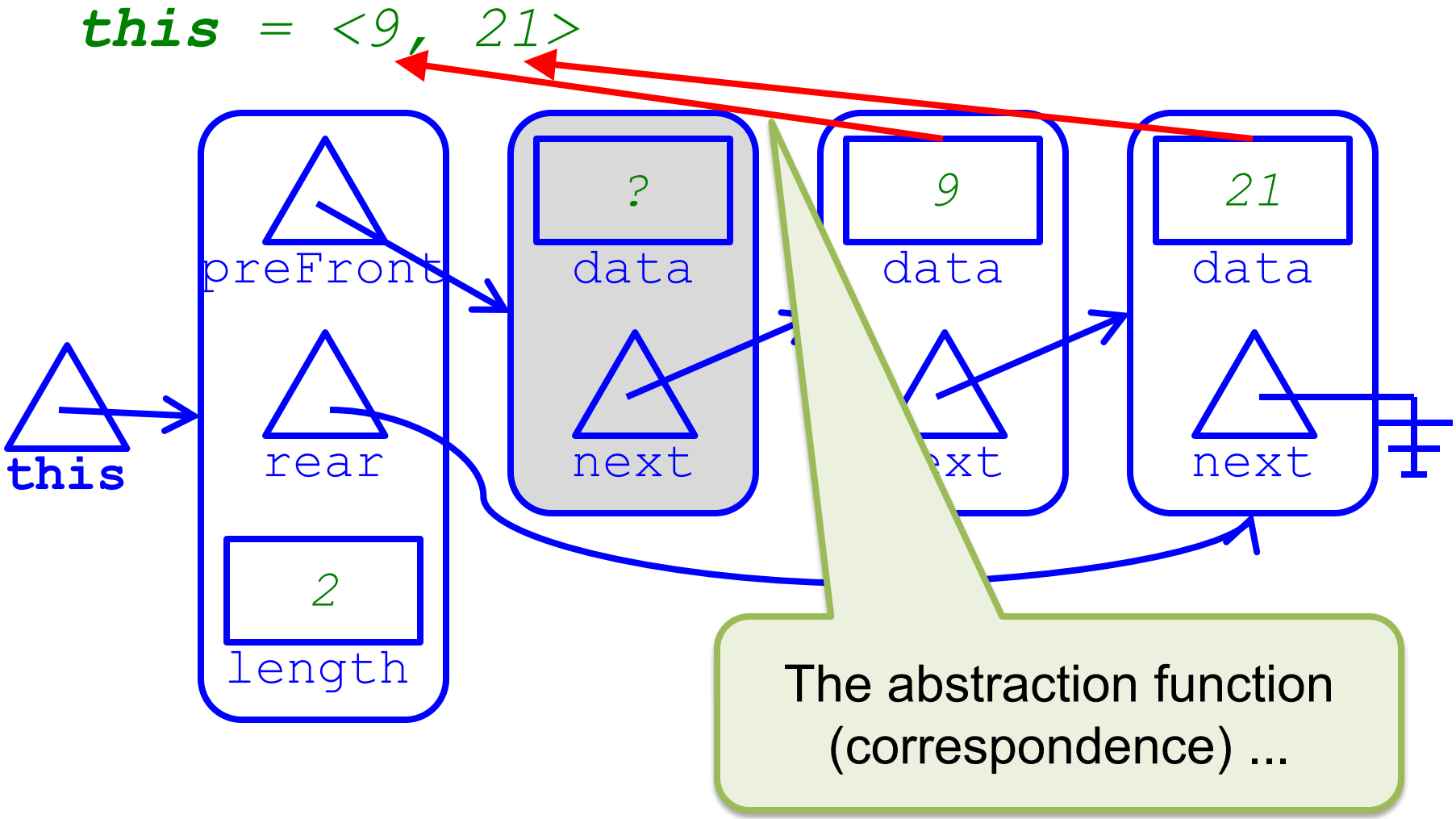


Why do we have two references into the linked data structure for `Queue`, but only one for `Stack`?

Example: `dequeue` for `Queue2`

```
public final T dequeue() {
   Node p = this.preFront;
   Node q = p.next;
   T result = q.data;
   this.preFront = q;
   this.length--;
   return result;
}
```

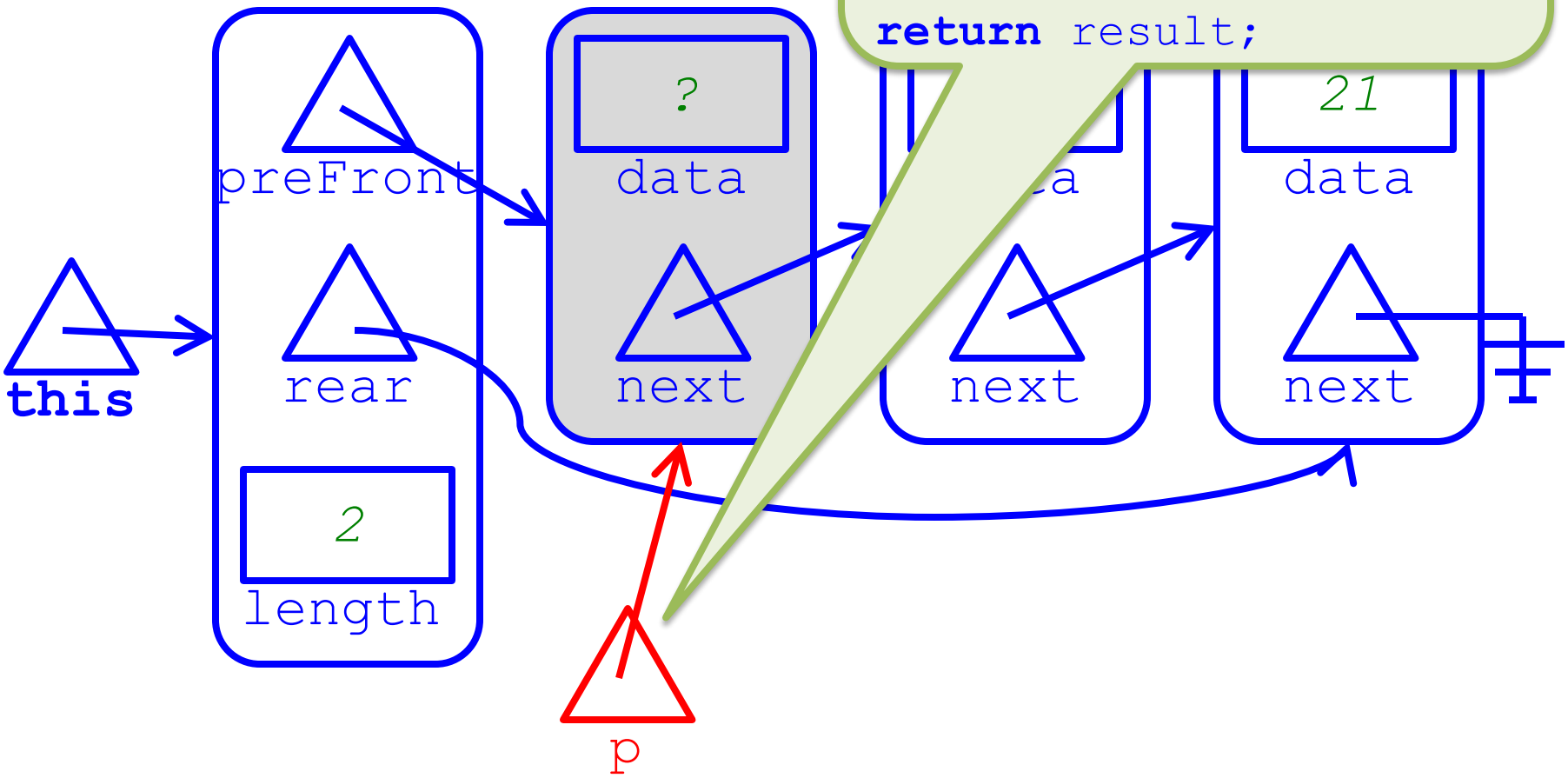
Example: `dequeue` for `Queue2`



Example: dequeu

this = <9, 21>

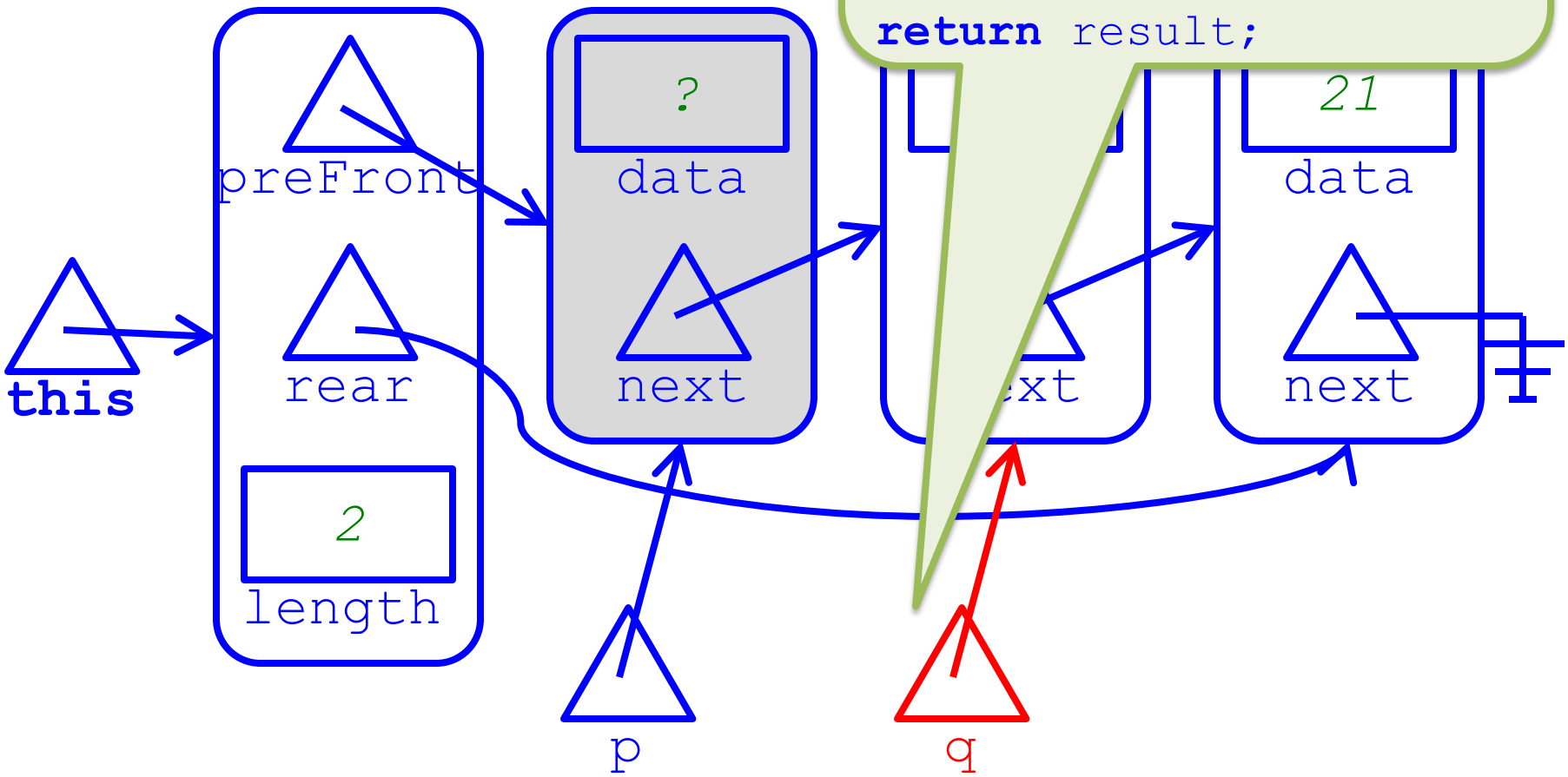
```
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```



Example: dequeu

this = <9, 21>

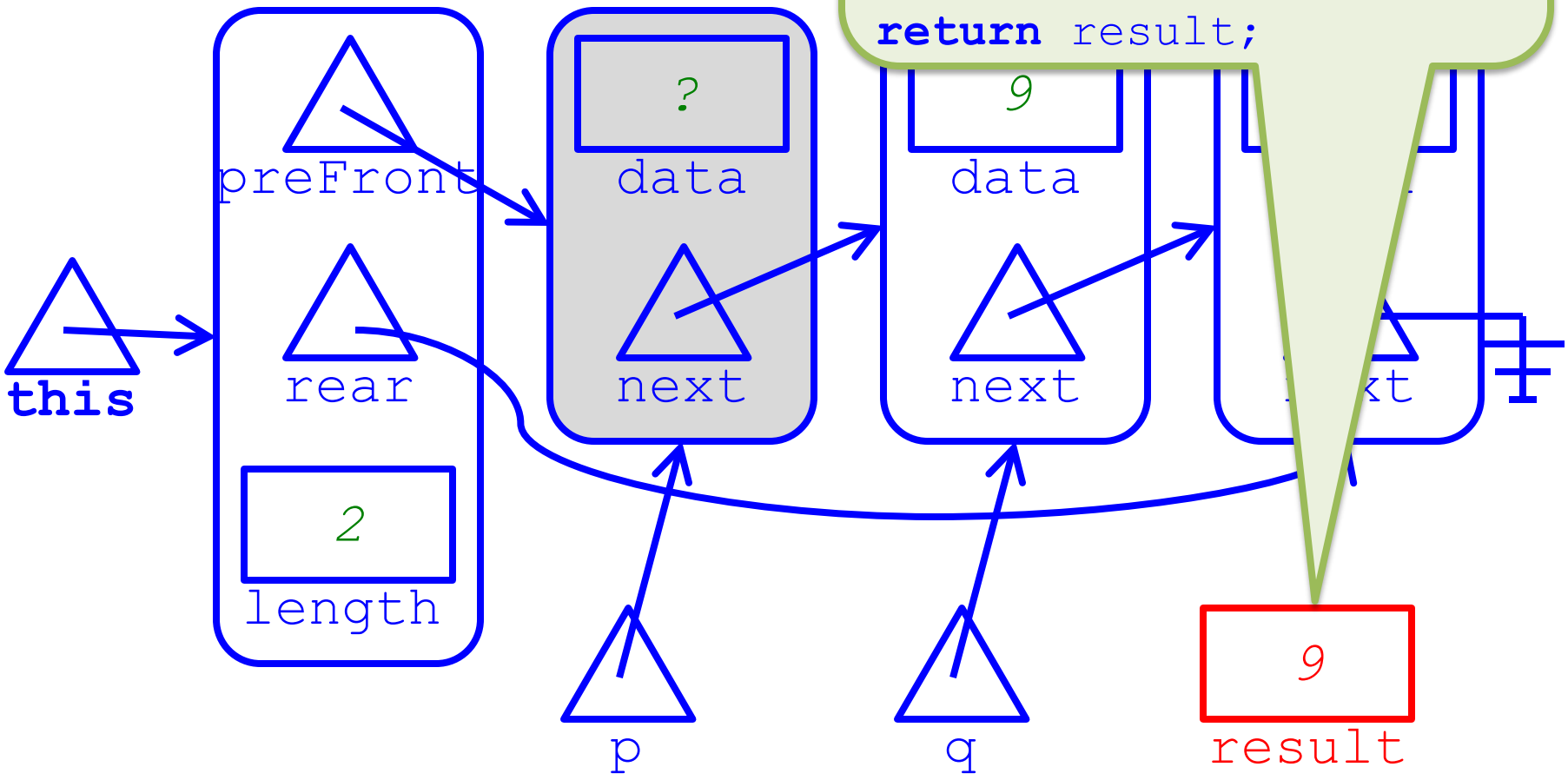
```
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```



Example: dequeu

this = <9, 21>

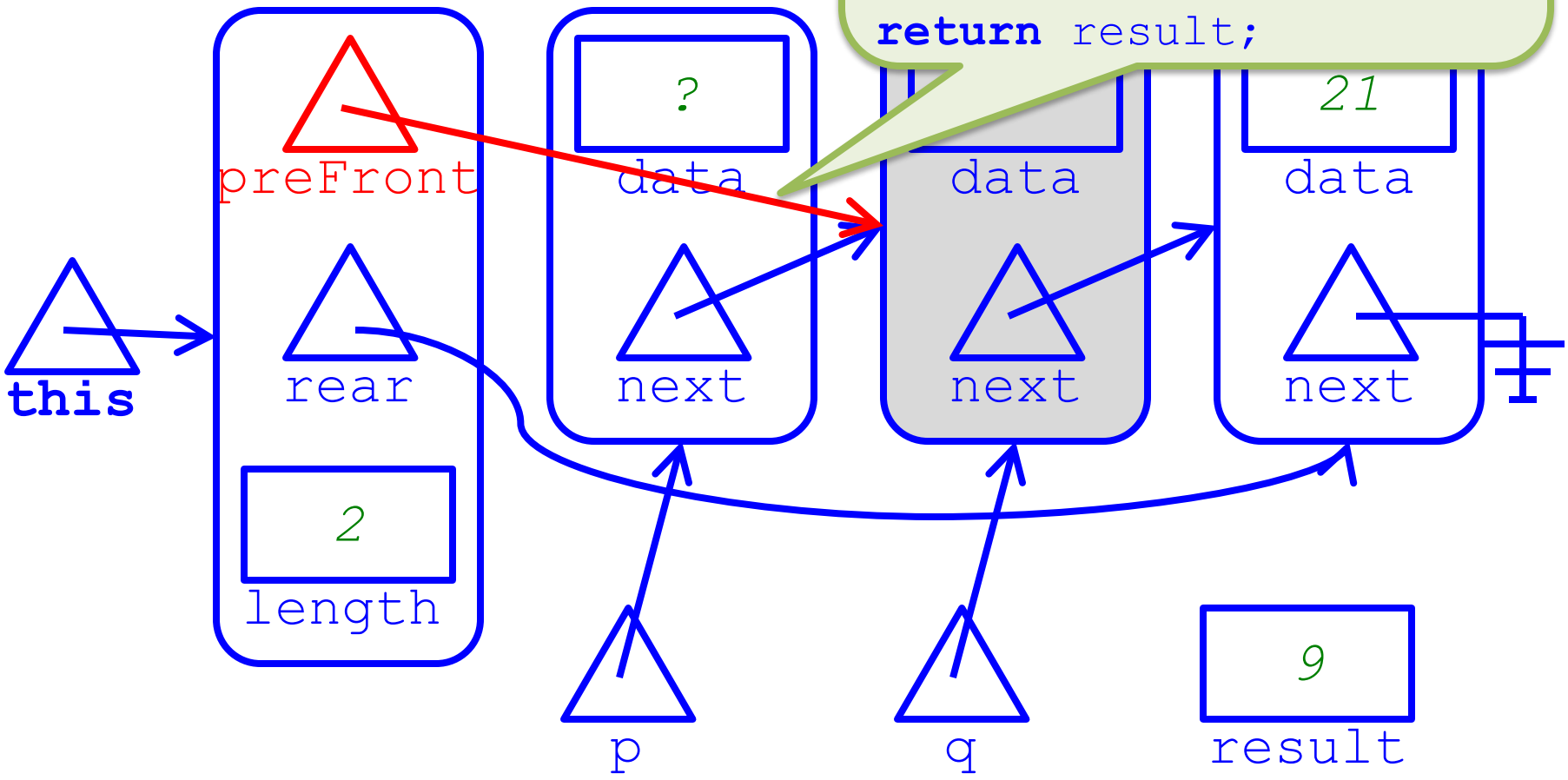
```
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```



Example: dequeu

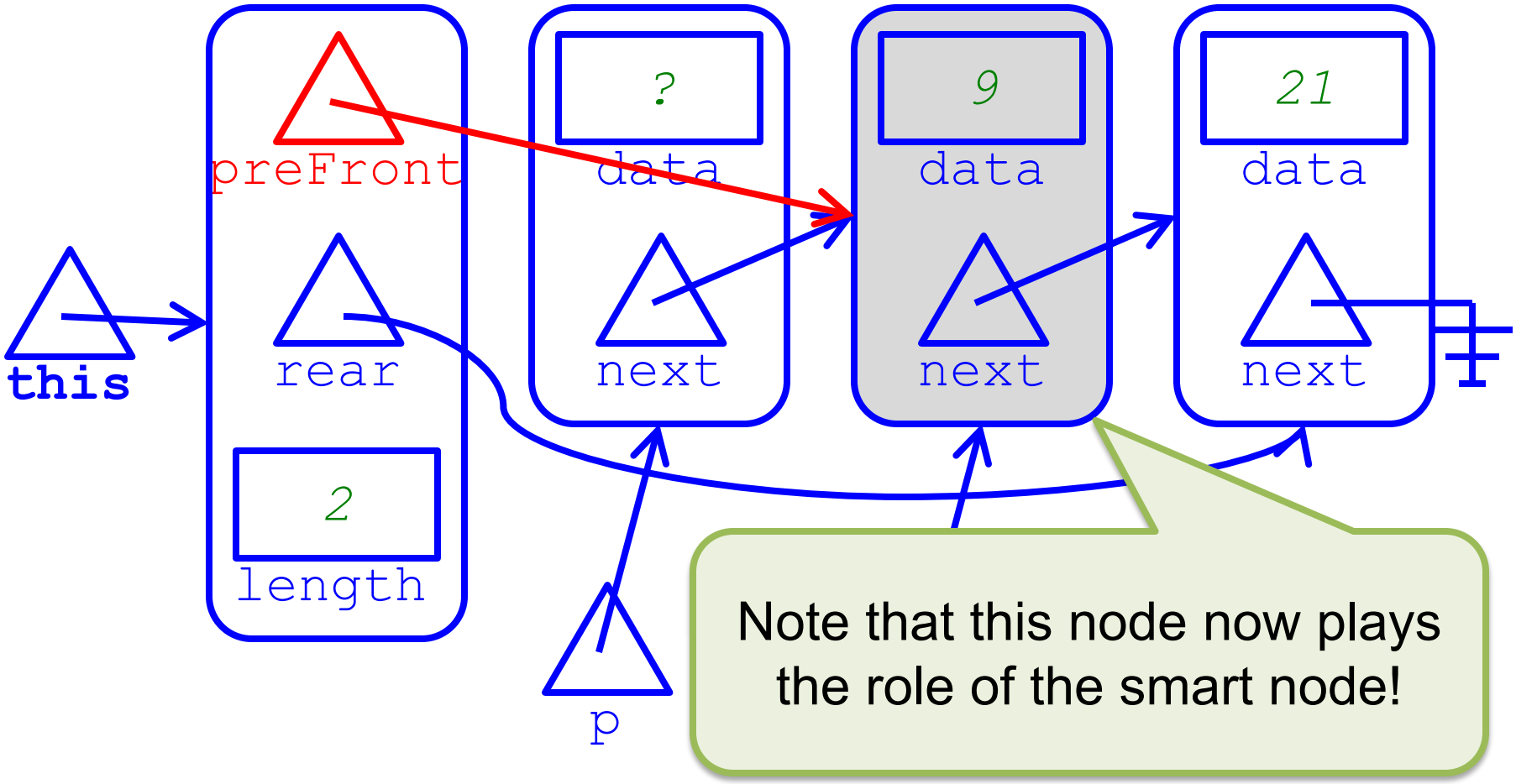
this = <9, 21>

```
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```



Example: `dequeue` for `Queue2`

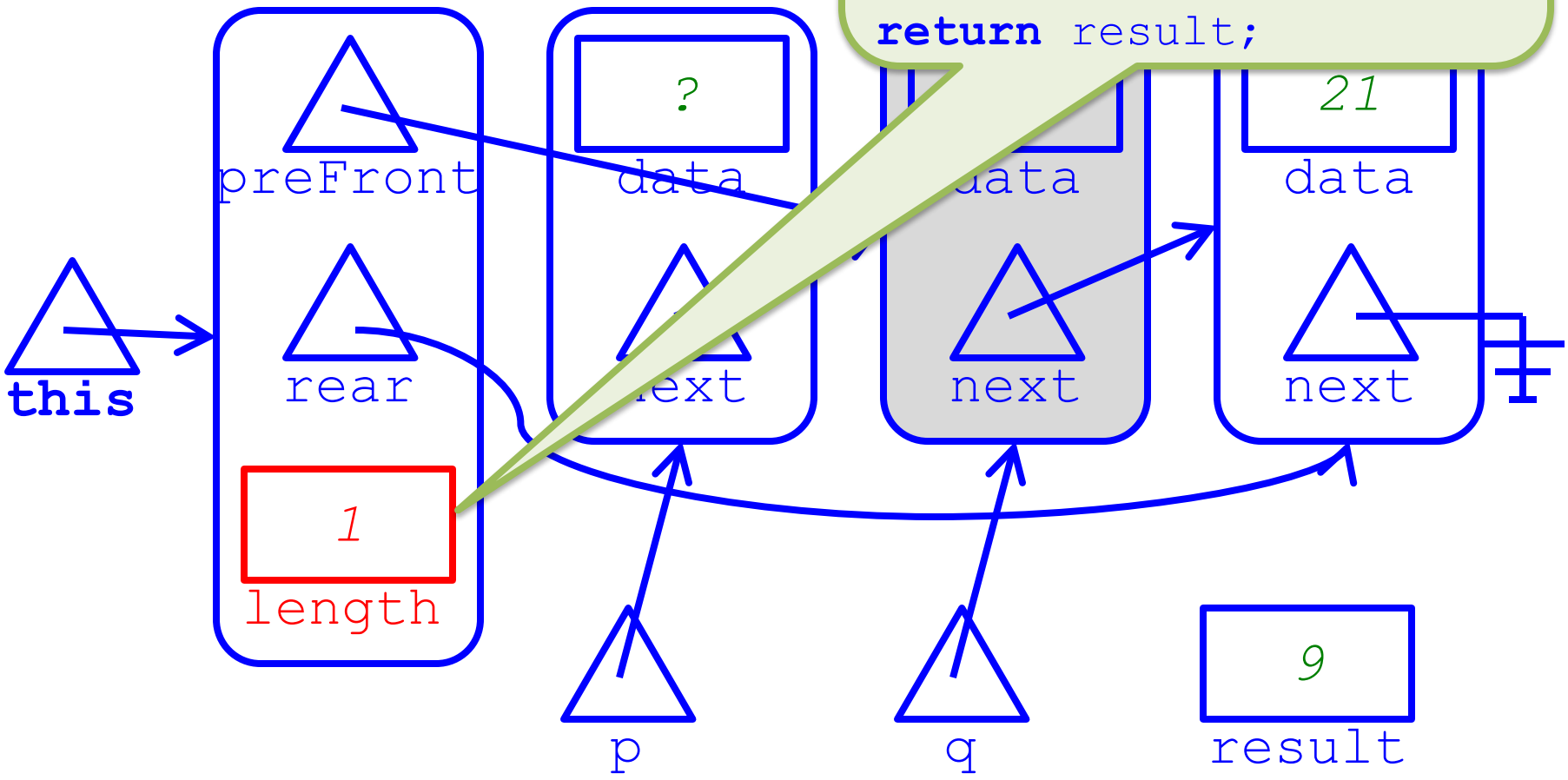
this = <9, 21>



Example: dequeu

this = <9, 21>

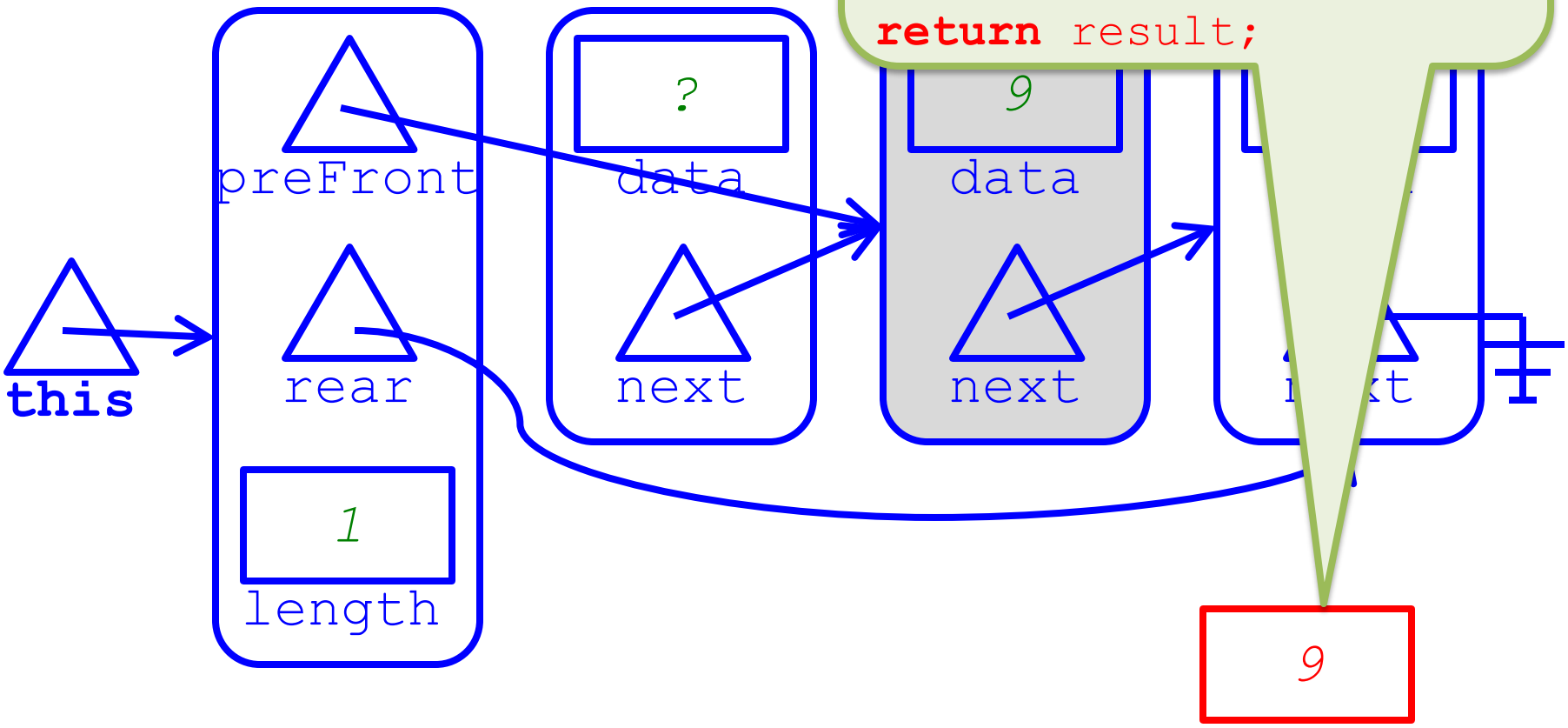
```
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```



Example: dequeu

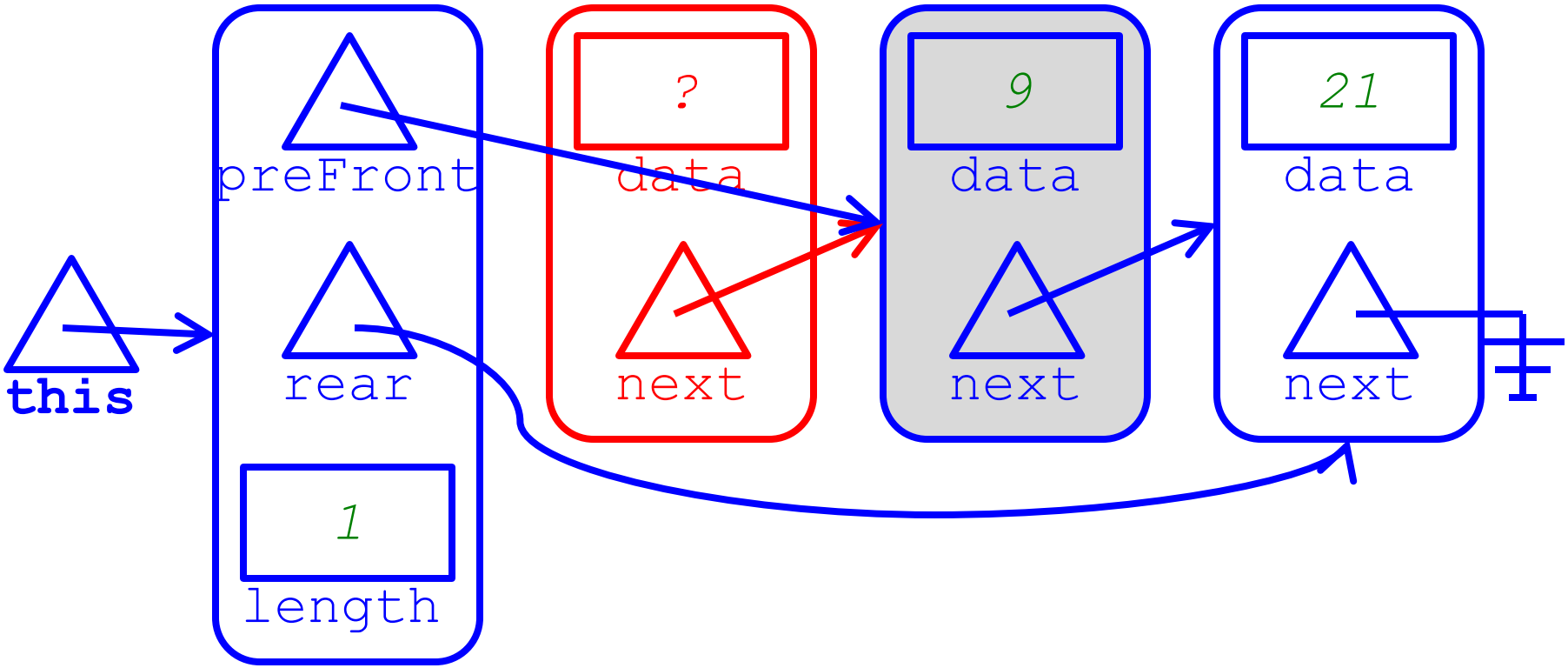
this = <21>

```
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```



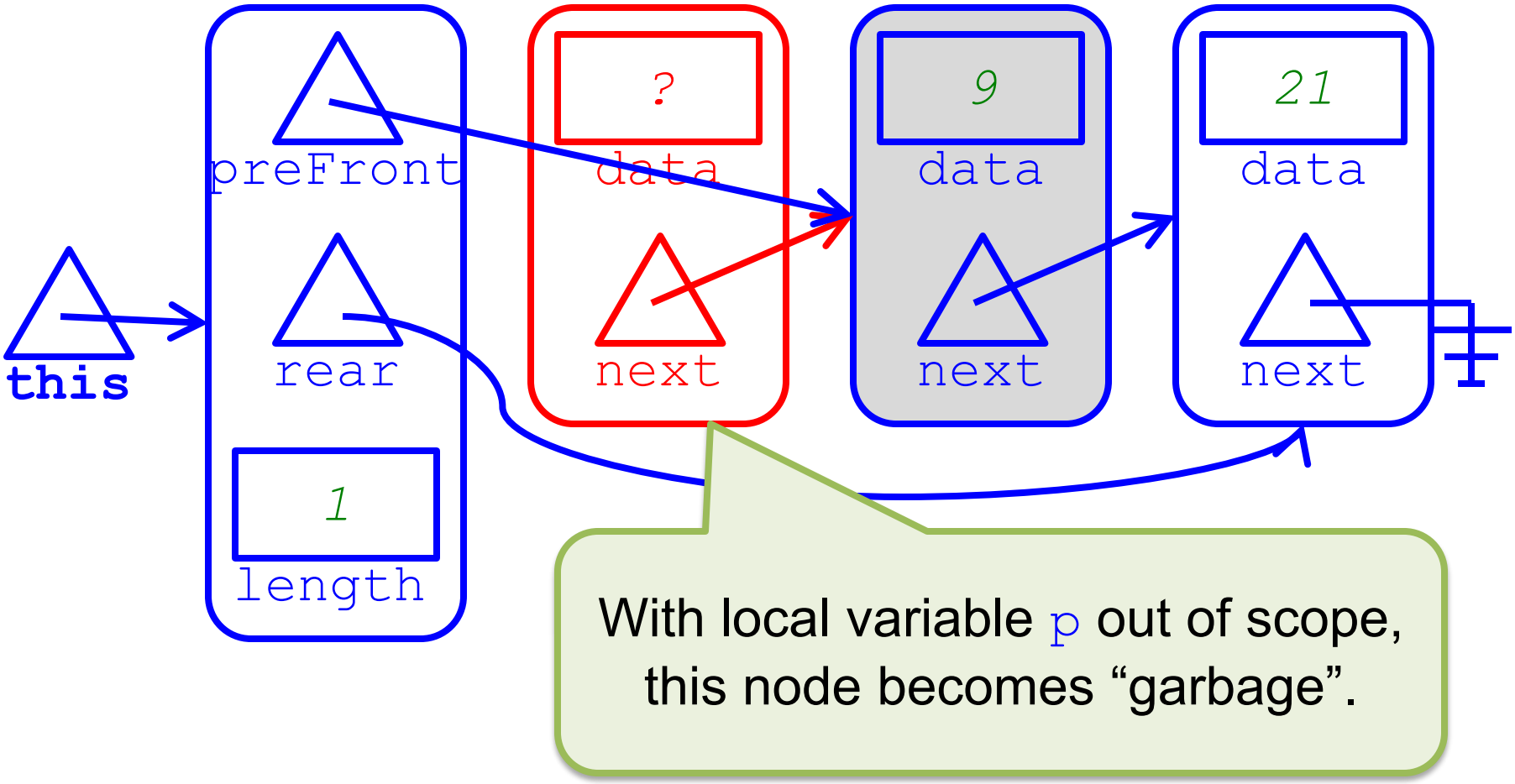
Example: `dequeue` for `Queue2`

this = <21>

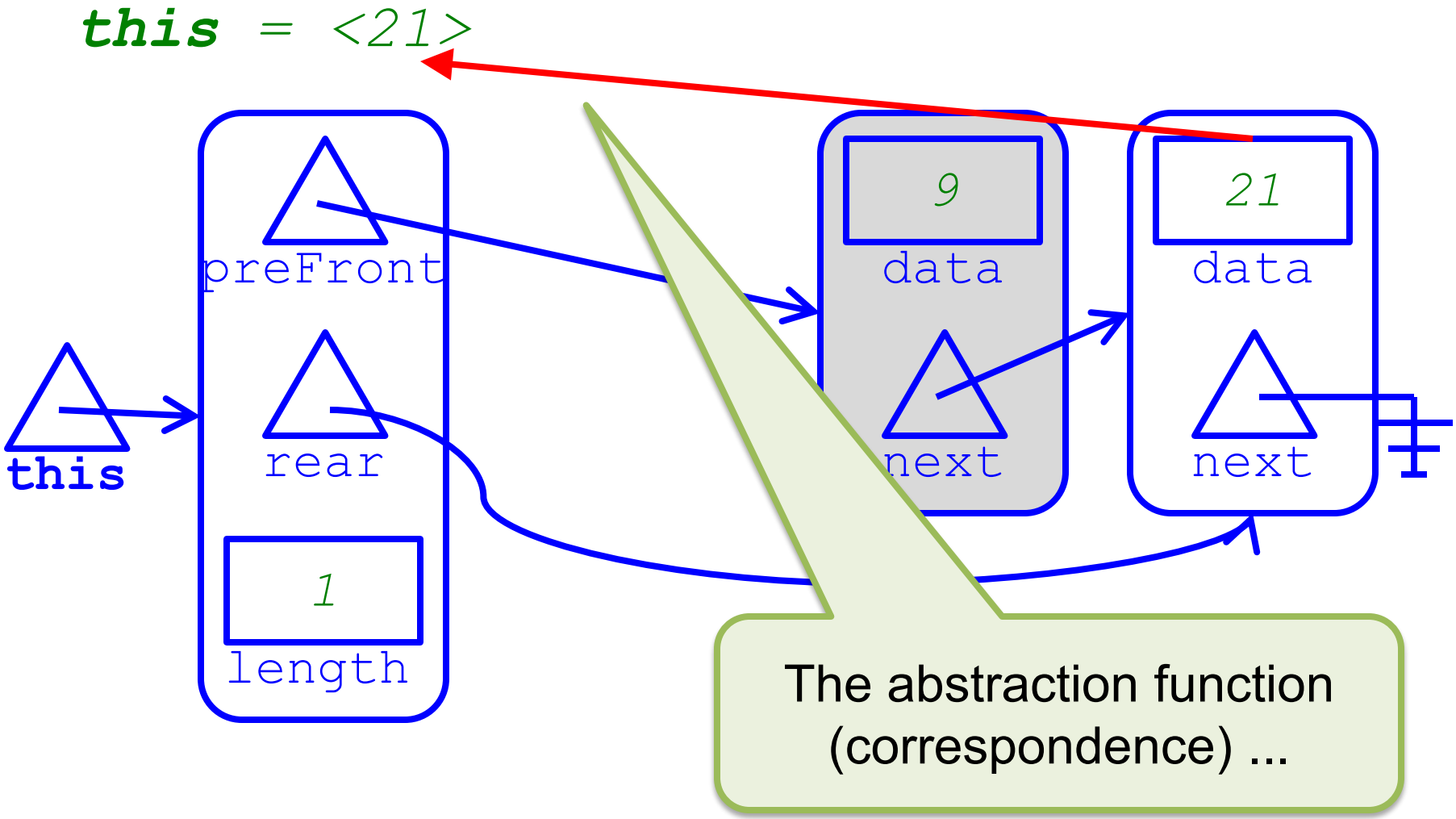


Example: `dequeue` for `Queue2`

this = <21>



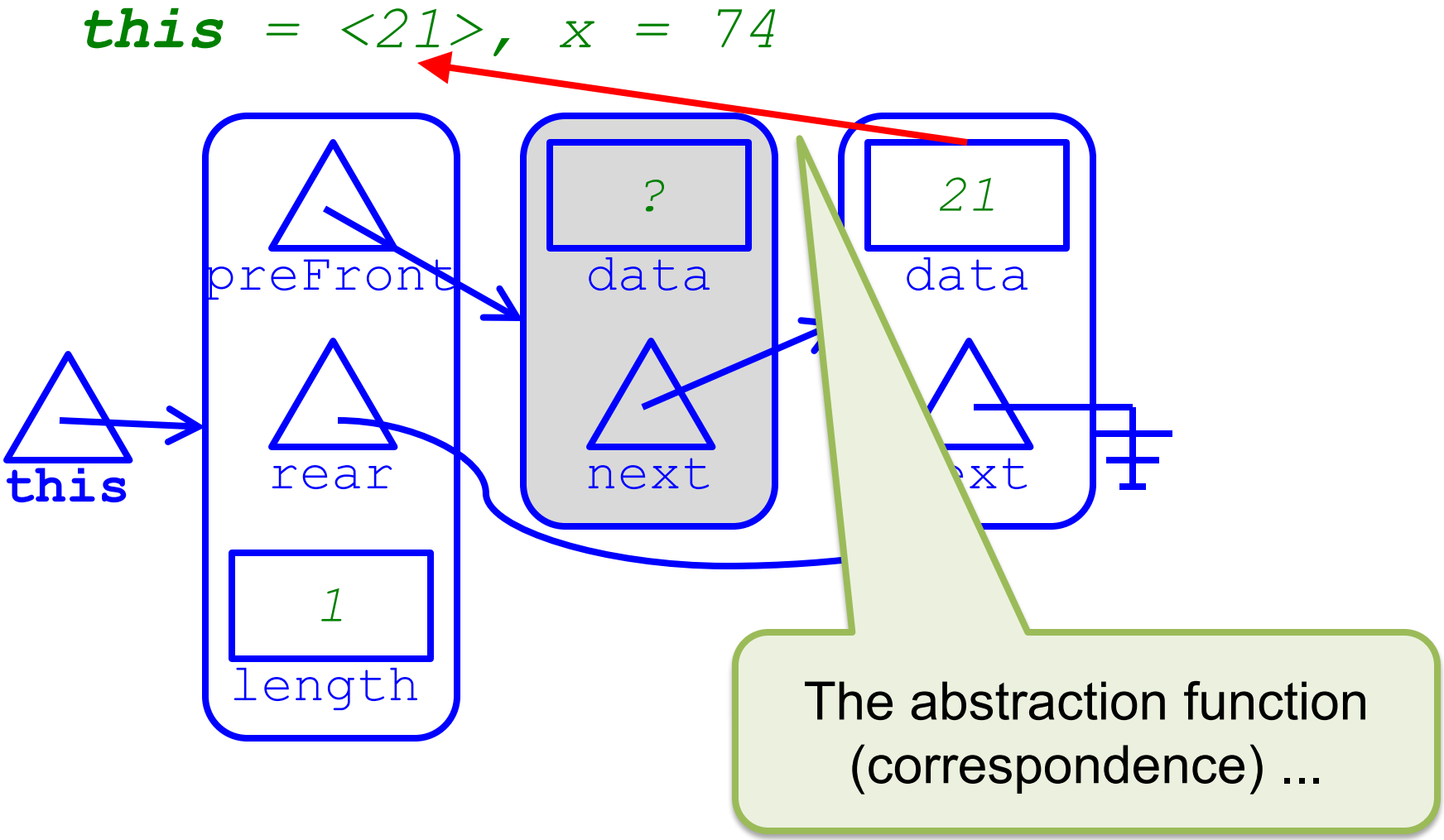
Example: `dequeue` for `Queue2`



Example: `enqueue` for `Queue2`

```
public final void enqueue(T x) {
   Node p = new Node();
   Node q = this.rear;
   p.data = x;
   p.next = null;
   q.next = p;
   this.rear = p;
   this.length++;
}
```

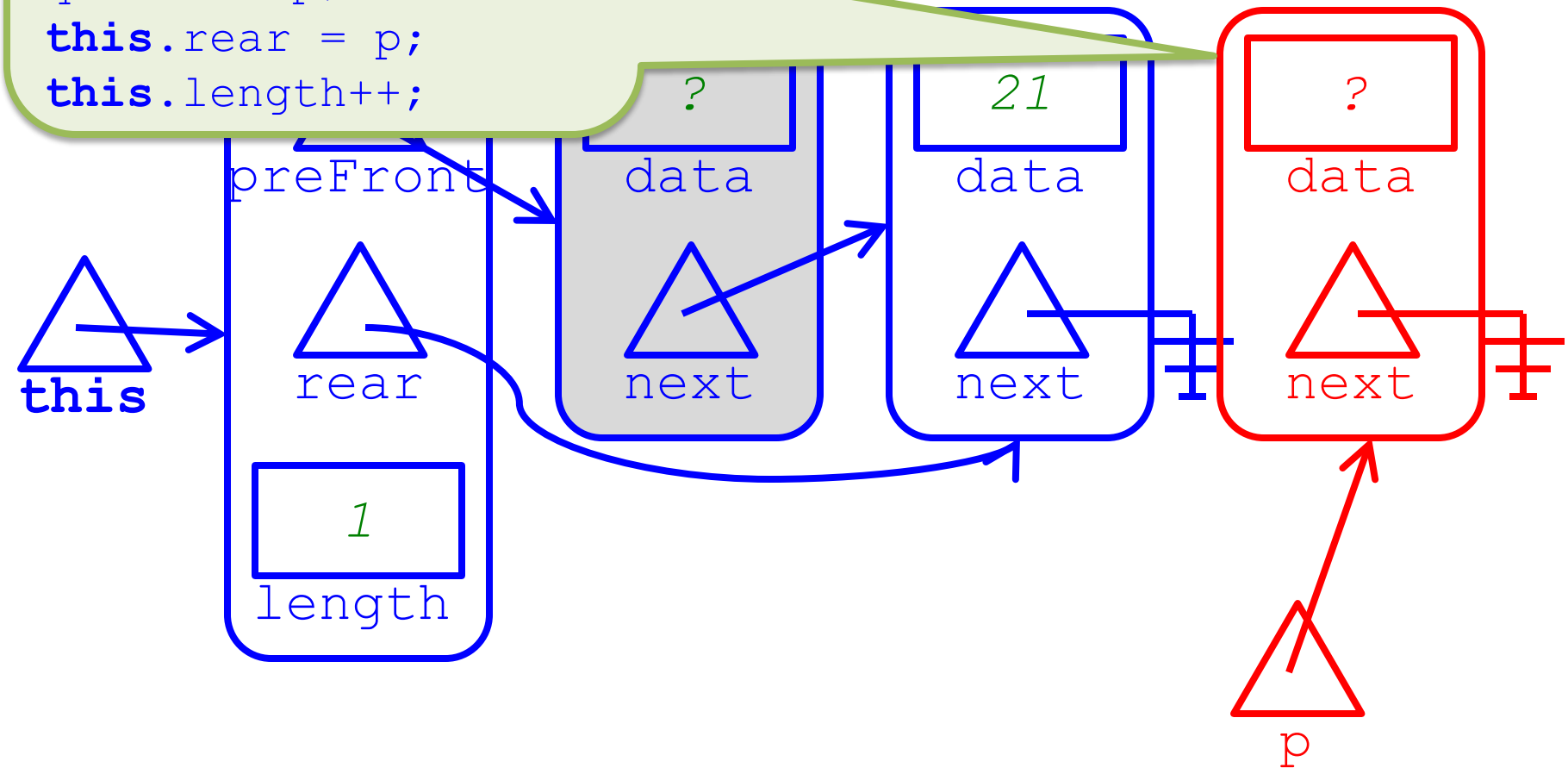
Example: `enqueue` for `Queue2`



queue for Queue2

```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```

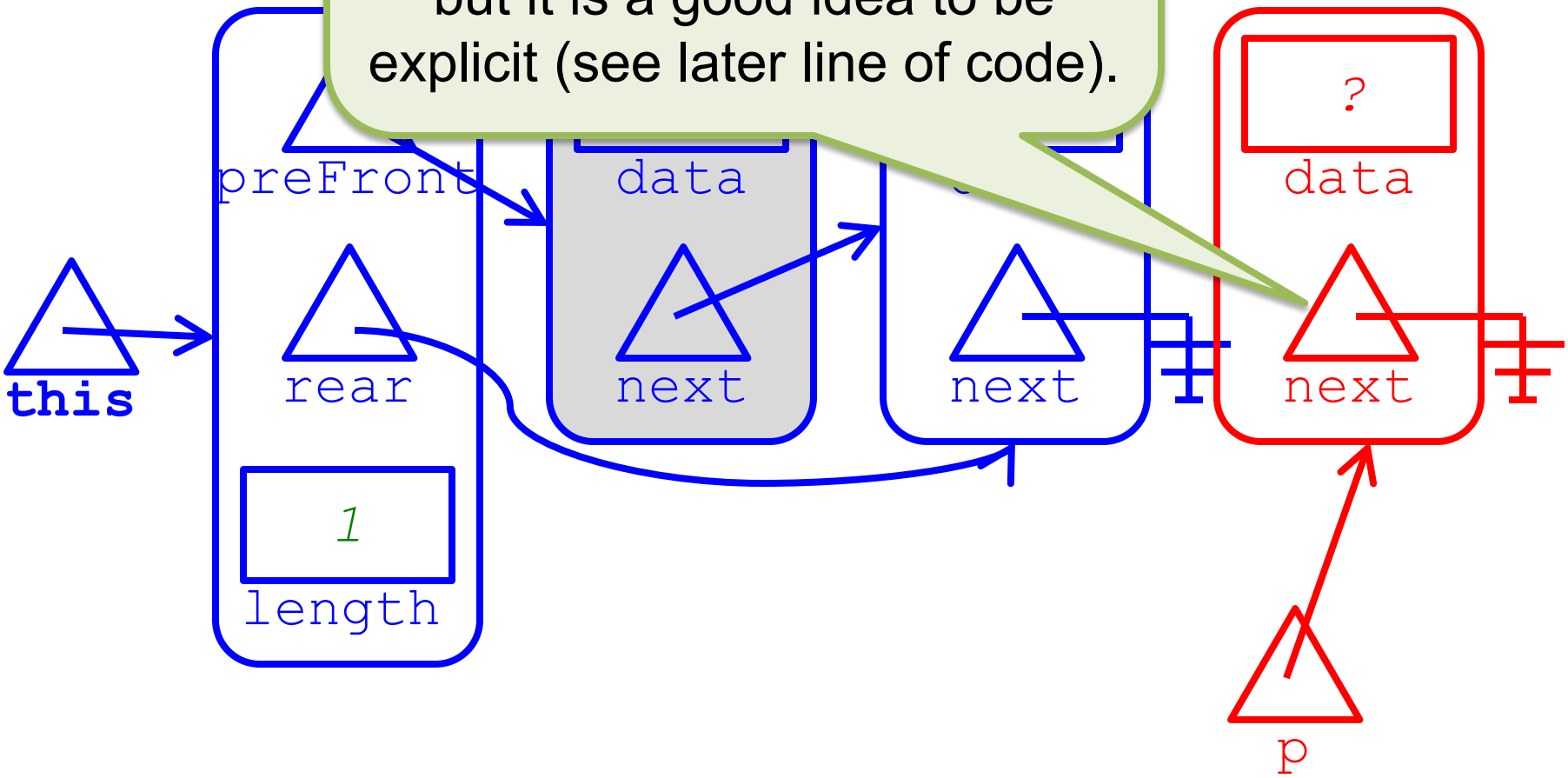
x = 74



Exam ueue2

this =

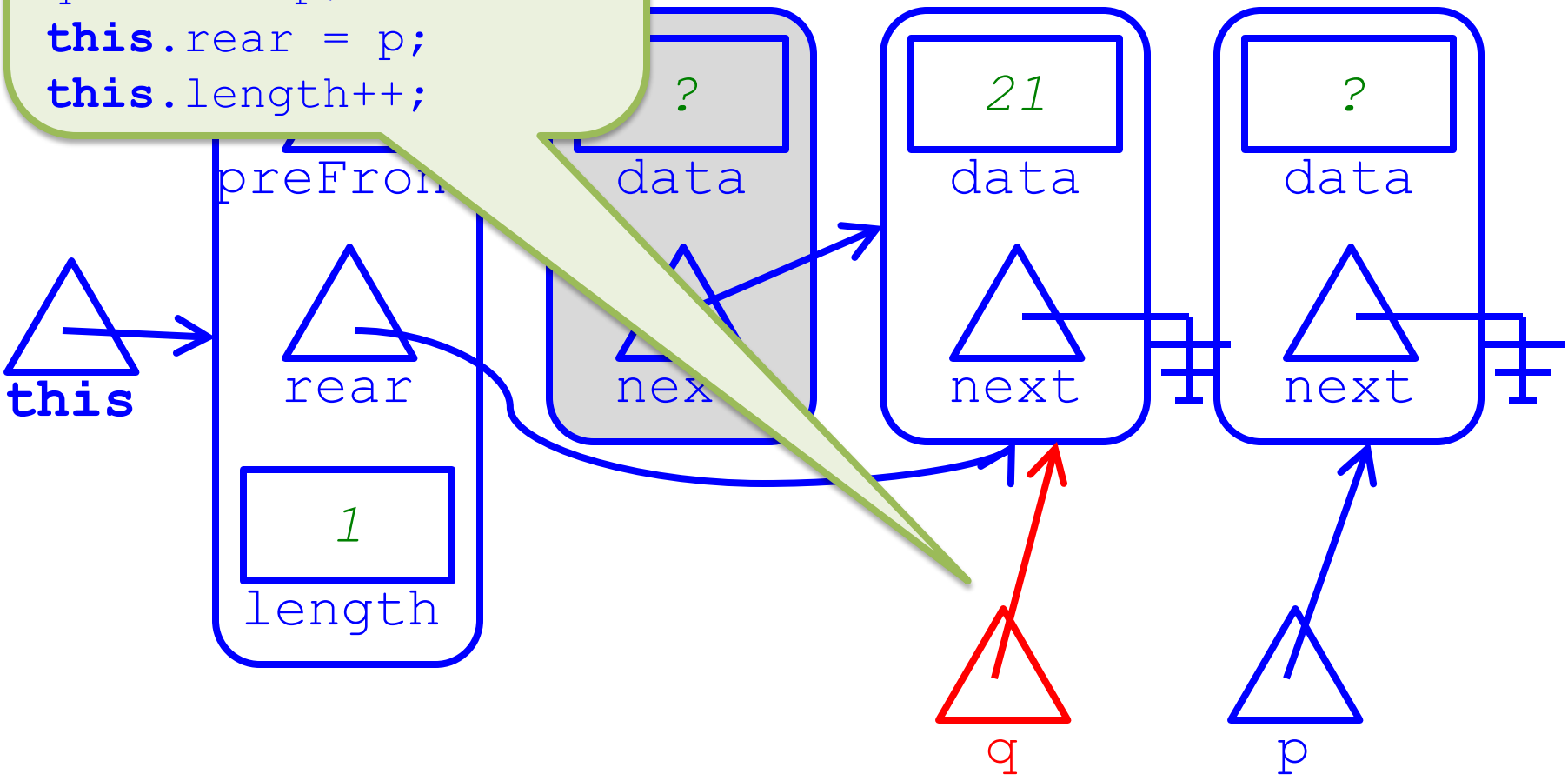
An instance variable of a reference type in Java is always initialized to **`null`**; but it is a good idea to be explicit (see later line of code).



queue for Queue2

= 74

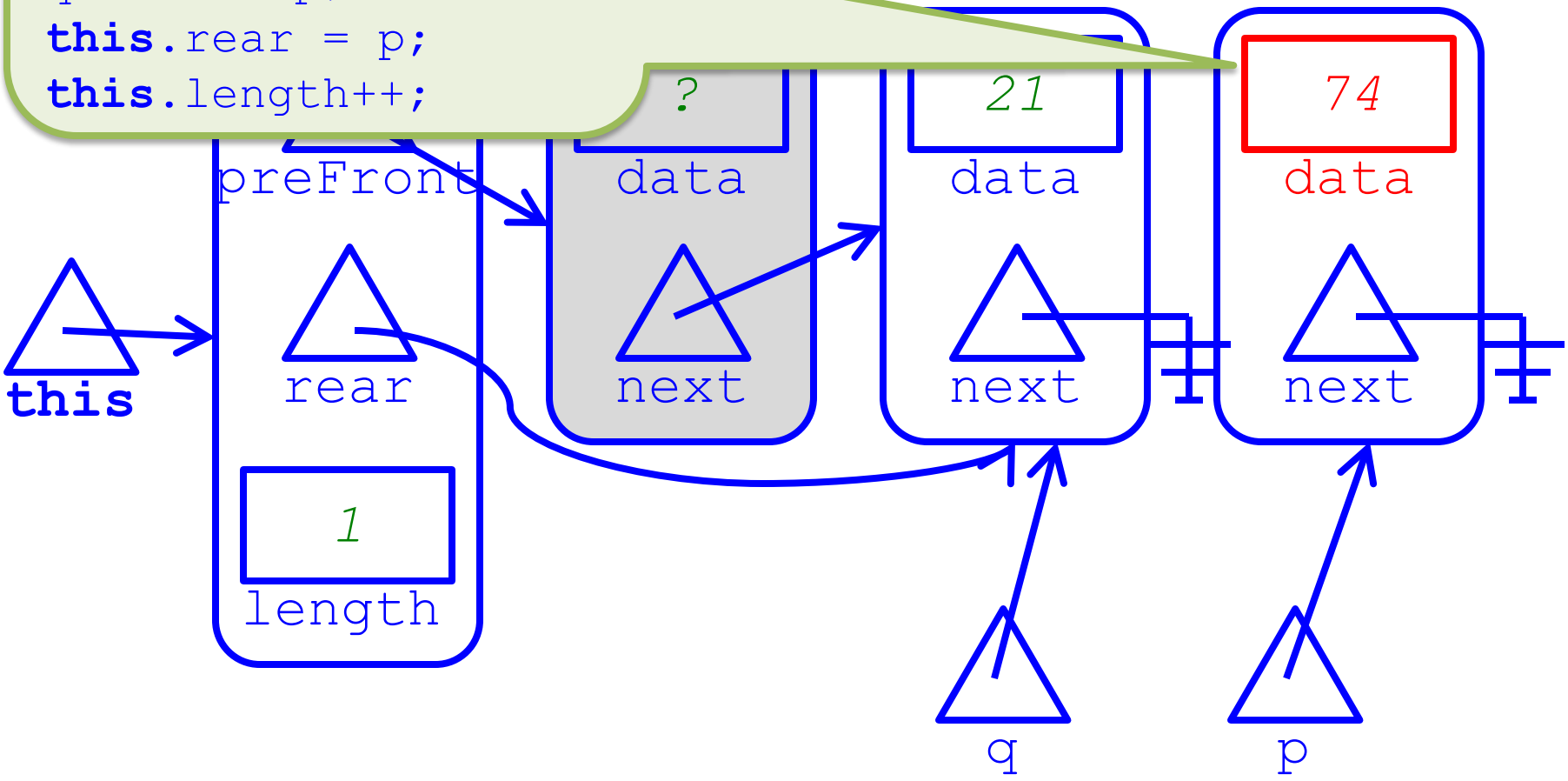
```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```



queue for Queue2

```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```

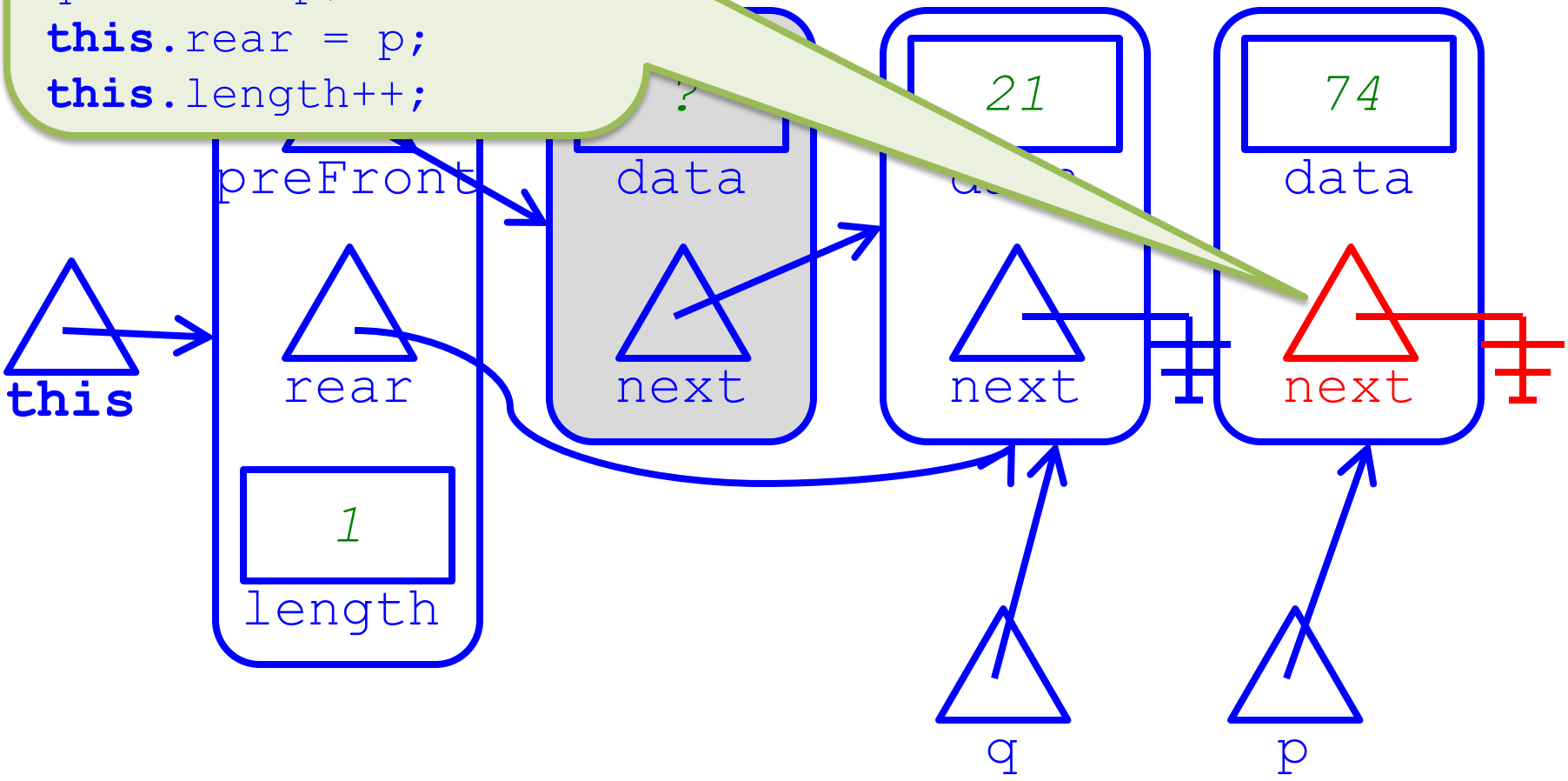
x = 74



queue for Queue2

```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```

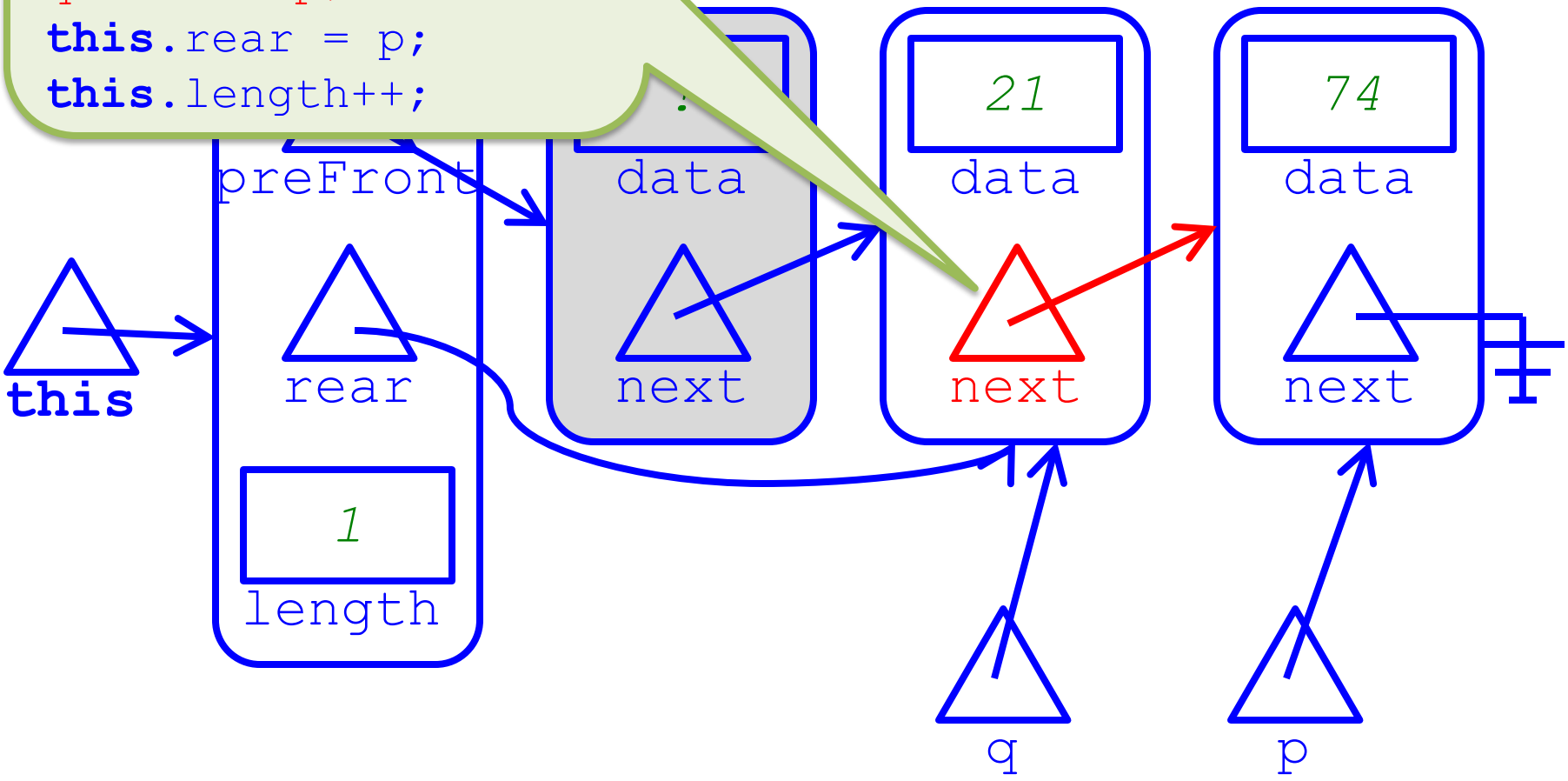
x = 74



queue for Queue2

```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```

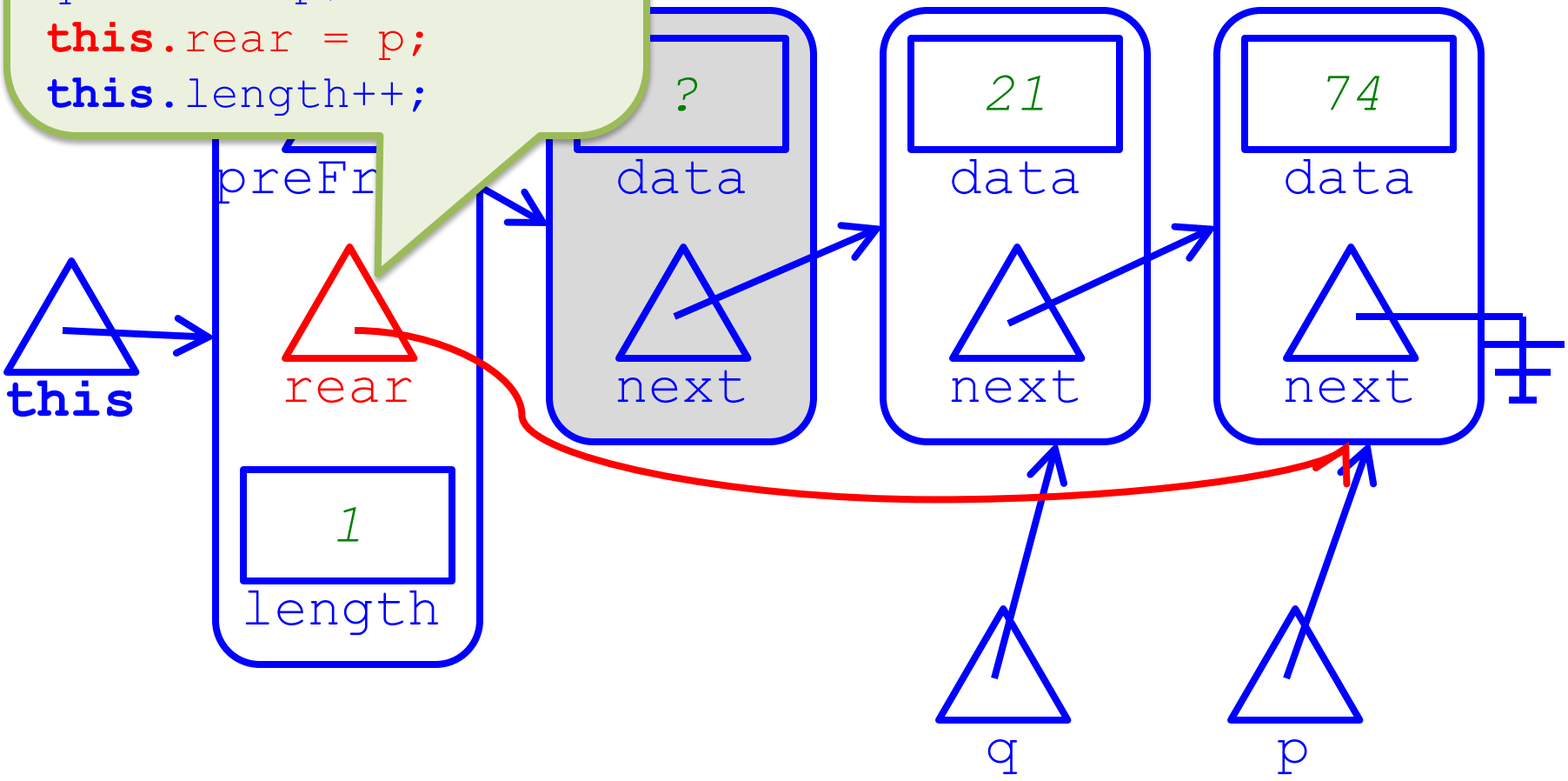
x = 74



queue for Queue2

= 74

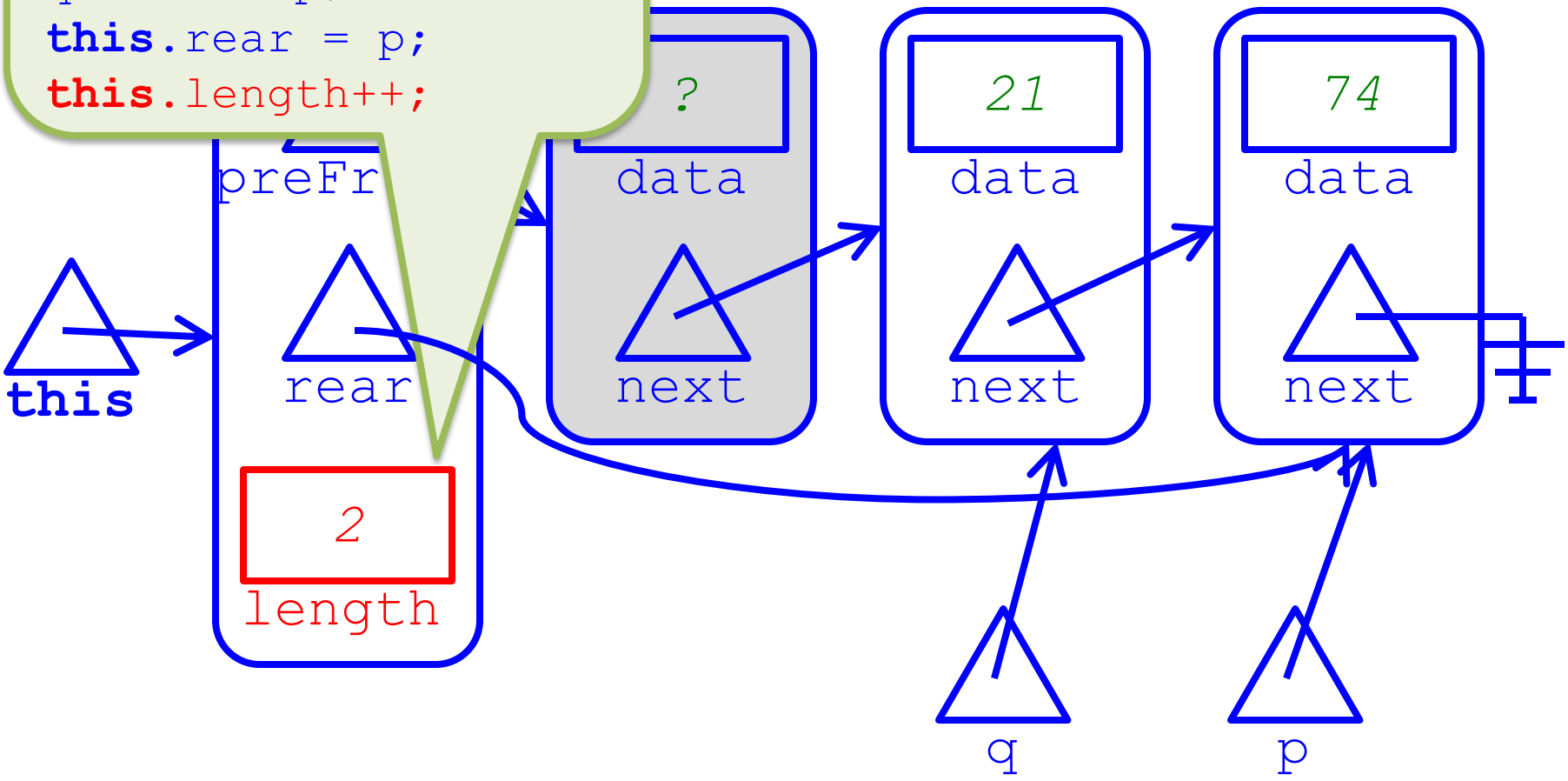
```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```



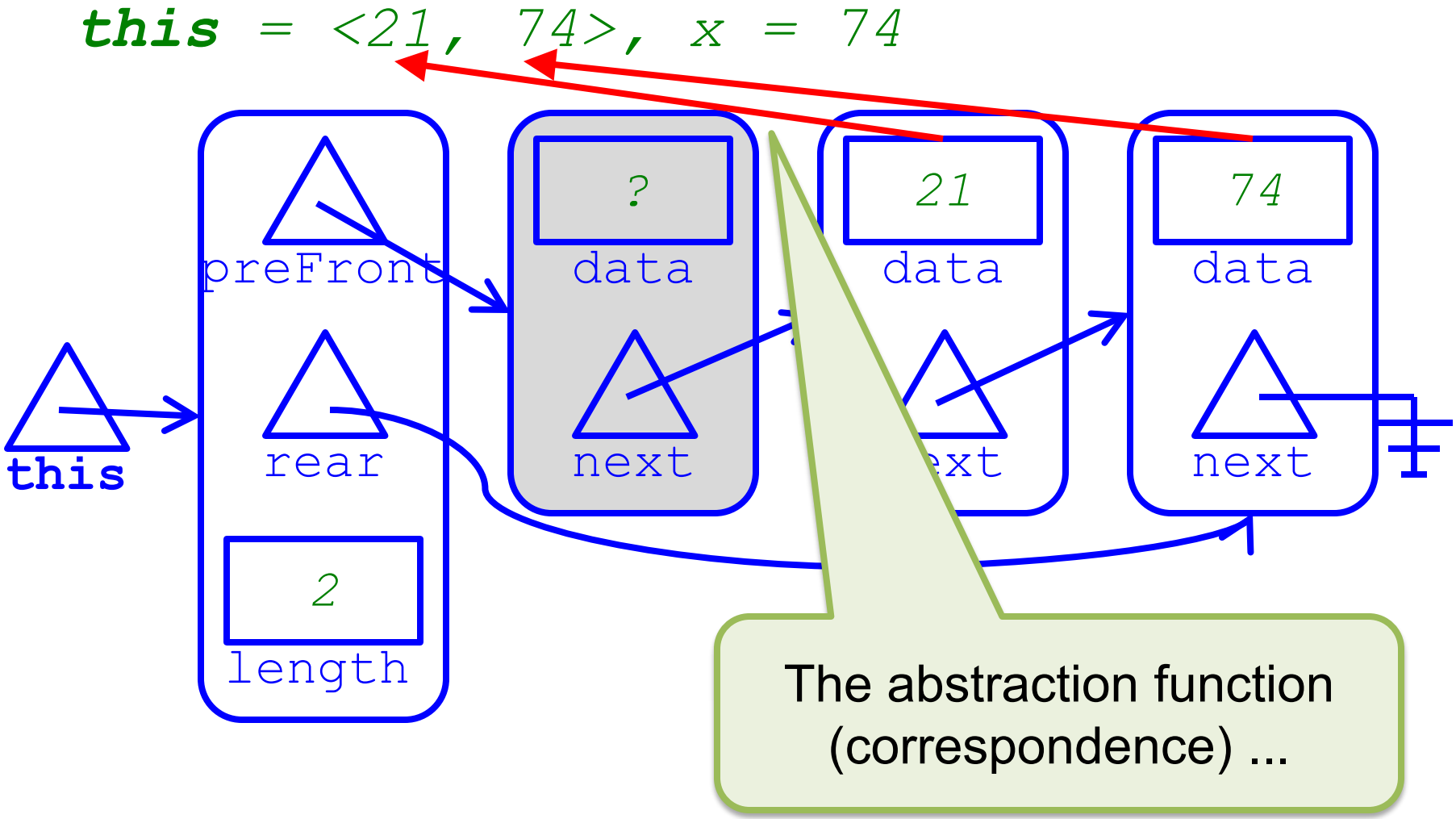
queue for Queue2

= 74

```
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
```



Example: `enqueue` for `Queue2`



The “Smart” Node

- To see why we want the extra node at the beginning of the linked list, write the code for `enqueue` without it (but be careful)
  - You should be able to see why it’s a ***smart node*** rather than a ***dummy node*** 😄
- Why is the smart node helpful in the representation of a `Queue`, but not of a `Stack`?

Resources

- Wikipedia: Linked Data Structure
  - http://en.wikipedia.org/wiki/Linked_data_structure
- *Big Java (4th ed)*, Sections 15.1, 15.2 (but not the part about iterators)
  - https://library.ohio-state.edu/record=b8540788~S7